Optimization
Loops & Prefetching

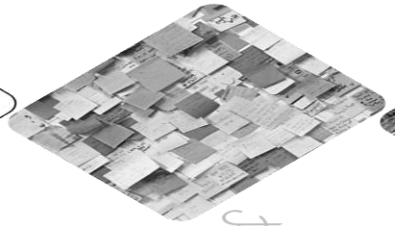
Luca Tornatore - I.N.A.F.

"**Foundation of HPC" course**

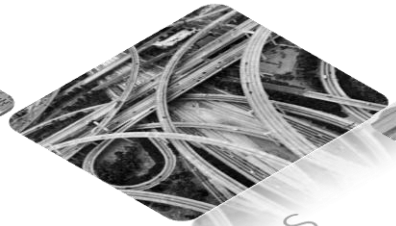
DATA SCIENCE &
SCIENTIFIC COMPUTING
2020-2021 @ Università di Trieste

Outline

First things first

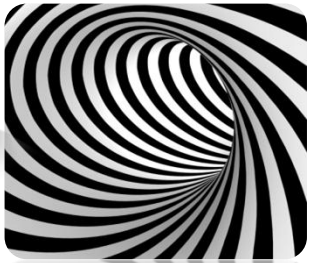


Cache & Memory

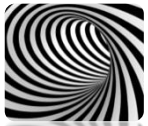


Branches

Pipelines

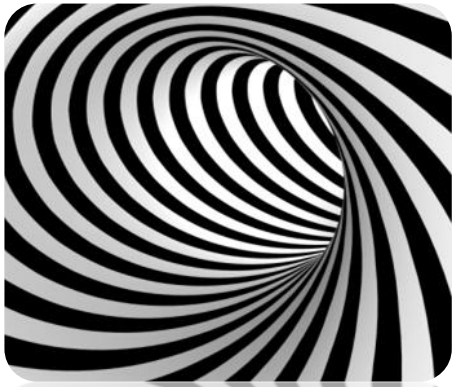


Loops



Outline

Avoid the avoidable inefficiencies

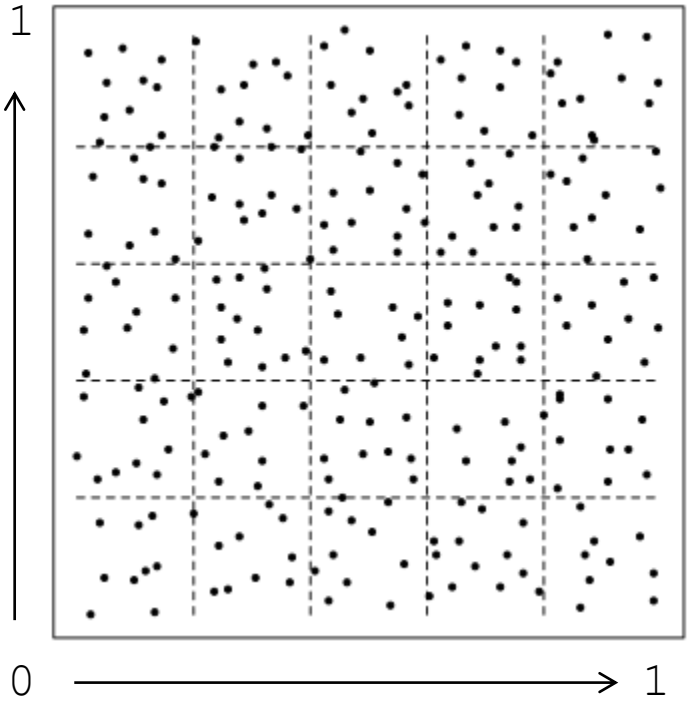


Loops techniques

Prefetching

Let's suppose that

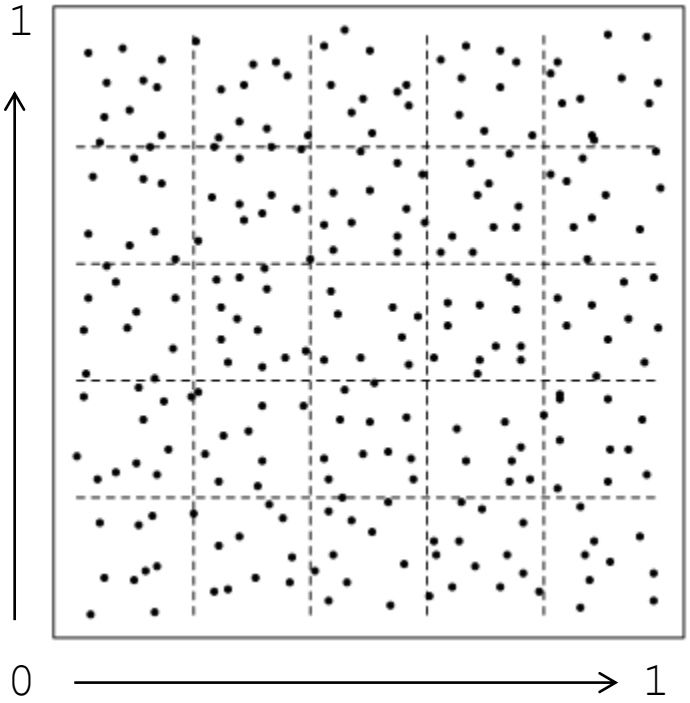
1) we have a distribution of random data points on a 2D plane which we subdivide in sub-regions using a grid.



Let's suppose that

1) we have a distribution of random data points on a 2D plane which we subdivide in sub-regions using a grid.

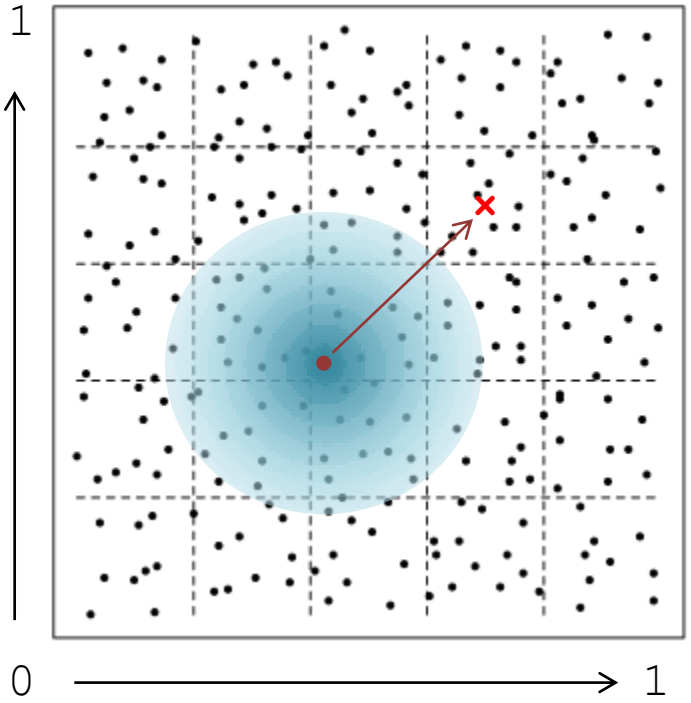
2) for each point *p*, we want to select all the grid cells whose center is closer to *p* than a given radius *r*, and to perform some operations accordingly to our search result.



Let's suppose that

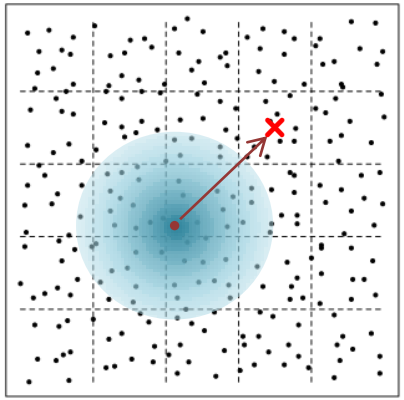
1) we have a distribution of random data points on a 2D plane which we subdivide in sub-regions using a grid.

2) for each point *p*, we want to select all the grid cells whose center is closer to *p* than a given radius *r*, and to perform some operations accordingly to our search result.



We may consider to use a nested loop like this one →

Is there anything you would change in this loop?



```
for(p = 0; p < Np; p++)

    for(i = 0; i < Ng; i++)
      for(j = 0; j < Ng; j++)
        for(k = 0; k < Ng; k++)
          {
            dist = sqrt(
                    pow(x[p] - (double)i/Ng - half_size, 2) +
                    pow(y[p] - (double)j/Ng - half_size, 2) +
                    pow(z[p] - (double)k/Ng - half_size, 2));

                  if(dist < R)
                  do something;
          }
```

(1) Avoid expensive function calls

Some function calls are particularly expensive. Those include, among others, `sqrt()`, …

Try to avoid them *if possible*.

```
for(p = 0; p < Np; p++)

  for(i = 0; i < Ng; i++)
    for(j = 0; j < Ng; j++)
      for(k = 0; k < Ng; k++)
        {
          dist2 = pow(x[p] - (double)i/Ng - half_size, 2) +
                  pow(y[p] - (double)j/Ng - half_size, 2) +
                  pow(z[p] - (double)k/Ng - half_size, 2));

                if(dist2 < R2)
                do something;
        }
```

(1) Avoid expensive function calls

Some function calls are particularly expensive. Those include, among others, `sqrt()`, `pow()`, ...

Try to avoid them *if possible*.

```
for(p = 0; p < Np; p++)

  for(i = 0; i < Ng; i++)
    for(j = 0; j < Ng; j++)
      for(k = 0; k < Ng; k++)
        {
          dx = x[p] - (double)i/Ng - half_size;
          dy = y[p] - (double)j/Ng - half_size;
          dz = z[p] - (double)k/Ng - half_size;

          dist2 = dx*dx + dy*dy + dz*dz;
          if(dist2 < R2)
             do something;
        }
```

(1) Avoid expensive function calls

```
for(p = 0; p < Np; p++)
```

Some function calls are particularly expensive. Those include, among others, `sqrt()`, `pow()`, `floating point division`, ..

Try to avoid them *if possible*.

```
for(i = 0; i < Ng; i++)
  for(j = 0; j < Ng; j++)
    for(k = 0; k < Ng; k++)
      {
        dx = x[p] - (double)i * Ng_inv - half_size;
        dy = y[p] - (double)j * Ng_inv - half_size;
        dz = z[p] - (double)k * Ng_inv - half_size;

        dist2 = dx*dx + dy*dy + dz*dz;
        if(dist2 < R2)
           do something with sqrt(dist2);
      }
```

(1) Avoid expensive function calls

`(double)<i,j,k> * Ng_inv + half_size`

was performed N^3+N^2+N times, always returning the same values.
Hoisting would save
$N(N^2+N^1+1)$ **`mul`**, **`add`** and **`mem`** accesses.

You can do better pre-computing the relevant values:

```
double ijk[Ng];
for(i = 0; i < Ng; i++)
   ijk[i] = i * Ng_inv + half_size
```

```
for(p = 0; p < Np; p++)

    for(i = 0; i < Ng; i++)
      for(j = 0; j < Ng; j++)
        for(k = 0; k < Ng; k++)
          {
            dx = x[p] - (double)i * Ng_inv - half_size;
            dy = y[p] - (double)j * Ng_inv - half_size;
            dz = z[p] - (double)k * Ng_inv - half_size;

            dist2 = dx*dx + dy*dy + dz*dz;
            if(dist2 < R2)
               do something with sqrt(dist2);
          }
```

(2) *Hoisting* of expressions

```
for(i = 0; i < Ng; i++) {
   dx2 = x[p] - (double)i * Ng_inv - half_size;
   dx2 = dx2*dx2;

      for(j = 0; j < Ng; j++) {
         dy2 = y[p] - (double)j * Ng_inv - half_size;
         dy2 = dy2*dy2;
         dist2_xy = dx2 + dy2;

         for(k = 0; k < Ng; k++) {
            dz = z[p] - (double)k * Ng_inv - half_size;
            dist2 = dist2_xy + dz*dz;
            if(dist2 < Rmax2)
              do something with sqrt(dist2);  } } }
```

`(double)<i,j,k> * Ng_inv + half_size`

was performed N^3+N^2+N times,
always returning the same values.
Hoisting would save
$N(N^2+N^1+1)$ **`mul`**, **`add`** and **`mem`** accesses.

(2) *Hoisting* of expressions

You could do even better by pre-computing the relevant values:

```
double ijk[Ng];
for(i = 0; i < Ng; i++)
   ijk[i] = i * Ng_inv + half_size
```

```
for(i = 0; i < Ng; i++) {
   dx2 = x[p] - Ng_inv[i] - half_size;
   dx2 = dx2*dx2;

   for(j = 0; j < Ng; j++) {
      dy2 = y[p] - Ng_inv[j] - half_size;
      dist2_xy = dx2 + dy2*dy2;

      for(k = 0; k < Ng; k++) {
         dz = z[p] - Ng_inv[k] - half_size;
         dist2 = dist2_xy + dz*dz;
         if(dist2 < Rmax2)
           do something with sqrt(dist2);  } } }
```

(3) Clarify the variables' scope

All these variables are very local, there's no need for them to have a wider scope.

That will help you in writing the code, and *may* help the compiler in optimizing the stack and perhaps the registers usage.

```
for(int i = 0; i < Ng; i++) {
   double dx2 = x[p] - (double)i * Ng_inv - half_size;
   dx2 *= dx2;

      for(j = 0; j < Ng; j++) {
        double dy2 = y[p] - (double)j * Ng_inv - half_size;
        double dist2_xy = dx2 + dy2*dy2;

         for(k = 0; k < Ng; k++) {
            double dz = z[p] - (double)k * Ng_inv - half_size;
            double dist2 = dist2_xy + dz*dz;

            if(dist2 < Rmax2)
              do something with sqrt(dist2);  } } }
```

(4) Suggest what is important

These variables are often calculated and reused subsequently.

Keeping a register dedicated to them may be useful.

Note: *this is a suggestion, the compiler, after analyzing the code, may decide differently*

```
double register Ng_inv = 1.0 / Ng;
for(int i = 0; i < Ng; i++) {
   double dx2 = x[p] - (double)i * Ng_inv - half_size;
   dx2 *= dx2;

      for(j = 0; j < Ng; j++) {
         double dy2 = y[p] - (double)j * Ng_inv - half_size;
         dy2 *= dy2;
         double register dist2_xy = dx2 + dy2;

         for(k = 0; k < Ng; k++) {
            double register dz = z[p] - (double)k * Ng_inv - …;
            double register dist2 = dist2_xy + dz*dz;

            if(dist2 < Rmax2)
              do something with sqrt(dist2);  } } }
```

CAVEAT !!

Do not suppose that your compiler is *always* able to re-arrange calculations – like avoiding expensive calls or using mathematically-equivalent more convenient expressions – all the time. It may be able to do that for ***integer*** calculations but it will not do it for ***floating-point*** calculations.

The reason is simple, and it is related to the fact that on a digital system the math is *not* always as it is on the blackboard:

Integer math (+ and ×) in 2's complement is commutative and associative.

Floating point math (+ and ×) is commutative between 2 operands is *never* associatve.

| CAVEAT !!

In fact, if you study, as you should, the “what every computer scientist should know about floating-point” paper (find it in the `materials/` folder on `github`), you discover that if **a**, **b**, and **c** are FP numbers,

$$(a + b) + c \neq a + (b + c)$$

due to the very nature of floating-point representation in a digital system (with a finite number of digits).
The issue is related to the limited number of digits available to represent the number which, in turn, limits the precision.

| CAVEAT !!

Let's suppose that we have 3 digits of precision for the mantissa. For the sake of simplicity, we consider a base 10 (so every single digit ranges in [0..9]).

Then the following hold:

```
1.00 + 0.01  = 1.01
1.00 + 0.001 = 1.00
```

The last is true because, although we can represent 0.001 (it is 1.00 with a -3 exponent) the summation of 1.00 and 0.001 is beyond our precision: 1.001 would require *4* digits. As a consequence, we are not able to distinguish it from 1.00.

CAVEAT !!

Then again:

These sum up to 0.01

```
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
1.00  =
1.01
```

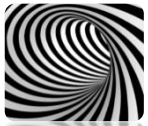
```
1.00 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 +
0.001 =
1.00
```

Each of these ops results in 1.00

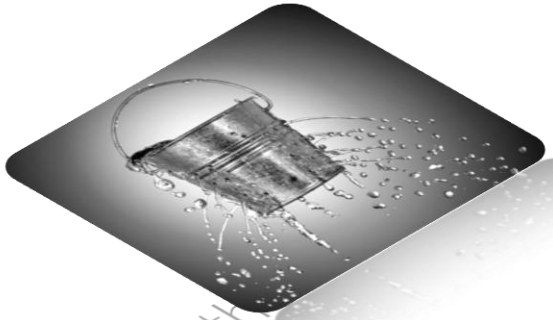
So, the compiler is NOT free to reshuffle the order of floating-point operations,

..even if a mathematically-equivalent formulation, different than the one you coded, would be more performant.

see the materials in the github folder `./materials/kahan_summation`



Outline



Avoid the avoidable inefficiencies

Loops techniques

Prefetching

Optimizing cache access in loops

Loop classification

$$A_I = \frac{f(n)}{n}$$

Arithmetic Intensity: the **ratio** between the **number of performed operations** and the **amount of the data**.

1. O(N) / O(N)
   optimization potential limited
2. $O(N^2)$ / $O(N^2)$
   some more opportunities for opt.
3. $O(N^3)$ / $O(N^2)$
   significant optimization potential

Cache access in loops: *O(N)/O(N)*

Example

1-level loops: Scalar products, vector additions, sparse matrix-vector multiplication

Inevitably memory-bound for very large *N*; in general, improvements come from *avoiding unnecessary* operations and/or *repeated memory accesses*, and increasing data reuse

[check the room for loops fusion]

O(N) / O(N)

```
for(int j=0; j<2; j++)
  A[i] = B[i] × C[i]

for(int j=0; j<2; j++)
  Q[i] = B[i] + D[i]
```

➔

```
for(int j=0; j<2; j++)
  {
    A[i] = B[i] × C[i]
    Q[i] = B[i] + D[i]
  }
```

Loop fusion: *in the version on the right, B is recalled from memory only once.*

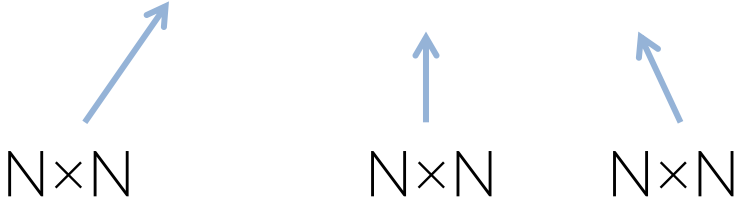
Cache access in loops: $O(N^2)/O(N^2)$

Example

2-levels loops: dense matrix-vector mul, matrix transpos., matrix add, …

Improvements comes again from increasing *data reuse*, exploiting *locality* and *avoiding unnecessary* operations and memory accesses.

```
for(int i=0; i < N; i++)
   for(int j=0; j<N; j++)
      C[i] += A[i][j] * B[j];
```



→ $3 \times N^2$ memory accesses

Cache access in loops: $O(N^2)/O(N^2)$

```
for(int i=0; i < N; i++)
   for(int j=0; j<N; j++)
      C[i] += A[i][j] * B[j];
```

→

Step 1: Avoid unecessary loads /stores

```
for(int i=0; i < N; i++) {
   c_temp = C[i];
   for(int j=0; j < N; j++)
      c_temp += A[i][j] * B[j];
   C[i] = c_temp; }
```

Now it is clearer for the compiler that `C[i]` need to be loaded and stored only 1 time

→ $2\times N^2+N$ memory accesses

Cache access in loops: $O(N^2)/O(N^2)$

```
for(int i=0; i < N; i++)
   for(int j=0; j<N; j++)
      C[i] += A[i][j] * B[j];
```

→ $N^2 \times (1+1/m) + N$

Step 2:
Unroll outern loop and *fuse* in the inner loop; there is potential for *vectorisation*.

```
for(int i=0; i < N; i += m)
   for(j = 0; j < N; j++){
      b_temp  = B[j];
      C[i]   += A[i][j] * b_temp;
      C[i+1] += A[i+1][j] * b_temp;
      …
      C[i+m] += A[i+m][j] * b_temp; }
```

N×N/m

N×N

N

Note: unrolling and register spill

Using a too large m in the previous example while the target CPU does not have enough registers to keep all the needed operands results in a "code bloating".
In this case, the CPU has to spill registers' content to cache and viceversa, slowing down the computation.

→ learn to inspect the compiler's *log*

A too much involute and obscure loop body may hamper the compiler to effectively perform *unroll & jam* optimizations targeted to the CPU it runs on.

→ hand code effort to clarify the code
→ hints / directives to the compiler
(directives are generally not portable across different compilers)

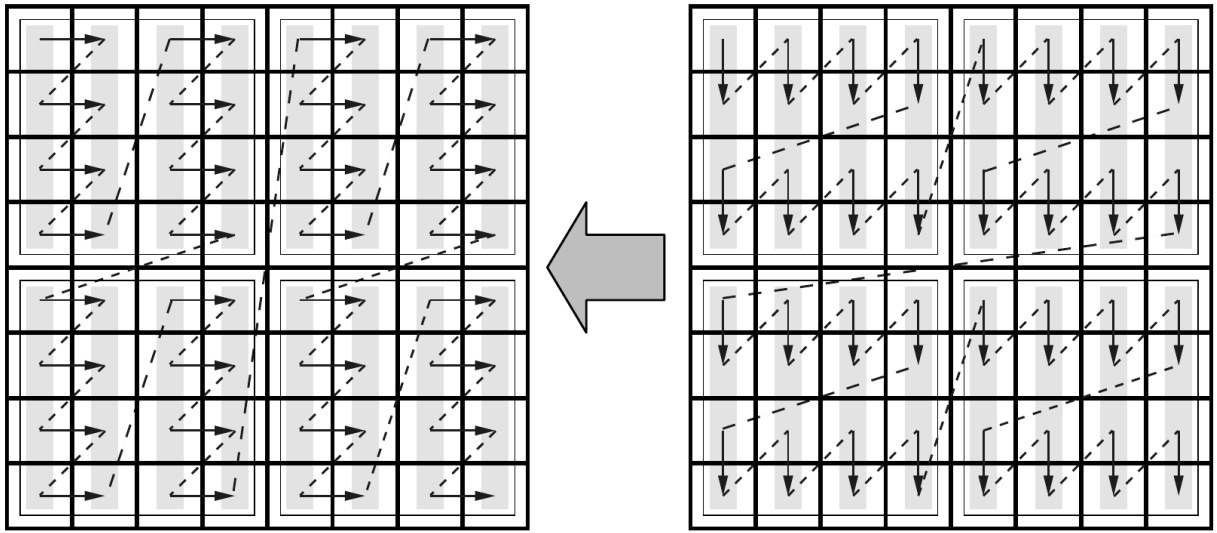
Cache access in loops: $O(N^2)/O(N^2)$

Sometimes no magic wand can cure the fact that you have to access N^2 memory locations.
For instance: in matrix transpose you have to access all the source matrix and all the destination matrix once.

Unroll & Jam strategy can bring benefits as long as the cache can hold N lines.
An L_C-way unrolling is too much aggressive and may easily result in register pressure.

Loop tailing (or blocking) is a good strategy that does not save memory loads but increase dramatically the cache hit ratio

Cache access in loops: $O(N^2)/O(N^2)$



Step 3:
Fully exploit locality of referenced data; cut TLB misses by accessing 2D arrays by blocks

Image taken from: introduction to HPC for scientists and engineers

Loop unrolling

Loop unrolling is a fundamental code transformation which usually helps significantly in improving your code performance:

- It reduces the loop overhead (counter update, branching)
- It exposes *critical data path* and dependencies
- It helps in exploiting ILP, especially in case of memory aliasing

We have already seen this technique in the examples from past lecture, although we did not really focus on it.
Now, let's understand it with some more detail.

Loop unrolling

Let's examine this simple *reduction*:

```
for ( int i=0; j<N; i++ )
  S += A[i];
```

-O0

```
.L3:
# reduction.c:41:     acc = acc OP array[ii];
        mov     rax, QWORD PTR -24[rbp] # ii
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -48[rbp] # array
        add     rax, rdx
        movsd   xmm0, QWORD PTR [rax]
        movsd   QWORD PTR -56[rbp], xmm0
        fld     QWORD PTR -56[rbp]
# reduction.c:41:     acc = acc OP array[ii];
        fld     TBYTE PTR -16[rbp]      # acc
        faddp   st(1), st
        fstp    TBYTE PTR -16[rbp]      # acc
# reduction.c:40:   for ( uLint ii = 1; ii < N; ii++ )
        add     QWORD PTR -24[rbp], 1   # ii,
.L2:
# reduction.c:40:   for ( uLint ii = 1; ii < N; ii++ )
        mov     rax, QWORD PTR -24[rbp] # ii
        cmp     rax, QWORD PTR -40[rbp] # N
        jb      .L3
```

-O3 -march=native

```
.L3:
# reduction.c:41:     acc = acc OP array[ii];
        fadd    QWORD PTR [rax] # array
        add     rax, 8          #
# reduction.c:40:   for ( uLint ii = 1; ii < N; ii++ )
        cmp     rdx, rax        # N
        jne     .L3

        ret
```

With optimization turned on the compiler manages much better the loop overhead, but does not optimize the FP ops in any way.

Loop unrolling

If we compile *exactly the same code* but using **`int`** as data type instead of **`double`**, we obtain a quite different result:

`-O0`

```
.L3:
# reduction.c:43:     acc = acc OP array[ii];
        movq    -8(%rbp), %rax  # ii
        leaq    0(,%rax,4), %rdx
        movq    -32(%rbp), %rax # array
        addq    %rdx, %rax
        movl    (%rax), %eax
        movl    %eax, %eax
# reduction.c:43:     acc = acc OP array[ii];
        addq    %rax, -16(%rbp)
# reduction.c:42:   for ( uLint ii = 1; ii < N; ii++ )
        addq    $1, -8(%rbp)    #, ii
.L2:
# reduction.c:42:   for ( uLint ii = 1; ii < N; ii++ )
        movq    -8(%rbp), %rax  # ii
        cmpq    -24(%rbp), %rax # N
        jb      .L3
```

`-O3 -march=native`

```
.L4:
# reduction.c:43:     acc = acc OP array[ii];
        vmovdqu 4(%rax), %ymm0
        addq    $32, %rax
        vpmovzxdq       %xmm0, %ymm1
        vextracti128    $0x1, %ymm0, %xmm0
        vpmovzxdq       %xmm0, %ymm0
# reduction.c:43:     acc = acc OP array[ii];
        vpaddq  %ymm0, %ymm1, %ymm0
        vpaddq  %ymm0, %ymm2, %ymm2
        cmpq    %rdx, %rax
        jne     .L4
```

Now the compiler opts for the complete *vectorization* of the loop, with a very strong impact on performances !!

Loop unrolling

Why does the compiler choose to optimize the loop when the data are of type **`int`** and it does not when the data are of type **`double`** ?

It is due to the fact that, as we have seen the compiler is NOT free to restructure the code that deals with floating-point numbers.
Since the math with floating point is not associative, changing the exact order of the operations in your code may – from what the compiler may judge at compile time – change the correctness of the calculations at run time.

For instance, in the example that we have considered, changing the order of the operations obviously impacts on the result. From the point of view of the compiler, you may have chosen a given workflow exactly because *you know* that it is the most correct with respect to the data it will apply to!

Loop unrolling

Then, we are left with the responsibility of optimizing this simple code.
Since we are traversing it continuously in natural memory order, the cache is not an issue.
Our aim is to re-structure the code so that the compiler could exploit the CPU's ILP.

```
for ( int i=0; j<N; i++ )
  S += A[i];
```

Step 1: unrolling

Our first attempt is to reduce the loop overhead and expose some parallelism among the data by explicitly processing 2 elements per iteration.

```
for ( int i=0; i<N; i++ )
  S = S OP A[i];
```

```
for ( int i=0; i<N-2; i+=2 )
  S = (S OP A[i]) OP A[i+1] ;
```

Step 1: unrolling 2×1

Our first attempt is to reduce the loop overhead and expose some parallelism among the data by explicitly processing 2 elements per iteration.

```
for ( int i=0; i<N; i++ )
  S = S OP A[i];
```

```
for ( int i=0; i<N-2; i+=2 )
  S = (S OP A[i]) OP A[i+1] ;
```

Note: when unrolling, you always have to care about the final iterations that would be left behind. A common way to do it for un unrolling factor U (usually U ranges in [2..16] of a loop with N iterations is:

```
for ( int i = 0; i < N-U; i += U )
   iteration_ops;

for ( int i = N-U; i < N; i++ )
   iteration_ops;
```

Step 1: unrolling 2×1

The compiler generates (*) the following assembly code:

```
.L17:
        vmovupd ymm1, YMMWORD PTR [rax]
        add     rax, 32
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd xmm2, xmm1, xmm1
        vextractf128    xmm1, ymm1, 0x1
        vaddsd  xmm0, xmm0, xmm2
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd xmm1, xmm1, xmm1
        vaddsd  xmm0, xmm0, xmm1
.LVL18:
        cmp     rax, rcx
        jne     .L17
```

load 32B (4 double) starting from *i*th element.
ymm1 has 256bits.

rax contains the address to the ith element of the array, [rax] means "the address pointed by rax"

registers' reshuffle to move each double at the begin, in order to use **vaddsd**

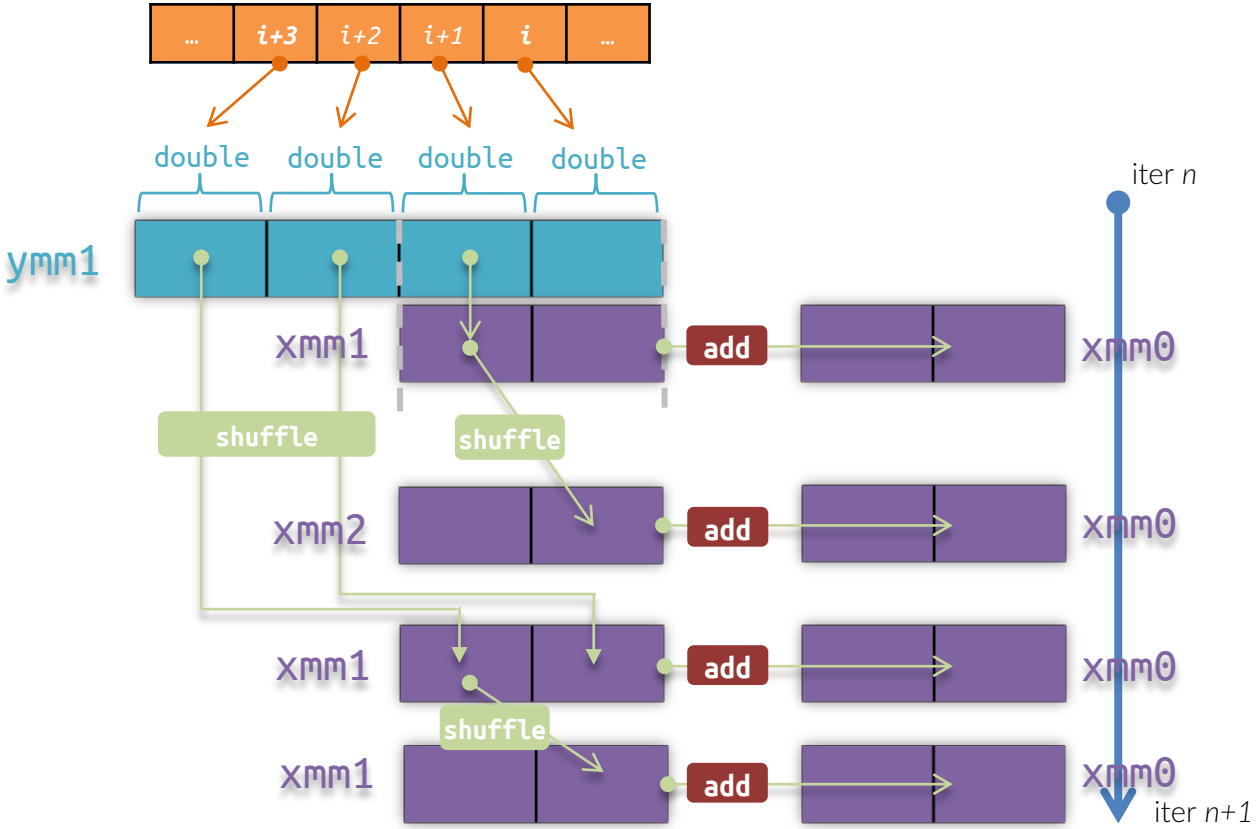
all these instructions sum with, and store in, **xmm0**

(*) in the following we will always comment the code generated with `-O3 -march=native`

Step 1: unrolling 2×1

In a (hopefully) simpler view, the scheme of what happens is the following →

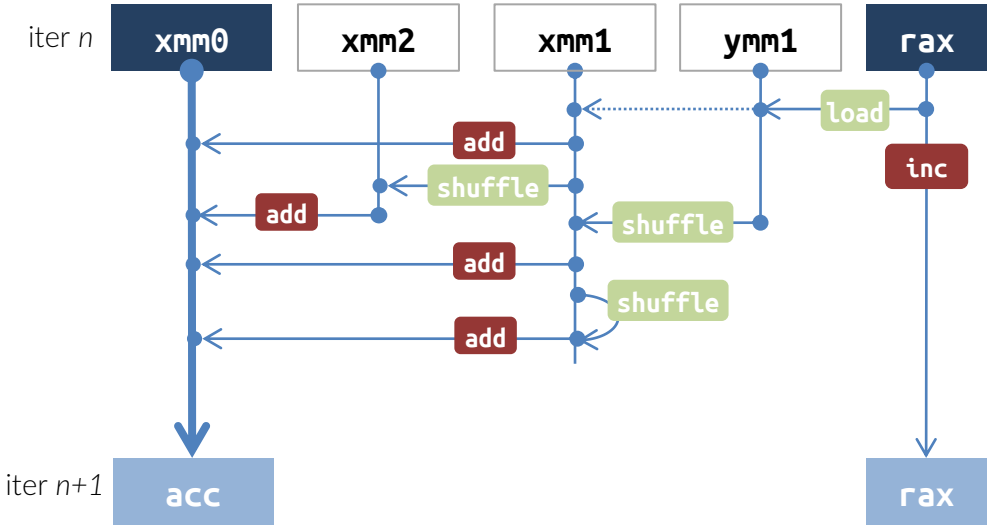
```
.L17:
        vmovupd ymm1, YMMWORD PTR [rax]
        add     rax, 32
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd xmm2, xmm1, xmm1
        vextractf128    xmm1, ymm1, 0x1
        vaddsd  xmm0, xmm0, xmm2
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd xmm1, xmm1, xmm1
        vaddsd  xmm0, xmm0, xmm1
.LVL18:
        cmp     rax, rcx
        jne     .L17
```



Step 1: unrolling 2×1

Then, we have the following abstraction:
(arrows indicate a dependency)

```
.L17:
        vmovupd ymm1, YMMWORD PTR [rax]
        add     rax, 32
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd xmm2, xmm1, xmm1
        vextractf128    xmm1, ymm1, 0x1
        vaddsd  xmm0, xmm0, xmm2
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd xmm1, xmm1, xmm1
        vaddsd  xmm0, xmm0, xmm1
.LVL18:
        cmp     rax, rcx
        jne     .L17
```



`xmm0` carries a loop dependency because its value is to be used in the next iteration (that is true for `rax` too, but its latency is smaller than that of FP operations)
It forms a ***critical path*** that limits the efficiency.

Step 2: unrolling 2×1+ reshuffle

Let's explore what happens if we apport a semantic change to our code, just using the fact that our `OP` is associative.

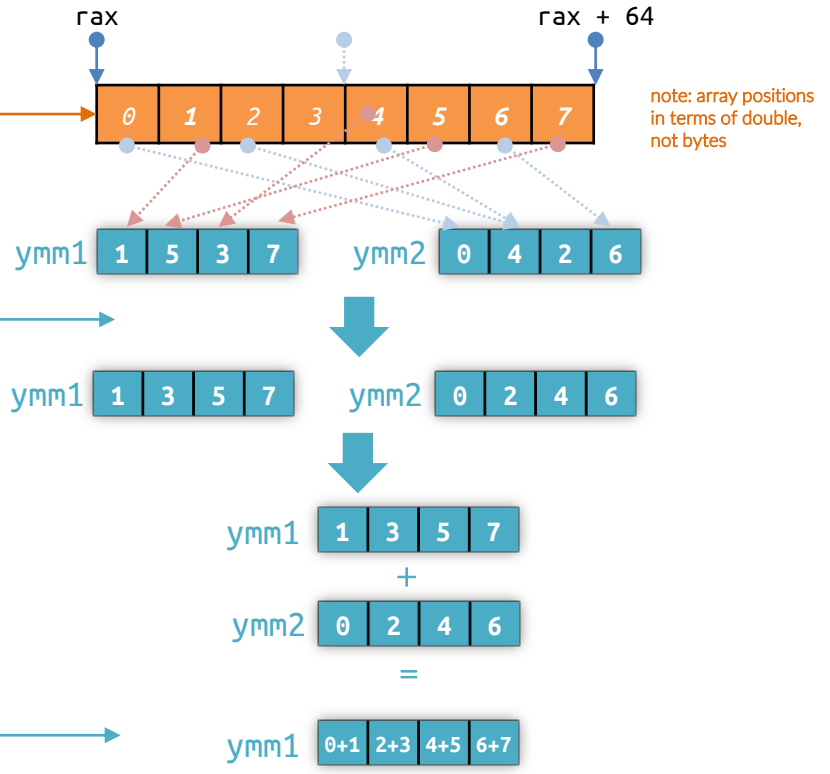
```
for ( int i=0; i<N-2; i+=2 )
  S = (S OP A[i]) OP A[i+1] ;
```

```
for ( int i=0; i<N-2; i+=2 )
  S = S OP (A[i] OP A[i+1]) ;
```

Step 2: unrolling 2×1+ reshuffle

```
for ( int i=0; i<N-2; i+=2 )
  S = S OP (A[i] OP A[i+1]) ;
```

```
.L33:
        vmovupd ymm4, YMMWORD PTR [rax]
        add     rax, 64
        vunpcklpd       ymm1, ymm4, YMMWORD PTR -32[rax]
        vunpckhpd       ymm2, ymm4, YMMWORD PTR -32[rax]
        vpermpd ymm1, ymm1, 216
        vpermpd ymm2, ymm2, 216
        vaddpd  ymm1, ymm1, ymm2
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd       xmm3, xmm1, xmm1
        vextractf128    xmm1, ymm1, 0x1
        vaddsd  xmm0, xmm0, xmm3
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd       xmm1, xmm1, xmm1
        vaddsd  xmm0, xmm0, xmm1
```



8 doubles are processed per iteration; thanks to the re-association of operations, the compiler can reshuffle operations in a more efficient way

4 summation of subsequent elements in the array!

Step 3: unrolling 2×2

As we have seen, what is the blocking element is the critical path of the accumulator, because we are using a unique place to store the summation.
A logical step is to *separate* partial results in multiple accumulators.

```
for ( int i=0; i<N; i++ )
  S = S OP A[i];
```

```
for ( int i=0; i<N-2; i+=2 )
  {
     s0 = s0 OP A[i];
     s1 = s1 OP A[i+1];
  }
return s0 = s0 OP s1;
```

Step 2: unrolling 2×2

```
for ( int i=0; i<N-2; i+=2 ) {
  s0 = s0 OP A[i];
  s1 = s1 OP A[i+1]; }
```

```
.L49:
        vmovupd         ymm4, YMMWORD PTR [rax]
        vmovupd         ymm3, YMMWORD PTR 32[rax]
        vmovapd         xmm2, xmm4
        vaddsd          xmm1, xmm1, xmm4
        vunpckhpd       xmm2, xmm2, xmm2
        vaddsd          xmm0, xmm0, xmm2
        vextractf128    xmm4, ymm4, 0x1
        vaddsd          xmm1, xmm1, xmm4
        vunpckhpd       xmm4, xmm4, xmm4
        vaddsd          xmm0, xmm0, xmm4
        vmovapd         xmm6, xmm3
        vaddsd          xmm5, xmm1, xmm3
        vunpckhpd       xmm6, xmm6, xmm6
        vaddsd          xmm0, xmm0, xmm6
        vextractf128    xmm3, ymm3, 0x1
        vaddsd          xmm1, xmm5, xmm3
        add             rax, 64
        vunpckhpd       xmm3, xmm3, xmm3
        vaddsd          xmm0, xmm0, xmm3

        cmp             rax, rcx
        jne     .L49
```

As before (2x1g) the compiler feels free to load 8 doubles per iteration, and then reshuffling them appropriately in order to respect the semantics of our coding, it sums them up using `xmm0` and `xmm1` as separate accumulators.

Loop unrolling

In the next few slides, before to discuss the results from the different implementation, let's have a look at what happens if the accumulator S is required to be `long double` instead of `double`
(`long double` is not supported by vector registers)

```
for ( int i=0; j<N; i++ )
  S += A[i];
```

Step 1: unrolling 2×1

Accumulator is long double

The compiler generates (*) the following assembly code:

```
.L10:
        fadd    QWORD PTR [rax]
        add     rax, 16
        fadd    QWORD PTR -8[rax]

        cmp     rdx, rax
        jne     .L10
```

`fadd` sums the content of the memory specified by the argument to the first register of the FPU

`rax` contains the address to the *i*th element of the array, `[rax]` means "the address pointed by `rax`"

Now `rax` points to array's element *i+2*

This sums the array's element *i+1* to the first FPU's register, which hold the updated result of the summation chain

`rdx` is equal to *N-2*, just check the exit condition

(*) in the following we will always comment the code generated with `-O3 -march=native`

Step 1: unrolling 2×1

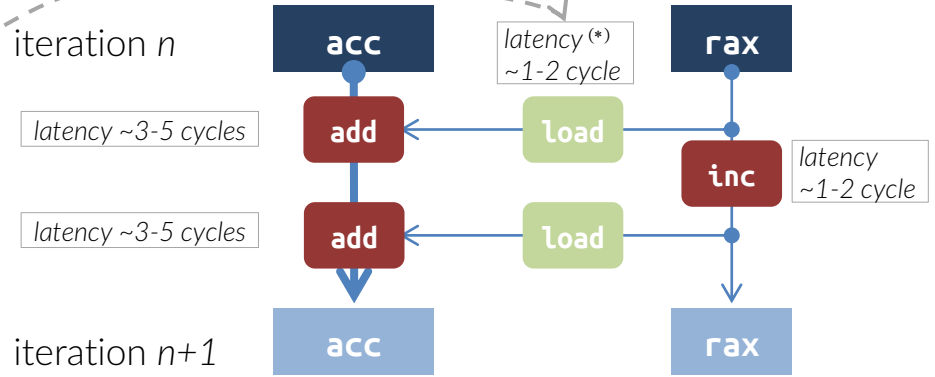
Accumulator is long double

Then, we have the following abstraction:
(arrows indicate a dependency)

```
.L10:

        fadd    QWORD PTR [rax]
        add     rax, 16
        fadd    QWORD PTR -8[rax]

        cmp     rdx, rax
        jne     .L10
```



Both the FPU's register, which we name **acc**, and **rax** carry a *loop dependency* because their result is used – and them waited for – in the subsequent iteration.
However, the FP addition has a larger latency (~3-5) than int addition (~1).
As such, the left one (in bold) is the *critical path* that limits our efficiency.

(*) obviously depends on whether the data are in L1, L2, L3 or RAM

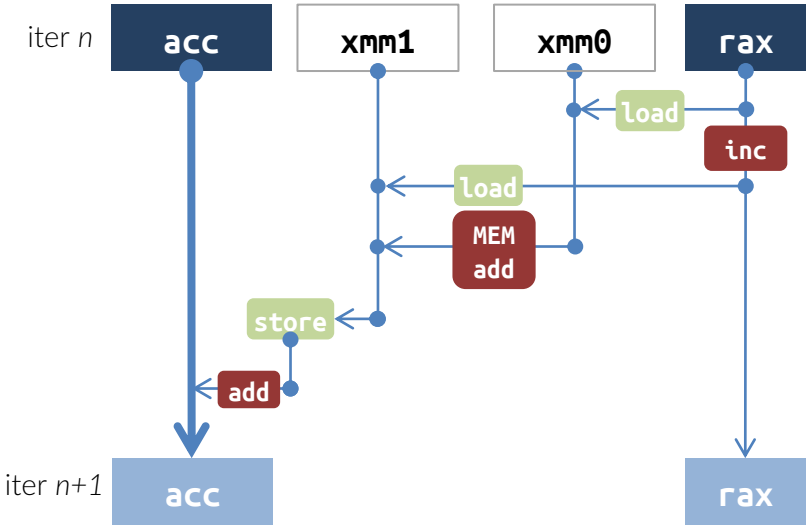
Step 2: unrolling 2x1 + reshuffle

Accumulator is long double

```
for ( int i=0; i<N-2; i+=2 )
  S = S OP (A[i] OP A[i+1]) ;
```

```
.L19:
        vmovsd  xmm0, QWORD PTR [rax]
        add     rax, 16
        vaddsd  xmm1, xmm0, QWORD PTR -8[rax]
        vmovsd  QWORD PTR -16[rsp], xmm1
        fadd    QWORD PTR -16[rsp]

        cmp     rdx, rax
        jne     .L19
```

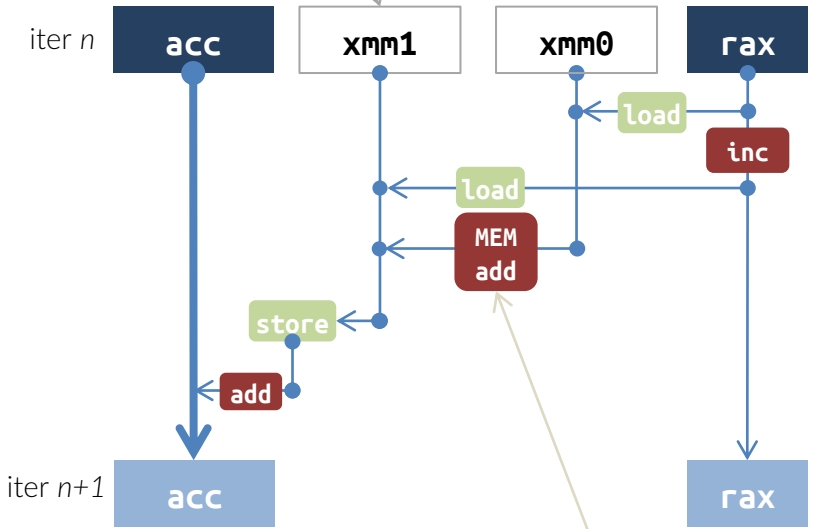


Step 2: unrolling 2×1+ reshuffle

Accumulator is long double

The scenario is somehow different now.
The loop dependency is still carried by `acc` (the latency of `rax` is negligible); however, its latency is now hidden by the operations on `xmm0` and `xmm1`.

We may expect some improvement from this code.



White background means that these regs are *local* to each iterations, i.e. they do not carry any loop dependency

This add is within xmm0 and the *(i+1)-th* array's element from memory

Step 3: unrolling 2×2

As we have seen, what is the blocking element is the critical path of the accumulator, because we are using a unique place to store the summation.
A logical step is to *separate* partial results in multiple accumulators.

```
for ( int i=0; i<N; i++ )
  S = S OP A[i];
```

```
for ( int i=0; i<N-2; i+=2 )
  {
     s0 = s0 OP A[i];
     s1 = s1 OP A[i+1];
  }
return s0 = s0 OP s1;
```

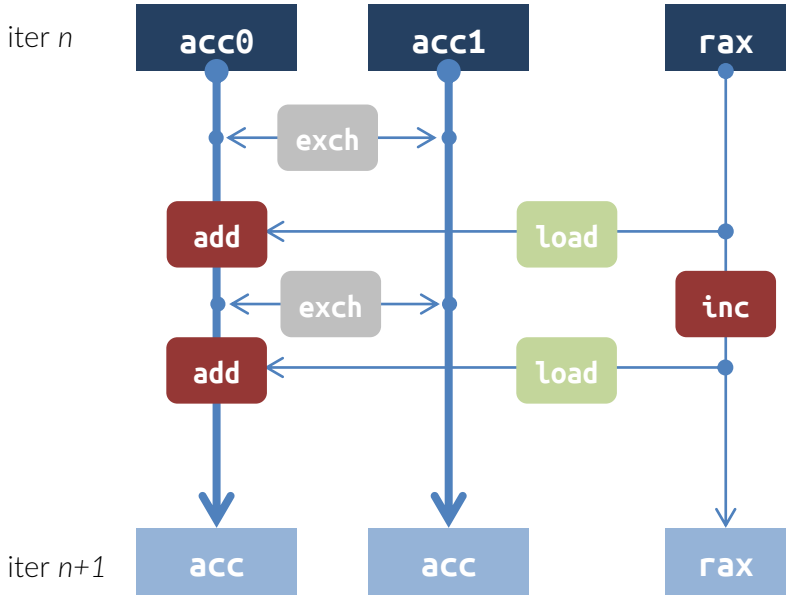
Step 3: unrolling 2×2

Accumulator is long double

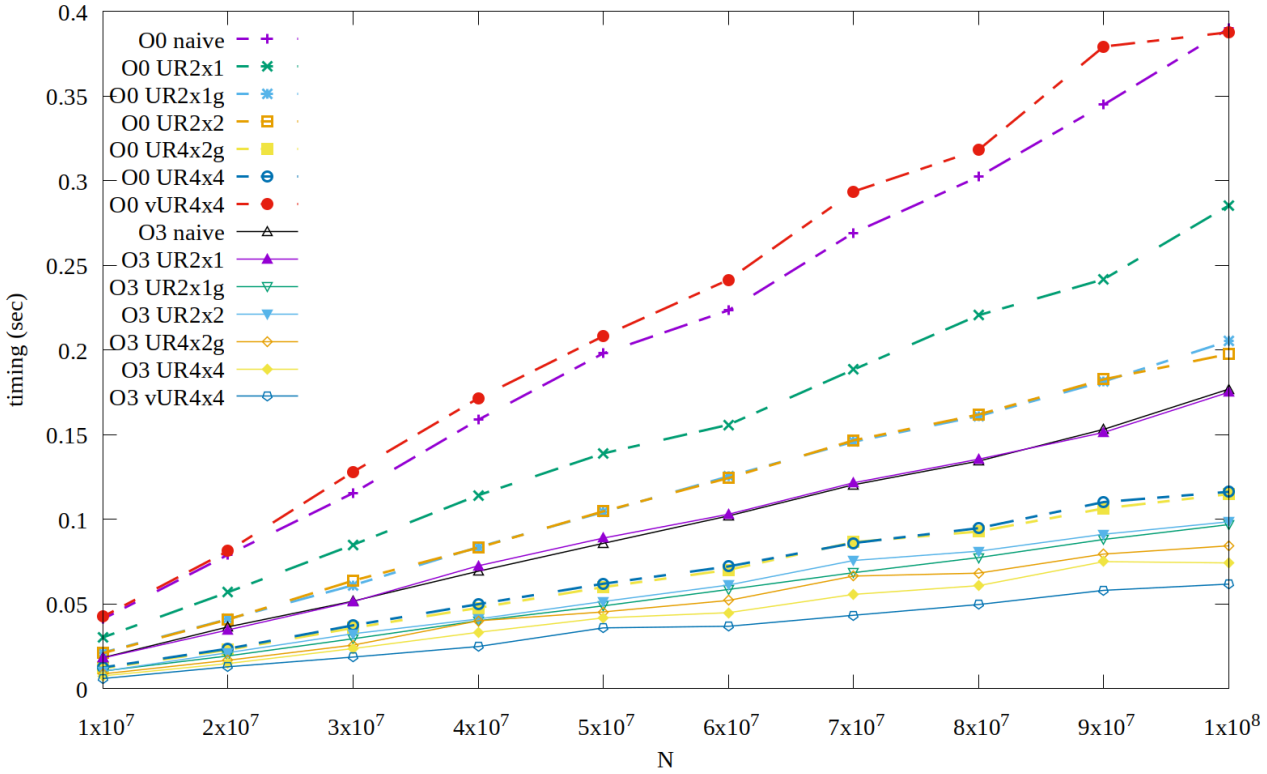
```
for ( int i=0; i<N-2; i+=2 ) {
  s0 = s0 OP A[i];
  s1 = s1 OP A[i+1]; }
```

```
.L34:
        fxch    st(1)
        fadd    QWORD PTR [rax]
        fxch    st(1)
        add     rax, 16
        fadd    QWORD PTR -8[rax]

        cmp     rdx, rax
        jne     .L34
```



Reduction: results



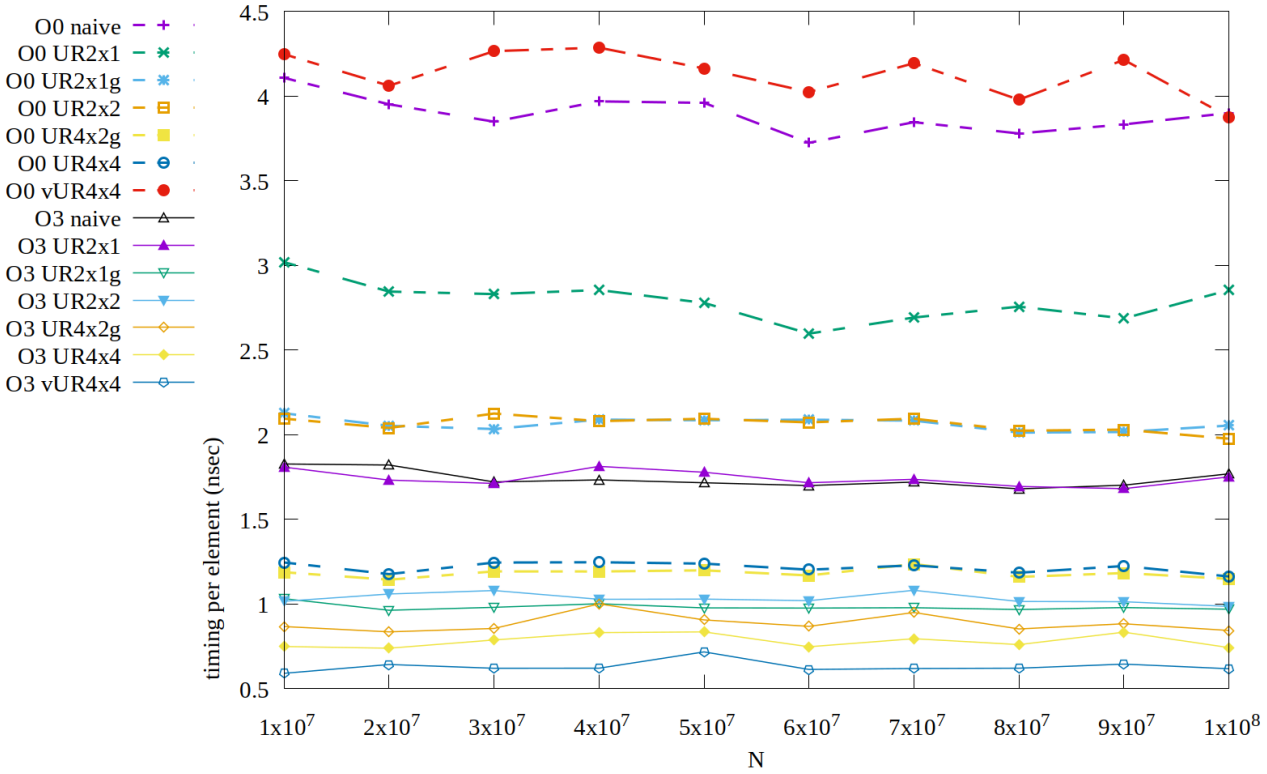
Run time of different implementation with and without compiler's optimization

UR NxM: unrolled *N* times using *M* accumulators.

vUR4x4: UR4x4 with explicit vectorization.

In this plot and in the following ones dashed lines are for `-O0` and solid line for `-O3 -march=native` (only `gcc` has been used)

Reduction: results



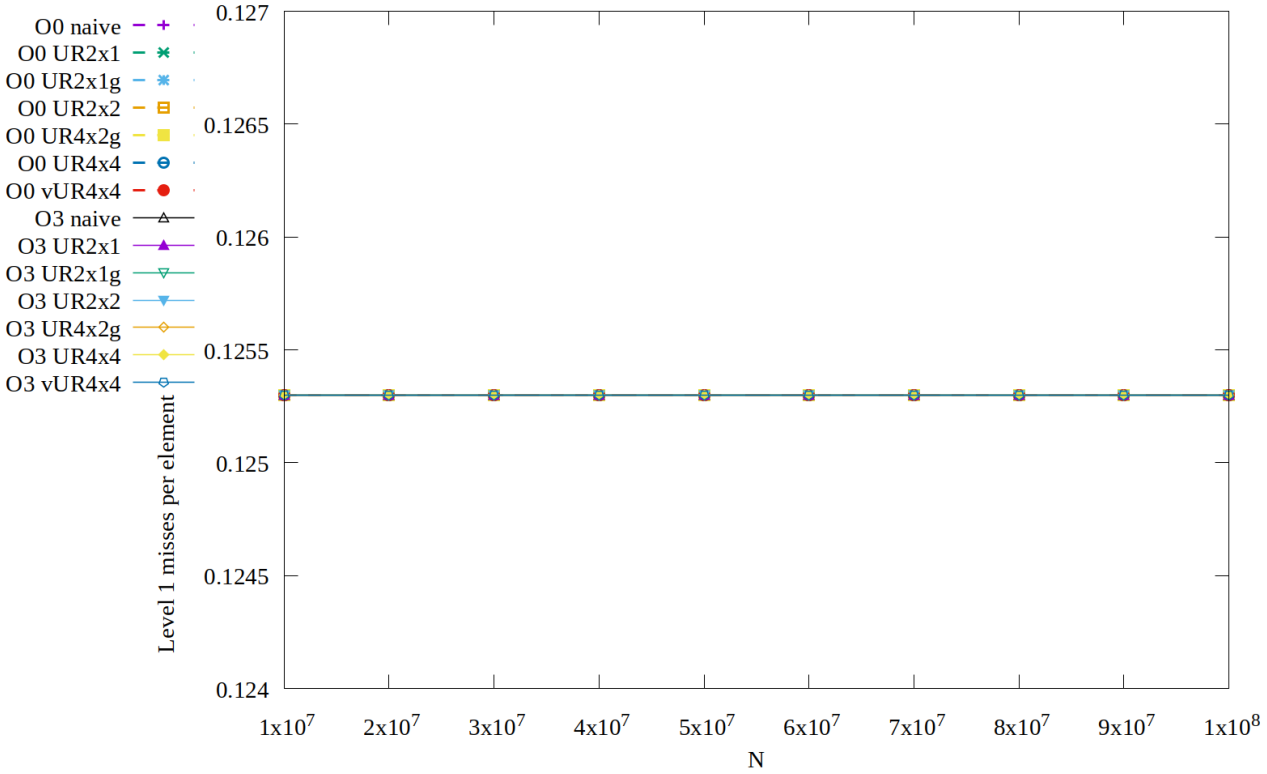
Run time of different implementation with and without compiler's optimization

UR NxM: unrolled *N* times using *M* accumulators.

vUR4x4: UR4x4 with explicit vectorization.

In this plot and in the following ones dashed lines are for `-O0` and solid line for `-O3 -march=native` (only `gcc` has been used)

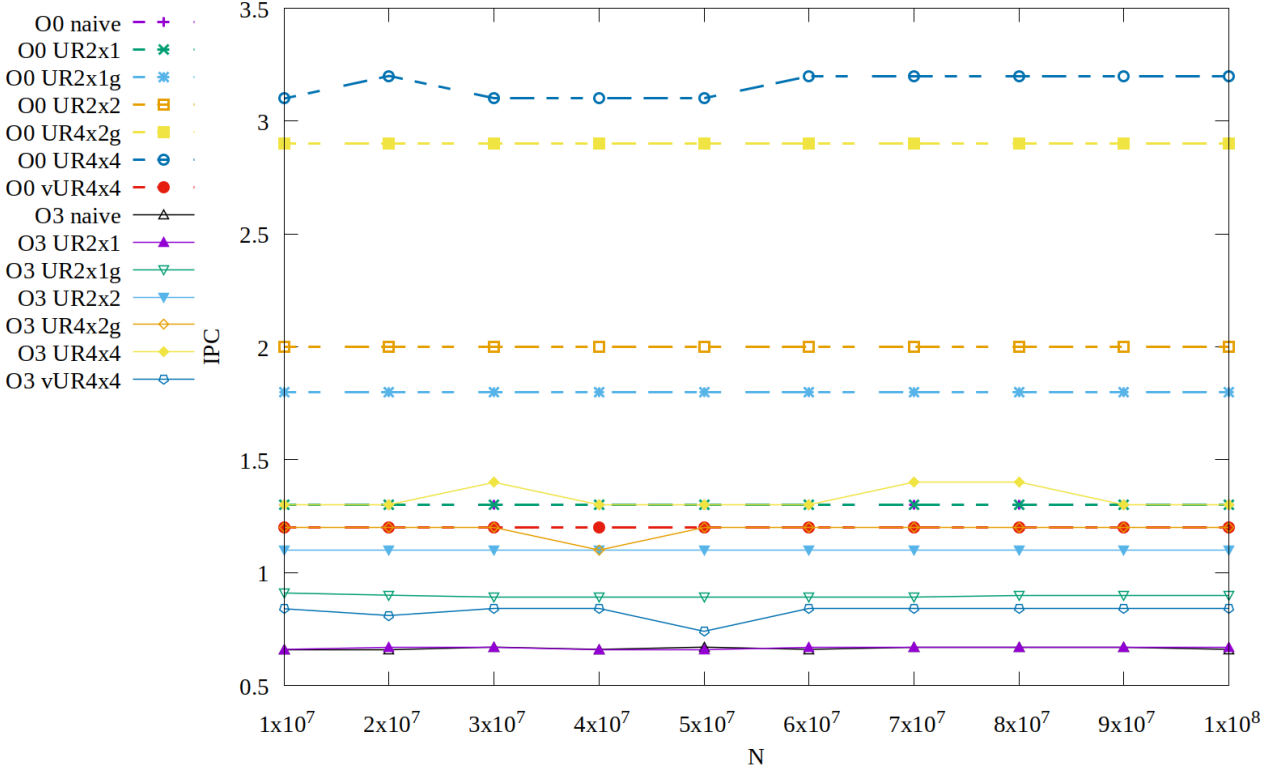
Reduction: results



Level 1 Data cache misses

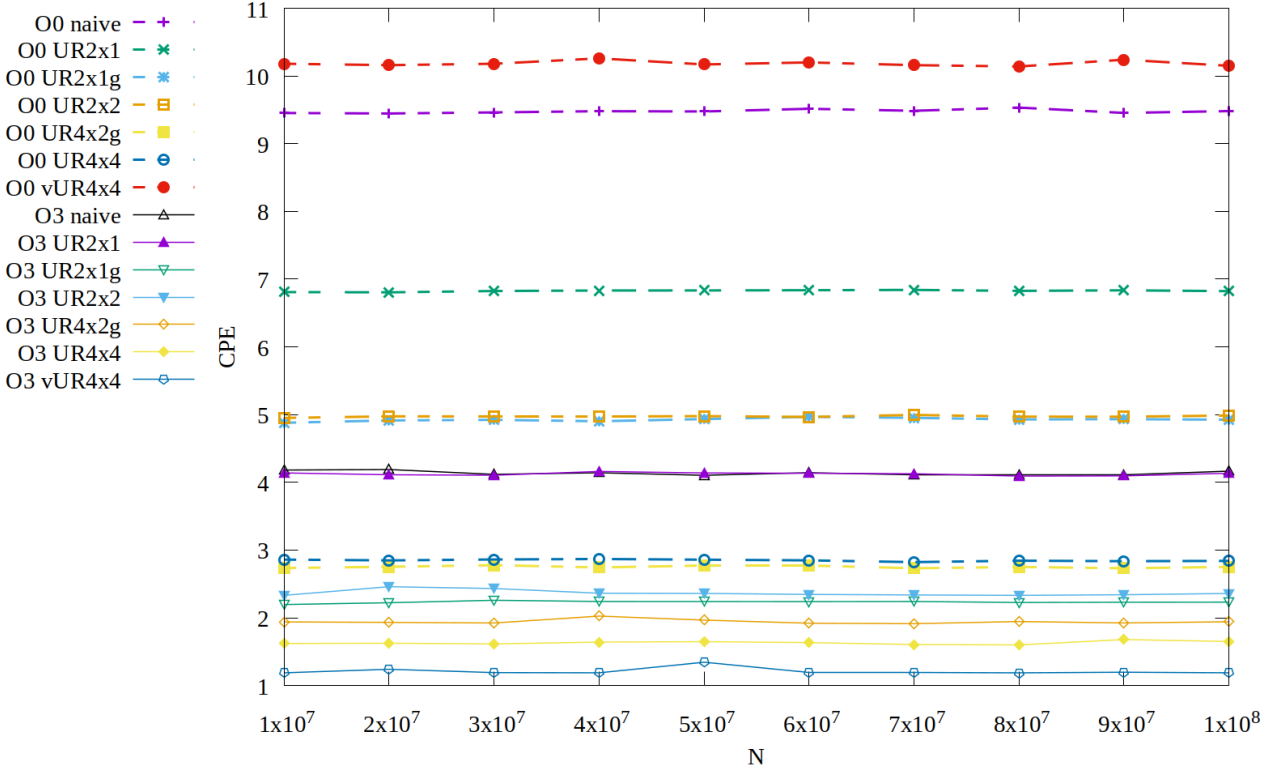
of course, the expected 1/8 since we are processing the array continuosly.

Reduction: results



Instructions per cycle

Reduction: results



Cycles per element

Cache access in loops: $O(N^3)/O(N^2)$

These algorithms (ex: matrix-matrix multiplication or dense matrix diagonalization) are very good candidates for optimizations that lead flop/s performance very close to the theoretical peak (in fact, MMM is at the core of `linpack`).

Tailing, unroll&jam + vectorization of operations, reorganization of ops to exploit CPU's pipelines and out-of-order capability, are all used by extremely specialized libraries.

→ It is a brilliant idea to link those library instead of developing your own algorithm, unless some very special needs must be met.

$O(N^3)/O(N^2)$ example

matrix-matrix multiplication is a very common task in HPC.
Although there are highly optimized library that performs the job, it is a very classical and useful case study to understand the loop tiling and the cache-oblivious algorithms.

Let's start from the definition of the problem.
Given 2 matrices, A and B, having respectively *(m, n)* and *(n, p)* rows and columns respectively, their product is defined as the matrix C(*m, p*)

$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} \times B_{k,j}$$

$O(N^3)/O(N^2)$ example

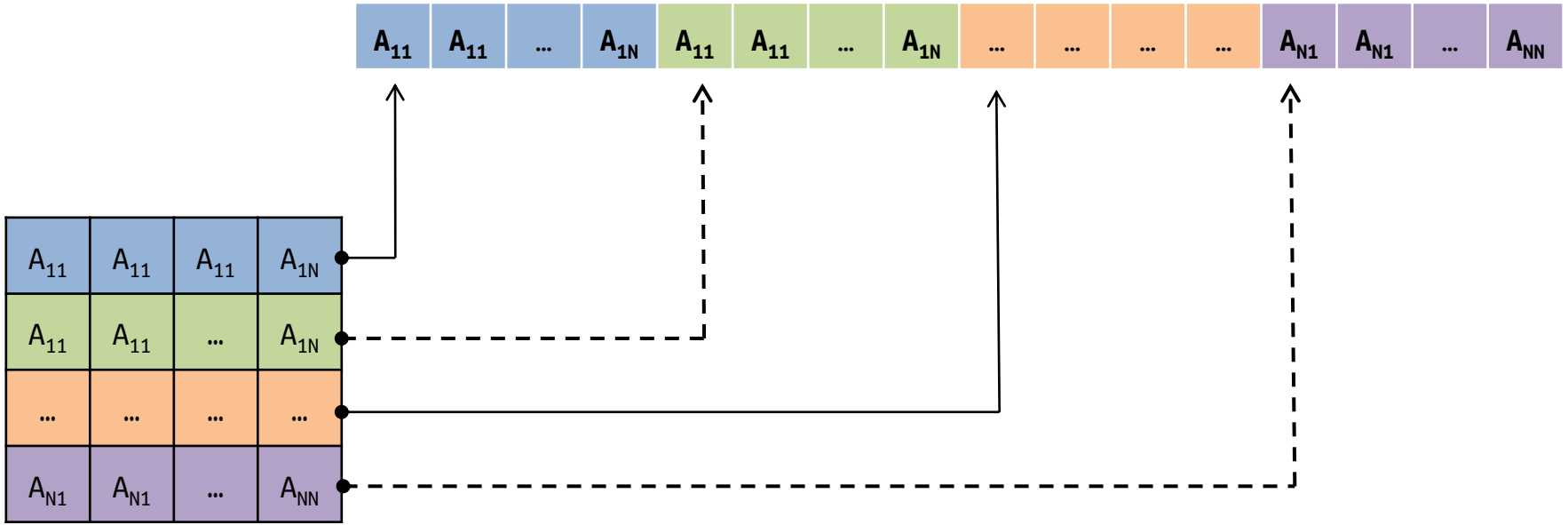
$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} \times B_{k,j}$$

A possible obvious straightforward implementation of this algorithm is as follows:

```
for ( int i = 0; i < m; i++ )          // traverse the A's (and C's) rows
   for ( int k = 0; k < p; k++ )       // traverse the B's rows ( Ac = Br )
      for ( int j = 0; j < n; j++ )
         C[i][k] += A[i][j] * B[j][k];
```

Note: how a matrix is stored in memory

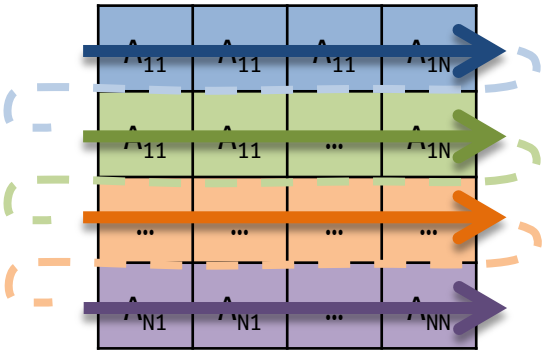
Remember the obvious fact that your memory is a continuous 1-dimensional stream of bytes.



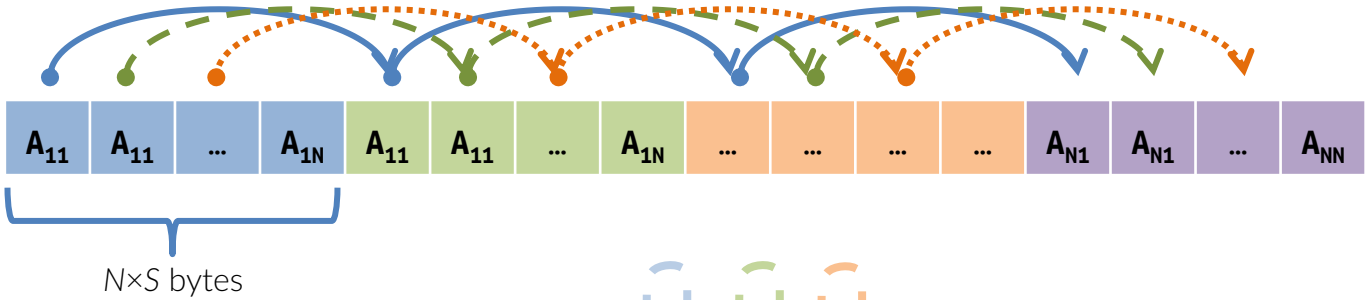
This convention is the C/C++ convention, which is labelled as *row-major order*. Note that the Fortran convention is opposite, with columns being contigous in memory (*column-major*).

Note: how a matrix is stored in memory

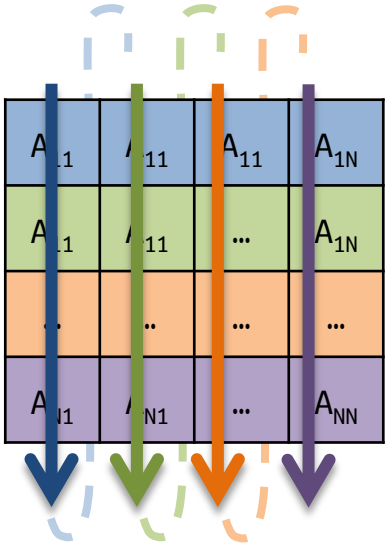
Then, traversing the matrix in the same row-major order amounts to traverse the memory in contiguous order.



Note: how a matrix is stored in memory



Whereas, traversing the matrix in the opposite column-major order amounts to jump in memory by *N* positions, i.e. *N*×*S* bytes is *S* is the size of each element.



$O(N^3)/O(N^2)$ example

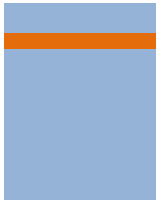
Matrix representation

Memory representation

C

=

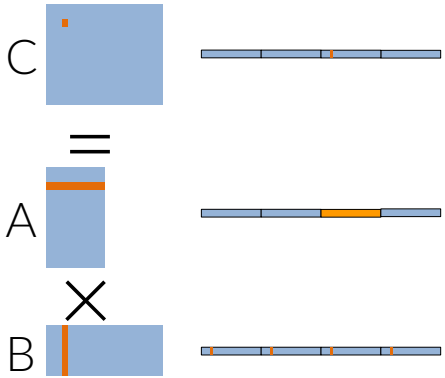
A



X

B

$O(N^3)/O(N^2)$ example



The naïve implementation has an obvious issue with data locality for large enough matrixes.
For each C's element, possibly all accesses to B result in a cache miss.
Then, we may have *mnp* cache misses (if L is the line capacity of the cache in terms of the data type used) only to traverse B.
The total number of expected misses is:

mn/L +	C is traversed once
mnp/L +	A is scanned entirely *n* times
mnp	B is accessed randomly

In fact, the naïve implementation is never used for any large matrix multiplication: since *2mnp* flop are required, it amounts to have nearly a cache miss per each flop.
How can we fix this problem ?

Transposing the matrix B before entering the loop should alleviate the problem; although the transposition requires some additional work, for large enough matrices there is still a performance gain.

$O(N^3)/O(N^2)$ example

A different strategy may consist in swapping the two inner loops:

```
for ( int i = 0; i < m; i++ )
   for ( int k = 0; k < p; k++ )
      for ( int j = 0; j < n; j++ )
         C[i][k] += A[i][j] * B[j][k];
```

```
// traverse the A's (and C's) rows
// traverse the B's rows ( Ac = Br )
```

Now we are still having lots of cache misses due to the fact that we are re-loading C[i][k] many times (Ac times).

```
for ( int i = 0; i < m; i++ )
   for ( int j = 0; j < n; j++ )
      for ( int k = 0; k < p; k++ )
         C[i][k] += A[i][j] * B[j][k];
```

Now we expect to have

mnp/L	+	running over C
mnp/L	+	running over A
mnp/L		running over B

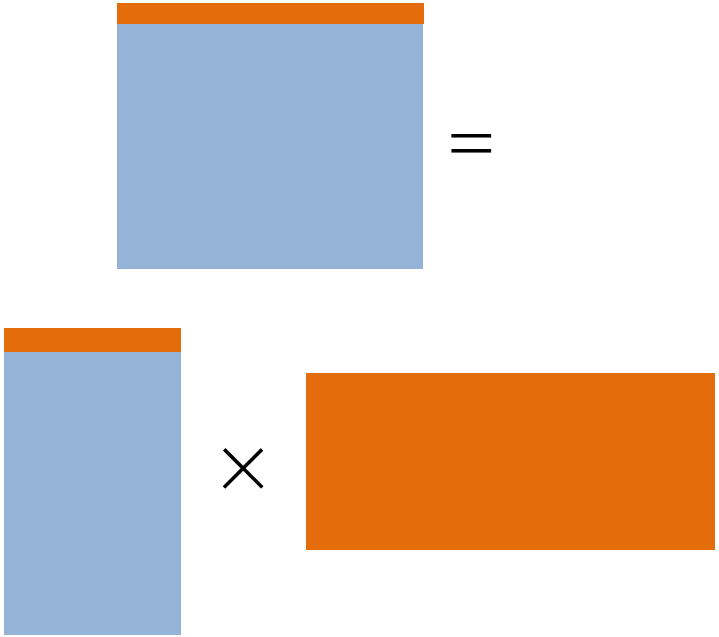
cache misses. Then, with respect to the previous nesting scheme we expect to have

~ mnp(L -2)

less cache misses.

$O(N^3)/O(N^2)$ example

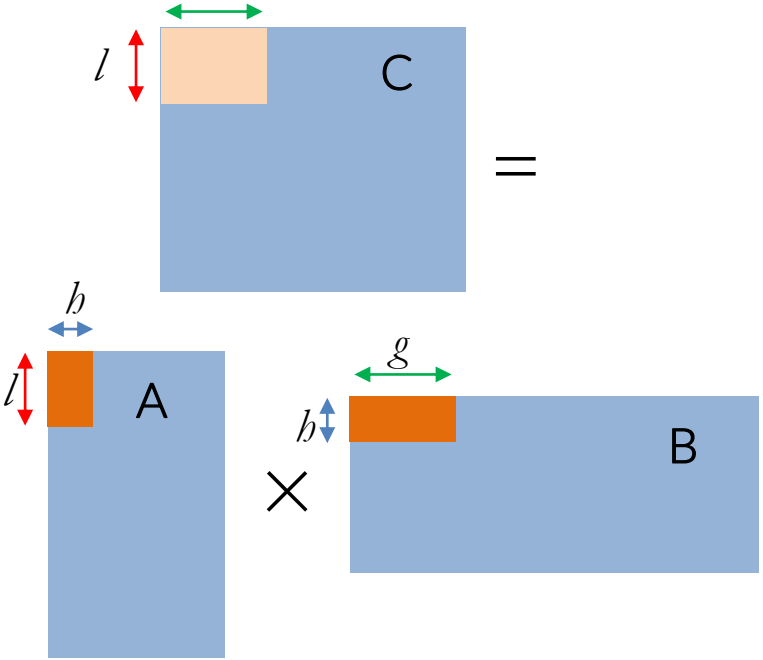
We can do even better by optimizing both the memory accesses and the data contiguity by *tailing* the loops:



To compute a single line in C we need to access the corresponding line in A *p* times. In the hypothesis that A,B and C can not fit in memory, each time the cache will have been flushed and the A's line will not be there anymore.
This amounts to have *n/L* compulsory misses per each column if B, i.e. *np/L* cache misses for each C's line, as we have already calculated.
The same holds for the B' s columns and so on…

$O(N^3)/O(N^2)$ example

We can do even better by optimizing both the memory accesses and the data contiguity by *tailing* the loops:



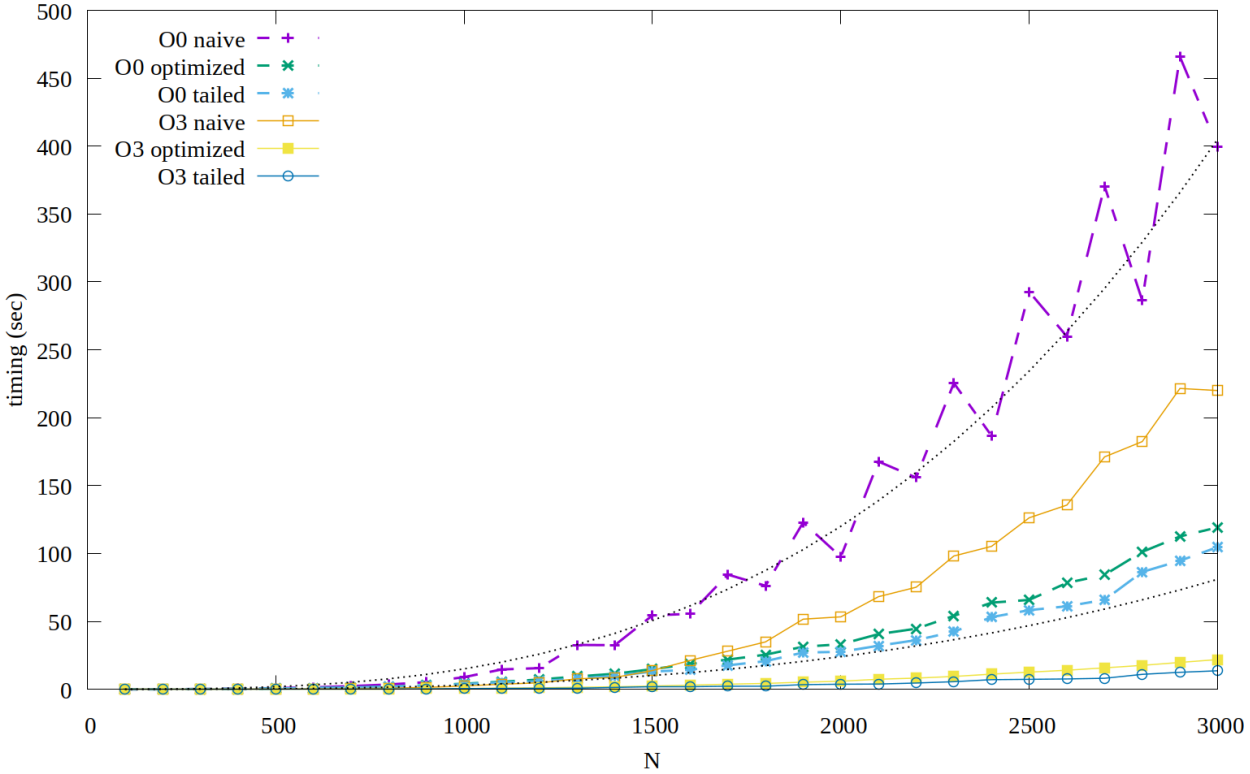
If, instead, we keep in the cache a segment of the A's line, re-using it against the columns of B – or, better, against a columns section tall as the line segment is large (blue arrows in the figure) – we could greatly reduce the amount of cache misses per C's element.

Traversing A and B by *blocks* as in the figure allows to accumulate partial results in the corresponding C's area while decreasing the number of cache misses by a factor

$$L / (l \times h \times g)$$

where L is the cache capacity and are the block factors. With standard value, this figure becomes of the order of 0.001.

Matrix multiplication results



Run time of different implementation with and without compiler's optimization

the results are for the case of 2 square matrix of dimension N

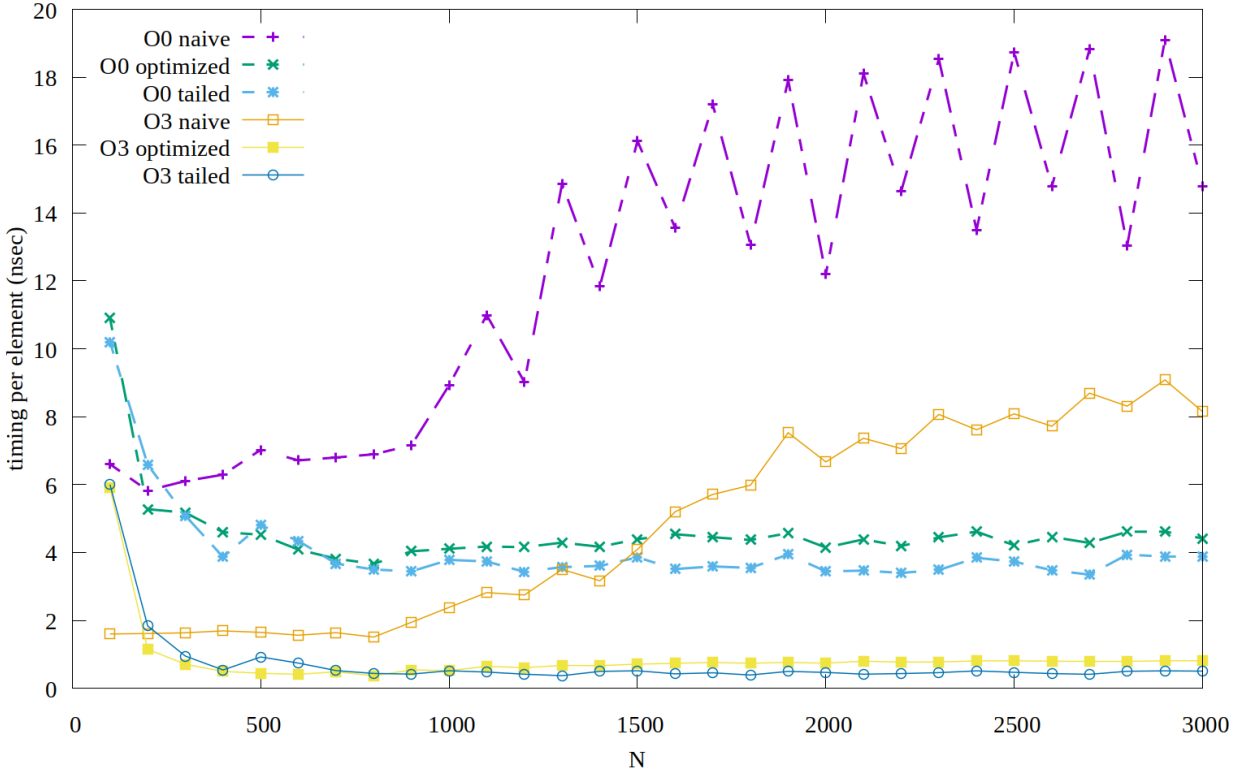
Naïve: the schoolbook's implementation

Optimized: inner loops swapped

Tailed: M-M by blocks

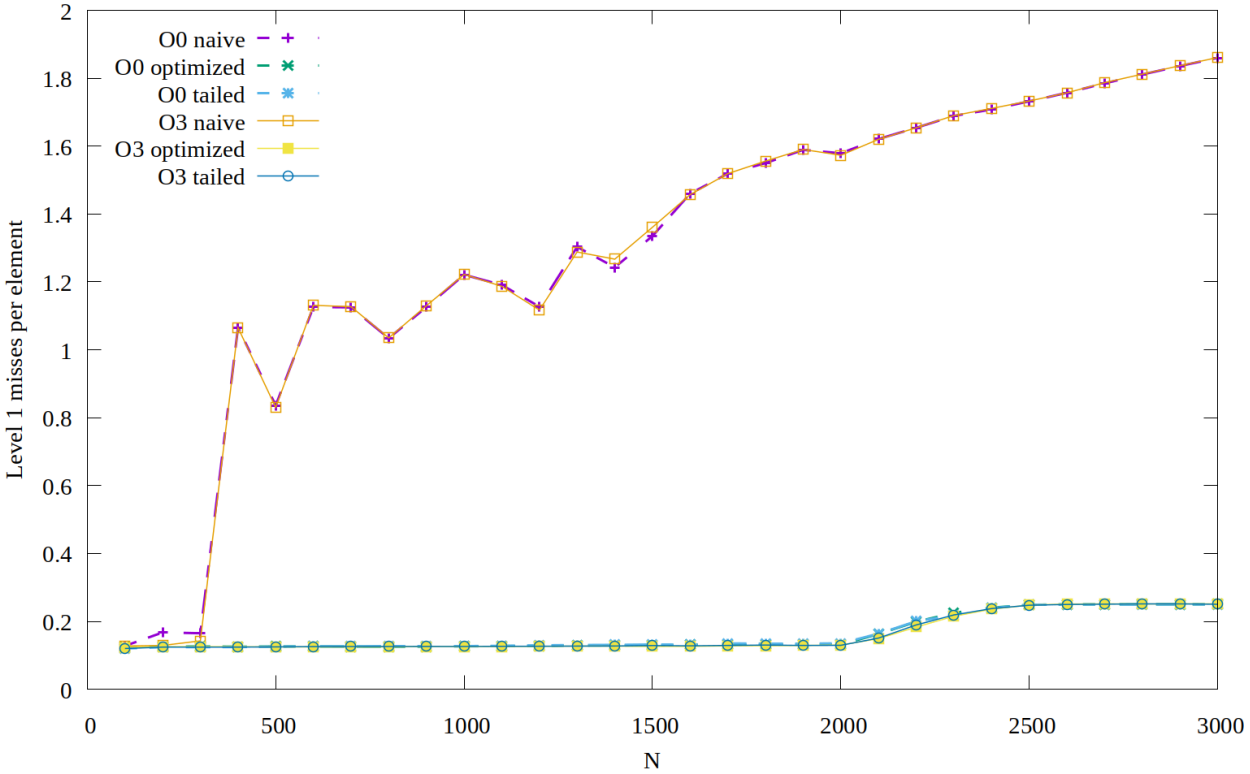
In this plot and in the following ones dashed lines are for `-O0` and solid line for `-O3 -march=native` (only `gcc` has been used)

Matrix multiplication results



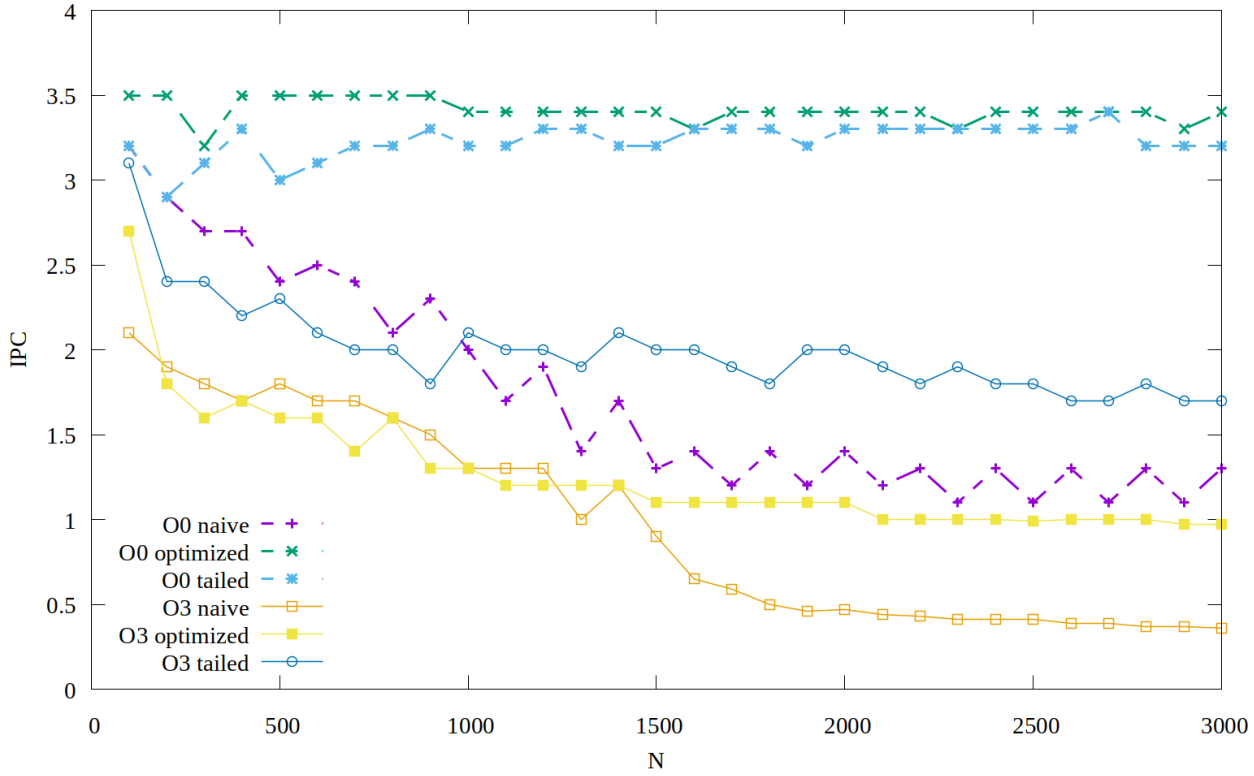
Time per element accessed (N^3)

Matrix multiplication results



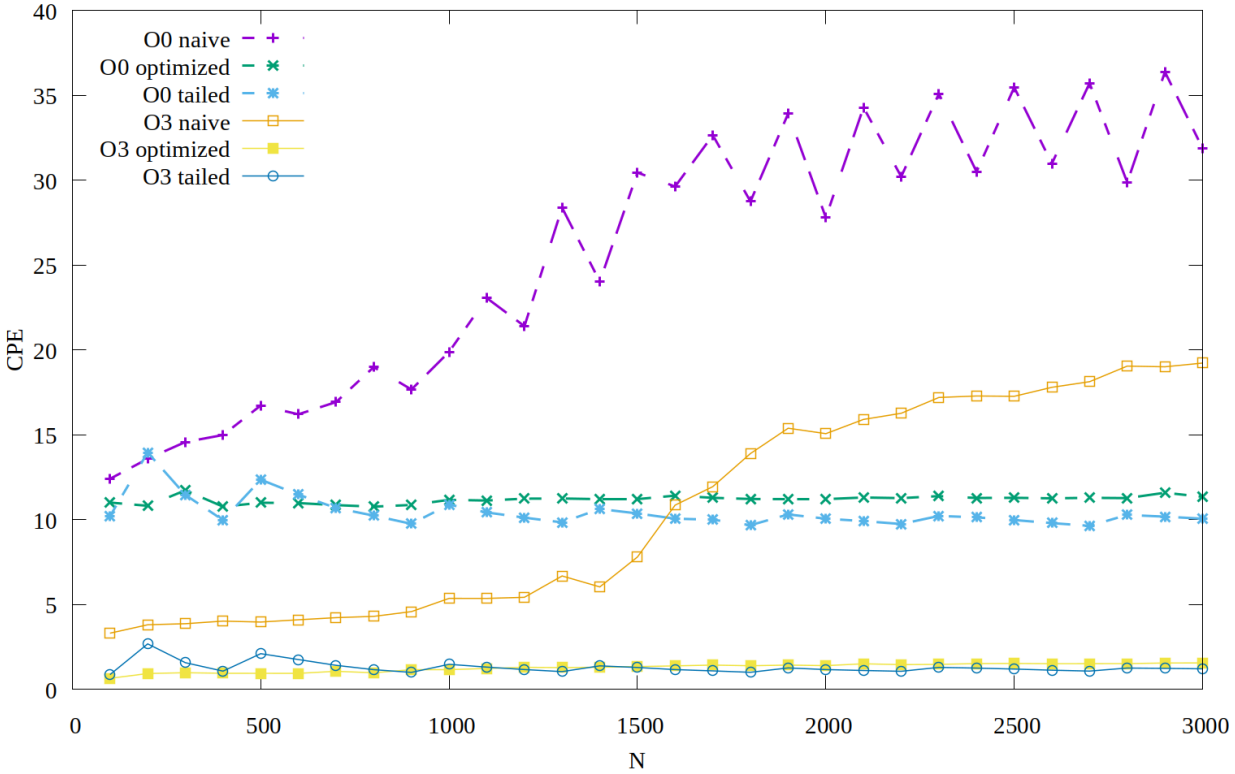
Level 1 Data cache misses per element

Matrix multiplication results

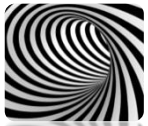


Instructions per cycle

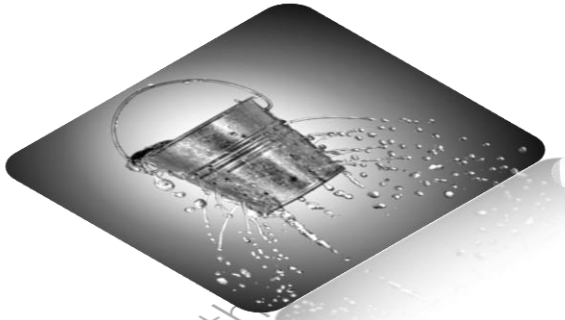
Matrix multiplication results



Cycles per element



Outline



Avoid the avoidable inefficiencies

Loops techniques

Prefetching

At the right moment, at the right place

We know that waiting for data and instructions is a major performance killer.

Modern CPUs have the capability of pre-emptively bring from memory into cache levels data that **will be needed shortly afterwards**.
They can do that following some speculative algorithm based on the current execution flow and assuming spatial locality and temporal locality.

Both *data* and *instructions* can be pre-fetched.

Pre-fetching may be both hardware-based and software-based (tipically the compiler insert pre-fetching instructions at compile-time).

At the right moment, at the right place

From the point of view of the programmer, there are 2 possible ways to deal with prefetching:

EXPLICIT — you explicitly insert a pre-fetching directive.
Very difficult to be achieved effectively: the directive must be inserted timely but not too early (data eviction) or too late (load latency).

INDUCED — you consciously arrange data layout and execution flow so that to make it obvious to the compiler what to prefetch.

Explicit prefetching

This is a standard binary search implementation.

Find the median element

Define the next search

```
int mybsearch(int *data, int N, int Key)
 {
   int register low = 0;
   int register high = N;
   int register mid;

   while(low <= high) {
     mid = (low + high) / 2;

     if(data[mid] < Key)
       low = mid + 1;
     else if(data[mid] > Key)
       high = mid-1;
     else
       return mid;
   }
   return -1;
 }
```

Explicit prefetching

We can make it better by simply making sure that the element to be compared for (the `mid`) is in the cache when requested

```
int mybsearch(int *data, int N, int Key)
 {
   int register low = 0;
   int register high = N;
   int register mid;

   while(low <= high) {
     mid = (low + high) / 2;

     if(data[mid] < Key)
       low = mid + 1;
     else if(data[mid] > Key)
       high = mid-1;
     else
       return mid;
   }
   return -1;
 }
```

Explicit prefetching

We can make it better by simply making sure that the element to be compared for (the `mid`) is in the cache when requested

```
int mybsearch(int *data, int N, int Key)
 {
   int register low = 0;
   int register high = N;
   int register mid;

   while(low <= high) {
     mid = (low + high) / 2;
      __builtin_prefetch (&data[(mid + 1 + high)/2], 0, 3);
      __builtin_prefetch (&data[(low + mid - 1)/2], 0, 3);

     if(data[mid] < Key)
       low = mid + 1;
     else if(data[mid] > Key)
       high = mid-1;
     else
       return mid;
   }   return -1; }
```

Explicit prefetching

```
luca@GGG:~/code/HPC_LECTURES/prefetching% ./prefetching_off
performing 13421772 lookups on 134217728 data..
set-up data.. set-up lookups..
start cycle.. time elapsed: 20.7534
luca@GGG:~/code/HPC_LECTURES/prefetching% ./prefetching_on
performing 13421772 lookups on 134217728 data with prefetching enabled..
set-up data.. set-up lookups..
start cycle.. time elapsed: 12.6204
```

Explicit prefetching

```
Samples: 71K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 13901140
Overhead       Samples  Memory access
  71,08%         42196  Local RAM hit
  24,14%         17022  LFB hit
   4,11%         10967  L3 hit
   0,63%          1714  L1 hit
   0,02%            75  L2 hit
   0,01%            15  L3 miss
   0,00%             1  Uncached hit
```

```
Samples: 61K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 11720387
Overhead       Samples  Memory access
  68,74%         29450  LFB hit
  27,04%         28208  L1 hit
   2,72%           909  Local RAM hit
   1,29%          2983  L3 hit
   0,20%           346  L2 hit
```

Explicit prefetching

Usage of direct prefetching directive is highly uncertain, since it is difficult to spot the exact point – both in the code and in the execution – where to place them (also because your C code is different than the generated assembly code).

Moreover, the “exact point” is very likely dependent on the system you run on, and then it is susceptible to change significantly.

It is normally much safer to re-organize your code so to have **prefetching by pre-loading**.

Prefetching by moral suasion

Let's discuss together this very simple example before putting the hands on the code you find in the `git`

```
elem a = elements[0]
for ( i = 0; i < 4*N_4; i+= 4 )
 {
        elem e = elem[i+4]; // non-blocking miss
        elem b = elem[i+1]; // possible cache-hit
        elem c = elem[i+2]; // possible cache-hit
        elem d = elem[i+3]; // possible cache-hit
        Elaborate(a);
        Elaborate(b);
        Elaborate(c);
        Elaborate(d);
        a = e;
 }
```

Hands-on

You find code snippets with different flavours of prefetching-by-preloading technique on our GitHub, with some comments about compilation.

Compile and run them with different options (and possibly different compilers) and try to understand what happens on your laptop and/or on HPC facility.

```
for ( i = 0; i < N; i++ )
    sum += array[ i ];
```

that's all, have fun