ENCE464 Assignment 2: Computer Architecture

Group 13: Jack Duignan, Isaac Cone, Daniel Hawes

The result across many cube sizes with 300 iterations and 20 threads is shown by Figure 1.

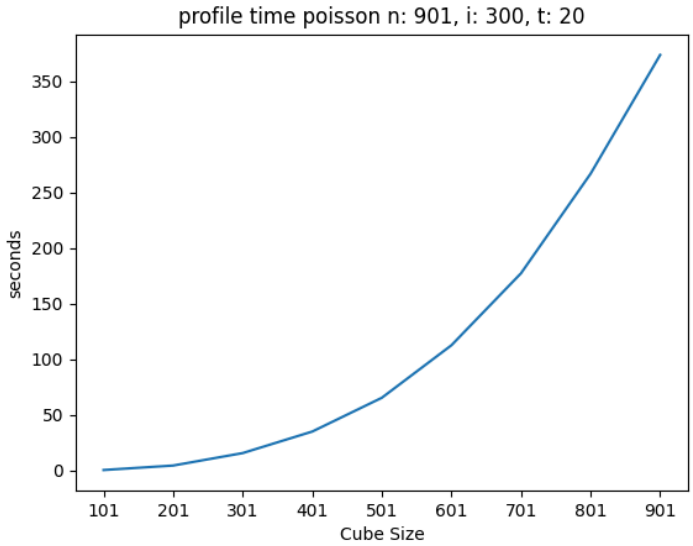

Figure 1: A complete run of the firmware across all cube sizes with 300 iterations and 20 threads.

Processor Architecture

The Central Processing Unit (CPU) described in this section is the AMD Ryzen 9 6900HX. This is a high-performance mobile CPU based on a 6nm process node. Its eight identical cores have two threads each for a total of 16 processing units and use the AMD64 (x86-64) instruction set architecture. The overall CPU structure is shown in Figure 2.

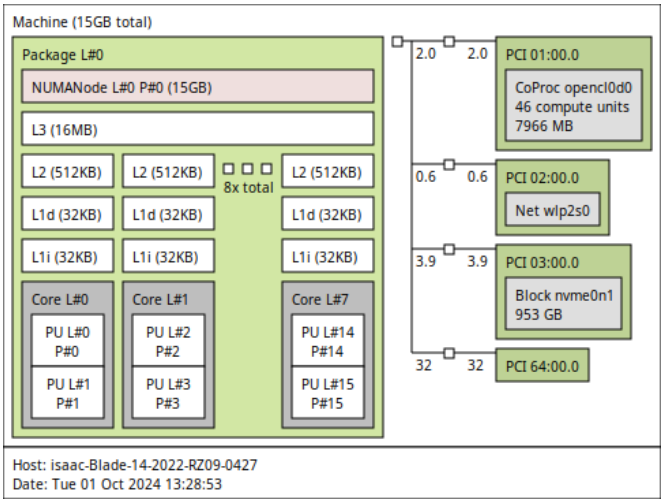

Figure 2: Central Processing Unit (CPU) architecture for the x86-64 AMD Ryzen 9 6900HX.

As mentioned, the cores use the AMD64 architecture, shown in Figure 3 [1]. Each core has several Functional Units (FUs). These include execution units such as Floating Point Units (FPUs), Arithmetic Logic Units (ALUs), memory units and I/O units for interfacing with the system. By having units specialised for various tasks, the core is able to achieve more than one instruction per clock cycle using pipelining. Each core supports multiple threads as not all functional units can be used simultaneously for example one thread can perform memory access, while the other completes a floating point operation. This is not as performant as two separate cores running a single thread but does provide an increase over a single thread.

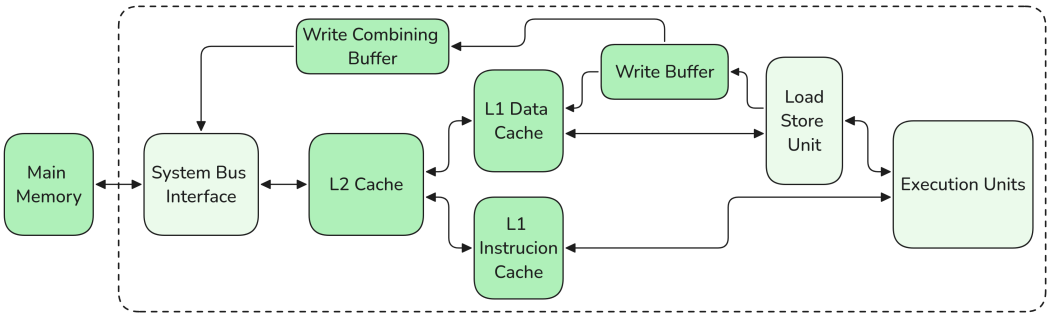

Figure 3: Typical AMD64 core architecture.

The computer has 15GB of available DRAM which is used for storage of volatile memory. As DRAM is comparatively slow inside the CPU, there are multiple levels of higher-speed cache memory which implement a modified Harvard architecture. This cache is split into three layers: the CPU has a shared 16MB L3 cache across all eight cores. Then each core has its own 512 kB L2 Cache and 32kB L1 data and instruction caches. This caching allows both data and instructions to be read in parallel from the shared address space, improving efficiency. Caching is important as shown in Table 1, where the instructions to access data in the different memory levels grow rapidly.

AMD64 CPUs use the x86-64 instruction set architecture developed by Intel. This is an extension of the ubiquitous x86 architecture that introduces a 64-bit bus while retaining backwards compatibility. The use of a 64-bit architecture allows for much higher addresses of RAM to be read, theoretically up to four exabytes. The x86-64 architecture also has 64-bit general-purpose registers and support for more complicated instructions that allow for more efficient operations on specific data types. An example

of this is the Single Instruction Multiple Data (SIMD) instructions that allow operation on vector data instead of single memory address.

Multithreading - easy - Daniel

Multithreading was used throughout this project to improve CPU utilisation this was done by separating the program from working on a single core to multiple. This is achieved by separating the Poisson algorithm across multiple worker threads. Each thread is responsible for computing a portion of the cube, divided across slices in the k dimension. The start of a slice is calculated through a simple formula seen below:

$$k_{\text{start}} = 1 + \frac{i\lceil(N-2)\rceil}{t} \tag{1}$$

Where i is the thread number N is the number of nodes and t is the number of threads. The end of a slice is calculated in a similar way as:

$$k_{\text{end}} = (i+1)\frac{\lceil(N-2)\rceil}{t} + 1, k_{\text{end}} = \begin{cases} N-1 \text{ if } k_{\text{end}} > N-1 \\ k_{\text{end}} \quad \text{otherwise} \end{cases} \tag{2}$$

Each worker thread is passed the program variables (`curr`, `next` etc.) and the slice it is assigned, it then performs the required iterations applying Von Neumann and inner iterations where required. To prevent race conditions caused by parallel execution a barrier is used. The barrier uses a `pthread_barrier_t` with a limit equal to the number of worker threads. When all threads have completed an iteration and entered the barrier it lifts thus ensuring they are synchronised. After each thread completes its calculations for one iteration, the buffers for the next and current iterations need to be changed. This is done by swapping the pointer's addresses which removes the need for expensive memory operations.

The results of the multithreading implementation can be seen in Figure 4. The results show that as the number of threads is increased the execution time decreases. This is as expected as the program can make use of multiple cores in parallel. The solution reaches an execution time asymptote at approximately 20 threads after this point the execution time is near constant. The minimum execution times occur at 12 and 24 threads this is due to the processor that is being used to produce these results. These results were captured on a Intel i5 12400f processor which has 6 cores and 12 threads. The most common operation completed by the Poisson software is floating point arithmetic which requires a floating point unit. Two threads at a time can use the FPU as it takes time to load memory back and forth. Thus the maximum number of threads that can complete floating-point operations simultaneously is 12 which is reflected in the minimum computational time value.

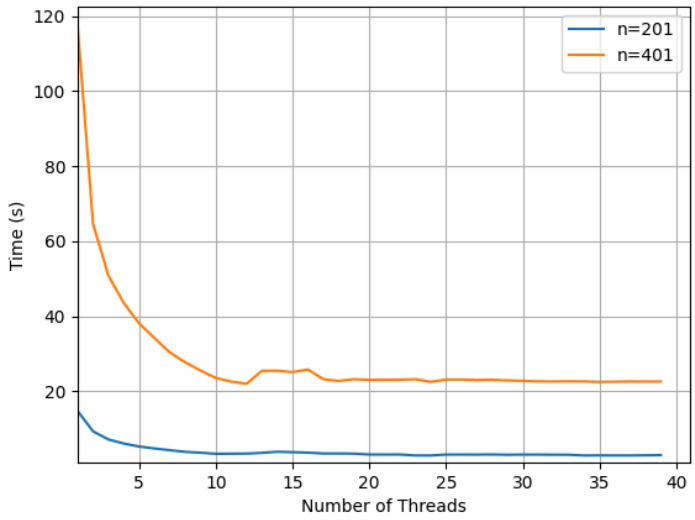

Figure 4: A comparison of the poisson solver software execution times with different number of threads used

Cache - hard

The Poisson algorithm requires a large number of memory operations to load the data from the respective arrays and store the result. These memory operations can add significant overhead. This is because the read/write time of DRAM is significantly higher than the execution time of the CPU. Modern CPUs minimise this impact with small high-speed memory caches. These store recently accessed data, minimising costly DRAM access operations. As discussed in Section 1, the largest cache on the analysed CPU is 16MB. This is not enough to store even one of the arrays needed for a 401 node cube which has a size of 515MB. This means the cache can only store small sections of memory. Optimising the use of cached memory improves cache utilisation, which can significantly improve program execution time.

The largest number of memory operations in the code occurs during the computation of the inner nodes of the cube. This uses three nested `for` loops that iterate over the layers, columns, and rows. These iterations can be executed in any order to achieve the correct result. One way to ensure optimal cache utilisation is to consider spatial locality when accessing memory. This means that neighbouring data in memory will be accessed consecutively, so the required data will almost always be located in the cache for each node. To achieve this, the optimal iteration scheme is for the inner loop to iterate along the i-axis (a row), as it increments by a single item in memory each iteration.

The performance for different iteration schemes is shown in Figure 5. As seen in the plot, iteration schemes where the inner most loop is the k-axis outperform any other scheme by a significant margin. With the execution time being directly proportional to what iteration level the x-axis is performed. The difference in execution time is significant with the i, k, j iteration scheme being five times slower than the optimal k, j, i scheme.

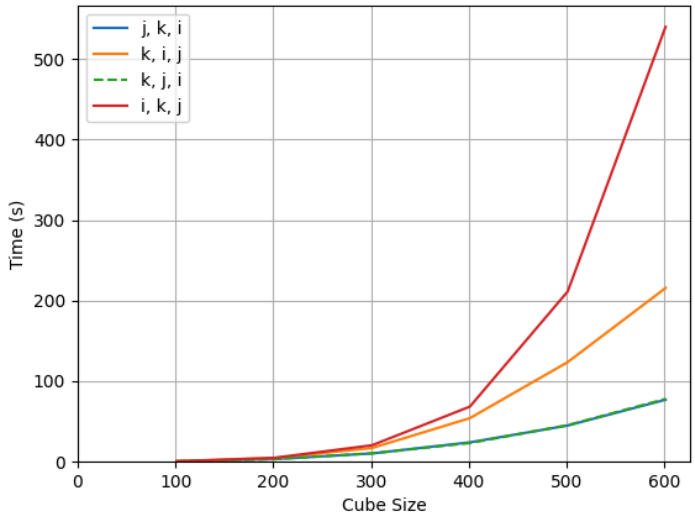

Figure 5: A comparison of execution time for various cube sizes with a range of iteration schemes.

The performance differences in iteration schemes can be explained by examining the number of cache misses occurring during execution. If the CPU needs to access data that is not in the cache, this is a miss, and the CPU must then read this data from DRAM which is significantly slower. Table 1 compares the L1 cache read and write miss rates for the same program using the best (k, j, i) and worst (i, k, j) performing iteration schemes respectively. The total number of memory accesses is equal, but the optimal iteration method has approximately five times fewer misses, indicating significantly improved cache utilisation hence the lower execution time.

Table 1: Memory access cost at various cache levels for the AMD Ryzen 6900HX.

Iteration	Dmr	Dmr %	Dmw	Dmw %
i, k, j	908,929,991	5.03	296,704,937	96.7
k, j, i	168,924,763	0.93	42,754,806	14.0

Profiling - easy - Jack

Profiling was used throughout all stages of this project's development to identify which areas of the program would benefit from optimisation. From these results, optimisations were made to the code to reduce execution time. When selecting areas of the code to optimise the sections called most often were prioritised as these give a larger performance benefit than optimising slower less frequent functions. To make profiling easier the various components of the code were compartmentalised into functions, while this does add some execution time (due to stack overheads) it allows the profiling tool gprof to provide more granular results.

Profiling was conducted on both optimised and non-optimised code to gain a holistic understanding of the program's execution. A breakdown of the execution times and call counts for a non-optimised run of the program with a 201-node cube over 300 iterations using 20 threads can be seen in Table 2. The result of profiling using debug optimisation (-0g) on the same cube size as before can be found in Table 3.

Table 2: GProf results for a non-optimised run of the program with 201 nodes 300 iterations and 30 threads.

Function	Percentage	Call Count	Time per call (ms)
`poisson_iteration_inner_slice`	96.47%	2251	24.05
`apply_von_neuman_boundary_slice`	3.53%	2222	0.91
`wait_to_copy`	0%	2400	0
Setup	0%	1	0

Table 3: GProf results for a 0g optimised run of the program with 201 nodes 300 iterations and 30 threads.

Function	Percentage	Call Count	Call Time (ms)
`poisson_iteration_inner_slice`	93.45%	2269	11.45
`apply_von_neuman_boundary_slice`	6.55%	2275	0.80
Barrier waits cumulative	0%	2389	0
Setup	0%	1	0

The results found in Table 2 and 3 show that in both runs the largest time cost is the iteration over the inner slice of the cube. This is expected as it performs the majority of the floating point operations in the software. As expected the compiler optimisations have reduced the iteration time by more than half. Interestingly the Von Neumann boundary condition execution time was only reduced by 12%. This may be due to the significant number of conditional checks required by this function which cannot be optimised out.

In earlier iterations of the program, the Von Neumann boundary was called at every inner loop of the main Poisson iteration. Based on profiling the team was able to identify this as a bottleneck as it is unnecessary to call this for all the inner nodes and moved the updates to its own self-contained iteration that only iterates over the outside nodes. Reducing the number of conditional checks needed thus reducing the execution time as there are fewer instructions per iteration.

Another example of profiling helping in the optimisation of code is the barrier wait that is used to synchronise the threads. Originally the team hypothesised that these waits would greatly increase the execution time as threads take different amounts of time to complete due to CPU allocation and the way the cube nodes are divided amongst them. By profiling the code with these barriers implemented it was discovered as can be seen in Table 2 that the barrier wait does not add any appreciable execution time and in the optimised version of the code seen in Table 3 is even expanded out of its function and executed in the code itself with no function call overhead. Without profiling this would have been much harder to identify.

Compiler Optimisation

Modern compilers, particularly for the C Programming Language, are extremely well optimised for general use. This makes it near-impossible for a programmer to "beat" the performance of compiler-generate assembly code. Compiler optimisation modifies the standard operation of a compiler to produce assembly code that is optimised for some specific purpose, usually execution time or program size. These optimisations result in tradeoffs. For example, a program optimised for execution speed may be significantly larger than an unoptimised program. Because of this, compiler optimisation are disabled by default, and are enabled using compiler flags.

There are a number

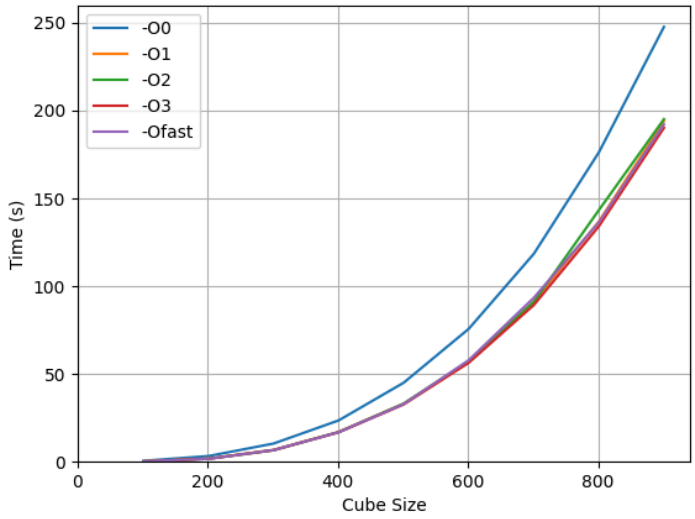

Figure 6: A comparison of the poisson solver software execution times with different compiler optimisations.

Individual Topic 1 Jack Duignan - Branch Prediction

Execution on a CPU is optimised using pipelining which allows a single clock cycle single instruction execution. It does this by loading the next set of instructions while the previous instructions are being executed. When branch instructions occur a control hazard develops as the CPU cannot predict which set of instructions to load. This can cause costly pipeline flushes if the CPU loads the wrong. To attempt to mitigate the number of times this pipeline flush occurs the CPU predicts which way execution will go before the conditional branch is executed. This is branch prediction and is implemented in various ways across CPU architectures.

In the Poisson algorithm software, there are several control hazards caused by conditional branching. One of the most significant of these is during the inner iteration of the Poisson equation. In this operation, the code uses several conditional instructions to check the location of the current node and apply the correct iteration. This results in our implementation of the iteration having 12 conditional jump instructions per node update (if complied without optimisations). To improve the speed of our implementation it was decided to reduce the number of these jumps where possible to reduce the amount of branch prediction required by the CPU.

The reason our implementation required so many conditional instructions is that we use the same nested `for` loop to iterate over both the Von Neumann boundary (the Dirichlet boundary can be applied once during initialisation) and the inner nodes of the cube. It was hypothesised that moving these out of the same loop and reducing the Von Neumann iteration to only over the outer nodes would reduce the number of condition branch control hazards per iteration and thus overall. This was achieved by splitting each iteration into two components first the Von Neumann boundary is applied to only the nodes required then the nested loop only updated the inner nodes. This change completely removed the conditional instructions in the main iteration loop (which is called most often) removing the largest control hazard from the program.

A comparison of the software with and without conditional control hazards can be seen in Figure 7. This shows that the program's execution has been reduced by 10%. With the real benefits occurring at larger cube sizes as more iterations mean more conditional branch issues. This result is expected as by reducing the number of conditional branches the CPU can optimise the use of pipelining as the number of possible flushes is reduced. This change also has the added benefit of reducing the number of instructions per iteration. This is due to the Von Neumann boundary application only iterating over nodes that it will need to be applied to as opposed to all nodes in the cube. Crucially this time gap still appears when optimisation is applied showing that the problem cannot be solved by the compiler.

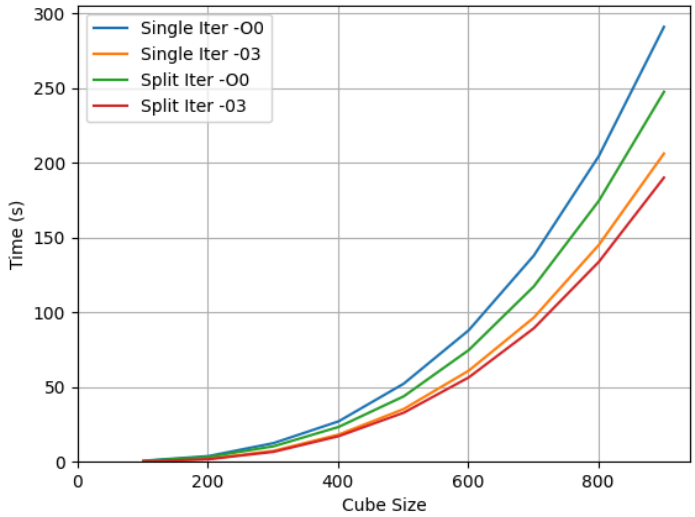

Figure 7: A comparison of the poisson solver software execution times with and without conditional branch reduction to reduce control hazards.

Individual Topic 2 Isaac Cone - GPU

Graphics Processing Units (GPUs) are specialised hardware with many processing optimised for performing repeated operations. GPUs are significantly faster than CPUs in some applications due to massively parallel execution, a much higher memory bandwidth, and greater instruction throughput. This section will discuss how a GPU, specifically NVIDIA 3070 Ti laptop GPU, can be leveraged to enhance the performance of the Poisson algorithm. The 3070 Ti architecture shown in Figure 8 consists of 48 Streaming Multiprocessors (SM) each with 128 CUDA cores for a total of 6144 cores. These cores run up to eight threads each. The hardware executes a custom kernel function using the Compute Unified Device Architecture (CUDA) API [2]. The API uses variably sized blocks of threads and automatically handles the allocation of threads and blocks to the hardware. The poisson algorithm involves a single repeated operation, making it ideal for implementation on the GPU. It is expected that the 3070 Ti, which has 6144 CUDA cores will significantly outperform a CPU, particularly on larger cube sizes.

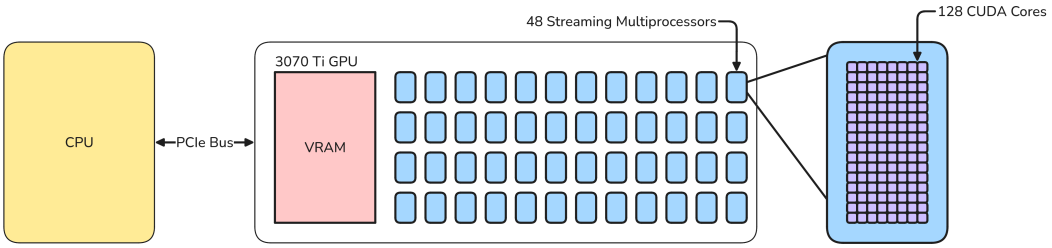

Figure 8: Multiprocessor and CUDA core architecture of the NVIDIA 3070 Ti GPU.

To simplify CUDA implementation, the Poisson algorithm was reduced to a single kernel function, no longer optimised for the CPU. To leverage the GPU VRAM bandwidth, computations are performed entirely in the GPU VRAM. This has the limitation that the data size cannot exceed the GPU VRAM capacity, 8GB in this case. For execution on the 3070 Ti, the threads per block were set to 512 in an 8-thread cube. The gridSize cube of blocks is then dynamically sized for a given N by Equation 3.

$$\text{gridSize} = \frac{N + \text{blockSize} - 1}{\text{blockSize}} \tag{3}$$

Figure 9 compares the CUDA program to the optimal CPU program. For a 101 cube, the CPU outperforms the GPU. This is because the overhead from copying to VRAM exceeds the benefit of increased thread count. As cube sizes grow, the GPU significantly outperforms the CPU. This is because the advantage of massively parallel execution becomes more pronounced with more computations. Above a 601-sized cube, data use exceeds 8GB, preventing execution on the 3070 Ti. This memory limitation could be avoided by batching data into smaller sections, but this would add overhead. Additionally, as the CUDA program was written as a proof of concept without further optimisation, there are likely CUDA-specific optimisations that could realise significant performance improvements.

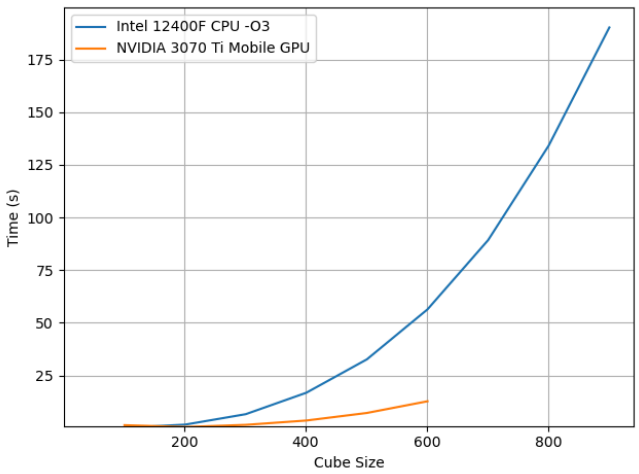

Figure 9: NVIDIA CUDA implementation of the poisson algorithm compared to optimal CPU solution.

Individual Topic 3 Daniel Hawes - SIMD

Single Instruction, Multiple Data (SIMD) is a technique used to perform the same operation on multiple data points simulataneously. This is particularly useful in applications where the same operation is performed over large data sets, such as large 3D arrays. This section will discuss how SIMD can optimise the poisson algorithm. SIMD uses the Advanced Vector Extension (AVX) instruction set. The results of the implementation are from tests on the AMD Ryzen 7 4700U CPU.

The AVX2 instruction set allows for 256-bit wide registers, which can hold 4x 64-bit double precision floating point numbers. This can be applied to the poisson algorithm by performing the same operation on 4 nodes simultaneously. The implementation of SIMD in the poisson algorithm calculates 4 nodes across the i dimension to solve for the following:

$$V_{\{i:i+4,j,k,n+1\}} \tag{4}$$

A calculation is manually performed at the end of the calculation if the inner slice is not divisble by 4 to finish off the remaining nodes in the i dimention.

Between `i` and `i+4` we can load each node surrounding the calculated node into a vector register. This results in 6 different vector registers holding values for the calculation. Since all the vectors will use the add operation, the add operation can be done simultaneously for the 4 nodes loaded in each register. We can hypothesise that there should be approximately a 4x increase on the inner slices of the cube since it can now calculate 4 nodes at once. With this hypothesis in mind, the results of the SIMD implementation can be seen in Figure 10. These results come from SIMD only implemented in the inner slice calculation since SIMD is only very useful when the same operation is performed on multiple data points multiple times.

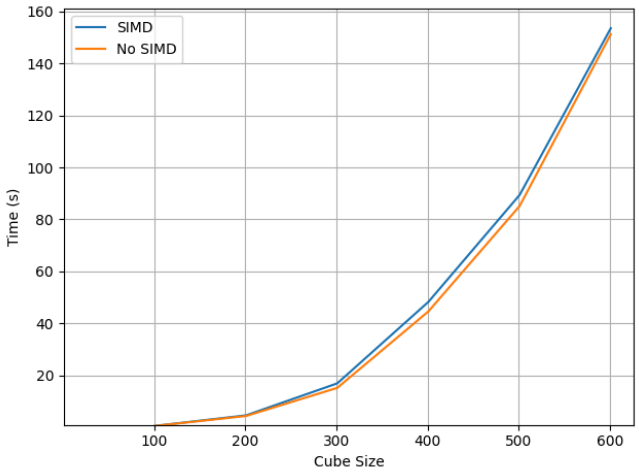

Figure 10: Comparison with SIMD implemented in the inner slice and without SIMD.

The results show that despite the implementation of SIMD in the inner slice layers, the execution time is decreased. Based on SIMD principles, this conclusion doesn't seem correct, so after profiling the SIMD functions it was discovered that the increase in speed is due to the loading of vector registers. Since in the normal calculation we can take the numbers directly from memory and perform the calculation this makes it more efficient to SIMD, as SIMD needs to load the numbers from memory into new vector registers to perform the calculation. Profiling indicated that this memory load step took 49.58% of the CPU time for the calculation. The actual calculation step only takes 7.5% of the program time. So it's clear that the time from this SIMD implementation bottlenecks at the memory load step.

It is clear that the principle of SIMD does same some clock cycles for the execution of the calculation, but the time taken to set up the calculation is longer than the time saved from doing it normally. Thus, the SIMD implementation was not included in the final poisson code.

References

[1] I. Advanced Micro Devices, “AMD64 Architecture Programmer’s Manual Volume 2: System Programming.” 2024. [Online]. Available: https://www.amd.com/system/files/2020-10/amd64-architecture-programmers-manual-volume-2-system-programming.pdf

[2] N. Corporation, “CUDA C Programming Guide.” 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/