|  |
| --- |
| Edinburgh Napier University |
| SET10108– Concurrent and Parallel Systems (Coursework 2) |
| Michael Suttie - 40541559 |

|  |
| --- |
| Suttie, Michael  12-6-2022 |

WORD COUNT EXCLUDING CONTENTS AND REFERENCES: 3095

Table of Contents

[1. Introduction 3](#_Toc121612410)

[1.1. Hardware Specs 3](#_Toc121612411)

[2. Serial 4](#_Toc121612412)

[2.1. Implementation 4](#_Toc121612413)

[2.2. Serial Performance Results 5](#_Toc121612414)

[3. Parallel – OpenMP 6](#_Toc121612415)

[3.1. Removing Recursion 6](#_Toc121612416)

[3.2. Method of Parallelisation 8](#_Toc121612417)

[3.2.2. Parallel For (CalculateSolutions) 8](#_Toc121612418)

[3.2.2. Scheduling Analysis 8](#_Toc121612419)

[3.2.2. Parallel For (boardIsValid) 11](#_Toc121612420)

[3.3. OpenMP Performance Results 12](#_Toc121612421)

[3.4. OpenMP vs Serial 14](#_Toc121612422)

[4. Parallel – GPU (CUDA) 15](#_Toc121612423)

[4.1. Method of Parallelisation 15](#_Toc121612424)

[4.2. CUDA Performance Results 17](#_Toc121612425)

[4.3. CUDA vs Serial and OpenMP 18](#_Toc121612426)

[5. Conclusions 20](#_Toc121612427)

[References 21](#_Toc121612428)

Table of Figures

[Figure 2‑1: Serial Implementation 4](#_Toc121612388)

[Figure 2‑2: Serial N-Queens Performance Table 5](#_Toc121612389)

[Figure 2‑3: Serial N-Queens Average Times 5](#_Toc121612390)

[Figure 3‑1: Initial Non-Recursive Attempt 6](#_Toc121612391)

[Figure 3‑2: Final Brute Force Implementation 7](#_Toc121612392)

[Figure 3‑3: Parallel For (calculateSolution) 8](#_Toc121612393)

[Figure 3‑4: Scheduling Performance (N=9) 9](#_Toc121612394)

[Figure 3‑5: Scheduling Performance (N=10) 10](#_Toc121612395)

[Figure 3‑6: Parallel For (boardIsValid) 11](#_Toc121612396)

[Figure 3‑7: Parallel For (boardIsValid) Enabled vs Disabled 11](#_Toc121612397)

[Figure 3‑8: Parallel For (boardIsValid) Comparison Graph 12](#_Toc121612398)

[Figure 3‑9: OpenMP performance Table 12](file:///C:\Users\Suttie\Desktop\Projects\C++%20Projects\N-Queens-CW2\Report\CW2%20Report%20-%20Michael%20Suttie.docx#_Toc121612399)

[Figure 3‑10: OpenMP Performance 13](#_Toc121612400)

[Figure 3‑11: Performance table for Serial and OpenMP 14](#_Toc121612401)

[Figure 4‑1: CUDA checkQueenPos() Kernel 15](#_Toc121612402)

[Figure 4‑2: CUDA Memory Copy 16](#_Toc121612403)

[Figure 4‑3: Kernel call from host 16](#_Toc121612404)

[Figure 4‑4: Further CUDA Memory Management 17](#_Toc121612405)

[Figure 4‑5: CUDA Performance Results Table 17](#_Toc121612406)

[Figure 4‑6: CUDA Performance 18](#_Toc121612407)

[Figure 4‑7: Three Method Comparison Table 18](#_Toc121612408)

[Figure 4‑8: Performance Graph Comparison 19](#_Toc121612409)

# Introduction

The purpose of this coursework is to solve the N-Queens problem – the problem of placing chess queens on an N x N chessboard in such a way that no two queens are a threat to another. This means that no queen can share a row, column or a diagonal space. Given the implementation provided, the task is to modify the existing serial program so that it is non-recursive and then to implement the non-recursive solution in parallel. It is important to ensure that the recursive elements are removed, as parallelisation through OpenMP and using the GPU does not play will with recursive code.

The parallelisation should be done twice; once in OpenMP and then again using the GPU using CUDA or OpenCL, and should run for N values between 4 and 10 – with the option to increase the threshold should the Laptop have the ability to handle it.

## Hardware Specs

The machine used for the creation of this program as well as running the experiments are an ASUS TUF F15 with the following specs:

* CPU – 11th Gen Intel(R) Core(TM) I5 11400H @ 2.70GHz (6 cores, 12 logical processors)
* RAM – 16GB DDR4 3200mhz
* GPU – NVIDIA GeForce RTX 3050 Ti Laptop GPU (4GB)
* OS – Windows 11 Home (Build 22000.1098)

# Serial

## Implementation

The initial serial program provided for the coursework completes the task using recursion. The code for this can be seen below:

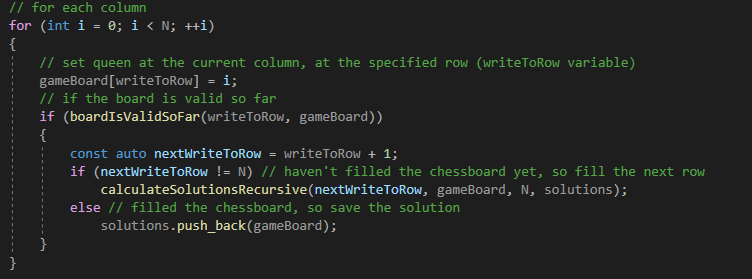


Figure ‑: Serial Implementation

Essentially, if the chess board is deemed valid, the function calculateSolutionsRecursive() will call upon itself continually in an attempt so solve the rest of the problem and send the solutions back to the solutions vector.

This solution is fine when done serially, however, when implemented in parallel, this won’t be the case. The reason for this is basically in the name, as recursion often relies on the results of previous recursive calls and typically means that the order of the recursive calls are important to the operation of the algorithm. For this reason, the future two sections on OpenMP and CUDA will run a different algorithm that aims to reduce or remove recursion completely.

## Serial Performance Results

This section will showcase and discuss the results specific to the running of the original recursive code provided. These results will be used as a comparison in the future sections when discussing the parallelisation of the N-Queens problem.

The performance of the serial program was measured by running it 10 times and recording how long each N-Queens problem took to solve. The table below shows these recorded times and their averages. The execution was so quick that the std::chrono couldn’t register a time at all for N = 4 through to N = 8, with a slight outlier in run 3 for N = 8. The only time properly registered was when calculating the solutions for N = 9 and N = 10. Even at that, the times were miniscule.

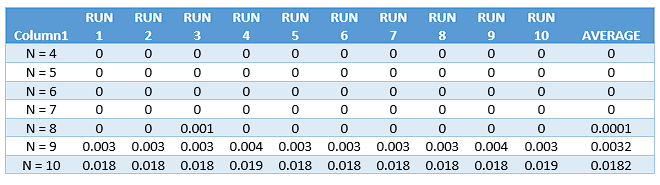


Figure ‑: Serial N-Queens Performance Table

The averages of these runs were then taken and the trend seems to show that the times get worse as the complexity of the problem increases, which is to be expected given the calculations that are occurring – a trend that will likely be replicated in the analysis of both parallel implementations too.

Figure ‑: Serial N-Queens Average Times

# Parallel – OpenMP

## Removing Recursion

As was touched upon in the Serial section – the first hurdle to overcome was changing the N-Queens solution from a recursive solution to a non-recursive solution. This was initially attempted using a backtracking algorithm. Essentially, backtracking works by trying to incrementally build the solution piece-by-piece and then going back on itself and undoing previous steps if the algorithm decides that it cannot go any further (Datta, 2022). The implementation of this can be seen in the figure below, inspiration of how to implement this was taken from Oxford College of Emory (Oxford College, n.d.)

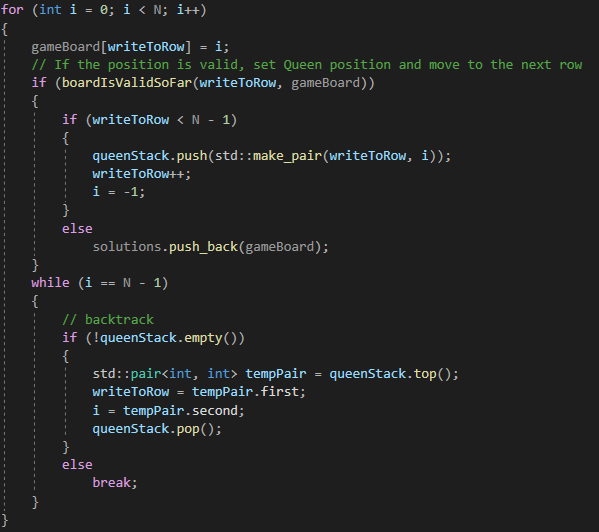
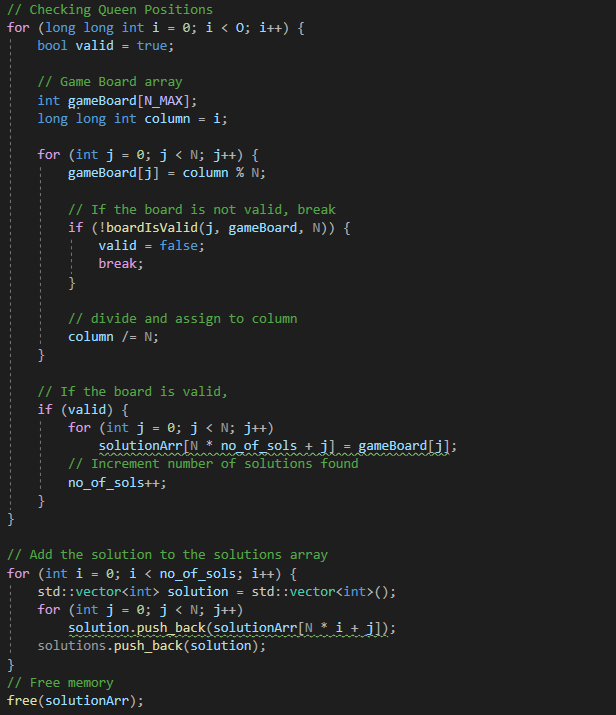


Figure ‑: Initial Non-Recursive Attempt

This code, however, proved to be difficult to implement in a way that would be parallelisable as it typically relies on recursion to be effective. Therefore, the decision was made to look into another iterative approach; the brute-force approach. It would be simple, and therefore likely to be rather slow, however it would be easier to parallelise using OpenMP and CUDA.

An acceptable approach to N-Queens with brute force was pieced together using techniques and examples found across the internet – these sources can be found in the references section of this paper and will be referenced in the following figure.

This brute force implementation will check the possible positioning of the all queens on the board and determine whether or not the positioning is valid. If the board is invalid, then the valid Boolean will be set to false. If the board remains valid, then the solution can be added to the solution array and the number of solutions can be incremented. A new solution vector is then created and populated with the solutions in the solution array and pushed back into the solution vector before the memory is freed up.



Initially, the game board was a vector – this came with some slowdown and as such was changed to an array. From working with OpenMP in the past, it was known that the array would likely work better across multiple threads than the vector would have anyway, so this change was made in preparation for the parallelisation.

The resources used to arrive at this solution are listed below:

* Simran (Simran, 2021)
* Stack Exchange (Stack Exchange, 2019)
* Geeks For Geeks (Geeks for Geeks, 2022)

Figure ‑: Final Brute Force Implementation

To accommodate the changes in the solution calculations, some changes had to be made in the boardIsValid method, however, these changes were easy to implement in comparison to researching and implementing the brute force method, especially as the brute force method was designed with the boardIsValids already functioning method in mind.

With this now in place, it came time to parallelise the new brute force solution using OpenMP.

## Method of Parallelisation

### 3.2.2. Parallel For (CalculateSolutions)

Parallisation in OpenMP was fairly straightforward given the way the brute force solution was implemented. Initially, the CalculateSolutions() method was parallelised. This was done by adding the OMP parallel for call before the main for loop, this is shown in the figure below:

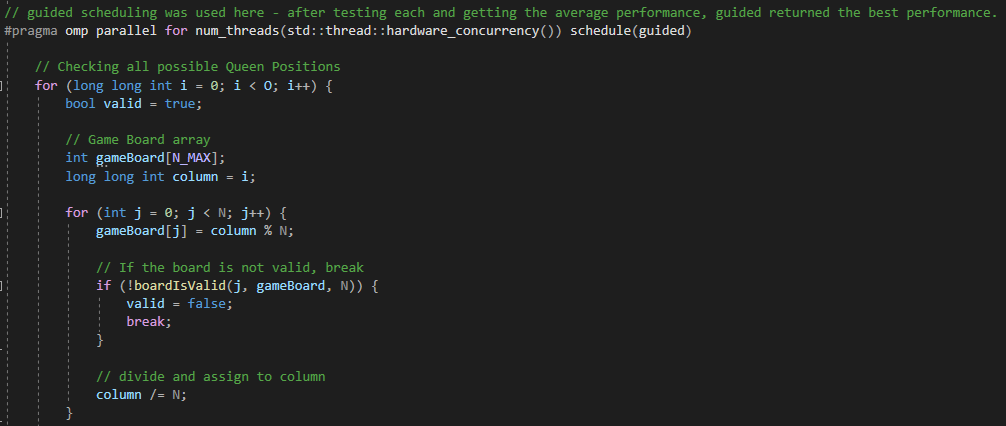


Figure ‑: Parallel For (calculateSolution)

The key thing to notice here is the use of the guided scheduling – after looking through recommendations of which scheduling types were best for certain scenarios (Jaka's Corner, 2016). It was determined that the OMPs Default, Guided and Static scheduling could work but it was unsure which would perform the best due to lack of experience. Therefore, the decision was made to test each of the three scheduling types to determine which once would perform the best. The findings of this will be discussed in the following section.

### 3.2.2. Scheduling Analysis

As stated before, were three scheduling types chosen to take forward for testing. There was initially a fourth which was Dynamic Scheduling. This was implemented but resulted in a slowdown rather than a speedup and was therefore axed because of this. The other three, Default, Guided and Static all performed similarly to expectations, with some providing speedup.

It is important to note that the results presented here are more limited in scope than the results presented in previous analysis. For the purpose of this test, the only numbers of interest were the performance times for the calculation of N = 9 and N = 10. This is because much like was shown in the serial program analysis, these were the only two that started to put serious load on the program whereas N = 4 through to 8 typically perform almost identically with a slight execution time increase expected each time. From N = 9 onwards, it becomes clear that the execution time is going to increase exponentially. This is shown in the figure below:

Figure ‑: Scheduling Performance (N=9)

Figure 3-4 shows the difference in performance between the three scheduling types when the program is asked to calculate the solutions for a 9x9 problem. The averages here show that the times are all within 0.10 of a second of eachother – so not a massive difference. Despite this, it tells us that the quickest scheduling type for this particular problem is a static scheduling type as it seems to perform better overall than guided or default. At this point, it was looking like static scheduling was going to be the best choice going forward due to the slight performance increase that it provided over the other two scheduling types.

This notion changed, however, once the results for the N = 10 runs came in, which was the highest run that the system was capable of doing due to performance issues beyond that point. The results for the N = 10 can be seen in the figure provided on the following page.

Figure ‑: Scheduling Performance (N=10)

Figure 3-5 shows the performance difference between the scheduling types, similar to the previous figure, but for a 10x10 problem. Instead of being split by around 0.10 of a second, the scheduling types have seconds worth of performance difference. In a turnaround from the results of Figure 3-4, Figure 3-5 indicates that guided is the quickest of the three scheduling types. When taking into consideration the difference in time saved shown in both figures, it was decided to discard the default scheduling and decided between static and guided.

Guided provided a larger boost when faced with the 9x9 problem whereas guided performed better over the 10x10 problem. It can be assumed from this that guided may perform better as the scale of the problem increases.

The key thing to note too, is the difference in time being saved. While they both provide a bonus to the speed of the program execution during different scenarios, no matter how you look at it, guided scheduling provided the solution with a larger time benefit than that of static scheduling. While static provides milliseconds worth, guided provides full seconds worth of a performance bonus – and in a system such as this where the waiting times for calculation seem to get exponentially worse with the size of the calculation, it seemed that guided would be the best scheduling choice to stick with for the final program.

### 3.2.2. Parallel For (boardIsValid)

An attempt was also made to parallelise the boardIsValid method. The implementation was once again the simple use of a parallel for – however, on this occasion, it seemed to come at a large performance cost to the system. The implementation is shown below:

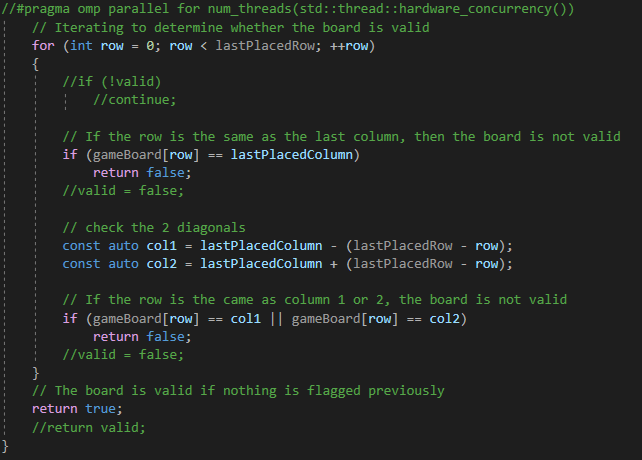


Figure ‑: Parallel For (boardIsValid)

Initially, the performance was in-line with what is expected from previous testing; relatively fast computation for the first 8 N values, followed by a more dramatic increase in execution time for N= 9 and 10. However, in the case of using a parallel for alongside the boardIsValid() method, the system began to struggle much more on N=8 and made testing N=9 and 10 unfeasible due to the length of time taken calculate the solutions for the final 2 N values. A comparison of N = 4 to 8 with the parallel for enabled vs disabled can be seen below and shows the difference in performance for those, given that N = 9 and 10 were impossible to test.

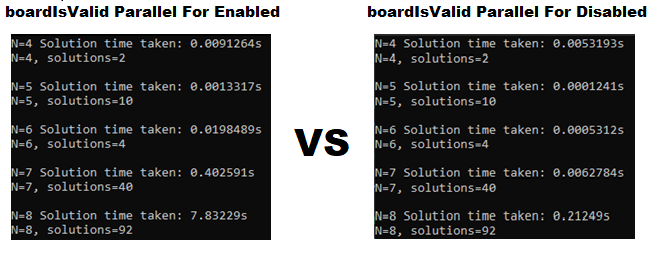


Figure ‑: Parallel For (boardIsValid) Enabled vs Disabled

When put into a line graph for visual comparison, the difference in performance in the complexities that could be handled become very clear:

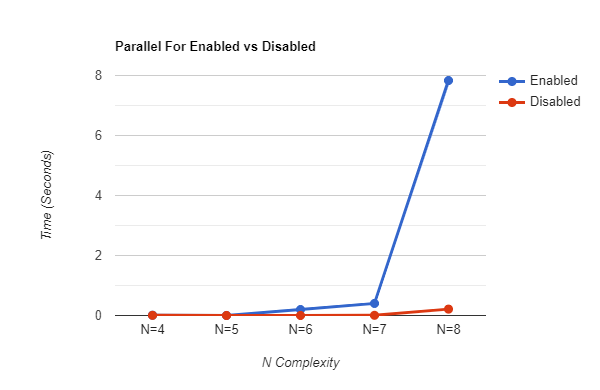


Figure ‑: Parallel For (boardIsValid) Comparison Graph

Evidentally, the times weren’t too far apart from eachother up to N = 7, however, when the Parallel For is enabled, the system suddenly struggles a lot more with N = 8 than it does with the Parallel For disabled. For that reason, it was decided to comment out the parallel for in the boardIsValid method and to remain with the single parallel for in the calculateSolutions method.

## OpenMP Performance Results

This section will discuss the results of the overall OpenMP performance tests. Below is a table of the results from the 10 test runs:

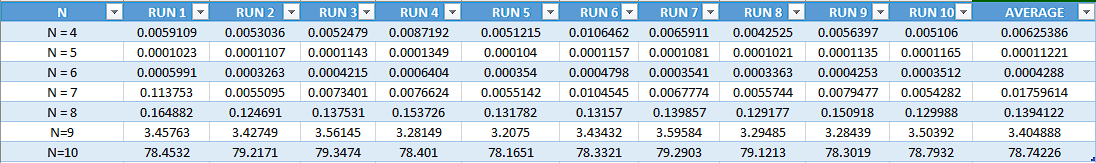


Figure ‑: OpenMP performance Table

Similar to the results of the Serial performance, it can be seen that the performance of the program gets worse as the complexity of the board increases – again, there is a large spike in performance loss when calculating anything above N = 9, which can be seen in the figure presented below:

Figure ‑: OpenMP Performance

Figure 3-10 clearly illustrates the spike mentioned previously and show how prominent the loss of performance is.

The reason for such a large time difference in this parallel solution is likely down to the nature of the N-Queens problem itself. It’s a problem that is notoriously difficult to parallelise – even when the solution is non-recursive, the problem is inherently sequential in nature. Each queen needs to be considered one at a time, and each queen must have the validity of its moves checked. This can be very intense on the CPU. When parallelised, this means that there is still a large amount of communication happening between the processes that are ensuring that no two queens can attack eachother, which when happening in parallel, is very taxing on the system.

## OpenMP vs Serial

Ideally, it would be useful to place both of the datasets of the Serial and OpenMP implementations into the same line graph and compare the times – however, the difference in times for the N = 9 and 10 is so large that it basically renders the visual elements of the Serial program uselss. As such, it is important to refer back to the two averaged individual datasets. When looking back to Figure 2:3, the upwards trajectory and the spike in time taken between N = 9 and 10 is visually apparent – this takes place over a matter of milliseconds. Comparing this to Figure 3:10, however, and we can see that while the time difference is much bigger, the trajectory is much the same. The time taken to calculate the solutions to the problems get longer when the complexity of the problem increases – starting off with a gentle increase in the time taken to calculate a solution from N = 4 to 8, but resulting in a massive spike from N = 9 to 10 as the complexity begins to rapidly increase.

The actual times, while they do not translate well over to a graph, can be seen in the table below. The tables shows a steady increase followed by a rapid upwards trend when the problem begins to reach N = 8. Again, these trends can be seen in Figures 2:3 and 3:10 for a visual representation of the steep climb.

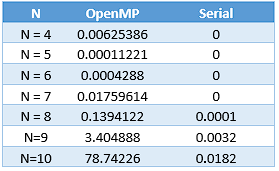


Figure ‑: Performance table for Serial and OpenMP

With this in mind, it seems reasonable to assume that if the program were able to keep going with reasonable performance past N = 10, we would see a case of exponential growth in a similar upwards trend from this point on. As is has been determined that the N-Queens problem, both in serial and parallel (OpenMP) both suffer from exponential performance loss, it seems reasonable to assume that the following section on parallelising the problem in CUDA will report a similar trend. Even if the CUDA program runs faster than the Serial and OpenMP ones – expect to see the same upwards trend starting at around N = 8.

# Parallel – GPU (CUDA)

## Method of Parallelisation

The first thoughts were directed towards the best way of using CUDA to parallelise the problem and were based around whether to attempt a 2D or 3D solution to the problem, with 2D referring to a 2D block of threads that are organised two dimentionally; x and y (E.g 2x2). The same is true for 3D, however, the threads are grouped in 3D blocks and organised three dimentionally; x, y and z ( e.g 2x2x2). With little time and having a more basic understanding of CUDA and GPU programming, 2D was chosen as the safer bet to implement effectively within the time allocated for this coursework.

Once this was decided, simple boiler code from the labs were copied over – the most important being the “gpuErrchk.h” file which will exit the program on the first CUDA error encountered. This is to stop any confusion of errors accumulating or causing unexpected behaviour from the beginning.

With the framework in place, the array based solution used for the OpenMP implementation was copied over for adaptation. With some minor changes, it was ready for use and for the parallelisation using CUDA. The first thing changed was the boardIsValid() Boolean – this was changed and was now prefixed with \_\_device\_\_ to indicate that it is to be ran on the GPU and not the host device. Naturally, this change ensured that the program would no longer run as other changes had not yet been implemented.

This was then followed up by breaking the calculateSolutions() method into two parts – the first being a new CUDA Kernel called checkQueenPos() with the \_\_global\_\_ prefix to indicate that it should be executed on the GPU and that its variables can be passed in from the host when the kernel is called upon by the host. The code can be seen in the figure below:

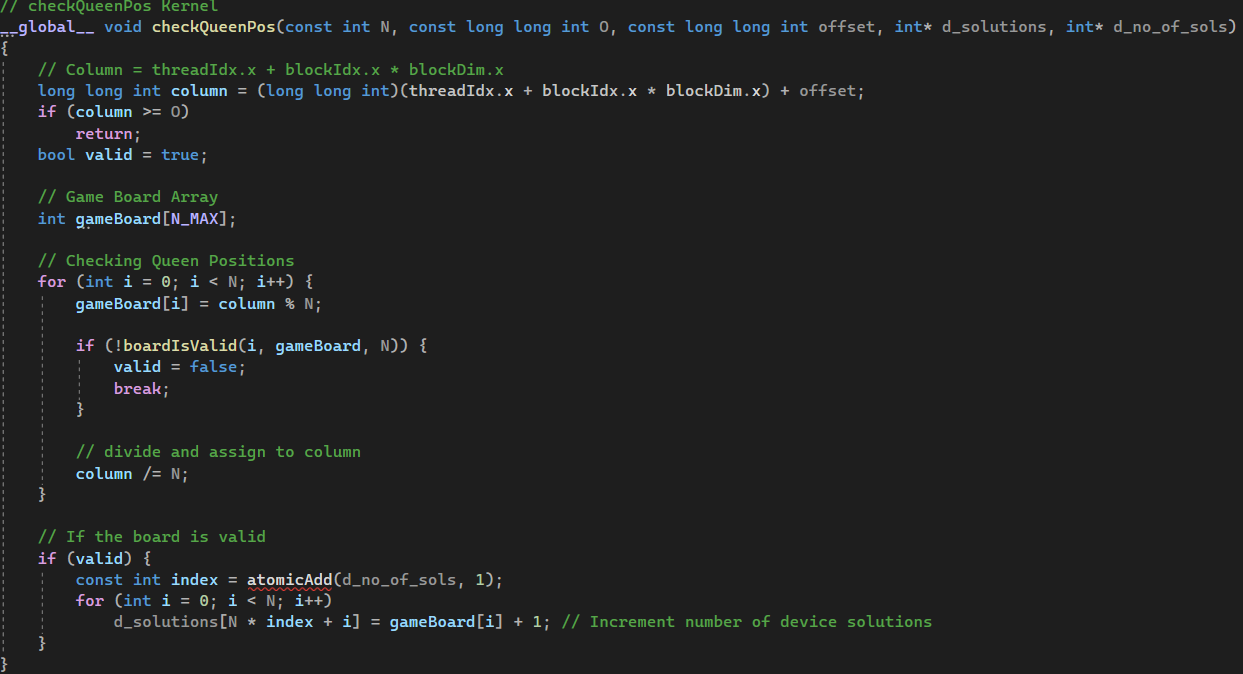


Figure ‑: CUDA checkQueenPos() Kernel

The kernel in Figure 4-1 is what is used to check the positioning of the queens on the board – it is very similar to the code seen in Figure 3-3. The two main things to note are the two final variables being passed in by reference called “d\_solutions” and “d\_no\_of\_sols”. These are marked as “d\_” to indicate that they are device variables. As well as this, the column is no longer being set as a loop index, but by adding the thread and block index and multiplying them by the block dimensions.

Moving on to the calculateSolutions() method, this method essentially sets up the host and device properly so that the variables are passed properly when the checkQueenPos() kernel is called. The source code for this entire method is too large to display – however snippets will be shown. The full code can be reviewed in the source code provided.

An example of the functionality within this method can be seen below:



Figure ‑: CUDA Memory Copy

Figure 4-2 shows memory copying – here, the destination of the copy is the d(device)\_no\_of\_sols which will receive the data being copied from the h(host)\_no\_of\_sols. This will be important when the checkQueenPos() kernel is called, shown next:

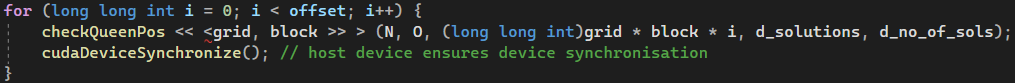


Figure ‑: Kernel call from host

When the checkQueenPos kernel is called, the host variables are passed into the device for use in the GPU. This Kernel will then perform the work set out within itself (see Figure 4-1) before returning and executing cudaDeviceSynchronize(). This piece of code is important as it ensures that each of the GPU computations have been completed before allowing the host to continue. Without this, there could be issues with missing results, errors and device overloading.

Following this, there is more CUDA memory management followed by the addition of solutions to the solutions array similar to what was seen back in figure 3-2:

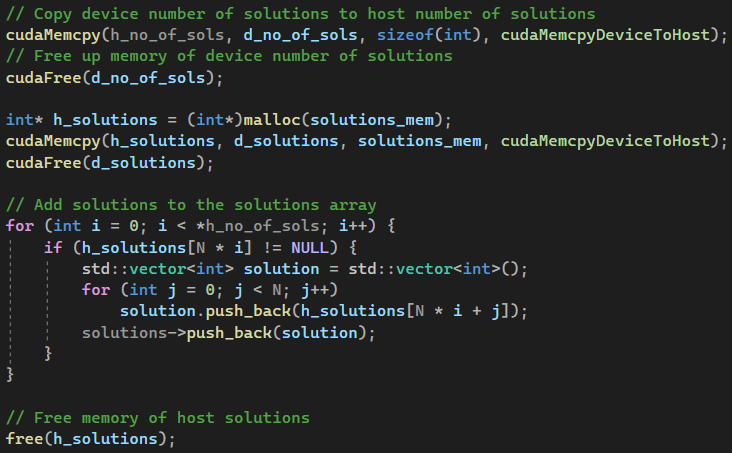


Figure ‑: Further CUDA Memory Management

Finally, we return to the calculateAllSolutions() method which is used to call the calculateSolutions() method, as well as measure performance and output information to the console. This is also where the solutions and no\_of\_sols are passed into the calculateSolutions() method.

## CUDA Performance Results

The results of the CUDA implementations performance can be seen in the table shown below:

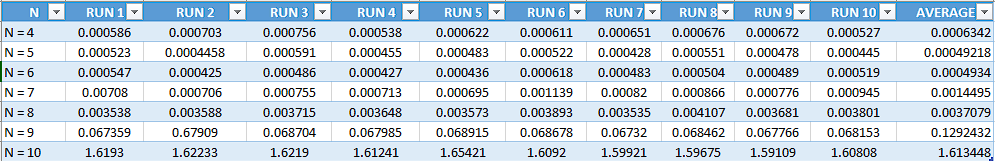


Figure ‑: CUDA Performance Results Table

In line with the predictions made previously, and mimicking the trend seen in both the serial and OpenMP programs, the CUDA performs the same way. We have a very slight deviation in the N complexity performance from N= 4 up to N = 8, but a general upwards curve can be seen. This curve then jumps up exponentially for N = 9 and N = 10. The results of this table can be seen visualised in the figure presented at the beginning of the following page.

Figure ‑: CUDA Performance

Figure 4-6 clearly illustrates the steady upwards curve discussed previously, followed by the rapid increase beginning at N=8. This again seems to suggest that the performance gets exponentially worse as the complexity of the NxN problem increases.

## CUDA vs Serial and OpenMP

When comparing the CUDA performance to that of both the Serial and OpenMP performances, there is a clear similarity in how they perform over the course of the N-Queens problem. All three programs exhibited the behaviour of exponential slowdown as the complexity of the problem increases, that being, each time the board increased in size, the time taken to calculate the solutions to the problem also increased in line with the number of solutions possible for the size of the board. Each program was very similar in that they all had a rather small performance loss from N=4 to N=8, the performance loss, however, began to increase at a much sharper rate from N=8 to N=10. The averaged times for each of the programs can be seen in the table below:

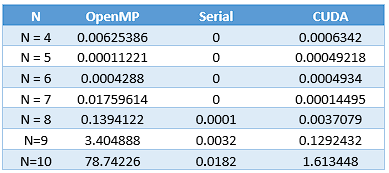


Figure ‑: Three Method Comparison Table

Figure 4-7 can then be translated into the visual representations show below, which clearly shows the gentle upwards trend followed by the sharp rise from N=8 onwards:

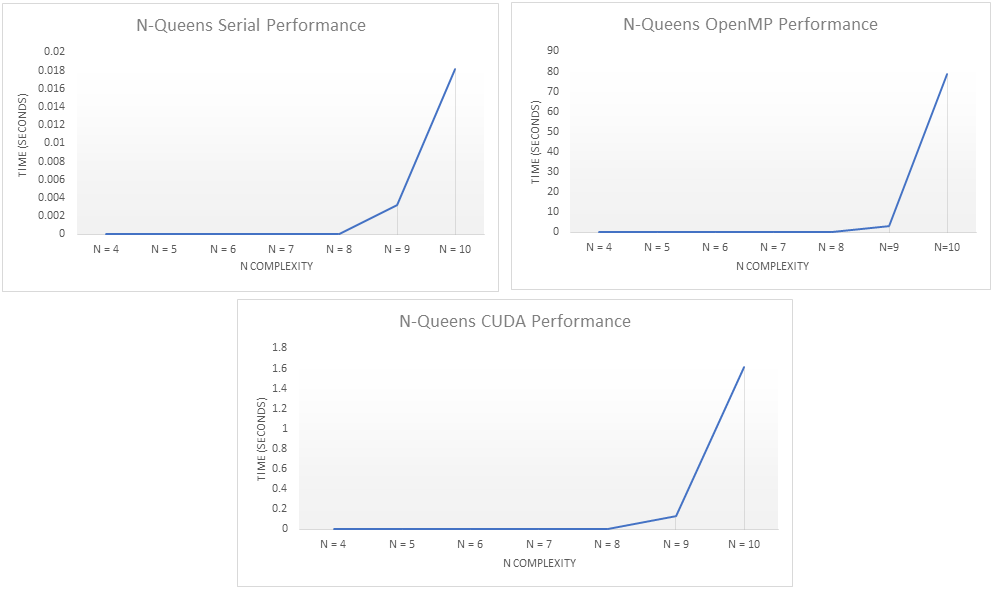


Figure ‑: Performance Graph Comparison

It has become clear that no matter how the solution is implemented, whether it be serial recursive and non-recursive, or a parallel method, that the N-Queens problem is destined to suffer from exponential performance decrease in line with the N complexity. It seems logical to assume from the results of the above N = 4 to N = 10 results that the upwards spike will continue for each of these implementations, and likely every implementation. This was to be expected due to the nature of the problem – the solution requires checking an increasingly complex board, so of course the time taken to solve the problem will increase, as it will take more computational power to find the all of the possible solutions to the problem.

It does seem that some solutions to the problem are better than others, however. In this case, with this specific implementation, it seems that the recursive serial program outperforms both parallel solutions, with the OpenMP solution lacking sorely behind in performance with over a minute of a difference in the N=10 calculation. With that in mind, the reason that OpenMP was so slow in comparison to CUDA is likely due to the fact that GPUs have a much larger number of cores in comparison to a CPU, the difference being that a GPU can have thousands vs a few dozen on a CPU. In addition to this, GPUs are built and optimized for parallel calculations, whereas CPUs are optimised more for serial operations.

# Conclusions

To conclude, it seems that the N-Queens problem lends itself very well to a recursive solution as the solutions to the problem are very interdependent – each solution requires the last in order to solve it properly. As a result of this, parallelising it by breaking the problem up can be very difficult – however as demonstrated, it is doable.

The results have shown that no matter how the problem is tackled, the problem will become exponentially more difficult to solve as the N number increases, so it is not abnormal to experience slowdown when parallelised that, when coupled with the recursive nature of the problem, makes a parallel solution slower than a serial solution.

That being said, when parallelising the problem, using a GPU to tackle the large number of solutions seems to be most efficient due to the higher number of cores available on a GPU than is available on a CPU.

# References

Datta, S. (2022, 11). *Backtracking Algorithms*. Retrieved from Baeldung.com: https://www.baeldung.com/cs/backtracking-algorithms#:~:text=Backtracking%20is%20an%20algorithmic%20technique,satisfy%20them%20will%20be%20removed.

Geeks for Geeks. (2022, 8 24). *Most important type of Algorithms*. Retrieved from Geeks For Geeks: https://www.geeksforgeeks.org/most-important-type-of-algorithms/?ref=lbp

Jaka's Corner. (2016, 6 13). *OpenMP: For & Scheduling*. Retrieved from Jaka's Corner: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Oxford College. (n.d.). *The N-Queens Problem & Backtracking*. Retrieved from Oxford Emory Math Center: http://mathcenter.oxford.emory.edu/site/cs171/nQueensProblemAndBacktracking/

Simran. (2021, 6). *4 Queens Problem without Backtracking (Brute Force Approach)*. Retrieved from Medium.com: https://simran17102004.medium.com/4-queens-problem-without-backtracking-brute-force-approach-3acab2d9119

Stack Exchange. (2019, 11 1). *Brute-Force algorithm in C++*. Retrieved from Stack Exchange Code Review: https://codereview.stackexchange.com/questions/231696/brute-force-algorithm-in-c