LINFO2241 - Project 3 - Group 20 - 2024

Vilhelm Persson
vilhelm.persson@student.uclouvain.be — 1212400

Ji-liang Leung
ji-liang.leungwing@student.uclouvain.be — 79412000

I. Introduction

In order to make the server faster, different optimization techniques will be tested and evaluated. First, scripts were made in order to test different packages and parse the output. Then, different optimizations were implemented, more about that in the next section.

A. Setup

To begin with, a script was made in *Python* that ran the server along with performance tests of combinations of different parameters using *wrk* and *perf*. The results were then saved to a *csv* file, more about these parameters in the second part of this introduction. A second *Python* script was also made in order to make plots from the data retrieved from the test-script using *matplotlib*, *pandas* and *numpy*. The tests were run on a single computer with charging cable attached and no unnecessary programs running in the background.

B. Measurements taken

Furthermore, to evaluate how well the different optimizations performed, measurements were taken from four different test cases. Small and big matrices, small and big patterns, small and large amount of patterns as well as different worker counts. Every test had a duration of 30 seconds, with 100 connections using a single thread.

1) Matrix size test:

- The matrix size as [64, 512].
- The pattern size as [4].
- The pattern count as [1].

2) Pattern size test:

- The matrix size as [64].
- The pattern size as [32, 128].
- The pattern count as [16].

3) Pattern count test:

- The matrix size as [64].
- The pattern size as [32].
- The pattern count as [8, 128].

4) Worker count test:

- The matrix size as [512].
- The pattern size as [8].
- The pattern count as [1].
- The worker count as [1-10].

C. Measurements retrieved

The measurements of the results retrieved were:

- Amount of cache-misses.
- Amount of cache-references.
- Data transfer per second.

II. Different optimizations

In this section of the report, the various optimizations implemented for the server will be described and discussed. Additionally, any available improvements in the % of the performance on the *Score* task will be presented in the section title. Note that some optimizations were tested together, so the percentage improvement is not specific to a single optimization.

Initially, *perf* was used to analyze the program and identify the functions that were most critical for optimization and responsible for performance bottlenecks. Unsurprisingly, the primary culprits were functions with multiple nested loops and those performing more computationally intensive mathematical operations, as discussed earlier.

We concluded that optimizing `parse_request` and `res_to_string` was unnecessary, as these functions neither involve computationally intensive operations nor contain deeply nested loops, so we focused our attention on the functions `multiply_matrix` and `test_patterns`.

A. ***Cache awareness*** *100% $\hookrightarrow$ 204%*

The data in a computer is stored in cache blocks and therefore fetched as blocks for CPU. One solution to speed up the program could be to align the data read by the program to the cache blocks. This was done in the matrix multiplication part of the program by changing the order of which values were multiplied. This improvement will lead to fewer cache misses and will in turn improve performance.
This difference is evident in the matrix multiplication code, where swapping certain loop indices demonstrates the impact of cache-aware versus non-cache-aware code.

B. ***Loop unrolling*** *100% $\hookrightarrow$ 204%*

In machine code, a loop is a branch condition followed by the instructions that are supposed to be repeated and ending with a jump statement back to the branch condition. A CPU uses pipelining to always have decoded instructions to execute. This pipelining can only be done if the program knows the following instructions, which is not always the case. Pipelining can be used more optimally if you unroll the loops so that the program needs to jump and run the branch conditions fewer times, resulting in improved prefetching. This will in theory improve the performance of the program.

Pseudocode

```
// K - size of computation
// A - object to compute
for k = 0 to K - 4 step 4 do
    process(A[k])
    process(A[k + 1])
    process(A[k + 2])
    process(A[k + 3])
end for
for k = k to K - 1 do
    process(A[k])
end for
```

The code includes two versions of loop unrolling: one that is cache-aware and another that is not.

C. Combination of the previous techniques and supplementary optimizations

*1) **Stack allocation**:* In the server implementation, we opted to allocate the request on the stack instead of the heap, under the assumption that stack allocation would be faster. However, we did not observe any significant performance improvements. To achieve this, we used the standard library function `alloca` as opposed to the ngx `malloc` wrapper. If the matrix size is too high, a heap allocation is done instead to avoid potential stack overflows.

*2) **Blocked matrix** 204% $\hookrightarrow$ 260%:* The code utilizes blocking to enhance the performance of matrix multiplication by improving cache locality. The matrix is partitioned into smaller submatrices of size `BLOCK_SIZE` which are processed one at a time. This approach is effective because modern processors have limited cache capacity, and by working on smaller chunks of data that fit within the cache, the number of cache misses is reduced. By ensuring that a block of the matrix remains in cache during computation, memory accesses are localized, leading to faster data retrieval and more efficient use of the cache, effectively minimizing the time spent fetching data from slower memory and thus best with larger matrix sizes (bigger than the caches available space).

Pseudocode

```
for iBlock = 0 to K-1 step BLOCK_SIZE
 for jBlock = 0 to K-1 step BLOCK_SIZE
  for kBlock = 0 to K-1 step BLOCK_SIZE
   for i = iBlock to min(iBlock + BLOCK_SIZE,
       K)
    for j = jBlock to min(jBlock + BLOCK_SIZE,
        K)
     for k = kBlock to min(kBlock + BLOCK_SIZE
         , K)
      result[i, j] += m1[i, k] * m2[k, j]
```

*3) **Manual prefetching** 204% $\hookrightarrow$ 260%:* If it is known that certain data will be accessed shortly, the GCC intrinsic `__builtin_prefetch` can be utilized to load the data into the cache in advance. This approach helps minimize latency during data access and is particularly effective when memory access patterns are predictable. It is mostly used in the nested loops of the project.

*4) **Multi-threading** 260% $\hookrightarrow$ 1274%:* The matrix multiplication is optimized through multi-threading. The computation is divided into multiple threads, each handling a subset of rows in the result matrix. The number of threads is determined by `num_threads = min(8, K)` to limit the thread count, where `K` is the matrix size. If the number of rows are not divisible by the number of threads, we dispatch one of each remaining row to a different thread.

The worker threads have an implementation of matrix multiplication for a specified amount of rows.

Finally, the threads are synchronized using `pthread_join` to ensure all computations are completed before accessing the result matrix.

Initially, the score with basic multi-threading and no optimization in the threaded function lead to a score of 760% which was later improved to 960% by adding all the aforementioned optimizations.

After noticing the boost in performance given by multi-threading for `multiply_matrix`, we implemented a similar approach for `test_patterns` and attained a final score of 1274%.

Note that the best implementation without multi-threading can be ran by disabling the multi-threading definition in *utils.c*.

Pseudocode

```
num_threads = min(8, K)  // arbitrary bound
thread_data[num_threads] {
    matrix1, matrix2, result, K, start_row,
        end_row
};

for t = 0 to num_threads - 1 do
    thread_data[t].matrix1 = matrix1
    thread_data[t].matrix2 = matrix2
    thread_data[t].result = result
    thread_data[t].K = K
    thread_data[t].start_row = t *
        rows_per_thread
    thread_data[t].end_row = if t ==
        num_threads - 1 then K else (t + 1) *
        rows_per_thread

    create thread t to run
        multiply_matrix_thread with
        thread_data[t]
end for

for t = 0 to num_threads - 1 do
    wait for thread t to finish
end for
```

III. Results

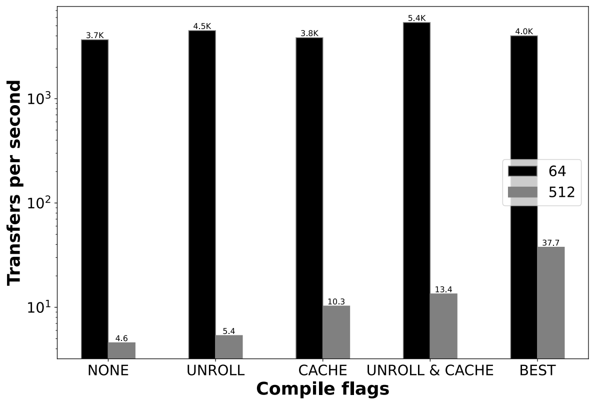

Fig. 1: Matrix size comparison

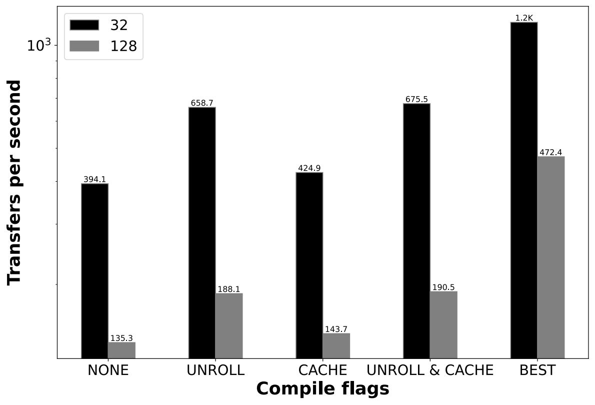

Fig. 2: Pattern size comparison

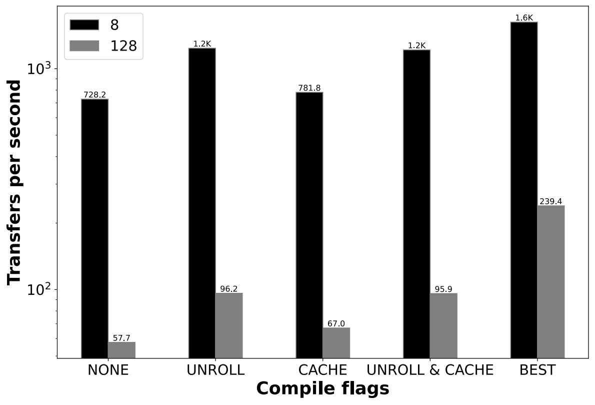

Fig. 3: Pattern count comparison

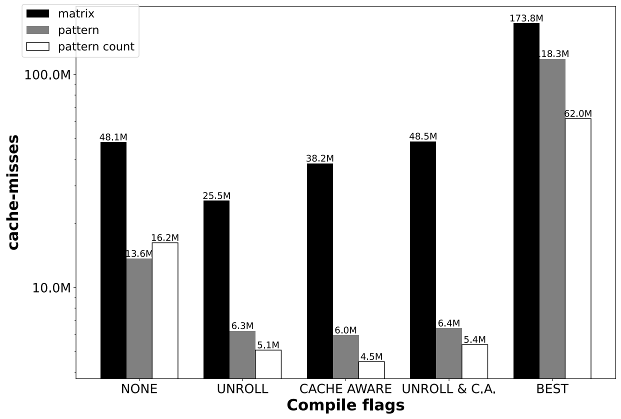

Fig. 4: Cache misses

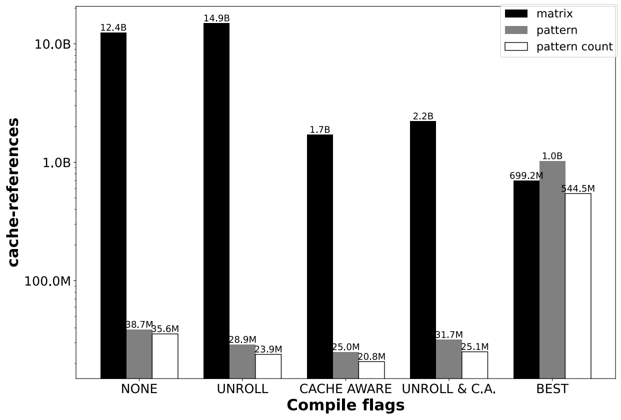

Fig. 5: Cache references

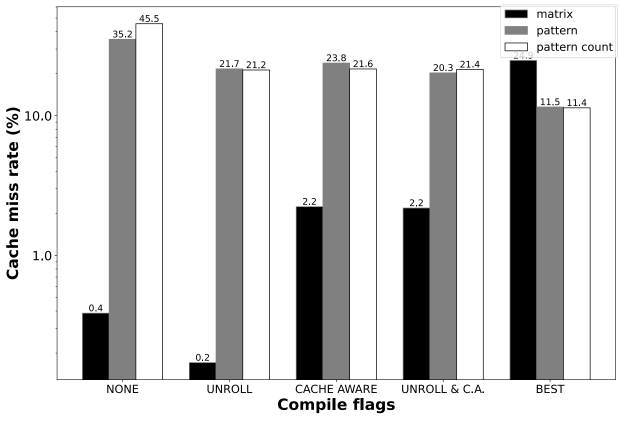

Fig. 6: Cache references

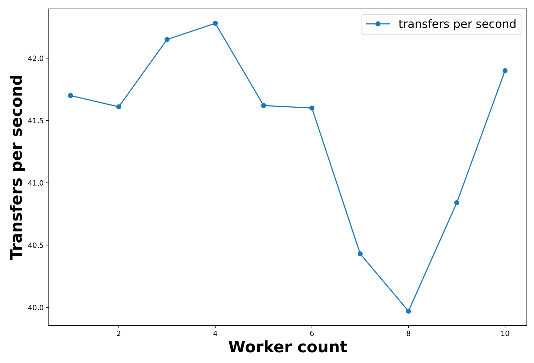

Fig. 7: Worker throughput

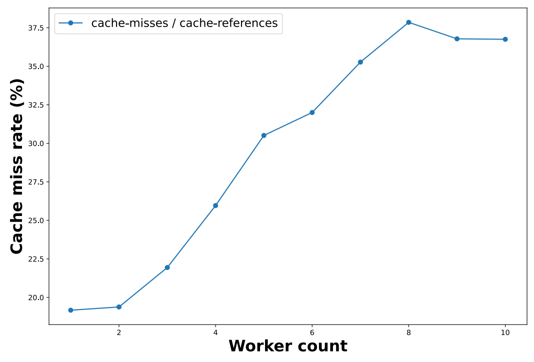

Fig. 8: Worker count

IV. Discussion

As can be viewed from the results above, the biggest difference in performance is when the load on the program is bigger with bigger matrix sizes, pattern sizes and larger number of patterns. This is the case because the optimizations allow for higher throughput despite higher workload.

A. Loop unrolling

The optimization of loop unrolling seems to make the most improvements for the pattern size and pattern size tests as seen in fig. 2 and fig. 3. This could probably be the case because there are more loops around the function `test_patterns()` which could benefit heavily from this. The pattern size and number of patterns tests would affect this function more since those parameters will have an influence on how much the program would need to loop.

B. Cache awareness

As opposed to loop unrolling, cache awareness improves most on the matrix size test, seen in fig. 1, and less on the pattern size and number of patterns tests, viewed in fig. 2 and fig. 3. The function that this optimization would increase performance the most is `multiply_matrix()`. By reading the results, one can be confused why the amount of cache-misses is higher despite having more throughput. The reason could be that since the throughput is higher, more requests are made which could increase the amount of cache-misses. However, this improvement does not affect the rest of the program which is why cache awareness is as slow as the original implementation when performing the tests of pattern size and number of patterns.

C. Loop unrolling combined with cache awareness

By reading the results, one can spot that the combination of the optimizations of loop unrolling with cache awareness is the highest bar of either of the optimizations in the cases. The reason for this could be that these test cases have different bottlenecks of the program. For the matrix size test it seems to be for `multiply_matrix()` and for the pattern size and number of patterns tests it is `test_patterns()`. This is probably why the combination of these optimizations will be as fast as the quickest one.

D. Number of workers

The graph in fig. 7 presents an interesting anomaly. We suspect that the multi-threading implementation in our program may be interfering with Nginx's multi-threading, potentially leading to performance degradation. This hypothesis aligns with the observation that throughput decreases as the number of threads approaches the machine's core count.

Similarly, fig. 8 illustrates an almost linear increase in the cache miss rate. This trend supports the above explanation: the contention between threads could be reducing the efficiency of the caching mechanism. It is reasonable to hypothesize that a single-threaded implementation might have yielded better performance as the thread count increased, highlighting the trade-offs introduced by multi-threading in this context, we can deduce that we should avoid multi-threading both nginx and our program.

E. Combination of the previous techniques and supplementary optimizations

Furthermore, the highest increase in performance comes from the own implemented optimization. One reason for this could be that multi-threading highly increases performance because the program can run some parts in parallel which would outpace a sequential program. As seen from the multi-threading description of the report, the highest increase of the score was trough multi-threading. The only category where the own implemented optimization was not quickest is in case 1 with smaller matrix size. In that case, the throughput is high on all the different optimizations. The reason for this is probably that the matrix sizes are so small that it is not the bottleneck for the program.

At first, the results of fig. 7 might seem a bit puzzling. However, when we factor in the cache miss rate per worker count, things start to make more sense. As concurrency increases, the cache miss rate also rises. This happens because the more threads accessing the cache simultaneously, the more likely it is for those accesses to conflict or miss.

It's also worth noting that the best-performing version handles more requests overall compared to the others. With this in mind, we can see why its miss rate is higher: more concurrency naturally causes more misses (fig. 4). At the same time, we observe fewer cache accesses **per request**, which might seem surprising at first but makes sense when we look at fig. 6 in the broader context.

V. Conclusion

The best optimization in this report is the own implemented one. The reason for this is mainly because of multi-threading. Cache awareness made visible improvements for the matrix size test, where the function `multiply_matrix()` is the bottleneck of the program. On the other hand, loop unrolling made improvements on the pattern size and number of patterns tests where `test_patterns()` is the bottleneck of the program.

Group Work Evaluation

Our group collaboration went very well, and we successfully achieved our goals through effective teamwork. Here are the key points:

- **Communication:** We maintained clear and open communication.
- **Task Distribution:** Tasks were assigned based on individual strengths.
- **Meetings:** We prioritized in-person meetings and minimized working individually to ensure both of us stayed in touch and updated.
- **Decision Making:** We consulted each other before making major changes to ensure mutual agreement and maintain consistency in our approach.

In summary, the group work was a positive experience that not only demonstrated the value of teamwork and provided valuable lessons for future projects, but also offered the unique opportunity for collaboration between two students from different countries.