Solving Hard Problems at Scale using Massively Parallel Manycore Processors

An Investigation of GPGPU Acceleration Techniques

Filippos Pantekis

Submitted to Swansea University in partial fulfilment
of the requirements for the Degree of Doctor of Philosophy

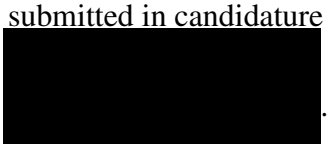
Prifysgol Abertawe
Swansea University

Faculty of Science and Engineering
Department of Computer Science
Swansea University

February 07, 2025

Declarations

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed .....  ..... (candidate)

Date .....February 07, 2025.....

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .....  ..... (candidate)

Date .....February 07, 2025.....

I hereby give my consent for my thesis, if accepted, to be available for electronic sharing.

Signed .....  ..... (candidate)

Date .....February 07, 2025.....

The University's ethical procedures have been followed and, where appropriate, that ethical approval has been granted.

Signed .....  ..... (candidate)

Date .....February 07, 2025.....

Abstract

Serial (i.e., non-parallel) algorithms have historically been superseded by parallel equivalents which keep up with the evolution of CPUs, specifically, from single-core to the modern-day multi-core. This is a non-trivial transition, typically involving complex analysis and adjustment of code to exploit the architecture. Furthermore, the need for such development has increased due to the emergence and widespread availability of massively parallel manycore (co)processors over recent years that have offered access to increased computational performance relative to conventional processors (CPUs). However, this has come with the cost of developing bespoke algorithms which exploit the specialities and requirements of such hardware.

Graphics Processing Units (GPUs) are a prime example of commercially available massively parallel co-processors that have been shown to offer significant performance gains when approached in an appropriate manner. At the same time, a plethora of 'hard' constraint satisfaction problems exist which, when approached carefully, benefit from the computational power of these devices, such as the Boolean Satisfiability (SAT) and the N-Queens problems.

In this work, we explore the applicability of current GPU technology to the SAT and N-Queens problems. We present our design of a hybrid solver for SAT, which utilises a fast implementation of a scalable, loosely coupled GPU-based checker. Furthermore, we present a fully GPU-based, and fully scalable N-Queens solver that is state-of-the-art, built around our `DoubleSweep-Light` algorithm, which surpasses results of other solvers. Beyond algorithms and approaches for these specific problems, our exploration yields lessons and identifies general optimisations that can be applied to a broad range of problems, both for memory- and compute-bound kernels.

In loving memory of my father, Yannis, who saw the start but not the end of this journey, and to my mother Elena whose love and care got me through.

Acknowledgements

I wish to deeply thank, my supervisors Liam O'Reilly, Phillip James, and Faron Moller for their much needed guidance and limitless support even when going got rough. Their efforts were great, extending far beyond the role of any supervisor and I'm very grateful for that. Through your guidance I discovered research which I enjoy thoroughly, as well as my love for teaching which you each facilitated in different ways. You really made these 5 years fun and full of knowledge. I hope you enjoyed them as much as I did!

Next, a massive thank you from the bottom of my heart goes to my family and family friend Mary who brought me up and got me here. Particularly, to my mother and biggest fan Elena, words can't capture how grateful I am for all you have done for me throughout my life and the love you so generously gift me!

During this PhD I had a number of accomplices, namely Afrodite Kypri, William Oldham, Chess Hutin, Olga Petrovska, Casey Hopkins, Lauren Powell and Maria Moller whose *infrequent* and *very gentle* nudges to 'write my thesis and quit procrastinating' *may* have been *a little* necessary, as I realise now from across the bridge... I can't help but admire your patience with me sometimes!

A huge thanks also goes to my friends Joseph Dawson, Michael Kenning, Sadeer Beden, Tyler Miller, Tereza Pashinska, Nikolina Antoniou, Michalis Mylonas, Cay Gaspar-Jones, Damian Glowala and the Technocamps family who each contributed in their own ways!

To my fellow PhD friends Ben Lloyd-Roberts and Connor Clarkson I enjoyed our paper writing sessions, occasional tear shedding episodes, rants about Python, late nights in the office and our civilised arguments over who is the biggest impostor (which is me, just to settle it). You're almost there, you've got this!

Going back a few years to early life, I also feel the need to thank and extend my love for my teachers Ms Katerina Mavroeidi, Ms Myrto Sifnaiou and Ms Jenny Vrotsou who's love for their respective subjects of Greek Literature, Mathematics and Computer Science intrigued me and sticks in my memory to this day. You each played a

big role in my learning and life growing up, and in ways, contributed to this piece of work coming to fruition.

Finally, I wish to thank the external examiners, Ciaran McCreesh and Andrew Ware for their efforts in checking this work and the intriguing conversations we had in the Viva. Furthermore, thanks to Swansea University and the Engineering and Physical Sciences Research Council for jointly funding this work, and Supercomputing Wales for facilitating access to Swansea University's AccelerateAI supercomputer and allowing me to get my (virtual) hands on this eye-wateringly expensive hardware!

Table of Contents

List of Figures

Acronyms

ALU Arithmetic Logic Unit. 61, 62, 67, 134

CC Compute Capability. 23, 25, 28–30, 42, 44, 47, 48, 66, 68, 70, 73, 104, 107, 116, 132, 138, 146, 147, 169

CDCL Conflict Driven Clause Learning. 2, 15, 18, 33, 36, 37, 39–41

CNF Conjunctive Normal Form. 12–14, 36, 72

CPU Central Processing Unit. v, xiv, 2–4, 17, 21, 25, 26, 31–34, 36, 38, 40, 41, 44, 45, 55, 57, 58, 65, 67, 76, 78, 82, 99, 113, 114, 128, 132, 136, 139, 145–147

CUDA Compute Unified Device Architecture. xiv, 21, 25, 32, 50, 71, 73, 100–103, 122, 124, 130, 146, 157

DPLL Davis Putnam Logemann Loveland. 3, 13–15, 18, 37, 65–67, 75

FLOPS Floating-point Operations Per Second. 73, 147

FMA Fused Multiply and Add. 61, 62

FPGA Field Programmable Gate Array. 17, 38, 45, 76, 113, 114, 143, 144, 148

GPGPU General Purpose Graphics Processing Unit. xi, xiii, 1–4, 6, 19, 21, 24, 31–33, 127, 128, 144–148

GPU Graphics Processing Unit. v, xi–xv, 1, 3–6, 9, 15, 20–24, 26–36, 38–68, 75, 76, 78–114, 118, 119, 121, 122, 126–128, 132, 133, 135–140, 142–151, 161–166, 170

HPC High Performance Computing. 62, 91, 140, 144

IOPS Integer Operations Per Second. 73, 147

ISA Instruction Set Architecture. 28, 29, 128, 155

LLM Large Language Model. 144

MMA Matrix Multiply and Accumulate. 1, 67, 73

MPI Message Passing Interface. 48

NUMA Non-Uniform Memory Access. 48

NVCC NVIDIA Cuda Compiler driver. 52, 101, 128, 129, 132

PTX Parallel Thread Execution. xiii, 28, 29, 51–54, 73, 101–107, 124, 129–135, 142, 150–156, 170–172

SAT Boolean Satisfiability. v, xi–xiv, 1–5, 11–13, 33–69, 72, 73, 75, 77, 80, 127, 136, 137, 143, 149, 157, 161, 162

SIMD Single Instruction Multiple Data. 21

SIMT Single Instruction Multiple Threads. 21, 22, 24, 39, 79

SM Streaming Multiprocessor. xiv, 3, 22–27, 29, 30, 39, 44, 46, 60–62, 65, 67, 70, 87, 95, 100, 103, 106–108, 116–118, 120, 126, 130–132, 138–141

VLSI Very Large Scale Integration. 17, 20

Chapter 1

Introduction

Contents

In early times, a Graphics Processing Unit (GPU) had a sole purpose: to perform graphics-related operations (e.g., shading, scaling, etc.) and perhaps display output onto a screen. A GPU is a special purpose, manycore processor, designed to excel in the logic and arithmetic operations often required by graphics workloads. More recently, the concept of a General Purpose Graphics Processing Unit (GPGPU) has emerged which in essence is a GPU with a programming interface that exposes its capabilities and can be used for non-graphics workloads as well. GPGPUs have seen an unprecedented evolution, broadening their potential areas of application by natively supporting more operations, some less useful to graphics workloads, under the overarching massively parallel model of computation they employ. GPGPUs have been adopted in areas benefiting from their capabilities such as cryptography [5], graph analytics [133], Boolean Satisfiability (SAT) [99] as well as machine learning and artificial intelligence [113, 130, 50]. Adoption of GPGPUs in machine learning applications in particular appears to have shaped the evolution of NVIDIA GPGPUs, which introduced Tensor Cores [95] to efficiently perform Matrix Multiply and Accumulate (MMA) operations over fixed matrix types, to better suit these applications that heavily depend on them.

Problems such as Boolean Satisfiability are deemed 'hard' as they require (near) full exploration of the search space in the worse case. The algorithms for some hard problems are relatively 'light' in that they are not complex to reason about or implement.

Typically algorithms exploring the majority or the full search space in these problems are backtracking-based [140, 32, 101, 21, 61, 4]. Such algorithms are typically better suited or perhaps 'moulded around' the model of computation employed by CPUs. For instance, the CDCL procedure for SAT is expressed in a serial manner that does not easily fall in line with parallel (or worse, massively parallel) computation. Adapting algorithms for parallel/concurrent implementations is actively explored [17, 104] with a range of techniques being identified.

In this work, we hypothesise that appropriating such 'conventional' algorithms, however light, for implementation on the GPGPU requires a great deal of adaptation on the algorithmic level, beyond just parallelising them efficiently. For instance, one must consider which memory type is being used for what data and for how long, what access patterns are being performed, to what degree can each thread be agnostic of others, and how is work balanced between the thousands of concurrently executing threads, to name a few. Despite its difficulty we believe this is a worth-while effort considering the run-time performance that can be gained by using special-purpose hardware in an appropriate manner [99, 50, 111, 102, 100]. This is a primary motivation that is multidimensional; the definition of 'performance' here may take different forms, but ultimately any savings in time, energy, computational cost, and so forth are welcome.

This work uses the N-Queens and Boolean Satisfiability problems, both computationally 'hard', as proxies to explore different approaches in algorithmic appropriation for GPGPUs. These approaches are used against the problems in question to evaluate them and gain an understanding of their effectiveness under different workloads. After our first-hand exploration of those, we present our observations and suggestions for others aiming to port 'hard' workloads on the GPGPU, and especially those solved through backtracking search.

Although GPGPUs are special-purpose manycore processors, certain abstractions in the programming model incur costs known as "Turing Tax"*. We hypothesise that implementations can reduce this Turing Tax [67] significantly if optimised iteratively, feeding back from the impact of previous optimisations and the hardware utilisation statistics. Our aim is to develop techniques and isolated examples of implementation-level optimisations applicable to a range of backtracking-based applications or beyond. Premature optimisation however is a hindrance; a methodical approach is taken to evaluate the applicability of each optimisation on the implementation level and the situations under which it would be better suited.

*To describe this phenomenon we borrow the term Turing Tax from Prof. Paul Kelly [67]

1.1 Main Contributions

The main contributions of the work presented in this thesis are outlined below.

***Exploration of the applicability of GPGPUs leveraging the current state-of-the-art for solving SAT*:** In Chapter 3, we present a hybrid CPU-GPGPU SAT solver design with two alternative designs for the GPGPU-side component. For one design, we draw inspiration from an approach documented in literature, and explore the limitations surrounding this memory-bound approach, which are largely the same for other memory bound kernels with small compute components. The other design presented is a novel SAT checking approach (published in [100]), designed with minimal memory requirements and emphasis on kernel latency reduction. Results obtained from the latter approach are presented which show the potential of this approach, and more broadly, the use of GPGPU assistance in SAT solving as well as the open scalability of the approach across multiple devices. Techniques presented as part of the aforementioned chapter are primarily relevant to memory bound kernels, as well as those with high dependence on specific SM pipes.

***Investigation of the conversion of conventional algorithms for hard combinatorial problems for the GPGPU*:** Both problems investigated as part of this work are instances of hard combinatorial problems and act as mediums for exploration of techniques to convert what we refer to as "conventional" algorithms (i.e., those designed with CPU implementation in mind), to highly parallel ones. Importantly, we demonstrate instances where algorithmic design choices, normally deemed poor, are necessary to meet the requirements of GPGPU computing and produce an efficient implementation. To aid future algorithmic development and conversion we present a set of considerations in the form of an alternative model for algorithmic design, for targeting massively parallel platforms, in particular GPGPUs.

***Development of a state-of-the-art N-Queens solver*:** In Chapter 4, we present our state-of-the-art, fully on-GPGPU N-Queens solver based around the novel `DoubleSweep-Light` procedure (published in [101] and [102]). The procedure itself is centred around the Unit Clause Propagation component of the DPLL procedure for SAT. We explore means of adapting a CPU-oriented algorithm into a massively parallel variant that overcomes the barriers of GPGPU parallelisation. We explore two different implementations for `DoubleSweep-Light` on the GPGPU, each targeting devices with different capabilities. We then explore the effects of a number of surface-level and lower-level optimisations to our implementation alongside results showing their effectiveness. Most such optimisations are general and can be applied to other kernels (and particularly compute-bound ones) as summarised in Chapter 5. The overall performance of our N-Queens solver is competitive, surpassing the performance of previous GPU-based N-Queens solvers in literature.

***Identification of general optimisation techniques targeting combinatorial problems on the GPGPU*:** Efficient use of GPGPU resources and the efforts associated with achieving it form a running theme through this work. In our pursuit of performant solutions to the problems considered, we encounter the need for hand-tuning optimisation, beyond what current compiler tooling offers. The two problems we examine, namely *N*-Queens and SAT allow us to explore memory-bound (Chapter 3) and compute-bound (Chapter 4) kernels and develop optimisation strategies for each, that are applicable to other backtracking-based approaches for combinatorial problems. For both types of kernels, we consider multi-kernel decomposition and pipelining based on our experience and that of others. We contrast this approach to the respective full on-GPGPU approaches and summarise hand-tuning optimisations involved alongside our recommendations stemming from them, in Chapter 5.

1.2 Structure of Thesis

In Chapter 2, we lay out the fundamentals necessary for the reader's understanding of the topics addressed in this work. We begin with an exploration of 'hard' problems and the main problem types considered in this work which include decision and counting problems, and offer examples of each, zoning in on the Boolean Satisfiability and *N*-Queens problems respectively. We explore the literature for each of these areas, which broadly converges on the use of flavours of backtracking search. We continue by laying out some graphics processor (GPU) preliminaries that are crucial for our work, particularly visiting the model of computation employed by General Purpose Graphics Processing Unit (GPGPU) devices, as well as the mapping of computation to hardware resources and the intricacies of those. Our work (and consequently, exploration) is based on a range of NVIDIA hardware architectures and implementations thereof, specific details of which are presented and discussed in the context of potential optimisations. Lastly, we present barriers to GPU optimisation and issues with reproducibility of work in the field arising from the intricacies of GPUs. We factor these difficulties in our presentation of techniques to appropriate algorithms for the GPU environment.

Following on from the preliminaries, Chapter 3 presents our work on 3-SAT acceleration using GPGPUs. We introduce GPU-specific work on the field of SAT and the two parallelisation strategies employed widely, namely portfolio and search-space splitting solvers. We subsequently explore two designs for SAT acceleration on the GPU, one where all solving takes place on-device using a backtracking algorithm and one where GPUs act as search assistants on a CPU-coordinated search brute-forcing the last few unassigned variables in each branch of the CPU search, based on the observation that those are the variables solvers spend the most time deciding [138]. The latter approach presents an openly scalable SAT checker producing good results both on commercial-

and data-centre- oriented devices, which is published in [100]. We explore kernel optimisations and techniques effective for memory-bound kernels such as this as well as offering insight on elastic scaling on any number of devices. We conclude this chapter with our ongoing and future work and with several opportunities for optimisation, which we explore in detail.

We continue with Chapter 4 where we detail our work to produce a loosely coupled N-Queens counting solver which enumerates (and can in principle also output) all solutions on the device. We once again explore existing GPU-related work and identify the shift in focus in this field, first to distributed systems and then to massively parallel special-purpose hardware (i.e., manycore processors) being used in the search for solutions. This appears to have historically been the trajectory leading up to the largest known solution to date, $N = 27$. Our exploration here is two-pronged; we draw inspiration from multi-kernel pipelines [83] for the parallelisation of other such irregular problems and delve in the design of a device-side multi-kernel N-Queens solver coordinated by the host system. Preliminary analysis highlights the drawbacks of such approach, which in turn leads us to a full on-device approach using the novel `DoubleSweep-Light` algorithm along a range of implementation optimisations, to achieve a performant N-Queens solver. The algorithm employs a procedure like Unit Clause Propagation in SAT as previously discussed. This work was initially published in [101] and following promising results, invited for an extended journal publication [102]. Finally, we conclude the chapter by outlining ongoing work and other work in the pipeline to be done in due course.

The two hard problems explored in this work nicely encapsulate the two types of performance bottlenecks that bound GPU kernels; memory (SAT) and compute (N-Queens). The implementations shown have been optimised extensively to make as much use of available resources as possible, without compromising on the scalability of the approach. Techniques employed and developed are not coupled to our specific applications and can thus be employed in other kernels. In Chapter 5 we review and reflect on the main learning points arising from this work. We envisage that our journey through the appropriation of algorithms for hard problems and their respective implementations will prove of value for others on a similar path. In this chapter we curate optimisation and algorithmic techniques as well as our recommendations on the use of GPUs in such applications.

Finally, we conclude this work with Chapter 6 with a reflection on the work that has been carried out, and our aims.

1.3 Publications and Talks

Parts of the work presented in this thesis have been published and/or presented in academic conferences. These and additional contributions forming part of the fundamentals for this work are presented in this section.

1.3.1 Publications Arising from This Work

Optimised Massively Parallel Solving of N-Queens on GPGPUs [102] (by F. Pantekis, P. James, O. Kullmann, and L. O'Reilly) This work is the continuation of our earlier work [101] following an invitation to the Concurrency and Computation: Practice and Experience (CCPE) journal. This publication is an extended version of our earlier work presenting our approach in more depth, and placing the spotlight on implementation details that result in performance gains. We introduce a number of new hand-tuning optimisations as well as a register-based kernel implementation, which eliminates the need for shared memory almost entirely, and results in higher performance on more recent hardware. We explain our rationale behind each algorithmic and implementation change and we provide experimental data showcasing the impact these changes had to the overall runtime and the main lessons learned from this exploration. The contents of this contribution are discussed in more detail throughout Chapter 4.

Scalable N-Queens Solving on GPGPUs via Interwarp Collaborations [101] (by F. Pantekis, P. James, and O. Kullmann) This paper presents a novel algorithm for the counting variant of the N-Queens problem called `DoubleSweep`, as well as the adaptations made to better-suit the algorithm for implementation on the GPU which result in `DoubleSweep-Light`. The paper details the reasoning for morphing the algorithm around the characteristics of hardware used to run its implementation, and provides implementation details along with an overview of specific optimisations applied to the implementation in places where compiler tooling produced a sub-optimal output. Product of combining `DoubleSweep-Light` with a performant, tuned implementation is a state-of-the-art solver for counting N-Queens solutions, for which the paper presents initial results that place it above other GPU-based solvers in literature. The contents of this paper form the fundamentals discussed in Chapter 4.

Towards Massively Parallel GPU Assisted SAT [100] (by F. Pantekis and P. James) This paper presents a GPU-based assistant for SAT solving intended to be used in a hybrid solver to aid the host-side with solving the target expression. The basis of this work is a simple brute force algorithm with an optimised thread-isolated implementation, used to explore techniques for memory requirement reduction and efficient use. The paper identifies some potential applications for such hybrid approaches, particularly in variants of SAT such as ALL-SAT, but highlights the challenges presented in

implementing 'conventional' state-of-the-art algorithms onto the constrained environment of GPUs. This paper covers work presented in Chapter 3.

1.3.2 Conference and Workshop Presentations

A number of talks concerning mostly work detailed in this thesis have been given, as summarised below.

- Presentation of [103] in the *Seventh International Workshop on Formal Techniques for Safety-Critical Systems* (FTSCS) in Shenzhen, China (2019).
- Talk presenting the initial algorithmic design for the work detailed in Chapter 3, in the *Doctoral Program* for the *Twenty-seventh International Conference on Principles and Practice of Constraint Programming* (CP-DP) delivered remotely (2021).
- Talk presenting preliminary results for the work detailed in Chapter 3 and co-organisation of the *Thirty-eighth British Colloquium for Theoretical Computer Science* (BCTCS) in Swansea (2021).
- Presentation of the work in [103] in the *Seventh Model-Driven Engineering Network Research Demonstration Workshop* (MDEnet) delivered remotely (2022).
- Presentation of [100] in the *Seventh International Workshop on GPU Computing and AI* (GCA'22) in Himeji, Japan (2022).
- Talk presenting preliminary results for the work detailed in Chapter 4 in the *Thirty-ninth British Colloquium for Theoretical Computer Science* (BCTCS) in Glasgow (2022).
- Presentation of [101] in the *Tenth International Symposium on Computing and Networking* (CANDARD'22) in Himeji, Japan (2022).

1.3.3 Contributions Beyond the Scope of This Thesis

Visualising Railway Safety Verification [103] (by F. Pantekis and P. James) This paper presents a method to lay out railway track plans on the OnTrack railway verification toolset [60] in-line with the expectations of railway engineers, and visualise counter-example traces onto them. The layout of track plans is achieved using a simulated annealing algorithm to achieve results that require few or no changes to conform with layout rules. The work presented in this publication is not strictly a part of this thesis, but contributes to the fundamentals for the load distribution approach mentioned in Section 4.5.1.

OnTrack: Reflecting on Domain Specific Formal Methods for Railway Designs [59] (by P. James, F. Moller and F. Pantekis) In this paper, the development of the OnTrack railway verification toolset [60] is presented as a case study for the use of model-driven engineering frameworks. The applicability of the model-driven paradigm to the various components of the toolset is examined and the challenges faced in the process are discussed. Challenges include that of scheme plan importation and the subsequent visualisation in an appropriate layout which were explored in our earlier work [103].

Chapter 2

Background

Contents

In this chapter we lay the fundamentals upon which our work is built, which are crucial for the readers understanding. We define what a 'hard' computational problem is, with specific focus to the problems we explore later on and others akin to them. We continue with an overview of work in parallelisation of backtracking algorithms, which are at the core of our work, followed by an overview of how computation on NVIDIA GPUs is performed. This overview covers the model of computation, hardware implementation, as well as specific capabilities of and changes between GPU versions that have an impact on our work. Lastly, we discuss the steps involved in adapting an algorithm for implementation on the GPU, as well as the barriers and other issues often faced in the process.

This chapter covers sufficient information for the reader to proceed to subsequent chapters, including some related work. Subsequent chapters provide further information and related work, specific to each of them.

2.1 Hard Problems

Several types of computational problems exist in the field of Computer Science. A computational problem can be viewed as a function $p : I \mapsto \mathscr{P}(S)$ which maps a problem instance from the set I to a set of solutions from the set S. More specifically, an

instance of a problem $i \in I$ is a list of input parameters to the function p which will be mapped to a set of solutions $s \in \mathscr{P}(S)$, which may be empty if no solutions exist for the given instance.

Computational problems come in different types. These types include Decision, Counting, and Optimisation problems amongst others.

Decision problems have exactly one solution which is either true or false. For example, deciding whether an integer x is a multiple of some integer y: $multiple\_of([x,y]) = \{(x \bmod y) = 0\}$.

Counting problems involve identifying the number of solutions for a given problem instance. For example, counting the number of prime divisors of some $y \in \mathbb{N}$: $count\_prime\_divs([y]) = \{|\{x|x \in \mathbb{N} \wedge prime(x) \wedge y \bmod x = 0\}|\}$ where the function $prime : \mathbb{N} \mapsto \{\top, \bot\}$ is used to decide if a natural number x is prime.

Optimisation problems aim to find the single 'best' solution for a given problem instance, where 'best' is a problem-specific metric, typically by minimising or maximising an objective function. For example, finding the shortest path in a directed graph [42].

A defining characteristic of the aforementioned problem types, is that they all yield up to one solution. Problems, irrespective of their type and solution set cardinally, may be more difficult (harder) than others in terms of practical difficulty in finding their solution set. The difficulty of solving a problem is typically measured in terms of algorithmic complexity [55] for each known algorithm capable of solving the given problem, or in other words, an algebraic function relating the size of input to the amount of work needed to reach a solution set. Problems are categorised into sets along with other problems with similar solving time or memory requirements known as complexity classes, including P, NP and $\#P$.

The complexity classes P and NP ($P \subseteq NP$) contain decision problems.

In P are problems that are solvable in polynomial time meaning that the time complexity of the algorithm is polynomial with respect to the size of the input. In NP, are problems for which a proposed solution can be verified in polynomial time. Unlike P, NP includes problems that may lack efficient solutions (i.e., those computable in polynomial time), yet if a potential solution is provided, its correctness can be verified efficiently. The $\#P$ class consists of counting problems which have an associated decision problem in NP. By solving a counting problem, one is implicitly solving its associated decision problem as well.

For a complexity class ε from the aforementioned, a corresponding $\varepsilon - Complete$ class exists. This class is such that for a problem p in $\varepsilon - Complete$, a polynomial-time reduction is possible to p from all other problems in ε. Accurately determining the

complexity of an algorithm and modelling its behaviour under the best, worst and average cases as well as proving its existence in a complexity class can be challenging and forms the basis of the field of Computational Complexity.

Slight variations in the formulation of a problem can have a profound effect in algorithmic complexity. Confusingly, it is often the case in literature that slightly different definitions of a problem are placed under the same umbrella name. A notable example is that of the "N-Queens" problem as explained in Section 2.1.2 where "the N-Queens problem" may refer to at least 3 different variants of problem with different complexity.

The N-Queens problem as presented in Section 2.1.2, is a typical example of a hard counting problem for which, to the best of our knowledge, no definitive complexity proof exists. We do conjecture however that the problem will be at least as hard as $\#P$ problems. This is since the completion counting variant has been shown [43] to belong in this class and the N-Queens problem can be seen as an instance of counting completions for an empty board. Further discussion on this point is made under Section 2.1.2. Recent work [117], has provided insight into the super-exponential growth of solutions to the N-Queens problem, which highlights the difficulty of solving it using a deterministic search algorithm (i.e., the only known way for exact enumeration [43]).

Another prominent example of a hard problem is the Boolean Satisfiability (SAT) problem presented in Section 2.1.1. SAT is a decision problem, where the goal is to determine whether a given Boolean formula is satisfiable (i.e., there exists an assignment of truth values to variables that makes the formula true). The SAT problem was famously proven [28] by Stephen Cook and Leonid Levin as the first to belong in the $NP-Complete$ class. The theoretical fundamentals of the problem are extensive, but stretch beyond the scope of this work, however, for further reading we refer the reader to [16]. SAT is studied widely and from a plethora of angles due to its large range of applications [80]. As such, a number of variants of the decision problem have been formulated, which include 2-SAT and 3-SAT. These are sub-variants of the k-SAT [31] variant where formulae are a conjunction of clauses comprised of the disjunction of two and three literals respectively. The distinction is made however, as 2-SAT is solvable in polynomial time [11] as opposed to the family of k-SAT variants for $k \geq 3$, which belong in the $NP-Complete$ class [31, 65].

Variants of the SAT problem have been formulated in different ways for applications that demand it. For instance, MAX-SAT [76] is an optimisation problem, variant of SAT, where the objective is to maximise the satisfied clauses (i.e., find the assignment satisfying the most clauses of a formula). #SAT [48] is a counting problem variant, with the objective of identifying the number of assignments that satisfy a given formula, which is not too dissimilar to ALL-SAT where the objective is to enumerate (and output) all satisfying assignments. Empirically, and as discussed in Section 2.1.2,

problems requiring the enumeration of all solutions, such as ALL-SAT and N-Queens, can perhaps be harder to solve in terms of practical difficulty compared to the respective problem of counting solutions. This is because heuristics capable of calculating solutions on a sub-problem level without explicitly exploring them may exist for the latter which are of no use to the former.

From a theoretical perspective, both enumeration and counting problems are viewed as similarly complex under traditional computational complexity frameworks. Efforts to differentiate and better express the hardness of enumeration problems are documented in literature [29]. Complexity classes, specific to enumeration problems, do exist, such as *DelayP*. In this instance, the *DelayP* class comprises enumeration problems for which a polynomial delay algorithm exists. These are algorithms under which the time to find the first solution, a successive solution, and to verify the absence of further solutions is bounded polynomially.

For hard problems such as the aforementioned, a well-informed algorithmic choice is important to solve them effectively and in sensible time frames in the general case, even if some hard problem instances persist. Such algorithms must however be met with fast and efficient implementations exploiting all potential of hardware to have a tangible effect. The focus of this work is not on the theoretical bounds of these algorithms but an investigation on the optimisation of practical implementations of those particularly for massively parallel environments.

2.1.1 The Boolean Satisfiability Problem

The Boolean Satisfiability problem (SAT) [16] is to decide if, for a given propositional formula ϕ, there exists an assignment μ of truth values such that ϕ is "satisfied", i.e., ϕ evaluates to true. A formula ϕ is defined as a set of literals connected using propositional connectives (i.e., $\wedge, \vee, \implies, \iff$). In this context, a literal is either a variable v or its negation $\neg v$.

Algorithmic design is complicated by having to account for all aforementioned propositional connectives and their properties, that may appear in a formula. To simplify this task, it is common to require formulae to conform to some encoding with fewer connectives, such as the Conjunctive Normal Form (CNF). Formulae in CNF are comprised by a conjunction of clauses, which in turn are comprised of a disjunction of literals. For instance, the formula $\phi_{cnf} = (l_1 \vee l_2 \vee l_4) \wedge (l_2 \vee l_3)$ is in CNF. Arbitrary formulae can be converted to equisatisfiable CNF formulae in linear time using the Tseytin transformation [127]. Such transformation is not required for problems with reductions to SAT which may naturally be expressed in CNF. This encoding is widely used in the SAT community with some of the most prominent solving algorithms being built around it [32, 140]. The representation of CNF formulae in textual form is

straight forward and is known as the DIMACS format. DIMACS encodes each variable as a signed integer which is negative to indicate a negated variable. DIMACS files are plain text files that consist of a header line followed by lines that each represent a clause. Each line lists one or more literals separated by spaces, which are part of that clause. For the sake of simplicity we will use the format seen in Example 1 to represent CNF formulae in this work.

The k-SAT variant [110] of the SAT problem restricts formulae to k many literals per clause, encoded in CNF*. Here, the value of $k \in (\mathbb{N} \setminus \{0\})$ can have a profound impact on the difficulty in solving the problem as for values of $k \leq 2$, the problem is trivially solvable [11] (i.e., in polynomial time) by treating the clauses as implications between literals which in turn represent edges between literals (nodes) in a graph. Upon such a graph, a strongly connected component finding algorithm can be applied to identify strongly connected literals in linear time, and checking for the absence of contradictions in these sub-graphs. This however, is not the case for instances of k-SAT where $k \geq 3$ as no polynomial time algorithm is known [31] and a polynomial time reduction exists from SAT to 3-SAT [65] which itself is the first problem in the $NP-Complete$ class [28]. These 'hard' instances are typically tackled (deterministically) using backtracking search algorithms.

Example 1 The CNF formula:

$$(\neg A \vee \neg B \vee I) \wedge (AI \vee \neg I \vee AR) \wedge \cdots \wedge (\neg I \vee AK \vee P)$$

can be represented in DIMACS format as:

$$\begin{array}{c} (-1,-2,9) \\ (32,-9,44) \\ \ldots \\ (-9,34,16) \end{array}$$

One of the first deterministic algorithms for solving SAT instances was the Davis Putnam Logemann Loveland (DPLL) procedure [32] which was named after its authors. This procedure is product of an exhaustive backtracking search over the possible truth assignments of each variable in the formula and effective heuristics to evade fruitless paths. The procedure can be summarised as the high level algorithm shown in Listing 2.1. The function `dpll_sat` that can be applied over a formula ϕ and a set of tuples $\mu = \{(v_1, \top), (v_2, \bot), ...\}$ of truth assignments to corresponding variables. At its core, the algorithm is a backtracking search enumerating assignments, yet the strength of this procedure stems from two subroutines: Unit Clause Propagation and Pure Literal Elimination (shown in Listing 2.1 as the high-level functions `find_unit` and `find_pure` respectively).

*Such formulae may be referred to as formulae in k-CNF.

Unit clause propagation is a form of resolution, inferring the truth assignments of variables in clauses with exactly one unbounded literal. Assuming the input formula is in CNF, at least one literal in every clause must somehow be satisfiable for the formula to stand a chance of being satisfiable. Pure literal elimination examines the literals of clauses remaining unsatisfied in the formula, and derives a truth value for literals which are pure between them. A literal is pure if it appears only negated or only un-negated, meaning the value of its corresponding variable can be inferred. Both these procedures are repeated until they yield no result (i.e., they make no new assignments).

Following the application of unit clause propagation and pure literal elimination, the formula may have been proven satisfiable, or a contradiction may have been identified, terminating the search in either case. If a proof hasn't been completed yet, the algorithm selects one of the remaining unbounded variables, and continues exploring the two new search paths (i.e., those with the selected variable assigned to either truth value). The choice of a variable shown as the function `assign_variable` in Listing 2.1, can be made arbitrarily, but a 'good' choice may lead to a conclusion sooner, therefore a number of heuristic approaches have been developed [52, 84].

```
1  fn dpll_sat(φ, μ):
2      let μ′ ← μ
3      let uc ← nil
4      while (uc ← find_unit(φ,μ′)) ≠ nil do:
5          μ′ ← propagate(φ,μ′,uc)
6      let pl ← nil
7      while (pl ← find_pure(φ,μ′)) ≠ nil do:
8          μ′ ← assign_literal(φ,μ′,pl)
9      if sat(φ, μ′) then:
10         return ⊤
11     if unsat(φ, μ′) then:
12         return ⊥
13     let v ← unassigned_variable(μ′)
14     μ′ ← assign_variable(φ,μ′,v);
15     if dpll_sat(φ, μ′) = ⊤ then:
16         return ⊤
17     μ′ ← assign_variable(φ,μ′,¬v);
18     return dpll_sat(φ, μ′)
```

Listing 2.1: Overview of the DPLL [32] procedure.

The DPLL procedure is effective, but can be improved upon. Following the exploration of an unsuccessful path, the search backtracks and follows a different search path which may ultimately be unsuccessful for the same reasons as its predecessors.

To counteract this, the Conflict Driven Clause Learning (CDCL) procedure [140], an extension of DPLL, was developed. The two main improvements CDCL introduces are the ability to backtrack non-chronologically (i.e., backtracking more than one assignment back), and conflict memorisation (i.e., adding new clauses to the formula upon reaching a conflict state to avoid reaching it again). Here, a conflict state is one where a variable must be assigned to both truth values simultaneously for the formula to potentially be satisfiable. These improvements to the original algorithm guide the search for a satisfying assignment more precisely and avoid the repeated exploration of search paths that proved fruitless.

Modern SAT solvers can solve complicated formulae quickly, as a result of their algorithmic capabilities paired with efficient implementations. Parallelisation of SAT algorithms is also explored [82] with two main approaches, namely search-space splitting solvers which divide the search space into sub-spaces [49] to be explored in parallel by the same search algorithm and portfolio solvers [41] which perform a number of independent searches over the same search space using different search algorithms in parallel. Attempts to utilise GPUs in parallelisation efforts exist [30, 100, 99, 83] and are discussed further in Section 3.1.

2.1.2 The N-Queens Problem

The N-Queens problem asks how many non-attacking configurations exist when placing N queens on an $N \times N$ chessboard. A non-attacking configuration is one in which no queen can attack any other queen on the chessboard. Two queens can attack one another if they are both occupying the same row, column or diagonal. The problem owes its roots to Max Bezzel who in 1848 asked how many possible placements of eight queens exist on a conventional (8×8) chessboard [20]. Figure 2.1 illustrates an example of a non-attacking configuration, which is one of the 92 non-attacking configurations for $N = 8$. This problem was later generalised [20] to the N-Queens problem as known today.

Like the Boolean Satisfiability problem discussed in Section 2.1.1, the N-Queens problem comes in a variety of 'flavours' and a distinction should be made on the alternative formulations that are sometimes used. Many pieces of work in literature exist [136, 64, 120, 128] which quote the variant of discovering a *single* non-attacking configuration of N queens for an $N \times N$ board, with further literature [46, 43] available, on the sub-variant of completing a partial N-Queens board. These alternative formulations of the problem vary significantly in solving difficulty. Namely, finding a single solution can be achieved in a number of ways including a reduction to an instance of SAT [18], simple search [120], randomised search through simulated annealing [116], or as an elementary solution from emerging patterns [136], whereas completing a partial N-

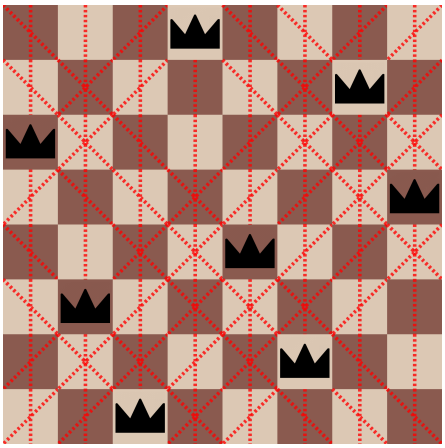


Figure 2.1: A non-attacking configuration of 8 queens on an 8×8 board shown with the per-queen attack indicators.

Queens board requires complete exploration [43] and is thus more computationally demanding.

For the rest of this work, we refer to the original variant of the generalised N-Queens problem, namely that of enumerating all solutions for a given value of N (and in principle even outputting each non-attacking configuration). The problem is computationally difficult to solve as the instance solutions known to date (OEIS [118] sequence `A000170`) exhibit super-exponential growth w.r.t. N (in line with the work of [117]). Whilst, to the best of our knowledge, there is no definitive work on the complexity of the N-Queens enumeration problem to date*, the earlier discussed N-Queens completion variant has been proven [43] to belong in the $NP-Complete$ complexity class and its corresponding counting problem (i.e., finding in how many ways a partial board can be completed in) in the $\#P-Complete$ class. This, paired with the fact the N-Queens problem can be seen as an instance of the N-Queens completion counting problem (i.e., completion of a board with no queens placed on it) leads to the conclusion that the N-Queens enumeration problem will be at least as difficult in terms of solving difficulty.

The term 'solution' must be disambiguated for the N-Queens problem, as it may refer to either the total number of non-attacking configurations possible for some value of N (i.e., solution to the N-Queens problem for some input of N), or an individual non-attacking configuration (i.e., a board of size $N \times N$ with N queens placed in a non-attacking configuration). For this work, we will use the latter definition of a solution

*Some literature [108] claims the decision problem belongs in the $NP-Hard$ class, however we have been unable to verify this claim.

and will refer to the total number of solutions for an instance of the problem (i.e., some input value N) as an 'instance solution'.

In practical terms, the N-Queens problem has long served as a challenge for mathematicians, programmers and machine learning models alike. Discovering solutions for smaller values of N, even up to 22, is relatively 'cheap' computationally with modern hardware, even using naïve solving approaches, due to the relatively small search space. For larger values of N however, the number of possible solutions to be enumerated is vast and requires a combination of 'smart' search algorithms and their efficient implementations. To the best of our knowledge, exhaustive search is the only known way [43] to obtain instance solutions for the N-Queens problem to date and solving efforts have stretched as far as collaborative crowd-backed computer grids [38]. Besides being used as a testbed, there are real-world applications for the N-Queens problem documented in literature [15], such as Very Large Scale Integration (VLSI) testing, deadlock prevention, conflict free parallel memory access schemes [36], and more. It must also be noted that algorithms for constraint satisfaction problems such as this and their respective implementations, have a lot of lessons to teach and can be adapted and applied to problems similar in nature.

Work by Jeff Somers [119] led to the computation of instance solutions for instances up to and including $N = 21$ in the early 2000's. Somers used an optimised backtracking search algorithm implemented in C and run serially on the CPU, which forms the basis of more recent examples of algorithms [37, 142, 108], including providing inspiration to our work presented in Chapter 4. As of yet, instance solutions are known for all $N \in [1,27]$, with the latest addition being that of $N = 27$ [111], discovered in 2016 following a year-long parallel computation effort involving several Field Programmable Gate Arrays (FPGAs). The authors of [111] were also the first to identify the solution to $N = 26$ [112] using a similar setup of parallel FPGAs. Beginning from these two examples, it is noteworthy that for all values of $N \geq 24$, solutions were first discovered using parallel algorithms on distributed systems. More specifically, the instance $N = 24$ was solved using a 34-node cluster of CPUs [69], and later $N = 25$ was solved on a grid of 260 machines [106]. Whilst instance solutions for $N \leq 26$ have been verified through an independent repetition of the computation, to the best of our knowledge, the result for $N = 27$ remains unverified.

Estimating the number of instance solutions for any N was recently proven possible [117] which also gives insight to the magnitude of search space in need of enumeration and its super-exponential nature, namely through the limit

$$\lim_{N\to\infty} \frac{Q(N)^{1/N}}{N} \approx 0.143$$

where $Q(N)$ is the number of non-attacking configurations for some N. Note that this is the minimum complexity and heuristics are needed to eliminate fruitless paths dur-

ing an enumeration attempt. The performance of implementations for such algorithms and respective heuristics, is of paramount importance which calls for optimisation and parallelisation. Parallelisation of backtracking algorithms is further discussed in Section 2.2.

2.2 Parallelisation of Backtracking Algorithms

An algorithm is said to be 'backtracking' if it can retrace its steps and follow a different search path upon encountering a dead-end during it's exploration of the search space. The backtracking paradigm is well established in a plethora of domains and applications, often combinatorial in nature [66], which include graph traversal [4], SAT solving [32, 56], N-Queens solving [119, 15, 3], AI [73] and many more. In such applications, the search space is often too large to feasibly explore fully even for smaller instances of the problem, and heuristic approaches have to be taken to identify and eliminate fruitless paths in the search before they are explored. In the domain of SAT for instance, the Conflict Driven Clause Learning (CDCL) [140] algorithm is an evolution of the Davis Putnam Logemann Loveland (DPLL) [32] algorithm with the addition of heuristics to 'learn' what caused a dead-end state to be reached so that the same dead-end can't be reached from a different search path. These algorithms are described in detail in Section 2.1.1.

Backtracking algorithms can generally be viewed as a depth-first traversal of a tree of search states, where at each state of the search, a set of branching paths emerges onto different search states. This tree analogy is somewhat simplistic as it does not account for backtracking searches which converge, yet it is used here as a proxy to explain why parallelisation of such non-converging algorithms is trivially possible. Starting from an initial state (the root of the search tree) and advancing the search onto neighbouring states, it is possible to create some number of new (distinct) starting points for parallel searches to begin branching from. Given that the search paths are not converging from that point onwards, result integrity is maintained as long as no parallel search backtracks beyond its starting point in the tree. This idea is not new and has been examined in depth in [66]. This approach has several benefits including the trivial nature of generating the initial ('frontier') search states for the parallel searches to branch off of.

Backtracking algorithms are easily implementable recursively, using the call stack to store search states and the recursive mechanism to return to them in turn. On one hand, these implementations are elegant and much less error prone when implemented in lower-level languages such as C as recursion requires no explicit memory allocation, 'housekeeping', nor release, abstracting (to a large degree) the code movement logic (jumps, etc.) and enabling compilers to perform optimisations [124]. On the

other hand, the abstracted implementation factors may come at a cost in such implementations, which can lead to inefficient resource utilisation. Side-effects stem from the absence of control of data that is preserved in stack frames which can result in unnecessary or unchanging data 'clogging' the call stack, the size of which in turn may be difficult to accurately control. These side-effects only magnify when parallel independent searches are performed.

Iterative implementations of backtracking algorithms in essence pass all responsibility for memory management and logic to the programmer leaving more room for error, but giving finer control over these parameters. Generally, a stack-like data structure can be used to store pieces of required data for each search state, and a loop to pop/push this data as required to simulate backtracking. In special processing environments such as that offered by GPGPUs, recursive implementations are (as of yet) discouraged or even impossible as a result of the severely limited default stack space available to each thread [89] which resides in off-chip memory, and the effects of raising this limit to memory utilisation and performance. Iterative implementations can be designed to use the minimum amount of memory necessary and make use of on-chip alternatives as we present later in Chapters 3 and 4. Facilities capable of simulating parallel recursive computations in the GPGPU environment do exist however, and have been explored in literature [21, 141, 108] with mixed results and a hefty list of special considerations.

The seemingly embarrassingly parallel nature of backtracking algorithms can give the false impression that parallelisation of these algorithms is trivial as a whole which is not the case [61]. When parallelising a backtracking algorithm, one has to take into account the characteristics of its implementation, and particularly in terms of memory requirements (i.e., communication requirements between parallel searches, output size, search space size and generation, and memory access patterns) as well as parallel search work balancing. More specifically, in algorithms where data exchange (communication) between parallel searches is required over some medium (e.g., common memory), safe concurrent access has to be ensured at whatever cost this comes with.

Algorithms with unpredictable memory requirements (either for each search state or for their output) pose a further challenge in ensuring sufficient resources are available. CDCL (discussed further in Section 2.1.1), is an example of an algorithm with communication requirements when parallelised and an unpredictable amount of clauses memorised during the search. Memorised clauses can be pruned heuristically to halt the potentially exponential memory growth, yet that requires further communication between parallel searches. Parallelisation of this algorithm and heuristic approaches to efficiently overcoming such issues have been explored widely [30, 115, 53, 14].

The aforementioned factors complicate implementation but most crucially, can have significant performance implications in specialist hardware in particular. In the case of General Purpose Graphics Processing Units (GPGPUs) (discussed in detail under

Section 2.3.1), workloads with unbalanced memory or computation requirements are often refereed to as 'irregular' tasks [108, 142]. On such hardware, ideally, parallel workers (i.e., units each executing a one search in parallel to the rest) should perform the same amount of work in their groups to fully utilise available resources, In practice, this means that the search paths explored by collaborating workers would have to be of approximately the same length and require approximately the same number of steps to explore. Workload balancing and redistribution is an active research area [19], and heuristics can be used to better distribute work in these environments, as we present in Section 4.5.1. Furthermore, the nature of backtracking introduces the requirement for memorisation of earlier search states in order to eventually return to them, which can require significant memory for problem instances with deep search paths [61].

2.3 Computation on Graphics Processors

The need for graphical interfaces on computer hardware dates back to the early days of computing. The first display attached to a machine is attributed to the University of Manchester's Small Scale Experimental Machine (called "The Baby") in 1948 [1], which displayed basic alphanumeric text in a dot matrix display. Graphical displays and graphics continued to evolve through the years, becoming more widely available, until the demanding nature of such processing overwhelmed the available hardware. In 1980's, the first graphics-dedicated processor was created by the Nippon Electric Company in the form of a Very Large Scale Integration (VLSI) chip, with NVIDIA releasing the first single-chip Graphics Processing Units (GPUs), closer to what we know as a GPU nowadays, in the late 1990's [105].

The term GPU has been given different meanings over the years. In more recent times, and perhaps as a result of the widespread adoption of GPUs anywhere from scientific cluster computers to laptops and the associated marketing campaigns, the term is commonly found being used interchangeably with the term "Graphics Card", which refers to computer add-in boards (cards). In principle, a GPU is a specialist coprocessor which can exist in various environments and forms, beyond add-in boards, and should be treated as such.

GPUs are designed to excel in certain operations including matrix multiplication, floating point arithmetic, some trigonometric functions and importantly, parallel computation – all qualities of great importance to graphics-related tasks. Operations needed for graphics-related tasks however are not unique to that domain. Over the course of evolution of GPUs, efforts have been made to map a plethora of non-graphics related tasks to an appropriate format to leverage the capabilities of GPUs and benefit from the superior performance it can offer [107, 68, 132, 134, 22, 5, 142, 100, 102, 104, 64, 98].

The use of GPUs for tasks beyond the graphics domain has given birth to the term GPGPUs in an attempt to better describe the function of these coprocessors. To one, the line between CPUs and GPUs may begin to blur with this in mind, as it appears we are now talking about two different types of generic processors each suited better for some tasks than others. This is not the case however, as calling a GPUs fit for 'general purpose' applications is an exaggeration. As Section 2.3.1 makes it apparent, GPUs are still specialised coprocessors with limited scope and stringent workload requirements which have a broadened area of application beyond what they were initially designed for. With this distinction in mind, throughout the rest of this work, the terms GPU and GPGPU will be used interchangeably unless otherwise indicated by the surrounding context.

The widespread availability of GPGPUs together with the benefits of work sharing between them and CPUs motivates acceleration efforts. Ever since NVIDIA created the first single-chip GPU, they remain a pioneer in the field, developing state-of-the-art hardware and relevant software resources as well as supporting research using their products. Due to prevalence of NVIDIA hardware and resources this work is based on the NVIDIA ecosystem and targets specific hardware and tooling, discussed further in Section 2.3.1 and thereafter in Chapters 3 and 4. In terms of computation model, NVIDIA GPUs are manycore processors and like others, classify as Single Instruction Multiple Data (SIMD) vector processors. SIMD is part of Flynn's taxonomy [40] of architectures and describes those architectures which can execute the same instruction for different data simultaneously. In actuality, the NVIDIA documentation [89] places their GPUs in a somewhat more specific (to GPUs) variant of SIMD which more closely models their architecture, namely Single Instruction Multiple Threads (SIMT). The difference lies in the fact NVIDIA GPUs allow different threads (i.e., the smallest unit of computation) to diverge during computation which is not in conformance with SIMD. Section 2.3.1 presents this model of computation and NVIDIA GPU specifics in more detail.

2.3.1 The CUDA Model

NVIDIA's Compute Unified Device Architecture (CUDA) [89] brings support for general-purpose computation on supported NVIDIA GPUs through a programming interface, drivers, and various tools.

CUDA operates on the principle that one or more devices (GPUs) are connected to and share resources with a host system, which is responsible for sharing data and coordinating computation on the device(s). The host system launches kernels of work on each of the devices associated with it, which are in turn executed by a number of threads. The threads involved in each kernel launch are logically partitioned into blocks, each of which can be mono-, bi- or tri-dimensional. In turn, blocks are logically grouped

Figure 2.2: Illustration of a kernel with a mono-dimensional grid, comprised of three bi-dimensional 8×8 blocks of threads.

into a grid which can also be either mono-, bi- or tri-dimensional. For instance, a mono-dimensional grid of bi-dimensional blocks is illustrated in Figure 2.2. The dimensionality of the blocks and grid is important for tasks exhibiting spatial locality that can benefit from this structure, however, for other tasks this geometry is of little significance and comes without penalty.

In hardware, GPUs are made up of multiple Streaming Multiprocessors (SMs) each of which is allocated a number of blocks which reside* and execute on it. The number of blocks allocated to an SM depends on the requirements of each block in terms of shared memory, register space, block size, etc. An illustration of the structure of each SM in the Ampere microarchitecture, as presented in the relevant whitepaper [91], is shown in Figure 2.3. This figure serves to highlight the fact SMs operate in isolation (w.r.t. one another) as well as the mapping between the earlier discussed logical structure of the kernel and hardware.

In the SM, a resident block is further partitioned into batches of (currently) 32 threads, called warps. All threads in a warp execute in lockstep and should ideally not diverge in their execution. Whilst SMs do not support speculative execution [89] (i.e., branch prediction [114]) nor thread level speculation [89] (i.e., speculative mulithreading) the main problem with branch divergence stems from the fact no two different instructions can be executed simultaneously in a warp. Therefore, due to the SIMT nature of the processor, threads executing divergent code will have to do so at different times (i.e., the execution of the different code segments is serialised) and consequently computation time increases as resource utilisation decreases. In the most degenerated case, all threads in the warp may diverge and each intend to execute different instructions to the rest.

*A block is resident when its threads are initialised and ready to execute.

Figure 2.3: Composition of a full chip implementing the Ampere microarchitecture which comprises of seven GPU Processing Clusters (GPCs), each in turn comprising of twelve SMs. Figure courtesy of [89].

In hardware, the SM is itself partitioned into four partitions as shown in Figure 2.4, which illustrates the structure of each SM in the Ampere microarchitecture, as presented in the relevant whitepaper [91]. Each partition consists of a number of cores, some capable of processing 32-bit floating point instructions only, and others processing 32-bit integer instructions as well. Instructions concerning wider types (i.e., 64-bit types) are not natively supported and are thus emulated. Each SM partition also houses a number of load/store units, an instruction cache, a warp scheduler, a register file, and a number of special function units. During execution, each partition is assigned a number of warps, which remain resident until they conclude. At each cycle, the warp scheduler of each partition selects one of its assigned warps that is ready to execute, and issues either one or two independent instructions from that warp. In some cases, an instruction may need to be re-issued several times (i.e., over several clock cycles) to satisfy all threads in one warp, which is the case for 64-bit instructions for instance. In compute-bound kernels in particular this has to be taken into consideration. The structure of SMs may differ across Compute Capabilitys (CCs) which in turn has implications to efficient utilisation of resources. These implications are detailed in the NVIDIA documentation [89] alongside recommendations for each major CC version.

Figure 2.4: Structure of an SM in the Ampere microarchitecture, comprising of four partitions, each containing a number of arithmetic cores, registers, and other function units. Figure courtesy of [89].

Recent changes in hardware mean that the effect of warp divergence is less detrimental [9] in newer microarchitectures. In particular, the Volta microarchitecture [95] introduces Independent Thread Scheduling whereby each thread in a warp has its own program counter and stack space, unlike previous architectures where a common program counter and stack were maintained for all threads in a warp. This enables the execution of code from divergent branches to be interleaved, thus enabling finer-grain scheduling, but inline with the Single Instruction Multiple Threads (SIMT) paradigm, different threads in the same warp still do not execute different instructions in parallel. Independent Thread Scheduling introduces more benefits, especially for starvation-free algorithms [51], however, it does not come free of cost; an additional register is used per thread to facilitate Independent Thread Scheduling, which we found is a relatively high price to pay for kernels with high register demands, as we detail in Section 4.3.4.

Another special consideration of GPGPUs is the handling of memory. There are several types of memory with different access costs, scope, and sizes. Global memory is the largest memory type on the GPU in terms of capacity, which is visible to all

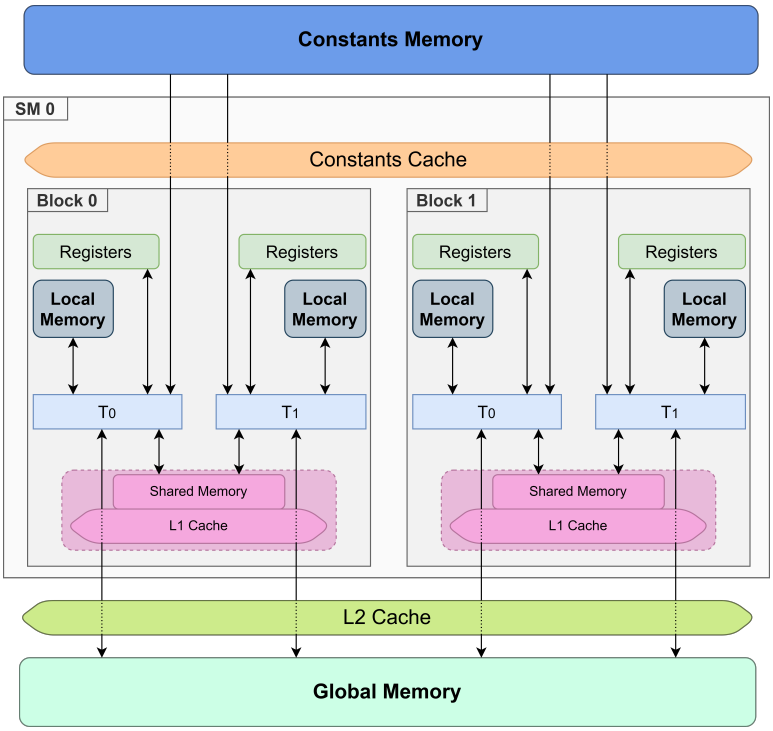


Figure 2.5: Diagrammatic depiction of CUDA memory types, their scope, and interactions with threads.

threads across all blocks. This type of memory is the most costly to access (in terms of clock cycles) but acts in a manner similar to main memory in relation to a CPU. Global memory is a means of communicating data between host and device, as it is memory both sides can manipulate. Shared memory is an on-chip memory region available per SM which has block-scope and is relatively limited in size, albeit being significantly faster than global memory when its access requirements are fulfilled. In addition, each SM has a number of registers available for resident blocks. This number is typically high (i.e., $65,535$ from CC 5.0 to date) however these registers must be shared between resident threads. Blocks requiring more registers per thread than their share may bar other blocks from becoming resident on the SM (for SMs that support multi-block residency) or result in spillage to local memory – a logical, thread-local, cached memory backed by global memory.

Figure 2.5 depicts diagrammatically the relation between different memory types and caches along with the interactions threads may have with each memory type. Memory types in bold (i.e., global memory, constants memory, local memory) are off-chip whereas the rest are on-chip. L1 cache and shared memory occupy the same portion of on-SM memory. The carve-out for the two is configurable by the host system before

launching a kernel. L2 cache is a distinct on-chip region residing outside any of the SMs and acting as a global cache.

Using warp-level primitives, it is possible for threads within the same warp to efficiently perform collective operations involving communication by sharing register contents between them, without the need for intermediate stores to some randomly accessible memory. Such collaborations can take the form of warp ballots (where threads each cast a vote by evaluating a condition, with the results visible to all) or warp shuffling (where threads exchange data with one another in some pre-defined pattern). These operations are synchronous, meaning threads within the warp that may have diverged by that point, will have to re-converge before the operation takes place. Warp synchronisation in that regard can be performed independently and comes with no notable overhead, but in the case of divergent execution, may result in unnecessary waiting. In a similar fashion, synchronisation barriers may be used on the block level* as well, as typically required by collective memory transfers (e.g., global to shared).

Shared memory is divided across a number of banks. While requests by threads to different banks are serviced simultaneously, access to the same bank by threads from different warps may result in a bank conflict. When a bank conflict occurs, the requests are serialised, reducing the overall throughput of shared memory. However, read access by multiple threads within the same 32-bit word causes a single read operation which is subsequently broadcast to all threads involved. Bank conflicts should be avoided whenever possible as they may degrade performance.

As units, threads are 'weak' in terms of computational power, but power is leveraged from the large number of concurrent threads at any given moment in a massively parallel environment such as this. Threads can compute independently of each-other†, but should not be thought about as individual workers. Instead, the hierarchical partitioning of threads into blocks and warps must be taken into consideration. Threads in a warp should execute the same instruction in lockstep ideally, which would mean that certain computationally expensive operations such as memory input/output will need to take place simultaneously by the threads. Whilst a warp of threads is stalled (e.g., waiting for the latency of a memory operation to elapse), the warp schedulers hide this latency by swapping the waiting warp out for one which is ready to compute. Unlike CPUs where context switching is typically an expensive operation, in GPUs it is performed cost-free.

Latencies may be caused for reasons other than memory operations, such as instruction latency or pipe utilisation. Instructions take a certain number of clock cycles to execute

*In current architectures. Pre-Volta architectures did not support this but mimicked similar functionality. See Section 5.9 for more discussion on this point.

†This applies to threads in different warps. Threads within the same warp cannot simultaneously execute different instructions at present.

all the while the respective threads await the results. Warp schedulers may elect to switch to other threads that are ready to execute. Likewise, over-utilised SM pipes may introduce additional latencies by disqualifying some warps from the next scheduling cycle. Warp schedulers are hardware components with the job of scheduling warps as efficiently as possible given the state of the SM partition they control*. The order or heuristics involved in warp scheduling remain largely undocumented, thus there should be no expectations on their behaviour for correct or efficient execution.

Besides hiding latency by warp switching, the cost of memory interactions, and particularly that of reading from global memory, can be reduced through carefully designed access patterns [9]. Specifically, when a thread requests some data from global memory, a memory transaction† will take place which will contain the requested data alongside some adjacent data. When multiple threads from within the same warp perform a memory read for adjacent data in global memory, these reads are combined into a minimal set of memory transactions needed to fulfil them, essentially minimising the associated read latency. Furthermore, since the Maxwell microarchitecture [89], these transactions pass through and may be cached in L2 cache which is common for all SMs. As expected, global memory transactions pass through and may be resolved by L1 and L2 before the expensive global memory interaction is performed. It should be noted that data alignment must be considered with care as, besides resulting in erroneous results when done improperly‡, they can also result in unnecessary instructions and poor cache behaviour. Namely, access to any word of 'standard' size $s \in \{1, 2, 4, 8, 16\}$ at a naturally aligned memory address m§ will be performed using a single instruction.

Typically, communication between host and device happens through memory transfers over a shared bus, specifically to the global memory of the GPU [7]. The speed of data transfers on the bus paired with the overheads of such transfers mean that continuous communication between host and device can act as a bottleneck. On multi-GPU systems the host can individually transfer data to each device as necessary, however, it is possible for data to be transferred directly between devices. The latter may benefit from superior transfer speeds if the devices are linked via a GPU-to-GPU bus such as NVLink [93].

Implementation choices must be made cautiously in terms of communication. Commonly, GPUs are either used to assist/accelerate the computational effort of software running on the host system (hybrid mode) [132], or themselves perform the computation with optional interventions by the host system (complete mode) [83], in which

*Currently, SMs are partitioned into four partitions.

†The transaction window size is predefined and architecture-dependent.

‡The documentation [89] notes that incorrect data will be read if an 8 or 16 bit word is read from a unaligned address.

§A memory address m is aligned up to some size s when $m \bmod s = 0$.

case the host's task is to coordinate the start of the computation, and perhaps assist in data sharing and workload balancing, whilst the actual solving of the given problem takes place in the GPUs. In a hybrid application, the cost of memory exchanges between host and device, along with initial kernel costs, should be weighed against the speedup the GPUs offer to the overall computation. In the case of complete applications such costs are usually irrelevant considering the overall solving effort, but a bigger challenge that arises is that of mapping conventional algorithms to an implementation suitable and optimised for the GPU environment. Consequently, new techniques and adaptations to the algorithm(s) are likely necessary to achieve a good mapping.

2.3.2 Barriers to Optimisation

The nature of GPUs is such, where minute computational savings for each thread can result in great time savings overall, especially in frequently repeated operations across all scheduled threads. Such optimisations can be achieved either through standard language and tooling functionality, or by lower-level adjustments to manually take optimisation opportunities compilation tooling missed.

Lower-level optimisations typically come in the form of assembly-level 'tweaks' embedded in code written in a higher-level language such as C. The translation from the high-level programming language to GPU machine instructions however is a complex multi-step process that may hinder such efforts as it introduces tight coupling to specific hardware and tooling versions.

The conversion process starts when high-level C code is compiled into instructions of the intermediate assembly-like Parallel Thread Execution (PTX) language*. The instruction set of PTX is restricted, but contains complex/high level instructions that do not necessarily map to hardware instructions directly. The PTX Instruction Set Architecture (ISA) is hardware agnostic, meaning that the same PTX code can be translated into machine code for a range of different hardware, regardless of their specifics, as long as they are capable of performing the operations at hand. This is determined through the Compute Capability (CC) of the device which is a version number, that identifies the features supported by the device alongside some of its characteristics. A device of a certain CC version, is capable of performing all operations of that CC, along with operations of all previous CC versions. As is to be expected, PTX has no knowledge of the register space available on each device, therefore, register variables are used instead. These variables don't reflect the final register usage which ultimately depends on the machine instructions used and assembler configuration, which makes it impossible to gauge register pressure at this stage.

*A brief summary of the main PTX instructions used through the rest of this work can be found in Appendix A.1.

The next stage in the conversion of high-level code to GPU machine instructions is the translation of PTX to SASS*. SASS is another assembly-like language which, unlike PTX, is tightly coupled to specific hardware. Hardware register allocation takes place during the conversion from PTX to SASS, and optimisations are applied meaning that the input PTX code will likely not map one-to-one to the output SASS code. SASS code is designed to not be interfered with unlike PTX and NVIDIA does not officially provide documentation for the ISA, nor disclose any instruction performance data (i.e., instructions per clock cycle per SM or clock cycles per instruction) related to it. Ultimately, analysis of SASS code is important when investigating performance improvements. As a result, performance information of SASS instructions for different NVIDIA hardware has been found through micro-benchmarks and is documented in literature alongside a wealth of other measured performance data [2, 143, 121, 77, 97, 62], which proves invaluable in optimisation tasks such as that presented in Chapter 4.

An essential component of achieving good theoretical occupancy† and thus standing a chance at achieving peak resource utilisation, is the careful allocation of the 'precious' resources in an SM. Resources such as registers and shared memory are considered 'precious' as they are limited and reside on-chip, whilst having significantly lower access costs than off-chip options discussed further in Section 2.3.1. While shared memory usage is largely controlled by the programmer, register usage (often refereed to as 'pressure') is dependent on the SASS instructions used, and the ability of the toolkit to simplify the code and reuse registers. The actual number of resident threads per SM depends on the resource requirements of these threads and what is available in the SM. For instance, devices of CC 8.0 can have 2048 resident threads sharing the 65535 available registers (see Table 2.1), allowing each thread to use up to 32 registers. Should the per-thread register requirements exceed 32 however, for example to 33, the maximum number of resident threads reduces to 1985. This, paired with the fact thread blocks cannot be divided across SMs, results in under utilisation of resources in each SM, whilst thread blocks are left waiting to become resident.

The disconnect between PTX and SASS code presents a further complication for optimisation, as we are only able to control hardware-unspecific code. We then anticipate that any changes made to that code, are well received by the transpiler (in its current version) when producing the SASS code. There is however no guarantee that this transpilation will persist in future versions of the tooling. Even after confirming that a hand-crafted optimisation in PTX code has resulted in an appropriate change in SASS code however, the mysticism surrounding this conversion from PTX to SASS and the SASS ISA itself means the optimisation is not guaranteed to work as intended with

*The expansion of this acronym is speculated to be '**S**hader **ASS**embly' however, to the best of our knowledge this is ambiguous and not documented officially.

†Theoretical occupancy is the ratio of active warps over the maximum active warps supported by an Streaming Multiprocessor (SM).

Architecture	Maxwell (2014)			Pascal (2016)			Volta (2017)		Turing (2018)	Ampere (2020)			Ada Lovelace (2022)	Hopper (2022)
Compute Capability (CC)	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.2**	**7.5**	**8.0**	**8.6**	**8.7**	**8.9**	**9.0**
Shared Memory per SM (KB)	64	96	64		96	64		96	64	164	100	164	128	228
Max Resident Warps per SM					64				32	64			48	64
Max Shared Memory per Block (KB)			48				96	48	64	163	99	163	99	227
Resident Threads per SM					2048				1024	2048			1536	2048
32-bit Register File Size									65536					
Number of INT32 ALUs per SM		128		64		128					64			

Table 2.1: Device features for a range of Compute Capability versions.

future versions of the hardware other than what was targeted in the conversion, making the tuning effort very tightly coupled to both hardware and toolkit versions. Of course, one can expect hardware-specific optimisations to result in specific hardware dependence, however the intermediate layers of abstraction further complicate these efforts.

An additional obstacle in performance optimisations relates to the frequent, non-linear fluctuations in hardware capabilities throughout the evolution of GPUs. Whilst on the instruction level, successive CC versions are backward compatible (i.e., able to perform the same functionality) with one another, assumptions made during development may be invalidated as a result of these fluctuations. Table 2.1 presents a variety of device features for different CC versions. Devices of CC 7.0 (Volta microarchitecture) for instance, feature 96KB of Shared Memory and support up to 2048 threads resident per SM. The later CC version 8.0 (Turing microarchitecture) however features 64KB of Shared Memory per SM and support up to 1024 threads resident per SM. This changes once more for devices of CC 8.6 (Ampere microarchitecture) which feature 100KB of Shared Memory per SM and support up to 1536 threads resident per SM. To achieve near optimal performance, the code has to take into consideration the specific characteristics of the architecture in question, which may impact the design choices made in the process. The necessary coupling of implementation choices to the architecture

introduces hand-tuning opportunities, but does so in a non-forward and non-backward compatible manner as has been documented in literature [37].

Regardless of the difficulties however, workload optimisation is a lucrative task which can have a significant effect on resource utilisation and by extension, runtime, as well as financial implications especially in modern pay-as-you-use cloud computing environments [63]. As observed in [37, 63] compilers and tooling alone produce efficient machine code, but peak efficiency typically requires intervention through hand-tuning. More specifically, the authors of [62] have conducted a thorough review of the hardware specifics of the Turing microarchitecture, and in the presentation [63] of their findings in the NVIDIA GTC2019 conference, state that code generated by NVCC, NVIDIA's CUDA compiler toolkit, achieves approximately 80% of the theoretical efficiency for compute-bound kernel. This efficiency, whilst undismissible, leaves plenty of room for improvement through hand-tuning. Such efforts can happen at any stage of the transformation from high-level code to machine instructions, but ultimately require deep understanding of and a wealth of performance data for the targeted hardware to achieve.

2.3.3 Issues of Compatibility and Reproducibility

Results in literature can act as a point of reference when establishing new techniques to achieve the same or a similar effect. Such comparisons often provide insights into the effectiveness of proposed techniques, their strengths and limitations, and can prove invaluable in deciding future directions for a body of work.

In the field of GPGPU computing however, several considerations exist which may sometimes prevent accurate conclusions from being drawn. Unlike the much more forgiving and general-purpose environment of the CPU, the complexity of GPU hardware means some choices have a significant effect to performance. For one, the tools and hardware evolve continuously with major (sometimes breaking) changes between successive versions (as briefly discussed in Section 2.3.2). The authors of [37] note that optimal performance cannot be achieved purely through libraries and tooling provided (i.e., without hand-tuning optimisations) and demonstrate through their experiments the lack of portability of performance assumptions (i.e., difficulty of gauging performance and the effect of optimisations without deep understanding of the specific hardware targeted) – a view we also echo through our work in Chapters 3 and 4. Our work in [102] where we present two kernel implementations of the same algorithm, serves as an attestation to the effects of meticulous optimisation on these platforms. In our work, the key difference between the two kernels is the (enforced) use of registers as means of storage in one, versus using on-chip shared memory on the other whilst the principle of operation is mostly identical. A rather small change like this yields a consistent improvement in measured results (in newer hardware), as explained in

Chapter 4. Other work in the field highlights issues of reproducibility of results concerning the use of GPGPUs, such as the authors of [21] who mention in their publication that in contrast to [141], they have identified important (for result integrity even) limitations of CUDA's dynamic parallelism through their work on the parallelisation of backtracking search algorithms for GPGPUs. Such differences can be attributed to the complex nature of this hardware, tooling, and the myriad of (sometimes undocumented) considerations to be accounted for when using them.

Another important factor in result reproducibility is the level of detail given of the work that lead to these results. Oftentimes space limitations imposed to works in literature, lead to the omission of crucial detail, required to reproduce the results shown. We ran into this difficulty when attempting to faithfully follow the steps of [83], and had to make certain design choices in the process which lead to very different results, as discussed in Chapter 3. The story is similar for [142] where the authors provide thorough and in-depth walk through of their methods and results, but once again due to the intricacies of the hardware and small but important details omitted, we are unable to reproduce these results to test their accuracy and applicability on modern hardware. This of course is not criticism of the authors of this work nor their intentions, but merely a remark on difficulties in the field as a result of the hardware and tooling complexity and to a lesser extent, publishing practices.

A knock-on effect of poor reproducibility of results and methods is the impact to fair comparison between approaches. Establishing a baseline for the effectiveness of a technique versus another is non-straight-forward even when the same metrics have been gathered for both. The task becomes more arduous when concerned with gathering runtime-dependent metrics (e.g., execution time, as is often the case [142, 108, 21, 30]) where the list of factors potentially affecting them grows.

Factors such as the library and compiler version, compile flags, communication bus capabilities, CPU and GPU models, memory, clock speeds, and many more can impact runtime-dependent results and must be considered. Furthermore, environmental factors become an issue considering the (commonly implemented) 'throttling' of CPUs and GPUs when approaching thermal limits, for which ambient temperature and ventilation must be considered. The pitfalls of runtime comparison (and other such metrics) are discussed further in [70]. Strangely, fair comparison of results is not always considered in the literature. For instance, in [45] the authors propose a search algorithm for the N-Queens problem (discussed in detail in Section 2.1.2). In their comparison, the authors argue their approach is definitively better than other work in literature and compare runtime results, without acknowledging the major differences in systems used to obtain those results, nor conditions of the experiment, nor providing tests on different systems to show a consistent improvement and in places even neglect to convert time results to the correct time unit.

In this work, when runtime results are gathered and comparison with other literature is made, it is intended as a reference for the reader. We have, to the best of our ability, tried to control the effects of external factors and provide as fair and accurate comparison as possible. The specifics of such comparisons are described in the relevant sections. The devices used to gather results for this work are also discussed in the relevant sections and have been briefly summarised in Table 2.2 for convenience.

	GTX 1080ti	RTX 2080ti	RTX 3090ti	Tesla A100	RTX 4090	Tesla H100
CUDA core count	3584	4352	10752	6912	16384	16896
L2 Cache (KB)	2816	5632	6144	40960	73728	51200
Shared memory (KB)	96		128	192	128	256
Shared memory / block (KB)	48	64	99	163	99	227
SM count	28	68	84	108	128	114
Microarchitecture	Pascal	Turing	Ampere	Ampere	Ada Lovelace	Hopper
Compute Capability	6.1	7.5	8.6	8	8.9	9
Global memory (GB)	11	12	24	40	24	80
Base clock (MHz)	1480	1350	1560	765	2235	1095

Table 2.2: Summary of devices used in this work, alongside their respective characteristics.

2.3.4 Adaptation of Algorithms for Implementation on GPGPUs

With relatively new technology such as the GPU, the question is raised, how can existing algorithms be mapped to make use of its resources? On the surface, it is no more complex to provide an implementation of an existing algorithm on the GPU than it is on the CPU. Complexity however increases significantly when we want the implementation to be performant and to make use of the special characteristics of the GPU coprocessor. This is to say that before a performant implementation can be identified, the algorithm should be moulded around some requirements and have any necessary adjustments made. We cover this point in detail later on in our work, under Sections 5.2 and 5.3.

Adapting existing algorithms to map to the GPU can be broken down into two subtasks; preparing the algorithm for a massively parallel environment and preparing it to conform to the special requirements of the GPU. In some cases, the former is relatively easy as the algorithm lends itself to massive parallelisation, as the authors of [10, 64, 108] have found in their respective work. This is not the case however for all algorithms [39] and especially those with large sequential components or common data dependencies requiring data synchronisation, such as the CDCL procedure for SAT solving [140]. Algorithmic adjustments to conform to the GPU's specialities may be somewhat more tedious to accommodate. In their work, the authors of [83] decompose a sequential 3-SAT algorithm into a sequential pipeline of parallel components, apt to

the requirements of the GPU, in an effort to overcome parallelisation difficulties. We approach this algorithm from a different angle in our work as detailed in Section 3.2 and discuss these difficulties in further detail. The authors in [22] also take a similar approach to parallelise the quicksort procedure where once again, a sequence of steps in the form of kernels are used, enforcing global waiting between each.

Steps taken to amend existing algorithms in preparation for a GPU implementation may on occasion not be enough to achieve good performance which is typically the objective. In the field of SAT where current algorithms are heavy on logic operations one would expect GPUs to be prevalent, especially factoring in their high capacity for such operations versus state of the art CPUs, yet we are not aware of any widely adopted GPU-based solver for any variant of the problem, despite the wide spread availability and affordability of these devices. This is attributed by the authors of [13] to the need for fundamentally different algorithms to those constructed with the CPU's model in mind. This is a view our later work in Chapter 3 comes to support. In these cases inspiration can be drawn by established algorithms to produce alternatives that are better suited for the GPU, but as discussed in Section 5.3, in doing so one may need to make some perhaps unconventional algorithmic choices.

Chapter 3

Scalable GPU SAT Checking

Contents

In this chapter we use present our work on the Boolean Satisfiability (SAT) problem. SAT has historically been a 'hard' problem that is commonly solved using backtracking algorithms [140]. From the perspective of GPU exploration, SAT is an example of a memory-bound workload that requires careful planning to achieve sensible data locality.

Our exploration of the problem takes us through two alternative approaches; one assessing the feasibility of a complete in-GPU SAT solver using a bespoke backtracking algorithm, and one where the GPU is designed to act as a small sub-problem solver under the coordination of the host system. The latter approach is designed to be loosely coupled and therefore openly scalable. The GPU-side code is created so it makes best use of available resources whilst mitigating the effects of heavy dependency on global memory where possible. Emphasis is placed on optimising the implementation, mainly relating to memory management and manipulation.

We conclude the chapter with an evaluation of the applicability of GPUs in their present state to SAT as well as some ongoing and future work we wish to undertake.

3.1 Approaches to SAT Solving

The significance of the Boolean Satisfiability (SAT) problem has lead to the development of numerous solvers for each variant of the problem, and the establishment of SAT competitions [27] during which, solvers are being benchmarked against one another to find the fastest among a range of inputs. Following hardware evolution trends, focus has nowadays shifted to parallel and distributed solvers, of which numerous successful examples exist [79, 104, 109, 86, 41, 82, 49, 75].

Over the years, numerous powerful CPU-based solvers have been developed with notable examples including MiniSAT [34], ManySAT [49], and more. Common to state-of-the-art solvers such as MiniSAT, is the use of some implementation of the Conflict Driven Clause Learning (CDCL) procedure discussed in Section 2.1.1, which at its core is an exhaustive search employing a number of powerful heuristics to guide the search space exploration into one solution. In the context of the decision problem, a solution is a proof that the formula can or cannot be satisfied, without necessarily identifying the assignment under which this is the case. In this work we consider the 3-SAT problem specifically (i.e., formulae in 3-CNF), to which the CDCL procedure can apply the same way it does to general formulae.

The restriction of SAT into 3-SAT presents opportunity for other algorithms besides CDCL. Notably, the authors of [56] present an algorithm specific to 3-SAT, exploiting the characteristics of the problem. Their divide and conquer algorithm performs a depth-first search on a given formula, by exploring up to 3 literals per clause. Importantly, at a certain cut-off point, a brute-force search for the remaining literals is launched. The algorithm is summarised in Listing 3.1 in the function `solveDnC` which is supplied with a formula ϕ, an (initially empty) partial assignment μ, and two constants C_v,C_c which represent the depth and number of literals before the brute-force search is started.

In Listing 3.1, the function `isSat` at Line 7 is used to check if the (partial) assignment μ satisfies ϕ. The function `selectShortestClause` returns the clause with the fewest unsatisfied literals, and the function `satisfyLiteral` returns a new assignment derived by assigning the corresponding variable of some literal l in μ, such that l is satisfied. This lends itself to a heterogeneous implementation [100], with CPU solvers being used before the cut-off and massively parallel GPU solvers being used for brute-force exploration after this cut-off. The variable selection process in this algorithm is rather naïve and could be replaced by a more sophisticated heuristic. Selecting the variable that satisfies the most clauses from the ones that are currently unsatisfied would perhaps be a better option as it would have the highest overall impact and may result in a certain subset of variables not being explored at all. Random variable selection might also be appropriate for formulae in which variables generally

```
1  fn solveDnC(φ,μ,C_v,C_c):
2    if (|unsetVariables(μ)| ≤ C_v ∨
3        unsatisfiedClauses(φ,μ) ≤ C_c) then:
4      return brute_force_solve(φ, μ);
5    endIf
6
7    if (isSAT(φ,μ)) then:
8      return SAT
9    endIf
10
11   let c ← selectShortestClause(φ,μ);
12   foreach (l ∈ c):
13     let μ' ←satisfyLiteral(μ,l);
14     if (solveDnC(φ,μ',C_v,C_c) =SAT):
15       return SAT
16     endIf
17   endLoop
18   return UNSAT
19 endFn
```

Listing 3.1: A divide and conquer algorithm for 3-SAT.

affect the same number of clauses (e.g., in randomly generated ones).

Powerful solving algorithms like CDCL supersede the aforementioned algorithm and others which are brute-force in nature, in terms of algorithmic and practical performance in the general case. Special cases and inputs of the problem that hinder the performance of these powerful algorithms do exist [35, 71, 23]. For instance, the conflict clause learning schemes employed by CDCL can, for some inputs (e.g., pebbling formulas), outnumber the clauses in the original input formula and deteriorate performance [71].

For 3-SAT, which is the focus of this chapter, existing work has identified phenomena that render some inputs harder to solve (and on occasion, substantially so) than others. In their work, the authors of [23] established that the hardest instances often lie at the "phase transition" point, where the probability of satisfiability changes sharply. This transition is characterised by a high density of local minima which traps solvers in exploring the many near-solutions. Moreover, in [85] the authors provide an analysis of the behaviour of the DPLL method on 3-SAT (which extends to k-SAT) instances and the impact the number of variables and clauses has to the number of backtracks the solver makes when searching for a solution. They show that the hardest instances

occur near a critical clause-to-variable ratio, approximately 4.25 for large formulas. This is the "phase transition" region where the probability of satisfiability is close to 50%. The authors distinguish the behaviours exhibited between satisfiable and unsatisfiable instances at the transition point. They identify that satisfiable instances are difficult since satisfying assignments are sparse in that situation, whereas unsatisfiable instances require extensive exploration to prove the impossibility of a satisfying assignment.

Parallelisation of such algorithms is an ongoing challenge discussed further in Section 3.1.1, however, often parallelisation efforts are made in the context of multicore processors such as modern day CPUs. Multicore processor architectures are perhaps more forgiving than their manycore counterparts since they typically feature a shared memory space available to all parallel workers that enables effortless communication between them. Manycore processors on the other hand, similar to distributed systems, do not normally benefit from such luxuries and require careful algorithmic design. In these processors, the scalability of an approach combined with loose coupling is core to achieving good performance.

The drive for faster SAT solvers however has not deterred efforts to integrate manycore coprocessors such as GPUs [30] and FPGAs [144] to form hybrid* or non-hybrid (complete) solvers. Yet, to the best of our knowledge, no GPU- or FPGA-based solver is widely adopted currently for generic SAT solving [13]. This appears to be the case even in variants of the problem which are perhaps more inviting (for reasons we discuss in Section 3.1.1) for this type of hardware and type of computation expected, such as #SAT [122]. Literature [13] identifies this 'gap' in GPU and FPGA adoption in the field of SAT, and suggests that a change in the way the problem is approached algorithmically is perhaps needed in these environments, a proposition often met with resistance. Our work underpins this suggestion and suggests some reasons why this is the case as well as possible mitigations and different algorithmic approaches for SAT, N-Queens, and by extension, other combinatorial problems.

3.1.1 SAT Parallelisation Strategies

Parallel SAT solvers are generally divided into two categories; search space splitting and portfolio solvers, which can broadly be visualised as seen in Figures 3.1 and 3.2 respectively.

*A hybrid solver in this instance is running on a main processor and delegating work to one or more coprocessors during solving.

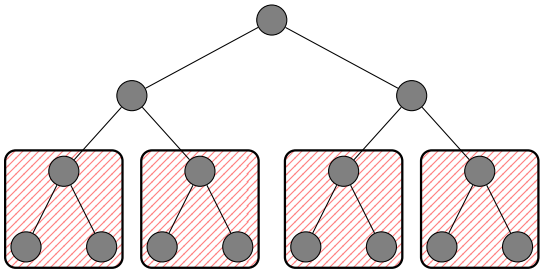


Figure 3.1: Visualisation of a search space splitting approach, dividing sub-search spaces among instances of the same solver.

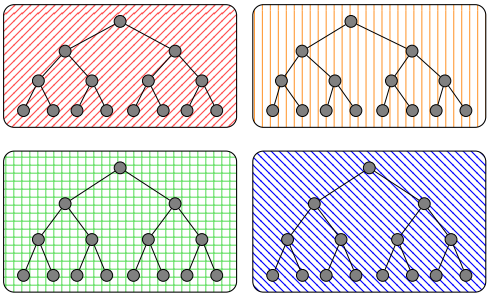


Figure 3.2: Visualisation of a portfolio approach that uses four different, or differently configured solvers over the same search space.

Portfolio solvers [49] operate by running several different searches, each based on a different algorithm* over the same input until one succeeds in satisfying or proving the input unsatisfiable. Workers in a solver such as this are not restricted from interacting with one another and sharing information, for example in the case of CDCL, a common pool of learnt clauses may be used over differently configured searches. Likewise the mixture of approaches permits searchers aiming to satisfy the formula to run concurrently with those aiming to prove the formula unsatisfiable.

Search space splitting solvers [41], as the name suggests, operate by dividing the search space into discrete sub-regions which are in turn explored by parallel searches, typically performing the same algorithm. This strategy is perhaps more common [81] in parallel SAT solving, and revolves around a 'good' partitioning of the search space being made to balance workload among parallel searches.

In terms of GPU utilisation in search efforts, the more prominent approach for full fledged GPU solvers is search-space splitting, with some examples available in literature [83, 30]. This is to be expected since, as discussed in Section 2.3.1, GPUs leverage power from performing the same instruction simultaneously over many threads, in line with their SIMT nature. This is not to say of course that a portfolio approach is infeasible on GPUs, as for instance blocks, resident on different SMs, can perform a different task between one another without inherent performance penalty. However, this is an idealistic situation where resources can be shared between the different parallel tasks efficiently, and block execution can to some degree be controlled. For reasons discussed in Section 2.3.1, and from our experience described in Section 4.5.1, an approach such as this will, in the current state of GPU computing, be plagued by obstacles that likely outweigh any benefits. It is therefore to no surprise that such approach

*Or differently configured algorithm relative to the rest.

has not yet been successfully implemented.

The authors of [83] present an intriguing approach for an exclusively GPU-based search space splitting solver implementing the algorithm presented in Listing 3.1 of Section 3.1. The parallelisation strategy employed is dynamic and begins with threads in the device branching* to generate partly explored search states as part of a pre-processing step. Subsequent branching during search will inevitably result in more formulae being generated during search. Due to the unbounded nature of such 'loose' branching, the authors employ a state exchange scheme between host and device to 'offload' surplus states to keep within the bounds of the, admittedly limited at the time, global memory. Following the initial pre-processing kernel, their technique involves a stepped kernel approach comprised of five kernels with each performing a different function over the set of search states. Their approach is heavy on prefix scan kernels and reorganisation of global memory before applying the "SAT kernel" which in turn explodes the search states by exploring them further. Between runs of their kernel pipeline the authors note that the swapping step takes place during which the host 'steals' some search states from the device to leave room for new states to be generated.

The results presented by the authors are encouraging, however, no comparison to a respective CPU-side solver is given, and it is difficult to gauge bottlenecks in performance and the accumulated cost of kernel launch/end and state transfer. We attempted to replicate this approach with the intention of optimising it as described in Section 3.2, however, due to the absence of detail from the publication we were unable to reproduce these results. Nevertheless, we do believe stepped kernel pipelines can be efficiently used in solving problems experiencing state explosion. Perhaps a limiting factor here is the unbounded nature of state generation during search; in the event of significant surplus state generation, resources on the host-side are likely to cause performance degradation (more frequent data transfers) or at worse case, exceed host-side capacity, something that is not addressed in this work. Likewise, host-device collaborations such as this couple the two tightly, restricting the scalability of this approach significantly.

Implementing a complete GPU-based SAT solver presents several challenges that hinder their current feasibility. SAT algorithms such as CDCL typically require significant amounts† of high-throughput randomly accessible memory which on the GPU is currently only matched by global memory, that incurs significant access costs especially when accessed in a non-coalesced manner. Furthermore complex control flow within threads is likely to result in a high degree of warp divergence, as experienced by the authors of [30] who also present ways of reducing it. Lastly, as previous work high-

*By making assumptions on the truth values of variables based on a given clause, in line with the implemented algorithm.

†Relative to on-chip memory availability on the GPU.

lights [142, 37, 63] and as our work demonstrates (see Section 4.3.6) the current state of GPU development tooling still requires a great amount of hand-tuning optimisations to achieve higher performance. In the case of such complex algorithms, assessing the effectiveness of hand-tuning optimisations to the implementation will be arduous.

Existing search space splitting approaches do show promise and may lead to a competitive solver in the future, however, to date they have been competitive only against non-parallel CPU-based counterparts [13], in part due to the strict computational model that is not suitable for these algorithms. Heterogeneous approaches on the other hand have been proposed and to date, have lead to good results. Such approaches are polymorphous but fundamentally involve sharing work between CPU and coprocessor(s) such as GPUs. In recent years, the authors of [99] have successfully employed a heterogeneous approach which employs a CDCL search on the CPU side, which interleaves formula simplification during searching, and offloads it to the GPU. This work is of great interest, as it demonstrates the potential of GPUs to the field, when used appropriately and comes to complement other work in literature [125] demonstrating performance gains resulting from coprocessor utilisation in heterogeneous approaches.

3.2 Full In-GPU SAT Solving

Our initial experimentation begun with an attempt to implement a fully GPU-based 3-SAT solver following the steps of existing literature, in an effort to explore optimisations on this workload. The work in [83] appeared an ideal candidate as it was presented a number of years ago, during what time the state of GPU technology has evolved significantly. In particular we wanted to investigate each of the following:

- The effect of modern L1 and L2 caches.
- How to best leverage (implicit or explicit) caching for each kernel through data control.
- Hand-tuning opportunities arising in the process.

Furthermore, we speculate some kernels in the pipeline can be grouped with no performance penalty to eliminate kernel launch overheads. Lastly, the authors rely on assumptions on the behaviour of warp schedulers for their work balancing which, although perhaps well informed at the time, is not officially documented and hence leads to an interesting question of how such assumptions hold in today's hardware. Due to the lack of detail at hand however and the inherent intricacy of GPU kernels in which subtle changes can have profound effect, we were unable to replicate the authors approach.

We therefore employed a different implementation of the algorithm presented in Listing 3.1 comprising of a single kernel. In the same manner as our later work (Section 4.3.1), a number of initial search states is generated on the host-side by exploring the search tree up to a certain cutoff depth, resulting in enough states to occupy each thread.

Each state is represented in practice as a fixed-size bit vector comprised of $B = 2 \times |vars(\phi)|$ bits, composed as an array of $\left\lceil \frac{B}{32} \right\rceil$ 32-bit unsigned integers. Each assignment is therefore broken down into two bits used to encode the following three values: $00 \rightarrow UNSET$, $01 \rightarrow TRUE$, $10 \rightarrow FALSE$. The integer representation for each of the values was chosen specifically to aid performance and critically reduce the number of operations required when checking if the assignment satisfies a given literal, as explained later in Section 3.3.4.

The formula itself is represented as an array of structures (clauses) each comprised of three 32-bit integers (one for each literal). The formula can vary in size greatly and therefore resides in global memory throughout computation. We saw little benefit in a more dense representation as the next sensible choice would be a single 32-bit word per clause, partitioned into three 10-bit sections. This would not restrict formulae very much as there can still be up to 512 variables when each 10-bit section is treated as a signed integer, however, more bitwise instructions would be needed to unpack the literals and determine their polarity. The emergence of new technologies with vastly different capabilities however, paves the path to revisit this decision in the future, as we intend to do, and we discuss this in Section 3.5.1.

3.2.1 Implementation Specifics

Since recursion is not easily possible on the GPU*, we implement this algorithm iteratively and preserve a fixed-size array of 'frames' (similar to recursive stack frames) in shared memory to facilitate backtracking. Each frame holds two pieces of information encoded as a 32-bit vector: the clause that was branched upon and the literal satisfied. The vector is partitioned into 30+2 bits with the 2 most significant representing an unsigned integer identifying the literal within the chosen clause that was satisfied, and the remaining representing an unsigned integer corresponding to the chosen clause from the formula. The number of frames in this array is $L = |vars(\phi)| - C_v$ where C_v is the number of unassigned variables before a brute force search is initiated. To contextualise, on a device with 64KiB of shared memory†, each thread within a block may

*Recursion is possible to a small depth. Dynamic parallelism may be used to emulate it, however results on this approach have not been promising [108].

†This example is based on devices of Compute Capability 7.5. Devices with CC ≥ 2.0 and ≤ 6.2 permit only 48KiB of the available 64KiB to be used by a single block.

maximally hold a stack of $\frac{\frac{65536}{1024}}{4} = 16$ frames. Space diminishes however with the assignment of each thread which must also reside in shared memory, the size of which depends on the number of variables in the formula as previously described.

The brute force operation would normally require an assignment buffer separate to the existing thread's assignment to not interfere with the state if unsuccessful. In the interest of space conservation however, this operation re-uses the existing buffer in a fully reversible manner. The operation does not require the additional bit used per literal as no distinction is made between unset variables and true/false assignments. As such, the brute force operation treats unset (00) variables as 'false', and utilises the remaining unused bit pair value (11) as 'true' in its own context. If the brute force operation satisfies the formula, it updates the assignment mapping $00 \rightarrow 10$ and $11 \rightarrow 01$ before exiting such that the assignment is completed and remains well-formed. Otherwise, it reverts $11 \rightarrow 00$ effectively returning the assignment back to its previous state (that is, before the brute force search).

Following the brute force operation a check is made using the current assignment against the formula, to establish if this thread has found a satisfying assignment. If that is the case a global flag is set atomically (bypassing on-chip cache) to terminate the search. Each thread reads this flag before each advancement of its search state and will exit without taking any action if the flag is set.

The branching clause and literal selection heuristic employed by each thread first finds the shortest clause (i.e., clause with fewest branching candidate literals) and subsequently identifies the first (in order) unset literal. Branching occurs by satisfying this literal (i.e., assigning the corresponding variable in the thread-local assignment). This variable selection mechanism is not chosen for its efficacy in reducing the search space, but for their computationally 'cheap' nature in a massively parallel environment. A random choice would also be relatively cheap to implement, but would result in non-deterministic behaviour which is undesirable. The computation of each thread is broken down into the following steps, between which warp synchronisation barriers are interleaved:

Step 1 Check global stop flag. If set, exit.

Step 2 Check brute force conditions and brute force search if necessary.

Step 3 Check satisfiability using current assignment. If UNSAT continue to the next step, otherwise set global stop flag and exit.

Step 4 Backtrack if necessary, select branching clause and literal.

Step 5 Construct frame, write frame to shared memory stack, satisfy chosen literal.

Step 6 Update shared memory assignment.

Here, Step 1 performs a global read of a 32-bit 'flag' determining if the rest of the steps will be followed or not. This operation is set to bypass cache, however, since all threads within the warp request the same address simultaneously only one 'expensive' global memory transaction is issued and the result broadcast to all threads. Following this, Step 2 takes place and naïvely enumerates truth assignments for each unset variable. This operation is data-dependent and therefore likely to be divergent between threads in the same warp, with worse divergence for larger values of C_v. In Step 3 our approach checks the current assignment against the formula in the manner described later in Section 3.3. Following this check the threads branch onto the next state in Step 4 backtracking* to an appropriate state if required to do so. If a thread clears its frame stack, it exits. For threads that successfully select a new literal to branch on they construct and store a stack frame in Step 5, as described earlier and update their assignment accordingly in Step 6.

3.2.2 Computation Bottlenecks

The steps of our implementation outlined in Section 3.2.1 have been chosen to maximise computation happening in lock-step across warps, however certain steps such as brute force searching inevitably lead to divergence, with re-convergence taking place at the next warp-level barrier. This approach benefits from reduced latencies compared to the work in [83] since there is a single continuously running kernel as opposed to a pipeline which starts and stops. Divergence is also controlled to maximise lockstep computation where possible. A drawback of this approach is that it suffers from significant memory constriction leading to low SM occupancy as a result of global memory operations together with reduced block sizes. In our initial experiments, this kernel was outperformed by a single threaded CPU implementation of the same algorithm, using a similar structure, as half-blocks (512 threads) occupied all shared memory per SM despite our hardware at the time† permitting up to 2048 threads and up to 32 blocks (whichever is lower) per SM. We attribute this to an amalgamation of factors with varying severity including: poor warp workload balancing leading to tail effects, global operations, and on-chip resource exhaustion.

The heavy reliance of this type of solver design on shared memory resources makes it an unlikely candidate for devices of most CCs to date, however, more recent hardware presents substantial increases on this resource. For instance, devices of CC 9.0, allow for 227KiB of shared memory to be used by threads of a single thread block, and may have up to 1024 threads resident per SM (one full block). Devices of this CC such as the NVIDIA H100 are intended for data centre use, however it is our estimation

*Backtracking simply sets the branching variables back to 'unset' (00) and moves the stack pointer back.

†One GTX 1080ti GPU which implements the Pascal microarchitecture.

that through the fast paced evolution, in coming years, such hardware will become commercially available. Revisiting our earlier calculation, on these devices each thread in a full block may use a, more realistic, stack of maximally $\frac{\frac{232448}{1024}}{4} = 56$ frames. At the time of writing, we are awaiting early access to an NVIDIA H100 node on Swansea University's AccelerateAI cluster [12]. Therefore we have set this design and associated experiments aside for the time being but wish to explore it further once this hardware becomes available to us, as detailed in Section 3.5. Present restrictions in hardware capabilities severely limit this approach baring it from handling meaningful inputs. We therefore chose to investigate a hybrid approach which utilises GPUs to support CPU-side searches as detailed in Section 3.3.

3.3 GPU SAT Checking for Assisted SAT Solving

Following on from our investigation of complete GPU-side SAT solving, we shifted focus to an alternative hybrid approach aiming to use GPUs to support search performed on the CPU side. Heterogeneous approaches for SAT are more prevalent in the literature with recent work [99] presenting good results. In [138], the authors used MiniSAT [34] with unsatisfiable instances of up to 110 literals, and observed that 45% of the CPU's time was spent in deciding or deducing the last 20 variables. Based on this observation the authors propose an FPGA based hardware brute-force solver to quickly explore problems with a restricted number of variables. At present, FPGAs are not widely available in existing systems due to their programming intricacies and limited scope which perhaps limits the scale of their applicability. In contrast, generically programmable GPUs are widely available and evolve continuously [95, 91, 96]. We hypothesised that similar search can take place on GPUs, with good results, in a fully scalable manner. This led to the development of a GPU-based low-tree SAT checker, designed to work in collaboration with a CPU-side search. Our focus is the GPU component of this solver, detailed under this section.

3.3.1 GPU Component Design

Our approach is loosely coupled, designed to scale across any number of GPUs with each device involved holding a copy of the formula in question in its memory throughout the search effort. Devices await partial assignments from the host system with the intention of completing them with a quick naïve search towards a potentially satisfying assignment. The device-side computation is launched once the partial assignment in need of completion is transferred to the respective device's memory. Following this, a single-shot kernel is launched with T many threads (the geometry of the grid and blocks is unimportant in this instance).

We employ a single-shot kernel in which each thread is given an equal amount of assignments to enumerate. Specifically, on the thread level, for an assignment with V many unset variables, each thread (of T many) is tasked with checking $K = \frac{2^V}{T}$ of them. Work distribution takes place at no cost, as each thread is assigned a unique sequential index starting at 0, which is in turn used as an encoding of the assignments that should be checked by the thread in question in a manner explained in Section 3.3.3. All threads have access to a *limit* representing the number of partial assignments they each should enumerate and check, as well as the partial assignment μ_p which is the partial assignment generated before the device-side brute-force search is started. In addition, a thread has knowledge of the total number T of threads working on the current problem. A thread's objective, is to find a partial assignment μ_l such that $\mu_p \cup \mu_l$ is a complete satisfying assignment.

Upon block start, threads within the block each transfer part of the input partial assignment μ_p, into shared memory. The representation of μ_p is dense and can easily be accommodated into shared memory. More specifically, each variable assignment in μ_p is represented as a pair of bits as described in Section 3.3.4, and the collective of these bit pairs are stored as an array of 32-bit bit vectors. Each vector can therefore hold 16 variable assignments in the space of 4 bytes. Taking for example the Pascal microarchitecture where 64KiB of shared memory are available per SM* and two blocks may reside on the same SM at once, assuming an even partitioning between the blocks, up to $\frac{32768}{4} \times 16 = 131072$ variable assignments can be stored per block. Although μ_p is not modified by any thread under any circumstances, each block still needs a separate copy in its own shared memory as, unfortunately, it is not currently possible for blocks to share this space.

Within the block, the first W many (where $W \times 4$ is the number of variable assignments in μ_p) threads each copy one 32-bit word of μ_p into shared memory, accessing global memory in a sequential pattern. A block-level barrier is placed subsequently, before the threads each construct their own assignment. The assignment μ_l of each thread is a 64-bit bit vector as explained in Section 3.3.3. Each thread initialises its local assignment by computing $(T \times D) + I$ where T is the number of threads participating in the effort, D is the identifier of the device in the host system† and I is the thread's unique identifier within the grid.

Threads then begin a cycle of checking and local assignment advancement, performed in lockstep. To check an assignment, threads each explore the input formula clause-by-clause, deciding if any of the literals within the clause have been satisfied, in the man-

*The 64KiB are by default partitioned into to 48KiB and 16KiB to be used as shared memory and L1 cache respectively although this can be changed. Each thread block is restricted to using up to 48KiB.

†A sequential identifier beginning from zero which uniquely identifies the device in the host system.

ner presented in Section 3.3.4. Unavoidably, this operation leads to warp divergence as threads that reach a contradiction (i.e., unsatisfied clause) stop checking, whilst their peers continue. We therefore have placed a warp-level barrier following this check to regroup threads before moving to the next operation (i.e., local assignment advancement). Advancing the local assignment is a trivial process due to its representation. Specifically, the 64-bit bit vector used by each thread is arithmetically incremented by T which results in a new assignment for each thread, and guarantees no two threads can have the same assignment*.

Execution of threads in lockstep during the checking process is important considering the memory access pattern they exhibit. All threads within a warp will attempt to access the same 12 bytes (i.e., clause) at once. For devices with CC ≥ 6.0, global memory access requests by the threads are grouped in as many 256-bit memory transactions as necessary to service them all. In this instance a clause fits within a single transaction which is subsequently broadcast to all threads within the warp. For these devices, the 256-bit transaction passes through L1 cache† before reaching the threads. The added benefit in this instance is that two clauses fit in the transaction window hence theoretically every second clause checked would result in a cache hit and require no global memory transaction.

3.3.2 Scaling over Multiple GPUs

A significant benefit of this approach is trivial scalability across multi-GPU systems. A host system with multiple devices attached can initialise each of the devices by transferring a copy of the input formula to each, which remains in the global memory of each device until explicitly removed by the host. Subsequently, the host can choose to either distribute partial assignments as they come each on a device in round-robin fashion, or spread the work of checking a single assignment across devices. The former benefits from reduced workload on the host-side, as results need no collation, yet the latter may provide finer control over work balancing as far more threads will be working on the problem at hand with less workload on each.

Scaling in either case is linear as our results show in Section 3.3.6. Performance from each device may vary however, especially since our approach can be used on host systems with different GPU models and CCs. Our implementation can in principle be used on devices of CC ≥ 2.0 yet we have confirmed the effect of our optimisations detailed in Section 3.3.4 for all CCs $\in [5.0, 8.9]$. We note that our implementation

*At the tail end of the computation, some threads will exit earlier than others, as their new local assignment contains more variable assignments than there are unset in the formula. This does not lead to significant divergence.

†L1 cache resides in on the same on-chip memory medium as shared memory.

was designed to target CC 6.2, with some margin for improvement stemming from the release of later CCs, as detailed in Section 4.5.

On distributed systems, this kernel may still be used and will scale linearly, however special consideration has to be taken on the work partitioning and distribution scheme. Whilst a central node can in principle use some message passing scheme (e.g., Message Passing Interface (MPI) [26] or simply distributed Non-Uniform Memory Access (NUMA)) to delegate work to other secondary GPU-equipped nodes, the costs associated with the coordination and management of devices is likely to nullify performance gains in that situation. We envisage that in a distributed environment a portfolio approach (see Section 3.1.1) will be chosen instead, where each of a number of nodes acts as a self-contained unit with a differently tuned search algorithm to the rest and, along with its GPUs, 'competes' to solve the problem. The specifics of this implementation do however depend strongly on the setup of the distributed systems concerned, yet fundamentally a plethora of approaches are enabled due the checker's agnostic (in terms of the overall computation) nature which is leveraged to achieve open scalability.

3.3.3 Thread-Level Integer Encoding of Partial Assignments

Each thread uses a local* partial assignment μ_l specific to itself and has access to the input partial assignment used at present, μ_p. μ_p is constant (i.e., not changing during in-GPU search), and μ_l is a completing assignment (i.e., an assignment to all unset variables in μ_p).

We use a 64-bit bit vector per thread to represent the thread's partial assignment μ_l. Each unset variable in the input partial assignment corresponds to a bit in this vector. The bit corresponding to a variable v is the v^{th} bit, which allows for the truth value of the variable to be extracted using simple bitwise operations. This approach does however require unset variables in the input partial assignment to be ordered sequentially starting at 1†. To enforce this, we have introduced a variable renaming pre-processing step to rename the literals in the formula appropriately. We use the function presented in Listing 3.2 that takes a parameter formula ϕ and a partial assignment μ and renames variables in ϕ and μ in accordance with the kernel's expectations.

More specifically, c is the set of unset variables in μ and the counter i holds the variable last renamed. At first, $i = 0$ since no renaming has taken place, and upon the first iteration of the loop (Lines 5–10), the first unset variable will be renamed to 1 using the function `swapAndUpdate`. This function 'swaps' the variables v and i in the formula

*Local scope to the thread, not in local memory.

†Here 1 is chosen as the natural choice (0) cannot be negative (i.e., -0 is the same as 0), hence a literal comprised of the negation of this variable would not be possible to represent. This offset is compensated for at no penalty.

and partial assignment, Lines 8 and 9 respectively. Upon renaming a variable, v has to be updated to the new set of unset variables and the process repeats until there are no unset variables left to rename.

```
1  fn renameUnset(φ, μ):
2      let i ← 0;
3      RENAME:
4      let c ← unsetVariables(φ, μ);
5      foreach (v ∈ c):
6          if (v > i):
7              i ← i + 1;
8              φ ←swapAndUpdate(φ, v, i);
9              μ ←swapAndUpdate(μ, v, i);
10             goto RENAME;
11         endIf
12     endLoop
13 endFn
```

Listing 3.2: Variable reordering algorithm.

3.3.4 Optimising Formula Satisfaction Testing

Once a thread has computed its current assignment, it checks if the combination of the input partial assignment and the local partial assignment satisfies the formula. To do this, it has to iterate over the clauses of the formula, and for each identify if any of the three literals are satisfied.

Since in our solver the input partial assignment μ_p and the local partial assignment μ_l reside in different memory, checking if a given literal is satisfied requires a decision to be made over which of the two partial assignments holds the truth value for the literal.

Listing 3.3 outlines the process of checking if the literal L is satisfied under the current assignment. An implementation of this check would require nested branches which in turn present threads with multiple opportunities for divergent execution. Flattening such branches is possible by constructing an expression (typically comprised of bit arithmetic and manipulation) that yields the same result without conditional execution. For instance, considering Lines 6-9 in Listing 3.3, the corresponding check can be collapsed to the following bitwise expression:

$$((L \gg 31) \oplus v) \,\&\, 1$$

When this check is reached, v is set to either *TRUE* (i.e., 01) or *FALSE* (i.e., 10) as dictated by either μ_l or μ_p (Lines 2-4). The expression $(L \gg 31)$ operates on the

```
1  fn isLiteralSatisfied(L, μ_l, μ_p):
2      let v ← getValueOfVariable(μ_p, L);
3      if (v = UNSET):
4          v ← getValueOfVariable(μ_l, L);
5      endIf
6      if (isNegated(L)):
7          return ¬ v;
8      else:
9          return v;
10     endIf
11 endFn
```

Listing 3.3: Description of the satisfiability check performed on the thread level.

literal L in 32-bit signed integer form*, and performs a sign-extending right shift that copies the sign bit across all 32 bits, in effect setting the least significant bit to the value of the sign bit. Subsequently, an XOR operation is performed between this value and the value of v, from the result of which the bit of interest (i.e., the least significant bit) is extracted by performing an AND operation with 1. This process calculates the value t based on whether L is negated, and whether v is *TRUE* or *FALSE*. For example, when L is negated and v is *TRUE*, the left operand would be 1 and the right operand would also be 1, which would result in 0 indicating that the literal L is not satisfied under either μ_l or μ_p.

The process of retrieving assignment values for a variable depends on whether the value is stored within μ_l or μ_p however, and crafting the check to account for this whilst eliminating branches is more intricate. To retrieve from μ_p, the correct 32-bit word has to be identified and the relevant 2 bits of this have to be extracted from it which in itself is a trivial process, discussed in Appendix B.2. Retrieving an assignment from μ_l is an even simpler operation, that is, a matter of extracting the relevant bit from the bit vector.

The discrepancy in representation, a necessary evil, complicates matters in terms of assignment checking. A possible incarnation of the complete implementation of Listing 3.3 may be that shown in Listing 3.4. In this listing, `l` holds the truth value of the given literal under the input partial assignment (`a`). If there is no assignment for the literal, the assignment is extracted from the local partial assignment (`alt_bw_assgn`).

*We note that bitwise right shift on negative integers is implementation-defined behaviour. Despite, to the best of our knowledge, there being no official documentation of this behaviour in CUDA, our tests indicate this is a sign-extending right shift operation. Care must be taken however when applying this procedure to other platforms.

Modern compilers and especially those targeting processors with no support for speculative execution such as GPUs* are advanced enough to anatomise such branches and produce equisatisfiable branchless predicates comprised of bitwise operations. Compiler heuristics may deem the cost of such change outweighs gains and preserve the branchful approach, or may be unable to determine an equisatisfiable predicate in cases of complex logic.

```
__device__ int complete_branchful_check(const uint32_t* const a,
    ↪ const int32_t lit, const uint64_t alt_bw_assgn) {
 const int l = get_lit(a, lit);
 if (l == LIT_UNSET) {
  int agn = (alt_bw_assgn >> FAST_32_BIT_ABS(l)) & 1;
  return lit < 0 ? agn == 0 : agn == 1;
 } else {
  return lit < 0 ? l == LIT_FALSE : l == LIT_TRUE;
 }
}
```

Listing 3.4: Naïve branching implementation of device-side literal satisfaction checks under either μ_p or μ_l.

We observed that the aforementioned device function resulted in branchful PTX (and by extension, SASS) code and concluded this is in all likelihood the result of complex branching conditions rather than a cost-benefit analysis. This is further reinforced by compiling the same function specifically for pre-Volta architectures. In those architectures it could be argued that warp divergence is even costlier due to the absence of per-thread program counters. Contrasting the results with compilations for Volta or later architectures, we observed the branchful code was preserved in all cases. Under certain 'unsafe' assumptions, the complexity of a lengthy (perhaps unfavourable) flattened branch can be reduced as explained later. We therefore manually optimised the branches out and produced the equivalent function shown in Listing 3.5.

Compiling this altered function for both pre- and post-Volta we observe no branching instructions in the generated PTX (nor SASS) code. The design of this function is such that no branching instructions are required, however, this is product of both sides of each branch evaluating at all times. From the perspective of performance, the added compute workload is not a concern here as the kernel it belongs to is heavy on memory interaction (i.e., memory-bound) and the additional compute load serves to hide memory latencies behind meaningful computation. However, a side-effect does exist in this instance, as words from μ_p will be read even if the assignment to this literal exists in

*At present NVIDIA GPUs offer no speculative execution pipelines. The nearest feature available to the programmer is the compiler intrinsic `__builtin_expect()` that hints to the compiler which way a branch is more likely to go, but concerns only the generation of code and not its execution.

```
1 __device__ int is_lit_satisfied_complete(const assignment_t*
    ↪ const m_partial, const int32_t lit, const uint64_t
    ↪ m_local) {
2  const int l = get_lit(m_partial, lit);
3  return ((l == LIT_UNSET)
4      & (((lit < 0) & (~((m_local >> (abs(lit) - 1)) & 1)))
5       | ((lit > 0) & (((m_local >> (abs(lit) - 1)) & 1)))))
6     | ((l != LIT_UNSET)
7      & ((l >> 1) ^ (l & 1))
8      & ((l & 1) ^ (lit < 0)));
9  }
```

Listing 3.5: Complete branch-free check determining satisfaction for a given literal under either the tread-local partial assignment μ_l or input partial assigniment μ_p, incorportating Listing B.6 from Appendix B.2.

the, much faster to query, μ_l. This is likely a worthy compromise to make given the impact of warp divergence.

The branchless expression is rather lengthy, but can be divided into two halves split either side of the OR operator on Line 6. The left-hand expression is asserting that an assignment for `lit` in μ_p is not set (`(l == LIT_UNSET)`) *and* one of the below holds:

- `lit` is negated (`(lit < 0)`), *and* its assignment in μ_l is false (`~((m_local >> (abs(lit)- 1))& 1)`) *or*
- `lit` is not negated (`(lit > 0)`), *and* its assignment in μ_l is true (`((m_local >> (lit - 1))& 1)`).

Extracting the truth assignment of `lit` from the thread-local assignment `m_local` is a trivial right-shift operation by `abs(lit)-1` many positions followed by an AND with 1 to isolate the least significant bit of the resulting bit word. When guarded by a branching statement asserting that `l` is indeed `LIT_UNSET`, and a subsequent branch asserting `lit` is indeed negative, an expression such as the aforementioned is safe. In this case however, all operations will be performed even if the necessary assumptions do not hold, which results in undefined behaviour. Specifically, `m_local >> (abs(lit)- 1))` will result in a right-shift beyond the bit length of the type in question (64-bit in this case, if `lit < -64` or `lit > 64`). Whilst the behaviour of the compiler is not defined, in our tests across different versions of NVCC using the PTX assembler optimisation levels `O3`, `O2`, and `O0`, we observed a consistent and direct mapping of the bitwise expressions to respective integer bit manipulation instructions in PTX. These instructions have well defined behaviour [90] for such situations; specifically `shr` and `shl` clamp the shift to the bit size of their respective operand, in this case meaning `shr.u64` will clamp

the right shift to 64 positions. Perhaps a safer approach would be to manually encode this check in inline assembly either in full or specifically for the parts leading to undefined behaviour, however we saw little benefit to this, as it would still be rendered a routine demanding high-maintenance.

Achieving best performance can sometimes require such risks to be taken, yet the overarching lesson in branch flattening is not that such code is inherently risky, rather that gains can be made from it, with careful crafting and analysis of post-compilation code. This does however hinder maintenance efforts in the future which must ensure code remains safe. For instance, further analysing our function in Listing 3.5, we note it can be made fully compliant with C/C++ standards by wrapping the right operand of the right-shifts within the bit length of the type (i.e., `(abs(lit)- 1) % 64`). The modulo operation in this instance does not* transpire to an expensive `rem.s32` instruction in PTX, rather multiple cheaper instructions as shown in Lines 3–7 of Listing 3.6†.

```
1  abs.s32    %r2, %r1;
2  add.s32    %r3, %r2, -1;
3  shr.s32    %r4, %r3, 31;
4  shr.u32    %r5, %r4, 26;
5  add.s32    %r6, %r3, %r5;
6  and.b32    %r7, %r6, -64;
7  sub.s32    %r8, %r3, %r7;
8  shr.u64    %rd2, %rd1, %r8;
9  cvt.u32.u64   %r9, %rd2;
10 and.b32    %r10, %r9, 1;
```

Listing 3.6: PTX output for the operation `(abs(lit)- 1) % 64`.

The compiler output indicates that the operation `(abs(lit)- 1) % 64` can be computed with plain addition and bitwise operations in the manner shown in high-level C code in Listing 3.7. Here, the operation `al>>31` assumes the definition of a right shift on a signed type by the implementation is sign-extending. The variable `a` holds a mask of either 0 or 6 set bits if `al` is positive or negative respectively. Subsequently, the variable `b` holds the result of the operation obtained by adding the earlier created mask (`a`) to `al` (in essence adding 64 to it if it is negative) and retaining the lower 6 bits of the result, which are in turn subtracted from `al`.

```
1  int32_t al = abs(lit)-1;
2  int32_t a = ((uint32_t)(al>>31))>>26;
3  int32_t b = al-((al+a) & -64);
```

Listing 3.7: C code equivalent of the compilers output for `(abs(lit)- 1) % 64`.

*When compiled using the maximum level of PTX optimisations.

†A brief reference of PTX instructions and their intended functionality can be found in Appendix A.1.

The computation in Listing 3.7 assumes (rightly) that `al` may be a negative integer (i.e., when `lit` is zero) hence the need for the computation of the mask `a`. In our case however, it is safe to assume this will never be the case, in line with our formula representation. With this in mind, the operation can be simplified simply to `(abs(lit)- 1) & 63`. We used the intrinsic hint function `__builtin_assume()` to offer this safe assumption to the compiler, alongside another more direct assumption that `abs(lit)> 0` anticipating the, substantially simpler, computation to take place instead. Neither one nor both of these assumptions had any tangible effect to the generated PTX code however, and we instead opted to manually introduce the bitwise AND operation in the expression shown in Listing 3.5.

3.3.5 Heuristic Clause Reordering for Memory Pressure Reduction

In a bid to enhance performance we implemented a sorting heuristic for the input formula that prioritises clauses by their "checking cost". The checking cost is determined by the literals of the clause and their position in the input partial assignment. Considering the representation of the input partial assignment, we hypothesised it may be the case that some clauses contain literals resident in different (non-adjacent) 32-bit words, thus possibly requiring more memory fetches than others. For instance, Figure 3.3 presents the mapping of the literals of a formula to the corresponding "cells" in the input partial assignment. The assignment for each variable is represented as a pair of bits; with pairs grouped in sequences of 16 (2-bit) cells. Some clauses such as the first (left-most) will contain literals that fall within the same 32-bit word in the input partial assignments, and others, such as the second, will span across multiple 32-bit words.

Whilst all clauses will potentially be checked under certain assignments, assignments that lead to a contradiction can be identified whilst checking the, cheap to access, clauses, in effect reducing memory pressure. To establish the cost of access we use the number of 32-bit words that need to be fetched in order to check the clause. Subsequently, clauses are sorted in descending order based on the number of fetches needed.

Using this sorting we observed small gains in performance likely due to the cheap access costs of shared memory of GPUs. Whilst such gains still ultimately contribute to more performant checking, we believe this heuristic will be more effective in other architectures or in future architectures with different memory performance and caching schemes, where the input partial assignment may reside in memory types the like of global memory.

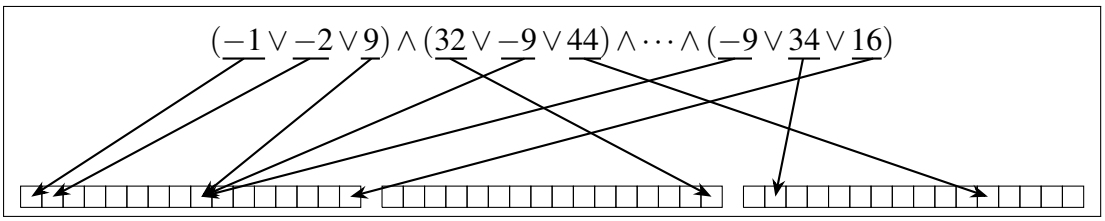


Figure 3.3: Mapping of literals to their corresponding assignments (as 16-block wide arrays where each block is the assignment of a given literal) in the input partial assignment.

3.3.6 Performance Evaluation and Results

To gather quantitative data on the performance of our approach, we used random unsatisfiable 3-CNF instances generated by CNFgen [74] as benchmark inputs. Our data set consists of problems with 200 and 400 clauses, each with 25, 30, 35 and 40 literals. We chose unsatisfiable instances in order to measure performance by fully exploring the search space for each of the problem inputs and gather data that highlights any performance fluctuations. As the focus of our work is the GPU-side solver assistant we did not perform initial solving on these instances on the host-side, rather we let the input partial assignment be blank to ensure that the full 2^n cases are explored for each instance by the GPU.

We profiled our approach on a single NVIDIA RTX 2080Ti (Turing), two NVIDIA RTX 1080ti* (Pascal) and a single NVIDIA RTX 4090 (Ada Lovelace). The GPUs were housed in different host systems with the exception of the two RTX 1080ti devices which reside under the same one. We took 10 measurements of execution time (in milliseconds) per input on each system. Since all inputs are unsatisfiable and a full search-space exploration was performed, the number of assignments tested can be computed. Using this information, we derived the average Checks Per Second (CPS) achieved for each instance by dividing the total number of assignments by the average solving time for that instance in seconds. This performance allows for easier comparison between devices and later, a CPU implementation. Figure 3.4 presents the average CPS for each benchmark suit†.

The results obtained highlight the power potential of GPUs, achieving several billions of assignment checks per second, even in a non-optimal implementation (discussed further in Section 3.3.6.2) which our future work aims to address. CPS counts by the GPUs in question are far from regular and vary sharply between instances, with

*Two sets of performance data were gathered using just one of the two devices, and both devices simultaneously.

†Raw data is shown in Table B.1 of Appendix B.5.1.

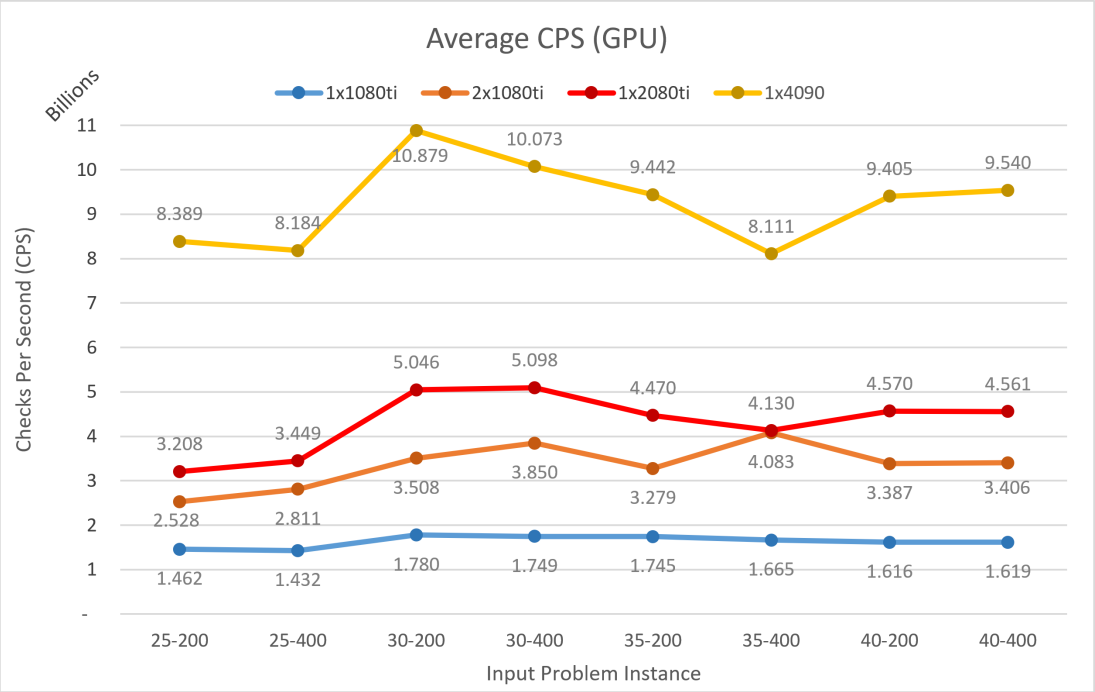


Figure 3.4: Achieved CPS per benchmarked instance per device.

notable example the input `35-400` for which both the 2080ti and 4090 fall below their respective averages whilst the single 1080ti remains as performant and the two 1080ti devices combined perform as well as the 2080ti. Analysis on these instances indicates this is the result of work imbalance leading to a long tail in computation for `35-400`.

Inputs `25-200` and `25-400` have consistently reduced CPS relative to the rest, which is attributable to the relatively small number of assignment permutations ($2^{25} = 33,554,432$). The small amount of work does not aid load balancing and latency hiding, whilst the little time needed for the computation to complete magnifies any timing inaccuracies and startup latencies which in an input this small, affect CPS significantly.

As is to be expected from a brute-force style algorithm such as the one used here, the effects of phase transition and fixation on local minima described in Section 3.1 do not affect the results shown here.

We note that the two 1080ti GPUs generally perform approximately twice as well as the single one as is logical to expect, yet for certain inputs (e.g., `30-400`, `35-400`) their performance is more than double that of the single device. This is likely the result of the characteristics of the two sub-problems when splitting the work between the devices leading to use of some of the available headway in throughput of the devices. Whilst not always performing twice as well as their single counterpart, on average

across all benchmark suits, the two 1080ti GPUs show a linear speedup compared to the one 1080ti, performing 2.04 times as fast. This illustrates the scalability of this approach and is expected to carry on scaling linearly across any number of devices (factoring in setup costs as the number of devices increases).

Our experiments on the 4090 show a significant improvement in performance, averaging $\approx 9.25 \times 10^9$ CPS, which relates directly to the higher core count (see Table 2.2) in this device compared to its predecessors, along with the many important architectural advancements made in that time.

In line with other work [134] for GPU-based implementations it is customary to contrast the performance of a GPU approach against an equivalent single-threaded implementation on the CPU-side. We performed this comparison using a system with one AMD Ryzen 9 3950x processor running at up to 4200MHz. The CPS for each input shown in Figure 3.5 were obtained on the CPU in the same way as the GPUs, with the exception of `35-200`, `35-400` `40-200` and `40-400`. For those inputs the required solving time made it unfeasible to collect sufficiently many samples in a sensible time frame. We therefore collected data by reporting an interval average of CPS at 5 second intervals during checking, and averaging across them.

The choice for a single-threaded implementation on the CPU was to allow full utilisation of a physical core at the boosted clock speeds stated, paired with the expensive nature of threading (when starting and context switching) and likelihood of thermal throttling affecting results. It is noteworthy however, that even extrapolating results to 16 identically performing threads assuming no overheads, still places the CPU implementation below that of the 2080ti. We compare specifically to that device purely because

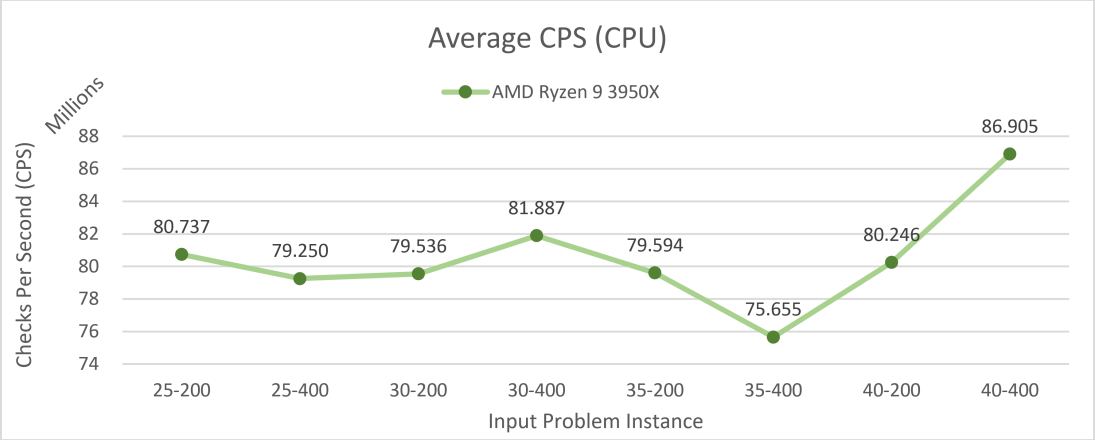


Figure 3.5: Achieved CPS by an equivalent single-threaded implementation running on a Ryzen 9 3950x CPU.

it was released approximately a year before the CPU was* meaning both would have been representations of the state of the art at around the same time. The 2080ti shows a speedup of ≈ 53.64 compared to the single-threaded CPU version, which would still remain positive at ≈ 3.35 in the ideally extrapolated scenario of all 16 CPU cores being utilised and performing uniformly. Like the GPU, on the CPU CPS is not consistent between instances, yet CPS don't follow the same trend between CPU and GPU either. We speculate this is the result of a single threaded implementation not experiencing a tail effect, together with speculative execution for the available branches, neither of which are supported by the GPU.

We highlight that this comparison is only a crude point of reference and should not be used to draw conclusions exclusively. The differences between CPU and GPU hardware are vast, and performing fair comparison between these two types of hardware would require large sets of data collected across many CPUs and GPUs which are not available to us.

3.3.6.1 Impact of Clause Reordering

Testing the real-life impact of clause reordering on the performance metrics we chose can be achieved by contrasting CPS before and after sorting in the method described above. This does not give a concrete picture of the impact of this heuristic however since the natural order of clauses in the input formulae is arbitrary relative to our ordering heuristic.

We therefore opted for a best-to-worse case comparison of results between instances. More specifically, we ordered the clause by memory cost as detailed in Section 3.3.5 and collected data as described earlier in this section using an RTX 2080ti GPU and a kernel of 136 blocks of 1024 threads each. We subsequently applied the inverse of our sorting function (i.e., the order of clauses is from worst to best) and contrasted the two.

Through this comparison we observed an average increase of 20% in CPS when clauses are ordered versus not. As seen by the data points for each input shown in Figure 3.6† however, CPS increase is not stable between instances, with some falling below the CPS of their inversely sorted counterpart. For reasons described earlier the instances `25-200` and `25-400` are difficult to draw conclusions from as minute differences in their (already short) runtime greatly impact their CPS. Besides those two however, we observe a slight drop in CPS when clauses are ordered for `35-200` versus their natural order or inverse order. The difference between the three is relatively small and falls within the margin of error. It is nevertheless important to highlight the

*AMD Ryzen 9 3950x was released in Nov. 2019 whilst the 2080ti was released in Sept. 2018.

†Raw data is shown in Appendix B.5.3.

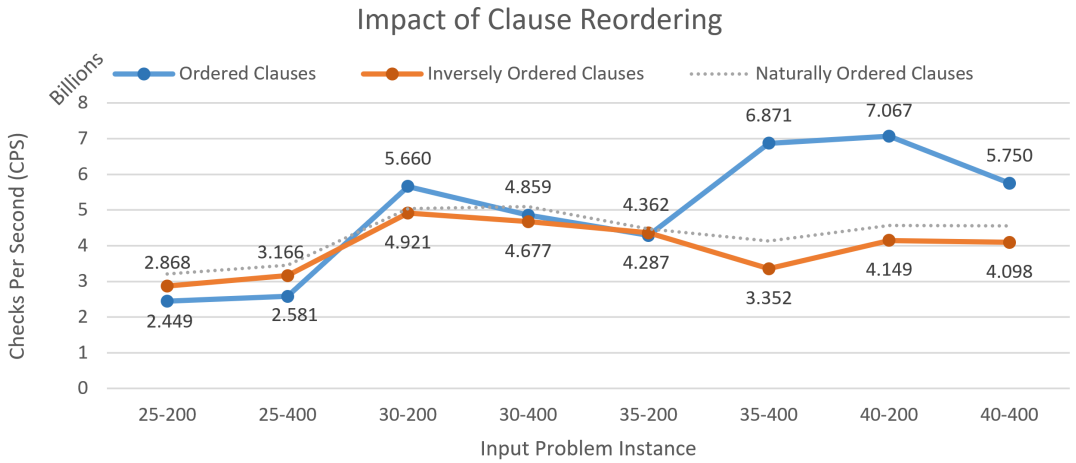


Figure 3.6: Achieved CPS on one RTX 2080ti when clauses are ordered normally and when inversely ordered.

absence of improvement in CPS for this instance, which we attribute to the characteristics of the input being such that the majority of clauses contain literals for which the assignment is stored in a different bit word in the input partial assignment. This can also explain the inconsistent increase in CPS for other instances (e.g., `40-200` and `40-400`).

As is typically the case with heuristics, we expect this ordering to lead to performance gains on average with variations in its effectiveness over different inputs of different characteristics. It is likely to be the case that this heuristic is effective for larger inputs (expressed in terms of variable and clause counts) as there is more room for variations in clause composition, yet we do not expect this heuristic to result in notable performance loss under any circumstances.

3.3.6.2 Analysis of Workload and Balancing

The workload at hand is interesting from the prospective of work balancing, as discussed earlier in Section 3.3.6. The kernel implementation discussed in previous sections, although performant, still has room for improvement. In particular, this is a memory-bound kernel meaning that heavy reliance on memory interactions does at times limit the performance of the kernel as a whole. Likewise, work re-distribution mechanisms are not employed resulting in some warps computing for significantly longer than others, or threads within the warps leading to warp stalls.

Taking for instance the inputs `35-200` and `35-400`, with the exception of the bridged 1080ti GPUs, other devices show consistently poorer performance for the latter relative

Figure 3.7: Composition of warp and block activity throughout the computation of `35-200`.

to the former. This is in all likelihood to do with the inputs themselves, yet it is worth investigating further as the core component of the inputs (i.e., number of unset variables) is the same, leading us to conclude that the random clauses are to blame. Looking at the performance counter data timeline on the 4090 (i.e., the device with the highest drop in CPS) for both inputs `35-200` and `35-400`, seen in Figure 3.7 and Figure 3.8 respectively, we observe a longer tail in computation for the latter than the former. It is also noteworthy that each block launched appears to consume more time for `35-400`, leading to a more coarse distribution of block launches.

The tail effect in this computation is caused when few active threads remain in each warp due to unbalanced work between them and their counterparts. On the hardware level, each SM is partitioned into four processing blocks, each with its own scheduler*, that is responsible for a subset of warps arising from the block(s) assigned to the SM. On each cycle, each scheduler selects an eligible (i.e., non-stalled) warp to issue instructions, if one exists. The warp is selected as a whole irrespective of how many threads within it are in a position to issue an instruction. In an ideal scenario, all warps are ready to issue the same instruction, and will exit (i.e., complete their work) at the same time. Yet, in unbalanced workloads such as this, some threads may have more work than others and exit later. Figure 3.9 illustrates how this imbalance leads to progressively fewer continuing threads in the block, scattered among different warps. We note that blocks and warps in this figure are depicted as rectangular structures for

Figure 3.8: Composition of warp and block activity throughout the computation of `35-400`.

*This is the case for current architectures, but may differ at the time of reading.

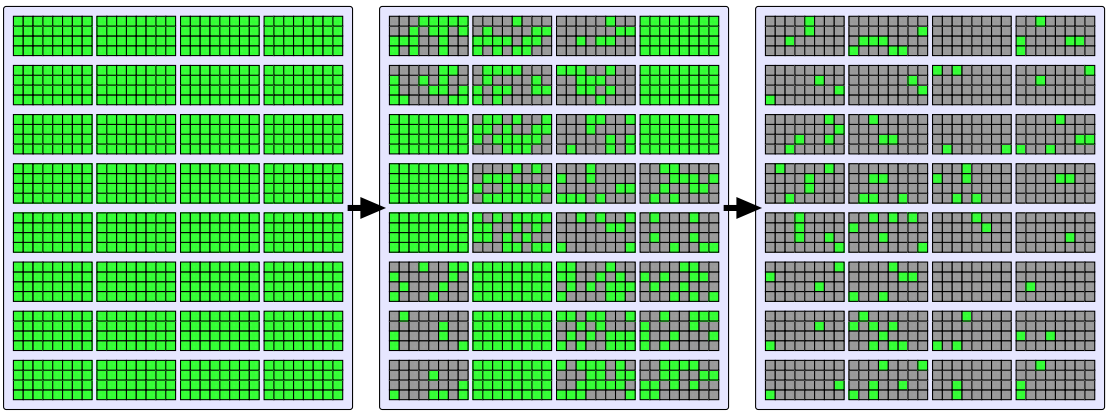


Figure 3.9: Example progression of active threads per warp during computation for an imbalanced workload, irrespective of geometry.

typographical purposes as their geometry does not influence this effect.

Performance analysis for this workload shows an average of 21.69 threads active per warp through the computation of `35-400`, which is approximately equal to that of the other inputs. This, paired with the composition of the computation (specifically the active warps per cycle) shown in Figure 3.8 indicates the tail in this instance results from a dropout in active threads per warp near the end of the computation.

These factors together lead to the conclusion that for the input `35-400`, more clauses had to be checked on average by each thread before falsifying its assignments. This is further evidenced by the increase in cumulative memory transfers between the two inputs with the computation for `35-400` reading 41.19GB from global memory, whilst its counterpart read 37.62GB*.

With these two inputs, like the rest, we observe a stable and consistent use of the SM (seen as the active cycles in Figures 3.7 and 3.8) and overall achieve good cache use with a 52%–58% L2 cache hit rate in line with expectations (i.e., every second clause requested during checking likely resulting in a hit).

This workload makes heavy use of bit manipulation instructions on integer datatypes, therefore it's natural to expect the ALU to be heavily utilised. Looking at the pipe utilisation for this workload in Figure 3.10 however, it is interesting that high (w.r.t. the workload at hand) usage of the FMA unit is also reported. This unit (particularly the so called `FMAHeavy` pipeline that is part of it) is responsible for handling some integer instructions including the SASS instruction `IMAD` which is used in critical places of our kernel. This is beneficial here, as high pipe utilisation on the ALU means that warp

*These figures do not account for cached reads.

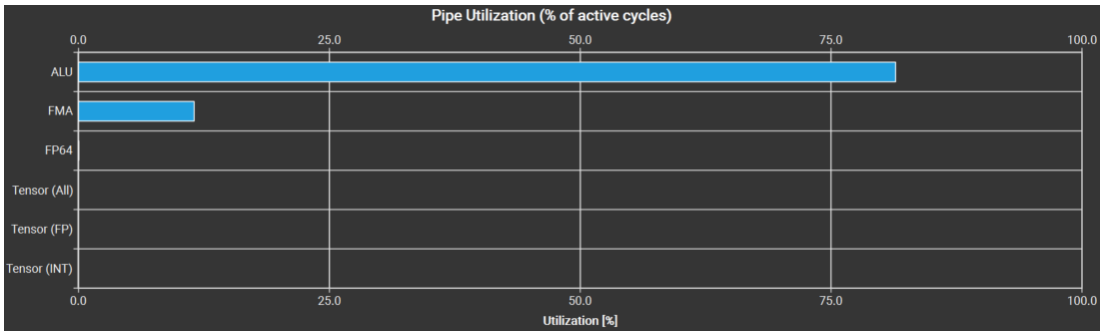


Figure 3.10: Pipe utilisation for the SAT checking kernel.

schedulers may be unable to schedule some warps due to the respective pipe being over-subscribed. Using the FMA unit where possible eases pressure on the ALU. This does appear to be the case with our kernel, with an average of up to 3* warps per cycle, stalled due to mathematics pipeline (i.e., ALU) over-subscription, which would have likely been worse had the ALU also had to handle the work the FMA unit is doing.

A remedy to this problem is adjusting the amount of warps available such that it is more likely some warp will be ready to perform an instruction in a different pipe when one is over-subscribed, yet given the degree of over-subscription and heavy reliance on ALUs we do not think this would be beneficial in practice. An alternative option with potential however is to adjust the workload to make use of other pipelines where possible. This is perhaps more intricate but in this case likely a sensible option. We discuss the potential ways in achieving this throughout Sections 3.5.1 to 3.5.3.

3.3.6.3 Evaluation of Multi-GPU Scalability

As discussed earlier in Section 3.3.2, we anticipate our approach to scale linearly w.r.t. the number of GPUs involved. Our results presented thus far in the parent section support our theory, particularly looking at data gathered on one 1080ti versus two 1080ti GPUs. To further reinforce this, we have gathered data for each input instance being checked on one, two, four, and eight A100 GPUs in the HPC GPU cluster of Swansea University [12]. Ten measurements of execution time were taken for each input as part of a job submitted to the cluster in a manner similar to that discussed in Section 4.4. We note that the host systems used to obtain results in Section 3.3.6 were all Windows systems† whereas in this instance the cluster nodes are Linux based‡. We did not expect nor observe any influence on data gathered as a result of this difference,

*Of four resident per SM, each in their respective partition.
†Using the same version of the MSVC compiler.
‡Using the GCC compiler.

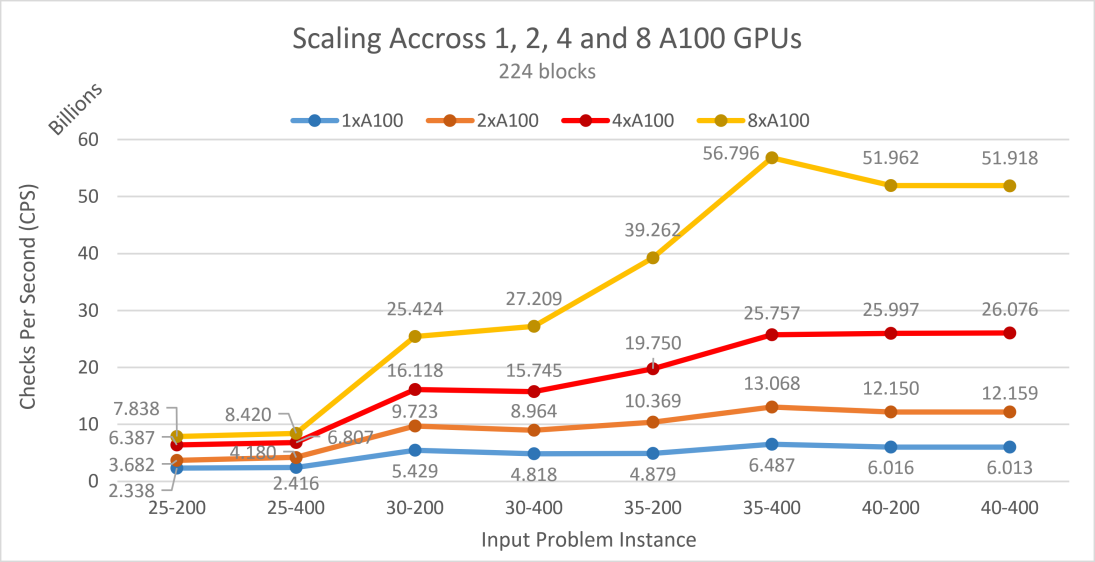


Figure 3.11: CPS achieved by one, two, four, and eight A100 GPUs for a grid comprised of 224 blocks.

as minimal processing takes place on the host system, which is discounted from GPU execution time measurements.

Figure 3.11 graphically* presents the data gathered. Focusing on the last instance (i.e., `40-400`) we observe a speedup consistent with the number of GPUs involved. More specifically, treating the data from one A100 as our baseline, two A100s result in a speedup of ≈ 2, four A100's result in a speedup of ≈ 4, and so fourth. This is the case for the majority of instances with a few exceptions, such as input `35-400` where the eight A100's offer a disproportionately high speedup of ≈ 8.75 (expected ≈ 8) from the baseline. Like before, we attribute this to the characteristics of the input and their effect on work balancing, discussed in more detail in Section 3.3.6.2.

It is noteworthy that the smaller instances (i.e., `25-200` and `25-400`) result in a very short kernel runtime and highlight the effect of setup costs and initial latencies (which would have otherwise been hidden behind the computational workload, as is the case for larger inputs). The case is largely the same for `30-200` where on one hand the speedup between the baseline and two A100s is ≈ 2, as expected, whereas the speedup achieved using four and eight A100s relative to the baseline is ≈ 2.97 and ≈ 4.68 respectively. A similar situation is observed for `30-400` albeit with a smaller speedup deviation from the expected. This happens for the same reason as the two smallest inputs, where the computational workload is not significant enough to adequately hide

*Raw data available in Table B.2 of Appendix B.5.2.

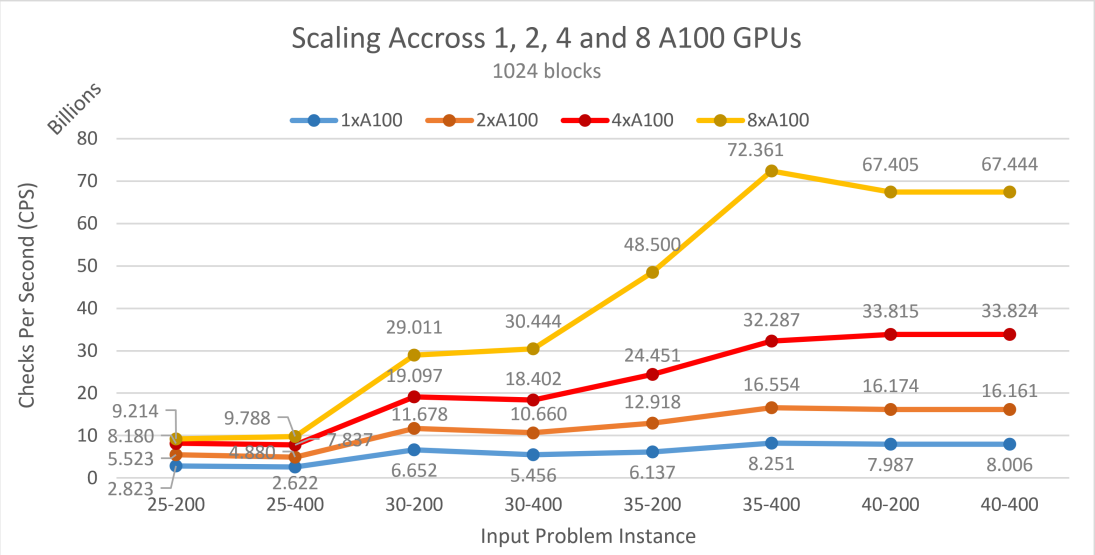


Figure 3.12: CPS achieved by one, two, four, and eight A100 GPUs for a grid comprised of 1024 blocks.

setup latencies, and the actual computation required by each device involved reduces even further as the devices increase, highlighting these latencies even more. For input `35-200` or larger, the speedup is as expected.

The effect of these overheads is exacerbated when the number of blocks is increased to some more favourable (for the A100 GPU) value, namely, 1024. Figure 3.12 shows the CPS achieved in this instance*. The increase in block count has resulted in consistently higher CPS to those in Figure 3.11, however the reduction in kernel execution time responsible for the increase in CPS, also magnifies the latencies, which in the case of eight A100's results in the speedup for the first four inputs (`25-200`, `25-400`, `30-200`, `30-400`) falling short of what is to be expected.

3.4 Reflections on the Applicability of Current GPU Architectures to SAT

Through our work, and the work of others, it is evident that full on-GPU SAT solvers are not feasible at the present state of GPU hardware. GPUs remain a promising candidate for the time being and based on their evolutionary trajectory, we anticipate that a full on-GPU solver will be feasible in the future. At the present time however, there

*Raw data available in Table B.3.

is potential in harnessing their current power to support CPU-side solvers, living up to their nature as coprocessors.

The 'traditional' SAT solvers are typically designed with sequential execution in mind, relying heavily on branch-and-bound techniques which exhibit irregular (for the GPU) memory access patterns and requirements. We have hit this roadblock ourselves with our work on a full GPU-based solver where a choice had to be made between search depth limitations when using fast on-chip memory, or expensive memory access when relying on the ample global memory. Such solvers could be parallelised in the GPU using techniques such as multi-kernel decomposition [83] which, for problems exhibiting state explosion, we believe may offer sensible speedups.

More promise is shown by approaches offloading part of the solving effort to the GPU in a heterogeneous solver system. The main reason for this, relates to the isolated tasks being offloaded having a viable mapping to the hardware requirements. In our case this takes the form of easily enumerable local assignments alongside cached reads in a fast exhaustive search implemented with no more branching than necessary.

A large number of factors under the surface must necessarily be considered on the implementation level to achieve good performance, far greater than modern CPUs demand. Factors such as pipe utilisation and over-subscription, alternative instruction use, work balancing within cooperative structures and SM partitions, cache-appropriate data representation, and so fourth, must be balanced but may not necessarily be optimally tuned all at once. To strike a good balance between them all, meticulous profiling and optimisation is required even for the simpler tasks.

The future is not bleak however; our observations suggest it will likely not be a long time before GPUs can be considered in the field. In fact, substantial changes have recently taken place in the design of GPU, that are already relaxing restrictions. These relaxations, subject to more investigation (see Section 3.5), may already be sufficient for some approaches to be implemented efficiently.

Beyond the decision problem however, the field of SAT is vast and other formulations of the problem exists for which the current state of the art in GPUs is perhaps more fitting. For instance, Model Counting (commonly referred to as #SAT or 'sharp' SAT) and particularly *exact* Model Counting [48] has the goal of counting the exact number of solutions to an input formula. This counting variant is typically approached with DPLL-based algorithms that do not halt upon identifying suitable assignments for a (sub)set of variables, but instead continue branching until available paths are exhausted. Whilst not necessarily performing exhaustive search over all possible search paths (i.e., assignments), substantially more work and in particular, assignment checks, are required versus the decision problem.

As our work here and later in Chapter 4 serves to show, GPU hardware can be used

effectively for the 'strenuous', repetitive tasks with straight-forward parallelisation strategies that involve no global communication. We stipulate that techniques and optimisations such as those shown in this chapter may suit variants such as #SAT due to their prolonged computation time (i.e., partial if not complete enumeration) and problem characteristics which already eliminate hard-to-map (on the GPU) heuristics such as look-aheads and clause learning.

ALL-SAT is another variant of SAT, that is arguably more 'difficult'. This variant requires not the count but the full set of all solutions to be produced. Unlike #SAT where solutions on some branches can be counted without necessarily being visited, room for such shortcuts is very limited in ALL-SAT [137]. This is because for this variant each satisfying assignment has to explicitly be computed and output. This variant is typically approached using the same prevalent backtracking techniques as others (i.e., DPLL derivatives) with a restriction on the search space limiting heuristics. For the same reasons as #SAT, the computational environment of GPUs is well suited for this type of computation with appropriate adjustments as presented in our work.

In all cases, a massively parallel environment such as that of the GPU together with the physical and computational restrictions associated with it, does require a shift in the focus of algorithmic design. Some heuristic optimisations may result in more harm than gain in practical performance even when mapped to this environment in the best feasible manner. Likewise, design of techniques and tooling for specific flavours of a problem variant may also be nessesary to utilise GPUs as they evolve. To some outside the domain of graphics processor optimisation, morphing algorithms around -what seems- a specific implementation on strict hardware, may appear a counter-intuitive move and is often met with (fierce on occasion) resistance, yet we firmly support this is necessary. Although not obvious at first, we wish to underline that non-traditional algorithmic reasoning is and has been successfully applied in this field, and it takes a quick stare over the fence into other domains such as computer graphics to realise that core algorithms have either undergone a transformation in this exact manner, or more commonly, designed since their infancy to meet the massively parallel model used by GPUs.

3.5 Future and Ongoing Work

A number of areas of exploration are still ongoing or pending following on from our current work, mainly revolving around the large and likely beneficial changes made to hardware since our current work was conducted. Recently obtained access to a device implementing the Ada Lovelace microarchitecture (CC 8.9) and the upcoming introduction of a node of eight devices implementing the latest, Hopper microarchitecture (CC 9.0) in the AccelerateAI supercomputer [12], for which we are receiving early

access.

The intriguing evolution in hardware and associated architectural changes place spotlight on our earlier work, particularly on the full in-GPU 3-SAT solver which may now be efficiently implementable with room for more optimisations to our existing work leading to general lessons for kernels of this nature. More specifically we wish to explore the following areas on these devices:

- The effects of increased shared memory capacity to the full in-GPU SAT solver. The 227KiB of available shared memory could be used to store $\frac{\frac{232448}{1024}}{4} = 56$ frames per block.
- The feasibility of an off-loading heuristic for full in-GPU SAT whereby frames from shared memory are off-loaded into global memory and retrieved as and when necessary. This has been impractical previously, however we hypothesise that the increase in shared memory capacity together with the high ALU utilisation of this kernel is sufficient to hide the infrequent memory transfer costs of such a scheme.
- The effect of under-subscribed SMs in both approaches we discussed. Given the integer-heavy nature of our kernel we have strong suspicions that a single block per SM makes good enough utilisation of resources that increasing this number will offer no tangible benefit to performance. This will give insight into the memory/compute trade-off of emerging GPU architectures as we believe the trend of increased on-chip memory will continue based on historical data.

Beyond exploring the latest hardware we will consider an alternative approach to the SAT assistant using fast Matrix Multiply and Accumulate (`mma`) operations on binary matrices, using Tensor Cores, as detailed in Section 3.5.3. We also wish to examine the impact of an alternative clause representation to the two kernels explained in Section 3.5.1 which may be beneficial when used with new intrinsic functions, ultimately offering good performance. Finally, we will examine an alternative assignment checking scheme described in Section 3.5.2 on the warp level to balance work better with warp-level granularity.

Beyond improvements to GPU-side kernels, we will implement a DPLL-based CPU-side solver that integrates with our SAT assistant kernel to understand the impact of memory transfers to the overall computation and by extension a memory transfer scheme involving buffering either end to reduce such transfers and to identify a good DPLL cutoff point before brute force search is initiated asynchronously.

3.5.1 Revisit Clause Representation and Caching

In both the full in-GPU and hybrid SAT kernels we opted for an explicit clause representation where each literal is a 32-bit signed integer. Given formulae are typically large and would necessarily reside in global memory, paired with the fact our kernels would have to unpack any alternative compressed representation that remains well aligned, (i.e., bit encoded literals) drove us away from such implementations.

In more recent hardware however such as devices with CC 9.0, shared memory (and by extension, L1 cache) has grown significantly to nearly four times that of devices at the time our work was conducted. With relatively low shared memory pressure from our SAT assistant kernel, the remainder can be used as block-level cache holding significant portions of the formula under test and strongly incentivises a more compact representation even at the cost of unpacking.

More specifically, we will represent each clause as a densely packed bit word of $2 + 10 + 10 + 10$ bits. The 30 LSBs are partitioned equally to represent three variables as unsigned integers (limiting the maximum number of variables in the formula to 1023*). Since each 10-bit portion should correspond to a literal, the two MSBs will represent the number of negated variables in this clause (ranging from zero, i.e., 00 to three, i.e., 11), and those that are negated will be adjacent to each other following the two MSBs.

For instance, the clause $(1 \vee -12 \vee -3)$ will have:

- The two MSBs set to 10 indicating there are two negated variables,
- followed by ten bits representing the variable 12: 0000001100,
- followed by ten bits representing the variable 3: 0000000011,
- followed by ten bits for variable 1: 0000000001.

This packed representation makes full use of the available 32 bits, whilst being relatively simple to unpack.

The benefit of such representation is that the expression (now comprised of a sequence of 32-bit bit words) remains naturally aligned and fits in a single 32-bit register. More crucially however, now a larger portion of the expression can be read with a single memory transaction (i.e., 8 literals in a 256 bit window) reducing global memory load and increasing cache hits instead, with the added benefit of multiple warps hitting the same cached data after one has spent the time loading.

*Perhaps even 1024 if offsetting is implemented however the cost of such operation will be significantly heavier versus the benefit it offers.

Furthermore, intrinsic SIMD functions such as `__vsubus2` which performs per-byte unsigned, saturated subtraction can be utilised to arithmetically check assignments against literals.

3.5.2 Examine Lazy Collective Assignment Checking Schemes on the Warp Level

As explained in Section 3.3.6.2, threads in our SAT assistant kernel will at times diverge as they need to check more clauses than their peers in the same warp. A warp-level barrier does on one hand re-converge threads before performing the next set of common operations and would be sufficient if data-dependent divergence was a rarity, however in our case time is spent with threads either stalled waiting for their peers in the warp to catch up, or having finished and the warp being kept 'alive' by a few remaining threads with work left to do.

To reduce the idle time of threads and minimise divergence we will target assignment checking and introduce a warp ballot operation in which threads vote whether or not to continue checking their respective assignments. When some calibrated fixed point number of threads vote to stop (i.e., have completed their checks), those that voted against it will 'stash' their current assignment and checks (i.e., lazy checking) and collectively move on to the next assignment. Stashed assignment checks will be revisited in a coordinated manner later on, and can be done simply (i.e., each thread revisits its stash) or globally (threads treat their buffers as part of a larger buffer and exchange between them using warp shuffling primitives). The former is simpler and requires less computation to achieve, but can result in divergence. The latter however, will result in the least divergence albeit costing more in terms of coordination and work sharing in a warp-synchronous manner. We will profile both schemes to identify the most suitable.

Each thread will have a pre-allocated shared memory delta buffer structure (stash buffer) for stashing its assignments. The stash buffer structure can either be empty, or store one or more stash frames. When non-empty, this data structure will store the first thread's local assignment (i.e., 64-bit bit vector) in full, along with a 32-bit unsigned integer representing the index of the next clause to be checked against this assignment. Subsequent entries in the stash buffer data structure will comprise of 4 byte long structures (deltas) comprised of a signed and an unsigned 16-bit integer. The signed 16-bit integer denotes the change in clause position of the current entry from the base entry, and the unsigned integer represents the change in assignment of the current entry versus the base entry.

For instance, the structure shown in Figure 3.13 stores four stashed checks. The first stashed check S_0 (leftmost blue node) is a full node storing the full assignment at the

$A_b = 01\ldots10110001_2$ $I_b = 49$	$\Delta A = 31$ $\Delta I = 5$	$\Delta A = 52$ $\Delta I = -23$	$\Delta A = 9855$ $\Delta I = 0$

Figure 3.13: Example stash delta buffer storing the base stash (blue node) followed by three delta stash nodes (orange nodes).

time of stashing and the index of the clause in the input formula to resume checking from. The following node (orange, second from the left) represents another stash entry S_1, the assignment for which is the 31^{st} increment of the assignment A_b in S_0, and the index of clause to continue checking from is derived by incrementing I_b by 5. The process is identical for subsequent nodes, which also store change in reference to the base node S_0. For any given delta node, the change in assignment value ΔA can trivially be combined with the base assignment A_b of S_0 to form the current assignment as $A_c = A_b + \Delta A$.

This structure will be used for its comparatively low memory footprint per thread (12 bytes for the base node plus 4 bytes per subsequent delta node). Assuming 64KiB of shared memory is available to a full block*, each thread can have $\frac{(65536/1024)-12}{4} + 1 = 14$ stashed checks, which we estimate are more than enough for work balancing purposes.

Aside from a compact representation, this structure can naturally be aligned without padding with the base node's 64-bit bit vector directly followed by the 32-bit index stored contiguously. Likewise, delta nodes are 32 bits wide and naturally remain 4 byte aligned. Besides, all a thread needs is a pointer to the head of this structure (likely residing in register space). With this approach however, limits must be taken into account to ensure correctness. Threads will not be able to represent a change greater than the type limits, hence no delta node can capture the $65,537^{\text{th}}$ advancement of the base assignment, or an index offset $o \notin [-32768, 32767]$, which in this context are relatively generous limits. A thread reaching one of these limits will immediately stash the state before exceeding the limit, and await the ballot for a stash clean operation.

Beyond being balloted on whether to stash or continue, threads will also hold a ballot to determine if a stash clean is required. If any thread votes positively in this ballot, all threads will begin working through their stashes, in one of the ways described earlier. A thread will vote positively if at least one of the following holds; the thread's stash buffer has reached capacity or the last check resulted in a thread reaching the limits of a delta frame.

*This calculation does not take into consideration the partial assignment residing in shared memory. It is however based on an older device with CC 6.2, assuming a full block per SM.

In terms of the global stash conquering effort, coordinating it between threads presents several difficulties, surrounding efficient work distribution. Our current effort is focused on establishing fast means of a thread with excess work* delegating some to a thread with less work, in an efficient and least-branchful manner as possible. Our current approach is for threads to first establish the total number C of work items to be tackled. Each thread then knows the average number of work items per thread in the warp as $A = \left\lfloor \frac{C}{32} \right\rfloor$† and can thus know if it is over- or under-subscribed, meaning it has K work items where $K \geq A$ and $K < A$ respectively.

Under-subscribed threads want to exchange no work for some work, whereas their over-subscribed counterparts want to exchange some work for no work. Threads communicate their status to the rest via a ballot. The greatest difficulty, is establishing an efficient, non-branching stable matching‡ without the need of pairwise communication between threads. In other words, a scheme where each thread t_i is matched (bidirectionally) with a thread t_k such that $i \neq k$, and there exists no thread $t_{j \notin \{i,k\}}$ which matches t_i or t_k. The difficulty here stems from the fact each thread must naturally come to this decision on its own, and any two matching threads must do so as a pair, selecting one another in a deterministic manner.

The current warp-wide scheme we employ is very simple, and has threads vote on whether or not they are over- or under- subscribed (vote 1 or 0 respectively). The ballot results are known to all threads as a 32-bit bit vector b in which a bit at index i is set iff t_i is over-subscribed. Each over-subscribed thread t_o then selects a thread t_u such that $o > u \wedge \neg\exists(x \in \mathbb{N} : o > x \wedge x > u)$ based on b. Respectively, under-subscribed threads pair themselves with the over-subscribed counterpart by applying this matching criterion in reverse. This approach views matching through the eyes of over-subscribed threads and leaves under-subscribed ones with no adjacent over-subscribed counterpart 'unmatched'.

Upon completion of this matching, matched threads use warp shuffling primitives (particularly `__shfl_sync`) to swap work between them in the manner described above, and then all threads proceed to tackle a work item either the one they have been given (if any), or one from their stash (if any). This round repeats until no thread has work left (determined once again through a ballot).

Whilst a first step, this approach is far from ideal; in a pathological case where threads $t_{i \in [16,31]}$ are over-subscribed whereas the rest sit idle, this mechanism is ineffective with the worse case being that where t_{31} is the only over-subscribed thread. Likewise, in

*In this context, referring to stashed assignment checks.

†In all CUDA architectures to date warps comprise of 32 threads. It is possible this number may change in the future, or reasoning may need to be done with sub-warp granularity.

‡For clarity, this is not an instance of the stable marriage problem, as each thread acts as a unit and there are no distinct 'banks' in need of matching as such.

cases of a sequence of under-subscribed threads adjoined to an over-subscribed one, all but the first will remain under-subscribed (i.e., under-utilised) throughout. This brings us on to a further complication resultant from the capabilities of warp shuffling primitives, which is that only one piece of data can be swapped between a pair of threads at once. In operations such as parallel reductions (outlined briefly in Section 4.3) with strictly defined order and step sequences this is ideal, yet in this application it means multiple shuffles (i.e., synchronisation and exchange points) will have to take place to achieve this effect. We consider the matching of threads an open problem which we will address in due course, even making the work balancing exchange step distinct, followed by the tackling of (now balanced stashes).

Nevertheless, even a far from perfect approach such as this is likely to improve work balancing in the general case. Exchange of work items will therefore take place at some point. When a thread delegates a check to another within the warp, it has to transfer its base assignment (A_b) and base index (I_b), along with the chosen delta frame (i.e., delta assignment ΔA and delta index ΔI). This amounts to a hefty 16 bytes of memory which is not thought of as a lot in modern day computing, but is not natively supported by warp shuffling primitives as a single operation. To counter this, threads instead communicate a pointer (i.e., 32-bit address) to the delta frame they want to surrender, which may be a null pointer if the thread does not wish to surrender any work. The receiving thread, knowing the start address of the shared memory pool, the per-thread delta buffer length (stored contiguously) and the pointer its given, can infer the position of the base frame for the delta frame, and fetch both. This necessitates shared memory reads which are inevitably performed in an unorderly manner, and further complicates efforts to reduce bank conflicts. We have reasons to suspect however that the impact of such conflicts will not be sever enough to overshadow performance gains through this work balancing.

3.5.3 Experiment with Matrix Representation of SAT and `mma.m8n8k128` Instructions for Tensor Core Utilisation

Existing work [33] presents a transformation of 3-CNF formulae into a matrix representation. A $2V \times C$ boolean matrix is used where V is the number of variables and C the number of clauses in the input formula, where each clause corresponds to a row of the matrix, and each column corresponds to a literal (for all possible literals, hence $2V$). For each column, the row of the matrix has 1 under the literals that would individually satisfy this clause, and 0 for the rest. The input formula matrix is then multiplied by a boolean matrix of instantiation sets having assignments to each possible literal on the rows. If the resulting boolean matrix has a column with no zeros, then the input formula is satisfiable.

Whilst, as the authors state, this is not a viable solution for generic 3-SAT (or even

simple inputs with few variables) as the matrix size and by extension memory requirements grow exponentially, we wish to explore its applicability in smaller scale, specifically for the SAT assistant. The intention here is to examine if, instead of exhaustive enumeration, a thread can use this representation for a viable subset of unset variables, and continue enumerating the remaining.

The main motivation is the existence of so called Tensor Cores in recent architectures, specifically designed for performant Matrix Multiply and Accumulate (MMA) operations, used primarily in AI applications for training models. Use of these cores is not yet fully and easily exposed on the programming interface since their support varies heavily by Compute Capability, however specific NVIDIA libraries expose this functionality for AI applications. PTX instructions do exist and can be used directly. We wish to focus on the Ampere microarchitecture for initial testing which is documented [91] as supporting $4,992$ TOPS* at peak clock speed on binary matrices. This architecture permits (maximally) an 8×8 binary matrix input to be multiplied with a binary matrix of up to 128 rows (hence the name of the respective PTX instruction: `mma.m8n8k128`).

Our experiments will examine the feasibility of a tiled matrix multiplication operation on tensor cores, to multiply two larger matrices. We will also gain insight on the potential performance gains of such approach versus the current exhaustive enumeration, and find a suitable 'cut-off' point (i.e., number of literals) for which this technique can be used efficiently and feasibly (in terms of memory) during enumeration which would then be focused around any remaining variables.

Whilst the authors of [33] briefly warn of the performance bottlenecks that limit the feasibility of their approach, CUDA-enabled devices were at their infancy at the time of their work, with the latest CC being 1.3 (Tesla microarchitecture). The far inferior architecture (w.r.t. current ones) had substantially stricter memory constrains and no special support for matrix multiplication operations and limited arithmetic capabilities unlike current state of the art devices. We are confident the current state of the art has evolved sufficiently to accommodate the approach described in this section.

*Tera-Operations Per Second, typically referring to integer operations (IOPS) as opposed to Tera-FLOPS which concerns Floating Point Operations.

Chapter 4

GPU-Based N-Queens Solver

Contents

The core principles of algorithms used in solving the Boolean Satisfiability problem that we discussed in Chapter 3, have applications stretching far beyond that problem itself. In this chapter we present our work on the N-Queens counting problem.

Looking at the SAT decision problem and the N-Queens counting problem it may be difficult to picture how one can be used to solve the other and especially when that is to use a decision problem to solve a counting one. The link between the two lies in the algorithmic techniques used and not so much a reduction from one to the other. As previously discussed, the DPLL procedure often used to solve SAT features a propagation step whereby the assignments of lone variables in unit clauses are inferred. This procedure can be used in the enumeration of N-Queens solutions to help the search exclude fruitless paths dynamically.

In this chapter, we present the `DoubleSweep-Light` procedure to enumerate N-Queens solutions, which is moulded around the computational environment of the GPU. We provide two implementations of `DoubleSweep-Light` tailored to devices of different capabilities. Through these (compute-bound) kernels we explore optimisation techniques both high- and low-level optimisations, aimed primarily at reducing and redistributing the computational workload. We present results showing the ef-

fectiveness of our techniques and the open scalability of our approach. Lastly, we conclude with future directions arising from the work presented in this chapter, as well as work that is presently underway.

4.1 Approaches to N-Queens Solving

An interesting observation made in Section 2.1.2 is that the latest and more demanding instance solutions for N-Queens were discovered using highly parallel approaches on distributed system configurations [69, 106, 112, 111]. Notably, the two most recent examples (i.e., $N \in \{26, 27\}$) were both discovered using thousands of workers on FPGA hardware [112, 111]. GPU hardware is a prime example of a widely available massively parallel coprocessor, nowadays found in mid- to high-end consumer systems as a standard, with a strong drive for their broader use in workloads beyond the graphics domain by industry and academic circles alike.

A review of the numerous examples [142, 107, 129, 126, 87, 37] of N-Queens solvers designed for GPUs reveals a common theme: multi-factor optimisation is paramount to any successful implementation on this hardware. The factors to consider start from the algorithmic level and stretch through the implementation, down to manual instruction-level optimisation.

Our journey in optimisation and the development of a fast, fully GPU-based solver for N-Queens begins with `DoubleSweep`, a powerful backtracking algorithm (Section 4.1.2) inspired by Somers' algorithm (Section 4.1.1). An initial attempt at mapping `DoubleSweep` to the GPU environment brought us to make some adaptations that led to `DoubleSweep-Light` being developed (Section 4.2.2), to better facilitate use on GPUs. The journey subsequently takes us through the development of a performant implementation (Section 4.3) and numerous optimisations (Section 4.3.6), finally arriving at a solver that outperforms previous work form literature (Section 4.4) after learning valuable lessons on optimisation.

4.1.1 Somers' Algorithm

In the early 2000's Jeff Somers developed a backtracking search algorithm [119] for the N-Queens problem with good results for instance solutions up to and including $N = 21$. An efficient iterative implementation of the algorithm meant that the hardware of the time, which differed significantly to CPUs as they are known today, was capable of finding instance solutions for such values of N in a sensible time frame. The algorithm today is known as "Somers' Algorithm".

The implementation used, represents the state of the board in part using three 32-bit words; one tracking the columns blocked by queens, viewing them only as rooks, and

the other two to track blocked diagonals and anti-diagonals in the current state of the board respectively. At every step of the search (i.e., search state) a copy of each of these words is preserved in the stack for the given state of the search. During backtracking, it is sufficient to return to the earlier state and evict from the stack the current state of the search. It is notable that the notion of a board is not represented as any sort of data structure directly, as the aforementioned words are sufficient to represent the search state.

This simple yet effective implementation leverages performance through heavy reliance on bit manipulation (i.e., bitwise operations) which are core to the function of processing hardware, thus typically among the most inexpensive operations in terms of processing time*. To simulate a recursive stack in an iterative implementation such as this, a number of parallel arrays are used to store each piece of data required (bit-words, etc.) that would otherwise be stored in a stack frame. This choice closely simulates the function of a recursive algorithm, albeit without any potential stack management overheads, and renders backtracking to an earlier state computationally inexpensive. Perhaps a drawback of such implementation however, is the amount of data that needs to be kept (estimated at $20 \times N^{\dagger}$ bytes) for a single search, and the reliance of caches to offer any form of data locality. The effect of such drawbacks are particularly detrimental to massively parallel implementations and we address them in our algorithm as presented in Section 4.2 and its implementations detailed in Sections 4.3.3 and 4.3.4.

4.1.2 The `DoubleSweep` Algorithm

The `DoubleSweep` algorithm combines basic word-level parallelism with basic ideas of look-ahead techniques [52, 72] from the domain of Boolean Satisfiability (SAT) solving. A key difference to Somers' algorithm (described in Section 4.1.1), is that `DoubleSweep` propagates placements through the whole board in such a manner as "unit-clause propagation" excludes unsatisfiable branches as part of SAT-solvers (discussed in more detail in Section 2.1.1). In essence, following each queen placement this process is initiated which then identifies unit rows, i.e., rows with only one available cell left, and infers their placements. The process repeats following every successful inference until a fixed-point, or until a row or column with no possible placements is identified.

Another key difference of `DoubleSweep` to Somers' algorithm is that `DoubleSweep` begins placements in the central row of the board rather than the first (top-most) row.

*It is noted that specific performance inferences cannot be made without knowledge of the processing hardware in question.

†As five 32-bit integers are stored per search state, in a stack with depth N. This calculation is purely for reference and does not reflect the actual memory requirements of an implementation, but rather a crude lower-bound estimate.

This branching heuristic helps make the propagation step more efficient, as central placements are more influential to the remaining rows. In this context, the influence of a placement is measured in terms of the cells of the board that become unusable, as this quantity is inversely proportional to the likelihood unit rows and potentially 'unsatisfiable' rows (i.e., row with no possible placement) emerge. Furthermore, `DoubleSweep` uses N words to represent the full board with current propagations on top of the three words used by Somers. In addition, the diagonal/anti-diagonal words used are 64-bit wide so that via a "sliding window" one can slide the bishop-moves over the whole board (back and forth) via the (word-level) shift-operations as explained in more detail in Section 4.3.

`DoubleSweep` is a powerful solving technique, however, special consideration has to be given when parallelising it, particularly in resource-constrained massively parallel environments such as that of the GPU, in part due to its branchful nature. Further discussion on this point can be found in Section 4.2.

4.1.3 Related Work on N-Queens Solving using GPUs

Numerous examples of GPU-based N-Queens solvers are available in literature [142, 129, 126, 87, 37, 108]. These attempts generally rely on single-kernel designs and place emphasis on heavy optimisation of implementations and the adaptation of conventional algorithms such as the aforementioned Somers' algorithm, to account for the specialities of the GPU environment. The need for such bespoke optimisations arises from the 'irregularity' of the computation at hand relative to the expectations of structure in computation imposed by GPUs [142]. The single-kernel approaches documented in literature attempt to combat the irregularity of the task at hand in different ways, providing an in-depth insight into different algorithms in the process.

The authors of [108] employ dynamic parallelism* to simulate backtracking search and in part alleviate data dependent divergence using three alternate branching strategies, each leading to new grids being spawned. In their work, the authors highlight the sensitivity of GPU hardware to data and computation representation, noting that their experimental results show consistently inferior performance to other GPU-based solvers, lagging behind even a reference sequential CPU implementation.

In their work, the authors of [129] take an approach, similar to that described in Figure 4.1, particularly Kern. 1 where workers in the GPU fetch a state from a queue, and explore the fetched state to produce N new states if it is valid, discarding it otherwise, dynamically pruning the state space. The authors note that despite the exponential growth of the problem which is ill-suited for GPUs, they were able to achieve a speedup over a threaded CPU implementation.

*The ability of a GPU kernel to launch and control sub-kernels.

Lastly, the authors of [37] implement Somers' algorithm opting for a densely packed representation of data on the GPU memory, and a thread-local stack implicitly stored in local memory. The authors iteratively improve their solution highlighting how shared memory can benefit this workload (primarily due to the long running nature of the kernel and the frequent memory transactions it involves), ultimately decomposing shared memory data into four shared memory arrays in a manner similar to Somers' implementation. The authors stop short of applying instruction-level optimisation, but underline the insufficiency of tooling in achieving peak performance for this hardware.

4.2 Adaptation of Solving Algorithms for GPUs

The `DoubleSweep` algorithm discussed in Section 4.1.2 makes heavy use of arithmetic and logic operations, while possessing traits that enable it to adapt to parallelisation, such as chronological backtracking and no look-backs.

A sensible parallelisation strategy in line with SIMT has independent searcher units to perform the same search procedure (instructions) over disjoint parts of the search space (data). In a search such as this, generating a number of disjoint search stating points trivially places parallel searchers on different non-converging search paths. The process of generating initial search states is itself non-costly, as in essence, it entails performing the same search algorithm to fixed-point over a number of paths as described in Section 4.3.1. In an implementation context of GPUs, the smallest unit of computation (i.e., thread) would act as the searcher unit. Particular details of such an implementation are presented in Section 4.3.

Some barriers do however exist in the implementation of 'conventional' algorithms on the GPU. Particularly with `DoubleSweep`, the propagation of queen placements is a branchful data dependent operation likely to result in severe thread divergence. In a worse-case scenario, in-sync threads in the same warp may diverge when all but one infer no placement while the remaining one infers several. In this scenario, placing a warp-level barrier would on one hand mitigate the divergence but would also result in the remaining threads idling unnecessarily and drastically reduce throughput. Likewise, enforcing no synchronisation would result in progressively worse thread divergence in the warp as threads continue to explore their respective portions of the search space.

The effects of centre branching are similar, as it is another operation introducing data-dependent branching. In early experiments with a full `DoubleSweep` implementation, we examined a potential remedy (or more appropriately, mitigation) to data-dependent centre row branching, which may be sufficient in preventing divergence for this operation. For further discussion, we refer the reader to Section 4.5.4.

We consider two approaches to map `DoubleSweep` to the GPU, namely that of decomposing the computation into multiple kernels described in Section 4.2.1 and that of a single long-running kernel performing the full computation which is detailed in Section 4.2.2.

4.2.1 Multi-kernel Decomposition of `DoubleSweep`

To alleviate the effect of the barriers encountered in parallelisation of `DoubleSweep`, a design comprised of a four-kernel pipeline is considered, where synchronisation is enforced globally by the host between every kernel invocation. This approach draws inspiration from the work of Meyer et al. [83] who use a six-kernel pipeline to simulate a divide-and-conquer algorithm for 3-SAT - a variant of the Boolean Satisfiability (SAT) problem.

In this application, S many initial states would be generated and stored in global memory on the GPU for S threads to operate on (1:1 mapping) across B many blocks of size $D = S \div B^*$. The following kernels are then used:

Kern. 1 **Propagation:** Thread t_i propagates queen placements in state s_i resulting in a state s'_i which replaces s_i in global memory. As part of this kernel, for each state an assessment is made and stored alongside the state in global memory, with one the following possible outcomes:

a) *solved*: The propagation of zero or more placements resulted in a solved board).

b) *unsolvable*: A conflict has been reached where no solution can result from further exploration.

c) *undecided*: Further exploration is required to conclude either of the above.

Kern. 2 **Summation**: Threads $t_0, t_1, \ldots, t_B$ within block $b_{x\in[0,B)}$ count collaboratively the number of states marked as *solved* in the region of states corresponding to them (i.e., $s_0, s_1, \ldots, s_B$) and atomically accumulate the results to a global counter. In doing so, the number of states with an *uncertain* assessment is also determined and stored at location u_x in a global array u of B many integers. Finally, the total number of states A with an *uncertain* assessment is accumulated atomically in global memory across all threads.

Kern. 3 **Defragmentation:** Threads $t_0, t_1, \ldots, t_{B-1}$ within block $b_{x\in[0,B)}$ transfer their respective states from global memory to an array in shared memory, then

*It is assumed that D is a whole number. The geometry of blocks is irrelevant to this application.

collaboratively discard *solved*, and *unsolvable* states and re-order those with an *uncertain* assessment such that they are packed densely at the beginning of the array. The threads also perform a parallel reduction operation subsequently, calculating $p = u_0 + u_1 + \cdots + u_x$. The integer p represents the number of states the blocks $b_0, b_1, \ldots, b_{x-1}$ are expected to accumulate and store in global memory, thus providing a memory location from which it is safe to write the states that remain following block b_x's processing.

Kern. 4 **Explosion:** Threads $t_0, t_1, \ldots, t_{A-1}$ each transfer their respective state $s_0, s_1, \ldots, s_{A-1}$ to shared memory. $N = S - A$ states have been eliminated by previous kernels that need to be replenished, therefore each thread t_i advances its corresponding state s_i, $w = (\lfloor N/A \rfloor) + [i < N \bmod A]^*$ times to produce w new states in global memory. Advancing a state here is performed in the manner of `DoubleSweep` and the following rules are adhered to:

a) If $w = 0$, s_i is advanced once and the new state s_i' takes the place of s_i in global memory.

b) If s_i is advanced to a *solved* state s_i' at any point during this process, s_i is returned to global memory.

c) If $w > e$ where e is the maximum number of advancements possible on s_i before no longer being assessed as *undecided*, t_i produces $w - e$ 'dummy' *unsolvable* states in place of the remaining states it should generate.

The host system launches the kernels in the aforementioned kernel pipeline repeatedly as shown in Figure 4.1, without parallel kernel execution (as supported by modern GPU hardware) at any point. The host continues until the integer A, read from the GPU memory, is zero, at what point the number of solutions can be read from GPU memory. It is noted that following the initial state generation, data remains resident on the GPU's global memory. There are two distinct buffers r_0, r_1, each with sufficient room for S states. On each invocation of the aforementioned kernel pipeline, Kern. 1 and Kern. 2 operate on buffer $r_{g \in \{0,1\}}$ whereas Kern. 3 reads data from r_g and writes back to $r_{(g+1) \bmod 2}$. Likewise, Kern. 4 operates entirely on $r_{(g+1) \bmod 2}$. This is done to prevent a data race whilst defragmenting buffer r_g, which may occur in Kern. 3 if some block b_x finishes before b_{x-1} begins, potentially overwriting the region of b_{x-1} as there is no defined order of execution of blocks in CUDA. On each subsequent invocation of the pipeline, r_g corresponds to the buffer last operated upon by Kern. 4 of the previous pipeline.

*Iverson bracket notation is used in this formula. $[P] = \begin{cases} 1 & \text{if predicate } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$

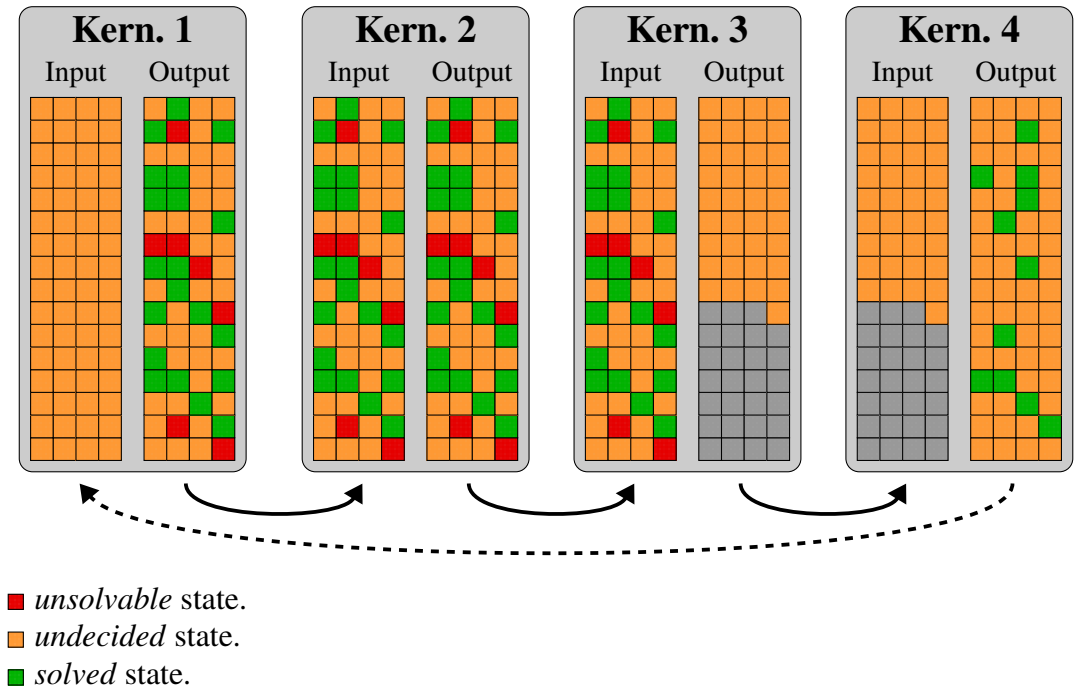


Figure 4.1: Input and output state buffer of each kernel in a four-kernel pipeline performing the `DoubleSweep` procedure.

This approach leaves little opportunity for threads to diverge as for most kernels (e.g., Kern. 2) in their warps and minimises the effects it has when it happens as the lifetime of each thread is limited. Additionally, this approach is flexible and can scale to any number of threads provided enough initial states can be generated. The costs however outweigh the benefits in this case. Preliminary tests yielded unsatisfactory results, making it unviable even for relatively small instances (e.g., $N = 15$) which took more time than a sequential CPU computation.

The failure of this approach is largely attributed to memory interaction forming a significant portion of each kernel's execution time (i.e., memory-bound kernels, bounded by latency). Thread context-switching is cost-free in the GPU and typically off-chip memory access latencies are hidden by swapping stalled threads for ones that are ready to execute. In this case however, the compute component for some kernels (i.e., Kern. 3 and Kern. 4) is insignificant compared to their memory interactions.

Besides the poor performance of the kernels themselves, communication between host and device is generally a costly operation due to overheads and data transfer speeds over the shared PCI bus which are difficult to hide behind parallel computation.

Perhaps the distinction between this unsuccessful attempt and the successful approach of Meyer et al. [83] is the nature of computation at hand. Solving the N-Queens problem forms an irregular task for the GPU yet the solving effort, particularly for `DoubleSweep`, is comprised of a very large number of small, inexpensive operations. Decomposing the task into distinct components (kernels), frames the computation in a favourable (for the GPU) manner however, the components themselves are not sufficiently large to even approach peak computation efficiency. Additionally, this approach suffers from work starvation at the tail-end of the computation, as Kern. 4 is unable to advance states the required number of times to produce sufficient candidate states for all threads of Kern. 1 to operate on, thus being forced to produce dummy unsolvable states instead.

4.2.2 `DoubleSweep-Light`: A GPU-centric Approach

As discussed in Section 4.1.2, `DoubleSweep` contains a number of features which are powerful, but incur high costs when implemented for GPUs. Perhaps the most significant cost stems from the branching nature of the algorithm which introduces further data-dependent branches in an implementation. As such, a number of adaptations have been made to produce the `DoubleSweep-Light` algorithm which bridges the divide of algorithmic performance and feasible optimisations in implementation by amalgamating elements of `DoubleSweep` and Somers' algorithm. Implementation details of this algorithm for the massively parallel GPU environment are detailed in Section 4.3.

`DoubleSweep-Light`, like `DoubleSweep`, works by making a placement of a queen on the board, followed by a propagation step. The main difference between the two is the propagation step: `DoubleSweep-Light` does not perform full propagation through multiple sweeps over the board, but instead propagates the rows directly following the row in which the placement was made only once, which reduces the overall degree of divergence between threads in a warp. More specifically, given a partial configuration of queens on a chessboard and a backtracking limit (i.e., the index of a row beyond which backtracking is not permitted), the function `advance_state` presented in Listing 4.1 is applied. This function identifies a column (Lines 4–9) suitable for a queen placement in the current row (i.e., the row following that of the last placement), making the placement and advancing the state (Lines 10–14) if such column is found, or backtracking (Line 16) and re-trying otherwise. In essence, backtracking is performed when no queen can be placed in the current row due to conflicts with previously placed queens. The reasons for limiting backtracking are detailed in Section 4.3.1.

It is worth noting that the form a state (`s`) takes in Listing 4.1* and subsequent listings is that of a structure, the contents of which include the current row (`current_row`)

*In this listing, the notation $]idx,N[$ signifies an interval exclusive of bounds (i.e., $\{a \in \mathbb{N} | idx < a < N\}$).

```
1  fn advance_state(s, locked_idx):
2    let cr ← s.current_row
3    while locked_idx ≤ cr.row_index < N do:
4      let idx ← cr.current_queen_index
5      foreach i ∈]idx,N[ do:
6        if ¬has_diagonal(s, i) ∧ ¬blocked_col(s, i) then:
7          idx ← i
8          break
9
10     if idx ≠ cr.current_queen_index then:
11       place_queen(s, cr, idx)
12       let x ← min(cr.row_index+1, N−1)
13       s.current_row ← s.row_at[x]
14       return ⊤
15     else:
16         cr ← s.row_at[cr.row_index−1]
17   return ⊥
```

Listing 4.1: State advancement algorithm.

and a list of structures (`row_at`) each containing per-row information. The per-row structure holds information such as the index of the queen placed on that row (`current_queen_index`) and the index of this row in the state (`row_index`).

```
1  fn derive_queens(s):
2    start:
3    let free ← nil
4    foreach i ∈ [0,N[ do:
5      if ¬has_diagonal(s, i) ∧ ¬blocked_col(s, i) then:
6        if free ≠ nil then:
7          return ⊥
8        free ← i
9    place_queen(s, s.current_row, free)
10   s.current_row ← s.row_at[s.current_row.row_index + 1]
11   goto start
```

Listing 4.2: The `DoubleSweep-Light` algorithm.

Following the advancement of a state by applying `advance_state`, the propagation step is performed on the state `s` by applying the function `derive_queens` presented

in Listing 4.2. This function identifies which column (if any) in the current row of `s` is free (Lines 3–8) and if such a column exists, places a queen, repeating the same operation in the following row (Lines 9–11).

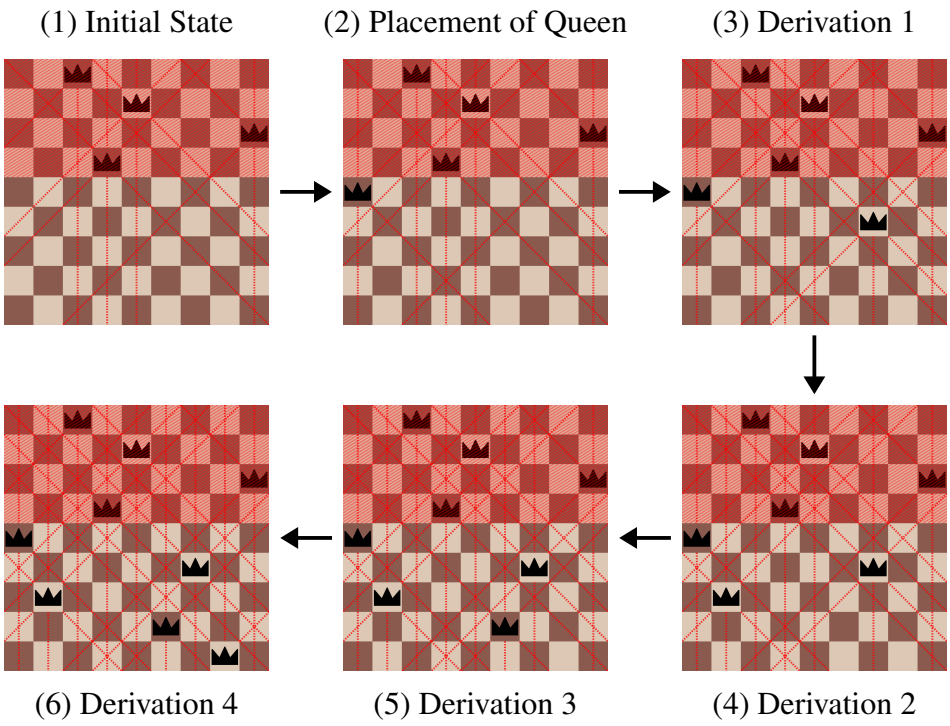


Figure 4.2: Step-by-step application of `DoubleSweep-Light`.

The process described in this section is presented visually in Figure 4.2. First, the initial state (1) is the partly complete non-attacking configuration upon which `DoubleSweep-Light` is applied. The red region signifies rows (0 to 3 from the top, counting from 0) for which modification is forbidden (i.e., backtracking will be limited to not modifying of any of these rows). Initially, `advance_state` is applied which results in the placement of a queen in row 4 in one of the two possible positions seen in (1) to yield (2). From here, `derive_queens` is applied starting from the following row (row 5). This row is 'unit' (i.e., only one valid placement can be made in that row), hence we place the queen there yielding (3). The placement of this queen results in the following row (row 6) becoming unit which results in another placement as seen in (4). Once again, the placement of this queen makes the following row (row 7) unit, and this cascade effect continues as seen in (5) and (6) resulting in a complete non-attacking configuration through derivations. If we found we could not place a queen, then we would backtrack undoing the unit derivations and explore the only other possible position of the queen in row 4.

4.3 Implementation of DoubleSweep-Light on GPUs

Enlightened by the observations and results of existing work in the field, we implement our DoubleSweep-Light algorithm using a single-shot kernel approach to produce a full-GPU solver. Our focus is solving performance and the regularisation of the task where possible. This motivates the choice for a single-shot kernel approach, as opposed to a multi-kernel one, as following our initial experiments (see Section 4.2.1) we deemed such approach unsuitable for this task.

Unlike existing work [141, 129] which relies on dynamic state generation during solving, our approach treats the smallest unit of computation, a thread, as a self-contained individual solver performing DoubleSweep-Light to explore its own sub-tree of the search space. More specifically, each thread of the kernel is assigned an initial, partly explored search state from a pool (generated in the manner explained under Section 4.3.1) upon which the lifetime of the thread depends.

Initial/partial search states are generated on the host-side, and transferred to the global memory of the device as a contiguous array of structures (structs). The structure of each state, visualised in Figure 4.3*, is comprised of the following components:

- Per-row projections of conflicting diagonals caused by placed queens (two 64-bit bit vectors for the diagonal and anti-diagonal respectively, see Section 4.3.2).
- The occupied columns (i.e., placed queens, a single 32-bit bit vector).
- The index of the current row (a single 8-bit integer).
- The indexes of placed queens on the current state (array of N many 8-bit integers).

The size of the state structure (struct) varies depending on the value of N, as a result of the placed queen indexes array member. The remaining components of the struct are 8-byte aligned and laid out as shown in Figure 4.3 with the 8-byte boundaries highlighted. This structure minimises padding, in an effort to reduce the overall size of state pools and allow a greater number of states to be stored in the device's memory. We note that this layout results in a well-packed struct on tested compiler versions, however, padding and struct member layout is ultimately determined by the compiler. Misaligned memory in the device does not, on the one hand, cause unexpected or erroneous behaviour, however it results in multiple interleaved memory instructions which disrupt access patterns and degrade memory fetch performance.

*The implementation of this struct is shown in Listing C.3 of Appendix C.1.

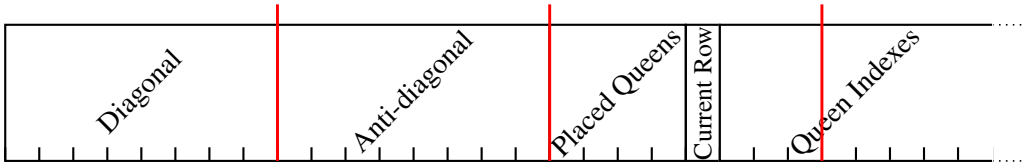


Figure 4.3: Visual representation of an N-Queens state in GPU memory. Red protruding lines denote eight-byte boundaries and each tick denotes a byte.

Besides well formed memory that minimises transactions, a space-conserving design such as the one shown is crucial to keeping each thread's state resident in on-chip memory. Data locality is of paramount importance for fast access, even in modern hardware with superior caching capabilities. In this application, cache schemes and on-chip memory partitioning is unlikely to fully eliminate transactions from reaching global memory and restricts control over the structure of on-chip memory which must be informed by the expected access patterns. Further discussion on this topic is presented by each kernel implementation (Sections 4.3.3 and 4.3.4).

It is noted that the value of N is set and known at compile time, which enables a fixed array member of N elements to exist in the aforementioned struct. In addition, the availability of this number at compile time aids the generation of commonly used bit masks at no run-time cost, as well as pre-computation of bounds by the compiler.

Following the transfer of a pool of states to the device, a kernel is launched comprising of T many threads mapping one-to-one to the states in the pool. It is desirable for $T \approx C \times P$ where P is an empirical over-subscription factor. Whilst it is often thought that threads map one-to-one to the number of cores in the SMs of the GPU and therefore as many threads as cores are needed, this is not strictly true. This mapping hinders the ability of warp schedulers to switch between threads to keep hardware resources occupied while other threads are blocked (i.e., hiding latencies) and the availability of additional work should a block complete sooner than the rest. Our aim is to have a surplus of blocks of threads to allow for finer balance of work across SMs. The degree of over-subscription is empirical, as a multitude of factors relating to the device's capabilities (e.g., core count, per-SM block capacity, potential block size, etc.) have to be taken into consideration.

This workload does not benefit from spatial locality, therefore we have adopted the simplest grid/block topology using mono-dimensional blocks which in turn are part of a mono-dimensional grid. This gives us a theoretical limit of threads per device of $2^{41} - 2^{10}$ threads based on CUDA limits, which is unlikely to be reached by any kernel. We have provided two kernel implementations of `DoubleSweep-Light` each suited best to devices with different characteristics, namely a shared memory-based

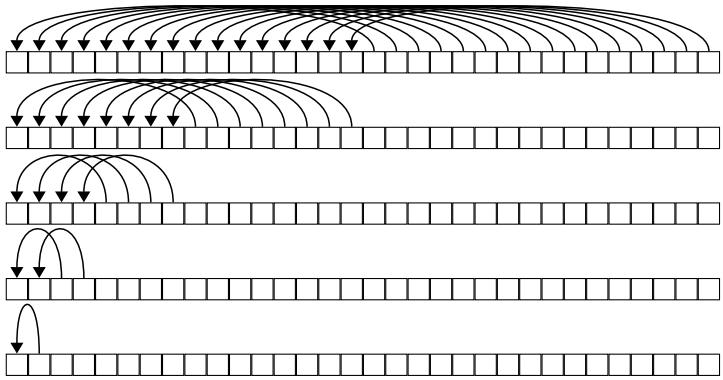


Figure 4.4: Visualisation of the steps involved in warp down-shuffling.

kernel and a register-based kernel presented in detail in Section 4.3.3 and Section 4.3.4 respectively.

In both kernels, each thread accumulates its number of solutions in a local 64-bit variable throughout its lifetime*. Upon completion, the thread first doubles its solution count where appropriate to account for vertical reflections (see Section 4.3.1) and then performs an atomic addition operation, accumulating its solution count in a global memory location. Each block is granted a separate accumulation point in an effort to spread atomic operations over different locations and reduce any performance impact they may have. In our earlier work [101] we employed warp down-shuffling as illustrated by Figure 4.4 to perform a parallel reduction operation within the block and summarise the block's total solution count efficiently, as opposed to using an atomic add operation. Warp-level primitives enable threads within the warp to exchange register contents instantaneously yet still in a synchronous manner to preserve data integrity. The later decision to use an atomic add operation came as optimisations enabled our solver to reach larger values of N, where per-block results surpassed the limits of a 32-bit unsigned integer - the largest supported in warp primitives. Atomic arithmetic read-update-write operations between 64-bit unsigned integers to global-memory are long supported in CUDA, but come at a comparatively higher cost to the previous approach. The cost of result summation however, is only relevant to the overall computation time of smaller values of N as after a point the computation effort sufficiently hides them.

Once a kernel completes execution, the host system which has been awaiting, reads the array of results from the device, and sums them to produce the total count of solutions identified by that GPU. If multiple devices attached to the host are involved in the

*As 64-bit operations are emulated on the GPU, this can be a relatively costly operation, with room for optimisation as detailed in Section 4.3.6.

computation, the host awaits for their completion before accumulating the final result across all. This result may not be the instance solution, either because the state pool used contained a subset of states generated (e.g., in a long computation performed in stages), or multiple other hosts are involved (e.g., in a distributed platform).

The remainder of this section presents the operation of the solver in more detail starting with the generation of initial states on the host-side (Section 4.3.1) and the scalability of our approach on multi-GPU platforms (Section 4.3.5) before diving into implementation details on the device-side starting from the tracking of diagonals (Section 4.3.2) and leading up to the specifics of each kernel implementation (Sections 4.3.3 and 4.3.4) and the optimisations employed in these kernels (Section 4.3.6) to produce results surpassing existing work in literature (Section 4.4). Finally, the criteria for choosing the most appropriate implementation per device are presented (Section 4.3.7).

4.3.1 Initial State Pool Generation

To generate a pool of initial states (i.e., a set of partial non-attacking configurations), a `DoubleSweep-Light` search is performed up to a certain depth. More specifically, a range of rows is chosen on an N-Queens board, which are to be populated with queens. The maximum number of possible (partial) states can easily be calculated for a given cut-off depth, however such naïvely generated states often contain a large number of 'invalid' search starting points such as ones which cannot be advanced further, therefore the number of valid, advanceable states generated is often far smaller.

The state generation process produces partial states, which can be advanced at least once. These states have a certain number of 'locked' rows, meaning that when `DoubleSweep-Light` is performed on these states, these rows must not be altered. The advancement algorithm presented in Listing 4.1 takes into consideration the index of the last locked row, on or prior to which backtracking must not occur. This approach is an incarnation of the top-down decomposition procedure for splitting the search space, presented in [78]. In their work, the authors present the benefits their method offers to parallel solving of constraint satisfaction problems through the N-Queens problem, approached using constraint satisfaction heuristics. The authors demonstrate the effectiveness of this technique in workload balancing which is of great importance to massively parallel applications such as this.

In practical terms, to generate a pool of approximately S many states for a given value of N, a ladder-climbing approach is employed, as presented in Listing 4.3. Initially, a row $R = \lfloor log_N(S) \rfloor$ (Line 2) is chosen, under the assumption that all naïvely generated states obtained by populating the first R many rows are valid and advanceable. A pool of states s_0 is subsequently generated by applying a modified version of

```
1  fn gen_state_pool(N, s, f):
2    let r ← ⌊log_N(s)⌋
3    let S ← ∅, S' ← ∅
4    do:
5      S' = S
6      S ← gen_states(N, r)
7      r ← r + 1
8    while |S| ≤ s ∧ r < N − 1 ∧ |S| > |S'|
9    if |S| ≤ (s ∗ f) then:
10     return (S, r)
11   else:
12     return (S', r)
```

Listing 4.3: Ladder-climbing algorithm for initial state pool generation.

`DoubleSweep-Light` (seen as the function `gen_states`) which stops placing/deriving queens after a certain row (Lines 3–8).

If $|s_0| < S$, R is incremented and state generation is repeated, until some generated pool s_i satisfies $|s_i| > R$. At that point, a choice is made to either keep s_i if $|s_i| < (R \times F)$ for some constant factor $F \geq 1$, or to discard s_i and keep s_{i-1}. Here, S is treated as a soft limit, and the constant factor F serves as means of determining the hard upper limit. The flexibility in the upper limit is employed as the number of states generated by 'locking' on subsequent rows may vary wildly, however discarding a pool of states for slightly exceeding the desired number of states in the pool s is undesirable.

If at any point $|s_i| \leq |s_{i-1}|$, state generation will stop and keep the set of states s_i, as the generation is following a downward slope. Generally, this is encountered when a large number of states is requested for a small value of N, at which point state generation explores the search paths enough to encounter conflicts and discard more than it generates, in effect, solving the given instance. For larger values of N, this is unlikely to be the case as generally, an immense number of states would have to be generated before the search plateaus and enters the downward slope.

It must be noted that, during state generation, only the leftmost $\left\lceil \frac{N}{2} \right\rceil$ cells of the first row are considered for queen placements. This operation reduces the search space by approximately half, as any states generated by queen placements on the remaining cells of the first row would be symmetries of those, on the vertical axis. Consequently, solution counts obtained by exploring these generated states would have to be doubled. An exception to this rule applies to solution counts obtained by exploring states with

a queen placed at cell $(\left\lfloor \frac{N}{2} \right\rfloor, 0)$ when $N \bmod 2 = 1$ where the solution count is not doubled as their vertical mirroring yields no new state.

In terms of implementation, the state generation process described above can be implemented on the device-side, however, we saw little benefit in doing so considering it is a relatively inexpensive process computationally, which demands large amounts of memory that the GPU would not be able to provide. As a result any implementation would have to transfer large amounts of data through the PCI bus to the host system's memory incurring significant costs. For this reason, we perform this process on the host-side using a threaded implementation which crudely produces a different starting point state for each thread by placing one queen in each of the first $\left\lceil \frac{N}{2} \right\rceil$ cells of the first row, in the fashion explained earlier. More specifically, each host-side thread t_i begins by placing a queen on the i^{th} column of the first (0^{th}) row in its (initially empty) state, which remains untouched throughout the state generation process while the thread explores all valid placements in the remaining rows up to the cut-off depth.

The need for an approximate target number of states in the state generation process arises from the need for a sufficiently large pool to supply every participating solver in the computation. This number is dependent upon a variety of factors such as:

- The total number of devices used.
- Memory and compute capabilities of each of these devices.
- The desired over-subscription factor per device.

Additionally, as provision for checkpointing is intentionally not implemented, it may be necessary (particularly in shared-resource environments such as supercomputer clusters) for longer computations to be partitioned into smaller sub-computations that collectively amount to the final instance solution, without having to be performed at once. This amounts to a requirement for yet more states to be generated.

With our approach, the overarching limitation in the depth of state generation is the memory available in the host machine. Particularly for large scale computations (e.g., $N = 25$, in the experimental setup described in Section 4.4) this may easily be reached even in High Performance Computing (HPC) systems. To compensate, our implementation makes use of memory-mapped secondary storage in this process. We opted to manually implement this feature as opposed to rely on swap mechanisms employed by operating systems to be able to use all available physical storage space as often swap space is limited. Furthermore, swap mechanisms broadly work on prediction models to determine which memory pages may be written to secondary storage with the least disruption to the system, which is associated with certain overheads.

Data is generated in a linear fashion* during state generation and can avoid these overheads by being written directly to secondary storage.

Following the generation of a state pool, the states in it are shuffled randomly using the Fisher-Yates shuffle algorithm [39]. Although memory transaction-intensive, this step is important to counter the effects of irregular work distribution which may result by clusters of 'difficult' states accumulating close together in the pool. This phenomenon is documented in literature [24] where the authors explore its impact on work distribution among parallel workers. Through our work, we reinforce their findings with an example we encountered during our experiments, shown in Section 4.4. By shuffling the pool randomly concentrations of computationally 'demanding' states are likely broken and distributed, meaning partitions given to devices are likely to contain a more uniformly balanced workload. This step does not however aid inter-GPU work distribution as more 'demanding' states may be part of warps comprised of less demanding ones, keeping the warp active for only a handful of states to be completed. We aim to address this problem in a way described in Section 4.5.1.

Once the state pool is finalised, states are written to either an individual or a number of files. As these files typically need to be transmitted over network mediums (particularly between cluster nodes), their size should ideally be kept to a minimum. In the interest of reproducibility, the format is detailed below.

The format of these files is comprised of a 22-byte header at the start of the file containing in order:

- A 32-bit file version integer which must match the implementation's value.
- A 64-bit vector of generation flags that may affect how the data in the file is interpreted. Currently only two bits are in use:
  - Bit 0 (LSB) is set if experimental optimisations were enabled during generation.
  - Bit 1 is set if states were generated excluding vertical mirrorings as described earlier in this section.
- An 8-bit integer holding the value of N for which the states were generated.
- An 8-bit integer L holding the index of the last locked row (common across all states).
- A 64-bit integer T of the total number of states contained within this file.

*With respect to each thread's execution.

This file header is followed by T many blocks each comprised of L many 8-bit integers. Each block specifies the configuration of a state, and specifically the column indexes of queen placements for the first L rows. Data typically found in the representation of a state in memory are stripped away as they can be reconstructed using only this information.

4.3.2 Tracking of Diagonal Occupation

Two 64-bit bit vectors are used to track occupied diagonals for each N-Queens state, as previously described in Section 4.3. On first examination, 16 bytes may appear a hefty sacrifice to make especially considering scalability and the use of size-constrained memory regions such as shared memory. This however, is an integral component of the DoubleSweep-Light implementation that allows for the construction of a per-row projection of diagonal/anti-diagonal occupation.

Initially, the state each thread is assigned contains the pair of bit vectors V_d and V_{ad} respectively, with bits set to match the queens currently tracked in the state. Subsequently, following every queen placement by the thread, this pair of values is updated to reflect the change. In essence, each bit in these bit vectors corresponds to a diagonal/anti-diagonal respectively, therefore for an $N \times N$ board, there may be up to $2 \times N - 1$ diagonals and as many anti-diagonals to track. For instance, Figure 4.5 depicts the mapping of diagonals to the diagonal tracking bit vector for a given 8×8 board, where 0 or 1 represent the absence or presence of a queen in the corresponding diagonal respectively. This process applies similarly to anti-diagonals.

During solving, and upon placement of a queen on a row r and column c, a pair of bit masks are calculated, one for the diagonal $m_d = 1 \ll (c + r)$ and one for the anti-diagonal $m_{ad} = (1 \ll c) \ll (64 - N - r)$ which are then used to compute the updated value of $V_d = V_d \,|\, m_d$ and $V_{ad} = V_{ad} \,|\, m_{ad}$ respectively. When eventually this placement is undone (during backtracking), the bits set by m_d and m_{ad} are simply toggled off as $V_d = V_d \,\&\, \overline{m_d}$ and $V_{ad} = V_{ad} \,\&\, \overline{m_{ad}}$.

To determine which columns are non-conflicting for a given row r, we utilise the above pair of bit vectors along with the bit vector tracking the blocked columns B that each tread maintains. First, we extract the projections of diagonals for the current row $p_d = V_d \gg r$ as well as the anti-diagonals $p_{ad} = V_{ad} \gg (64 - N - r)$, and then derive a bit word of available columns for this row $a = \overline{B \,|\, p_d \,|\, p_{ad}} \,\&\, X$, where X is a bit mask with N set bits, computed at compile time. Following this, the positions of set bits in a correspond to the columns where a queen can be placed without conflicting with existing placements.

Bitwise expressions such as these are quick and involve no branching, rendering the cost of bit manipulation and mask construction relatively small. This is ideal for a

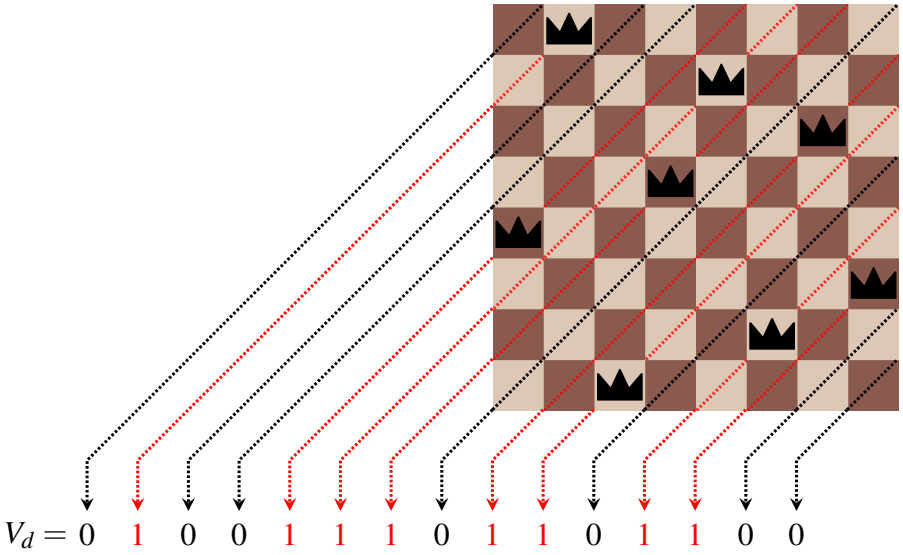


Figure 4.5: Mapping of queens on diagonals to the diagonal tracking word.

situation like the above, since the aforementioned computations, and especially the checking of available columns, are performed very frequently in our implementation. Newer device architectures (discussed further under Section 4.3.6) introduce hardware support for bit mask calculations.

It must be noted that, GPUs are (currently) 32-bit devices, with support for 64-bit instructions resulting from emulation. Whilst official documentation on the performance of such emulated instructions is shallow, micro-benchmarks in literature [2, 62, 77] show that simple instructions such as the ones used for diagonal tracking, require roughly twice as many cycles as their 32-bit counterparts (i.e., are emulated using two 32-bit instructions). We discuss the importance of micro-gains in performance to this task, and potential optimisations to them in Section 4.3.6.

4.3.3 Shared Memory-based Kernel

Shared memory offers significantly cheaper access costs than global memory. The shared memory-based implementation of `DoubleSweep-Light` has each thread in each block reserve a portion of shared memory for its computation exclusively. At the beginning, the thread transfers its corresponding state to shared memory following the representation presented in Section 4.3. The implementation of this kernel is shown in Listing C.12 of Appendix C.5.1.

The per-row indexes of each N-Queens state are necessary to facilitate backtracking. During backtracking, placed queens need to be removed from the state, meaning the

tracking of diagonals, anti-diagonals and occupied columns needs to be updated, which can only be done knowing the position of the removed queen. Traditional recursive implementations would use the call stack for this purpose, but due to limitations in size and control of data we chose to manually track this data in an iterative implementation instead.

During computation, each thread in a warp operates on the data in its shared memory, by performing the DoubleSweep-Light procedure. Warp divergence cannot be eliminated completely in the implementation of DoubleSweep-Light as the number of iterations made by each thread is data-dependent. Warp synchronisation barriers are interleaved between state advancement and propagation in an effort to re-converge divergent threads where appropriate, mitigating some of the effects of divergence.

Following every successful propagation and derivation, involved threads within a warp are balloted to determine if all have completed their task. If at least one thread votes against stopping in the ballot, the warp continues, with completed threads being left inactive. Performance may degrade if all but a few threads in a warp remain active, however, generally, we found this to not be a detriment. Additionally, the choice to store queen indexes as 8-bit integers was driven primarily by the constrained size of shared memory. This, paired with the data-dependent access patterns exhibited by the threads renders bank conflicts unavoidable. Due to the high compute load of the kernel, however, the overall impact of such conflicts does not seem to greatly impact the kernel.

Each thread uses an unsigned 64-bit integer to count the number of solutions it finds for its given state. After successfully advancing its state, each thread increments this counter by either 1 or 0 depending on if the state is a complete non-attacking configuration or not. This is 'cheap' to establish computationally purely through the occupied columns bit vector since having N set bits in this vector guarantees all N columns have been populated and the configuration is non-attacking. In practice, this is a simple comparison with a compile-time generated bit mask, which paves the way for further optimisation discussed in Section 4.3.6.

Reliance on shared memory does in some cases impact the number of threads per block. Our goal is to maximise the utilisation of SMs and concurrent solver threads. We calculate the size of the block in a warp-centric manner, taking into account the size of each struct t in bytes, the maximum shared memory size s in bytes, the maximum number of warps in a full block w, and the maximum number of threads in a full block m by first calculating the number of threads per block

$$b = \left\lfloor \frac{s}{t} \right\rfloor$$

and rounding to the nearest multiple of w, to obtain the number of threads per block b'

$$b' = min(b - (b \bmod w), m)$$

Each piece of information required for this computation is known at compile time allowing for the above computation to be encoded as a preprocessor-evaluated directive (see Listing C.5). In turn, information on the launch bounds of the kernel can aid the device code compiler in more efficiently allocating resources for the kernel. Such information is provided through the `__launch_bounds__()` qualifier available in the CUDA toolkit [89].

The number of blocks in the grid is equally easy to calculate for a pool of p many states, as $\left\lceil \frac{p}{b'} \right\rceil$. It must be noted that some architectures support multiple blocks residing in the same SM, provided sufficient resources are available for all of them to co-exist. It may be preferable depending on the architecture's capabilities for multiple smaller blocks to be launched versus maximally sized ones. Due to the high degree of architecture-specificity associated with this decision, we opted to maximise the block size as means of achieving good performance irrespective of the architecture specifics, a decision biased in part by the hardware available to us. Additionally, the transfer of data from global to shared memory forms an insignificant portion of the computation, therefore global memory access coalescing is not considered in this instance.

4.3.4 Register-based Kernel

As discussed in Section 2.3.1, register space can be seen as the fastest type of 'memory' available. Whilst strictly, hardware registers are a very different component to addressable memory (either on- or off-chip), as they behave differently and are intended to be used as (perhaps temporary) storage 'pockets' for instruction execution, they are significantly faster to interact with than any addressable memory in the GPU. This of course is to be expected, since no explicit fetching or writing instructions need to be used to interact with them. Register access and register-to-register instructions should not be thought of as instantaneous however, as register fetch and pipeline latencies are involved. For more information we refer the reader to [62, 89] and [131].

In compute-intensive workloads such as ours, performance gains can be made simply by altering the expression of the computation at hand on the high-level program side, to enable the compiler to better distribute data in the registers available. Similar techniques such as warp caching* are one of many advanced optimisations known to the domain of high-performance GPU programming.

*For further discussion on this matter, we refer the reader to: `https://developer.nvidia.com/blog/register-cache-warp-cuda/`

In earlier times, C-based programming languages offered an interface for hinting at the compiler which pieces of data the programmer deemed beneficial to belong in register space through language constructs such as the `register` qualifier [58]. More recently, such hints are often ignored entirely by compiler tooling in favour of register allocation heuristics [135]. This renders such optimisations not definitive nor forward-compatible, with new heuristics or fluctuating register capacity in newer hardware being factors potentially influencing register allocation.

We present adjustments made to the kernel implementation described in Section 4.3.3 in order to format data such that they are eligible for placement in registers by the compiler. As part of the process, we confirmed our changes reflected in the resulting binaries through different compiler versions and hardware capabilities* to exclusive use of register space as desired. We have applied these changes to form a register-based kernel implementing `DoubleSweep-Light`.

Unlike the shared memory implementation of this kernel, at the start of computation, all but one members of the state struct of each thread are loaded directly into thread-local variables from global memory. Surrounding code has been adjusted to implement `DoubleSweep-Light` using these variables instead of addressable memory, without significant change to the code flow (see full implementation in Listing C.13 of Appendix C.5.1). The one member of the struct which remains in shared memory is the array member as by its nature, it must remain in addressable memory space. We examined the potential impact of morphing the computation in a way that this data was encoded in register space (implicitly addressed using conditionals) but concluded it would most likely prove detrimental to performance and decided against it.

As a result of the aforementioned changes, shared memory requirements are significantly reduced for this kernel implementation. For a given value of N, each thread requires N bytes of shared memory. Given the maximum number of threads per block b_{max}, number of threads in a warp w, and shared memory size s we calculate the number of threads per block as

$$b = min(b_{max}, \left\lfloor \frac{s}{w \times N} \right\rfloor \times w)$$

Like with the shared memory-based implementation, this calculation is performed at compile time, aiding the compiler in resource allocation.

We note that the `register` keyword appears prepended to variable declarations in our implementation more so as means of self-documenting code, and less for its value in guiding the compilers register allocation heuristics which may or may not consider it. In our limited, crude examination of its effects, we observed no difference in final

*Platforms used to gather performance results for this kernel as shown in Section 4.4 made full use of register space as intended.

register allocation when using register hinting than without doing so*. As a result of these qualifiers however, we did have to alter code flow and manually inline most function calls requiring pointer passing, as pointers to register-qualified variables must not be obtained in C-style languages even if some compilers define the behaviour.

The benefits of the change from shared memory to registers are two-fold:

1. The reduced shared memory requirements allow for full blocks to be allocated per SM. Typically, a warp is comprised of 32 threads, and there can be up to $1,024$ threads per block. Even the earlier architectures featuring $48,000$ bytes of shared memory per block per SM would end up having at least one full block per SM for all $N \leq 46$.

2. Register space is thread-local, significantly faster and with less stringent access pattern expectations than shared memory†.

It must be noted however that high-level code has no direct control over register allocation. Attempts to interfere with the compilation toolchain in register utilisation are objectionable, as beyond violating programming standards, they often hinder compiler optimisations and overall result in performance loss. Likewise, exhaustion of register space has adverse effects on overall performance.

In our experiments, this kernel generally compiled without excess register usage and resulted in significant performance gains on tested architectures, detailed in Section 4.4.

4.3.5 Solving on Multi-GPU Systems

We have designed our solver to scale on any number of parallel GPUs provided enough initial states can be generated, in the manner described in Section 4.3.1, which renders it easily horizontally scalable in distributed environments such as clusters. Here, the idea remains the same as before where the number of required initial states is product of the number of GPUs involved in the computation as a whole. The generated state pool is partitioned upon generation to sub-pools, specific to each GPU involved. Following the completion of the computation, the results from each GPU can be collated to produce the final sum in the manner described in Section 4.4. The absence of dependency between solvers (and by extension, GPUs) renders this approach linearly scalable w.r.t. the number of solvers in theory, yet practically scaling follows a quansi-linear trend as discussed in Section 4.4.

*Tested on NVCC version `11.7.64`.

†This is not to say register allocation is guaranteed to be perfect. Refer to Section 4.5.5 for further detail.

In practical terms, scaling is limited by resources as catering for an arbitrary number of GPUs requires a sufficient amount states to be generated. For larger values of N, the search space is large enough to cater for very large numbers of initial states to be generated, leaving the availability of sufficient computational resources for the state generation process as the only limitation thereafter.

Operating this type of solver on distributed computing environments (e.g., cluster topologies) does in some cases present a disadvantage. Typically, cluster systems are comprised of any number of nodes, each of which may be of different specifications to the rest, even at different physical locations. A job is typically submitted through a 'login node' and scheduled automatically to execute in a set of nodes when sufficient resources become available. Jobs hold various parameters for the work that will eventually be carried out and often come with a limit on the time available to carry out such work in the interest of fair resource sharing. The specifics of cluster topologies, available hardware, limits, job scheduling, etc. can vary greatly between cluster setups.

4.3.5.1 Assessing the Potential Performance Impact of Checkpointing

Overcoming time limits in cluster environments for long running GPU computations is significantly harder than their CPU counterparts, as the CPU-side can receive and process interruption signals near the end of the allotted computation time slot, and take action to save the current state of the computation (i.e., create a checkpoint). For computations on a co-processor such as the GPU however, this is not as easily achievable, as any signal would have to be received by the host system which then has to take action to halt the computation on the GPUs it hosts, in a controlled manner, collecting and preserving the state of the computation in the process. Stopping the threads in a massively parallel environment is difficult and further complicated by the absence of an overarching thread controller as is the design in GPUs.

A controlled stop of threads in the GPU can be achieved, if initiated by each thread upon a condition being met which triggers the thread to cease further work. In this scenario, threads individually and periodically poll a shared (with the host) memory location in anticipation of a termination signal. Upon receipt of such signal, threads write their search state to a corresponding memory location alongside the number of solutions they identified up to that point, enabling the host to store this state and resume it at a later time.

Perhaps the most important consideration in such a design is how to minimise the performance impact of per-thread logic for periodic checks of the global memory flag. The most light-weight approach we devised was to base the period of checks on the number of elapsed clock cycles. Using the special 64-bit register `%clock64`, accessible using a standard `mov.u64` instruction, a thread can access an unsigned counter holding

the number of clock cycles elapsed since the start of the computation*. If the value of this register is greater than the value was when the last check was performed by the thread plus a fixed constant, then a check can be performed again. Listing 4.4 shows the relevant CUDA-C code for this check which we considered for our solver. `gmem_stop_flag` is a volatile pointer to a 32-bit global memory location flag, controlled by the host, and `last_clock` is a local 64-bit variable storing the GPU clock at the time of completion of the last check. The actions taken to terminate the computation by the thread are abstracted as a call to the `nq_stop_computation()` function.

```
uint64_t clock_ticks;
asm volatile("mov.u64 %0, %%clock64;" : "=l"(clock_ticks));
if (last_clock + CHECK_CLK_THRESHOLD <= clock_ticks) {
 if (*gmem_stop_flag) {
  nq_stop_computation();
 }
 last_clock = clock_ticks;
}
```

Listing 4.4: CUDA-C code executed by each thread between processing tasks to check for a termination signal from the host.

In an application such as ours, threads are explicitly designed to avoid any interaction with global memory following the initial setup of their on-chip memory at the beginning of the computation, due to the high access cost of such interactions†. Introducing infrequent non-cached global memory 'checks' as described above would perhaps not result in significant performance impact, yet embedding logic in each thread to decide when time has come to perform such check would, in all likelihood. This is due to the work of each thread consisting of a large number of inexpensive tasks between which such logic is interleaved, in effect repeating this check logic very frequently. The incurred cost here takes the form of additional frequently repeating work for the SM.

We deemed the cost of checkpointing logic in the GPU too great and decided against implementing this functionality. We instead chose to partition the state pool empirically, for problem sizes likely to surpass time limits of the target cluster and instead dispatch multiple jobs to tackle each sub-pool without exceeding the limits.

4.3.6 Kernel Optimisations

Reliance on high-level program constructs and compilation tooling alone in CUDA, will likely achieve good but not peak performance [37, 63, 62]. Some sources [63]

*Or since the last wrap-around of the counter which occurs silently.

†Although not officially documented, it is generally accepted that global memory access is slower than shared memory by a factor of ≈ 10.

empirically claim the code generated by NVIDIA Cuda Compiler driver (NVCC) achieves approximately 80% of the theoretical efficiency in compute-bound applications. Whilst no experimental methods or data are presented to substantiate the claim, our experiments have also lead us to conclude that hand-tuning optimisations are required in pursuit of peak performance.

We focus our optimisation efforts to the N-Queens kernels described earlier in this section and in the process, identify a number of areas of improvement which can broadly be classified into two categories: high-level code alterations; and low-level instruction tinkering. Our optimisations under each are described in Section 4.3.6.1 and Section 4.3.6.2 respectively. Unless otherwise stated, our experiments and tests of these optimisations were carried out using CUDA version `11.7.64`. We refer the reader to the Godbolt Compiler Explorer [47] for ease of replication.

4.3.6.1 Surface Optimisations for N-Queens Kernels

Some degree of optimisation of kernels, like other programs in C-based languages, can be achieved simply by using language constructs [142, 37] and incorporating behaviour information of the compiler and hardware in re-structuring code. We regard such optimisations as 'surface-level' as they can generally be performed safely and in a forward-compatible manner.

As part of this process, we examined the impact of the pointer aliasing restriction. In both kernel implementations (Listings C.12 and C.13), pointers are used to refer to data in global and, importantly, shared memory. Whilst pointers to global memory are only used a handful of times in the lifetime of the kernel, those to shared memory are used throughout. We altered our kernel to eliminate pointer aliasing where appropriate, and used the `__restrict__` keyword in CUDA which is the equivalent of `restrict` in the C language, introduced in C99*.

Whilst in itself, `__restrict__` is merely an offer of a guarantee to the compiler (that the target pointer will not be aliased in the current context), this permits the compiler to eliminate load/store instructions in between instructions, necessary to guarantee correctness when aliasing pointers may be used. As an isolated example, we considered the kernel shown in Listing 4.5, which has threads subtract a value belonging to a global memory array `c` from values in two other global memory arrays `a` and `b`. As expected the PTX produced for this operation (shown in Listing 4.6) requires the values from `a` and `c` to be loaded, subtracted, and stored back before the values from `b` and `c` are loaded, subtracted, and stored back.

*The `restrict` keyword is not natively supported by the C++ language standard. Compilers that support it provide alternative keywords such as `__restrict__` available in CUDA for the device-side, and `__restrict` for `MSVC` and `GCC` for the host-side. We note that, as the behaviour is implementation-defined, the exact effect is compiler-specific.

```
1  __global__ void no_restrict_kern(int* a, int* b, int* c) {
2   const unsigned idx = blockIdx.x * blockDim.x + threadIdx.x;
3   a[idx] -= c[idx];
4   b[idx] -= c[idx];
5  }
```

Listing 4.5: Minimal kernel with no aliasing guarantees on parameter pointers.

```
1  ld.param.u64 %rd1, [no_restrict_kern(int*, int*, int*)_param_0];
2  ld.param.u64 %rd2, [no_restrict_kern(int*, int*, int*)_param_1];
3  ld.param.u64 %rd3, [no_restrict_kern(int*, int*, int*)_param_2];
4  cvta.to.global.u64   %rd4, %rd2;
5  cvta.to.global.u64   %rd5, %rd1;
6  cvta.to.global.u64   %rd6, %rd3;
7  mov.u32   %r1, %ctaid.x;
8  mov.u32   %r2, %ntid.x;
9  mov.u32   %r3, %tid.x;
10 mad.lo.s32   %r4, %r1, %r2, %r3;
11 mul.wide.u32 %rd7, %r4, 4;
12 add.s64   %rd8, %rd6, %rd7;
13 ld.global.u32   %r5, [%rd8]; //ld.global.nc.u32 with __restrict__
14 add.s64   %rd9, %rd5, %rd7;
15 ld.global.u32   %r6, [%rd9];
16 sub.s32   %r7, %r6, %r5;
17 st.global.u32   [%rd9], %r7;
18 ld.global.u32   %r8, [%rd8]; //Removed with __restrict__
19 add.s64   %rd10, %rd4, %rd7;
20 ld.global.u32   %r9, [%rd10];
21 sub.s32   %r10, %r9, %r8;
22 st.global.u32   [%rd10], %r10;
23 ret;
```

Listing 4.6: PTX code emmited from the compilation of the CUDA code in Listing 4.5.

The operation is simplified however, if it is known that pointer `c` does not alias any other, as the need for re-loading its value is eliminated. Marking `c` with the `__restrict__` qualifier results in this effect, by eliminating the global memory load instruction at Line 18 of Listing 4.6*. Besides this typical behaviour seen across compilers, particularly for CUDA, additional optimisations are permissible. Specifically, knowing `c` is a non-aliasing pointer guarantees† its values are not modified throughout the kernel's lifetime and thus there is value in caching the transaction‡ to spare the

*A brief explanation of each PTX instruction can be found in Appendix A.1.

†External modification of this value in global memory during kernel execution is not anticipated.

‡Memory transactions are typically 32-, 64-, or 128-bytes long. In an access pattern such as this values are likely in the same cache line, thus the effects of caching stretch beyond this singular read.

expensive global memory load for some threads in the block. As such, Line 13 is adjusted to a non-coherent cached load `ld.global.nc.u32`. We note that cache coherency in CUDA is currently not possible globally (w.r.t. all SMs) but it is between different cache layers within the SM.

While examining PTX code to establish the impact of `__restrict__` in the register-based kernel, we observed a large number of type conversion instructions (`cvt`). The shared memory-based kernel made use of minimally sized types to represent different data, in an effort to maximise block sizes for the available shared memory. These types were matched by temporary variables despite offering no benefit to register use. Undesirably however, their use required repeated conversions between these types to maintain their respective extension and truncation properties.

Such conversions consume clock cycles, in many cases, unnecessarily. In the latest compute capability versions as of yet ($8.0 \leq cc \leq 9.0$), only 16 such conversions from 32-bit types to smaller ones can take place per clock cycle per SM [89], resulting in performance deterioration, even between threads of the same warp.

A simple change of types in some parts of the high-level code eliminated the need for such instructions and resulted in approximately 20% higher performance for this kernel overall. Whilst not strictly an optimisation, in compute-bound kernels such as this one, performance gains can be made through the removal of unnecessary instructions.

Lastly, as outlined in Section 4.3.3, we elected to make data used in computing the dimensions of blocks and grids in a kernel available at compile time, rendering such calculations statically commutable. Part of the reasoning for this decision was to make use of the `__launch_bounds__()` directive to aid the compiler in resource allocation, especially in terms of registers. However, we observed no effect in register utilisation nor in the emitted PTX code when making use of this directive versus when not. This likely results from an optimal (to the compiler) allocation already having been achieved without this information, yet we keep this directive in code in case it helps future iterations of compiler tooling.

The overall effect of these surface-level optimisations in this instance was a reduction in unnecessary instructions, and gains in performance. Since both kernels in this instance are compute bound with little memory interaction, performance gains can only result from a reduction in instructions as appropriate, and better utilisation of compute resources, as is generally the case for such workloads.

4.3.6.2 Low-level Optimisations for N-Queens Kernels

For our register-based and shared memory-based kernels, we examined a number of hand-tuning opportunities in areas where the output of compiler tooling is perhaps not optimal. Improving in these areas requires hand-crafting PTX code and injecting it into

the output at compile time [68, 5, 90]. We regard such optimisation 'experimental' and have implemented them in a manner such that they can be disabled at compile time. We reiterate that interference with the compiler can result in code tightly coupled to some tooling and hardware versions and must be done carefully as to not disable or override optimisations made by the compiler and preserve result integrity. Despite coupling code to specific hardware and tooling, such optimisations are necessary to increase performance.

Our exploration of such optimisation begun by analysing PTX code produced by the compiler for our register-based kernel where curiously, we observed reluctance by the compiler* in emitting `bmsk` PTX instructions, even when properly suggested. The `bmsk` PTX instruction generates a 32-bit mask (bit vector) with n contiguous set bits starting from index i and higher, and has two modes: `clamp` and `wrap` to cut-off the mask at bit bounds and wrap around respectively.

Looking further, we presented the compiler with an ideal scenario, namely that of a kernel the threads of which construct one such mask each, shown in Listing 4.10. We hypothesised that the rather lengthy set of instructions involved in producing this mask would be replaced by a single `bmsk.clamp.b32` instruction when compiled and provided the compiler with valid assumptions† such as $i+n<32$ and that $n>0$ in an effort to encourage the use of `bmsk` (specifically `clamp` mode) but it was in vain. Instead, the PTX output shown in part under Listing 4.7 was produced which in turn compiled to the SASS instructions shown in Listing 4.8‡, without making use of `bmsk` in either case.

```
...
mov.u32    %r4, -1;
shl.b32    %r5, %r4, %r3;
shl.b32    %r6, %r4, %r1;
not.b32    %r7, %r5;
and.b32    %r8, %r6, %r7;
...
```

Listing 4.7: Part of the PTX code after compining Listing 4.10 for CC 8.0.

```
...
 MOV R0, 0xffffffff
 UIADD3 UR4, UR4, UR5, URZ
 S2R R3, SR_TID.X
 SHF.L.U32 R5, R0,
    ↪ c[0x0][0x168], RZ
 SHF.L.U32 R0, R0, UR4, RZ
 ULDC.64 UR4, c[0x0][0x118]
 LOP3.LUT R5, R5, R0, RZ,
    ↪ 0x30, !PT
...
```

Listing 4.8: SASS produced when compiling instructions in Listing 4.10.

*Tested on some NVCC versions up to 12.3.1 with flags including: `-Xptxas -O3 -arch compute_80 -code sm_80`.

†The `__builtin_assume(p)` intrinsic guarantees to the compiler that the predicate `p` holds in the current scope.

‡A brief reference of SASS instructions and their syntactical structure can be found in Appendix A.2.

```
__global__ void make_mask_optim(unsigned* array, unsigned from,
    ↪ unsigned many) {
 unsigned mask;
 asm("bmsk.clamp.b32 %0, %1, %2;" : "=r"(mask) : "r"(from),
    ↪ "r"(many));
 array[blockIdx.x * blockDim.x + threadIdx.x] = mask;
 }
```

Listing 4.9: Alternative implementation of the kernel in Listing 4.10, hand-tuned to use `bmsk`.

We note that the kernel in Listing 4.10 made use of 8 registers when compiled as opposed to that in Listing 4.9 which required 10. Whilst general conclusions cannot be drawn from that fact, we speculate the compiler's reluctance may be product of a cost-gain analysis on the potential performance gains against factors such as register preservation. Despite no official performance data being available for instructions such as `bmsk`, it is not unreasonable to assume they offer performance benefits or at worse case, perform identically to the discrete bit operation equivalents. It is noted that despite requiring additional registers in this isolated implementation, using `bmsk` as part of a more complex kernel register reuse may hide such requirements, as was our experience when forcibly introducing this instruction in our kernels.

Following the aforementioned optimisation, we examined the PTX code aiming to eliminate unnecessary instructions. During the propagation sweep of `DoubleSweep-Light`, we traverse unpopulated rows and compute the bit word a of available columns for that row in the manner described in Section 4.3.2. We then apply the standard library function `__popc` on a, to compute the number of set bits. If only one bit is set, then the row is unit and a placement should be made in the only available column, which is the index of the set bit. To identify that index, the standard `__ffs` function can be used, which finds the index of the first (least significant)

```
__global__ void make_mask_bitops(unsigned* res, unsigned i,
    ↪ unsigned n) {
 __builtin_assume(i + n < 32);
 __builtin_assume(n > 0);
 unsigned top_bits = -1 << n + i;
 unsigned bottom_bits = ~(-1 << i);
 unsigned mask = ~(top_bits | bottom_bits);
 res[blockIdx.x * blockDim.x + threadIdx.x] = mask;
}
```

Listing 4.10: Simple kernel of which all threads compute a 32-bit bit vector with n set bits starting from bit at index i and higher.

set bit (one-indexed). The PTX code produced by the compiler for the `__ffs` function is shown in Listing 4.11 where a is stored in the register variables `%r1`. Since there is no single instruction to find the first set variable in PTX, the process is performed by first reversing the bits of a from least to most significant (`brev.b32`), then determining the number of left-shifts i needed to bring the most significant set bit to the most significant bit position of the type (`bfind.shiftamt.u32`), and finally add 1 to i.

```
1 brev.b32 %r2, %r1;
2 bfind.shiftamt.u32 %r3, %r2;
3 add.s32 %r4, %r3, 1;
```

Listing 4.11: PTX instructions from the compilation of the library function `__ffs`.

```
1 bfind.u32 %r2, %r1;
```

Listing 4.12: Single-instruction equivalent of the instructions shown in Listing 4.11.

In this instance however, we know that there is exactly one set bit, therefore the reversal of the bits of a can be omitted, along with the offsetting of the resulting index by 1 as that is not useful in our application either. Such factors simplify the PTX instructions needed as shown in Listing 4.12, where we use `bfind.u32` to identify the natural index of the most significant (and only) set bit directly. This improvement is especially important for devices of compute capability 7.0 and currently up to 8.6, where the throughput of 32-bit bit reversal is relatively low at 16 per clock cycle per SM [89].

Continuing on with the removal of redundant instructions, we shifted our focus to the solution accumulation code. Following every propagation sweep of the thread's state, a check is performed to determine if the state has been solved, and if so, the 64-bit solution counter s is incremented by one. The check compares the occupied column tracking bit word e, with a bit mask of N set bits m. If the two are equal, all columns have been populated with queens, and therefore the state is a valid solution. This check takes the form of three PTX instructions shown in Listing 4.13 where e, m and s are stored in the register variable `%r1`, `%r2` and `%rd1` respectively. In this instance, the predicate register variable `%p1` is set to the result of $e == m$ (`setp.eq.s32`) first, and subsequently, a 64-bit register variable `%rd4` is set to either 1 or 0 depending on if `%p1` is true or false respectively (`selp.u64`). Finally, the value of `%rd4` is added to `%rd1`. Although details on the performance impact of 64-bit instructions such as `selp.u64` and `add.s64` is not officially available, it is safe to assume they will be the same or less performant than the respective 32-bit instruction, an assumption reinforced by available micro-benchmarks [2].

```
setp.eq.s32 %p1, %r1, %r2;
selp.u64 %rd4, 1, 0, %p1;
add.s64 %rd1, %rd1, %rd4;
```

Listing 4.13: PTX instructions for per-thread result accumulation.

```
setp.eq.u32 %p1, %r1, %r2;
@%p1 add.u64 %rd1, %d1, 1;
```

Listing 4.14: Optimised set of PTX instructions equivalent to Listing 4.13.

Since incrementation of s occurs only when the predicate register `%p1` is set to true, `selp.u64` can be eliminated and replaced by predicating the subsequent `add.s64` instruction. Predicated instructions do not involve transfer of control but instead, execution flow continues through them only allowing them to modify the surrounding state if the associated predicate is true. With this in mind, we have modified the instructions as shown in Listing 4.14. In tested architectures, this optimisation was reflected in the SASS translation and offered a reduction of approximately four clock cycles in performing this computation due to the removal of `setp.u64`. We highlight the frequency of execution of this optimised sub-routine in the computation, and the significance a small gain such as this can have in the overall computation.

4.3.7 Kernel Applications and Selection

Earlier in this section, a pair of implementations of `DoubleSweep-Light` were presented. Each implementation is better suited for some environments than others, with the choice between them being made manually. Our criterion for this choice is maximising solving performance, achieved by maximising occupancy and minimising any bottlenecks. Frequent fluctuations in GPU hardware specification and capabilities makes automating such decisions especially difficult, which is why we decided against it.

The register-based kernel likely requires higher numbers of registers when compiled than the respective shared memory-based kernel. Registers are by no means a plentiful resource in the GPU, rendering this kernel non-ideal for some past, and potentially, future architectures. Besides the number of available registers one has to consider the partitioning of the register file between threads, the maximal number of registers available to each thread, as well as the instructions each device supports, and their register requirements. For instance, devices with compute capability 6.2 support $2,048$ threads resident on each SM at a time, sharing a register file of $32,000$ registers between them. To achieve maximum thread residency, each thread must use 15 or fewer registers of which two are reserved for reasons discussed in Section 2.3.1. During compilation of the register kernel for this Compute Capability (CC), the compiler* reports 25 registers in use, effectively reducing the number of threads possibly resident on an SM, and potentially reducing performance.

*NVCC version 11.7.64 on a Windows host system.

On the other hand, the requirements of the shared memory-based kernel are significantly easier to discern as they are based on static information (i.e., shared memory availability) and not so much on the heuristics of the compiler. Perhaps the only real limitation of this kernel is the shared memory availability in the SM which should once again be sufficiently large to accommodate the states of enough threads achieve maximal tenancy.

4.4 Experimental Results and Performance Evaluation

In this section we present results for both the register-based and shared memory-based implementations of `DoubleSweep-Light` we obtained using Swansea University's Accelerate-AI cluster [12]. This cluster is comprised of six identical GPU nodes, each of which houses eight NVIDIA A100 GPUs (Ampere architecture). For reasons outlined in Section 4.3.5, our control over the systems (nodes) involved in this cluster was limited to the scheduling of jobs and accumulation of results. Therefore, to obtain results with as little interference from other concurrent jobs, we submitted jobs requiring 8 GPUs, such that they occupy a full node (i.e., no other GPU job can concurrently reside on the same node).

To evaluate the performance of our `DoubleSweep-Light` implementations, we tackled a range of problem sizes $N \in [19, 25]$ using data-centre GPUs in a controlled cluster environment. Figure 4.6 presents the solving time in seconds required for each

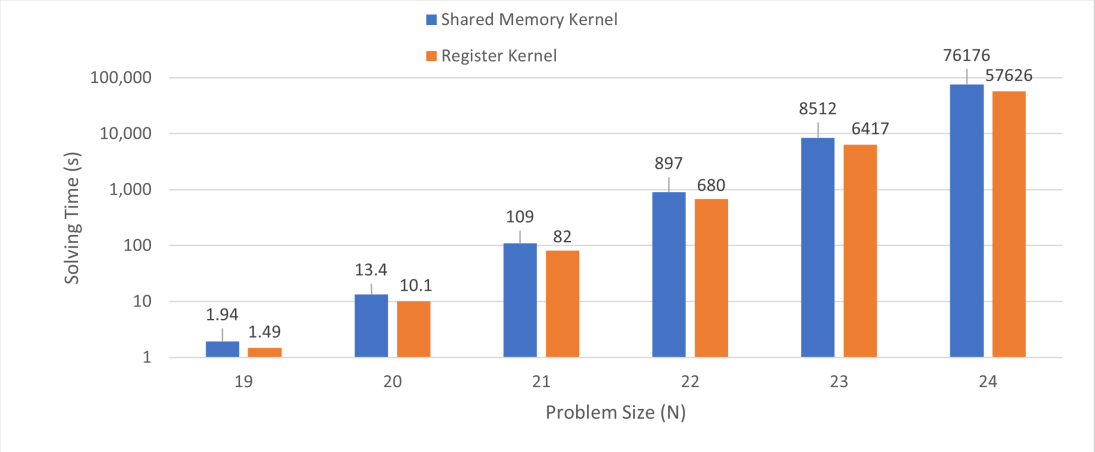


Figure 4.6: Solving time in seconds required to tackle each $N \in [19, 24]$ using both the register-based and shared memory-based kernels over eight A100 GPUs.

N	Number of States	Last Locked Row
19	80,392,450	8
20	22,781,426	7
21	39,430,182	7
22	60,760,010	7
23	9,843,545	7
24	13,335,292	6

Table 4.1: Number of states in state pools generated for our experiments.

$N \in [19, 24]$ using our shared memory-based kernel (Kernel 1) and the register-based kernel (Kernel 2) implementations. For each value of N, a state pool was generated which was subsequently shuffled as described in Section 4.3.1, and instances of each of the two kernels were submitted as separate jobs to the cluster, each tasked with tackling the same state pool. The sizes of pools generated for each experiment are shown in Table 4.1, alongside the index of the last row that was 'locked' to produce this number of initial states. For $N \leq 18$, the runtimes are sub-second which would not offer meaningful data for analysis and is why we elected not to include those here.

For each experiment, we aimed to generate $80,000,000$ initial states, which would equate to $10,000,000$ per device over eight devices. This large factor of over-subscription was chosen to allow for finer control of workload per thread in the devices, however, we note that due to the nature of the problem, the time needed to tackle each, cannot be estimated accurately to employ a better work-balancing heuristic (we refer the reader to Section 4.5.1). Interestingly, in some instances such as $N \in \{20, 23, 24\}$ the number of states generated fell well short of that of the desired state pool size. In these instances, 'locking' and exploring a further row exceeded the limit by a significant amount and the state generator reverted to the earlier pool as described in Section 4.3.1. However, even in these instances, there were sufficient states available to supply all devices, and despite potentially worse balancing of work between and within devices, the overall solving time does not appear to deviate significantly. The two kernels perform well, showing a consistent difference in performance between them. As is to be expected, Kernel 1 is consistently slower than Kernel 2 for the hardware at hand due to the availability of sufficient registers.

Varying the number of devices involved in solving an instance of the problem follows a quasi-linear improvement in overall solving time, as seen in Figure 4.7 which plots solving time against the number of A100 GPUs involved in the computation simultaneously. The speedup, calculated based on these times, can be seen in Figure 4.8. Here, speedup is a factor of solving time using one GPU over solving time using n GPUs. In theory, the speedup should be linear and overlap the ideal speedup line, however, this

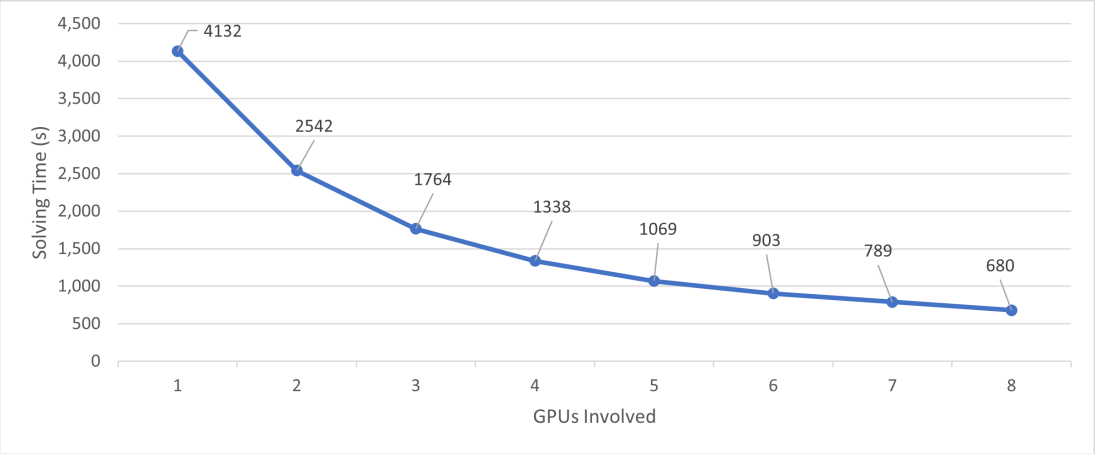


Figure 4.7: Impact to time of varying the number of devices (NVIDIA A100 GPUs) involved in the computation of $N = 22$.

is not the case; we attribute the drop-off to factors such as imperfect work balancing between devices. As explained earlier, it is not possible to accurately determine the solving time required for each state to balance work evenly, therefore it is often the case that some devices complete solving earlier than others and remain idle while the others continue to solve, as was the case in this instance. We aim to investigate this phenomenon further and address such work imbalance (see Section 4.5.1).

Results presented thus far were collected via a single job submission to the cluster that carried out all work. The cluster used for our results however imposes a strict 48-hour limit on jobs to enforce fair resource sharing between users, which was insufficient

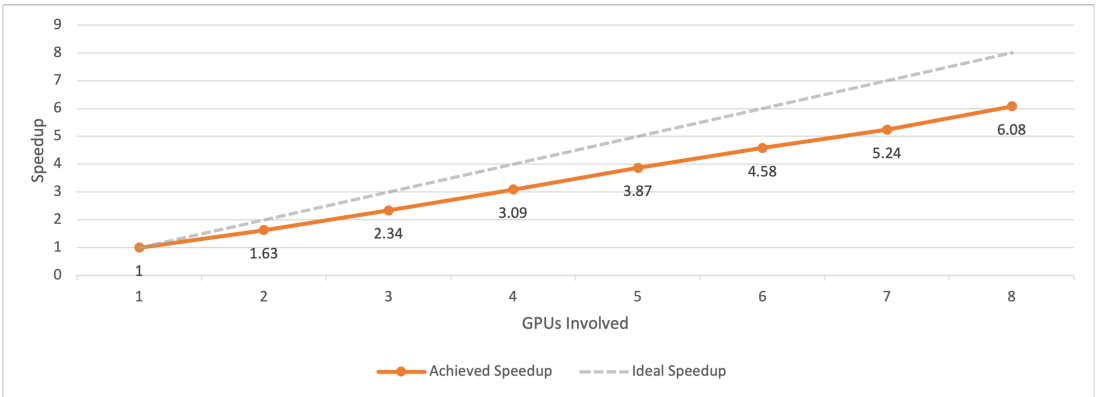


Figure 4.8: Speedup achieved by varying the number of involved devices in the computation of $N = 22$.

time to collect results for larger instances such as $N = 25$. Following the solving time progression we had observed up to that point, whereby the time needed to solve a problem instance n with some solver configuration is approximately 8.35 times that of $n-1$ using the same configuration, we estimated the time necessary to carry out the full computation would be approximately 5.6 days based on the time taken to solve $N = 24$ with eight A100 GPUs. Based on this (very crude) approximation, we generated a large initial state pool of $20,746,561,752$ states by 'locking' up to row 9, and partitioned it into 20 sub-pools. Despite the considerable memory requirements of the state generation process which peaked at 1.75TB of total memory usage*, the overall state generation process lasted approximately 40 minutes.

Subsequently, to tackle these sub-pools of states, we manually scheduled 20 jobs over the course of two weeks, each using eight A100 GPUs and solving one sub-pool of states and completed the computation in a combined $670,747$ seconds of run-time or approximately 1 week and 18 hours. We launched individual jobs in succession with intervals between them to enable better sharing of resources with other users (as the job scheduling algorithm in place schedules jobs in a first come first serve manner with a cap on the number of GPUs each user can utilise). It is worth noting that the combined solving time here is the result of accumulating the time each job required and is a reflection of the total solving time necessary for $N = 25$ on eight A100 GPUs†. The time necessary would however be reduced greatly in a scenario where all 20 jobs execute concurrently over 160 GPUs. In such a scenario, the overall time would be that of the longest-running job, which in this instance was $46,481$ seconds or approximately 13 hours.

Figure 4.9 presents the solving time taken by each job submitted, to solve the respective sub-pool of states. It is noteworthy that during state generation, we elected to not shuffle the pool of states but rather shuffle each individual sub-pool, to save on time and persistent memory input/output operations as it is a resource-demanding process for such large data sets. We did however shuffle each sub-pool in an effort to distribute the workload better between the GPUs involved in tackling that pool, but we accepted that the time each job required to complete would likely not align with other jobs, as is the case. We observe large discrepancies between job completion times, for instance between `Job 2` and `Job 20`, which we attribute to the absence of shuffling of states in the state pool before partitioning into sub-pools, as some initial states will likely have more candidate solutions to be explored dependent on where the first few queens are placed.

Our initial work [101] on this problem comprised of just the shared memory-based ker-

*Using RAM and persistent storage as explained inSection 4.3.1.

†Perhaps with some added overheads sourcing from the multiple kernel launches. This however is an insignificant factor in this instance.

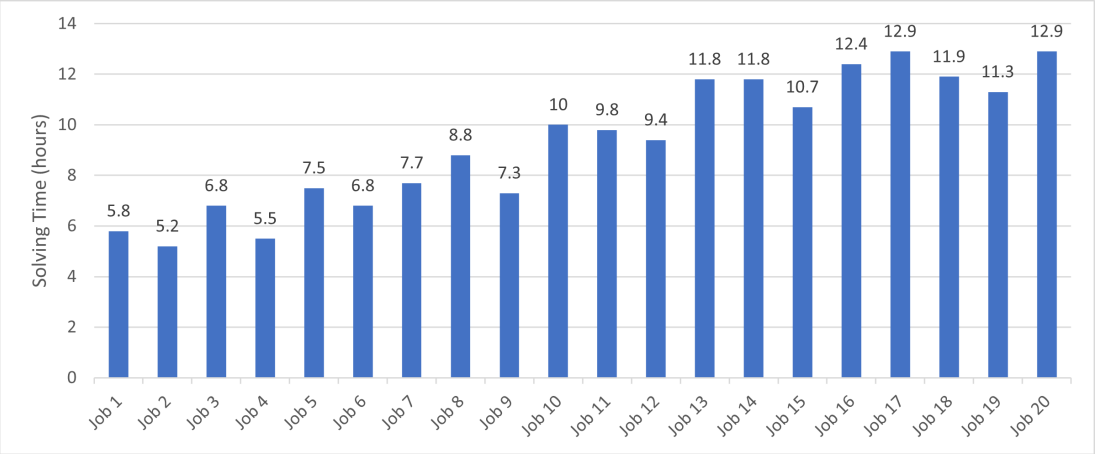


Figure 4.9: Time taken per job, for a total of 20 jobs submitted to solve $N = 25$.

nel as described in Section 4.3.3 and lacked elimination of vertically mirrored states during state generation and the optimisations shown in Section 4.3.6. At the time, we obtained results in the form of runtimes for $N \in [14, 20]$ which can be seen in Table 4.2. These results were collected from two systems housing two GTX 1080ti (Pascal architecture) and one RTX 3090 (Ampere architecture) GPUs respectively, by performing ten runs of our solver over the same input for each test case. We recall these results and contrast them to those presented in this section, as a means of highlighting the effectiveness of the improvements presented, which stretch beyond what is attributable to the different* hardware.

Table 4.2: Solving times in milliseconds from our earlier work [101] for different values of N across a number of benchmark configurations.

N	**1x1080ti** (ms)	**2x1080ti** (ms)	**1x3090** (ms)
14	1.95 ±0.296	0.95 ±0.122	0.93 ±0.024
15	11.49 ±1.038	6.53 ±0.602	5.44 ±0.369
16	138.36 ±0.678	71.17 ±5.255	64.3 ±1.59
17	961 ±21.1	477.2 ±39.2	421 ±2.7
18	6887.8 ±11.5	3439.2 ±6.6	2998.6 ±12.2
19	50445.9 ±190	25354.7 ±196.1	21394.1 ±324.8
20	437200.7 ±1614	213023.4 ±3163.9	176120.9 ±2198.2

*Despite the difference between the RTX 3090 and A100 being somewhat significant, the compute performance of the two for integer operations (which our workload is heavy on) is not as significant as other areas.

Favourably, in our earlier work 32-bit result counters were utilised which made the (frequently repeated) accumulation step in each thread significantly less demanding than the respective 64-bit implementation used currently in our threads. The impact of this additional work however appears to have been mitigated sufficiently by optimisations described in Section 4.3.6.

The N-Queens problem has attracted the attention of the optimisation and parallel processing communities, with several contributions in the literature of GPU-based solvers prior to ours. We highlight that data in literature is collected using different methods and tooling over different hardware, making it difficult to produce an objective comparison. Ideally, such a comparison of approaches would be performed on similar hardware using similar library versions and tooling, and even in a controlled environment such as this, design choices influenced by the current state of the art on hardware architectures would have to be considered. We discuss existing contributions in the field including ours, to highlight existing work and results alongside the potential of our approach.

Our approach differs significantly from what is available in literature, exceeding the performance of previous work using GPUs for N-Queens solving [87, 37, 142, 107, 126], in a naïve computation time comparison. Notably, our approach does not yet include complete elimination of transformations of previously found solutions, from being found in separate search paths. Specifically, we only eliminate symmetries on the vertical axis in the manner described in Section 4.3.1, yet there are several more such transformations to consider. For instance, currently, horizontally symmetric solutions are not eliminated, thus discovered on different search paths. This stands in contrast to work in literature [142, 37] which reduces the search space further by eliminating more transformations. Despite lacking this, rather substantial, reduction however, our approach appears to surpass these results in its present state. We anticipate significant solving performance gains once full symmetry breaking is implemented (see Section 4.5.3) and wish to explore it ideally as a post-processing step following state generation as opposed to an in-kernel operation to not compromise on in-GPU performance.

Beyond GPUs, a number of other computing devices and arrangements have been used to tackle the N-Queens problem. Notably, Kise et al. [69] solved $N = 24$ in 2004 using a cluster of 68 Pentium4 Xeon processors, in approximately 22 days. Similarly, in 2007 Caromel et al. [106] utilised a grid of 260 machines to solve $N = 25$ over the course of 185 days. Both approaches utilise large numbers of CPUs to perform the significant at the time workload associated with each instance of the problem. The scene however, changes in 2009, when Preußer et al. [112] utilise Field Programmable Gate Array (FPGA) devices to design and implement a solver that later solved $N = 26$ in 270 days; work they later complemented [111] in 2017 by computing the result of

$N = 27$. As of yet, this is the largest problem instance solved and remains unverified to date. The fast-paced evolution of alternative computation hardware such as FPGAs and GPUs paired with the shift to massive parallelism shows significant performance potential over 'traditional' CPUs and paves the way for further advances in this and many other fields.

In terms of our approach, using the aforementioned trend in solving time required by our solver for different instances of the N-Queens problem, we estimate crudely that it would take approximately $8.35^2 * 670,747 \approx 46,766,158$ seconds, or just over 1 year and 5 months to verify the solution of $N = 27$ using eight A100 GPUs. This estimate is in line with the time taken to solve the same instance in the work of Preußer et al. [111], who used a total of 14 FPGA boards varying in model for their computation.

4.4.1 Notes on Achieved Kernel Performance

NVIDIA provides tooling to support kernel performance analysis. To gain an understanding of the interactions of our kernels with hardware resources we used the NVIDIA Visual Profiler on a GTX 1080ti device and profiled each kernel separately, when solving the $N = 16$ instance. Choosing this instance was a forced choice, as any long-running kernels accumulate very large volumes of data points the tool cannot handle. Smaller instances (e.g., $N = 14$, $N = 15$) showed progressively worse performance when profiled, the smaller the instance, the worse performance. $N = 16$ appears to have offered sufficient data to allow inferences on kernel performance to be made, however we speculate that larger instances could perhaps offer slightly better results.

The two kernel implementations make significantly different use of available resources, as the results presented earlier in this section serve to show. In terms of resource use, both shared memory-based (Kernel 1) and register-based (Kernel 2) kernels achieve high GPU utilisation as seen in Figures 4.10 and 4.11 respectively. Utilisation is restricted by factors such as unavoidable divergence between threads and can rarely reach 100% with the more complex workloads. As is to be expected, Kernel 1 utilises shared memory very highly at 85% of the theoretical limit, which in turn affects the compute component of the kernel due to the volume of memory load/stores and memory address arithmetic*. In contrast, Kernel 2 halves shared memory utilisation and therefore address arithmetic, which frees up compute time to be used for workload-specific calculations instead.

Better insight in distribution of instruction times can be gathered through the utilisation of different function units, which for Kernels 1 and 2 is given in Figures 4.12 and 4.13 respectively. Single precision units, deal with integer instructions and are highly utilised in both kernels due to the nature of the workload. Interestingly, the

*Which is captured by the 'arithmetic operations' component as it cannot be distinguished.

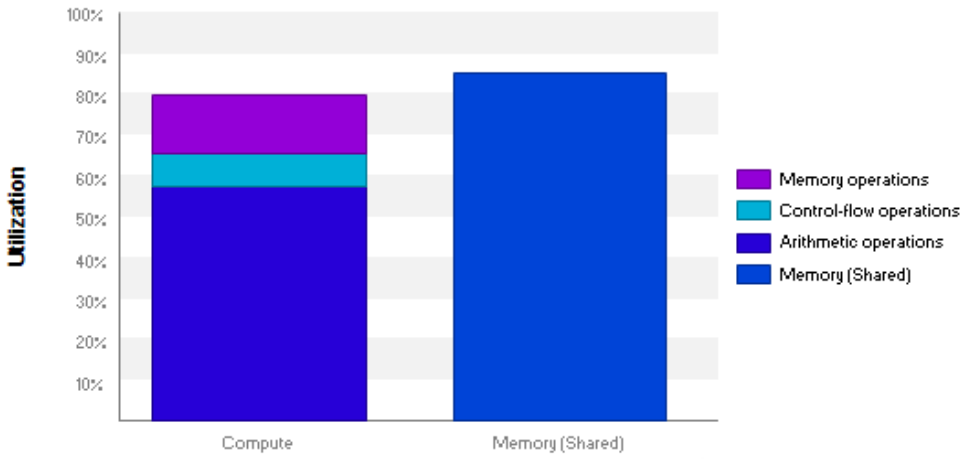


Figure 4.10: Compute and shared memory utilisation by the shared memory-based `DoubleSweep-Light` kernel (Kernel 1).

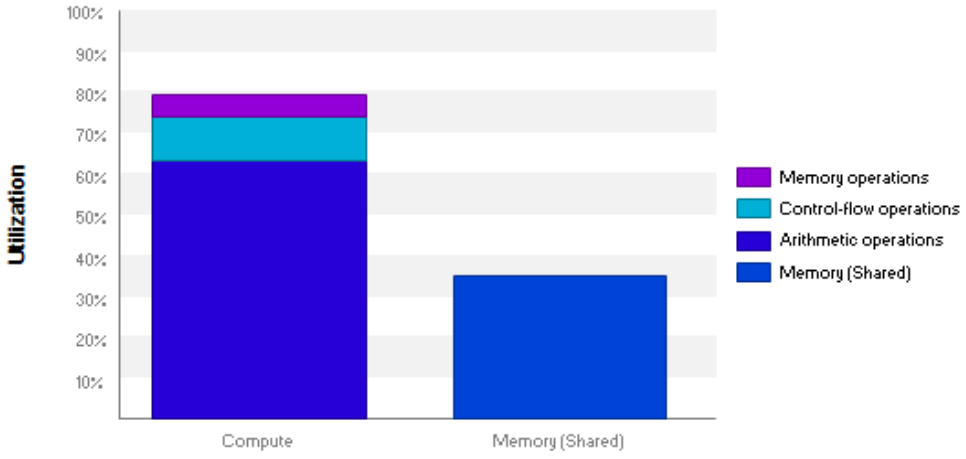


Figure 4.11: Compute and shared memory utilisation by the register-based `DoubleSweep-Light` kernel (Kernel 2).

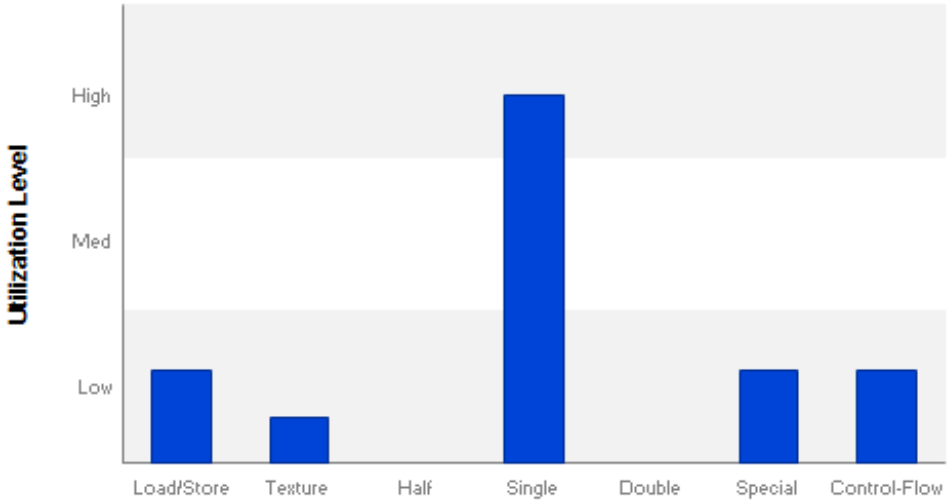


Figure 4.12: Function unit utilisation breakdown for the register-based `DoubleSweep-Light` kernel (Kernel 2).

Special function unit in Kernel 2 is utilised nearly twice as much as Kernel 1. It is unclear exactly which instructions this unit is responsible for beyond transcendental functions as noted in [89], with the relevant white paper [94] not offering insights. We presume this unit might be responsible for functions such as bit reversal, bit position search, and so on, yet we see no reason for the difference in utilisation between kernels and wish to conduct isolated experiments to identify the reason. This may be a bottleneck, especially in architectures with fewer special function units such as Ampere [91]. Lastly, the texture* and control-flow units are utilised equally between kernels, as is to be expected over the same workload.

Profiling also shows that Kernel 2 makes heavier use of registers using 25 registers per thread, compared to 23 for Kernel 1. Interestingly, the difference is somewhat small in this instance even though register spillage did not occur in either compilation. The device used in this instance is of Compute Capability 6.1† which supports two resident blocks per SM, enabled by the relatively low register requirements of each full block ($1,024 \times 25 = 25,600$) placing it well below the half-point ($32,768$) and leaving room for another resident block in the SM.

*Although the name eludes to texture memory and computations, this unit is also used to interact with global and constant memory.

†For CC features, refer to Table 2.1. The kernel code has also been compiled specifically targeting this CC.

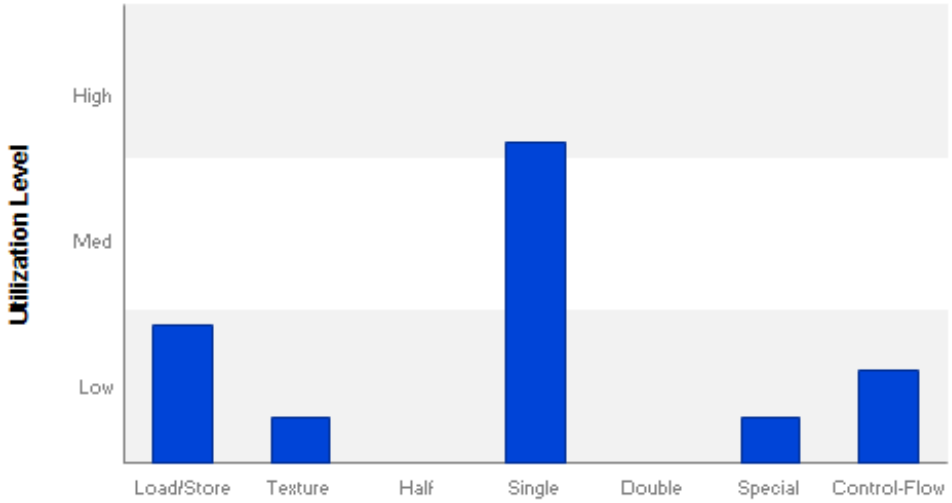


Figure 4.13: Function unit utilisation breakdown for the shared memory-based `DoubleSweep-Light` kernel (Kernel 1).

High occupancy is achieved for both kernels, which is nearly identical between them. Kernel 1 achieves 71.5% with an average of 45.73 active warps per SM coming slightly higher than Kernel 2 with 71.2% and 45.54 active warps on average. We postulate that achieved occupancy will improve once our ongoing work on workload balancing within and between warps is completed (see Section 4.5.1), by reducing the tail effect experienced by warps and by extension, blocks.

4.5 Work in Progress and Future Directions

Whilst our work on N-Queens has resulted in a fast solver for the problem and is in a good stand-alone state, there are a number of areas we have planned and will explore in due course to further the performance of the solver. Our aim through this exploration is to assess the effectiveness of such techniques for compute-bound applications with large amounts of work as, like with most aforementioned optimisations, their application stretches far beyond the problem of N-Queens.

We begin with an ongoing body of work aimed at achieving better work balancing at each level of the hierarchy, from the inner-most warp-level all through to node-level in distributed systems (Section 4.5.1). This is complemented by a secondary work balancing scheme we aim to explore, introducing work stealing between 'live' kernels during computation (Section 4.5.2). Beyond balancing work, we plan to explore means

of reducing the amount work through further breaking of symmetries and envision 'static' implementations for them, as well as a new version of `DoubleSweep-Light` to bringing it closer to `DoubleSweep` without compromising on solving performance.

4.5.1 Interwarp Workload Balancing via State Difficulty Surveying

A likely bottleneck of our long running kernels is that of lone threads running longer than their counterparts in the same warp. This results in a low-throughput warp remaining resident and occupying resources from the SM. Whilst we have not found the pathological case (i.e., that of a thread running significantly longer than the rest) occurring in our experiments thus far, we recognise that time savings can be made by equally balancing work within and between warps in a block, and the possibility of such pathological scenario materialising. We hypothesise that through a computationally inexpensive post-processing step following state generation, we can predict the difficulty of solving each state in the generated pool.

The initial idea for this approach stems from the need for workload balancing amongst GPUs in multi-gpu systems, where we observed the run time of some devices was substantially higher than others. We have conducted initial tests using this approach for such systems and saw potential in this approach which we aim to extend to blocks and warps as described.

To counter that, we have devised a post-processing step to run following state generation, with the aim of surveying the states in the generated state pool and determining the potential difficulty of solving each, in the form of a 'score'. More specifically, a difficulty scoring function $d := S \to \mathbb{Z}$ will be applied to each element of the set of states S in the state pool, and subsequently, states will be distributed to balanced (in difficulty) 'buckets' which in turn map to blocks/warps on the GPU when the pool is tackled.

The function $d(x)$ captures the potential effort required to tackle this state. We intend on expressing this difficulty based on heuristics (and combinations thereof) including:

1. The number of steps (i.e., advancements and backtracks) involved in finding one solution.

2. The number of steps required to advance between each pair of contiguous rows during solving (i.e., the 'difficulty' of transitioning from each row to the next leading up to one solution being found).

3. The ratio of blocked to available cells in unpopulated rows.

Once the predicted difficulty of solving each state is known, states will be distributed to $b = \frac{|S|}{32}$ buckets of equal or near-equal difficulty. As discussed in Sections 4.3.3 and 4.3.4, these kernels require different resources thus the number of threads per block and consequently blocks per grid varies. However, the formulae for determining block sizes are crafted such that we always have full warps (i.e., the number of threads per warp is a multiple of 32) meaning the aforementioned grouping is guaranteed to match up with warps irrespective of block size. Secondarily, we would like to equalise the difficulty between warps in the same block as well, with information on the warps per block, following a similar approach only this time it is a matter of balancing (already balanced) buckets such that each block tackles buckets (near-) equal in difficulty.

Balancing the contents of buckets and subsequently the buckets themselves, is an instance of the balanced number partitioning problem which in itself is a variant of the multi-way number partitioning problem [139]. The decision problem is quoted as belonging to the *NP*-complete class which perhaps renders deterministic solutions a poor choice for the scale of inputs we consider, since our state pools typically consist of many thousands or millions of states. On the other hand, approximations to the problem are known [139].

The aforementioned technique has been applied to partition a large state pool into k buckets of n states, where k is the number of GPUs involved in tackling the pool, with no particular regard for the inter-warp work balance. At present, we have defined the function d as the number of steps required to find one solution for the given state. Our intuition is that this metric determines the difficulty of completion based on the static restrictions of the state (i.e., diagonals and occupation from fixed placements)*. For instance, the states shown in Figures 4.14 and 4.15 appear (visually) fairly similar to one-another, yet the restrictions imposed by existing placements in these states render the former significantly easier to complete versus the latter. Namely the state shown in Figure 4.14 requires one queen placement (advancement) to take place (2^{nd} column of 4^{th} row) before the propagation step of `DoubleSweep-Light` is able to derive all other placements and complete the state. The same cannot be said for the state shown in Figure 4.15 however, which requires 26 advancements with a total of 15 backtracks in the process. With larger (more realistically sized) boards, the difference between 'easy' and 'difficult' states is more pronounced. For such examples we refer the reader to Appendix C.6.

*That is not to say of course this metric correlates to the number of solutions possible stemming from the current state.

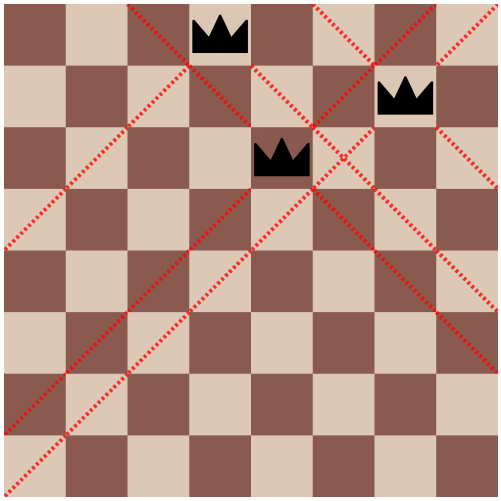


Figure 4.14: Incomplete N-Queens state requiring 1 advancement and no backtracks to complete.

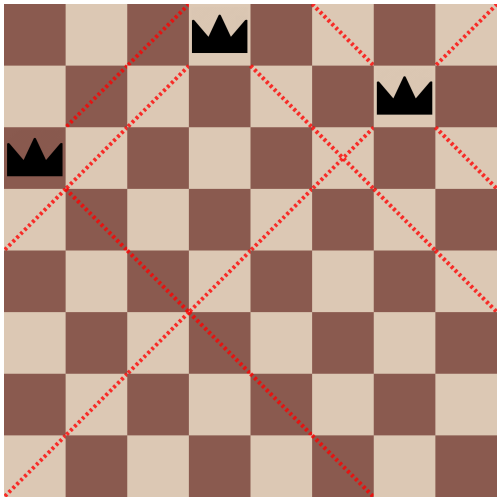


Figure 4.15: Incomplete N-Queens state requiring 26 advancements and 15 backtracks to complete.

Since our metric of choice requires time to compute and in some cases (i.e., states with no possible solution) may lead to a full exploration of the search sub-tree, we set a fixed-point limit on the number of steps. The surveying process is performed on the device-side using a modified implementation of `DoubleSweep-Light`. As a welcome side-effect, the survey process is able to labels each state as solvable, unsolvable, or undecided as part of its assessment. Solvable and unsolvable indicate that the search found a solution, or a contradiction respectively, before the fixed-point limit was reached. An undecided assessment occurs when no solution nor contradiction was identified before the fixed-point limit was reached. With this information, we eliminate unsolvable states, and focus on bucketing the solvable ones as discussed earlier. States for which the search limit was reached are considered 'too difficult' and are randomly partitioned into buckets. These buckets will be part of the same block(s) which we can distribute in the pool as pairs with blocks deemed 'easier' to solve. The intuition for such placement is to target architectures supporting multiple blocks resident on the same SM, where under-utilisation by the warps of the 'difficult' block could perhaps be hidden behind that of the easier block(s)*.

Simulated annealing is employed currently to balance s many states across k buckets of $n = \frac{s}{k}$ states each, configured with the following parameters (chosen empirically):

*This is speculation based on the assumption that block allocation happens sequentially from the grid. Details of the block-to-SM mapping algorithm employed are not documented, and code should be agnostic of the order of execution and SM allocation for blocks. We make this as an educated guess that would otherwise have no adverse effect.

- **Initial Temperature t_{init}:** $10 \times s$
- **Cooled Temperature t_{new}:** $t_{current} \times 0.985$
- **Modifications per Temperature Point m:** 5000
- **Transfer Window Size w:** $\lceil n \times 0.0005 \rceil$

First, k buckets are randomly populated with n states from the pool. At each temperature point during cooling, a pair of distinct buckets (b_1, b_2) is selected and subsequently, a window of size w is slid over each bucket at a random position, effectively selecting w many states from each bucket. The two windows are swapped between the buckets. If the swap was beneficial (i.e., if the difference in difficulty between buckets decreased), the change is preserved. If the swap was not beneficial, and the energy of the system permits* the swap is reverted. This process repeats m many times at each temperature point, before the new (cooled) temperature t_{new} takes the place of the current system temperature.

Since we are working with large buckets in this application, we chose for a number of states to be swapped between buckets proportional to their size, as opposed to swapping one since that would be an insignificant change to the difficulty of the bucket. Constants used here are chosen empirically based on crude testing over randomly shuffled state pools varying in size.

Initial testing of this approach balances states between buckets with a high degree of accuracy, achieving identical or near-identical difficulty scores between them. When these pools are subsequently solved, we observe a better distribution of work between GPUs, yet on occasion we observe worse overall performance (measured in terms of run time) compared to a randomly shuffled pool naively partitioned in k buckets. We speculate this is the result of uneven work within blocks/warps being exaggerated by the shuffling of states and inclusion of the comparatively difficult ones in each bucket. This further motivates this body of future work, aiming to achieve better balancing in the warp and block level before considering outer layers in the hierarchy, namely multi-GPU systems and distributed systems.

4.5.2 State Sharing Among Parallel GPU Solvers

Besides the surveying approach presented in Section 4.5.1, we aim to explore a work-sharing scheme specifically for multi-GPU cluster applications. GPUs in close physical proximity (generally but not necessarily attached to the same host system) can

*This is a random choice where the probability of the change being accepted is proportional to the energy of the system. The higher the energy, the more likely the change is accepted.

typically be interconnected using GPU-to-GPU links such as NVLink [93], and more recently NVSwitch [92] and benefit from superior data transfer speeds among them. Using the latter, GPU-to-GPU communication can be established between up to 256 GPUs [92].

A unified virtual address space can be established between devices set to communicate between them, meaning that one device is able to deference a pointer to the memory of an other connected device and access its memory transparently. With this in mind, one possibility we consider is establishing unified memory between multiple devices, in which live the states the devices (as a whole) must tackle. Each device launches enough thread blocks to tackle the full pool on its own. The intention is for overlap to exist, and states to be consumed on a first come first serve manner. Upon a device beginning to operate on a subset of states on the pool, these states will globally be marked as 'reserved' such that no other device attempts to tackle them. The threads of the block corresponding to that same subset of states from the other devices will terminate early upon examining the flag and exiting without performing any solving.

A drawback in this approach stems from the absence of node-wide synchronisation between devices. To the best of our knowledge, it is currently not possible to employ any form of locking on GPU-to-GPU unified memory (in line with the lock-free principles CUDA enforces). As such, a lock-free data structure will have to be implemented over the unified memory space so that states can be taken by each device without risk of multiple devices tackling the same state set.

4.5.3 Pre-processing for Full Symmetry Elimination

As described in Section 4.3.1, our current state generator is eliminating horizontally symmetric solutions by constructing starting points that cannot lead to horizontally mirrored states (i.e., populating the first $\lceil \frac{N}{2} \rceil$ many cells of the first row). This rather 'cheap' step reduces the search space size by a factor of ≈ 2. This partial symmetry breaking step is performed as part of state generation (i.e., 'statically' w.r.t. the GPU-side computation). As such, it comes at no cost to the performance of our kernels which motivates us to investigate symmetry breaking that is static in the same manner to as high degree as possible.

Dynamically eliminating search paths within the kernel is perhaps more intricate to balance within high throughput kernels such as ours, albeit a possibility. This type of symmetry breaking is referred to as 'dynamic' as it takes place during search. Full symmetry breaking is explored in [44] where the authors propose the generation of constraints (in the form of forbidden cells) during exploration of the search space, such that further exploration will be guided away from exploring symmetries of already explored states. This can offer great advantage in the exploration as, following the

exploration of a state, its symmetric states can be excluded immediately. The main complications arising from such approach in a massively parallel setting relate mainly to memory and communication between workers (threads).

In `DoubleSweep-Light`, where workers independently explore different parts of the search tree (from the same level), the discovery of constraints by one thread would have to be communicated to the rest, to avoid the exploration of symmetric states. Requirements for communication allow for a large array of problems concerning data races and availability of data at the time of checking. Synchronisation between threads cannot be achieved globally (without unsafe assumptions), hence a data race would easily be possible when threads have to be kept up to date with constraints introduced by their peers. Moreover, a global collection of search constraints that grows during search would require a dynamic (likely unknown at the start of solving) amount of memory. This can only be facilitated by global memory, which renders fast checking against this collection impossible. However, the above are not a concrete deterrent from an implementation of this technique. Perhaps by re-structuring the computation and allowing for families of states to be allocated to the same block, we are able to keep constraints block-local. Provided shared memory (the remaining portion) is sufficient for these constraints to be housed, this would become a viable option. Further investigation is required to conclude if this can feasibly offer a reduction in solving time overall.

Partial dynamic state symmetry elimination may also be required as a trade-off between computational constraints and the increased search space in need of exploration. An example thereof, focusing on the horizontal axis, may be by considering the first cell of the lower $\lceil \frac{N}{2} \rceil$ rows 'blocked' in the threads (i.e., never eligible for a queen placement). This can be achieved easily and with minimal impact to the kernel's performance (i.e., without requirement for additional memory nor a significant increase in kernel size), since the column tracking mask can have the corresponding bit set once the branching row reaches or exceeds the mid-point of the board (resp. cleared when backtracking beyond this point) rendering the column ineligible for queen placements.

4.5.4 Closer Alignment of `DoubleSweep-Light` to `DoubleSweep`

Despite our `DoubleSweep-Light` implementation resulting in performance greater than earlier work in literature, undeniably, the `DoubleSweep` procedure is more powerful than `DoubleSweep-Light`. We wish to continue exploring ways to bring more features of `DoubleSweep` into our current, effective approach. Specifically, two important components that are not fully, or at all implemented, at the moment are *centre row branching* and *multi-pass propagation*.

Our initial multi-kernel implementation of `DoubleSweep` presented in Section 4.2.1

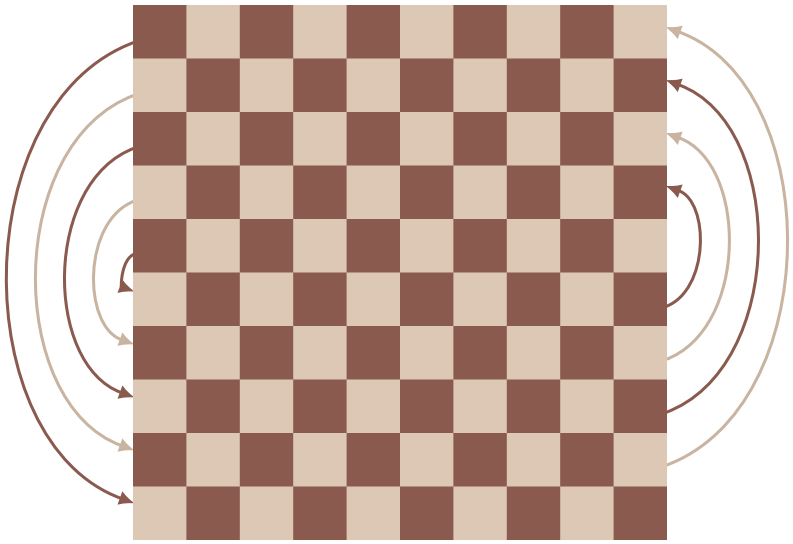


Figure 4.16: Zig-zag branching pattern from the centre of an N-Queens board where $N = 10$.

included branching from the centre row outwards in a zig-zag fashion, as seen in Figure 4.16. A conventional depth counter was tracked (i.e., count of rows visited at the current depth of the search), which in turn was mapped to the corresponding row as if branching from the centre using the below function:

$$branching\_row(c,d) = \begin{cases} c - \frac{d}{2}, & \text{if } c \bmod 2 = 1 \\ c + \frac{d}{2}, & \text{otherwise} \end{cases}$$

Programmatically, we implemented this using predominately bitwise operations as shown in Listing 4.15, to eliminate branching and expensive arithmetic where possible, which in turn compiles to the PTX code shown in Listing 4.16. Whilst perhaps more optimisable, this code is non-divergent. In the expansion of this macro, $(depth \gg 1) + (depth \,\&\, 1)$ quickly calculates $\left\lceil \frac{depth}{2} \right\rceil$* while $(depth \,\&\, 1)\ ?\ 1 : -1$ is a ternary conditional dictating the sign of the earlier calculation by being multiplied with it. Ultimately the combined result of these two components (offset) is added to the centre row index achieving the desired zig-zag branching pattern. Multiplication with arbitrary operands is typically seen as an expensive operation and it appears to carry through the code to the resulting PTX, as seen in Listing 4.16. However, we note

*Care must be taken with the expansion of this and other such macros with signed and unsigned types. Right shift on a signed type is implementation defined. To the best of our knowledge, this behaviour is not explicitly documented for CUDA.

```
#define BRANCHING_ROW(centre_row, depth) ((centre_row) +
    ↪ (((depth) >> 1) + ((depth) & 1)) * (((depth) & 1) ? 1 :
    ↪ -1))
```

Listing 4.15: Directive to map search depth to a zig-zag branching pattern from the board's centre.

```
shr.u32    %r3, %r2, 1;
and.b32    %r4, %r2, 1;
add.s32    %r5, %r3, %r4;
setp.eq.s32  %p1, %r4, 0;
selp.b32   %r6, -1, 1, %p1;
mad.lo.s32   %r7, %r6, %r5, %r1;
```

Listing 4.16: Compiler's output when compiling code from Listing 4.15.

that the multiply-and-add instruction (`mad.lo.s32`) seen here is exploiting hardware support to calculate $\%r6 \times \%r5 + \%r1$ efficiently (quoted in literature [2] as requiring just two cycles to compute in the Ampere microarchitecture). Alternative formulations of this expression that replace multiplication with some other set of operations would be competitive only if the set of operations used is performed in fewer than 2 clock cycles, which we conjecture cannot be achieved.

What lead us to discontinue its use in our new implementation and instead branch from the first locking row, was the implications central branching has to the propagation step. We considered two propagation sweep 'flavours', namely *directional sweep* and *global sweep*. The former was the same principle as our current propagation step, whereby following a placement, propagation applies to the rows below the one which had a queen placed. In the case of central branching, the direction of the sweep changed depending on the direction of branching: going from a higher row index to a lower in branching would perform a sweep toward the zeroth row; branching from lower to higher would perform a sweep travelling towards the highest row index. The latter flavour on the other hand performed a sweep from the state's locked row to the last row of the board irrespective of the row and direction of the last placement.

We found the directional sweep was on one hand relatively straight-forward to facilitate yet due to its restricted nature, propagating just half the board, it offered limited benefit in contrast to a global sweep. The global sweep however, was much harder to facilitate whilst maintaining our low per-thread memory requirements. Queen placements derived through a global sweep had to be marked so, as to not disrupt the order of backtracking. Queen indexes had the most significant bit of their type* used as a flag to indicate whether or not the queen at that index was a derived placement. Backtrack-

*The type was `unsigned char`, and the bit at index 7 was used.

ing from the current depth of the search required any derivations made to be undone to resume to the previous branching state; that incurred a significant cost compared to the current strategy.

Resulting of the added complexity of the propagation sweep and additional operations involved in tracking derivations, we observed consistently poorer performance versus our `DoubleSweep-Light` approach as discussed in Section 4.2.2.

Following the optimisation of our current implementation described in Section 4.3 however, we have not re-visited multi-pass sweeps of the uncompleted region of the board, which constitutes future work. Specifically, we will examine a collaborative, global multi-sweep propagation, performed $k \geq 1$ times in lockstep across all threads in each warp, with inter-warp balloting determining whether or not all threads have made no propagations in their latest sweep, in what case they discontinue propagations before the fixed-point limit is reached. Provided this technique proves at least as effective as our current propagation strategy, we will subsequently explore heuristics to determine a suitable value for k based on the problem instance. Like before, queen placements beyond the natural order of search will have to be tracked as derivations, cleared upon backtracking. This will require us re-visiting our derivation tracking and identify an efficient approach to recall derivations during backtracking whilst preserving a low-memory profile.

4.5.5 Investigate Register Bank Clashes

A lesser known and thus overlooked aspect of GPUs are register bank conflicts. In their work, the authors of [62] explore the physical register layout of the Volta architecture which is comprised of $16,384$ registers per SM partition (processing block). The authors highlight that these registers are split among two physical banks, each of which has two ports, with each port capable of servicing one request per clock cycle.

A register conflict occurs when, an instruction is performed involving data from more than two registers which happen to be in the same bank. This happens as the two ports can only service a request for two register values per clock cycle (one each), thus the third value has to be retrieved in a subsequent cycle. The effect is an increased latency caused by conflicting registers. The authors report gains in performance of up to 15% [143], by minimising such conflicts.

We intend to investigate the effect of such conflicts to our workload by first identifying the register bank structure in more recent architectures (targeting Ampere onwards) and then examining how our workload's instructions and the respective compiler's register allocation influence performance. This is motivated particularly by the register-based kernel which relies heavily on register use, while profiling tools do not give a concrete indication of delays due to register conflicts.

Chapter 5

Insights for Wider and Practical Applications

Contents

Over the course of the the previous two chapters (Chapters 3 and 4) we have explored in detail a range of topics surrounding the use of GPUs to solve hard problems. Observations made in our work have been, up to this point, presented in a task-specific manner. Beyond designing tools such as the N-Queens solver and SAT checker, our aim has been to offer targeted optimisations and recommendations to readers wishing to follow a course similar to ours and use GPUs to solve or accelerate other hard problems.

Generally, the best place to begin an undertaking such as this, is the documentation of the tools that will be used. Studying the documentation and having a reasonably deep

understanding of the tools at hand, such as the hardware, drivers and other utilities involved, is key. As is inevitable however, intricate and specialised hardware such as GPUs has more room for corner cases to occur. Those are impractical, if at all possible, to fully document and offer remedies for in tooling documentation. Discovering and mitigating their effects requires significant effort and time as we have shown in our work. Through this chapter we wish to contribute some guidance, recommendations, and best practices on GPU optimisation to literature.

5.1 Low-Level Tinkering May Appear Scary but is Necessary

In a number of places, our work delves into the depths of GPU hardware and formulates low-level optimisation techniques that encompass the behaviour of hardware, whether officially documented by the manufacturer, or determined through investigation. In modern CPU-side applications, our experience suggests that this, rather pedantic, level of exploration is less necessary. We attribute this to the well matured and competitive compilers that have been written for the well established (and to a degree, simpler) model of parallelism used by CPUs. Arguably, CPUs are not a monolith either; a number of ISAs such as x86, MIPs, ARM, etc. exist. For each of those however, one can find numerous 'competing' and in many cases, open-source compilers, such as GCC [123], CLANG [25] and MSVC [57]. This 'competition' between compilers promotes the collective identification of optimisations and patterns. Unlike programmable GPUs* which are a relatively recent advancement and come with an intricate environment to work with, CPUs and microprocessors have been the core of computing for many years, maturing alongside compiler tooling. Whilst a handful of compilers exist for NVIDIA GPUs, we believe that the fast paced evolution of the still-young GPGPUs does not allow enough time for these compilers to mature fully, which leaves more room for hand-tuning [37, 63], which prevails in a range of high performance applications [5, 68].

Compiler tooling should not be treated as omniscient however. The set of assumptions a compiler can safely make, is a subset of (sometimes equal to) those the author of the code can make. This was the case in our work as explained in Section 4.3.6.2 where instructions emitted by the compiler were unnecessary considering the surrounding code logic. One may be compelled to argue in favour of offering the compiler assumptions (where supported) and guiding it to the optimised assembly, as opposed to taking control and hand-crafting the operation (hence burdening themselves with maintaining it in the future). In the case of NVCC's device compiler and optimiser, support exists for offering these assumptions through intrinsic functions such

*Specifically GPGPUs as described in Section 5.8, and less so programmable shaders on GPUs.

as `__builtin_assume(predicate)`. We encourage and promote this approach *where it succeeds*, yet we have been unsuccessful in efforts of guiding the compiler optimiser with suitable, more complex, assumptions.

```
__global__ void ba_brev(uint32_t* wb, unsigned num){
 __builtin_assume(__popc(num) == 1);
 *wb = __ffs(num);
}
```

Listing 5.1: Minimal kernel testing the effect of `__builtin_assume` on a bit position finding task.

To illustrate this point we present the kernel shown in Listing 5.1, and the associated PTX code emitted by the compiler* shown in Listing 5.2†. Here, the kernel's sole purpose is to write the index of the first set bit of the parameter `num` in a specific global memory address. Writing the result is only done to prevent the optimiser from eliminating a lone call to `__ffs` with no tangible result. We furnished this kernel with `__builtin_assume(__popc(num)== 1)` intended to let the compiler assume there is exactly 1 set bit in `num`. With or without this assumption offered however, the emitted code is the same, and uses `brev.b32` to reverse `num` regardless, before finding the first set bit using `bfind.shiftamt.u32`‡.

```
brev.b32    %r2, %r1;
bfind.shiftamt.u32  %r3, %r2;
add.s32     %r4, %r3, 1;
st.global.u32     [%rd2], %r4;
```

Listing 5.2: PTX code emmitted when compiling the code from Listing 5.1.

```
mov.u32     %r1, 1073741824;
bfind.shiftamt.u32  %r2, %r1;
add.s32     %r3, %r2, 1;
st.global.u32     [%rd2], %r3;
```

Listing 5.3: PTX code emmitted when compiling the code from Listing 5.1 with the assumption that num = 2.

Altering the assumption given to the compiler to `__builtin_assume(num == 1 || num == 2)` which explicitly bounds `num` to be either the value 1 or 2 and implicitly bounds `num` to have exactly 1 set bit, makes no difference to the generated PTX code. Taking this one step further, and setting the assumption to `__builtin_assume(num == 2)` which guarantees that `num` will only ever have the value 2, does result in different PTX code (shown in Listing 5.3), but once again, more is to be desired. Given that `num` will only ever be 2, the code could have been optimised to a simple `st.global.u32 [%rd2], 2;`, however the compiler instead reverses the binary representation of 2 in a

*Compiled using NVCCs version 12.3.1 with flags including: –Xptxas "–O3" –arch compute_80 –code sm_80.

†A description of the functionality of each PTX instruction used can be found in Appendix A.1.

‡The connection between this and the more appropriate sounding `clz` instruction is discussed further in Appendix A.1.1

32-bit type which results in the denary value 1073741824 and performs the unnecessary `bfind.shiftamt.u32` on that constant followed by an `add.s32`, before storing the result.

This observation is particularly interesting given that the assumption offered to the compiler's optimiser did impact the resultant PTX code, which will now always calculate and store the same value (2), yet it has done so in a way that remains easily optimisable and has pragmatically resulted in no performance difference to the earlier discussed general form of this operation. It is also noteworthy that this code translates to SASS without further optimisation.

It is clear through this example and other aspects of our work (e.g., Section 3.3.4) that hand-tuning is not a possibility but rather a requirement to achieve better performance. Another area of potential improvement stems from alternative formulations of the same computation. In our work, and specifically Section 4.3.6, we introduce a predicated instruction to reduce the number of additional instructions (steps) involved in facilitating the default compiler-emitted computation. On one hand, this may appear counter-intuitive as predication can affect instruction flow, however in our case it yields performance gains. The kernel in question is bound by its compute component which is what we reduce through this change. The compiler has no accurate 'feel' of the kernel's resource utilisation at compile time, and can therefore not make inferences leading to these changes.

In some occasions, the compiler will perform an optimisation that is effective, but still leaves room for further improvement. In the course of our work, we had two 32-bit bit words `a` and `b` for which we had to compute the expression $r = a \,\&\, \overline{b}$, view the bits of r through a sliding window mask (`mask`) and count the number of set bits visible through this mask. The CUDA code for this task is shown in an isolated kernel in Listing 5.4.

```
1 __global__ void set_count(unsigned* result, unsigned a, unsigned
    ↪ b, unsigned mask) {
2  *result = __popc((a & ~b) & mask);
3 }
```

Listing 5.4: Isolated kernel to count the set bits of the result of a bitwise expression.

The PTX code produced for Listing 5.4 shown in Listing 5.5 is in essence a translation of the bitwise operations to respective PTX instructions, showing no optimisation has been applied at this stage. What is interesting to observe in this instance, is the SASS code produced from this PTX input, shown in Listing 5.6, where the bitwise operations have been 'stripped away' and replaced by the `ULOP3.LUT` instruction. This instruction has the same purpose as `lop3` in PTX, and runs on the uniform data path. This is a parallel data path to the floating point data paths on the SM, since the Turing microarchitecture, intended to maximise arithmetic throughput [62]. It appears that

priority was given in using the uniform data path in this instance, even though multiple `ULOP3.LUT` instructions have been produced, when in fact only one is necessary.

```
cvta.to.global.u64 %rd2, %rd1;
not.b32    %r4, %r2;
and.b32    %r5, %r4, %r1;
and.b32    %r6, %r5, %r3;
popc.b32   %r7, %r6;
st.global.u32    [%rd2], %r7;
```

Listing 5.5: PTX code emmitted when compiling the code from Listing 5.4.

```
ULDC.64 UR4, c[0x0][0x168]
MOV R2, c[0x0][0x160]
ULOP3.LUT UR4, UR5, UR4,
    ↪ URZ, 0xc, !UPT
MOV R3, c[0x0][0x164]
ULDC UR5, c[0x0][0x170]
ULOP3.LUT UR4, UR4, UR5,
    ↪ URZ, 0xc0, !UPT
POPC R5, UR4
ULDC.64 UR4, c[0x0][0x118]
STG.E [R2.64], R5
```

Listing 5.6: SASS code from compiling the PTX in Listing 5.5.

We intervened and manually inserted a `lop3.b32 %0, %1, %2, %3, 0x20;` instruction in place of the bitwise expression of Listing 5.4 where `%0`, `%1`, `%2`, `%3` matched a result holder variable, `a`, `b` and `mask` respectively. This resulted in the PTX and SASS codes shown in Listings 5.7 and 5.8 respectively being produced. We used the lookup table constant `0x20` here that encodes the truth table for the expression $a \,\&\, \overline{b} \,\&\, mask$. We notice that the uniform data path is no longer used by the produced SASS instructions and only one `LOP3.LUT` instruction is used instead. In practical terms, this resulted in an increase in SM utilisation for our workload (when tested 'hot') of $\approx 3.1\%$ which translated to a reduction in runtime of $\approx 5\%$. We speculate the use of the uniform data path did not offer benefit for our workload that featured no floating point operations, thus rendering this hand crafted optimisation more effective.

```
cvta.to.global.u64 %rd2, %rd1;
lop3.b32 %r1, %r2, %r3, %r4,
    ↪ 0x20;
st.global.u32    [%rd2], %r1;
```

Listing 5.7: PTX code from compiling the tweaked code for Listing 5.4.

```
IMAD.MOV.U32 R1, RZ, RZ,
    ↪ c[0x0][0x28]
IMAD.MOV.U32 R0, RZ, RZ,
    ↪ c[0x0][0x170]
MOV R5, c[0x0][0x168]
IMAD.MOV.U32 R3, RZ, RZ,
    ↪ c[0x0][0x164]
MOV R2, c[0x0][0x160]
ULDC.64 UR4, c[0x0][0x118]
LOP3.LUT R5, R5,
    ↪ c[0x0][0x16c], R0,
    ↪ 0x20, !PT
STG.E [R2.64], R5
```

Listing 5.8: SASS code from compiling the PTX in Listing 5.7.

When engaging in the level of manual intervention such as the aforementioned, there are some rather 'scary' aspects to consider. Beyond the effort required in identifying areas of improvement, the interventions themselves add burden on the shoulders of their author. GPU hardware evolves fast, and so do compilers for it, meaning that a, once effective, hand-tune may be superseded by a 'smarter' compiler or better yet, native hardware support introduced at a later version. For instance, pre-Volta, the PTXs instruction `bmsk` which can construct 32-bit bit masks, did not exist. As such, hand-tuned components embedding PTX assembly in higher-level code would be unaware of this instruction and achieve the same effect using multiple discrete bitwise operations. Compiled code on the other hand, relies on the compiler's knowledge of such instructions in newer architectures and would likely* make use of newer instructions such as these when compiled for higher targets. A certain degree of device-specificity is therefore introduced as a result of these hand-crafted assembly blocks, particularly as the performance of instructions can vary significantly between hardware versions, which a compiler can account for, specifically in later hardware versions. For instance, considering the earlier discussed `brev` instruction, it is documented [89] that devices of CC 6.1 and 6.2 can perform 64 bit reversals per clock cycle per SM, whereas their successors of CC 7.0 onwards† can only perform 16 per clock cycle per SM.

Overall, we believe that accepting the 'risks' associated with using such optimisations is worthwhile for the performance gains there are to be made. We suggest however that restraint is exercised when using them, since completely overtaking the compiler and its optimiser is no viable solution either. Thus, we encourage careful thinking of where it is appropriate to investigate an optimisation of this sort, and consideration not only to present and past architectures, but the potential supersedence of these optimisations by future architectures and what this might entail.

5.2 Branch Flattening Beyond Conventional Unrolling Pays Off

Modern compilers, such as that used by default by NVCC, are generally good at 'flattening' branches where appropriate. Speculative execution in the CPU can reduce the necessity for this optimisation as branch predictors are good at identifying patterns of execution dynamically [114]. Even the simpler static predictors offer performance benefits especially for frequent branching (e.g., loops).

GPUs do not feature any kind of branch prediction‡, neither static nor dynamic. These

*This is not what our experiments suggest for `bmsk` in particular, as described in Section 4.3.6.2, however may well be the case for other instructions.

†At the time of writing, this includes the latest CC 9.0.

‡At present, up to CC 9.0. We believe this is unlikely to change in future versions.

devices are not general purpose as often portrayed (see discussion in Section 5.9) and are intended for a certain 'style' of computation to be run on them (i.e., data parallel tasks), which are generally expected to require light branching. With the absence of predictors, branching can be expensive from the get-go on GPUs and it becomes a lot more costly when threads within the same warp diverge in the branching paths they take, as detailed in Section 2.3.1.

The GPU compiler may put more effort in applying some static branch prediction at compile time, and re-order conditions for branches in their more likely order (e.g., a loop's condition is more likely to be true than false, in the general case). On occasion, the compiler may* flatten branches, or in other words, replace branching code with a set of instructions resulting in the same change of state without conditional jumps. A simple example is shown in Listing 5.9 with the respective compiled PTX code shown in Listing 5.10. In its output, the compiler determines if `b` is even using a bitwise `and` operation and sets the predicate register `%p1` accordingly in the process (`setp`). Based on the truth value of `%p1`, the compiler decides the amount of left shifts required for either side of the branch. This number is either 1 (to multiply by 2) or 2 (to multiply by 4) left shifts.

```
1 __global__ void tk(int* a,
    ↪ int b, int c) {
2  if (b % 2 == 0)
3     *a = c * 2;
4  else
5     *a = c * 4;
6 }
```

Listing 5.9: Simple example of a branch with multiplication operations on either side.

```
1 cvta.to.global.u64 %rd2, %rd1;
2 and.b32   %r3, %r1, 1;
3 setp.eq.b32  %p1, %r3, 1;
4 selp.b32  %r4, 2, 1, %p1;
5 shl.b32   %r5, %r2, %r4;
6 st.global.u32   [%rd2], %r5;
```

Listing 5.10: PTX code emmited by the compiler for the code shown in Listing 5.9.

In the case of Listing 5.9, the branch can also be flattened manually by performing a similar computation, with the same effect, instead of the branch itself, namely that shown in Listing 5.11. Here, `b & 1` evaluates to 1 if and only if `b` is even and 0 otherwise. The integer 1 is then shifted left 1 or 0 times depending upon the result of the previous expression. The result of this shift which is either 1 or 2, is then used as the number of positions to left-shift `c` by. This manual flattening operates in a similar manner to the earlier discussed output of the compiler, yet when compiled, produces very different PTX code as shown in Listing 5.12. In the compilers output, unlike before, no predicate registers are used. Instead, the bitwise operators have mapped directly to the respective PTX instructions.

*In our work, we noticed some hesitation from the compiler specifically for more complex branches.

```
__global__ void tk(int* a,
    ↪ int b, int c) {
 *a = c << (1 << (b & 1));
}
```

Listing 5.11: Manual branch flattening, equivalent to Listing 5.9.

```
cvta.to.global.u64    %rd2,
    ↪ %rd1;
and.b32    %r3, %r1, 1;
mov.u32    %r4, 1;
shl.b32    %r5, %r4, %r3;
shl.b32    %r6, %r2, %r5;
st.global.u32    [%rd2], %r6;
```

Listing 5.12: PTX code emmited when compiling the code from Listing 5.11.

The difference between Listings 5.9 and 5.11 in terms of performance (i.e., real-time runtime performance and instruction pipe utilisation) boils down to the impact of `setp` and `selp`, versus a left shift (`shl`). Without in-context benchmarks it is therefore difficult to say which of the two is more performant, however we are inclined to believe that in general the hand-crafted expression will perform better, by examining individual instruction performances. As documented in literature [2] the `setp.ne.s32` instruction* requires 10 clock cycles to return results, whereas arithmetic instructions including bitwise `shl`, `shr` and `and` are documented [89] as taking 4. To the best of our knowledge, no source states the clock cycles required for the data movement instruction `mov` on Ampere, however we deduce that it will be less than or equal to the cost of any cheap arithmetic instruction (e.g., `add.u32`) which could achieve the same result (i.e., adding instead of moving a value to the register in question), thus we assume 2 clock cycles. With this information and some naïve calculations, we deduce that the compiler's flattening of the branch (i.e., Listing 5.10) requires over† 18 clock cycles whereas the manually flattened branch (i.e., Listing 5.12) requires 14‡.

Whilst a 4 clock cycle difference appears rather small, as discussed earlier in our work (see Section 4.3.6.2), such small gains can accumulate especially for compute bound kernels. Clock cycles alone however are not enough to definitively say that manual flattening is more preferable here, as they do not account for pipe utilisation. We are unaware of any documentation on the pipe(s)§ used for the `setp` instruction, however we speculate it is the ALU that handles this instruction, in which case, the two discussed versions (manually crafted, and compiler generated) are no different in pipe utilisation either.

Further exploration of branch flattening offers a testament to the expensive nature of

*Rather, its SASS equivalent, `ISETP.NE.AND`.

†The clock cycles required for `selp` is also not documented for Ampere, and for the sake of example was assumed to be 0 clock cycles.

‡For both snippets, clock cycles required were calculated for instructions between, and excluding, address conversion instructions (`cvta`) and memory store instructions (`st`).

§Plural, in the event the compiler produces different, equivalent SASS for this instruction running on different pipes as may be the case [2].

branching on the GPU and the lengths the compiler will go to avoid it. In some cases, the compiler opts to explicitly evaluate both sides of the branch, and use `selp` to discard one of the computed values whilst keeping the other, based on the value of a predicate register. Had (dynamic) branch prediction existed, perhaps the trade-off between extra, unnecessary computation and branching would not have. For instance, the code shown in Listing 5.13* compiles to the PTX code shown in Listing 5.14. Here, the registers `%r5` and `%r4` contain the result of multiplying by 2 and adding 4 to the value of `c` respectively. Both results are computed, and then register %%r6 is set to either the value of `%r4`, if the register `%r3` holds an odd integer, or the value of `%r5` otherwise. Whilst branch prediction is not expected to have any serious performance effect in a 50-50 branch such as this, perhaps the compiler's efforts on branch flattening and restructuring might have been less aggressive in this case.

```
__global__ void tk(int* a,
    ↪ int b, int c) {
 if (b % 2 == 0)
    *a = c * 2;
 else
    *a = c + 4;
}
```

Listing 5.13: Simple branch performing different arithmetic operations on either side.

```
cvta.to.global.u64 %rd2, %rd1;
and.b32   %r3, %r1, 1;
setp.eq.b32  %p1, %r3, 1;
add.s32   %r4, %r2, 4;
shl.b32   %r5, %r2, 1;
selp.b32  %r6, %r4, %r5, %p1;
st.global.u32   [%rd2], %r6;
```

Listing 5.14: PTX code emmited following the compilation of the code shown in Listing 5.13.

From this analysis it is apparent that compilers are designed very intelligently, but lack the contextual knowledge the programmer has. Flattening branches wherever possible can certainly be beneficial, but may step in the way of the compiler. We therefore suggest that both compiler-generated and manually-crafted alternatives are considered (if indeed they are different) and the more suitable approach is chosen based on the workload at hand. This perhaps applies more to compute-bound kernels versus memory-bound, however the effects of warp divergence can have profound effects for both, thus we recommend that potential divergent branches and any benefits in flattening them are examined either way.

*The difference between Listing 5.13 and Listing 5.9 is the change in operation, performed in one side of the branch, from multiplication to addition.

5.3 Massively Parallel Algorithmic Reasoning can be Unconventional

Strictly speaking, the model of parallelism employed by GPUs is data-level parallelism which is well studied. Likewise, the model often found in CPUs is thread-level parallelism which is as well known. From this viewpoint there is nothing unique about either, and algorithmic design can simply account for one or another (or in many cases, abstract this detail away all together).

Under this reductionist reasoning, there is no purpose to this (or any) work that offers algorithm insights on constrained massively parallel environments such as the GPU, which we believe is shortsighted. For one, a massively parallel environment itself introduces constraints on top of these models, specifically on work distribution and communication, as our work has addressed. Then, additional constraints are introduced by the device(s) implementing this model under a massively parallel environment such as the GPU, whereby fundamental operations such as memory access may incur substantial costs in practice, when done in an unorthodox (to the hardware) way. At the same time, nothing forbids a massively- and data-parallel approach from relying on a random memory access pattern on (sequential) memory items, yet in all likelihood, this will result in poor kernel performance on GPUs. Memory access patterns are not on the more severe side of 'problems' that may be encountered either. As our work has addressed, factors such as warp work balancing, non-divergent warp-level coordination, memory region choices, along with different access patterns and constraints *for each*, instruction pipe over-subscription, global data structures, host-device interactions and many others must also be accounted for.

Admittedly, some of the performance bottlenecks encountered on the GPU relate strictly to and can be addressed by the implementation alone, however, we find it more robust to consider those in algorithmic design rather than try to compensate for them after the fact. We came face-to-face with this problem in our work discussed in Section 3.2, when we tried to implement an existing (admittedly, easily data-parallel) 3-SAT algorithm on the GPU. During this effort, we were faced with the choice of either using shared or global memory for a per-thread call stack (to mimic recursive search), neither of which is a viable option. Using global memory would introduce substantial access costs per thread and, due to the frequency of access, force threads in a long queue of waiting for data to be fetched, as caching would be rather ineffective in this scenario. Using shared memory on the other hand, restricted the available space* enough to make this approach usable only for dummy inputs. Earlier work [83] had implemented this algorithm using a multi-kernel pipeline. This approach shows some

*On architectures of the time. Newer architectures present new opportunities as explained in Section 3.5.

potential (despite potentially facing a range of utilisation related problems) but was not strictly implementing the sequence of steps of the original 3-SAT algorithm.

A great deal of adaptation and heuristic choices have to be made to fill the voids of algorithms designed without consideration of massive data-parallelism, which we believe should form part of the algorithmic design rather than being an after-thought portrayed as implementation detail. To revisit the earlier example of our work on a full on-GPU 3-SAT solver, a heuristic scheme for hybrid use of global and shared memory (once again, enabled by hardware advancements) is being considered (see Section 3.5) which in itself is an algorithm alongside an adapted version of the original algorithm [56] that we are using.

This does not of course go to say algorithms must be developed with specific hardware in mind (i.e., a link back from one potential implementation), overriding all abstraction. What we instead suggest is that more precisely defined algorithms are designed for these devices. This can be done by constructing a better, more informed model of data-parallelism, which accounts for typical constraints of massively parallel environments and specific examples thereof, such as GPUs. The purpose of this kind of model is to bound choices that can and cannot be made while approaching a problem. For instance, under this model, communication will have to be restricted to small clusters of threads (in the case of GPUs, the cluster will likely be of 32 threads or as many as each block will hold). Likewise memory will have to be treated as a multilayer entity with costs associated with using each layer, and capturing the interactions between these layers in order to mandate that any algorithmic choices that have to be made as such are done so in context, and during the design phase. We also stipulate that such a model would be beneficial when transposing existing algorithms, to produce scalable, massively parallel equivalents irrespective of any implementation detail.

We suggest that the following points are included in the formulation of such a model (perhaps a more GPU-oriented one), which is open to further extension.

Abolishing assumptions on thread scheduling and order. The order of thread* execution, simultaneous execution of particular threads, or the state of each thread cannot be relied upon. This is certainly the case for the GPU where no guarantees are offered on any of these aspects of thread execution. At first glance this appears to be relevant purely to the implementation of any algorithm, however, the implications this has, reach far wider. Synchronous or simultaneous global access, waiting, etc. cannot be guaranteed as a side-effect. This is because context switching may result in some (blocked) threads becoming inactive whilst waiting for data, dependent upon other inactive threads resulting in a deadlock, as is the case on the GPU where threads on

*Here, we use the term 'thread' to refer to the smallest unit of execution.

different SMs have no knowledge or guarantee over the state of other threads* in other SMs. Other mechanisms of concurrency are also not possible globally, such as locks and semaphores. The model here should account for this, and led to the creation of wait- and lock-free algorithms rather than attempting to circumvent these limitations.

Clustering thread communication. Communication between threads can be facilitated, however the environment imposes some restrictions. For the same reasons described in the earlier point, global communication between threads (particularly bi-directional, or involving timed data exchange) is not recommended, nor possible some times. This does not go to say that threads must not communicate. To the contrary, performance can be leveraged from thread collaboration for instance in a parallel reduction operation, but this should be done in localised clusters of threads in accordance with the surrounding environment. On the GPU this should be done between threads that are part of same warp or, maybe less preferably, block.

Treating memory as a multi-layer fabric. Often memory is treated as a monolith in algorithmic design, which we believe abstracts too much detail. Memory is a complex entity in massively parallel environments as a whole, and particularly the GPU. We view it as multi-layer fabric† where each layer has a range of benefits, drawbacks, and special requirements. Take for example the global memory layer which is plentiful but costly to access, especially when a non-ideal access pattern is used. Similarly, the shared memory layer, which is on-chip, is much cheaper to access but beyond size restrictions, a plethora of access-related constraints exist. For instance, bank conflicts should be minimised which impacts the order of access by threads and/or the structure and distribution of data. Paired with strict size limits‡, this can introduce memory management schemes in coordination with threads in the same warp.

Limiting divergent branches. Conditional branching (repeated or not) can be detrimental in environments such as the GPU, where lock-step execution requirements are imposed. Placing restrictions upon conditional branching and iteration in algorithmic design allows for a more direct implementation later on, without need for ad-hoc 'hot-fixes' to the algorithm. Beyond lockstep execution, one has to consider data availability specifically for data involved in the branching condition. Typically, this condition comes in the form of a predicate encompassing multiple data elements which may or may not be available (e.g., results from other threads) or easily accessible (e.g., residing in far-away/costly memory regions with special access requirements). The availability and locality of this data must be considered for, perhaps requiring coordination or access techniques to manage effectively.

*A recent change in the Hopper microarchitecture [92], introduces block clusters with this intention, but that does not make global synchronisation possible nor desirable.

†The term 'fabric' here is not intended to alias fabric computing, although in some regards it could.

‡Relaxed in more recent CCs but nevertheless still quite restrictive.

Befriending context switching. In 'conventional' threading, frequent context switching is often treated as a costly compromise, particularly for performance-critical applications. On platforms such as the GPU however, the incarnation of a thread is significantly 'lighter' (generally speaking) than that on the CPU. This is because on the GPU, threads are logically grouped into blocks, which in turn 'reside' on a Streaming Multiprocessor. Blocks become resident iff sufficient resources are available for all their threads on the SM. Residency is no guarantee of simultaneous execution of threads, as each block is partitioned to warps which in turn are 'given attention' by the SM, through their respective warp schedulers which are responsible for choosing which warp is next to execute an instruction. Context switching therefore comes at no cost, as practically, no cache lines are flushed, no program counter and registers are being stashed/loaded, and so fourth, unlike the typical implementation on the CPU. Instead, the warp scheduler switches between ready-to-run threads, attempting to maximise utilisation (i.e., least idling by the partition of warps the scheduler manages). Context switching can therefore be 'friendly', particularly in hiding latencies, as threads left waiting for some resource to become available (e.g., fetching from memory, waiting for results from an instruction etc.) can do so whilst some of their peers in the block are utilising resources. Algorithmic design should be mindful of this fact in that memory- and compute-heavy tasks are likely to get in the way of efficient switching for utilisation maximisation.

5.4 Optimisations on the Implementation May Exist in Unexpected Ways

In complex environments such as the GPU a multitude of factors affect performance (especially when that is utilisation and/or execution time). As discussed further in Section 5.9, these devices are (at present) quite specific about the order and manner in which things must happen to achieve good speedups and performance, sometimes in a counter-intuitive fashion. For instance, one might think that the superior read/write performance of on-chip (i.e., shared) memory, makes it an unequivocal choice over direct fetches/writes to the 'far away' global memory. One of our explorations however, highlighted an instance where this is not the case; direct global memory transactions performed better (in terms of utilisation and execution time) than shared memory.

Specifically, we had constructed a kernel which, broadly, aimed at detecting the presence of a specific set of data a among a large set of equally sized sets $K = \{k_1, k_2, k_3, \ldots k_{n-1}, k_n\}$. Each block in the kernel has a 1:1 mapping to a set of data k_i, and the purpose of the block is to compare that set to a, and write (to a given location in global memory) whether or not the two matched. We had allocated shared memory to transfer a into, after which point threads within the block would start fetch-

ing data from their assigned k_i, and checking them against a, accessing global and shared memory with a sequential access pattern. The idea here was that a could be accessed efficiently at the same time as k_i as they resided in different memory regions, and warps could therefore maintain sequential access patterns for both. To our surprise however, the kernel underutilised the SM and took longer to execute, versus an identical kernel which does not use shared memory and instead has threads read their respective elements of a and k_i directly from global memory as they check them.

This task is naturally morphed around most GPU requirements, meaning that lockstep execution, sequential access, etc. are rather simple to achieve, and as such features a very limited and simple compute component. Reading data into shared memory from global memory took more time than the cumulative of accessing them at the point they were required for checking. Since threads are executing in lockstep, sequential global memory reads within a warp are performed as a single transaction and the over-subscription of the SM with active warps greatly aids latency hiding. Intuitively, reading some data to shared memory *as a block* that would later be accessed by each warp as and when necessary appears more preferable than each thread (of any warp) reading its own data from global memory, however as experience shows, this was not the case here*.

We complemented this implementation with some cache hints to the compiler (i.e., writing the result through, whilst caching a on both L2 and L1 as all threads depend on it) which also resulted in performance gains. This example is one of a few we identified upon which support that what is sometimes intuitive, may in the end, not behave as expected. The complexity of these devices is high, which makes general 'catch-all' statements and techniques hard to justify.

5.5 Define Metrics for Performance First

Throughout our work we have been using the term 'performance' to be measured either as execution time, or instruction/memory throughput for our kernels. These metrics are commonly used in HPC applications and particularly in the field of GPU computing. What counts as performance however is relative to the application in question; for datacentre applications performance may be measured as a product of response time (i.e., execution time) and energy consumption for instance.

In general, this term encapsulates one or more important (to the application) metrics, and the relation between them. Gaining performance is achieved by increasing or decreasing one or more of the chosen metrics, however once again this is relative to the

*Our experimentation on this was performed on an RTX 4090 GPU with data ranging from few small sets within K, to those reaching the 24GiB device memory limit, either with large sets in a smaller set K or vise-versa.

application. To use the datacentre example once more, reducing power consumption alone may be desirable, even if response time increases slightly, as one may be weighed heavier than the other.

We treat performance loosely as a scoring function, with our aim to optimise its result, and recommend that metrics involved in this 'function' are clearly defined before any attempts at optimisation are made. As previously detailed in our work, optimisations can be costly both in development time and future compatibility, therefore we strongly recommend that performance metrics are clearly defined before any attempt to optimise is made. Some metrics intertwine with others and the whole set has to be considered in performance evaluation, as we found from experience.

In particular, whilst evaluating the performance (measured in execution time and SM utilisation) of our optimisations discussed in Section 4.3.6.2, we did so under the assumption that SM utilisation loosely correlates to solving performance. We soon came to the realisation however that SM utilisation relates/is comprised of other metrics such as active warps per partition, and work distribution across SM partitions which more accurately captured the impact (or in many cases, lack thereof) of our optimisations. Some other metrics to consider in this situation (perhaps more applicable to compute-bound kernels) is SM instruction pipe utilisation and branch divergence which can set a ceiling to overall SM utilisation.

Regarding our earlier work described in Section 3.3, we explored a different definition of performance which was initially a problem-specific metric: Checks Per Second (CPS). This metric is determined by the execution time and the size of the solved problem, however this captured little meaningful data on the effect of micro-optimisations, as general, minute time fluctuations due to uncontrolled factors (e.g., system load) would result in CPS fluctuations anyway. As the kernel is bound by memory interactions, we amended the performance metrics used to assess the effectiveness of optimisations to include L1/L2 cache hit rate, eligible warps and issuing warps per scheduler, and instruction pipe utilisation. We deemed it unnecessary to include the 'catch-all' global memory throughput metric here, as our optimisations were not targeting load/stores from off-chip memory, nor the access patterns involved. However, this metric as well as others relevant to off-chip memory interactions (e.g., request sizes and transaction count, overall utilisation, etc.) may be useful to other memory-bound applications.

5.6 Evaluation of Optimisations is Multidimensional and Domain Specific

In line with the theme that emerges from this chapter and our work, as well as the work of others [142, 108, 83, 37], a great deal of domain-specificity prevails in the realm of high performance GPU programming. It is therefore crucial to consider this when evaluating optimisations.

Optimisations should not be thought of as binary whereby they are either effective or ineffective, but in a rather more granular, analog way. The effect of optimisations may be beneficial for all performance metrics chosen, or perhaps benefiting some performance metrics whilst not impacting, or even worsening others. In either case, the effect of some may be greater than others. It is therefore difficult to produce a definitive evaluation technique for all cases.

Assessing the effectiveness of an optimisation is relative to the application in question and should ideally be done 'hot', or in other words, as part of the context in which it is intended to live. In our register-based kernel detailed in Section 4.3.4, we found that some hand-crafted PTX snippets we tried in pursuit of greater performance, had adverse effects to register utilisation for instance. Without context, one is unable to determine how well the compiler will be able to re-use registers for instance, or what side-effects may be introduced, such as pipe oversubscription, warp stalls, etc. Specifically for hand tuning optimisations, we believe that the compiler and its optimiser(s) should be treated as assistants rather than the enemy. We initially found ourselves 'fighting' the compiler whilst tinkering with manual adjustments to PTX and in particular those described in Section 4.3.6. We believed the reordering of instructions by the compiler to be detrimental to performance, only to ultimately come to the realisation there was nothing inherently 'bad' about the way it was done, and in hindsight, the compiler probably knows more than us on this one.

There is value in testing the effects of an optimisation in isolation as well, to gain insight on how it is treated by the compiler. In our initial journey through the N-Queens problem, we isolated components of our implementation of `DoubleSweep-Light` and explored their compilation in isolation using the GodBolt Compiler Explorer [47]. This was what led us to realise that the compiler intrinsic `__ffs` compiles to a pair of PTX instructions, namely `brev` followed by `bfind`, as no single instruction can implement this functionality. That was no cause for concern, however it led to the observation that one of the instructions, `brev`, is redundant when it is known that the 32-bit bit vector provided is comprised of exactly one set bit. We also constructed and most crucially, tested most hand-crafted optimisations presented in this work in isolation, to ensure their correct function and even gather some crude performance results versus alternatives, before examining them in context.

5.7 There is Future in GPU Acceleration

With all discussion on the necessary efforts for development of optimised software for GPUs, one may feel dissuaded from using such intricate technology. The intention of this work is by no means to underline the complexity of the devices and thus the complexity of making efficient use of them. Our work is aimed as a contribution to the existing pool of knowledge for efficient use of such devices, and simplify the efforts of others. We believe that the strong drive for GPU utilisation goes hand-in-hand with the evolutionary 'boom' these devices have seen within the last decade, and we have no reason to believe this will cease in the foreseeable future, especially given the current (financial) investment in these devices by cloud providers, and the increasing number of applications dependent on them.

One may rightly wonder why we chose to focus our efforts on GPUs and not other specialised processor technology (e.g., FPGAs). We do not believe that a general and well founded argument can be made in favour of one versus the other, that would be applicable to most or all cases. We do not offer absolute arguments against either (GPUs, nor FPGAs) or others, however, we wish to present some food for thought from our point of view.

Our choice of GPUs was in part driven by the wide availability of these devices, even for consumers. It appears that almost all mid- to high-end modern desktops and even laptops are equipped with a GPUs, capable of being programmed to performing non-graphics tasks. Techniques for optimised utilisation of GPUs are therefore applicable to a wide audience and set of applications, from large scientific computations, down to every-day applications such as data processing. Simultaneously, the availability of these devices has facilitated a rapidly expanding set of applications that support them, which may have been the reason NVIDIA designed and produced a plethora of data centre oriented GPUs* since, such as the Tesla and Quadro ranges and more recently, H100.

At the same time, FPGAs are reconfigurable hardware, or in other words, a set of building blocks (typically logic gates, but sometimes more complex blocks such as adders etc.) that can be configured to encode circuits performing certain computations. FPGAs remain at present quite specialised with support for configuration tools being provided by each manufacturer. Likewise their area of application is restricted by the circuits constructable using the available building blocks in each. In the course of this work we looked into examples [112, 111, 138] of FPGA use for the N-Queens and SAT problems which we are also exploring. In the case of both problems, the algorithms being implemented are rather simple, which we speculate is the product of the capabilities of these devices paired with the difficulty in programming them. This

*And more recently, other hardware.

of course is not to say that good results (in the case of N-Queens, similar results to ours, see Section 4.4) are not achieved.

Modern data centre GPUs allow for concurrent kernel execution, even in multi-user environments with inbuilt data isolation. To the best of our knowledge, FPGA hardware cannot be used by multiple users simultaneously or switch hardware configuration as quickly as a GPU can switch between kernels. This may be the reason FPGAs are not adopted as widely as GPUs in cloud computing platforms. Likewise GPU, and particularly those intended for data centre use, offer data integrity guarantees and error correction which would have to be manually implemented on FPGAs.

The investment in and state of GPU technology as well as its continuous (and fast paced) evolution may drive or be driven by its wide adoption in, presently, significant applications such as AI and LLMs, HPC, etc. This leads us to believe this is not the product of 'temporary enthusiasm' and in all likelihood will continue to be part of modern day computing.

5.8 Reducing Turing Tax is Non-Trivial

In the words of Prof. Paul Kelly from his keynote in HeteroPar 2022:

> *"Turing tax is the price you pay for running on a general-purpose computer rather than a specialised one."*

We believe that the notion of Turing tax accurately captures the "overhead of universality" as is intended, but we note the applicability of this term on various levels of abstraction, whether it be on the fundamental computational model level, the hardware, or higher levels. We have come to conclude such overheads (i.e., performance tax) are very much applicable to General Purpose Graphics Processing Units which after all are labelled general-purpose devices*, or in other words, attempt to shift from special-purpose processors to more general coprocessors.

In this work we examine ways to reduce Turing tax for devices (i.e., GPUs) which are programmable and follow the Von Neumann architecture yet on some operation seem to get close to the performance of special-purpose hardware. Due to the model of computation used by these devices, it is therefore not possible to completely alleviate Turing tax. From the perspective of the computation in need of doing and how this computation is expressed under this model, we are able to offer a reduction or 'tax break' in this overhead in various places. We argue that the (considerable) effort this takes is worth-while for a number of reasons listed below.

*Perhaps misleadingly, further discussion on this point follows in Section 5.9.

For one, better performance is generally not unwelcome in any application, whatever performance is quantified as in that application (see Section 5.5). For all metrics which directly or indirectly relate to efficient utilisation of GPU hardware techniques such as those presented in this work are useful. Such techniques take considerable effort to identify and trial, often requiring some 'guesswork' on the complex and undocumented behaviour of hardware and tooling. Following their identification however, these techniques remain somewhat low-maintenance for future (hardware) architectures, even if their effectiveness fades (in relation to the state and capabilities of hardware and tooling) over time. Furthermore, techniques for optimising certain aspects of performance such as those presented in this work are not strictly application-bound. They instead revolve around certain fundamental operations which can be applied out-of-the-box in other areas. This is to say that once the effort and time has been invested into these operations, a wide range of applications may benefit. Lists of documented application-unspecific optimisations have existed for years and been used extensively in the domain of graphics processor programming and beyond. As an example we refer the reader to [6].

5.9 There are Only Few Things General about General Purpose Graphics Processing Units

Whilst we are not the first to come to this realisation [105], our work contributes in gaining an understanding of the (often times, unclear and variable) limitations of the so called General Purpose Graphics Processing Unit. Historically, GPGPUs have made leaps in broadening their applicability to different domains beyond computer graphics, whilst still offering superior performance for them compared to the respective state of the art CPUs. In part, GPUs exploit a very different model of parallelism to conventional CPUs, which is simply better suited for some applications by nature. For instance, large-scale matrix arithmetic becomes trivial under an uncoupled, massively parallel model, such as that of the GPU. The model of parallelism itself however is not to be thought as "simply better" than conventional threading employed by CPUs, as it bars from existence many constructs of concurrency aimed at global concurrent access (e.g., locking, semaphores, thread waiting), which are necessary in tasks involving for example, network client management (which itself is not possible to achieve from the GPU, as we discuss later in this section).

In our work however, we have been using the terms GPGPU and GPU interchangeably, to refer to PCI-connected graphics coprocessors that attach on a host system. A distinction between the two has to be made and lies in the fact the former refers to a programmable device with some level of support for such programming and open to a broader range of tasks, whereas the latter may have no such support. In other

words, the term GPGPU is a specialised instance of GPU. Modern day GPUs are all programmable* and it appears to us that this is the trajectory of their evolution: remain well suited for graphics tasks, and open to other areas.

Our exploration has visited a wide window of Compute Capability versions of NVIDIA GPUs both consumer- and datacenter-oriented, that includes the Pascal, Volta, Turing, Ampere and Ada Lovelace architectures, followed by a review of and steps in preparation for the latest, Hopper architecture. Through this exploration it has been made clear to us that successive hardware revisions add new or evolve existing features, typically highly specialised, which are not seen on current CPUs, and on occasion reduce support for others. For a device seemingly designed for 'general purpose' use, it is rather odd that such specific and highly specialised features are embedded on the die where space is constrained and proximity is important. Implementation details (inevitably influencing design choices) do fluctuate as well, on occasion invalidating assumptions from one CCs version to the next. This was the case with the introduction of independent thread scheduling as briefly discussed in Section 2.3.1, where the assumption that threads (necessarily) execute in lockstep was invalidated. This was not explicitly documented as a safe assumption to make, albeit documentation allowed for the inference to be made. Moreover, the effect of the `__syncthreads()` primitive which has historically and consistently been documented as imposing a *block-wide* thread fence, changes from the Volta microarchitecture onwards. To quote the CUDAs programming guide [89] directly:

> *Although `__syncthreads()` has been consistently documented as synchronizing all threads in the thread block, Pascal and prior architectures could only enforce synchronization at the warp level. In certain cases, this allowed a barrier to succeed without being executed by every thread as long as at least some thread in every warp reached the barrier. Starting with Volta, the CUDA built-in `__syncthreads()` and PTX instruction `bar.sync` (and their derivatives) are enforced per thread and thus will not succeed until reached by all non-exited threads in the block. Code exploiting the previous behavior will likely deadlock and must be modified to ensure that all non-exited threads reach the barrier.*

This may result in code written for pre-Volta devices to deadlock in devices implementing Volta or later†, and forms part of a number of such incompatibilities that may render code non-operational or unsafe for some later hardware.

*Referring to NVIDIA devices, perhaps with the exception of some low-/medium-range embedded GPUs.

†An interesting and thorough explanation of deadlock situations can be found in [9].

GPUs typically boast speedups on standard performance metrics such as IOPS or FLOPS, orders of magnitude higher than top-range CPUs, in part owed to the specialised arithmetic hardware they feature, targeting these operations. A certain level of excitement and for lack of a better term, 'hype' surrounds naïve comparisons circled around those, which may perhaps be pushing the development of on-GPU applications on commercial settings. Throughout our investigation of our own applications and more general optimisation techniques, we encountered a plethora of community posts requesting help in mapping workloads to the GPU. In many cases, it was evident that the chosen workload would not map onto the GPU efficiently, or the way a workload was approached was in violation of the model of parallelism of these devices. In academic literature however, we have encountered very limited documentation of such failed attempts, or of otherwise intuitive techniques, and their reasons for the lack of their success on the GPU. Some examples do however exist, such as [108] where the authors document, otherwise undocumented, details on dynamic parallelism overheads and its lack of potential when parallelising a recursive task (which is intuitive for this type of task). We commend this and other such work for the insights they have offered to us, and the field.

In reality, and as our work has shown, development of performant techniques and applications on the GPU, compliant with the model of parallelism at hand, is an arduous task which involves substantial effort especially when targeting a range of current CCs or attempting to foresee future ones and any drawbacks they may introduce in the context of these techniques. The fast evolving hardware and variable support for operations, which may seize completely from some point onwards, is an unstable bridge to cross. Optimised code may under-perform in subsequent hardware iterations, or even fail to operate all together as a result of this instability. Beyond this instability and unconventionality, GPGPUs have rather restricted scope for being "general purpose". They are indeed coprocesors, and depend upon a host system (itself featuring some form of CPU) to coordinate them and supply/consume all data required. They cannot interact with peripheral devices, storage, or other host resources (e.g., network) without the intervention of the host system and hence no operating system can run purely on a GPGPU at present.

Our work and observations presented here are based exclusively on hardware and software produced by NVIDIA who lead the field of programmable GPUs and are arguably the most adopted in this field, if that were to be measured through adoption by large-scale cloud computing providers (e.g., Google, AWS, OVH, IONOS, etc.)*. Whilst we have not investigated other such platforms ourselves, we speculate that the majority of techniques and points made here apply to other massively parallel platforms utilising

*All listed providers own and rent GPU supplied exclusively by NVIDIA. We are unaware of mainstream providers renting GPUs of other manufacturers.

the same model of parallelism. Our views presented here are not intended to suggest inapplicability and discourage the use of GPUs in the scientific sphere, or every day computing, rather the opposite. As our work has shown, it is possible to put these coprocessors to good use and benefit from their performance offerings in a wide range of applications. In a more playful manner, we like to think of the GPU as the "teenager of (co)processors" that requires special handling to do something within its merit, but can do it exceptionally well.

We suggest a level headed and methodical approach is used by anyone aiming to harness the power of these devices, to establish what can and cannot be done with these intricate coprocessors, before significant efforts (as typically required) are invested in them. This suggestion is not strictly bound to GPUs, as many of these challenges are applicable to other massively parallel (co)processors (e.g., FPGAs).

Chapter 6

Summary

The overarching objective of this thesis has been to examine the applicability of GPUs in tackling computationally demanding ('hard') problems and the means by which this can be achieved. In particular, we have presented the design and implementation of a GPU-based SAT checker and a full on-GPU N-Queens solver that surpasses the performance of existing work in literature. Substantial effort was involved in making the GPU kernels performant, in essence using them as proxies to study optimisation techniques that can be applied to a broader range of hard problems. To aid future efforts in the solving of such problems we have presented a list of techniques, observations and optimisations, that we believe will be of help. We will continue our work in pursuit of future directions, presented under the respective chapters, for each problem to explore further opportunities for performance gains.

For both the Boolean Satisfiability and N-Queens problems, we used backtracking search initially in line with common solving techniques for these particular problems. For SAT, backtracking search was found unsuitable for implementation on the GPU, primarily because of the large dependency on memory. On the other hand, for the N-Queens problem, the search had minimal memory requirements and using the paradigm proved successful in producing a competitive solver. Work focusing on the applicability of the backtracking paradigm on GPUs has previously been conducted [61], with which our findings concur. The findings of this work illustrate that the backtracking paradigm itself can be successfully used on the GPU, but its success depends strongly on the task at hand and its properties.

Throughout our work we have highlighted the role of optimisation in every stage of development on the GPU and produced techniques and recommendations for each stage, for others wishing to follow a course similar to ours. More specifically, starting from the algorithm design stage, the model of computation on the GPU needs to be factored in to decide which operations can and cannot be efficiently implemented.

Subsequently, the implementation stage needs to bring the algorithmic optimisations to fruition, making relevant implementation choices to achieve good performance. Finally, following a successful implementation, a multi-step optimisation stage should be performed to identify areas restricting performance and mitigate their effects.

On occasion, algorithmic design choices for massively parallel manycore platforms may appear counter-intuitive from the viewpoint of serial, or even for mildly* parallel computation. This may be for instance the avoidance of any concurrent access at the cost of persistent re-computation. As our work highlights, these choices are necessary for later performant implementations.

From the perspective of the implementation, intricacies of GPUs may require different implementations of the same algorithm to better suit different hardware generations. This was the case for our N-Queens solver, for which the shared memory-based kernel was better suited for some (earlier) devices in contrast to the register-based kernel. The degree to which this applies to most or all massively parallel manycore platforms in general is hard to estimate without a large scale study of many such platforms, hence we are not in a position to assess its prevalence. Nevertheless, it may be required by some and therefore should not be ruled out.

Lastly, optimising a working implementation requires a two-pronged approach. On one hand, the implementation itself can be optimised using high-level programming constructs to trigger more of the compiler's optimisations. This can be something rather small, such as asserting the truth of predicates which are in turn offered as assumptions to the compiler, or something more substantial such as a manual memory re-structure. On the other hand, even with optimisations on the high-level code, the output of the compiler is likely not to approach the performance optimum as much as possible [63, 37]. To counter that, one has to drop a level lower and tweak the compiler's output, manually optimising subroutines in the PTX code generated. These lower-level tweaks can offer performance increases but must be maintained and tested thoroughly.

*This is to distinguish parallel computation or relatively small degree, to massive parallelism.

Appendix A

General Appendix

A.1 PTX Instruction Format and Reference

PTX instructions are documented thoroughly in NVIDIA's documentation pages [90]. This documentation is in-depth and assumes deeper understanding of GPU hardware as it is aimed at experienced developers. To aid the reader's understanding of PTX listings, Table A.1 presents a quick, minimalist reference containing instructions used throughout this document and explaining them at a higher-level. PTX instructions are generally of the form `instruction dest, op1, op2, ...;` where `dest` is a destination register variable, followed by zero or more operands (`op1`, `op2`, ...). Operands may be registers, or constant expressions (e.g., constant values). It is worth noting that a register variable does not necessarily map one-to-one to a register following compilation into SASS.

This work makes use primarily of integer, logic, and bitwise operations. Instructions listed in Table A.1 may support more types than shown, or have different manifestations to those listed under different types (e.g., floating point types). The types each instruction is applicable to are given only for reference. The exact behaviour of the instruction under each type may differ, and in some cases, the operands may be of different types between them, or to the destination register. Types listed are either `pred` (i.e., a predicate register variable type) or take the form of a single letter (i.e., `b` for *bits*, `u` for *unsigned* or `s` for *signed*) followed by the bit width of the type (i.e., 16, 32, 64).

Instruction	Types	Description
`abs rd, r1`	s{16,32,64}	Computes the absolute value of `r1`, storing the result in `rd`.

add rd, r1, r2	s{16,32,64}, u{16,32,64}	Performs addition between operands r1 and r2, storing the result in operand rd.
and rd, r1, r2	b{16,32,64}, pred	Computes the bitwise AND of operands r1 and r2, storing the result in operand rd.
bfe rd, r1, r2, r3	u{32,64}, s{32,64}	Extracts r3 many bits from r1, starting at position r2. The resulting bit field is stored in rd.
bfi rd, r1, r2, r3, r4	b{32,64}	Inserts into r2 r1 bit field of r4 many bits from r1, starting at position r3. The result is stored in rd.
bfind rd, r1	u{32,64}, s{32,64}	Finds the index of the first bit set in operand r1, storing the result in operand rd.
bmsk rd, r1, r2	b32	Constructs r1 mask of r2 many bits starting from position r1, storing the result in rd.
bra t1	*N/A**	Branches to the instruction labelled by the PTX label t1, and continue execution from there.
brev rd, r1	b{32,64}	Reverses the bits in operand r1, storing the result in operand rd.
clz rd, r1	b{32,64}	Counts leading zeros in operand r1, storing the result in operand rd.
cvta [...]	*Omitted*†	Converts r1 generic address to r1 typed address.
cvt [...]	*Omitted*†	Converts between different data types.
div rd, r1, r2	u{16,32,64}, s{16,32,64}	Computes the quotient of operands r1 and r2, storing the result in operand rd.
fns rd, r1, r2, n	b32	Finds the n^{th} set bit in r1 starting from bit index r2, and stores its index in rd.
ld rd, [...]	*Omitted*†	The exact semantics of this instruction are complex, but broadly, the right hand side evaluates to either a register or constant expression indicating a location from where data can be read. This instruction performs a load of data into rd from the indicated location.

*This is an instruction for flow of control which does not operate on any typed data.

†This instruction is complex, and the types supported depend on which mode is being used. Types have been omitted as they are unimportant here.

lop3 rd, r1, r2, r3, c	b32	Performs r1 logical operation between r1, r2 and r3 based on the look-up values in the constant value operand c.
mad rd, r1, r2, r3	u{16,32,64}, s{16,32,64}	Performs a multiply-and-add operation multiplying r1 with r2 and adding r3, storing this result in rd.
max rd, r1, r2	u{16,32,64}, s{16,32,64}	Computes the maximum of operands r1 and r2, storing the result in operand rd.
min rd, r1, r2	u{16,32,64}, s{16,32,64}	Computes the minimum of operands r1 and r2, storing the result in operand rd.
mov rd, r1	b{16,32,64}, s{16,32,64}, u{16,32,64}, pred	Moves the value of register r1 into rd. If r1 is not r1 register, the instruction moves the non-generic address of the operand r1 into rd instead.
mul rd, r1, r2	u{16,32,64}, s{16,32,64}	Computes the product of operands r1 and r2, storing the result in operand rd.
neg rd, r1	s{16,32,64}	Computes the arithmetic negation (i.e., change of sign) of r1, storing the result in rd.
not rd, r1	b{16,32,64}, pred	Computes the bitwise negation of r1, storing the result in rd.
or rd, r1, r2	b{16,32,64}, pred	Computes the bitwise OR between operands r1 and r2, storing the result in rd.
popc rd, r1	b{32,64}	Counts the number of set bits in operand r2, storing the count in operand rd.
rem rd, r1, r2	u{16,32,64}, s{16,32,64}	Computes the remainder of integer division between r1 and r2 and stores it in rd.
ret	*N/A*[*]	Transfers control back to caller of a procedure (i.e., return).
sad rd, r1, r2, r3	u{16,32,64}, s{16,32,64}	Computes the sum of absolute differences between r1 and r2 and r3 as $abs(a-b)+c$, storing the result in rd.
selp rd, r1 , r2, r3	b{16,32,64}, s{16,32,64}, u{16,32,64}, pred	Selects operand r2 if predicate operand r1 is true, otherwise selects operand r3, and stores the value of the chosen operand in rd.

[*]This is an instruction for flow of control which does not operate on any typed data.

`setp p1, r1, r2`	b{16,32,64}, s{16,32,64}, u{16,32,64}, pred	Sets predicate register operand `p1` based on comparison between operands `r2` and `r3` under some logical operation (e.g., equality, less-than, more-than, etc.).
`shl rd, r1, r2`	b{16,32,64}	Shifts the value in `r1` left by `r2` many positions (clamping to the maximum width of the type) and stores the shifted value in `rd`.
`shr rd, r1, r2`	b{16,32,64}	Shifts the value in `r1` right by `r2` many positions (clamping to the maximum width of the type) and stores the shifted value in `rd`.
`st [...], r1`	*Omitted*[*]	Like `ld`, the semantics of this instruction are complex. Broadly, the instruction stores the data from `r1` to some location indicated by the left hand side operand(s).
`sub rd, r1, r2`	u{16,32,64}, s{16,32,64}	Computes the difference between operands `r1` and `r2`, storing the result in operand `rd`.

Table A.1: Quick reference of basic integer and bitwise PTX instructions, the types upon which they operate, and their intended functionality.

The `lop3.lut` instruction is described as a logic operation which is somewhat misleading as one may expect the result to be a truth value, when in fact it is a 32-bit bit word. The instruction does indeed apply the logic operation at hand, but does so between each bit on its three inputs, resulting in 32 truth values that are in turn encoded as a 32-bit word. More specifically, for each $0 \leq i < 32$, the logic expression at hand is applied between bits a_i, b_i and c_i of the input operands a, b and c, with the resulting truth value represented as the bit r_i of the resulting 32-bit word.

A.1.1 Clarification on Bitwise Leading Zero Counting

The PTX instruction `bfind` is seen in this work in two variants. The pure instructon (i.e., `bfind`) and the shift-amount calculation variant (i.e., `bfind.shiftamt`). The latter performs the same function as `bfind`, with the difference that the value it produces is no longer the index of the most significant (non-sign in the case of signed inputs) set bit, but the number of left shifts required to place said bit at the most significant position in the bit word.

The purpose of `clz` comes into question considering the function of `bfind.shiftamt`. This is because, the number of left shifts required to raise the highest set bit to the most

[*]This instruction is complex, and the types supported depend on which mode is being used. Types have been omitted as they are unimportant here.

significant position in an unsigned bit word is also the number of preceeding zeros to the highest set bit. The two instructions behave differently only in the case where the input register holds the value of 0. In that case, `clz` will produce the value *b* which will be 32 or 64 when the input is a 32- or 64-bit type respectively. To the contrary, `bfind.shiftamt` will result in the value 0xFFFFFFFF in that case.

In terms of the implementation of these instructions, both `clz` and `bfind.shiftamt` result in some use of the `FLO` SASS instruction*. Specifically, for 32-bit types, `bfind.shiftamt` reslts in a single `FLO.U32.SH` SASS instruction whereas `clz` results in two SASS instructions, namely `FLO.U32` and `IADD`. The exact semantics and differences between the two `FLO` instructions are not officially documented, in line with the rest of the SASS ISA. We speculate that the definitions of the PTX instructions `bfind` and `bfind.shiftamt` map directly to `FLO.U32` and `FLO.u32.SH` respectively, based on our experiments.

Based on observations made in this section, it would appear that `clz` acts as an alias of `bfind.shiftamt` in the majority of cases, for the intersection of types they both support (i.e., `u32` and `u64`). Currently, `bfind.shiftamt` translates to fewer SASS instructions than its counterpart and should be preferred where possible for the relevant microarchitectures. In future microarchitectures, the potential for the introduction of direct hardware support for `clz` or an alternative (perhaps more performant) implementation should not be dismissed.

*Tested for architectures up to and including Ampere.

A.2 SASS Instruction Format and Reference

The syntactical structure of SASS instructions is largely the same as that of PTX instructions, as detailed in Appendix A.1. Whilst at first it may appear as if a one-to-one mapping between PTX and SASS instructions exists, this is not always the case. SASS instructions are architecture-specific whereas PTX acts as an intermediate layer, agnostic of hardware-specifics. The conversion from PTX to SASS may in some cases result in one PTX instruction compiling into multiple SASS instructions. The syntax of SASS instructions is documented, but their operands, side-effects, performance and other characteristics are kept 'secret' by NVIDIA [88].

Appendix A.2 presents a handful of instructions used throughout this work, alongside a brief explanation (or mapping to an earlier discussed PTX instruction) to aid the reader. This table is intended as a reference to better understand this work and not as guide to altering SASS code.

Instruction	Description
`LOP3`	See `lop3` in Table A.1.
`MOV`	See `mov` in Table A.1.
`S2R`	Moves the value of a special-purpose register into a regular register. This instruction may result from a PTX `mov` instruction.
`SHF`	Funnel shift operation. A funnel shift concatenates two 32-bit operands, shifts them by k bits left/right and produces the 32 most/least significant bits respectively.
`UIADD3`	Uniform integer addition between three operands. Similar to `add` in Table A.1.
`ULDC`	Load data from constants memory* into a uniform register. See `ld` in Table A.1.

Appendix B

SAT Checker

This appendix covers content relevant to Chapter 3. Content presented here (i.e., code listings, data tables, etc.) is either referenced by the relevant section(s) in the main body of this work, or additional information for the interested reader.

B.1 General Structures and Definitions

This section collates the definitions of directives and types of/or structures (primarily in CUDA-C code). These structures appear as-is in the code used to gather experimental results and, unless otherwise indicated, their padding and memory layout are left in the hands of the compiler to decide.

The standard type `uint32_t` defined in the standard header `inttypes.h` is masked as the type `assignment_t` for the purposes of our code (see Listing B.1). The intention is for this definition to change to an appropriate bit-manipulation-safe type, at least 32-bits in width if needs be.

```
1 typedef uint32_t assignment_t;
```

Listing B.1: Definition of type `assignment_t`. A complete or partial assignment is comprised of zero or more instances of this type.

The type `litval_t` (see Listing B.2) is an enumeration of possible assignments to literals/variables (as appropriate) including two non-binary values representing the absence of an assignment and an 'error' state involving the assignment.

The types `clause3_t` and `expression_t` (Listing B.3) are used together to represent a SAT formula. The type `expression_t` simplifies a structure comprising of the non-negative integers `varCount` and `length` which represent the number of distinct variables and the number of clauses in the formula respectively. Furthermore, the `expression_t`

```
typedef enum {
 LIT_UNSET = 0,
 LIT_TRUE = 1,
 LIT_FALSE = 2,
 LIT_ERROR = 3
} litval_t;
```

Listing B.2: Enumeration of possible assignment values with their corresponding integer representation.

structure refers to a number of `clause3_t` structures, each of which represents a clause with three literals.

```
typedef struct {
 int32_t lits[3];
} clause3_t;

typedef struct {
 uint32_t varCont, length;
 clause3_t* clauses;
} expression_t;
```

Listing B.3: Definitions for the structure types `clause3_t` and `expression_t`.

The preprocessor definition shown in Listing B.4 expands to a bitwise alternative to the absolute value of a two's complement represented integer.

```
#define FAST_32_BIT_ABS(a)((((a)>>31)^a)-((a)>>31))
```

Listing B.4: Definition of preprocessor directive for sign removal of 32-bit integer types.

B.2 Literal Satisfiability Checking

```
__device__ __host__ litval_t get_lit(const assignment_t*
    ↪ __restrict__ const a, int32_t lit) {
 lit = FAST_32_BIT_ABS(lit);
 const uint32_t word = ((lit - 1) / LITS_PER_WORD);
 const uint32_t indx = (lit - 1) % LITS_PER_WORD;
 return (litval_t)((a[word] >> (indx << 2)) & 0x3);
}
```

Listing B.5: Retrieval of assignment to a given variable from the input partial assignment, in the form of a `litval_t`.

The retrieval of an assignment from the data structure defined in Appendix B.1 involves identifying the 32-bit word containing the assignment and extracting the relevant bit pair, as shown in Listing B.5. Checking whether or not a literal has been satisfied by an assignment is trivial considering the numeric representation of truth values (see Listing B.2), since the least significant bit of the relevant assignment values (i.e., `LIT_TRUE` and `LIT_FALSE`) is 1 and 0 respectively. The other pair of possible assignments (namely the absence of one or an 'error') must also be accounted for however. As such, the result of the check of the truth value against the literal is voided (i.e., zero) if the two least significant bits of the assignment value put through an XOR between them result in zero. This way, the outcome of the check is only true if the assignment value is appropriate for the literal in question, and not `LIT_UNSET` or `LIT_ERROR`. The implementation of this check can be seen in Listing B.6.

```
1 __device__ int is_lit_satisfied(const assignment_t* const a,
    ↪ const  int32_t lit) {
2  int l = get_lit(a, lit);
3  return ((l >> 1) ^ (l & 1)) & ((l & 1) ^ (lit < 0));
4 }
```

Listing B.6: Literal satisfaction check under a given assignment.

B.3 Host-Side Literal Re-Mapping

```
void swap(expression_t* e, assignment_t* a, int32_t j, int32_t
    ↪ i) {
 if (j == i)
  return;
 for (uint32_t o = 0; o < e->length; o++) {
  int32_t* cl = e->clauses[o].lits;

  for (uint32_t x = 0; x < 3; x++) {
   if (cl[x] == j || cl[x] == -j)
    cl[x] = (cl[x]<0?-1:1) * i;
   else if (cl[x] == i || cl[x] == -i)
    cl[x] = (cl[x]<0?-1:1) * j;
  }
 }
 litval_t s = get_lit(a, j);
 set_lit(a, j, get_lit(a, i));
 set_lit(a, i, s);
}

int remap_unsat_lits(expression_t* e, assignment_t* a) {
 uint32_t i, j;
 for (i = 0, j = 0; j <= e->varCont && i <= 64; j++) {
   if (j > i && get_lit(a, j) == LIT_UNSET) {
      swap(e, a, j, ++i);
      j = 0;
   }
 }
 return i;
}
```

Listing B.7: Literal remaping proceedure shifting unset literals in the expression and corresponding assignment within the range 0 to 64.

As described in Section 3.2, unset variables must be sequential in order (i.e., all unset variables in the input formula must be represented as sequential integers). This may not be the case, hence a literal re-mapping process is employed as shown in Listing B.7. In essence this procedure identifies the unset variables in the formula and if those are not in the range of unset variables permitted (i.e., break sequence), swaps them with the corresponding (set) variable. Whilst initially such algorithm may be reminiscent of sorting, the process is much simpler. For each variable v starting from 1 and iterating to the maximum number of unset variables supported (64), if an assignment exists for v, then the first unset variable $u > v$ is identified and swapped with v. The swap in this instance refers to the names identifying the two variables and not the order or composition of clauses in the formula.

B.4 Full GPU-Side SAT Checking

```
__global__ void bf_solve_kernel(const expression_t* const
    ↪ __restrict__ g_e, assignment_t* __restrict__ g_a, volatile
    ↪ uint32_t* __restrict__ solution_flag, const uint8_t
    ↪ unsetCnt, const uint8_t device_id, const uint8_t devices,
    ↪ const uint32_t solvers) {
 extern __shared__ assignment_t s_assgn[];
 if (*solution_flag) return;

 if (THREAD_INDX_2D_BLOCK < ASSIGNMENT_COUNT(g_e->varCont))
  s_assgn[THREAD_INDX_2D_BLOCK] = g_a[THREAD_INDX_2D_BLOCK];
 __syncthreads();

 const uint64_t limit = (uint64_t) ceil((1LLU << unsetCnt) /
    ↪ (((double)solvers) * devices));
 uint64_t i, local_assignmen = (solvers * device_id) +
    ↪ THREAD_INDX_1D_GRID_2D_BLOCK;
 for (i = 0; i < limit; ++i) {
  if (local_assignmen > (1LLU << unsetCnt)) break;
  const int check = d_check(g_e, s_assgn, local_assignmen);
  __syncwarp();
  if (check) {
   atomicAdd(solution_flag, 1);
   break;
  }
  local_assignmen += solvers;
 }
}
```

Listing B.8: SAT formula checking kernel with per-thread local assignments.

Part of the kernel described in Section 3.2 is shown in Listing B.8. This code is provided for the interested reader, and relates to the discussion in the relevant section. As part of this kernel, each thread performs a check of its local assignment together with the input partial assignment, against the input expression. The check itself is rather involved as each literal of each clause has to be checked in a manner shown in Listing B.9. During this check it is enough to confirm that one literal in each clause is satisfied, yet in practical terms, this would result in high degrees of divergence. For this reason, all three literals of each clause are checked against their respective assignments.

```
__device__ __forceinline__ int d_check(const expression_t* const
    ↪ __restrict__ g_e, assignment_t* const __restrict__ g_a,
    ↪ const uint64_t assgn) {
 register bool whole = 1;
 const uint32_t len = g_e->length;
 for (uint32_t i = 0; whole & (i < len); ++i) {
  const int32_t* __restrict__ const clause =
    ↪ g_e->clauses[i].lits;
  const int32_t a = clause[0], b = clause[1], c = clause[2];
  whole &= (is_lit_satisfied_complete(g_a, a, assgn)
    | is_lit_satisfied_complete(g_a, b, assgn)
    | is_lit_satisfied_complete(g_a, c, assgn));
 }
 return whole;
}
```

Listing B.9: Helper function for Listing B.8, performing a complete formula satisfaction check using both the thread-local partial assignment μ_l and input partial assignment μ_p.

B.5 Experimental Results

In Section 3.3.6, a range of experimental setups are discussed, involving different GPU hardware belonging to different systems. The 'raw' data collected is presented in this section beginning with results from different commercial GPUs in Appendix B.5.1, followed by results on data centre devices in Appendix B.5.2 and results exploring the impact of clause re-ordering in Appendix B.5.3.

B.5.1 Raw Data per Tested Device

Three different systems were used to collect the results shown in Table B.1. Specifics of each system beyond the model of GPU used are largely irrelevant to these results, as the times shown reflect only the time taken by the GPU to complete the work. To minimise any timing inconsistencies, it was ensured, where possible, that systems used were properly cooled (to mitigate potential thermal throttling). The ten runs shown for each input were made in sequence of one-another.

Problem Instance	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average CPS
GTX1080ti, 136 blocks with 1024 threads											
20-200	23	22.9	22.9	23.1	24	22.9	22.9	23.3	23.5	21	1,462,066,754
20-400	24.5	24.7	25.4	24.3	22.4	22.9	23.4	22.3	22.2	22.2	1,432,114,042
30-200	659	591	599	602	595	597	599	598	598	595	1,779,780,912
30-400	661	613	607	612	610	608	605	610	604	609	1,749,050,047
35-200	19442	19791	19473	19701	19765	19784	19614	19748	19826	19743	1,745,150,181
35-400	20525	20428	20552	20443	20718	20778	20739	20685	20828	20697	1,664,772,467
40-200	682746	675096	679068	680868	682389	677592	683239	682012	675020	687510	1,615,612,615
40-400	682699	679557	675212	684010	676601	676690	683290	678234	678291	678794	1,618,505,003
Two GTX1080ti, 136 blocks with 1024 threads											
20-200	13.46	13.46	13.46	13.585	13.865	13.235	12.735	12.84	12.865	12.79	2,528,143,536
20-400	12.08	12.09	12.115	12.38	12.535	12.14	12.11	11.485	11.22	11.24	2,811,408,970
30-200	348.59	304.1	298.075	301.615	300.765	301.1	300.99	299.58	301.245	301.15	3,508,240,498
30-400	341.94	287.62	269.465	269.055	268.125	269.94	269.295	271.26	267.785	270.215	3,849,879,105
35-200	10469.5	10439.5	10462.5	10486.5	10490.5	10452.5	10488	10498.5	10526.5	10509	3,278,645,227
35-400	8478.5	8397.5	8420.5	8422.5	8369.5	8387	8428	8390	8462.5	8383	4,082,560,663
40-200	325694.5	324153.5	324123	324901.5	324191.5	324313	323029.5	326022	325321.5	323724.5	3,387,165,146
40-400	321324	321724	322771	322552	321537.5	324881	323524	323748	323199	322561.5	3,405,993,242
RTX2080ti, 136 blocks with 1024 threads											
20-200	11	10.9	10.9	10.9	10.9	10.9	10.9	10.9	8.7	8.6	3,207,880,688
20-400	9.8	9.8	9.7	9.8	9.7	9.7	9.7	9.7	9.7	9.7	3,448,554,162
30-200	241	208	209	210	210	210	210	210	210	210	5,045,779,248
30-400	259	205	206	206	205	205	205	205	205	205	5,098,489,193
35-200	7649	7671	7677	7678	7680	7679	7680	7681	7735	7739	4,469,908,333
35-400	8246	8272	8276	8317	8341	8343	8343	8344	8344	8366	4,130,173,378
40-200	239479	240867	241225	241274	241267	241007	241011	240746	239558	239507	4,569,985,830
40-400	239052	240792	241042	241346	241471	241422	241383	241380	241364	241319	4,561,208,227
RTX4090, 224 blocks with 1024 threads											
20-200	4	4	4	4	4	4	4	4	4	4	8,388,608,000
20-400	4	4	4	4	4	4	5	4	4	4	8,184,007,805
30-200	99	99	98	98	99	99	98	99	99	99	10,878,843,202
30-400	108	107	107	107	107	106	107	105	106	106	10,072,624,991
35-200	3645	3639	3633	3637	3639	3644	3638	3634	3641	3640	9,442,082,541
35-400	4236	4236	4236	4236	4236	4236	4236	4236	4236	4236	8,111,364,110
40-200	116900	116900	116909	116904	116912	116888	116900	116895	116902	116900	9,405,493,775
40-400	115201	115199	115296	115280	115235	115229	115233	115254	115253	115310	9,540,313,823

Table B.1: Raw execution time measurements (runs) and average CPS per input problem, for each of the devices tested.

B.5.2 Raw Data for Concurrently Executing Tesla A100 GPUs

Besides the results shown in Appendix B.5.1, the scalability of this approach was tested using A100 GPUs as discussed in Section 3.3.2, the results for which can be seen in Table B.2. Experiments ran in one node* of the AccelerateAI [12] cluster. Each test suit was run using one, two, four and eight GPUs, in the same fashion as other results (i.e., ten repeated runs per input). The speedup relative to the baseline (i.e., a single A100) for each input is also shown.

Experiment	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average CPS	Speedup
1xA100, 224 blocks with 1024 threads												
20-200	18.83	13.87	13.86	13.85	13.85	13.84	13.85	13.85	13.84	13.84	2,338,416,746	**N/A (Baseline)**
20-400	19.61	14.57	14.58	14.59	14.58	14.57	12.57	11.27	11.27	11.25	2,416,216,270	**N/A (Baseline)**
30-200	282.38	192.82	186.89	189.23	189.23	189.65	186.27	186.89	186.89	187.63	5,428,769,186	**N/A (Baseline)**
30-400	262.23	218.5	218.5	218.5	218.52	218.5	218.5	218.53	218.51	218.49	4,817,590,241	**N/A (Baseline)**
35-200	7107.42	7034.88	7035.08	7035.07	7034.86	7034.83	7034.8	7034.88	7034.85	7034.81	4,879,154,606	**N/A (Baseline)**
35-400	5301.12	5296.41	5296.32	5296.1	5296.08	5296.11	5296.06	5296.08	5296.05	5296.09	6,487,079,210	**N/A (Baseline)**
40-200	182883	182741	182736	182734	182733	182737	182736	182768	182737	182747	6,016,308,409	**N/A (Baseline)**
40-400	183060	182852	182823	182826	182814	182808	182804	182825	182851	182830	6,013,215,105	**N/A (Baseline)**
2xA100, 224 blocks with 1024 threads												
20-200	14.76	8.49	8.48	8.49	8.48	8.49	8.49	8.48	8.49	8.48	3,682,252,915	1.57
20-400	13.65	7.41	7.4	7.4	7.4	7.4	7.4	7.4	7.4	7.4	4,179,911,702	1.73
30-200	187.54	126.23	100.59	98.74	98.45	98.66	98.48	98.74	98.45	98.46	9,722,901,169	1.79
30-400	200.71	133.15	108	107.99	107.99	108	107.97	107.99	107.99	107.99	8,964,440,565	1.86
35-200	3433.62	3298.93	3298.91	3298.8	3298.91	3298.87	3304.74	3298.94	3305.99	3298.93	10,369,104,212	2.13
35-400	2719.98	2619.19	2619.2	2619.2	2619.17	2619.41	2619.45	2619.18	2619.21	2619.17	13,067,936,063	2.01
40-200	90505.7	90672.8	90429.7	90444	90689.9	90494.6	90462	90559.3	90342.4	90350	12,149,966,828	2.02
40-400	90741.5	90631.5	90324.9	90329.8	90632.3	90332.4	90327.3	90318.1	90318.9	90315.1	12,159,084,017	2.02
4xA100, 224 blocks with 1024 threads												
20-200	12.78	4.42	4.42	4.42	4.42	4.41	4.42	4.42	4.42	4.42	6,387,150,983	2.73
20-400	14.23	3.91	3.9	3.89	3.9	3.9	3.9	3.89	3.89	3.89	6,807,378,954	2.82
30-200	108.39	87.12	81.85	64.18	53.5	53.85	53.83	54.13	54.65	54.68	16,118,027,156	2.97
30-400	105.88	99.81	89.91	60.47	54.42	54.26	54.25	54.25	54.26	54.42	15,745,448,188	3.27
35-200	1853.97	1727.07	1727.07	1727.08	1727.07	1727.05	1727.08	1727.07	1727.05	1727.05	19,749,766,159	4.05
35-400	1429.92	1322.79	1323.37	1322.52	1323.28	1323.28	1323.35	1324.16	1323.87	1323.47	25,756,918,714	3.97
40-200	42496.6	42267.5	42251.9	42264	42273.1	42263.3	42257.8	42272.8	42295.3	42293.3	25,997,145,113	4.32
40-400	42277.1	42149.8	42151.3	42149.9	42150.6	42150.7	42150.9	42156.9	42162.4	42162.5	26,075,662,673	4.34
8xA100, 224 blocks with 1024 threads												
20-200	21.94	2.33	2.32	2.32	2.32	2.32	2.32	2.32	2.32	2.32	7,837,546,885	3.35
20-400	20.45	2.16	2.16	2.15	2.16	2.16	2.15	2.15	2.15	2.16	8,420,187,323	3.48
30-200	68.79	49.16	49.14	49.12	49.13	44.32	32.48	26.69	26.74	26.75	25,424,092,302	4.68
30-400	63.93	46.12	46.11	45.98	41.24	37.84	37.63	25.36	25.22	25.2	27,208,744,043	5.65
35-200	978.63	863.25	863.11	863.73	864.08	863.97	864	863.75	863.31	863.54	39,262,073,088	8.05
35-400	722	592.43	591.84	591.74	592.08	591.73	592.16	592.08	591.6	591.97	56,796,427,715	8.76
40-200	21301.6	21168.2	21185.2	21137.8	21134.9	21182.8	21121.5	21124.2	21124.3	21117.7	51,962,226,898	8.64
40-400	21284.6	21176.5	21144.5	21192.7	21166.2	21140.9	21177.5	21170.2	21183.6	21140.7	51,918,292,620	8.63

Table B.2: Raw execution time measurements (runs) and average CPS per input on 1, 2, 4, and 8 A100 GPUs using a grid of 224 blocks.

The kernel configuration for the results shown in Table B.2 involved 224 blocks of 1024 threads each. This figure is suitable for this application on the commercial devices presented in Appendix B.5.1. On the A100's however, we found that increasing the over-subscription factor quite significantly, to 1024 blocks of 1024 threads each, resulted in greater checking performance and more consistent speedups (discussed further in Section 3.3.2). This is illustrated by contrasting the results involving 224 blocks to those involving 1024 shown in Table B.3.

*Each node comprises of eight identical A100 GPUs.

Experiment	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average CPS	Speedup
1xA100, 1024 blocks with 1024 threads												
25-200	16.7997	11.3408	11.3674	11.3306	11.3152	11.39	11.3111	11.3326	11.3336	11.3306	2,823,221,283	**N/A (Baseline)**
25-400	16.642	12.374	12.3556	12.3648	12.3914	12.373	12.3679	12.3699	12.3761	12.3576	2,622,006,351	**N/A (Baseline)**
30-200	253.211	151.479	151.173	151.134	151.205	151.121	151.174	151.246	151.203	151.182	6,652,147,406	**N/A (Baseline)**
30-400	288.119	189.511	186.053	186.331	186.313	185.931	186.37	186.173	186.372	186.67	5,456,442,481	**N/A (Baseline)**
35-200	5689.31	5588.19	5597.66	5595.49	5601.13	5592.19	5585.72	5585.57	5575.39	5581.42	6,136,537,205	**N/A (Baseline)**
35-400	4273.29	4154.21	4153.8	4153.49	4149.98	4143.86	4157.03	4157.72	4152.96	4147.93	8,250,772,063	**N/A (Baseline)**
40-200	137696	137693	137773	137636	137786	137610	137600	137611	137619	137599	7,987,011,820	**N/A (Baseline)**
40-400	137375	137358	137324	137321	137324	137402	137332	137322	137319	137310	8,005,837,503	**N/A (Baseline)**
2xA100, 1024 blocks with 1024 threads												
25-200	8.5504	5.78355	5.82042	5.80403	5.77638	5.80915	5.79277	5.81632	5.81427	5.78662	5,523,006,947	1.96
25-400	13.4298	6.21466	6.15014	6.12659	6.13888	6.13888	6.13478	6.13888	6.1399	6.14502	4,880,112,023	1.86
30-200	151.815	133.073	79.2934	79.4655	79.5156	79.2924	79.0518	79.1951	79.3917	79.3579	11,678,067,356	1.76
30-400	165.989	126.292	89.3419	89.4536	89.3717	89.2836	89.4024	89.3553	89.2559	89.5242	10,659,921,782	1.95
35-200	2748.85	2647.25	2651.28	2654.53	2650.61	2649.11	2648.02	2649.98	2648.62	2649.57	12,918,249,880	2.11
35-400	2115.29	2071.5	2072.59	2068.9	2070.98	2072.26	2067.83	2071.54	2072.3	2073.14	16,553,864,938	2.01
40-200	68081.9	67987.7	68001.6	67976.9	67958.4	67947.2	67944.8	67960.4	67957.6	67995.5	16,173,759,098	2.03
40-400	68146.3	68045.3	68020.9	68089.2	68000.9	68011.6	67995.1	68059.7	67986.5	68007.8	16,160,653,352	2.02
4xA100, 1024 blocks with 1024 threads												
25-200	11.6255	3.30342	3.25222	3.28294	3.25939	3.25325	3.26042	3.2512	3.25734	3.2727	8,180,343,031	2.90
25-400	12.4334	3.38022	3.3792	3.3792	3.3792	3.37613	3.37613	3.37306	3.36589	3.37306	7,836,984,653	2.99
30-200	86.6048	78.3432	78.3626	58.8237	46.8337	42.5554	42.5953	42.6148	42.6936	42.8206	19,097,309,977	2.87
30-400	88.8586	82.2057	82.0992	61.7011	44.7754	44.8143	44.7457	44.7242	44.7396	44.8276	18,402,010,445	3.37
35-200	1511.97	1392.51	1393.42	1393.7	1392.84	1393.54	1393.71	1393.75	1393.13	1393.89	24,451,031,554	3.98
35-400	1167.87	1052.23	1054.33	1052.99	1051	1053.05	1052.68	1052.35	1053.07	1052.25	32,287,450,051	3.91
40-200	32611.1	32537.2	32498.5	32480.9	32526.7	32513.1	32483.4	32513.2	32480	32506.9	33,815,416,455	4.23
40-400	32615.1	32523.4	32510.2	32513.6	32479.6	32482.4	32483.4	32489	32472.5	32496.6	33,824,282,267	4.22
8xA100, 1024 blocks with 1024 threads												
25-200	20.5261	1.77357	1.76026	1.7623	1.76128	1.77562	1.75616	1.76128	1.75616	1.78278	9,214,330,156	3.26
25-400	18.645	1.74387	1.73773	1.73773	1.7367	1.7367	1.7367	1.73568	1.73466	1.7367	9,787,920,197	3.73
30-200	58.0772	39.8152	39.7916	39.7036	39.6739	39.6943	39.6605	27.3459	24.1551	22.2034	29,010,585,519	4.36
30-400	53.3801	38.8413	38.827	38.8239	38.8178	38.8321	27.7135	27.1933	27.2148	23.0502	30,443,982,893	5.58
35-200	819.343	696.23	695.104	696.322	695.844	695.609	697.009	696.866	695.817	696.39	48,499,640,383	7.90
35-400	597.074	461.207	460.715	462.262	460.872	460.908	462.09	460.498	462.086	460.654	72,361,201,725	8.77
40-200	16442	16299.3	16303.3	16292.1	16303.2	16292.5	16297	16294.1	16300.2	16297.7	67,404,501,068	8.44
40-400	16395.2	16286.5	16294.6	16291.3	16289.6	16290	16288.8	16299.7	16289.9	16301.4	67,443,509,263	8.42

Table B.3: Raw execution time measurements (runs) and average CPS per input on 1, 2, 4, and 8 A100 GPUs using a grid of 1024 blocks.

B.5.3 Raw Data on the Impact of Clause Reordering

The impact of sorting is assessed as described in Section 3.3.5. Measurements were taken with the sorting heuristic applied (Table B.4) and with the sorting heuristic applied in reverse (Table B.5). No other controlled variables were changed besides the order of clauses.

Experiment	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average CPS
25-200	15	15	15	15	15	13	12	12	13	12	2,449,228,613
25-400	13	13	13	13	13	13	13	13	13	13	2,581,110,154
30-200	249	183	183	184	183	182	184	183	182	184	5,660,209,931
30-400	258	216	217	217	217	217	217	217	217	217	4,858,560,290
35-200	8032	7980	7996	7999	8023	8009	8003	8003	8033	8065	4,287,303,741
35-400	5005	4960	4958	4978	4998	4999	4999	5030	5039	5041	6,870,985,736
40-200	153676	155248	155601	155761	155852	155869	155914	155957	155945	155974	7,067,192,107
40-400	189766	191141	191351	191389	191667	191433	191441	191454	191512	191002	5,750,114,676

Table B.4: Checking time readings in milliseconds and average CPS following clause reordering, on one RTX2080ti GPU with 136 blocks of 1024 threads each.

Experiment	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average CPS
25-200	13	12	12	12	13	13	12	12	9	9	2,867,900,171
25-400	11	11	11	11	11	11	10	10	10	10	3,165,512,453
30-200	259	225	213	209	222	209	214	213	209	209	4,920,906,618
30-400	290	222	223	223	223	223	223	223	223	223	4,676,575,889
35-200	7864	7818	7852	7855	7860	7858	7912	7916	7922	7919	4,361,701,326
35-400	10179	10210	10215	10212	10213	10277	10287	10288	10294	10329	3,352,038,786
40-200	265294	265846	265800	265224	264662	264641	264628	264626	264619	264590	4,149,210,084
40-400	268034	268657	268689	268536	267426	267406	268426	268634	268642	268724	4,097,802,184

Table B.5: Checking time readings in milliseconds and average CPS following reverse clause reordering, on one RTX 2080 GPU with 136 blocks of 1024 threads each.

Appendix C

N-Queens

This appendix covers content relevant to Chapter 4. Like other appendices, the content presented is either referenced by the relevant section(s) in the main body of text, or constitutes additional information for the interested reader.

Code presented here is segments referenced from the main text. The full code is available with DOI `10.5281/zenodo.14588371`.

C.1 General Structures and Definitions

The below structs and type definitions are used in subsequent sections of this chapter.

The types `bitset32_t` and `bitset64_t` are defined in Listing C.1 as a bit-manipulable type of at least 32 and 64 bits respectively.

```
1 typedef uint32_t bitset32_t;
2 typedef uint64_t bitset64_t;
```

Listing C.1: Definitions of `bitset` types.

Tracking diagonals as described in Section 4.3.2 involves two 64-bit words for the diagonals and antidiagonals respectively. The type `diagonals_t` (Listing C.2) simplifies a structure comprised of these two words.

```
1 typedef struct {
2  bitset64_t diagonal, antidiagonal;
3 } diagonals_t;
```

Listing C.2: Definition of the `diagonals_t type`.

During search, each thread maintains a search state represented as a structure (show in Listing C.3) and simplified by the type `nq_state_t`. The structure comprises of

a `diagonals_t` structure to track the occupation of diagonals, a 32-bit word to track column occupation, the current branching row of the search (as an 8-bit integer), and an array of N many 8-bit integers that store the index of a queen for each row. Here, and in following listings, N is a preprocessor definition which expands to the value of N being solved (as an integer).

The exact size of `nq_state_t` is known at compile time (i.e., no flexible array member) however this size will depend on the value of N.

```
typedef struct {
 diagonals_t diagonals;
 bitset32_t queens_in_columns;
 char curr_row;
 char queen_at_index[N];
} nq_state_t;
```

Listing C.3: Definition of the `nq_state_t` type representing an N-Queens search state.

During search, each thread accumulates the number of solutions it discovers in a variable of type `nq_result_t`. This type is defined as shown in Listing C.4, conditional to whether or not 64-bit result counters are intended to be used (determined by the definition of `USE_64_BIT_RESULT_COUNTERS` or absence thereof). Cases where 32-bit counters may be used are those where it is either guaranteed that no thread can overflow its counter, or where result integrity is not so important (i.e., benchmarking, testing, etc.). It must not be assumed that no thread will individually exceed the bounds of a type when the total number of solution is greater than these bounds, as solution counts per thread may, from our experience, be vastly different.

```
#ifndef USE_64_BIT_RESULT_COUNTERS
 typedef unsigned nq_result_t
#else
 typedef unsigned long long int nq_result_t;
#endif
```

Listing C.4: Definition of type `nq_result_t` used for solution accumulation.

Dependent upon the choice of kernel employed to count solutions, the calculation of shared memory space required will vary and ultimately determine the maximum number of threads per block. The register-based kernel uses shared memory to host the array component of each search state (i.e., queen indexes) for each thread. On the other hand, the shared memory-based kernel stores one full `nq_state_t` (see Listing C.3) for each thread. For each kernel, their shared memory requirements are variable, controlled by N, and thus the maximum number of threads per block needs to be calculated accordingly. The relevant computations determining these values can be seen in Listing C.5, where:

- `WARP_SIZE` is an integer constant definition, expanding to the number of threads per warp (in current architectures, this is always 32).
- `SMEM_SIZE` is an integer constant definition, expanding to the size (in bytes) of shared memory in the target device. This value varies between CCs, yet for the purposes of this calculation an under-approximation is safe and may only lead to block under-subscription.
- `MIN(a,b)` is a preprocessor definition, expanding to a simple ternary conditional operation resulting in the minimum between comparable values `a` and `b`.

```
#ifndef USE_REGISTER_ONLY_KERNEL
 // Register-based kernel block calculation
 #define COMPLETE_KERNEL_BLOCK_THREAD_COUNT MIN(((SMEM_SIZE -
   ↪ (WARPS_PER_BLOCK * sizeof(unsigned))) /
   ↪ sizeof(nq_state_t)) - (((SMEM_SIZE - (WARPS_PER_BLOCK *
   ↪ sizeof(unsigned))) / sizeof(nq_state_t)) % WARP_SIZE),
   ↪ 1024)
#else
 // Shared meemory-based kernel block calculation
 #define COMPLETE_KERNEL_BLOCK_THREAD_COUNT MIN((SMEM_SIZE /
   ↪ ((WARP_SIZE * N) + (sizeof(unsigned) * WARP_SIZE))) *
   ↪ WARP_SIZE, 1024)
#endif
```

Listing C.5: Preprocessor definitions of calculations determining the size of blocks depending on the choice of kernel.

C.1.1 Manipulation Functions and Definitions for `bitset` types

The types `bitset32_t` and `bitset64_t` presented in Appendix C.1 define a 32 and 64 bit-manipulable type. Shorthand expressions shown in Listing C.6 are used to manipulate bits within these types. As C-based languages default to treating integer literals as 32-bit, 64-bit variants of these macros are also defined where appropriate to ensure correctness with `bitset64_t`. These variants can safely be used with smaller types (i.e., `bitset32_t`) yet may result in unnecessary 64-bit manipulation instructions if the compiler cannot safely optimise those away.

```
#define BS_GET_BIT(bw,idx) (0x01&((bw)>>(idx)))
#define BS_SET_BIT(bw,idx) ((bw)|(1<<(idx)))
#define BS_CLEAR_BIT(bw,idx) ((bw)&~(1<<(idx)))
#define BS64_SET_BIT(bw,idx) ((bw)|(1LLU<<(idx)))
#define BS64_CLEAR_BIT(bw,idx) ((bw)&~(1LLU<<(idx)))
```

Listing C.6: Preprocessor definitions of bit manipulation on `bitset` types.

The aforementioned preprocessor definitions can be used independently throughout code. However, on the device it may be preferable (in terms of optimisation) to manually substitute those for PTX binary manipulation instructions, exposed by functions in Appendix C.2. From our experience, the compiler showed reluctance in using such instructions over multiple discrete bit manipulation operations. To facilitate this optimisation, inline device functions have been created as shown in Listing C.7. These functions will fall back to the use of the aforementioned definitions if experimental optimisations are disabled, or if compilation is not done for the device. The latter is a technicality that may not affect all compilation environments.

```
__device__ __host__ __forceinline__ bitset32_t
    ↪ bs_set_bit(bitset32_t bw, const unsigned idx) {
#if defined(__CUDA_VER__) &&
    ↪ defined(NQ_ENABLE_EXPERIMENTAL_OPTIMISATIONS)
    return intrin_bit_field_insert_b32(1, bw, idx, 1);
#else
    return BS_SET_BIT(bw, idx);
#endif
}

__device__ __host__ __forceinline__ bitset32_t
    ↪ bs_get_bit(bitset32_t bw, const unsigned idx) {
#if defined(__CUDA_VER__) &&
    ↪ defined(NQ_ENABLE_EXPERIMENTAL_OPTIMISATIONS)
    return intrin_bit_field_extract_u32(bw, idx, 1);
#else
    return BS_GET_BIT(bw, idx);
#endif
}

__device__ __host__ __forceinline__ bitset32_t
    ↪ bs_clear_bit(bitset32_t bw, const unsigned idx) {
#if defined(__CUDA_VER__) &&
    ↪ defined(NQ_ENABLE_EXPERIMENTAL_OPTIMISATIONS)
    return intrin_bit_field_insert_b32(0, bw, idx, 1);
#else
    return BS_CLEAR_BIT(bw, idx);
#endif
}
```

Listing C.7: Bit manipulation functions for both host and device code, including GPU intrinsic optimisations.

C.2 PTX-level Device Optimisation Functions

On some instances discussed throughout this work, manual intervention during compilation may be required to achieve better performance. To facilitate this, particularly for bit manipulation functionality, the functions shown in Listing C.8 are used. These functions compile to the respective PTX instruction(s), and are declared as forcibly inlined meaning no function call should result in the compiler output*. These instructions are not volatile and may be moved by the compiler in the final assembly output.

```
1  // Functions for SM_20 or newer
2  __device__ __forceinline__ unsigned int
       ↪ intrin_bit_field_extract_u32(unsigned bits, unsigned from,
       ↪ unsigned cnt) {
3      unsigned res;
4      asm("bfe.u32 %0, %1, %2, %3;" : "=r"(res) : "r"(bits),
           ↪ "r"(from), "r"(cnt));
5      return res;
6  }
7
8  __device__ __forceinline__ unsigned
       ↪ intrin_bit_field_insert_b32(unsigned bitfield_from,
       ↪ unsigned bitfield_to, unsigned start, unsigned cnt) {
9      unsigned res;
10     asm("bfi.b32 %0, %1, %2, %3, %4;" : "=r"(res) :
       ↪ "r"(bitfield_from), "r"(bitfield_to), "r"(start),
       ↪ "r"(cnt));
11     return res;
12 }
13
14 __device__ __forceinline__ int intrin_ffs_nosub(int of) {
15     unsigned res;
16     asm("brev.b32 %0, %1;" : "=r"(res) : "r"(of));
17     asm("bfind.shiftamt.u32 %0, %1;" : "=r"(res) : "r"(res));
18     return res;
19 }
20
21 __device__ __forceinline__ unsigned
       ↪ intrin_find_leading_one_shiftamt_u32(unsigned of) {
22     unsigned res;
23     asm("bfind.shiftamt.u32 %0, %1;" : "=r"(res) : "r"(of));
24     return res;
25 }
26
27 __device__ __forceinline__ int intrin_find_leading_one_u32(int
       ↪ of) {
```

*__forceinline__ is a specifier, documented as forcing the compiler to inline a function. This is different to inline or __inline__ which may be ignored.

```
28  unsigned res;
29  asm("bfind.u32 %0, %1;" : "=r"(res) : "r"(of));
30  return res;
31 }
32
33 // Functions for SM_30 or newer
34 __device__ __forceinline__ unsigned intrin_find_nth_set(unsigned
      ↪ mask, unsigned base, unsigned n) {
35  unsigned res;
36  // WARNING: Undefined behaviour for base >31.
37  asm("fns.b32 %0, %1, %2, %3;" : "=r"(res) : "r"(mask),
      ↪ "r"(base), "r"(n));
38  return res;
39 }
40
41 // Functions for SM_70 or newer
42 __device__ __forceinline__ unsigned
      ↪ intrin_bit_mask_32_clamp(unsigned bit_idx_from, unsigned
      ↪ how_many) {
43  unsigned res;
44  asm("bmsk.clamp.b32 %0, %1, %2;" : "=r"(res) :
      ↪ "r"(bit_idx_from), "r"(how_many));
45  return res;
46 }
```

Listing C.8: Assembly wrapper functions for device-side optimisations.

C.3 Diagonal Tracking Implementations

Blocked diagonals are tracked by each thread as part of their search state (see Appendix C.1). To extract a projection of diagonals onto the current row, update blocked diagonals based on the placement of a queen, or to clear blocked diagonals when a queen is removed, the helper functions shown in Listing C.9 are provided.

It is worth noting that, in the register-based kernel, the `diagonals_t` structure is not used nor can pointers to register variables be used. Thus, the function `dad_extract_explicit` is defined.

```
1  __host__ __device__ inline bitset32_t dad_extract(const
       ↪ diagonals_t* const from, const unsigned i) {
2   return dad_extract_explicit(from->diagonal, from->antidiagonal,
       ↪ i);
3  }
4
5  __host__ __device__ __forceinline__ bitset32_t
       ↪ dad_extract_explicit(const uint64_t diagonal, const
       ↪ uint64_t antidiagonal, const unsigned i) {
6     return (uint32_t)((diagonal >> i) | (antidiagonal >> (N -
          ↪ i))) & UINT32_MAX;
7  }
8
9  __host__ __device__ inline void dad_add(diagonals_t* const to,
       ↪ const bitset32_t row, const unsigned i) {
10  to->diagonal |= ((bitset64_t)row) << i;
11  to->antidiagonal |= ((bitset64_t)row) << (N - i);
12 }
13
14 __host__ __device__ inline void dad_remove(diagonals_t* const
       ↪ from, const bitset32_t row, const unsigned i) {
15  from->diagonal &= ~(((bitset64_t)row) << i);
16  from->antidiagonal &= ~(((bitset64_t)row) << (N - i));
17 }
```

Listing C.9: Implementation of diagonal tracking operations.

C.4 Queen Placement and Removal

```
__host__ __device__ __forceinline__ void
    ↪ place_queen_at(nq_state_t* __restrict__ s, const unsigned
    ↪ char row, const unsigned char col) {
 s->queens_in_columns = bs_set_bit(s->queens_in_columns, col);
 s->queen_at_index[row] = col;
 dad_add(&s->diagonals, 1U << col, row);
}

__host__ __device__ __forceinline__ void
    ↪ remove_queen_at(nq_state_t* __restrict__ s, const unsigned
    ↪ char row, const unsigned char col) {
 s->queens_in_columns = bs_clear_bit(s->queens_in_columns, col);
 s->queen_at_index[row] = UNSET_QUEEN_INDEX;
 dad_remove(&s->diagonals, 1U << col, row);
}
```

Listing C.10: Implementation of queen placement and removal functions for a given `nq_state_t`.

The steps performed for a queen placement or removal from an `nq_state_t` are encapsulated in a respective function as presented in Listing C.10. For the register-based kernel (see Section 4.3.4) such functions were not used to more carefully guide register allocation at compilation. As such, the respective operations were defined as preprocessor directives, shown in Listing C.11.

```
#define PLACE_QUEEN_AT(col, row, queens_in_columns,
    ↪ queen_indexes, diagonal, antidiagonal) { \
 queens_in_columns = bs_set_bit(queens_in_columns, (col)); \
 queen_indexes[(row)] = (col); \
 diagonal |= (1LLU << (col)) << (row); \
 antidiagonal |= (1LLU << (col)) << (N - (row)); \
}

#define REMOVE_QUEEN_AT(col, row, queens_in_columns,
    ↪ queen_indexes, diagonal, antidiagonal) { \
 queens_in_columns = bs_clear_bit(queens_in_columns, (col));\
 queen_indexes[(row)] = UNSET_QUEEN_INDEX;\
 diagonal &= ~((1LLU << (col)) << (row));\
 antidiagonal &= ~((1LLU << (col)) << ( N - (row))); \
}
```

Listing C.11: Expression of the operations in Listing C.10 as preprocessor directives.

C.5 Kernels Implementing DoubleSweep-Light

As discussed in Chapter 4, two kernels implementing DoubleSweep-Light are used in this work. The implementations of these two kernels are given under this section, to be studied in conjunction with the contents of Chapter 4 and offer deeper insight of the design choices discussed and crucially, their manifestations into code which may affect surrounding code as well. Functions and definitions shown under this appendix complement largely the implementations of the kernels shown, but the complete implementation is not shown. The complete code base is available upon request.

C.5.1 Shared Memory Kernel Implementation

The implementation of the shared memory-based kernel discussed in Section 4.3.3 is shown in Listing C.12.

```
1  __global__ void
       ↪ __launch_bounds__(COMPLETE_KERNEL_BLOCK_THREAD_COUNT,
       ↪ MAX_THREADS_PER_SM / (COMPLETE_KERNEL_BLOCK_THREAD_COUNT))
       ↪ kern_doitall_v2_smem(const nq_state_t* const __restrict__
       ↪ states, const uint_least32_t state_cnt, nq_result_t* const
       ↪ __restrict__ sols) {
2      const uint_least32_t local_idx = threadIdx.x;
3      const uint_least32_t global_idx = blockIdx.x * blockDim.x +
           ↪ local_idx;
4      __shared__ nq_state_t smem[COMPLETE_KERNEL_BLOCK_THREAD_COUNT
           ↪ + CEILING((sizeof(unsigned int) * WARP_SIZE),
           ↪ sizeof(nq_state_t))];
5      register nq_result_t t_sols = 0;
6
7      if (global_idx < state_cnt) {
8          smem[local_idx].queens_in_columns =
               ↪ states[global_idx].queens_in_columns;
9          smem[local_idx].diagonals = states[global_idx].diagonals;
10         smem[local_idx].curr_row = states[global_idx].curr_row;
11 #pragma unroll
12         for (int i = 0; i < N; ++i) {
13             smem[local_idx].queen_at_index[i] =
                   ↪ states[global_idx].queen_at_index[i];
14         }
15         __syncthreads();
16         do {
17             int res = smem[local_idx].curr_row >= locked_row_end;
18             bool any_alive = __ballot_sync(0xFFFFFFFF, res);
19             if (!any_alive) // Whole warp finished
20                 break;
21             if (res) device_advance_nq_state(&smem[local_idx],
                   ↪ locked_row_end);
```

```
22          __syncwarp(); // Threads made to converge before
               ↪ doublesweep_light
23          if (res) device_doublesweep_light_nq_state(
24          &smem[local_idx]);
25 #if defined(NQ_ENABLE_EXPERIMENTAL_OPTIMISATIONS) &&
      ↪ defined(USE_64_BIT_RESULT_COUNTERS)
26          static_assert(sizeof(smem[0].queens_in_columns) == 4,
               ↪ "Experimental optimisation: queens_in_columns
               ↪ MUST be a 32 bit type.");
27          static_assert(sizeof(t_sols) == 8, "Experimental
               ↪ optimisation: t_sols MUST be a 64 bit type.");
28          asm("{\n\t"
29             ".reg .pred %p;\n\t"
30             "setp.eq.u32 %p, %1, %2;\n\t"
31             "@%p add.u64 %0, %0, 1;\n"
32             "}" : "+l"(t_sols) : "r"
                  ↪ (smem[local_idx].queens_in_columns),
                  ↪ "r"(N_MASK));
33 #else
34          t_sols += (smem[local_idx].queens_in_columns == N_MASK);
35 #endif
36       __syncwarp();
37       } while (1);
38 #ifdef ENABLE_STATIC_HALF_SEARCHSPACE_REFLECTION_ELIMINATION
39       t_sols <<= (states[global_idx].queen_at_index[0] < N/2);
40 #endif
41    }
42 #ifdef USE_64_BIT_RESULT_COUNTERS
43    atomicAdd(&sols[blockIdx.x], t_sols);
44 #else
45    __syncthreads();
46    t_sols = block_reduce_sum_shfl_variwarp((unsigned)t_sols,
          ↪ (unsigned int*)
          ↪ &smem[COMPLETE_KERNEL_BLOCK_THREAD_COUNT]);
47
48    if (!local_idx) {
49       sols[blockIdx.x] += t_sols;
50    }
51 #endif
52
53 }
```

Listing C.12: Shared memory-based kernel implementation of DoubleSweep-Light.

C.5.2 Register Kernel Implementation

The implementation of the regiser-based kernel discussed in Section 4.3.4 is shown in Listing C.13.

```
__global__ void
    ↪ __launch_bounds__(COMPLETE_KERNEL_BLOCK_THREAD_COUNT,
    ↪ MAX_THREADS_PER_SM / (COMPLETE_KERNEL_BLOCK_THREAD_COUNT))
    ↪ kern_doitall_v2_regld(const nq_state_t* const __restrict__
    ↪ states, const unsigned state_cnt, nq_result_t* const
    ↪ __restrict__ sols) {
   const unsigned local_idx = threadIdx.x;
   const unsigned global_idx = blockIdx.x * blockDim.x +
      ↪ local_idx;
   __shared__ unsigned char
      ↪ smem[COMPLETE_KERNEL_BLOCK_THREAD_COUNT * N +
      ↪ sizeof(unsigned int) * WARP_SIZE];
   register nq_result_t t_sols = 0;

   if (global_idx < state_cnt) {
      unsigned char* const __restrict__ l_smem = smem +
         ↪ local_idx * N;
      register bitset32_t queens_in_columns =
         ↪ states[global_idx].queens_in_columns;
      register uint64_t diagonal =
         ↪ states[global_idx].diagonals.diagonal, antidiagonal
         ↪ = states[global_idx].diagonals.antidiagonal;
      register int curr_row = states[global_idx].curr_row;
   #pragma unroll
      for (int i = 0; i < N; ++i)   l_smem[i] =
         ↪ states[global_idx].queen_at_index[i];
#ifdef ENABLE_STATIC_HALF_SEARCHSPACE_REFLECTION_ELIMINATION
      register int mult_res_2 = l_smem[0] < (N >> 1);
#endif
      do {
         int res = curr_row >= locked_row_end;
         if (!__ballot_sync(0xFFFFFFFF, res)) break;
         if (res) {
            while (curr_row >= locked_row_end) {
               const register int queen_index = l_smem[curr_row];
               register bitset32_t free_cols =
                  ↪ (~(queens_in_columns |
                  ↪ dad_extract_explicit(diagonal,
                  ↪ antidiagonal, curr_row)) & N_MASK);
               if (queen_index != UNSET_QUEEN_INDEX) {
                  free_cols &= (N_MASK << (queen_index+1));
                  REMOVE_QUEEN_AT(queen_index, curr_row,
                     ↪ queens_in_columns, l_smem, diagonal,
                     ↪ antidiagonal);
```

```
27                }
28                if (!free_cols) {
29                   --curr_row;
30                }
31                else {
32                   const int col = intrin_ffs_nosub(free_cols);
33                   PLACE_QUEEN_AT(col, curr_row,
                        ↪ queens_in_columns, l_smem, diagonal,
                        ↪ antidiagonal);
34                   if (curr_row < N - 1)
35                      ++curr_row;
36                done: break;
37                }
38             }
39          }
40          __syncwarp();
41          if (res) {
42             while (l_smem[curr_row] == UNSET_QUEEN_INDEX) {
43                const bitset32_t free_cols = (~(queens_in_columns
                     ↪ | dad_extract_explicit(diagonal,
                     ↪ antidiagonal, curr_row)) & N_MASK);
44                const int POPCNT(free_cols, popcnt);
45                if (popcnt == 1) {
46 #ifdef NQ_ENABLE_EXPERIMENTAL_OPTIMISATIONS
47
48                   const int col =
                        ↪ intrin_find_leading_one_u32(free_cols);
49 #else
50                   const unsigned col = __ffs(free_cols) - 1;
51 #endif
52                   PLACE_QUEEN_AT(col, curr_row,
                        ↪ queens_in_columns, l_smem, diagonal,
                        ↪ antidiagonal);
53                   if (curr_row < N - 1) ++curr_row;
54                }
55                else break;
56             }
57          }
58          __syncwarp();
59 #if defined(NQ_ENABLE_EXPERIMENTAL_OPTIMISATIONS) &&
    ↪ defined(USE_64_BIT_RESULT_COUNTERS)
60          static_assert(sizeof(queens_in_columns) == 4,
               ↪ "Experimental optimisation: queens_in_columns
               ↪ MUST be a 32 bit type.");
61          static_assert(sizeof(t_sols) == 8, "Experimental
               ↪ optimisation: t_sols MUST be a 64 bit type.");
62          asm("{\n\t"
63             ".reg .pred %p;\n\t"
64             "setp.eq.u32 %p, %1, %2;\n\t"
```

```
65              "@%p add.u64 %0, %0, 1;\n"
66          "}" : "+l"(t_sols) : "r" (queens_in_columns) ,
                ↪ "r"(N_MASK));
67  #else
68          t_sols += (queens_in_columns == N_MASK);
69  #endif
70      } while (1); //Infinite loop has termination conditions
            ↪ and modifies global state -- no risk of compiler
            ↪ elimination.
71  #ifdef ENABLE_STATIC_HALF_SEARCHSPACE_REFLECTION_ELIMINATION
72      t_sols <<= mult_res_2;
73  #endif
74    }
75  #ifdef USE_64_BIT_RESULT_COUNTERS
76    atomicAdd(&sols[blockIdx.x], t_sols);
77  #else
78    __syncthreads();
79    t_sols = block_reduce_sum_shfl_variwarp((unsigned)t_sols,
        ↪ (unsigned int*)
        ↪ &smem[COMPLETE_KERNEL_BLOCK_THREAD_COUNT * N]);
80
81    if (!local_idx)
82      sols[blockIdx.x] += t_sols;
83  #endif
84  }
```

Listing C.13: Register-based kernel implementation of DoubleSweep-Light.

C.6 *N*-Queens State Difficulty Assessment Example

As described in Section 4.5.1, our current approach to obtaining a rough understanding of the difficulty of a state is based on the number of search steps (i.e., advancements and backtracks) involved in finding one solution. The range of such difficulty is generally greater for the more 'complex' states (i.e., those generated for larger values of *N*, especially those with more pre-completed columns. An alternative example to that shown in the aforementioned section is that shown in Figures C.1 and C.2. In this instance the former requires one queen placement (11^{th} column of the 8^{th} row) followed by a propagation by `DoubleSweep-Light` before requiring another placement (2^{nd} column of the 13^{th} row) followed by another run of `DoubleSweep-Light` to derive placements in the remaining five rows. The latter example however, required 844 advancements, backtracking 541 times in the process before reaching a solution.

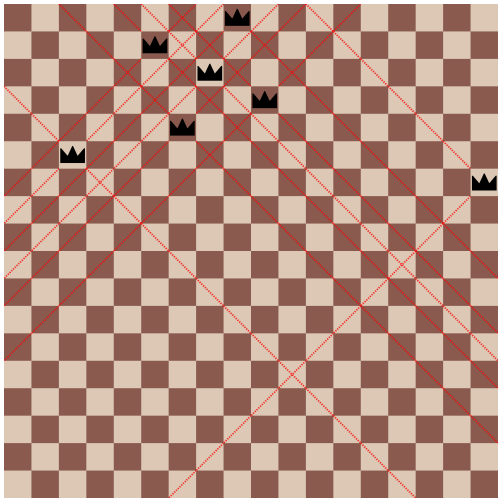


Figure C.1: Incomplete *N*-Queens state requiring 2 advancements and no backtracks to complete.

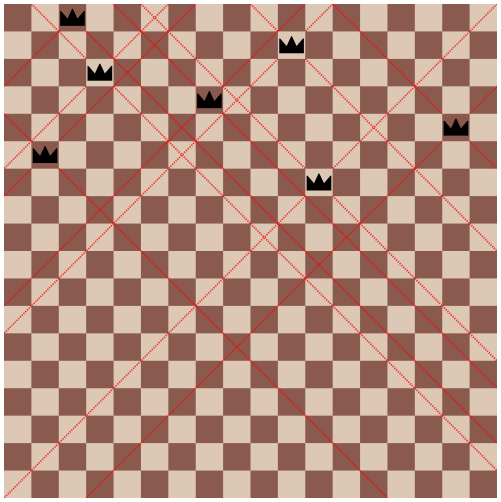


Figure C.2: Incomplete *N*-Queens state requiring 844 advancements and 541 backtracks to complete.

Bibliography

[1] Apr. 2023. URL: https://web.archive.org/web/20230416035400/http://curation.cs.manchester.ac.uk/computer50/www.computer50.org/mark1/new.baby.html.

[2] Hamdy Abdelkhalik et al. "Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis". In: *2022 IEEE High Performance & Extreme Computing Conference (HPEC)*. 2022. DOI: 10.1109/HPEC55821.2022.9926299.

[3] Bruce Abramson and Moti Yung. "Divide and conquer under global constraints: A solution to the N-queens problem". In: *Journal of parallel and distributed computing.* 6.3 (1989). ISSN: 0743-7315.

[4] Faisal N. Abu-Khzam et al. "A Hybrid Graph Representation for Recursive Backtracking Algorithms". In: *Frontiers in Algorithmics*. Ed. by Der-Tsai Lee, Danny Z. Chen, and Shi Ying. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 136–147. ISBN: 978-3-642-14553-7.

[5] SangWoo An and Seog SEO. "Highly Efficient Implementation of Block Ciphers on Graphic Processing Units for Massively Large Data". In: *Applied Sciences* 10 (May 2020), p. 3711.

[6] Sean Eron Anderson. URL: https://web.archive.org/web/20240402062019/https://graphics.stanford.edu/~seander/bithacks.html.

[7] Richard Ansorge. "Introduction to GPU Kernels and Hardware". In: *Programming in Parallel with CUDA: A Practical Guide*. Ed. by Richard Ansorge. Cambridge University Press, 2022, pp. 1–20. ISBN: 978-1-108-47953-0.

[8] Richard Ansorge, ed. *Programming in Parallel with CUDA: A Practical Guide*. Cambridge University Press, 2022, p. 395. ISBN: 978-1-108-47953-0.

[9] Richard Ansorge. "Warps and Cooperative Groups". In: *Programming in Parallel with CUDA: A Practical Guide*. Ed. by Richard Ansorge. Cambridge University Press, 2022, pp. 72–105. ISBN: 978-1-108-47953-0.

[10] Ramnik Arora, Rupesh Tulshyan, and Kalyanmoy Deb. "Parallelization of binary and real-coded genetic algorithms on GPU using CUDA". In: *IEEE Congress on Evolutionary Computation*. 2010, pp. 1–8. DOI: 10.1109/CEC.2010.5586260.

[11] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. "A linear-time algorithm for testing the truth of certain quantified boolean formulas". In: *Inform. Process. Lett.* 8.3 (1979), pp. 121–123. ISSN: 0020-0190. DOI: 10.1016/0020-0190(79)90002-4.

[12] *Atos delivers BullSequana X410 supercomputer to Swansea University*. Press Release. River Ouest, 80 quai Voltaire, 95877 Bezons cedex: Atos Group, 2021.

[13] Tomás Balyo and Carsten Sinz. "Parallel Satisfiability". In: *Handbook of Parallel Constraint Reasoning*. Ed. by Youssef Hamadi and Lakhdar Sais. Springer, 2018. Chap. 1, pp. 1–23. ISBN: 978-3-319-63515-6. DOI: 10.1007/978-3-319-63516-3_1.

[14] Tomáš Balyo, Peter Sanders, and Carsten Sinz. "HordeSat: A Massively Parallel Portfolio SAT Solver". In: *Theory and Applications of Satisfiability Testing – SAT 2015*. Ed. by Marijn Heule and Sean Weaver. Cham: Springer International Publishing, 2015, pp. 156–172. ISBN: 978-3-319-24318-4.

[15] Jordan Bell and Brett Stevens. "A survey of known results and research areas for n-Queens". In: *Universitext.* (2009), pp. 1–31. ISSN: 0012-365X.

[16] Armin Biere et al., eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009, p. 980. ISBN: 978-1-58603-929-5.

[17] Richard P. Brent. "Some Parallel Algorithms for Integer Factorisation". In: *Euro-Par'99 Parallel Processing*. Ed. by Patrick Amestoy et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1–22. ISBN: 978-3-540-48311-3.

[18] Curtis Bright et al. "Effective Problem Solving Using SAT Solvers". In: *Maple in Mathematics Education and Research*. Ed. by Jürgen Gerhard and Ilias Kotsireas. Cham: Springer International Publishing, 2020, pp. 205–219. ISBN: 978-3-030-41258-6.

[19] Federico Busato and Nicola Bombieri. "A performance, power, and energy efficiency analysis of load balancing techniques for GPUs". In: *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2017, pp. 1–8. DOI: 10.1109/SIES.2017.7993387.

[20] Paul J. Campbell. "Gauss and the eight queens problem: A study in miniature of the propagation of historical error". In: *Hist. Math.* 4.4 (1977), pp. 397–404. ISSN: 0315-0860. DOI: 10.1016/0315-0860(77)90076-3.

[21] Tiago Carneiro Pessoa et al. “GPU-accelerated backtracking using CUDA Dynamic Parallelism”. In: *Concurrency and Computation: Practice and Experience* 30.9 (2018). e4374 cpe.4374, e4374. DOI: 10.1002/cpe.4374.

[22] Daniel Cederman and Philippas Tsigas. “GPU-Quicksort: A practical Quicksort algorithm for graphics processors”. In: *ACM J. Exp. Algorithmics* 14 (Jan. 2010). ISSN: 1084-6654. DOI: 10.1145/1498698.1564500.

[23] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. “Where the really hard problems are”. In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI’91. Sydney, New South Wales, Australia: Morgan Kaufmann Publishers Inc., 1991, pp. 331–337. ISBN: 1558601600.

[24] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. “Confidence-Based Work Stealing in Parallel Constraint Programming”. In: *Principles and Practice of Constraint Programming - CP 2009*. Ed. by Ian P. Gent. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 226–241. ISBN: 978-3-642-04244-7.

[25] *Clang: a C language family frontend for LLVM*. https://web.archive.org/web/20240420082431/https://clang.llvm.org/. Apr. 2024.

[26] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. “The MPI Message Passing Interface Standard”. In: *Programming Environments for Massively Parallel Distributed Systems*. Ed. by Karsten M. Decker and René M. Rehmann. Basel: Birkhäuser Basel, 1994, pp. 213–218. ISBN: 978-3-0348-8534-8.

[27] SATComp Organizing committee. *The International Sat Competition Web Page*. URL: http://www.satcompetition.org/.

[28] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC)* (1971), pp. 151–158. DOI: 10.1145/800157.805047.

[29] Nadia Creignou et al. “A complexity theory for hard enumeration problems”. In: *Discrete Appl. Math.* 268 (2019), pp. 191–209. ISSN: 0166-218X. DOI: 10.1016/j.dam.2019.02.025.

[30] Alessandro Dal Palù et al. “CUD@SAT: SAT solving on GPUs”. In: *J. Exp. Theor. Artif. In.* 27 (2014).

[31] Evgeny Dantsin and Edward A. Hirsch. “Worst-Case Upper Bounds”. In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009. Chap. 12, pp. 403–424. ISBN: 978-1-58603-929-5. DOI: 10.3233/978-1-58603-929-5-403.

[32] Martin Davis, George Logemann, and Donald Loveland. “A Machine Program for Theorem-Proving”. In: *Commun. ACM* 5.7 (1962).

[33] Hervé Deleau, Christophe Jaillet, and Michaël Krajecki. "GPU 4 SAT : solving the SAT problem on GPU". In: 2008. URL: https://www.academia.edu/20928445/GPU4SAT_solving_the_SAT_problem_on_GPU.

[34] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-24605-3.

[35] Jan Elffers et al. "Seeking practical CDCL insights from theoretical SAT benchmarks". In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. IJCAI'18. Stockholm, Sweden: AAAI Press, 2018, pp. 1300–1308. ISBN: 9780999241127.

[36] C. Erbas, M.M. Tanik, and V.S.S. Nair. "A circulant matrix based approach to storage schemes for parallel memory systems". In: *Proceedings of 1993 5th IEEE Symposium on Parallel and Distributed Processing*. 1993, pp. 92–99. DOI: 10.1109/SPDP.1993.395546.

[37] Frank Feinbube et al. "NQueens on CUDA: Optimization issues". In: Ninth International Symposium on Parallel and Distributed Computing. 2010, pp. 63–70. DOI: 10.1109/ISPDC.2010.22.

[38] Israel Figueroa. *NQueens@Home*. https://web.archive.org/web/20091130024643/http://nqueens.ing.udec.cl/. 2007.

[39] Ronald Aylmer Fisher and Frank Yates. "Statistical tables for biological, agricultural and Medical Research". In: Sixth. Oliver & Boyd., 1963, pp. 37–37.

[40] Michael Flynn. "Flynn's Taxonomy". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, July 2011, pp. 689–697. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4.

[41] Sean L Forman and Alberto M Segre. "NAGSAT: A randomized, complete, parallel solver for 3-SAT". In: *Fifth international symposium on the theory and applications of satisfiability testing*. 2002, pp. 236–243.

[42] Stefano Gallo Giorgionand Pallottino. "Shortest path algorithms". In: *Ann. Oper. Res.* 13.1 (Dec. 1988), pp. 1–79. ISSN: 1572-9338. DOI: 10.1007/BF02288320.

[43] Ian P. Gent, Christopher Jefferson, and Peter Nightingale. "Complexity of n-Queens completion". In: *J. Artif. Int. Res.* 59.1 (May 2017), pp. 815–848. ISSN: 1076-9757.

[44] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. "Chapter 10 - Symmetry in Constraint Programming". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 329–376. DOI: https://doi.org/10.1016/S1574-6526(06)80014-3.

[45] Suklav Ghosh and Sarbajit Manna. "A Non-Recursive Space-Efficient Blind Approach to Find All Possible Solutions to the N-Queens Problem". In: *Innovations in Data Analytics*. Ed. by Abhishek Bhattacharya et al. Singapore: Springer Nature Singapore, 2023, pp. 327–339. ISBN: 978-981-99-0550-8.

[46] Stefan Glock, David Munhá Correia, and Benny Sudakov. "The n-Queens completion problem". In: *Res. Math. Sci.* 9.3 (July 2022), p. 41. ISSN: 2197-9847. DOI: 10.1007/s40687-022-00335-1.

[47] Matt Godbolt. *Compiler explorer*. May 2012. URL: https://godbolt.org/.

[48] Carla P Gomes, Ashish Sabharwal, and Bart Selman. "Model counting". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009. Chap. 20, pp. 633–651. ISBN: 978-1-58603-929-5. DOI: 10.3233/978-1-58603-929-5-633.

[49] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. "ManySAT: a parallel SAT solver". In: *JSAT* 6 (June 2009), pp. 245–262. DOI: 10.3233/SAT190070.

[50] Bagus Hanindhito and Lizy K. John. "Accelerating ML Workloads using GPU Tensor Cores: The Good, the Bad, and the Ugly". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 178–189. DOI: 10.1145/3629526.3653835. URL: https://doi.org/10.1145/3629526.3653835.

[51] Wim H. Hesselink and Mark IJbema. "Starvation-free mutual exclusion with semaphores". In: *Form. Asp. Comput.* 25.6 (Nov. 2013), pp. 947–969. ISSN: 1433-299X. DOI: 10.1007/s00165-011-0219-y.

[52] Marijn J. H. Heule and Hans van Maaren. "Look-Ahead Based SAT Solvers". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009. Chap. 5, pp. 155–184. ISBN: 978-1-58603-929-5. DOI: 10.3233/978-1-58603-929-5-155.

[53] Marijn J. H. Heule et al. "Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads". In: *Hardware and Software: Verification and Testing*. Ed. by Kerstin Eder, João Lourenço, and Onn Shehory. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 50–65. ISBN: 978-3-642-34188-5.

[54] Juraj Hromkovič, ed. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Springer, Mar. 2013. ISBN: 978-3-642-07909-2. DOI: 10.1007/978-3-642-07909-2.

[55] Juraj Hromkovič. "Complexity Theory". In: *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Ed. by Juraj Hromkovič. Springer, Mar. 2013, pp. 107–128. ISBN: 978-3-642-07909-2. DOI: 10.1007/978-3-642-07909-2.

[56] Juraj Hromkovič. "Solving 3SAT in Less than 2^n Complexity". In: *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Ed. by Juraj Hromkovič. Springer, Mar. 2013, pp. 169–173. ISBN: 978-3-642-07909-2. DOI: 10.1007/978-3-642-07909-2.

[57] Microsoft Inc. *C and C++ in Visual Studio*. https://web.archive.org/web/20240328031154/https://learn.microsoft.com/en-us/cpp/overview/visual-cpp-in-visual-studio?view=msvc-170. Sept. 2022.

[58] ISO. *ISO C Standard 1999, ISO/IEC 9899:1999*. Tech. rep. 1999. URL: https://web.archive.org/web/20240505000952/https://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.

[59] Phillip James, Faron Moller, and Filippos Pantekis. "OnTrack: Reflecting on domain specific formal methods for railway designs". In: *Sci. Comput. Program.* 233 (2024), p. 103057. ISSN: 0167-6423. DOI: 10.1016/j.scico.2023.103057.

[60] Phillip James et al. "OnTrack: An Open Tooling Environment for Railway Verification". In: *NASA Formal Methods*. Ed. by Guillaume Brat, Neha Rungta, and Arnaud Venet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 435–440. ISBN: 978-3-642-38088-4.

[61] John Jenkins et al. "Lessons Learned from Exploring the Backtracking Paradigm on the GPU". In: *Euro-Par 2011 Parallel Processing*. Ed. by Emmanuel Jeannot, Raymond Namyst, and Jean Roman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 425–437. ISBN: 978-3-642-23397-5.

[62] Zhe Jia et al. *Dissecting the NVidia Turing T4 GPU via Microbenchmarking*. 2019. DOI: 10.48550/arxiv.1903.07486. arXiv: 1903.07486 [cs.DC].

[63] Zhe Jia et al. *Dissecting the Turing GPU Architecture through Microbenchmarking*. CITADEL. Mar. 18, 2019. URL: https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9839-discovering-the-turing-t4-gpu-architecture-with-microbenchmarks.pdf (visited on 04/20/2023).

[64] Cao Jianli et al. “Parallel genetic algorithm for N-Queens problem based on message passing interface-compute unified device architecture”. In: *VDI Bericht* 36.4 (2020), pp. 1621–1637. DOI: 10.1111/coin.12300.

[65] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Springer, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9.

[66] Richard M. Karp and Yanjun Zhang. “Randomized parallel algorithms for backtrack search and branch-and-bound computation”. In: *J. ACM* 40.3 (July 1993), pp. 765–789. ISSN: 0004-5411. DOI: 10.1145/174130.174145.

[67] Paul Kelly. “The stored program concept and the Turing Tax”. https://web.archive.org/web/20221212205034/https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/Lectures/pdfs/Ch01-part4-TuringTaxDiscussion.pdf. 2022.

[68] Yuma Kikuchi et al. “Calculation of Cross-correlation Function Accelerated by Tensor Cores with TensorFloat-32 Precision on Ampere GPU”. In: *Computational Science – ICCS 2022*. Ed. by Derek Groen et al. Cham: Springer International Publishing, 2022, pp. 277–290. ISBN: 978-3-031-08754-7.

[69] Kenji Kise et al. “Solving the 24-queens Problem using MPI on a PC Cluster”. In: *Graduate School of Information Systems, The University of Electro-Communications, Tech. Rep* (2004).

[70] Hans-Peter Kriegel, Erich Schubert, and Arthur Zimek. “The (black) art of runtime evaluation: Are we comparing algorithms or implementations?” In: *Knowl. Inf. Syst.* 52.2 (Aug. 2017), pp. 341–378. ISSN: 0219-3116. DOI: 10.1007/s10115-016-1004-2.

[71] Tom Krüger, Jan-Hendrik Lorenz, and Florian Wörz. “Too much information: Why CDCL solvers need to forget learned clauses”. In: *PLOS ONE* 17.8 (Aug. 2022), pp. 1–28. DOI: 10.1371/journal.pone.0272967. URL: https://doi.org/10.1371/journal.pone.0272967.

[72] Oliver Kullmann. “Fundaments of Branching Heuristics”. In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009. Chap. 7, pp. 205–244. ISBN: 978-1-58603-929-5. DOI: 10.3233/978-1-58603-929-5-205.

[73] Vipin Kumar. “Algorithms for Constraint-Satisfaction Problems: A Survey”. In: *AI Magazine* 13.1 (1992), pp. 32–44. DOI: 10.1609/aimag.v13i1.976.

[74] Massimo Lauria et al. “CNFgen: A Generator of Crafted Benchmarks”. In: *Theory and Applications of Satisfiability Testing*. Springer, 2017.

[75] Matthew Lewis, Tobias Schubert, and Bernd Becker. "Multithreaded SAT Solving". In: *2007 Asia and South Pacific Design Automation Conference*. 2007, pp. 926–931. DOI: 10.1109/ASPDAC.2007.358108.

[76] Chu Min Li and Felip Manyá. "MaxSAT". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009. Chap. 19, pp. 613–628. ISBN: 978-1-58603-929-5. DOI: 10.3233/978-1-58603-929-5-613.

[77] Weile Luo et al. *Benchmarking and Dissecting the Nvidia Hopper GPU Architecture*. 2024. DOI: 10.48550/arxiv.2402.13499. arXiv: 2402.13499 [cs.AR].

[78] Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. "Embarrassingly parallel search in constraint programming". In: *J. Artif. Int. Res.* 57.1 (Sept. 2016), pp. 421–464. ISSN: 1076-9757.

[79] Norbert Manthey. *Parallel SAT Solving - Using More Cores*. Tech. rep. TU Dresden, Knowledge Representation and Reasoning, 2011.

[80] Joao Marques-Silva. "Practical applications of Boolean Satisfiability". In: *2008 9th International Workshop on Discrete Event Systems*. 2008, pp. 74–80. DOI: 10.1109/WODES.2008.4605925.

[81] Ruben Martins, Vasco Manquinho, and Ines Lynce. "Improving Search Space Splitting for Parallel SAT Solving". In: *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*. Vol. 1. 2010, pp. 336–343. DOI: 10.1109/ICTAI.2010.56.

[82] Ruben Martins, Vasco Manquinho, and Inundefineds Lynce. "An Overview of Parallel SAT Solving". In: *Constraints* 17.3 (July 2012), pp. 304–347. ISSN: 1383-7133. DOI: 10.1007/s10601-012-9121-3.

[83] Quirin Meyer et al. "3-SAT on CUDA: Towards a massively parallel SAT solver". In: *2010 International Conference on High Performance Computing & Simulation*. International Conference on High Performance Computing & Simulation (HPCS). 2010, pp. 306–313. DOI: 10.1109/HPCS.2010.5547116.

[84] Chao Ming-Te and John Franco. "Probabilistic analysis of a generalization of the unit-clause literal selection heuristics for the k satisfiability problem". In: *Information Sciences* 51.3 (1990), pp. 289–314. ISSN: 0020-0255.

[85] David Mitchell, Bart Selman, and Hector Levesque. "Hard and easy distributions of SAT problems". In: *Proceedings of the Tenth National Conference on Artificial Intelligence*. AAAI'92. San Jose, California: AAAI Press, 1992, pp. 459–465. ISBN: 0262510634.

[86] M. W. Moskewicz et al. “Chaff: engineering an efficient SAT solver”. In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 2001, pp. 530–535.

[87] Ravie Muniyandi and All Maroosi. “Enhancing the simulation of membrane system on the GPU for the N-Queens problem”. In: *Chinese J. Electron.* 24 (2015), pp. 740–743. DOI: 10.1049/cje.2015.10.012.

[88] NVIDIA. *CUDA Binary Utilities 12.4*. https://web.archive.org/web/20240324124217/https://docs.nvidia.com/cuda/cuda-binary-utilities/. Mar. 2024.

[89] NVIDIA. *CUDA C++ Programming Guide Release 12.4*. https://web.archive.org/web/20240324062313/https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Mar. 2024.

[90] NVIDIA. *Parallel Thread Execution ISA Version 8.4*. https://web.archive.org/web/20240324062329/https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Mar. 2024.

[91] *NVIDIA Ampere GA102 GPU Architecture*. Whitepaper. 2788 San Tomas Expressway Santa Clara, CA. 95050: NVIDIA, 2020.

[92] *NVIDIA H100 Tensor Core GPU Architecture*. Whitepaper. 2788 San Tomas Expressway Santa Clara, CA. 95050: NVIDIA, 2024.

[93] *NVIDIA NVLink TM high speed interconnect: Application performance*. Whitepaper. 2788 San Tomas Expressway Santa Clara, CA. 95050: NVIDIA, 2014.

[94] *NVIDIA Tesla P100*. Whitepaper. 2788 San Tomas Expressway Santa Clara, CA. 95050: NVIDIA, 2016.

[95] *NVIDIA Tesla V100 GPU Architecture*. Whitepaper. 2788 San Tomas Expressway Santa Clara, CA. 95050: NVIDIA, 2017.

[96] *NVIDIA Turing GPU Architecture*. Whitepaper. 2788 San Tomas Expressway Santa Clara, CA. 95050: NVIDIA, 2018.

[97] Ignacio Sañudo Olmedo et al. “Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective”. In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 213–225. DOI: 10.1109/RTAS48715.2020.000-5.

[98] Muhammad Osama and Anton Wijs. “SIGmA: GPU Accelerated Simplification of SAT Formulas”. In: *Integrated Formal Methods*. Ed. by Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa. International Conference on Integrated Formal Methods. Cham: Springer International Publishing, 2019, pp. 514–522. ISBN: 978-3-030-34968-4.

[99] Muhammad Osama, Anton Wijs, and Armin Biere. "SAT Solving with GPU Accelerated Inprocessing". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Cham: Springer International Publishing, 2021, pp. 133–151. ISBN: 978-3-030-72016-2.

[100] Filippos Pantekis and Phillip James. "Towards Massively Parallel GPU Assisted SAT". In: *2022 Tenth International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE Computer Society, Nov. 2022, pp. 120–126. DOI: 10.1109/CANDARW57323.2022.00080.

[101] Filippos Pantekis, Phillip James, and Oliver Kullmann. "Scalable N-Queens Solving on GPGPUs via Interwarp Collaborations". In: Tenth International Symposium on Computing and Networking (CANDAR). Himeji, Japan: IEEE, 2022, pp. 158–164. DOI: 10.1109/CANDAR57322.2022.00029.

[102] Filippos Pantekis et al. "Optimised Massively Parallel Solving of N-Queens on GPGPUs". In: *Concurrency and Computation: Practice and Experience* (2023). DOI: 10.1002/cpe.8004.

[103] Filippos Pantekis et al. "Visualising Railway Safety Verification". In: *Formal Techniques for Safety-Critical Systems*. Ed. by Osman Hasan and Frédéric Mallet. Springer International Publishing, 2020, pp. 95–105. ISBN: 978-3-030-46902-3.

[104] *Parallelization of SAT Algorithms on GPUs*. Technical Report. Technical University of Lisbon, 2013. URL: https://api.semanticscholar.org/CorpusID:16436811.

[105] Jon Peddie. "Introduction". In: *The History of the GPU - Steps to Invention*. Cham: Springer International Publishing, 2022, pp. 1–30. ISBN: 978-3-031-10968-3. DOI: 10.1007/978-3-031-10968-3_1.

[106] "Peer-to-peer for computational grids: mixing clusters and desktop machines". In: *Parallel Comput.* 33.4 (2007). Large Scale Grids, pp. 275–288. ISSN: 0167-8191. DOI: 10.1016/j.parco.2007.02.011.

[107] Tiago Pessoa et al. "Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators". In: Feb. 2021.

[108] Max Plauth et al. "Using Dynamic Parallelism for Fine-Grained, Irregular Workloads: A Case Study of the N-Queens Problem". In: *2015 Third International Symposium on Computing and Networking (CANDAR)*. 2015, pp. 404–407. DOI: 10.1109/CANDAR.2015.26.

[109] Stephen Plaza, Igor Markov, and Valeria Bertacco. "Low-latency SAT Solving on Multicore Processors with Priority Scheduling and XOR Partitioning". In: (Jan. 2008).

[110] Steven Prestwich. "CNF Encodings". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009. Chap. 2, pp. 75–97. ISBN: 978-1-58603-929-5. DOI: 10.3233/978-1-58603-929-5-75.

[111] Thomas B. Preußer and Matthias R. Engelhardt. "Putting Queens in Carry Chains, №27". In: *J. Signal Process. Syst.* 88.2 (2017), pp. 185–201. DOI: 10.1007/s11265-016-1176-8.

[112] Thomas B. Preußer, Bernd Nagel, and Rainer G. Spallek. *Putting queens in carry chains*. Technical Report. Fakultat Informatik, Technische Universitat Dresden, 2013.

[113] Albert Reuther et al. "Survey of Machine Learning Accelerators". In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 2020, pp. 1–12. DOI: 10.1109/HPEC43674.2020.9286149.

[114] Ahmed Samara and James Tuck. "The Case for Domain-Specialized Branch Predictors for Graph-Processing". In: *IEEE Comput. Archit. Lett.* 19.2 (2020), pp. 101–104. DOI: 10.1109/LCA.2020.3005895.

[115] Tobias Schubert, Matthew Lewis, and Bernd Becker. "PaMiraXT: Parallel SAT solving with threads and message passing". In: *JSAT* 6 (June 2009), pp. 203–222. DOI: 10.3233/SAT190068.

[116] R. Shonkwiler, F. Ghannadian, and C.O. Alford. "Parallel simulated annealing for the n-queen problem". In: *[1993] Proceedings Seventh International Parallel Processing Symposium*. 1993, pp. 690–694. DOI: 10.1109/IPPS.1993.262797.

[117] Michael Simkin. "The number of n-Queens configurations". In: *Adv. Math.* 427 (2023), p. 109127. ISSN: 0001-8708. DOI: 10.1016/j.aim.2023.109127.

[118] Neil J. A. Sloane and The OEIS Foundation Inc. *The on-line encyclopedia of integer sequences*. 2020. URL: http://oeis.org/?language=english.

[119] Jeff Somers. *The N Queens Problem: a study in optimization*. http://users.rcn.com/liusomers/nqueen_demo/nqueens.html.

[120] Rok Sosic and Jun Gu. "A polynomial time algorithm for the N-Queens problem". In: *SIGART Bulletin* 1.3 (1990), pp. 7–11. ISSN: 0163-5719. DOI: 10.1145/101340.101343.

[121] Wei Sun et al. "Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors". In: *IEEE Trans. Parallel Distrib. Syst.* 34.1 (2023), pp. 246–261. DOI: 10.1109/TPDS.2022.3217824.

[122] Chico Sundermann et al. “Applications of #SAT Solvers on Feature Models”. In: *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS ’21. Krems, Austria: Association for Computing Machinery, 2021. ISBN: 9781450388245. DOI: 10.1145/3442391.3442404.

[123] GCC Team. *GCC, the GNU Compiler Collection*. https://web.archive.org/web/20240304084921/https://gcc.gnu.org/. Mar. 2023.

[124] GCC Team. *Optimize options (using the GNU compiler collection (GCC))*. https://web.archive.org/web/20240417084018/https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. June 2021.

[125] Jason Thong and Nicola Nicolici. “SAT solving using FPGA-based heterogeneous computing”. In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2015, pp. 232–239. DOI: 10.1109/ICCAD.2015.7372575.

[126] Krishnahari Thouti and S. R. Sathe. “Solving N-Queens problem on GPU architecture using OpenCL with special reference to synchronization issues”. In: 2nd IEEE International Conference on Parallel, Distributed and Grid Computing. 2012, pp. 806–810. DOI: 10.1109/PDGC.2012.6449926.

[127] G. S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28.

[128] Shunsuke Tsukiyama et al. “Solving the N-Queens Puzzle by a QUBO Model with Quadratic Size”. In: *2023 Eleventh International Symposium on Computing and Networking (CANDAR)*. 2023, pp. 59–67. DOI: 10.1109/CANDAR60563.2023.00015.

[129] Stanley Tzeng, Brandon Lloyd, and John D. Owens. “A GPU task-parallel model with dependency resolution”. In: *Computer* 45.8 (2012), pp. 34–41. DOI: 10.1109/MC.2012.255.

[130] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[131] Vasily Volkov and James W. Demmel. “Benchmarking GPUs to tune dense linear algebra”. In: *SC ’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 2008, pp. 1–11. DOI: 10.1109/SC.2008.5214359.

[132] Christof Vömel, Stanimire Tomov, and Jack Dongarra. "Divide and Conquer on Hybrid GPU-Accelerated Multicore Systems". In: *SIAM J. Sci. Comput.* 34.2 (2012), pp. C70–C82. DOI: 10.1137/100806783.

[133] Yangzihao Wang et al. "Gunrock: GPU Graph Analytics". In: *ACM Trans. Parallel Comput.* 4.1 (Aug. 2017). ISSN: 2329-4949. DOI: 10.1145/3108140.

[134] Anton Wijs, Thomas Neele, and Dragan Bošnački. "GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking". In: *Formal Methods 2016*. Ed. by John Fitzgerald et al. Springer, 2016.

[135] Xiaolong Xie et al. "CRAT: Enabling Coordinated Register Allocation and Thread-Level Parallelism Optimization for GPUs". In: *IEEE Transactions on Computers* 67.6 (2018), pp. 890–897. DOI: 10.1109/TC.2017.2776272.

[136] A. M. Yaglom and I. M. Yaglom. *Challenging Mathematical Problems with Elementary Solutions*. Dover Publications, 1987, pp. 92–98. ISBN: 0486655369.

[137] Yinlei Yu et al. "All-SAT Using Minimal Blocking Clauses". In: *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. 2014, pp. 86–91. DOI: 10.1109/VLSID.2014.22.

[138] Zhongda Yuan, Yuchun Ma, and Jinian Bian. "SMPP: Generic SAT Solver over Reconfigurable Hardware Accelerator". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 2012.

[139] Jilian Zhang, Kyriakos Mouratidis, and HweeHwa Pang. "Heuristic Algorithms for Balanced Multi-Way Number Partitioning." In: *IJCAI International Joint Conference on Artificial Intelligence*. Jan. 2011, pp. 693–698. DOI: 10.5591/978-1-57735-516-8/IJCAI11-122.

[140] Lintao Zhang et al. "Efficient conflict driven learning in a Boolean satisfiability solver". In: *IEEE/ACM International Conference on Computer Aided Design*. 2001, pp. 279–285. DOI: 10.1109/ICCAD.2001.968634.

[141] Peter Zhang et al. "Dynamic parallelism for simple and efficient GPU graph algorithms". In: *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. IA^3 '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340014. DOI: 10.1145/2833179.2833189.

[142] Tao Zhang, Wei Shu, and Min-You Wu. "Optimization of N-Queens solvers on graphics processors". In: *Advanced Parallel Processing Technologies*. International Workshop on Advanced Parallel Processing Technologies. Springer, 2011, pp. 142–156. ISBN: 978-3-642-24151-2.

[143] Jia Zhe et al. *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking*. Online. 2018. DOI: 10.48550/arxiv.1804.06826. arXiv: 1804.06826 [cs.DC].

[144] P. Zhong, M. Martonosi, and P. Ashar. “FPGA-based SAT solver architecture with near-zero synthesis and layout overhead”. English. In: *IEE Proceedings - Computers and Digital Techniques* 147 (3 May 2000), pp. 135–141. ISSN: 1350-2387. DOI: 10.1049/ip-cdt_20000482.