Universiteit Leiden

Universiteit Leiden

Computer Architectuur

4032CMPA6

Assignment 3 — Optimizing Single-Thread Performance

Authors:
Seyed Saqlain Zeidi s2982048
Yvo Hu s2962802

December 19, 2021

1 Introduction

In this document we analyze the effectiveness of caching
and the reduction of pipeline stalls by applying instruction
scheduling. The purpose is to reduce the execution time so
programs run faster. We do perform three experiments:
CPU caching, Instruction scheduling for a static in-order
pipeline and finally The effectiveness of loop unrolling
for a dynamically scheduled pipeline.
We have used the following CPU when performing these experiments:
Intel(R) Core(TM) i7-9700 CPU clocked at 3.00GHz with an
x86_64 architecture. This CPU has 32 KB L1 data cache,
256 KB L2 cache and 12 MB L3 cache shared with all of the cores.
We used the gcc 7.5 compiler when compiling the C/c++ code.

Contents

2 Task 1

2.0.1 Task 1.1

This loop is responsible for the matrix multiplication in task 1.1

```
for (size_t i = 0; i < N; i += block)
    for (size_t j = 0; j < N; j += block)
      {
        for(int blockRow = i; blockRow < i + block; blockRow++){
          for(int blockCol = j; blockCol < j + block; blockCol++){

            float dotproduct = 0.0;

            for (size_t k = 0; k < N; k++)
              dotproduct += A[blockRow][k] * B[blockCol][k];
            C[blockRow][blockCol] = dotproduct;
          }
        }
      }
```

In addition to loop blocking, we have also transposed matrix B so we can make full use of locality of reference so our results can solely focus on the cache miss rate improvements caused by loop blocking.

Figure 1 illustrates the cache miss percentage for D1 cache over multiple matrices, each with a different size, and with different block sizes. The block sizes are presented with different colours. The simulation is done in cachegrind. The cache miss rate seems to dip when the matrix size multiplied by the block size equals 4096. This might be the size of the cache and would explain why the miss rate increases drastically after the dip. On the other hand, the higher cache miss rate preluding this is likely caused because of the lack of loop blocking. The results were performed once for cachegrind, but with perf, the smaller matrices were done in orders of magnitude more than the bigger matrices to dominate the results.

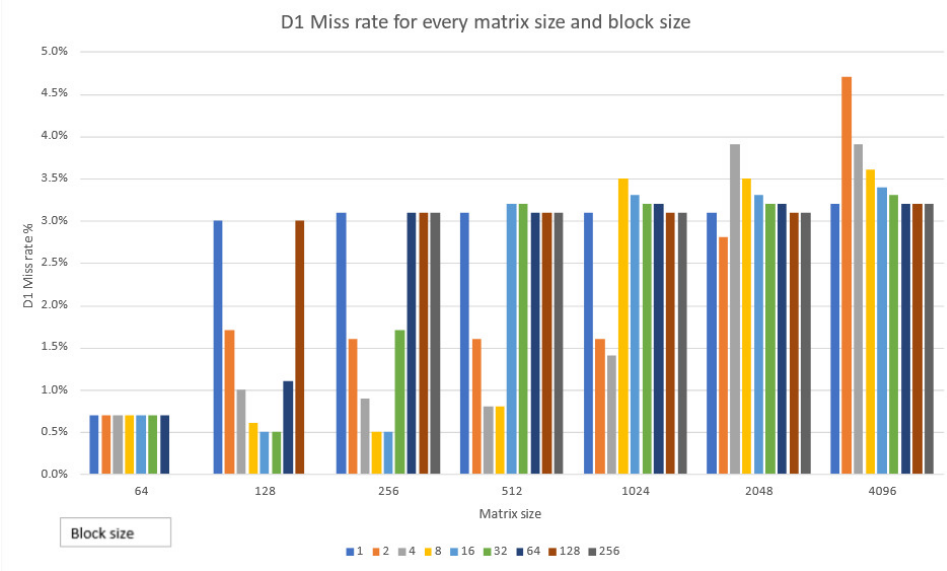

Figure 1:
D1 cache miss rate, shorter bars are better.
X-axis: matrix size
Legend: block size
Note: Some entries are N/A because the block size is greater than a side of the matrix

Figure 2 shows the L3 miss rate when the matrix size is bigger than the L3 cache. In the smaller cases this number was 0% for smaller matrices. As alluded to in the previous paragraph, 4096 holds a special significance because the three matrices in the program do not fit in the L3 cache, and therefore causes cachegrind to report cache misses in the L3 cache and in turn implies that the d1 caches are also going to report more cache misses because it did not fit in the overarching cache.

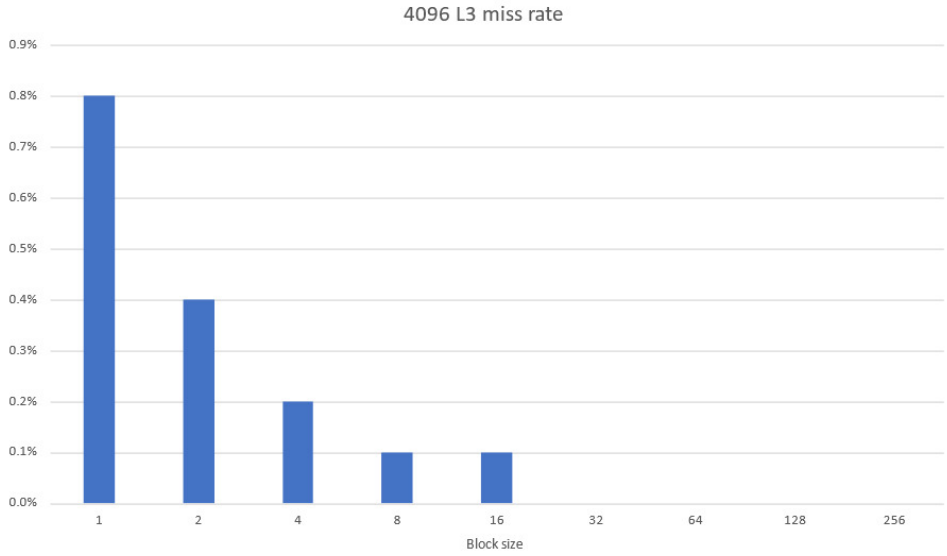

Figure 2: L3 cache miss rate

We ran the program again and calculated the cache misses but this time using perf. The results are shown in figure 3.

A huge decrease in cache misses is reported between matrix size 128 and matrix size 1024. This suggests a lack of loop blocking under matrix size 128, and not enough cache size above matrix size 1024. The block size has small, but noticeable effect on the cache miss rate, as one can see by analysing the slope in matrix sizes 1024 to 4096. The greater the block size, the greater the decrease as more direct cpu-cache operations occur instead of cpu-memory operations because more directly usable data is stored in the cache lines.

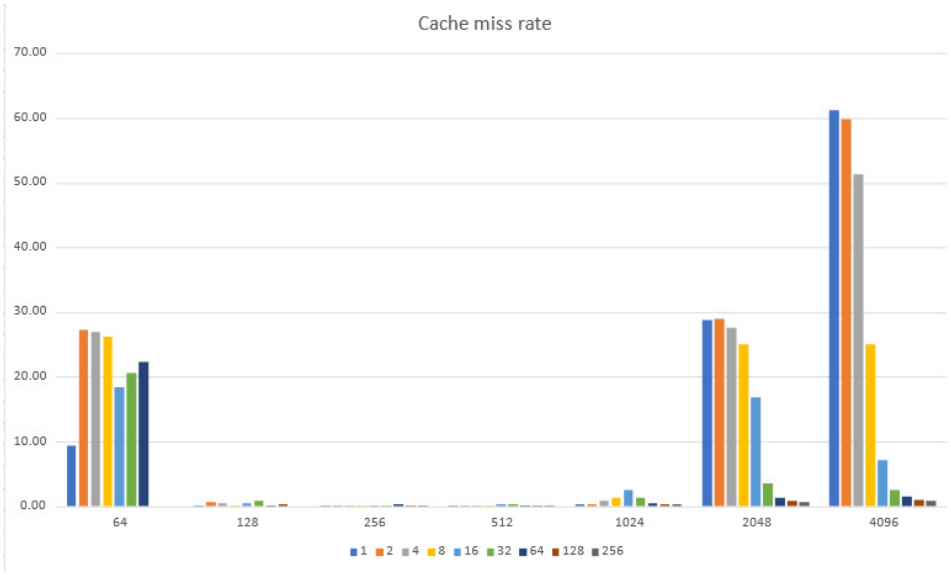

Figure 3:
Cache miss percentage in D1 cache.
X-Axis: Matrix size
Legend: Block size
Note: Some entries are N/A because the block size is greater than the side of a matrix

2.0.2 Task 1.2

We have switched two loops with one another to implement loop interchange. The great reduction in calculation time as illustrated in figure 5, can be attributed to a greater usage of locality of reference. The data for the x component will most likely already be in cache, which greatly reduces the amount of time needed to access that data. If instead the loops were reversed, the cache lines would be filled with unnecessary data for the y component, and thus cannot be used and we have to constantly fetch catch lines again and wasting a large part of the space in the cache lines for unnecessary data.

```
void
op_grayscale2(image_t *dst, const image_t *src)
{
  for (int y = 0; y < dst->height; y++)
    {
    for (int x = 0; x < dst->width; x++)
        {
          rgba_t color, gray;
          RGBA_unpack(color, *image_get_pixel(src, x, y));
          float intensity = compute_intensity(color);
          RGBA(gray, intensity, intensity, intensity, 1.f);
          RGBA_pack(*image_get_pixel(dst, x, y), gray);
        }
    }
```

The D1 miss rate is consistently reduced by a significant amount after loop interchange as is illustrated in figure 4.

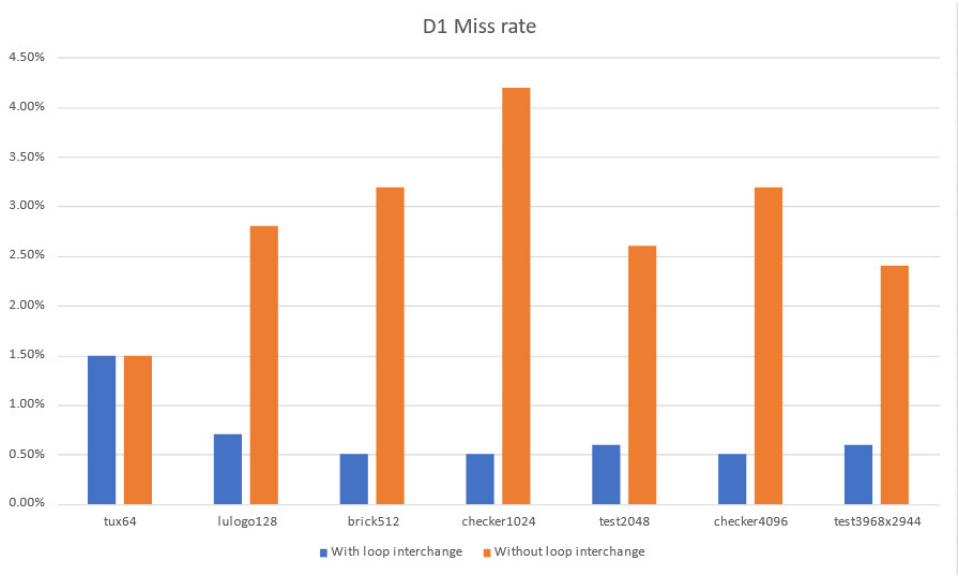

Figure 4: Difference in miss rate corresponds with loop interchange efficiency

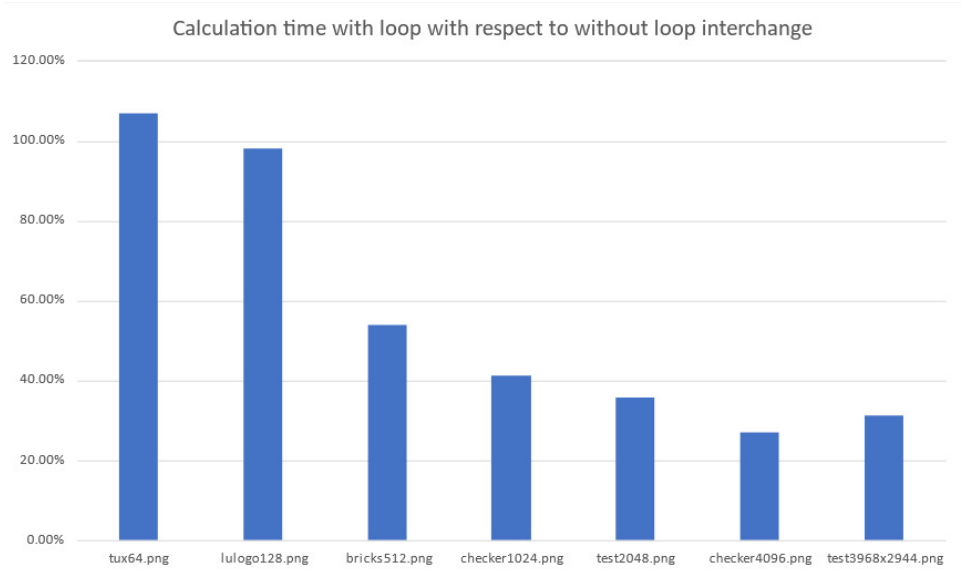

Figure 5: Loop interchange efficiency increases with the size of the images or loops within the program

2.0.3 Task 1.3

We did not find any statistical significant distinctions between block sizes, so we have set the block size as the size of the tile.

The shape of the blocks however, are set to be rectangular. This is because a tile, which is stored in memory can be placed consecutively in the memory addresses horizontally next to the first tile that are stored in cache. This allows us to omit having to keep fetching the memory addresses from memory to decide where to place the tile, but we can use what is already in the cache to determine where to place out next tile.

There is a consistent 2% reduction in calculation time after loop blocking which is shown in figure 6.

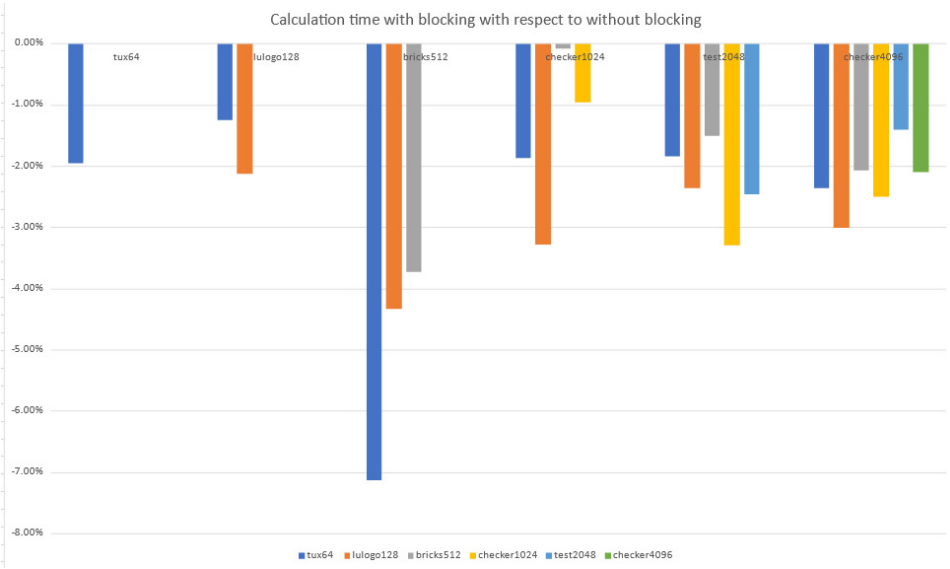

Figure 6:

X axis is background image
Legend is the tile image
Block size = tile size
Note: Some entries are N/A because the tile image is bigger than the background image

2.0.4 Task 1.4

We did not find any statistical significant distinctions between block sizes, so we have set the block size as the size of the tile.

Loop blocking actually made the calculation time longer for the smaller matrices up until matrix size 512 which got a slight improvement on its calculation speed as is shown in figure 7.

This can be attributed to the slight overhead which the loop blocking causes, and which is exacerbated by the already small calculation times with the small matrices. This slight overhead however, proves to be very useful in the long run with an ever decreasing calculation time for bigger matrices.

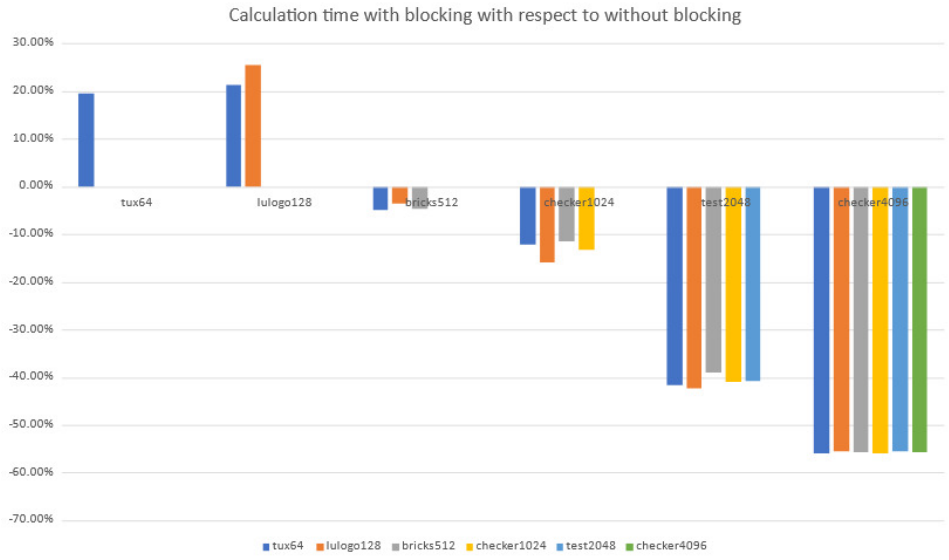

Figure 7:
Note: Some entries are N/A because the tile image is bigger than the background image

3 Task 2

We have not done this task.

4 Task 3

Figure 8 Shows the runtime of grayscale.c when run 10.000 times with 5 different values for N Unroll. 0, 2, 4, 6 and 8. The expected result was a decrease in runtime, which occured in N Unroll = 6, but only for one second difference. Perhaps there would have been more reliable data if the amount of runs was higher, but we did not try it.

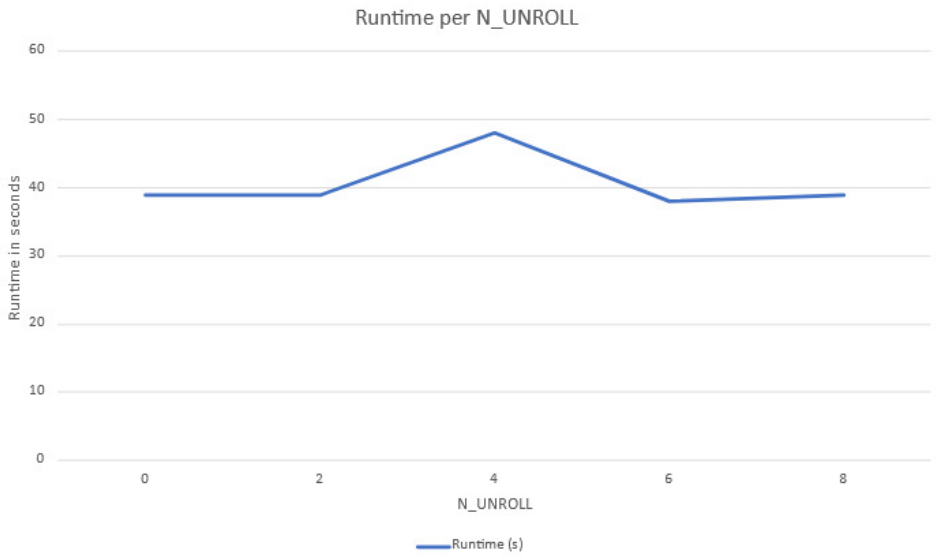

Figure 8: Runtime in seconds when running the program 10.000 times per N UNROLL

5 Appendix

With this report we have included an additional excel sheet with all the data points, the code to the programs, and bash scripts to run the programs.

We have omitted the standard deviation from the graphs because it was not applicable in most circumstances. All graphs contain separate elements, those of which were either singular data points with their own category, or were themselves the difference between aggregates (because those are easier to comprehend than 2 separate aggregates). However, these standard deviations and all other datapoints for the graphs can be found in the excel sheet.

In the excel sheet:
Sheet 1 - 4 correspond to task 1

In the files:
taskn_n.sh shows the script used for each assignment.

valrunn_n.sh shows valgrind related scripts for taskn_n.sh

6 Collaboration

We worked together on the assignment while calling on discord. Yvo Hu did task 1.2, 1.3 and 1.4, Seyed Saqlain Zeidi did task 1.1 and 3. Each member wrote about their own tasks in the report. The collaboration was good, the contribution was fair and both team members should receive the same grade for this assignment.