Check for updates

Received: 30 June 2023 | Revised: 7 November 2023 | Accepted: 12 December 2023

DOI: 10.1002/cpe.8004

SPECIAL ISSUE PAPER

WILEY

Optimized massively parallel solving of N-Queens on GPGPUs

Filippos Pantekis | **Phillip James** | **Oliver Kullmann** | **Liam O'Reilly**

Department of Computer Science, Swansea University, Swansea, UK

Correspondence
Filippos Pantekis, Department of Computer Science, Swansea University, Swansea, UK.
Email: filippos.pantekis@swansea.ac.uk

Funding information
Engineering and Physical Sciences Research Council, Grant/Award Number: EP/S015523/1; Swansea University

Summary

Continuous evolution and improvement of GPGPUs has significantly broadened areas of application. The massively parallel platform they offer, paired with the high efficiency of performing certain operations, opens many questions on the development of suitable techniques and algorithms. In this work, we present a novel algorithm and create a massively parallel, GPGPU-based solver for enumerating solutions of the N-Queens problem. We discuss two implementations of our algorithm for GPGPUs and provide insights on the optimizations we applied. We also evaluate the performance of our approach and compare our work to existing literature, showing a clear reduction in computational time.

KEYWORDS

GPGPUs, GPGPU optimization, massively parallel, N-Queens

1 | INTRODUCTION

In recent years, general purpose graphics processing units (GPGPUs) have arguably become a prevalent part of both the parallelization and high performance computing communities due to the massively parallel platform they offer, and their superior capabilities in performing certain operations more efficiently than other hardware. However, it is important to note that GPGPUs do not substitute 'traditional' CPUs and their surrounding environments, as there are many architectural differences between the two. This necessitates the need to revisit and adapt existing algorithms or to approach problems from a different angle all together. Crucially, tailored algorithms must be matched by performant implementations with carefully targeted optimizations to achieve optimal performance.

A variety of problems have been successfully mapped onto GPGPUs and benefit from the varying capabilities of such devices. The N-Queens problem is no exception. N-Queens is a long standing mathematical problem initially proposed as the Eight Queens problem in 1848,[2] where the challenge was to find the total number of non-attacking arrangements of eight queens on an 8×8 chessboard. In the general form, the objective of the N-Queens problem is to identify the number of non-attacking placements of N queens on an $N \times N$ board, yet solutions are currently only known for $N \in [1, 27]$ due to the dramatic increase in combinations for larger values of N, and the associated increase in computational cost.

In this work, we present a novel algorithm for solving the N-Queens problem that is well suited to parallelization, and discuss adaptations made to appropriate it for implementation on GPGPUs. In particular, we give details of two kernel implementations of this algorithm that are suited to modern GPGPU architectures. These implementations form our state-of-the-art, distributed, GPGPU-based, N-Queens solver. Through the discussion of our implementation choices, we offer in depth insights on the optimizations we applied and their appropriacy over different hardware architectures of NVIDIA devices, as well as specifics of work partitioning and distribution. Finally, we present runtimes for our solver obtained from both a GPU cluster and more localized hardware which surpass those available in literature, and examine opportunities for further performance gains. This work is a continuation of our earlier contribution,[1] where we presented initial results from a more rudimentary iteration of our solver.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.
© 2024 The Authors. *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

2 | BACKGROUND

We first review the N-Queens problem in more depth, before discussing modern GPGPU architectures and features upon which we build.

2.1 | 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 on a conventional (8×8) chessboard exist.[2] Figure 1 illustrates an example of a non-attacking configuration, which is one of the 92 non-attacking configurations (solutions) for $N = 8$. This problem was later generalised [2] to the N-Queens problem as known today.

A distinction should be made on the alternative formulation that is sometimes used for the N-Queens problem. Some literature[3,4] quotes a variant, which is that of discovering a *single* non-attacking configuration of N queens for an $N \times N$ board. For this work, we consider the original variant of the generalized problem, namely that of enumerating all solutions for a given value of N (and in principle even outputting each solution).

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, 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' algorithms and their efficient implementations. Real-world applications for the N-Queens problem are documented in literature,[5] such as very large-scale integration (VLSI) testing and deadlock prevention. Importantly, algorithms for constraint satisfaction problems such as N-Queens, and their implementations, can be applied to problems of a similar nature.

As of yet, the number of non-attacking configurations is known for all $N \in [1, 27]$, with the latest addition being that of $N = 27$.[6] It is noteworthy that for all of the higher values of N, the solutions were discovered using parallel algorithms on distributed systems.[6–9] This problem lends itself to parallelization, due to the triviality of deciding if a configuration is non-attacking or not, paired with the vast number of candidate solutions to be checked.

2.2 | Computing on GPGPUs

NVIDIA's compute unified device architecture (CUDA) [10] brings support for general-purpose computation on supported NVIDIA Graphics Processing Units (GPUs) through a programming interface, drivers, and various tools. GPGPUs expose their superior mathematical capabilities and massively parallel environment for use in 'generic' tasks*.

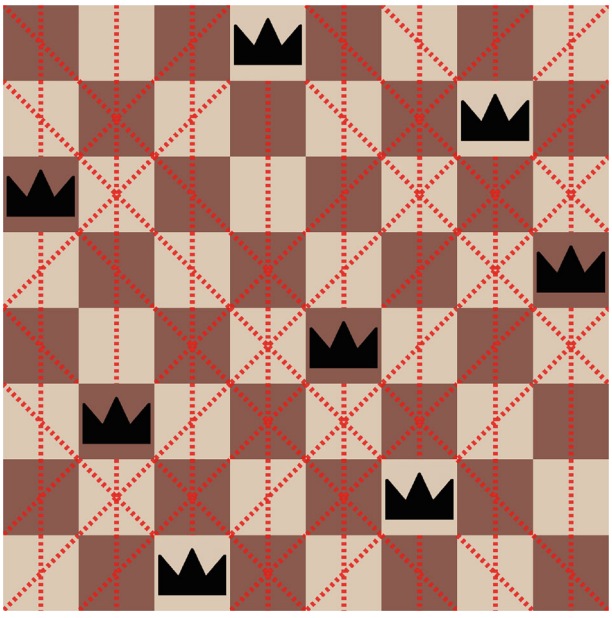

FIGURE 1 A non-attacking configuration of 8 queens on an 8×8 board shown with the per-queen attack indicators.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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 the devices associated with it, which are executed by a number of threads. The threads 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 into a grid which can also be either mono-, bi- or tri-dimensional. The dimensionality of the blocks and grid is important for tasks exhibiting spatial locality, however, for other tasks this geometry is of little significance.

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 factors such as memory requirements and configuration, block size, and other hardware-specifics.

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. Thread divergence should be avoided as it typically reduces the number of threads executing in parallel in the warp, increasing the overall time required for the computation to be performed. Section 2.2.1 discusses recent changes in hardware which enable greater flexibility with respect to diverging threads in warps.

Threads on their own are 'weak' to compute as a unit, but power is leveraged from the large number of concurrent threads at any given moment in a massively parallel environment such as this. 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 threads across all blocks. This type of memory is the most costly to access (in terms of clock cycles) even when coalescing requirements are met, but it is typically the largest memory type available. It 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. Additionally, a number of 32-bit registers are available to each thread with thread-scope. Using warp-level primitives, it is possible for threads within the same warp to efficiently perform collective operations involving communication by sharing register contents.

Shared memory is divided across a number of banks. Whilst 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 serialized, 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 degrade performance significantly.

Typically, communication between host and device happens through memory transfers over a shared bus. 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.[11]

Commonly, GPUs are either used to assist in the computational effort of a solver running on the host (hybrid solver[12]), or act as solvers themselves coordinated by the host (complete solver[13]). In a hybrid solver, 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 solvers 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 optimized for the GPU environment. Consequently, new techniques and adaptations to the algorithm(s) are likely necessary to achieve a good mapping.

In terms of the programming model, a number of programming languages are supported by the CUDA toolkit. Our work is primarily using C, with a number of lower-level optimizations detailed in Section 5.5. The translation from the high-level programming language to GPU machine instructions is a multi-step process. Initially, high-level C code is compiled into the PTX (Parallel Thread Execution) instruction set which is an assembly-like language, abstracting away hardware details and using 'register variables' in place of registers. PTX instructions can then be compiled into SASS[‡] assembly through a process which maps the PTX code to device-specific SASS code and performs operations such as register allocation. The details on how this conversion is performed, along with the SASS Instruction Set Architecture (ISA) are mostly undocumented and the produced instructions are intended not to be modified. It is therefore very difficult to gauge post-compilation specifics, such as register pressure from the PTX level.

2.2.1 | Comparison of GPGPU architectures

GPGPU technology is being continually improved in order to increase computational performance and to introduce expanded hardware support for a wider range of operations.

NVIDIA names different GPU generations after famous scientists such as Pascal, Turing, Ampere, and so forth. Devices of each generation are classified by their Compute Capability (CC) version number, which identifies the particular features that the device supports. For a generation, there may be several such versions implemented by different hardware.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

This evolution of hardware results in a high degree of volatility which often comes at a cost to optimization approaches. For instance, the Volta architecture supports, for each SM, up to 2048 resident threads with up to 32 registers per thread and 96 KB of shared memory.[14] Its successor, Turing, halves the maximum number of resident threads, keeping the same register file size which leaves 64 registers per thread, and also reduces the size of available shared memory to 64 KB per SM.[15] This changes once again by the succeeding architecture, Ampere, where a total of 164 KB of shared memory is available to a maximum of 2048 threads per SM, each of which can access 32 registers.[16] In short, not forward compatible for optimization. This volatility often binds optimizations to particular generations relying on the characteristics of the architecture, which may render them ineffective or even a hindrance on others.[17]

Register use per thread is dependent on a multitude of factors, one of which is the subset of instructions involved in the computations and the combinations thereof. The evolution of hardware sometimes introduces support for specialized operations via a single instruction which would otherwise be constructed using multiple instructions with intermediate results. For instance, the Volta architecture introduces support for 32-bit mask creation in the form of the `bmsk` instruction in the PTX ISA. Whilst the specifics of hardware performance and the translation of such instructions to SASS is not publicized, it is safe to assume their purpose is to optimize specific operations for the benefit of the overall computation–an assumption supported by literature.[18,19]

Further to the above, NVIDIA introduced 'Independent Thread Scheduling' since the Volta architecture which, to some degree, mitigates the effects of warp divergence albeit at the cost of a register. Prior to Volta, threads in a warp shared a single program counter, but since Volta, each thread maintains an independent program counter and its own stack space. More specifically, in this architecture, schedule optimizers are introduced, which group threads of a warp which diverged into sub-units of threads which are in sync with each other, to be run in parallel. Warp divergence should still be avoided in current architectures as it may still have a profound effect on performance.

Of course, different generations of GPUs are better suited for different tasks, however, software optimizations aiming to get closer to optimal performance have to be done with a high degree of device-specificity. As Feinbube et al. highlight [17] compilers and programming languages alone are not sufficient to achieve optimal performance. They observe some of their optimizations being a detriment to performance on some architectures and beneficial on others. We show that this continues to be the case with current architectures and is a view we echo through our work.

2.2.2 | Scalability on distributed GPGPU systems

The model of computation described in Section 2.2, is host-centric as it works under the assumption that a host computer shares a bus with the devices (GPUs) attached to it, and is coordinating the computation in one or more devices. Devices attached to the host machine may use the shared bus to communicate if a direct link between them is absent, at the cost of lower data transfer rates.

A host system is limited in the number of devices it can house (vertical scaling), therefore increasing computational power beyond this point is achieved by increasing the number of such host systems (nodes) participating in the computation (horizontal scaling). Communication becomes an even greater concern for horizontal scaling, as communication between devices on these nodes is subject to even more restrictions. Commonly, the Message Passing Interface (MPI) [20] is utilized in such cases, for which CUDA-aware implementations exist, but the mediums involved may introduce additional overheads and slow down the communication process, which is the reason we chose not to use it in our implementation.

Work partitioning across devices is typically coordinated manually and influences the design of device-side code to mitigate any performance penalty. However, loosely coupled workloads with no cross-device communication requirements do not have to account for such communication penalties and form an ideal scenario. A problem may be partitioned into several sub-problems [21] which can be tackled individually and independently of one another to form such a workload, which is the approach we opted for as detailed in Section 4.1.

3 | APPROACHES TO SOLVING N-QUEENS

We review related work in the field focusing in particular on parallel approaches for solving the N-Queens problem, and present the `DoubleSweep` algorithm from which the backbone of our solver is derived.

3.1 | Related work on parallel N-Queens solving

The N-Queens problem has been approached from a plethora of angles throughout its existence. When the aim is to identify a single non-attacking configuration for a given value of N, several techniques have been explored which do so in a 'fast' manner using search-based algorithms [4,22–24] or by applying a pattern directly.[3]

Estimating the number of non-attacking configurations (solutions) for any N was recently proven possible.[25] However, the effort of identifying the exact number of non-attacking configurations remains significant and requires brute-force-like algorithms with search-space limiting heuristics.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Often such algorithms rely on backtracking and are based around what is known as Somers' Algorithm, which we will discuss at length in Section 3.2. The theoretical fundamentals of divide-and-conquer approaches for the problem have also been explored deeply [26] that further evidences that this style of algorithm is suitable for the N-Queens problem. Such algorithms treat the board as a 'ladder' upon which a search is performed, recursively attempting to place a queen in a valid position on each rung, and backtracking when dead-ends are reached.

With all such approaches, the search space needed for enumeration is super-exponential as claimed in Reference 25, namely the limit $\lim_{N\to\infty} \frac{Q(N)^{1/N}}{N} \approx 0.143$ where $Q(N)$ is the number of solutions. Note that this is the minimum complexity and heuristics are needed to eliminate fruitless paths. The effectiveness of such heuristics can be observed and contributes to the reduction of the search space, placing higher values of N within the realm of possibility. The performance of implementations for such algorithms and respective heuristics is of paramount importance, which puts focus on the parallelization and optimization of such implementations.

Highly parallel approaches have been used in distributed environments to identify the number of solutions for instances of the N-Queens problem such as $N = 24$, which was first calculated by Kise et al. [7] using a 34-node cluster of CPUs, or the later work of Caromel et al. [8] who solved the $N = 25$ instance using a grid of 260 machines. Both approaches serve as good examples of highly parallel approaches and underline the difficulty of the task at hand.

Utilization of GPUs in this effort has already been proven successful with numerous examples available.[17,27–30] These attempts are the product of heavy optimization of implementations and the adaptation of algorithms such as the aforementioned Somers' algorithm, to account for the specialities of the GPU environment, and achieve good performance. The need for such bespoke optimizations arises from the 'irregularity' of the computation at hand relative to the expectation of a certain computation structure of GPUs.[27]

Another challenge faced in the use of GPUs with this type of problem is the rapid evolution of hardware which often creates a mismatch between optimization techniques and evolving architectures. Likewise, tooling provided for this hardware may not have matured enough to make optimal use of the hardware.[17] Architecture-specific optimizations beyond what compilers offer are therefore quite common and are also a prominent component in our work, discussed in Section 5.5.

Of course, GPU hardware is not the only hardware relevant to the N-Queens problem. Previous work has successfully utilized Field Programmable Gate Array (FPGA) hardware. Most recently, the work of Preußer et al. [6] used a naïve search algorithm with a search space limiting heuristic to discover, over the course of a year, the latest solution to the N-Queens problem, namely $N = 27$. This result remains unverified to date but is nevertheless a significant achievement [9] that follows the earlier achievements of the authors in calculating $N = 26$, once again using FPGA hardware. These two latest results highlight the shift in focus away from conventional algorithms and the 'traditional' computational model into parallel algorithms and less conventional highly parallel computational hardware. General purpose GPUs provide several advantages over FPGAs for such computations, such as their relative ease of programming which renders them applicable to a multitude of tasks, without special re-programming requirements, as well as their wide-spread availability as consumer hardware and conventional cloud computing platforms owed in part to reasonable prices driven by mass production.

3.1.1 | Applications of GPGPUs to similar problems

N-Queens is often seen as a 'benchmark' problem, acting as a proxy for developing techniques, algorithms, and optimizations that apply to other problems similar in nature. The problem of Boolean satisfiability [31] (SAT) is a notable example which has received the attention of the GPGPU community. Successful attempts have been made in harnessing the capabilities of GPUs to accelerate the solving of SAT instances [12] as well as developing GPGPU-based SAT solvers.[13,32]

Recursive algorithms often do not map directly to the massively parallel environment of GPUs meaning that significant adjustments have to be made to any implementation of such algorithms. Meyer et al. [32] present an implementation of a recursive divide-and-conquer algorithm for solving 3-SAT instances by decomposing the implementation of the algorithm to a pipeline of six kernels, each with a single function. This stepped design breaks the recursive algorithm into major components, each of which is performed by multiple threads at once with little chance for divergence between them, and with global synchronization enforced by the host.

3.2 | The `DoubleSweep` algorithm

Perhaps a more common approach to enumerating solutions for the N-Queens problem, is using a backtracking search over all possible placements of queens on the board. Such algorithms typically begin by placing a queen in the first row of an N-Queens board, and recursively exploring subsequent rows, making a non-attacking queen placement on each, until either the board is completed, or a row is found where no such placement can be made. In the latter case, the search backtracks to a row where the queen can be moved to a different position, and resumes from that point. Notably, Jeff Somers [33] provided an efficient iterative implementation for such an algorithm, representing the state of the board in part using three 32-bit words,

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

tracking the columns blocked by queens, viewing them only as rooks, and a further two to track blocked diagonals and anti-diagonals in the current state of the board.

The `DoubleSweep` algorithm combines basic word-level parallelism with basic ideas of look-ahead techniques [34,35] from the domain of Boolean satisfiability (SAT) solving. A key difference 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. This process identifies rows with only one open cell left, whereby a queen placement must be made, and repeats following every successful placement until a fixed-point, or a row or column with no possible placements is identified. Another key difference here is that `DoubleSweep` begins placements in the central row of the board rather than the first (top-most) row. This branching heuristic helps make the propagation step more efficient, as central placements are more influential to the remaining rows.

`DoubleSweep` uses *N* words to represent the full board with current propagations on top of the three words used by Somers. In addition, the (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 Section 5.3.

The algorithm introduced in this paper is `DoubleSweep-Light`, which is a lighter version of `DoubleSweep`: unlike Somers' algorithm, branching starts at the top row, and proceeds (only) to the next row, while the data-structure of `DoubleSweep` is used only for one sweep down without iteration until the first row is found with at least two open cells or the end of the board is reached as illustrated by Figure 2. In light of the high complexity of implementing such dynamic algorithms on GPUs in an efficient manner, this light version, which compromises on some of `DoubleSweep`'s advantages, was chosen as a means of balancing algorithmic complexity with the need for an optimized implementation for our solver.

4 | `DOUBLESWEEP-LIGHT`–A GPU-CENTRIC APPROACH

The `DoubleSweep` algorithm presented in Section 3.2 contains a number of features which are powerful, but incur significant costs when implemented for GPUs. Perhaps most significant is the branching nature of the algorithm which introduces further data-dependent branches in an implementation. As such, we have made a number of adaptations and created the `DoubleSweep-Light` algorithm, to bridge the divide of algorithmic performance and feasible optimizations in implementation. Implementation details of this algorithm for the massively parallel GPU environment are detailed in Section 5.

`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

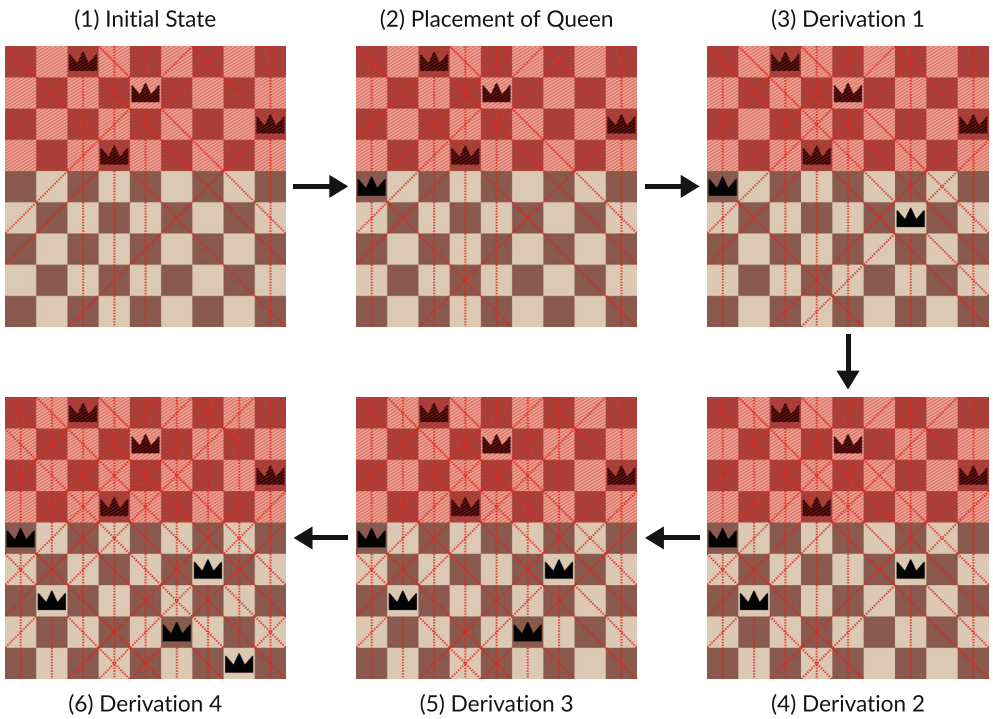

FIGURE 2 Step-by-step application of `DoubleSweep-Light`.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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 (referred to as a 'state') and a backtracking limit (i.e., the index of a row beyond which backtracking is not permitted), the function `advance_state` presented in Listing 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 4.2.

It is worth noting that the form a state (`s`) takes in Listing 1 and subsequent listings is that of a structure, the contents of which include the current row (`current_row`) 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 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 1: State advancement 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 2. This function identifies which column (if any) in the current row of `s` is free (Lines 3– 8) and if such column exists, places a queen, repeating the same operation in the following row (Lines 9– 11).

```
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 2: The `DoubleSweep-Light` algorithm.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The process described in this section is presented visually in Figure 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) which should not be modified (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., there is only one position available), 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.

4.1 | Parallelizing `DoubleSweep-Light`

A crucial component in the design of this algorithm is its dependency-free parallelizability, whereby parallel search paths have no reason to converge or share information between them.

As `DoubleSweep-Light` is a backtracking-based algorithm, its exploration of the search space can be visualized as a tree of candidate configurations, as shown in Figure 3. Parallelizing this algorithm across disjoint searches can be achieved by splitting the search tree into a forest of disjoint sub-trees. The sub-trees produced can then be operated upon by a number of parallel `DoubleSweep-Light` searches, without risk of overlap or dependencies between the searches. More specifically, a certain depth (level) of the search tree may be chosen, the sub-trees of which can be used as starting points for non-converging parallel searches.

It must be noted that the first level of the tree in Figure 3 has been reduced to just four states, as the removed states are symmetries of the remaining states on the vertical axis. This part of the algorithm is further discussed in Section 4.2.

As discussed in Section 4, `DoubleSweep-Light` attempts to complete the placement of queens on a partly complete non-attacking configuration. Therefore, to parallelize the search, it is enough to construct a pool of incomplete non-attacking configurations which act as starting points for parallel search paths. Section 4.2 presents the specifics of this process.

Besides the diversification of the search via unique search paths, `DoubleSweep-Light` is designed to require no awareness of parallel searches or previously explored paths. This is particularly important to maintain the loosely coupled nature of the solver and not limit its horizontal scalability.

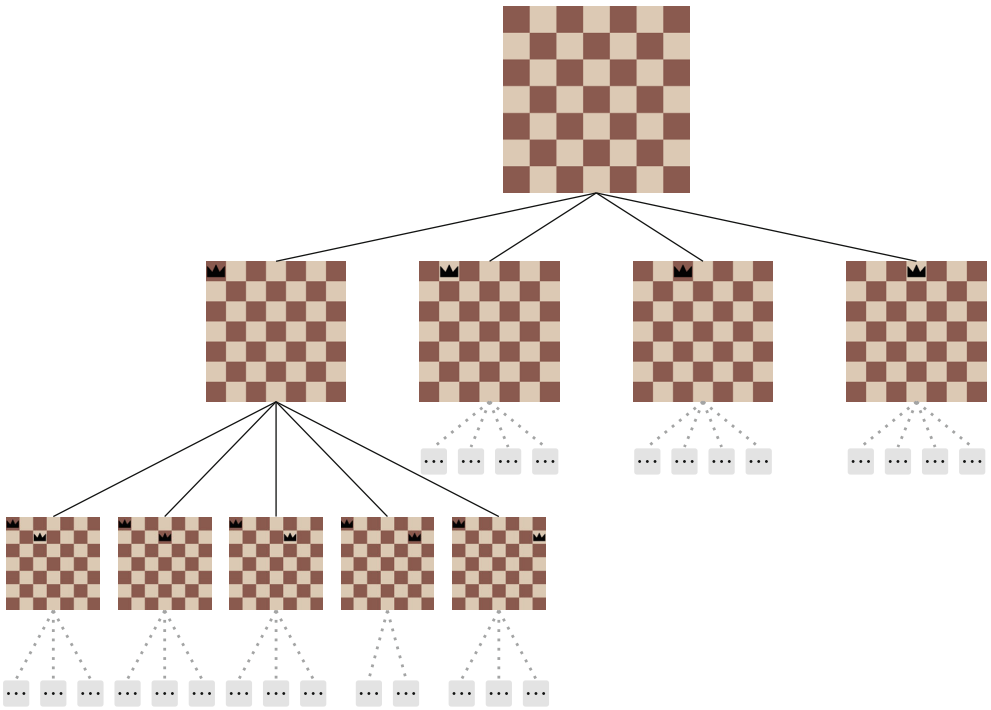

FIGURE 3 Visualization of search tree for `DoubleSweep-Light`.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.2 | 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 1 takes into consideration the index of the last locked row, on or prior to which backtracking must not occur.

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 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_1 is subsequently generated by applying a modified version of `DoubleSweep-Light` (seen as the function `gen_states`) which stops placing/deriving queens after a certain row (Lines 3– 8).

If $|S_1| < s$, r is incremented and state generation is repeated, until some generated pool S_i satisfies $|S_i| > r$, then 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 on 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 essence, solving the given instance. For larger values of N, this is very unlikely to happen as generally, an immense number of states would have to be generated before the search plateaus and enters the downward slope.

```
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 3: Ladder-climbing algorithm for initial state pool generation.

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. For the generated states, the number of solutions found under each would have to be doubled, unless N is an odd integer, in which case the solutions are doubled for all states except those with a queen in the $\left\lceil \frac{N}{2} \right\rceil^{\text{th}}$ column.

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 quick process, which has stringent memory requirements that the GPU environment is not suited for. Instead, we implemented this process on the host-side. Our implementation on the host-side is parallelised across $\left\lceil \frac{N}{2} \right\rceil$ many threads, in the same fashion explained in Section 4.1. More specifically, each thread t_i begins by placing a queen on the i^{th} column of the first row in its (initially empty) state, which remains untouched throughout the state generation process.

The pool of initial states generated needs to be sufficiently large to supply every participating solver. Dependent upon the number of nodes, devices, the maximum number of concurrent threads, and the over-subscription factor per device (discussed further in Section 5), a very large pool

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

could be produced having high memory requirements that exceed the available memory of the system. State generation is implemented with the option of using secondary storage as memory.

After generation, the pool of states is shuffled randomly using the Fisher-Yates shuffle algorithm.[36] This step is important to counter the effects of irregular work distribution caused by some states having more possible candidate solutions than others. By shuffling the pool randomly concentrations of 'heavier' states are broken, meaning partitions given to devices are likely to contain a more uniformly balanced workload. Further discussion on this point can be found in Section 6.

5 | IMPLEMENTATION ON GPUS

For our implementation of `DoubleSweep-Light`, we used CUDA-C as the high-level language with targeted optimizations through inline assembly code where appropriate. The details of the optimizations we applied are presented in Section 5.5. The lightweight design of `DoubleSweep-Light` paired with its algorithmic efficiency allows it to be implemented in a constrained environment, such as a GPU thread. Each GPU thread acts as an isolated `DoubleSweep-Light` solver, which operates on a different starting point to the rest.

The search begins with a pool of states being generated (as described in Section 4.2). A subset of the states in the pool is then transferred to the GPU, and the search kernel is launched with at least as many threads as states in the pool. The cumulative number of threads in the grid is typically significantly higher than the number of cores in the GPU. This degree of over-subscription reflects on the number of blocks in each kernel and allows for finer balance of work across SMs. This workload does not benefit from spatial locality, therefore we have grouped threads in mono-dimensional blocks which in turn are part of a mono-dimensional grid.

We have implemented two `DoubleSweep-Light` kernels, one relying on shared memory and one exploiting the superior performance of registers, described in Sections 5.1 and 5.2 respectively.

5.1 | Shared memory-based kernel

Shared memory offers significantly cheaper access costs than global memory as discussed in Section 2.2. The shared memory-based implementation of `DoubleSweep-Light` has each thread in each block reserve a portion of shared memory for its computation exclusively. In the beginning, the thread transfers its corresponding state to shared memory, which is represented as a struct comprised of the following information:

1. Per-row projections of conflicting diagonals caused by placed queens (two 64-bit bit vectors, see Section 5.3).
2. The occupied columns (i.e., placed queens, a single 32-bit bit vector).
3. The index of the current row (a single 8-bit integer).
4. The indexes of placed queens on the current state (array of N 8-bit integers).

Having the indexes of the queens on the board renders the other pieces of data, such as the occupied columns bit vector, redundant, as they can be derived. However, this data plays a crucial role in the implementation explained below, trading memory for reduced repeated computation.

The per-row indexes are necessary in this instance to facilitate backtracking. During backtracking, placed queens need to be removed from the state, meaning the tracking of diagonals, antidiagonals 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.

The size of this struct varies depending on the value of N (known at compile time), as a result of the array member. The remaining components are 8-byte aligned and laid out as shown in Figure 4 with the 8-byte boundaries highlighted. This structure minimizes 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 as well as shared 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.

During computation, each thread in a warp operates on the data in its shared memory, by performing the `DoubleSweep-Light` procedure as outlined in Section 4. Warp divergence cannot be eliminated completely in the implementation of `DoubleSweep-Light` as the number of iterations

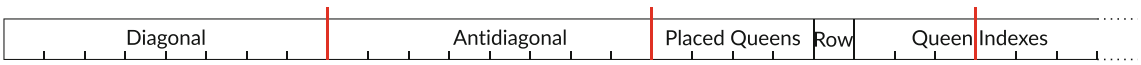

FIGURE 4 Layout of per-thread data in shared memory.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

made by each thread is data-dependent. Warp synchronization barriers are interleaved between state advancement and propagation in an effort to re-converge divergent threads where appropriate.

Following every successful propagation and derivation, threads within a warp vote to determine if all threads involved 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 the case. 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 optimization discussed in Section 5.5.

After all threads in a warp vote to vacate, the individual results of each are accumulated in a common global memory location, through an atomic add operation. At the end of the computation, this location (known to the host) contains the number of solutions found for the set of states given to that device. It is the job of the host system(s) to collect results across devices and accumulate them. Formerly, we implemented result accumulation using a warp shuffling operation, however, such operations are currently not available for 64-bit types, and we deem them as offering insignificant performance gains w.r.t. the overall computation.

Reliance on shared memory does in some cases impact the number of threads per block. Our goal is to maximize the utilization 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'

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

The number of blocks in the grid is 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 maximize 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 so global memory access coalescing is not considered in this instance.

5.2 | Register exploitation for memory latency reduction

As discussed in Section 2.2, register space can be viewed as the fastest 'memory space' available. Whilst registers are not addressable memory, adjustments to the kernel described in Section 5.1 give the opportunity to the compiler to place this data in registers. We have applied these changes to form a register-based kernel implementing `DoubleSweep-Light`.

Members of the state struct of each thread are loaded directly into thread-local variables. Surrounding code has been adjusted to implement `DoubleSweep-Light` using these variables instead of addressable memory, without significant change to the code flow. The only component of the struct which remains in shared memory is the array member as that can't be housed in registers.

Shared memory requirements are significantly reduced. For a given value of N, each thread requires N bytes of shared memory. Given the maximum number of threads per block b, number of threads in a warp w, and shared memory size s we calculate the number of threads per block as

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

The benefits of this change are two-fold; 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 1024 threads per block. Even the earlier architectures featuring 48,000 bytes of shared memory

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

per block per SM would end up having at least one full block per SM for all $N \leq 46$. Additionally, register space is thread-local and significantly faster and with less stringent access requirements 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 utilization are objectionable, as beyond violating programming standards, they often hinder compiler optimizations and overall result in performance loss. Likewise, exhaustion of register space has adverse effects on overall performance for reasons outlined in Section 2.2.1.

In our experiments, this kernel generally compiled without excess register usage and resulted in significant performance gains discussed in Section 6. We note however that registers are by no means a plentiful resource, rendering this kernel nonideal for some past and potentially future architectures. For instance, devices with compute capability 6.2 support 2048 threads resident on each SM at a time, sharing 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 presented in Section 2.2.1. During compilation, the compiler reports 25 registers being used by this kernel. Therefore, the decision on which kernel is better suited for the resources at hand should be made on a case-by-case basis.

5.3 | Diagonal tracking

As described in Section 5.1, each thread uses two 64-bit bit vectors to track the projections of diagonals placed in its state. Although these 16 bytes may seem a hefty sacrifice to make considering space constraints, this is an integral component of the `DoubleSweep-Light` implementation that allows us to effortlessly determine which columns are non-conflicting with diagonals from placed queens, for a given row.

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 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.

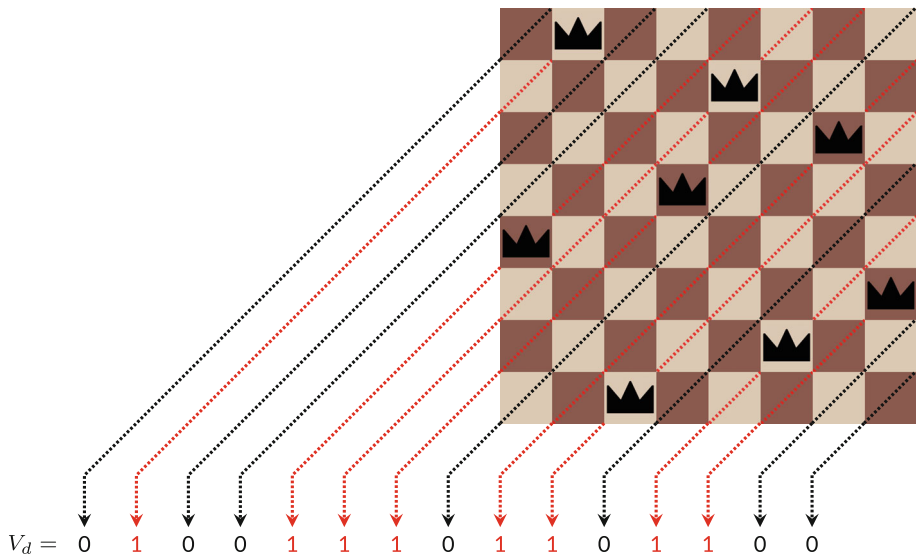

FIGURE 5 Mapping of queens on diagonals to the diagonal tracking word.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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 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 5.5) introduce hardware support for bit mask calculations.

5.4 | Impact of datatype conversions

Currently, NVIDIA GPUs feature 32-bit wide registers. Unlike shared memory, using types smaller than 32 bits in high-level code yields no benefit or reduction in register space requirements. Throughout experiments, we observed that the use of such types in fact came at a cost to performance. In many cases, their use forced the compiler to append type conversion instructions (`cvt`) in the resulting PTX code to enforce the properties of the respective type, which carried through to the SASS code.

A simple change of some types in 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 optimization, in compute-bound kernels such as this one, performance gains can be made through the removal of unnecessary instructions.

The CUDA Programming Guide [10] provides details on the throughput of arithmetic instructions including type conversions. It is noteworthy that in the latest compute capability versions as of yet ($8.0 \leq cc \leq 9.0$), only 16 conversions from 32-bit types to smaller ones can be performed per clock cycle per SM, potentially acting as a bottleneck.

5.5 | Reflections on architecture changes and optimization

Through the chain of evolution of NVIDIA architectures, several features have been introduced or removed, overall amounting to incremental performance gains and offering several advantages. Adversely however, such fluctuations in design and feature availability complicate the identification of areas where optimizations are applicable and hinder portability across architectures.

We have developed our solver using features of the CUDA library which are compatible with a large range of device architectures and opted to hand-tune the solver for specific architectures, namely Pascal, Turing, and Ampere. The process of identifying those optimizations is the product of reading the resulting PTX code to identify areas of improvement and performing isolated micro-benchmarks on alternative formulations of some operations. We note that our control from the higher-level CUDA-C code is limited to just introducing inline PTX assembly instructions, which may not reflect in the later translation to device-specific SASS instructions. To ensure optimizations had an effect, we analyzed the resulting PTX code as well as SASS instructions after compilation for the targeted architectures following the steps of Abdelkhalik et al.[18]

Optimizations typically arise from identifying opportunities for shortcuts the compiler did not take. For instance, during the propagation sweep of `DoubleSweep-Light`, we go over unpopulated rows and compute the bit word a of available columns for that row as described in Section 5.3. 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) set bit (one-indexed). The PTX code produced by the compiler for the `__ffs` function is shown in Listing 4 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: PTX instructions generated for the standard function `__ffs`.

In this instance, we know however 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 5, where we use `bfind.u32` to identify the natural index of the most significant (and only) set bit directly. This improvement is especially important

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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.[10]

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

Listing 5: Reduced set of instructions, equivalent of Listing 4.

Perhaps more frequently, hand-tuning opportunities arise from carefully analysing PTX instructions of frequently repeated sub-routines. In our solver, 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 6 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.[18]

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

Listing 6: PTX instructions generated by the compiler for per-thread result accumulation.

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 (branching) 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 7. In tested architectures, this optimization 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 optimized sub-routine in the computation, and the significance a small gain such as this can have in the overall computation.

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

Listing 7: Optimised equivalent of Listing 6.

Finally, we note that optimizations such as the aforementioned may have adverse effects in future versions of hardware or compilers, but currently deliver performance gains which are essential for our application, therefore we regard them as "experimental". Optimizations may also be enabled through the use of standard library features, such as the restructuring of non-aliasing pointers. The keyword `__restrict__` in CUDA-C has the same purpose as the C keyword `restrict` introduced in C99. Pointer declarations marked with this keyword give information to the compiler that they will never alias any other pointer. This information can be useful to the compiler in reordering load and stores and the order of operations in between to aid performance. We observed significant gains, particularly with the shared memory-based kernel presented in Section 5.1 by making use of this keyword.

5.6 | Work partitioning on multi-GPU distributed systems

We have designed our solver to scale on any number of parallel instances provided enough initial states can be generated, as described in Section 4.2, which renders it ideal for scaling horizontally in distributed environments such as clusters. Typically, clusters are comprised of any number of nodes,

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

each of which may be of different specifications to the rest, even at different physical locations. A job is typically submitted through the login node and scheduled automatically to run in a node when sufficient resources become available. Jobs hold various parameters for the work that will eventually be carried out and often limit the time available to carry out such work. The specifics of cluster topologies, available hardware, limits, job scheduling, and so forth, may vary greatly between clusters.

Overcoming time limits in such environments can be a challenge and is typically achieved through checkpointing. The application running as part of a job is thereby issued a signal by the job scheduler when the execution time limit is about to be reached, to perform the necessary actions and save the current state of the computation before exiting gracefully. For CPU-side computations, the signal can be handled by the main process to terminate the computation. However, this is more difficult for GPU-based applications since the host system has to signal the device to stop and return the partial progress made. This is further complicated in applications such as ours where there is no communication between the host system and the device throughout the computation, since all threads of the device act as isolated solvers, relying on on-chip memory resources only.

We decided against implementing this functionality, as the associated changes to the kernel would incur significant performance costs. We instead chose to partition the state pool empirically, for problem sizes likely to surpass time limits of the cluster, and instead dispatch multiple jobs to tackle each sub-pool.

6 | PERFORMANCE EVALUATION

To evaluate the performance of our `DoubleSweep-Light` implementations described in Section 5, we tackled a range of problem sizes $N \in [14, 25]$ on various systems with different GPU device architectures. In our earlier work [1] we presented solving times for $N \in [14, 20]$ shown in Table 1 obtained using the Shared memory-based implementation of `DoubleSweep-Light` described in Section 5.2. 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 note that at the time, the elimination of vertically symmetric states was not implemented.

In this work, we present results for both the register-based and shared memory-based implementations of `DoubleSweep-Light` we collected using Swansea University's GPU cluster. This cluster is comprised of six identical GPU nodes, each of which houses eight NVIDIA A100 GPUs (Ampere architecture). Unlike our earlier results and for reasons outlined in Section 5.6, 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, meaning they would occupy a full node.

Figure 6 presents the solving time in seconds required for each $N \in [19, 24]$ 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 4.2, and the two kernels were submitted as separate jobs to the cluster, each tasked with tackling that same input state pool. The sizes of pools generated for each experiment are shown in Table 2, alongside the index of the last row that was 'locked' to produce this number of initial states. 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. In some instances such as

TABLE 1 Solving times in milliseconds from our earlier work [1] for different values of N across a number of benchmark configurations.

N	1 × 1080ti (ms)	2 × 1080ti (ms)	1 × 3090 (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

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

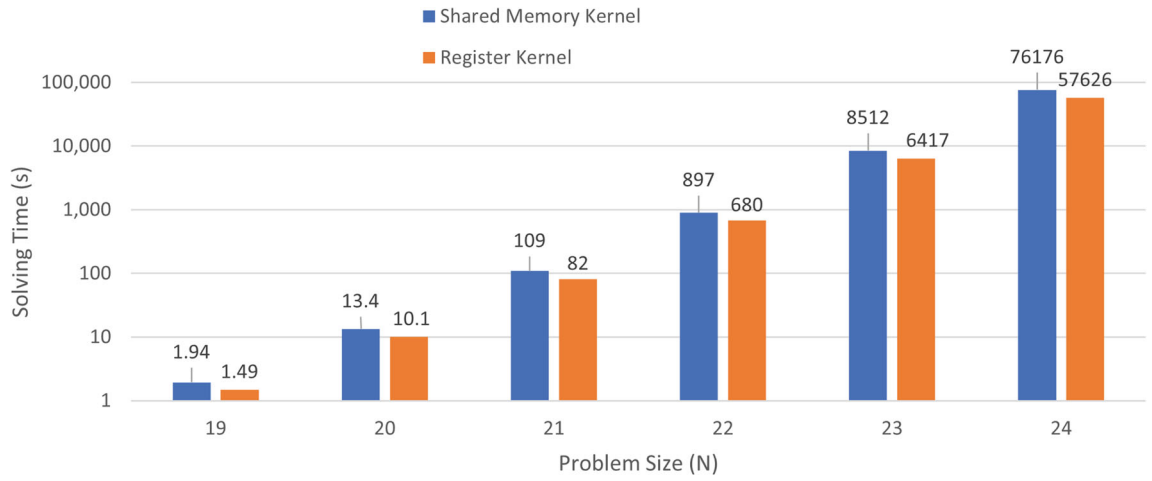

FIGURE 6 Solving time in seconds required to tackle each $N \in [19, 24]$ using both the register-based and shared memory-based kernels over eight A100 GPUs.

TABLE 2 Number of states in state pools generated for our experiments.

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

$N \in \{20, 23, 24\}$ the number of states generated was substantially smaller than 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 4.2. However, even in such instances, there were sufficient states available for each device, and despite potentially worse balancing of work between devices, the overall solving time does not appear to deviate significantly. The two kernels perform well and we observe that as expected, Kernel 1 is consistently slower than Kernel 2 for reasons identified in Section 5.1.

Varying the number of devices used to solve an instance of the problem shows a quasi-linear improvement in overall solving time. For instance, Figure 7 shows how the time needed to solve $N = 22$ varies with the number of A100 GPUs involved in the computation. In theory, there should be a linear improvement in solving time as more devices are involved, however, this is not the case; we attribute the quasi-linear trend to factors such as imperfect work balancing between devices. As mentioned earlier, it is not possible to accurately determine the solving time required per state, 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.

Results presented so far were collected via a single job submission that carried out all work. The cluster used for our results however imposes a strict 48-h limit on jobs to enforce fair resource sharing, which was insufficient time to collect results for $N = 25$. 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 $N = 24$ using eight A100 GPUs. With 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 into 20 sub-pools. Despite the considerable memory requirements of the state generation process which peaked at 1.75 TB of total memory usage (RAM and persistent storage as explained in Section 4.2), the overall state generation process lasted approximately 40 min.

To tackle these sub-pools of states, we manually scheduled 20 jobs over the course of 2 weeks, each using eight A100 GPUs and solving one sub-pool of states, and completed the computation in a combined 670,747 s of run time or approximately 1 week and 18 h. We launched individual jobs in succession with intervals between them to enable better sharing of resources with other users than what is implemented by the job scheduler. 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

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

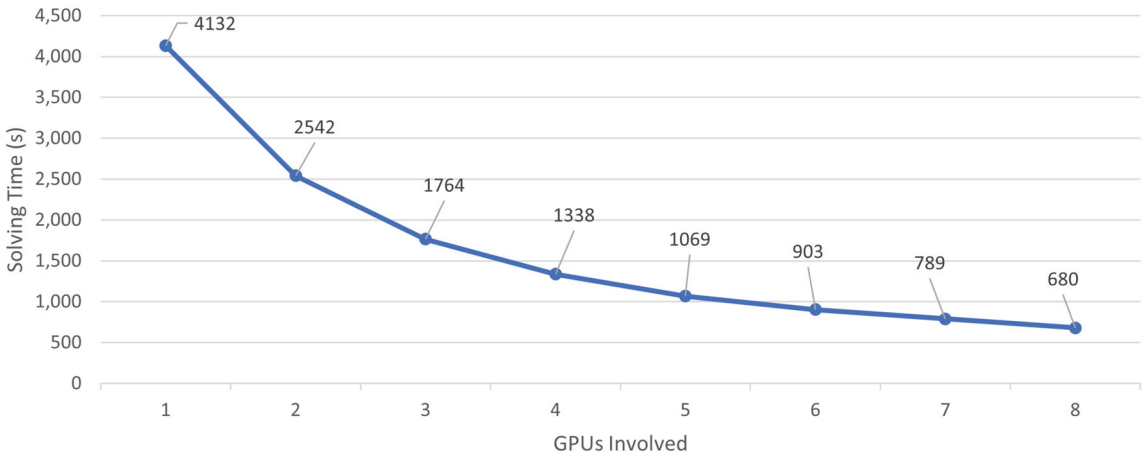

FIGURE 7 Impact of varying the number of devices (NVIDIA A100) involved in the computation of $N = 22$.

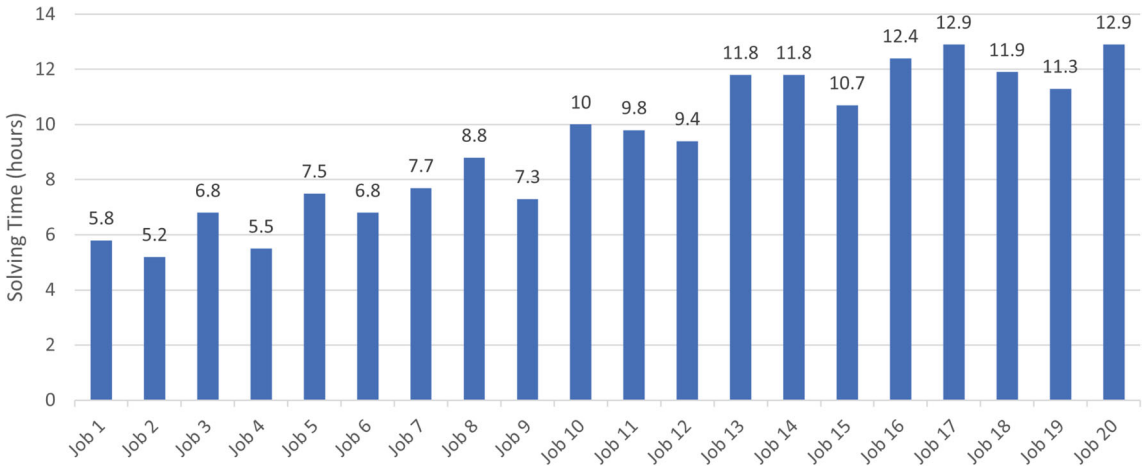

FIGURE 8 Time taken per job, for each of 20 jobs used to solve $N = 25$.

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 s or approximately 13 h.

Figure 8 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 can be a demanding process for such large data sets. We shuffled 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, which is the case. There are 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 have more possible solutions to be explored, dependent where the first few queens are placed.

Results presented in this paper show a significant improvement in performance over our earlier results, shown in Table 1. The performance gains were made by optimizing the state generation process to eliminate vertical reflections of generated states, introduction of the register-based kernel implementation, and several hand-tuning optimizations to the implementation to maximize the performance of both kernels. Use of 64-bit counters for result accumulation by each solver thread is also being made in our solver currently unlike our earlier work, the negative performance impact of which we have countered with hand-tuning optimizations.

The N-Queens problem has attracted the attention of the optimization and parallel processing communities, with several contributions in the literature of GPU-based solvers such as 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

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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,[17,27,29,30] 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, yet there are several more such transformations to consider. Some work in literature [17,27] reduces the search space further by eliminating more transformations, however, our approach appears to surpass these results even in its current state. We intend on examining suitable heuristic approaches to remove further transformations from the search space whilst maintaining high throughput in our GPU implementations.

Beyond GPUs, a number of other computing devices and arrangements have been used to tackle the N-Queens problem. Notably, Kise et al. [7] solved $N = 24$ in 2004 using a cluster of 68 Pentium4 Xeon processors, in approximately 22 days. Similarly, in 2007 Caromel et al. [8] utilized 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. [9] utilize Field Programmable Gate Arrays (FPGAs) to design and implement a solver that solved $N = 26$ in 270 days; work they later complemented [6] in 2017 by computing the result of $N = 27$, which is the largest problem instance solved to date and as of yet remains unverified. The fast-paced evolution of alternative computation hardware such as FPGAs and GPUs shows significant performance potential over 'traditional' CPUs and paves the way for further advances in this and many other fields.

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 $8.35^2 \times 670{,}747 \approx 46{,}766{,}158$ s, 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.,[6] who used a total of 14 FPGA boards varying in model for their computation.

7 | CONCLUSION AND FUTURE DIRECTIONS

In this paper, we present the most recent version of our GPU-based N-Queens solver which implements our `DoubleSweep-Light` algorithm. We explain the implementations of two kernels for `DoubleSweep-Light` and discuss how we achieved high performance, enabled by the design of both the algorithm and its implementation. Finally, we discuss our observations of cases where the compiler failed to identify some possible optimisations and explain how we intervened, underlining the gains such hand-tuning has to offer despite the tight architecture coupling it introduces. Our evaluation places our solver above previous work on N-Queens solving using GPUs, which paves the path for further success in the field. We hope the techniques presented as part of our work prove insightful for others seeking to design or implement similar algorithms for GPU hardware. We also hope that our work underlines the importance that architecture specifics play in high level optimization.

Looking at the future, it is our intention to continue developing the solver to examine techniques we can utilize to move closer to a full implementation of `DoubleSweep` and introduce complete symmetry elimination, without compromising on solving performance on the GPU, as currently, the `DoubleSweep` algorithm is implemented partially as `DoubleSweep-Light` in our solver. Furthermore, we plan to explore work-stealing options between devices on the host, in an effort to better distribute work for longer-running jobs, and achieve linear scaling in practice. More specifically, we will examine how a device that has completed its work may 'steal' a number of states from other devices to continue working alongside them. Lastly, we would like to explore heuristics for the initial state pool generation, to eliminate more fruitless paths and potentially classify states based on how likely they are to produce more solutions than the rest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ENDNOTES

*The terms GPU and GPGPU will be used interchangeably throughout this work.
†A block is resident when its threads are initialized and ready to execute.
‡The expansion of this acronym is ambiguous, and not documented officially.

ORCID

Filippos Pantekis https://orcid.org/0000-0001-7817-6450
Phillip James https://orcid.org/0000-0002-4307-649X
Oliver Kullmann https://orcid.org/0000-0003-3021-0095
Liam O'Reilly https://orcid.org/0000-0002-4894-2158

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

REFERENCES

1. Pantekis F, James P, Kullmann O. Scalable N-Queens solving on GPGPUs via Interwarp collaborations. Paper presented at: Tenth International Symposium on Computing and Networking (CANDAR). IEEE, Himeji, Japan. 2022 158-164.
2. Campbell PJ. Gauss and the eight queens problem: a study in miniature of the propagation of historical error. *Historia Math*. 1977;4(4):397-404. doi:10.1016/0315-0860(77)90076-3
3. Yaglom AM, Yaglom IM. *Challenging Mathematical Problems with Elementary Solutions*. Dover Publications; 1987.
4. Jianli C, Zhikui C, Yuxin W, He G. Parallel genetic algorithm for N-Queens problem based on message passing interface-compute unified device architecture. *Comput Intell*. 2020;36(4):1621-1637. doi:10.1111/coin.12300
5. Bell J, Stevens B. A survey of known results and research areas for n-queens. *Discrete Math*. 2009;309(1):1-31.
6. Preußer TB, Engelhardt MR. Putting Queens in carry chains, 27. *J Signal Process Syst*. 2017;88(2):185-201. doi:10.1007/s11265-016-1176-8
7. Kise K, Katagiri T, Honda H, Yuba T. Solving the 24-queens problem using MPI on a PC cluster. *Graduate School of Information Systems*. The University of Electro-Communications, Tech. Rep; 2004.
8. Caromel D, Costanzo dA, Mathieu C. Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Comput*. 2007;33(4):275-288. doi:10.1016/j.parco.2007.02.011
9. Preußer TB, Nagel B, Spallek RG. Putting queens in carry chains. Fakultat Informatik, Technische U Niversitat Dresden, Tech. Rep. TUD-FI09–03. 2009.
10. NVIDIA. CUDA C++ Programming Guide. 2023 https://docs.nvidia.com/cuda/cuda-c-programming-guide/
11. NVIDIA. NVLink TM high speed interconnect: Application performance. Whitepaper, NVIDIA: 2788 San Tomas Expressway Santa Clara, CA. 95050. 2014.
12. Osama M, Wijs A. SIGmA: GPU accelerated simplification of SAT formulas. In: Ahrendt W, Tapia Tarifa SL, eds. *Integrated Formal Methods*. International Conference on Integrated Formal Methods. Springer International Publishing; 2019:514-522.
13. Dal Palù A, Dovier A, Formisano A, Pontelli E. CUD@SAT: SAT solving on GPUs. *J Exper Theor Artif Intell*. 2014;27(3):27.
14. NVIDIA. Tesla V100 GPU Architecture. Whitepaper, NVIDIA; 2788 San Tomas Expressway Santa Clara, CA. 95050. 2017.
15. NVIDIA. Turing GPU Architecture. Whitepaper, NVIDIA; 2788 San Tomas Expressway Santa Clara, CA. 95050. 2018.
16. NVIDIA. Ampere GA102 GPU Architecture. Whitepaper, NVIDIA; 2788 San Tomas Expressway Santa Clara, CA. 95050. 2020.
17. Feinbube F, Rabe B, Löwis M, Polze A. NQueens on CUDA: optimization issues. Paper presented at: Ninth International Symposium on Parallel and Distributed Computing. 2010 63-70.
18. Hamdy A, Yehia A, Nandakishore S, Abdel-Hameed B. Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis. 2022.
19. Zhe J, Marco M, Benjamin S, Daniele PS. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. 2018.
20. Clarke L, Glendinning I, Hempel R. The MPI message passing Interface standard. *Programming Environments for Massively Parallel Distributed Systems*. Birkhäuser Basel; 1994:213-218.
21. Heule MJH, Kullmann O, Wieringa S, Biere A. *Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. Hardware and Software: Verification and Testing*. Springer; 2012:50-65.
22. Sosic R, Gu J. A polynomial time algorithm for the N-Queens problem. *SIGART Bulletin*. 1990;1(3):7-11. doi:10.1145/101340.101343
23. Martinjak I, Golub M. Comparison of heuristic algorithms for the N-queen problem. Paper presented at: 29th International Conference on Information Technology Interfaces. 2007 759-764.
24. Cao J, Chen Z, Wang Y, Guo H. Parallel implementations of candidate solution evaluation algorithm for N-Queens problem. *Complexity*. 2021;2021:1-15. doi:10.1155/2021/6694944
25. Simkin M. The number of n-queens configurations. *Adv Math*. 2023;427:109127. doi:10.1016/j.aim.2023.109127
26. Abramson B, Yung M. Divide and conquer under global constraints: a solution to the N-queens problem. *J Parallel Distrib Comput*. 1989;6(3):649-662.
27. Zhang T, Shu W, Wu MY. Optimization of N-Queens solvers on graphics processors. *International Workshop on Advanced Parallel Processing Technologies*. Springer; 2011:142-156.
28. Tzeng S, Lloyd B, Owens JD. A GPU task-parallel model with dependency resolution. *Computer*. 2012;45(8):34-41. doi:10.1109/MC.2012.255
29. Thouti K, Sathe SR. Solving N-Queens problem on GPU architecture using OpenCL with special reference to synchronization issues. Paper presented at: 2nd IEEE International Conference on Parallel, Distributed and Grid Computing. 2012 806-810.
30. Muniyandi R, Maroosi A. Enhancing the simulation of membrane system on the GPU for the N-Queens problem. *Chin J Electron*. 2015;24:740-743. doi:10.1049/cje.2015.10.012
31. Biere A, Heule MJ, Maaren VH, Walsh T. *Handbook of Satisfiability. 185 of Frontiers in Artificial Intelligence and Applications*. IOS Press; 2009.
32. Meyer Q, Schönfeld F, Stamminger M, Wanka R. 3-SAT on CUDA: towards a massively parallel SAT solver. Paper presented at: International Conference on High Performance Computing & Simulation (HPCS). 2010 306-313.
33. Somers J. The N Queens Problem: a study in optimization. 2023 http://users.rcn.com/liusomers/nqueen.demo/nqueens.html
34. Heule MJH, Maaren VH. Look-Ahead Based SAT Solvers. In Biere et al. [31]ch. 5. 155–184.
35. Kullmann O. Fundaments of Branching Heuristics. In Biere et al. [31]ch. 7. 205–244.
36. Fisher RA, Yates F. *Statistical Tables for Biological, Agricultural and Medical Research*. sixth ed. Oliver & Boyd; 1963:37.

How to cite this article: Pantekis F, James P, Kullmann O, O'Reilly L. Optimized massively parallel solving of N-Queens on GPGPUs. *Concurrency Computat Pract Exper*. 2024;e8004. doi: 10.1002/cpe.8004

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.8004 by Swansea University, Wiley Online Library on [24/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License