Parallel computing - Lab 1

Benjamín Ocampo[1], **Mateo de Mayo**[2]

[1]nicolasbenjaminocampo@gmail.com, [2]mateodemayo@gmail.com

Description of the Problem

The codebase is a mathematical implementation of the so-called Navier-Stokes equations for fluid flows occurring in a space. The idea is to represent fluids as a number of particles that live in a point of the space. Each set of particles will have an associated velocity in order to estimate how they move around it. This representation will be used to figure out the behavior-flows if forces are applied to a certain group of particles.

In this case, the space is modeled on a finite square $(N+2)(N+2)$ where the cell (i, j) will store the number of particles on that point of the space (denoted by *density*), and their *velocity*.

Measurements

In order to find out how to measure the performance of a program, we look for the main, computer-intensive, and repeatable action that the program was doing. They were three functions called:

- react: Adds density and velocity to a particular fluid in the space when they decrease to a particular threshold. It might be called the source force.
- vel_step: Updates the velocity of all the cells in the space.
- dens_step: Updates the density of all the cells in the space.

Since the three of these functions updates squillions of cells every step in the execution of headless (Giving rise to a *memory-bound* problem), we might use the time needed to perform each of them. Nevertheless, that is a size-dependent measure. So we decided to use the time needed to update each cell of the grid normalized in terms of its size. This is calculated as:

$$time\_p\_cell = \frac{function\_time}{grid\_size}$$

Since these updates lead to a huge amount of loads, and cache references, another measure we used:

$$cacheref\_p\_cell\_it = \frac{References}{(grid\_size * steps)}$$

Let us explain what it means. Given a $grid\_size = N * N$ and $steps = m$, we take the number of *cache references* needed to perform the m steps. How many of them are necessary to update one cell?. We normalized all the references dividing them by *m* times the *grid_size* in order to obtain the number of cache references needed to update one cell per iteration.

How have we compared two instances of our program?

Different configuration of *steps* and *N* (which is the large of our matrix representation) were used. In order to obtain 2^{16} updates on each run, the parameters were as following:

- $N = 128$, $steps = 512$
- $N = 512$, $steps = 128$
- $N = 2048$, $steps = 32$
- $N = 4096$, $steps = 16$
- $N = 8192$, $steps = 8$

After that, by means of our measures and plots we decided to reject or not to reject an heuristic. Another process of comparison was *test of hypotheses*.

A test of hypotheses is a method for using sample data to decide whether a null hypothesis (H_0) should be rejected in terms of an alternative hypothesis (H_a). On one hand, the prior belief that the current version of our program has an average performance $\mu = \mu_0$, where μ_0 is the average ns needed to update a cell, is the null hypothesis denoted by H_0. On the other hand, the alternative hypothesis, denoted by H_a, is the claim $\mu < \mu_0$, i.e, the average performance decreases with the proposed approach.

In order to deal with hypotheses proofs, random variables were needed:

- $X_1, X_2, ..., X_n$ Where X_i: ns per cell needed to perform *react* in step i.
- $Y_1, Y_2, ..., Y_n$ Where Y_i: ns per cell needed to perform *vel_step* in step i.
- $W_1, W_2, ..., W_n$ Where W_i: ns per cell needed to perform *dens_step* in step i.
- $Z_1, Z_2, ..., Z_n$ Where $Z_i = X_i + Y_i + W_i$

Since the amount of samples that can be produced is large, each of them represented by a random variable equally distributed in every step, a ztest hypothesis process about a population mean can be used. So, the final recipe to decide if a program is faster than another one by means of test of hypotheses is:

1. A heuristic is proposed to make the code faster.
2. Two versions of our program are obtained.
3. The functions *react*, *vel_step*, and *dens_step* will be executed during **N** steps. Where **N** is sufficiently large.
4. Samples $x_i's, y_i's, w_i's, z_i's$ are computed to obtain the average ns needed to perform all the functions, that is $\overline{z}$.
5. The observed value is normalized and used to calculate the p-value.
6. If the p-value is lower than our level of significance $\alpha = 0.1$, H_0 is rejected. Otherwise, H_0 is not rejected.
7. An output is produced according to the decision given by 6.

Scripts

Every time we had a heuristic, results, hypotheses proofs, branch comparison, graph ploting, and profiler checking were necessary. It's hard to do manually. How can we do them without batting an eyelid? We have made Python scripts that:

- Run different versions of the program (and with different parameters) in a bunch.
- Compare two git branches and their outcomes.
- Record in directories each result given by our tests.
- Take results, summarize them, and produce human-readable information.

Optimizations

i,j swap

It was not hard to uncover the first heuristic. Ater inspecting the codebase, we noticed that all the loops were accessing badly to the grid. If we look at the following code:

```
// Cache unfriendly
#define IX(i,j) ((i)+(N+2)*(j))
void expensive_loop ( int N, int *grid){
    for ( i=1 ; i<=N ; i++ ) {
        for ( j=1 ; j<=N ; j++ ) {
            // Do something with grid[IX(i, j)]
        }
    }
}
```

In this case, the grid will be read taking jumps of length N+2 due to the inner loop. Since C is row-major, that might lead to squillion caché misses if N is sufficiently large. Nevertheless, it can be solved interchanging the loop order, or i, and j in the definition of IX.

```
// Cache friendly
#define IX(i,j) ((j)+(N+2)*(i))
void expensive_loop ( int N, int *grid){
    for ( i=1 ; i<=N ; i++ ) {
        for ( j=1 ; j<=N ; j++ ) {
            // Do something with grid[IX(i, j)]
        }
    }
}
```

In order to know if the approach was fruitful we had to compare it with the codebase, wich we called *baseline*. Both of them were compiled with the flag -O3.

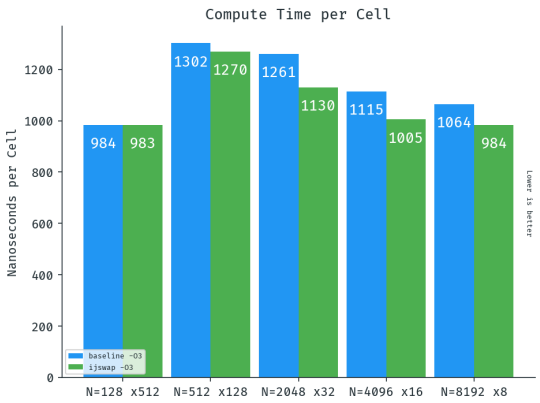

Figure 1.

As we thought, the performance tends to decrease when **N** increases. We can also see how the cache is affected by means of *perf*.

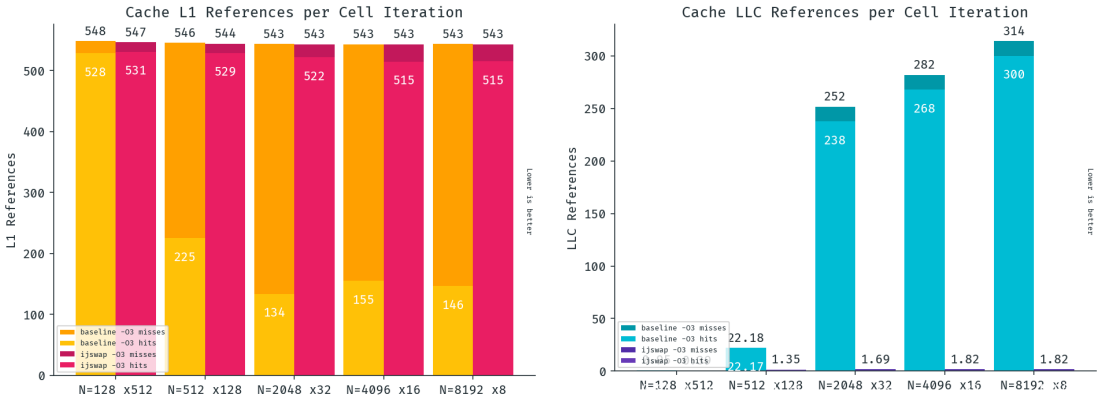

Figure 2.

Here we have that the number of cache L1 hits per cell iteration needed by *i,j swap* are greater than *baseline* when N increases. It is important to note the relation given by the cache L1 hits, and the LLC (*Last level caché*) references. As long as the number of cache hits increase, there is no reason to reference the LLC cache. Therefore, it is going to decrease.

Then, we found out that there is a flag called *-floop-interchange* which might do the job for us, and *-floop-nest-optimize* which assures loop blocking. So, we decided to check its usefulness. Unfortunately, we got no improvement.

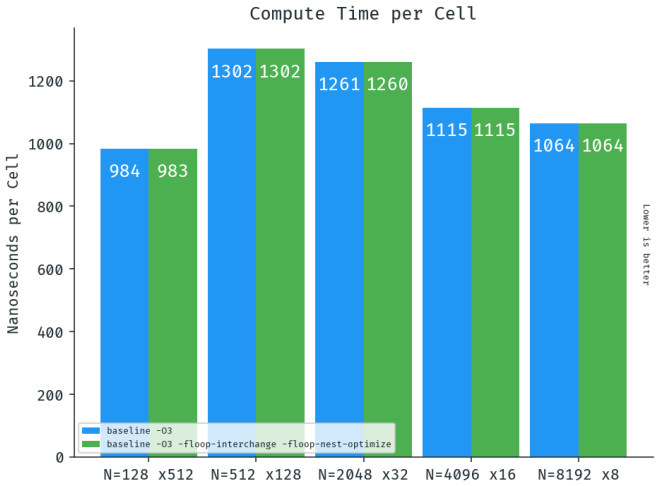

Figure 3.

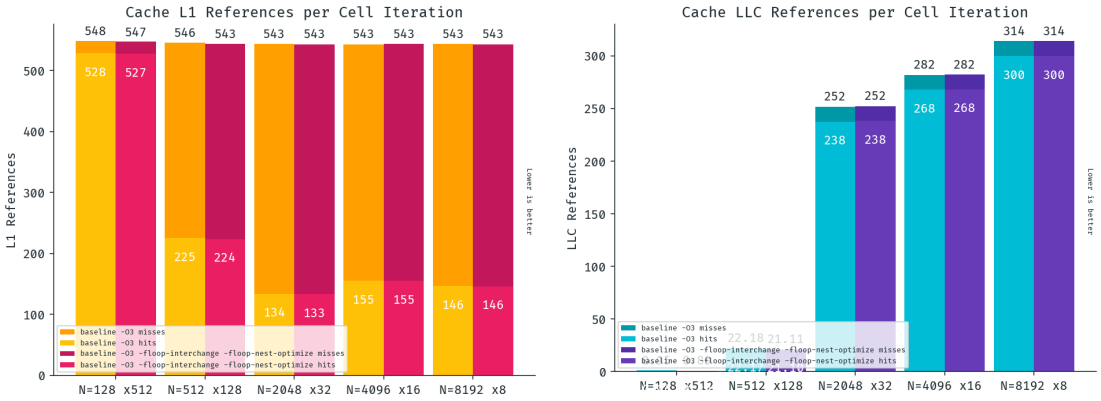

Figure 4.

Nevertheless, we noticed that the optimization is done by the compiler if it knows the value of N. An example of this is the following function:

```
float square(short n, float x){
    float sum = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            sum += x[i + j * n];
    return sum;
}
```

In this case, the compiler does not know the value of **n** which will be passed as an argument to the function *square*. Nevertheless, if we replace it by another value in the body of the function the compiler interchanges the loops. Since this a loss of modularity, re-define **n** is not an option. So, we take the *i,j swap* approach as an improvement.

Multiplicative inverse (invc)

Up to this point, there was no evident approach so the use of *cozprofiler* and *perf* lead to our second heuristic.

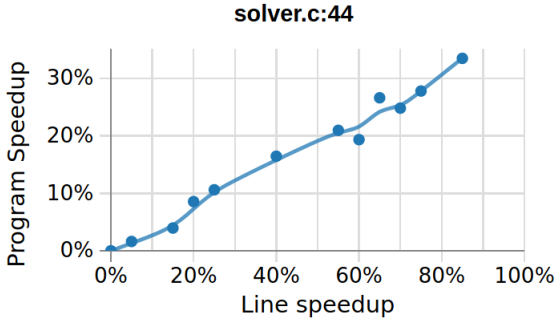

Figure 5. Line-speedup plot given by cozprofiler

The graph assures that if we improve the line 44 of *solver.c*, a program speedup will be obtained. So, we check that line which leads us to the procedure *lin_solve*.

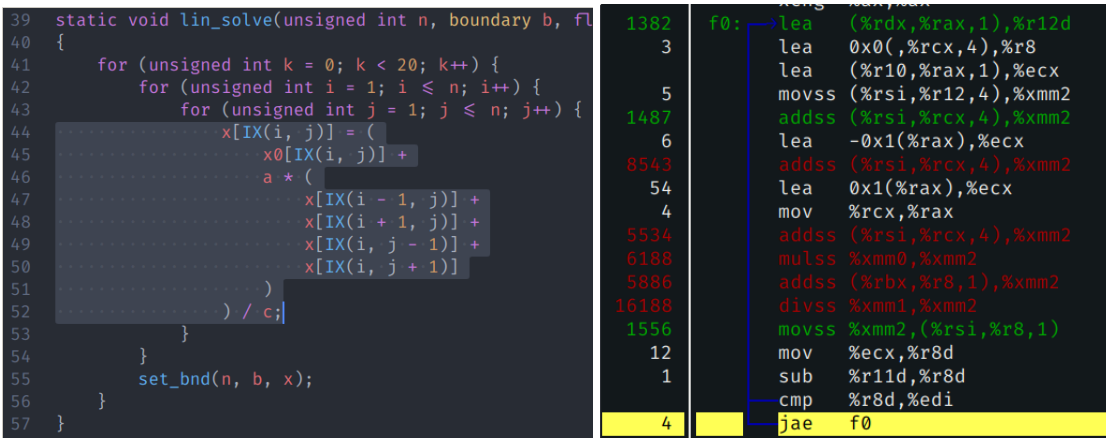

Figure 6. Line 44 of solver.c and assembly code given by perf of that line

They wanted we put our focus on the division taken by the procedure in *lin_solve*. That would bring us the most ”bang for our buck” so we focused on it. Thinking about a possible replacement, we decided to use the multiplicative inverse of c (Which we called **invc**), instead of a division.

```
// Expensive division repeated inside of the loop.
for k=1 to 20 do
    for i=1 to n do
        for j=1 to n do
            x[IX(i, j)] =  big_operation / c;
        od
    od
od
```

```
// Cheaper multiplication.
inv_c = 1 / c;
for k=1 to 20 do
    for i=1 to n do
        for j=1 to n do
            x[IX(i, j)] =  big_operation * inv_c;
        od
    od
od
```

After this change, *perf* showed us that the multiplication replacement is less expensive than the division.

```
1450  100:  lea   (%rdx,%rax,1),%r12d
   5        lea   0x0(,%rcx,4),%r8
   1        lea   (%r10,%rax,1),%ecx
   7        movss (%rsi,%r12,4),%xmm1
1538        addss (%rsi,%rcx,4),%xmm1
   8        lea   -0x1(%rax),%ecx
8504        addss (%rsi,%rcx,4),%xmm1
  31        lea   0x1(%rax),%ecx
   3        mov   %rcx,%rax
5693        addss (%rsi,%rcx,4),%xmm1
6143        mulss %xmm0,%xmm1
5751        addss (%rbx,%r8,1),%xmm1
6157        mulss %xmm2,%xmm1
1511        movss %xmm1,(%rsi,%r8,1)
   4        mov   %ecx,%r8d
            sub   %r11d,%r8d
            cmp   %r8d,%edi
   1        jae   100
```

Figure 7. Assembly code given by perf after invc optimization

Repeating again our comparison between our current program (*i,j swap* approach), and this new likely improvement. Both of them compiled with -O3. We have got the following results:

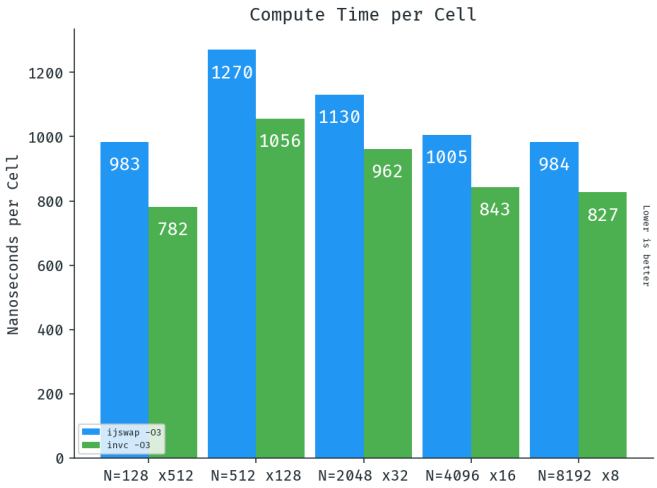

Figure 8.

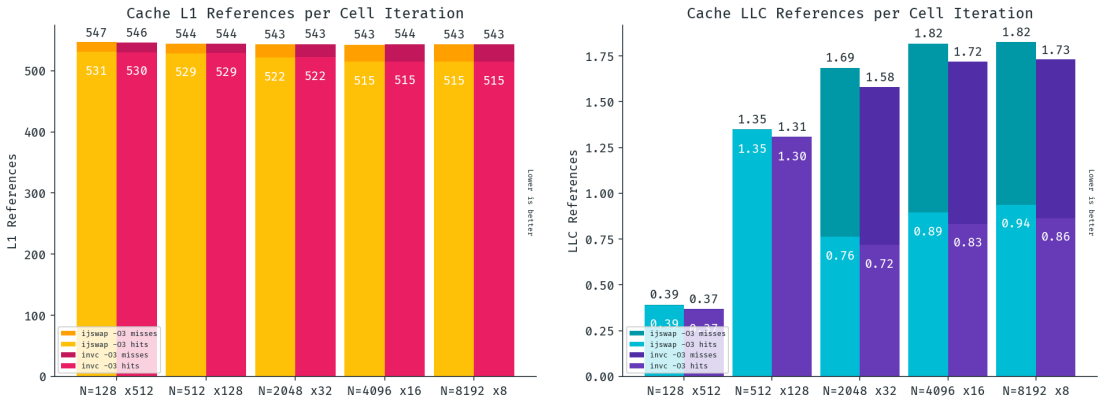

Figure 9.

Effectively, **invc -O3** is better than **i,j swap -O3**! This made us think that maybe upon the possible loss of precision that multiplying might represent, *-O3* abstained from implementing this optimization. And indeed, when using -Ofast which includes *-ffast-math* the compiler implemented the same optimization by means of the flag *-freciprocal-math*. In order to verify our hypotheses, a comparison between **invc -O3** and **i,j swap -Ofast** was performed.

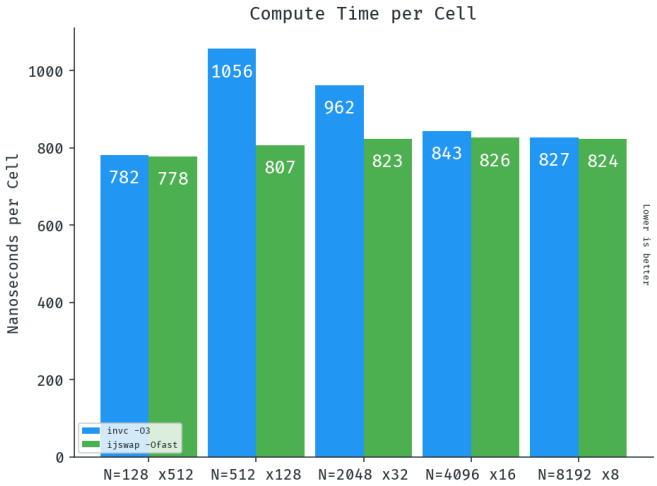

Figure 10.

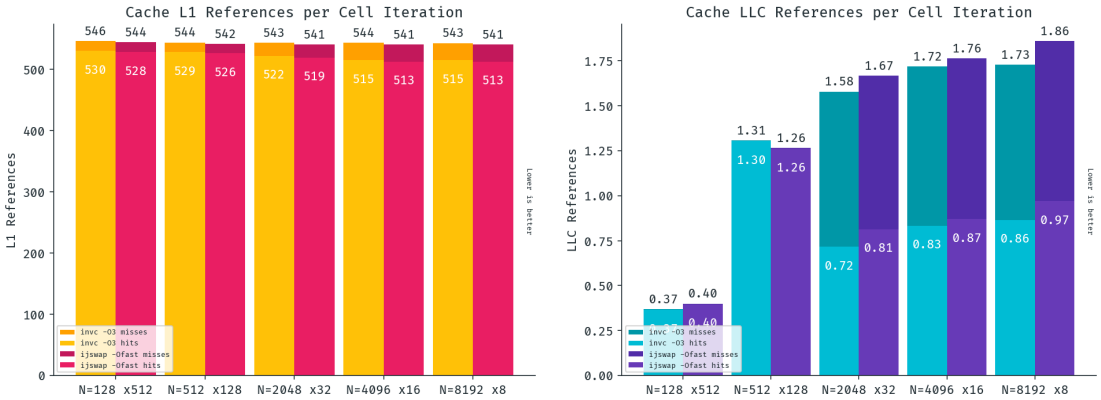

Figure 11.

Results were in favor of **i,j swap -Ofast**. Nevertheless, it is almost the same that our approach (And even better). This result was dissapointing for us because we lost so much time with this idea when the use of just one flag solved it. Since -Ofast did not degrade our simulation results, we decided to use it instead of our ”ad-hoc” approach.

We also wanted to know if the optimization **i,j swap -Ofast** might be replicated by only using *-freciprocal-math* (Which does the multiplicative inverse optimization). Unfortunately, that was not the case, and the results were in favor of **i, j swap -Ofast**

After all this process we got the following conclusions:

- *-Ofast* increases the performance in comparison with -O3 and does the **invc** optimization.
- *-freciprocal-math* did not do the **invc** optimization.
- *-Ofast* did not degrade the simulation results.
- If we know our flags we might save a lot of time!

Trying so many flags

Up to this point, we decided to try possible improvement by means of flag combinations in order to avoid our previous mistake. These approaches were unrolling loops, loop blocking, functions and modules optimizations. Therefore, the following flags were checked:

- *-Ofast*
- *-Ofast -march=native*
- *-Ofast -march=native -funroll-loops*
- *-Ofast -march=native -funroll-loops -floop-nest-optimize*
- *-Ofast -march=native -funroll-loops -floop-nest-optimize -flto*

Comparing these possible combinations we lead to the following conclusions:

- -march=native increases the performance by itself.
- There was no improvement with *-floop-nest-optimize*, *-flto*, or *-funroll-loops*.

Declaring N as const (constn)

Recalling our previous discovery by means of the function *square*, a new heuristic emerged. Again, if we declare `const int N = 2048;` but in this case for the entire program all the previous flags which did not work lead to a huge improvement in the processor *I7 7700HQ*. Nevertheless, that was not the case for *zx81*. The results of our tests were as following:

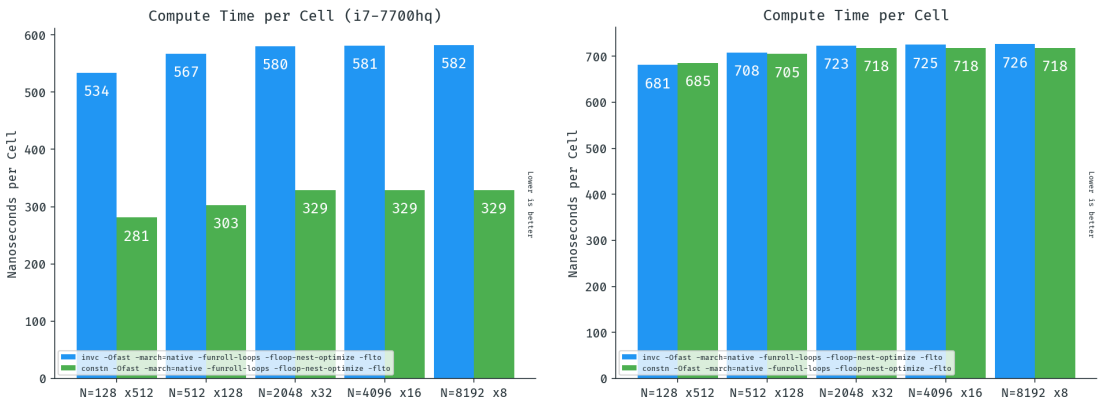

Figure 12. i7-770hq — zx81

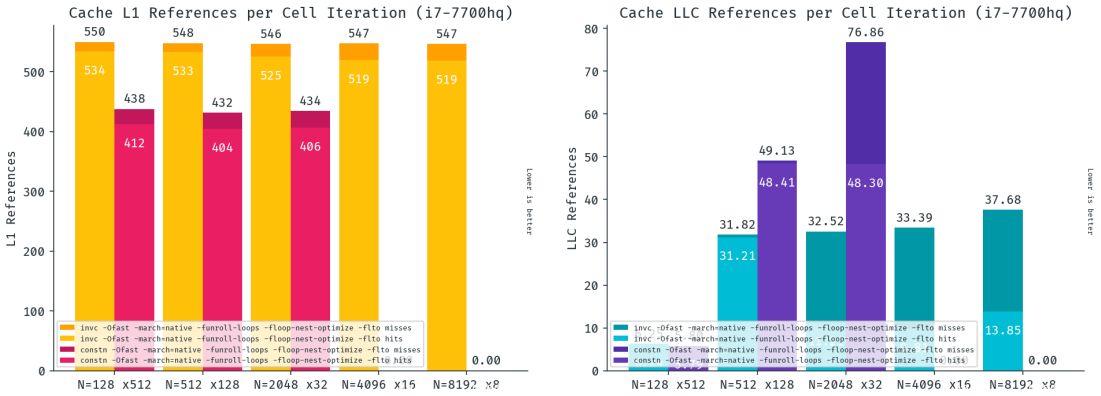

Figure 13. i7-770hq

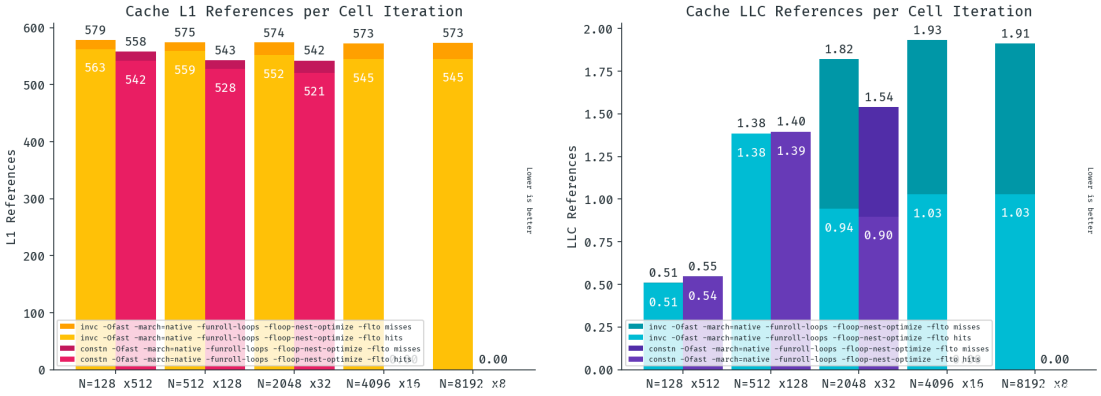

Figure 14. zx81

Taking a thorough look at the generated assembly code of the procedure *lin_solve* (Since it is the most expensive procedure of our program) we found out small differences. On one hand, in the *no-constn version*, **lea** instructions are used in order to know the address of the values stored in the grid by means of the parameter **N**. On the other hand, in the *constn version*, they were not necessary because the value of **N** is propagated and used to know the address without an extra instruction. Therefore, we thought that our bottleneck might be those **lea** instructions. Nevertheless, in *zx81* was not a bottleneck and the improvement was not so much without them.

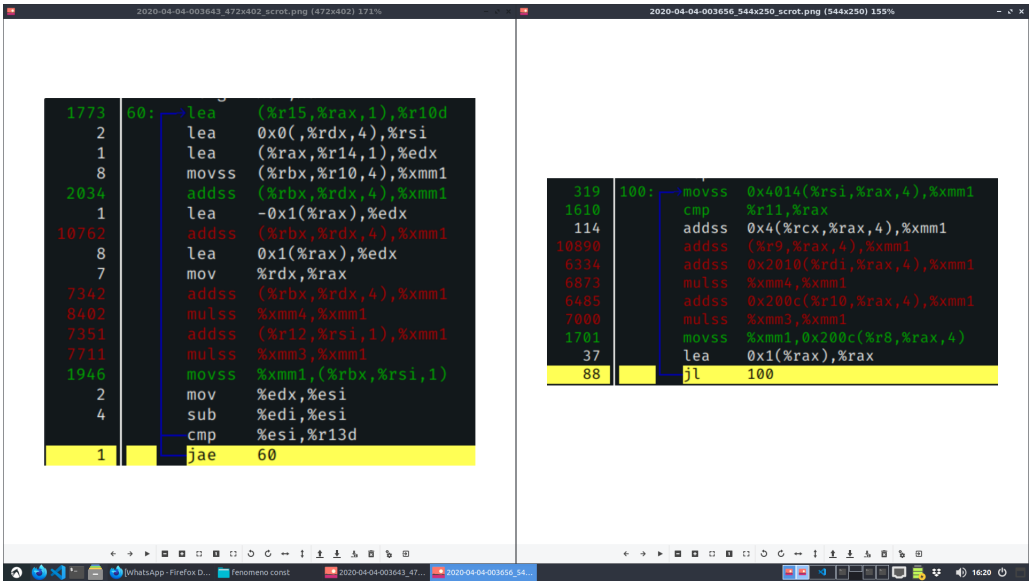

Figure 15. i7-7700hq assembly code. Left: no constn — Right: constn

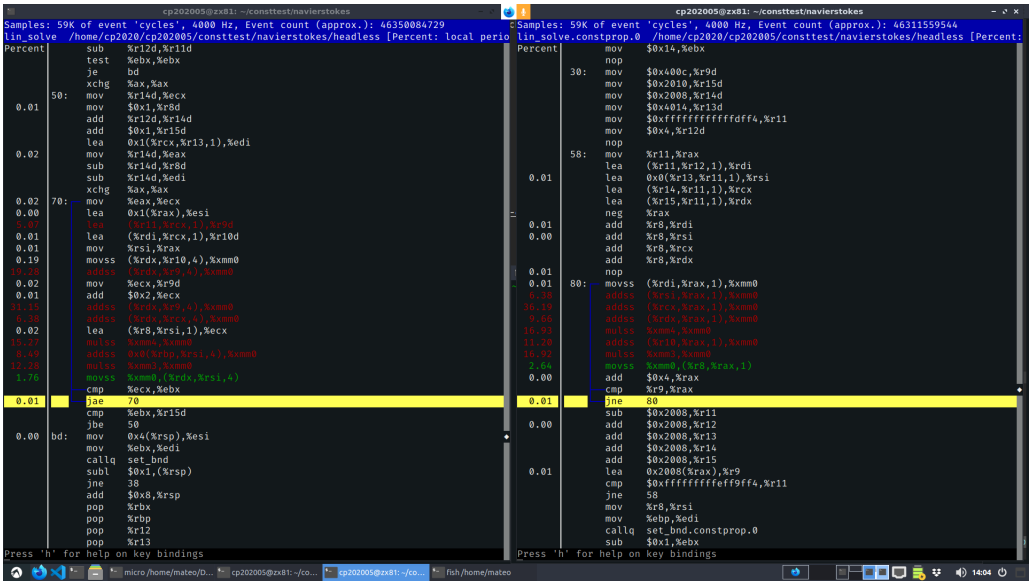

Figure 16. zx81 assembly code. Left: no constn — Right: constn

More on lin_solve

The *lin_solve* function is implementing a very basic gauss seidel linear ecuation solver for $k = 20$ iterations. This approach traverses the entire grid k times missing the cache. We thought about other numerical methods like jacobi. However, it is not yet clear how they would be implemented in this context. We also looked at a second paper from the author on the same topic where he recommends using the fishpack library from netlib. Also the time-skew appeared many times when looking for performant stencil iterations. However, these last two methods were not deeply studied.

We figured an ad-hoc solution that tries to imitate the gauss-seidel iterations but doing so in small tiles that fit in cache, doing the k iterations with the cells that com-

prises the tile without removing them until they have been completely iterated over. The algorithm can be found in the custom-linsolve branch. It improved the performance by a factor of up to four. However, it created artifacts in the tile borders which we considered deal breaker, so we ended up not using it.

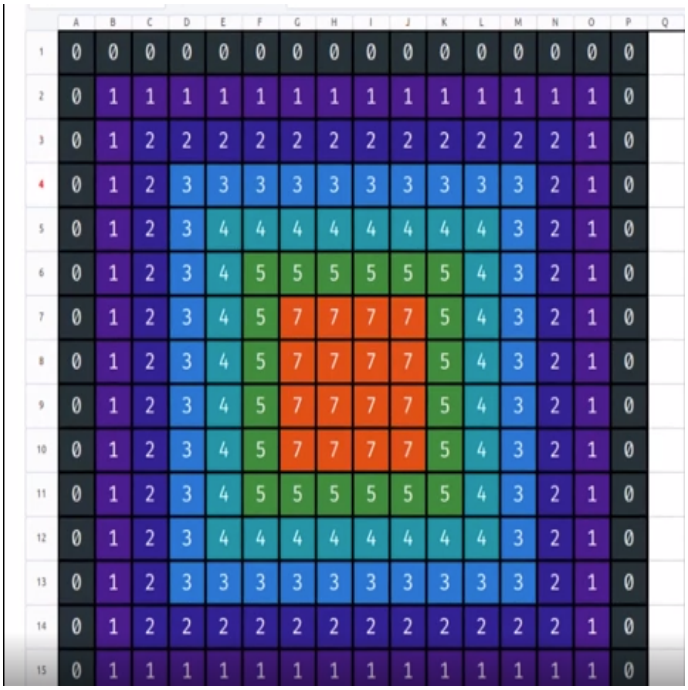

Figure 17. Ad-hoc algorithm video
Link to the gif if it can not be reproduced

Basic Blocking (bblock)

What we learned from our custom-made *lin_solve* is that even with a much heavier computational solution we benefited from the cache utilization, maybe we could get some of it from somewhere else. So, we tried doing manual loop blocking, even though we thought -floop-block had already done it.

Since *-floop-nest-optimize* were not useful to assure loop blocking inside of *lin_solve*. We decided to implement loop blocking by ourselves. The idea was to decrease the number of cache misses that *lin_solve* might be doing because of its cross-shaped accesses. After running our scripts we noticed that it did improve the performance.

```
for (unsigned int k = 0; k < 20; k++) {
    for (int ti = 0; ti < N - 2; ti += tile_width) {
        for (int tj = 0; tj < N - 2; tj += tile_height) {
            for (int ii = 0; ii < tile_width; ii++) {
                for (int jj = 0; jj < tile_height; jj++) {
                    const int i = 1 + ti + ii;
                    const int j = 1 + tj + jj;
                    x[IX(i, j)] = (
                        x0[IX(i, j)] +
                        a * (
                            x[IX(i - 1, j)] +
                            x[IX(i + 1, j)] +
                            x[IX(i, j - 1)] +
                            x[IX(i, j + 1)]
                        )
                    ) * invc;
                }
            }
        }
    }
    set_bnd(n, b, x);
}
```

Figure 18. Basic blocking of *lin_solve* procedure

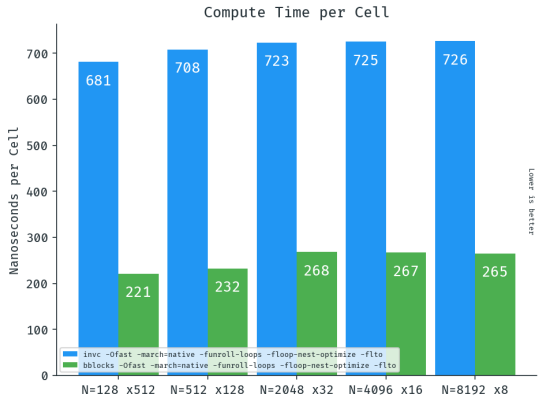

Figure 19.

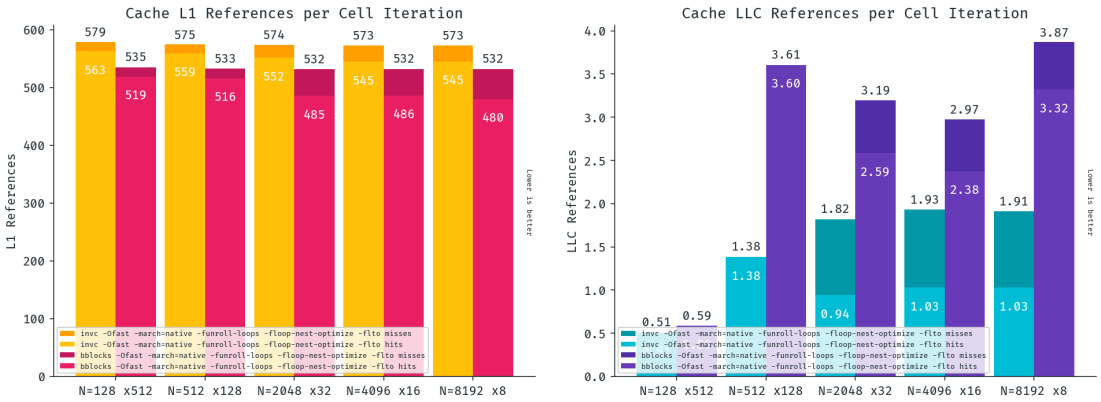

Figure 20.

How much have we improved?

In order to compare where we were at the beginning of the laboratory, the last comparison was performed. In this case, our *baseline* compiled with -O0 and -O3, and all our heuristics and flags together, *bblock* compiled with *-Ofast -march=native -funroll-loops -floop-nest-optimize -flto*.

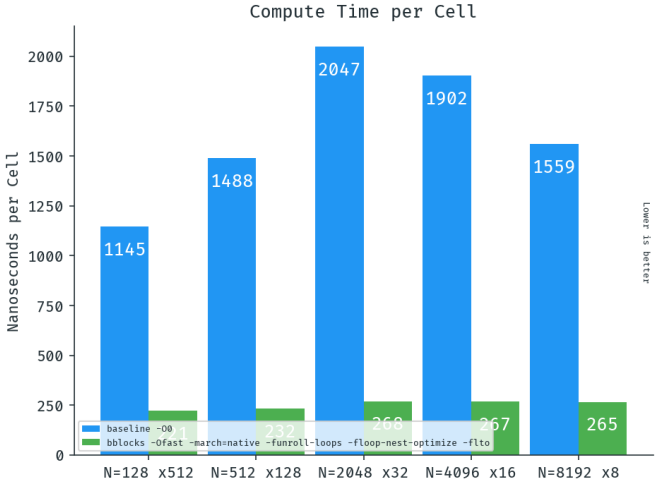

Figure 21.

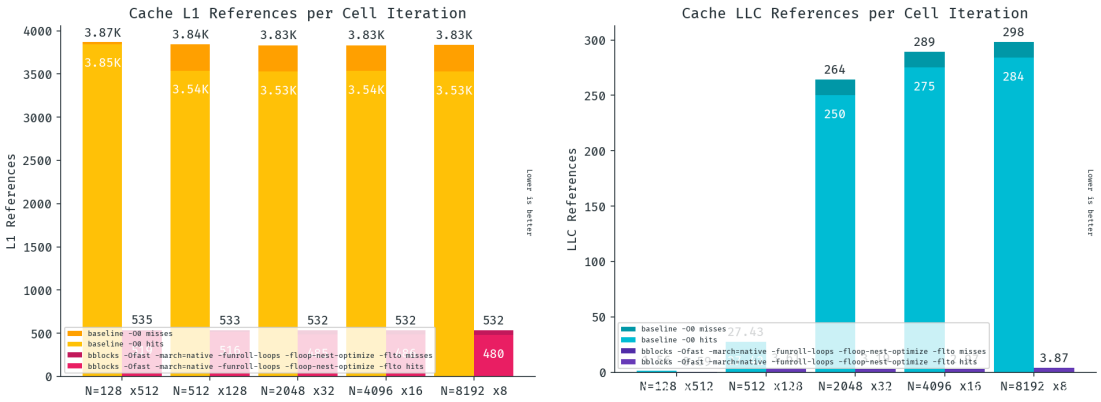

Figure 22.

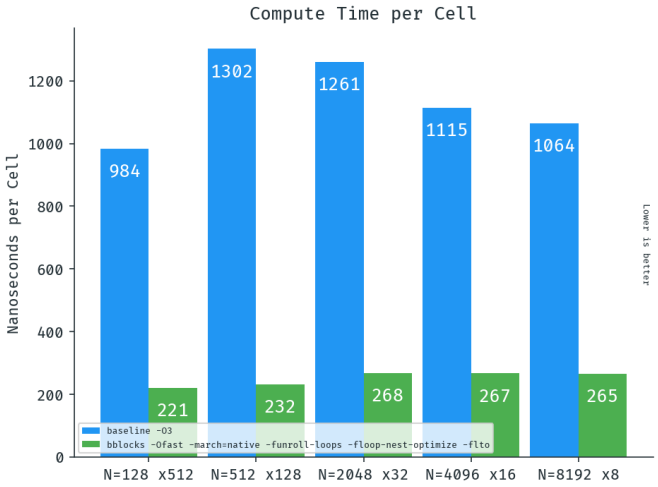

Figure 23.

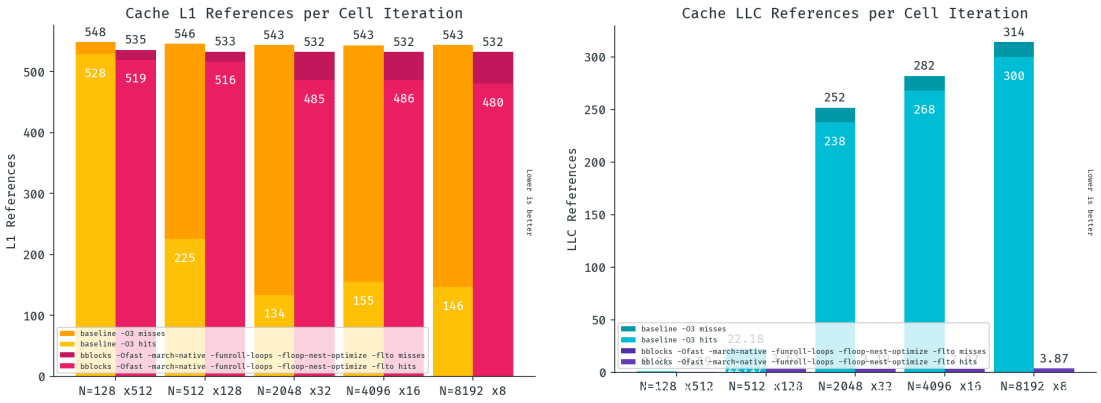

Figure 24.

Things that ended under the table

llvm-mca

Llvm-mca tries to emulate your cpu microarchitecture and tells you which ports are being under/overused. It draws a pipeline diagram to better understand how the instructions are flowing through the processor. We didn't dive too deep with this tool, but we are eager to do so as it seems promising.

Autofdo

It is a google system that aims to simplify feedback directed optimizations (fdo) to the compiler. The idea is that you compile your program with the ability to record some feedback during its run. Then feed the runtime data back into the compiler so that it can get a better picture of your program and of which flags it would benefit from. Unfortunately in our tests, it worsen the performance by a bit so we opted out of it.

Cachegrind

It simulates a cpu with a cache trying to much the host machine cache, however in our tests it predicted our bblocks solution would have performed two times worse than without the cache blocking, even though that was no what we saw in the final benchmarks.

-flto Crashes

How can we crash a compiler? The use of -flto leads to this unexpected errors which were really strange for us.

Array of Structures

One possible heuristic was the use of AoS (*Array of Structures*) in order to store the *velocity* of each cell of the grid. Since they were accessed together, they should be together we thought. Nevertheless, the same results were obtained by means of SoA (*Structure of Arrays*) which had been given to us in the *baseline*.