ENCE464 Assignment 2: Computer Architecture

Group 13: Jack Duignan, Isaac Cone, Daniel Hawes

The result of running the completed program over a range of cube sizes for 300 iterations using 20 threads can be found in Figure 1.

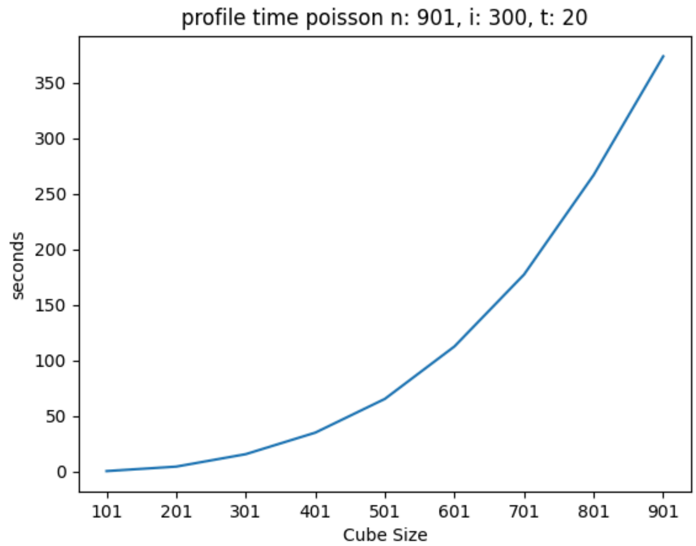

Figure 1: A complete run of the firmware across all cube sizes with 300 iterations and 20 threads.

Architecture Overview

The Central Processing Unit (CPU) described in this section is the AMD Ryzen 9 6900HX. Released in 2022, this CPU 8 identical cores with 2 threads per core for a total of 16 logical cores. The CPU uses the x86-64 instruction set architecture. The CPU structure is shown in Figure 2.

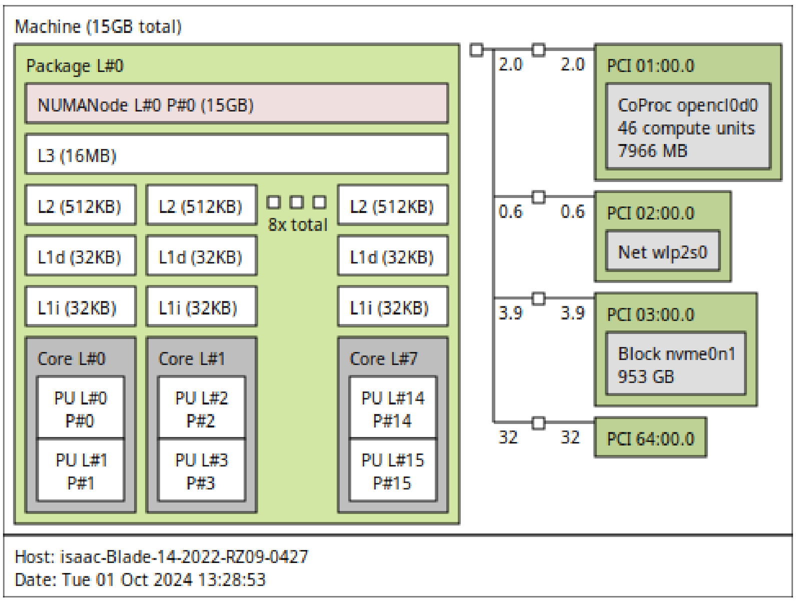

Figure 2: Central Processing Unit (CPU) architecture for the x86-64 AMD Ryzen 9 6900HX.

Cores

Memory

Instruction Set

big picture

ALU

FPL

Cache

Intruction decodeing etc.

Multithreading - easy - Daniel

row selection memcopy barrier

Cache - hard

Profiling - easy - Jack

Profiling was used throughout 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. The non-optimised program was profiled and used in the initial development stage and once the program was at an acceptable level profiling switched to using the optimised code as this was the more effient source. 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 1. The result of profiling using 03 optimised code on the same cube size as before can be found in Table 2.

Table 1: 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 2: 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 1 and 2 show that in both runs the largest time cost is the iteration over the inner slice of the cube. This is expected as this the largest iteration over the nodes in the cube in the program. The next highest is the memcopy that occurs at the end of each iteration. Third is the application of the Von Neuman boundary. In earlier iterations of the program the Von Neuman boundary was called at every inner loop of the main poission iteration. Based on profiling the team was able to identify this as a bottle neck and move the function to its own self contained iteration that only is called once.

One interesting finding of the profiling was how little time is spent at the synchronisation barriers in the code. The team was originally concered that these will cause large delays in the program as different threads took longer to excute. By using profiling this was found to not be the case and thus didn't need to be optimised.

- Python script
- gprof outputs and how they were used

Compiler Optimisation - easy

Individual Topic 1 Jack Duignan - Branch Prediction

- WHat is branch Prediction

Excution on a CPU is optimised using piplineing which allows single clock cycle, single instruction excution. It does this by loading the next set of instructions while the previous instructions are being excuted. When branch instructions occur a control hazard developes 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 excution will go before the conditional branch is excuted. This is branch prediction and is implemented in various ways across CPU architectures.

- Branch prediction errors

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 poission 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 implemention 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.

- Changing of program

The reason our implementation required so many conditional instrcutions is that we use the same nested `for` loop to iterate over both the Von Nueman boundary (the dircilc boundary can be applied once during initilisation) and the inner nodes of the cube. It was hypothosised that moving these out of the same loop and reducing the Von Nueman iteration to only over the outer nodes would reduce the number of condition branch control hazards per iteration and thus overall. This was achived by splitting each iteration into to components first the Von Nueman 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 hazrd from the program.

- Results of changing

The a comparison of the old program with conditional control hazards and without can be seen in Figure 3. This figure shows that the programs excution has been reduced by 30%. 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 Nueman boundary application only iterating over nodes that will need to be applied to as apposed to all nodes in the cube.

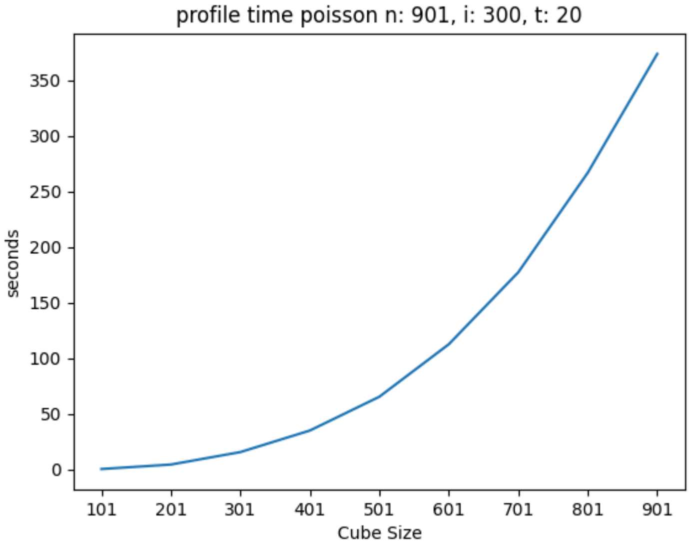

Figure 3: A comparison of the poission solver software excution times with and without conditional branch reduce to reduce control hazards.

Individual Topic 2 Isaac Cone - GPU

Individual Topic 3 Daniel Hawes - SIMD

References