

GPU computing applied to linear and mixed-integer programming

Vincent Boyer, Didier El Baz, M.A. Salazar-Aguilar

▶ **To cite this version:**

Vincent Boyer, Didier El Baz, M.A. Salazar-Aguilar. GPU computing applied to linear and mixed-integer programming. Advances in GPU, Research and Practice, Elsevier, pp.247-271, 2017, 978-0-12-803738-6. 10.1016/B978-0-12-803738-6.00010-0 . hal-02091756

HAL Id: hal-02091756
https://laas.hal.science/hal-02091756

Submitted on 6 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

GPU Computing Applied to Linear and Mixed Integer Programming

Vincent Boyer · Didier El Baz · M. Angélica Salazar-Aguilar

the date of receipt and acceptance should be inserted later

Abstract Thanks to CUDA and OpenCL, Graphics Processing Units (GPUs) have recently gained considerable attention in science and engineering as accelerators for High Performance Computing (HPC). In this chapter, we show how the Operations Research (OR) community can take great benefit of GPUs. In particular, we present a survey of the main contributions to the field of GPU computing applied to linear and mixed-integer programming. The OR field is rich in complex problems and sophisticated algorithms that can take advantage of parallelization. However, all algorithms in the literature do not fit to the SIMT paradigm. Therefore, we highlight the main issues tackled by different authors to overcome the difficulties of implementation and the results obtained with their optimization algorithms via GPU computing.

Keywords GPU Computing · Operations Research · Linear Programming · Mixed-Integer Programming · Metaheuristics · Exact Solution Methods · Parallel Computing

1 Introduction

GPUs are many cores parallel architectures that have originally been designed for visualization purpose. They have also evolved during the last decade towards powerful computing accelerators for High Performance Computing (HPC).

Vincent Boyer and M. Angelica Salazar-Aguilar
Graduate Program in Systems Engineering
Universidad Autónoma de Nuevo León, Mexico
Facultad de Ingeniería Mecánica y Eléctrica
E-mail: {vincent.boyer, maria.salazaragl}@uanl.edu.mx

Didier El Baz
CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse Cedex 4, France,
Université de Toulouse, LAAS, F-31400 Toulouse France
E-mail: elbaz@laas.fr

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	2,688	0.732	6

Table 1 Overview of NVIDIA GPUs quoted in the chapter (see `http://www.nvidia.com` for more details)

The exploitation 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 CUDA, like 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 out of any GPU memory to any other GPU memory in any other cluster without involving 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, one terabyte per second memory bandwith and twice as much flops as Maxwell. NVLink technology will also permit data to move five to ten times faster between GPUs and CPUs than with current PCI-Express, making GPU computing accelerators very efficient devices for HPC. Going back at the GPU computing accelerators previously released (some of which are presented in Table 1, that summarizes also the characteristics of GPUs considered in this paper) we can measure the progress accomplished during a decade.

GPUs have been widely applied to signal processing and linear algebra. The interest in GPU computing is now wide-spread. Almost all domains in science and engineering are concerned. We can quote for example astrophysics, seismic, oil industry, and nuclear industry, e.g., see Nguyen (2008). Most of the time, GPUs 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 is prolific in 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. (2013) and Schulz et al. (2013) deals with routing problems. Luong (2011b) considers Metaheuristics on GPU. More generally, Alba et al. (2013) study parallel metaheuristics.

In this chapter, we present an overview on research contributions of GPU computing applied to OR; each section contains a short introduction and useful references of the algorithm implementations. It is dedicated to 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 to identify domains of research in this field that have not been addressed so far.

The organization of this chapter is the following: Section 2 introduces the field of Operations Research. The main exact optimization algorithms implemented via GPU computing in the domain of OR are described in Section 3. Section 4 is dedicated to present 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

Operations research 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. Many new analytical methods have evolved, such as: mathematical programming, simulation, game theory, queuing theory, network analysis, decision analysis, multicriteria analysis, etc., which have 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, modelling, problem structuring, simulation and other analytical techniques to examine assumptions, facilitate an in-depth understanding and decide on practical action.

Nowadays, many decision problems are formulated as mathematical programs, which require the maximization or minimization of an objective function subject to a set of constraints. A general representation of an optimization problem is the following:

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

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

where $x = (x_1, x_2, ..., x_n)$, $n \in \mathbb{N}$, is the vector of decision variables, (1) is the objective function, and (2) imposes that x belongs to a defined domain $\mathcal{D}$. A solution x^* is said 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 $f(x) = p^T.x$, and it exists 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 have the following general form:

$$\max p^T x \tag{3}$$
$$\text{s.t. } Ax = b \tag{4}$$

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 (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), etc. There is not a single method or algorithm that works best on all classes of problems.

Linear programming problem are generally solved with the simplex algorithm and its variants (see Schrijver (1986)). A basis solution is defined such that $x = (x_B, x_H)$ and $A.x = A_B x_B + A_H x_H$, where $A_B = \mathbb{R}^n \times \mathbb{R}^n$ and $det(A_B) \neq 0$. In this case, $x_B = A_B^{-1} b$ and $x_H = 0$. The principle of the simplex algorithm is to build at each iteration new basis solution that improve the current objective value $p^T x$ by swapping one variable in x_B with one in x_H, until none improvement is possible.

Mixed-integer programming and combinatorial problems are generally much harder to solve since, in the worst case, all possible solutions for x should be explored in order to prove optimality. The branch-and-bound 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 in 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 have been designed to tackle complex optimization problems where other optimization methods have failed to provide a good feasible solution in a convenient processing time. These methods have come to be 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 Ólafsson (2006)). Simulated Annealing (SA), Tabu Search (TS), Scatter Search (SS), Genetic Algorithms (GA), Variable Neighborhood Search (VNS), Greedy Randomized Adaptive Search Procedure (GRASP), Adaptive Large Neighborhood Search (ALNS), and Ant Colony (ACO) are some of the most widely used metaheuristics.

The purpose of a metaheuristic is to find an optimal or near optimal solution without guarantee of optimality in order to save processing time. These algorithms generally start from a feasible solution obtained through any constructive method and try to improve it by exploring one or more defined neighborhoods. A neighborhood is composed by 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 like in GA or ACO, and information are exchanged between these solutions in order to improve the convergency of the approach.

The solution of real-world decision problems represented as mathematical programs (optimization problems) is often hindered by size. In mathematical programing, 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 reduction in computer memory requirements may be achieved. Hence, practitioners and researchers have concentrated their efforts to develop optimization algorithms that exploit the capabilities of the GPU computing.

In the literature related to GPU computing applied to OR, there are mainly two classes of optimization problems that have been studied: linear programming problems and mixed-integer programming problems. For solving linear optimization problems the Simplex method is by far one of the most widely used exact method and for mixed-integer optimization problems the Branch-and-Bound 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 Branch-and-Bound algorithms (see Winston and Goldberg (2004)). Due that 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 speeding part of the algorithm on GPU.

3.1 The Simplex Method

Originally designed by Dantzig (1951), the simplex algorithm and its variants (see Schrijver (1986)) are largely used to solve linear programming (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 MIPs (Mixed-Integer Problems)

in many heuristics and exact approaches like the Branch-and-Bound. Furthermore, it is known that in algorithms like the Branch-and-Bound, the major part of the processing time is spent in solving these linear relaxations. Hence, faster simplex algorithms benefit to all fields of Operations Research. Table 2 summarizes the contributions related to GPU implementations of simplex algorithms that can be found in the literature.

Algorithm	Reference
The Simplex Tableau	Lalami et al. (2011a,b)
The Two-Phase Simplex	Meyer et al. (2011)
The Revised Simplex	Ploskas and Samaras (2015) Nikolaos and Nikolaos (2013) Bieling et al. (2010) Spampinato and Elster (2009) Greeff (2005)
The Interior Point Method	Jung and O'Leary (2008)
The Exterior Point Method	Ploskas and Samaras (2015)

Table 2 Linear Programming and GPUs

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

Simplex algorithm, like the revised simplex algorithm, involves many operations on matrices and many authors tried to take advantage to recent advance in linear programming. 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 (2009), with cuBLAS (`https://developer.nvidia.com/cublas`), achieve a speedup of 2.5 for problems with 2,000 variables and 2,000 constraints when comparing their GPU implementation on a NVIDIA GeForce GTX 280 GPU to the ATLAS-based solver Whaley and Dongarra (1999) on an Intel Core 2 Quad 2.83GHz processor. Nikolaos and Nikolaos (2013) propose an implementation based on MATLAB and CUDA environments and report a speedup of 5.5 with an Intel Core i7 3.4GHz and a NVIDIA Quadro 6000 with instances up to 5,000 variables and 5,000 constraints.

In order to improve the efficiency of their approach, Ploskas and Samaras Nikolaos and Nikolaos (2013) made a complete study on the basis update for the revised simplex method. They propose a GPU implementation of the Product Form of the Inverse (PFI) from Dantzig and Orchard-Hays (1954) and of Modification of the PFI (MPFI) from Benhamadou (2002). Both approaches

tend to reduce the computation effort of the matrices operations. Their results show that PFI is slightly better than MPFI.

Ploskas and Samaras (2015) present 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 use the MATLAB environment for their implementation and compare their results to MATLAB large-scale linprog built-in function. All the main phases of both algorithms are performed on GPU. 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 with a maximum speedup of 181 on dense LPs and 20 on the sparse ones.

Bieling et al. (2010) propose some algorithm optimizations for the revised simplex used by Ploskas and Samaras (2015). They use the steepest-edge heuristic from Goldfarb and Reid (1977) to select the entering variables and an arbitrary bound process in order to select the leaving variables. The authors compare their results to the GLPK solver (`http://www.gnu.org/software/glpk/`) and report a reduction in computation time by a factor of 18 for instances with 8,000 variables and 2,700 constraints on a system with Intel Core 2 Duo E8400 3.0 GHz processor and NVIDIA GeForce 9600 GT GPU.

Like Bieling et al. (2010) show, 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 to the GPU architecture. Lalami et al. (2011a,b) and Meyer et al. (2011) propose two implementations of this algorithm, on one GPU and on multi-GPUs, and they reported that both algorithms reach a significant speedup.

Lalami et al. (2011b) use the algorithm of Garfinkel and Nemhauser (1972) which improves the algorithm of Dantzig by reducing the number of operations and the memory occupancy. They extend this implementation to the multi-GPU context in Lalami et al. (2011b). They adopt a horizontal decomposition where the constraints, i.e. 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 use a server with an Intel Xeon E5640 2.66GHz CPU and two NVIDIA C2050 GPUs, and instances with up to 27,000 variables and 27,000 constraints. They observe a maximum speedup of 12.5 with a single GPU and 24.5 with two GPUs.

Meyer et al. (2011) propose a multi-GPU implementation of the two-phase simplex. The authors consider a vertical decomposition of the simplex tableau, i.e. the variables are distributed amongst the GPUs, in order to have less communications between GPUs. Like in Lalami et al. (2011a,b), they consider the implementation of the pivoting phase and the selection of the entering and leaving variables. They use a system with two Intel Xeon X5570 2.93GHz processors and four NVIDIA Tesla S1070. They solve instances with up to 25,000 variables and 5,000 constraints and show that their approach outper-

forms the open-source solver CLP (`https://projects.coin-or.org/Clp`) of the COIN-OR project.

Jung and O'Leary (2008) study the implementation of the Interior Point Method. They propose 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 like matrix assembly, Cholesky factorization and forward and back substitution. However, computational tests show that the proposed approach does not clearly outperforms the sequential version on CPU due to 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 that it is one of the most effective in sequential implementations.

3.2 Dynamic Programming

The Dynamic Programming algorithm has been introduced by Bellman (1957). The main idea of this algorithm consists in solving complex problems by decomposing them in smaller problems that are iteratively solve. This algorithm has a natural parallel structure. An overview of the literature dealing with the implementation of Dynamic Programming on GPU can be found in Table 3. As we can see, only Knapsack Problems (KP) have been studied so far, more details on these contributions are given in the sequel.

Algorithm	Problem	Reference
Dense Dynamic Programming	Knapsack Problem	Boyer et al. (2011, 2012)
Dense Dynamic Programming	Multi-Choice Knapsack Problem	Suri et al. (2012)

Table 3 Dynamic Programming on GPU

3.2.1 Knapsack Problems

The KP (see Martello and Toth (1990)) is one of the most studied problems in OR. It consists in selecting a set of items which are associated with a profit and a weigh. 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. (2011) propose a hybrid dense dynamic programming algorithm implementation on GPU. Data are organized in a table where the columns represent the items and the raws, the increasing capacity of the knapsack. At each iteration, a raw is filled based on information provided by the previous one. The authors also propose a data compression technique in order to deal with the high memory requirement 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 are 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 are considered. A reduction in computation time by a factor of 26 is reported and the reduction in memory occupancy appears more efficient when the size of the problem increases while the overhead does not exceed 3% of the overall processing time.

Boyer et al. (2012) gave an extension of their approach whereby a multi-GPU hybrid implementation of the dense dynamic programming method is 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, i.e., 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 is 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, show 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 are considered.

3.2.2 Multiple-Choice Knapsack Problem

Suri et al. (2012) studied a variant of the knapsack problem which is called the multiple-choice knapsack problem (see Martello and Toth (1990)). In this case, the items are grouped in subsets and exactly one item of each subset should be selected without exceeding the capacity of the knapsack. Their dynamic programming algorithm is similar to the one of Boyer et al. (2011, 2012), however, in order to ensure high processor utilization multiple cells of the table are computing by one GPU thread.

They report 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 show that their implementation outperforms the one of Boyer et al. (2011, 2012) on randomly generated instances of the multi-choice knapsack problem. For their experimental tests, they use two Intel Xeon E5520 CPUs and a NVIDIA Tesla M2050. However, no information is given on the memory occupancy of their algorithm.

3.3 Branch and Bound

The Branch and Bound (B&B) algorithm has been designed to explore in a smart way the solution space of a MIP. From the original problem, the B&B generates new nodes which corresponds to subproblems obtained by fixing variables or adding constraints. Each node generates in a similar way other nodes and so on until the optimality condition is 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 GPU is in many cases a challenge.

To the best of our knowledge, as we can see in Table 4, there are three types of problems that have been solved by a B&B GPU implementation: Knapsack Problems (KP) , Flow-shop Scheduling Problems (FSP) (see Pinedo (2012)), and a Traveling Salesman Problem (TSP) (see Reinelt (1994)). 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.

Algorithm	Problem	Reference
Branch-And-Bound	Knapsack Problem	Boukedjar et al. (2012)
		Lalami and El Baz (2012)
		Lalami (2012)
Branch-And-Bound	Flow-shop Scheduling Problem	Chakroun and Melab (2012)
		Chakroun et al. (2012, 2013)
		Melab et al. (2012)
Branch-And-Bound	Traveling Salesman Problem	Carneiro et al. (2011)

Table 4 Branch and Bound on GPU

3.3.1 Knapsack Problem

Boukedjar et al. (2012), Lalami and El Baz (2012), and Lalami (2012) studied the GPU implementation of the B&B algorithm for KPs. The nodes are first generated in sequential on the host. When their number reach a threshold, the GPU is then used to explore the nodes in parallel, i.e. one node per GPU thread. Almost all the phases of the algorithm are implemented on the device, i.e. bounds computation, generation of the new nodes, and updates of the best lower bound found via atomic operations. Parallel bounds comparison and identification of non-promising 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 2.66GHz processor and a NVIDIA C2050 GPU are used for the computational tests. The authors report a speedup of 52 in Lalami (2012) 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 is studied by Melab et al. (2012), Chakroun and Melab (2012), and Chakroun et al. (2012). In this problem, a set of jobs have to be scheduled on a set of available machines. In Melab et al. (2012) and Chakroun et al. (2012), the authors identify that 99% of the time spent by the B&B algorithm is in the bounding process. Hence, they focus their effort on parallelizing this operator on a GPU, and eliminating, selecting, and branching are carried out by the CPU. Indeed, at each step in the tree exploration of the B&B, a pool of subproblems is selected and is sent to the GPU which performs, in parallel, the evaluation of their lower bound through the algorithm proposed by Lageweg et al. (1978).

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

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

Table 5 Branch Refactoring From Chakroun et al. (2012)

Computational experiments are carried out on a system with an Intel Xeon E5520 2.27GHz and a NVIDIA C2050 computing system. Some instances of FSP proposed by Taillard (1993) and a maximum speedup factor of 77 is observed for instances with 200 jobs on 20 machines as compared with a sequential version. This approach is extended in Chakroun and Melab (2012) to the multi-GPU case where a maximum speedup of 105 is reported with two Tesla T10.

Finally, Chakroun et al. (2013) consider 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 add the branching and the pruning operator on GPU to the bounding operator. Two approaches are then considered. Firstly, a concurrent exploration of the B&B tree, where a pool of subproblems is partitioned between the CPU cores. Secondly a cooperative exploration of the B&B tree, where CPU threads handle a part of the pool of the subproblems to explore on GPU, which allows to interleave and overlap data transfer through

asynchronous operations. The pool of subproblems to explore is determined dynamically with a heuristic according to the instance being solve and the GPU configuration.

With an Intel Xeon E5520 2.27GHz and a NVIDIA C2050, on the instances of Taillard (1993), they achieve an acceleration of 160 with the cooperative approach. Indeed, the cooperative approach appears to be 36% faster than the concurrent one. From these results, the authors recommend to use 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. (2011) consider the solution of the TSP on a GPU. The TSP consists in finding the shortest route that will follow a salesman 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 processes the exploration of the resulting subtree through a depth first strategy with backtracking. This strategy permits 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 have been 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. (2011) report a maximum speedup of 11 on a system with an Intel Core i5 750 2.66GHz and an NVIDIA GeForce GTS 450. The authors use randomly generated instances of asymmetric traveling salesman problem with up to 16 cities.

4 Metaheuristics

A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently 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 Osman and Laporte (1996)).

GPU implementations of metaheuristics has received a particular attention by practicioners 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 which 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 (Genetic Algorithms, Ant Colony, Tabu Search, amongst others) that have been implemented under a GPU architecture. .

Algorithm	Problem	Reference
Cellular Genetic Algorithm	Independent Tasks Scheduling Problem	Pinel et al. (2013, 2010)
Systolic Genetic Search	Knapsack Problem	Pedemonte et al. (2012)
Island-Based Genetic Algorithm	Flow-Shop Scheduling Problem	Zajíček and Šucha (2011)
Genetic Algorithm	Traveling Salesman Problem	Chen et al. (2011)
Immune Algorithm	Traveling Salesman Problem	Li et al. (2009b)

Table 6 Genetic Algorithms on GPU

4.1 Genetic Algorithms

As shown in Table 6, Genetic Algorithms (GAs) and their variants on GPU 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 in the search space.

4.1.1 The Traveling Salesman Problem

Li et al. (2009b) and Chen et al. (2011) propose a Fine Grained Parallel GA on GPU in order to solve the well-known TSP. In the approach of Li et al. (2009b), 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 a partially mapped crossover method (see Sivanandam and Deepa (2007)) and the selection of the parents is done via an *adjacency-partnership method*, however no details are provided on this process. The tournament selection is preferred to the classic roulette wheel selection. On a GeForce

9600GT, they report an acceleration between 2.42 and 11.5 on instances of the literature with up to 226 cities.

Chen et al. (2011) use an order crossover operator where parents exchange their sequence orders of a portion of their chromosome, which prevents a city to appear more than once in a solution. Furthermore, they implement the *2-opt mutation operator* which seems to be particularly adapted to the TSP. They also use a simple selection process, that conserves the best chromosome. Due to the need of synchronization at each step of their GA, they carry 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 do not exploit all the capability of computation of their GPU, they show that their parallel GA on GPU outperforms the sequential one on an Intel Xeon E5504.

4.1.2 Scheduling Problems

Pinel et al. (2013) propose a fine grained parallel GA for a scheduling problem, i.e., 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 (1977) and this solution is added to the initial population of a Cellular Genetic Algorithm (see Alba and Dorronsoro (2008)). The two main steps of this algorithm are conducted in parallel on the GPU, i.e. the search for the best machine assignment for each task and the update of the solution.

In the Cellular Genetic Algorithm, the population is arranged in a two-dimensional 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. (2013) with a Tesla C2050 on randomly generated instances with up to 65536 tasks and 2048 machines show that:

- the Min-Min heuristic on GPU outperforms the parallel implementation on CPUs (two Intel Xeon E5440 processors with 2×4 cores);
- the Cellular Genetic Algorithm is able to improve up to 3% in the first generation the initial solution provided by the Min-Min heuristic.

In Zajíček and Šucha (2011), a parallel island-based Genetic Algorithm is implemented on GPU for the solution of the FSP. In this variant of the GA, the population is divided in multiple sub-populations isolated on an island in order to preserve genetic diversity. These populations can share genetic information through an operator of migration. Zajíček and Šucha perform evaluations, mutations and crossovers of individuals in sub-populations in parallel and independently from other populations, and they report a speedup of 110 on a Tesla C1060.

4.1.3 Knapsack Problems

Pedemonte et al. (2012) propose a Systolic Genetic Search (SGS) for the solution of the KP using GPU. The population is arranged in a two-dimensional toroidal grid of cells, and at each iteration, solutions are transiting horizontally and vertically in a determined direction within the grid. This communication scheme is based on the model of systolic computation from Kung (1982) and Kung and Leiserson (1978), i.e. 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 associate a block of GPU threads to a cell of the grid.

Experiments are carried out on a system with a GeForce GTX 480 GPU. Problems without correlation and up to 1,000 variables are considered. Experimental results show that the SGS method produces solutions of very good quality and that the reduction in computation time ranges from 5.09 to 35.7 times according to the size of the tested instances.

4.2 Ant Colony Optimization

In this subsection, we focus on ant colony approaches which 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 GPU with ant colony.

Ant Colony Optimization (ACO) (see Dorigo et al. (2006)) 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 is composed by two main operators, i.e. 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 is due to Catala et al. (2007) for the solution of the orienteering problem (OP), also known as the selective traveling salesman problem (see Laporte and Martello (1990)). In this variant of the TSP, the visits are optional and each customer has a positive score which is collected by the salesman if and only if the customer is visited. Hence the OP consists in finding a route that maximizes the total collected score within a time limit.

The authors propose to arrange the path followed by each ant in a two-dimensional table where each row is associated to an ant and a column represents the position of a node in the ant's path. The attractiveness between a pair of nodes (or pheromones) is stored also in a table. In order to build the paths for the ants, they use a *selection by projection* which is based on the

Algorithm	Problem	Reference
Ant Colony Optimization	Transit stop inspection and maintenance scheduling	Kallioras et al. (2015)
Ant Colony Optimization	Traveling Salesman Problem	Uchida et al. (2014)
Max-Min Ant System	Traveling Salesman Problem	Delévacq et al. (2013)
Ant Colony Optimization	Traveling Salesman Problem	Cecilia et al. (2011)
Max-Min Ant System	Traveling Salesman Problem	Fu et al. (2010)
Max-Min Ant System	Traveling Salesman Problem	Bai et al. (2009)
Max-Min Ant System	Traveling Salesman Problem	Jiening et al. (2009)
Ant Colony Optimization	Orienteering Problem	Catala et al. (2007)

Table 7 Ant Colony Algorithms on GPU

principle of an orthographic camera clipping a space to determine, in parallel, the next node to visit. The results obtained show that their 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. (2011) study 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 propose to use a scatter-to-gather transformation (see Scavo (2010)) for the pheromone update, in place of build in atomic operations. Their results obtained with a C1060 GPU with instances with up to 2,396 nodes show a speedup of 25.

Uchida et al. (2014) propose an extensive study on strategies to accelerate the ACO on GPU. In particular, they study 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 to select a node already visited by using the prefix-sum algorithm (see Harris et al. (2007)), eliminating them through a compression method or by stochastic trial.They also study the update of the pheromones through the shared memory in order to avoid un-coalesced memory access. The computational tests carried out with

a GTX580 and a set of benchmark instances from the TSPLIB (see Reinelt (1991)) show that the efficiency of the proposed approches depends on the number of visited cities and hence proposed a hybrid approach. A maximum speedup of 22 is reported.

The solution of the transit stop inspection and maintenance scheduling problem is presented by Kallioras et al. (2015). In this problem, the transit stops should be grouped in districts and the visits, of the transit stops, for each vehicle within a district should be scheduled. They propose a hybrid CPU-GPU implementation where the lengh of the ant's path, the pheromone update, addition, and comparaison operations are performed on the GPU. The implementation is not detailed in the paper and they report a speedup of 21 with a GTX 660M.

Reference is also made to You (2009) and Li et al. (2009a) who also study the GPU implementation of ACO on 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 Stützle and Hoos (2000)) 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. (2009) and Bai et al. (2009) propose the first implementations of the MMAS on GPU for the solution of TSP. In Jiening et al. (2009), only the tour construction stage is processed on GPU, whilst in Bai et al. (2009), the pheromone update on GPU is also studied. In these works, the reported speedup do not exceed 2.

Fu et al. (2010) use the Jacket toolbox which connects MATLAB to GPU for their implementation of MMAS on GPU. Ants share only one pseudo-random number matrix, one pheromone matrix, one tabu matrix and one probability matrix in order to reduce communication between CPU and GPU. Furthermore, they present a variation of the traditional roulette wheel selection, i.e. 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 are carried out on a system with an Intel i7 3.3GHz processor and a NVIDIA Tesla C1060 GPU instances of the literature with up to 1,002 cities. They also show that the solution obtained is close to the one provided by their sequential algorithm.

More recently, Delévacq et al. (2013) present a MMAS for the parallel ant and the multiple colony approaches. In this paper, the authors discuss extensively about the drawbacks encountered in such an implementation and propose some solutions based on previous works. In particular, they use the Linear Congruential Generator as proposed by Yu et al. (2005) and a GPU

3-opt local search to improve solution quality. Furthermore, the authors compare different GPU implementations where ants are associated to one GPU thread or to one GPU block of threads, also when considering multiple colonies distributed among the GPU block. Experiments conducted with two GPUs of a NVIDIA Fermi C2050 server on instances taken from the literature up to 2,396 cities show not only a maximum speedup of 24 but also a conservation of the quality of the reported solution. These results are obtained when multiple colonies are considered combined with a local search strategy.

4.3 Tabu Search

As we can see in Table 8, Tabu Search (TS) approaches have been also extensively used for the solution of scheduling problems. TS, created by Glover (1989, 1990), uses local 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, for instance, the set of solutions recently explored.

Algorithm	Problem	Reference
Tabu Search	Resource Constrained Project Scheduling Problem	Bukata et al. (2015) Bukata and Šucha (2013)
Tabu Search	Permutation Flow Shop Scheduling Problem	Czapiński and Barnes (2011)
Tabu Search	Traveling Salesman Problem / Flow Shop Scheduling Problem	Janiak et al. (2008)

Table 8 Tabu Search on GPU

The first reported GPU implementation of a TS algorithm is due to Janiak et al. (2008) 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 define a neighborhood on swap move, i.e. interchanging the position of two customers or two jobs in the current solution. A table of two dimensions (Neighborhood texture) is created on the GPU which computes in each cell of coordinates (i, j) the new solution, resulting from the 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 use commercial GPUs (GeForce 7300 GT, GeForce 8600 GT and GeForce 8800 GT) for their experimental tests and, with randomly generated instances, achieve a speedup of 4 with the PFSP and almost no speedup with the TSP.

Based on the results of Janiak et al. (2008), who showed that 90% of the processing time is spent in the evaluation function, Czapiński and Barnes (2011) designed an improved parallelization of TS for the PFSP. They use a limited parallelization of evaluations as proposed by Bożejko (2009). They propose 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 to not 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 (2011) reach a maximum speedup of 89 with a Tesla C1060 on instances from the literature and show that their implementation outperforms the one of Janiak et al. (2008).

Bukata et al. (2015) and Bukata and Šucha (2013) subsequently present a parallel TS method for the Resource Constrained Project Scheduling Problem (RCPSP). In this variant of the FSP, each job in order to be executed 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 a preprocessing is done to eliminate unfeasible swaps, i.e. swaps that violate 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 two dimensional 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. (2015) propose to concurrently run a TS in each GPU block which manages its own incumbent solution. A list of incumbent solutions is maintained on the global memory that each block access through atomic operations. Hence, blocks can co-operate through the exchange of solutions and a diversification technique is processed when a solution has not been improved after a certain number of iterations. Experiments are carried out on a server with a GTX 650 with benchmark from the literature with up to 600 projects and 120 activities. The results are compared with a sequential and a parallel CPU version of their algorithm. They show that the GPU version achieved a speedup of almost 2 compared to the parallel one without degrading the solution quality.

4.4 Other Metaheuristics

Aside from the metaheuristics presented in the last sections which have been widely studied by different authors, other metaheuristics have been given less attention in the literature on GPU computing. In this subsection we present these approaches. Table 9 gives an overview of the literature which is detailed in this subsection.

Algorithm	Problem	Reference
Constructive Heuristic	Nurse Rerostering Problem	Zdeněk et al. (2013)
Multiobjective Local Search	Multi-Objective Flow Shop Scheduling Problem	Luong et al. (2011) Luong (2011a)
Deep Greedy Switching	Linear Sum Assignment Problem	Roverso et al. (2011)

Table 9 Other Metaheuristics on GPU

4.4.1 Deep Greedy Switching

The Deep Greedy Switching (DGS) heuristic (see Naiem and El-Beltagy (2009)) starts from a random initial solution and moves to better solutions by considering a neighborhood resulting from a restricted *2-exchange* approach. This algorithm has been implemented on GPU by Roverso et al. (2011) for the solution of the Linear Sum Assignment Problem. This problem consists in assigning to a set of agents a set of jobs. An agent can only perform 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 focus 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 is processed in parallel on the GPU. Computational experiments are carried out on a system with a NVIDIA GTX 295 GPU wit randomly generated instances with up to 9744 jobs. The authors report a reduction of computation time by a factor of 27.

4.4.2 Multiobjective Local Search

Luong et al. (2011) and Luong (2011a) study the implementation on GPU 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 Tabu Search, where the objectives

are aggregated in order to obtain a mono-objective problem; and a Pareto Local Search Algorithms from Paquete and Stützle (2006). Furthermore, in order to overcome the non-coalesced accesses to the memory, they propose to use the texture memory of the GPU.

They carry out their experimental tests on problems range from 20 jobs and 10 machines to 200 jobs and 20 machines. They consider three objectives: the makespan, total tardiness, and number of jobs delayed with regards to their due date. With a GTX 480, they observe a maximum speedup of 16 times with the aggregated Tabu Search and 15.7 times with the Pareto local search algorithm.

4.4.3 Constructive Heuristic

Zdeněk et al. (2013) present an implementation of the Constructive Heuristic of Moz and Pato (2007) who proposed this heuristic to initialize their genetic algorithm for the solution of the Nurse Rerostering Problem. This Constructive Heuristic proceeds 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 some 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 reassignment and to take advantage of the GPU.

Experimental tests are carried out with a GTX 650 on the instances from Moz and Pato (2007), considering up to 32 nurses with a planning horizon of 28 days. They achieve a maximum speedup of 2.51 with an optimality gap between 4% and 18%.

5 Conclusions

The domain of OR is rich in difficult problems. In this chapter, we have concentrated on GPU computing in the field of OR. In particular, we have surveyed the major contributions to Integer Programming and Linear Programming. In many cases, significant reduction in computing time have been observed for OR problems. Nevertheless, it is difficult to establish a quantitative comparison between the different approaches quoted in this paper since the reported results have been 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 still to be designed and commonly accepted. Issues related to the quality of solutions have also to be taken into account accordingly.

As shown in this chapter, most of the classic OR algorithms have been implemented on GPU. 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 in the GPU architecture. We note for instance that important acceleration in GA and ACO were achieved by modifying the roulette wheel selection. Besides, the solution coding and the way it is stored in the GPU memory play a major role in the performance of the algorithms. The main drawbacks of implementation come from the fact that we try to fit in a parallel architecture algorithms that are sequential by nature. Dedicated parallel OR algorithms need to be designed.

For many applications including OR problems, the future of GPU computing seems very promising. New feature like dynamic parallelism, i.e., the possibility for GPU threads to automatically spawn new threads simplifies parallel programming and seems particularly suited to Integer Programming applications. The new NVIDIA Pascal architecture should feature more memory, one terabyte per second memory bandwidth and twice as much FLOPS as current Maxwell architecture. NVLink technology should also permit data to move five to ten 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 pragmas that enables C/C++ and Fortran programmers to exploit highly parallel processors with much of the convenience of OpenMP.

In the future, OR industrial codes will be able to take benefit of accelerators like GPUs that are widely available and to propose attractive and fast solutions to customers. Nevertheles, 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 thanks 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

Alba, E., Dorronsoro, B., 2008. Cellular genetic algorithms. Vol. 42. Springer.

Alba, E., Luque, G., Nesmachnow, S., 2013. Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research 20 (1), 1–48.

Bai, H., Ouyang, D., Li, X., He, L., Yu, H., dec. 2009. MAX-MIN Ant System on GPU with CUDA. In: Fourth International Conference on Innovative Computing, Information and Control (ICICIC 2009). pp. 801–804.

Bellman, R., 1957. Dynamic Programming. Princeton University Press.

Benhamadou, M., 2002. On the simplex algorithm 'revised form'. Advances in Engineering Software 33 (11), 769–777.

Bieling, J., Peschlow, P., Martini, P., april 2010. An efficient GPU implementation of the revised simplex method. In: 24th IEEE International Parallel Distributed Processing Symposium, Workshops and Phd Forum (IPDPSW 2010). pp. 1–8.

Boukedjar, A., Lalami, M., El Baz, D., feb. 2012. Parallel branch and bound on a CPU-GPU system. In: 20th International Conference on Parallel, Distributed and network-based Processing (PDP 2012). pp. 392–398.

Boyer, V., El Baz, D., Elkihel, M., feb. 2011. Dense dynamic programming on multi GPU. In: 19th International Conference on Parallel, Distributed and network-based Processing (PDP 2011). pp. 545–551.

Boyer, V., El Baz, D., Elkihel, M., 2012. Solving knapsack problems on GPU. Computers and Operations Research 39 (1), 42–47.

Bożejko, W., 2009. Solving the flow shop problem by parallel programming. Journal of parallel and distributed computing 69 (5), 470–481.

Brodtkorb, A. R., Hagen, T. R., Schulz, C., Hasle, G., 2013. GPU computing in discrete optimization. Part I: Introduction to the GPU. EURO Journal on Transportation and Logistics, 1–29.

Bukata, L., Šůcha, P., Hanzálek, Z., 2015. Solving the resource constrained project scheduling problem using the parallel tabu search designed for the {CUDA} platform. Journal of Parallel and Distributed Computing 77 (0), 58 – 68.

Bukata, L., Šucha, P., 2013. A GPU algorithm design for resource constrained scheduling problem. In: 21st Conference on Parallel, Distributed and networked-based Processing (PDP). pp. 367–374.

Carneiro, T., Muritiba, A. E., Negreiros, M., Lima de Campos, G. A., 2011. A new parallel schema for branch-and-bound algorithms using gpgpu. In: Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on. IEEE, pp. 41–47.

Catala, A., Jaen, J., Modioli, J., sept. 2007. Strategies for accelerating ant colony optimization algorithms on graphical processing units. In: 2007 IEEE Congress on Evolutionary Computation (CEC 2007). pp. 492–500.

Cecilia, J., Garcia, J., Ujaldon, M., Nisbet, A., Amos, M., may 2011. Parallelization strategies for ant colony optimisation on GPUs. In: 25th IEEE International Parallel and Distributed Processing Symposium, Workshops and Phd Forum (IPDPSW 2011). pp. 339–346.

Chakroun, I., Melab, N., 2012. 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). pp. 389 – 395.

Chakroun, I., Melab, N., Mezmaz, M., Tuyttens, D., 2013. Combining multi-core and gpu computing for solving combinatorial optimization problems. Journal of Parallel and Distributed Computing 73 (12), 1563–1577.

Chakroun, I., Mezmaz, M., Melab, N., Bendjoudi, A., 2012. Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. Concurrency and Computation: Practice and Experience 25 (8), 1121–1136.

Chen, S., Davis, S., Jiang, H., A., N., 2011. CUDA-based genetic algorithm on traveling salesman problem. In: Lee, R. (Ed.), Computers and Information Science. Springer Berlin Heidelberg, pp. 241–252.

Czapiński, M., Barnes, S., 2011. Tabu Search with two approaches to parallel flow shop evaluation on CUDA platform. Journal of Parallel and Distributed Computing 71, 802 – 811.

Dantzig, G., 1951. Maximization of a linear function of variables subject to linear inequalities. In: Activity Analysis of Production and Allocation. Wiley and Chapman-Hall, pp. 339–347.

Dantzig, G. B., Orchard-Hays, W., 1954. The product form for the inverse in the simplex method. Mathematical Tables and Other Aids to Computation, 64–67.

Delévacq, A., Delisle, P., Gravel, M., Krajecki, M., 2013. Parallel ant colony optimization on graphics processing units. Journal of Parallel and Distributed Computing 73 (1), 52–61.

Dorigo, M., Birattari, M., Stützle, T., 2006. Ant colony optimization. Computational Intelligence Magazine, IEEE 1 (4), 28–39.

Fu, J., Lei, L., Zhou, G., aug. 2010. 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). pp. 260–264.

Garfinkel, R. S., Nemhauser, G. L., 1972. Integer programming. Vol. 4. Wiley New York.

Glover, F., 1989. Tabu search - part i. ORSA Journal on computing 1 (3), 190–206.

Glover, F., 1990. Tabu search - part ii. ORSA Journal on computing 2 (1), 4–32.

Goldfarb, D., Reid, J., 1977. A practicable steepest-edge simplex algorithm. Mathematical Programming 12 (1), 361–371.

Greeff, G., 2005. The revised simplex algorithm on a GPU. Univ. of Stellenbosch, Tech. Rep.

Harris, M., Sengupta, S., Owens, J. D., 2007. Parallel prefix sum (scan) with CUDA. GPU gems 3 (39), 851–876.

Ibarra, O. H., Kim, C. E., Apr. 1977. Heuristic algorithms for scheduling independent tasks on nonidentical processors. Journal of the ACM 24 (2), 280–289.

Janiak, A., Janiak, W., Lichtenstein, M., jul 2008. Tabu Search on GPU. Journal of Universal Computer Science 14 (14), 2416–2427.

Jiening, W., Jiankang, D., Chunfeng, Z., aug. 2009. Implementation of ant colony algorithm based on GPU. In: Sixth International Conference on Com-

puter Graphics, Imaging and Visualization, 2009 (CGIV '09). pp. 50 –53.
Jung, J., O'Leary, D., 2008. Implementing an interior point method for linear programs on a CPU-GPU system. Electronic Transactions on Numerical Analysis 28, 174–189.
Kallioras, N. A., Kepaptsoglou, K., Lagaros, N. D., 2015. Transit stop inspection and maintenance scheduling: A gpu accelerated metaheuristics approach. Transportation Research Part C: Emerging Technologies 55, 246–260.
Kung, H., 1982. Why systolic architectures? IEEE computer 15 (1), 37–46.
Kung, H., Leiserson, C. E., 1978. Systolic arrays (for vlsi). In: Sparse Matrix Proceedings. pp. 256–282.
Lageweg, B., Lenstra, J., Kan, A. R., 1978. A general bounding scheme for the permutation flow-shop problem. Operations Research 26 (1), 53–67.
Lalami, M., 2012. Contribution à la résolution de problèmes d'optimisation combinatoire : méthodes séquentielles et parallèles. Ph. D. Thesis, Université Paul Sabatier, Toulouse.
Lalami, M., Boyer, V., El Baz, D., may 2011a. 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. pp. 1999–2006.
Lalami, M., El Baz, D., may 2012. 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. pp. 1769–1777.
Lalami, M., El Baz, D., Boyer, V., sept. 2011b. Multi GPU implementation of the simplex algorithm. In: 13th IEEE International Conference on High Performance Computing and Communications (HPCC 2011). pp. 179 –186.
Laporte, G., Martello, S., 1990. The selective travelling salesman problem. Discrete Applied Mathematics 26 (2), 193–207.
Li, J., Hu, X., Pang, Z., Qian, K., 2009a. A parallel ant colony optimization algorithm based on fine-grained model with GPU-acceleration. International Journal of Innovative Computing, Information and Control 5 (11), 3707–3716.
Li, J., Zhang, L., Liu, L., 2009b. A parallel immune algorithm based on fine-grained model with gpu-acceleration. In: 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC). pp. 683–686.
Luong, T., 2011a. Métaheuristiques parallèles sur GPU. Ph. D. Thesis, Université Lille 1.
Luong, T. V., 2011b. Métaheuristiques parallèles sur GPU. Ph.D. thesis, Université Lille 1 - Sciences et Technologies.
Luong, T. V., Melab, N., Talbi, E., 2011. GPU-based approaches for multiobjective local search algorithms. a case study: the flowshop scheduling problem. Evolutionary Computation in Combinatorial Optimization, 155–166.

Martello, S., Toth, P., 1990. Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc.

Melab, N., Chakroun, I., Mezmaz, M., Tuyttens, D., sept. 2012. A GPU-accelerated branch-and-bound algorithm for the flow-shop scheduling problem. In: 2012 IEEE International Conference on Cluster Computing (CLUSTER). pp. 10–17.

Meyer, X., Albuquerque, P., Chopard, B., 2011. A multi-GPU implementation and performance model for the standard simplex method. In: Euro-Par 2011. pp. 312–319.

Moz, M., Pato, M. V., 2007. A genetic algorithm approach to a nurse rerostering problem. Computers & Operations Research 34 (3), 667–691.

Naiem, A., El-Beltagy, M., 2009. Deep greedy switching: A fast and simple approach for linear assignment problems. In: 7th International Conference of Numerical Analysis and Applied Mathematics.

Nguyen, H., 2008. GPU GEMS 3. Addison Wesley Professional.

Nikolaos, P., Nikolaos, S., 2013. A computational comparison of basis updating schemes for the simplex algorithm on a cpu-gpu system. American Journal of Operations Research 3, 497.

Ólafsson, S., 2006. Chapter 21 metaheuristics. In: Henderson, S. G., Nelson, B. L. (Eds.), Simulation. Vol. 13 of Handbooks in Operations Research and Management Science. Elsevier, pp. 633 – 654.

Osman, I. H., Laporte, G., 1996. Metaheuristics: A bibliography. Annals of Operations Research 63 (5), 511–623.

Paquete, L., Stützle, T., 2006. Stochastic local search algorithms for multiobjective combinatorial optimization: A review. Tech. rep., Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle.

Pedemonte, M., Alba, E., Luna, F., may 2012. 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. pp. 1778–1786.

Pinedo, M. L., 2012. Scheduling: theory, algorithms, and systems. Springer Science & Business Media.

Pinel, F., Dorronsoro, B., Bouvry, P., 2010. A new cellular genetic algorithm to solve the scheduling problem designed for the gpu. In: Metaheuristics Conference (META).

Pinel, F., Dorronsoro, B., Bouvry, P., 2013. Solving very large instances of the scheduling of independent tasks problem on the GPU. Journal of Parallel and Distributed Computing 73 (1), 101–110.

Ploskas, N., Samaras, N., 2015. Efficient GPU-based implementations of simplex type algorithms. Applied Mathematics and Computation 250, 552–570.

Reinelt, G., 1991. TSPLIB - a traveling salesman problem library. ORSA Journal on Computing 3 (4), 376–384.

Reinelt, G., 1994. The traveling salesman: computational solutions for TSP applications. Springer-Verlag.

Roverso, R., Naiem, A., El-Beltagy, M., El-Ansary, S., Haridi, S., 2011. A GPU-enabled solver for time-constrained linear sum assignment problems. In: 7th International Conference on Informatics and Systems (INFOS). pp. 1–6.

Scavo, T., Aug 2010. Scatter-to-gather transformation for scalability. URL `https://hub.vscse.org/resources/223`

Schrijver, A., 1986. Theory of integer and linear programming. Wiley, Chichester.

Schulz, C., Hasle, G., Brodtkorb, A. R., Hagen, T. R., 2013. GPU computing in discrete optimization. Part II: Survey focused on routing problems. EURO Journal on Transportation and Logistics, 1–28.

Sivanandam, S., Deepa, S., 2007. Introduction to genetic algorithms. Springer.

Spampinato, D., Elster, A., may 2009. Linear optimization on modern GPUs. In: 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 1–8.

Stützle, T., Hoos, H. H., 2000. MAX-MIN ant system. Future generation computer systems 16 (8), 889–914.

Suri, B., Bordoloi, U., Eles, P., 2012. A scalable GPU-based approach to accelerate the multiple-choice knapsack problem. In: Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1126–1129.

Taillard, E., 1993. Benchmark for basic scheduling problems. Journal of Operational Research 64, 278 – 285.

Uchida, A., Ito, Y., Nakano, K., 2014. Accelerating ant colony optimisation for the travelling salesman problem on the gpu. International Journal of Parallel, Emergent and Distributed Systems 29 (4), 401–420.

Whaley, R. C., Dongarra, J., 1999. Automatically tuned linear algebra software. In: Ninth SIAM Conference on Parallel Processing for Scientific Computing. pp. 1–27.

Winston, W. L., Goldberg, J. B., 2004. Operations research: applications and algorithms. Vol. 3. Duxbury press Boston.

You, Y., 2009. Parallel ant system for traveling salesman problem on GPUs. In: Eleventh annual conference on genetic and evolutionary computation. pp. 1–2.

Yu, Q., Chen, C., Pan, Z., 2005. Parallel genetic algorithms on programmable graphics hardware. In: Advances in Natural Computation. Springer, pp. 1051–1059.

Zajíček, T., Šucha, P., 2011. Accelerating a flow shop scheduling algorithmon the GPU. In: Workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP).

Zdeněk, B., Jan, D., Přemysl, Š., Zdeněk, H., 2013. An acceleration of the algorithm for the nurse rerostering problem on a graphics processing unit. Lecture Notes in Management Science 5, 101–110.