Compilation & Performance Project Report

Roussel Desmond Nzoyem

Abstract— **The purpose of this study is to optimize a kernel using OpenMP and measure our improvements through `Perf`. The kernel itself is a `C++` matrix-matrix product, important in all areas of computational science. In total, five optimization techniques were implemented. These include: memory access improvement, code parallelization, code vectorization, switching to 32 bits floating point numbers, and using `BLAS`. The code for each technique was written in the `VSCode` editor, then compiled and run on a Linux operating system using `CMake` and the `GCC` suite. The main device used for tests was the Atlas computing cluster, due to the fact that it can run `Perf` with most performance counters available. It was found that each of the aforementioned optimization techniques is an improvement on the non-optimized code, especially the latter ones. Additionally, a multi-threading benchmark was carried out on the parallelized and vectorized code; it was determined that those two optimizations can become inefficient when too many threads are used.**

I. Materials

A non-optimized code (kernel) was provided to us as a reference for this project. Using that code, we will write a series of optimizations and test them using the `perf` profiling tool. Due to difficulties running this tool locally, most of our tests will be run on the `Atlas` computing cluster.

I-A. The non-optimized C++ code

The provided code performs a matrix-matrix product $C = AB^T$; where A, B, and C are arrays of size N^2. C is initialized at 0.

```
for(long int k = 0 ; k < N ; ++k){
    for(long int j = 0 ; j < N ; ++j){
        for(long int i = 0 ; i < N ; ++i)
            C[i*N+j] += A[i*N+k] * B[j*N+k];
    }
}
```

Listing 1: Non-optimized C++ code

The resulting binary (called `matrix`) has three execution modes:

- `-check`: to check the result.
- `-no-check`: to avoid checking the result.
- `-no-optim`: to avoid running the optimized code.

I-B. Atlas

`Atlas` is a powerful computing cluster with cutting-edge computing nodes. However, we will only connect to its frontal node and access the 64 cores available, with 512 GB of RAM. Further documentation on `Atlas` can be found at [1].

I-C. Perf

`perf` is a lightweight Linux profiling tool with performance counters. We are mostly interested in the following indicators:

- the execution time (`elapsed time`) in seconds, for a single execution (not an average).
- the number of cache-miss events (`cache-misses`) expressed as a percentage over all cache references.
- the number of instructions per cycle (`insns per cycle`).

Our processes will be bound to their cores, meaning that to get all our values of interest for the non-optimized version of the code, we could simply run the command:

```
$ OMP_PROC_BIND=TRUE perf stat -B -e cache-
    references,cache-misses,cycles,instructions
     matrix -no-optim
```

In addition, we will use

```
$ perf record -e cache-misses matrix -no-optim
```

to record and analyze cache-misses, hence detecting bottlenecks in the code.

II. Results

II-A. Memory access improvement

Reordering the loops yields the following code (the resulting optimized matrix C_{optim} was carefully initialized at 0).

```
for(long int i = 0 ; i < N ; ++i){
    for(long int j = 0 ; j < N ; ++j){
        for(long int k = 0 ; k < N ; ++k)
            COptim[i*N+j] += A[i*N+k] * B[j*N+k];
    }
}
```

Listing 2: Memory access optimized code

As a result, we get the table below. We can see that

	Non-optimized	Optimized
elapsed time	43,42	1,97
insns per cycle	0,05	0,83
cache-misses	0,022 %	0,013 %

TABLE I: Memory access optimization for $N = 1024$

the execution time and the percentage of cache-misses is greatly decreased, along with the number of instructions per cycle that increases. The next section attempts to add to this optimization by parallelizing the loops.

II-B. Code parallelization

In this optimization, we need to add a statement like `OMP_NUM_THREADS=64` to specify the number of parallelization threads to use. Remembering that our matrices are stored as 1-dimensional arrays, we should only focus on one loop. Let's parallelize the loop on C_{optim}'s rows (indexed by i). The code will look like this:

```
#pragma omp parallel for collapse(1)
for(long int i = 0 ; i < N ; ++i){
    for(long int j = 0 ; j < N ; ++j){
        for(long int k = 0 ; k < N ; ++k)
            COptim[i*N+j] += A[i*N+k] * B[j*N+k];
    }
}
```

Listing 3: OpenMP parallelization for a single loop

And the resulting measures are presented in the table below. On their own, the results are convincing. When

	Non-optimized	Optimized
elapsed time	43,42	0,36
insns per cycle	0,05	0,18
cache-misses	0,022 %	0,039 %

TABLE II: OpenMP parallelization for a single loop, for $N = 1024$

compared to the previous optimization, we notice a poorer cache-miss ratio, and a worse number of instructions per cycle, even though the execution time has been considerably decreased.

II-C. Code vectorization

Let's vectorize the inner loop, the one in k. Using OpenMP's SIMD program, we replace the arrays A and B by pointers $ptr1$ and $ptr2$ on their respective first elements. We will use the fact that these pointers are 64-bytes aligned, and set `simdlen` (the preferred number of iterations to be executed concurrently) to 64. The computation loop looks like the following:

```
#pragma omp parallel for collapse(1)
for(long int i = 0 ; i < N ; ++i){
    for(long int j = 0 ; j < N ; ++j){
        double sum = 0;
        #pragma omp simd reduction(+: sum) aligned(
            ptr1, ptr2: 64) safelen(N) simdlen(64)
        for(long int k = 0 ; k < N ; ++k)
            sum += *(ptr1+i*N+k) * *(ptr2+j*N+k);
        COptim[i*N+j] = sum;
    }
}
```

Listing 4: OpenMP inner loop vectorization

The resulting comparison yields the table below. This is good, but it doesn't bring any performance improvement when compared to the best optimization so far. On the contrary, the cache-misses and instructions per cycle gets poorer. It seems the code has already been optimally improved. This can be explained by

	Non-optimized	Optimized
elapsed time	43,42	0,36
insns per cycle	0,05	0,12
cache-misses	0,022 %	0,082 %

TABLE III: OpenMP vectorization, for $N = 1024$

the size limit on the caches. At some point, asking 64 instructions to be executed concurrently requires that most of the values be loaded out of the cache to make room for new ones.

Using `perf`'s `record` and `annotate` features, we can confirm the location of the bottlenecks in the images below.

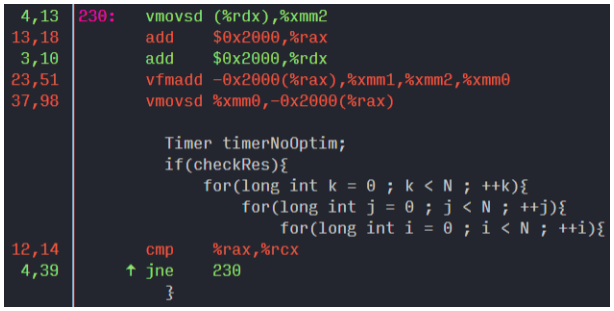

Fig. 1: Location of the bottleneck in the non-optimized version. The majority of cache-misses is located in the outer loops in i and j.

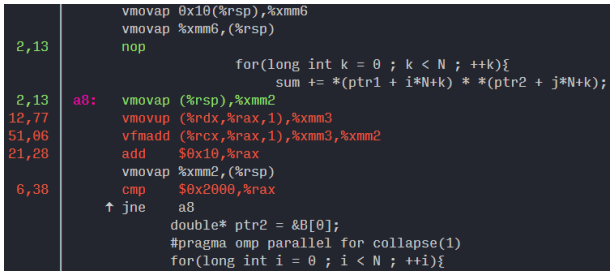

Fig. 2: Location of the bottleneck in the optimized version. The majority of cache-misses is located in the inner loop in k.

II-D. Other improvements

II-D.1. Using double vs. float

In order to easily achieve this comparison, we simply include the instruction `typedef float decimal;`, and declare all variable of interest as `decimal`. This way, we can easily switch between `double` and `float` for benchmarking. The results are presented in the table below. As expected, the floats lead to better performance, because of the lesser precision. In fact, in modern architectures, twice as many registers can be used for floats (32 bits) compared to doubles (64 bits). This in turn is effective for parallelization and especially for vectorization.

	double	float
elapsed time	0,53	0,19
insns per cycle	0,13	0,34
cache-misses	0,078 %	0,110 %

TABLE IV: float vs. double comparison, for $N = 1024$

II-D.2. Using OpenBlas

BLAS is a common HPC library that we will use to test our implementation. Due to permission issues while using `OpenBlas` on Linux with CMake, we adopted a straightforward installation option `sudo apt-get install libopenblas-dev`, then a compilation of the code with `g++ tests/matrix.cpp -I src -lblas`. In the code, we can compute a matrix-matrix (transposed) product using the following instruction :

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans
    , N, N, N, 1.0, A, N, B, N, 0.0, COptim, N);
```

Listing 5: BLAS implementation

The table below clearly indicates better optimization, in every indicator, by OpenBlas. This library earns its reputation as one of the best HPC libraries available nowadays.

	With OpenMP	With OpenBlas
elapsed time	0,36	0,135
insns per cycle	0,12	1,60
cache-misses	0,082	0,399 %

TABLE V: Our optimization (OpenMP) and OpenBlas comparison for $N = 1024$

II-D.3. Small, medium and large matrices

This final part of the benchmarking (and all the remainder) has not been completed on Atlas. Instead, it has been completed on my personal PC, with its 8 cores.

Let's plot the increase in efficiency as the number of threads grows from 1 to 8. As we have seen so far, the execution time is the most coherent indicator of performance. We obtained the table below for a small ($N = 256$), a medium ($N = 1024$), and a large ($N = 4096$) matrix size. For all the plots to be visible in the same figure, we will have to use a "log" scale.

Number of threads	Small	Medium	Large
1	0.0063	0.560	41.32
2	0.0035	0.369	26.13
3	0.0025	0.270	21.61
4	0.0019	0.217	19.76
5	0.0016	0.189	18.60
6	0.0014	0.177	16.53
7	0.0013	0.188	18.96
8	0.0012	0.181	26.52

TABLE VI: Improvement comparison

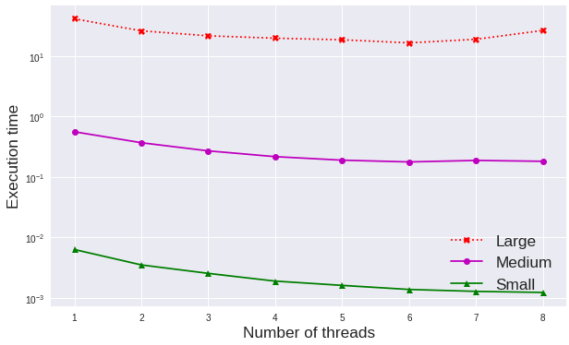

Fig. 3: Improvement comparison for small, medium and large matrix size (in log scale).

We can see from the plots that the execution time tends to decrease as the number of threads increases. However, there seems to be no additional benefit with more than 6 threads, in fact, the large matrix size shows that we lose performance. As we have seen in the previous sections, this is probably due to large amounts of cache-misses.

III. Conclusion

Optimizing a kernel can be as simple as reordering its nested loops to using HPC libraries such as OpenMP or BLAS for parallelization and vectorization. These findings suggest that whenever possible while performing matrix-matrix computations, BLAS should be the default choice. Moreover, when a multi-threading approach is used, it is important to know the kernel and the device's specifications, in order to properly calibrate the optimization and avoid loosing performance.

References

[1] M. Boileau. "Atlas cluster documentation". In: *GitLab Unistra* (2020). URL: `https://gitlab.math.unistra.fr/atlas/cluster-doc/-/wikis/description`.