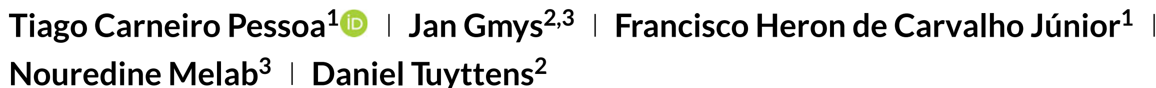
Received: 24 February 2017 | Revised: 5 September 2017 | Accepted: 1 October 2017

DOI: 10.1002/cpe.4374

WILEY

SPECIAL ISSUE PAPER

GPU-accelerated backtracking using CUDA Dynamic Parallelism



Tiago Carneiro Pessoa[1] | **Jan Gmys**[2,3] | **Francisco Heron de Carvalho Júnior**[1] | **Nouredine Melab**[3] | **Daniel Tuyttens**[2]

[1]ParGO Research Group (Parallelism, Graphs, and Optimization), Mestrado Doutorado em Ciência da Computação, Universidade Federal do Ceará, Fortaleza, Brazil
[2]Mathematics and Operational Research Department (MARO), University of Mons, Mons, Belgium
[3]INRIA Lille Nord Europe, Université Lille 1, CNRS/CRIStAL, Villeneuve-d'Ascq, France

Correspondence
Tiago Carneiro Pessoa, ParGO Research Group (Parallelism, Graphs, and Optimization), Mestrado e Doutorado em Ciência da Computação, Universidade Federal do Ceará, Fortaleza, Brazil.
Email: carneiro@lia.ufc.br

Funding information
PDSE-CAPES, Grant/Award Number: 3376/2015-00

Summary

New GPGPU technologies, such as CUDA Dynamic Parallelism (CDP), can help dealing with recursive patterns of computation, such as divide-and-conquer, used by backtracking algorithms. In this paper, we propose a GPU-accelerated backtracking algorithm using CDP that extends a well-known parallel backtracking model. The search starts on CPU, processing the search tree until a first cutoff depth. Based on this partial backtracking tree, the algorithm analyzes the memory requirements of subsequent kernel generations. The proposed algorithm performs no dynamic allocation of memory on GPU, unlike related works from the literature. The proposed algorithm has been extensively tested using the N-Queens Puzzle problem and instances of the Asymmetric Traveling Salesman Problem (ATSP) as test-cases. The proposed CDP algorithm may, under some conditions, outperform its non-CDP counterpart by a factor up to 25. But, it may also be up to twice slower. The CDP-based implementation has much better worst case execution times and makes algorithm's performance less dependent on the tuning of parameters. Compared to other CDP-based strategies from the literature, the proposed algorithm is on average 8× faster. The proposed algorithm is also hybridized with another CDP-based strategy from the literature. The combination of strategies is in average 4.5× faster than the related strategy. We also identify some difficulties, limitations, and bottlenecks concerning the CDP programming model which may be useful for helping potential users.

KEYWORDS

CUDA dynamic parallelism, depth-first search, GPU computing, parallel backtracking

1 | INTRODUCTION

Graphics Processing Units (GPUs) can substantially accelerate many regular applications. In such applications, identical operations are performed on contiguous portions of data in a statically predictable manner.[1] In contrast, irregular or unstructured applications are characterized by unpredictable and irregular control flow, degree of parallelism, memory access, and communication patterns.[2,3] Backtracking is a divide-and-conquer search strategy that consists in dynamically building and exploring a tree in depth-first order. As the shape and size of this tree are irregular and unknown in advance, backtracking falls into the class of irregular applications. Using GPUs for processing such applications is an emerging trend in GPU computing.[4,5]

Backtracking is a fundamental problem-solving paradigm in many areas, such as Artificial Intelligence and Combinatorial Optimization. The degree of parallelism in this class of algorithms is potentially very high, because the search space can be partitioned into a large number of disjoint portions which can be explored in parallel.[6] A search strategy defines which node of the tree is to be processed next. Due to its low memory requirements depth-first search (DFS) is often preferred.[7] Also, its ability to quickly find new solutions increases the efficiency of the pruning process.[8] While the pruning of branches reduces the size of the explored tree it also makes its shape irregular and unpredictable. This results in unbalanced workloads, diverging control flow and scattered memory access patterns. These irregularities can be highly detrimental to the overall performance of GPU-based backtracking algorithms.[4] Thus, the implementation of such algorithms on GPUs is challenging.

 Copyright © 2017 John Wiley & Sons, Ltd.

GPU-based backtracking strategies have been efficiently used in regular scenarios, such as using DFS to perform a complete enumeration of the solution space.[9-11] However, they face huge performance penalties in irregular ones, being outperformed even by their serial counterparts.[12] New GPGPU technologies, such as CUDA Dynamic Parallelism* (CDP), can raise the expressiveness of GPU programming because it enables GPU threads to launch new kernels dynamically. Therefore, recursive patterns of computation, such as nested parallelism and divide-and-conquer are better addressed by the GPU programming model.[5,14,15]

This paper aims at exploiting CDP in a new GPU-based backtracking algorithm. Our objective is to verify whether and how much the use of CDP can improve the performance of GPU-based backtracking algorithms in irregular scenarios. The proposed CDP-based algorithm extends a well-known parallel backtracking model.[10,-12,16]

In the proposed algorithm, the search starts on the CPU, where the tree is processed until a first cutoff depth. Based on this partial backtracking tree, the algorithm performs an analysis of the memory required by the subsequent kernel generations, and preallocates the required memory. Therefore, differently from other CDP-based backtracking algorithms,[17] GPU threads do not perform dynamic memory allocations. Our CDP-based algorithm has been extensively tested, solving instances of the Asymmetric Traveling Salesman Problem (ATSP) by implicit enumeration and by enumerating all valid solutions of the N-Queens problem.

Results show that the CDP-based implementation reaches speedup up to 25× compared to its non-CDP counterpart, for some configurations. Moreover, the experimental results show that the proposed CDP-based implementation has much better worst case execution times and makes algorithm's performance less dependent on the tuning of parameters. However, as the use of CDP induces significant overheads, the comparison also shows that a well-tuned non-CDP version can be more than twice as fast as its CDP-based counterpart. Furthermore, we show that the non-CDP implementation has performance equivalent to a multi-core backtracking that uses load balance and runs on two CPUs, with 20 cores and 40 threads.

Compared to related CDP-based strategies from the literature, our algorithm is, on average, from 4× to 11× faster. The CDP-based backtracking algorithm we propose is easily extended. Therefore, we present a hybridization of it with a related algorithm. Results show that the combination of strategies is on average 4.5× faster than the related strategy used alone.

The main contributions of this paper are the following:

- We present a CDP-based backtracking algorithm that dynamically deals with the memory requirements of the problem and avoids dynamic allocations on GPU;
- We show that the use of CDP improves the worst case execution time as it makes the algorithm's performance less sensitive to parameter tuning;
- We show that the hybridization of our algorithm with another existing CDP-based strategy from the literature considerably improves the performance of this related algorithm;
- We consider a constraint satisfaction and an optimization problem for validating our approach. Both problems can be represented as permutation combinatorial problems. However, they have different characteristics and requirements, as detailed further;
- We identify some difficulties, limitations, and bottlenecks concerning the CDP programming model, which may be useful for helping potential users and motivate lines for further investigations;
- We show that it is worth using GPU for processing backtracking, even for fine-grained and irregular workloads.

The paper is structured as follows. Section 2 presents the background and works related to this paper. Section 3 introduces the proposed algorithm. Section 4 presents details about the methodology of evaluation, parameters settings, and analysis of results. Section 5 presents a discussion of the results. Finally, Section 6 presents the conclusions and directions of further investigations.

2 | BACKGROUND AND RELATED WORKS

This section presents the background and provides an overview of related contributions in the literature. We first introduce the fundamentals of CUDA Dynamic Parallelism (CDP), followed by a discussion of existing works that apply CDP to irregular applications and recursive patterns of computations. Then we present the state-of-the-art in GPU-accelerated backtracking algorithms. Finally, we introduce the two benchmark problems used to validate the CDP-based backtracking algorithm proposed by this paper.

2.1 | CUDA dynamic parallelism

NVIDIA's Kepler architecture[13] introduced CUDA Dynamic Parallelism (CDP), making it possible to launch new grids of threads without CPU interference. CDP may be useful for refining the granularity in critical regions, or even for adapting the workload dynamically.

In CDP terminology, a thread that launches a new kernel is called *parent*. The grid, kernel and block to which this thread belongs are also called parents. The launched grid is called a *child*. The launching of a child grid is non-blocking, but the parent grid only finishes its execution

* CUDA (Compute Unified Device Architecture).[13] Dynamic parallelism is also present in OpenCL 2.0.

after the termination of its child. If any sort of synchronization between parent and child is required, CUDA synchronization functions such as `cudaDeviceSynchronize()` must be applied.[18,19] Inside a block, different kernel launches are serialized. To avoid serializations of kernel launches, the programmer must create and initialize a set of streams and issue each kernel launch to a different stream.

Concerning memory organization, child grid's blocks have their own shared memory, and threads belonging to these blocks that have their own local memory. A child grid is not aware of its parent context data. Thus, the communication between parent and child is performed through global memory. The parent thread shall not pass pointers to its shared or local memory.

The runtime system reserves memory for management, eg, for saving parent grid states or to store the pool of pending grids.

A *pending grid* is a grid that is suspended, being executed or waiting for execution.[19] A GPU can hold more pending grids than the size of the fixed pool defined by the `cudaLimitDevRuntimePendingLaunchCount` variable. This is done by using a virtualized queue, and its management consumes memory, as mentioned above. Therefore, launching a huge amount of kernels can be detrimental to the performance of the program.

Also, the GPU may be unable to handle a huge number of dynamic kernel launches, which may negatively impact the correctness of the program. The main issue is that the programmer may not be aware of such problems, as tracking runtime errors on device side is not trivial.[14]

Some works have applied CDP to develop programming abstractions for recursive computations and nested parallelism.[5,15,20] These works have found that the benefit of parallelizing nested loops by using CDP cannot outweigh its overhead, and the benefits of using CDP are still unclear. Another issue reported is that the communication between parent and child must be done through global memory, which requires additional programming efforts and imposes overhead. A lightweight mechanism to spawn parallel work dynamically without using CDP is proposed in.[20]

CDP has also been used for processing irregular applications, such as graph algorithms, clustering and simulations.[21,22] In particular,[21] proposes a strategy that launches new grids when a kernel is able to find a predetermined and regular load during its execution. Although results show speedups up to 2.73×, the use of CDP causes a slowdown on the overall performance of the benchmark algorithms. CDP-based algorithms for breadth-first search (BFS) and single-source shortest path (SSSP) are presented in.[22] According to the authors, CDP can simplify the development of GPU-based graph algorithms, because the use of CDP leads to a simpler code closer to its high-level description.

2.2 | GPGPU backtracking strategies

Backtracking algorithms dynamically build and explore a tree in depth-first fashion.[6] Internal nodes of this tree are incomplete solutions and leaves are valid solutions. The root node represents the initial problem to be solved. The algorithm iteratively generates and evaluates new nodes, where each child node is more restricted than its parent. Newly generated nodes are stored inside a data structure, conventionally a stack for depth-first search. At each iteration, the leftmost deepest node is removed from the data structure and evaluated. If this node can lead to a valid solution, it is decomposed, and its children nodes are added to the data structure. Otherwise, it is discarded from the search, and the algorithm backtracks to an unexplored (frontier) node. This action prunes (eliminates) some regions of the solution space, preventing the algorithm from unnecessary computations. The search strategy continues to generate and evaluate nodes until the data structure is empty.

There are several approaches to the parallelization of backtracking search strategies. One node-oriented parallel model consists in evaluating and expanding nodes in parallel.[2] The degree of parallelism in this model is limited and strongly depends on the characteristics of the node evaluation function. Another approach, used in this work, consists in having multiple backtracking processes to explore different parts of the search space independently.[6,23] The degree of parallelism in this tree-based approach can be very high. However, it depends on the shape of the explored tree, as the splitting of the tree among processes leads to an imbalanced workload repartition.

GPU strategies for fine-grained combinatorial problems usually consist of two steps: initial CPU backtracking and parallel tree-based backtracking on GPU.[10-12,16,24-26] The initial CPU backtracking performs DFS until a cutoff depth d_{cpu}. All objective nodes (frontier nodes at d_{cpu}) are stored in the *Active Set*A_{cpu}, which keeps all evaluated but not yet branched nodes, as shown in Figure 1. The cutoff depth d_{cpu} is a problem-dependent parameter, determined ad-hoc or through manual tuning.[27] For instance, in the N-Queens puzzle problem, d_{cpu} corresponds to the configurations of the puzzle after placing d_{cpu} queens on the board. Algorithm 1 presents a pseudocode for this strategy.

After the initial CPU backtracking (*line 4*), a subset $S \subseteq A_{gpu}$ of size $chunk \leq |A_{cpu}|$ is chosen (*line 7*). This choice may be made, for instance, based on hardware limitations. Next, A_{cpu} is updated, and the host (CPU) transfers S to the GPU's global memory (*lines* 8 − 11). Then, the host configures and launches the kernel (*lines* 12 − 14). In the kernel, each node belonging to S represents a concurrent backtracking root R_i, $i \in \{0, \dots, chunk - 1\}$.

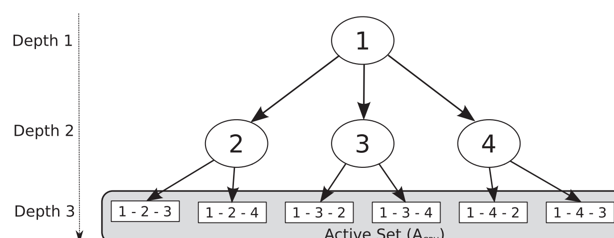


FIGURE 1 Initial CPU search for a permutation-based combinatorial problem of size 4 and $d_{cpu} = 3$

Therefore, each thread Th_i explores a subset S_i of the solution space S concurrently. The kernel ends when all threads have finished their exploration of S_i. The kernel may be called several times until A_{cpu} is empty.

Algorithm 1: CPU-GPU parallel backtracking algorithm

1 $I \leftarrow get_problem()$
2 $p \leftarrow get_gpu_properties()$
3 $d_{cpu} \leftarrow get_cpu_cutoff_depth()$
4 $A_{cpu} \leftarrow generate_initial_active_set(d_{cpu}, I)$
5 $S \leftarrow \emptyset$
6 **while** A_{cpu} *is not empty:* **do**
7     $S \leftarrow select_subset(A_{cpu})$
8     $chunk \leftarrow |S|$
9     $A_{cpu} \leftarrow A_{cpu} \setminus S$
10     $allocate_data_on_gpu(chunk)$
11     $transfer_data_to_gpu(S, chunk)$
12     $nt \leftarrow get_block_size()$
13     $nb \leftarrow ceil(chunk/nt)$
14     $parallel_backtracking <<< nb, nt >>> (I, S, chunk, d_{cpu})$
15     $synchronize_gpu_cpu_data()$
16 **end**

The described GPU backtracking strategy performs well in regular scenarios,[9,11,24] but it faces a decrease of performance in more irregular ones, being outperformed even by the serial CPU implementation in some situations.[12] The main reason for this decrease of performance is that GPUs suffer from load imbalance and diverging instruction flow. Thus, to achieve a proper utilization of the multiprocessors, this parallel backtracking strategy must launch a huge amount of GPU threads.[9]

2.2.1 | Related GPU-based backtracking strategies

GPU-based algorithms that follow the backtracking model presented in Section 2.2 are proposed by other works[10,25] for solving the ATSP. Results show that these algorithms are much faster than their multi-core counterparts in regular scenarios. For irregular workloads, the performance depends strongly on the shape of the tree and parameters tuning. In Carneiro et al,[24] two different GPU-based backtracking strategies are compared using the ATSP as a benchmark: the one proposed by Carneiro et al[10] and an improved GPU-based version of the Jurema Search Strategy.[28] It was observed that both backtracking strategies suffer from the same problems of load imbalance, and the results between them are similar. Even in the irregular scenarios used as benchmarks, both GPU implementations are faster than their multi-core counterparts.

CDP-based backtracking strategies for enumerating all valid solutions of the N-Queens problem are proposed by Plauth et al.[17] The strategies are called DP1, DP2, and DP3. The strategy called DP1 is equivalent to a parallel breadth-first search, where each frontier node found at depth d is a root of a new kernel launch. The next generation searches for frontier nodes at depth $d + 1$. The search continues this way until the whole solution space is completely evaluated.

The search called DP2 is based on two depths: d_{cpu} and d_{gpu}. Each backtracking search starting at depth d_{cpu} searches for frontier nodes at depth d_{gpu}. The first thread in a block that finds a frontier node at d_{gpu} allocates enough memory for the maximum number of frontier nodes its block can find at d_{gpu}. Then a recursive new generation of kernels is launched, searching from d_{gpu} to N. Finally, DP3 applies the concepts of DP2. However, DP3 doubles d_{gpu} at each new recursive kernel launch, adapting the new kernel launch to the shape of the tree. Results show that DP3 is superior to DP2, as it produces a much regular load to the GPU.

According to Plauth et al,[17] the overhead caused by dynamic memory allocations and dynamic kernel launches outweighs the benefits of the improved load balance yielded by CDP. All proposed CDP-based implementations cannot outperform the control non-CDP counterpart. Moreover, it is mentioned that the performance of all studied algorithms strongly depends on the tuning of several parameters, such as search depth and block size.

2.3 | Experimental benchmarks: N-Queens and the asymmetric traveling salesman problem

The Traveling Salesman Problem (TSP) 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 by 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. It is one of the most studied Combinatorial Optimization Problems (COP), having plenty of real-world applications.[29] Due to TSP's relevance, it is often used as a benchmark for novel problem-solving strategies.[30]

Solving the ATSP on GPUs by performing implicit enumeration is challenging.[10,24,25,27] Node evaluation can be done in constant time and requires few arithmetic operations (three integer operations and a comparison). In this fine-grained situation, there is no parallel node evaluation, and the main focus is put on the implementation of the parallel search process.

ATSP instances used in this work come from a generator that creates instances based on real world situations.[29] Three classes of instances have been 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.

The N-Queens puzzle[31] problem consists in placing N non-attacking queens on a $N \times N$ chessboard. We use the version of N-Queens that consists in finding *all* feasible board configurations. N-Queens is easily modeled as a permutation problem: position r of a permutation of size N designates the column in which a queen is placed in row r. To evaluate the feasibility of a partial solution, a function checks for diagonal conflicts in this partial solution. Although symmetries in this problem can be exploited, we do not make use of any of them in this work.

N-Queens is often used as a benchmark for algorithms that solve constraint satisfaction problems. It is also often used as a benchmark for new GPU backtracking strategies.[12,17,32] In contrast to ATSP, N-Queens does not require a cost matrix, and its node evaluation complexity is $\mathcal{O}(N)$. Concerning the tree size, a valid solution for the ATSP always contains the starting city in the first position. Therefore, N-Queens has the maximum theoretical tree N times bigger than the one returned by ATSP for a problem of size N. Another important difference between the two problems is that each instance of ATSP has its own characteristics.[33,34] Two instances of the same size N may result in a different behavior for the same algorithm. So the algorithm may perform well for one class of instance and poorly for another.

Because of the combinatorial nature of ATSP and N-Queens, the concepts presented by this work can be applied to solve other combinatorial optimization and constraint satisfaction problems, such as flow shop scheduling, minimum linear arrangement, and quadratic assignment.

3 | THE PROPOSED CDP-BASED BACKTRACKING ALGORITHM

In this section, we present a new GPU-accelerated backtracking strategy based on CDP and mainly designed for solving permutation-based combinatorial problems. This section is organized as follows: we first introduce the initial premises considered in the conception of the strategy, then we detail all steps of our new CDP-based backtracking algorithm.

3.1 | Initial premises

Related CDP-based strategies dynamically allocate memory on GPU if at least one objective node is found by a block of threads at d_{gpu}.[17] This strategy works well for the N-Queens for problem sizes up to $N = 16$ and using symmetries, which considerably decreases the size of the explored tree.[35] Figure 2 shows, for ATSP and N-Queens instances of size 15, the percentage of survivor nodes at depths 5 to 9. For all ATSP instances, the number of survivors is bigger than 80% when the depth is 50% of the maximum depth (N). For N-Queens, this number is much lower, around 15%.

In a CUDA-based algorithm, if dynamic allocations on GPU require more than 8*MB*, the variable `cudaLimitMallocHeapSize` must be set accordingly. However, knowing these requirements in advance is difficult, and insufficient heap size leads to runtime errors. For example, Figure 2 illustrates that the number of survivors is much bigger for class *tsmat* than for class *crane*, although both are of size ($N = 15$). Moreover, such memory requirements may be huge for permutation-based combinatorial problems.[7]

Consider the instance *tsmat*15 and the algorithm DP3, introduced in Section 2.2.1. As DP3 doubles d_{gpu} on each recursive dynamic kernel launch, for $N = 15$, d_{cpu} is 2, and d_{gpu} has the values 4 and 8. In this situation, for storing the frontier nodes at d_{gpu} the search must dynamically allocate approximately the following amount of memory (in *MB*):

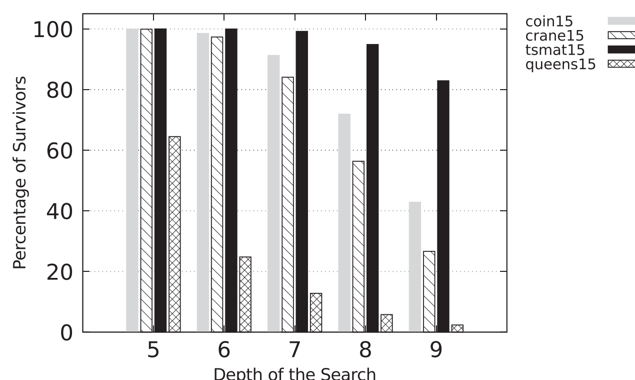


FIGURE 2 Percentage of survivor nodes at depths 5 to 9 for ATSP and N-Queens instances of size $N = 15$. The initial upper bound is set to the optimal solution

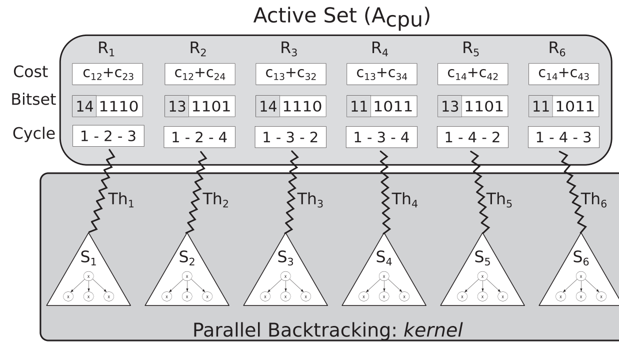


FIGURE 3 Representation of A_{cpu} for $d_{cpu} = 3, N = 4$ and $survivors = 6$

$$\left[\frac{(15-1)!}{(15-4)!} \times size\ of(Node)\right] + \left[\frac{(15-1)!}{(15-8)!} \times size\ of(Node) \times 0.9\right],$$

which is a value bigger than 600*MB*. Performing dynamic allocations of such a big amount of memory may be harmful to the performance of a GPU-based algorithm.[17,36]

Based on these premises, we propose a CDP-based algorithm that extends the parallel backtracking model presented in Section 2.2 and does not perform dynamic allocations on GPU. Memory requirements and allocation in global memory are managed by the host to avoid runtime errors. This management takes into consideration runtime requirements of CDP and properties of the device. The communication between father and child grids is performed through global memory via a thread-to-data mapping. So threads of different generations can identify data for initialization and pass data to its child grids.

In what follows, we provide a detailed description of the main steps of the proposed algorithm: Initial CPU Search, Memory Requirement Analysis, Intermediate GPU Search, and Final GPU Search. The algorithm for solving ATSP can be adapted for solving any permutation-based combinatorial problem with straightforward modifications. For instance, the algorithm for enumerating all feasible configurations for N-Queens only differs in the node evaluation function.

3.2 | Initial CPU search

The Initial CPU Search procedure is described in Algorithm 2. Before the search begins, the algorithm reads the problem size *N*, the cutoff depth d_{cpu}, and the cost matrix $M^h_{N\times N}$ (*lines 1-3*). *Host* (CPU) and *device* (GPU) data structures will be further distinguished by the superscripts *h* and *d*, respectively.

The Initial CPU Search performs DFS from the root (depth 1) until the cutoff depth d_{cpu}, storing all frontier nodes at depth d_{cpu} (valid incomplete solutions) in the active set A^h_{cpu}.

After the initial search, the variable *survivors* receives the size of A^h_{cpu} (*line 6*).

Algorithm 2: Initial CPU search

1 $N \leftarrow get_problem_size()$
2 $M^h \leftarrow get_cost_matrix()$
3 $d_{cpu} \leftarrow get_cpu_cutoff_depth()$
4 $A^h_{cpu} \leftarrow \emptyset$
5 $initial_CPU_search(A^h_{cpu}, d_{cpu}, N, M^h, upper_bound)$
6 $survivors \leftarrow |A^h_{cpu}|$

The initial search can be performed in parallel. However, the initial search explores just a small fraction of the solution space. The programmer needs to ensure that the overhead of initializing a parallel search procedure is negligible compared to the time spent exploring this small fraction of the search space.

3.2.1 | Data structures and search procedure

The data structure `Node`, which stores the current state of the search, is represented in Figure 3.

It contains a vector of integers (unsigned 8 bit for $N < 255$) of size d_{cpu}, identified by *cycle*, and two integer variables. The vector *cycle* stores the incomplete Hamiltonian cycle (solution). In turn, the first integer variable keeps the cost of this valid incomplete solution. Finally, the second integer variable, identified by *bitset*†, keeps track of visited cities by setting its bit *n* to 1 each time the salesmen visits the *n*th city.

† Backtracking algorithm may use *bitsets* to accelerate set operations and to reduce the amount of memory a GPU thread requires.[12,17,37] Algorithms that apply this kind of instruction level parallelism are often called bit-parallel algorithms (BP).

The search strategy (*Algorithm* 2, *line* 5) is a non-recursive backtracking that does not use dynamic data structures, such as stacks. The semantics of a stack are obtained by using a variable *depth* and by trying to increment the value of the vector *cycle* at position *depth*. If this increment results in a valid incomplete solution, the *depth* variable is incremented and the search proceeds to the next depth. After trying all configurations for a depth, the search backtracks to the previous depth.

When the search reaches the cutoff depth d_{cpu} with a valid incomplete solution, this node is put into the position *survivors* of the active set ($A^h_{cpu}[survivors]$) and the variable *survivors* is incremented. Thus, after the search performs line 5 (*Algorithm* 2), the active set A^h_{cpu} has the size $survivors \times sizeof(Node)$.

3.3 | Analysis of memory requirement

To avoid dynamic allocations and to cope with possible GPU memory limitations, we proceed as described in Algorithm 3.

Initially, the algorithm reads the GPU properties and the second cutoff depth d_{gpu} (lines 1-2). Then the algorithm calculates the maximum number of nodes expected at depths d_{cpu} and d_{gpu} (lines 3-4). These values will be further identified by max_{cpu} and max_{gpu}. Next, the maximum number of children nodes a survivor node at depth d_{cpu} can have at depth d_{gpu} is calculated (line 5) by

$$expected_children_d_{gpu} = \frac{max_{gpu}}{max_{cpu}}.$$

This value is used to deduce the maximum number of nodes A_{gpu} may contain (line 6):

$$max_A_{gpu}_size = survivors \times expected_children_d_{gpu}.$$

Knowing $max_A_{gpu}_size$, the next step is to calculate the amount of global memory requested by CDP (r_{cdp} bytes). This amount includes memory for allocation of A^d_{cpu} (*survivor* nodes), A^d_{gpu} ($max_A_{gpu}_size$ nodes), and control data.

If r_{cdp} exceeds the amount of available global memory on the GPU, the algorithm proceeds as follows (lines 7-13). An integer *chunk*, less or equal than *survivors*, is defined. Thus, A^d_{gpu} will contain at most $chunk \times expected_children_d_{gpu}$ nodes. The value of *chunk* is decreased until it is possible to allocate data on the GPU. After finding a suitable value for *chunk*, data is allocated on the global memory of the GPU. This allocation is done only once, because GPU processes at most *chunk* nodes. So this memory can be reused by data transfer operations and future generations of CDP kernels.

It is important to notice that CDP stores memory to keep track of the parent block states if `cudaDeviceSynchonize()` is called after a child kernel launch. According to Adinetz,[14] up to 150 MB are stored for each parent kernel generation (host included), depending on the hardware. We consider this value equal to 150 MB because the exact value a GPU stores cannot be easily obtained. Therefore, we remove 2 × 150 MB from the available global memory.

If subsequent generations of kernels dynamically allocate memory on GPU's heap, `cudaLimitMallocHeapSize` must also be added to r_{cdp}. CDP also uses memory for the management of pending grids. Thus, only a fraction of the global memory is taken into account in the call to *memo_requirement* (lines 8-9). Parameter configurations will be further detailed in Section 4.2.

Algorithm 3: Analysis of memory requirement and data allocation

1 $p \leftarrow get_gpu_properties()$
2 $d_{gpu} \leftarrow get_gpu_cutoff_depth()$
3 $max_{cpu} \leftarrow \frac{(N-1)!}{(N-d_{cpu})!}$
4 $max_{gpu} \leftarrow \frac{(N-1)!}{(N-d_{gpu})!}$
5 $expected_children_d_{gpu} \leftarrow \frac{max_{gpu}}{max_{cpu}}$
6 $max_A_{gpu}_size \leftarrow survivors \times expected_children_d_{gpu}$
7 $chunk \leftarrow survivors$
8 $r_{cdp} \leftarrow memo_requirement(p, chunk, max_A_{gpu}_size, N, M^h)$
9 **while** $r_{cdp} > p.memorySize$ **do**
10 $\quad chunk \leftarrow chunkUpdate(chunk)$
11 $\quad max_A_{gpu}_size \leftarrow chunk \times expected_children_d_{gpu}$
12 $\quad r_{cdp} \leftarrow memo_requirement(p, chunk, max_A_{gpu}_size, N, M^h)$
13 **end**
14 $cudaMalloc(A^d_{cpu}, chunk \times sizeof(Node))$
15 $cudaMalloc(A^d_{gpu}, max_A_{gpu}_size \times sizeof(Node))$
16 $cudaMalloc(M^d, N \times N \times sizeof(int))$
17 $cudaMalloc(control_data^d, \ldots)$

3.4 | Launching the intermediate GPU search

Having determined a suitable *chunk* size, the host launches the first kernel, further mentioned as *Intermediate GPU Search*, possibly several times until A^h_{cpu} is empty.

Algorithm 4 shows how the intermediate search call proceeds. Initially, two integer variables *counter* and *remaining* are initialized to 0 and *survivors*, respectively. The first, *counter*, is used to make A^h_{cpu} point to unexplored nodes (line 8) and to verify the termination of the whole search process (line 5). The second, *remaining*, is used to avoid unnecessary or out of bounds data transfers. Before the beginning of the search on GPU, a *host-to-device* (*H2D*) copy sends the active set (A^h_{cpu}) to the GPU.

After each run, the pointers of A^h_{cpu} are updated to unexplored data (line 11). Therefore, on the next iteration, up to *chunk* unexplored nodes from A^h_{cpu} are transferred to the GPU. It is not necessary to perform *H2D* copies of control data, because the GPU is responsible for this data. However, enough host memory must be allocated for $control_data^h$ before calling the intermediate GPU search for the fist time. When the GPU search is completed, the variables *counter* and *remaining* are updated and the control data from the GPU is retrieved (lines 10-15).

Algorithm 4: Launching the intermediate GPU search

1 $counter \leftarrow 0$
2 $remaining \leftarrow survivors$
3 $set_CDP_variables()$
4 $cudaMemCpy(M^d, M^h, N \times N \times sizeof(int), H2D)$
5 **while** $counter < survivors$ **do**
6     $nt \leftarrow get_block_size()$
7     $nb \leftarrow ceil(chunk/nt)$
8     $cudaMemCpy(A^d_{cpu}, (A^h_{cpu} + counter), chunk \times sizeof(Node), H2D)$
9     $intermediate_GPU_search <<< nb, nt >>> (M^d, chunk, expected_children_d_{gpu}, A^d_{cpu}, A^d_{gpu}, d_{cpu}, d_{gpu}, upper_bound, control_data^d)$
10     $syncDataD2H(control_data^h, control_data^d, chunk, expected_children_d_{gpu})$
11     $counter \leftarrow counter + chunk$
12     $remaining \leftarrow remaining - chunk$
13   **if** $remaining < chunk$ **then**
14         $chunk \leftarrow remaining$
15   **end**
16 **end**

If subsequent kernel generations allocate memory on GPU's heap, the parameter `cudaLimitMallocHeapSize` must be modified to a suitable value.[‡] If any synchronization between parent and child grids is required, the variable `cudaLimitDevRuntimeSyncDepth` must be set to the deepest synchronization level. Otherwise, the GPU may not keep memory to store the state of the parent grid. The configuration of these parameters are performed by a call to the function *set_CDP_variables* (*line 3*).

3.5 | Intermediate GPU search

Global memory is the communication interface between the host and the device, as well as between two grid generations. The Intermediate GPU Search provides a thread-to-data mapping for its subsequent generations of child grids and launches the first generation of CDP kernels. The Intermediate GPU Search is detailed in Algorithm 5.

3.5.1 | Initialization and search procedure

All frontier nodes in A^d_{cpu} are roots $R_i, i \in \{0, \dots, chunk - 1\}$, of a disjoint search space S_i, as shown in Figure 3. Thread Th_i initializes its local data with root node R_i (*lines 1-3*) and then starts its search.

The search is performed from the root's depth d_{cpu} until the GPU's cutoff depth d_{gpu}, using a global upper bound to prune. All objective nodes are stored in the memory previously allocated for A^d_{gpu}. The data structures of a node in A^d_{gpu} are the same of a node in A^d_{cpu}. However, the incomplete Hamiltonian cycle now has size d_{gpu}.

[‡] Insufficient heap size may result in the "*an illegal memory access was encountered*" error.

Algorithm 5: Intermediate GPU search

Input: Cost matrix $M_{N \times N}$, *chunk*, *expected_children_d*$_{gpu}$, A^d_{cpu}, A^d_{gpu}, d_{cpu}, d_{gpu}, and the global upper bound.

```
1  idx ← blockIdx.x × blockDim.x + threadIdx.x
2  if idx < chunk then
3      local_root ← A^d_cpu[idx]
4      if is_master_thread(idx) then
5          initialize_block_ASet(A^b_gpu)
6          block_load ← 0
7      end
8      Perform backtracking based on local root information, using d_gpu for cutoff condition, and upper_bound for pruning
9      if new objective node is found then
10         local_counter ← atomicIncrement(block_load)
11         update(A^b_gpu, local_counter, objectiveNode)
12     end
13 end
14 block_barrier()
```

3.5.2 | Block-based active set

In a block-based organization, each block bl_b, $b \in \{0, \dots, nb-1\}$, has its own active set A^b_{gpu}, where $A^b_{gpu} \subseteq A^d_{gpu}$. All threads belonging to block bl_b populate A^b_{gpu} concurrently. The active set A^b_{gpu} is then initialized by the block's master thread (line 5), pointing to some distinct region of A^d_{gpu}, as shown below:

$$A^b_{gpu} \leftarrow A^d_{gpu}[b \times (nt \times expected_children_d_{gpu}) : (b+1) \times (nt \times expected_children_d_{gpu})]$$

where the variable *nt* represents the size of block *b* (*blockDim.x*) defined in Algorithm 4, and *b* corresponds the index of the block (*blockIdx.x*).

Each block *b* has a counter *block_load*. It is atomically incremented each time an objective node is found. By using this counter, new frontier nodes are placed in contiguous positions of A^b_{gpu}. The variable *block_load* is stored in *shared memory* and also initialized by bl_b's master thread (line 6).

If the programmer wants to avoid block synchronization and to perform the search independently by each thread Th_i, each thread Th_i must have its own active set A^i, such that $A^i \subseteq A^d_{gpu}$, with the following mapping:

$$A^i \leftarrow A^d_{gpu}[i \times expected_children_d_{gpu} : (i+1) \times expected_children_d_{gpu}]$$

where *i* is the global identifier *idx* of the thread in a grid (*line 1*). In a thread-based situation, the variable *block_load* is replaced by a variable *local_load*. So that, an objective node is stored at position $A^i[local_load]$.

3.5.3 | Launching the final GPU Search

After performing the intermediate backtracking and having reached all the frontier nodes of depth d_{gpu}, the intermediate search launches the final GPU search. Some factors should be taken into consideration before presenting the algorithm. If one grid per block is launched, it is not necessary to divide *block_load* among *number_of_kernels* $\leq |bl_b|$ child kernels, which simplifies data mapping. Also, it is not necessary to deal with stream creation and destruction, since the child grid is launched on the default stream.

In contrast, launching one stream per thread based on the thread's active set requires the creation of $|bl_b|$ streams before launching the child grid. These streams avoid serialization of $|bl_b|$ kernel launches. Another issue is the number of pending grids. If we launch one kernel per thread in bl_b, the GPU could run out of memory or compromise the correctness of the algorithm.

Based on these premises, we proceed as presented in Algorithm 6. After the block barrier (Algorithm 5, line 14), if *threadIdx.x* < *number_of_kernels*, thread Th_l of block *b*, $l \in \{0, \dots, number_of_kernels - 1\}$, creates and initializes stream St_l (lines 1-4). In line 5, stream St_l receives the number of nodes to explore (*stream_load*) based on the *block_load*, *number_of_kernels*, and its index (*stream_idx*).

Each stream St_l has its own active set A^l_s, such that $A^l_s \subseteq A^b_{gpu}$. The data are mapped according to the following mapping:

$$A^l_s \leftarrow A^b_{gpu}[l \times stream_load : (l+1) \times stream_load].$$

After the initialization of A^l_s (line 6), thread Th_l launches the kernel K^l (line 9).

Algorithm 6: Launching final GPU search

```
1  if threadIdx.x < number_of_kernels and block_load > 0 then
2      cudaStream_t stream
3      steam_idx ← threadIdx.x
4      initialize(stream)
5      stream_load ← get_stream_load(block_load, number_of_kernels, stream_idx)
6      initialize_stream_ASet(A_s^l)
7      nt ← get_cdp_block_size()
8      nb ← ceil(stream_load/nt)
9      final_GPU_search <<< nt, nb, stream >>> (M^d, d_gpu, A_s^l, stream_load, upper_bound)
10     deviceSynch()
11     error_vector[b] ← get_last_error()
12     destroyStream(stream)
13 end
```

3.6 | Final GPU search

By using CDP, it is easier to combine different search strategies. Thus, it is not mandatory for the next generations of kernel calls to be based on the recursive calls of the Intermediate GPU Search. In this section, we present two different ways of performing search from d_{cpu} to the solution depth N. The first algorithm is called CDP-BP. It combines the Intermediate GPU Search with the Algorithm 1, avoiding dynamic allocations. The second one is the so-called CDP-DP3, which combines the Intermediate GPU Search with the DP3 parallel backtracking strategy introduced in Section 2.2.

3.6.1 | CDP-BP

This final kernel uses the search procedure described in Section 2.2 to search from d_{gpu} to solution depth (N). The algorithm for this kernel is described in Algorithm 7. Each frontier node belonging to the active set A_s^i, passed as argument, is a root of DFS.

Solutions are shared by keeping a block's solution *bl_sol* being updated by the block's threads (Algorithm 7, lines 5-8). Also, the global solution is updated less often by all master threads.[10,24] This operation is done also by the intermediate kernel, which checks for new solutions. However, we have omitted these details in Algorithm 4. Finally, by the end of the algorithm, each thread updates the information required by the host, such as the number of solutions, the best solution found and local tree size.

Algorithm 7: Final GPU search

```
   Input: Cost matrix M_{N×N}, cutoff depth d_gpu, A_s^i, stream_load, and the global upper bound.
1  if is_master_thread(idx) then
2      initiate_block_solution(bl_sol, upper_bound)
3  end
4  block_barrier()
5  if idx < stream_load then
6      Perform backtracking using node A_s^i[idx] as the root of the search, and using bl_sol for pruning
7      Update bl_sol if any better solution is found
8  end
9  update local information to be retrieved by the host
```

3.6.2 | CDP-DP3

This search strategy is a combination of the intermediate search, and its data mappings, with the DP3 strategy introduced in Section 2.2. It searches from d_{gpu} to the depth of a solution (N), doubling d_{gpu} at each new recursive call. DP3 is similar to the Intermediate GPU Search, but A_{gpu}^b is dynamically allocated by one thread of the block as soon as one frontier node is found at depth d_{gpu}. DP3 is also different because it launches one new grid per thread. Thus, this search strategy creates, configures, and destroys one GPU stream for each thread of the block.

3.7 | GPU-CPU synchronization

After the GPU search several *device-to-host* (*D2H*) operations are performed to get the information generated by the kernels. Information such as resulting tree size, best solution found, and the number of solutions found is returned. There is also an error checking on the error information

returned by each CDP kernel launch (Algorithm 6, line 11). There is also an error checking on the host. If any error is found, the program reports the error and aborts the execution. Otherwise, it returns the best solution found, the number of solutions found, tree size, and the kernel/total execution times.

4 | PERFORMANCE EVALUATION

In this section, we evaluate the CDP-based backtracking strategies proposed in Section 3. This section is organized as follows: First, we present the experimental protocol and parameter settings. Then we compare all CDP-based implementations. Finally, we perform a worst and best case analysis.

4.1 | Experimental protocol

We consider several backtracking approaches for solving ATSP instances and the N-Queens problem. The strategies we propose are the following:

- **CDP-BP**: corresponds to the algorithms presented in Section 3, and summarized in Section 3.6.1;
- **CDP-DP3**: implementation of the algorithm described in Section 3.6.2. It is an extension of CDP-BP. This algorithm performs the same Intermediate GPU Search as CDP-BP and calls DP3 as the Final GPU Search, which doubles d_{gpu} at each new recursive call of DP3;
- **REC-CDP** and **REC-DP3**: these implementations have the same semantics of CDP-BP and CDP-DP3. However, all kernels calls and memory allocations/deallocations are performed by the host.

For comparison, the following backtracking strategies were implemented:

- **DP2** and **DP3**: apply the ideas presented by Plauth et al[17] and introduced in Section 2.2.1;
- **BP-DFS**: bit-parallel version of the algorithm proposed by Carneiro et al[10] and used in their further works as a GPU control implementation.[24,25,27] It corresponds to the GPU-based backtracking algorithm described in Section 2.2;
- **Multi-core**: multi-threaded version of BP-DFS that uses a pool scheme for load balancing;
- **Serial**: backtracking used by Pessoa and Gomes[28] as a serial control implementation; this approach is optimized for single-core serial execution. It is around 1.4× faster than the serial implementation of BP-DFS' kernel.

All implementations listed above have ATSP and N-Queens versions. All parallel searches use the data structure described in Section 3.2.1. Table 1 presents the key differences of all GPU-based implementations. In our experiments, we are also considering the highly optimized serial backtracking algorithm which is available at Somers.[35] This implementation is also used as a CPU baseline by Plauth et al.[17] It should be noted that this algorithm uses bitsets to check for diagonal conflicts in the board configuration, leading to node evaluation in constant time. This algorithm also applies symmetries, which considerably decreases the solution space size.

To compare the performance of two backtracking algorithms, both should explore exactly the same search space.[2] This is always the case for the N-Queens problem, as the order of exploration does not affect the shape of the backtracking tree. However, for ATSP, the pruning mechanism depends on the decrease of the best solution cost found so far. Hence, when an ATSP instance is solved twice using a parallel tree search algorithm, the number of explored nodes varies between two resolutions. Therefore, for all ATSP instances, the initial upper bound is set to the optimal value. This initialization ensures that only the critical subtree is explored, ie, the search proves the optimality of the initial upper bound by visiting exactly those nodes who have a partial cost lower than the optimal solution.[27]

In each experiment, kernel and application execution times, and the size of the explored tree have been collected. We also used NVIDIA CUDA Profiler to get additional metrics. For N-Queens, we use instances from size 10 to 18. One may notice that none of the exploitable symmetries of the N-Queens problem have been used. For the ATSP, we use instances from size 10 to 19. The size of the explored tree increases rapidly with the

TABLE 1 Key differences of all GPU-Based implementations: use of CDP, number of GPU streams / CDP kernels launched, use of dynamic memory allocations, and algorithm reference

Implementation	CDP	GPU streams/CDP kernels	Dynamic allocation	Algorithms
DP2	*yes*	$\lvert A^h_{cpu}\rvert$	*yes*	DP2
DP3	*yes*	$\lvert A^h_{cpu}\rvert + k_1^{(a)}$	*yes*	DP3
CDP − BP	*yes*	$nb_h \times number_of_kernels^{(1,2)}$	*no*	2 - 7
CDP − DP3	*yes*	$nb_h + k_1$	*yes*	2 - 7 + DP3
BP − DFS	*no*	-	*no*	1, 7
REC − CDP	*no*	-	*no*	2 - 7
REC − DP3	*no*	-	*yes*	DP3

Values are for ATSP and N-Queens. $^{(a)}k_1 = \left(\sum_{d=d_{gpu}}^{base-1} survivors_d\right)^{(3,4)}$.

instance size, ranging from a few thousands to billions of nodes.[27] Instances *tsmat*18 − 19 have been excluded because the time limit of 6 hours of parallel processing was exceeded. Due to the huge amount of data collected, some results are summarized or are shown only for one size or class of instances,

where:

1. The subscript *d* means the variable concerns a given depth *d*. The subscript *h* means that the variable is used by the host to configure/launch the first kernel.
2. The variable *nb* stores the number of blocks used for kernel configuration. Thus, $nb_h = ceil(|A^h_{cpu}|/nt_h)$.
3. $survivors_d$ is the total number of survivor nodes of depth *d*.
4. DP3 doubles the value of d_{gpu} at each recursive CDP kernel launch until $d_{gpu} = base$, the recursion base. In this case, the algorithm searches from *base* to *N*. We are using the following notation: $d + 1$ means the next recursive depth. For $N = 15$: $d_{cpu=2}$, $d_{gpu} = 4, 8, 15$, and $base = 8$.

4.2 | Parameters settings

All CUDA programs have been parallelized using CUDA C 7.5 and compiled with NVCC 7.5 and GCC 4.8.2. All multi-core versions have been parallelized using OpenMP. The kernel execution time has been measured through the `cudaEventRecord` function of CUDA, whereas the overall application time has been measured through the `clock` function of C. The testbed environment, operating under CentOS 7.1 64 bits, is composed of *two* Intel Xeon E5-2650v3 @ 2.30 GHz with 20 cores, 40 threads, and 32 GB RAM. It is equipped with a GeForce NVIDIA Tesla K40 m (GK110B chipset), 12 *GB* RAM, 2880 CUDA cores @ 745 MHz. According to our experiments, the K40 m reserves 109*MB* for nesting level synchronization.

The performance of a GPU-based backtracking algorithm depends on a set of parameter configurations. Preliminary experiments have been conducted to find a suitable block size, d_{cpu} and d_{gpu} for all GPU-based parallel implementations. Figure 4A shows the experimental block size calibration for BP-DFS. All GPU-based implementations use the value of 128 for the first kernel configuration. Figure 4B shows the experimental block size calibration for the second kernel generation launched by CDP-BP. Table 2 presents the best parameter configurations for all parallel implementations. It is important to say that the chosen parameters are the best for most of instances, but not for all of them. In Table 2, the subscripts *Q* and *A* indicate that the parameter setting concerns the N-Queens or, respectively, the ATSP problem.

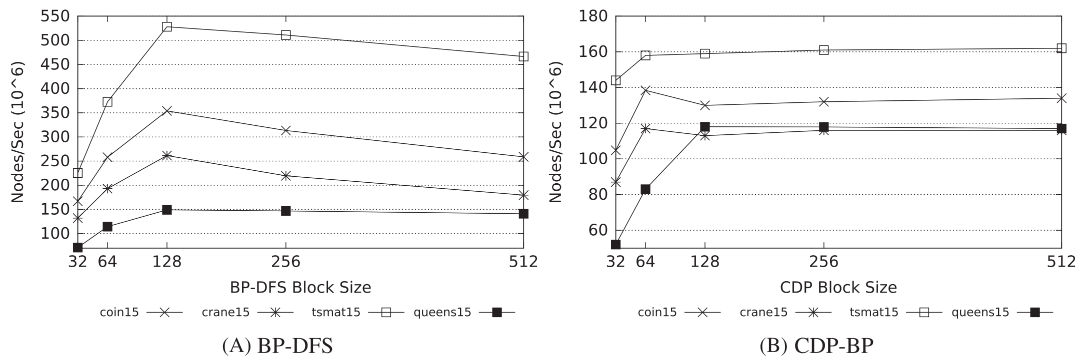


FIGURE 4 A, Experimental block size calibration for BP-DFS. B, Experimental block size calibration for the second kernel generation launched by CDP-BP. In the figure, block size vs processing rate (in 10^6 nodes/s)

TABLE 2 List of best parameters found experimentally for all parallel implementations

Implementation	Parameters settings Block Size	Bl. Size-CDP	d_{cpu}	d_{gpu}
$BP-DFS^a$	128	-	7	-
$CDP-BP_A$	128	64	6	8
$CDP-BP_Q$	128	128	5	7
$CDP-DP3^a$	128	32	-	-
$DP2^a$	128	32	5	7
$DP3^a$	128	32	-	-
$REC-CDP^a$	128	-	5	7
$REC-DP3$	128	-	-	-
Multicore[a]	-	-	4	-

[a] The parameters are the same for ATSP and N-Queens.

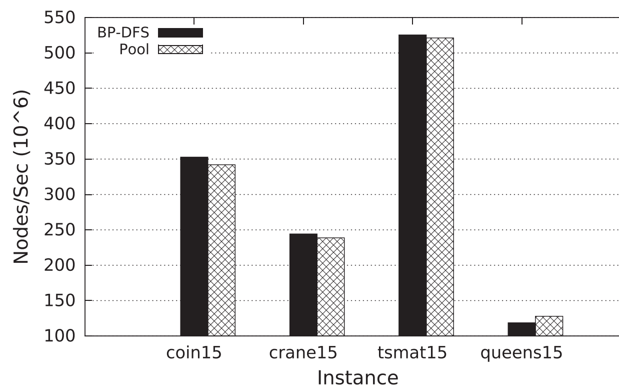


FIGURE 5 Comparison between the processing rate (in 10^6 nodes/second) of BP-DFS using pool scheme for load balance (*pool*) and BP-DFS. The version of BP-DFS with load balance is using 32 768 GPU threads. All other parameters for BP-DFS and *pool* are the ones presented in Table 2. Results are shown for instances of size $N = 15$

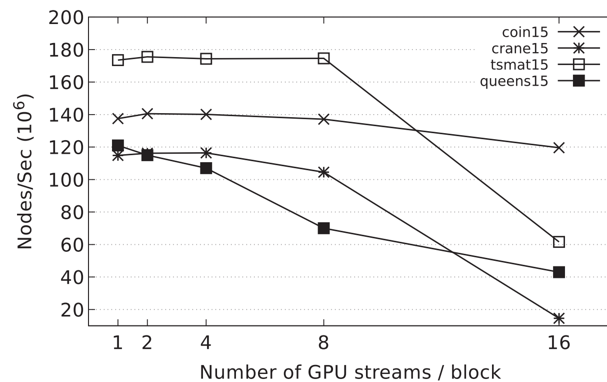


FIGURE 6 Influence of the number of streams created/kernel calls on the processing rate (in 10^6 nodes/s) for CDP-BP. The figure shows number of GPU streams/block vs processing rate (in 10^6 nodes/s)

Figure 5 shows the processing rate of BP-DFS using a pool scheme for load balance and BP-DFS without load balance. The results show that the use of load balance improves performance of BP-DFS only for the N-Queens problem.

DP2 and DP3 perform dynamic allocations on GPU. However, Plauth et al[17] do not provide information concerning the maximum GPU heap size or maximum depth of synchronization. Without setting up the variable `cudaLimitMallocHeapSize`, these CDP-based implementations can solve only instances of sizes up to $N = 11$. Therefore, to make a performance comparison, we set the value of `cudaLimitMallocHeapSize` to size of the available global memory. Concerning the choice of d_{cpu} and d_{gpu} for CDP-BP and DP2, best performance is reached when $d_{gpu} = d_{cpu} + 2$. All CDP-based implementations use the default size for the fixed pending grid queue.

Preliminary experiments show that it is not possible to use $r_{cdp} > p.memorySize$, as in Algorithm 3 (line 9). The reason is that CDP uses a lot of additional memory to handle the dynamically generated kernels. Using $r_{cdp} \geq (0.75 \times p.memorySize)$, CUDA returns an "out of memory" error. For avoiding the error, we set $r_{cdp} \geq (0.7 \times p.memorySize)$.

We have carried out experiments to verify whether it is worth running a multi-threaded Initial CPU Search. In this experiment, we run this search with 2 to 40 threads. For *tsmat*15 and $d_{cpu} = 7$, the initial tree corresponds to only 0.03% of the solution space. On the one hand, the Initial CPU Search takes 55 milliseconds. On the other hand, the multi-threaded Initial CPU Search is from 6.3× to 29× slower than its serial counterpart, depending on the number of threads the Initial CPU Search uses. This behavior is observed for all instances sizes and classes. The multi-threaded initial CPU search initializes threads, has mutual exclusive accesses, and function calls. Moreover, there is a reduction on the tree size when the search finishes. Even for the biggest instances (*tsmat*19) and the deepest cutoff depth ($d_{cpu} = 7$), the initial tree is less than 1% of the whole solution space. Therefore, it is not worth using multi-threading to explore such a small load.

We also run experiments to configure CDP-BP's variable *number_of_kernels* (Algorithm 6). It defines the number of streams created/kernel launches for each GPU block of the Intermediate GPU Search. In Figure 6, one can see the influence of the number of streams created/kernel calls on the processing rate for instances of size 15. According to Figure 6, using more than two streams per block brings no benefits. This behavior is observed in all test cases.

4.3 | Comparison between CDP-based implementations

In this section, all CDP-based implementations and the recursive ones are compared using the parameter configuration presented in Table 2. In Figure 7, one can see the average speedup reached by all CDP-based implementations and their recursive counterparts compared to the serial control baseline.

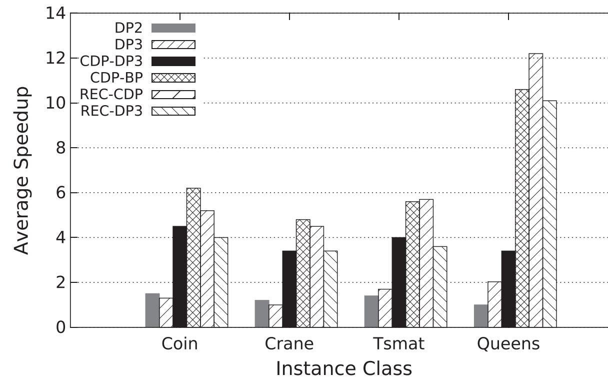


FIGURE 7 Average speedup reached by all CDP-based implementations and their recursive counterparts compared to the serial one. Results are considering all classes of instances. Problem sizes are ranging from $N = 10$ to 15

CDP-BP is the only implementation faster than the serial implementation for all experiments. Speedups observed for CDP-BP range from 2.5× (*crane*10) to 13× (*queens*12 − 14). CDP-BP is considerably faster than all other CDP-based implementations while solving small instances (sizes $N = 10 - 12$). In this situation, the overhead caused by dynamic allocations, streams initialization, and recursive kernel launches amount for the major part of the execution time. For small instances, CDP-BP is up to 20× faster than DP3 (*coin*10 and *crane*10), 13× faster than DP2 (*queens*12), and up to 6× (*tsmat*10) faster than CDP-DP3. As the tree size grows and the overhead becomes relatively less important, this difference decreases: for $N = 15$, CDP-BP is up to 2.5×, 9×, and 2× faster than DP3, DP2, and CDP-DP3, respectively.

With regard to DP2, speedups compared to the serial implementation are observed only for instances bigger than $N \geq 12$, and the highest speedup observed is 2.3× (*tsmat*15).

DP3 is superior to the serial implementation for $N \geq 13$. Speedups range from 1.38× (*tsmat*13) to 5.6× (*queens*15).

CDP-DP3 has the second best overall result, with speedups ranging from 1.2× (*tsmat*12) to 7.15× (*coin*14) for all sizes bigger than $N = 10$. This implementation is faster than DP3 for all test cases. CDP-DP3 launches the Intermediate GPU Search from depth $d_{cpu} = 2$ to $d_{gpu} = 4$, then deploys one DP3 for each block. The first dynamic allocation occurs for the second d_{gpu}. CDP-DP3 performs less dynamic allocations and launches less kernels than DP3. As the tree size grows, the benefits of a more regular load produced by DP3 strategy[17] is observed. For $N = 13$ and 14, CDP-DP3 has the performance close to CDP-BP's one. For $N = 15$ CDP-DP3 is slightly superior to CDP-BP for *coin* and *tsmat*.

The two best overall performances for CDP-BP and CDP-DP3 evidences that a smaller number of kernel launches and less dynamic allocations can lead to a higher nodes/second processing rate. This is the case also for the recursive implementations. REC-CDP is faster than REC-DP3, for all test cases. REC-CDP makes no dynamic allocation/deallocation before launching a new generation of kernels.

Despite the overhead of using CDP, both CDP-BP and CDP-DP3 are faster than their recursive counterparts for two instance classes, and both have an equivalent performance for one class. These results evidence that a smaller interference of the host combined with a block-based child search seems worthwhile for irregular tree search algorithms. However, it is not the case for *queens*: Using CDP is less efficient because the load processed by the child kernels is too small (refer to Figure 2). No CDP or recursive implementation is faster than the highly optimized bit-parallel N-Queens solver that applies symmetries and node evaluation at a constant time.

4.4 | Comparing CDP-BP to BP-DFS: Best and worst case analysis

To identify scenarios where the CDP implementation is advantageous compared to a non-CDP one, we use different values of d_{cpu}: 3, 4, 5, 6, and 7. For the CDP implementation, the second kernel is launched two levels deeper, as shown in Table 2. Therefore, the values used of d_{gpu} are 5, 6, 7, 8, and 9, respectively.

For instances of size 17, Table 3 shows the execution times (in seconds) obtained when selecting the best, respectively the worst value for d_{cpu}. It also shows the median (ie, the third best) execution time obtained for d_{cpu} from 3 to 7 and the relative standard deviation (RSD, defined as $\frac{standard \quad deviation}{average} \times 100\%$). In brackets, beneath these execution times the corresponding parameter d_{cpu} (or d_{cpu}-d_{gpu}) are shown. For comparison, Table 3 also shows the serial execution time in angled brackets beneath the instance name.

Considering the N-Queens problem with $N = 17$, CDP-BP reaches a speedup of 10× over the serial implementation even for the worst configuration ($d_{cpu} = 7$). In contrast, the worst configured BP-DFS is clearly outperformed by the serial implementation. Using its best configuration ($d_{cpu} = 4$), CDP-BP reaches a speedup of 13.8×. This is less than BP-DFS on its best configuration, which is 22× faster than the serial implementation.

Similar results are observed for the ATSP problem. For the ATSP instances of size $N = 17$, CDP-BP presents speedups over its sequential counterpart even for the worst-case parameter d_{cpu} (ranging from 1.9× (*crane*) to 6.3× (*tsmat*)). In contrast, for the worst-case configuration BP-DFS is outperformed by its sequential counterpart. However, if the best configuration is chosen for both algorithms, BP-DFS outperforms CDP-BP by a factor of ≈ 2.5×.

TABLE 3 Worst, best case and median execution times (in seconds), relative standard deviation (defined as 100%×(standard deviation)/(average)) for instances of size 17

Inst.	$T_{worst}(s)$			$T_{best}(s)$			$T_{median}(s)$		RSD(%)	
$< T_{seq} >$	BP-DFS	CDP-BP	Rate	BP-DFS	CDP-BP	Rate	BP-DFS	CDP-BP	BP-DFS	CDP-BP
queens17	3394	132	25	58	94	0.6	65	101	163	14
< 1295 >	(3)	(7-9)		(6)	(4-6)		(5)	(5-7)		
coin17	2632	106	25	16	34	0.4	110	38	153	52
< 311 >	(3)	(3-5)		(7)	(7-9)		(5)	(6-8)		
crane17	3115	208	15	31	71	0.4	206	76	185	59
< 395 >	(3)	(3-5)		(7)	(4-6)		(5)	(6-8)		
tsmat17	39850	1642	24	560	1441	0.4	889	1496	186	5
< 10423 >	(3)	(3-5)		(7)	(4-6)		(5)	(6-8)		

Below each execution time the corresponding configuration is shown (in brackets). The serial execution time is shown in angled brackets <>.

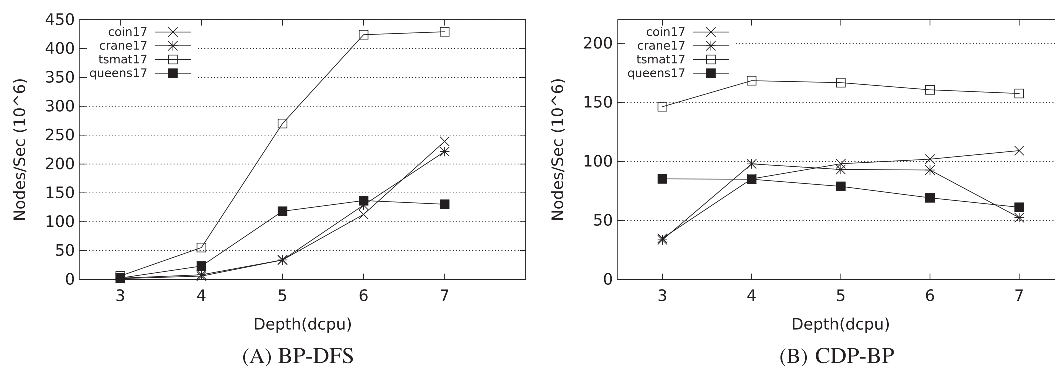


FIGURE 8 A, Influence of d_{cpu} on the processing rate for BP-DFS. B, Influence of d_{cpu} on the processing rate for CDP-BP. Processing rates are shown in 10^6 nodes/s for instances of size $N = 17$

For instance, if the worst parameter choice is made for BP-DFS and CDP-BP, the former spends 2632 seconds solving instance *coin*17, while the latter is about 25× faster, spending 106 seconds to perform the same task. On the other hand, if both versions use the respectively best parameters, BP-DFS solves *coin*17 in only 16 seconds, which is 2× faster than its CDP-based counterpart.

For all ATSP instances of different sizes, a similar behavior is observed. Thus, if well configured, BP-DFS provides the best overall performance. However, if poorly configured, it may lead to the worst performance. Such significant performance differences demonstrate the sensitivity of BP-DFS concerning the calibration of the parameter d_{cpu}. Figure 8A shows the influence of d_{cpu} choice on the processing rate for BP-DFS.

According to NVIDIA CUDA Profiler, BP-DFS uses the GPU resources poorly with $d_{cpu} = 3, 4$. For $d_{cpu} = 3$, the occupancy and multiprocessor activity reached by BP-DFS are around 4% and 6%, respectively. In turn, CDP-BP by launching a new generation of kernels reaches occupancy and multiprocessor activity 5× and 10× higher, respectively. With $d_{cpu} = 4$, CDP-BP reaches values of occupancy and multiprocessor activity compared to the values its non-CDP counterpart reaches only for $d_{cpu} = 7$. The number of dynamically deployed kernels grows along with d_{cpu} and the overhead involved in launching and managing these kernels tends to penalize the CPD-based implementation. Therefore, for $d_{cpu} = 6, 7$ (and 5, in some cases), BP-DFS outperforms CDP-BP by a factor of 2.0× or more.

Figure 8B shows the influence of d_{cpu} on the processing rate for CDP-BP. Even when using CDP, the obtained performance still depends strongly on the tuning of the CPU search depth, especially for the *coin* and *crane* instances. However, a comparison of Figure 8A and 8B shows that CDP-BP is less dependent on parameter tuning than BP-DFS. Indeed, between the best- and worst-case CDP performances for *coin17* and *crane17*, a speedup of more than 3.0 can be observed. In contrast, when solving *tsmat17* or *queens17*, the speedup which can be gained by optimally tuning the CDP-based algorithm is only 1.15 and 1.39, respectively, showing that the CDP algorithm's behavior depends not only on the active set size at depths d_{cpu} and d_{gpu} but also on the shape of the explored tree.

This comparison shows, on the one hand, that a well-tuned BP-DFS can be more than twice faster than its ideally configured CDP-based counterpart. On the other hand, it shows that the use of CDP allows to have a much better worst case execution time and to make the algorithm's performance less dependent on the tuning of the parameter d_{cpu}. This is confirmed by the obtained RSD, which is lower for all cases when CDP is used. We have also carried this same experiment on two other testbeds, one equipped with a Kepler GPU (Tesla K20c), another equipped with a Maxwell GPU (GTX 980). According to the results, the same behavior observed in Table 3 was observed for both testbeds.

As one can see in Figure 9, the multi-core implementation that uses a pool strategy for load balance is less dependent on parameter tuning and the shape of the tree. For the N-Queens problem, the variation of processing rates for d_{cpu} ranging from 4 to 7 is not significant. For the ATSP, when $d_{cpu} > 3$, the processing rate for all instance classes are close.

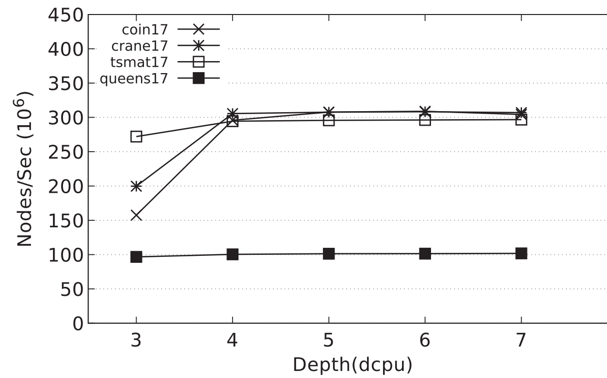


FIGURE 9 Influence of the d_{cpu} choice on the processing rate for the multi-core implementation. On the graphics, d_{cpu} vs processing rate (in 10^6 nodes/s). Results are for instances of size $N = 17$

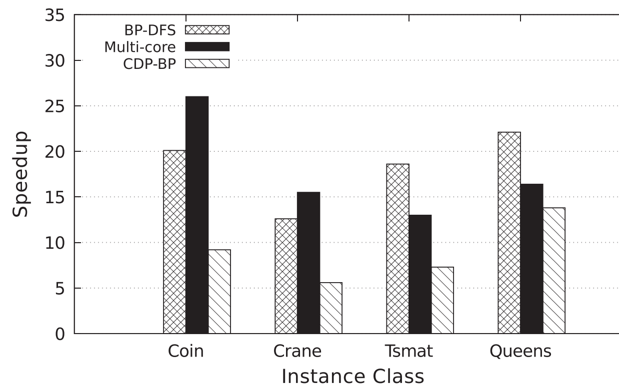


FIGURE 10 Speedup reached by BP-DFS, CDP-BP and multi-core implementations compared to the serial one. Results are for instances of size $N = 17$. The GPU-based implementations are using their best configuration shown in Table 3

Figure 10 presents the speedup reached by BP-DFS, CDP-BP and multi-core implementations. The performance of CDP-BP is usually inferior to both the BP-DFS and the multi-core implementations. Besides the overhead of launching/managing child kernels, CDP-BP presents several sources of overhead. Firstly, the Intermediate GPU Search uses atomic operations, performs error checking and block synchronizations. On the host side, CDP-BP *H2D* and *D2H* copies and allocations are bigger than BP-DFS' ones.

5 | DISCUSSION

Concerning the programmability, our results contrast with the results of Zhang et al,[22] where the use of CDP simplified the development of GPU-based graph algorithms. According to our experience, using CDP is challenging and brings complexity to the code. The programmer must learn extensions of the CUDA programming model, as introduced in Section 2.1. Furthermore, due to the characteristics of the problem solved, using CDP also requires additional efforts to handle increasing memory requirements.

Tracking device-side errors is difficult: in a situation where the maximum GPU heap size does not fit the requirements of the application, CUDA runtime returns an *"illegal memory access"* error. This error can be confused with an out of bounds memory access. Another situation is when the virtualized pool keeps a huge number of pending grids. If the program is using almost the whole global memory, and the queue run out of memory, the program may return incorrect results, and no errors are returned by the runtime error tracking on device and host side. Therefore, to cope with this situation, control data needs to be passed via global memory, which makes the code more complex and increases time spent in memory operations.

There are skeletons and frameworks designed to make GPU programming easier.[38] Using such frameworks to implement our search methods would not be worthwhile. It would be necessary to express our algorithms using the data parallel functions the framework provides. It would not be possible to modify some specific details, such as the trajectory of the search. Moreover, GPU skeletons/frameworks use dynamic data structures, like STL vectors. Such dynamic data structures perform very poorly on GPU, which is why alternative data structures such as bitsets are used for GPU-based algorithms. We can extend our work as a parallel backtracking skeleton for solving permutation-based combinatorial problems. The user would have only to provide the termination criteria, problem instance, and bounding functions.

CDP-BP presented the best overall performance among all studied CDP-based implementations, but it has been outperformed by its well-tuned non-CDP and multi-core counterparts. Concerning the applicability of CDP, it is useful in a situation where it is not possible to tune all parameters.

For example, in programs made available to nonexpert users. Moreover, a parameter configuration is not general enough for all classes of instances, and tuning a backtracking for solving a huge number of instances of different sizes and classes is prohibitive. In such situations, CDP is preferable, as it is less dependent on parameter tuning. As future work, we plan the development of a function that decides dynamically whether CDP or even the GPU should be used or not. Such a function can be based on the analysis of the partial backtracking tree[39] and properties of the underlying hardware.

According to the results, the ideas[17] of DP3 contribute to regularizing the workload processes on the GPU. Even with dynamic allocations and a recursive CDP kernel launches, CDP-DP3 achieves performance close to CDP-BP for $N = 13$ and 14, and slightly outperforms CDP-BP for $N = 15$. However, all DP3-based strategies cannot solve instances bigger than $N = 15$, even with the size of the heap set to the global memory available. The use of this kind of strategy needs more programming expertise and deep knowledge about the problem at hand, because the tuning of the maximum heap size is not straightforward. The allocation for the whole block happens if at least one thread finds a survivor node at depth d_{gpu}, and this space may not be entirely used. As the depth of the search increases, the memory requirements increase exponentially. We propose as a future work a different and more adaptive way of allocation and heap tuning, which takes into account the state of the tree or thread load individually.

Although it is intrinsically difficult to cope with fine-grained and irregular applications on GPUs, our results show that it is worth programming backtracking for GPUs. BP-DFS shows performance sometimes superior to a multi-core code that applies load balance and runs on two CPUs, 20 cores and 40 threads. Results also show that load balance strategies for GPU-based backtracking need to be more complex than a pool strategy, as the use of it did not bring benefits for BP-DFS.

5.1 | Main insights

In what follows, we summarize the main insights from our experimental evaluation of GPU-based backtracking algorithms using CDP:

- The use of CDP is preferable in situations where BP-DFS is not able to use the GPU resources properly;
- Using CDP-BP provides a better worst-case performance, and it is less dependent on good parameter settings than BP-DFS. However, if well-tuned, BP-DFS shows better results;
- Regarding programmability, the use of CDP may require additional expertise;
- The programmer needs to use extra control data, as errors on device side are difficult to detect;
- CDP has several sources of overhead, such as stream creation and destruction, and kernel launches. According to the results, avoiding a huge number of dynamic kernel launches may increase the processing rate of a CDP-based application.

6 | CONCLUSIONS AND FUTURE WORKS

We have presented a GPU-accelerated parallel backtracking strategy that uses CUDA Dynamic Parallelism (CDP). The proposed algorithm has been extensively tested using the N-Queens problem and instances of the Asymmetric Traveling Salesman Problem (ATSP) as test-cases.

Through a comprehensive experimental evaluation, we have shown that an unstructured tree search algorithm can take advantage of CDP in irregular scenarios, even for small instances. Also, CDP is preferable in situations where the Initial CPU Search does not generate enough load to the GPU. In this case, the second kernel produces more load, resulting in a better device utilization.

It is hard to tune a tree-search algorithm for running efficiently on GPU. The use of CDP provides a better worst-case performance and our CDP-based backtracking algorithm achieves speedup over its sequential counterpart even without prior parameter calibration.

However, all CDP-based implementations have been outperformed by a non-CDP bitset-based implementation (BP-DFS) with well-tuned parameters. Despite removing the overhead of dynamic allocations, CDP-BP still suffers from the cost imposed by dynamically launched kernels.

This research work has also identified challenges in developing CDP-based implementations. Programming efforts to deal with the growing memory requirements, difficulties in detecting device-side runtime errors, and the requirement of additional programming expertise are examples of challenges.

Another future research direction is to investigate the use of CDP for redesigning GPU-based Branch & Bound (B&B) search algorithms. 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. This class of unstructured tree search algorithm has the bounding operator very time-consuming, the opposite situation of the present work, which is dealing with fine-grained workloads.

ACKNOWLEDGMENTS

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

ORCID

Tiago Carneiro Pessoa http://orcid.org/0000-0002-6145-8352

REFERENCES

1. Burtscher M, Nasre R, Pingali K. A quantitative study of irregular programs on GPUs. In: IEEE International Symposium on Workload Characterization (IISWC); 2012; San Diego, CA, USA. 141-151.
2. Karypis G, Kumar V. Unstructured tree search on SIMD parallel computers. *IEEE Trans Parallel Distrib Syst*. 1994;5(10):1057-1072.
3. Yelick KA. Programming models for irregular applications. *ACM SIGPLAN Notices*. 1993;28(1):28-31.
4. Defour D, Marin M. Regularity versus load-balancing on GPU for treefix computations. *Procedia Comput Sci*. 2013;18:309-318.
5. Li D, Wu H, Becchi M. Nested parallelism on GPU: exploring parallelization templates for irregular loops and recursive computations. In: 44th International Conference on Parallel Processing (ICPP). Beijing, China: IEEE; 2015:979-988.
6. Karp RM, Zhang Y. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *J ACM (JACM)*. 1993;40(3):765-789.
7. Zhang W. Branch-and-bound search algorithms and their computational complexity, DTIC Document; 1996.
8. Zhang W, Korf RE. Depth-first vs. best-first search: New results. In: Proceedings of the National Conference on Artificial Intelligence. Washington, D.C., USA: John Wiley & Sons Ltd; 1993:769-769.
9. Jenkins J, Arkatkar I, Owens JD, Choudhary A, Samatova NF. Lessons learned from exploring the backtracking paradigm on the GPU. *Euro-par 2011 Parallel Processing*. Bordeaux, France: Springer; 2011:425-437.
10. 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); 2011; Vitória, ES, Brazil. 41-47.
11. Li L, Liu H, Wang H, Liu T, Li W. A parallel algorithm for game tree search using GPGPU. *IEEE Trans Parallel Distrib Syst*. 2015;26(8):2114-2127.
12. Feinbube F, Rabe B, von Lowis M, Polze A. Nqueens on CUDA: Optimization issues. In: Ninth International Symposium on Parallel and Distributed Computing (ISPDC). Innsbruck, Austria: IEEE; 2010:63-70.
13. NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110. NVIDIA Corporation Whitepaper v1.0; 2012.
14. Adinetz A. CUDA dynamic parallelism: API and principles. https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles/. 2014.
15. Yang Y, Zhou H. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. *ACM SIGPLAN Notices*, Vol. 49. New York, NY, USA: ACM; 2014:93-106.
16. Rocki K, Suda R. Parallel minimax tree searching on GPU. *Parallel Processing and Applied Mathematics*: Springer; 2009:449-456.
17. Plauth M, Feinbube F, Schlegel F, Polze A. A performance evaluation of dynamic parallelism for fine-grained, irregular workloads. *Int J Networking Comput*. 2016;6(2):212-229.
18. NVIDIA. *CUDA Dynamic Parallelism Programming Guide*. Santa Clara, CA, USA: NVIDIA Corporation; 2012.
19. NVIDIA. *CUDA C Programming Guide (Version 8.0)*. Santa Clara, CA, USA: NVIDIA Corporation; 2016.
20. Wang J, Rubin N, Sidelnik A, Yalamanchili S. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. *ACM SIGARCH Computer Architecture News*, Vol. 43. New York, NY, USA: ACM; 2015:528-540.
21. Wang J, Yalamanchili S. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In: 2014 IEEE International Symposium on Workload Characterization (IISWC). Raleigh, NC, USA: IEEE; 2014:51-60.
22. Zhang P, Holk E, Matty J, et al. Dynamic parallelism for simple and efficient GPU graph algorithms. In: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms. Austin, TX, USA: ACM; 2015:11.
23. Grama AY, Kumar V. A survey of parallel search algorithms for discrete optimization problems. *INFORMS J Comput*. 1995;7(4):365-385.
24. Carneiro T, Nobre RH, Negreiros M, de Campos GAL. Depth-first search versus Jurema search on GPU branch-and-bound algorithms: A case study. *In: NVIDIA's GPU Computing Developer Forum / XXXII Congresso da Sociedade Brasileira de Computação (CSBC)*. Curitiba,PR, Brazil: Sociedade Brasileira de Computação (SBC); 2012. http://www.imago.ufpr.br/csbc2012/anais_csbc/eventos/gpu/index.html
25. Carneiro T, Muritiba AEF, Negreiros M, de Campos GAL. Solving ATSP hard instances by new parallel branch and bound algorithm using GPGPU. *In: Iberian Latin American Congress on Computational Methods in Engineering (CILAMCE)*, Vol. 1. Ouro Preto, MG, Brazil: Associação Brasileira de Métodos Computacionais em Engenharia (ABMEC); 2011.
26. Krajecki M, Loiseau J, Alin F, Jaillet C. Many-core approaches to combinatorial problems: case of the Langford problem. *Supercomputing Frontiers and Innovations*. 2016;3(2):21-37.
27. Pessoa TCa, Gmys J, Melab N, de Carvalho Junior FH, Tuyttens D. A GPU-based backtracking algorithm for permutation combinatorial problems. *Algorithms and Architectures for Parallel Processing*. Granada, Spain: Springer International Publishing; 2016:310-324.
28. Pessoa TC, Gomes MJN. Jurema, a new branch & bound anytime algorithm for the asymmetric travelling salesman problem. *In: XLIII Simp Brasileiro de Pesquisa Operacional*. Bento Gonçalves, RS, Brazil: Sociedade Brasileira de Pesquisa Operacional (SOBRAPO); 2010.
29. Cirasella J, Johnson D, McGeoch L, Zhang W. The asymmetric traveling salesman problem: algorithms, instance generators, and tests. *Algorithm Eng Experimentation*. LNCS, 2001;2153:32-59. https://link.springer.com/chapter/10.1007/3-540-44808-X_3
30. Cook W. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton, New Jersey, USA: Princeton University Press; 2012.
31. Abramson B, Yung M. Divide and conquer under global constraints: a solution to the N-Queens problem. *J Parallel Distrib Comput*. 1989;6(3):649-662.
32. Zhang T, Shu W, Wu M-Y. Optimization of n-queens solvers on graphics processors. In: International Workshop on Advanced Parallel Processing Technologies. Shanghai, China: Springer; 2011:142-156.
33. Johnson D, Gutin G, McGeoch L, Yeo A, Zhang W, Zverovitch A. Experimental analysis of heuristics for the ATSP. *The Traveling Salesman Problem and its Variations*. Boston, MA: Springer; 2004:445-487.
34. Fischer T, Stützle T, Hoos H, Merz P. An analysis of the hardness of TSP instances for two high performance algorithms. In: Proceedings of the Sixth Metaheuristics International Conference; 2005; Vienna, Austria. 361-367.
35. Somers J. The N-Queens problem: A study in optimization. http://jsomers.com/nqueen_demo/nqueens.html. Accessed March 09, 2016.
36. Widmer S, Wodniok D, Weber N, Goesele M. Fast dynamic memory allocator for massively parallel architectures. In: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. Houston, TX, USA: ACM; 2013:120-126.

37. San Segundo P, Rossi C, Rodriguez-Losada D. *Recent Developments in Bit-Parallel Algorithms.* London, United Kingdom: INTECH Open Access Publisher; 2008.
38. Enmyren J, Kessler CW. Skepu: A multi-backend skeleton programming library for multi-gpu systems. Proceedings of the fourth international workshop on high-level parallel programming and applications, HLPP '10. New York, NY, USA: ACM; 2010:5-14, https://doi.org/10.1145/1863482.1863487.
39. Cornuéjols G, Karamanov M, Li Y. Early estimates of the size of branch-and-bound trees. *INFORMS J Comput*. 2006;18(1):86-96.

How to cite this article: Carneiro Pessoa T, Gmys J, de Carvalho Júnior FH, Melab N, Tuyttens D. GPU-accelerated backtracking using CUDA Dynamic Parallelism. *Concurrency Computat Pract Exper*. 2018;30:e4374. https://doi.org/10.1002/cpe.4374