CONTRIBUTION À L'OPTIMISATION DES RÉSEAUX ÉLECTRIQUES INTELLIGENTS PAR LE DÉVELOPPEMENT D'UN CADRICIEL POUR MÉTAHEURISTIQUES PARALLÈLES SUR PROCESSEURS GRAPHIQUES

CONTRIBUTION TO THE OPTIMIZATION OF SMART GRIDS BY THE DEVELOPMENT OF A SOFTWARE FRAMEWORK FOR PARALLEL METAHEURISTICS ON GRAPHICS PROCESSING UNITS

Une thèse soumise à la division des études supérieures
du Collège militaire royal du Canada
par

Vincent Rémi Roberge, CD, M.A.Sc., P.Eng.
Capitaine

en vue de l'obtention du diplôme de
Doctorat en philosophie

Janvier 2016

Résumé

Roberge, Vincent Rémi. Ph.D. Collège militaire royal du Canada, janvier 2016. *Contribution à l'optimisation des réseaux électriques intelligents par le développement d'un cadriciel pour métaheuristiques parallèles sur processeurs graphiques.* Thèse dirigée par professeur M. Tarbouchi, Ph.D. et professeur F. Okou, Ph.D.

Les progrès technologiques dans les domaines de l'information et des communications ont permis la modernisation du réseau électrique pour former le réseau électrique intelligent. Le déploiement d'un grand nombre de capteurs jumelé à un système de communication bidirectionnelle robuste permet la surveillance et le contrôle en temps réel de la production, du transport et de la distribution de l'électricité. Plusieurs des problèmes d'optimisation impliqués sont à grande échelle, non linéaires, non convexes et à variables mixtes. Ils sont souvent trop complexes pour être résolus efficacement par les méthodes d'optimisation classiques et nécessitent généralement l'emploi des métaheuristiques. Les métaheuristiques sont des algorithmes d'optimisation non déterministes qui se basent sur l'amélioration itérative d'une ou plusieurs solutions candidates pour obtenir une solution quasi optimale. Elles ont l'avantage d'être résilientes aux optima locaux, de pouvoir considérer les variables mixtes et d'être utilisables avec des fonctions de coût non différentiables. Par contre, à cause de leur fonctionnement itératif, elles nécessitent une puissance de calcul considérable qui limite leur utilisation dans des applications où le temps de réaction est critique tel que pour le contrôle en temps réel des réseaux électriques. Dans cette thèse, nous proposons de mitiger cette limitation par une implémentation parallèle des métaheuristiques sur processeurs graphiques (GPU). Équipés de quelques milliers de cœurs, les GPU représentent une technologie émergente qui a le potentiel de réduire le temps de calcul des métaheuristiques par une exécution parallèle et d'offrir une solution avantageuse pour le contrôle des réseaux électriques intelligents. Pour vérifier cette hypothèse, nous proposons dans cette thèse un cadriciel pour métaheuristiques parallèles sur GPU et utilisons l'outil développé pour résoudre trois problèmes d'optimisation d'intérêt relié au contrôle des réseaux électriques, soit la minimisation des harmoniques d'un onduleur multiniveau, l'optimisation de l'écoulement de puissance et la reconfiguration des réseaux de distribution. Ces problèmes ont été choisis puisqu'ils couvrent l'étendue des réseaux intelligents en touchant à la production, au transport et à la distribution de l'électricité. Dans les trois cas, le cadriciel proposé permet de calculer des solutions de meilleure qualité, à des problèmes de plus grande dimension, dans un temps plus court comparativement aux méthodes actuelles. En exploitant l'architecture massivement parallèle des GPU, nos algorithmes offrent des accélérations allant jusqu'à 534x comparées à une exécution séquentielle sur microprocesseur standard (CPU).

Mots clés : CUDA™, cadriciel, reconfiguration des réseaux de distribution, optimisation de l'écoulement de puissance, métaheuristique, minimisation des harmoniques, onduleur multiniveau, processeurs graphiques, programmation parallèle, réseau électrique intelligent

Abstract

Roberge, Vincent Rémi. Ph.D. Royal Military College of Canada, January 2016. *Contribution to the Optimization of Smart Grids by the Development of a Software Framework for Parallel Metaheuristics on Graphics Processing Units.* Supervised by Dr. M. Tarbouchi and Dr. F. Okou.

Advancements in the fields of information technology and communications have enabled the modernization of the power grid to form the smart grid. The deployment of a large number of sensors combined with a robust two-way communication system now allow for the real time monitoring and control of the generation, the transmission and the distribution of electricity. Several of the optimization problems involved in the control of smart grids are large-scale, non-linear, non-convex and have mixed variables. They are often too complex to be solved efficiently by traditional optimization methods and require the use of metaheuristics. Metaheuristics are non-deterministic optimization algorithms that are based on the iterative improvement of one or more candidate solution in order to obtain a near optimal solution. They have the advantage of being resilient to local optima, they can consider discrete variables and they can be used with non-differentiable cost functions. However, because of their iterative operation, they often require considerable computational power which limits their use in applications where the reaction time is critical as in the real-time control of smart grids. In this thesis, we propose to mitigate this limitation by developing parallel implementations of metaheuristics on graphics processing units (GPUs). Equipped with a few thousand cores, GPUs represent an emerging technology in the field of high performance computing that has the potential to reduce the computation time of metaheuristics through parallel execution and provide an advantageous solution for the control of smart grids. To verify this hypothesis, we propose a software framework for parallel metaheuristics on GPUs and use this framework to solve three optimization problems of great interest for the control of smart grids. The three problems are the harmonic minimization problem in multilevel inverters, the optimal power flow problem and the optimal distribution feeder reconfiguration problem. These problems were selected because they cover the entirety of smart grids by touching the generation, the transmission and the distribution of electricity. In all three cases, the proposed framework calculates better solutions, to problems of greater sizes, in shorter times compared to current methods. By leveraging the massively parallel architecture of the GPU, our algorithms provide accelerations of up to 534x compared to equivalent sequential algorithms executed on a microprocessor (CPU).

Keywords : CUDA™, distribution network reconfiguration, graphics processing unit, harmonics minimization, metaheuristic, multilevel inverter, optimal power flow, parallel programming, smart grid, software framework

Table des matières

Liste des tables

Liste des figures

Liste des acronymes

API	Interface de programmation d'applications (de l'anglais *Application program interface*)
BDDB	Matrice bloc-diagonale avec doubles bordures
B-F	Méthode régressive-progressive (de l'anglais *Backward-forward*)
BFS	Parcours en largeur (de l'anglais *Breadth first search*)
CC	Courant continu
COO	Format coordonnées pour matrices creuses
CPU	Processeur (de l'anglais *Central processing unit*)
CSR	Format rangées compressées pour matrices creuses (de l'anglais *Compressed sparse row*)
DFR	Reconfiguration optimale des réseaux de distribution (de l'anglais *Distribution feeder reconfiguration*)
DFS	Parcours en profondeur (de l'anglais Depth first search)
FACTS	Modules de transmission flexible (de l'anglais *Flexible alternating current transmission system*)
FFT	Transformation de Fourier rapide (de l'anglais *Fast Fourier transform*)
FPGA	Matrice prédiffusée programmable par l'utilisateur (de l'anglais *Field programmable gate array*)
G$	Milliards de dollars
GA	Algorithme génétique (de l'anglais *Genetic algorithm*)
GDDR5	Mémoire graphique à débit de données double, version 5 (de l'anglais *Graphics double data rate memory*)
G-N	Gauss-Newton
GPU	Processeur graphique (de l'anglais *Graphics processing unit*)
gpuMF	*GPU Metaheuristic Framework*
G-S	Gauss-Seidel
HPC	Calcul haute performance (de l'anglais *High performance computing*)

IEEE	Institute of Electrical and Electronics Engineers
M$	Millions de dollars
MICP	Programmation conique en nombres entiers (de l'anglais *Mixed integer conic programming*)
MILP	Programmation linéaire en nombres entier (de l'anglais *Mixed integer linear programming*)
MIQP	Programmation quadratique en nombres entiers (de l'anglais *Mixed integer quadratic programming*)
MOF	Cadriciel pour les métaheuristiques (de l'anglais *Metaheuristic optimization framework*)
Mt	Millions de tonnes
N-R	Newton-Raphson
OPF	Optimisation de l'écoulement de puissance (de l'anglais *Optimal power flow*)
PF	Écoulement de puissance (de l'anglais *Power flow*)
PI	Points intérieurs
PSO	Optimisation par essaim de particules (de l'anglais *Particle swarm optimization*)
SDP	Semi-définie positive
SIMD	Instruction unique, données multiples (de l'anglais *Single-instruction, multiple data*)
SIMT	Instruction unique, threads multiples (de l'anglais *Single-instruction, multiple threads*)
SMM	Multiprocesseur *Maxwell* (de l'anglais *Maxwell Streaming Multiprocessor*)
SVAR	Compensateur statique d'énergie réactive (de l'anglais *Static VAR compensator*)
THD	Taux de distortion harmonique (de l'anglais *Total harmonic distortion*)
TWh	Terawatt heure
UML	Langage de modélisation unifié (de l'anglais *Unified modeling language*)
XML	Langage à balise extensible (de l'anglais *Extensible markup language*)

Liste des symboles

Métaheuristiques

$\vec{x}, \vec{y}$	Vecteur de solution candidate
x_i, y_i	Élément i du vecteur de solution candidate $\vec{x}$ ou $\vec{y}$

Optimisation par essaim de particules

$\vec{b}_t$	Vecteur de la meilleure position occupée par la particule
$\vec{g}_t$	Vecteur de la meilleure position occupée par l'essaim
$\vec{r}_1, \vec{r}_2$	Vecteurs de nombres aléatoires entre 0 et 1
t	Numéro de l'itération
$\vec{v}_t$	Vecteur de vitesse d'une particule
$\vec{x}_t$	Vecteur de position d'une particule
ω, c_1, c_2	Paramètres d'inertie, d'influence personnelle et d'influence sociale

Algorihtme génétique

α	Paramètre d'enjambement mélangé

Processeur graphique

S	Accélération d'un programme parallèle calculée par T_{seq}/T_{par}
T_{seq}	Temps d'exécution de la version séquentielle du programme
T_{par}	Temps d'exécution de la version parallèle du programme

Onduleur multiniveau

f_{cost}	Fonction de coût
E_k	Tension de la source continue k
H	Fonction Heaviside
k	Indice de la source
m	Indice de modulation

M	Nombre de solutions candidates
N	Nombre d'harmoniques considérées
P	Fonction d'impulsion
s	Nombre de sources
$V(t)$	Tension de sortie en fonction du temps t
V_1^*	Amplitude spécifiée pour la composante fondamentale
V_1	Amplitude obtenue pour la composante fondamentale
V_n	Amplitude de la $n^{ième}$ harmonique
V_{dc_k}	Tension de la source continue k
$\vec{\theta}$	Vecteur des angles de commutation (vecteur de solution)
θ_k	Angle de commutation pour la source k
λ	Multiplicateur de Lagrange constant

Écoulement de puissance

a_i, b_i, c_i	Coefficients du coût d'opération du générateur i
b	Susceptance
f	Extrémité de départ d'une branche
$f(\bar{x}, \bar{u})$	Fonction objective
$f_{coûts}(\bar{x}, \bar{u})$	Coûts de production
$f_{émissions}(\bar{x}, \bar{u})$	Émissions polluantes
$f_{pertes}(\bar{x}, \bar{u})$	Pertes de transport
g	Conductance
$g(\bar{x}, \bar{u})$	Contraintes d'égalités
$h(\bar{x}, \bar{u})$	Contraintes d'inégalités
i_f, i_t	Courants injectés aux extrémités f et t d'une branche
$\boldsymbol{I}$	Vecteur des courants complexes injectés au bus d'un réseau
j	Unité imaginaire
$\boldsymbol{J}$	Matrice Jacobienne
$N:1$	Rapport de transformation d'un transformateur $N = \tau e^{j\theta_{shift}}$
n_{branch}	Nombre de branches dans le réseau
n_{bus}	Nombre de bus dans le réseau
n_{gen}	Nombre de générateurs dans le réseau
n_{PQ}	Nombre de bus PQ dans le réseau
n_{PQPV}	Nombre de bus PQ et PV dans le réseau
n_{PV}	Nombre de bus PV dans le réseau

n_{tr}	Nombre de transformateurs dans le réseau
NNZ	Nombre d'éléments non nuls dans une matrice creuse
P_i, Q_i	Puissance active et réactive injectées au bus i
P_{Gi}, Q_{Gi}	Puissance active et réactive générées au bus i
P_{Di}, Q_{Di}	Puissance active et réactive demandées au bus i
q_{ci}, Qc	Puissance réactive du compensateur statique d'énergie réactif i
r	Résistance
S_i	Puissance complexe injectée au bus i
t	Extrémité d'arrivée d'une branche
T	Rapport d'un transformateur
$\bar{u}$	État du réseau
$\boldsymbol{V}$	Vecteur des tensions complexes aux bus d'un réseau
v_f, v_t	Tensions complexes aux extrémités d'une branche
V_i	Tension complexe au bus i
$\lvert V_i \rvert, \delta_i$	Amplitude et angle de la tension au bus i
w_1, w_2 et w_3	Poids des termes de la fonction objective
x	Réactance
$\bar{x}$	Variable de contrôle du réseau (vecteur de solution)
y	Admittance $y = g + j\,b$
$\boldsymbol{Y}_{bus}$	Matrice d'admittance du réseau
$\boldsymbol{Y}_{ft}$	Matrice d'admittance de la branche ft
y_{ij}	Élément i, j de la matrice d'admittance du réseau
z	Impédance $z = r + j\,x$
$\alpha_i, \beta_i, \gamma_i, \zeta_i, \lambda_i$	Coefficients des émissions polluantes du générateur i
θ_{shift}	Déphasage d'un transformateur
τ	Rapport d'un transformateur

Reconfiguration des réseaux de distribution

E	Excès relatif
E_{total}	Excès relatif total pour le réseau en entier
$\mathcal{F}(\bar{x})$	Fonction d'aptitude
$f(\bar{x})$	Fonction objective
$f_{NORM}(\bar{x})$	Fonction objective normalisée
G	Graphe représentant la topologie du réseau
i, j, k	Indice d'un bus

$I_{br\ i,j}$	Courant complexe dans la branche reliant le bus i au bus j
$I_i, I_{bus\ i}$	Courant complexe injecté aux bus i
I_l	Courant complexe dans la branche l
l	Indice d'une branche
L	Pertes actives totales dans le réseau de distribution
$N_{branches}$	Nombre de branches dans le réseau
N_{bus}	Nombre de bus dans le réseau
R_l	Résistance de la branche l
S_i	Puissance complexe demandée ou injectée au bus i
S_l	Puissance complexe dans la branche l
t	Itération de l'algorithme régressif-progressif
V_i	Tension complexe au bus i
$\mathcal{V}_{NORM}$	Facteur de violation normalisé
$\bar{x}$	Configuration du réseau (vecteur de solution)
Y_i	Admittance de shunt au bus i
$Z_{br\ i,j}$	Impédance de la branche connectant le bus i au bus j

Chapitre 1
Introduction

1.1 Généralités

Le réseau électrique est l'un des systèmes d'ingénierie les plus complexes jamais construits par l'homme. Il représente une composante fondamentale de l'infrastructure des sociétés modernes. L'électrification de notre pays a changé notre mode de vie, révolutionné notre économie et initié une course sans répit aux progrès technologiques. C'est grâce à l'électricité que nous vivons dans le monde informatisé et connecté d'aujourd'hui. Depuis ses débuts, il y a un peu plus de 100 ans, le réseau électrique a vu une série d'améliorations, mais sa structure fondamentale n'a jamais changé. Celle-ci est organisée en trois composantes primaires : le système de production, le système de transport et le système de distribution. Tel qu'illustré à la Figure 1.1, ces trois systèmes sont organisés linéairement. L'énergie voyage essentiellement dans une seule direction. Elle est produite par les centrales électriques, transportée sur de longues distances vers les régions habitées par les lignes à haute tension et acheminée aux clients par les réseaux de distribution. Le contrôle du réseau électrique traditionnel se fait en grande partie au niveau de la production et est fortement basé sur l'expérience des opérateurs de façon à assurer que la quantité d'énergie produite soit suffisante pour supporter les pics de demandes.

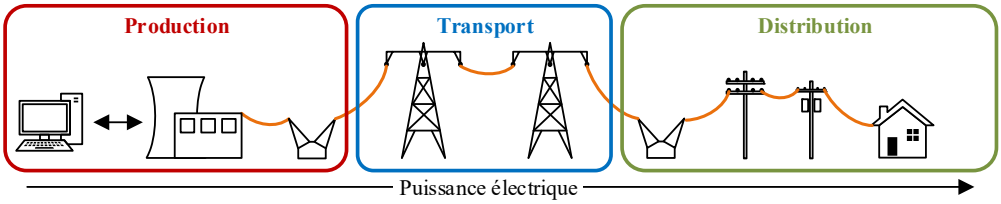


Figure 1.1: Organisation d'un réseau électrique traditionnel

En ce début du 21^{e} siècle, le réseau électrique traditionnel est confronté à plusieurs défis importants incluant la demande grandissante en énergie, le vieillissement de l'infrastructure, l'intégration des sources d'énergie renouvelable, l'augmentation du nombre de véhicules électriques, l'exigence d'une fiabilité accrue et la diminution des émissions polluantes [1]. Face à ces défis, le réseau actuel s'avère mal adapté. Il offre une mauvaise visibilité sur l'état du système, une capacité limitée d'analyse automatique et un temps de réponse trop long à cause des interrupteurs

mécaniques [2]. Il est de nos jours évident qu'une modernisation majeure du réseau électrique est nécessaire.

Ce sont les avancées technologiques dans le domaine de l'informatique et des communications qui ont permis depuis une dizaine d'années la modernisation du réseau électrique pour former le réseau électrique intelligent. Le réseau intelligent est souvent perçu comme étant une révolution visant à remplacer le système actuel. En fait, il s'agit plutôt d'une évolution où l'infrastructure existante est gardée, mais améliorée par l'ajout de capteurs variés, d'un réseau de communication moderne, d'un système informatique de gestion et d'équipements de contrôle sophistiqués tel que des interrupteurs automatiques ou des modules de transmission flexibles (FACTS de l'anglais *Flexible Alternating Current Transmission System*). Ces technologies permettent une surveillance en temps réel et un contrôle rapide du système. L'organisation du réseau électrique intelligent est illustrée à la Figure 1.2. Il est toujours divisé en trois segments, mais contrairement au réseau traditionnel, le réseau intelligent permet l'intégration de sources d'énergie alternatives à tous les niveaux du système. La puissance peut s'écouler dans les deux directions permettant au client de consommer ou de redonner de l'énergie au réseau. Tels qu'identifiés par l'Agence internationale de l'énergie [1], les avantages du réseau électrique intelligent incluent :

- la participation active et informée des clients;
- l'intégration de sources et de banques d'énergie distribuées incluant les énergies vertes;
- un nouveau marché économique offrant des opportunités aux producteurs, aux opérateurs et aux consommateurs;
- une qualité de puissance électrique accrue;
- une meilleure utilisation de l'infrastructure existante;
- une amélioration de l'efficacité au niveau de la production, du transport et de la distribution;
- une résistance supérieure aux fautes matérielles, aux attaques malicieuses et aux catastrophes naturelles; et
- une diminution des émissions polluantes.

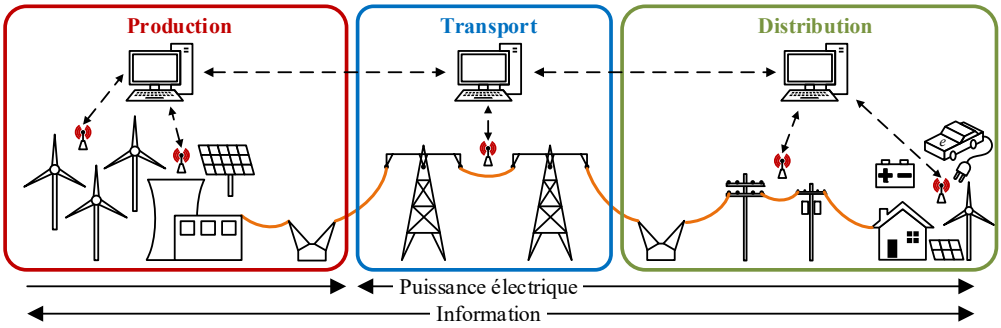


Figure 1.2: Organisation d'un réseau électrique intelligent

L'implémentation du réseau électrique intelligent demande un investissement matériel important, mais aussi le développement de programmes logiciels capables d'analyser l'état du réseau et de calculer les signaux de contrôle qui assurent l'opération optimale du système. Plusieurs des problèmes de contrôle impliqués s'expriment sous forme de problèmes d'optimisation. Sans donner une liste exhaustive, des exemples de problèmes d'optimisation pour le contrôle des réseaux intelligents sont, au niveau du déploiement du système, le calcul des locations optimales pour l'installation des capteurs [3], des générateurs distribués [4], des panneaux solaires [5], des éoliennes [6] ou des unités de stockage d'énergie [7]. Au niveau de la communication et de la gestion des données, il y a les problèmes de réduction des données [8], de la priorisation des messages [9] et du routage des messages [10]. Au niveau de l'opération, il y a l'optimisation de l'orientation des panneaux solaires [11] et de l'inclinaison des pales des éoliennes [12], le contrôle des onduleurs [13], la prédiction de la production d'énergie renouvelable [14], la minimisation des émissions polluantes [15], l'optimisation de l'écoulement de puissance (OPF de l'anglais *optimal power flow*) [16], la reconfiguration optimale des réseaux de distribution (DFR de l'anglais *distribution feeder reconfiguration*) [17] et la gestion optimale des véhicules électriques [18]. Au niveau économique, il y a l'ajustement optimal du prix de l'électricité [19] et le contrôle de la demande par le consommateur [20]. Finalement, des exemples au niveau de la sécurité du réseau sont le calcul de la topologie optimale avec contrainte $N-1$ [21], la reconnaissance et la classification des fautes [22] et l'auto-rétablissement du système suite à une perturbation [23]. Plusieurs de ces problèmes sont non linéaires, non convexes et dans certains cas, à variables mixtes. Ces problèmes sont par conséquent difficilement résolubles par les méthodes d'optimisation classiques et nécessitent l'emploi des métaheuristiques.

Les métaheuristiques sont des algorithmes d'optimisation non déterministes qui se basent sur l'amélioration itérative d'une ou plusieurs solutions candidates. Elles ne garantissent pas l'optimalité de la solution finale, mais permettent de calculer de très

bonnes solutions dans un temps acceptable pour des problèmes difficilement traitables par les méthodes classiques. Les métaheuristiques ont plusieurs avantages. Elles peuvent échapper aux optima locaux des problèmes non convexes, elles sont utilisables avec des fonctions objectives non différentiables et elles considèrent les variables de contrôle discrètes. Par contre, à cause de leur fonctionnement itératif, elles nécessitent une puissance de calcul considérable ce qui limite leur utilisation dans des applications où le temps de réaction est critique tel que les problèmes d'optimisation identifiés au paragraphe précédent au niveau de l'opération des réseaux électriques intelligents.

Comme le suggèrent les auteurs de [24], une solution possible au contrôle rapide des réseaux électriques vient des domaines de la programmation parallèle et du calcul haute performance (HPC de l'anglais *high performance computing*). En effet, des implémentations de métaheuristiques parallèles ont déjà été proposées pour les processeurs (CPU de l'anglais *central processing unit*) multicœurs [25], les réseaux d'ordinateurs [26] ou même les superordinateurs [27]. Généralement, ce sont les superordinateurs qui offrent la plus grande puissance de calcul avec leurs milliers de processeurs et leur réseau d'interconnexion à très haut débit. Ces systèmes informatiques sont cependant très dispendieux et restent peu répandus dans le secteur industriel. Avec l'arrivée du langage NVIDIA® CUDA™ en 2007, les processeurs graphiques (GPU de l'anglais *graphics processing unit*) font émergence dans le domaine du HPC. Ce langage permet la programmation des GPU pour le calcul scientifique. Comme on le voit à la Figure 1.3, comparé à un CPU multicœur, le GPU est équipé de plusieurs centaines voire quelques milliers de cœurs et offre au programmeur un système massivement parallèle sur une seule puce. Abordables et présents dans la plupart des ordinateurs, les GPU représentent une solution innovatrice pour l'accélération du contrôle des réseaux électriques intelligents.

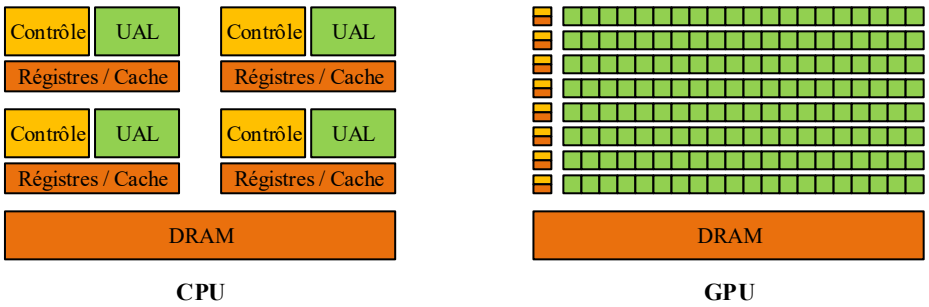


Figure 1.3: Architecture typique d'un CPU multicœur et d'un GPU

1.2 Énoncé de la thèse

Afin d'étudier le potentiel des GPU quant au contrôle des réseaux électriques intelligents, nous formulons l'hypothèse de recherche suivante :

> « *Le développement de métaheuristiques parallèles sur GPU contribue à l'amélioration du contrôle des réseaux électriques intelligents en permettant de calculer des solutions de meilleure qualité à des problèmes d'optimisation de plus grande dimension tout en réduisant les temps de calcul.* »

1.3 Objectifs de recherche

Pour vérifier cette hypothèse, nous avons jugé nécessaire d'accomplir les six objectifs de recherche suivants :

1. Développer un cadriciel pour les métaheuristiques parallèles sur GPU afin de créer un outil extensible, facile à utiliser et adaptable à une large gamme de problèmes d'optimisation.

2. Implémenter l'algorithme d'optimisation par essaim de particules (PSO de l'anglais *particle swarm optimisation*) et le paralléliser sur GPU à l'aide du cadriciel développé. Premièrement, ceci permettra de valider la conception du cadriciel. Deuxièmement, ceci permettra de mesurer le gain de performance apporté par la parallélisation sur GPU. Troisièmement, l'implémentation du PSO nous donnera un outil d'optimisation que l'on pourra appliquer au contrôle des réseaux électriques intelligents.

3. Implémenter l'algorithme génétique (GA de l'anglais *genetic algorithm*) et le paralléliser sur GPU à l'aide du cadriciel développé. Tout comme pour l'objectif précédent, ceci permettra de valider le cadriciel développé, de mesurer l'avantage du GPU et d'optimiser des solutions pour le contrôle des réseaux électriques intelligents.

4. Utiliser le cadriciel développé pour la minimisation des harmoniques d'un onduleur multiniveau. Dans le cadre du réseau électrique intelligent, les onduleurs multiniveaux sont entre autres utilisés pour intégrer des panneaux solaires au réseau électrique. Le défi est de calculer les angles de commutation des différentes sources de tension continue afin de produire la tension alternative désirée tout en minimisant les harmoniques qui entraînent des pertes électriques importantes sous forme de surchauffe au niveau des lignes de transport, des transformateurs et des moteurs électriques. Il s'agit d'un problème d'optimisation non linéaire, non convexe et trop complexe pour qu'il soit résolu

efficacement par des méthodes d'optimisation classiques. Les métaheuristiques ont été utilisées avec succès, mais leur temps d'exécution reste trop long pour réagir rapidement aux variations des tensions d'entrées. Une implémentation parallèle sur GPU semble pouvoir mitiger cette lacune.

5. Utiliser le cadriciel développé pour l'optimisation de l'écoulement de puissance. Ce problème consiste à trouver les réglages optimaux des générateurs et de l'équipement de transport de façon à maximiser une fonction objective pour le fonctionnement en régime permanent du réseau de transport d'électricité. Il s'agit d'un problème d'optimisation à grande échelle, non linéaire, non convexe et à variables mixtes. Pour résoudre ce problème, les méthodes d'optimisation classiques sont limitées à une optimisation locale, nécessitent une formulation simplifiée du problème et ne peuvent considérer efficacement les variables de contrôle discrètes telles que le rapport des transformateurs ou le réglage des compensateurs statiques. De leur côté, les métaheuristiques ont l'avantage de permettre une optimisation globale et de considérer nativement les variables discrètes. De plus, elles offrent une grande flexibilité au niveau de la définition de l'objectif d'optimisation. Toutefois, tout comme pour le problème précédent, leur temps d'exécution est trop long pour assurer un contrôle réactif d'où vient notre motivation pour une implémentation parallèle sur GPU.

6. Finalement, le dernier objectif consiste à utiliser le cadriciel développé pour la reconfiguration optimale des réseaux de distribution. Les réseaux de distribution représentent le dernier stage de l'acheminement de l'électricité des centrales de production aux consommateurs. Ils sont généralement structurés suivant une topologie radiale, mais incluent également des commutateurs de liaison supplémentaires pour permettre la reconfiguration en cas d'entretien planifié ou de pannes inattendues. L'implémentation du réseau électrique intelligent permet d'automatiser cette reconfiguration afin de réagir plus rapidement aux perturbations, mais aussi aux fluctuations de la demande en énergie. Le problème de la reconfiguration des réseaux de distribution consiste à calculer la configuration radiale du réseau qui permet de minimiser les pertes de distribution. Il s'agit d'un problème d'optimisation combinatoire, non linéaire et non convexe. L'utilisation des métaheuristiques est possible, mais leur temps d'exécution trop long est encore ici un désavantage important. Une parallélisation sur GPU pourrait accélérer le calcul de la topologie optimale.

Le premier objectif consiste à développer le cadriciel qui définit la structure de l'outil d'optimisation. Ce cadriciel doit tenir compte des caractéristiques communes aux métaheuristiques et de l'architecture parallèle des GPU. Le cadriciel développé doit faciliter l'implémentation et la parallélisation des métaheuristiques et permettre leurs applications à différents problèmes d'optimisation. Les deuxième et troisième

objectifs consistent à programmer deux métaheuristiques parallèles sur GPU à l'aide du cadriciel développé. Ces métaheuristiques sont ensuite utilisées aux quatrième, cinquième et sixième objectifs pour résoudre trois problèmes d'optimisation concrets dans le domaine des réseaux électriques intelligents. Ces problèmes couvrent l'étendue des réseaux électriques en considérant la production, le transport et la distribution de l'électricité. Ils sont tous les trois très complexes, non linéaires et non convexes ce qui nous force à utiliser les métaheuristiques pour les résoudre. Étant donné que ces trois problèmes concernent l'opération du réseau électrique, le temps d'exécution du logiciel développé est critique ce qui nous motive à exploiter l'architecture massivement parallèle des GPU pour accélérer le calcul des signaux de contrôle. Pour chacune des trois applications considérées, l'objectif est d'exploiter la parallélisation sur GPU afin de calculer des solutions de meilleure qualité, à des problèmes de plus grande dimension tout en réduisant le temps de calcul. Pour évaluer la qualité des solutions calculées et les dimensions des problèmes considérés, nous comparerons nos résultats à ceux publiés dans la littérature scientifique. Pour évaluer le gain de performance apportée par la parallélisation sur GPU, nous comparerons les temps d'exécutions obtenus à ceux de programmes séquentiels et parallèles équivalents sur CPU multicœurs que nous développerons aussi avec le cadriciel proposé. En confirmant que la parallélisation de métaheuristiques sur GPU améliore ces trois objectifs pour chacune des applications considérées, nous démontrerons la validité de l'hypothèse que nous avons formulée.

1.4 Motivation

L'optimisation des réseaux électriques intelligents par l'implémentation de métaheuristiques parallèles sur GPU est un sujet de recherche d'importance pour trois raisons principales. Premièrement, le réseau électrique intelligent est un phénomène récent et en pleine expansion qui soulève une multitude de défis d'ingénierie, entre autres au niveau du contrôle et de l'optimisation. Deuxièmement, étant donné l'envergure du système électrique, les bénéfices d'un progrès technologique, aussi petit qu'il soit, sont simplement énormes. Troisièmement, plusieurs des problèmes d'optimisation impliqués ne peuvent être résolus efficacement par des ordinateurs séquentiels. Les GPU sont une technologie émergente du domaine du HPC qui pourrait permettre d'attaquer la complexité de ces problèmes grâce à leur architecture massivement parallèle.

1.4.1 Un sujet actuel

Le réseau électrique intelligent est une technologie récente dont l'implémentation vient tout juste de commencer et continuera sur plusieurs années. D'après un rapport publié par le département des Ressources naturelles du Canada en 2013 [28], plus de

4300 véhicules électriques sont déjà sur les routes canadiennes, presque 1100 bornes de recharge sont en fonction et permettraient d'accueillir un nombre plus grand de véhicules électriques. Plus de 49% des compteurs ont été remplacés par des compteurs intelligents. Des programmes de facturation variable, du contrôle de la demande par le consommateur et de l'auto-rétablissement du système suite à une perturbation ont été initiés dans plusieurs provinces. Des prototypes de micro réseaux intelligents avec production et stockage d'énergie local ont été déployés sur la côte ouest et dans les territoires. Plusieurs autres projets et actes du gouvernement ont été annoncés par les provinces pour continuer l'implémentation du réseau intelligent. En 2012, l'Association canadienne de l'électricité a mis sur pied un programme de normalisation visant à établir les normes canadiennes pour faciliter la coopération des efforts provinciaux. Ces efforts incluent évidemment un investissement matériel important, mais aussi intellectuel afin de développer les technologies du futur. Sans compter les investissements nécessaires au remplacement ou à l'expansion de l'infrastructure actuelle, les investissements au niveau de la recherche et du développement des réseaux électriques sont passés de 380 M$ en 2011 à 900M$ en 2012 et 1050M$ en 2013. Finalement, le secteur du réseau intelligent a été identifié par le département des Affaires étrangères, du commerce et du développement du Canada comme étant un domaine d'exportations potentielles important [28]. Les travaux de recherche effectués dans le contexte de cette thèse visent à supporter cet effort national pour l'avancement des technologies des réseaux intelligents.

1.4.2 Des bénéfices énormes

Le réseau électrique est simplement un système énorme. D'après les statistiques publiées par l'Association canadienne de l'électricité [29], le Canada a produit en 2013 plus de 611.3 TWh d'énergie électrique dont 511.0 TWh ont été consommés par les Canadiens, 62.6 TWh ont été exportés aux États-Unis et 37.7 TWh ont été perdus au niveau du transport et de la distribution. Les ventes de cette énergie ont généré un revenu brut de 31.6 G$. Étant donné la grandeur de ces chiffres, tout progrès technologique qui permettait d'amélioration l'efficacité du système engendrerait des bénéfices énormes. À titre d'exemple, une toute petite amélioration de 1% au niveau de l'efficacité globale du système apporterait des économies de 310 M$. Ces gains sont encore plus énormes si l'on regarde les États-Unis qui ont eu une production de 4066 TWh en 2013 générant un revenu brut de 375.1 G$ [30]. Du point de vue environnemental, malgré que 63% de la production canadienne proviennent de centrales hydroélectriques et que 9% proviennent des éoliennes, le Canada a généré plus de 88 Mt d'émissions polluantes. Encore ici, une amélioration de 1% permettrait d'éliminer 80 000 tonnes de gaz polluants. À eux seuls, ces chiffres justifient les efforts de recherche sur l'optimisation des réseaux électriques intelligents.

1.4.3 Une technologie émergente

Plusieurs des problèmes d'optimisation liés au contrôle des réseaux électriques intelligents sont trop complexes pour être résolus efficacement par un programme séquentiel sur CPU. Les techniques actuelles utilisent des approximations afin de réduire la complexité des problèmes et de les rendre traitables par les méthodes d'optimisation classiques. Les solutions calculées sont optimales d'après la formulation simplifiée utilisée, mais pas d'après le problème original. Pour éviter ces approximations, il est possible de recourir aux métaheuristiques, mais celles-ci engendrent souvent des temps d'exécution trop longs ce qui limite leur utilisation pour l'optimisation en ligne telle que pour le contrôle des réseaux intelligents. Depuis l'arrivée des GPU pour le calcul scientifique en 2007, cette limitation n'est plus nécessairement vraie. Conçu avec un très grand nombre de cœurs, le GPU est un coprocesseur parallèle qui offre une puissance de calcul bien supérieure à celle d'un CPU multicœur. Des problèmes trop complexes pour être résolus sur un CPU peuvent maintenant être considérés grâce aux GPU. Le GPU est une technologie émergente du domaine du HPC qui est de plus en plus utilisé par les grands centres de calculs. À titre d'exemple, nous illustrons à la Figure 1.4 le nombre de superordinateurs classés au Top 500 qui utilisent des coprocesseurs tel que les GPU NVIDIA® ou ATI®. Les autres coprocesseurs identifiés sur cette figure ont une architecture semblable au GPU en ce qui concerne l'intégration d'un très grand nombre de cœurs et l'accès à une mémoire uniforme, mais sont spécialement conçus pour le calcul scientifique. À cause de leur usage plus précis, ces coprocesseurs sont plus dispendieux et moins répandus que les GPU. Abordables et présents dans la plupart des ordinateurs, les GPU sont une technologie nouvelle qui a le potentiel d'accélérer l'optimisation des réseaux électriques intelligents pour un contrôle plus réactif et efficace.

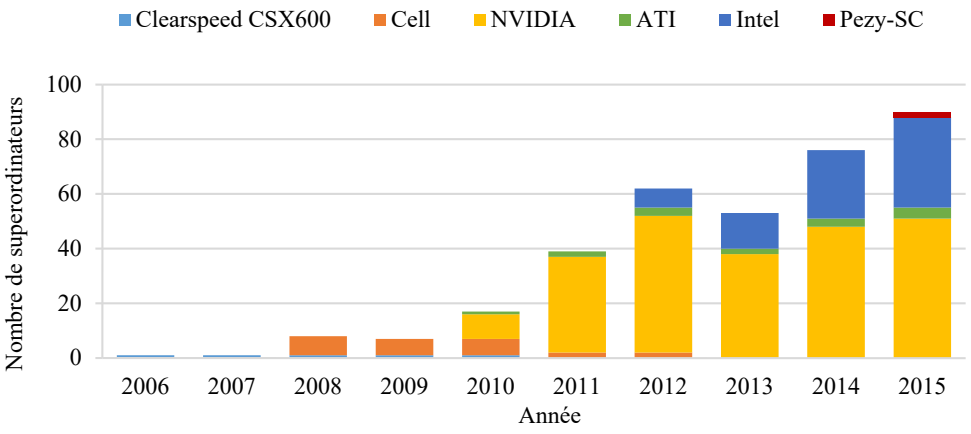


Figure 1.4: Superordinateurs classés au Top 500 qui utilisent des coprocesseurs (source [31])

1.5 Contributions scientifiques

La contribution principale de cette thèse est évidemment de démontrer la validité de l'hypothèse posée quant à savoir si la parallélisation de métaheuristiques sur GPU améliore l'optimisation des réseaux électriques intelligents en permettant de calculer des solutions de meilleure qualité, à des problèmes de plus grandes tailles tout en diminuant les temps de calcul comparé à une exécution séquentielle sur CPU. Pour vérifier cette hypothèse, nous avons identifié plusieurs objectifs de recherches. Comme nous l'expliquons dans la liste qui suit, chacun de ces objectifs représente aussi une contribution scientifique significative.

1. Le développement d'un cadriciel pour métaheuristiques parallèles sur GPU est une contribution importante. Tel que nous le verrons au Chapitre 2, les cadriciels existants sont difficilement adaptables à l'architecture des GPU et permettent uniquement de paralléliser l'évaluation de la fonction de coût. Le reste de l'algorithme est exécuté séquentiellement sur le CPU ce qui limite l'accélération permise. Dans cette thèse, nous proposons un nouveau cadriciel appelé *GPU Metaheuristic Framework* ou simplement *gpuMF*. Ce cadriciel initialise, garde et efface les solutions candidates directement dans la mémoire du GPU, ce qui permet d'exécuter la métaheuristique en entier sur le GPU et d'éviter les transferts de données entre le GPU et le CPU qui affectent la performance.

2. Les implémentations parallèles sur GPU des algorithmes de PSO et de GA sont aussi deux contributions importantes. Nos implémentations utilisent des primitives parallèles standards et suivent les pratiques de programmation recommandées au Chapitre 4 afin d'être le mieux possible adaptées à l'architecture des GPU. Nos implémentations maximisent les accès coalescents à la mémoire et minimisent la divergence d'exécution des threads. Les accélérations obtenues comparativement à une exécution séquentielle sur CPU sont de 304.4x pour le PSO et de 249.3x pour le GA, ce qui est bien supérieur aux implémentations antérieures.

3. L'application du cadriciel *gpuMF* pour la minimisation des harmoniques d'un onduleur multiniveau est aussi une contribution importante. Il s'agit en fait de la première fois que le GPU est utilisé pour résoudre ce problème. Comparé aux méthodes antérieures, notre algorithme parallèle permet d'optimiser des onduleurs avec un nombre beaucoup plus grand de sources afin de minimiser davantage le taux de distorsion harmonique tout en réduisant le temps d'exécution. Les temps de calcul mesurés varient de 38.7 ms à 117.5 ms dépendamment du nombre de sources de l'onduleur et l'accélération maximale apportée par la parallélisation est de 453.7x comparée à une implémentation séquentielle sur CPU.

4. En plus de la métaheuristique parallèle, nous proposons dans cette thèse une seconde méthode sur GPU pour la minimisation des harmoniques. Cette méthode est basée sur la formulation analytique présentée dans [32] et permet de calculer directement les angles de commutations de l'onduleur. Le temps de calcul est extrêmement court au détriment d'une moins bonne flexibilité quant à l'indice de modulation et au choix des harmoniques à minimiser. Notre implémentation sur GPU permet d'optimiser un onduleur à 1000 sources en seulement 16.41 µs, soit 534x plus rapide que pour une implémentation séquentielle sur CPU.

5. Au niveau de l'optimisation de l'écoulement de puissance, nous effectuons trois contributions importantes. Premièrement, il s'agit de la première fois qu'une métaheuristique parallèle sur GPU est proposée pour résoudre ce problème. Cette parallélisation est en effet très avantageuse et résulte en une accélération de 17.2x. Deuxièmement, la stratégie d'optimisation proposée est aussi une contribution importante puisqu'elle permet d'obtenir des solutions de meilleure qualité que les méthodes antérieures tout en respectant les contraintes telles que la puissance réactive aux générateurs. Troisièmement, nos implémentations parallèles des algorithmes de Gauss-Seidel (G-S) et de Newton-Raphson (N-R) pour l'analyse de l'écoulement de puissance lors de l'évaluation des solutions candidates représentent une troisième contribution majeure. Contrairement aux tentatives antérieures, nos implémentations utilisent des matrices creuses et parallélisent toutes les étapes des algorithmes offrant des accélérations maximales de 45.2x pour G-S et de 17.8x pour N-R.

6. Finalement, l'utilisation du cadriciel *gpuMF* pour la reconfiguration optimale des réseaux de distribution représente deux contributions majeures. D'abord, il s'agit de la première solution parallèle sur GPU proposée pour ce problème. Cette parallélisation n'est pas triviale puisqu'elle nécessite le calcul d'un arbre couvrant de poids minimal, le parcours d'un graphe et une analyse d'écoulement de puissance. La parallélisation est toutefois très avantageuse et permet une accélération de 66.2x. En second lieu, l'encodage proposé pour représenter les solutions candidates est unique et complètement différent des techniques antérieures. Cet encodage garantit la topologie radiale du réseau et permet à la métaheuristique d'optimiser des réseaux cinq fois plus grands que les méthodes antérieures référencées.

1.6 Organisation de la thèse

Cette thèse est organisée comme suit. Au Chapitre 2, nous présentons une revue de la littérature afin d'identifier les lacunes des méthodes actuelles quant à l'optimisation des réseaux intelligents et de mieux comprendre l'importance des contributions que nous apportons. Nous discutons ensuite des métaheuristiques et des

processeurs graphiques aux Chapitres 3 et 4. Cette théorie préliminaire aide à la compréhension des chapitres suivants. Le Chapitre 5 présente le cadriciel *gpuMF* que nous proposons pour l'implémentation de métaheuristiques parallèles sur GPU. Ce chapitre explique l'architecture de l'outil logiciel et donne trois cas d'utilisation qui démontrent son fonctionnement. Il traite aussi de la parallélisation des algorithmes du PSO et du GA. Le cadriciel est testé à l'aide de six fonctions tests afin de valider son efficacité et de mesurer le gain de performance apporté par la parallélisation sur GPU. Les Chapitres 6, 7 et 8 sont le corps de cette thèse. Ils expliquent comment le cadriciel développé est utilisé pour résoudre trois problèmes d'optimisation complexes quant au contrôle des réseaux électriques intelligents. Ces problèmes couvrent l'étendue du système en considérant la production, le transport et la distribution de l'électricité. Finalement, au Chapitre 9, nous concluons en résumant le contenu de ce document, en réitérant les contributions scientifiques apportées, en discutant la validité de l'hypothèse posée et en proposant des opportunités de recherches futures dans le domaine.

Chapitre 2
Revue de la littérature

Comme nous l'avons expliqué au chapitre précédent, les progrès technologiques dans le domaine de l'informatique et des communications ont permis l'informatisation du réseau électrique pour former le réseau électrique intelligent. Le déploiement d'un grand nombre de capteurs jumelé à un système de communication bidirectionnelle robuste permet de surveiller en temps réel l'état du réseau et de mieux le contrôler afin d'améliorer la production, le transport et la distribution de l'électricité. Plusieurs des problèmes d'optimisation impliqués sont extrêmement complexes et mettent au défi les méthodes actuelles dus à leur dimension et à leur non linéarité. Les travaux de recherches dans le domaine se divisent principalement en deux écoles d'idées. La première tendance consiste à effectuer certaines approximations afin de simplifier la formulation du problème et de le rendre tangible par les méthodes d'optimisation classiques. Ces algorithmes sont généralement très efficaces, mais la solution calculée n'est optimale que pour le modèle simplifié et non pas le problème original. La deuxième tendance consiste à formuler le problème le plus proche possible de la réalité et à le résoudre à l'aide d'une métaheuristique. Ces algorithmes non déterministes ont l'avantage de considérer les problèmes non linéaires et offrent une meilleure résilience contre la convergence prématurée vers un optimum local. Cependant, les métaheuristiques nécessitent une puissance de calculs considérable ce qui limite leur utilisation dans des applications où le temps de réaction est critique. Dans cette thèse, nous proposons de mitiger cette limitation par une implémentation parallèle sur GPU.

Afin de mieux comprendre la motivation derrière cette thèse et de démontrer l'importance de la contribution apportée, nous présentons dans ce chapitre une revue de la littérature sur les métaheuristiques parallèles et leurs applications à l'optimisation des réseaux électriques intelligents. Les trois problèmes considérés sont la minimisation des harmoniques d'un onduleur multiniveau, l'optimisation de l'écoulement de puissance (OPF de l'anglais *optimal power flow*) et la reconfiguration des réseaux de distribution (DFR de l'anglais *distribution feeder reconfiguration*). Ces problèmes sont d'une grande complexité et couvrent l'étendue des réseaux intelligents en considérant le contrôle de la production, du transport et de la distribution de l'électricité. Les méthodes actuelles ne peuvent assurer un contrôle efficace puisqu'elles se basent sur des approximations ou requièrent des temps de calcul plus longs.

La revue de la littérature présentée dans ce chapitre est organisée comme suit. Nous introduisons d'abord quelques concepts communs au calcul parallèle. Nous expliquons ensuite quatre modèles de parallélisation pour l'implémentation de métaheuristiques sur GPU. Nous poursuivons avec la présentation de quelques cadriciels notables qui permettent de faciliter l'utilisation et le développement futur de métaheuristiques. Nous enchaînons ensuite avec les trois applications aux réseaux électriques intelligents, soit la minimisation des harmoniques d'un onduleur multiniveau, l'optimisation de l'écoulement de puissance et la reconfiguration des réseaux de distribution. Pour chacune des applications, notre discussion inclut les méthodes déterministes et les métaheuristiques. En conclusion, nous réitérons les lacunes des méthodes actuelles et expliquons comment les travaux de recherches présentés dans cette thèse permettent de les adresser.

2.1 Objectif de la parallélisation

Le but premier d'une implémentation parallèle est de réduire le temps d'exécution d'un programme séquentiel. On dit alors que le programme parallèle offre une accélération par rapport au programme séquentiel. D'après [33], l'accélération S_p d'un programme parallèle ayant un temps d'exécution $T_p(n)$ est définie comme suit :

$$S_p(n) = \frac{T_s(n)}{T_p(n)} \tag{2.1}$$

où T_s est le temps d'exécution du programme séquentiel, l'indice p est le nombre de processeurs et n est la dimension du problème. Plus le temps du programme parallèle est réduit, plus l'accélération est grande. L'accélération est une métrique simple qui permet de quantifier le gain de performance d'une implémentation parallèle. Toutefois, cette métrique est quelque peu limitée et ne permet pas toujours d'identifier pleinement l'avantage de la parallélisation. Prenons l'exemple illustré à la Figure 2.1 où l'on a un programme avec une fraction f intrinsèquement séquentielle et une fraction $(1 - f)$ qui elle peut être parallélisée. Si le temps d'exécution du programme est de T_s, alors le temps de la partie intrinsèquement séquentielle est de $f * T_s$. De plus, si nous assumons une accélération parfaite, le temps de la partie parallélisée est réduit à $(1 - f) * T_s/p$. Finalement, le temps d'exécution total du programme parallèle est $[f * T_s] + [(1 - f) * T_s/p]$, soit la somme des parties séquentielles et parallèles.

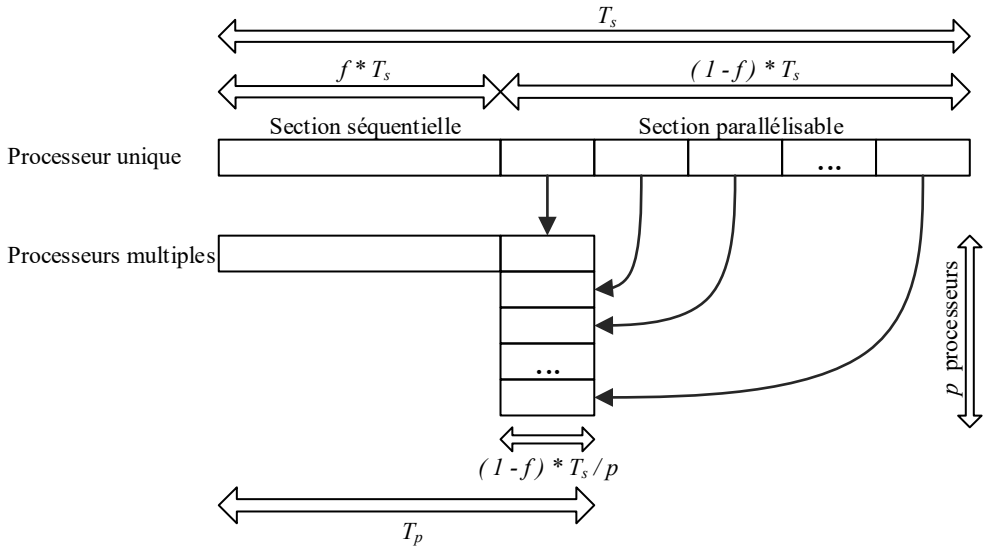


Figure 2.1: Loi d'Amdahl

Par conséquent, l'accélération maximale du programme est limitée par sa partie intrinsèquement séquentielle et est nécessairement plus petite que $1/f$ comme il est montré à l'équation (2.2). Cette observation est connue sous le nom de la loi d'Amdahl [33]. Prenons par exemple un programme séquentiel dont 20% du temps d'exécution ne peut être parallélisé. Ce programme aura une accélération maximale de 5x même si le temps de calcul de la partie parallèle est infiniment réduit. Selon l'application, ce gain médiocre ne justifie pas toujours l'effort supplémentaire associé à la parallélisation.

$$S_p = \frac{T_s}{f * T_s + \frac{1-f}{p} T_s} = \frac{1}{f + \frac{1-f}{p}} = \frac{p}{1 + (p-1) * f} \leq \frac{1}{f} \tag{2.2}$$

D'un autre côté, l'objectif de la parallélisation n'est pas toujours de réduire le temps d'exécution, mais peut consister à augmenter la quantité de calculs effectués dans un temps constant. On parle alors d'accélération ajustée S'_p. Introduite par Gustafson [33], l'accélération ajustée se définit comme suit :

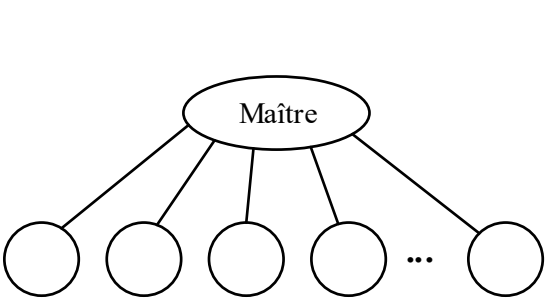
$$S'_p(p * n) = \frac{T_s(p * n)}{T_p(p * n)} \tag{2.3}$$

où T_p est le temps d'exécution du programme parallèle, T_s est le temps d'exécution du programme séquentiel, p est le nombre de processeurs et $p * n$ est la dimension du problème. Autrement dit, à mesure qu'on ajoute des processeurs, on augmente la

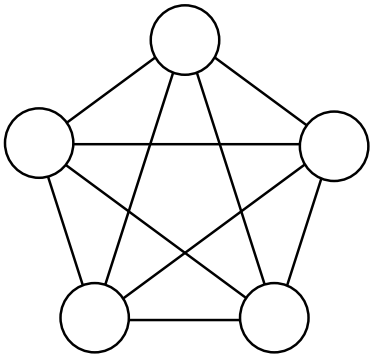
quantité de travail et on compare le temps d'exécution de l'algorithme parallèle au temps que prend l'algorithme séquentiel pour effectuer cette même quantité de travail. L'accélération ajustée est donc une mesure du gain de performance d'un programme parallèle dont l'objectif est d'augmenter la quantité de calculs exécutés en maintenant un temps d'exécution fixe. Dans le cas de l'optimisation des réseaux intelligents, nous verrons plus tard pour chacune des applications que le résultat désiré est en fait une combinaison des deux objectifs afin de pouvoir considérer des problèmes plus grands tout en assurant un temps de calcul plus petit. Maintenant que nous avons expliqué les objectifs de la programmation parallèle, regardons quelles sont les techniques possibles pour paralléliser les métaheuristiques.

2.2 Métaheuristiques parallèles

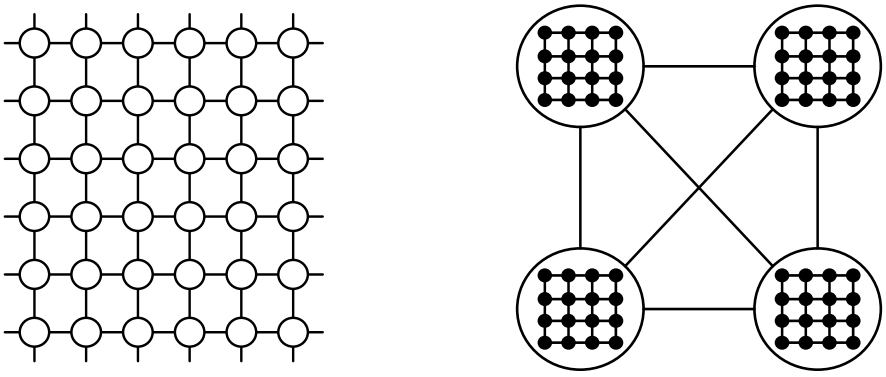
Dans la littérature, la parallélisation des métaheuristiques suit essentiellement quatre modèles. Ces modèles sont discutés dans [34] et [35] où ils sont principalement appliqués à l'algorithme génétique (GA de l'anglais *genetic algorithm*). Or, comme l'explique les auteurs de [36], les métaheuristiques partagent plusieurs similarités et une technique de parallélisation développée pour un algorithme peut habituellement être appliquée à un autre. Illustrés à la Figure 2.2, ces quatre modèles parallèles sont le modèle maître-esclave, le modèle à gros grains, le modèle à grains fins et le modèle hybride.



(a) Modèle maître-esclave



(b) Modèle à gros grains



(c) Modèle à grains fins sur grille 2D **(d) Modèle hybride**

Figure 2.2: Les quatre modèles parallèles des métaheuristiques

2.2.1 Modèle maître-esclave

Ce modèle est composé d'un processus maître qui contrôle séquentiellement l'exécution de l'algorithme et délègue le calcul de la fonction de coût aux processus esclaves. Chaque processus esclave calcule le coût d'une ou plusieurs solutions. Simple d'implémentation, ce modèle offre une accélération limitée puisqu'une seule étape de l'algorithme est parallélisée. Les auteurs de [37] et [38] utilisent ce modèle pour implémenter l'algorithme d'optimisation par essaim de particules (PSO de l'anglais *particle swarm optimization*) sur GPU. Une accélération de 27x est atteinte dans [37] et de 43x dans [38]. Cependant, ces deux approches ne sont pas extensibles à un nombre plus grand de particules puisque l'initialisation, la recherche pour la meilleure particule et le calcul des vitesses et des positions sont effectués séquentiellement sur le CPU nécessitant un transfert de données entre le CPU et le GPU à chaque itération. Ce modèle est aussi utilisé dans [39] pour paralléliser le GA sur GPU avec une accélération de 6.2x. Dans le cas des métaheuristiques basées sur une seule solution, les auteurs de [40] et [41] suivent une approche maître-esclave où le GPU est utilisé pour explorer le voisinage et évaluer le coût de plusieurs solutions candidates afin de sélectionner celle qui est la meilleure avant de mettre à jour la solution unique gardée sur le CPU.

2.2.2 Modèle à gros grains

Le modèle à gros grains (ou modèle avec îlots) utilise plusieurs populations de solutions candidates qui évoluent simultanément sur chacun des processus. L'algorithme inclut un système de migration qui permet d'échanger des solutions entre les populations. Ce modèle modifie cependant le comportement de la métaheuristique originale et la topologie de migration choisie aura un impact positif ou négatif sur la qualité du résultat final [42]. Il est intéressant de noter que le modèle avec îlots peut

être utilisé pour combiner différentes métaheuristiques dans un seul programme d'optimisation permettant d'exploiter les forces de chacune comme le font avec succès les auteurs de [43]. Le modèle à gros grains est adapté pour les CPU multicœurs, mais n'exploite pas un niveau de parallélisme suffisant pour les GPU. Ce modèle est utilisé dans [44] [45], [46] et [47] pour l'implémentation du PSO et du GA sur processeurs multicœurs.

2.2.3 Modèle à grains fins

Le modèle à grains fins utilise une population unique. Habituellement implémentées sur un système massivement parallèle tel qu'un GPU, les solutions candidates sont distribuées sur les multiples processeurs et peuvent uniquement interagir avec les solutions voisines selon la topologie du matériel afin de minimiser les délais de communication. Le comportement de la métaheuristique est nécessairement affecté par la topologie choisie. Dans [48], Zhou et Tan implémentent le PSO sur GPU d'après ce modèle. Au lieu d'utiliser un modèle de communication globale comme dans le PSO original [49], ils limitent la communication de chaque particule aux deux voisines les plus proches. Ils obtiennent une accélération de 6.47x. Dans [50], [51] et [52], nous proposons des implémentations parallèles à grains fins du PSO sur GPU. Ces implémentations utilisent un essaim unique, ne modifient pas le comportement de l'algorithme original, utilisent une réduction parallèle optimisée pour les GPU [53] et minimisent la divergence d'exécution entre les threads. Le transfert de données entre le CPU et le GPU se fait uniquement au début et à la fin de l'algorithme. Notre approche permet une accélération de 215.6x. Le modèle à grains fins est aussi utilisé dans [54] pour paralléliser le GA et permet une accélération de 10.9x. Cette implémentation est plus avantageuse que l'approche maître-esclave publiée dans [39] puisqu'elle parallélise toutes les étapes de l'algorithme. Toutefois, l'accélération résultante reste limitée pour deux raisons. Premièrement, l'utilisation d'une seule population ne permet pas de séparer les solutions candidates. Celles-ci doivent donc être sauvegardées dans la mémoire globale du GPU et ne peuvent être chargées dans les mémoires partagées au début de chaque fonction parallèle. Contrairement aux mémoires partagées, la mémoire globale n'est pas embarquée sur la puce du processeur graphique et induit des délais d'accès importants. De plus, ces délais sont augmentés par un accès à la mémoire non coalescent. Deuxièmement, l'approche publiée dans [39] requiert l'ordonnancement des solutions parents avant d'effectuer l'opération d'enjambement. Cet ordonnancement est implémenté à l'aide d'un tri bitonique parallèle qui a un ordre de complexité de $O(n * \log^2 n)$ [55]. Cette opération devient donc très coûteuse lorsque le nombre de chromosomes augmente. Une approche préférable est de sélectionner les parents par un processus de tournoi ce qui ne requiert pas d'ordonnancement [56]. Les auteurs de [57] utilisent cette technique et obtiennent une accélération de 46x pour 16 384 chromosomes. Le

modèle à grains fins sur GPU a aussi été utilisé pour l'algorithme d'optimisation par colonies de fourmis dans [58], [59], [60] et [61] et le celui du recuit simulé dans [62], [63], [64] et [65].

2.2.4 Modèle hybride

Finalement, le modèle parallèle hybride est un mixte des deux méthodes précédentes. Cette approche est conçue sur mesure, pour un problème spécifique, exécuté sur un système en particulier. Par exemple, la topologie à la Figure 2.2d illustre un algorithme hybride sur un réseau interconnecté de quatre grilles 2D. Le modèle parallèle hybride est complexe et s'applique à des calculs scientifiques demandant une très grande puissance de calcul. Ce modèle s'adapte parfaitement aux processeurs graphiques où plusieurs populations peuvent être exécutées indépendamment sur différents multiprocesseurs tandis que chaque solution est exécutée sur un cœur unique à l'intérieur d'un multiprocesseur. À une fréquence prédéterminée, les différentes populations peuvent s'échanger des solutions par l'entremise de la mémoire globale comme le proposent les auteurs de [66]. Une telle approche permet d'exploiter la rapidité de la mémoire partagée et de minimiser les opérations de synchronisation entre les populations. Les auteurs de [67] utilisent le modèle parallèle hybride pour paralléliser le GA afin d'accélérer la résolution de casse-têtes sudoku. Toutefois, leur solution n'implémente aucun processus de migrations entre les populations ce qui évite les délais de communications, mais affecte la qualité de la solution finale. Zheng et coll. [66] présentent eux aussi un GA parallèle en îlots sur GPU, mais cette fois avec migrations.

2.3 Cadriciel pour métaheuristiques

Jusqu'à maintenant, nous avons introduit le concept d'accélération, présenté quatre modèles de parallélisation et discuté de plusieurs implémentations parallèles sur GPU. Il est évident qu'un effort important est requis de la part du programmeur pour implémenter une métaheuristique performante et bien adaptée à une application spécifique. Heureusement, au cours des dernières années, plusieurs cadriciels ont été développés pour faciliter l'utilisation de ces algorithmes d'optimisation. Un cadriciel offre des fonctions ou modules logiciels prêts à être utilisés, mais contrairement à une bibliothèque logicielle, le cadriciel définit aussi le cadre de travail et la structure générique de ces modules de façon à permettre leur extensibilité. Un programmeur peut donc utiliser une métaheuristique déjà implémentée par le cadriciel ou développer une nouvelle métaheuristique en suivant la structure logicielle imposée par l'outil. Le cadriciel facilite l'utilisation des métaheuristiques et leur adaptation à des problèmes spécifiques, mais aussi l'amélioration ou même le développement de nouvelles métaheuristiques.

Dans une étude conduite en 2012 [68], Parejo et coll. identifient 33 cadriciels pour les métaheuristiques. Ils sélectionnent les dix plus importants et effectuent une comparaison exhaustive en utilisant 271 critères d'évaluations afin de souligner leurs forces, mais aussi d'identifier les possibilités de développements futurs. Dans leur analyse, Parejo et coll. discutent entre autres des éléments typiques d'un cadriciel pour les métaheuristiques (MOF de l'anglais *Metaheuristic Optimization Framework*). Ils utilisent une carte conceptuelle que nous reproduisons à la Figure 2.3 pour représenter les notions communes aux MOF. Cette carte nécessite quelques explications. Il y a d'abord l'application logicielle qui utilise un MOF pour résoudre un problème. Le MOF contient plusieurs métaheuristiques déjà implémentées, mais définit aussi un cadre bien structuré permettant le développement de nouvelles métaheuristiques. Les métaheuristiques sont indépendantes du problème et offrent une stratégie de recherche pour améliorer de façon itérative les solutions candidates. Ces solutions sont encodées et leur décodage est spécifique au problème et nécessaire avant d'évaluer leur coût à l'aide de la fonction objective. Malgré que les métaheuristiques restent génériques, elles peuvent utiliser des heuristiques, ou opérateurs, spécialement adaptées au problème en question. Utiliser un MOF pour résoudre un problème précis nécessite

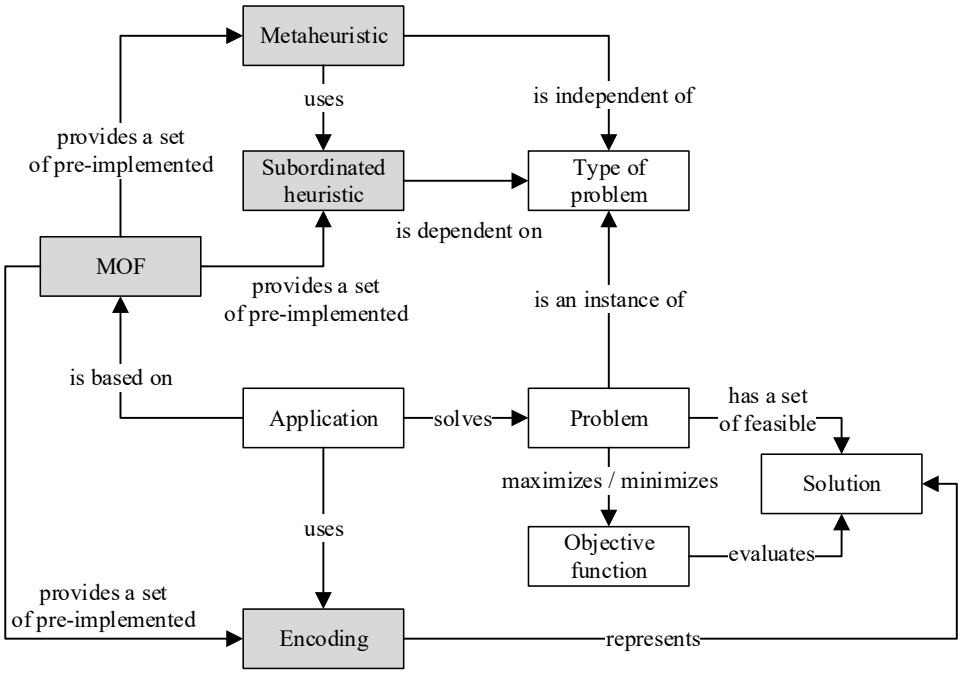


Figure 2.3: Carte conceptuelle des notions communes aux MOF (reproduit de [68])

donc la sélection de la métaheuristique, le choix d'un encodage, le développement d'une fonction de décodage, le développement d'une fonction de coût et le développement d'heuristiques ou d'opérateurs adaptés au problème considéré et à l'encodage utilisé. Ce travail est considérable, mais beaucoup plus facile que de programmer une métaheuristique de A à Z. Bref, le MOF diminue l'effort de programmation et offre une structure robuste minimisant les erreurs d'implémentation.

Dans les paragraphes qui suivent, nous présentons quelques cadriciels pour les métaheuristiques. *FOM* ou *Framework for Metaheuristic Optimization* [69] est l'un des premiers cadriciels à être publiés. Apparu en 2003, ce cadriciel utilise le paradigme de programmation orientée objet et des classes abstraites pour définir la structure du problème, des solutions et de la métaheuristique. L'outil s'efforce de bien séparer l'algorithme d'optimisation du problème de façon à permettre sa réutilisation. Une autre approche pour implémenter cette séparation est l'utilisation d'agents. Par exemple, l'auteur de [70] divise les responsabilités du programme d'optimisation en quatre agents. Les solutions candidates sont gérées par l'agent *Solution*. L'agent *Problème* est responsable d'évaluer le coût des solutions. L'agent *Algorithme* modifie les solutions de façon à les améliorer. Finalement, l'agent *Conseiller* s'occupe de configurer l'algorithme. L'exécution des agents et les communications inter-agents sont implémentées à l'aide de *A-Globe* [71], une librairie logicielle pour systèmes multi-agents. Telle qu'illustrée à la Figure 2.4, cette approche multi-agents permet l'exécution concurrentielle de plusieurs métaheuristiques. On parle alors de métaheuristiques hybrides coopératives.

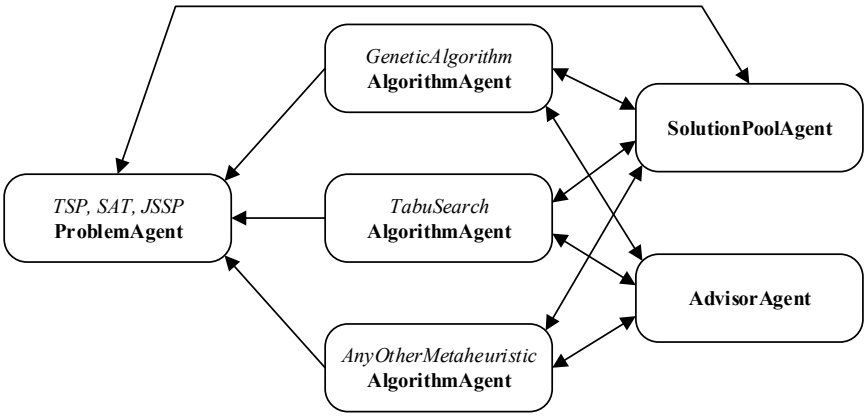


Figure 2.4: Système multi-agents pour les métaheuristiques (reproduit de [72])

Comparativement à *FOM*, ce cadriciel multi-agents est lourd et peut être difficile à intégrer dans un programme logiciel. C'est pourquoi la plupart des cadriciels récents suivent l'exemple de *FOM* et utilisent une approche orientée objet. *jMetal* [72] et

Opt4J [73] en sont des exemples. Programmés en Java, ces cadriciels offrent eux aussi une bonne séparation entre le problème, l'encodage des solutions et la métaheuristique. À titre d'exemple, nous avons choisi de reproduire le diagramme UML de classes de *jMetal* à la Figure 2.5. Sur ce diagramme, nous avons encadré à l'aide d'une ligne pointillée les classes centrales à l'outil. Ces classes sont abstraites et définissent les interfaces. Les autres sont des classes dérivées qui implémentent ces interfaces. Par exemple, il y a la classe abstraite *Algorithm* qui définit les caractéristiques et fonctions communes à tous les algorithmes tandis que la classe *SMPSO* est une implémentation particulière de l'algorithme d'optimisation par essaim de particules. D'après cette structure, un algorithme utilise différents opérateurs pour modifier et améliorer un ensemble de solutions candidates afin de résoudre un problème. Le problème définit l'encodage à utiliser et chaque variable ou élément d'une solution est construit d'après cet encodage. Par exemple, une solution qui utilise l'encodage *Real* sera composée d'une ou plusieurs variables où chacune sera représentée par un nombre réel.

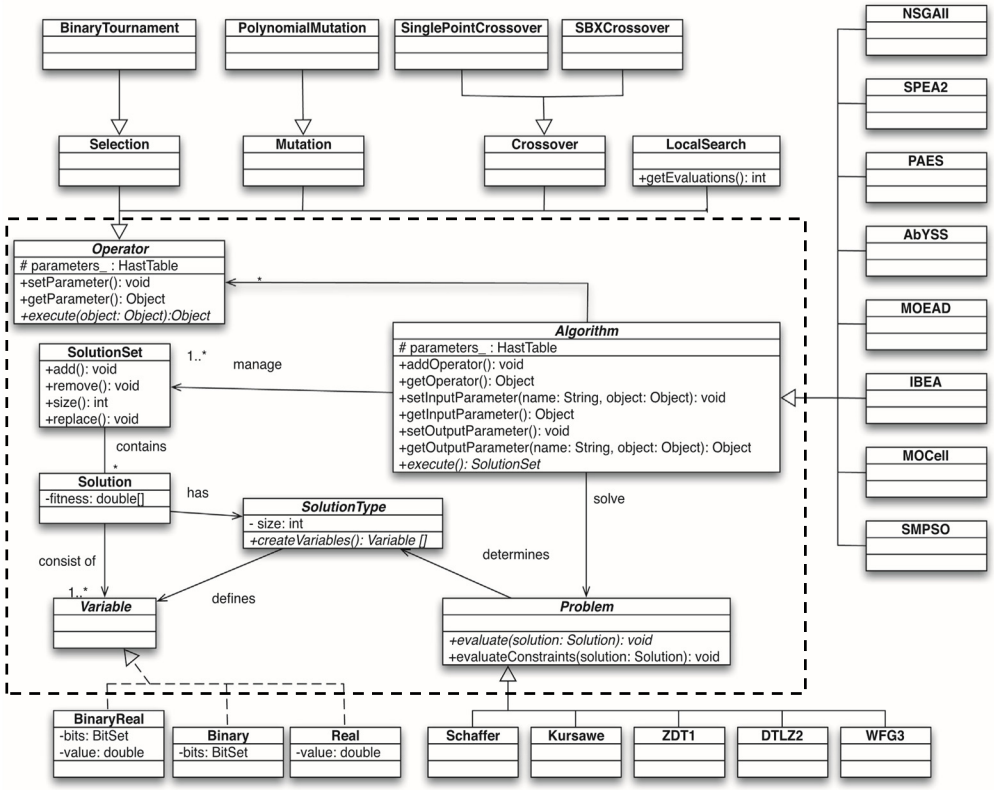


Figure 2.5: Diagramme UML de classes du cadriciel *jMetal* (reproduit de [72])

Au niveau des implémentations parallèles, *jMetal* [72] et *Opt4J* [73] offrent un support limité. Seuls certains éléments du cadriciel sont parallélisés. Par exemple, l'algorithme NSGAII (de l'anglais *Non-dominated Sorting Genetic Algorithm-II*) implémenté dans *jMetal* supporte une exécution multi-thread. Similairement, *Opt4J* permet le calcul de la fonction de coût en parallèle sur des systèmes multicœurs à mémoire partagée. Pour un meilleur support, il faut se tourner vers [74] où un cadriciel pour les métaheuristiques parallèles sur système distribué est présenté. Basé sur *Opt4J*, ce cadriciel utilise un modèle parallèle à gros grains pour diviser les solutions candidates en îlots et partager l'exécution de ces îlots sur plusieurs processeurs. Le cadriciel proposé définit un modèle de communication efficace permettant une exécution sur un grand nombre de processeurs autant homogènes qu'hétérogènes. En adoptant le modèle parallèle à gros grains, ce cadriciel exploite un parallélisme au niveau des tâches et ne peut être implémenté sur des systèmes massivement parallèles tels que les GPU. En fait, les cadriciels pour métaheuristiques massivement parallèles sont très rares. Notre revue de la littérature nous a permis d'en identifier qu'un seul. Il s'agit de *ParadisEO-MO-GPU* [41]. Développé en 2013, ce cadriciel supporte uniquement les algorithmes de recherche locale basés sur une seule solution candidate et utilise le GPU pour générer et évaluer en parallèle un grand nombre de solutions voisines. Les coûts des solutions voisines sont ensuite transférés vers le CPU pour que l'algorithme puisse identifier le meilleur voisin avant de mettre à jour la solution actuelle. Malgré que le cadriciel offre une accélération par rapport à une implémentation séquentielle sur CPU, celle-ci est limitée par le transfert de données entre le CPU et le GPU à chaque itération de la recherche et par le code séquentiel exécuté sur le CPU.

Étant donné que les métaheuristiques sont intrinsèquement parallèles et que les GPU offrent une très grande puissance de calcul, il est surprenant qu'il n'existe pas d'autre cadriciel qui supporte les GPU. Or, ceci s'explique par le fait que la plupart des MOF utilisent une architecture orientée objet pour définir la structure des différents éléments qui composent le cadriciel. Cette approche permet à l'utilisateur de développer de nouveaux modules sans avoir à repenser leur architecture. D'un autre côté, pour bénéficier de l'architecture parallèle du GPU, il est nécessaire d'exploiter une parallélisation au niveau des données. Pour ce faire, les données doivent être organisées séquentiellement en mémoire de façon à permettre un accès coalescent. Dans le cas de cadriciels tels que *jMetal* [72] et *Opt4J* [73], un objet est créé pour chaque solution et chaque variable de la solution. Pour accéder une donnée, il faut donc déréférencer la solution puis la variable ce qui résulte en deux niveaux d'accès indirectes à la mémoire. Ceci est acceptable pour une implémentation séquentielle sur CPU, mais impossible dans le cas d'une implémentation parallèle sur GPU puisque tous ces accès aléatoires à la mémoire seraient exécutés séquentiellement. Presque la totalité des cœurs du GPU serait en attente pendant qu'un seul exécuterait. Adapter un cadriciel existant pour le GPU nécessite donc des

modifications au cœur même de son architecture. En fait, *ParadisEO-MO-GPU* [41] n'est qu'une extension du cadriciel *ParadisEO* [75] et utilise le GPU uniquement pour la génération et l'évaluation des solutions voisines. *ParadisEO-MO-GPU* [41] ne modifie pas l'architecture de *ParadisEO*. L'algorithme et l'amélioration de la solution candidate continuent d'être exécutés séquentiellement sur le CPU et nécessitent un transfert de données entre le CPU et le GPU à chaque itération de l'optimisation.

Pour réellement exploiter l'architecture massivement parallèle des GPU, il serait nécessaire de développer un cadriciel qui permettrait de paralléliser toutes les étapes de l'algorithme et non pas uniquement celle de la fonction de coût. Ce cadriciel devrait utiliser le paradigme de programmation orientée objet à un haut niveau pour bien définir la structure des différents modules qui le composent, mais utiliser des tableaux continus pour sauvegarder les solutions candidates permettant un accès coalescent à la mémoire. De plus, ces solutions devraient être créées, modifiées et supprimées sur le GPU même, de façon à minimiser tout échange de données sur le bus PCI-Express. Tout comme *jMetal* [72], *Opt4J* [73] et *ParadisEO* [75], ce cadriciel devrait aussi supporter les métaheuristiques hybrides afin d'améliorer l'efficacité de la recherche. Pour faire le lien avec l'étude de Parejo et coll. [68] dont nous avons discuté au début de cette section, nos recommandations sont en ligne avec celles des auteurs. En effet, suite à leur comparaison de dix cadriciels pour les métaheuristiques, Parejo et coll. ont identifié le besoin pour des implémentations parallèles plus performantes, des méthodes hybrides et le développement de MOF qui suivent de meilleures pratiques du génie logiciel.

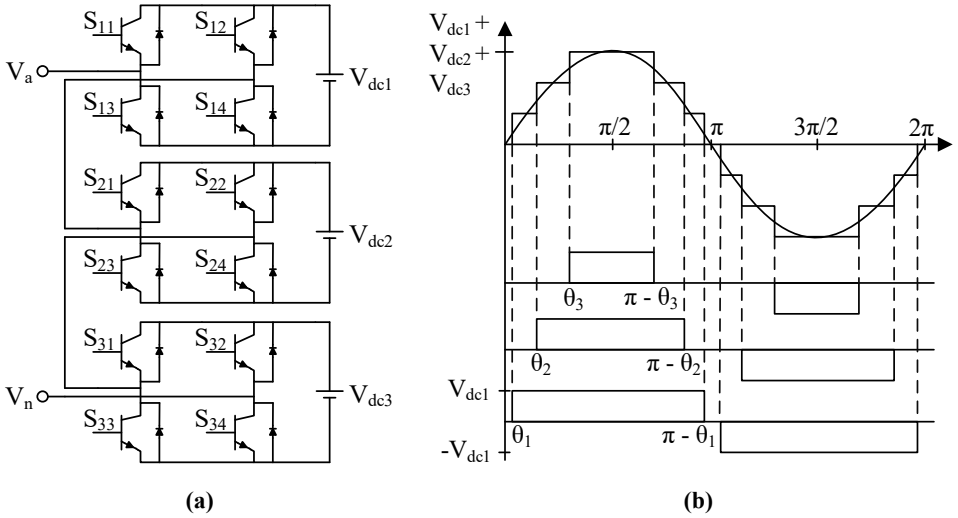
Dans de but de répondre à ce besoin, nous présentons dans cette thèse le cadriciel *GPU Metaheuristic Framework*, ou simplement *gpuMF*, pour l'implémentation de métaheuristiques parallèles sur GPU. Ce cadriciel remplit toutes les exigences énoncées au paragraphe précédent et inclus des implémentations complètes des algorithmes de PSO et de GA. Le cadriciel *gpuMF* offre une division claire entre la méthode d'optimisation et le problème considéré ce qui permet de le réutiliser facilement pour chacune des applications considérées dans cette thèse. En exploitant l'architecture massivement parallèle des GPU, *gpuMF* offre une accélération significative comparée à une implémentation séquentielle sur CPU. Comme le montrent les tests expérimentaux au Chapitre 5, cette accélération dépassent les 400x.

2.4 Minimisation des harmoniques d'un onduleur multiniveau

La première application considérée dans cette thèse est la minimisation des harmoniques d'un onduleur multiniveau. Les onduleurs sont des dispositifs d'électronique de puissance qui permettent de générer une puissance alternative à partir d'une source continue. Au cours des dernières années, il y a eu un intérêt croissant pour les onduleurs multiniveaux avec plusieurs sources indépendantes, en

particulier dans le secteur de l'énergie distribuée renouvelable en raison du fait que les batteries, les condensateurs, les panneaux solaires et les piles à combustible sont disponibles et peuvent être connectés via un onduleur pour produire la tension requise [76]. Comparés aux onduleurs traditionnels à deux niveaux, les onduleurs multiniveaux produisent une puissance de meilleure qualité, réduisent les pertes de commutation, offrent une meilleure compatibilité électromagnétique, supportent des niveaux de tension plus élevés et permettent de minimiser davantage les harmoniques indésirables [77], [78]. Les onduleurs multiniveaux sont utilisés dans les réseaux électriques intelligents tels que pour les systèmes de transport haute tension à courant continu [79], les véhicules hybrides [80], les centrales solaires photovoltaïques [81], les moteurs à vitesse variable [82] et les modules de transmission flexibles (FACTS de l'anglais *flexible alternating current transmission system*) [83].

Les trois architectures les plus communes pour les onduleurs multiniveaux sont : l'architecture avec diodes de blocage [84], l'architecture avec condensateurs à pompe de charge [85] et l'architecture cascadée [86]. C'est cette dernière configuration qui est la plus populaire étant donné son organisation modulaire et sa simplicité d'implémentation [87]. L'architecture cascadée utilise plusieurs ponts en H connectés en série, chacun alimenté par une source de tension continue comme à la Figure 2.6 (a) pour un onduleur à trois sources. Le contrôle de l'onduleur se fait en ajustant les angles de commutation afin de générer une forme d'onde en escalier comme à la Figure 2.6 (b). Puisque cette sortie est formée de niveaux de tension discrets, le taux de distorsion harmonique résultant est souvent élevé. Or, ces harmoniques affectent la qualité de la puissance produite et entraînent des pertes significatives sous forme de surchauffe au niveau des lignes de transport à haute tension, des transformateurs et des moteurs électriques [87], [88], [89]. Le problème de la minimisation des harmoniques d'un onduleur multiniveau consiste à calculer précisément les angles de commutation de façon à produire la tension désirée d'après l'indice de modulation tout en minimisant les harmoniques qui sont indésirables.



(a) **(b)**

Figure 2.6: (a) Topologie d'un onduleur de tension multiniveau avec trois modules pont en H cascadés et (b) onde de tension de sortie associée

Dans la littérature, les techniques de minimisation des harmoniques peuvent être catégorisées en deux groupes : les méthodes déterministes et les méthodes non déterministes. Un exemple appartenant au premier groupe est la méthode de Newton-Raphson (N-R) [90], [91], [92]. Suivant cette technique, la composante fondamentale et les harmoniques de l'onde de sortie en escalier sont exprimées sous forme de séries de Fourier. Un système d'équations transcendantes est ensuite obtenu en spécifiant la valeur de la composante fondamentale d'après l'indice de modulation et en forçant les premiers harmoniques à zéro. Un exemple d'un tel système d'équations pour éliminer trois harmoniques d'un onduleur à quatre sources est donné ci-dessous :

$$\begin{aligned} V_1 &= \frac{4}{\pi}\left[V_{dc_1}\cos(\theta_1) + V_{dc_2}\cos(\theta_2) + V_{dc_3}\cos(\theta_3) + V_{dc_4}\cos(\theta_4)\right] = \frac{4M\sum_{s=1}^{3} V_{dc_s}}{\pi} \\ V_3 &= \frac{4}{3\pi}\left[V_{dc_1}\cos(3\theta_1) + V_{dc_2}\cos(3\theta_2) + V_{dc_3}\cos(3\theta_3) + V_{dc_4}\cos(3\theta_4)\right] = 0 \\ V_5 &= \frac{4}{5\pi}\left[V_{dc_1}\cos(5\theta_1) + V_{dc_2}\cos(5\theta_2) + V_{dc_3}\cos(5\theta_3) + V_{dc_4}\cos(5\theta_4)\right] = 0 \\ V_7 &= \frac{4}{7\pi}\left[V_{dc_1}\cos(7\theta_1) + V_{dc_2}\cos(7\theta_2) + V_{dc_3}\cos(7\theta_3) + V_{dc_4}\cos(7\theta_4)\right] = 0 \end{aligned} \quad (2.4)$$

où V_1 est l'amplitude de la composante fondamentale, V_h est l'amplitude de l'harmonique d'ordre h, V_{dc_s} est la tension de la source s, θ_s est l'angle de commutation de la source s et M est l'indice de modulation spécifiée. Il est important de noter que les harmoniques d'ordres pairs sont nulles étant donné la symétrie de l'onde de sortie. C'est pourquoi le système d'équations (2.4) considère uniquement les harmoniques impairs. D'après cette formulation, il est évident qu'avec quatre

angles de commutation, trois harmoniques peuvent être éliminées [93]. De façon plus générale, N sources de tension sont nécessaires pour éliminer $N-1$ harmoniques. La méthode de N-R utilise un procédé itératif basé sur la dérivée des équations transcendantes pour calculer les angles de commutation qui satisfont le système d'équations. L'algorithme de N-R dépend toutefois sur un estimé initial, peut converger vers un minimum local ou peut diverger spécialement lorsque le nombre de sources de l'onduleur est grand [13]. De façon à mitiger ce dernier problème, les auteurs de [87] proposent une formule empirique pour calculer un meilleur estimé initial augmentant ainsi les chances de convergence. L'efficacité de leur approche est cependant limitée à un onduleur à cinq sources. Malgré qu'il puisse exister plusieurs solutions au système d'équations, la méthode de N-R permet d'en identifier qu'une seule. Pour obtenir toutes les racines possibles, les auteurs de [94] suggèrent d'utiliser la fonction *fsolve* de MATLAB®. Cette fonction se base sur la méthode de Gauss-Newton (G-N) et peut intégrer une recherche linéaire avec interpolation quadratique ou cubique. Une autre méthode possible est de convertir les équations transcendantes en équations polynomiales et de les résoudre par la théorie mathématique des résultants [95], [96]. Cette approche a l'avantage d'identifier l'ensemble des solutions possibles, mais le désavantage d'être compliquée, d'être longue à calculer et de nécessiter la formulation de nouvelles expressions lorsque les sources de tension varient. De plus, quand le nombre de sources est augmenté, le degré des polynômes devient très grand. Ce dernier peut être réduit par la théorie des polynômes symétriques [97], mais la région des solutions faisables reste toutefois étroite et le système est difficile à résoudre [98]. La méthode des résultants a été démontrée efficace pour le contrôle d'un onduleur à cinq sources [97].

Les méthodes déterministes basées sur N-R, G-N ou la théorie des résultants ont plusieurs limitations importantes. Tout d'abord, leur efficacité est limitée à des onduleurs avec un petit nombre de sources. Lorsque celui-ci augmente, elles ont tendance à diverger [99]. De plus, d'après la formulation utilisée, il peut y avoir plusieurs valeurs de M pour lesquelles le système d'équations (2.4) est insoluble. Dans ces situations, les algorithmes déterministes présentés n'offrent aucune solution alternative [99]. Finalement, étant donné que ces algorithmes considèrent uniquement les harmoniques de bas niveau, le taux de distorsion total de la tension de sortie peut rester élevé entraînant ainsi des pertes de puissance importantes [77] comme nous l'avons expliqué précédemment.

Malgré que notre description des méthodes déterministes soit quelque peu négative, il en existe une qui est très intéressante. Dans [32] et [100], Liu, Hong, et Huang proposent une formulation différente du problème de minimisation des harmoniques d'un onduleur multiniveau. Au lieu d'éliminer complètement quelques harmoniques de bas niveaux, leur approche vise plutôt à minimiser le taux de distorsion harmonique total considérant ainsi toutes les harmoniques. Leur méthode

est supportée par une preuve mathématique qui démontre que l'onde en escalier associée aux angles de commutations calculés a un taux de distorsion harmonique minimum. L'algorithme est implémenté sur une matrice prédiffusée programmable par l'utilisateur (FPGA de l'anglais *field-programmable gate array*) et permet de contrôler en temps réel un onduleur à trois sources. Cependant, comme l'identifient les auteurs de [101], cette méthode n'est pas optimale pour une application triphasée puisqu'elle inclut obligatoirement les harmoniques dont l'ordre est divisible par trois. Dans une application triphasée, ces harmoniques n'ont pas besoin d'être réduites puisqu'elles sont automatiquement éliminées. Une formulation qui omettrait ces harmoniques permettrait de réduire les autres davantage. De plus, la méthode de Liu, Hong, et Huang [32], [100] impose une limite sur les valeurs possibles pour l'indice de modulation. Toutefois, de par sa simplicité et sa rapidité d'exécution, cette méthode est une alternative très intéressante pour la minimisation des harmoniques d'un onduleur multiniveau. Pour cette raison, nous reprenons dans [102] cette méthode et dérivons une formule mathématique permettant d'identifier précisément les limites sur l'indice de modulation. Nous identifions que ces limites varient en fonction du nombre et de la tension des sources. Dans [102], nous implémentons cette méthode sur un GPU et calculons les angles pour des onduleurs contenant jusqu'à 1000 sources tout en gardant un temps d'exécution plus court que l'implémentation sur FPGA dans [32], qui elle, considère seulement trois sources.

La deuxième catégorie de techniques de minimisation des harmoniques d'un onduleur multiniveau regroupe les méthodes non déterministes telles que les réseaux de neurones artificiels et les métaheuristiques. Dans [81], un réseau perceptron à deux couches est utilisé pour contrôler un onduleur à cinq sources. Le réseau est entraîné de façon à calculer les angles de commutation pour différents indices de modulation étant donné des tensions nominales aux sources. Comparé aux méthodes déterministes, le réseau de neurones a l'avantage de pouvoir interpoler les résultats et de générer des angles adéquats même lorsqu'aucune solution au système d'équations transcendantes discuté précédemment n'existe pas. Le réseau de neurones a aussi l'avantage d'être très rapide et peut-être implémenté sur un système parallèle. D'un autre côté, le réseau proposé considère uniquement des sources de tension fixes. Si une des tensions varie, il faut répéter l'entraînement pour la nouvelle valeur. Afin de supporter des tensions variables, les auteurs de [103] utilisent un algorithme génétique pour calculer les angles de commutation optimaux pour différentes valeurs de tension et entraîne le réseau de neurones à l'aide des résultats obtenus. Leur approche est testée sur un onduleur à cinq sources et minimise efficacement les quatre premières harmoniques impaires. L'extensibilité de cette méthode à de plus gros onduleurs reste toutefois à vérifier. En effet, le nombre de combinaisons possibles pour les valeurs des tensions augmente exponentiellement avec le nombre de sources. Rapidement, la quantité d'échantillons nécessaire à l'entraînement du réseau devient démesurée [104].

Pour considérer des onduleurs avec un plus grand nombre de sources, il faut faire recours aux métaheuristiques. Les métaheuristiques sont des algorithmes d'optimisation non déterministes qui proposent différentes stratégies de recherche basées sur l'amélioration itérative de solutions candidates. Dans le contexte de la minimisation des harmoniques, les métaheuristiques ont l'avantage de considérer n'importe quelles harmoniques et de rester efficaces même lorsque le nombre de sources augmente [52], [104]. Quelques exemples de métaheuristiques appliquées au problème de la minimisation des harmoniques sont l'algorithme de la colonie d'abeilles [13], l'algorithme de la compétition coloniale [99], l'optimisation par essaim de particules [105], l'algorithme génétique [106], l'algorithme de la mouche à feu [107] et l'algorithme mémétique [108]. Bien que la stratégie de recherche diffère d'une métaheuristique à l'autre, la formulation du problème est essentiellement la même et requiert la définition d'une fonction de coût qui permet de comparer la qualité des solutions candidates. La métrique utilisée n'a pas besoin d'être absolue, mais uniquement relative afin de pouvoir ordonnancer les solutions d'après leur qualité. La métaheuristique utilise ce classement pour modifier et améliorer les solutions candidates sur plusieurs itérations avant d'obtenir une solution finale quasi optimale. Nous présentons à l'équation suivante la fonction de coût définie à la référence [99] :

$$f_{Cost}(\vec{\theta}) = \left(100 * \frac{V_1^* - V_1}{V_1^*}\right)^4 + \sum_{h=3,5,7\ldots} \frac{1}{h} * \left(50 * \frac{V_h}{V_1}\right)^2 \tag{2.5}$$

où $\vec{\theta}$ représente la solution à évaluer, soit le vecteur des angles de commutation, V_1^* est l'amplitude désirée pour la composante fondamentale, V_1 est l'amplitude obtenue pour la composante fondamentale et V_h est l'amplitude de l'harmonique d'ordre h. Les calculs de ces termes ont été donnés à l'équation (2.4) pour le cas spécifique d'un onduleur à quatre sources. Comme nous le verrons au Chapitre 6, ces termes sont facilement généralisés pour n'importe quel nombre de sources. Dans la fonction de coût définie à l'équation (2.5), le premier terme, soit celui dont l'exposant est 4, pénalise sévèrement toute solution qui génère une onde en escalier dont l'amplitude de la composante fondamentale diffère de l'amplitude spécifiée par l'indice de modulation de plus de 1%. Le deuxième terme, soit la sommation, pénalise les solutions dont les harmoniques sont grandes. Cette somme peut être modifiée pour inclure n'importe quelles harmoniques.

Comparativement aux méthodes déterministes ou aux réseaux de neurones, les métaheuristiques offrent une grande flexibilité au niveau de l'objectif d'optimisation et permettent de minimiser efficacement les harmoniques d'onduleur avec un très grand nombre de sources. En effet, nous avons démontré dans [104] qu'il est possible de minimiser efficacement les 100 premières harmoniques d'un onduleur à 100 sources en utilisant l'algorithme génétique. Malgré leur supériorité par rapport

aux méthodes déterministes, les métaheuristiques ont toutefois deux lacunes importantes. Premièrement, elles nécessitent une puissance de calcul considérable et engendrent souvent un temps d'exécution trop long pour un contrôle en temps réel de l'onduleur. À titre d'exemple, les auteurs de [13] mesurent un temps d'exécution de 1000 secondes pour le GA et de 700 secondes pour l'algorithme de colonie d'abeilles pour le calcul des angles d'un onduleur à trois sources. Bien que les implémentations dans [13] soient programmées en MATLAB®, ces temps d'exécution semblent un peu trop longs pour être valides. Dans [104], nous avons implémenté un algorithme de GA en C++ pour la minimisation des harmoniques. Les temps d'exécution mesurés variaient de 1.2 secondes à 77 secondes selon le nombre de sources de tension et le nombre d'harmoniques considérées. Ces temps sont beaucoup plus courts que ceux publiés dans [13], mais restent quand même trop longs pour un contrôle adéquat. Deuxièmement, les métaheuristiques sont, par leur nature même, non déterministes. Ceci signifie que leur stratégie d'optimisation a un aspect stochastique et ne trouve pas nécessairement la même solution à chaque exécution. De même, la métaheuristique peut occasionnellement converger prématurément vers un minimum local et retourner une solution sous-optimale. Afin d'éviter ce problème, les auteurs de [77] suggèrent d'exécuter la métaheuristique à dix reprises avant de sélectionner la meilleure solution. Cette technique améliore la fiabilité de la méthode, mais allonge évidemment le temps de calcul et aggrave le problème précédent.

En résumé, les techniques publiées dans la littérature pour la minimisation des harmoniques d'un onduleur multiniveau sont soit limitées à un petit nombre de sources de tension dans le cas des méthodes déterministes ou nécessitent un temps de calcul trop long pour un contrôle efficace dans le cas des métaheuristiques. Dans cette thèse, nous proposons d'implémenter le PSO sur un processeur graphique afin d'exploiter l'architecture massivement parallèle du GPU et d'accélérer significativement le calcul des angles de commutation optimaux. Le temps d'exécution sera réduit d'un facteur de 453.7x comparé à une exécution séquentielle sur CPU. En utilisant une métaheuristique, notre solution considère des onduleurs avec un très grand nombre de sources et offre une flexibilité totale pour la sélection des harmoniques à minimiser.

2.5 Écoulement de puissance

La deuxième application considérée dans cette thèse est l'optimisation de l'écoulement de puissance (OPF de l'anglais *optimal power flow*). Les progrès dans les technologies de l'information et de la communication ont permis la modernisation du réseau électrique pour former le réseau intelligent. La surveillance et le contrôle en temps réel de la grille sont maintenant possibles et introduisent de nouveaux défis d'ingénierie en matière de contrôle. L'OPF est l'un des problèmes d'optimisation à la

base de la production et du transport de l'énergie électrique. Il consiste à trouver les réglages optimaux des générateurs et de l'équipement de transport de façon à maximiser une fonction objective pour le fonctionnement en régime permanent du réseau de transport d'électricité. Un exemple d'un tel réseau de transport se trouve à la Figure 2.7. Il s'agit du réseau test IEEE à 14 bus qui représente une partie du système d'alimentation électrique américain dans le Midwest des États-Unis en février 1962. Ce système comprend deux générateurs, trois compensateurs statiques d'énergie réactive, trois transformateurs et 11 charges. Le problème d'OPF a initialement été formulé par Carpentier en 1962 [109], mais a depuis évolué pour inclure des contraintes de sécurité, des variables de contrôle discrètes et des fonctions multi-objectives. À cet égard, l'OPF est un problème d'optimisation à grande échelle, non linéaire, non convexe et avec variables de contrôle continues et discrètes. De par sa grande complexité, le problème d'OPF demeure toujours un défi d'ingénierie significative [110].

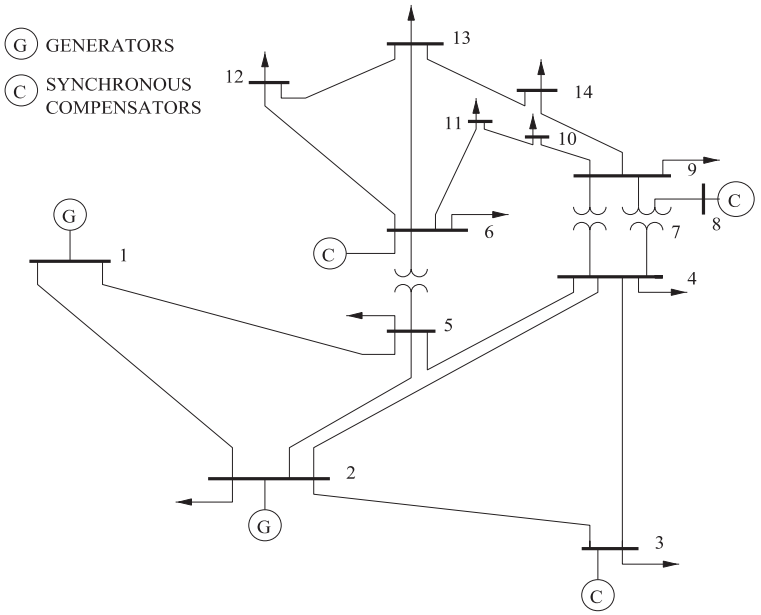


Figure 2.7: Réseau test IEEE à 14 bus (reproduit de [111])

Dans [112] et [113], Frank et coll. présentent un sommaire exhaustif des solutions actuelles au problème de l'OPF. Ils reconnaissent que les méthodes déterministes sont limitées à une optimisation locale et incapables de tenir compte des variables discrètes. Pour surmonter ces difficultés, les auteurs recommandent l'utilisation de métaheuristiques comme l'optimisation par essaim de particules [114], l'algorithme génétique [115] ou l'algorithme d'exploration bactérienne [110]. Les

métaheuristiques représentent une famille importante d'algorithmes d'optimisation non déterministes. Elles se basent sur l'amélioration itérative de solutions candidates pour résoudre des problèmes d'optimisation qui sont insolubles par les méthodes classiques. Dans le cas du problème d'OPF, chaque solution candidate représente un ensemble de paramètres de contrôle pour le réseau. Ces paramètres peuvent inclure le réglage des générateurs, le rapport des transformateurs et l'ajustement des compensateurs statiques d'énergie réactive. Pour évaluer la qualité d'une solution candidate, il est nécessaire d'effectuer une analyse d'écoulement de puissance (PF de l'anglais *power flow*) afin d'obtenir l'état du système en régime permanent associé aux paramètres de contrôle de la solution. L'utilisation d'une métaheuristique pour résoudre le problème d'OPF requiert donc le calcul du PF pour des centaines de solutions candidates sur plusieurs itérations. La quantité de calcul nécessaire est si grande que les techniques d'OPF basées sur les métaheuristiques sont souvent limitées à de petits réseaux d'où vient notre motivation pour une solution parallèle sur GPU.

Étant donné que l'analyse de PF est un élément commun à toutes les méthodes d'OPF basées sur les métaheuristiques, il est essentiel que la revue de la littérature présentée dans ce chapitre inclut non seulement les solutions au problème d'OPF, mais aussi celles pour l'analyse de PF sur GPU. Pour cette raison, nous divisons cette section en deux parties. Premièrement, nous présentons un sommaire des derniers travaux de recherche sur la parallélisation de l'analyse de PF sur GPU. Deuxièmement, nous enchaînons avec les solutions au problème de l'OPF.

2.5.1 Analyse d'écoulement de puissance

Le calcul parallèle est une approche efficace pour réduire le temps d'exécution d'une analyse de PF. Des implémentations parallèles ont été développées pour des systèmes multicœurs à mémoire partagée [116], des systèmes distribués [117] et des circuits intégrés spécialisés [118]. Cependant, ces solutions sont soit liées à un matériel spécifique, nécessitent une grande infrastructure informatique ou offrent une accélération limitée. Abordables et largement accessibles, les processeurs graphiques représentent une technologie émergente et une alternative intéressante pour les applications parallèles. Conçus avec plusieurs centaines voire même quelques milliers de cœurs intégrés sur une même puce, les GPU permettent un gain de performance important au niveau de la vitesse de calcul pour plusieurs applications scientifiques, y compris dans le domaine des systèmes de puissance. Pour nommer quelques exemples, des implémentations sur GPU de l'élimination Gauss-Jordan ainsi que de la factorisation LU sont proposées dans [119] afin d'accélérer l'analyse de réseaux de transport haute tension à courant continu. Les auteurs de [120] et [121] utilisent avec succès le GPU pour accélérer l'analyse électromagnétique de systèmes électriques par des facteurs respectifs de 70x et 107x. Le temps d'exécution de l'analyse est réduit davantage dans [122] par l'utilisation de matrices creuses. Les GPU sont aussi utilisés

pour la simulation de la stabilité transitoire de systèmes d'alimentation électrique à grande échelle dans [123], [124] et [125]. Une amélioration subséquente est proposée dans [126] par une implémentation sur système informatique multi-GPU. L'architecture parallèle du GPU est également prouvée avantageuse pour la simulation [127], [128] et l'analyse de puissance des circuits intégrés [129], [130]. Enfin, une expérience est menée dans [131] pour évaluer le gain de performance que pourrait apporter le GPU à un logiciel commercial d'analyse de systèmes d'alimentation électrique en déléguant les opérations vectorielles et matricielles au GPU. Encore ici, les résultats montrent une amélioration significative de la performance.

Spécifiquement pour l'analyse de PF d'un réseau de transport d'électricité, Ablakovis et Dzafic proposent dans [132] une implémentation parallèle sur GPU de l'algorithme progressif-rétrogressif (traduit de l'anglais *forward-backward algorithm*). Ils obtiennent une accélération de 1.58x, mais leur approche est limitée à l'analyse de réseaux avec topologie radiale. Dans [133], Singh et Arumi programment à l'aide de MATLAB® une implémentation parallèle sur GPU des algorithmes de Gauss-Seidel (G-S) et de N-R et mesurent un gain de performance de 3.27x pour le réseau test IEEE à 300 bus. Dans [134], Vilacha et coll. publient un algorithme Gauss-Jacobi parallèle pour l'analyse d'écoulement de puissance sur GPU d'un réseau synthétique à 65000 bus. Finalement, Guo et coll. comparent dans [135] des implémentations parallèles sur GPU de l'algorithme de G-S, de l'algorithme de N-R et de l'algorithme rapide découplé. Les accélérations obtenues sont de 0.06x, 1.75x et 1.30x respectivement.

Les implémentations sur GPU proposées par les auteurs de [132], [133], [134] et [135] offrent toutes un gain de performance en réduisant le temps nécessaire à l'analyse d'écoulement de puissance par rapport à des implémentations séquentielles équivalentes sur CPU. Cependant, elles utilisent des matrices pleines ce qui limite leur extensibilité à des réseaux plus grands. À titre d'exemple, l'algorithme de N-R publié dans [133] prend 10 ms, 313 ms et 4689 ms pour respectivement analyser les réseaux test IEEE à 30, 118 et 300 bus. Même si la parallélisation sur GPU permet de diminuer ces temps d'exécution, il est évident que la performance de l'algorithme est limitée par l'ordre de complexité engendré par l'utilisation de matrices pleines. En fait, cette complexité est d'ordre $O(n^2)$, mais peut être réduite à $O(n)$ en représentant les données sous forme de matrices creuses. Contrairement à la matrice pleine, la matrice creuse encode uniquement les éléments non nuls ce qui diminue la quantité de calculs nécessaire à l'analyse de PF. Pour cette raison, la plupart des logiciels récents d'analyse de PF utilisent des matrices creuses. Un exemple est le logiciel MATPOWER [136], un projet à code source ouvert programmé en MATLAB® pour la simulation de système de puissance. MATPOWER est accepté par la communauté scientifique comme étant un logiciel de référence en termes de facilité d'utilisation,

d'exactitude des résultats et de performance. Comparativement aux temps d'exécution publiés dans [133], MATPOWER nécessite uniquement 3 ms, 6 ms et 20 ms pour analyser respectivement les réseaux test IEEE à 30, 118 et 300 bus tels que mesurés dans [137]. Ces temps d'exécution sont beaucoup plus petits que ceux de la référence [133] et démontrent clairement que l'utilisation de matrices creuses est indispensable à la conception d'un programme performant d'analyse de PF, soit un des objectifs de cette thèse.

Malheureusement, le développement d'algorithmes parallèles sur GPU qui utilisent des matrices creuses est très difficile à cause de la structure irrégulière des données [138]. Ceci entraîne la divergence d'exécution entre les multiples threads et limite les accès parallèles à la mémoire. Malgré tout, différentes techniques de parallélisation ont été publiées dans la littérature et certains auteurs ont obtenu des résultats encourageants ce qui illustre le potentiel des GPU pour l'accélération de l'analyse de PF. Pour nommer quelques exemples, Gopal et coll. développent dans [139] une implémentation parallèle sur GPU de l'algorithme de Gauss-Jacobi pour l'analyse de PF et obtiennent une accélération maximale de 4.82x. Dans [140], Garcia publie un algorithme parallèle de gradient biconjugué sur GPU et mesure un gain de performance de 2.1x. X. Li et F. Li proposent dans [141] une parallélisation sur GPU de l'algorithme de gradient conjugué avec un préconditionneur de Chebyshev afin d'améliorer la convergence. Ils observent une accélération de 10.8x. Bien que les travaux publiés dans [139], [140] et [141] prétendent adresser le problème d'analyse de PF, ils sont tous limités à la solution du système d'équations linéaires et ne considèrent pas les autres étapes nécessaires à l'analyse de PF telles que la construction de la matrice d'admittance, la construction de la matrice Jacobienne ou le calcul de la puissance complexe injectée aux bus. En fait, il n'existe à ce jour aucune solution sur GPU pour l'analyse de PF qui utilise les matrices creuses et qui implémente toutes les étapes de l'algorithme.

Néanmoins, dans le cas de l'algorithme de N-R pour l'analyse de PF, la solution du système d'équations linéaires est une étape importante et une qui est particulièrement difficile à paralléliser. En plus des méthodes itératives discutées au paragraphe précédent, quelques travaux récents ont été complétés sur le développement de solveurs linéaires directs sur GPU. Entre autres, des implémentations parallèles sur GPU de la factorisation QR utilisant des matrices denses sont publiées dans [142] et [143]. Similairement, les références [144], [145] et [146] présentent des implémentations de la factorisation LU, aussi avec matrices denses. À cause de la difficulté liée à l'organisation irrégulière des données d'une matrice creusent, les implémentations parallèles sur GPU de la factorisation QR et LU utilisant des matrices creuses sont beaucoup plus rares, mais quand même disponibles dans [129], [130] et [147]. Il est toutefois important de noter que l'accélération offerte par la parallélisation de solveurs linéaires denses est typiquement plus grande que pour

celle de solveurs linéaires creux. Pour prendre avantage de l'accélération supérieure des solveurs denses dans le cadre d'une analyse transitoire, Jalili-Marandi et coll. démontrent dans [126] qu'il est possible de représenter la matrice jacobienne de la méthode de N-R sous forme d'une matrice bloc-diagonale. Cette représentation est possible grâce à la technique de relaxation instantanée proposée par les auteurs afin de diviser le système électrique en sous-systèmes indépendants avant de compléter l'analyse transitoire. La matrice résultante est creuse, ce qui réduit l'ordre de complexité de l'analyse, mais peut être résolue efficacement sur le GPU à l'aide d'un solveur dense en traitant concurremment les multiples blocs de la diagonale [148]. Malgré que l'approche de Jalili-Marandi et coll. soit très avantageuse en terme de performance, elle ne peut malheureusement pas être utilisée dans le contexte d'une analyse de PF étant donné que cette dernière, contrairement à l'analyse transitoire, nécessite que le réseau soit analysé en entier. Il est alors impossible d'obtenir une matrice jacobienne de pure forme bloc-diagonale. Cependant, tel que démontré par les auteurs de [149], il est faisable dans le cadre d'une analyse de PF d'utiliser une méthode de séparation des nœuds [150] pour transformer la matrice jacobienne en matrice creuse bloc-diagonale avec doubles bordures (BDDB). Cette représentation permet de calculer plus rapidement une solution au système d'équations linéaires. Des implémentations parallèles de solveurs BDDB ont été proposées pour les systèmes multicœurs à mémoire partagée [151], pour les systèmes distribués [152] et pour les FPGA [153], ce qui montre la possibilité d'une implémentation parallèle sur GPU.

En résumé, les méthodes actuelles pour l'analyse de PF parallèle sur GPU ont deux désavantages importants. Premièrement, plusieurs d'entre elles utilisent des matrices pleines. Celles-ci s'adaptent bien à l'architecture parallèle des GPU et offrent une bonne accélération. Cependant, l'utilisation de matrices pleines augmente arbitrairement l'ordre de complexité de l'analyse. Même après la parallélisation, leurs temps d'exécution restent beaucoup plus élevés que pour des implémentations séquentielles équivalentes avec matrices creuses. Deuxièmement, malgré que plusieurs travaux récents aient été effectués sur l'analyse de PF sur GPU à l'aide de matrices creuses, ceux-ci couvrent uniquement la solution du système d'équations linéaires et n'implémentent pas les autres étapes de l'analyse. Dans cette thèse, afin de combler ces lacunes, nous proposons des implémentations parallèles des algorithmes de G-S et de N-R pour l'analyse de PF. Nos implémentations utilisent uniquement des matrices creuses et parallélisent toutes les étapes de l'analyse sur un environnement hybride GPU-CPU. L'accélération obtenue est de 45.2x pour l'algorithme de G-S et 17.8x pour celui de N-R comparé à une implémentation séquentielle hautement optimisée sur CPU, qui elle aussi utilise que des matrices creuses. Les gains de performance apportés par nos implémentations sont bien supérieurs à ceux des travaux antérieurs publiés dans la littérature.

2.5.2 Optimisation de l'écoulement de puissance

Le problème d'OPF a été formulé par Carpentier en 1962 [109]. En raison de la non convexité introduite par les équations de l'écoulement de puissance, ce problème est très difficile à résoudre et reste encore aujourd'hui un sujet de recherche d'intérêt. Plusieurs méthodes déterministes et non déterministes ont été développées pour le problème d'OPF au fil des ans. Un sommaire exhaustif est présenté dans [112] et [113]. Deux familles de méthodes déterministes qui sont particulièrement intéressantes sont les méthodes de points intérieurs (PI) [154] et les méthodes de programmation semi-définie positive (SDP) [155]. Les méthodes de PI définissent des fonctions de barrière afin de considérer les contraintes d'inégalité intrinsèques au problème d'OPF et de limiter la recherche à l'intérieur du domaine des solutions faisables. À partir d'une solution connue, la direction de la recherche est calculée à chaque itération et la méthode converge vers l'optimum local le plus proche. Plusieurs logiciels renommés tels que MATPOWER [136] utilisent cette méthode; elle est bien établie, fiable et rapide. Toutefois, la solution trouvée par la technique de PI dépend du point initial choisi et n'est pas garantie d'être l'optimum global. Dans le cas de la famille des méthodes SDP, le problème d'OPF est reformulé suivant une relaxation convexe et peut alors être résolu dans un temps polynomial. La solution trouvée est garantie d'être globale et correcte si l'écart entre la formulation relaxée et celle du problème original est prouvé nul. C'est en fait le cas pour les réseaux tests IEEE à 30, 118 et 300 bus [156]. Cependant, un exemple simple d'un réseau à trois bus est donné dans [157] pour montrer que cet écart n'est pas toujours nul. Les derniers travaux de recherche dans le domaine visent à identifier les conditions pour lesquelles cet écart est nul [158], [159] ou à fournir des stratégies de mitigations lorsqu'il ne l'est pas [160].

Les méthodes déterministes sont conçues pour une formulation purement continue du problème de l'OPF et sont incapables de considérer nativement des variables de contrôle discrètes telles que le rapport des transformateurs ou le réglage des compensateurs statiques d'énergie réactive. Pour inclure ces variables discrètes, les méthodes déterministes doivent être jumelées à une heuristique comme la stratégie d'arrondissement numérique [161]. D'après cette technique, le problème d'OPF est d'abord résolu en considérant toutes les variables comme si elles étaient continues. Une fois qu'une solution est trouvée, les variables discrètes sont arrondies à leur valeur discrète la plus proche. Ces dernières sont ensuite gardées fixes et le problème d'OPF est solutionné une seconde fois afin de réajuster les variables continues. Spécialement pour le réglage des compensateurs statiques d'énergie réactive, l'opération d'arrondissement peut engendrer des changements de valeurs importants puisque le pas entre les valeurs discrètes possibles peut être aussi grand que 40 MVAR [162]. Pour atténuer l'impact causé par l'arrondissement de toutes les variables discrètes d'un seul coup, une approche progressive est proposée dans [163]

afin d'échelonner le processus sur plusieurs itérations. Chaque itération consiste à arrondir quelques variables choisies aléatoirement et à solutionner l'OPF pour ajuster les autres. Le problème est résolu lorsque toutes les variables discrètes ont été arrondies. Il est même suggéré dans [164] d'utiliser une fonction de probabilité lors du choix des variables à arrondir afin de sélectionner celles qui auront le plus petit impact possible sur les autres, soit celles qui ont déjà des valeurs très proches des valeurs discrètes permises. Dans [162] et [165], plusieurs combinaisons possibles pour les variables discrètes sont évaluées à chaque itération. Malgré tout, l'heuristique d'arrondissement peut choisir des valeurs discrètes qui mèneront à une solution sous-optimale ou même une solution infaisable lors du calcul de l'OPF pour l'ajustement des variables continues [162].

Plusieurs méthodes non déterministes ont aussi été développées pour résoudre le problème d'OPF [113], [166]. Celles-ci incluent entre autres les métaheuristiques qui ont l'avantage de pouvoir considérer nativement les variables discrètes, de permettre l'utilisation de fonctions objectives non différentiables et de prévenir la convergence prématurée vers un optimum local. Dans [114], Soliman et Mantawy proposent un algorithme de PSO pour minimiser les coûts de production du réseau test IEEE à 30 bus. Leur approche emploie la méthode rapide découplée d'analyse de PF pour calculer les variables dépendantes associées à chaque solution candidate avant d'évaluer la fonction objective. La solution calculée permet une économie importante au niveau des coûts de production. Toutefois, l'extensibilité de l'algorithme proposé n'est pas vérifiée sur des réseaux plus grands. Dans [115], Bakirtzis et coll. utilisent un GA pour optimiser le même réseau et observent eux aussi une diminution importante des coûts de production. Cependant, comme pour l'approche précédente, ils emploient la méthode rapide découplée d'analyse de PF ce qui introduit inévitablement des approximations lors de l'évaluation des solutions candidates. Dans [167], Sousa et Soares proposent un algorithme de recuit simulé pour l'OPF. Étant donné que le recuit simulé se base sur l'amélioration d'une seule solution candidate, cet algorithme est beaucoup plus rapide que ceux basés sur une population tels que le PSO ou le GA. Cependant, la qualité de la solution trouvée par les auteurs de [167] n'est pas aussi bonne que pour les deux algorithmes précédents. De plus, tout comme pour le PSO [114] et le GA [115], l'extensibilité de la méthode de recuit simulé proposée dans [167] est limitée à un réseau à 30 bus. Pour repousser cette limite et considérer des réseaux électriques plus grands, Amjady et coll. [110] suggèrent une version améliorée de l'algorithme d'exploration bactérienne et démontrent l'efficacité de leur méthode sur des réseaux de différentes grandeurs allant jusqu'à 118 bus. De même, Bouchekara et coll. prouvent dans [168] que l'algorithme d'optimisation enseignement-apprentissage est lui aussi efficace pour des réseaux allant jusqu'à 118 bus. Finalement, pour attaquer la complexité inhérente à des réseaux plus grands, Mahdad et Srairi [169] ont tout récemment proposé une approche à multiples phases où un algorithme d'évolution différentielle est utilisé à la première phase pour calculer

une solution au problème d'OPF. Un PSO est ensuite utilisé à la deuxième phase pour explorer l'espace de recherche autour de cette première solution afin de converger vers une meilleure solution. Enfin, la solution résultante est améliorée une dernière fois à la troisième phase par un algorithme de recherche de motifs. Cette approche sur trois phases augmente significativement la capacité d'exploration de l'algorithme et permet l'optimisation efficace du réseau test IEEE à 300 bus. Malgré son avantage, la méthode multi-phase de Mahdad et Srairi [169] nécessite une quantité énorme de calculs ce qui entraîne un temps d'exécution beaucoup trop long pour une optimisation en ligne. Heureusement, comme nous l'avons discuté précédemment à la section 2.2, il est possible de réduire le temps de calcul des métaheuristiques par la programmation parallèle. À ce jour, plusieurs implémentations parallèles de métaheuristiques appliquées au problème d'OPF existent pour systèmes multicœurs à mémoire partagée [170], [171] ou systèmes distribués [172], [173] et [174]. Cependant, il n'en existe aucune pour les processeurs graphiques.

Le seul exemple où un GPU est utilisé pour accélérer le calcul d'une solution au problème d'OPF est celui à la référence [175]. Dans cet article, Rakai et Rosehart utilisent la librairie *Parallel Computing Toolbox* de MATLAB® pour programmer la méthode de PI discutée précédemment sur un GPU. Leur implémentation parallèle est très rapide étant donné l'aspect déterministe de l'algorithme choisi et permet une accélération de 4x en exploitant l'architecture massivement parallèle du GPU. Toutefois, comme nous l'avons expliqué plus haut, la méthode de PI ne considère pas les variables discrètes et ne peut pas assurer l'optimalité de la solution trouvée puisque cette méthode converge obligatoirement vers l'optimum local le plus près.

En résumé, les solutions antérieures au problème de l'OPF ont deux désavantages importants. Tout d'abord, les méthodes déterministes ne peuvent considérer efficacement les variables discrètes telles que le rapport des transformateurs ou le réglage des compensateurs statiques d'énergie réactive. De plus, ces méthodes permettent uniquement une optimisation locale ou requièrent des approximations sous forme d'une formulation relaxée du problème. D'un autre côté, les métaheuristiques ont l'avantage de considérer les variables discrètes et d'offrir une meilleure résilience contre la convergence prématurée vers un optimum local. Cependant, ces algorithmes nécessitent une puissance de calculs considérable résultant en un temps d'exécution trop long pour une optimisation en ligne. Dans le but de mitiger cette lacune, nous proposons dans cette thèse d'utiliser le cadriciel *gpuMF* pour implémenter un PSO parallèle sur GPU afin de résoudre plus rapidement le problème d'OPF. L'évaluation des solutions candidates s'effectue à l'aide d'une analyse de PF utilisant la méthode de N-R. Similairement à l'approche proposée dans [169], notre algorithme exécute en trois phases pour améliorer l'exploration de l'espace de recherche. L'approche que nous proposons trouve des solutions de meilleure qualité que les références étudiées

tout en offrant une accélération de 17.2x comparée à une exécution séquentielle sur CPU.

2.6 Reconfiguration optimale des réseaux de distribution

La troisième et dernière application considérée dans cette thèse est la reconfiguration optimale des réseaux de distribution électriques (DFR de l'anglais *distribution feeder reconfiguration*). Les réseaux de distribution sont généralement structurés suivant une topologie radiale, mais incluent également des commutateurs de liaison supplémentaires pour permettre la reconfiguration en cas d'entretien planifié ou de pannes inattendues. Un exemple d'un tel réseau est illustré à la Figure 2.8.

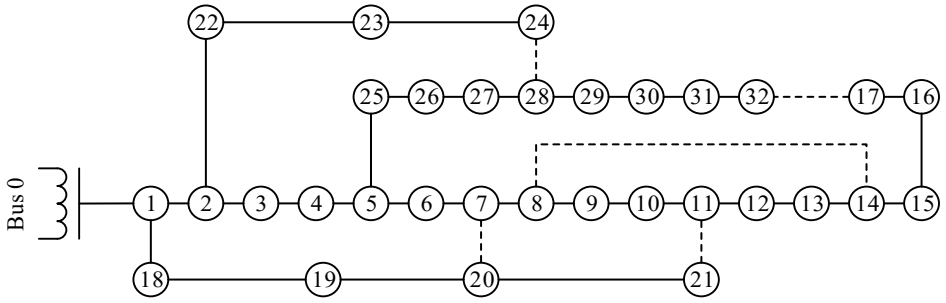


Figure 2.8: Réseau de distribution à 33 bus

Il s'agit d'un réseau test à 33 bus créé par Baran et Wu en 1989 [176] et depuis utilisé à maintes reprises par d'autres auteurs. Sur ce schéma, les sommets représentent les bus et les arêtes, les lignes de distribution. Celles-ci sont équipées de commutateurs et peuvent être ouvertes ou fermées. Les arêtes pointillées identifient les lignes ouvertes tandis que les arêtes pleines désignent les lignes fermées. Le bus 0 est le bus d'alimentation, soit le point d'intersection entre le réseau de transport discuté à la section précédente et le réseau de distribution discuté ici. On remarque aussi que le réseau est arrangé suivant une topologie radiale. Il est possible de le reconfigurer en ouvrant et fermant des lignes, mais la contrainte sur la topologie radiale doit être respectée. Avec la mise en œuvre du réseau intelligent et le développement de commutateurs haute tension ultras rapides, il est faisable d'automatiser cette reconfiguration. Le système automatique peut alors réagir plus rapidement aux imprévus ou même considérer la fluctuation de la demande en énergie lors de sa reconfiguration afin de toujours opérer dans une topologie optimale, minimisant ainsi les pertes de transport et les coûts d'opération. Cette automatisation nécessite cependant le développement d'algorithmes d'optimisation hautement efficaces pouvant calculer la reconfiguration optimale dans les plus brefs délais. Il s'agit en fait d'un problème d'optimisation combinatoire à grande échelle, non linéaire

et non convexe. Dans cette section, nous présentons un sommaire de techniques actuelles publiées dans la littérature pour le problème de DFR.

Les solutions antérieures au problème de DFR sont souvent divisées en trois catégories : les heuristiques, les méthodes conventionnelles et les métaheuristiques [177]. Les heuristiques sont des techniques d'optimisation appliquées et développées à partir de l'expérience accumulée dans l'opération des réseaux de distribution. Elles incluent la méthode d'échange de branches [178] et celle de la segmentation des boucles [179]. Dans la méthode d'échange de branches, la recherche débute à partir d'une topologie faisable et une paire de branches ouvertes et fermées est sélectionnée pour l'échange. Cet échange n'a lieu que si la topologie résultante est radiale et avantageuse comparée à la configuration actuelle. L'optimisation se termine lorsqu'aucune amélioration n'est possible. Dans la méthode de segmentation des boucles, toutes les branches sont initialement fermées. Le graphe maillé résultant est analysé afin d'identifier les boucles fondamentales. L'algorithme ouvre ensuite une branche par boucle pour finalement obtenir une structure radiale. Tant pour la méthode d'échange de branches que pour celle de la segmentation des boucles, l'opération effectuée à chaque itération est déterministe et choisie de façon à obtenir le plus grand gain. Ces deux techniques convergent nécessairement vers l'optimum local le plus proche et ont le désavantage de ne pas pouvoir garantir une optimisation globale.

De leur côté, les méthodes conventionnelles appliquées au problème de DFR incluent la programmation linéaire en nombres entiers (MILP de l'anglais *mixed integer linear programming*) [180], la programmation conique en nombres entiers (MICP de l'anglais *mixed integer conic programming*) [180] et la programmation quadratique en nombres entiers (MIQP de l'anglais *mixed integer quadratic programming*) [181], [182]. Suivant ces trois techniques, le problème non linéaire de DFR est simplifié et reformulé de sorte qu'il devienne résoluble par des méthodes d'optimisation classiques. Contrairement aux heuristiques, les méthodes conventionnelles ne nécessitent pas de solutions initiales faisables et garantissent la convergence vers l'optimum global. Ces deux avantages viennent toutefois au prix d'un temps d'exécution très long et la solution trouvée n'est optimale que d'après la formulation simplifiée du problème. À titre d'exemple, les algorithmes publiés dans [181] et [182] se basent sur la méthode de MIQP et nécessitent respectivement 3192 et 1134 secondes pour reconfigurer un réseau de distribution de 880 bus. Pour éviter la formulation simplifiée associée aux méthodes conventionnelles, il est aussi possible d'employer une méthode d'optimisation non linéaire. Dans [183], une décomposition de Benders est utilisée pour réduire la complexité du problème et le diviser en deux sous-problèmes. Ceux-ci sont ensuite résolus à l'aide d'un outil commercial d'optimisation non linéaire. Bien que les résultats présentés sont encourageants, le

solveur non linéaire ne peut garantir une solution optimale dans un délai raisonnable [182].

Enfin, les métaheuristiques sont des algorithmes d'optimisation non déterministes capables de résoudre une multitude de problèmes d'ingénierie intangibles par les méthodes classiques. Ces algorithmes nécessitent très peu d'information sur le problème et agissent en améliorant des solutions candidates sur plusieurs itérations afin d'obtenir une solution finale quasi optimale. Plusieurs métaheuristiques ont été utilisées pour le problème de DFR telles que l'optimisation par essaims de particules [184], l'algorithme génétique [185], [186], l'optimisation par colonie de fourmis [187], l'algorithme du système immunitaire artificiel [188], l'algorithme de la colonie d'abeilles [189] ou l'algorithme d'impérialisme [190]. Un sommaire exhaustif des métaheuristiques appliquées au problème de DFR est disponible dans [191]. Les métaheuristiques ont l'avantage d'être polyvalentes, de pouvoir échapper aux optima locaux et de pouvoir considérer des fonctions objectives non différentiables. D'un autre côté, elles sont particulièrement difficiles à appliquer au problème de DFR étant donné que leurs solutions candidates ne peuvent être entièrement aléatoires, mais doivent satisfaire la contrainte sur la topologie radiale du réseau pour être valides. Faire respecter cette contrainte représente un défi important lors du développement d'une méthode de DFR basée sur les métaheuristiques. Différentes techniques ont été proposées.

Une première approche permettant d'intégrer la contrainte sur la topologie radiale du réseau est proposée dans [192]. Elle consiste à exécuter la métaheuristique sans se soucier de la topologie du réseau, mais de définir une fonction objective qui pénalise sévèrement les solutions qui ne sont pas radiales. Bien que cette stratégie fonctionne pour des petits réseaux, elle est inefficace pour les plus grands étant donné que la probabilité de générer aléatoirement des topologies radiales est extrêmement minime. Une deuxième approche consiste à débuter la recherche à l'aide de solutions qui sont déjà radiales et d'accepter les modifications suggérées par la métaheuristique uniquement lorsque celles-ci respectent la contrainte sur la topologie [188]. Le désavantage est que l'optimisation ne peut être globale puisque la recherche est limitée à l'espace faisable autour des solutions initiales. Une troisième approche est de recourir à la méthode de segmentation des boucles, mais d'utiliser la métaheuristique pour choisir la branche à ouvrir à l'intérieur de chaque boucle ou même l'ordre dans lequel les boucles sont visitées [187], [190], [193]. Comparativement à une méthode de segmentation des boucles purement déterministe, la métaheuristique améliore la capacité de recherche de l'algorithme, mais son efficacité est quand même limitée à de petits réseaux. Finalement, une quatrième approche consiste à développer des opérateurs complexes qui permettent de réparer les solutions modifiées par la métaheuristique afin de retrouver une topologie radiale [186], [194]. Cette opération assure la faisabilité de la solution résultante, mais nécessite une analyse laborieuse du

réseau après chaque modification afin d'identifier l'apparition de boucles. Similairement, des opérateurs complexes qui permettent de modifier les solutions tout en assurant leur topologie radiale sont proposées dans [185] et [195]. Ces opérateurs sont toutefois limités au voisinage de la solution actuelle et limitent l'exploration de l'espace de recherche. De plus, en raison de leur aspect stochastique et du grand nombre de solutions candidates et d'itérations nécessaires, les métaheuristiques appliquées au problème de DFR demandent une puissance de calcul considérable. Ceci résulte en un temps d'exécution trop long pour une reconfiguration automatique du réseau de distribution. Par exemple, le GA proposé dans [186] nécessite un peu plus de trois jours pour optimiser un réseau de distribution de 83 bus sur un processeur 3 GHz. Pour cette raison, les métaheuristiques sont habituellement limitées à la reconfiguration de réseaux de distribution de petite ou moyenne envergure. En fait, au cours de notre revue de la littérature, le plus grand réseau trouvé qui a été reconfiguré par une métaheuristique n'avait que 476 bus [188].

En résumé, les solutions antérieures au problème de DFR ont plusieurs lacunes. Dans le cas des heuristiques, elles sont limitées à une optimisation locale. Dans le cas des méthodes conventionnelles, elles sont moins précises puisqu'elle nécessite une formulation simplifiée du problème original. De plus, leur temps d'exécution peut être considérable dépendamment de la grandeur du réseau. Finalement, dans le cas des métaheuristiques, la contrainte sur la topologie radiale du réseau représente une difficulté importante, complique l'implémentation et réduit significativement la capacité d'exploration de l'algorithme. Finalement, les métaheuristiques souffrent elles aussi d'un trop long temps d'exécution. Pour remédier à ces lacunes, nous proposons dans cette thèse d'utiliser le cadriciel *gpuMF* pour implémenter un algorithme génétique parallèle sur GPU pour la reconfiguration optimale des réseaux de distribution. En exploitant l'architecture massivement parallèle des processeurs graphiques, le temps d'exécution du programme est réduit d'un facteur de 66.2x permettant ainsi une reconfiguration extrêmement rapide. De plus, une nouvelle approche basée sur la théorie des graphes est proposée pour encoder les solutions candidates de la métaheuristique. La représentation développée maintient la topologie radiale et augmente grandement l'efficacité de l'algorithme d'optimisation permettant de résoudre des réseaux de 4400 bus, soit cinq fois plus grands que pour les méthodes antérieures les plus performantes publiées dans la littérature.

2.7 Conclusion

Dans ce chapitre, nous avons présenté une revue de la littérature de plusieurs articles de références intéressants qui touchent le sujet de cette thèse. Nous avons d'abord discuté de deux objectifs de la parallélisation et identifié que le résultat désiré est souvent une combinaison des deux, soit augmenter la quantité de calculs effectués

tout en minimisant le temps d'exécution. Nous avons ensuite discuté de quatre techniques de parallélisation pour les métaheuristiques. Des exemples d'implémentations ont été donnés pour chacune. Ce sont le modèle à grains fins et le modèle hybride qui sont les mieux adaptés à l'architecture parallèle des GPU. Plusieurs cadriciels pour faciliter l'utilisation de métaheuristiques ont ensuite été présentés. Ces cadriciels offrent des implémentations complètes pour plusieurs métaheuristiques, mais aussi la structure et le cadre de travail nécessaire au développement facile d'autres métaheuristiques. Nous avons souligné qu'il n'existe à ce jour aucun cadriciel qui permet une implémentation parallèle complète d'une métaheuristique sur GPU. Finalement, nous avons présenté un sommaire des solutions actuelles aux trois problèmes d'optimisation du domaine des réseaux intelligents qui sont considérés dans cette thèse, soit la minimisation des harmoniques d'un onduleur multiniveau, l'optimisation de l'écoulement de puissance et la reconfiguration des réseaux de distribution. Pour chaque application, nous avons couvert les méthodes déterministes ainsi que les métaheuristiques. Dans le cas de la minimisation des harmoniques, nous avons conclu que les méthodes déterministes sont limitées à un petit nombre de sources de tension et permettent uniquement d'éliminer les harmoniques de plus bas niveaux. De leur côté, les métaheuristiques restent efficaces pour un très grand nombre de sources et offrent une flexibilité totale au niveau de l'objectif d'optimisation. Dans le cas de l'optimisation de l'écoulement de puissance, contrairement aux métaheuristiques, les méthodes déterministes ne peuvent considérer efficacement les variables discrètes et sont limitées à une optimisation locale. Enfin, dans le cas de la reconfiguration des réseaux de distribution, les méthodes déterministes nécessitent une formulation simplifiée du problème. La solution trouvée n'est optimale que d'après le modèle simplifié. De leur côté, les métaheuristiques ont la capacité d'optimiser directement des solutions au problème combinatoire non linéaire original.

D'après ces observations, il est évident que les métaheuristiques ont un avantage clair face aux méthodes déterministes pour chacune des trois applications étudiées. Cependant, en raison de leur fonctionnement basé sur l'amélioration itérative de solutions candidates, les métaheuristiques demandent un temps d'exécution souvent trop long pour une optimisation en ligne. L'objectif de cette thèse adresse directement cette lacune en proposant le cadriciel *gpuMF* pour l'implémentation de métaheuristiques parallèles sur GPU. En exploitant l'architecture massivement parallèle des GPU, le cadriciel *gpuMF* permet de calculer des solutions de meilleures qualités pour des problèmes de plus grandes dimensions dans des temps de calcul plus petits que les techniques antérieures publiées dans la littérature.

Chapitre 3
Métaheuristiques

Plusieurs problèmes d'optimisation en ingénierie sont trop complexes pour que nous puissions calculer une solution optimale. En pratique, cette optimalité n'est pas essentielle au fonctionnement du système et nous nous satisfaisons de trouver une solution faisable de bonne qualité. Cette solution peut être calculée par une heuristique ou une métaheuristique. Les heuristiques sont des algorithmes développés pour des applications spécifiques. Leur stratégie de recherche est basée sur notre expérience dans le domaine en question. De leur côté, les métaheuristiques sont des algorithmes génériques qui peuvent s'appliquer à une multitude de problèmes d'optimisation. Dans ce chapitre, nous traitons spécifiquement des métaheuristiques. Nous commençons par une formulation mathématique des problèmes d'optimisation. Nous enchaînons avec la théorie de la complexité afin d'identifier les types de problèmes qui sont plus facilement résolus par les métaheuristiques que par les méthodes classiques. Nous présentons ensuite les concepts communs à toutes les métaheuristiques et poursuivons avec une description détaillée de l'algorithme d'optimisation par essaim de particules (PSO de l'anglais *particle swarm optimization*) et l'algorithme génétique (GA de l'anglais *genetic algorithm*). Nous avons choisi de décrire ces deux algorithmes puisqu'ils sont parmi les métaheuristiques les plus utilisées en ingénierie [196]. Le PSO est un algorithme du domaine de l'intelligence distribuée tandis que le GA est une méthode de calcul évolutif. Dans cette thèse, nous implémentons ces deux métaheuristiques dans le cadriciel proposé au Chapitre 5 et nous les utilisons pour l'optimisation des réseaux intelligents aux Chapitres 6, 7 et 8. Finalement, nous terminons ce chapitre en présentant deux techniques possibles pour le développement de métaheuristiques hybrides.

3.1 Problème d'optimisation

Un problème d'optimisation consiste à trouver la meilleure solution parmi les solutions faisables. Les problèmes d'optimisation peuvent être continus ou discrets, avec ou sans contrainte, à objectif simple ou multiple et déterministes ou stochastiques. Un problème d'optimisation peut être défini mathématiquement sous la forme standard suivante :

$$\underset{\vec{x}}{\text{minimiser}} \quad f_i(\vec{x}), \qquad i = 1, \dots, I \tag{3.1}$$

$$\text{sujet à} \quad g_j(\vec{x}) = 0, \qquad j = 1, \dots, J \tag{3.2}$$

$$h_k(\vec{x}) \leq 0, \qquad k = 1, \dots, K \tag{3.3}$$

où $\vec{x}$ est le vecteur solution défini comme $\vec{x} = (x_1, x_2, \dots x_n)$, n est la dimension du problème, $f_i(\vec{x})$ sont les fonctions objectives, $g_j(\vec{x})$ sont les contraintes d'égalités et $h_k(\vec{x})$ sont les contraintes d'inégalités. Le problème est à objectif unique lorsque I est égal à un ou à objectifs multiples lorsque I est plus grand que un. Similairement, le problème peut être sans contrainte si J et K sont égalent à zéro ou avec contraintes si J ou K est plus grand que zéro.

Un problème est de type linéaire avec contrainte si $f(\vec{x})$, $g(\vec{x})$ et $k(\vec{x})$ sont linéaires. Pour résoudre un problème de ce type, il est possible d'utiliser la méthode du simplexe [196]. Cette méthode d'optimisation classique permet de trouver efficacement le minimum global de la fonction objective tout en respectant les contraintes. D'autre part, si f n'est pas linéaire, mais qu'il est possible de calculer sa dérivée f', une méthode de gradient tel que celle de Newton [196] peut être utilisée. Dans le cas où la dérivée f' n'existe pas, on peut alors recourir à la méthode de Nelder-Mead [196]. Celle-ci emploie un polytope dont les $n + 1$ sommets sont déplacés itérativement vers l'optimum local. Toutefois, si la fonction objective f est multimodale ou non convexe, la méthode de Nelder-Mead est limitée à une optimisation locale et la solution trouvée dépend du point de départ de la recherche. Il existe en fait plusieurs autres types de problèmes d'optimisation et chaque type demande des techniques différentes. Comme on s'en doute bien, les problèmes d'optimisation n'ont pas tous la même complexité. Celle-ci est généralement liée à la dimension du problème et à ses aspects linéaires ou non linéaires, convexes ou non convexes et continus ou discrets. Lorsque la difficulté d'un problème d'optimisation est trop grande, les méthodes classiques sont inefficaces et il faut recourir aux métaheuristiques.

3.2 Théorie de la complexité

La théorie de la complexité est un domaine de l'informatique théorique qui analyse formellement la quantité de ressources nécessaires pour résoudre un problème à l'aide d'un algorithme. Les ressources sont définies en termes de temps, soit le nombre d'étapes, et d'espace, le nombre d'éléments gardés en mémoire, requis par l'algorithme pour solutionner un problème de dimension n. La théorie de la complexité s'applique principalement aux problèmes décisionnels, ceux dont la réponse est oui ou non. Cependant, elle permet aussi d'analyser les problèmes d'optimisation puisque ceux-ci peuvent toujours être réduits à un problème

décisionnel. Par exemple, un problème d'optimisation qui consiste à minimiser la fonction $f(\vec{x})$ peut être réduit à vérifier l'existence d'une solution $\vec{x}_i$ qui donne une valeur de $f(\vec{x}_i)$ plus petite qu'un seuil spécifié. Telle que présenté dans [56], la théorie de la complexité divise les problèmes décisionnels en quatre classes fondamentales : la classe P, la classe NP, la classe NP-complet et la classe NP-difficile.

3.2.1 Classe P

Un problème est dans la classe P s'il peut être résolu dans un temps polynomial à l'aide d'une machine déterministe de Turing. Cette machine est un modèle abstrait d'un appareil de calcul qui a un ruban infini, une tête pour lire et écrire sur ce ruban, un registre d'état et une table d'actions qui dicte les déplacements de la tête et les écritures à effectuer d'après l'entrée lue et l'état interne de l'appareil. La machine est dite déterministe puisqu'une action précise est définie pour chaque combinaison possible de la valeur lue et de l'état interne. Si la machine rencontre une valeur et un état pour lesquels aucune action n'est définie, elle s'arrête. Le modèle de la machine déterministe de Turing est facile à comprendre puisqu'il décrit les ordinateurs que nous utilisons dans la vie de tous les jours. Un exemple d'un problème de la classe P est celui du calcul du chemin le plus court reliant deux points d'un graphe. Ce problème est illustré à la Figure 3.1. Sous une forme décisionnelle, il peut être énoncé comme suit : « Étant donné le graphe suivant, existe-t-il un chemin qui relie le point de départ au point d'arrivée et dont la longueur est plus petite qu'une constante d ? » Pour répondre à cette question, on pourrait utiliser l'algorithme de Dijkstra [197] pour calculer le chemin le plus court et de vérifier que la longueur de ce chemin est plus petite que d. Étant donné que l'algorithme de Dijkstra a une complexité de $O(n^2)$, ce problème est effectivement résoluble dans un temps polynomial par une machine déterministe de Turing.

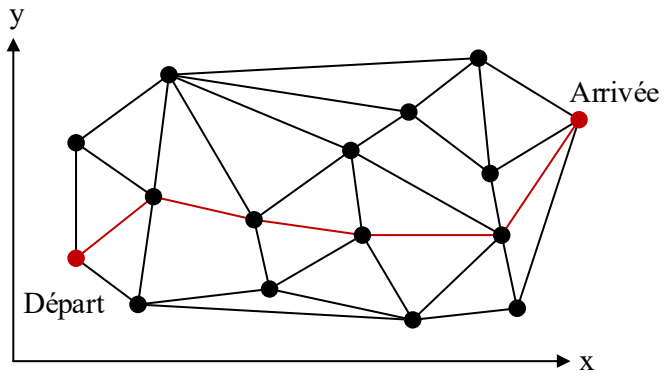


Figure 3.1: Problème du chemin le plus court reliant deux points d'un graphe

3.2.2 Classe NP

Un problème est dans la classe NP s'il peut être résolu dans un temps polynomial par une machine non déterministe de Turing. Contrairement à la machine déterministe qui se déplace vers un état précis à chaque itération, la machine non déterministe peut se déplacer dans plusieurs états à la fois. De façon pratique, on peut considérer la machine non déterministe comme un ordinateur avec un niveau de parallélisme infini. Étant donné qu'il n'existe pas de tel ordinateur, un problème NP ne peut être résolu dans un temps polynomial, cependant une solution au problème peut quant à elle être vérifiée dans un temps polynomial. Un exemple d'un problème de la classe NP est celui du commis voyageur illustré à la Figure 3.2. Ce problème consiste à calculer le cycle hamiltonien le plus court qui permet de visiter une seule fois chacun des points du graphe. Sous une forme décisionnelle, il peut s'énoncer comme suit : « Étant donné le graphe suivant, existe-t-il un cycle hamiltonien de longueur plus petite qu'une constante *d* ? » En réalité, il n'y a aucun algorithme connu capable de calculer le cycle minimal dans un temps polynomial [56]. Toutefois, si l'on nous donne une solution candidate, soit un chemin valide, il est possible de vérifier dans un temps polynomial si la longueur totale de ce chemin est plus petite qu'une constante *d*.

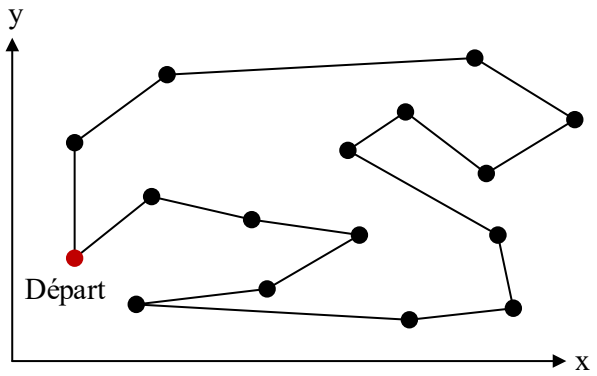


Figure 3.2: Problème du commis voyageur

3.2.3 Classe NP-complet

Un problème X est dans la classe NP-complet s'il est NP et si l'on peut réduire n'importe quel autre problème Y de la classe NP à ce problème X dans un temps polynomial. En d'autres mots, nous pouvons résoudre rapidement le problème Y si nous connaissons comment résoudre le problème X. Le problème du commis voyageur dans sa forme décisionnelle est un problème NP, mais aussi NP-complet [56].

3.2.4 Classe NP-difficile

Finalement, la classe NP-difficile contient les problèmes qui sont au moins aussi difficiles que le plus difficile des problèmes NP. Puisque tous les problèmes NP-complets peuvent être réduits à un autre problème NP-complet dans un temps polynomial, ils sont aussi NP-difficile. Cependant, l'inverse n'est pas vrai : les problèmes NP-difficiles ne sont pas nécessairement NP-complets. Un exemple est le problème de l'arrêt qui consiste à déterminer si un programme informatique quelconque finira par s'arrêter ou non. Ce problème idéaliste est en fait indécidable puisqu'il est tout simplement impossible de concevoir un algorithme qui permettrait d'analyser n'importe quel programme informatique qui existe déjà ou qui sera développé dans le futur. La classe NP-difficile inclut aussi tous les problèmes d'optimisation dont la formulation décisionnelle est NP-complète [56]. Prenons par exemple le problème commis voyageur formulé sous forme de problème d'optimisation dont l'objectif est de calculer le cycle hamiltonien de longueur minimale. Même si l'on nous donne un cycle valide, il n'est pas possible de vérifier l'optimalité de la solution dans un temps polynomial puisqu'il faudrait connaître la longueur de la solution optimale qui est évidemment inconnue.

La relation entre les quatre classes de complexité est illustrée à la Figure 3.3. Sur ce diagramme, on remarque que les problèmes de la classe P sont les moins complexes et que ceux de la classe NP-difficile sont les plus complexes. La classe P fait partie de la classe NP puisqu'un problème résoluble dans un temps polynomial sur une machine déterministe est nécessairement résoluble dans un temps égal ou plus petit sur une machine non déterministe. La question à savoir si P égale NP est d'une grande importance et reste encore à ce jour sans réponse. Cette question demande si tous les problèmes dont la solution peut être rapidement vérifiée par un ordinateur peuvent également être rapidement résolus par un ordinateur. Pour démontrer l'égalité, il faudrait trouver un algorithme qui permette de résoudre un des problèmes NP-complet dans un temps polynomial. Si cela était possible, tous les problèmes NP seraient aussi résolubles dans un temps polynomial puisqu'ils sont tous réductibles à n'importe lequel des problèmes NP-complets dans un temps polynomial.

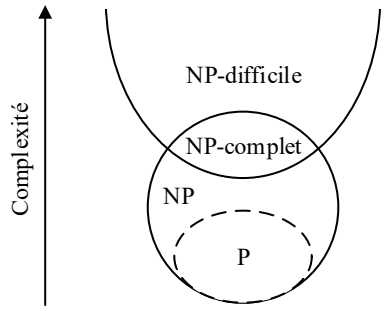


Figure 3.3: Relation entre les classes de complexité

3.3 Quand utiliser les métaheuristiques?

Les métaheuristiques sont des algorithmes d'optimisation génériques qui s'appliquent à presque n'importe quel problème d'optimisation. Pour cette raison, elles sont quelquefois utilisées à tort lorsque des méthodes classiques pourraient effectuer le même travail de façon beaucoup plus efficace. C'est le cas par exemple pour la plupart des problèmes de la classe P. Les méthodes classiques permettent de calculer des solutions exactes à ces problèmes dans un temps polynomial. À moins que la dimension du problème soit trop grande et que le temps d'exécution soit trop long, il est préférable d'utiliser une méthode classique. Dans le cas des classes NP, NP-complet et NP-difficile, les méthodes classiques sont souvent incapables de calculer des solutions dans des temps acceptables, il faut alors recourir aux métaheuristiques. Une autre situation où les méthodes classiques sont inadéquates est lorsque le modèle du problème considéré n'est pas analytique comme pour plusieurs problèmes des domaines de la logistique, des finances, des télécommunications et de la biologie computationnelle [56]. Dans ce cas, il faut utiliser les métaheuristiques qui offrent une bien plus grande flexibilité quant à la définition de la fonction objective. Finalement, les métaheuristiques sont souvent préférables aux autres méthodes lorsque le problème considéré inclut une certaine incertitude ou lorsqu'une optimisation robuste est nécessaire.

Dans le cadre de cette thèse, chacune des trois applications du domaine des réseaux électriques intelligents que nous considérons motive l'utilisation des métaheuristiques. Dans la première application, la minimisation des harmoniques d'un onduleur multiniveau est un problème d'optimisation non linéaire, non convexe et avec contrainte sur l'ordre des angles de commutation. Comme nous le verrons au Chapitre 6, il existe un algorithme déterministe qui puisse résoudre ce problème dans un temps polynomial. Cet algorithme impose toutefois une limite sur l'indice de modulation et minimise obligatoirement toutes les harmoniques de la tension de sortie. Les métaheuristiques offrent quant à elles une flexibilité complète sur le choix des

harmoniques à minimiser et ne limitent aucunement l'indice de modulation. Dans la deuxième application, l'optimisation de l'écoulement de puissance dans les réseaux de transport d'électricité est un problème non linéaire, non convexe et avec contraintes. De plus, à cause de la coexistence de variables continues et discrètes, ce problème est considéré NP-difficile [198], [199], [200] ce qui justifie l'utilisation des métaheuristiques. Finalement, dans la troisième application, la reconfiguration des réseaux de distribution est un problème d'optimisation combinatoire non linéaire, non convexe, à variables discrètes et avec contraintes. Ce problème est aussi NP-difficile [181], [185] et [201]. Comme il est expliqué dans [202], il peut être réduit au problème de l'arbre quadratique couvrant de poids minimal qui est formellement prouvé NP-complet. L'utilisation d'une métaheuristique pour résoudre ce problème de reconfiguration est encore une fois justifiée.

3.4 Concepts communs

Afin de mieux comprendre ce qu'est une métaheuristique, nous présentons dans cette section leur fonctionnement ainsi que cinq caractéristiques importantes et communes aux algorithmes de cette famille. Après cette description générique, nous étudions plus précisément les algorithmes du PSO et du GA aux sections 3.5 et 3.6.

3.4.1 Fonctionnement des métaheuristiques

Les métaheuristiques sont divisées en deux catégories, celles basées sur une seule solution (*métaheuristiques-S*) et celles basées sur une population de solutions (*métaheuristiques-P*). Ces deux familles ont des caractéristiques complémentaires. Les *métaheuristiques-S* intensifient la recherche dans une région locale tandis que les *métaheuristiques-P* encouragent l'exploration globale de l'espace de recherche. Le fonctionnement pour ces deux familles est toutefois le même. Celui-ci est illustré à l'aide d'un diagramme de flux à la Figure 3.4. Ce diagramme est spécifique aux *métaheuristiques-P* puisque le terme « solutions » est employé au pluriel. Pour une *métaheuristique-S*, il faudrait simplement l'employer au singulier. L'algorithme débute par l'initialisation des solutions candidates. Celle-ci peut être aléatoire ou suivant une approche appliquée au problème considéré. L'important est que la population générée soit diversifiée. Les solutions sont ensuite décodées avant d'être évaluées par la fonction d'aptitude. Cette fonction mesure la qualité de chaque solution et permet à la métaheuristique de guider la recherche en modifiant les solutions candidates suivant la stratégie définie par la métaheuristique. Ces trois opérations sont exécutées sur plusieurs itérations jusqu'à ce que le critère de terminaison soit satisfait. Pour terminer, l'algorithme retourne la meilleure solution visitée.

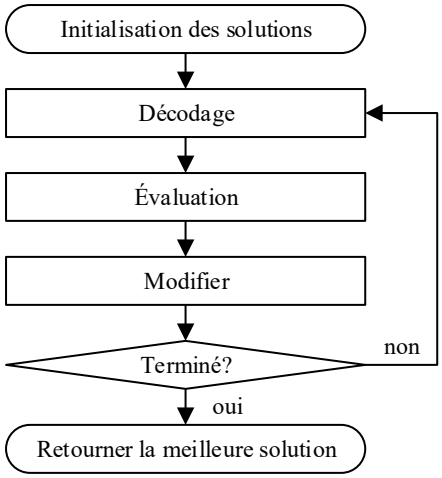


Figure 3.4: Digramme de flux des métaheuristiques

3.4.2 Représentation des solutions

Appliquer une métaheuristique à un problème d'optimisation demande certaines décisions de conception. D'abord, il est nécessaire de choisir l'encodage utilisé pour représenter les solutions candidates. Cet encodage doit être complet, continu et efficace. Être complet signifie que toutes les solutions aux problèmes peuvent être représentées par l'encodage. Être continu signifie qu'un chemin de recherche existe entre n'importe quelles deux solutions quelconques possibles. En d'autres mots, il faut que la représentation permette d'atteindre l'optimum global à partir de n'importe quelle solution initiale. Finalement, être efficace signifie que la représentation peut être manipulée facilement par les opérateurs de la métaheuristique en ayant la plus petite complexité possible. Les représentations les plus courantes sont un vecteur de nombre binaire, un vecteur de valeurs discrètes, un vecteur de nombres réels et une permutation d'une séquence donnée [56].

3.4.3 Fonction objective

La fonction objective représente l'objectif de l'optimisation. Elle permet à la métaheuristique d'évaluer la qualité des solutions candidates et de diriger la recherche vers des solutions de meilleure qualité. Une fonction objective peut être autosuffisante ou indicative. Elle est autosuffisante lorsqu'elle exprime directement l'objectif de la formulation mathématique du problème. C'est souvent le cas pour des problèmes d'optimisation continue sans contrainte. Une fonction objective est indicative lorsqu'elle n'exprime pas directement l'objectif, mais guide quand même la recherche vers des solutions qui satisfont mieux le problème. Un exemple de fonction indicative est lorsque les contraintes du problème sont exprimées sous forme de pénalités et

intégrés à la fonction objective. Puisque cette fonction indicative ne représente pas directement l'objectif d'optimisation, on l'appelle plutôt la fonction d'aptitude; elle permet d'évaluer l'aptitude des solutions candidates.

3.4.4 Considération des contraintes

Une solution candidate est dite faisable lorsqu'elle respecte les contraintes du problème et infaisable lorsqu'elle ne les respecte pas. Cinq stratégies communes pour intégrer les contraintes à la métaheuristique sont le rejet, la réparation, la préservation, le décodage et la pénalité. La stratégie du rejet consiste à rejeter toute solution qui ne satisfait pas aux contraintes. Cette stratégie fonctionne bien lorsque l'espace des solutions faisables est large et que seules quelques solutions infaisables sont rejetées. À l'opposé, la stratégie de réparation ne rejette pas les solutions infaisables, mais utilise des opérateurs qui permettent de les modifier afin de les rendre faisables. La stratégie de préservation consiste quant à elle à implémenter la métaheuristique de façon à ce qu'elle maintienne la faisabilité des solutions lors de leur modification. La stratégie de décodage définit plutôt une représentation qui assure la faisabilité des solutions. Autrement dit, l'encodage utilisé ne permet pas de représenter des solutions infaisables. Finalement, la stratégie de pénalité assigne une pénalité aux solutions infaisables lors de l'évaluation de la fonction d'aptitude. Les solutions infaisables sont donc incluses dans le processus d'optimisation, mais la valeur de leur aptitude est moindre que les solutions faisables.

3.4.5 Critère de terminaison

Le critère de terminaison est une autre caractéristique commune à toutes les métaheuristiques. Il définit la condition d'arrêt de l'algorithme. Dépendamment de l'application, le critère de terminaison peut être défini par un nombre fixe d'itérations, un temps fixe d'exécution, une valeur spécifiée pour la qualité de la solution finale ou lorsque la qualité de la meilleure solution cesse de s'améliorer. Cette dernière condition permet d'arrêter la recherche lorsque la métaheuristique a convergé.

3.4.6 Recherche non déterministe

Finalement, une dernière caractéristique commune aux métaheuristiques est l'aspect stochastique de leur recherche. À chaque itération, les métaheuristiques modifient les solutions candidates de façon à diriger la recherche vers de meilleures solutions, mais incluent aussi un facteur aléatoire qui permet d'explorer de nouvelles solutions. À titre d'exemple, le PSO introduit des vecteurs de nombres aléatoires lors du calcul des vitesses des particules afin de simuler une vibration stochastique. Dans le cas du GA, cet aspect aléatoire est implémenté par les opérateurs de sélection, d'enjambement et de mutation. À cause de cet aspect aléatoire, on qualifie les

métaheuristiques d'algorithmes non déterministes. Elles ne produisent pas nécessairement la même solution chaque fois qu'on les exécute.

3.5 Optimisation par essaim de particules

La première métaheuristique considérée dans cette thèse est l'optimisation par essaim de particules (PSO de l'anglais *particle swarm optimization*). Le PSO est un algorithme de recherche non déterministe basé sur une population et a été développé par Kennedy et Eberhard en 1995 [49]. L'algorithme simule le mouvement d'un essaim de particules dans un espace de recherche à une ou plusieurs dimensions vers une position optimale. La position de chaque particule représente une solution candidate et est initialisée aléatoirement. À chaque étape du procédé itératif, la vitesse des particules est calculée individuellement basée sur la vitesse précédente (inertie), la meilleure position jamais occupée par la particule (influence personnelle) et la meilleure position jamais occupée par n'importe quelle particule de l'essaim (influence sociale). Tel que défini dans [203], les équations permettant de mettre à jour la vitesse et la position d'une particule à l'itération t sont comme suit :

$$\vec{v}_{t+1} = \omega\vec{v}_t + c_1\vec{r}_1.*\left(\vec{b}_t - \vec{x}_t\right) + c_2\vec{r}_2.*\left(\vec{g}_t - \vec{x}_t\right) \tag{3.4}$$

$$\vec{x}_{t+1} = \vec{x}_t + \vec{v}_{t+1} \tag{3.5}$$

où $\vec{v}$ est la vitesse d'une particule; $\vec{x}$ est sa position; $\vec{b}$ la meilleure position jamais occupée par la particule; $\vec{g}$ la meilleure position jamais occupée par n'importe quelle particule de l'essaim; $\vec{r}_1$ et $\vec{r}_2$ sont des vecteurs de valeurs aléatoires entre 0 et 1; et ω, c_1 et c_2 sont respectivement les paramètres d'inertie, d'influence personnelle et d'influence sociale. Toujours d'après [203], le pseudocode de l'optimisation par essaim de particules est donné à l'algorithme 3.1. Tel qu'initialement proposé par Kennedy et Eberhard [49], l'algorithme que nous présentons ici utilise une approche synchrone où la meilleure position atteinte pas l'essaim est mise à jour une seule fois par itération et est ensuite utilisée pour ajuster la vitesse de chaque particule.

Algorithme 3.1: Optimisation par essaim de particules (PSO)

1: *Initialiser le compteur d'itération* $t = 0$
2: *Initialiser aléatoirement la position* $\vec{x}_{i,t}$ *de chaque particule* i *de l'essaim*
3: **tant que** *critère de terminaison n'est pas atteint* **{**
4: *Calculer le coût* $f(\vec{x}_{i,t})$ *de chaque particule*
5: *Mettre à jour la meilleure position* $\vec{b}_{i,t}$ *visitée par chaque particule*
6: *Mettre à jour la meilleure position* $\vec{g}_t$ *visitée par l'essaim*
7: *Calculer la vitesse* $\vec{v}_{i,t+1}$ *de chaque particule*
8: *Calculer la nouvelle position* $\vec{x}_{i,t+1}$ *de chaque particule*
9: *Incrémenter le compteur d'itérations* $t = t + 1$
10: **}**
11: **Retourner** *la meilleure position* $\vec{g}_t$ *atteinte par l'essaim*

Depuis son apparition, le PSO a été utilisé avec succès dans plusieurs domaines tels que l'ingénierie, l'économie, la médecine, la sécurité, etc. Sa popularité vient du fait qu'il est simple à implémenter, facilement adaptable à des problèmes d'optimisation continue non linéaire et a une complexité moindre que la plupart des autres métaheuristiques basées sur les populations comme l'algorithme génétique [204]. Malgré que le PSO original soit une méthode très compétitive, plusieurs optimisations ou variantes ont été proposées au cours des dernières années afin d'améliorer sa performance.

Une première variante du PSO est l'ajout d'une restriction sur l'amplitude de la vitesse des particules. Cette limite est proposée dans [205] et prévient une explosion possible de l'essaim. La limite est imposée après le calcul de la vitesse à la ligne 7 de l'algorithme 3.1 et est appliquée indépendamment à chaque dimension du vecteur de vitesse d'après les équations ci-dessous :

$$v_{t+1} = \begin{cases} v_{max}, & si \;\; v_{t+1} > v_{max} \\ -v_{max}, & si \;\; v_{t+1} < -v_{max} \end{cases} \tag{3.6}$$

$$v_{max} = \frac{limite_{supérieure} \; - \; limite_{inférieure}}{k} \tag{3.7}$$

où v_{t+1} est la vitesse de la particule au temps $t + 1$; v_{max} est la vitesse maximale et se calcule d'après les limites de l'espace de recherche pour chaque dimension; et k est une constante.

Une seconde optimisation consiste à diminuer l'inertie ω de la particule de façon linéaire tout au long de l'algorithme. Proposée par Shi et Eberhart [206], cette opération résulte en une grande inertie en début d'exécution permettant une meilleure exploration de l'espace de recherche et une petite inertie en fin d'exécution de façon

à améliorer le raffinement des solutions et la convergence vers un optimum. L'indice d'inertie variable se calcule comme suit :

$$\omega = \omega_{max} - \frac{t}{t_{max}}(\omega_{max} - \omega_{min}) \tag{3.8}$$

où ω_{max} et ω_{min} sont respectivement les valeurs maximale et minimale de l'indice d'inertie, t est l'itération présente et t_{max} est le nombre total d'itérations. Similairement, les auteurs de [207] ont récemment proposé une réduction exponentielle de l'indice d'inertie tandis que ceux de [208] suggèrent une technique qui tient compte de la distance entre chaque particule et la meilleure position de l'essaim.

Finalement, nous croyons que la contribution la plus importante apportée au PSO soit l'ajout d'un facteur de constriction pour obtenir une variante appelée le PSO canonique. Proposé par Clerc et Kennedy et supporté par une preuve théorique [209], ce facteur de constriction assure la convergence de la recherche sans avoir à recourir à d'autres techniques. L'équation (3.4) devient alors :

$$\vec{v}_{t+1} = \chi\left[\vec{v}_t + c_1\vec{r}_1.*\left(\vec{b}_t - \vec{x}_t\right) + c_2\vec{r}_2.*\left(\vec{g}_t - \vec{x}_t\right)\right] \tag{3.9}$$

$$\text{avec} \quad \chi = \frac{2k}{\left|2 - \phi - \sqrt{\phi^2 - 4\phi}\right|} \quad \text{pour} \quad \phi > 4 \quad \text{et} \quad \phi = c_1 + c_2 \tag{3.10}$$

où χ est le facteur de constriction. Les deux auteurs suggèrent d'utiliser des valeurs de coefficients de $\chi = 0.729$ et $c_1 = c_2 = 2.05$ afin d'obtenir un meilleur résultat. Le PSO canonique est de nos jours accepté comme étant l'algorithme PSO contemporain par défaut [205].

3.6 Algorithme génétique

La deuxième métaheuristique considérée dans cette thèse est l'algorithme génétique. Le GA est une méthode d'optimisation non déterministe basée sur une population et a été développé par John Holland dans les années 60 et publié pour la première fois en 1975 [210]. Basé sur la théorie de l'évolution de Darwin, le GA simule l'évolution d'une population de solutions pour l'optimisation d'un problème. Tout comme les organismes vivants qui s'adaptent à leur environnement, les solutions de le GA s'adaptent à la fonction de coût au cours d'un procédé itératif qui simule la sélection, l'enjambement et les mutations. Le GA peut être utilisé pour des problèmes d'optimisation, d'ordonnancement et de routage.

En nous basant sur [211], nous présentons le pseudocode du GA à l'algorithme 3.2. Au début, une population de solutions aléatoires est générée. Ces

solutions sont appelées chromosomes afin de faire le lien avec l'évolution des espèces vivantes. Même si le GA peut être utilisé pour l'optimisation de problèmes continus, dans sa version originale, il utilise un encodage binaire. À chaque itération, des chromosomes sont sélectionnés parmi la population pour ensuite être croisés. Cette sélection est stochastique, mais favorise les solutions de meilleure qualité. Chaque opération d'enjambement génère un nouveau chromosome qui fera partie de la nouvelle génération. Le processus d'enjambement doit créer autant d'enfants qu'il y a de parents afin que le nombre de solutions reste constant tout au long du processus évolutif. Certains chromosomes de la nouvelle génération subissent ensuite une mutation qui consiste à modifier un ou plusieurs bits du chromosome. Les chromosomes mutés sont choisis au hasard suivant le paramètre du taux de mutation. Finalement, les chromosomes parents sont remplacés par les enfants et le processus est répété pour plusieurs centaines ou milliers de générations. La solution retournée par le GA n'est pas nécessairement le meilleur chromosome à la dernière génération, mais le meilleur rencontré au cours de l'évolution. Les opérations d'enjambement et de mutations sont illustrées à la Figure 3.5.

Algorithme 3.2: Algorithme génétique (GA)

Initialiser le compteur d'itération $t = 0$
Initialiser aléatoirement les chromosomes $\vec{x}_{i,t}$
tant que *critère de terminaison n'est pas atteint* **{**
    Calculer le coût $f(\vec{x}_{i,t})$ *de chaque chromosome* i
    Sélectionner les parents pour l'enjambement
    Générer les enfants $\vec{x}_{i,t+1}$ *par enjambement*
    Muter les enfants suivant le taux de mutation
    Remplacer la génération des parents $\vec{x}_{i,t}$ *par celle des enfants* $\vec{x}_{i,t+1}$
    $t = t + 1$
}
Retourner *le chromosome* $\vec{x}_{i,t}$ *ayant le plus petit coût* $f(\vec{x}_{i,t})$

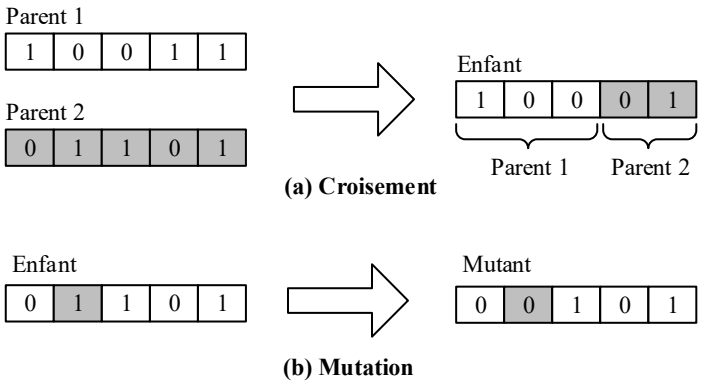


Figure 3.5: Opérations (a) d'enjambement et (b) de mutation pour le GA avec encodage binaire

Le GA utilise quatre concepts qui le différencient des autres métaheuristiques : la stratégie de sélection, la stratégie de reproduction, les stratégies de mutation et la stratégie de remplacement. Nous utilisons ici le mot stratégie puisqu'il existe différentes approches pour l'implémentation de ces opérations. Évidemment, la performance de chaque technique est intimement liée au problème d'optimisation en question et dépend de l'encodage utilisé pour représenter les solutions. En effet, les recherches récentes dans le domaine du GA reposent souvent sur l'adaptation de l'algorithme à un problème d'optimisation spécifique d'intérêt.

Les processus de sélection et de reproduction permettent l'amélioration des solutions et donc la convergence vers un optimum. La sélection favorise un individu proportionnellement à sa qualité ou à son rang. La seconde option est plus complexe puisqu'elle nécessite d'ordonner les solutions à chaque itération, mais permet d'éviter un biais lorsque la qualité des solutions n'est pas normalisée. Les techniques de sélection couramment utilisées sont la roue de fortune, l'échantillonnage universel stochastique et le tournoi [211]. La reproduction permet de combiner deux solutions et favorise l'échange d'information entre les solutions d'aptitude supérieure. Des méthodes de reproduction acceptées sont l'enjambement en un seul point, l'enjambement en plusieurs points ou l'enjambement arithmétique [211].

De leur côté, les processus de mutation et de remplacement encouragent l'exploration de l'espace de recherche. En modifiant aléatoirement un aspect d'une solution, la mutation permet de s'éloigner de la population et de visiter des solutions différentes. Quant à lui, le remplacement prévient une convergence trop rapide vers une solution supérieure trouvée en début de recherche.

Malgré que le GA date des années 1970, il est encore aujourd'hui considéré comme l'une des méthodes d'optimisation non linéaires les plus performantes [211] et suscite l'intérêt de plusieurs scientifiques. Par exemple, les auteurs de [212]

présentent une étude détaillée des différentes techniques de sélection et proposent un GA qui s'ajuste d'après la distribution de la qualité des solutions offrant une meilleure résistance à la convergence hâtive vers un optimum local. Les auteurs de [213] et [214] proposent d'utiliser le concept d'élitisme lors du remplacement afin de conserver les meilleurs chromosomes d'une génération à l'autre permettant une convergence accrue et de meilleurs résultats dans les problèmes étudiés. Finalement les auteurs de [215] suggèrent d'utiliser une hiérarchie à deux niveaux ou un GA de haut niveau opère dans l'espace de recherche globale et utilise un GA de bas niveau pour raffiner le meilleur chromosome à chaque génération.

3.7 Métaheuristiques hybrides

Chaque métaheuristique présente des atouts spécifiques et son efficacité varie d'un problème à l'autre. C'est pourquoi il est important de bien choisir l'algorithme et de l'adapter à l'application en question. De manière à développer des méthodes d'optimisation plus robustes face à une large gamme de problèmes, certains travaux récents dans le domaine cherchent à développer des algorithmes hybrides qui exploitent les forces de plusieurs métaheuristiques. Une première approche consiste à utiliser des concepts spécifiques à une métaheuristique à l'intérieur d'une autre. C'est ce que font les auteurs de [216] en ajoutant des opérateurs génétiques de sélection, d'enjambement et de mutation à un algorithme de PSO. Similairement, Singh et Ghosh [217] combinent le GA et le recuit simulé. Au lieu de remplacer systématiquement la génération des parents par celle des enfants, ils utilisent le critère d'acceptation du SA afin de minimiser les chances de remplacer une bonne solution par une moins bonne en fin d'exécution. Pour les problèmes étudiés, leur algorithme hybride démontre une convergence plus rapide et offre une meilleure performance. Il faut cependant faire attention, en modifiant le comportement de la métaheuristique, la meilleure convergence observée est habituellement limitée à un ou quelques problèmes précis au détriment d'un moins bon rendement sur une gamme plus large de problèmes [218]. Une méthode d'hybridation que nous jugeons préférable consiste à exécuter différentes métaheuristiques en parallèle tout en permettant leur collaboration par un processus de migration des solutions. On parle alors d'un modèle en îlots parce que chaque algorithme optimise sa propre population de solutions candidates. Puisque cette approche ne modifie pas les métaheuristiques utilisées, leurs propriétés d'exploration et de convergence sont conservées. Les métaheuristiques hybrides coopératives suivant un modèle en îlots ont été utilisées avec succès dans [43], [219] et [220] où une amélioration de la solution finale a été observée pour plusieurs problèmes tests. Le seul désavantage à utiliser des métaheuristiques hybrides coopératives est l'augmentation du temps de calcul. Nous croyons cependant qu'une implémentation parallèle puisse contrecarrer ce problème.

3.8 Conclusion

Dans ce chapitre, nous avons discuté des métaheuristiques. Nous avons commencé par la formulation mathématique des problèmes d'optimisation. Nous avons ensuite expliqué à l'aide de la théorie de la complexité que certains problèmes d'optimisation sont plus difficiles que d'autres à résoudre. Ce sont pour les problèmes intraitables par les méthodes classiques qu'il est avantageux et nécessaire d'utiliser les métaheuristiques. Pour mieux comprendre cette famille d'algorithmes d'optimisation non déterministes, nous avons discuté de leur fonctionnement et de cinq caractéristiques communes. Nous avons ensuite présenté en détail deux métaheuristiques importantes : le PSO et le GA. Ces deux méthodes ont été choisies puisqu'elles sont parmi les plus utilisées en ingénierie et parce que nous les implémentons aux chapitres suivants pour l'optimisation des réseaux électriques intelligents. Finalement, nous avons terminé en touchant aux métaheuristiques hybrides qui représentent une nouvelle direction de recherche dans le domaine de l'optimisation moderne. Les métaheuristiques hybrides exploitent les forces de différents algorithmes afin de créer des méthodes d'optimisation plus robuste. Dans le cadre de cette thèse, il est important d'introduire ces algorithmes hybrides puisque le cadriciel pour métaheuristiques parallèles sur GPU que nous proposons au Chapitre 5 implémente cette approche.

Chapitre 4
Processeurs graphiques

4.1 Introduction

Comme nous l'avons vu au chapitre précédent, les métaheuristiques sont des algorithmes d'optimisation génériques utilisés pour résoudre une multitude de problèmes d'ingénierie difficilement solubles par les méthodes classiques. Les métaheuristiques utilisent différentes stratégies stochastiques pour modifier une ou plusieurs solutions candidates jusqu'à l'obtention d'un résultat quasi optimal. Spécialement pour celles basées sur une population de solutions candidates, les métaheuristiques démontrent un très grand niveau de parallélisme et peuvent profiter d'une accélération considérable lorsqu'elles sont implémentées sur des processeurs parallèles.

Dans ce chapitre, nous présentons une famille de processeurs parallèles bien particulière, soient les processeurs graphiques. Conçus avec des centaines voire quelques milliers de cœurs, on considère les GPU comme étant des systèmes massivement parallèles et on les qualifie de processeurs *manycore*. Pour l'implémentation de métaheuristiques parallèles, les GPU ont l'avantage d'exploiter un niveau de parallélisme bien supérieur aux processeurs multicœurs et offrent donc une meilleure accélération, soit un avantage significatif pour la résolution de plusieurs problèmes d'optimisation dans le domaine des systèmes de puissance.

Ce chapitre est divisé comme suit. Nous présentons d'abord une brève histoire des processeurs graphiques. Par la suite, nous décrivons leur architecture, leur modèle de programmation et leur modèle de la mémoire. Nous continuons ensuite avec différentes stratégies de programmation essentielles pour bien adapter un algorithme parallèle aux particularités de l'architecture du GPU afin d'obtenir la meilleure performance possible. Pour terminer, nous présentons une série de primitives parallèles qui seront utilisées aux chapitres ultérieurs pour l'implémentation des différents algorithmes sur GPU.

Afin d'obtenir la meilleure performance possible, les travaux expérimentaux présentés dans cette thèse sont exécutés sur trois processeurs graphiques différents, soit les cartes NVIDIA® GTX 750 Ti [221], GTX Titan [222] et Tesla K20 [223]. La première fait partie de la famille d'architectures Maxwell tandis que les deux autres sont de la famille Kepler. Bien qu'il existe certaines différences entre les deux familles, les principes de base et les stratégies de programmation restent les mêmes.

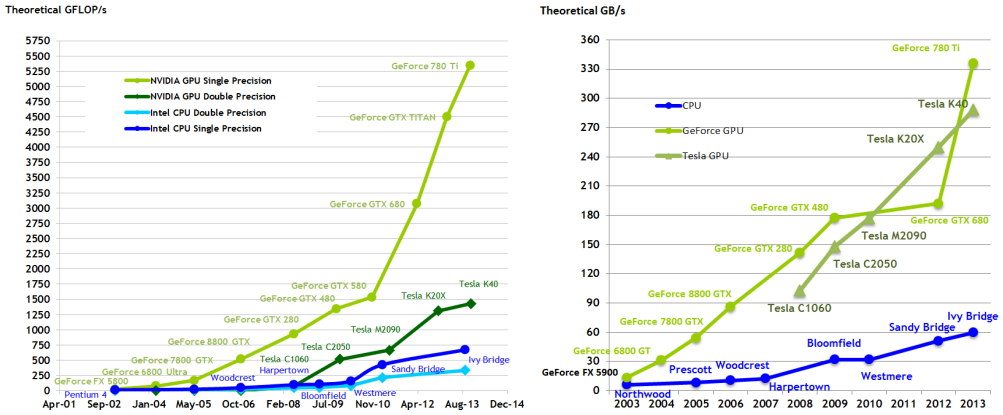
Sans perte de généralité, ce chapitre se limite donc à la description de l'architecture Maxwell, soit la plus récente des deux. L'étude de cette architecture est suffisante pour donner au lecteur une bonne compréhension des processeurs graphiques NVIDIA® ou même des processeurs graphiques en général.

4.2 Histoire des processeurs graphiques

Depuis les années 70, les avancements dans l'industrie des semi-conducteurs ont permis une augmentation continuelle de la fréquence d'exécution des microprocesseurs et par conséquent, de leur puissance de calcul. Nous nous sommes alors habitués à voir le temps d'exécution d'un programme séquentiel diminuer avec l'avènement de nouveaux processeurs. Or, cette tendance se termine aux alentours de 2003 lorsque la fréquence d'horloge des CPU atteint un plafond à environ 4 GHz à cause d'une trop grande consommation d'énergie et dissipation de chaleur. D'un autre côté, la loi de Moore [224] qui dit que le nombre de transistors sur une puce double environ tous les deux ans continue d'être observée. Les fabricants utilisent cette abondance de transistors et continuent sans cesse de sophistiquer les CPU pour inclure toute sorte d'optimisations telles que l'exécution dans le désordre, l'*hyperthreading*, la prédiction des branchements, l'exécution spéculative et les opérations vectorielles qui exploitent le parallélisme au niveau des instructions. Ces optimisations ont permis d'améliorer la performance des programmes séquentiels au cours des dernières années, mais les opportunités sont de plus en plus restreintes [225]. Il devient donc évident que la seule manière d'augmenter la puissance de calcul des CPU est de développer une architecture parallèle avec plusieurs cœurs.

Depuis 2003, on remarque que deux approches différentes ont été suivies par l'industrie quant au développement des processeurs parallèles [226]. Il y a l'approche *multicœur* qui vise à augmenter le nombre de cœurs sans affecter la vitesse d'exécution des programmes séquentiels. D'après cette architecture, chaque cœur est grandement sophistiqué de façon à implémenter les optimisations discutées précédemment et utilise une très grande quantité de transistors limitant ainsi le nombre de cœurs possible sur un même CPU. Un exemple de processeurs multicœurs est le Intel Xeon E5-2650 qui inclut 8 cœurs fonctionnant à 2.00 GHz [227]. Ce processeur est utilisé comme référence dans les travaux expérimentaux présentés dans cette thèse. De l'autre côté, l'approche *manycore* vise à intégrer le plus grand nombre de cœurs possible sur le même processeur maximisant ainsi la puissance de calculs des programmes parallèles à défaut d'une puissance réduite pour les programmes séquentiels. La simplicité de chaque cœur permet d'en intégrer plusieurs centaines sur la même puce comme dans le cas du processeur graphique NVIDIA® GTX Titan qui contient 2 688 cœurs [222]. Comme on le voit à la Figure 4.1, ces processeurs massivement parallèles offrent une puissance de calculs et une bande passante à la

mémoire bien supérieures à celles des CPU multicœurs ce qui en fait des plateformes de choix pour les applications à calculs intensifs. Il n'est pas surprenant que le superordinateur actuellement à la tête du Top500 [228], soit le Titan du Laboratoire national d'Oak Ridge, Tennessee, utilise 18 688 processeurs graphiques NVIDIA®. Il est tout aussi intéressant de constater qu'une seule carte graphique NVIDIA® GTX Titan offre une puissance de calcul de 4.494 Tflops, ce qui est 1.4x supérieur au ASCI Red, le superordinateur qui était à la tête du Top500 en 2000 [228].



(a) **(b)**

Figure 4.1: Comparaisons de (a) la puissance de calcul et (b) la bande passante entre les CPU et les GPU.

L'avantage des GPU pour les applications graphiques est indéniable, mais comment ces processeurs peuvent-ils aussi être utilisés pour accélérer des calculs scientifiques? Pour répondre à cette question, il faut se référer à [229] et revisiter l'histoire des processeurs graphiques. Jusqu'en 2001, les cartes graphiques suivent une architecture pipeline non programmable où l'image est générée par des circuits numériques massivement parallèles. En 2001, NVIDIA® lance les cartes GeForce 3 et remplace certains stages du pipeline par un grand nombre d'unités programmables fonctionnant en parallèle. En 2003, avec l'arrivée des GPU GeForce FX, ces unités programmables supportent aussi des opérations à virgule flottante de 32 bits. Initialement programmables que pour des applications graphiques, depuis la sortie du compilateur CUDA™ C/C++ en 2006 [230], les cartes NVIDIA® peuvent maintenant être programmées pour effectuer des calculs scientifiques. Elles supportent les opérations à virgule flottante de 64 bits et offrent au programmeur une architecture massivement parallèle sur une seule puce, ce qui est tout à fait révolutionnaire dans le domaine du calcul de haute performance.

4.3 Architecture

L'architecture typique d'un processeur graphique NVIDIA® est illustrée à la Figure 4.2. Ces processeurs sont composés d'un grand nombre de cœurs appelés les *CUDA™ cores* qui sont regroupés pour former les multiprocesseurs. Dans le cas de l'architecture Maxwell, on réfère à ces groupes par le nom de *Maxwell Streaming Multiprocessors* (SMM). Un GPU Maxwell peut contenir jusqu'à 24 SMM, chacun composé de 128 *CUDA™ cores* pour un total de 3072 cœurs. Chaque cœur comprend une unité arithmétique logique pour nombre entier et une unité de calcul à virgule flottante qui respecte la précision spécifiée par le standard IEEE-754 [231]. Le processeur Maxwell inclut une mémoire cache L2 entièrement cohérente d'une grosseur maximale de 2 Mo qui est partagée par les 24 multiprocesseurs. L'interface à la mémoire globale (hors puce) a une largeur de 384 bits et permet de communiquer avec une mémoire vive GDDR5 d'une grosseur maximale de 12 Go.

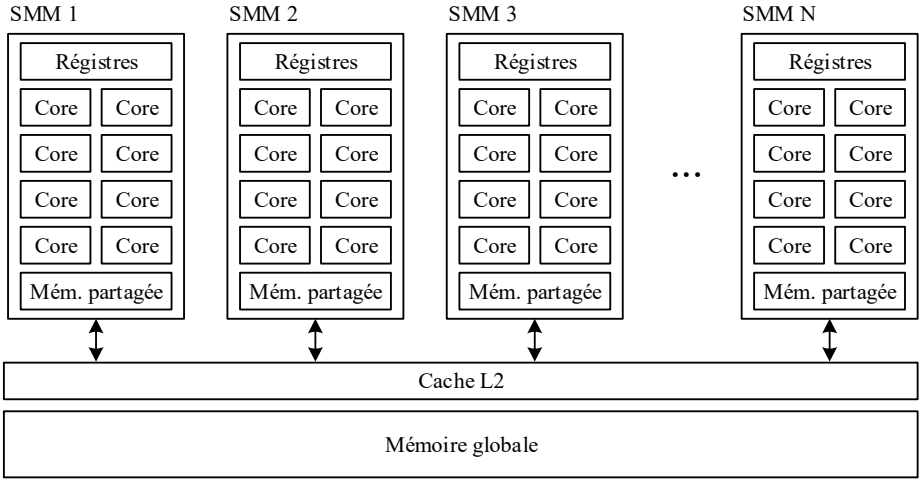


Figure 4.2: Architecture typique d'un processeur graphique

La structure interne d'un multiprocesseur SMM est présentée à la Figure 4.3. Les détails de cette architecture sont importants puisqu'ils définissent le modèle d'exécution d'un programme CUDA™. À l'intérieur du SMM, les 128 cœurs sont divisés en quatre blocs d'exécution. Chaque bloc comprend entre autres 32 cœurs, huit unités de chargement/sauvegarde (L/S de l'anglais *load/store*) pour le calcul des adresses et huit unités de fonctions spéciales pour les fonctions transcendantes. Malgré le nombre de cœurs, le bloc d'exécution est limité à un tampon d'instructions, un ordonnanceur de *warps* et deux unités d'envoi d'instructions. Un *warp* est défini comme étant un ensemble logique de 32 fils d'exécution ou threads CUDA™. D'après cette architecture, chaque bloc peut exécuter deux instructions sur 32 threads simultanément. On parle ici d'un modèle d'exécution à « instruction unique, threads

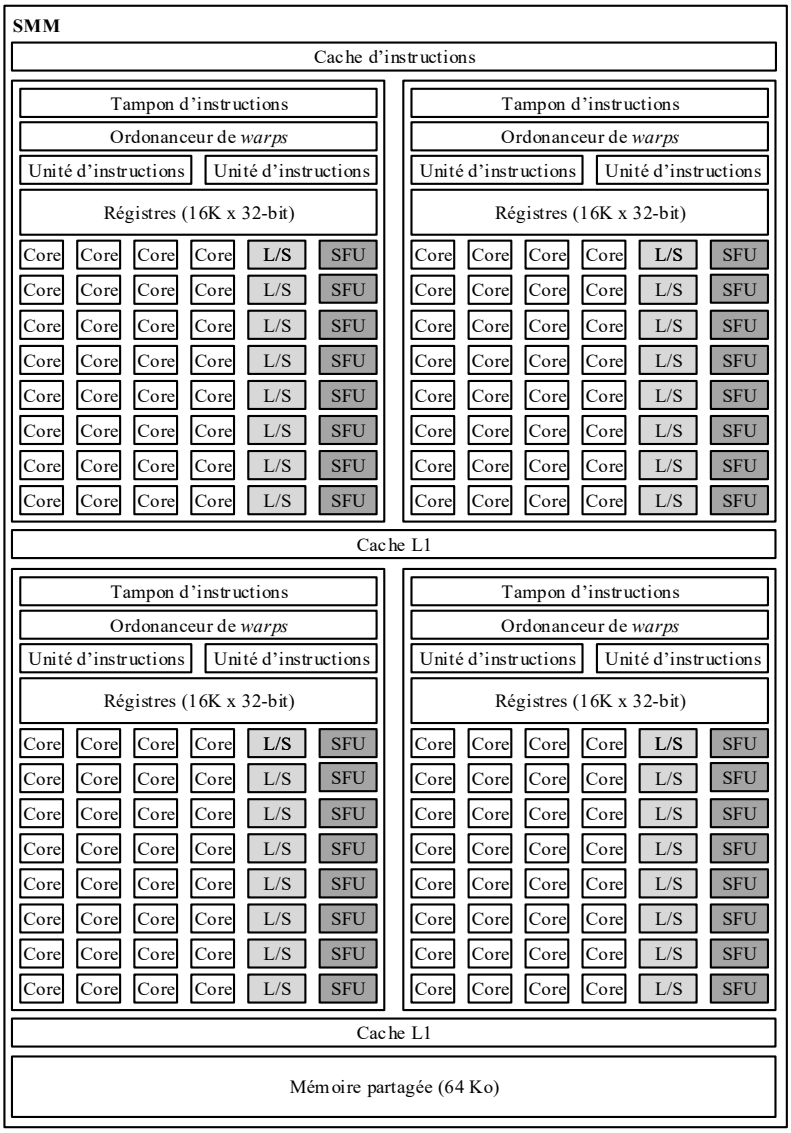


Figure 4.3: Architecture interne d'un multiprocesseur SMM

multiples » (en anglais, « *single instruction, multiple threads* » ou SIMT). Le terme SIMT a été introduit par NVIDIA® afin d'éviter toute confusion avec les systèmes à « instruction unique, données multiples » (en anglais, « *single instruction, multiple data* » ou SIMD). En fait, les deux modèles SIMT et SIMD utilisent des approches très semblables pour implémenter l'exécution parallèle. Tous les deux envoient une instruction unique à plusieurs unités de calcul. La différence est qu'un système SIMD doit exécuter l'instruction sur tous les éléments du vecteur sans exception. Dans le cas

d'un système SIMT, chaque thread a son propre compteur d'instruction et ses propres registres ce qui lui permet de prendre un chemin d'exécution unique. C'est à ce niveau que les GPU se démarquent des systèmes SIMD et offrent une flexibilité d'exécution bien supérieure. Finalement, le multiprocesseur Maxwell inclut 64K registres de 32 bits et une mémoire configurable de 64 Ko. Cette mémoire peut être séparée de trois façons différentes (16-48 Ko, 32-32 Ko et 48-16 Ko) en une mémoire partagée et une mémoire cache L1 non cohérente. La mémoire partagée est contrôlée par le programmeur et permet aux threads qui exécutent sur un même multiprocesseur d'échanger de l'information. La mémoire L1 cache est utilisée pour réduire le nombre d'accès à la mémoire cache L2.

4.4 Modèle de programmation

CUDA™ est une plateforme de développement pour le calcul parallèle et un modèle de programmation développé pour les processeurs graphiques NVIDIA® [230]. CUDA™ inclut un environnement de développement, une interface de programmation et un langage de programmation basé sur le C++ et permet au programmeur de développer des programmes parallèles pour les GPU NVIDIA®. Un programme CUDA™ s'écrit en C++ et comprend des sections de code qui exécutent sur CPU et d'autres qui sont décorées par des mots clés CUDA™ pour exécuter sur le GPU. Au moment de la compilation, le compilateur NVCC (NVIDIA® C Compiler) sépare les différentes sections et génère le code assembleur pour le GPU tandis que le code pour exécution sur CPU est envoyé à un compilateur C++. Il en résulte deux programmes qui collaborent, mais qui sont indépendants afin de permettre une exécution simultanée sur le CPU et le GPU.

Le modèle de programmation CUDA™ est illustré à la Figure 4.4. Le programme débute toujours sur le CPU (*host*) et délègue les calculs au GPU (*device*) en appelant des fonctions parallèles que l'on nomme *kernels*. Lorsqu'un *kernel* est lancé, ce dernier est ajouté à la queue d'exécution du GPU et le contrôle est immédiatement retourné au CPU. On parle ici d'un appel non bloquant. Sur le GPU, l'exécution des *kernels* respectent l'ordre dans lequel ils ont été appelés. Pour exécuter un *kernel*, le GPU crée une grille de threads (appelée *grid*). La grosseur de cette *grid* dépend des paramètres spécifiés par le programmeur. Un même *kernel* peut être appelé plusieurs fois avec différentes grosseurs de *grid*. L'organisation de la *grid* se fait à deux niveaux; une *grid* est composée de plusieurs blocs et chaque bloc est composé de plusieurs threads. La *grid* créée par le *kernel* 1 à la Figure 4.4 comprend six blocs de 16 threads pour un total de 96 threads. Tous les threads d'un même bloc partagent un index de bloc identique accessible par la variable *blockIdx*. Pareillement, à l'intérieur d'un bloc, chaque thread a un index unique accessible par la variable *threadIdx*. Au niveau de la *grid*, chaque thread peut donc être identifié par un index global

*id = blockIdx * blockSize + threadIdx* ce qui permet au programmeur de définir les calculs et les données spécifiques à chaque thread. Malgré que les blocs et les threads à la Figure 4.4 soient organisés en 2D, CUDA™ accepte aussi une organisation 1D ou 3D. En spécifiant plus d'une queue, CUDA™ peut également exécuter plusieurs *kernels* simultanément lorsque les ressources matérielles le permettent comme pour les *kernels* 2 et 3 à la Figure 4.4. Lors de l'exécution, l'ordonnanceur global du GPU distribue les blocs de threads aux différents multiprocesseurs. Si nécessaire, plus d'un bloc peut être alloué et résider sur un même multiprocesseur. Dans le cas des processeurs Maxwell, un maximum de 32 blocs ou 2048 threads (selon la valeur la plus petite) peuvent physiquement résider sur un même multiprocesseur.

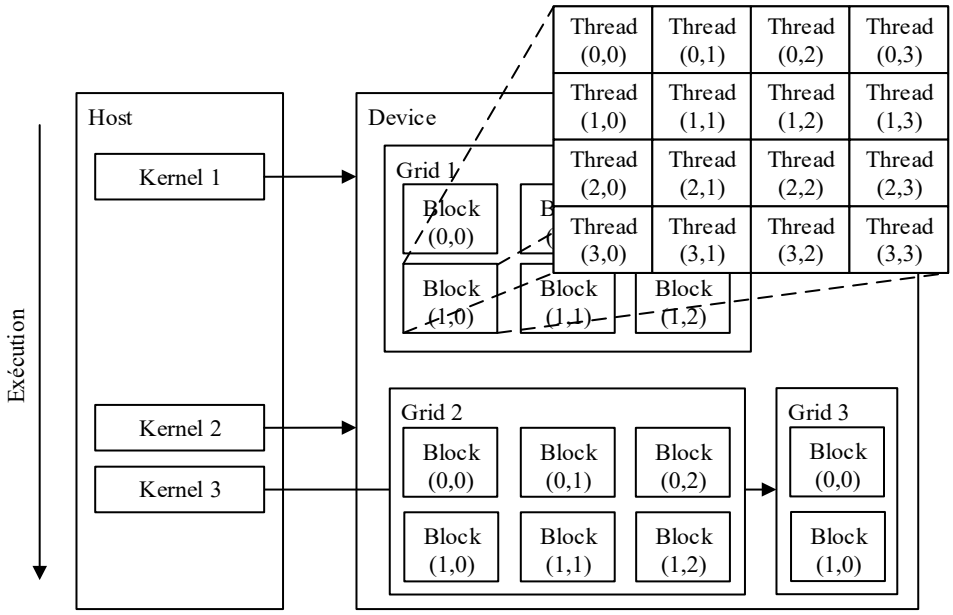


Figure 4.4: Modèle d'exécution d'un programme CUDA™

4.5 Modèle de la mémoire

La puissance théorique d'un GPU est définie par le nombre de cœurs fois leur vitesse d'exécution en opérations à virgule flottante par secondes (*flops*). Or, cette puissance est possible uniquement lorsque la mémoire supporte le flux de données nécessaire aux calculs. Puisque le modèle de la mémoire d'un GPU est non uniforme et que chaque type de mémoire a des caractéristiques différentes telles que sa grosseur, sa bande passante et son délai d'accès, il est important de bien comprendre l'architecture de la mémoire pour en maximiser l'utilisation.

Le modèle de la mémoire en CUDA™ est illustré à la Figure 4.5 et les caractéristiques de chaque type de mémoire, listées au Tableau 4.1. Les *registres* d'un thread sont visibles que par le thread. La *mémoire partagée* est visible à tous les threads d'un même bloc et rend possible l'échange d'information entre les threads. Cette mémoire est divisée en 32 banques et permet un accès parallèle à celles-ci, mais séquentiel à l'intérieur d'une même banque. La *mémoire globale* est visible par tous les threads de la *grid*. Cette mémoire est beaucoup plus large, mais aussi beaucoup plus lente puisqu'elle n'est pas située sur le processeur, mais connectée à celui-ci par un bus de 384 bits. La mémoire globale ne peut pas être utilisée pour communiquer entre les blocs à l'intérieur d'un *kernel* puisque CUDA™ n'offre pas de mécanisme de synchronisation inter-bloc. Cependant, elle peut être utilisée pour sauvegarder de l'information d'un *kernel* à l'autre. Elle est aussi utilisée pour communiquer avec le CPU. Si un thread requiert plus de registres que disponible, ceux-ci déborderont dans ce qu'on appelle la *mémoire locale*. Réservée dans un espace de la mémoire globale lors de la compilation, la mémoire locale est uniquement visible par le thread. Il y a aussi la *mémoire constante*. Cette mémoire a un rôle important pour les applications graphiques, mais peut aussi être très utile dans un programme CUDA™. Elle est limitée à 64 Ko et offre la même performance que la mémoire globale. Cependant, chaque multiprocesseur inclut une cache de 8 Ko spécifique à cette mémoire. La mémoire constante offre donc une bonne performance lorsque ses données sont réutilisées souvent. Finalement, il y a aussi les *mémoires caches* L1 et L2 que nous n'avons pas incluses à la Figure 4.5 puisque celles-ci ne sont pas contrôlées par les programmeurs. Ces deux caches aident à réduire le nombre d'accès à la mémoire globale.

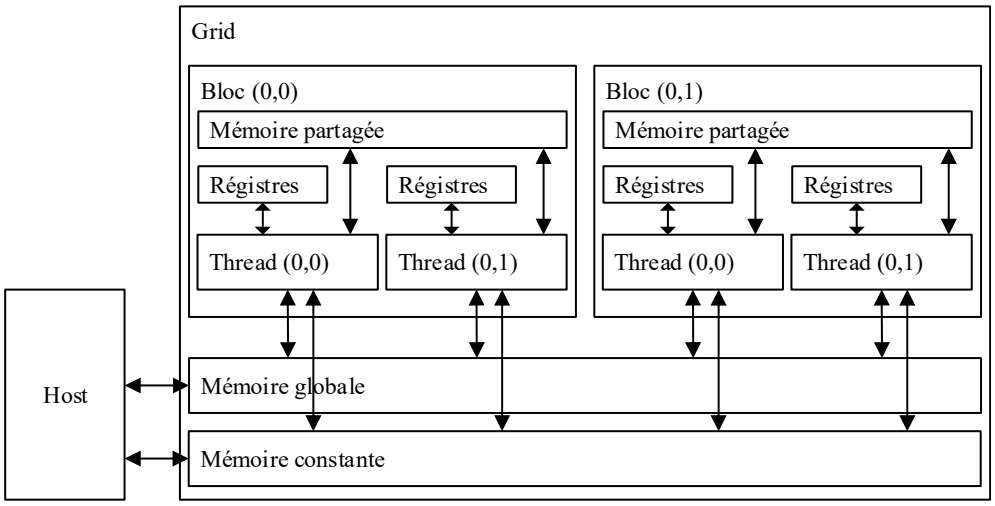


Figure 4.5: Modèle de la mémoire du processeur graphique

TABLEAU 4.1
CARACTÉRISTIQUES POUR CHAQUE TYPE DE MÉMOIRE [232]

Type mémoire	deAccès	Grosseur	Bande passante	Délais d'accès	Durée de vie
Registres	R/W par thread	32K x 32 bits par SMM	~8 To/s	1 cycle	*kernel*
Locale	R/W par thread	max 12 Go	~200 Go/s	~400 à 600	*kernel*
Partagée	R/W par bloc	max 48 Ko par SMM	~1.5 To/s	1 à 32 cycles	*kernel*
Globale	R/W par grid	max 12 Go	~200 Go/s	~400 à 600	programme
Constante	R par grid	64 Ko	~200 Go/s	~400 à 600	programme

4.6 Pratiques de programmation parallèle

Le modèle de programmation et le modèle de la mémoire que nous venons juste de présenter permettent d'abstraire les détails de l'architecture du GPU et sont suffisants pour le développement d'applications parallèle en CUDA™. Sans se soucier de l'organisation matérielle du GPU, le programmeur peut rapidement écrire un programme en CUDA™ et obtenir une accélération de 10x comparée à une version séquentielle sur CPU [229]. Cependant, pour obtenir une meilleure performance, il faut connaître ces détails d'architecture et optimiser le programme afin qu'il utilise pleinement le potentiel du GPU. Une bonne implémentation sur GPU peut facilement permettre une accélération de plus de 100x [229]. Dans cette section, nous présentons six stratégies de programmation pour aider le programmeur à maximiser le potentiel du GPU. Ces techniques seront utilisées aux chapitres suivants pour s'assurer que les algorithmes parallèles développés soient le plus performants possible. Les stratégies discutées dans cette section proviennent principalement de [229] et [233].

1. **Maximiser le parallélisme exploité.** Pour profiter de la grande puissance de calcul du GPU, il faut que l'application parallèle utilise un très grand nombre de threads de façon à maximiser l'utilisation des cœurs. Pour atteindre ce nombre de threads, l'application doit nécessairement exploiter le parallélisme au niveau des données.

2. **Minimiser la divergence d'exécution.** Bien que les multiprocesseurs soient composés de plusieurs cœurs, ils possèdent un nombre limité d'ordonnanceurs et d'unités d'envoi d'instructions. Le multiprocesseur peut donc exécuter plusieurs threads simultanément à condition que ces threads exécutent tous la même instruction. Lorsqu'un programme contient des instructions conditionnelles (*if, elseif*) à l'intérieur d'un *warp*, le multiprocesseur doit exécuter séquentiellement chacun des chemins possibles de la condition. Ce phénomène est appelé la divergence d'exécution et doit être minimisé le plus possible afin de maximiser la performance.

3. **Minimiser les transferts sur le bus PCI Express.** Étant donné que le GPU et le CPU ont des espaces de mémoire différents, les données doivent être transférées par le bus PCI Express limité à une bande passante théorique de 8 Go/s. Comparée à celle de la mémoire globale ou de la mémoire partagée, la bande passante du bus PCI Express est beaucoup plus petite et représente un goulot d'étranglement important à la performance. Il est donc recommandé de minimiser le plus possible les transferts sur le bus PCI Express.

4. **Accès coalescent à la mémoire globale.** La mémoire globale du GPU a une bande passante effective maximale d'environ 330 Go/s, ce qui est bien supérieur à celle d'un CPU. Pour atteindre cette limite, il faut cependant que l'accès soit coalescent. En fait, l'accès à la mémoire se fait par *warp* et pour une largeur de ligne de cache de 128 octets. Chaque lecture en mémoire force donc le chargement de 128 octets voisin. Lorsque tous les threads d'un même *warp* demandent la lecture d'un mot de 4 octets et que ces mots sont coalescents et alignés à une ligne de cache, le GPU agglomère ces opérations et lit une seule ligne de cache de 128 octets résultant en une très bonne performance. Dans le cas contraire où l'accès est totalement irrégulier, jusqu'à 32 lectures indépendantes seront exécutées séquentiellement. Pour bénéficier du débit maximal de la mémoire, il faut éviter tout désalignement, tout espacement entre les données de threads voisins et tout accès irrégulier à la mémoire.

5. **Utilisation de la mémoire partagée.** La mémoire partagée de l'architecture Maxwell est restreinte à 64 Ko par multiprocesseur, mais offre d'un délai d'accès 20 fois plus rapide que la mémoire globale. Il est donc profitable de l'utiliser comme mémoire tampon lorsque des données sont accédées plus d'une fois à l'intérieur d'une *kernel*.

6. **Maximiser le taux d'occupation.** Chaque multiprocesseur de l'architecture Maxwell est équipé de 128 cœurs, mais possède suffisamment de registres pour contenir 2048 threads résidents. Un *kernel* qui crée 2048 threads résidents par multiprocesseur résulte donc en un taux d'occupation de 100%. Un taux d'occupation élevé est préférable pour maximiser l'utilisation des cœurs, mais aussi pour permettre au GPU de masquer les délais d'accès à la mémoire globale en réordonnançant les threads. Contrairement au CPU, l'ordonnancement des threads CUDA™ est implémenté par un circuit matériel sur le GPU et ne consomme aucun cycle d'horloge.

Finalement, il y a une dernière considération à suivre qui n'est pas nécessairement spécifique aux GPU, mais à tous les programmes parallèles. Comme nous l'avons discuté au Chapitre 2, la loi d'Amdahl [33] souligne que l'accélération maximale d'un programme est limitée par sa partie séquentielle. Prenons l'exemple d'un programme séquentiel dont 20% du temps d'exécution ne peut être parallélisé. Ce programme

aura une accélération maximale de 5x même si le temps de calcul de la partie parallèle est infiniment réduit. Lors du développement d'algorithmes parallèles sur GPU, il est crucial de limiter la partie séquentielle et de paralléliser toutes les étapes de l'algorithme.

4.7 Primitives parallèles

Les algorithmes parallèles implémentés sur CPU multicœurs exploitent habituellement un parallélisme au niveau des tâches. Le programme est divisé en plusieurs tâches et celles-ci sont exécutées simultanément par les multiples cœurs. Chaque tâche peut être très complexe et l'accélération maximale est atteinte lorsque la communication interprocessus est minimisée. Dans le cas des GPU, une parallélisation au niveau des tâches est difficilement possible pour deux raisons. Tout d'abord, le niveau de parallélisme exploité n'est pas suffisant pour occuper les milliers de cœurs disponibles. D'autre part, l'architecture interne des cœurs du GPU est beaucoup plus simple que celle d'un CPU et ne peut pas gérer efficacement le flux de logique complexe de la tâche. Pour tirer avantage du GPU, il faut exploiter un niveau de parallélisme beaucoup plus élevé tout en gardant un flux de logique simple. La conception de l'application parallèle doit donc être repensée pour exploiter un parallélisme au niveau des données. Chaque opération du programme séquentiel original est analysée et substituée, lorsque possible, par une primitive parallèle qui effectue le même travail, en parallèle, en moins d'étapes. Comme il y a habituellement beaucoup plus de données que les tâches dans un programme, le parallélisme de niveau de données conduit généralement à un niveau de parallélisme beaucoup plus élevé.

Dans cette section, nous présentons sept primitives parallèles qui seront utilisées aux chapitres suivants pour la conception des algorithmes sur GPU. Nous introduisons d'abord quatre primitives fondamentales, soient les fonctions de map, de réduction, de scan et de dispersion. Nous utilisons ensuite ces primitives de base pour implémenter des opérations plus complexes telles que la compaction, la réduction segmentée et le tri bitonique.

4.7.1 Map parallèle

Une boucle séquentielle où chaque itération s'exécute indépendamment des autres peut facilement être parallélisée par une fonction map parallèle. Cette opération prend comme entrée une série de valeurs, exécute les itérations de la boucle concurremment et produit un élément de sortie par élément d'entrée. Un exemple de fonction map pour calculer le carré de chaque élément d'un vecteur d'entrée est illustré à la Figure 4.6. Si le nombre de cœurs disponibles est suffisant, la fonction

map exécute en une seule étape comparée à la boucle séquentielle qui prend n étapes où n représente le nombre d'éléments à traiter. La fonction map est la primitive parallèle qui offre le plus grand gain de performance. Pour une implémentation sur GPU, il s'agit simplement de lancer une *grid* de blocs de threads où chaque thread CUDA™ traite un ou plusieurs éléments. Aucun mécanisme de synchronisation n'est nécessaire entre les threads ou les blocs.

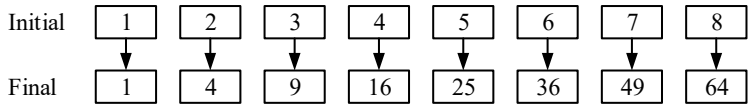


Figure 4.6: Fonction map pour calculer le carré des éléments d'un vecteur

4.7.2 Réduction parallèle

La réduction parallèle est une autre primitive fondamentale. Cette opération prend comme entrée une série de valeurs et produit comme sortie une seule valeur. Une fonction de réduction calculant la somme des éléments d'un vecteur est illustrée à la Figure 4.7. Pour un vecteur de n éléments, la réduction parallèle complète en $log_2\ (n)$ étapes comparées à n étapes pour une implémentation séquentielle à l'aide d'une boucle. À la Figure 4.7, les cases foncées représentent les threads actifs. Par exemple, à l'étape 1, quatre threads sont nécessaires et chaque thread additionne deux éléments voisins. À l'étape suivante, le nombre de threads actifs est réduit de moitié et chaque thread traite encore deux éléments voisins. Ce processus continue jusqu'à ce qu'il y ait un seul thread actif. La somme produite par ce thread représente alors le résultat final. Étant donné que les threads traitent des éléments voisins, on dit que cette réduction suit une topologie voisine. Pour une implémentation sur GPU, la topologie voisine est problématique pour deux raisons. Premièrement, l'accès à la mémoire n'est pas coalescent puisque les adresses de mémoire accédées ne sont pas voisines. Le GPU ne pourra donc pas bénéficier de sa large ligne de mémoire cache et plusieurs accès séquentiels à la mémoire globale devront être faits. Deuxièmement, la topologie voisine engendre la divergence d'exécution à l'intérieur des *warps* [234]. À chaque étape, le nombre de threads actifs à l'intérieur du *warp* diminue de moitié. Or, étant donné que l'architecture interne du GPU force les 32 threads d'un *warp* à exécuter en harmonie, plusieurs des cœurs seront inactifs affectant ainsi la performance du GPU.

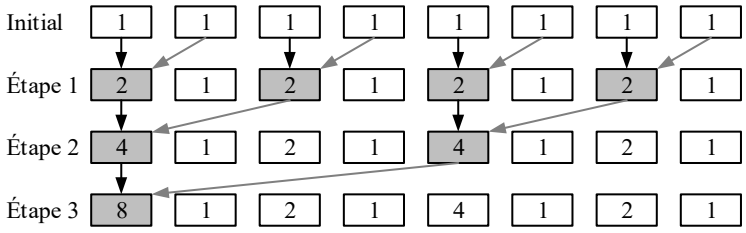


Figure 4.7: Réduction parallèle pour calculer la somme des éléments d'un vecteur (topologie voisine)

Comme il est expliqué dans [234], une topologie préférable est d'entrelacer les éléments accédés par chaque thread tel qu'illustré à la Figure 4.8. La topologie entrelacée permet un accès coalescent à la mémoire globale puisque des threads voisins accèdent toujours des éléments de mémoire voisins. De plus, cette topologie évite la divergence d'exécution à l'intérieur des *warps*. Prenons le cas où un bloc de 1024 threads CUDA™ serait utilisé pour réduire un vecteur de 2048 éléments. Au niveau du matériel, les 1024 threads sont automatiquement divisés en 32 *warps*. À l'étape 1, tous les 32 *warps* sont actifs. À l'étape 2, seule la moitié des *warps* exécute, mais tous les threads à l'intérieur de chaque *warp* sont actifs ce qui permet d'utiliser pleinement les cœurs du groupe multiprocesseur assurant ainsi la meilleure performance possible.

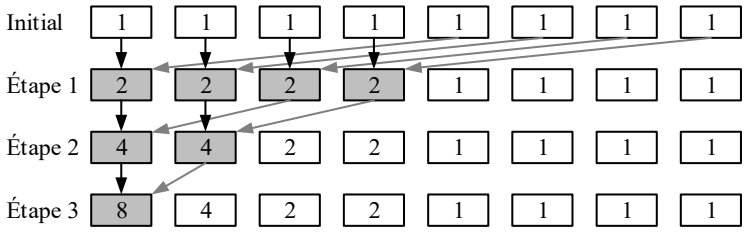


Figure 4.8: Réduction parallèle pour calculer la somme des éléments d'un vecteur (topologie entrelacée)

Finalement, une troisième optimisation pour la réduction parallèle sur GPU est possible par l'utilisation de la mémoire partagée qui est beaucoup plus rapide que la mémoire globale. Étant donné que le nombre maximal de threads CUDA™ par bloc est de 1024, la réduction parallèle n'utilise jamais plus de 512 éléments de mémoire temporaire. Au lieu de sauvegarder ces éléments dans la mémoire globale du GPU, on peut les sauvegarder dans la mémoire partagée du groupe multiprocesseur et bénéficier d'un délai d'accès beaucoup plus petit. La performance est alors grandement améliorée. Lorsque le nombre d'éléments à réduire dépasse 2048, il est possible d'assigner plusieurs éléments à chaque thread avant d'entreprendre de

mécanisme de réduction parallèle afin de garder le nombre de threads à 1024. Pour exploiter un niveau de parallélisme plus élevé, il est aussi possible d'effectuer la réduction en deux phases comme à la Figure 4.9. À la phase 1, le vecteur d'entrées est divisé en sous-ensembles, chacun réduit par un bloc de threads CUDA™. Les sommes partielles sont ensuite additionnées à la phase 2 en utilisant un seul bloc de threads. Lorsque le nombre d'éléments est grand, une réduction en deux phases donne une meilleure performance puisqu'elle exploite un niveau de parallélisme supérieur.

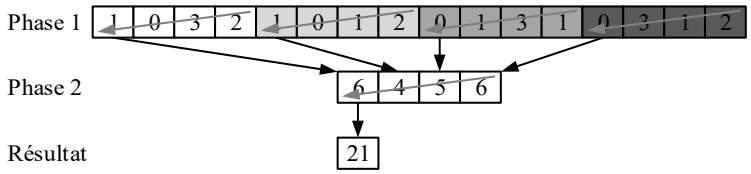


Figure 4.9: Réduction parallèle en deux phases

Pour mesurer la différence de performance entre les trois implémentations discutées, nous exécutons une réduction parallèle pour différents nombres d'éléments n et présentons à la Figure 4.10 les temps d'exécution en microsecondes. Pour les trois implémentations, le programme test utilise un seul bloc de threads. Le nombre de threads est égal à n/2. Dans les cas où n est égal à 4096 et 8192, nous limitons le nombre de threads à 1024 et chaque thread traite respectivement 4 et 8 éléments. Les tests sont exécutés sur un processeur graphique NVIDIA® GTX 750 Ti. D'après le graphe, on remarque bien l'avantage d'une implémentation suivant la topologie entrelacée et de l'utilisation de la mémoire partagée.

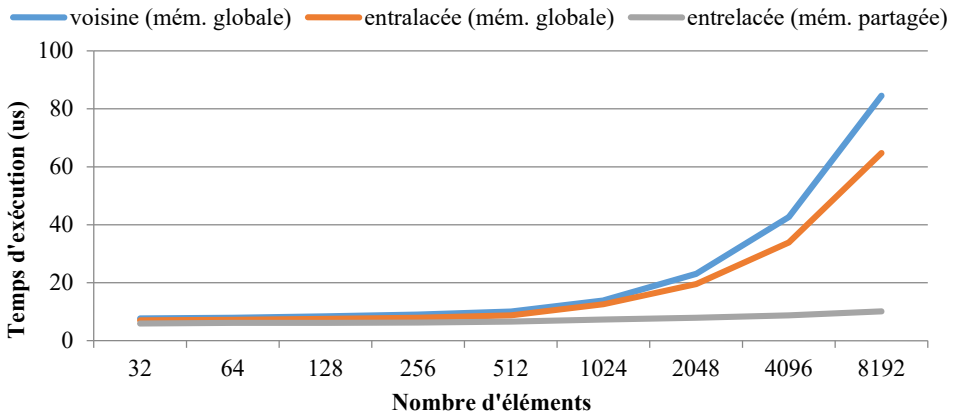


Figure 4.10: Temps d'exécution (µs) pour trois implémentations de la réduction parallèle

4.7.3 Scan parallèle

La troisième primitive fondamentale discutée dans ce chapitre est le scan parallèle. Cette opération calcule la réduction partielle de chaque élément d'un vecteur d'entrée et produit un vecteur de sortie. Ceci signifie que l'élément i du vecteur de sortie est la réduction des éléments 0 à i du vecteur d'entrée. L'opération de scan est utilisée dans plusieurs algorithmes parallèles. Prenons par exemple un algorithme parallèle où chaque thread produit un nombre variable d'éléments qui doivent être sauvegardés séquentiellement dans un seul vecteur de sortie. Le scan parallèle serait alors utilisé pour calculer l'adresse de destination pour chaque thread. Dans un programme séquentiel, ce calcul se ferait à l'aide d'une variable y que l'on incrémenterait à chaque itération i utilisant $y_i = y_{i-1} + x_i$, où x est le vecteur d'entrée. Le scan parallèle est régulièrement implémenté à l'aide du réseau de Kooge-Stone [235] tel qu'illustré à la Figure 4.11. Tout comme à l'exemple précédent, les cases foncées représentent les threads actifs. L'opération parallèle prend $log_2(n)$ étapes comparées à n étapes pour une implémentation séquentielle à l'aide d'une boucle. Pour une implémentation en CUDA™, le réseau de Kooge-Stone permet un accès coalescent à la mémoire et évite la divergence d'exécution à l'intérieur des *warps*. Cependant, la quantité de calculs effectuée par l'algorithme parallèle est bien plus grande que celle nécessaire par l'implémentation séquentielle. D'après le réseau à la Figure 4.11, on remarque que le nombre d'additions effectuées est de $7 + 6 + 4 = 17$ ou d'une forme plus générale, $n * log_2(n) - (n - 1)$, où n est le nombre d'éléments [229]. Dans le cas d'une implémentation séquentielle, le nombre d'additions est uniquement de $n - 1$. On dira alors que le scan parallèle basé sur le réseau de Kooge-Stone n'est pas efficace en termes de travail. Dans le cas où n est grand, le nombre de cœurs disponibles sur le GPU ne sera pas suffisant pour effectuer tous les calculs simultanément. Chaque étape à la Figure 4.11 prendra alors plusieurs cycles d'exécution.

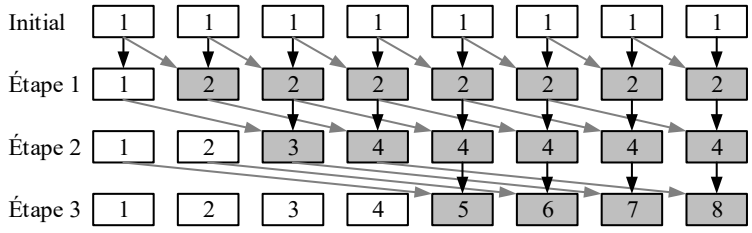


Figure 4.11: Fonction de scan inclusif basée sur le réseau de Kooge-Stone

Pour effectuer un scan parallèle sur un grand nombre d'éléments, il est recommandé d'utiliser le réseau de Brent-Kung [235] illustré à la Figure 4.12. Comparativement à la méthode précédente, le nombre d'additions nécessaire est

réduit à $2n - 3$ ce qui en fait une implémentation beaucoup plus efficace en termes de travail. Malgré que le nombre d'étapes soit augmenté à $2 * log_2(n) - 1$, la performance d'un réseau de Brent-Kung est habituellement supérieure à celle d'un réseau de Kooge-Stone lorsque le vecteur d'entrée est large.

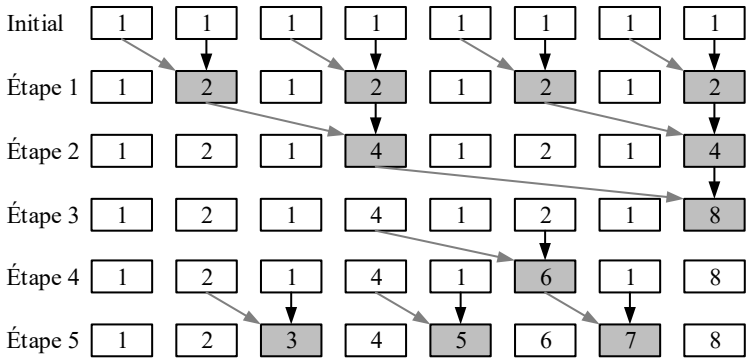


Figure 4.12: Fonction de scan inclusif basée sur le réseau de Brent-Kung

Toutefois, le réseau de Brent-Kung n'est pas l'approche optimale pour une implémentation sur GPU. Comme nous l'avons discuté précédemment, l'architecture interne du GPU force les threads d'un même *warp* à exécuter en harmonie. Or, la divergence d'exécution présente dans le réseau de Brent-Kung limite le nombre de cœurs actifs à l'intérieur du multiprocesseur. Pour exploiter pleinement l'architecture parallèle du GPU et obtenir une performance supérieure, il est possible de diviser le vecteur d'entrée en sous-ensembles et d'effectuer le scan en quatre phases comme à la Figure 4.13. À la phase 1, un scan basé sur le réseau de Kooge-Stone est effectué concurremment sur tous les sous-ensembles en utilisant un *warp* par ensemble. À la phase 2, la valeur obtenue pour le dernier élément de chaque sous-ensemble est copiée dans un vecteur temporaire. Un scan est ensuite exécuté sur ce vecteur à la phase 3. Finalement, les valeurs obtenues sont distribuées et additionnées au vecteur calculé à la phase 1. Cette approche en quatre phases assure un accès coalescent à la mémoire, minimise la divergence d'exécution et s'adapte parfaitement à l'architecture du GPU.

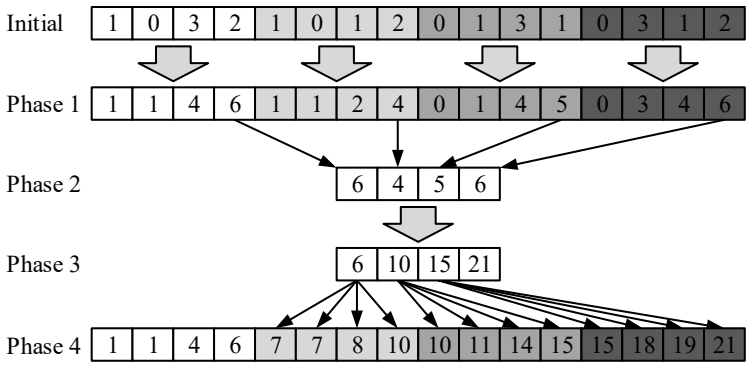


Figure 4.13: Fonction de scan inclusive multiniveau

Comme nous l'avons fait précédemment, nous exécutons un scan parallèle à l'aide des trois implémentations discutées et présentons à la Figure 4.14 les temps d'exécution en microsecondes pour différents nombres d'éléments n. Les deux premières implémentations sont basées sur [229] tandis que la troisième vient de [236]. Pour les trois implémentations, le programme test utilise un seul bloc de threads. Le nombre de threads est égal à n/2. Dans les cas où n est égal à 4096 et 8192, nous limitons le nombre de threads à 1024 et effectuons respectivement 2 et 4 passes pour compléter le scan. Les tests sont encore une fois exécutés sur un processeur graphique NVIDIA® GTX 750 Ti. D'après les résultats, on remarque que l'implémentation basée sur le réseau de Kooge-Stone est plus performante que celle basée sur le réseau de Brent-Kung lorsque le vecteur d'entrée contient moins de 2048 éléments. Ceci démontre l'importance de l'accès coalescent à la mémoire et de la convergence d'exécution. Cependant, lorsque le nombre d'éléments dépasse 2048, le réseau de Brent-Kung est plus performant grâce à sa meilleure efficacité en termes de travail. Finalement, l'implémentation multiniveau offre une performance significativement supérieure aux deux précédentes par l'utilisation de la mémoire partagée et une meilleure adaptation à l'architecture du GPU.

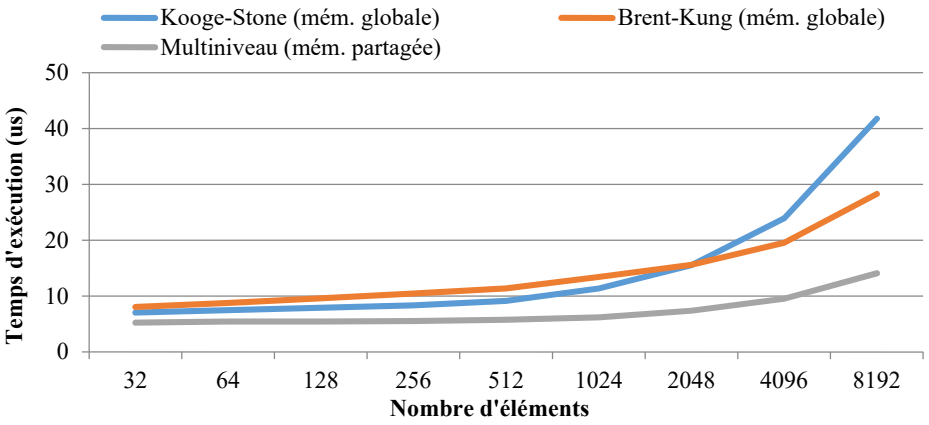


Figure 4.14: Temps d'exécution (µs) pour trois implémentations du scan parallèle

Les opérations de scan discutées jusqu'à présent sont toutes inclusives; l'élément i du vecteur de sortie est la réduction des éléments 0 à i du vecteur d'entrée. Il existe aussi une variante exclusive où l'élément i du vecteur de sortie est la réduction des éléments 0 à $i-1$ du vecteur d'entrée. Le scan exclusif est illustré à la Figure 4.15. L'implémentation est identique à celle du scan inclusif, mais décale les entrées vers la droite lors de la lecture. Le scan exclusif est une primitive important qui sera utilisée plus loin dans cette section pour implémenter les opérations de compaction et de réduction segmentée.

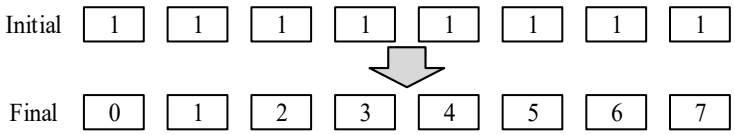


Figure 4.15: Fonction de scan exclusif pour calculer la somme cumulative des éléments d'un vecteur

4.7.4 Dispersion parallèle

La quatrième et dernière primitive fondamentale introduite dans cette section est la dispersion parallèle illustrée à la Figure 4.16. Cette fonction consiste à distribuer les éléments d'un vecteur d'entrée vers le vecteur de sortie d'après leur indice de destination. Lors d'une opération de dispersion, il est important de s'assurer que deux éléments ne pointent jamais vers la même adresse de destination afin d'éviter une situation de compétition. L'implémentation sur GPU de la dispersion parallèle est similaire à celle de la fonction map. L'opération s'exécute en une seule étape comparée à n pour une implémentation séquentielle.

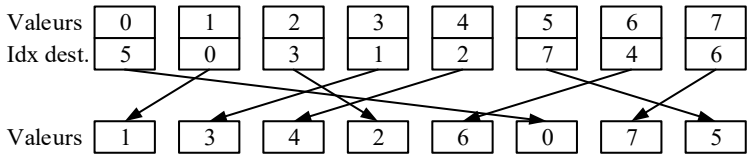


Figure 4.16: Fonction de dispersion

4.7.5 Compaction parallèle

Les quatre primitives fondamentales discutées aux paragraphes précédents peuvent être utilisées pour construire d'autres primitives plus complexes. Par exemple, le scan exclusif est utilisé pour implémenter la primitive de compaction parallèle. L'opération de compaction est illustrée à la Figure 4.17 et consiste à sélectionner certains éléments du vecteur d'entrés et à les sauvegarder de façon continue dans le vecteur de sortie. Cette opération sera utilisée entre autres par les algorithmes d'analyse d'écoulement de puissance au chapitre 7 pour construire les listes des bus PV et PQ. La compaction parallèle se fait en trois étapes. À l'étape 1, une fonction map permet d'identifier les éléments à copier en assignant une valeur de "1" à leur drapeau. Une opération de scan exclusif est ensuite exécutée sur le vecteur de drapeaux à l'étape 2 afin de calculer les adresses de destination des éléments choisis. Finalement, une fonction de dispersion est utilisée à l'étape 3 pour copier les éléments choisis du vecteur d'entrée vers le vecteur de sortie d'après leur indice de destination.

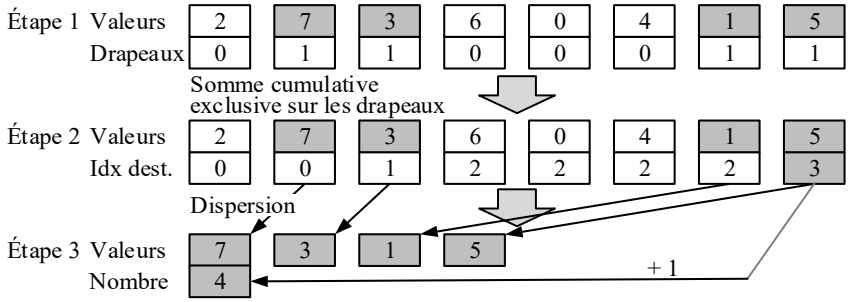


Figure 4.17: Compaction des données sélectionnées par un drapeau

4.7.6 Réduction segmentée parallèle

Le scan exclusif est aussi à la base de la réduction segmentée. Cette primitive parallèle effectue une opération de réduction sur les éléments de segments délimités par une série continue de clefs identiques. La réduction segmentée sera utilisée au chapitre 7 pour construire la matrice d'admittance creuse d'un réseau de transport d'électricité. Un exemple de réduction segmentée est illustré à la Figure 4.18. Dans

cet exemple, les valeurs du vecteur d'entrée sont toutes égales à "1" et les segments sont identifiés par le vecteur de clefs. On remarque que le vecteur d'entrée est divisé en quatre segments. À la première étape, une fonction map est utilisée pour identifier les éléments qui marquent la fin de chaque segment en leur assignant une valeur de drapeau égale à "1". L'opération de réduction est ensuite exécutée à l'étape 2, mais doit inclure une vérification supplémentaire afin de considérer les démarcations entre les segments. Par exemple, l'opération de réduction utilisée à la Figure 4.18 est définie comme suit et additionne deux valeurs uniquement lorsque celles-ci font partie du même segment:

$$Valeurs[i] = \begin{cases} Valeurs[i] + Valeurs[j], & si\ \ Clefs[i] = Clefs[j] \\ Valeurs[i] + 0 \quad , & si\ \ Clefs[i] \neq Clefs[j] \end{cases} \tag{4.1}$$

L'étape 2 inclut aussi un scan exclusif sur le vecteur de drapeaux afin de calculer les indices de destinations. Finalement, une fonction de dispersion est utilisée à l'étape 3 pour copier la clef et la valeur du dernier élément de chaque segment vers le vecteur de sortie.

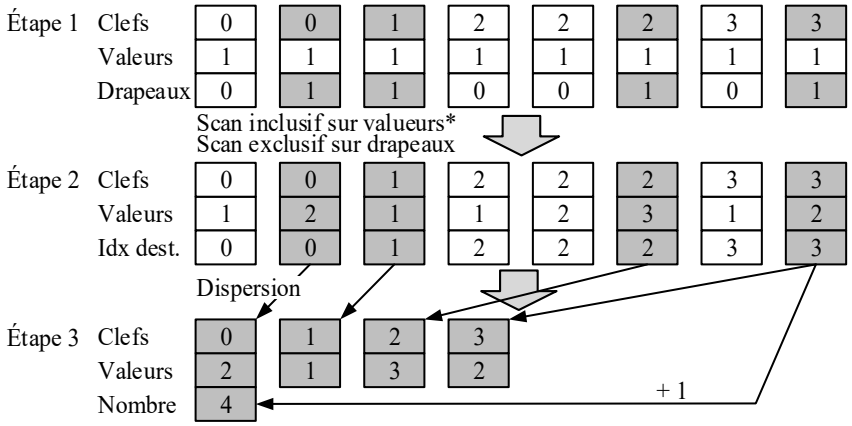


Figure 4.18: Réduction parallèle segmentée pour calculer la somme segmentée des éléments d'un vecteur

Dans le cas particulier où toutes les valeurs du vecteur d'entrée sont égales à "1" comme à l'exemple précédent, la réduction segmentée calcule en fait la longueur de chaque segment comme à la Figure 4.19. Cette opération sera utilisée entres autres au chapitre 7 pour calculer le nombre d'éléments dans chaque rangé de la matrice d'admittance creuse.

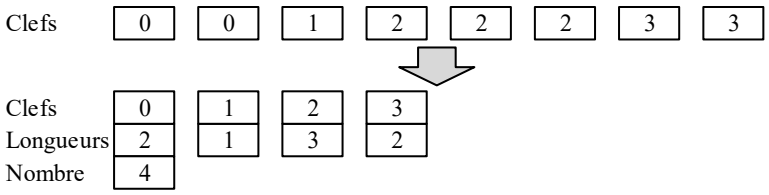


Figure 4.19: Calcul de la longueur des segments

4.7.7 Tri parallèle

Le tri d'une séquence de nombres est une autre opération courante qui peut être accélérée par la parallélisation sur GPU. Dans le cadre de cette thèse, un algorithme de tri parallèle sera critique pour la minimisation des harmoniques de la tension de sortie d'un onduleur multiniveau. Pour une implémentation séquentielle, un des algorithmes les plus performants est le tri rapide (en anglais *quicksort*) [237]. Dépendamment de l'ordre initial des données, cet algorithme complète après $n\,log_2(n)$ étapes dans le meilleur des cas ou, n^2 étapes dans le pire des cas, où n est le nombre d'éléments à ordonner. Malheureusement, étant donné que le flux d'exécution du tri rapide dépend de l'ordre initial, il est difficile de le paralléliser efficacement sur GPU. Pour une implémentation parallèle sur GPU, il faut faire recours à un réseau de tri. Le réseau est caractérisé par un flux d'exécution fixe, peu importe la valeur des données.

Le réseau de tri le plus simple est probablement celui du tri par transposition pair-impair [238] illustré à la Figure 4.20. Chaque flèche horizontale représente une comparaison entre deux valeurs. Pour un tri en ordre croissant, ces valeurs sont échangées lorsque nécessaire afin que la plus petite soit à la queue de la flèche et la plus grande, à la tête. Comparé aux algorithmes de tri séquentiels, le réseau de trie pair-impair est avantageux puisqu'il nécessite seulement $n-1$ étapes pour trier n éléments. Cependant, cet algorithme n'est pas efficace en terme de travail puisque le nombre de comparaisons nécessaires est proportionnel à n^2. Sa performance diminue rapidement lorsque n dépasse le nombre de processeurs disponibles sur le GPU.

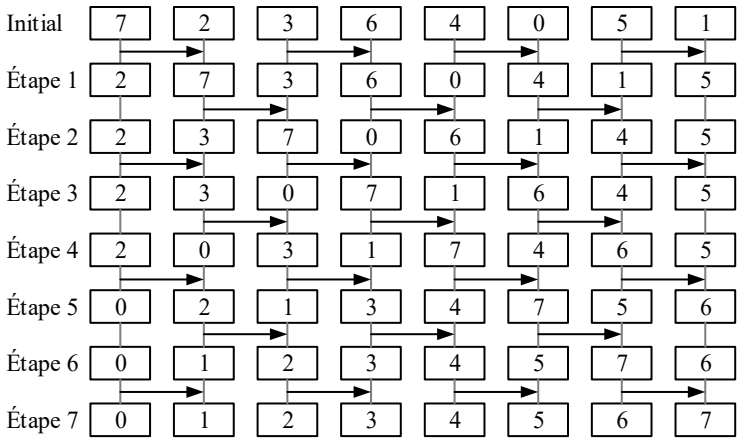


Figure 4.20: Réseau de tri pair-impair

Un autre exemple de réseau de tri parallèle est le tri bitonique. Développé par Batcher [239] et récemment adapté pour le GPU par Peters et coll. [55], cet algorithme offre une meilleure efficacité en termes de travail que la méthode par transposition pair-impair. Le tri bitonique exploite une propriété particulière des séquences dites bitoniques. Une séquence x_0 ... x_{n-1} de n éléments est bitonique si $x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$ est vrai pour la séquence originale ou tout décalage circulaire de la séquence originale. Un exemple d'une telle séquence est {2, 3, 6, 7, 5, 4, 1, 0}. Même si l'on décale circulairement cette séquence, disons de trois éléments vers la droite, la séquence résultante {4, 1, 0, 2, 3, 6, 7, 5} respecte toujours la propriété que nous venons d'énoncer. Or, les séquences bitoniques ont une caractéristique très intéressante qui les rend utiles pour le développement de réseau de tri. Si on compare et échange élément x_i avec élément $x_{i+n/2}$ pour tous les $i < n/2$, on obtient deux séquences bitoniques où tous les éléments de la première sont plus petits où égaux aux éléments de la seconde. On peut ensuite répéter cette opération de fusion pour chacune des deux séquences afin d'obtenir quatre séquences. Ce procédé peut continuer jusqu'à ce que tous les éléments de la séquence initiale soient ordonnés. Cette opération est illustrée à la Figure 4.21 pour la séquence {2, 3, 6, 7, 5, 4, 1, 0} et prend $log_2(n)$ étapes à compléter.

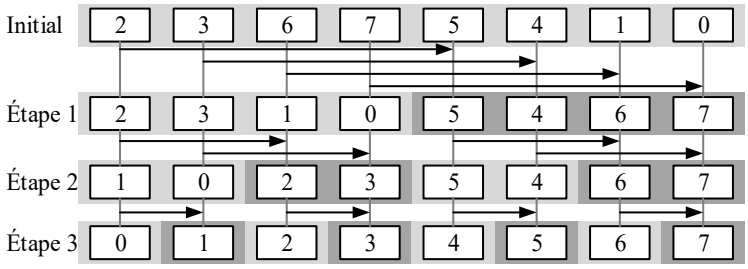


Figure 4.21: Réseau de tri d'une séquence bitonique

La fusion successive discutée ci-dessus est limitée à l'ordonnancement d'une séquence bitonique. Or, puisqu'une séquence de deux éléments est nécessairement bitonique, il est possible d'utiliser la fusion pour ordonner ces deux éléments. Dans le réseau de tri bitonique illustré à la Figure 4.22, le vecteur d'entrée est divisé en quatre séquences bitoniques, chacune composée de deux éléments. Si la direction (croissante, décroissante) du tri est alternée d'une séquence à l'autre, des séquences bitoniques de quatre éléments seront obtenues à la phase 1. L'opération de fusion est répétée une seconde fois, toujours en alternant la direction, afin d'obtenir une seule séquence bitonique de huit éléments à la phase 2. Finalement, une dernière fusion est exécutée pour obtenir une séquence ordonnée à la phase 3. Pour un vecteur de n éléments, le réseau de tri bitonique exécute $log_2(n)$ phases et le nombre d'étapes à la phase i est égal à $log_2(i)$. Tel que dérivé dans [240], le nombre total d'étapes est de $log_2(n) * (log_2(n) + 1)/2$ et chaque étape requiert $n/2$ comparaisons. Pour ordonner 1024 éléments, le réseau de tri bitonique effectue uniquement 55 étapes et est beaucoup plus efficace que le réseau de transposition pair-impair qui lui nécessiterait 1023 étapes.

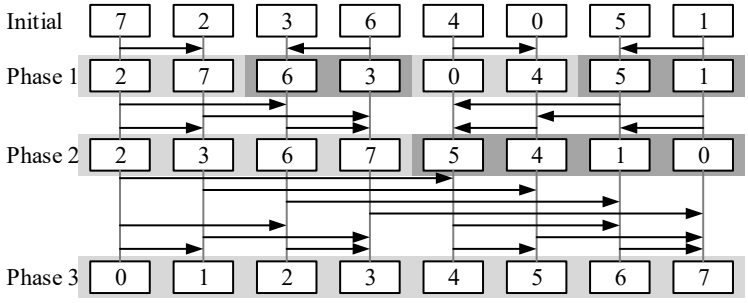


Figure 4.22: Réseau de tri bitonique

Tout comme pour les exemples précédents, afin de visualiser la différence de performance entre les deux approches discutées, nous effectuons le tri d'une séquence pour différents nombres d'éléments n et présentons à la Figure 4.23 les temps

d'exécution en microsecondes. Les deux implémentations utilisent un seul bloc de thread et copient les valeurs dans la mémoire partagée avant d'effectuer le tri. Le nombre de threads utilisés est égal à $n/2$. Dans les cas où n est égal à 4096 et 8192, nous limitons le nombre de threads à 1024 et chaque thread traite respectivement 4 et 8 éléments. Le processeur graphique utilisé pour le test est toujours le NVIDIA® GTX 750 Ti. À la Figure 4.23, il est important de noter que l'échelle verticale est logarithmique. D'après les temps mesurés, il est évident que le tri bitonique est beaucoup plus performant que celui par transposition pair-impair à cause d'un ordre de complexité plus petit. Pour cette raison, l'algorithme de minimisation des harmoniques développé dans le cadre de cette thèse utilise le tri bitonique.

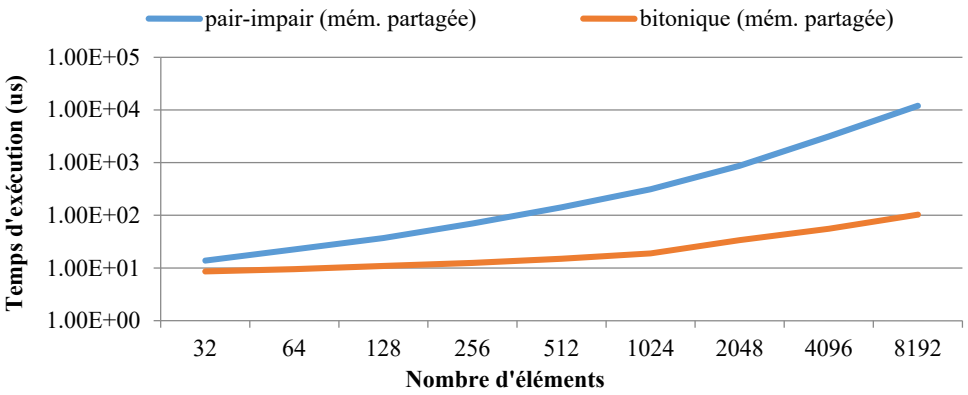


Figure 4.23: Temps d'exécution (µs) pour deux implémentations d'un tri parallèle

Avant de terminer, il est important de noter que plusieurs des primitives discutées dans ce chapitre supposent que le nombre d'éléments n à traiter est une puissance de deux (c.-à-d. $n = 2^x$ où x est un entier). Or, il est possible de considérer un nombre d'éléments arbitraires. Il faut tout simplement copier le vecteur d'entrée dans une mémoire tampon (préférablement dans la mémoire partagée) dont la grandeur est une puissance de deux et effectuer l'opération sur cette mémoire tampon. Lors de la copie, les éléments excédentaires de la mémoire tampon sont initialisés à une valeur par défaut qui n'affecte pas les calculs. Par exemple, dans le cas d'une réduction parallèle calculant la somme, ces éléments doivent être initialisés à "0".

4.8 Conclusion

Dans ce chapitre, nous avons discuté de l'histoire des GPU et expliqué comment ces processeurs ont évolué pour permettre les calculs scientifiques. Équipés d'un très grand nombre de cœurs, les GPU bénéficient d'une puissance de calculs et d'une bande passante à la mémoire bien supérieure à celles des CPU. Leur architecture

massivement parallèle est relativement complexe, mais peut être abstraite par le modèle de programmation et le modèle de la mémoire. Cependant, pour maximiser la performance d'une application parallèle CUDA™ il faut tenir compte des détails de l'architecture du GPU et suivre certaines stratégies de programmation telles que maximiser le parallélisme exploité, minimiser la divergence d'exécution, minimiser les transferts sur le bus PCI Express, maximiser les accès coalescents à la mémoire globale, maximiser l'utilisation de la mémoire partagée et maximiser le taux d'occupation. Pour atteindre un degré de parallélisme suffisant, il faut exploiter le parallélisme au niveau des données. Chaque opération du programme séquentiel original doit être analysée et substituée, lorsque possible, par une primitive parallèle qui effectue le même travail, en parallèle, en moins d'étapes. Dans ce chapitre, nous avons présenté quatre primitives fondamentales. Soit la fonction de map, la réduction, le scan et la dispersion. Nous avons ensuite expliqué comment ces primitives fondamentales peuvent être utilisées pour implémenter les opérations de compaction, de réduction segmentée et de tri. Les implémentations ont été faites de façon à respecter les stratégies de programmation discutées afin qu'elles soient adaptées le mieux possible à l'architecture du GPU pour obtenir une performance supérieure. Ces primitives parallèles sont utilisées aux chapitres suivants pour le développement de métaheuristiques parallèles sur GPU, pour la minimisation des harmoniques d'un onduleur multiniveau, pour l'optimisation de l'écoulement de puissance et pour la reconfiguration de réseaux de distribution.

Chapitre 5
Cadriciel pour métaheuristiques sur GPU

Les travaux de recherche complétés dans le cadre de ce chapitre ont été publiés dans les articles suivants :

V. Roberge, M. Tarbouchi, and F. Okou, "*gpuMF: A Framework for Parallel Hybrid Metaheuristics on GPU with application to the Minimization of Harmonics in Multilevel Inverters*," International Journal on Process System Engineering, vol. 3, no. 1/2/3, pp. 20–41, 2015.

V. Roberge, M. Tarbouchi, and F. Okou, "*Collaborative Parallel Hybrid Metaheuristics on Graphics Processing Unit*," International Journal of Computational Intelligence and Applications, vol. 14, no. 01, p. 1550002, Mar. 2015.

V. Roberge, M. Tarbouchi, and F. Okou, "*gpuMF: A Framework for Parallel Hybrid Metaheuristics on GPU with application to the Minimization of Harmonics in Multilevel Inverters*," in proceedings of the International Conference on Smart Energy Grid Engineering, Oshawa, Canada, 2014.

V. Roberge, M. Tarbouchi, and F. Allaire, "*Parallel Hybrid Metaheuristic on Shared Memory System for Real-Time UAV Path Planning*," International Journal of Computational Intelligence and Applications, vol. 13, no. 2, pp. 1450008–1 – 1450008–16, Jun. 2014.

V. Roberge and M. Tarbouchi, "*Comparison of Parallel Metaheuristics for Flux Optimization for Induction Motor*," World Scientific and Engineering Academy and Society Transactions on Power Systems, vol. 9, pp. 352–359, 2014.

V. Roberge and M. Tarbouchi, "*Comparison of Parallel Particle Swarm Optimizers for Graphical Processing Units and Multicore Processors*," International Journal of Computational Intelligence and Applications, vol. 12, no. 01, p. 135006, Mar. 2013.

V. Roberge, M. Tarbouchi, and G. Labonte, "*Comparison of Parallel Genetic Algorithm and Particle Swarm Optimization for Real-Time UAV Path Planning*," IEEE Transactions on Industrial Informatics, vol. 9, no. 1, pp. 132–141, Feb. 2013

V. Roberge and M. Tarbouchi, "*Parallel Particle Swarm Optimization on Graphical Processing Unit for Pose Estimation*," World Scientific and Engineering Academy and Society Transactions on Computers, Issue 6, Volume 11, June 2012, pp. 170–179.

V. Roberge, M. Tarbouchi, and G. Labonté, "*Parallel Implementation and Comparison of Two UAV Path Planning Algorithms*," in proceedings of the International Conference on Evolutionary Computation Theory and Applications (ECTA 2011), Paris, France, 24-26 Oct 2011, pp. 162-167.

5.1 Introduction

Les métaheuristiques sont des algorithmes d'optimisation intrinsèquement parallèles et leur exécution peut être accélérée sur des systèmes à architecture parallèle. Des implémentations parallèles ont été antérieurement proposées sur processeurs multicœurs [25], sur réseaux d'ordinateurs [26] ou même sur des superordinateurs [27]. Depuis l'arrivée du langage CUDA™ en 2007, plusieurs se sont tournés vers le développement de métaheuristiques parallèles sur GPU. Les processeurs graphiques ont l'avantage d'offrir une architecture massivement parallèle, typique des superordinateurs, mais intégrée sur une seule puce. Malgré que l'accent soit souvent mis sur l'accélération obtenue, les détails d'implémentation ainsi que les tests de validation restent souvent peu convaincants. Le niveau de parallélisme exploité varie et les pratiques de programmation identifiées à la section 4.6, qui permettent d'adapter le programme à l'organisation matérielle du GPU, sont rarement respectées affectant ainsi la performance du programme logiciel. De plus, contrairement aux implémentations sur CPU, il n'existe aucun cadriciel pour métaheuristiques sur GPU qui permet de paralléliser l'algorithme en entier. Les outils actuels sont développés pour le CPU et utilisent le processeur graphique uniquement pour quelques opérations telles que l'évaluation de la fonction de coût. Le développement d'un cadriciel permettait de synthétiser, d'intégrer et d'améliorer les derniers avancements sur les métaheuristiques sur GPU dans un outil logiciel qui serait extensible, adaptable à plusieurs problèmes et réutilisable par d'autres.

Pour ces raisons, nous proposons dans ce chapitre *gpuMF* (ou *GPU Metaheuristic Framework*), un cadriciel pour les métaheuristiques parallèles sur GPU. La structure du cadriciel est basée sur les solutions actuelles pour CPU, mais améliorée de façon à :

1. Être adaptée à l'organisation matérielle du GPU. C'est-à-dire :
    a. maximiser le parallélisme exploité;
    b. minimiser la divergence d'exécution;
    c. minimiser les transferts sur le bus PCI Express;
    d. assurer la coalescence des données en mémoire;
    e. maximiser l'utilisation de la mémoire partagée; et
    f. maximiser le taux d'occupation des multiprocesseurs.

2. Permettre l'utilisation d'algorithmes hybrides afin d'offrir une méthode d'optimisation plus robuste face à une large gamme de problèmes.

3. Permettre une exécution sur GPU, mais aussi sur CPU. À cause de la complexité reliée à la programmation parallèle, la pratique commune est de d'abord développer une version séquentielle du programme avant de le paralléliser. Ceci permet au programmeur de toujours vérifier que l'algorithme parallèle génère le bon résultat avant d'optimiser son accélération. Étant donné que l'un des buts primaires du cadriciel proposé est de définir la structure nécessaire pour permettre le développement futur d'algorithmes, il est essentiel que notre cadriciel offre au programmeur la possibilité de vérifier la justesse de son implémentation parallèle à chaque étape du développement. Nous voulons donc que notre cadriciel permette à la fois le développement de modules séquentiels sur CPU et parallèles sur GPU.

4. Paralléliser toutes les étapes de l'algorithme d'optimisation.

5. Offrir une séparation claire entre l'outil et le problème afin d'être facilement réutilisable pour une large gamme de problèmes.

6. Offrir une structure logique afin de faciliter le développement d'algorithmes futurs.

7. Inclure au minimum une implémentation de l'algorithme d'optimisation par essaim de particules (PSO de l'anglais *particle swarm optimization*) et de l'algorithme génétique (GA de l'anglais *genetic algorithm*). Ces deux métaheuristiques ont été choisies puisqu'elles sont historiquement à la base de plusieurs autres métaheuristiques plus récentes. En parallélisant ces deux algorithmes, nous suggérons qu'il est aussi possible de paralléliser plusieurs autres métaheuristiques.

Dans ce chapitre, nous présentons d'abord l'architecture du cadriciel proposé à l'aide de diagrammes UML (de l'anglais *Unified Modeling Language*). Nous enchaînons ensuite avec trois exemples pour démontrer l'architecture fonctionnelle du cadriciel. Le premier exemple illustre comment on peut créer manuellement les modules nécessaires pour exécuter le PSO sur le CPU. Cette approche manuelle est adéquate pour l'exécution de métaheuristiques simples (composées de quelques modules). Dans le deuxième exemple, un fichier de configuration en langage à balise extensible (XML de l'anglais *Extensible Markup Language*) est utilisé pour définir un GA qui est ensuite créé et exécuté automatiquement par le cadriciel sur le GPU. Cette deuxième approche facilite l'exécution de métaheuristiques plus complexes (composées de plusieurs modules). Finalement, nous présentons un troisième exemple où un fichier de configuration XML est utilisé pour décrire et exécuter une métaheuristique hybride PSO-GA sur le GPU. Par la suite, pour valider le cadriciel

proposé, nous le testons à l'aide de six fonctions tests avec différentes dimensions. Nous vérifions d'abord que nos implémentations séquentielles sur CPU donnent des résultats comparables à d'autres implémentations publiées dans la littérature. Par la suite, nous démontrons que nos versions séquentielles sur CPU et parallèles sur GPU génèrent les mêmes solutions afin de valider notre parallélisation. Finalement, nous mesurons les temps d'exécution des métaheuristiques développées afin de calculer leur accélération. Étant donné que les métaheuristiques sont des algorithmes non déterministes, leur résultat n'est pas nécessairement identique à chaque test. Il faut donc baser notre comparaison sur plusieurs essais et comparer la moyenne obtenue en effectuant un *test T* [241] pour vérifier si la différence entre les moyennes est statistiquement significative.

5.2 Architecture du cadriciel

gpuMF est un cadriciel pour les métaheuristiques parallèles sur GPU. Similaire à une bibliothèque logicielle, le cadriciel *gpuMF* inclut des implémentations complètes du PSO et du GA, mais définit aussi la structure logicielle nécessaire pour faciliter le développement futur d'autres métaheuristiques. L'architecture du cadriciel *gpuMF* est présentée aux différents diagrammes UML qui suivent. Le diagramme de classes de haut niveau est présenté à la Figure 5.1. Ce diagramme définit les modules qui composent le cadriciel ainsi que leurs relations. Tout comme le cadriciel *jMetal* [72] pour exécution sur CPU, *gpuMF* définit une classe *Algorithm* qui utilise des opérateurs afin de modifier, améliorer et optimiser des solutions candidates à un problème donné. La différence entre *gpuMF* et *jMetal* est que *gpuMF* utilise une seule classe pour représenter l'ensemble des solutions candidates et ne sépare pas chaque dimension de chaque solution dans des objets différents comme le fait *jMetal*. Ceci assure que toutes les données soient placées séquentiellement en mémoire et permet à *gpuMF* d'exploiter le parallélisme au niveau des données, ce qui est essentiel pour une implémentation sur GPU. Les classes à la Figure 5.1 sont pour la plupart abstraites (imprimées en *italique* sur le diagramme). Ces classes ne peuvent être instanciées, mais définissent les attributs et les méthodes communes aux classes dérivées, soit celles qui héritent de la classe abstraite. Nous présentons à la Figure 5.2 les classes dérivées de *Operator*.

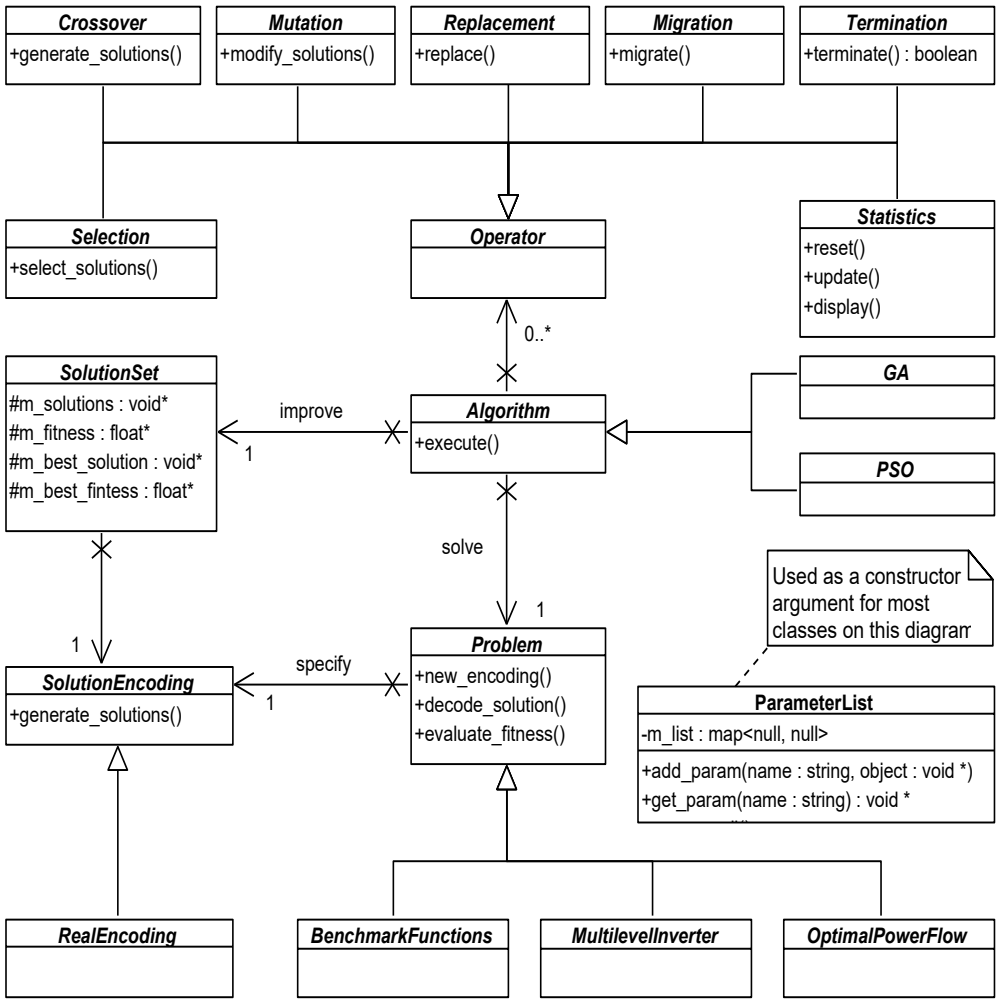


Figure 5.1: Diagramme UML de classes de l'architecture de haut niveau du cadriciel *gpuMF*

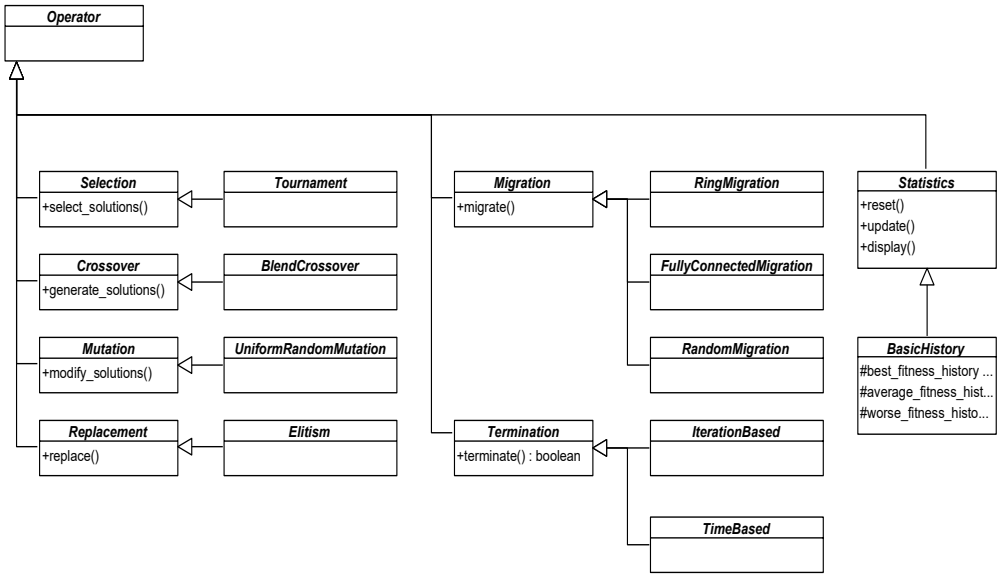


Figure 5.2: Diagramme UML des classes dérivées de *Operator*

Les classes à la Figure 5.2 sont aussi des classes abstraites et ne peuvent être instanciées. Elles définissent les interfaces (soit les fonctions que l'utilisateur peut appeler) pour chacun des opérateurs, mais ne définissent pas les implémentations puisque celles-ci varient pour une exécution sur le CPU ou le GPU. Nous présentons ces implémentations pour quelques-unes des classes à la Figure 5.3. Il est important de noter qu'une telle spécialisation pour le CPU et le GPU est entièrement innovatrice et n'a jamais été implémentée dans un cadriciel, pas même dans *ParadisEO-MO-GPU* [41].

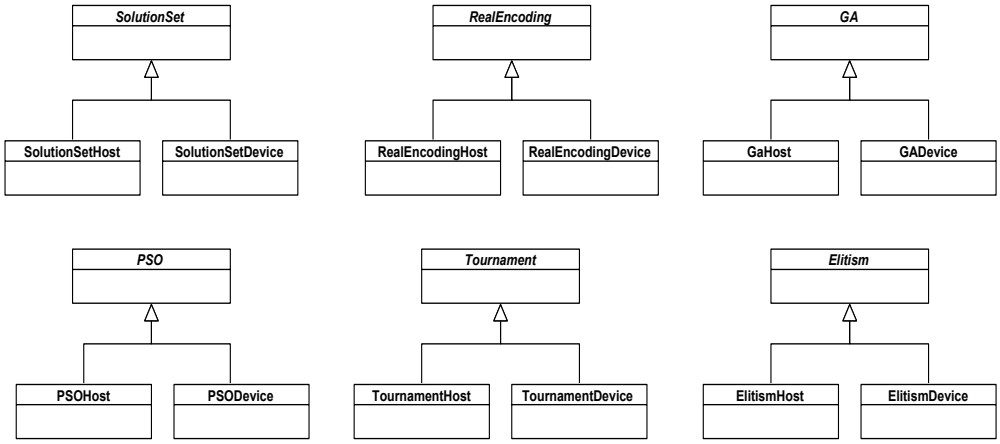


Figure 5.3: Implémentation sur CPU et GPU pour chacune des classes abstraites

Afin de permettre l'utilisation d'algorithmes hybrides composés de métaheuristiques différentes qui exécutent en parallèle et qui collaborent en échangeant leurs solutions candidates, nous avons ajouté trois classes à notre architecture de base présentée précédemment à la Figure 5.1. Ces classes sont encadrées en pointillé sur la Figure 5.4. La classe *Optimizer* utilise un ou plusieurs *Algorithm* afin de modifier et d'optimiser les solutions du *SolutionSetContainer*. Ce dernier est composé d'un *SolutionSet* par *Algorithm*, mais assure que toutes les solutions des *SolutionSet* soient placées séquentiellement en mémoire. Chaque *Algorithm* optimise donc son propre *SolutionSet*. L'avantage du *SolutionSetContainer* est qu'il permet d'utiliser les mêmes opérateurs de migration pour échanger les solutions à l'intérieur d'un algorithme (à l'intérieur du *SolutionSet*) ou entre les algorithmes (entre les *SolutionSet*). L'architecture de *gpuMF* est hautement modulaire ce qui facilite la réutilisation des modules d'une métaheuristique à l'autre. Cependant, exécuter un algorithme en *gpuMF* demande de créer et d'assembler manuellement chacun des modules ce qui peut être long et propice aux erreurs pour une métaheuristique hybride dues au nombre et à l'ordre des modules à créer. Pour éviter ce problème, la classe *Optimizer* lit un fichier de configuration XML dont nous avons défini le format avant de créer et d'assembler automatiquement tous les modules. Notre implémentation utilise le patron de conception logiciel de la *Fabrique abstraite* pour créer les modules sur le champ lors de l'exécution du programme au lieu de le

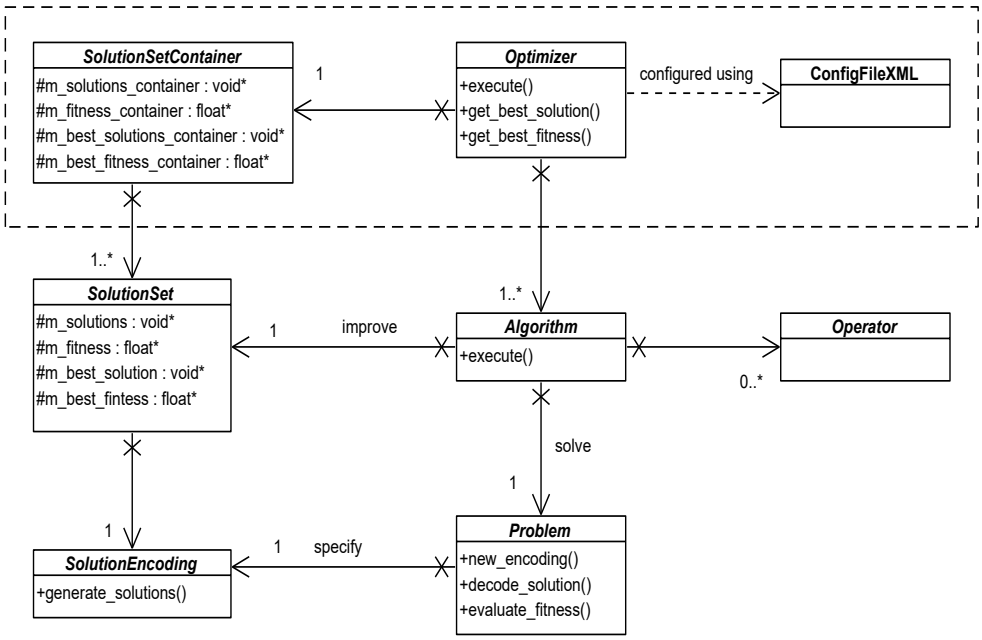


Figure 5.4: Classes permettant la création d'algorithmes hybrides

faire au moment de la compilation. Tel qu'illustré à la Figure 5.5, nous implémentons la *Fabrique abstraite* à l'aide de deux fabriques, soit une pour exécution sur CPU et l'autre, pour le GPU. Un utilisateur peut donc écrire un seul fichier de configuration pour exécuter des métaheuristiques sur le CPU ou le GPU. Le module *Optimizer* s'assurera d'utiliser la bonne fabrique lors de la création des objets.

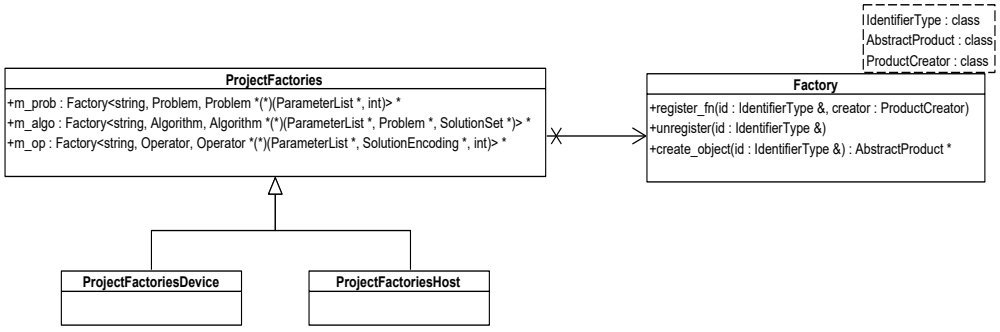


Figure 5.5: Classes permettant la création des objets lors de l'exécution

5.3 Comportement du cadriciel

La section précédente décrit l'architecture statique du cadriciel, soit les différents modules logiciels qui composent *gpuMF*. Dans cette section, nous présentons plutôt une vue dynamique du système et expliquons comment les modules présentés précédemment interagissent lors de l'exécution d'une métaheuristique. Pour ce faire, nous présentons trois exemples d'utilisation du cadriciel et expliquons pour chacun le comportement interne du logiciel. Le premier exemple permet d'expliquer les interactions de haut niveau entre les objets logiciels. Le deuxième exemple décrit les calculs à bas niveau effectué en parallèle sur le GPU. Finalement, le troisième exemple illustre la capacité de *gpuMF* à exécuter des métaheuristiques hybrides où plusieurs algorithmes exécutent concurremment, chacun sur leur propre ensemble de solutions candidates tout en collaborant en échangeant leurs solutions par un processus de migration. Ce type d'algorithme hybride coopératif a été utilisé avec succès sur CPU par les auteurs de [43]. Dans cette section, nous montrons comment *gpuMF* implémente cette approche sur le GPU afin d'exploiter une puissance de calcul bien supérieure.

5.3.1 Exemple 1 : Approche manuelle pour PSO sur CPU

Dans le premier exemple, dont le code est présenté à la Figure 5.6, nous créons manuellement les modules nécessaires pour exécuter le PSO sur le CPU afin d'optimiser une solution à la fonction hypersphère à 20 dimensions. Cette fonction test est définie plus loin à la section 5.4.1. Dans ce code, nous créons d'abord à la

ligne 13 un objet *ParameterList* qui est utilisé pour faciliter la construction des autres objets. Nous créons ensuite une instance de *BenchmarkFunctionHost*. Cette classe définit le problème à optimiser et contient le code pour calculer la fonction hypersphère sur le CPU. Cette classe sera appelée par la métaheuristique pour évaluer la qualité des solutions candidates à chaque itération du processus d'optimisation. Il est intéressant de noter que le code exemple spécifie un nombre de threads égal à quatre lors de la création de la fonction hypersphère. Ceci permet d'utiliser la capacité multithread du cadriciel *gpuMF* afin de bénéficier d'une meilleure performance sur un CPU multicœur. La classe *BenchmarkFunctionHost* est aussi utilisée à la ligne 25 pour obtenir une instance de *SolutionEncoding*, soit l'encodage utilisé par la fonction de coût. Dans le cas de *BenchmarkFunctionHost*, l'encodage consiste en un vecteur de nombre réel normalisé où chaque dimension varie entre 0 et 1. En spécifiant l'encodage à l'intérieur de la classe *BenchmarkFunctionHost*, nous permettons une séparation claire entre l'algorithme d'optimisation et le problème à résoudre. La métaheuristique exécute simplement sur des vecteurs de nombre réels tandis que la classe *BenchmarkFunctionHost* s'occupe de décoder les solutions candidates et d'évaluer leur qualité. Dans le cas de la fonction hypersphère, ce décodage est relativement simple et consiste à modifier l'échelle du vecteur pour que chaque dimension varie entre -5.12 et 5.12. Ces limites ont été choisies afin de permettre la comparaison à la section 5.4.2 entre nos résultats et ceux publiés dans la référence [242]. Malgré que le décodage effectué pour la fonction hypersphère soit simple, cette opération peut être très complexe comme nous le verrons lors de l'utilisation de *gpuMF* pour l'optimisation des réseaux électriques intelligents aux chapitres suivants. Après avoir obtenu l'encodage, nous créons l'ensemble des solutions candidates à la ligne 26. Ces solutions seront modifiées et améliorées par la métaheuristique. Nous créons ensuite le module de terminaison à la ligne 31. Le module choisi ici permettra à la métaheuristique de s'arrêter lorsque le nombre d'itérations spécifié sera atteint. Ce module est passé en paramètre lors de la création de l'algorithme PSO à la ligne 36. À ce moment, tous les modules nécessaires pour exécuter le PSO sont instanciés. La structure du programme en exécution est illustrée à la Figure 5.7. En préparation pour l'exécution, nous initialisons aléatoirement les solutions candidates à la ligne 39 et exécutons l'algorithme à la ligne 40. Après l'exécution, nous affichons à l'écran la solution finale trouvée par le PSO ainsi que sa qualité aux lignes 43 à 46. Finalement, nous détruisons les objets créés avant de terminer l'exécution du programme.

```
1     // Problem settings
2     int fn = 1;                     // hypersphere function
3     int dim = 20;                   // dimension of the problem
4     real lower_limit = (real)-5.12; // search space lower bound
5     real upper_limit = (real)5.12;  // search space upper bound
6     int num_threads = 4;            // number of OpenMP® threads
7
8     // PSO metaheuristic settings
9     int num_solutions = 512;        // number of candidate solutions used
10    int max_iterations = 800;       // number of iterations used
11
12    // Create a list of parameters used to configure the different objects
13    ParameterList *param = new ParameterList();
14
15    // Create the problem object on the HOST
16    param->remove_all();
17    param->add_param("fn", &fn);
18    param->add_param("dim", &dim);
19    param->add_param("lower_limit", &lower_limit);
20    param->add_param("upper_limit", &upper_limit);
21    param->add_param("num_threads", &num_threads);
22    Problem *problem = new BenchmarkFunctionHost(param, num_solutions);
23
24    // Create the solutions set object on the HOST
25    SolutionEncoding *encoding = problem->new_encoding(num_solutions, rand());
26    SolutionSet *solutions = new SolutionSetHost(encoding, num_solutions);
27
28    // Create the termination object on the HOST
29    param->remove_all();
30    param->add_param("max_iterations", &max_iterations);
31    Operator *terminator = new IterationBased(param, encoding, num_solutions);
32
33    // Create the metaheuristic object on the HOST
34    param->remove_all();
35    param->add_param("terminator", terminator);
36    Algorithm *pso = new PSOHost(param, problem, solutions);
37
38    // Execute the algorithm on the HOST
39    solutions->generate_initial_solutions();
40    pso->execute();
41
42    // Print the final solution
43    fprintf(stdout, "The best solutions is: \n");
44    for (int i = 0; i < dim; i++)
45        fprintf(stdout, "%f\n", ((real*)solutions->get_best_solution())[i]);
46    fprintf(stdout, "Fitness: %f\n", *solutions->get_best_fitness());
47
48    // Free memory
49    ...
```

Figure 5.6: Code C++ utilisant *gpuMF* pour exécuter le PSO sur le CPU afin d'optimiser une solution à la fonction hypersphère à 20 dimensions

Le diagramme UML à la Figure 5.7 montre les objets instanciés nécessaires à l'exécution du PSO sur CPU aussi que leurs relations. On peut voir que l'objet *SolutionSetHost* est connecté à *RealEncodingHost* ce qui lui permet d'utiliser

ce dernier pour initialiser aléatoirement les solutions candidates en début d'exécution. L'objet *PSOHost* évalue à chaque itération du processus d'optimisation la qualité des solutions candidates à l'aide de l'objet *BenchmarkFunctionHost* pour les modifier et les améliorer d'après la stratégie d'optimisation du PSO. Finalement, l'objet *PSOHost* utilise *IterationBasedHost* pour vérifier si le nombre d'itérations maximal a été atteint avant de terminer l'exécution. Dépendamment de l'application, ce dernier module peut être remplacé par d'autres modules de terminaison afin de, par exemple, terminer l'exécution après un temps de calcul fixe ou lorsque la qualité de la meilleure solution stagne et cesse d'être améliorée.

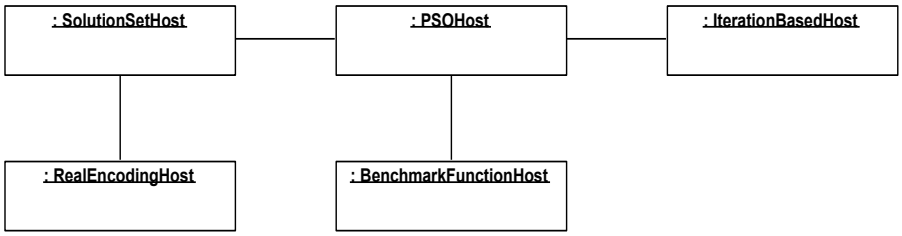


Figure 5.7: Diagramme UML d'objets pour le PSO sur CPU

Le comportement du cadriciel *gpuMF* pour l'exécution du PSO sur le CPU est illustré sur le diagramme UML de séquence à la Figure 5.8. Ce diagramme est une représentation séquentielle des interactions entre les différents objets logiciels décrits ci-dessus. Les messages 1 à 5 permettent la création des objets. Le message 6 provoque l'initialisation des solutions. On voit ensuite les détails du fonctionnement de PSO à l'intérieur de la boucle déclenchée par le message 7. Finalement, le message 8 permet d'afficher le résultat à l'écran. Ceci conclut la description du comportement du cadriciel *gpuMF* pour l'exécution du PSO sur CPU. Ce premier exemple sert à expliquer les interactions de haut niveau entre les objets logiciels. Dans le prochain exemple, nous regarderons comment *gpuMF* effectue les calculs à bas niveau pour exploiter l'architecture massivement parallèle du GPU.

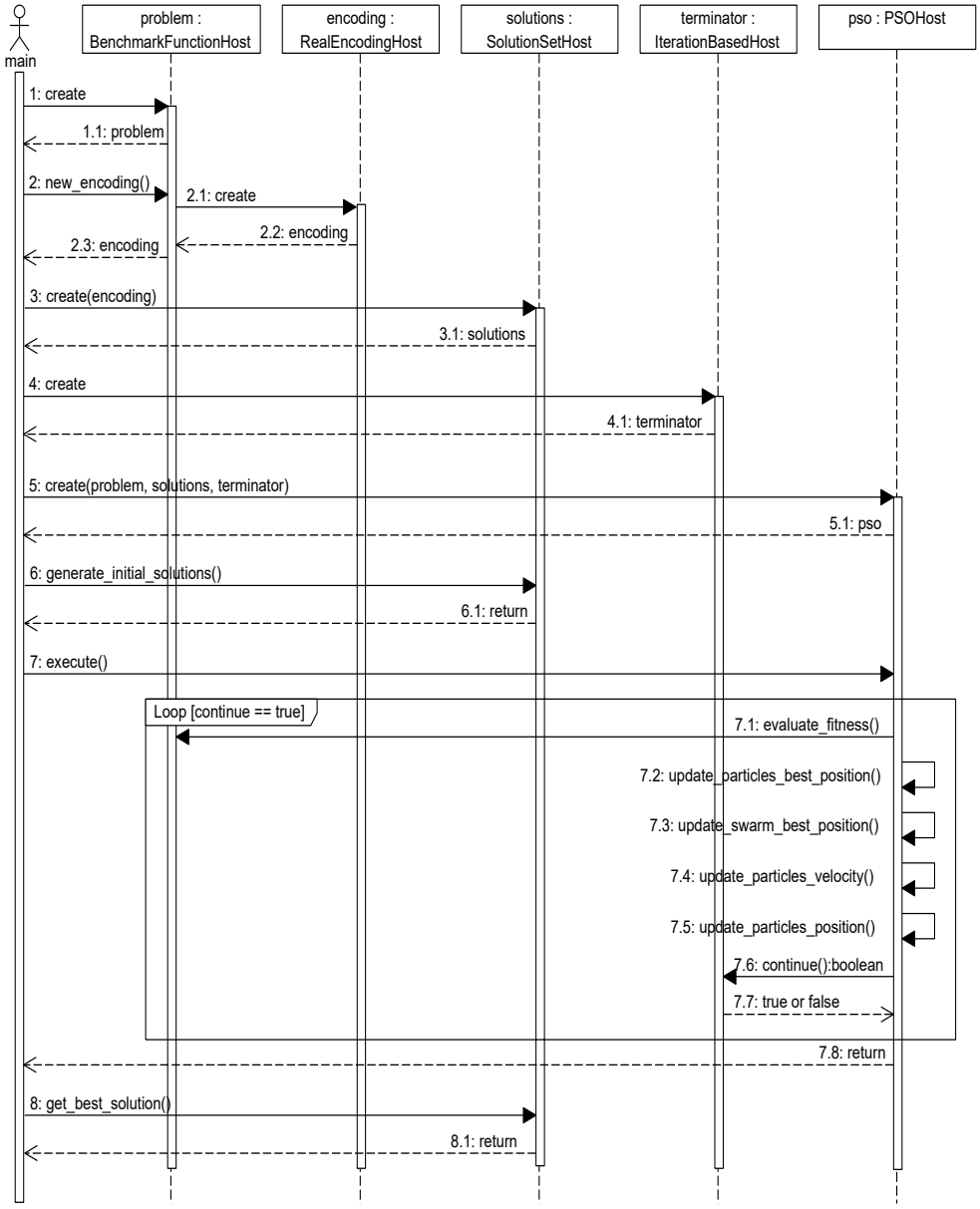


Figure 5.8: Diagramme UML de séquence illustrant l'exécution du PSO sur CPU

5.3.2 Exemple 2 : Approche automatique pour GA sur GPU

Dans le second exemple, nous considérons toujours le problème de l'hypersphère à 20 dimensions, mais cette fois à l'aide d'un GA parallèle sur GPU. Nous utilisons le fichier XML à la Figure 5.9 pour configurer le GA. Ce fichier ne contient pas de

code exécutable, mais uniquement des directives de configuration qui sont lues par le cadriciel *gpuMF* au moment de l'exécution. Le code exécutable se trouve en fait à la Figure 5.10. Comparativement à l'exemple précédent, ce code est plus court puisqu'il instancie simplement à la ligne 24 un objet *Optimizer* qui à son tour lit le fichier de configuration XML et assemble automatiquement les modules du GA. La structure du programme après la création automatique des objets est illustrée à la Figure 5.11. En raison de sa nature, on remarque que le GA contient beaucoup plus de modules que le PSO. La capacité de *gpuMF* à utiliser un fichier de configuration XML est ici avantageuse puisqu'elle évite d'avoir à créer manuellement chacun de ces modules. De retour au code exemple à la Figure 5.10, le GA est exécuté sur l'architecture parallèle du GPU à la ligne 28. Par après, puisque l'ensemble des solutions candidates incluant la solution finale se trouve dans l'espace mémoire du GPU, le programme doit transférer aux lignes 33 et 35 la solution finale ainsi que sa qualité vers la mémoire du CPU avant de les afficher à l'écran aux lignes 39 à 42.

```
1   <configuration>
2
3     <algorithm class="GA">
4
5       <operator type="selection" class="Tournament">
6         <parameter type="int" name="pool_size">2</parameter>
7       </operator>
8
9       <operator type="crossover" class="BlendCrossover">
10        <parameter type="real" name="alpha">0.5</parameter>
11      </operator>
12
13      <operator type="mutation" class="UniformRandomMutation">
14        <parameter type="real" name="probability">0.1</parameter>
15        <parameter type="real" name="range">0.05</parameter>
16      </operator>
17
18      <operator type="replacement" class="Elitism">
19        <parameter type="int" name="num_survivors">4</parameter>
20      </operator>
21
22      <terminator class="IterationBased">
23        <parameter type="int" name="max_iterations">400</parameter>
24      </terminator>
25
26    </algorithm>
27
28  </configuration>
```

Figure 5.9: Fichier de configuration XML pour le GA

```
    // Problem settings
    int fn = 1;                         // hypersphere function
    int dim = 20;                       // dimension of the problem
    real lower_limit = (real)-5.12;     // search space lower bound
    real upper_limit = (real)5.12;      // search space upper bound
    int block_size = 128;               // number of thread per CUDA™ block

    // PSO metaheuristic settings
    int num_solutions = 512;           // number of candidate solutions used

    // Create a list of parameters used to configure the different objects
    ParameterList *param = new ParameterList();

    // Create the problem object on the DEVICE
    param->remove_all();
    param->add_param("fn", &fn);
    param->add_param("dim", &dim);
    param->add_param("lower_limit", &lower_limit);
    param->add_param("upper_limit", &upper_limit);
    param->add_param("block_size", &block_size);
    Problem *problem = new BenchmarkFunctionDevice(param, num_solutions);

    // Create the optimizer on the DEVICE
    Optimizer *optimizer = new OptimizerDevice("ga_vpuml.xml", problem,
        num_solutions, 1, time(NULL), block_size);

    // Execute the algorithm on the DEVICE
    optimizer->execute();

    // Transfer the final solution from the DEVICE to the HOST
    real *h_final_soln = (real*)malloc(dim * sizeof(real));
    real *h_final_fitness = (real*)malloc(sizeof(real));
    CUDA™Memcpy(h_final_soln, optimizer->get_best_solution(), dim *
        sizeof(real), CUDA™MemcpyDeviceToHost);
    CUDA™Memcpy(h_final_fitness, optimizer->get_best_fitness(), sizeof(real),
        CUDA™MemcpyDeviceToHost);

    // Print the final solution
    fprintf(stdout, "The best solutions is: \n");
    for (int i = 0; i < dim; i++)
        fprintf(stdout, "%f\n", h_final_soln[i]);
    fprintf(stdout, "Fitness: %f\n", *h_final_fitness);

    // Free memory
    ...
```

Figure 5.10: Code C++ utilisant *gpuMF* pour exécuter le GA sur le GPU afin d'optimiser une solution à la fonction hypersphère à 20 dimensions

Tout comme pour l'exemple précédent, le diagramme UML à la Figure 5.11 représente les objets logiciels nécessaires pour l'exécution du GA sur le GPU. En plus des objets qui composent le GA, on remarque l'addition de l'objet *OptimizerDevice*, de l'objet *ConfigFileXML* et de l'objet *SolutionSetContainerDevice*. L'objet *OptimizerDevice* est responsable de lire le fichier de configuration XML à l'aide de *ConfigFileXML* pour ensuite instancier

automatiquement tous les modules qui composent le GA. Par défaut, *OptimizerDevice* crée un *SolutionSetContainer* qui contient un ou plusieurs *SolutionSetDevice*. Dans cet exemple, un seul *SolutionSetDevice* est instancié puisqu'une seule métaheuristique est utilisée. À l'exemple suivant, nous instancierons plusieurs algorithmes afin de créer une métaheuristique hybride coopérative et chaque algorithme aura son propre ensemble de solutions candidates ou objet *SolutionSetDevice*. Dans cette situation, l'objet *SolutionSetContainerDevice* est nécessaire puisqu'il inclut ses propres routines d'allocation de mémoire afin d'assurer que tous les ensembles de solutions candidates soient placés séquentiellement dans la mémoire globale du GPU pour garantir un accès coalescent aux données, une caractéristique essentielle pour une implémentation performante sur GPU.

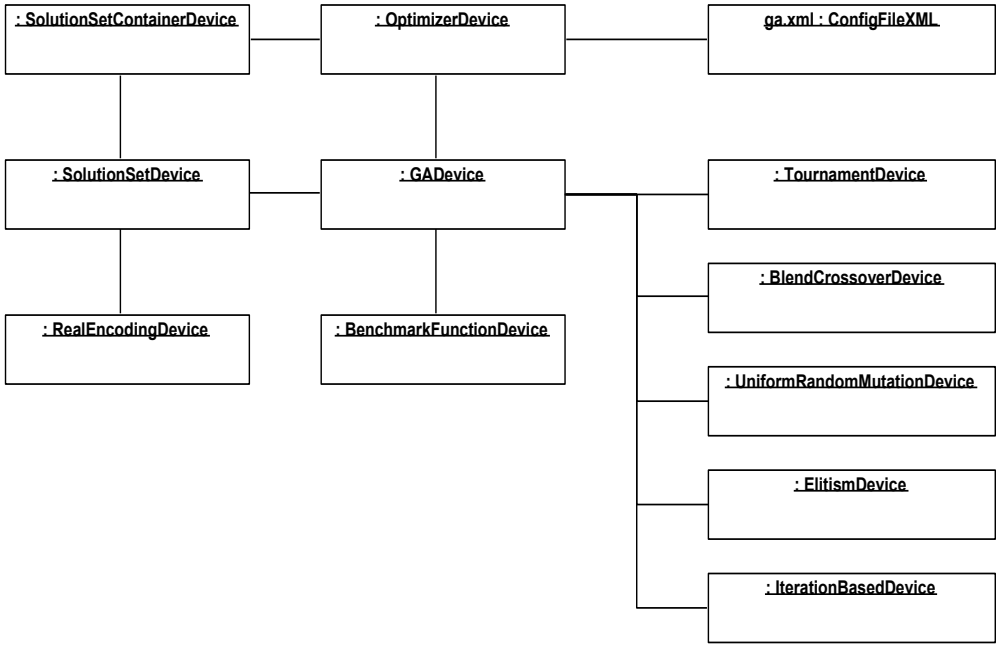


Figure 5.11: Diagramme UML d'objets pour le GA sur GPU

Le comportement de *gpuMF* pour l'exécution du GA sur le GPU est illustré au diagramme UML à la Figure 5.12. Ce diagramme de séquence montre l'ordre des interactions entre les différents objets logiciels. Pour simplifier le diagramme et nous concentrer sur l'exécution du GA, nous avons omis les étapes nécessaires à la création des objets, à l'initialisation des solutions candidates et à la destruction des objets. Nous avons aussi utilisé une appellation abrégée pour nommer les objets afin de réduire la taille de la figure. Sur ce diagramme, on remarque la boucle qui est effectuée pour un nombre fixe d'itérations spécifié dans le fichier de configuration XML. Chaque itération consiste à évaluer la qualité des solutions candidates, choisir des

paires de solutions parents, générer des solutions enfants par enjambement, muter les solutions enfants et finalement, remplacer la population de solutions parents par celle des enfants. D'après le fichier de configuration, la sélection des parents s'effectue par tournois [56]. Pour sélectionner un parent, deux solutions candidates sont choisies au hasard et celle avec la meilleure qualité est sélectionnée. Ce processus est répété jusqu'à ce que $n/2$ paires de parents soient choisies où n représente le nombre de solutions candidates. Chaque paire de parents ($\overrightarrow{x^1}$, $\overrightarrow{x^2}$) génère ensuite deux solutions enfants ($\overrightarrow{y^1}$, $\overrightarrow{y^2}$). Dans l'exemple code, les solutions enfants sont générées par enjambement mélangé (de l'anglais *blend crossover*) [56]. L'opération est effectuée indépendamment sur chaque élément (ou dimension) des solutions parents et calcule les éléments des solutions enfants suivant :

$$y_i^1 = rand\left(x_i^1 - \alpha\left(x_i^2 - x_i^1\right), \quad x_i^2 + \alpha\left(x_i^2 - x_i^1\right)\right) \tag{5.1}$$

$$y_i^2 = rand\left(x_i^1 - \alpha\left(x_i^2 - x_i^1\right), \quad x_i^2 + \alpha\left(x_i^2 - x_i^1\right)\right) \tag{5.2}$$

où x_i^1 est l'élément i du parent 1 et x_i^2 est l'élément i du parent 2. Similairement, y_i^1 est l'élément i de l'enfant 1 et y_i^2 est l'élément i de l'enfant 2. La constante α est ici égale à 0.5 et définit les limites possibles pour y_i^1 et y_i^2. L'opérateur $rand(a, b)$ génère un nombre aléatoire suivant une distribution uniforme entre a et b. L'enjambement mélangé est une opération commune souvent utilisée pour un GA lorsque les solutions candidates sont encodées par des vecteurs de nombres réels comme il est le cas dans cet exemple. Après que les solutions enfants soient créées, le GA les modifie par une mutation aléatoire uniforme [56]. D'après le fichier de configuration à la Figure 5.9, chaque élément y_i de chaque solution enfant $\vec{y}$ est muté avec une probabilité de 0.1 et la modification est effectuée aléatoirement comme suit :

$$y_i = rand(y_i - \Delta, \quad y_i + \Delta) \tag{5.3}$$

où Δ représente la variation maximale et l'opérateur $rand(a, b)$ est défini comme à l'équation précédente. Après la mutation, les solutions parents sont remplacées par les solutions enfants et le processus se répète à la prochaine itération. D'après le fichier de configuration XML, ce remplacement intègre le concept d'élitisme [56] où les meilleures solutions parents sont gardées à l'itération suivants au détriment des pires solutions enfants. Dans le GA, les opérations de sélections et d'enjambement permettent de diriger la recherche stochastique vers des solutions de meilleure qualité tandis que l'opération de mutation stimule l'exploration de l'espace de recherche. La stratégie de remplacement avec élitisme permet d'accélérer la convergence et de sauvegarder les meilleures solutions visitées au cours de la recherche.

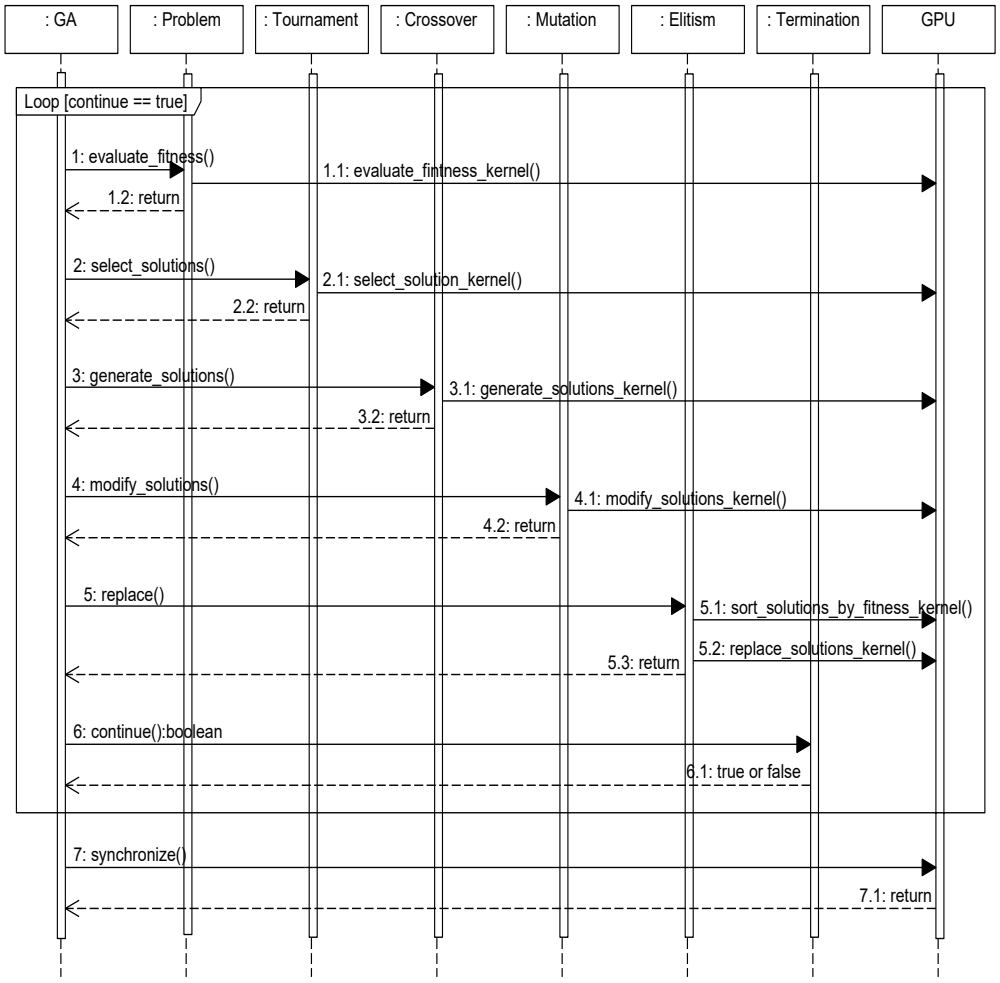


Figure 5.12: Diagramme UML de séquence illustrant l'exécution du GA sur le GPU

Le diagramme UML de séquence à la Figure 5.12 illustre les interactions de haut niveau entre les objets logiciels. On remarque cependant que, lorsqu'appelé, chaque objet appelle à son tour des *kernels* CUDA™ sur le GPU. La séquence des opérations est donc définie au niveau supérieur par l'architecture orientée objet de *gpuMF* tandis que les calculs sont effectués au niveau inférieur, en parallèle, sur le GPU. Cette dualité permet une architecture modulaire facilement utilisable tout en maximisant la performance grâce à la parallélisation ce qui représente l'avantage innovateur du cadriciel *gpuMF*. Pour visualiser les calculs de bas niveau effectué sur le GPU, il faut se référer à la Figure 5.13.

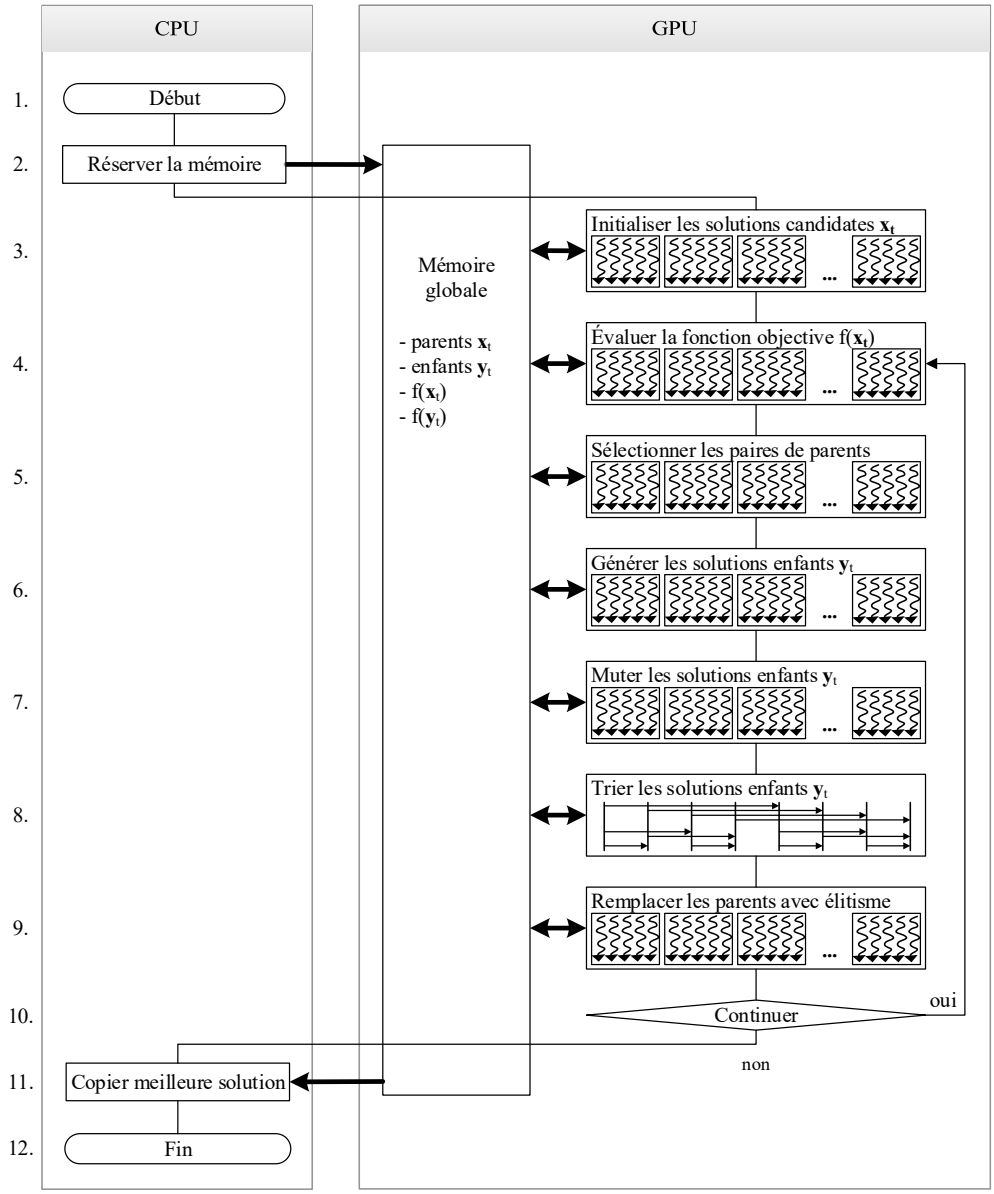


Figure 5.13: Diagramme de flux des calculs parallèles sur le GPU pour le GA

Le programme débute son exécution sur le CPU et alloue la mémoire nécessaire pour les solutions parents et enfants dans l'espace globale du GPU lors de la création des différents objets logiciels. Tous les appels de *kernels* listés à la Figure 5.12 sont ensuite envoyés de façon asynchrone sur la queue du GPU. Ces appels sont non-bloquants ce qui signifie que le CPU continue son exécution jusqu'à ce qu'il atteigne

le point de synchronisation au message 7 de la Figure 5.12. Cette synchronisation assure que tous les *kernels* CUDA™ aient terminé avant que *gpuMF* retourne le contrôle à l'utilisateur. De son côté, le GPU débute l'exécution dès qu'il reçoit son premier appel de *kernel*. Tel qu'illustré à la Figure 5.13, chaque *kernel* est exécuté par une *grid* de blocs de threads CUDA™. Trois primitives parallèles sont utilisées pour implémenter les GA sur GPU : la fonction de map, le tri bitonique et la fonction de dispersion. Les *kernels* aux lignes 3 à 7 utilisent une fonction de map avec un thread par dimension. Puisque nous exécutions ici le GA avec 512 solutions candidates pour la fonction hypersphère à 20 dimensions, le programme lance 10 240 threads CUDA™. Pour des problèmes plus complexes comme ceux aux chapitres 6, 7 et 8, un nombre bien plus grand de threads sera utilisé. Le *kernel* à la ligne 8 utilise un tri bitonique afin d'ordonnancer les solutions candidates d'après leur qualité. Pour éviter de réorganiser physiquement les données en mémoire, l'implémentation du *kernel* 8 génère un vecteur d'index représentant l'ordre des solutions. Finalement, le *kernel* à la ligne 9 utilise une fonction de dispersion afin de copier les solutions enfants tout en gardant les meilleures solutions parents. L'implémentation parallèle sur GPU du GA maximise le parallélisme exploité, minimise la divergence d'exécution, limite les transferts de données sur le bus PCI Express uniquement à la solution finale, permet un accès coalescent à la mémoire globale, exploite au maximum la mémoire partagée à l'intérieur des *kernels* lorsque des données sont utilisées plus d'une fois et maximise le taux d'occupation des multiprocesseurs. Ceci résulte en un programme parallèle hautement adapté à l'architecture du GPU afin de garantir la meilleure performance possible.

L'exemple que nous venons tout juste de discuter permet d'illustrer le lien entre l'architecture orientée-objet du cadriciel et les calculs de bas niveau sur le GPU. La modularité au niveau supérieur permet une configuration flexible de l'outil d'optimisation. Chaque module peut être remplacé afin d'adapter la métaheuristique au problème considéré. Au niveau inférieur, l'organisation des données en mémoire permet d'exploiter le parallélisme au niveau des données pour une exécution performante sur le GPU.

5.3.3 Exemple 3 : Approche automatique pour PSO-GA sur GPU

Dans ce troisième et dernier exemple, nous démontrons la pleine capacité du cadriciel *gpuMF* en exécutant une métaheuristique hybride coopérative et parallèle sur GPU composée du PSO et du GA. Afin d'éviter de créer manuellement chacun des modules qui composent les deux algorithmes d'optimisation, nous utilisons un fichier de configuration XML que nous reproduisons à la Figure 5.14. Dans ce fichier, on peut noter quatre sections.

La première section, identifiée par le mot clé <*définitions*>, définit les paramètres et les modules qui sont communs au PSO et au GA. Ceci permet d'éviter de redéfinir un même module plusieurs fois. On réfère ensuite à ces définitions d'après leur alias unique. Dans cet exemple, nous définissons comme alias le nombre d'îlots, le processus de migration ainsi que la condition de terminaison qui sont tous les trois communs au PSO et au GA. Subdiviser la population de solutions candidates en îlots ou sous-ensembles permet de ralentir la convergence de l'algorithme d'optimisation et d'améliorer sa capacité d'exploration [42].

Dans la deuxième section identifiée par le mot clé <*optimizer*>, nous définissons les modules spécifiques au comportement coopératif de haut niveau de l'algorithme hybride. Nous identifions d'abord le processus de migration qui consiste ici à partager la meilleure solution parmi toutes les métaheuristiques coopératives. Ce partage s'effectue à chaque itération de l'algorithme coopératif où une itération consiste à exécuter le PSO et le GA jusqu'à terminaison, soit 50 itérations. Nous spécifions ensuite le nombre d'itérations maximales pour l'algorithme coopératif. Ce nombre est ici égale à dix ce qui signifie que l'algorithme coopératif exécutera le PSO et le GA à dix reprises.

Finalement, dans les troisièmes et quatrièmes sections identifiées par le mot clé <*algorithm*>, nous définissons les paramètres et modules pour le PSO et GA. Les paramètres utilisés ici sont semblables à ceux utilisés dans les exemples précédents et n'ont pas besoin d'être expliqués à nouveau.

```
1   <configuration>
2
3     <definitions>
4       <parameter type="int" name="num_islands" alias="islands">4</parameter>
5       <migration class="RingMigration" alias="algo_migration_module">
6         <parameter type="int" name="frequency">10</parameter>
7         <parameter type="int" name="num_migrants_per_island">8</parameter>
8       </migration>
9       <terminator class="IterationBased" alias="algo_termination_module">
10         <parameter type="int" name="max_iterations">50</parameter>
11      </terminator>
12    </definitions>
13
14    <optimizer>
15      <migration class="FullyConnectedMigration">
16        <parameter type="int" name="frequency">1</parameter>
17        <parameter type="int" name="num_migrants_per_island">1</parameter>
18      </migration>
19      <terminator class="IterationBased">
20        <parameter type="int" name="max_iterations">8</parameter>
21      </terminator>
22    </optimizer>
23
24    <algorithm class="PSO">
25      <use alias="islands"/>
```

```
26      <use alias="algo_migration_module"/>
27      <use alias="algo_termination_module"/>
28    </algorithm>
29
30    <algorithm class="GA">
31      <use alias="islands"/>
32      <operator type="selection" class="Tournament">
33        <parameter type="int" name="pool_size">2</parameter>
34      </operator>
35      <operator type="crossover" class="BlendCrossover">
36        <parameter type="real" name="alpha">0.5</parameter>
37      </operator>
38      <operator type="mutation" class="UniformRandomMutation">
39        <parameter type="real" name="probability">0.1</parameter>
40        <parameter type="real" name="range">0.05</parameter>
41      </operator>
42      <operator type="replacement" class="Elitism">
43        <parameter type="int" name="num_survivors">4</parameter>
44      </operator>
45      <use alias="algo_migration_module"/>
46      <use alias="algo_termination_module"/>
47    </algorithm>
48
49  </configuration>
```

Figure 5.14: Fichier de configuration XML pour le PSO-GA hybride coopératif

Pour exécuter l'algorithme hybride coopératif sur le GPU, nous utilisons le code à la Figure 5.10, mais spécifions le nombre de métaheuristiques à instancier lors de la création de l'objet *optimizer* comme à la Figure 5.15. Dans cet exemple, nous instancions quatre métaheuristiques. L'objet *optimizer* lira donc le fichier de confirmation XML et itérera au travers des définitions d'algorithmes de façon circulaire jusqu'à ce que toutes les métaheuristiques soient instanciées. Dans notre exemple, puisque nous avons défini deux algorithmes, mais instancions quatre métaheuristiques, l'objet *optimizer* créera dans l'ordre un PSO, un GA, un autre PSO et un autre GA.

```
1     ...
2     // Create the optimizer on the DEVICE
3     int num_metaheuristics = 4;
4     Optimizer *optimizer = new OptimizerDevice("pso_ga.xml", problem,
5         num_solutions, num_metaheuristics, time(NULL), block_size);
6
7     // Execute the algorithm on the DEVICE
8     optimizer->execute();
9     ...
```

Figure 5.15: Code C++ utilisant *gpuMF* pour exécuter le PSO-GA hybride sur le GPU afin d'optimiser une solution à la fonction hypersphère à 20 dimensions

L'organisation logique de l'algorithme hybride est représentée à la Figure 5.16. Chaque métaheuristique opère sur son propre ensemble de solutions candidates qui est subdivisé à l'interne en îlots. Les flèches entre les îlots illustrent le processus de migration exécutée tous les 10 itérations du PSO et du GA suivant une topologie en anneau. Les flèches entre les métaheuristiques représentent quant à eux le processus de migration suivant une topologie entièrement connectée. Cette seconde migration est effectuée tous les 50 itérations des algorithmes PSO et GA.

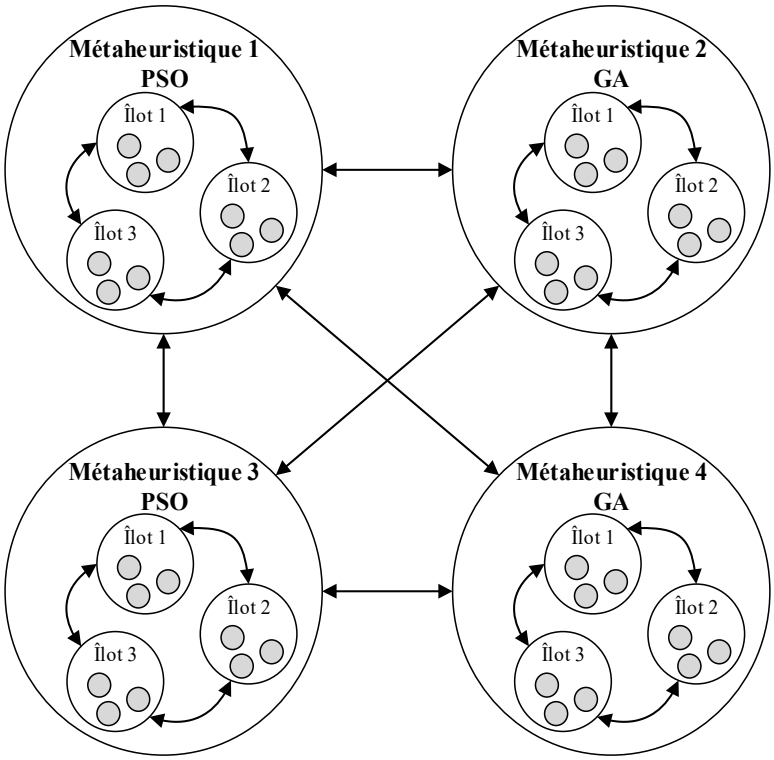


Figure 5.16: Organisation logique des solutions candidates du PSO-GA hybride

Lors de l'exécution du PSO-GA hybride, l'objet *optimizer* lance quatre threads OpenMP® sur le CPU et chaque thread exécute sa propre métaheuristique de façon asynchrone comme à la Figure 5.17. Les threads se synchronisent uniquement avant les migrations inter-métaheuristiques tous les 50 itérations des algorithmes PSO et GA.

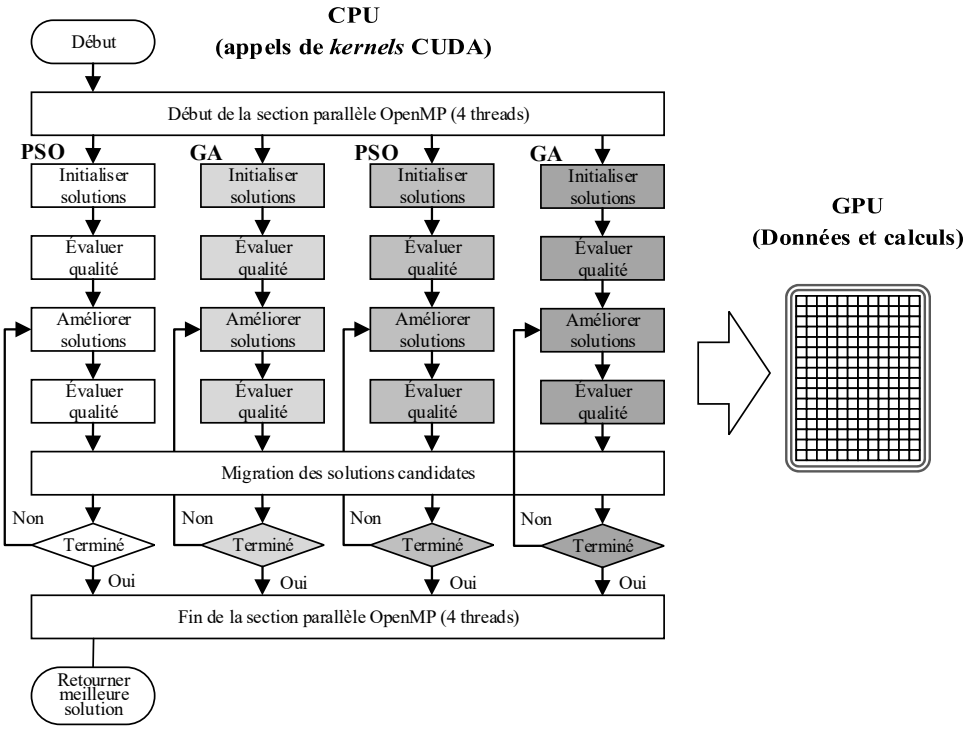


Figure 5.17: Modèle d'exécution du PSO-GA hybride

L'avantage d'utiliser un thread indépendant pour chaque métaheuristique permet d'exploiter la fonctionnalité multiqueue des GPU NVIDIA®. Tel qu'illustré à la Figure 5.18, chaque thread OpenMP® place ses appels de *kernels* dans une queue unique sur le GPU. L'ordonnanceur du GPU assure que l'ordre des *kernels* dans une même queue est respecté, mais permet d'exécuter concurremment des *kernels* provenant de queues différentes lorsque les ressources matérielles le permettent résultant ainsi en une meilleure performance.

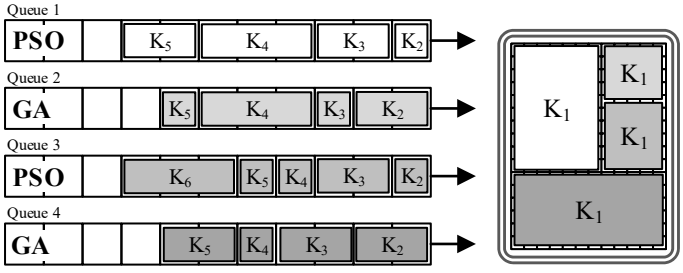


Figure 5.18: Exécution concurrente sur GPU grâce aux queues multiples

Ceci conclut le troisième exemple d'utilisation du cadriciel *gpuMF*. Cet exemple a permis de démontrer la capacité du cadriciel à exécuter des métaheuristiques hybrides coopératives. Ceci est possible grâce aux objets logiciels *Optimizer* et *SolutionSetContainer*. L'objet *Optimizer* permet de créer et d'assembler automatiquement les métaheuristiques définies dans le fichier de configuration XML et de coordonner leur exécution. Quant à lui, l'objet *SolutionSetContainer* permet d'organiser les solutions candidates des multiples métaheuristiques de façon séquentielle dans la mémoire globale du GPU pour maximiser la coalescence des données en mémoire.

5.4 Résultats expérimentaux

Avant d'utiliser le cadriciel *gpuMF* pour l'optimisation des réseaux intelligents, nous devons valider l'outil développé. Dans un premier lieu, nous voulons nous assurer que l'implémentation parallèle sur GPU est correcte et produit un résultat équivalent à une implémentation séquentielle sur CPU. Nous utilisons ici le terme équivalent puisque les résultats ne sont pas nécessairement identiques à cause de la nature non déterministe des métaheuristiques. Pour effectuer cette vérification, nous exécutons l'algorithme à plusieurs reprises et comparons les moyennes des résultats obtenus. Un *test T* [243] est ensuite complété pour identifier si les différences entre les valeurs moyennes sont statistiquement significatives ou non. Dans un second lieu, nous évaluons le gain de performance apporté par l'implémentation parallèle. Ce gain est mesuré en termes d'accélération S définie comme étant le rapport du temps d'exécution séquentielle T_{seq} divisé par le temps d'exécution parallèle T_{par} comme à l'équation suivante :

$$S = \frac{T_{seq}}{T_{par}} \tag{5.4}$$

5.4.1 Fonctions tests

Pour évaluer le cadriciel *gpuMF*, nous utilisons six fonctions tests, soit les fonctions hypersphère (*Sp*), Griewank (*Gr*), Rastrigin (*Ra*), Rosenbrock (*Ro*), Schwefel (*Sw*) et hybride (*Hy*) que nous définissons aux équations (5.5) à (5.10). Ces fonctions ont été choisies de façon à permettre une comparaison avec les implémentations séquentielles du PSO et du GA publiées dans [242]. La dimension d de chaque fonction peut être variée de façon à ajuster la complexité du problème d'optimisation étudié. Tout comme les auteurs de [242], nous normalisons chacune des fonctions entre 0 et 1 en utilisant l'équation (5.11) et restreignons l'espace de recherche entre -5.12 et 5.12 pour chaque dimension.

$$f_{Sp}(\vec{x}) = \sum_{d=1}^{N} x_d^2 \tag{5.5}$$

$$f_{Gr}(\vec{x}) = \frac{1}{4000}\sum_{d=1}^{N} x_d^2 - \prod_{d=1}^{N} \cos\left(\frac{x_d}{\sqrt{d}}\right) + 1 \tag{5.6}$$

$$f_{Ra}(\vec{x}) = \sum_{d=1}^{N} \left(x_d^2 - 10\cos(2\pi x_d) + 10\right) \tag{5.7}$$

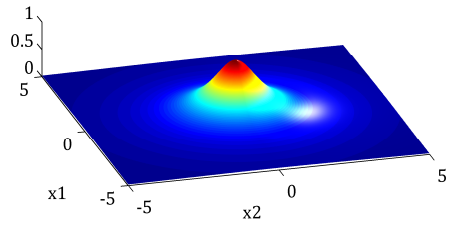
$$f_{Ro}(\vec{x}) = \sum_{d=1}^{N-1} \left(\left(100 * \left(x_{d+1} - x_d^2\right)\right)^2 + (x_d - 1)^2\right) \tag{5.8}$$

$$f_{Sw}(\vec{x}) = \sum_{d=1}^{N} \left(\sum_{j=1}^{d} x_j\right)^2 \tag{5.9}$$

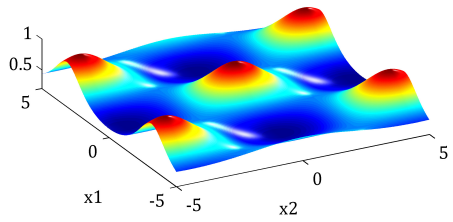
$$f_{Hy}(\vec{x}) = f_{Ra}(\vec{x}) + 2f_{Sw}(\vec{x}) + \frac{1}{12}f_{Gr}(\vec{x}) + \frac{1}{20}f_{Sp}(\vec{x}) \tag{5.10}$$

$$F(\vec{x}) = \frac{1}{1 - f(\vec{x})} \tag{5.11}$$

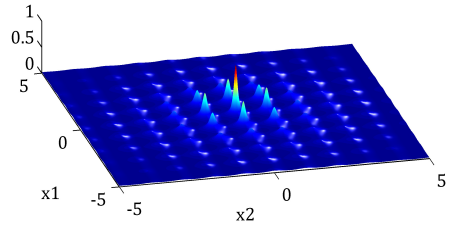
Afin de visualiser la complexité des fonctions tests choisies, nous affichons à la Figure 5.19 leur surface pour le cas spécifique à deux dimensions. On remarque que la valeur maximale pour chaque fonction est égale à 1. L'objectif de l'optimisation effectuée dans notre test de validation est de calculer les valeurs de x_1 et x_2 qui permettent d'obtenir ce maximum. La complexité du problème accroît rapidement lorsque le nombre de dimensions augmente. Les fonctions hypersphère, Rosenbrock et Schwefel sont unimodales (un seul maximum) tandis que les fonctions Griewank, Rastrigin et hybride sont multimodales. À l'exemption de la fonction Rosenbrock, toutes les fonctions démontrent une certaine symétrie.



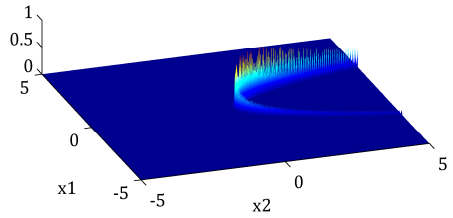
Fonction hypersphère $f_{Sp}(x_1, x_2)$



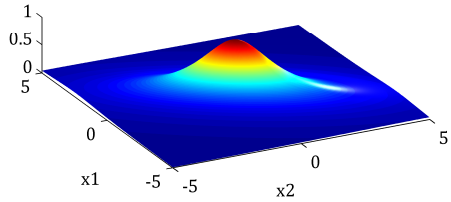
Fonction Griewank $f_{Gr}(x_1, x_2)$



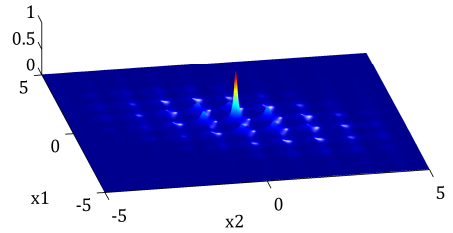
Fonction Rastring $f_{Ra}(x_1, x_2)$



Fonction Rosenbrock $f_{Ro}(x_1, x_2)$



Fonction Schwefel $f_{Sw}(x_1, x_2)$



Fonction hybride $f_{Hy}(x_1, x_2)$

Figure 5.19: Surfaces des fonctions tests pour le cas où la dimension $d = 2$

5.4.2 Exactitude des résultats

Dans un premier test, afin de vérifier que nos implémentations séquentielles et parallèles sont correctes, nous exécutons le PSO, le GA ainsi que le PSO-GA hybride pour calculer des solutions aux six fonctions tests avec différents nombres de dimensions. Chaque test est répété 100 fois et les résultats moyens sont listés au Tableau 5.2. Nous incluons aussi dans ce tableau les résultats obtenus par les auteurs de [242] pour le PSO et le GA (il faut noter que la référence [242] n'inclut pas le PSO-GA hybride). De manière à assurer une comparaison juste, nous configurons dans la mesure du possible nos algorithmes avec les mêmes paramètres que dans la

référence [242]. Ces paramètres sont reproduits ici au Tableau 5.1. Lorsque nous listons les résultats au Tableau 5.2, la valeur la plus grande pour chacune des lignes est imprimée en gras afin de pouvoir l'identifier rapidement.

TABLEAU 5.1
PARAMÈTRES DE CONFIGURATION DU PSO, DU GA ET DU PSO-GA HYBRIDE

Métaheuristiques	Paramètres	Valeurs
PSO	Nombre de solutions candidates	20
	Nombre d'itérations	400
	Constante d'inertie ω	0.7298
	Constante d'influence personnelle c_1	1.4960
	Constante d'influence sociale c_2	1.4960
	Vitesse maximale des particules	0.25
GA	Nombre de solutions candidates	20
	Nombre d'itérations	400
	Stratégie de sélection	tournoi
	Stratégie d'enjambement	mélangée
	Stratégie de mutation	aléatoire uniforme
	Stratégie de remplacement	élitisme
	Nombre de candidats pour sélection par tournoi	2
	Probabilité d'enjambement	1.0
	Paramètre α de l'enjambement mélangé	0.5
	Probabilité de sélection d'une solution pour mutation	0.5
	Probabilité de mutation pour chaque élément d'une solution	1/dimension
	Variation maximale Δ pour mutation aléatoire uniforme	0.5
	Nombre de solutions élites pour remplacement	2
PSO-GA Hybride	Nombre de solutions candidates	20
	Nombre d'itérations	400
	Topologie de migration	entièrement connectée
	Fréquence de migration	50 itérations
	Nombres de solutions partagées	1

TABLEAU 5.2
VALEURS MOYENNES DE LA QUALITÉ DES SOLUTIONS FINALES CALCULÉES PAR LE PSO, LE GA ET L'ALGORITHME HYBRIDE PSO-GA POUR SIX FONCTIONS TESTS ET DIFFÉRENTES DIMENSIONS (100 ESSAIS, 20 SOLUTIONS CANDIDATES, 400 ITÉRATIONS)

Fn	Dim.	PSO				GA				PSO-GA Hybride		
		Ref [242]	CPU	GPU	p*	Ref [242]	CPU	GPU	p*	CPU	GPU	p*
Sp	5	**1.000**	**1.000**	**1.000**	1.00	0.999	**1.000**	**1.000**	0.28	**1.000**	**1.000**	1.00
	10	**1.000**	**1.000**	**1.000**	1.00	0.996	**1.000**	**1.000**	0.94	**1.000**	**1.000**	1.00
	20	0.736	**1.000**	**1.000**	0.73	0.921	0.999	0.999	0.94	**1.000**	**1.000**	0.74
Gr	5	0.869	0.986	0.990	0.53	0.945	0.930	0.940	0.34	0.994	**0.995**	0.56
	10	0.981	0.996	0.993	0.36	0.914	0.912	0.891	0.19	**0.998**	0.995	0.20
	20	0.969	0.998	**0.999**	0.40	0.882	0.942	0.926	0.30	**0.999**	**0.999**	0.32
Ra	5	0.183	0.273	0.292	0.44	0.906	**1.000**	**1.000**	0.67	**1.000**	**1.000**	0.56
	10	0.031	0.096	0.092	0.68	0.669	0.898	0.866	0.26	**0.931**	0.921	0.69
	20	0.011	0.034	0.036	0.26	0.084	0.194	0.209	0.71	0.214	**0.227**	0.70
Ro	5	**0.621**	0.280	0.280	0.97	0.390	0.357	0.353	0.90	0.383	0.400	0.58
	10	0.079	0.119	0.115	0.48	0.124	0.071	0.075	0.66	**0.138**	0.130	0.47
	20	0.011	0.050	0.050	0.98	0.011	0.003	0.005	0.26	**0.052**	0.051	0.84
Sw	5	0.995	**1.000**	**1.000**	1.00	0.988	0.997	0.998	0.77	**1.000**	**1.000**	1.00
	10	0.790	**1.000**	**1.000**	0.18	0.703	0.822	0.808	0.45	**1.000**	**1.000**	0.93
	20	0.034	0.926	0.926	0.97	0.092	0.151	0.155	0.66	0.936	**0.941**	0.51
Hy	5	0.135	0.275	0.264	0.80	0.153	0.564	0.515	0.40	**0.866**	0.824	0.35
	10	0.020	0.058	0.050	0.44	0.013	0.076	0.076	0.96	**0.116**	0.114	0.94
	20	0.003	0.016	0.016	0.92	0.003	0.013	0.012	0.90	**0.024**	0.023	0.22

* valeur *p* obtenue par le *test T* pour comparer la qualité des solutions trouvées par la version séquentielle sur CPU et la version parallèle sur GPU

D'après les résultats au Tableau 5.2, nous effectuons trois remarques. Premièrement, nous notons que la qualité des solutions trouvées par les implémentations à la référence [242] et par nos implémentations séquentielles sur CPU du PSO et du GA est comparable. Étant donné que nous n'avons pas accès aux détails des résultats obtenus dans la référence [242], la comparaison faite ici se limite à examiner les valeurs moyennes. Celles-ci ne sont pas exactement identiques, mais elles suivent les mêmes tendances ce qui suggère que nos implémentations sont correctes. En fait, dans la plupart des tests, nos implémentations du PSO et du GA produisent des résultats de meilleure qualité. Cette amélioration est probablement causée par certains paramètres de configuration que nous avons dû choisir puisqu'ils ne sont pas spécifiés dans la référence [242] tel que la limite sur la vitesse des particules du PSO ou la sélection par tournoi du GA.

Toujours d'après les résultats au Tableau 5.2, nous remarquons aussi que nos implémentations séquentielles sur CPU et parallèles sur GPU produisent des résultats équivalents. Pour effectuer cette vérification, nous comparons la valeur des moyennes, mais effectuons aussi un *test T* [243] afin de vérifier si les différences observées sont

significatives ou non. Lorsque la valeur p calculée par le *test T* est plus grande que 0.05, l'hypothèse nulle, qui stipule que les deux moyennes sont équivalentes, ne peut être réfutée à un niveau de signifiance de 5%. En d'autres mots, lorsque la valeur p est plus grande que 0.05, la différence entre les deux moyennes n'est pas significative et celle-ci sont considérées comme étant équivalentes. D'après les valeurs p calculées au Tableau 5.2, il n'y a aucune différences statistiquement significative entres les résultats produits par les implémentations sur CPU et sur GPU. Ceci confirme que la nos parallélisations sur GPU n'a pas affecté le comportement des métaheuristiques.

Finalement, nous remarquons que l'utilisation d'une métaheuristique hybride peut être avantageuse et permet d'améliorer quelque peu la qualité de la solution finale. L'amélioration n'est pas très grande sauf dans le cas de la fonction hybride où elle est significative. En fait, la fonction hybride est composée de quatre fonctions : les fonctions hypersphère, Griewank, Rastrigin et Schwefel. D'après les résultats, le PSO et le GA calculent des solutions semblables pour les fonctions hypersphère et Griewank. Cependant, le GA est supérieur au PSO pour l'optimisation de la fonction Rastrigin. De son côté, le PSO est supérieur au GA pour la fonction Schwefel. Lorsque l'on regroupe ces quatre fonctions, aucun des deux algorithmes ne permet à lui seul d'attaquer la complexité du problème résultant. C'est dans ce genre de situations qu'un algorithme hybride coopératif est avantageux. En échangeant les solutions candidates entre les différentes métaheuristiques, l'algorithme hybride permet leur collaboration et exploite les forces que chacun résultant en un algorithme plus efficace pour une gamme de problèmes plus large. L'observation que nous portons ici au sujet des métaheuristiques hybrides coopératives a aussi été faite par les auteurs de [43] dans le cas d'algorithmes séquentiels sur CPU. Dans cette thèse, en développant le cadriciel *gpuMF*, nous amenons les algorithmes hybrides au GPU.

Après avoir complété le premier test, nous concluons que nos implémentations séquentielles du PSO et du GA inclus dans le cadriciel *gpuMF* sont équivalentes à celles d'autres auteurs dans la littérature. Nous confirmons aussi que notre parallélisation sur GPU n'affecte aucunement le comportement des algorithmes. Finalement, nous concluons que la fonctionnalité hybride de *gpuMF* peut être avantageuse pour l'optimisation de solutions à certains problèmes complexes.

5.4.3 Temps d'exécution et accélération

Dans le second test, nous mesurons le gain de performance apporté par la parallélisation sur GPU. Nous exécutons le PSO séquentiel et le PSO parallèle pour calculer des solutions aux six fonctions tests. Pour démontrer l'avantage du GPU, nous augmentons ici le nombre de dimensions à 60 et faisons varier le nombre de solutions candidates utilisées. Les autres paramètres de configuration sont identiques à ceux listés dans le Tableau 5.1. Le PSO séquentiel est exécuté sur un CPU Intel Xeon

E5-2650 fonctionnant à une fréquence de 2.00 GHz [227]. Le PSO parallèle est exécuté sur un GPU NVIDIA® GTX Titan équipé de 2 688 cœurs fonctionnant à 837 MHz. Les programmes test sont compilés sous Windows 8.1 Pro 64 bits à l'aide de Visual Studio 2013 et CUDA™ SDK 6.5. Les temps d'exécution mesurés et l'accélération calculée sont montrés aux Figures 5.20 et 5.21. Les temps d'exécution varient de 88.65 ms à 132 s pour la version séquentielle et de 36.31 ms à 563.43 ms pour la version parallèle. L'accélération minimum est de 2.3x pour l'optimisation de la fonction Schwefel à l'aide de 64 solutions candidates et augmente jusqu'à 304.4x pour la fonction Griewank à l'aide de 16 384 solutions candidates. D'après ces résultats, l'avantage de la parallélisation sur CPU est évident. Le gain de performance maximal est significatif, mais dépasse les 100x uniquement lorsque le nombre de solutions candidates est plus grand ou égal à 2048. Ceci démontre que le niveau de parallélisme exploité par l'algorithme doit être suffisamment grand pour bénéficier de la puissance de calcul du GPU.

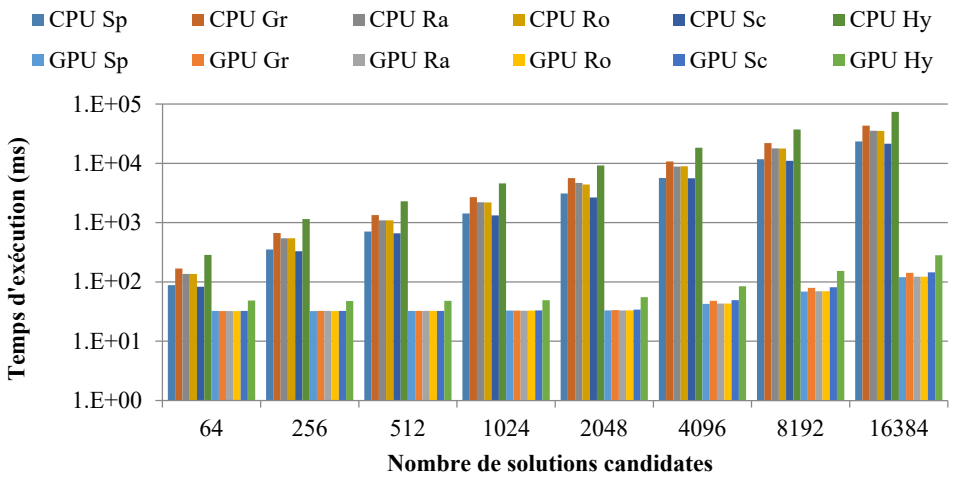


Figure 5.20: Temps d'exécution du PSO séquentiel sur CPU et du PSO parallèle sur GPU pour l'optimisation des six fonctions tests à 60 dimensions, 400 itérations et différents nombres de solutions candidates (moyenne de 10 essais)

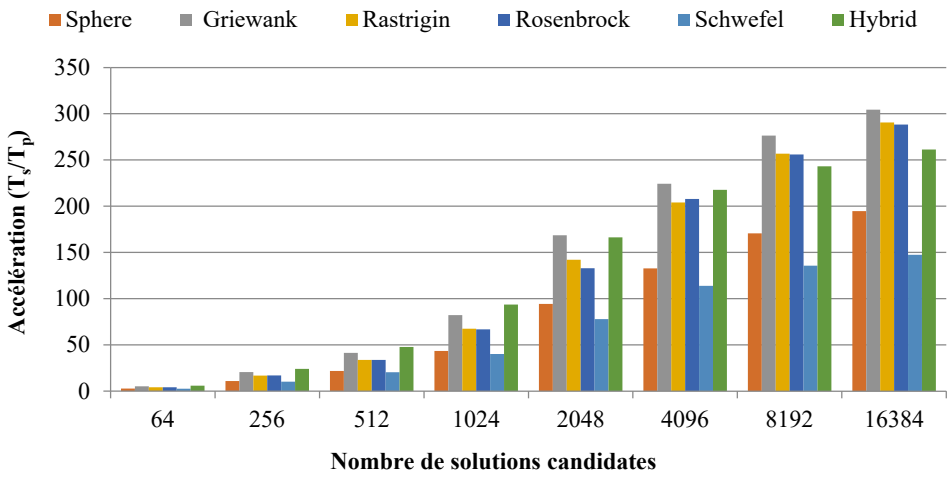


Figure 5.21: Accélération du PSO parallèle sur GPU pour l'optimisation de six fonctions tests à 60 dimensions, 400 itérations et différents nombres de solutions candidates (moyenne de 10 essais)

Utiliser un très grand nombre de solutions candidates tel qu'à l'exemple précédent peut être avantageux pour l'optimisation de problèmes très complexes. Par exemple, nous avons proposé dans [52] un programme CUDA™ qui utilise le PSO sur GPU pour calculer les angles de commutation optimaux d'un onduleur multiniveau. Suivant l'approche suggérée dans [13] et [77], nous exécutons plusieurs instances de l'algorithme d'optimisation afin de garantir la qualité de la solution finale malgré l'aspect non déterministe du PSO. Pour l'implémentation sur GPU, 128 instances du PSO sont exécutées concurremment sur le GPU et chaque instance comprend 512 solutions pour un total de 65 536 solutions candidates. Étant donné le très haut niveau de parallélisme exploité, l'accélération résultante dépasse facilement les 100x. Dans cet exemple, le grand nombre de solutions candidates permet de garantir la qualité de la solution finale tout en maintenant un temps d'exécution acceptable grâce à la parallélisation. Cependant, cette approche ne peut être généralisée à tous les problèmes d'optimisation. Pour plusieurs applications, l'utilisation d'un grand nombre de solutions candidates résulterait en un temps de calcul trop long pour être acceptable. Pour bénéficier de l'architecture du GPU, il faut donc trouver une façon d'exploiter un haut niveau de parallélisme sans augmenter le nombre de solutions candidates. Ceci peut être possible en utilisant un thread CUDA™ pour chaque élément ou dimension de chaque solution candidate. Le niveau de parallélisme exploité devient alors proportionnel à la dimension du problème considéré. Pour démontrer l'avantage d'une telle approche, nous exécutons le PSO pour résoudre la fonction Griewank. Le nombre de solutions candidates est maintenu à 1024 tandis que la dimension du problème varie de 5 à 1000. Pour ce test, deux implémentations différentes sur GPU sont exécutées. La première utilise un thread par solution et chaque thread évalue séquentiellement la fonction Griewank. Un total de

1024 threads est donc créé. La deuxième implémentation exploite un plus haut niveau de parallélisme et utilise un thread par dimension. L'évaluation de la fonction Griewank se fait à l'aide de deux réductions parallèles, une pour la somme et une pour le produit. Le nombre de threads créé varie de 5120 pour la fonction à 5 dimensions jusqu'à 1 024 000 pour la fonction à 1000 dimensions. D'après les temps d'exécution et l'accélération montrés à la Figure 5.22, on remarque que la seconde implémentation sur GPU est bien supérieure à la première et atteint une accélération maximale de 408x contre 152x. Ceci s'explique par le fait que la première implémentation utilise uniquement 1024 threads ce qui ne permet pas d'occuper les 2 688 cœurs du GPU GTX Titan au même point que pour la seconde implémentation. C'est pour cette même raison que tous les programmes CUDA™ développés dans le cadre de cette thèse et présentés au chapitres suivants ont été conçus de façon à utiliser plusieurs threads par solution candidate afin d'exploiter le parallélisme au niveau de données lors de l'évaluation de la fonction objective.

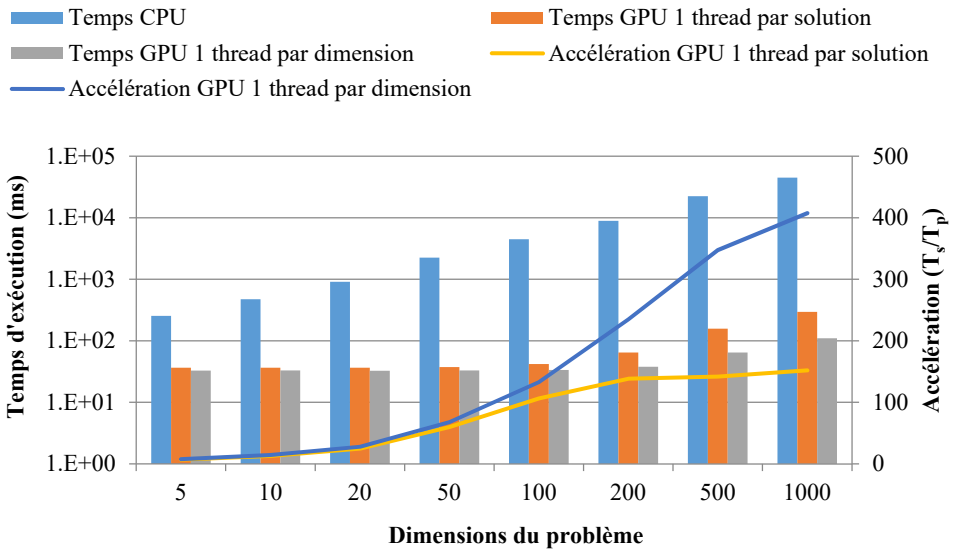


Figure 5.22: Temps d'exécution et accélération du PSO séquentiel et parallèle pour l'optimisation de la fonction Griewank de 5 à 1000 dimensions (moyenne de 10 essais)

Finalement, dans un dernier test, nous exécutons le PSO, le GA et l'algorithme hybride PSO-GA pour les six fonctions tests avec 60 dimensions. L'optimisation utilise 400 itérations et 16 384 solutions candidates par algorithme. L'algorithme hybride est configuré pour exécuter deux populations de solutions candidates, une pour le PSO et l'autre pour le GA. Un processus de migration suivant une topologie entièrement connectée est exécuté tous les 50 itérations afin d'échanger les 16 meilleures solutions entre les deux populations. Nous affichons les temps

d'exécutions et les accélérations mesurés aux Figures 5.23 et 5.24. D'après les résultats obtenus, on remarque que le GA parallèle sur GPU a un temps d'exécution légèrement plus grand que le PSO parallèle. Ceci est normal étant donné que les opérations effectuées par le GA à chaque itération de l'algorithme sont plus compliquées que celles du PSO. De plus, contrairement au PSO, certaines des opérations du GA comme la sélection par tournoi et l'enjambement mélangé nécessitent un accès irrégulier à la mémoire globale du GPU, ce qui entraîne des délais et réduit légèrement la performance finale de l'algorithme. Finalement, on remarque que le temps d'exécution de l'algorithme hybride PSO-GA est un peu plus du double de celui du PSO ou du GA. Ceci est normal puisque l'algorithme hybride est composé de deux populations de 16 384 solutions candidates, mais inclus aussi un processus de migration qui exécute sept fois au cours de l'optimisation. Les accélérations maximales mesurées pour le PSO, le GA et le PSO-GA hybride sont respectivement de 304.4x, 249.3x et 216.7x.

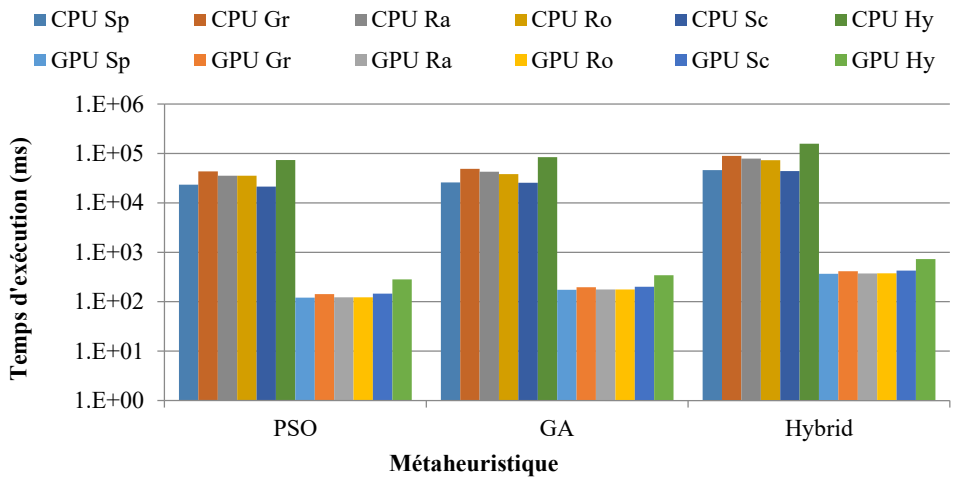


Figure 5.23: Temps d'exécution des algorithmes séquentiels sur CPU et parallèles sur GPU pour l'optimisation des six fonctions tests à 60 dimensions, 400 itérations et 16 384 solutions candidates (moyenne de 10 essais)

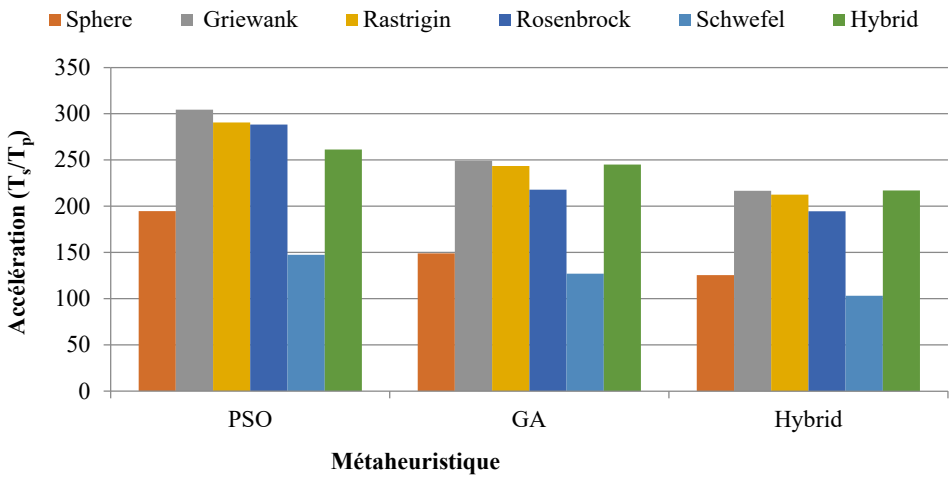


Figure 5.24: Accélération des algorithmes parallèles sur GPU pour l'optimisation des six fonctions tests à 60 dimensions, 400 itérations et 16 384 solutions candidates (moyenne de 10 essais)

5.5 Conclusion

Dans ce chapitre, nous avons présenté *gpuMF*, un cadriciel pour métaheuristiques parallèles sur GPU. Nous avons d'abord décrit sa structure à l'aide de diagrammes UML de classes. Nous avons ensuite expliqué son fonctionnement en discutant trois exemples d'utilisation. Chaque exemple présente le code nécessaire pour lancer l'outil et enchaîne avec les détails de son exécution interne. Le premier exemple a permis de décrire les interactions entre les différents objets logiciels qui composent l'algorithme d'optimisation. Le deuxième exemple met en valeur la dualité de l'architecture de *gpuMF* et démontre comment le cadriciel utilise le paradigme orienté objet au niveau supérieur pour contrôler l'exécution de l'algorithme sur le CPU et des *kernels* CUDA™ au niveau inférieur pour effectuer les calculs en parallèle sur le GPU. Finalement, le troisième exemple démontre le plein potentiel de *gpuMF* en exécutant un algorithme hybride PSO-GA sur le GPU. Pour valider le cadriciel développé, nous avons utilisé six fonctions tests et avons comparé nos solutions à celles obtenues par d'autres implémentations dans la littérature. Ce premier test a permis de conclure que les implémentations séquentielles du PSO et du GA inclus avec *gpuMF* sont correctes. Nous avons ensuite comparé nos implémentations séquentielles sur CPU et parallèles sur GPU à l'aide d'un *test T*. Les résultats obtenus ont permis de confirmer statistiquement que la parallélisation sur GPU n'a aucunement modifié le comportement des métaheuristiques. Dans un dernier test, nous avons mesuré le temps d'exécution de nos implémentations séquentielles sur CPU et parallèles sur GPU afin de calculer l'accélération apportée par *gpuMF*. Cette accélération varie d'après le problème considéré et le niveau de parallélisme exploité.

Une accélération maximale de 408x a été mesurée pour l'optimisation de solutions à la fonction Griewank à 1000 dimensions à l'aide de 1024 solutions candidates.

Le développement du cadriciel *gpuMF* représente une contribution importante. C'est en fait la première fois qu'un cadriciel pour métaheuristiques parallèles sur GPU est proposé. Similairement à une librairie, le cadriciel *gpuMF* inclut des modules logiciels prêts à être utilisé, mais définit aussi le cadre de travail et la structure générique de ces modules de façon à permettre l'extensibilité. Le cadriciel *gpuMF* offre une séparation claire entre l'outil d'optimisation et le problème considéré permettant ainsi sa réutilisation pour une large gamme de problèmes. En proposant une structure logicielle bien définie, le cadriciel facilite l'amélioration et même le développement futur de nouvelles métaheuristiques. Finalement, le cadriciel *gpuMF* synthétise, intègre et perfectionne les derniers avancements dans le domaine des métaheuristiques parallèles sur GPU afin de proposer un outil logiciel qui facilitera les travaux futurs sur de développement de nouveaux algorithmes d'optimisation ou leurs applications à des problèmes d'ingénierie complexes. Comme nous le verrons aux chapitres suivants, en exploitant l'architecture massivement parallèle des GPU, le cadriciel *gpuMF* permet de réduire les temps de calcul, d'attaquer des problèmes plus complexes et de trouver des solutions de meilleure qualité que les méthodes récentes pour plusieurs problèmes d'optimisation d'intérêt dans le domaine des réseaux intelligents.

Chapitre 6
Minimisation des harmoniques d'un onduleur multiniveau

Les travaux de recherche complétés dans le cadre de ce chapitre ont été publiés dans les articles suivants :

V. Roberge, M. Tarbouchi, and G. Labonte, "*Parallel Algorithm on Graphics Processing Unit for Harmonic Minimization in Multilevel Inverters*," IEEE Transactions on Industrial Informatics, vol. 11, no. 3, pp. 700–707, Jun. 2015.

V. Roberge, M. Tarbouchi, and F. Okou, "*Strategies to Accelerate Harmonic Minimization in Multilevel Inverters Using a Parallel Genetic Algorithm on Graphical Processing Unit*," IEEE Transactions on Power Electronics, vol. 29, no. 10, pp. 5087–5090, Oct. 2014.

V. Roberge, M. Tarbouchi, "*Efficient Parallel Particle Swarm Optimizer for Real-Time Harmonic Minimization in Multi-level Inverters*," in proc. of the 38th Annual Conf. of the IEEE Industrial Electronics Society, Montréal, Québec, 25-28 Oct 2012.

6.1 Introduction

La première application considérée dans cette thèse est la minimisation des harmoniques dans un onduleur multiniveau. Les onduleurs multiniveaux forment une classe populaire d'onduleurs de puissance en raison de leur fonctionnement à haute tension, de leur rendement élevé, de leurs pertes de commutation minimes et de leurs faibles perturbations électromagnétiques. Au cours des dernières années, il y a eu un intérêt croissant pour des onduleurs avec plusieurs sources de tension indépendantes de façon à connecter des panneaux solaires, des batteries et des piles à combustible au réseau électrique intelligent afin d'offrir des sources d'énergie alternative [76]. Les onduleurs multiniveaux sont aussi utilisés dans des véhicules hybrides où ils permettent de brancher un grand nombre de batteries ou de charges capacitives au moteur électrique [80], [244], [245].

L'architecture d'un onduleur multiniveau est représentée à la Figure 6.1. L'onduleur est composé d'une série de ponts en H connectés en cascade, chacun alimenté par une source de tension continue. Le contrôle de l'onduleur se fait en ajustant les angles de commutation de façon à produire une sortie alternative en générant une forme d'onde en escalier comme à la Figure 6.2. La difficulté est de

calculer ces angles de façon à produire la tension désirée d'après l'indice de modulation spécifié tout en minimisant les harmoniques qui sont indésirables. Il s'agit en fait d'un problème d'optimisation non linéaire difficilement soluble par les méthodes classiques. Comme nous l'avons expliqué dans la revue de la littérature au Chapitre 2, les solutions actuelles à la minimisation des harmoniques d'un onduleur multiniveau sont limitées à un petit nombre de sources ou nécessitent un temps de calcul trop long pour assurer un contrôle efficace.

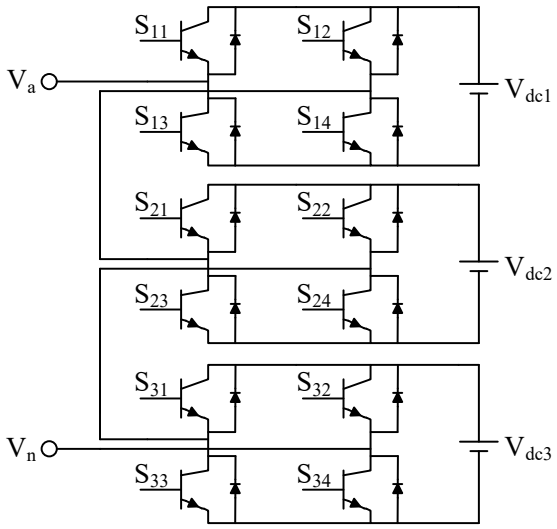


Figure 6.1: Architecture en cascade d'un onduleur multiniveau avec trois modules pont en H

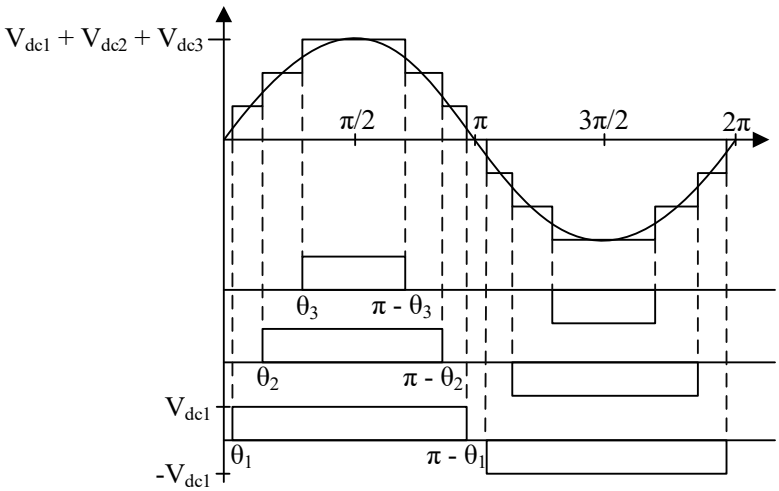


Figure 6.2: Onde de tension de sortie d'un onduleur à 3 niveaux

Dans ce chapitre, nous proposons l'utilisation du cadriciel *gpuMF* pour la minimisation des harmoniques d'un onduleur multiniveau. Contrairement aux autres méthodes, les métaheuristiques ont l'avantage de pouvoir considérer des onduleurs avec un très grand nombre de sources. D'un autre côté, ces algorithmes nécessitent une puissance de calcul considérable et un temps d'exécution souvent trop long pour un contrôle efficace. En utilisant le cadriciel *gpuMF*, nous tirons avantage de l'architecture massivement parallèle des GPU pour accélérer les calculs et garantir un contrôle rapide. L'algorithme proposé utilise l'optimisation par essaim de particules (PSO de l'anglais *particle swarm optimization*) comme moteur d'optimisation et permet de minimiser efficacement les 100 premières harmoniques d'un onduleur à 100 sources de tension indépendantes. Quatre stratégies de parallélisation sont testées afin de déterminer l'approche qui est la mieux adaptée à l'architecture du GPU. Les temps de calcul mesurés varient de 38.7 ms à 117.5 ms dépendamment du nombre de sources de l'onduleur et l'accélération maximale apportée par la parallélisation est de 453.7x comparée à une implémentation séquentielle sur CPU. Nous comparons aussi cette performance à celle d'une implémentation OpenMP® sur processeur à 16 cœurs. Encore ici, le GPU s'avère beaucoup plus rapide. En plus de sa vitesse d'exécution, la méthode proposée supporte un nombre de sources beaucoup plus élevé que les méthodes antérieures et permet un taux de distorsion harmonique plus petit ce qui démontre clairement l'avantage d'une métaheuristique parallèle sur GPU.

En plus du PSO parallèle, nous proposons dans ce chapitre une seconde approche sur GPU pour la minimisation des harmoniques d'un onduleur multiniveau. Cette approche se base sur une formulation mathématique différente du problème et utilise la méthode de Newton ou celle de la bissection pour calculer directement les angles optimaux. Comparée au PSO, la méthode directe nécessite moins de calculs et a l'avantage d'être beaucoup plus rapide. Cependant, elle impose une limite sur l'indice de modulation qui varie d'après le nombre de sources et leur tension. Le taux de distorsion harmonique obtenu est aussi plus élevé que pour le PSO. L'approche directe sur GPU représente toutefois une alternative très intéressante aux métaheuristiques étant donné son temps d'exécution extrêmement court. Elle permet de calculer les angles d'un onduleur à 1000 sources en seulement 16.41 µs.

Le reste de ce chapitre est organisé comme suit. La section 6.2 introduit la formulation mathématique du problème. La méthode d'optimisation qui utilise le cadriciel *gpuMF* est ensuite présentée à la section 6.3. La parallélisation de cette méthode sur GPU incluant quatre stratégies pour l'implémentation de la fonction de coût est discutée à la section 6.4. Les résultats expérimentaux sont ensuite présentés à la section 6.5. Finalement, la section 6.6 traite de la méthode directe sur GPU et inclut la formulation mathématique, l'implémentation basée sur la méthode de Newton et celle de la bissection ainsi que les tests de validation.

6.2 Définition du problème

Étant donnée l'architecture illustrée à la Figure 6.1, le problème de la minimisation des harmoniques d'un onduleur multiniveau consiste à calculer les angles de commutation optimaux de façon à générer une tension alternative dont l'amplitude de la composante fondamentale est le plus près possible de celle désirée pour un indice de modulation donné tout en minimisant le taux de distorsion harmonique (THD de l'anglais *total harmonic distortion*). Le modèle mathématique présenté ici provient de [77], mais a été généralisé afin de considérer un onduleur multiniveau avec des sources de tensions inégales. Pour un indice de modulation m et une série de sources CC inégales, l'amplitude désirée V_1^* de la composante fondamentale peut être calculée ainsi :

$$V_1^* = \frac{4 * m * \sum_{k=1}^{s} V_{dc_k}}{\pi} \tag{6.1}$$

où s est le nombre de sources CC et V_{dc_k} est la tension de la k^e source. Par la suite, une analyse par séries de Fourier de l'onde de tension de sortie en escalier à la Figure 6.2 permet de déterminer l'amplitude de la composante fondamentale $V_{n=1}$ produite par l'onduleur ainsi que les harmoniques associées $V_{n=3,5,7,\dots}$ comme suit :

$$V_n = \frac{4}{n * \pi} \sum_{k=1}^{s} V_{dc_k} cos(n * \theta_k) \tag{6.2}$$

où n est l'ordre de l'harmonique et θ_k est le k^e angle de commutation. Ces angles doivent satisfaire la contrainte suivante afin d'assurer une sortie en escalier :

$$0 \leq \theta_1 \leq \cdots \leq \theta_k \leq \cdots \leq \theta_s \leq \frac{\pi}{2} \tag{6.3}$$

Finalement, le taux de distorsion harmonique THD est calculé en divisant la racine carrée de la somme des harmoniques par la composante fondamentale :

$$THD = \frac{\sqrt{\sum_{h=3,5,7,9,\dots} V_n^2}}{V_1} \tag{6.4}$$

Comme il est fait dans la référence [77], seules les harmoniques d'ordre impair sont considérées dans le calcul du THD puisque celles d'ordre pair sont supprimées par la symétrie de l'onde de tension de sortie. D'après ces équations, le problème de la minimisation des harmoniques d'un onduleur multiniveau consiste à calculer les angles de commutation $\vec{\theta}$ qui produisent une tension de sortie dont l'amplitude V_1 de la composante fondamentale est le plus près possible de l'amplitude désirée V_1^* tout en minimisant le THD. Il s'agit d'un problème d'optimisation non linéaire et

multimodale difficilement soluble par les méthodes classiques, spécialement lorsque le nombre de sources ou d'harmoniques considérés est grand.

6.3 Méthode d'optimisation proposée

Comme nous l'avons identifié dans la revue de la littérature, les métaheuristiques ont été utilisées avec succès pour la minimisation des harmoniques d'un onduleur multiniveau. Contrairement aux méthodes déterministes, les métaheuristiques maintiennent une bonne efficacité même lorsque le nombre de sources CC est augmenté. Au lieu de dériver analytiquement une solution au problème, ces algorithmes débutent le processus d'optimisation avec une ou plusieurs solutions candidates et offrent une stratégie de recherche itérative basée sur l'évaluation de la qualité des solutions candidates pour finalement obtenir une solution quasi optimale. La qualité des solutions se calcule à l'aide d'une fonction de coût aussi appelée fonction objective. La fonction de coût doit inclure le ou les objectifs d'optimisation ainsi que les contraintes. Dans notre cas, nous définissons la fonction de coût suivante en nous basant sur celle suggérée par les auteurs de [13], mais modifiée de façon à restreindre l'erreur sur V_1 et à inclure le THD directement dans l'équation.

$$f_{Cost}(\vec{\theta}) = \left(1000 * \frac{\mathrm{V}_1^* - \mathrm{V}_1}{\mathrm{V}_1^*}\right)^4 + \mathrm{THD} \tag{6.5}$$

Dans cette équation, le premier terme pénalise sévèrement toute solution $\vec{\theta}$ dont l'erreur entre V_1 et V_1^* est plus grande que 0.1%. Le second terme pénalise les solutions qui génèrent un THD élevé. Cette fonction de coût permet à la métaheuristique de comparer la qualité des solutions candidates et de trouver des angles de commutation pour produire la bonne amplitude tout en minimisant les harmoniques.

Quant au choix de la métaheuristique, étant donné que le cadriciel *gpuMF* présenté au chapitre 5 implémente le PSO et l'algorithme génétique (GA de l'anglais *genetic algorithm*), il est possible d'utiliser l'un ou l'autre. En fait, certains auteurs favorisent le premier [77] tandis que d'autres préfèrent le second [106]. Pour cette raison nous avons testé les deux algorithmes et publié les résultats dans [52] et [104]. La différence entre les deux méthodes n'est pas très grande, mais les résultats semblent démontrer que le PSO soit mieux adapté au problème en question. Une caractéristique plus intéressante est le fait que le PSO soit plus simple à implémenter que le GA. De plus, contrairement au GA, le PSO requiert uniquement des accès à la mémoire qui sont coalescents permettant ainsi une meilleure performance lorsqu'implémenté sur GPU. Étant donné que le temps de calcul est critique dans l'application considérée, le PSO est donc préférable. Pour ces raisons, notre implémentation utilise le PSO comme moteur d'optimisation. Les solutions

candidates sont initialisées aléatoirement suivant une distribution uniforme de 0 à $\pi/2$ et ces limites sont respectées lors du calcul des nouvelles positions des particules. De plus, les angles de commutations formant chacune des solutions candidates sont triés avant l'évaluation de la fonction de coût afin d'assurer que la contrainte à l'équation (6.3) soit toujours respectée. Aussi, suivant la recommandation faite par les auteurs de [77], nous organisons les solutions candidates en sous-ensembles, ou îlot, et implémentons un processus de migration permettant de partager les meilleures solutions à un intervalle fixe suivant une topologie circulaire bidirectionnelle. En fait, les auteurs de [77] avaient remarqué que, dû à son aspect non déterministe, la métaheuristique peut occasionnellement converger prématurément vers un optimum local et retourner une solution de qualité médiocre. Afin d'éviter ce problème, ils suggèrent d'exécuter le PSO à dix reprises avant de retourner la meilleure solution finale. Nous pourrions nous aussi implémenter cette même stratégie, mais il a été démontré dans [42] que l'ajout d'un processus de migration comme nous le faisons permet d'améliorer encore plus l'efficacité de l'algorithme. Finalement, les autres détails du PSO sont tels que décrits au chapitre 3 et n'ont pas besoin d'être répétés ici.

6.4 Parallélisation

Les métaheuristiques sont des méthodes efficaces pour la minimisation des harmoniques d'un onduleur multiniveau. Cependant, elles nécessitent une puissance de calcul considérable ce qui entraîne souvent un temps d'exécution trop long pour une application en temps réel. Dans cette section, nous mitigeons cette lacune et expliquons comment il est possible de paralléliser l'algorithme pour une implémentation sur GPU afin de réduire le temps de calcul. Étant donné que le PSO parallèle est déjà implémenté par le cadriciel *gpuMF*, nous concentrons nos explications sur la parallélisation de la fonction de coût. Cette fonction nécessite des calculs complexes et est exécutée pour chaque solution candidate à chaque itération du PSO. C'est principalement la fonction de coût qui consomme la majorité des cycles d'horloge du CPU lors de l'exécution de l'algorithme d'optimisation. Il est donc important de paralléliser efficacement cette fonction pour garantir la performance du programme final. Aux paragraphes suivants, nous analysons le nombre d'étapes nécessaire pour calculer séquentiellement la fonction de coût. Nous présentons ensuite quatre stratégies de parallélisation et expliquons comment chacune permet de réduire le nombre d'étapes nécessaires au calcul.

Considérons l'évaluation de M solutions candidates pour un onduleur multiniveau à s sources de tension en minimisant les N premières harmoniques. Pour une implémentation séquentielle, le nombre d'étapes nécessaire au calcul de la fonction de coût peut être évalué comme suit :

1) En moyenne, $s * log(s)$ étapes sont nécessaires pour trier les angles d'une seule solution à l'aide d'un algorithme de tri rapide;
2) s étapes pour calculer les s termes d'une harmonique V_n;
3) s étapes pour additionner ces s termes et calculer l'harmonique V_n; et
4) N étapes pour additionner les harmoniques et calculer le THD.

Ainsi, si l'on omet les opérations dont la complexité ne dépend pas de la dimension du problème, le nombre d'étapes nécessaires pour calculer séquentiellement le coût de M solutions candidates est :

$$T_{seq} = M * [s\, log(s) + N\ (s + s) + N] \tag{6.6}$$

Regardons maintenant comment une implémentation parallèle permet de diminuer ce nombre d'étapes.

6.4.1 Implémentation 1 : un thread par solution

L'approche parallèle la plus simple consiste à évaluer le coût des solutions candidates en utilisant un thread par solution. Le nombre de threads créés est donc égal à M pour l'évaluation de M solutions. Nous avons proposé cette méthode dans [52]. Elle est facile à implémenter et elle offre une accélération significative lorsque le nombre de solutions candidates est large. Dans le cas contraire, le niveau de parallélisme exploité n'est pas suffisant pour utiliser pleinement l'architecture massivement parallèle du GPU. Dans cette première implémentation, le nombre d'étapes est réduit comme suit étant donné que les M solutions candidates sont évaluées concurremment :

$$T_{par\ 1} = s \log(s) + N\ (s + s) + N \tag{6.7}$$

6.4.2 Implémentation 2 : un thread par harmonique

Pour exploiter un niveau de parallélisme supérieur, il est possible d'utiliser un bloc de threads par solution et un thread par harmonique. Le nombre de threads créés est donc égal à $M * N$. Les harmoniques sont alors calculées concurremment en $(s + s)$ étapes au lieu de $N\ (s + s)$ comme pour l'implémentation précédente. Cependant, étant donné que les valeurs des harmoniques se retrouvent sur différents threads, il est nécessaire d'effectuer une réduction parallèle [234] pour les additionner et calculer le THD. L'opération de réduction est expliquée en détail au chapitre 4 et requiert $log(N)$ étapes au lieu de N étapes pour une implémentation séquentielle. Le nombre total d'étapes pour l'évaluation de M solutions candidate est alors réduit à :

$$T_{par\ 2} = s \log(s) + (s + s) + \log(N) \tag{6.8}$$

6.4.3 Implémentation 3 : avec tri bitonique

Dans l'implémentation précédente, à cause de la contrainte à l'équation (6.3), le premier thread de chaque bloc doit trier séquentiellement, à l'aide de l'algorithme de tri rapide, les angles de commutation afin d'assurer qu'ils soient toujours en ordre croissant. Cette opération séquentielle prend en moyenne $s * log(s)$ étapes à exécuter, mais peut être remplacée par un tri parallèle bitonique [55] qui lui prend $\log^2(s)$ étapes. Le nombre total d'étapes est alors :

$$T_{par\,3} = \log^2(s) + (s + s) + \log(N) \tag{6.9}$$

6.4.4 Implémentation 4 : un thread par terme pour chaque harmonique

Finalement, le niveau de parallélisme exploité peut encore une fois être augmenté en utilisant un thread par terme de l'équation (2.4). De cette façon, chaque bloc de threads calcule une seule harmonique. Le nombre total de threads créés est alors $M * N * s$ pour l'évaluation de M solutions à s sources considérant les N premières harmoniques. Contrairement aux trois implémentations précédentes, cette quatrième approche nécessite la programmation de deux *kernels* CUDA™. Dans le premier, les termes de l'équation (2.4) sont calculés en une seule étape et additionnés en $log(s)$ étapes à l'aide d'une réduction parallèle pour calculer une harmonique V_h par bloc de threads. Dans le second *kernel*, une autre réduction parallèle est utilisée pour additionner les harmoniques et calculer le THD des solutions. De cette façon, le nombre d'étapes pour le calcul des harmoniques est réduit de $(s + s)$ à $(1 + \log(s))$. Le nombre total d'étapes pour l'évaluation des M solutions candidates devient alors à :

$$T_{par\,4} = \log^2(s) + (1 + \log(s)) + \log(N) \tag{6.10}$$

Tout comme pour l'analyse de l'implémentation séquentielle, notre analyse des implémentations parallèles n'inclut pas les opérations dont la complexité ne dépend pas de la dimension du problème tel que la racine carrée nécessaire au calcul du THD. On remarque qu'augmenter le parallélisme exploité diminue le nombre d'étapes nécessaire à l'évaluation de la fonction de coût puisque les opérations sont effectuées concurremment sur les multiples cœurs du GPU. Cette diminution est cependant possible uniquement lorsque le nombre de cœurs disponibles est suffisant pour exécuter tous les threads simultanément, ce qui n'est pas toujours vrai. De plus, un haut niveau de parallélisme introduit nécessairement un délai supplémentaire pour la synchronisation des threads. Ceci est évident pour la quatrième implémentation qui nécessite l'utilisation de deux *kernels* CUDA™ au lieu d'un seul. Pour identifier la meilleure stratégie de parallélisation, il est donc nécessaire de tester chacune des implémentations parallèles pour différentes dimensions de problème dans le but

d'identifier celle qui offre le meilleur compromis entre le nombre d'étapes nécessaire et les délais de synchronisation afin d'obtenir la meilleure accélération possible. Ces tests sont discutés à la section suivante.

6.4.5 Comparaison expérimentale des quatre implémentations parallèles

Afin d'illustrer l'avantage d'une implémentation sur GPU et d'identifier la stratégie de parallélisation qui offre la meilleure accélération, nous comparons dans cette section les quatre implémentations parallèles discutées précédemment. Dans ces tests, la version séquentielle est exécutée sur un CPU Intel Xeon E5-2650 fonctionnant à 2 GHz tandis que les versions parallèles sont exécutées sur un processeur graphique NVIDIA® Tesla K20C équipé de 2496 cœurs fonctionnant à 706 MHz. Les programmes test sont compilés sous Windows 8.1 Pro 64 bits à l'aide de Visual Studio 2013 et de CUDA™ SDK 6.5. Nous listons au Tableau 6.1 les temps d'exécution pour différentes dimensions du problème et identifions en gras le temps le plus petit. Tous les temps sont exprimés en millisecondes et mesurent uniquement l'évaluation de la fonction de coût. Dans le premier test, le nombre de sources et le nombre d'harmoniques considérés dans le calcul du THD sont gardés constants tandis que l'on varie le nombre de solutions. On remarque que pour des petits nombres de solutions, l'implémentation 3 offre la meilleure performance et donc le meilleur compromis entre le niveau de parallélisme exploité et la surcharge de travail due à la parallélisation. Dans le cas d'un grand nombre de solutions candidates, c'est l'implémentation 1 qui offre la meilleure performance. Le nombre de thread créés est suffisant pour occuper pleinement les 2496 cœurs du GPU tandis que la surcharge de travail dû à la parallélisation est presque nulle. Dans le test 2, nous varions le nombre de sources. On remarque que l'implémentation 3 est supérieure aux autres puisqu'elle utilise un algorithme de tri parallèle pour ordonnancer les angles de commutation avant de calculer la fonction de coût. Or, d'après les explications données à la section 4.7.7, lorsque le nombre de sources est égal à 100, le tri bitonique prend uniquement 49 étapes à compléter tandis que le tri rapide prend en moyenne 700 étapes, ce qui explique pourquoi la troisième implémentation est si performante. Finalement, dans le test 3, nous varions le nombre d'harmoniques considérées et remarquons une fois de plus que la troisième implémentation est la plus rapide. Nous concluons donc, que dans la majorité des cas, la stratégie de parallélisation qui utilise un tri bitonique pour ordonnancer les angles ainsi qu'un thread par harmonique permet d'obtenir la meilleure performance. Pour cette raison, nous utilisons cette troisième implémentation dans le programme d'optimisation présenté dans ce chapitre pour la minimisation des harmoniques d'un onduleur multiniveau.

TABLEAU 6.1
TEMPS D'EXÉCUTION POUR L'ÉVALUATION DE LA FONCTION DE COÛT POUR DIFFÉRENTES DIMENSIONS DU PROBLÈME ET DIFFÉRENTES IMPLÉMENTATIONS (MOYENNE DE 100 ESSAIS, E5-2650, K20C)

Test	Paramètres			Temps d'exécution (ms)				
	Sources	Harmoniques	Solutions	CPU	GPU1	GPU2	GPU3	GPU4
1	20	20	128	1.439	0.221	0.065	**0.040**	0.087
	20	20	512	5.754	0.223	0.143	**0.070**	0.231
	20	20	4092	46.117	**0.232**	0.819	0.305	1.581
2	20	20	512	5.758	0.207	0.142	**0.068**	0.228
	50	20	512	13.277	0.463	0.348	**0.095**	0.248
	100	20	512	25.354	0.989	0.730	**0.135**	0.328
3	20	20	512	5.761	0.207	0.142	**0.069**	0.229
	20	50	512	14.844	0.384	0.147	**0.080**	0.465
	20	100	512	30.118	0.681	0.194	**0.105**	0.853

6.4.6 Implémentation sur CPU multicœur

Le gain de performance apporté par une implémentation parallèle est habituellement mesuré en termes d'accélération par rapport au temps d'exécution d'une implémentation séquentielle sur CPU. Ceci offre un point de référence clair, peu importe le type de processeur utilisé pour l'exécution parallèle [233]. De façon à être consistant avec la littérature publiée dans le domaine, nous mesurons nous aussi les gains de performance des programmes parallèles développés dans le cadre de cette thèse en termes d'accélération par rapport à un programme séquentiel sur CPU. Cependant, étant donné que les CPU multicœurs sont maintenant une technologie bien établie et qu'ils peuvent offrir un gain de performance significatif, nous les incluons dans nos comparaisons. Pour chacune des trois applications étudiées, nous développons une version séquentielle sur CPU, mais aussi une version parallèle pour CPU multicœurs à l'aide d'OpenMP®. OpenMP® est une interface de programmation d'applications (API de l'anglais *Application Program Interface*) simple et flexible pour le développement de programme parallèle sur système à mémoire partagée.

Pour développer une version parallèle sur CPU de l'algorithme de minimisation des harmoniques d'un onduleur multiniveau, nous exécutons d'abord le programme séquentiel et mesurons le temps d'exécution des différentes fonctions logicielles qui composent le programme. Tout comme précédemment, le test est exécuté sur un ordinateur Dell T7600 équipé de deux CPU Intel Xeon E5-2650. Chaque CPU contient huit cœurs qui implémentent la technologie *hyperthreading*. L'ordinateur utilisé contient donc 16 cœurs physiques et 32 cœurs virtuels. Dans le cas où le PSO utilise 512 solutions candidates sur 400 itérations pour l'optimisation d'un onduleur à

50 sources tout en minimisant les 100 premières harmoniques, nous mesurons un temps d'exécution total de 27.874 secondes. Nous remarquons aussi que 27.385 s, soit 98.2% du temps d'exécution total, sont utilisées par l'évaluation de la fonction de coût. D'après ces mesures, une stratégie de parallélisation simple et efficace serait de partager les calculs nécessaires à l'évaluation de la fonction de coût des solutions candidates sur les multiples cœurs du CPU suivant l'approche maître-esclave que nous avons présenté au chapitre 2. D'après cette approche, le thread maître exécute le PSO séquentiellement sur un seul cœur et délègue le calcul de la fonction de coût aux threads esclaves. Cette stratégie est simple d'implémentation, ne nécessite aucune modification au cadriciel *gpuMF*, exploite un niveau de parallélisme suffisant pour occuper les multiples cœurs du CPU et minimise la communication inter-threads à une seule synchronisation à chaque itération du PSO. Pour déterminer le nombre optimal de threads à utiliser, nous calculons la fonction de coût pour les 512 solutions candidates et différents nombres de sources et d'harmoniques. Nous varions le nombre de threads de 1 à 36 et affichons à la Figure 6.3 l'accélération mesurée.

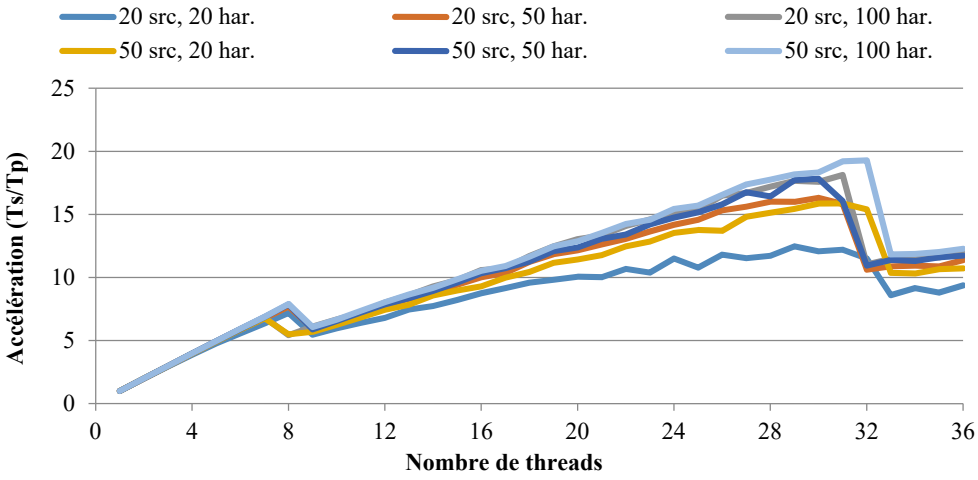


Figure 6.3: Accélération pour l'évaluation de la fonction de coût sur CPU multicœur par rapport au nombre de threads OpenMP® (moyenne de 10 essais, E5-2650, K20c)

Pour la plupart des cas, nous remarquons que l'accélération obtenue atteint son maximum à 31 threads. Nous utilisons donc cette configuration lors de l'exécution de notre algorithme de minimisation des harmoniques sur processeur multicœur. Dans le cas d'un onduleur à 50 sources considérant les 100 premières harmoniques, l'accélération maximale est de 19.20x. D'après cette valeur, nous utilisons la loi d'Amdahl [33] pour calculer à l'équation (6.11) l'accélération attendue pour le programme en entier. Cette accélération est définie comme le temps d'exécution du

programme séquentiel divisé par celui du programme parallèle. Dans cette équation, la valeur 27.874 s représente le temps d'exécution du programme séquentiel en entier. La valeur 0.462 s représente le temps nécessaire à la partie du programme parallèle qui n'a pas été parallélisé, soit l'exécution du PSO par le thread maître. Finalement, le terme 27.385/19.20 s représente une estimation du temps nécessaire pour calculer en parallèle le coût des solutions candidates par les threads esclaves. Ce calcul nous donne une accélération attendue de 14.74x, ce qui est très bon étant donné que le processeur utilisé contient 16 cœurs.

$$S = \frac{T_{séq}}{T_{par}} = \frac{27.874\ s}{0.462 + (27.385/19.20)\ s} = \frac{27.874\ s}{1.888\ s} = 14.74x \tag{6.11}$$

6.5 Résultats expérimentaux

Dans cette section, nous présentons les résultats expérimentaux obtenus par l'algorithme d'optimisation proposé pour la minimisation des harmoniques d'un onduleur multiniveau. Dans notre implémentation, le PSO utilise 512 solutions candidates divisées en 16 sous-ensembles ou îlots. Cette structure en îlots a été discutée à la section 6.3 et permet d'augmenter la fiabilité de la métaheuristique [13] et [77]. Chaque îlot maintient sa propre meilleure solution et un processus de migration entre les îlots, suivant une topologie circulaire bidirectionnelle, est effectué toutes les 50 itérations et permet de partager les meilleures solutions. Pour favoriser la convergence [209], le PSO utilise un paramètre d'inertie ω égale à 0.729 et des paramètres d'influence personnelle c_1 et d'influence sociale c_2 tous les deux égaux à 1.493. L'exécution de l'algorithme se termine après 400 itérations. La meilleure solution parmi tous les îlots est alors retournée par le programme.

Dans un premier test, nous utilisons le PSO parallèle sur GPU pour calculer les angles optimaux d'un onduleur multiniveau à 10 sources dont les tensions sont : V_{DC1} = 1.10, V_{DC2} = 1.04, V_{DC3} = 0.98, V_{DC4} = 1.02, V_{DC5} = 1.08, V_{DC6} = 0.92, V_{DC7} = 0.96, V_{DC8} = 1.06, V_{DC9} = 0.94 et V_{DC10} = 0.90 p.u.. L'optimisation minimise le THD de la tension alternative produite en considérant les 100 premières harmoniques. Les angles sont calculés pour des indices de modulation de 0.5 à 1.0 et affichés à la Figure 6.4. Le THD ainsi que l'erreur entre le V_1^* désiré et le V_1 obtenu sont affichés aux Figures 6.5 et 6.6. Les résultats confirment que la fonction de coût que nous avons définie précédemment est appropriée et garantit une erreur extrêmement petite pour V_1 tout en minimisant avec succès le THD jusqu'à la 100^e harmonique. Comparée aux résultats publiés dans [13], [77] et [106], la méthode parallèle proposée dans cette thèse permet un THD beaucoup plus faible puisqu'elle considère un nombre plus élevé de sources de tension, ce qui n'est pas possible avec une implémentation séquentielle à cause d'un trop long temps d'exécution. Pour

illustrer ce fait, nous avons aussi inclus aux Figures 6.5 et 6.6 le THD et l'écart entre le V_1^* désiré et le V_1 obtenu pour des onduleurs avec 5, 10, 15 et 20 sources. D'après ces figures, il est évident qu'augmenter le nombre de sources d'un onduleur multiniveau est avantageux et permet de réduire le THD de la tension alternative produire. Cette augmentation est rendue possible grâce à l'algorithme parallèle sur GPU que nous proposons pour calculer les angles optimaux de commutation tout en minimisant le temps de calcul.

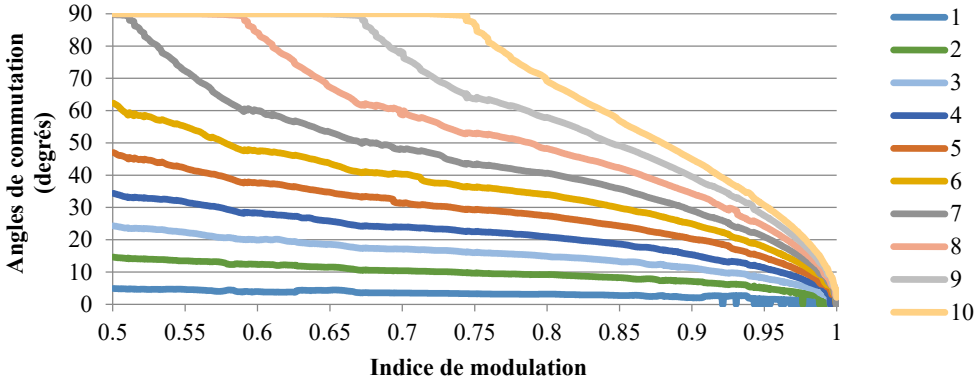


Figure 6.4: Angles de commutation optimaux calculés par le PSO parallèle sur GPU pour un onduleur à 10 sources et un indice de modulation variant de 0.5 à 1.0

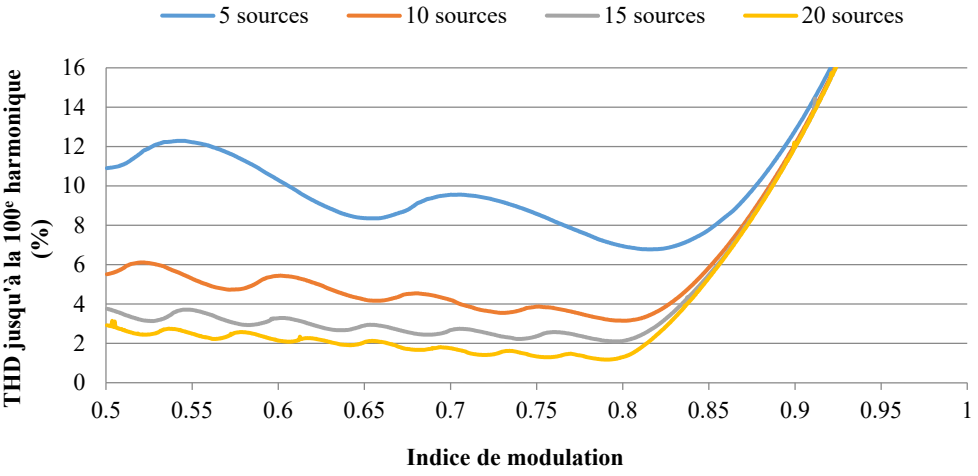


Figure 6.5: Taux de distorsion harmonique de la tension de sortie considérant les 100 premières harmoniques pour des onduleurs avec différents nombres de sources

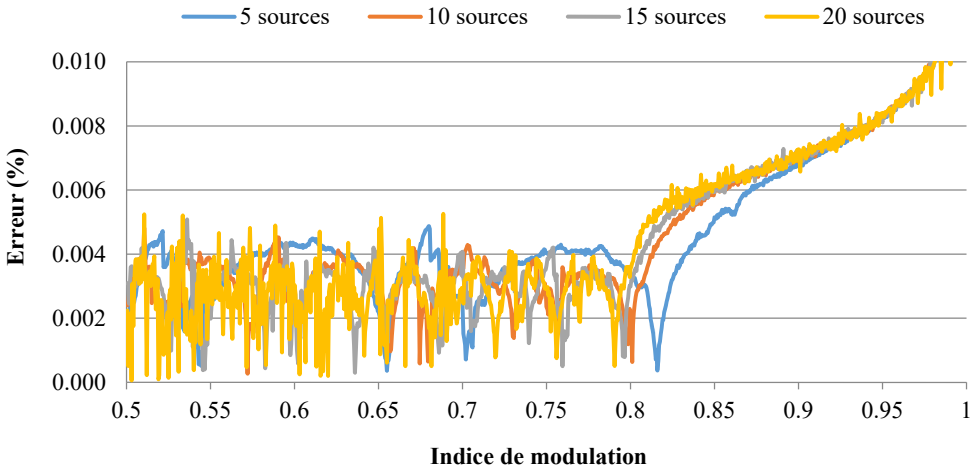


Figure 6.6: Erreur entre l'amplitude de la composante fondamentale de la tension de sortie et celle désirée pour des onduleurs avec différents nombres de sources

Dans un second test, afin de vérifier la justesse de la formulation mathématique utilisée pour modéliser l'onduleur multiniveau, nous fixons l'indice de modulation à 0.8 et calculons les angles optimaux pour l'onduleur à 10 sources décrit au paragraphe précédent. La tension des sources ainsi que les angles calculés sont listés au Tableau 6.2. La sortie en escalier produite par l'onduleur est représentée sous forme d'un diagramme de front d'onde à la Figure 6.7. Cette onde de sortie est ensuite analysée à l'aide d'une transformation de Fourier rapide (FFT de l'anglais *Fast Fourier Transform*) sous MATLAB® afin de mesurer l'amplitude des harmoniques que nous affichons à la Figure 6.8. Les valeurs obtenues par la transformation FFT sont identiques à celles que nous calculons à l'aide de l'équation (2.4) ce qui confirme la validité du modèle mathématique utilisé. Le THD mesuré est de 3.15%, ce qui est beaucoup plus petit que les valeurs de 12.89% et 12.25% obtenues respectivement par les auteurs de [13] et [100], aussi pour un indice de modulation de 0.8. Tel qu'expliqué au paragraphe précédent, cette différence est due à l'utilisation d'un nombre de sources plus élevé rendu possible grâce à l'algorithme parallèle proposé.

TABLEAU 6.2
ANGLES DE COMMUTATION OPTIMAUX CALCULÉS PAR LE PSO PARALLÈLE SUR GPU POUR UN ONDULEUR À 10 SOURCES ET UN INDICE DE MODULATION DE 0.8

Source	Tension (p.u.)	Angle de commutation (rad)	Source	Tension (p.u.)	Angle de commutation (rad)
V_{dc1}	1.10	0.0549	V_{dc6}	0.92	0.5939
V_{dc2}	1.04	0.1608	V_{dc7}	0.96	0.7091
V_{dc3}	0.98	0.2597	V_{dc8}	1.06	0.8425
V_{dc4}	1.02	0.3654	V_{dc9}	0.94	1.0106
V_{dc5}	1.08	0.4790	V_{dc10}	0.90	1.2062

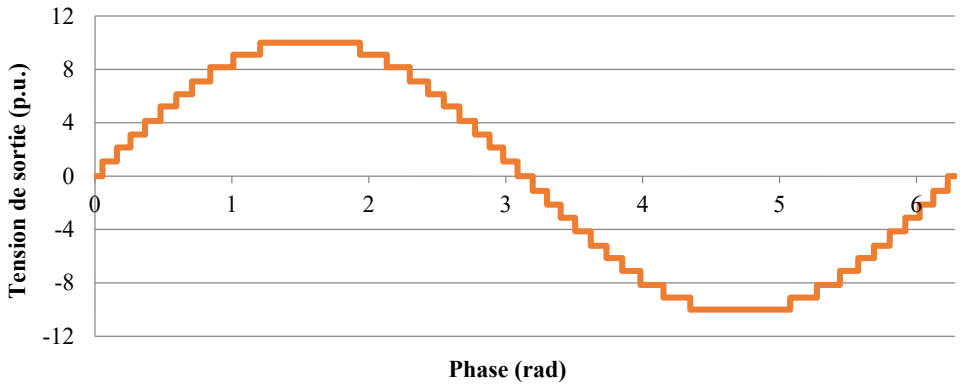


Figure 6.7: Tension de sortie générée par les angles de commutation optimaux calculés par le PSO parallèle sur GPU pour l'onduleur à 10 sources et un indice de modulation de 0.8

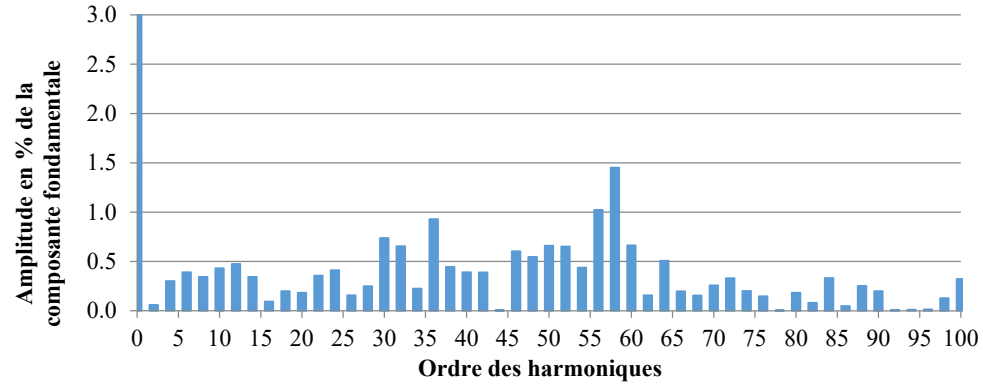


Figure 6.8: Amplitude (en % de la composante fondamentale) des harmoniques de la tension de sortie illustrée à la Figure 6.7. Malgré que l'axe vertical s'arrête à 3.0, l'amplitude de la composante fondamentale se rend à exactement 100%. (THD = 3.153%)

Dans un dernier test, afin d'évaluer la performance de l'algorithme parallèle proposé, nous exécutons trois implémentations du programme pour différents nombres de sources et d'harmoniques. Les temps d'exécutions mesurés ainsi que les accélérations calculées sont respectivement affichés aux figures 6.9, 6.10 et 6.11. Le programme identifié par l'acronyme "CPU" à la figure 6.9 représente l'implémentation séquentielle du PSO exécutée sur le CPU. Le programme identifié par "OMP" dénote l'implémentation parallèle maître-esclave codée à l'aide de OpenMP®. Ce programme est lui aussi exécuté sur le CPU, mais utilise tous les cœurs disponibles pour une meilleure performance. Finalement, le programme identifié par "GPU" représente l'implémentation parallèle sur GPU. Les tests sont exécutés sur un ordinateur Dell T7600 équipé de deux CPU Intel Xeon E5-2650 et d'un processeur graphique NVIDIA® Tesla K20C. Chaque CPU contient huit cœurs fonctionnant à 2 GHz tandis que le processeur graphique contient 2496 cœurs fonctionnant à 706 MHz. Les programmes test sont compilés sous Windows 8.1 Pro 64 bits à l'aide de Visual Studio 2013 et de CUDA™ SDK 6.5. Les temps d'exécutions pour la version séquentielle sur CPU varie de 313.4 ms pour un onduleur à cinq sources considérant les dix premières harmoniques jusqu'à 53.3 s pour un onduleur à 100 sources considérant les 100 premières harmoniques. Dans le cas de l'implémentation OMP, ces temps sont réduits à 85.1 ms et 3.98 s. Finalement, l'implémentation sur GPU est celle qui offre les plus petits temps d'exécutions variant seulement de 38.7 ms à 117.5 ms. Ces temps d'exécution se traduisent en une accélération maximale de 13.44x pour le cas de l'implémentation parallèle sur CPU multicœur et de 453.7x pour celle sur GPU. Ces résultats démontrent sans aucun doute l'avantage du cadriciel *gpuMF* et de la parallélisation sur GPU pour le calcul des angles de commutation optimaux d'un onduleur multiniveau. En réduisant le temps de calcul, l'approche parallèle sur GPU permet d'optimiser des onduleurs avec un plus grand nombre de sources et de générer une tension de sortie avec un plus petit THD.

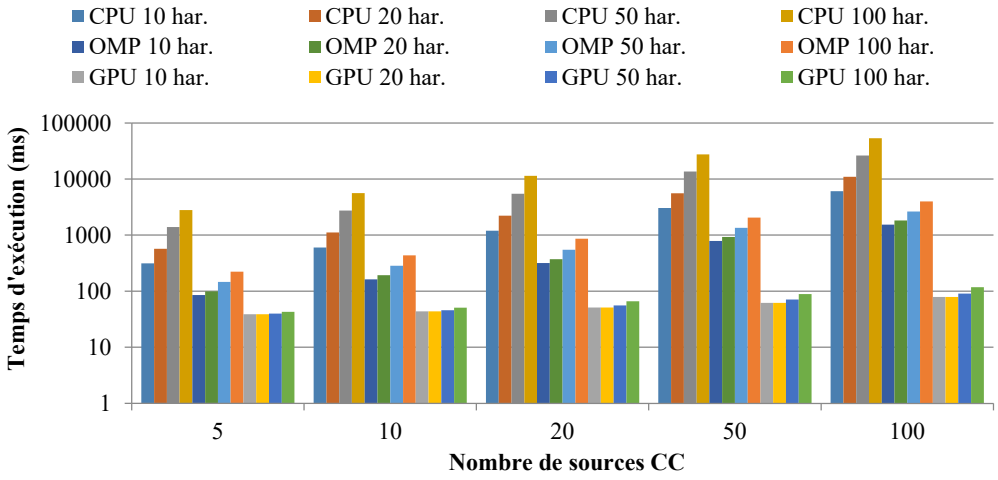


Figure 6.9: Temps d'exécution du PSO séquentiel sur CPU, du PSO parallèle sur CPU (31 threads OpenMP®) et du PSO parallèle sur GPU pour la minimisation des harmoniques d'onduleurs multiniveaux pour différents nombres de sources et harmoniques considérées (moyenne de 10 essais, E5-2650, K20c)

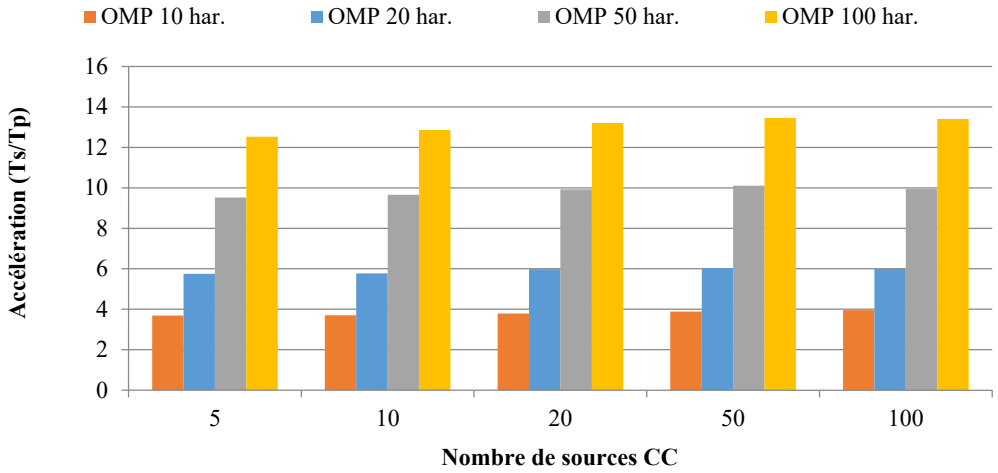


Figure 6.10: Accélération du PSO parallèle sur CPU (31 threads OpenMP®) pour la minimisation des harmoniques d'onduleurs multiniveaux pour différents nombres de sources et harmoniques considérées (moyenne de 10 essais, E5-2650, K20c)

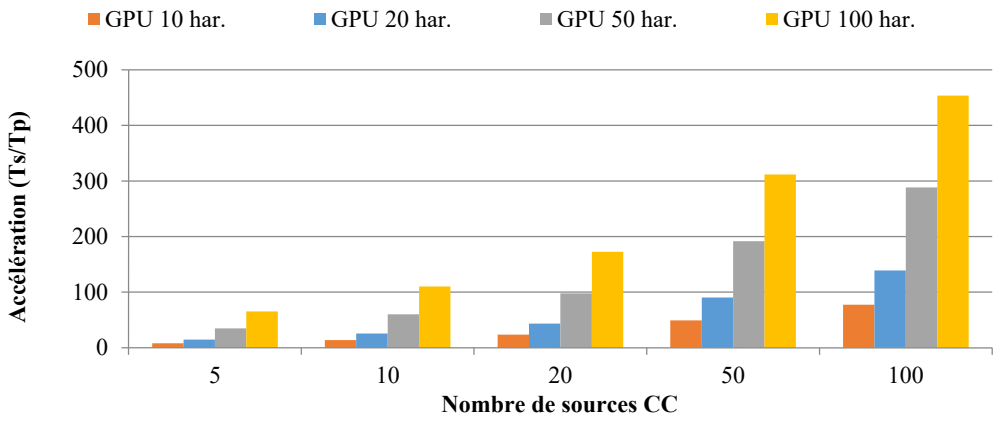


Figure 6.11: Accélération du PSO parallèle sur GPU pour la minimisation des harmoniques d'onduleurs multiniveaux pour différents nombres de sources et harmoniques considérées (moyenne de 10 essais, E5-2650, K20c)

6.6 Méthode d'optimisation directe

Dans ce chapitre, nous avons démontré comment le cadriciel *gpuMF* permet d'accélérer le calcul des angles de communication optimaux d'un onduleur multiniveau afin de minimiser les harmoniques de la tension alternative de sortie. En exploitant l'architecture massivement parallèle du GPU, la méthode proposée permet de contrôler des onduleurs avec un nombre de sources bien plus grand que les méthodes antérieures tout en assurant un temps de calcul minimum pour un contrôle qui réagit rapidement aux changements de tension des sources. La méthode proposée est basée sur les métaheuristiques, elle n'impose aucune limite sur l'indice de modulation et elle permet de spécifier précisément le nombre d'harmoniques à considérer. Or, dans le cas où l'on voudrait considérer toutes les harmoniques lors de la minimisation du THD, il est possible d'utiliser une approche analytique. Celle-ci a été développée par Liu, Hong et Huang [32] et implémentée sur une matrice prédiffusée programmable par l'utilisateur (FPGA). Étant donné que cette approche calcule directement les angles de commutation, elle est beaucoup plus rapide que celles basées sur les métaheuristiques. Cependant, cette méthode analytique impose une limite inférieure sur l'indice de modulation. De plus, l'implémentation sur FPGA publiée dans [32] se limite à un onduleur à trois sources et ne peut être facilement modifiée pour un onduleur avec un plus grand nombre de sources. Toutefois, dépendamment de l'application, cette méthode analytique peut être avantageuse étant donné sa rapidité d'exécution. Dans cette section, nous proposons une implémentation parallèle sur GPU de la méthode analytique publiée dans [32] afin de réduire d'avantage son temps d'exécution et de considérer des onduleurs avec de très grands nombres de sources.

6.6.1 Formulation mathématique

6.6.1.1 L'Énoncé du problème

Dans la référence [32], Liu, Hong et Huang dérivent des formules mathématiques pour calculer les angles de commutation optimaux d'un onduleur multiniveau. Dans cette section, nous dérivons essentiellement les mêmes formules, mais en utilisant des calculs et des notations un peu plus simples.

Considérons un onduleur avec une sortie en escalier à *(2s+1)* niveaux. Cet onduleur a s pas de tension positive d'amplitude E_i pour $i = 1$ à s, s pas de tension négative avec les mêmes amplitudes et un niveau avec une tension de zéro. Pour simplifier la notation, utilisons la variable "t" au lieu de "ωt" où ω est la pulsation de la tension alternative. Les angles de commutations pour une tension dans le domaine $[0, \pi]$ sont θ_i où $i = 1$ à s, avec $0 < \theta_i < \theta_j < \pi/2$ si $i < j$.

Définissons ensuite P, la fonction d'impulsion de l'unité périodique de période 2π, qui est impaire et définie en termes de la fonction de Heaviside H comme :

$$P(\theta, t) = H(t - \theta)\, H(\pi - \theta - t) \qquad \forall t \in [0, \pi] \tag{6.12}$$

Sa valeur est de un dans le sous-intervalle $[\theta,\ \pi - \theta]$ et zéro ailleurs dans l'intervalle $[0,\ \pi]$. En termes de P, la tension de sortie de l'onduleur peut être représentée comme :

$$V(t) = \sum_{k=1}^{s} E_k\ P(\theta_k, t). \tag{6.13}$$

Les coefficients de Fourier pairs de V sont tous nuls tandis que les coefficients impairs sont égaux à :

$$V_n = \frac{4}{n\pi} \sum_{k=1}^{s} E_k\ cos(n\theta_k) \tag{6.14}$$

La première composante de V, qui est $V_1 * sin(t)$, a la variation temporelle de la tension de sortie désirée. L'indice de modulation m est défini comme :

$$m = \frac{\pi V_1}{4\ E_{max}} \qquad \text{avec} \qquad E_{max} = \sum_{k=1}^{s} E_k \tag{6.15}$$

Le taux de distorsion harmonique total (THD) de la tension produite par l'onduleur est défini comme :

$$THD = \frac{1}{V_1}\sqrt{\sum_{n=3,5,\ldots}^{\infty} V_n^2} \tag{6.16}$$

Le problème considéré consiste donc à trouver les angles de commutation θ_i de façon à minimiser le THD tout en s'assurant que l'équation (6.15) demeure valide.

6.6.1.2 Le Problème d'optimisation

Étant donné que V_1 est constant d'après l'équation (6.15), on peut déduire à partir de l'équation (6.16) que le THD est minimum lorsque $\sum_{n=3,5,\ldots}^{\infty} V_n^2$ est minimum. Cette somme peut être calculée à l'aide du théorème de Parseval qui dit que :

$$\frac{\pi}{2}\sum_{n=1,3,5,\ldots}^{\infty} V_n^2 = \int_0^{\pi} [V(t)]^2\, dt, \tag{6.17}$$

ou de façon équivalente :

$$\frac{\pi}{2}\sum_{n=3,5,\ldots}^{\infty} V_n^2 = -\frac{\pi V_1^2}{2} + \int_0^{\pi} [V(t)]^2 \;\; dt. \tag{6.18}$$

Étant donné que V_1 est constant d'après l'équation (6.15), le THD minimum se produit donc lorsque l'intégrale du côté droit de l'équation (6.18) est minimum. Représentons la valeur de cette intégrale à l'aide de la notation $F(\vec{\theta})$, où $\vec{\theta} = [\theta_1, \theta_2, \ldots, \theta_s]$. La fonction $F(\vec{\theta})$ est facilement calculée à l'aide de la relation :

$$P(\theta_i, t)\, P(\theta_j, t) \;=\; P(\theta_j, t) \qquad \text{lorsque} \qquad i \le j, \tag{6.19}$$

ce qui implique

$$F(\vec{\theta}) = \int_0^{\pi} \left[\sum_{k=1}^{s} E_k^2 \;\; P(\theta_k, t) + 2\sum_{k=1}^{s-1} E_k \sum_{j=k+1}^{s} E_j \;\; P(\theta_j, t)\right] \; dt. \tag{6.20}$$

Étant donné que

$$\int_0^{\pi} P(\theta_k, t)\, dt = \pi - 2\theta_k \qquad \forall\ k, \tag{6.21}$$

il est vrai que

$$F(\vec{\theta}) = \sum_{k=1}^{s} E_k^2(\pi - 2\theta_k) + 2\sum_{k=1}^{s-1} E_k \sum_{j=k+1}^{s} E_j(\pi - 2\theta_j). \tag{6.22}$$

Le problème d'optimisation en question peut donc être résumé comme suit : déterminer les angles $\theta_1, \theta_2, \ldots, \theta_s$, de façon à minimiser $F(\vec{\theta})$, tout en respectant la contrainte

$$g(\vec{\theta}) = \sum_{k=1}^{s} E_k\, cos(\theta_k) - mE_{max} = 0, \tag{6.23}$$

qui correspond à l'équation (6.15).

6.6.1.3 Solution

Selon la méthode des multiplicateurs de Lagrange, la solution à ce problème d'optimisation avec contrainte peut être obtenue en solutionnant pour $\vec{\theta}$ l'équation:

$$\frac{\partial F(\vec{\theta})}{\partial \theta_i} - \lambda \frac{\partial g(\vec{\theta})}{\partial \theta_i} = 0 \qquad \forall i\ = 1, .., s \tag{6.24}$$

Dans laquelle λ est le multiplicateur de Lagrange constant, ensemble avec l'équation (6.24). Ces équations sont :

$$-2\ E_1^2\ +\ \lambda\ E_1 sin(\theta_1) = 0 \qquad \text{pour} \qquad i = 1 \tag{6.25}$$

$$-2E_i^2 - 4E_i \sum_{k=1}^{i-1} E_k + \lambda E_i sin(\theta_i) = 0 \qquad \text{pour} \qquad i \neq 1 \tag{6.26}$$

Ainsi, en définissant

$$\eta = \frac{4}{\lambda} \qquad \text{et} \qquad u_i = \sum_{k=1}^{i} E_k - \frac{E_i}{2} \tag{6.27}$$

il est possible d'écrire la formule générale pour les angles comme suit :

$$sin(\theta_i) = \eta \ u_i \tag{6.28}$$

et la contrainte à l'équation comme suit :

$$\sum_{k=1}^{s} E_k \sqrt{1 - (\eta u_k)^2} = m \ E_{max}. \tag{6.29}$$

6.6.1.4 Limites sur l'indice de modulation

Le côté de gauche de l'équation (6.29) est monotone décroissant en terme de η. Il est donc maximum lorsque η est minimum et minimum lorsque η est maximum. Équation (6.28) implique que $\eta \geq 0$ de sorte que l'équation (6.29) implique que $m_{max} \leq 1$. D'un autre côté, puisque $sin(\theta_i) < 1 \ \forall i$, équation (6.28) nécessite que $\eta u_i < 1 \ \forall i$ et puisque $u_i < u_s \ \forall i$, alors $\eta_{max} < 1/u_s$. Conséquemment, l'indice de modulation m est borné inférieurement puisque

$$m_{min} > \frac{1}{E_{max}} \sum_{k=1}^{s} E_k \sqrt{1 - \left[\frac{u_k}{u_s}\right]^2}. \tag{6.30}$$

6.6.1.5 Détermination de η

Nous notons que le côté gauche de l'équation (6.29) dépend de η^2, et donc, pour simplifier les choses, nous définissions $\gamma = \eta^2$ et solutionnons l'équation suivante pour γ:

$$f(\gamma) = 0 \qquad \text{avec} \qquad f(\gamma) = \sum_{k=1}^{s} E_k \sqrt{1 - \gamma \ u_k^2} - m \ E_{max} \tag{6.31}$$

Cette équation ne peut toujours être résolue exactement; toutefois, une solution numérique peut facilement être obtenue avec la méthode de Newton-Raphson. Selon ce procédé, la valeur de γ est obtenue en calculant la limite de la formule itérative suivante:

$$\gamma_{i+1} = \gamma_i - \frac{f(\gamma_i)}{f'(\gamma_i)}, \tag{6.32}$$

dans laquelle

$$f'(\gamma) = -\frac{1}{2}\sum_{k=1}^{s}\frac{E_k u_k^2}{\sqrt{1-\gamma\ u_k^2}}. \tag{6.33}$$

Étant donné que l'équation (6.31) est monotone décroissante en termes de γ, une autre approche possible pour trouver le zéro de la fonction est la méthode de bissection. Cette méthode est similaire à une recherche binaire et consiste à diviser l'intervalle des valeurs possibles pour γ et à évaluer les bornes de chaque sous-intervalle afin d'identifier la location du zéro. Le processus continue jusqu'à ce que la précision désirée pour la valeur de γ soit atteinte. Pour une implémentation séquentielle, la division est habituellement faite de façon à produire deux sous-intervalles à chaque itération, d'où le lien avec la recherche binaire. Dans le cas d'une implémentation parallèle, un nombre bien plus grand de sous-intervalles peut-être évalué concurremment à chaque itération afin d'accélérer la recherche. La méthode de bissection a été implémentée sur GPU dans [246] pour la solution de systèmes d'équations polynomiales et dans [247] pour la simulation de systèmes stellaires denses. Dans les deux cas, les résultats obtenus sont positifs ce qui nous motive à tester cette méthode pour le calcul des angles optimaux d'un onduleur multiniveau. Grâce à l'architecture massivement parallèle du GPU, il est possible d'évaluer concurremment un très grand nombre de sous-intervalles et d'arriver à la précision désirée en seulement une ou deux itérations. La méthode de bissection a donc le potentiel de calculer une solution à l'équation (6.31) dans un nombre d'itérations plus petit que la méthode de Newton-Raphson. Une comparaison expérimentale est toutefois nécessaire afin d'évaluer la performance de chacune.

6.6.2 Implémentation parallèle sur GPU

Dans cette section, nous présentons les détails d'implémentation du programme parallèle proposé pour le calcul des angles de commutation optimaux d'un onduleur multiniveau. Deux algorithmes ainsi que trois implémentations parallèles sont considérés.

6.6.2.1 Algorithmes d'optimisation

Le premier algorithme proposé utilise la méthode de Newton-Raphson pour trouver la valeur de γ qui satisfait l'équation (6.31). Dans le cas spécifique du calcul des angles de commutation, le nombre d'itérations nécessaire pour atteindre une précision de 1E-6 est généralement de six ou moins. Similairement, le deuxième algorithme calcule lui aussi le zéro de la fonction à l'équation (6.31) afin de déterminer les angles de commutation, mais utilise quant à lui la méthode de bissection. Ce deuxième algorithme effectue *N* coupures sur l'intervalle des valeurs

possibles pour γ et évalue $f(\gamma)$ pour chacune. Étant donné que la fonction $f(\gamma)$ est strictement monotone décroissante, il y aura un seul endroit où $f(\gamma_{j-1})$ est positif et $f(\gamma_j)$ est négatif. Le zéro de la fonction réside nécessairement dans ce sous-intervalle. Pour augmenter la précision, une itération subséquente peut être effectuée sur le sous-intervalle $[\gamma_{j-1},\ \gamma_j]$. Dans notre implémentation, nous utilisons deux itérations et effectuons 1024 coupures par itération. Dans le cas où $u_s = 1$, ces paramètres permettent d'identifier le zéro de la fonction avec une précision de $1/1024^2$ ou 4.77E-7. Nous présentons aux algorithmes 6.1 et 6.2 le pseudocode pour les implémentations séquentielles des deux algorithmes et discutons à la section suivante comment nous les parallélisons afin de réduire leur temps d'exécution.

Algorithme 6.1: Calcule des angles optimaux utilisant la méthode de Newton-Raphson

1: Calculer $E_{max} = \sum_{k=1}^{s} E_k$
2: Calculer $u_i = \sum_{k=1}^{i}(E_k - E_i/2)$ pour *i = 1..s*
3: Assigner une valeur initiale à γ où $0 < \gamma < 1/u_s^2$
4: Assigner l'indice de modulation désiré à m_{ref}
5: Calculer $m_{calc} = \left(\sum_{k=1}^{s} E_k \sqrt{1 - \gamma\ u_k^2}\right)/E_{max}$
6: Initier le compteur d'itérations $iteration = 0$
7: **Tant que** $\left((iteration < iteration_{max})$ **ET** $\left(\left|m_{ref} - m_{calc}\right| > tolerance\right)\right)$ {
8: Calculer $f(\gamma) = \left(\sum_{k=1}^{s} E_k \sqrt{1 - \gamma\ u_k^2}\right) - m\ E_{max}$
9: Calculer $f'(\gamma) = -\frac{1}{2}\sum_{k=1}^{s}\left(E_k u_k^2 / \sqrt{1 - \gamma\ u_k^2}\right)$
10: Calculer $\gamma = \gamma - f(\gamma)/f'(\gamma)$
11: Calculer $m_{calc} = \left(\sum_{k=1}^{s} E_k \sqrt{1 - \gamma\ u_k^2}\right)/E_{max}$
12: $iteration = iteration + 1$
13: }
14: Calculer $\eta = \sqrt{\gamma}$
15: Calculer les angles $\theta_i = asin(\eta * u_i)$ pour *i = 1..s*
16: **Retour**

Algorithme 6.2 : Calcule des angles optimaux utilisant la méthode de bissection avec *N* itérations

1: Calculer $E_{max} = \sum_{k=1}^{s} E_k$
2: Calculer $u_i = \sum_{k=1}^{i}(E_k - E_i/2)$ pour $i = 1..s$
3: Assigner $\gamma_{min} = 0$ et $\gamma_{max} = 0.999 * 1/u_s$
4: Assigner $\gamma_{best} = \gamma_{min}$
5: **Pour** ($i = 1 ; j \leq N_{itérations} ; i++$) {
6: Assigner $\gamma_0 = \gamma_{best}$
7: Calculer $step = (\gamma_{max} - \gamma_{min})/(N_{cuts} - 1)^i$
8: Calculer $f(\gamma_0)$
9: **Pour** ($j = 1 ; j < N_{cuts} ; j++$) {
10: Incrémenter $\gamma_j = \gamma_{best} + j * step$
11: Calculer $f(\gamma_j)$
12: **Si** ($f(\gamma_{j-1}) > 0$ *AND* $f(\gamma_j) < 0$) {
13: $\gamma_{best} = \gamma_{j-1}$
14: }
15: }
16: }
17: Calculer $\gamma_{final} = \gamma_{best} + 0.5 * (\gamma_{max} - \gamma_{min})/(N_{cuts} - 1)^{N_{passes}}$
18: Calculer $\eta = \sqrt{\gamma_{final}}$
19: Calculer les angles $\theta_i = asin(\eta * u_i)$ pour $i = 1..s$
20: **Retour**

Avant de poursuivre avec les implémentations parallèles, nous voulons souligner une particularité dans l'algorithme 6.2. Si l'on regarde la condition à la ligne 12, on peut noter une certaine inefficacité au niveau du code. Dans une implémentation purement séquentielle, étant donné que $f(\gamma)$ est strictement monotone décroissante, une approche préférable pour trouver le zéro de la fonction serait de quitter la boucle aussitôt que $f(\gamma_j) \leq 0$. Ceci permettrait de réduire le temps d'exécution de l'algorithme en évitant d'effectuer des itérations inutiles. Cependant, le pseudocode à l'algorithme 6.2 a été écrit de façon à permettre une parallélisation directe où les itérations de la boucle à la ligne 9 sont effectuées en parallèle. Dans ce cas, plusieurs threads vont rencontrer une valeur de $f(\gamma_j)$ négative et la condition $f(\gamma_j) \leq 0$ ne permettra pas d'identifier le zéro. Cependant, le pseudocode à l'algorithme 6.2 permet d'identifier correctement le zéro de la fonction puisqu'il existe seulement deux threads voisins pour lesquelles les valeurs de $f(\gamma_j)$ ont des signes différents. Le zéro de la fonction réside alors sur le sous-intervalle délimité par ces deux threads.

6.6.2.2 Stratégie de parallélisation

1) CPU-1 : Newton-Raphson séquentiel sur CPU

Le programme CPU-1 est utilisé comme référence pour évaluer l'accélération des implémentations parallèles. Le programme est une implémentation séquentielle de l'algorithme 6.1 et est exécuté sur le CPU. Le code a été développé avec une attention particulière pour assurer la meilleure performance possible. À titre d'exemple, la valeur calculée pour m_{calc} à la ligne 11 est gardée en mémoire et réutilisée à l'itération suivante lorsque $f(\gamma)$ est évaluée afin d'éviter d'avoir à recalculer la sommation à la ligne 8.

2) GPU-1 : Newton-Raphson parallèle sur GPU

Le programme GPU-1 est une implémentation parallèle sur GPU de l'algorithme 6.1. Il utilise un seul bloc de threads CUDA™ et le nombre de threads est égal au nombre de sources de tension de l'onduleur. Toutes les opérations sont effectuées à l'intérieur d'un même *kernel* afin de minimiser la communication entre le CPU et le GPU et de garantir la meilleure performance possible. Le calcul de E_{max} à la ligne 1 de l'algorithme 6.1 est effectué à l'aide d'une opération de réduction parallèle tel qu'illustrée à la Figure 6.12 pour le cas spécifique d'un onduleur à huit sources dont les tensions sont identiques et égalent à une unité réduite (p.u.). Cette réduction est basée sur [248] et prend $log(s)$ étapes à compléter comparativement à s étapes pour une implémentation séquentielle où s est le nombre de sources de tension. Ensuite, un scan parallèle inclusif basé sur le réseau de Kooge-Stone [235] est utilisé pour calculer concurremment tous les u_i à la ligne 2. Cette opération est illustrée à la Figure 6.13, encore ici, pour un onduleur à huit sources dont les tensions sont toutes égales à 1 p.u.. Comme nous l'avons discuté précédemment à la section 4.7.3, le scan parallèle basé sur le réseau de Kooge-Stone souffre d'une mauvaise efficacité en termes de travail et il est habituellement préférable d'utiliser un autre réseau, tel que celui de Brent-Kung [235], qui offre une bien meilleure efficacité. Cependant, pour le problème de minimisation des harmoniques, le nombre de sources de tension considérées est habituellement plus petit que le nombre de cœurs présents sur le GPU et la méthode la plus rapide n'est pas nécessairement celle qui est la plus efficace en termes de travail, mais celle qui nécessite le plus petit nombre d'étapes pour compléter. Conséquemment, le réseau de Kooge-Stone est préférable à celui de Brent-Kung pour le calcul des u_i à la ligne 2 puisqu'il nécessite seulement $log(s)$ étapes à compléter comparativement à $2 * log(s)$ étapes pour le réseau de Brent-Kung et s étapes pour une implémentation séquentielle. Finalement, le calcul de $f(\gamma)$, de $f'(\gamma)$ et de m_{calc} aux lignes 8, 9 et 11 est effectué à l'aide de deux réductions parallèles. Tout comme pour le programme séquentiel, l'implémentation sur GPU de la méthode de Newton-Raphson réutilise la valeur de m_{calc} pour le calcul de $f(\gamma)$ à l'itération suivante.

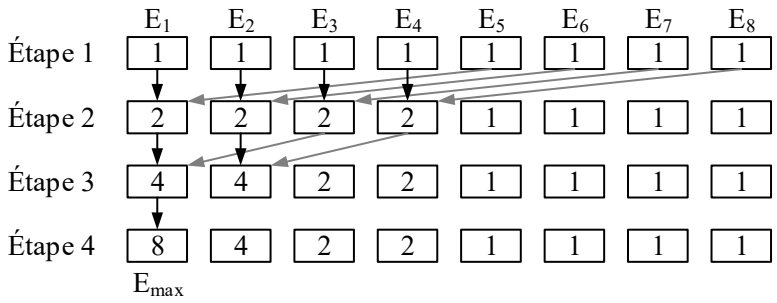


Figure 6.12: Réduction parallèle pour le calcul du terme E_{max}

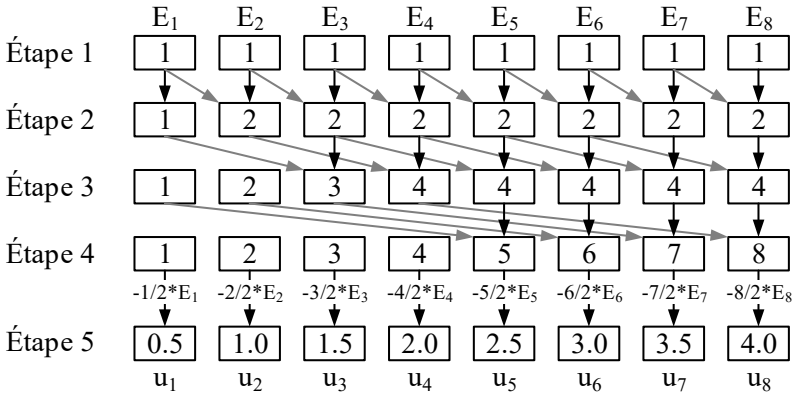


Figure 6.13: Scan parallèle inclusif pour calculer les valeurs de u_i

3) GPU-2 : Bissection parallèle sur GPU, un thread par coupure

Le programme GPU-2 implémente l'algorithme 6.2 sur le GPU. Le flux d'exécution est illustré sur le diagramme à la Figure 6.14 où N est le nombre de coupures et s est le nombre de sources de l'onduleur. Contrairement à la méthode de Newton-Raphson, la méthode de bissection nécessite uniquement deux itérations. À la première itération (étapes 1 et 2), le programme utilise un thread CUDA™ par coupure et calcul en parallèle les valeurs de $f(\gamma_i)$ pour tous les $i = 1..N$ de façon à couvrir l'intervalle des valeurs possibles pour γ. Chaque thread compare ensuite le signe de sa valeur calculée à celui de la valeur du thread voisin. Puisque $f(\gamma)$ est strictement monotone décroissant, il y aura exactement un seul thread j pour lequel $f(\gamma_{j-1})$ est positif et $f(\gamma_j)$, négatif. Le zéro de $f(\gamma)$ se situe alors dans l'intervalle $[\gamma_{j-1}, \gamma_j]$. À la Figure 6.14, c'est le thread $j = 3$ qui détecte l'intervalle où se trouve le zéro. Une deuxième itération (étapes 3 et 4) est ensuite exécutée de la même

manière afin de localiser le zéro à l'intérieur du sous-intervalle avec une meilleure précision. Une fois identifiée, cette valeur est utilisée pour calculer η à l'étape 5. Finalement, s threads CUDA™ sont utilisés pour calculer en parallèle les s angles de commutation à l'étape 6. Si l'on assume que le nombre de cœurs sur le GPU est suffisant pour calculer tous les $f(\gamma_j)$ simultanément, la méthode de bissection a l'avantage de nécessiter un nombre plus petit d'itérations que la méthode de Newton-Raphson. De plus, chaque itération de la méthode de bissection nécessite le calcul d'une seule sommation contrairement à deux pour Newton-Raphson.

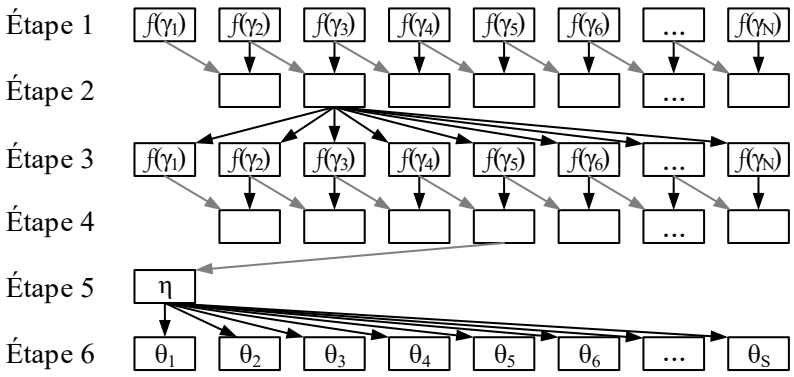


Figure 6.14: Bissection parallèle

4) GPU-3 : Bissection parallèle sur GPU, un bloc de threads par coupure

Finalement, le programme GPU-3 est une deuxième tentative pour paralléliser l'algorithme de bissection sur GPU. À la Figure 6.14, on peut remarquer que le calcul de $f(\gamma_j)$ à l'étape 1 nécessite que chaque thread effectue la sommation de s termes d'après l'équation (6.31). Pour un onduleur avec un grand nombre de sources, cette opération séquentielle peut être longue à exécuter et devrait être évitée. Dans le but d'exploiter un niveau de parallélisme supérieur et de réduire le nombre de sous-étapes nécessaires, cette sommation peut être remplacée par une réduction parallèle semblable à celle illustrée à la Figure 6.12. Pour cela, nous modifions les étapes 1 et 3 à la Figure 6.14 afin d'utiliser un bloc de thread CUDA™ pour chaque $f(\gamma_j)$. Le nombre de threads nécessaire pour calculer les N coupures en parallèle augmente alors de N à $N * s$ tandis que le nombre de sous-étapes diminue de s à $log(s)$. Ce niveau de parallélisme plus élevé devrait augmenter la rapidité de l'algorithme, spécialement pour des onduleurs avec un grand nombre de sources de tension.

6.6.3 Résultats expérimentaux

Dans cette section, nous présentons les résultats obtenus par les algorithmes proposés. Afin d'être consistant, l'ordre dans lequel nous présentons les résultats est le même que pour la méthode précédente, soit celle basée sur les métaheuristiques. Dans un premier test, afin de démontrer le bon fonctionnement de l'approche mathématique proposée, nous calculons et nous affichons à la Figure 6.15 les angles de commutations optimaux d'un onduleur à 10 sources dont les tensions sont les mêmes qu'à l'exemple précédent, soit : $V_{DC1} = 1.10$, $V_{DC2} = 1.04$, $V_{DC3} = 0.98$, $V_{DC4} = 1.02$, $V_{DC5} = 1.08$, $V_{DC6} = 0.92$, $V_{DC7} = 0.96$, $V_{DC8} = 1.06$, $V_{DC9} = 0.94$ et $V_{DC10} = 0.90$ p.u.. Les limites sur l'indice de modulation sont déterminées d'après les équations (6.29) et (6.30) et fixées à $m_{min} = 0.758$ et $m_{max} = 1.0$. Le THD associé aux angles obtenus est calculé à l'aide de l'équation (6.4) en incluant les 100 premières harmoniques et est affiché à la Figure 6.16. Tout comme pour l'exemple précédent, nous incluons aussi sur la Figure 6.16 le THD pour des onduleurs de 5, 15 et 20 sources afin de démontrer l'avantage d'utiliser un grand nombre de sources pour réduire le THD résultant. On peut remarquer sur le diagramme du THD que la limite inférieure pour l'indice de modulation varie pour chacun des tests effectués. En fait, cette limite varie d'après le nombre de sources de tension suivant l'équation (6.30). Cette limite représente un désavantage important de la méthode analytique comparativement aux métaheuristiques.

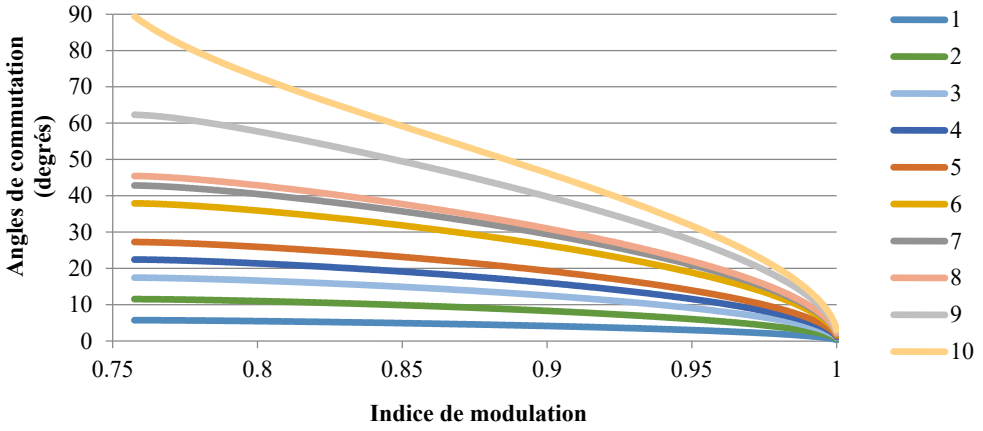


Figure 6.15: Angles de commutation optimaux calculés par la méthode directe sur GPU pour un onduleur à 10 sources et un indice de modulation variant de 0.76 à 1.0

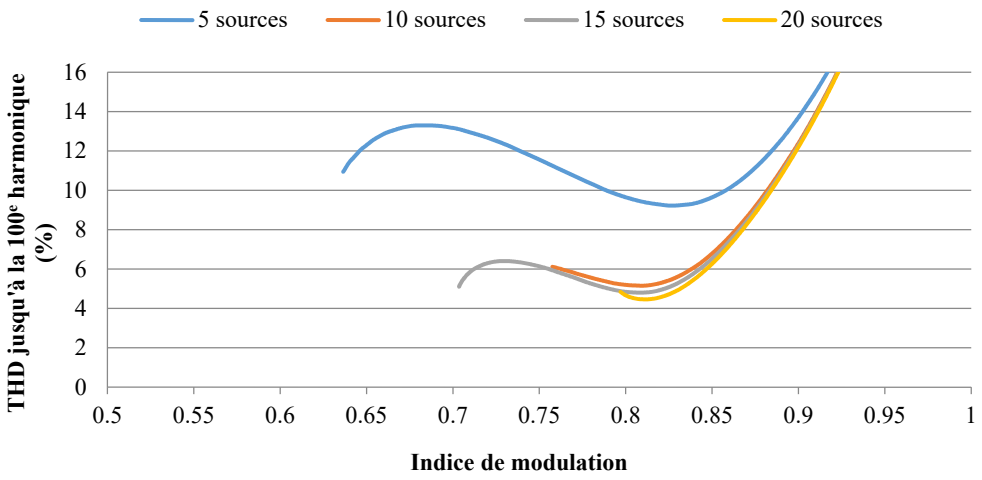


Figure 6.16: Taux de distorsion harmonique de la tension de sortie considérant les 100 premières harmoniques

Dans un second test, nous fixons l'indice de modulation à 0.8 et calculons les angles optimaux pour l'onduleur à 10 sources. La tension des sources ainsi que les angles calculés sont listés au Tableau 6.3. La sortie en escalier produite par l'onduleur est représentée sous forme d'un diagramme de front d'onde à la Figure 6.17. Tout comme nous l'avons fait précédemment, cette sortie en escalier est analysée à l'aide d'une transformation FFT et l'amplitude des harmoniques est affichée à la Figure 6.18. La distorsion harmonique totale est de 5.191%.

TABLEAU 6.3
ANGLES DE COMMUTATION OPTIMAUX CALCULÉS PAR LA MÉTHODE DIRECTE SUR GPU POUR UN ONDULEUR À 10 SOURCES ET UN INDICE DE MODULATION DE 0.8

Source	Tension (p.u.)	Angle de commutation (rad)	Source	Tension (p.u.)	Angle de commutation (rad)
V_{dc1}	1.10	0.0956	V_{dc6}	0.92	0.6267
V_{dc2}	1.04	0.1920	V_{dc7}	0.96	0.7062
V_{dc3}	0.98	0.2904	V_{dc8}	1.06	0.7480
V_{dc4}	1.02	0.3730	V_{dc9}	0.94	1.0066
V_{dc5}	1.08	0.4526	V_{dc10}	0.90	1.2675

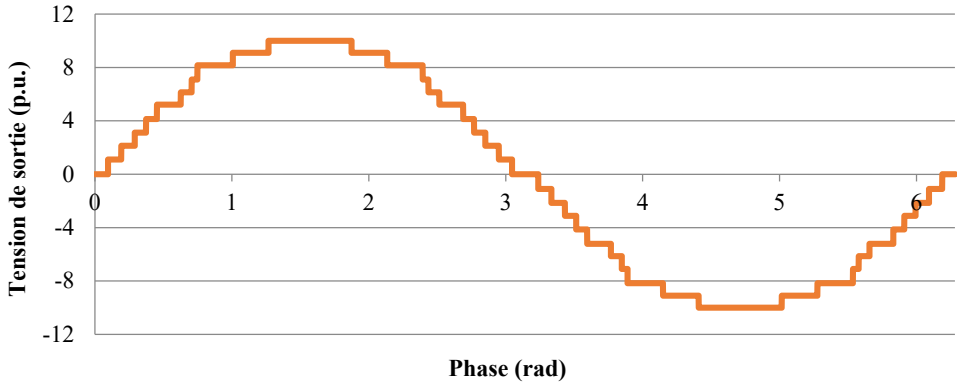


Figure 6.17:Tension de sortie générée par les angles de commutation optimaux calculés par la méthode directe sur GPU pour l'onduleur à 10 sources et un indice de modulation de 0.8

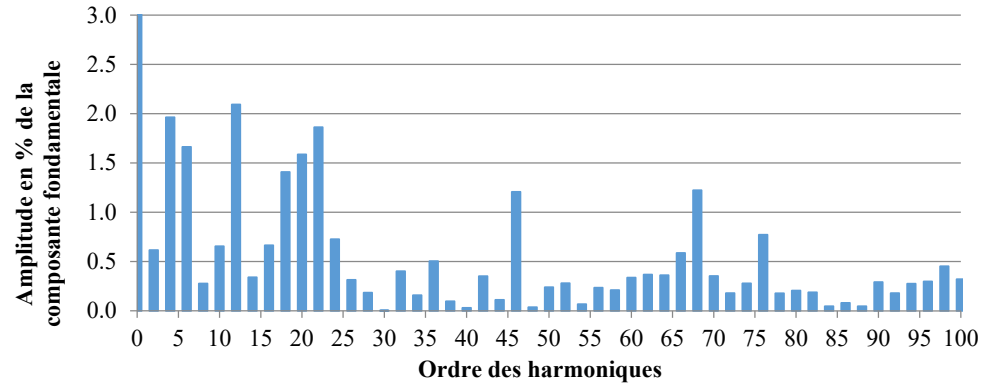


Figure 6.18: Amplitude (en % de la composante fondamentale) des harmoniques de la tension de sortie illustrée à la Figure 6.17. Malgré que l'axe vertical s'arrête à 3.0, l'amplitude de la composante fondamentale se rend à exactement 100%. (THD = 5.191%)

Dans un dernier test, afin de démontrer l'avantage d'une implémentation parallèle sur GPU et d'identifier la stratégie de parallélisation la plus efficace, nous calculons les angles optimaux à l'aide de l'implémentation séquentielle sur CPU (CPU1) et comparons le temps d'exécution à ceux des implémentations parallèles sur GPU (GPU1, GPU2 et GPU3). Les temps d'exécution sont mesurés en microsecondes (µs) et listés au Tableau 6.4 pour différents onduleurs dont le nombre de sources augmente. Il est important de noter que les quatre implémentations produisent les mêmes résultats avec une précision de 1^{E}-6, un paramètre spécifié par l'utilisateur. Tout comme pour la méthode basée sur les métaheuristiques, le programme séquentiel est exécuté sur un CPU Intel Xeon E5-2650. Cependant, les implémentations

parallèles sont exécutées sur un GPU NVIDIA® GTX 750 Ti SC. Contrairement au processeur K20C que nous avons utilisé précédemment, le GTX 750 possède uniquement 640 cœurs, mais bénéficie d'une fréquence d'opération plus élevée de 1255 MHz et d'un délai d'exécution inférieur. Dans le cas de la minimisation des harmoniques à l'aide de la méthode directe, la quantité de calculs à effectuer ne permet pas d'exploiter pleinement la capacité de la carte K20C et une meilleure performance est observée à l'aide de la carte GTX 750.

D'après les temps d'exécutions listés au Tableau 6.4, il est évident que l'implémentation GPU-1 (Newton-Raphson parallèle sur GPU) offre la meilleure performance. L'implémentation GPU-2 (bissection parallèle sur GPU, 1 thread par coupure) exploite un niveau de parallélisme plus élevé étant donné que les 1024 coupures sont évaluées simultanément, mais chacune de ces évaluations se fait séquentiellement. Finalement, l'implémentation GPU-3 (bissection parallèle sur GPU, 1 bloc de threads par coupure) augmente davantage le niveau de parallélisme exploité puisque l'évaluation de chacune des coupures est faite à l'aide d'une réduction parallèle. L'accélération qui en résulte est très avantageuse lorsque le nombre de sources est large, mais moindre que pour l'implémentation GPU-1. Ceci s'explique par le fait que la méthode de bissection est beaucoup moins efficace en termes de travail que la méthode de Newton-Raphson. Bien que la puissance de calcul du GPU soit énorme, la quantité de calculs requise par l'algorithme GPU-3 est trois ordres de magnitude plus élevée que pour les algorithmes CPU-1 et GPU-1. Bien que la méthode de bissection semble prometteuse, c'est la méthode de Newton-Raphson parallélisée sur GPU qui offre la meilleure accélération comparée à une implémentation séquentielle sur CPU.

Comparée aux solutions basées sur les métaheuristiques, la méthode directe présentée ici est beaucoup plus rapide. Par exemple, à la section 6.5, le plus petit temps mesuré pour le PSO était de 38.7 ms. Dans le cas de la méthode directe sur GPU, le plus grand temps mesuré est de seulement 16.41 µs, ce qui représente une différence de plus de trois ordres de magnitude. Cependant, il faut se souvenir que la méthode directe impose une limite sur l'indice de modulation ce qui représente un désavantage important dans certaines situations. Afin d'offrir une autre comparaison, une méthode directe similaire à la nôtre a été implémentée sur un FPGA dans la référence [32]. Le temps d'exécution nécessaire au calcul des angles d'un onduleur à trois sources sur le FPGA a été mesuré à 20 µs, ce qui est plus élevé que le temps d'exécution de notre implémentation GPU-1. En fait, notre implémentation est capable d'optimiser un onduleur avec 1000 sources de tension dans un temps plus court ce qui démontre sans aucun doute l'avantage d'une parallélisation sur GPU.

TABLEAU 6.4
TEMPS D'EXÉCUTION ET ACCÉLÉRATION POUR LE CALCUL DES ANGLES DE COMMUTATION OPTIMAUX POUR DIFFÉRENTES GROSSEURS D'ONDULEURS ET IMPLÉMENTATIONS (MOYENNE DE 100 ESSAIS, E5-2650, GTX 750 TI SC)

Nombre de sources	Temps d'exécution (µs)				Accélération ($T_{séq}/T_{par}$)		
	CPU-1	GPU-1	GPU-2	GPU-3	GPU-1	GPU-2	GPU-3
5	11.36	7.61	42.46	34.48	1.5	0.3	0.3
6	11.59	7.62	49.46	34.06	1.5	0.2	0.3
7	11.78	7.56	56.42	34.14	1.6	0.2	0.3
8	11.68	7.04	62.75	33.57	1.7	0.2	0.3
9	16.71	7.57	69.50	34.20	2.2	0.2	0.5
10	16.93	7.37	76.36	34.17	2.3	0.2	0.5
20	19.51	7.65	143.95	35.07	2.6	0.1	0.6
40	28.84	8.23	279.55	42.77	3.5	0.1	0.7
60	44.82	8.30	414.24	43.70	5.4	0.1	1.0
100	115.71	8.33	684.69	64.45	13.9	0.2	1.8
500	2185.60	10.81	3388.08	190.20	202.2	0.6	11.5
1000	8768.78	16.41	6767.71	392.63	534.4	1.3	22.3

Finalement, pour mettre en valeur l'efficacité des deux méthodes proposées dans ce chapitre pour la minimisation des harmoniques d'un onduleur multiniveau, nous comparons au Tableau 6.5 les THD que nous avons obtenues à ceux d'autres auteurs. Nous remarquons que nos deux méthodes produisent un THD inférieur aux autres à cause du plus grand nombre de sources utilisées. Ce nombre est possible grâce à l'utilisation d'une métaheuristique et à la parallélisation sur GPU. Nous notons aussi que notre méthode basée sur le PSO génère un THD inférieur à notre méthode directe. Ceci est normal puisque la fonction de coût que nous avons définie permet une certaine variation (< 0.1%) sur l'amplitude de la composante fondamentale de la tension de sortie ce qui n'est pas le cas pour la méthode directe. Cette petite variation permet au PSO d'ajuster les angles de commutation afin de réduire davantage le THD de la tension de de sortie sans affecter significativement son amplitude.

TABLEAU 6.5
COMPARAISON DU THD DES SOLUTIONS CALCULÉES PAR LES ALGORITHMES PARALLÈLES SUR GPU PROPOSÉS DANS CETTE THÈSE ET D'AUTRES MÉTHODES PUBLIÉES DANS LA LITTÉRATURE

Méthodes	Nombre de sources	Indices de modulation	THD
Réseau de neurones artificielles [103]	5	Pas spécifié	8.70%
Algorithme génétique [106]	3	0.8	8.05%
Essaim de particules [101]	5	0.7	5.42%
Algorithme d'abeilles [13]	3	0.8	12.52%
Méthode directe [32]	3	0.8	12.24%
Essaim de particules proposé sur GPU	10	0.8	3.15%
Méthode directe proposée sur GPU	10	0.8	5.19%

6.7 Conclusion

Dans ce chapitre, nous avons présenté deux méthodes parallèles sur GPU pour la minimisation des harmoniques d'un onduleur multiniveau. La première approche utilise le cadriciel *gpuMF* et un PSO parallèle sur GPU pour calculer les angles optimaux. La deuxième approche se base sur une formulation mathématique différente et calcule la solution par la méthode de Newton ou celle de la bissection. Pour chaque méthode, nous avons présenté la formulation mathématique et proposé une technique d'optimisation. Différentes stratégies de parallélisation ont été développées pour exécuter les algorithmes sur un GPU. Des tests expérimentaux ont ensuite été complétés afin d'identifier l'approche de parallélisation la mieux adaptée à l'architecture du GPU pour chacune des méthodes. La qualité des solutions calculées par les algorithmes proposés a aussi été comparée à celle d'autres approches publiées dans la littérature. Dans tous les cas, nos algorithmes ont permis d'obtenir une tension alternative de sortie avec un THD moins élevé dû au plus grand nombre de sources utilisé.

L'implémentation d'une métaheuristique parallèle sur GPU pour la minimisation des harmoniques d'un onduleur multiniveau représente un défi important et une contribution scientifique significative. Les deux méthodes proposées sont entièrement innovatrices et représentent la première fois que le GPU est utilisé pour résoudre ce problème d'optimisation. Même si le cadriciel *gpuMF* implémente déjà le PSO sur GPU, il faut paralléliser le calcul de la fonction de coût. Cette opération inclut le tri des angles ainsi qu'une boucle imbriquée pour le calcul du THD. Plusieurs stratégies de parallélisation sont possibles et des tests expérimentaux ont été nécessaires afin d'identifier l'approche qui offre le meilleur compromis entre le niveau de parallélisme exploité et la surcharge de travail induite par la parallélisation. L'algorithme résultant exploite pleinement la puissance de calcul du GPU et réduit le temps d'exécution par un facteur de 453.7x comparé à une implémentation séquentielle sur CPU. Il permet l'optimisation d'onduleur avec un nombre de sources beaucoup plus grand tout en produisant des solutions de meilleure qualité que les méthodes antérieures publiées dans la littérature. Dans un contexte plus large, les deux algorithmes parallèles sur GPU que nous avons proposés permettent un avancement significatif dans le domaine des réseaux intelligents.

Chapitre 7
Optimisation de l'écoulement de puissance

Les travaux de recherche complétés dans le cadre de ce chapitre ont été soumis et publiés dans les articles suivants :

V. Roberge, M. Tarbouchi, et F. Okou, "*Optimal Power Flow Based on Parallel Metaheuristics on Graphics Processing Units*," IEEE Transactions on Smart Grid, article soumis, Mai 2015.

V. Roberge, M. Tarbouchi, et F. Okou, "*Parallel Power Flow Analysis on Graphics Processing Units for Concurrent Evaluation of Many Networks*," IEEE Transactions on Smart Grid, vol. PP, no. 99, pp. 1–10, 2015.

7.1 Introduction

La deuxième application considérée dans cette thèse est l'optimisation de l'écoulement de puissance (OPF de l'anglais *optimal power flow*) d'un réseau de transport d'électricité. Le réseau de transport couvre habituellement une grande surface et permet d'acheminer l'énergie électrique des centrales de production aux postes électriques. Sa structure est légèrement maillée de façon à améliorer la fiabilité du système et à offrir une meilleure flexibilité au niveau du contrôle de la production d'énergie. Un exemple d'un système de transport d'électricité est donné à la Figure 7.1 à la page suivante. Il s'agit du réseau test IEEE à 30 bus dont la description complète est disponible dans [249]. Ce réseau est régulièrement utilisé dans la littérature pour valider les algorithmes proposés. Il représente une partie du système d'alimentation électrique américain dans le Midwest des États-Unis en décembre 1961. Ce réseau contient six générateurs, quatre transformateurs, neuf compensateurs statiques d'énergie réactive (SVAR de l'anglais *static VAR compensator*) et 21 charges. Le problème de l'OPF a initialement été formulé par Carpentier en 1962 [109]. Il consiste à calculer les réglages optimaux d'un système de transport d'électricité en régime permanent afin de satisfaire la demande en énergie tout en optimisant une fonction objective. Au cours des années, cette formulation a évolué de façon à mieux représenter la réalité des réseaux de transport d'électricité et à inclure des contraintes de sécurité, des variables de contrôle discrètes et des fonctions multiobjectives. L'OPF est un problème d'optimisation à grande échelle, non linéaire, non convexe et à variables de contrôle mixtes. Les solutions antérieures au problème d'OPF peuvent être divisées en deux catégories : les méthodes déterministes et les

méthodes non déterministes. Comme nous l'avons identifié au Chapitre 2, les méthodes déterministes sont limitées à une optimisation locale, nécessitent une formulation simplifiée du problème et ne peuvent considérer efficacement les variables de contrôle discrètes telles que le rapport des transformateurs ou le réglage des compensateurs statiques. De leur côté, les méthodes non déterministes ou plus spécifiquement les métaheuristiques ont l'avantage de permettre une optimisation globale et de considérer nativement les variables discrètes. De plus, ces méthodes offrent une flexibilité incomparable au niveau de la définition de l'objectif d'optimisation. Toutefois, de par leur fonctionnement basé sur l'amélioration de solutions candidates, les métaheuristiques nécessitent une puissance de calcul énorme ce qui limite leur utilisation à l'optimisation de petits réseaux de transport.

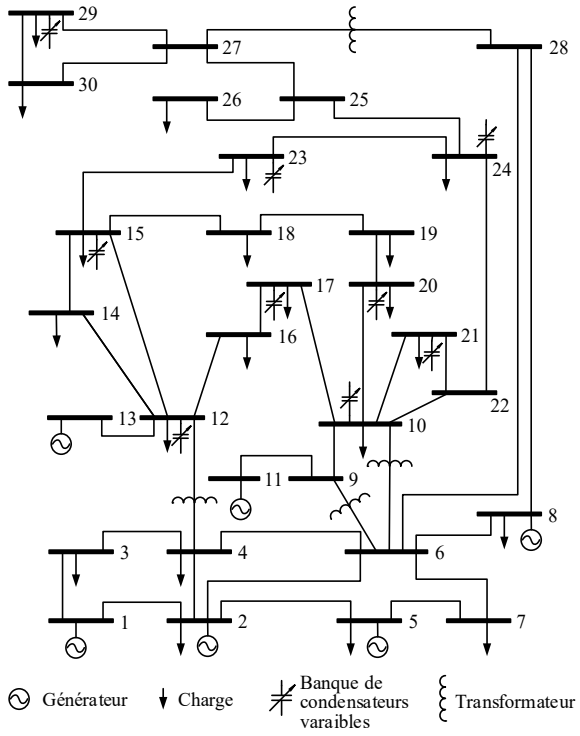


Figure 7.1. Réseau test de transport d'électricité IEEE à 30 bus

Dans le bus de mitiger cette lacune, nous proposons dans ce chapitre l'utilisation du cadriciel *gpuMF* pour l'implémentation d'une métaheuristique parallèle sur GPU pour l'optimisation de l'écoulement de puissance d'un réseau de transport d'électricité. La méthode parallèle proposée utilise l'optimisation par essaim de particules (PSO de l'anglais *particle swarm optimization*) et calcule la puissance et la tension des générateurs, le rapport des transformateurs et la capacitance des

compensateurs statiques afin de minimiser le coût de production, les pertes de transport ou les émissions polluantes. L'algorithme considère les variables de contrôle discrètes et utilise une analyse complète de l'écoulement de puissance suivant la méthode de Newton-Raphson lors de l'évaluation des solutions candidates. De plus, la méthode proposée implémente une approche à multiples phases en explorant à la phase suivante l'espace de recherche autour de la solution obtenue à la phase précédente. Cette stratégie améliore la qualité de la solution finale et permet une meilleure optimisation de la fonction objective. L'efficacité de l'algorithme parallèle proposé est testée sur les réseaux tests IEEE à 30, 118 et 300 bus. Dans tous les cas, l'algorithme proposé permet de calculer des solutions de meilleure qualité que les méthodes antérieures tout en assurant le respect des contraintes de sécurité. Finalement, en exploitant l'architecture massivement parallèle des GPU, le programme développé permet une réduction du temps de calcul d'un facteur de 17.2x comparativement à une exécution séquentielle sur CPU.

Étant donné que la technique d'optimisation proposée dans ce chapitre utilise une métaheuristique, elle requiert nécessairement une analyse de l'écoulement de puissance (PF de l'anglais *power flow*) pour chaque solution candidate à chaque itération. Cette opération complexe et laborieuse représente en fait une partie importante du temps d'exécution du programme. Par conséquent, le développement d'un programme d'OPF sur GPU doit aussi inclure une implémentation parallèle de l'analyse de PF. Pour cette raison, ce chapitre est divisé en deux parties. La première est consacrée à la parallélisation de l'analyse de PF sur GPU et inclus des implémentations des algorithmes de Gauss-Seidel (G-S) et de Newton-Raphson (N-R). La deuxième partie traite quant à elle de l'utilisation de *gpuMF* pour l'optimisation de solutions au problème d'OPF. Chaque partie inclut la formulation du problème, la stratégie utilisée pour le résoudre, la parallélisation sur GPU et les tests expérimentaux afin de vérifier la qualité des solutions trouvées et l'accélération obtenue par l'implémentation parallèle.

7.2 Analyse de l'écoulement de puissance

Plusieurs implémentations parallèles d'algorithmes d'analyse de PF sur GPU ont été discutées au Chapitre 2. Ces implémentations sont toutefois limitées à l'utilisation de matrices pleines qui augmentent arbitrairement la complexité des calculs ou parallélisent uniquement la solution du système d'équations linéaires sans considérer les autres étapes de l'algorithme telles que la construction de la matrice d'admittance, la construction de la matrice Jacobienne ou le calcul de la puissance complexe injectée aux bus. En fait, il n'existe à ce jour aucune solution sur GPU pour l'analyse de PF qui utilise les matrices creuses et qui parallélise l'algorithme en entier.

De façon à combler cette lacune, nous proposons dans cette section des implémentations parallèles sur GPU pour l'analyse simultanée d'un très grand nombre de réseaux de transport d'électricité. Dans le cadre de l'OPF, le module logiciel développé ici permettra à la métaheuristique d'évaluer en parallèle la qualité de toutes les solutions candidates d'un seul coup. Nos implémentations se basent sur un modèle c.a. de l'écoulement de puissance et incluent les algorithmes de G-S et de N-R. Elles n'utilisent que des matrices creuses, parallélisent toutes les étapes des algorithmes et respectent la limite sur la puissance réactive des générateurs. Des tests expérimentaux sont exécutés sur des réseaux de différentes grandeurs allant jusqu'à 2383 bus. Tous les calculs sont effectués avec une précision double et les résultats obtenus sont comparés à ceux de l'outil logiciel MATPOWER [136] afin de valider l'exactitude de nos implémentations. En exploitant l'architecture parallèle des GPU, les implémentations proposées permettent une accélération de 45.2x dans le cas de l'algorithme de G-S et de 17.8x dans le cas de celui de N-R.

7.2.1 Définition du problème

Suivant la formulation mathématique présentée dans [136], les lignes de transport, les transformateurs et les déphaseurs d'un réseau de transport d'électricité comme celui à la Figure 7.1 peuvent être modélisés par une branche en π composé d'une impédance en série $z_s = r_s + jx_s$ (où j représente l'unité imaginaire), d'une capacitance totale b_c et d'un transformateur à décalage de phase idéal dont le rapport de transformation est $N : 1$. À la Figure 7.2, le côté gauche représente l'extrémité de départ de la branche, indiquée par l'indice *f* (de l'anglais « *from* »); tandis que le côté droit représente l'extrémité d'arrivée de la branche, indiquée par l'indice *t* (de l'anglais « *to* »). On réfère à une branche par les indices des bus qu'elle relie. Par exemple, la branche ft relie le bus f au bus t. L'effet résistif et inductif de chaque branche est identifié par y_s. L'effet capacitif est modélisé par des admittances imaginaires égalent à $jb_c/2$ aux deux extrémités de la ligne. Le transformateur a un rapport de transformation complexe N d'une magnitude τ et d'un angle de déphasage θ_{shift}.

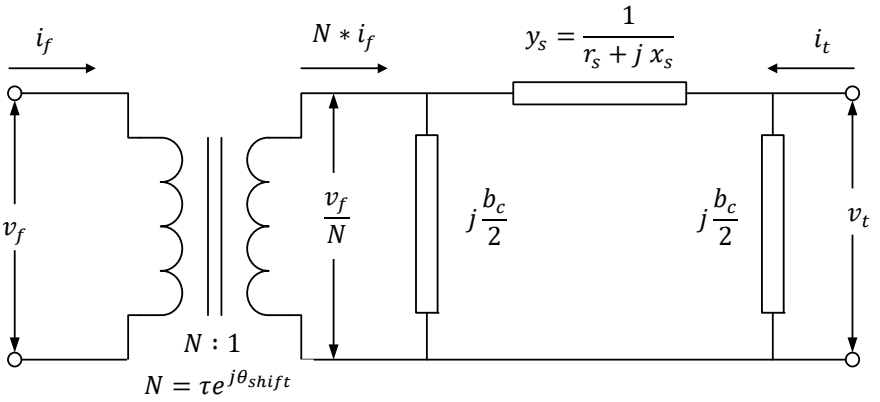


Figure 7.2: Modèle de branche en π

D'après ce modèle, les courants complexes i_f et i_t injectés aux extrémités *from* et *to* de la branche ft peuvent être calculés comme suit à l'aide d'une matrice d'admittance de branche $\boldsymbol{Y_{ft}}$ et des tensions complexes v_f et v_t :

$$\boldsymbol{Y_{ft}} * \begin{bmatrix} v_f \\ v_t \end{bmatrix} = \begin{bmatrix} i_f \\ i_t \end{bmatrix} \tag{7.1}$$

où la matrice d'admittance $\boldsymbol{Y_{ft}}$ de la branche ft est égale à :

$$\boldsymbol{Y_{ft}} = \begin{bmatrix} \left(y_s + j\frac{b_c}{2}\right)\frac{1}{\tau^2} & -y_s\frac{1}{\tau e^{-j\theta_{shift}}} \\ -y_s\frac{1}{\tau e^{j\theta_{shift}}} & y_s + j\frac{b_c}{2} \end{bmatrix} \tag{7.2}$$

ou simplement

$$\boldsymbol{Y_{ft}} = \begin{bmatrix} y_{ff} & y_{ft} \\ y_{tf} & y_{tt} \end{bmatrix} \tag{7.3}$$

Après avoir calculé la matrice à l'équation (7.3) pour chacune des branches du réseau, il est possible d'obtenir la matrice d'admittance $\boldsymbol{Y_{bus}}$ pour le réseau entier. Celle-ci est construite à partir d'une matrice nulle en insérant les éléments y_{ff}, y_{ft}, y_{tf} et y_{tt} de l'équation (7.3) à la location identifiée par leur indice. Lorsque plusieurs éléments sont destinés au même endroit, il faut les additionner et insérer leur somme. Finalement, les admittances de shunt des bus doivent être ajoutées aux éléments de la diagonale. On obtient alors la matrice $\boldsymbol{Y_{bus}}$ de dimension $n_{bus}\ x\ n_{bus}$ pour un réseau

où le nombre de bus est égal à n_{bus}. La relation entre les courants $\boldsymbol{I}$ injectés aux bus et les tensions $\boldsymbol{V}$ de ces derniers est définie comme suit :

$$\boldsymbol{I} = \boldsymbol{Y_{bus}} * \boldsymbol{V} \tag{7.4}$$

où $\boldsymbol{I}$ et $\boldsymbol{V}$ sont des vecteurs verticaux complexes de dimension n. Les éléments y_{ij} de la matrice d'admittance $\boldsymbol{Y_{bus}}$ sont aussi complexes et définis comme suit :

$$y_{ij} = g_{ij} + j\, b_{ij} \tag{7.5}$$

où la partie réelle g_{ij} est la conductance tandis que la partie imaginaire b_{ij} est la suceptance. La puissance complexe S_i injectée au bus i se calcule suivant :

$$S_i = V_i \sum_{k=1}^{n_{bus}} y_{ij}^* V_k^* = P_i + j\, Q_i \tag{7.6}$$

où le symbole « * » dénote l'opération du conjugué complexe. Finalement la puissance complexe injectée au bus i peut aussi être divisée en la puissance active P_i et la puissance réactive Q_i :

$$P_i = \sum_{k=1}^{n_{bus}} |V_i|\,|V_k|\,(g_{ik}\cos(\delta_i - \delta_k) + b_{ik}\sin(\delta_i - \delta_k)) \tag{7.7}$$

$$Q_i = \sum_{k=1}^{n_{bus}} |V_i|\,|V_k|\,(g_{ik}\sin(\delta_i - \delta_k) - b_{ik}\cos(\delta_i - \delta_k)) \tag{7.8}$$

où δ_i est l'angle du phaseur de tension au bus i. Aux équations (7.7) et (7.8), les valeurs de g_{ik} et de b_{ik} sont connues pour tous les bus, les valeurs de P_i et Q_i sont connues uniquement pour les bus connectés à des charges tandis que les valeurs de $|V_i|$ et δ_i sont connues uniquement pour les bus connectés à des générateurs. Par conséquent, deux variables sont inconnues pour chaque bus du réseau. Le problème d'analyse de PF consiste à utiliser les équations (7.7) et (7.8) pour calculer ces deux inconnus afin d'obtenir l'état complet du réseau en régime permanent. Ceci requiert la résolution d'un système de $2n_{bus}$ équations non linéaires afin de trouver $2n_{bus}$ inconnus, où n_{bus} est le nombre de bus dans le réseau.

7.2.2 Méthodes pour l'analyse de l'écoulement de puissance

Deux techniques communes pour l'analyse de PF d'un réseau de transport d'électricité sont la méthode de G-S et la méthode de N-R. Ces deux algorithmes se basent sur un processus itératif pour calculer numériquement une solution au système d'équations non linéaires. La méthode de N-R est sans aucun doute supérieure à celle

de G-S car elle nécessite beaucoup moins d'itérations pour converger. Toutefois, la méthode de G-S est plus facile à paralléliser et offrira une meilleure accélération. Dans cette thèse, nous proposons des implémentations parallèles des deux algorithmes afin de vérifier que la méthode de N-R reste l'approche préférable même après la parallélisation sur GPU. Dans ces deux méthodes, les bus du réseau sont divisés en trois catégories :

1. **Les bus PQ** sont les bus connectés aux charges. Les valeurs de leur puissance active P et de leur puissance réactive Q sont connues.

2. **Les bus PV** sont les bus connectés aux générateurs. Les valeurs de leur puissance réelle P et de l'amplitude de leur tension |V| sont connues.

3. **Le bus de référence** est un bus connecté à un générateur avec une grande capacité de production. Il est utilisé de façon à éviter que le système d'équation à résoudre soit surdéterminé. Ses valeurs de P et de |V| sont calculées pendant l'analyse de PF de façon à équilibrer la puissance active et réactive dans le réseau.

7.2.2.1 Méthode de Gauss-Seidel

L'algorithme de Gauss-Seidel débute avec des valeurs initiales pour la tension complexe des bus. Celles-ci peuvent être nominales ou basées sur des estimations. À chaque itération, tous les bus PQ et PV sont visités afin d'ajuster leur phaseur de tension V_i en utilisant l'équation suivante :

$$V_i^{(k+1)} = \frac{1}{y_{ii}}\left(\frac{P_i - jQ_i}{V_i^{*(k)}} - \sum_{j=1}^{i-1} y_{ij}\, V_j^{(k+1)} - \sum_{j=i+1}^{n_{bus}} y_{ij}\, V_j^{(k)}\right) \tag{7.9}$$

où i est l'indice du bus et k est le numéro de l'itération actuelle. Dans le cas des bus PQ, les variables du côté droit de l'équation (7.9) sont toutes connues et la tension $V_i^{(k+1)}$ peut facilement être calculée. Dans le cas des bus PV, la variable Q_i est inconnue et doit être évaluée comme suit avant de pouvoir calculer $V_i^{(k+1)}$:

$$Q_i^{(k+1)} = -Im\left\{V_i^{(k)} * \left(\sum_{j=1}^{i-1} y_{ij}\, V_j^{(k+1)} - \sum_{j=i}^{n_{bus}} y_{ij}\, V_j^{(k)}\right)\right\} \tag{7.10}$$

Encore dans le cas des bus PV, le phaseur de tension $V_i^{(k+1)}$ calculé doit aussi être mis à l'échelle afin de maintenir l'amplitude $|V_i|$ spécifiée par le réglage du générateur. Aux équations (7.9) et (7.10), on peut noter que la sommation est divisée en deux parties de façon à toujours utiliser la valeur de V_j la plus récente. Ceci est une caractéristique importante qui contribue à accélérer la convergence de l'algorithme.

À la fin de chaque itération, la puissance active P_i et la puissante réactive Q_i sont calculées pour chacun des bus en utilisant les équations (7.7) et (7.8). Les valeurs obtenues sont ensuite comparées aux valeurs actuelles de P et Q dans le cas des bus PQ ou simplement de P dans le cas des bus PV. Si l'erreur maximale est plus petite qu'une certaine tolérance, l'algorithme de G-S a convergé et le processus itératif se termine. Les valeurs de P et de Q au bus de référence sont alors calculées d'après les équations (7.7) et (7.8) et l'analyse de PF est finalement complétée.

7.2.2.2 Méthode de Newton-Raphson

Tout comme la méthode précédente, l'algorithme de Newton-Raphson débute aussi avec des estimations initiales pour la tension complexe des bus et met à jour ces valeurs à l'aide d'un processus itératif. Les équations non linéaires pour la puissance active et réactive sont exprimées par des séries de Taylor et seule la partie linéaire est gardée afin d'obtenir un système d'équations linéaires de la forme $\boldsymbol{A} * \boldsymbol{x} = \boldsymbol{b}$, où $\boldsymbol{A}$ est la matrice Jacobienne, $\boldsymbol{x}$ est le vecteur de corrections, et $\boldsymbol{b}$ est le vecteur des différences entre les puissances actuelles injectées aux bus et les valeurs calculées à l'aide des équations (7.7) et (7.8). Le système d'équations est solutionné pour $\boldsymbol{x}$ et les corrections obtenues sont appliquées aux valeurs estimées des tensions complexes aux bus avant de passer à l'itération suivante. Ce processus est répété jusqu'à ce que la plus grande valeur dans le vecteur $\boldsymbol{b}$ soit plus petite que la tolérance spécifiée. Étant donné que la tension complexe des bus peut être exprimée à l'aide de coordonnées polaires ou rectangulaires, la méthode de N-R a deux représentations. Dans cette thèse, nous utilisons celle basée sur les coordonnées polaires puisque le système linéaire qui en résulte contient moins d'équations que pour la représentation avec coordonnées rectangulaires. Une description détaillée des algorithmes de G-S et de N-R est donnée à la référence [250].

7.2.3 Parallélisation

7.2.3.1 Structure logicielle et représentation des données

L'architecture du programme parallèle que nous proposons dans cette thèse pour l'analyse de PF sur GPU est illustrée au diagramme UML de classes à la Figure 7.3. Le programme est divisé en plusieurs classes C++. Les objets logiciels résident dans la mémoire du CPU et contiennent des pointeurs vers les tableaux de données placés dans la mémoire globale du GPU. Au niveau supérieur, le flux d'exécution de l'algorithme de PF est contrôlé par les méthodes des classes qui exécutent sur le CPU. Au niveau inférieur, ces méthodes appellent des *kernels* CUDA™ et les opérations parallèles sont effectuées directement sur les données du GPU.

1) Description du réseau

Lorsque le programme est lancé, la description du réseau est lue d'un fichier de données et stockée dans un objet *PFCase*. Notre programme est compatible avec

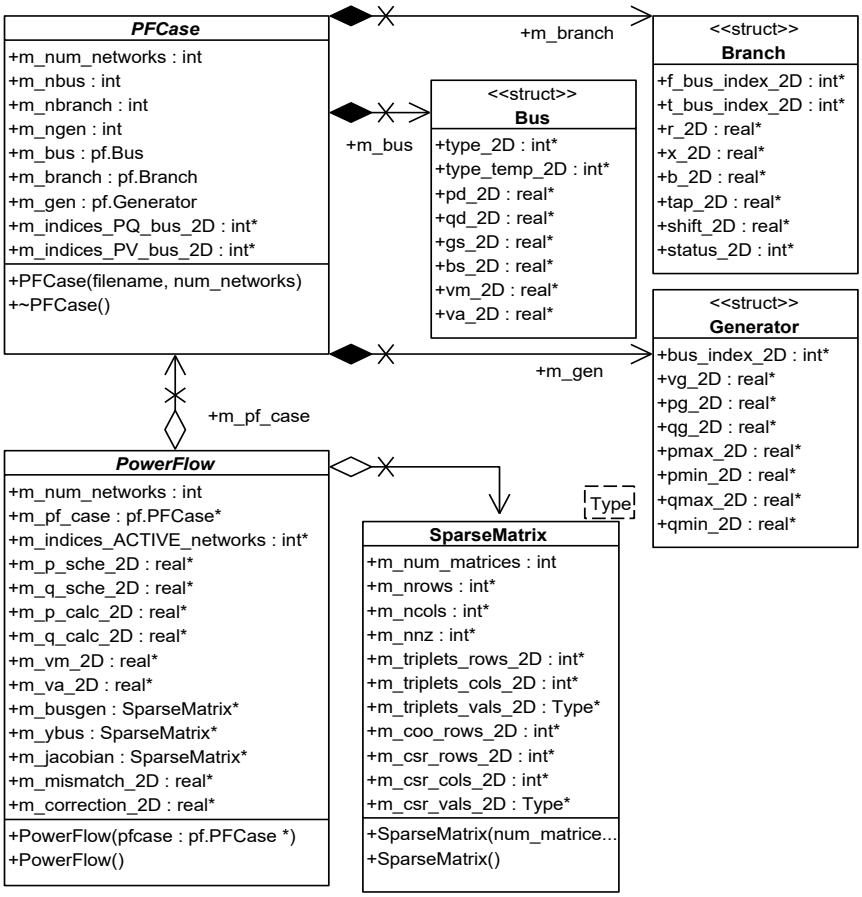


Figure 7.3: Diagramme UML de classes du programme d'analyse de l'écoulement de puissance

l'outil MATPOWER [136] et supporte le format de fichier d'entrée MPC v2. Au moment du chargement, la description du réseau est divisée en structures contenant les données des bus, les données des branches et les données des générateurs. Ces structures sont gardées dans la mémoire du CPU et contiennent uniquement des pointeurs vers les données qui sont quant à elles sauvegardées sur le GPU. La classe *PFCase* maintient aussi des tableaux d'indices pour chaque type de bus afin de permettre un accès direct. Par exemple, le tableau *m_indices_PQ_bus_2D* contient les indices de tous les bus PQ. Ces tableaux d'indices sont construits en utilisant une opération parallèle de compaction sur le tableau contenant le type de chaque bus. De plus, il est important de noter que le constructeur de la classe *PFCase* a un paramètre pour identifier le nombre de réseaux. Lorsque le fichier d'entrée est chargé, la description du réseau est en fait copiée plusieurs fois dans les structures de données de l'objet *PFCase*, une fois pour chaque instance du réseau. C'est pourquoi les pointeurs dans les structures *Bus*, *Branch* et *Generator* ont tous le suffixe « *_2D* » afin

d'identifier qu'ils pointent vers des tableaux à deux dimensions où chaque ligne i contient les données du i^e réseau. Une fois copié dans la mémoire globale du GPU, les données de chaque instance de réseau peuvent être accédées et modifiées afin de créer des scénarios contingents qui seront testés par l'analyse de PF. Par exemple, il est possible de modifier la puissance et la tension des générateurs, le rapport et l'angle de déphasage des transformateurs et la susceptance des compensateurs statiques d'énergie réactive. Comme nous verrons à la section suivante, ces paramètres sont modifiés par l'algorithme de PSO afin d'évaluer la qualité des solutions candidates. Pendant l'exécution de l'analyse de PF, les threads CUDA™ sont divisés en blocs de threads et chaque bloc traite un réseau unique. Cette approche permet d'exploiter un très grand niveau de parallélisme afin de bénéficier pleinement de la puissance de calcul du GPU.

2) Analyse de l'écoulement de puissance

Une fois que les données du réseau sont chargées dans l'objet *PFCase*, l'analyse de l'écoulement de puissance peut être exécutée à l'aide de l'objet *PowerFlow*. Les attributs *m_p_sched_2D* et *m_q_sched_2D* pointent vers les valeurs de références pour les puissances actives et réactives aux bus. Quant à eux, les attributs *m_p_calc_2D* et *m_q_calc_2D* pointent vers les valeurs calculées et sont utilisés par les algorithmes de G-S et de N-R pour mettre à jour *m_vm_2D* et *m_va_2D*, soit l'amplitude et l'angle des phaseurs de tension aux bus. Tout comme pour la classe *PFCase*, les tableaux de la classe *PowerFlow* sont à deux dimensions. L'attribut *m_busgen* est une matrice de connexion creuse qui permet de relier chaque bus à ses générateurs. Les attributs *m_ybus* et *m_jacobian* sont respectivement la matrice d'admittance du réseau et la matrice Jacobienne spécifique à la méthode de N-R. Ces matrices utilisent elles aussi une représentation creuse et incluent uniquement les éléments non nuls.

3) Matrices creuses

Toutes les matrices dans le logiciel d'analyse de PF proposé ont une représentation creuse. Contrairement à la représentation pleine, la représentation creuse inclut uniquement les éléments non nuls et permet de diminuer l'ordre de complexité de l'analyse de PF de $O(n^2)$ à $O(n)$. Une représentation pleine ainsi que trois formats différents pour une représentation creuse sont illustrés à la Figure 7.4. Étant donnée la matrice $3x3$ utilisée dans l'exemple, la représentation pleine consiste à lister séquentiellement tous les éléments de la matrice. Cette énumération peut se faire suivant l'ordre des rangées comme dans l'exemple ou celui des colonnes. Pour une matrice de dimension NxM, le vecteur résultant a une longueur de $N * M$ et peut contenir plusieurs éléments nuls. Dans le cas d'une analyse de PF, la matrice d'admittance et la matrice Jacobienne ont une faible densité et contiennent un très grand nombre d'éléments nuls. Il est alors avantageux d'utiliser une représentation creuse. Le premier format creux à la Figure 7.4 consiste à lister chacun des éléments

non nuls sous forme d'un triplet contenant l'indice de la ligne, l'indice de la colonne et la valeur de l'élément. Les triplets ne sont dans aucun ordre spécifique, peuvent pointer vers la même location ou peuvent avoir une valeur de zéro. Lorsque plusieurs triplets pointent vers la même location, il est important de définir leur relation. Dans l'exemple, la valeur de l'élément en question est définie comme étant la somme des valeurs des triplets. En réalité, la représentation par triplets n'est pas un format standard, mais uniquement une étape transitoire vers une autre représentation comme la représentation par coordonnées (COO). Ce format est semblable au précédent, mais n'inclut aucun duplicata et nécessite que les éléments soient entièrement ordonnés. Le format COO permet un accès direct à chaque élément de la matrice. Finalement, dans le format par rangées compressées (CSR de l'anglais *compressed sparse row*), le vecteur *CSR_rangées* contient l'indice du début de chaque rangée dans les vecteurs *CSR_colonnes* et *CSR_valeurs*. Pour une matrice avec *N* rangées, le vecteur *rangées* contient *N + 1* éléments et la longueur de la rangée *i* est calculée par *CSR_rangées[i+1] - CSR_rangées[i]*. Le format CSR permet un accès direct à chaque rangée de la matrice.

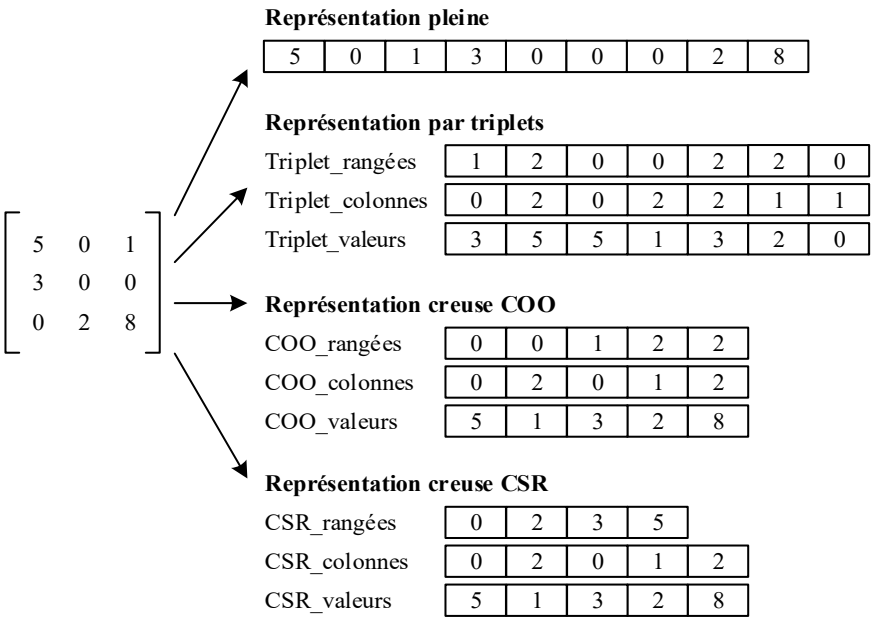


Figure 7.4: Différentes représentations d'une matrice

Dans notre implémentation, nous utilisons le format par triplets comme point de départ pour obtenir le format COO. La transformation est effectuée en parallèle sur le GPU et inclut un tri bitonique pour ordonnancer les triplets et une réduction segmentée pour additionner leur valeur lorsque plusieurs triplets pointent vers une même

location. À son tour, le format COO est utilisé pour obtenir le format CSR. Pour ce faire, seul le vecteur *COO_rangées* doit être modifié. Le processus est illustré à la Figure 7.5 pour la matrice utilisée précédemment à la Figure 7.4. La première étape consiste à utiliser une primitive parallèle de réduction segmentée pour calculer la longueur des segments du vecteur *COO_rangées*. Cette opération calcule en fait le nombre d'éléments dans chaque rangée de la matrice. Par la suite, il faut initialiser un vecteur nul de $N + 1$ éléments, où N est égale au nombre de rangées dans la matrice, Les longueurs calculées à l'étape 1 sont alors copiées à l'étape 2 dans ce vecteur nul à l'aide d'une primitive de dispersion parallèle en utilisant le numéro de la rangée comme indice de destination. Finalement, un scan parallèle permettant de calculer la somme cumulative exclusive est utilisé à l'étape 3 pour obtenir le vecteur *CSR_rangées*. Ce vecteur contient les indices du début de chaque rangée dans les vecteurs *CSR_colonnes* et *CSR_valeurs* de la représentation CSR à la Figure 7.4.

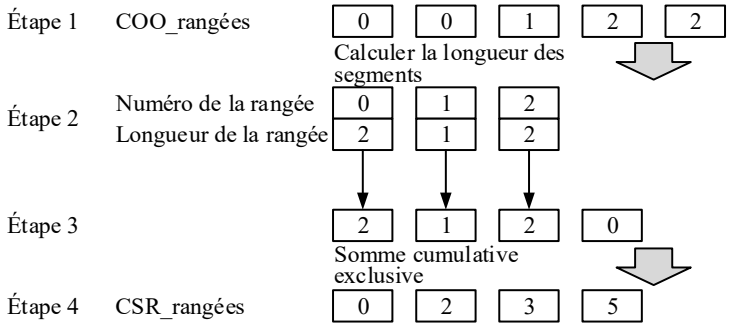


Figure 7.5: Conversion du vecteur *COO_rangées* en vecteur *CSR_rangées*

Notre programme d'analyse de PF utilise principalement le format CSR pour représenter les différentes matrices. Ce format permet une parallélisation efficace en associant un thread CUDA™ à chaque rangée de la matrice. Par exemple, lors du calcul des puissances actives et réactives injectées aux bus, chaque thread calcule les puissances pour un bus spécifique à l'aide des équations (7.7) et (7.8) en parcourant une rangée de la matrice d'admittance. Toutefois, lors de la construction de la matrice Jacobienne dans la méthode de N-R, nous utilisons aussi le format COO pour la matrice d'admittance. Le format COO nous permet de lancer un thread par élément de la matrice d'admittance. Le niveau de parallélisme exploité est alors plus grand et la divergence d'exécution causée par des longueurs de rangées différentes est évitée résultant en une exécution plus rapide.

7.2.3.2 *Implémentation parallèle de Gauss-Seidel*

Dans cette section, nous expliquons les détails de notre implémentation parallèle de l'algorithme de Gauss-Seidel sur GPU. Nous listons au Tableau 7.1 le pseudocode de l'algorithme ainsi que les temps d'exécution pour chacune des fonctions. Ces temps sont pour l'analyse de 500 instances du réseau test IEEE à 300 bus et incluent l'exécution séquentielle sur CPU et l'exécution parallèle sur GPU. Les données pour le réseau de transport d'électricité utilisé sont disponibles à la référence [136]. Le code est développé à l'aide de Visual Studio 2013, sous Windows 8.1 x64 Pro en utilisant CUDA™ SDK 6.5. Les temps listés sont les moyennes de 100 essais indépendants et sont mesurés sur un système Dell Précision T7600 équipé de deux processeurs Intel Xeon E5-2650 et d'un processeur graphique NVIDIA® Tesla K20c. La version CPU est programmée en C++ et utilise un thread pour analyser séquentiellement les 500 instances du réseau. La version GPU est programmée en CUDA™ C++ et maximise le nombre de threads pour chacune des fonctions afin d'exploiter le plus possible les 2496 cœurs du processeur graphique. Par exemple, lors du calcul de la puissance complexe à la ligne 7 du Tableau 7.1, le programme CUDA™ lance un thread par bus. Comme nous résolvons 500 réseaux simultanément et que chaque réseau contient 300 bus, le programme CUDA™ lance en fait 150 000 threads. Les paragraphes suivants expliquent l'implémentation pour chacune des fonctions listées au Tableau 7.1. À l'exception de celle à la ligne 9, toutes les fonctions sont exécutées en mode batch et utilisent un bloc de threads CUDA™ par réseau.

TABLEAU 7.1
PSEUDOCODE POUR L'ALGORITHME DE GAUSS-SEIDEL AVEC TEMPS D'EXÉCUTION ET ACCÉLÉRATIONS MOYENS (500 RÉSEAUX, IEEE 300-BUS, 100 ESSAIS)

	Pseudocode	Temps d'exécution (ms)		Accélération
		CPU	GPU	
1:	Construire la matrice de connexion *bus-générateurs*	1.069	0.205	5.2 x
2:	Initialiser les variables et le vecteur des réseaux actifs	6.329	0.323	19.6 x
3:	Construire la matrice d'admittance	154.716	2.798	55.3 x
4:	**Tant que (***Itération* **<** $N_{itération\ max}$**) et (**$N_{réseaux\ actifs}$ **> 0) {**	-	-	-
5:	Mettre à jour *V* pour tous les bus PQ	17.329	0.289	60.0 x
6:	Mettre à jour *V* pour tous les bus PV	6.654	0.205	32.5 x
7:	Calculer *S* pour tous les bus	13.083	0.234	55.9 x
8:	Trouver l'erreur maximale pour *S*	0.391	0.061	6.4 x
9:	Mettre à jour le vecteur d'indices des réseaux actifs	0.003	0.047	0.1 x
10:	*Itération* = *Itération* + 1	-	-	-
11:	**}**	-	-	-
12:	Sauvegarder *V* pour tous les bus	9.419	0.094	100.2 x
13:	Calculer *P* et *Q* pour tous les bus	27.234	0.659	41.3 x
14:	Sauvegarder *P* et *Q* pour les générateurs au bus de référence	0.060	0.035	1.7 x
15:	Sauvegarder *Q* et *V* pour tous les autres générateurs	0.695	0.061	11.4 x
16:	Calculer *S* et les pertes pour chaque ligne de transport	68.793	0.964	71.4 x
17:	**Retour**	-	-	-
	TOTAL	**305.775**	**5.975**	**51.2x**

À la ligne 1, la matrice de connexion *bus-générateurs* est formée afin de relier chaque bus à ses générateurs. Pour ce faire, n_{gen} threads par réseau sont lancés et chaque thread traite un générateur de la structure *m_gen* identifiée à la Figure 7.3 et produit un triplet de la forme $\{bus_{idx}, gen_{idx}, status\}$. Les triplets sont ensuite ordonnés à l'aide d'un tri bitonique. Les générateurs avec un état hors-ligne sont enlevés en utilisant une primitive de compaction parallèle afin d'obtenir une matrice creuse conforme au format COO. Cette matrice est finalement convertie en format CSR en suivant la procédure que nous venons tout juste de décrire à la section précédente.

À la ligne 2, les différentes variables nécessaires à l'analyse de PF sont initialisées. Le calcul de la puissance injectée aux bus se fait en utilisant un thread par bus. Chaque thread traite une ligne de la matrice *bus-générateurs* et additionne les puissances des multiples générateurs connectés au bus. Cette approche est possible grâce au format CSR de la matrice de connexion. Finalement, la charge au bus est soustraite de la somme afin d'obtenir la puissance injectée.

La matrice d'admittance $\boldsymbol{Y}_{bus}$ est construire à la ligne 3. Pour ce faire, un thread par branche de la structure *m_branch* est lancé. Chaque thread calcule la matrice d'admittance pour sa propre branche d'après l'équation (7.3) définie précédemment. La matrice d'admittance $\boldsymbol{Y}_{bus}$ pour le réseau en entier est ensuite formée à partir d'une matrice nulle de dimension $n_{bus} * n_{bus}$. Pour chaque branche, les quatre termes du côté droit de l'équation (7.3) sont insérés dans la matrice nulle aux emplacements identifiés par leurs indices. Lorsque plusieurs termes sont destinés au même endroit, leurs valeurs sont additionnées. Enfin, les admittances de shunt des bus sont lues de la structure *m_bus* et ajoutées aux éléments diagonaux de la matrice. Le résultat final est un tableau non ordonné de $4 * n_{branch} + n_{bus}$ triplets incluant quelques éléments nuls. L'approche parallèle discutée à la section 7.2.3.1 est ensuite utilisée en prenant soin d'enlever les triplets nuls par une primitive parallèle de compaction pour obtenir la matrice d'admittance $\boldsymbol{Y}_{bus}$ pour le réseau en entier dans les formats COO et CSR.

Aux lignes 5 et 6, un thread par bus PQ et un thread par bus PV sont utilisés pour parcourir les lignes associées de la matrice d'admittance en format CSR et mettre à jours les estimées des tensions complexes aux bus. Dans le cas des bus PV, si la puissance réactive calculée à l'équation (7.10) excède la limite permise par le générateur, cette puissance est fixée à la limite et l'estimation de tension complexe est mise à jour suivant l'équation (7.9) en considérant le bus comme étant de type PQ pour l'itération courante seulement. Ceci permet d'ajuster la tension spécifiée au bus PV afin de respecter la limite physique sur la puissance réactive des générateurs. Lors du calcul des tensions complexes au bus, les tableaux d'indices *m_indices_PQ_bus_2D* et *m_indices_PV_bus_2D* sont utilisés pour mapper directement les threads aux bons bus. L'utilisation d'une matrice d'admittance creuse est essentielle au calcul rapide des tensions puisqu'elle réduit la complexité de

l'opération à $O(NNZ)$ comparativement à $O(n_{bus}^2)$ si une matrice pleine avait été utilisée, où NNZ représente le nombre d'éléments non nuls de la matrice creuse.

Après avoir mis à jour les valeurs des tensions, la puissance complexe injectée aux bus est calculée à la ligne 7. Cette opération se fait de la même façon que celle précédente, mais ne différencie pas les bus PQ des bus PV.

À la ligne 8, n_{bus} threads sont utilisés pour évaluer les différences entre les valeurs calculées et prévues pour les puissances complexes injectées aux bus. L'erreur maximale pour le réseau en entier est ensuite obtenue par une primitive de réduction parallèle.

Lorsque cette erreur maximale est plus petite que la tolérance spécifiée, l'analyse de l'écoulement de puissance a convergé. Étant donné que chaque réseau analysé peut prendre un nombre différent d'itérations avant d'arriver à la tolérance désirée, un vecteur contenant les indices des réseaux actifs (ceux qui n'ont pas encore convergé) est maintenu et seuls les réseaux actifs sont traités aux lignes 5, 6 et 7. Cette approche évite de gaspiller inutilement la puissance de calcul du GPU sur les réseaux qui ont déjà convergé. Le vecteur des indices des réseaux actifs contient initialement tous les réseaux et est mis à jour à la ligne 9 en utilisant une primitive de compaction parallèle. La mauvaise accélération mesurée à la ligne 9 est causée par le transfert du nombre de réseaux actifs du GPU au CPU par l'entremise du bus PCIe avant de pouvoir évaluer la condition à la ligne 4. Toutefois, lorsque nous exécutons l'algorithme en entier, nous exploitons la fonction de parallélisme dynamique des processeurs NVIDIA® pour implémenter la boucle *while* à l'intérieur d'un seul *kernel* CUDA™. La condition à la ligne 4 est alors vérifiée directement sur le GPU et aucun transfert de données entre le GPU et le CPU n'est nécessaire, assurant ainsi une meilleure performance.

Lorsque tous les réseaux ont convergé ou lorsque le nombre maximal d'itérations a été atteint, le processus itératif termine. Les tensions et les puissances injectées aux bus sont sauvegardées dans l'objet *PFCase* aux lignes 12 à 15. Les puissances écoulées dans les banches ainsi que les pertes des branches sont calculées à la ligne 16 en utilisant un thread par branche et sont, elles aussi, sauvegardées dans l'objet *PFCase*. L'analyse de l'écoulement de puissance suivant la méthode de Gauss-Seidel est alors terminée.

7.2.3.3 Implémentation parallèle de Newton-Raphson

Dans cette section, nous présentons les détails de notre implémentation parallèle de l'algorithme de Newton-Raphson sur GPU. Nous listons au Tableau 7.2 le pseudocode de l'algorithme ainsi que les temps d'exécution pour chacune des fonctions. Tout comme pour l'exemple précédent, ces temps sont pour l'analyse concurrente de 500 instances du réseau test IEEE à 300 bus et incluent l'exécution

séquentielle sur CPU et l'exécution parallèle sur GPU. L'ordinateur utilisé est le même qu'à la section précédente, soit un Dell T7600 équipé de deux processeurs Intel Xeon E5-2650 et d'un processeur graphique NVIDIA® Tesla K20c. Tous les calculs sont effectués avec une précision double. Au Tableau 7.2, on note que certaines des fonctions listées ont été utilisées précédemment dans l'algorithme de G-S. De façon à rester brefs, nous omettons de les décrire à nouveau. Finalement, à l'exception de celle à la ligne 15, toutes les fonctions exécutent en mode batch et utilisent un bloc de threads CUDA™ par réseau.

Notre implémentation de l'algorithme de N-R débute à la ligne 1 avec la construction de la matrice de connexion qui relie les bus à leurs générateurs. Les variables nécessaires à l'algorithme ainsi que la matrice d'admittance sont initialisées aux lignes 2 et 3. D'après les estimations initiales pour les tensions complexes, les puissances actives P et réactives Q injectées aux bus sont calculées à la ligne 4. Ces opérations sont effectuées de la même façon que dans notre implémentation de l'algorithme de G-S.

TABLEAU 7.2
PSEUDOCODE POUR L'ALGORITHME DE NEWTON-RAPHSON AVEC TEMPS D'EXÉCUTION ET ACCÉLÉRATIONS MOYENS (500 RÉSEAUX, IEEE 300-BUS, 100 ESSAIS)

	Pseudocode	Temps d'exécution (ms)		Accélération
		CPU	GPU	
1:	Construire la matrice de connexion bus-générateurs	1.070	0.210	5.1 x
2:	Initialiser les variables et le vecteur des réseaux actifs	6.348	0.330	19.2 x
3:	Construire la matrice d'admittance	154.285	2.820	54.7 x
4:	Calculer P et Q pour tous les bus	27.503	0.662	41.5 x
5:	Mettre à jour le type et les vecteurs d'indices des bus	1.049	0.103	10.2 x
6:	Construire le vecteur des différences	0.777	0.073	10.6 x
7:	**Tant que (***Itération* **<** $Itération_{max}$**) et (**$N_{réseaux\ actifs}$ > 0**) {**	-	-	-
8:	Construire la matrice Jacobienne	61.308	2.925	21.0 x
9:	Résoudre le système d'équations linéaires	972.043	55.104	17.6 x
10:	Mettre à jour $\|V\|$ et δ	2.451	0.075	32.7 x
11:	Calculer P et Q pour tous les bus	27.350	0.658	41.6 x
12:	Mettre à jour le type et les vecteurs d'indices des bus	1.092	0.101	10.8 x
13:	Construire le vecteur des différences	0.777	0.072	10.8 x
14:	Trouver l'erreur maximale pour S	0.362	0.039	9.3 x
15:	Mettre à jour le vecteur d'indices des réseaux actifs	0.000	0.002	0.0 x
16:	*Itération* = *Itération* + 1	-	-	-
17:	}	-	-	-
18:	Sauvegarder V_m et V_a pour tous les bus	0.534	0.052	10.3 x
19:	Sauvegarder P et Q pour les générateurs au bus de référence	0.061	0.045	1.4 x
20:	Sauvegarder Q et V_m pour tous les autres générateurs	0.702	0.068	10.3 x
21:	Calculer S et les pertes pour chaque ligne de transport	69.094	0.971	71.2 x
22:	**Retour**	-	-	-
	TOTAL	**1326.806**	**64.310**	**20.6 x**

Notre implémentation de l'algorithme de N-R débute à la ligne 1 avec la construction de la matrice de connexion qui relie les bus à leurs générateurs. Les variables nécessaires à l'algorithme ainsi que la matrice d'admittance sont initialisées aux lignes 2 et 3. D'après les estimations initiales pour les tensions complexes, les puissances actives *P* et réactives *Q* injectées aux bus sont calculées à la ligne 4. Ces opérations sont effectuées de la même façon que dans notre implémentation de l'algorithme de G-S.

De façon à respecter les limites sur les puissances réactives des générateurs, les valeurs de *Q* calculées à la ligne 4 pour les bus PV sont vérifiées à chaque itération de l'algorithme. Si une valeur dépasse la limite permise, le type du bus est changé temporairement de PV à PQ et la puissance réactive injectée est ramenée à la limite. Ce changement de type est sauvegardé dans le tableau *PFCase.bus.type_temp_2D* et est valide uniquement pour l'itération courante. Cette opération utilise un thread par bus PV et est effectuée au début de l'algorithme, à la ligne 5, et à chaque itération, à la ligne 12. Ces valeurs temporaires sont ensuite utilisées pour mettre à jour les tableaux d'indices *m_indices_PQ_bus_2D* et *m_indices_PV_bus_2D* à l'aide d'une primitive de compaction parallèle. Modifier le type d'un bus entraîne une certaine complexité. En effet, le nombre de bus PQ et PV peut varier d'une itération à l'autre ce qui affecte la taille de la matrice Jacobienne et le nombre d'équations linéaires à résoudre. Par contre, traiter un bus PV comme étant de type PQ permet d'ajuster la tension des générateurs afin de ramener la puissance réactive requise à sa limite permise [250].

À la ligne 6, le vecteur des différences entre les valeurs actuelles des puissances injectées aux bus et leurs valeurs calculées est construit à l'aide d'un thread par bus. Deux éléments sont générés pour chaque bus PQ tandis qu'un seul est produit par bus PV. Ce vecteur contient alors $2 * n_{PQ} + n_{PV}$ éléments, où n_{PQ} et n_{PV} sont respectivement le nombre de bus PQ et PV dans le réseau.

Le processus itératif de l'algorithme de N-R est initié à ligne 7. Chaque itération débute par la construction de la matrice Jacobienne à la ligne 8. La taille de celle-ci varie d'une itération à l'autre selon le nombre de bus PQ et PV. La matrice Jacobienne $\boldsymbol{J}$ est formée de quatre sous-matrices comme suit :

$$\boldsymbol{J} = \begin{bmatrix} \boldsymbol{J}^{11}_{n_{PQPV} * n_{PQPV}} & \boldsymbol{J}^{12}_{n_{PQPV} * n_{PQ}} \\ \boldsymbol{J}^{21}_{n_{PQ} * n_{PQPV}} & \boldsymbol{J}^{22}_{n_{PQ} * n_{PQ}} \end{bmatrix} \tag{7.11}$$

où les caractères surélevés identifient l'indice des sous-matrices et ceux surbaissés représentent leur taille. Le terme n_{PQPV} est la somme du nombre de bus PQ et PV. La sous-matrice $\boldsymbol{J^{11}}$ comprend une rangée et une colonne pour chaque bus PQ et PV. Par contre, $\boldsymbol{J^{12}}$ n'inclut pas de colonnes pour les bus PV, $\boldsymbol{J^{21}}$ n'inclut pas de rangées pour

les bus PV et $\boldsymbol{J^{22}}$ n'inclus pas de colonnes ni de rangées pour les bus PV. Pour expliquer notre implémentation, il est suffisant de limiter notre analyse à la sous-matrice $\boldsymbol{J^{11}}$. Les détails complets quant à la construction des trois autres sous-matrices sont donnés dans la référence [250]. Dans $\boldsymbol{J^{11}}$, les éléments non diagonaux $J_{i,j}^{11}$ et les éléments diagonaux $J_{i,i}^{11}$ sont calculés comme suit :

$$J_{i,j}^{11} = -\left|V_i V_j y_{ij}\right| \sin(\theta_{ij} + \delta_j - \delta_i) \tag{7.12}$$

$$J_{i,i}^{11} = -Q_i - |V_i|^2 \, b_i \tag{7.13}$$

D'après la première équation, on note que l'élément non diagonal $J_{i,j}^{11}$ est non nul seulement si y_{ij} de la matrice d'admittance est aussi non nul. Similairement, dans la deuxième équation, on note que l'élément diagonal $J_{i,i}^{11}$ est non nul seulement si b_i est aussi non nul où b_i est la partie imaginaire de l'élément diagonal y_{ii} de la matrice d'admittance. D'après ces deux observations, il est possible d'utiliser une matrice d'admittance creuse pour construire directement une matrice Jacobienne creuse. Il faut simplement itérer à travers les éléments non nuls y_{ij} de la matrice d'admittance et utiliser l'équation (7.12) si $i \neq j$ ou l'équation (7.13) si $i = j$ pour calculer l'élément $J_{i,j}^{11}$ de la sous-matrice $\boldsymbol{J^{11}}$. La construction des sous-matrices $\boldsymbol{J^{12}}$, $\boldsymbol{J^{21}}$ et $\boldsymbol{J^{22}}$ se fait de façon semblable.

Dans notre implémentation, nous utilisons un thread par élément Y_{ij} de la matrice d'admittance creuse en format COO pour construite la matrice Jacobienne en format triplet. Chaque thread génère jusqu'à quatre triplets ($J_{i,j}^{11}$, $J_{i,j}^{12}$, $J_{i,j}^{21}$ et $J_{i,j}^{22}$) dépendamment du type PQ ou PV des bus i et j. La rangée et la colonne de chaque triplet sont calculées à l'aide de tableaux d'indices qui mappent les bus à leur position dans les tableaux *m_indices_PQ_bus_2D* et *m_indices_PV_bus_2D*. De plus, de façon à écrire les triplets dans l'ordre, leur indice de destination dans le tableau de triplets est évalué à l'aide du vecteur *CSR_rangées* de la matrice d'admittance en format CSR. Cette approche nous évite d'avoir à trier les triplets qui est une opération parallèle très coûteuse. Finalement, étant donné que le nombre de triplets générés n'est pas égal pour tous les threads, nous utilisons une primitive parallèle de compaction ainsi qu'une primitive parallèle de dispersion pour réorganiser les triplets dans un tableau compact afin d'obtenir la matrice Jacobienne en format creux COO. La représentation CSR est ensuite obtenue en suivant la technique discutée précédemment à la section 7.2.3.1.

Une fois que la matrice Jacobienne creuse en format CSR est construite, le système d'équations linéaires est résolu à la ligne 9 afin d'obtenir le vecteur de corrections à appliquer aux valeurs estimées des tensions complexes aux bus. Comme nous l'avons identifié dans la revue de la littérature au Chapitre 2, la résolution d'un

système d'équations linéaires avec matrices creuses est très difficile à implémenter sur un GPU à cause de la structure irrégulière des données [138]. Celle-ci entraîne la divergence d'exécution entre les multiples threads et limite les accès parallèles à la mémoire. Malgré tout, quelques implémentations pour GPU de solveurs supportant les matrices creuses ont été publiées dans [140], [141], [251] et [252]. Il existe aussi quelques bibliothèques logiciels qui supportent cette opération comme CULA [253], cuSPARSE [254] ou PARALUTION [255]. Malheureusement, ces bibliothèques ne peuvent pas être utilisées dans notre application puisqu'elles sont développées dans le but de résoudre un seul, mais très grand, système d'équations linéaires. Quant à lui, l'algorithme d'analyse de PF proposé dans cette thèse vise plutôt à résoudre simultanément un grand nombre de réseaux de petite à moyenne taille. Pour cette raison, au lieu de limiter notre implémentation au GPU, nous optons pour une approche hybride GPU-CPU afin d'exploiter les avantages de ces deux types de processeurs. Nous utilisons Eigen [256], une bibliothèque C++ d'algèbre linéaire, pour résoudre en parallèle les multiples systèmes d'équations linéaires sur processeurs multicœurs grâce à OpenMP®. Chaque système est résolu séquentiellement et le parallélisme est exploité en distribuant les multiples systèmes à plusieurs threads OpenMP®. Sur l'ordinateur Dell T7600 utilisé pour les tests, l'accélération maximale est possible en utilisant 32 threads afin d'exploiter pleinement la technologie *hyperthreading* des 16 cœurs disponibles. Les délais causés par les transferts de données du GPU au CPU (D2H de l'anglais *device to host*) avant la solution du système linéaire et du CPU au GPU (H2D de l'anglais *host to device*) après celle-ci sont masqués par un ordonnancement asynchrone. Tel qu'illustré à la Figure 7.6, les multiples systèmes linéaires ne sont pas transférés d'un seul coup, mais un à la fois ce qui permet au CPU de débuter les calculs dès que le premier transfert est terminé et de chevaucher les transferts subséquents avec les calculs. En utilisant un ordonancement asynchrone, le temps nécessaire pour solutionner les 500 systèmes d'équations linéaires est réduit de 63.2 ms à 55.1 ms, soit une amélioration de 14.7%. L'accélération mesurée pour la solution des systèmes linéaires sur CPU multicœurs incluant le transfert des données est de 17.6x. Comparativement à une implémentation qui se limite au GPU, notre approche parallèle sur système hybride GPU-CPU offre une meilleure utilisation des ressources de calculs présentes dans l'ordinateur et permet un temps de calcul beaucoup plus court. À titre d'exemple, l'implémentation sur GPU avec matrice creuse de l'algorithme de gradient conjugué publié dans [141] par X. Li et F. Li prend 16.3 ms pour résoudre une seule instance du réseau IEEE à 300 bus. Dans notre cas, notre approche sur CPU multicœurs résout 500 instances du même réseau en 55.1 ms, ce qui représente un débit de calcul 147.9x plus grand. L'écart est encore plus lorsqu'on compare notre implémentation à celle de Jalili-Marandi et coll. [126] qui nécessite 136 ms pour résoudre un seul réseau synthétique à 312 bus par une factorisation LU sur GPU avec représentation creuse. Ces deux exemples démontrent clairement l'avantage de notre approche hybride GPU-CPU.

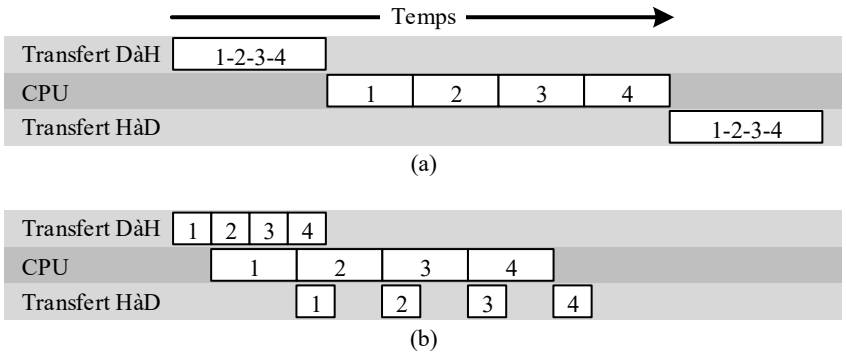


Figure 7.6: Transfert synchrone (a) et asynchrone (b) des systèmes d'équations linéaires

Après la résolution des systèmes d'équations linéaires, les vecteurs de corrections obtenues sont utilisés à la ligne 10 pour corriger en parallèle l'estimation des tensions complexes aux bus à l'aide d'un thread par bus. Le calcul des puissances complexes, la vérification de la limite sur Q, la modification temporaire du type des bus PV, la construction du vecteur des différences ainsi que l'identification de l'erreur maximale sont effectués aux lignes 11 à 16 avant de poursuivre avec l'itération suivante.

Finalement, le processus itératif de l'algorithme de N-R s'arrête lorsque le nombre maximal d'itérations est atteint ou lorsque l'erreur maximale est plus petite que la tolérance spécifiée pour tous les réseaux. Les tensions et les puissances injectées aux bus sont sauvegardées dans l'objet *PFCase* aux lignes 18 à 20. Les puissances écoulées dans les banches ainsi que les pertes des branches sont calculées à la ligne 21 en utilisant un thread par branche et sont, elles aussi, sauvegardées dans l'objet *PFCase*. L'analyse de l'écoulement de puissance suivant la méthode de Newton-Raphson est alors terminée.

7.2.4 Résultats expérimentaux

Dans cette section, nous présentons les résultats des tests expérimentaux effectués pour valider nos implémentations parallèles des algorithmes de G-S et de N-R. Le premier test vérifie l'exactitude des résultats tandis que le second mesure le temps d'exécution et l'accélération obtenue par la parallélisation sur GPU.

7.2.4.1 Exactitude de l'implémentation parallèle

Le premier test consiste à vérifier l'exactitude des algorithmes parallèles proposés. Pour ce faire, nous comparons les résultats obtenus par nos implémentations à ceux de MATPOWER [136], un logiciel de référence pour la simulation des systèmes de puissance. La tolérance est fixée à 1E-8 p.u. pour tous les algorithmes. En raison d'un trop long temps d'exécution, la comparaison pour méthode de G-S en

utilisant le réseau test à 2383 bus n'est pas effectuée. Le nombre d'itérations requises ainsi que les différences entre les valeurs des tensions et des puissances obtenues par MATPOWER et nos algorithmes sont listées au Tableau 7.3. Ces résultats confirment que notre parallélisation sur GPU n'affecte pas l'exactitude des algorithmes de G-S et de N-R. Toutefois, il est intéressant de noter que le nombre d'itérations requises par notre implémentation de la méthode de G-S est plus grand que pour celle de MATPOWER. Cette différence est due au fait que les threads CUDA™ exécutent en parallèle et sont forcés d'utiliser les valeurs calculées à l'itération précédente lors de l'évaluation des tensions aux bus à l'équation (7.9). Dans le cas d'une implémentation séquentielle, il est possible d'utiliser les valeurs les plus récentes ce qui accélère la convergence de la méthode. Toutefois, le nombre plus élevé d'itérations n'affecte pas la précision de l'algorithme, mais seulement le temps d'exécution qui sera facilement caché par la parallélisation sur GPU.

TABLEAU 7.3
COMPARAISON DES RÉSULTATS DE L'ANALYSE DE L'ÉCOULEMENT DE PUISSANCE OBTENUS PAR LES ALGORITHMES PARALLÈLES SUR GPU ET PAR MATPOWER [136]

Réseaux	Nombre d'itérations		Erreurs comparés à MATPOWER			
	MATPOWER	GPU	Vm (p.u.)	Va (p.u.)	P (p.u.)	Q (p.u.)
Gauss-Seidel:						
30-bus	670	1,034	1.5E-10	1.8E-07	1.6E-08	4.0E-09
118-bus	2,111	2,695	4.2E-11	1.5E-07	3.4E-08	2.6E-09
300-bus	21,536	32,466	1.4E-10	2.1E-07	3.8E-08	2.0E-09
Newton-Raphson:						
30-bus	3	3	5.6E-16	2.5E-13	2.5E-14	1.2E-14
118-bus	3	3	8.9E-16	1.2E-12	4.6E-13	5.9E-14
300-bus	5	5	2.0E-14	6.7E-12	1.2E-12	5.6E-13
2383-bus	6	6	2.7E-13	2.4E-10	2.4E-10	2.3E-11

7.2.4.2 Temps d'exécution et accélération

Le second test mesure les temps d'exécution et les accélérations de nos algorithmes. Pour rendre la comparaison plus intéressante, nous avons également développé une implémentation parallèle sur CPU multicœur utilisant OpenMP®. En fonction du nombre de réseaux à analyser, le programme lance jusqu'à 32 threads OpenMP® et chaque thread analyse un ou plusieurs réseaux suivant une répartition statique du travail. Cette stratégie de parallélisation à gros grains minimise la communication inter-thread et maximise l'accélération. Le nombre maximal de threads OpenMP® est fixé à 32 afin de prendre plein avantage de la fonctionnalité *hyperthreading* des 16 cœurs présents sur le système Dell T7600 utilisé pour le test. Les temps d'exécution pour nos implémentations séquentielles sur CPU, parallèles sur CPU et parallèles sur GPU ainsi que les accélérations associées sont listés au

Tableau 7.4 pour des réseaux tests dont les grosseurs varient de 4 à 2383 bus. Nous comparons aussi au Tableau 7.5 les temps d'exécution de nos implémentations à ceux d'autres implémentations publiées dans littératures. Pour chacune des implémentations citées, nous donnons la référence, le type d'implémentation (séquentielle sur CPU, parallèle sur CPU utilisant OpenMP® ou parallèle sur GPU utilisant CUDA™) et la représentation utilisée (matrices creuses ou pleines).

TABLEAU 7.4
TEMPS D'EXÉCUTION DES IMPLÉMENTATIONS SÉQUENTIELLES SUR CPU, PARALLÈLES SUR CPU ET PARALLÈLES SUR GPU (MOYENNES DE 100 ESSAIS)

Réseau	Nombre d'itérations	Pour 1 réseau					Pour 10 réseaux				
		Temps moyens (ms)			Accélération		Temps moyens (ms)			Accélération	
		CPU	OMP	GPU	OMP	GPU	CPU	OMP	GPU	OMP	GPU
Gauss-Seidel											
4-bus	28	0.1	2.1	4.5	**0.0x**	0.0x	0.3	2.0	4.3	**0.2x**	0.1x
30-bus	1,034	8.5	37.3	145.0	**0.2x**	0.1x	74.7	45.0	146.9	**1.7x**	0.5x
118-bus	2,695	85.8	174.9	402.5	**0.5x**	0.2x	831.9	221.0	416.9	**3.8x**	2.0x
300-bus	32,466	2,506.5	3,913.5	5,598.8	**0.6x**	0.4x	24,858.1	5,019.5	5,616.3	**5.0x**	4.4x
Newton-Raphson											
4-bus	3	0.1	0.2	1.6	**0.3x**	0.0x	0.5	0.3	2.4	**1.7x**	0.2x
30-bus	3	0.7	1.1	2.7	**0.6x**	0.2x	5.7	1.3	3.2	**4.5x**	1.8x
118-bus	3	2.3	3.2	6.0	**0.7x**	0.4x	20.2	4.0	8.1	**5.0x**	2.5x
300-bus	5	12.0	16.5	23.4	**0.7x**	0.5x	117.1	18.1	28.3	**6.5x**	4.1x
2383-bus	6	125.5	173.1	226.5	**0.7x**	0.6x	1,208.4	201.7	250.1	**6.0x**	4.8x

Réseau	Nombre d'itérations	Pour 100 réseaux					Pour 500 réseaux				
		Temps moyens (ms)			Accélération		Temps moyens (ms)			Accélération	
		CPU	OMP	GPU	OMP	GPU	CPU	OMP	GPU	OMP	GPU
Gauss-Seidel											
4-bus	28	2.7	2.1	4.5	**1.3x**	0.6x	13.5	3.0	5.8	**4.5x**	2.3x
30-bus	1,034	742.5	89.3	149.6	**8.3x**	5.0x	3,720.8	267.5	255.7	13.9x	**14.5x**
118-bus	2,695	8,373.0	675.5	528.1	12.4x	**15.9x**	41,929.5	2,634.1	1172.9	15.9x	**35.7x**
300-bus	32,466	249,506.4	17,997.9	9,594.9	13.9x	**26.0x**	1,272,433.3	73,150.5	28,146.3	17.4x	**45.2x**
Newton-Raphson											
4-bus	3	4.7	0.9	3.0	**5.3x**	1.6x	23.0	3.8	6.1	**6.0x**	3.7x
30-bus	3	53.3	5.1	6.6	**10.5x**	8.0x	268.6	26.2	26.2	**10.3x**	**10.3x**
118-bus	3	200.3	21.3	19.0	9.4x	**10.6x**	1,030.1	123.4	70.9	8.3x	**14.5x**
300-bus	5	1,155.6	102.5	96.6	11.3x	**12.0x**	5,628.4	458.6	315.9	12.3x	**17.8x**
2383-bus	6	11,773.1	1,319.9	1,024.3	8.9x	**11.5x**	59,620.0	5,705.0	4,504.9	10.5x	**13.2x**

TABLEAU 7.5
COMPARAISON ENTRE LES TEMPS D'EXÉCUTION DES IMPLÉMENTATIONS PROPOSÉES SUR GPU ET CEUX D'AUTRES RÉFÉRENCES POUR L'ANALYSE DE DIFFÉRENTS RÉSEAUX TESTS

Réseau	Référence	Temps d'exécution (secondes)			
		Gauss-Seidel		Newton-Raphson	
		1 réseau	500 réseaux	1 réseau	500 réseaux
30 bus	[133] GPU pleine	0.023	*11.276	0.003	*1.293
	[135] GPU pleine	0.705	*352.550	0.009	*4.700
	[136] CPU creuse	0.505	*252.600	0.003	*1.650
	Notre CPU creuse	**0.008**	3.721	**0.001**	0.269
	Notre OMP creuse	0.037	0.267	0.001	0.026
	Notre GPU creuse	0.145	**0.256**	0.003	**0.026**
118 bus	[133] GPU pleine	0.051	*25.360	0.016	*8.081
	[135] GPU pleine	3.296	*1648.150	0.200	*99.850
	[136] CPU creuse	3.845	*1922.350	0.006	*3.050
	Notre CPU creuse	**0.086**	41.929	**0.002**	1.030
	Notre OMP creuse	0.175	2.634	0.003	0.123
	Notre GPU creuse	0.403	**1.173**	0.006	**0.071**
300 bus	[133] GPU pleine	0.119	*59.260	0.116	*57.762
	[135] GPU pleine	7.299	*3649.600	2.685	*1342.400
	[136] CPU creuse	10.388	*5193.950	0.020	*9.950
	Notre CPU creuse	**2.507**	1272.433	**0.012**	5.628
	Notre OMP creuse	3.913	73.151	0.017	0.459
	Notre GPU creuse	5.599	**28.146**	0.023	**0.316**

* Étant donné que ces implémentations sont conçues pour l'analyse d'un seul réseau à la fois, les temps d'exécutions listés sont calculés en multipliant par 500 les temps pour l'analyse d'un seul réseau.

D'après les résultats obtenus aux Tableaux 7.4 et 7.5, nous effectuons trois observations. Premièrement, nous concluons que la parallélisation sur GPU offre une accélération significative, jusqu'à 45.2x dans le cas de l'algorithme de G-S et 17.8x dans le cas de N-R. Toutefois, comparé à l'implémentation OpenMP® sur processeur multicœur, l'avantage du GPU est observé uniquement lorsque le nombre de réseaux analysés concurremment est égal ou supérieur à 100. Dans le contexte de l'optimisation de l'écoulement de puissance par une métaheuristique, le nombre de scénarios contingents analysés est très grand. La parallélisation sur GPU de l'analyse de PF est alors avantageuse.

Deuxièmement, nous confirmons que l'utilisation de matrices creuses est essentielle au développement d'algorithmes performants pour une analyse rapide de l'écoulement de puissance. À cause de la complexité associée à l'organisation irrégulière des données dans les matrices creuses, les travaux antérieurs sur la parallélisation d'algorithmes de PF sur GPU se sont limités à l'utilisation de matrices pleines. Même si le GPU d'accélérer les calculs, la complexité de l'algorithme est quadratique et le temps d'exécution résultant demeure considérable. Par exemple,

comme nous l'avons identifié au Tableau 7.5, l'implémentation parallèle sur GPU de l'algorithme de N-R publiée dans [135] nécessite 2.685 s pour résoudre une seule instance du réseau test IEEE à 300 bus ce qui représente en une accélération de 1.75x comparativement à une exécution séquentielle sur GPU. Dans notre cas, 500 instances du réseau IEEE à 300 bus sont analysées en seulement 0.315 s, soit trois ordres de magnitude plus rapidement.

Finalement, malgré que la méthode de G-S s'adapte bien à l'architecture du GPU et offre une excellente accélération, elle reste beaucoup plus lente que la méthode de N-R à cause du nombre supérieur d'itérations nécessaires avant convergence. Donc, à moins que le réseau soit limité à 4 bus, il est préférable d'utiliser la méthode de N-R pour une meilleure performance. Pour cette raison, la métaheuristique proposée à la section suivante pour l'optimisation de l'écoulement de puissance utilise la méthode de N-R lors de l'évaluation des solutions candidates.

Dans cette section, nous avons présenté les détails de nos implémentations parallèles sur GPU des algorithmes d'analyse d'écoulement de puissance de Gauss-Seidel et de Newton-Raphson. Nos implémentations utilisent des matrices creuses afin de réduire l'ordre de complexité des calculs et parallélisent toutes les étapes des algorithmes en tirant avantage des différents processeurs disponibles dans un système hybride CPU-GPU. Le logiciel résultant offre une excellente accélération et un temps de calcul réduit comparé aux solutions antérieures publiées dans la littérature. Il est réutilisé à la section suivante pour le développement d'une métaheuristique pour l'optimisation de l'écoulement de puissance.

7.3 Optimisation de l'écoulement de puissance

L'optimisation de l'écoulement de puissance consiste à calculer les réglages optimaux des générateurs et de l'équipement d'appoint d'un système de transport d'électricité en régime permanent afin de satisfaire la demande en énergie tout en maximisant une fonction objective. Il s'agit d'un problème d'optimisation à variables mixtes, non linéaire et non convexe. Dans cette section, nous proposons un algorithme d'optimisation par essaim de particules pour calculer des solutions au problème d'OPF. Les variables de contrôle considérées sont les puissances actives et les tensions des générateurs, les puissances réactives des compensateurs statiques et les ratios des transformateurs. La métaheuristique est implémentée sur le GPU à l'aide du cadriciel *gpuMF* et utilise le module d'analyse de l'écoulement de puissance de N-R développé à la section précédente pour l'évaluation parallèle des solutions candidates. La méthode proposée considère les variables discrètes, trouve des solutions de meilleure qualité que les implémentations antérieures et minimise le temps de calcul en offrant une accélération de 17.2x comparée à une exécution séquentielle sur CPU. Cette section est organisée comme suit. Nous présentons d'abord la formulation

mathématique du problème d'OPF. Nous enchaînons avec la stratégie d'optimisation proposée. Nous discutons ensuite de la parallélisation sur GPU. Finalement, nous terminons avec les tests expérimentaux effectués sur les réseaux tests IEEE à 30, 118 et 300 bus.

7.3.1 Définition du problème

La formulation mathématique du problème de l'optimisation de l'écoulement de puissance peut être représentée sous la forme générique suivante d'après [113] :

$$\begin{aligned} &\text{minimiser} && f(\bar{x},\bar{u}) \\ &\text{sujet à} && g(\bar{x},\bar{u}) = 0 \\ & && h(\bar{x},\bar{u}) \leq 0 \end{aligned} \tag{7.14}$$

où f est la fonction objective, g représente les contraintes d'égalités et h, les contraintes d'inégalités. Le vecteur $\bar{x}$ contient les variables de contrôle et le vecteur $\bar{u}$ est l'état du réseau. Dans ce travail, le vecteur de contrôle $\bar{x}$ inclut les puissances actives des générateurs, la tension des générateurs, le rapport des transformateurs et la capacitance des compensateurs statiques d'énergie réactive. Le vecteur d'état $\bar{u}$ inclut la tension complexe et la puissance complexe injectées à chaque bus du réseau.

7.3.1.1 Fonction objective

La fonction objective $f(\bar{x},\bar{u})$ représente l'objectif de l'optimisation. Dans ce travail, nous la définissons comme suit:

$$f(\bar{x},\bar{u}) = w_1 * f_{coûts}(\bar{x},\bar{u}) + w_2 * f_{pertes}(\bar{x},\bar{u}) + w_3 * f_{émissions}(\bar{x},\bar{u}) \tag{7.15}$$

où $f_{coûts}$ sont les coûts de production, f_{pertes} sont les pertes de transport, $f_{émissions}$ sont les émissions polluantes et w_1, w_2 et w_3 sont les poids respectifs. Cette fonction contient trois termes afin de donner à l'utilisateur la flexibilité de choisir l'objectif d'optimisation. Il faut simplement assigner une valeur de 1 au poids associé à l'objectif choisi et une valeur de 0 aux deux autres. Malgré que nous considérons un seul objectif à la fois, une optimisation réellement multi objective serait aussi possible en utilisant la méthode de partitionnement de données démontrée dans [257] pour assigner les poids dans l'équation de façon à obtenir le front Pareto de solutions non dominées.

Dans l'équation (7.15), les coûts de production pour le réseau en entier sont calculés en additionnant les coûts pour chaque générateur. Ceux-ci sont habituellement modélisés par une fonction quadratique [136]. Les coûts totaux pour le réseau sont donc :

$$f_{coûts}(\bar{x}, \bar{u}) = \sum_{i=1}^{n_{gen}} a_i + b_i P_{Gi} + c_i P_{Gi}^2 \tag{7.16}$$

où n_{gen} est le nombre de générateurs, P_{Gi} est la puissance active produite par le générateur i et a_i, b_i et c_i sont les coefficients de l'équation quadratique modélisant les coûts du générateur i. À l'équation (7.15), les pertes de transport totales sont calculées comme suit en soustrayant la puissance demandée de la puissance produite pour le réseau en entier :

$$f_{pertes}(\bar{x}, \bar{u}) = \sum_{i=1}^{n_{gen}} P_{Gi} - \sum_{j=1}^{n_{bus}} P_{Dj} \tag{7.17}$$

où n_{bus} est le nombre de bus dans le réseau et P_{Dj} est la demande en puissance au bus j. Finalement, les émissions polluantes sont mesurées en tonnes métriques et calculées d'après le modèle publié dans [114] comme suit :

$$f_{émissions}(\bar{x}, \bar{u}) = \sum_{i=1}^{n_{gen}} \frac{\alpha_i + \beta_i P_{Gi} + \gamma_i P_{Gi}^2}{100} + \zeta_i e^{\lambda_i P_{Gi}} \tag{7.18}$$

où α_i, β_i, γ_i, ζ_i et λ_i sont les coefficients d'émission polluante pour le générateur i.

7.3.1.2 Contraintes d'égalités

Les contraintes d'égalités $g(\bar{x}, \bar{u})$ dans la formulation du problème d'OPF sont dérivées à partir des équations d'écoulement de puissance aux équations (7.7) et (7.8) et reflètent les lois de la physique du réseau électrique. Elles affirment que l'injection nette de puissance active et réactive doit être nulle à chaque bus.

$$P_{Gi} - P_{Di} - \sum_{k=1}^{n_{bus}} |V_i|\,|V_k|\,(g_{ik}\cos(\delta_i - \delta_k) + b_{ik}\sin(\delta_i - \delta_k)) = 0 \tag{7.19}$$

$$Q_{Gi} - Q_{Di} - \sum_{k=1}^{n_{bus}} |V_i|\,|V_k|\,(g_{ik}\,sin(\delta_i - \delta_k) + b_{ik}\,cos(\delta_i - \delta_k)) = 0 \tag{7.20}$$

où P_{Gi} et P_{Di} sont respectivement les puissances actives générées et demandées aux bus i. Similairement, Q_{Gi} et Q_{Di} représentent les puissances réactives générées et demandées aux bus i. $|V|$ et δ sont la magnitude et l'angle des phaseurs de tension aux bus. g_{ik} et b_{ik} sont la conductance et la susceptance de la branche connectant le bus i au bus k. Leurs valeurs sont prises de la matrice d'admittance du réseau. Dans la méthode d'OPF proposée dans ce chapitre, le respect des contraintes d'égalités est garanti par l'analyse de l'écoulement de puissance de N-R.

7.3.1.3 Contraintes d'inégalités

Les contraintes d'inégalités $h(\bar{x}, \bar{u})$ dans la formulation du problème d'OPF reflètent les limites physiques de l'infrastructure du réseau de transport d'électricité. Elles sont définies comme suit :

$$|V_i|_{min} \leq |V_i| \leq |V_i|_{max} \qquad \text{pour } i = 1, \dots, n_{bus} \tag{7.21}$$

$$P_{Gi\,min} \leq P_{Gi} \leq P_{Gi\,max} \qquad \text{pour } i = 1, \dots, n_{gen} \tag{7.22}$$

$$Q_{Gi\,min} \leq Q_{Gi} \leq Q_{Gi\,max} \qquad \text{pour } i = 1, \dots, n_{gen} \tag{7.23}$$

$$q_{ci\,min} \leq q_{ci} \leq q_{ci\,max} \qquad \text{pour } i = 1, \dots, n_{SVAR} \tag{7.24}$$

$$T_{i\,min} \leq T_i \leq T_{i\,max} \qquad \text{pour } i = 1, \dots, n_{trans} \tag{7.25}$$

$$|S_i| \leq |S_{i\,max}| \qquad \text{pour } i = 1, \dots, n_{br} \tag{7.26}$$

où $|V_i|$ est la magnitude de la tension au bus i, P_{Gi} et Q_{Gi} sont les puissances actives et réactives de générateurs, q_{ci} est la puissance réactive des compensateurs statiques d'énergie réactive (SVAR de l'anglais *static VAR compensator*), T_i est le rapport des transformateurs et $|S_i|$ est la puissance apparente qui circule dans la branche i. Les termes n_{bus}, n_{gen}, n_{SVAR}, n_{trans} et n_{br} sont respectivement les nombres totaux de bus, de générateurs, de SVAR, de transformateurs et de branches dans le réseau.

7.3.2 Stratégie d'optimisation

7.3.2.1 Optimisation par essaims de particules

Dans [113], Frank et coll. présentent une revue de la littérature sur les méthodes non déterministes pour l'optimisation de l'écoulement de puissance. Ils identifient plusieurs métaheuristiques capables d'optimiser des solutions au problème d'OPF. D'après le nombre de références citées, ce sont l'algorithme génétique (GA de l'anglais *genetic algorithm*) et le PSO qui sont les plus populaires. Bien que des comparaisons entre le GA et le PSO appliqués au problème d'OPF sont disponibles dans [258], [259] et [260], Frank et coll. nous avisent qu'il est difficile d'identifier la meilleure méthode étant donné que l'efficacité de chacune dépend de plusieurs facteurs incluant les paramètres de configuration et l'encodage des solutions candidates. Dans cette thèse, nous choisissons d'utiliser l'optimisation par essaim de particules. Cet algorithme est simple à implémenter, facile à configurer et sa convergence a été prouvée analytiquement dans [209]. Finalement, comparé au GA, le PSO requiert un temps d'exécution plus court [259] et s'adapte mieux à l'architecture parallèle du GPU [261].

Une description complète du PSO a été donnée au Chapitre 3. L'algorithme se base sur une population de solutions candidates et simule le mouvement d'un essaim

de particules dans un espace multidimensionnel. Le mouvement est inspiré par celui d'une volée d'oiseaux ou d'un banc de poissons. La position d'une particule représente une solution candidate et celle-ci est modifiée à chaque itération de l'algorithme de façon à se diriger vers les meilleures positions visitées antérieurement par la particule et par l'essaim. La nouvelle position est calculée en utilisant les équations (3.1) et (3.2) données précédemment au Chapitre 3. Dans notre implémentation, les paramètres ω, c_1 et c_2 sont fixées à 0.729, 1.496 et 1.496 de façon à garantir la convergence de l'algorithme [209]. De plus, les vitesses $\vec{v}_{t+1}$ des particules sont limitées à 25% de l'intervalle permis pour chaque dimension afin d'éviter une convergence prématurée et améliorer l'exploration de l'espace de recherche. Finalement, les nouvelles positions $\vec{x}_{t+1}$ sont aussi bornées de façon à respecter les limites des variables de contrôles optimisées.

7.3.2.2 Représentation des solutions candidates

Les solutions candidates sont encodées à l'aide d'un vecteur de nombres réels défini comme suit et dont les valeurs sont normalisées entre 0 et 1.

$$solution = \left\{P_2, \dots, P_{n_{gen}}, |V|_1, \dots, |V|_{n_{gen}}, Q_1, \dots, Q_{n_{SVAR}}, T_1, \dots, T_{n_{tr}}\right\} \quad (7.27)$$

où P_i est la puissance produite au générateur i, $|V|_i$ est l'amplitude de la tension au générateur i, Q_j est la puissance réactive du SVAR j et T_k est le rapport du transformateur k. Pour un réseau où les nombres de générateurs, de SVAR et de transformateurs sont respectivement n_{gen}, n_{SVAR} et n_{tr}, le vecteur de solution inclus $n_{gen} - 1$ éléments pour les puissances actives des générateurs, n_{gen} éléments pour les magnitudes des tensions des générateurs, n_{SVAR} éléments pour les puissances réactives des SVAR et n_{tr} éléments pour les ratios des transformateurs. Il est important de noter que la puissance active du générateur au bus de référence n'est pas incluse dans le vecteur de solution puisque celle-ci est une variable dépendante calculée par l'analyse de PF.

À chaque itération du PSO, les solutions sont décodées et leur qualité est évaluée à l'aide de la fonction d'aptitude. Pour décoder une solution, les éléments normalisés sont multipliés par un facteur afin de les ramener à leur échelle originale d'après leurs limites respectives définies aux équations (7.21), (7.22), (7.24) et (7.25). Dans le cas des puissances réactives des SVAR et des ratios des transformateurs, les valeurs sont aussi arrondies à leur valeur discrète la plus proche après avoir été mises à l'échelle. Cet arrondissement est effectué avant l'évaluation de la fonction de coût ce qui signifie que le PSO proposé considère uniquement des solutions candidates avec des valeurs discrètes pour le réglage des SVAR et des transformateurs. De plus, contrairement à la technique d'arrondissement [161] discuté au Chapitre 2, notre approche n'alterne pas entre les variables continues et discrètes, mais optimise toutes les variables d'un seul coup permettant ainsi une optimisation globale. Finalement, après avoir décodé

les vecteurs des solutions candidates, les valeurs obtenues sont insérées dans les structures de données de l'objet *PFCase* de notre module d'analyse de PF développé à la section 7.2. L'analyse de l'écoulement de puissance de N-R est alors effectuée sur le GPU en traitant concurremment tous les scénarios contingents. Cette analyse permet d'obtenir l'état complet des réseaux associés aux solutions candidates.

7.3.2.3 Fonction d'aptitude

La fonction d'aptitude permet au PSO de comparer la qualité des solutions candidates et d'identifier la meilleure à chaque itération de l'algorithme. Cette solution est alors utilisée par le PSO pour modifier les autres afin de les améliorer. La fonction d'aptitude doit tenir compte de l'objectif d'optimisation et des multiples contraintes. Une approche commune pour intégrer les contraintes dans la fonction d'aptitude est d'assigner des pénalités aux solutions qui ne les respectent pas. Tel qu'il est fait dans [110], [167] et [262] ces pénalités peuvent être calculées de façon à être proportionnelles au niveau de violation des contraintes. Toutefois, cette approche n'assure pas toujours une séparation évidente entre les solutions faisables et celles qui ne le sont pas. Dans cette thèse, nous proposons une fonction d'aptitude unique qui normalise l'objectif d'optimisation et la pénalité appliquée afin d'offrir une séparation claire basée uniquement sur la valeur numérique calculée. Ceci assure qu'une solution faisable ait toujours une valeur supérieure à une solution infaisable. Finalement, si aucune des solutions candidates initiales n'est faisable, la fonction d'aptitude définie permet quand même de comparer la qualité des solutions infaisables. La métaheuristique peut alors améliorer ces solutions pour éventuellement en obtenir qui sont faisables.

Dans la méthode d'OPF proposée, les contraintes d'égalités sont imposées par l'analyse de PF et n'ont pas besoin d'être vérifiées par la fonction d'aptitude. De plus, les contraintes d'inégalités sur P_{Gi}, q_{ci} et T_i aux équations (7.22), (7.24) et (7.25) sont assurées par l'encodage utilisée pour les solutions candidates. Elles aussi n'ont pas besoin d'être vérifiées. La seule exception est la limite sur P_{Gi} au bus de référence étant donné que cette variable de contrôle ne fait pas partie du vecteur de solution puisqu'elle est calculée par l'analyse de PF. Cette contrainte ainsi que les autres définies aux équations (7.21), (7.23) et (7.26) doivent donc être vérifiées par la fonction d'aptitude.

La première étape pour calculer l'aptitude $\mathcal{F}(\bar{x}, \bar{u})$ d'une solution candidate est d'évaluer la fonction objective $f(\bar{x}, \bar{u})$ à l'aide de l'équation (7.15) et de normaliser la valeur obtenue sur l'intervalle 0 à 1 somme suit, où 0 représente une très mauvaise solution tandis que 1 en représente une très bonne :

$$f_{NORM}(\bar{x}, \bar{u}) = \frac{1}{1 + f(\bar{x}, \bar{u})} \tag{7.28}$$

Deuxièmement, un facteur de violation est calculé pour tenir compte des contraintes d'inégalités. Pour vérifier la contrainte définie par l'équation (7.21) sur les limites maximums et minimums des tensions aux bus, chaque bus est inspecté indépendamment et tout excès de tension est quantifié comme suit :

$$E(|V_i|) = \begin{cases} \dfrac{|V_i| - |V_i|_{max}}{|V_i|_{max} - |V_i|_{min}}, & |V_i| > |V_i|_{max} \\ \dfrac{|V_i|_{min} - |V_i|}{|V_i|_{max} - |V_i|_{min}}, & |V_i| < |V_i|_{min} \\ 0, & |V_i|_{min} \leq |V_i| \leq |V_i|_{max} \end{cases} \tag{7.29}$$

où $E(|V_i|)$ est l'excès relatif de la tension au bus i. En divisant la différence absolue par l'intervalle des valeurs permises, nous obtenons une valeur relative pour l'excès de tension ce qui permet une comparaison plus juste lorsque les limites varient d'un bus à l'autre. Une approche similaire est utilisée pour calculer l'excès relatif $E(P_{Gi})$ sur les puissances actives des générateurs au bus de référence, l'excès relatif $E(Q_{Gi})$ sur les puissances réactives de tous les générateurs et l'excès relatif $E(|S_i|)$ sur la puissance maximale d'écoulement des branches du réseau. Les valeurs relatives obtenues sont ensuite additionnées comme suit pour obtenir l'excès relatif $E_{total}(\bar{u}, \bar{x})$ pour le réseau en entier :

$$E_{total}(\bar{u}, \bar{x}) = \sum_{i=1}^{n_{bus}} E(|V_i|) + \sum_{i=1}^{n_{SLACK\ gen}} E(P_{Gi}) + \sum_{i=1}^{n_{gen}} E(Q_{Gi}) + \sum_{i=1}^{n_{branch}} E(|S_i|) \tag{7.30}$$

Comme nous l'avons fait pour la fonction objective, nous normalisons la somme obtenue entre 0 et 1 afin de former le facteur de violation normalisé $\mathcal{V}_{NORM}(\bar{u}, \bar{x})$. Un facteur près de 0 représente une solution infaisable qui excède largement plusieurs contraintes. À l'inverse, un facteur exactement égal à 1 représente une solution faisable qui respecte toutes les contraintes du réseau.

$$\mathcal{V}_{NORM}(\bar{x}, \bar{u}) = \frac{1}{1 + E_{total}(\bar{u}, \bar{x})} \tag{7.31}$$

Finalement, les valeurs normalisées de la fonction objective et du facteur de violation sont regroupées comme suit afin d'obtenir l'aptitude $\mathcal{F}(\bar{x}, \bar{u})$ de la solution candidate:

$$\mathcal{F}(\bar{u}, \bar{x}) = \begin{cases} 1 + f_{NORM}(\bar{x}, \bar{u}), & \mathcal{V}_{NORM}(\bar{x}, \bar{u}) = 1 \\ \mathcal{V}_{NORM}(\bar{x}, \bar{u}), & \mathcal{V}_{NORM}(\bar{x}, \bar{u}) < 1 \end{cases} \tag{7.32}$$

Cette définition a été soigneusement pensée de façon à distinguer facilement les solutions faisables de celles qui ne le sont pas. En effet, les solutions faisables auront

toutes des aptitudes dans l'intervalle 1 à 2 tandis que les solutions infaisables seront entre 0 et 1. De plus, lorsque deux solutions faisables sont comparées, celle qui a la plus grande valeur d'aptitude est de meilleure qualité puisqu'elle minimise davantage la fonction objective, soit les coûts de production, les pertes de transport ou les émissions polluantes. Similairement, lorsque deux solutions infaisables sont comparées, celle qui a la plus grande valeur d'aptitude est de meilleure qualité puisqu'elle enfreint les contraintes physiques du réseau à un degré moindre que l'autre. Même si toutes les solutions candidates initiales sont infaisables, la fonction d'aptitude proposée permet à la métaheuristique de classer ces solutions en ordre de qualité et de les améliorer pour éventuellement en obtenir qui sont faisables.

7.3.2.4 Approche à multiples phases

Toutes les métaheuristiques offrent un compromis entre l'exploration de l'espace de recherche et la convergence vers la solution finale. Dans le cas du PSO, l'exploration est stimulée par l'influence sociale de l'essaim, soit la meilleure position occupée par l'essaim, qui peut forcer une particule à traverser l'espace de recherche. La convergence est quant à elle encouragée par l'influence personnelle qui incite une particule à retourner vers sa meilleure position antérieure. Pour des problèmes à grande échelle avec de nombreuses variables de contrôle comme celui de l'OPF, la capacité d'exploration de la métaheuristique choisie n'est pas toujours suffisante et la convergence prématurée vers un optimum local est possible. Pour éviter ce problème et stimuler davantage l'exploration de l'algorithme de recherche, la méthode d'OPF que nous proposons utilise une approche à multiples phases similaire à celle de Mahdad et Srairi publiée dans la référence [169]. Cependant, au lieu d'utiliser trois métaheuristiques différentes comme le font les auteurs, notre approche utilise le PSO à chacune des phases afin de simplifier la conception. À la phase initiale, les solutions candidates sont initialisées aléatoirement suivant une distribution uniforme couvrant l'espace de recherche en entier. Aux phases suivantes, elles sont initialisées suivant une distribution normale centrée sur la meilleure solution trouvée à la phase précédente. Comme nous le verrons dans la section des résultats, utiliser un PSO à chacune des phases n'affecte pas l'efficacité de la recherche. En fait, notre approche permet de trouver des solutions de meilleure qualité que celles des auteurs de [169] pour un même réseau test.

7.3.3 Parallélisation

7.3.3.1 Métaheuristique

La technique d'optimisation proposée dans ce chapitre pour résoudre le problème d'OPF offre plusieurs possibilités pour une parallélisation sur GPU. Tout d'abord, le PSO simule le mouvement d'un grand nombre de particules. La vitesse et la position de chaque particule peuvent être calculées en parallèle en utilisant un thread CUDA™ par particule ou même un thread par dimension de chaque particule afin d'exploiter

un niveau de parallélisme plus élevé. La recherche de la solution avec la plus grande valeur d'aptitude peut se faire avec une primitive de réduction parallèle. Dans cette thèse, nous utilisons le cadriciel *gpuMF* pour implémenter le PSO parallèle sur GPU. Comme nous l'avons vu au Chapitre 5, *gpuMF* exploite bien l'architecture du GPU, parallélise efficacement toutes les étapes de la métaheuristique et permet une accélération considérable comparée à une exécution séquentielle sur CPU. De plus, *gpuMF* crée, modifie et évalue les solutions candidates directement sur le GPU afin de minimiser les transferts de données sur le bus PCIe. Dans notre implémentation, le PSO est configuré de façon à organiser les solutions candidates en îlots pour améliorer l'exploration de l'espace de recherche et éviter une convergence prématurée vers un optimum local [43]. Un mécanisme de communication suivant une topologie circulaire bidirectionnelle est implémenté pour échanger les meilleures solutions à une fréquence prédéterminée. La métaheuristique est exécutée en plusieurs phases comme nous l'avons expliqué précédemment et chaque phase compte un nombre fixe d'itérations. Le nombre de phases et d'itération est déterminé expérimentalement afin d'obtenir la meilleure solution possible tout en minimisant le temps de calcul.

7.3.3.2 Fonction d'aptitude

À chaque itération du PSO, la qualité des solutions candidates doit être évaluée à l'aide de la fonction d'aptitude. Pour ce faire, les vecteurs de solution sont décodés à l'aide d'un thread par élément et les valeurs obtenues sont insérées dans l'objet *PFCase* du module d'analyse de PF développé à la section 7.2 afin de former les scénarios contingents. Chaque solution candidate est associée à un scénario différent. L'analyse de PF utilisant la méthode de N-R est ensuite exécutée en parallèle sur le GPU et permet d'obtenir l'état complet pour chaque réseau. Cet état est défini par les puissances complexes injectées aux bus, les tensions complexes aux bus et les puissances qui s'écoulent dans les branches. À ce moment, toutes les données nécessaires pour évaluer la fonction d'aptitude sont disponibles sur le GPU dans l'objet *PFCase*. La fonction objective normalisée à l'équation (7.28) est calculée en parallèle en utilisant un thread par générateur et les valeurs obtenues sont additionnées à l'aide d'une primitive de réduction parallèle pour chaque solution candidate. Les excès relatifs définis à l'équation (7.29) pour les contraintes d'inégalités sont évalués à l'aide d'un thread par bus, par générateur et par branche. Ils sont ensuite additionnés avec une opération de réduction parallèle. Finalement l'excès total, le facteur de violation normalisé et la fonction d'aptitude sont évalués par une fonction map parallèle utilisant un thread par réseau. Après que l'aptitude de chaque fonction candidate soit calculée, l'algorithme de PSO reprend son exécution.

7.3.3.3 Parallélisation sur CPU

Comme nous l'avons fait pour la minimisation des harmoniques d'un onduleur multiniveau, nous développons une version parallèle pour processeur multicœur de notre programme d'optimisation de l'écoulement de puissance. Cette implémentation

utilise OpenMP® pour distribuer les calculs sur plusieurs threads afin d'exploiter pleinement la puissance de calcul des processeurs multicœurs.

Avant de développer la version parallèle sur CPU de notre algorithme d'optimisation, nous exécutons le programme séquentiel et mesurons le temps d'exécution des différentes fonctions logicielles qui le composent. Tout comme précédemment, le test est exécuté sur un ordinateur Dell T7600 équipé de deux CPU Intel Xeon E5-2650. Chaque CPU contient huit cœurs qui implémentent la technologie *hyperthreading*. L'ordinateur utilisé contient donc 16 cœurs physiques et 32 cœurs virtuels. Dans ce test, nous optimisons le réseau test IEEE à 30 bus. Le PSO est configuré de façon à utiliser 512 solutions candidates et à exécuter en deux phases de 100 itérations chaque. Les temps mesurés montrent que l'algorithme séquentiel prend 212.849 s à exécuter et que 212.643 s sont utilisées pour l'évaluation de la fonction d'aptitude, soit plus de 99.9% du temps d'exécution. Ceci n'est pas surprenant étant donné la complexité des calculs de l'analyse de PF. D'après ces mesures, une stratégie de parallélisation simple et efficace serait de partager les calculs nécessaires à l'évaluation de l'aptitude des solutions candidates sur les multiples cœurs du CPU suivant l'approche maître-esclave que nous avons discutée au Chapitre 2. D'après cette approche, le thread maître exécute le PSO séquentiellement sur un seul cœur et délègue le calcul de la fonction d'aptitude aux threads esclaves. Cette stratégie est simple d'implémentation, ne nécessite aucune modification au cadriciel *gpuMF*, exploite un niveau de parallélisme suffisant pour occuper les multiples cœurs du CPU et minimise la communication inter-threads à une seule synchronisation à chaque itération du PSO. Pour déterminer le nombre optimal de threads à utiliser, nous évaluons la fonction d'aptitude pour 512 solutions candidates en utilisant trois réseaux tests différents. Nous varions le nombre de threads de 1 à 48 et affichons à la Figure 7.7 l'accélération mesurée.

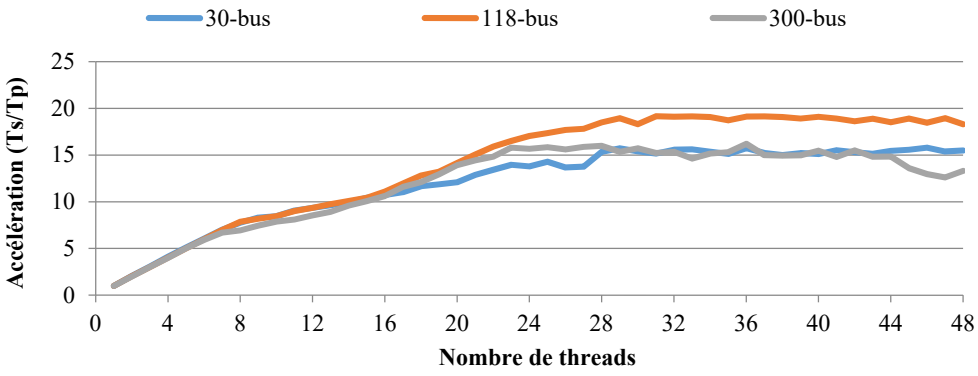


Figure 7.7: Accélération du PSO parallèle sur CPU par rapport au nombre de threads OpenMP® pour l'évaluation de la fonction objective du problème d'OPF (moyenne de 10 essais, E5-2650)

Sur cette figure, on remarque que l'accélération plafonne à 32 threads pour les trois réseaux considérés. Dans le cas du réseau IEEE à 30 bus, l'accélération maximale est de 15.58x. Nous utilisons la loi d'Amdahl [33] pour calculer à l'équation (7.33) l'accélération attendue pour le programme en entier. Cette accélération est définie comme le temps d'exécution du programme séquentiel divisé par celui du programme parallèle. Dans cette équation, la valeur 212.849 s représente le temps d'exécution du programme séquentiel en entier. La valeur 0.206 s représente le temps nécessaire à la partie du programme parallèle qui n'a pas été parallélisé, soit l'exécution du PSO par le thread maître. Finalement, le terme 212.643/15.58 s représente une estimation du temps nécessaire pour calculer en parallèle l'aptitude des solutions candidates par les threads esclaves. Ce calcul nous donne une accélération attendue de 15.35x, ce qui est très bon étant donné que le processeur utilisé contient 16 cœurs.

$$S = \frac{T_{séq}}{T_{par}} = \frac{212.849\ s}{0.206 + (212.643/15.58)\ s} = \frac{212.849\ s}{13.854\ s} = 15.36x \qquad (7.33)$$

7.3.4 Résultats expérimentaux

Dans cette section, nous testons l'efficacité de l'algorithme proposé en optimisant les réseaux tests IEEE à 30, 118 et 300 bus. Ces réseaux sont choisis puisqu'ils sont régulièrement utilisés dans la littérature scientifique ce qui nous permet de comparer nos résultats à ceux d'autres auteurs.

7.3.4.1 Réseau de 30 bus

Le premier réseau de transport d'électricité considéré dans ce travail est le réseau test IEEE à 30 bus illustré au début de ce chapitre à la Figure 7.1. Ce réseau représente une partie du système d'alimentation électrique américain dans le Midwest des États-Unis en décembre 1961. Les données de description sont disponibles à la référence [249] et reproduites à l'Appendice A. Le réseau contient six générateurs, quatre transformateurs et neuf compensateurs statiques d'énergie réactive pour un total de 24 variables de contrôle (P et $|V|$ des générateurs, T des transformateurs et Q des SVAR). Il est important de noter que la puissance active P du générateur au bus référence n'est pas optimisée par le PSO, mais bien calculée par l'analyse de PF. Cette valeur dépend alors des autres variables de contrôle. Le bus 1 est le bus de référence. Les limites permises pour la tension aux bus sont de 0.95 à 1.1 p.u. pour tous les bus. Les branches 11, 12, 15 et 36 sont des transformateurs en phase dont le rapport peut être varié de 0.9 à 1.1 par des incréments de 0.0125. Les bus 10, 12, 15, 17, 20, 21, 23, 24 et 29 sont équipés de SVAR avec une puissance réactive de 0 à 5 MVAR réglable par incréments de 0.5 MVAR. Finalement, les demandes totales de puissance active et réactive pour le réseau en entier sont respectivement de 283.6 MW et 126.2 MVAR.

Pour résoudre le cas test IEEE à 30 bus, nous configurons le cadriciel *gpuMF* de façon à exécuter le PSO en parallèle sur le GPU en utilisant 512 solutions candidates organisées en huit îlots avec un processus de migration tous les 20 itérations. L'optimisation est exécutée en deux phases. Une distribution uniforme sur l'espace de recherche en entier est utilisée à la première phase pour initialiser aléatoirement les solutions candidates. Une distribution normale centrée sur la solution trouvée est ensuite utilisée à la deuxième phase pour générer de nouvelles solutions et relancer la recherche. Chaque phase exécute le PSO pour 100 itérations. Pour permettre une comparaison avec les travaux antérieurs, nous assignons des valeurs de zéro au poids w_2 et w_3 à l'équation (7.15) afin que l'objectif d'optimisation considère uniquement la minimisation des coûts de production. Nous listons au Tableau 7.6 les coûts de production obtenus par notre algorithme et ceux d'autres références. Le cas de base représente les paramètres de contrôle réels qui étaient employés en décembre 1961. Les autres cas représentent les réglages optimaux calculés par différentes techniques d'optimisation publiées dans la littérature. On peut noter que les coûts de 799.03 $/h trouvé par notre métaheuristique parallèle sur GPU sont plus petits que n'importe lesquelles des références citées. Le vecteur de solution trouvé est reproduit à l'Appendice B et disponible pour téléchargement en format MATPOWER sur le site internet à la référence [263].

TABLEAU 7.6
COMPARAISON DE LA QUALITÉ DES SOLUTIONS CALCULÉES PAR DIFFÉRENTS ALGORITHMES POUR LE RÉSEAU TEST IEEE 30-BUS

Méthodes	Années de publication	Variables de contrôle	Résultats publiés	
			Coûts ($/h)	Pertes (MW)
Cas de base [249]	-	-	900.76	5.3900
Méthode de gradient[1] [249]	1985	P, |V|, T, Q_C	804.85	10.4850
Algorithme génétique[1] [115]	2002	P, |V|, T, Q_C	802.60	9.3800
Essaim de particules[2] [264]	2002	P, |V|, T, Q_C	800.41	9.0000
GA-PSO[2] [265]	2011	P, |V|, T, Q_C	801.81	9.3300
Point intérieur[1] [136]	2011	P, |V|	799.73	8.8070
Évolution différentielle[2] [266]	2013	P, |V|, T, Q_C	800.53	8.8144
Colonie d'abeilles[2] [267]	2013	P, |V|, T, Q_C	800.66	9.0328
Trou noir[2] [268]	2014	P, |V|, T, Q_C	799.92	8.6789
PSO-GPU proposé[2]	2015	P, |V|, T, Q_C	799.03	8.6128

[1] méthode déterministe
[2] méthode non déterministe

7.3.4.2 Réseau de 118 bus

Le deuxième réseau considéré est le réseau test IEEE à 118 bus. Ce réseau contient 54 générateurs, neuf transformateurs et 12 SVAR pour un total de 128 variables de contrôle. Les données de description du réseau sont prises de la référence [136]. Les limites des tensions sont de 0.94 et 1.06 p.u. pour tous les bus.

Les ratios des transformateurs peuvent être variés de 0.9 à 1.1 par incréments de 0.0125. Les SVAR sont configurables entre 0 et 30 MVAR par incréments de 0.5 MVAR. Pour tenir compte de la complexité plus élevée du réseau, le solveur est configuré de façon à exécuter en trois phases et le nombre d'itérations par phase est augmenté à 500. Les autres paramètres restent les mêmes. Comme pour le test précédent, la fonction objective est limitée à la minimisation des coûts de production afin de permettre une comparaison avec les travaux d'autres auteurs. Les détails de la solution trouvée par l'algorithme proposé sont listés à l'Appendice B et disponibles pour téléchargement en format MATPOWER sur le site internet à la référence [263]. Au Tableau 7.7, nous comparons la solution obtenue par notre méthode à celles d'autres auteurs. Les références citées utilisent la même description du réseau provenant de [136] et, dans la plupart des cas, inclus les détails des vecteurs de solutions obtenus. Ceci nous permet de confirmer leurs résultats à l'aide de l'outil MATPOWER. Dans le cas de notre approche parallèle sur GPU, les coûts de production de la solution finale sont de 129 627.03 $/h et les pertes de transport sont de 76.984 MW.

Comparé aux méthodes antérieures listées au Tableau 7.7, notre algorithme permet de trouver une solution d'excellente qualité tout en respectant les limites permises pour la tension aux bus et la puissance réactive des générateurs. En fait, il n'y a qu'une seule méthode antérieure qui permette un coût de production moindre que la nôtre, il s'agit de l'algorithme de recherche gravitationnelle publié dans [269]. Toutefois, comme pour plusieurs des métaheuristiques citées au Tableau 7.7, cet algorithme ne respecte pas les limites sur la puissance réactive Q permise aux générateurs. Pour certains générateurs, la solution trouvée nécessite une puissance réactive qui dépasse significativement la limite permise. Par exemple, la solution trouvée dans [269] demande des puissances réactives de 303.6 et -351 MVAR aux générateurs connectés aux bus 10 et 28 tandis que les limites permises sont respectivement de 140 et -67 MVAR. Les solutions calculées par ces métaheuristiques sont donc infaisables puisqu'elles enfreignent les contraintes physiques du réseau de transport d'électricité. Dans notre cas, les contraintes d'inégalités de la formulation de l'OPF sont respectées par 1) l'encodage des solutions candidates dans le cas de puissance active des générateurs, de la puissance réactive des SVAR et du rapport des transformateurs, 2) par l'analyse de l'écoulement de puissance dans le cas de la puissance réactive des générateurs ou 3) par la fonction d'aptitude dans le cas de la tension aux bus, de la puissance active au générateur de référence et de la puissance d'écoulement des branches. Pour illustrer ceci, nous présentons aux Figures 7.8 et 7.9 les tensions aux bus et les puissances réactives des générateurs associés à la solution finale calculée par notre algorithme. Dans tous les cas, il est évident que les limites permises sont respectées ce qui démontre la faisabilité de la solution trouvée et par conséquent, l'avantage de l'algorithme proposé comparé aux plusieurs des métaheuristiques publiées dans la littérature.

TABLEAU 7.7
COMPARAISON DE LA QUALITÉ DES SOLUTIONS CALCULÉES PAR DIFFÉRENTS ALGORITHMES POUR LE RÉSEAU TEST IEEE 118-BUS

Méthodes	Années de publication	Variables de contrôle	Résultats publiés		Violation de la contrainte sur Q
			Coûts ($/h)	Pertes (MW)	
Cas de base [136]	-	-	131,220.63	132.790	Q_{19}, Q_{32}, Q_{34}, Q_{92}, Q_{103}, Q_{105}
Algorithme génétique[2] [168]	2014	P, |V|, T	132,746.35	80.555	Q_1, Q_6, Q_{12}, Q_{15}, Q_{19}, Q_{25}, Q_{32}, Q_{34}, Q_{36}, …
Essaim de particules[2] [168]	2014	P, |V|, T, Q_C	129,756.22	77.609	Q_1, Q_{19}, Q_{56}, Q_{65}, Q_{70}, Q_{74}, Q_{76}, Q_{92}, …
Recherche différentielle[2] [168]	2014	P, |V|, T, Q_C	129,691.61	77.949	Q_1, Q_{25}, Q_{65}, Q_{66}, Q_{74}, Q_{76}, Q_{85}, Q_{92}
Méthode enseignement / apprentissage[2] [168]	2014	P, |V|, T, Q_C	129,682.84	76.192	Q_1, Q_{25}, Q_{65}, Q_{66}, Q_{74}, Q_{76}, Q_{85}, Q_{92}
Recherche gravitationnelle[2] [269]	2012	P, |V|, T, Q_C	129,565.03	76.190	Q_1, Q_{19}, Q_{25}, Q_{65}, Q_{66} Q_{74}, Q_{76}, Q_{92}, …
Métaheuristique hybride[2] [270]	2013	P, |V|, T, Q_C	130,288.21	84.006	Ne peut être vérifiée
Programmation non-linéaire mixte[1] [271]	2012	P, |V|, T, Q_C	130,114.43	77.848	Ne peut être vérifiée
Relaxation SDP[1] [16]	2015	P, |V|	129,668.60	-	Ne peut être vérifiée
Point intérieur[1] [272]	2010	P, |V|	129,720.70	-	Ne peut être vérifiée
Point intérieur[1] [136]	2011	P, |V|	129,660.69	77.401	Respectés
PSO-GPU proposé	2015	P, |V|, T, Q_C	129,627.03	76.984	Respectés

[1] méthode déterministe
[2] méthode non déterministe

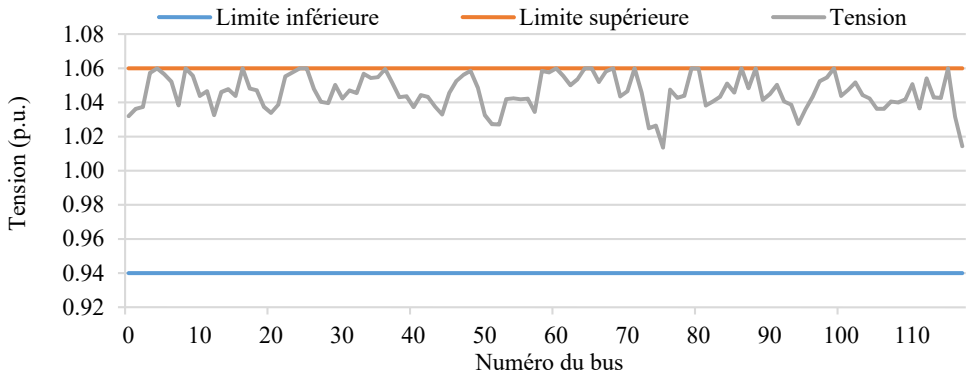


Figure 7.8: Tensions aux bus pour la solution finale calculée par le PSO sur GPU pour le réseau test IEEE 118-bus

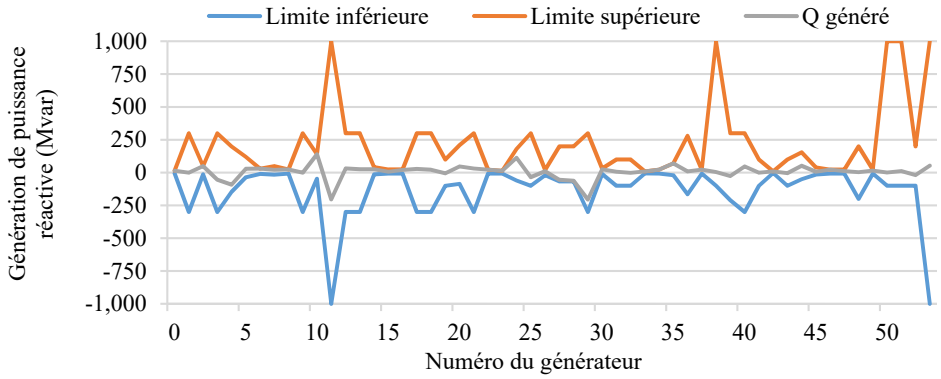


Figure 7.9: Puissance réactive aux générateurs pour la solution finale calculée par le PSO sur GPU pour le réseau test IEEE 118-bus

7.3.4.3 Réseau de 300 bus

Le dernier réseau considéré dans cette section est le réseau test IEEE à 300 bus dont les données sont disponibles à la référence [136]. Il contient 69 générateurs, 62 transformateurs et 14 SVAR pour un total de 213 variables de contrôle. Pour optimiser ce réseau, nous exécutons le PSO en trois phases de 1000 itérations. Toutefois, étant donné le grand nombre de variables de contrôle à optimiser, l'initialisation des solutions candidates à la première itération n'utilise pas une distribution uniforme, mais plutôt une distribution normale centrée sur le cas de base avec une déviation standard de 25% de l'intervalle possible pour chaque variable. Cette approche limite la région de recherche initiale et permet d'obtenir plus rapidement des solutions candidates faisables, améliorant ainsi la convergence de la méthode. Le vecteur de solution trouvé pour la minimisation des coûts de production est reproduit à l'Appendice B et disponible pour téléchargement en format MATPOWER sur le site internet à la référence [263]. Nous présentons aussi au Tableau 7.8 les coûts de production et les pertes de transport associés aux solutions calculées par notre algorithme et celles calculées par d'autres méthodes. Étant donné la complexité du problème considéré, nous avons trouvé une seule référence qui utilise une métaheuristique et la même description du réseau que nous. Il s'agit de la méthode en trois phases développée par Mahdad et Srairi [169] que nous avons présentée précédemment au Chapitre 2. Contrairement à notre approche, cette méthode utilise une métaheuristique différente à chaque phase. Ceci augmente la complexité de l'algorithme, mais n'accroît pas nécessairement son efficacité comme le montrent nos résultats. Pour rendre la comparaison plus complète, nous incluons au Tableau 7.8 les résultats pour la minimisation des coûts de production, mais aussi ceux pour la minimisation des pertes de transport. Dans les deux cas, l'algorithme parallèle que nous proposons trouve des solutions de meilleure qualité que toutes les méthodes déterministes et non déterministes citées, incluant la métaheuristique à trois phases de

Mahdad et Srairi [169]. De plus, comme nous l'avons expliqué à la section précédente, notre approche garantit le respect des contraintes d'inégalités. Ceci est visible aux Figures 7.10 et 7.11 où l'on remarque bien que les tensions aux bus et les puissances réactives des générateurs associés à la solution finale sont toutes à l'intérieur des limites permises.

TABLEAU 7.8
COMPARAISON DE LA QUALITÉ DES SOLUTIONS CALCULÉES PAR DIFFÉRENTS ALGORITHMES POUR LE RÉSEAU TEST IEEE 300-BUS

Méthodes	Années de publication	Variables de contrôle	Résultats publiés	
			Coûts ($/h)	Pertes (MW)
Minimisation des coûts de production :				
Cas de base [136]	-	-	724 466	408.316
Relaxation semi-définie [1] [16]	2015	P, |V|	720 031	-
Relaxation basée sur le moment [1] [264]	2015	P, |V|	720 000	-
Point intérieur[1] [136]	2011	P, |V|	719 725	302.776
Métaheuristiques en trois phase[2] [169]	2014	P, |V|, T, Q_C	719 721	302.720
PSO-GPU proposé[2]	2015	P, |V|, T, Q_C	719 443	298.580
Minimisation des pertes de transport :				
Cas de base [136]	-	-	724 466	408.316
Métaheuristiques trois phase[2] [169]	2014	P, |V|, T, Q_C	757 242	210.678
Point intérieur[1] [136]	2011	P, |V|	757 715	210.604
PSO-GPU proposé[2]	2015	P, |V|, T, Q_C	757 168	204.305

[1] méthode déterministe
[2] méthode non déterministe

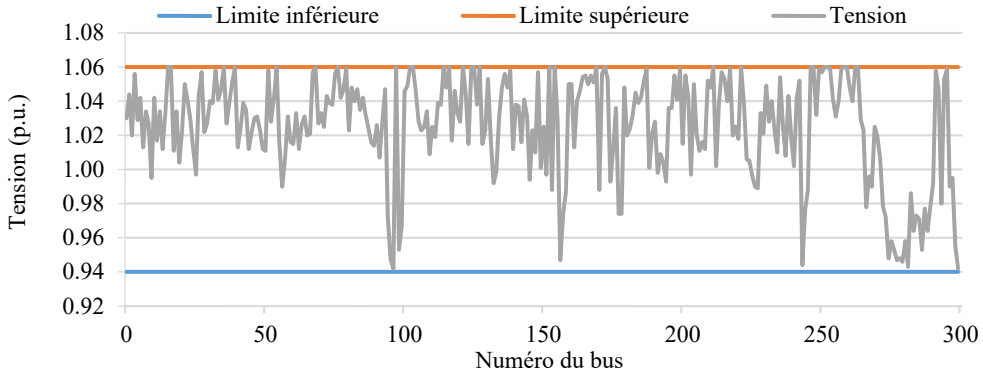


Figure 7.10: Tensions aux bus pour la solution finale calculée par le PSO sur GPU pour le réseau test IEEE 300-bus

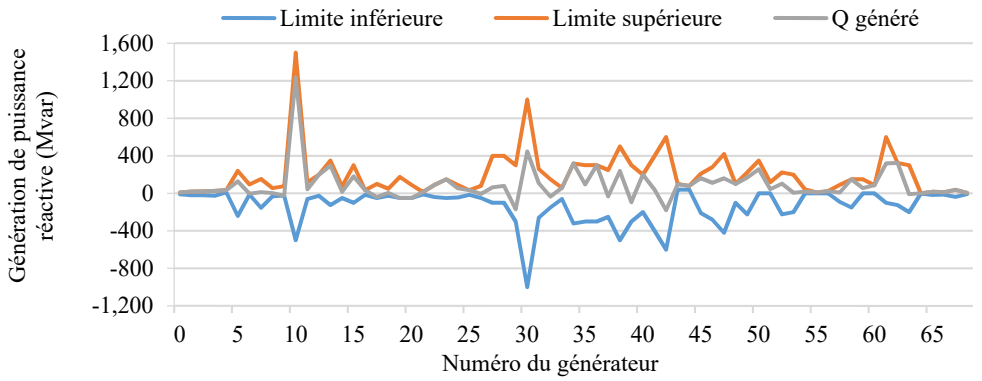


Figure 7.11: Puissance réactive aux générateurs pour la solution finale calculée par le PSO sur GPU pour le réseau test IEEE 300-bus

L'efficacité de l'approche proposée est due à l'exécution en trois phases, à la capacité du PSO à considérer les variables discrètes et au très grand nombre de solutions candidates permis par la parallélisation sur GPU. Pour illustrer l'avantage de notre approche à multiples phases, nous montrons à la Figure 7.12 l'aptitude moyenne et le coût de production moyen des solutions candidates à chaque itération du PSO. On peut facilement noter les pics sur les deux courbes aux itérations 1000 et 2000 lorsque les solutions candidates sont réinitialisées aléatoirement suivant une distribution normale centrée sur la meilleure solution trouvée à la phase précédente. Facilement visible sur la courbe illustrant les coûts de production, on remarque que cette réinitialisation permet au PSO d'échapper à un minimum local et de continuer sa recherche afin d'obtenir une solution finale de meilleure qualité.

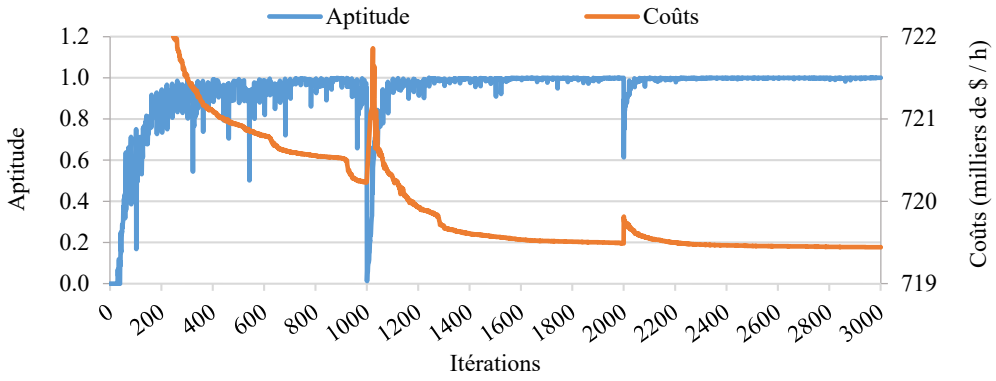


Figure 7.12: Aptitude moyenne et coût de production moyen des solutions candidates en fonction de l'itération du PSO sur GPU pour le réseau test IEEE 300-bus

7.3.4.4 *Temps d'exécution et accélération*

Pour démontrer l'avantage de notre implémentation parallèle du PSO sur GPU, nous exécutons l'algorithme à 20 reprises pour l'optimisation des réseaux tests IEEE à 30, 118 et 300 bus et listons au Tableau 7.9 les temps d'exécution moyens ainsi que les accélérations calculées. Les temps d'exécution pour l'implémentation séquentielle varient de 3 min 33 s pour le réseau à 30 bus jusqu'à 10 h 21 min pour le réseau à 300 bus. De son côté, l'implémentation parallèle réduit ces temps à 13.1 s et 38 min respectivement, ce qui représentent des accélérations de 16.2x et 17.2x. Le gain de performance permis par la parallélisation sur GPU est indéniable et essentiel à l'optimisation de réseaux de transport d'électricité à grande échelle tels que le réseau test IEEE à 300 bus. Finalement, pour montrer la fiabilité du solveur parallèle proposé, nous incluons aussi au Tableau 7.9 les moyennes et les écarts types des coûts de production et des pertes de transport associés aux solutions calculées pour les 20 essais. Les valeurs obtenues confirment que les résultats listés précédemment au Tableau 7.8 n'ont pas été sélectionnés à la main, mais représentent bien l'efficacité et la fiabilité du solveur parallèle proposé.

TABLEAU 7.9
TEMPS D'EXÉCUTION ET QUALITÉ DE LA SOLUTION FINALE POUR LE PSO PARALLÈLE SUR GPU (MOYENNES DE 20 ESSAIS)

Réseau	Temps d'exécution						Accélération		Qualité de la solution finale			
	CPU		OMP		GPU		OMP	GPU	Coûts ($/h)		Pertes (MW)	
	Moy.	É.T.	Moy.	É.T.	Moy.	É.T.			Moy.	É.T.	Moy.	É.T.
30	212.8	5.8	12.0	0.4	13.1	0.4	17.8x	16.2x	799.0	0.01	2.84	0.0
118	5653.3	308.2	397.6	23.7	328.1	17.2	14.2x	17.2x	129,631.7	6.9	9.83	0.1
300	37,229.5	2076.0	2541.8	146.9	2295.4	124.1	14.6x	16.2x	719,508.3	57.1	206.87	0.9

7.4 Conclusion

Dans ce chapitre, nous avons utilisé le cadriciel *gpuMF* pour implémenter un PSO parallèle sur GPU pour l'optimisation de l'écoulement de puissance d'un réseau de transport d'électricité. L'algorithme proposé calcule la puissance active des générateurs, la tension des générateurs, le rapport des transformateurs et la puissance réactive des compensateurs statiques afin de minimiser les coûts de production, les pertes de transport ou les émissions polluantes du réseau de régime permanent. La qualité des solutions candidates est évaluée à chaque itération de l'algorithme à l'aide d'une analyse de l'écoulement de puissance. Pour ce faire, nous avons développé des implémentations parallèles sur GPU des algorithmes de Gauss-Seidel et de Newton-Raphson. Nos implémentations utilisent des matrices creuses afin de réduire l'ordre

de complexité des calculs et respectent les limites sur la puissance réactive des générateurs. Les modules d'analyse développés sont testés sur des réseaux de différentes grosseurs allant jusqu'à 2383 bus. Les résultats sont comparés à ceux obtenus par l'outil MATPOWER afin de valider l'exactitude de nos implémentations. Les accélérations mesurées sont de 45.2x dans le cas de l'algorithme de G-S et de 17.8x dans le cas de celui de N-R. Malgré que la méthode de G-S exploite mieux l'architecture parallèle du GPU, c'est la méthode de N-R qui est la plus performante puisqu'elle nécessite un nombre d'itérations beaucoup plus petit avant de converger. Le PSO développé utilise donc la méthode de N-R pour obtenir l'état complet des réseaux associés aux solutions candidates. La qualité de ces solutions est évaluée à l'aide d'une fonction d'aptitude qui intègre l'objectif d'optimisation ainsi que les contraintes d'égalités et d'inégalités. La stratégie d'optimisation proposée exécute la métaheuristique sur plusieurs phases afin d'éviter une convergence prématurée et de stimuler davantage l'exploration de l'espace de recherche. L'algorithme développé est testé sur les réseaux tests IEEE à 30, 118 et 300 bus. Les solutions calculées sont supérieures à celles obtenues par les méthodes antérieures publiées dans la littérature. De plus, la parallélisation sur GPU permet d'accélérer l'optimisation du réseau par un facteur de 17.2x comparé à une implémentation séquentielle sur CPU.

Les travaux de recherche présentés dans ce chapitre représentent trois contributions importantes. Premièrement, il s'agit de la première fois qu'une métaheuristique parallèle sur GPU est proposée pour le problème d'OPF. Contrairement aux méthodes déterministes, les métaheuristiques permettent une optimisation globale et considèrent nativement les variables discrètes telles que le rapport des transformateurs et le réglage des compensateurs statiques. Par contre, les métaheuristiques demandent une puissance de calcul considérable ce qui résulte souvent en un temps d'exécution trop long pour une application en ligne. En exploitant l'architecture parallèle des GPU, notre implémentation réduit significativement le temps d'exécution et permet un contrôle plus réactif. Deuxièmement, la stratégie d'optimisation proposée est aussi une contribution importante puisqu'elle permet d'obtenir des solutions de meilleure qualité que les méthodes antérieures tout en respectant les contraintes telles que la puissance réactive aux générateurs. La stratégie que nous avons développée inclut l'application du PSO, l'encodage des solutions candidates, la fonction d'aptitude et l'approche d'optimisation en multiples phases. Finalement, nos implémentations parallèles des algorithmes de G-S et de N-R représentent une troisième contribution majeure. Contrairement aux tentatives antérieures, nos implémentations utilisent des matrices creuses et parallélisent toutes les étapes des algorithmes. L'utilisation de matrices creuses n'est pas triviale, mais essentielle au développement d'une méthode performante puisqu'elle réduit l'ordre de complexité des calculs. Pour conclure, dans le contexte de l'optimisation de l'écoulement de puissance, la parallélisation de métaheuristiques sur GPU permet d'optimiser des réseaux de transport d'électricité plus grands, dans des temps de calcul

plus courts tout en obtenant des solutions de meilleure qualité. Dans un contexte plus large, nous avons démontré par cette deuxième application que le développement de métaheuristiques parallèles sur GPU représente une contribution significative dans le domaine des réseaux intelligents.

Chapitre 8
Reconfiguration optimale des réseaux de distribution

Les travaux de recherche complétés dans le cadre de ce chapitre ont été publiés dans l'article suivant :

V. Roberge, M. Tarbouchi, et F. Okou, "*Distribution System Optimization on Graphics Processing Unit,"* IEEE Transactions on Smart Grid, IEEE Transactions on Smart Grid, vol. PP, no. 99, pp. 1–10, 2015.

8.1 Introduction

La troisième et dernière application considérée dans cette thèse est la reconfiguration des réseaux de distribution électrique (DFR du terme anglais *Distribution Feeder Reconfiguration*). Le réseau de distribution représente le dernier stage de l'acheminement de l'électricité de la centrale de production vers le client. Pour minimiser l'infrastructure requise et faciliter la coordination des systèmes de sécurité des postes électriques, ces réseaux suivent habituellement une structure radiale où chaque bus est alimenté par une seule branche. Cependant, des interrupteurs d'interconnexion normalement ouverts sont installés de façon à relier certains bus pour permettre de transférer des charges en cas de surcharge ou de panne. Des interrupteurs de segmentation normalement fermés sont aussi installés sur la plupart des branches du réseau afin d'ouvrir un segment pour isoler une faute ou effectuer des réparations. Un exemple d'un réseau de distribution de 16 bus se trouve à la Figure 8.1. Sur ce diagramme, les bus connectés aux charges sont représentés par des cercles, les interrupteurs d'interconnexion par des lignes pointillées et les interrupteurs de segmentation par des lignes pleines. Ce réseau est électrifié par trois bus d'alimentation.

Historiquement, les interrupteurs d'un réseau de distribution sont opérés manuellement et la topologie radiale du système est rarement changée. Lorsque les charges varient au cours de l'année ou même d'une journée, cette topologie n'est plus nécessairement optimale et entraîne des pertes supplémentaires dans les lignes de distribution. Les véhicules électriques [273] ainsi que les sources distribuées d'énergies renouvelables [274] sont des facteurs nouveaux qui aggravent cette

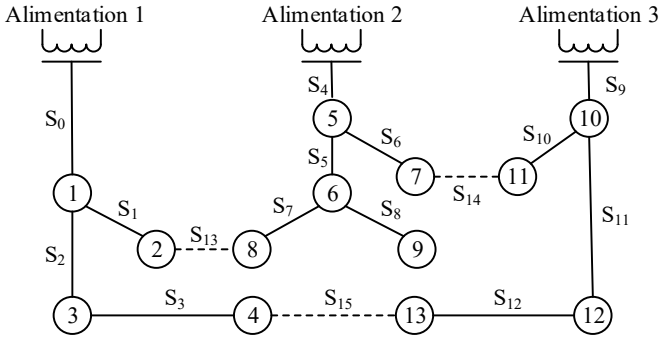


Figure 8.1: Réseau de distribution de 16 bus

variation de la demande. Heureusement, avec l'implémentation des réseaux électriques intelligents et le développement de commutateurs haute tension ultrarapides, il est possible d'automatiser cette reconfiguration pour réagir aux fluctuations de la demande en énergie et toujours opérer le réseau dans sa topologie optimale, minimisant ainsi les pertes de transport et les coûts d'opération. Cette reconfiguration automatique permet aussi de modifier rapidement la structure du réseau suite à un bris afin de minimiser les pannes électriques. Toutefois, la reconfiguration automatique nécessite le développement d'algorithmes d'optimisation hautement efficaces pouvant calculer la topologie optimale du réseau de distribution électrique dans les plus brefs délais.

Comme nous l'avons discuté au Chapitre 2, les solutions antérieures au problème de DFR ont plusieurs lacunes importantes. Dans le cas des heuristiques, elles sont limitées à une optimisation locale. Dans le cas des méthodes conventionnelles, elles sont moins précises puisqu'elle nécessite une formulation simplifiée du problème original. De plus, leur temps d'exécution peut être considérable dépendamment de la grandeur du réseau. Finalement, dans le cas des métaheuristiques, la contrainte sur la topologie radiale du réseau représente une difficulté importante, complique l'implémentation et réduit significativement la capacité d'exploration de l'algorithme. Les métaheuristiques souffrent elles aussi d'un trop long temps d'exécution.

Dans ce chapitre, nous proposons d'utiliser le cadriciel *gpuMF* pour paralléliser la reconfiguration des réseaux de distribution afin de minimiser les pertes de transport. La méthode proposée implémente un algorithme génétique (GA de l'anglais *genetic algorithm*) et exploite pleinement l'architecture parallèle du GPU permettant une accélération de 56.8x comparée à une exécution séquentielle sur CPU. Le programme final est extrêmement rapide et capable de réagir aux perturbations du réseau en calculant une nouvelle configuration optimale dans les plus brefs délais. De plus, une technique innovatrice basée sur la théorie des graphes est proposée pour encoder les solutions candidates. L'approche fait recours au calcul de l'arbre couvrant de poids

minimal et garantit la topologie radiale du réseau final. L'algorithme est testé sur plusieurs réseaux de distribution dont la grandeur varie de 16 à 4400 bus. Dans tous les cas, la qualité de la solution finale calculée par l'algorithme proposé est égale ou meilleure à celle obtenue par d'autres techniques récentes publiées dans la littérature. De plus, grâce à la parallélisation sur GPU, le temps de calcul de l'algorithme proposé est beaucoup plus court que toutes les autres méthodes.

Le reste de ce chapitre est organisé comme suit. Tout d'abord, le problème de DFR est défini à la section 8.2. La stratégie d'optimisation incluant l'encodage des solutions candidates est expliquée à la section 8.3. La section 8.4 traite de la parallélisation sur GPU et finalement, la section 8.5 présente les résultats expérimentaux qui démontrent clairement l'avantage de la solution parallèle proposée.

8.2 Définition du problème

Le problème de réduction des pertes par la reconfiguration du réseau de distribution a été défini pour la première fois en 1975 par Merlin et Back [275]. Il s'agit d'un problème d'optimisation combinatoire à grande échelle, non convexe et non linéaire. Le problème consiste à trouver la topologie radiale du réseau qui :

$$\text{minimise} \quad L(\bar{x}) = \sum_{l=1}^{N_{branches}} R_l \cdot |I_l|^2 \tag{8.1}$$

$$\text{sujet à} \quad |V_i|_{min} \leq |V_i| \leq |V_i|_{max} \quad \text{pour } i = 1 \text{ à } N_{bus} \tag{8.2}$$

$$|S_l| \leq |S_{l\,max}| \quad \text{pour } l = 1 \text{ à } N_{branches} \tag{8.3}$$

$$G(\bar{x}) \text{ est un graphe connexe} \tag{8.4}$$

$$N_{branches} = N_{bus} - 1 \tag{8.5}$$

où i est l'indice d'un bus et l est l'indice d'une la branche. L sont les pertes actives totales, $\bar{x}$ est le vecteur représentant la configuration du réseau ou l'état (ouvert/fermé) des interrupteurs d'interconnexion et de segmentation, R_l est la résistance de la branche l et I_l est le courant dans la branche l. Dans les équations de contraintes, $|V_i|$ est l'amplitude de la tension au bus i et $|S_l|$ est la puissance apparente dans la branche l. $|V_i|_{min}$ et $|V_i|_{max}$ sont les limites de la tension au bus et $|S_{l\,max}|$ est la capacité maximale de la branche l. Finalement, la contrainte sur l'arborescence du réseau est exprimée en spécifiant que la topologie définie par le vecteur $\bar{x}$ doit former un graphe $G(\bar{x})$ connexe et que le nombre de branches fermées doit être égal à $N_{bus} - 1$. Un graphe G est connexe si deux sommets quelconques peuvent toujours être reliés par une suite d'arêtes. Dans un système électrique, le réseau est connexe si tous les

bus sont énergisés. Quant à elle, la contrainte sur le nombre de branches fermées évite la formation de boucle. Un réseau connexe sans boucle a nécessairement une topologie radiale.

8.3 Stratégie d'optimisation

8.3.1 Algorithme génétique

L'algorithme génétique est une méthode d'optimisation non déterministe inspirée de l'évolution naturelle des espèces. Comme nous l'avons expliqué au Chapitre 3, le GA utilise des opérations de sélection, d'enjambement et de mutation pour améliorer une population de solutions candidates imitant la façon dont les organismes vivants évoluent et s'adaptent à leur environnement. Dans le GA, les solutions candidates sont modifiées pour mieux les adapter à la fonction d'aptitude qui intègre les objectifs d'optimisation et les contraintes du problème considéré. Les opérations de sélection et d'enjambement encouragent la convergence de l'algorithme tandis que l'opération de mutation stimule l'exploration de l'espace de recherche. Dans la version originale [210], le GA utilise un encodage binaire et l'enjambement s'effectue en combinant les éléments de deux solutions sélectionnées pour en créer une nouvelle. De nos jours, le GA supporte aussi un encodage réel et une multitude d'opérations adaptées aux problèmes étudiés. Grâce à l'opération d'enjambement, le GA s'applique bien aux problèmes d'optimisation combinatoire [276], [277], [278] et [279]. Spécifiquement pour la reconfiguration des réseaux de distribution, les auteurs de [280] ont observé que le GA nécessite moins d'itérations que l'algorithme d'optimisation par essaim de particules (PSO de l'anglais *particle swarm optimization*) avant de converger vers une solution finale. Dans le but de minimiser les temps de calcul, nous choisissons donc le GA comme moteur d'optimisation pour le développement de notre logiciel de DFR.

Le pseudocode du GA a été donné au Chapitre 3. Dans notre implémentation, les solutions sont encodées par des vecteurs de nombres réels entre 0 et 1. Elles sont initialisées aléatoirement suivant une distribution uniforme. Le processus itératif débute ensuite par l'évaluation de l'aptitude des solutions candidates. Chaque solution parent est sélectionnée par un tournoi [56] entre trois solutions choisies aléatoirement. Chaque couple de solutions parents produit deux solutions enfants suivant la méthode d'enjambement uniforme [56]. Chaque solution enfant peut ensuite être retenue avec une probabilité 0.1 pour une mutation. Lorsqu'une solution enfant est retenue, ses éléments sont modifiés avec une probabilité de 0.2 par une mutation aléatoire uniforme [56]. Cette opération déplace aléatoirement l'élément dans un intervalle de 0.25. Les opérations d'enjambement uniforme et de mutation aléatoire uniforme sont illustrées à la Figure 8.2. Finalement, après que les mutations soient complétées, les

solutions parents sont remplacées par les solutions enfants suivant la technique d'élitisme [56] qui permet de transférer les meilleures solutions parents à la génération suivante afin d'améliorer la convergence de l'algorithme.

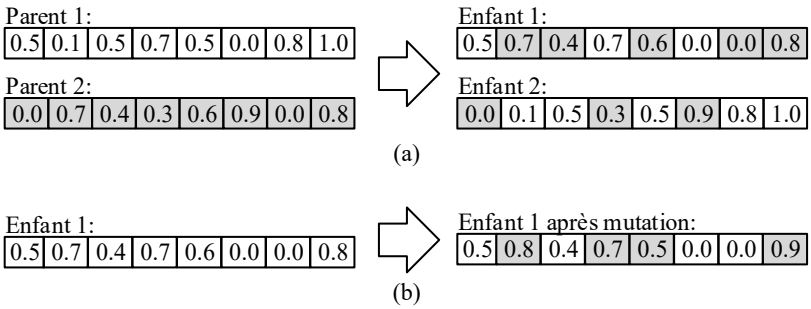


Figure 8.2: Opérations (a) d'enjambement uniforme et (b) de mutation aléatoire uniforme

8.3.2 Encodage des solutions candidates

Comme nous l'avons vu au Chapitre 2, la plupart des travaux antérieurs sur le développement de métaheuristiques pour le problème de DFR encodent les solutions candidates en utilisant (a) un vecteur binaire représentant l'état (ouvert ou fermé) de chaque branche du réseau ou (b) un vecteur de nombres entiers identifiant les indices des branches ouvertes [185]. Dans les deux cas, maintenir la topologie radiale du réseau est un défi important.

Dans cette thèse, nous proposons un encodage unique où les solutions candidates sont représentées par des vecteurs de nombres réels entre 0 à 1 et dont la longueur est égale au nombre de branches dans le réseau. Un exemple de solution aléatoire pour le réseau de distribution à 16 bus discuté précédemment est illustré à la Figure 8.3. Pour décoder cette solution, les bus d'alimentation sont regroupés pour former un nœud racine et le réseau est représenté par un graphe non orienté. Les éléments du vecteur de solution sont assignés aux branches du graphe. Ils représentent les poids des branches et permettent d'obtenir le graphe pondéré non orienté montré à la Figure 8.4. L'algorithme de Borůvka est ensuite utilisé pour calculer l'arbre couvrant de poids minimal illustré à la Figure 8.5. Cet algorithme permet d'obtenir une topologie radiale qui connecte tous les bus du réseau aux bus d'alimentation. Finalement, l'état ouvert ou fermé de chaque branche est sauvegardé dans le vecteur de solution décodé montré à la Figure 8.6.

L'encodage que nous proposons est une contribution importante. Il est entièrement nouveau, il est simple et il garantit la topologie radiale des solutions candidates sans faire recours à des opérateurs complexes qui limitent la

métaheuristique à une optimisation locale. De plus, comme nous le verrons à la section 8.4, la méthode proposée s'adapte bien à l'architecture parallèle du GPU. Nous verrons aussi à la section 8.5 que l'encodage proposé améliore grandement l'efficacité du GA et permet d'optimiser des systèmes beaucoup plus grands que les méthodes antérieures.

N° branche	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Solution	0.5	0.1	0.5	0.7	0.5	0.0	0.8	1.0	0.0	0.7	0.4	0.3	0.6	0.9	0.0	0.8

Figure 8.3: Vecteur de solution encodée

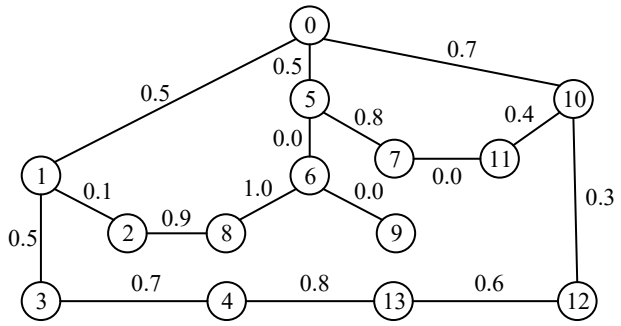


Figure 8.4: Graphe pondéré non orienté pour le réseau de 16 bus

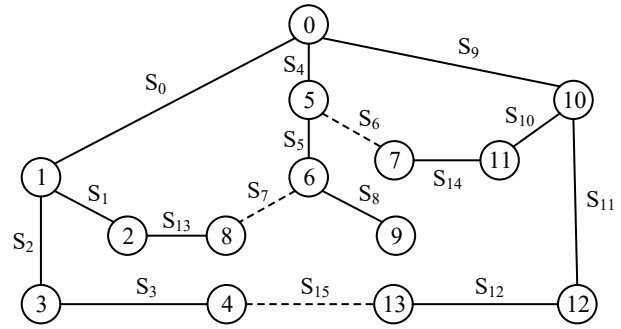


Figure 8.5: Arbre couvrant de poids minimal associé à la solution candidate

N° branche	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Solution	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	0

Figure 8.6: Vecteur de solution décodée

8.3.3 Analyse de l'écoulement de puissance

Une fois que les solutions candidates sont décodées, une analyse de PF utilisant la méthode régressive-progressive (B-F de l'anglais *backward-forward*) [281] est exécutée pour calculer les tensions complexes aux bus. L'algorithme B-F est la méthode de choix pour cette application puisqu'il est conçu pour l'analyse de réseaux radiaux où le rapport R/X est généralement élevé causant la divergence d'autres méthodes telles que celles de Newton-Raphson (N-R) ou de Gauss-Seidel (G-S) [282]. De plus, parce que l'algorithme B-F considère uniquement des réseaux radiaux, les calculs impliqués sont plus simples que pour la méthode de N-R et le nombre d'itérations requis est plus petit que pour celle de G-S. Enfin, des travaux antérieurs ont montré que l'implémentation parallèle sur GPU de la méthode B-F permet une accélération plus élevée et un temps d'exécution plus petit que des implémentations parallèles sur GPU des algorithmes de N-R ou de G-S [132], [135].

L'algorithme B-F fonctionne en parcourant les bus du réseau niveau par niveau, du bas vers le haut et du haut vers le bas, sur plusieurs itérations pour mettre à jour les valeurs des tensions aux bus. Ce balayage régressif-progressif nécessite que les nœuds du réseau soient organisés en niveaux comme à la Figure 8.7. Le niveau d'un bus est calculé par le nombre de branches parcourues pour relier le bus au sommet du graphe. Dans notre implémentation, nous utilisons une opération de parcours de graphe pour calculer le niveau de chaque nœud. Nous construisons ensuite une matrice de connexion qui relie les niveaux du réseau à leurs bus respectifs. Cette matrice permet à la méthode B-F de parcourir les bus du réseau, niveau par niveau.

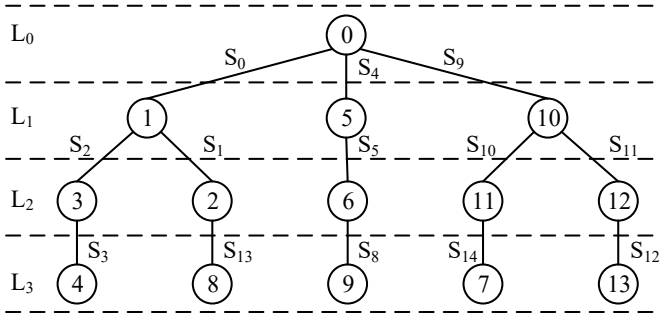


Figure 8.7: Arbre couvrant de poids minimal organisé en niveaux

Chaque itération t de l'algorithme B-F exécute comme suit :

1) **Courant injecté aux bus :** pour tous les bus i du réseau, calculer le courant $I^t_{bus\ i}$ injecté au bus en utilisant :

$$I^t_{bus\ i} = \left(\frac{S_i}{V_i^{t-1}}\right)^* - Y_i\ V_i^{t-1} \tag{8.6}$$

où i est l'indice du bus, t est l'indice de l'itération, S_i est la puissance demandée, V_i^{t-1} est la tension calculée à l'itération précédente, Y_i est l'admittance de shunt au bus, et l'opérateur « * » dénote le conjugué complexe.

2) **Balayage régressif:** en commençant au niveau le plus bas et en remontant jusqu'au niveau 1, calculer les courants $I^t_{br\ i,j}$ dans la branche connectant chaque bus i à son bus parent j en utilisant :

$$I^t_{br\ i,j} = -I^t_{bus\ i} + \sum_{bus\ k\ \in\ \{enfants\ de\ i\}} I^t_{br\ i,k} \tag{8.7}$$

où $I^t_{br\ i,k}$ est le courant dans la branche connectant le bus i à son enfant k.

3) **Balayage progressif:** en commençant au niveau 1 et en descendent jusqu'au niveau le plus bas, calculer les tensions V_i^t au bus i en utilisant :

$$V_i^t = V_j^{t-1} + Z_{br\ i,j}\ I^t_{br\ i,j} \tag{8.8}$$

où $Z_{br\ i,j}$ est l'impédance de la branche connectant le bus i à son parent j.

4) **Puissance injectée au bus:** pour tous les bus i du réseau, calculer la puissance S_i^t injectée au bus en utilisant :

$$S_i^t = V_i^t \left(\sum_{bus\ k\ \in\ \{enfants\ de\ i\}} Y^*_{br\ i,k}\ V_k^{t*} \right) \tag{8.9}$$

où $Y_{br\ i,k}$ est la susceptance de la branche connectant le bus i à son enfant k et où « * » dénote encore ici le conjugué complexe.

Ce processus itératif continue jusqu'à ce que l'erreur maximale entre les valeurs calculées et actuelles des puissances injectées aux bus soit plus petite que la tolérance spécifiée.

8.3.4 Fonction d'aptitude

Après que l'analyse de PF ait été exécutée pour chacune des solutions candidates, la fonction d'aptitude peut être évaluée. Celle-ci intègre l'objectif d'optimisation et

les contraintes du problème. Elle permet au GA de mesurer la qualité des solutions candidates avant de sélectionner les parents utilisés pour l'enjambement. Pour évaluer l'aptitude $\mathcal{F}(\bar{x})$ d'une solution, il faut d'abord calculer la fonction objective $f(\bar{x})$ à l'aide de l'équation (8.1) et normaliser la valeur obtenue entre 0 et 1 comme suit de façon à ce qu'une valeur près de 0 représente une très mauvaise solution tandis qu'une valeur près de 1 en représente une très bonne :

$$f_{NORM}(\bar{x}) = \frac{1}{1 + f(\bar{x})} \tag{8.10}$$

Deuxièmement, un facteur de violation est calculé afin de considérer les contraintes d'inégalités aux équations (8.2) et (8.3). Dans le cas des limites sur les tensions aux bus, chaque bus est vérifié et tout excès est quantifié comme suit :

$$E(|V_i|) = \begin{cases} \dfrac{|V_i| - |V_i|_{max}}{|V_i|_{max} - |V_i|_{min}}, & |V_i| > |V_i|_{max} \\ \dfrac{|V_i|_{min} - |V_i|}{|V_i|_{max} - |V_i|_{min}}, & |V_i| < |V_i|_{min} \\ 0, & |V_i|_{min} \leq |V_i| \leq |V_i|_{max} \end{cases} \tag{8.11}$$

où $E(|V_i|)$ est l'excès relatif de la tension au bus i. En divisant la différence absolue par l'intervalle des valeurs permises, nous obtenons une valeur relative pour l'excès de tension. Ceci permet une comparaison plus juste lorsque les limites des tensions varient d'un bus à l'autre. Une approche similaire est utilisée pour calculer l'excès relatif $E(S_i)$ sur les puissances permises dans les branches d'après leur capacité électrique. Les valeurs calculées pour chaque bus et chaque branche sont ensuite additionnées pour obtenir l'excès total $E_{total}(\bar{x})$ pour le réseau en entier:

$$E_{total}(\bar{x}) = \sum_{i=1}^{N_{bus}} E(|V_i|) + \sum_{l=1}^{N_{branch}} E(|S_l|) \tag{8.12}$$

Comme pour la fonction objective, la somme calculée est normalisée entre 0 et 1 pour obtenir le facteur de violation normalisé $\mathcal{V}_{NORM}(\bar{x})$:

$$\mathcal{V}_{NORM}(\bar{x}) = \frac{1}{1 + E_{total}(\bar{x})} \tag{8.13}$$

Troisièmement, les valeurs normalisées de la fonction objective et du facteur de violation sont regroupées comme suit afin d'obtenir l'aptitude $\mathcal{F}(\bar{x})$ de la solution candidate :

$$\mathcal{F}(\bar{x}) = \begin{cases} 1 + f_{NORM}(\bar{x}), & \mathcal{V}_{NORM}(\bar{x}) = 1 \\ \mathcal{V}_{NORM}(\bar{x}), & \mathcal{V}_{NORM}(\bar{x}) < 1 \end{cases} \tag{8.14}$$

Tout comme au chapitre précédent, cette fonction d'aptitude permet de séparer clairement les solutions faisables de celles qui ne le sont pas simplement à partir des valeurs numériques calculées. En effet, les solutions infaisables, soient celles qui enfreignent les contraintes, ont toutes des aptitudes entre 0 et 1. De leur côté, les solutions faisables, celles qui respectent toutes les contraintes, ont des aptitudes entre 1 et 2. De plus lorsque deux solutions infaisables sont comparées, celle qui a la plus grande valeur d'aptitude est de meilleure qualité puisqu'elle enfreint les contraintes physiques du réseau à un degré moindre que l'autre. Pareillement, lorsque deux solutions faisables sont comparées, celle qui a la plus grande valeur d'aptitude est de meilleure qualité puisqu'elle minimise davantage les pertes de distribution.

8.3.5 Approche à multiple phases

Similairement à l'algorithme d'OPF présenté au chapitre précèdent, notre implémentation du GA pour le problème de DFR utilise une optimisation en multiples phases afin d'échapper aux optima locaux et d'améliorer l'exploration de l'espace de recherche. À la fin de chaque phase, quelques-unes des meilleures solutions sont gardées tandis que les autres sont réinitialisées aléatoirement avant d'entreprendre la phase suivante. Le nombre de solutions gardées, le nombre de phases et le nombre d'itérations par phase sont des paramètres de configuration que nous choisissons expérimentalement d'après la complexité du réseau considéré. Les valeurs choisies sont listées dans la section des travaux expérimentaux au Tableau 8.3. Contrairement à la stratégie utilisée précédemment pour l'OPF, l'initialisation aléatoire se fait ici suivant une distribution uniforme à toutes les phases de l'algorithme afin d'explorer l'étendue complète de l'espace de recherche. Dans le cas de l'OPF, la région des solutions faisables était beaucoup plus petite et une distribution normale centrée sur la meilleure solution devait être utilisée pour rester près de la région faisable. Dans le cas du DFR, même si toutes les solutions initiales sont infaisables, un cas commun pour les très grands réseaux, la métaheuristique réussit rapidement à améliorer les solutions pour en obtenir qui sont faisables.

8.4 Implémentation parallèle

Dans cette section, nous discutons de la parallélisation sur GPU de la méthode d'optimisation proposée. Le logiciel est conçu de façon à offrir une séparation claire entre l'algorithme d'optimisation et le problème considéré. Cette modularité nous permet d'utiliser le cadriciel *gpuMF* pour implémenter le GA sur le GPU. Contrairement à d'autres cadriciels [41], *gpuMF* parallélise toutes les étapes de la métaheuristique et permet une accélération supérieure allant jusqu'à 249.3x dans le cas du GA. Puisque les détails d'implémentation du cadriciel ont été donnés au

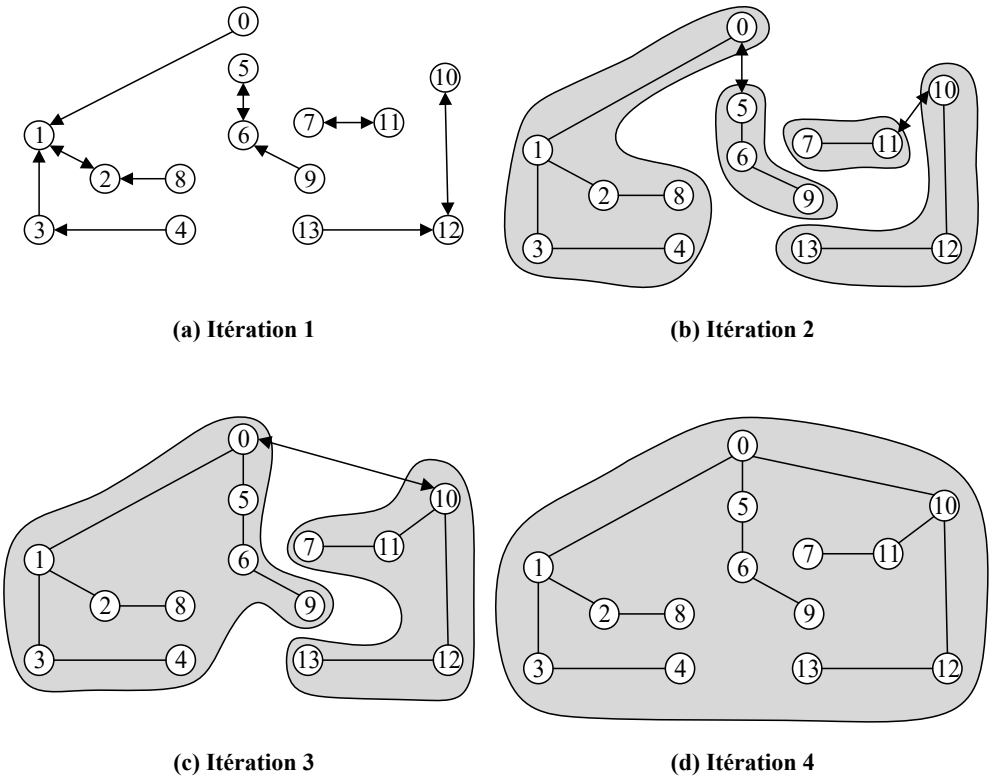
Chapitre 5, ils ne sont pas répétés ici. Dans cette section, nous nous concentrons plutôt sur la parallélisation de la fonction d'aptitude qui modélise le problème considéré.

L'évaluation de la fonction d'aptitude est une opération complexe qui requiert (a) le calcul de l'arbre couvrant de poids minimal pour obtenir la topologie radiale associée à la solution candidate, (b) le parcours de graphe pour mesurer la profondeur de chaque nœud, (c) l'analyse d'écoulement de puissance B-F pour calculer les tensions aux bus, et finalement (d) l'évaluation de $\mathcal{F}(\bar{x})$, soit la fonction d'aptitude en tant que telle. Ces quatre opérations sont exécutées en mode batch où les solutions candidates sont traitées couramment en utilisant un bloc de threads CUDA™ par solution (ou réseau). Dans cette section, nous expliquons comment ces quatre opérations sont implémentées sur le GPU à l'aide des primitives parallèles discutées au Chapitre 4. Ces primitives permettent un parallélisme au niveau des données et exploitent pleinement l'architecture massivement parallèle du GPU.

8.4.1 Arbre couvrant de poids minimal parallèle

La première étape de l'évaluation de fonction d'aptitude est le calcul de l'arbre couvrant de poids minimal. Cette opération permet de décoder la solution candidate afin d'obtenir la topologie radiale associée. Trois algorithmes communs pour calculer l'arbre couvrant de poids minimal sont les algorithmes de Prim, de Kruskal et de Borůvka. L'algorithme de Prim utilise une matrice dense pour représenter la structure du graphe et nécessite deux boucles imbriquées. Une implémentation parallèle sur GPU est proposée dans [283], mais celle-ci se limite à la parallélisation de la boucle interne offrant une accélération limitée de 2x. Il serait difficile de faire mieux étant donné que les itérations de la boucle externe doivent être exécutées dans l'ordre et ne peuvent être parallélisées. L'algorithme de Prim s'adapte donc difficilement au GPU. De son côté, l'algorithme de Kruskal parcourt le graphe en entier à chaque itération afin d'identifier la branche avec le poids minimal. Cette branche est ajoutée à l'arbre couvrant si elle ne crée pas de boucle. Dans le cas contraire, la branche est rejetée. La recherche se termine lorsque toutes les branches ont été soit ajoutées ou rejetées. L'algorithme de Kruskal est implémenté sur GPU dans [284], mais offre seulement un gain de performance modeste de 3x. Parmi les trois méthodes, c'est l'algorithme de Borůvka qui s'adapte le mieux à l'architecture parallèle du GPU. Son fonctionnement est illustré à la Figure 8.8 pour calculer l'arbre couvrant minimale du graphe pondéré montré précédemment à la Figure 8.4. Ce graphe représente le réseau à 16 bus, mais contient uniquement 14 nœuds puisque les trois bus d'alimentation ont été regroupés afin de former le nœud racine. À la première itération de l'algorithme de Borůvka, tous les nœuds du graphe sont visités et la branche avec le poids minimal est identifiée pour chaque nœud. Par exemple, lorsque le nœud 0 est visité, c'est la branche qui le relie au nœud 1 qui a le plus petit poids et qui est sélectionnée. C'est pourquoi une flèche du nœud 0 au nœud 1 est dessinée à la Figure 8.8a. Les sous-

arbres construits par ces connexions sont groupés pour former les super-nœuds représentés par les régions grises à la Figure 8.8b. À l'itération 2, les super-nœuds sont visités et la banche avec le poids minimal est identifiée. Les sous-arbres construits par les branches sélectionnées sont regroupés pour former les nouveaux super-nœuds. Ce processus est répété jusqu'à ce qu'il reste un seul super-nœud qui est en fait l'arbre couvrant de poids minimal. Dans l'exemple illustré à la Figure 8.8, quatre itérations sont nécessaires pour calculer cet arbre qui représente la topologie radiale associée à la solution candidate.



(a) Itération 1 **(b) Itération 2**

(c) Itération 3 **(d) Itération 4**

Figure 8.8: Fonctionnement de l'algorithme de Borůvka pour calculer l'arbre couvrant de poids minimal pour le graphe à la Figure 8.4

Dans [285], Vineet et coll. proposent une parallélisation sur GPU de l'algorithme de Borůvka. Malgré que l'accélération obtenue soit avantageuse, leur approche est critiquée par les auteurs de [286] puisqu'elle est compliquée, qu'elle est optimisée à la main pour un GPU spécifique et qu'elle effectue des accès non coalescents à la mémoire globale du GPU. Dans leur article, Sousa et coll. [286] proposent une parallélisation mieux adaptée à l'architecture du GPU qui utilise des primitives

parallèles standards et qui effectue uniquement des accès coalescents à la mémoire globale. Leur approche parallèle est très efficace et permet une accélération de 16.2x. Dans le logiciel de DFR proposé, notre implémentation du calcul de l'arbre couvrant de poids minimal est faite suivant la méthode de Sousa et coll., mais modifiée pour exécuter en mode batch afin de traiter plusieurs graphes simultanément en utilisant un bloc de threads CUDA™ par graphe. Ceci nous permet de décoder toutes les solutions candidates d'un seul coup. De plus, afin de réduire la mémoire requise, nous encodons les nombres entiers en type *short* à 16 bit au lieu du type *int* à 32 bits. Lorsque le nombre de bus dans le réseau est inférieur à 1024, cette approche nous permet de sauvegarder toutes les données temporaires dans la mémoire partagée du GPU et d'exécuter l'algorithme en entier dans un seul *kernel* CUDA™. Les délais d'accès à la mémoire globale sont ainsi évités. Comme nous le verrons dans la section des travaux expérimentaux, le gain de performance obtenu est de 50x comparé à une exécution séquentielle sur GPU.

Notre implémentation de l'algorithme de Borůvka utilise une matrice d'adjacence creuse en format CSR pour représenter le graphe maillé. La méthode fonctionne uniquement pour les graphes non orientés et requiert que chaque branche soit insérée deux fois dans la matrice, une fois pour chaque direction. La représentation CSR du graphe illustré précédemment à la Figure 8.4 est montrée ici à la Figure 8.9. Étant donné que le graphe est composé de 16 branches, la matrice CSR contient 32 entrées. À l'itération 1 de l'algorithme de Borůvka, une primitive parallèle de map est utilisée pour traiter tous les nœuds concurremment et identifier leur branche minimum. Les branches inverses sont enlevées et celles restantes sont ajoutées à l'arbre couvrant minimal. Une seconde primitive parallèle de map est utilisée pour identifier les super-nœuds associés à chaque nœud. Cette opération requiert plusieurs passages où le successeur d'un nœud est remplacé par le successeur de son successeur jusqu'à ce que chaque nœud pointe vers la racine de son super-nœud. De nouveaux indices sont ensuite attribués aux super-nœuds à l'aide d'une primitive parallèle de scan. Enfin, la matrice d'adjacence CSR pour l'itération suivante est construite par des fonctions parallèles de map et de scan. Ce processus est répété à chaque itération jusqu'à ce qu'il ne reste qu'un seul super-nœud. Ce super-

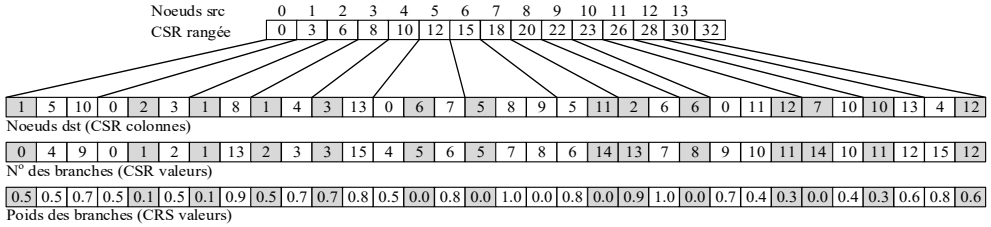


Figure 8.9: Matrice d'adjacence CSR du graphe

nœud représente alors la topologie radiale du réseau de distribution associée à la solution candidate décodée. Les détails des opérations parallèles ainsi qu'un exemple du calcul de l'arbre couvrant de poids minimal sont donnés à la référence [286].

8.4.2 Parcours de graphe parallèle

L'opération du parcours de graphe est utilisée pour calculer la profondeur des nœuds du réseau radial obtenu à l'étape précédente. Deux techniques communes sont l'algorithme de parcours en profondeur (DFS de l'anglais *depth first seach*) et l'algorithme de parcours en largeur (BFS de l'anglais *breadth first search*). Ces deux méthodes sont également efficaces et ont toutes deux une complexité linéaire à condition que le graphe soit représenté à l'aide d'une matrice d'adjacence creuse. Malheureusement, le DFS est intrinsèquement séquentiel et ne peut pas être parallélisé. De son côté, le BFS peut présenter un bon niveau de parallélisme si le degré des nœuds est grand. Malgré cela, le modèle d'accès à la mémoire est irrégulier et s'adapte mal à l'architecture du GPU. Des implémentations parallèles sur GPU de l'algorithme de BFS sont proposées dans [287], [288] et [289]. Les accélérations mesurées sont limitées et respectivement de 2.3x, 2.5x et 3x.

Pour obtenir une accélération supérieure, il est possible de recourir à la stratégie de parallélisation quadratique simple. Cette approche est expliquée en détail dans la référence [290]. Chaque nœud du graphe maintient un état binaire « visité » ou « non visité ». L'algorithme débute en traitant le nœud racine et en marquant son état comme étant « visité ». Tous les autres nœuds du réseau se voient assigner un état « non visité ». À chaque itération, les branches $(n_i \rightarrow n_j)$ du réseau sont inspectées en parallèle et un nœud n_j est traité seulement si n_i est « visité » et n_j est « non visité ». Lorsqu'un nœud n_j est traité, sont état est changée à « visité ». La recherche termine lorsque tous les nœuds ont été visités. Pour calculer la profondeur des nœuds d'un réseau en utilisant la stratégie de parallélisation quadratique simple, il faut assigner une profondeur de 0 au nœud racine et calculer la profondeur du nœud n_j en ajoutant un à la profondeur du nœud n_i qui se trouve à l'autre extrémité de la branche. Le pseudocode pour le calcul de la profondeur des nœuds d'un réseau radial suivant la stratégie de parallélisation quadratique simple est listé à l'algorithme 8.1. Dans ce code, le vecteur *profondeur* est utilisé pour sauvegarder la profondeur de chaque nœud, mais aussi pour identifier l'état binaire des nœuds en assignant une valeur de -1 aux nœuds « non visités ». L'inspection des branches se fait en parallèle à l'aide d'une primitive de map. La variable *recherche_complétée* est partagée par tous les threads. Elle est initialisée à vrai au début de chaque itération et mise à faux lorsqu'un nœud est visité. Cette variable représente le critère de terminaison du processus itératif défini par la boucle. L'algorithme s'arrête lorsqu'aucun nœud n'est visité au cours de l'itération et que la variable *recherche_complétée* est restée vraie.

Algorithme 8.1: Pseudocode pour calculer la profondeur de chaque nœud

```
1:    pour tout i = 0 à nombre_nœuds faire en parallèle
2:        profondeur[i] = -1
3:    profondeur[nœud_racine] = 0
4:    recherche_complétée = faux
5:    tant que (recherche_complétée == faux)
6:        recherche_complétée = vrai
7:        pour tout i = 0 à nombre_branches faire en parallèle
8:            si (profondeur[nœud_src[i]] == -1) && (profondeur[nœud_dest[i]] != -1)
9:                profondeur[nœud_src[i]] = profondeur[nœud_dest[i]] + 1;
10:               recherche_complétée = faux;
11:           else if (profondeur[nœud_src [i]] != -1) && (profondeur[nœud_dest[i]] == -1)
12:               profondeur[nœud_dest[i]] = profondeur[nœud_src [i]] + 1;
13:               recherche_complétée = faux;
14:   retourne profondeur[ ]
```

L'approche de parallélisation quadratique simple a été utilisée à quelques reprises pour implémenter l'opération de parcours de graphes sur le GPU [291], [292]. Cette approche est bien adaptée à l'architecture parallèle du GPU et nécessite un mécanisme de synchronisation très simple comme le montre le pseudocode que nous avons donné. Par contre, elle s'applique difficilement aux réseaux plus larges à cause de sa mauvaise efficacité en termes de travail. En effet, au lieu d'avoir un ordre de complexité linéaire comme les méthodes de DFS et de BFS, l'approche de parallélisation quadratique simple a, comme le dit son nom, une complexité quadratique en termes de travail. Cependant, si le nombre de cœurs disponibles sur le GPU permet de visiter toutes les branches d'un seul coup, la parallélisation quadratique simple nécessite un nombre d'étapes égales à la profondeur maximale du réseau. Ce nombre d'étapes est beaucoup plus petit que pour les recherches DFS et BFS. Comme nous le verrons dans la section sur les tests expérimentaux, l'accélération obtenue par la stratégie de parallélisation quadratique simple est excellente comparée à une implémentation séquentielle du DFS.

8.4.3 Écoulement de puissance régressif-progressif parallèle

Après que la profondeur des bus ait été calculée et que le réseau ait été organisé en niveaux à l'aide d'une matrice de connexion reliant chaque niveau à ses bus, l'analyse de l'écoulement de puissance suivant la méthode régressive-progressive est exécutée. Nous listons au Tableau 8.1 le pseudocode de l'algorithme B-F ainsi que les temps d'exécution pour chacune des fonctions. Ces temps sont pour l'analyse de 1000 instances d'un réseau test à 880 bus et incluent l'exécution séquentielle sur CPU et l'exécution parallèle sur GPU. Les données pour le réseau de distribution utilisé sont disponibles à la référence [182]. Le code est développé à l'aide de Visual Studio 2013, sous Windows 8.1 x64 Pro en utilisant CUDA™ SDK 6.5. Les

temps listés sont les moyennes de 100 essais indépendants et sont mesurés sur un système Dell Précision T7600 équipé de deux processeurs Intel Xeon E5-2650 et d'un processeur graphique NVIDIA® GTX Titan. La version CPU est programmée en C++ et utilise un thread pour analyser séquentiellement les 1000 instances du réseau. La version GPU est programmée en CUDA™ C++ et maximise le nombre de threads pour chacune des fonctions afin d'exploiter le plus possible les 2688 cœurs du processeur graphique. Les paragraphes suivants expliquent l'implémentation pour chacune des fonctions listées au Tableau 8.1. Le programme parallèle s'exécute entièrement sur le GPU. Les données sont gardées dans la mémoire globale du processeur graphique et aucun transfert sur le bus PCIe n'est nécessaire. Lorsque la grandeur du réseau considéré le permet, les données sont chargées dans la mémoire partagée du GPU au début de chaque *kernel* CUDA™ afin d'éviter le plus possible les accès non coalescents à la mémoire globale. À l'exception de celle à la ligne 11, toutes les fonctions sont exécutées en mode batch et utilisent un bloc de threads CUDA™ par réseau.

TABLEAU 8.1
PSEUDOCODE DE LA MÉTHODE RÉGRESSIVE-PROGRESSIVE AVEC TEMPS D'EXÉCUTION ET ACCÉLÉRATION MOYENS (1000 INSTANCES, RÉSEAU TEST DE 880 BUS, 100 ESSAIS)

Pseudocode	Temps d'exécution (ms)		Accélération
	CPU	GPU	
1: *calculer la profondeur de chaque bus*	247.052	0.913	165.8x
2: *construire une matrice de connexion profondeur-bus*	68.908	0.679	60.4x
3: *construire une matrice de connexion bus-enfants*	58.224	1.053	32.2x
4: **Tant que** *(itérations < nombre d'itérations maximal)*	-	-	-
5: **&&** *(erreur maximale > tolérance)*	-	-	-
6: *calculer le courant injecté aux bus*	41.410	0.138	171.5x
7: *compléter la passe régressive*	30.170	1.814	9.4x
8: *compléter la passe progressive*	31.183	1.477	11.9x
9: *calculer l'erreur sur la puissance S*	86.356	0.184	270.8x
10: *trouver l'erreur maximale*	0.890	0.036	14.8x
11: *itération = itération + 1*	-	-	-
12: *sauvegarder les valeurs de V calculées*	30.915	0.079	239.8x
13: *calculer la puissance S requise au bus d'alimentation*	0.300	0.031	4.6x
14: **retour**			
TOTAL	**595.408**	**6.404**	**92.9x**

La fonction à la ligne 1 implémente l'opération de parcours de graphe décrite à la section précédente et calcule en parallèle la profondeur de chaque bus. Aux lignes 2 et 3, deux matrices de connexions creuses en format CSR sont construites afin de relier chaque niveau du réseau aux bus qu'il contient et chaque bus du réseau à ses bus enfants, soit les bus connectés du niveau inférieur. Ces matrices creuses sont essentielles à l'exécution rapide de l'algorithme et permettent un accès direct au bus d'un niveau et aux enfants d'un bus. Ces matrices sont construites directement sur le

GPU à l'aide des primitives parallèles de tri bitonique, de réduction segmentée et de scan suivant la technique discutée précédemment à la section 7.2.3.1. Le processus itératif de l'algorithme débute à la ligne 6 avec le calcul des courants injectés aux bus à l'aide de l'équation (8.6). Tous les bus sont traités en une seule étape en utilisant un thread par bus. Le balayage régressif à la ligne 7 commence au niveau le plus bas et remonte jusqu'au niveau 1. Ce balayage permet de calculer les courants dans les branches à l'aide de l'équation (8.7). Les bus d'un niveau sont traités simultanément en utilisant un thread par bus, mais les niveaux sont visités séquentiellement afin de respecter l'ordre des opérations de la méthode B-F. Cette stratégie de parallélisation est possible grâce aux deux matrices de connexions construites précédemment. Le balayage progressif à la ligne 8 commence quant à lui au niveau 1 et descend jusqu'au niveau le plus bas suivant la même technique de parallélisation que le balayage régressif. Cette opération permet de mettre à jour les valeurs des tensions aux bus à l'aide de l'équation (8.8). Une fois les tensions calculées, les puissances complexes injectées aux bus sont évaluées à la ligne 9 à l'aide de l'équation (8.9) utilisant encore ici un thread par bus. L'erreur maximale entre les valeurs calculées et actuelles des puissances est identifiée à la ligne 10 à l'aide d'une primitive de réduction parallèle. Lorsque cette erreur maximale est plus petite de la tolérance spécifiée ou lorsque le nombre maximal d'itérations a été atteint, le processus itératif de l'algorithme B-F termine. Les valeurs calculées pour les tensions complexes aux bus sont alors sauvegardées en mémoire à la ligne 12 et la puissance requise au bus d'alimentation est calculée à la ligne 13 en utilisant l'équation (8.9).

D'après les temps d'exécution listés au Tableau 8.1, on note que la stratégie de parallélisation proposée offre une accélération significative pour la plupart des fonctions. Celle-ci est causée par le très haut niveau de parallélisme exploité. Par exemple, lors du calcul de la puissance complexe à la ligne 9, le programme CUDA™ lance un thread par bus. Comme nous résolvons 1000 réseaux simultanément et que chaque réseau contient 880 bus, le programme CUDA™ lance en fait 880 000 threads ce qui permet d'occuper pleinement les 2688 cœurs du GPU résultant en une très bonne accélération. Dans le cas des balayages régressifs et progressifs aux lignes 7 et 8, l'accélération mesurée est un peu moins bonne étant donné que seuls les nœuds d'une même couche sont traités simultanément résultant en un niveau de parallélisme moins élevé. Ce parallélisme est réduit davantage à la ligne 13 à un seul thread par réseau lors du calcul des puissances aux bus d'alimentation. Malgré tout, l'accélération globale fournie par le GPU est excellente et essentielle à l'exécution rapide de notre logiciel de reconfiguration optimale des réseaux de distribution.

8.4.4 Fonction objective parallèle

Une fois que l'analyse de PF est terminée et que les tensions aux bus sont connues, les pertes de transport sont calculées simultanément pour chaque branche du réseau en utilisant un thread par branche et additionnées à l'aide d'une réduction parallèle. Ces deux opérations sont effectuées en mode batch traitant concurremment toutes les topologies associées aux solutions candidates. Les excès sur les contraintes d'inégalités sont calculés de la même manière en utilisant les équations (8.11) et (8.12). Finalement, l'aptitude des solutions candidates est calculée à l'équation (8.14) à l'aide d'un thread par réseau avant que le GA reprenne son exécution.

8.4.5 Implémentation parallèle sur CPU

Comme nous l'avons fait pour les deux applications précédentes, nous développons une version parallèle pour processeur multicœur de notre programme de reconfiguration des réseaux de distribution. Cette implémentation utilise OpenMP® pour distribuer les calculs sur plusieurs threads afin d'exploiter pleinement la puissance de calcul des processeurs multicœurs.

Pour ce faire, nous exécutons d'abord le programme séquentiel et mesurons le temps d'exécution des différentes fonctions logicielles qui le composent. Tout comme précédemment, le test est exécuté sur un ordinateur Dell T7600 équipé de deux CPU Intel Xeon E5-2650. Chaque CPU contient huit cœurs qui implémentent la technologie *hyperthreading*. L'ordinateur utilisé contient donc 16 cœurs physiques et 32 cœurs virtuels. Dans ce test, nous optimisons un réseau de distribution à 136 bus dont la description est disponible dans [180]. Le GA est configuré de façon à utiliser 1000 solutions candidates et à exécuter en deux phases de 100 itérations chaque. Les temps mesurés montrent que l'algorithme séquentiel prend 51.882 s à exécuter et que 50.276 s sont utilisées pour l'évaluation de la fonction d'aptitude, soit plus de 97.0% du temps d'exécution. Ceci n'est pas surprenant étant donné la complexité du calcul de l'arbre couvrant de poids minimal, de l'opération de parcours de graphe et de l'analyse de PF nécessaires à la fonction d'aptitude. D'après ces mesures, une stratégie de parallélisation simple et efficace serait de partager les calculs requis par l'évaluation de l'aptitude des solutions candidates sur les multiples cœurs du CPU suivant l'approche maître-esclave que nous avons discutée au Chapitre 2. D'après cette approche, le thread maître exécute le GA séquentiellement sur un seul cœur et délègue le calcul de la fonction d'aptitude aux threads esclaves. Cette stratégie est simple d'implémentation, ne nécessite aucune modification au cadriciel *gpuMF*, exploite un niveau de parallélisme suffisant pour occuper les multiples cœurs du CPU et minimise la communication inter-threads à une seule synchronisation à chaque itération du GA. Pour déterminer le nombre optimal de threads à utiliser, nous évaluons la fonction d'aptitude pour 1000 solutions candidates en utilisant cinq

réseaux tests différents. Nous varions le nombre de threads de 1 à 48 et affichons à la Figure 8.10 l'accélération mesurée. Tout d'abord, nous observons que l'accélération obtenue est plus élevée pour les grands réseaux que pour les petits. En fait, dans le cas des réseaux à 14 et 33 bus, la quantité des calculs impliqués n'est pas suffisante pour masquer la surcharge de travail entraînée par la parallélisation. De plus, nous notons que l'accélération plafonne à 32 threads dans la plupart des cas. Ceci est logique puisque l'ordinateur utilisé contient 32 cœurs virtuels.

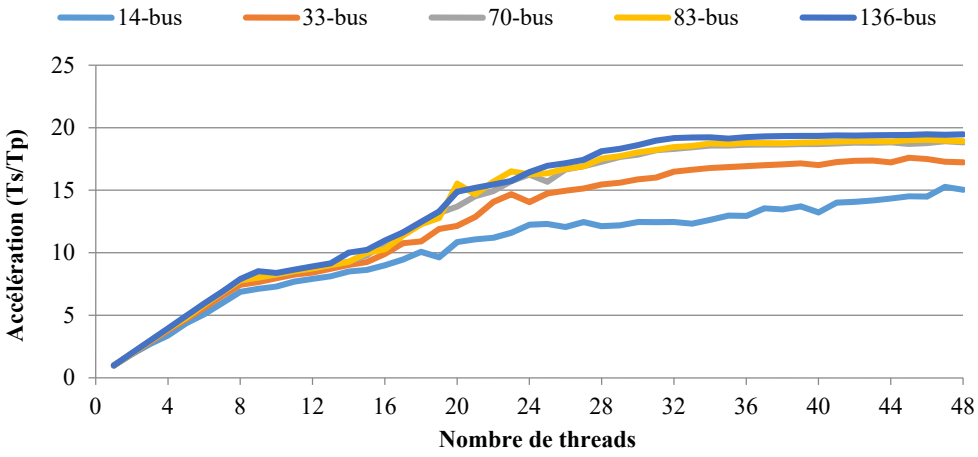


Figure 8.10: Accélération du GA parallèle sur CPU par rapport au nombre de threads OpenMP® pour l'évaluation de la fonction objective du problème de DFR (moyenne de 10 essais, E5-2650)

Dans le cas du réseau à 136 bus, l'accélération maximale est de 19.18x. Nous utilisons la loi d'Amdahl [33] pour calculer à l'équation (8.15) l'accélération attendue pour le programme en entier. Cette accélération est définie comme le temps d'exécution du programme séquentiel divisé par celui du programme parallèle. Dans cette équation, la valeur 51.882 s représente le temps d'exécution du programme séquentiel en entier. La valeur 1.606 s représente le temps nécessaire à la partie du programme parallèle qui n'a pas été parallélisé, soit l'exécution du GA par le thread maître. Finalement, le terme 50.276/19.18 s représente une estimation du temps nécessaire pour calculer en parallèle l'aptitude des solutions candidates par les threads esclaves. Ce calcul nous donne une accélération attendue de 12.27x, ce qui est très bon étant donné que le processeur utilisé contient 16 cœurs.

$$S = \frac{T_{séq}}{T_{par}} = \frac{51.882\ s}{1.606\ s + (50.276/19.18)\ s} = \frac{51.882\ s}{4.227\ s} = 12.27x \qquad (8.15)$$

8.5 Résultats expérimentaux

Dans cette section, nous testons le programme parallèle de DFR sur GPU à l'aide des neuf réseaux de distribution listés au Tableau 8.2. Les données de descriptions du réseau à 16 bus sont données à l'Appendice C. Les données des autres réseaux sont disponibles pour téléchargement dans un répertoire en ligne à la référence [293]. Sept de ces réseaux proviennent de la littérature tandis que deux ont été créés spécialement pour ce travail. Le réseau à 1760 bus a été construit à l'aide de deux instances du réseau à 880 bus et de 20 commutateurs de liaison supplémentaires afin d'interconnecter les deux instances. Le réseau à 4400 bus a été assemblé de la même manière, mais en utilisant cinq instances et 50 commutateurs supplémentaires. Tous les tests sont exécutés sur un ordinateur Dell T7600 équipé de deux processeurs huit cœurs Intel Xeon E5-2650 et un processeur graphique NVIDIA® GTX Titan. Les programmes sont développés sous Visual Studio 2013 et compilés avec optimisation complète pour une meilleure performance. Le programme séquentiel est développé en C++ et utilise la bibliothèque logicielle Boost [294] pour implémenter certaines fonctions de base telles que l'algorithme de tri utilisé pour la construction des matrices de connexions creuses. La version parallèle est développée en CUDA™ et utilise le cadriciel *gpuMF* pour implémenter l'algorithme de GA. Les primitives parallèles sont codées à l'aide de la bibliothèque logicielle NVIDIA® CUB [236], mais modifiées de façon à permettre une exécution en mode batch.

TABLEAU 8.2
DÉTAILS DES RÉSEAUX TESTS

Réseaux tests	Alimentations	Bus	Branches	Loads (MVA)
16 bus [295]	3	13	16	28.70 + *i* 17.30
33 bus [176]	1	32	37	3.72 + *i* 2.30
70 bus [296]	2	68	79	4.47 + *i* 3.06
83 bus [297]	11	83	96	28.35 + *i* 20.70
136 bus [298]	1	135	156	18.31 + *i* 7.93
415 bus [180]	55	415	480	141.75 + *i* 103.50
880 bus [182]	7	873	900	124.87 + *i* 74.36
1760 bus	14	1746	1800	249.74 + *i* 147.72
4400 bus	35	4365	4500	624.36 + *i* 371.81

8.5.1 Évaluation de la stratégie de parallélisation

Dans un premier test, les temps d'exécution et les accélérations pour l'algorithme de l'arbre couvrant de poids minimal de Borůvka, l'algorithme de parcours de graphe et l'algorithme B-F d'analyse de PF sont mesurés et montrés aux Figures 8.11, 8.12 et 8.13 afin de vérifier l'efficacité des stratégies de parallélisation proposées. Les tests sont exécutés en mode batch pour le traitement simultané de 1000 instances des neuf réseaux considérés. Résoudre un nombre si grand de réseaux peut sembler exagéré,

mais ce nombre est typique pour le GA proposé dans ce chapitre. Il est important de noter que les temps montrés aux trois figures n'incluent pas le transfert des données entre le CPU et le GPU puisqu'aucun transfert n'est nécessaire. Dans le GA proposé, les solutions candidates sont créées, modifiées et supprimées sur le GPU même. Seule la solution finale est transférée du GPU au CPU à la toute fin de la recherche. Lors de l'évaluation des solutions candidates, les données nécessaires aux calculs de l'arbre couvrant de poids minimal, au parcours de graphe et à l'analyse de PF sont déjà positionnées dans la mémoire globale du GPU.

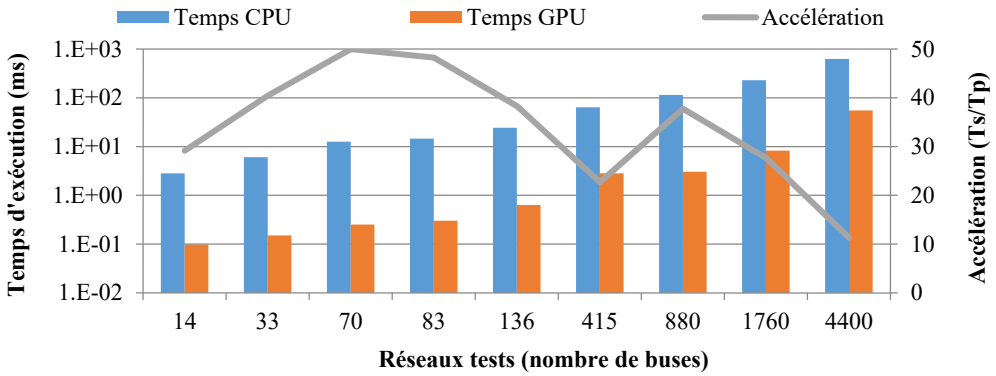


Figure 8.11: Temps d'exécution et accélération de l'implémentation parallèle sur GPU de l'algorithme de Borůvka

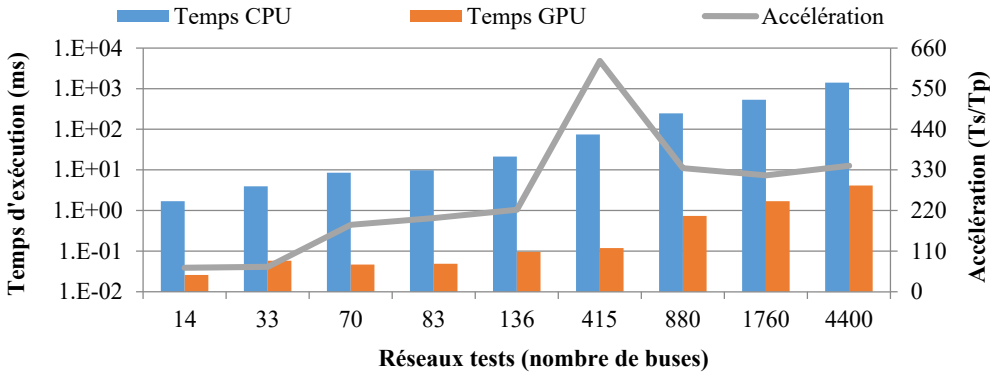


Figure 8.12: Temps d'exécution et accélération de l'implémentation parallèle sur GPU de l'algorithme de parcours de graphe

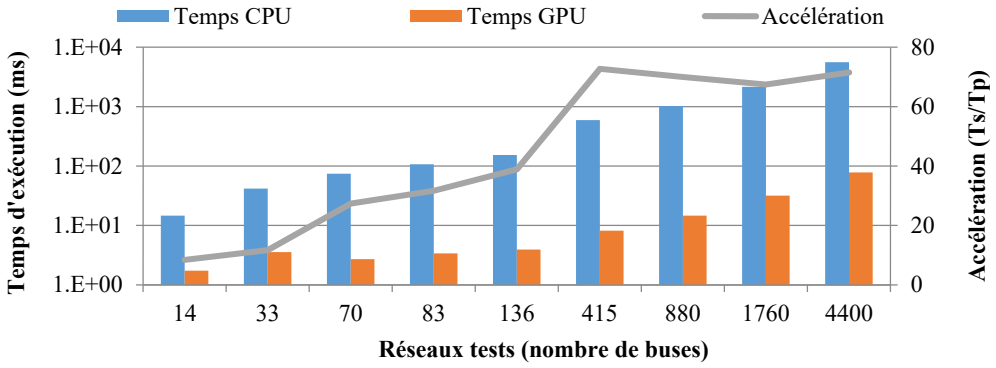


Figure 8.13: Temps d'exécution et accélération de l'implémentation parallèle sur GPU de l'algorithme d'analyse de PF régressif-progressif

Dans le cas du calcul de l'arbre couvrant de poids minimal, nous notons à la Figure 8.11 que l'accélération maximale possible est de 50x, soit trois fois mieux que l'implémentation publiée dans [286]. Cet avantage est causé par l'opération en mode batch qui permet d'exploiter un très grand niveau de parallélisme en traitant tous les graphes d'un seul coup. De plus, en représentant les nombres entiers à l'aide de variables de type *short* à 16 bits, notre implémentation peut sauvegarder toutes les données nécessaires aux calculs dans la mémoire partagée du GPU et peut exécuter l'algorithme en entier dans un seul *kernel* CUDA™. Cette approche évite les longs délais d'accès à la mémoire globale du GPU et les opérations de synchronisation entre les blocs de threads. Dans le cas du réseau à 4400 bus, les données sont trop nombreuses pour tenir dans la mémoire partagée et doivent être stockées dans la mémoire globale du GPU. L'accélération résultante est alors réduite, mais toujours très avantageuse.

La stratégie de parallélisation quadratique utilisée pour implémenter le parcours de graphe se révèle aussi extrêmement profitable. Comme on peut le voir à la Figure 8.12, l'accélération apportée atteint un maximum de 625x comparativement à une implémentation séquentielle d'une recherche DFS. Cette accélération est permise à cause de la grandeur des réseaux considérés. Malgré que la parallélisation quadratique ait une mauvaise efficacité en termes de travail, le nombre de branches dans les réseaux considérés est suffisamment petit pour permettre au GPU de vérifier toutes les branches d'un seul coup ou presque. Le nombre d'étapes nécessaires pour parcourir le graphe en entier est alors égal à la profondeur maximale de l'arbre, soit beaucoup moins que pour une recherche BFS ou DFS. De plus, la stratégie de parallélisation quadratique ne nécessite pas de matrice de connexion, mais uniquement les bus de départ et d'arrivée pour chacune des branches.

Finalement, la stratégie de parallélisation proposée pour l'analyse de PF suivant la méthode B-F est également très avantageuse. Spécifique aux réseaux radiaux, cette méthode est beaucoup plus efficace que l'algorithme de Gauss-Seidel et beaucoup plus simple que celui de Newton-Raphson. Notre implémentation utilise deux matrices de connexions qui permettent d'accéder directement les bus d'un niveau et les enfants d'un bus. Toutes les étapes de l'algorithme sont parallélisées sur le GPU et aucun calcul n'est délégué au CPU. Tel qu'on le voit à la Figure 8.13, l'accélération maximale pour la méthode B-F parallèle est de 73x.

Enfin, bien que la parallélisation pour chacune des trois opérations est essentielle à la performance globale du solveur DFR proposé, c'est le module d'analyse de PF qui a le plus grand influence étant donné qu'il représente la partie la plus fastidieuse de l'évaluation de l'aptitude des solutions candidates. L'accélération finale du logiciel de DFR sera donc fortement influencée par l'accélération obtenue par la parallélisation de la méthode B-F.

8.5.2 Évaluation de la qualité de la solution

Le deuxième test vérifie l'efficacité du GA à calculer des topologies radiales qui minimisent les pertes de puissance active pour les neuf réseaux de distribution considérés. Pour effectuer ce test, le GA est configuré à l'aide des paramètres listés au Tableau 8.3. Les nombres de phases, d'itérations et de solutions candidates tiennent compte de la complexité de chaque réseau et sont fixés expérimentalement afin d'assurer la qualité de la solution finale tout en minimisant le temps d'exécution du programme. Les calculs sont effectués en précision simple au cours de l'optimisation

TABLEAU 8.3
CONFIGURATION DE L'ALGORITHME GÉNÉTIQUE

Paramètres	Réseaux	Valeurs
Taille de l'échantillon du tournoi	tous	3
Probabilité de sélection pour mutation	tous	0.1
Probabilité de mutation	tous	0.2
Portée de la mutation	tous	0.25
Fraction des solutions gardées à la phase suivante	tous	0.01
Tolérance pour l'analyse de PF (précision simple)	sous	1E-6 p.u
Tolérance pour l'analyse de PF (précision double)	tous	1E-10 p.u
Nombre de stages / itérations / solutions	16 bus	1 / 10 / 128
	33 bus	1 / 20 / 512
	70 bus	2 / 50 / 1024
	83 bus	2 / 50 / 1024
	136 bus	2 / 100 / 1024
	415 bus	3 / 200 / 1024
	880 bus	3 / 200 / 1024
	1760 bus	3 / 300 / 1024
	4400 bus	3 / 500 / 1024

et en précision double lors de l'analyse d'écoulement de puissance de la solution finale. Une précision simple est suffisante pour identifier les différences de qualité entre les topologies radiales associées aux solutions candidates et permet de réduire le temps de calcul du logiciel. De l'autre côté, la précision double n'affecte pas la performance du programme puisqu'elle est utilisée qu'une seule fois à la toute fin de la recherche, mais elle assure que les résultats retournés par l'algorithme soient les plus exacts possible. Les détails des solutions trouvées par l'algorithme proposé pour les neuf réseaux tests sont listés à l'Appendice D et disponibles pour téléchargement sur le site internet à la référence [293]. Pour permettre une comparaison avec les travaux antérieurs publiés dans la littérature, nous présentons au Tableau 8.4 les pertes

TABLEAU 8.4
COMPARAISON DES SOLUTIONS OBTENUES PAR L'ALGORITHME PROPOSÉ ET CELLES D'AUTRES RÉFÉRENCES POUR LA MINIMISATION DES PERTES DE PUISSANCE ACTIVE (P_{LOSS})

Réseaux	Source	P_{loss} (kW)	Q_{loss} (kVAR)	V_{min} (p.u.)	V_{avg} (p.u.)	Temps d'exécution (s)
16 bus	Cas de base	511.4	590.4	0.969	0.985	-
	ACO [39]	466.1	544.9	0.972	0.987	1.81
	GA [11]	466.1	544.9	0.972	0.987	2.1
	GA-GPU	466.1	544.9	0.972	0.987	0.02
33 bus	Cas de base	211.0	143.0	0.904	0.945	-
	AIS [40]	139.6	102.3	0.938	0.965	16.9
	PSO [41]	139.6	102.3	0.938	0.965	5.69
	GA-GPU	139.6	102.3	0.938	0.965	0.09
70 bus	Cas de base	227.5	204.9	0.905	0.950	-
	GA [11]	203.2	186.6	0.931	0.953	4.64
	GA [42]	203.9	191.1	0.927	0.953	160
	GA-GPU	201.4	185.1	0.931	0.954	0.53
83 bus	Cas de base	532.0	1374.3	0.929	0.970	-
	PSO [41]	471.1	1252.1	0.952	0.972	36.1
	AIS [40]	469.9	1248.0	0.953	0.972	160
	GA-GPU	469.9	1248.0	0.953	0.972	0.50
136 bus	Cas de base	320.3	702.7	0.931	0.975	-
	GA [43]	280.7	611.0	0.961	0.977	32.6
	MICP [28]	280.1	611.1	0.959	0.977	1800
	GA-GPU	280.1	611.1	0.959	0.977	1.12
415 bus	Cas de base	2660.0	6871.6	0.929	0.969	-
	MICP [28]	2359.9	-	-	-	1800
	MILP [28]	2350.7	-	-	-	1800
	GA-GPU	2349.4	6240.0	0.953	0.972	9.29
880 bus	Cas de base	1496.4	1396.5	0.956	0.987	-
	MIQP [38]	461.4	-	0.982	0.990	3192
	MIQP [27]	461.0	566.7	0.992	0.995	1134
	GA-GPU	457.0	563.3	0.992	0.995	12.00
1760 bus	Cas de base	2992.9	2793.0	0.956	0.987	-
	GA-GPU	821.7	1014.9	0.992	0.996	39.27
4400 bus	Cas de base	7482.2	6982.5	0.956	0.987	-
	GA-GPU	1905.3	2399.5	0.992	0.996	226.17

de distribution et les tensions aux bus des solutions que nous avons obtenues et celles calculées par d'autres auteurs. Comme ce travail met l'accent sur le développement d'un solveur rapide, nous avons sélectionné au Tableau 8.4 des références qui incluent les temps d'exécution afin de pouvoir les comparer aux nôtres. De plus, nous nous sommes assuré que les références choisies donnent les vecteurs de solutions trouvées, soit la liste des branches ouvertes, afin que nous puissions calculer les différences valeurs énumérées au Tableau 8.4 pour une comparaison plus juste.

D'après les résultats publiés au Tableau 8.4, nous formulons les conclusions suivantes. Premièrement, nous notons que notre GA parallèle sur GPU trouve des solutions de qualité égale ou supérieure aux autres méthodes pour tous les réseaux tests analysés. Deuxièmement, nous observons que les temps d'exécutions de notre implémentation sur GPU sont beaucoup plus petits que pour les autres méthodes. Par exemple, dans le cas du réseau à 415 bus, notre méthode est 193x plus rapide que les techniques MICP et MILP publiées dans la référence [180]. Similairement, dans le cas du réseau à 880 bus, notre méthode est 94.5x et 266x plus rapide que les techniques MIQP publiées dans [181] et [182] respectivement. Même si chaque référence utilise un système informatique différent, les écarts de performance sont assez grands pour démontrer clairement l'avantage de la parallélisation sur GPU. Troisièmement, nous remarquons que les métaheuristiques sont utilisées principalement pour optimiser de petits réseaux. Au Tableau 8.4, le plus grand réseau reconfiguré par une métaheuristique n'a que 136 bus. Dépassé ce nombre, seules les méthodes déterministes semblent possibles. Dans notre cas, le GA proposé utilise un encodage unique qui assure la topologie radiale des solutions candidates sans avoir recours à des opérateurs complexes qui limitent l'exploration de l'espace de recherche. Grâce à cette nouvelle représentation, notre GA permet d'optimiser des réseaux bien plus gros que n'importe lesquelles des métaheuristiques ou méthodes déterministes citées dans ce travail. En fait, le GA proposé permet de reconfigurer des réseaux cinq fois plus grands que la deuxième méthode la plus efficace, soit la technique de MIQP publiée dans [181] et [182].

8.5.3 Évaluation de l'approche à multiples phases

Dans un troisième test, pour vérifier l'efficacité de l'approche d'optimisation à multiples phases, nous exécutons le GA pour calculer la configuration optimale du réseau à 4400 bus et montrons à la Figure 8.14 l'aptitude et les pertes de distributions moyennes des solutions candidates à chaque itération. Nous notons d'abord que l'aptitude moyenne des solutions est initialement très mauvaise et augmente rapidement dans l'intervalle des solutions faisables. Ceci démontre l'efficacité de la stratégie d'optimisation du GA et la convenance de la fonction d'aptitude que nous avons définie. Deuxièmement, nous notons que les pertes de distribution stagnent à environ 1940 kW après l'itération 250 ce qui signifie que le GA a convergé vers un

optimum local. Malgré que ces pertes soient beaucoup plus petites que pour le cas de base, elles ne sont pas optimales. À l'itération 500, les meilleures solutions sont gardées et les autres sont réinitialisés aléatoirement suivant une distribution uniforme sur l'espace de recherche en entier en préparation pour la phase suivante. À la phase 2, on remarque une amélioration significative et les pertes stagnent à environ 915 kW. Finalement, à la troisième phase, le GA trouve une topologie radiale avec des pertes de seulement 905 kW. Aucune amélioration n'a pu être observée en exécutant l'algorithme après la troisième phase. D'après les résultats à la Figure 8.14, nous concluons que l'approche à multiples phases est avantageuse et permet d'échapper plus facilement les optima locaux, spécialement lors de la reconfiguration de très grands réseaux de distribution.

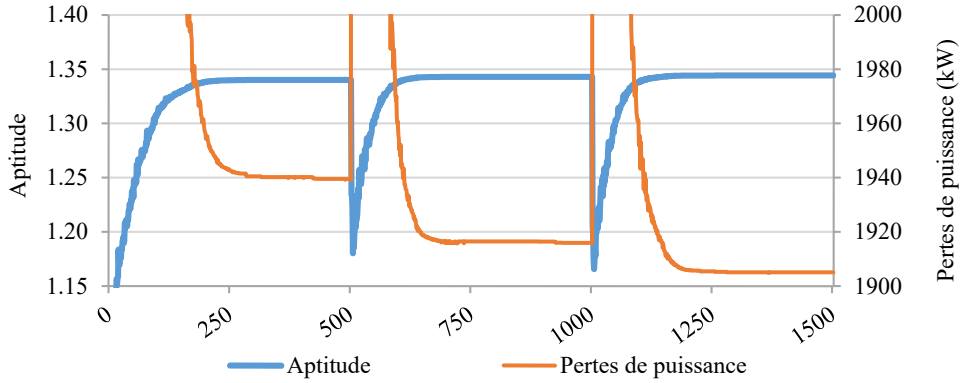


Figure 8.14: Aptitude et pertes de puissance de la meilleure solution à chaque itération pour le réseau test à 4400 bus

8.5.4 Évaluation de l'accélération

Finalement, dans un quatrième test, nous exécutons le GA à 100 reprises pour chaque réseau de distribution considéré et montrons au Tableau 8.5 et à la Figure 8.15 les temps moyens pour l'implémentation séquentiel sur CPU, l'implémentation parallèle sur CPU utilisant OpenMP® et l'implémentation parallèle sur GPU utilisant CUDA™. Nous incluons aussi au Tableau 8.5 la moyenne et l'écart type des pertes de distribution. Dans les résultats obtenus, on remarque que l'accélération maximale offerte par le CPU multicœur est de 13.4x tandis que celle offerte par le GPU est de 66.2x ce qui démontre encore une fois l'avantage d'une implémentation parallèle sur GPU. Finalement, les moyennes des pertes de distribution confirment que les valeurs listées précédemment au Tableau 8.4 n'ont pas été choisies à la main, mais sont bien représentatives de l'efficacité de la métaheuristique parallèle proposée dans cette thèse.

TABLEAU 8.5
TEMPS D'EXÉCUTION DE L'ALGORITHME PROPOSÉ ET PERTES DE DISTRIBUTION DES SOLUTIONS FINALES CALCULÉES (MOYENNES DE 100 ESSAIS)

Réseaux	Temps d'exécution (s)			Accélération		P_{loss} (kW)	
	CPU	OMP	GPU	OMP	GPU	Moy	ÉT
16 bus	0.05	0.01	0.02	5.2x	2.2x	466.1	0.00
33 bus	0.69	0.06	0.09	10.6x	7.3x	140.3	0.93
70 bus	14.1	1.2	0.53	12.1x	26.5x	201.4	0.13
83 bus	17.3	1.4	0.50	12.4x	35.5x	469.9	0.00
136 bus	51.9	4.3	1.1	12.2x	45.9x	280.2	0.05
430 bus	532.1	39.8	9.3	13.4x	58.8x	2,349.5	0.11
880 bus	793.8	63.6	12.0	12.5x	66.2x	457.2	0.61
1760 bus	2,488.7	195.7	39.3	12.7x	63.4x	823.2	0.80
4400 bus	10,806.1	931.2	226.2	11.6x	47.8x	1911.8	3.31

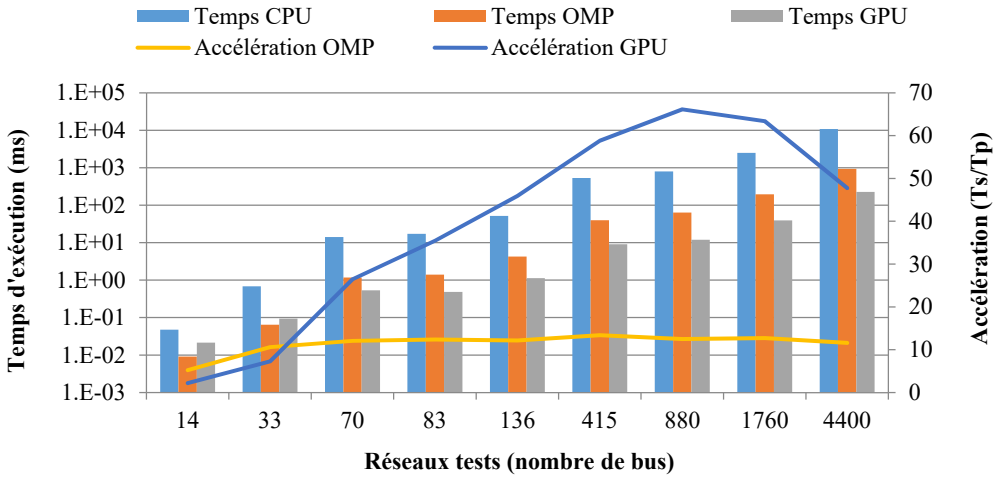


Figure 8.15: Temps d'exécution et accélérations pour l'implémentation séquentielle sur CPU, l'implémentation parallèle sur CPU et l'implémentation parallèle sur GPU

8.6 Conclusion

Dans ce chapitre, nous avons utilisé le cadriciel *gpuMF* pour implémenter un GA parallèle sur GPU pour la reconfiguration optimale des réseaux de distribution. L'algorithme proposé utilise un encodage unique basé sur le calcul de l'arbre couvrant de poids minimal pour représenter les solutions candidates. Cet encodage assure la topologie radiale du réseau et s'adapte mieux à l'architecture parallèle du GPU puisque, contrairement aux méthodes antérieures, il ne nécessite aucun opérateur complexe. Les topologies candidates sont analysées à l'aide de l'algorithme B-F pour obtenir l'état complet sur réseau. Cet algorithme de PF s'applique uniquement à l'analyse de réseaux radiaux, il reste efficace lorsque le rapport R/X est élevé et il est

beaucoup plus rapide que les méthodes de G-S ou de N-R ce qui en fait l'algorithme de choix pour les réseaux de distribution. Après l'analyse de PF, la qualité des solutions candidates est évaluée par une fonction d'aptitude qui intègre l'objectif d'optimisation, soit la minimisation des pertes de distribution, et les contraintes physiques telles que les tensions minimales aux bus et les puissances maximales dans les branches. Le GA est exécuté en plusieurs phases afin d'améliorer l'exploration de l'espace de recherche et rendre l'algorithme plus résistant à la convergence prématurée vers un optimum local. Le GA est parallélisé sur le GPU à l'aide du cadriciel *gpuMF* tandis que la fonction d'aptitude est programmée à la main. Cette fonction est complexe et sa parallélisation n'est pas triviale. Elle inclut l'algorithme de Borůvka, une opération de parcours de graphe, l'algorithme de B-F et le calcul numérique de l'aptitude des solutions. Les détails de la parallélisation pour chacune de ces quatre opérations ont été discutés extensivement. Le programme parallèle de DFR a été testé sur neuf réseaux de 16 à 4400 bus. Les résultats obtenus ont été comparés à ceux publiés dans la littérature. Dans tous les cas, les solutions calculées par notre programme permettent des pertes de distribution égales ou plus petites que pour toutes les méthodes antérieures citées dans ce chapitre. De plus, en exploitant l'architecture massivement parallèle du GPU, l'algorithme proposé offre une accélération impressionnante de 66.2x comparée à une exécution séquentielle sur CPU. Cette accélération est un avantage important puisqu'elle permet la reconfiguration rapide du réseau de distribution afin de toujours opérer dans une topologie quasi optimale même après une perturbation, une panne ou tout simplement une fluctuation de la demande en énergie.

Le logiciel de DRF présenté dans ce chapitre représente deux contributions importantes. Premièrement, il s'agit de la première fois qu'un algorithme de DFR est parallélisé sur GPU. Notre implémentation parallélise toutes les étapes de l'algorithme et exécute entièrement sur le GPU. L'accélération obtenue est très avantageuse et essentielle pour un contrôle réactif des réseaux de distribution. Deuxièmement, l'encodage des solutions candidates est unique et complètement différent des techniques antérieures. La représentation proposée est simple, elle garantit la topologie radiale du système et elle permet à la métaheuristique d'optimiser de très grands réseaux, soit cinq fois plus grands que n'importe lesquelles des références citées dans notre revue de la littérature. Pour conclure, dans le contexte du problème de DFR, la parallélisation de métaheuristiques sur GPU permet d'optimiser des réseaux plus grands, dans des temps de calcul plus courts tout en obtenant des solutions de meilleure qualité. Dans un contexte plus large, nous avons démontré par cette troisième application que le développement de métaheuristiques parallèles sur GPU représente une contribution significative dans le domaine des réseaux intelligents.

Chapitre 9
Conclusion

En conclusion, nous récapitulons brièvement le contenu de chacun des chapitres, nous soulignons les principales contributions apportées par notre recherche, nous discutons de la validité de l'hypothèse posée dans l'introduction et nous proposons quelques sujets de travaux futurs.

9.1 Sommaire

Au premier chapitre, nous avons introduit le réseau électrique intelligent et les défis d'optimisation associés. Nous avons présenté l'énoncé de la thèse, nos objectifs de recherche, nos motivations et les contributions scientifiques apportées.

Au deuxième chapitre, nous avons présenté une revue de la littérature sur les travaux antérieurs dans le domaine des métaheuristiques parallèles et de l'optimisation des réseaux intelligents. Nous avons identifié les lacunes des méthodes actuelles afin de mieux comprendre l'importance des contributions effectuées par cette thèse. Parmi ces lacunes, nous avons souligné le manque d'un cadriciel adéquat pour les métaheuristiques parallèles sur GPU, l'incapacité des méthodes déterministes à optimiser efficacement des solutions aux trois problèmes considérés et le trop long temps d'exécution des métaheuristiques.

Au troisième chapitre, nous avons traité des métaheuristiques. Nous avons débuté par une formulation mathématique des problèmes d'optimisation et avons enchaîné avec la théorie de la complexité afin d'expliquer l'avantage qu'ont les métaheuristiques pour résoudre les problèmes NP-difficiles. Nous avons donné une explication générale du fonctionnement ainsi que des caractéristiques communes à toutes les métaheuristiques. Nous avons donné des descriptions détaillées des algorithmes de PSO et de GA. Nous avons terminé ce chapitre en identifiant deux approches possibles pour le développement de métaheuristiques hybrides.

Au quatrième chapitre, nous avons décrit l'architecture et le modèle de programmation des processeurs graphiques. Comparativement au CPU, les GPU permettent d'exploiter un niveau de parallélisme bien plus grand à condition de minimiser la divergence d'exécution entre les nombreux threads. Ce chapitre a présenté plusieurs primitives parallèles adaptées à l'architecture des GPU. Ces primitives ont été utilisées aux chapitres suivants pour l'implémentation des métaheuristiques sur le GPU.

Au cinquième chapitre, nous avons présenté le cadriciel *gpuMF* que nous avons développé pour l'implémentation de métaheuristiques parallèles sur GPU. Ce cadriciel représente la première contribution de cette thèse et comporte plusieurs améliorations comparées aux logiciels existants. Une programmation orientée-objet en C++ est utilisée au niveau supérieur pour définir la structure et la séquence d'opérations des algorithmes tandis que le langage CUDA™ est utilisé au niveau inférieur pour exécuter les calculs sur le GPU. Le cadriciel proposé implémente les algorithmes de PSO et de GA. Son bon fonctionnement ainsi que sa performance ont été mesurés à l'aide de six fonctions tests.

Au sixième chapitre, nous avons utilisé le cadriciel *gpuMF* pour la minimisation des harmoniques d'un onduleur multiniveau, soit la première des trois applications considérées dans cette thèse. Dans le contexte des réseaux électriques intelligents, ce problème d'optimisation se situe au niveau de la production de l'énergie. Dans ce chapitre, nous avons donné la formulation mathématique du problème, la stratégie d'optimisation proposée, quatre techniques de parallélisation possibles ainsi que les résultats des tests expérimentaux vérifiant la qualité des solutions calculées et l'accélération apportée par le GPU. De plus, nous avons proposé une seconde méthode parallèle sur GPU pour ce même problème. Celle-ci se base sur une formulation mathématique différente et permet un calcul plus rapide au détriment d'une moins bonne flexibilité au niveau du choix de l'indice de modulation et des harmoniques à minimiser.

Au septième chapitre, nous avons utilisé le cadriciel gpuMF pour l'optimisation de l'écoulement de puissance. Dans le contexte des réseaux électriques intelligents, ce problème d'optimisation se situe au niveau du transport de l'énergie électrique. Étant donné que la métaheuristique doit effectuer une analyse de PF pour évaluer l'aptitude des solutions candidates, nous avons aussi proposé des parallélisations sur GPU des algorithmes de Gauss-Seidel et de Newton-Raphson. Dans ce chapitre, nous avons couvert la formulation du problème, la technique d'optimisation, l'approche de parallélisation ainsi que les tests expérimentaux.

Au huitième chapitre, nous avons présenté la troisième application considérée, soit la reconfiguration des réseaux de distribution à l'aide du cadriciel *gpuMF*. Dans le contexte des réseaux électriques intelligents, ce problème d'optimisation combinatoire se situe au niveau de la distribution de l'énergie, la dernière étape de l'acheminement de l'électricité vers le client. Tout comme pour les deux autres applications, nous avons donné la formulation mathématique du problème, la stratégie d'optimisation proposée, la parallélisation sur GPU et les résultats expérimentaux.

Finalement, dans ce chapitre de conclusion, nous résumons les contributions de notre recherche, nous discutons de la validité de l'hypothèse posée dans l'introduction et nous proposons des opportunités de recherches futures.

9.2 Contributions

Les travaux de recherche effectués dans le cadre de cette thèse ont permis de répondre à plusieurs des limitations des méthodes actuelles présentées dans la revue de la littérature au Chapitre 2. Les contributions apportées ont été identifiées dans l'introduction, mais sont réitérées à nouveau ci-dessous :

1. Le développement du cadriciel *gpuMF* pour l'implémentation de métaheuristiques parallèles sur GPU est une contribution importante. C'est le seul cadriciel disponible qui permette de paralléliser en entier l'algorithme sur le GPU afin d'obtenir la meilleure accélération possible. Dans le cadre de notre recherche, ce cadriciel s'est révélé très avantageux puisqu'il nous a permis de réutiliser les métaheuristiques pour les trois applications considérées sans avoir à les modifier. Dans le futur, ce cadriciel pourrait être utilisé pour résoudre d'autres problèmes d'optimisation. Seules la représentation des données et la fonction d'aptitude doivent être définies.

2. Les parallélisations sur GPU des algorithmes de PSO et de GA représentent aussi deux contributions importantes. Les accélérations que nous avons obtenues sont supérieures à celle de plusieurs implémentations existantes à cause du plus haut niveau de parallélisme exploité, de la maximisation des accès coalescents à la mémoire et de la minimisation de la divergence d'exécution des threads.

3. L'utilisation de *gpuMF* et l'implémentation d'une méthode directe sur GPU pour la minimisation des harmoniques d'un onduleur multiniveau sont aussi deux contributions importantes. Les deux méthodes proposées permettent d'optimiser des onduleurs avec un très grand nombre de sources et offrent des accélérations respectives de 454x et 534x comparées à une exécution séquentielle sur CPU. À cause du grand nombre de sources, nos algorithmes permettent de minimiser davantage le taux de distortion harmonique de l'onde de tension de sortie comparativement aux méthodes actuelles.

4. L'application du cadriciel *gpuMF* pour l'optimisation de l'écoulement de puissance dans les réseaux de transport d'électricité est aussi une contribution significative. C'est en fait la première fois qu'une métaheuristique parallèle sur GPU est proposée pour le problème d'OPF. L'accélération résultante de 17.2x est un avantage important qui assure un ajustement rapide des réglages du système de transport d'électricité afin de toujours opérer dans le régime optimal.

5. Pour évaluer les solutions candidates au problème d'OPF, nous avons dû développer des implémentations parallèles des algorithmes de PF de G-S et de N-R. Ces deux implémentations représentent des contributions majeures puisque, contrairement aux tentatives antérieures, nous avons utilisé des matrices creuses afin de diminuer l'ordre de complexité des calculs et nous avons

parallélisé toutes les étapes des algorithmes. Nos implémentations parallèles permettent des accélérations maximales de 45.2x pour la méthode de G-S et de 17.8x pour la méthode de N-R.

6. Finalement, l'application du cadriciel *gpuMF* pour la reconfiguration des réseaux de distribution représente deux contributions majeures. Premièrement, il s'agit aussi de la première fois que le GPU est proposé pour résoudre ce problème. La parallélisation est très avantageuse et permet une accélération de 66.2x. Deuxièmement, l'encodage proposé pour représenter les solutions candidates est une autre contribution importante. Il permet d'améliorer l'efficacité de la recherche et de considérer des réseaux cinq fois plus grands que les méthodes antérieures référencées.

Comme nous l'avons identifié au début des chapitres, les travaux de recherche effectués dans le cadre de cette thèse ont été, pour la majorité, publiés dans des journaux scientifiques ou présentés à des conférences. Pour récapituler, les travaux sur les parallélisations de métaheuristiques sur CPU multicœurs ont été publiés dans [44], [45] et [47]. Les travaux sur le développement de métaheuristiques hybrides sur CPU ont été publiés dans [46]. La parallélisation de métaheuristiques sur GPU a été publiée dans [50] et [51]. Les travaux sur le développement de métaheuristiques hybrides sur GPU ont été publiés dans [261]. Le cadriciel *gpuMF* a été publié dans [299] et [300]. Les algorithmes de PSO et de GA sur GPU pour la minimisation des harmoniques d'un onduleur multiniveau ont été publiés dans [52] et [104]. La méthode directe sur GPU pour résoudre ce même problème a été publiée dans [102]. Les implémentations parallèles sur GPU des algorithmes de G-S et de N-R ont été publiées dans [137]. Les travaux sur l'optimisation de l'écoulement de puissance sur GPU ont été soumis pour publication dans le journal *IEEE Transaction on Smart Grid*. Finalement, la parallélisation sur GPU de l'algorithme génétique pour la reconfiguration des réseaux de distribution a été publiée dans [301].

9.3 Discussion

Dans le chapitre d'introduction, nous avons énoncé l'hypothèse de recherche suivante :

> « *Le développement de métaheuristiques parallèles sur GPU contribue à l'amélioration du contrôle des réseaux électriques intelligents en permettant de calculer des solutions de meilleure qualité à des problèmes d'optimisation de plus grande dimension tout en réduisant les temps de calcul.* »

Pour vérifier cette hypothèse, nous avons développé un cadriciel pour métaheuristiques parallèles sur GPU et utilisé ce cadriciel pour résoudre trois problèmes d'optimisation d'intérêts pour le contrôle des réseaux électriques intelligents, soit la minimisation des harmoniques d'un onduleur multiniveau, l'optimisation de l'écoulement de puissance et la reconfiguration optimale des réseaux de distribution. Ces problèmes ont été choisis puisqu'ils couvrent l'étendue des réseaux électriques en considérant la production, le transport et la distribution de l'électricité. De plus, ces problèmes sont non linéaires, non convexes et difficilement résolubles par les méthodes classiques ce qui justifie l'emploi des métaheuristiques. Finalement, ces problèmes se situent au niveau de l'opération du réseau électrique et par conséquent exigent des temps de calcul les plus courts possible ce qui nous motive à développer des implémentations logicielles qui prennent avantage de l'architecture massivement parallèle des GPU. Pour valider l'hypothèse posée, il faut vérifier si le cadriciel développé pour l'implémentation de métaheuristiques parallèles sur GPU permet les trois améliorations suivantes :

1. considérer des problèmes de plus grande dimension;
2. améliorer la qualité de la solution calculée; et
3. réduire le temps de calcul comparé à une implémentation séquentielle sur CPU.

Dans le cas du premier problème qui consiste à minimiser les harmoniques de la tension de sortie d'un onduleur multiniveau, la parallélisation sur GPU a permis d'optimiser des onduleurs ayant jusqu'à 100 sources de tension continue. Ce nombre est beaucoup plus grand que pour les méthodes classiques qui dépassent rarement cinq sources [87], [93], [94], [95], [96], [97] à cause de la complexité des équations ou que pour les métaheuristiques séquentielles qui sont limitées à une douzaine de sources [76], [244] à cause d'un trop long temps d'exécution [13], [104]. La qualité de la solution obtenue est aussi améliorée. En considérant des onduleurs avec un très grand nombre de sources, le nombre de niveaux de la tension de sortie en escalier est augmenté ce qui permet de réduire davantage le taux de distortion harmonique résultant. Finalement, le cadriciel proposé permet aussi de réduire significativement le temps de calcul. L'accélération maximale obtenue est de 453x comparée à une implémentation séquentielle sur CPU.

Dans le cas de la deuxième application, soit l'optimisation de l'écoulement de puissance, la parallélisation sur GPU a permis d'optimiser des réseaux de transport d'électricité de plus grande dimension. Comme nous l'avons vu au Chapitre 7, même les travaux récents qui utilisent les métaheuristiques sont souvent limités à de petits réseaux tels que le réseau test IEEE à 30 bus [265], [266], [267] et [268]. Quelques-uns considèrent des réseaux plus grands tels que le IEEE à 118 bus [168], [269] et [270], mais ceux-ci sont incapables de faire respecter les limites sur la puissance

réactive des générateurs. Finalement, notre revue de littérature a identifié une seule référence [169] qui utilise une métaheuristique pour optimiser un réseau plus grand, soit le réseau test IEEE à 300 bus. Toutefois, cette référence ne donne pas le vecteur solution obtenu ce qui nous empêche de vérifier si la contrainte sur la puissance réactive des générateurs est respectée. Dans notre cas, le cadriciel développé permet d'optimiser les réseaux IEEE à 30, 118 et 300 bus tout en respectant les limites des générateurs. De plus les coûts de production de la solution calculée sont plus petits que pour toutes les méthodes déterministes et non déterministes référencées. La qualité supérieure de la solution calculée est due à la fonction d'aptitude que nous avons définie, à l'approche à multiples phases que nous avons implémentée et au très grand nombre de solutions candidates rendu possible grâce à la parallélisation sur GPU. Finalement, notre cadriciel parallèle a permis une accélération de 17.2x comparée à une implémentation séquentielle sur CPU.

Dans le cas de la troisième application, soit la reconfiguration optimale des réseaux de distribution, la parallélisation sur GPU a permis encore une fois d'optimiser des réseaux de plus grande dimension. Comme nous l'avons identifié dans au Chapitre 2, les métaheuristiques sont généralement limitées à de petits réseaux à cause de leur temps d'exécution trop long et de la complexité engendrée par la contrainte sur la topologie radiale du réseau. Suite à notre revue de la littérature, le plus grand réseau trouvé qui a été reconfiguré par une métaheuristique n'avait que 476 bus [188]. Dans le cas des méthodes déterministes, la dimension maximale augmente à 880 bus [181], [182]. Dans notre cas, le cadriciel proposé permet de reconfigurer des réseaux allant jusqu'à 4400 bus, soit cinq fois plus grands que les références citées. De plus, les pertes de dispersion associées aux topologies trouvées par notre cadriciel sont plus petites ou égales à celles des références citées pour tous les tests effectués. Finalement, la parallélisation sur GPU s'est révélée encore une fois extrêmement avantageuse en offrant une accélération de 66.2x comparée à une implémentation séquentielle sur CPU.

D'après les résultats obtenus pour chacun des trois problèmes d'optimisation considérés, nous validons l'hypothèse posée et concluons que la parallélisation de métaheuristiques sur GPU contribue à l'amélioration du contrôle des réseaux électriques intelligents.

9.4 Travaux futurs

Avant de terminer cette thèse, nous proposons quatre opportunités pour des travaux de recherche futurs.

Premièrement, il serait possible de poursuivre l'implémentation d'autres métaheuristiques parallèles sur GPU. Chaque métaheuristique a des caractéristiques

uniques et il se peut qu'une autre métaheuristique permette d'améliorer davantage la qualité des solutions que nous avons calculées pour les trois problèmes d'optimisation considérés dans cette thèse. L'implémentation d'une autre métaheuristique serait facilitée par le cadriciel *gpuMF* qui définit la structure commune aux métaheuristiques.

Deuxièmement, il serait possible de poursuivre la recherche sur l'optimisation de l'écoulement de puissance dans un réseau de transport d'électricité afin de tenir compte du bris possible d'une ligne à haute tension. On parle alors du problème d'OPF avec contrainte *N-1* [302], [303] et [304]. Ce problème consiste à calculer les réglages des générateurs et de l'équipement d'appoint de façon à assurer l'opération optimale du système tout en assurant la faisabilité de la solution en cas de faute. La dimension du problème est alors multipliée par N puisque l'évaluation de solutions candidate doit tenir compte des N scénarios contingents. La parallélisation sur GPU pourrait permettre de mieux gérer cette augmentation de la complexité.

Troisièmement, du côté de la reconfiguration optimale des réseaux de distribution, il serait possible d'inclure la présence de générateurs distribués comme il est suggéré dans [305] et [306]. Contrairement aux panneaux solaires ou aux éoliennes, ces générateurs ont un coût d'opération et un impact environnemental que l'on doit considérer. Dépendamment de l'état du système, il se peut qu'il soit plus économique ou mieux pour l'environnement de réduire la demande aux bus d'alimentation et d'augmenter la production des générateurs distribués aux autres bus. Pour inclure des générateurs distribués dans le calcul de la reconfiguration, il faudrait ajouter des éléments aux vecteurs solutions afin de représenter les réglages de ces générateurs. Il faudrait aussi modifier la fonction d'aptitude pour non seulement inclure les pertes de distribution, mais aussi les coûts de production et les émissions polluantes.

Finalement, une quatrième opportunité de recherche serait d'étudier d'autres problèmes d'optimisation reliés au contrôle des réseaux électriques intelligents. Comme cette thèse a démontré que la parallélisation de métaheuristiques sur GPU est avantageuse pour trois problèmes qui couvrent l'étendue du système électrique en touchant la production, le transport et la distribution de l'électricité, il est fort probable qu'elle soit aussi avantageuse pour d'autres problèmes importants.

Références

[1] “Smart Grids Technology Roadmap.” International Energy Agency, 2011.

[2] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati, and G. P. Hancke, “Smart Grid Technologies: Communication Technologies and Standards,” *IEEE Trans. Ind. Inform.*, vol. 7, no. 4, pp. 529–539, Nov. 2011.

[3] J. Liu, F. Ponci, A. Monti, C. Muscas, P. A. Pegoraro, and S. Sulis, “Optimal Meter Placement for Robust Measurement Systems in Active Distribution Grids,” *IEEE Trans. Instrum. Meas.*, vol. 63, no. 5, pp. 1096–1105, May 2014.

[4] D. K. Khatod, V. Pant, and J. Sharma, “Evolutionary programming based optimal placement of renewable distributed generators,” *IEEE Trans. Power Syst.*, vol. 28, no. 2, pp. 683–695, May 2013.

[5] Y. Ghiassi-Farrokhfal, F. Kazhamiaka, C. Rosenberg, and S. Keshav, “Optimal Design of Solar PV Farms With Storage,” *IEEE Trans. Sustain. Energy*, vol. 6, no. 4, pp. 1586–1593, Oct. 2015.

[6] H. Yang, K. Xie, H.-M. Tai, and Y. Chai, “Wind Farm Layout Optimization and Its Application to Power System Reliability Analysis,” *IEEE Trans. Power Syst.*, vol. PP, no. 99, pp. 1–9, 2015.

[7] G. Carpinelli, G. Celli, S. Mocci, F. Mottola, F. Pilo, and D. Proto, “Optimal Integration of Distributed Energy Storage Devices in Smart Grids,” *IEEE Trans. Smart Grid*, vol. 4, no. 2, pp. 985–995, Jun. 2013.

[8] M. Allalouf, G. Gershinsky, L. Lewin-Eytan, and J. Naor, “Smart Grid Network Optimization: Data-Quality-Aware Volume Reduction,” *IEEE Syst. J.*, vol. 8, no. 2, pp. 450–460, Jun. 2014.

[9] J. Huang, H. Wang, Y. Qian, and C. Wang, “Priority-Based Traffic Scheduling and Utility Optimization for Cognitive Radio Communication Infrastructure-Based Smart Grid,” *IEEE Trans. Smart Grid*, vol. 4, no. 1, pp. 78–86, Mar. 2013.

[10] S. Canale, A. Di Giorgio, A. Lanna, A. Mercurio, M. Panfili, and A. Pietrabissa, “Optimal Planning and Routing in Medium Voltage PowerLine Communications Networks,” *IEEE Trans. Smart Grid*, vol. 4, no. 2, pp. 711–719, Jun. 2013.

[11] Y. S. Khoo, A. Nobre, R. Malhotra, D. Yang, R. Ruther, T. Reindl, and A. G. Aberle, “Optimal Orientation and Tilt Angle for Maximizing in-Plane Solar Irradiation for PV Applications in Singapore,” *IEEE J. Photovolt.*, vol. 4, no. 2, pp. 647–653, Mar. 2014.

[12] H. Jafarnejadsani and J. Pieper, “Gain-Scheduled -Optimal Control of Variable-Speed-Variable-Pitch Wind Turbines,” *IEEE Trans. Control Syst. Technol.*, vol. 23, no. 1, pp. 372–379, Jan. 2015.

[13] A. Kavousi, B. Vahidi, R. Salehi, M. Bakhshizadeh, N. Farokhnia, and S. S. Fathi, “Application of the Bee Algorithm for Selective Harmonic Elimination Strategy in Multilevel Inverters,” *IEEE Trans. Power Electron.*, vol. 27, no. 4, pp. 1689–96, Apr. 2012.

[14] W. Wu, J. Chen, B. Zhang, and H. Sun, “A Robust Wind Power Optimization Method for Look-Ahead Power Dispatch,” *IEEE Trans. Sustain. Energy*, vol. 5, no. 2, pp. 507–515, Apr. 2014.

[15] E. T. Lau, Q. Yang, G. A. Taylor, A. B. Forbes, P. Wright, and V. N. Livina, “Optimization of carbon emissions in smart grids,” in *Power Engineering Conference (UPEC), 2014 49th International Universities*, 2014, pp. 1–4.

[16] S. Bose, S. H. Low, T. Teeraratkul, and B. Hassibi, “Equivalent Relaxations of Optimal Power Flow,” *Autom. Control IEEE Trans. On*, vol. 60, no. 3, pp. 729–742, Mar. 2015.

[17] A. M. Eldurssi and R. M. O’Connell, “A Fast Nondominated Sorting Guided Genetic Algorithm for Multi-Objective Power Distribution System Reconfiguration Problem,” *Power Syst. IEEE Trans. On*, vol. 30, no. 2, pp. 593–601, Mar. 2015.

[18] M. Zhang and J. Chen, “The Energy Management and Optimized Operation of Electric Vehicles Based on Microgrid,” *IEEE Trans. Power Deliv.*, vol. 29, no. 3, pp. 1427–1435, Jun. 2014.

[19] C. Joe-Wong, S. Sen, S. Ha, and M. Chiang, “Optimized Day-Ahead Pricing for Smart Grids with Device-Specific Scheduling Flexibility,” *IEEE J. Sel. Areas Commun.*, vol. 30, no. 6, pp. 1075–1085, Jul. 2012.

[20] J. S. Vardakas, N. Zorba, and C. V. Verikoukis, “A Survey on Demand Response Programs in Smart Grids: Pricing Methods and Optimization Algorithms,” *IEEE Commun. Surv. Tutor.*, vol. 17, no. 1, pp. 152–178, Firstquarter 2015.

[21] G. Poyrazoglu and H. Oh, “Optimal Topology Control With Physical Power Flow Constraints and N-1 Contingency Criterion,” *IEEE Trans. Power Syst.*, vol. 30, no. 6, pp. 3063–3071, Nov. 2015.

[22] E. De Santis, L. Livi, F. M. F. Mascioli, A. Sadeghian, and A. Rizzi, "Fault recognition in smart grids by a one-class classification approach," in *2014 International Joint Conference on Neural Networks (IJCNN)*, 2014, pp. 1949–1956.

[23] W. Sun, P. Zhang, and Q. Zhou, "Optimization-based strategies towards a self-healing smart grid," in *2012 IEEE Innovative Smart Grid Technologies - Asia (ISGT Asia)*, 2012, pp. 1–6.

[24] R. C. Green, L. Wang, and M. Alam, "Applications and Trends of High Performance Computing for Electric Power Systems: Focusing on Smart Grid," *Smart Grid IEEE Trans. On*, vol. 4, no. 2, pp. 922–931, Jun. 2013.

[25] R. M. Ragnarsson, H. Stefánsson, and E. I. Ásgeirsson, "Meta-Heuristics in Multi-Core Environments," *Eng. Risk Manag.*, vol. 1, no. 0, pp. 457–464, 2011.

[26] G. A. Sena, D. Megherbi, and G. Isern, "Implementation of a parallel Genetic Algorithm on a cluster of workstations: Traveling Salesman Problem, a case study," *Workshop Bio-Inspired Solut. Parallel Comput. Probl.*, vol. 17, no. 4, pp. 477–488, Jan. 2001.

[27] N. Melab, S. Cahon, and E.-G. Talbi, "Grid computing for parallel bioinspired algorithms," *Spec. Issue Parallel Bioinspired Algorithms Spec. Issue Parallel Bioinspired Algorithms*, vol. 66, no. 8, pp. 1052–1061, Aug. 2006.

[28] J. Hiscock and D. Beauvais, "Smart Grid in Canada 2012-2013," Natural Resources Canada, Canada, 2013-171 RP-ANU 411-SGPLAN, Oct. 2013.

[29] "Canadian Electricity Association - Industry Data." [Online]. Available: http://www.electricity.ca/resources/industry-data.php. [Accessed: 12-Nov-2015].

[30] "U.S. Energy Information Administration (EIA) - Data." [Online]. Available: http://www.eia.gov/electricity/data.cfm. [Accessed: 12-Nov-2015].

[31] "Top 500 Supercomputer List Reflects Shifting State of Global HPC Trends," *The Next Platform*. [Online]. Available: http://www.nextplatform.com/2015/07/13/top-500-supercomputer-list-reflects-shifting-state-of-global-hpc-trends/. [Accessed: 12-Nov-2015].

[32] Yu Liu, Hoon Hong, and A. Q. Huang, "Real-Time Algorithm for Minimizing THD in Multilevel Inverters With Unequal or Varying Voltage Steps Under Staircase Modulation," *Ind. Electron. IEEE Trans. On*, vol. 56, no. 6, pp. 2249–2258, Jun. 2009.

[33] T. Rauber and G. Rünger, *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2010.

[34] M. G. Arenas, A. M. Mora, G. Romero, and P. A. Castillo, “GPU computation in bioinspired algorithms: a review,” in *Proceedings of the 11th international conference on Artificial neural networks conference on Advances in computational intelligence - Volume Part I*, Torremolinos-Mÿlaga, Spain, 2011, pp. 433–440.

[35] G. Luque and E. Alba, “Parallel Models for Genetic Algorithms,” in *Parallel Genetic Algorithms*, vol. 367, Springer Berlin Heidelberg, 2011, pp. 15–30.

[36] Madhuri and K. Deep, “A state-of-the-art review of population-based parallel meta-heuristics,” *Nat. Biol. Inspired Comput. 2009 NaBIC 2009 World Congr. On*, pp. 1604–1607, 2009.

[37] G. A. Laguna-Sánchez, M. Olguín-Carbajal, N. Cruz-Cortés, R. Barrón-Fernández, and J. Álvarez-Cedillo, “Comparative Study of Parallel Variants for a Particle Swarm Optimization Algorithm Implemented on a Multithreading GPU,” *J. Appl. Res. Technol.*, vol. 7, no. 3, pp. 292–309, 2010.

[38] M. Cárdenas-Montes, M. A. Vega-Rodríguez, J. J. Rodríguez-Vázquez, and A. Gómez-Iglesias, “Effect of the block occupancy in GPGPU over the performance of particle swarm algorithm,” in *Proceedings of the 10th international conference on Adaptive and natural computing algorithms - Volume Part I*, Ljubljana, Slovenia, 2011, pp. 310–319.

[39] M. Wahib, A. Munawar, M. Munetomo, and K. Akama, “Optimization of parallel Genetic Algorithms for nVidia GPUs,” *Evol. Comput. CEC 2011 IEEE Congr. On*, pp. 803–811, 2011.

[40] N. Melab, T.-V. Luong, K. Boufaras, and E.-G. Talbi, “Towards paradisEO-MO-GPU: a framework for GPU-based local search metaheuristics,” in *Proceedings of the 11th international conference on Artificial neural networks conference on Advances in computational intelligence - Volume Part I*, Torremolinos-Mÿlaga, Spain, 2011, pp. 401–408.

[41] N. Melab, T. V. Luong, K. Boufaras, and E.-G. Talbi, “ParadisEO-MO-GPU: a framework for parallel GPU-based local search metaheuristics,” in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, Amsterdam, The Netherlands, 2013, pp. 1189–1196.

[42] M. Ruciński, D. Izzo, and F. Biscani, “On the impact of the migration topology on the Island Model,” *Parallel Archit. Bioinspired Algorithms*, vol. 36, no. 10–11, pp. 555–571, Oct. 2010.

[43] D. Izzo, M. Ruciński, and F. Biscani, “The Generalized Island Model,” in *Parallel Architectures and Bioinspired Algorithms*, vol. 415, F. Fernández de Vega, J. I. Hidalgo Pérez, and J. Lanchares, Eds. Springer Berlin Heidelberg, 2012, pp. 151–169.

[44] V. Roberge, G. Labonté, and M. Tarbouchi, “Parallel Implementation and Comparison of Two UAV Path Planning Algorithms,” in *International Conference on Evolutionary Computation Theory and Applications*, Paris, 2011.

[45] V. Roberge, M. Tarbouchi, and G. Labonte, “Comparison of Parallel Genetic Algorithm and Particle Swarm Optimization for Real-Time UAV Path Planning,” *Ind. Inform. IEEE Trans. On*, vol. 9, no. 1, pp. 132–141, Feb. 2013.

[46] V. Roberge, M. Tarbouchi, and F. Allaire, “Parallel Hybrid Metaheuristic on Shared Memory System for Real-Time UAV Path Planning,” *Int. J. Comput. Intell. Appl.*, vol. 13, no. 2, pp. 1450008–1 – 1450008–16, Jun. 2014.

[47] V. Roberge and M. Tarbouchi, “Comparison of Parallel Metaheuristics for Flux Optimization for Induction Motor,” *World Sci. Eng. Acad. Soc. Trans. Power Syst.*, vol. 9, pp. 352–359, 2014.

[48] You Zhou and Ying Tan, “GPU-based parallel particle swarm optimization,” *Evol. Comput. 2009 CEC 09 IEEE Congr. On*, pp. 1493–1500, 2009.

[49] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of the 1995 IEEE International Conference on Neural Networks. Part 4 (of 6), Nov 27 - Dec 1 1995*, 1995, vol. 4, pp. 1942–1948.

[50] V. Roberge and M. Tarbouchi, “Parallel Particle Swarm Optimization on Graphical Processing Unit for Pose Estimation,” *World Sci. Eng. Acad. Soc. Trans. Comput.*, vol. 11, no. 6, pp. 170–179, Jun. 2012.

[51] V. Roberge and M. Tarbouchi, “Comparison of Parallel Particle Swarm Optimizers for Graphical Processing Units and Multicore Processors,” *Int. J. Comput. Intell. Appl.*, vol. 12, no. 01, p. 1350006, Mar. 2013.

[52] V. Roberge and M. Tarbouchi, “Efficient parallel Particle Swarm Optimizers on GPU for real-time harmonic minimization in multilevel inverters,” *IECON 2012 - 38th Annu. Conf. IEEE Ind. Electron. Soc.*, pp. 2275–2282, 2012.

[53] J. Hoberock and D. Tarjan, “NVIDIA Lecture Slides for Course CS193G,” Standford University, 2010.

[54] M. Oiso, T. Yasuda, K. Ohkura, and Y. Matumura, “Accelerating steady-state genetic algorithms based on CUDA architecture,” *Evol. Comput. CEC 2011 IEEE Congr. On*, pp. 687–692, 2011.

[55] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, “Fast in-place sorting with CUDA based on bitonic sort,” in *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, Wroclaw, Poland, 2010, pp. 403–410.

[56] E.-G. Talbi, *Metaheuristics: From Design to Implementation (Wiley Series on Parallel and Distributed Computing)*. Wiley, 2009.

[57] R. Arora, R. Tulshyan, and K. Deb, “Parallelization of binary and real-coded genetic algorithms on GPU using CUDA,” *Evol. Comput. CEC 2010 IEEE Congr. On*, pp. 1–8, 2010.

[58] Hongtao Bai, Dantong Ouyang, Ximing Li, Lili He, and HaiHong Yu, “MAX-MIN Ant System on GPU with CUDA,” *Innov. Comput. Inf. Control ICICIC 2009 Fourth Int. Conf. On*, pp. 801–804, 2009.

[59] H. Nguyen, *GPU Gems 3*. Addison-Wesley Professional, 2007.

[60] J. M. Cecilia, J. M. Garcia, M. Ujaldon, A. Nisbet, and M. Amos, “Parallelization strategies for ant colony optimisation on GPUs,” *Parallel Distrib. Process. Workshop Phd Forum IPDPSW 2011 IEEE Int. Symp. On*, pp. 339–346, 2011.

[61] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón, “Enhancing data parallelism for Ant Colony Optimization on GPUs,” *Metaheuristics GPUs*, vol. 73, no. 1, pp. 42–51, Jan. 2013.

[62] T. V. Luong, N. Melab, and E.-G. Talbi, “GPU-Based multi-start local search algorithms,” in *Proceedings of the 5th international conference on Learning and Intelligent Optimization*, Rome, Italy, 2011, pp. 321–335.

[63] Yiding Han, S. Roy, and K. Chakraborty, “Optimizing simulated annealing on GPU: A case study with IC floorplanning,” *Qual. Electron. Des. ISQED 2011 12th Int. Symp. On*, pp. 1–7, 2011.

[64] Hui Li and Chunmei Liu, “Prediction of Protein Structures Using GPU Based Simulated Annealing,” *Mach. Learn. Appl. ICMLA 2012 11th Int. Conf. On*, vol. 1, pp. 630–633, 2012.

[65] E. Bajrami, M. Asic, E. Cogo, D. Trnka, and N. Nosovic, “Performance comparison of simulated annealing algorithm execution on GPU and CPU,” *MIPRO 2012 Proc. 35th Int. Conv.*, pp. 1785–1788, 2012.

[66] Long Zheng, Yanchao Lu, Mengwei Ding, Yao Shen, Minyi Guoz, and Song Guo, “Architecture-based Performance Evaluation of Genetic Algorithms on Multi/Many-core Systems,” *Comput. Sci. Eng. CSE 2011 IEEE 14th Int. Conf. On*, pp. 321–334, 2011.

[67] Y. Sato, N. Hasegawa, and M. Sato, “GPU acceleration for Sudoku solution with genetic operations,” *Evol. Comput. CEC 2011 IEEE Congr. On*, pp. 296–303, 2011.

[68] J. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez, “Metaheuristic optimization frameworks: a survey and benchmarking,” *Soft Comput.*, vol. 16, no. 3, pp. 527–561, Mar. 2012.

[69] J. A. Parejo, J. Racero, F. Guerrero, T. Kwok, and K. A. Smith, "FOM: A Framework for Metaheuristic Optimization," in *Computational Science — ICCS 2003*, vol. 2660, P. A. Sloot, D. Abramson, A. Bogdanov, Y. Gorbachev, J. Dongarra, and A. Zomaya, Eds. Springer Berlin Heidelberg, 2003, pp. 886–895.

[70] R. Malek, "Collaboration of Metaheuristic Algorithms through a Multi-Agent System," in *Proceedings of the 4th International Conference on Industrial Applications of Holonic and Multi-Agent Systems: Holonic and Multi-Agent Systems for Manufacturing*, Linz, Austria, 2009, pp. 72–81.

[71] D. Sislak, M. Rehak, and M. Pechoucek, "A-globe: multi-agent platform with advanced simulation and visualization support," *Web Intell. 2005 Proc. 2005 IEEEWICACM Int. Conf. On*, pp. 805–808, Sep. 2005.

[72] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Adv Eng Softw*, vol. 42, no. 10, pp. 760–771, 2011.

[73] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4J: a modular framework for meta-heuristic optimization," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, Dublin, Ireland, 2011, pp. 1723–1730.

[74] J. Kim, M. Kim, M.-O. Stehr, H. Oh, and S. Ha, "A parallel and distributed meta-heuristic framework based on partially ordered knowledge sharing," *J. Parallel Distrib. Comput.*, vol. 72, no. 4, pp. 564–578, Apr. 2012.

[75] S. Cahon, N. Melab, and E.-G. Talbi, "ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics," *J. Heuristics*, vol. 10, no. 3, pp. 357–380, May 2004.

[76] L. S. Nathan, S. Karthik, and S. R. Krishna, "The 27-level multilevel inverter for solar PV applications," *Power Electron. IICPE 2012 IEEE 5th India Int. Conf. On*, pp. 1–6, Dec. 2012.

[77] A. K. Kaviani, S. H. Fathi, N. Farokhnia, and A. J. Ardakani, "PSO, an effective tool for harmonics elimination and optimization in multi-level inverters," in *2009 4th IEEE Conference on Industrial Electronics and Applications, 25-27 May 2009*, 2009, pp. 2902–7.

[78] M. Sabahi, A. Marami Iranaq, K. M. Bahrami, K. M. Bahrami, and M. B. B. Sharifian, "Harmonics elimination in a multilevel inverter with unequal DC sources using genetic algorithm," *Electr. Mach. Syst. ICEMS 2011 Int. Conf. On*, pp. 1–5, Aug. 2011.

[79] N. Flourentzou, V. G. Agelidis, and G. D. Demetriades, "VSC-Based HVDC Power Transmission Systems: An Overview," *IEEE Trans. Power Electron.*, vol. 24, no. 3, pp. 592–602, Mar. 2009.

[80] A. K. Verma, P. R. Thakura, K. C. Jana, and G. Buja, "Cascaded multilevel inverter for Hybrid Electric Vehicles," in *2010 India International Conference on Power Electronics (IICPE)*, 2011, pp. 1–6.

[81] F. Filho, L. M. Tolbert, Yue Cao, and B. Ozpineci, "Real-Time Selective Harmonic Minimization for Multilevel Inverters Connected to Solar Panels Using Artificial Neural Network Angle Generation," *Ind. Appl. IEEE Trans. On*, vol. 47, no. 5, pp. 2117–2124, Oct. 2011.

[82] J. Ewanchuk, J. Salmon, and A. M. Knight, "Performance of a High-Speed Motor Drive System Using a Novel Multilevel Inverter Topology," *IEEE Trans. Ind. Appl.*, vol. 45, no. 5, pp. 1706–1714, Sep. 2009.

[83] P. Sotoodeh and R. D. Miller, "Design and Implementation of an 11-Level Inverter With FACTS Capability for Distributed Energy Systems," *IEEE J. Emerg. Sel. Top. Power Electron.*, vol. 2, no. 1, pp. 87–96, Mar. 2014.

[84] N. Hatti, K. Hasegawa, and H. Akagi, "A 6.6-kV Transformerless Motor Drive Using a Five-Level Diode-Clamped PWM Inverter for Energy Savings of Pumps and Blowers," *IEEE Trans. Power Electron.*, vol. 24, no. 3, pp. 796–803, Mar. 2009.

[85] A. K. Sadigh, S. H. Hosseini, M. Sabahi, and G. B. Gharehpetian, "Double Flying Capacitor Multicell Converter Based on Modified Phase-Shifted Pulsewidth Modulation," *IEEE Trans. Power Electron.*, vol. 25, no. 6, pp. 1517–1526, Jun. 2010.

[86] A. Nami, F. Zare, A. Ghosh, and F. Blaabjerg, "A Hybrid Cascade Converter Topology With Series-Connected Symmetrical and Asymmetrical Diode-Clamped H-Bridge Cells," *IEEE Trans. Power Electron.*, vol. 26, no. 1, pp. 51–65, Jan. 2011.

[87] R. M. Hossam, G. M. Hashem, and M. I. Marei, "Optimized harmonic elimination for cascaded multilevel inverter," in *Power Engineering Conference (UPEC), 2013 48th International Universities'*, 2013, pp. 1–6.

[88] "Harmonics in power systems - Causes, effects and control." Siemens, 2013.

[89] G. Nageswara Rao, P. Sangameswara Raju, and K. Chandra Sekhar, "Harmonic elimination of cascaded H-bridge multilevel inverter based active power filter controlled by intelligent techniques," *Int. J. Electr. Power Energy Syst.*, vol. 61, pp. 56–63, Oct. 2014.

[90] M. G. Hosseini Aghdam, S. H. Fathi, and G. B. Gharehpetian, "Comparison of OMTHD and OHSW harmonic optimization techniques in multi-level voltage-source inverter with non-equal DC sources," in *7th Internatonal Conference on Power Electronics, 2007. ICPE '07*, 2007, pp. 587–591.

[91] W. Fei, X. Ruan, and B. Wu, “A Generalized Formulation of Quarter-Wave Symmetry SHE-PWM Problems for Multilevel Inverters,” *IEEE Trans. Power Electron.*, vol. 24, no. 7, pp. 1758–1766, Jul. 2009.

[92] R. Saravanakumar, R. M. Anusuya, V. Kavitha, and A. Gopi, “Selective harmonic elimination in seven level cascaded inverter,” in *2013 International Conference on Renewable Energy and Sustainable Energy (ICRESE)*, 2013, pp. 51–57.

[93] J. Wang and D. Ahmadi, “A Precise and Practical Harmonic Elimination Method for Multilevel Inverters,” *IEEE Trans. Ind. Appl.*, vol. 46, no. 2, pp. 857–865, Mar. 2010.

[94] T. Tang, J. Han, and X. Tan, “Selective Harmonic Elimination for a Cascade Multilevel Inverter,” in *2006 IEEE International Symposium on Industrial Electronics*, 2006, vol. 2, pp. 977–981.

[95] L. M. Tolbert, J. N. Chiasson, Z. Du, and K. J. McKenzie, “Elimination of harmonics in a multilevel converter with nonequal DC sources,” *IEEE Trans. Ind. Appl.*, vol. 41, no. 1, pp. 75–82, Jan. 2005.

[96] Z. Du, L. M. Tolbert, J. N. Chiasson, and B. Ozpineci, “Reduced Switching-Frequency Active Harmonic Elimination for Multilevel Converters,” *IEEE Trans. Ind. Electron.*, vol. 55, no. 4, pp. 1761–1770, Apr. 2008.

[97] J. N. Chiasson, L. M. Tolbert, K. J. McKenzie, and Z. Du, “Elimination of harmonics in a multilevel converter using the theory of symmetric polynomials and resultants,” *IEEE Trans. Control Syst. Technol.*, vol. 13, no. 2, pp. 216–223, Mar. 2005.

[98] B. Ozpineci, L. M. Tolbert, and J. N. Chiasson, “Harmonic optimization of multilevel converters using genetic algorithms,” *IEEE Power Electron. Lett.*, vol. 3, no. 3, pp. 92–95, Sep. 2005.

[99] M. H. Etesami, N. Farokhnia, and S. H. Fathi, “Colonial Competitive Algorithm Development toward Harmonic Minimization in Multilevel Inverters,” *Ind. Inform. IEEE Trans. On*, vol. PP, no. 99, pp. 1–1, 2015.

[100] Yu Liu, Hoon Hong, and A. Q. Huang, “Real-Time Calculation of Switching Angles Minimizing THD for Multilevel Inverters With Step Modulation,” *Ind. Electron. IEEE Trans. On*, vol. 56, no. 2, pp. 285–293, Feb. 2009.

[101] H. Taghizadeh and M. T. Hagh, “Harmonic Elimination of Cascade Multilevel Inverters with Nonequal DC Sources Using Particle Swarm Optimization,” *Ind. Electron. IEEE Trans. On*, vol. 57, no. 11, pp. 3678–3684, Nov. 2010.

[102] V. Roberge, M. Tarbouchi, and G. Labonte, “Parallel Algorithm on Graphics Processing Unit for Harmonic Minimization in Multilevel Inverters,” *Ind. Inform. IEEE Trans. On*, vol. 11, no. 3, pp. 700–707, Jun. 2015.

[103] F. Filho, H. Z. Maia, T. H. A. Mateus, B. Ozpineci, L. M. Tolbert, and J. O. P. Pinto, “Adaptive Selective Harmonic Minimization Based on ANNs for Cascade Multilevel Inverters With Varying DC Sources,” *Ind. Electron. IEEE Trans. On*, vol. 60, no. 5, pp. 1955–1962, May 2013.

[104] V. Roberge, M. Tarbouchi, and F. Okou, “Strategies to Accelerate Harmonic Minimization in Multilevel Inverters Using a Parallel Genetic Algorithm on Graphical Processing Unit,” *Power Electron. IEEE Trans. On*, vol. 29, no. 10, pp. 5087–5090, Oct. 2014.

[105] M. T. Hagh, H. Taghizadeh, and K. Razi, “Harmonic Minimization in Multilevel Inverters Using Modified Species-Based Particle Swarm Optimization,” *IEEE Trans. Power Electron.*, vol. 24, no. 10, pp. 2259–2267, Oct. 2009.

[106] N. Yousefpoor, S. H. Fathi, N. Farokhnia, and H. A. Abyaneh, “THD Minimization Applied Directly on the Line-to-Line Voltage of Multilevel Inverters,” *Ind. Electron. IEEE Trans. On*, vol. 59, no. 1, pp. 373–380, Jan. 2012.

[107] F. Nawaz, M. Yaqoob, Z. Ming, and M. T. Ali, “Low order harmonics minimization in multilevel inverters using firefly algorithm,” in *Power and Energy Engineering Conference (APPEEC), 2013 IEEE PES Asia-Pacific*, 2013, pp. 1–6.

[108] A. Niknam Kumle, S. H. Fathi, F. Jabbarvaziri, M. Jamshidi, and S. S. Heidari Yazdi, “Application of memetic algorithm for selective harmonic elimination in multi-level inverters,” *IET Power Electron.*, vol. 8, no. 9, pp. 1733–1739, 2015.

[109] J. Carpentier, “Contribution à l’étude du dispatching économique,” *Bull. Société Fr. Électriciens*, vol. 3, pp. 431–447, 1962.

[110] N. Amjady, H. Fatemi, and H. Zareipour, “Solution of Optimal Power Flow Subject to Security Constraints by a New Improved Bacterial Foraging Method,” *Power Syst. IEEE Trans. On*, vol. 27, no. 3, pp. 1311–1323, Aug. 2012.

[111] S. K. M. Kodsi and C. Canizares, “Modeling and Simulation of IEEE 14 Bus System With FACTS Controllers,” ECE Dept, University of Waterloo, Waterloo, ON, 2003-3, 2003.

[112] S. Frank, I. Steponavice, and S. Rebennack, “Optimal power flow: a bibliographic survey I,” *Energy Syst.*, vol. 3, no. 3, pp. 221–258, Sep. 2012.

[113] S. Frank, I. Steponavice, and S. Rebennack, “Optimal power flow: a bibliographic survey II,” *Energy Syst.*, vol. 3, no. 3, pp. 259–289, Sep. 2012.

[114] S.-H. Soliman and A.-A. Mantawy, “Optimal Power Flow,” in *Modern Optimization Techniques with Applications in Electric Power Systems*, Springer New York, 2012, pp. 281–346.

[115] A. G. Bakirtzis, P. N. Biskas, C. E. Zoumas, and V. Petridis, “Optimal power flow by enhanced genetic algorithm,” *Power Syst. IEEE Trans. On*, vol. 17, no. 2, pp. 229–236, May 2002.

[116] Jun Qiang Wu and A. Bose, “Parallel solution of large sparse matrix equations and parallel power flow,” *Power Syst. IEEE Trans. On*, vol. 10, no. 3, pp. 1343–1349, 1995.

[117] Limin Ao, Bin Cheng, and Feifei Li, “Research of Power Flow Parallel Computing Based on MPI and P-Q Decomposition Method,” *Electr. Control Eng. ICECE 2010 Int. Conf. On*, pp. 2925–2928, 2010.

[118] X. Wang, S. G. Ziavras, C. Nwankpa, J. Johnson, and P. Nagvajara, “Parallel solution of Newton’s power flow equations on configurable chips,” *Int. J. Electr. Power Energy Syst.*, vol. 29, no. 5, pp. 422–431, Jun. 2007.

[119] P. Blaskiewicz, M. Zawada, P. Balcerek, and P. Dawidowski, “An Application of GPU Parallel Computing to Power Flow Calculation in HVDC Networks,” *Parallel Distrib. Netw.-Based Process. PDP 2015 23rd Euromicro Int. Conf. On*, pp. 635–641, Mar. 2015.

[120] J. K. Debnath, Wai-Keung Fung, A. M. Gole, and S. Filizadeh, “Electromagnetic transient simulation of large-scale electrical power networks using graphics processing units,” *Electr. Comput. Eng. CCECE 2012 25th IEEE Can. Conf. On*, pp. 1–4, Apr. 2012.

[121] X.-X. Liu, S. X.-D. Tan, Z. Liu, H. Wang, and T. Xu, “Transient analysis of large linear dynamic networks on hybrid GPU-multicore platforms,” in *New Circuits and Systems Conference (NEWCAS), 2012 IEEE 10th International*, 2012, pp. 173–176.

[122] Z. Zhou and V. Dinavahi, “Parallel Massive-Thread Electromagnetic Transient Simulation on GPU,” *IEEE Trans. Power Deliv.*, vol. 29, no. 3, pp. 1045–1053, Jun. 2014.

[123] W. Baijian, G. Wenxin, H. Jiayi, W. Fangzong, and Y. Jing, “GPU based parallel simulation of transient stability using symplectic Gauss algorithm and preconditioned GMRES method,” in *2012 IEEE Power Engineering and Automation Conference (PEAM)*, 2012, pp. 1–5.

[124] V. Jalili-Marandi and V. Dinavahi, “Large-scale transient stability simulation on graphics processing units,” in *IEEE Power Energy Society General Meeting, 2009. PES ’09*, 2009, pp. 1–6.

[125] V. Jalili-Marandi and V. Dinavahi, “SIMD-Based Large-Scale Transient Stability Simulation on the Graphics Processing Unit,” *IEEE Trans. Power Syst.*, vol. 25, no. 3, pp. 1589–1599, Aug. 2010.

[126] V. Jalili-Marandi, Z. Zhou, and V. Dinavahi, “Large-Scale Transient Stability Simulation of Electrical Power Systems on Parallel GPUs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 7, pp. 1255–1266, Jul. 2012.

[127] Z. Feng, X. Zhao, and Z. Zeng, “Robust Parallel Preconditioned Power Grid Simulation on GPU With Adaptive Runtime Performance Modeling and Optimization,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 30, no. 4, pp. 562–573, Apr. 2011.

[128] X.-X. Liu, H. Wang, and S. X.-D. Tan, “Parallel power grid analysis using preconditioned GMRES solver on CPU-GPU platforms,” in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 561–568.

[129] Xiaoming Chen, Ling Ren, Yu Wang, and Huazhong Yang, “GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling,” *Parallel Distrib. Syst. IEEE Trans. On*, vol. 26, no. 3, pp. 786–795, Mar. 2015.

[130] K. He, S. X.-. Tan, H. Wang, and G. Shi, “GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis,” *Very Large Scale Integr. VLSI Syst. IEEE Trans. On*, vol. PP, no. 99, pp. 1–1, 2015.

[131] Zhao Li, Jinxiang Zhu, and Fang Yang, “How far is the GPU technology from practical power system applications?,” *PES Gen. Meet. Conf. Expo. 2014 IEEE*, pp. 1–5, Jul. 2014.

[132] D. Ablakovic, I. Dzafic, and S. Kecici, “Parallelization of radial three-phase distribution power flow using GPU,” *Innov. Smart Grid Technol. ISGT Eur. 2012 3rd IEEE PES Int. Conf. Exhib. On*, pp. 1–7, Oct. 2012.

[133] J. Singh and I. Aruni, “Accelerating Power Flow studies on Graphics Processing Unit,” *India Conf. INDICON 2010 Annu. IEEE*, pp. 1–5, Dec. 2010.

[134] C. Vilacha, J. Moreira, E. Miguez, and A. Otero, “Massive Jacobi power flow based on SIMD-processor,” presented at the Environment and Electrical Engineering (EEEIC), 2011 10th International Conference on, 2011, pp. 1–4.

[135] Chunhui Guo, Baochen Jiang, Hao Yuan, Zhiqiang Yang, Li Wang, and Shangping Ren, “Performance Comparisons of Parallel Power Flow Solvers on GPU System,” *Embed. Real-Time Comput. Syst. Appl. RTCSA 2012 IEEE 18th Int. Conf. On*, pp. 232–239, Aug. 2012.

[136] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "MATPOWER: Steady-State Operations, Planning, and Analysis Tools for Power Systems Research and Education," *Power Syst. IEEE Trans. On*, vol. 26, no. 1, pp. 12–19, Feb. 2011.

[137] V. Roberge, M. Tarbouchi, and F. Okou, "Parallel Power Flow Analysis on Graphics Processing Units for Concurrent Evaluation of Many Networks," *IEEE Trans. Smart Grid*, vol. PP, no. 99, pp. 1–10, 2015.

[138] Zhao Li, V. D. Donde, J.-C. Tournier, and Fang Yang, "On limitations of traditional multi-core and potential of many-core processing architectures for sparse linear solvers used in large-scale power system applications," *Power Energy Soc. Gen. Meet. 2011 IEEE*, pp. 1–8, Jul. 2011.

[139] A. Gopal, D. Niebur, and S. Venkatasubramanian, "DC Power Flow Based Contingency Analysis Using Graphics Processing Units," *Power Tech 2007 IEEE Lausanne*, pp. 731–736, Jul. 2007.

[140] N. Garcia, "Parallel power flow solutions using a biconjugate gradient algorithm and a Newton method: A GPU-based approach," *Power Energy Soc. Gen. Meet. 2010 IEEE*, pp. 1–4, Jul. 2010.

[141] X. Li and F. Li, "GPU-based power flow analysis with Chebyshev preconditioner and conjugate gradient method," *Electr. Power Syst. Res.*, vol. 116, no. 0, pp. 87–93, Nov. 2014.

[142] D. Kim and K.-H. Park, "Tiled QR Decomposition and Its Optimization on CPU and GPU Computing System," in *2013 42nd International Conference on Parallel Processing (ICPP)*, 2013, pp. 744–753.

[143] R. Andrew and N. Dingle, "Implementing QR factorization updating algorithms on GPUs," *Parallel Comput.*, vol. 40, no. 7, pp. 161–172, Jul. 2014.

[144] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov, "LU factorization for accelerator-based systems," in *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, 2011, pp. 217–224.

[145] Y. Jia, P. Luszczek, and J. Dongarra, "Multi-GPU Implementation of LU Factorization," *Procedia Comput. Sci.*, vol. 9, pp. 106–115, 2012.

[146] E. D'Azevedo and J. C. Hill, "Parallel LU Factorization on GPU Cluster," *Procedia Comput. Sci.*, vol. 9, pp. 67–75, 2012.

[147] O. Schenk, M. Christen, and H. Burkhart, "Algorithmic performance studies on graphics processing units," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1360–1369, Oct. 2008.

[148] A. Tumeo, N. Gawande, and O. Villa, “A Flexible CUDA LU-Based Solver for Small, Batched Linear Systems,” in *Numerical Computations with GPUs*, V. Kindratenko, Ed. Springer International Publishing, 2014, pp. 87–101.

[149] X. Wang and S. G. Ziavras, “A multiprocessor-on-a-programmable-chip reconfigurable system for matrix operations with power-grid case studies,” *Int J Comput Sci Eng*, vol. 10, no. 1/2, pp. 181–191, 2015.

[150] A. Sangiovanni-Vincentelli, L.-K. Chen, and L. O. Chua, “An efficient heuristic cluster algorithm for tearing large-scale networks,” *IEEE Trans. Circuits Syst.*, vol. 24, no. 12, pp. 709–717, Dec. 1977.

[151] L. Qian, D. Zhou, X. Zeng, F. Yang, and S. Wang, “A parallel sparse linear system solver for large-scale circuit simulation based on Schur Complement,” in *2013 IEEE 10th International Conference on ASIC (ASICON)*, 2013, pp. 1–4.

[152] S. Rajamanickam, E. G. Boman, and M. A. Heroux, “ShyLU: A Hybrid-Hybrid Solver for Multicore Platforms,” in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 631–643.

[153] X. Wang, “Hardware–software optimizations of reconfigurable multi-core processors for floating-point computations of large sparse matrices,” *J. Real-Time Image Process.*, vol. 9, no. 1, pp. 187–204, Mar. 2014.

[154] Q. Y. Jiang, H.-D. Chiang, C. X. Guo, and Y. J. Cao, “Power-current hybrid rectangular formulation for interior-point optimal power flow,” *Gener. Transm. Distrib. IET*, vol. 3, no. 8, pp. 748–756, Aug. 2009.

[155] S. H. Low, “Convex Relaxation of Optimal Power Flow—Part I: Formulations and Equivalence,” *Control Netw. Syst. IEEE Trans. On*, vol. 1, no. 1, pp. 15–27, Mar. 2014.

[156] J. Lavaei and S. H. Low, “Zero Duality Gap in Optimal Power Flow Problem,” *Power Syst. IEEE Trans. On*, vol. 27, no. 1, pp. 92–107, Feb. 2012.

[157] B. C. Lesieutre, D. K. Molzahn, A. R. Borden, and C. L. DeMarco, “Examining the limits of the application of semidefinite programming to power flow problems,” *Commun. Control Comput. Allerton 2011 49th Annu. Allerton Conf. On*, pp. 1492–1499, Sep. 2011.

[158] R. Madani, S. Sojoudi, and J. Lavaei, “Convex Relaxation for Optimal Power Flow Problem: Mesh Networks,” *Power Syst. IEEE Trans. On*, vol. 30, no. 1, pp. 199–211, Jan. 2015.

[159] B. Kocuk, S. S. Dey, and X. A. Sun, “Inexactness of SDP Relaxation and Valid Inequalities for Optimal Power Flow,” *Power Syst. IEEE Trans. On*, vol. PP, no. 99, pp. 1–10, 2015.

[160] C. Josz, J. Maeght, P. Panciatici, and J. C. Gilbert, “Application of the Moment-SOS Approach to Global Optimization of the OPF Problem,” *Power Syst. IEEE Trans. On*, vol. 30, no. 1, pp. 463–470, Jan. 2015.

[161] A. D. Papalexopoulos, C. F. Imparato, and F. F. Wu, “Large-scale optimal power flow: effects of initialization, decoupling and discretization,” *Power Syst. IEEE Trans. On*, vol. 4, no. 2, pp. 748–759, May 1989.

[162] F. Capitanescu and L. Wehenkel, “Sensitivity-Based Approaches for Handling Discrete Variables in Optimal Power Flow Computations,” *Power Syst. IEEE Trans. On*, vol. 25, no. 4, pp. 1780–1789, Nov. 2010.

[163] K. Karoui, L. Platbrood, H. Crisciu, and R. A. Waltz, “New large-scale security constrained optimal power flow program using a new interior point algorithm,” *Electr. Mark. 2008 EEM 2008 5th Int. Conf. Eur.*, pp. 1–6, May 2008.

[164] P. J. Macfie, G. A. Taylor, M. R. Irving, P. Hurlock, and Hai-Bin Wan, “Proposed Shunt Rounding Technique for Large-Scale Security Constrained Loss Minimization,” *Power Syst. IEEE Trans. On*, vol. 25, no. 3, pp. 1478–1485, Aug. 2010.

[165] W. Murray, T. T. De Rubira, and A. Wigington, “Optimal power flow with limited and discrete controls,” *Innov. Smart Grid Technol. Conf. ISGT 2015 IEEE Power Energy Soc.*, pp. 1–5, Feb. 2015.

[166] V. Radziukynas and I. Radziukyniene, “Optimization Methods Application to Optimal Power Flow in Electric Power Systems,” in *Optimization in the Energy Industry*, J. Kallrath, P. Pardalos, S. Rebennack, and M. Scheidt, Eds. Springer Berlin Heidelberg, 2009, pp. 409–436.

[167] T. Sousa, J. Soares, Z. A. Vale, H. Morais, and P. Faria, “Simulated Annealing metaheuristic to solve the optimal power flow,” *Power Energy Soc. Gen. Meet. 2011 IEEE*, pp. 1–8, Jul. 2011.

[168] H. R. E. H. Bouchekara, M. A. Abido, and M. Boucherma, “Optimal power flow using Teaching-Learning-Based Optimization technique,” *Electr. Power Syst. Res.*, vol. 114, no. 0, pp. 49–59, Sep. 2014.

[169] B. Mahdad and K. Srairi, “Multi objective large power system planning under sever loading condition using learning DE-APSO-PS strategy,” *Energy Convers. Manag.*, vol. 87, no. 0, pp. 338–350, Nov. 2014.

[170] B. Mahdad, K. Srairi, and T. Bouktir, “Optimal power flow for large-scale power system with shunt FACTS using fast parallel GA,” in *Electrotechnical Conference, 2008. MELECON 2008. The 14th IEEE Mediterranean*, 2008, pp. 669–676.

[171] Y. Fukuyama, “Parallel particle swarm optimization for reactive power and voltage control verifying dependability,” in *2015 IEEE Congress on Evolutionary Computation (CEC)*, 2015, pp. 304–310.

[172] C. H. Lo, C. Y. Chung, D. H. M. Nguyen, and K. P. Wong, “Parallel evolutionary programming for optimal power flow,” in *Proceedings of the 2004 IEEE International Conference on Electric Utility Deregulation, Restructuring and Power Technologies, 2004. (DRPT 2004)*, 2004, vol. 1, pp. 190–195 Vol.1.

[173] J.-Y. Kim, K.-J. Mun, H.-S. Kim, and J. H. Park, “Optimal power system operation using parallel processing system and PSO algorithm,” *Int. J. Electr. Power Energy Syst.*, vol. 33, no. 8, pp. 1457–1461, Oct. 2011.

[174] C.-J. Ye and M.-X. Huang, “Multi-Objective Optimal Power Flow Considering Transient Stability Based on Parallel NSGA-II,” *Power Syst. IEEE Trans. On*, vol. 30, no. 2, pp. 857–866, Mar. 2015.

[175] L. Rakai and W. Rosehart, “GPU-Accelerated Solutions to Optimal Power Flow Problems,” *Syst. Sci. HICSS 2014 47th Hawaii Int. Conf. On*, pp. 2511–2516, Jan. 2014.

[176] M. E. Baran and F. F. Wu, “Network reconfiguration in distribution systems for loss reduction and load balancing,” *Power Deliv. IEEE Trans. On*, vol. 4, no. 2, pp. 1401–1407, 1989.

[177] L. Tang, F. Yang, and J. Ma, “A Survey on Distribution System Feeder Reconfiguration: Objectives and Solutions,” in *IEEE Innovative Smart Grid Technologies (ISGT)*, Kuala Lumpur, Malaysia, 2014.

[178] Q. Peng and S. H. Low, “Optimal Branch Exchange for Distribution System Reconfiguration,” *Eprint ArXiv*, Sep. 2013.

[179] Fei Ding and K. A. Loparo, “A simple heuristic method for smart distribution system reconfiguration,” *Energytech 2012 IEEE*, pp. 1–6, May 2012.

[180] R. A. Jabr, R. Singh, and B. C. Pal, “Minimum Loss Network Reconfiguration Using Mixed-Integer Convex Programming,” *Power Syst. IEEE Trans. On*, vol. 27, no. 2, pp. 1106–1115, May 2012.

[181] J. A. Taylor and F. S. Hover, “Convex Models of Distribution System Reconfiguration,” *Power Syst. IEEE Trans. On*, vol. 27, no. 3, pp. 1407–1413, Aug. 2012.

[182] H. Ahmadi and J. R. Marti, “Distribution System Optimization Based on a Linear Power-Flow Formulation,” *Power Deliv. IEEE Trans. On*, vol. 30, no. 1, pp. 25–33, Feb. 2015.

[183] H. M. Khodr, J. Martinez-Crespo, M. A. Matos, and J. Pereira, “Distribution Systems Reconfiguration Based on OPF Using Benders Decomposition,” *Power Deliv. IEEE Trans. On*, vol. 24, no. 4, pp. 2166–2176, Oct. 2009.

[184] Wu-Chang Wu and Men-Shen Tsai, “Application of Enhanced Integer Coded Particle Swarm Optimization for Distribution System Feeder Reconfiguration,” *Power Syst. IEEE Trans. On*, vol. 26, no. 3, pp. 1591–1599, Aug. 2011.

[185] B. Enacheanu, B. Raison, R. Caire, O. Devaux, W. Bienia, and N. HadjSaid, “Radial Network Reconfiguration Using Genetic Algorithm Based on the Matroid Theory,” *Power Syst. IEEE Trans. On*, vol. 23, no. 1, pp. 186–195, Feb. 2008.

[186] B. Tomoiagă, M. Chindriş, A. Sumper, R. Villafafila-Robles, and A. Sudria-Andreu, “Distribution system reconfiguration using genetic algorithm based on connected graphs,” *Electr. Power Syst. Res.*, vol. 104, no. 0, pp. 216–225, Nov. 2013.

[187] Chung-Fu Chang, “Reconfiguration and Capacitor Placement for Loss Reduction of Distribution Systems by Ant Colony Search Algorithm,” *Power Syst. IEEE Trans. On*, vol. 23, no. 4, pp. 1747–1755, Nov. 2008.

[188] L. W. de Oliveira, E. J. de Oliveira, F. V. Gomes, I. C. Silva Jr., A. L. M. Marcato, and P. V. C. Resende, “Artificial Immune Systems applied to the reconfiguration of electrical power distribution networks for energy loss minimization,” *Int. J. Electr. Power Energy Syst.*, vol. 56, no. 0, pp. 64–74, Mar. 2014.

[189] H. B. Tolabi, M. H. Ali, Shahrin Bin Md Ayob, and M. Rizwan, “Novel hybrid fuzzy-Bees algorithm for optimal feeder multi-objective reconfiguration by considering multiple-distributed generation,” *Energy*, vol. 71, no. 0, pp. 507–515, Jul. 2014.

[190] S. H. Mirhoseini, S. M. Hosseini, M. Ghanbari, and M. Ahmadi, “A new improved adaptive imperialist competitive algorithm to solve the reconfiguration problem of distribution systems for loss reduction and voltage profile improvement,” *Int. J. Electr. Power Energy Syst.*, vol. 55, no. 0, pp. 128–143, Feb. 2014.

[191] S. Kalambe and G. Agnihotri, “Loss minimization techniques used in distribution network: bibliographical survey,” *Renew. Sustain. Energy Rev.*, vol. 29, no. 0, pp. 184–200, Jan. 2014.

[192] B. Amanulla, S. Chakrabarti, and S. N. Singh, “Reconfiguration of Power Distribution Systems Considering Reliability and Power Loss,” *Power Deliv. IEEE Trans. On*, vol. 27, no. 2, pp. 918–926, Apr. 2012.

[193] V. Farahani, S. H. H. Sadeghi, H. Askarian, and K. Mazlumi, “An improved reconfiguration method for maximum loss reduction using Discrete Genetic algorithm,” *Power Eng. Optim. Conf. PEOCO 2010 4th Int.*, pp. 178–183, Jun. 2010.

[194] J. Torres, J. L. Guardado, F. Rivas-Dávalos, S. Maximov, and E. Melgoza, "A genetic algorithm based on the edge window decoder technique to optimize power distribution systems reconfiguration," *Int. J. Electr. Power Energy Syst.*, vol. 45, no. 1, pp. 28–34, Feb. 2013.

[195] A. Mazza, G. Chicco, and A. Russo, "Optimal multi-objective distribution system reconfiguration with multi criteria decision making-based solution ranking and enhanced genetic operators," *Int. J. Electr. Power Energy Syst.*, vol. 54, no. 0, pp. 255–267, Jan. 2014.

[196] X.-S. Yang, *Engineering Optimization: An Introduction with Metaheuristic Applications*. New Jersey: Wiley, 2010.

[197] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer Math*, vol. 1, no. 1, pp. 269–271, 1959.

[198] S.-Y. Lin, Y.-C. Ho, and C. 'i-Hsin Lin, "An ordinal optimization theory-based algorithm for solving the optimal power flow problem with discrete control variables," *IEEE Trans. Power Syst.*, vol. 19, no. 1, pp. 276–286, Feb. 2004.

[199] L. Gan and S. H. Low, "Optimal Power Flow in Direct Current Networks," *IEEE Trans. Power Syst.*, vol. 29, no. 6, pp. 2892–2904, Nov. 2014.

[200] S. Magnusson, P. C. Weeraddana, and C. Fischione, "A Distributed Approach for the Optimal Power-Flow Problem Based on ADMM and Sequential Convex Approximations," *IEEE Trans. Control Netw. Syst.*, vol. 2, no. 3, pp. 238–253, Sep. 2015.

[201] Q. Peng, Y. Tang, and S. H. Low, "Feeder Reconfiguration in Distribution Networks Based on Convex Relaxation of OPF," *IEEE Trans. Power Syst.*, vol. 30, no. 4, pp. 1793–1804, Jul. 2015.

[202] A. Ahuja, S. Das, and A. Pahwa, "An AIS-ACO Hybrid Approach for Multi-Objective Distribution System Reconfiguration," in *Computational Intelligence in Power Engineering*, vol. 302, B. Panigrahi, A. Abraham, and S. Das, Eds. Springer Berlin Heidelberg, 2010, pp. 19–73.

[203] M. Clerc, *Particle Swarm Optimization*. ISTE Publishing Company, 2006.

[204] A. Khare and S. Rangnekar, "Particle swarm optimization: A review," *Appl. Soft Comput.*, 2013.

[205] K. E. Parsopoulos and M. N. Vrahatis, *Particle Swarm Optimization and Intelligence: Advances and Applications*. IGI Global, 2009.

[206] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," *Evol. Comput. Proc. 1998 IEEE World Congr. Comput. Intell. 1998 IEEE Int. Conf. On*, pp. 69–73, May 1998.

[207] T. O. Ting, Y. Shi, S. Cheng, and S. Lee, “Exponential inertia weight for particle swarm optimization,” in *Proceedings of the Third international conference on Advances in Swarm Intelligence - Volume Part I*, Shenzhen, China, 2012, pp. 83–90.

[208] K. Deep, M. Arya, and J. C. Bansal, “A non-deterministic adaptive inertia weight in PSO,” in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, Dublin, Ireland, 2011, pp. 1155–1162.

[209] M. Clerc and J. Kennedy, “The particle swarm - explosion, stability, and convergence in a multidimensional complex space,” *Evol. Comput. IEEE Trans. On*, vol. 6, no. 1, pp. 58–73, 2002.

[210] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[211] X. Yu and M. Gen, *Introduction to Evolutionary Algorithms*. Springer London, 2010.

[212] H. Xie and M. Zhang, “Parent Selection Pressure Auto-Tuning for Tournament Selection in Genetic Programming,” *Evol. Comput. IEEE Trans. On*, vol. 17, no. 1, pp. 1–19, Feb. 2013.

[213] H. A. Bashir and R. S. Neville, “A hybrid evolutionary computation algorithm for global optimization,” *Evol. Comput. CEC 2012 IEEE Congr. On*, pp. 1–8, 2012.

[214] Hui Cheng and Shengxiang Yang, “Genetic algorithms with elitism-based immigrants for dynamic load balanced clustering problem in mobile ad hoc networks,” *Comput. Intell. Dyn. Uncertain Environ. CIDUE 2011 IEEE Symp. On*, pp. 1–7, 2011.

[215] Zhao Xin and Xiu Chunbo, “New genetic algorithm improved and its applications,” *Electron. Commun. Control ICECC 2011 Int. Conf. On*, pp. 926–928, 2011.

[216] Y. Fu, M. Ding, C. Zhou, and H. Hu, “Route Planning for Unmanned Aerial Vehicle (UAV) on the Sea Using Hybrid Differential Evolution and Quantum-Behaved Particle Swarm Optimization,” *Syst. Man Cybern. Syst. IEEE Trans. On*, vol. PP, no. 99, pp. 1–1, 2013.

[217] N. Singh and B. Ghosh, “Simulated annealing Vs. Genetic Simulated Annealing for automatic transistor sizing,” *Semicond. Electron. ICSE 2012 10th IEEE Int. Conf. On*, pp. 478–481, 2012.

[218] D. H. Wolpert and W. G. Macready, “No free lunch theorems for optimization,” *Evol. Comput. IEEE Trans. On*, vol. 1, no. 1, pp. 67–82, Apr. 1997.

[219] C. Li and S. Yang, “An Island Based Hybrid Evolutionary Algorithm for Optimization,” in *Simulated Evolution and Learning*, vol. 5361, X. Li, M. Kirley, M. Zhang, D. Green, V. Ciesielski, H. Abbass, Z. Michalewicz, T. Hendtlass, K. Deb, K. Tan, J. Branke, and Y. Shi, Eds. Springer Berlin Heidelberg, 2008, pp. 180–189.

[220] A. Munawar, M. Wahib, M. Munetomo, and K. Akama, “Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework,” *Genet. Program. Evolvable Mach.*, vol. 10, no. 4, pp. 391–415, Dec. 2009.

[221] “NVIDIA GeForce GTX 750 Ti.” [Online]. Available: http://www.nvidia.com/gtx-700-graphics-cards/gtx-750ti/. [Accessed: 30-Jun-2015].

[222] “GeForce GTX TITAN.” [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan. [Accessed: 22-Feb-2013].

[223] “Tesla K20 GPU Active Accelerator.” NVIDIA Corporation, Jan-2013.

[224] G. Moore, “Cramming More Components onto Integrated Circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.

[225] H. Sutter, “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,” 2009. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm. [Accessed: 23-Jan-2013].

[226] Wen-Mei Hwu, K. Keutzer, and T. G. Mattson, “The Concurrency Challenge,” *Des. Test Comput. IEEE*, vol. 25, no. 4, pp. 312–320, Aug. 2008.

[227] “Intel® Xeon® Processor E5-2650.” [Online]. Available: http://ark.intel.com/products/64590/Intel-Xeon-Processor-E5-2650-20M-Cache-2_00-GHz-8_00-GTs-Intel-QPI. [Accessed: 29-Jun-2015].

[228] “TOP500 Supercomputer Sites, June 2003.” [Online]. Available: http://www.top500.org/lists/2003/06/. [Accessed: 23-Jan-2013].

[229] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd edition. Morgan Kaufmann Publishers Inc., 2013.

[230] Nvidia, *CUDA Programming Guide 6.0*. Santa Clara, CA: NVIDIA Corporation, 2014.

[231] IEEE Microprocessor Standards Committee, “Draft standard for floating-point arithmetic P754.” Jan-2008.

[232] S. Cook, *CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs*. Elsevier Science, 2012.

[233] A. R. Brodtkorb, T. R. Hagen, and M. L. Saetra, “Graphics processing unit (GPU) programming strategies and trends in GPU computing,” *J Parallel Distrib Comput*, vol. 73, no. 1, pp. 4–13, 2013.

[234] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. Wiley, 2014.

[235] Sang-Won Ha and Tack-Don Han, “A Scalable Work-Efficient and Depth-Optimal Parallel Scan for the GPGPU Environment,” *Parallel Distrib. Syst. IEEE Trans. On*, vol. 24, no. 12, pp. 2324–2333, Dec. 2013.

[236] D. Merrill, “NVIDIA CUB.” [Online]. Available: http://nvlabs.github.io/cub/index.html. [Accessed: 13-Jan-2015].

[237] R. Sedgewick and K. Wayne, *Algorithms*. Pearson Education, 2011.

[238] P. Kipfer and W. Rüdiger, “Improved GPU Sorting,” in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley Professional, 2005, p. 814.

[239] K. E. Batcher, “Sorting networks and their applications,” in *Proceedings of the April 30--May 2, 1968, spring joint computer conference*, Atlantic City, New Jersey, 1968, pp. 307–314.

[240] H. W. Lang and F. H. Flensburg, “Bitonic sort,” *Algorithms*, 18-May-2010. [Online]. Available: http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm. [Accessed: 07-Jul-2015].

[241] “Two-sample t-test - MATLAB ttest2.” [Online]. Available: http://www.mathworks.com/help/stats/ttest2.html. [Accessed: 08-May-2014].

[242] H. Zhang and M. Ishikawa, “The performance verification of an evolutionary canonical particle swarm optimizer,” *18th Int. Conf. Artif. Neural Netw. ICANN 2008*, vol. 23, no. 4, pp. 510–516, May 2010.

[243] J. T. McClave and T. Sincich, *Statistics*. Pearson, 2013.

[244] J. Pereda and J. Dixon, “23-Level Inverter for Electric Vehicles Using a Single Battery Pack and Series Active Filters,” *Veh. Technol. IEEE Trans. On*, vol. 61, no. 3, pp. 1043–1051, Mar. 2012.

[245] S. H. Hosseini, M. Ahmadi, and S. G. Zadeh, “Reducing the output harmonics of cascaded H-bridge multilevel inverter for Electric Vehicle applications,” in *2011 8th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2011, pp. 752–755.

[246] R. A. Klopotek and J. Porter-Sobieraj, “Solving systems of polynomial equations on a GPU,” *Comput. Sci. Inf. Syst. FedCSIS 2012 Fed. Conf. On*, pp. 539–544, Sep. 2012.

[247] B. Pattabiraman, S. Umbreit, Wei-keng Liao, F. Rasio, V. Kalogera, G. Memik, and A. Choudhary, “GPU-accelerated Monte Carlo simulations of dense stellar systems,” *Innov. Parallel Comput. InPar 2012*, pp. 1–10, May 2012.

[248] P. J. Martin, L. F. Ayuso, R. Torres, and A. Gavilanes, “Algorithmic strategies for optimizing the parallel reduction primitive in CUDA,” *High Perform. Comput. Simul. HPCS 2012 Int. Conf. On*, pp. 511–519, Jul. 2012.

[249] K. Y. Lee, Y. M. Park, and J. L. Ortiz, “A United Approach to Optimal Real and Reactive Power Dispatch,” *Power Appar. Syst. IEEE Trans. On*, vol. PAS-104, no. 5, pp. 1147–1153, May 1985.

[250] J. J. Grainger and W. D. Stevenson Jr., *Power System Analysis*. McGraw-Hill, Inc, 1994.

[251] Dan Zou and Yong Dou, “Implementation of parallel sparse Cholesky factorization on GPU,” *Comput. Sci. Netw. Technol. ICCSNT 2012 2nd Int. Conf. On*, pp. 2228–2232, 2012.

[252] A. F. Peixoto de Camargos, V. C. Silva, J.-M. Guichon, and G. Munier, “Efficient Parallel Preconditioned Conjugate Gradient Solver on GPU for FE Modeling of Electromagnetic Fields in Highly Dissipative Media,” *Magn. IEEE Trans. On*, vol. 50, no. 2, pp. 569–572, 2014.

[253] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, “CULA: hybrid GPU accelerated linear algebra routines,” in *SPIE Defense and Security Symposium (DSS)*, 2010.

[254] “cuSPARSE | NVIDIA Developer Zone.” [Online]. Available: https://developer.nvidia.com/cuSPARSE. [Accessed: 14-Apr-2014].

[255] “PARALUTION - The Library for Iterative Sparse Methods on CPU and GPU.” [Online]. Available: http://www.paralution.com/. [Accessed: 14-Apr-2014].

[256] “Eigen.” [Online]. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page. [Accessed: 09-Jul-2015].

[257] M. A. Abido, “Multiobjective Particle Swarm Optimization for Optimal Power Flow Problem,” in *Handbook of Swarm Intelligence*, vol. 8, B. Panigrahi, Y. Shi, and M.-H. Lim, Eds. Springer Berlin Heidelberg, 2011, pp. 241–268.

[258] M. Sailaja Kumari, G. Priyanka, and M. Sydulu, “Comparison of Genetic Algorithms and Particle Swarm Optimization for Optimal Power Flow Including FACTS devices,” in *Power Tech, 2007 IEEE Lausanne*, 2007, pp. 1105–1110.

[259] S. Kahourzade, A. Mahmoudi, and H. Mokhlis, “A comparative study of multi-objective optimal power flow based on particle swarm, evolutionary programming, and genetic algorithm,” *Electr. Eng.*, vol. 97, no. 1, pp. 1–12, Mar. 2015.

[260] C. N. Ravi and C. C. A. Rajan, “A comparative analysis of differential evolution and genetic algorithm for solving optimal power flow,” *Power India Conf. 2012 IEEE Fifth*, pp. 1–6, Dec. 2012.

[261] V. Roberge, M. Tarbouchi, and F. Okou, “Collaborative Parallel Hybrid Metaheuristics on Graphics Processing Unit,” *Int. J. Comput. Intell. Appl.*, vol. 14, no. 01, p. 1550002, Mar. 2015.

[262] U. Leeton, D. Uthitsunthorn, U. Kwannetr, N. Sinsuphun, and T. Kulworawanichpong, “Power loss minimization using optimal power flow based on particle swarm optimization,” *Electr. Eng. Comput. Telecommun. Inf. Technol. ECTI-CON 2010 Int. Conf. On*, pp. 440–444, May 2010.

[263] V. Roberge, “Optimal Power Flow Test Cases.” [Online]. Available: http://roberge.segfaults.net/joomla/index.php/opf. [Accessed: 21-May-2015].

[264] M. A. Abido, “Optimal power flow using particle swarm optimization,” *Int. J. Electr. Power Energy Syst.*, vol. 24, no. 7, pp. 563–571, Oct. 2002.

[265] J. P. Roselyn, D. Devaraj, and S. Dash, “Economic Emission OPF Using Hybrid GA-Particle Swarm Optimization,” in *Swarm, Evolutionary, and Memetic Computing*, vol. 7076, B. Panigrahi, P. Suganthan, S. Das, and S. Satapathy, Eds. Springer Berlin Heidelberg, 2011, pp. 167–175.

[266] S. T. Suganthi and D. Devaraj, “An improved differential evolution based approach for emission constrained optimal power flow,” *Energy Effic. Technol. Sustain. ICEETS 2013 Int. Conf. On*, pp. 1308–1314, Apr. 2013.

[267] M. Rezaei Adaryani and A. Karami, “Artificial bee colony algorithm for solving multi-objective optimal power flow problem,” *Int. J. Electr. Power Energy Syst.*, vol. 53, no. 0, pp. 219–230, Dec. 2013.

[268] H. R. E. H. Bouchekara, “Optimal power flow using black-hole-based optimization approach,” *Appl. Soft Comput.*, vol. 24, no. 0, pp. 879–888, Nov. 2014.

[269] A. Bhattacharya and P. K. Roy, “Solution of multi-objective optimal power flow using gravitational search algorithm,” *Gener. Transm. Distrib. IET*, vol. 6, no. 8, pp. 751–763, Aug. 2012.

[270] V. H. Hinojosa and R. Araya, “Modeling a mixed-integer-binary small-population evolutionary particle swarm algorithm for solving the optimal power flow problem in electric power systems,” *Appl. Soft Comput.*, vol. 13, no. 9, pp. 3839–3852, Sep. 2013.

[271] A. Lashkar Ara, A. Kazemi, S. Gahramani, and M. Behshad, "Optimal reactive power flow using multi-objective mathematical programming," *Sci. Iran.*, vol. 19, no. 6, pp. 1829–1836, Dec. 2012.

[272] Quanyuan Jiang, Guangchao Geng, Chuangxin Guo, and Yijia Cao, "An Efficient Implementation of Automatic Differentiation in Interior Point Optimal Power Flow," *Power Syst. IEEE Trans. On*, vol. 25, no. 1, pp. 147–155, Feb. 2010.

[273] Rung-Fang Chang, Ya-Chin Chang, and Chan-Nan Lu, "Loss Minimization of Distribution Systems with Electric Vehicles by Network Reconfiguration," *Control Eng. Commun. Technol. ICCECT 2012 Int. Conf. On*, pp. 551–555, Dec. 2012.

[274] T. Niknam, A. K. Fard, and A. Seifi, "Distribution feeder reconfiguration considering fuel cell/wind/photovoltaic power plants," *Renew. Energy*, vol. 37, no. 1, pp. 213–225, Jan. 2012.

[275] A. Merlin and H. Back, "Search for a Minimal-Loss Operating Spanning Tree Configuration in an Urban Power Distribution System," presented at the Proc. 5th Power System Computation Conference (PSCC), 1975.

[276] Z. H. Ahmed, "An experimental study of a hybrid genetic algorithm for the maximum traveling salesman problem," *Math. Sci.*, vol. 7, no. 1, pp. 1–7, Feb. 2013.

[277] C. de Andrade, R. Toso, M. Resende, and F. Miyazawa, "Biased Random-Key Genetic Algorithms for the Winner Determination Problem in Combinatorial Auctions," *Evol. Comput.*, vol. 23, no. 2, pp. 279–307, Jun. 2015.

[278] S. Chen, S. Davis, H. Jiang, and A. Novobilski, "CUDA-Based Genetic Algorithm on Traveling Salesman Problem," in *Computer and Information Science 2011*, R. Lee, Ed. Springer Berlin Heidelberg, 2011, pp. 241–252.

[279] J. Li, Q. Sun, M. Zhou, and X. Dai, "A New Multiple Traveling Salesman Problem and Its Genetic Algorithm-Based Solution," in *2013 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2013, pp. 627–632.

[280] A. Swarnkar, N. Gupta, and K. R. Niazi, "Distribution network reconfiguration using population-based AI techniques: A comparative analysis," *Power Energy Soc. Gen. Meet. 2012 IEEE*, pp. 1–6, Jul. 2012.

[281] D. Shirmohammadi, H. W. Hong, A. Semlyen, and G. X. Luo, "A compensation-based power flow method for weakly meshed distribution and transmission networks," *Power Syst. IEEE Trans. On*, vol. 3, no. 2, pp. 753–762, May 1988.

[282] M. F. AlHajri and M. E. El-Hawary, “Exploiting the Radial Distribution Structure in Developing a Fast and Flexible Radial Power Flow for Unbalanced Three-Phase Networks,” *Power Deliv. IEEE Trans. On*, vol. 25, no. 1, pp. 378–389, Jan. 2010.

[283] Wei Wang, Shaozhong Guo, Fan Yang, and Jianxun Chen, “GPU-Based Fast Minimum Spanning Tree Using Data Parallel Primitives,” *Inf. Eng. Comput. Sci. ICIECS 2010 2nd Int. Conf. On*, pp. 1–4, Dec. 2010.

[284] S. Rostrup, S. Srivastava, and K. Singhal, “Fast and memory-efficient minimum spanning tree on the GPU,” *Int J Comput Sci Eng*, vol. 8, no. 1, pp. 21–33, 2013.

[285] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, “Fast minimum spanning tree for large graphs on the GPU,” in *Proceedings of the Conference on High Performance Graphics 2009*, New Orleans, Louisiana, 2009, pp. 167–171.

[286] C. da Silva Sousa, A. Mariano, and A. Proença, “A Generic and Highly Efficient Parallel Variant of Boruvka’s Algorithm,” presented at the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turku, Finland, 2015.

[287] M. Daga, M. Nutter, and M. Meswani, “Efficient breadth-first search on a heterogeneous processor,” *Big Data Big Data 2014 IEEE Int. Conf. On*, pp. 373–382, Oct. 2014.

[288] S. Dashora and N. Khare, “Implementation of graph algorithms over GPU: A comparative analysis,” *Electr. Electron. Comput. Sci. SCEECS 2012 IEEE Stud. Conf. On*, pp. 1–8, Mar. 2012.

[289] D. S. Banerjee, S. Sharma, and K. Kothapalli, “Work efficient parallel algorithms for large graph exploration,” *High Perform. Comput. HiPC 2013 20th Int. Conf. On*, pp. 433–442, Dec. 2013.

[290] D. Merrill, M. Garland, and A. Grimshaw, “Scalable GPU graph traversal,” in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, New Orleans, Louisiana, USA, 2012, pp. 117–128.

[291] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA,” in *Proceedings of the 14th international conference on High performance computing*, Goa, India, 2007, pp. 197–208.

[292] Yangdong Deng, B. D. Wang, and Shuai Mu, “Taming irregular EDA applications on GPUs,” *Comput.-Aided Des. - Dig. Tech. Pap. 2009 ICCAD 2009 IEEEACM Int. Conf. On*, pp. 539–546, Nov. 2009.

[293] V. Roberge, “Distribution Feeder Reconfiguration Test Cases.” [Online]. Available: http://roberge.segfaults.net/joomla/index.php/dfr. [Accessed: 07-May-2015].

[294] "Boost C++ Libraries." [Online]. Available: http://www.boost.org/. [Accessed: 26-Aug-2015].

[295] S. Civanlar, J. J. Grainger, H. Yin, and S. S. H. Lee, "Distribution feeder reconfiguration for loss reduction," *Power Deliv. IEEE Trans. On*, vol. 3, no. 3, pp. 1217–1223, Jul. 1988.

[296] D. Das, "A fuzzy multiobjective approach for network reconfiguration of distribution systems," *Power Deliv. IEEE Trans. On*, vol. 21, no. 1, pp. 202–209, Jan. 2006.

[297] Ching-Tzong Su and Chu-Sheng Lee, "Network reconfiguration of distribution systems using improved mixed-integer hybrid differential evolution," *Power Deliv. IEEE Trans. On*, vol. 18, no. 3, pp. 1022–1027, Jul. 2003.

[298] J. R. Mantovani, F. Casari, and R. A. Romero, "Reconfiguração de sistemas de distribuição radiais utilizando o critério de queda de tensão," *Controle Autom.*, pp. 150–159, 2000.

[299] V. Roberge, M. Tarbouchi, and F. Okou, "gpuMF: A Framework for Parallel Hybrid Metaheuristics on GPU with application to the Minimization of Harmonics in Multilevel Inverters," in *International Conference on Smart Energy Grid Engineering*, Oshawa, Canada, 2014.

[300] V. Roberge, M. Tarbouchi, and F. Okou, "gpuMF: a framework for parallel hybrid metaheuristics on GPU with application to the minimisation of harmonics in multilevel inverters," *Int J Process Syst. Eng.*, vol. 3, no. 1/2/3, pp. 20–41, 2015.

[301] V. Roberge, M. Tarbouchi, and F. Okou, "Distribution System Optimization on Graphics Processing Unit," *IEEE Trans. Smart Grid*, vol. 99, no. 1, pp. 1–1, 2015.

[302] G. Hug-Glanzmann and G. Andersson, "N-1 security in optimal power flow control applied to limited areas," *IET Gener. Transm. Distrib.*, vol. 3, no. 2, pp. 206–215, Feb. 2009.

[303] Y. Xu, Z. Y. Dong, R. Zhang, K. P. Wong, and M. Lai, "Solving Preventive-Corrective SCOPF by a Hybrid Computational Strategy," *Power Syst. IEEE Trans. On*, vol. 29, no. 3, pp. 1345–1355, May 2014.

[304] M. Khanabadi, H. Ghasemi, and M. Doostizadeh, "Optimal Transmission Switching Considering Voltage Security and N-1 Contingency Analysis," *IEEE Trans. Power Syst.*, vol. 28, no. 1, pp. 542–550, Feb. 2013.

[305] S. Chen, W. Hu, and Z. Chen, "Comprehensive Cost Minimization in Distribution Networks Using Segmented-Time Feeder Reconfiguration and Reactive Power Control of Distributed Generators," *IEEE Trans. Power Syst.*, vol. PP, no. 99, pp. 1–11, 2015.

[306] F. Ding and K. A. Loparo, “Feeder Reconfiguration for Unbalanced Distribution Systems With Distributed Generation: A Hierarchical Decentralized Approach,” *IEEE Trans. Power Syst.*, vol. PP, no. 99, pp. 0–10, 2015.

[307] O. Alsac and B. Stott, “Optimal Load Flow with Steady-State Security,” *Power Appar. Syst. IEEE Trans. On*, vol. PAS-93, no. 3, pp. 745–751, May 1974.

Appendice A

Cet appendice donne la description du réseau de transport d'électricité IEEE à 30 bus. Les données sont prises des références [249] et [307]. Le bus 1 est le bus de référence. Les limites inférieures et supérieures des tensions aux bus sont respectivement de 0.95 et 1.1 p.u. pour tous les bus. Les branches 11, 12, 15 et 36 sont des transformateurs en phase dont le rapport de transformation est ajustable de 0.9 à 1.1 par des incréments de 0.0125. Les bus 10, 12, 15, 17, 20, 21, 23, 24 et 29 sont équipés de compensateurs statiques d'énergie réactive réglables de 0 à 5 MVAR par incréments de 0.5 MVAR. La demande de puissance totale pour le réseau est de 283.6 MW et 126.2 MVAR. Les tableaux qui suivent contiennent les données de description des branches, des bus, des générateurs et des variables de contrôles. Les données pour les réseaux IEEE à 118 et 300 bus sont disponibles pour téléchargement en format MATPOWER [136] sur le site internet à la référence [263].

TABLEAU A.1
DONNÉES DE DESCRIPTION DES BRANCHES DU RÉSEAU IEEE À 30 BUS

No de la branche	Bus		R (p.u.)	X (p.u.)	B (p.u.)	Rapport de transf.	Limite (MVA)
	Source	Dest.					
1	1	2	0.0192	0.0575	0.0528	1	130
2	1	3	0.0452	0.1852	0.0408	1	130
3	2	4	0.0570	0.1737	0.0368	1	65
4	3	4	0.0132	0.0379	0.0084	1	130
5	2	5	0.0472	0.1983	0.0418	1	130
6	2	6	0.0581	0.1763	0.0374	1	65
7	4	6	0.0119	0.0414	0.0090	1	90
8	5	7	0.0460	0.1160	0.0204	1	70
9	6	7	0.0267	0.0820	0.0170	1	130
10	6	8	0.0120	0.0420	0.0090	1	32
11	6	9	0.0	0.2080	0.0	0.978	65
12	6	10	0.0	0.5560	0.0	0.969	32
13	9	11	0.0	0.2080	0.0	1	65
14	9	10	0.0	0.1100	0.0	1	65
15	4	12	0.0	0.2560	0.0	0.932	65
16	12	13	0.0	0.1400	0.0	1	65
17	12	14	0.1231	0.2559	0.0	1	32
18	12	15	0.0662	0.1304	0.0	1	32
19	12	16	0.0945	0.1987	0.0	1	32
20	14	15	0.2210	0.1997	0.0	1	16
21	16	17	0.0824	0.1923	0.0	1	16
22	15	18	0.1070	0.2185	0.0	1	16
23	18	19	0.0639	0.1292	0.0	1	16

No de la branche	Bus		R (p.u.)	X (p.u.)	B (p.u.)	Rapport de transf.	Limite (MVA)
	Source	Dest.					
24	19	20	0.0340	0.0680	0.0	1	32
25	10	20	0.0936	0.2090	0.0	1	32
26	10	17	0.0324	0.0845	0.0	1	32
27	10	21	0.0348	0.0749	0.0	1	32
28	10	22	0.0727	0.1499	0.0	1	32
29	21	22	0.0116	0.0236	0.0	1	32
30	15	23	0.1000	0.2020	0.0	1	16
31	22	24	0.1150	0.1790	0.0	1	16
32	23	24	0.1320	0.2700	0.0	1	16
33	24	25	0.1885	0.3292	0.0	1	16
34	25	26	0.2544	0.3800	0.0	1	16
35	25	27	0.1093	0.2087	0.0	1	16
36	28	27	0.0	0.3960	0.0	0.968	65
37	27	29	0.2198	0.4153	0.0	1	16
38	27	30	0.3202	0.6027	0.0	1	16
39	29	30	0.2399	0.4533	0.0	1	16
40	8	28	0.0636	0.2000	0.0428	1	32
41	6	28	0.0169	0.0599	0.0130	1	32

TABLEAU A.2
DONNÉES DE DESCRIPTION DES BUS DU RÉSEAU IEEE À 30 BUS

No du bus	Demande		Tension nominale (kV)	No du bus	Demande		Tension nominale (kV)
	P_D (MW)	Q_D (MVAR)			P_D (MW)	Q_D (MVAR)	
1*	0.0	0.0	135.0	16	3.5	1.8	135.0
2	21.7	12.7	135.0	17	9.0	5.8	135.0
3	2.4	1.2	135.0	18	3.2	0.9	135.0
4	7.6	1.6	135.0	19	9.5	3.4	135.0
5	94.2	19.0	135.0	20	2.2	0.7	135.0
6	0.0	0.0	135.0	21	17.5	11.2	135.0
7	22.8	10.9	135.0	22	0.0	0.0	135.0
8	30.0	30.0	135.0	23	3.2	1.6	135.0
9	0.0	0.0	135.0	24	8.7	6.7	135.0
10	5.8	2.0	135.0	25	0.0	0.0	135.0
11	0.0	0.0	135.0	26	3.5	2.3	135.0
12	11.2	7.5	135.0	27	0.0	0.0	135.0
13	0.0	0.0	135.0	28	0.0	0.0	135.0
14	6.2	1.6	135.0	29	2.4	0.9	135.0
15	8.2	2.5	135.0	30	10.6	1.9	135.0

* Le bus 1 est le bus de référence

TABLEAU A.3
DONNÉES DE DESCRIPTION DES GÉNÉRATEURS DU RÉSEAU IEEE À 30 BUS

No du bus	Production				Coefficient du coût		
	P_G^{min} (MW)	P_G^{max} (MW)	Q_G^{min} (MVAR)	Q_G^{max} (MVAR)	a	b	c
1	50	200	-20	250	0.0	2.00	0.00375
2	20	80	-20	100	0.0	1.75	0.0175
5	15	50	-15	80	0.0	1.00	0.0625
8	10	35	-15	60	0.0	3.25	0.00834
11	10	30	-10	50	0.0	3.00	0.025
13	12	40	-15	60	0.0	3.00	0.025

Les coûts de production ($/h) se calculent comme suit : $F = a + b * P_G + c * {P_G}^2$

TABLEAU A.4
DONNÉES DE DESCRIPTION DES VARIABLES DE CONTRÔLE DU RÉSEAU IEEE À 30 BUS

	Variables de contrôle	Limite inférieure	Limite supérieure	Incrément
Puissance active des générateurs (MW)	P_{G2}	20	80	continue
	P_{G5}	15	50	
	P_{G8}	10	35	
	P_{G11}	10	30	
	P_{G13}	12	40	
Tension des générateurs (p.u.)	V_{G1}	0.95	1.1	continue
	V_{G2}	0.95	1.1	
	V_{G5}	0.95	1.1	
	V_{G8}	0.95	1.1	
	V_{G11}	0.95	1.1	
	V_{G13}	0.95	1.1	
Rapports des transformateurs	T_{Br11}	0.9	1.1	0.0125
	T_{Br12}	0.9	1.1	
	T_{Br15}	0.9	1.1	
	T_{Br36}	0.9	1.1	
Puissance réactive des compensateurs statiques (MVAR)	Q_{Bus10}	0	5	0.5
	Q_{Bus12}	0	5	
	Q_{Bus15}	0	5	
	Q_{Bus17}	0	5	
	Q_{Bus20}	0	5	
	Q_{Bus21}	0	5	
	Q_{Bus23}	0	5	
	Q_{Bus24}	0	5	
	Q_{Bus29}	0	5	

Note: Il y a 24 variables de contrôle pour le réseau IEEE à 30 bus. Il est important de noter que P_{G1} n'est pas une variable d'optimisation, mais une variable dépendante calculée lors de l'analyse d'écoulement de puissance. Toutefois, la valeur calculée pour P_{G1} est considérée lors de l'évaluation de la fonction d'aptitude pour assurer que sa valeur respecte les limites physiques du générateur au bus de référence.

Appendice B

Cet appendice liste les solutions optimales calculées par le PSO parallèle proposé dans cette thèse pour l'optimisation de l'écoulement de puissance pour trois réseaux tests de transport d'électricité.

TABLEAU B.1
VARIABLES DE CONTRÔLE OPTIMALES CALCULÉES PAR LE PSO PARALLÈLE SUR GPU POUR LE SYSTÈME TEST IEEE À 30 BUS

Variable de contrôle	Valeur	Variable de contrôle	Valeur	Variable de contrôle	Valeur	Variable de contrôle	Valeur
P_1, MW	177.046	V_1, p.u.	1.1000	$T_{11\,(6\text{-}9)}$	1.0375	Q_{C15}, Mvar	5.0
P_2, MW	48.695	V_2, p.u.	1.0878	$T_{12\,(6\text{-}10)}$	0.9000	Q_{C17}, Mvar	5.0
P_5, MW	21.304	V_5, p.u.	1.0615	$T_{15\,(4\text{-}12)}$	0.9750	Q_{C20}, Mvar	4.5
P_8, MW	21.081	V_8, p.u.	1.0693	$T_{36\,(28\text{-}27)}$	0.9625	Q_{C21}, Mvar	5.0
P_{11}, MW	11.886	V_{11}, p.u.	1.1000	Q_{C10}, Mvar	5.0	Q_{C23}, Mvar	2.5
P_{13}, MW	12.000	V_{13}, p.u.	1.1000	Q_{C12}, Mvar	5.0	Q_{C24}, Mvar	5.0
						Q_{C29}, Mvar	2.5
TOTAL des coûts de production:		799.03 $/h					
TOTAL des pertes de transport :		8.6128 MW					

Note: La description du réseau est donnée à l'Appendice A.

TABLEAU B.2
VARIABLES DE CONTRÔLE OPTIMALES CALCULÉES PAR LE PSO PARALLÈLE SUR GPU POUR LE SYSTÈME TEST IEEE À 118 BUS

Variable de contrôle	Valeur	Variable de contrôle	Valeur	Variable de contrôle	Valeur	Variable de contrôle	Valeur
P_1, MW	27.006	P_{74}, MW	18.692	V_{27}, p.u.	1.0479	V_{103}, p.u.	1.0517
P_4, MW	0.000	P_{76}, MW	23.886	V_{31}, p.u.	1.0423	V_{104}, p.u.	1.0443
P_6, MW	0.000	P_{77}, MW	0.000	V_{32}, p.u.	1.0471	V_{105}, p.u.	1.0423
P_8, MW	0.000	P_{80}, MW	431.789	V_{34}, p.u.	1.0568	V_{107}, p.u.	1.0363
P_{10}, MW	401.516	P_{85}, MW	0.000	V_{36}, p.u.	1.0548	V_{110}, p.u.	1.0417
P_{12}, MW	85.669	P_{87}, MW	3.648	V_{40}, p.u.	1.0437	V_{111}, p.u.	1.0507
P_{15}, MW	0.000	P_{89}, MW	502.529	V_{42}, p.u.	1.0443	V_{112}, p.u.	1.0366
P_{18}, MW	15.450	P_{90}, MW	0.000	V_{46}, p.u.	1.0456	V_{113}, p.u.	1.0541
P_{19}, MW	24.119	P_{91}, MW	0.000	V_{49}, p.u.	1.0586	V_{116}, p.u.	1.0600
P_{24}, MW	0.000	P_{92}, MW	0.000	V_{54}, p.u.	1.0420	$T_{8\,(8–5)}$	0.9625
P_{25}, MW	194.771	P_{99}, MW	0.000	V_{55}, p.u.	1.0425	$T_{32\,(26–25)}$	1.0625
P_{26}, MW	281.113	P_{100}, MW	232.358	V_{56}, p.u.	1.0418	$T_{36\,(30–17)}$	0.9750
P_{27}, MW	12.258	P_{103}, MW	38.408	V_{59}, p.u.	1.0585	$T_{51\,(38–37)}$	0.9750
P_{31}, MW	7.278	P_{104}, MW	0.000	V_{61}, p.u.	1.0600	$T_{93\,(63–59)}$	0.9875
P_{32}, MW	16.385	P_{105}, MW	6.840	V_{62}, p.u.	1.0556	$T_{95\,(64–61)}$	0.9875
P_{34}, MW	4.987	P_{107}, MW	29.720	V_{65}, p.u.	1.0600	$T_{102\,(65–66)}$	0.9875
P_{36}, MW	10.757	P_{110}, MW	0.000	V_{66}, p.u.	1.0600	$T_{107\,(68–69)}$	0.9500
P_{40}, MW	49.865	P_{111}, MW	35.377	V_{69}, p.u.	1.0600	$T_{127\,(81–80)}$	0.9875
P_{42}, MW	41.289	P_{112}, MW	39.804	V_{70}, p.u.	1.0436	Q_{C34}, Mvar	14.0
P_{46}, MW	19.091	P_{113}, MW	0.004	V_{72}, p.u.	1.0600	Q_{C44}, Mvar	10.0
P_{49}, MW	193.695	P_{116}, MW	0.000	V_{73}, p.u.	1.0457	Q_{C45}, Mvar	10.0
P_{54}, MW	49.440	V_1, p.u.	1.0320	V_{74}, p.u.	1.0249	Q_{C46}, Mvar	10.0
P_{55}, MW	31.025	V_4, p.u.	1.0572	V_{76}, p.u.	1.0136	Q_{C48}, Mvar	15.0
P_{56}, MW	32.916	V_6, p.u.	1.0566	V_{77}, p.u.	1.0475	Q_{C74}, Mvar	12.0
P_{59}, MW	149.955	V_8, p.u.	1.0383	V_{80}, p.u.	1.0600	Q_{C79}, Mvar	20.0
P_{61}, MW	148.641	V_{10}, p.u.	1.0557	V_{85}, p.u.	1.0511	Q_{C82}, Mvar	20.0
P_{62}, MW	0.000	V_{12}, p.u.	1.0466	V_{87}, p.u.	1.0600	Q_{C83}, Mvar	10.0
P_{65}, MW	353.866	V_{15}, p.u.	1.0478	V_{89}, p.u.	1.0600	Q_{C105}, Mvar	20.0
P_{66}, MW	350.033	V_{18}, p.u.	1.0481	V_{90}, p.u.	1.0415	Q_{C107}, Mvar	6.0
P_{69}, MW	454.805	V_{19}, p.u.	1.0471	V_{91}, p.u.	1.0450	Q_{C110}, Mvar	6.0
P_{70}, MW	0.000	V_{24}, p.u.	1.0576	V_{92}, p.u.	1.0503		
P_{72}, MW	0.000	V_{25}, p.u.	1.0600	V_{99}, p.u.	1.0546		
P_{73}, MW	0.000	V_{26}, p.u.	1.0600	V_{100}, p.u.	1.0600		
TOTAL des coûts de production:		129 627.03 $/h					
TOTAL des pertes de transport :		76.9847 MW					

Notes: - La description du réseau provient de MATPOWER [136]

- Les capacités négatives de -40 Mvar et -25 Mvar aux bus 5 et 37 sont gardées constantes

TABLEAU B.3
VARIABLES DE CONTRÔLE OPTIMALES CALCULÉES PAR LE PSO PARALLÈLE SUR GPU POUR LE SYSTÈME TEST IEEE À 300 BUS

Variable de contrôle	Valeur	Variable de contrôle	Valeur	Variable de contrôle	Valeur	Variable de contrôle	Valeur
P_{8}, MW	0.000	P_{7044}, MW	39.710	V_{236}, p.u.	1.0600	$Tap_{(87-94)}$	0.9750
P_{10}, MW	0.000	P_{7049}, MW	70.071	V_{238}, p.u.	1.0552	$Tap_{(114-207)}$	1.0250
P_{20}, MW	9.831	P_{7055}, MW	47.715	V_{239}, p.u.	1.0584	$Tap_{(116-124)}$	0.9750
P_{63}, MW	55.457	P_{7057}, MW	158.264	V_{241}, p.u.	1.0311	$Tap_{(121-115)}$	0.9000
P_{76}, MW	99.996	P_{7061}, MW	373.868	V_{242}, p.u.	1.0465	$Tap_{(130-131)}$	1.0875
P_{84}, MW	402.367	P_{7062}, MW	380.634	V_{243}, p.u.	1.0487	$Tap_{(130-150)}$	1.0625
P_{91}, MW	156.840	P_{7071}, MW	137.193	V_{7001}, p.u.	1.0600	$Tap_{(132-170)}$	1.0250
P_{92}, MW	284.363	P_{7130}, MW	1226.605	V_{7002}, p.u.	1.0586	$Tap_{(141-174)}$	1.0000
P_{98}, MW	75.670	P_{7139}, MW	716.654	V_{7003}, p.u.	1.0532	$Tap_{(143-144)}$	0.9375
P_{108}, MW	140.345	P_{7166}, MW	492.571	V_{7011}, p.u.	1.0538	$Tap_{(143-148)}$	0.9875
P_{119}, MW	1990.699	P_{9002}, MW	27.405	V_{7012}, p.u.	1.0600	$Tap_{(151-170)}$	1.0125
P_{124}, MW	249.151	P_{9051}, MW	49.572	V_{7017}, p.u.	1.0278	$Tap_{(153-183)}$	0.9750
P_{125}, MW	90.714	P_{9053}, MW	47.278	V_{7023}, p.u.	1.0600	$Tap_{(155-156)}$	0.9875
P_{138}, MW	0.000	P_{9054}, MW	53.279	V_{7024}, p.u.	1.0582	$Tap_{(159-117)}$	1.0750
P_{141}, MW	277.753	P_{9055}, MW	8.579	V_{7039}, p.u.	0.9839	$Tap_{(160-124)}$	1.0125
P_{143}, MW	661.005	V_{8}, p.u.	1.0311	V_{7044}, p.u.	0.9789	$Tap_{(163-137)}$	0.9250
P_{146}, MW	80.031	V_{10}, p.u.	1.0014	V_{7049}, p.u.	1.0290	$Tap_{(164-155)}$	1.0125
P_{147}, MW	198.259	V_{20}, p.u.	1.0266	V_{7055}, p.u.	1.0439	$Tap_{(182-139)}$	1.0250
P_{149}, MW	96.604	V_{63}, p.u.	1.0559	V_{7057}, p.u.	1.0600	$Tap_{(189-210)}$	1.0375
P_{152}, MW	339.324	V_{76}, p.u.	1.0175	V_{7061}, p.u.	1.0470	$Tap_{(193-196)}$	1.0250
P_{153}, MW	199.987	V_{84}, p.u.	1.0600	V_{7062}, p.u.	1.0245	$Tap_{(195-212)}$	0.9875
P_{156}, MW	0.000	V_{91}, p.u.	1.0523	V_{7071}, p.u.	0.9847	$Tap_{(201-69)}$	1.0500
P_{170}, MW	199.897	V_{92}, p.u.	1.0600	V_{7130}, p.u.	1.0600	$Tap_{(202-211)}$	1.0250
P_{171}, MW	40.800	V_{98}, p.u.	1.0382	V_{7139}, p.u.	1.0600	$Tap_{(204-2040)}$	1.0500
P_{176}, MW	195.150	V_{108}, p.u.	1.0221	V_{7166}, p.u.	1.0381	$Tap_{(209-198)}$	1.0375
P_{177}, MW	79.660	V_{119}, p.u.	1.0600	V_{9002}, p.u.	1.0330	$Tap_{(218-219)}$	1.0375
P_{185}, MW	203.406	V_{124}, p.u.	1.0476	V_{9051}, p.u.	0.9410	$Tap_{(229-230)}$	0.9750
P_{186}, MW	1176.741	V_{125}, p.u.	1.0468	V_{9053}, p.u.	0.9439	$Tap_{(234-236)}$	0.9750
P_{187}, MW	1177.695	V_{138}, p.u.	1.0600	V_{9054}, p.u.	0.9858	$Tap_{(238-239)}$	1.0000
P_{190}, MW	507.353	V_{141}, p.u.	1.0600	V_{9055}, p.u.	1.0010	$Tap_{(119-1190)}$	1.1000
P_{191}, MW	1867.749	V_{143}, p.u.	1.0600	$Tap_{(37-9001)}$	1.0250	$Tap_{(120-1200)}$	1.0125
P_{198}, MW	441.558	V_{146}, p.u.	1.0600	$Tap_{(9001-9006)}$	0.9750	$Tap_{(7062-62)}$	0.9625
P_{213}, MW	274.886	V_{147}, p.u.	1.0595	$Tap_{(9001-9012)}$	0.9625	$Tap_{(7017-17)}$	0.9375
P_{220}, MW	108.116	V_{149}, p.u.	1.0600	$Tap_{(9005-9051)}$	1.0875	$Tap_{(7039-39)}$	0.9250
P_{221}, MW	487.210	V_{152}, p.u.	1.0600	$Tap_{(9005-9052)}$	0.9000	$Tap_{(7057-57)}$	1.0250
P_{222}, MW	269.215	V_{153}, p.u.	1.0329	$Tap_{(9005-9053)}$	1.1000	$Tap_{(7044-44)}$	0.9125
P_{227}, MW	328.235	V_{156}, p.u.	1.0300	$Tap_{(9005-9054)}$	1.0500	$Tap_{(7055-55)}$	0.9750

Variable de contrôle	Valeur	Variable de contrôle	Valeur	Variable de contrôle	Valeur	Variable de contrôle	Valeur
P_{230}, MW	361.286	V_{170}, p.u.	0.9913	$Tap_{(9005-9055)}$	1.0500	$Tap_{(7071-71)}$	0.9250
P_{233}, MW	326.399	V_{171}, p.u.	1.0038	$Tap_{(9053-9533)}$	1.0000	Q_{C117}, Mvar	325
P_{236}, MW	634.584	V_{176}, p.u.	1.0600	$Tap_{(3-1)}$	1.0500	Q_{C120}, Mvar	100
P_{238}, MW	265.525	V_{177}, p.u.	1.0187	$Tap_{(3-2)}$	0.9625	Q_{C154}, Mvar	7
P_{239}, MW	577.340	V_{185}, p.u.	1.0507	$Tap_{(3-4)}$	0.9625	Q_{C164}, Mvar	-54
P_{241}, MW	613.307	V_{186}, p.u.	1.0600	$Tap_{(7-5)}$	0.9500	Q_{C166}, Mvar	-26
P_{242}, MW	175.927	V_{187}, p.u.	1.0587	$Tap_{(7-6)}$	0.9750	Q_{C173}, Mvar	70
P_{243}, MW	86.013	V_{190}, p.u.	1.0540	$Tap_{(10-11)}$	0.9625	Q_{C179}, Mvar	76
P_{7001}, MW	436.485	V_{191}, p.u.	1.0600	$Tap_{(12-10)}$	1.0000	Q_{C190}, Mvar	-150
P_{7002}, MW	560.790	V_{198}, p.u.	1.0572	$Tap_{(15-17)}$	0.9500	Q_{C231}, Mvar	-300
P_{7003}, MW	1152.129	V_{213}, p.u.	1.0600	$Tap_{(16-15)}$	0.9750	Q_{C238}, Mvar	100
P_{7011}, MW	226.078	V_{220}, p.u.	1.0502	$Tap_{(21-20)}$	0.9500	Q_{C240}, Mvar	-140
P_{7012}, MW	356.542	V_{221}, p.u.	1.0463	$Tap_{(24-23)}$	1.0125	Q_{C248}, Mvar	36
P_{7017}, MW	386.544	V_{222}, p.u.	1.0218	$Tap_{(36-35)}$	0.9500	Q_{C9003}, Mvar	7
P_{7023}, MW	166.865	V_{227}, p.u.	1.0115	$Tap_{(45-44)}$	0.9875	Q_{C9034}, Mvar	1
P_{7024}, MW	386.643	V_{230}, p.u.	1.0600	$Tap_{(62-61)}$	0.9625		
P_{7039}, MW	516.508	V_{233}, p.u.	1.0274	$Tap_{(63-64)}$	1.0500		
TOTAL des coûts de production:		719 442.74 $/h					
TOTAL des pertes de transport :		298.5800 MW					

Note: La description du réseau provient de MATPOWER [136]

Appendice C

Cet appendice donne la description du réseau de distribution à 16 bus dont les données proviennent de la référence [295]. Les bus 1, 2 et 3 sont les bus d'alimentation. Les autres sont connectées à des charges. Les branches 13, 14 et 15 sont des interrupteurs d'interconnexion normalement ouverts. La puissance nominale du réseau est de 100 MW. Les descriptions pour les réseaux de distribution à 33, 70, 83, 136, 415, 880, 1760 et 4400 bus disponibles pour téléchargement sur le site internet à la référence [293].

TABLEAU C.1
DONNÉES DE DESCRIPTION DU RÉSEAU DE DISTRIBUTION À 16 BUS

No de la branche	Bus		R	X	Demande (au bus dest.)	
	Source	Dest.	(p.u.)	(p.u.)	P_D (MW)	Q_D (MVAR)
0	1	4	0.075	0.1	2.0	1.6
1	4	5	0.08	0.11	3.0	0.4
2	4	6	0.09	0.18	2.0	-0.4
3	6	7	0.04	0.04	1.5	1.2
4	2	8	0.11	0.11	4.0	2.7
5	8	9	0.08	0.11	5.0	1.8
6	8	10	0.11	0.11	1.0	0.9
7	9	11	0.11	0.11	0.6	-0.5
8	9	12	0.08	0.11	4.5	-1.7
9	3	13	0.11	0.11	1.0	0.9
10	13	14	0.09	0.12	1.0	-1.1
11	13	15	0.08	0.11	1.0	0.9
12	15	16	0.04	0.04	2.1	-0.8
13	5	11	0.04	0.04	-	-
14	10	14	0.04	0.04	-	-
15	7	16	0.12	0.12	-	-

Appendice D

Cet appendice liste les solutions optimales calculées par le GA parallèle proposé dans cette thèse pour la reconfiguration optimale des réseaux de distribution.

TABLEAU D.1
CONFIGURATIONS OPTIMALES CALCULÉES PAR LE GA PARALLÈLE SUR GPU POUR DIFFÉRENTS RÉSEAUX DE DISTRIBUTION

Réseaux	Index des branches ouvertes
16 bus	6, 7, 15
33 bus	6, 8, 13, 31, 36
70 bus	12, 29, 44, 50, 65, 69, 74, 75, 76, 77, 78
83 bus	6, 12, 33, 38, 41, 54, 61, 71, 82, 85, 88, 89, 91
136 bus	6, 34, 50, 89, 95, 105, 117, 125, 134, 136, 137, 140, 141, 143, 144, 145, 146, 147, 149, 150, 154
415 bus	6, 12, 33, 38, 41, 54, 61, 71, 82, 89, 95, 116, 121, 124, 137, 144, 154, 165, 172, 178, 199, 204, 207, 220, 227, 237, 248, 255, 261, 282, 287, 290, 303, 310, 320, 331, 338, 344, 365, 370, 373, 386, 393, 403, 414, 417, 420, 421, 423, 430, 433, 434, 436, 443, 446, 447, 449, 456, 459, 460, 462, 469, 472, 473, 475
880 bus	83, 129, 140, 158, 189, 281, 287, 305, 311, 408, 410, 451, 493, 595, 615, 629, 630, 636, 697, 814, 843, 884, 887, 888, 889, 895, 899
1760 bus	83, 118, 128, 133, 166, 191, 243, 281, 288, 305, 311, 323, 406, 414, 451, 458, 493, 585, 613, 627, 629, 633, 638, 647, 680, 697, 793, 814, 839, 843, 956, 1003, 1012, 1028, 1040, 1062, 1096, 1110, 1114, 1138, 1155, 1160, 1185, 1188, 1281, 1284, 1306, 1323, 1365, 1450, 1468, 1488, 1501, 1505, 1510, 1553, 1570, 1687, 1716, 1718, 1754, 1757, 1761, 1763, 1765, 1772, 1778, 1784, 1787, 1789, 1799, 1806, 1807, 1811
4400 bus	51, 79, 115, 133, 167, 189, 242, 281, 283, 310, 315, 329, 330, 400, 405, 417, 423, 446, 450, 493, 558, 612, 615, 626, 633, 639, 644, 645, 697, 729, 793, 842, 870, 915, 952, 961, 1001, 1014, 1040, 1061, 1099, 1116, 1153, 1157, 1161, 1178, 1197, 1280, 1297, 1356, 1426, 1466, 1484, 1499, 1503, 1508, 1512, 1517, 1570, 1666, 1687, 1714, 1729, 1736, 1743, 1792, 1829, 1864, 1878, 1884, 1960, 1989, 2027, 2031, 2051, 2070, 2075, 2076, 2088, 2129, 2151, 2161, 2179, 2196, 2203, 2236, 2241, 2304, 2319, 2340, 2359, 2361, 2374, 2377, 2381, 2383, 2393, 2443, 2475, 2563, 2588, 2608, 2692, 2741, 2751, 2760, 2785, 2810, 2858, 2896, 2906, 2924, 2930, 2933, 2998, 3011, 3027, 3029, 3066, 3068, 3075, 3110, 3114, 3197, 3214, 3231, 3239, 3248, 3261, 3264, 3267, 3308, 3430, 3462, 3575, 3621, 3633, 3650, 3681, 3773, 3779, 3797, 3803, 3899, 3902, 3942, 3985, 4065, 4087, 4107, 4121, 4122, 4128, 4189, 4306, 4334, 4367, 4376, 4381, 4382, 4391, 4402, 4407, 4409, 4418, 4429, 4445, 4449, 4462, 4472, 4484, 4487, 4488, 4495, 4499, 4512, 4519, 4522, 4523, 4527, 4529, 4536, 4537, 4538, 4549

FIN