Debunking execution times

Unguru Dragos-Gabriel, 336CC

- ***Brief introduction***

This project represents an analysis of the possible optimization methods for multiplying two matrices.

The implementation scans the execution times and uses a vectorized square matrix of varying dimensions. This was ran on the university's cluster on an Intel Xeon, E5630, 2.53 GHz, IBM HS22 with 4 nodes.

- ***Methods used***

The main approach of the project is to examine 5 different implementations to optimize the computing time:

1. Using a FORTRAN-implemented library (BLAS).
2. A normal, unoptimized implementation as a reference point.
3. A manual "by hand" optimization using various methods.
    a. Without any optimizing compilation directives.
    b. Compiled with -O3.
    c. Compiled with -O3 alongside other potential optimization flags.

The first two methods are taken into account more as a reference point in analyzing the behavior of the 3rd implementation and to see how far we can go with it, so further we'll dive in with our "by hand" method.

There are 3 main techniques used to improve the execution time in our implementation:

- **Cache friendly memory accesses** are obtained using the *ikj* addressing scheme which minimizes cache misses. When multiplying matrices, we have 3 different matrices to address and by ordering our indices we can take advantage of the **spatial locality** characteristic. Using this addressing scheme we'll obtain only sequential and constant addressing.
- **Easing pointer arithmetics** by putting the task of pointer addressing and arithmetic on the programmer's shoulders. We can take the load of index calculations off of the compiler's hands into ours by carefully moving through the memory space.

- **Forcing the use of registers** by telling the compiler to keep certain values into registers using the “register” keyword in C. This goes perfectly hand-in-hand with the pointer arithmetic method as it makes computations way faster by avoiding memory accesses.

- ***Execution times***

Having these three methods at hand, we can now look at the improvements these brought. We’ll run a series of tests with N varying in the 120-1600 interval and we’ll obtain the following execution times:

BLAS:

N=120: Time=0.002272

N=200: Time=0.008367

N=400: Time=0.047913

N=800: Time=0.230258

N=1200: Time=0.683977

N=1600: Time=1.534171

NEOPT:

N=120: Time=0.043363

N=200: Time=0.152548

N=400: Time=1.059998

N=800: Time=8.561721

N=1200: Time=28.962095

N=1600: Time=90.703903

OPT_M:

N=120: Time=0.006648

N=200: Time=0.016975

N=400: Time=0.136567

N=800: Time=1.036693

N=1200: Time=3.473088

N=1600: Time=9.308175

OPT_F:

N=120: Time=0.006236

N=200: Time=0.019492

N=400: Time=0.208531

N=800: Time=1.795025

N=1200: Time=5.810676

N=1600: Time=18.981875

OPT_F_EXTRA:

N=120: Time=0.004158

N=200: Time=0.013661

N=400: Time=0.198918

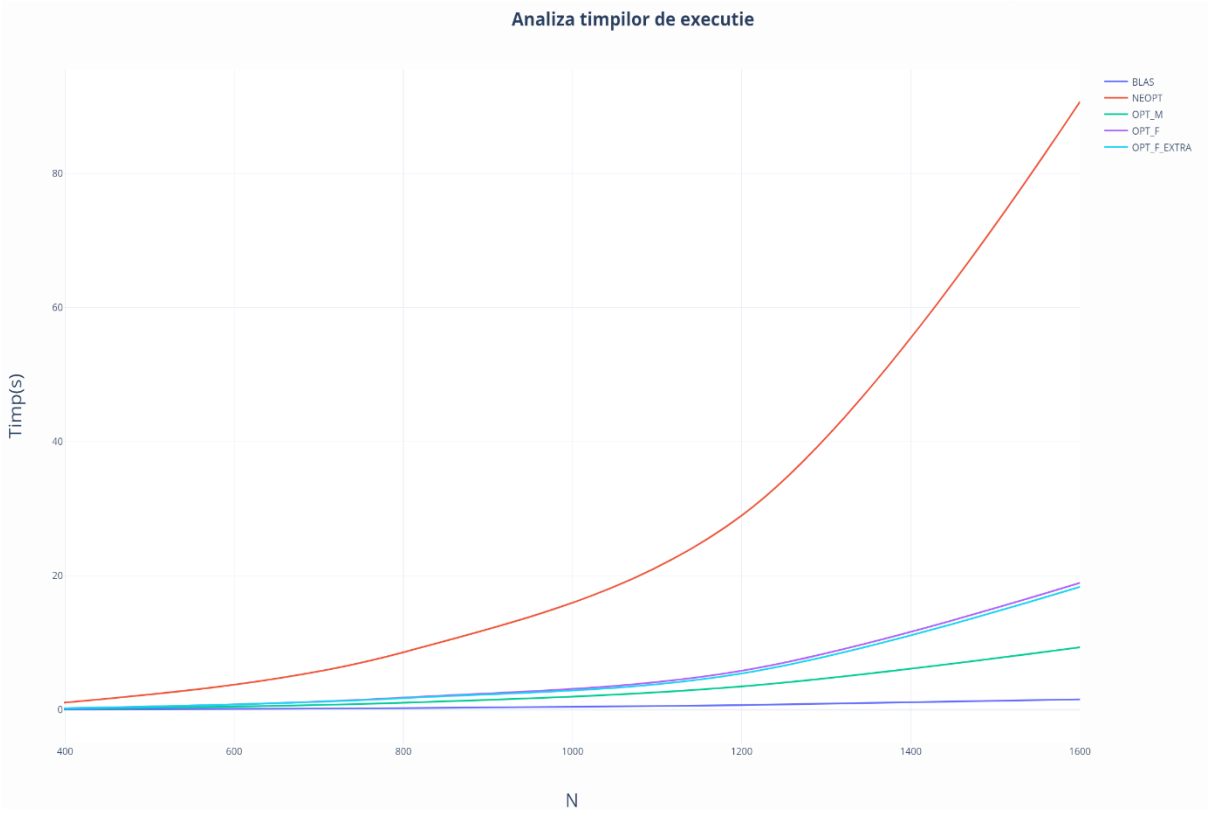
N=800: Time=1.719354

N=1200: Time=5.398800

N=1600: Time=18.101095

Looking at the *N=1200* tests, we can see that we’ve obtained a roughly 88% improvement in execution time using our optimization method compared to the unoptimized one and a constant under 4 seconds execution time.

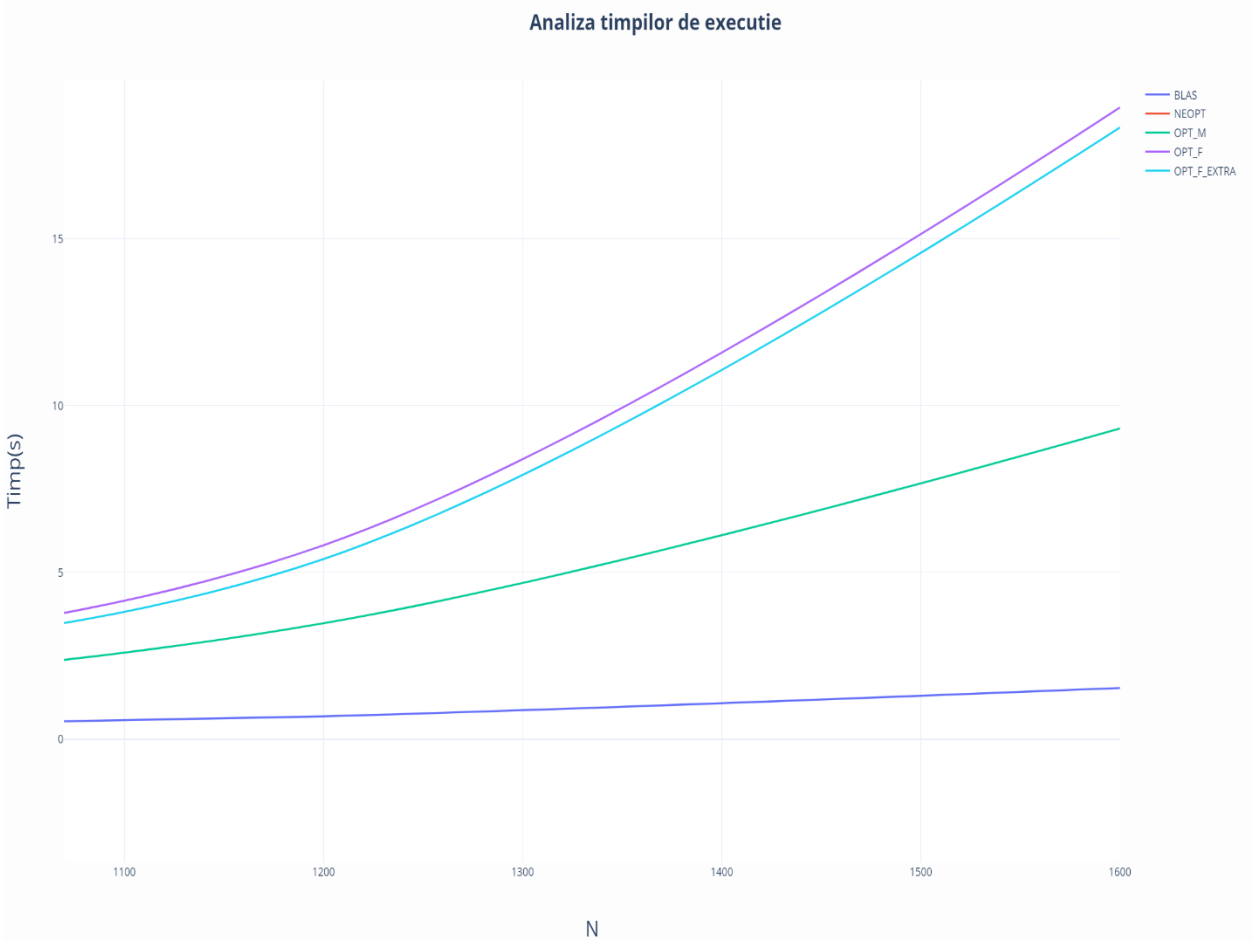
To better visualize, we'll plot a graph to grasp the data.



At a zoomed-out first glance, the results are nothing short of obvious and expected. Every approach is following an exponential curve as the value of N increases, the number of floating point operations increases yet more (the matrix is of size NxN). And as the number of elements increases we can observe the gap between every other method greatly increases at a faster pace.

An observation worth noting besides the obvious fact that the unoptimized method is by far the slowest and the FORTRAN library is by way faster that we can obtain, is the fact that none of the enhanced-optimization compilation flags did any good to our "by hand" method, not even the -O3 flag.

Besides the BLAS library, our three optimization methods described earlier are by far the best results we can acquire and any other flag only prevents further improvements to our method.



We now focus on *OPT_M, OPT_F* and *OPT_F_EXTRA*. At a closer look we can see that the flags did absolutely no good to our methods. But we did find some flags that did some good comparing to just the -O3 optimization flags.

When looking for compilation directives to further improve our execution times, we look for flags that affect the two scenarios that take most of the time when multiplying matrices: **floating point operations and loop optimization.**

The one we can be certain that it comes with advantages in minimizing the execution times is by forcing the compiler to optimize the floating point operations by all means, even if we sacrifice some precision. Let’s take a look at the chosen flags and what tradeoffs we’ve made.

Basically, we've enabled everything that ***-ffast-math*** entails, for once:

1. -fno-math-errno: Tells the compiler not to set errno after calling math functions that are executed with a single instruction. This still maintains IEEE arithmetic compatibility.

2. -funsafe-math-optimizations: Allow optimizations for floating-point arithmetic that assume that arguments and results are valid and may violate IEEE or ANSI standards.

3. –ffinite-math-only: Skip checks and allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or +-Infs.

4. -frounding-math: Disable transformations and optimizations that assume default floating-point rounding behavior.

5. -fcx-limited-range: States that a range reduction step is not needed when performing complex division. Also, there is no checking whether the result of a complex multiplication or division is NaN + I*NaN

6. -fno-trapping-math: Compile code assuming that floating-point operations cannot generate user-visible traps. These traps include division by zero, overflow, underflow, inexact result and invalid operation

With the above presented flags we can have the certainty that we're going in the right direction as we heavily rely on floating point operations.

Although, we can't be certain that the loop optimizations might bring any improvements. So we have to further investigate if they bring any.

When talking about loop-optimizing flags we talk about:

1. -frename-registers: Avoid false dependencies in scheduled code by making use of registers left over after register allocation.
2. -funroll-loops: Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop.
3. -fweb: Constructs webs as commonly used for register allocation purposes and assign each web individual pseudo register.

To see if loop optimizations brings any further improvement, we'll run a some tests, now with a wider interval of values for N.

With loop-optimization:

OPT_F:

N=1200: Time=5.664481

N=1600: Time=19.009760

N=2000: Time=38.391289

OPT_F_EXTRA:

N=1200: Time=5.174283

N=1600: Time=17.343134

N=2000: Time=36.132530

5.89% improvement

(for N = 2000)

Without loop-optimization:

OPT_F:

N=1200: Time=5.610603

N=1600: Time=18.659763

N=2000: Time=39.002316

OPT_F_EXTRA:

N=1200: Time=5.385399

N=1600: Time=17.008125

N=2000: Time=38.137341

2.22% improvement

(for N = 2000)

For a better understanding of the impact of these flags, we'll plot the data to a graph and observe how the gap becomes wider and more importantly, it gets wider even with small tests. Seeing this, we're certain that this isn't a one time thing that occurred accidentally, but is a consistent improvement.