Thèse présentée pour obtenir le grade de docteur

Université Lille 1 - Sciences et Technologies

École Doctorale Sciences Pour l'Ingénieur Université Lille Nord-de-France

Laboratoire d'Informatique Fondamentale de Lille (UMR CNRS 8022)

Centre de Recherche INRIA Lille Nord Europe

Numéro d'ordre: 40705

Discipline : Informatique

Métaheuristiques parallèles sur GPU

Par : **Thé Van Luong**

Membres du jury:

Rapporteur : Pascal Bouvry, Professeur des Universités, Luxembourg

Rapporteur : Pierre Manneback, Professeur des Universités, Mons (Belgique)

Examinateur : Didier El Baz, Chargé de Recherche HDR, LAAS CNRS Toulouse

Président : Pierre Boulet, Professeur des Universités, Lille 1

Directeur : Nouredine Melab, Professeur des Universités, Lille 1

Co-directeur : El-Ghazali Talbi, Professeur des Universités, Lille 1

Date de soutenance : 1/12/2011

Remerciements

Je souhaite tout d'abord remercier mes directeurs de thèse, Nouredine Melab et El-Ghazali Talbi. Je tiens à leur exprimer toute ma profonde reconnaissance pour leur soutien, leurs encouragements et leurs précieux conseils, tout au long de ce doctorat.

Mes vifs remerciements vont également à tous les membres du jury : Pierre Boulet, président, Pierre Manneback et Pascal Bouvry, rapporteurs, ainsi que Didier El Baz, examinateur de cette thèse.

Je remercie tous les membres de l'équipe DOLPHIN avec qui j'ai pris énormément de plaisirs à travailler durant ces trois années de thèse. Un merci en particulier aux doctorants, ingénieurs et stagiaires de l'équipe : Marie-Éléonore (alias mon petit boudin), Yacine (alias Yaya), Mathieu (alias petit Mathieu), Nadia, Karima, Mostepha, Ahcène, Tuan, Moustapha, Martin, les deux Julie, Imen, Sezin, Ines, Clive et François Legillon. Un remerciement aux permanents de l'équipe : Arnaud, Sébastien, Bilel, Clarisse, Laetitia, François, Luce et Dimo. Mais également aux anciens membres de l'équipe : Mahmoud, Waldo, Mathieu (alias le Breton), Mohand, Jean-Charles Boisson (comme la boisson), Rémy Chevrier, Feryal, Salma, Emilia, Alexandru, Ali, Jérôme, Gaël, Antonio, Jérémie (merci pour SC2), Loukil et Malika Mehdi. Sans oublier les différentes assistantes : Malika et Julie Jonas.

Par ailleurs, je tiens également à exprimer ma gratitude auprès d'Éric Taillard pour m'avoir chaleureusement accueilli en Suisse dans le cadre d'un post-doctorat.

De plus, je voudrais adresser mes profonds remerciements à tous mes amis de Nice pour le témoignage de leur amitié : David (alias Dav la frite), Cédric (alias Debeaume), Sarco (ou Elium le tombeur d'Idra), Aurélie, Rivière, Stéphane, Christophe, Christopher, Oliv, Guitou, Savino et Julien. Je tiens également à rendre hommage aux différentes femmes qui ont partagé une partie de ma vie, et qui m'ont apporté un certain équilibre tout au long de cette thèse : Jo Ferrari, Fleur, Eva, Caroline Gastaldi, Héléna et Isabelle.

Et enfin, un grand merci à ma mère, mon père et mes deux frères Vinh et Phuc, pour leur affection et leur soutien tout au long de mon doctorat, mais aussi lors de ces vingt-huit dernières années.

Contents

Introduction 1

1 GPU Computing for Parallel Metaheuristics 5

1.1 Parallel Metaheuristics 6

1.1.1 Optimization Context 6

1.1.2 Principles of Metaheuristics 7

1.1.2.1 Solution Representation 7

1.1.2.2 Evaluation Function 8

1.1.2.3 Principles of S-metaheuristics 8

1.1.2.4 Principles of P-metaheuristics 10

1.1.3 Parallel Models of Metaheuristics 11

1.2 Metaheuristics and GPU Computing 13

1.2.1 GPU Architecture 13

1.2.2 GPU Challenges for Metaheuristics 15

1.2.3 General GPU Model: CPU-GPU Cooperation 16

1.2.4 GPU Threads Model: Parallelism Control 16

1.2.5 Kernel Management: Memory Management 17

1.3 Related Works on Parallel Metaheuristics 20

1.3.1 Metaheuristics on Parallel and Distributed Architectures 20

1.3.2 Research Works on GPU-based Metaheuristics 21

1.4 Experimental Protocol 26

1.4.1 Optimization Problems 26

1.4.1.1 Permuted Perceptron Problem 27

1.4.1.2 The Quadratic Assignment Problem 27

1.4.1.3 The Weierstrass Continuous Function 27

1.4.1.4 The Traveling Salesman Problem 28

1.4.1.5 The Golomb Rulers 28

1.4.1.6 Problem Characteristics 29

1.4.2 Machines Configuration 30

1.4.3 Metric and Statistical Tests 31

2 Efficient CPU-GPU Cooperation **35**
2.1 Task Repartition for Metaheuristics on GPU 37
2.1.1 Model of Parallel Evaluation of Solutions 37
2.1.2 Parallelization Scheme on GPU 37
2.2 Data Transfer Optimization 38
2.2.1 Generation of the Neighborhood in S-metaheuristics 39
2.2.2 The Proposed GPU-based Algorithm 40
2.2.3 Additional Data Transfer Optimization 42
2.3 Performance Evaluation 43
2.3.1 Analysis of the Data Transfers from CPU to GPU 43
2.3.2 Additional Data Transfer Optimization 49
2.4 Comparison with Other Parallel and Distributed Architectures 50
2.4.1 Parallelization Scheme on Parallel and Distributed Architectures . . 52
2.4.2 Configurations 52
2.4.3 Cluster of Workstations 54
2.4.4 Workstations in a Grid Organization 55

3 Efficient Parallelism Control **59**
3.1 Thread Control for Metaheuristics on GPU 61
3.1.1 Execution Parameters at Runtime 61
3.1.2 Thread Control Heuristic 62
3.2 Efficient Mapping of Neighborhood Structures on GPU 64
3.2.1 Binary Encoding 64
3.2.2 Discrete Vector Representation 65
3.2.3 Vector of Real Values 65
3.2.4 Permutation Representation 66
3.2.4.1 2-exchange Neighborhood 66
3.2.4.2 3-exchange Neighborhood 68
3.2.4.3 Mapping Tables for General Neighborhoods 69
3.3 First Improvement S-metaheuristics on GPU 69
3.4 Performance Evaluation 71
3.4.1 Thread Control for Preventing Crashes 71
3.4.1.1 Application to the Traveling Salesman Problem 71
3.4.1.2 Thread Control Applied to the Traveling Salesman Problem 73
3.4.2 Thread Control for Further Optimization 74
3.4.3 Performance of User-defined Mappings 74
3.4.4 First Improvement S-metaheuristics on GPU 77
3.5 Large Neighborhoods for Improving Solutions Quality 80

3.5.1 Application to the Permuted Perceptron Problem 81
3.5.1.1 Neighborhood based on a 1-Hamming Distance 81
3.5.1.2 Neighborhood based on a 2-Hamming Distance 82
3.5.1.3 Neighborhood based on a 3-Hamming Distance 82
3.5.1.4 Performance Analysis 83

4 Efficient Memory Management 87
4.1 Common Concepts of Memory Management 89
4.1.1 Memory Coalescing Issues 89
4.1.2 Coalescing Transformation 90
4.1.3 Texture Memory 91
4.1.4 Memory Management 92
4.2 Memory Management in Cooperative Algorithms 94
4.2.1 Parallel and Cooperative Model 94
4.2.2 Parallelization Strategies for Cooperative Algorithms 96
4.2.2.1 Parallel Evaluation of Populations on GPU 96
4.2.2.2 Full Distribution of Cooperative Algorithms on GPU . . . 98
4.2.2.3 Full Distribution Using Shared Memory 100
4.2.3 Issues Related to the Fully Distributed Schemes 101
4.3 Performance Evaluation 106
4.3.1 Coalescing accesses to global memory 106
4.3.2 Memory Associations of Optimization Problems 107
4.4 Performance of Cooperative Algorithms 108
4.4.1 Configuration 109
4.4.2 Measures in Terms of Efficiency 109
4.4.3 Measures in Terms of Effectiveness 113

5 Extension of ParadisEO for GPU-based Metaheuristics 115
5.1 The ParadisEO Framework 117
5.1.1 Motivations and Goals 117
5.1.2 Presentation of the Framework 117
5.2 GPU-enabled ParadisEO 118
5.2.1 Architecture of ParadisEO-GPU 119
5.2.2 ParadisEO-GPU Components 120
5.2.3 A Case Study: Parallel Evaluation of a Neighborhood 122
5.2.4 Automatic Construction of the Mapping Function 124
5.3 Performance Evaluation 126
5.3.1 Experimentation with ParadisEO-GPU 126

5.3.1.1 Application to the Permuted Perceptron Problem 126
5.3.1.2 Application to the Quadratic Assignment Problem 129

Conclusion and Future Works **133**

Appendix **137**
.1 Mapping Proofs 137
.1.1 Two-to-one Index Transformation 137
.1.2 One-to-two Index Transformation 138
.1.3 One-to-three Index Transformation 139
.1.4 Three-to-one index transformation 141
.2 Statistical Tests 143

Bibliography **158**

International Publications **160**

Introduction

In the optimization field, both academic and industrial problems are often complex and NP-hard. In practice, their modeling is continuously evolving in terms of constraints and objectives. Thereby, a large number of real-life optimization problems in science, engineering, economics, and business are complex and difficult to solve. Their resolution cannot be performed in an exact manner within a reasonable amount of time, and their resource requirements are ever increasing. To deal with such an issue, the design of resolution methods must be based on the joint use of advanced approaches from combinatorial optimization, large-scale parallelism and engineering methods.

In the last decades, metaheuristics are approximate algorithms that have been successfully applied to solve optimization problems. Indeed, this class of methods allows to produce near-optimal solutions in a reasonable time. Metaheuristics may solve instances of problems that are believed to be hard in general, by exploring the usually large solution search space of these instances. These algorithms achieve this by reducing the effective size of the search space and by exploring that space efficiently. However, although metaheuristics allow to reduce the temporal complexity of problems resolution, they remain unsatisfactory to tackle large problems. Experiments using large problems are often stopped without any convergence being reached. Thereby, in designing metaheuristics, there is often a trade-off to be found between the size of the problem instance and the computational complexity to explore it. As a result, only the use of parallelism allows to design new methods to tackle large problems.

Over the last decades, parallel computing has been revealed as an unavoidable way to deal with large problem instances of difficult optimization problems. The design and implementation of parallel metaheuristics are strongly influenced by the computing platform. Many contributions have been proposed for the design and implementation of parallel metaheuristics using massively parallel processors [CSK93], networks or cluster of workstations [CTG95, BSB$^+$01], and shared memory machines [JRG09, Bev02]. The proposed approaches are based on three parallel models: parallel evaluation of a single solution, parallel evaluation of solutions and parallel (cooperative or independent) execution of several metaheuristics. These parallel approaches have been later revisited for large-scale computational grids [TMT07]. Indeed, grid computing is an impressively powerful way to solve challenging instances in combinatorial optimization. However, computational grids providing a huge amount of resources are not easily available and accessible for any user.

Recently, graphics processing units (GPU) have emerged as a new popular support for massively parallel computing [RRS+08, OML+08]. Such resources supply a great computing power, are energy-efficient, and unlike grids, they are highly available everywhere: laptops, desktops, clusters, etc. During many years, the use of GPU computing was dedicated to graphics and video applications. Its utilization has recently been extended to other application domains [CBM+08, GLGN+08] (e.g. scientific computing) thanks to the publication of the CUDA (Compute Unified Device Architecture) development toolkit that allows GPU programming in C-like language [NBGS08]. In some areas such as numerical computing [TSP+08], we are now witnessing the proliferation of software libraries such as CUBLAS for GPU. However, in other areas such as combinatorial optimization, in particular metaheuristics, the utilization of GPU does not grow at the same pace. With the arrival of open standard programming languages on GPU and the arrival of future compilers for these languages, like other application areas, combinatorial optimization on GPU will generate a growing interest.

Indeed, GPU computing has emerged in the recent years as an important challenge for the parallel computing research area. This new emerging technology is believed to be extremely useful to speed up many complex algorithms. One of the major issues for metaheuristics is to rethink existing parallel models and programming paradigms to allow their deployment on GPU accelerators. In other words, the challenge is to revisit the parallel models and paradigms to efficiently take into account the characteristics of GPUs. However, the exploitation of parallel models is not trivial, and many issues related to the GPU memory hierarchical management of this architecture have to be considered. Generally speaking, the major issues we have to deal with are: the distribution of data processing between CPU and GPU, the thread synchronization, the optimization of data transfer between the different memories, the memory capacity constraints, etc.

The contribution of this thesis is to deal with such issues for the redesign of parallel models of metaheuristics to allow solving of large scale optimization problems on GPU architectures. Our objective is to rethink the existing parallel models and to enable their deployment on GPUs. In this purpose, we propose in this document a new generic guideline for building efficient parallel metaheuristics on GPU. Our challenge is to come out with the GPU-based design of the whole hierarchy of parallel models. Different contributions and salient issues in this document are dealt with: (1) an effective cooperation between the CPU and the GPU, which requires to optimize the data transfers between the CPU and the GPU; (2) an efficient parallelism control to associate working units with threads, and to meet the memory constraints; (3) efficient mappings of data structures of the different models on the hierarchy of memories (with different access latencies) provided by the GPU; (4) software framework-based implementations to make the GPU as transparent as

possible for the user.
Very efficient approaches are proposed for CPU-GPU data transfer optimization, thread control, mapping of solutions to GPU threads or memory management. These approaches have been exhaustively experimented using five optimization problems and four GPU configurations. Compared to a CPU-based execution, accelerations up to $\times 80$ are reported for large combinatorial problems and up to $\times 2000$ for a continuous problem. The different works related to this thesis have been accepted in a dozen of publications, including the IEEE Transactions on Computers journal.

Document Organization

Chapter 1

The first chapter describes general concepts for parallel metaheuristics and GPU computing. In this purpose, we will first introduce parallel models of metaheuristics. Thereafter, we will present the general GPU architecture, and highlight the different challenges that appear when dealing with metaheuristics. The rest of the chapter is dedicated to all the definitions and protocols required to the general comprehension of the document.

Chapter 2

The next chapter contributes to the efficient cooperation between the CPU and the GPU. Thereby, we will highlight how the optimization of data transfers between the two components has a crucial impact on the performance of metaheuristics on GPU. Furthermore, extensive experiments demonstrate the strong potential of GPU-based metaheuristics compared to cluster or grid-based parallel architectures.

Chapter 3

In the third chapter, the focus is on the efficient control of parallelism when designing metaheuristics on GPU. On the one hand, such a step allows to establish a clear association of the elements to be processed according to the spatial thread organization of GPUs. On the other hand, controlling the generation of threads will introduce some robustness in GPU applications. Therefore, it may prevent applications from crashing, and it may lead to some performance improvements.

Chapter 4

In the fourth chapter, we will describe different memory associations of optimization structures to deal with different GPU-based metaheuristics. As an illustration, the scope of

this chapter is to redefine parallel and cooperative algorithms, in which this memory management is prominent. We will investigate on how the redesign of a same algorithm using different memory associations has an impact on the global performance.

Chapter 5

The final chapter introduces an extension of the ParadisEO framework for the transparent deployment of metaheuristics on GPU. In this purpose, conceptual aspects are exposed to allow the user to program GPU-based metaheuristics, while minimizing his or her involvement in its management.

Chapter 1

GPU Computing for Parallel Metaheuristics

This first chapter presents all the background and prerequisites necessary to the general comprehension of the global document.

First, we will describe the principles of metaheuristics within the optimization context. An overview is made on the parallel models of metaheuristics to accelerate the search process. Thereafter, GPU computing is introduced in the context of metaheuristics. In this purpose, we will present the different advantages and the challenges associated with this emergent technology. Furthermore, a review of different works of the literature for metaheuristics on parallel and GPU architectures will be made. Finally, we will emphasize on the experimental protocol used for the experiments performed in this manuscript.

Contents

1.1 Parallel Metaheuristics 6

1.1.1 Optimization Context 6

1.1.2 Principles of Metaheuristics 7

1.1.3 Parallel Models of Metaheuristics 11

1.2 Metaheuristics and GPU Computing 13

1.2.1 GPU Architecture 13

1.2.2 GPU Challenges for Metaheuristics 15

1.2.3 General GPU Model: CPU-GPU Cooperation 16

1.2.4 GPU Threads Model: Parallelism Control 16

1.2.5 Kernel Management: Memory Management 17

1.3 Related Works on Parallel Metaheuristics 20

1.3.1 Metaheuristics on Parallel and Distributed Architectures 20

1.3.2 Research Works on GPU-based Metaheuristics 21

1.4 Experimental Protocol 26

1.4.1 Optimization Problems 26

1.4.2 Machines Configuration 30

1.4.3 Metric and Statistical Tests 31

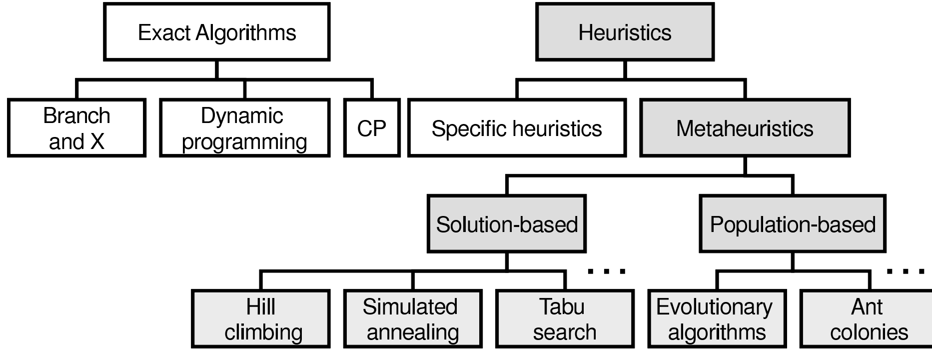


Figure 1.1: Taxonomy of resolution methods for optimization problems.

1.1 Parallel Metaheuristics

1.1.1 Optimization Context

An optimization problem can be formulated as an optimization (minimization or maximization) of a cost function (monoobjective optimization) or a vector of cost functions (multiobjective optimization). This (these) function(s) is (are) called *objective function(s)*. In this document, minimization problems are considered, without loss of generality. An optimization problem (OP) can be formulated as follows:

$$(OP) = \begin{cases} Min \quad F(x) = (f_1(x), f_2(x), \ldots, f_n(x)) \\ s.t. \quad x \in S \end{cases}$$

where n is the number of objectives, $x = (x_1, \ldots, x_k)$ is the vector representing the decision variables, and S represents the set of feasible solutions associated with equality and inequality constraints and explicit bounds. $F(x) = (f_1(x), f_2(x), \ldots, f_n(x))$ is the vector of objectives to be optimized. For $n = 1$ (respectively $n \geq 2$), monoobjective (respectively multiobjective) optimization is considered.

Thereby, the resolution of a monoobjective optimization problem consists in finding the feasible solution that minimizes the objective function. In the multiobjective context, the problem resolution aims at finding a set of Pareto optimal solutions, which is called the *Pareto front*.

Following the complexity of the problem, two main families of resolution methods can be used: *exact methods* and *heuristics*. Figure 1.1 illustrates the different methods of resolution. Exact methods (e.g. branch-and-x, dynamic programming or constraints programming) allow to find optimal solutions and guarantee their optimality. However, they become impractical for large problems.

Conversely, heuristics produce high-quality solutions in a reasonable time practical use on

large-size problem instances. Heuristics can be *specific* i.e. designed to solve a particular problem and/or instance. They can also be generic and applicable to different problem types. In this case, they are called *metaheuristics*. These latter are based on the iterative improvement of either a single solution (e.g. hill climbing, simulated annealing or tabu search) or a population of solutions (e.g. evolutionary algorithms or ant colonies) of a given optimization problem. In this document, the focus will be exclusively on metaheuristics.

1.1.2 Principles of Metaheuristics

Unlike exact methods, metaheuristics allow to tackle large-size problem instances by delivering satisfactory solutions in a reasonable time. There is no guarantee to find global optimal solutions or even bounded solutions. Metaheuristics have received more and more popularity in the past 20 years. Their use in many applications shows their efficiency and effectiveness to solve large and complex problems. Metaheuristics fall into two categories: *single-solution based metaheuristics* (S-metaheuristics) and *population-based metaheuristics* (P-metaheuristics).

S-metaheuristics manipulate and transform a single solution during the search, while in P-metaheuristics a whole population of solutions is evolved. These two families have complementary characteristics: S-metaheuristics are exploitation oriented; they have the ability to intensify the search in local regions. P-metaheuristics are exploration oriented; they provide a better diversification in the entire search space. In the next chapters of this document, we will mainly use this classification.

1.1.2.1 Solution Representation

Designing any iterative metaheuristic requires an encoding (representation) of a solution. The encoding plays a major role in the efficiency and effectiveness of any metaheuristic, so it constitutes an essential step in designing a metaheuristic. The encoding must be suitable and relevant to the optimization problem at hand. Moreover, the quality of a representation has a considerable influence on the efficiency of the search operators applied on this representation. Four major encodings in the literature can be highlighted: binary encoding (e.g. knapsack, satisfiability), vector of discrete values (e.g. location problem, assignment problem), permutation (e.g. traveling salesman problem, scheduling problems) and vector of real values (e.g. continuous functions). Figure 1.2 illustrates an example of each representation.

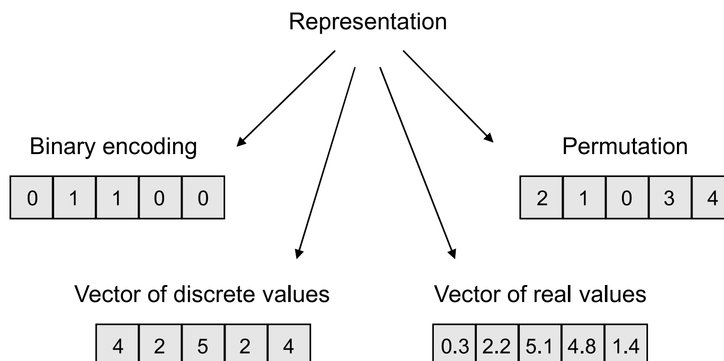


Figure 1.2: Major encodings for optimization problems.

1.1.2.2 Evaluation Function

The objective function f formulates the goal to be achieved. It associates with each solution of the search space a real value that gives the quality or the *fitness* of the solution, $f : S \rightarrow \mathbb{R}$. Then, it represents an absolute value and allows a complete ordering of all solutions of the search space. The objective function is an essential element in designing a metaheuristic. It guides the search toward "good" solutions of the search space. If the objective function is improperly defined, it can lead to non-acceptable solutions whatever is the used metaheuristic.

1.1.2.3 Principles of S-metaheuristics

S-metaheuristics are iterative techniques that have been successfully applied for solving many real and complex problems. These methods could be viewed as "walks through neighborhoods" meaning search trajectories through the solutions domains of the problems at hand. The walks are performed by iterative procedures that allow to move from a solution to another one in the solution space (see Algorithm 1).
A S-metaheuristic usually starts with a randomly generated solution. At each iteration of the algorithm, the current solution is replaced by another one selected from the set of its neighboring candidates, and so on. An evaluation function associates a fitness value to each solution indicating its suitability to the problem (selection criterion). Many strategies related to the considered S-metaheuristic can be applied in the selection of a move: best improvement, first improvement, random selection, etc.
The simplest S-metaheuristic is the hill climbing algorithm (see Algorithm 2). It starts with at a given solution. At each iteration, the heuristic replaces the current solution by a neighbor that improves the objective function. The search stops when all candidate neighbors are worse than the current solution, meaning a local optimum is reached.
Another widespread method is the tabu search algorithm [Glo89, Glo90] (see Algorithm

Algorithm 1 S-metaheuristic pseudo-code.

```
1: Generate(s_0);
2: Specific S-metaheuristic pre-treatment
3: t := 0;
4: repeat
5:    m(t) := SelectMove(s(t));
6:    s(t + 1) := ApplyMove(m(t), s(t));
7:    Specific S-metaheuristic post-treatment
8:    t := t + 1;
9: until Termination_criterion(s(t))
```

Algorithm 2 Hill climbing pseudo-code

```
1: Generate(s_0);
2: t := 0;
3: repeat
4:    m(t) := SelectBestMove(s(t));
5:    s(t + 1) := ApplyMove(m(t), s(t));
6:    t := t + 1;
7: until Local_optimum(s(t))
```

3). In this local search, the best solution in the neighborhood is selected as the new current solution even if it is not improving the current solution. This policy may generate cycles, i.e. previous visited solutions could be selected again. To avoid these cycles, the algorithm manages a memory of the moves recently applied, which is called the *tabu list*. This list is a short-term memory which contains the solutions (moves) that have been visited in the recent past.

Algorithm 3 Tabu search pseudo-code

```
1: Generate(s_0);
2: InitTabuList(l_0);
3: t := 0;
4: repeat
5:    m(t) := SelectBestAdmissibleMove(s(t));
6:    s(t + 1) := ApplyMove(m(t), s(t));
7:    l(t + 1) := UpdateTabuList(l(t));
8:    t := t + 1;
9: until Termination_criterion(s(t))
```

Other popular examples of S-meheuristics are simulated annealing, iterative local search and variable neighborhood search. A survey of the history and a state-of-the-art of S-metaheuristics can be found in [DPST06, Tal09].

1.1.2.4 Principles of P-metaheuristics

P-metaheuristics are recognized as promising methods for solving complex problems in science and industry. They start from an initial population of solutions. Then, they iteratively apply the generation of a new population and the replacement of the current population. In the generation phase, a new population of solutions is created. In the replacement phase, a selection is carried out from the current and the new populations. This process iterates until a given stopping criterion is satisfied. Popular examples of P-metaheuristics are evolutionary algorithms, ant colony optimization, scatter search and particle swarm optimization. A review of these methods is available in [Tal09]. Algorithm 4 illustrates the high-level template of P-metaheuristics.

Algorithm 4 P-metaheuristics pseudo-code.

1: Generate(P_0);
2: $t := 0$;
3: **repeat**
4:    Generate($P'(t)$);
5:    $P(t+1)$:= Replace($P(t)$, $P'(t)$);
6:    $t := t + 1$;
7: **until** Termination_criterion($P(t)$)

One of the best-known P-metaheuristics concerns evolutionary algorithms. These latter are stochastic search techniques that have been successfully applied for solving many real and complex problems. An evolutionary algorithm is an iterative technique that applies stochastic operators on a pool of individuals (see Algorithm 5). Every individual in the population is the encoded version of a tentative solution. Initially, this population is usually generated randomly. At each generation of the algorithm, solutions are selected, paired and recombined in order to generate new solutions that replace worse ones according to some criteria, and so on. An evaluation function associates a fitness value to every individual indicating its suitability to the problem (selection criterion).

Algorithm 5 Evolutionary algorithms pseudo-code.

1: Generate(P_0);
2: $t := 0$;
3: **repeat**
4:    $P'(t)$:= Selection($P(t)$);
5:    $P'(t)$:= ApplyReproductionOps($P'(t)$);
6:    $P(t+1)$:= Replace($P(t)$, $P'(t)$);
7:    $t := t + 1$;
8: **until** Termination_criterion($P(t)$)

There exists several well-accepted subclasses of evolutionary algorithms depending on rep-

resentation of the individuals or how each step of the algorithm is designed. The main subclasses of evolutionary algorithms are the genetic algorithms, genetic programming, evolution strategies, etc. A large review of these evolutionary computation techniques is done in [BFM97].

1.1.3 Parallel Models of Metaheuristics

Although the use of metaheuristics allows to considerably reduce the computational complexity of the search process, the latter remains time-consuming for many problems in diverse domains of application. In particular, in the case where the objective function and the constraints associated with the problem are resource-intensive, and the size of the search space is huge. For nontrivial problems, executing the iterative process of a simple S-metaheuristic on large neighborhoods requires a large amount of computational resources. The same phenomenon occurs when executing the reproductive cycle of a simple P-metaheuristic on long individuals and/or large populations.
In general, evaluating a fitness function for each solution is frequently the most costly operation of the metaheuristic. Consequently, a variety of algorithmic issues is being studied to design efficient heuristics. These issues commonly consist of defining new move operators, hybrid algorithms, parallel models, and so on. Parallelism arises naturally when dealing with a population of solutions (or a neighborhood), since each of the solutions belonging to it is an independent unit. Due to this, the performance of metaheuristics is significantly improved when running in parallel.
Parallel and distributed computing can be used in the design and implementation of metaheuristics for the following reasons:

- *Speed up the search*: One of the main goals of parallelizing a metaheuristic is to reduce the search time. This helps designing real-time and interactive optimization methods. This is a crucial aspect for some class of problems where there are hard requirements on the search time.

- *Improve the quality of the obtained solutions:* Some parallel models for metaheuristics allow to improve the quality of the search. Indeed, exchanging information between cooperative metaheuristics will alter their behavior in terms of searching in the landscape associated with the problem. The main goal of a parallel cooperation between metaheuristics is to improve the quality of solutions.

- *Improve the robustness:* A parallel metaheuristic may be more robust in terms of solving in an effective manner different optimization problems and different instances of a given problem. Robustness may also be measured in terms of the sensitivity of the metaheuristic to its parameters.

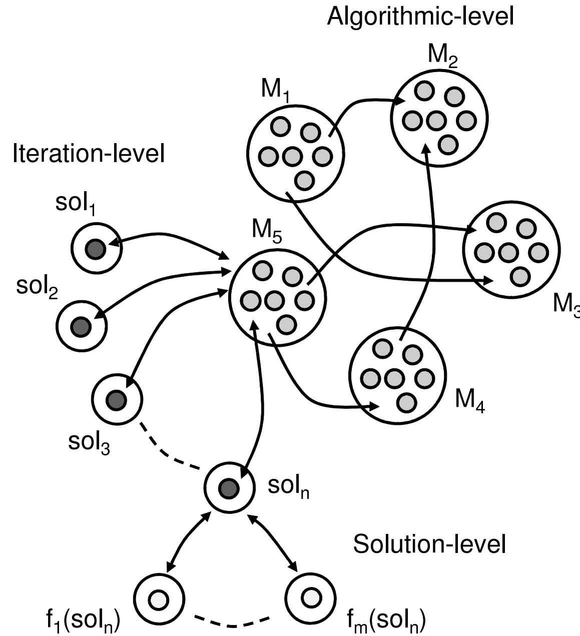


Figure 1.3: Parallel models of local search metaheuristics.

- *Solve large-scale problems:* Parallel metaheuristics allow to solve large-scale instances of complex optimization problems. An issue is to solve very large instances that cannot be solved by a sequential machine. Another similar challenge is to solve more accurate mathematical models associated with different optimization problems.

In this purpose, three major parallel models for metaheuristics can be distinguished: solution-level, iteration-level and algorithmic-level (see Figure 1.3).

- *Solution-level Parallel Model.* The focus is on the parallel evaluation of a single solution. Problem-dependent operations performed on solutions are parallelized. That model is particularly interesting when the evaluation function can be itself parallelized as it is CPU time-consuming and/or IO intensive. In that case, the function can be viewed as an aggregation of a given number of partial functions.

- *Iteration-level Parallel Model.* This model is a low-level Master-Worker model that does not alter the behavior of the heuristic. The evaluation of solutions is performed in parallel. At the beginning of each iteration, the master duplicates the solutions to be evaluated between parallel nodes. Each of them manages some candidates, and the results are returned back to the master. An efficient execution is often obtained especially when the evaluation of each solution is costly.

- *Algorithmic-level Parallel Model.* Several metaheuristics are simultaneously launched for computing better and robust solutions. They may be heterogeneous or homo-

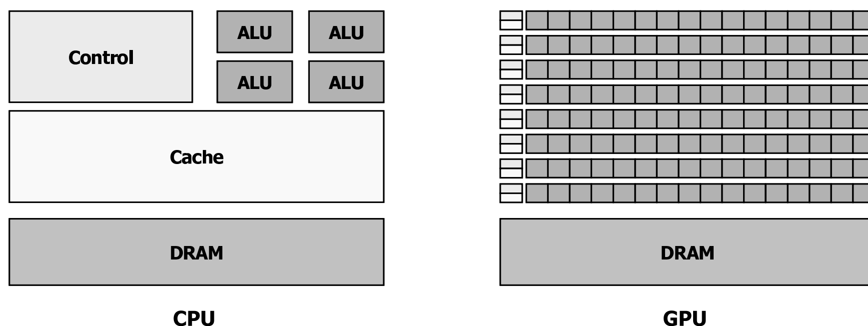


Figure 1.4: Repartition of transistors for CPU and GPU architectures.

geneous, independent or cooperative, start from the same or different solution(s), configured with the same or different parameters.

From a parallelization point of view, the solution-level model is problem-dependent and does not present many generic concepts. As a consequence, in this document, we will not deal with this parallel model.

1.2 Metaheuristics and GPU Computing

Most personal computers integrated with GPUs are usually far less powerful than their add-in counterparts. That is the reason why it would be very interesting to exploit this enormous capacity of computing to implement parallel metaheuristics. In this section, the focus is on the description of GPU computing. A clear understanding of GPU characteristics is required to provide an efficient implementation of parallel metaheuristics.

1.2.1 GPU Architecture

For years, the use of graphics processors was dedicated to graphics applications. Driven by the demand for high-definition 3D graphics on personal computers, GPUs have evolved into a highly parallel, multithreaded and many-core environment. Indeed, this architecture provides a tremendous computational horsepower and a very high memory bandwidth compared to traditional CPUs. Figure 1.4 illustrates the repartition of transistors between the two architectures.

One can see that a CPU does not have a lot of arithmetic-logic units (ALU), but a large cache and an important control unit. As a result, the CPU is specialized to manage multiple and different tasks in parallel that require lots of data. Thereby, data are stored within a cache to accelerate its accesses. The control unit will handle the instructions flow to maximize the occupation of arithmetic-logic units, and to optimize the cache management.

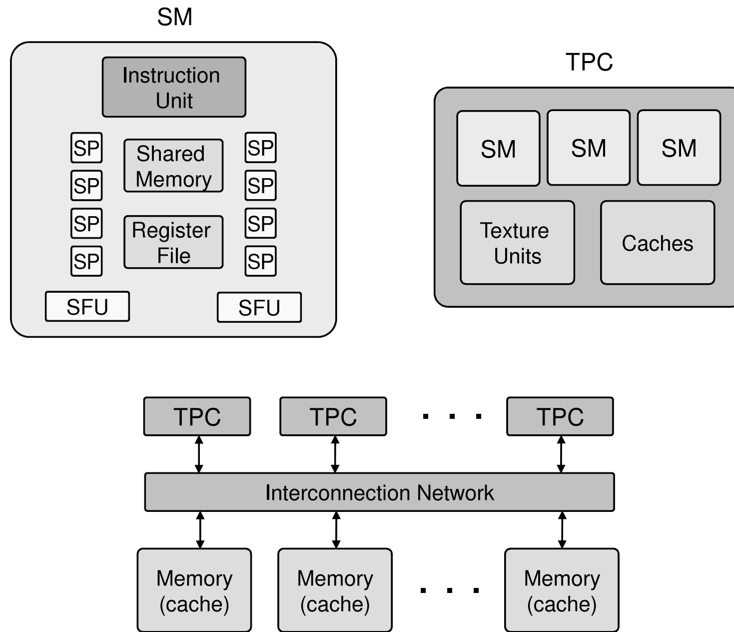


Figure 1.5: General GPU architecture composed of streaming multiprocessors.

Conversely, a GPU has a large number of arithmetic units with a limited cache and few control units. This allows the GPU to calculate in a massive and parallel way the rendering of small and independent elements, while having a large flow of data processed. Since more transistors are devoted to data processing rather than data caching and flow control, GPU is specialized for compute-intensive and highly parallel computations.

Figure 1.5 depicts the general GPU architecture. It is composed of streaming multiprocessors (SMs), each containing a certain number of streaming processors (SPs), or processor cores. Each core executes a single thread instruction in a SIMD (single-instruction multiple-data) fashion, with the instruction unit distributing the current instruction to the cores. Each core has one multiply-add arithmetic unit that can perform single-precision floating-point operations or 32-bit integer arithmetic. In addition, each SM has special functional units (SFUs), which execute more complex floating-point operations such as reciprocal sine, cosine and square root with low cycle latency.

The SM contains other resources such as shared memory and the register file. Groups of SMs belong to thread processing clusters (TPCs). The latter also contain resources (e.g. caches and texture fetch units) that are shared among the SMs. The GPU architecture comprises the collection of TPCs, the interconnection network, and the memory system (DRAM memory controllers).

Figure 1.6 gives a comparison of the execution model for both CPU and GPU architectures. Basically, a CPU thread proceeds one data element per operation. With the extension of

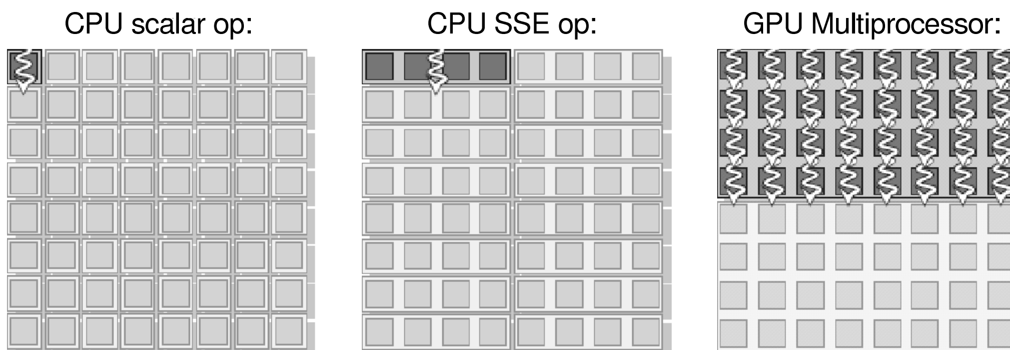


Figure 1.6: Comparison of CPU and GPU execution models.

SSE (streaming SIMD execution) instructions, such a CPU thread can operate between two and four data elements. Regarding a GPU multiprocessor, 32 threads proceed 32 data elements. These groups of 32 threads are called *warps*. They are exposed as individual threads but execute the same instruction. Therefore, a divergence in the threads execution provokes a serialization of the different instructions.
A complete review of GPU architectures can be found in [RRS+08, ND10].

1.2.2 GPU Challenges for Metaheuristics

Parallel combinatorial optimization on GPU is not straightforward and requires a huge effort at design as well as at implementation level. Indeed, several scientific challenges mainly related to the hierarchical memory management have to be achieved. The major issues are the efficient distribution of data processing between CPU and GPU, the thread synchronization, the optimization of data transfer between the different memories, the capacity constraints of these memories, etc. Such issues must be taken into account for the redesign of parallel metaheuristic models in order to solve large scale optimization problems on GPU architectures.
We aim at identifying such issues for building efficient parallel metaheuristics on GPU:

1. **Cooperation between the CPU and the GPU**. Such a step requires defining the task repartition in metaheuristics. In this purpose, the optimization of data transfer between the two components is necessary to achieve the best performance.

2. **Parallelism control**. GPU computing is based on massively parallel multi-threading, and the order in which the threads are executed is not known. Therefore, on the one hand, an efficient thread control must be applied to meet the memory constraints. On the other hand, an efficient mapping has to be defined between each candidate solution and a thread designated by a unique identifier assigned at GPU runtime.

3. **Memory management**. Optimizing the performance of GPU applications often involves optimizing data accesses which includes the appropriate use of the various GPU memory spaces. The different optimization structures have to be placed efficiently on the different memories taking into account their sizes and access latencies.

The contribution of this document is to deal with these challenges. Throughout this manuscript, we will mainly use this classification. The next sections provide more details about these different challenges.

1.2.3 General GPU Model: CPU-GPU Cooperation

In general-purpose computing on graphics processing units, the CPU is considered as a host and the GPU is used as a device coprocessor. This way, each GPU has its own memory and processing elements that are separate from the host computer. Data must be transferred between the memory space of the host and the memory of GPU during the execution of programs.
Each processor device on GPU supports the single program multiple data (SPMD) model, i.e. multiple autonomous processors simultaneously execute the same program on different data. For achieving this, the concept of kernel is defined. The kernel is a function callable from the host and executed on the specified device simultaneously by several processors in parallel. Figure 1.7 illustrates an example of the general GPU model.
Memory transfer from the CPU to the device memory is a synchronous operation which is time consuming. Indeed, PCI-e bus bandwidth and latency between CPU and GPU can significantly reduce the performance of the search. As a result, one objective when programming GPU applications is to establish an efficient task repartition. In this purpose, the optimization of data transfer between CPU and GPU is clearly an issue to be tackled.

1.2.4 GPU Threads Model: Parallelism Control

The kernel handling is dependent of the general-purpose language. For instance, CUDA [NVI11] or OpenCL [Gro10] are parallel computing environments which provide an application programming interface for GPU architectures. Indeed, these toolkits introduce a model of threads which provides an easy abstraction for SIMD architectures. The concept of a GPU thread does not have exactly the same meaning as a CPU thread. A thread on GPU can be seen as an element of data to be processed. Compared to CPU threads, GPU threads are lightweight. It means that changing the context between two threads is not a costly operation.
Regarding their spatial organization, threads are organized within so called thread blocks (see Figure 1.8). A kernel is executed by multiple equally threaded blocks. Blocks can be

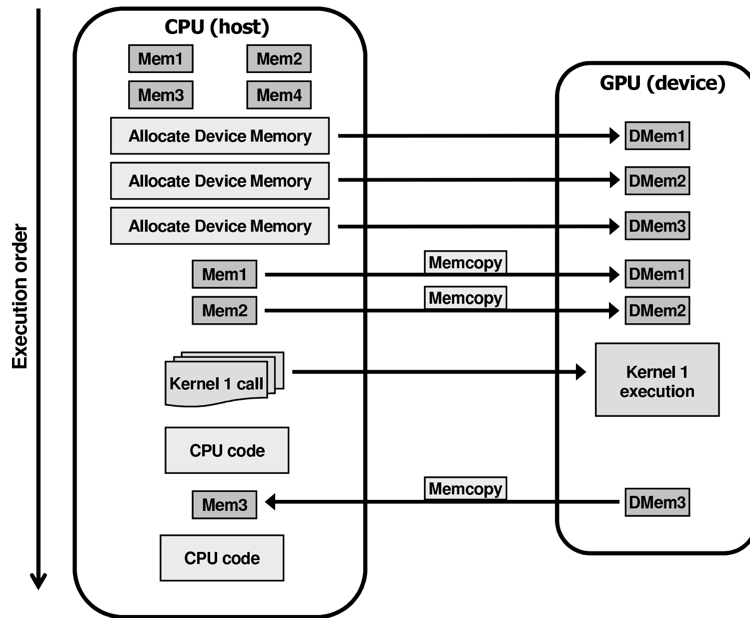


Figure 1.7: Illustration of the general GPU model. The GPU can be seen as a coprocessor where data transfers must be performed between the CPU and the GPU.

organized into a one-dimensional or two-dimensional grid of thread blocks, and threads inside a block are grouped in a similar way. Each thread is provided with a unique *id* that can be used to compute on different data. The advantage of grouping is that the number of blocks processed simultaneously by the GPU is closely linked to hardware resources. All the threads belonging to the same thread block will be assigned as a group to a single multiprocessor, while different thread blocks can be assigned to different multiprocessors. Hence, a major issue is to control the threads parallelism to meet the memory constraints.

1.2.5 Kernel Management: Memory Management

From a hardware point of view, graphics cards consist of streaming multiprocessors, each with processing units, registers and on-chip memory. Since multiprocessors are organized according to the SPMD model, threads share the same code and have access to different memory areas. Figure 1.9 illustrates these different available memories and connections with thread blocks.

Communication between the CPU host and its device is done through the global memory. Since in some GPU configurations, this memory is not cached and its access is slow, one needs to minimize accesses to global memory (read/write operations) and reuse data within the local multiprocessor memories. Graphics cards provide also read-only texture memory to accelerate operations such as 2D or 3D mapping. Texture memory units are provided to

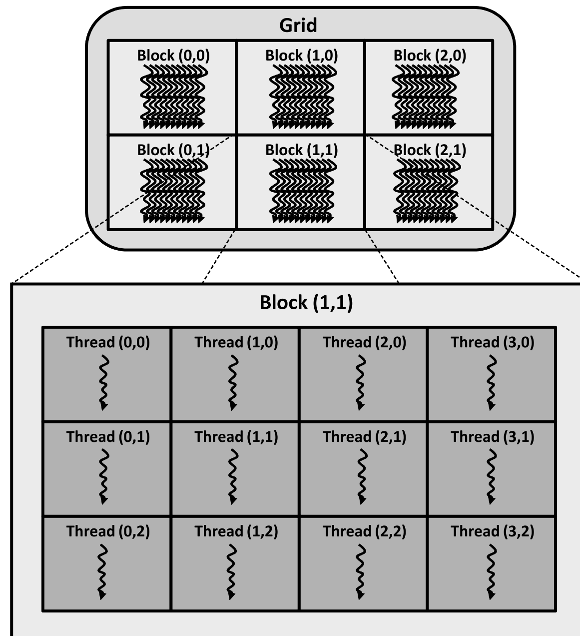
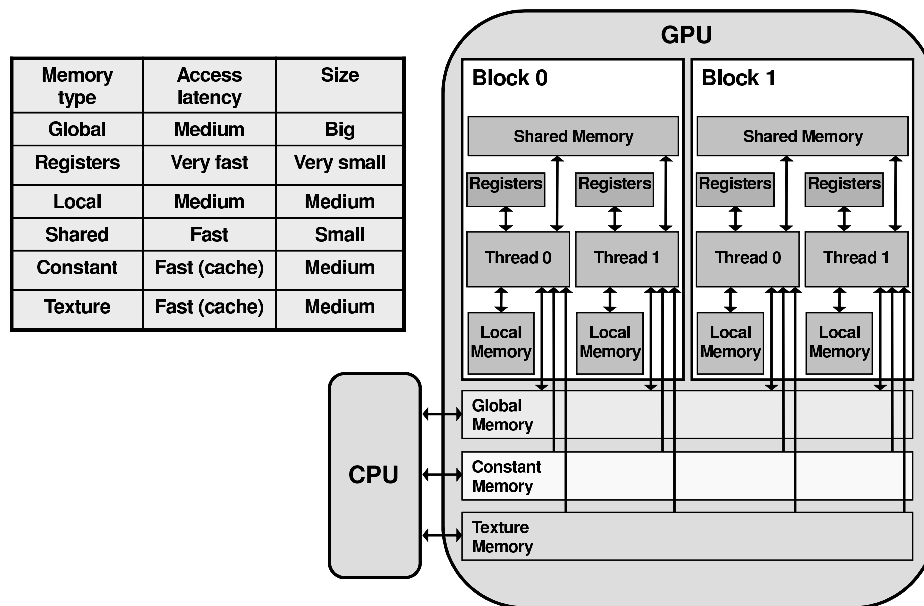


Figure 1.8: GPU threads model. GPU threads are organized into block structures.



Memory type	Access latency	Size
Global	Medium	Big
Registers	Very fast	Very small
Local	Medium	Medium
Shared	Fast	Small
Constant	Fast (cache)	Medium
Texture	Fast (cache)	Medium

Figure 1.9: GPU memory model. Different on-chip memories and connections with thread blocks are available.

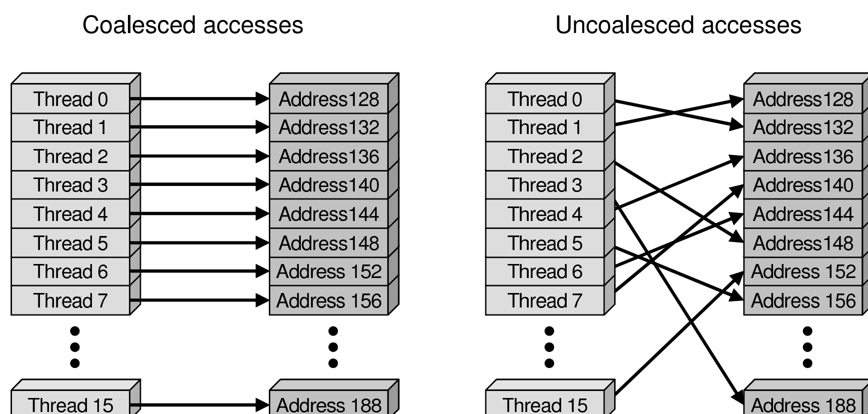


Figure 1.10: Illustration of accesses patterns that lead to coalesced and uncoalesced accesses to the global memory.

allow faster graphic operations. This way, binding texture on global memory can provide an alternative optimization. Indeed, it improves random accesses or uncoalesced memory access patterns that occur in common applications. Constant memory is read only from kernels and is hardware optimized for the case where all threads read the same location. Shared memory is a fast memory located on the multiprocessors and shared by threads of each thread block. This memory area provides a way for threads to communicate within the same block. Registers among streaming processors are exclusive to an individual thread; they constitute a fast access memory. In the kernel code, each declared variable is automatically put into registers. Local memory is a memory abstraction and is not an actual hardware component. In fact, local memory resides in the global memory allocated by the compiler. Complex structures such as declared arrays will reside in local memory.

Regarding the execution model, each block of threads is split into SIMD groups of threads called warps. At any clock cycles, each processor of the multiprocessor selects a warp that is ready to execute the same instruction on different data. For being efficient, global memory accesses must be coalesced. This means that a memory read by consecutive threads in a warp is combined by the hardware into several memory reads. The requirement is that threads of the same warp must read global memory in an order manner (see Figure 1.10). Global memory accesses patterns that are non-coalesced may significantly decrease the performance of a program. As a result, an efficient management of the optimization structures with the different available memories has to be considered.

1.3 Related Works on Parallel Metaheuristics

1.3.1 Metaheuristics on Parallel and Distributed Architectures

The rapid development of technology in designing processors (e.g. multicore processors and dedicated architectures), networks (e.g. local area networks such as Myrinet and Infiniband or wide area networks such as optical networks), and data storage, has made the use of parallel computing more and more popular. Such evolution raises new challenges for the design and implementation of parallel metaheuristics. Indeed, sequential architectures are reaching physical limitation. Nowadays, even laptops and workstations are equipped with multicore processors, which represent a given class of parallel architectures. Moreover, the cost/performance ratio is constantly decreasing. The proliferation of powerful workstations and fast communication networks has shown the emergence of clusters of processors (COWs), networks of workstations, and large-scale network of machines (grids) as platforms for high-performance computing.

During these two last decades, many parallel approaches and implementations have been proposed for metaheuristics [ABR03, ALNT04, AT02, AK96, BHX00, CB96, CG02, CGHM04, GrTA99, GLMBMPMV03, HL00, LL96, RL02, PDB10, SBA97]. Some of them using massively parallel processors [CSK93], clusters of workstations [CTG95, GPR94, BSB$^+$01] and shared memory or SMP machines [JRG09, Bev02]. These contributions have been later revisited for large-scale computational grids [TMT07].

These architectures often exploit the coarse-grained asynchronous parallelism based on work-stealing. This is particularly the case for computational grids. To overcome the problem of network latency, the grain size is often increased, limiting the degree of parallelism.

Recently, GPU accelerators have emerged as a powerful support for massively parallel computing. Indeed, these architectures offer a substantial computational horsepower and a remarkably high memory bandwidth compared to CPU-based architectures. For instance, the parallel evaluation of the solutions (iteration-level) is a Master-Worker and a problem-independent, regular data-parallel application. Therefore, GPU computing may be highly efficient in executing such synchronized parallel algorithms that involve regular computations and data transfers.

In general, for distributed architectures, the global performance in metaheuristics is limited by high communication latencies whilst it is just bounded by memory access latencies in GPU architectures. Indeed, when evaluating solutions in parallel, the main obstacle in distributed architectures is the communication efficiency. GPUs are not that versatile.

However, since the execution model of GPUs is purely SIMD, it may not be well-adapted for few irregular problems in which the execution time cannot be predicted at compile time

and varies during the search. For instance, this happens when the evaluation cost of the objective function depends on the solution. When dealing with such problems in which the computations or the data transfers become irregular or asynchronous, parallel and distributed architectures such as COWs or computational grids may be more appropriate.

1.3.2 Research Works on GPU-based Metaheuristics

With the emergence of standard programming languages on GPU and the arrival of compilers for these languages, combinatorial optimization on GPU has generated a growing interest. Historically, due to their embarrassingly parallel nature, P-metaheuristics such as evolutionary algorithms have been the first subject of parallelization on GPU architectures. One of the first pioneering works on genetic algorithms was proposed by Wong *et al.* [WWF05, WW06, FWW07]. In their works, the population evaluation and a specific mutation operator (Cauchy mutation) are performed on GPU. However, since replacement and selection operators are implemented on CPU, massive data transfers are performed between the CPU and the GPU. Such techniques limit the performance of the algorithm. Concurrently, to deal with this drawback, Yu *et al.* [YCP05] were the first authors to establish a full parallelization of the genetic process on GPU. To achieve this, the population is organized into a 2D toroidal grid, which allows to apply selection operators similar to those ones used for cellular genetic algorithms. However, the implementation is only specific to a vector of real values. Later, Li *et al.* [LWHC07] extended this work for binary representations, and implemented further specific genetic operators. In a similar manner, Luo *et al.* [LL06] were among the first authors to implement a cellular genetic algorithm on GPU for the 3-SAT problem. To perform this, the semantics of the original cellular algorithm are completely modified to meet the GPU constraints.

The previous implementations quoted above are based on a transformation of evolutionary structures into a series of raster operations on GPU using shader libraries based on Direct3D or OpenGL. In other words, to implement metaheuristics with such libraries, one needs to solve the problem of texture storage of relevant information in arrays.

The following works on GPU are implemented with the CUDA development toolkit, which allows programming on GPUs in a more accessible C-like language. In addition to this, such a thread-based approach is easier in terms of reproducibility in comparison with shader libraries.

Zhu suggested in [Zhu09] an evolution strategy algorithm to solve a bench of continuous problems using the CUDA toolkit. In his implementation, multiple kernels are designed for some evolutionary operators such as the selection, the crossover, the evaluation and the mutation. The rest of the search process is handled by the CPU.

Later, Arora *et al.* presented a similar implementation in [ATD10] for genetic algorithms.

In addition, the contribution of their work is to investigate the effect of a bench of parameters (e.g. threads size, problem size or population size) on the acceleration of their GPU implementation in regards with a sequential genetic algorithm.

Tsutsui *et al.* were among the first authors to establish memory management concepts of combinatorial optimization problems [TF09]. In their implementation for the quadratic assignment problem, accesses to the global memory (the population) are coalesced, the shared memory is used to store as many individuals as possible, and matrices are associated with the constant memory. Their approach is based on a full parallelization of the search process to deal with data transfers. For doing that, the global genetic algorithm is divided into multiple independent genetic algorithms, where each sub-population represents a thread block. The obtained speed-ups are of course less convincing than for continuous problems, due to the management of data structures in combinatorial problems.

In [MBL+09], Maitre *et al.* submitted a framework tentative for the automatic parallelization of the evaluation function on GPU. In this purpose, the user does not need to know about any CUDA keywords and only the evaluation function code must be specified. Such a strategy allows to evaluate the population in a transparent way on GPU. However, this automatic parallelization presents some drawbacks. Indeed, it lacks flexibility due to the data transfers and non-optimized memory accesses. Moreover, the application is restricted to problems which do not require any data structures (e.g. continuous problems).

Another framework tentative were introduced in [SBPE10] by Soca *et al.* for cellular genetic algorithms. Multiple kernels are used for each evolutionary operator. The platform offers a collection of predetermined operators (three crossovers and two mutations) that the user must instantiate. In addition, management of problem structures in global memory seems to be taken into account since an application to the quadratic assignment problem is done. Unfortunately, unless the previous one, the framework seems to remain in the design step since no link is available for downloading.

Another implementation of a cellular evolutionary algorithm is provided by Vidal *et al.* in [VA10a]. It is based on a full parallelization of the search process on GPU. An application of the approach is made for continuous and discrete problems without any problem structures. Later, the authors submitted a multi-GPU implementation of their algorithm in [VA10b]. Nevertheless, due to the challenging issue of the context management (e.g. two separate GPU global memories), their multi-GPU implementation does not provide any significant benefits in terms of performance.

Zhang *et al.* introduced a design of an island model on GPU in [ZH09]. The parallelization of the entire algorithm is performed on GPU. Each sub-population is stored on the shared memory. Regarding the topology of the different islands, they are similar to the exchange topology present in cellular evolutionary algorithm (toroidal grid). Unfortunately, the

model only remains in the design step since the authors did not produce any experimental results.

Pospichal *et al.* performed an implementation similar to the previous model for continuous optimization problems in [PJS10]. Each sub-population is stored on the shared memory and organized according to a ring topology. The obtained speed-ups are impressive in comparison with a sequential algorithm (thousands of times). However, the implementation is only dedicated to few continuous optimization problems.

Since no general methods can be outlined from the two previous works, we investigated the parallel island model on GPU in [6]. We addressed its redesign, implementation, and associated issues related to the GPU execution context. In this purpose, we designed three parallelization strategies involving different memory managements. We demonstrated the effectiveness of the proposed approaches and their capabilities to fully exploit the GPU architecture.

Regarding S-metaheuristics, Janiak *et al.* implemented a multi-start tabu search algorithm applied to the traveling salesman problem and the flowshop scheduling problem [JJL08]. Using shader libraries, the parallelization is entirely performed on GPU, and each thread process is associated with one tabu search. However, such a parallelization is not effective since a large number of local search algorithms is required to cover the memory access latency.

Concurrently, a similar approach based on the CUDA toolkit was proposed by Zhu *et al.* [ZCM08]. The implementation has been applied to the quadratic assignment problem, where the memory management of optimization structures is made on the global memory. Nevertheless, the global performance is limited to the instance size, since each thread is associated with one local search.

Although the multi-start model has already been applied in the context of the tabu search on GPU, it has never been widely investigated in terms of reproducibility and memory management. We provided in [5] a general methodology for the design of multi-start algorithms applicable to any local search algorithms such as hill climbing, tabu search or simulated annealing. Furthermore, we contributed with efficient associations between the different available memories and the data commonly used for these algorithms.

However, as quote above, the application of the multi-start model on GPU is limited since a large number of local search algorithms is required at launch time to be effective. As a matter of fact, the parallelization of the evaluation of neighborhood on GPU might be more valuable. In this purpose, we came up with the pioneering work in [9] on the redesign of the parallel evaluation of the neighborhood on GPU. We introduced the generation of neighbors on the GPU side to minimize the data transfers. Furthermore, we proposed to manage the commonly used structures in combinatorial optimization with the different

available memories.

Munawar *et al.* introduced a hybrid genetic algorithm on GPU in [MWMA09a]. In their implementation, an island model is implemented where each population represents a cellular genetic algorithm. In addition to this, a hill climbing algorithm follows the mutation step of the hybrid genetic algorithm. Each sub-population is associated with the shared memory and traditional code optimization such as memory coalescing is performed. The implementation is performed for the maximum satisfiability problem.

Since a full parallelization is investigated, the previous work requires that the hill climbing must be combined with the island model. To perform a hybridization with a local search in the general case, we contributed with the redesign of hybrid evolutionary algorithms on GPU in [7]. In this purpose, the focus is on the generation of the different neighborhoods on GPU, corresponding to each individual of the evolutionary process to be mutated. Such a mechanism guarantees more flexibility since any local search algorithms can be broached with any evolutionary algorithms.

Wong was the first author to introduce a multiobjective evolutionary algorithm on GPU in [Won09]. In his implementation, most of the multiobjective algorithm (NSGA-II) is implemented on GPU except the selection of non-dominated solutions.

In a similar manner, we contributed in [3] with the first multiobjective local search algorithms on GPU. The parallelization strategy is based on the parallel evaluation of the neighborhood on GPU using a set of non-dominated solutions to generate the neighborhood.

Table 1.1 reports the major works on GPU according to the classification proposed in Section 1.2.2. Basically, most approaches of the literature are based on either the parallel evaluation of solutions on GPU (iteration-level) or the execution of simultaneous independent/cooperative algorithms (algorithmic-level). Regarding the CPU-GPU cooperation, for the first category, some implementations also consider the parallelization of other treatments on GPU (e.g. selection or variation operators in evolutionary algorithms). One may argue on the validity of these choices since an execution profiling may show that such treatments are negligible in comparison with the evaluation of solutions. As quoted above, a full parallelization of metaheuristics on GPU may be also performed to reduce the data transfers between the CPU and the GPU. In this case, the original semantics of the metaheuristic are modified to fit the GPU execution model. This can explain the reason why an important group of works only deals with concurrent independent/cooperative algorithms.

For the parallelism control, most implementations associate one thread with one solution. In addition, some cooperative algorithms may take advantage of the threads model by associating one threads block with one sub-population. However, to the best of our knowledge, no work has been investigated to efficiently manage the threads parallelism to

Table 1.1: Classification of the major works of the literature.

Authors	**CPU-GPU cooperation**	**Parallelism control**	**Memory management**	**Optimization problems**
Wong *et al.* [WWF05, WW06, FWW07]	evaluation and mutation on GPU	panmictic one thread per individual	texture	continuous
Yu *et al.* [YCP05]	full parallelization on GPU	2D toroidal grid one thread per individual	texture	continuous
Li *et al.* [LWHC07]	full parallelization on GPU	2D toroidal grid one thread per individual	texture	continuous and binary
Luo *et al.* [LL06]	full parallelization on GPU	cellular one thread per individual	texture	3-SAT
Zhu [Zhu09]	evaluation, selection and variation on GPU	2D toroidal grid one thread per individual	global, shared and texture memory	continuous
Arora *et al.* [ATD10]	evaluation, selection and variation on GPU	2D toroidal grid one thread per individual	global and shared memory	continuous and binary
Tsutsui *et al.* [TF09]	full parallelization on GPU	2D toroidal grid one block per population	global, constant and shared memory	quadratic assignment problem
Maitre *et al.* [MBL$^+$09]	evaluation on GPU	panmictic one thread per individual	global memory	continuous
Soca *et al.* [SBPE10]	evaluation and variation on GPU	cellular one thread per individual	global and shared memory	quadratic assignment problem
Vidal *et al.* [VA10a]	full parallelization on GPU	cellular one thread per individual	global and shared memory	continuous and discrete
Zhang *et al.* [ZH09]	full parallelization on GPU	island model one block per population	global and shared memory	-
Pospichal *et al.* [PJS10]	full parallelization on GPU	island model one block per population	global and shared memory	continuous
Luong *et al.* [6]	evaluation and full parallelization on GPU	island model one block per population	global and shared memory	continuous
Janiak *et al.* [JJL08]	full parallelization on GPU	multi-start one thread per algorithm	texture	traveling salesman problem and flowshop problem
Zhu *et al.* [ZCM08]	full parallelization on GPU	multi-start one thread per algorithm	global memory	quadratic assignment problem
Luong *et al.* [5]	full parallelization on GPU	multi-start one thread per algorithm	global and texture memory	quadratic assignment problem
Luong *et al.* [9]	generation and evaluation on GPU	neighborhood one thread per neighbor	global and texture memory	permuted perceptron problem
Munawar *et al.* [MWMA09a]	full parallelization on GPU	island model+cellular one block per population	global and shared memory	maximum satisfiability problem
Luong *et al.* [7]	generation and evaluation on GPU	neighborhood one thread per neighbor	global and texture memory	quadratic assignment problem
Wong [Won09]	evaluation, selection and variation on GPU	panmictic one thread per individual	global and shared memory	continuous
Luong *et al.* [3]	generation and evaluation on GPU	neighborhood one thread per neighbor	global and texture memory	flowshop scheduling problem

meet the memory constraints. When dealing with a large set of solutions or large problem instances, the previous implementations might not be robust. We will show in Chapter 3 how an efficient thread control allows to introduce fault-tolerance mechanisms in GPU applications.

Regarding the memory management, in some implementations, no explicit efforts have been made for memory access optimizations. For instance, memory coalescing is used to be one of the key element for speedups in CUDA, and local memories could be additionally considered to reduce non-coalesced accesses. Some authors have just relied on the simple way to use the shared memory to cache spatially local accesses to global memory, which does not guarantee performance improvement. In some other implementations, explicit efforts have been performed to handle optimization structures with the different available memories. However, no general guideline can be outlined from the previous works. Indeed, most of the time, these memory associations strictly depend on the target optimization problem (e.g. small size of problems instances or no data inputs). We will try to examine such issues for the general case in Chapter 4.

Many other works on P-metaheuristics on GPU have been proposed so far. The parallelization strategies used for these implementations are similar to the prior techniques mentioned above. These works include particle swarm optimization [MCD09, ZT09, RK10], ant colonies [BOL+09, SAGM10, TF11, CGU+11], genetic programming [HB07, Chi07, LB08, Lan11] and other evolutionary computation techniques [MWMA09b, dPVK10, FKB10].

In comparison with previous works on P-metaheuristics, the spread of S-metaheuristics on GPU does not occur at the same pace. Indeed, the parallelization on GPU architectures is harder, due to the improvement of a single solution (and not a population of solutions). In Chapter 2 and Chapter 3, we will fully contribute on the design and implementation of S-metaheuristics on GPU.

1.4 Experimental Protocol

In this section, we will introduce the protocol used for the experiments presented in the rest of the document. It corresponds to the different optimization problems implemented for the experiments, the different hardware graphics cards and the analysis of the different statistical tests.

1.4.1 Optimization Problems

To validate the approaches proposed in this manuscript, five optimization problems with different encodings have been considered on GPU: the permuted perceptron problem, the quadratic assignment problem, the continuous Weierstrass function, the traveling salesman

problem and the Golomb rulers.

1.4.1.1 Permuted Perceptron Problem

In [Poi95], Pointcheval introduced a cryptographic identification scheme based on the perceptron problem, which seems to be suited for resource-constrained devices such as smart cards. An ϵ-vector is a vector with all entries being either +1 or -1. Similarly, an ϵ-matrix is a matrix in which all entries are either +1 or -1. The permuted perceptron problem is defined as follows:
Given an ϵ-matrix A of size $m \times n$ and a multi-set S of non-negative integers of size m, find an ϵ-vector V of size n such that $\{\{(AV)_j/j = \{1, \ldots, m\}\}\} = S$.
The permuted perceptron problem has been implemented using a binary encoding. Part of the full evaluation of a solution can be seen as a matrix-vector product (quadratic time complexity). Regarding S-metaheuristics by using a structure in a linear space complexity, the evaluation of a neighbor (Δ evaluation) can be reduced in linear time.

1.4.1.2 The Quadratic Assignment Problem

The quadratic assignment problem [BcRW98] arises in many applications such as facility location or data analysis. Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ matrices of positive integers. Finding a solution of the quadratic assignment problem is equivalent to finding a permutation $\pi = (1, 2, \ldots, n)$ that minimizes the objective function:

$$z(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\pi(i)\pi(j)}$$

The problem has been implemented using a permutation representation. The evaluation function has a $O(n^2)$ time complexity where n is the instance size. When considering a Δ evaluation for S-metaheuristics, some moves evaluations can be calculated in constant time, and some other has a time complexity of $O(n)$. The requirement is a structure which stores previous Δ evaluations in a quadratic space complexity.

1.4.1.3 The Weierstrass Continuous Function

The Weierstrass functions belong to the class of continuous optimization problems. These functions have been widely used for the simulation of fractal surfaces. According to [LV98], Weierstrass-Mandelbrot functions are defined as follows:

$$W_{b,h}(x) = \sum_{i=1}^{\infty} b^{-ih} sin(b^i x) \quad with\ b > 1\ and\ 0 < h < 1 \tag{1.1}$$

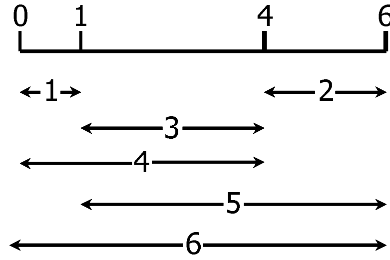


Figure 1.11: A Golomb ruler and its associated distances.

The parameter h has an impact on the irregularity ("noisy" local perturbation of limited amplitude) and these functions possess many local optima. The problem has been implemented using a vector of real values. The domain definition has been set to $-1 \leq x_k \leq 1$, h has been fixed to 0.25, and the number of iterations to compute the approximation to 100 (instead of ∞). Such parameters are in accordance with the one dealt with in [LV98]. The complexity of the evaluation function is quadratic. Regarding S-metaheuristics, since traditional neighborhoods for continuous optimization impact on all the elements composing a solution, no easy technique can be applied for computing a Δ evaluation.

1.4.1.4 The Traveling Salesman Problem

Given n cities and a distance matrix $d_{n,n}$, where each element d_{ij} represents the distance between the cities i and j, the traveling salesman problem [Kru56] consists in finding a tour which minimizes the total distance. A tour visits each city exactly once.
The chosen representation is a permutation structure. The evaluation function can be performed in a linear time complexity. Regarding S-metaheuristics, a usual neighborhood for the traveling salesman is a two-opt operator. In such a neighborhood, moves evaluations can be performed in constant time.

1.4.1.5 The Golomb Rulers

Golomb rulers are numerical sequences or a class of graphs named for Solomon W. Golomb [GB77] which have applications in a wide variety of fields including communications when setting up an interferometer for radio astronomy.
A Golomb ruler is an ordered sequence of positive integer numbers. These numbers are referred to as marks. The distance is the difference between two points, and all distances must be distinct for that ruler. The last mark is referred to as the length of the Golomb ruler. By convention, the first mark of the ruler must be placed at the position 0. Figure 1.11 shows an example for a solution with 4 marks.
More exactly, a Golomb ruler with n marks is a set $a_1 < a_2 < ... < a_n$ of positive integer

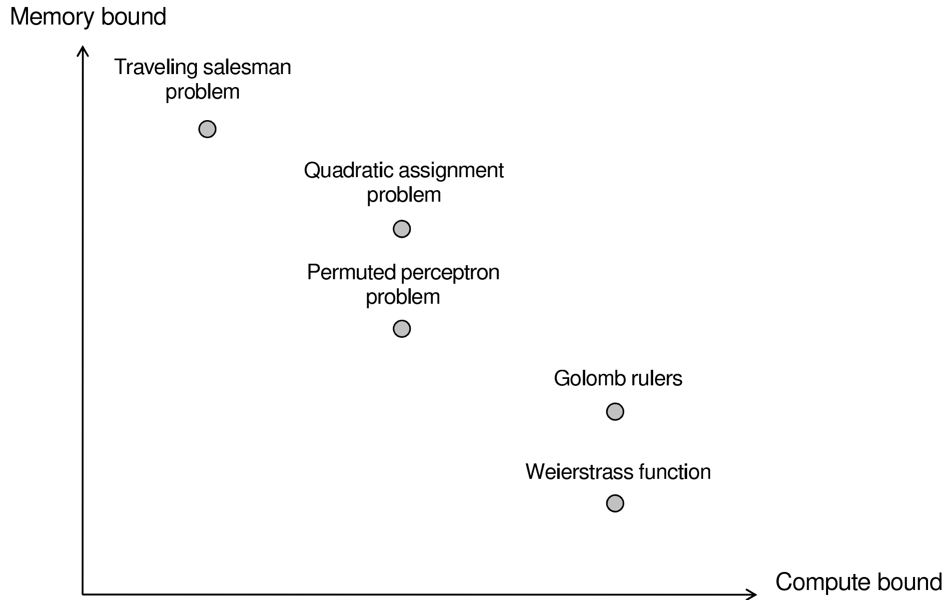


Figure 1.12: Compute bound and memory bound applications.

positions such that the differences $a_i - a_j$ $(\forall i \neq j)$ are distinct and where $a_1 = 0$. By convention, a_n is the length of the solution and this ruler has $\frac{n \times (n-1)}{2}$ distances. A solution with n marks is optimal if it does not exist any shorter Golomb ruler with the same number of marks.

The problem has been implemented using a discrete vector representation. A solution evaluation has a $O(n^3)$ time complexity. When considering a S-metaheuristic, moves evaluations can be performed in a quadratic time involving a structure (quadratic space complexity).

1.4.1.6 Problem Characteristics

The quadratic assignment problem and the traveling salesman problem are permutation problems, the permuted perceptron problem is based on the binary encoding, the Weierstrass function is represented by a vector of real values and the Golomb rulers requires a discrete vector encoding.

The selected problems deal with the four principal encodings of the literature presented in Section 1.1.2.1. They present different time complexities which worth being investigated. Regarding the traveling salesman problem, it has been chosen since very large instances of this problem are considered. Indeed, applications which require a huge number of threads might fail at runtime, due to the hardware registers limitation.

To have an appreciation of the problems performance, one has to examine the differ-

Table 1.2: Configurations used for the experiments.

Machines	CPU / GPU	GFLOPS	Cores	Memory
Configuration 1	Core 2 Duo T5800 GeForce 8600M GT	16 60.8	2 CPU cores 32 GPU cores	2 GB 512 MB
Configuration 2	Core 2 Quad Q6600 GeForce 8800 GTX	38.4 345.6	4 CPU cores 128 GPU cores	4 GB 768 MB
Configuration 3	Intel Xeon E5450 GeForce GTX 280	96 885.12	8 CPU cores 240 GPU cores	4 GB 1 GB
Configuration 4	Intel Xeon E5620 Tesla M2050	76.8 1029.28	8 CPU cores 448 GPU cores	4 GB 3 GB

ent operations composing an application to point out the limiting factors. A program is considered *compute bound* if the number of calculating operations (e.g. additions or multiplications) dominates the total number of operations. Conversely, an application is said *memory bound* if memory accesses are the leading operations. Figure 1.12 illustrates these concepts for the evaluation functions of the different problems. Such a classification is achieved by investigating the number of operations in the evaluation function code. Indeed, a code profiling highlights that the evaluation function represents the time-consuming in a metaheuristic implementation. For instance, most operations in the Weierstrass evaluation function are power calculations, sine functions and multiplications; while in the traveling salesman problem, the leading operations are memory accesses. Since a GPU is composed of lot of arithmetic-logical units, it can clearly outperform the CPU when considering pure compute bound algorithms. Thereby, when dealing with both compute bound and memory bound algorithms, one of the issues is to take advantage of both the GPU execution model and the GPU local memories to achieve the best performance.

1.4.2 Machines Configuration

The different approaches proposed in this document have been experimentally validated on the five optimization problems quoted above, using four GPU configurations (see Table 1.2). These latter have different performance capabilities in terms of threads that can be created simultaneously and memory caching. The GPU cards have a different number of cores (respectively 32, 128, 240 and 448), which determines the number of active threads being executed. The three first cards are relatively modest and old, whereas the last one is a modern Fermi card. The number of floating point operations per second (GFLOPS) represents a metric for a device performance. Table 1.3 reports the technical specifications for the different graphic cards.

As previously said, each multiprocessor has a number of registers that are shared between its processing cores. The maximum number of threads per block is essentially constrained by the number of registers that can be distributed across all the threads running in all

Table 1.3: Technical specifications between the different cards.

Specifications	8600M GT	8800 GTX	GTX 280	Tesla M2050
Maximum number of threads per block	512		1024	
Maximum number of active blocs per multiprocessor	512			
Maximum number of active warps per multiprocessor	24		32	48
Number of registers per multiprocessor	8K		16K	32K
Maximum amount of shared memory per multiprocessor	16KB			48KB
Number of schedulers per multiprocessor	1			2
Coalescing rules relaxation on global memory	no		yes	
L1/L2 cache on global memory	no			yes

the blocks assigned to a multiprocessor. The warp size is the number of threads running concurrently on a multiprocessor. These threads are running both in parallel and pipelined. When a multiprocessor is given thread blocks to execute, it partitions them into warps that get processed by a warp scheduler at runtime.
The shared memory is a small memory within each multiprocessor that can be read/written by any thread in a block assigned to that multiprocessor. The thread limit constrains the amount of cooperation between threads. Indeed, only threads within the same block can synchronize with each other, and exchange data through the fast shared memory in a multiprocessor. The way the threads access global memory also affects the global performance. The execution process goes much faster if the GPU can coalesce several global addresses into a single contiguous access over the wide data bus that goes to the external SDRAM. Such coalescing rules are relaxed when dealing with recent architectures. In Fermi series such that the Tesla M2050, there is L1/L2 cache structure. When threads require more registers than the hardware can support, they will spill into L1 cache, which is very fast. If L1 cache is full, or there are other conflicts, these registers will spill into L2 cache, which is significantly larger. Still, L2 cache is much faster than accessing memory off chip. Texture memory in prior architectures used to be an alternative to cache global memory.

1.4.3 Metric and Statistical Tests

To assess the performance of the proposed GPU-based algorithms in this thesis, execution times and acceleration factors are reported in comparison with a single-core CPU. The

speed-up from GPU implementations can be given by:

$$Acceleration = \frac{GPU\ time}{CPU\ time}$$

Statistical analysis must be performed to ensure that the conclusions deduced from the experiments are meaningful. Furthermore, an objective is also to prove that a specific algorithm outperforms another one. However, the comparison between two average values might be not enough. Indeed, it may differ from the comparison between two distributions. Therefore, a test has to be performed to ensure the statistical significance of the obtained results. In other words, one has to determine whether an observation is likely to be due to a sampling error or not.

The first test consists in checking if the data set is normally distributed from a number of experiments above 30. This is done by applying a Kolmogorov-Smirnov's test, which is a powerful and accurate method.

To compare two different distributions (i.e. whether an algorithm is better than another or not), the Student's t-test is widely used to compare averages of normal data. The prerequisites for such a test are to check the data normality (Kolmogorov-Smirnov) then to examine the variances equality of the two samples. This latter can be done by the Levene's test, which is an inferential statistic used to assess the equality of variances in different sample.

The statistical confidence level is fixed to 95%, and the p-values are represented for all the statistical analysis tables.

Since lot of experiments are presented in this document, Section .2 in Appendix 5.3.1.2 only reports the results, in which statistical test cannot conclude if an algorithm is better than another one.

Conclusion

In this chapter, we have described all the concepts necessary to the general understanding of the document. In this purpose, we have introduced the optimization context, the principles of metaheuristics, the different optimization problems at hand, and the individual GPU configurations. More important, we have mainly focused on the principles of parallel metaheuristics for parallel and GPU architectures. Understanding the hierarchical organization of the GPU architecture is useful to provide an efficient implementation of parallel metaheuristics.

- **Parallel models of metaheuristics**. The parallel evaluation of a single solution (solution-level) is specific to the problem at hand. Hence, it is not addressed in the present document. Thereby, the focus is on the parallel evaluation of solutions (iteration-level) and the cooperative parallel model (algorithmic-level). Indeed, these two last models provide generic aspects that are independent of the addressed problem.

- **GPU challenges**. The goal of this thesis is to redesign the different parallel models on GPU architectures. In this purpose, we have proposed a classification of the different challenges involved in the design and implementation of metaheuristics on GPU accelerators. These challenges concern the efficient cooperation between the CPU and the GPU, the efficient parallelism control and the efficient management of the hierarchical memory. These three challenges constitute the heart of this document. In relation to this classification, we have shown how each work of the literature can be classified into different categories according to these challenges. The next three chapters will be dedicated to solving each of these challenges for the design of GPU-based metaheuristics.

Chapter 2

Efficient CPU-GPU Cooperation

The scope of this chapter is to establish an efficient cooperation between the CPU and the GPU, which requires to share the work and to optimize the data transfer between the two components. First, we will briefly describe a parallelization scheme on GPU common to all metaheuristics. Then, we will focus on the optimization of data transfers between the CPU and the GPU. Indeed, this represents one of the critical issues to achieve the best performance in GPU applications. We will show how this optimization impacts in particular on S-metaheuristics on GPU. Finally, we will investigate the powerful potential of GPU-based S-metaheuristics compared to cluster or grid-based parallel architectures.

Contents

2.1 Task Repartition for Metaheuristics on GPU 37

2.1.1 Model of Parallel Evaluation of Solutions 37

2.1.2 Parallelization Scheme on GPU 37

2.2 Data Transfer Optimization 38

2.2.1 Generation of the Neighborhood in S-metaheuristics 39

2.2.2 The Proposed GPU-based Algorithm 40

2.2.3 Additional Data Transfer Optimization 42

2.3 Performance Evaluation 43

2.3.1 Analysis of the Data Transfers from CPU to GPU 43

2.3.2 Additional Data Transfer Optimization 49

2.4 Comparison with Other Parallel and Distributed Architectures 50

2.4.1 Parallelization Scheme on Parallel and Distributed Architectures 52

2.4.2 Configurations 52

2.4.3 Cluster of Workstations 54

2.4.4 Workstations in a Grid Organization 55

Main publications related to this chapter

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU Computing for Local Search Metaheuristic Algorithms. *IEEE Transactions on Computers*, in press, 2011.

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU-based Approaches for Multiobjective Local Search Algorithms. A Case Study: the Flowshop Scheduling Problem. *11th European Conference on Evolutionary Computation in Combinatorial Optimization, EVOCOP 2011*, pages 155–166, volume 6622 of Lecture Notes in Computer Science, Springer, 2011.

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Local Search Algorithms on Graphics Processing Units. A case study: the Permutation Perceptron Problem. *Evolutionary Computation in Combinatorial Optimization, 10th European Conference, EvoCOP 2010*, pages 264–275, volume 6022 of Lecture Notes in Computer Science, Springer, 2010.

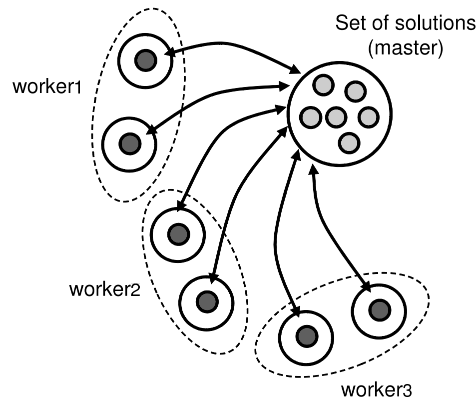


Figure 2.1: The parallel evaluation of solutions (iteration-level). The solutions are decomposed into different partitions which are evaluated in a parallel independent way.

2.1 Task Repartition for Metaheuristics on GPU

2.1.1 Model of Parallel Evaluation of Solutions

In the iteration-level model, the focus is on the parallelization of each iteration of metaheuristics. The iteration-level parallel model is mainly based on the distribution of the handled solutions. Indeed, the most time-consuming part in a metaheuristic is the evaluation of the generated solutions. The parallelization concerns search mechanisms that are problem-independent operations, such as the generation and evaluation of the neighborhood in S-metaheuristics and the evaluation of successive populations in P-metaheuristics. In other words, the iteration-level model is a low-level Master-Worker model that does not alter the behavior of the heuristic. Figure 2.1 gives an illustration of this model.
At each iteration, the master generates the set of solutions to be evaluated. Each worker receives from the master a partition of the solutions set. These solutions are evaluated and returned back to the master. For S-metaheuristics, the neighbors can also be generated by the workers. In this case, each worker receives a copy of the current solution, generates one or several neighbor(s) to be evaluated and returned back to the master. A challenge of this model is to determine the granularity of each partition of solutions to be allocated to each worker according to the communication delays of the given architecture. In terms of genericity, as the model is problem-independent, it is generic and reusable.

2.1.2 Parallelization Scheme on GPU

As quoted above, the evaluation of solution candidates is often the most time-consuming part of metaheuristics. Thereby, it must be done in parallel in regards with the iteration-level parallel model. Thereby, according to the Master-Worker paradigm, the idea is to

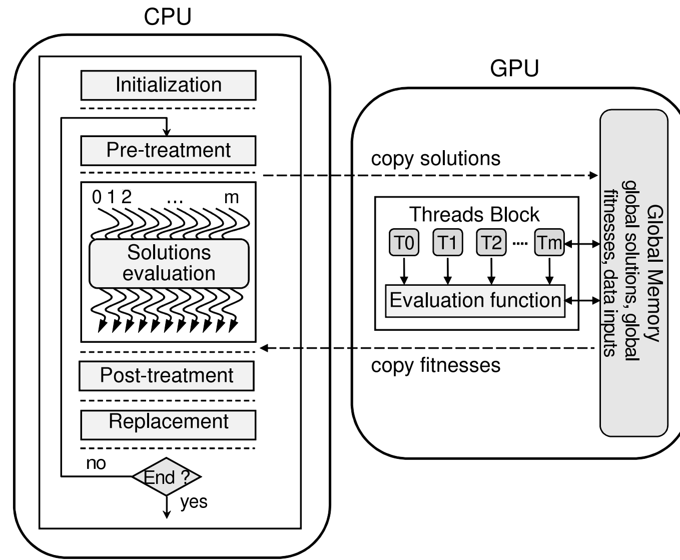


Figure 2.2: The parallel evaluation of solutions on GPU (iteration-level). The evaluation of solutions is performed on GPU and the CPU executes the sequential part of the search process.

evaluate the solutions in parallel on GPU.

To achieve this, the parallel iteration-level model has to be designed according to the data-parallel single program multiple data model of GPUs. As illustrated in Figure 2.2, the CPU-GPU task partitioning is such that the CPU hosts and executes the whole serial part of the handled metaheuristic. The GPU is in charge of the evaluation of the solutions set at each iteration. In this model, a function code called kernel is sent to the GPU to be executed by a large number of threads grouped into blocks. The granularity of each partition is determined by the number of threads per block.

This parallelization strategy has been widely used for P-metaheuristics on GPU especially for evolutionary algorithms due to their embarrassingly parallel workload (e.g in [WWF05, WW06, FWW07, Zhu09, ATD10, MBL$^+$09, SBPE10, Won09]).

2.2 Data Transfer Optimization

The GPU has its own memory and processing elements that are separate from the host computer. Thereby, data transfer between CPU and GPU through the PCIe bus might be a serious bottleneck in the performance of GPU applications. Table 2.1 gives an insight of the different transfer rates for the 4 different configurations presented in Chapter 1. This sample from the CUDA SDK [NVI11] delivers practical transfer rates by considering one data transfer of 30 MB. By repeating the process thousands of times, a higher amount of

Table 2.1: External bandwidth test. The program produces the elapsed time to copy 30 MB data and it delivers bi-directional transfer rates for 4 different configurations.

Configuration	CPU -> GPU		GPU -> CPU	
Core 2 Duo T5800 GeForce 8600M GT	1.76×10^{-2}s	1700 MB/s	6×10^{-2}s	500 MB/s
Core 2 Quad Q6600 GeForce 8800 GTX	1.66×10^{-2}s	1800 MB/s	2×10^{-2}s	1500 MB/s
Xeon E5450 GeForce GTX 280	1.25×10^{-2}s	2400 MB/s	1.36×10^{-2}s	2200 MB/s
Xeon E5620 Tesla M2050	0.81×10^{-2}s	3700 MB/s	1×10^{-2}s	3000 MB/s

data to be copied obviously has an influence on the execution time.

Therefore, one of the major issues is to optimize the data transfer between the CPU and the GPU. For metaheuristics, these copies are essentially 1) the solutions to be evaluated and 2) their resulting fitnesses.

Most P-metaheuristics are exploration oriented which implies that solutions at hand are usually uncorrelated. This does not occur in S-metaheuristics since each neighboring solution corresponds to a slight variation of the initial candidate solution. When it comes to parallelization, the optimization of data transfers is thus more prominent for S-metaheuristics. As a consequence, in the rest of the chapter, the focus will be exclusively on S-metaheuristics on GPU. We will emphasize how the optimization of data transfers is essential for the overall performance. Even if two tentatives of a tabu search algorithm on GPU have been proposed in [JJL08, ZCM08], they just operate on the multi-start execution of independent local search algorithms. In this purpose, we have contributed with the pionnering work on GPU-based S-metaheuristics in [1, 5, 9] for the parallel evaluation of neighborhoods.

2.2.1 Generation of the Neighborhood in S-metaheuristics

In deterministic S-metaheuristics (e.g. hill climbing, tabu search, variable neighborhood search), the generation and evaluation of the neighborhood can be done in parallel. Indeed, this step is the most computation intensive of a S-metaheuristic.

For optimizing the data transfers between the CPU and the GPU, one challenge is to say where the neighborhood in S-metaheuristics must be generated. For doing that, there are fundamentally two approaches:

- *Generation of the neighborhood on CPU and its evaluation on GPU.* At each iteration of the search process, the neighborhood is generated on the CPU side and its associated structure storing the solutions is copied on GPU. This approach is

the most straightforward since a thread is automatically associated with its physical neighbor representation. This is what it is usually done for the parallelization of P-metaheuristics on GPU. Thereby, the data transfers are essentially the set of neighboring solutions copied from the CPU to the GPU and the fitnesses structure which are copied from the GPU to the CPU.

- *Generation of the neighborhood and its evaluation on GPU.* In the second approach, the neighborhood is generated on GPU. This generation is performed in a dynamic manner which implies that no explicit structure needs to be allocated. This is achieved by considering a neighbor as a slight variation of the candidate solution which generates the neighborhood. Thereby, only the representation of this candidate solution must be copied from the CPU to the GPU. The advantage of such an approach is to reduce drastically the data transfers since the whole neighborhood does not have to be copied. The resulting fitnesses are the only structure which has to be copied back from the GPU to the CPU. However, such an approach raises another issue: a mapping between a thread and a neighbor must be determined. In some cases, it might be challenging. Such an issue will be discussed in Chapter 3.

Even if the first approach is easier, applying it on S-metaheuristics on GPU will end in a lot of data transfers for large neighborhoods. This is the case for P-metaheuristics on GPU since the entire population is usually copied from CPU to GPU. Such an approach will lead to a great loss of performance due to the limitation of the external bandwidth. That is the reason why, in the rest of the chapter, we will consider the second approach: the generation and the evaluation of the neighborhood on GPU. An experimental comparison of the two approaches is broached in Section 2.3.1.

2.2.2 The Proposed GPU-based Algorithm

Adapting traditional S-metaheuristics to GPU is not a straightforward task. We propose a methodology to rethink S-metaheuristics on GPU in a generic way (see Algorithm 6). First of all, at initialization stage, memory allocations on GPU are made: data inputs and candidate solution of the problem must be allocated (lines 4 and 5). As previously said, GPUs require massive computations with predictable memory accesses. Hence, a structure has to be allocated for storing the results of the evaluation of each neighbor (neighborhood fitnesses structure) at different addresses (line 6). Additional solution structures which are problem-dependent can also be allocated to facilitate the computation of neighbor evaluation (line 7). Second, problem data inputs, initial candidate solution and additional structures associated with this solution have to be copied onto the GPU (lines 8 to 10). It is important to notice that problem data inputs (e.g. a matrix in TSP) are a read-only

Algorithm 6 S-metaheuristic Template on GPU

1: Choose an initial solution
2: Evaluate the solution
3: Specific initializations
4: Allocate problem data inputs on GPU device memory
5: Allocate a solution on GPU device memory
6: Allocate a neighborhood fitnesses structure on GPU device memory
7: Allocate additional solution structures on GPU device memory
8: Copy problem data inputs on GPU device memory
9: Copy the solution on GPU device memory
10: Copy additional solution structures on GPU device memory
11: **repeat**
12:     **for** each neighbor in parallel on GPU **do**
13:         Evaluation of the candidate solution
14:         Insert the resulting fitness into the neighborhood fitnesses structure
15:     **end for**
16:     Copy the neighborhood fitnesses structure on CPU host memory
17:     Specific solution selection strategy on the neighborhood fitnesses structure
18:     Specific post-treatment
19:     Copy the chosen solution on GPU device memory
20:     Copy additional solution structures on GPU device memory
21: **until** a stopping criterion satisfied

structure and never change during all the execution of the S-metaheuristic. Therefore, their associated memory is copied only once during all the execution. Third, comes the parallel iteration-level, in which each neighboring solution is generated, evaluated and copied into the neighborhood fitnesses structure (from lines 12 to 15). Fourth, since the order in which candidate neighbors are evaluated is undefined, the neighborhood fitnesses structure has to be copied to the host CPU (line 16). Then, a specific solution selection strategy is applied to this structure (line 17): the exploration of the neighborhood fitnesses structure is carried out by the CPU in a sequential way. Finally, after a new candidate has been selected, this latter and its additional structures are copied to the GPU (lines 19 and 20). The process is repeated until a stopping criterion is satisfied.

This parallelization can be seen as an acceleration model which does not change the semantic of the S-metaheuristic. The iteration-level parallel model on GPU may be easily extended to variable neighborhood search (VNS) metaheuristics, in which the same parallel exploration is applied to the various neighborhoods associated with the problem. Its extension to iterative local search (ILS) metaheuristics is also straightforward as the parallelization on GPU could be used at each iteration of the ILS metaheuristic. The same goes on when dealing with hybrid genetic algorithms.

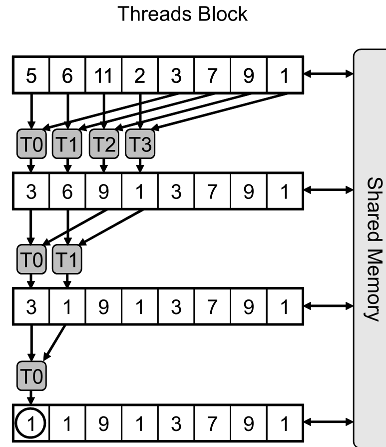


Figure 2.3: Reduction operation to find the minimum of each block. Local synchronizations are performed between threads of a same block via the shared memory.

2.2.3 Additional Data Transfer Optimization

In other S-metaheuristics such as hill climbing or variable neighborhood descent, the selection operates on the minimal/maximal fitness for finding the best solution. Therefore, only one value of the fitnesses structure may be merely copied from the GPU to the CPU. However, since read/write operations on memory are performed in an asynchronous manner, finding the appropriate minimal/maximal fitnesses is not straightforward. Indeed, traditional parallel techniques such as semaphores which imply the global synchronization (via atomic operations) of thousands of threads can drastically lead to diminished performance. To deal with this issue, adaptation of parallel reduction techniques for each thread block must be considered (see Fig. 2.3).

Algorithm 7 gives a template of the parallel reduction for a thread block (partition of the neighborhood). Basically, each thread loads one element from global to shared memory (lines 1 and 2). At each loop iteration, elements of the array are compared by pairs (lines 3 to 7). Then, by using local synchronizations between threads in a given block via the shared memory, one can find the minimum/maximum of a given array since threads operate at different memory addresses. For the sake of simplicity, the template is given for dealing with a neighborhood size which is a power of two, but adaptation of the template for the general case is straightforward. The complexity of such an algorithm is in $O(log_2(n))$ where n is the size of each thread block. If several iterations are performed on reduction kernels, the minimum of all the neighbors can be found. Thereby, the GPU reduction kernel makes it possible to get the minimum/maximum of each block of threads. More details of the method are given in [Har08]. The benefits of such a technique will be pointed out in Section 2.3.2.

Algorithm 7 Reduction kernel on the fitnesses structure.

```
Require: input_fitnesses;
 1: shared[thread_id] := input_fitnesses[id];
 2: local synchronization;
 3: for i := nbThreadsPerBlock/2 ; i > 0; i := i / 2 do
 4:    if thread_id < i then
 5:       shared[thread_id] := compare(shared[thread_id], shared[thread_id + i]);
 6:       local synchronization;
 7:    end if
 8: end for
 9: if thread_id = 0 then
10:    output_fitnesses[blockId] := shared[0];
11: end if
Ensure: output_fitnesses;
```

2.3 Performance Evaluation

2.3.1 Analysis of the Data Transfers from CPU to GPU

As previously said, one of the major issues in obtaining the best performance resides in optimizing the data transfer between the CPU and the GPU. To validate the performance of our algorithms, we propose to make an analysis of the time consumed by each major operation in two different approaches to assess the impact of such operations in terms of efficiency: 1) the generation of the neighborhood on CPU and its evaluation on GPU; 2) the generation of the neighborhood and its evaluation on GPU.

For the next experiments, a tabu search with 10000 iterations is considered on the GTX 280 configuration. The GPU adaptation of the tabu search is straightforward according to the proposed GPU algorithm (see Algorithm 6 in Section 2.2.2). First, the metaheuristic pre-treatment (line 3) is the tabu list initialization. Second, the replacement strategy (line 17) is performed by the best admissible neighbor according to its availability in the tabu list. Finally, the post-treatment (line 18) represents the tabu list update.

A single CPU core implementation and a CPU-GPU one are considered. The number of threads per block has been arbitrary chosen to 256 (multiple of 32), and the total number of threads created at run time is equal to the neighborhood size.

As an application, the permuted perceptron problem has been considered. Table 2.2 reports the time spent by each operation in the two approaches by using a neighborhood based on a Hamming distance of one (n neighbors). For the first approach, one can observe that the time spent by the data transfer is significant. It represents almost 25% of the total execution time for each instance. In the second approach, in comparison with the previous one, the time spent on the data transfer is drastically reduced with the instance

Table 2.2: Measures of the benefits of generating the neighborhood on GPU on the GTX 280. The permuted perceptron problem using a neighborhood based on a Hamming distance of one is considered.

Instance	CPU	**Evaluation on GPU**				**Gen. and eval. on GPU**			
		GPU	process	transfers	kernel	GPU	process	transfers	kernel
73-73	1.1	$3.4_{\times 0.3}$	4.0%	22.7%	73.3%	$3.0_{\times 0.4}$	1.0%	23.2%	75.8%
81-81	1.3	$3.8_{\times 0.3}$	5.3%	25.9%	68.8%	$3.3_{\times 0.4}$	1.1%	22.9%	75.9%
101-117	2.2	$5.1_{\times 0.4}$	4.8%	24.5%	70.7%	$4.2_{\times 0.5}$	1.1%	19.1%	79.8%
201-217	8.1	$11_{\times 0.7}$	6.8%	25.3%	67.9%	$7.7_{\times \mathbf{1.1}}$	1.2%	12.0%	86.8%
401-417	31	$27_{\times \mathbf{1.2}}$	7.1%	25.0%	67.9%	$14_{\times \mathbf{2.2}}$	1.1%	6.2%	92.7%
601-617	105	$68_{\times \mathbf{1.5}}$	7.1%	26.1%	66.8%	$43_{\times \mathbf{2.4}}$	1.0%	3.9%	95.1%
801-817	200	$98_{\times \mathbf{2.0}}$	7.1%	24.2%	68.7%	$50_{\times \mathbf{4.0}}$	0.6%	1.4%	98.0%
1001-1017	336	$106_{\times \mathbf{3.2}}$	5.5%	23.5%	71.0%	$58_{\times \mathbf{5.8}}$	0.3%	0.6%	99.1%
1301-1317	687	$146_{\times \mathbf{4.7}}$	5.2%	22.4%	72.4%	$85_{\times \mathbf{8.0}}$	0.2%	0.4%	99.4%

size increase. Indeed, for the instance $m = 73$ and $n = 73$, this time corresponds to 19% of the total running time and it reaches the value of 1% for the last instance ($m = 1301$ and $n = 1317$).

Another observation concerns the time taken by the generation and the evaluation of the neighborhood on GPU. Generally speaking, the algorithm in the second approach takes advantage of resource use since most of the total running time is dedicated to the GPU kernel execution. For example, in the fourth instance $m = 201$ and $n = 217$, the time associated with the evaluation of the neighborhood accounts for 86% of the total execution time. This time grows along with the instance size (more than 90% for the other larger instances).

As a result, the second approach outperforms the first one in terms of efficiency. Indeed, regarding the related acceleration factors for the two approaches, the reported results are in accordance with the previous observations. This difference of performance tends to grow with the instance size. In a general manner, the speed-up grows with the problem size augmentation (up to $\times 8$ for $m = 1301$, $n = 1317$). The acceleration factor for this implementation is significant but not spectacular. Indeed, since the neighborhood is relatively small (n threads), the number of threads per block is not enough to fully cover the memory access latency.

To validate this point, a neighborhood based on a Hamming distance of two has been implemented. Table 2.3 details the time spent by each operation in the two approaches by using a neighborhood based on a Hamming distance of two.

For the first approach, most of the time is devoted to data transfer. It accounts for nearly 75% of the execution time. As a consequence, such an approach is actually inefficient since the time spent on the data transfers dominates the whole algorithm. The produced measures of the speed-up confirm the previous observations. Indeed, since the amount of

Table 2.3: Measures of the benefits of generating the neighborhood on GPU on the GTX 280. The permuted perceptron problem using a neighborhood based on a Hamming distance of two is considered.

Instance	CPU	**Evaluation on GPU**				**Gen. and eval. on GPU**			
		GPU	process	transfers	kernel	*GPU*	process	transfers	kernel
73-73	2.1	$0.2_{\times\mathbf{3.2}}$	4.8%	73.7%	21.5%	$0.2_{\times\mathbf{10.9}}$	19.0%	11.2%	69.8%
81-81	2.7	$0.8_{\times\mathbf{3.3}}$	4.9%	74.6%	20.6%	$0.2_{\times\mathbf{12.2}}$	18.8%	10.7%	70.5%
101-117	7.0	$2.1_{\times\mathbf{3.3}}$	5.2%	74.1%	20.7%	$0.4_{\times\mathbf{18.1}}$	18.7%	10.1%	71.2%
201-217	48	$23_{\times\mathbf{2.1}}$	4.7%	74.3%	21.0%	$1.9_{\times\mathbf{25.3}}$	18.5%	7.3%	74.2%
401-417	403	$311_{\times\mathbf{1.3}}$	4.4%	75.6%	20.0%	$14_{\times\mathbf{28.8}}$	18.2%	6.3%	75.5%
601-617	2049	$3047_{\times 0.6}$	3.5%	75.8%	20.7%	$51_{\times\mathbf{40.1}}$	17.7%	4.5%	77.8%
801-817	5410	.	.	.	.	$128_{\times\mathbf{42.3}}$	13.3%	2.5%	84.2%
1001-1017	11075	.	.	.	.	$252_{\times\mathbf{43.9}}$	12.7%	1.5%	85.8%
1301-1317	25016	.	.	.	.	$568_{\times\mathbf{44.1}}$	10.9%	1.5%	87.6%

Table 2.4: Amount of data transfers at each iteration from the CPU to the GPU for the permuted perceptron problem.

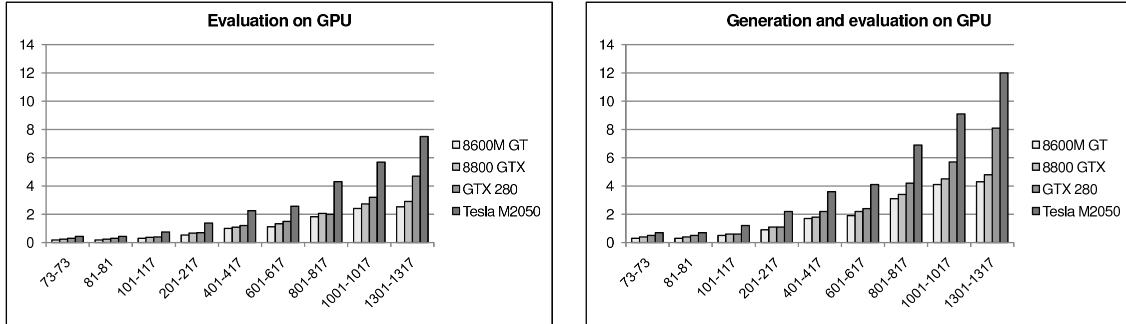
Instance	**Evaluation on GPU**	**Gen. and eval. on GPU**
	CPU -> GPU transfers	CPU -> GPU transfers
73-73	0.77 MB	0.58 KB
81-81	1.05 MB	0.65 KB
101-117	3.18 MB	0.94 KB
201-217	20.34 MB	1.74 KB
401-417	144.68 MB	3.34 KB
601-617	469.01 MB	4.94 KB
801-817	1089.35 MB	6.54 KB
1001-1017	2101.68 MB	8.14 KB
1301-1317	4565.18 MB	10.54 KB

data transferred tends to grow as the size increases, the acceleration factors diminish with the instance size (from ×3.3 to ×0.6). Furthermore, the algorithm could not be executed for larger instances since it exceeds the 1GB global memory of the GTX 280. Table 2.4 emphasizes these previous points when looking at the amount of data transfers.

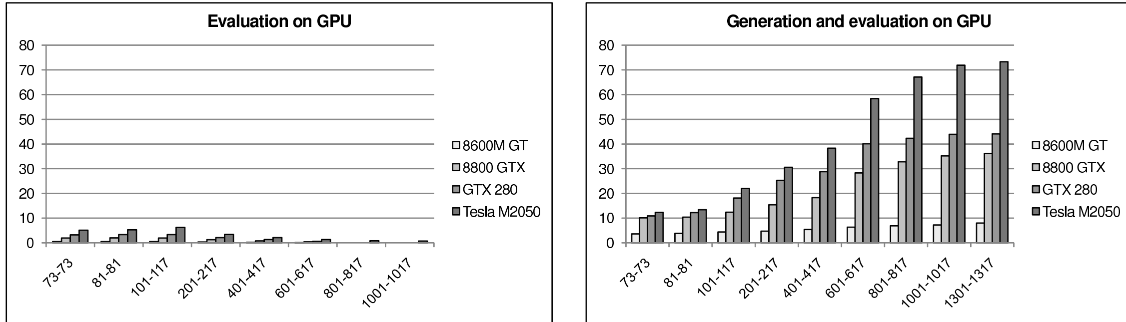
Table 2.5 reports the obtained speed-ups for the 4 configurations. Considering the generation and evaluation on GPU for a neighborhood based on a Hamming distance of two, for the first instance ($m = 73$, $n = 73$), acceleration factors are already significant (from ×3.6 to ×12.3). As long as the instance size increases, the acceleration factor grows accordingly (from ×3.6 to ×8 for the first configuration). Since a large number of cores are available on both 8800 and GTX 280, efficient speed-ups can be obtained (from ×10.1 to ×44.1). The application also scales well when performing on the Tesla Fermi card (speed-ups varying from ×11.7 to ×73.3). In comparison with the second approach, generating on CPU and evaluating on GPU is clearly inadequate in terms of performance. A conclusion to this

Table 2.5: Benefits of generating the neighborhood on GPU. The acceleration factors are reported for a tabu search on GPU on the permuted perceptron problem.

A neighborhood based on a Hamming distance of one is considered.



A neighborhood based on a Hamming distance of two is considered.



analysis highlights that the neighborhood should be always generated on GPU.

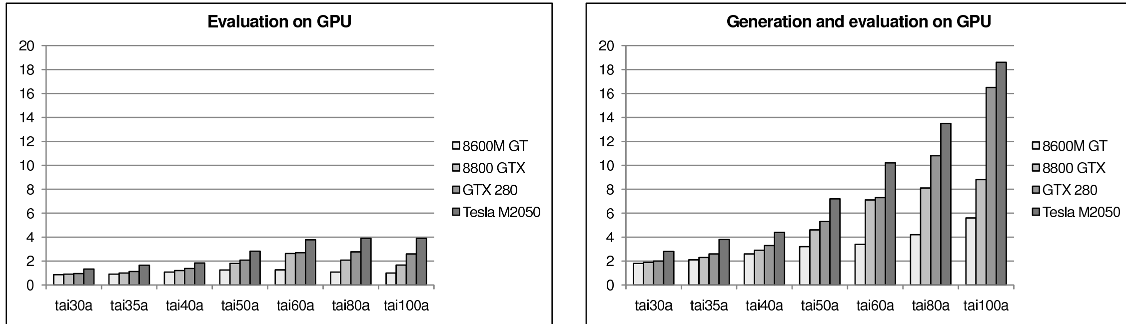
In a general manner, the same observations can be made for other optimization problems. The obtained speed-ups for the other problems are reported in Table 2.6 and in Table 2.7 for a tabu search using the same parameters.

The obtained performances strongly depends on the characteristics of each problem (see Table2.8). Data inputs such as matrices have a negative impact on the global performance. Indeed, non-coalescing memory drastically reduces the performance of GPU implementations. This is due to high-misaligned accesses to global memories when dealing with optimization problems. The time complexity represents the complexity of a neighbor evaluation (or Δ evaluation). In general, GPU applications get a better global performance when this evaluation is costly. The space complexity denotes the amount of data allocated for the evaluation of the entire neighborhood. As quoted above, such additional accesses via the global memory may lead to a global performance decrease.

To sum up, best GPU accelerations are obtained for problems, which are time-consuming and memory accesses free.

Table 2.6: Benefits of generating the neighborhood on GPU. The acceleration factors are reported for a tabu search on GPU on the quadratic assignement problem and the Weierstrass continuous function.

The quadratic assignment problem. A neighborhood based on a pair-wise exchange operator is considered.



The Weierstrass continuous function. 10000 neighbors are considered.

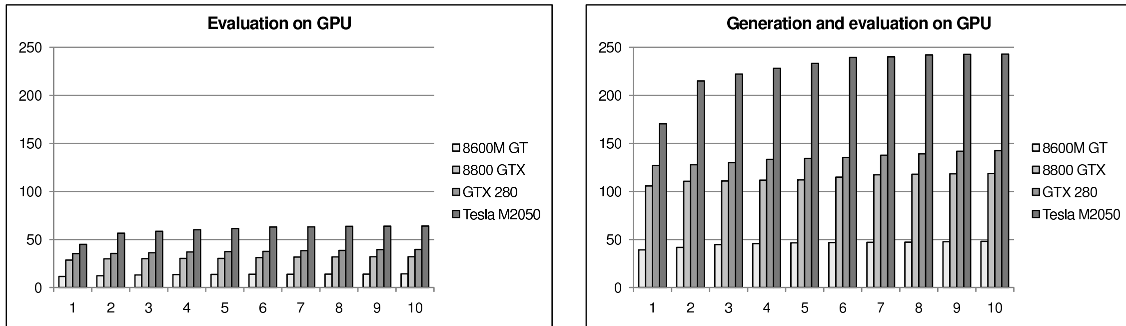
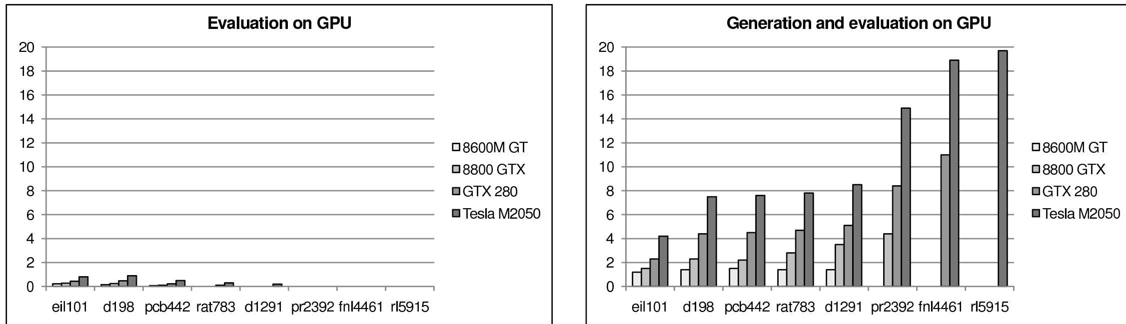


Table 2.7: Benefits of generating the neighborhood on GPU. The acceleration factors are reported for a tabu search on GPU on the traveling salesman problem and the Golomb Rulers.

The traveling salesman problem. A neighborhood based on a two-opt operator is considered.



The Golomb rulers. $n^3 - n$ neighborhors are considered.

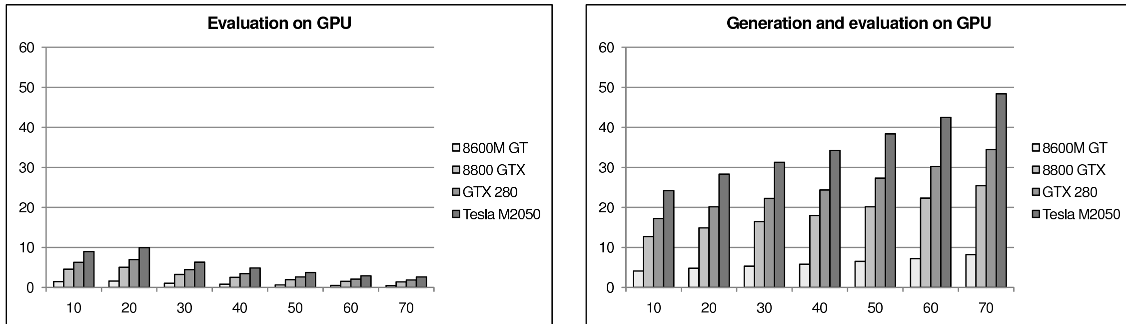


Table 2.8: Characteristics of the 5 optimization problems.

Problem	**Data inputs**	**Time complexity**	**Space complexity**
Permuted perceptron	One matrix	linear	linear
Quadratic assignment	Two matrices	constant / linear	quadratic
Weierstrass function	–	quadratic	–
Traveling salesman	One matrix	constant	–
Golomb rulers	–	quadratic	quadratic

Table 2.9: Measures of the benefits of using the reduction operation on the GTX 280. The permuted perceptron problem is considered for two different neighborhoods using an iterative local search composed of 100 hill climbing algorithms.

Instance	n neighbors			$\frac{n \times (n-1)}{2}$ neighbors		
	CPU	GPU	GPUR	CPU	GPU	GPUR
73-73	0.08	$0.22_{\times 0.4}$	$0.25_{\times 0.3}$	5.29	$0.42_{\times \mathbf{12.6}}$	$0.35_{\times \mathbf{15.1}}$
81-81	0.13	$0.29_{\times 0.4}$	$0.32_{\times 0.4}$	9.47	$0.65_{\times \mathbf{14.6}}$	$0.52_{\times \mathbf{18.2}}$
101-117	0.27	$0.42_{\times 0.6}$	$0.47_{\times 0.6}$	28.4	$1.2_{\times \mathbf{23.7}}$	$1.1_{\times \mathbf{25.9}}$
201-217	1.5	$1.4_{\times \mathbf{1.1}}$	$1.5_{\times 1.0}$	94.7	$3.1_{\times \mathbf{30.5}}$	$2.8_{\times \mathbf{33.8}}$
401-417	12.1	$5.4_{\times \mathbf{2.2}}$	$4.8_{\times \mathbf{2.5}}$	923	$27.3_{\times \mathbf{33.8}}$	$25_{\times \mathbf{36.9}}$
601-617	102	$32.1_{\times \mathbf{3.2}}$	$29.4_{\times \mathbf{3.5}}$	4754	$110_{\times \mathbf{43.2}}$	$103_{\times \mathbf{46.1}}$
801-817	199	$49.3_{\times \mathbf{4.0}}$	$45.7_{\times \mathbf{4.4}}$	13039	$270_{\times \mathbf{48.3}}$	$251_{\times \mathbf{51.9}}$
1001-1017	395	$67.4_{\times \mathbf{5.9}}$	$62.2_{\times \mathbf{6.3}}$	29041	$593_{\times \mathbf{48.9}}$	$551_{\times \mathbf{52.7}}$
1301-1317	1132	$141_{\times \mathbf{8.0}}$	$125_{\times \mathbf{9.0}}$	74902	$1512_{\times \mathbf{49.5}}$	$1395_{\times \mathbf{53.7}}$

2.3.2 Additional Data Transfer Optimization

Another point concerns the data transfer from the GPU to the CPU. Indeed, in some S-metaheuristics such as hill climbing, the selection of the best neighbor is operated by choosing the minimal/maximal fitness at each iteration. Hence, for these algorithms, there is no need to transfer the entire fitnesses structure, and further optimizations are possible. The following experiment consists in comparing two GPU-based approaches of the hill climbing algorithm.

In the first approach, the standard GPU-based algorithm is considered i.e. the fitnesses structure is copied back from the GPU to the CPU. In the second one, a reduction operation is iterated on GPU to find the minimum of all the fitnesses at each iteration.

Since the hill climbing heuristic rapidly converges, an iterated local search composed of 100 hill climbing algorithms has been considered. Such an important number of methods is in accordance with the previous running time for the tabu search.

Results for the permuted perceptron problem by considering two different neighborhoods are reported in Table 16. Regarding the version using a reduction operation (GPUR), significant improvements in comparison with the standard version (GPU) can be observed. For example, for the instance $m = 73$ and $n = 73$, in the case of $\frac{n \times (n-1)}{2}$ neighbors, the speed-up is equal to $\times 15.1$ for the version using reduction and $\times 12.6$ for the other one. Such improvement between 10% and 20% is maintained for most of the instances. A peek performance is reached with the instance $m = 1301$ and $n = 1317$ ($\times 53.7$ for GPUR against $\times 49.5$ for GPU).

An analysis on the average percentage of the time consumed by each operation can clarify this improvement. Table 2.10 highlights the analysis of the time dedicated to each major operation for a neighborhood based on a Hamming distance of two. On the one hand,

Table 2.10: Analysis of the time dedicated for each operation for an iterative local search composed of 100 hill climbing algorithms. The permuted perceptron problem using a Hamming distance of two and the reduction operation are considered.

Instance	GPU			GPUR		
	process	transfers	kernel	process	transfers	kernel
73-73	19.0%	11.2%	69.8%	1.43%	1.46%	97.11%
81-81	18.8%	10.7%	70.5%	0.91%	0.98%	98.01%
101-117	18.7%	10.1%	71.2%	0.46%	0.44%	99.10%
201-217	18.5%	7.3%	74.2%	0.36%	0.11%	99.53%
401-417	18.2%	6.3%	75.5%	0.08%	0.04%	99.88%
601-617	17.7%	4.5%	77.8%	0.04%	0.02%	99.94%
801-817	13.3%	2.5%	84.2%	0.03%	0.02%	99.96%
1001-1017	12.7%	1.5%	85.8%	0.02%	0.01%	99.97%
1301-1317	10.9%	1.5%	87.6%	0.01%	0.01%	99.98%

for the second approach, whatever the size of the neighborhood used or the instance size, the data transfer is nearly constant (varying between 0.01% and 1.46%). It can be explained by the fact that only one solution is transferred from the GPU to the CPU at each iteration. On the other hand, one can also notice that the time spent on the search process on CPU is also minimized for the second approach. Indeed, by definition, the reduction operation consists in finding the minimum which is performed on the GPU-side in a logarithmic time. While for the first approach, most of the CPU search process time corresponds to the search of the minimum in the fitnesses structure (linear time). Therefore, both minimization of the data transfers and complexity reduction can justify such an improvement of performance.

The same observations can be stated for the other problems where the reduction operator provides a 10% to 20% performance improvement (see Table 2.11).

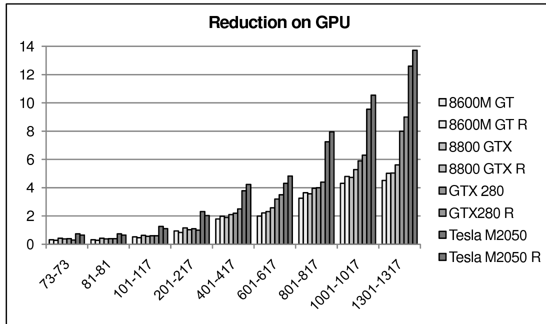
2.4 Comparison with Other Parallel and Distributed Architectures

During the last decade, cluster of workstations (COWs) and computational grids have been largely deployed to provide standard high-performance computing platforms. Hence, it will be interesting to compare the performance provided by GPU computing with such multi-level architectures in regards with S-metaheuristics. For the next experiments, we intend to compare each GPU configuration with COWs then with computational grids.

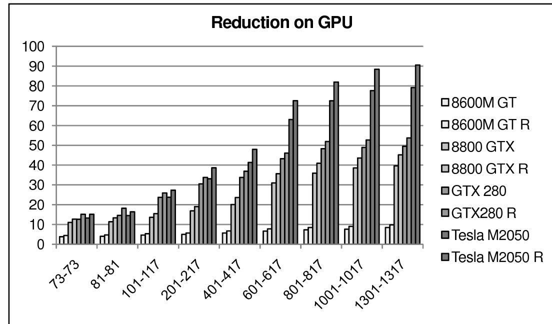
Table 2.11: Benefits of the reduction operator on GPU. The acceleration factors are reported for an iterative local search on different optimization problems.

The permuted perceptron problem.

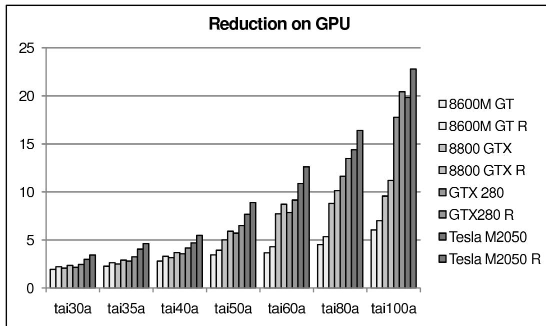
A neighborhood based on a Hamming distance of one is considered.



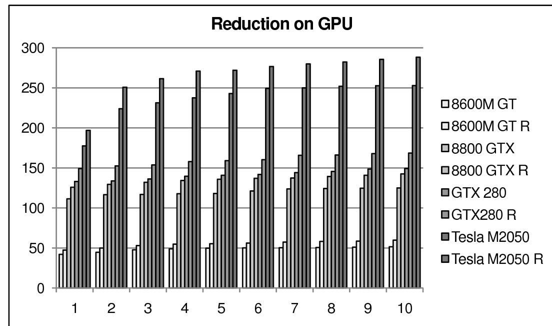
A neighborhood based on a Hamming distance of two is considered.



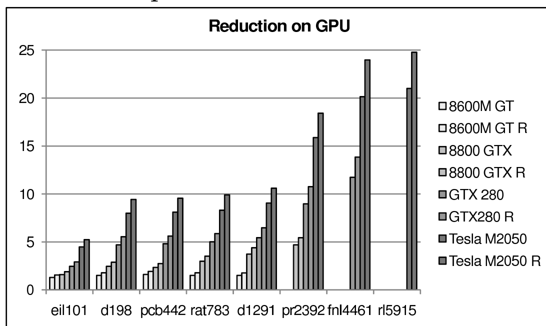
The quadratic assignment problem.
A neighborhood based on a pair-wise exchange operator is considered.



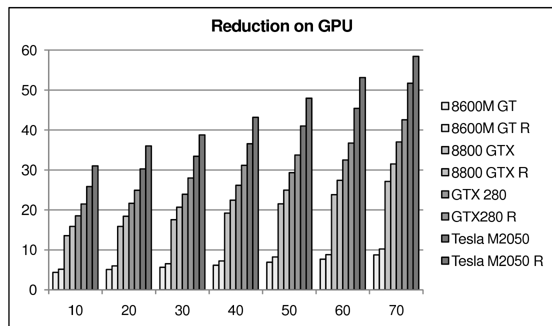
The Weierstrass continuous function.
10000 neighbors are considered.



The traveling salesman problem.
A neighborhood based on a two-opt operator is considered.



The Golomb rulers.
$n^3 - n$ neighbors are considered.



2.4.1 Parallelization Scheme on Parallel and Distributed Architectures

Algorithm 8 provides the template parallelization on top of these parallel and distributed architectures. Basically, the neighborhood is decomposed into separate partitions of equal size, which are distributed among the cores of the different machines (line 4). The only data which have to be copied concern the candidate solution and additional structures for its evaluation (lines 5 and 6). Each working node (CPU core) is in charge of the evaluation of its own neighborhood partition (lines 8 to 10). For deterministic S-metaheuristics, the parallelization is synchronous, and one has to wait for the termination of the exploration of all partitions (lines 11). Such a synchronous step ensures that the semantics of the S-metaheuristic are preserved. Then, the master handles the sequential part of the S-metaheuristic. The process is repeated until a stopping criterion is satisfied.

Algorithm 8 S-metaheuristic template on parallel and distributed architectures

```
 1: Choose an initial solution
 2: Evaluate the solution
 3: Specific initializations
 4: Define a partition size for each working node
 5: Copy the solution on each working node
 6: Copy additional solution structures on each working node
 7: repeat
 8:    for each working node in parallel do
 9:       Sequential evaluation of its own neighborhood partition
10:    end for
11:    Copy and gather each fitnesses partition on the master
12:    Specific solution selection strategy on the neighborhood fitnesses structure
13:    Specific post-treatment
14:    Copy the chosen solution on each working node
15:    Copy additional solution structures each working node
16: until a stopping criterion satisfied
```

In comparison with the GPU parallelization, the iteration-level model on parallel and distributed architectures is more flexible. Indeed, due to the asynchronous nature of these architectures, the copy and the exploration of all partitions might also be done in an asynchronous manner. Thereby, it ensures to deal with S-metaheuristics which explore a partial neighborhood; or few irregular problems in which the execution time varies during the search process.

2.4.2 Configurations

For doing a fair comparison with the previous results on GPU, the different parallel and distributed architectures must have the same computational power. Table 2.12 presents

Table 2.12: Parallel and distributed machines used for the experiments on COWs and Grid'5000.

Architecture	**Configuration 1** Machines	GFLOPS	**Configuration 2** Machines	GFLOPS
GPU	Core 2 Duo T5800 GeForce 8600M GT	76.8	Core 2 Quad Q6600 GeForce 8800 GTX	384
COWs	Intel Xeon E5440 8 CPU cores	90.656	4 Intel Xeon E5440 32 CPU cores	362.624
Grid	2 Intel Xeon E5440 2 × 4 CPU cores	90.656	Amd Opteron 2218 Intel Xeon E5520 2 Intel Xeon E5420 Intel Xeon E5440 40 CPU cores	406.368

Architecture	**Configuration 3** Machines	GFLOPS	**Configuration 4** Machines	GFLOPS
GPU	Intel Xeon E5450 GeForce GTX 280	981.12	Intel Xeon E5620 Tesla M2050	1106.08
COWs	11 Intel Xeon E5440 88 CPU cores	995.236	13 Intel Xeon E5440 104 CPU cores	1176.188
Grid	2 Intel Xeon E5520 2 AMD Opteron 2218 2 Intel Xeon E5520 4 Intel Xeon E5520 Intel Xeon X5570 Intel Xeon E5520 96 CPU cores	979.104	4 Intel Xeon E5520 2 AMD Opteron 2218 2 Intel Xeon E5520 4 Intel Xeon E5520 Intel Xeon X5570 Intel Xeon E5520 112 CPU cores	1160.056

the different machines used for the experiments. The number of potential GFLOPS is calculated from the theoretical ones provided by constructors.

The different machines used for the experiments for COWs and grid are described in Table 2.12. Most of them are octo-core workstations. The different computers have been chosen accordingly to the different GPU configurations i.e. in agreement with their computational power. Such a metric has been deduced from the potential GFLOPS delivered by the different machines.

From an implementation point of view, a hybrid OpenMP/MPI version has been produced to take advantage of both multi-core and distributed environments. Such a hybrid implementation has widely proved in the past its efficiency for multi-level architectures [JJMH03]. The tabu search previously seen has been implemented on top of these architectures.

The permuted perceptron problem using a neighborhood based on a Hamming distance of two is considered on the two architectures. A Myri-10G gigabit ethernet connects the different machines of the COWs. For the workstations distributed in a grid organiza-

Table 2.13: Measures in terms of efficiency for a cluster of workstations. The permuted perceptron problem using a neighborhood based on a Hamming distance of two is considered.

Instance	Intel Xeon E5440 8 CPU cores		4 Intel Xeon E5440 32 CPU cores	
	GPU	COW	GPU	COW
73-73	0.8 $_{\times 3.6}$	0.9 $_{\times 3.5}$	0.2 $_{\times 10.1}$	1.4 $_{\times 1.6}$
81-81	1.1 $_{\times 3.8}$	1.2 $_{\times 3.6}$	0.3 $_{\times 10.4}$	1.6 $_{\times 1.8}$
101-117	2.5 $_{\times 4.4}$	2.9 $_{\times 3.8}$	0.6 $_{\times 12.4}$	1.9 $_{\times 3.8}$
201-217	15 $_{\times 4.7}$	18 $_{\times 3.9}$	3.3 $_{\times 15.4}$	6.2 $_{\times 8.2}$
401-417	103 $_{\times 5.4}$	139 $_{\times 4.1}$	24 $_{\times 18.3}$	39 $_{\times 11.3}$
601-617	512 $_{\times 6.3}$	966 $_{\times 3.3}$	89 $_{\times 28.3}$	258 $_{\times 9.8}$
801-817	1245 $_{\times 6.9}$	2828 $_{\times 3.0}$	212 $_{\times 32.8}$	737 $_{\times 9.4}$
1001-1017	2421 $_{\times 7.2}$	6307 $_{\times 2.8}$	409 $_{\times 35.2}$	1708 $_{\times 8.5}$
1301-1317	4903 $_{\times 8.0}$	15257 $_{\times 2.6}$	911 $_{\times 36.2}$	3968 $_{\times 8.4}$

Instance	11 Intel Xeon E5440 88 CPU cores		13 Intel Xeon E5440 104 CPU cores	
	GPU	COW	GPU	COW
73-73	0.2 $_{\times 10.9}$	5.4 $_{\times 0.4}$	0.2 $_{\times 12.3}$	5.9 $_{\times 0.4}$
81-81	0.2 $_{\times 12.2}$	5.6 $_{\times 0.5}$	0.2 $_{\times 13.4}$	6.3 $_{\times 0.4}$
101-117	0.4 $_{\times 18.1}$	6.0 $_{\times 1.2}$	0.3 $_{\times 22.0}$	6.7 $_{\times 1.1}$
201-217	1.9 $_{\times 25.3}$	7.6 $_{\times 6.3}$	1.6 $_{\times 30.6}$	7.0 $_{\times 6.8}$
401-417	14 $_{\times 28.8}$	21 $_{\times 19.2}$	10 $_{\times 38.3}$	19 $_{\times 21.2}$
601-617	51 $_{\times 40.1}$	115 $_{\times 17.8}$	35 $_{\times 58.4}$	108 $_{\times 19.0}$
801-817	128 $_{\times 42.3}$	322 $_{\times 16.8}$	81 $_{\times 67.1}$	311 $_{\times 17.4}$
1001-1017	252 $_{\times 43.9}$	793 $_{\times 14.0}$	154 $_{\times 71.9}$	778 $_{\times 14.3}$
1301-1317	568 $_{\times 44.1}$	1807 $_{\times 13.8}$	342 $_{\times 73.3}$	1789 $_{\times 14.0}$

tion, experiments have been carried out on the high-performance computing Grid'5000[1] [BCC+06] involving two, five and seven French sites. The acceleration factors are established from each single CPU core used for the previous experiments.

2.4.3 Cluster of Workstations

Table 2.13 presents the produced results for this architecture. Whatever the used configuration, the acceleration factors keep growing up until reaching a particular instance, then they immediately decrease with the instance size. For example, for the second configuration, the acceleration factors begin from $\times 1.6$ until reaching a peak value of $\times 11.3$ for the instance $m = 401$ and $n = 417$. After, the speed-ups start decreasing until reaching the value $\times 8.4$. This behaviour can be elucidated by the following reason: a performance improvement can be made as long as the part reserved to the partitions evaluation (worker) is not too much dominated by the communication time. An analysis of the time spent to transfers including synchronizations confirms this fact (see Table 2.14). The percentage dedicated to the transfer operations varies accordingly with the speed-ups observed.

[1] http://grid5000.fr

Table 2.14: Analysis of the time dedicated to each operation for a cluster of workstations. The permuted perceptron problem using a neighborhood based on a Hamming distance of two is considered.

Instance	Intel Xeon E5440 8 CPU cores			4 Intel Xeon E5440 32 CPU cores		
	process	transfers	workers	process	transfers	workers
73-73	4.2%	45.7%	50.1%	1.8%	93.0%	5.2%
81-81	3.3%	45.3%	51.4%	2.3%	91.9%	5.8%
101-117	2.6%	43.1%	54.3%	3.9%	83.8%	12.3%
201-217	1.9%	42.4%	55.7%	5.7%	67.8%	26.5%
401-417	1.8%	39.6%	58.6%	6.5%	57.1%	36.4%
601-617	0.9%	52.0%	47.1%	3.5%	64.9%	31.6%
801-817	0.6%	56.5%	42.9%	2.3%	67.4%	30.3%
1001-1017	0.5%	59.4%	40.1%	1.9%	70.7%	27.4%
1301-1317	0.4%	62.5%	37.1%	1.6%	71.3%	27.1%
Instance	**11 Intel Xeon E5440 88 CPU cores**			**13 Intel Xeon E5440 104 CPU cores**		
	process	transfers	workers	process	transfers	workers
73-73	0.7%	98.8%	0.5%	0.7%	98.9%	0.4%
81-81	0.7%	98.7%	0.6%	0.7%	98.8%	0.5%
101-117	1.2%	97.4%	1.4%	1.1%	97.6%	1.3%
201-217	4.6%	88.2%	7.2%	4.5%	88.5%	7.0%
401-417	12.1%	63.8%	24.1%	11.9%	64.0%	24.1%
601-617	7.8%	71.7%	20.5%	7.9%	71.9%	20.6%
801-817	5.3%	75.4%	19.3%	5.4%	75.7%	19.5%
1001-1017	4.0%	79.9%	16.1%	3.9%	81.2%	15.9%
1301-1317	3.5%	80.6%	15.9%	3.4%	80.8%	15.8%

Furthermore, increasing the number of machines (i.e. the number of communications) has a negative impact on the performance for small instances such as $m = 73$ and $n = 73$. Indeed, the associated time dedicated to the transfers clearly dominates the algorithm (93% and 98% for the second and the third configurations). Such a behaviour does not appear as well in the first configuration since communication is based only on an inter-core communication.

Regarding the overall performance, whatever the instance size, acceleration factors are less salient than their GPU counterparts. For COWs, these acceleration factors diversify from $\times 0.4$ to $\times 21.2$ whereas for GPUs they alternate from $\times 3.6$ to $\times 73.3$.

2.4.4 Workstations in a Grid Organization

All the previous observations made for COWs are valid when dealing with workstations distributed in a grid organization. In general, the overall performance is less significant than COWs for a comparable computational horsepower. Indeed, the acceleration factors vary from $\times 0.3$ to $\times 16.1$ (see Table 2.15). This performance diminution is explained by the growth of the communication time since clusters are distributed among different sites.

Table 2.15: Measures in terms of efficiency for workstations distributed in a grid organization. The permuted perceptron problem using a neighborhood based on a Hamming distance of two is considered.

Instance	2 machines 8 CPU cores		5 machines 40 CPU cores	
	GPU	Grid	GPU	Grid
73-73	0.8 $\times$**3.6**	1.1 $\times$**2.6**	0.2 $\times$**10.1**	1.8 $\times$**1.2**
81-81	1.1 $\times$**3.8**	1.4 $\times$**3.0**	0.3 $\times$**10.4**	2.0 $\times$**1.4**
101-117	2.5 $\times$**4.4**	3.5 $\times$**3.1**	0.6 $\times$**12.4**	2.4 $\times$**3.0**
201-217	15 $\times$**4.7**	22 $\times$**3.2**	3.3 $\times$**15.4**	7.8 $\times$**6.5**
401-417	103 $\times$**5.4**	167 $\times$**3.4**	24 $\times$**18.3**	49 $\times$**9.0**
601-617	512 $\times$**6.3**	1159 $\times$**2.8**	89 $\times$**28.3**	323 $\times$**7.8**
801-817	1245 $\times$**6.9**	3394 $\times$**2.5**	212 $\times$**32.8**	922 $\times$**7.5**
1001-1017	2421 $\times$**7.2**	7568 $\times$**2.3**	409 $\times$**35.2**	2135 $\times$**6.8**
1301-1317	4903 $\times$**8.0**	18308 $\times$**2.2**	911 $\times$**36.2**	4960 $\times$**6.7**

Instance	12 machines 96 CPU cores		14 machines 112 CPU cores	
	GPU	Grid	GPU	Grid
73-73	0.2 $\times$**10.9**	7.3 $\times$0.3	0.2 $\times$**12.3**	7.9 $\times$0.3
81-81	0.2 $\times$**12.2**	7.6 $\times$0.4	0.2 $\times$**13.4**	8.3 $\times$0.4
101-117	0.4 $\times$**18.1**	8.1 $\times$0.9	0.3 $\times$**22.0**	8.8 $\times$0.8
201-217	1.9 $\times$**25.3**	10 $\times$**4.7**	1.6 $\times$**30.6**	9.5 $\times$**4.9**
401-417	14 $\times$**28.8**	28 $\times$**14.4**	10 $\times$**38.3**	25 $\times$**16.1**
601-617	51 $\times$**40.1**	155 $\times$**13.2**	35 $\times$**58.4**	148 $\times$**13.8**
801-817	128 $\times$**42.3**	425 $\times$**12.7**	81 $\times$**67.1**	411 $\times$**13.1**
1001-1017	252 $\times$**43.9**	1071 $\times$**10.3**	154 $\times$**71.9**	1043 $\times$**10.6**
1301-1317	568 $\times$**44.1**	2439 $\times$**10.2**	342 $\times$**73.3**	2405 $\times$**10.3**

An analysis of the time dedicated to transfers in Table 2.16 confirms this observation. In comparison with COWs, the transfer time corresponding to the partitions sending and the synchronization is significantly more prominent whatever the instance size is. This can be explained by the distribution of computers among the different sites (respectively two, five and seven according to the configuration). Indeed, in COWs, such an extra inter-sites communication does not occur since the computers are directly linked by a gigabit ethernet.

.

Table 2.16: Analysis of the time dedicated to each operation for workstations distributed in a grid organization. The permuted perceptron problem using a neighborhood based on a Hamming distance of two is considered.

Instance	**Configuration 2 machines 8 CPU cores**			**Configuration 5 machines 40 CPU cores**		
	process	transfers	workers	process	transfers	workers
73-73	3.4%	59.5%	37.1%	2.0%	94.1%	3.9%
81-81	2.6%	54.5%	42.9%	2.0%	93.5%	4.5%
101-117	2.1%	53.6%	44.3%	3.1%	87.2%	9.7%
201-217	1.6%	52.7%	45.7%	4.5%	74.5%	21.0%
401-417	1.5%	49.9%	48.6%	5.2%	65.8%	29.0%
601-617	0.8%	59.2%	40.0%	2.8%	72.0%	25.2%
801-817	0.5%	63.8%	35.7%	1.8%	74.1%	24.1%
1001-1017	0.4%	66.7%	32.9%	1.5%	76.6%	21.9%
1301-1317	0.3%	68.3%	31.4%	1.3%	77.1%	21.6%
Instance	**Configuration 12 machines 96 CPU cores**			**Configuration 14 machines 112 CPU cores**		
	process	transfers	workers	process	transfers	workers
73-73	0.5%	99.1%	0.4%	0.4%	99.3%	0.3%
81-81	0.6%	98.9%	0.5%	0.6%	99.0%	0.4%
101-117	0.8%	98.2%	1.0%	0.7%	98.5%	0.8%
201-217	3.5%	91.2%	5.3%	3.4%	91.4%	5.2%
401-417	10.4%	70.5%	19.1%	10.2%	70.9%	18.9%
601-617	6.4%	77.1%	16.5%	6.3%	77.3%	16.4%
801-817	4.1%	80.2%	15.7%	3.9%	80.6%	15.5%
1001-1017	3.0%	83.3%	13.7%	2.9%	83.5%	13.6%
1301-1317	2.6%	86.4%	11.0%	2.4%	86.9%	10.7%

Conclusion

In this chapter, we have proposed an efficient cooperation between the CPU and the GPU. This challenge represents one of the critical issues when dealing with the parallel evaluation of solutions on GPU (iteration-level model). Indeed, since the evaluation of solutions is often the time-consuming part of metaheuristics, it has to be done in parallel on GPU.

- **Optimization of data transfers from CPU to GPU**. One of the crucial issues is to minimize the data transfer between the CPU and the GPU. For S-metaheuristics, the generation of the neighborhood constitutes a must to achieve the best performance. In this purpose, we have proposed an efficient algorithm which performs the generation and evaluation of the neighborhood in parallel on GPU, while optimizing the associated data transfers. In Chapter 3, we will show how to accomplish this generation on the GPU side.

- **Additional optimization from GPU to CPU**. In addition, when dealing with S-metaheuristics which operates on minimal/maximum fitness, a reduction operator can be performed to reduce the data transfers from GPU to CPU. As a result, when doing such a mechanism, a further performance improvement can be obtained.

- **Comparison with COWs and grids**. For a same computational power, implementations on GPU architectures are much more efficient than COWs and grids for dealing with data-parallel regular applications. Indeed, the main issue in such distributed architectures concern the communication cost. This is also due to the synchronous nature of the parallel iteration-level model (tabu search). However, since GPUs execute threads in a SIMD fashion, they could not be adapted for few irregular problems (e.g. [MCT06]), in which the computations become asynchronous.

Chapter 3

Efficient Parallelism Control

In this chapter, the focus is on the efficient control of parallelism for the iteration-level on GPU. Indeed, GPU computing is based on hyper-threading, and the order in which the threads are executed is unknown.
First, an efficient thread control must be applied to meet the memory constraints. It allows to add some robustness in the developed metaheuritics on GPU, and to improve the overall performance. Second, regarding S-metaheuristics on GPU, an efficient mapping has to be defined between each neighboring candidate solution and a thread designated by a unique identifier. Then, we will examine the design on GPU of S-metaheuristics which explore a partial neighborhood. We will show why they are not well-adapted to GPU architectures. Finally, having in hand new tools to design new S-metaheuristics, we will assess the impact on how the increase of the neighborhood size can improve the quality of the obtained solutions.

Contents

3.1 Thread Control for Metaheuristics on GPU 61
3.1.1 Execution Parameters at Runtime 61
3.1.2 Thread Control Heuristic 62
3.2 Efficient Mapping of Neighborhood Structures on GPU 64
3.2.1 Binary Encoding 64
3.2.2 Discrete Vector Representation 65
3.2.3 Vector of Real Values 65
3.2.4 Permutation Representation 66
3.3 First Improvement S-metaheuristics on GPU 69
3.4 Performance Evaluation 71
3.4.1 Thread Control for Preventing Crashes 71
3.4.2 Thread Control for Further Optimization 74
3.4.3 Performance of User-defined Mappings 74
3.4.4 First Improvement S-metaheuristics on GPU 77
3.5 Large Neighborhoods for Improving Solutions Quality 80
3.5.1 Application to the Permuted Perceptron Problem 81

Main publications related to this chapter

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU Computing for Local Search Metaheuristic Algorithms. *IEEE Transactions on Computers*, in press, 2011.

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Neighborhood Structures for GPU-based Local Search Algorithms. *Parallel Processing Letters*, 20(4):307–324, 2010.

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Large Neighborhood Local Search Optimization on Graphics Processing Units. *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, pages 1–8, Workshop Proceedings, IEEE, 2010.

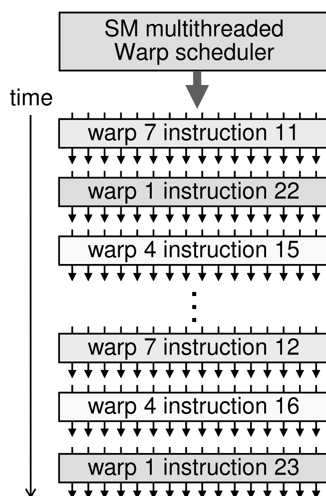


Figure 3.1: Illustration of the operation of the warp-based thread scheduling scheme.

3.1 Thread Control for Metaheuristics on GPU

3.1.1 Execution Parameters at Runtime

GPU multiprocessor is based on thread-level parallelism to maximize the exploitation of its functional units. The multiprocessor executes threads by groups of 32 threads called *warps*. Thread blocks are partitioned into warps that are organized by a scheduler at runtime (see Figure 3.1). At each instruction issue time, the scheduler selects a warp that is ready to execute and sets the next instruction to the active threads of the warp. Full utilization is achieved when the scheduler always has some instructions to process at every clock cycle. In other words, best performance is reached when the latency of each warp is completely hidden by other warps [ND10, OML+08].

Hence, one of the key points to obtain high performance is to keep the GPU multiprocessors as busy as possible. Latency hiding depends on the number of active warps per multiprocessor, which is implicitly determined by the execution parameters along with register constraints. That is the reason why, it is necessary to use threads and blocks in a way that maximizes hardware utilization. To achieve this, a GPU application can be tuned by two leading parameters: the number of threads per block and the total number of threads. There are many different factors involved in selecting these parameters and some experimentation are inevitably required.

In the approach presented in the previous chapter, the execution of a metaheuristic on GPU consists in launching a kernel with a large number of threads where one thread is associated with one solution. In general, threads per block should be a multiple of the warp size (i.e. 32 threads) to avoid wasting computation on under-populated warps. Thereby,

good performances for applications such as metaheuristics are usually reached for 64, 128, 256 and 512 threads per block.

However, for a very large solutions set, some experiments might not be conducted. The major issue is then to control the number of threads to meet the memory constraints like the limited size and number of registers to be allocated to each thread. Unlike the previous approach, one thread might not be associated with one neighbor but several neighbors. As a result, on the one hand, having an efficient thread control will prevent GPU programs for crashing. On the other hand, it will allow to find an optimal number of threads required at runtime to get the best multiprocessor occupancy, leading to a better performance.

3.1.2 Thread Control Heuristic

Different works [CSV10, NM09] have been investigated for parameters auto-tuning. The heuristics are *a priori* approaches which are based on enumerating all the different values of the two parameters (threads per block and the total number of threads). However, such approaches are too much time-consuming and may be not well-adapted for metaheuristics due to their *a priori* natures. For dealing with this issue, we have proposed in [1] a dynamic heuristic for parameters auto-tuning at runtime. To the best of our knowledge, such approach has never been investigated regarding the different works on GPU-based metaheuristics. Algorithm 9 gives the general template for this heuristic. Such a method is common to all metaheuristics on GPU (i.e. P-metaheuristics and S-metaheuristics).

The main idea of this approach is to send threads by "waves" to the GPU kernel to perform the parameters tuning during the first metaheuristic iterations. Thereby, the time measurement for each selected configuration according to a certain number of trials (lines 5 to 14) will yield the best configuration parameters. Regarding the number of threads per block, as quoted above, it is set to a multiple of the warp size (see line 19). The starting total number of threads is set as the nearest power of two of the solution size to be in accordance with the previous point. For decreasing the total number of configurations, the algorithm terminates when the logarithm of the neighborhood size is reached. In some cases, when a thread block allocates more registers than are available on a multiprocessor, the kernel execution will fail since too many threads are requested. Therefore, a fault-tolerance mechanism is provided to detect such a situation (from lines 8 to 12). In this case, the heuristic terminates and returns the best configuration parameters previously found.

The only parameter to determine is the number of trials per configuration. The more this value is, the more will be accurate the final tuning at the expense of an extra computational time. The benefits of the thread control will be presented in Section 3.4.1 and 3.4.2.

Algorithm 9 Dynamic parameters tuning heuristic

```
Require: nb_trials;
 1: nb_threads := nearest_power_of_2 (solution_size);
 2: while nb_threads <= neighborhood_size do
 3:   nb_threads_block := 32;
 4:   while nb_threads_block <= 512 do
 5:     repeat
 6:       Metaheuristic iteration pre-treatment on host side
 7:       Evaluation kernel on GPU
 8:       if GPU kernel failure then
 9:         Restore (best_nb_threads);
10:         Restore (best_nb_threads_block);
11:         Exit procedure
12:       end if
13:       Metaheuristic iteration post-treatment on host side
14:     until Time measurements of nb_trials
15:     if Best time improvement then
16:       best_nb_threads := nb_threads;
17:       best_nb_threads_block := nb_threads_block;
18:     end if
19:     nb_threads_block := nb_threads + 32;
20:   end while
21:   nb_threads := nb_threads * 2;
22: end while
23: Exit procedure
Ensure: best_nb_threads and best_nb_threads_block;
```

3.2 Efficient Mapping of Neighborhood Structures on GPU

In Chapter 2, we have defined a parallelization scheme of S-metaheuristics on GPU. The key success is based on the generation and evaluation of the neighborhood on GPU to reduce the data transfers and thus achieve the best performance. In this section, the focus is on the neighborhood generation to control the threads parallelism. This generation step of the iteration-level parallel model is not generic and must be handled efficiently.
The neighborhood structures play a crucial role in the performance of S-metaheuristics and are problem-dependent. For a GPU application, a kernel is launched with a large number of threads which are provided with a unique *id*. Regarding the generation and evaluation kernel on GPU (see Algorithm 10), according to the thread control, it just consists in associating one thread to many neighbors (also called kernel persistence). This association is legitimate in S-metaheuristics since each solution evaluation is independent.

Algorithm 10 Generation and evaluation kernel on GPU

```
Require: offset := nb_threads;
1: id := get_thread_id();
2: while id < neighborhood_size do
3:    neighbor := mapping (id,solution);
4:    fitness[id] := evaluate (neighbor);
5:    id := id + offset;
6: end while
```

The main difficulty which remains, is to find an efficient mapping between a GPU thread and neighbor candidate solutions. In other words, the issue is to say which solution must be handled by which thread. The answer is dependent of the solution representation. Indeed, the neighborhood structure strongly depends on the target optimization problem representation. In the following, we provide a methodology to deal with the main structures of the literature.

3.2.1 Binary Encoding

In a binary representation, a solution is coded as a vector (string) of bits. The neighborhood representation for binary problems is based on Hamming distance (see Fig. 3.2). A neighbor of a given solution is obtained by flipping one bit of the solution (for a Hamming distance of one).
Mapping between LS neighborhood encoding and GPU threads is rather trivial. Indeed, on the one hand, for a binary vector of size n, the size of the neighborhood is exactly n. On the other hand, threads are provided with a unique *id*. That way, a thread is directly associated with at least one neighbor. Hence, such a mapping is straightforward. This case

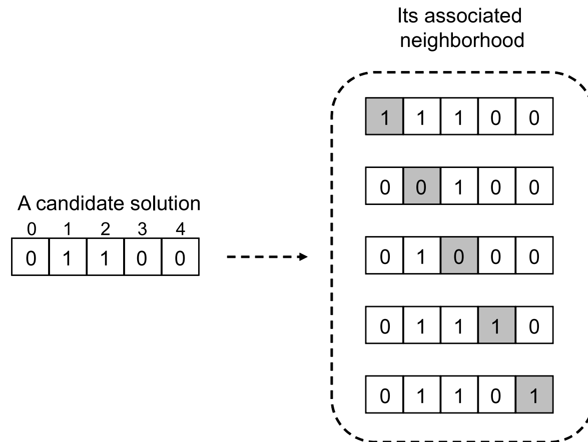


Figure 3.2: Binary representation. For a Hamming distance of one, the neighborhood of a solution consists in flipping one bit of the solution.

has been investigated for the permuted perceptron problem [Poi95] with a neighborhood based on Hamming distance of one.

3.2.2 Discrete Vector Representation

Discrete vector representation is an extension of binary encoding using a given alphabet Σ. In this representation, each variable acquires its value from the alphabet Σ. Assuming that the cardinality of the alphabet Σ is k, the size of the neighborhood is $(k-1) \times n$ for a discrete vector of size n. Fig. 3.3 illustrates an example of discrete representation with $n = 3$ and $k = 5$.
Let id be the identity of the thread corresponding to a given candidate solution of the neighborhood. Compared to the initial solution which allowed to generate the neighborhood, $id/(k-1)$ represents the position which differs from the initial solution and $id\%(k-1)$ is the available value from the ordered alphabet Σ (both using zero-index based numbering). Therefore, such a mapping is possible. An application of this neighborhood has been investigated for the Golomb rulers [GB77].

3.2.3 Vector of Real Values

For continuous optimization, a solution is coded as a vector of real values. A usual neighborhood for such a representation consists in discretizing the solution space. The neighborhood is defined in [CS00] by using the concept of "ball". A ball $B(s, r)$ is centered on s with radius r; it contains all points s' such that $||s' - s|| \leq r$. To obtain a homogeneous exploration of the space, a set of balls centered on the current solution s is considered with radius h_0, h_1, $\ldots$, h_m.

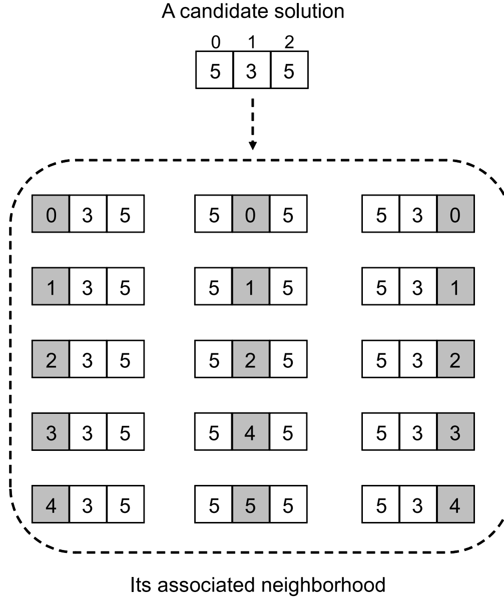


Figure 3.3: Discrete vector representation. The neighborhood of a solution consists in replacing the discrete value of a vector element by any other character of a given alphabet.

Thus, the space is partitioned into "crowns" $C_i(s, h_{i-1}, h_i)$ such that $C_i(s, h_{i-1}, h_i) = s'|h_{i-1} \leq ||s' - s|| \leq h_i$. The m neighbors of s are chosen by random selection of one point inside each crown C_i for i varying from 1 to m (see Fig. 3.4). This can be easily done by geometrical construction. The mapping consists in associating one thread with at least one neighbor corresponding to one point inside each crown. Thus, such a mapping is feasible. An application of this mapping is done for the Weierstrass function [LV98].

3.2.4 Permutation Representation

3.2.4.1 2-exchange Neighborhood

Building a neighborhood by pair-wise exchange operations is a standard way for permutation problems. For a permutation of size n, the size of the neighborhood is $\frac{n \times (n-1)}{2}$. Fig. 3.5 illustrates a permutation representation and its associated neighborhood.

Unlike the previous representations, in the case of a permutation encoding, the mapping between a neighbor and a GPU thread is not straightforward. Indeed, on the one hand, a neighbor is composed of two element indexes (a swap in a permutation). On the other hand, threads are identified by a unique *id*. Consequently, one mapping has to be considered to transform one index into two ones. In a similar way, another one is required to transform two indexes into one. Finding a nearly constant time mapping is clearly a challenging issue for permutation representation.

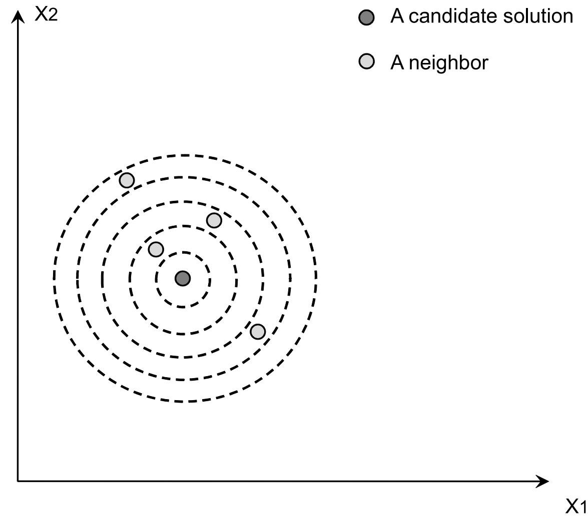


Figure 3.4: A neighborhood for a continuous problem with two dimensions. The neighbors are taken by random selection of one point inside each crown.

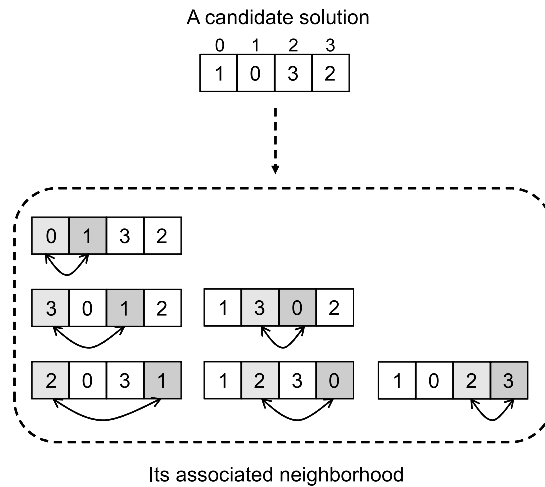


Figure 3.5: Permutation representation. A usual neighborhood is based on the swap operator which consists in exchanging the location of two elements of the candidate solution.

Proposition 3.2.1 ***Two-to-one index transformation***
Given i and j the indexes of two elements to be exchanged in the permutation representation, the corresponding index $f(i,j)$ in the neighborhood representation is equal to $i \times (n-1) + (j-1) - \frac{i \times (i+1)}{2}$, where n is the permutation size.

Proposition 3.2.2 ***One-to-two index transformation***
Given $f(i,j)$ the index of the element in the neighborhood representation, the corresponding index i is equal to $n - 2 - \lfloor \frac{\sqrt{8 \times (m - f(i,j) - 1) + 1} - 1}{2} \rfloor$ and j is equal to $f(i,j) - i \times (n-1) + \frac{i \times (i+1)}{2} + 1$ in the permutation representation, where n is the permutation size and m the neighborhood size.

The proofs of two-to-one and one-to-two index transformations can be found in Appendix .1.1 and Appendix .1.2. The complexity of such mappings is nearly in constant time i.e. it depends on the calculation of the square root on GPU (solving quadratic equation). Application of this mapping is done for the quadratic assignment problem. In a similar way, a mapping for a neighborhood based on a 2-opt operator has been applied to the traveling salesman problem. Moreover, a slight modification of the mapping has been applied to the permuted perceptron problem for a neighborhood based on a Hamming distance of two.

3.2.4.2 3-exchange Neighborhood

An instance of a large neighborhood is a neighborhood built by exchanging three values. Variants of this neighborhood such as 3-opt have been used for permutation problems [DG97].

For an array of size n, the size of this neighborhood is $\frac{n \times (n-1) \times (n-2)}{6}$. A mapping here between a neighbor and a GPU thread is also particularly challenging. One-to-three and three-to-one index transformations must be handled efficiently.

The mapping for this neighborhood is a generalization of the 2-exchange with a third index (see Appendix .1.3 and Appendix .1.4). The complexity of the mappings is logarithmic in practice i.e. it depends on the numerical Newton-Raphson method (solving cubic equation).

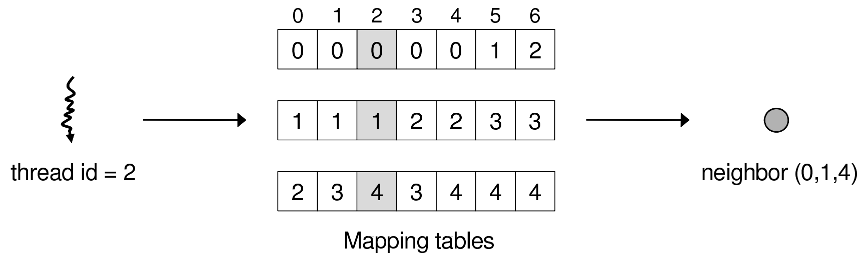


Figure 3.6: Construction of mapping tables.

3.2.4.3 Mapping Tables for General Neighborhoods

For dealing with more complex neighborhoods, finding a mapping might be more difficult. To release from such constraints, a common solution is to construct mapping tables on CPU, and to copy them once to the GPU global memory. In this manner, each thread just needs to retrieve its corresponding indexes in the mapping tables. Figure 3.6 illustrates this idea with a neighborhood based on 3-exchange operator.

The construction of mapping tables allows to deal with any neighborhoods. The drawback of this method is the extra cost due to additional global memory accesses. A study on how it impacts the global performance of S-metaheuristics on GPU will be investigated in Section 3.4.3.

3.3 First Improvement S-metaheuristics on GPU

The thread control heuristic and the efficient mapping of neighborhood structures are the key components of the parallelism control. Having these tools at hand, new S-metaheuristics on GPU can be designed. Therefore, we propose to investigate S-metaheuristics based on first improvement. These latter explore a partial neighborhood (e.g. simulated annealing). To achieve this, the parallelization scheme presented in Chapter 2 needs to be adapted when dealing with such S-metaheuristics. The difficulty essentially comes from the sequential nature of first improvement-based metaheuristics.

When it comes to the parallelization on traditional architectures, the neighborhood is decomposed into separate partitions that are generally of equal size. The partitions are generated and then evaluated in a parallel independent way. The exploration stops when an improving neighbor is found. This parallel model is asynchronous, and there is no need to explore the whole neighborhood.

Since the execution model of GPUs is purely SIMD, GPU computing is clearly inefficient in executing such algorithms in which the computations become asynchronous. In addition to this, since the execution order of threads in undefined, there is no inherent mechanism to stop the kernel during its execution.

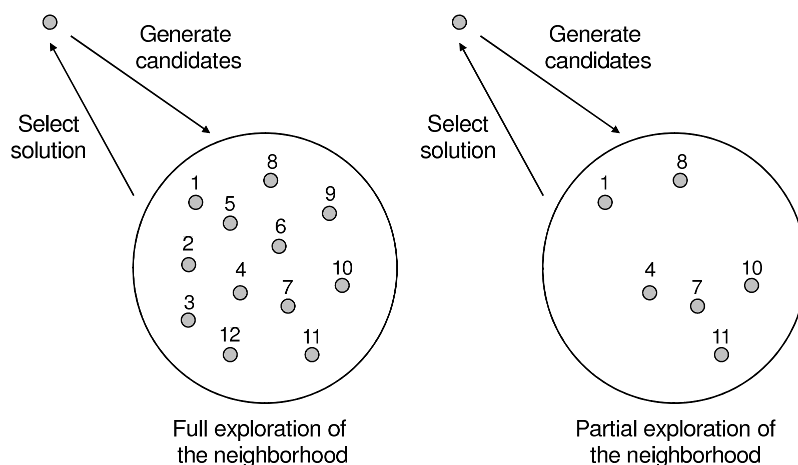


Figure 3.7: Full exploration and partial exploration of neighborhoods.

A way to deal with such asynchronous parallelization is to transform these algorithms into a data-parallel regular application. Thereby, one has to consider the previous parallelization scheme of the iteration-level on GPU, not applied to the whole neighborhood, but to a partial set of solutions. In other words, another approach is to generate and evaluate a partial set of solutions on GPU (see Figure 3.7). After this parallel evaluation, according to the S-metaheuristic, a specific post-treatment is performed on CPU on this partial set of solutions. Such a mechanism can be seen as a way to simulate a first improvement-based S-metaheuristic in a parallel way. From an implementation point of view, it is similar to the Algorithm 6 proposed in the previous chapter. The only difference concerns the partial set that has to be handled in which the neighbors are randomly chosen from the entire neighborhood. Once the number of neighbors has been set, the thread control heuristic presented in Section 3.1 can automatically adjust the remaining parameters.

Even if this approach to deal with such an asynchronous algorithm may be normal, it may be not efficient in comparison with a S-metaheuristic in which a full exploration of the neighborhood is performed on GPU. Indeed, to get a better global memory performance, memory accesses must constitute a contiguous range of addresses to be coalesced. This is not achieved for the exploration of a partial neighborhood since neighbors are randomly chosen.

Figure 3.8 illustrates a memory access pattern for the two different neighborhoods explorations. In the full exploration of the neighborhood (left case), all the neighbors are generated. Hence, the elements of the structures are dispatched such that many thread accesses will be coalesced into a single memory transaction. This does not happen in the partial exploration of the neighborhood (right case) since there is no connection between the elements to be accessed. Indeed, the different moves to be performed from the partial

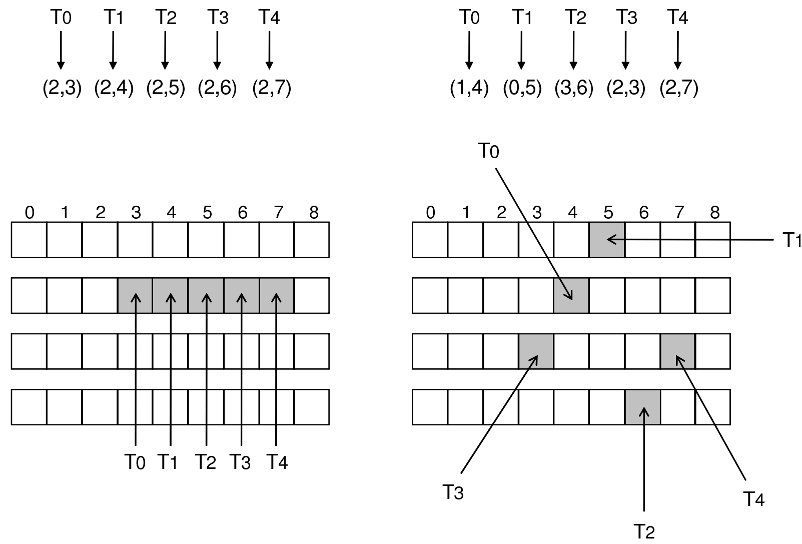


Figure 3.8: Illustration of a memory access pattern for two different neighborhoods explorations.

set of solutions are randomly chosen from the whole neighborhood. In this case, the different memory accesses have to be serialized, increasing the total number of instructions executed for this warp.
In Section 3.4.4, we will examine how such uncoalesced accesses to the global memory have an impact on the overall performance of S-metaheuristics.

3.4 Performance Evaluation

3.4.1 Thread Control for Preventing Crashes

As previously said, when dealing with a large solutions set, some experiments might fail at execution time. This is typically the case when too many threads are requested. This is due to the fact that a thread block allocates more registers than are available on a multiprocessor. Hence, the main issue is then to control the number of threads to meet the memory constraints. In the next section, we will feature an application which provokes errors at runtime. Thereafter, we will show how the thread control heuristic allows to prevent such errors.

3.4.1.1 Application to the Traveling Salesman Problem

For the next experiments, a tabu search with 10000 iterations is considered on the GTX 280 configuration. The number of threads per block has been arbitrary chosen to 256, and the total number of threads created at run time is equal to the neighborhood size. A 2-opt

Table 3.1: Measures in terms of efficiency for the traveling salesman problem using a pair-wise-exchange neighborhood (permutation representation).

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores**		**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores**	
	CPU	GPU	CPU	GPU
eil101	2.3	$1.8_{\times \mathbf{1.2}}$	1.7	$1.1_{\times \mathbf{1.5}}$
d198	10	$6.9_{\times \mathbf{1.4}}$	7.3	$3.2_{\times \mathbf{2.3}}$
pcb442	57	$36_{\times \mathbf{1.5}}$	29	$14_{\times \mathbf{2.2}}$
rat783	196	$144_{\times \mathbf{1.4}}$	107	$42_{\times \mathbf{2.8}}$
d1291	692	$503_{\times \mathbf{1.4}}$	492	$140_{\times \mathbf{3.5}}$
pr2392	3389	.	2318	$531_{\times \mathbf{4.4}}$
fnl4461	14817	.	11710	.
rl5915	27946	.	20935	.
Instance	**Xeon E5450 GeForce GTX 280 240 GPU cores**		**Xeon E5620 Tesla M2050 448 GPU cores**	
	CPU	GPU	CPU	GPU
eil101	1.4	$0.6_{\times \mathbf{2.3}}$	1.7	$0.4_{\times \mathbf{4.2}}$
d198	5.6	$1.3_{\times \mathbf{4.4}}$	6.1	$0.8_{\times \mathbf{7.5}}$
pcb442	29	$6.0_{\times \mathbf{4.5}}$	27	$3.5_{\times \mathbf{7.6}}$
rat783	93	$20_{\times \mathbf{4.7}}$	86	$11_{\times \mathbf{7.8}}$
d1291	365	$71_{\times \mathbf{5.1}}$	364	$43_{\times \mathbf{8.5}}$
pr2392	2389	$286_{\times \mathbf{8.4}}$	2399	$161_{\times \mathbf{14.9}}$
fnl4461	11274	$1125_{\times \mathbf{11.0}}$	11642	$616_{\times \mathbf{18.9}}$
rl5915	20710	.	16499	$837_{\times \mathbf{19.7}}$

operator for the TSP has been implemented on GPU. The considered instances have been selected among the TSPLIB instances presented in [DG97].

Table 10 presents the results for the traveling salesman problem. On the one hand, even if a large number of threads are executed ($\frac{n \times (n-1)}{2}$ neighbors), the values for the first configuration are not significant (acceleration factor from $\times 1.2$ to $\times 1.5$). Indeed, the neighbor evaluation function consists of replacing two edges of a solution. As a result, this computation can be given in constant time, which is not enough to hide the memory latency. Regarding the other configurations, using more cores overcomes the issue and results in a better global performance. Indeed, for the GeForce 8800, accelerations start from $\times 1.5$ with the eil101 instance and grows up to $\times 4.4$ for pr2392. In a similar manner, the GTX 280 starts from $\times 2.3$ and goes up to an acceleration factor of $\times 11$ for the fnl4461 instance. Nevertheless, for the three first configurations, for larger instances such as pr2392, fnl4461 or rl5915, the program has provoked an execution error because of the hardware register limitation. Such a problem does not exist for the Tesla M2050 since more registers are available on this card.

Table 3.2: Measures of the benefits of applying thread control. The traveling salesman problem using a neighborhood based on a 2-opt operator is considered.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores**			**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores**		
	CPU	GPU	GPUTC	CPU	GPU	GPUTC
eil101	2.3	$1.8_{\times \mathbf{1.2}}$	$1.7_{\times \mathbf{1.3}}$	1.7	$1.1_{\times \mathbf{1.5}}$	$0.9_{\times \mathbf{1.8}}$
d198	10	$6.9_{\times \mathbf{1.4}}$	$6.8_{\times \mathbf{1.5}}$	7.3	$3.2_{\times \mathbf{2.3}}$	$3.0_{\times \mathbf{2.4}}$
pcb442	57	$36_{\times \mathbf{1.5}}$	$34_{\times \mathbf{1.6}}$	29	$14_{\times \mathbf{2.2}}$	$13_{\times \mathbf{2.4}}$
rat783	196	$144_{\times \mathbf{1.4}}$	$141_{\times \mathbf{1.4}}$	107	$42_{\times \mathbf{2.8}}$	$39_{\times \mathbf{3.0}}$
d1291	692	$503_{\times \mathbf{1.4}}$	$498_{\times \mathbf{1.4}}$	492	$140_{\times \mathbf{3.5}}$	$133_{\times \mathbf{3.7}}$
pr2392	3389	.	$1946_{\times \mathbf{1.7}}$	2318	$531_{\times \mathbf{4.4}}$	$519_{\times \mathbf{4.5}}$
fnl4461	14817	.	$7133_{\times \mathbf{2.1}}$	11710	.	$1789_{\times \mathbf{6.6}}$
rl5915	27946	.	$9142_{\times \mathbf{3.1}}$	20935	.	$2471_{\times \mathbf{8.5}}$

Instance	**Xeon E5450 GeForce GTX 280 240 GPU cores**			**Xeon E5620 Tesla M2050 448 GPU cores**		
	CPU	GPU	GPUTC	CPU	GPU	GPUTC
eil101	1.4	$0.6_{\times \mathbf{2.3}}$	$0.6_{\times \mathbf{2.4}}$	1.7	$0.4_{\times \mathbf{4.2}}$	$0.4_{\times \mathbf{4.3}}$
d198	5.6	$1.3_{\times \mathbf{4.4}}$	$1.3_{\times \mathbf{4.5}}$	6.1	$0.8_{\times \mathbf{7.5}}$	$0.8_{\times \mathbf{7.6}}$
pcb442	29	$6.0_{\times \mathbf{4.5}}$	$5.8_{\times \mathbf{4.7}}$	27	$3.5_{\times \mathbf{7.6}}$	$3.4_{\times \mathbf{7.8}}$
rat783	93	$20_{\times \mathbf{4.7}}$	$19_{\times \mathbf{4.9}}$	86	$11_{\times \mathbf{7.8}}$	$10_{\times \mathbf{8.3}}$
d1291	365	$71_{\times \mathbf{5.1}}$	$68_{\times \mathbf{5.4}}$	364	$43_{\times \mathbf{8.5}}$	$41_{\times \mathbf{8.9}}$
pr2392	2389	$286_{\times \mathbf{8.4}}$	$278_{\times \mathbf{8.6}}$	2399	$161_{\times \mathbf{14.9}}$	$157_{\times \mathbf{15.3}}$
fnl4461	11274	$1125_{\times \mathbf{11.0}}$	$1110_{\times \mathbf{11.2}}$	11642	$616_{\times \mathbf{18.9}}$	$609_{\times \mathbf{19.1}}$
rl5915	20710	.	$1461_{\times \mathbf{14.2}}$	16499	$837_{\times \mathbf{19.7}}$	$828_{\times \mathbf{19.9}}$

3.4.1.2 Thread Control Applied to the Traveling Salesman Problem

Since the GPU may fail to execute large neighborhoods on large instances, the next experiment consists in highlighting the benefits of thread control presented in Section 3.1. The associated heuristic based on thread "waves" has been applied for the traveling salesman problem previously seen. The only required parameter is the number of trials which defines the accuracy of each parameter configuration (i.e. the number of threads per block and the total number of threads). The value of this parameter has been fixed to 10. Table 13 presents the obtained results for the traveling salesman problem.

The first observation concerns the robustness provided by the thread control version for large instances pr2392, fnl4461 and rl5915. Indeed, one can clearly see the great benefits of such control since the execution of these instances on GPU has been successfully terminated whatever the used card. Indeed, according to some execution logs, the heuristic is fault-tolerant since it is able to detect kernel errors at run time (e.g. the number of registers, a bad configuration of the parameters, etc.), and to restore the previous functional state of the algorithm. Regarding the acceleration factors using the thread control, they

alternate between ×1.3 and ×19.9 according to the instance size (GPU_{TC}). Performance improvement in comparison with the standard version varies between 1% and 5%, which is not uncommonly significant. Furthermore, statistical analysis for some instances cannot determine if the distribution of the averages between the two algorithms is different. This can be explained by the fact that the instances are really large thus the neighborhood size is also impacted. Indeed, since the number of iterations for tuning is directly linked to the neighborhood size, the algorithm may take too much iterations to get a suitable parameters tuning.

3.4.2 Thread Control for Further Optimization

The thread control prevents the application from crashing and may improve the algorithm performance rather than a parameters tuning made at compile time. The following experiment intends to highlight the benefits of applying thread control in terms of performance optimization. The considered problem is the permuted perceptron problem using a neighborhood based on a hamming distance of two. Table 3.3 reports the obtained results. In a general manner, performance results obtained for the thread control (GPU_{TC}) are significantly improved in comparison with its counterpart without control. Indeed, the acceleration factors alternate from ×3.8 to ×81.4 for GPU_{TC} against ×3.6 to ×73.3. This is significant since the results correspond to a performance enhancement from 5% to 20%. In comparison with the traveling salesman problem, such an improvement is explained by the fact that the neighborhood is smaller. That is the reason why, the dynamic heuristic spends less time to find a suitable parameters auto-tuning.
In a general manner, the same observations can be made for other optimization problems. A thread control can provide a performance improvement from 3% to 20%. The obtained speed-ups for the other problems are reported in Table 3.4 for a tabu search using the same parameters.

3.4.3 Performance of User-defined Mappings

As previously said, for generating the neighborhood on GPU, one of the critical issues is to find efficient mappings between a GPU thread and a particular neighbor. Indeed, this step is crucial in the design of S-metaheuristics on GPU since it is clearly identified as the gateway between a GPU process and a candidate neighbor. On the one hand, the thread id is represented by a single index. On the other hand, the move representation of a neighbor varies according to the neighborhood.
A simple way to overcome this issue is to build mapping tables. In this way, each thread just needs to access to its corresponding value via the global memory (see Section 3.2.4.3). However, even if this approach undoubtedly simplifies the mapping between a thread

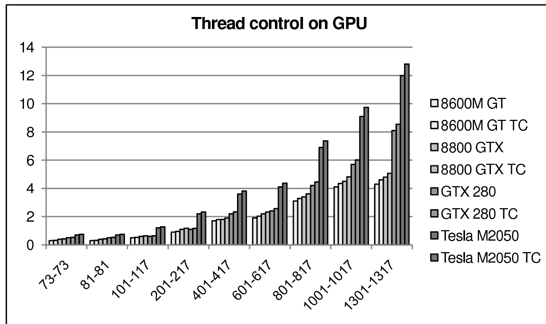
Table 3.3: Measures of the benefits of applying thread control. The permuted perceptron problem using a neighborhood based on a Hamming distance of two is considered.

Instance	**Core 2 Duo T5800 GeForce 8600M GT**		**Core 2 Quad Q6600 GeForce 8800 GTX**	
	GPU	GPUTC	GPU	GPUTC
73-73	$0.8_{\times \mathbf{3.6}}$	$0.8_{\times \mathbf{3.8}}$	$0.2_{\times \mathbf{10.1}}$	$0.2_{\times \mathbf{10.3}}$
81-81	$1.1_{\times \mathbf{3.8}}$	$1.0_{\times \mathbf{4.2}}$	$0.3_{\times \mathbf{10.4}}$	$0.3_{\times \mathbf{10.9}}$
101-117	$2.5_{\times \mathbf{4.4}}$	$2.4_{\times \mathbf{4.6}}$	$0.6_{\times \mathbf{12.4}}$	$0.6_{\times \mathbf{13.7}}$
201-217	$15_{\times \mathbf{4.7}}$	$13_{\times \mathbf{5.4}}$	$3.3_{\times \mathbf{15.4}}$	$2.9_{\times \mathbf{17.6}}$
401-417	$103_{\times \mathbf{5.4}}$	$92_{\times \mathbf{6.2}}$	$24_{\times \mathbf{18.3}}$	$21_{\times \mathbf{21.0}}$
601-617	$512_{\times \mathbf{6.3}}$	$446_{\times \mathbf{7.2}}$	$89_{\times \mathbf{28.3}}$	$78_{\times \mathbf{32.3}}$
801-817	$1245_{\times \mathbf{6.9}}$	$1064_{\times \mathbf{8.1}}$	$212_{\times \mathbf{32.8}}$	$182_{\times \mathbf{38.1}}$
1001-1017	$2421_{\times \mathbf{7.2}}$	$2087_{\times \mathbf{8.4}}$	$409_{\times \mathbf{35.2}}$	$350_{\times \mathbf{41.3}}$
1301-1317	$4903_{\times \mathbf{8.0}}$	$4265_{\times \mathbf{9.2}}$	$911_{\times \mathbf{36.2}}$	$779_{\times \mathbf{42.5}}$

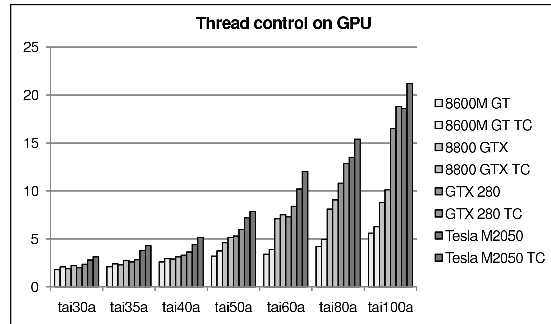
Instance	**Xeon E5450 GeForce GTX 280**		**Xeon E5620 Tesla M2050**	
	GPU	GPUTC	GPU	GPUTC
73-73	$0.2_{\times \mathbf{10.9}}$	$0.2_{\times \mathbf{11.6}}$	$0.2_{\times \mathbf{12.3}}$	$0.2_{\times \mathbf{12.8}}$
81-81	$0.2_{\times \mathbf{12.2}}$	$0.2_{\times \mathbf{12.8}}$	$0.2_{\times \mathbf{13.4}}$	$0.2_{\times \mathbf{13.7}}$
101-117	$0.4_{\times \mathbf{18.1}}$	$0.4_{\times \mathbf{19.9}}$	$0.3_{\times \mathbf{22.0}}$	$0.3_{\times \mathbf{23.2}}$
201-217	$1.9_{\times \mathbf{25.3}}$	$1.6_{\times \mathbf{30.1}}$	$1.6_{\times \mathbf{30.6}}$	$1.4_{\times \mathbf{34.9}}$
401-417	$14_{\times \mathbf{28.8}}$	$13_{\times \mathbf{31.2}}$	$10_{\times \mathbf{38.3}}$	$8.9_{\times \mathbf{43.0}}$
601-617	$51_{\times \mathbf{40.1}}$	$45_{\times \mathbf{45.3}}$	$35_{\times \mathbf{58.4}}$	$31_{\times \mathbf{65.9}}$
801-817	$128_{\times \mathbf{42.3}}$	$109_{\times \mathbf{49.6}}$	$81_{\times \mathbf{67.1}}$	$72_{\times \mathbf{75.5}}$
1001-1017	$252_{\times \mathbf{43.9}}$	$216_{\times \mathbf{51.3}}$	$154_{\times \mathbf{71.9}}$	$138_{\times \mathbf{80.2}}$
1301-1317	$568_{\times \mathbf{44.1}}$	$485_{\times \mathbf{51.7}}$	$342_{\times \mathbf{73.3}}$	$308_{\times \mathbf{81.4}}$

Table 3.4: Benefits of the thread control on GPU. The acceleration factors are reported for a tabu search on GPU on different optimization problems.

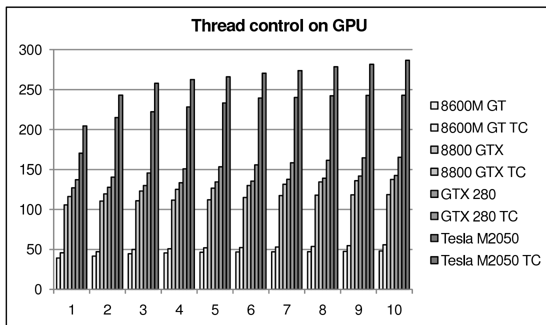
The permuted perceptron problem. A neighborhood based on a Hamming distance of one is considered.



The quadratic assignment problem. A neighborhood based on a pair-wise exchange operator is considered.



The Weierstrass continuous function. 10000 neighbors are considered.



The Golomb rulers. $n^3 - n$ neighbors are considered.

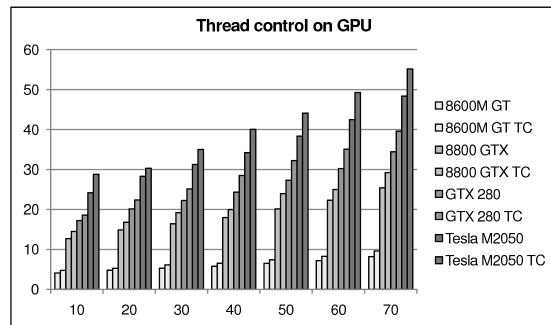


Table 3.5: Measures of the benefits of using user-defined mappings instead of constructed mapping tables on the GTX 280. The permuted perceptron problem is considered for two different neighborhoods using a tabu search.

Instance	n neighbors			$\frac{n \times (n-1)}{2}$ neighbors		
	CPU	GPU	GPUMT	CPU	GPU	GPUMT
73-73	1.1	$3.0_{\times 0.4}$	$3.2_{\times 0.4}$	2.1	$0.2_{\times \mathbf{10.9}}$	$0.2_{\times \mathbf{9.5}}$
81-81	1.3	$3.3_{\times 0.4}$	$3.5_{\times 0.4}$	2.7	$0.2_{\times \mathbf{12.2}}$	$0.3_{\times \mathbf{10.6}}$
101-117	2.2	$4.2_{\times 0.5}$	$4.5_{\times 0.5}$	7.0	$0.4_{\times \mathbf{18.1}}$	$0.4_{\times \mathbf{15.6}}$
201-217	8.1	$7.7_{\times \mathbf{1.1}}$	$8.3_{\times 1.0}$	48	$1.9_{\times \mathbf{25.3}}$	$2.2_{\times \mathbf{21.5}}$
401-417	31	$14_{\times \mathbf{2.2}}$	$16_{\times \mathbf{2.0}}$	403	$14_{\times \mathbf{28.8}}$	$16_{\times \mathbf{24.8}}$
601-617	105	$43_{\times \mathbf{2.4}}$	$48_{\times \mathbf{2.2}}$	2049	$51_{\times \mathbf{40.1}}$	$61_{\times \mathbf{33.7}}$
801-817	200	$50_{\times \mathbf{4.0}}$	$54_{\times \mathbf{3.8}}$	5410	$128_{\times \mathbf{42.3}}$	$149_{\times \mathbf{36.0}}$
1001-1017	336	$58_{\times \mathbf{5.8}}$	$66_{\times \mathbf{5.1}}$	11075	$252_{\times \mathbf{43.9}}$	$297_{\times \mathbf{37.3}}$
1301-1317	687	$85_{\times \mathbf{8.0}}$	$94_{\times \mathbf{7.3}}$	25016	$568_{\times \mathbf{44.1}}$	$661_{\times \mathbf{37.9}}$

and neighboring solutions, it implies additional accesses to the global memory. Table 3.5 reports the obtained results with user-defined mappings and constructed mapping tables for the permuted perceptron problem.

Regarding the neighborhood based on a Hamming distance of one (n neighbors), the obtained acceleration factors from the version using a mapping table (GPUMT) are quite close to the original version. Indeed, they alternate from $\times 0.4$ to $\times 7.3$ (against from $\times 0.4$ to $\times 8.0$). This can be clarified by the fact the number of accesses is not majorly important (i.e. mapping table of n elements). However, when considering a bigger neighborhood based on a Hamming distance of two, the performance gap which occurs is quite remarkable. The new obtained speed-ups vary from $\times 9.5$ to $\times 37.9$ for GPUMT (against from $\times 10.9$ to $\times 44.1$ for the original version). Such performance difference alternates from 5% to 15% according to the instance. As a consequence, such a performance diminution is fairly significant, and it justifies the use of user-defined mapping when increasing the neighborhood size.

In a similar way, the same observations can be stated for the other problems where the use of mapping tables generates a 5% to 20% performance degradation (see Table 3.6).

3.4.4 First Improvement S-metaheuristics on GPU

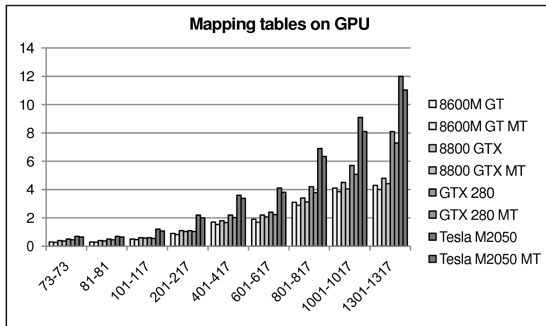
S-metaheuristics which explore a partial neighborhood may look difficult to parallelize on GPU due to their sequential natures. One solution resides in changing the original semantic of the algorithm. It consists in generating and evaluating a partial set of solutions of the neighborhood. Then according to the S-metaheuristic, a post-treatment is performed on the CPU side.

However, as previously said in Section 3.3, since moves to be performed are randomly

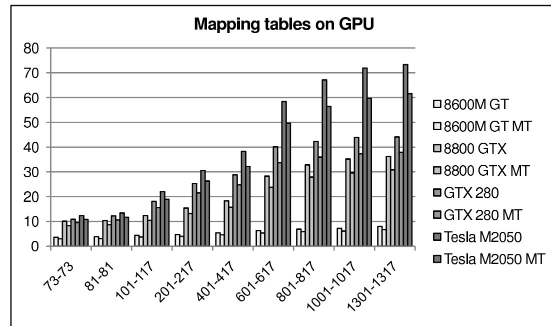
Table 3.6: Benefits of the use of user-defined mappings in comparison with constructed mapping tables. The acceleration factors are reported for a tabu search on GPU on different optimization problems.

The permuted perceptron problem.

A neighborhood based on a Hamming distance of one is considered.

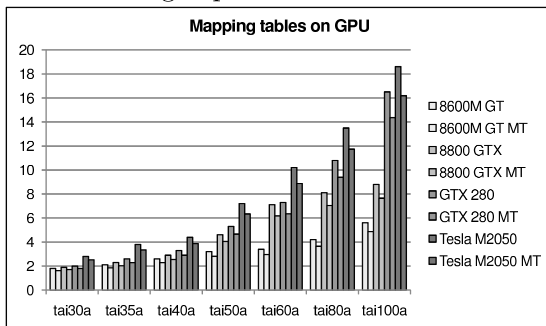


A neighborhood based on a Hamming distance of two is considered.



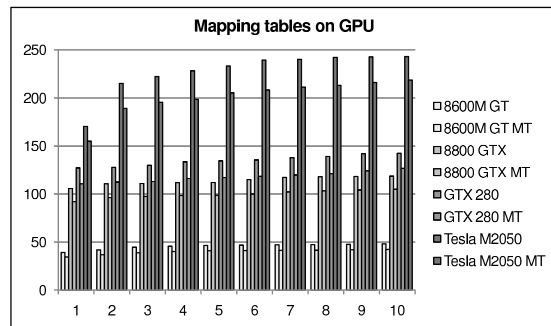
The quadratic assignment problem.

A neighborhood based on a pair-wise exchange operator is considered.



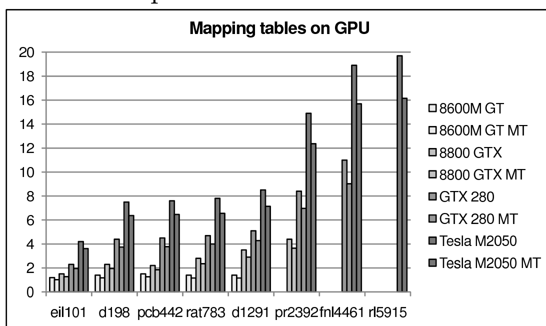
The Weierstrass continuous function.

10000 neighbors are considered.



The traveling salesman problem.

A neighborhood based on a two-opt operator is considered.



The Golomb rulers.

$n^3 - n$ neighbors are considered.

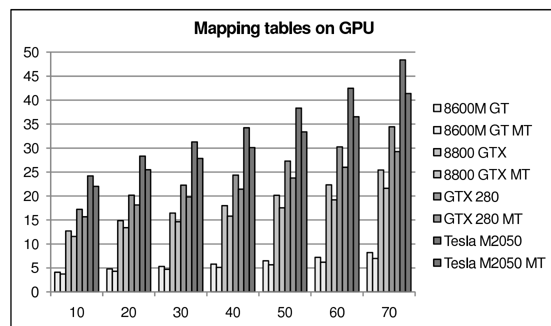


Table 3.7: Measures in terms of efficiency of two different exploration strategies. The quadratic assignment problem using a 3-exchange neighborhood is considered.

Instance	Intel Xeon E5450 GeForce GTX 280 240 GPU cores		
	CPU	$GPU_{ILSHCFE}$	$GPU_{ILSHCPE}$
tai30a	14	$1.5_{\times \mathbf{9.5}}$	$4.8_{\times \mathbf{3.8}}$
tai35a	26	$2.5_{\times \mathbf{10.2}}$	$6.3_{\times \mathbf{4.1}}$
tai40a	45	$4.1_{\times \mathbf{11.1}}$	$8.5_{\times \mathbf{5.3}}$
tai50a	112	$8.2_{\times \mathbf{13.7}}$	$17_{\times \mathbf{6.5}}$
tai60a	217	$14_{\times \mathbf{15.4}}$	$29_{\times \mathbf{7.4}}$
tai80a	757	$43_{\times \mathbf{17.8}}$	$91_{\times \mathbf{8.3}}$
tai100a	1978	$93_{\times \mathbf{21.2}}$	$206_{\times \mathbf{9.6}}$

chosen, accesses to data structures via the global memory may be serialized, leading to a performance decrease. A next experiment consists in evaluating this performance degradation for a S-metaheuristic which explores a partial neighborhood. To measure this, this latter algorithm is directly compared with another one which operates a full exploration of the neighborhood.

For doing this, an iterated local search is considered for the quadratic assignment problem is considered on the GTX 280 configuration. The embedded algorithm is a hill climbing heuristic. The first version concerns a full exploration of the neighborhood whereas the second one considers a partial exploration. The neighborhood is based on a 3-exchange operator. Such a large neighborhood ensures that the number of threads is enough to cover the access latency for not introducing any bias into the current experiment.

The stopping criterion is fixed to a certain number of evaluations which is equivalent to 10000 iterations for a full exploration. Regarding the strategy of a partial exploration, experiments have been performed with a number of neighbors per iteration fixed to 1024, 2048 and 4096. The thread control heuristic is in charge of tuning the parameters. Thereafter, only the average of the best of the three configurations is reported in the next results. Furthermore, since there is no way to detect the convergence to a local optima in the partial exploration strategy, the number of iterations is fixed to 50 before applying the perturbation. All the obtained acceleration factors are made in comparison with an iterated local search with a full exploration strategy made on a single CPU core. Table 3.7 reports the obtained results for the two algorithms.

In a general manner, for the iterative local search based on a full exploration ($GPU_{ILSHCFE}$), speed-ups grow with the instance size. They vary from ×9.5 to ×21.2. One can also observe that these acceleration factors are more salient than those provided with a neighborhood based on a 2-exchange. Regarding the iterative local search based on a partial exploration ($GPU_{ILSHCPE}$), accelerations also grow with the size increase (from ×3.8 to ×9.6). How-

Table 3.8: Analysis of the execution path of two different exploration strategies provided by the CUDA Profiler. The instance tai50a is considered.

Algorithm	**Intel Xeon E5450 GeForce GTX 280 240 GPU cores**			
	Execution time	Branches	Divergent branches	Warp serializations
ILSHCFE	$8.2_{\times \mathbf{13.7}}$	439010221	28386975	157034001
ILSHCPE 1024	$22_{\times \mathbf{5.2}}$	480112132	68814552	661318575
ILSHCPE 2048	$17_{\times \mathbf{6.5}}$	479829773	68788731	576102293
ILSHCPE 4096	$19_{\times \mathbf{6.0}}$	480053219	68787615	604398213

ever, as expected, the obtained results for the partial exploration of the neighborhood are clearly less effective than a full exploration strategy. This performance difference essentially comes from random moves in the partial exploration, leading to non-coalesced accesses to the global memory.

To confirm this point, an analysis of the execution path of the two different exploration strategies is required. To achieve this, the CUDA Profiler [NVI11] provides a tool to examine the number of branches taken. Table 3.8 reports the results for the instance $tai50a$.

Regarding the presented results, in addition to the full exploration-based algorithm, partial exploration-based algorithms are considered with three different neighborhood sizes. In general, the number of branches taken by threads is similar for each algorithm. However, one can clearly observe that the number of divergent branches taken by threads in each first exploration-based algorithm is at least twice more important than the full exploration strategy. This is typically the results from additional non-coalesced accesses to the global memory. Hence, such a threads divergence leads to many memory accesses that have to be serialized, increasing the total number of instructions executed. Indeed, the number of warp serializations for a partial exploration is about five times more important than a full exploration-based algorithm. Such an analysis confirms the performance difference which occurs in the two exploration strategies.

3.5 Large Neighborhoods for Improving Solutions Quality

Up to now, as the original iteration-level does not change the semantics of the sequential algorithm, the effectiveness in terms of quality of solutions has not been yet investigated. Thereby, it will be interesting to see how the definition of the neighborhood in S-metaheuristics plays a crucial role in the performance of the algorithm. Indeed, theoretical and experimental studies have shown that the increase of the neighborhood size may improve the quality of the obtained solutions [AGM+07]. Nevertheless, as it is mostly CPU

time-consuming, this mechanism is not often fully exploited in practice. So, in practice, large neighborhoods algorithms are unusable because of their high computational cost. Indeed, experiments using large neighborhoods are often stopped without convergence being reached. Thereby, in designing S-metaheuristics, there is often a trade-off between the size of the neighborhood to use and the computational complexity to explore it. To deal with such issues, only the use of parallelism allows to design methods based on large neighborhood structures. We have shown in [2, 8] how the use of GPU computing allows to exploit parallelism in such algorithms.

3.5.1 Application to the Permuted Perceptron Problem

As an application, a tabu search has been implemented on GPU for the permuted perceptron problem. Three different neighborhoods based on different Hamming distances are considered. Indeed, usual neighborhoods for solving binary problems are in general a linear (e.g. 1-Hamming distance) or quadratic (e.g. 2-Hamming distance) function of the input instance size. Some large neighborhoods may be high-order polynomial of the size of the input instance (e.g. 3-Hamming distance).

The used configuration is the third configuration with the NVIDIA GTX 280 card. The following experiments intend to assess the quality of solutions for the four instances of the literature addressed in [KM99]. A tabu search has been executed 50 times with a maximum number of $\frac{n \times (n-1) \times (n-2)}{6}$ iterations. The tabu list size has been arbitrary set to $\frac{m}{6}$ where m is the number of neighbors. The average value of the evaluation function (fitness) and its standard deviation (in sub index) have been measured. The number of successful tries (fitness equal to zero) and the average number of iterations to converge to a solution are also represented.

3.5.1.1 Neighborhood based on a 1-Hamming Distance

Table 3.9 reports the results for the tabu search based on the 1-Hamming distance neighborhood. In a short execution time, the algorithm has been able to find few solutions for the instances $m = 73, n = 73$ (11 successful tries on 50) and $m = 81, n = 81$ (5 successful tries on 50). The two other instances are well-known for their difficulties and no solutions were found. Regarding the execution time, the GPU version does not offer anything in terms of efficiency. Indeed, since the neighborhood is relatively small (n threads), the number of threads per block is not enough to fully cover the memory access latency. To measure the efficiency of the GPU-based implementation of this neighborhood, bigger instances of the permuted perceptron problem should be considered.

Table 3.9: Results obtained using a neighborhood based on a 1-Hamming Distance.

Problem	Fitness	# iterations	# solutions	CPU time	GPU time
73×73	$9.8_{6.2}$	59891.2	11/50	$4s$	$9s$
81×81	$11.3_{6.6}$	72345.2	5/50	$6s$	$13s$
101×101	$20.7_{12.2}$	166650	0/50	$16s$	$33s$
101×117	$16.8_{8.4}$	260130	0/50	$29s$	$57s$

Table 3.10: Results obtained using a neighborhood based on a 2-Hamming Distance.

Problem	Fitness	# iterations	# solutions	CPU time	GPU time	Acc.
73×73	$15.5_{18.1}$	42142.7	22/50	$81s$	$10s$	$\times 8.2$
81×81	$16.2_{13.2}$	65421.3	17/50	$174s$	$16s$	$\times 11.0$
101×101	$13.1_{13.8}$	133211.2	13/50	$748s$	$44s$	$\times 17.0$
101×117	$12.7_{9.9}$	260130	0/50	$1947s$	$105s$	$\times 18.5$

3.5.1.2 Neighborhood based on a 2-Hamming Distance

A tabu search has been implemented on GPU using a neighborhood based on a Hamming distance of two. Results of the experiment for the permuted perceptron problem are reported in Table 3.10.
By using this other neighborhood, in comparison with Table 3.9, the quality of solutions has been significantly improved: on the one side the number of successful tries for both $m = 73, n = 73$ (22 solutions) and $m = 81, n = 81$ (17 solutions) is more prominent. On the other side, 13 solutions were found for the instance $m = 101, n = 101$. Regarding the execution time, the acceleration factor for the GPU version is remarkably efficient (from $\times 8.2$ to $\times 18.5$). Indeed, since a large number of threads are executed, the GPU can take full advantage of the multiprocessors occupancy.

3.5.1.3 Neighborhood based on a 3-Hamming Distance

A tabu search using a neighborhood based on Hamming distance of three has been implemented. The obtained results are collected in Table 3.11.
In comparison with Knudsen and Meier's article [KM99], the results found by the generic

Table 3.11: Results obtained using a neighborhood based on a 3-Hamming Distance.

Problem	Fitness	# iterations	# solutions	CPU time	GPU time	Acc.
73×73	$2.5_{4.1}$	19341.5	39/50	$1202s$	$50s$	$\times 24.2$
81×81	$3.2_{4.2}$	40636.6	33/50	$3730s$	$146s$	$\times 25.5$
101×101	$5.8_{5.1}$	100113.1	22/50	$24657s$	$955s$	$\times 25.8$
101×117	$7.1_{2.3}$	214092.9	3/50	$88151s$	$3351s$	$\times 26.3$

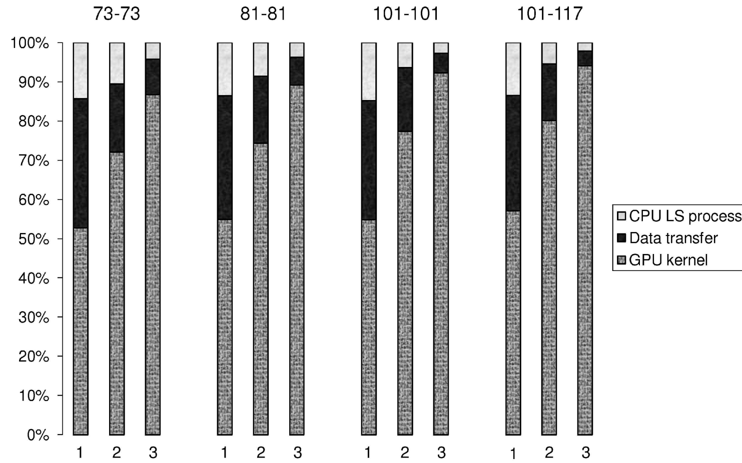


Figure 3.9: Analysis of the time spent for each major operation. The 3 different neighborhoods are compared and the instances are ordered according to their size.

tabu search are competitive without any use of cryptanalysis techniques. Indeed, the number of successful solutions has been drastically improved for every instance (respectively 39, 33 and 22 successful tries) and 3 solutions have been even found for the difficult instance $m = 101, n = 117$. Regarding the execution time, acceleration factors using the GPU are highly significant (from $\times 24.2$ to $\times 26.2$).
The conclusions from this experiment indicate that the use of the GPU provides an efficient way to deal with large neighborhoods. Indeed, a neighborhood based on a Hamming distance of three on the permuted perceptron problem was unpractical in terms of single CPU computational resources. Therefore, implementing this algorithm on GPU has allowed to exploit the parallelism in such neighborhood and to improve the quality of solutions. Furthermore, we strongly believe that the quality of the solutions would be drastically enhanced by (1) increasing the number of running iterations of the algorithm and (2) introducing appropriate cryptanalysis heuristics.

3.5.1.4 Performance Analysis

To validate the performance of the algorithms, we propose to make an analysis of the time spent by each major operation to assess its impact in terms of efficiency. The obtained results are reported in Fig. 3.9.
For the first neighborhood (n neighbors), one can notice that the time spent by the data transfer is significant. For example, it represents 28% of the total execution time for the instance $m = 73, n = 73$. The same goes on for the other instances. Furthermore, regarding the time spent on the search process on CPU, almost 15% is dedicated to this task whatever the instance size. As a consequence, since only half of the total execution

time is dedicated to the GPU kernel, the amount of computation may not be enough to fully cover the memory access latency. That is the reason why, no acceleration is provided for a tabu search based on a neighborhood with a Hamming distance of one (see Table 3.9). To go on with the idea, if a neighborhood based on a Hamming distance of two is considered ($\frac{n \times (n-1)}{2}$ neighbors), one can notice that the time spent of the GPU calculation is greater than 70% for each instance. This time is almost equal to 90% for a neighborhood based on Hamming distance of three. Thereby, regarding the time spent on both the data transfer and the search process on CPU, it tends to decrease with 1) the number of neighbors; 2) the instance size increase. Indeed, this can be explained by the fact that in designing S-metaheuristics, these two latter parameters usually have a significant influence on the total execution time and thus the amount of calculation. As a result, in accordance with the previous results (see Table 3.10 and Table 3.11), algorithms based on bigger neighborhoods clearly improve the GPU occupancy i.e. the amount of calculation performed by the GPU, leading to a better global performance.

Conclusion

In this chapter, we have dealt with the efficient control of parallelism for the iteration-level on GPU. In this challenge, a clear understanding of the thread-based execution model allows to design new algorithms, and to improve the performance of metaheuristics on GPU.

- **Thread control heuristic**. When dealing with applications requiring a large number of threads, some experiments might not be conducted. The control of the generation of threads is a must to meet the memory constraints like the limited number of registers allocated to each thread. We have proposed an efficient thread control heuristic to automatically tune the different parameters involved during the kernel execution. Such a control introduces some fault-tolerance mechanisms in GPU applications. Further, it may provide an additional performance improvement.

- **Mapping of neighborhood structures**. Regarding the generation of the neighborhood for S-metaheuristics, finding a mapping between a GPU thread and neighbor solutions might represent a complex issue. In this purpose, we have provided a methodology to deal with the main structures of the literature. Performance results indicate that such user-defined mappings provide a performance improvement in comparison with predetermined tables.

- **First improvement-based S-metaheuristics**. The parallelism control allows to develop new algorithms. In this purpose, we have discussed the possible utilization of first improvement-based S-metaheuristics on GPU architectures. Thereby, we have shown that their use might be well-adapted to GPU architectures, due to their asynchronous and sequential natures.

- **Large neighborhoods**. The application of S-metaheuristics based on large neighborhoods might not be conducted on traditional architectures because of its high computational cost. In this purpose, GPU computing allows not only to speed up the search process, but also to exploit parallelism to improve the quality of the obtained solutions.

Chapter 4

Efficient Memory Management

Efficient cooperation and parallelism control have been proposed in the previous chapters. In this chapter, the scope is on the memory management on GPU architectures. Understanding the hierarchical organization of the different memories is useful to provide an efficient implementation of parallel metaheuristics.
First, we will introduce concepts of memory management common to all metaheuristics. We will briefly explain how different memory techniques affect the performance of GPU applications. Then, the focus of this chapter will be on the parallel and cooperative model on GPU. Indeed, the memory management is more prominent when dealing with these cooperative algorithms. Thereby, we will investigate on how interactions between the threads in P-metaheuristics might be exploited on the hierarchical GPU. In this purpose, traditional mechanisms of cooperative algorithms are revisited on GPU to achieve better performance.

Contents

4.1 Common Concepts of Memory Management 89
4.1.1 Memory Coalescing Issues 89
4.1.2 Coalescing Transformation 90
4.1.3 Texture Memory 91
4.1.4 Memory Management 92
4.2 Memory Management in Cooperative Algorithms 94
4.2.1 Parallel and Cooperative Model 94
4.2.2 Parallelization Strategies for Cooperative Algorithms 96
4.2.3 Issues Related to the Fully Distributed Schemes 101
4.3 Performance Evaluation 106
4.3.1 Coalescing accesses to global memory 106
4.3.2 Memory Associations of Optimization Problems 107
4.4 Performance of Cooperative Algorithms 108
4.4.1 Configuration 109
4.4.2 Measures in Terms of Efficiency 109
4.4.3 Measures in Terms of Effectiveness 113

Main publications related to this chapter

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU-based Island Model for Evolutionary Algorithms. *Genetic and Evolutionary Computation Conference, GECCO 2010* pages 1089–1096, Proceedings, ACM, 2010.

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Parallel Hybrid Evolutionary Algorithms on GPU. In *IEEE Congress on Evolutionary Computation, CEC 2010*, pages 1–8, Proceedings, IEEE, 2010.

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU-based Multi-start Local Search Algorithms. *4th International Learning and Intelligent Optimization Conference, LION 5*, in press, Lecture Notes in Compute Science, Springer, 2011.

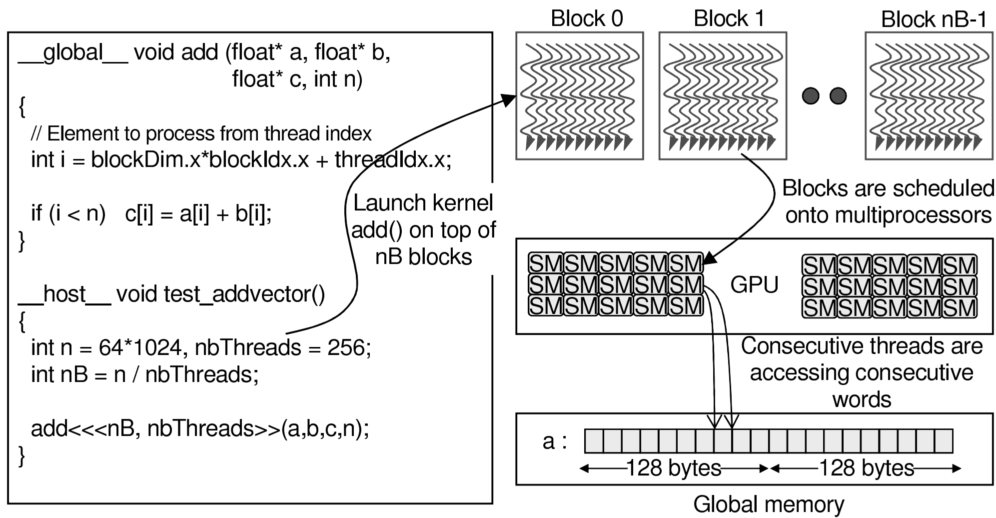


Figure 4.1: An example of kernel execution for vector addition.

4.1 Common Concepts of Memory Management

4.1.1 Memory Coalescing Issues

In the GPU execution model, each block of threads is split into SIMD groups (warps). At any clock cycle, each processor of the multiprocessor selects a half-warp (16 threads) that is ready to execute the same instruction on different data. Global memory is conceptually organized into a sequence of 128-byte segments. The number of memory transactions performed for a half-warp will be the number of segments having the same addresses used by that half-warp. Figure 4.1 illustrates an example of the memory management for a basic vector addition.

For more efficiency, global memory accesses must be coalesced, which means that a memory request performed by consecutive threads in a half-warp is strictly associated with one segment. The requirement is that threads of the same warp must read global memory in an ordered pattern. If per-thread memory accesses for a single half-warp constitute a contiguous range of addresses, accesses will be coalesced into a single memory transaction. In the example of vector addition, memory accesses to the vectors a and b are fully coalesced, since threads with consecutive thread indices access contiguous words.

Otherwise, accessing scattered locations results in memory divergence and requires the processor to produce one memory transaction per thread. The performance penalty for non-coalesced memory accesses varies according to the size of the data structure. In Chapter 3, an insight of this issue has already been given for first improvement-based S-metaheuristics (Section 3.3 and Section 3.4.4).

Indeed, regarding structures in optimization problems, coalescing is sometimes hardly

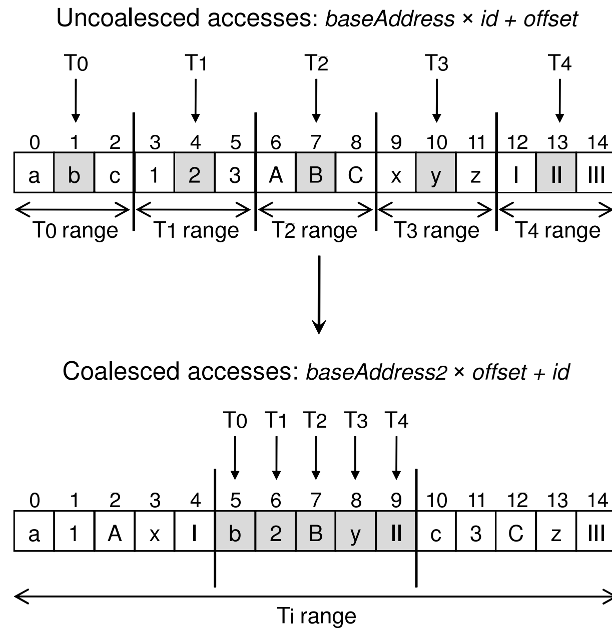


Figure 4.2: An example of coalescing transformation for local structures.

feasible since global memory accesses have a data-dependent unstructured pattern (especially for permutation representation). Many research works on GPU such as [MBL+09, SBPE10, VA10a, VA10b, ZCM08] ignore coalescing transformations of data structures. Therefore, non-coalesced memory accesses imply many memory transactions that lead to a significant performance decrease for these metaheuristics.

4.1.2 Coalescing Transformation

Nevertheless, for some optimization structures which are particular to a given thread, memory coalescing on global memory can be performed. This is typically the case for the data organization of a population in P-metaheuristics; or large local structures used for the evaluation function. Figure 4.2 exhibits an example of a coalescing transformation for local structures. As illustrated in the top of the figure, a natural wrong approach to arrange the elements is to align the different structures one after the other. Thereby, each thread can have access to the elements of its own structure with a logical pattern $baseAddress \times id + offset$. For instance, in the figure, each thread has access to the second element of its structure with $baseAddress = 3$ and $offset = 2$.
Even if this way of organizing the elements on global memory is natural, it is clearly not efficient. Indeed, to get a better global memory performance, memory accesses must constitute a contiguous range of addresses to be coalesced. This is done in the bottom of the figure. In the second approach, the elements of the structures are dispatched such that thread

accesses will be coalesced into a single memory transaction. In the figure, for instance, accessing to the second element is done by using the pattern $baseAddress 2 \times offset + id$. An experimental comparison of the two approaches is conducted in Section 4.3.1. In this case, each solution uses a large private structure which cannot be stored in a local private memory. We will show how this transformation mechanism is well-adapted for large local structures.

4.1.3 Texture Memory

Optimizing the performance of GPU applications often involves optimizing data accesses which includes the appropriate use of the various GPU memory spaces. Some research works on GPU [TF09, SBPE10, VA10a] have considered the memory management of structures specific to combinatorial problems. However, the use of texture memory has never been investigated for dealing with problem inputs.
Hence, in our contributions [1, 7], we have mainly use this memory for reducing memory transactions due to non-coalesced accesses. Indeed, the texture memory provides a surprising aggregation of capabilities including the ability to cache global memory (separate from register, global and shared memory). This memory can be seen as an alternative memory access path that can be bound to regions of the global memory. Each texture unit has some internal memory that buffers data from global memory. Therefore, texture memory can be seen as a relaxed mechanism for the threads to access the global memory, since the coalescing requirements do not apply to texture memory accesses. Thereby, the use of texture memory is well-adapted for metaheuristics for the following reasons:

- Data accesses are frequent in the computation of evaluation methods. Then, using texture memory can provide a high performance improvement by reducing the number of memory transactions.
- Texture memory is a read-only memory i.e. no writing operations can be performed on it. This memory is adapted to metaheuristics since the problem inputs are also read-only values.
- Cached texture data is laid out so as to give the best performance for 1D/2D access patterns. The best performance will be achieved when the threads of a warp read locations that are close together from a spatial locality perspective. Since optimization problem inputs are generally 2D matrices or 1D solution vectors, optimization structures can be bound to texture memory.

The use of textures in place of global memory accesses is a totally mechanical transformation. Details of texture coordinate clamping and filtering is given in [NBGS08, NVI11].

Table 4.1: Kernel memory management. Summary of the different memories used in the evaluation function.

Type of memory	**Optimization structure**
Texture memory	data inputs, solution representation
Global memory	fitnesses structure, large structures
Registers	additional variables
Local memory	small structures
Shared memory	partial fitnesses structure

Notice that, in the Fermi based GPUs, global memory is easier to access. This is due to the relaxation of the coalescing rules and the presence of L1 cache memory. It means that applications developed on GPU get a better global memory performance on this card. Hence, the benefits of the use of texture memory as a data cache are less pronounced.

4.1.4 Memory Management

Table 4.1 summarizes the kernel memory management in accordance with the different structures used in metaheuristics. The inputs of the problem (e.g. a matrix in the traveling salesman problem) are associated with the texture memory. In the case of S-metaheuristics, the solution which generates the neighborhood can also be placed in this memory. The fitnesses structure, which stores the obtained results for each solution, is declared as global memory. Indeed, since only one writing operation per thread is performed at each iteration, this structure is not part of intensive calculations. Declared variables for the computation of each solution evaluation are automatically associated with registers by the compiler. Additional complex structures, which are private to a solution, will reside in local memory. In the case where these structures are too large to fit into local memory, they should be stored in global memory using the coalescing transformation mentioned above. Finally, the shared memory may be used to perform additional reduction operations on the fitness structure to collect the minimal/maximal fitness (see Section 2.2.3 in Chapter 2). All the different memories quoted above have been widely used in the previous experiments presented in Chapter 2 and Chapter 3.

Even if the shared memory has been widely investigated to reduce non-coalesced accesses in regular applications (e.g. [RRS+08, OML+08]), its use may not be well-adapted for the parallel iteration-level model. Due to the limited capacity of each multiprocessor (varying from 16KB to 48KB), data inputs such as matrices cannot be completely stored on shared memory. Thus, the use of shared memory must be considered as a user-managed cache. This implies an explicit effort of code transformation: one has to identify common sub-structures which are likely to be concurrently accessed by SIMD threads of a same block. Unfortunately, such common accesses are not always predictable in evaluation functions

since most of access patterns to data inputs differ from a solution to another (especially for permutation-based problems).

The following transformation intends to show how to take advantage of the shared memory in the case of the quadratic assignment. By considering a S-metaheuristic with a pair-wise exchange operator, one part of the Δ calculation (slight variation) of the evaluation function for a neighbor (i,j) is given by:

$$\Delta_1 = (a_{ii} - a_{jj}) \times (b_{\pi(j)\pi(j)} - b_{\pi(i)\pi(i)}) + (a_{ii} - a_{jj}) \times (b_{\pi(j)\pi(i)} - b_{\pi(i)\pi(j)}) \quad (4.1)$$

$$\Delta = \Delta_1 + \sum_{\substack{k=0 \\ k \neq i \\ k \neq j}}^{n} (a_{ki} - a_{kj}) \times (b_{\pi(k)\pi(j)} - b_{\pi(k)\pi(i)}) + (a_{ik} - a_{jk}) \times (b_{\pi(j)\pi(k)} - b_{\pi(i)\pi(k)}) \quad (4.2)$$

A depth look at (4.2) indicates that a and b sub-matrices involving the variable k might be concurrently accessed in parallel. Therefore, the idea is to associate such sub-structures with the shared memory as a user-managed cache. The equation (4.2) can be transformed into another one:

$$\Delta = \Delta_1 + \sum_{\substack{k=0 \\ k \neq i \\ k \neq j \\ ra \leftarrow row(a,k) \\ rb \leftarrow row(b,\pi(k)) \\ ca \leftarrow col(a,k) \\ cb \leftarrow col(b,\pi(k))}}^{n} (ra_i - ra_j) \times (rb_{\pi(j)} - rb_{\pi(i)}) + (ca_i - ca_j) \times (ca_{\pi(j)} - cb_{\pi(i)}) \quad (4.3)$$

Rows and columns are copied on shared memory at the beginning of each loop iteration via a synchronization mechanism. In this manner, accesses to these structures are performed through this memory in (4.3). Therefore, sub-matrices can benefit from the shared memory since it is a low-latency memory in which non-coalescing accesses are reduced. However, this transformation is problem-dependent, and it might not be applied to some evaluation functions.

We will explore the performance obtained for the use of the different memories in Section 4.3.2. Even if some research works on GPU has considered the association of data structures with the shared memory [TF09, VA10a], we will experimentally show why this

memory is not adapted to the parallelization of metaheuristics on GPU.

4.2 Memory Management in Cooperative Algorithms

One of the main goals of parallelizing a metaheuristic is to reduce the search time. This is a fundamental aspect for some class of problems where there are hard requirements on search time. In this purpose, the iteration-level model concerns the parallel evaluation of solutions. It can be seen an acceleration model in charge of the evaluation of parallel and independent computations. This is typically the case for S-metaheuristics which iteratively improve a single solution. In these algorithms, neighborhood moves do not directly interact with each other.

Things are different when it comes to P-metaheuristics. Indeed, the solutions representing a population may cooperate during the search process. For instance, for evolution-based P-metaheuristics (e.g. evolutionary algorithms or scatter search), the solutions composing the population are selected and reproduced using variation operators. A new solution is constructed from the different attributions of solutions belonging to the current population. Another example concerns P-metaheuristics which participate in the construction of a common and shared structure (e.g. ant colonies or estimation of distribution algorithms). This shared structure will be the main input in generating the new population of solutions, and the previously generated solutions participate in updating such a common structure. Hence, P-metaheuristics provide additional cooperative aspects which are not addressed in S-metaheuristics. Such a cooperation is far more prominent when dealing with multiple metaheuristics simultaneously run in parallel. The exploitation of these cooperative properties for GPU architectures clearly constitutes a challenging issue.

4.2.1 Parallel and Cooperative Model

In the algorithmic-level model, independent or cooperating self-contained metaheuristics are used. It is a problem-independent inter-algorithm parallelization. The different algorithms can be executed either independently or by exchanging information. They may be homogeneous or heterogeneous i.e. configured with the same or different parameters. If the different metaheuristics are independent, the search will be equivalent to the successive execution of metaheuristics in terms of the quality of solutions. However, the cooperative model will alter the behavior of the metaheuristics and enable the improvement of the quality of solutions.

In the (a)synchronous cooperative model (see Figure 4.3), different metaheuristics are simultaneously deployed to cooperate for computing better and robust solutions. They exchange in a(n) (a)synchronous way different information to diversify the search. The

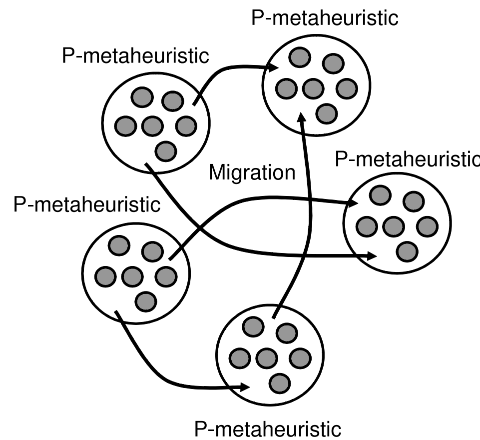


Figure 4.3: The cooperative and parallel model of metaheuristics.

objective is to allow the delay of the global convergence, especially when the metaheuristics are heterogeneous. The migration of solutions follows a policy defined by few parameters:

- **Exchange topology**: The topology specifies for each P-metaheuristic its neighbors with respect to the migration process. In other words, it indicates for each P-metaheuristic the other metaheuristics to which it may send its emigrants, and the ones from which it may receive immigrants). Different well-known topologies are proposed such as ring, mesh, torus, hypercube, etc.

- **Number of emigrants**: This parameter is usually defined either as a percentage of the population size or a fixed number of individuals.

- **Emigrants selection policy**: The selection policy indicates in a deterministic or stochastic way how to select emigrant individuals from the source P-metaheuristic. Different selection policies are defined in the literature: roulette wheel, ranking, stochastic or deterministic tournaments, uniform sampling, etc.

- **Replacement/integration policy**: Symmetrically, the replacement policy defines how to integrate the immigrant individuals in the population. Different replacement strategies may be used including stochastic replacement, tournaments, elitist and pure random replacements.

- **Migration decision criterion**: Migration can be decided either periodically or according to a given criterion. Periodic decision making consists in performing the migration by each P-metaheuristic at a fixed or user defined frequency.

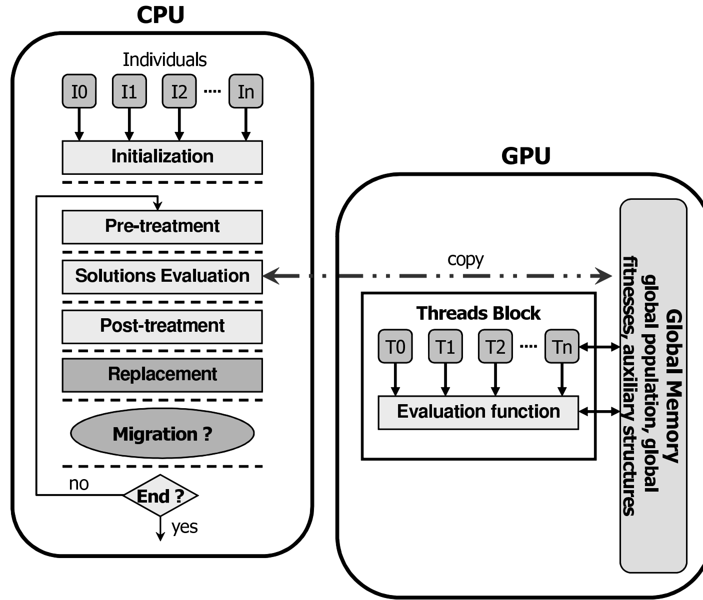


Figure 4.4: Scheme of the parallel evaluation of the population on GPU.

4.2.2 Parallelization Strategies for Cooperative Algorithms

Previous works on P-metaheuristics on GPU have tried hard to implement parallel cooperative algorithms [ZH09, PJS10, MWMA09a, SBPE10]. However, GPU-based cooperative algorithms have never been deeply investigated since no general methods can be outlined from the previous works. In this section, the focus is on the redesign of the cooperative algorithmic-level. To accomplish this, we have designed in [5, 6] three different parallelization schemes allowing a clear separation of the GPU memory hierarchical management concepts.

4.2.2.1 Parallel Evaluation of Populations on GPU

A first scheme for the design and the implementation of cooperative algorithms is based on a combination with the parallel iteration-level model on GPU (parallel evaluation of the solutions). Indeed, in general, evaluating a fitness function for each solution is frequently the most costly operation of a P-metaheuristic. Therefore, in this scheme, task distribution is clearly defined: the CPU manages the whole sequential search process for each cooperative algorithm, and the GPU is dedicated to the parallel evaluation of populations. Figure 4.4 shows the principle of the parallel evaluation of each P-metaheuristic on GPU. The CPU sends a certain number of solutions to be evaluated to the GPU via the global memory, and these solutions will be processed on GPU. Each thread associated with one solution executes the same evaluation function kernel. Finally, results of the evaluation

function are returned back to the host via the global memory.

Regarding the details of the corresponding algorithm (see Algorithm 11), first, memory allocations on GPU are made: data inputs of the problem, populations and their corresponding fitnesses structures must be allocated (lines 3 to 5). Additional solution structures, which are problem-dependent, can also be allocated to facilitate the computation of solution evaluation (line 5). Second, problem data inputs have to be copied onto the GPU (lines 7). These data are read-only structure, and their associated memory are copied only once during all the execution. Third, regarding the main body of the algorithm, the different populations and their associated structures have to be copied at each iteration (lines 11 and 12). Thereafter, the evaluation of solutions is performed in parallel on GPU (lines 13 to 16). Fourth, the fitnesses structures have to be copied to the host CPU (line 17). Then, a specific post-treatment strategy and the replacement of the population are done (lines 18 and 19). Finally, at the end of each generation, a possible migration may be performed on CPU to exchange information between the different P-metaheuristics (line 20). The process is repeated until a stopping criterion is satisfied.

Algorithm 11 Cooperative algorithms template on GPU based on the parallel evaluation of populations

1: Choose initial populations
2: Specific initializations
3: Allocate problem data inputs on GPU device memory
4: Allocate the different populations on GPU memory
5: Allocate fitnesses structures on GPU memory
6: Allocate additional structures on GPU memory
7: Copy problem data inputs on GPU device memory
8: **repeat**
9:   **for** each P-metaheuristic **do**
10:     Specific P-metaheuristic pre-treatment
11:     Copy the different populations on GPU device memory
12:     Copy additional structures on GPU memory
13:     **for** each solution in parallel on GPU **do**
14:       Evaluation of the solution
15:       Insert the resulting fitness into the corresponding fitnesses structure
16:     **end for**
17:     Copy fitnesses structures on CPU hosts memory
18:     Specific P-metaheuristic post-treatment
19:     Replacement of the population
20:   **end for**
21:   Possible Migration between the different P-metaheuristics
22: **until** a stopping criterion satisfied

In this scheme, the GPU is used as a coprocessor in a synchronous manner. However, as

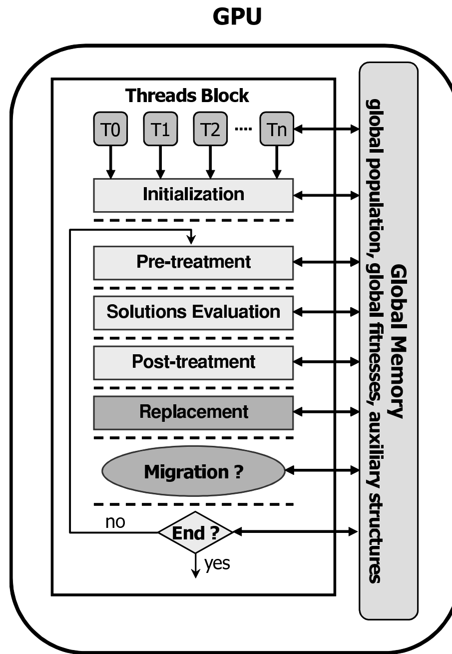


Figure 4.5: Scheme of the full distribution of cooperative algorithms on GPU.

previously seen in Chapter 2, copying operations from CPU to GPU (i.e. population and fitnesses structures) might be a bottleneck and thus can lead to a significant performance decrease.

The main goal of this scheme is to accelerate the search process, and it does not alter the semantics of the algorithm. As a result, the migration policy between the P-metaheuristics remains unchanged in comparison with a classic design on CPU. Since the GPU is used as a coprocessor to evaluate all individuals in parallel, this scheme is intrinsically dedicated to synchronous cooperative algorithms.

4.2.2.2 Full Distribution of Cooperative Algorithms on GPU

A second parallelization strategy is to parallelize the whole process of cooperative algorithms on GPU. This way, the main advantage of this approach is to minimize the data transfers between the host memory and the GPU. Figure 4.5 gives an illustration of each cooperative P-metaheuristic on GPU.

In this scheme, a logical parallelization is to associate each P-metaheuristic with one threads block. One solution is represented by one thread, and each operation (e.g. pre-treatment and post-treatment) is separated by block barriers to ensure the synchronization between the threads. For example, in the case of evolutionary algorithms, synchronizations between the different genetic operators (e.g. selection, crossover or mutation) ensure that

all the individuals of a same block (island) reach the same point. Unless S-metaheuristics which iterate a single solution, the full parallelization of cooperative algorithms on GPU can be naturally achieved since solutions (threads) of each local P-metaheuristic are involved during the whole algorithm process.
Regarding the migration policy, communications are performed via the global memory which stores the global population. This way, each local P-metaheuristic can communicate with any other one according to the given topology.
Algorithm 11 gives more details of the proposed algorithm. In comparison with the previous scheme, all the different allocations and data transfers are performed only once at the beginning of the algorithm (lines 3 to 9). Thereafter, all the standard operations of P-metaheuristics are fully performed to GPU. In other words, they are hard-coded in the GPU kernel and not on CPU. One of the key points to pay attention concerns the synchronizations previously mentioned (lines 12 and lines 17). Such a procedure ensures that operations involving the cooperation of various solutions are valid.

Algorithm 12 Cooperative algorithms template on GPU based on the full distribution

1: Choose initial populations
2: Specific initializations
3: Allocate problem data inputs on GPU device memory
4: Allocate the different populations on GPU memory
5: Allocate fitnesses structures on GPU memory
6: Allocate additional structures on GPU memory
7: Copy problem data inputs on GPU device memory
8: Copy the different populations on GPU device memory
9: Copy additional structures on GPU memory
10: **repeat**
11:     **for** each P-metaheuristic in parallel on GPU **do**
12:         Specific P-metaheuristic pre-treatment with synchronization points
13:         **for** each solution in parallel on GPU **do**
14:             Evaluation of the solution
15:             Insert the resulting fitness into the corresponding fitnesses structure
16:         **end for**
17:         Specific P-metaheuristic post-treatment with synchronization points
18:         Replacement of the population
19:     **end for**
20:     Possible Migration between the different P-metaheuristics
21: **until** a stopping criterion satisfied

In comparison with the previous parallelization scheme, one of the limitations to move the entire algorithm on GPU is the fact that heterogeneous strategies cannot be chosen. Indeed, since threads work in a SIMD fashion, the same parameter configurations and the same different search components (e.g. mutation or crossover in evolutionary algo-

rithms) for each local P-metaheuristic must be applied. Another drawback of this scheme concerns the maximal number of solutions per P-metaheuristic since this latter is limited to the maximal number of threads per block (up to 512 or 1024 according to the GPU architecture). A natural wrong idea to solve this restriction would be to associate one P-metaheuristic with multiple threads blocks. However, it cannot be easily achieved in practice since (1) threads work in an asynchronous manner; (2) threads synchronizations are local to a same block. For instance, for evolutionary algorithms, one can imagine a scenario in which a selection is made on two individuals of a different block, in which one of the two threads has not yet updated its associated fitness value (i.e. one of the two individuals has not yet been evaluated). Such a situation would provoke incoherent results.

4.2.2.3 Full Distribution Using Shared Memory

Regarding the kernel memory management, from a hardware point of view, graphics cards consist of multiprocessors, each with processing units, registers and on-chip memory. Accessing global memory incurs an additional 400 to 600 clock cycles of memory latency. As previously said, since this memory is not cached and its access is slow, one needs to minimize accesses to global memory (read/write operations) and reuse data within the local multiprocessor memories.

To accomplish this, the shared memory, presented in Section 4.1.4, is a fast on-chip memory located on the multiprocessors and shared by threads of each thread block. This memory can be considered as a user-managed cache which can deliver substantial speedups by conserving bandwidth to main memory [NVI11]. Furthermore, since the shared memory is accessible to each threads block, it provides an efficient way for threads to communicate within the same block.

Therefore, a last parallelization strategy is to associate each local P-metaheuristic with a threads block on GPU with the use of the shared memory. An illustration for each P-metaheuristic of this scheme is shown in Figure 4.6. This strategy is similar to the previous one except the fact that local populations and their associated fitnesses are stored in the on-chip shared memory. In this purpose, each solution (thread) of each P-metaheuristic (block) performs the algorithm process (initialization, evaluation, etc.) via the shared memory.

A more refined view is given in Algorithm 11. Basically, the template is similar to the previous one. The main difference which occurs, is all the algorithm operations are not performed on the global memory but on the shared memory. In this purpose, at the beginning of the algorithm, populations are copied to the shared memory (lines 10 to 12). Regarding the migration between P-metaheuristics, since it requires an inter-metaheuristic

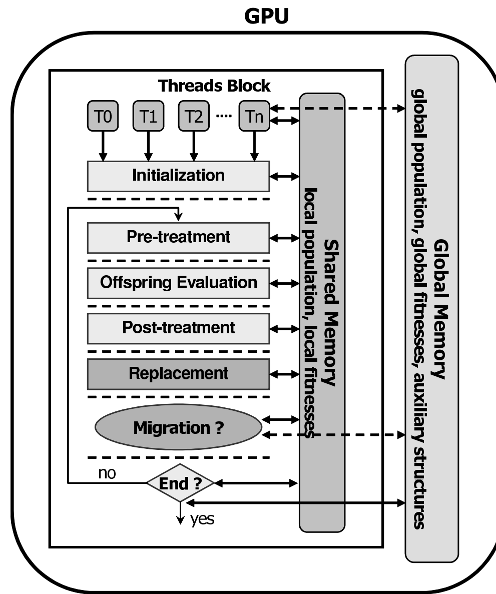


Figure 4.6: Scheme of the full distribution of cooperative algorithms on GPU using shared memory.

communication, copying operations from each local population (shared memory) to the global population (global memory) have to be considered (line 23). Details of this point will be given in the next section.

Even if this scheme can improve the efficiency of cooperative algorithms, it presents a major limitation: since each multiprocessor has a limited capacity of shared memory (varying from 16KB to 48KB), only small problem instances can be dealt with. Indeed, the amount of allocated shared memory per block depends on both the size of the local population and the size of the problem instance. Therefore, a trade-off must be considered between the number of threads per block and the size of the handled problem.

4.2.3 Issues Related to the Fully Distributed Schemes

We have presented three different parallelization strategies of the cooperative algorithmic-level on GPU. The first scheme introduces the parallel evaluation of the global population (iteration-level), and it has been widely investigated in Chapter 2 and in Chapter 3. However, the two other schemes, which involve the entire parallelization of cooperative algorithms on GPU, present many challenging problems related to the GPU execution model. In this section, we propose to re-visit the parameters involving the migration process on GPU.

- **Exchange topology**: Each threads block can be identified using a one-dimensional or two-dimensional index. Therefore, two natural topologies can be defined on GPU:

Algorithm 13 Cooperative algorithms template on GPU based on the full distribution using shared memory

1: Choose initial populations
2: Specific initializations
3: Allocate problem data inputs on GPU device memory
4: Allocate the different populations on GPU memory
5: Allocate fitnesses structures on GPU memory
6: Allocate additional structures on GPU memory
7: Copy problem data inputs on GPU device memory
8: Copy the different populations on GPU device memory
9: Copy additional structures on GPU memory
10: **for** each P-metaheuristic in parallel on GPU **do**
11:     Copy populations from the global memory to the shared memory
12: **end for**
13: **repeat**
14:     **for** each P-metaheuristic in parallel on GPU **do**
15:         Specific P-metaheuristic pre-treatment with synchronization points
16:         **for** each solution in parallel on GPU **do**
17:             Evaluation of the solution
18:             Insert the resulting fitness into the corresponding fitnesses structure
19:         **end for**
20:         Specific P-metaheuristic post-treatment with synchronization points
21:         Replacement of the population
22:     **end for**
23:     Possible Migration between the different P-metaheuristics using shared and global memories
24: **until** a stopping criterion satisfied

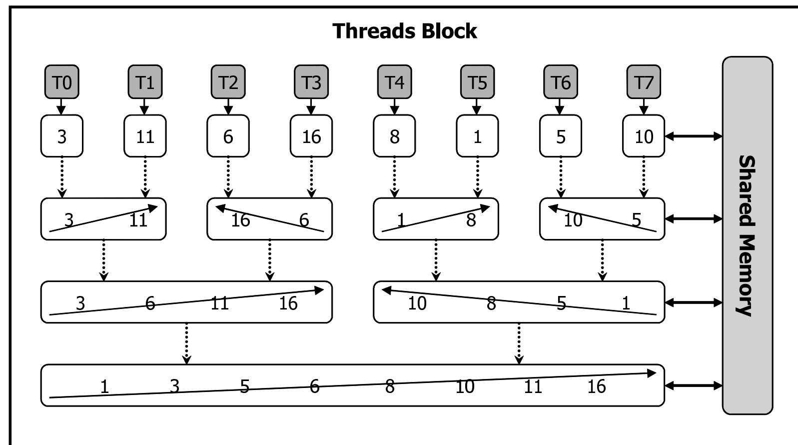


Figure 4.7: Principle of the bitonic sort.

ring (one-dimensional threads block) and 2D toroidal grid (two-dimensional threads block). Finding an efficient mapping between the GPU threads spatial organization and sophisticate topologies might be a difficult issue.

- **Number of emigrants**: This parameter does not represent any pertinent issue except the fact that if the number of emigrants is too high, accesses to the global memory will be more frequent leading to a small performance decrease.

- **Emigrants selection policy**: The selection policy is similar to the traditional selection in evolutionary algorithms. Basically, most of selection operators (e.g. tournament or roulette wheel) operate on unsorted data structures. Therefore, their implementation on GPU is similar to a CPU one. However, some selection operators such as rank-based selections manipulate sorted data structures. As a consequence, a serious issue occurs since sorting on GPU is not as straightforward as on CPU, due to the highly parallel SIMD architecture of GPUs.

  To deal with this issue, one commonly used and efficient sort on GPU is the bitonic sort (see Figure 4.7). Basically, this sort is based on (1) the concept of the bitonic sequence i.e. the composition of two subsequences: one monotonically non-decreasing and the other monotonically non-increasing; (2) merging procedures to create a new bitonic sequence. Despite its complexity of $O(log_2^2 n)$, this sort has a high degree of parallelism. Thereby, it has been stated as one of the fastest sort on GPU for a relatively small number of elements [GZ06]. As a consequence, this sort is particularly well-adapted for cooperative algorithms on GPU since the population size of each P-metaheuristic is also relatively small.

- **Replacement/integration policy**: Symmetrically, the previous strategies of em-

igrants selection on GPU can be similarly applied for the replacement/integration policy. However, in practice, pure elitist replacements can be also considered, in which the worst local individuals are replaced. Since read/write operations on memory are performed in an asynchronous manner, finding the proper minimal/maximal fitnesses of each local population is not straightforward. To deal with this issue, reduction techniques [NVI11] for each thread block must be considered (presented in Chapter 2). Local synchronizations between threads in a same block guarantee to get the minimum/maximum of a given array since threads operate at different memory addresses.

- **Migration decision criterion**: Whatever the migration decision criterion (whether periodically or according to a particular criterion) is, since the order of the execution of threads is undefined, the search process of each P-metaheuristic can be in a different state (e.g. different generation). As a result, the migration on GPU between the different metaheuristics is intrinsically done in an asynchronous manner. Nevertheless, there might be some cases where each P-metaheuristic needs to be in the same state. Thereby, performing a global synchronization between cooperative algorithms on GPU represents a significant issue. Indeed, performing a global synchronization on threads (e.g. by implementing semaphores) would lead to a great loss of performance. To deal with this, implicit synchronizations between the CPU and the GPU must be considered by associating one kernel execution with one generation of the search process. This way, when the execution of one generation in all P-metaheuristics on GPU is accomplished, the command is returned back to the CPU ensuring an implicit global synchronization (see Figure 4.8). Performing such synchronous mechanisms leads to some unavoidable decrease of the performance due to the implicit synchronization and the overhead provoked by kernel calls.

Whether for a synchronous or an asynchronous model, regarding the scheme using shared memory, emigrants must be copied into the global memory to ensure their visibility between the different metaheuristics. Figure 4.9 summarizes the memory management during the migration process through an example of cooperative P-metaheuristics on GPU using the shared memory.

In this example, solutions of each local population are associated with the shared memory, and a ring topology is considered. The two best individuals of each local population (block) are first copied from the shared memory to the global memory. Such a procedure ensures that the solutions to migrate are globally visible between the different local populations. Thereafter, the two worst solutions of each P-metaheuristic are replaced by their corresponding emigrants.

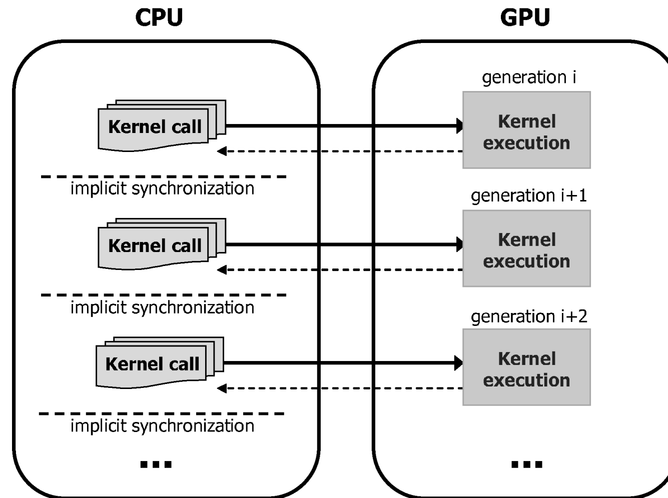


Figure 4.8: Implicit synchronization of threads to ensure that each cooperative P-metaheuristics is in the same state.

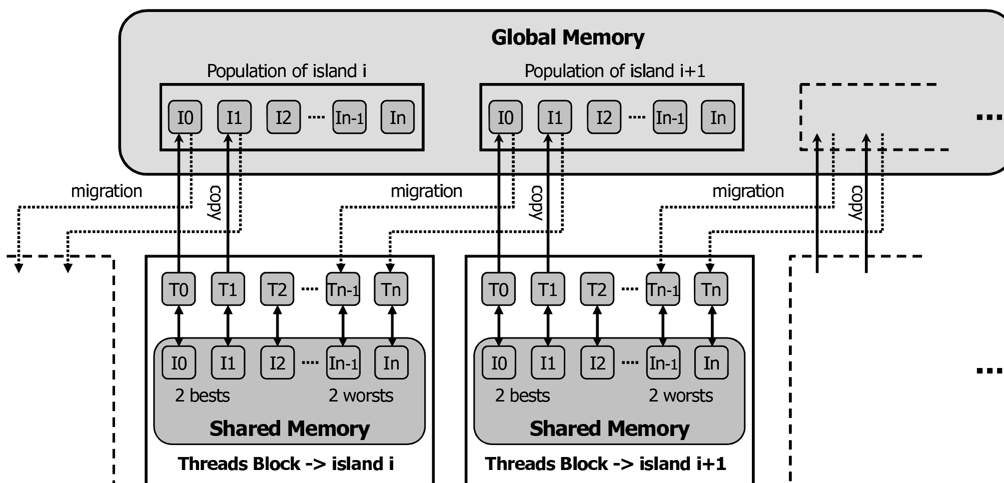


Figure 4.9: Migration between P-metaheuristics on GPU using a ring topology.

Table 4.2: Measures of the benefits of using coalescing accesses to the global memory on the GTX 280. The CUDA Profiler is used for analysing the execution path. The permuted perceptron problem using a neighborhood based on a Hamming distance of two is considered.

Instance	CPU	**Non-coalesced accesses**		**Coalesced accesses**	
		GPU	Warp serializations	GPU	Warp serializations
401-417	403	$64_{\times \mathbf{6.3}}$	3306512379	$14_{\times \mathbf{28.8}}$	435067418
601-617	2049	$249_{\times \mathbf{8.2}}$	6613185750	$51_{\times \mathbf{40.1}}$	612332013
801-817	5410	$665_{\times \mathbf{8.1}}$	11242415771	$128_{\times \mathbf{42.3}}$	1003787122
1001-1017	11075	$1361_{\times \mathbf{8.1}}$	17855601525	$252_{\times \mathbf{43.9}}$	1594250136
1301-1317	25016	$3180_{\times \mathbf{7.9}}$	29759335873	$568_{\times \mathbf{44.1}}$	1859958499

4.3 Performance Evaluation

4.3.1 Coalescing accesses to global memory

As previously said in Section 4.1.2, non-coalesced accesses to the global memory have a remarkable impact on the performance of GPU applications. To achieve the best performance, such an issue has to be dealt with in the case of metaheuristics on GPU.

In Section 4.1.2, two different access patterns have been described to deal with large structures involving the global memory. Even if the first one is natural, this associated performance might be limited because of non-coalesced memory accesses. That is the reason why, the second pattern has always been used for the experiments presented in Chapter 2 and in Chapter 3.

To confirm this point, a next experiment consists in comparing the performance results obtained by the two different access patterns. For doing this, the permuted perceptron problem is considered. In this problem, the neighbor evaluation requires the calculation of a structure called histogram. Since this structure is particular to a neighbor, the local memory is managed to store the histogram. However, for large instances (from $m = 401$ and $n = 417$), the amount of local memory may be not enough to store this structure, and the program will fail at compilation time. As a consequence, in that case, the histogram must be stored on global memory. Table 4.2 presents the obtained results on the GTX 280 for a tabu search with a neighborhood based on a Hamming distance of two (10000 iterations).

In comparison with the second approach, the obtained performance results for the first one are drastically reduced whatever the instance. For the version using non-coalesced accesses, the speed-up obtained from the first approach varies between $\times 4.6$ and $\times 7.0$ whilst the speed-up for the second approach alternates from $\times 28.8$ and $\times 44.1$. An analysis of the execution path of the two algorithms can confirm this observation. One can observe that the number of warp serializations for the version using non-coalesced accesses is be-

tween seven and sixteen times more important than for its counterpart. Indeed, a threads divergence caused by non-coalesced accesses leads to many memory accesses that have to be serialized, increasing the instructions to be executed. This explains the difference performance between the two versions. A conclusion of this experiment indicates that memory coalescing applied on local structures is a must to obtain the best performance.

4.3.2 Memory Associations of Optimization Problems

Optimizing the performance of metaheuristics on GPU requires optimizing data accesses. It involves the appropriate use of the different GPU memories. As previously seen in Section 4.1.4, the use of texture memory is a solution for reducing memory transactions due to non-coalesced accesses. It is essentially as a cache to the global memory in association with data inputs of a given combinatorial problem. In a same manner, the shared memory might also be used as a user-manage cache to reduce non-coalesced accesses. Such a memory management does not appear in problems in which there are no data inputs (e.g. the Weierstrass function).

The following experiment consists in comparing the performance of a same tabu search (10000 iterations) with different memory associations for the quadratic assignment problem. The first version is a pure tabu search on GPU using only the global memory. In the second one, the texture memory provides an alternative memory access path that is bound to regions of the global memory in regards with the different data inputs (used in the previous experiments). The last implementation uses the shared memory to deal with data inputs. For doing this, data from the global memory need to be copied to the shared memory.

Table 4.3 reports the obtained results for these different implementations. In a general manner, the shared memory version ($\mathrm{GPU_{Sh}}$) obtains better performance results than the basic GPU version without memory optimization ($\mathrm{GPU_{Glo}}$). The acceleration factors in comparison with a single-core on CPU diversify between $\times 0.7$ and $\times 16.1$ for $\mathrm{GPU_{Sh}}$ against $\times 0.5$ and $\times 15.7S$ for the standard version.

Nevertheless, such an improvement does not occur in regards with the texture version. Indeed, for the two first configurations, the shared memory version is clearly outperformed by the texture one. This is due to the extra cost of data copies from global memory to shared memory (thus extra non-coalesced accesses) including local synchronizations for each loop iteration. In the two last configurations, the gap is less relevant since global memory is easier to access due to the relaxation of the coalescing rules.

A conclusion from this experiment indicates that the use of shared memory gives further performance improvement. However, on the one hand, an effort of code rewriting has to be provided. Furthermore, it is not clear that such a transformation is always feasible. On the

Table 4.3: Measures in terms of efficiency of three different versions (global, texture, and shared memories). The quadratic assignment problem using a pair-wise-exchange neighborhood is considered.

Instance	Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores			Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores		
	GPU_{Glo}	GPU_{Tex}	GPU_{Sh}	GPU_{Glo}	GPU_{Tex}	GPU_{Sh}
tai30a	$4.2_{\times 0.5}$	$1.3_{\times \mathbf{1.8}}$	$3.1_{\times 0.7}$	$2.3_{\times 0.8}$	$1.0_{\times \mathbf{1.9}}$	$2.1_{\times 0.9}$
tai35a	$6.5_{\times 0.5}$	$1.6_{\times \mathbf{2.1}}$	$5.0_{\times 0.7}$	$2.9_{\times 0.9}$	$1.2_{\times \mathbf{2.3}}$	$2.8_{\times 1.0}$
tai40a	$9.7_{\times 0.5}$	$1.8_{\times \mathbf{2.6}}$	$5.5_{\times 0.9}$	$3.7_{\times \mathbf{1.1}}$	$1.5_{\times \mathbf{2.9}}$	$3.5_{\times \mathbf{1.2}}$
tai50a	$16_{\times 0.6}$	$3.0_{\times \mathbf{3.2}}$	$8.9_{\times \mathbf{1.1}}$	$5.7_{\times \mathbf{1.4}}$	$1.8_{\times \mathbf{4.6}}$	$5.0_{\times \mathbf{1.7}}$
tai60a	$28_{\times 0.6}$	$4.9_{\times \mathbf{3.4}}$	$13_{\times \mathbf{1.3}}$	$8.4_{\times \mathbf{1.6}}$	$2.0_{\times \mathbf{7.1}}$	$6.1_{\times \mathbf{2.3}}$
tai80a	$63_{\times 0.7}$	$10_{\times \mathbf{4.2}}$	$41_{\times \mathbf{1.5}}$	$19_{\times \mathbf{1.9}}$	$4.5_{\times \mathbf{8.1}}$	$9.6_{\times \mathbf{3.7}}$
tai100a	$139_{\times 0.8}$	$19_{\times \mathbf{5.6}}$	$67_{\times \mathbf{1.6}}$	$33_{\times \mathbf{2.3}}$	$8.7_{\times \mathbf{8.8}}$	$15.6_{\times \mathbf{4.8}}$

Instance	Intel Xeon E5450 GeForce GTX 280 240 GPU cores			Xeon E5620 Tesla M2050 448 GPU cores		
	GPU_{Glo}	GPU_{Tex}	GPU_{Sh}	GPU_{Glo}	GPU_{Tex}	GPU_{Sh}
tai30a	$1.1_{\times \mathbf{1.4}}$	$0.8_{\times \mathbf{2.0}}$	$1.0_{\times \mathbf{1.7}}$	$0.5_{\times \mathbf{2.2}}$	$0.4_{\times \mathbf{2.8}}$	$0.5_{\times \mathbf{2.4}}$
tai35a	$1.2_{\times \mathbf{1.9}}$	$0.9_{\times \mathbf{2.6}}$	$1.1_{\times \mathbf{2.2}}$	$0.6_{\times \mathbf{2.4}}$	$0.5_{\times \mathbf{3.8}}$	$0.6_{\times \mathbf{2.7}}$
tai40a	$1.3_{\times \mathbf{2.7}}$	$1.1_{\times \mathbf{3.3}}$	$1.2_{\times \mathbf{3.0}}$	$0.7_{\times \mathbf{3.2}}$	$0.5_{\times \mathbf{4.4}}$	$0.6_{\times \mathbf{3.8}}$
tai50a	$1.7_{\times \mathbf{4.1}}$	$1.3_{\times \mathbf{5.3}}$	$1.5_{\times \mathbf{4.7}}$	$0.8_{\times \mathbf{5.4}}$	$0.6_{\times \mathbf{7.2}}$	$0.7_{\times \mathbf{6.1}}$
tai60a	$2.0_{\times \mathbf{5.6}}$	$1.6_{\times \mathbf{7.3}}$	$1.7_{\times \mathbf{6.5}}$	$1.1_{\times \mathbf{8.4}}$	$0.9_{\times \mathbf{10.2}}$	$1.0_{\times \mathbf{9.1}}$
tai80a	$3.2_{\times \mathbf{9.1}}$	$2.8_{\times \mathbf{10.8}}$	$2.9_{\times \mathbf{10.3}}$	$1.9_{\times \mathbf{12.4}}$	$1.7_{\times \mathbf{13.5}}$	$1.8_{\times \mathbf{12.7}}$
tai100a	$5.5_{\times \mathbf{10.9}}$	$3.7_{\times \mathbf{16.5}}$	$4.1_{\times \mathbf{14.6}}$	$3.1_{\times \mathbf{15.7}}$	$2.6_{\times \mathbf{18.6}}$	$3.0_{\times \mathbf{16.1}}$

other hand, performance results for the quadratic assignment problem indicate that this version is dominated by the texture one especially for low-graphic cards configurations. Therefore, it seems that the stand-alone use of shared memory is not well-adapted for the parallel iteration-level. That is the reason why, all the previous experiments on S-metaheuristics have been performed by using the texture memory.

4.4 Performance of Cooperative Algorithms

The exploitation of the threads spatial organization and the different available memories allows the design of cooperative algorithms on GPU. In Section 4.2, we have introduced a guideline for redesign of the algorithmic-level parallel model on GPU.

To validate the performance of the proposed approaches, a cooperative island model for evolutionary algorithms has been implemented on GPU. In this purpose, the Weierstrass-Mandelbrot functions have been considered.

4.4.1 Configuration

The used configuration for the experiments is an Intel Xeon 8 cores 2.4 Ghz with a NVIDIA GTX 280 card. For each experiment, different implementations of the island model are given: a standalone single-core implementation of the island model on CPU (CPU), the synchronous island model using the parallel evaluation of populations on GPU (CPU+GPU), the asynchronous and fully distributed island model on GPU (GPU), the synchronous version (SGPU) and their associated versions using the shared memory (GPUSh and SGPUSh).

Regarding the full implementation for evolutionary algorithms on GPU, previous techniques presented in Section 4.2.3 are applied to the selection and the replacement. Regarding the components which are dependent of the problem (i.e. initialization, evaluation function and variation operators), they do not represent specific issues related to the GPU kernel execution. From an implementation point of view, the focus is on the management of random numbers. To achieve this, efficient techniques are provided in several books such as [NVI10] to implement random generators on GPU (e.g. Gaussian or Mersenne Twister generators).

The operators used in the evolutionary process for the implementations are the following: the crossover is a standard two-point crossover (crossover rate fixed to 80%) which generates two offspring, the mutation consists in changing randomly one gene with a real value taken in $[-1; 1]$ (with a mutation rate of 30%), the selection is a deterministic tournament (tournament size fixed to the block size divided by four), the replacement is a $(\mu + \lambda)$ replacement and the number of generations has been fixed to 100. Regarding the migration policy, a ring topology has been chosen, a deterministic tournament has been performed for both emigrants selection and migration replacement (tournament size fixed to the block size divided by four), the migration rate is equal to the number of local individuals divided by four and the migration frequency is set to 10 generations.

4.4.2 Measures in Terms of Efficiency

The objective of the following experiments is to assess the impact of the different GPU-based implementations in terms of efficiency. Only execution times (in seconds) and acceleration factors (compared to a single CPU core) are reported. The average time has been measured in seconds for 50 runs. The first experiment consists in varying the dimension of the Weierstrass function. The obtained results are reported in Table 4.4.

As long as the size of the dimension increases, each GPU version gives some outstanding accelerations compared to a CPU version (up to $\times 1757$ for the GPUSh version). The use of the shared memory provides a way to accelerate the search process even if the GPU

Table 4.4: Measures in terms of efficiency of the island model for evolutionary algorithms on GPU. The Weierstrass function is considered. The number of individuals per island is fixed to 128 and the global population to 8192 (64 islands).

	CPU	**CPU+GPU**		**GPU**		**GPUSh**		**SGPU**		**SGPUSh**	
instance	**time**	**time**	**acc.**	**time**	**acc.**	**time**	**acc.**	**time**	**acc.**	**time**	**acc.**
1	23	0.92	×25	0.16	×143	0.04	×845	0.36	×63	0.06	×375
2	43	0.94	×46	0.16	×268	0.03	×1150	0.36	×119	0.08	×511
3	64	0.95	×67	0.17	×375	0.05	×1365	0.38	×167	0.11	×607
4	85	0.97	×87	0.21	×403	0.06	×1442	0.47	×179	0.13	×641
5	105	1.00	×105	0.22	×479	0.07	×1579	0.50	×213	0.15	×702
6	127	1.02	×125	0.24	×519	0.08	×1639	0.55	×231	0.17	×728
7	148	1.04	×142	0.28	×529	0.09	×1659	0.63	×235	0.20	×737
8	168	1.09	×154	0.30	×554	0.10	×1684	0.68	×246	0.23	×748
9	190	1.19	×159	0.31	×610	0.11	×1736	0.70	×271	0.25	×772
10	211	1.28	×165	0.33	×639	0.12	×1757	0.74	×284	0.27	×781
11	231	1.36	×170	0.35	×666	–	–	0.79	×293	–	–

version is already impressive. However, due to its limited capacity, bigger instances such as a dimension of 11 cannot be handled in any shared memory versions.

Regarding the fully distributed synchronous versions, since implicit synchronizations are performed, a certain reduction of acceleration factors (from ×63 to ×293 for the SGPU version) can be observed in comparison with their associated asynchronous versions. Nevertheless, the acceleration factors are still outstanding. For the scheme of the parallel evaluation populations (CPU+GPU version), the speed-ups are less remarkable even if they remain significant (from ×25 to ×170). This is explained by the important number of data transfers between the CPU and the GPU.

A conclusion of the first experiment indicates that the full distribution of the search process on GPU and its own memory management deliver high performance results. However, the importance of these results should be minimized due to the nature of the Weierstrass function, which is a compute-bound application. When considering a standard application which is both compute and memory bound, things are different. Table 4.5 reports the obtained results for the quadratic assignment problem using the same parameters used before.

In a general manner, one can see that the obtained acceleration factors are more similar than those ones presented in the previous chapters. The acceleration factors grow with the instance size. They alternate from ×1.3 to ×15.2 according to the different parallelization schemes. Regarding the versions using the shared memory, unless the Weierstrass function, the performance improvements are less pronounced since accesses to matrix structures (global and texture memories) are prominent in the entire algorithm. Further-

Table 4.5: Measures in terms of efficiency of the island model for evolutionary algorithms on GPU. The quadratic function is considered. The number of individuals per island is fixed to 128 and the global population to 8192 (64 islands)

.

	CPU	CPU+GPU		GPU		GPUSh		SGPU		SGPUSh	
instance	**time**	**time**	**acc.**	**time**	**acc.**	**time**	**acc.**	**time**	**acc.**	**time**	**acc.**
tai30a	8.4	6.5	×1.3	1.3	×6.2	1.2	×7.1	2.3	×3.6	2.0	×4.2
tai35a	11.1	6.9	×1.6	1.4	×7.8	1.3	×8.5	2.5	×4.4	2.2	×5.1
tai40a	15	7.1	×2.1	1.6	×9.5	1.5	×10.2	2.8	×5.4	2.4	×6.3
tai50a	22	7.9	×2.8	2.0	×11.2	1.8	×12.3	3.5	×6.3	3.1	×7.0
tai60a	31	8.9	×3.5	2.4	×12.8	2.3	×13.6	4.2	×7.4	3.6	×8.5
tai80a	60	11.5	×5.2	4.3	×13.9	–	–	7.1	×8.5	–	–
tai100a	101	13	×7.7	6.6	×15.2	–	–	10.4	×9.7	–	–

more, the same limitation of the problem size occurs from the instance tai80a, in which the application could not be executed.

After evaluating the performance of the island model for evolutionary algorithms on GPU, another experiment consists in measuring the scalability of our approaches. For doing this, varying the number of islands with extreme values is needed in order to determine the application scalability. Results of this experiment are reported in Table 4.6 for the Weierstrass function.

Regarding each fully distributed version, for a small number of islands (i.e. one or two islands), the acceleration factor is significant but not spectacular (from ×7 to ×51). This is explained by the fact that since the global population is relatively small (less than 1024 threads), the number of threads per block is not enough to fully cover the memory access latency. This is not the same situation when considering more islands. Indeed, the speed-up grows accordingly with the increase of the number of islands and remains impressive (up to ×2074 for GPUSh).

Regarding the CPU+GPU version, speed-ups for one or two islands are more important than for fully distributed versions. Indeed, since only the evaluation of the population is distributed on GPU, fewer registers are allocated for each thread. As a result, this version benefits from a better occupancy of the multiprocessors for a small number of islands. GPU keeps accelerating the process with the islands increase until reaching a peak performance of ×165 for 64 islands. Thereafter, the acceleration factor decreases with the augmentation of the number of islands. Indeed, for each parallel evaluation of the population, the amount of data transfers is proportional to the number of individuals (e.g. 524288 threads for 4096 islands). Thus, from a certain number of islands, the time dedicated to copy operations becomes significant, leading clearly to a decrease of the performance.

Table 4.6: Measures in terms of scalability of the island model for evolutionary algorithms on GPU. The Weierstrass function is considered. The dimension of the problem is fixed to 2 and the number of individuals per island to 128.

	CPU	**CPU+GPU**		**GPU**		**GPUSh**		**SGPU**		**SGPUSh**	
islands	**time**	**time**	**acc.**	**time**	**acc.**	**time**	**acc.**	**time**	**acc.**	**time**	**acc.**
1	3	0.10	×33	0.20	×17	0.12	×27	0.45	×7	0.27	×12
2	7	0.12	×55	0.20	×33	0.13	×51	0.45	×15	0.29	×23
4	13	0.15	×89	0.20	×65	0.13	×104	0.45	×29	0.29	×46
8	26	0.19	×139	0.20	×132	0.13	×207	0.45	×59	0.29	×92
16	53	0.34	×154	0.21	×256	0.13	×403	0.46	×114	0.29	×179
32	106	0.66	×160	0.26	×406	0.13	×828	0.59	×180	0.29	×368
64	211	1.28	×165	0.33	×644	0.14	×1560	0.74	×286	0.30	×693
128	422	2.68	×158	0.45	×939	0.26	×1596	1.01	×417	0.60	×709
256	845	5.61	×151	0.69	×1222	0.50	×1677	1.56	×543	1.13	×746
512	1692	11.81	×143	1.24	×1365	1.00	×1691	2.79	×607	2.25	×752
1024	3382	25.72	×132	–	–	1.70	×1990	–	–	3.82	×885
2048	6781	53.23	×127	–	–	3.27	×2074	–	–	7.36	×922
4096	13585	143.71	×95	–	–	–	–	–	–	–	–

Regarding the scalability of the fully distributed versions, from a certain number of islands, the GPU failed to execute the program because of the hardware register limitation. For instance, for a number of 1024 islands (131072 threads), the SGPU implementation could not be executed. In the GPUSh and the SGPUSh versions, since the shared memory is used conducting to fewer registers, this limit is reached for a larger number of 4096 islands. The CPU+GPU version provides a higher scalability since fewer registers are allocated (only the evaluation kernel is executed on GPU).

In the previous experiments, the number of individuals is fixed to 128. Another experiment consists in varying the number of individuals per island i.e. the number of threads per block. In this purpose, it will allow to determine the effect of this parameter on the global performance. Table 4.7 reports the obtained results. Regarding the execution time of each version, it varies accordingly to the number of threads per block. In general, best performances are obtained for 16, 32 or 64 threads per block. Indeed, the measured results depend on the multiprocessor occupancy of a GPU. This latter varies according to the number of threads used in a kernel, the amount of registers and shared memory used. For instance, the CUDA occupancy calculator [NVI11] is an efficient tool to adjust the different configurations. Another observation that can be made concerns the threads limitation for the shared memory versions. Indeed, for 768 or 1024 threads per block, the amount of allocated shared memory exceeds the memory capacity of each multiprocessor (16KB). Therefore, for these versions, a trade-off must be found between the dimension of the problem and the number of individuals per island.

Table 4.7: Measures in terms of efficiency by varying the number of individuals per island. The dimension of the problem is fixed to 2 and the global population to 8192.

individuals	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**768**	**1024**
CPU	41.8	42.0	42.1	42.2	42.4	42.9	43.4	44.8	49.8	55.2
CPU+GPU	0.27	0.22	0.24	0.34	0.54	0.96	1.72	2.90	3.23	4.12
GPU	0.18	0.17	0.16	0.16	0.15	0.16	0.19	0.25	0.28	0.33
GPUSh	0.05	0.03	0.02	0.02	0.02	0.04	0.14	0.17	–	–
SGPU	0.41	0.38	0.36	0.36	0.34	0.36	0.42	0.55	0.63	0.74
SGPUSh	0.10	0.07	0.05	0.05	0.05	0.09	0.30	0.37	–	–

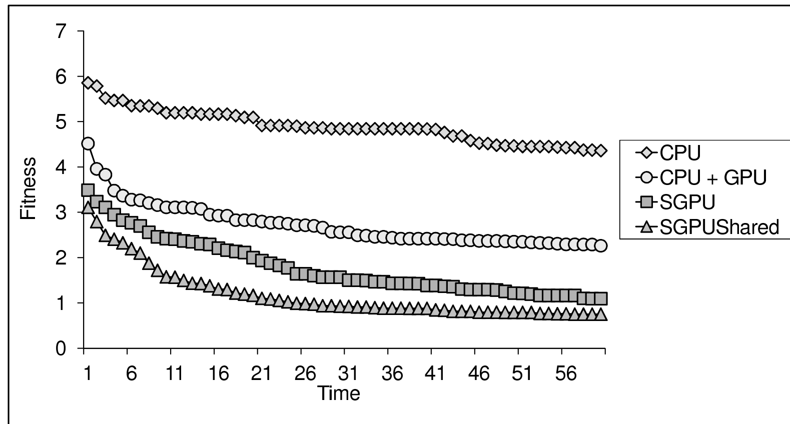


Figure 4.10: Measures in terms of effectiveness of the different island models during the first minute. The dimension of the problem is fixed to 10, the global population is set to 8192 and the number of individuals per island to 128.

4.4.3 Measures in Terms of Effectiveness

After measuring the efficiency of our approaches, the last experiment consists in evaluating the quality of the obtained solutions. To accomplish this in a fair manner, only GPU synchronous versions are considered to have the same algorithm semantic. Figure 4.10 reports the evolution of the average of the best fitness values during the first minute. In agreement with the previous obtained results, the quality of the solutions differs accordingly to the employed scheme. Indeed, for instance, the best results are obtained with the full distribution of the island model on GPU using shared memory. Precise measurements of the first second are not visible on the figure (populations have the same initializations), but the evolution of the fitness for each GPU version decreases drastically during the first second. The evolution of the fitness keeps decreasing with time but with a rather slow convergence, due to a high migration rate (25%).

Conclusion

In this chapter, the focus has been made on the memory management in metaheuristics on GPU. The comprehension of the hierarchical organization of the different memories available on GPU architectures is the key issue to develop efficient parallel metaheuristics. In this purpose, we have investigated the complete redesign of the algorithmic-level model for parallel and cooperative algorithms.

- **Management of data structures**. Through an example, we have shown that coalescing transformations are essential for getting more global memory efficiency. Furthermore, optimizing the performance of GPU-based metaheuristics involves the appropriate use of various GPU memory spaces. The use of texture memory has been revealed as a solution for reducing memory transactions due to non-coalesced accesses. When dealing with optimization problems with data inputs, its use may be more appropriated than the shared memory.

- **Parallel and cooperative metaheuristics**. Cooperative properties available in P-metaheuristics may be exploited to design efficient cooperative algorithms. Thereby, we have established three parallelization schemes for the algorithmic-level on GPU: the parallel evaluation of populations on GPU, the full distribution on GPU, and the entire distribution on GPU using the shared memory. For achieving this, all the mechanisms for the migration of solutions between the different populations have been revisited according to the hierarchical organization of memories. The obtained results from the experiments suggest that such parallelization strategies are fully efficient.

Chapter 5

Extension of the ParadisEO Framework for GPU-based Metaheuristics

Previous chapters have shown that parallel combinatorial optimization on GPU is not straightforward and requires a huge effort at design as well as at implementation level. Indeed, the design of GPU-aware metaheuristics often involves the cost of a sometimes painful apprenticeship of parallelization techniques and GPU computing technologies. In order to free from such burden those who are unfamiliar with those advanced features, ParadisEO integrates the up-to-date parallelization techniques and allow their transparent exploitation and deployment on COWs and computational grids.
First, a brief overview of the ParadisEO framework will be done. Then, we will extend ParadisEO to deal with GPU accelerators. The challenges and contributions consist in making the GPU as transparent as possible for the user minimizing his or her involvement in its management. In this purpose, we offer solutions to this challenge as an extension of the ParadisEO framework.

Contents

5.1 The ParadisEO Framework 117
5.1.1 Motivations and Goals 117
5.1.2 Presentation of the Framework 117
5.2 GPU-enabled ParadisEO 118
5.2.1 Architecture of ParadisEO-GPU 119
5.2.2 ParadisEO-GPU Components 120
5.2.3 A Case Study: Parallel Evaluation of a Neighborhood 122
5.2.4 Automatic Construction of the Mapping Function 124
5.3 Performance Evaluation 126
5.3.1 Experimentation with ParadisEO-GPU 126

Main publications related to this chapter

Nouredine Melab, Thé Van Luong, K. Boufaras, and El-Ghazali Talbi. Towards ParadisEO-MO-GPU: A Framework for GPU-based Local Search Metaheuristics. *11th International Work-Conference on Artificial Neural Networks, IWANN 2011*, pages 401–408, volume 6691 of Lecture Notes in Computer Science, Springer, 2011.

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Neighborhood Structures for GPU-based Local Search Algorithms. *Parallel Processing Letters*, 20(4):307–324, 2010.

Submitted article

Nouredine Melab, Thé Van Luong, K. Boufaras, and El-Ghazali Talbi. ParadisEO-MO-GPU: A Framework for GPU-based Local Search Metaheuristics. *Journal of Heuristics*, submitted.

5.1 The ParadisEO Framework

5.1.1 Motivations and Goals

A framework is usually designed to be operated by the maximum number of users. Its exploitation can only be successful if a significant number of criteria is satisfied. Therefore, the main objectives of the ParadisEO framework are the following ones [CMT04]:

- *Maximum design and code reuse.* The framework must provide a full architecture design for using metaheuristics. In this purpose, the programmer may redo as little code as possible. This objective requires a clear and maximal conceptual separation between the different methods and problems to be solved. Therefore, the user might develop only the minimal code specific to the problem at hand.

- *Flexibility and adaptability.* It must be possible for the user to easily add new features or to modify existing ones without involving other components. Moreover, as existing problems evolve and new ones appear, the framework components must be specialized and adapted to the general demand.

- *Utility.* The framework must allow the user to cover a wide range of metaheuristics, problems, parallel distributed models, hybridization mechanisms, etc.

- *Transparent and easy access to performance and robustness.* As optimization applications are often time-consuming, the performance issue is crucial. The use of parallelism is necessary to achieve high performance execution. In order to facilitate its use, parallel techniques need to be implemented in a transparent manner for the user. Moreover, the execution of the algorithms must be robust to guarantee the reliability and the quality of the obtained results. In this purpose, the hybridization mechanism allows to obtain robust and better solutions.

- *Portability.* In order to satisfy a large number of users, the framework must support different hardware architectures and operating systems.

5.1.2 Presentation of the Framework

ParadisEO[1] [CMT04] is a white-box object-oriented software framework dedicated to the flexible design of metaheuristics for optimization problems. Based on EO[2] (evolving objects) [KGRS01], this template-based C++ library is portable across different operating systems.

[1] http://paradiseo.gforge.inria.fr
[2] http://eodev.sourceforge.net

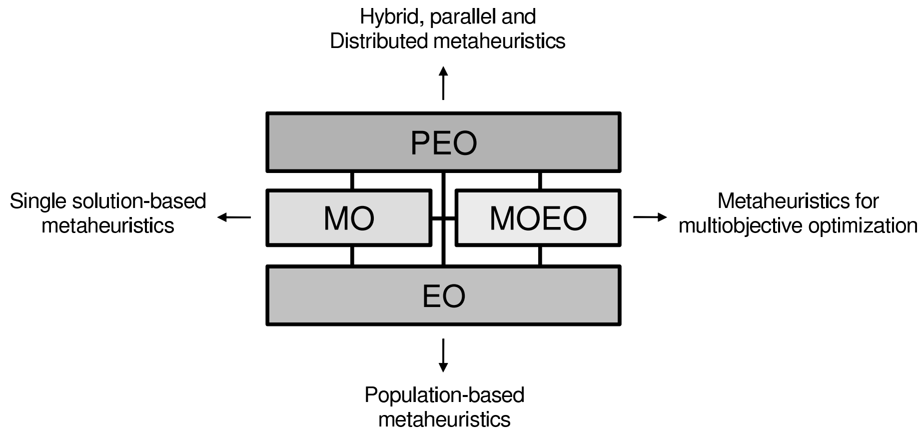


Figure 5.1: Main modules of ParadisEO.

The framework is composed of four modules that constitute a global framework. Figure 5.1 illustrates the different modules available in ParadisEO. Each module is based on a clear conceptual separation of the metaheuristics from the problems they are intended to solve. Such a separation provides a maximum code and design reuse to the user. The first module ParadisEO-EO provides a broad range of classes for the development of P-metaheuristics, including evolutionary algorithms and particle swarm optimization techniques. Second, ParadisEO-MO contains a set of tools for S-metaheuristics such as hill climbing, simulated annealing, or tabu search. Third, ParadisEO-MOEO is specifically dedicated to the reusable design of metaheuristics for multiobjective optimization. Finally, ParadisEO-PEO provides a powerful set of classes for the design of parallel and distributed metaheuristics: parallel evaluation of solutions, parallel evaluation of the objective function and parallel cooperative algorithms.
ParadisEO is one of the rare frameworks that provide the most common parallel and distributed models. These models are portable on distributed-memory machines and shared-memory multiprocessors as they are implemented using standard libraries such as MPI and PThreads. The models can be exploited in a transparent way. One has just to instantiate their associated ParadisEO components. The ParadisEO-PEO module is an open source framework originally intended to the design and deployment of parallel hybrid local search metaheuristics on dedicated clusters and networks of workstations, shared-memory machines and computational grids.

5.2 GPU-enabled ParadisEO

Two framework tentatives have been proposed in [MBL+09, SBPE10]. However, the first one is limited to an application of genetic algorithms for continuous problems, while the

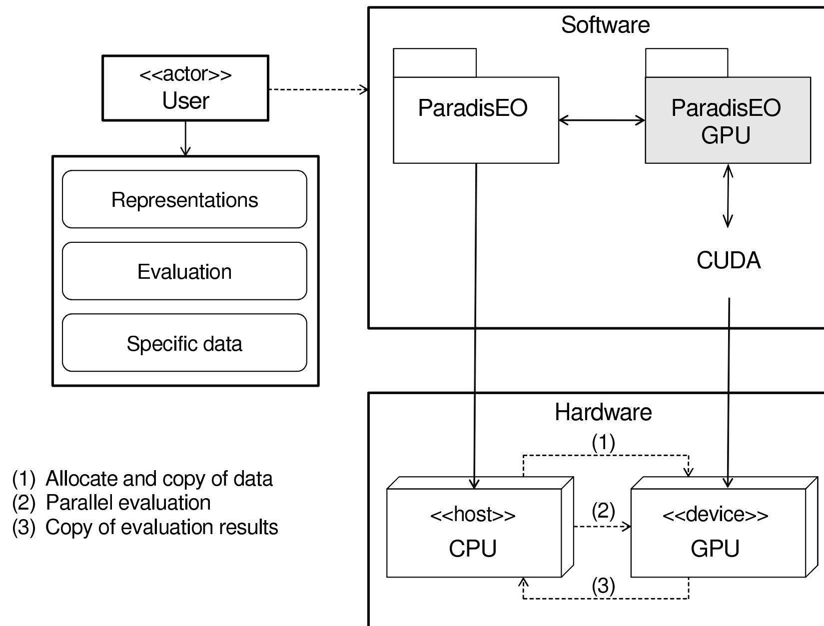


Figure 5.2: A layered architecture of ParadisEO-GPU.

second one remains in a design step. To the best of our knowledge, there does not exist any software framework for GPU-based metaheuristics applied to combinatorial optimization. Regarding the different parallel implementations of ParadisEO, in [CMT04], the first version of ParadisEO-PEO was dedicated to the reusable design of parallel and distributed metaheuristics for only dedicated parallel hardware platforms. Later, the framework was dedicated in [MCT06] to dynamic and heterogeneous large-scale environments using Condor-MW middleware and in [TMDT07] to computational grids using Globus. In this section, we will present a step towards a ParadisEO framework for the reusable design and implementation of the GPU-based parallel metaheuristics.

5.2.1 Architecture of ParadisEO-GPU

ParadisEO-GPU is an extension of the ParadisEO-PEO module, which is a coupling between ParadisEO and CUDA for the design and implementation of reusable metaheuristics on GPU. It is composed by a set of new C++ abstract and predefined classes that enables an easy and transparent development of metaheuristics on GPU accelerators. The actual available components concern the iteration-level parallel model (parallel evaluation of solutions). From an implementation point of view, this model presents many generic concepts that can be parallelized. The architecture of ParadisEO-GPU is layered as illustrated in Figure 5.2.

The user layer indicates the different problem-dependent components that must be defined:

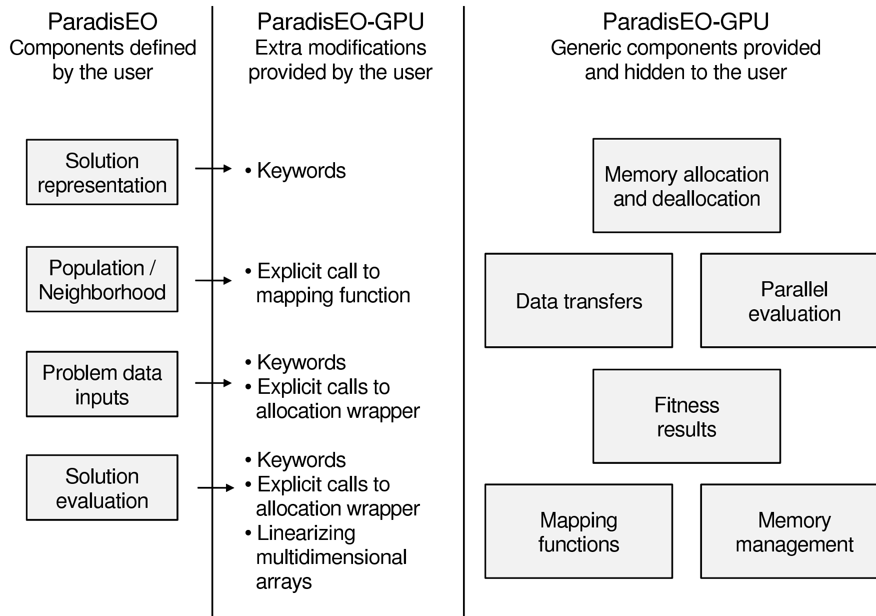


Figure 5.3: The major components of ParadisEO-GPU.

input data, the evaluation function and representations. The software layer supplies the ParadisEO components including optimization solvers embedding metaheuristics. The ParadisEO-GPU module provides a CUDA interface allowing the transparent interaction with the hardware layer. The hardware layer supplies the different transparent tools provided by ParadisEO-GPU such as the allocation and copy of data or the parallel generation and evaluation of the considered set of solutions. In addition, the platform offers predefined structures (e.g. neighborhood structures) and mapping wrappers adapted to hardware characteristics to deal with binary and permutation problems.
The layered architecture of ParadisEO-GPU has been designed in such a way that the user does not need to build his or her own CUDA code for the specific problem to be solved. Indeed, ParadisEO-GPU provides facilities for automatic execution of metaheuristics on GPU. The only thing that must be user-managed is the different components described in the user level quoted above.

5.2.2 ParadisEO-GPU Components

Figure 5.3 illustrates the major components of the platform. The advantage of the decomposition into components is to separate the components that must be defined by the user and those which are generic and provided in ParadisEO-MO-GPU.
Initially, to implement a sequential metaheuristic, the user must overload required classes of ParadisEO-EO and ParadisEO-MO. The classes coding the problem-specific part are

abstract classes to be specialized and implemented by the user. To GPU-enable their metaheuristics, users need to derivate their own classes with a bench of new provided ones. In the following components, the main modifications to take into account are detailed:

- *Solution representation.* Some keywords must be specified by the user to indicate which part of the structure will be executed on GPU. Some predefined extensions for binary or permutation representations are already provided in the ParadisEO-GPU module.
- *Population / Neighborhood.* For S-metaheuristics, according to the used neighborhood, the user needs to make an explicit function call to a predefined mapping function. It will automatically allow to find the right association between a neighbor and a GPU thread. Such a problem does not appear when dealing with a population in P-metaheuristics.
- *Problem data inputs.* The user must specify which inputs of the problem to be solved will be allocated on GPU. This can be achieved by introducing additional keywords.
- *Solution evaluation.* In this component, structures, which are likely to be allocated and managed on GPU, must be indicated by the user. All private structures specific to a solution (such as an array) must be declared in a static way. Furthermore, the user might need to adapt multidimensional arrays into one-dimensional ones. Such a hard constraint is required to execute programs in the CUDA platform.

To summarize, regarding new user-defined classes, the user must specify the structures which are likely to be accessed on the GPU device. This strict restriction enables more efficiency and flexibility. Indeed, on the one hand, unnecessary structures for the parallel evaluation of solutions should not be automatically allocated on GPU to reduce the complexity memory space. On the other hand, some additional structures required for each solution evaluation might be copied for each iteration of the metaheuristic, whilst some others are transferred only once during the program. Such a restriction makes it possible to avoid undesirable transfers, which would lead to a performance decrease.
Regarding the components supplied in the software framework, the associated classes establish a hierarchy of classes implementing the invariant part of the code. The different features of these generic components are the following:

- *Memory allocation and deallocation.* According to the specification made by the user, this generic component enables the automatic allocation on GPU of the different structures used for the problem. Type inference and size detection are managed by this component. The same goes on for the deallocation.

- *Data transfers.* For P-metaheuristics, the population is automatically copied from CPU to GPU at each iteration. Regarding S-metaheuristics, the transfer of the candidate solution, which is used to generate the neighborhood, is automatically performed from CPU to GPU. Moreover, this component also ensures the transparent copy of the fitness results from GPU to CPU. Type inference and size detection of the different structures are also supported.

- *Parallel evaluation.* This component manages the kernel of the solutions evaluation. Concepts involving kernel such as thread blocks and block grids are completely hidden to the user.

- *Fitnesses results.* The corresponding structure is manipulated in a transparent manner to store the results of the solutions evaluated on GPU. Afterwards, this structure is sent back to the CPU to continue the search process of the given metaheuristic in a sequential manner.

- *Mapping functions.* For S-metaheuristics, predefined mapping functions control the generation of the neighborhood on GPU. They consist in associating one thread with a specific neighbor. Such a mapping differs according to the used neighborhood.

- *Memory management.* Based on the user specifications, this component manages all the previous structures which are stored on global memory. This management is performed in a transparent way to the user.

The decomposition of the components in ParadisEO-GPU allows to separate the features specific to metaheuristics (ParadisEO-EO and ParadisEO-MO) from those which are related to the GPU code. This separation of concerns makes it possible to split the software framework into distinct features that overlap in functionality as little as possible.

5.2.3 A Case Study: Parallel Evaluation of a Neighborhood

The ParadisEO-GPU execution is illustrated in Figure 5.4 through an UML sequence diagram. In this example, a S-metaheuristic is considered. The scenario shows the design and implementation of the parallel neighborhood evaluation on GPU. At each iteration, the different stages of the parallel evaluation process on GPU are the following:

1. The neighborhood component *moGPUNeighborhood* prepares all the steps for the parallel generation of the neighborhood on GPU. The initialization consists in setting a mapping table between GPU threads and neighbors. Thereafter, the associated data are sent only once to the GPU global memory since the mapping structure does not change during the execution process of S-metaheuristics. The last step relies on

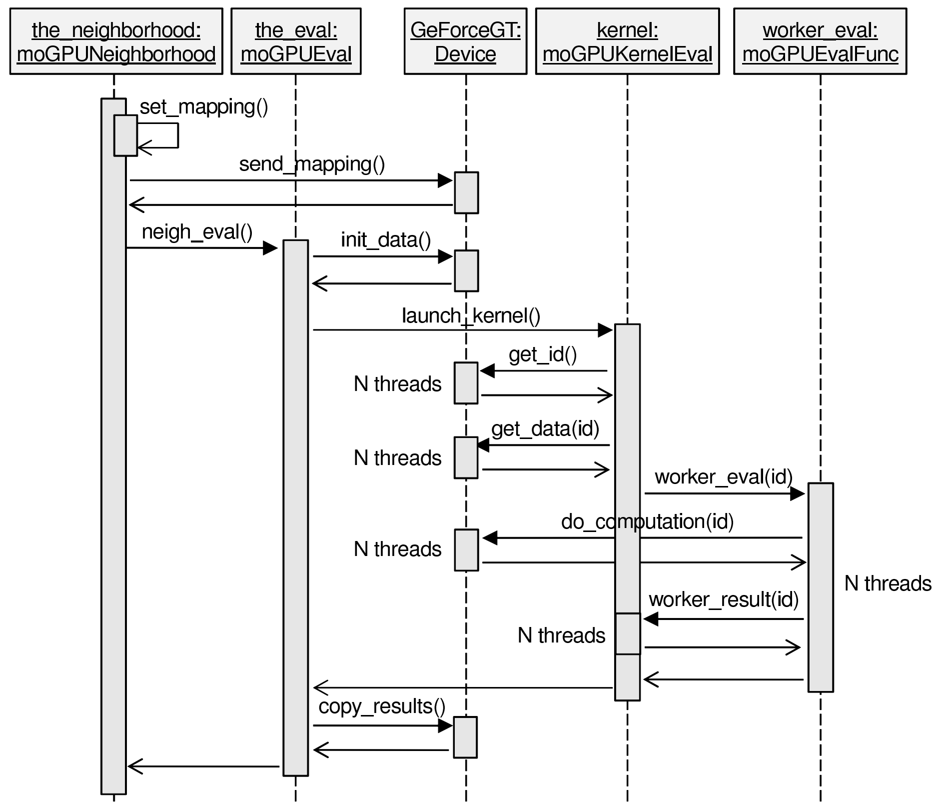


Figure 5.4: The parallel generation and evaluation of a neighborhood provided in ParadisEO-GPU.

the evaluation kernel invocation. It will be informed later on its termination to retrieve the precomputed fitnesses structure.

2. Before proceeding to the parallel evaluation, the component *moGPUEval* configures the kernel with m threads such that each thread is associated exactly with one neighbor evaluation (m designates the neighborhood size). During the first iteration, the component allocates the neighborhood fitnesses structure in which the result of the evaluated neighbors will be stored. Otherwise, in any case, it only sends to the GPU device the candidate solution which generates the neighborhood.

3. The component *moGPUKernelEval* modelizes the main body which will be executed by m concurrent threads on different input data. A first step consists in getting the thread identifier then the set of its associated data. This mechanism is done through the mapping table previously mentioned. The second step calculates the evaluation of the corresponding neighbor. Finally, the resulting fitness is stored in the corresponding index of the fitnesses structure.

4. The worker component *moGPUEvalFunc* is the specific object with computes on the GPU device the neighbor evaluation and returns back the produced result to the CPU.

Once the entire neighborhood has been carried out in parallel on GPU, the precalculated fitness structure is copied back to the CPU and given as input to the ParadisEO-MO module. In this way, the S-metaheuristic continues the neighborhood exploration (iteration) on the CPU side. Instead of reevaluating each neighbor, the corresponding fitness value will be retrieved from the precomputed fitnesses structure. Hence, this mechanism has the advantage of allowing both the deployment of any metaheuristic and the use of toolboxes provided in ParadisEO (e.g. statistical or fitnesses landscape analysis, checkpoint monitors, etc.). This common technique is also currently available for P-metaheuristics.

5.2.4 Automatic Construction of the Mapping Function

As previously said in Chapter 3, for S-metaheuristics, the advantage of generating the neighborhood on GPU is to drastically reduce the data transfers since the whole neighborhood does not have to be copied. However, the main difficulty is to find an efficient mapping between a GPU thread and neighbor candidate solution(s). In other words, the issue is to say which solution must be handled by which thread. The answer is dependent of the solution representation. In Chapter 3, we have provided some mappings for the main neighborhood structures of the literature. However, from an implementation point

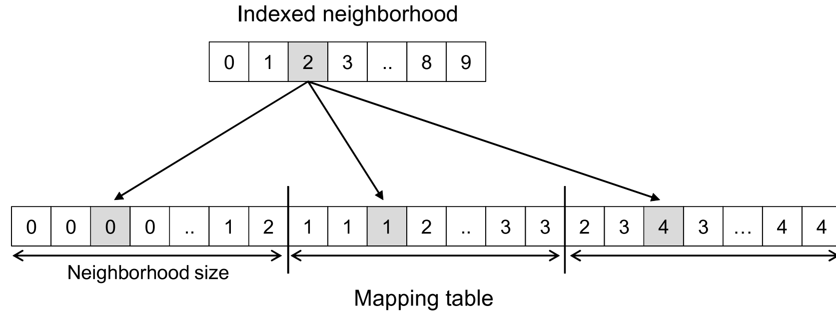


Figure 5.5: Automatic construction of the mapping function.

of view, they are still user-managed. Indeed, the neighborhood structure strongly depends on the target optimization problem representation.

To deal with these issues, mappings for different neighborhoods could be hard-coded in the software framework. However, such a solution does not ensure any flexibility. Hence, we propose to add a supplementary layer in terms of transparency for the deployment of S-metaheuristics on GPU. The main idea is to find a generic mapping which is common for a set of neighborhoods. To achieve this, we provide an automatic construction of the mapping function for k-swaps and k-Hamming distance neighborhoods. Figure 5.5 depicts such a construction of a mapping table. In this example, each neighbor associated with a particular thread can retrieve its three corresponding indexes from the mapping table.

Considering a given vector of size n and a given neighborhood whose neighbors are composed of k indexes with k in $\{1, 2, 3, ...\}$, the size of the associated neighborhood is exactly $m = \dfrac{n \times (n-1) \times ... \times (n-k+1)}{k!}$. The resulting mapping table associates each thread id with a set of k indexes. Each index can be respectively retrieved from the mapping table with the access pattern:

$$\{id, id + m, ..., id + (k-2) \times m, id + (k-1) \times m\}$$

The corresponding mapping table will be used at each iteration of the local search. This table is dynamically constructed on CPU according to the neighborhood size and transferred only once to the GPU global memory during the program execution.

5.3 Performance Evaluation

5.3.1 Experimentation with ParadisEO-GPU

5.3.1.1 Application to the Permuted Perceptron Problem

The objective is to assess the impact in terms of efficiency of an implementation done with ParadisEO-GPU compared with an optimized version done outside the software framework. In this purpose, a tabu search is considered for the permuted perceptron problem. To measure the performance difference, three neighborhoods based on increasing Hamming distances are considered for the experiments. In such neighborhoods, a neighbor is produced by flipping respectively 1, 2 and 3 bits of the candidate solution.
The considered instances of the permuted perceptron problem are the difficult ones presented in [Poi95] for cryptanalysis applications. A tabu search has been implemented in four different versions: 1) a ParadisEO-MO implementation on CPU and its counterpart on GPU; 2) an optimized CPU implementation and its associated GPU version. ParadisEO versions are pure object-based implementations, whilst the optimized ones are pointer-based made outside the software framework.
Experiments have been carried out on top of an Intel Core i7 970 3.2 Ghz with a GTX 480 card (15 multiprocessors with 32 cores). This is actually the machine dedicated to the engineering production. To measure the acceleration factors, only a single CPU core has been considered using the Intel i7 turbo mode (3.46 Ghz). For the different neighborhoods, 50 executions for each different version are considered. The stopping criterion of the S-metaheuristic has been set to 10000 iterations.
Table 5.1 reports the results obtained for the tabu search based on a Hamming distance of one. From the instance $m = 171$ and $n = 187$, both GPU versions start to yield positive accelerations (from $\times 1.1$ to $\times 1.2$). As long as the instance size increases, the acceleration factor grows accordingly (from $\times 1.3$ to $\times 1.7$). The acceleration factor for this implementation is not really significant. This can be explained by the fact that since the neighborhood is relatively small (n threads), the number of threads per block is not enough to fully cover the memory access latency. Furthermore, since the execution time for CPU versions is not meaningful, one can also argue on the use of GPU computing in that case.
An experiment on a larger scale concerns a tabu search using a neighborhood based on a Hamming distance of two. For this neighborhood, the evaluation kernel is executed by $\frac{n \times (n-1)}{2}$ threads. The obtained results from experiments are reported in Table 5.2.
For the first instance ($m = 73$, $n = 73$), acceleration factors are already significant (from $\times 8.2$ and $\times 13.1$). As long as the instance size increases, the acceleration factor grows accordingly. A peak performance is obtained for the last instance (efficient speed-ups varying

Table 5.1: Measures in terms of efficiency of a ParadisEO-GPU implementation with an optimized version made outside the platform. The permuted perceptron problem is considered with a neighborhood based on a Hamming distance of one (n neighbors).

Instance	ParadisEO-MO			Optimized version		
	CPU	GPU	Acc.	CPU	GPU	Acc.
73-73	0.5	1.1	×0.4	0.3	0.8	×0.4
81-81	0.6	1.2	×0.5	0.4	1.0	×0.4
101-117	1.0	1.4	×0.7	0.7	1.2	×0.6
121-137	1.4	1.5	×0.9	1.1	1.3	×0.8
151-167	2.1	1.7	×**1.2**	1.7	1.5	×**1.1**
171-187	2.7	1.9	×**1.4**	2.3	1.7	×**1.4**
201-217	3.8	2.2	×**1.7**	3.3	1.9	×**1.7**

Table 5.2: Measures in terms of efficiency of a ParadisEO-GPU implementation with an optimized version made outside the platform. The permuted perceptron problem is considered with a neighborhood based on a Hamming distance of two ($\frac{n \times (n-1)}{2}$ neighbors).

Instance	ParadisEO-MO			Optimized version			Perf. degradation	
	CPU	GPU	Acc.	CPU	GPU	Acc.	CPU	GPU
73-73	19.8	2.4	×**8.2**	17.1	1.3	×**13.1**	86%	54%
81-81	26	2.8	×**9.3**	22	1.6	×**13.8**	84%	57%
101-117	61	3.9	×**15.6**	51	2.2	×**23.2**	83%	56%
121-137	106	5.2	×**20.4**	91	2.9	×**31.3**	86%	56%
151-167	193	8.0	×**24.1**	134	4.1	×**32.7**	69%	51%
171-187	305	11.3	×**26.9**	208	5.6	×**37.1**	68%	49%
201-217	455	17.6	×**29.5**	343	8.2	×**41.8**	75%	46%

Table 5.3: Measures in terms of efficiency of a ParadisEO-GPU implementation with an optimized version made outside the platform. The permuted perceptron problem is considered with a neighborhood based on a Hamming distance of three ($\frac{n \times (n-1) \times (n-2)}{6}$ neighbors).

Instance	**ParadisEO-MO**			**Optimized version**			**Perf. degradation**	
	CPU	GPU	Acc.	CPU	GPU	Acc.	CPU	GPU
73-73	565	32	×**17.7**	303	15.3	×**19.8**	54%	47%
81-81	877	44	×**19.9**	464	21	×**22.1**	53%	47%
101-117	2468	99	×**24.9**	1375	50	×**27.5**	56%	50%
121-137	4887	182	×**26.8**	3137	89	×**35.2**	64%	48%
151-167	11983	401	×**29.9**	7781	190	×**41.0**	65%	49%
171-187	20239	612	×**33.1**	13089	288	×**45.4**	64%	47%
201-217	37956	1111	×**34.1**	26706	504	×**53.0**	70%	45%

from ×29.5 to ×41.8). A thorough examination of the acceleration factors points out that the performance obtained with ParadisEO-GPU is not so far from an optimized implementation. The performance degradation that occurs is certainly due to the additional cost provided by ParadisEO-MO.

Indeed, regarding the two CPU versions, initially, there is already a performance gap regarding the execution time (between 68% and 86%). This difference can be explained by the overhead caused by the creation of generic objects in ParadisEO whereas the optimized version on CPU is a pure pointer-based implementation. Indeed, the tabu search in ParadisEO-MO is a specialized instantiation of a common template to any S-metaheuristic, whilst the optimized version is a specific tabu search implementation. This may also clarify the performance difference between the two different GPU counterparts in which the same phenomenon occurs. However, for such a transparent exploitation and flexibility, the obtained results are remarkably convincing. A conclusion of this experiment indicates that the performance results of the GPU version provided by ParadisEO are not much degraded compared to the GPU pointer-based one.

As previously said, the definition of the neighborhood is a major step for the performance improvement of the algorithm. Indeed, the increase of the neighborhood size may improve the quality of the obtained solutions. However, its exploitation for solving real-world problems is possible only by using a great computing power. The following experiment intend to perform a large neighborhood obtained with a Hamming distance of 3. For such neighborhood, the evaluation kernel is executed by $\frac{n \times (n-1) \times (n-2)}{6}$ threads. Table 5.3 presents the obtained results for such a large neighborhood.

In general, for the same problem instance, the obtained acceleration factors are much more important than for the previous neighborhoods. For example, for the first instance, the obtained speed-up already varies from ×17.7 to ×19.8. GPU keeps accelerating the

Table 5.4: Measures in terms of efficiency of a ParadisEO-GPU implementation with an optimized version made outside the platform. The quadratic assignment problem is considered with a neighborhood based on a pair-wise exchange operator.

Instance	ParadisEO-MO			Optimized version			Perf. degradation	
	CPU	GPU	Acc.	CPU	GPU	Acc.	CPU	GPU
tai30a	0.8	0.9	×0.9	0.7	0.7	×1.0	87%	77%
tai40a	1.9	1.2	×**1.6**	1.7	1.0	×**1.7**	89%	83%
tai50a	3.6	1.6	×**2.2**	3.0	1.3	×**2.3**	83%	81%
tai60a	6.0	2.0	×**3.0**	5.0	1.6	×**3.1**	83%	80%
tai80a	15	2.8	×**5.4**	12	2.1	×**5.8**	81%	75%
tai100a	32	3.8	×**8.3**	26	2.8	×**9.2**	82%	73%
tai150b	120	8.9	×**13.4**	98	6.5	×**15.1**	82%	73%

process as long as the size grows. A highly significant acceleration varies from ×34.1 to ×53 for the biggest instance (m = 201, n = 217). Regarding the difference between ParadisEO-MO implementations and the optimized ones, the performance degradation is more important between the CPU versions (from 53% to 70%). Indeed, the increase of the neighborhood size may induce more creations of objects. This is also the same case for the performance degradation regarding the GPU counterparts. Nevertheless, according to the reported time measurements, the performance results of ParadisEO-GPU are still satisfactory for such a transparency.

5.3.1.2 Application to the Quadratic Assignment Problem

Another experiment consists in assessing the performance of ParadisEO-GPU for a permutation-based problem. In this purpose, the quadratic assignment problem is considered. Results are reported in Table 5.4 for the different versions.

From the instance tai40a, both GPU versions start giving positive accelerations (from ×1.6 to ×1.7). The poor performance for small instances is explained by the fact that the neighborhood is relatively small. Therefore, the number of threads per block is not enough to fully cover the memory access latency. However, as long as the instance size increases, the acceleration factor grows accordingly (e.g. from ×5.4 to ×5.8 for tai80a). Finally, significant speed-ups are obtained for the instance tai150b. They vary between ×13.4 and ×15.1. In a general manner, since the turbo mode is activated for the core i7 (boost of one single-core), the GPU acceleration factors are less pronounced.

Regarding the performance difference of the two GPU versions, acceleration factors are quite similar. The difference which appears is due to the CPU version provided by ParadisEO-MO. Indeed, regarding the two CPU versions, initially, there is already an existing difference regarding the execution time. Such a gap varies between 81% and 89%

according to the instance. As previously said, this difference is explained by the overhead caused by the creation of generic objects in ParadisEO. This clarifies the performance difference between the two different GPU counterparts (between 73% and 83%). However, for such a transparent exploitation, the obtained results are still satisfactory.

Conclusion

In this chapter, we have proposed a pioneering framework called ParadisEO-GPU for the reusable design and implementation of parallel metaheuristics on GPU architectures. We have revisited the ParadisEO software framework to allow its utilization on GPU accelerators focusing on the parallel iteration-level model.

- **Framework architecture**. The architecture of ParadisEO-GPU has been thought in such a way that the user does not need to have any knowledge about CUDA code for the particular problem to be solved. In this way, ParadisEO-GPU provides facilities for the automatic execution of metaheuristics on GPU.
- **Framework components**. The components in ParadisEO-GPU have been separated to distinguish the features specific to metaheuristics from those which are particular to the CUDA code. This conceptual distinction allows to divide the global framework into distinct features that overlap in functionality as little as possible.
- **Functionalities**. The ParadisEO-GPU provides different transparent tool such as the allocation and copy of data or the parallel evaluation of solutions. In addition, predefined structures (e.g. neighborhood and mapping wrappers for S-metaheuristics) are proposed. Such structures are adapted to hardware characteristics to deal with combinatorial problems.

Conclusion and Future Works

Parallel metaheuristics allow to improve the effectiveness and robustness in combinatorial optimization. Their exploitation for solving real-world problems is possible only by using an important computational power. High-performance computing based on the use of GPUs has been revealed to be a good way to provide such computational power [OML+08, RRS+08, ND10]. However, the exploitation of parallel models is not trivial and many issues related to the GPU memory hierarchical management of this architecture have to be considered. To the best of our knowledge, GPU-based parallel approaches have never been deeply and widely investigated so far. In this document, a new guideline has been established to design and implement efficiently metaheuristics on GPU.

An efficient mapping of the iteration-level parallel model on the hierarchical GPU has been proposed in Chapter 2 and in Chapter 3. For the iteration-level, the CPU manages the whole search process and allows the GPU to be used as a coprocessor dedicated to intensive calculations. Regarding S-metaheuristics, we came up with the pioneering work on GPU-based S-Metaheuristics. Indeed, previous research works [JJL08, ZCM08] just rely on a multi-start execution of local search algorithms. In our contributions, an efficient CPU-GPU cooperation, which minimizes the data transfer between the two components, is a must to achieve the best performance. Then, the purpose of the parallelism control is 1) to control the generation of the neighborhood to meet the memory constraints; 2) to find efficient mappings between neighborhood candidate solutions and GPU threads. The redesign of the parallel iteration-level model on GPU is a good fit for deterministic S-metaheuristics such as hill climbing, tabu search, variable neighborhood search or iterative local search. Furthermore, we proved the robustness of our approach by applying an efficient thread control. This latter allows to prevent GPU-based metaheuristics from crashing when considering a large set of solutions to be evaluated. In addition to this, such a thread control provides additional acceleration improvements.

Another contribution is the redesign of the algorithmic-level through an effective use of the memory management on GPU in Chapter 4. In this purpose, we have particularly focused on P-metaheuristics where there are a significant number of works on GPU (see Section 1.3 in Chapter 1). However, to the best of our knowledge, GPU-based cooperative algorithms have never been intensely studied in terms of memory management, and no general models can be outlined from previous research works [ZH09, PJS10, MWMA09a, SBPE10]. In this document, we have contributed with the entire redesign of parallel cooperative algorithms

on GPU. More exactly, in our contribution, we have proposed three different general schemes for building efficient parallel and cooperative metaheuristics on GPU. In the first scheme, cooperative algorithms are combined with the parallel evaluation of the population on GPU (iteration-level). From an implementation point of view, this approach is the most generic since only the evaluation kernel is considered. However, the performance of this scheme is limited due to the data transfers between the CPU and the GPU. To deal with this issue, the two other schemes operate on the full distribution of the search process on GPU, involving the appropriate use of local memories. Applying such a strategy allows to drastically improve the performance. However, these schemes could present some restrictions due to the memory limitation with some problems that could be more demanding in terms of resources.

In a general manner, we proved the effectiveness of our methods through extensive experiments. In particular, we showed that it enables to gain up on GPU cards to a factor of $\times 80$ in terms of acceleration (compared with a single-core architecture) when deploying it for well known combinatorial instances and up to $\times 2000$ for a continuous problem. In addition to this, experiments indicate that the approaches performed on these problems scale well with last GPU cards such as a Tesla Fermi card. Moreover, the experiments highlight that GPU computing allows not only to speed up the search process, but also to exploit parallelism to improve the quality of the obtained solutions.

In this document, we have also presented a step towards a ParadisEO framework [CMT04] for the reusable design and implementation of the GPU-based parallel metaheuristics. In this contribution, the focus has been set on the iteration-level parallel evaluation of the solutions. We have revisited the design and implementation of this last model in ParadisEO to allow its efficient execution and its transparent use on GPU. In order to reduce the cost of the data transfer, mapping functions are defined and implemented into ParadisEO allowing to assign a thread identifier to each solution. These mapping functions may be used in a fully transparent way for dealing with many problem representations. An implementation made in ParadisEO using CUDA has been experimentally validated and compared to the same implementation realized outside ParadisEO. The experimental results show that the performance degradation that occurs between the two implementations is satisfactory. Indeed, for such a flexibility and an easiness of reuse at implementation, the results obtained with ParadisEO-GPU are really promising. Hence, the use of ParadisEO-GPU is a viable solution. The first release of ParadisEO for GPU architectures is currently available on the ParadisEO website[3]. Tutorials and documentation are provided to facilitate its reuse. This release is dedicated to parallel metaheuristics based on the iteration-level parallel model. In the future, the framework will be extended with further features to be

[3]http://paradiseo.gforge.inria.fr

validated on a wider range of problems.
For a same computational power, GPU computing is much more efficient than cluster of workstations (COWs) and grids for dealing with data-parallel regular applications. Indeed, the main issue in such parallel and distributed architectures is the communication cost. This is due to the synchronous nature of the algorithms proposed in this document. However, since GPU follows a SIMD execution model, it might not be well-adapted for few irregular problems (e.g. [MCT06]). When dealing with such problems in which the computations become asynchronous, using COWs or computational grids might be more relevant.

Most of GPU-accelerated algorithms designed in this manuscript only exploit a single CPU core. With the arrival of GPU resources in COWs and grids, the next objective is to examine the conjunction of GPU computing and distributed computing to fully and efficiently exploit the hierarchy of parallel models of metaheuristics. Indeed, all processors are nowadays multi-core and when coupled with GPU devices hybrid or heterogeneous computing, which is definitely a new trend of parallel computing, performance of GPU-based algorithms might be drastically improved. Furthermore, heterogeneous computing raises other challenging issues that require a large study and many experiments. The challenge will be to find the best mapping in terms of efficiency and effectiveness of the hierarchy of parallel models on the hierarchy of CPU-GPU resources provided by multi-level architectures. This is a very challenging perspective of this work that we will consider in the near future. OpenCL for heterogeneous computing might be the key to address a range of fundamental parallel algorithms on multiple platforms.

Furthermore, in the context of multiobjective optimization, the methods developed in this document could be easily applied to this class of optimization. However, the archiving of non-dominated solutions might represent a prominent issue in the design of multiobjective algorithms on GPU architectures. Indeed, for multiobjective optimization problems whose objectives are uncorrelated, the number of non-dominated solutions could be huge, leading to a serious performance decrease of GPU-based implementations. Even if some archiving techniques could be applied to restrict the archive size, they would not completely solve the issue at all. As a consequence, for being complete, an extension of this work is to provide a SIMD parallel archiving on GPU. This way, it will allow to produce new algorithms for multiobjective optimization. Nevertheless, performing a parallel archiving of non-dominated solutions is challenging. In this purpose, it requires to ensure additional synchronizations, non-concurrent writing operations and to manage some dynamic allocations on GPU.

Appendix

.1 Mapping Proofs

.1.1 Two-to-one Index Transformation

Let us consider a 2D abstraction in which the elements of the neighborhood are disposed in a zero-based indexing 2D representation. This repartition is performed in a similar way as a lower triangular matrix. Let n be the size of the solution representation and let $m = \dfrac{n \times (n-1)}{2}$ be the size of its neighborhood. Let i and j be the indexes of two elements to be exchanged in a permutation. A candidate neighbor is then identified by both i and j indexes in the 2D abstraction. Let $f(i,j)$ be the corresponding index in the 1D neighborhood fitnesses structure. Fig. 6 is an example illustrating this abstraction.

In this example, $n = 6, m = 15$ and the neighbor identified by the coordinates ($i = 2$, $j = 3$) is mapped to the corresponding 1D array element $f(i,j) = 9$.

The neighbor represented by the (i,j) coordinates is known and its corresponding index $f(i,j)$ on the 1D structure has to be calculated. If the 1D array size was $n * n$, the 2D abstraction would be similar to a matrix and thus the mapping would be:

$$f(i,j) = i \times (n-1) + (j-1)$$

Since the 1D array size is $m = \dfrac{n \times (n-1)}{2}$, in the 2D abstraction, elements above the diagonal preceding the neighbor do not have to be considered (illustrated in Fig. 6 by a

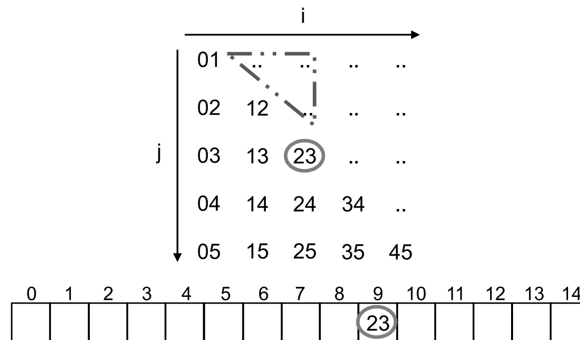


Figure 6: An illustration of the two-to-one transformation.

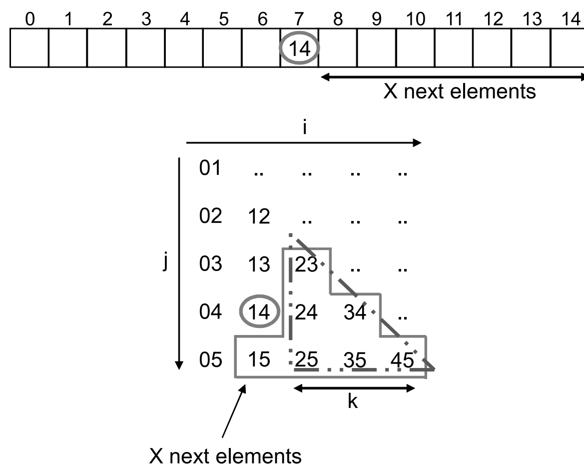


Figure 7: An illustration of the one-to-two transformation.

triangle). The corresponding two-to-one transformation is therefore:

$$f(i,j) = i \times (n-1) + (j-1) - \frac{i \times (i+1)}{2} \tag{1}$$

.1.2 One-to-two Index Transformation

Let us consider the 2D abstraction previously presented. If the element corresponding to $f(i,j)$ in the 2D abstraction has a given i abscissa, then let k be the distance plus one between the $i+1$ and $n-2$ abscissas. If k is known, the value of i can be deduced:

$$i = n - 2 - \lfloor \frac{\sqrt{8X+1}-1}{2} \rfloor \tag{2}$$

Let X be the number of elements following $f(i,j)$ in the neighborhood index-based array numbering:

$$X = m - f(i,j) - 1 \tag{3}$$

Since this number can be also represented in the 2D abstraction, the main idea is to maximize the distance k such as:

$$\frac{k \times (k+1)}{2} \leq X \tag{4}$$

Fig. 7 gives an illustration of this idea (represented by a triangle).
Resolving (4) yields the greatest distance k:

$$k = \lfloor \frac{\sqrt{8X+1}-1}{2} \rfloor \tag{5}$$

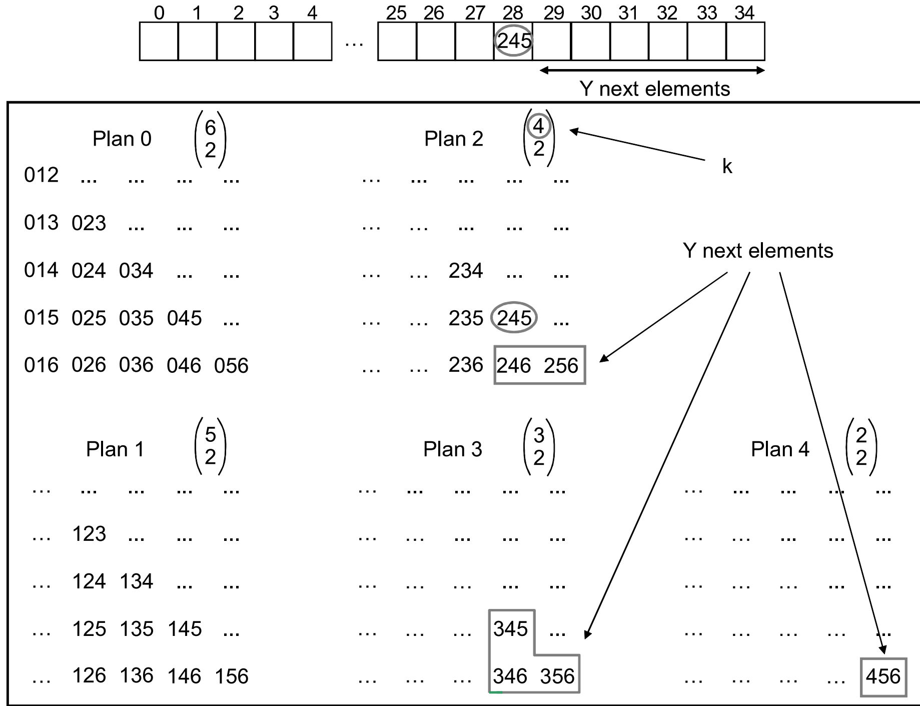


Figure 8: An illustration of the one-to-three transformation.

A value of i can then be calculated according to (2). Finally, by using (1), j can be given by:

$$j = f(i,j) - i \times (n-1) + \frac{i \times (i+1)}{2} + 1 \tag{6}$$

.1.3 One-to-three Index Transformation

$f(x, y, z)$ is a given index of the 1D neighborhood fitnesses structure and the objective is to find the three indexes x, y and z. Let n be the size of the solution representation and $m = \frac{n \times (n-1) \times (n-2)}{6}$ be the size of the neighborhood. The main idea is to find in which plan (coordinate z) corresponds the given element $f(x, y, z)$ in the 3D abstraction. If this corresponding plan is found, then the rest is similar to the one-to-two index transformation. Figure 8 illustrates an example of the 3D abstraction.

In this representation, since each plan is a 2D abstraction, the number of elements in each plan is the number of combinations C_k^2 where $k \in \{2, 3, \dots, n-1\}$ according to each plan. For a specific neighbor, if a value of k is found, then the value of the corresponding plan z is:

$$z = n - k - 1 \tag{7}$$

For a given index $f(x, y, z)$ belonging to the plan k in the 3D abstraction, the number of

elements contained in the next plans is C_k^2 (also equal to $\frac{k \times (k-1) \times (k-2)}{6}$).
Let Y be the number of elements following $f(x,y,z)$ in both the 1D neighborhood fitnesses structure and the 3D abstraction:

$$Y = m - f(x,y,z)$$

Then the main idea is to minimize k such as:

$$\frac{k \times (k-1) \times (k-2)}{6} >= Y \quad (8)$$

By reordering (8), in order to find a value of k, the next step is to solve the following equation:

$$k_1^3 - k_1 - 6Y = 0 \quad (9)$$

Cardano's method in theory allows to solve cubic equation. Nevertheless, in the case of finite discrete machine, this method can lose precision especially for big integers. As a consequence, a simple Newton-Raphson method for finding an approximate value of k_1 is enough for our problem. Indeed, this iterative process follows a set guideline to approximate one root, considering the function, its derivative, an initial arbitrary k_1-value and a certain precision (see Algorithm 14).

Algorithm 14 Newton-Raphson method for solving $k_1^3 - k_1 - 6Y = 0$.

1: $k_1 \leftarrow initial_value$;
2: **repeat**
3: $\quad term \leftarrow (k_1 * k_1 * k_1 - k_1 - 6 * Y) \ / \ (3 * k_1 * k_1 - 1)$;
4: $\quad k_1 \leftarrow k_1 - term$;
5: **until** $|term \ / \ k_1| > precision$

Finally, since the minimization of k in (8) is expected, the value of k is:

$$k = \lceil k_1 \rceil$$

Then a value of z can be deduced with (7). At this step, the plan corresponding to the element $f(x,y,z)$ is known. The next steps for finding x and y are identically the same as the one-to-two index transformation with a change of variables.

First, the number of elements preceding $f(x,y,z)$ in the neighborhood index-based array numbering is exactly:

$$nbElementsBefore = m - \frac{(k+1) \times k \times (k-1)}{6}$$

Second, the number of elements contained in the same plan z as $f(x,y,z)$ is:

$$nbElements = \frac{k \times (k-1)}{2}$$

Finally the index of the last element of the plan z is:

$$lastElement = nbElementsBefore + nbElements - 1$$

As a result, the one-to-two index transformation is applied with a change of variables:

$$f(i,j) = f(x,y,z) - nbElementsBefore$$

$$n' = n - (z+1)$$

$$X = lastElement - f(x,y,z)$$

After performing this transformation, a value of x and y can be deduced:

$$x = i + (z+1)$$

$$y = j + (z+1)$$

.1.4 Three-to-one index transformation

x, y and z are known and its corresponding index $f(x,y,z)$ have to be found. According to the 3D abstraction, since a value of z is known, k can be calculated:

$$k = n - 1 - z$$

Then the number of elements preceding $f(x,y,z)$ in the neighborhood index-based array numbering can be also deduced.

If each plan size was $(n-2) * (n-2)$, each 2D abstraction would be similar to a matrix and the $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping would be:

$$f_1(x,y,z) = z \times (n-2) \times (n-2) + (x-1) \times (n-2) + (y-2) \qquad (10)$$

Since each 2D abstraction looks like a triangular matrix, some elements must not be considered. The advantage of the 3D abstraction is that these elements can be found by geometric construction (see Fig. 9).

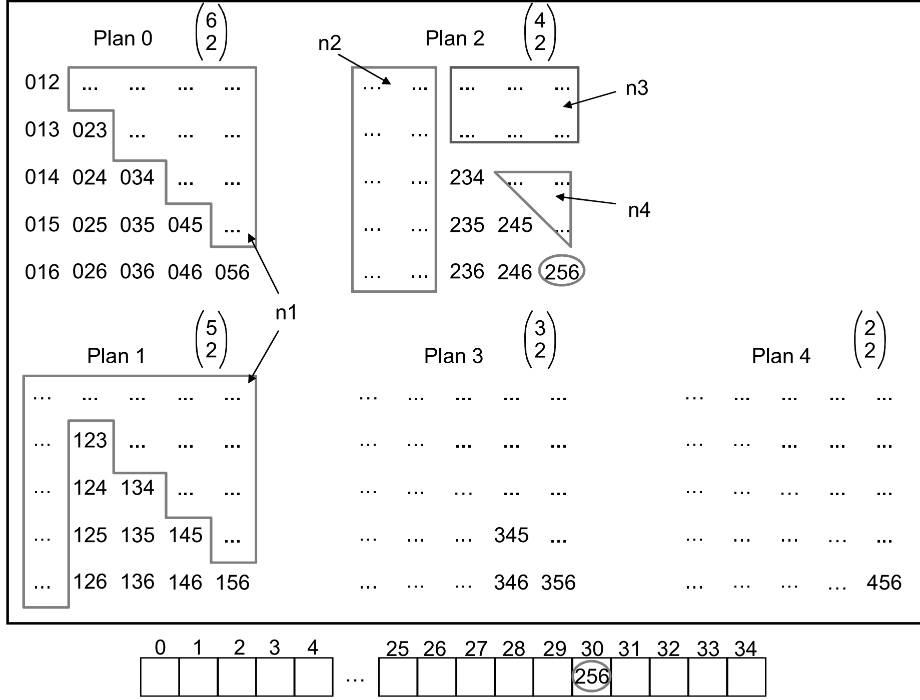


Figure 9: $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping.

First, given a plan z, the number of elements in the previous plans to not consider is:

$$n1 = z \times (n-2) \times (n-2) - nbElementsBefore$$

Second, the number of elements on the left side to not consider in the plan z is:

$$n2 = z \times (n-2)$$

Third, the number of elements on the upper side to not consider in the plan z is:

$$n3 = (y-z) \times (n-k-1)$$

Fourth, the number of elements on the upper triangle above $f(x,y,z)$ to not consider is:

$$n4 = \frac{(y-z) \times (y-z-1)}{2}$$

Finally a value of $f(x,y,z)$ can be deduced:

$$f(x,y,z) = f_1(x,y,z) - n1 - n2 - n3 - n4 \tag{11}$$

Table 5: Measures in terms of efficiency for the permuted perceptron problem using a neighborhood based on a Hamming distance of two (binary representation). Test of the null hypothesis of normality with the Kolmogorov-Smirnov's test.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores**		**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores**		**Intel Xeon E5450 GeForce GTX 280 240 GPU cores**		**Xeon E5620 Tesla M2050 448 GPU cores**	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
73-73	+	+	+	+	+	+	+	+
81-81	+	+	+	+	+	+	+	+
101-117	+	+	+	+	+	+	+	+
201-217	+	+	+	+	+	+	+	+
401-417	+	+	+	+	+	+	+	+
601-617	+	+	+	+	+	+	+	+
801-817	+	+	+	+	+	+	+	+
1001-1017	+	+	+	+	+	+	+	+
1301-1317	+	+	+	+	+	+	+	+

Table 6: Measures in terms of efficiency for the permuted perceptron problem using a neighborhood based on a Hamming distance of two (binary representation). Test of the null hypothesis of variances equality with the Levene's test.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores**	**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores**	**Intel Xeon E5450 GeForce GTX 280 240 GPU cores**	**Xeon E5620 Tesla M2050 448 GPU cores**
	CPU - GPU	CPU - GPU	CPU - GPU	CPU - GPU
73-73	+	+	+	+
81-81	+	+	+	+
101-117	+	+	+	+
201-217	+	+	+	+
401-417	+	+	+	+
601-617	+	+	+	+
801-817	+	+	+	+
1001-1017	+	+	+	+
1301-1317	+	+	+	+

.2 Statistical Tests

Table 7: Measures in terms of efficiency for the permuted perceptron problem using a neighborhood based on a Hamming distance of two (binary representation). Test of the null hypothesis of the averages equality with the Student's t-test.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores** CPU - GPU	**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores** CPU - GPU	**Intel Xeon E5450 GeForce GTX 280 240 GPU cores** CPU - GPU	**Xeon E5620 Tesla M2050 448 GPU cores** CPU - GPU
73-73	-	-	-	-
81-81	-	-	-	-
101-117	-	-	-	-
201-217	-	-	-	-
401-417	**0.083**	-	-	-
601-617	-	-	-	-
801-817	-	-	-	-
1001-1017	-	-	-	-
1301-1317	-	-	-	-

Table 8: Measures in terms of efficiency for the traveling salesman problem using a 2-opt neighborhood (permutation representation). Test of the null hypothesis of normality with the Kolmogorov-Smirnov's test.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores**		**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores**		**Intel Xeon E5450 GeForce GTX 280 240 GPU cores**		**Xeon E5620 Tesla M2050 448 GPU cores**	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
eil101	+	+	+	+	+	+	+	+
d198	+	+	+	+	+	+	+	+
pcb442	+	+	+	+	+	+	+	+
rat783	+	+	+	+	+	+	+	+
d1291	+	+	+	+	+	+	+	+
pr2392	+	.	+	+	+	+	+	+
fnl4461	+	.	+	.	+	+	+	+
rl5915	+	.	+	.	+	.	+	+

Table 9: Measures in terms of efficiency for the traveling salesman problem using a 2-opt neighborhood (permutation representation). Test of the null hypothesis of variances equality with the Levene's test.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores** CPU - GPU	**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores** CPU - GPU	**Intel Xeon E5450 GeForce GTX 280 240 GPU cores** CPU - GPU	**Xeon E5620 Tesla M2050 448 GPU cores** CPU - GPU
eil101	+	+	+	+
d198	+	+	+	+
pcb442	+	+	+	+
rat783	+	+	+	+
d1291	+	+	+	+
pr2392	.	+	+	+
fnl4461	.	.	+	+
rl5915	.	.	.	+

Table 10: Measures in terms of efficiency for the traveling salesman problem using a 2-opt neighborhood (permutation representation). Test of the null hypothesis of the averages equality with the Student's t-test.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores**	**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores**	**Intel Xeon E5450 GeForce GTX 280 240 GPU cores**	**Xeon E5620 Tesla M2050 448 GPU cores**
	CPU - GPU	CPU - GPU	CPU - GPU	CPU - GPU
eil101	-	**0.064**	-	-
d198	-	-	-	-
pcb442	-	-	-	-
rat783	-	-	-	-
d1291	-	-	-	-
pr2392	.	-	-	-
fnl4461	.	.	-	-
rl5915	.	.	.	-

Table 11: Measures of the benefits of applying thread control. The traveling salesman problem using a 2-opt neighborhood is considered. Test of the null hypothesis of normality with the Kolmogorov-Smirnov's test.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores**		**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores**		**Intel Xeon E5450 GeForce GTX 280 240 GPU cores**		**Xeon E5620 Tesla M2050 448 GPU cores**	
	GPU	GPU_{TC}	GPU	GPU_{TC}	GPU	GPU_{TC}	GPU	GPU_{TC}
eil101	+	+	+	+	+	+	+	+
d198	+	+	+	+	+	+	+	+
pcb442	+	+	+	+	+	+	+	+
rat783	+	+	+	+	+	+	+	+
d1291	+	+	+	+	+	+	+	+
pr2392	.	+	+	+	+	+	+	+
fnl4461	.	+	.	+	+	+	+	+
rl5915	.	+	.	+	.	+	+	+

Table 12: Measures of the benefits of applying thread control. The traveling salesman problem using a 2-opt neighborhood is considered. Test of the null hypothesis of variances equality with the Levene's test.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores**	**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores**	**Intel Xeon E5450 GeForce GTX 280 240 GPU cores**	**Xeon E5620 Tesla M2050 448 GPU cores**
	GPU - GPU_{TC}	GPU - GPU_{TC}	GPU - GPU_{TC}	GPU - GPU_{TC}
eil101	+	+	+	+
d198	+	+	+	+
pcb442	+	+	+	+
rat783	+	+	+	+
d1291	+	+	+	+
pr2392	.	+	+	+
fnl4461	.	.	+	+
rl5915	.	.	.	+

Table 13: Measures of the benefits of applying thread control. The traveling salesman problem using a 2-opt neighborhood is considered. Test of the null hypothesis of the averages equality with the Student's t-test.

Instance	**Core 2 Duo T5800 GeForce 8600M GT 32 GPU cores**	**Core 2 Quad Q6600 GeForce 8800 GTX 128 GPU cores**	**Intel Xeon E5450 GeForce GTX 280 240 GPU cores**	**Xeon E5620 Tesla M2050 448 GPU cores**
	GPU - GPU_{TC}	GPU - GPU_{TC}	GPU - GPU_{TC}	GPU - GPU_{TC}
eil101	**0.58**	-	**0.65**	-
d198	**0.62**	-	**0.60**	-
pcb442	-	-	-	-
rat783	**0.55**	-	-	-
d1291	**0.64**	-	-	-
pr2392	.	-	-	-
fnl4461	.	.	-	-
rl5915	.	.	.	-

Table 14: Measures of the benefits of using the reduction operation on the GTX 280. The permuted perceptron problem is considered for two different neighborhoods using 100 hill climbing algorithms. Test of the null hypothesis of normality with the Kolmogorov-Smirnov's test.

Instance	n neighbors			$\frac{n\times(n-1)}{2}$ neighbors		
	CPU	GPU	GPU_R	CPU	GPU	GPU_{TexR}
73-73	+	+	+	+	+	+
81-81	+	+	+	+	+	+
101-117	+	+	+	+	+	+
201-217	+	+	+	+	+	+
401-417	+	+	+	+	+	+
601-617	+	+	+	+	+	+
801-817	+	+	+	+	+	+
1001-1017	+	+	+	+	+	+
1301-1317	+	+	+	+	+	+

Table 15: Measures of the benefits of using the reduction operation on the GTX 280. The permuted perceptron problem is considered for two different neighborhoods using 100 hill climbing algorithms. Test of the null hypothesis of variances equality with the Levene's test.

Instance	n neighbors		$\frac{n\times(n-1)}{2}$ neighbors	
	CPU - GPU_R	GPU - GPU_R	CPU - GPU_R	GPU - GPU_{TexR}
73-73	+	+	+	+
81-81	+	+	+	+
101-117	+	+	+	+
201-217	+	+	+	+
401-417	+	+	+	+
601-617	+	+	+	+
801-817	+	+	+	+
1001-1017	+	+	+	+
1301-1317	+	+	+	+

Table 16: Measures of the benefits of using the reduction operation on the GTX 280. The permuted perceptron problem is considered for two different neighborhoods using 100 hill climbing algorithms. Test of the null hypothesis of the averages equality with the Student's t-test.

Instance	n neighbors		$\frac{n \times (n-1)}{2}$ neighbors	
	CPU - GPU_R	GPU - GPU_R	CPU - GPU_R	GPU - GPU_{TexR}
73-73	**0.064**	**0.059**	-	-
81-81	**0.058**	**0.057**	-	-
101-117	**0.071**	**0.060**	-	-
201-217	**0.062**	**0.063**	-	-
401-417	-	-	-	-
601-617	-	-	-	-
801-817	-	-	-	-
1001-1017	-	-	-	-
1301-1317	-	-	-	-

Bibliography

[ABR03] Renata M. Aiex, S. Binato, and Mauricio G. C. Resende. Parallel grasp with path-relinking for job shop scheduling. *Parallel Computing*, 29(4):393–430, 2003.

[AGM+07] Ravindra K. Ahuja, Jon Goodstein, Amit Mukherjee, James B. Orlin, and Dushyant Sharma. A very large-scale neighborhood search algorithm for the combined through-fleet-assignment model. *INFORMS Journal on Computing*, 19(3):416–428, 2007.

[AK96] David Andre and John R. Koza. Parallel genetic programming: a scalable implementation using the transputer network architecture. *Advances in genetic programming: volume 2*, pages 317–337, 1996.

[ALNT04] Enrique Alba, Francisco Luna, Antonio J. Nebro, and José M. Troya. Parallel heterogeneous genetic algorithms for continuous optimization. *Parallel Computing*, 30(5-6):699–719, 2004.

[AT02] Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *IEEE Trans. Evolutionary Computation*, 6(5):443–462, 2002.

[ATD10] Ramnik Arora, Rupesh Tulshyan, and Kalyanmoy Deb. Parallelization of binary and real-coded genetic algorithms on gpu using cuda. In *IEEE Congress on Evolutionary Computation* [DBL10], pages 1–8.

[BCC+06] Raphael Bolze, Franck Cappello, Eddy Caron, Michel J. Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quétier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *IJHPCA*, 20(4):481–494, 2006.

[BcRW98] Rainer E. Burkard, Eranda Çela, Günter Rote, and Gerhard J. Woeginger. The quadratic assignment problem with a monotone anti-monge and a symmetric toeplitz matrix: Easy and hard cases. *Math. Program.*, 82:125–158, 1998.

[Bev02] Alessandro Bevilacqua. A methodological approach to parallel simulated annealing on an smp system. *J. Parallel Distrib. Comput.*, 62(10):1548–1570, 2002.

[BFM97] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation.* IOP Publishing Ltd., Bristol, UK, UK, 1st edition, 1997.

[BHX00] Maria J. Blesa, Lluis Hernandez, and Fatos Xhafa. Parallel skeletons for tabu search method. In *In Proceedings of International Conference on Parallel and Distributed Systems, ICPADS '01, IEEE*, 2000.

[BOL+09] Hongtao Bai, Dantong OuYang, Ximing Li, Lili He, and Haihong Yu. Max-min ant system on gpu with cuda. In *Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control*, ICICIC '09, pages 801–804, Washington, DC, USA, 2009. IEEE Computer Society.

[BSB+01] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra A. Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, 2001.

[CB96] John A. Chandy and Prithviraj Banerjee. Parallel simulated annealing strategies for vlsi cell placement. In *VLSI Design* [DBL96], pages 37–42.

[CBM+08] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distributed Computing*, 68(10):1370–1380, 2008.

[CG02] Teodor Gabriel Crainic and Michel Gendreau. Cooperative parallel tabu search for capacitated network design. *J. Heuristics*, 8(6):601–627, 2002.

[CGHM04] Teodor Gabriel Crainic, Michel Gendreau, Pierre Hansen, and Nenad Mladenovic. Cooperative parallel variable neighborhood search for the -median. *J. Heuristics*, 10(3):293–314, 2004.

[CGU^{+}11] José M. Cecilia, José M. García, Manuel Ujaldon, Andy Nisbet, and Martyn Amos. Parallelization strategies for ant colony optimisation on gpus. In *IPDPS Workshops*, pages 339–346. IEEE, 2011.

[Chi07] Darren M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1566–1573, London, 2007. ACM Press.

[CMT04] Sébastien Cahon, Nordine Melab, and El-Ghazali Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *J. Heuristics*, 10(3):357–380, 2004.

[CS00] Rachid Chelouah and Patrick Siarry. Tabu search applied to global optimization. *European Journal of Operational Research*, 123(2):256–270, 2000.

[CSK93] J. Chakrapani and J. Skorin-Kapov. Massively Parallel Tabu Search for the Quadratic Assignment Problem. *Annals of Operations Research*, 41:327–341, 1993.

[CSV10] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *SIGPLAN Not.*, 45:115–126, January 2010.

[CTG95] T.G. Crainic, M. Toulouse, and M. Gendreau. Parallel Asynchronous Tabu Search for Multicommodity Location-Allocation with Balancing Requirements. *Annals of Operations Research*, 63:277–299, 1995.

[DBL96] *9th International Conference on VLSI Design (VLSI Design 1996), 3-6 January 1996, Bangalore, India.* IEEE Computer Society, 1996.

[DBL10] *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010.* IEEE, 2010.

[DG97] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Trans. on Evolutionary Computation*, 1(1):53–66, 1997.

[DPST06] J. Dréo, A. Pétrowski, P. Siarry, and E. Taillard. *Metaheuristics for Hard Optimization.* Springer, 2006.

[dPVK10] Lucas de P. Veronese and Renato A. Krohling. Differential evolution algorithm on the gpu with c-cuda. In *IEEE Congress on Evolutionary Computation* [DBL10], pages 1–7.

[FKB10] María A. Franco, Natalio Krasnogor, and Jaume Bacardit. Speeding up the evaluation of evolutionary learning systems using gpgpus. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 1039–1046, New York, NY, USA, 2010. ACM.

[FWW07] Ka-Ling Fok, Tien-Tsin Wong, and Man-Leung Wong. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, 22:69–78, 2007.

[GB77] S.W Golomb and G.S. Bloom. Applications of numbered undirected graphs. *Proceedings of the IEEE*, 65(4):562–570, 1977.

[GLGN+08] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, 2008.

[GLMBMPMV03] Félix García-López, Belén Melián-Batista, José A. Moreno-Pérez, and J. Marcos Moreno-Vega. Parallelization of the scatter search for the p-median problem. *Parallel Computing*, 29(5):575–589, 2003.

[Glo89] Fred Glover. Tabu search - part i. *INFORMS Journal on Computing*, 1(3):190–206, 1989.

[Glo90] Fred Glover. Tabu search - part ii. *INFORMS Journal on Computing*, 2(1):4–32, 1990.

[GPR94] Bruno-Laurent Garcia, Jean-Yves Potvin, and Jean-Marc Rousseau. A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers & OR*, 21(9):1025–1033, 1994.

[Gro10] Khronos Group. *OpenCL 1.0 Quick Reference Card*, 2010.

[GrTA99] Luca Maria Gambardella, Éric Taillard, and Giovanni Agazzi. Macs-vrptw: A multiple colony system for vehicle routing problems with time windows. In *New Ideas in Optimization*, pages 63–76. McGraw-Hill, 1999.

[GZ06] Alexander Greß and Gabriel Zachmann. Gpu-abisort: optimal parallel sorting on stream architectures. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[Har08] Mark Harris. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2008.

[HB07] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on GPUs. In *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2007.

[HL00] Francisco Herrera and Manuel Lozano. Gradual distributed real-coded genetic algorithms. *IEEE Trans. Evolutionary Computation*, 4(1):43–63, 2000.

[JJL08] Adam Janiak, Wladyslaw A. Janiak, and Maciej Lichtenstein. Tabu search on gpu. *J. UCS*, 14(14):2416–2426, 2008.

[JJMH03] Gabriele Jost, Haoqiang Jin, Dieter An Mey, and Ferhat F. Hatay. Comparing the openmp, mpi, and hybrid programming paradigm on an smp cluster. NASA Technical Report, 2003.

[JRG09] T. James, C. Rego, and F. Glover. A cooperative parallel tabu search algorithm for the quadratic assignment problem. *European Journal of Operational Research*, 195:810–826, 2009.

[KGRS01] Maarten Keijzer, Juan J. Merelo Guervós, Gustavo Romero, and Marc Schoenauer. Evolving objects: A general purpose evolutionary computation library. In Pierre Collet, Cyril Fonlupt, Jin-Kao Hao, Evelyne Lutton, and Marc Schoenauer, editors, *Artificial Evolution*, volume 2310 of *Lecture Notes in Computer Science*, pages 231–244. Springer, 2001.

[KM99] Lars R. Knudsen and Willi Meier. Cryptanalysis of an identification scheme based on the permuted perceptron problem. In *EUROCRYPT*, pages 363–374, 1999.

[Kru56] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In *Proceedings of the American Mathematical Society, 7*, 1956.

[Lan11] William B. Langdon. Graphics processing units and genetic programming: an overview. *Soft Comput.*, 15(8):1657–1669, 2011.

[LB08] William B. Langdon and Wolfgang Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85, Naples, 26-28 March 2008. Springer.

[LL96] Soo-Young Lee and Kyung-Geun Lee. Synchronous and asynchronous parallel simulated annealing with multiple markov chains. *IEEE Trans. Parallel Distrib. Syst.*, 7(10):993–1008, 1996.

[LL06] Zhongwen Luo and Hongzhi Liu. Cellular genetic algorithms and local search for 3-sat problem on graphic hardware. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2988 –2992, 2006.

[LV98] Evelyne Lutton and Jacques Lévy Véhel. Holder functions and deception of genetic algorithms. *IEEE Trans. on Evolutionary Computation*, 2(2):56–71, 1998.

[LWHC07] JIAN-MING LI, XIAO-JING WANG, RONG-SHENG HE, and ZHONG-XIAN CHI. An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, NPC ’07, pages 855–862, Washington, DC, USA, 2007. IEEE Computer Society.

[MBL+09] Ogier Maitre, Laurent A. Baumes, Nicolas Lachiche, Avelino Corma, and Pierre Collet. Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO ’09, pages 1403–1410, New York, NY, USA, 2009. ACM.

[MCD09] Luca Mussi, Stefano Cagnoni, and Fabio Daolio. Gpu-based road sign detection using particle swarm optimization. In *ISDA*, pages 152–157. IEEE Computer Society, 2009.

[MCT06] Nouredine Melab, Sébastien Cahon, and El-Ghazali Talbi. Grid computing for parallel bioinspired algorithms. *J. Parallel Distributed Computing*, 66(8):1052–1061, 2006.

[MWMA09a] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework. *Genetic Programming and Evolvable Machines*, 10:391–415, 2009. 10.1007/s10710-009-9091-4.

[MWMA09b] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. Theoretical and empirical analysis of a gpu based parallel bayesian optimization algorithm. In *Parallel and Distributed Computing, Applications and Technologies*, PDCAT, pages 457–462. IEEE Computer Society, 2009.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, 2008.

[ND10] John Nickolls and William J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, 2010.

[NM09] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[NVI10] NVIDIA. *GPU Gems 3. Chapter 37: Efficient Random Number Generation and Application Using CUDA*, 2010.

[NVI11] NVIDIA. *CUDA Programming Guide Version 4.0*, 2011.

[OML+08] J. D. Owens, M.Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[PDB10] Frédéric Pinel, Bernabé Dorronsoro Díaz, and Pascal Bouvry. A new parallel asynchronous cellular genetic algorithm for scheduling in grids. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

[PJS10] Petr Pospichal, Jirí Jaros, and Josef Schwarz. Parallel genetic algorithm on the cuda architecture. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna Esparcia-Alcázar, Chi Keong Goh, Juan J. Merelo Guervós, Ferrante Neri, Mike Preuss, Julian Togelius, and Georgios N. Yannakakis, editors, *EvoApplications (1)*, volume 6024 of *Lecture Notes in Computer Science*, pages 442–451. Springer, 2010.

[Poi95] David Pointcheval. A new identification scheme based on the perceptrons problem. In *EUROCRYPT*, pages 319–328, 1995.

[RK10] Boguslaw Rymut and Bogdan Kwolek. Gpu-supported object tracking using adaptive appearance models and particle swarm optimization. In Leonard Bolc, Ryszard Tadeusiewicz, Leszek J. Chmielewski, and Konrad W. Wojciechowski, editors, *ICCVG (2)*, volume 6375 of *Lecture Notes in Computer Science*, pages 227–234. Springer, 2010.

[RL02] Marcus Randall and Andrew Lewis. A parallel implementation of ant colony optimization. *J. Parallel Distrib. Comput.*, 62(9):1421–1432, 2002.

[RRS+08] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen mei W. Hwu. Program optimization carving for gpu computing. *J. Parallel Distributed Computing*, 68(10):1389–1401, 2008.

[SAGM10] Nicholas A. Sinnott-Armstrong, Casey S. Greene, and Jason H. Moore. Fast genome-wide epistasis analysis using ant colony optimization for multifactor dimensionality reduction analysis on graphics processing units. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 215–216, New York, NY, USA, 2010. ACM.

[SBA97] Franciszek Seredynski, Pascal Bouvry, and Farhad Arbab. Parallel evolutionary computation: Multi agents genetic algorithms. In *Euro-PDS*, pages 293–298. IASTED/ACTA Press, 1997.

[SBPE10] Nicolas Soca, Jose Luis Blengio, Martin Pedemonte, and Pablo Ezzatti. Pugace, a cellular evolutionary algorithm framework on gpus. In *IEEE Congress on Evolutionary Computation* [DBL10], pages 1–8.

[Tal09] El-Ghazali Talbi. *Metaheuristics: From design to implementation.* Wiley, 2009.

[TF09] Shigeyoshi Tsutsui and Noriyuki Fujimoto. Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2523–2530, New York, NY, USA, 2009. ACM.

[TF11] Shigeyoshi Tsutsui and Noriyuki Fujimoto. Fast qap solving by aco with 2-opt local search on a gpu. In *IEEE Congress on Evolutionary Computation*, pages 812–819. IEEE, 2011.

[TMDT07] A-A. Tantar, N. Melab, C. Demarey, and E-G. Talbi. Building a Virtual Globus Grid in a Reconfigurable Environment - A case study: Grid5000. In *INRIA Research Report*. HAL INRIA, 2007.

[TMT07] Alexandru-Adrian Tantar, Nouredine Melab, and El-Ghazali Talbi. A comparative study of parallel metaheuristics for protein structure prediction on the computational grid. In *IPDPS*, pages 1–10. IEEE, 2007.

[TSP+08] Christian Tenllado, Javier Setoain, Manuel Prieto, Luis Piñuel, and Francisco Tirado. Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting. *IEEE Transactions on Parallel and Distributed Systems*, 19(3):299–310, 2008.

[VA10a] Pablo Vidal and Enrique Alba. Cellular genetic algorithm on graphic processing units. In Juan González, David Pelta, Carlos Cruz, Germán Terrazas, and Natalio Krasnogor, editors, *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, volume 284 of *Studies in Computational Intelligence*, pages 223–232. Springer Berlin / Heidelberg, 2010.

[VA10b] Pablo Vidal and Enrique Alba. A multi-gpu implementation of a cellular genetic algorithm. In *IEEE Congress on Evolutionary Computation* [DBL10], pages 1–7.

[Won09] Man Leung Wong. Parallel multi-objective evolutionary algorithms on graphics processing units. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2515–2522, New York, NY, USA, 2009. ACM.

[WW06] Man-Leung Wong and Tien-Tsin Wong. Parallel hybrid genetic algorithms on consumer-level graphics hardware. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2973–2980, 2006.

[WWF05] Man Leung Wong, Tien-Tsin Wong, and Ka-Ling Fok. Parallel evolutionary algorithms on graphics processing unit. In *Congress on Evolutionary Computation*, pages 2286–2293. IEEE, 2005.

[YCP05] Qizhi Yu, Chongcheng Chen, and Zhigeng Pan. Parallel genetic algorithms on programmable graphics hardware. In *Lecture Notes in Computer Science 3612*, page 1051. Springer, 2005.

[ZCM08] W Zhu, J Curry, and A Marquez. Simd tabu search with graphics hardware acceleration on the quadratic assignment problem. *International Journal of Production Research*, 2008.

[ZH09] Sifa Zhang and Zhenming He. Implementation of parallel genetic algorithm based on cuda. In Zhihua Cai, Zhenhua Li, Zhuo Kang, and Yong Liu, editors, *Advances in Computation and Intelligence*, volume 5821 of *Lecture Notes in Computer Science*, pages 24–30. Springer Berlin / Heidelberg, 2009.

[Zhu09] Weihang Zhu. A study of parallel evolution strategy: pattern search on a gpu computing platform. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, GEC '09, pages 765–772, New York, NY, USA, 2009. ACM.

[ZT09] You Zhou and Ying Tan. Gpu-based parallel particle swarm optimization. In *IEEE Congress on Evolutionary Computation*, pages 1493–1500. IEEE, 2009.

International Publications

---

[1] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU Computing for Local Search Metaheuristic Algorithms. *IEEE Transactions on Computers*, in press, 2011.

[2] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Neighborhood Structures for GPU-based Local Search Algorithms. *Parallel Processing Letters*, 20(4):307–324, 2010.

[3] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU-based Approaches for Multiobjective Local Search Algorithms. A Case Study: the Flowshop Scheduling Problem. *11th European Conference on Evolutionary Computation in Combinatorial Optimization, EVOCOP 2011*, pages 155–166, volume 6622 of Lecture Notes in Computer Science , Springer, 2011.

[4] Nouredine Melab, Thé Van Luong, K. Boufaras, and El-Ghazali Talbi. Towards ParadisEO-MO-GPU: A Framework for GPU-based Local Search Metaheuristics. *11th International Work-Conference on Artificial Neural Networks, IWANN 2011*, pages 401–408, volume 6691 of Lecture Notes in Computer Science, Springer, 2011.

[5] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU-based Multi-start Local Search Algorithms. *4th International Learning and Intelligent Optimization Conference, LION 5*, in press, Lecture Notes in Compute Science, Springer, 2011.

[6] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU-based Island Model for Evolutionary Algorithms. *Genetic and Evolutionary Computation Conference, GECCO 2010* pages 1089–1096, Proceedings, ACM, 2010.

[7] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Parallel Hybrid Evolutionary Algorithms on GPU. In *IEEE Congress on Evolutionary Computation, CEC 2010*, pages 1–8, Proceedings, IEEE, 2010.

[8] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Large Neighborhood Local Search Optimization on Graphics Processing Units. *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, pages 1–8, Workshop Proceedings, IEEE, 2010.

[9] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Local Search Algorithms on Graphics Processing Units. A case study: the Permutation Perceptron Problem. *Evolutionary Computation in Combinatorial Optimization, 10th European Conference, EvoCOP 2010*, pages 264–275, volume 6022 of Lecture Notes in Computer Science, Springer, 2010.

[10] Thé Van Luong, Lakhdar Loukil, Nouredine Melab, and El-Ghazali Talbi. A GPU-based Iterated Tabu Search for Solving the Quadratic 3-dimensional Assignment Problem. *The 8th ACS/IEEE International Conference on Computer Systems and Applications, AICCSA 2010*, pages 1–8, Proceedings, IEEE, 2010.

[11] Naouel Ayari, Thé Van Luong, and Abderrazak Jemai. A hybrid Genetic Algorithm for Golomb Ruler Problem. *The 8th ACS/IEEE International Conference on Computer Systems and Applications, AICCSA 2010*, pages 1–4, Proceedings, IEEE, 2010.

Résumé :
Les problèmes d'optimisation du monde réel sont souvent complexes et NP-difficiles. Bien que des algorithmes approchés telles que les métaheuristiques permettent de réduire la complexité de leur résolution, ces méthodes restent insuffisantes pour traiter des problèmes de grande taille. De nos jours, le calcul sur GPU s'est révélé efficace pour traiter des problèmes coûteux en temps de calcul. Un des enjeux majeurs pour les métaheuristiques est de repenser les modèles existants pour permettre leur déploiement sur les accélérateurs GPU. La contribution de cette thèse porte sur la reconception de ces modèles parallèles pour permettre la résolution des problèmes d'optimisation à large échelle sur les architectures GPU. Pour cela, des approches efficaces ont été proposées pour l'optimisation des transferts de données entre le CPU et le GPU, le contrôle de threads ou encore la gestion de la mémoire. Les approches ont été expérimentées de façon exhaustive en utilisant cinq problèmes d'optimisation et quatre configurations GPU. En comparaison avec une exécution sur CPU, les accélérations obtenues vont jusqu'à 80 fois plus vite pour des problèmes d'optimisation combinatoire et jusqu'à 2000 fois pour un problème d'optimisation continue.

Mots-clés : métaheuristiques parallèles, calcul sur GPU, recherche locale, algorithmes évolutionnaires.

Abstract:
Real-world optimization problems are often complex and NP-hard. Although near-optimal algorithms such as metaheuristics make it possible to reduce the temporal complexity of their resolution, they fail to tackle large problems satisfactorily. Nowadays, GPU computing has recently been revealed effective to deal with time-intensive problems. One of the major issues for metaheuristics is to rethink existing parallel models and programming paradigms to allow their deployment on GPU accelerators. The contribution of this thesis is to deal with such issues for the redesign of parallel models of metaheuristics to allow solving of large scale optimization problems on GPU architectures. Our objective is to rethink the existing parallel models and to enable their deployment on GPUs. In this purpose, very efficient approaches are proposed for CPU-GPU data transfer optimization, thread control or memory management. These approaches have been exhaustively experimented using five optimization problems and four GPU configurations. Compared to a CPU-based execution, experiments report up to 80-fold acceleration for large combinatorial problems and up to 2000-fold speed-up for a continuous problem.

Keywords: parallel metaheuristics, GPU computing, local search, evolutionary algorithms.