CHAPTER

GPU computing applied to linear and mixed-integer programming

10

V. Boyer[1], **D. El Baz**[2], **M.A. Salazar-Aguilar**[1]

Graduate Program in Systems Engineering, Universidad Autónoma de Nuevo León, Mexico[1]

LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France[2]

1 INTRODUCTION

Graphics processing units (GPUs) are many-core parallel architectures that were originally designed for visualization purposes. Over the past decade, they evolved toward becoming powerful computing accelerators for high-performance computing (HPC).

The study of GPUs for HPC applications presents many advantages:

- GPUs are powerful accelerators featuring thousands of computing cores.
- GPUs are widely available and relatively cheap devices.
- GPUs accelerators require less energy than classical computing devices.

Tesla NVIDIA computing accelerators are currently based on Kepler and Maxwell architectures. The recent versions of Compute Unified Device Architecture (CUDA), such as CUDA 7.0, coupled with the Kepler and Maxwell architectures facilitate the dynamic use of GPUs. Moreover, data transfers can now happen via high-speed network directly from any GPU memory to any other GPU memory in any other cluster without involving the assistance of the CPU. At present efforts are placed on maximizing the GPU resources and fast data exchanges between host and device. In 2016 the PASCAL architecture should feature more memory, 1 TB/s memory bandwith, and twice as many flops as Maxwell. NVLink technology will also permit data to move 5–10 times faster between GPUs and CPUs than with current PCI-Express, making GPU computing accelerators very efficient devices for HPC. Looking at the GPU computing accelerators previously released (some of which are presented in Table 1, which also summarizes the characteristics of GPUs considered in this chapter), we can measure the progress accomplished during one decade.

Advances in GPU Research and Practice. http://dx.doi.org/10.1016/B978-0-12-803738-6.00010-0

Table 1 Overview of NVIDIA GPUs Quoted in the Chapter (see http://www.nvidia.com for More Details)

GPU	# Cores	Clock (GHz)	Memory (GB)
GeForce 7800 GTX	24	0.58	0.512
GeForce 8600 GTX	32	0.54	0.256
GeForce 9600 GT	64	0.65	0.512
GeForce GTX 260	192	1.4	0.9
GeForce GTX 280	240	1.296	1
GeForce GTX 285	240	1.476	1
GeForce GTX 295	240	1.24	1
GeForce GTX 480	480	1.4	1.536
Tesla C1060	240	1.3	4
T10 (Tesla S1070)	240	1.44	4
C2050	448	1.15	3
K20X	2688	0.732	6

GPUs have been widely applied to signal processing and linear algebra. The interest in GPU computing is now widespread. Almost all domains in science and engineering are interested, for example, astrophysics, seismic studies, the oil industry, and the nuclear industry (e.g., see [1]). Most of the time GPU accelerators lead to dramatic improvements in the computation time required to solve complex practical problems. It was quite natural for the operations research (OR) community, whose field of interest has prolific difficult problems, to be interested in GPU computing.

Some works have attempted to survey contributions on a specific topic in the OR field. Brodtkorb et al. [2] and Schulz et al. [3] deal with routing problems. Van Luong [4] considers metaheuristics on GPUs. More generally, Alba et al. [5] study parallel metaheuristics.

In this chapter, we present an overview of the research contributions for GPU computing applied to OR. Each section contains a short introduction and useful references for the algorithm implementations. The chapter is directed toward researchers, engineers, and students working in the field of OR who are interested in the use of GPU to accelerate their optimization algorithms. This work will also help readers identify new areas for research in this field.

The organization of this chapter is as follows: Section 2 introduces the field of OR. The primary optimization algorithms implemented via GPU computing in the domain of OR are described in Section 3. Section 4 presents relevant metaheuristics that have been developed with GPU computing. Finally, some conclusions and future research lines are discussed in Section 5.

2 OPERATIONS RESEARCH IN PRACTICE

OR can be described as the application of scientific and especially mathematical methods to the study and analysis of problems involving complex systems. It has been used intensively in business, industry, and government realms. Many new analytical methods have evolved, such as mathematical programming, simulation, game theory, queuing theory, network analysis, decision analysis, and multicriteria analysis, these methods have a powerful application to practical problems with the appropriate logical structure.

Most of the problems OR tackles are messy and complex, often entailing considerable uncertainty. OR can use advanced quantitative methods, modeling, problem structuring, simulation, and other analytical techniques to examine assumptions, facilitate an in-depth understanding, and determine practical actions.

Many decisions currently are formulated as mathematical programs that require the maximization or minimization of an objective function subject to a set of constraints. A general representation of an optimization problem is

$$\max f(x) \tag{1}$$

$$\text{s.t.} \quad x \in \mathcal{D} \tag{2}$$

where $x = (x_1, x_2, \ldots, x_n)$, $n \in \mathbb{N}$, is the vector of decision variables, Eq. (1) is the objective function, and Eq. (2) imposes that x belongs to a defined domain $\mathcal{D}$. A solution x^* is said to be feasible when $x^* \in \mathcal{D}$ and x^* is optimal when $\forall x \in \mathcal{D}$, $f(x^*) \geq f(x)$.

When the problem is linear, the objective function is linear, and the domain $\mathcal{D}$ can be described by a set of linear equations. In this case, it exists $p = (p_1, p_2, ..., p_n)$ called the vector of profits such that $f(x) = p^T \cdot x$, and there exist a matrix $A \in \mathbb{R}^n \times \mathbb{R}^m$, $m \in \mathbb{N}$, and a vector $b \in \mathbb{R}^m$ such that $x \in \mathcal{D} \Leftrightarrow Ax = b$. Hence a linear program has the following general form:

$$\max p^T x \tag{3}$$

$$\text{s.t.} \quad Ax = b \tag{4}$$

The relationships among the objective function, constraints, and decision variables determine how hard it is to solve and the solution methods that can be used for optimization. There are different classes of linear optimization problems according to the nature of the variable x: linear programming (LP) (x is continuous), mixed-integer programming (a part of the decision variables in x should be integer), combinatorial problem (the decision variables can take only 0–1 values), and so on. No single method or algorithm works best on all classes of problems.

LP problems are generally solved with the simplex algorithm and its variants (see [6]). A basis solution is defined such that $x = (x_B, x_H)$ and $A \cdot x = A_B x_B + A_H x_H$, where $A_B = \mathbb{R}^n \times \mathbb{R}^n$ and $det(A_B) \neq 0$. In this case, it is $x_B = A_B^{-1} b$ and $x_H = 0$.

The principle of the simplex algorithm is to build at each iteration a new basis solution that improves the current objective value $p^T x$ by swapping one variable in x_B with one in x_H, until no further improvement is possible.

Mixed-integer programming and combinatorial problems are generally much harder to solve because, in the worst-case scenario, all possible solutions for x should be explored in order to prove optimality. The Branch-and-Bound (B&B) algorithm is designed to explore these solutions in a smart way by building an exploration tree where each branch corresponds to a subspace of solutions. For instance, in combinatorial optimization, two branches can be generated by fixing a variable to 0 and 1. During the exploration, the encountered feasible solution is used to eliminate branches in the tree through bounding techniques. They consist of evaluating the best solution that can be found in a subspace (relaxing the integrality of the variables and solving the resulting linear subproblem with the simplex is commonly used).

Metaheuristics are designed to tackle complex optimization problems where other optimization methods failed to provide a good feasible solution in a convenient processing time. These methods are now recognized as one of the most practical approaches for solving many complex problems, and this is particularly true for many real-world problems that are combinatorial in nature (see [7]). Simulated Annealing (SA), Tabu Search (TS), Scatter Search (SS), Genetic Algorithms (GAs), Variable Neighborhood Search (VNS), Greedy Randomized Adaptive Search Procedure (GRASP), Adaptive Large Neighborhood Search (ALNS), and Ant Colony Optimization (ACO) are some of the most widely used metaheuristics.

The purpose of a metaheuristic is to find an optimal or near optimal solution without a guarantee of optimality in order to save processing time. These algorithms generally start from a feasible solution obtained through a constructive method and then try to improve it by exploring one or more defined neighborhoods. A neighborhood is composed of all solutions that are obtained by applying a specific change (move) in the current solution. So the goal of the exploration is to find better solutions than the current one. This process can be repeated until a stopping criterion is reached. In order to reinforce the search process, sometimes multiple initial solutions are generated and explored in parallel, such as GA and ACO, and information is exchanged between these solutions in order to improve the convergency of the approach.

The solutions for real-world problems represented as mathematical programs (optimization problems) are often hindered by size. In mathematical programming, size is determined by the number of variables, the number and complexity of the constraints, and objective functions. Hence the methods for solving optimization problems tend to be complex and require considerable numerical effort. By developing specialized solution algorithms to take advantage of the problem structure, significant gains in computational efficiency and a reduction in computer memory requirements may be achieved. Hence, practitioners and researchers have concentrated their efforts on developing optimization algorithms that exploit the capabilities of GPU computing.

In the literature related to GPU computing applied to OR, two main classes of optimization problems have been studied: LP problems and mixed-integer programming problems. For solving linear optimization problems, the simplex method is by far one of the most widely used exact methods, and for mixed-integer optimization problems the B&B method is the most common exact method. For solving different mixed-integer optimization problems metaheuristics like TS, GA, ACO, and SA have been proposed by using GPU computing, and their high performance is remarkable with respect to their sequential implementation.

3 EXACT OPTIMIZATION ALGORITHMS

In this section, the GPU implementation of exact optimization methods is presented. These methods are essentially the simplex, the dynamic programming, and the B&B algorithms (see [8]). Because these algorithms should follow a strict scheme to guarantee optimality, and tend to have a tree structure, their implementation on GPUs is particularly challenging. Research in this area mainly focuses on data arrangement for coalesced memory accesses or on speeding part of the algorithm on GPUs.

3.1 THE SIMPLEX METHOD

Originally designed by Dantzig [9], the simplex algorithm and its variants (see [6]) are largely used to solve LP problems. Basically, from an initial feasible solution, the simplex algorithm tries, at each iteration, to build an improved solution while preserving feasibility until optimality is reached. Although this algorithm is designed to solve LPs, it is also used to solve the linear relaxation of mixed-integer problems (MIPs) in many heuristics and exact approaches like the B&B. Furthermore, it is known that in algorithms like the B&B, the major part of the processing time is spent in solving these linear relaxations. Hence, faster simplex algorithms benefit to all fields of OR. Table 2 summarizes the contributions related to GPU implementations of simplex algorithms that can be found in the literature.

The first GPU implementation of a simplex algorithm, that is, the revised simplex method, was proposed by Greeff [17] in 2005. Most of the GPU computing drawbacks encountered by Greeff at that time have been addressed since then, with the development of the GPU architecture and CUDA. However, in this early work, a speedup of 11.5 was achieved as compared with an identical CPU implementation.

Simplex algorithm, like the revised simplex algorithm, involves many operations on matrices, and many authors have tried to take advantage of recent advances in LP. Indeed, some well-known tools like BLAS (Basic Linear Algebra Subprograms) or MATLAB have some of their matrix operations, such as inversions or multiplication, implemented in GPU. Spampinato and Elster [16], with cuBLAS (https://developer.nvidia.com/cublas), achieved a speedup of 2.5 for problems with 2000 variables

Table 2 Linear Programming and GPUs

Algorithm	Reference
The Simplex Tableau	[10,11]
The Two-Phase Simplex	[12]
The Revised Simplex	[13]
	[14]
	[15]
	[16]
	[17]
The Interior Point Method	[18]
The Exterior Point Method	[13]

and 2000 constraints when comparing their GPU implementation on a NVIDIA GeForce GTX 280 GPU to the ATLAS-based solver [19] on an Intel Core 2 Quad 2.83 GHz processor. Ploskas and Samaras [14] proposed an implementation based on MATLAB and CUDA environments and reported a speedup of 5.5 with an Intel Core i7 3.4 GHz and a NVIDIA Quadro 6000 with instances of up to 5000 variables and 5000 constraints.

In order to improve the efficiency of their approach, Ploskas and Samaras [14] did a complete study on the basis update for the revised simplex method. They proposed a GPU implementation of the Product Form of the Inverse (PFI) from Dantzig and Orchard-Hays [20] and of Modification of the PFI (MPFI) from Benhamadou [21]. Both approaches tend to reduce the computation effort of the matrices operations. Their results showed that PFI is slightly better than MPFI.

Ploskas and Samaras [13] presented a comparison of GPU implementations of the revised simplex and the exterior point method. In the exterior point method, the simplex algorithm can explore infeasible regions in order to improve the convergence of the algorithm. They also used the MATLAB environment for their implementation and compared their results to the MATLAB large-scale linprog built-in function. All the main phases of both algorithms are performed on GPUs. The experimental tests carried out with some instances of the netlib benchmark and a NVIDIA Quadro 6000 show that the exterior point method outperforms the revised simplex method with a maximum speedup of 181 on dense LPs and 20 on the sparse ones.

Bieling et al. [15] proposed some algorithm optimizations for the revised simplex used by Ploskas and Samaras [13]. They used the steepest-edge heuristic from Goldfarb and Reid [22] to select the entering variables and an arbitrary bound process in order to select the leaving variables. The authors compared their results to the GLPK solver (http://www.gnu.org/software/glpk/) and reported a reduction in computation time by a factor of 18 for instances with 8000 variables and 2700 constraints on a system with an Intel Core 2 Duo E8400 3.0 GHz processor and a NVIDIA GeForce 9600 GT GPU.

Like Bieling et al. [15] showed, sometimes controlling all the implementation of the algorithm can lead to better performance. The simplex tableau algorithm is very appealing in this context. Indeed, in this case, data are organized in a table structure that fits particularly well to the GPU architecture. Lalami et al. [10,11] and Meyer et al. [12] proposed two implementations of this algorithm, on one GPU and on multi-GPUs, and they reported that both algorithms reached a significant speedup.

Lalami et al. [11] used the algorithm of Garfinkel and Nemhauser [23], which improved the algorithm of Dantzig by reducing the number of operations and the memory occupancy. They extended this implementation to the multi-GPU context in Lalami et al. [11]. They adopted a horizontal decomposition where the constraints, that is, the lines in the tableau, are distributed on the different GPUs. Hence, each GPU updates only a part of the tableau, and the work of each GPU is managed by a distinct CPU thread. For their experimental tests, they used a server with an Intel Xeon E5640 2.66 GHz CPU and two NVIDIA C2050 GPUs, and instances with up to 27,000 variables and 27,000 constraints. They observed a maximum speedup of 12.5 with a single GPU and 24.5 with two GPUs.

Meyer et al. [12] proposed a multi-GPU implementation of the two-phase simplex. The authors consider a vertical decomposition of the simplex tableau, that is, the variables are distributed among the GPUs, in order to have less communications between GPUs. Like in Lalami et al. [10,11], they considered the implementation of the pivoting phase and the selection of the entering and leaving variables. They used a system with two Intel Xeon X5570 2.93 GHz processors and four NVIDIA Tesla S1070. They solved instances with up to 25,000 variables and 5000 constraints and showed that their approach outperforms the open-source solver CLP (https://projects.coin-or.org/Clp) of the COIN-OR project.

Jung and O'Leary [18] studied the implementation of the interior point method. They proposed a mixed precision hybrid algorithm using a primal-dual interior point method. The algorithm is based on a rectangular-packed matrix storage scheme and uses the GPU for computationally intensive tasks such as matrix assembly, Cholesky factorization, and forward and backward substitution. However, computational tests showed that the proposed approach does not clearly outperform the sequential version on CPU because of the data transfer cost and communication latency. To the best of our knowledge, it is the only interior point method that has been proposed in the literature even though it is one of the most effective for sequential implementations.

3.2 DYNAMIC PROGRAMMING

The Dynamic Programming algorithm was introduced by Bellman [24]. The main idea of this algorithm is to solve complex problems by decomposing them into smaller problems that are iteratively solved. This algorithm has a natural parallel structure. An overview of the literature dealing with the implementation of Dynamic Programming on GPUs can be found in Table 3. As we can see, only Knapsack

Table 3 Dynamic Programming on GPUs

Algorithm	Problem	Reference
Dense Dynamic Programming	Knapsack	[25,26]
Dense Dynamic Programming	Multi-Choice Knapsack	[27]

Problems (KP) have been studied so far. We provide more details on these contributions in the following section.

3.2.1 Knapsack problems

The KP (see [28]) is one of the most studied problems in OR. It consists of selecting a set of items that are associated with a profit and a weight. The objective is to maximize the sum of the profits of the chosen items without exceeding the capacity of the knapsack. In this context, the dynamic algorithm starts to explore the possible solutions with a capacity equal to zero and increases the capacity of the knapsack by one unit at each iteration until the maximum capacity is reached.

Boyer et al. [25] proposed a hybrid dense dynamic programming algorithm on a GPU. Data were organized in a table where the columns represented the items and the raws, represent the capacity of the knapsack in increasing order. At each iteration, a raw was filled based on information provided by the previous one. The authors also proposed a data compression technique in order to deal with the high memory requirements of the approach. This technique permits one to reduce the memory occupancy needed to reconstruct the optimal solution and the amount of data transferred between the host and the device. Computational experiments were carried out on a system with an Intel Xeon 3.0 GHz and a NVIDIA GTX 260 GPU, and randomly generated correlated problems with up to 100,000 variables were considered. A reduction in computation time by a factor of 26 was reported, and the reduction in memory occupancy appeared to be more efficient when the size of the problem increased, while the overhead did not exceed 3% of the overall processing time.

Boyer et al. [26] extended their approach whereby a multi-GPU hybrid implementation of the dense dynamic programming method was proposed. The solution presented is based on multithreading and the concurrent implementation of kernels on GPUs: each kernel is associated with a given GPU and managed by a CPU thread; the context of each host thread is maintained all along the application, that is, host threads are not killed at the end of each dynamic programming step. This technique also tends to reduce data exchanges between the host and the devices. A load balancing procedure was also implemented in order to maintain efficiency of the parallel algorithm. Computational experiments, carried out on a machine with an Intel Xeon 3 GHz processor and a Tesla S1070 computing system, showed a speedup of 14 with

one GPU and 28 with two GPUs, without any data compression techniques. Strongly correlated problems with up to 100,000 variables were considered.

3.2.2 Multiple-choice knapsack problem

Suri et al. [27] studied a variant of the KP, called the multiple-choice KP (see [28]). In this case, the items are grouped in subsets and exactly one item of each subset is selected without exceeding the capacity of the knapsack. Their dynamic programming algorithm is similar to the one of Boyer et al. [25,26]; however, in order to ensure high processor utilization, multiple cells of the table are computing by one GPU thread.

They reported an important speedup of 220 as compared to a sequential implementation of the algorithm and a speedup of 4 compared to a CPU multicore one. Furthermore, they showed that their implementation outperformed the one of Boyer et al. [25,26] on randomly generated instances of the multichoice KP. For their experimental tests, they used two Intel Xeon E5520 CPUs and a NVIDIA Tesla M2050. However, no information was given on the memory occupancy of their algorithm.

3.3 BRANCH-AND-BOUND

The B&B algorithm was been designed to explore in a smart way the solution space of an MIP. In the original problem, the B&B generated new nodes that corresponded with subproblems obtained by fixing variables or adding constraints. Each node generated in a similar way other nodes and so on until the optimality condition was reached. The tree structure of the B&B is irregular and generally leads to branching performance issues with a GPU; thus implementing this algorithm on a GPU is in many cases a challenge.

To the best of our knowledge, as shown in Table 4, three types of problems were solved by a B&B GPU implementation: KP, Flow-shop Scheduling Problems (FSP) (see [29]), and a Traveling Salesman Problem (TSP) (see [30]). Two parallel approaches have been proposed:

- either MIP is entirely solved on GPU(s) through a specific or adapted parallel algorithm
- or GPUs are used to accelerate only the most time-consuming activities or parts of codes.

3.3.1 Knapsack problem

Boukedjar et al. [31], Lalami et al. [32], and Lalami [33] studied the GPU implementation of the B&B algorithm for KPs. In this algorithm, the nodes are first generated sequentially on the host. When their number reached a threshold, the GPU is then used to explore the nodes in parallel, that is, one node per GPU thread. Almost all the phases of the algorithm are implemented on the device, that is, bounds computation, generation of the new nodes, and updates of the best lower bound found via atomic

Table 4 Branch-and-Bound on a GPU

Algorithm	Problem	Reference
Branch-and-Bound	Knapsack	[31]
		[32]
		[33]
Branch-and-Bound	Flow-Shop Scheduling	[34]
		[35,36]
		[37]
Branch-and-Bound	Traveling Salesman	[38]

operations. Parallel bounds comparison and identification of nonpromising nodes are also performed on the GPU. At each step, a concatenation of the list of nodes is performed on the CPU. An Intel Xeon E5640 2.66 GHz processor and a NVIDIA C2050 GPU were used for the computational tests. The authors reported a speedup of 52 in Lalami [33] for strongly correlated problems with 1000 variables.

3.3.2 Flow-shop scheduling problem

The solution of the FSP via parallel B&B methods using GPU was studied by Melab et al. [37], Chakroun et al. [34], and Chakroun et al. [35]. In this problem, a set of jobs has to be scheduled on a set of available machines. In Melab et al. [37] and Chakroun et al. [35], the authors identified that 99% of the time spent by the B&B algorithm is in the bounding process. Hence they focused their efforts on parallelizing this operator on a GPU, and eliminating, selecting, and branching were carried out by the CPU. Indeed, at each step in the tree exploration of the B&B, a pool of subproblems is selected and sent to the GPU which performs, in parallel, the evaluation of their lower bound through the algorithm proposed by Lageweg et al. [39].

Furthermore, in order to avoid divergent threads in a warp resulting from conditional branches, the authors proposed a branch refactoring which involves rewriting the conditional instructions so that threads of the same warp execute an uniform code (see Table 5).

Computational experiments were carried out on a system with an Intel Xeon E5520 2.27 GHz and a NVIDIA C2050 computing system. Some instances of FSP proposed by Taillard [40] and a maximum speedup factor of 77 was observed for instances with 200 jobs on 20 machines as compared to a sequential version. This approach was extended in Chakroun et al. [34] to the multi-GPU case where a maximum speedup of 105 was reported with two Tesla T10.

Finally, Chakroun et al. [36] considered a complete hybrid CPU-GPU implementation to solve the FSP, where CPU cores and GPU cores cooperate in parallel for the exploration of the nodes. Based on the results obtained in their previous work, they added the branching and the pruning operator on the GPU to the bounding operator. Two approaches were then considered: first, a concurrent exploration of the B&B tree, where a pool of subproblems is partitioned between the CPU cores; second, a

Table 5 Branch Refactoring From Chakroun et al. [35]

Original Condition	Branch Refactoring
if $(x \neq 0)$ $a = b$;	$int\ coef = \_\_\cos f(x)$;
else $a = c$;	$a = (1 - coef) \times b + coef \times c$;
if $(x > y)$ $a = b$;	$int\ coef = \min(1, \_\_\exp f(x - y - 1))$;
(x and y are integers)	$a = coef \times b + (1 - coef) \times a$;

cooperative exploration of the B&B tree, where CPU threads handle a part of the pool of the subproblems to explore on the GPU, which allows interleaving and overlapping data transfer through asynchronous operations. The pool of subproblems to explore is determined dynamically with a heuristic according to the instance being solved and the GPU configuration.

With an Intel Xeon E5520 2.27 GHz and a NVIDIA C2050, on the instances of Taillard [40], they achieved an acceleration of 160 with the cooperative approach. Indeed, the cooperative approach appeared to be 36% faster than the concurrent one. From these results, the authors recommended to using the GPU cores for the tree exploration and the CPU cores for the preparation and the transfer of data.

3.3.3 Traveling salesman problem

Carneiro et al. [38] considered the solution of the TSP on a GPU. The TSP consists of finding the shortest route that a salesman will follow to visit all his customers. At each step of their B&B algorithm, a pool of pending subproblems is sent to the GPU. A GPU thread processed the exploration of the resulting subtree through a depth-first strategy with backtracking. This strategy permitted them to generate only one child at each iteration, and the complete exploration of the subspace of solution is ensured through the backtracking process. The process is repeated until all pending subproblems are solved. Hence, in this approach, the GPU explores in parallel different portions of the solution space. As compared to an equivalent sequential implementation, Carneiro et al. [38] reported a maximum speedup of 11 on a system with an Intel Core i5 750 2.66 GHz and a NVIDIA GeForce GTS 450. The authors used randomly generated instances of an asymmetric TSP for up to 16 cities.

4 METAHEURISTICS

A metaheuristic is formally defined as an iterative generation process that guides a subordinate heuristic by logically combining different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions (see [41]).

GPU implementations of metaheuristics have received particular attention from practitioners and researchers. Unlike exact optimization procedures, metaheuristics allow high flexibility on their design and implementation, and they are usually easier to implement. However, they are approximate methods that sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a short computation time. In this chapter, we discuss the most relevant metaheuristics (GA, ACO, TS, among others) that have been implemented under a GPU architecture.

4.1 GENETIC ALGORITHMS

As shown in Table 6, GAs and their variants on GPUs have been proposed in the literature for the solution of complex optimization problems. GAs try to imitate the natural process of selection. GAs are based on three main operators:

- Selection (a subset of the population is selected in order to generate the new generation)
- Crossover (a pair of parents are recombined in order to produce a child)
- Mutation (the initial gene of an individual is partially or entirely altered in order to provide diversification)

At the beginning of the algorithm, a population is created. Each individual in the population represents a solution of the problem to solve. At each iteration, a subset of the individuals are selected according to a fitness function, and a new population is created through the crossing operator. The mutation operator is then applied in order to bring diversity into the search space.

4.1.1 The traveling salesman problem

Li et al. [47] and Chen et al. CDJN2011 proposed a Fine-Grained Parallel GA on a GPU in order to solve the well-known TSP. In the approach of Li et al. [47], a tour is assigned to a block of GPU threads, and each thread within this block is associated to a city. All the operators are treated on the GPU. In particular, they use

Table 6 Genetic Algorithms on GPU

Algorithm	Problem	Reference
Cellular Genetic Algorithm	Independent Tasks Scheduling	[42,43]
Systolic Genetic Search	Knapsack	[44]
Island-Based Genetic Algorithm	Flow-Shop Scheduling	[45]
Genetic Algorithm	Traveling Salesman	[46]
Immune Algorithm	Traveling Salesman	[47]

a partially mapped crossover method (see [48]), and the selection of the parents is done via an *adjacency-partnership method*; however, no details have been provided on this process. The tournament selection was preferred to the classic roulette-wheel selection. On a GeForce 9600GT, they reported an acceleration between 2.42 and 11.5, on instances with up to 226 cities.

Chen et al. [46] used an order crossover operator where parents exchange their sequence orders in a portion of their chromosome, which prevents a city from appearing more than once in a solution. Furthermore, they implemented the *2-opt mutation operator*, which seems to be particularly adapted to the TSP. They also used a simple selection process, that preserves the best chromosome. Because of the need for synchronization at each step of their GA, they carried out experimental tests with only one block on a Tesla C2050. Indeed, we recall that within the same block during a kernel call, threads can be synchronized, which is not possible with threads belonging to different blocks. However, it is possible to synchronize all the blocks through multiple kernel calls. Although they did not explore all the capability of computation of their GPU, they showed that their parallel GA on the GPU outperforms the sequential one on an Intel Xeon E5504.

4.1.2 Scheduling problems

Pinel et al. [42] proposed a fine-grained parallel GA for a scheduling problem, that is, the Independent Tasks Scheduling Problem. In this variant of the FSP, no precedence relation is considered between the tasks. The proposed algorithm, called GraphCell, starts by building a good feasible solution using the Min-Min heuristic from Ibarra and Kim [49], and this solution is added to the initial population of a Cellular GA (see [50]). The two main steps of this algorithm are conducted in parallel on the GPU, that is, the search for the best machine assignment for each task and the update of the solution.

In the Cellular GA, the population is arranged in a 2D grid, and only individuals close to each others are allowed to interact. This approach, whereby one individual is managed by one thread, reduces the communications involved. Computational tests carried out by Pinel et al. [42] with a Tesla C2050 on randomly generated instances with up to 65,536 tasks and 2048 machines showed the following:

- The Min-Min heuristic on GPUs outperformed the parallel implementation on CPUs (two Intel Xeon E5440 processors with 2×4 cores).
- The Cellular GA was able to improve, in the first generation, up to 3% the initial solution provided by the Min-Min heuristic.

In Zajíček and Šucha [45], a parallel island-based GA is implemented on a GPU for the solution of the FSP. In this variant of the GA, the population is divided in multiple subpopulations isolated on an island in order to preserve genetic diversity. These populations could share genetic information through an operator of migration. Zajíček and Šucha performed evaluations, mutations, and crossovers of individuals

in subpopulations in parallel and independently from other populations, and they reported a speedup of 110 on a Tesla C1060.

4.1.3 Knapsack problems

Pedemonte et al. [44] proposed a Systolic Genetic Search (SGS) for the solution of the KP using GPU. The population is arranged in a 2D toroidal grid of cells, and at each iteration, solutions transit horizontally and vertically in a determined direction within the grid. This communication scheme is based on the model of systolic computation from Kung [51] and Kung and Leiserson [52], that is, the synchronous circulation of data through a grid of processing units.

Cells are in charge of the crossover and mutation operators, the fitness function evaluation, and the selection operator. The authors associated a block of GPU threads with a cell of the grid.

Experiments were carried out on a system with a GeForce GTX 480 GPU. Problems without correlation and up to 1000 variables were considered. Experimental results showed that the SGS method produced solutions of very good quality and that the reduction in computation time ranged from 5.09 to 35.7 according to the size of the tested instances.

4.2 ANT COLONY OPTIMIZATION

In this section, we focus on ant colony approaches that have been receiving a particular attention in the literature for the solution of routing problems. As we can see in Table 7, to the best of our knowledge, no other class of problems has been addressed in the literature on GPUs with ant colony.

ACO (see [53]) is an other population-based metaheuristic for solving complex optimization problems. This algorithm mimics the behavior of ants searching for a path from their colony to a point of interest (food). It comprises two main operators; that is, the constructive operator and the pheromone update operator. Artificial ants are used to construct solutions by considering pheromone trails that reflect the search procedure.

The first implementation of the ACO was due to Catala et al. [61] for the solution of the orienteering problem (OP), also known as the selective TSP (see [62]). In this variant of the TSP, the visits are optional, and each customer has a positive score that is collected by the salesman if and only if the customer is visited. Hence the OP consists of finding a route that maximizes the total collected score within a time limit.

The authors proposed to arrange the path followed by each ant in a 2D table where each row is associated with an ant, and a column represents the position of a node in the ant's path. The attraction between a pair of nodes (or pheromones) is stored also in a table. In order to build the paths for the ants, the authors used a *selection by projection* based on the principle of an orthographic camera clipping a space to determine, in parallel, the next node to visit. The results obtained showed that their

Table 7 Ant Colony Algorithms on GPU

Algorithm	Problem	Reference
Ant Colony Optimization	Transit Stop Inspection and Maintenance Scheduling	[54]
Ant Colony Optimization	Traveling Salesman	[55]
Max-Min Ant System	Traveling Salesman	[56]
Ant Colony Optimization	Traveling Salesman	[57]
Max-Min Ant System	Traveling Salesman	[58]
Max-Min Ant System	Traveling Salesman	[59]
Max-Min Ant System	Traveling Salesman	[60]
Ant Colony Optimization	Orienteering	[61]

approach, implemented on a single GeForce 6600 GT GPU, is competitive with a parallel ACO running on a GRID with up to 32 nodes.

Cecilia et al. [57] studied different strategies for the GPU implementation of the constructive operator and the pheromone update operator involved in the ACO. Each ant is associated with a block of threads, and each block thread represents a set of nodes (customers) to visit. Hence the parallelism in the tour constructor phase is improved and warp divergence is reduced. They also proposed to use a scatter-to-gather transformation (see [63]) for the pheromone update, in place of built-in atomic operations. Their results obtained with a C1060 GPU with instances with up to 2396 nodes showed a speedup of 25.

Uchida et al. [55] proposed an extensive study on strategies to accelerate the ACO on a GPU. In particular, they studied different selection methods for the construction operator to determine randomly the next city to visit. In their implementation a thread is associated to a city which computes its fitness value. Then a random number is generated and a city is selected based on a roulette-wheel scheme. The proposed methods differ in how to avoid selecting a node already visited by using the prefix-sum algorithm (see [64]), eliminating them through a compression method or by stochastic trial. They also studied the update of the pheromones through the shared memory in order to avoid uncoalesced memory access. The computational tests carried out with a GTX580 and a set of benchmark instances from the TSPLIB (see [65]) showed that the efficiency of the proposed approaches depends on the number of visited cities, so they proposed a hybrid approach. A maximum speedup of 22 was reported.

The solution of the transit stop inspection and maintenance scheduling problem was presented by Kallioras et al. [54]. In this problem, the transit stops need to be grouped in districts, and the visits, of the transit stops, for each vehicle within a district need be scheduled. They proposed a hybrid CPU-GPU implementation where the length of the ant's path, the pheromone update, addition, and comparison operations are performed on the GPU. The implementation was not detailed in the paper, and they reported a speedup of 21 with a GTX 660M.

Reference was also made to You [66] and Li et al. [67] who also studied the GPU implementation of ACO on a GPU. However, very few details on the implementation are provided in their published article.

4.2.1 Max-min ant system

The max-min ant system (MMAS) (see [68]) is a variant of the ACO. It adds the following features to the classic ACO:

- Only the best ants are allowed to update the pheromone trails.
- Pheromone trail values are bounded to avoid premature convergences.
- It can be combined with a local search algorithm.

Jiening et al. [60] and Bai et al. [59] proposed the first implementations of the MMAS on a GPU for the solution of TSP. In Jiening et al. [60], only the tour construction stage is processed on a GPU, while in Bai et al. [59], the pheromone update on a GPU was also studied. In these works, the reported speedup did not exceed 2.

Fu et al. [58] used the Jacket toolbox, which connects MATLAB with GPU, for their implementation of MMAS on a GPU. Ants share only one pseudorandom number matrix, one pheromone matrix, one tabu matrix, and one probability matrix in order to reduce communication between the CPU and GPU. Furthermore, they presented a variation of the traditional roulette-wheel selection, that is, the *All-In-Roulette*, which appears to be more adapted to the GPU architecture. With their approach, they achieved a speedup of 30 with a Tesla C1060 GPU, and computational tests were carried out on a system with an Intel i7 3.3 GHz processor and a NVIDIA Tesla C1060 GPU, based on benchmark instances, with up to 1002 cities. They also showed that the solution obtained was close to the one provided by their sequential algorithm.

More recently, Delévacq et al. [56] presented an MMAS for the parallel ant and the multiple colony approaches. In this paper, the authors discussed extensively the drawbacks encountered in such an implementation and proposed solutions based on previous works. In particular, they used the Linear Congruential Generator as proposed by Yu et al. [69] and a GPU 3-opt local search to improve solution quality. Furthermore, the authors compared different GPU implementations where ants were associated with one GPU thread or with one GPU block of threads, also when considering multiple colonies distributed among the GPU block. In experiments conducted with two GPUs of a NVIDIA Fermi C2050 server, on benchmark instances with up to 2396 cities, showed not only a maximum speedup of 24 but also preservation of the quality of the reported solution. These results were obtained when multiple colonies were considered combined with a local search strategy.

4.3 TABU SEARCH

As we can see in Table 8, TS approaches have also been used extensively for the solution of scheduling problems. TS, created by Glover [70,71], uses local

Table 8 Tabu Search on GPU

Algorithm	Problem	Reference
Tabu Search	Resource Constrained Project Scheduling	[72,73]
Tabu Search	Permutation Flow-Shop Scheduling	[74]
Tabu Search	Traveling Salesman Problem/Flow-Shop Scheduling	[75]

search approaches to find a good solution for a problem. From an initial solution, it iteratively selects a new solution from a defined neighborhood. The neighborhood is updated according to the information provided by the new solution. Furthermore, in order to filter the search space, a tabu list is maintained, which corresponds to forbidden moves, such as the set of solutions explored in the previous iterations of the procedure.

The first reported GPU implementation of a TS algorithm was due to Janiak et al. [75] in 2008. This paper deals with the solution of the TSP and the Permutation Flow-Shop Scheduling Problem (PFSP), in which the sequence of operations at each machine should be the same. The authors defined a neighborhood on swap move, that is, by interchanging the position of two customers or two jobs in the current solution. A table of two dimensions is created on the GPU, which computes in each cell of coordinates (i, j) the new solution that results from swapping position i with j. All possible swap moves are then covered, and the CPU selects the best solution from the neighborhood according to the tabu list, which contains forbidden swap moves from previous iterations. This new solution is used to generate the new neighborhood, and so on, until the maximum number of iterations is reached. They used commercial GPUs (GeForce 7300 GT, GeForce 8600 GT, and GeForce 8800 GT) for their experimental tests and, with randomly generated instances, achieved a speedup of 4 with the PFSP and almost no speedup with the TSP.

Based on the results of Janiak et al. [75], who showed that 90% of the processing time is spent in the evaluation function, Czapiński and Barnes [74] designed an improved parallelization of TS for the PFSP. They used a limited parallelization of evaluations as proposed by Bożejko [76]. They proposed to reorder the way a solution is coded in the GPU to ensure a coalesced memory access. Furthermore, the evaluation of the starting time of each job on each machine is done through the use of the shared memory, in an iterative manner in order not to saturate this memory. The matrix of processing time of the jobs on each machine is also stored on the constant memory for faster memory access. Czapiński and Barnes [74] reached a maximum speedup of 89 with a Tesla C1060, based on instances from the literature, and showed that their implementation outperformed the one by Janiak et al. [75].

Bukata et al. [72] and Bukata and Šucha [73] subsequently presented a parallel TS method for the Resource Constrained Project Scheduling Problem (RCPSP). In

this variant of the FSP, in order to be executed, each job uses a renewable resource that is available in limited quantity. The used resource of a job is released at the end of its execution. Swap moves are used to define a new neighborhood; however, at each iteration, preprocessing is done to eliminate unfeasible swaps, that is, swaps that violated the precedence constraints on the jobs. Moreover, only a subset of the swaps is performed in order to limit the neighborhood size. The tabu list is represented by a 2D table where position (i, j) contains the value *false* if swapping position i with position j is permitted, or *true* otherwise. The authors also proposed some algorithmic optimization in order to update the starting time of each job and the resources consumed after a swap.

Bukata et al. [72] proposed to concurrently run a TS in each GPU block that manages its own incumbent solution. A list of incumbent solutions is maintained on the global memory that each block accesses through atomic operations. Hence blocks could cooperate through the exchange of solutions, and a diversification technique is processed when a solution is not improved after a certain number of iterations. Experiments were carried out on a server with a GTX 650, with a benchmark from the literature of up to 600 projects and 120 activities. The results were compared with a sequential and a parallel CPU version of their algorithm. They showed that the GPU version achieved a speedup of almost 2 compared to the parallel one without degrading the solution quality.

4.4 OTHER METAHEURISTICS

In addition to the metaheuristics presented in the previous sections, which have been widely studied by different authors, other metaheuristics have received less attention in the literature on GPU computing. In this section, we present these approaches. Table 9 gives an overview of the literature that is detailed in this section.

4.4.1 Deep greedy switching

The deep greedy switching (DGS) heuristic (see [81]) starts from a random initial solution and moves to better solutions by considering a neighborhood resulting from a restricted *2-exchange* approach. This algorithm was implemented on a GPU by Roverso et al. [80] for the solution of the Linear Sum Assignment Problem. This problem consists of assigning a set of jobs to a set of agents. An agent can perform

Table 9 Other Metaheuristics on GPU

Algorithm	Problem	Reference
Constructive Heuristic	Nurse Rerostering	[77]
Multiobjective Local Search	Multiobjective Flow-Shop Scheduling	[78,79]
Deep Greedy Switching	Linear Sum Assignment	[80]

only one job, and when it is performed, a specific profit is collected. The objective is to maximize the sum of the collected profits.

The authors focused on the neighborhood exploration, which is generated by swapping jobs between agents (*2-exchange operator*). Hence the evaluation of the new solution obtained after a swap was processed in parallel on the GPU. Computational experiments were carried out on a system with a NVIDIA GTX 295 GPU with randomly generated instances of up to 9744 jobs. The authors reported a reduction of computation time by a factor of 27.

4.4.2 Multiobjective local search

Luong et al. [78] and Luong [79] studied the implementation on GPUs of a multiobjective local search for the multiobjective FSP. The neighborhood exploration is done on GPU, and they consider different algorithms for the Pareto frontier estimation: an aggregated TS, where the objectives are aggregated in order to obtain a monoobjective problem, along with a Pareto Local Search Algorithms from Paquete and Stützle [82]. Furthermore, in order to overcome the noncoalesced accesses to the memory, they propose to use the texture memory of the GPU.

They carried out their experimental tests on problems ranging from 20 jobs and 10 machines to 200 jobs and 20 machines. They considered three objectives: the makespan, total tardiness, and number of jobs delayed with regard to their due date. With a GTX 480, they observed a maximum speedup of 16 times with the aggregated TS and 15.7 times with the Pareto local search algorithm.

4.4.3 Constructive heuristic

Zdeněk et al. [77] presented an implementation of the Constructive Heuristic of Moz and Pato [83], who proposed this heuristic to initialize their GA for the solution of the Nurse Rerostering Problem. This Constructive Heuristic proceeded as follows: from an original roster, a randomly ordered shift list is generated; then the current roster is cleared, and the shifts are assigned back to the modified roster one by one according to certain rules. In their heterogeneous model, the GPU is used to do the shift assignment, and the rest of the algorithm is performed on the CPU. Furthermore, multiple randomly ordered shift lists are generated in order to explore in parallel multiple shift reassignments and to take advantage of the GPU.

Experimental tests were carried out with a GTX 650, based on the instances from Moz and Pato [83]; up to 32 nurses with a planning horizon of 28 days were considered. They achieved a maximum speedup of 2.51 with an optimality gap between 4% and 18%.

5 CONCLUSIONS

The domain of OR is ripe with difficult problems. In this chapter, we concentrated on GPU computing in the OR field. In particular, we surveyed the major contributions to integer programming and LP. In many cases, significant reduction in computing time

was observed for OR problems. Nevertheless, it is difficult to establish a quantitative comparison between the different approaches quoted in this chapter because the reported results were obtained via different GPUs architectures. Therefore it is not possible to identify the best implementation for a given algorithm. Metrics that facilitate comparisons between the various parallel algorithms have not been designed and commonly accepted. Issues related to the quality of solutions must also to be taken into account.

As shown in this chapter, most of the classic OR algorithms have been implemented on GPUs. Exact methods have received less attention than metaheuristics, essentially because of their lack of flexibility. In order to achieve consequent speedup, the operators of the different metaheuristics have to be adapted to fit into the GPU architecture. We note for instance that important acceleration in GA and ACO were achieved by modifying the roulette-wheel selection. Also, the coding solutions and the way they are stored in the GPU memory play major roles in the performance of the algorithms. The main drawbacks of implementation come from the fact that we try to fit algorithms that are sequential by nature into a parallel architecture. Dedicated parallel OR algorithms need to be designed.

Because many applications include OR problems, the future of GPU computing seems very promising. New features such as dynamic parallelism (i.e., the possibility for GPU threads to automatically spawn new threads) simplify parallel programming and seem particularly suited to integer programming applications. The new NVIDIA Pascal architecture should feature more memory, 1 TB/s memory bandwidth, and twice as much FLOPS as the current Maxwell architecture. NVLink technology should also permit data to move 5–10 times faster than PCI-Express technology.

CUDA updates and OpenCL updates (or the recent OpenACC http://www.openacc-standard.org/) always tend to facilitate programming and improve efficiency of accelerators by hiding programming difficulties. We note in particular that OpenACC is a set of high-level programs that enables C/C++ and Fortran programmers to use highly parallel processors with much of the convenience of OpenMP.

In the future OR industrial codes will be able to benefit from accelerators like GPUs that are widely available and to propose attractive and fast solutions to customers. Nevertheless, an important challenge remains in the exact solution of industrial problems of significant size via GPUs.

ACKNOWLEDGMENTS

Dr Didier El Baz thanks NVIDIA Corporation for support. Dr Vincent Boyer and Dr M. Angélica Salazar-Aguilar thank the Program for Teachers Improvement (PROMEP) for support under the grant PROMEP/103.5/13/6644.

CONFLICTS OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

[1] H. Nguyen, GPU GEMS 3, Addison Wesley Professional, Santa Clara, CA, 2008.
[2] A.R. Brodtkorb, T.R. Hagen, C. Schulz, G. Hasle, GPU computing in discrete optimization. Part I: introduction to the GPU, EURO J. Transp. Logist. 2 (2013) 129–157.
[3] C. Schulz, G. Hasle, A.R. Brodtkorb, T.R. Hagen, GPU computing in discrete optimization. Part II: survey focused on routing problems, EURO J. Transp. Logist. 2 (2013) 159–186.
[4] T. Van Luong, Métaheuristiques parallèles Sur GPU, Ph.D. thesis, Université Lille 1—Sciences et Technologies, 2011.
[5] E. Alba, G. Luque, S. Nesmachnow, Parallel metaheuristics: recent advances and new trends, Int. Trans. Oper. Res. 20 (1) (2013) 1–48, ISSN 1475-3995.
[6] A. Schrijver, Theory of Integer and Linear Programming, Wiley, Chichester, 1986.
[7] S. Ólafsson, Chapter 21 metaheuristics, in: S.G. Henderson, B.L. Nelson (Eds.), Simulation, Handbooks in Operations Research and Management Science, vol. 13, Elsevier, 2006 pp. 633–654.
[8] W.L. Winston, J.B. Goldberg, Operations Research: Applications and Algorithms, vol. 3, Duxbury Press, Boston, 2004.
[9] G.B. Dantzig, Maximization of a linear function of variables subject to linear inequalities, in: Activity Analysis of Production and Allocation, Wiley and Chapman-Hall, 1951, pp. 339–347.
[10] M.E. Lalami, V. Boyer, D. El Baz, Efficient implementation of the simplex method on a CPU-GPU system, in: 25th IEEE International Parallel and Distributed Processing Symposium, Workshops and PhD Forum (IPDPSW), Workshop PCO'11, ISSN 1530-2075, 2011, pp. 1999–2006.
[11] M.E. Lalami, D. El Baz, V. Boyer, Multi GPU implementation of the simplex algorithm, in: 13th IEEE International Conference on High Performance Computing and Communications (HPCC 2011), 2011, pp. 179–186.
[12] X. Meyer, P. Albuquerque, B. Chopard, A multi-GPU implementation and performance model for the standard simplex method, in: Euro-Par 2011, 2011, pp. 312–319.
[13] N. Ploskas, N. Samaras, Efficient GPU-based implementations of simplex type algorithms, Appl. Math. Comput. 250 (2015) 552–570.
[14] P. Nikolaos, S. Nikolaos, A computational comparison of basis updating schemes for the simplex algorithm on a CPU-GPU system, Am. J. Oper. Res. 3 (2013) 497.
[15] J. Bieling, P. Peschlow, P. Martini, An efficient GPU implementation of the revised simplex method, in: 24th IEEE International Parallel Distributed Processing Symposium, Workshops and PhD Forum (IPDPSW 2010), 2010, pp. 1–8.
[16] D.G. Spampinato, A.C. Elster, Linear optimization on modern GPUs, in: 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS), ISSN 1530-2075, 2009, pp. 1–8.
[17] G. Greeff, The revised simplex algorithm on a GPU, Univ. of Stellenbosch, Tech. Rep., 2005.
[18] J.H. Jung, D.P. O'Leary, Implementing an interior point method for linear programs on a CPU-GPU system, Electron. Trans. Numer. Anal. 28 (2008) 174–189.

[19] R.C. Whaley, J. Dongarra, Automatically tuned linear algebra software, in: Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999, pp. 1–27.
[20] G.B. Dantzig, W. Orchard-Hays, The product form for the inverse in the simplex method, Math. Tables Other Aids Comput. 8 (1954) 64–67.
[21] M. Benhamadou, On the simplex algorithm "revised form", Adv. Eng. Software 33 (11) (2002) 769–777.
[22] D. Goldfarb, J.K. Reid, A practicable steepest-edge simplex algorithm, Math. Program. 12 (1) (1977) 361–371.
[23] R.S. Garfinkel, G.L. Nemhauser, Integer Programming, vol. 4, Wiley, New York, 1972.
[24] R. Bellman, Dynamic Programming, Princeton University Press, Princeton, NJ, 1957.
[25] V. Boyer, D. El Baz, M. Elkihel, Dense dynamic programming on multi GPU, in: 19th International Conference on Parallel, Distributed and network-based Processing (PDP 2011), ISSN 1066-6192, 2011, pp. 545–551.
[26] V. Boyer, D. El Baz, M. Elkihel, Solving knapsack problems on GPU, Comput. Oper. Res. 39 (1) (2012) 42–47, ISSN 0305-0548.
[27] B. Suri, U.D. Bordoloi, P. Eles, A scalable GPU-based approach to accelerate the multiple-choice knapsack problem, in: Design, Automation Test in Europe Conference Exhibition (DATE), ISSN 1530-1591, 2012, pp. 1126–1129.
[28] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, Inc., Chichester, UK, 1990.
[29] M.L. Pinedo, Scheduling: Theory, Algorithms, and Systems, Springer Science & Business Media, New York, 2012.
[30] G. Reinelt, The Traveling Salesman: Computational Solutions for TSP Applications, Springer-Verlag, Berlin, 1994.
[31] A. Boukedjar, M.E. Lalami, D. El Baz, Parallel branch and bound on a CPU-GPU system, in: 20th International Conference on Parallel, Distributed and Network-Based Processing (PDP 2012), ISSN 1066-6192, 2012, pp. 392–398.
[32] M.E. Lalami, D. El Baz, GPU implementation of the branch and bound method for knapsack problems, in: 26th IEEE International Parallel and Distributed Processing Symposium, Workshops and PhD Forum (IPDPSW), Workshop PCO'12, 2012, pp. 1769–1777.
[33] M.E. Lalami, Contribution à la résolution de problèmes d'optimisation combinatoire: méthodes séquentielles et parallèles, Ph.D. Thesis, Université Paul Sabatier, Toulouse, 2012.
[34] I. Chakroun, N. Melab, An adaptative multi-GPU based Branch-and-Bound. A case study: the flow-shop scheduling problem, in: IEEE 14th International Conference on High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012, pp. 389–395.
[35] I. Chakroun, M. Mezmaz, N. Melab, A. Bendjoudi, Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm, Concurr. Comput. 25 (8) (2012) 1121–1136.
[36] I. Chakroun, N. Melab, M. Mezmaz, D. Tuyttens, Combining multi-core and GPU computing for solving combinatorial optimization problems, J. Parallel Distrib. Comput. 73 (12) (2013) 1563–1577.
[37] N. Melab, I. Chakroun, M. Mezmaz, D. Tuyttens, A GPU-accelerated Branch-and-Bound algorithm for the flow-shop scheduling problem, in: 2012 IEEE International Conference on Cluster Computing (CLUSTER), 2012, pp. 10–17.

[38] T. Carneiro, A.E. Muritiba, M. Negreiros, G.A.L. de Campos, A new parallel schema for Branch-and-Bound algorithms using GPGPU, in: 2011 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE, 2011, pp. 41–47.
[39] B. Lageweg, J. Lenstra, A.R. Kan, A general bounding scheme for the permutation flow-shop problem, Oper. Res. 26 (1) (1978) 53–67.
[40] E. Taillard, Benchmark for basic scheduling problems, J. Oper. Res. 64 (1993) 278–285.
[41] I.H. Osman, G. Laporte, Metaheuristics: a bibliography, Ann. Oper. Res. 63 (5) (1996) 511–623.
[42] F. Pinel, B. Dorronsoro, P. Bouvry, Solving very large instances of the scheduling of independent tasks problem on the GPU, J. Parallel Distrib. Comput. 73 (1) (2013) 101–110, ISSN 0743-7315.
[43] F. Pinel, B. Dorronsoro, P. Bouvry, A new cellular genetic algorithm to solve the scheduling problem designed for the GPU, in: Metaheuristics Conference (META), 2010.
[44] M. Pedemonte, E. Alba, F. Luna, Towards the design of systolic genetic search, in: 26th IEEE International Parallel and Distributed Processing Symposium, Workshops and PhD Forum (IPDPSW 2012), Workshop PCO'12, 2012, pp. 1778–1786.
[45] T. Zajíček, P. Šucha, Accelerating a flow Shop scheduling algorithm on the GPU, in: Workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP), 2011.
[46] S. Chen, S. Davis, H. Jiang, A. Novobilski, CUDA-based genetic algorithm on traveling salesman problem, in: R. Lee (Ed.), Computers and Information Science, Springer, Berlin, Heidelberg, 2011, pp. 241–252.
[47] J. Li, L. Zhang, L. Liu, A parallel immune algorithm based on fine-grained model with GPU-acceleration, in: 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC), 2009, pp. 683–686.
[48] S.N. Sivanandam, S.N. Deepa, Introduction to Genetic Algorithms, Springer, New York, 2007.
[49] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, J. ACM 24 (2) (1977) 280–289, ISSN 0004-5411.
[50] E. Alba, B. Dorronsoro, Cellular Genetic Algorithms, vol. 42, Springer, New York, 2008.
[51] H. Kung, Why systolic architectures?, IEEE Comput. 15 (1) (1982) 37–46.
[52] H. Kung, C.E. Leiserson, Systolic Arrays (for VLSI), in: Sparse Matrix Proceedings, 1978, pp. 256–282.
[53] M. Dorigo, M. Birattari, T. Stützle, Ant colony optimization, IEEE Comput. Intell. Mag. 1 (4) (2006) 28–39.
[54] N.A. Kallioras, K. Kepaptsoglou, N.D. Lagaros, Transit stop inspection and maintenance scheduling: a GPU accelerated metaheuristics approach, Transp. Res. C Emerg. Technol. 55 (2015) 246–260.
[55] A. Uchida, Y. Ito, K. Nakano, Accelerating ant colony optimisation for the travelling salesman problem on the GPU, Int. J. Parallel Emergent Distrib. Syst. 29 (4) (2014) 401–420.
[56] A. Delévacq, P. Delisle, M. Gravel, M. Krajecki, Parallel ant colony optimization on graphics processing units, J. Parallel Distrib. Comput. 73 (1) (2013) 52–61.

[57] J.M. Cecilia, J.M. Garcia, M. Ujaldon, A. Nisbet, M. Amos, Parallelization strategies for ant colony optimisation on GPUs, in: 25th IEEE International Parallel and Distributed Processing Symposium, Workshops and PhD Forum (IPDPSW 2011), ISSN 1530-2075, 2011, pp. 339–346.
[58] J. Fu, L. Lei, G. Zhou, A parallel Ant Colony Optimization algorithm with GPU-acceleration based on All-In-Roulette selection, in: Third International Workshop on Advanced Computational Intelligence (IWACI 2010), 2010, pp. 260–264.
[59] H. Bai, D. Ouyang, X. Li, L. He, H. Yu, MAX-MIN Ant System on GPU with CUDA, in: Fourth International Conference on Innovative Computing, Information and Control (ICICIC 2009), 2009, pp. 801–804.
[60] W. Jiening, D. Jiankang, Z. Chunfeng, Implementation of Ant Colony algorithm based on GPU, in: Sixth International Conference on Computer Graphics, Imaging and Visualization, 2009 (CGIV '09), 2009, pp. 50–53.
[61] A. Catala, J. Jaen, J.A. Modioli, Strategies for accelerating ant colony optimization algorithms on graphical processing units, in: 2007 IEEE Congress on Evolutionary Computation (CEC 2007), 2007, pp. 492–500.
[62] G. Laporte, S. Martello, The selective travelling salesman problem, Discret. Appl. Math. 26 (2) (1990) 193–207.
[63] T. Scavo, Scatter-to-gather transformation for scalability, 2010, https://hub.vscse.org/resources/223.
[64] M. Harris, S. Sengupta, J.D. Owens, Parallel prefix sum (scan) with CUDA, GPU Gems 3 (39) (2007) 851–876.
[65] G. Reinelt, TSPLIB—a traveling salesman problem library, ORSA J. Comput. 3 (4) (1991) 376–384.
[66] Y.S. You, Parallel ant system for traveling salesman problem on GPUs, in: Eleventh Annual Conference on Genetic and Evolutionary Computation, 2009, pp. 1–2.
[67] J. Li, X. Hu, Z. Pang, K. Qian, A parallel ant colony optimization algorithm based on fine-grained model with GPU-acceleration, Int. J. Innov. Comput. Inf. Control 5 (11) (2009) 3707–3716.
[68] T. Stützle, H.H. Hoos, MAX-MIN ant system, Futur. Gener. Comput. Syst. 16 (8) (2000) 889–914.
[69] Q. Yu, C. Chen, Z. Pan, Parallel genetic algorithms on programmable graphics hardware, in: Advances in Natural Computation, Springer, 2005 pp. 1051–1059.
[70] F. Glover, Tabu Search—part I, ORSA J. Comput. 1 (3) (1989) 190–206.
[71] F. Glover, Tabu Search—part II, ORSA J. Comput. 2 (1) (1990) 4–32.
[72] L. Bukata, P. Šůcha, Z. Hanzálek, Solving the resource constrained project scheduling problem using the parallel Tabu Search designed for the {CUDA} platform, J. Parallel Distrib. Comput. 77 (2015) 58–68.
[73] L. Bukata, P. Šucha, A GPU algorithm design for resource constrained scheduling problem, in: 21st Conference on Parallel, Distributed and Networked-Based Processing (PDP), 2013, pp. 367–374.
[74] M. Czapiński, S. Barnes, Tabu Search with two approaches to parallel flow shop evaluation on CUDA platform, J. Parallel Distrib. Comput. 71 (2011) 802–811.
[75] A. Janiak, W. Janiak, M. Lichtenstein, Tabu Search on GPU, J. Univ. Comput. Sci. 14 (14) (2008) 2416–2427.
[76] W. Bożejko, Solving the flow shop problem by parallel programming, J. Parallel Distrib. Comput. 69 (5) (2009) 470–481.

[77] B. Zdeněk, D. Jan, Š. Přemysl, H. Zdeněk, An acceleration of the algorithm for the nurse rerostering problem on a graphics processing unit, Lect. Notes Manag. Sci. 5 (2013) 101–110.
[78] T.V. Luong, N. Melab, E.G. Talbi, GPU-based approaches for multiobjective local search algorithms. A case study: the flowshop scheduling problem, Evol. Comput. Comb. Optim. 6622 (2011) 155–166.
[79] T.V. Luong, Métaheuristiques parallèles sur GPU, Ph.D. Thesis, Université Lille 1, 2011.
[80] R. Roverso, A. Naiem, M. El-Beltagy, S. El-Ansary, S. Haridi, A GPU-enabled solver for time-constrained linear sum assignment problems, in: 7th International Conference on Informatics and Systems (INFOS), 2011, pp. 1–6.
[81] A. Naiem, M. El-Beltagy, Deep greedy switching: a fast and simple approach for linear assignment problems, in: 7th International Conference of Numerical Analysis and Applied Mathematics, 2009.
[82] L. Paquete, T. Stützle, Stochastic local search algorithms for multiobjective combinatorial optimization: a review, Tech. Rep., Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle, 2006.
[83] M. Moz, M.V. Pato, A genetic algorithm approach to a nurse rerostering problem, Comput. Oper. Res. 34 (3) (2007) 667–691.