

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.

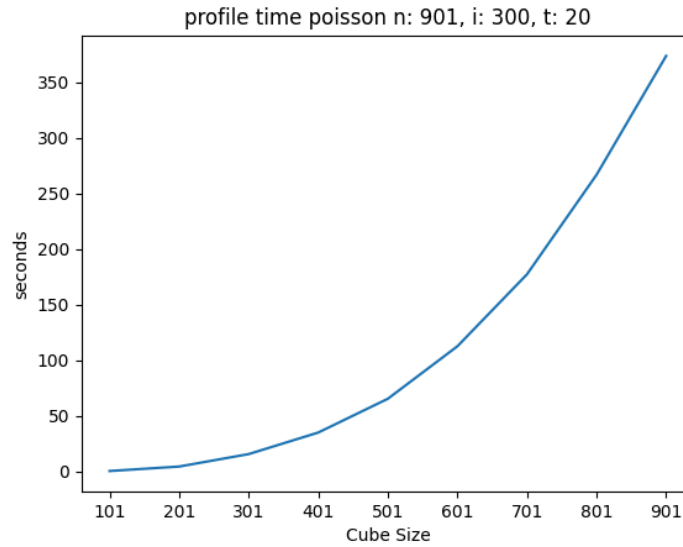


Figure 1: A complete run of the firmware across all cube sizes with 300 iterations and 20 threads.

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 uses the AMD64 (x86-64) instruction set architecture. The overall CPU structure is shown in Figure 2.

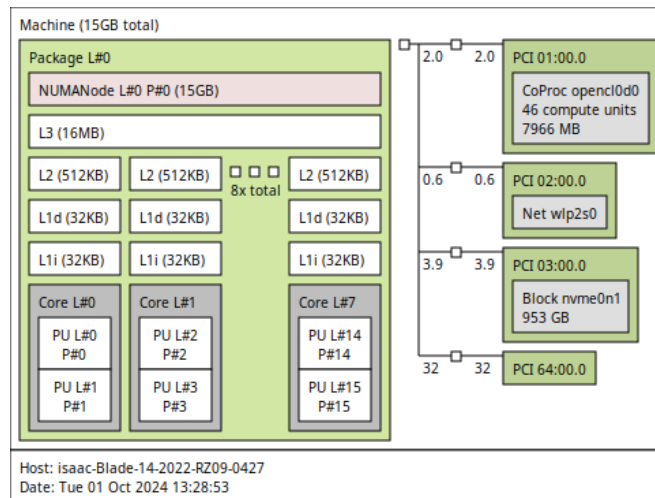


Figure 2: Central Processing Unit (CPU) architecture for the x86-64 AMD Ryzen 9 6900HX.

As mentioned, the cores use the AMD64 architecture, which is shown in Figure 3 [1]. Each core has several Functional Units (FUs). These include execution units such as Floating Point Units (FPUs) and Arithmetic Logic Units (ALUs), and memory and I/O units for interfacing with the system. By having units specialised for various tasks, the core is able to achieve less than one cycle per instruction by using pipelining. The performance of the core is limited by the number of FUs, and since these are shared between threads, two threads are not guaranteed to achieve the performance of one core per thread. Despite this, maximising FU utilisation can drastically improve performance.

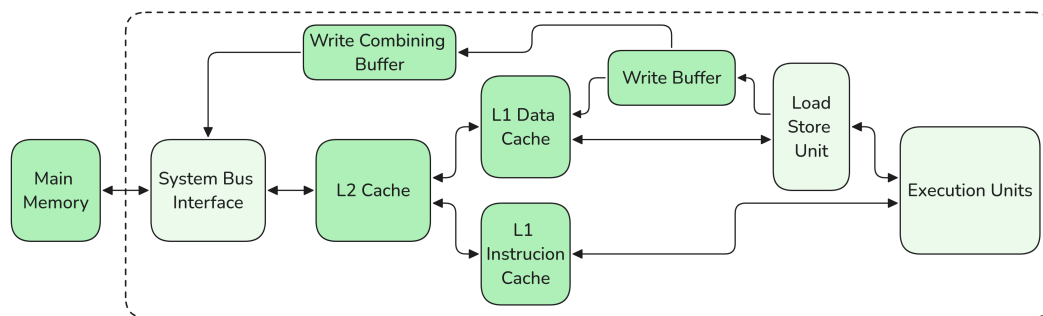


Figure 3: Typical AMD64 core architecture.

The computer has 15GB of available DRAM. Inside the CPU, there are multiple levels of memory caching which implement a modified-harvard architecture. The CPU has a shared 16MB L3 cache across all eight cores. Each core then 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 grows rapidly.

AMD64 CPUs use the x86-64 instruction set architecture. This is an extension of the ubiquitous x86 architecture that introduces 64 bit computing 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 are able to efficiently execute complex operations. Some of these include the use Single Instruction Multiple Data (SIMD) instructions to operate on vectors of data at once.

Multithreading - easy - Daniel

Multithreading was used 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 calculation 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:

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

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:

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

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 the 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 next and current need to be changed. This is done by swapping the pointers 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; However, the minimum 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 12500f 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. But there is no additional benefit beyond this as the FPUs are saturated. Which is why the minimum value occurs at 12 threads.

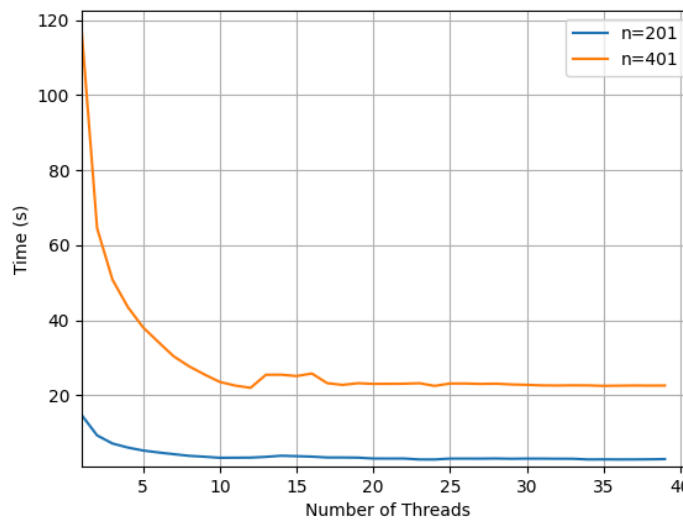


Figure 4: A comparison of the poisson solver software execution times with different number of threads used

Cache - hard

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

Memory	Read Instructions	Access Count
L1 Cache	X	X
L2 Cache	X	X
L3 Cache	X	X
System	X	X

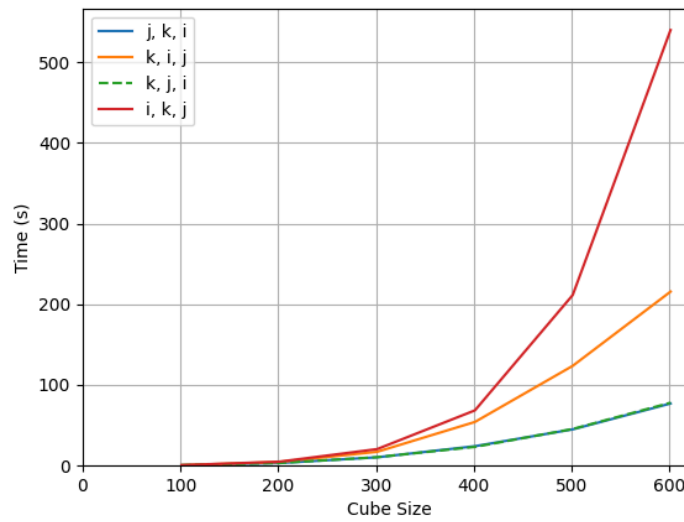


Figure 5: A comparison of the poisson solver software execution times with different iteration schemes showing the effect of cache utilisation.

Profiling - easy - Jack

Profiling was used through-out all stages of this projects development to identify which areas of the program would benefit from optimisation. From these results optimisations where 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 where 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 wholistic understanding of the programs 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 (-Og) 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 Og 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 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 bottle neck as it is un-necessary to call this for all if the inner nodes and move 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 less instructions per iteration.

Another example of profiling helping in optimisation of code is with the barrier waits that are 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 waits do not add any appreciable execution time and in the optimised version of the code seen in Table 3 are even expanded out of there respective functions and executed in the code itself with no function call overhead. Without profiling this would have been much harder to identify and solve.

Compiler Optimisation - easy

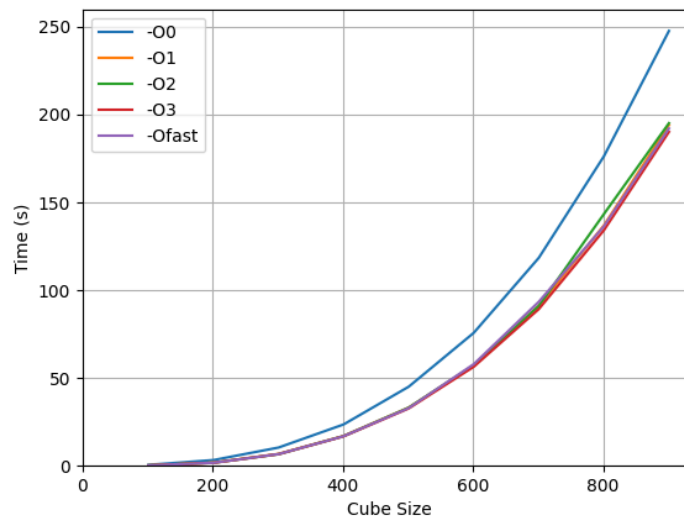


Figure 6: A comparison of the poisson solver software execution times with different compiler optimisations.

Individual Topic 1 Jack Duignan - Branch Prediction

Execution on a CPU is optimised using pipelineing which allows 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 is unsure which set of instructions to load. This can cause costly pipeline flushes if the CPU predicts 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 iteration 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 has use several conditional instructions to check the location of the current node and apply the correct iteration. This results in our implementing of the iteration to have 12 conditional jump instructions per node update (if compiled 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 dircile 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 to 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.

The a comparison of the software with and without conditional control hazards can be seen in Figure 7. This shows that the programs execution has been reduced by 10%. With the real benefits occurring at larger cube sizes as the more iterations mean more conditional branch issues. This result is expected as by reducing the number of conditional branches the CPU can optimise use of pipelining as the number of possible pipeline 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 gap still appears when optimisation is applied showing that the problem cannot be solved by the complier.

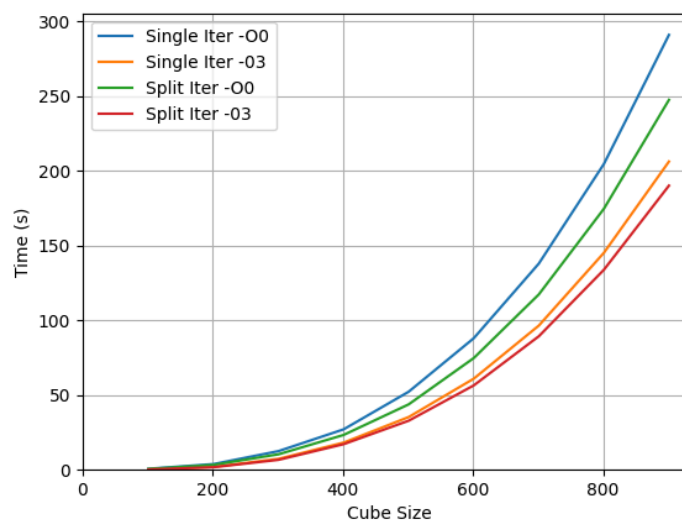


Figure 7: A comparison of the poisson solver software execution times with and without conditional branch reduction to reduce control hazards.

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 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.

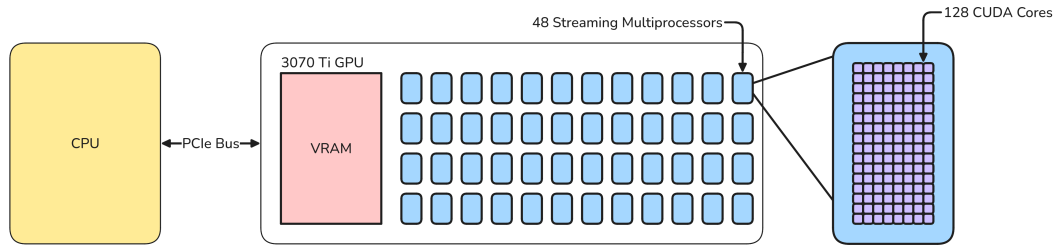


Figure 8: Multiprocessor and CUDA core architecture of the NVIDIA 3070 Ti GPU.

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 was set to 512 in an 8 thread cube. The gridSize cube of blocks is then dynamically sized for a given N by Equation 3.

$$\text{gridSize} = \frac{N + \text{blockSize} - 1}{\text{blockSize}} \quad (3)$$

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, it is likely there are CUDA-specific optimisations that could realise significant performance improvements.

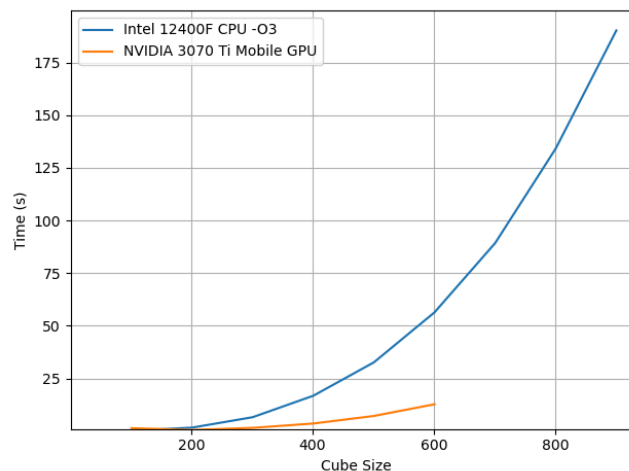


Figure 9: NVIDIA CUDA implementation of the poisson algorithm compared to optimal CPU solution.

Individual Topic 3 Daniel Hawes - SIMD

Single Instruction, Multiple Data (SIMD) is a technique used to perform the same operation on multiple data points simultaneously. This is particularly useful in applications where the same operation is performed over large data sets, such as with large 3D arrays.

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/>