A GPU-Based Backtracking Algorithm for Permutation Combinatorial Problems

Tiago Carneiro Pessoa[1](✉), Jan Gmys[2,3], Nouredine Melab[3], Francisco Heron de Carvalho Junior[1], and Daniel Tuyttens[2]

[1] ParGO Research Group (Parallelism, Optimization and Graphs), Mestrado e Doutorado em Ciência da Computação, Universidade Federal do Ceará, Fortaleza, Brazil
{carneiro,heron}@lia.ufc.br

[2] Mathematics and Operational Research Department (MARO), University of Mons, Mons, Belgium
{jan.gmys,daniel.tuyttens}@umons.ac.be

[3] INRIA Lille Nord Europe, Université Lille 1, CNRS/CRIStAL, Cité scientifique, 59655 Villeneuve D'Ascq, France
Nouredine.Melab@univ-lille1.fr

Abstract. This work presents a GPU-based backtracking algorithm for permutation combinatorial problems based on the Integer-Vector-Matrix (IVM) data structure. IVM is a data structure dedicated to permutation combinatorial optimization problems. In this algorithm, the load balancing is performed without intervention of the CPU, inside a work stealing phase invoked after each node expansion phase. The proposed work stealing approach uses a virtual n-dimensional hypercube topology and a triggering mechanism to reduce the overhead incurred by dynamic load balancing. We have implemented this new algorithm for solving instances of the Asymmetric Travelling Salesman Problem by implicit enumeration, a scenario where the cost of node evaluation is low, compared to the overall search procedure. Experimental results show that the dynamically load balanced IVM-algorithm reaches speed-ups up to 17× over a serial implementation using a bitset-data structure and up to 2× over its GPU counterpart.

Keywords: GPU computing · Backtracking · Depth-first search · Load balancing · Work stealing

1 Introduction

Graphics Processing Units (GPUs) have been used to substantially accelerate many regular applications. In such applications, the threads, organized in 1D, 2D or 3D blocks, perform identical operations on contiguous portions of data in

T.C. Pessoa was partially supported by the Institutional Program of Overseas Sandwich Doctorate (PDSE-CAPES) grant 3376/2015-00.

a statically predictable manner [1]. However, there are applications where the degree of parallelism, control flow, memory access and communication patterns are irregular and unpredictable. They are known as irregular or unstructured applications [11,18]. Backtracking, a search strategy that dynamically generates and explores a tree in a depth-first order, falls into this class of applications.

Backtracking algorithms are highly parallelizable, because many processes can explore different regions of the search space in parallel [10]. Due to the pruning of branches, the shape of the explored tree is irregular and unpredictable, resulting in load imbalance, diverging control flow and scattered memory accesses. These irregularities can be highly detrimental to the overall performance of GPU-based backtracking algorithms [6]. Thus, load balancing is one of the most critical components of parallel backtracking algorithms [11].

Some efficient strategies for load balancing inside the GPU were proposed for depth-first Branch-and-Bound (B&B) algorithms applied to permutation problems [8]. B&B is a systematic tree search strategy that uses a bounding operator which computes bounds on the optimal cost of subproblems to decide whether to continue their exploration. These bounds are often obtained by solving a relaxation of the problem at hand. In B&B algorithms the bounding operator is often very time-consuming. In this situation, the overhead induced by dynamic load balancing is easily compensated by the performance gains that result from a more regular workload. In such coarse-grained cases GPUs can provide speedup factors of 100 and more over sequential single-core CPU implementations. However, GPU-based backtracking performs poorly in cases where the evaluation cost of a node is low. It is reported that, in such irregular and fine-grained scenarios, GPU-based algorithms may be outperformed by single-core CPU implementations, or, attain much lower speed-ups than the ones obtained in situations where the node evaluation is computationally intensive [3,7–9].

In this paper we consider the case where the cost of evaluating a node is very low, meaning that the focus is put on the implementation of the parallel search process. For fine-grained problems, the implementations of such mechanisms is even more challenging and it has been loosely addressed in the literature [9,11,13]. This work proposes a new load balance strategy for GPU-based backtracking, based on the Integer-Vector-Matrix (IVM) data structure. In this algorithm, load balancing is performed without intervention of the CPU, inside a work stealing phase, which is invoked after each node expansion phase. The proposed approach uses a virtual n-dimensional hypercube topology and a triggering mechanism to reduce the overhead incurred by dynamic load balancing.

As a test-case, we solve instances of the Asymmetric Travelling Salesman Problem (ATSP) by implicit enumeration, a scenario where the evaluation of a node requires almost no arithmetic operations (two integer additions and one comparison). We compare the proposed IVM-based algorithm with a bitset-based GPU-backtracking algorithm for fine-grained problems and its serial version. Experimental results show that the dynamically load balanced IVM-algorithm outperforms the static bitset-based GPU algorithm on 60% of the test-cases that show speedup. The proposed GPU-based backtracking algorithm reaches speedups up to 17× over the serial algorithm using a bitset-data structure and up to 2× over the bitset GPU algorithm with no load balancing.

The remainder of this paper is organized as follows. Section 2 presents the background and related works. Section 3 describes the proposed algorithm. Section 4 presents details about methodology of evaluation and analysis of results. Conclusions are presented in Sect. 5.

2 Context

2.1 Test Case: Asymmetric Travelling Salesman Problem

The Asymmetric Travelling Salesman Problem (ATSP) is a well-known permutation-based combinatorial optimization problem with many real-world applications [5]. It consists in finding the shortest Hamiltonian cycle(s) through a given number of cities in such a way that each city is visited exactly once. For each pair of cities i, j a cost c_{ij} is given and stored in a cost matrix $C_{N \times N}$. The TSP is called *symmetric* if the cost matrix is symmetric ($\forall i, j : c_{ij} = c_{ji}$), and *asymmetric* otherwise.

The ATSP instances used in this case-study come from the instance generator proposed by [4], which creates instances using properties found in real-world situations. Three classes of instances were selected: *coin*, modeling a person collecting money from pay phones in a grid-like city; *crane*, modeling stacker crane operations and *tsmat*, consisting of asymmetric instances where the triangle inequality holds. We use instances from size 10 to 20.

2.2 Parallel Backtracking

Backtracking is a search strategy that consists in exploring the nodes of a tree, which is dynamically generated in depth-first fashion [10]. Internal nodes of this tree are incomplete solutions and leaves are solutions. The search begins at the root of the tree. Each step of the algorithm generates and evaluates a new node, more restricted than its father node. These newly generated nodes are kept inside a data structure, usually a stack. At each iteration, a node is removed from the data structure. The search strategy continues to generate and evaluate nodes until the data structure is empty. During the search procedure, an undesirable node may be reached, so the algorithm backtracks to an unexplored (frontier) node. This action prunes some regions of the solution space, keeping the algorithm from unnecessary computation.

Backtracking search strategies are well suited candidates for parallelization. One parallel model consists in evaluating/expanding nodes in parallel. Another approach, used in this work, consists in splitting the tree among processes, such that each process independently explores a different part of the search space in parallel [10, 11].

2.3 GPU-Based Backtracking Strategies

GPU backtracking strategies for fine-grained combinatorial problems usually consist in two steps: initial CPU backtracking and parallel backtracking on GPU [2,3,7,13,15,16]. The initial CPU search performs a depth-first search (DFS) until a cutoff depth d_{cpu} is reached. All objective nodes (frontier nodes at d_{cpu}) are stored in the Active Set A_{cpu}, which keeps all nodes evaluated but not yet branched. The cutoff depth is problem-dependent and ad-hoc defined parameter. For a puzzle problem, the depth d_{cpu} may, for instance, express the configuration of the puzzle after d_{cpu} modifications. For ATSP, it means a permutation represented by an incomplete Hamiltonian cycle with d_{cpu} cities.

After the initial CPU search, A_{cpu} is sent to the GPU, and the backtracking kernel is configured and launched by the CPU. In the kernel, each node belonging to A_{cpu} is a concurrent backtracking root R_i. Each thread Th_i is responsible for evaluating a subset S_i of the solutions space concurrently, as one can see in Fig. 1. The GPU search ends when all threads Th_i have finished the exploration of S_i.

This kind of parallel backtracking strategy performs well in regular scenarios [3,9,13], but faces strong performance degradations in more irregular ones [7]. The main reason is that it suffers from load imbalance and instruction flow divergences. In order to achieve a good utilization of the multiprocessors, this kind of parallel backtracking strategy needs to launch a huge amount of threads [9].

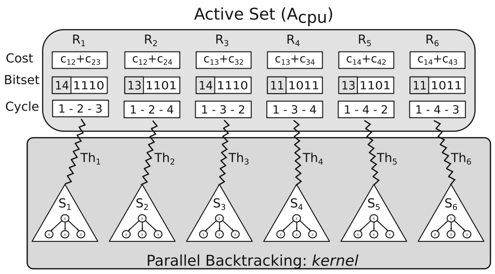


Fig. 1. Each thread Th_i evaluates a subset S_i of the solutions space.

2.4 Data Structures for GPU-Based Backtracking

Backtracking algorithms usually use a stack to store the frontier nodes. However, dynamic memory allocations on GPUs are slow. So, the use of dynamically allocated stacks may be harmful for the performance of GPU-based algorithms [15].

Other data structures may be used, such as bitsets [17]. Instead of performing operations on each position of a vector, set operations can be done in constant time using instruction-level parallelism. Backtracking algorithms may use bitsets to accelerate set operations and reduce the amount of memory used per thread [7,15]. Algorithms that apply this kind of instruction level parallelism are often called bit-parallel algorithms (BP).

The GPU-based backtracking algorithms mentioned in Sect. 2.3 use very similar data structures, thus, we describe the one used by [2,3] for solving ATSP by implicit enumeration. It is illustrated in Fig. 1.

The search strategy applied is a non recursive version of backtracking that uses no dynamic data structures. On the CPU, for representing a node that keeps the current state of the search the algorithm uses a `char` vector of size d_{cpu} and two integer variables. The vector of `char` keeps the incomplete solution and the first integer contains the cost of this sub-cycle. The second integer is seen as a bitset that keeps track of the cities already visited by the salesman. The bit k of this integer is set if the city k has already been visited.

When the initial CPU search finds an objective node at depth d_{cpu}, it stores this node and its properties in the so-called *active set* A_{cpu}. The active set A_{cpu} contains three vectors: $A_{cpu}.cycles$, $A_{cpu}.bitsets$, $A_{cpu}.costs$. In order to avoid dynamic memory allocations, the memory required for storing A_{cpu} is pre-allocated, based on the upper bound of nodes expected at depth d_{cpu}, i.e. $max_{cpu} = \frac{(N-1)!}{(N-d_{cpu})!}$ nodes. So, the vector $A_{cpu}.cycles$ requires $max_{cpu} \times d_{cpu}$ bytes. The vectors $A_{cpu}.bitsets$ and $A_{cpu}.costs$ require $max_{cpu} \times sizeof(int)$ bytes each. Once the active set A_{cpu} is filled, it is sent to the GPU.

On the GPU, each thread Th_i uses its own vector of `char` of size N, bitset and integer variables. Before beginning the search, each explorer thread initializes its local data structure with the values of the root node represented by A^i_{cpu}.

2.5 Load Balancing Strategies for GPU-Based Backtracking

Load balancing mechanisms are critical components of parallel backtracking algorithms [11]. The employed load balancing approach is intimately linked to the data structure which is used to store and manage the pool of frontier nodes. For instance, if each explorer has its own stack, a work stealing approach using stack splits may be used. In this approach idle explorers steal a fixed portion of another explorer's stack.

A CPU-GPU stack-splitting strategy is proposed in [9]. In this algorithm, each warp has its own stack and the load-balancing is done by the CPU after each iteration of the algorithm. However, due to the irregular and fine-grained nature of the problem solved, this strategy could not obtain high speedups compared to the serial version of the same algorithm, reaching speedups up to 2.25×. In the node-based approach described on Sect. 2.3, the node representation makes difficult to share work among processes. On this scope, a trigger mechanism to halt the kernel and redistribute the load among the processors was proposed for SIMD architectures by [11].

In order to tackle the workload imbalance in GPU DFS-B&B algorithms for permutation problems, a work stealing approach has been proposed by [8]. This approach uses an Integer-Vector-Matrix (IVM) [14] data structure, dedicated to permutation COPs, that will shortly be discussed in detail. The load balancing is performed without intervention of the CPU, inside a work stealing phase, which is invoked after each node expansion phase.

3 A GPU-Based Backtracking Algorithm for Permutation Problems

IVM-based work stealing approaches are proposed for multi-core and, respectively, GPU-based B&B algorithms in [8,14]. In B&B algorithms the cost of evaluating a node is usually high and the overhead induced by work stealing is easily compensated by the performance gains that result from a more regular workload. Furthermore, the high cost of the node evaluation function may justify the use of a second level of parallelism, in which the generated children nodes are evaluated in parallel. Using such a two-level parallelization, as in [8], means that fewer explorers are needed to yield good device occupancy.

In this paper, we consider the opposite case, where the cost of evaluating a node is very low. As a consequence, only parallel tree exploration but no further parallelization of the node evaluation is used. In this section, we present a GPU backtracking algorithm based on the IVM data structure. In this algorithm, load balancing is performed without intervention of the CPU, inside a work stealing phase, which is invoked as soon as the workload decreases below a certain level. The new algorithm has been implemented for solving ATSP by implicit enumeration.

The evaluation of a node consists in updating the current cost of the subcycle and comparing this partial cost with the best solution found so far. Nodes whose partial cost is greater than the best solution found so far are eliminated. Using such a naive bounding operator, the considered workload is extremely fine-grained, providing a good test-case for the proposed load-balancing mechanism. The concepts herein applied can be used to solve other permutation based problems, such as N-Queens, flow shop scheduling problem and quadratic assignment problem.

In what follows, we detail the IVM data structure, the work stealing phase and the trigger mechanism used by the proposed algorithm.

3.1 IVM Data Structure

Integer-Vector-Matrix (IVM) [14] is a data structure dedicated to permutation problems. It is illustrated in Fig. 2, using a permutation problem of size four. The left-hand side (Fig. 2a) shows a tree-based representation in which the horizontal and vertical solid lines represent the state of the corresponding stack. In the tree-based representation, each node designates an incomplete solution (partial permutation) or a full solution (permutation). For the ATSP, the cities before the "/" symbol are visited while the following ones remain to be visited. On the right-hand side (Fig. 2b) the corresponding IVM data structure is represented.

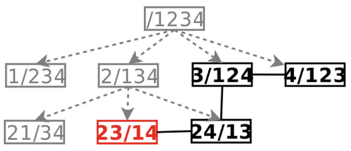
At each moment, IVM indicates the next node (subproblem) to be processed. The integer I of IVM indicates the level of the next subproblem and at each level $k \leq I$ the components $V[k]$ of the vector point to the selected cities. In this example, cities 2 and 3 are visited at levels 0 and 1, respectively. The triangular matrix M contains the cities that remain to be visited at each level. A subproblem is decomposed by increasing the level, i.e. the value of I, and copying all cities except the selected one to the next row in M. In this example, cities 1, 3 and 4 are copied to row 1, as city 2 is selected at level 0.

Using the matrix M, the vector V indicates the position of a subproblem among its sibling nodes in the tree. Therefore, throughout the depth-first exploration process, the vector V behaves like a factoradic counter. In the example of Fig. 2b, the vector successively takes the values *0000*, *0010*, *0100*, …, *3200*, *3210*. These 24 values correspond to the numbering of the 4! solutions using the *factorial number system* [12] in which the weight of the i^{th} position is equal to $i!$ and the digits allowed for the i^{th} position are $0, 1, \ldots, i$.

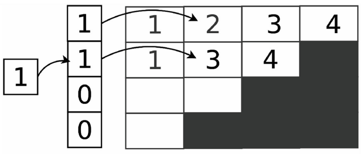
In the example of Fig. 2b, the serial backtracking algorithm explores the interval $[0000, 3210[$. It is possible to have two IVM-based workers, R_1 and R_2, such that R_1 explores $[0000, X[$ and R_2 explores $[X, 3210[$. If R_2 ends exploring its interval before R_1 does, then R_2 steals a portion of R_1's interval. Therefore, R_1 and R_2 can exchange their interval portions until the exploration of all $[0, N![$. In the IVM-based approach intervals of factoradics are used as work units.

Each cell of the triangular $N \times N$ matrix corresponds to a subproblem which, in the tree-based representation, is kept in memory as a permutation of N integers. Thus, the worst-case memory footprint of IVM is N times lower than the worst-case memory requirements for stack-based DFS. Moreover (as for bitset representations), the memory requirements of IVM are known in advance, making the IVM data structure particularly suitable for a GPU implementation of DFS.

However, because of the matrix M IVM has a larger memory footprint than the bitset representation introduced in Sect. 2.3. This larger memory footprint can be seen as a tradeoff for having work units that can be efficiently split among workers.



(a) pool of subproblems



(b) IVM-structure

Fig. 2. Example of a pool of subproblems and an IVM-structure obtained when solving a permutation problem of size four.

3.2 Work Stealing for GPU-based Parallel Backtracking

In this section we present the work stealing mechanism for the IVM-based backtracking algorithm. The pseudo-code of this algorithm is shown in Algorithm 1. Before starting the exploration, the interval $[O, N![$ is partitioned into T parts, where T is the number of IVM structures used (*line 2*). These intervals are copied to the device (*line 3*).

Algorithm 1. `IVM-Backtracking`

```
1: procedure IVM-BACKTRACK
2:     intervals←divide-interval([0,N!])
3:     copyH2D(intervals)
4:     repeat
5:         count←0 ; copyHostToDevice(count) ;
6:         call kernel TreeExplore(count)
7:         call WorkStealing-phase                         ▷ shown in Algorithm 2
8:         call kernel determine-end(end-all,states)
9:         copy DeviceToHost(end-all)
10:    until (end-all)
11: end procedure
12: procedure KERNEL : TREEEXPLORE(count)
13:    $Th_i$ ← blockIdx.x*blockDim.x+threadIdx.x
14:    repeat
15:        if (not-interval-empty($Th_i$)) then
16:            go to next node using IVM($Th_i$)
17:            state[$Th_i$] ← 1
18:        else
19:            state[$Th_i$] ← 0
20:            atomicIncrement(count);
21:            break;
22:        end if
23:    until (count<TRIGGER)
24: end procedure
```

The algorithm starts by initializing a global device variable `count` at 0 (*line 6*). Its purpose will be explained shortly. Then, the kernel `TreeExplore` is launched with T threads (*line 6*). The body of this kernel is shown in lines 11 to 24 of Algorithm 1. In this kernel each thread $Th_i \in \{0, ..., T-1\}$ uses its IVM structure to explore a distinct interval (*line 16*).

When a thread finds its interval empty, then its `state` flag is set to *empty* and the variable `count` is atomically incremented in global memory (*lines 19–21*). As mentioned, this global counter is reset to 0 before the launching of the `TreeExplore` kernel and it is only incremented by threads that have run out of work. Before exploring a new node, each thread checks the value of this counter and compares it to a fixed value `trigger` (*line 23*). If the value of the counter is greater than `trigger`, the thread stops the exploration process. In other words, the kernel `TreeExplore` kernel terminates only if at least `trigger` explorers have finished exploring their local interval.

A trigger value equal to 1 corresponds to the situation where the work stealing phase is invoked as soon as a single explorer finds it interval empty. If `trigger` is set to a value $\geq T$ no load balancing is used, since the work stealing phase is only triggered if all intervals have been explored.

After the termination of the `TreeExplore` kernel, the algorithm enters a work stealing phase where workers with empty intervals try to acquire work from workers with non-empty intervals (*line 7*). The pseudo-code for the work stealing phase, which is explained shortly, is shown in Algorithm 2. Following each work stealing phase a parallel reduction is performed on the vector of `states` in order to determine whether all IVMs are in the *empty* state (Algorithm 1, *line 8*). Until this condition is true (*line 10*), the algorithm continues to alternate exploration and work stealing phases.

In the work stealing phase, an empty IVM becomes a *thief* that tries to steal a portion of an exploring IVM's (a *victim*'s) interval. These work stealing operations must be performed in parallel and without using synchronization primitives. The work stealing phase, described in Algorithm 2, is composed of several kernels. First, the length of all intervals and the mean interval-length are computed (*lines 2–3*). This information is used in the victim selection phase where a mapping of *empty* onto *exploring* IVMs is build. As the stealing operations are carried out in parallel, the parallel victim selection must avoid the double selection of victims.

In the victim selection phase (*lines 4–8*), IVMs are seen as vertices of a n-dimensional p-ary hypercube. Each IVM is labeled with a unique ID R that can be written as $R = (a_{n-1}, a_{n-2}, ..., a_0)$ in base p. We assume that the number of IVMs, say T, is a power of p, i.e. $T = p^n$. Two IVMs whose base-p label differs in one single digit are connected to each other. Thus, in this topology, each IVM has $n(p-1)$ neighbors and the diameter of the graph connecting IVMs is n.

The victim selection phase is an iterative procedure which consists in launching a kernel $(n-1)(p-1)$ times (*line 6*). The pseudo-code of this kernel is given in lines 11–17. At iteration $i \cdot p + j$ $(i = 0, \dots, n-1; j = 1, \dots, p-1)$, an attempt is made for all empty IVMs to select the IVM of ID $R_v = (a_{n-1}, a_{n-2}, \cdots, (a_i + j)\%p, ..., a_0)$ as a victim.

The matching succeeds if the IVM R_v has not yet been selected, and IVM R_v has a non-empty interval whose length is greater than the mean interval-length. The ID of an IVM R_v that is selected as a victim by IVM R is stored at the R^{th} position of a vector `victim`, i.e. $victim[R] = R_v$ (*lines* 14, 15). This mapping of empty onto exploring IVMs is used in the kernel `stealWork`, where IVM R steals the right half of IVM R's interval.

Algorithm 2. `Work stealing phase`

```
1:  procedure WorkStealing-phase
2:      call kernel computeLengths
3:      call kernel computeMeanLength
4:      for (i:1→n) do
5:          for (j:1→p) do
6:              call kernel trySelect(i, j, victim, ...)
7:          end for
8:      end for
9:      call kernel stealWork(victim, ...)
10: end procedure
11: procedure Kernel trySelect(i, j, victim, ...)
12:     /*NB: the operations in this kernel can be performed in base 10 - for clarity this
        pseudo-code describes them in base p*/
13:     $(a_{n-1}, a_{n-2}, ..., a_0)$←blockIdx.x*blockDim.x+threadIdx.x;
14:     if (has-work($a_{n-1}, ..., (a_i + j)\%p, ..., a_0$) AND length($a_{n-1}, ..., (a_i +
        j)\%p, ..., a_0$)>mean-length) then
15:         victim[$(a_{n-1}, a_{n-2}, ..., a_0)$]← $(a_{n-1}, ..., (a_i + j)\%p, ..., a_0)$
16:     end if
17: end procedure
18: procedure Kernel stealWork(victim,...)
19:     ivm←blockIdx.x*blockDim.x+threadIdx.x
20:     v←victim[ivm]
21:     interval[ivm]←steal-half(interval[v])
22: end procedure
```

4 Performance Evaluation

In this section, we evaluate the parallel backtracking algorithm proposed in Sect. 3. In Sect. 4.1, we describe the experimental protocol. In Sect. 4.2, we provide additional parameter settings. Finally, we report and discuss the results in Sect. 4.3.

4.1 Experimental Protocol

We compare the algorithm proposed in Sect. 3 to the bit-parallel version of the GPU-based backtracking algorithm described in Sect. 2.3 (BP-DFS) and its serial version.

To compare the performance of two backtracking algorithms, both should explore the same search space. When an instance is solved twice using a parallel tree search algorithm, the number of explored nodes varies between two resolutions. Therefore, for all instances, the initial upper bound (cost of the best found solution) is set to the optimal value, and the search proves the optimality of this solution. This initialization ensures that exactly the critical subtree is explored, i.e. the nodes visited are exactly those nodes who have a partial cost lower than the optimal solution. For each of the ATSP instances *coin*10–20, *crane*10–20, *tsmat*10–19, the number of nodes (in millions) that are decomposed for proving the optimality of the initial upper bound is shown in Table 1. For *tsmat*20, the time limit of 6 h of parallel processing was exceeded.

As one can see from Table 1 the size of the explored tree increases rapidly with the instance size, ranging from a few thousand to millions of millions of nodes. On each experiment, execution times and resulting backtracking tree have been collected.

Table 1. Number of nodes decomposed during the resolution of ATSP instances *coin*10–20, *crane*10–20, *tsmat*10–19 (in 10^6 nodes), initialized at the optimal solution

Instance-#	10	11	12	13	14	15	16	17	18	19	20
crane	0.04	0.11	0.67	3.81	43.6	218.8	1,088	6,954	37,916	245,204	1,055,804
coin	0.11	0.43	1.87	10.7	107.8	500.4	1,379	3,710	15,089	116,840	674,308
tsmat	0.03	1.01	0.71	26.4	89.8	6,578	3,979	240,292	2,903,808	6,866,667	–

4.2 Parameters Settings

According to preliminary experiments the number of used IVM-structures is set to $T = 8^5 = 32,768$. Moreover, preliminary experiments were conducted to find a suitable value for the work stealing trigger. Figure 3 shows the node processing speed (in *decomposed nodes/second*) for different values of the trigger when solving instance `coin15`.

As explained in Subsect. 3.2, if the value of the trigger is too low, too much time is spend in the work stealing phase. On the other hand, if the trigger-value

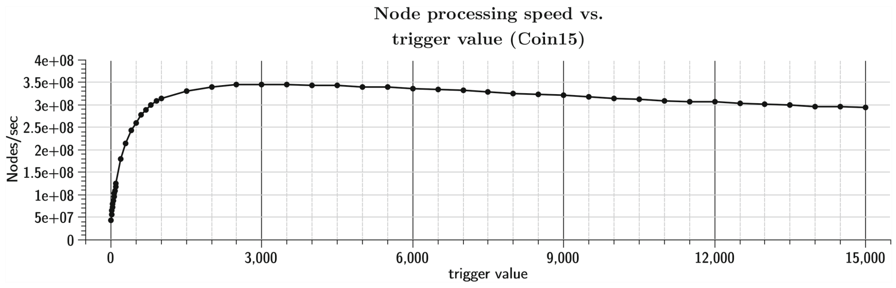


Fig. 3. Experimental calibration of the trigger value: node processing speed for different trigger-values, solving instance `coin15` using 32,768 IVM structures.

is too high, load balance degrades too much as the work stealing mechanism becomes inefficient. As one can see in Fig. 3, the best performance is reached for a value of about 3, 000, i.e., about 10% of the used 32, 768 IVM structures.

BP-DFS is a revisited version of the backtracking algorithm proposed by [2] for solving ATSP. This implementation is different from the original version as it uses a bitset-based data structure, as described in Sect. 2.3. The values chosen for block size and depth d_{cpu} are 128 and 7, respectively. The setting of parameters, such as block size, d_{cpu} and number of explorers was determined experimentally, since these parameters are influenced by instance's properties and system's characteristics [2,8,13,15].

Both GPU-implementations are based on CUDA C 7.5 and compiler versions NVCC 7.5 and GCC 4.8.2 are used. The kernel execution time is measured through CUDA's `cudaEventRecord` function, whereas the `clock` function of C is used to measure the overall application time. The testbed environment, operating under Linux Ubuntu 14.04.3 LTS 64 bits, is composed of an Intel Xeon E5-2630 v3 @ 2.40 GHz with eight cores and 32 GB RAM. It is equipped with a GeForce NVIDIA GTX 980 (GM204 chipset, 4 GB RAM, 2048 CUDA cores @ 1126 MHz).

4.3 Experimental Results and Discussion

Figure 4 shows the achieved node processing speed (in 10^6 nodes/sec) for both GPU-based implementations, considering only the kernel time and the tree processed on GPU. For the BP-DFS implementation, the node processing rate grows for instances from size 10 to 14, where it reaches a peak. This behavior is strongly related to the occupancy achieved, as mentioned in Sect. 2.3.

For instance, the initial CPU search is able to generate more threads for the instance $tsmat15$. Thus, $tsmat15$ reached occupancy of 75% (the same for $tsmat11$), resulting in a bigger $nodes/sec$ rate. Instances $coin15$ and $crane15$ reached occupancy of 64% and 57%, respectively. For problems of size 14, $coin14$ could reach occupancy of 66%, while $tsmat14$ and $crane14$ could reach 61% and 56% respectively. For the IVM-based implementation, the occupancy value is almost constant for instances of size bigger than 13 cities, reaching on average

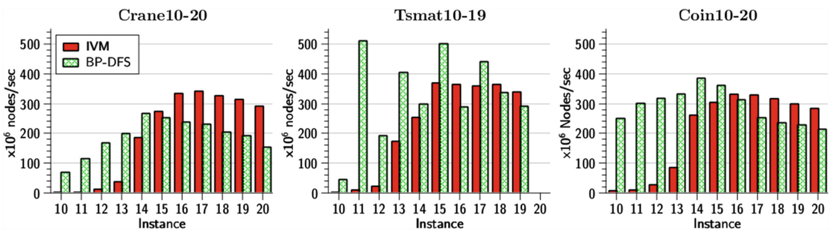


Fig. 4. Node processing speed (in 10^6 Nodes/sec) for ATSP instances *crane*10–20, *tsmat*10–19 and *coin*10–20 initialized at the optimal solution.

65% of occupancy. Due to shared memory utilization, the occupancy of the IVM-kernel was limited to 75%.

The IVM implementation reaches low *nodes*/*seconds* rates for instances of size 10 to 13, being much slower than the BP-DFS implementation. The resolution time for those instances is in the order of a few milliseconds. There are two reasons for this poor performance on small instances. On the one hand, for instances 10–13 the overhead induced by work stealing amounts on average for 60% of the execution time. One can see this in Fig. 5, which shows the percentage of time the IVM-based algorithm spends in both phases. For instances of size 14–16 the IVM-algorithm spends 10–20% of time in the load balancing phase and work stealing amounts for less than 2% for instances of sizes 17–20. The time spent in the work stealing phase is also a good indicator for the irregularity of an instance. For example, comparing the instances of size 15 one can see that relatively few time is spent in work stealing for the *tsmat* class. At the same time Fig. 4 shows that BP-DFS perform particularly well for *tsmat*15. On the other hand, the IVM-based exploration process is more costly than the bitset-based approach. In other words, for small and/or regular workloads the IVM-based approach is clearly outperformed by its bitset-based counterpart.

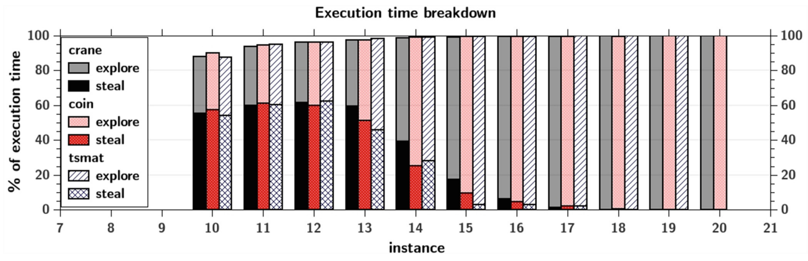


Fig. 5. Percentage of execution time spent in work stealing and exploration phases for ATSP instances *crane*10–20, *tsmat*10–19 and *coin*10–20.

Figure 4 also shows that node processing rate achieved by BP-DFS varies strongly according to the instance being solved (comparing, for example *tsmat*15 and *tsmat*16). In contrast, the IVM-based algorithm maintains similar node processing rates for all instances larger than size 14. Table 2 shows the serial execution time (in seconds) and the speedups obtained by the IVM and BP-DFS algorithms, respectively. It also shows the speedup of the IVM-based implementation over BP-DFS. Contrary to Fig. 4, the time required for initial memory allocations, copies and CUDA API calls is included in Table 2. Speedups greater than one are printed in boldface characters.

Both parallel implementations are unable to obtain speedups on instances of sizes smaller than 13 cities. Instances of size 10–12 have the serial execution time of a fraction of second. For these instances, the cost of memory allocation on GPU, data transfer and CUDA calls exceeds the time required to processing the tree sequentially on the CPU.

Table 2. Serial execution time (in seconds) and speedups for the IVM and BP-DFS algorithms.

	crane				**coin**				**tsmat**			
Inst-#	t_{serial} (sec)	$\frac{t_{serial}}{t_{IVM}}$	$\frac{t_{serial}}{t_{BP-DFS}}$	$\frac{t_{IVM}}{t_{BP-DFS}}$	t_{serial} (sec)	$\frac{t_{serial}}{t_{IVM}}$	$\frac{t_{serial}}{t_{BP-DFS}}$	$\frac{t_{IVM}}{t_{BP-DFS}}$	t_{serial} (sec)	$\frac{t_{serial}}{t_{IVM}}$	$\frac{t_{serial}}{t_{BP-DFS}}$	$\frac{t_{IVM}}{t_{BP-DFS}}$
10	0.004	0.007	0.01	0.70	0.01	0.02	0.03	0.67	0.002	0.004	0.006	0.67
11	0.012	0.02	0.04	0.50	0.04	0.06	0.11	0.55	0.06	0.1	0.17	0.59
12	0.066	0.1	0.2	0.50	0.16	0.25	0.44	0.57	0.06	0.09	0.16	0.56
13	0.265	0.4	0.7	0.57	0.66	0.6	**1.6**	0.38	1.2	**1.1**	**2.8**	0.39
14	2.2	**1.9**	**4.0**	0.48	4.7	**3.7**	**6.9**	0.54	3.6	**1.3**	**5.0**	0.26
15	10.2	**5.9**	**8.0**	0.74	22.2	**8.8**	**12.2**	0.72	220	**11.6**	**16.2**	0.72
16	53.0	**13.4**	**10.6**	1.26	63.8	**13.1**	**13.0**	1.01	171	**14.7**	**12.0**	1.23
17	350	**16.6**	**11.4**	1.46	183	**15.3**	**12.0**	1.28	8,952	**13.3**	**16.3**	0.92
18	1,946	**16.6**	**10.4**	1.60	830	**17.1**	**12.9**	1.33	110,282	**13.8**	**12.7**	1.09
19	12,839	**16.4**	**8.3**	1.98	6,662	**17.4**	**13.1**	1.33	325,744	**16.1**	**13.7**	1.18
20	56,900	**15.7**	**8.2**	1.91	37,191	**15.8**	**11.7**	1.35	–	–	–	–

Compared to the IVM-based version, the BP-DFS implementation has low overhead of CUDA API calls. It has only one kernel call and low memory requirements per thread. The IVM-based algorithm requires more memory, performs an initial partitioning of the interval $[0, N![$ and uses multiple kernel calls. This explains, for example why BP-DFS reaches a higher speedup for *crane*15 (8.0$\times$ against 5.9$\times$ for IVM) although the node processing rate (excluding initialization) achieved by the IVM-based algorithm is nearly equivalent (Fig. 4).

The IVM implementation is outperformed by its BP-DFS counterpart in 9 of 22 test cases that reach speedups over the sequential algorithm. These instances are the medium-sized instances *crane*14–15, *coin*13–15 and *tsmat*13–15. In these cases the benefit of a more balanced work load does not outweigh the penalty of using a less efficient data structure and of performing work stealing operations. In *tsmat*17, BP-DFS could reach high *nodes*/*sec* rate and outperform the IVM-based implementation as well.

However, as shown in Table 2, the dynamically load-balanced IVM implementation outperforms BP-DFS in 13 of 22 test cases that reach speedups over the sequential algorithm. The IVM-based algorithm reaches speedups up to 17$\times$ over a serial algorithm using a bitset-data structure, and is up to 2 times faster then the BP-DFS version for instances *crane*19–20. Based on the results, BP-DFS may be preferable to solve small instances or to perform a complete enumeration of the search space in a depth-first manner, a quite regular application [2,9,13]. However, this is not an usual backtracking application. For larger instances, the experimental results reveal the superiority of the IVM-based algorithm over BP-DFS, an optimized version of a well-known lightweight backtracking strategy. Aside from being faster than BP-DFS for most instances that last at least one minute, the IVM-based approach is less subject to performance variations according to different tree shapes.

5 Conclusions and Future Works

We have presented a GPU-based backtracking algorithm for permutation combinatorial problems. The presented algorithm is based on the IVM data structure. In this algorithm, the load balancing is performed without intervention of the CPU, inside work stealing phases which are triggered as soon as the workload decreases below a predefined level.

The experimental results show that algorithm benefits from a more regular load even in extremely fine-grained irregular scenarios. The performance of the load-balanced IVM-based algorithm has been compared to a bitset-based backtracking (BP-DFS) implementation. Although the bitset-based data structure is less costly to maintain than IVM, the IVM-based algorithm outperforms BP-DFS in 60% of the test-cases that show speedup over the sequential implementation and is up to 2 times faster.

For some smaller instances, the cost of work-stealing strategy is high compared to the cost of exploring the whole tree. In such cases, the BP-DFS algorithm outperforms its IVM-based counterpart. For medium and large-sized instances the overhead induced by work stealing is largely compensated by the benefits of a more regular workload.

As a future work, a dynamic trigger mechanism could be considered to overcome the limitations of the proposed algorithm while solving smaller instances. Also, we plan to investigate whether the use of CUDA Dynamic Parallelism (CDP) can help to better address recursive patterns of computation on GPUs, like the ones considered in this work.

References

1. Burtscher, M., Nasre, R., Pingali, K.: A quantitative study of irregular programs on GPUs. In: 2012 IEEE International Symposium on Workload Characterization (IISWC), pp. 141–151. IEEE (2012)
2. Carneiro, T., Muritiba, A., Negreiros, M., de Campos, G.: A new parallel schema for branch-and-bound algorithms using GPGPU. In: 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 41–47 (2011)

3. Carneiro, T., Nobre, R.H., Negreiros, M., de Campos, G.A.L.: Depth-first search versus jurema search on GPU branch-and-bound algorithms: a case study. In: NVIDIA's GCDF - GPU Computing Developer Forum on XXXII Congresso da Sociedade Brasileira de Computação (CSBC) (2012)
4. Cirasella, J., Johnson, D.S., McGeoch, L.A., Zhang, W.: The asymmetric traveling salesman problem: algorithms, instance generators, and tests. In: Buchsbaum, A.L., Snoeyink, J. (eds.) ALENEX 2001. LNCS, vol. 2153, pp. 32–59. Springer, Heidelberg (2001). doi:10.1007/3-540-44808-X_3
5. Cook, W.: In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation. Princeton University Press, Princeton (2012)
6. Defour, D., Marin, M.: Regularity versus load-balancing on GPU for treefix computations. Procedia Comput. Sci. **18**, 309–318 (2013)
7. Feinbube, F., Rabe, B., von Lowis, M., Polze, A.: NQueens on CUDA: optimization issues. In: 2010 Ninth International Symposium on Parallel and Distributed Computing (ISPDC), pp. 63–70. IEEE (2010)
8. Gmys, J., Mezmaz, M., Melab, N., Tuyttens, D.: A GPU-based Branch-and-Bound algorithm using Integer–Vector–Matrix data structure. Parallel Comput. (2016). http://www.sciencedirect.com/science/article/pii/S0167819116000387
9. Jenkins, J., Arkatkar, I., Owens, J.D., Choudhary, A., Samatova, N.F.: Lessons learned from exploring the backtracking paradigm on the GPU. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6853, pp. 425–437. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23397-5_42
10. Karp, R.M., Zhang, Y.: Randomized parallel algorithms for backtrack search and branch-and-bound computation. J. ACM (JACM) **40**(3), 765–789 (1993)
11. Karypis, G., Kumar, V.: Unstructured tree search on SIMD parallel computers. IEEE Trans. Parallel Distrib. Syst. **5**(10), 1057–1072 (1994)
12. Knuth, D.: The Art of Computer Programming. Seminumerical Algorithms, vol. 2, p. 192. Addison-Wesley, Reading (1997). iSBN=9780201896848
13. Li, L., Liu, H., Wang, H., Liu, T., Li, W.: A parallel algorithm for game tree search using GPGPU. IEEE Trans. Parallel Distrib. Syst. **26**(8), 2114–2127 (2015)
14. Mezmaz, M., Leroy, R., Melab, N., Tuyttens, D.: A multi-core parallel branch-and-bound algorithm using factorial number system. In: 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Phoenix, AZ, pp. 1203–1212, May 2014
15. Plauth, M., Feinbube, F., Schlegel, F., Polze, A.: Using dynamic parallelism for fine-grained, irregular workloads: a case study of the n-queens problem. In: 2015 Third International Symposium on Computing and Networking (CANDAR), pp. 404–407. IEEE (2015)
16. Rocki, K., Suda, R.: Parallel minimax tree searching on GPU. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 449–456. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14390-8_47
17. San Segundo, P., Rossi, C., Rodriguez-Losada, D.: Recent Developments in Bit-Parallel Algorithms. INTECH Open Access Publisher (2008)
18. Yelick, K.A.: Programming models for irregular applications. ACM SIGPLAN Not. **28**(1), 28–31 (1993)