Parallel computing

Lab 1 - Navier Stokes

Mateo de Mayo - Benjamín Ocampo

FaMAF

April 20, 2020

Description of the Problem

Mathematical model so-called Navier-Stokes Equations for fluid flows ocurring in Nature.
Fluids modeled on a finite square $(N+2)x(N+2)$.
Basic structure of the solver. Three main operations:

- Add forces.
- Diffuse.
- Move.

These operations were repeated over and over during n steps in three functions:

- react
- dens_step
- vel_step

Goal: Compare different programs with different sizes.
Squillions of cells were updated by the procedures *react*, *dens_step*, and *vel_step*.
So we decided to use the time needed to update each cell of the grid, which 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:

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

Two contradictory hypotheses under consideration.
One assures an improvement (in terms of performance) on the previous one.
A *test of hypotheses* was used to decide if an approach provides strong support to reject the previous version.

Let us define the following random variables.

- $X_1, X_2, ..., X_n$: ns per cell needed to perform react in step i.
- $Y_1, Y_2, ..., Y_n$: ns per cell needed to perform vel_step in step i.
- $W_1, W_2, ..., W_n$: ns per cell needed to perform dens_step in step i.
- $Z_1, Z_2, ..., Z_n$: ns per cell needed to perform the three functions in step i.

Where $Z_i = X_i + Y_i + W_i$.

Compare two programs by means of samples $z_i's$.
N sufficiently large.
$H_0 : \mu = \mu_0$
$H_a : \mu < \mu_0$
Where μ_0: average ns needed to update a cell.
H_a is the claim that the time needed to update a cell decreases with the proposed approach.

Test of Hypotheses Based on Samples

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.
4. Samples $z_i's$ are computed to obtain $\bar{z}$.
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 α, H_0 is rejected. Otherwise, H_0 is not rejected.
7. An output is produced according to the decision given by the test.

Scripts

Every time we had an heuristic, we had to get results, test hypotheses, plot graphics, check the profiler, etc.
It's hard to do manually.
How can we do them without batting an eyelid?

Scripts

We needed Python scripts which do the following:

- Run different version 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.

Hardware

CPU:

- Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- 2 tri-core processors with hyperthreading (12 virtual)
- Freq: min 1200mhz, base 2400mhz, max 3200mhz
- L1d 64 sets, 8-way, 32KiB * 12 = 384KiB
- L1i 64 sets, 8-way, 32KiB * 12 = 384KiB
- L2 200 sets, 8-way, 256KiB * 12 = 3MiB
- L3 12Ki sets, 20-way, shared, 15MiB * 2 = 30MiB

Memory:

- Capacity : 126GiB
- Channels: 4
- Speed: 1600/1866Mhz
- Max BW: 59GB/s

Hardware

Compiler:

- gcc (Debian 9.2.1-31) 9.2.1 20200306

Uname:

- Linux zx81 5.4.0-4-amd64 #1 SMP Debian 5.4.19-1 (2020-02-13) x86_64 GNU/Linux
- Linux jupiterace 5.4.0-4-amd64 #1 SMP Debian 5.4.19-1 (2020-02-13) x86_64 GNU/Linux

Baseline: The basecode compiled with -O3

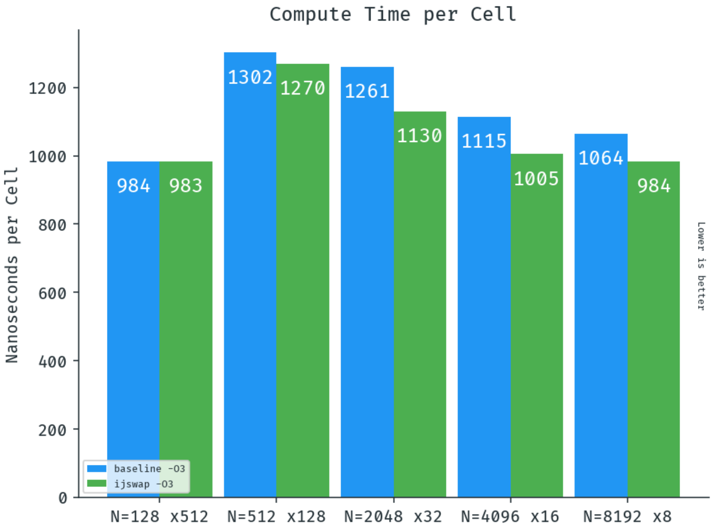
Heuristic 1: i, j swap.

```
#define IX(i,j) ((i)+(N+2)*(j))
// Cache unfriendly
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)]
        }
    }
}
```

```
#define IX(i,j) ((j)+(N+2)*(i))
// Cache friendly
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)]
        }
    }
}
```

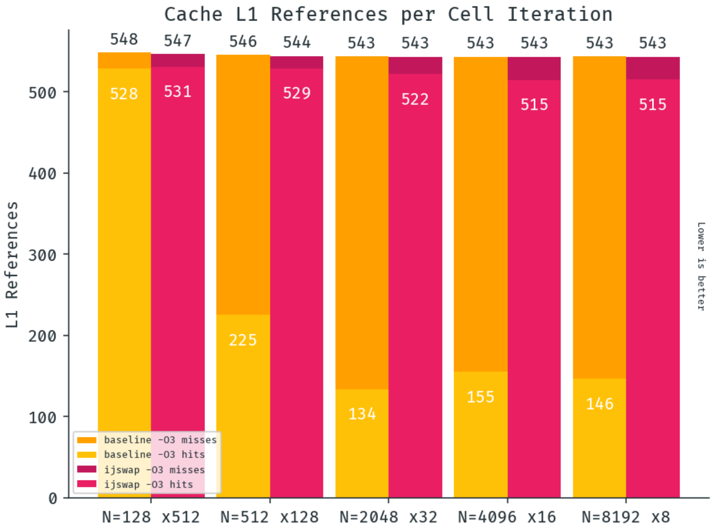
Baseline (Flags= -O3)
vs
i,j swap (Flags= -O3).

Results

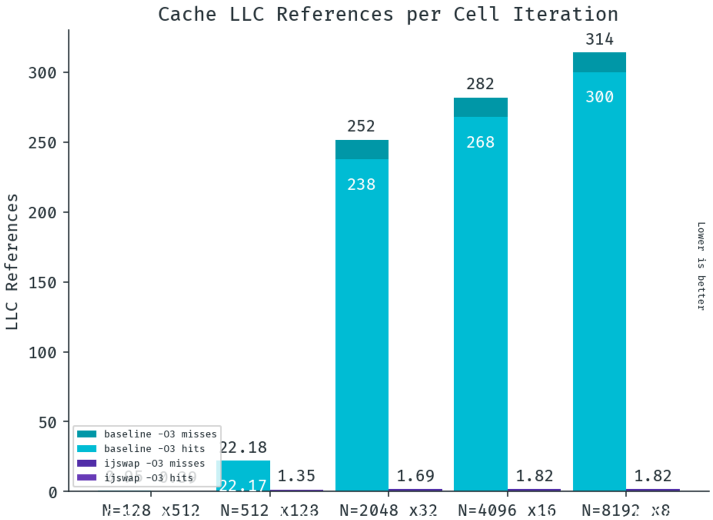


Compute Time per Cell
Nanoseconds per Cell
0
200
400
600
800
1000
1200
984
983
1302
1270
1261
1130
1115
1005
1064
984
baseline -O3
ijswap -O3
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Perf results



Perf Results

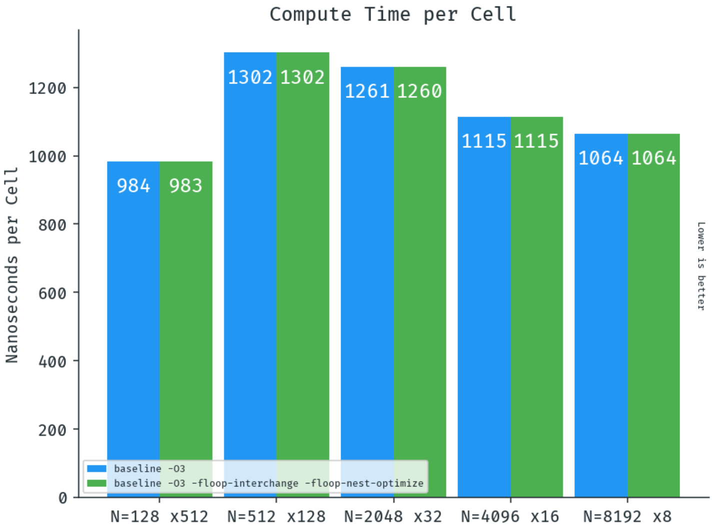


Is there a flag that interchange the loops for us? Can we do loop blocking?

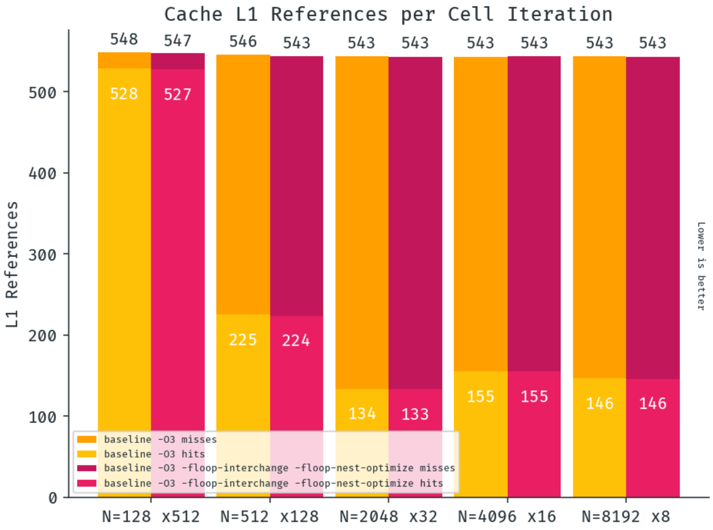
Baseline (Flags=-O3)
vs
Baseline (Flags= -O3 -floop-interchange -floop-nest-optimize).

-floop-nest-optimize => -floop-block

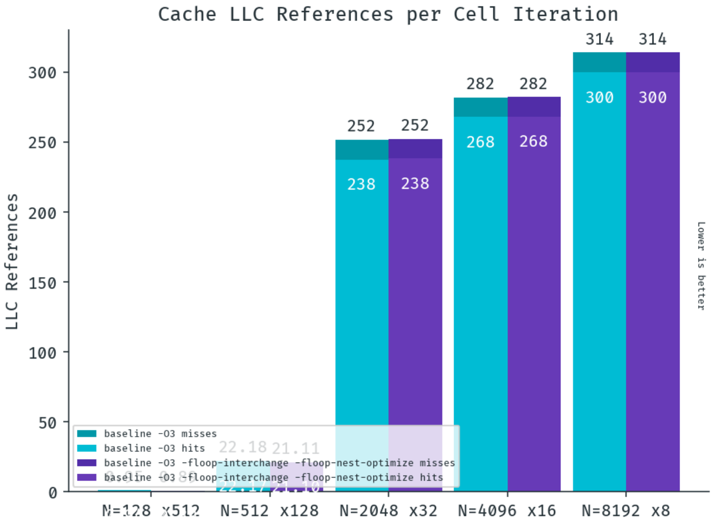
Results


Compute Time per Cell
Nanoseconds per Cell
0
200
400
600
800
1000
1200
984
983
1302
1302
1261
1260
1115
1115
1064
1064
baseline -O3
baseline -O3 -floop-interchange -floop-nest-optimize
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Perf results



Perf Results

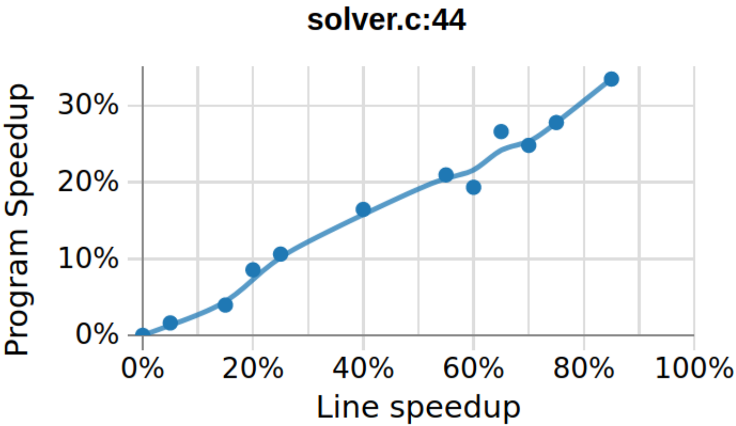


```c
float square(float* x, int n) {
    // n = 32767;
    float sum = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            sum += x[j * n + i];
    return sum;
}
```

Conclusion: i, j swap in the function **IX(i,j)** is the best approach so far.

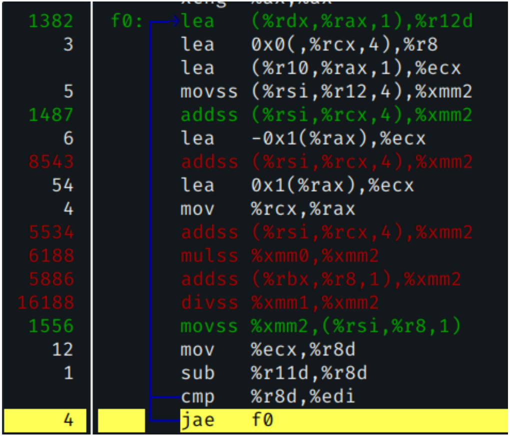
And now what? There is no an evident improvement that we can do.

The use of *cozprofiler* and *perf* lead to our second heuristic.


solver.c:44
Program Speedup
30%
20%
10%
0%
0%
20%
40%
60%
80%
100%
Line speedup

Profilers

```c
static void lin_solve(unsigned int n, boundary b, fl
{
    for (unsigned int k = 0; k < 20; k++) {
        for (unsigned int i = 1; i <= n; i++) {
            for (unsigned int j = 1; j <= n; j++) {
                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)]
                    )
                ) / c;
            }
        }
        set_bnd(n, b, x);
    }
}
```


1382 f0: lea (%rdx,%rax,1),%r12d
3 lea 0x0(,%rcx,4),%r8
lea (%r10,%rax,1),%ecx
5 movss (%rsi,%r12,4),%xmm2
1487 addss (%rsi,%rcx,4),%xmm2
6 lea -0x1(%rax),%ecx
8543 addss (%rsi,%rcx,4),%xmm2
54 lea 0x1(%rax),%ecx
4 mov %rcx,%rax
5534 addss (%rsi,%rcx,4),%xmm2
6188 mulss %xmm0,%xmm2
5886 addss (%rbx,%r8,1),%xmm2
16188 divss %xmm1,%xmm2
1556 movss %xmm2,(%rsi,%r8,1)
12 mov %ecx,%r8d
1 sub %r11d,%r8d
cmp %r8d,%edi
4 jae f0

Heuristic 2: Change c for the multiplicative inverse. (invc)

Procedure *lin_solve*.

```
// Expensive divition 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
```

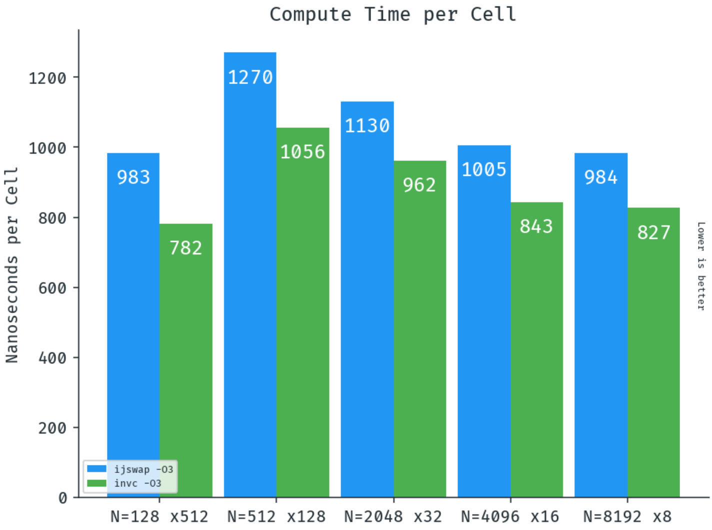
```
// Cheaper multiplication.
inv_c = 1 / c;
for k=1 to 20 do
    for i=1 to n do
        for j=1 to n do
            x[IX(i, j)] =  big_operation * inv_c;
        od
    od
od
```

Results

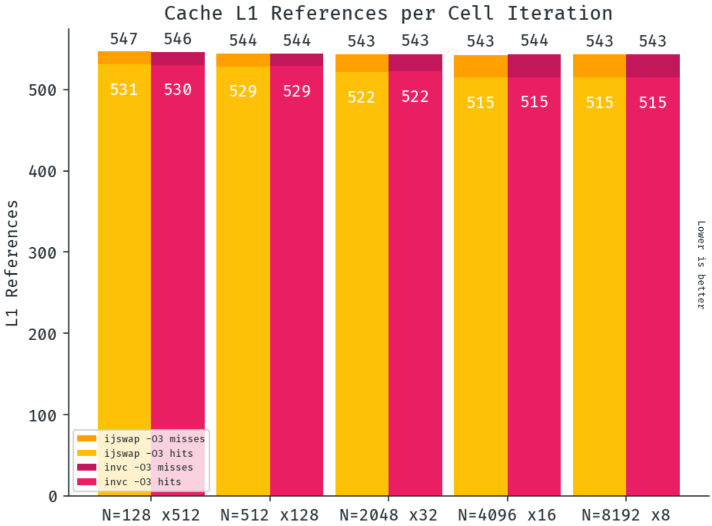
```
1450 │ 100:┌─→lea    (%rdx,%rax,1),%r12d
   5 │     │  lea    0x0(,%rcx,4),%r8
   1 │     │  lea    (%r10,%rax,1),%ecx
   7 │     │  movss  (%rsi,%r12,4),%xmm1
1538 │     │  addss  (%rsi,%rcx,4),%xmm1
   8 │     │  lea    -0x1(%rax),%ecx
8504 │     │  addss  (%rsi,%rcx,4),%xmm1
  31 │     │  lea    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
```

i,j swap (Flags= -O3)
vs
invc (Flags= -O3)

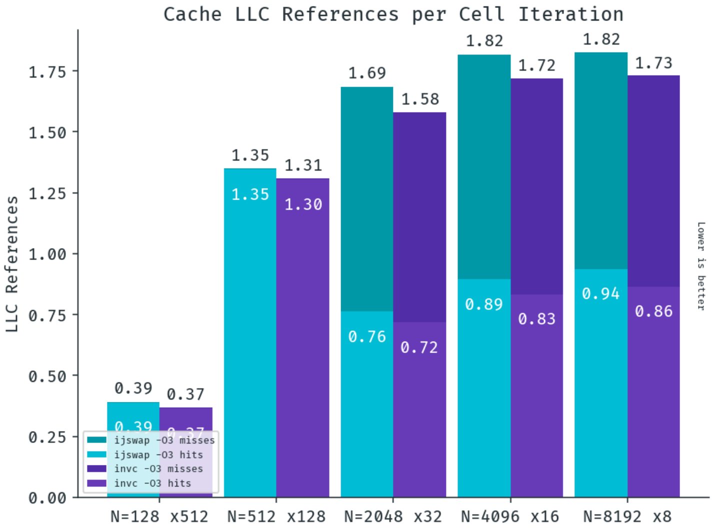
Results


Compute Time per Cell
Nanoseconds per Cell
0
200
400
600
800
1000
1200
983
782
1270
1056
1130
962
1005
843
984
827
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
ijswap -O3
invc -O3
Lower is better

Perf results



Perf Results

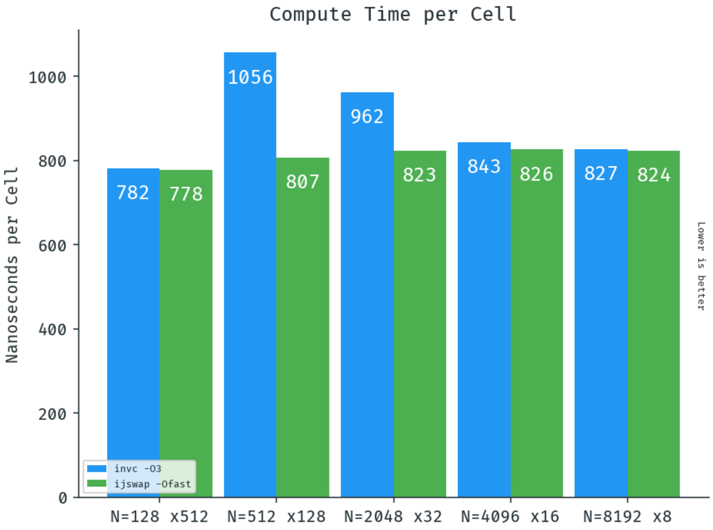


This is done by the compiler with -Ofast!

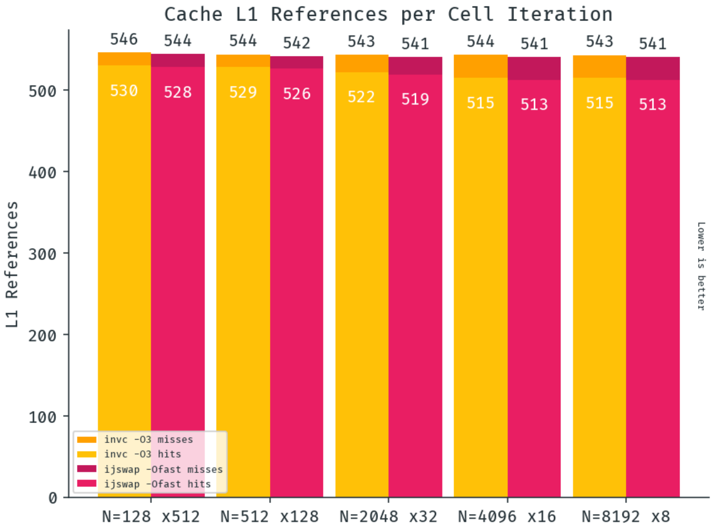
- Does -Ofast improve our heuristic?
- Can our approach avoid the use of dangerous flags?

invc (Flags= -O3)
vs
i,j swap(Flags=-Ofast)

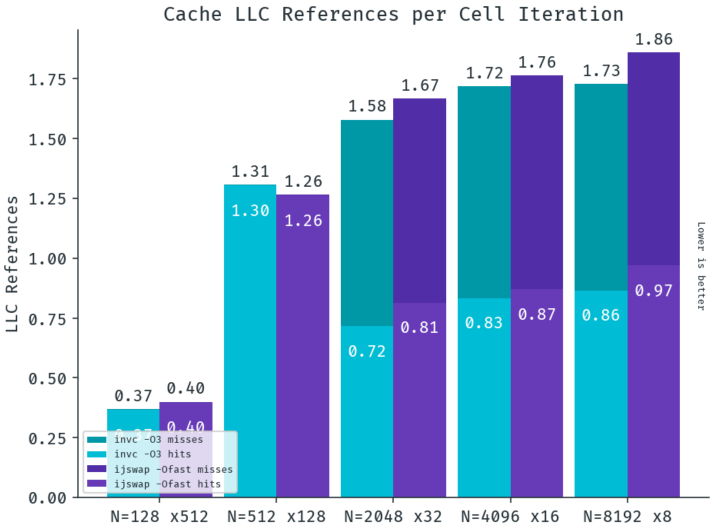
Results


Compute Time per Cell
Nanoseconds per Cell
0
200
400
600
800
1000
1056
807
962
823
843
826
827
824
782
778
invc -O3
ijswap -Ofast
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Perf results



Perf Results



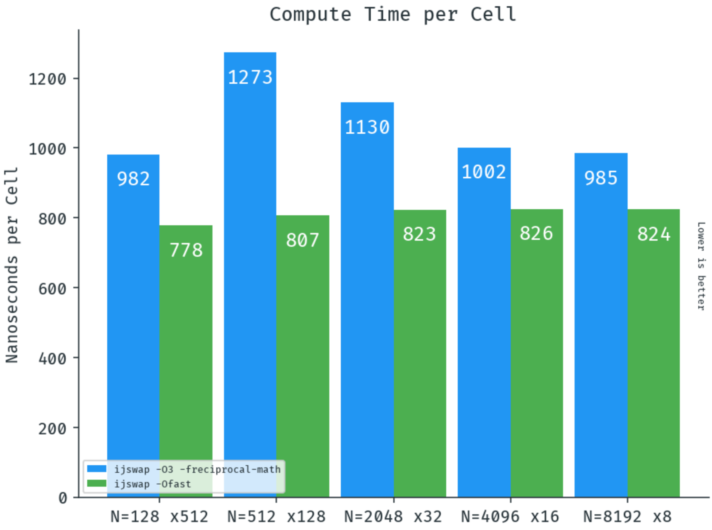
Results

Every step the performance tend to decrease with -O3 up to certain threshold.

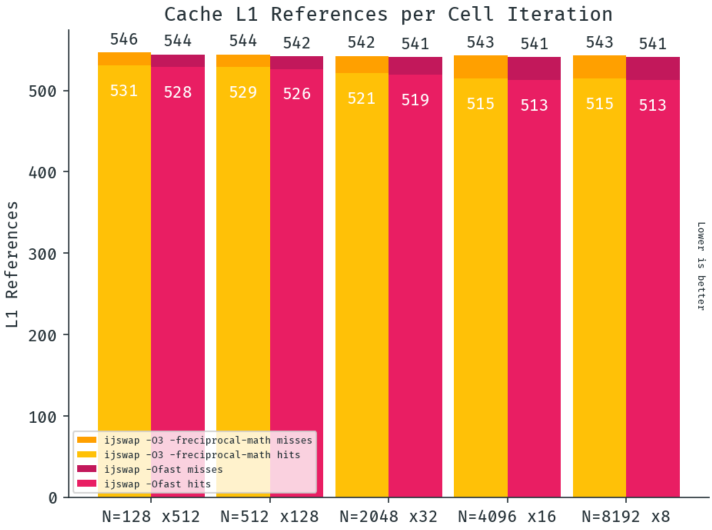
Can we use -freciprocal-math instead of -Ofast?

i,j swap (Flags= -O3 -freciprocal-math)
vs
i,j swap (Flags=-Ofast)

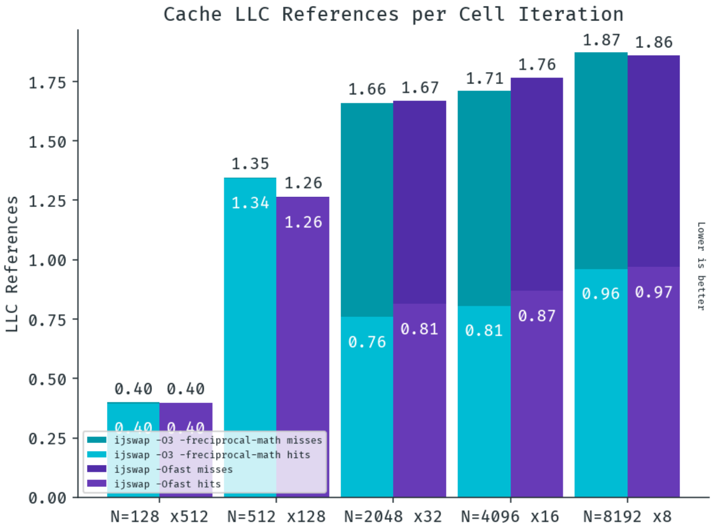
Results


Compute Time per Cell
Nanoseconds per Cell
0
200
400
600
800
1000
1200
982
778
1273
807
1130
823
1002
826
985
824
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
ijswap -O3 -freciprocal-math
ijswap -Ofast
Lower is better

Perf results



Perf Results



Conclusion:

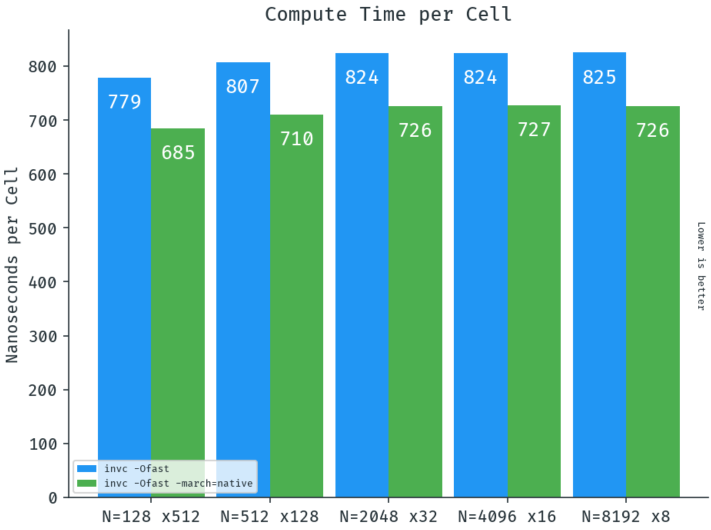
- Ofast increases the performance in comparison with -O3. Unfortunately we could not trace the reason.
- Therefore, we could not trace the flag of Ofast which does the optimization.
- Know your flags!

Therefore, we decided to try many flags:

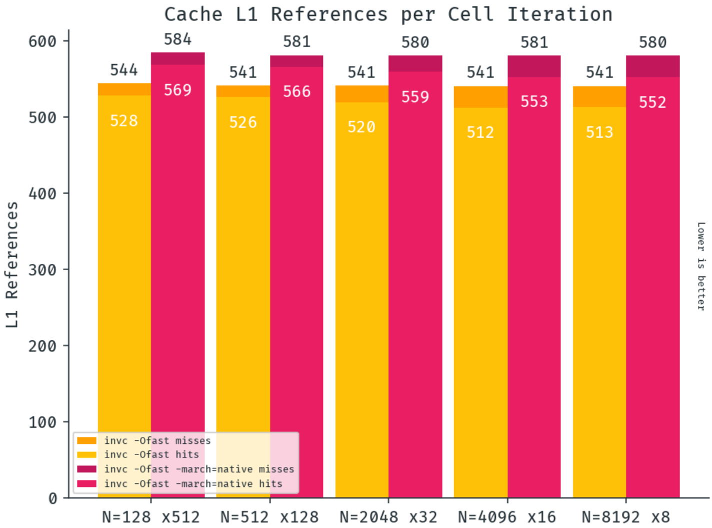
- -Ofast
- -Ofast -march=native
- -Ofast -march=native -funroll-loops
- -Ofast -march=native -funroll-loops -floop-nest-optimize
- -Ofast -march=native -funroll-loops -floop-nest-optimize -flto

invc (Flags= -Ofast)
vs
i,j swap (Flags=-Ofast -march=native)

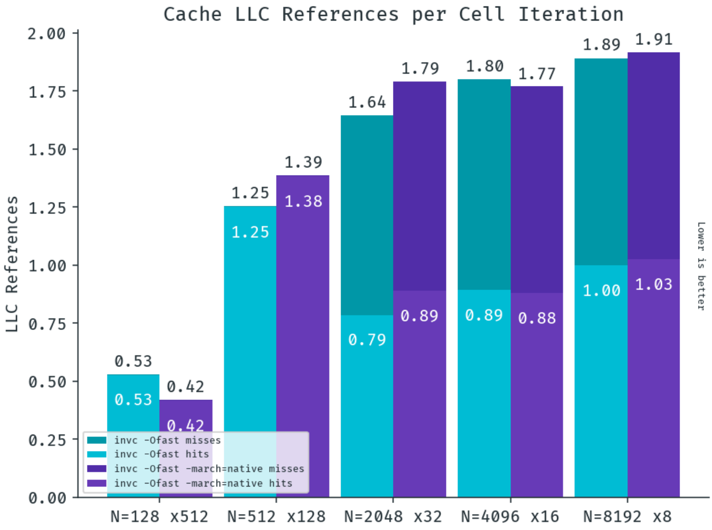
Results


Compute Time per Cell
Nanoseconds per Cell
0
100
200
300
400
500
600
700
800
779
685
807
710
824
726
824
727
825
726
invc -Ofast
invc -Ofast -march=native
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Perf results



Perf Results



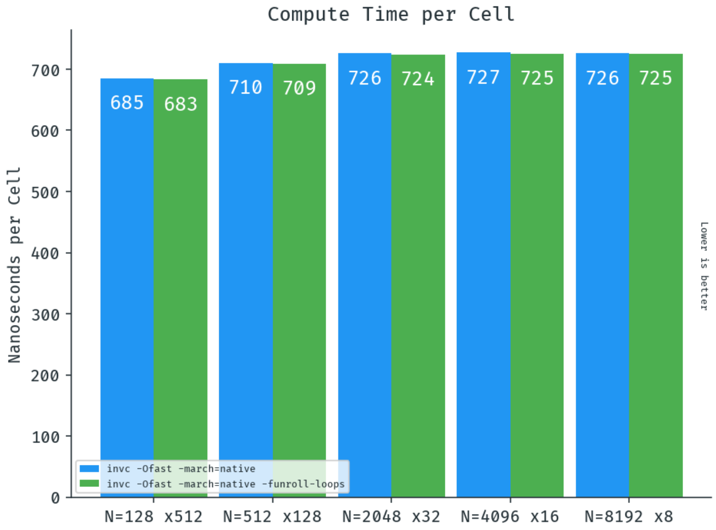
-march=native increases the performance by itself.

i,j swap (Flags=-Ofast -march=native)

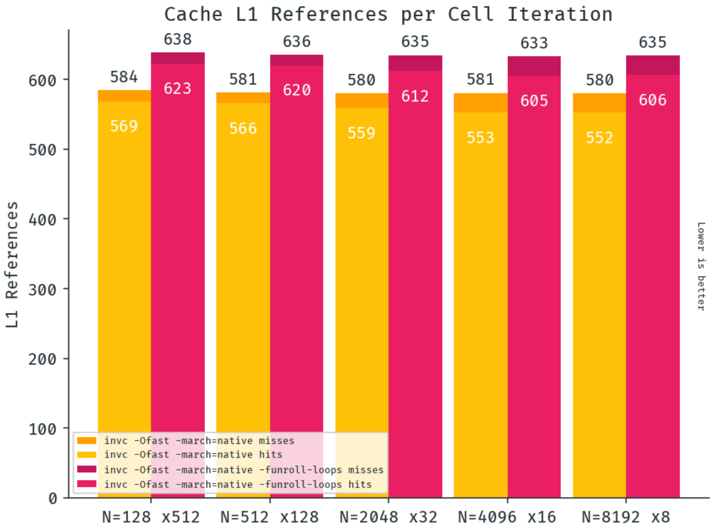
vs

invc (Flags= -Ofast -march=native -funroll-loops)

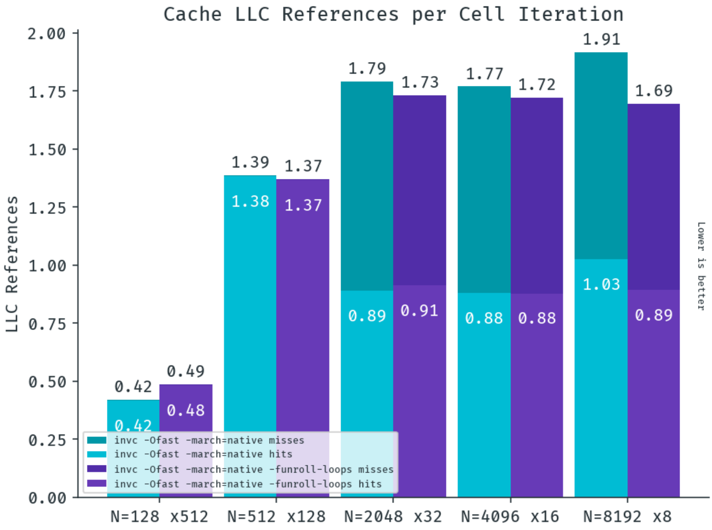
Results


Compute Time per Cell
Nanoseconds per Cell
0
100
200
300
400
500
600
700
685
683
710
709
726
724
727
725
726
725
invc -Ofast -march=native
invc -Ofast -march=native -funroll-loops
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Perf Results



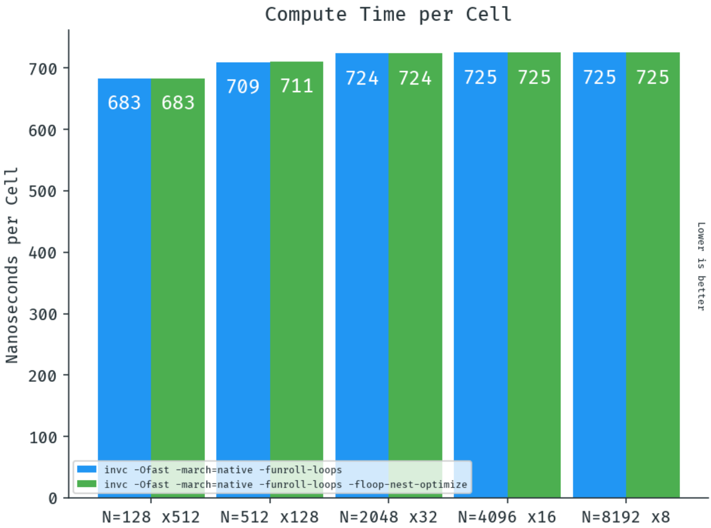
Perf results



No improvement with -funroll-loops.

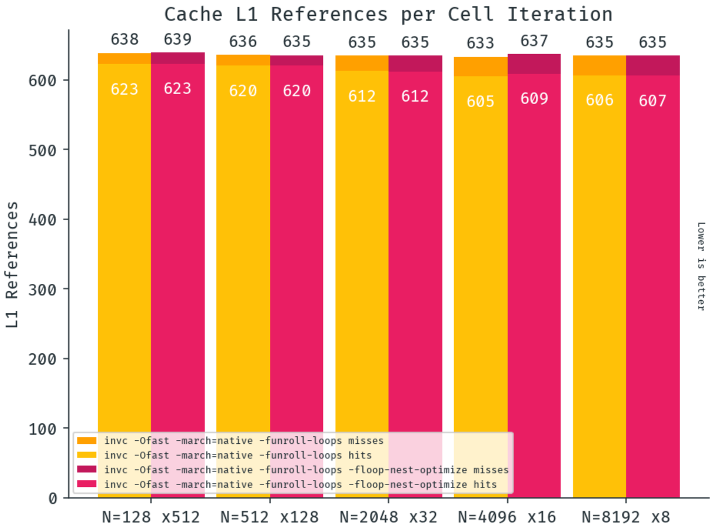
i,j swap (Flags=-Ofast -march=native -funroll-loops)
vs
invc (Flags= -Ofast -march=native -funroll-loops -floop-nest-optimize)

Results

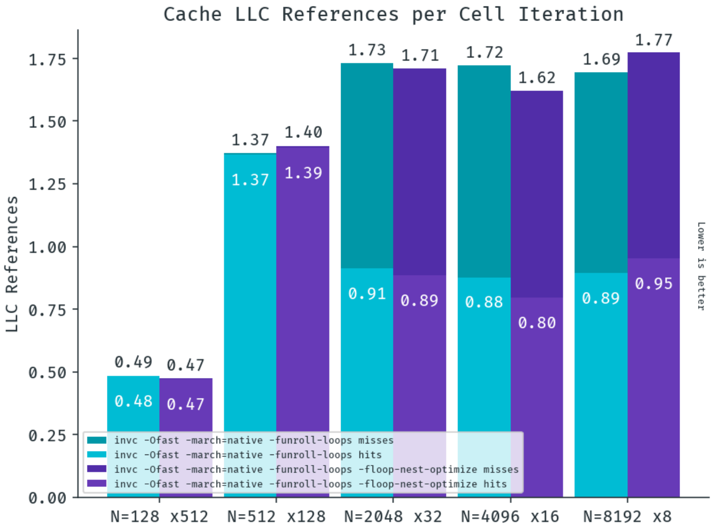


Compute Time per Cell
Nanoseconds per Cell
0
100
200
300
400
500
600
700
683
683
709
711
724
724
725
725
725
725
Lower is better
invc -Ofast -march=native -funroll-loops
invc -Ofast -march=native -funroll-loops -floop-nest-optimize
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8

Perf Results



Perf results

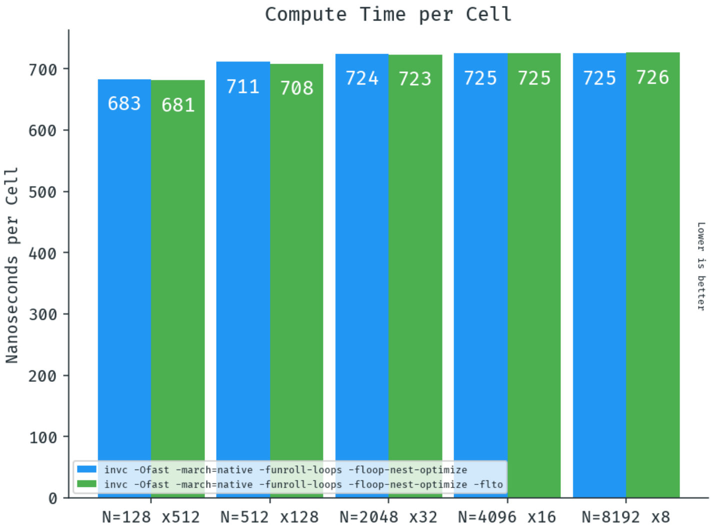


Results

No improvement with -floop-nest-optimize. Remember that -floop-nest-optimize $\Rightarrow$ -floop-block

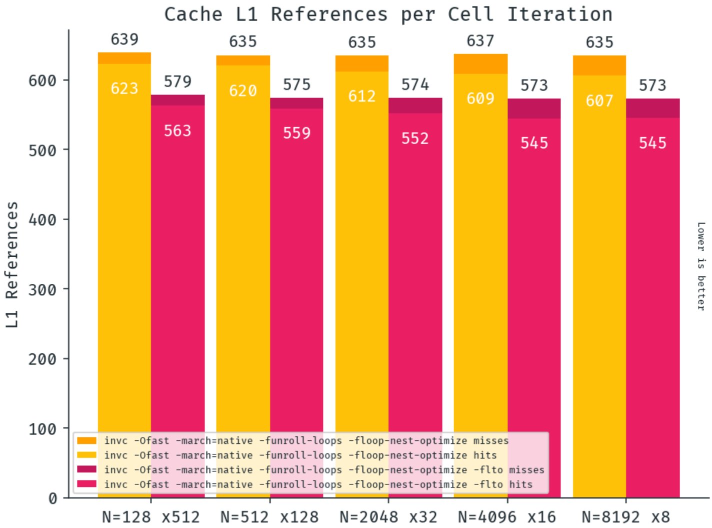
i,j swap (Flags=-Ofast -march=native -funroll-loops -floop-nest-optimize)
vs
invc (Flags= -Ofast -march=native -funroll-loops -floop-nest-optimize -flto)

Results

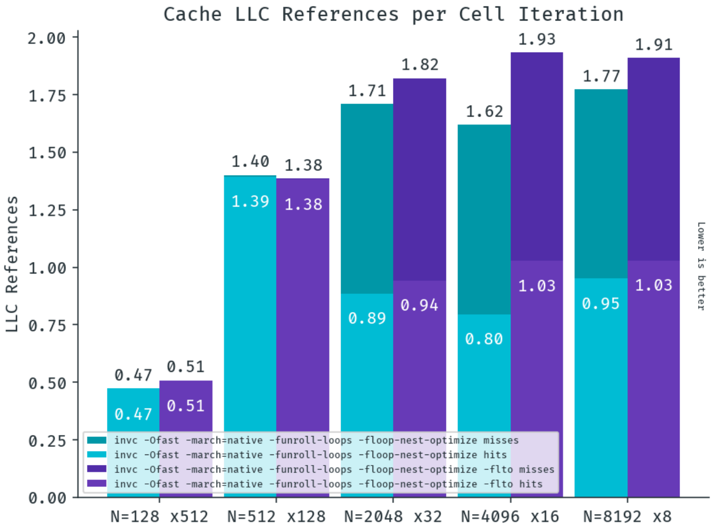


Compute Time per Cell
Nanoseconds per Cell
700
600
500
400
300
200
100
0
683
681
711
708
724
723
725
725
725
726
invc -Ofast -march=native -funroll-loops -floop-nest-optimize
invc -Ofast -march=native -funroll-loops -floop-nest-optimize -flto
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Perf Results



Perf results



Results

It seems not working! But why? These results lead to our third heuristic.

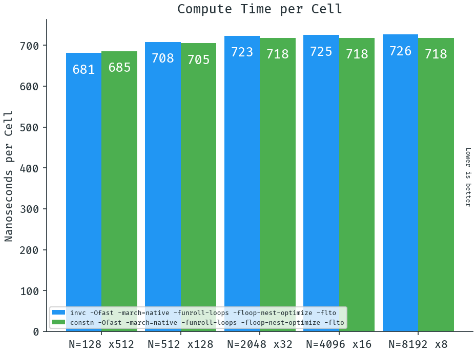
Heuristic 3: Declare N, which is the size of the grid, as const.

```
const int N = 2048;
```

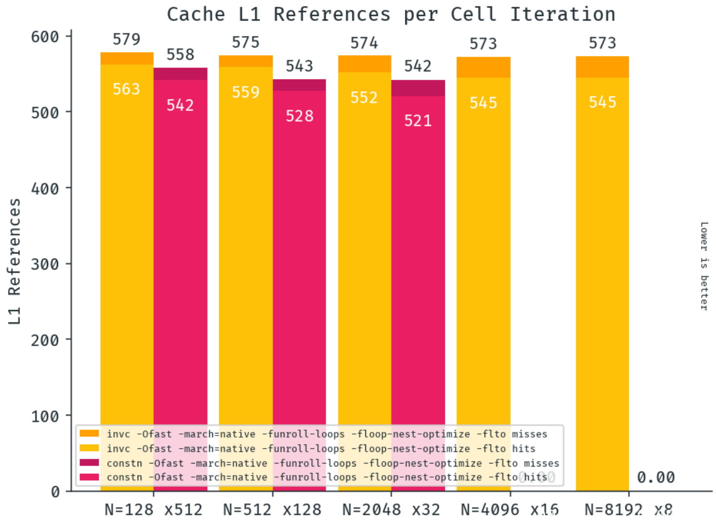
invc(Flags= -Ofast -march=native -funroll-loops
-floop-nest-optimize -flto)
vs
constn (Flags= -Ofast -march=native -funroll-loops
-floop-nest-optimize -flto)

Mateo's results

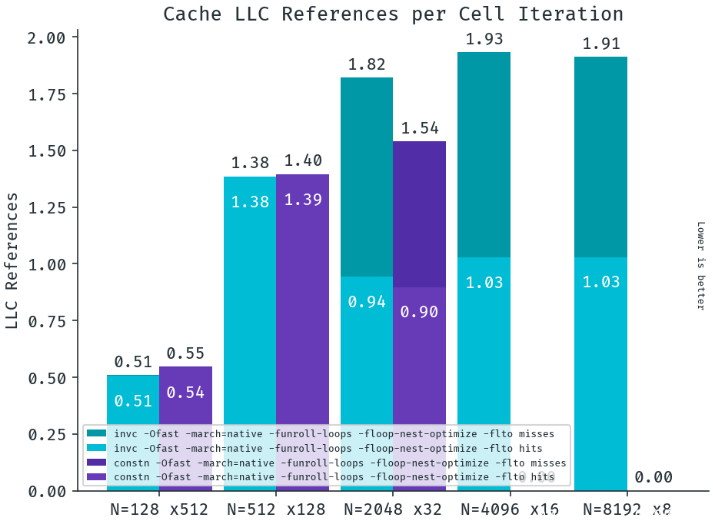
Results - zx81



Perf Results



Perf results

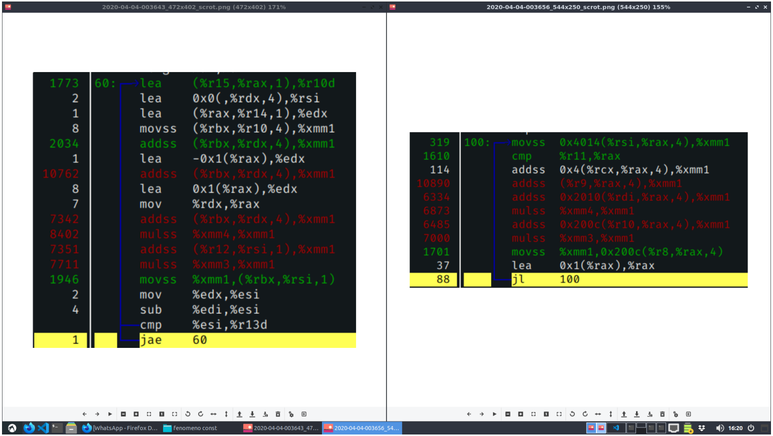


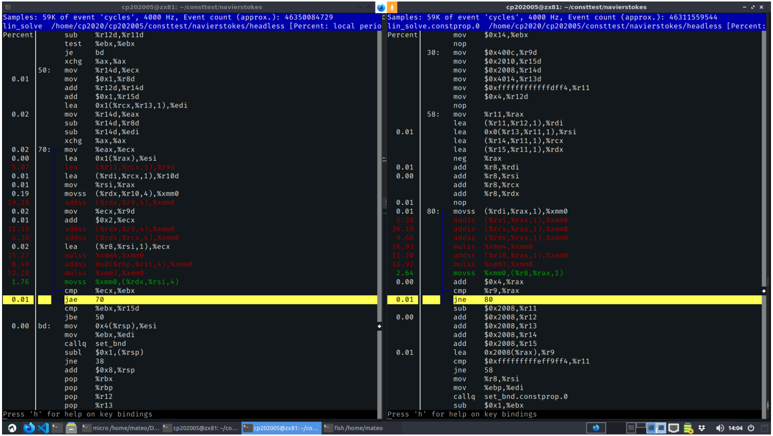
Results

Huge differences. Why?

If we take a look at the assembler with *perf*.

Results - Left: no constn, Right: constn - i7 7700HQ




cp202005@zx81: ~/consttest/navierstokes
Samples: 59K of event 'cycles', 4000 Hz, Event count (approx.): 46350084729
lin_solve /home/cp2020/cp202005/consttest/navierstokes/headless [Percent: local perio
Percent
sub %r12d,%r11d
test %ebx,%ebx
je bd
xchg %ax,%ax
50: mov %r14d,%ecx
0.01 mov $0x1,%r8d
add %r12d,%r14d
add $0x1,%r15d
lea 0x1(%rcx,%r13,1),%edi
0.02 mov %r14d,%eax
sub %r14d,%r8d
sub %r14d,%edi
xchg %ax,%ax
0.02 70: mov %eax,%ecx
0.00 lea 0x1(%rax),%esi
5.07 lea (%r11,%rcx,1),%r9d
0.01 lea (%rdi,%rcx,1),%r10d
0.01 mov %rsi,%rax
0.19 movss (%rdx,%r10,4),%xmm0
19.28 addss (%rdx,%r9,4),%xmm0
0.02 mov %ecx,%r9d
0.01 add $0x2,%ecx
31.15 addss (%rdx,%r9,4),%xmm0
6.38 addss (%rdx,%rcx,4),%xmm0
0.02 lea (%r8,%rsi,1),%ecx
15.27 mulss %xmm4,%xmm0
8.49 addss 0x0(%rbp,%rsi,4),%xmm0
12.28 mulss %xmm3,%xmm0
1.76 movss %xmm0,(%rdx,%rsi,4)
cmp %ecx,%ebx
0.01 jae 70
cmp %ebx,%r15d
jbe 50
0.00 bd: mov 0x4(%rsp),%esi
mov %ebx,%edi
callq set_bnd
subl $0x1,(%rsp)
jne 38
add $0x8,%rsp
pop %rbx
pop %rbp
pop %r12
pop %r13
Press 'h' for help on key bindings
cp202005@zx81: ~/consttest/navierstokes
Samples: 59K of event 'cycles', 4000 Hz, Event count (approx.): 46311559544
lin_solve.constprop.0 /home/cp2020/cp202005/consttest/navierstokes/headless [Percent:
Percent
mov $0x14,%ebx
nop
30: mov $0x400c,%r9d
mov $0x2010,%r15d
mov $0x2008,%r14d
mov $0x4014,%r13d
mov $0xfffffffffffffdff4,%r11
mov $0x4,%r12d
nop
58: mov %r11,%rax
lea (%r11,%r12,1),%rdi
0.01 lea 0x0(%r13,%r11,1),%rsi
lea (%r14,%r11,1),%rcx
lea (%r15,%r11,1),%rdx
neg %rax
0.01 add %r8,%rdi
0.00 add %r8,%rsi
add %r8,%rcx
add %r8,%rdx
0.01 nop
0.01 80: movss (%rdi,%rax,1),%xmm0
6.38 addss (%rsi,%rax,1),%xmm0
36.19 addss (%rcx,%rax,1),%xmm0
9.66 addss (%rdx,%rax,1),%xmm0
16.93 mulss %xmm4,%xmm0
11.20 addss (%r10,%rax,1),%xmm0
16.92 mulss %xmm3,%xmm0
2.64 movss %xmm0,(%r8,%rax,1)
0.00 add $0x4,%rax
cmp %r9,%rax
0.01 jne 80
sub $0x2008,%r11
0.00 add $0x2008,%r12
add $0x2008,%r13
add $0x2008,%r14
add $0x2008,%r15
0.01 lea 0x2008(%rax),%r9
cmp $0xffffffffffeff9ff4,%r11
jne 58
mov %r8,%rsi
mov %ebp,%edi
callq set_bnd.constprop.0
sub $0x1,%ebx
Press 'h' for help on key bindings
micro /home/mateo/D...
cp202005@zx81: ~/co...
cp202005@zx81: ~/co...
fish /home/mateo
14:04

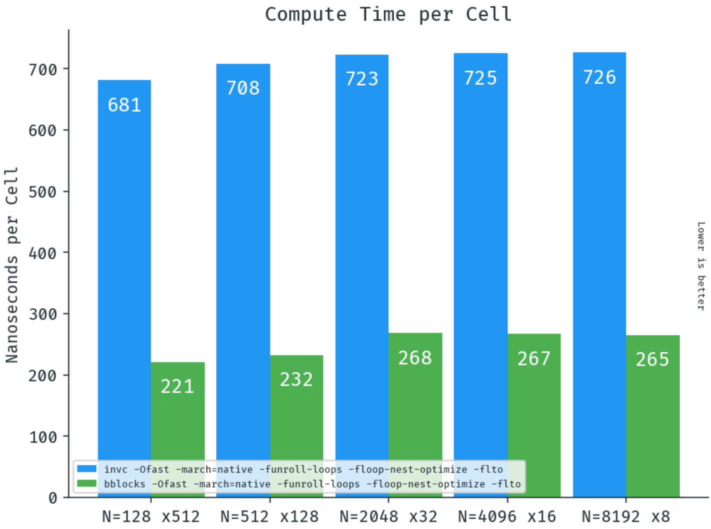
More on lin_solve:

- Gauss-Seidel? $x_{i,j}^{n+1} = x_{i,j}^{0} + \frac{a(x_{i-1,j}^{n+1} + x_{i,j-1}^{n+1} + x_{i+1,j}^{n} + x_{i,j+1}^{n})}{1+4a}$
- Jacobi?
- Ad-hoc idea

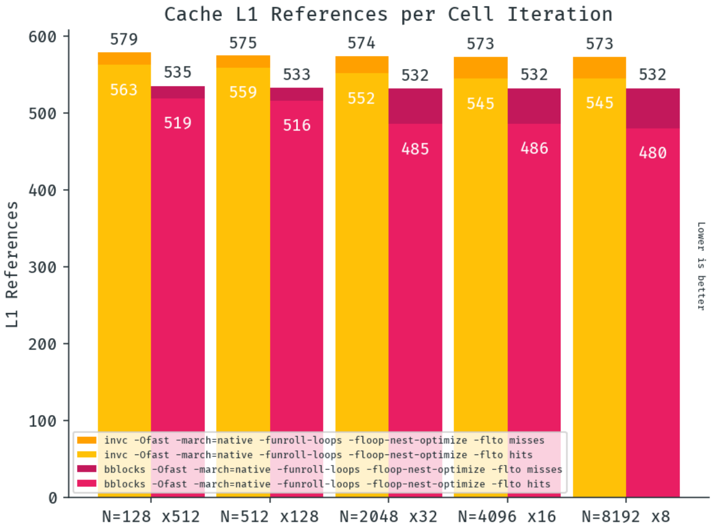
Heuristic 4: Basic blocking

```
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);
}
```

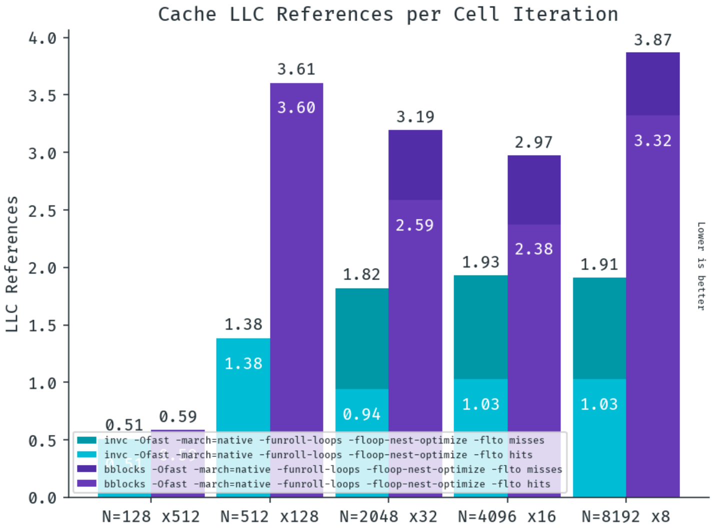
Results


Compute Time per Cell
Nanoseconds per Cell
0
100
200
300
400
500
600
700
681
221
708
232
723
268
725
267
726
265
invc -Ofast -march=native -funroll-loops -floop-nest-optimize -flto
bblocks -Ofast -march=native -funroll-loops -floop-nest-optimize -flto
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Perf Results



Perf results



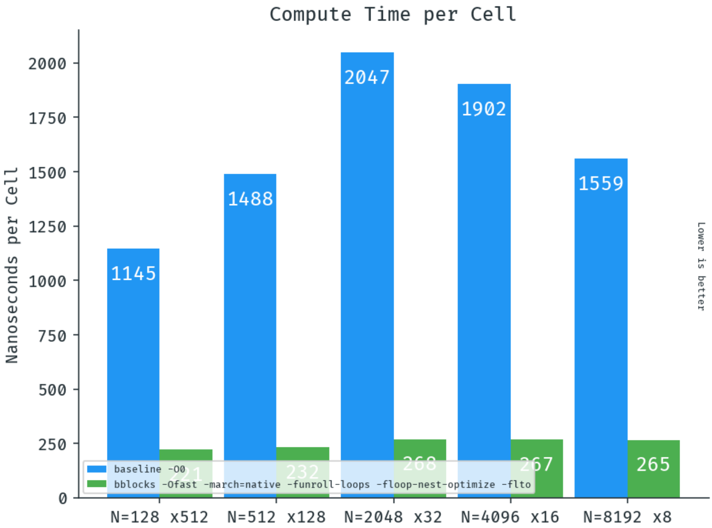
How much have we improved?

Baseline (Flags -O0)

vs

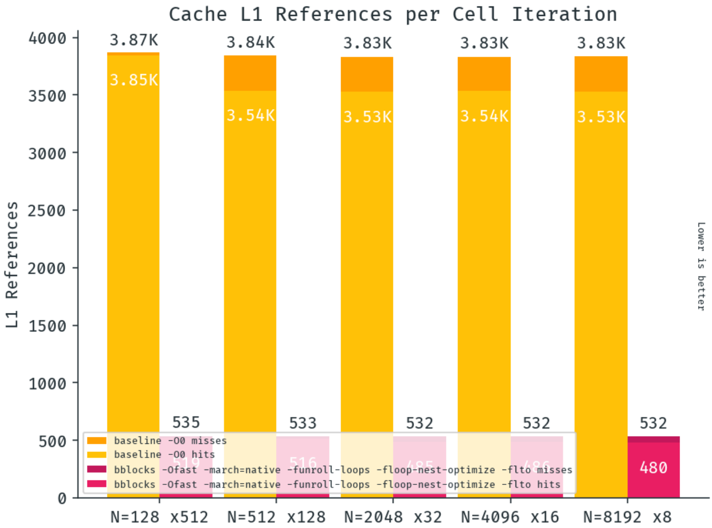
bblock (Flags = -Ofast_-march=native_-funroll-loops_-floop-nest-optimize_-flto)

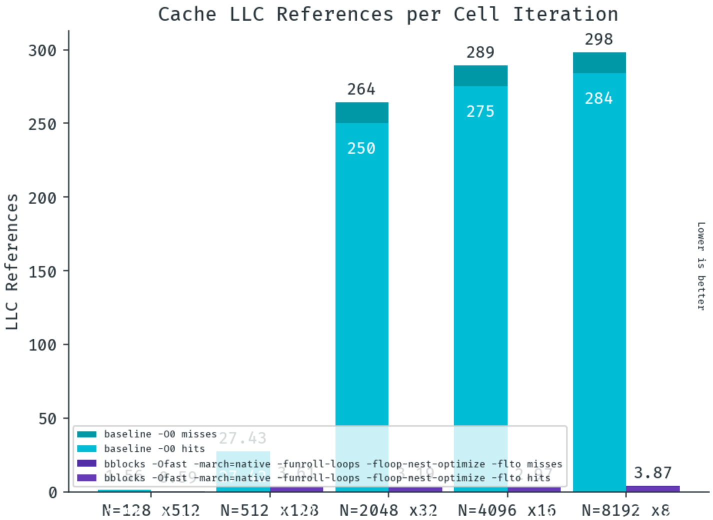
Results



Compute Time per Cell
Nanoseconds per Cell
2000
1750
1500
1250
1000
750
500
250
0
1145
1488
2047
1902
1559
221
232
268
267
265
baseline -O0
bblocks -Ofast -march=native -funroll-loops -floop-nest-optimize -flto
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Perf Results





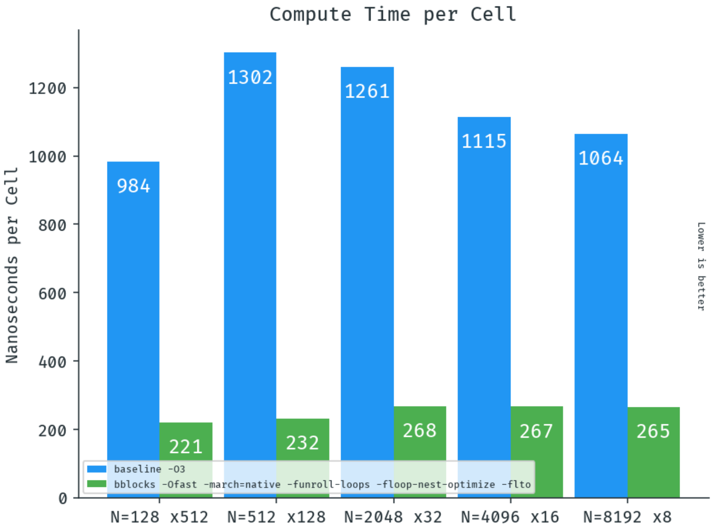
Cache LLC References per Cell Iteration
LLC References
300
250
200
150
100
50
0
298
289
264
284
275
250
27.43
3.87
baseline -O0 misses
baseline -O0 hits
bblocks -Ofast -march=native -funroll-loops -floop-nest-optimize -flto misses
bblocks -Ofast -march=native -funroll-loops -floop-nest-optimize -flto hits
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Baseline (Flags -O3)

vs

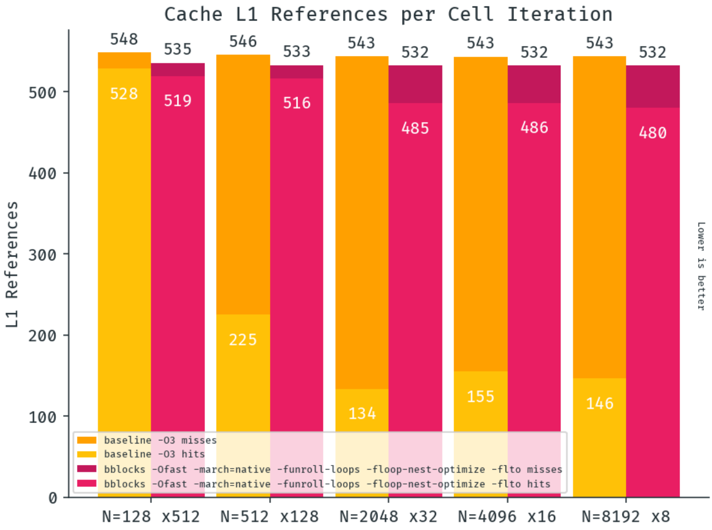
bblock (Flags = -Ofast_-march=native_-funroll-loops_-floop-nest-optimize_-flto)

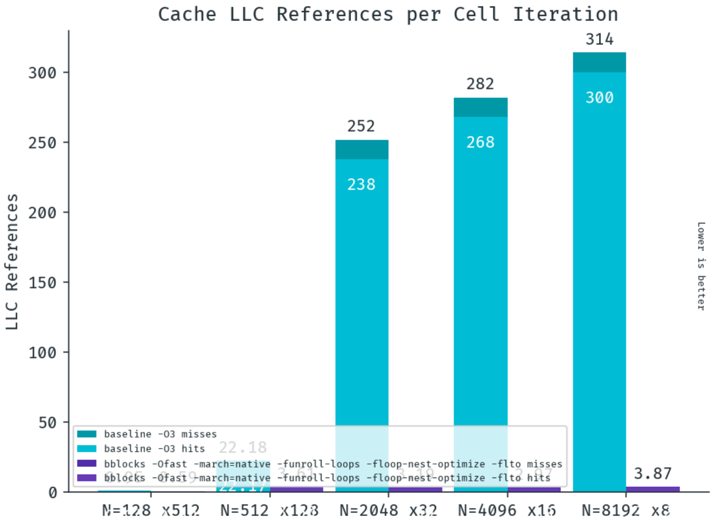
Results



Compute Time per Cell
Nanoseconds per Cell
0
200
400
600
800
1000
1200
984
1302
1261
1115
1064
221
232
268
267
265
baseline -O3
bblocks -Ofast -march=native -funroll-loops -floop-nest-optimize -flto
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8
Lower is better

Perf Results




Cache LLC References per Cell Iteration
LLC References
300
250
200
150
100
50
0
314
300
282
268
252
238
22.18
3.87
Lower is better
baseline -O3 misses
baseline -O3 hits
bblocks -Ofast -march=native -funroll-loops -floop-nest-optimize -flto misses
bblocks -Ofast -march=native -funroll-loops -floop-nest-optimize -flto hits
N=128 x512
N=512 x128
N=2048 x32
N=4096 x16
N=8192 x8

Things that never were

- llvm-mca
- autofdo
- cachegrind
- flto crashes
- Array of Structures