PERFORMANCE MEASUREMENTS WITH MATRIX OPERATIONS

TD 2

By
PASQUIER Matéo

OBHPC
November 14, 2022

Summary

1 Problem Specification

We want to measure the speed at which a cpu can perform different matrix operations and the accuracy of those operations. Each operation will have multiple implementation in code and we will also use different compilators and optimization flags, in order for us to determine what would be the best configuration for a given operation. Matrix operations are very present in today's world of computing and are known to be computationally heavy, which is why it is important to optimize them the best we can.
We will test three matrix operations:

- DGEMM (Double Precision General Matrix Multiplication)
- Dot Product
- Reduction

Each of which will have different algorithms implemented in code.

2 Program and Architecture Design

We will be using code written in *C*. We will run the tests using a benchmark function whose purpose is mainly to calculate the time taken by an operation to complete. We will also use a shell script to automatize the execution of the different algorithms, which we will run in batches to make sure we get a decent and representative average result.
We will use the benchmark function's ability to print its measurements to save performance files of our tests, both to keep easier track of our work and later to generate plots which we will then use to more easily compare the results. The plots will be generated using the *gnuplot* library.

All of the experiment will be run in a Linux Mint environment (version Cinnamon 21) installed on a laptop that will be plugged into mains for the whole time to make sure we get optimal performances from the machine.
The cpu is an *x86 IntelCore i3* running at 2.53GHz with the following cache sizes:

Cache	Size
L1	32K
L2	256K
L3	3072K

Before running the experiments, we will fix the frequency of the cpu using *cpupower* and will also assign the execution to a single core to make sure the process scheduler doesn't migrate it at some point during the execution, which could negatively impact our resulting performances.

3 Test Cases

We'll test three different operations, each one of them will have multiple algorithms implemented in order to find out which is the fastest in a specific situation.

The implementations are as follows:

- DGEMM
  - IJK
  - IKJ
  - IEX
  - Unrollx4
  - CBLAS
  - Unrollx8
- Dot Product
  - Naive algortihm
  - Unrollx8
- Reduction
  - Naive algortihm
  - Unrollx8

Each of these implementations will be run using different optimization flags during compilation:

- OFast
- O1
- O2
- O3

Moreover, we will be using different compilators; namely *gcc* and *clang*. Intel compilers *icx* and *icc* were originally planned to be used as well, but since installation on the testing hardware turned out to be more complicated than expected, that idea was thus dropped.

4 Analysis and Conclusion

Let's focus on the DGEMM results first. When comparing each implementation and taking into account the compiler and optimization flag, it is apparent that *gcc* has an overall better execution speed than clang, except for the O2 flag where *clang* seems to have the upper hand, with as much as twice the execution speed as *gcc* on IKJ and IEX.
We observe though minimal differences for every implementations with the O1 flag, as it is only the first level of optimization the compiler will try less to improve the code. The more the code is changed, the more differences between compilers will be felt so those results make sense.

We can notice that the 8xUnroll seems a bit slower than the 4xUnroll. The goal of unrolling is to reduce the number of loops done by the program at the cost of a less maintainable code and a larger binary. This is done because the cpus will usually prefer executing instructions linearly rather than having to branch all the time. One would assume that reducing even further the overall number of loops would keep decreasing the execution speed, but it does not seem to be the case here.

One potential reason for that could be a too big of an unroll size. The more instructions we'll put in the loop, the more data the compiler will try to store in caches. If the unroll is too big, this might cause the cpu to have to access memory instead of caches, which will ultimately slow down the execution as a cache access is faster than a memory access.

When comparing all of the other implementation, on both compilers CBLAS is by far the fastest, no matter the optimization flags. On the contrary, the IJK algorithm seems to be the slowest, as expected since it is only a naive implementation with no regard to optimization. An interesting thing to note is that even though the resulting speeds will vary a bit, the result are pretty much constant relative to each others, meaning that a certain optimization flag or compiler will hardly make one implementation better than another one, which implies that we cannot rely on a compiler to better a program and thus optimization has to come mainly from the code we're writing.

For both the dot product and the reduction operation, the unroll implementation seem to be faster than the base one (this is more apparent for the dot product, the reduction results are closer to each others but still a difference can be made). The clang Ofast is always the fastest but at a price of a more limited precision, so it wouldn't be interesting to use that version on a daily basis to solve real problems. other than that, gcc has a slight upper hand on the dotprod with the unroll version, but seems to lose overall against clang with the naive implementation.
For the reduction, clang also has a better execution speed except with the O2 flag where gcc runs faster.
The O2 flag seem to have the best performances out of the other 2 (excluding Ofast because of its bad precision) with the reduction, when on the other hand with the dot product O3 has the better results. One interesting thing to notice

is that even when clang has the overall upper hand with execution speed, the fastest version will be with gcc no matter the optimization flags. This means that unlike what we could think with our first look at the results, gcc might be the best compiler choice as it can give the better performances, we only need to find the good parameters and flags for it beforehand, where clang is statistically more likely to give better results but it won't give us the most optimal execution speed.

5 References

- Github : `https://github.com/Datky/OBHPC-TD2`