

Optimization Loops & Prefetching

Luca Tornatore - I.N.A.F.



DATA SCIENCE &
ARTIFICIAL INTELLIGENCE



SCIENTIFIC &
DATA-INTENSIVE COMPUTING

2024-2025 @ Università di Trieste

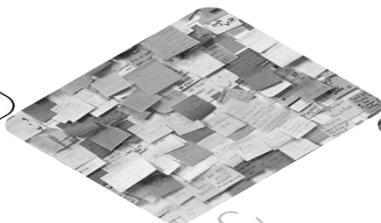


Optimization

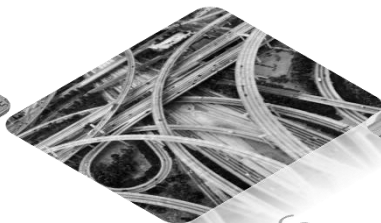
Outline



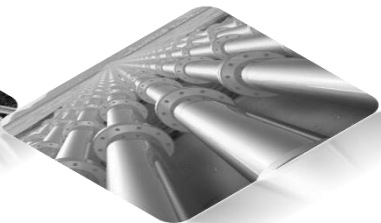
First
things
first



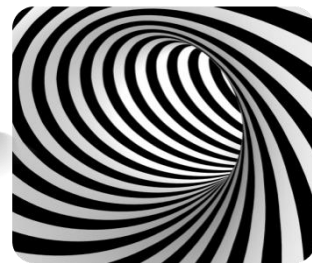
Cache &
Memory



Branches



Pipelines

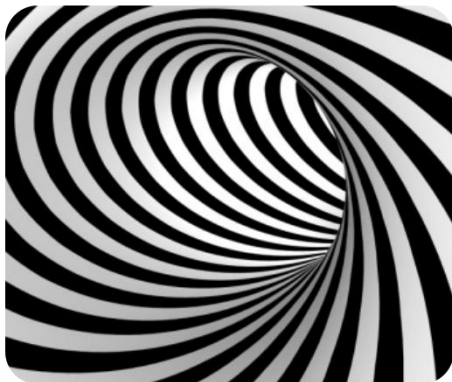


Loops

Loops



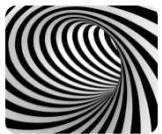
Outline



Loops
techniques



Prefetching



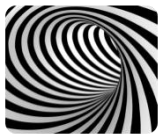
Optimizing cache access in loops

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. $O(N) / O(N)$
scanning arrays, 1D vector ops, ..
optimization potential limited
2. $O(N^2) / O(N^2)$
matrix \times vect, matr. transp, matr. add, ...
some more opportunities for opt.
3. $O(N^3) / O(N^2)$
matrix \times matr,...
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

EX. READ AN ENTIRE
ARRAY IS $O(N)$
TIME

[check the possibility for loops fusion]

$$O(N) / O(N)$$

```
for(int j=0; j<2; j++)
```

```
    A[i] = B[i] × C[i]
```

$\hookrightarrow O(n) + O(n)$ (TIME) \rightarrow

```
for(int j=0; j<2; j++)
```

```
    Q[i] = B[i] + D[i]
```

```
for(int j=0; j<2; j++)
```

```
{
```

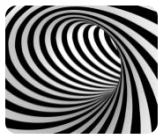
```
    A[i] = B[i] × C[i]
```

```
    Q[i] = B[i] + D[i]
```

```
}
```

$\hookrightarrow O(n)$ (TIME)

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



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

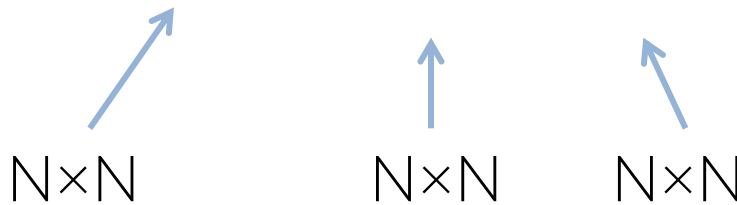
Example

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

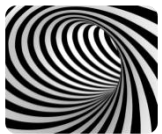
Improvements comes again from increasing *data reuse*, exploiting *locality* and *avoiding unnecessary* operations and memory accesses.

\hookrightarrow Ex. READ MATRIX
IS A $O(N^2)$
TIME

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



$\rightarrow 3 \times N^2$ memory accesses



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

Step 1:

Avoid unnecessary loads /stores

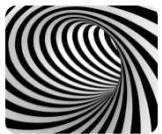
```
for(int i=0; i < N; i++)  
    for(int j=0; j<N; j++)  
        C[i] += A[i][j] * B[j];
```

→

```
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 clear for the compiler that **C[i]** needs to be loaded and stored only 1 time

→ $2 \times N^2 + N$ memory accesses



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

Step 2:

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

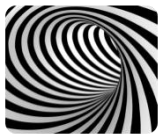
```
for(int i=0; i < N; i++)  
    for(int j=0; j<N; j++)  
        C[i] += A[i][j] * B[j];
```

→ $N^2 \times (1 + 1/m) + N$

→

```
for(int i=0; i < N; i += m)      N×N/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
```

Diagram illustrating the transformation of the loop structure. The original loop is transformed into a loop with a step size m for the outer loop. The inner loop is unrolled, and the operations are fused. The diagram shows the transformation of the loop structure, with annotations indicating the complexity of the transformed code: $N \times N/m$ for the outer loop, $N \times N$ for the inner loop, and N for the total complexity.



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.

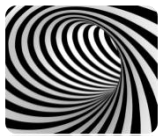
→ **learn to inspect the compiler's log**

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.

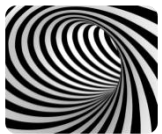
Example: in matrix transpose you have to access all the source matrix and all the destination matrix once.

Unroll & Jam strategy can bring benefits as long as the cache can hold N lines.

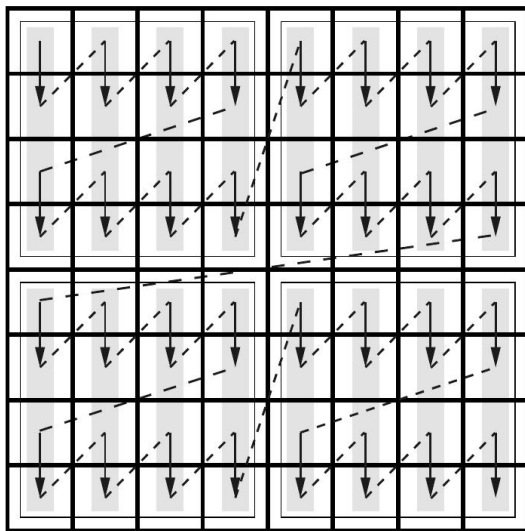
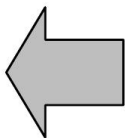
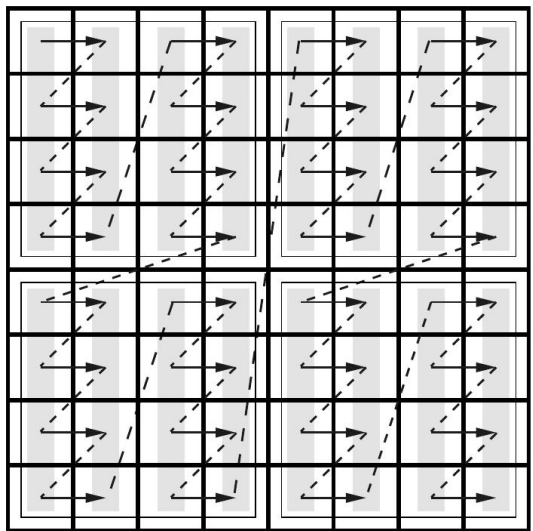
An L_C -way unrolling is too much aggressive and may easily result in register pressure.

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

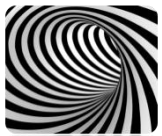
We have mentioned it in the past, we'll see it in more detail at the end of this lecture.



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



| Loop unrolling

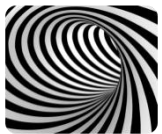


Loop unrolling is a fundamental code transformation which usually helps significantly in improving your code performance:

- 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

```

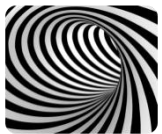
-O3 -march=native

```

.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:

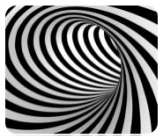
-00

```
.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
```

-03 -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

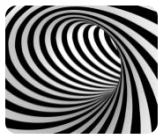


Why does the compiler choose to optimize the loop when the data are of type **int** and it does not when the data are of type **double** ?

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



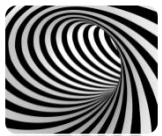
Then, we are left with the responsibility of optimizing this simple code. Since we are traversing it continuously in natural memory order, the cache is not an issue.

Our aim is to re-structure the code so that the compiler could exploit the CPU's ILP (*Instruction-Level Parallelism*).

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



SC0/examples_on_pipelines
reduction



Step 1: unrolling 2×1

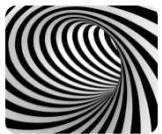


Our first attempt is to reduce the loop overhead and expose some parallelism among the data by explicitly processing 2 elements per iteration.

```
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] ;
```



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];
```

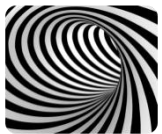


```
for ( int i=0; i < N-2; i+=2 )  
    S = (S OP A[i]) OP A[i+1] ;
```

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

```
int N_ = (N/U)*U;      // by construction this is the largest multiple of U  
                        // smaller than N  
for ( int i = 0; i < N_; i += U )  
    iteration_ops;  
  
for ( int i = N_; i < N; i++ )  
    iteration_ops;
```

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 *ith* element.

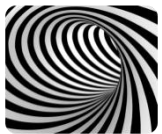
ymm1 has 256bits.

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`



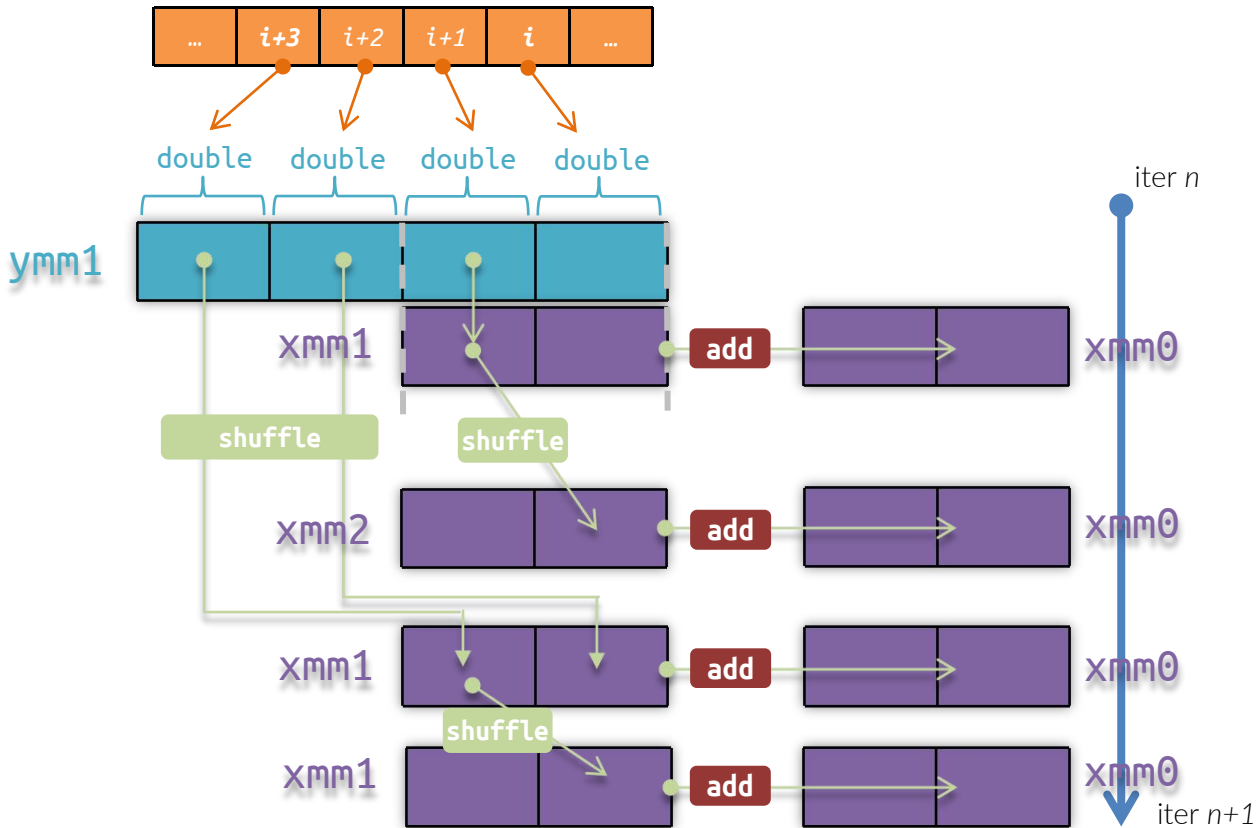
Step 1: unrolling 2×1

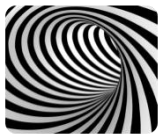


In a (hopefully) simpler view,
the scheme of what happens
is the following →

```
.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:

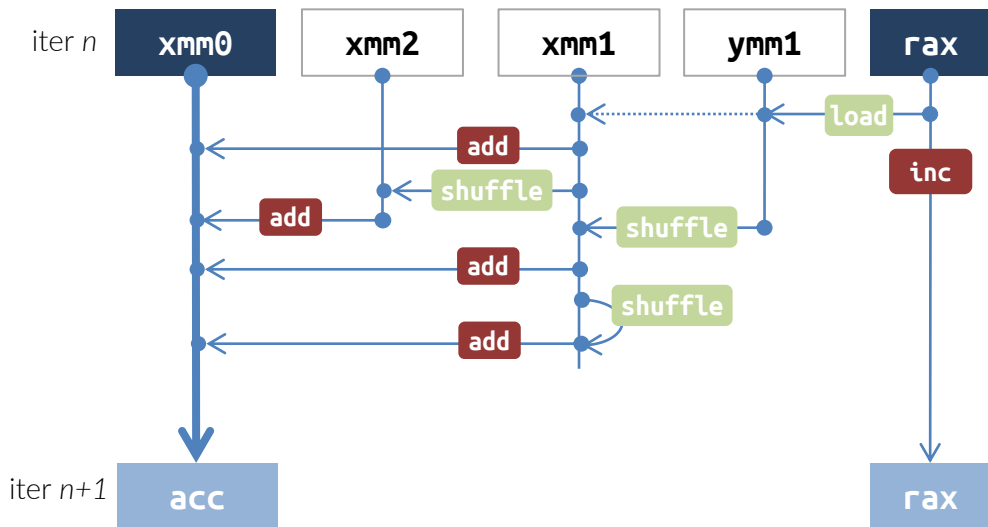
(arrows indicate a dependency)

.L17:

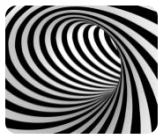
```
vmovupd ymm1, YMMWORD PTR [rax]
add     rax, 32
vaddsd  xmm0, xmm0, xmm1
vunpckhpd xmm2, xmm1, xmm1
vextractf128 xmm1, ymm1, 0x1
vaddsd  xmm0, xmm0, xmm2
vaddsd  xmm0, xmm0, xmm1
vunpckhpd xmm1, xmm1, xmm1
vaddsd  xmm0, xmm0, xmm1
```

.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

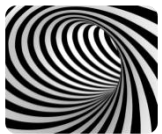


Let's explore what happens if we apport a semantic change to our code, just using the fact that our `OP` is associative.

```
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



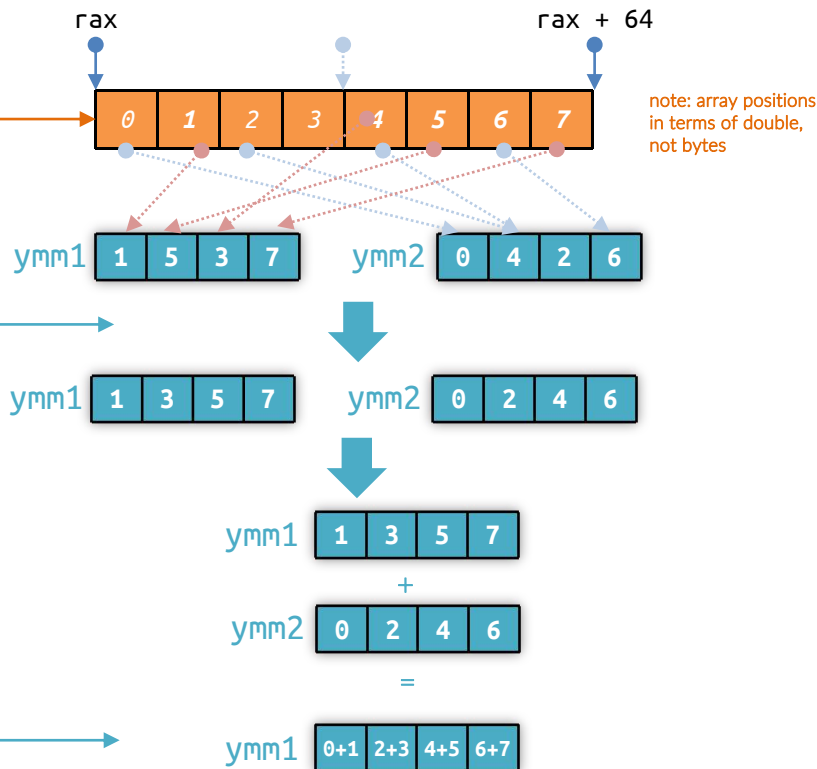
```
for ( int i=0; i<N-2; i+=2 )
    S = S OP (A[i] OP A[i+1]) ;
```



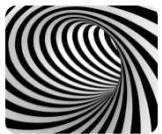
.L33:

```
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 2: unrolling 2×1 + reshuffle

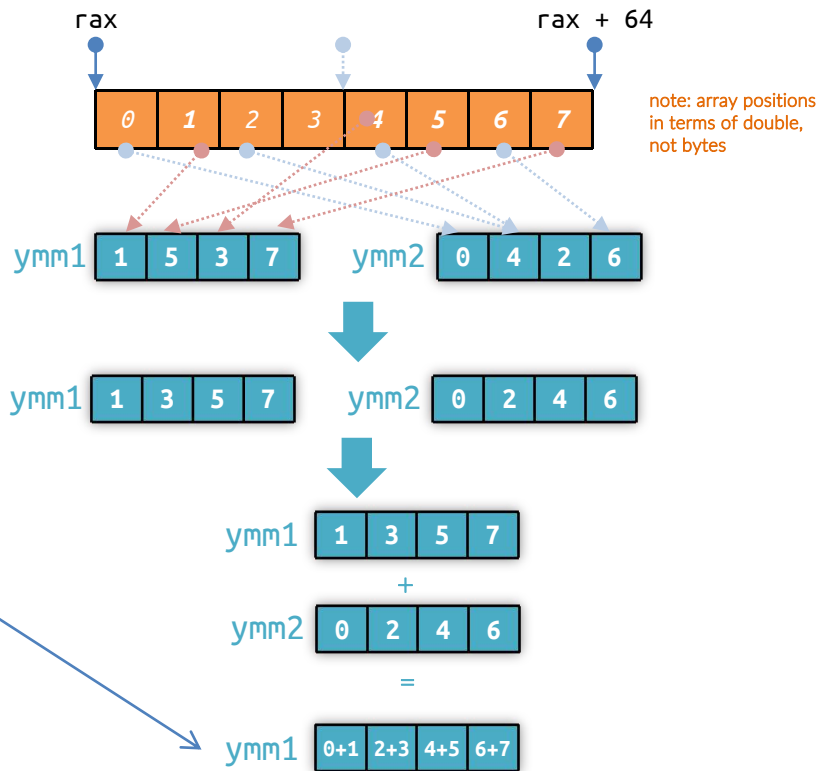


To be clearer: just because we re-associated the math expression in the loop,
 from $S = (S \text{ OP } A[i]) \text{ OP } A[i+1]$;
 to $S = S \text{ OP } (A[i] \text{ OP } A[i+1])$;

the compiler is entitled to exploit the fact that *in the semantics that **now** we are giving*, the operation order will be

$$\text{Sum} = ((([0] + [1]) + ([2] + [3])) + ([4] + [5])) + ([6] + [7]) \dots$$

And the result is what we have just discussed, more efficient than what we obtained with the previous code



4 summation of subsequent elements in the array!



Let's inquire more (still accounting for the fact that in floating point the OP is *not* associative) the difference among

```
1 for ( int i = 0; i < N; i++ )      [A]
2   S = S OP a[i];
3
4 for ( int i = 0; i < N; i+=2 )    [B]
5   S = S OP (a[i] OP a[i+1]);
```

If we define OP to be '+' and S_i to be the i_{th} partial result, i.e. the value of S after the first $i - 1$ iterations, the cases [A] and [B] expand to (square brackets cluster what happens in each single iteration)

$$[A] \rightarrow [[[0 + a_0] + a_1] + a_2] + a_3] + \dots$$

$$S_0 = a_0$$

$$S_1 = S_0 + a_1$$

$$S_2 = S_1 + a_2$$

...

$$S_i = S_{i-1} + A_i$$

$$[B] \rightarrow [[[0 + (a_1 + a_2)] + (a_3 + a_4)] + (a_5 + a_6)] + \dots$$

$$S_0 = 0 + (a_0 + a_1)$$

$$S_1 = S_0 + (a_2 + a_3)$$

$$S_3 = S_2 + (a_4 + a_5)$$

...

$$S_i = S_{i-1} + (a_i + a_{i+1})$$



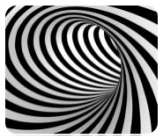
It is evident that for $[A]$ the basic “*element*” of each iteration is the single array entry a_i , whereas in $[B]$ the basic element is the sum $a_i + a_{i+1}$. Hence, while in the first case there is nothing we can do but subsequently calculate the S_i , in the second case we can *separately* calculate as many $[a_i + a_{i+1}]$ elements as possible and *then* sum them up subsequently. Actually, in the second case (because of how we specified the operations!) each a_i, a_{i+1} pair *must* be summed up *before* being summed to S_i .

Q : And how many $[a_i + a_{i+1}]$ elements can we separately calculate in a cycle?

A : In this case, since we are using only 1 accumulator, most probably the choice of the compiler will depend on how wide a vector register is. In fact, since

1. there is the computation of only 1 S_i per iteration, with a subsequent summation
2. it is most effective to exploit a single vector load

the most effective choice is the one made by the compiler, i.e. to use a 2 vector registers to load 8 a 's entries and to reshuffle them in order to obtain 4 $[a_i + a_{i+1}]$ elements in just one $+$ operation, and to sum them up subsequently. If the vector register was 512bits instead of 256bits, it would have loaded 16 entries in order to obtain 8 $[a_i + a_{i+1}]$ elements and so on.



Step 3: unrolling 2×2

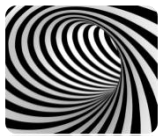


As we have seen, what is the blocking element is the critical path of the accumulator, because we are using a unique place to store the summation. A logical step is to *separate* partial results in multiple accumulators.

```
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



As we have seen, what is
the blocking element is the
critical path of the

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

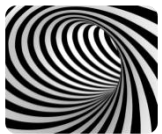
NOTE:

The unrolling is expressed in general as $n \times m$ (in this slide $n=2$, $m=2$; in the previous slides, $n=2$, $m=1$).

n refers to the number of iterations that are unrolled

m refers to the number of accumulators that are being used

So, both the case presented in this slide and the one discussed in the previous slide unroll 2 iterations (in other words: the iteration counter is increased by 2!). However, this case uses 2 accumulators, and so it is 2×2 , while the previous one uses only one and then it is 2×1 .



Step 2: unrolling 2×2



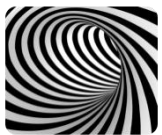
```
for ( int i=0; i<N-2; i+=2 ) {
    s0 = s0 OP A[i];
    s1 = s1 OP A[i+1]; }
```



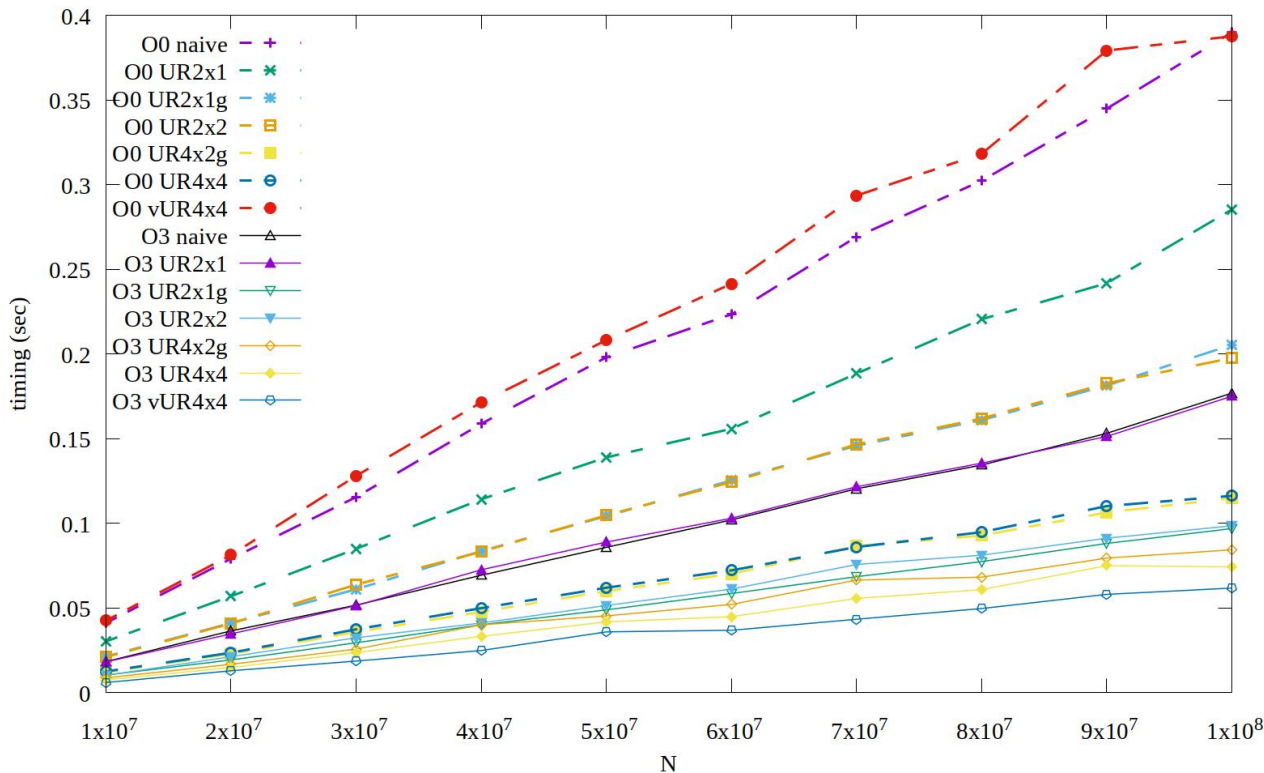
```
.L49:
vmovupd    ymm4, YMMWORD PTR [rax]
vmovupd    ymm3, YMMWORD PTR 32[rax]
vmovapd    xmm2, xmm4
vaddsd     xmm1, xmm1, xmm4
vunpckhpd  xmm2, xmm2, xmm2
vaddsd     xmm0, xmm0, xmm2
vextractf128 xmm4, ymm4, 0x1
vaddsd     xmm1, xmm1, xmm4
vunpckhpd  xmm4, xmm4, xmm4
vaddsd     xmm0, xmm0, xmm4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



Reduction: results (timing)

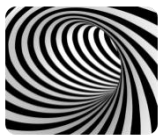


Run time of different implementation with and without compiler's optimization

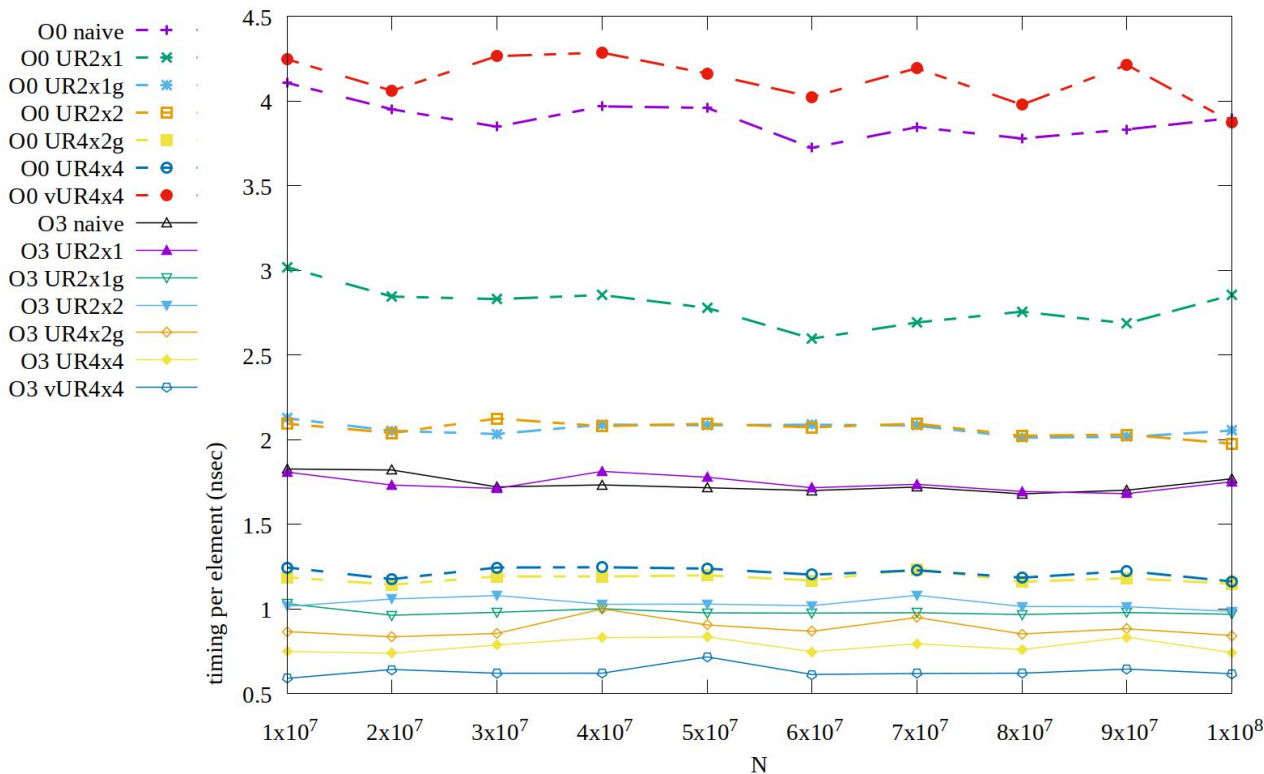
UR NxM: unrolled N times using M accumulators.

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 (timing-per-element)

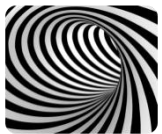


Run time of different implementation with and without compiler's optimization

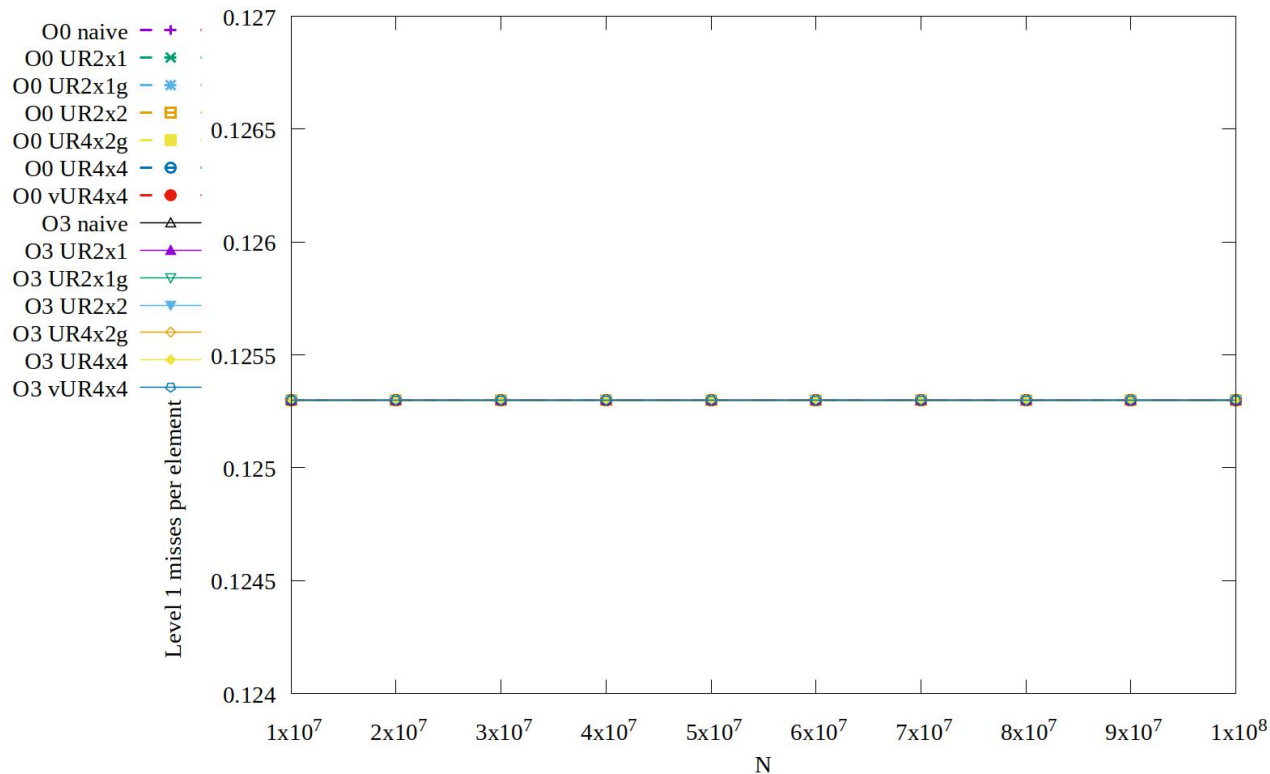
UR NxM: unrolled N times using M accumulators.

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)

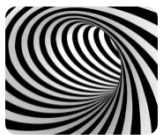


Reduction: results

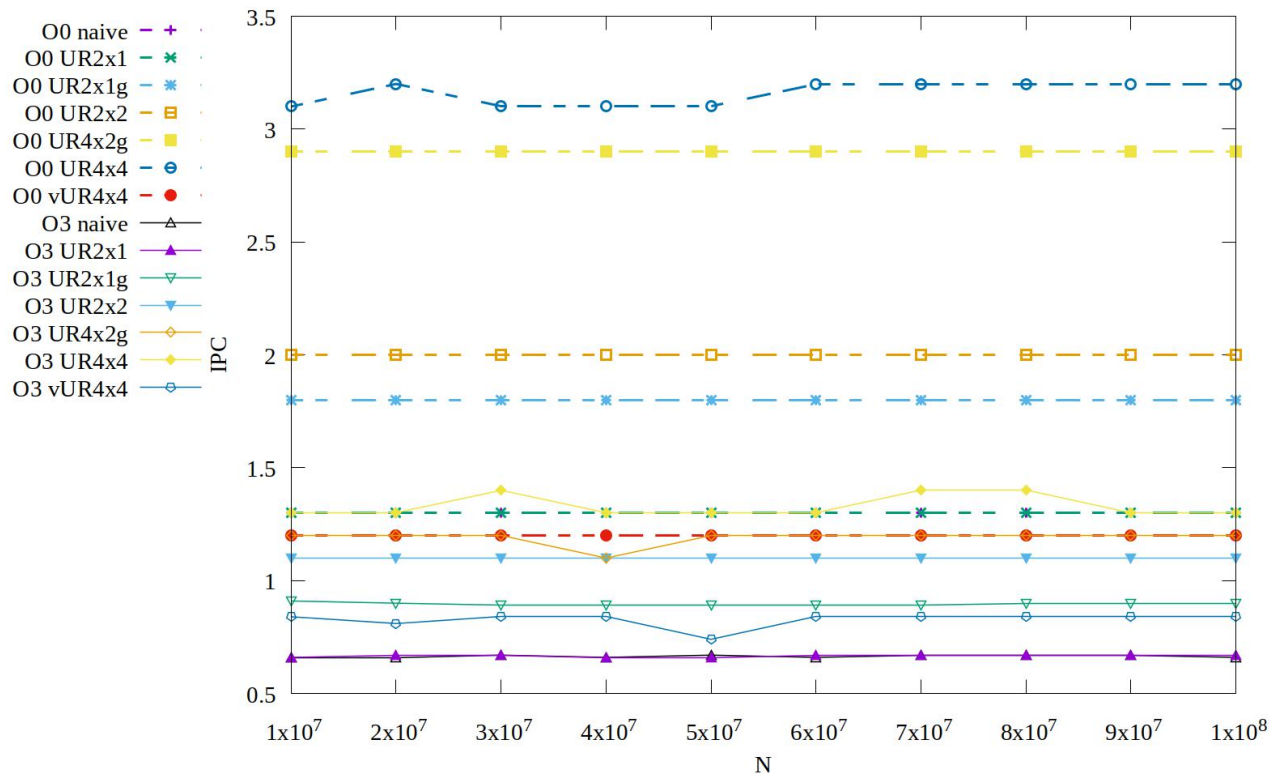


Level 1 Data cache misses

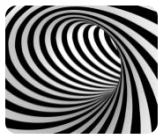
of course, we get the expected $1/8$ since we are processing the array continuously.



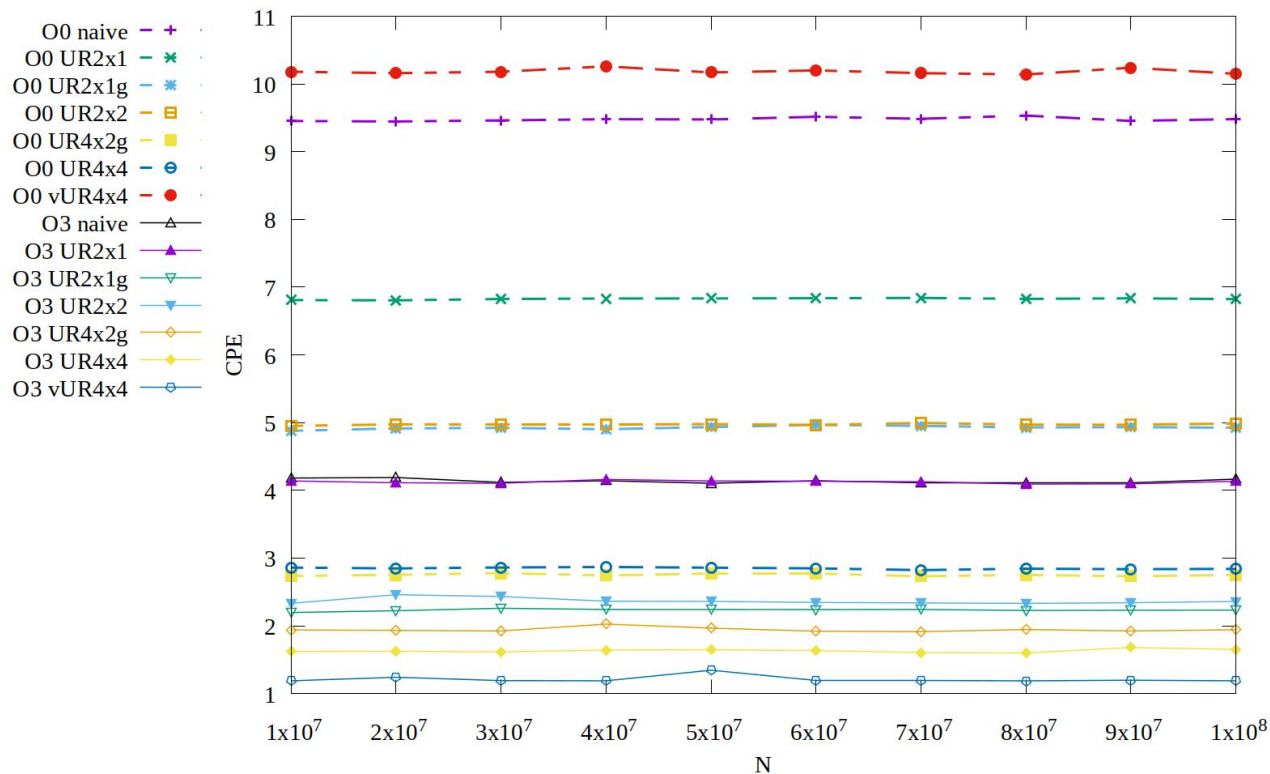
Reduction: results

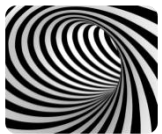


Instructions per cycle



Reduction: results



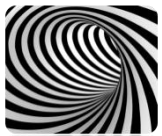


| Cache access in loops: $O(N^3)/O(N^2)$

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

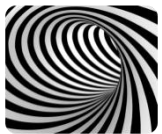
matrix-matrix multiplication is a very common task in HPC.

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

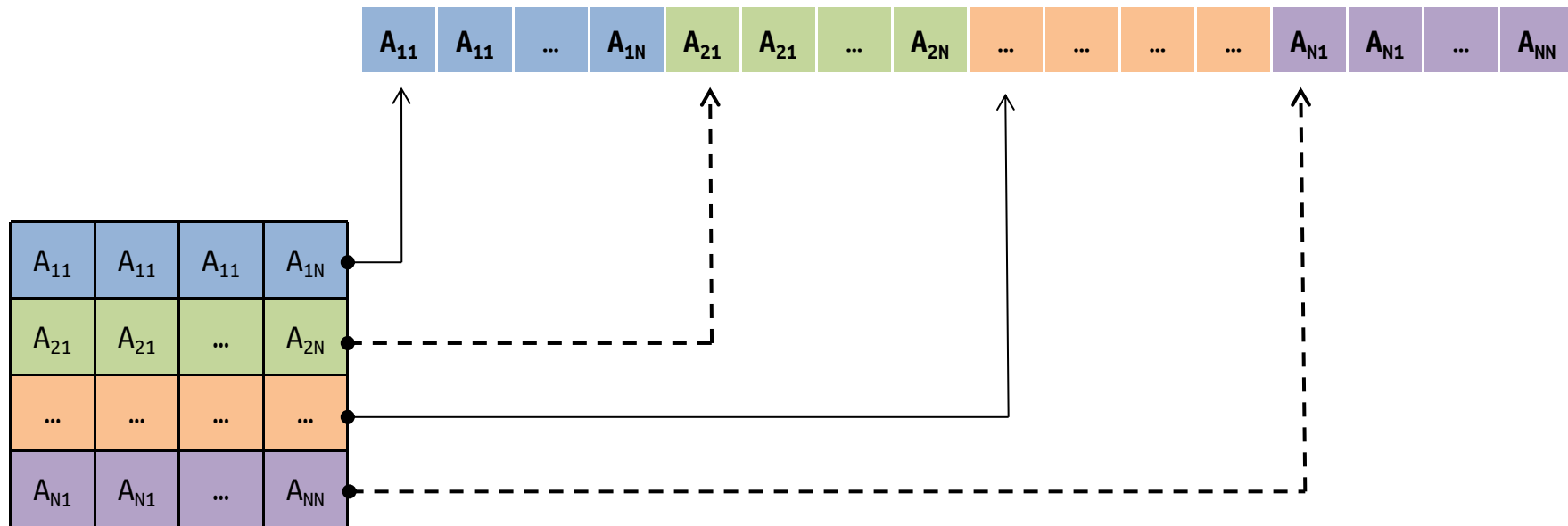
$$C_{i,j} = \sum_{k=0}^n A_{i,k} \times B_{k,j}$$

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.

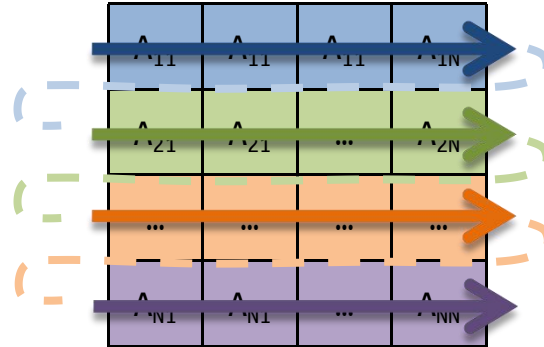


This convention is the C/C++ convention, which is labelled as *row-major order*. Note that the Fortran convention is opposite, with columns being contiguous in memory (*column-major*).

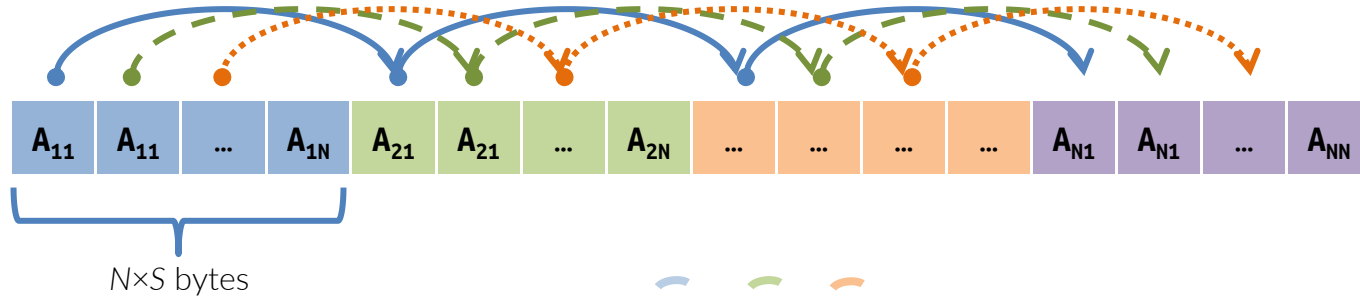
Note: how a matrix is stored in memory



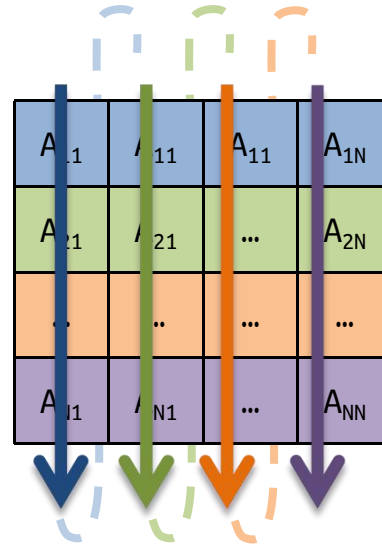
Then, traversing the matrix in the same row-major order amounts to traverse the memory in contiguous order.

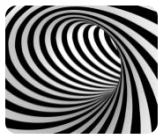


Note: how a matrix is stored in memory



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





Loops

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

Matrix representation

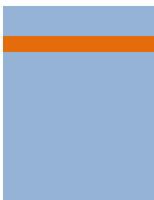
Memory representation

C



=

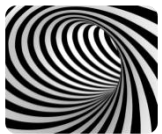
A



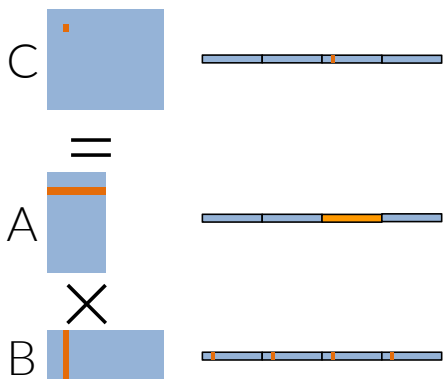
×

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/L 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:

$$\begin{aligned} & mp/L + \\ & mnp/L + \\ & mnp \end{aligned}$$

C is traversed once

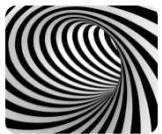
A is scanned entirely p times

B is accessed sparsely p times

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 (A_c 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

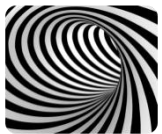
mnp/L	+	running over C
mnp/L	+	running over A
mnp/L		running over B

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

$\sim mnp$

less cache misses.



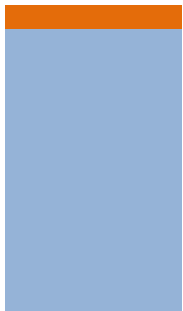


| $O(N^3)/O(N^2)$ example

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



=



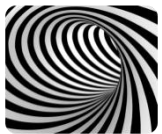
×



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.

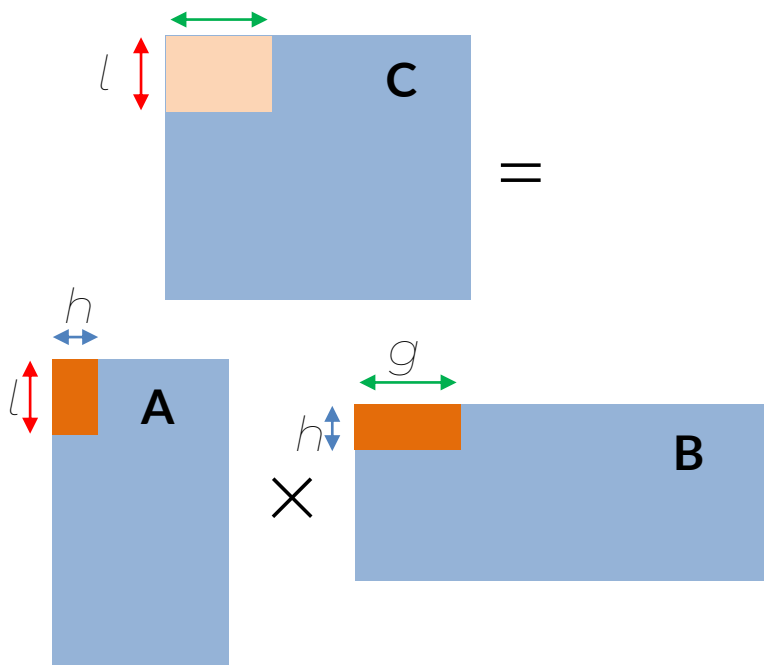
This amounts to have n/L compulsory misses per each column of B, i.e. np/L cache misses for each C's line, as we have already calculated.

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:

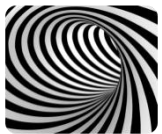


If, instead, we keep in the cache a segment of the A's line, re-using it against the columns of B – or, better, against a columns section tall as the line segment is large (blue arrows in the figure) – we could greatly reduce the amount of cache misses per C's element.

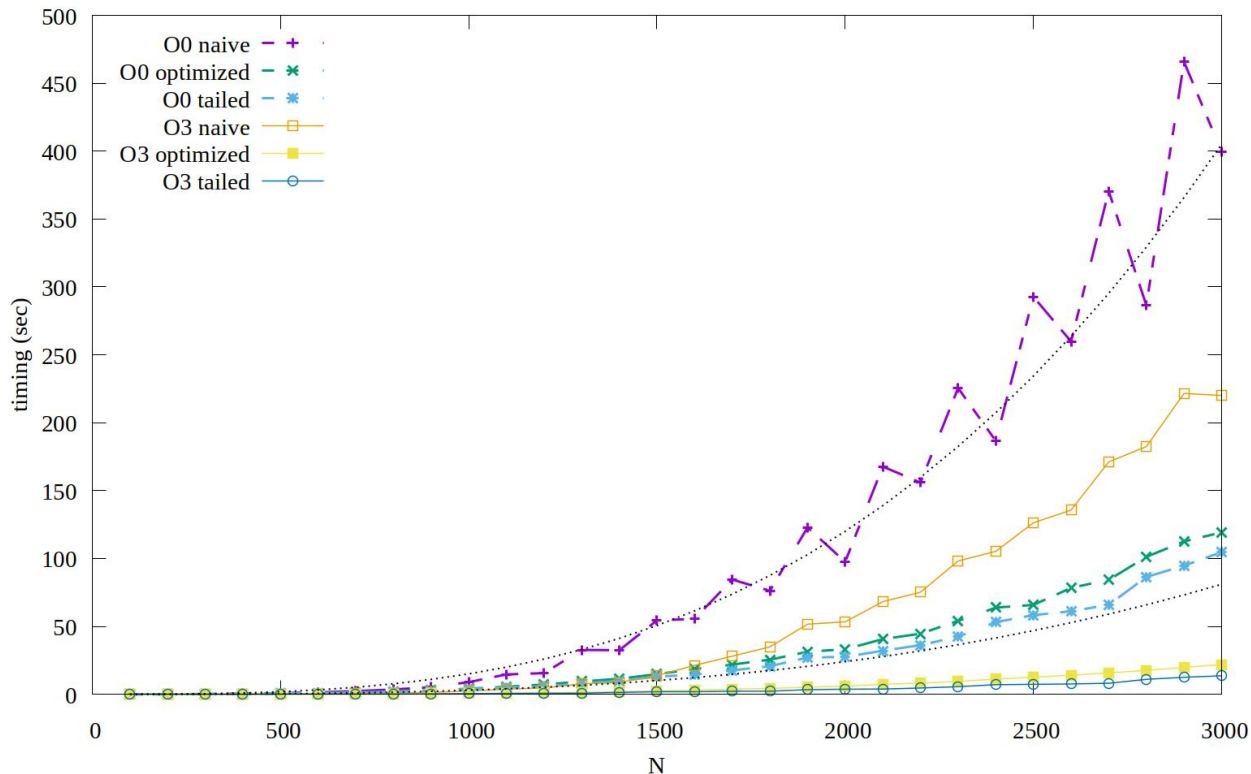
Traversing A and B by *blocks* as in the figure allows to accumulate partial results in the corresponding C's area while decreasing the number of cache misses by a factor

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

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

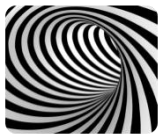
the results are for the case of 2 square matrix of dimension N

Naïve: the schoolbook's implementation

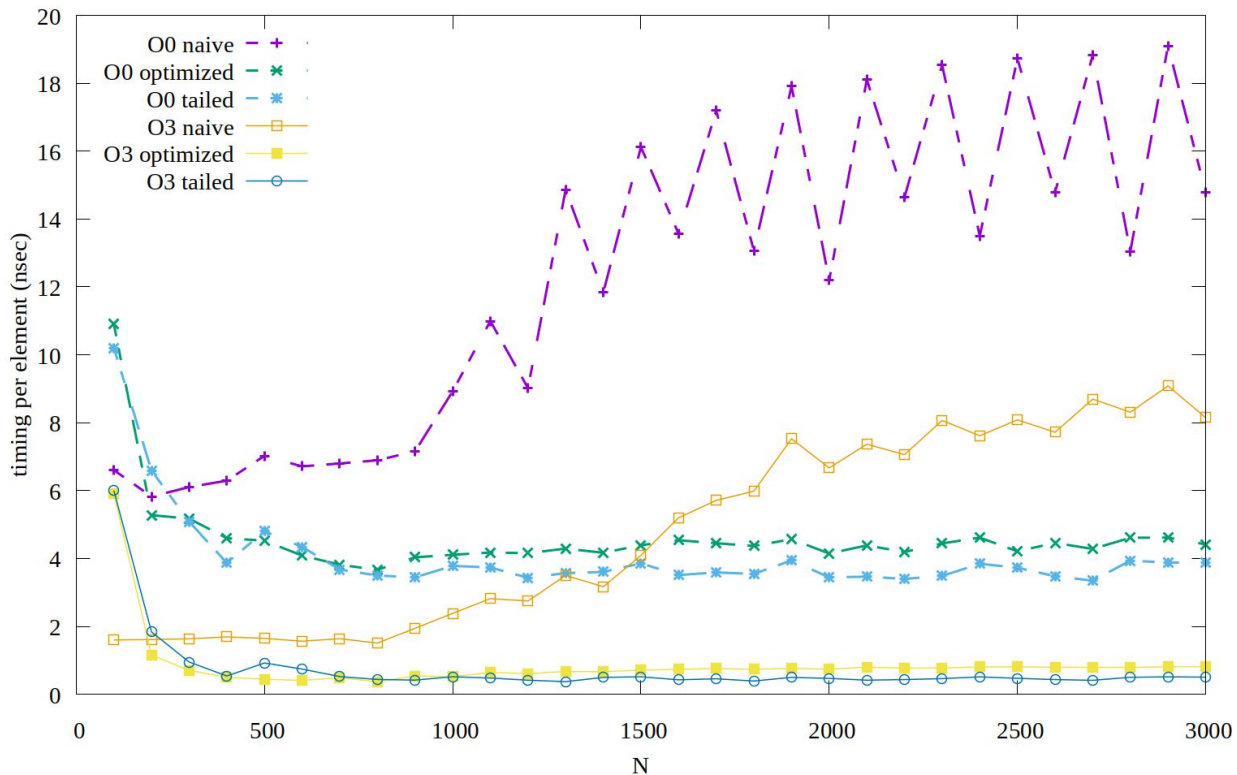
Optimized: inner loops [swapped](#)

Tailed: M-M by blocks

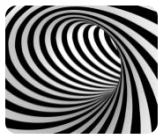
In this plot and in the following ones dashed lines are for `-O0` and solid line for `-O3 -march=native` (only gcc has been used)



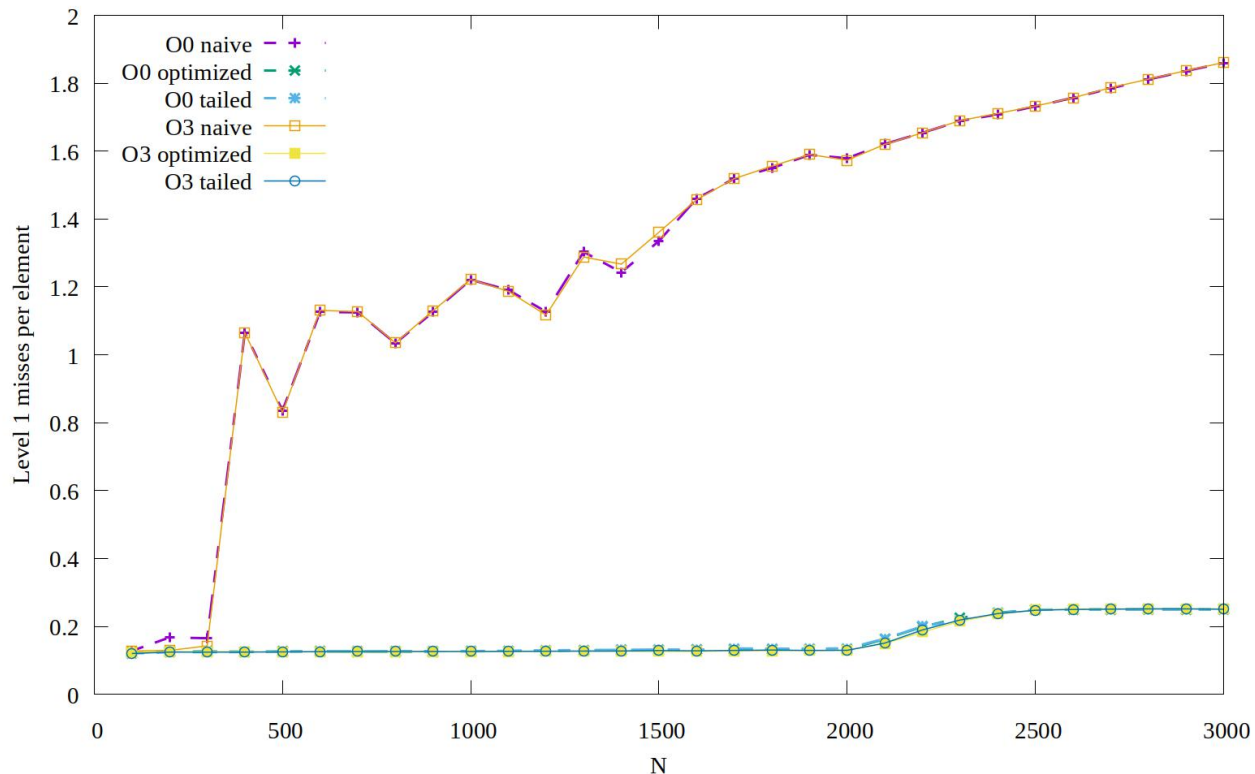
Matrix multiplication results



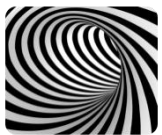
**Time per element
accessed (N^3)**



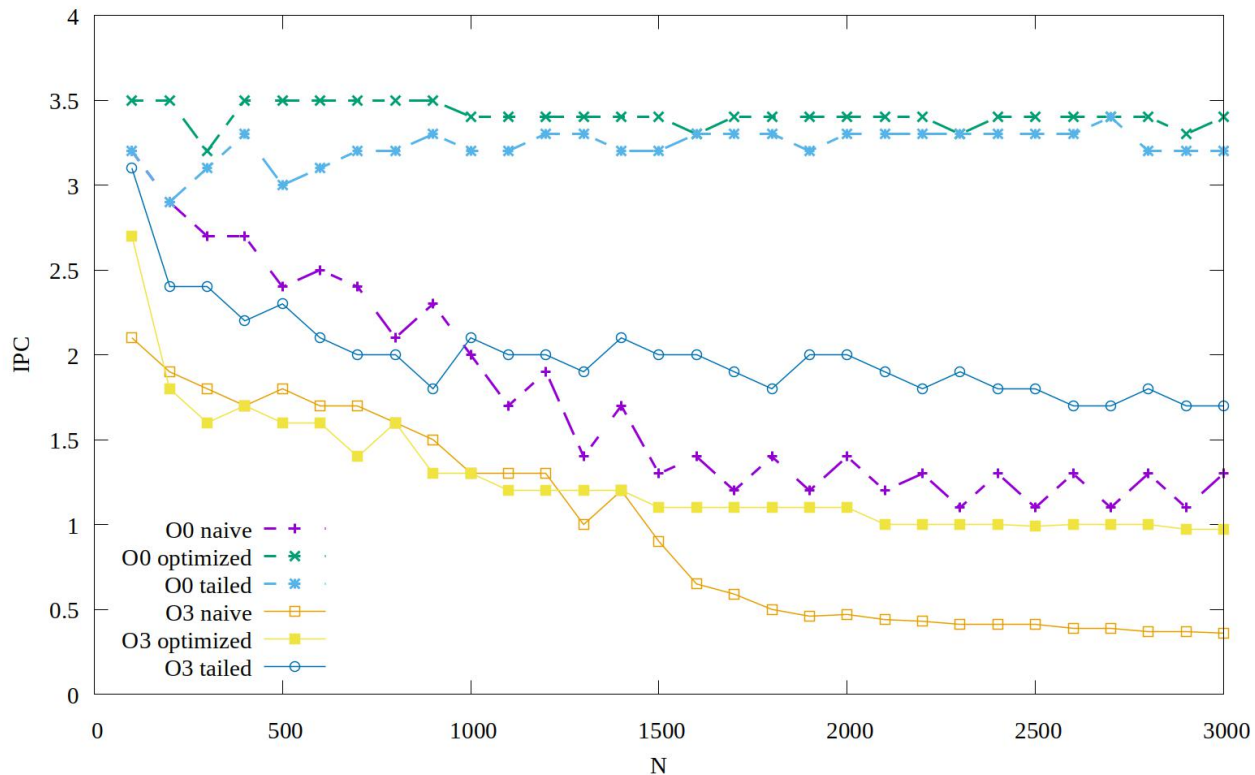
Matrix multiplication results



Level 1 Data cache misses per element

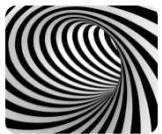


Matrix multiplication results

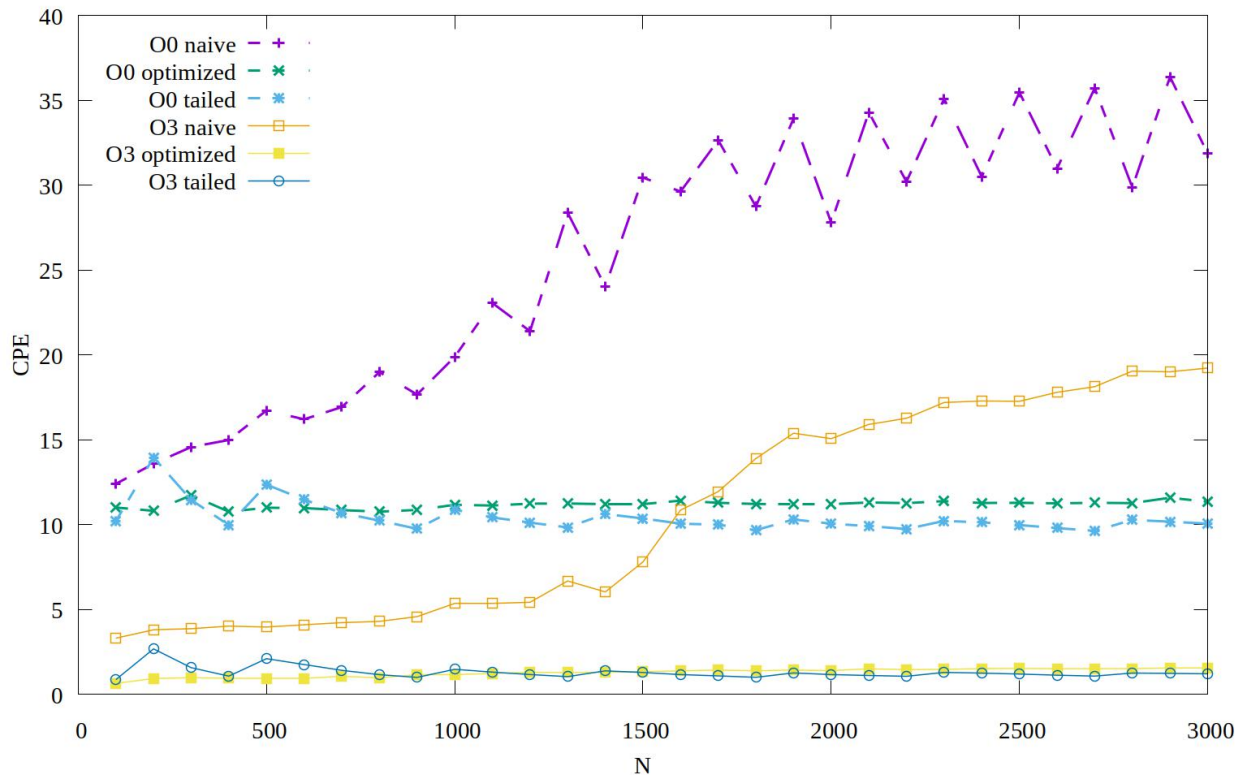


Instructions per cycle

(the larger the better)



Matrix multiplication results



Cycles per element

(the smaller the better)

Loops



Outline



Advanced



Prefetching



At the right moment, at the right place

Advanced

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 (typically the compiler insert pre-fetching instructions at compile-time).



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

Advanced

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

Advanced

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

Advanced

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; }
```



SC0/examples_on_prefetching/
examples_on_prefetching_1



Explicit prefetching

Advanced

```
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
```



Prefetching

Explicit prefetching

Advanced

Samples: 71K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 1396140

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

Advanced

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”

Advanced

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;
}
```



SC0/examples_on_prefetching/
examples_on_prefetching_2



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 ];
```



[SC0/examples_on_prefetching/
examples_on_prefetching_2](#)

that's all, have fun



“So long
and thanks
for all the fish”