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.

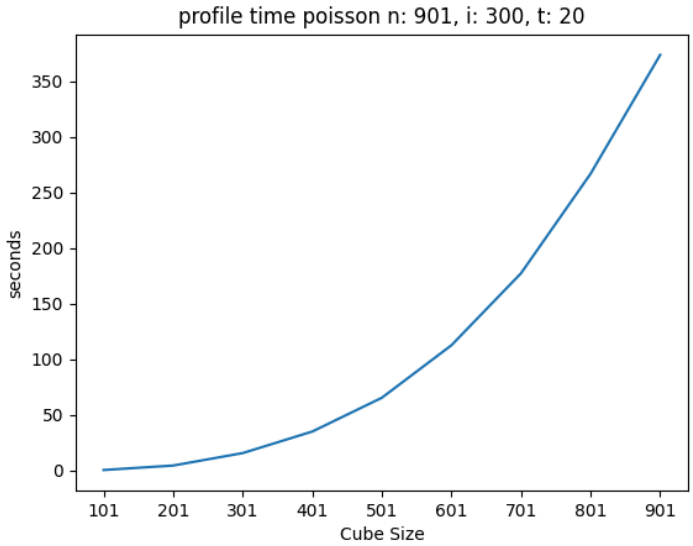

Figure 1: A complete run of the firmware across all cube sizes with 300 iterations and 20 threads.

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.

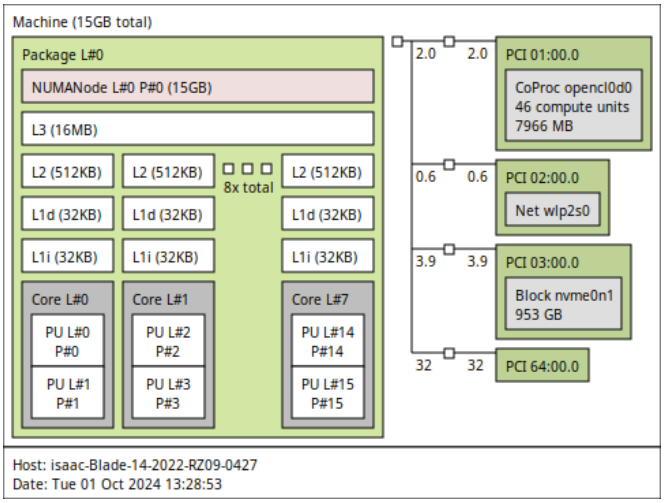

Figure 2: Central Processing Unit (CPU) architecture for the x86-64 AMD Ryzen 9 6900HX.

As mentioned, the cores use the AMD64 archiecture, 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 specialsed 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, maximisng FU utilisation can drastically improve performance.

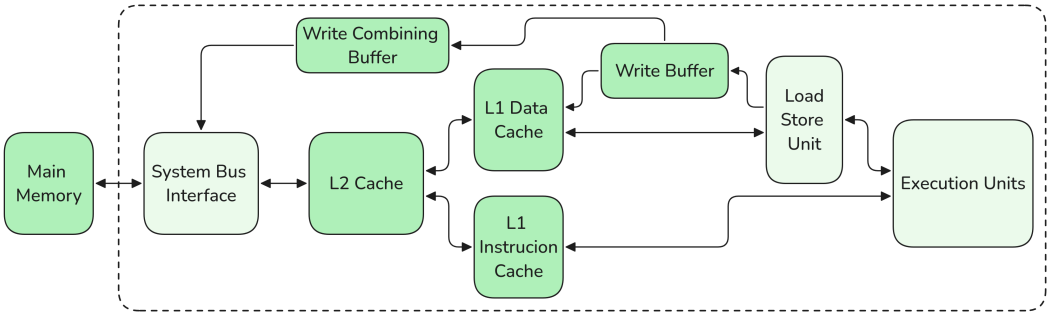

Figure 3: Typical AMD64 core architecture.

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 can be used to separate a program from working on a single core to multiple. This optimizes the CPU performance. Multithreading is achieved by separating the calculation across multiple worker threads. Each thread is responsible for computing a portion of the cube, split across the k dimension.

Each worker thread takes in the cube size, pointers to the current and next buffer of the cube, the slice dimensions and the number of iterations. This information is passed through the 'workerThread_t' structure.

Each thread works on a portion of the cube defined by it's i, j and k parameters given when the thread is created. Each thread works on its portion of the cube and either copies its calculation to memory of the 'next' buffer. After the last thread completes it calculations for the iteration, the contents of the current buffer are copied to a temporary buffer, and the next buffer is copied to the current buffer. (DIAGRAM?) To ensure that race conditions don't occur during this process, a mutex is used to ensure that no threads are writing to the same memory location at the same time.

To ensure there is synchronization between threads swapping buffers, a barrier is used at the end if each iteration. The barrier uses 'pthread_barrier_t'. This barrier is a counter that is incremented each time a thread reaches the end of it's calculations. When the counter reaches the number of threads, the barrier is removed and the threads can continue to the next iteration. When the last thread lifts the barrier, the threads can begin their calculations for the next iteration.

row selection memcopy barrier

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

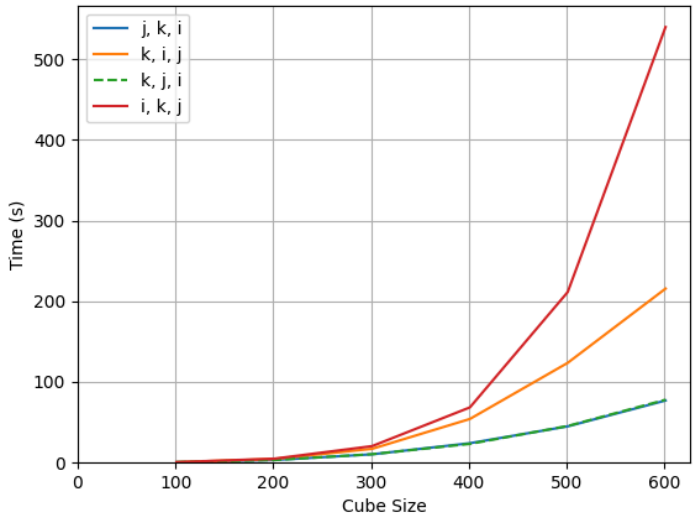

Figure 4: A comparison of the poisson solver software execution times with different iteration schemes showing the effect of cache utilisation.

Profiling - easy - Jack

Profiling was used through-out all stages of this projects development. This was done to identify which areas of the program where the slowest and how often these slow areas where called. 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 then 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 final optimisation parameters (-03) 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	Call Count	Time per call (ms)
`poisson_iteration_inner_slice`	5965	1.25
`memcopy_3D`	5977	0.61
`apply_von_neuman_boundary_slice`	5956	0.05
Barrier waits cumulative	11945	0
Setup	0	0

Table 3: GProf results for a O3 optimised run of the program with 201 nodes 300 iterations and 30 threads.

Function	Call Count	Time per call (us)
`poisson_iteration_inner_slice`	5958	676.40
`memcopy_3D`	5971	410.32
`apply_von_neuman_boundary_slice`	5926	37.12
Barrier waits cumulative	N/A	N/A
Setup	0	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. The next highest execution time is the application of the Von Neumann boundary.

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 unnecessary 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. 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.

- Python script
- gprof outputs and how they were used

Compiler Optimisation - easy

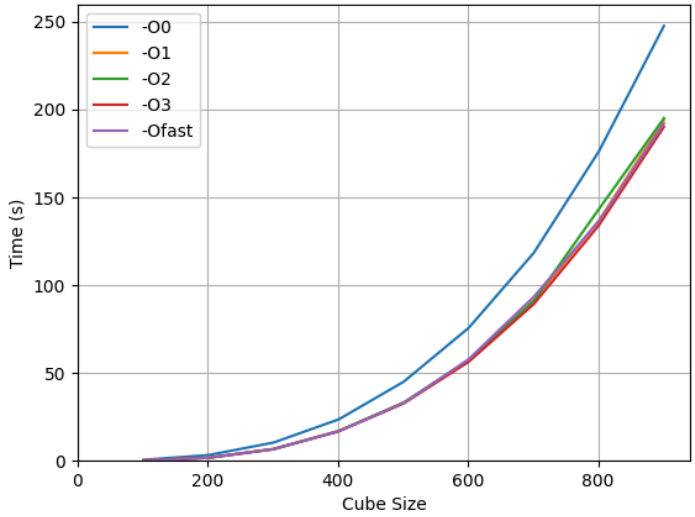

Figure 5: A comparison of the poisson solver software execution times with different compiler optimisations.

Individual Topic 1 Jack Duignan - Branch Prediction

Execution on a CPU is optimised using pipelineing which allows single clock cycle, single instruction excution. 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 poission 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 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 dircilc 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 old program with conditional control hazards and without can be seen in Figure 6. This figure shows that the programs execution has been reduced by 10%. With the real benefits occur at the large 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 will need to be applied to as opposed to all nodes in the cube.

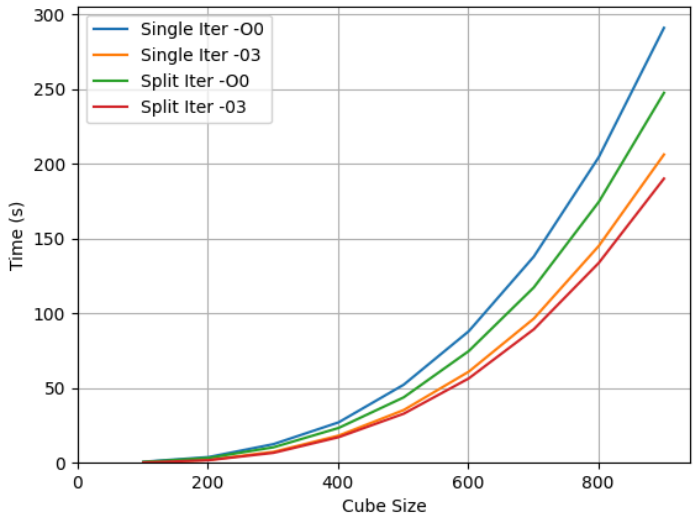

Figure 6: A comparison of the poisson solver software execution times with and without conditional branch reduction to reduce control hazards.

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. GPUs are useful across a range of domains including artificial intelligence and modelling complex real-world phenomena such as weather. This section will discuss how a GPU, specifically NVIDIA 3070 Ti laptop GPU, can be leveraged to enhance the performance of the poisson algorithm.

NVIDIA GPU hardware can execute any user-defined operation using the Compute Unified Device Architecture (CUDA) API [2]. 3070 Ti CUDA architecture consists of 48 Streaming Multiprocessors (SM) each with 128 CUDA cores, as shown in Figure 7. These CUDA cores can each run up to eight threads. The CUDA API is configured to execute the kernel with a set number of blocks, which are allocated one to each SM, and a number of threads per block. The API automatically handles the allocation of computation to these threads. The poisson iterations are ideal for GPU computation, so it is expected that the 3070 Ti with 6144 CUDA cores will significantly outperform a CPU.

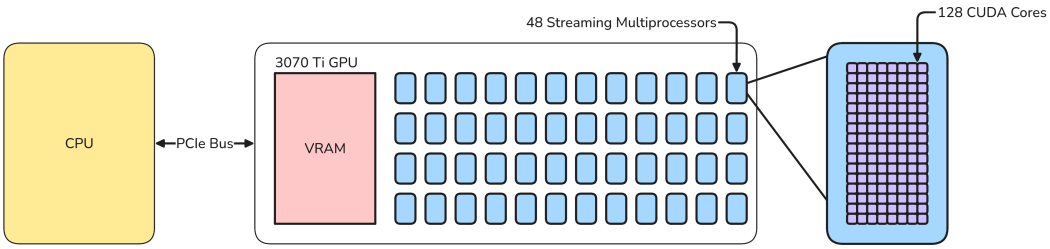

Figure 7: Multiprocessor and CUDA core architecture of the NVIDIA 3070 Ti GPU.

To simplify CUDA implementation, the poisson algorithm was reduced to a single kernel function, meaning it is unoptimised for the CPU. To leverage the GPU memory bandwidth the data needed for computation is entirely copied into the GPU VRAM. A limitation was identified from doing so, as for larger cubes the 8GB of VRAM on the 3070 Ti is too small to hold all of the data without more advanced memory management. The implementation in this report uses #BLKS blocks with #THRDS each.

The GPU performance is compared to the optimal CPU result in Figure 8. This shows the GPU reaches the same solution an order of magnitude faster, far outperforming any possible optimisations. The suspected VRAM limitations do impact results for cubes sized over 701. This is because the requried data for computation exceeds the 8GB of VRAM in the GPU. This would be avoided by using a GPU with more VRAM or by using memory management to segment the cube for copying in stages. The latter solution would introduce additional overhead however. Overall the GPU result is as expected.

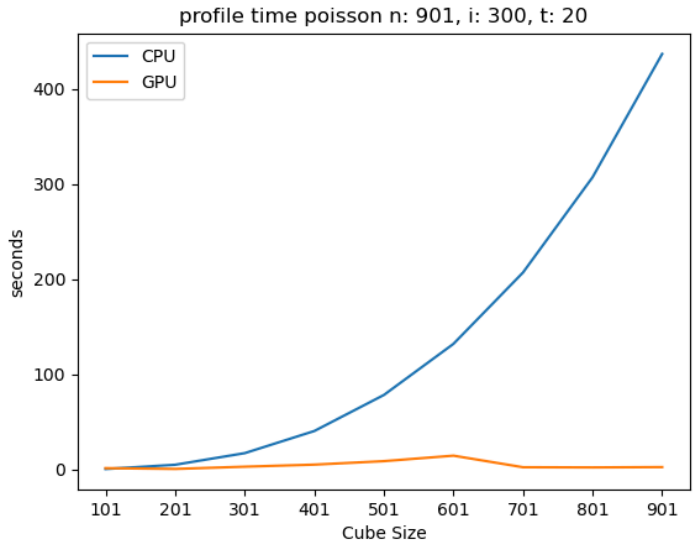

Figure 8: NEED TO REDO THIS FOR NICER PLOT (COMPARE TO BEST CPU ONE)

Individual Topic 3 Daniel Hawes - SIMD

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/