



A GPU-based Branch-and-Bound algorithm using Integer-Vector-Matrix data structure

Jan Gmys, Mohand Mezmaz, Nouredine Melab, Daniel Tuytens

► To cite this version:

Jan Gmys, Mohand Mezmaz, Nouredine Melab, Daniel Tuytens. A GPU-based Branch-and-Bound algorithm using Integer-Vector-Matrix data structure. *Parallel Computing*, 2016, *Parallel Computing*, 59, pp.119-139. 10.1016/j.parco.2016.01.008 . hal-01389471

HAL Id: hal-01389471

<https://inria.hal.science/hal-01389471>

Submitted on 28 Oct 2016

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.

A GPU-based Branch-and-Bound algorithm using Integer-Vector-Matrix data structure

J. Gmys², M. Mezma², N. Melab¹ and D. Tuytens²

¹ INRIA Lille Nord Europe, Université Lille 1, CNRS/LIFL, Cité scientifique - 59655, Villeneuve d'Ascq cedex, France

²Mathematics and Operational Research Department (MARO), University of Mons, Belgium

Abstract

Branch-and-Bound (B&B) algorithms are tree-based exploratory methods for solving combinatorial optimization problems exactly to optimality. These problems are often large in size and known to be NP-hard to solve. The construction and exploration of the B&B-tree are performed using four operators: branching, bounding, selection and pruning. Such algorithms are irregular which makes their parallel design and implementation on GPU challenging. Existing GPU-accelerated B&B algorithms perform only a part of the algorithm on the GPU and rely on the transfer of pools of subproblems across the PCI Express bus to the device. To the best of our knowledge, the algorithm presented in this paper is the first GPU-based B&B algorithm that performs all four operators on the device and subsequently avoids the data transfer bottleneck between CPU and GPU. The implementation on GPU is based on the Integer-Vector-Matrix (IVM) data structure which is used instead of a conventional linked-list to store and manage the pool of subproblems. This paper revisits the IVM-based B&B algorithm on the GPU, addressing the irregularity of the algorithm in terms of workload, memory access patterns and control flow. In particular, the focus is put on reducing thread divergence by making a judicious choice for the mapping of threads onto the data. Compared to a GPU-accelerated B&B based on a linked-list, the algorithm presented in this paper solves a set of standard flowshop instances on average 3.3 times faster.

Keywords: GPU computing, Branch-and-Bound, Combinatorial optimization, Irregular applications

Introduction

Many industrial and economic problems, like flowshop, are permutation combinatorial optimization problems. Solving these problems consists in finding an optimal permutation of elements among a large finite set of permutations. A wide range of these problems are known to be large in size and NP-hard to be solved. The branch-and-bound (B&B) algorithm is one of the most used exact methods to solve these permutation optimization problems. It is based on an implicit enumeration of all the feasible solutions of the problem to be tackled. Building and exploring the B&B tree are performed using four operators: branching, bounding, selection and pruning. In a B&B algorithm, if the lower bound for some tree node A is greater than the best solution found so far for some other node B, then A may be discarded from the search. This key idea of the B&B algorithm significantly reduces the number of explored nodes. However, the execution time of a B&B significantly increases with the size of the instance, and often only small or moderately-sized instances can be practically solved. For this reason, over the last decades, parallel computing has been revealed as an attractive way to deal with larger instances of combinatorial optimization problems.

Because of their massive data processing capability and their remarkable cost efficiency, graphics processing units (GPU) are an attractive choice for providing the computing power needed to solve such instances. While GPU accelerators are used in today's largest high-performance computing systems, their usage is often restricted to regular, data-parallel applications. Indeed, the irregular nature, in terms of workload, control flow and memory access patterns, of applications such as B&B may seriously degrade the performance of the GPU. The acceleration of B&B algorithms using GPUs is therefore a challenging task which is addressed by only a few works in the literature, such as [1], using flowshop as a test case, [2], applied to the travelling salesman problem and [3], applied to the knapsack problem where the search tree is binary. All these approaches use linked-lists (or dequeues, stacks)

to store and manage the pool of subproblems, likewise most parallel B&B algorithms in the literature. Such data structures are very difficult to handle on the GPU and often induce prohibitive performance penalties. For this reason all GPU-accelerated B&B algorithms at our knowledge perform the management of the pool of subproblems at least partially on the CPU, requiring costly data transfers between host and device. In [4] it is shown that the bounding operator for flowshop consumes 97 – 99% of the execution time of a sequential B&B and that the GPU-based parallelization of this operator can provide a substantial acceleration of the algorithm. However, as the management of a list of pending nodes is performed on the CPU, the transfer of data between CPU and GPU constitutes a bottleneck for GPU-accelerated B&B algorithms.

Our parallel GPU-B&B algorithm is, to the best of our knowledge, the first one that implements all four B&B operators on the GPU, requiring virtually no interaction with the CPU during the exploration process. It is based on the Integer-Vector-Matrix (IVM) data structure, a recently developed [5] data structure which allows the efficient storage and management of the pool of subproblems in permutation-based combinatorial optimization problems. In [6] private IVM data structures and IVM-based work stealing techniques are used in a multi-core parallel B&B algorithm. The IVM structure provides some regularization as it allows to store and manage the pool of subproblems with data structures of constant size. However, the IVM-based parallel B&B is still highly irregular in terms of workload, control flow and memory access patterns. None of these three issues can be ignored when implementing the B&B algorithm on the GPU and all three are addressed in this paper. The focus is put on the reduction of thread divergence which arises in CUDA’s SIMD execution model as a consequence of control flow irregularities. For a set of flowshop problem instances that consist in scheduling 20 jobs on 20 machines our IVM-based GPU-B&B processes on average 3.3 times as many nodes per second as the GPU-accelerated linked-list-based (GPU-LL) B&B presented in [1].

The paper is organized in four main sections. Section 1 presents the B&B algorithm in its sequential form, the parallelization model used in our approach and provides some more details on the GPU-LL B&B-algorithm. Section 2 explains the functioning of the Integer-Vector-Matrix (IVM) data structure which is used for the storage and the management of the pool of subproblems. Section 3 describes our GPU-based B&B algorithm and Section 4 proposes alternative mapping schemes for the algorithm with the aim of reducing thread divergence. In Section 5, we report the obtained experimental results, comparing the performance of different mapping schemes and evaluating the performance of our GPU-based algorithm in comparison to a GPU-accelerated linked-list based B&B. Moreover, the scalability of our algorithm is analyzed, considering two different work stealing strategies. The stability of our algorithm towards instances of different size and irregularity is as well investigated. The paper ends with the conclusions drawn from this work and its perspectives.

1. Parallel branch-and-bound algorithms

This section presents the B&B algorithm and its parallelization using different models. The focus is put on the parallel tree exploration model and the parallel evaluation of bounds model which are used in our GPU IVM-based B&B.

1.1. Sequential branch-and-bound

Several exact resolution methods used in combinatorial optimization are branch-and-bound (B&B) like algorithms. These methods are mainly divided into three basic variants: simple B&B, branch-and-cut (B&C), and branch-and-price (B&P). There are other B&B variants less known such as branch-and-peg [7], branch-and-win [8], and branch-and-cut-and-solve [9]. This list is certainly not exhaustive. It is also possible to consider a divide-and-conquer algorithm as a B&B algorithm. It is enough to remove the pruning operator from the B&B to get a divide-and-conquer algorithm. Some authors consider B&C, B&P, and the other variants as different algorithms than B&B. These authors use B&X to refer to algorithms like B&B, B&C, B&P, etc. In what follows, B&B algorithm refers to simple B&B or any other variant of this algorithm.

B&B 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 where, the **root node** represents the initial problem to be solved, the **leaf nodes** are the possible solutions and the **internal nodes** are subspaces of the total search space. Possible solutions are full N -element permutations, like 2134 for $N = 4$. In practice, a solution often corresponds to a scheduling of jobs. Internal nodes can be seen as a partial permutations, consisting

of scheduled and unscheduled jobs. For instance, we will denote $2/13/4$ the subproblem where jobs 1 and 3 are unscheduled while 2 is scheduled in the beginning and 4 is scheduled in the end. The subspace $2/13/4$ contains solutions 2134 and 2314 , so, closer to the leaves the size of the subspaces is smaller and smaller. Using this notation, the initial problem writes $/1234/$. The construction of such a tree and its exploration are performed using four operators: branching, bounding, selection and pruning. B&B proceeds in several iterations where the best solution found so far is saved and can be improved from an iteration to another. All subproblems generated and not yet processed are kept in a data structure, for example a linked-list. At the beginning, this data structure contains the initial problem. Then, at each iteration of the algorithm:

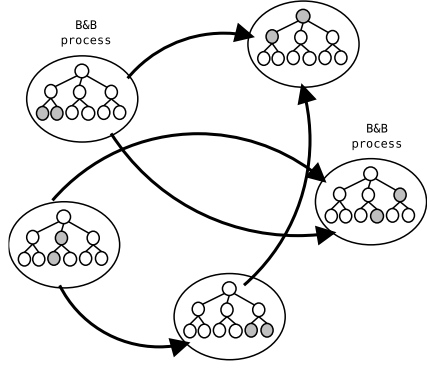
- The **branching operator** partitions a subproblem into several smaller, pairwise disjoint subproblems. For instance, a subproblem with k unscheduled jobs can be decomposed into k subproblems by fixing each unscheduled job either in the beginning or in the end. The generated subproblems are then inserted into the data structure, according to the semantics of the latter.
- The **bounding operator** is used to compute a bound value of the optimal solution of each generated subproblem.
- And the **pruning operator** uses this bound to decide whether to eliminate a subproblem or to continue its exploration.
- The **selection operator** chooses one subproblem among all pending subproblems stored in the data structure 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. In this paper only the *depth-first strategy* is used.

The branching of a subproblem may consist in placing unscheduled jobs either in the beginning (before the first “/”) or in the end (after the second “/”). The choice of the branching rule has an impact on the size of the explored tree. It is possible to take advantage of this by choosing at each iteration the “better” decomposition, according to some heuristic criterion. In our approach both sets of subproblems are generated and evaluated at each iteration, but only the decomposition for which the sum of lower bounds is greater is retained. Indeed, the retained set is likely to contain more subproblems to be pruned, because the average lower bound in this set is greater than in the other set. This branching rule aims at reducing the tree size more efficiently (at the expense computing twice as many bound values per decomposed node).

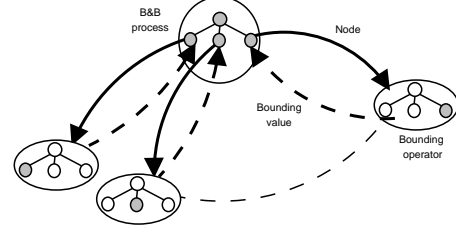
1.2. Parallel branch-and-bound models

B&B algorithms can significantly reduce the computing power needed to explore the whole solution space. However, such power may still be huge, especially when solving large instances. Using many processors or cores in parallel is an effective way to reduce the exploration time. Many approaches to parallelize B&B algorithms are proposed in the literature. A taxonomy of these models is presented in [10]. This taxonomy is based on the classifications proposed in [11] and [12]. Four models are identified: the multi-parametric parallel model, the parallel evaluation of a bound model, the parallel evaluation of bounds model, and the parallel tree exploration model. This paper focuses on the latter two as our GPU-based B&B is based on a combination of these two models.

The **parallel tree exploration model** consists in simultaneously exploring several subproblems that define different search subspaces of the initial problem (Figure 1a). This means that the selection, branching, bounding and pruning operators are executed in parallel, synchronously or asynchronously, by different B&B processes which explore these subspaces independently. In asynchronous mode, the B&B processes communicate in an unpredictable manner to exchange work units and information, such as the best solution found so far. This requires pairwise synchronization between B&B-processes. In multi-core implementations this can be done using mutexes and semaphores. Without such synchronization primitives, the parallel tree exploration is necessarily performed synchronously on GPUs. In synchronous mode, a B&B algorithm has different phases between which the B&B processes are synchronized and may exchange information. Compared to other models, the parallel tree exploration model is more frequently used and is the subject of much research. One important reason is that the degree of parallelism of this model may be important, especially in large instances. Indeed, the number of parallel exploration processes is only limited by the capacity to supply them continuously with subproblems to explore. This work supply depends, on one hand, on the size of the instance being solved. On the other hand, as the B&B tree is



(a) Illustration of the parallel tree exploration model



(b) Illustration of the parallel evaluation of bounds model

Figure 1: Illustrations of models for parallel B&B algorithms

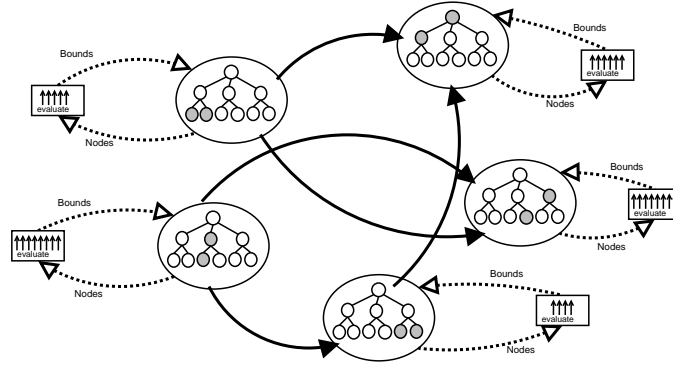


Figure 2: Illustration of the combined parallel tree exploration/parallel evaluation of bounds model.

highly irregular, it depends on the distribution and sharing of the load, which is one of the main issues raised by this model. Among other issues one can include the placement and management of the set of pending subproblems. Also, the communication of the best solution found so far, the detection of the termination of the algorithm and fault tolerance can be challenging, especially in heterogeneous environments [13]. In the case of a GPU implementation other issues arise, such as branch divergence due to control-flow irregularities. Besides potentially yielding a very high degree of concurrency, an important aspect of this model is that it can be combined with other parallel B&B models. At least conceptually, each B&B process that participates in the parallel tree exploration may in turn be parallelized, adding a second level of parallelism.

For instance, each independent B&B process may use the **parallel evaluation of bounds model** (Figure 1b). In this model a single B&B process is launched and the subproblems generated by the branching operator are evaluated in parallel. This model is well-adapted in cases where the cost of the bounding operator is high, compared to the rest of the algorithm. For combinatorial problems this model's degree of parallelism depends on the depth of the current active node in the tree. Moreover the model is data-parallel, synchronous and fine-grained (the cost of the evaluation of a bound) which is the execution model that better fits many-core architectures like GPU. The **combined parallel tree exploration/parallel evaluation of bounds model** (Figure 2) yields a much higher degree of parallelism than using one model alone. In synchronous execution mode the subproblems generated by all B&B processes are evaluated in a single parallel bounding phase. When enough parallel exploration processes are used, the number of generated subproblems per iteration approaches the maximum number of concurrent threads on the GPU. So, it is theoretically possible to reach a very good utilization of the GPU resources. This combined model is used in our GPU-based B&B algorithm.

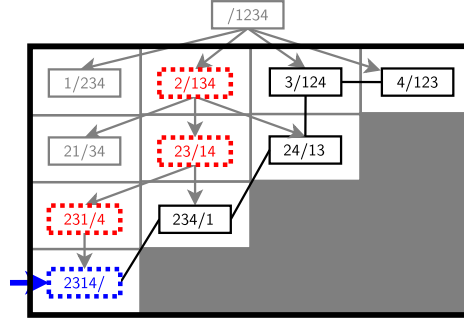


Figure 3: An example of a pool obtained when solving a permutation problem of size 4.

1.3. Related works

In [4] the authors have investigated the benefits of using a GPU for the parallelization of the bounding operator for the flowshop scheduling problem. They have shown that parallel evaluation of subproblems on the GPU can provide a considerable acceleration of the algorithm, as the flowshop bounding operator consumes $> 97\%$ of a the computation time in a sequential B&B. While [4] focuses on optimizing the placements of data in the hierarchical GPU memory, the thread divergence issue is addressed in [14]. The reduction of the overhead induced by data transfers between host and device is another important challenge to be faced when using GPU for the acceleration of B&B. In [1] the two contributions previously cited in this paragraph are extended, proposing an operator-driven approach that implements the branching, bounding and pruning operators as CUDA-kernels. In this approach a pool of nodes is selected on CPU side, according to selection strategy based on the depth of a node, resulting in depth-first exploration. The selected pool is transferred to the device, where the subproblems are branched, the resulting children-nodes evaluated and only the non-pruned children nodes are sent back to the CPU for insertion into the pool of pending subproblems. The pools of subproblems are implemented as stacks for depth-first search (DFS) – however, as other data structures, like priority queues, may be used for other search strategies it will be referred to as “linked-list-based”. The choice of DFS is motivated by the fact that it results in much lower memory requirements than other search strategies, like breadth-first. Branching is performed as described in Subsection 1.1, generating two pools *begin* and *end*, retaining only the one where the sum of lower bounds is greater. A particularity of [1] consists in the dynamic adjustment of the size of the offloaded pools using an auto-tuning heuristic, providing a regularization of the workload. The authors show that branching and pruning on the GPU reduces the amount of data transferred between the host and the device. However, those data transfers remain a bottleneck, even for the algorithm proposed in [1]. The use of linked-list-based data structures actually prevent the efficient implementation of the selection operator inside the GPU. In [5] a multi-core B&B algorithm based on an alternative data structure called IVM has been proposed. This data structure, which is described in the following Section 2, has a constant memory footprint, making it more suitable for a GPU implementation.

2. IVM-based Branch and Bound

This section describes the Integer-Vector-Matrix (IVM) based B&B algorithm. For comparison, Subsection 2.1 explains the working of a conventional linked-list of nodes. This is illustrated with an example of a pool obtained when solving a flowshop instance defined by 4 jobs. Flowshop, as explained in Subsection 5.1, is a permutation problem for which the objective is to find the optimal permutation of jobs according to one criterion or several criteria. This example with 4 jobs is used, in Subsection 2.2, to explain the management of a pool using the IVM data structure. Subsection 2.3 introduces the factorial (or factoradic) number system and Subsection 2.4 describes how intervals of factoradic numbers are used to encode and communicate work units between different IVM-structures.

2.1. Serial linked-list-based B&B

The pool of Figure 3 is represented as a tree in order to visualize the problem/subproblem relationship between nodes, and as a matrix to facilitate the comparison with our IVM-based approach described in Subsection 2.2.

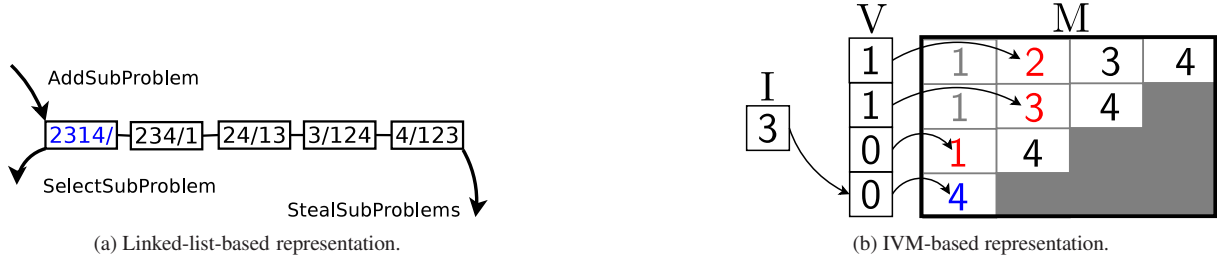


Figure 4: An example of a pool obtained when solving a permutation problem of size 4

However, this pool is usually implemented with a linked-list as shown in Figure 4a. For the sake of simplicity, jobs are only scheduled in the beginning of the partial permutations. For instance, in Figure 3, the node $24/13$ means that job 2 is scheduled at the first position, job 4 at the second position, and jobs 1 and 3 are not yet scheduled. In this figure, dashed nodes represent subproblems which are added into the linked-list and selected from it. At each B&B iteration, the algorithm points to a node of the B&B pool. In the example of Figure 3, the algorithm is currently pointing to the solution $2314/$. Therefore, Figure 4a represents the state of the pool just before removing $2314/$. Before selecting $2314/$, the linked-list contains five nodes, namely $4/123$, $3/124$, $24/13$, $234/1$ and $2314/$.

Before having a linked-list in this state, some operations are applied. At the beginning of the B&B, none of the four jobs is scheduled (i.e. $/1234$). The node $/1234$ is branched/decomposed into four nodes which are $1/234$, $2/134$, $3/124$ and $4/123$. In each of these nodes, one job is scheduled and the three other jobs are not yet scheduled. This example assumes that the first node $1/234$ is processed or pruned, and the algorithm selects and branches the second node $2/134$. The decomposition of this node gives three nodes, namely $21/34$, $23/14$ and $24/13$. The example also assumes that the first node $21/34$ is processed or pruned. Therefore, the algorithm decomposes the second node $23/14$, and obtains two new nodes which are $231/4$ and $234/1$. The node $231/4$ represents a simple subproblem and accepts only one solution $2314/$.

2.2. Serial IVM-based B&B

Figure 4b shows the representation of the state of the pool of Figure 3 using an integer (I), a square matrix (M) of integers and a vector (V) instead of the conventional linked-list used in Subsection 2.1. The size of the square matrix M and the vector V are equal to N , the number of jobs. In this example, $N = 4$. In matrix M a cell with a column number strictly greater than its row number is never used (upper triangular matrix). Each node of the B&B pool is represented by one cell of the matrix. In other words, it is represented by a single integer instead of a permutation of integers. For instance, the first row of M contains all job numbers $1, 2, \dots, N$, each cell representing one of the N subproblems of depth 1 obtained by fixing respectively job $1, 2, \dots, N$ at the first position. After the decomposition of the root node, only the first row of M is filled, all following rows are empty.

- To **select** one of these subproblems on the current level I , the value of $V(I)$ is set such that it points to the corresponding cell. For instance, setting $V(0) = 1$ selects $2/134$. The so-called *position-vector* V always points to the currently active node.
- In order to **branch** a selected node, all elements of the active row, except the one pointed by the position-vector, are copied to the next row. For instance, to decompose $2/134$, the elements of row $I = 0$, except the scheduled job $M(0, V(0)) = 2$ are copied to the next row. Also, the integer $I \in [0, N[$ is incremented by one when a subproblem is decomposed.
- To **prune** a subproblem whose lower bound is greater than the best solution found so far, the corresponding cell should be ignored by the selection operator. For instance, to select the next node in row $I = 1$, the node $21/34$ is skipped by incrementing $V(1)$. To flag a cell as “pruned” its value is multiplied by -1 . With this convention the branch procedure actually consists in copying the absolute values to the next row, i.e. copying job $-j$ as j and j as j .

Each of the pool management operators can be expressed as an action on the IVM-structure. Before the bounding operator can compute the lower bounds of the generated subproblems, a **decode** operation is required. For example, the solution currently encoded in Figure 4b is 2314/, which can be directly read by looking (from row 0 to row $I = 3$) at the values that are pointed by the position-vector. With the same vector and matrix, if the integer is $I = 1$, the subproblem encoded by the IVM-structure is 23/14.

Using the IVM data structure, the **depth-first search** (DFS) strategy consists in selecting deepest leftmost non-negative cell in M . The depth-first select-and-branch procedure is described in Algorithm 1. First, a promising (i.e. non-pruned) node is searched in the current row I , right of the current cell $M(I, V(I))$, which is done by incrementing the position-vector (line 6). If no promising node is found in the current row, the search continues in the row above ($I - 1$), starting from the cell right of the previously selected $M(I - 1, V(I - 1))$ (line 4). When a promising node is found it is branched by generating the next line (line 8). The search stops without branching if the position-vector V has reached its maximum allowed value (*end-vector*) without finding a promising node. When using a linked-list for the implementation of the pool of subproblems the choice of DFS is motivated by the reduced memory requirements of DFS compared to breadth-first search (BFS) or other selection strategies. The IVM structure is conceived as an alternative data structure for DFS and it is not possible to perform a BFS using IVM.

In order to allow the scheduling of jobs at both ends of the partial permutations, an additional vector called *direction-vector* is used. This vector indicates for each row if the job pointed by the position-vector is to be placed in the beginning, or the end of the schedule. For instance, if the IVM-structure in Figure 4b is completed with the direction-vector (0110), then the jobs 3 and 1, pointed in the second and third row are scheduled at the end. The currently encoded node is then 24//13, which is a solution.

2.3. Position-vector: factoradic numbers

Throughout the exploration process, the position-vector behaves like a factoradic counter. In the example of Figure 4b, the value of this vector is equal to 0000 when the algorithm points to the first solution of the B&B tree, and its value is equal to 3210 when the algorithm points to the last solution of the tree. Between these two values, the vector successively takes the following values: 0010, 0100, 0110, 0200, 0210, ..., 3200. For each of these values, the algorithm points to a different solution of the tree. There are 24 possible values since there are 24 solutions (i.e. 4!). In reality, these 24 position-vector values correspond to the numbering of the 24 solutions using a special numbering system, called factorial number system. In the decimal number system, the weight of the i^{th} position is equal to 10^i , while in the factorial number system, the weight of the i^{th} position is equal to $i!$. In the decimal number system, the digits allowed at each position are 0 – 9, while in the factorial number system, the digits allowed for the i^{th} position are 0 – i . Therefore, the digit of the first position is always 0. The factorial number system, also called factoradic, is a mixed radix numeral system adapted to numbering permutations. It satisfies the conditions of what G. Cantor called a *simple* number system in [15]. Applied to the numbering of permutations, the French term *numération factorielle* was first used in 1888 [16]. Knuth [17] uses the term *factorial number system* and the term *factoradic*, which seems to be of more recent date is used, for instance, in [18].

2.4. Parallel IVM-based B&B

The properties of the position-vector allow us to say that “a B&B-process explores an interval $[A, B[$ using its IVM-structure”. In the example of Figure 4b, the interval explored by the algorithm is $[0000, 3210[$. It is therefore possible to have two processes $R1$, $R2$ such as $R1$ explores $[0000, X[$ and $R2$ explores $[X, 3210[$, each process using its private IVM-structure. Instead of sets of nodes, the work units of the IVM-based parallel B&B are intervals of factoradics. Because of the irregular and unpredictable shape of the explored tree, dynamic load balancing is necessary to maintain the degree of parallelism induced by the parallel tree exploration model. If $R2$ ends exploring its interval before $R1$, then $R2$ requests a portion of its interval from $R1$. Therefore, $R1$ and $R2$ can exchange their interval portions until the exploration of all $[0000, 3210[$. With the exception of rare works such as [13], work units

Algorithm 1 Serial select-and-branch

```

1: procedure SELECT-AND-BRANCH
2:   while (positionVector ≤ end) do
3:     if (row-end) then                                ▷ ( $V(I) > I$ )?
4:       cell-upward                                     ▷  $I \leftarrow I - 1; V(I) \leftarrow V(I) + 1$ 
5:     else if (cell-eliminate) then                     ▷  $M(I, V(I)) < 0$ ?
6:       cell-rightward                                  ▷  $V(I) \leftarrow V(I) + 1$ 
7:     else
8:       generate-next-line (branch)
9:       break
10:    end if
11:  end while
12: end procedure

```

exchanged between processes are sets of nodes.

To implement this strategy based on intervals of factoradics, it is necessary to allow a thread to explore any interval $[A, B]$. To begin the exploration at a given vector $V = (P_1 P_2 P_3 \dots P_N)$ ($= A$, expressed as a factoradic) the IVM-structure needs to be initialized accordingly. The correct initialized state is such that it would be the same if the new position-vector V had been reached through the exploration process. Therefore, the initialization process differs from the normal B&B process only in the selection operator. Instead of running the depth-first selection procedure (Algorithm 1), the initialization process selects at each level k the node pointed by $V(k)$ as long as the selected subproblem is promising. If a pruned node is selected, the initialization process is finished and the IVM resumes exploration, searching for the next node to decompose. If the first l positions of the newly received position-vector coincide with the position-vector of the victim IVM, then the victim's matrix and direction-vector for lines $1, 2, \dots, l$ can be copied to the thief. The thief IVM then starts initializing at line $l + 1$. The initialization process may thus last for 1 to N iterations.

3. GPU IVM-based Branch and Bound

This section describes our GPU-B&B algorithm based on the IVM data structure. The memory requirements of the IVM structure are very advantageous for a GPU-implementation of the B&B algorithm. The required amount of memory and possible data placements in the hierarchical device memory are discussed in Subsection 3.1. The amount of used memory depends on the number of IVM-structures used by the algorithm. This number also has a direct impact on the degree of parallelism which is analysed in Subsection 3.2. Both these subsections consider the framework for the GPU-based B&B algorithm. The algorithm itself is explained in Subsection 3.3. This subsection starts from a general illustration of the algorithm which is followed by a more detailed description of its components, which are 6 CUDA-kernels.

3.1. Memory requirements

Compared to a conventional linked-list-based approach, the IVM data structure allows to reduce the CPU time and memory required for the storage and management of the pool of subproblems [6]. Contrary to a linked-list, the IVM data structure is well adapted to the GPU memory model. Instead of using a variable length queue that requires dynamic memory allocations and tends to be scattered in memory, the IVM structures are constant in size and need only one allocation of contiguous memory. For a problem instance with N jobs, the storage of the matrix M requires N^2 bytes of memory (for $N < 127$, using 1-byte integers). Moreover $3N$ bytes are needed to store the position-, end- and direction-vectors, 1 byte to store the integer, and N bytes to store permutations before calling the bounding operator. In total, the IVM data structure requires a constant amount of $1 + 4N + N^2$ bytes of memory, i.e. 481 bytes per IVM for a 20-job instance. It is possible to store only the upper triangular part of M , requiring $1 + 4N + \frac{N(N+1)}{2}$ bytes per IVM, i.e. 291 bytes when $N = 20$. For $N = 20$ it is therefore possible to fit ≥ 100 IVM structures into 48 kB of shared memory. In this paper, only the upper triangular part of M is stored.

From a programming perspective the IVM-structures are easy to handle. The components of all IVMs are merged into single one-dimensional arrays. For instance, solving a N -job instance using T IVM structures, the matrices are stored in a one-dimensional array `matrices` of size $T \times \frac{N(N+1)}{2}$ allocated in global device memory. The element $M(i, j)$ of the k^{th} IVM is accessed by `matrices[indexM(i, j, k)]`, where `indexM` is a wrapper-function defined as in Equation (1) if M is stored as a square and as in Equation (2) if the upper triangular part of M is stored.

$$indexM(i, j, k) = k \times N \times N + i \times N + j \quad (1)$$

$$indexM(i, j, k) = k \times \frac{N(N+1)}{2} + i \times N - \frac{i(i-1)}{2} + j \quad (2)$$

The data needed for the computation of the lower bounds is mostly read-only and requires 34.5 kB of memory. This data is stored in the constant memory space, residing in global device memory but accessed through a cache on each streaming multiprocessor (SM). Some of the data structures used for the bounding may be loaded to shared memory during the computation of the lower bounds. Concerning the use of shared memory for those data structures, this paper follows the recommendations made in [19], where this difficult choice is examined.

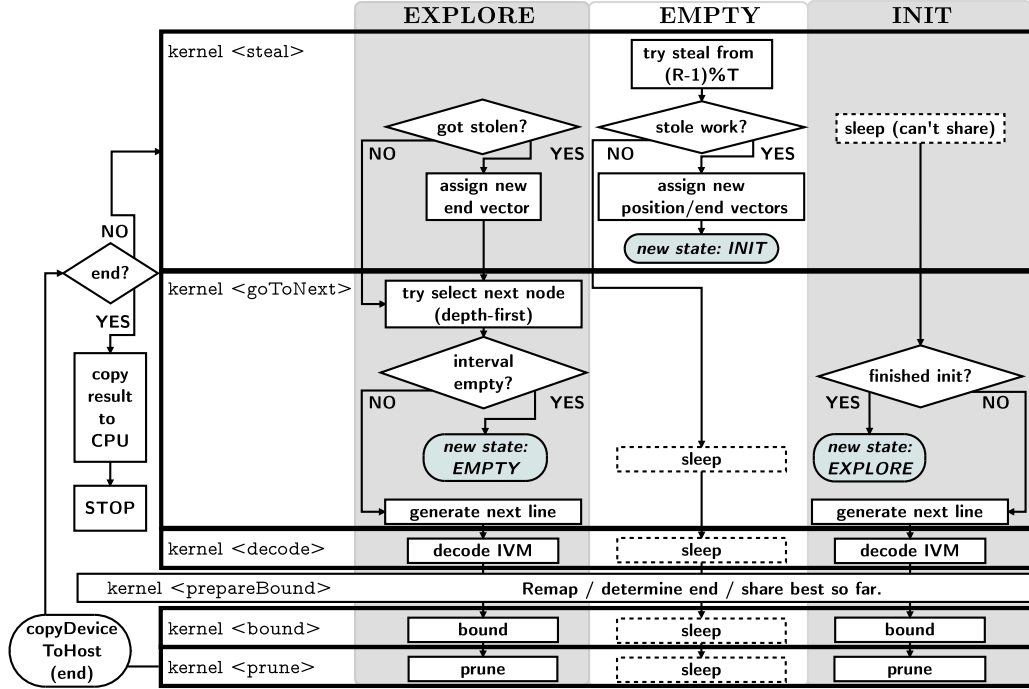


Figure 5: Flowchart of the GPU-based IVM-B&B algorithm

3.2. Degree of concurrency

Often, the bounding operator is by far the most time consuming part of a B&B algorithm. As mentioned before, in the case of flowshop it amounts for about 97 – 99% [4] of the total execution time for a sequential B&B. It is therefore crucial for the performance of our GPU-based B&B that the parallel bounding operation makes the best use of the GPU resources. The choice of the number of B&B processes (=IVMs) to use is therefore guided by its impact on the performance of the bounding kernel. On the one hand, if too few IVMs participate in the exploration process, the bounding kernel underutilizes the GPU. On the other hand, if too many IVMs are used, then the number of generated subproblems per iteration exceeds the maximum occupancy of the device and the computation of bounds is partially serialized. The number of subproblems generated per IVM per iteration is variable and unpredictable. However, the workload for the bounding kernel can be roughly estimated. For flowshop instances of 20 jobs, the bulk of subproblems is situated at depth 10, leading to approximately 20 bound evaluations per IVM and iteration. Supposing that the number of empty IVMs is low thanks to dynamic load balancing, and given the approximative number of 20,000 concurrent threads at full occupancy, the number T of used IVMs should be around $T = 1000$.

3.3. IVM-based GPU-B&B

In consistency with the CUDA programming model, the GPU-based parallel tree exploration is performed synchronously. The algorithm consists of different phases between which the B&B processes are synchronized. Although some implementations of global synchronization primitives are proposed in the literature [20], the global synchronization of an arbitrary number of thread blocks can only be achieved implicitly through kernel termination. Therefore the GPU-based B&B is implemented as a series of CUDA-kernels which are launched in a loop until the termination of the tree exploration. Figure 5 provides an overview of the algorithm. All B&B operators are entirely performed on the GPU and correspond to five kernels: `share`, `goToNext`, `decode`, `bound` and `prune`. Moreover an auxiliary kernel `prepareBound` is used to build the mapping for the bounding operation (explained in Section 4). In this phase the best found solution so far is determined by a min-reduce of the best solutions found by all IVMs. In the same reduce procedure the termination of the algorithm is detected by searching the maximum of a per-IVM state variable where the state *empty* is encoded as 0. In order to stop iterating through

the B&B loop this information needs to be copied to the host at each iteration. Throughout the exploration process this is the only data (1 byte) that is transferred between host and device memory.

3.3.1. Load-balancing: *kernel share*

Work stealing (WS) is well-adapted for irregular applications. Like threads of a multi-core application, the IVM structures must share their work units. In a multi-core environment, a thread that runs out of work becomes a *thief* that attempts to steal a portion of work from a *victim* thread which is selected according to a victim selection strategy. The same principle can be applied to the GPU-based B&B. The proposed load balancing strategy is conceptually different in the sense that an IVM-based B&B process does not necessarily correspond to any particular thread but only to a segment of data. Secondly, compared to multi-core WS strategies, the WS operations between IVMs are lock-free and performed synchronously. The *kernel share* implements the 1D-ring WS strategy presented in [5]. Algorithm 2 shows the pseudo-code of this procedure. Although designed for multi-core IVM-based B&B, the 1D-ring strategy suits the synchronous execution mode of the GPU. The T IVM structures are numbered $R = 0, 1, \dots, T - 1$ and are arranged as an oriented ring, i.e. such that IVM 0 is IVM $(T - 1)$'s successor. Each empty IVM R tries to steal work from its predecessor $(R - 1) \% T$. This operation can be performed in parallel, as the mapping of empty IVMs onto their respective victims is one-to-one. If the selected victim has a non-empty interval, then all but $1/T^{th}$ of its interval is stolen. The function `computeNewPos` (line 6) receives the victim's interval $[A, B[$ as input and returns a point $C = (1 - \frac{1}{T})A + \frac{1}{T}B$. The division of intervals can be performed directly on the factoradic numbers without explicitly converting them to decimals. The IVM which got stolen continues the exploration of the remaining interval $[A, C[$, while the stealing IVM needs to initialize its matrix at the new position-vector C before starting the exploration of $[C, B[$. Its state-variable is therefore set to *init* (line 10). Each IVM cycles through three distinct states, from *exploring* to *empty* to *initializing* and back to *exploring*. An IVM can be in one of these three states at any given stage of the algorithm. Depending on the state of an IVM, different actions are performed during an iteration. In this kernel one thread per IVM is required. More parallelism can hardly be exposed. However, more threads can eventually be used to assign vectors in one parallel operation (lines 7 – 9). In Subsection 4.4 it is explained how, based on the *kernel share*, this WS strategy can be extended.

Algorithm 2 *Kernel share*

```

1: procedure SHARE
2:   thld ← blockIdx.x * blockDim.x + threadIdx.x
3:   ivm ← map(thld)
4:   victim ← (ivm - 1) % T
5:   if (state[ivm] == empty .and. state[victim] == exploring) then
6:     new-pos ← computeNewPos( pos[victim], end[victim] )
7:     pos[ivm] ← new-pos
8:     end[ivm] ← end[victim]
9:     end[victim] ← new-pos
10:    state[ivm] ← init
11:   end if
12: end procedure

```

3.3.2. Selection and branching: *kernel goToNext*

The *goToNext* kernel corresponds to the selection and branching operators. Algorithm 3 shows the pseudo-code of this kernel. It performs the selection operator for both, exploring and initializing IVMs. It also updates the IVM-states if necessary. For each exploring IVM it performs the *select-and-branch* procedure described in Algorithm 1 (line 12). If an exploring IVM finds no promising node (line 13), then its state variable is set to *empty*. If the end of an IVM's initialization process is detected (line 5) it switches to *exploring*. It is possible that, within one iteration, an empty IVM receives an interval, finishes initializing and returns to the empty state. As explained in Subsection 2.4, the initialization process differs from the normal exploration process only in the selection operator. The initialization-selection consists in choosing the node pointed by the position-vector. Thus, only the *generate-next-line* branching procedure is performed. Each IVM is handled by a single thread as the operations that modify each IVM structure are essentially of sequential nature. This kernel contains a very high number of conditional instructions depending on the state of an IVM as well as on its current depth in the B&B tree. In order to avoid thread divergence the mapping of threads onto the IVM structures (line 3) must be chosen carefully. This mapping is discussed in Section 4.

Algorithm 3 *Kernel goToNext*

```

1: procedure GOToNEXT
2:   thld ← blockIdx.x * blockDim.x + threadIdx.x
3:   ivm ← map(thld)
4:   if (state[ivm] == init) then
5:     if (init-finished(ivm)) then
6:       state[ivm] ← exploring
7:     else
8:       generate-next-line           ▷ branch
9:     end if
10:  end if
11:  if (state[ivm] == exploring) then
12:    select-and-branch
13:    if (exploration-finished(ivm)) then
14:      state[ivm] ← empty
15:    end if
16:  end if
17: end procedure

```

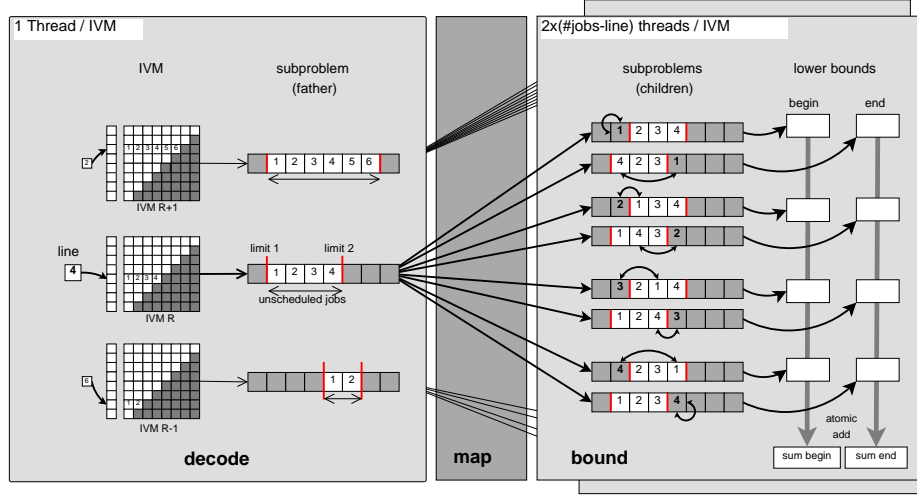


Figure 6: Illustration of the decode and bounding phases

3.3.3. Preparation of subproblems: kernel `decode`

For each non-empty IVM a father subproblem is selected in the `goToNext` kernel. The `decode` kernel brings these subproblems to the form $2/13/4$ which can be evaluated by the bounding operator. Each non-empty IVM data structure is read and the kernel outputs a N -integer sequence *schedule* with two integers *limit1* and *limit2*. The integers *limit1*, *limit2* represent the “/”’s in the adopted notation. This decoding operation is essentially sequential, so each IVM is handled by a single thread. Between IVM structures the number of jobs affected in the beginning or the end differs, which induces thread divergence if several threads within the same warp are assigned to different IVMs.

3.3.4. Parallel evaluation of lower bounds: kernel `bound`

The efficiency of the tree pruning mechanism directly depends on the relevance of the bounding operator. The lower bound proposed by Lageweg *et al.* [21] is used in our bounding operator. This bound is known for its good results and has complexity of $O(M^2 N \log(N))$, where N is the number of jobs and M the number of machines. This lower bound is mainly based on Johnson’s theorem [22] which provides a procedure for finding an optimal solution for flowshop scheduling problem with 2 machines. The computation of the lower bound includes several control flow instructions that depend on the depth of a subproblem and on the number of jobs placed at each end of the partial permutation that is evaluated. In [14] several techniques are proposed to reduce the thread divergence related to these control flow instructions. These optimizations are taken into account in the procedure `computeLB` which returns a lower bound (LB) value for a subproblem provided in the form $2/13/4$ (`schedule=2134`, `limit1=0`, `limit2=3`).

The granularity in this kernel is the computation of one bound. Each active thread in this kernel generates a distinct subproblem from the father node and computes its lower bound. These lower bound values are stored and atomically added to the values `sumBegin` and `sumEnd`, which are used to decide which decomposition is retained. For each father subproblem of depth I , the lower bounds for $2 \times (N - I) = 2 \times (\text{limit2} - \text{limit1} - 1)$ children are computed. The father-children relation and the bounding procedure are illustrated in Figure 6. The number of active threads in the bounding kernel is therefore given by

$$2 \times \text{todo} = 2 \times \sum_{ivm \neq \emptyset}^{\#IVM} (\#jobs - \text{line}[ivm]) \leq 2 \times \#IVM \times N.$$

At a given iteration, this quantity depends unpredictably on the number of non-empty IVMs and on their depth in the B&B tree. The maximum $2 \times \#IVM \times N$ occurs in the case where all IVMs have non-empty intervals at level 0. Each thread that computes a lower bound must be provided the following information: (1) on which IVM it is

working, (2) which unscheduled job it is scheduling and (3) on what end of the partial permutation to schedule. A static mapping of threads onto potentially generated children nodes (thus launching $2 \times \#IVM \times \#jobs$ threads at each invocation) is possible. As this mapping is critical for the performance of the bounding kernel, and thus for the entire algorithm, a remapping phase should precede the calling of the bounding kernel. Building such a mapping generates extra overhead which must be kept low. The mapping and implementation details of the bounding kernel are further discussed in Section 4.

3.3.5. *kernel prune*

In a first step the pruning kernel compares the values *sumBegin* and *sumEnd* for each IVM. Depending on this comparison it uses the set of lower bounds *costBegin* or *costEnd* to perform the pruning of nodes. The pruning itself consists in multiplying the corresponding cell in the matrix by -1 if the associated lower bound is greater than the best found solution so far. This kernel is the computationally less intensive one.

4. Mapping and thread divergence reduction

The shape of the tree explored by a B&B algorithm is highly irregular and unpredictable, resulting in an irregular workload, irregular control flow and irregular memory access pattern. If not addressed properly, these irregularities may cause a low occupancy of the device, serialized execution of instructions and poor bandwidth usage due to uncoalesced memory accesses. Both, the application’s memory access pattern and the divergent behaviour of threads depend strongly on the chosen mapping of threads onto the data. When a GPU application runs, each streaming multiprocessor (SM) is assigned one or more thread block(s) to execute. Those threads are partitioned into groups of 32 threads¹, called warps, which are scheduled for execution. CUDA’s single-instruction multiple-thread (SIMT) execution 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. In a very similar way, if the threads in a warp agree on the location of a requested piece of data, it may be fetched in single cycle, otherwise serialization of the data accesses occurs. In this paper the focus is put on reducing thread divergence and increasing warp execution efficiency by making judicious mapping choices. In Subsection 4.1 two different mapping strategies for the bounding kernel are presented. Subsection 4.2 discusses how to reduce the overhead induced by the building of the mapping. Finally, 4.3 presents alternative mappings for the IVM management kernels where the number of conditional instructions is very high.

4.1. Mapping the bounding operation

The most straightforward approach probably consists in mapping each thread onto a child subproblem directly from its *threadId*. This naive approach is shown in Algorithm 4. For instance, launching $2 \times N \times \#IVM$ threads (line 2), the first $N \times \#IVM$ threads place unscheduled jobs in the beginning, the second $N \times \#IVM$ threads in the end. Regardless of the IVM’s state or current depth in the tree, $2 \times N$ threads are reserved for each IVM. Each thread is assigned an IVM to work on and a job to schedule, like shown in lines 4-6 of Algorithm 4. The approach of Algorithm 4 has several disadvantages. The *if*-conditionals in line 8 and 9 mask many of the launched threads, precisely $2 \times k$ threads per father subproblem of depth k , plus $2N$ threads per empty IVM. Moreover, different lanes in the same warp work on different IVMs, thus thread divergence occurs due to different values of *limit1* and *limit2*. If $T \times N$ is a multiple of *warp-size*, then the *if-else* conditional (lines 10 and 14) does not cause any thread divergence.

The goal of the remapping procedure which prepares the bounding is to build two maps *ivm-map* and *job-map* which contain, for *todo* threads, the information which IVM to work on and which job to swap. Using an even/odd pattern these maps provide sufficient information for both groups of threads. After building these maps, the bounding kernel (as shown in Algorithm 5) is called with $2 \times todo$ threads, where:

¹We assume using the GK110 model

Algorithm 4 Kernel STATIC-MAP-BOUND**in:**fathers (father,limit1,limit2)**out:**lower bounds begin, lower bounds end, sums of lower bounds

```

1: procedure NAIVE-BOUND
2:   <<< 2 × #jobs × #IVM threads >>>
3:   thld ← blockIdx.x * blockDim.x + threadIdx.x
4:   if (state[ivm] == not-empty) then
5:     if (limit1[ivm] < job < limit2[ivm]) then
6:       if (dir == 0) then                                ▷ evaluate begin
7:         swap( schedule, limit1[ivm]+1, job )
8:         LB-begin[ivm][job] ← computeLB( schedule )
9:         sum-begin[ivm] += LB-begin[ivm][job]             ▷ atomic
10:      else if (dir == 1) then                             ▷ evaluate end
11:        swap( schedule, limit2[ivm]-1, job )
12:        LB-end[ivm][job] ← computeLB( schedule )
13:        sum-end[ivm] += LB-end[ivm][job]                 ▷ atomic
14:      end if
15:    end if
16:  end if
17: end procedure

```

Algorithm 5 Kernel REMAPPED-BOUNDING**in:**fathers (father,limit1,limit2), ivm-map, job-map**out:**lower bounds begin, lower bounds end, sums of lower bounds

```

1: procedure REMAPPED-BOUND
2:   <<< 2 × todo threads >>>
3:   thld ← blockIdx.x * blockDim.x + threadIdx.x
4:   dir ← thld mod 2
5:   ivm ← ivm-map[thld/2]
6:   job ← job-map[thld/2]
7:   schedule ← fathers[ivm]
8:   toSwap ← (1-dir)*(limit1[ivm]+1) + dir*(limit2[ivm]-1)
9:   swap( schedule, toSwap, job )
10:  LB[dir][ivm][job] ← computeLB( schedule )
11:  sum[dir][ivm] += LB[dir][ivm][job]                     ▷ atomic
12: end procedure

```

- threads 0 and 1 work on IVM *ivm-map[0]*, swapping job *job-map[0]* respectively to begin/end,
- threads 2 and 3 work on IVM *ivm-map[1]*, swapping job *job-map[1]* respectively to begin/end,
- ...
- threads $2 \times \text{todo} - 2$ and $2 \times \text{todo} - 1$ work on IVM *ivm-map[todo-1]*,...

The remapped bounding kernel is launched at each iteration with a kernel configuration of $(2 * \text{todo} / \text{blockDim}) + 1$ blocks (simplified in Algorithm 5) which is adapted to the workload. The proposed approach is known as *stream compaction* in the literature. It reduces the number of idle lanes per warp as well as the number of threads launched per kernel invocation. However, any thread divergence resulting from the begin-end distinction should also be avoided, as this involves a serialization of the costly `computeLB` procedure. To achieve this, the bodies of the *if-else* conditional (Alg. 4, lines 10 – 18) can be merged into a single one (Alg. 5, lines 8 – 11). Two different arguments of the same type, occurring on the right-hand side of a statement can often be refactored into a single one, like in Algorithm 5, line 8. The different arrays on the left-hand side are merged into larger ones. This allows to merge the statements of lines 12,13 and 16,17 of Algorithm 4 into single statements (Alg. 5, lines 10,11). The separation of data within these merged arrays is assured by indexing with the variable `dir`, which evaluates differently for even/odd threads.

4.2. Efficient building of the remapping

Algorithm 6 describes how to build the maps *ivm-map* and *job-map* sequentially. However, sequential execution of this procedure on the device has prohibitive cost, exceeding 25% of the total execution time. The remapping should therefore be built in parallel. The parallelization of the outer *for*-loop (Alg. 6, line 3) is not straightforward, because it is unknown at which location the data for each IVM is to be written to. Computing the *prefix-sum* of a vector containing the number of jobs to be scheduled per IVM allows its parallelization.

The operation *prefix-sum* is defined as

$$\text{prefix-sum} : [a_0 \ a_1 \ a_2 \ \dots \ a_n] \mapsto [0 \ a_0 \ (a_0 + a_1) \ (a_0 + a_1 + a_2) \ \dots \ \sum_{i=0}^{n-1} a_i].$$

Efficient parallel CUDA-implementations for this operation have been proposed in the literature [23]. It is also available in the CUDA Thrust library. However, for relatively small vectors it may be preferable to reimplement the operation, in order to avoid casting the input data to a `thrust::device_ptr`.

A first building step consists in filling an array `todo-per-IVM` with $\text{limit2} - \text{limit1} - 1$ for each IVM. The element R of `prefix-sum(todo-per-IVM)` indicates at which position of *ivm-map* and *job-map* the data of an IVM R starts to be written. The complete parallelized building of the mapping is shown in Algorithm 7. The

Algorithm 6 Build mapping (serial)

```
1: procedure SERIAL PREPARE BOUND
2:   running-index  $\leftarrow$  0
3:   for (ivm = 0  $\rightarrow$  T) do
4:     if (state[ivm] = not-empty) then
5:       for (job = limit1[ivm] + 1  $\rightarrow$  limit2[ivm]) do
6:         ivm-map[running-index]  $\leftarrow$  ivm
7:         job-map[running-index]  $\leftarrow$  job
8:         running-index++
9:       end for
10:    end if
11:  end for
12:  todo  $\leftarrow$  running-index
13: end procedure
```

Algorithm 7 Build mapping (parallel)

```
1: for all (non-empty ivm) do
2:   todo-per[ivm]  $\leftarrow$  (limit2[ivm]-limit1[ivm]-1)  $\triangleright$  else  $\leftarrow$  0
3: end for
4: Aux  $\leftarrow$  parallel-prefix-sum(todo-per)
5: prepare-bound<<< #IVM  $\times$  #JOBS >>>
6: procedure [KERNEL] PREPARE-BOUND
7:   thld  $\leftarrow$  blockIdx.x*blockDim.x + threadIdx.x
8:   ivm  $\leftarrow$  thld / N
9:   thPos  $\leftarrow$  thld % N
10:  if (thPos < todo-per[ivm]) then
11:    ivm-map[Aux[ivm]+thPos]  $\leftarrow$  ivm
12:    job-map[Aux[ivm]+thPos]  $\leftarrow$  limit1[ivm]+1+thPos
13:  end if
14:  todo  $\leftarrow$  Aux[#IVM]+todo-per[#IVM]
15: end procedure
```

Algorithm 8 Mapping 1

```
1: kernel<<<#IVM threads>>>
2: ivm  $\leftarrow$  blockIdx.x*blockDim.x + threadIdx.x
3: do-something-with[ivm]
```

Algorithm 9 Mapping 2

```
1: kernel<<<warpsize  $\times$  #IVM threads>>>
2: thld  $\leftarrow$  blockIdx.x*blockDim.x + threadIdx.x
3: ivm  $\leftarrow$  thld/32
4: thPos  $\leftarrow$  thld%32
5: if (thPos == 0) then
6:   do-something-with[ivm]
7: end if
```

building of the mapping ranges over several kernels. The filling of `todo-per-IVM` can be done, for instance, in the decode-kernel.

4.3. Mapping choices for IVM management kernels

The IVM-management kernels `goToNext`, `decode` and `prune` require a single thread per IVM. The naive approach consists in launching T threads and mapping thread k on IVM k , for $k = 0, 1, \dots, T - 1$ (see Algorithm 8). Given the high number of conditional instructions in the IVM-management kernels it is very unlikely that all 32 threads in a warp follow the same execution path if this mapping is used. Indeed, in these kernels control flow divergence results from different IVM-states, different numbers of scheduled jobs at both ends of the active subproblem and from the search for the next node which requires an unknown number of iterations.

An alternative mapping, shown in Algorithm 9, can solve this issue. An entire warp is assigned to each IVM, so all threads belonging to the same warp follow the same execution path. This strategy goes in the opposite direction of the stream compaction approach proposed for the bounding kernel. As only one thread per IVM is needed, all lanes in a warp except this first are masked. Thus, the kernels are launched with $32 \times$ as many threads as necessary (i.e. $32 \times T$). Using this mapping, the overhead induced by thread divergence completely disappears (although technically, the disabled threads are diverging at line 5 of Algorithm 9). The drawback is obviously the launching of $31T$ idle threads. However, in Subsection 3.2 we argued that T should be chosen around $T = 1000$, which is small compared to $\text{\#SM} \times (\text{max. threads per SM})$. This, and the fact that the control flow irregularity is very high, justifies the approach of using 1 warp per IVM. Moreover, using only 4-8 IVM-structures per block allows to store them into shared memory without limiting the theoretical occupancy of the device. The loading of data from global to shared memory can be done very efficiently, using the additional threads which are not used for computation.

4.4. Work stealing strategies

The topology used in the WS strategy described in Subsection 3.3.1 is a unidirectional 1-dimensional ring (1D-ring). The maximal distance between two IVMs in the 1D-ring is T . Work units propagate through the ring as they are passed downstream from exploring to empty IVMs. As most of the explored B&B nodes are actually contained in a relatively small interval, the workload tends to be concentrated in some part of the ring. Thus, workers situated far away from the source are only kept busy if the overall workload is large enough. With an increasing number T of IVM structures it becomes more likely that no work is dripping down to some of the workers. A topology that

0	1	2	...	(C-1)
C
2C
...
(R-1)C	RxC -1 =T-1

Figure 7: Illustration of 2D-ring topology for T IVMs using R rings of ring-size C .

reduces the maximum distance between two workers should therefore improve the scaling with T .

The 1D-ring can be easily generalized to a 2D-ring, or torus, topology. Instead of using a single ring, IVMs are arranged in R rings of ring-size $C = T/R$. In a first step each empty IVM attempts to steal from its left neighbour within the same ring. A second step connects the rings between each other: each empty IVM selects the IVM with the corresponding ID in the preceding ring (with ring $R - 1$ being connected to ring 0). The roles played by both directions are symmetric. Ideally, the number R is therefore such that $R = \sqrt{T}$, which is only possible if T is square. In that case the 2D-ring reduces the maximum distance between two IVMs to $2\sqrt{T}$. If $C \neq R$, then the diameter of the 2D-ring is $(C + R)$.

The 2D-ring topology is implemented by two subsequent calls of kernel `share` (Algorithm 2), where only line 4 of the algorithm needs to be modified. In particular, line 4 of Algorithm 2 is replaced by the following.

$$\begin{aligned} \text{In Step 1 IVM } i \text{ selects } victim(i) &= \begin{cases} i-1, & \text{if } i \bmod C \neq 0 \\ i+(C-1), & \text{otherwise} \end{cases} \\ \text{In Step 2 IVM } i \text{ selects } victim(i) &= \begin{cases} i-C, & \text{if } i > (C-1) \\ (R-1)C+i, & \text{otherwise} \end{cases} \end{aligned}$$

Figure 7 illustrates the 2D-ring topology in the form of a 2D-grid. A torus, used in the 2D-ring WS strategy is obtained by connecting the upper with the lower and the leftmost with the rightmost cells. Similarly, the topology can be extended to a hypercube, which is used for instance in [24] for unbalanced tree search.

5. Experiments

In this section the performance of the IVM-based GPU-B&B is analysed for different mapping choices, a varying number of IVM structures and different work stealing strategies. Subsection 5.1 explains the flowshop scheduling problem, the problem instances used for benchmarking and the hardware test-bed. In Subsection 5.3 the mapping strategies for the bounding kernel are evaluated and Subsection 5.4 compares the different mapping strategies for the pool management kernels. The algorithm's scalability and load balancing issues are examined in Subsection 5.5. Finally, our IVM-based GPU-B&B algorithm is compared to the GPU-accelerated linked-list based algorithm presented in [1].

5.1. Flowshop scheduling problem

Flowshop belongs to the category of scheduling problems. A scheduling problem is defined by a set of jobs and resources. Flowshop is a multi-operation problem, where each operation is the execution of a job on a machine. In this problem, the resources are machines in a production workshop. The machines are arranged in a certain order. As illustrated in the example of Figure 8, the machines process jobs according to the chain production principle. Thus, a machine can start processing only those jobs which have completed processing on all the machines which are located upstream. A duration is associated with each operation. This duration is the time required for a machine

M1	J2	J4	J5	J1	J6	J3						
M2		J2	J4	J5	J1		J6	J3				
M3			J2	J4	J5	J1		J6	J3			

Figure 8: Example of a solution of a flowshop problem instance defined by 6 jobs and 3 machines.

to finish the processing of a job. An operation can not be interrupted, and machines are critical resources, because a machine processes one job at a time. The *makespan* of a solution corresponds to the time when the last job ends on the last machine. The objective is to find a solution that minimizes the *makespan*. In [25], it is shown that the minimization of *makespan* is NP-hard from 3 machines upwards.

In our experiments, the flowshop instances defined by Taillard [26] are used to validate our approach. These instances are divided into 12 groups: 20x5 (i.e. group of instances defined by 20 jobs and 5 machines), 20x10, 20x20, 50x5, 50x10, 50x20, 100x5, 100x10, 100x20, 200x10, 200x20, and 500x20. In each group, 10 different instances are generated. For each instance, the duration of each job on each machine is randomly generated by [26]. These standard instances are often used in the literature to evaluate the performance of methods that minimize the makespan. The instances of the 6 groups where the number of machines is equal to 5 or 10 (i.e. 20x5, 20x10, 50x5, 50x10, 100x5, 100x10, and 200x10) are easy to solve. For these instances, the used bounding operator gives such good lower bounds that it is possible to solve them in few seconds using a sequential B&B. Instances where the number of jobs is equal to 50, 100, 200, or 500, and the number of machines is equal to 20 (i.e. 50x20, 100x20, 200x20, and 500x20) are very hard to solve. For example, the resolution of *Ta056* in [13], which is one of the 10 instances defined with 50 jobs and 20 machines (i.e. the 50x20 group), lasted 25 days with an average of 328 processors and a cumulative computation time of about 22 years. Therefore, in our experiments, the validation is performed using the 10 instances where the number of machines and the number of jobs are equal to 20 which belong to the group 20x20.

When an instance is solved twice using a B&B performing a parallel tree exploration, the number of explored subproblems is often different between the two resolutions, because the order of exploration varies. To compare the performance of two B&B algorithms, the number of explored subproblems should be exactly the same between the different tests. Therefore, we choose to always initialize our B&B by the optimal solution of the instance to be solved. This initialization ensures that the tree-shape does not depend on the decrease of the best solution found so far and that the number of explored subproblems is the same between the two resolutions. Table 1 shows the number of decomposed nodes during the resolution of instances *Ta021-Ta030* initialized with the optimal solution. This number represents the total amount of work to be done and ranges from 1.6 for the smallest to 140.8 million nodes for the largest instance.

Table 1: Number of decomposed subproblems during the resolution of Taillard’s instances *Ta021-Ta030* initialized with the optimal cost (in millions of nodes).

Instance	21	22	23	24	25	26	27	28	29	30	Average
#Nodes (in millions)	41.4	22.1	140.8	40.1	41.4	71.4	57.1	8.1	6.8	1.6	43.1

5.2. Hardware/Experimental protocol

All the experiments are run on a computer equipped with a NVIDIA Tesla K20m GPU based on the GK110 architecture. The device is composed of 2496 CUDA cores (clock speed 705MHz). Its maximum power consumption is 225W. Version 6.5.14 of the CUDA Toolkit is used. The CPU is a 8-core Sandy Bridge E5-2650 processor. The operation system installed is a CentOS 6.5 Linux distribution. For the evaluation of the elapsed execution time the UNIX `time` command is used. The duration of each CUDA-kernel and profiling of the kernels is done with the `nvprof` command line profiler. In order to reduce the profiling time, sample data was collected every 100 iterations of the algorithm. The chosen size for the threadblocks is 128. The configurable size of the device’s shared memory/L1 cache is set to 48/16kB for kernels except `bound`, where the opposite configuration 16/48kB is used.

For the comparison of the mapping strategies the number of used IVM-structures is set to $T = 768$, according to preliminary experiments. The best mapping found in Subsections 5.3 and 5.4 is used to determine an optimal value for T and the better work stealing strategy in Subsection 5.5.

5.3. Evaluation of mappings for bounding kernel

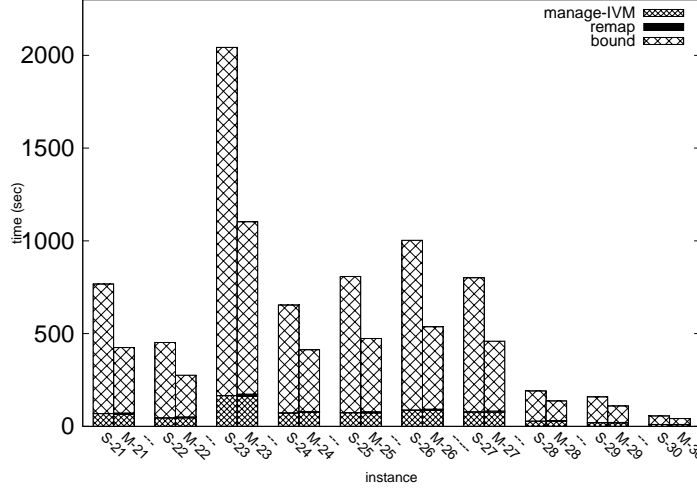


Figure 9: Execution time for instances *Ta021-Ta030* for thread-data mappings *static* (S) and *remap* (M) for the kernel *bound*.

In this subsection the two mapping schemes for the bounding kernel, presented in Subsection 4.1, are compared to each other in terms of elapsed execution time of the algorithm. The first, using the remapping shown in Algorithm 5 is referred to as *remap*, the second, using the static mapping of Algorithm 4, as *static*. Figure 9 shows the total elapsed time for solving instances *Ta021-Ta030*. For both mappings and for each instance it shows the portion of time spent in the kernel *bound*, in the IVM-management kernels (*share*, *goToNext*, *decode* and *prune*) as well as in the remapping phase (for *remap*). However, as the building of the mapping consumes only 0.9% of computation time, the latter portion is barely visible in Figure 9. Table 2 shows total elapsed time as well as the time spent in the different phases of the algorithm as an average over the 10 instances *Ta021-Ta030*.

The compacted mapping *remap* is clearly advantageous as it reduces the average time spent in the *bound* kernel by a factor 1.9. As the bounding operation amounts for more than 80% of the total execution time, the latter decreases by a factor 1.7. The overhead induced by compacting the mapping at each iteration is largely compensated by these performance gains. Indeed, thanks to the parallelization of this phase using the parallel prefix sum, the remapping operation amounts for less than 1% of the elapsed time. For comparison, using the CPU for the remapping, it amounts for about 7% of the algorithm’s total execution time, mainly because of the transfer of the maps back to the device.

Using the more compact mapping *remap* instead of *static* improves the control flow efficiency² (CFE) of the kernel. For *static* the average CFE is 0.43, meaning that for an executed instruction on average more than half

Table 2: Average elapsed time (in seconds) and average repartition of execution time among bounding, IVM management and remapping phases. Average taken over instances *Ta021-Ta030*.

Mapping	elapsed walltime	bound		manage		remap	
	sec	sec	%	sec	%	sec	%
<i>static</i>	696.4	632.9	89.4	63.5	10.6	0.0	0.0
<i>remap</i>	395.7	329.1	82.0	63.4	17.1	3.4	0.9

²defined as $CFE = \frac{\text{not_predicated_off_thread_inst_executed}}{32 * \text{inst_executed}}$

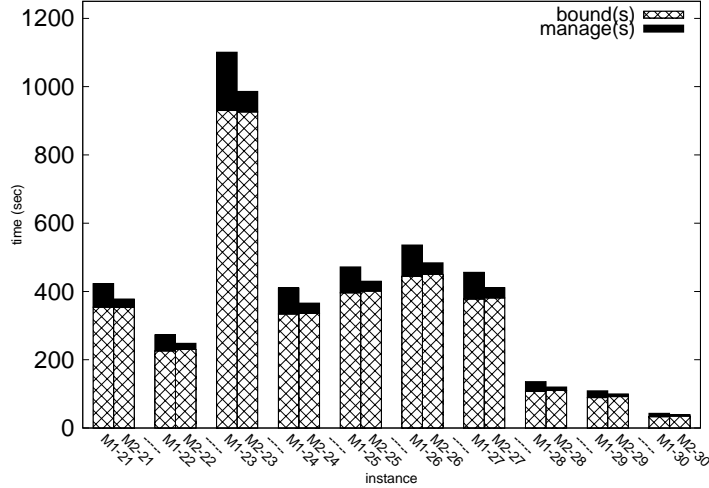


Figure 10: Execution time for instances *Ta021-Ta030* for different mapping choices in IVM-management kernels.

Table 3: Duration of different kernels per call (in μsec or msec), percentage of total elapsed time (%) and instruction replay overhead (IRO%), total execution time of GPU-B&B. Average values for instances *Ta021-Ta030*.

Mapping	goToNext			decode			bound		elapsed sec
	μsec	%	IRO%	μsec	%	IRO%	msec	%	
<i>1 thread/IVM</i>	380	10.0	40.6	168	4.4	40.3	3.07	82.0	395.7
<i>1 warp/IVM</i>	130	4.0	14.0	94	2.8	14.7	3.07	91.1	364.2
<i>1 warp/IVM (shared)</i>	85	2.6	7.9	79	2.4	12.4	3.06	92.5	356.6

of the execution slots are wasted. For the mapping *remap* the average CFE is 0.83 - the launched warps are used almost twice as efficiently. The number of warps launched at each kernel call is 960 for mapping *static*, which exceeds theoretical maximum of $13 \times 64 = 832$ resident warps for the K20m. The average number of warps launched with mapping *remap* is 300 (average per kernel call and per instance), the average maximum (per instance) being 825 warps and the minimum 4. These results show that it is a high priority optimization to adapt the configuration of the algorithm’s most cost intensive part to the varying workload.

5.4. Mapping for IVM management

In this subsection the two mapping schemes presented in Subsection 4.3 are evaluated and compared to each other. The kernels concerned by these mapping schemes are the IVM-management kernels (*share*, *goToNext*, *decode* and *prune*). Figure 10 shows the time spent for completing the exploration with both mapping schemes. Both, version *one-thread-per-IVM* (*M1*) and version *one-warp-per-IVM* (*M2*) use the same bounding kernel (with remapping). Although the time spent managing the IVM structures is moderate compared to the bounding operation, the mapping *M2* allows a reduction of the total execution time by a factor 1.1 compared to the mapping *M1*. With respect to *M1*, mapping *M2* decreases the share of IVM-management operations from 18% to 7.5%. Table 3 shows the average duration per call of the kernels *bound* (in msec), *goToNext* and *decode* (in μsec) and their respective share of the elapsed time (in %). The kernels *prune* and *share* amount for at less than 2% of total execution time, so they are not evaluated.

The mapping *M2* allows to use the supplementary lanes for efficient loading of the IVM structures into shared memory. In order to dissociate the impact of shared memory usage from the impact of remapping, the profiling of mapping *M2* is performed with and without shared memory usage.

Table 3 also shows the instruction replay overhead (IRO%)³, which is a measure for instruction serialization (due

³defined as $\text{IRO}\% = 100\% \times \frac{\text{instructions_issued} - \text{instructions_executed}}{\text{instructions_issued}}$

Table 4: Per-call average of branch instructions executed and diverging branches (incremented by one per branch evaluated differently across a warp). Instance *Ta022*.

kernel mapping	goToNext		decode		share		prune	
	branch	diverge	branch	diverge	branch	diverge	branch	diverge
<i>1 thread/IVM (M1)</i>	11592	802	5875	860	851	15	404	121
<i>1 warp/IVM (M2)</i>	59921	1536	62020	768	3655	0	3131	768
	=2×#IVM		=#IVM				=#IVM	

to memory operations only). These results show that the fact of spacing the mapping to 1 warp=1 IVM also substantially improves the memory access pattern. It should be noted that the metric *control flow efficiency*, used in Subsection 5.3 drops from a poor average 0.22 for *M1* to $0.03 \approx 1/32$ for *M2* - as intended. Table 4 shows, for the different kernels, the number of branch instructions executed (per call average) and the number of branches that are evaluated differently across a warp. The results show that, as intended, undesired thread divergence completely disappears. Only instance *Ta022* is evaluated as one instance sufficiently illustrates the behaviour.

The `divergent_branch` counter indicates that the average number of diverging branches is a multiple of the number of IVMs. Indeed, the counter increments by one at the instruction `if (thId%32 == 0)` (Algorithm 9, line 5) which masks all but the leading thread in each warp. However, as the remaining 31 lanes of the warp are simply waiting for lane 0 to complete, no significant serialization of instructions occurs. Besides showing that the spaced mapping *M2* is better adapted to the IVM-management kernels, the results presented in this subsection illustrate that performance metrics for thread divergence or control flow must be interpreted very carefully.

5.5. Scalability and stability analysis

Table 5: Groups of similar-sized flowshop instances (20 jobs × 20 machines) and the corresponding average number of nodes decomposed when initialized with the optimal solution.

Instance group	#decomposed nodes	Instances	Average #decomposed nodes
<i>small</i>	< 10M	<i>Ta028, Ta029, Ta030</i>	5.5M
<i>medium</i>	∈ [10M, 50M]	<i>Ta021, Ta022, Ta024, Ta025</i>	36.3M
<i>large</i>	∈ [50M, 100M]	<i>Ta026, Ta027</i>	64.3M
<i>huge</i>	> 100M	<i>Ta023</i>	140.8M

In this subsection the behavior of the algorithm according to the instance sizes and its scalability with the number of used IVM structures (T) is examined. The algorithm’s performance for problem instances of different sizes is compared in terms of node processing speed (#decomposed nodes/second), which is computed from wall-clock time. Obviously, using more explorers can only be beneficial if they can be supplied with enough work. Therefore, the relationship between instance-size, the node processing speed and T needs to be studied for both proposed WS strategies, as they strongly impact this relationship. For the experimental study of scalability only the best version of the previous subsection is considered, i.e. the one using parallel remapping for the bounding kernel and the spaced mapping *M2* for the management kernels. Another factor that has a significant impact on the algorithm’s performance is the irregularity of the explored B&B tree, which is very hard to quantify. In order to obtain a clearer dissociation between tree-size and tree-irregularity, the average node processing speed for instances of similar size is considered. The instances have been grouped as shown in Table 5.

Figure 11 shows the average node processing speed for these four groups of instances, according to different values of T and WS strategies 1D-Ring and 2D-Ring. T is chosen as a multiple of 64, as the GK110 architecture allows up to 64 warps per SM and the management kernels reserve one warp per IVM. The number of rings R in the 2D-Ring WS strategy is chosen such that R divides T while being as close to \sqrt{T} as possible – the goal being to approach the ideal configuration where R equals the ring-size C .

From Figure 11 one can see that the node processing speed for *small* instances is lower than for intermediate-sized instances, regardless of parameter T or the used WS strategy. This is partially due to warm-up and shut-down phases of the parallel exploration, which, for *small* instances last relatively long with respect to the total exploration time. In these phases a low overall workload limits the degree of parallelism and the WS mechanism must handle sharp variations of the workload. Using the 2D-Ring topology significantly improves the nodes-per-second rate

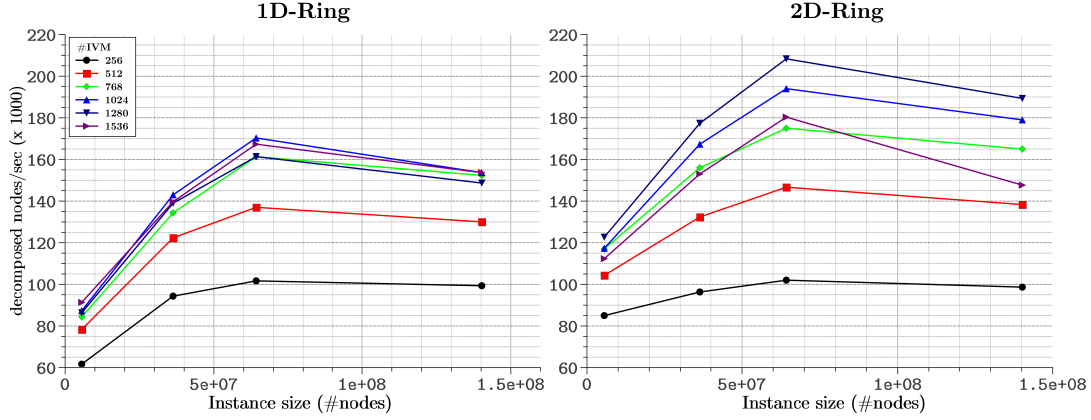


Figure 11: Average node processing speed for groups of instances *small*, *medium*, *large*, *huge* (Table 5), and $T = k \cdot 256$ ($k = 1, 2, \dots, 6$).
Left: 1D-Ring work stealing ; Right: 2D-Ring work stealing with dimensions $R \times C = 16 \times 16, 16 \times 32, 24 \times 32, 32 \times 32, 32 \times 40$ and 32×48

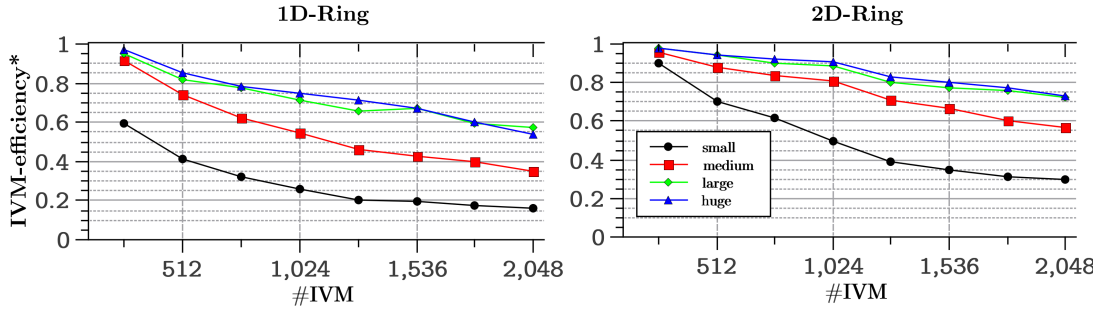


Figure 12: $\text{IVM-efficiency}^* = \frac{\text{decomposed nodes}}{\text{iterations} \times \text{\#IVM}}$, measures the overall share of IVM structures in the *exploring* state. An IVM-efficiency of 1 indicates that each IVM decomposes a node at each iteration.

for *small* instances, as the reduced diameter of the 2D-Ring accelerates the distribution of the workload. Another reason for the poorer performance on *small* is that the finer granularity of allocated tasks per IVM does not allow one to hide the initialization costs as efficiently as in larger instances. Similarly, the nodes-per-second performance for *medium* instances is below the performance for *large* and *huge* instances. As the groups *large* and *huge* do only count one, respectively two members, the results for these groups must be interpreted more carefully – however, they suggest that the node processing speed for large or very large instances is less influenced by the tree’s size. The size of the instance also impacts the scalability of the algorithm with T , especially when the 2D-ring WS strategy is used. Using the 2D-Ring topology, for *small*-sized problems the best value for T improves performance to $\approx 120\text{k}$ nodes/sec from $\approx 85\text{k}$ nodes/sec for the worst T , while for *large*-sized problems the node processing speed doubles from ≈ 105 to $\approx 210\text{k}$ nodes/sec, doing the same comparison.

Comparing both WS strategies, one can see from Figure 11 that the 2D-ring topology improves the scalability of the algorithm. Indeed, for the 1D-Ring almost no performance is gained above $T = 768$ because additional workers are left idle or inefficiently initializing. In contrast, the 2D-Ring strategy allows one to use up to $T = 1280$ IVMs efficiently. For $T = 1536$, and only when using the 2D-Ring WS strategy, performance drops significantly for all instance sizes. The most likely explanation for this performance drop is that the computation of the bounds is partially serialized due to hardware limitations. On average, each node decomposition leads to the evaluation of approximately 20 bounds, for all considered instance-sizes. So, if all $T = 1280$ IVMs are busy, 25600 bounds are evaluated per average iteration, which is close to the hardware limit of 26624 concurrent threads ($13 \text{ SM} \times 64 \text{ warps/SM} \times 32 \text{ threads/warp}$) for the GK110 architecture. Therefore, supposing a well balanced workload, no performance improvement can be expected from increasing T beyond 1280 on this device and at constant tree-size

However, Figure 12 shows that there is some margin left to increase the percentage of busy explorers. In

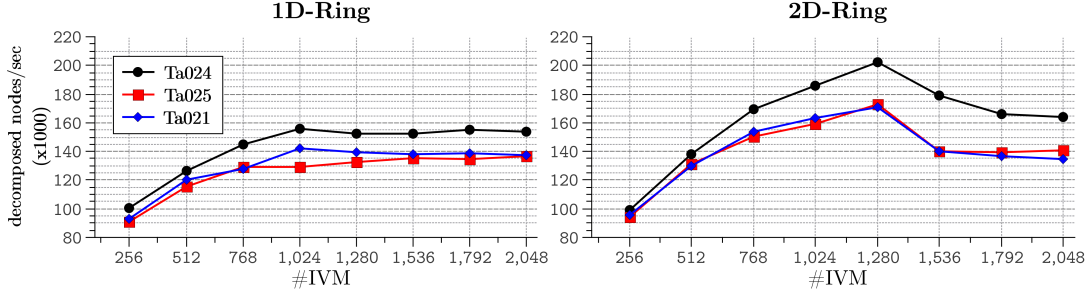


Figure 13: Scaling with T of the medium-sized instances

order to measure the overall efficiency of the WS strategies, the number of decomposed nodes is divided by $(\#iterations \times \#IVM)$, which gives a measure for the portion of IVMs doing meaningful work. If this *IVM-efficiency* equals 1, then each IVM decomposes a node at each iteration – if it equals 0, then all IVMs remain idle for an infinity of iterations. An *IVM-efficiency* of 0.5 indicates that an average IVM spends half of its iterations idle or initializing. The results shown in Figure 12 confirm that the 2D-Ring WS allows to keep more explorers busy. Moreover, the fact that at $T = 1536$ no sharp degradation of the IVM-efficiency occurs, confirms that the performance drop at $T = 1536$ is due to hardware limitations. One can also see in Figure 12 that for *small* and *medium* instances the IVM-efficiency is lower than for larger instances. This explains the relatively poorer performances for these instances. The smaller the instance being solved, the likelier it becomes that a given IVM frequently steals intervals that are explored within a few or zero iterations. In that case, the cost of initialization is not covered by meaningful work. Without any *a priori* knowledge concerning the amount of work contained in a given interval, it seems difficult to resolve this problem.

Concerning the algorithm’s sensitivity towards the irregularity of the tree structure, the performances obtained when solving same-sized instances are compared in Figure 13. Figure 13 zooms on the scalability with T of the three instances with $\approx 40M$ nodes. For all three instances the performance drop increasing the number of IVMs from $T = 1280$ to $T = 1536$ is clearly visible. Figure 13 also shows that the instance-irregularity impacts performance significantly. Exploring a roughly equal number of nodes, the peak node processing speed attained for instance *Ta024* is 202k nodes/sec while it remains below 175k nodes/sec for instances *Ta021* and *Ta025*. However, in sequential execution *Ta024* has the best nodes-per-second rate of these three instances, which means that the performance gap is not due to a higher number of nodes in the upper part of the tree (which are more costly to evaluate). The performance variety under same-sized instances rather seems to be due to sharper workload variations for instances *Ta021* and *Ta025*. Although for the three instances the proposed 2D-Ring strategy scales nicely up to 1280 IVM structures, it apparently has a moment of inertia which makes the algorithm sensitive to rapid workload variations. Maybe this gap can be closed with a further improved WS strategy, in the sense that the WS strategy should deliver a faster response to those variations. This is a difficult task, in particular because IVM need to go through initialization after the reception of a new work unit.

5.6. Comparison with GPU-accelerated linked-list-based B&B

In this subsection the best version (according to the previous experimentations) of the IVM-based GPU-B&B is compared to the GPU-accelerated B&B presented in [1] which uses a conventional linked-list for the storage and management of the pool of subproblems. The linked-list-based GPU-B&B (GPU-LL) algorithm is described in Subsection 1.3. Like the IVM-based GPU-B&B, GPU-LL performs a depth-first search, retaining after each branching the better of two generated pools (as described in Subsection 1.1). Moreover, both algorithms use the same device function to compute the lower bounds for a given subproblem. Table 6 shows the elapsed time for solving each of the ten 20-job instances, as well as the number of decomposed nodes per second, in order to take into account the size of the instances. The average time spent by the IVM-based GPU-B&B for exploring the ten instances is 229.7 seconds, while the GPU-LL algorithm requires on average 865.6 seconds for the same tasks. In terms of node processing speed, the GPU-IVM algorithm decomposes on average 3.3 times more nodes per second than its linked-list counterpart. The IVM-based GPU-B&B outperforms GPU-LL by at least a factor

Table 6: Elapsed execution time and number of decomposed nodes per second (in 1000 nodes/sec) for the IVM-based GPU-B&B (GPU-IVM) and the GPU-accelerated linked-list-based B&B [1] (GPU-LL), described in Subsection 1.3. The GPU-IVM algorithm uses the 2D-ring work stealing strategy and $T = 1280$ IVMs for all instances. Results for instances *Ta021-Ta030*.

Instance	GPU-LL		GPU-IVM		ratio
	elapsed	k nodes/sec	elapsed	k nodes/sec	
21	833	49.7	242	171.1	3.4
22	415	53.3	134	163.6	3.1
23	3089	45.6	740	189.6	4.2
24	738	54.3	200	202.2	3.7
25	865	47.9	239	173.1	3.6
26	1292	55.3	348	205.6	3.7
27	1094	52.2	268	213.3	4.1
28	171	47.4	53	152.0	3.2
29	125	54.4	56	121.4	2.2
30	34	47.1	17	94.6	2.0
Average	865.6	50.7	229.7	168.6	3.3

2 for all ten instances. The highest nodes-per-second performance is attained for instance *Ta027*, decomposing 213,000 nodes/sec, which is almost 4 times more than the highest rate attained by the linked-list-based algorithm. As examined in Subsection 5.5, for the GPU-IVM algorithm the node processing speed varies from one instance to another, depending on the instance’s size and irregularity. In contrast, as the GPU-LL algorithm regularizes the workload by dynamically adapting the size of the offloaded pools it provides an almost constant node processing speed for all instances. This performance variety is discussed in Subsection 5.5 – while further improved WS strategies should allow to reach a higher node processing speed for small instances, the algorithm needs to be extended to a multi-GPU version in order to improve the performance for larger instances.

6. Conclusions and future work

Our paper proposes a GPU-based branch-and-bound (B&B) parallel algorithm which performs all B&B operators on the GPU. During the exploration of the B&B-tree the CPU is only used for launching the CUDA-kernels in a loop until a boolean variable, which the CPU receives at each iteration from the GPU, indicates the end of the algorithm. To the best of our knowledge, our GPU-based B&B algorithm is the first one that does not rely on the transfer of pools of subproblems between host and device. The proposed approach is based on the Integer-Vector-Matrix (IVM) data structure, better adapted to the GPU than linked-list-based data structures, which are conventionally used for the storage and management of the pool of subproblems. The algorithm has two levels of parallelism. On a lower level it efficiently uses up to 1280 IVM structures to perform the branching, selection and pruning operators in parallel, exploring different parts of the B&B tree simultaneously. For each exploring IVM the bounding operator is in turn parallelized, leading to an increased overall degree of parallelism in the bounding phase. At the junction of the two levels a remapping phase is introduced in order to adapt the configuration and the mapping of the bounding kernel to the varying workload. While the mapping for the bounding is compacted, the mapping in the management phase – characterized by a very high number of data-dependent conditional instructions – is spaced, adding idle threads to the kernel. As a result, the bounding phase is accelerated on the one hand, as the control flow efficiency is improved – on the other hand, in the IVM-management phase the opposed strategy is better adapted, as it reduces thread divergence and improves memory accesses.

We have proposed two work stealing strategies for work load balancing and analyzed the scalability of our algorithm with respect to both strategies. The reported experimental results show that the performance of the proposed algorithm depends crucially on the choice of the mapping, as well as on the used work stealing strategy. The proposed GPU-IVM algorithm explores the B&B-trees of 10 Taillard’s flowshop instances *Ta021-Ta030* on average with a 3.3 times higher node processing speed than a GPU-accelerated linked-list-based B&B (GPU-LL) algorithm. For all instances *Ta021-Ta030* the GPU-IVM algorithm outperforms the GPU-LL algorithm at least by a factor 2.0 and by up to a factor 4.0 on large instances, where our GPU-IVM algorithm reaches its best performances.

As a future work we plan to investigate other work stealing strategies for the IVM-based GPU-B&B algorithm. We also plan to extend our algorithm to a multi-GPU and to a hybrid multi-core/multi-GPU Branch-and-Bound

algorithm capable of solving large instances of permutation-based combinatorial problems.

- [1] I. Chakroun, N. Melab, [Operator-level gpu-accelerated branch and bound algorithms](#), in: Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013, 2013, pp. 280–289.
URL <http://dx.doi.org/10.1016/j.procs.2013.05.191>
- [2] T. Carneiro, A. Muritiba, M. Negreiros, G. Lima de Campos, A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU, in: 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2011, pp. 41–47.
[doi:10.1109/SBAC-PAD.2011.20](#).
- [3] M. Lalami, D. El-Baz, [Gpu implementation of the branch and bound method for the knapsack problems](#), in: Parallel and Distributed Processing Symposium Workshops, 2012 IEEE 26th International, IPDPSW '12, IEEE, 2012, pp. 1769–1777.
URL <http://dx.doi.org/10.1109/IPDPSW.2012.219>
- [4] N. Melab, I. Chakroun, A. Bendjoudi, Graphics processing unit-accelerated bounding for branch-and-bound applied to a permutation problem using data access optimization, *Concurrency and Computation: Practice and Experience* 26 (16) (2014) 2667–2683.
- [5] R. Leroy, M. Mezma, N. Melab, D. Tuytens, Work stealing strategies for multi-core parallel branch-and-bound algorithm using factorial number system, in: Proceedings of Programming Models and Applications on Multicores and Manycores, PMAM'14, ACM, New York, NY, USA, 2007, pp. 111:111–111:119. [doi:10.1145/2560683.2560694](#).
- [6] M. Mezma, R. Leroy, D. Tuytens, N. Melab, [A multi-core parallel branch-and-bound algorithm using factorial number system](#), in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IPDPS '14, IEEE, 2014.
URL <http://dx.doi.org/10.1109/IPDPS.2014.124>
- [7] B. Goldengorin, D. Ghosh, G. Sierksma, Branch and peg algorithms for the simple plant location problem, *Computers & Operations Research* 31 (2004) 241–255.
- [8] R. Pastor, A. Corominas, Branch and win: Or tree search algorithms for solving combinatorial optimisation problems, *Top* 1 (2004) 169–192.
- [9] S. Climer, W. Zhang, Cut-and-solve: an iterative search strategy for combinatorial optimization problems, *artificial intelligence* 170 (2006) 714–738.
- [10] N. Melab, Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul, LIFL, USTL, thèse HDR (Novembre 2005).
- [11] V. Cung, S. Dowaji, B. L. Cun, T. Mautor, C. Roucairol, Parallel and distributed branch-and-bound/A* algorithms, Tech. Rep. 94/31, Laboratoire PRISM, Université de Versailles (1994).
- [12] B. Gendron, T. Crainic, Parallel Branch and Bound Algorithms: Survey and Synthesis, *Operations Research* 42 (1994) 1042–1066.
- [13] M. Mezma, N. Melab, E.-G. Talbi., A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems, in: In Proc. of 21th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS), Long Beach, California, 2007.
- [14] I. Chakroun, M. Mezma, N. Melab, A. Bendjoudi, Reducing thread divergence in a gpu-accelerated branch-and-bound algorithm, *Concurrency and Computation: Practice and Experience* 25 (8) (2013) 1121–1136. [doi:10.1002/cpe.2931](#).
- [15] G. Cantor, Ueber die einfachen zahlensysteme (1869).
- [16] C.-A. Laisant, Sur la numération factorielle, application aux permutations, *Bulletin de la Société Mathématique de France* 16 (1888) 176–183.
- [17] D. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Reading, Ma (1997) 192 ISBN=9780201896848.
- [18] J. McCaffrey, Using permutations in .NET for improved systems security (2003).
- [19] I. Chakroun, Parallel heterogeneous branch and bound algorithms for multi-core and multi-gpu environments, Ph.D. thesis, Université Lille 1 (2013).
- [20] S. Xiao, W.-c. Feng, Inter-block gpu communication via fast barrier synchronization, in: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–12. [doi:10.1109/IPDPS.2010.5470477](#).
- [21] J. Lenstra, B. Lageweg, A. R. Kan, A General bounding scheme for the permutation flow-shop problem, *Operations Research* 26 (1) (1978) 53–67.
- [22] S. Johnson, Optimal two and three-stage production schedules with setup times included, *Naval Research Logistics Quarterly* 1 (1954) 61–68.
- [23] M. Harris, S. Sengupta, J. D. Owens, Parallel prefix sum (scan) with cuda, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley, 2007.
- [24] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, S. Krishnamoorthy, [Lifeline-based global load balancing](#), in: Proc. of the 16th symposium on Principles and practice of parallel programming, PPOPP '11, ACM, New York, NY, USA, 2011, pp. 201–212.
URL <http://doi.acm.org/10.1145/1941553.1941582>
- [25] M. Garey, D. Johnson, R. Sethi, The complexity of flow-shop and job-shop scheduling, *Mathematics of Operations Research* 1 (1976) 117–129.
- [26] E. Taillard, Benchmarks for basic scheduling problems, *Journal of Operational Research* 64 (1993) 278–285.