Dynamic Configuration of CUDA Runtime Variables for CDP-based Divide-and-Conquer Algorithms

Tiago Carneiro, Jan Gmys, Nouredine Melab, Francisco Heron de Carvalho Junior, Pedro Pedrosa Rebouças Filho, Daniel Tuyttens

Dynamic Configuration of CUDA Runtime Variables for CDP-based Divide-and-Conquer Algorithms

Tiago Carneiro[1], Jan Gmys[2,4], Nouredine Melab[4], Francisco Heron de Carvalho Junior[3], P. P. Rebouças Filho[1], and Daniel Tuyttens[2]

[1] Instituto Federal de Educação, Ciência e Tecnologia do Ceará
{tiago.carneiro,pedrosa}@ppgcc.ifce.edu.br
[2] Mathematics and Operational Research Department (MARO), University of Mons, Belgium {jan.gmys,daniel.tuyttens}@umons.ac.be
[3] Programa de Mestrado e Doutorado em Ciência da Computação, Universidade Federal do Ceará, Brazil
heron@lia.ufc.br
[4] INRIA Lille Nord Europe, Université Lille 1, CNRS/CRIStAL, France
Nouredine.Melab@univ-lille1.fr

Abstract. CUDA Dynamic Parallelism (CDP) is an extension of the GPGPU programming model proposed to better address irregular applications and recursive patterns of computation. However, processing memory demanding problems by using CDP is not straightforward, because of its particular memory organization. This work presents an algorithm to deal with such an issue. It dynamically calculates and configures the CDP runtime variables and the GPU heap on the basis of an analysis of the partial backtracking tree. The proposed algorithm was implemented for solving permutation combinatorial problems and experimented on two test-cases: N-Queens and the Asymmetric Travelling Salesman Problem. The proposed algorithm allows different CDP-based backtracking from the literature to solve memory demanding problems, adaptively with respect to the number of recursive kernel generations and the presence of dynamic allocations on GPU.

Keywords: CUDA dynamic parallelism; Backtracking; Divide-and-conquer

1 Introduction

Irregular applications are present in different research fields, such as combinatorial optimization, data mining, and simulations [1]. The difficulty of parallelizing an application is closely related to its degree of irregularity [2]. Applications that present irregular control structure, irregular data structures, and irregular pattern of communication are notably difficult to parallelize [3]. Unstructured tree search methods for solving combinatorial problems, such as *backtracking* and *branch-and-bound*, are examples of such applications. These problem solver paradigms are present in many different areas, e.g., combinatorial optimization,

artificial intelligence, and operations research [4]. The program model usually applied to parallelize backtracking algorithms for GPUs, allied to characteristics of the problems commonly solved, results in fine-grained and irregular workloads, which is detrimental to the performance of the GPU.

Although GPUs suffer from performance degradation while processing irregular applications, they are still attractive accelerators. They are ubiquitous, energy efficient, and deliver a high price/GFLOP rate [5]. Furthermore, GPU programming interfaces and tools have become more flexible and expressive. Recent extensions to the general-purpose graphics processing unit (GPGPU) programming model, such as CUDA dynamic parallelism (CDP), can raise the expressiveness of the GPGPU programming model, making it possible to better address irregular applications and recursive patterns of computation [6], such as divide-and-conquer, used by backtracking algorithms.

Despite CDP's purpose of better coping with recursive computations, it is not straightforward to use this technology for processing memory demanding problems [7]. CDP presents several hardware limitations and a different memory organization [6, 8]: it is required to configure the CUDA runtime to reserve memory for synchronization between kernel generations. Moreover, in case of dynamic allocations on GPU, it is also necessary to configure the GPU heap size. Both GPU heap and memory for synchronization are not available for use. The related work on CDP does not address these issues.

This work presents an algorithm that dynamically calculates and configures the CDP runtime requirements and the GPU heap size. This calculus is based on an analysis of the partial backtracking tree. The proposed algorithm was implemented for solving permutation combinatorial problems, experimented on two test-cases: N-Queens and the Asymmetric Travelling Salesman Problem. The proposed algorithm allows different CDP-based backtracking from the literature to solve memory demanding problems, autonomously with respect to the number of recursive kernel generations and the presence of dynamic allocations on GPU.

The remainder of this paper is structured as follows. Section 2 brings background information and related works. Section 3 presents the proposed algorithm, and Section 4 brings the performance evaluation. Finally, conclusions and directions for further investigations are outlined in Section 5.

2 Background and Related Works

2.1 N-Queens and ATSP

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 (ATSP). Due to its relevance, the TSP is often used as a benchmark for novel problem-solving strategies [9].

The ATSP instances used in this work come from a generator that creates instances based on real-world situations [10]. Three classes of instances have

been selected: *crane*, modeling stacker crane operations; *coin*, modeling a person collecting money from pay phones in a grid-like city; and *tsmat*, consisting of asymmetric instances where the triangle inequality holds. Each class of instances has its own characteristics. Hence, two instances of the same size N may result in a different behavior for the same algorithm.

The N-Queens problem consists in placing N non-attacking queens on a $N \times N$ chessboard. It is also often used as a benchmark for new GPU-based backtracking strategies [11, 12]. We consider the version of N-Queens that consists in finding *all* feasible board configurations. N-Queens can be 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.

2.2 CUDA Dynamic Parallelism Programming Model

In the CDP terminology, the 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 *child*. The launch of a child grid is non-blocking, but the parent grid only finishes its execution after the termination of all child grids. Inside a block, different kernel launches are serialized. To avoid serialization of kernel launches, the programmer must link each kernel launch to a different stream [8]. Concerning the memory model, blocks of a child grid also have shared memory, and child threads also have local and register memories. A Child grid is not aware of its parent context data, and a parent thread should not pass to child threads pointers to its local or shared memories. Thus, the communication between parent and child is performed through global memory.

2.2.1 Related Works on CUDA Dynamic Parallelism

The parallelization of irregular applications using CDP has received little attention in the literature. Particularly, CDP has been used for processing graphs, clustering, simulations, and backtracking algorithms [1, 7, 11, 13, 14]. According to related works, CDP is beneficial for processing applications whose data are hierarchically arranged. In such situations, the use of CDP results in performance gains and a code closer to the high-level description of the algorithm [13, 14]. However, when these requirements are not met, using CDP may result in significant overheads and makes the code much more complex [7, 11].

2.3 GPU-accelerated Backtracking

Backtracking algorithms explore the solution space by dynamically building a tree in a depth-first order [15]. The algorithm iteratively generates and evaluates new nodes, where each child node is more restricted than its father node. If a child node leads to a feasible and valid solution, it is branched, and its child nodes are stored in the *Active Set*. Otherwise, the node in question is discarded, and the algorithm backtracks to an unbranched node in the Active Set. The search generates and evaluates nodes until the Active Set is empty.

GPU-based backtracking algorithms usually consist of two stages: backtracking on CPU until a cutoff depth d_{cpu} and parallel backtracking on GPU [7, 11, 12, 16, 17]. Algorithm 1 presents a pseudocode for the GPU-accelerated backtracking in question.

Initially, the algorithm gets the problem to be solved (*line* 1) and the properties of the GPU (*line* 2). Next, the variable d_{cpu} receives the initial cutoff depth (*line* 3). The cutoff depth d_{cpu} is a problem-dependent parameter, usually determined by manual tuning. For the ATSP, the cutoff depth d_{cpu} corresponds to all feasible and valid permutations with d_{cpu} cities. The initial backtracking on CPU (*line* 4) fills the active set A_{cpu} with all objective nodes found at d_{cpu}, as illustrated in Figure 1. In the present context, an objective node is a valid, feasible and incomplete solution (permutation) at d_{cpu}.

Before launching the backtracking on GPU, a subset $S \subseteq A_{cpu}$ of size $chunk \leq |A_{cpu}|$ is chosen (*line* 7). Next, the CPU updates A_{cpu} and transfers S to GPU's global memory (*lines* $8-11$). Then, the host configures and launches the kernel (*lines* $12-14$). In the kernel, each node in S represents a concurrent backtracking root R_i, $i \in \{0, ..., chunk-1\}$. Therefore, each thread Th_i explores a subset S_i of the solution space S concurrently. The kernel ends when all threads have finished the exploration of S. The kernel may be called several times until A_{cpu} is empty (*lines* $6-16$).

Algorithm 1: CPU-GPU parallel backtracking algorithm.

```
1  I ← get_problem()
2  p ← get_gpu_properties()
3  d_cpu ← get_cpu_cutoff_depth()
4  A_cpu ← generate_initial_active_set(d_cpu, I)
5  S ← ∅
6  while A_cpu is not empty do
7  |  S ← select_subset(A_cpu, p)
8  |  chunk ← |S|
9  |  A_cpu ← A_cpu \ S
10 |  allocate_data_on_gpu(chunk)
11 |  transfer_data_to_gpu(S, chunk)
12 |  nt ← get_block_size()
13 |  nb ← ⌈chunk/nt⌉
14 |  parallel_backtracking <<< nb, nt >>> (I, S, chunk, d_cpu)
15 |  synchronize_gpu_cpu_data()
16 end
```

2.3.1 CDP-based Backtracking Algorithms

Plaut *et al.* [11] propose two CDP-based backtracking for enumerating all feasible and unique solutions of the N-Queens, called DP2 and DP3. Both approaches launch recursively Algorithm 1 by using CDP. The strategy called DP2 is based on two depths: d_{cpu} and d_{gpu}. Each node in A_{cpu} (at depth d_{cpu}) is a root of a backtracking that searches for objective nodes at depth d_{gpu}. To store the objective nodes, the first thread of block b that finds an objective node allocates memory for the maximum number of objective nodes block b can find. This block-based active set will be further referred to as A^b_{gpu}. Then, a recursive new generation of kernels is launched by using CDP, searching from d_{gpu} to N, as

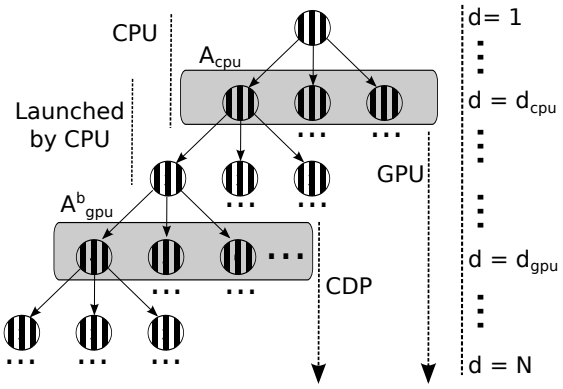

Fig. 1: Illustration of the DP2 strategy.

illustrated in Figure 1. In turn, DP3 doubles d_{gpu} at each new recursive kernel launch, until the search reaches the base depth of the recursion.

Results show that the overhead caused by dynamic allocations and dynamic kernel launches outweighs the benefits of the improved load balance yielded by CDP. Moreover, the performance of both algorithms strongly depends on the tuning of several parameters, such as block size and cutoff depth.

Applications that perform dynamic allocations on GPU and/or launch more than two kernel generations that synchronize require the configuration of the CUDA runtime, which is not straightforward [6, 8]. Under this scope, Carneiro Pessoa *et al.* [7] propose CDP-BP: a CDP-based backtracking that performs no dynamic allocations on GPU and avoids the need for dynamic setup of CUDA runtime variables. CDP-BP is also based on two depths, like DP2. The memory requirements of the application are dynamically calculated on the host, taking into account an analysis of the partial backtracking tree. The *host* allocates memory for further kernel generations and launches the search on GPU. Each GPU thread identifies its portion of the device-side active set based on thread-to-data mappings. Results show that CDP-BP has much better worst-case execution times and smaller dependence on parameters tuning than its non-CDP counterpart. Additionally, the authors also reported a difficulty in comparing CDP-BP to DP2 and DP3 using memory demanding instances, due to the complexity of dynamically calculating the upper bound on the required CUDA heap size.

3 The Proposed Algorithm

As previously pointed out, it is not straightforward to calculate an upper bound on the GPU heap size a CDP-based application needs. It is necessary to take into account the memory requirements of several CDP kernel generation until the search reaches the base depth. If the programmer does not configure the CUDA heap size before launching the first kernel generation, the GPU reserves a default memory space of 8 MB, which may be insufficient to store the objective

nodes found by several kernel generations. However, it is not possible to configure the GPU heap size equal to the GPU's global memory size, as the CDP runtime reserves memory for other purposes.

Up to 150 MB of global memory is reserved for each kernel generation that performs parent-child synchronization. This memory is used to keep track of the state of the parent grid and cannot be used by the programmer. Additionally, in case the application launches more than two kernel generations that perform synchronization, a CUDA runtime variable must be explicitly configured with such a number of generations to avoid runtime errors [8].

This section presents a new algorithm to calculate the memory requirements of the search, independently of the number of launched kernel generations. The proposed algorithm works on the host. The upper bound on the GPU heap size is dynamically calculated according to an analysis of the partial backtracking tree, by applying for different kernel generations the memory requirement analysis of [7]. Additionally, the proposed algorithm also configures the CUDA runtime accordingly, before the first kernel generation is launched.

The following sections provide a detailed description of the proposed algorithm. It consists of two main parts: *memory requirement analysis* and *launching the first kernel generation*. For the sake of greater simplicity, only the algorithm for solving instances of the ATSP to optimality is presented, which can be adapted for solving other permutation combinatorial problems with straightforward modifications.

3.1 Memory Requirement Analysis

As pointed out in Section 2.3, GPU-based backtracking performs an initial search on CPU to generate A^h_{cpu}. Before launching the first kernel generation, it is necessary to find a subset $S \subseteq A^h_{cpu}$ of size *chunk* for which its memory requirement fits the device limitations. The *Device* (GPU) and *Host* (CPU) data structures will be further distinguished by the superscripts d and h, respectively.

Backtracking algorithms that dynamically allocate memory on GPU's heap, such as DP3, need to store in global memory A^d_{cpu}, the cost matrix $C_{N\times N}$, and control data for the subsequent kernel generations. Moreover, it is necessary to reserve memory for parent-child synchronization and the heap. The memory requirement analysis consists of three steps: getting the number of kernel generations, heap size calculation, and calculation of the required global memory. These steps are detailed in the following sections.

3.1.1 Calculating the Number of Kernel Generations

It is necessary to know the number of kernel generations before launching the first one. This value is used to configure the runtime and to calculate the memory reserved by the GPU to keep track of context data of the parent grid.

One can see in Algorithm 2 the function that returns the number of kernel generations. Initially, the function receives the *intial_depth* of the search and the size N of the problem. Next, it gets the base of the recursion (*line* 1). Then, the

Algorithm 2: Calculating the number of kernel generations.

Input: The size N of the problem, and the initial cutoff depth *initial_depth*.
Output: Number of kernel generations that perform synchronization. Also including the one launched by the host.

```
1 base ← get_base_depth(N)
2 current_depth ← initial_depth
3 kernel_gen ← 1
4 while current_depth ≤ base_depth do
5 |   current_depth ← get_next_depth(current_depth, N)
6 |   kernel_gen ← kernel_gen + 1
7 end
```

number of generations that perform parent-child synchronization is calculated in lines 4 – 6. The programmer must provide two functions: *get_base_depth*() and *get_next_depth*(). The first one is responsible for calculating the base of the recursion (*line* 1). In turn, the second one is responsible for returning the next depth of the recursion (*line* 5), which works like an iterator.

3.1.2 Upper Bound on the Required GPU Heap Size

The heap size calculation is the main step of the memory requirement analysis. The strategy employed in this step takes into account the maximum number of children nodes that a node at *current* depth can have at *next* depth, which is:

$$expected\_children_{next} = \frac{max_{next}}{max_{current}}$$

where the maximum number of nodes of a given depth d is:

$$max_d = \frac{(N-1)!}{(N-d)!}$$

Algorithm 3 presents a function that estimates an upper bound on the required GPU heap size. This function receives as parameters *chunk*, which is the size of a subset $S \subseteq A^h_{cpu}$, and the size N of the problem. Then, it calculates, for each recursive kernel call until the search reaches the base depth, the memory required to store the upper bound on the number of objective nodes at depth *next* (*lines* 4 – 11). After getting the heap requirements for *chunk* nodes, it is possible to determine the amount of global memory required by the application.

3.1.3 Global Memory Required by the Application

Algorithm 4 returns the amount of global memory required by the application based on a subset $S \subseteq A^h_{cpu}$ of size *chunk*. Initially, the memory reserved for synchronization is calculated in line 1. As previously pointed out, a memory space of 150 MB is reserved for each kernel generation that performs parent-child synchronization. The amount of global memory required to store the control data, A^d_{cpu}, and the heap is calculated in lines 3 – 4. Finally, in line 6, the required global memory is calculated by adding the values got in lines 2 – 5.

Algorithm 3: Upper bound on the GPU heap size.

Input: The size $chunk$ of $S \subseteq A^h_{cpu}$, and the size N of the problem.
Output: Upper bound on the requested GPU heap size (in bytes).

```
1  ub_requested_heap ← sizeof(Node) × chunk
2  base ← get_base_depth(N)
3  current_depth ← get_initial_depth()
4  while current < base do
5  |   next_depth ← get_next_depth(current_depth)
6  |   max_current ← (N−1)! / (N−current)!
7  |   max_next ← (N−1)! / (N−next)!
8  |   expected_children_next ← max_next / max_current
9  |   ub_requested_heap ← ub_requested_heap × expected_children_next
10 |   current_depth ← next_depth
11 end
```

All algorithms presented in this section take into consideration a subset $S \subseteq A^h_{cpu}$ of size $chunk$. The next section shows how to choose S, to set up the CUDA runtime variables, and to launch the first kernel generation.

3.2 Launching the First Kernel Generation

Before launching the first kernel generation from the host, the application must find a subset $S \subseteq A^h_{cpu}$ of size $chunk$ such that an upper bound on its requirements fit into the global memory. Algorithm 5 shows how to get such a subset. Initially, the upper bound on the requirements of S is calculated in line 1, using Algorithm 4. If the memory required by S is bigger than the available global memory, $chunk$ is decreased until its requirements fit into the available global memory (*lines* 3 – 5). If there is no S such that its requirement fits into the available global memory, the program returns an error (*lines* 6 – 8).

Algorithm 6 presents the launching of the first kernel generation on GPU. After determining a suitable S (*line* 1), the CUDA runtime variables of heap size and number of kernel generations are set in line 2. All allocations on device memory are performed in line 3. There is no host allocation for other active sets than A^d_{cpu} because threads on device dynamically allocate memory on GPU's

Algorithm 4: Global memory required by the application.

Input: The size $chunk$ of $S \subseteq A^h_{cpu}$, the size N of the problem, and the number k of kernel generations that perform parent-child synchronization.
Output: The total of global memory required by the application (in bytes).

```
1  required_memory ← 0
2  nesting_memory ← k × 150MB
3  activeSet_memory ← chunk × sizeof(Node)
4  control_memory ← chunk × sizeof(ControlData)
5  required_heap ← get_heap(chunk, N)
6  required_memory ←
     required_heap + nesting_memory + activeSet_memory + control_memory
```

Algorithm 5: Calculating a suitable chunk size.

Input: $chunk$, the size N of the problem, and the number k of kernel generations.
Output: A suitable chunk size.

```
1  total_required ← required_memory(chunk, N, k)
2  available_memory ← get_GPU_properties(global_memory)
3  while total_required > available_memory do
4  |   chunk ← decrease_chunk(chunk)
5  |   total_required ← required_memory(chunk, N, k)
6  |   if chunk < 1 then
7  |   |   return error
8  |   end
9  end
```

heap. Finally, lines 7 – 18 process $S \subseteq A^h_{cpu}$ of size *chunk* until A^h_{cpu} is empty. After each kernel call, control data is retrieved (*line* 12) and the variables *counter* and *remaining* are updated (*lines* 13–17). The variable *counter* is used to make A^d_{cpu} point to unexplored nodes (*line* 10) and as termination criteria (*line* 7).

Algorithm 6: Launching the first kernel generation on GPU.

Input: Cost matrix $C^d_{N\times N}$,A^h_{cpu}, the global upper bound, and the size N of the problem.

```
1  chunk ← get_suitable_chunk(survivors_d_cpu, N)
2  set_CDP_variables(get_heap(chunk, N), get_num_gen(N, d_cpu))
3  device_memory_allocation(A^d_cpu, chunk, sizeof(Node), control_data^d)
4  survivors_d_cpu ← |A^h_cpu|
5  counter ← 0
6  remaining ← survivors_d_cpu
7  while counter < survivors_d_cpu do
8  |   nt ← get_block_size()
9  |   nb ← ⌈chunk/nt⌉
10 |   cudaMemCpy(A^d_cpu, (A^h_cpu + counter), chunk × sizeof(Node), H2D)
11 |   GPU_search <<< nb, nt >>> (C^d, chunk, A^d_cpu, control_data^d, upper_bound, N)
12 |   syncDataD2H(control_data^h, control_data^d, chunk)
13 |   counter ← counter + chunk
14 |   remaining ← remaining − chunk
15 |   if remaining < chunk then
16 |   |   chunk ← remaining
17 |   end
18 end
```

4 Performance Evaluation

The proposed algorithm was implemented to manage three recursive CDP-based backtracking from the literature:

- **DP2** and **DP3**: CDP-based algorithms introduced in Section 2.3.1.
- **CDP-DP3**: hybridization between CDP-BP and DP3 proposed by [7]. Compared to DP3, CDP-DP3 also doubles d_{cpu} until the search reaches the base depth. However, CDP-DP3 launches less CDP kernels than DP3, and dynamic allocations start on the second CDP kernel generation.

These GPU-based searches are launched in line 11 of Algorithm 6. For comparison, the following backtracking strategies are also considered.

- **BP-DFS**: non-CDP implementation of Algorithm 1 proposed by [17].
- **CDP-BP**: CDP-based algorithm introduced in Section 2.3.1 that makes no dynamic allocations on GPU and launches two kernel generations.
- **Multicore**: multi-threaded version of BP-DFS that applies a pool scheme for load balancing.
- **Serial**: serial control implementation optimized for single-core execution.

All implementations (but the serial one) use the data structures, the algorithm for consistency of the incumbent solution, and the kernel code of BP-DFS. For more details, refer to [7].

4.1 Experimental Protocol and Parameters Settings

All CUDA programs were parallelized using CUDA C 8.0 and compiled with GCC 5.4. The testbed, 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 an NVIDIA Tesla K40m (GK110B), 12 GB RAM, and 2880 CUDA cores @ 745 MHz.

In the experiments, ATSP instances of sizes (N) ranging from 10 to 19 are solved to optimality. In turn, N-Queens problems of sizes (N) ranging from 10 to 18 are also considered. The memory requirement of the test-cases ranges from few KB to several GB.

To compare the performance of two parallel backtracking algorithms, both should explore the *same* search space [18]. Therefore, for all ATSP instances, the initial upper bound is set to the optimal value. This initialization ensures that all implementations above outlined explore the same feasible region, which is always the case for the N-Queens problem.

The performance of GPU-accelerated backtracking algorithms strongly depends on the tuning of several parameters [7, 11]. Preliminary experiments were carried out to find out a suitable block size, d_{cpu} and d_{gpu} for all GPU-based implementations. Table 1 summarizes the best parameter configurations of all parallel implementations. The chosen parameters are the best for most of the instances, but not for all of them.

Both DP3 and CDP-DP3 have no cutoff depth tuning because they follow the strategy proposed by [11], which doubles d_{cpu}. For the N-Queens, there are N possibilities of starting node at depth d_{cpu}. This way, four kernel generations can be launched: at $d_{cpu} = 1$, $d_{gpu} = 2, 4$ and 8 (base). For the ATSP, there is only one possible starting city. This way, three kernel generations are launched.

4.2 Results

First of all, it is important to point out that the proposed algorithm is necessary to make DP2, DP3, and CDP-DP3 solve all test-cases without runtime

Table 1: List of best parameters found experimentally for all parallel implementations. The superscript Q and A indicate that the settings are for the N-Queens or, respectively, the ATSP implementation.

	Parameters Settings			
Implementation	Block Size	Bl. Size-CDP	d_{cpu}	d_{gpu}
BP-DFS[1]	128	-	7	-
CDP-BP$_A$	128	64	6	8
CDP-BP$_Q$	128	64	5	7
CDP-DP3[1]	128	64	-	-
DP2^A	128	64	4	7
DP2^Q	128	32	4	7
DP3^A	128	64	-	-
DP3^Q	128	32	-	-
Multicore[1]	-	-	4	-

[1)] Parameter are the same for ATSP and N-Queens.

errors. According to preliminary experiments, these implementations that make dynamic allocations on GPU cannot solve instances of size $N > 12$ using the default GPU heap size of 8 MB. The CUDA runtime returns an "*illegal memory access*" error in situations where the GPU heap requirement of the application is bigger than the configured one.

Table 2 reports the average speedup achieved by all parallel implementations for different ranges of problem sizes compared to the serial baseline. In turn, Figure 2 presents the average speedup reached by all parallel implementations compared to the serial baseline.

The lowest values of average speedup in Table 2 are observed for DP2, DP3, and CDP-DP3. These applications perform dynamic allocations/deallocations on GPU and launch a new kernel per GPU thread. The overhead of dynamic allocations, multiple streams creation/destruction, and several kernel launches amount negatively for small sizes. However, as the solution space grows, this overhead becomes less significant, and the benefits of a more regular load yielded by the DP3 algorithm are observed: as can be seen in Figure 2, CDP-DP3 is the CDP-based implementation with the best overall results for the ATSP.

The benefits of using the DP3 strategy would not be realized without the proposed algorithm. As previously said, DP2, DP3, and CDP-DP3 can only solve instances of size up to $N = 12$ without GPU heap configuration. According to

Table 2: Average speedup reached by all parallel implementations for different ranges of sizes compared to the serial baseline.

	Average Speedup			
Implementation	$10-12$	$13-15$	$16-19(18)$	*All sizes*
DP2	1.3×	3.26×	5.11×	2.99×
DP3	1.08×	5.20×	6.68×	4.20×
CDP-DP3	2.10×	7.89×	7.12×	5.72×
CDP-BP	5.24×	8.38×	7.72×	7.10×
BP-DFS	5.82×	15.86×	15.70×	12.37×
Multicore	4.63×	14.71×	16.00×	11.41×

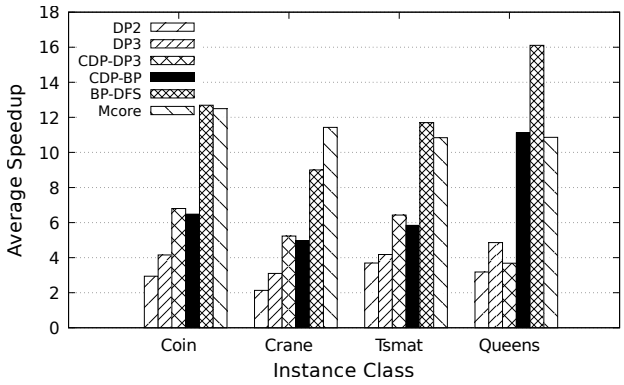

Fig. 2: Average speedup reached by all parallel implementations compared to the serial baseline.

Table 2, CDP-BP is 2.5× faster than CDP-DP3 and 4.8× faster than DP3 for sizes ranging from 10 to 12. These values change considerably for bigger problems, and CDP-DP3 becomes superior to CDP-BP for solving the ATSP, as previously pointed out. However, it is not the case for the N-Queens, as one can see in Figure 2.

Launching one new kernel for each GPU thread is not a good strategy for the N-Queens problem. The load processed by a child kernel is much smaller than the one launched when solving the ATSP [7]. This way, DP2, DP3, CDP-DP3 produce a significant overhead for processing small loads. In turn, CDP-BP launches a new kernel generation based on the load of the whole block, instead of the load of a single thread. According to NVIDIA Visual Profiler, by using the block-based kernel launch, CDP-BP can archive *twice* more occupancy and *three times* more eligible warps per active cycle than all other CDP-based implementations while enumerating all feasible configurations of the N-Queens. Thus, CDP-BP is much faster than its CDP-based counterparts and as faster as the multi-threaded implementation. The results of CDP-BP for the N-Queens and instances of sizes ranging from 10–12 justifies its better values than the ones of CDP-DP3 in Table 2.

BP-DFS is the fastest implementation when using its best configuration since it is highly optimized and does not suffer from CDP's intrinsic performance penalties. Moreover, BP-DFS is also superior to the multi-threaded implementation that applies load balance and runs on *two* CPUs, 20 cores / 40 threads.

4.3 Discussion

One of CDP's purposes is to better cope with divide-and-conquer applications. This is in part true for our implementations of both CDP-DP3 and DP3. The use of dynamic allocations removes the need for complex thread-to-data mapping. Furthermore, the strategy of doubling d_{gpu} makes almost the whole search to

be executed on GPU, which avoids d_{cpu}/d_{gpu} tuning. However, solving memory demanding problems requires extra programming efforts.

Using a different value than the default one makes it necessary to configure the CUDA heap beforehand, which brings complexity to the code. The heap size calculation takes into account a subtree rooted at d_{cpu} that goes down to the *base* depth. Moreover, the definition of a recursion base is also challenging: for an ATSP instance of size $N = 18$, 4 generations of kernels would be launched. Furthermore, it would require an enormous amount of memory to store the possible children nodes at $d_{cpu} = 16$ (refer to Section 3.1.2). However, the last kernel generation would perform no search at all. For example, consider instance $tsmat18$. The fourth kernel generation of DP3 would evaluate less than 3% of the solution space.

Moreover, the initial d_{cpu} for both DP3 and CDP-DP3 is 2. Taking into account the ATSP, it is unlikely that pruning of unfeasible nodes happens in such a shallow depth. As a consequence, the memory requirement analysis of both DP3 and CDP-DP3 would return a heap requirement close to the maximum possible size, which wastes memory. This is not the case for DP2: both d_{cpu} and d_{gpu} need tuning, and the number of objective nodes at deeper depths is much smaller than the maximum one. This way, DP2 can solve instance $crane15$ in a configuration with the default heap and a deep d_{cpu}. However, this configuration is slower than the one of Table 1 that needs GPU heap setup.

Finally, the memory for parent-child synchronization cannot be ignored, and it can be a limitation for more modest hardware: there may be a situation where the memory reserved for depth synchronization takes almost the whole global memory, leaving nearly no memory space for the search procedure on GPU.

5 Conclusion and Future Works

This work has presented an algorithm to calculate the memory requirements of CDP-based divide-and-conquer algorithms and configure the CUDA runtime accordingly. The proposed algorithm was implemented to manage different CDP-based backtracking from the literature and experimented on two test-cases: N-Queens and ATSP.

Using CDP may be a good choice for programmers that intent to parallelize recursive divide-and-conquer application for solving nondemanding problems. The use of recursion removes much of the complexity in programming and parameters tuning. Moreover, despite the intrinsic overhead of CDP, speedups are observed for all CDP-based implementations. However, using CDP for processing memory demanding recursive applications brings extra complexity on configuring the runtime before launching the first kernel generation.

A future research direction is on investigating different ways of calculating the next d_{gpu} and a rule for determining the base depth. Another future work is to investigate limitations of modest gamer hardware while processing memory demanding CDP-based recursive applications.

References

1. Wang, J., Yalamanchili, S.: Characterization and analysis of dynamic parallelism in unstructured GPU applications. In: 2014 IEEE International Symposium on Workload Characterization (IISWC), IEEE (2014) 51–60
2. Mukherjee, S.S., Sharma, S.D., Hill, M.D., Larus, J.R., Rogers, A., Saltz, J.: Efficient support for irregular applications on distributed-memory machines. In: ACM SIGPLAN Notices. Volume 30., ACM (1995) 68–79
3. Yelick, K.A.: Programming models for irregular applications. ACM SIGPLAN Notices **28**(1) (1993) 28–31
4. Gendron, B., Crainic, T.G.: Parallel branch-and-bound algorithms: Survey and synthesis. Operations Research **42**(6) (1994) 1042–1066
5. Brodtkorb, A., Dyken, C., Hagen, T., Hjelmervik, J., Storaasli, O.: State-of-the-art in heterogeneous computing. Scientific Programming **18**(1) (2010) 1–33
6. Adinetz, A.: CUDA dynamic parallelism: API and principles (2014) Accessed: 2018-05-10.
7. Carneiro Pessoa, T., Gmys, J., de Carvalho Junior, F.H., Melab, N., Tuyttens, D.: GPU-accelerated backtracking using CUDA dynamic parallelism. Concurrency and Computation: Practice and Experience (2017) e4374–n/a
8. NVIDIA: CUDA C programming guide (version 9.1). (2018)
9. Cook, W.: In pursuit of the traveling salesman: mathematics at the limits of computation. Princeton University Press (2012)
10. Cirasella, J., Johnson, D., McGeoch, L., Zhang, W.: The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. Algorithm Engineering and Experimentation (2001) 32–59
11. Plauth, M., Feinbube, F., Schlegel, F., Polze, A.: A performance evaluation of dynamic parallelism for fine-grained, irregular workloads. International Journal of Networking and Computing **6**(2) (2016) 212–229
12. Zhang, T., Shu, W., Wu, M.Y.: Optimization of N-Queens solvers on graphics processors. In: International Workshop on Advanced Parallel Processing Technologies, Springer (2011) 142–156
13. Zhang, P., Holk, E., Matty, J., Misurda, S., Zalewski, M., Chu, J., McMillan, S., Lumsdaine, A.: Dynamic parallelism for simple and efficient GPU graph algorithms. In: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, ACM (2015) 11
14. DiMarco, J., Taufer, M.: Performance impact of dynamic parallelism on different clustering algorithms and the new GPU architecture. In: Proceedings of SPIE Defense, Security, and Sensing Symposium. (2013)
15. Zhang, W.: Branch-and-bound search algorithms and their computational complexity. Technical report, DTIC Document (1996)
16. Feinbube, F., Rabe, B., von Löwis, M., Polze, A.: NQueens on CUDA: Optimization issues. In: Ninth International Symposium on Parallel and Distributed Computing (ISPDC), IEEE (2010) 63–70
17. 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) 41–47
18. Karypis, G., Kumar, V.: Unstructured tree search on SIMD parallel computers. IEEE Transactions on Parallel and Distributed Systems **5**(10) (1994) 1057–1072