Scalable N-Queens Solving on GPGPUs via Interwarp Collaborations

Filippos Pantekis
Department of Computer Science
Swansea University
Swansea, Wales, United Kingdom
Email: filippos.pantekis@swansea.ac.uk

Phillip James
Department of Computer Science
Swansea University
Swansea, Wales, United Kingdom
Email: p.d.james@swansea.ac.uk

Oliver Kullmann
Department of Computer Science
Swansea University
Swansea, Wales, United Kingdom
Email: o.kullmann@swansea.ac.uk

***Abstract*—In this paper we present how recent hardware revisions and newly introduced approaches to thread collaboration in NVIDIA GPUs and the CUDA toolkit can be used to design an extensible, scalable GPU-based solver for the N-Queens problem. We discuss various design choices ranging from memory structure, to low-level optimisations on newer GPU hardware that result in strong performance when solving the N-Queens problem using an optimised solving algorithm that can be applied to other similar in nature problems.**

I. Introduction

The N-Queens problem [1] asks how many possible placements of queens exist on an $N \times N$ chess board such that no queen can attack another. A queen can attack another if it is in the same row, column or diagonal as it. This has been a long standing open problem for which no general mathematical solution or efficient algorithm has been identified. Finding an efficient solution would bring benefits to applications in areas such as job scheduling, deadlock prevention, and traffic control [2], [3]. One of the most prevalent solving techniques for the N-Queens problem is a trial-and-error backtracking search which is based on a simple algorithm that leaves room for heuristic optimisations. GPU acceleration of this type of search however, is a challenging topic that requires a range of techniques to be employed for successful parallelisation, due to its 'irregular' nature (see Section II-A for details).

In this work, we propose a new GPU-based solver for the N-Queens problem that relies on thread collaboration within warps, and whose structure inherits from algorithms used in the domain of SAT [4]. It also utilises warp balloting and shuffling to tackle the problem and can be scaled across any number of GPUs. We begin by introducing background material before detailing both our algorithm and the technical implementation choices we have made. Finally, we present results highlighting that GPUs can improve upon current results for solving the N-Queens problem.

II. Background

Finding a single solution for instances of the N-Queens problem , i.e. finding a single placement of queens on a N x N chess board, can be achieved in a 'fast' manner using both search based algorithms [5]–[7] or directly without any search [8]. However, counting all possible solutions still requires a brute-force search, with approaches often applying search-space limiting heuristics. Perhaps the most prominent solving technique [9], treats the board as a 'ladder' upon which a depth-first search is performed, recursively attempting to place a queen in a valid position on each rung, and backtracking when no more moves can be made. The complexity of this algorithm remains exponential making it computationally intensive for larger orders of N. As of yet, the solution counts are known up to $N = 27$, the enormous computation effort for which was undertaken by purpose-built FPGA hardware [10] over the course of a year. This result is pending verification.

With the growing commercial availability and cost effectiveness of GPUs, a number of promising attempts have been made to harness their computational power for solving the N-Queens problem and other problems of a similar nature [11]–[14]. Such attempts require a large degree of optimisation and algorithmic re-structuring to account for the specialties of GPUs, and achieve good performance. In [11], Zhang et al. present a range of steps that result in speedups when solving on a GPU, and highlight the fact this type of computation is highly 'irregular' considering the usual requirements for parallelisation on such devices (discussed further in Section II-A). This is further echoed in [14] by Feinbube et al. who also highlight that programming languages and compilers are not sufficient to achieve optimal performance on these devices.

A. NVIDIA's Compute Unified Device Architecture (CUDA)

NVIDIA's CUDA is a programming platform for NVIDIA's compute capable GPUs, often referred to as "General Purpose Graphics Processing Units" (GPGPUs). CUDA provides a framework for massively parallel computations on these devices, where the "host" computer launches kernels of work on the "devices" (GPUs) it controls. A kernel houses an (often large) number of threads that will execute on the same GPU to tackle a problem. Threads are logically divided across a number of blocks which can be of up to three dimensions. The blocks are then grouped into a grid arrangement which can also have up to 3 dimensions. This grouping is of use to tasks exhibiting spatial locality and impacts access to different memory types. Generally, memory types can be divided between two categories: On chip and off chip memory.

Supported by EPSRC grant EP/S015523/1

The former is size-restricted but can be accessed in few clock cycles, yet the latter is larger in size but requires significantly more clock cycles to be accessed. In summary, some of the available memory types are shown in Table I.

On the hardware level, a number of Streaming Multiprocessors (SMs) are available. A number of blocks are scheduled per SM to run and use its resources. For execution, each block is partitioned into sub-groups of (currently) 32 threads, called 'warps'. When processing time is given to a warp, threads within it should ideally be ready to execute the same instruction. However, branching instructions leading to different execution paths can introduce the potential for 'warp divergence' whereby threads in the same warp are waiting to execute different instructions. This results in a degeneration in performance since only one instruction can be executed at once per warp, leaving the rest of the threads waiting.

Communication between threads in a grid is possible through Global Memory, however it should be avoided where possible or alternatively hidden behind computation loads to reduce memory latency costs. Similarly, synchronisation of threads across the grid is not natively supported, but synchronisation of threads within a block, as well as within warps, is. A range of synchronising constructs that perform data exchange or accumulation is also available on the warp level, enabling 'cheap' communication between threads.

TABLE I
SUMMARY OF MEMORY TYPE CHARACTERISTICS IN CUDA

Type	Scope	Access Cost	Capacity	Writable
Global Memory	Global	High	Large	✓
Constants Memory	Global	Low	Small	✗
Shared Memory	Block	Low	Small	✓
Local Memory	Thread	High	Large	✓
Registers	Thread	Very low	Very small	✓

B. The `DoubleSweep` Algorithm

At the basis of this paper there is a novel algorithm, called `DoubleSweep-light`, for counting all N-Queens solutions, whose main motivation and features we discuss next. This algorithm is based upon the `DoubleSweep` created and implemented in the open-source platform `OKlibrary` [4].[1]

Backtracking algorithms for counting (or enumerating) all N-Queens solutions are a popular programming exercise. The most efficient incarnation of such basic algorithms, known as "Somer's algorithm" is well know.[2]

The most basic feature of Somer's algorithm is that it represents the current board by three computer-(bit-)words: one word for columns blocked by queens, viewing them only as rooks, and two words for the blocked diagonals resp. antidiagonals (for the two forms of semi-bishops, moving diagonally resp. antidiagonally). Splitting of the problem starts with the bottom row, placing a queen at each of the N available positions. This is recursively repeated, always choosing the next row (from the bottom) for splitting – it is crucial to note, that the cells blocked by the current queens are obtained by copying the column-word, and shifting resp. anti-shifting the diagonal-words, and then performing the logical-or of these three words as further discussed in Section IV,

The `DoubleSweep` algorithm combines this word-level parallelism with basic ideas from SAT-solving (see [15] for an overview on this class of SAT-algorithm). The two main features are:

- Instead of just propagating the rook and bishop moves to the next row, propagation is (nearly) done for the *whole board*. That is, whether for any row or column there is no open cell left, or whether for any row there is only one open cell left, in which case a queen must be placed there (and this again is propagated until fixed-point).[3] This corresponds to "unit-clause propagation" for SAT-solvers, and helps cutting off unsatisfiable branches.
- Instead of splitting on the bottom-most row, splitting is done on the central row. This simple branching heuristic helps to make propagation more efficient – as in chess playing, figures placed more centrally have a greater influence.

Now besides the three words as above, N words are used to represent the full board with current propagations. And the two (anti-)diagonal-words now use 64 bits, 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.

We introduce the `DoubleSweep-light` algorithm, which is the lightest version of `DoubleSweep`: unlike Somer's algorithm, splitting 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 as shown in Figure 1. This light version of `DoubleSweep` is the natural starting point, given the high complexity of implementing such dynamic algorithms on GPUs.

III. SOLVING N-QUEENS IN PARALLEL

Solving the N-Queens problem in parallel is often done via search-space splitting by partly exploring the tree of solutions to a certain level, and using the partial solutions as starting points for a number of parallel workers to tackle without the risk of converging paths [11], [12], [14].

```
1 fn solve(s, locked_idx):
2   let sols ← 0;
3   while advance_state(s, locked_idx) do:
4     double_sweep_light(s)
5     if state_solved(s, locked_idx) then:
6       sols ← sols + 1
```

Listing 1. Overview of each worker's operation.

[1] The C++ code is available from https://github.com/OKullmann/oklibrary/blob/master/Satisfiability/Transformers/Generators/Queens/SimpleBacktracking/Queens_RUCP_ct.cpp.

[2] A copy of the code is available in the `OKlibrary` at https://github.com/OKullmann/oklibrary/tree/master/Satisfiability/Transformers/Generators/Queens/SomersCounting.

[3] A forced placement of a queen is detected only for rows, not for columns, due to the row-centric storage of the board.

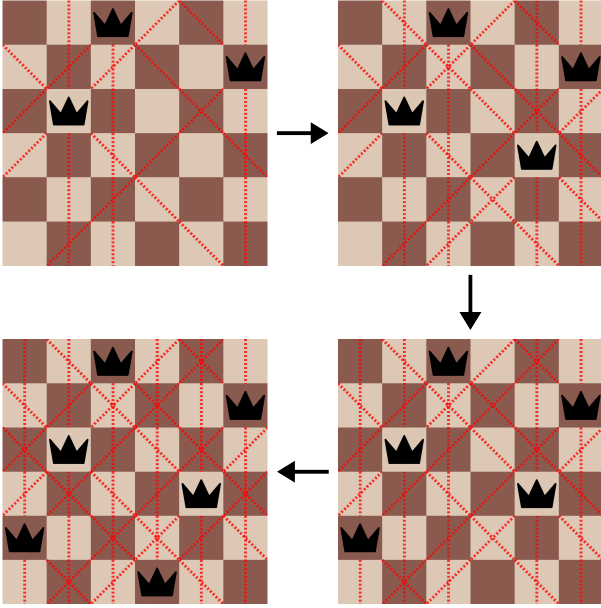

Fig. 1. Visualisation of DoubleSweep-light.

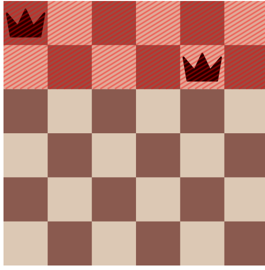

Fig. 2. A partial state of $N = 6$, locked at row 2.

For our approach, each worker receives a partly-explored state and operates on it, keeping track of its own solution count. A state is in essence a 'snapshot' of the recursive exploration which holds information such as the placement of queens and current row.

A worker's operation can be broken down to the algorithm shown in Listing 1. The worker receives a (partly explored) state with the aforementioned information, as well as the index of the lowest 'locked' row. This index marks a region of the board which must not be touched by the worker. An example of a partial state with the first two rows 'locked' can be seen in Figure 2. The worker then attempts to complete the board by making successive moves on the non-locked region, keeping track of solutions and stopping once all possible arrangements of queens in the non-locked region are explored. The function advance_state (detailed in Listing 2), takes the current state of the board as well as the index of the last locked row, and attempts to make a single move. A move in this context may be placing a queen in the next available row, or if that is not possible, moving the last placed queen to the next non-conflicting, unexplored position in the same row.

A non-conflicting position is one where no currently placed queen can attack. If the state is such that no move can be performed, then the queen in the last row is removed, and the process repeats recursively as shown in Figure 3. A state is considered completed when this function backtracks into the locked region of the board.

```
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) ∧
7            ¬blocked_col(s, i) then:
8          idx ← i
9          break
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 2. State advancement algorithm.

Following each advancement of a state, the function double_sweep_light (detailed in Listing 3) takes the current state of the board and performs the DoubleSweep-light procedure described in Section II-B. Starting from the last populated row, each row that has exactly one non-conflicting position is iteratively discovered, and a queen is placed in it.

```
1  fn double_sweep_light(s):
2    let free ← nil
3    foreach i ∈ [0, N[ do:
4      if ¬has_diagonal(s, i) ∧
5          ¬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.current_row + 1
```

Listing 3. The DoubleSweep-light algorithm.

Lastly, once the DoubleSweep-light procedure is performed, a check is made to establish if the solution count should be incremented. This is the case when the last row of the state has been populated i.e. N queens have been placed.

A. Generation of Initial States

To generate an initial pool of states for parallel exploration a recursive search has to be performed up to a certain cut-off depth. In the case of an N-Queens board, this would be a

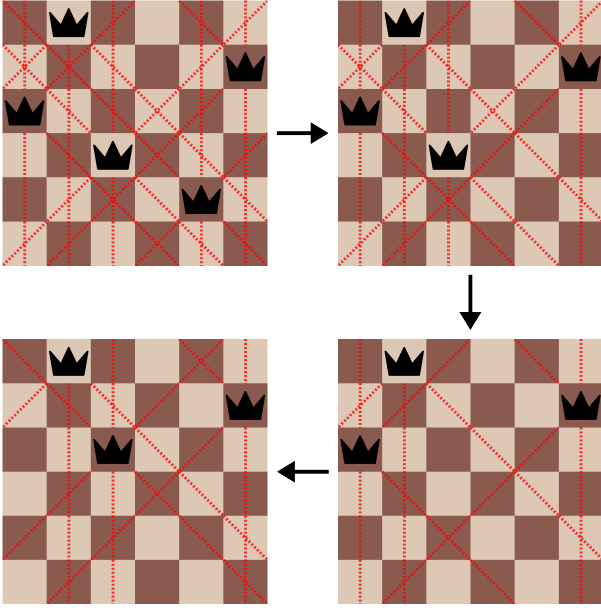

Fig. 3. Steps involved in advancing a state.

cut-off row (or column depending on the direction of search). The maximum possible number of states generated can be pre-calculated, however, often a large subset of the naïvely generated states are invalid or can be advanced no further. To generate a pool of approximately w many valid states, we employ a ladder-climbing approach presented in Listing 4. Given an $N \times N$ board, a row $r = log_N(w)$ is chosen such that the maximum number of naïve states generated, is at least w. The recursive search is performed up to r, keeping only the set S of valid states generated. If $|S| < w$, then r is incremented and state generation is repeated. Since the number of states generated between two successive rows can vary wildly, we also employ an upper limit determined as a constant factor f over w. The resulting pool comprises of different states which are all 'locked' at a row r′ and can be advanced at least once. However, it is not possible to guarantee that states in the pool will result in uniform search paths.

```
1 fn gen_state_pool(N, w):
2   let r ← log_N(w)
3   let S ← ∅, S′ ← ∅
4   do:
5     S′ = S
6     S ← gen_states(N, r)
7     r ← r + 1
8   while |S| ≤ (w ∗ f) ∧ r < N − 1
9   return (S′, r)
```

Listing 4. Ladder-climbing for initial state generation

This algorithm is implemented in parallel on the host-side, where each thread places a queen in a different position w.r.t the rest on the first row and then continues state generation. Additionally, it may be the case that by incrementing r, state generation is on a downward slope. This results in $|S_2| \leq |S_1|$ for two sets of states S_1, S_2 generated by locking at rows r and $r+1$ respectively. To remedy this, a flag is introduced in the implementation to decide if state generation must terminate in this case or continue.

IV. Implementation on the GPU

We chose to let each thread on the GPU act as a solver which works on its own state as described in Section III. Each state is a struct, encoding the following information:

1) The indexes of placed queens on the current board (array of N many unsigned 8-bit integers)
2) The index of the first row without a queen (unsigned 8-bit integer)
3) Per-row projections of conflicting diagonals caused by placed queens (two 64-bit bit vectors)
4) The occupied columns (32-bit bit vector)

Items 3 and 4 can be derived from the rest, and are thus are unessential to the state, however we believe their repeated re-computation effort outweighs their memory impact.

The search begins with a pool of states being generated (as described in Section III-A). Then, the solver kernel is launched which starts the search. Each thread then copies one state from global to shared memory where it stays for the rest of the computation. We chose one dimensional grids and blocks since we are not making use of spatial information. The number of threads per block is dependent on the size of each (8-byte aligned) state in conjunction with the size of the shared memory partition on the target device(s). Whilst transferring of complex structs results in uncoalesced memory access by the warps, we note this transfer is only performed once at the beginning of the computation which results in minimal impact.

Following the transfer of data to shared memory, each thread, in its respective warp, starts to successively advance, and performs `DoubleSweep-light` on its own state as explained in Sections IV-C and Section IV-B. Due to the branchfull nature of the operations, this typically results in a high degree of warp divergence. To mitigate this, warp synchronization barriers are interleaved between the operations. Such barriers force threads in a warp to wait until the rest reach the same barrier. During each step of the combined search effort of each warp, the threads in the warp are balloted on whether or not the warp should exit. A thread votes 'yes' if it has reached the end of its search path. The warp will continue this process until all threads vote 'yes'. Balloting is implemented efficiently using warp-level primitives.

A. Tracking Diagonals

As described in Section II-B, two bit vectors are used to form the projection of diagonal attacks on each row. More specifically, the 64-bit bit vector v is updated upon placement of a queen at some row-column pair (r, c) by first creating a bit mask $m = 1 << ((N-1-c)+r)$ and subsequently computing the new value of the vector $v' = v|m$. This operation is

reversed when a queen is removed from the board (i.e. during backtracking) in a similar fashion, where the new vector value is computed as $v' = v \& \neg m$.

Given the vector v, the projection of diagonals affecting row r' is derived as: $d_{r'} = (v >> r') \ \& \ ((1 << N) - 1)$, which results in an N-bit vector where 1's represent a 'blocked' column. In most cases, the expression can be simplified to $d_{r'} = v >> r'$ provided that only the first N bits are considered in the computation.

The above process is sufficient to track diagonals in one direction. Anti-diagonals are tracked in the same way, except the vector is treated from higher order bits to lower.

B. Performing `DoubleSweep-light`

The `DoubleSweep-light` procedure described in Section II-B is triggered after each advancement of the state. To derive queen placements, it must be the case that on some row r, there is exactly one non-conflicting position at column c. To find c, a bit mask m is created by combining the diagonal projections d_r for this row, with the occupied columns mask o as: $m = \neg(d_r | o)$. In the resulting mask, set bits mark the free, non-conflicting positions in row r. From this mask, the set bit population count is taken. If the count is exactly one, the column c is identified by finding the position of the only set bit, and a queen is placed. This operation results in a degree of thread divergence since `DoubleSweep-light` may perform a varying number of steps depending on the state of each thread in a warp. This is unavoidable, however a synchronisation barrier following the `DoubleSweep-light` operation is inserted re-converging threads ready for the next operation.

C. Advancing States

Advancing a state, as described in Section III, aims to make a valid change to the state without overlapping with previously explored states. The process of identifying the next valid placement on a given row r is performed by first computing the mask m as described in Section IV-B. If no queen has been placed in r, then the position of the first set bit of m is used. If a queen is placed in column q of r a new mask $m' = m \& (P << (q + 1))$ is computed where P is a mask with only the lower N bits set. This effectively isolates all positions up to and including the current queen's position from being chosen again. Like before, the position of the first set bit is where the new queen is placed. If no bit is set in m', backtracking is performed by choosing the previous row and repeating the above process. This is done using iteration as opposed to recursion for which support is limited on the device. Again we use synchronisation barriers as the number of steps this takes varies depending on the current state.

D. Summarising Results

Once each warp reaches the end of solution counting, each of its threads has the number of solutions it found stored in a register. A parallel summation operation is then performed within the warp via register 'shuffling'. During this operation,

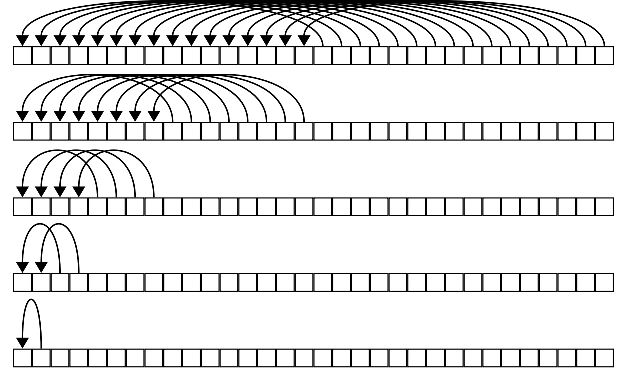

Fig. 4. Visualisation of steps involved in warp down-shuffling.

the warp is partitioned into two halves H_l and H_r, and each thread in H_l adds to its solution count to that of the corresponding thread in H_r. Then, H_l is once again partitioned into two halves, and the process repeats until the first thread of the warp holds the collective sum of solutions as illustrated in Figure 4. The exchange of information is performed between the registers of the threads without the need of repeated loading and storing to shared memory.

The first thread writes the sum to a distinct location in shared memory. Once all warps complete this operation, one of the warps is chosen to perform the final summation of the shared memory elements as described above, resulting in a single value per block. A thread from each block writes this sum to a distinct location in shared memory, and the final summation of results is performed by the host.

The choice to let the host perform the final summation of block results was made on the basis that results for larger N are likely not representable by a 64 bit integer (currently the maximum natively supported integer type by NVIDIA GPUs).

E. Performance Optimizations and Observations

In Section IV-D the summation of results through a 'warp shuffling' operation is presented. Warp shuffling instructions have been supported by NVIDIA's Parallel Thread Execution (PTX) virtual machine, as well as the low level GPU assembly language used by NVIDIA known as 'SASS', since the Kepler micro-architecture [16].

In Section IV-D the process described is achieved through five `shfl.sync.down.b32` PTX instructions, each paired with addition. In the recent Ampere micro-architecture [17] support was introduced for a warp-wide addition reduction as a single instruction, namely `redux.sync.add.u32`. We found the latter to perform much better in our benchmarks compared to the former, on a system with an Ampere device as described in Section V-A.

In Section IV-B, the intrinsics `__popc` and `__ffs` are used which are defined in CUDA to find the set bit count and the first set bit of a mask respectively. Currently, the former is backed by the instruction `popc` in PTX (Since the Fermi architecture), however the latter results in two separate operations being performed: the reversal of the bits using

TABLE II
SOLVING TIMES IN MILLISECONDS FOR N ACROSS 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

`brev.b32` followed by the identification of the first higher order set bit using `bfind.shiftamt.u32`. Bit reversal is unnecessary during the `DoubleSweep-light` procedure, as it is known that exactly one bit is set. This is not an assumption the compiler can safely make, thus it is manually implemented.

Further to the above, we chose to make use of the PTX instructions `bfi` (Bit Field Insert), `bfe` (Bit Field Extract) and `bmsk` (Bit field Mask) on code targeted to devices that support them, following the observation that some opportunity for their use was not taken by the compiler. Since interfering with the compiler via inline assembly may hinder optimisations, we regard those as "experimental" optimizations that may result in lower performance in future generations of hardware.

Lastly, significant performance gains were observed when in code, non-aliasing pointers were annotated with `__restrict__`, a CUDA keyword akin to the type qualifier `restrict` in the C language. The extra information offered on pointers, enables a set of optimisations that would otherwise be deemed "unsafe" by the compiler.

F. Using Multiple GPUs

Each warp in our solver works in isolation, and has no dependency or communication requirements with other warps. As such, our approach lends itself to parallelisation across multiple GPUs in the same machine or distributed (e.g. Beowulf clusters [18]).

When preparing the solver for such systems, care must be taken in the compilation of the solver. When multiple GPUs of different compute capabilities are used for a computation, code must be compiled targeting the device with the lowest compute capability. This may result in fewer optimizations for the newer devices. Alternatively separate instances of the solver may be compiled specifically for the targeted compute capability. However, this will require additional manual work for scheduling.

V. PERFORMANCE EVALUATION

To test the performance of our solver, we took 10 kernel execution time measurements for each $N \in \{14, \ldots, 20\}$. Three sets of benchmarks were taken on devices across two systems with the following main characteristics:

System 1: CPU - AMD Ryzen 9 3950X, GPU - NVIDIA RTX 3090 @ 1745MHz.

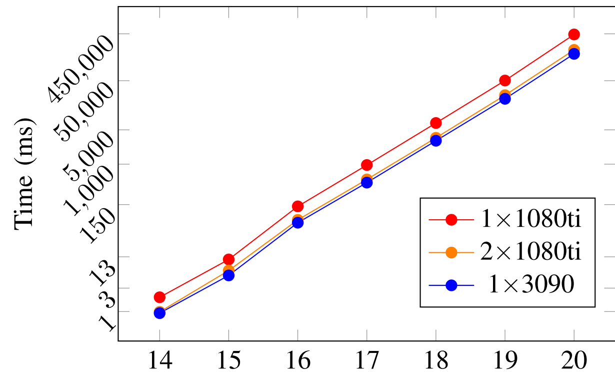

Fig. 5. Visualisation of time trend.

System 2: CPU - AMD Ryzen Threadripper 2950X, GPU - 2× NVIDIA GTX 1080ti @ 1683MHz (SLI).

On both systems, a cool-down period was enforced between benchmarks to prevent impacts from thermal throttling. On System 2, two sets of benchmarks were taken, one on a single GPU and one using both GPUs (see Section IV-F). The average solving time for each N tested, along with the standard deviation (error) are presented in Table II and Figure 5.

Solving on two GTX 1080ti GPUs yields a 2x speedup compared to solving on a single GTX 1080ti, affirming our aim to design a scalable solver with loose coupling between devices involved. Interestingly, the results indicate that two 1080ti GPUs perform almost as well in this task as the RTX 3090. The RTX 3090 features the Ampere architecture and is significantly more capable than the 1080ti which features the Pascal architecture, thus it may be expected that the 3090 would outperform the 1080ti by a larger margin, however we believe that such comparison of devices may only be done in the context of the application at hand.

Overall the performance of our solver matches or surpasses that of previous work in the field [11], [13], [14], whilst being openly scalable. Its extensible design allows for problems of size N up to 64 to be explored with minor changes. Further heuristics can also be easily introduced. We note that currently our solver does not include heuristics to prevent transformations of already found solutions from being discovered which remains future work. As experienced by others [11], [14] we envisage such techniques may improve performance further.

We also note that comparisons with other work would ideally be performed on comparable hardware and must consider several factors including aspects such as GPU driver versions, library and compilers versions, which is generally unavailable.

A. Benchmark Remarks

For each value of N, a number of states were generated following the process described in Section III-A. As the number generated cannot be tightly controlled, the number of blocks in each kernel profiled (shown in Table III) varies. Larger kernels were preferred to compensate for the fact some of the blocks have longer run times than others, and to provide more stable results. There is some cost associated with blocks retiring or becoming resident on an SM during computation, however details on such costs are unpublished, and is likely compensated for by the weight of the computation performed.

TABLE III
NUMBER OF BLOCKS AND THEIR RESPECTIVE SIZE IN KERNELS, FOR EACH VALUE OF N BENCHMARKED.

N	Block Count	Block Size
14	4462	1024
15	27199	1024
16	45266	992
17	35540	992
18	76526	992
19	26727	992
20	45931	992

The size of blocks in our benchmarks is different for some N. This is a limit imposed by the shared memory per block requirements of our solver which varies for different values of N as described in Section IV. The size of a block can be of significance to performance, however a range of other factors must be considered such as register usage, shared memory requirements and L2 cache split, etc. For benchmarking, we fixed the shared memory size to the largest supported across all devices, irrespective of shared memory partitioning capabilities. We note however that deciding block size based on a device's capabilities may result in better performance.

VI. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we have presented both a new algorithm – `DoubleSweep-light` – for solving the N-Queens problem and given a detailed discussion of how such an algorithm can be implemented for high performance on GPUs. Our approach relies on considerable thread collaboration within warps and utilises warp balloting. Our results demonstrate that this approach outperforms existing approaches and is also openly scalable in terms of parallelisation across devices.

Looking to the future, we plan to extend the solver by introducing more search space reducing heuristics such as the elimination of symmetries. With the current algorithm, different workers will, during their exploration, arrive at 90°, 180° and 270° rotations as well as the horizontal and vertical mirroring of a "fundamental" solution. This will be part of our implementation of the full `DoubleSweep` algorithm.

Additionally, we observed that during search, some threads have shorter search paths and hence come to an end sooner than others. As a result warps have fewer active threads, and do not perform at full potential. For longer running computations, we would like to devise a restart heuristic for the search so that active search paths can be re-distributed periodically to fewer, full warps. Our early explorations indicate that 'smart' warp-activity based heuristics involve a degree of communication between each device and the host that coordinates it. This incurs a large cost, however we believe this can be remedied with reduced host polling or a different restart strategy.

Finally, we intend to evaluate the performance of our solver for larger N against a cluster of 48 NVIDIA A100 GPUs on a newly acquired ATOS BullSequana X410 supercomputer deployment. With the rise in GPU availability in large scale computers, we hope that the solution count to $N = 27$ can be verified and that of $N = 28$ can be found using these devices.

REFERENCES

[1] P. J. Campbell, "Gauss and the eight queens problem: A study in miniature of the propagation of historical error," *Historia Mathematica*, vol. 4, no. 4, pp. 397–404, 1977.

[2] M. M. Waqas and A. A. Bhatti, "Optimization of N+1 Queens problem using discrete neural network," *Neural Network World*, vol. 27, 2017.

[3] C. Erbas, S. Sarkeshik, and M. M. Tanik, "Different perspectives of the N-Queens problem," in *Proceedings of the ACM Annual Conference on Communications*. ACM, 1992, p. 99–108.

[4] O. Kullmann, "The OKlibrary: Introducing a "holistic" research platform for (generalised) SAT solving," *Studies in Logic*, vol. 2, no. 1, pp. 20–53, 2009.

[5] R. Sosic and J. Gu, "A polynomial time algorithm for the N-Queens problem," *SIGART Bulletin*, vol. 1, no. 3, pp. 7–11, 1990.

[6] I. Martinjak and M. Golub, "Comparison of heuristic algorithms for the N-Queen problem," in *29th International Conference on Information Technology Interfaces*, 2007, pp. 759–764.

[7] J. Cao, Z. Chen, Y. Wang, and H. Guo, "Parallel implementations of candidate solution evaluation algorithm for N-Queens problem," *Complexity*, pp. 1–15, Feb. 2021.

[8] A. M. Yaglom and I. M. Yaglom, *Challenging Mathematical Problems with Elementary Solutions*. Dover Publications, 1987.

[9] B. Abramson and M. Yung, "Divide and conquer under global constraints: A solution to the N-queens problem," *Journal of parallel and distributed computing.*, vol. 6, no. 3, 1989.

[10] T. B. Preußer and M. R. Engelhardt, "Putting Queens in Carry Chains, №27," *J. Signal Process. Syst.*, vol. 88, no. 2, pp. 185–201, 2017.

[11] T. Zhang, W. Shu, and M.-Y. Wu, "Optimization of N-Queens solvers on graphics processors," in *Advanced Parallel Processing Technologies*. Springer, 2011, pp. 142–156.

[12] S. Tzeng, B. Lloyd, and J. D. Owens, "A GPU task-parallel model with dependency resolution," *Computer*, vol. 45, no. 8, pp. 34–41, 2012.

[13] R. Muniyandi and A. Maroosi, "Enhancing the simulation of membrane system on the GPU for the N-Queens problem," *Chinese Journal of Electronics*, vol. 24, pp. 740–743, 2015.

[14] F. Feinbube, B. Rabe, M. Löwis, and A. Polze, "NQueens on CUDA: Optimization issues," 2010, pp. 63–70.

[15] M. J. H. Heule and H. van Maaren, "Look-ahead based SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. J. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, February 2009, vol. 185, ch. 5, pp. 155–184.

[16] NVIDIA, "NVIDIA GeForce GTX 680," https://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf, White Paper, 2012.

[17] ——, "NVIDIA Ampere GA102 GPU Architecture," https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, White Paper, 2020.

[18] D. Ridge, D. Becker, P. Merkey, and T. Sterling, "Beowulf: harnessing the power of parallelism in a pile-of-PCs," in *IEEE Aerospace Conference*, vol. 2, 1997, pp. 79–91.