First Individual Report

João Mena
s223186@dtu.dk
Technical University of Denmark
Lyngby, Denmark

1 EXERCISE 1: CACHE PERFORMANCE

This databar exercise was done during the second week of classes in order to explore concepts like cache optimizations and the use of SIMD instructions for performance enhancement in C programs. For this purpose, DTU's HPC was used. This first exercise was divided in the following two tasks:

1.1 Observing Cache Performance

In this first task we were asked to download a C file called *cachetest.c* into the HPC machine and study it's code. The code on the *cachetest.c* file consist of a main *for* loop with secondary loops within it. The main *for* loop starts with a *csize* value corresponding to CACHE_MIN and doubles with each iteration of the cycle up to CACHE_MAX. Inside this *for* loop there is another *for* loop with a *stride* starting with the value 1 and also doubling with each iteration until it reaches the value 128. Nested in these two *for* loops, there are two *do while* loops, the first of which performs, for a maximum time of one second, a series of memory reads and writes, sequentially reading the value of a position of an array with a size equal to CACHE_MAX, for an amount of times proportional to the value of *stride*, and writing a modified value to it, while also calculating the time it took to go through the whole array. This means that the value used in the operation in this loop is very unpredictable since it's always changing, making it hard for the processor to prefetch instructions for the following iterations. The remaining loop also runs the same amount of iterations, however it only adds a variable that is not altered and the index, making it easier for the processor to predict which values will be used in future iterations. At the end of this loop the program also registers the difference in time needed to complete the second loop and the time the previous loop took from start to finish. When analysing this values we can see they are mostly negative, meaning that the second loop was almost always faster than the first.
Analysing the output of this program we can see some accentuated differences between 'read+write' times, indicating that the program has reached a different level of cache with different access times. I was able to identify 3 different jumps, indicating that this machine has 3 levels of cache. The first jump identified was between the strides of 32 and 64 bytes for a cache size of 65536 bytes, indicating the first level of cache probably has 32k bytes. The next jump identified was between the strides of 32 and 64 bytes for a cache size of 524288 bytes, indicating the second level of cache probably has 256k bytes. The last jump found was between the strides of 32 and 64 bytes for a cache size of 33554462. After that, however, there was another big discrepancy, more severe than the ones before, between strides for a cache size of 67108864, which I believe indicates a point where the cache has ended and an access to the main memory is needed.

1.2 Working with SIMD instructions

For this exercise we followed a Berkely lab about SSE intrinsics. The first task we had was to vectorize a piece of code based on a naive implementation. For this, I started by creating a *__m128i sum128* variable and set it to zero, and declare a *__mm128i* reg variable. Using the *_mm_loadu_si128()* function we can load up to 4 integers into a *__mm128i* variable, so I created a *for* loop to load all elements of array *a*, 4 at a time, into *reg*, and add the 128 bit vector value of *reg* to the current 128 bit value of *sum128* using the *_mm_add_epi32()* function. At the end of this loop, *sum128* is a 128 bit value corresponding to the sum of all the other 128 bit *reg* values that corresponded to 4 elements of *a*. To get the 32 bit integer value of the sum of all elements of *a*, I used the *_mm_extract_epi32()* function to break the 128 bit *sum128* variable into chunks of 32 bits and add them to the *sum32* variable.
This method means that essentially for each instruction there is 4 times the amount of data being processed, allowing for a nearly 4 times speedup.

2 EXERCISE 2: PERFORMANCE COUNTERS

For the second databar, we used a lab from Grinnel College.[2] where we used performance counters, namely Linux's *perf* tool, to identify performance bottlenecks and ultimately determine why one version of each program runs faster than the other version presented. The following two subsections describe the different programs analyzed, the results obtained and the conclusions drawn from them.

2.1 Part A

On the first part of this exercise, we were presented with two programs with very similar implementations but very different execution times, version 1 takes around 2,83 times the amount of time that version 2 takes to execute. Looking at the code on both versions of the program and using the performance analyzing tool mentioned above, we can understand the disparity in performance observed. In both programs there is an array which is initially filled with pseudo-random values up to 256, and the end goal is to sum all elements of that array with a value higher or equal to 128. In version 1, the program simply loops through the entire array and compares the value of each element to 128 in order to determine if it's eligible to be added to the sum or not. In version 2, the same happens, but before the comparison loop, the array is sorted in ascending order.
Using the *perf* tool, I analyzed various different metrics, from which

October, 2022, Lyngby, Denmark,

I found the results from the following commands particularly interesting:

```
perf stat -e instructions
```

This command returns the number of instructions executed for a given program. After running this command for both versions of the program we see that, even though it runs quicker, version 2 executes more instructions than version 1. This is due to the added sorting of the array that it does.

```
perf stat -e cycles
```

Using this command we can see how many clock cycles were ran in each program's execution. The amount of cycles measured in v1's execution was about 2,84 times the amount of cycles measured when running v2. This ratio is similar to the difference in execution time, as expected.

```
perf stat -e branch-misses a-v1
```

This is the command that I found to be the most telling. The number of branch misses measured in v1 amounts to 24,23% of all branches, while on v2 it amounts to only 0,03%. The sorting of the array in v2 appears to be the key factor that allows it to execute in such a shorter amount of time than v1. Since the main loops in both programs contain a conditional statement that compares the value of each element to a constant value, having the elements of the array sorted allows the processor to make more accurate predictions about the result of said conditional statements in the program, severely reducing the amount of time wasted in branch misses and ultimately allowing for v2 to display a much better performance than v1.
In conclusion, the root cause for the discrepancy in performance in this program is the sorting of the array before the loop with the conditional statements regarding value comparisons.

2.2 Part B

For the second part of this exercise we had another program, once again split into two different version with different performances. In this case, version 1 takes around 1,62 times the amount of time that version 2 takes to run.
In both versions, the program starts with a loop to fill an array with integers. After that there is a second loop to calculate the sum of all elements in the array. Where the two versions diverge is that in version 1, the index used to access the next element of the array is read from the current element, while in version 2 there is a simple addition of 1 to the current index, meaning that in version 1 there is one more memory access in every iteration of the loop than in version 2. To better understand what this meant in terms of impact to the performance of the program, I used once again the Linux *perf* tool and analyzed the returns of some commands, from which I highlighted the following:

```
perf stat -e cycles
```

Using the *cycles* command I observed that, as I had already mentioned for version A, the discrepancy in cycles is very close to the execution time difference. In this case, the cycles measured for version 1 were around 1,61 times the amount of cycles measured in version 2.

```
perf stat -e instructions
```

Looking at the output of this command I noticed that version 1 also had around 43% more instructions ran than version 2. The added memory accesses imply *LOAD* instructions involved on every cycle of the second loop followed by an *ADD* instruction just to update the index, while version 2 only needs to perform an *ADD* instruction to do an equivalent operation.
As I reflected a bit more about this topic, I realized that *LOAD* operations can cause dependencies, which can lead to pipeline stalls.

```
perf stat -e cycle_activity.cycles_no_execute
```

Using the command above, we can verify that there is also a significant difference in execution stalls (about 2,7 times more stalls in version 1), which also account for a slower execution time in v1.

```
perf stat -e cache-misses
```

Another important performance counter to check is the amount of cache misses, since the main difference between both version is the amount of memory accesses. Running this command a few times for each version it was hard to take definitive conclusions because the results fluctuated a lot, but it was clear that on average version 1 had a higher rate of cache misses than version 2.
In conclusion, the root cause for the discrepancy in performance in this program is the use of values read from elements of the array to update the index.

3 EXERCISE 3: THINKING PARALLEL

For this exercise we were given a C code file and asked to analyze the code and identify where parallelization can be applied to optimize the program and which challenges can limit the performance, among some other questions.

The first opportunity for parallelization that I found in the given code was the first *for* loop:

```
for(long i = 1; i < INTERVALS; i++)
    from[i] = 0.0;
```

Since the macro *INTERVALS* is a very high number, this loop will run a lot of cycles, which means we can use parallelization to divide the iterations between processors to make it faster.
Parallelizing this loop could raise some issues since it requires at least 3 accesses to memory in each iteration, for 3 different elements of an array. Having this loop running in parallel would mean that different processors would be performing various accesses to memory at the same time, which could prove to be a bottleneck in performance.

The next *for* loop is another opportunity for parallelization:

```
for(long i = 1; i < (INTERVALS - 1); i++)
     to[i] = from[i] + 0.1*(from[i - 1] - 2*from[i]
             + from[i + 1]);
```

There is yet another *for* loop that could be parallelized:

```
for(long i = 2; i < 30; i += 2)
    printf("Interval %ld: %f\n", i, to[i]);
```

In this case, however, the amount of iterations in the loop is so small that the overhead that an API used for parallel programming would add to the program would cancel any performance improvements. Using different variables that take the value returned by the function *clock()* and subtracting them to measure the amount of time each part of the code makes up for, I observed that the first loop only amounts to around 1,8% of the total execution time, however for the second *for* loop, the sum of all the times it runs inside the *while* loop makes up to around 97,5% of the execution time. This means that optimally, paralellizing this section of the code through N processors, with a local speedup of N, could lead to a overall speedup of $\frac{1}{(1-0.975)+(0.975/N)}$, according to Ahmdahl's Law. This value is only an optimistic theoretical estimation, since in reality the overhead associated with an API like, for example, OpenMP[1], would lower the real speedup value.

REFERENCES

[1] OpenMP Architecture Review Board. 2020. OpenMP. https://www.openmp.org/.
[2] Jerod Weinman Charlie Curtsinger, Janet Davis. 2015. Lab 11: Hardware Performance Counters. https://curtsinger.cs.grinnell.edu/teaching/2016F/CSC211/labs/lab11.html.