40618059

SET10108

Concurrent and Parallel Systems

Contents

[**Hardware Setup:** 2](#_Toc153366445)

[OpenMP 3](#_Toc153366446)

[Non-parallel Recursion to OpenMP Parallelisation 3](#_Toc153366447)

[Second Iteration 4](#_Toc153366448)

[GPU - CUDA - Implementation 5](#_Toc153366449)

[GPU – CUDA – Future Iterations 6](#_Toc153366450)

[Performance Capturing 7](#_Toc153366451)

[Serial – Time 7](#_Toc153366452)

[Serial – Hardware Monitoring Before and During Execution 7](#_Toc153366453)

[OpenMP – Iteration 1 – Time 8](#_Toc153366454)

[OpenMP – Iteration 1 – Hardware Monitoring Before and During Execution 8](#_Toc153366455)

[OpenMP – Iteration 2 – Time 9](#_Toc153366456)

[OpenMP – Iteration 2 – Hardware Monitoring Before and During Execution 9](#_Toc153366457)

[CUDA – Time 10](#_Toc153366458)

[CUDA – Hardware Monitoring Before and During Execution 10](#_Toc153366459)

[Time Comparisons: 11](#_Toc153366460)

[Performance Conclusion: 13](#_Toc153366461)

[Scalability: 14](#_Toc153366462)

[Overall Results: 16](#_Toc153366463)

[References 17](#_Toc153366464)

# **Hardware Setup:**

**AMD Ryzen 5 3600 CPU**

**NVIDIA RTX4060Ti GPU**

**16GB DIMM4 RAM**

**1TB SAMSUNG SSD**

**B450-APRO-MAX MOTHERBOARD**

# OpenMP

## Non-parallel Recursion to OpenMP Parallelisation

In order to transform the code from non-parallel recursive format to a parallelisation style, the code will have OpenMP integrated into it. OpenMP does not work well with recursion, as such it will need to be reformulated in order for the code to run in parallel.

Although it sounds daunting, it was quite simple in the steps towards the solution. Firstly, before reformulating the code, I read through and made myself aware of what the original code does to better modify it in the future.

The next step is to start modifying the code. Firstly, I dealt with some redundant code, I removed a section called “(print)” which would execute if “print” command is called, which would then run a block of code displaying each attempt at a solution being formed. This code was removed in the OpenMP version but kept in the original untouched version. It was removed since for starters, it wasn’t being used and secondly, it makes the program run slower and makes reading results more difficult.

Next, the main block of code needs to be addressed. However, first the appropriate headers must be used, including <omp.h> and <thread> to allow openMP to work. Now, with the main it will be almost identical to the original, however it will use openMP’s functionality of “#pragma omp parallel for num\_threads(num\_threads)” to parallelise the for loop getting the solutions with a defined set of threads from the hardware.

The following module to be called is caclulateAllSolutions, which works in tandem with CalculateSolutionsRecursive. This is where the major changes need to be implemented. Firstly, CalculateSolutionsRecursive is commented out for now. After spending some time researching techniques used for the removal of recursion, one such technique that seemed popular due to its efficiency is a backtracking method. While researching into the different ways past developers tackled this problem I began to notice the main similarity they followed. From the more beginner friendly methods to the expertly crafted and complicated solutions I started from this rough draft of pseudocode:

1. While rows exist {
2. Create column incrementing current row
3. If statement checking row does not exceed board size, otherwise backtrack
4. Set current column to the gameboard column
5. Pass gameboard to boardIsValidSoFar for check
6. If board full, solution stored
7. Otherwise continue to next row

It’s important to note that, the boardIsValidSoFar from the original code is unchanged as it still does exactly what is needed.

While implementing the pseudocode, I defined the column as a vector to work with the gameboard, row as an int and to for timing each solution I added a clock timer from when the calculation is about to start it stops right as it finishes.

So, now the program is running in a non-recursive OpenMP parallel format using pragma omp with a set number of available threads from the hardware. The results seen in the performance section will detail the speed at which the program completed at.

## Second Iteration

Now, the program is running well and working as intended. However, parallel programming is all about speed and performance gains, so I took my modified code and looked for tweaks that could be done to improve speed.

While analysing my code, I paid special attention to the values and how they are declared. This drew my attention to gameboard and solutions, both stored as a vector with pushback being used to store solutions. Based on previous experience with the first coursework and feedback received, along with personal code teaching I know that this can end up hindering performance for this openMP parallel version.

As such, these values were converted to dynamic arrays which also meant column would need to be converted to an array as well. The dynamic arrays provided quite a performance boost, this will be shown and discussed in the performance section.

The first iteration of OpenMP took 0.77 seconds to find the 12th solution in one run. Compared to the serial version, it was around .1 seconds slower. The second iterations, however, saw a big speed increase. With the 12th solution taking only .27 seconds beating the original recursive format and serial version.

It is important to always keep in mind however that, hardware plays an important role in this. Better hardware could be even faster, or more middle of the line hardware could run slower.

# GPU - CUDA - Implementation

As for the GPU implementation, this required a lot more research and understanding in order for me to properly take on the task. While researching, I came across a now deprecated course that was once used to teach GPU programming, specifically CUDA. The original, free to enrol, course is no longer on the original website, but I managed to find a GitHub.

After studying the slides, practicing on the problem programs, and making myself aware of CUDA, I looked towards potential solutions, however it was more complicated than for OpenMP and less straightforward.

As such, I took the OpenMP code and began to think, boardIsValidSoFar and Main would remain the almost identical same, the building blocks. As for what would change, the main would need to include an error check on the GPU before allowing the program to proceed. This will use the gpuErrchk.h macro provided.

Next, I set boardIsValidSoFar to \_device\_ allowing it to use the GPU and take advantage of CUDA kernels, allowing for parallelisation.

Now, the main part of the code. For this section, it will be declared as a collection of “Result” containing the time is takes to get all of the solutions for one N and the total number of solutions for that N. In similar fashion to openMP, there will be a row and columns, defined as grid and block. As for defining how many Kernels will be run to find all solutions, this can be found by multiplying N by itself.

Memory management is very important during this phase, as mishandling or using it where not needed could severely impact performance. For this section, I had solutionsMemory set to the size of the gameboard multiplied by 5. The number 5 was used as it gives a good amount of memory space in the event there is a lot of solutions, while performing well. Originally I had it set to 7 but after a few tests, it was found that 5 executed faster.

Furthermore, I then declare two more memory values using a device pointer, these are d\_allSolutions and d\_totalSolutions. The first contains every possible solution for a board, the second contains only the number total for the board. d\_allSolutions size is set to the solutionMemory calculated before, and totalSolution is started at 0 done by utilising cudaMalloc, an allocation for memory in CUDA..

In order to ensure that Kernels are being executed, we start off by manually initialising one, then defining the grid and blocks using the Threads defined (512 is currently set, this number functioned best on my hardware with 1024 not working at all and 256 performing slower).

Now, after a validation check that solutions are possible, there will be a call to a new module called \_\_global\_\_ boardPossibillites tagging it as a kernel of the GPU. The code in here will be very similar to the recursive manner of the serial version, however this code will be run in kernels and in parallel with each other.

Returning back to systemStart, cudaDeviceSynchronize is called. This CUDA function synchronizes the GPU and CPU to ensure that all kernels have completed their tasks before moving on. After this, cudaMemcpy will be used to copy the results from the GPU to the host (cpu) into results.totalSolution. After this, d\_totalSolutions is cleared as it’s done its job and can be wiped for performance reasons with memory management.

Then the results are displayed with the time taken to complete.

# GPU – CUDA – Future Iterations

Unfortunately, due to time constraints, I was unable to find the time to make further iterations of the CUDA version. However, I will still discuss possible changes and modifications that would provide possible performance gains.

With the initial CUDA implementation being done via 2D board generation for the N-Queens problem, another possible route to take is a 3D board version. This could possibly introduce performance gains and more of an optimised algorithm.

Adding a third dimension to the board provides a new structure and offers and different approach to the problem. The dimensions would contain rows, columns, and diagonals adding a new layer for the N-Queens problem.

Another potential iteration would be a hybrid CPU/GPU build. This would allow the workload to be intelligently distributed amongst the CPU cores and GPU threads. This type of approach would allow the system to scale more and take more intensive amounts of workload.

# Performance Capturing

In this section, we will go over each of the 3 methods with their iterations and capture their performances (time taken to complete, hardware usage).

## Serial – Time

|  |  |
| --- | --- |
| **N Queens Value** | **Result (Microseconds)** |
| **4** | **11** |
| **5** | **36** |
| **6** | **42** |
| **7** | **218** |
| **8** | **833** |
| **9** | **4276** |
| **10** | **19556** |

Firstly, starting off with serial. In order to capture the timing, the code had to be slightly added to, with a timer. Using averages from 5 runs for each N value up to 10, the following graph will display each average.

## Serial – Hardware Monitoring Before and During Execution

Before running:  
A graph with numbers and a graph

Description automatically generated with medium confidence

During Execution:

A screen shot of a graph

Description automatically generated

During execution, CPU utilisation and speed is being ramped up at expected. Thread count is not that high due to the lack of threads and parallelisation.

## OpenMP – Iteration 1 – Time

Next, performance will be taken the same way as serial. With the 5 runs complete, these are the results:

|  |  |
| --- | --- |
| **N Queens Value** | **Result (Microseconds)** |
| **4** | **31** |
| **5** | **105** |
| **6** | **191** |
| **7** | **612** |
| **8** | **1885** |
| **9** | **6546** |
| **10** | **27413** |

## OpenMP – Iteration 1 – Hardware Monitoring Before and During Execution

Before executing:

A white background with black text

Description automatically generated

After executing:

A white background with black text

Description automatically generated

Note the increase in thread usage before and during running. Compared to serial, threads are being used more during execution and the CPU is being utilised well.

## OpenMP – Iteration 2 – Time

Next, performance will be taken the same way as the last two. With the 5 runs complete, these are the results:

|  |  |
| --- | --- |
| **N Queens Value** | **Result (Microseconds)** |
| **4** | **2** |
| **5** | **7** |
| **6** | **22** |
| **7** | **79** |
| **8** | **334** |
| **9** | **2172** |
| **10** | **9423** |

## OpenMP – Iteration 2 – Hardware Monitoring Before and During Execution

Before executing:

A white background with black text and black text

Description automatically generated

After executing:

A white background with black text

Description automatically generated

Performance measurement is almost identical increases compared to iteration 1, however with the less performance heavy version of the code CPU utilisation is not as high since it does not need to be.

## CUDA – Time

|  |  |
| --- | --- |
| **N Queens Value** | **Result (Microseconds)** |
| **4** | **4184** |
| **5** | **958** |
| **6** | **2732** |
| **7** | **3950** |
| **8** | **10175** |
| **9** | **55155** |
| **10** | **1123969** |

Next, performance taken for CUDA in the same way as before. With the 5 runs complete, these are the results:

## CUDA – Hardware Monitoring Before and During Execution

Before executing: CPU Above, GPU below

A white background with black text and numbers

Description automatically generated A computer screen shot of a computer

Description automatically generated

After executing:

A white background with black text and black letters

Description automatically generated A computer screen shot of a computer

Description automatically generated with medium confidence

Performance measurement is unique in this case, for the other iterations the GPU is not being used for any algorithmic reasons, but with CUDA it is a key component. GPU utilization is at a peak of 100% while running the algorithm with 64% utilisation.

# Time Comparisons:

N4:

N5:

N6:

N7:

N8:

N9:

N10:

# Performance Conclusion:

From the performance of the software execution time, hardware utilisation and time comparisons from each version of the code the following conclusions can be made:

* Serialisation was faster than the initial OpenMP in every case, the second iteration of OpenMP was faster than serialisation in every case and CUDA compilation was the slowest in every case.
* Serialisation compared to every other version of the program didn’t utilise as much of the hardware as others, the first iteration had a good utilisation of thread usage and CPU usage/speed, the second iteration was a very good balance, using around the same number of threads and a little less intensive on the CPU than the first iteration while performing faster. GPU was much more demanding than the rest, reaching a higher CPU usage and fully 100% utilising the GPU.

Combining all the averages per N for serial, then OpenMP 1, OpenMP 2, and CUD we get the following:

* 25072 microseconds for Serial
* 37793 microseconds for OpenMP 1
* 10939 microseconds for OpenMP 2
* 1204123 microseconds for CUDA

From these numbers, we can deduce how much faster as a percentage each version is against the original serial, then against each other.

* Serial is faster than OpenMP 1 by 50.74%
* Serial is slower than OpenMP 2 by 56.37%.
* Serial is faster than CUDA by 4702.66%.
* OpenMP1 is slower than OpenMP 2 by 71.06%.
* OpenMP 1 is faster than CUDA by 3086.1%.
* OpenMP 2 is faster than CUDA by 10907.61%.

# Scalability:

This section will showcase the program being extended past its requirement of N = 10, for the case of this section there will be a time limit of 1 minute 8 seconds for reasonable testing time purposes. The N limit will be 20 and testing will see which gets the most N of queens within the 1-minute time limit. Starting with serial:

Serial:

A screenshot of a computer

Description automatically generated

OpenMP 1:

A screenshot of a computer

Description automatically generated

OpenMP 2:

A screenshot of a computer

Description automatically generated

CUDA:

A screenshot of a computer

Description automatically generated

From the results, serial, and both version of OpenMP showcase their ability to scale within a reasonable amount of time with the number of queens being increased. CUDA, however, could not reach above 12 queens while the other 3 could reach 14. CUDA has potential to not only catch up but also has a chance to be faster than the rest of the implementations with more iterations and modifications to the code such as making it 3D.

# Overall Results:

In the end, the iteration that showed the fastest output with good performance was the second iteration of OpenMP. Through the use of parallelisation and thread usage, it iterates through possible solutions faster and more efficiently than serial version when using an array, otherwise serial is faster slightly.

# References

Udacity. (2022). Introduction to Parallel Programming class code [GitHub repository]. GitHub. <https://github.com/udacity/cs344>

Feinbube, Frank & Rabe, Bernhard & Löwis, Martin & Polze, Andreas. (2010). NQueens on CUDA: Optimization Issues. 63-70. 10.1109/ISPDC.2010.22, [Download citation of NQueens on CUDA: Optimization Issues (researchgate.net)](https://www.researchgate.net/publication/221261956_NQueens_on_CUDA_Optimization_Issues/citation/download)

Babis Koniaris. (2023). SET10108 Concurrent and Parallel Systems