Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm

Imen Chakroun, Mohand Mezmaz, Nouredine Melab, Ahcène Bendjoudi

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm

I. Chakroun[1], M. Mezmaz[2], N. Melab[1] and A. Bendjoudi[3]

[1]*Université Lille 1 CNRS/LIFL, INRIA Lille Nord Europe, Cité scientifique - 59655, Villeneuve d'Ascq cedex, France*
[2]*Mathematics and Operational Research Department (MathRO), University of Mons, Belgium*
[3]*CEntre de Recherche sur l'Information Scientifique et Technique (CERIST), 16030 Ben-Aknoun, Algiers, Algeria*

SUMMARY

In this paper, we address the design and implementation of GPU-accelerated Branch-and-Bound algorithms (B&B) for solving Flow-shop scheduling optimization problems (FSP). Such applications are CPU-time consuming and highly irregular. On the other hand, GPUs are massively multi-threaded accelerators using the SIMD model at execution. A major issue which arises when executing on GPU a B&B applied to FSP is thread or branch divergence. Such divergence is caused by the lower bound function of FSP which contains many irregular loops and conditional instructions. Our challenge is therefore to revisit the design and implementation of B&B applied to FSP dealing with thread divergence. Extensive experiments of the proposed approach have been carried out on well-known FSP benchmarks using an Nvidia Tesla C2050 GPU card. Compared to a CPU-based execution, accelerations up to $\times 77.46$ are achieved for large problem instances.

Received . . .

KEY WORDS: GPU computing; Branch-and-bound algorithms; Data parallelism; Thread divergence.

1. INTRODUCTION

Many real world problems encountered in different industrial and economic fields, such as task allocation, job scheduling, network routing, cutting, packing, etc. are of combinatorial nature. Such combinatorial optimization problems (COPs) are known to be large in size and NP-hard to solve. One of the most popular exact methods for solving a COP to optimality, is the Branch-and-Bound (B&B) algorithm. This algorithm is based on an implicit enumeration of all the feasible solutions of the problem to be tackled. The space of potential solutions, called the search space, is explored by dynamically building a tree which root node designates the initial problem. The internal or intermediate nodes represent subproblems obtained by the decomposition of upper subproblems. The leaf nodes designate potential solutions or subproblems that cannot be decomposed. Building the B&B tree and its exploration are performed using four operators: branching, bounding, selection and elimination.

The execution time of B&B often increases significantly with the instance size, and often only small or moderately-sized instances can be practically solved. For this reason, over the last decades, parallel computing has been identified as an attractive way to deal with larger instances of COPs. However, while many contributions have been proposed for parallel B&B methods using Massively Parallel Processors [1], Networks or Clusters of Workstations [17] and SMP machines [2], to

*Correspondence to: imen.chakroun@inria.fr

the best of our knowledge no contribution has been proposed for designing B&B algorithms on Graphical Processing Units (GPUs). For years, the use of graphics processors was dedicated to graphics applications. Driven by the demand for high-definition 3D graphics on personal computers, GPUs have evolved into a highly parallel, multi-threaded and many-core environment. Its utilization has recently been extended to other application domains such as scientific computing [5]. In combinatorial optimization, GPU computing is successfully used for meta-heuristics (near-optimal methods) [12] but not yet for exact methods such as B&B algorithms.

In this work, we rethink the design and implementation of a GPU based B&B algorithm using a node-based parallelization strategy: the parallelization of the bounding operator. Preliminary experiments carried out on some Taillard's problem instances [19], have shown that the evaluation of subproblems takes on average between 98% and 99% of the total execution time of the B&B. No doubt, such observations completely justify the parallelization strategy. However, revisiting this parallel model for GPUs architectures is not straightforward. Indeed, several challenges mainly related to the characteristics of the GPU architecture have to be considered. Firstly, having in mind that the execution model of GPUs is Single Instruction Multiple Data (SIMD), irregular computations (kernels) containing loops and conditional instructions lead to a challenging issue: the thread or branch divergence. Such problem arises when for instance the threads of the same warp (the smallest executable unit of parallelism on a the GPU) [†] have to execute simultaneously different branches of a conditional instruction. Since the GPU SIMD model assumes that a warp executes one common instruction at a time, the different branches of a conditional instruction which is data-dependent are then run in a serial way slowing down the execution time.

Another challenging concern of GPU-based B&B is supplying the device with a large pool of subproblems. In fact, in [3] it has been proven that better efficiency is obtained when the device is supplied with a large amount of subproblems to be evaluated in parallel. This behavior results from the GPU parallel model which is based on a high degree of multi-threading and for which a large data parallelism is required to exploit all the multi-processors of a GPU device.

The major contribution of this paper consists in rethinking the design of B&B algorithms to allow highly efficient solving of large instances of the Flow-shop Permutation Problem (FSP) on GPU accelerators. To do so, the challenge is twofold: (1) defining a new selection operator that could supply the GPU device with a large number of subproblems to evaluate. (2) proposing a new approach for thread/branch divergence reduction through a thorough analysis of the different loops and conditional instructions of the bounding function. Our approach is validated using standard flow-shop instances defined by Taillard [19].

This paper is organized in five main sections. Section 2 summarizes some works related to thread divergence reduction in GPU. Section 3 presents B&B algorithm in general, its parallelization, flow-shop scheduling problem, and GPU. In Section 4, the thread divergence issue related to the location of nodes in the B&B tree and to the control flow instructions within the bounding operator is described. An overview of the GPU memory hierarchy and the used memory access pattern is also given. Section 5 details our GPU-accelerated B&B algorithm. In Section 6, the obtained experimental results are reported. The paper is ended by the conclusions drawn from this work and their perspectives.

2. RELATED WORK

Using GPUs has become increasingly popular in high performance computing. Indeed, such resources supply a substantial computational horsepower and a remarkably high memory bandwidth compared to CPU-based architectures. A large number of optimizations have been proposed to improve the performance of GPU programs. The majority of these optimizations target the GPU memory hierarchy by adjusting the pattern of accesses to the device memory [18]. In contrast, there

[†] 32 threads in the G80 GPU model

DOI: 10.1002/cpe

has been less work on optimizations that tackle another fundamental aspect of GPU performance, namely its SIMD execution model. This section presents some major existing related works.

Dynamic Warp Formation (DWF) [6] is a hardware mechanism proposed in order to improve the efficiency of SIMD branch execution. Every cycle the thread scheduler recomposes warps from the active threads by grouping those that are executing the same path into the same warp. To achieve this, DWF requires an additional hardware that does thread regrouping. Meng *et al.* [15] also propose a hardware mechanism. This tool called Dynamic Warp Subdivision (DWS), splits a warp into sub-warps at divergent branches that can be scheduled independently.

In [20], the proposed approach performs a runtime data remapping across multiple warps. It is proposed that the remapping happens on the fly because of the dependence of thread divergences on runtime values. The major inconvenient with this approach is that it requires a CPU-GPU pipelining scheme, feature that incurs extra host-device communications.

In [9], the authors intervene at code level and introduce two software-based optimizations: iteration delaying and branch distribution. Iteration delaying improves the utilization of execution units in the presence of a divergent branch within a loop, by executing only one branch path at each iteration and delaying the threads that follow the other path until later iterations. Branch distribution aims to reduce the divergent portion of a branch by factoring out structurally similar code from the branch paths.

The existing techniques for handling branch divergence either demand hardware support [6] or require host-GPU interaction [20], which incurs overhead. Some other works such as [9] intervene at the code level. They expose a branch distribution method that aims to reduce the divergent portion of a branch by factoring out structurally similar code from the branch paths. In our work, we have also opted for software-based optimizations like [9]. In fact, we figure out how to literally rewrite the branching instructions into basic ones in order to make thread execution paths uniform. We also demonstrate that we could ameliorate performances by appropriately reordering data being assigned to each thread.

3. GPU-BASED PARALLEL B&B: ISSUES AND CHALLENGES

3.1. Branch-and-bound algorithm

The B&B algorithm is one of the most used algorithms to solve exactly combinatorial optimization problems. Solving exactly a combinatorial optimization problem consists in finding a solution having the optimal cost. This algorithm is based on an implicit enumeration of all the solutions of the problem being solved. The space of potential solutions (search space) is explored by dynamically building a tree which root node represents the initial problem. The leaf nodes are the possible solutions and the internal nodes are subspaces of the total search space. Each internal node of the tree represents a subproblem of the problem to be solved.

The construction of such a tree and its exploration are performed using four operators: branching, bounding, selection and pruning. The bounding operator is used to compute a bound value of the optimal solution of the subproblem being tackled. The pruning operator uses this bound to decide whether to prune the subproblem or to continue its exploration. The selection operator selects one subproblem among all pending subproblems according to an exploration strategy.

The selection of a subproblem could be based on its depth in the B&B tree which leads to a depth-first exploration strategy. A selection based on the breadth of the subproblem is called a breadth-first exploration. A best-first selection strategy could also be used. It is based on the presumed capacity of the node to yield good solutions.

3.2. Flow-shop scheduling problem

The flow-shop belongs to the category of scheduling problems. A scheduling problem is defined by a set of jobs and resources. The flow-shop is a multi-operation problem, where each operation is the execution of a job by a machine. In this problem, the resources are machines in a production workshop. The machines process jobs according to the production chain principle. The machines

DOI: 10.1002/cpe

are arranged in a certain order. Thus, a machine cannot start processing a job if all the machines, which are located upstream, did not finish their treatment. Duration is associated to each operation. This duration is the time required for the machine to finish its treatment. An operation cannot be interrupted, and the machines are critical resources, because a machine processes one job at a time. The objective is to find a solution that minimizes the *makespan*. In [7], it was shown that the minimization of *makespan* is NP-hard from 3 machines upwards.

The performance of a B&B algorithm depends mainly on the relevance of the used bounding operator. The lower bound proposed by Lageweg et al. [11] is used in our bounding operator. This bound is known for its strong results with a complexity of $O(M^2Nlog(N))$ where N is the number of jobs and M the number of machines. This lower bound is mainly based on Johnson's theorem [10] which provides a procedure for finding an optimal solution for flow-shop scheduling problem with 2 machines. Johnson algorithm assumes to assign jobs at the beginning and at the end of a partial schedule associated with a subproblem. Figure 1 shows an example of a partial schedule. In this schedule, jobs 1 and 2 are scheduled at the beginning, before the index *limit1*, jobs 9 and 10 are scheduled in the end, after the index *limit2*, and the other jobs are not yet scheduled.

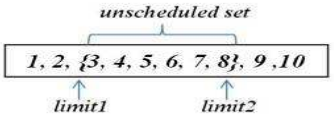

Figure 1. Representation of a partial schedule associated with a subproblem

3.3. *Parallel branch-and-bound*

Thanks to the bounding operator, B&B allows to reduce considerably the computation time needed to explore the whole solution space. However, the exploration time remains significant and parallel processing is thus required. In [13], three parallel models are identified for B&B algorithms: (1) the parallel multi-parametric model , (2) the parallel tree exploration, and (3) the parallel evaluation of the bounds. The model (1) consists in launching simultaneously several B&B processes. These processes differ by one or more operator(s), or have the same operators differently parameterized. The trees explored in this model are not necessarily the same. Model (2) consists in launching several B&B processes to explore simultaneously different paths of the same tree. Unlike the two previous models, model (3) suppose the launching of only one B&B process. It does not assume to parallelize the whole B&B algorithm but only the bounding operator. Each calculation unit evaluates the bounds of a distinct pool of subproblems. The model (3) suits for GPU computing and will be used in our approach. In fact, bounding is often a very time consuming operation. Preliminary experiments are carried out on some Taillard's problem instances [19] by running the sequential version of the B&B algorithm during 600 seconds. As reported in Table I, the results of these experiments show that the bounding of subproblems takes on average between 97% and 99% of the total execution time of this algorithm.

Type of instances	B&B (seconds)	Bounding (seconds)	B&B - Bounding (seconds)	Bounding / B&B (%)
200×20	600	582,968	17,032	97,16%
100×20	600	591,329	8,671	98,55 %
50×20	600	592,560	7,440	98,76 %
20×20	600	592,785	7,215	98,79 %

Table I. Duration of the bounding operator compared to the duration of the whole B&B algorithm.

DOI: 10.1002/cpe

3.4. Graphics processing unit

When a GPU application runs, each GPU multiprocessor is given one or more thread block(s) to execute. Those threads are partitioned into warps that get scheduled for execution. In this paper, the G80 model, in which a warp is a pool of 32 threads, is used. At any clock cycle, each processor of the multiprocessor selects a half warp that is ready to execute the same instruction on different data. The GPU SIMD model assumes that a warp executes one common instruction at a time. Consequently, full efficiency is realized when all 32 threads of a warp agree on their execution path. However, if threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken. Threads that are not on that path are disabled, and when all paths complete, the threads converge back to the same execution path. This phenomenon is called *thread divergence* and often causes serious performance degradations.

The parallel evaluation of bounds is a subproblem-based parallelism. This feature implies an irregular computation depending on the data of each subproblem. Irregularities calculation are reflected in several control flow instructions that would conduct to different behaviors. As explained before, such data-dependent conditional branches are the main cause of thread divergence.

4. THREAD DIVERGENCE ISSUE

This section discusses thread divergence issue encountered when computing the bounds by GPU. The thread divergence occurs for two main reasons, namely the locations of nodes in the search tree and the control flow instructions within the bounding operator.

4.1. Divergence related to the location of nodes

This divergence is related to the positions of the nodes in the B&B search tree. The position of a node determines the values of *limit1* and *limit2*. In the bounding operator, the thread divergence between two nodes N_1 and N_2 occurs if *limit1* of N_1 is not equal to *limit1* of N_2 or *limit2* of N_1 is not equal to *limit2* of N_2. The more these differences are big, the more the thread divergence increases. Indeed, the execution flow of the bounding operator depends on the values of *limit1* and *limit2* of the node.

Below is an example from the source code of our bounding operator showing that the execution flow depends on the values of *limit1* and *limit2*. In this source code written in C++, three methods are used *is_leaf()*, *makespan()* and *lower_bound()*. *is_leaf()* tests if the node *_node* is a leaf or an internal node. If *_node* is a leaf, *makespan()* computes the cost of its makespan. Otherwise, *_node* is an internal node and *lower_bound()* computes the value of its lower bound. *is_leaf()* uses the values of *limit1* and *limit2* to determine whether *_node* is a leaf or an internal node. *_node* is a leaf if the difference between its *limit1* and *limit2* is equal to 1.

```
if (_node.is_leaf())
   return _node.makespan();
else
   return _node.lower_bound();
```

4.2. Divergence related to the control flow instructions

Control flow refers to the order in which the instructions, statements or function calls are executed in a program. This flow is determined by instructions such as *if-then-else*, *for*, *while-do*, *switch-case*, etc. There are a dozen of such instructions in the implementation of our bounding operator. The source code examples below show two scenarios in which this kind of instructions is used.

- Example 1:

```
if( pool[thread_idx].begin != 0 )
   time = TimeMachines[1] ;
```


DOI: 10.1002/cpe

```
else
   time = TimeArrival[1] ;
```

- Example 2:

```
for(int k = 0 ; k < pool[thread_idx].begin; k++)
   jobTime = jobEnd[k] ;
```

In these two examples, *thread_idx* is the index associated to the current thread. Let suppose that the code of Example 1 is executed by 32 threads, *pool[thread_idx].begin* is equal to 0 for the first thread, and *pool[thread_idx].begin* is not equal to 0 for the other 31 threads. When the first thread executes the statement *"time = TimeArrival[1];"*, all the other 31 threads remain idle. Therefore, the GPU cores on which these 31 threads are executed remain idle and cannot be used during the execution of the statement *"time = TimeArrival[1];"*.

The same scenario occurs during the execution of the code of Example 2. Let suppose that the code of Example 2 is executed by 32 threads, *pool[thread_idx].begin* is equal to 100 for the first thread, and *pool[thread_idx].begin* is equal to 0 for the other 31 threads. When the first thread executes the loop for, all the other 31 threads remain idle.

5. OUR GPU-ACCELERATED BRANCH-AND-BOUND ALGORITHM

This section begins by explaining the overall architecture of our approach. One of the main goals of this approach is to supply the GPU with enough subproblems. Then the section describes the thread-data reordering technique which purpose is to supply the GPU by homogeneous subproblems. Afterwards, the branch refactoring technique is explained. Its objective is to avoid the thread divergence which is the consequence of the control flow instructions within the bounding operator. The section ends by describing the memory access pattern adopted in our GPU-based B&B.

5.1. Overall architecture of our approach

As illustrated in Figure 2, our approach introduces three main adaptations, shown in gray, on the sequential B&B in order to parallelize it and use the GPU.

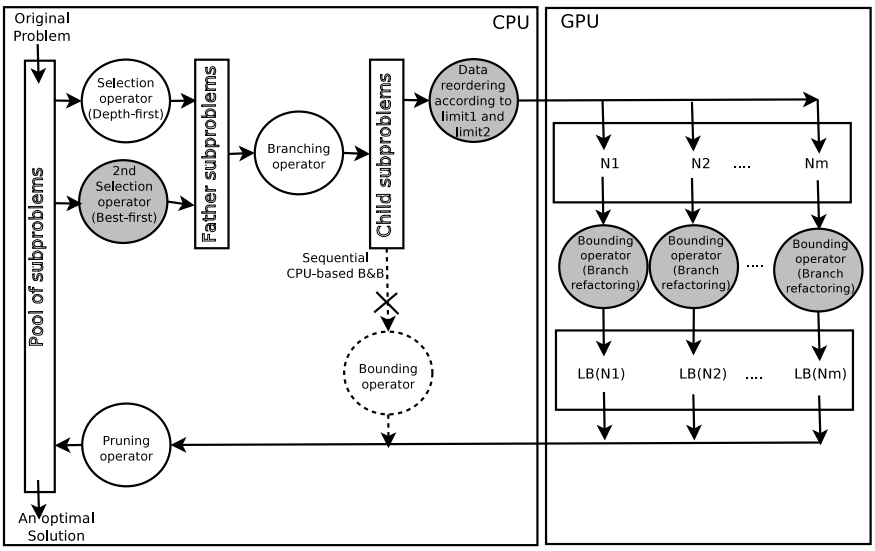

Figure 2. The overall architecture of our GPU-accelerated branch-and-Bound algorithm

The first adaptation is the use of a second selection operator. The depth-first strategy is often used in a classical sequential B&B. This strategy guarantees that the number of subproblems in the pool

DOI: 10.1002/cpe

will not exceed a certain size. Therefore, the depth-first strategy allows to control the size of the used memory. However, the GPU requires to be provided by a large number of subproblems to exploit its computing power. And a selection operator based on depth-first strategy is often not enough to provide the GPU by this huge number of subproblems. For this reason, a selection operator based on the best-first strategy is added to our approach. This operator allows to significantly increase the number of subproblems in the pool. Our B&B uses the selection operator based on the best-first strategy until the pool size reaches a certain maximum size. Once this maximum size is reached, our B&B uses the selection operator based on the depth-first strategy until the pool size reaches a certain minimum size. And once this minimum size is reached, our B&B again uses the selection operator based on the best-first strategy.

The second adaptation is related to the use of a new operator to sort the subproblems before sending them to the GPU. This operator allows the GPU to receive subproblems which are more homogeneous and less heterogeneous. Section 5.2 presents the thread-data reordering technique used to sort these subproblems.

The third adaptation is related to the bounding operator. In our new approach, the bounding operator is not run on the CPU but on the GPU. The GPU launches several bounding operator at the same time. Each of these operators accepts as an input a subproblem and gives as an output its lower bound. A technique of branch refactoring is used to reduce the impact of the thread divergence related to the control flow instructions within the bounding operator. This branch refactoring technique is presented in Section 5.3.

5.2. Thread-data reordering

In a GPU, a block of threads is executed in groups of 32 threads. Each of these groups is called a warp. At any time, all threads in a warp execute the same instruction. So, the GPU is organized as multiple SIMD groups called warps. Some of the threads in the last warp do not execute any instruction if the total number of threads in a block is not divisible by 32. As already written, there is some performance loss if threads within a single warp follow divergent execution paths.

In our B&B approach, each GPU thread computes the lower bound of one subproblem. Therefore, each warp computes the lower bounds of 32 subproblems. Our B&B algorithm executes several iterations, and at each iteration, several thousands of subproblems are sent to the GPU. The GPU groups the received subproblems into several warps according to their reception order. The first 32 subproblems belong to the first warp, the following 32 subproblems belong to the second warp, etc.

Therefore, thread-data reordering technique sorts subproblems before sending them to the GPU. These subproblems are sorted according to the values of *limit1* and *limit2* (i.e. their position in the B&B tree). This sorting of subproblems lets warps work on more homogeneous subproblems, and reduces the number of thread divergences.

5.3. Branch refactoring

As mentioned, thread or branch divergence occurs when the kernel includes conditional instructions and loops that make the threads executing it performing different control flows leading to their serial execution. For FSP, such behavior is due to the lower bound function which contains several conditional instructions and loops that depend on the data associated to the subproblem on which it is applied. In this paper, we investigate the branch refactoring approach to deal with thread divergence for solving FSP on GPU. Branch refactoring consists in rewriting the conditional instructions so that threads of the same warp execute an uniform code avoiding their divergence. To do that, two major "if" scenarios are studied and some optimizations are proposed accordingly. These two scenarios correspond to the conditional instructions contained in the LB kernel code. In the first scenario, the conditional expression is a comparison of the content of a variable to 0. For instance, the following example extracted from the pseudo-code of the lower bound LB illustrates such scenario.

$$\begin{array}{ll} if(pool[thread\_idx].limit1 \neq 0) & tmp = MM[1]; \\ else & tmp = RM[1]; \end{array}$$

DOI: 10.1002/cpe

The refactoring idea is to replace the conditional expression by two functions namely f and g as shown in Equation 1.

$$\begin{array}{l} if(x \neq 0)\ a = b[1]; \\ else\ \ a = c[1]; \\ \Rightarrow a = f(x) \times b[1] + g(x) \times c[1]; \end{array} \quad \Rightarrow \quad \begin{array}{l} if(x \neq 0)\ a = b[1] + 0 \times c[1]; \\ else\ \ a = 0 \times b[1] + c[1]; \end{array}$$

where:

$$\text{f(x)=}\begin{cases} f(x) = 0 & if \quad x = 0 \\ 1 & else \end{cases}$$

and

$$\text{g(x)=}\begin{cases} g(x) = 1 & if \quad x = 0 \\ 0 & else \end{cases}$$

(1)

The behavior of f and g fits the cosine trigonometric function. These functions return values between 0 and 1. An integer variable is used to store the result of the cosine function. Its value is 0 or 1 since it is rounded to 0 if it is not equal to 1. In order to increase the performance the CUDA runtime math operations are used: $sinf(x)$, $expf(x)$ and so forth. Those functions are mapped directly to the hardware level [22]. They are faster but provide lower accuracy which does not matter in our case because the results are rounded to int. The throughput of $sinf(x)$, $cosf(x)$, $expf(x)$ is one operation per clock cycle [22]. The refactoring result for the "if" pseudo-code given above is the following:

```
int coeff = __cosf (pool[thread_idx].limit1);
tmp = (1 - coeff) × MM[1] + coeff × RM[1];
```

The second "if" scenario considered in our study compares two values between themselves as shown in Equation 2.

$$\begin{array}{l} if(x > y)a = b[1]; \quad \Rightarrow if(x - y \geq 1)a = b[1]; \\ \Rightarrow \quad if(x - y - 1 \geq 0) \quad a = b[1]; \quad (x, y) \in N \\ \Rightarrow \quad a = f(x, y) \times b[1] + g(x, y) \times a; \end{array}$$

where:

$$\text{f(x,y)=}\begin{cases} 1 & if \quad x - y - 1 \geq 0 \\ 0 & if \quad x - y - 1 < 0 \end{cases}$$

and

$$\text{g(x,y)=}\begin{cases} 0 & if \quad x - y - 1 \geq 0 \\ 1 & if \quad x - y - 1 < 0 \end{cases}$$

(2)

For instance, the following example extracted from the pseudo-code of the lower bound LB illustrates such scenario.

```
if(RM[1]] > MIN ){  Best_idx = Current_idx; }
```

The same transformations as those applied for the first scenario are applied here using the exponential function. Recall that the exponential is a positive function which is equal to 1 when applied to 0. Thus, if x is greater than y then $expf(x - y - 1)$ returns a value between 0 and 1. If the result is rounded to an integer value 0 will be obtained. Now, if x is less than y then $expf(x - y - 1)$ returns a value greater than 1 and since the minimum between 1 and the exponential is get, the returned result would be 1. Such behavior satisfies exactly our prerequisites. The above "if" instruction pseudo-code is now equivalent to:

```
int coeff = min(1, __expf(RM[1] - MIN - 1));
Best_idx = coeff × Current_idx + ( 1 - coeff ) × Best_idx ;
```

5.4. Memory access Management

Adjusting the pattern of accesses to the GPU device memory grants programmers to further improve the throughput of many high-performance CUDA applications. In [14], optimizing the data access pattern of the proposed parallel bounding approach is investigated. Indeed, having in mind the characteristics of both the lower bound (LB) function and the used GPU accelerator, our challenge

DOI: 10.1002/cpe

is to define an optimal mapping of the large and intermediate data structures of the lower bound function on the hierarchy of memories provided in the GPU device. A careful analysis was required of both the data structures (size and access frequency) and the GPU memories (size and access latency). Such analysis allowed us to identify six data structures for which a complexity analysis in terms of memory size and access frequency is proposed. Regarding the GPU device memories, the focus is put on the shared memory which is a key enabler for achieving higher throughput. We also take care of adequately using the global memory by judiciously configuring the L1 cache that greatly enables improving performance over direct access to global memory. Indeed, in the used C2050 GPU which is based on the Fermi architecture, each multiprocessor is provided with a 64 KB local storage that can be configurable into shared memory and L1 cache. For this reason and in order to achieve further performances, the 64 KB memory is divided according to the experimented scenario. For the scenario were the data structures are put on the shared memory the 64 KB of available storage are split on 48 KB for shared memory and 16 KB for L1 cache. For the scenario where the data sets are put on global memory, 16 KB is used for shared memory and 48 KB for L1 cache.

As quoted above, for B&B applied to the Flow-shop Problem, threads of the same block perform concurrent accesses to the six data structures of the problem when they execute the LB lower bound function. To maximize the throughput, the best mapping of the data structures is to copy them on the shared memory of the GPU device. However, for large problem instances all of the data structures do not fit into the shared memory which size is limited and depends on the GPU hardware configuration. The challenge is therefore to decide which data structure must be put in the shared memory to get the best performance. According to the complexity study performed, the recommendation is to put in the shared memory the Johnson's and the processing time matrices (JM and PTM) if they fit in together. The other data structures are mapped to the global memory combined with the L1 cache.

6. EXPERIMENTS

In the following, we present the experimental study we have performed with the aim to evaluate the performance impact of the GPU-accelerated bounding, the reordering of the nodes to submit to the GPU and the software-based thread divergence reduction using a B&B.

6.1. Experimental settings

In our experiments, the flow-shop instances defined by Taillard [19] are used. These standard instances are often used in the literature to evaluate the performance of methods that minimize the makespan. Optimal solutions of some of these instances are still not known. These instances are divided into groups of 10 instances. In each group, the 10 instances are defined by the same number of jobs and the same number of machines. The groups of 10 instances have different numbers of jobs, namely 20, 50, 10, 200 and 500, and different numbers of machines, namely 5, 10 and 20. For example, there are 10 instances with 200 jobs and 20 machines belonging to the same group of instances. In our experiments, only the instances where the number of machines is equal to 20 are used. Indeed, instances where the number of machines is equal to 5 or 10 are easy to solve. For these instances, the used bounding operator gives so good lower bounds that it is possible to solve them in few minutes using a sequential B&B. Therefore, these instances do not require the use of a GPU.

Our approach has been implemented using C-CUDA 4.0. The experiments have been carried out using a an Intel Xeon E5520 bi-processor coupled with a GPU device. The bi-processor is 64-bit, quad-core and has a clock speed of 2.27GHz. The GPU device is an Nvidia Tesla C2050 with 448 CUDA cores (14 multiprocessors with 32 cores each), a clock speed of 1.15GHz, a 2.8GB global memory, a 49.15KB shared memory, and a warp size of 32.

In order to evaluate the performances of our approach, we need to compute its speedup. This speedup is obtained by comparing our GPU B&B version to a sequential B&B version deployed on one CPU core. However, the resolution of most of Flow-shop instances requires several months of computation on one CPU core. Using the approach defined in [16], it is possible to obtain a

DOI: 10.1002/cpe

random list L of subproblems such as the resolution of L lasts $Tcpu$ when the the sequential B&B algorithm is initialized by (1) this list L and (2) the optimal solution of L. If these two conditions are met, then, for all exploration strategies, (1) the sequential B&B algorithm always explores the same sub-problems, and (2) the resolution time of this sequential algorithm is always the same regardless the used strategy. Furthermore, the subproblems explored by the GPU and CPU B&B versions will be exactly the same. Therefore, it will be possible to initialize the pool of our GPU B&B with the same list L of subproblems and the optimal solution of this list in order to compute the speedup. Let suppose that the resolution of the GPU B&B will last $Tgpu$. So the speedup of our GPU algorithm will be equal to $Tcpu/Tgpu$. In our experiments, the chosen value of $Tcpu$ increases with the size of the instance in order to be sure that the number of subproblems explored is significant for all instances. These values are 10, 50, 150, 300 minutes for the instances while the number of jobs are 20, 50, 100, et 200, respectively.

6.2. Tuning the number of blocks and number of threads

At the launch of a kernel function on a GPU, it is necessary to specify the number of blocks N and the size S of each block. Therefore, N×S is the pool size of a kernel call. N×S threads are run when the pool size is equal to N×S. For reasons related to the architecture of GPU, the number of blocks and the size of a block are set by programmers to powers of 2. For each instance, the following powers 2^4, 2^5, 2^6, 2^7, 2^8, 2^9 and 2^{10} (i.e. 16, 32, 64, 128, 256, 512 and 1024) are tested for the number of blocks. The preliminary experiments, reported in Table II, show the comparison between average execution time for all the instances obtained with different number of blocks and block sizes. The first column represents the size of the pool off loaded to the GPU. The other columns give the corresponding number of blocks, number of threads per block and average normalized execution time. The average normalized execution times are calculated for the instances with 20, 50, 100 and 200 jobs over 20 machines. For each row, the execution times are normalized and divided by the execution time obtained with the pair (number of blocks $\times$ number of threads per block) given the same pool size and having the lower number of block. For instance, for a pool size of 4096, all the execution times are divided by the execution time obtained using 16 blocks and 256 threads per blocks. For this pool size, the obtained execution time using 32 blocks and 128 threads per blocks is almost half (54%) of the execution time obtained using 16 blocks and 256 threads per blocks.

During the tuning process, the primary concern when choosing the number of blocks per grid was keeping the entire GPU busy. Indeed, this parameter should be larger than the number of multiprocessors of the used device so that all multiprocessors have at least one block to execute. Thus, the number of blocks is first initialized as the nearest power of 2 from the number of the multiprocessors detected. Namely on the C2050 GPU used card, there are 14 multiprocessors so we started the number of blocks from 16.

Results show that the worst execution times are always obtained with a number of blocks equal to 16. As quoted above, with less than 16 blocks some of the multiprocessors of the device are idle. With 16 blocks all the multiprocessors are used, however there is only one block per multiprocessor which does not allow to hide the latency of the memory. With more than 16 blocks the speed scales better. Results also show that for all the pool sizes except 4096 a block size equal to 256 gives the best results. The block size (i.e. the number of threads per block) is equal to 256 in all our experiments. For almost all pool sizes, using a number of threads equal to 256 gives the best execution time. For a pool size of 4096 subproblems, using 256 threads with a number of blocks equal to 16 decreases the execution time.

The experimentally found best value for the block size (i.e. 256) was consolidated using the CUDA occupancy calculator provided by Nvidia. This tool allows to easily calculate the best block size based on register and shared memory usage of the kernel.

6.3. Performance evaluation

6.3.1. **Performance of our GPU B&B-based approach.** Table III is organized into two parts. The first part of the table gives the size of the pool to be submitted to the GPU. The second part of the table gives the average speedup related to each group of instances and to each pool size. Each line

DOI: 10.1002/cpe

Pool size	**#Blocks×#Threads**	16×256	**32×128**	64×64	128×32	256×16		
4096	**Average normalized execution time**	1	**0.547**	0.579	0.762	0.859		
Pool size	**#Blocks×#Threads**	16×512	**32×256**	64×128	128×64	256×32	512×16	
8192	**Average normalized execution time**	1	**0.503**	0.524	0.606	0.722	0.808	
Pool size	**#Blocks×#Threads**	16×1024	32×512	**64×256**	128×128	256×64	512×32	1024×16
16384	**Average normalized execution time**	1	0.523	**0.494**	0.549	0.600	0.733	0.813
Pool size	**#Blocks×#Threads**	32×1024	64×512	**128×256**	256×128	512×64	1024×32	
32768	**Average normalized execution time**	1	0.948	**0.898**	0.969	1.083	1.336	
Pool size	**#Blocks×#Threads**	64×1024	128×512	**256×256**	512×128	1024×64		
65536	**Average normalized execution time**	1	0.924	**0.864**	0.959	1.073		
Pool size	**#Blocks×#Threads**	128×1024	**256×512**	512×256	1024×128			
131072	**Average normalized execution time**	1	**0.939**	1.117	1.128			
Pool size	**#Blocks×#Threads**	256×1024	512×512	**1024×256**				
262144	**Average normalized execution time**	1	0.922	**0.866**				

Table II. Average normalized execution times as a function of the number of blocks and the number of threads per block.

Pool size (N × S)	4,096	8,192	16,384	32,768	65,536	131,072	262,144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	42.83	56.23	57.68	61.21	66.75	68.30	**71.69**
100×20	42.59	56.18	57.53	60.95	65.52	65.70	**65.97**
50×20	42.57	**56.15**	55.69	55.49	55.39	55.27	55.14
20×20	38.74	**46.47**	45.37	41.92	39.55	38.90	38.40
Total average speedup	41.68	53.76	54.07	54.89	56.80	57.04	57.80

Table III. Parallel efficiency for different problem instances and pool sizes.

in this part of the table is related to a group of 10 instances defined by the same number of jobs and the same number of machines which is always equal to 20. Each column of this part gives the average speedup obtained by our GPU B&B version using a certain pool size. For example, for the 10 instances with 200 jobs and 20 machines, the average speedup obtained is equal to 42.83 when the pool size is equal to 4,096.

In this section, we experiment the effectiveness of the parallelization of the bounding operator in a B&B algorithm. The objective here is to demonstrate that the use of GPU for evaluating in parallel a selected pool of subproblems allows to significantly accelerate the execution of the B&B. The experimental results are reported in Table III. The results show that evaluating in parallel the bounds of a selected pool, allow to significantly speedup the execution of the B&B. Indeed, an acceleration factor up to 71.69 is obtained for the 200 × 20 problem instances using a pool of 262,144 (1024 × 256) subproblems. The results show also that the parallel efficiency grows with the size of the problem instance. For a fixed number of machines (here 20 machines) and a fixed pool size, the obtained speedup grows accordingly with the number of jobs. For instance for a pool size of 262144, the acceleration factor obtained with 200 jobs (71.69) is almost the double of the one obtained with 20 jobs (38.40). As far the pool size tuning is considered, we could notice that whatever the FSP instance is, the pool size has an important impact on the performance of a GPU based B&B applied to FSP. However, the results show that this parameter depends strongly on the problem instance being solved. It is thus hard to be fixed a priori and so has to be tuned dynamically depending on the problem. To deal with this issue, an empirical heuristic for parameters auto-tuning at runtime is proposed in [4]. The idea of the heuristic is to dynamically tune the size of the pool being off-loaded to the GPU taking into consideration both the characteristics of the used device and the problem instance being tackled.

6.3.2. **Data reordering performances.** As explained in Section 4, any control flow instruction (if, switch, for, while) can significantly affect the instruction throughput by causing threads of the same warp to diverge. If this happens, the different paths are executed in a serial way, increasing the total

DOI: 10.1002/cpe

amount of instructions executed for this warp. Those threads execution path are data-dependent, this means that the data input set of a thread determines its behavior in a given kernel. Starting from this observation, we propose to reorder the data sets that the GPU threads work on. The purpose of thread-data reordering is essentially to find an appropriate mapping between threads and input sets. In our work, we propose a reordering based on the data of the thread rather than its identifier like it is usually done. Let us remember that the proceeding of our proposed GPU-based approach assumes selecting a pool of subproblems at the same time from the search tree. Recall also that the insertion in the tree is performed based on the depth of the subproblems and their lower bounds. Thus the selected pool may contain subproblems from different levels in the tree which implies different data. Before off-loading the pool of subproblems to the GPU device, we sorted the pool of selected subproblems to be evaluated in parallel according to their data particularly *limit1* and *limit2*. Those values correspond respectively to the begin and the end of the range of the unscheduled jobs. This sorting process is used in order to make the pool as homogeneous as possible.

Pool size (N × S)	4,096	8,192	16,384	32,768	65,536	131,072	262,144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	44.04	57.67	60.13	63.10	68.94	71.23	**74.20**
100×20	43.93	57.43	58.11	60.95	62.47	66.30	**66.66**
50×20	43.58	**57.26**	56.81	56.73	56.54	56.28	55.93
20×20	39.92	**48.58**	47.53	44.72	41.14	40.63	40.59
Total average speedup	42.87	55.24	55.65	56.76	58.22	58.61	59.35

Table IV. Parallel efficiency for different problem instances and pool sizes using a sorted pool.

In the following, we study the impact of using a homogeneous pool to be off-loaded to the device on the performance of the GPU accelerated B&B. The results, reported in Table IV, show that reordering data makes the kernel run fast with a homogenous pool than with an unordered pool. Indeed, the approach improves the GPU acceleration compared to the results reported in Table III whatever the instance and the pool size are. This is expected since assembling the subproblems by their values of *limit1* and *limit2* allows to reduce the impact of conditional instructions that depend on these values.

6.3.3. **Branch refactoring performances.** The objective here is to demonstrate that the thread divergence reduction has an impact on the performance of the GPU accelerated B&B and to evaluate how this impact is significant. Table V shows the experimental results obtained using the refactoring approach presented in Section 4. Results show that the refactoring based optimizations accentuate the GPU acceleration reported in Table III et Table IV and obtained without thread divergence reduction. For example, for the instances of 200 jobs over 20 machines and a pool size of 262144, the average reported speedup is 77.46 while the average acceleration factor obtained without thread divergence management for the same instances and the same pool size is 74.20 which corresponds to an improvement of 4.21%. Such considerable but not outstanding improvement is predictable, as claimed in [9], since the factorized part of the branches in the FSP lower bound is very small.

Pool size (N × S)	4,096	8,192	16,384	32,768	65,536	131,072	262,144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	46.63	60.88	63.80	67.51	73.47	75.94	**77.46**
100×20	45.35	58.49	60.15	62.75	66.49	66.64	**67.01**
50×20	44.39	**58.30**	57.72	57.68	57.37	57.01	56.42
20×20	41.71	**50.28**	49.19	45.90	42.03	41.80	41.65
Total average speedup	44.52	56.99	57.72	58.46	59.84	60.35	60.64

Table V. Parallel efficiency for different instances and pool sizes using thread divergence management.

DOI: 10.1002/cpe

In order to better investigate the impact of the thread divergence reduction, we draw in Figure 3 the number of divergent branches within a warp measured using the Nvidia Compute Visual Profiler [21]. Counter values obtained from the Compute Visual Profiler are not the same as numbers obtained by inspecting kernel code. They are best used to identify relative performance differences between un-optimized and optimized code. These performance counter values represent events within a thread warp; they do not correspond to individual thread activity. Indeed, the divergent branch counter, we plot, is incremented by one at each point of divergence in a warp: if at least one thread in a warp diverges via a data dependent conditional branch, the counter is incremented.

Figure 3 also shows the time elapsed for executing the instructions contained in the divergent branches. For measuring the latter execution time we used the time function $clock()$ which once executed in the device function returns the value of a per-multiprocessor counter that is incremented every clock cycle. Sampling this counter at the beginning and at the end of all conditional instructions, taking the difference of the two samples and recording the result provides a measure of the number of clock cycles taken by the device to completely execute these divergent instructions.

The reported results show that the number of divergent branches measured using the code optimization we proposed is on average three times less than the number measured without code optimizations. However, the difference between the measured elapsed time for executing conditional instructions with and without code optimization is very tiny (on average around 0.12s). As claimed above, this little difference in execution time is due to the factorized part of the branches in the FSP lower bound which is very small and which explains the obtained improvement.

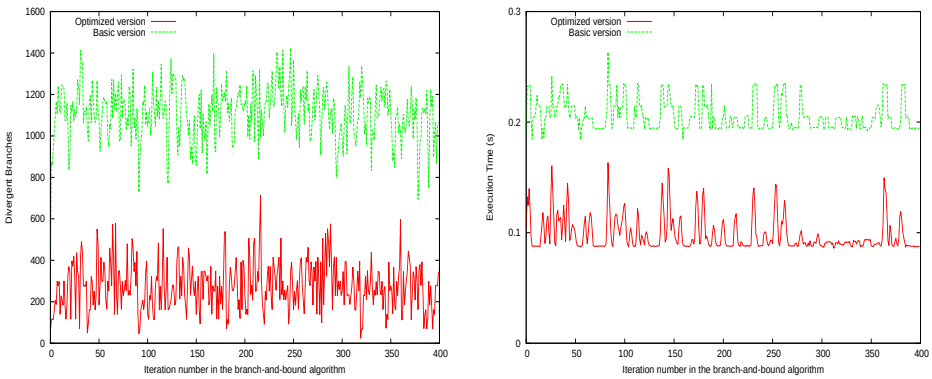

Figure 3. Number of divergent branches and corresponding elapsed time with and without thread divergence reduction

As an additional enhancement of the proposed techniques, the branch refactoring method has been applied to the Monte Carlo simulation for Multi-Layered media (MCML) problem. This real-world medical application is highly parallelizable, where a large number of photons are propagated independently, but according to identical rules and different random number sequences. The parallel nature of this special type of Monte Carlo simulation renders it highly suitable for execution on a GPU. This problem has been chosen in order to compare the proposed contribution with the work proposed in [9]. In [9], the authors also intervene at code level and introduce some software-based optimizations for reducing branch divergence in GPU programs: iteration delaying and branch distribution. Iteration delaying improves the utilization of execution units in the presence of a divergent branch within a loop. Branch distribution aims to reduce the divergent portion of a branch by factoring out structurally similar code from the branch paths. In our work, the focus is on transforming the if-then-else conditional instructions which is more related with the branch distribution method rather than the iteration delay approach that target the loop instructions. For experimentation, the GPU implementation proposed in [8] has been used and tested on the same GPU device used in [9] namely a GTX 480 card. MCML has one kernel where each thread is assigned a number of photons to be simulated. Paths of the if-then-else instructions for which our transformations are applied, contain on average 80 fused multiply add instructions. Table VI reports

DOI: 10.1002/cpe

the improvement percentage obtained when applying the branch refactoring method compared to the original if-then-else instruction. Results show that the improvement grows accordingly to the number of simulated photons. The acceleration achieved by our refactoring method varies from 10% to 29% while the acceleration achieved by the branch distribution proposed in [9] varies from 5.6% to 16.1%.

Number of Photons	10000	50000	100000	500000	1000000	5000000
Improvement of the branch refactoring (%)	10.16 %	12.56 %	16.91 %	22.83 %	25.615 %	29.27 %

Table VI. Improvement obtained for the MCML problem using the branch refactoring method.

6.4. Comparison with a multi-threaded parallel B&B algorithm

In order to further evaluate the performances of the proposed GPU-based B&B algorithm, we compare it to a multi-threaded B&B designed on top of a multi-core system. In this multi-threaded version, CPU threads runs a B&B using a shared queue of sub-problems and shared variable for updating the best solution. The access to the shared data (taking or inserting a sub-problem from the queue, and updating the best solution) is controlled using synchronization primitives to ensure mutual exclusion between threads.

In order to compute a fair comparison with the obtained results of our GPU-based approach, the used multi-core system must have the same computational power in term of theoretical peak of FLOPS. As quoted in 6.1, the experiments have been carried out on a Nvidia Tesla C2050. According to [23], the theoretical double precision floating-point performance peak of this GPU device is about 515 GFLOPS. For the multi-threaded version of the B&B, experimentation have carried out on an Intel Core i7-970 Processor which is 64-bit and composed of six physical cores and 12 threads [26] having each a theoretical double precision floating-point performance peak of 76.8 GFLOPS [25]. Table VII reports the speedup of the parallel multi-threaded B&B averaged on different problem instances (sizes). The columns correspond to the number of parallel running B&B process and the corresponding theoretical peak of GFLOPS. The rows correspond to the problem instances defined by (Number of jobs $\times$ Number of machines). The same experimental protocol as the for GPU computation is used (see section 6.1). The reported speedups are calculated relatively to a serial B&B on a single CPU core. Results shows that the parallel efficiency grows on average with the growing of the number of computing core used. However, the improvement is not linear and the slope decrease as long as the number of the used computing core arises. This behavior might be due to the operating system which handles additional page faults and context switches when the number of threads increases.

Number of B&B Threads	3	5	**7**	9	11
Theoretical Peak of GFLOPS	230.4	384	**537.6**	691.2	844.8
200×20	4.03	6.98	**8.76**	9.04	9.32
100×20	4.27	7.08	**8.82**	9.39	9.85
50×20	4.38	7.27	**9.06**	9.64	10.17
20×20	4.43	7.35	**9.22**	10.04	10.85

Table VII. Parallel efficiency for different problem instances and pool sizes.

The speedups of GPU-based B&B and the multi-threaded based B&B are calculated relatively to the same sequential version of the B&B algorithm. For a same computational power, our approach for designing B&B algorithms on top of GPU accelerators is much more efficient than the multi-threaded B&B whatever the instance is. Indeed, for a computational power around 500 GFLOPS, the acceleration calculated when using the GPU-based B&B for the instances 200 jobs over 20 machines is $\times$ 74.20. For the same category of instances (200 jobs over 20 machines) and a same computational power of 500 GFLOPS which corresponds to 7 CPU computing cores for the Intel Core i7970 Processor, the speedup over a sequential version of the multi-threaded based B&B is

DOI: 10.1002/cpe

8.76. On average over all the experimented instance categories, the GPU-based B&B run 7 times faster than the multi-threaded based B&B.

7. CONCLUSION AND FUTURE WORK

In this paper, we have revisited the design and implementation of B&B algorithms using a parallel bounding model for solving permutation-based combinatorial optimization problems such as FSP on GPU accelerators. The contributions consist in: (1) proposing a GPU-based parallel bounding model; (2) rethinking the selection operator in order to supply the GPU device with a large number of subproblems; (3) proposing a refactoring approach to deal with the thread divergence issue.

In our proposed parallel GPU-based approach, the generation of the subproblems (branching, selection and pruning operations) is performed on CPU and the evaluation of their lower bounds (bounding operation) is executed on the GPU device. According to our selection protocol, a pool of the deepest pending subproblems (thousands of subproblems) and having the smallest lower bound generated on CPU is off-loaded to the GPU device where it is evaluated by a pool of threads. Each thread applies the lower bound function to one subproblem. Once the evaluation is completed, the lower bound values corresponding to the different subproblems is returned back to the CPU to be used by the elimination operator. The process is iterated until the exploration is completed and the optimal solution is found.

The Flow-Shop scheduling problem has been considered as a case study. The proposed approach has been experimented using a Tesla C2050 GPU card on different classes of FSP instances. The experimental results show that accelerations up to $\times 71.69$ can be obtained especially for large problem instances and large pools of subproblems. Results demonstrate also that the size of the pool to be off-loaded to the GPU has an important impact on the performance of the B&B. Since this parameter depends strongly on the problem instance being solved, the recommendation is to tune it dynamically depending on the problem. Experiments show also that the proposed refactoring approach improves the parallel efficiency whatever the FSP instance and the pool size are. However, the improvement was not significant because the factorized part of the branches in the FSP lower bound is very small. The optimizations obtained with the proposed approaches allowed us to achieve accelerations up to $\times 77.46$ compared to a sequential B&B and up to $\times 7$ compared to a multi-threaded B&B.

In the near future, we plan to extend this work to a cluster of GPU-accelerated multi-core processors. From the application point of view, the objective is to optimally solve challenging and unsolved Flow-Shop instances as we did it for one 50×20 problem instance with grid computing [16]. Finally, we plan to investigate other lower bound functions to deal with other combinatorial optimization problems.

8. ACKNOWLEDGMENT

Experiments presented in this paper have been carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several universities as well as other funding bodies (see https://www.grid5000.fr).

REFERENCES

1. R. Allen, L. Cinque, S. Tanimoto, L. Shapiro and D. Yasuda. A parallel algorithm for graph matching and its MasPar implementation. IEEE Transactions on Parallel and Distributed Systems, Volume 8, Number 5, 1997.
2. L.G. Casadoa, J.A. Martneza, I. Garcaa and E.M.T. Hendrixb. Branch-and-Bound interval global optimization on shared memory multiprocessors. Optimization Methods and Software, Volume 23, Number 5, Pages 689-701, 2008.
3. I. Chakroun, A. Bendjoudi, and N. Melab. Reducing thread divergence in GPU-based B&B applied to the Flow-shop problem. 9th International Conference on Parallel Processing and Applied Mathematics PPAM'11, Poland, Torun, September 11-14, 2011 (To appear).

DOI: 10.1002/cpe

4. I.Chakroun and N.Melab. An Adaptative Multi-GPU based Branch-and-Bound. A Case Study: the Flow-Shop Scheduling Problem. The 14th IEEE International Conference on High Performance Computing and Communications (HPCC-2012) Liverpool, England.
5. J.J. Dongarra, D.A. Bader, J. Kurzak. Scientific computing with multi-core and accelerators, CRC Press Inc, ISBN-10 143982536X, ISBN-13: 978-1439825365, Pages 1-514, 2010.
6. W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Micro-architecture, Washington, DC, USA, Pages 407-420, 2007.
7. M.R. Garey, D.S. Johnson and R. Sethi. The complexity of flow-shop and job-shop scheduling. Mathematics of Operations Research, Volume 1, Pages 117-129, 1976.
8. Erik Alerstam, William Chun Yip Lo, Tianyi David Han, Jonathan Rose, Stefan Andersson-Engels, and Lothar Lilge Next-generation acceleration and code optimization for light transport in turbid media using GPUs Biomed. Opt. Express 1, 658-675 (2010)
9. T. Han and T.S. Abdelrahman. Reducing branch divergence in GPU programs. In Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4). ACM, New York, USA, Pages 1-8, 2011.
10. S.M. Johnson. Optimal two and three-stage production schedules with setup times included. Naval Research Logistis Quarterly, Volume 1, Pages 61-68, 1954.
11. J.K. Lenstra, B.J. Lageweg, and A.H.G. Rinnooy Kan. A General bounding scheme for the permutation flow-shop problem. Operations Research, Volume 26, Number 1, Pages 53-67, 1978.
12. T.V. Luong, N. Melab, E. Talbi. GPU computing for parallel local search metaheuristic algorithms. IEEE Transactions on Computers, in press, preprint available in IEEE computer Society Digital Library, 18 October 2011.
13. N. Melab. Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul. HDR thesis, LIFL, USTL, Novembre 2005.
14. N. Melab, I.Chakroun and A.Bendjoudi. GPU-accelerated Bounding for Branch-and-Bound applied to a Permutation Problem using Data Access Optimization. Under submission in Concurrency and Computation: Practice and Experience - Manuscript CPE-12-0085
15. J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In Proc. of ISCA, Pages 235246, 2010.
16. M. Mezmaz, N. Melab and E-G. Talbi. A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. In Proc. of 21th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS), Long Beach, California, March 26th-30th, 2007.
17. M.J. Quinn. Analysis and implementation of branch-and-bound algorithms on a hypercube multicomputer. IEEE transactions on computers, Volume 39, Number 3, Pages 384-387, 1990.
18. S. Ryoo, C.I. Rodrigues, S.S. Stone, J.A. Stratton, S-Z. Ueng, S.S. Baghsorkhi and W-M.W. Hwu. Program optimization carving for GPU computing. J.Parallel Distributed Computing, Volume 68, Number 10, Pages 1389-1401, 2008.
19. E. Taillard. Benchmarks for basic scheduling problems. Journal of Operational Research, Volume 64, Pages 278-285, 1993.
20. E.Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10). ACM, New York, NY, USA, Pages 115-126, 2010.
21. COMPUTE VISUAL PROFILER User Guide. http://developer.Nvidia.com/Nvidia-gpu-computing-documentation#VisualProfiler
22. Nvidia CUDA C Programming Best Practices Guide. http://developer.download.Nvidia.com/compute/cuda/2_3/toolkit/docs/Nvidia_CUDA_BestPracticesGuide_2.3.pdf.
23. http://www.Nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf
24. http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units
25. http://download.intel.com/support/processors/corei7/sb/core_i7-900_d.pdf
26. http://ark.intel.com/products/47933/Intel-Core-i7-970-Processor-%2812M-Cache-3_20-GHz-4_80-GTs-Intel-QPI%29

DOI: 10.1002/cpe