Submission for High Performance Computing Exercise 1

Authors: Danylo Kravchenko (kravch0000), Lukas Daugaard Schröder (schroe0006)

March 20, 2023

1 Cache Aware Programming

To create a cache-aware matrix multiplication, we need to know the cache size of our machine to store the data optimally in blocks. The mini-HPC supports the C directive to get the cache block size using the `sysconf(_SC_LEVEL1_DCACHE_LINESIZE)`. The main idea behind cache-aware multiplication is to perform multiplication of small blocks that could fit inside the cache block. So the data multiplication would be performed inside the fast memory. An important note on the blocks is to adjust the block size, so that we wouldn't get out-of-bounds errors.

Now we could define `T1` calls with correct matrix sizes inside the job script:

```
ml intel/2018b

icc T1.c -o T1
srun T1 200 0
srun T1 200 1
srun T1 400 0
srun T1 400 1
srun T1 600 0
srun T1 600 1
srun T1 800 0
srun T1 800 1
srun T1 1000 0
srun T1 1000 1
srun perf stat -e cache-misses ./T1
```

Output of the job:

```
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 200 and opt = 0.
MATRIX SIZE: 200, GFLOPS: 2.122146, RUNS: 5
Mean execution time: 0.007022
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 200 and opt = 1.
MATRIX SIZE: 200, GFLOPS: 2.516405, RUNS: 5
Mean execution time: 0.005922
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 400 and opt = 0.
MATRIX SIZE: 400, GFLOPS: 1.498924, RUNS: 5
Mean execution time: 0.079530
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 400 and opt = 1.
```

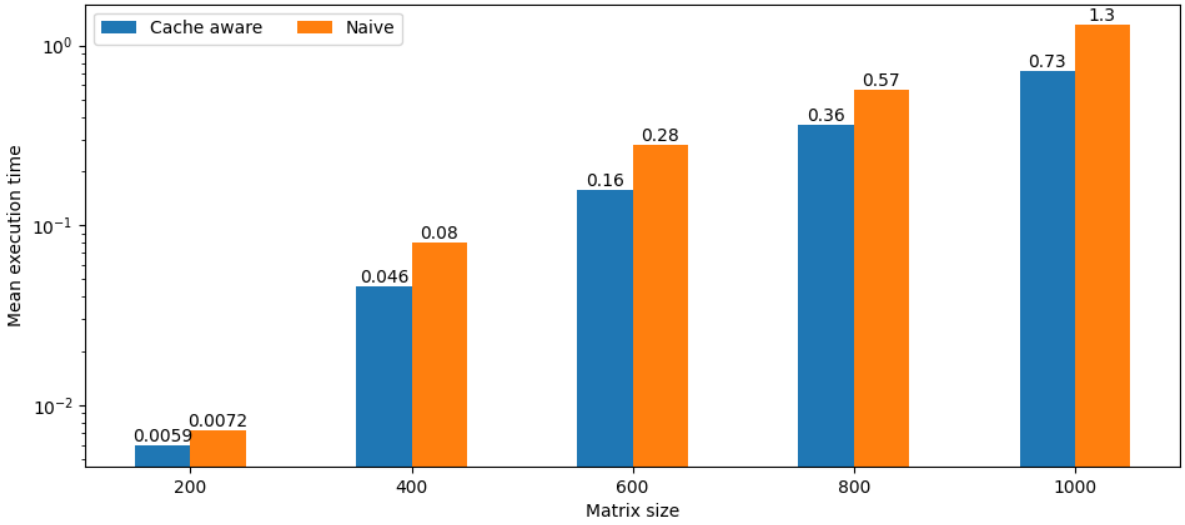
```
MATRIX SIZE: 400, GFLOPS: 2.593444, RUNS: 5
Mean execution time: 0.045966
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 600 and opt = 0.
MATRIX SIZE: 600, GFLOPS: 1.441385, RUNS: 5
Mean execution time: 0.279128
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 600 and opt = 1.
MATRIX SIZE: 600, GFLOPS: 2.569090, RUNS: 5
Mean execution time: 0.156605
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 800 and opt = 0.
MATRIX SIZE: 800, GFLOPS: 1.673019, RUNS: 5
Mean execution time: 0.570032
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 800 and opt = 1.
MATRIX SIZE: 800, GFLOPS: 2.634963, RUNS: 5
Mean execution time: 0.361931
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 1000 and opt = 0.
MATRIX SIZE: 1000, GFLOPS: 1.436531, RUNS: 5
Mean execution time: 1.296627
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 1000 and opt = 1.
MATRIX SIZE: 1000, GFLOPS: 2.567838, RUNS: 5
Mean execution time: 0.725375
USAGE
   ./T1 [SIZE] [opt]

 Performance counter stats for './T1':

               418      cache-misses

       0.001627772 seconds time elapsed

       0.000000000 seconds user
       0.001542000 seconds sys
```



2 Memory Access Patterns

The order of the loops can affect the memory access pattern and cache behavior, which in turn can affect the performance of the algorithm. For caching purposes, the most optimal ordering of loops will be slowest to fastest. We need sequential access to data to be fast.

We tried 2 different approaches:

KIJ (column by column): This variant goes through C and B column by column and A row by row. However, this is slower than another variant IKJ, since going through row by row is faster than column by column, due to how the matrices are stored in memory.

IKJ (row by row): This variant tries ordering of loops from slowest to fastest, we also go row by row in all matrices, so accessing elements in A, B and C in a contiguous manner improves efficiency. So, to calculate C we need only to go i times through B in a sequential manner.

So we decided to try both variants and added a new option `2` for IKJ loop ordering.

Define a new job script for T2:

```
# for T2:
icc T2.c -o T2
srun ./T2 2000 0
srun ./T2 2000 1
srun ./T2 2000 2
```

Output of the job:

```
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 2000 and ordering = 0.
MATRIX SIZE: 2000, GFLOPS: 0.668449, RUNS: 5
Mean execution time: 22.292131
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 2000 and ordering = 1.
MATRIX SIZE: 2000, GFLOPS: 2.243846, RUNS: 5
Mean execution time: 6.640902
```

```
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 2000 and ordering = 2.
MATRIX SIZE: 2000, GFLOPS: 3.772445, RUNS: 5
Mean execution time: 3.950001
```

We see that the order IKJ is the fastest because it reads data sequentially row by row. Theoretically, this means that our code should now be less memory bound than the IJK ordering.

3 Data Organization in the Memory

To perform operations on 1D array of size $[N \times N]$ instead of 2D array of $[N][N]$, we need to rethink how we access elements in the array. Now the element i, j could be accessed in the array by the index $i * N + j$. Converting the array to 1D increases the locality of data, now we need only sequential reads on 1D array.

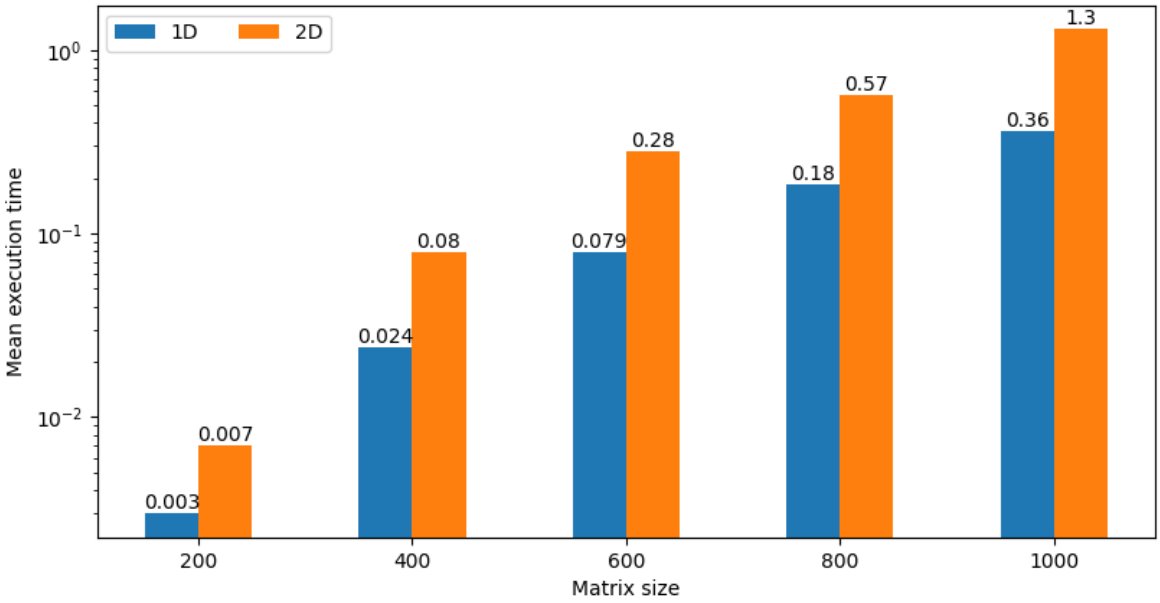
Define a new job script for T2:

```
# for T3:
icc T3_1D.c -o T3_1D
icc T3_2D.c -o T3_2D
srun ./T3_2D 200
srun ./T3_1D 200
srun ./T3_2D 400
srun ./T3_1D 400
srun ./T3_2D 600
srun ./T3_1D 600
srun ./T3_2D 800
srun ./T3_1D 800
srun ./T3_2D 1000
srun ./T3_1D 1000
```

Output of the job:

```
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 200.
MATRIX SIZE: 200, GFLOPS: 2.088798, RUNS: 5
Mean execution time: 0.007134
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 200.
MATRIX SIZE: 200, GFLOPS: 5.025872, RUNS: 5
Mean execution time: 0.002965
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 400.
MATRIX SIZE: 400, GFLOPS: 1.499483, RUNS: 5
Mean execution time: 0.079500
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 400.
MATRIX SIZE: 400, GFLOPS: 4.987887, RUNS: 5
Mean execution time: 0.023900
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 600.
MATRIX SIZE: 600, GFLOPS: 1.441978, RUNS: 5
Mean execution time: 0.279013
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 600.
```

```
MATRIX SIZE: 600, GFLOPS: 5.110714, RUNS: 5
Mean execution time: 0.078723
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 800.
MATRIX SIZE: 800, GFLOPS: 1.679252, RUNS: 5
Mean execution time: 0.567916
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 800.
MATRIX SIZE: 800, GFLOPS: 5.182760, RUNS: 5
Mean execution time: 0.184009
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 1000.
MATRIX SIZE: 1000, GFLOPS: 1.435568, RUNS: 5
Mean execution time: 1.297497
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 1000.
MATRIX SIZE: 1000, GFLOPS: 5.195882, RUNS: 5
Mean execution time: 0.358485
```



4 Auto-vectorization

Output of the report before optimization:

```
Intel(R) Advisor can now assist with vectorization and show optimization
  report messages with your source code.
See "https://software.intel.com/en-us/intel-advisor-xe" for details.


Begin optimization report for: multiply(int, double **, double **, double **)
```

```
    Report from: Interprocedural optimizations [ipo]

INLINE REPORT: (multiply(int, double **, double **, double **))


    Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]


LOOP BEGIN at T4.c(108,1)
   remark #25460: No loop optimizations reported

   LOOP BEGIN at T4.c(109,3)
      remark #25460: No loop optimizations reported

      LOOP BEGIN at T4.c(110,5)
         remark #25439: unrolled with remainder by 2
      LOOP END

      LOOP BEGIN at T4.c(110,5)
      <Remainder>
      LOOP END
   LOOP END
LOOP END

    Report from: Code generation optimizations [cg]

T4.c(100,61):remark #34051: REGISTER ALLOCATION : [multiply] T4.c:100

    Hardware registers
        Reserved      :    2[ rsp rip]
        Available     :   39[ rax rdx rcx rbx rbp rsi rdi r8-r15 mm0-mm7 zmm0-zmm15]
        Callee-save   :    6[ rbx rbp r12-r15]
        Assigned      :   16[ rax rdx rcx rbx rbp rsi rdi r8-r11 r14-r15 zmm0-zmm2]

    Routine temporaries
        Total         :      43
            Global    :      22
            Local     :      21
        Regenerable   :       3
        Spilled       :       4

    Routine stack
        Variables     :       0 bytes*
            Reads     :       0 [0.00e+00 ~ 0.0%]
            Writes    :       0 [0.00e+00 ~ 0.0%]
        Spills        :      32 bytes*
            Reads     :       4 [3.60e+00 ~ 0.2%]
            Writes    :       4 [1.60e+00 ~ 0.1%]

    Notes

        *Non-overlapping variables and spills may share stack space,
         so the total stack size might be less than this.
```

Compiler gives us an intuition, that we need to vectorize the innermost loop. We could do that by separating the addition of values `a[i][k] * b[k][j]` into a sum, and then write this sum into `c[i][j]` outside the innermost loop. This reduces the number of memory accesses required during the inner loop, allowing the compiler to better optimize the code for vectorization.

Report after optimization:

```
Intel(R) Advisor can now assist with vectorization and show optimization
  report messages with your source code.
See "https://software.intel.com/en-us/intel-advisor-xe" for details.


Begin optimization report for: multiply(int, double **, double **, double **)

    Report from: Interprocedural optimizations [ipo]

INLINE REPORT: (multiply(int, double **, double **, double **))


    Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]


LOOP BEGIN at T4.c(104,2)
   remark #25460: No loop optimizations reported

   LOOP BEGIN at T4.c(105,3)
      remark #25460: No loop optimizations reported

      LOOP BEGIN at T4.c(107,4)
      <Peeled loop for vectorization>
      LOOP END

      LOOP BEGIN at T4.c(107,4)
         remark #15300: LOOP WAS VECTORIZED
      LOOP END

      LOOP BEGIN at T4.c(107,4)
      <Alternate Alignment Vectorized Loop>
      LOOP END

      LOOP BEGIN at T4.c(107,4)
      <Remainder loop for vectorization>
      LOOP END
   LOOP END
LOOP END

    Report from: Code generation optimizations [cg]

T4.c(100,61):remark #34051: REGISTER ALLOCATION : [multiply] T4.c:100
```

```
    Hardware registers
        Reserved      :    2[ rsp rip]
        Available     :   39[ rax rdx rcx rbx rbp rsi rdi r8-r15 mm0-mm7 zmm0-zmm15]
        Callee-save   :    6[ rbx rbp r12-r15]
        Assigned      :   31[ rax rdx rcx rbx rbp rsi rdi r8-r15 zmm0-zmm15]

    Routine temporaries
        Total         :     125
            Global    :      32
            Local     :      93
        Regenerable   :       3
        Spilled       :       8

    Routine stack
        Variables     :       0 bytes*
            Reads     :       0 [0.00e+00 ~ 0.0%]
            Writes    :       0 [0.00e+00 ~ 0.0%]
        Spills        :      64 bytes*
            Reads     :       8 [3.29e+01 ~ 0.2%]
            Writes    :       8 [2.94e+01 ~ 0.2%]

    Notes

        *Non-overlapping variables and spills may share stack space,
         so the total stack size might be less than this.
```

---

5 Optional Task

The final step is to combine every optimization we have seen so far into a single method. We did 1D, cache awareness, vectorization, and IKJ loop order. Our final version:

```
void non_cache_aware_multiply(int n, double * a, double * b, double * c) {
    // combine 1D, vectorization optimization, and ikj loop ordering
    int i, j, k;

    for (i = 0; i < n; i++) {
        for (k = 0; k < n; k++) {
            double a_value = a[i*n + k];

            for (j = 0; j < n; j++) {
                c[i*n + j] += a_value * b[k*n + j];
            }
        }
    }
}


void multiply(int n, double * a, double * b, double * c) {
    // find a cache block size for data to be cache aware
```

```
    int cache_block_size = sysconf(_SC_LEVEL1_DCACHE_LINESIZE);

    // matrix size is smaller than cache_block_size, naive version is preferred
    if (cache_block_size > n) {
        return non_cache_aware_multiply(n, a, b, c);
    }

    int i, ii, j, jj, k, kk;

    // Go to the start of the block
    for (i = 0; i < n; i += cache_block_size) {
        for (j = 0; j < n; j += cache_block_size) {
            for (k = 0; k < n; k += cache_block_size) {
                int ii_upper_bound = (n < i + cache_block_size) ? n : i + cache_block_size
                int jj_upper_bound = (n < j + cache_block_size) ? n : j + cache_block_size
                int kk_upper_bound = (n < k + cache_block_size) ? n : k + cache_block_size

                // Go through the block elements and perform the multiplications
                for (ii = i; ii < ii_upper_bound; ii++){
                    for (kk = k; kk < kk_upper_bound; kk++) {
                        double a_value = a[ii*n + kk];

                        for (jj = j; jj < jj_upper_bound; jj++) {
                            c[ii * n + jj] += a_value * b[kk*n + jj];
                        }
                    }
                }
            }
        }
    }
}
```

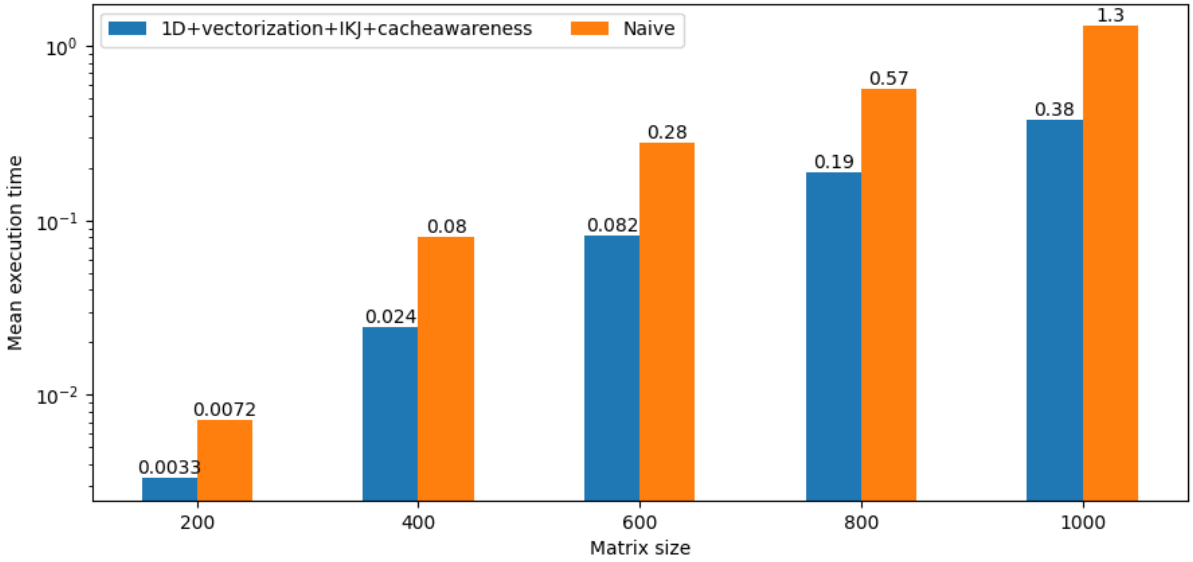
Define a new job script for T5:

```
# for T5:
icc T5.c -O3 -o T5
srun ./T5 200
srun ./T5 400
srun ./T5 600
srun ./T5 800
srun ./T5 1000
```

Output of the job:

```
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 200.
MATRIX SIZE: 200, GFLOPS: 4.478859, RUNS: 5
Mean execution time: 0.003327
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 400.
MATRIX SIZE: 400, GFLOPS: 4.879935, RUNS: 5
Mean execution time: 0.024428
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 600.
```

```
MATRIX SIZE: 600, GFLOPS: 4.900004, RUNS: 5
Mean execution time: 0.082108
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 800.
MATRIX SIZE: 800, GFLOPS: 5.095039, RUNS: 5
Mean execution time: 0.187177
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 1000.
MATRIX SIZE: 1000, GFLOPS: 4.927926, RUNS: 5
Mean execution time: 0.377977
```



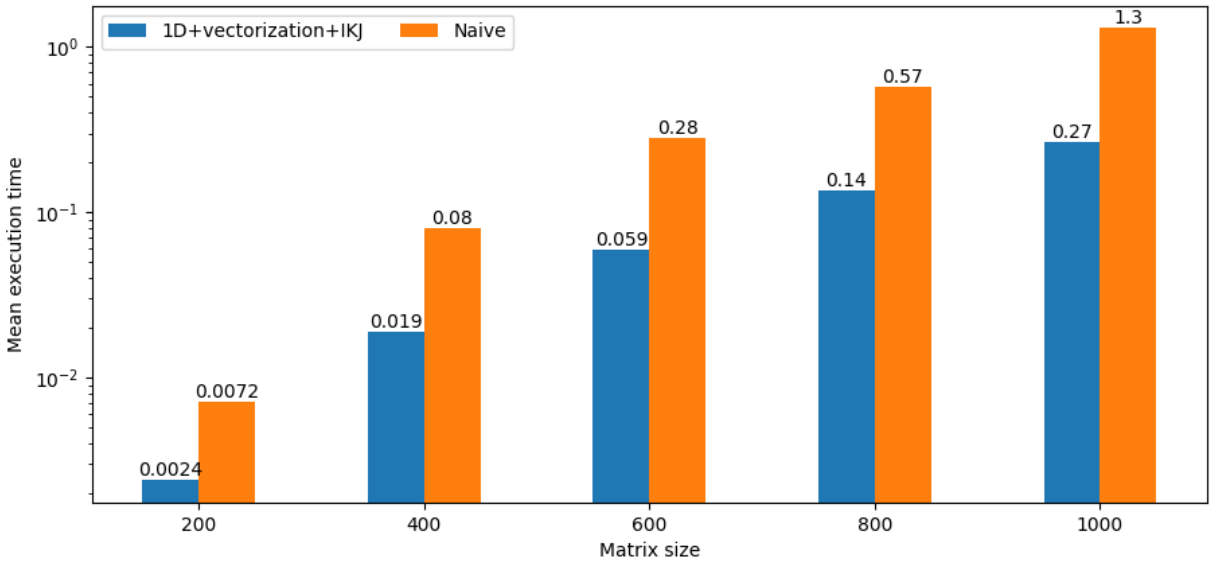
What's interesting, compiled using `gcc -O3` it is way faster on Danylo Kravchenko's laptop (Apple M1 Silicon) than on the HPC machine.

```
Timer resolution is 1000 nano seconds.
Starting benchmark with mSize = 1000.
MATRIX SIZE: 1000, GFLOPS: 9.327955, RUNS: 5
Mean execution time: 0.199684
```

Output of the job without cache awareness (method above):

```
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 200.
MATRIX SIZE: 200, GFLOPS: 6.218790, RUNS: 5
Mean execution time: 0.002396
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 400.
MATRIX SIZE: 400, GFLOPS: 6.320834, RUNS: 5
Mean execution time: 0.018860
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 600.
MATRIX SIZE: 600, GFLOPS: 6.824041, RUNS: 5
Mean execution time: 0.058958
```

```
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 800.
MATRIX SIZE: 800, GFLOPS: 7.034505, RUNS: 5
Mean execution time: 0.135571
Timer resolution is 1 nano seconds.
Starting benchmark with mSize = 1000.
MATRIX SIZE: 1000, GFLOPS: 7.023130, RUNS: 5
Mean execution time: 0.265216
```



Again, compiled using `gcc -O3` it is way faster on the laptop than on the HPC machine.

```
Timer resolution is 1000 nano seconds.
Starting benchmark with mSize = 1000.
MATRIX SIZE: 1000, GFLOPS: 11.185217, RUNS: 5
Mean execution time: 0.166527
```