UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE MESTRADO E DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

TIAGO CARNEIRO PESSOA

GPU-BASED BACKTRACKING STRATEGIES FOR SOLVING PERMUTATION COMBINATORIAL PROBLEMS

FORTALEZA

2017

TIAGO CARNEIRO PESSOA

GPU-BASED BACKTRACKING STRATEGIES FOR SOLVING PERMUTATION COMBINATORIAL PROBLEMS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação, da Universidade Federal do Ceará, como requisito para a obtenção do Título de Doutor em Ciência da Computação. Área de concentração: Computação de Alto Desempenho.

Orientador: Prof. Dr. Francisco Heron de Carvalho Junior

FORTALEZA

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P568g Pessoa, Tiago Carneiro.
GPU-based backtracking strategies for solving permutation combinatorial problems / Tiago Carneiro Pessoa. – 2017.
108 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2017.
Orientação: Prof. Dr. Francisco Heron de Carvalho Junior.

1. CUDA Dynamic Parallelism. 2. Device-side enqueue. 3. Backtracking paralelo. 4. Busca em profundidade. I. Título.

CDD 005

TIAGO CARNEIRO PESSOA

GPU-BASED BACKTRACKING STRATEGIES FOR SOLVING PERMUTATION COMBINATORIAL PROBLEMS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação, da Universidade Federal do Ceará, como requisito para a obtenção do Título de Doutor em Ciência da Computação. Área de concentração: Computação de Alto Desempenho.

Aprovada em 05 de Dezembro de 2017 .

BANCA EXAMINADORA

Prof. Dr. Francisco Heron de Carvalho Junior(Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Nouredine Melab
Université Lille 1 (UFR d'IEEA).
CNRS/CRIStAL - INRIA Lille Nord Europe.

Prof. Dr. Igor Machado Coelho
Universidade do Estado do Rio de Janeiro (UERJ)

Prof. Dr. Rafael Castro de Andrade
Universidade Federal do Ceará (UFC)

Prof. Dr. Albert Einstein Fernandes Muritiba
Universidade Federal do Ceará (UFC)

ACKNOWLEDGEMENTS

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

RESUMO

Novas extensões ao modelo de programação GPGPU, tais como o Paralelismo Dinâmico (*CUDA Dynamic Parallelism - (CDP)*), podem facilitar a programação para GPUs de padrões recursivos de computação, como o de divisão-e-conquista, utilizado por algoritmos *backtracking*. O presente trabalho propõe um novo algoritmo *backtracking* que utiliza CDP, baseado em um modelo paralelo para buscas não estruturadas. Diferentemente dos trabalhos da literatura, o algoritmo proposto não realiza alocação dinâmica em GPU. A memória requerida pela gerações de kernel subsequentes é previamente alocada de acordo com uma análise de uma árvore *backtracking* parcial. A Segunda parte desta tese generaliza as ideias do algoritmo inicial para abordagens que realizam alocação dinâmica em GPU e lançam mais que duas gerações de kernels. Essa generalização é necessária para que tais estratégias não apresentem erros em tempo de execução. A parte final desta tese investiga, no escopo dos algoritmos de busca não estruturada, se o uso de CDP é vantajoso ou não, comparando uma versão CDP e a versão equivalente que realiza várias chamadas do kernel através do host. Todos os algoritmos propostos foram extensamente validados utilizando o problema das N-Rainhas e o Problema do Caixeiro Viajante Assimétrico como casos de teste. A presente tese também identificou dificuldades, limitações e gargalos relacionadas ao modelo de programação CDP que podem ser úteis para ajudar potenciais usuários.

Palavras-chave: CUDA Dynamic Parallelism. Device-side enqueue. Backtracking paralelo. Busca em profundidade.

ABSTRACT

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. The initial part of this thesis proposes a GPU-accelerated backtracking algorithm using CDP that extends a well-known parallel backtracking model. Unlike related works from the literature, the proposed algorithm does not dynamically allocate memory on GPU. The memory required by the subsequent kernel generations is preallocated based on the analysis of a partial backtracking tree. The second part of this work generalizes the ideas of the first algorithm for approaches that dynamically allocate memory on GPU and launch more than two kernel generations. The final part of this thesis investigates whether the use of CDP is advantageous over a non-CDP and equivalent counterpart. All approaches have been extensively validated using the N-Queens Puzzle problem and instances of the Asymmetric Traveling Salesman Problem as test-cases. This thesis has also identified some difficulties, limitations, and bottlenecks concerning the CDP programming model which may be useful for helping potential users.

Keywords: CUDA Dynamic Parallelism. Device-side enqueue. Parallel Backtracking. Depth-first search.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS AND ACRONYMS

ATSP	Asymmetric Traveling Salesman Problem
B&B	Branch-and-Bound Search Strategy
BP	Bit-Parallel algorithm
CDP	CUDA Dynamic Parallelism
COP	Combinatorial Optimization Problem
CUDA	Compute Unified Device Architecture
D2H	Device-to-Host data transfer
GFLOP	10^9 Floating Point Operations per Second
GPU	Graphics Processing Unit
GPGPU	General-purpose computing on Graphics Processing Units
H2D	Host-to-Device data transfer
IVM	Integer-Vector-Matrix data structure
OpenCL	Open Computing Language
PC	Personal Computer
PCP	Permutation Combinatorial Problem
PCOP	Permutation Combinatorial Optimization Problem
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SMX	Next Generation Streaming Multiprocessor
TSP	Traveling Salesman Problem

CONTENTS

1 INTRODUCTION

Graphics Processing Units (GPUs) can be used to substantially accelerate many regular applications. In such applications, identical operations are performed on contiguous portions of data in a statically predictable manner (BURTSCHER; NASRE; PINGALI, 2012). In contrast, applications that are characterized by unpredictable and irregular control flow, degree of parallelism, memory access, and communication patterns are known as irregular or unstructured (KARYPIS; KUMAR, 1994; YELICK, 1993). 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* (GOLOMB; BAUMERT, 1965; BITNER; REINGOLD, 1975) falls into the class of irregular applications. Using GPUs for processing such applications is an emerging trend in GPU computing (DEFOUR; MARIN, 2013; LI; WU; BECCHI, 2015).

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 that can be explored in parallel (KARP; ZHANG, 1993). A search strategy defines which node of the tree will be processed next. Due to its low memory requirements, depth-first search (DFS) is often preferred (ZHANG, 1996). Also, its ability to quickly find new solutions increases the efficiency of the pruning process (ZHANG; KORF, 1993). While the pruning of branches reduces the size of the explored tree, it also makes its shape irregular and unpredictable, causing 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 (DEFOUR; MARIN, 2013). Thus, the implementation of such algorithms for GPUs is challenging.

GPU-based backtracking strategies have been efficiently used in regular scenarios, such as using DFS to perform a complete enumeration of the solutions space (JENKINS *et al.*, 2011; CARNEIRO *et al.*, 2011a; LI *et al.*, 2015). However, they face huge performance penalties in irregular ones, being outperformed even by their serial counterparts (FEINBUBE *et al.*, 2010).

1.1 Contextualization

Irregular applications are present in many fields of research: combinatorial optimization, data mining, social network analysis, simulation, etc. (WANG; YALAMANCHILI, 2014). The difficulty to parallelize an application is closely related to its degree of irregularity (MUKHERJEE *et al.*, 1995). There are three classes of irregularity, explained in the

next paragraphs (YELICK, 1993).

Irregular control structure is related to conditional structures that generate diverging control fluxes. Applications that belong to this class are difficult to be processed by *Singe Instruction, Multiple Data* (SIMD) architectures (FLYNN, 1972). Modern GPUs belong to the *Single Instruction, Multiple Threads* (SIMT) architecture, a more flexible version of SIMD where a group of threads execute concurrently the same instruction.

Irregular data structure appears in applications based on data structures such as unbalanced trees, sparse matrices, and graphs based on pointers. It is difficult to predict statically the processing load required by an irregular data structure. Thus, processing such structures usually require dynamic load balancing

Irregular pattern of communication appears in applications for which it is not possible to know in advance the order that communication events between processes. Irregular communication patterns are usually a consequence of processing either irregular data structures or control structures.

Applications that belong to the above three classes at the same time are the hardest to parallelize. Unstructured tree search applications for solving combinatorial problems, such as backtracking and *branch-and-bound* (LAWLER; WOOD, 1966), are examples of such applications. These problem solver paradigms are present in many different areas, such as combinatorial optimization, artificial intelligence, logic, and operations research (GRAMA; KUMAR, 1993; GENDRON; CRAINIC, 1994; ZHANG, 1996).

The focus of this thesis is on GPU-based backtracking algorithms. The program model usually applied for backtracking parallelization on GPUs, allied to characteristics of the problem, results in fine-grained and irregular workloads. In such fine-grained situations, there is no parallel node evaluation and the primary focus of the implementation is on the parallel search process.

Although GPUs suffer performance degradation while processing irregular applications, they are still an attractive alternative of an accelerator. They are ubiquitous, energy efficient, and deliver a high price/GFLOP rate. Furthermore, on each new GPU generation, the number of GPU cores doubles, and GPU programming interfaces and tools have become more flexible and expressive (DONGARRA *et al.*, 2007; FATAHALIAN; HOUSTON, 2008; BRODTKORB *et al.*, 2010; NVIDIA, 2016b; NVIDIA, 2016c).

Recent extensions to the general-purpose graphics processing unit (GPGPU) programming model, such as dynamic parallelism (DP), enables any GPU thread to launch dynamically new kernels without CPU interference (NVIDIA, 2012b). Dynamic parallelism may be useful for increasing the granularity in some critical regions of code, or

even for adapting the workload dynamically. This feature can raise the expressiveness of the GPGPU programming model, making it possible to better address irregular applications and recursive patterns of computation, such as divide-and-conquer and nested parallelism (ADINETZ, 2014; LI; WU; BECCHI, 2015; YANG; ZHOU, 2014). Although dynamic parallelism was first introduced as CUDA Dynamic Parallelism (CDP) (NVIDIA, 2012a), an extension of the CUDA programming model, it is also present in OpenCL 2.0 under the name of *device-side enqueue* (BOURD, 2017).

1.2 Objectives

According to the context presented in the previous section, it is possible to outline the objectives of this thesis.

1.2.1 Primary objective

The fundamental idea of the present work is to revisit a well-known programming model for GPU-based backtracking and propose new irregular tree search algorithms, based on recent enhancements to the GPU/CUDA programming model.

1.2.2 Secondary objectives

Besides the primary goal, this thesis has the following secondary objectives:

- To investigate the state-of-the-art on GPU-based irregular tree search algorithms according to their qualities and limitations.
- To study the use of dynamic parallelism through a large and irregular application.
- To verify whether and how much the use of CDP can improve the performance of a backtracking algorithm for solving permutation combinatorial problems in irregular scenarios.
- To investigate the effects of different parameters that influence the performance of a GPU-based backtracking algorithm.
- To identify difficulties, limitations, and bottlenecks of the CDP programming model.

1.3 Contributions

This thesis investigates several aspects involved in the development of GPU-based backtracking algorithms: search parameters, memory requirement, data structures, characteristics of the problem and the search algorithm. The focus of this thesis is on

solving permutation combinatorial problems. In this kind of combinatorial problems, a valid and complete solution is an N-sized permutation.

First, the present work revisits a well-known programming model for GPU-based backtracking, proposing a modification of the node representation: instead of using a vector of integers to keep track of the state of the search, the new data structure uses a bitset. The new data structure is applied as the foundation for all other parallel backtracking algorithms presented in this document.

The second and central part of this work proposes an extension of the referred well-known programming model using CDP, called CDP-BP. Differently from other related works, CDP-BP performs no dynamic allocation on GPU. Memory requirements and allocations on global memory are managed by the host to avoid runtime errors. The communication between parent and child grids is performed through global memory via a thread-to-data mapping.

Related CDP-based approaches give no details concerning the setup of the following CUDA runtime variables: the size of GPU heap and the number of kernel generations. Without setting up these runtime variables, algorithms that perform dynamic allocation and/or launch more than two kernel generations may present runtime errors. The third part of this work generalizes the ideas of CDP-BP for algorithms that dynamically allocate memory on GPU and launch more than two kernel generations.

CDP-based algorithms can be implemented without using CDP, returning the control to the host and then launching a new kernel generation. The final part of this thesis reports the findings of an investigation into whether the use of CDP is advantageous over a non-CDP and equivalent counterpart.

In short, this thesis presents the following main contributions:

- A CDP-based approach that dynamically deals with the memory requirements of the problem and avoids dynamic allocations on GPU. This strategy can achieve speedups greater than 20 compared to the control non-CDP counterpart under some conditions. The proposed strategy is also less dependent on parameter tuning.
- It shows that using CDP for processing a complex and demanding application, such as parallel backtracking for solving combinatorial problems, is not straightforward. The programmer must learn extensions of the CUDA programming model, and using CDP also requires additional efforts to handle increasing memory requirements.
- It proposes several approaches for coping with diverse hardware limitations that CDP has. This work has also identified several difficulties, limitations, and bottlenecks concerning the CDP programing model, which may be useful for helping potential users and motivate lines for further investigations.

- Finally, it shows that it is worth using GPUs for processing backtracking, even in irregular scenarios.

1.4 Organization

The remainder of this document is organized as follows. Chapter 2 introduces the background topics related to the present work, including the CUDA programming model, CUDA Dynamic Parallelism, permutation combinatorial problems, GPU-based backtracking strategies for solving combinatorial problems, and the related works. Chapter 3 presents a new GPU-accelerated backtracking algorithm using CDP that performs no dynamic allocation on global memory. Chapter 4 generalizes the ideas of Chapter 3 for search procedures that dynamically allocates memory on GPU's heap and launches various kernel generations. Chapter 5 presents an investigation into whether the use of CDP is advantageous over a non-CDP and equivalent counterpart. Finally, Chapter 6 brings the conclusion, which exposes the concluding remarks, future lines of research, and outlines the publications related to this thesis.

1.4.1 Presentation of the algorithms

This document presents several algorithms, which are mainly high-level descriptions of CUDA C programs. This way, some of them contain notations and well-known functions of the CUDA C programming language.

The GPU-based backtracking strategies herein proposed are for solving permutation combinatorial problems. For the sake of greater simplicity, only algorithms for solving instances of the ATSP to optimality are presented. These strategies can be adapted for solving other permutation combinatorial problems with straightforward modifications. For example, the algorithm for enumerating all feasible configurations of the N-Queens only differs in the node evaluation function.

2 BACKGROUND

This chapter contains background topics of this thesis. First, Section 2.1 briefly introduces the CUDA programming model, CUDA Dynamic Parallelism (CDP), and related works on dynamic parallelism. Section 2.3 presents the combinatorial problems used to validate the GPU-based algorithms proposed by this work. Section 2.4 is dedicated to GPU-based backtracking strategies for solving combinatorial problems and related works on this subject. In turn, Section 2.5 brings details concerning the control GPU-based implementation. Finally, Section 2.6 lists the concluding remarks of this chapter.

2.1 CUDA Programming Model

CUDA is a parallel computing platform and programming model designed to program and exploit NVIDIA's GPUs for general-purpose computations (NVIDIA, 2016a). In the CUDA programming model, the code that runs on CPU (*host*) launches a code to be executed by the GPU (*device*). The code to be processed by the device is called *kernel*. The working units of CUDA are *threads*, each one responsible for processing an instance of the kernel. The group of all threads that process the kernel forms a *grid*, where the threads are organized according to a linear, bi-dimensional or tri-dimensional topology.

The architecture of modern GPUs can be seen as a set of multiprocessors that share a memory interface. They are called SMX (*Next Generation Streaming Multiprocessor*). An SMX has a set of specialized cores, small banks of memory and a set of registers. These multiprocessors execute instructions in a *single instruction, multiple threads* (SIMT) manner, i.e., a group of threads executes the same instruction per clock cycle.

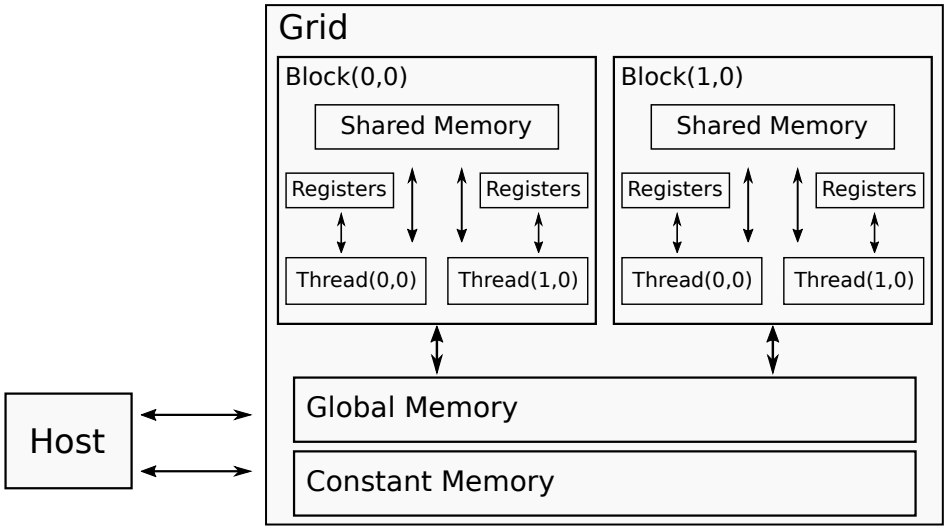
After the kernel launch, the GPU groups threads in *blocks* for execution. In turn, blocks are also divided into groups of threads that are mapped to the SMXs, called *warps*. In an SMX, the warp scheduler is responsible for issuing instructions from its warps, which execute threads in a SIMT fashion. If divergences of instructions occur inside a warp, the diverging threads execute their instructions separately, until the end of the branch, after which they join for continuing parallel execution. Threads divergence may cause severe performances degradation in a CUDA application (BURTSCHER; NASRE; PINGALI, 2012).

The CUDA programming model has a memory hierarchy with several levels, and the programmer must be aware of them to use the GPU resources properly. The most important levels are: *global*, *shared*, *register*, and *local* memories. The communication between the host and the device memory occurs through the global memory by using two types of operations:

1. *host to device* - *H2D*: copies from host's memory to device's global memory.

2. *device to host* - *D2H*: copies from device's global memory to host's memory.

Figure 1 – A summarized view of the CUDA programming model memory hierarchy. Arrows indicate how threads, blocks, and the host communicate through the memory hierarchy. The grid in question consists of two bi-dimensional blocks.



Source – Based on (NVIDIA, 2016a).

All threads being executed by the GPU have the same vision and can read/write into the global memory through pointers. The local memory is a region of the global memory that is exclusive to an individual thread. It keeps all thread's local information that the registers cannot store. Figure 1 shows a summarized view of the CUDA programming model memory hierarchy. Threads inside a block communicate via shared memory. This resource is limited, around 64 KB. Register and shared memory are the fastest because they reside inside the GPU chip. The number of threads being executed at the same time may be limited by the availability of these two memories.

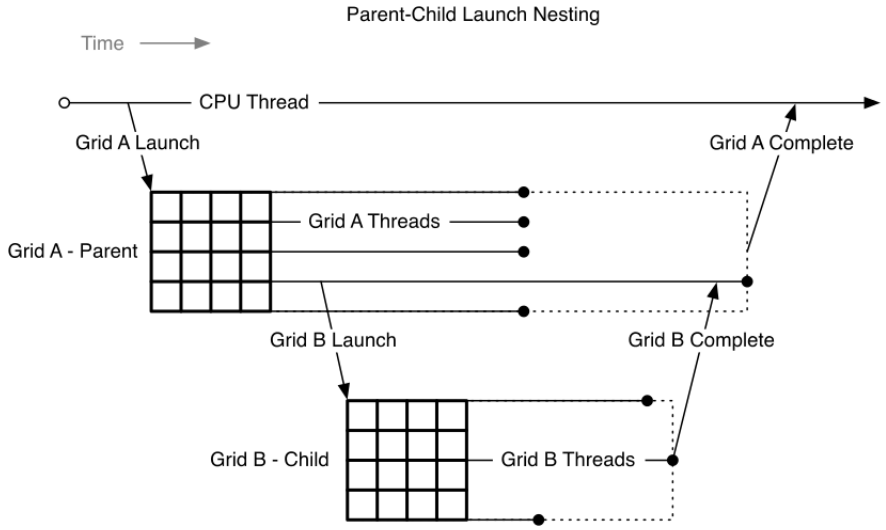
For more details concerning the CUDA programming model and other NVIDIA's GPU architectures, refer to (NVIDIA, 2016a; NVIDIA, 2016b; NVIDIA, 2016c).

2.2 CUDA Dynamic Parallelism

NVIDIA's Kepler architecture (NVIDIA, 2012b) has introduced CUDA Dynamic Parallelism (CDP), making it possible to launch new grids of threads without CPU interference. CDP may be useful for increasing the granularity in some critical regions of code, or even for adapting the workload dynamically. This feature can raise the expressiveness of the GPU programming model, making it possible to better address recursive patterns of computation, such as divide-and-conquer and nested parallelism (ADINETZ, 2014; LI; WU; BECCHI, 2015; YANG; ZHOU, 2014).

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 a *child*. The launching of a child grid is non-blocking, but the parent grid only finishes its execution after the termination of its child grid, as shown in Figure 2. If any synchronization between parent and child is required, CUDA synchronization functions such as `cudaDeviceSynchronize` must be applied (NVIDIA, 2012a; NVIDIA, 2016a). 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 link each kernel launch to a different stream.

Figure 2 – The execution of the parent grid only finishes after the termination of its child grids.



Source – (NVIDIA, 2012a)

Concerning the memory organization, blocks of a child kernel work like the blocks of a kernel launched by the host: they also have shared memory, and threads that belong to a child block also have register and local memories. 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, e.g., for saving parent grids states, and for keeping the pools of pending child grids. This memory is removed from the amount available to the program. A *pending child grid* is a grid that is suspended, being executed or waiting for execution (NVIDIA, 2016a). A GPU can hold more pending grids than the size of the fixed pool defined by the `cudaLimitDevRuntimePendingLaunchCount` variable. This is done by using the virtualized queue, and its management consumes memory, as mentioned above. Launching a huge amount of kernels is detrimental to the

performance of the program, because, when the fixed pool is full, the CUDA runtime uses the virtualized one. The GPU may not be able to handle a massive number of kernel launches, which may influence the correctness of the program. The main issue is that the programmer may not be aware of such problems, as tracking runtime errors on the device portion of the code are not trivial (ADINETZ, 2014).

2.2.1 OpenCL device-side enqueue

OpenCL is an open standard that provides a common API for heterogeneous programming (STONE; GOHARA; SHI, 2010). Diverse vendors such as IBM, ARM, AMD, Qualcomm, and Xilinx provide an OpenCL implementation for their devices. Dynamic parallelism is also present in OpenCL 2.0 under the name of *device-side enqueue*, and it is supported by AMD and Intel accelerators (MUKHERJEE *et al.*, 2015; IOFFE; SHARMA; STONER, 2015).

Device-side enqueue has a syntax close to CDP's one and uses the same parent-child terminology. Both dynamic parallelism approaches proceed the same way in what concerns the launching of a child kernel, the termination of the parent grid, the vision of the global memory by a child grid, and built-in means of synchronization (BOURD, 2017; IOFFE; SHARMA; STONER, 2015).

2.2.2 Related works on dynamic parallelism

Some works have applied CDP to develop programming abstractions for recursive computations and nested parallelism (YANG; ZHOU, 2014; WANG *et al.*, 2015; LI; WU; BECCHI, 2015). These works have concluded that the benefit of parallelizing nested loops by using CDP cannot outweight 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 by Wang *et al.* (2015).

CDP has also been used for processing irregular applications, such as graph algorithms, clustering and simulations (DIMARCO; TAUFER, 2013; ALANDOLI *et al.*, 2016). In particular, Dimarco and Taufer (2013) investigate the effects of CDP on the K-means and hierarchical clustering algorithms. Results shows that the use of CDP slowdowns the K-means algorithm. On the other hand, the hierarchical clustering has a tree-like data dependency, and the use of CDP results in speedups between 1.78 and 3.03. Alandoli *et al.* (2016) employ CDP to implement a clustering-based community detection algorithm. Results show that the non-CDP version is twice as faster as its CDP-based counterpart.

A strategy that launches new grids when a kernel finds a predetermined and regular load during its execution is proposed by Wang and Yalamanchili (2014). Results show speedups up to 2.73. However, 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 Zhang *et al.* (2015). 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.

CDP has also been applied in the scope of energy-efficient computing (MEHTA; ENGINEER, 2015; ODEN; KLENK; FRONING, 2014). Mehta *et al.* (2015) aims at using CDP to generate a more energy-efficient code for ARM devices. According to the results, it is possible to produce a code with less CPU-GPU synchronization by using CDP. This smaller host intervention results in an energy saving of 20%. In turn, Oden, Klenk, and Froning (2014) propose a GPU-controlled communication pattern where communication functions are kernels dynamically launched by the GPU. The CDP-based pattern results in a power saving of 10%. However, using CDP results in performance slowdowns.

Aliaga *et al.* (2016) studies the use of CDP to redesign a GPU-based conjugate gradient method for solving sparse linear systems. Results show that the CDP version outperforms its non-CDP counterpart, in both execution time and energy consumption. According to the results, the CDP version is on average 3.65% faster and 14.23% more energy efficient than its non-CDP counterpart. According to the authors, the benefits of using CDP depends on the granularity of the CUDA kernels. Regarding the programmability, the use of CDP leads to easier kernel implementations.

Jarząbek and Paweł (2017) apply CDP for solving three benchmark applications: heat distribution, numerical integration, and Goldbach conjecture. According to the experiments, the use of CDP is beneficial for applications that use hierarchically arranged data. In such situations, CDP can be applied naturally, which results in performance gains and programmability. Otherwise, the incorporation of CDP increases the complexity of the code.

Mukherjee *et al.* (2015) aims at exploring new features present in OpenCL 2.0, such as dynamic parallelism. The authors have redesigned a kernel for word recognition using device-side enqueue. Another contribution of Mukherjee *et al.* is that they have carried out experiments on an AMD Radeon GPU, which is an NVIDIA concurrent on the PC gamer market. According to the results, the use of device-side enqueue leads to a decrease in the overall throughput of the chosen application.

Intel has also provided a tutorial on using OpenCL device-side enqueue to implement a GPU-based Quicksort and a kernel to draw fractals, called Sierpinski Carpet. According to the results, the dynamic parallelism-based Quicksort is from 44% to 62% faster than its OpenCL 1.2 counterpart on Intel HD Graphics 5500 GPUs. Moreover, the authors also

state that device-side enqueue is syntactically close to CDP (IOFFE; SHARMA; STONER, 2015).

The documentation of AMD implementation of OpenCL 2.0 (AMD, 2016) presents a binary search on a vector using device-side enqueue as a case-study. This application was chosen because it returns the control to the host several times to call a new kernel. Results evidence that the binary search takes advantage of enqueuing on the device instead of returning the control to the host: The OpenCL 2.0 implementation can be more than 3 times faster than its OpenCL 1.2 counterpart for huge vectors.

2.3 Permutation Combinatorial Problems

A combinatorial problem is a problem to find a solution x, represented through discrete variables, in a set of solutions F. A combinatorial optimization problem seeks for a solution $x \in F$ such that $c(x) \leq c(y), \forall y \in F$ (minimization problem), where c is a cost function $c : F \rightarrow \mathbb{R}^1$ (PAPADIMITRIOU; STEIGLITZ, 1998).

This work focuses on permutation combinatorial problems, for which an N-sized permutation represents a valid and complete solution. Permutation combinatorial problems are used to model diverse real-world situations, such as workload scheduler for a processor, factory assembly lines, vehicle routing, and mathematical puzzles. Furthermore, their decision versions are usually NP-Complete. Permutation combinatorial problems have also been used as a test-case in several related works: (FEINBUBE *et al.*, 2010; CHAKROUN; BENDJOUDI; MELAB, 2011; LEROY, 2015; ZHANG; SHU; WU, 2011).

Algorithms for solving combinatorial optimization problems can be divided into exact (complete) or approximate strategies (BLUM; ROLI, 2003). Exact strategies guarantee to return an optimal solution for any instance of the problem in a finite amount of time. In the scope of the NP-hard problems, complete algorithms usually apply concepts of enumerative strategies and therefore have exponential worst-case execution times (WOEGINGER, 2003). In contrast, the approximate ones sacrifice the optimality and return a good and valid solution in reasonable time (ALBA *et al.*, 2005).

The backtracking is an exact strategy that performs implicit enumeration of the solution space (KARP; ZHANG, 1993). There are also some cases where the backtracking strategy halts when it finds a valid and complete solution, e.g., when solving a localization problem (GRAMA; KUMAR, 1993). The backtracking search strategy is also used as a component of branch-and-bound (B&B) algorithms. This kind of search strategy solves a relaxation of the problem at each iteration, making new subproblems more restrict than its parent node (LAWLER; WOOD, 1966; MITTEN, 1970). A B&B strategy that uses backtracking as a search component is usually referred as DFS-B&B, as the most recently generated node is always explored first (ZHANG, 1996; ZHANG, 2000).

The following introduces the two problems used to validate the GPU-based backtracking algorithms proposed in this work. The Asymmetric Traveling Salesman Problem (ATSP) and N-Queens are well known permutation combinatorial problems. Indeed, they are widely used in research works related to the contributions of this thesis.

2.3.1 The Asymmetric Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is an NP-hard problem that consists in finding the shortest Hamiltonian cycle(s) through a given number of cities. 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 (LAPORTE, 2006). Due to TSP's relevance, it is often used as a benchmark for novel problem-solving strategies (COOK, 2012).

The ATSP instances used in this work come from a generator that creates instances based on real-world situations (CIRASELLA *et al.*, 2001). 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. This thesis does not use instances from the well known TSPLIB (REINELT, 1991) library. These instances are too big to be solved to optimality by using backtracking without solving a relaxation of the problem on each iteration of the algorithm. For this purpose, B&B search algorithms (and its variations) are usually employed (FISCHETTI; LODI; TOTH, 2003; TURKENSTEEN *et al.*, 2008; GERMS; GOLDENGORIN; TURKENSTEEN, 2012).

2.3.2 The N-Queens Puzzle Problem

The N-Queens puzzle (ABRAMSON; YUNG, 1989) is the problem of placing N non-attacking queens on a $N \times N$ chessboard. The version of the N-Queens used in this work consists in finding *all* feasible board configurations. N-Queens is easily modeled as a permutation problem: position i of a permutation of size N designates the column in which a queen is placed in row j. 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, this work makes no use of them.

N-Queens is often used as a benchmark for new GPU-based backtracking strategies (ROCKI; SUDA, 2009; FEINBUBE *et al.*, 2010; ZHANG; SHU; WU, 2011; THOUTI; SATHE, 2012; PLAUTH *et al.*, 2016).

2.3.3 Comparison between ATSP and N-Queens

A first significant difference between the two problems is that each ATSP instance has its peculiarities (JOHNSON *et al.*, 2004; FISCHER *et al.*, 2005). 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. Moreover, the ATSP is an optimization problem, and there is the need to keep track of an incumbent solution. In contrast to the ATSP, N-Queens does not require a cost matrix, and solving the problem with size N means enumerating all feasible configurations for an $N \times N$ board. For this reason, there are several backtracking algorithms designed only to solve the N-Queens that explore specific properties of the board (SOMERS, 2002; BELL; STEVENS, 2009).

Solving the ATSP on GPUs by backtracking, performing implicit enumeration, is challenging (CARNEIRO *et al.*, 2011a; CARNEIRO *et al.*, 2011b; CARNEIRO *et al.*, 2012; PESSOA *et al.*, 2016). Node evaluation can be done in constant time and requires few arithmetic operations. In this fine-grained situation, there is no parallel node evaluation, and the primary focus of the implementation is in the parallel search process. For the N-Queens, the algorithm performs a diagonal check for conflicts before placing a queen in position i of the N-sized permutation. Therefore, the complexity of the node evaluation function for the N-Queens is $\mathcal{O}(N)$.

A valid solution for the ATSP always contains the starting city in the first position. For the N-Queens problem, there is no such requirement. This way, an exact algorithm for the ATSP must check up to $(N-1)!$ solutions, whereas the number of solutions for the N-Queens problem is $N!$. This characteristic also influences the upper bound on the solution space size, which is N times bigger for the N-Queens, taking into account an N-sized problem.

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, quadratic assignment, minimum linear arrangement problem, knight's tour problem, etc.

2.4 GPU-based backtracking strategies for solving combinatorial problems

This section presents a brief introduction to the backtracking tree search strategy and the most widely applied strategies for its parallelization. It also introduces the most used programming model for backtracking parallelization on GPUs, followed by related works on the literature.

2.4.1 Backtracking

Backtracking algorithmsBacktracking algorithms explore the solution space dynamically building a tree in depth-first fashion (GOLOMB; BAUMERT, 1965; BITNER; REINGOLD, 1975). The root node of this tree represents the initial problem to be solved, internal nodes are incomplete solutions, and leaves are solutions (valid or not).

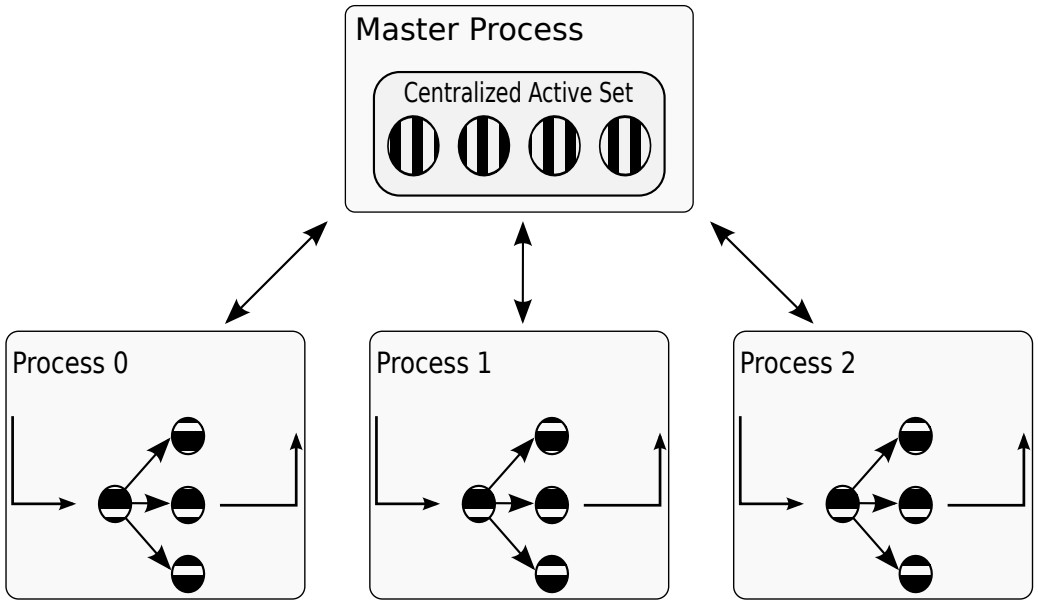
The algorithm iteratively generates and evaluates new nodes, where each child node is more restricted than its father node. Newly generated nodes are stored inside a data structure, conventionally a stack for depth-first search. At each iteration, the leftmost deepest node is removed from the data structure 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 or until a valid and complete solution is found (KARP; ZHANG, 1993; ZHANG, 2000).

2.4.2 General strategies for parallelization

There are several approaches to the parallelization of backtracking search strategies. One node-oriented parallel model consists of evaluating and expanding nodes in parallel (GENDRON; CRAINIC, 1994; CRAINIC; CUN; ROUCAIROL, 2006). This strategy usually has a master process which manages a central data structure that keeps the pending nodes. This data structure that stores nodes not yet branched is also called Active Set (ZHANG, 1996). The master process sends/receives nodes to/from slave processes, which are responsible for branching these nodes, i.e., for generating nodes more restricted than the node's father, and for evaluating these nodes, as shown in Figure 3. That is the reason why the node-oriented model is also called master/slave model. The degree of parallelism in the master/slave model is limited and strongly depends on the characteristics of the node evaluation function and the branching factor of a node. Parallel tree search algorithms and skeletons for shared memory and distributed memory systems usually apply the node-oriented programming model (DORTA *et al.*, 2003; CUN; ROUCAIROL, 1995; GALEA; CUN, 2007; DANELUTTO *et al.*, 2016).

Another approach, used in this work, consists in having multiple backtracking processes exploring different parts of the solution space independently. In the tree-oriented model, each process has its Active Set. The load balancing and coherency of the incumbent solution are performed through communication between processes, as shown in Figure 4. The degree of parallelism in this tree-based approach can be very high. However, it depends on the shape of the explored tree: splitting of the tree among processes leads to an imbalanced workload repartition, and the node representation makes challenging

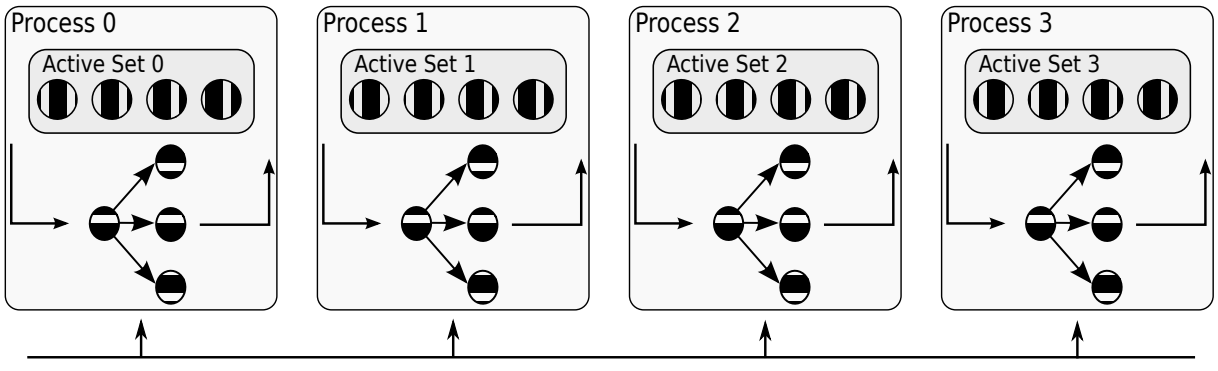
Figure 3 – Node-oriented (master/slave) programming model for parallel backtracking.



Source – Based on (CRAINIC; CUN; ROUCAIROL, 2006).

to share work among processes (GRAMA; KUMAR, 1993; KARYPIS; KUMAR, 1994; JENKINS *et al.*, 2011).

Figure 4 – Tree-oriented programming model for parallel backtracking.



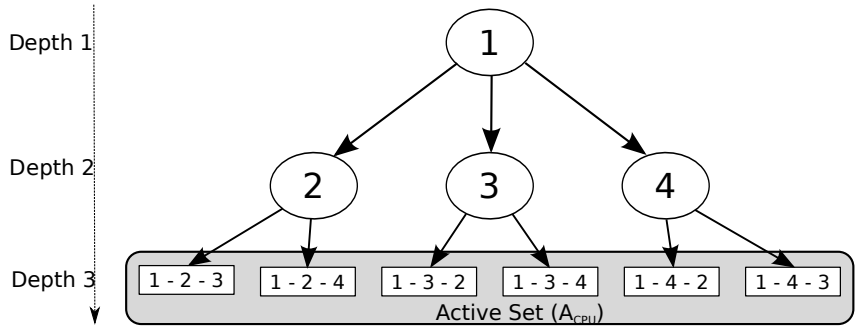
Source – Based on (CRAINIC; CUN; ROUCAIROL, 2006).

2.4.3 GPU-based strategies

GPU (ROCKI; SUDA, 2009; FEINBUBE *et al.*, 2010; CARNEIRO *et al.*, 2011a; CARNEIRO *et al.*, 2011b; CARNEIRO *et al.*, 2012; ZHANG; SHU; WU, 2011; LOPEZ-ORTIZ; SALINGER; SUDERMAN, 2013; LI *et al.*, 2015; PLAUTH *et al.*, 2016; KRAJECKI *et al.*, 2016). The initial backtracking on CPU performs DFS until a cutoff depth d_{cpu}. All objective nodes (valid, feasible and incomplete solutions at d_{cpu}) are stored in the *Active Set* A_{cpu}, which keeps all nodes evaluated but not yet branched.

The cutoff depth d_{cpu} is a problem-dependent parameter, determined ad-hoc or through manual tuning. For the N-Queens puzzle problem, introduced in Section 2.3, d_{cpu} corresponds to all valid configurations of the puzzle after placing d_{cpu} queens on the board. For the ATSP, nodes at depth d_{cpu} correspond to partial permutations with d_{cpu} cities. Figure 5 shows an illustration of the initial CPU backtracking whereas Algorithm 1 presents a pseudocode for the GPU-based backtracking in question.

Figure 5 – Initial CPU search for a generic permutation combinatorial problem of dimension $N = 4$ and $d_{cpu} = 3$.



Source – The author.

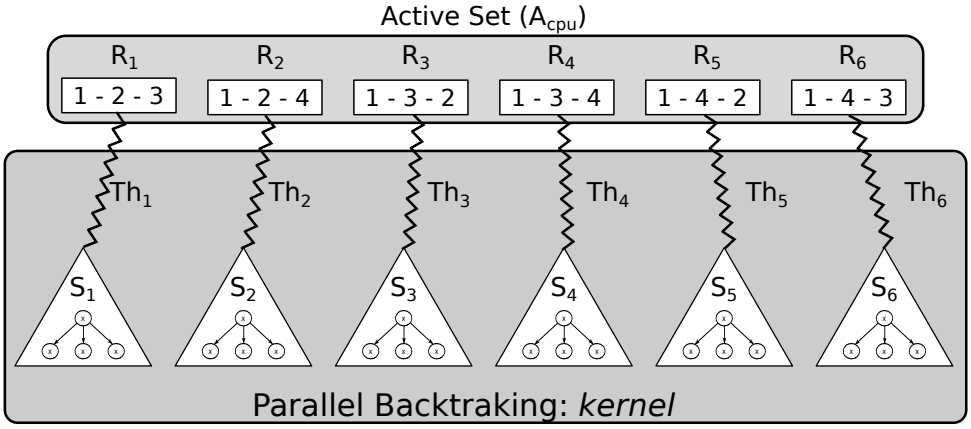
Algorithm 1: CPU-GPU parallel backtracking algorithm.

```
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
```

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*). 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 the properties p of the GPU. Next, the host (CPU) updates A_{cpu}, 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, as illustrated by Figure 6. 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.

Figure 6 – Each node in S represents a concurrent backtracking root R_i. In the kernel, each thread Th_i explores a subset S_i of the solutions space that has R_i as the root.



Source – The author.

The described GPU backtracking strategy performs well in regular scenarios (ZHANG; SHU; WU, 2011; CARNEIRO *et al.*, 2012; LI *et al.*, 2015; JENKINS *et al.*, 2011), but it faces a decrease of performance in more irregular ones, being outperformed even by the serial CPU implementation in some situations (FEINBUBE *et al.*, 2010). The problems commonly solved by the referred backtracking strategy usually produce fine-grained and irregular workloads. In such scenarios, the GPU suffers from load imbalance, uncoalesced memory accesses, and diverging instruction flow. For this reason, to achieve a proper utilization of the multiprocessors, this parallel backtracking strategy must launch a huge amount of GPU threads (JENKINS *et al.*, 2011). Therefore, besides the characteristics of the problem at hand, the performance of a GPU-based algorithm also depends on the tuning of several parameters, such as d_{cpu}, block size, and whether to use or not different levels of the memory hierarchy (ZHANG; SHU; WU, 2011; PLAUTH *et al.*, 2016).

2.4.4 Related GPU-Based Backtracking Strategies

Sommers (2002) presents a backtracking strategy to enumerate all valid solutions of the N-Queens problem. Such an algorithm is highly optimized for execution on one core of the CPU. It uses bit-sets to keep track of the position of the queens and to be aware of diagonal conflicts and employs instruction-level parallelism to speedup set operations. The use of bitsets allied to the instruction level parallelism results in node evaluation in

constant time. Moreover, this algorithm uses symmetry properties of the board to reduce the size of the solution space. Sommer's algorithm is also considered as a CPU-baseline for several works related to this thesis.

Feinbube *et al.* (2010) present a GPU-based version of the well-known bit-parallel code of Sommers (2002). This GPU-based version calculates all valid and unique configurations for a given number of rows. In this sense, the number of rows means the cutoff depth d_{cpu}. Feinbube *et al.*, rather than performing a CPU *vs.* GPU comparison, have presented several code optimization strategies. The programming model based on cutoff depth is also applied by Zhang, Shu, and Wu (2011) for enumerating all valid solutions of the N-Queens problem. This work presents diverse memory access optimizations and also states the difficulty of programming irregular applications for GPUs. Thouti and Sathe (2012) investigate the implementation of a program for enumerating all valid solutions of the N-Queens problem by using OpenCL (STONE; GOHARA; SHI, 2010), rather than the CUDA C language.

GPU-based algorithms that apply the tree-based strategy with cutoff depth d_{cpu} are commonly used to solve game tree problems, such as the minimax tree search (ROCKI; SUDA, 2009; LI *et al.*, 2015). Both related works state that the GPU-based game tree search is dozens of times faster than their serial counterparts in regular scenarios. However, the GPU-based versions suffer from warp divergence in irregular scenarios. In such a situation, both works pointed out that the speedups obtained are around 10× smaller. Strnad and Guid (2011) present a variation of the algorithm of Rocki and Suda (2009) for solving zero-sum board games. Speedups ranging from 1.2 to 23 are observed for boards bigger than 32×32.

Carneiro *et al.* (2011a) propose a GPU-based backtracking strategy for solving permutation combinatorial problems that follows the programming model introduced in Section 2.4.3. Carneiro *et al.* (2011b) extend the algorithm of Carneiro *et al.* (2011a) for solving to optimality instances of the ATSP. Both works report that these GPU-based approaches are much faster than their multi-core counterparts in regular scenarios. In irregular scenarios, the performance depends strongly on the shape of the tree and the tuning of many parameters. Carneiro *et al.* (2012) present a GPU-based version of the Jurema Search Strategy (PESSOA; GOMES, 2010) and compare it to the backtracking strategy of Carneiro *et al.* (2011a) using instances of the ATSP as benchmark. According to the results, both approaches suffer from the same problems: load imbalance and warp divergence. Even in the irregular scenarios used as benchmark, both GPU implementations are faster than their multi-core counterparts.

In the tree-based model usually applied by the GPU-based strategies listed above, the node representation makes it difficult to share work among processes. In this scope, Jenkins *et al.* (2011) propose a CPU-GPU stack-splitting strategy that does not apply

the tree-based programming model in question. In this algorithm, each warp has its stack, and the CPU does the load-balancing after each iteration of the algorithm. However, due to the irregular and fine-grained nature of the problem solved, this strategy cannot obtain high speedups compared to its serial counterpart, reaching speedups up to 2.25. Karypis and Kumar (1994) introduce a trigger mechanism that halts the kernel and redistributes the load among the processors if some unbalance rule is met. Although this strategy was mainly designed for SIMD architectures, it can be redesigned for modern GPUs (PESSOA *et al.*, 2016).

The GPU-based works above cited are CPU-GPU approaches in the sense that there is a search on the CPU until a cutoff condition happens. Differently, Krajecki *et al.* (2016) and Lópes-Ortiz, Salinger, and Suderman (2013) exploit the CPU to perform the final search along with the GPU. Krajecki *et al.* (2016) propose a multi-GPU and distributed backtracking for solving the Langford Problem (SIMPSON, 1983). Initially, the search strategy starts just like in Algorithm 1, generating the initial load. Then, tasks are distributed among multi-GPU nodes in a client-server manner. Lópes-Ortiz, Salinger, and Suderman (2013) present a generic CPU-GPU approach for divide-and-conquer algorithms. Moreover, the referred work also proposes cutoff conditions and load distribution rules.

There are also algorithms designed to tackle the workload imbalance of GPU-based DFS-B&B algorithms for solving permutation combinatorial problems (GMYS *et al.*, 2016). In this approach, the load balancing is performed without intervention of the host, inside a work stealing phase, which is invoked after each branching phase. The referred algorithm uses a data structure dedicated to permutation combinatorial optimization problems called Integer-Vector-Matrix (IVM) (MEZMAZ *et al.*, 2014).

2.4.4.1 Related CDP-based backtracking strategies

Plaut *et al.* (2016) proposes three CDP-based strategies based on Sommers (2002) algorithm for enumerating all valid solutions of the N-Queens problem. These approaches are called DP1, DP2, and DP3.

DP1 is equivalent to a parallel breadth-first search. 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 it evaluates the solution space completely.

The strategy called DP2 follows the model introduced in Section 2.4.3, and it 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 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 by using CDP, searching from d_{gpu} to N.

Finally, DP3 applies the concepts of DP2. However, DP3 doubles d_{gpu} at each new recursive kernel launch, until the search reaches the base depth of the recursion. The strategy of doubling d_{gpu}, adapting the launch of a new kernel generation to the shape of the tree, is closely related to the CPU-GPU approach of load balance proposed by Jenkins *et al.* (2011).

Results show that DP3 is superior to DP2, as it produces a more regular load to the GPU. Results also report that all proposed CDP-based implementations cannot outperform the control non-CDP counterpart. The overhead caused by dynamic memory allocations and dynamic kernel launches outweighs the benefits of the improved load balance yielded by CDP. Moreover, the performance of all proposed CDP-based algorithms for solving the N-Queens strongly depends on the tuning of several parameters, such as cutoff depth and block size.

2.5 Data structure and Search strategy employed

This section introduces the data structure and the backtracking search procedure applied by all GPU-based algorithms presented in this thesis. As pointed out in Section 1.4.1, the present section details the control backtracking implementation for solving the ATSP.

2.5.1 Control GPU-based implementation: BP-DFS

The control GPU-based implementation is an improvement of the GPU-based strategy proposed by Carneiro *et al.* (2011a) for solving permutation combinatorial problems, further referred as GPU-DFS. In addition to several code improvements, the main difference between the control implementation and GPU-DFS lies on the data structure that represents the current state of the search, called `Node`. GPU-DFS' `Node` contains a vector of integers (4 bytes) of size d_{cpu}, identified by *cycle*. This vector stores a feasible, valid and incomplete solution for the ATSP. Moreover, the data structure `Node` also contains a vector of integers (4 bytes) of size N, identified by *visited*, and an integer variable, identified by *cost*, which stores the cost of the partial solution at hand. The search keeps track of visited cities by setting $visited[n]$ to 1 each time the salesman visits the n-th city.

Backtracking algorithm may use bitsets to accelerate set operations. Algorithms that use this kind of instruction level parallelism are often called bit-parallel (BP) algorithms (SEGUNDO; ROSSI; RODRIGUEZ-LOSADA, 2008). Differently from GPU-DFS, the control implementation employs the bitset data structure to keep track of the visited cities. Moreover, this new data structure also aims at reducing the amount of memory required by a thread and the number of accesses to the local memory of the GPU. The bitset-based implementation of GPU-DFS will be further referred as *BP-DFS*.

The `Node` data structure of BP-DFS contains a vector of `char` of size d_{cpu}, identified

by *cycle*, and two integer variables. The vector *cycle* stores a feasible, valid, and incomplete solution. In turn, the first integer variable keeps the cost of this partial solution at hand. 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.

The data structure `Node` is similar to any permutation combinatorial problem. Taking into consideration the N-Queens problem, it does not contain the variable *cost*. Also, the vector *cycle* is the vector *board*, which keeps a valid configuration of the chess board.

2.5.1.1 Search procedure

BP-DFS is a direct implementation of Algorithm 1. The initial CPU search performs a backtracking from depth 1 until the cutoff depth d_{cpu}, storing all frontier nodes in A_{cpu}. Once the initial search is finished, the host transfers A_{cpu} (or a subset of it) to the GPU. After each kernel execution, there is a synchronization of data between the CPU and the GPU that retrieves information such as the number of solutions found, tree size, and best solution found.

The search strategy applied by BP-DFS is a non-recursive backtracking that does not use dynamic data structures, such as stacks. The semantics of a stack is 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 cost of this solution is compared to the value of the incumbent one. In the case the cost of the incomplete solution at hand is smaller than the cost of the incumbent one, *depth* is incremented, and the search proceeds to the next depth. After trying all configurations for a given depth, the search backtracks to the previous one.

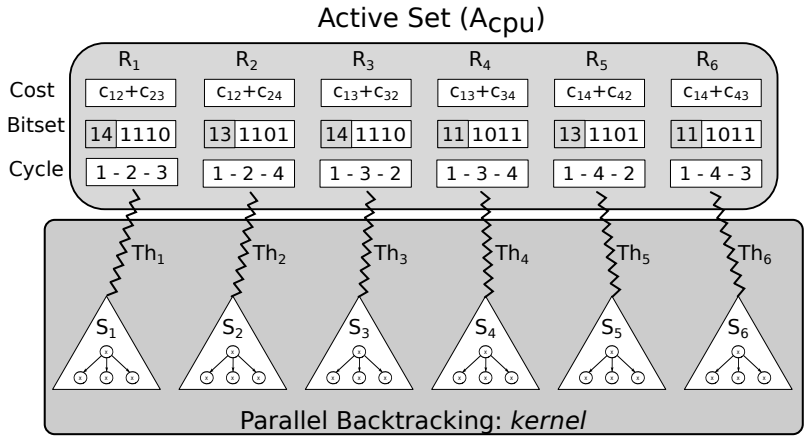
For all GPU-based implementations further proposed by this work, the search strategy is an instance of the kernel. Referring to Figure 7, thread $Th_i, i \in \{0, 1, ..., |A_{cpu}| - 1\}$ starts its data with root R_i, of type `Node`. By using the informations of R_i, thread Th_i implicitly enumerates the solutions space S_i.

2.6 Concluding Remarks

This chapter has presented the background topics related to this thesis. Based on the previous sections, it is possible to draw the following conclusions.

Dynamic parallelism is an extension of the GPGPU programming model to better cope with nesting parallelism and divide-and-conquer patterns of computation. Although DP was first introduced as CDP, it is also present in OpenCL 2.0 and supported by devices of two rivals of NVIDIA in the market: Intel and AMD. DP is beneficial for processing applications whose data are arranged in a hierarchical way and programs that return the

Figure 7 – Illustration of A_{cpu} for BP-DFS. This active set is generated by the initial backtracking on CPU while solving an instance of the ATSP of size $N = 4$ with cutoff depth $d_{cpu} = 3$. Each node in A_{cpu} represents a concurrent backtracking root R_i of type `Node`.



Source – The Author.

control to the host several times. In such situations, the use of DP may lead to performance gains and result in a code closer to a high-level description of the algorithm. However, when these requirements are not met, the use of DP may result in significant overheads and makes the code more complex. In the scope of energy-efficient computing, it is a consensus that using DP leads to a more energy-efficient application.

The problems chosen as test-cases are the ATSP and the N-Queens. Although both are permutation combinatorial problems, they have many differences. Among the exact strategies for solving combinatorial problems, the backtracking implicitly evaluates the solution space systematically, building a tree in a depth-first order. Moreover, the backtracking is also a component of DFS-B&B algorithms.

GPU-based backtracking algorithms often apply the tree-oriented approach for parallelization, and usually follow the CPU-GPU model introduced in Section 2.4.3. It is challenging to run backtracking efficiently on GPUs. Characteristics of the problem commonly solved, allied to properties of DFS, result in fine-grained and irregular workloads. Moreover, an efficient code usually requires several problem-dependent parameters tuning.

Finally, the background section also has presented the base data structure and the control GPU-based implementation: BP-DFS. The base data structure uses bitsets to represent the current state of the search and aims at reducing the number of accesses to the local memory of the thread. In turn, BP-DFS was conceived mainly for solving permutation combinatorial problems. This implementation applies the base data structure, and follows the CPU-GPU model introduced in Section 2.4.3.

3 GPU-ACCELERATED BACKTRACKING ALGORITHM USING CUDA DYNAMIC PARALLELISM

This chapter presents a new GPU-accelerated backtracking strategy based on CDP and mainly designed for solving permutation combinatorial problems. The objective of this chapter is to verify whether and how much the use of CDP can improve the performance of a GPU-based backtracking algorithm in irregular scenarios.

In the proposed algorithm, the search starts on CPU, processing the tree until a first cutoff depth. Based on this partial backtracking tree, the memory required by the subsequent kernel generations is calculated and preallocated. Therefore, differently from the CDP-based strategies introduced in Section 2.4.4.1, GPU threads do not perform dynamic memory allocations.

The proposed algorithm has been extensively validated by solving instances of the 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 (BP-DFS), for some configurations. Moreover, the experimental results show that the proposed CDP-based implementation has much better worst-case execution times, and its performance is 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.

The main contributions reported in this chapter are the following:

- A CDP-Based backtracking algorithm that dynamically deals with the memory requirements of the problem and avoids dynamic allocations on GPU.
- The hybridization of this algorithm with another existing from the literature significantly boosts the performance of this related algorithm.
- This chapter presents conclusions on different aspects of the CDP programming model, such as the number of GPU streams, dynamic allocation, global memory mapping, and block size.
- The identification of difficulties, limitations, and bottlenecks concerning the CDP programming model may be useful for helping potential users and motivate lines for further investigations.

The remainder of this chapter is organized as follows. Section 3.1 introduces the initial premises considered in the conception of the strategy. Sections 3.2 to 3.7 detail the

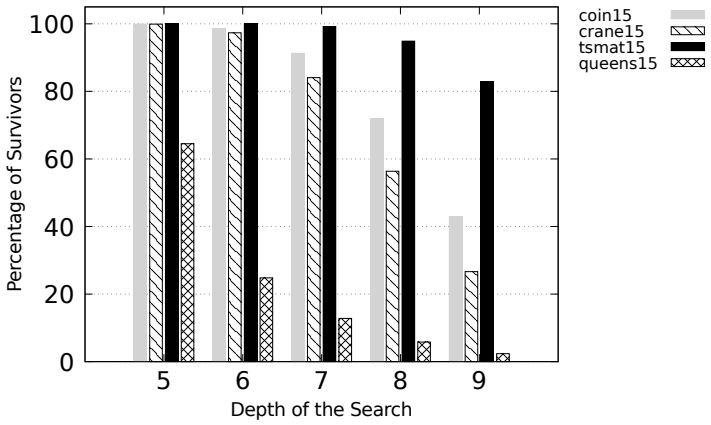
new CDP-based backtracking algorithm. Next, a computational evaluation is presented in Section 3.8. Finally, Section 3.9 outlines the concluding remarks.

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} (PLAUTH *et al.*, 2016). 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 (SOMERS, 2002).

Figure 8 shows, for ATSP instances and N-Queens of size 15, the percentage of nodes not pruned at depths 5 to 9 compared to the maximum theoretical number of nodes, called survivors. For all ATSP instances, the percentage at depths 5–7 is superior to 80% of the maximum theoretical number of nodes. For N-Queens, this percentage is much lower, around 15% at depth 7.

Figure 8 – Percentage of nodes not pruned at depths 5 to 9 compared to the maximum theoretical number of nodes. Results are for ATSP instances and N-Queens of size $N = 15$. The initial upper bound was set to the optimal solution.



Source – The author.

In a CUDA-based algorithm, if dynamic allocations on GPU require more than 8 MB, the variable `cudaLimitMallocHeapSize` must be set accordingly. However, to know these requirements in advance is difficult, and insufficient heap size leads to runtime errors. For example, Figure 8 illustrates that the solution space is much bigger for class *tsmat* than for class *crane*, although both instances are of size $N = 15$. Moreover, such memory requirements may be huge for permutation combinatorial problems (ZHANG, 1996).

Consider instance *tsmat*15 and the algorithm DP3, previously introduced in Section 2.4.4.1. 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):

$$\left\lceil \frac{(15-1)!}{(15-4)!} \times sizeof(Node) \right\rceil + \left\lceil \frac{(15-1)!}{(15-8)!} \times sizeof(Node) \times 0.9 \right\rceil,$$

which is a value bigger than $600MB$. Performing dynamic allocations on GPU's heap of such a big amount of memory may be harmful to the performance of GPU-based algorithms (STEINBERGER *et al.*, 2012; WIDMER *et al.*, 2013).

Based on these premises, this chapter presents a CDP-based algorithm that extends the parallel backtracking model introduced in Section 2.4 and does not perform dynamic allocations on GPU. As BP-DFS, the proposed approach begins on CPU and stores frontier nodes at d_{gpu} in the active set. The first kernel generation searches from d_{cpu} to the second cutoff depth d_{gpu}, and stores the frontier nodes in an active set on global memory. The second generation of kernels is responsible for searching from d_{cpu} to the solution depth N. Memory requirement and allocations on global memory are managed by the host to avoid runtime errors. This management takes into consideration the partial backtracking tree, the requirements of the subsequent kernel generations launched by CDP, and properties of the device. The communication between parent 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 child grids.

The next sections provide a detailed description of the main steps of the proposed algorithm: *initial CPU search*, *memory requirement analysis*, *launching the Intermediate GPU Search*, *Intermediate GPU Search*, and *Final GPU Search*.

3.2 Initial CPU search

The initial CPU search procedure is described in Algorithm 2. Before the search begins, the algorithm reads the instance size N, the cost matrix $C^h_{N\times N}$, and the cutoff depth d_{cpu} (*lines* 1 – 3). Then, to make the pruning process more efficient, the variable *upper_bound* gets the cost of a valid complete solution for the instance at hand (*line* 4). *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} (feasible and valid incomplete solutions) in the active set A^h_{cpu}. After the initial search, the variable *survivors* receives the $|A^h_{cpu}|$ (*line* 7).

The initial CPU backtracking can be performed in parallel. However, the initial search explores just a small fraction of the solution space. The programmer needs to ensure

Algorithm 2: Initial CPU Search.

1 $N \leftarrow get\_instance\_size()$
2 $C^h \leftarrow get\_cost\_matrix()$
3 $d_{cpu} \leftarrow get\_cpu\_cutoff\_depth()$
4 $upper\_bound \leftarrow get\_initial\_solution(N, C^h)$
5 $A^h_{cpu} \leftarrow \emptyset$
6 $initial\_CPU\_search(A^h_{cpu}, d_{cpu}, N, C^h, upper\_bound)$
7 $survivors \leftarrow |A^h_{cpu}|$

that the overhead of initializing a parallel search procedure is negligible compared to the time spent exploring this small fraction of the search space. The `Node` representation and search procedure are the ones introduced in Section 2.5.

3.3 Analysis of Memory Requirement

To avoid dynamic allocations and to cope with possible GPU memory limitations, the Analysis of Memory Requirement proceeds 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} may 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 the 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 (*line* 10) 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 (*lines* 14 – 17). 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 kernel status if `cudaDeviceSynchronize()` is called after a child kernel launch. According to Adinetz (2014), up to 150 MB are stored for each parent kernel generation (host included), depending on the hardware. Algorithm 3 considers this value equal to 150 MB because the exact value a GPU stores cannot be easily obtained (refer to Annex A). Therefore, $2 \times 150MB$ are removed 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 and 12). Parameter configurations will be further detailed in Section 3.8.2.

Algorithm 3: Analysis of Memory Requirement and data allocation.

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

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 $r_{cdp} \leftarrow memo\_requirement(p, chunk, max\_A_{gpu}\_size, N, C^h)$
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 $\quad r_{cdp} \leftarrow memo\_requirement(p, chunk, max\_A_{gpu}\_size, N, C^h)$
13 **end**
14 $cudaMalloc(A^d_{cpu}, chunk \times sizeof(Node))$
15 $cudaMalloc(A^d_{gpu}, max\_A_{gpu}\_size \times sizeof(Node))$
16 $cudaMalloc(C^d, N \times N \times sizeof(int))$
17 $cudaMalloc(control\_data^d, ...)$

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 GPU Search call proceeds.

Initially, two integer variables *counter* and *remaining* are initialized to 0 and *survivors*, respectively (*lines* 1 – 2). 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 GPU search, 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.

Input: Cost matrices $C^h_{N\times N}$, $C^d_{N\times N}$, $survivors$, $chunk$, A^h_{cpu}, A^d_{cpu}, A^d_{gpu}, d_{cpu}, d_{gpu}, $control\_data^d$, $expected\_children\_d_{gpu}$, the global upper bound, and the size N of the problem.

```
1  counter ← 0
2  remaining ← survivors
3  set_CDP_variables()
4  cudaMemCpy(C^d, C^h, N × N × sizeof(int), H2D)
5  while counter < survivors do
6      nt ← get_block_size()
7      nb ← ⌈chunk/nt⌉
8      cudaMemCpy(A^d_cpu, (A^h_cpu + counter), chunk × sizeof(Node), H2D)
9      intermediate_GPU_search <<< nb, nt >>>
         (C^d, chunk, expected_children_d_gpu, A^d_cpu, A^d_gpu, d_cpu, d_gpu, upper_bound, control_data^d)
10     retrieveControlData(control_data^h, control_data^d, chunk, expected_children_d_gpu)
11     counter ← counter + chunk
12     remaining ← remaining − chunk
13     if remaining < chunk then
14         chunk ← 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 [1]. 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

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

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

The Global memory is the communication interface between the host and the device, as well as between two generations of kernels. 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 and its data mappings are detailed in Algorithm 5.

3.5.1 Initialization and Search Procedure

All frontier nodes in A^d_{cpu} are roots $R_i, i \in \{0, ..., chunk - 1\}$, of a disjoint search space S_i, as shown in Figure 7. The 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 valid and incomplete solution now has size d_{gpu}.

3.5.2 Block-Based Active Set

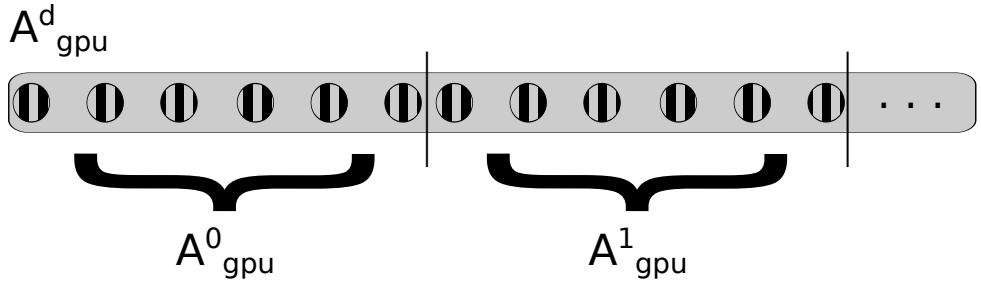
In a block-based organization, each block bl_b, $b \in \{0, ..., nb - 1\}$, has its active set A^b_{gpu}, where $A^b_{gpu} \subseteq A^d_{gpu}$. All threads belonging to block bl_b populate A^b_{gpu} concurrently. The active set A^b_{gpu} is then initialized by the block's master thread (*line 5*), pointing 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 variable $expected\_children\_d_{gpu}$ is the number of children nodes at depth d_{gpu} expected by each node at d_{cpu}. The variable nt represents the size of the block b ($blockDim.x$) defined in Algorithm 4. Finally, b corresponds to the index of the block ($blockIdx.x$). Figure 9 illustrates a division of A^d_{gpu} into nb block-based active sets.

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*).

Figure 9 – Division of A^d_{gpu} for $nt = 2$ and $expected\_children\_d_{gpu} = 3$. For this configuration, each block-based active set stores at most 6 children nodes from depth d_{gpu}. This way, A^0_{gpu} corresponds to positions $A^d_{gpu}[0:6]$; A^1_{gpu} corresponds to positions $A^d_{gpu}[6:12]$, and so on.



Source – The author.

Algorithm 5: Intermediate GPU Search.

Input: Cost matrix $C^d_{N\times N}$, $chunk$, $expected\_children\_d_{gpu}$, A^d_{cpu}, A^d_{gpu}, d_{cpu}, d_{gpu}, $number\_of\_kernels$, the global upper bound, and the size N of the problem.

```
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()
   /* Continues on Algorithm 6                                              */
```

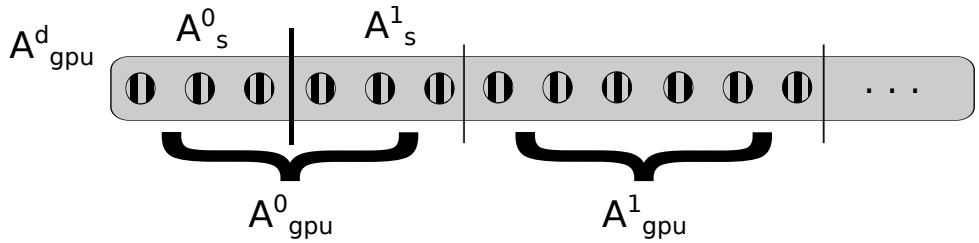
3.5.3 Launching the final GPU search

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 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. In the case the algorithm launches 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, the algorithm to launch the Final GPU Search proceeds as follows. Algorithm 2 receives the number of kernels that each block b launches ($number\_of\_kernels$). After the block barrier (*Algorithm 5, line* 14), if the identifier of the thread inside the block is smaller than the number of kernels that the block is supposed to launch ($threadIdx.x < number\_of\_kernels$), then thread Th_l of block b, $l \in \{0, 1, ..., number\_of\_kernels - 1\}$, creates and initializes stream St_l (*Algorithm 6, lines* 1 – 4). In *line* 5 (*Algorithm 6*), stream St_l receives the number of nodes to explore ($stream\_load$) according to $block\_load$, $number\_of\_kernels$, and its index ($stream\_idx$). Figure 10 illustrates a subdivision of A^b_{gpu} into $number\_of\_kernels$ subsets.

Figure 10 – Subdivision of A^b_{gpu} into $number\_of\_kernels = 2$ subsets. Before launching the second generation of kernels via CDP, A^0_{gpu} is divided into two other active sets: A^0_s and A^1_s. In this example $block\_load = 6$, which results in $stream\_load = 3$ for both St_0 and St_1. A^0_s corresponds to positions $A^0_{gpu}[0:3]$, and A^1_s corresponds to positions $A^0_{gpu}[3:6]$.



Source – The author.

Each stream St_l has its active set A^l_s, such that $A^l_s \subseteq A^b_{gpu}$. The data is 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*), as shown in Figure 11. After the kernel execution, there is an error checking to be further analyzed by the host ($line\ 11 - 12$).

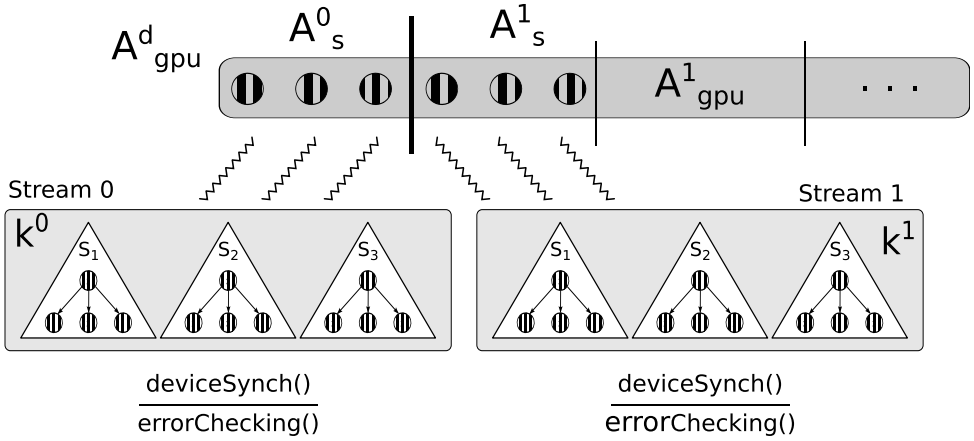
3.6 Final GPU Search

It is straightforward to combine different search strategies by using CDP. Thus, it is not mandatory for the next generations of kernel calls to be based on the recursive calls of

Algorithm 6: Launching Final GPU Search.

```
/* Continues from Algorithm 5                                        */
1  if threadIdx.x < number_of_kernels and block_load > 0 then
2      cudaStream_t stream
3      stream_idx ← threadIdx.x
4      initialize(stream)
5      stream_load ←
         get_stream_load(block_load, number_of_kernels, stream_idx)
6      initialize_stream_ASet(A^l_s)
7      nt ← get_cdp_block_size()
8      nb ← ⌈stream_load/nt⌉
9      final_GPU_search <<< nt, nb, stream >>>
         (C^d, d_gpu, A^l_s, stream_load, upper_bound, N)
10     deviceSynch()
11     error_vector[b] ← get_last_error()
12     destroyStream(stream)
13 end
```

Figure 11 – Threads Th_0 and Th_1 of block 0 initialize streams 0 and 1, respectively. All kernel launches are linked to a stream. Therefore, kernels K_0^0 and K_0^1 are launched by block 0 concurrently.



Source – The author.

the Intermediate GPU Search. This section presents two different ways of performing the search from d_{cpu} to the solution depth N. The first algorithm is called CDP-BP, which is a hybridization of the Intermediate GPU Search with BP-DFS, avoiding dynamic allocations. The second one, called CDP-DP3, is a hybridization of the Intermediate GPU Search with the DP3 parallel backtracking strategy, introduced in Section 2.4.

3.6.1 CDP-BP

This final kernel uses BP-DFS, introduced in Section 2.5.1, 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 (CARNEIRO *et al.*, 2011a; CARNEIRO *et al.*, 2012). This operation is done also by the intermediate kernel, which checks for new solutions. However, these details are not present in Algorithm 7. 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 $C_{N\times N}^d$, cutoff depth d_{gpu}, A_s^i,*stream_load* , the global upper bound, and the size N of the problem.

```
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 GPU Search, and its data mappings, with the DP3 strategy introduced in Section 2.4. 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 of the block. Thus, such a search strategy creates, configures, and destroys one GPU stream per thread of the block.

3.7 Final 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, 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.

3.8 Performance Evaluation

This section presents a performance evaluation of the CDP-based backtracking strategies proposed in this chapter: CDP-BP and CDP-DP3. The reminder of this section is organized as follows. Section 3.8.1 presents the experimental protocol. Section 3.8.2 lists the parameter settings. CDP-BP and CDP-DP3 are compared in Section 3.8.3 to the CDP-based strategies introduced in Section 2.4.4.1. Section 3.8.4 presents a best-worst comparison between CDP-BP and BPDFS. Finally, Section 3.8.5 brings portability experiments.

3.8.1 Experimental Protocol

The present performance evaluation compares different approaches for solving to optimality instances of the ATSP and for enumerating all valid configurations of the N-Queens problem. The strategies proposed in this chapter are the following:

- **CDP-BP**, which corresponds to the implementation summarized in Section 3.6.1;
- **CDP-DP3**, which implements the algorithm described in Section 3.6.2.

CDP-DP3 is an extension of CDP-BP that performs the same intermediate GPU search as CDP-BP and calls DP3 as the final GPU search, doubling d_{gpu} at each new recursive call of DP3.

For comparison, the following backtracking strategies have been implemented:

- **DP2** and **DP3**, which apply the ideas presented by Plauth *et al.* (2016) and introduced in Section 2.4.4;
- **BP-DFS**, corresponding to the GPU-based backtracking algorithm described in section 2.4;
- **Multi-core**, a multi-threaded version of BP-DFS that uses a pool scheme for load balancing;
- **Serial**, the backtracking used as a serial control implementation by a related work (PESSOA; GOMES, 2010). It is optimized for single-core execution and it is 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 2.5.1. Table 2 presents the key differences of all GPU-based implementations. The present performance evaluation also considers the highly optimized serial backtracking algorithm which is available at (SOMERS, 2002). This implementation is also used as a CPU baseline by several related works (FEINBUBE *et al.*, 2010; PLAUTH *et al.*, 2016; THOUTI; SATHE, 2012). 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 (KARYPIS; KUMAR, 1994). 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, i.e., 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 (PESSOA *et al.*, 2016).

Each experiment collects the following metrics: the kernel and application execution times, and the size of the tree. The NVIDIA Visual Profiler has also been used to get additional metrics. For the N-Queens, instances from size $N = 10$ to 18 are considered in the experiments. One may notice that none of the exploitable symmetries of the N-Queens problem have been explored. For the ATSP, instances of sizes $N = 10$ to 19 are considered in the experiments. The size of the explored tree increases rapidly with the instance size, ranging from a few thousand to billions of nodes, as shown in Table 1.

Table 1 – Number of nodes decomposed during the resolution of N-Queens of size $10 - 19$, and ATSP instances $coin10 - 20$, $crane10 - 20$, $tsmat10 - 19$ (in 10^6 nodes), initialized at the optimal solution.

Instance-#	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
crane	0.04	0.11	0.67	3.81	43.6	218.8	1,088	6,954	37,916	245,204
coin	0.11	0.43	1.87	10.7	107.8	500.4	1,379	3,710	15,089	116,840
tsmat	0.03	1.01	0.71	26.4	89.8	6,578	3,979	240,292	2,903,808	6,866,667
queens	0.03	0.16	0.85	4.67	27.35	171.12	1,141	8,017	59,365	461,939

Source – The Author.

Instances $tsmat18 - 19$ 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.

Table 2 – 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. Values are for ATSP and N-Queens. Numbers in brackets correspond to the explanations below the table.

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*	$nb_h * number\_of\_kernels^{(1,2)}$	*no*	2 - 7
$CDP - DP3$	*yes*	$nb_h + k_1$	*yes*	2 - 7 + DP3
$BP - DFS$	*no*	-	*no*	1 + 7

[a] $k_1 = (\sum_{d=d_{gpu}}^{base-1} survivors_d)^{(3,4)}$

Source – The Author.

where:

1. The subscript d means that such 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, $nb_h = \lceil |A^h_{cpu}|/nt_h \rceil$.

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. According to the notation, $d+1$ means the next recursive depth. For $N = 15$: $d_{cpu=2}$, $d_{gpu} = 4, 8, 15$, and base depth $base = 8$.

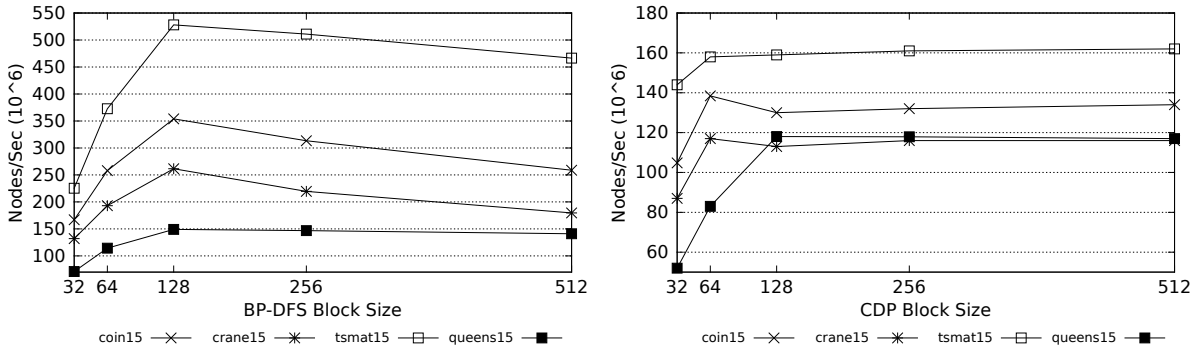
3.8.2 Parameters settings

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 K40m (GK110B chipset), 12 GB RAM, 2880 CUDA cores @ 745 MHz. According to the experiments reported in Annex A, the K40m reserves $109MB$ 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 12a shows the

experimental block size calibration for BP-DFS. All GPU-based implementations use the value of 128 for the first kernel configuration. Figure 12b shows the experimental block size calibration for the second kernel generation launched by CDP-BP. Table 3 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 3, the subscripts Q and A indicate that the parameter setting concerns the N-Queens or, respectively, the ATSP problem.

Figure 12 – (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/second).



(a) BP-DFS (b) CDP-BP

Source – The Author.

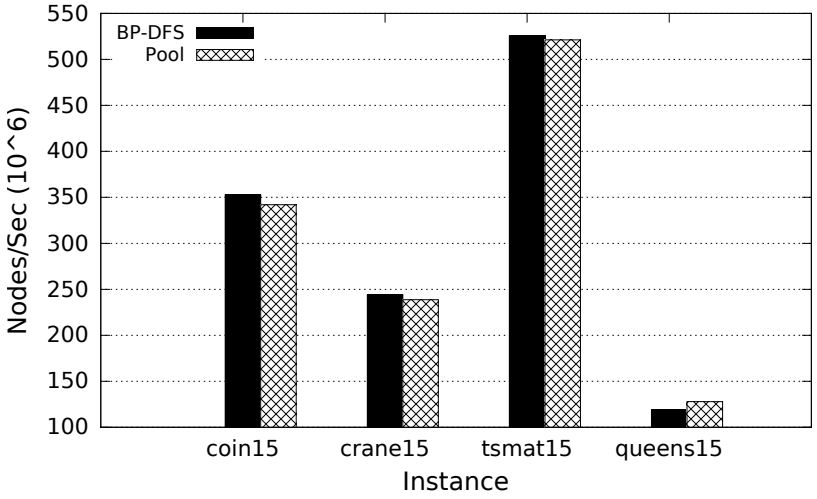
Figure 13 shows the processing rate of BP-DFS using a pool scheme for load balance and BP-DFS without load balance. The results show that the use of load balance improves performance of BP-DFS only for the N-Queens problem.

DP2 and DP3 perform dynamic allocations on GPU. However, Plauth *et al.* (PLAUTH *et al.*, 2016) 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, the value of `cudaLimitMallocHeapSize` was set 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, the available memory used in the experiments corresponds to $(0.7 \times p.memorySize)$.

Experiments to verify whether it is worth running a multi-threaded initial CPU

Figure 13 – 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 32768 GPU threads. All other parameters for BP-DFS and *pool* are the ones presented in Table 3. Results are shown for instances of size $N = 15$.



Source – The Author.

search were carried out. These tests run the Initial CPU Search with 2 to 40 threads. For $tsmat15$ 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 $55ms$. On the other hand, the multi-threaded initial CPU search is from 6.3 to 29 times 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 ($tsmat19$) 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.

The variable $number\_of\_kernels$ (Algorithm 6) defines the number of streams created/kernel launches for each GPU block of the Intermediate GPU Search. According to preliminaries experiments, using more than two streams per block brings no benefits to CDP-BP. This behavior is observed in all test-cases. Figure 14 shows, for instances of size 15, the influence of the number of streams created/kernel calls on the processing rate. Annex C presents a visual profile of CDP-BP for different values of $number\_of\_kernels$.

3.8.3 Comparison Between CDP-based Implementations

In this section, all CDP-based implementations are compared using the parameter configuration of Table 3. In Figure 15, one can see the average speedup reached by all CDP-based implementations compared to the serial control implementation.

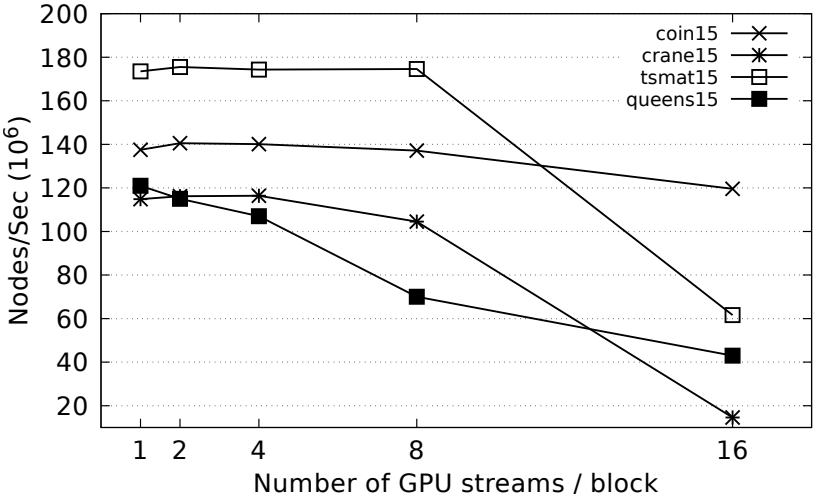
Table 3 – List of best parameters found experimentally for all parallel implementations.

	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)] The parameter are the same for ATSP and N-Queens.

Source – The Author.

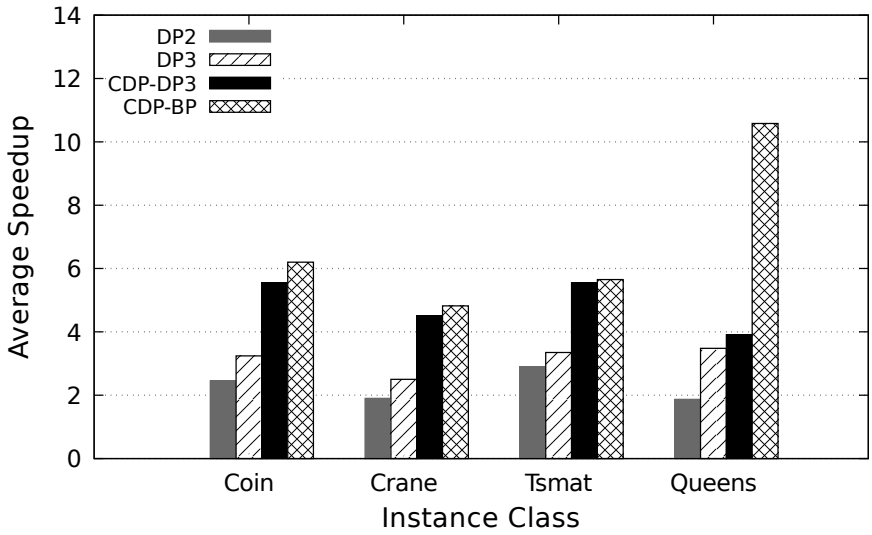
Figure 14 – Influence of the number of streams created/kernel calls on the processing rate (in 10^6 nodes/second) for CDP-BP. The figure shows the number of GPU streams/block *vs.* processing rate (in 10^6 nodes/second).



Source – The Author.

CDP-BP is the only implementation faster than the serial control implementation for all experiments. Speedups observed for CDP-BP range from 2.5 ($crane10$) to 13 ($queens12-14$). 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 6 times faster than DP3 ($coin10$), 13 times faster than DP2 ($queens12$), and up to 4.5 times ($tsmat10$) faster than CDP-DP3. As the solution space grows and the overhead becomes relatively less important, this difference decreases: for $N = 15$, CDP-BP is up to 3.7, 3, and 2.7 faster than DP2, DP3, and CDP-DP3, respectively.

Figure 15 – Average speedup reached by all CDP-based implementations compared to the serial one. Results are considering all classes of instances. Problem sizes are ranging from $N = 10$ to 15.



Source – The Author.

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 of $6\times$ ($tsmat15$). DP3 is superior to the serial implementation for $N \geq 12$. Speedups range from 1.6 ($coin12$) to 7.3 ($tsmat15$).

CDP-DP3 is a hybridization of CDP-BP and DP3: it 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}, and therefore CDP-DP3 performs less dynamic allocations and launches fewer kernels than DP3. CDP-DP3 has the second best overall result, with speedups ranging from 1.2 ($tsmat10$) to 9 ($coin15$) for all sizes bigger than $N = 10$. As the tree size grows, the benefits of a more regular load produced by DP3 strategy is observed. As a consequence, for sizes ranging from $N = 13$ to 15, CDP-DP3 has its performance close to CDP-BP's one.

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. Annex C presents a visual profile for each CDP-based implementation. It is possible to see the characteristics of each method concerning granularity of the kernels, number of kernel launches, and presence of recursion. No CDP-based strategy is faster than the highly optimized bit-parallel N-Queens solver that applies symmetries.

3.8.4 Comparing CDP-BP to BP-DFS: best and worst case analysis

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

For instances of size 17, Table 4 shows the execution times (in seconds) obtained when selecting the best, respectively the worst value for d_{cpu}. It also shows the median (i.e. the third best) execution time obtained for d_{cpu} from 3 to 7 and the relative standard deviation (RSD, defined as $\frac{standard\ deviation}{average} \times 100\%$). In brackets, beneath these execution times the corresponding parameter d_{cpu} (or d_{cpu}-d_{gpu}) are shown. For comparison, Table 4 also shows the serial execution time in angled brackets beneath the instance name.

Table 4 – Worst, best case and median execution times (in s), relative standard deviation (defined as 100%×(standard deviation)/(average)) for instances of size 17. Below each execution time the corresponding configuration is shown (in brackets). The serial execution time is shown in angled brackets $< >$.

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)		

Source – The Author.

Considering the N-Queens problem with $N = 17$, CDP-BP is 10× faster than the serial contro implementation even for the worst configuration ($d_{cpu} = 7$). In contrast, BP-DFS using its worst configuration is outperformed by the serial implementation. CDP-BP reaches speedup of 13.8 using its best configuration ($d_{cpu} = 4$). In turn, BP-DFS on its best settings is 22 times faster than the serial control implementation.

Similar results are observed for ATSP. 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 times.

For instance, if the worst parameter choice is made for BP-DFS and CDP-BP, the former spends $2,632$ seconds solving instance *coin*17, while the latter is about 25 times 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 times faster than its CDP-based counterpart.

For all ATSP instances of different size, 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 16a shows the influence of d_{cpu} choice on the processing rate for BP-DFS.

According to the NVIDIA Visual Profiler, BP-DFS uses the GPU resources poorly with d_{cpu} set to 3 and 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 times 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} set to 6 and 7 (and 5, in some cases), BP-DFS outperforms CDP-BP by a speedup factor of 2 or more.

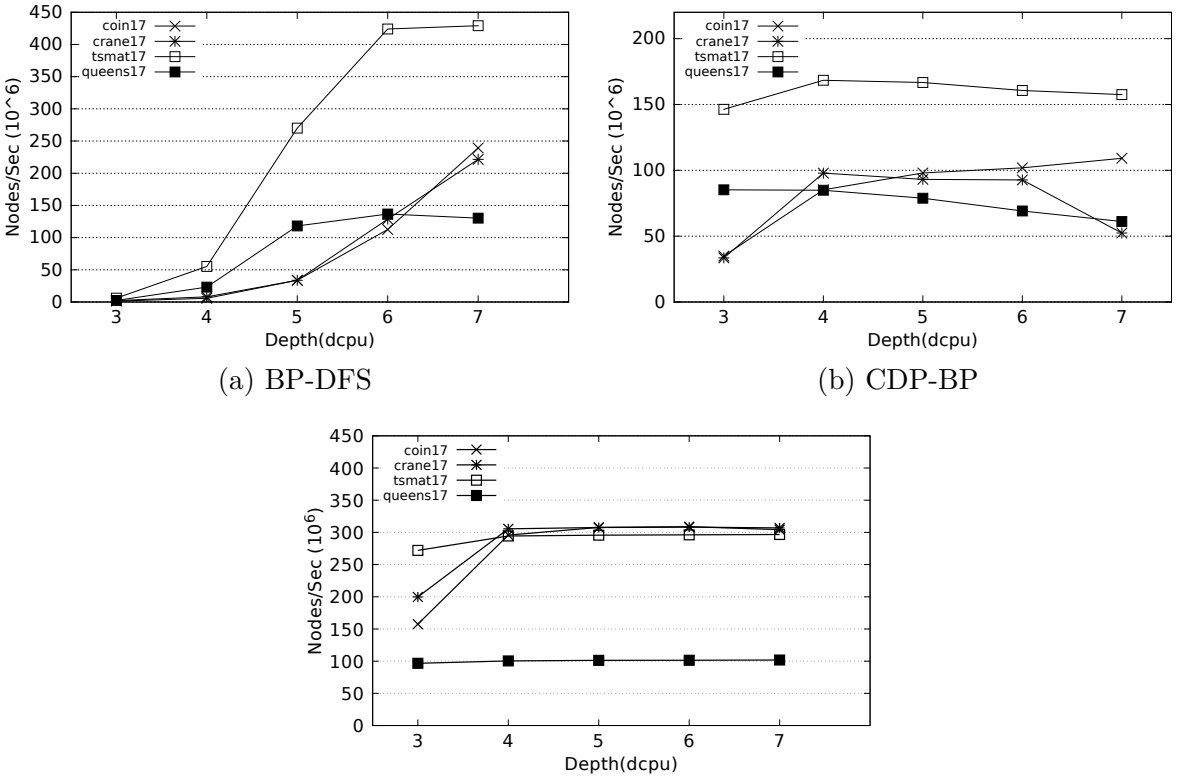
Figure 16b shows the influence of d_{cpu} on the processing rate for CDP-BP. Even when using CDP, the obtained performance still depends strongly on the tuning of the CPU search depth, especially for the *coin* and *crane* instances. However, a comparison of Figures 16a and 16b shows that CDP-BP is less 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 achieved by optimally tuning the CDP-based algorithm are 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.

As Figure 16c suggests, the multi-core implementation that uses a pool 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 17 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

Figure 16 – (a) Influence of d_{cpu} on the processing rate for BP-DFS. (b) Influence of d_{cpu} on the processing rate for CDP-BP. (c) Influence of d_{cpu} on the processing rate for the multi-core implementation. Processing rates are shown in 10^6 nodes/second for instances of size $N = 17$.



(a) BP-DFS

(b) CDP-BP

(c) Multi-core

Source – The Author.

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.

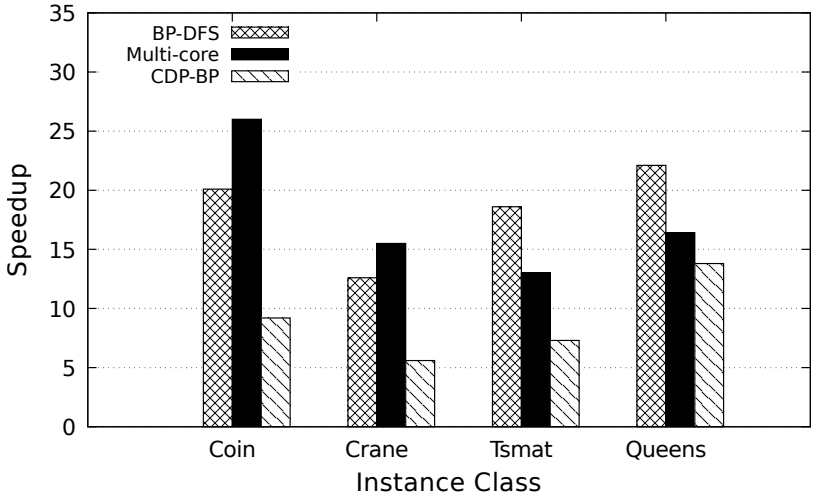
3.8.5 Portability experiments

This section presents two portability experiments. The first one performs the best-worst case analysis of Section 3.8.4 on different test-beds. The second one limits the amount of global memory available to $512MB, 1GB, ..., 8GB$.

3.8.5.1 Experiments on different test-beds

The objective of this new experiment is not to verify the speedups of CDP-BP and BP-DFS on different systems because metrics such as speedup strongly rely on the underlying CPU (GRAMA *et al.*, 2003). The objective of this new experiment is to confirm

Figure 17 – Speedup reached by BP-DFS, CDP-BP and multi-core implementations compared to the serial one. Results are considering all classes of instances for size $N = 17$. All implementations are using their best configuration for each instance, as shown in Table 4.



Source – The Author.

whether the behavior observed in Section 3.8.4 is hardware-dependent or not. For this purpose, two other different test-beds are considered: The first one is equipped with a Kepler GPU, and the second one is equipped with a Maxwell GPU. Both systems are detailed below.

- **Kepler test-bed:** it operates under Linux Ubuntu 14.04.3 LTS 64 bits, composed of an Intel Core i5-3330@ 3.20 GHz with 4 cores, 4 threads, and 32 *GB* RAM. It is equipped with a GeForce NVIDIA Tesla K20c (GK110 chipset - Kepler), 5 *GB* RAM, 2496 CUDA cores @ 706 MHz.
- **Maxwell test-bed:** it operates under Linux Ubuntu 14.04.3 LTS 64 bits, composed of an Intel Xeon E5-2630 v3 @ 2.40GHz with eight cores and 32 GB RAM. It is equipped with a GeForce NVIDIA GTX 980 (GM204 chipset - Maxwell), 4 GB RAM, 2048 CUDA cores @ 1126 MHz.

The GPUs used are from different architectures, and they have a distinct purpose. The Kepler K20c GPU is a hardware designed only for GPGPU that has a higher number of CUDA cores and a bigger global memory. In turn, the GTX 980 is a gamer hardware that has fewer CUDA cores than the K20c. On the other hand, GTX 980's CUDA cores are almost twice as fast as the ones of K20c.

Table 5 and 6 show the results for Kepler and Maxwell GPUs, respectively. In both tables, one can see the execution times (in seconds) obtained when selecting the

best, respectively the worst value for d_{cpu}. It also shows the median (i.e. the third best) execution time obtained for d_{cpu} from 3 to 7.

Table 5 – Worst, best case and median execution times (in s), for instances of size 17. Results are for the Maxwell GPU.

Inst.	$T_{worst}(s)$			$T_{best}(s)$			$T_{median}(s)$	
	BP-DFS	CDP-BP	Rate	BP-DFS	CDP-BP	Rate	BP-DFS	CDP-BP
*queens*17	1815	61	29	34	51	0.6	38	53
	(3)	(7-9)		(6)	(4-6)		(5)	(5-7)
*coin*17	2384	102	23	16	31	0.5	100	43
	(3)	(3-5)		(7)	(5-7)		(5)	(6-8)
*crane*17	2752	190	14	34	57	0.6	190	82
	(3)	(3-5)		(7)	(5-7)		(5)	(6-8)
*tsmat*17	36995	2051	18	615	1607	0.38	825	1677
	(3)	(6-8)		(6)	(5-7)		(5)	(7-9)

Table 6 – Worst, best case and median execution times (in s), for instances of size 17. Results are the Kepler GPU.

Inst.	$T_{worst}(s)$			$T_{best}(s)$			$T_{median}(s)$	
	BP-DFS	CDP-BP	Rate	BP-DFS	CDP-BP	Rate	BP-DFS	CDP-BP
*queens*17	3941	160	24	69	114	0.6	72	124
	(3)	(7-9)		(6)	(4-6)		(5)	(5-7)
*coin*17	3032	122	25	16	40	0.4	122	45
	(3)	(3-5)		(7)	(5-7)		(5)	(5-7)
*crane*17	3588	253	14	33	85	0.38	220	90
	(3)	(3-5)		(7)	(7-9)		(5)	(5-7)
*tsmat*17	45729	1699	27	515	1427	0.36	844	1508
	(3)	(3-5)		(6)	(4-6)		(5)	(6-8)

According to the results, the same behavior observed in Section 3.8.4 is observed on both testbeds: On the one hand, 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 a much better worst case execution time and makes the algorithm's performance less dependent on the tuning of d_{cpu}.

3.8.5.2 Memory experiments

CDP-BP stores enough memory to store the maximum number of children nodes that all survivor nodes at d_{cpu} can have at depth d_{gpu}. The number of nodes of a given depth d grows exponentially (ZHANG, 1996), and the memory requirements of d may be enormous. The amount of global memory limits the number of nodes the Initial GPU Search can process and, as a consequence, the number of kernels that CDP launches. There are different CDP-capable GPUs with varying sizes of memory, and this new experiment aims at verifying the influence of the global memory size on CDP-BP's performance.

This experiment solve instances of size $N = 18$ with $d_{cpu} = 7$ and $d_{gpu} = 9$, limiting the available global memory to $512MB, 1GB, 2GB, ..., 8GB$. Table 7 shows the memory requirements of CDP-BP and BP-DFS for $d_{cpu} = 7$. The requirements of CDP-BP for all ATSP instances are bigger than $25GB$, for the N-Queens problem is around $8GB$. In contrast, the memory requirements of BP-DFS are modest: even a GPU with $512MB$ of global memory can run BP-DFS with $d_{cpu} = 7$.

Table 7 – The number of survivor nodes (in 10^6) at $d_{cpu} = 7$, and memory requirements of CDP-BP and BP-DFS (in MB).

Instance	*Survivors*	Req_{BP-DFS}	Req_{CDP-BP}
*coin*18	8.143	228	25,311
*crane*18	8.450	236	26,263
*tsmat*18	8.910	249	27,695
*queens*18	2.215	57	7,645

Source – The Author.

Table 8 shows the best case execution, worst case, average execution times, and median execution times for CDP-BP. Table 8 also presents the number of times the host launches the Intermediate GPU Search.

Table 8 – Worst case, best case, median, and average (AVG) execution times (in s), and STDEV for instances of size 18. Below each execution time, the corresponding configuration is shown (in brackets). The table also shows the number of kernel launches by the host for $512MB, 1GB, ..., 8GB$.

Instance	$T_{worst}(s)$	$T_{best}(s)$	$T_{median}(s)$	AVG	STDEV	*Kernel launches*				
						0.5	1	2	4	8
*coin*18	162.97 ($512MB$)	161.337 ($4GB$)	162.212 ($8GB$)	162.2	0.635	109	36	15	8	4
*crane*18	466.45 ($4GB$)	461.21 ($1GB$)	464.55 ($512MB$)	464.2	1.9	136	36	15	8	4
*tsmat*18	22,164 ($4GB$)	21,876 ($512MB$)	22,065 ($2GB$)	22,033	94.96	136	45	19	8	4
*queens*18	1,037.39 ($512MB$)	1,028.32 ($8GB$)	1,031.10 ($4GB$)	1,032	3.182	36	12	5	2	1

Source – The Author.

As one can see in Algorithm 3, the variable *chunk* initially is set to *survivors* and then, it is decreased until it fits into the available memory (*lines* 9 – 13). It has been decreased in 20%. Having a suitable *chunk*, CDP-BP launches several times the Initial GPU Search (Algorithm 4, *lines* 5 – 15). For the scenario of $512MB$, CDP-BP has around $212MB$ to store nodes (see Section 3.3), and the host launches the first kernel more than 100 times for all ATSP instances, and 36 times for *queens*18. Even launching 136 times the Intermediate GPU Search, the configuration of $512MB$ is the best for *tsmat*.

The difference between the best and worst execution times is small compared to the overall execution time in all test-cases. This fact is confirmed by the average and standard deviation times. Therefore, the global memory size has a small influence on CDP-BP performance. However, it is important to point out that CDP-BP requires a huge effort to handle its high memory requirements, which contrasts to BP-DFS. The control GPU-based implementation presents low memory requirements even for the most demanding test-case.

3.9 Concluding remarks

This section brings the concluding remarks. First, conclusions concerning programmability, performance, and applicability of the CDP-based strategies proposed are outlined. Then, Section 3.9.3 introduces future research directions. Finally, Section 3.9.4 lists the main insights that summarize this chapter.

3.9.1 Programmability

Concerning the programmability, the results reported in this chapter contrast with the results of (ZHANG *et al.*, 2015), 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 2.2. 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 uses 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 brings extra complexity to the code and increases time spent in memory operations.

3.9.2 Performance and Applicability

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 non-expert 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.

According to the results, the ideas of DP3 (PLAUTH *et al.*, 2016) 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 if the size of the heap is 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.

Although it is intrinsically difficult to cope with fine-grained and irregular applications on GPUs, results herein reported 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.

3.9.3 Future research directions

A future research direction is on developing a function that decides dynamically whether CDP or even the GPU should be used or not. Such a decision function can be based on the analysis of the partial backtracking tree (CORNUÉJOLS; KARAMANOV; LI, 2006) and properties of the underlying hardware.

3.9.4 Main insights

The following summarizes the main insights from the experimental evaluation of GPU-backtracking algorithms using CDP presented in this chapter.

- The use of CDP is preferable in situations where BP-DFS is not able to use the GPU resources properly.
- The hybridization of BP-DFS with the Intermediate GPU Search results in a better worst-case performance for the control implementation. It also makes BP-DFS less dependent on good parameter settings.
- If well-tuned, BP-DFS shows better results than CDP-BP.

- 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, kernel launches, etc. According to the results, avoiding a huge number of dynamic kernel launches may increase the processing rate of a CDP-based application.

4 DYNAMIC SETUP OF CUDA RUNTIME VARIABLES FOR CDP-BASED BACKTRACKING ALGORITHMS

According to the experimental results of the last chapter, implementations that perform dynamic allocation on GPU's heap and/or launch more than two kernel generations require the setup of CUDA runtime variables. Otherwise, they may present runtime errors. The objective of this chapter is to provide means for such implementations to solve problems with different sizes and memory requirements: from few KB to several GB. To accomplish this objective, this chapter presents a generalization of the ideas of the previous one that lie mainly in the memory requirement analysis.

DP3-based strategies allocate memory for the block active set if at least one thread of the block finds a frontier node. However, blocks have a fixed and predetermined size, and a block may have inactive threads. Such a property can make the memory requirement analysis imprecise. The second part of this chapter presents a variation of DP3 that allocates memory in a thread-based manner, called TB-DP3.

According to the experimental results, the objectives of the chapter were accomplished. By using the set of proposed algorithms, all DP3-name implementations can solve all test-cases. Moreover, the thread-based allocation results in insignificant performance losses.

The main contributions reported in this chapter are the following:

- A strategy that dynamically calculates the memory requirements of the algorithm, independently of how many generations of kernels it launches. This approach is also used to setup the CUDA runtime variables accordingly.
- A thread-based allocation that results in insignificant performance losses compared to approaches of the literature.

The remainder of this chapter is organized as follows. Section 4.1 lists the challenging issues considered to generalize the ideas of the last chapter. Sections 4.2 to 4.3 detail all algorithms proposed in this chapter. Section 4.4 introduces TB-DP3. Section 4.5 presents a performance evaluation. Finally, Section 4.6 outlines the concluding remarks of this chapter.

4.1 Challenging issues

According to the last chapter, DP3 and CDP-DP3 are much more complex than CDP-BP and BP-DFS. First of all, these implementations launch recursively more than two

kernel generations. Running DP3 and CDP-DP3 without a proper setup of the maximum nesting depth may result in runtime errors because the default value is two. Moreover, up to $150MB$ are reserved for each kernel generation. This way, the memory reserved for nesting level synchronization may be huge and cannot be ignored.

It was also observed in the last chapter that without setting up the GPU heap size, all implementations that allocate memory on GPU's heap cannot solve problems bigger than $N = 11$. The size of the default CUDA heap is $8MB$ (see Annex B), and if an application requires a larger heap, the variable `cudaLimitMallocHeapSize` must be set accordingly. Furthermore, DP3-based implementations dynamically allocate memory according to the block size.

In the CUDA programming model, the block has a fixed and predetermined size $block\_size$. Suppose that Th_t is the only active thread of block bl^b, and assume that the number of expected children nodes a node at d_{cpu} expects at d_{gpu} is $expected\_chidren\_d_{gpu}$. If Th_t finds the first survivor at d_{gpu}, it allocates memory enough to store $block\_size \times expected\_childen\_d_{gpu}$ nodes. When the number of active threads in a block is smaller than the block size, more memory than the necessary is allocated on GPU's heap. Thus, the block-based allocation strategy makes the calculation of the required heap less precise.

Based on the above premises, this chapter proposes modifications on the Memory Requirement Analysis. Instead of calculating the size of A^d_{gpu}, the host calculates the heap size by recursively applying the strategy of Algorithm 3 until the base depth is reached. Moreover, there is an algorithm to discover, based on the way the next d_{gpu} is calculated, the number of kernel generations the search launches based on the size N of the problem. Finally, the Memory Requirement Analysis is used to get a subset $S \subseteq A^h_{cpu}$ of size $chunk$ before launching the search.

The following sections provide a detailed description of the updated memory requirement analysis and the kernel launch. TB-DP3, the implementation of DP3 with thread-based allocation, is introduced next.

4.2 Memory requirement analysis

Unstructured tree search algorithms that dynamicallty allocate memory on GPU's heap, such as DP3, need to store in global memory A^d_{cpu}, the cost matrix, and control data for the subsequent kernel generations. Moreover, it is necessary to reserve memory for nesting level synchronization and the heap. The new memory requirement analysis proposed in this section is based on Algorithm 3, and it consists of three steps: getting the number of kernel generations, heap size calculation, and calculating the required global memory.

4.2.1 Getting the number of kernel generations

It is necessary to know the number of kernel generations to set up the CUDA runtime variable `cudaLimitDevRuntimeSyncDepth` and to calculate the amount of memory reserved by the GPU to keep track of the parent block context data.

Algorithm 8 shows the function that returns the maximum synchronization depth based on the base depth and the size N of the problem. Initially, the algorithm receives the *current_depth* of the search and the size N of the problem. Next, it gets the base of the recursion (*line* 1). Then, the number of synchronization depths 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 depth of the recursion (*line* 1). In turn, the second one is responsible for returning the next depth of the recursion (*line* 5).

Algorithm 8: Maximum Synchronization Depth (number of kernel generations).

Input: The size N of the problem, and the initial cutoff depth *current_depth*.
Output: The number of synchronization depths (kernel generations), also including the one launched by the host.

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

4.2.2 Requested heap size

The strategy employed in this step is the same one used by CDP-BP for calculating the size of A^d_{gpu} (Algorithm 3, *lines* 3 – 5), which takes into consideration the maximum number of children nodes that a node at d_{cpu} can have at d_{gpu}.

Algorithm 9 presents the function responsible for returning the maximum requested heap size. The heap size calculation receives as parameters *chunk*, which is the size of a subset $S \subseteq A^h_{cpu}$, and the size N of the problem. The algorithm calculates for each recursive kernel call the memory required to store $survivors_{next\_depth}$, until it reaches the base depth (*lines* 6 – 12).

The heap size calculation is the most important step of the memory requirement analysis. Having the heap size for *chunk* nodes, it is possible to determine the amount of global memory required.

Algorithm 9: Calculation of the requested heap size.

Input: The size *chunk* of $S \subseteq A^h_{cpu}$, and the size N of the problem.
Output: The maximum requested heap size in bytes.

```
1  requested_heap ← sizeof(Node) × chunk
2  base_depth ← get_base_depth(N)
3  current_depth ← get_initial_depth()
4  max_nodes ← 0
5  nodes_current_depth ← chunk
6  while (current_depth < base_depth) do
7      next_depth ← get_next_depth(current_depth)
8      max_current ← get_max(current, N)
9      max_next ← get_max(next, N)
10     expected_childen_d_next ← max_next / max_current
11     requested_heap ← requested_heap × expected_childen_d_next
12     current_depth ← next_depth
13 end
```

4.2.3 Required global memory

Algorithm 10 returns the global memory to be required based on a subset $S \subseteq A^h_{cpu}$ of size *chunk*. Initially, the memory reserved for depth synchronization is calculated in line 1. As pointed out in Section 3.3, a value equals to 150MB is reserved for each kernel generation. Annex A shows a way to retrieve the exact amount of memory a CDP-capable GPU keeps for this purpose. The amount of global memory required to store the control data, A_{cpu}, and the requested heap is get in lines 3 – 4. Finally, in line 5, the required global memory is calculated by adding the values got in lines 2 – 5.

Algorithm 10: Required memory based on *chunk* nodes and the size of the problem.

Input: The size *chunk* of $S \subseteq A^h_{cpu}$, the size N of the problem, and the number k of kernel generations.
Output: Total of global memory required.

```
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_maximum_heap(chunk, N)
6  required_memory ←
     required_heap + nesting_memory + activeSet_memory + control_memory
```

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.

4.3 Launching the first kernel generation

Before launching the first kernel generation, it is necessary to get a subset $S \subseteq A^h_{cpu}$ of size *chunk* such that its requirements fit into the global memory. For this purpose, it is necessary to take into consideration, for each node in A^h_{cpu}, the amount of memory required by its whole subtree, from d_{cpu} to *base*.

Algorithm 11 shows how to get a suitable $S \subseteq A^h_{cpu}$ of size *chunk*. Initially, this algorithm receives as parameters the size of A^h_{cpu} ($suvivors\_d_{cpu}$), and the size N of the problem. In first place, the variable *chunk* receives $survivors\_d_{cpu}$ and the algorithm calculates the amount of memory required based on this *chunk* (*lines* 1 – 2). Thus, the calculation of a suitable chunk employs Algorithm 10.

Algorithm 11: Algorithm that returns a suitable chunk size.

Input: $survivors\_d_{cpu}$, the size N of the problem, and the number k of kernel generations.
Output: A suitable chunk size.

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

If the memory required by S is bigger than the available global memory, the variable *chunk* is decreased until its required memory fits into the available global memory (*lines* 4 – 6). If there is no S such that its required memory fits into the available global memory, the program returns an error (*lines* 7 – 9).

Algorithm 12 presents the launching of the first kernel generation on GPU. Its implementation is close to Algorithm 4, which launches the Intermediate GPU Search. After determining a suitable S (*line* 1), the values of variables `cudaLimitMallocHeapSize` and `cudaLimitDevRuntimeSyncDepth` are set in line 2. Finally, lines 5 – 15 process $S \subseteq A^h_{cpu}$ of size *chunk* until A^h_{cpu} is empty. After each kernel call, control data is retrieved.

4.4 TB-DP3

This section presents TB-DP3, a variation of DP3 that performs dynamic allocation in a thread-based manner. Unlike CDP-BP and CDP-DP3, TB-DP3 is not a hybridization of search strategies. It is based on recursive kernel calls, like DP3 and DP2.

Algorithm 12: Launching the search on GPU.

Input: Cost matrix $C^h_{N\times N}$, $C^d_{N\times N}$, $survivors\_d_{cpu}$,A^h_{cpu}, A^d_{cpu},d_{cpu}, $control\_data^d$, $expected\_children\_d_{gpu}$, the global upper bound, and the size N of the problem.

1 $chunk \leftarrow get\_suitable\_chunk(survivors, N)$
2 $set\_CDP\_variables(chunk, N)$
3 $counter \leftarrow 0$
4 $remaining \leftarrow survivors$
5 **while** $counter < survivors$ **do**
6 $\quad nt \leftarrow get\_block\_size()$
7 $\quad nb \leftarrow \lceil chunk/nt \rceil$
8 $\quad cudaMemCpy(A^d_{cpu}, (A^h_{cpu} + counter), chunk \times sizeof(Node), H2D)$
9 $\quad GPU\_search <<< nb, nt >>>$
$\quad (C^d, chunk, expected\_childen\_d_{gpu}, A^d_{cpu}, d_{cpu}, d_{gpu}, control\_data^d, upper\_bound, N)$
10 $\quad syncDataD2H(control\_data^h, control\_data^d, chunk)$
11 $\quad counter \leftarrow counter + chunk$
12 $\quad remaining \leftarrow remaining - chunk$
13 $\quad$ **if** $remaining < chunk$ **then**
14 $\quad\quad chunk \leftarrow remaining$
15 $\quad$ **end**
16 **end**

4.4.1 The algorithm

The initial CPU search is the one described in Section 3.2. It performs backtracking from the root depth (1) until the first cutoff depth $d_{cpu} = 2$. The size of A^h_{cpu} at $d_{cpu} = 2$ is at most $N - 1$ nodes[1]. After the initial CPU search, TB-DP3 uses Algorithm 12 to launch the first kernel generation on GPU and set up properly the CUDA runtime variables. The next section gives details of TB-DP3 kernel.

4.4.1.1 The kernel

Algorithm 13 presents a pseudo-code for TB-DP3's kernel. Initially, each thread $Th_i, i \in \{0, 1, ..., chunk - 1\}$, verifies if the current depth is the base one (*line* 3). Then, thread Th_i initializes its data with root node $R_i \in A^d_{cpu}$ (*line* 5). If the current depth is not the base, thread Th_i initializes its local active set by allocating memory for $expected\_children\_d_{gpu}$ children nodes. Next, it gets the current depth and initializes the variable *local_load* to zero (*lines* 7 – 9).

After evaluating its solution space from *current_depth* to d_{gpu}, Th_i initializes one stream and launches recursively a next generation of TB-DP3 (*lines* 12 – 17). TB-DP3

[1] For the N-Queens it is $N \times (N - 1)$

launches several generations of kernels, which makes it difficult to create a thread-to-data mapping for the control data. Therefore, the control data is atomically accessed. If the parent node of a kernel c is $f \in A^d_{cpu}$, the kernel c accesses $control\_data[f]$ to return all information collected by the search (*line* 19).

TB-DP3 is not a hybridization of search strategies. It uses TB-DP3 as the intermediate GPU search and also the final one, controlled by a base condition (*line* 6). If the search reaches the base depth, TB-DP3 evaluates the solution space from the base depth until N (*lines* 23 – 25).

Algorithm 13: A pseudo-code for the kernel of TB-DP3.

Input: Cost matrix $C^d_{N\times N}$,A_{gpu},$chunk$,$expected\_children\_d_{gpu}$, d_{cpu}, d_{gpu}, the global upper bound, and the size N of the problem.

```
1  idx ← blockIdx.x × blockDim.x + threadIdx.x
2  if idx < chunk then
3      base_get_base_depth(N)
4      current_depth ← d_cpu
5      local_root ← A^d_cpu[idx]
6      if current_depth < base then
7          local_load ← 0
8          initialize_local_active_set(A^idx_gpu)
9          d_gpu ← get_next_depth(current_depth, N)
10         Perform backtracking based on local root information, using d_gpu for cutoff
             condition, and upper_bound for pruning.
11         if local_load > 0 then
12             cudaStream_t stream
13             stream_idx ← idx
14             initialize(stream)
15             nt ← get_cdp_block_size()
16             nb ← ⌈local_load/nt⌉
17             TB_DP3 <<< nt, nb, stream >>> (C^d, A^idx_gpu, local_load, ...)
18             deviceSynch()
19             error_vector[f] ← get_last_error()
20             destroyStream(stream)
21         end
22     end
23     else
24         Perform the Final GPU Search as presented by Algorithm 7.
25     end
26 end
```

4.5 Computational Evaluation

This section intends to validate the algorithms proposed in Sections 4.2 to 4.4. The remainder of this evaluation is organized as follows. First, the experimental protocol and parameter settings are outlined in Sections 4.5.1 and 4.5.2. Then, new implementations of DP3-based algorithms are compared and analyzed in Section 4.5.3.

4.5.1 Experimental Protocol

The primary objective of the present evaluation is to validate Algorithms 8 to 11. To archive this objective, the following implementations that dynamically allocate memory on GPU and launch more than two kernel generations were reimplemented using the algorithms listed above:

- **DP3**: strategy introduced in Section 2.4.4 that revisits the ideas of Plauth *et al.* (2016).
- **CDP-DP3**: hybridization of CDP-BP and DP3 introduced in Section 3.6.2.

Moreover, for the purpose of this evaluation, two other implementations are proposed:

- **TB-DP3**: variation of DP3, introduced in Section 4.4, that allocates memory on GPU in a thread-based manner.
- **CDP-TBDP3**: variation of CDP-DP3 that uses TB-DP3 as the Final GPU Search.

The second objective of this evaluation is to investigate whether the use of a thread-based allocation decreases the nodes/second rate of the tested algorithms or not.

All implementations listed above apply Algorithms 8 to 11 to calculate their memory requirements, setup the CUDA runtime variables, and launch the first kernel generation. All parallel strategies listed above use the data structure described in Section 2.5.1. For comparison, the present performance evaluation also uses the serial CPU-baseline already introduced in Section 3.8.1, as well as the same experimental protocol detailed in Section 3.8.1. Table 9 presents the key differences of all DP3-based implementations.

Table 9 – Key differences between all DP3-based implementations: number of dynamic allocations on GPU's heap, number of GPU streams / CDP kernels launched, and algorithms employed. Values are for ATSP and N-Queens. Numbers in brackets correspond to the explanations below the table.

Implementation	*# Dynamic Allocations*	*GPU streams/CDP kernels*	*Algorithms*
$DP3$	$nb_h + k_2^{(b)}$	$\lvert A^h_{cpu}\rvert + k_1^{(a)}$	8 - 11 + DP3
$TB-DP3$	$\lvert A^h_{cpu}\rvert + k_1$	$\lvert A^h_{cpu}\rvert + k_1$	8 - 11 + TB-DP3
$CDP-DP3$	k_2	$nb_h + k_1$	5 - 7 + 8 - 11 + DP3
$CDP-TBDP3$	k_1	$nb_h + k_1$	5 - 7 + 8 - 11 + TB-DP3

[a] $k_1 = (\sum_{d=d_{gpu}}^{base-1} survivors_d)$
[b] $k_2 = (\sum_{d=d_{gpu}}^{base-1} nb_d)^{(1,2)}$

Source – The Author.

where:

1. The subscript d means that such variable concerns a given depth d. The subscript h means that such 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, $nb_h = \lceil \lvert A^h_{cpu}\rvert / nt_h \rceil$. In k_2, this notation is extended to say that nb_d is the number of blocks configurations for launching all kernels in depth d.

4.5.2 Parameters Settings

The present evaluation follows the same parameters settings of Section 3.8.2 with respect to the programming languages, metrics, instances, and test-bed used. The heap size calculation is not precise for DP3-named implementations because it is block based and a block may have inactive threads. According to preliminary experiments, the heap for a DP3-named implementation must be 1.5× bigger than the heap of its TB-DP3-named counterpart to run all tests without runtime errors. Moreover, the implementations herein evaluated do not require to set up d_{cpu} and d_{gpu}. Table 10 presents the best parameter configurations for all DP3-based implementations.

4.5.3 Comparison Between all DP3-based implementations

In this section, all DP3-based implementations are compared using the parameter configuration of Table 10. First of all, it is important to point out that by using Algorithms 8 to 11, all implementations listed in Section 4.5.1 can run all the test-cases without runtime errors.

Figure 18 shows the average speedup reached by all DP3-based implementations for each instance class, problem sizes range from $N = 10$ to 12. Table 11 presents the average

Table 10 – List of best parameters found experimentally for all DP3-based implementations.

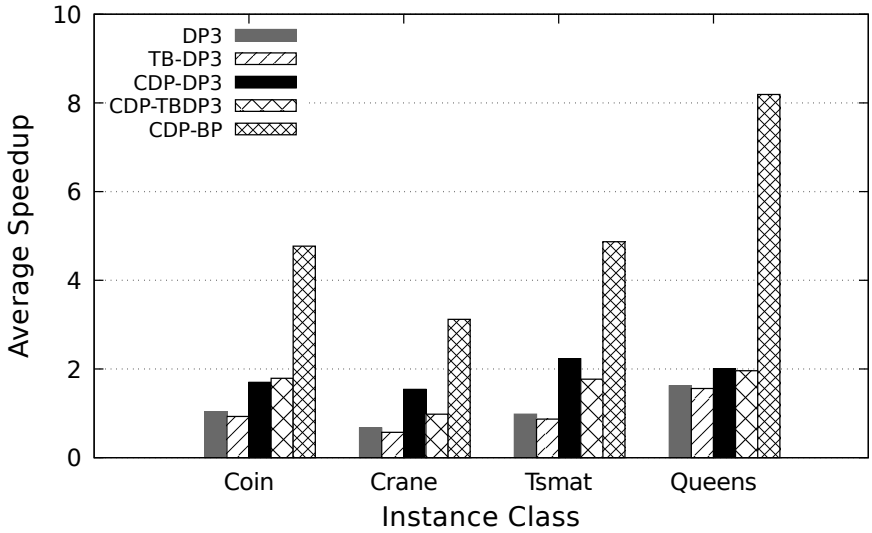
	Parameters Settings		
Implementation	Block Size	Bl. Size-CDP	Heap
$TB - named^A$	128	64	$h\ MB$
$TB - named^Q$	128	32	$h\ MB$
$DP3 - named^A$	128	64	$1.5 \times h\ MB$
$DP3 - named^Q$	128	32	$1.5 \times h\ MB$

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

Source – The Author.

speedup reached by all DP3-based implementation, problem sizes range from $N = 10$ to 12. For this range of sizes, DP3 is on average 12% faster than TB-DP3 to perform the same task. The same behavior is observed for CDP-DP3 and CDP-TBDP3. For this range of sizes, CDP-DP3 is on average 34% faster than $CDP - TBDP3$ to perform the same task. The dynamic allocations of CDP-DP3 happen only once at the second d_{gpu}, in the first generation of DP3. The number of calls to `malloc()` and `free()` is high, and they impact negatively on the execution time for instances of sizes from $N = 10$ to 12.

Figure 18 – Average speedup reached by all DP3-based implementations for each instance class. Problem sizes range from $N = 10$ to 12.



Source – The Author.

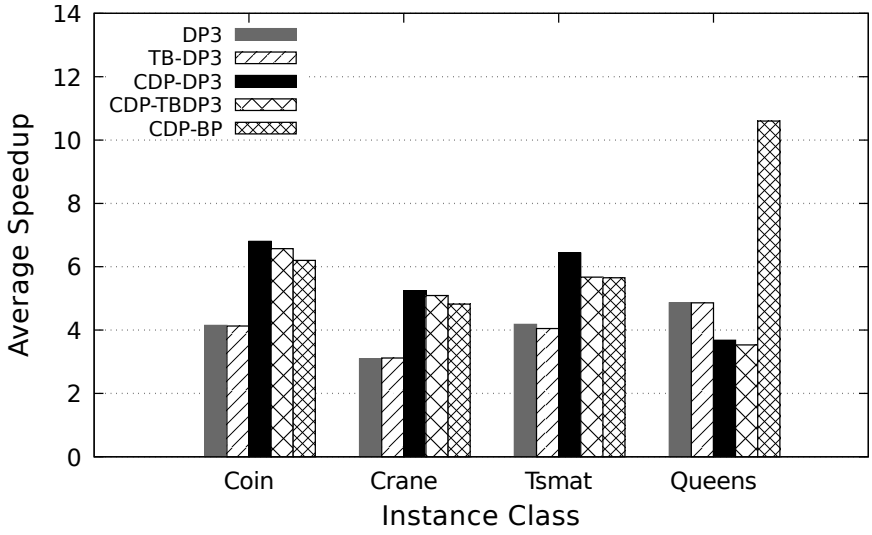
Figure 19 shows the average speedup reached by all DP3-based implementations for problem sizes ranging from $N = 10$ to 18(19). Table 12 presents the average speedup for each DP3-based implementation, for problem sizes ranging from $N = 10$ to 18(19). For this range of sizes, TB-DP3 and DP3 have an equivalent performance. Speedups of 4.04× and 4.07× are observed for DP3 and TB-DP3, respectively. In turn, speedups of 5.62× and 5.21× are observed for $CDP - DP3$ and $CDP - TBDP3$, respectively.

Table 11 – Average speedup reached by all DP3-based implementations. Results are the average speedup for all instance classes and sizes ranging from $N = 10$ to 12.

Implementation	Average Speedup
$DP3$	1.08×
$TB - DP3$	0.98×
$CDP - DP3$	2.10×
$CDP - TBDP3$	1.61×
$CDP - BP$	5.24×

Source – The Author.

Figure 19 – Average speedup reached by all DP3-based implementations for each instance class. Problem sizes range from $N = 10$ to 18(19).



Source – The Author.

Table 12 – Average speedup reached by all DP3-based implementations. Results are the average speedup for all instance classes and sizes ranging from $N = 10 - 18(19)$.

Implementation	Average Speedup
$DP3$	4.07×
$TB - DP3$	4.04×
$CDP - DP3$	5.62×
$CDP - TBDP3$	5.21×
$CDP - BP$	7.10×

Source – The Author.

A DP3-named implementation differs only from its TB-named counterpart in the way the dynamic allocation is performed. Both strategies create the same number of streams, and launch the same number of kernels, as shown in Table 9. According to Figure 19 and Table 12, all implementations that allocate memory in a thread-based manner are on average slower than their block-based counterparts. However, such a performance difference lies mainly for smaller instances. Solving larger problems by using a DP3-based

implementation requires launching thousands of kernels, as can be seen in Annex C. As blocks may have inactive threads, the heap of a DP3-named implementation needs to be up to 1.5× bigger than the heap of its TB-DP3-named counterpart. This way, a DP3-named implementation may allocate more memory for solving bigger problems, which explains the similar performance between DP3-named and TB-DP3 named implementations when taking into account all problem sizes.

4.6 Concluding remarks

This section presents the concluding remarks of this chapter. Sections 4.6.1 to 4.6.2 outline the conclusions concerning programmability, performance, and applicability of the DP3-based implementations. Section 4.6.3 introduces future directions of research. Finally, Section 4.6.4 lists the main insights that summarize this chapter.

4.6.1 Programmability

The objectives of the chapter were accomplished. All DP3-based implementations can solve the selected test-cases without runtime errors by using Algorithms 8 to 11.

Although removing the need for d_{cpu} tuning, the challenges faced while programming the CDP-BP strategy are amplified while programming the DP3-based ones. The memory requirement analysis requires several steps and takes into account a subtree rooted at d_{cpu} that goes down to the base depth. Moreover, there is the need of calculating the number of kernel generations and setup CUDA runtime variables. Furthermore, functions for getting the next depth and the base of the recursion are also required.

The memory requirement analysis based on $expected\_children\_d_{gpu}$ is not precise for DP3-named algorithms. A block may contain several inactive threads, and a block-based allocation takes into account such inactive threads. The experiments carried out to build Table 10 were performed based on trial and error, increasing the value of the variable `cudaLimitMallocHeapSize`, until the heap gets large enough for solving all test-cases.

The block size is a parameter that influences not only the performance, but the safety of the code of a DP3-named application. In a situation where a DP3-based strategy is made available for non-expert users, the programmer is not aware of the instances to be solved by the user. This way, the code may run correctly for the test-cases of the programmer, but, may present runtime errors for the user.

The thread-based allocation makes the memory requirement analysis more precise and removes the need for thread-to-data mapping and block synchronization, which considerably decrease the complexity of the kernel. Furthermore, it makes the code safer, as stated above.

The CDP programming model has a limit of 24 nesting levels. Thus, recursive CDP-based algorithms that launch several kernel generations, such as the quick-sort of the CUDA samples, launch new generations of kernels until a predefined base, avoiding to trespass this limit. For a DP3-based strategy, as it was originally proposed, this limit is not reachable. It is enough to go until $d_{gpu} = 2^{23}$.

4.6.2 Performance and applicability

The number of calls that a TB-named implementation makes to `malloc()` is more significant than the number of calls that a DP3-named makes. However, the thread-based allocation results in insignificant performance losses compared to the block-based one.

DP3 was initially proposed as a CDP-based load balance strategy for backtracking. However, it is difficult to make it launch more than three generations of kernels, due to the massive memory requirement of more deep depths. The *pool* strategy presented in the last chapter looks more worthwhile for load balance: There is no need for CDP expertise, and it is a straightforward modification of BP-DFS, which is the simplest and fastest GPU-based implementation proposed in this work.

The algorithms proposed in this chapter are not only for recursive approaches. They have also been used in the CDP-DP3 named algorithms, which have an implementation of CDP-BP internally.

4.6.3 Future research directions

It was pointed out that doubling d_{gpu} at each new recursive call of DP3 works well for sizes up to $N = 16$. In such situations, 3 generations of kernels are launched. For sizes bigger than $N = 16$, this strategy is prohibitive due to the massive memory requirements involved. Moreover, the definition of a base depth is also challenging: for $N = 18$, 4 generations of kernels would be launched. However, it would require an enormous amount of memory to store the possible children nodes at $d_{cpu} = 16$, and the last generation would perform no search at all. For example, consider instance $tsmat18$, that belongs to the instance class with the biggest solution space (refer to Table 1). In such a situation, the last kernel generation would evaluate less than 3% of the solution space. A future research direction is on investigating different ways of calculating the next d_{gpu} and a rule for determining the base depth based on the partial backtracking tree and the size N of the problem.

4.6.4 Main insights

The following summarizes the main insights from this chapter:

- Despite removing the need of d_{cpu} tuning, the use of the DP3 strategy makes the code complex.
- Setting up the CUDA runtime variables is not straightforward. It requires several algorithms that take into consideration the subtree of a node at d_{cpu}.
- The per-thread dynamic memory allocation makes the memory requirement analysis more precise and the code safer.
- Despite the higher number of calls to `malloc()` and `free()`, the thread-based strategy results in minor performance losses for smaller instances.

5 RECURSIVE NON-CDP IMPLEMENTATIONS

Before the advent of dynamic parallelism, it was necessary to return the control to the host to launch a new kernel. Thus, one of the primary goals of dynamic parallelism is to avoid halting the kernel execution to return the control to the host (NVIDIA, 2012a; ADINETZ, 2014; AMD, 2016). All CDP-based algorithms studied in this thesis can be implemented without using CDP, recursively invoking CUDA kernels from the host. Differently from related works that apply dynamic parallelism to replace several kernel calls from the host (ODEN; KLENK; FRONING, 2014; AMD, 2016), the CDP-based implementations studied in this work launch few kernel generations. Under these circumstances, this chapter reports an investigation into whether the use of CDP is advantageous over a non-CDP and equivalent counterpart.

This chapter presents two recursive non-CDP implementations: REC-CDP, which has the semantics of CDP-BP and DP2, and REC-DP3, which has the semantics of DP3. According to the results, smaller interference of the host combined with a block-based child search seems worthwhile for irregular tree search algorithms, even in situations where few kernels are launched.

The main contributions reported in this chapter are the following:

- This chapter shows that despite the performance penalties intrinsically related to dynamic parallelism, a CDP-based algorithm can be superior to its equivalent non-CDP counterpart.
- Results evidence that CDP-BP is less dependent on the size of the solution space than BP-DFS and all other CDP-based implementations.

The remainder of this chapter is structured as follows. Sections 5.1.1 and 5.1.2 detail the recursive non-CDP implementations. Next, both recursive implementations are compared to their CDP-based counterparts in Section 5.2. Finally, concluding remarks are outlined in Section 5.3.

5.1 Recursive non-CDP implementations

The following subsections detail the non-CDP implementations: REC-CDP and REC-DP3. REC-CDP has the semantics of CDP-BP and DP2, and performs the Intermediate GPU Search from d_{cpu} to d_{gpu}. After this stage, BP-DFS is called as the final GPU search. In turn, REC-DP3 has the semantics of DP3. However, differently from DP3, the host performs the allocation before the kernel launch and deallocations after.

5.1.1 REC-CDP

REC-CDP applies all algorithms and data structures used by CDP-BP. The biggest difference between them is that REC-CDP is not a block-based search strategy. This way, REC-CDP defines a global counter $survivors\_d_{gpu}$ instead of a block counter. The variable $survivors\_d_{gpu}$ is atomically incremented in the first kernel every time a GPU thread finds a frontier node at d_{gpu}. Thus, the use of $survivors\_d_{gpu}$ places frontier nodes in contiguous positions of A^d_{gpu}.

Algorithm 14 presents a pseudo-code for REC-CDP. This algorithm is similar to Algorithm 4, which launches the Intermediate GPU Search for CDP-BP. Therefore, for the sake of greater simplicity, some steps and details concerning the parameters have been omitted. The algorithm for REC-CDP proceeds as follows.

First, the host initializes $survivors\_d^d_{gpu}$ on GPU (*line* 7). Then, the Intermediate GPU Search evaluates a subset $S \subseteq A^h_{cpu}$ of size *chunk* until A^h_{cpu} is empty. After the first kernel (*line* 9), the value of $survivors\_d_{gpu}$ is retrieved by the host (*line* 10) to configure the Final GPU Search launch (*lines* 16 – 17). There is no need to retrieve A^d_{gpu} because the host uses this pointer on the launch of the last kernel (*line* 17). After the Final GPU Search, the algorithm retrieves the control data from the device, checks for errors, and calculates metrics.

Figure 20 illustrates the search procedure of REC-CDP. It is possible to observe that the control returns to the host when the first kernel finishes. This action is necessary to retrieve $suvivors\_d_{gpu}$, which stores the size of A^d_{gpu}, and to configure the next kernel launch. Comparing Figure 11 to Figure 20, it is possible to see that CDP-BP performs the same operations REC-CDP does, but without returning the control to host after the Intermediate GPU Search, and in a block-based manner.

5.1.2 REC-DP3

Algorithm 15 shows a pseudo-code for REC-DP3. Although this algorithm has the semantics of DP3, REC-DP3 uses the code of REC-CDP, launching the Intermediate GPU Search until it reaches the recursion base. REC-DP3 also perform a memory requirement analysis. However, these steps have been omitted in Algorithm 15, as well as information concerning parameters.

First, the host initializes *current_depth* with the value of d_{gpu} and sets the variable $first$ to *true* (*lines* 1 – 2). After these steps, the recursion begins (*line* 4). Initially, d_{gpu} receives the next depth of the recursion, and then the algorithm calculates the size of A^d_{gpu} for the current d_{gpu} as shown in Section 3.3 (*lines* 5 – 7).

Before launching the Intermediate GPU Search, the host initializes $survivors\_d^d_{gpu}$ of the current kernel call and allocates memory enough for A^d_{gpu} (*lines* 13 – 14). Note

Algorithm 14: Launching REC-CDP.

```
1  counter ← 0
2  survivors_d^h_gpu ← 0
3  remaining ← survivors_d_cpu
4  while counter < survivors do
5  |  nt ← get_block_size()
6  |  nb ← ⌈chunk/nt⌉
7  |  cudaMemCpy(survivors_d^d_gpu, survivors_d^h_gpu, sizeof(int), H2D)
8  |  ...
9  |  intermediate_GPU_search <<< nb, nt >>>
   |   (survivors_d^d_gpu, C^d, chunk, expected_children_d_gpu, A^d_cpu, A^d_gpu, d_cpu, d_gpu, upper_bound)
10 |  cudaMemCpy(survivors_d^h_gpu, survivors_d^d_gpu, sizeof(int), D2H)
11 |  counter ← counter + chunk
12 |  remaining ← remaining − chunk
13 |  if remaining < chunk then
14 |  |  chunk ← remaining
15 |  end
   |  /* Configuration of the final GPU search.                    */
16 |  nt ← get_final_block_size()
17 |  nb ← ⌈survivors_d^h_gpu/nt⌉
18 |  final_GPU_search <<< nb, nt >>>
   |   (survivors_d^h_gpu, C^d, A^d_gpu, d_gpu, upper_bound)
19 end
```

that there is only one $H2D$ transference that copies A^h_{cpu} on GPU. This data transference happens only for the first kernel launch (*lines* 7 – 11) because the A^d_{gpu} of launch x corresponds to the A^d_{cpu} of the launch $x+1$, as will be detailed further.

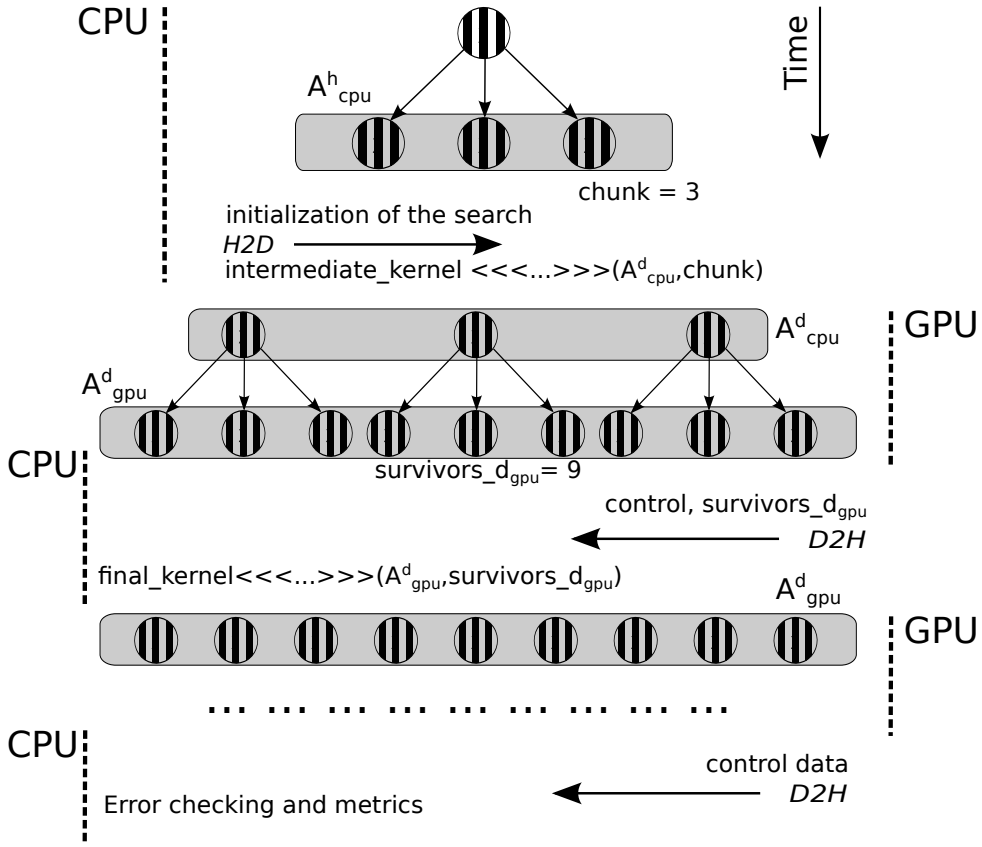
After the kernel call, there is a $D2H$ copy that gets $survivors\_d^d_{gpu}$ (*line* 18). Then, the host deallocates A^d_{cpu} and swap pointers (*lines* 19 – 21). This way, A^d_{cpu} of the current kernel call points to A^d_{gpu} of the former one. When the search reaches the base of the recursion, the host launches the Final GPU Search using the pointers to A^d_{gpu} and $survivors\_d^d_{gpu}$ (*lines* 24 – 26).

Consider Figure 20 for illustrating how REC-DP3 proceeds. This algorithm performs the operations recursively from the configuration of the Intermediate GPU Search to the D2H transference that retrieves $survivors\_d^d_{gpu}$. When REC-DP3 reaches the recursion depth, the host configures and launches the Final GPU Search.

5.2 Performance Evaluation

This section evaluates the CDP-based backtracking strategies proposed in Sections 5.1.2 and 5.1.1. Firstly, Section 5.2.1 presents the experimental protocol. Then,

Figure 20 – This figure illustrates the steps of REC-CDP. After the initial CPU search, the host initializes data on GPU and deploys the Intermediate GPU Search, which fills A^d_{gpu} with frontier nodes. Next, the host retrieves $survivors\_d_{gpu}$ to launch the final backtracking on GPU. After this kernel, control data is retrieved to calculate metrics and check for errors. Dashed lines are illustrative and do not mean the time spent on an operation.



Source – The Author.

Section 5.2.2 lists the parameter settings. Finally, all parallel implementations are compared in Section 5.2.3.

5.2.1 Experimental Protocol

The present performance evaluation compares different approaches for solving to optimality instances of the ATSP and for enumerating all valid configurations of the N-Queens problem. The strategies proposed in this chapter are the following:

- **REC-CDP**: implementation detailed in Algorithm 14, which has the semantics of CDP-DP and DP2.
- **REC-DP3**: implementation detailed in Algorithm 15, which has semantics of DP3.

Both parallel strategies listed above use the data structure described in Section 2.5.1.

Algorithm 15: REC-DP3

```
1  current_depth = d_cpu
2  first ← true
3  chunk ← survivors_d_cpu
4  while (current_depth ≠ recursion_base(N)) do
5      d_gpu ← get_next_depth(current_depth, N)
6      expected_children_d_gpu ← max_gpu / max_current_depth
7      max_A_gpu_size ← chunk × expected_children_d_gpu
8      if first then
9          cudaMalloc(A^d_cpu, sizeof(Node) * chunk)
10         cudaMemCpy(A^d_cpu, A^h_cpu, chunk * sizeof(Node), H2D)
11         first ← false
12     end
13     cudaMalloc(A^d_gpu, max_A_gpu_size * sizeof(Node))
14     cudaMemCpy(survivors_d^d_gpu, survivors_d^h_gpu, H2D)
15     nt ← get_block_size()
16     nb ← ⌈survivors_d^h_cpu / nt⌉
17     intermediate_GPU_search <<< nb, nt >>>
         (survivors_d^d_gpu, C^d, chunk, expt_children_d_gpu, A^d_cpu, A^d_gpu, current_depth, d_gpu)
18     cudaMemCpy(survivors_d^h_gpu, survivors_d^d_gpu, sizeof(int), D2H)
19     chunk ← survivors_d^h_gpu
20     pointer_to_release ← A^d_cpu
21     A^d_cpu ← A^d_gpu
22     cudaFree(pointer_to_release)
23 end
   /* Configuration of the final GPU search.                          */
24 nt ← get_final_block_size()
25 nb ← ⌈survivors_d^h_gpu / nt⌉
26 final_GPU_search <<< nb, nt >>>
     (survivors_d^d_gpu, C^d, A^d_cpu, d_gpu, upper_bound)
```

For comparison, the present performance evaluation considers all backtracking strategies already introduced in Section 3.8.1, as well as the same experimental protocol detailed in Section 3.8.1. Table 13 presents the key differences of all GPU-based implementations.

Table 13 – 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. Values are for ATSP and N-Queens. Numbers in brackets correspond to the explanations below the table.

Implementation	*CDP*	*GPU streams/CDP kernels*	*Dynamic Allocation*	*Algorithms*
$BP-DFS$	*no*	-	*no*	1 + 7
$DP3$	*yes*	$\|A^h_{cpu}\| + k_1$	*yes*	8 - 11 + DP3
$DP2$	*yes*	$\|A^h_{cpu}\|$	*yes*	DP2
$CDP-BP$	*yes*	$nb_h * number\_of\_kernels$	*no*	2 - 7
$CDP-DP3$	*yes*	$nb_h + k_1$	*yes*	5 - 7 + 8 - 11 + DP3
$REC-DP3$	*no*	-	*yes*	5 + 7 + 15 + DP3
$REC-CDP$	*no*	-	*no*	5 + 7 + 14

Source – The Author.

5.2.2 Parameters Settings

The present performance evaluation follows the parameters settings of Section 3.8.2 with respect to the programming languages, metrics, instances, and test-bed used. For both recursive implementations, there is no setup of CUDA runtime variables. REC-DP3 is based on DP3. Therefore, there is no tune of cutoff depth. Table 14 presents the best parameter configurations for all GPU-based implementations.

Table 14 – List of best parameters found experimentally for all parallel implementations.

	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	-	-
$REC-CDP$[1]	128	-	4	7
$REC-DP3$	128	-	-	-
$Multicore$[1]	-	-	4	-

[1)] The parameter are the same for ATSP and N-Queens.
Source – The Author.

5.2.3 Comparison Between All GPU-based implementations

In this section, all parallel implementations are compared using the parameter configuration of Table 14. Figure 21 shows the average speedup reached by all parallel implementations compared to the serial baseline. Table 15 reports the average speedup

achieved by all parallel implementations for different ranges of sizes. The *higher*/*lower* column presents the value of the best average speedup observed for a range of sizes over the worst one. For example, $BP - DFS$ has this value equals to 2.12: 12.37 (*higher*) / 5.82 (*lower*).

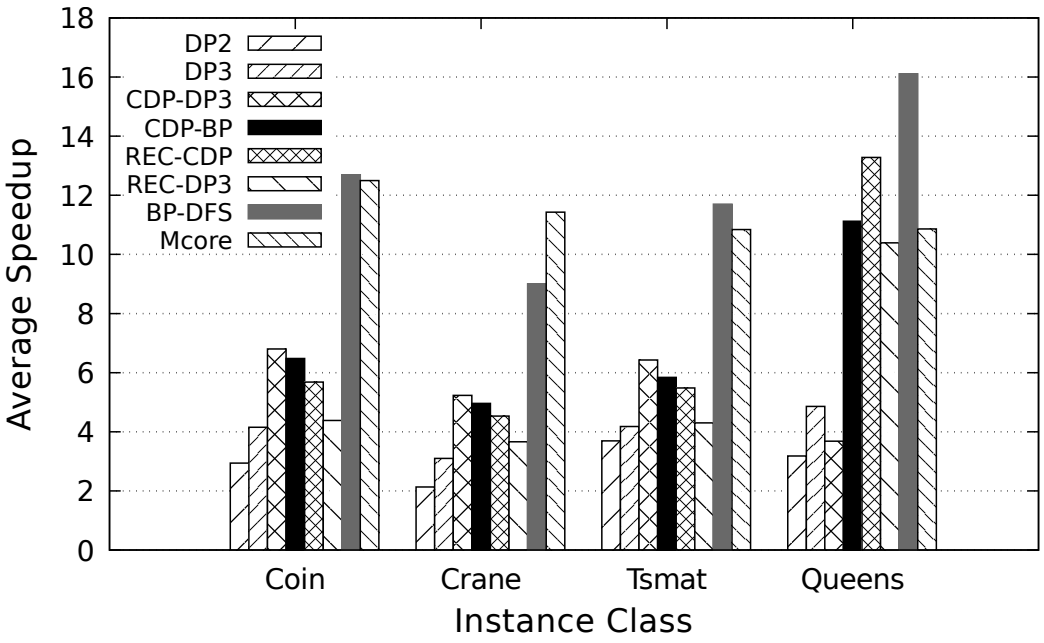


Figure 21 – Average speedup reached by all parallel implementations compared to the serial baseline. Results are considering all classes of instances. Problem sizes are ranging from $N = 10 - 19$ ($10 - 18$ for *queens* and *tsmat*).

According to Table 15, all implementations increase the speedup as the solution space grows. The highest values of *higher*/*lower* average speedup are observed for DP2, DP3, and CDP-DP3. These applications are CDP-based, perform dynamic allocations/deallocations on GPU, and launch one new kernel generation per thread. As observed in Section 3.8.3, the overhead of dynamic allocation, multiple stream creations/destruction, and several kernel launches amount negatively for small sizes. This way, the lowest values of average speedup are also observed for DP2, DP3, and CDP-DP3. As the solution space grows, this overhead becomes less significant, and the benefits of a more regular load provided by the DP3 strategy take place. Hence, according to Figure 21, CDP-DP3 is the CDP-based implementation with the best overall results while solving instances of the ATSP to optimality. It is also on average slightly superior to CDP-BP to perform this task.

CDP-BP presents the second smallest *higher*/*lower* value of Table 15. For sizes ranging from $N = 10$ to 12, CDP-BP and BP-DFS have similar performances. It is important to point out that both implementations are on their best parameters configuration. Such a small value of *higher*/*lower* confirms that the performance of CDP-BP is less dependent on the solution space size than BP-DFS and all other CDP-based

implementations.

Table 15 – Average speedup reached by all parallel implementations for different range of sizes: $N = 10 - 12$, $N = 10 - 15$, and $N = 10 - 18(19)$. The column *higher/lower* shows, among the three range of sizes, the value of the highest average speedup over the lowest one.

	Average Speedup			
Implementation	$10 - 12$	$10 - 15$	*All sizes*	*higher/lower*
$BP - DFS$	5.82×	10.84×	12.37×	2.12
$CDP - BP$	5.24×	6.81×	7.10×	1.35
$DP2$	1.3×	2.28×	2.99×	2.30
$DP3$	1.08×	3.14×	4.07×	3.76
$CDP - DP3$	1.87×	4.88×	5.54×	2.96
$REC - CDP$	6.05×	7.18×	7.24×	1.19
$REC - DP3$	3.48×	5.39×	5.68×	1.63
$Multicore$	4.63×	9.67×	11.41×	2.46

Source – The Author.

Despite the performance penalties intrinsically related to dynamic parallelism, a CDP-based implementation can be faster than its non-CDP and equivalent counterpart, even in the situations analyzed, where the control is returned few times to the host. CDP-BP is superior to both REC-CDP and REC-DP3 for solving the ATSP, which means that a smaller interference of the host combined with a block-based child search seems worthwhile for irregular tree search algorithms. Concerning the implementations that perform dynamic allocations, DP3 is not superior to REC-DP3 in any experiment. As pointed out before, DP3 has several sources of overhead.

Launching one new kernel generation for each thread is not a good strategy for the N-Queens problem. The load processed by a child kernel is too small (refer to Figure 8), and the problem faced by BP-DFS while handling small loads takes place (refer to Section 3.8.4). Both recursive implementations launch the next kernel based on the number of survivors returned by the former one, which is closely related to the load balance strategy proposed by Jenkins *et al.* (2011). That is the reason why the recursive implementations are superior to their CDP-based counterparts for enumerating all feasible solutions of the N-Queens. Speedups observed for REC-CDP are close to the ones observed for BP-DFS and superior to the ones reached by the multicore implementation that uses the pool strategy for load balance. In turn, REC-DP3 is considerably faster than its CDP-based counterpart for the N-Queens. However, it is slower than REC-CDP to perform the same task. REC-DP3 also has several sources of overhead: it allocates memory for the next kernel generation and also deallocates when a kernel finishes its execution. Moreover, it is based on DP3, which follows a predetermined rule for updating d_{gpu}. This way, it not possible to tune this parameter, which may penalize the performance of $REC - DP3$ and $DP3$.

CDP-BP launches a new kernel generation based on the load of the block instead

of the load of the thread, which makes the implementation of CDP-BP more similar to REC-CDP than to DP3 (refer to Section 5.1.1). 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 problem. The results of CDP-BP for the N-Queens and instances of sizes ranging from $N = 10 - 12$ justifies its better values than the ones of CDP-DP3 in Table 15.

5.3 Concluding remarks

This section presents the concluding remarks of this chapter. Sections 5.3.1 to 5.3.3 outline the conclusions concerning programmability, performance, and applicability of the recursive implementations. Section 5.3.3 introduces future lines of research. Finally, Section 5.3.4 lists the main insights that summarize this chapter.

5.3.1 Programmability

The recursive non-CDP implementations have the semantics of the CDP-based ones. However, they do not face the limitations of the CDP programming model and do not suffer from the different performance penalties intrinsically related to dynamic parallelism. Moreover, the load balance and higher granularity provided by calling multiple kernel generations are also present in both REC-CDP and REC-DP3. Furthermore, both recursive implementations do not require thread-to-data mappings, which considerably simplifies the coding process.

Although REC-DP3 is based on DP3, it does not require algorithms to set up the maximum heap size or maximum nesting depth. The most significant challenging issue of a DP3-based search is the memory requirement. Launching multiple kernels generations means dealing with increasing memory requirements. Compared to the memory requirement analysis of DP3, the analysis of REC-DP3 is more precise and straightforward, as it uses the number of survivors returned by the previous kernel call to calculate the requirements of the next one.

CDP-DP3 is the CDP-based application with the best overall performance for the ATSP. However, its code is complex and consists in a combination of several algorithms. As a consequence, the programmer needs to cope with programming challenges of CDP-BP and DP3 at the same time. This programming effort seems not worthwhile for such a small performance gain.

5.3.2 Performance and applicability

CDP-BP is faster than its non-CDP and equivalent counterpart for solving the ATSP. These results evidence that a smaller interference of the host combined with a block-based child search seems worthwhile for irregular tree search algorithms. Results also report that CDP-BP is less dependent than BP-DFS on the size of the solution space.

Using the recursive implementation is beneficial over using CDP in situations where the number of survivors generated by the previous kernel is much smaller than the expected number. In such circumstances, the next kernel generation is launched with a small number of active threads, and the resources of the GPU are not adequately used. The non-CDP implementations deploy a new kernel generation based on the survivors found by the previous kernel. This load balancing is similar to the one applied by Jenkins *et al.* (2011) and found to be useful in this situation.

5.3.3 Future research directions

REC-DP3 seems a good starting point to implement a hybrid GPU-multicore algorithm. Such an algorithm would call the multicore search if the number of survivors returned by the previous kernel call is less than a threshold. However, the way DP3 calculates the next depth does not allow such an implementation. Doubling the next d_{gpu} makes the memory requirements huge, while the number of nodes rapidly decreases. This way, this hybrid application would have to provide a different way of choosing the next d_{gpu}.

5.3.4 Main insights

The following summarizes the main insights from this chapter:

- CDP-based algorithms may be superior to their equivalent non-CDP counterparts. However, the CDP-based code is more complex and must deal with hardware and programming model limitations.
- CDP-BP is less dependent on the size of the solution space than BP-DFS and all other CDP-based implementations.
- A smaller interference of the host combined with a block-based child search seems worthwhile for irregular tree search algorithms.
- REC-DP3 looks good starting point to program hybrid multicore-GPU algorithms.

6 CONCLUSION

This work presents several GPU-based backtracking strategies for solving permutation combinatorial problems. The first one, called BP-DFS, revisits GPU-DFS and it is based on a well-known parallel backtracking model. Besides several improvements, BP-DFS uses bitsets to keep track of the elements of the permutation that have already been used. The modifications are straightforward and mean no complexity to the code. Although BP-DFS was not presented as a contribution of this work, the new data structure was used as the foundation for all other algorithms presented by this thesis.

The main contribution of this thesis consists in a study on GPU-accelerated parallel backtracking strategies that uses CUDA Dynamic Parallelism (CDP). Although CUDA programming model was the one chosen for parallelizing the algorithms, dynamic parallelism is not a proprietary technology. It is present in OpenCL 2.0 and supported by the accelerators of NVIDIA's concurrence. The kind of study herein reported is highly needed to evaluate experimentally the efficiency of the mechanisms, such as dynamic parallelism, provided in CUDA/GPU computing. This thesis identifies situations where it is advantageous to use CDP through a large and irregular application. The present research also allows identifying the limitations regarding the CDP programming model and the considered applications: backtracking algorithms for solving instances of the ATSP and enumerating all feasible configuration of the N-Queens problem.

According to the experimental results, using CDP provides a better worst-case performance and CDP-BP reaches speedups over its sequential counterpart even without prior knowledge of calibration results. However, all CDP-based implementations have been outperformed by BP-DFS with well-tuned parameters. Despite removing the overhead of dynamic allocations, CDP-BP implementation still suffers from the cost imposed by dynamically launched kernels. Iterative kernel calls may replace the use of CDP. Results report that a smaller interference of the host combined with a block-based child search seems worthwhile for irregular tree search algorithms. On the other hand, in situations where the load processed by the child kernels is small, a non-CDP version of the algorithm is a better option.

This research also identifies challenges in developing CDP-based applications. 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 such challenges. CDP suffers from several hardware limitations that may cause runtime failures, and some of these limitations are difficult to cope with. The present thesis identifies CDP limitations faced in the development of the proposed search strategies. For each one found, a way of dealing with it is presented. Therefore, all proposed algorithms

can solve problems with memory requirements from few kilobytes to several gigabytes. This work also brings conclusions on other aspects of the CDP programming model, such as the number of streams, CUDA runtime variables, dynamic allocation, global memory mapping, and block size. These conclusions may be useful for helping potential users of CDP or researchers that program irregular applications on GPU.

6.1 Future works

There are skeletons and frameworks designed to make GPU programming easier (ENMYREN; KESSLER, 2010; STEUWER; KEGEL; GORLATCH, 2011; MAJEED; DASTGEER; KESSLER, 2013). Using such frameworks to implement the search strategies studied in this thesis would not be worthwhile. It would be necessary to express these algorithms using the data parallel functions the frameworks provide. 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. There are also parallel divide-and-conquer skeletons (DORTA *et al.*, 2003; CUN; ROUCAIROL, 1995; GALEA; CUN, 2007; DANELUTTO *et al.*, 2016). These parallel tree search skeletons are mainly designed for shared and distributed memory systems. To use such skeletons, the user defines the problem and the solution data structures, as well as a set of functions, e.g., solution evaluation, termination criteria, problem initialization, and bounding function. The skeleton is the responsible for the parallel search procedure.

It is possible to extend the set of algorithms resulting from this thesis as a parallel backtracking skeleton for solving permutation combinatorial problems. Such a skeleton would consist of sequential, multi-threaded and GPU-based algorithms. There would be no need to define the problem and the solution data types because the problems are permutation-based. The user would have only to provide the termination criteria, problem instance, and bounding functions.

A contribution related to this thesis is a comparison between the IVM data structure and BP-DFS (PESSOA *et al.*, 2016). A natural extension of this work is combining BP-DFS and IVM. This new data structure would provide the low computing requirements of BP-DFS and the load balancing properties of IVM.

Xeon Phi (JEFFERS; REINDERS, 2013) is an Intel accelerator for massively parallel computing. This device has been attracting the attention of the combinatorial optimization community because it has a large number of independent cores and it can also be programmed using OpenMP (BARKALOV; GERGEL; LEBEDEV, 2015; MELAB *et al.*, 2015; LEROY, 2015). This scenario looks promising on the scope of unstructured tree search, because diverging flow instructions may not result in high-performance degradations,

like on GPUs. The experimental results report that the multi-core version of BP-DFS is scalable on the number of CPU cores. For this reason, another future work is to perform a comparison between Xeon Phi and GPU in the fine-grained situation investigated in this thesis.

Another research direction is to investigate the use of CDP for redesigning GPU-based Branch& Bound 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.

6.2 Main publications related to this Thesis

There are four main publications related to this thesis:

- Carneiro *et al.* (2014) is a systematic study of the literature that investigates the state-of-the-art on solving combinatorial problems by using GPUs and other heterogeneous architectures. This paper was published in the annals of the *XXXV Ibero-Latin American Congress on Computational Methods in Engineering* (CILAMCE). This work was used by the author to define the research directions of his thesis.

- Pinheiro *et al.* (2014) present the FUSION language, a programming language for heterogeneous computing in Java. This work uses as a case-study a multi-streaming version of GPU-DFS. For this purpose, GPU-DFS was rewritten in FUSION. The *Brazilian Symposium on Programming Languages* (SBLP) is a B3-ranked conference, and the paper was published in the Lecture Notes in Computer Science.

- Pessoa *et al.* (2016) revisits the IVM data structure for load balancing on DFS-B&B algorithms and compares it to BP-DFS. Results show that BP-DFS is advantageous for more regular loads and smaller instances. However, the IVM-based algorithm performs load balance, which results in a better performance for bigger and more irregular loads. The *International Conference on Algorithms and Architectures for Parallel Processing* (ICA3PP) is a B1-ranked conference, and the paper was published in the Lecture Notes in Computer Science.

- Carneiro Pessoa *et al.* (2017) is the final and most important publication related to this thesis. It contains information present in Chapters 3 and 5. This paper was submitted to *Concurrency and Computation: Practice and Experience* (CCPE), and will be published in the special issue *Emerging Trends in High-Performance Computing and Simulation*. CCPE is an A2-ranked journal.

The last two works above listed are the result of a collaboration between members of the ParGO-UFC group and researchers of INRIA-Lille (France) and Mons University (Belgium).

6.2.1 Other publications

The author of this thesis published three other papers during his doctorate. These research works are the result of a collaboration between the author and members of the ParGO-UFC group, and between the author and researchers of other institutions: Fraunhofer Institute (Germany) and IFCE.

- Nepomuceno, Pessoa, and Nepomuceno introduce the Formula Optimizer: a new software designed to formulate and solve multi-objective combinatorial optimization problems with no programming expertise. This work was published in the annals of the *XLVIII Simpósio Brasileiro de Pesquisa Operacional*. SBPO is a B4-ranked conference.

- Arruda *et al.* (2014a) study the influence of L2-cache organizations on the performance of GPU-based algorithms. This work was published in the annals of the *XV Simpósio em Sistemas Computacionais de Alto Desempenho*. WSCAD is a B3-ranked conference.

- Arruda *et al.* (2014b) investigate the impact of different code optimizations on GPU-based algorithms. This paper was published in the annals of the *XXXV Ibero-Latin American Congress on Computational Methods in Engineering* (CILAMCE).

BIBLIOGRAPHY

ABRAMSON, B.; YUNG, M. Divide and conquer under global constraints: a solution to the N-Queens problem. **Journal of Parallel and Distributed Computing**, Elsevier, v. 6, n. 3, p. 649–662, 1989. Cited in page 28.

ADINETZ, A. **CUDA dynamic Parallelism: API and principles**. 2014. Available from Internet: <https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles/>. Cited 4 times in pages 18, 23, 25, and 80.

ALANDOLI, M.; AL-AYYOUB, M.; AL-SMADI, M.; JARARWEH, Y.; BENKHELIFA, E. Using dynamic parallelism to speed up clustering-based community detection in social networks. In: IEEE. **IEEE International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)**. 2016. p. 240–245. Cited in page 25.

ALBA, E.; TALBI, E.-G.; LUQUE, G.; MELAB, N. Metaheuristics and parallelism. **Parallel Metaheuristics: A New Class of Algorithms**, Wiley Online Library, p. 79–103, 2005. Cited in page 27.

ALIAGA, J.; DAVIDOVIĆ, D.; PÉREZ, J.; QUINTANA-ORTÍ, E. S. Harnessing CUDA dynamic parallelism for the solution of sparse linear systems. **Parallel Computing: On the Road to Exascale**, IOS Press, v. 27, p. 217, 2016.

AMD. **The AMD ROCm Implementation of OpenCL**. 2016. Accessed: 2017-10-10. Available from Internet: <http://rocm-documentation.readthedocs.io/en/latest/Programming_Guides/Opencl-programming-guide.html>. Cited 2 times in pages 26 and 80.

ARRUDA, N. G. P. B.; JUNIOR, F. H. de C.; CARNEIRO, T.; PINHEIRO, A. B. Análise de drawbacks no desdobramento de laços relativo a caches associativas de GPUs. In: **XV Simpósio em Sistemas Computacionais de Alto Desempenho - WSCAD 2014**. 2014.

ARRUDA, N. G. P. B.; JUNIOR, F. H. de C.; CARNEIRO, T.; PINHEIRO, A. B. Uma avaliação de técnicas de otimização de código aplicadas a aceleradores gráficos modernos. In: **XXXV Ibero-Latin American Congress on Computational Methods in Engineering**. 2014.

BARKALOV, K.; GERGEL, V.; LEBEDEV, I. Use of Xeon Phi coprocessor for solving global optimization problems. In: SPRINGER. **International Conference on Parallel Computing Technologies**. 2015. p. 307–318. Cited in page 91.

BELL, J.; STEVENS, B. A survey of known results and research areas for N-Queens. **Discrete Mathematics**, Elsevier, v. 309, n. 1, p. 1–31, 2009. Cited in page 28.

BITNER, J. R.; REINGOLD, E. M. Backtrack programming techniques. **Communications of the ACM**, ACM, v. 18, n. 11, p. 651–656, 1975. Cited 2 times in pages 17 and 29.

BLUM, C.; ROLI, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. **ACM Computing Surveys (CSUR)**, ACM, v. 35, n. 3, p. 268–308, 2003. Cited in page 27.

BRODTKORB, A.; DYKEN, C.; HAGEN, T.; HJELMERVIK, J.; STORAASLI, O. State-of-the-art in heterogeneous computing. **Scientific Programming**, IOS Press, v. 18, n. 1, p. 1–33, 2010. Cited in page 18.

BURTSCHER, M.; NASRE, R.; PINGALI, K. A quantitative study of irregular programs on GPUs. In: **IEEE International Symposium on Workload Characterization (IISWC)**. 2012. p. 141–151. Cited 2 times in pages 17 and 22.

CARNEIRO PESSOA, T.; GMYS, J.; de CARVALHO JÚNIOR, F. H.; MELAB, N.; TUYTTENS, D. GPU-accelerated backtracking using CUDA dynamic parallelism. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, p. e4374–n/a, 2017. ISSN 1532-0634. Available from Internet: <http://dx.doi.org/10.1002/cpe.4374>.

CARNEIRO, T.; ARRUDA, N. G. P. B.; JUNIOR, F. H. de C.; PINHEIRO, A. B. Um levantamento na literatura sobre a resolução de problemas de otimização combinatória através do uso de aceleradores gráficos. In: **XXXV Ibero-Latin American Congress on Computational Methods in Engineering**. 2014.

CARNEIRO, T.; MURITIBA, A.; NEGREIROS, M.; CAMPOS, G. d. 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. p. 41–47. ISSN 1550-6533. Cited 4 times in pages 17, 29, 31, and 49.

CARNEIRO, T.; MURITIBA, A. E. F.; NEGREIROS, M.; CAMPOS, G. A. L. de. Solving ATSP hard instances by new parallel branch and bound algorithm using GPGPU. In: **XXXII Iberian Latin American Congress on Computational Methods in Engineering**. 2011. v. 1. Cited 2 times in pages 29 and 31.

CARNEIRO, T.; NOBRE, R. H.; NEGREIROS, M.; CAMPOS, G. A. L. de. Depth-first search versus Jurema search on GPU branch-and-bound algorithms: A case study. **NVIDIA's GCDF - GPU Computing Developer Forum on XXXII Congresso da Sociedade Brasileira de Computação (CSBC)**, 2012. ISSN 2175-2761. Cited 4 times in pages 29, 31, 32, and 49.

CHAKROUN, I.; BENDJOUDI, A.; MELAB, N. Reducing thread divergence in GPU-based B&B applied to the flow-shop problem. In: SPRINGER. **International Conference on Parallel Processing and Applied Mathematics**. 2011. p. 559–568. Cited in page 27.

CIRASELLA, J.; JOHNSON, D.; MCGEOCH, L.; ZHANG, W. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. **Algorithm Engineering and Experimentation**, Springer, p. 32–59, 2001. Cited in page 28.

COOK, W. **In pursuit of the traveling salesman: mathematics at the limits of computation**. : Princeton University Press, 2012. Cited in page 28.

CORNUÉJOLS, G.; KARAMANOV, M.; LI, Y. Early estimates of the size of branch-and-bound trees. **INFORMS Journal on Computing**, INFORMS, v. 18, n. 1, p. 86–96, 2006. Cited in page 64.

CRAINIC, T. G.; CUN, B. L.; ROUCAIROL, C. Parallel branch-and-bound algorithms. **Parallel combinatorial optimization**, Wiley, v. 1, p. 1–28, 2006. Cited 2 times in pages 30 and 31.

CUN, B. L.; ROUCAIROL, C. **Bob: a unified platform for implementing branch-and-bound like algorithms**. 1995. Cited 2 times in pages 30 and 91.

DANELUTTO, M.; MATTEIS, T. D.; MENCAGLI, G.; TORQUATI, M. A divide-and-conquer parallel pattern implementation for multicores. In: ACM. **Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems**. 2016. p. 10–19. Cited 2 times in pages 30 and 91.

DEFOUR, D.; MARIN, M. Regularity versus load-balancing on GPU for treefix computations. **Procedia Computer Science**, Elsevier, v. 18, p. 309–318, 2013. Cited in page 17.

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. Cited in page 25.

DONGARRA, J.; GANNON, D.; FOX, G.; KENNEDY, K. The impact of multicore on computational science software. **CTWatch Quarterly**, Citeseer, v. 3, n. 1, p. 1–10, 2007. Cited in page 18.

DORTA, I.; LEÓN, C.; RODRÍGUEZ, C.; ROJAS, A. Parallel skeletons for divide-and-conquer and branch-and-bound techniques. In: IEEE. **Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on**. 2003. p. 292–298. Cited 2 times in pages 30 and 91.

ENMYREN, J.; KESSLER, C. W. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In: **Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications**. New York, NY, USA: ACM, 2010. (HLPP '10), p. 5–14. ISBN 978-1-4503-0254-8. Available from Internet: <http://doi.acm.org/10.1145/1863482.1863487>. Cited in page 91.

FATAHALIAN, K.; HOUSTON, M. A closer look at GPUs. **Commun. ACM**, ACM, New York, NY, USA, v. 51, p. 50–57, out. 2008. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/1400181.1400197>. Cited in page 18.

FEINBUBE, F.; RABE, B.; LöWIS, M. von; POLZE, A. NQueens on CUDA: Optimization issues. In: IEEE. **Ninth International Symposium on Parallel and Distributed Computing (ISPDC)**. 2010. p. 63–70. Cited 6 times in pages 17, 27, 28, 31, 32, and 51.

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. p. 361–367. Cited in page 28.

FISCHETTI, M.; LODI, A.; TOTH, P. Solving real-world atsp instances by branch-and-cut. **Lecture Notes in Computer Science**, Springer, v. 2570, p. 64–77, 2003. Cited in page 28.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE transactions on computers**, IEEE, v. 100, n. 9, p. 948–960, 1972. Cited in page 18.

GALEA, F.; CUN, B. L. Bob++: a framework for exact combinatorial optimization methods on parallel machines. In: **International Conference High Performance Computing & Simulation**. 2007. p. 779–785. Cited 2 times in pages 30 and 91.

GENDRON, B.; CRAINIC, T. G. Parallel branch-and-bound algorithms: Survey and synthesis. **Operations Research**, INFORMS, v. 42, n. 6, p. 1042–1066, 1994. ISSN 0030364X, 15265463. Available from Internet: <http://www.jstor.org/stable/171985>. Cited 2 times in pages 18 and 30.

GERMS, R.; GOLDENGORIN, B.; TURKENSTEEN, M. Lower tolerance-based branch and bound algorithms for the ATSP. **Computers & Operations Research**, Elsevier, v. 39, n. 2, p. 291–298, 2012. Cited in page 28.

GMYS, J.; MEZMAZ, M.; MELAB, N.; TUYTTENS, D. A GPU-based branch-and-bound algorithm using Integer–Vector–Matrix data structure. **Parallel Computing**, p. –, 2016. ISSN 0167-8191. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167819116000387>. Cited in page 34.

GOLOMB, S. W.; BAUMERT, L. D. Backtrack programming. **Journal of the ACM (JACM)**, ACM, v. 12, n. 4, p. 516–524, 1965. Cited 2 times in pages 17 and 29.

GRAMA, A.; KUMAR, V. A survey of parallel search algorithms for discrete optimization problems. **ORSA Journal on Computing**, Citeseer, v. 7, 1993. Cited 3 times in pages 18, 27, and 31.

GRAMA, A.; KUMAR, V.; GUPTA, A.; KARYPIS, G. **Introduction to parallel computing**. : Pearson Education, 2003. Cited in page 59.

IOFFE, R.; SHARMA, S.; STONER, M. Achieving performance with opencl 2.0 on intel® processor graphics. In: ACM. **Proceedings of the 3rd International Workshop on OpenCL**. 2015. p. 3. Cited 2 times in pages 25 and 26.

JARZĄBEK, Ł.; CZARNUL, P. Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. **The Journal of Supercomputing**, Springer, p. 1–24, 2017.

JEFFERS, J.; REINDERS, J. **Intel Xeon Phi coprocessor high-performance programming**. : Newnes, 2013. Cited in page 91.

JENKINS, J.; ARKATKAR, I.; OWENS, J. D.; CHOUDHARY, A.; SAMATOVA, N. F. Lessons learned from exploring the backtracking paradigm on the GPU. In: **Euro-Par 2011 Parallel Processing**. : Springer, 2011. p. 425–437. Cited 3 times in pages 17, 31, and 32.

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**, Springer, p. 445–487, 2004. Cited in page 28.

KARP, R. M.; ZHANG, Y. Randomized parallel algorithms for backtrack search and branch-and-bound computation. **Journal of the ACM (JACM)**, ACM, v. 40, n. 3, p. 765–789, 1993. Cited 3 times in pages 17, 27, and 30.

KARYPIS, G.; KUMAR, V. Unstructured tree search on SIMD parallel computers. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 5, n. 10, p. 1057–1072, 1994. Cited 3 times in pages 17, 31, and 51.

BOURD, A. (Ed.). **The OpenCL Specification**, v2.2-3. : Khronos OpenCL Working Group, 2017. Cited 2 times in pages 18 and 25.

KRAJECKI, M.; LOISEAU, J.; ALIN, F.; JAILLET, C. Many-core approaches to combinatorial problems: case of the langford problem. **Supercomputing frontiers and innovations**, v. 3, n. 2, p. 21–37, 2016. Cited in page 31.

LAPORTE, G. A short history of the traveling salesman problem. **Canada Research Chair in Distribution Management, Centre for Research on Transportation (CRT) and GERAD HEC Montréal, Canada**, 2006. Cited in page 28.

LAWLER, E.; WOOD, D. Branch-and-bound methods: A survey. **Operations research**, JSTOR, p. 699–719, 1966. Cited 2 times in pages 18 and 27.

LEROY, R. **Parallel Branch-and-Bound revisited for solving permutation combinatorial optimization problems on multi-core processors and coprocessors**. Thesis (Ph.D) — Université Lille 1, 2015. Cited 2 times in pages 27 and 91.

LI, D.; WU, H.; BECCHI, M. Nested parallelism on GPU: Exploring parallelization templates for irregular loops and recursive computations. In: IEEE. **44th International Conference on Parallel Processing (ICPP)**. 2015. p. 979–988. Cited 4 times in pages 17, 18, 23, and 25.

LI, L.; LIU, H.; WANG, H.; LIU, T.; LI, W. A parallel algorithm for game tree search using GPGPU. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 26, n. 8, p. 2114–2127, 2015. Cited 4 times in pages 17, 31, 32, and 33.

LOPEZ-ORTIZ, A.; SALINGER, A.; SUDERMAN, R. Toward a generic hybrid CPU-GPU parallelization of divide-and-conquer algorithms. In: IEEE. **IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)**. 2013. p. 601–610. Cited in page 31.

MAJEED, M.; DASTGEER, U.; KESSLER, C. Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters. In: The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). **Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)**. 2013. p. 468. Cited in page 91.

MEHTA, V.; ENGINEER, B. S. C. Exploiting CUDA dynamic parallelism for low power ARM based prototypes. In: **GPU Technology Conference, San Jose**. 2015. Cited in page 26.

MELAB, N.; LEROY, R.; MEZMAZ, M.; TUYTTENS, D. Parallel branch-and-bound using private IVM-based work stealing on Xeon Phi MIC coprocessor. In: IEEE. **2015 International Conference on High Performance Computing & Simulation (HPCS)**. 2015. p. 394–399. Cited in page 91.

MEZMAZ, M.; LEROY, R.; MELAB, N.; TUYTTENS, D. A multi-core parallel branch-and-bound algorithm using factorial number system. In: **28th IEEE International. Parallel & Distributed Processing Symposium (IPDPS)**. Phoenix, AZ: , 2014. p. 1203–1212. Cited in page 35.

MITTEN, L. Branch-and-bound methods: General formulation and properties. **Operations Research**, INFORMS, v. 18, n. 1, p. 24–34, 1970. Cited in page 27.

MUKHERJEE, S.; GONG, X.; YU, L.; MCCARDWELL, C.; UKIDAVE, Y.; DAO, T.; PARAVECINO, F. N.; KAELI, D. Exploring the features of opencl 2.0. In: ACM. **Proceedings of the 3rd International Workshop on OpenCL**. 2015. p. 5. Cited in page 25.

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. **ACM SIGPLAN Notices**. 1995. v. 30, n. 8, p. 68–79. Cited in page 17.

NEPOMUCENO, T. G.; PESSOA, T. C.; NEPOMUCENO, T. G. Formula optimizer: fast way to formulate and solve multi-objective combinatorial optimization problems. **XLVIII Simpósio Brasileiro de Pesquisa Operacional**, 2016.

NVIDIA. CUDA dynamic parallelism programming guide. **NVIDIA Corporation Whitepaper**, 2012. Cited 3 times in pages 18, 24, and 80.

NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110. **NVIDIA Corporation Whitepaper v1.0**, 2012. Cited 2 times in pages 18 and 23.

NVIDIA. CUDA C programming guide (version 8.0). **NVidia Corporation**, 2016. Cited 3 times in pages 22, 23, and 24.

NVIDIA. NVIDIA GeForce GTX 1080: Gaming perfected. **NVIDIA Corporation Whitepaper**, 2016. Cited 2 times in pages 18 and 23.

NVIDIA. NVIDIA Tesla P100: The most advanced datacenter accelerator ever built. featuring pascal GP100, the world's fastest GPU. **NVIDIA Corporation Whitepaper**, 2016. Cited 2 times in pages 18 and 23.

ODEN, L.; KLENK, B.; FRONING, H. Energy-efficient computing on distributed GPUs using dynamic parallelism and GPU controlled communication. In: **2nd Workshop on Energy-efficient SuperComputing (E2SC)**. 2014. Cited 2 times in pages 26 and 80.

PAPADIMITRIOU, C.; STEIGLITZ, K. **Combinatorial optimization: algorithms and complexity**. : Dover Pubns, 1998. Cited in page 27.

PESSOA, T. C.; GMYS, J.; MELAB, N.; JUNIOR, F. H. de C.; TUYTTENS, D. A GPU-based backtracking algorithm for permutation combinatorial problems. In: **Algorithms and Architectures for Parallel Processing**. : Springer International Publishing, 2016. p. 310–324. Cited 4 times in pages 29, 34, 51, and 91.

PESSOA, T. C.; GOMES, M. J. N. Jurema, a new branch & bound anytime algorithm for the asymmetric travelling salesman problem. **XLIII Simpósio Brasileiro de Pesquisa Operacional**, 2010. Cited 2 times in pages 34 and 50.

PINHEIRO, A. B.; JUNIOR, F. H. de C.; ARRUDA, N. G. P. B.; CARNEIRO, T. Fusion: Abstractions for multicore/manycore heterogenous parallel programming using GPUs. In: SPRINGER, CHAM. **Brazilian Symposium on Programming Languages**. 2014. p. 109–123.

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**, v. 6, n. 2, p. 212–229, 2016. Cited 7 times in pages 28, 31, 32, 40, 51, 53, and 64.

REINELT, G. Tsplib — a traveling salesman problem library. **ORSA journal on computing**, INFORMS, v. 3, n. 4, p. 376–384, 1991. Cited in page 28.

ROCKI, K.; SUDA, R. Parallel minimax tree searching on GPU. In: **Parallel Processing and Applied Mathematics**. : Springer, 2009. p. 449–456. Cited 3 times in pages 28, 31, and 33.

SEGUNDO, P. S.; ROSSI, C.; RODRIGUEZ-LOSADA, D. **Recent developments in bit-parallel algorithms**. : INTECH Open Access Publisher, 2008. Cited in page 36.

SIMPSON, J. E. Langford sequences: perfect and hooked. **Discrete Mathematics**, Elsevier, v. 44, n. 1, p. 97–104, 1983. Cited in page 34.

SOMERS, J. **The N-Queens problem: A study in optimization**. 2002. Accessed: 2016-09-03. Available from Internet: <http://jsomers.com/nqueen_demo/nqueens.html>. Cited 3 times in pages 28, 40, and 51.

STEINBERGER, M.; KENZEL, M.; KAINZ, B.; SCHMALSTIEG, D. Scatteralloc: Massively parallel dynamic memory allocation for the GPU. In: IEEE. **Innovative Parallel Computing (InPar), 2012**. 2012. p. 1–10. Cited in page 41.

STEUWER, M.; KEGEL, P.; GORLATCH, S. SkelCL: a portable skeleton library for high-level GPU programming. In: IEEE. **IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)**. 2011. p. 1176–1182. Cited in page 91.

STONE, J. E.; GOHARA, D.; SHI, G. OpenCL: A parallel programming standard for heterogeneous computing systems. **Computing in science & engineering**, IEEE, v. 12, n. 3, p. 66–73, 2010. Cited 2 times in pages 25 and 33.

STRNAD, D.; GUID, N. Parallel alpha-beta algorithm on the GPU. In: IEEE. **Proceedings of the 33rd International Conference on Information Technology Interfaces (ITI)**. 2011. p. 571–576.

THOUTI, K.; SATHE, S. Solving N-Queens problem on GPU architecture using OpenCL with special reference to synchronization issues. In: IEEE. **2nd IEEE International Conference on Parallel Distributed and Grid Computing (PDGC)**. 2012. p. 806–810. Cited 2 times in pages 28 and 51.

TURKENSTEEN, M.; GHOSH, D.; GOLDENGORIN, B.; SIERKSMA, G. Tolerance-based branch and bound algorithms for the ATSP. **European Journal of Operational Research**, Elsevier, v. 189, n. 3, p. 775–788, 2008. Cited in page 28.

WANG, J.; RUBIN, N.; SIDELNIK, A.; YALAMANCHILI, S. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs. In: ACM. **ACM SIGARCH Computer Architecture News**. 2015. v. 43, n. 3, p. 528–540. Cited in page 25.

WANG, J.; YALAMANCHILI, S. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In: IEEE. **2014 IEEE International Symposium on Workload Characterization (IISWC)**. 2014. p. 51–60. Cited in page 17.

WIDMER, S.; WODNIOK, D.; WEBER, N.; GOESELE, M. Fast dynamic memory allocator for massively parallel architectures. In: ACM. **Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units**. 2013. p. 120–126. Cited in page 41.

WOEGINGER, G. J. Exact algorithms for np-hard problems: A survey. **Lecture notes in computer science**, Springer, v. 2570, n. 2003, p. 185–207, 2003. Cited in page 27.

YANG, Y.; ZHOU, H. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. In: ACM. **ACM SIGPLAN Notices**. 2014. v. 49, n. 8, p. 93–106. Cited 3 times in pages 18, 23, and 25.

YELICK, K. A. Programming models for irregular applications. **ACM SIGPLAN Notices**, ACM, v. 28, n. 1, p. 28–31, 1993. Cited in page 17.

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: ACM. **Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms**. 2015. p. 11. Cited in page 63.

ZHANG, T.; SHU, W.; WU, M.-Y. Optimization of N-Queens solvers on graphics processors. In: SPRINGER. **International Workshop on Advanced Parallel Processing Technologies**. 2011. p. 142–156. Cited 4 times in pages 27, 28, 31, and 32.

ZHANG, W. **Branch-and-Bound Search Algorithms and Their Computational Complexity**. 1996. Cited 6 times in pages 17, 18, 27, 30, 40, and 61.

ZHANG, W. Depth-first branch-and-bound versus local search: A case study. In: MENLO PARK, CA; CAMBRIDGE, MA; LONDON; AAAI PRESS; MIT PRESS; 1999. **Proceedings of the National Conference on Artificial Intelligence**. 2000. p. 930–936. Cited 2 times in pages 27 and 30.

ZHANG, W.; KORF, R. Depth-first vs. best-first search: New results. In: JOHN WILEY & SONS LTD. **Proceedings of The National Conference On Artificial Intelligence**. 1993. p. 769–769. Cited in page 17.

APPENDIX A – GETTING THE AMOUNT OF MEMORY RESERVED FOR NESTING LEVEL SYNCHRONIZATION

CDP-capable GPUs reserve global memory for each nesting level that performs synchronization. This value is not easily obtained via CUDA functions and varies accordingly to properties of the GPU. In this annex brings details concerning an experiment to get the amount of memory a GPU reserves for each nesting level.

The experiment

Listing A.1 presents a simple CDP program that performs nesting level synchronization. This program receives an integer parameter to initialize the variable `cudaLimitDevRuntimeSyncDepth` (*line 7-8*). This variable keeps the value of the deepest synchronization level, host call included.

To find out the exact amount of memory that a CDP-capable GPU stores per nesting level, first, it is necessary to compile the program, then:

- Run the program passing an integer $x > 2$ as a parameter, and run NVIDIA's `nvidia-smi` in parallel, which returns the amount of global memory a CUDA program is using.
- Having the amount of memory the GPU stores for `cudaLimitDevRuntimeSyncDepth` set to x, run again the program passing $x + 1$ as a parameter along with `nvidia-smi` in parallel.

The difference between the memory reserved by both runs is the amount of memory the GPU reserves for each nesting level.

According to the results, the CUDA runtime reserves global memory even if `cudaLimitDevRuntimeSyncDepth` is set to 0. The CUDA runtime reserves at least memory for 2 nesting levels of synchronization, even if the programmer manually sets the value of this variable to 1. That is the reason why values bigger than 2 are used in this experiment. The program presented by Listing A.1 has a quick execution. Therefore, the infinite loop (*line* 11) is used to keep the CUDA program running while `nvidia-smi` is run in parallel.

This experiment was also run on four different GPUs that belong to three different NVIDIA architectures. One can see in Table 16 the specifications of the GPUs used, as well as the amount of memory reserved by each device.

```
1  __global__ void kernel(int a){
2          int b = a;
3          kernel<<<1,1>>>(b);
4          cudaDeviceSynchronize();
5  }
6  int main(int argc, char *argv[]){
7          int number_synch_depth = atoi(argv[1]);
8          cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth,
               number_synch_depth);
9          kernel<<<1,1>>>(1);
10         cudaDeviceSynchronize();
11         while(1){}
12
13         return 0;
14 }
```

Listing A.1 – A CDP program that performs nesting level synchronization and sets up the variable `cudaLimitDevRuntimeSyncDepth`. Source: The Author.

Table 16 – Specification of each GPU used, as well as the amount of memory reserved for each nesting level.

GPU	**Architecture**	**Compute capability**	**CUDA cores**	**Global memory**	**Reserved memory**
Tesla K40m	Kepler	3.5	2880	12 GB	109 MB
Tesla K20c	Kepler	3.5	2496	5 GB	96 MB
GTX-750 Ti	Maxwell	5.0	768	1 GB	41 MB
GTX-1050 Ti	Pascal	6.1	768	4 GB	48 MB

APPENDIX B – GETTING THE AMOUNT OF MEMORY RESERVED FOR THE DEFAULT HEAP SIZE

If the programmer does not sets up the CUDA runtime heap size, the GPU stores a default value, which is not easily obtained via CUDA functions. 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.

This annex shows details concerning an experiment to get the default CUDA runtime heap size.

The experiment

Listing B.1 presents a CUDA program that sets up the value of the `cudaLimitMallocHeapSize` variable and retrieves this value via the `cudaDeviceGetLimit()` function. To find out the exact value of the default CUDA Runtime heap size, it is necessary to run the program without line 6, which sets up the value of the heap size. Table 17 show the values of default heap size for the GPUs used in Annex A.

According to the results, the default heap size is the same for all GPUs: $8MB$. Even with the value of `cudaLimitMallocHeapSize` set to $0MB$, all GPUs used in the tests store at least $4MB$ of global memory for the heap.

```
1
2 int main(int argc, char *argv[]){
3         unsigned long max_heap = (unsigned long)(atoi(argv[1])
             *1000000UL);
4         cudaDeviceSetLimit(cudaLimitMallocHeapSize,max_heap);
5         cudaDeviceGetLimit(&max_heap_size,
             cudaLimitMallocHeapSize);
6         printf("\nHeap size found to be %d.",max_heap/1000000UL);
7         return 0;
8 }
```

Listing B.1 – A CUDA program that returns the default value of the `cudaLimitMallocHeapSize` variable. Source: The Author.

Table 17 – Specification of each GPU used, as well as the default CUDA Runtime default heap size.

GPU	**Architecture**	**Compute capability**	**CUDA cores**	**Global memory**	**Default Heap Size**
Tesla K40m	Kepler	3.5	2880	12 GB	8 MB
Tesla K20c	Kepler	3.5	2496	5 GB	8 MB
GTX-750 Ti	Maxwell	5.0	768	1 GB	8 MB
GTX-1050 Ti	Pascal	6.1	768	4 GB	8 MB

APPENDIX C – VISUAL PROFILE OF CUDA KERNELS

This annex presents a visual profile for all GPU-based implementations studied in this work. The objective of this annex is to provide a visual profile of all GPU-based implementations while processed by the GPU. This way, it is possible to show the details of each implementation, such as recursiveness, irregularity, the number of kernels launched, and the interleave of kernel launches/executions. All figures presented in this annex are screenshots of The NVIDIA Visual Profiler, which is a part of the CUDA C suit.

Visual Profile of All CDP-based implementations

Figure 22 shows a visual profile of all CDP-based implementation studied by this work: CDP-BP, DP2, DP3, and CDP-DP3. The TB named implementations differ from the non-TB ones only in the way they perform dynamic allocations. Hence, there are no visual profile for the TB-named ones.

The NVIDIA Visual Profiler shows different kernels with different colors. In Figures 22a and 22d the Intermediate GPU Search is in light blue, whereas the Final GPU Search employed is in purple. Moreover, it is possible to see that the first generation of kernels only finishes its execution after the execution of the kernels it has launched[1]. DP3 launches recursively three generations of the same kernel. This way, it is possible to see more clearly in Figure 22c that kernels of the third generation have an execution time smaller than kernels from the second one, and so on.

Visual Profile of CDP-BP with different parameters configurations

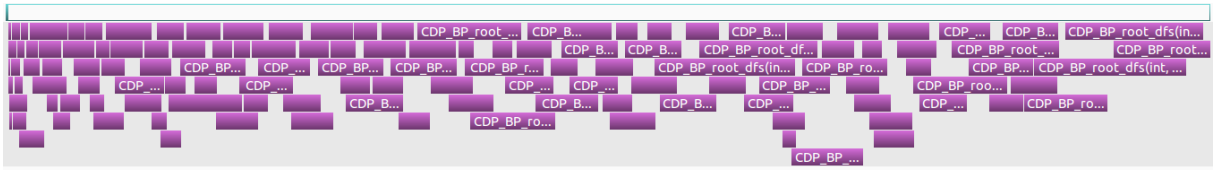
Figures 23a and 23b show a visual profile of CDP-BP with different configurations: the first one with cutoff depths $d_{cpu} = 6$ and $d_{gpu} = 8$; the second one with cutoff depths $d_{cpu} = 6$, $d_{gpu} = 8$, and launching four kernels per block. It is possible to see in Figures 23a and 23b the enormous number of kernels launched by using CDP. Compared to the execution time of the whole search (first light blue line), kernels from the second generation have a quick execution.

Visual Profile of all non-CDP implementations

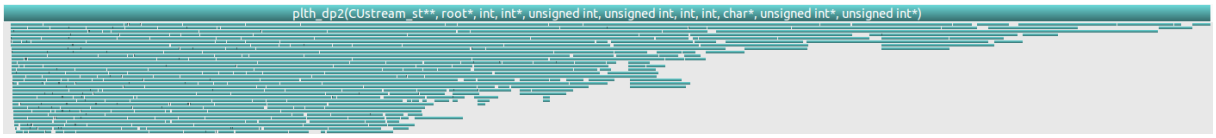
Figure 24 provides a visual profile of all non-CDP implementations: BP-DFS, REC-CDP, and REC-DP3. Figure 24a shows a visual profile of BP-DFS. It is straightforward to understand BP-DFS' execution: There is on the first line a $H2D$ memory transference,

[1] For a reason beyond our comprehension, differently from Figure 22d, the Intermediate GPU Search of Figure 22a is not represented by a full light blue line.

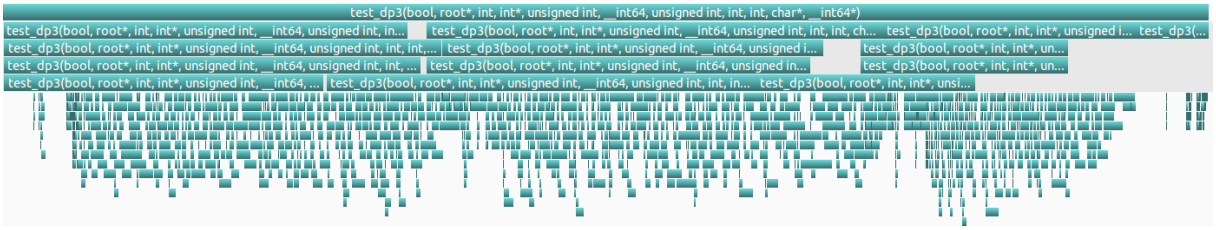
Figure 22 – Visual profile of CDP-BP, DP2, DP3, and CDP-DP3 solving ATSP instance *coin*14. Parameters are the ones of Table 14. Different colors mean different kernels for all sub-figures.



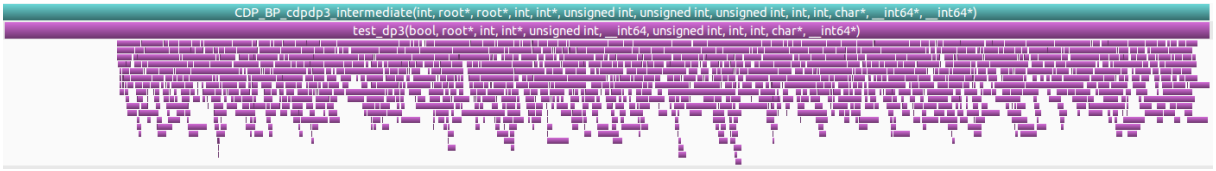
(a) CDP-BP: 134 launches of BP-DFS via CDP.



(b) DP2: 1714 launches of DP2 via CDP.



(c) DP3: 1721 launches of DP3 via CDP.



(d) CDP-DP3: 1709 launches of DP3 via CDP.

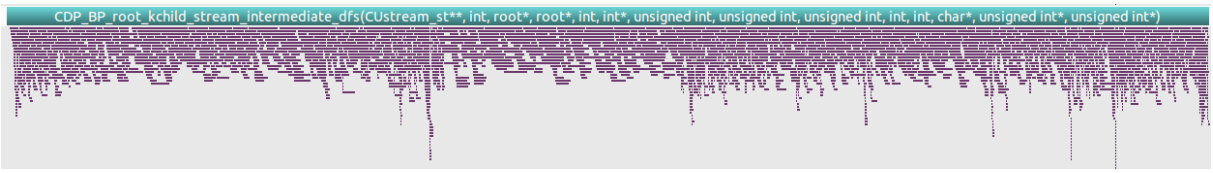
Source – The Author.

followed by a kernel execution. Finally, there is a $D2H$ transference, which is much smaller than the $H2D$ transference. Each one of these operations is on a different line to symbolize that they are performed by different channels.

Figure 24b shows a visual profile of REC-CDP using $d_{cpu} = 3$ and $d_{gpu} = 8$. By using this configuration, it is possible to see each step of REC-CDP's execution: $H2D$ copies, Intermediate GPU Search (purple), $D2H$ copies, $H2D$ copies, Final GPU search (light blue), and $D2H$ copies. Finally, Figure 24c shows a visual profile of REC-DP3. In this figure, it is difficult the see the first kernel generation because its execution time is much quicker than the execution time of the other kernels. On the other hand, the final search amounts for the great majority of the time.

Figure 23 – Visual profile of CDP-BP solving ATSP instance *coin*14 and using $d_{cpu} = 6$ and $d_{gpu} = 8$. Different colors mean different kernels for all sub-figures.

(a) CDP-BP: 1185 launches of BP-DFS via CDP.

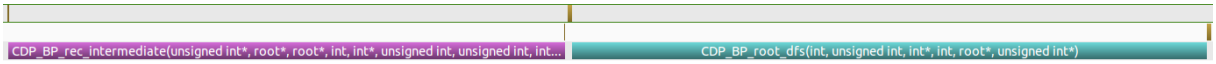


(b) CDP-BP launching 4 stream per block: 4740 launches of BP-DFS via CDP.

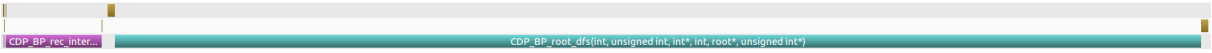
Source – The Author.

Figure 24 – Visual profile of BP-DFS, REC-CDP, and REC-DP3 solving ATSP instance *coin*14. Different colors mean different kernels for all sub-figures.Yellow operations are copies between the host and the device. All subfigures have three lines: the first one shows $H2D$ copies, the second one shows $D2H$ copies, and the third one shows kernel executions.

(a) BP-DFS using the parameters of Table 14.



(b) REC-CDP using $d_{cpu} = 3$ and $d_{gpu} = 8$.



(c) REC-DP3 using the parameters of Table 14.

Source – The Author.