Homework Assigment 3: Matrix-matrix multiplication

Scientific Software / Technisch Wetenschappelijke Software

Duplat Clémence

Practical info

Machine name: **SSH : SERAING**

Questions

Q1: What is the computational cost of the straight-forward implementation of the matrix-matrix multiplication of two $N \times N$, double precision floating point matrices? (1 line)

> $\mathcal{O}(N^3)$: 3 loops over N (complexity $\mathcal{O}(N)$) and in the loop basic operations(complexity $\mathcal{O}(1)$)

Q2: What are the storage requirements of the straight-forward implementation of the matrix-matrix multiplication of two $N \times N$, double precision floating point matrices? (1 line)

> $3 \times N^2 \times 8$ bytes : 3 matrices of N elements and each element stored in double precision (8 bytes)

Q3: Compile `matrixop.f90`, `timings.f90` and `mm_driver.f90` with each of the following compilers: `gfortran`, `ifort` and `nagfor`. For each compiler, use the `-O3` optimisation level. Discuss the most important differences between these compilers? (max. 7 lines)

> When compiling with ifort, we see a real improvement of the CPU timing performance and the relative error. It is maybe due to the fact that it utilize advanced vectorization and parallelization techniques making the loop order insignificant. However, only the function *matmul* has a less good performance. When using the nagfor compiler we don't see a huge difference

Q4: Compile `matrixop.f90`, `timings.f90` and `mm_driver.f90` with the `gfortran` compiler and the following compiler flag combinations: `-O0`, `-Og -fbounds-check`, `-O3 -funroll-loops` and `-Ofast -march=native -ffast-math`. Compare the results. What is the role of the different compiler flags? What do you conclude? When would you use which combination? (max. 8 lines)

> We know that -O0, which disable optimization, is the default flag. We can compare it to the *-Og -fbounds-check* which is a debugging flag. It is supposed to be slower because it checks to the array bounds, but in my case there is no big difference. Then we have an optimisation flag : *-O3 -funroll-loops*. *-O3* optimize compilation in a high level (turn on more optimization flags) and *-funroll-loops* optimizes loop execution by unrolling loops. However, in our case we see that it barely improve our computation. The final optimization flag is *-Ofast -march=native -ffast-math*. This is supposed to be the most aggressive flag with the highest performance but in my case I don't see a major difference.

We can conclude that the choice of algorithm is way more important for the performance of matrix-matrix multiplication than the compilers flags. When choosing an optimization flag, you must know if you are looking for debugging or performance and choose the corresponding one.

Q5: For `gfortran` using the optimization flag `-O3`, which method with three nested loops is the fastest? (1 line)

mm_jki with 1.3955 sec

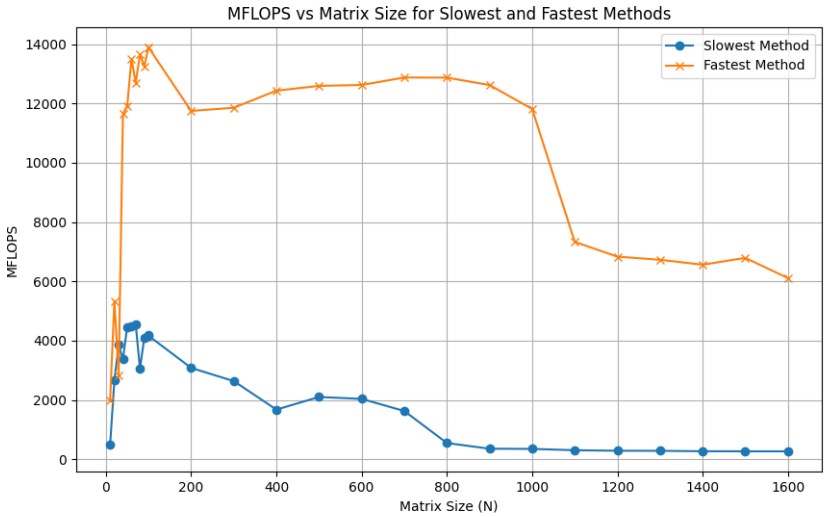
Q6: For `gfortran` using the optimization flag `-O3`, which method with three nested loops is the slowest? (1 line)

mm_kij with 30.4158 sec

Q7: Explain the difference in execution time between these two methods. Try to be as detailed as possible, but remain to the point. Highlight the most important concepts using **boldface**. (max. 15 lines)

The first thing to know is that in Fortran, unlike some other languages, the arrays are stored in **column-major order**. This means that two elements in a column are next to each other in the memory. To store the operations, the CPU operate on **caches**, intermediate storage that retains data. The method mm_jki access to the matrix A in a column-order and to the matrix B in a row-order. Thanks to this order loop when accessing to 2 elements, they are close to each other allowing the CPU to perform many operations within the same cache before going to a new one. For the method mm_kij it is the contrary: it access to the matrix A in a row-order and to the matrix B in a column-order. This implies that when the CPU perform, he has to access two elements that are far away in memory and it will bring in a new **chunck of memory** called **cache line** (the smallest unit of memory that can be transferred between the cache and main memory). This will imply **cache misses**: the required data is not in te cache line and we will to take it from the main memory, which is slower.

F1: Use the output of **I2** (compile with `gfortran -O3`) to make a figure that shows the number of floating point operations per second in function of N for the fastest and slowest method with three nested loops.



Q8: Briefly discuss Figure **F1**. Avoid repeating information from **Q7**. (max. 8 lines)

> We can see that for very small matrices, the difference of performance, indicated by MFLOPS, of the slowest and fastest method doesn't differ until it reach a certain matrix size. At this level we can clearly see that the fastest method is more performant. We can see a spike for both methods and then a decline. A decline indicate a cache memory or in other words the use of a new cache line and thus a cache miss. We can see that it correspond to the declarations said in **Q7**.

Q9: What are the (dis)advantages of blocking? (max. 5 lines)

> Blocking is used in matrix-matrix multiplication to divide the input matrices into smaller blocks that fits into low level caches and thus minimize cache misses. Large matrices that does not fit in the cache will be reduced to smaller matrices that well fit in them. The size of the blocks is very important, ideally the size should be such that the blocks of all three matrices can reside in the cache at the same time. This technique can improve the performance but it can also add complexity.

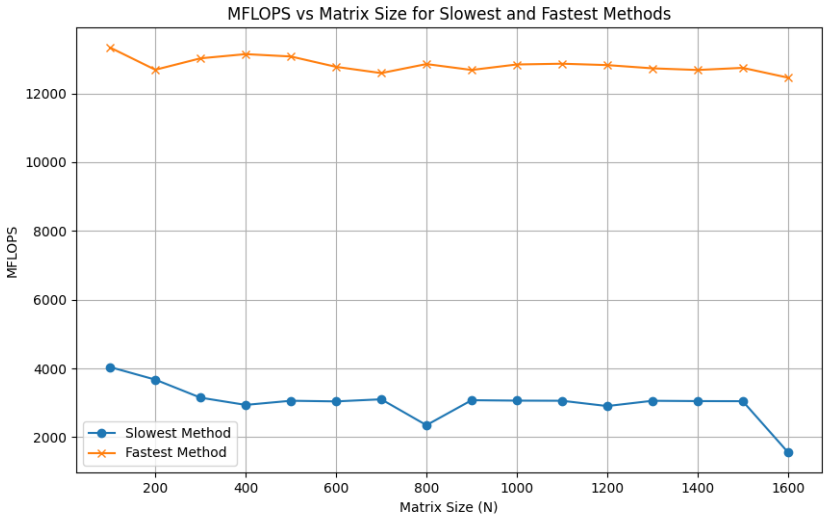
Q10: You can use `valgrind --tool=cachegrind ./executable` with `executable` the name of your executable to simulate the cache architecture of your machine for the given executable. For the fastest and slowest implementation with three nested loops, compare the cache efficiency of the variant without blocking and the blocked variant for blocksizes 25, 100 and 500. Also compare the execution time of these variants (run them without `valgrind`). Write several short programs to perform this experiment and compile them with `gfortran -O3`. Use N equal to 2000. Do you get the results you expect based on the previous questions. Discuss. (max. 8 lines)

> First we should know that L1 cache misses occurs when the data is not found in the first cache line and LL cache misses occurs when the data is not found in the last level of cache available in the cache hierarchy. This way, we will be able to see how the blocksize affects cache utilization and execution time. I was able to use valgrind but I don't really

		L1 cache misses	LL cache misses	Execution time
Fastest	No blocking			2.7570 sec
	Blocksize = 25			1.5650 sec
	Blocksize = 100			1.2967 sec
	Blocksize = 500			1.3616 sec
Slowest	No blocking			64.7674 sec
	Blocksize = 25			4.0775 sec
	Blocksize = 100			5.5315 sec
	Blocksize = 500			18.5714 sec

understand to what the output corrsepond, so I was not been able to write it down in the table.
However, we know that having a too big blocksize may be not effecient because the smaller matrices will also not fit in the cache. Choosing a too small blocksize will lead to a cache line that is not entirely used, also leading to cache misses. Therefore, it is usefull to find an intermediate blocksize.

F2: As in **F1**, plot the number of floating point operations per second in function of N for `mm_blocks_a` and N between 100 and 1600 using a blocksize of 100.



Q11: Briefly discuss the difference between **F1** and **F2**. (max. 3 lines)

We can see that now for mm_blocks_a, corresponding to the fastest method, there is no cache misses. Blocking will do a better cache line utilization because he works on smaller blocks that, this time, will fit into the cache. We can conclude that the choice of 100 for the blocksize is good.

Q12: Briefly discuss the practical relevance of the divide and conquer algorithm. A reference implementation - which assumes $N = 2^k$- is provided in `matrixop.f90`. (max. 5 lines)

This algorithm breaks recursively a large problem into smaller and more easy subproblems in order to simplify the problem. Each of these subproblems are then solved independently and combined to form a final solution. So, this is useful when we are

handeling large matrices that do not fit entirely in cache. It makes a more efficient use of the memory hierarchy. However, this is a recursive algorithm which can slow down the algorithm.

Q13: Briefly discuss the practical relevance of the Strassen's algorithm. A reference implementation - which assumes $N = 2^k$- is provided in `matrixop.f90`. (max. 5 lines)

We can modify the last algorithm to obtain the Strassen algorithm. The cache advantages are conserved but it increase the speed due to a restrict matrix multiplication of blocks(only compute untill M7 with divide and conquer). Thus, theoretically, it reduces the complexity of the algorithm to $\mathcal{O}(N^{2.8074})$ instead of $\mathcal{O}(N^3)$. It is easy to understand that for large matrices it will fasten the computation.