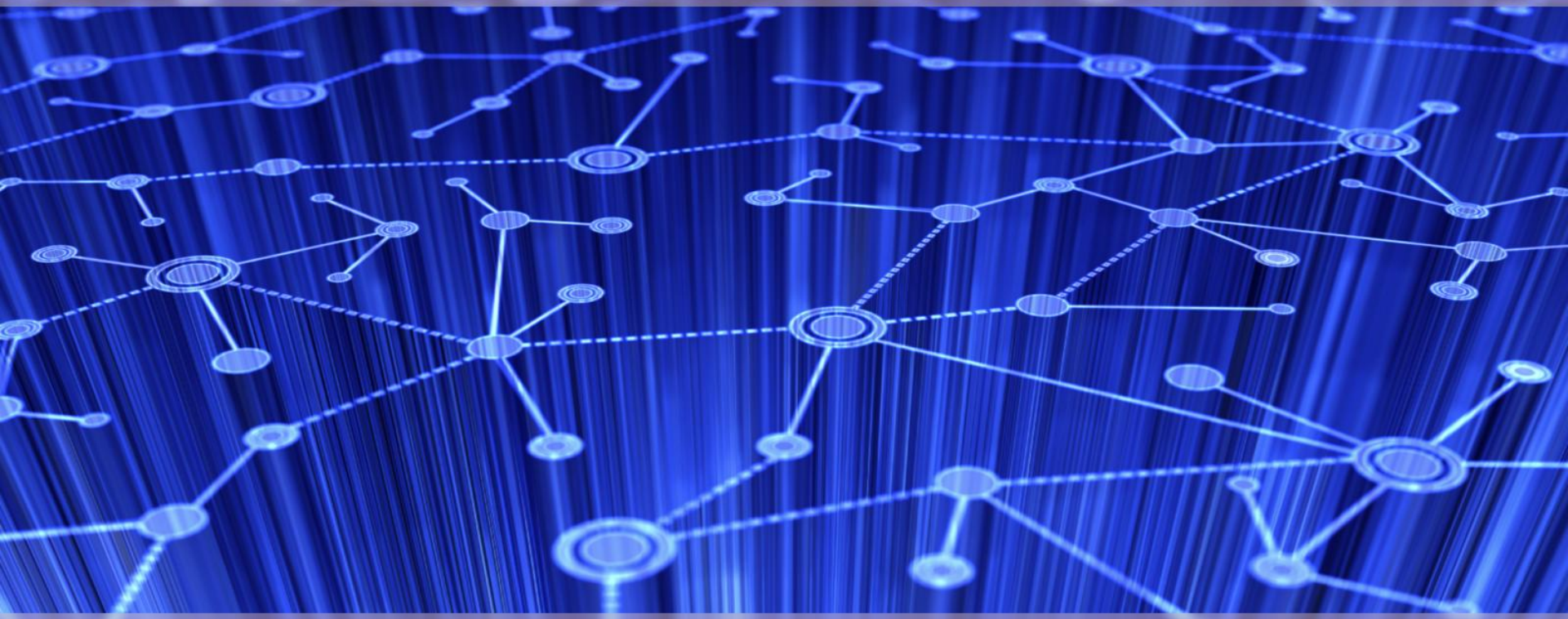


# Foundations of High Performance Computing

**Luca Tornatore**

## Code optimization

## *Part III – Coding*

# Outline

General recap and some more things..

- **Memory** optimizations
- **Loops** optimizations
- Other optimizations

# First things first

# First things first

"Optimization" may be a tag for several different concepts, as we have seen at the beginning of this course.

Many concurrent facts and factors must be kept together, and it is quite difficult to give general statements about which ones are more fundamental.

Top-level design plays a key role, as well as algorithm choice and implementation details.
Sometime even a single line of code – for instance a prefetching – may have brilliant consequences

...

# First things first

However, unless you are seeking some extreme performance, as a general guideline just keep in mind that “optimization” reads

“let the compiler squeeze the maximum from your code”

Compilers are quite good indeed, and know the hardware they are running on better than you (99% of times).

So, as first, just learn how to :

- write non-obfuscated code
- design a good data structure layout
- design a “good” workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

# First things first

- write non-obfuscated code
  → avoid memory aliasing
  → make it clear what a variable is used for and when

- design a good data structure layout
- design a “good” workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

# First things first

- write non-obfuscated code
- design a good data structure layout
  → be cache-conscious
  → be NUMA-conscious
  → avoid race conditions in multi-threaded codes
- design a “good” workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

# First things first

- write non-obfuscated code
- design a good data structure layout
- design a “good” workflow
  → compiler will be able to optimize branches and memory access patterns
  → prefetching will work better
  → make it easier to use multi-threading
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

# First things first

- write non-obfuscated code
- design a good data structure layout
- design a “good” workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures
  → let the compiler exploit pipelining through operation ordering and unloop
  → let the compiler exploit the vectorization capabilities of CPUs
  → think task-based, data-driven

# Some C-specific hints

## Storage class

May load/store latency of heavily used variables

- `extern`
  Global variables, they exist forever
- `auto`
  Local variables, allocated on the stack for a limited scope, and then destroyed. They must be initialized
- `register`
  Suggests that the compiler puts this variable directly in a CPU register

## Variable qualifiers

- `const`
  Global variables, they exist forever
- `volatile`
  Indicates that this variable can be accessed from outside the program.
- `restrict`
  A memory address is accessed only via the specified pointer

# Some C-specific hints

:: note about **`restrict`** qualifier ::

```
void my_function( double *a, double *b, int n)
{
        for( int i = ; i < n; i++ )
                a[ i ] = s * b[ i - 1 ];
}
```

The compiler can not optimize the access to **`a`** and **`b`** because it can not assume that **`a`** and **`b`** are pointing to the same memory locations.
That is called *aliasing*, formally forbidden in fortran: which is the reason why in some cases fortran may compile in faster executables.

Help your C compiler in doing the best effort, either writing a clean code or using restrict or using **`-fstrict-aliasing -Wstrict-aliasing`** options.

→ code snippet :: **`pointers_aliasing.c`**

# Some C-specific hints

## Memory allocation

We have already discussed that… try to **allocate contiguous memory** in large bunch, and to **reuse it** efficiently.

## C++

That is a big topic, there's a lot of literature on it.
You're learning it in a dedicated course, so you know about that.
Let me just stress that a common illusion among non-professionals about C++ is that the compiler can clearly seen through all the abstraction that an advanced C++ programming style contains.

However, C++ should be preferably seen as a language that enables complexity management: most of its sophisticated features are not well suited for a extremely efficient low-level code.

# Use of the compiler

## Language standard

Some constructs can be better implemented if you specify a specific standard to the compiler.

## Architecture specification

The compiler is able to detect the architecture it runs on, but – to maximize executable's compatibility – it will not turn on very specific optimization unless you tell him so.

Read *carefully* the manual of your compiler.
In `gcc`, for instance, `-march=native` turns on a lot of sophisticated optimizations

# Use of the compiler

## Optimization level: -O*n*

It is not granted that `-O3`, although often generating a faster code, is what you really need.
For instance, sometimes expensive optimizations may generate more code that on some architecture (e.g. with smaller Icaches) run slower, and using `-Os` may bring surprising results.

Take into accounts that modern compilers allows for local specific optimizations or compilation flags. In gcc for instance:

```
__attribute__ ((__option__ ("…")))
__attribute__ ((optimize(n)))
```

## Special options

Not all the optimizations are enabled via `-O`*n*, even for the highest *n*.
Check your manual.

# Use of the compiler

## Profile-guided optimization

Compilers (**`gcc`**, **`icc`** and **`clang`**) are able to instrument the code so to generate run-time information to be used in a subsequent compilations. Knowing the typical execution patterns enables the compiler to perform more focused optimizations, especially if several branches are present.

For **`gcc`**:

```
gcc -fprofile-arcs
< … run … >
gcc -fbranch-probabilities
```

Specific for branch prediction

```
gcc -fprofile-generate
< … run … >
gcc -fprofile-use
```

More general; enables also `-fprofile-values -freorder-functions`

# Outline

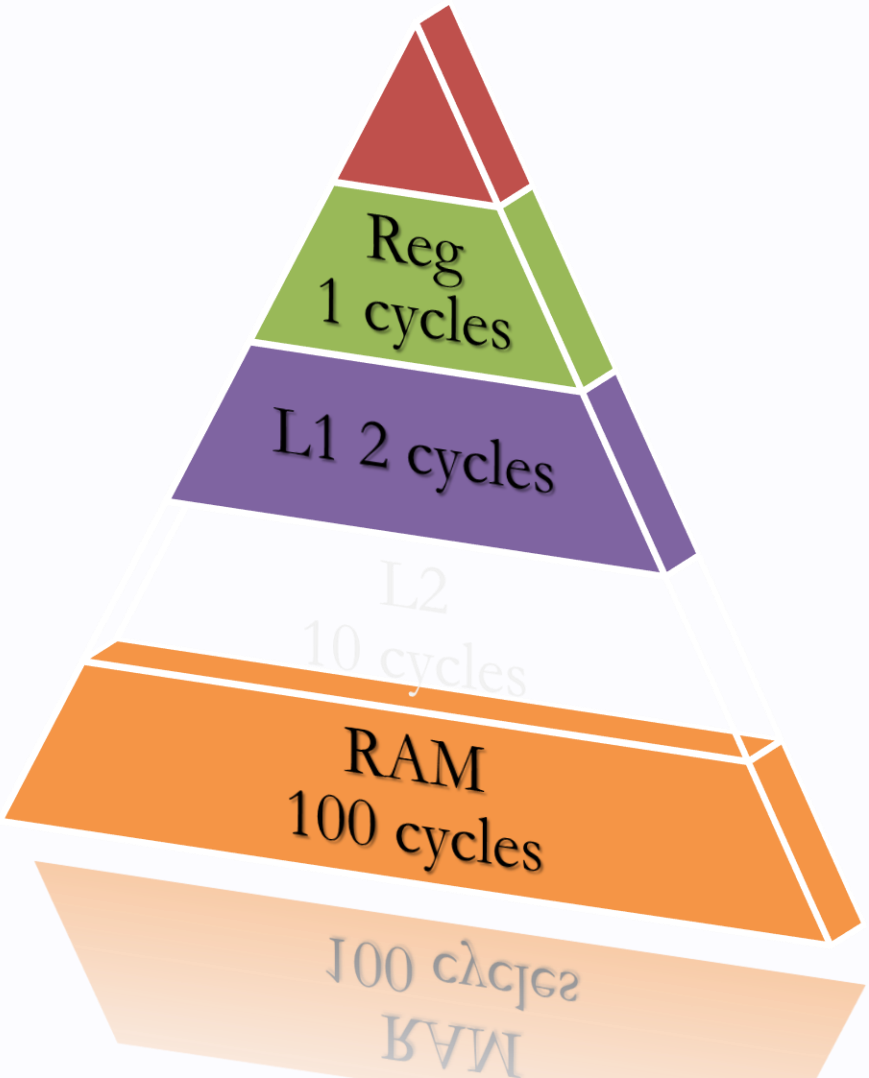
- **Memory** optimizations
  - > the importance of memory access
  - > the importance of the cache
  - > cache access optimization in loops
  - > cache-oblivious algorithms
  - > cache-oblivious data structures
- **Loops** optimizations
- Other optimizations

# The memory access

*Again on cache*

# The importance of memory access

Impact of cache misses on average memory access time



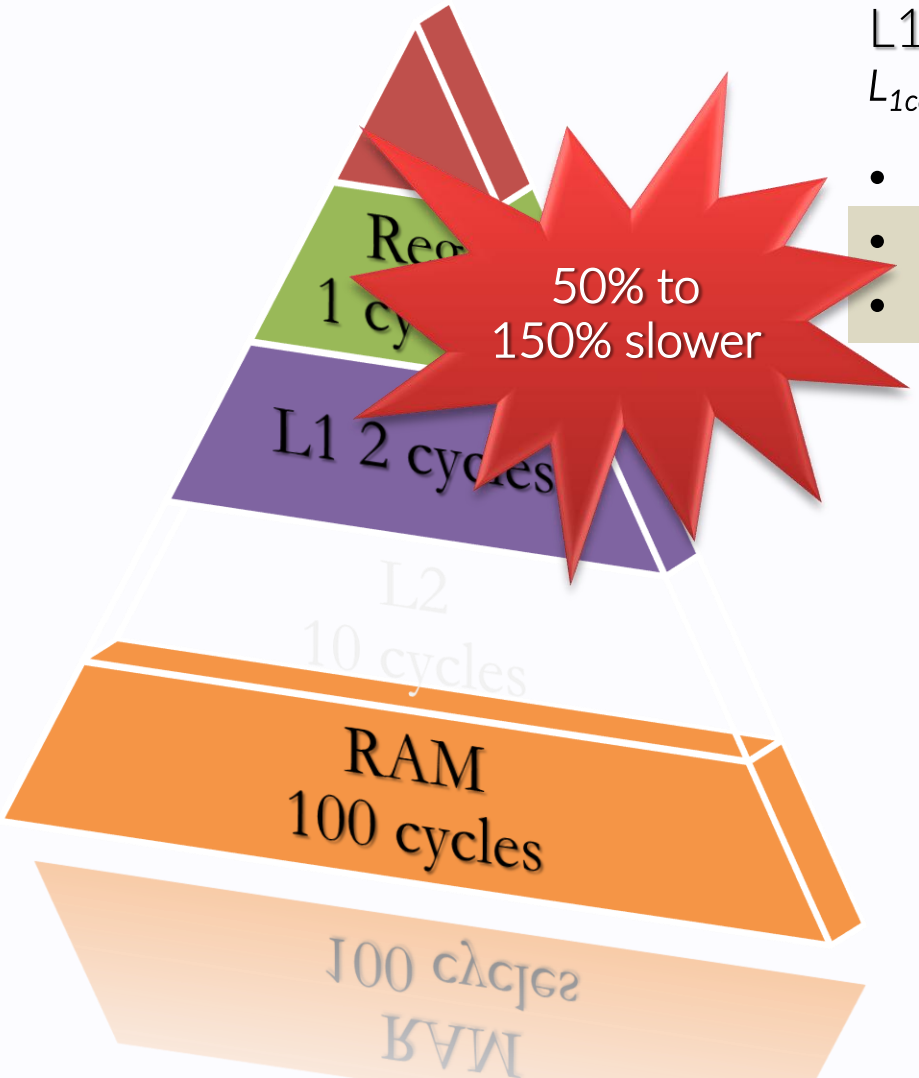

L1 cache + RAM

$L_{1cost} + Miss_1 \times RAM_{cost}$

- 100% L1 hit → 2 cycles
- 99% L1 hit → 3 cycles
- 97% L1 hit → 5 cycles

# The importance of memory access

Impact of cache misses on average memory access time



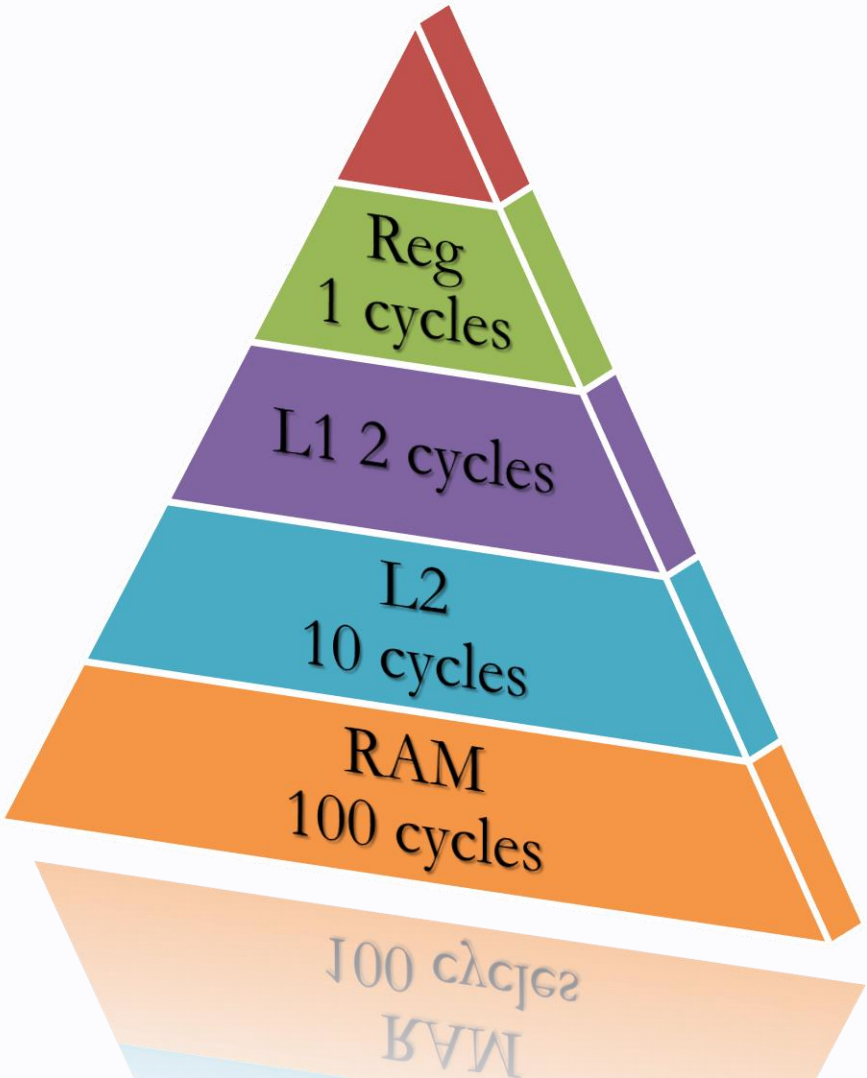

L1 cache + RAM

$L_{1cost} + Miss_1 \times RAM_{cost}$

- 100% L1 hit → 2 cycles
- 99% L1 hit → 3 cycles
- 97% L1 hit → 5 cycles

# The importance of memory access

## Impact of cache misses on average memory access time




L1 cache + RAM

$L_{1cost} + Miss_1 \times RAM_{cost}$

- 100% L1 hit → 2 cycles
- 99% L1 hit → 3 cycles
- 97% L1 hit → 5 cycles

L1 cache + L2 cache + RAM

$L_{1cost} + Miss_1 \times (L2_{cost} + Miss_2 \times RAM_{cost})$

- 100% L1 hit → 2 cycles
- 99% L1 hit, 100% L2 hit → 2.1cycles
- 97% L1 hit, 100% L2 hit → 2.3 cycles
- 90% L1 hit, 97% L2 hit → 3.3 cycles

What would happen if you had not separate L1 caches for data and instructions?

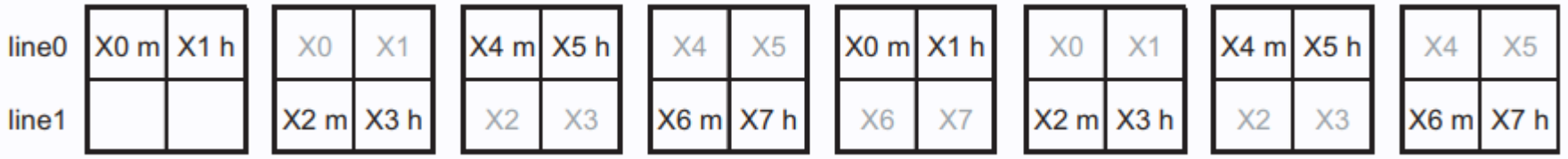
# The importance of cache

## When the cache is hit and when it is not: a simple model

Consider a simple direct mapped **16 byte data cache** with **two cache lines**, each of size 8 bytes (two floats per line)

Consider the following code sequence, in which the array `X` is cache-aligned (that is, `X[0]` is always loaded into the beginning of the first cache line) and accessed twice in consecutive order:

```
float X[8];
for(int j=0; j<2; j++)
   for(int i=0; i<8; i++)
     access(X[i]);
```

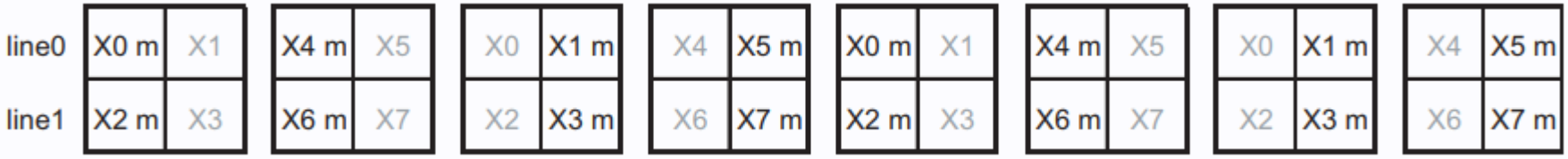



The hit-miss pattern is : MH MH MH MH MH MH MH MH,
the miss-rate is 50% (the first miss is compulsory miss).

# The importance of cache

Let's consider another code sequence that access the array twice as before, but with a strided access

```
float X[8];
for(int j=0; j<2; j++)
   {
      for(int i=0; i<7; i+=2)
         access(X[i]);
      for(int i=1; i<8; i+=2)
         access(X[i]);
   }
```

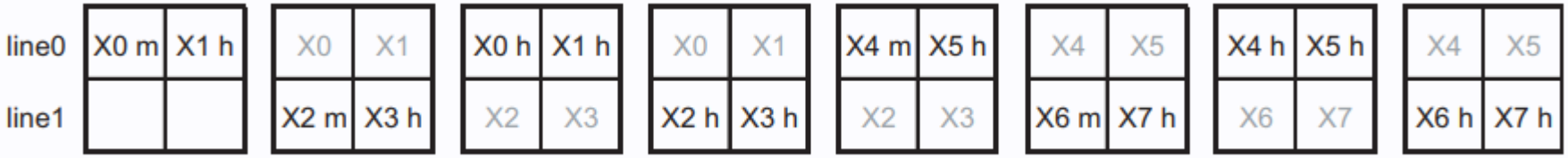



The hit-miss pattern now is : MM MM MM MM MM MM MM MM, the miss-rate is 100%

# The importance of cache

Finally, consider a third code sequence that again access the array twice:

```
for(int i = 0; i < 2; i++)
   for(int k = 0; k < 2; k++)
      for(int j = 2*i; j < (i+1)*2; j ++)
         access(X[ j ]);
```

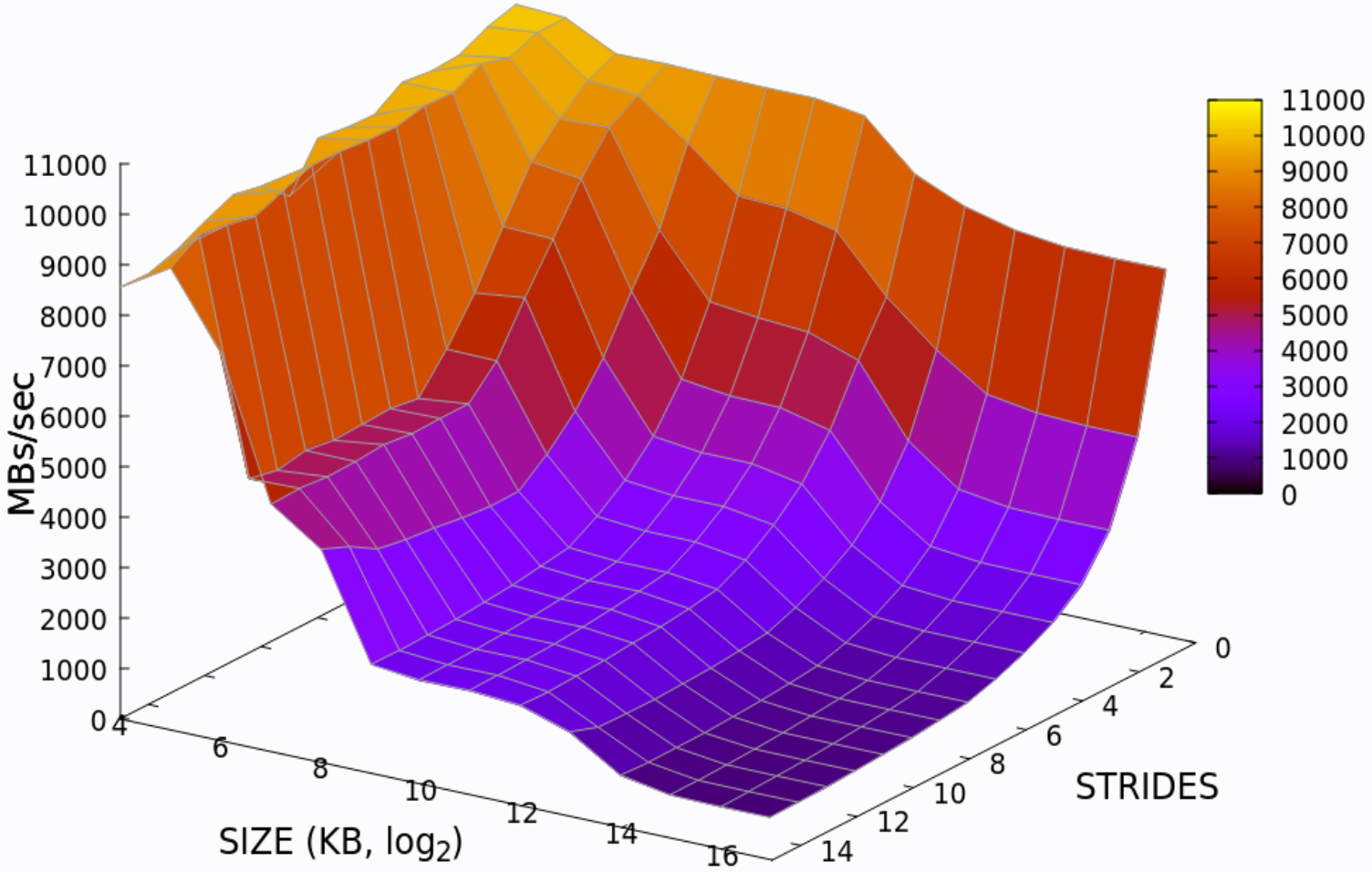



The hit-miss pattern now is : MH MH HH HH MH MH HH HH,
the miss-rate is 25%

**The main message is: memory access pattern is of primary importance**

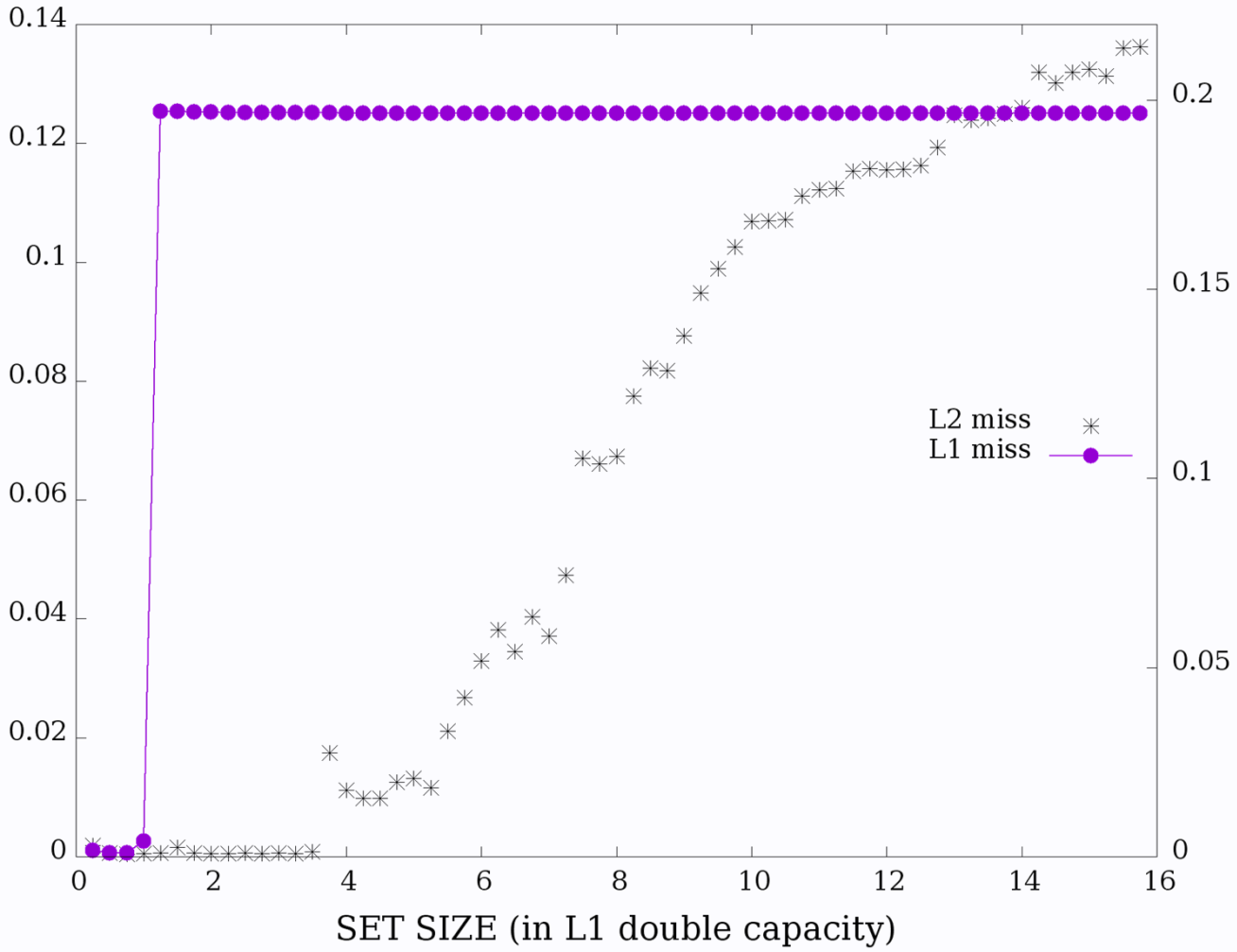
# The importance of cache

The result is.. the memory mountain



# The importance of memory access pattern

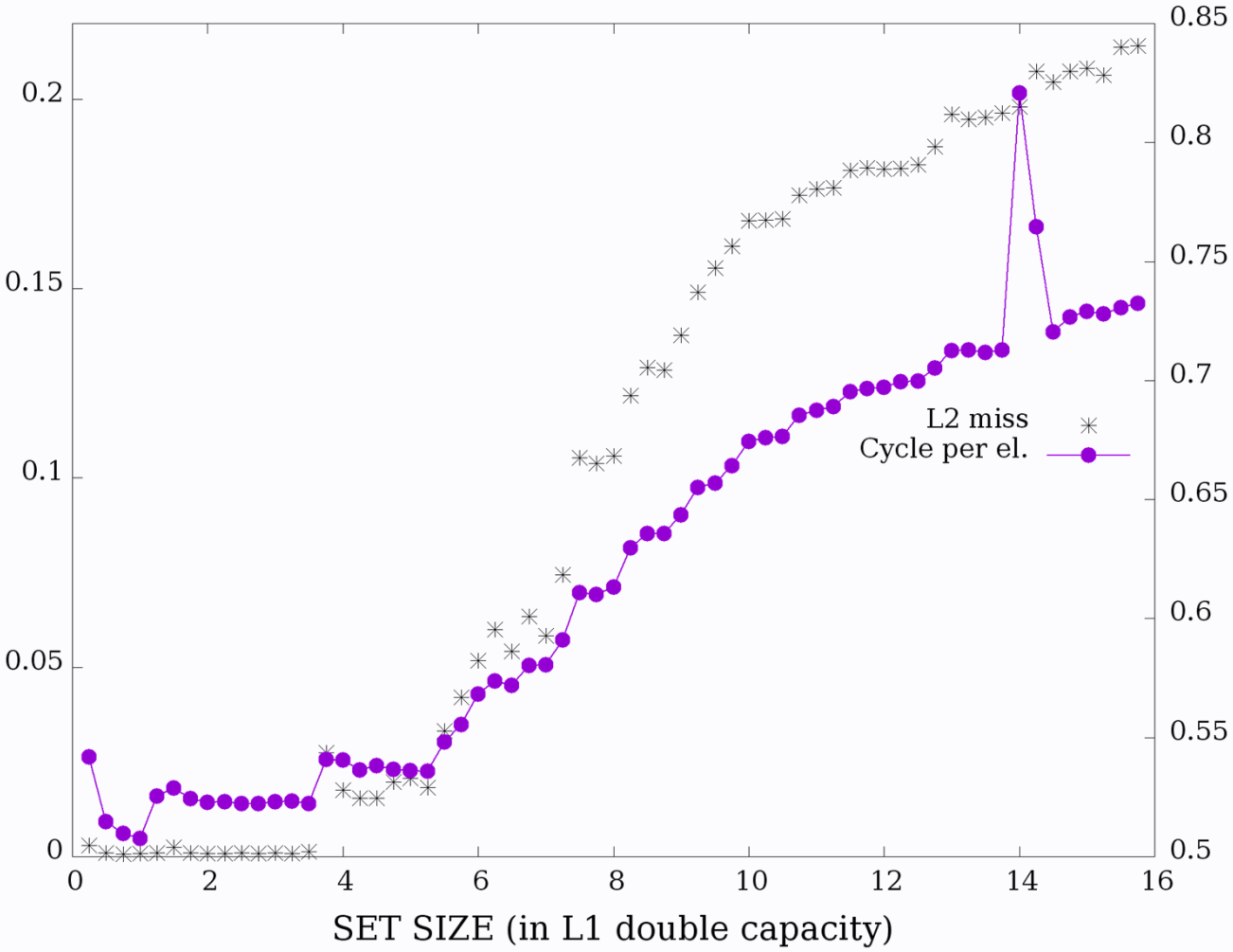
## Let's find our L1 – and L2– cache sizes




```
for (j=0; j<size; j++)
   array[j] =
     2.3*array[j]+1.2;
```

# The importance of memory access pattern

..and the effect on cycles-per-operation




```
for (j=0; j<size; j++)
    array[j] =
      2.3*array[j]+1.2;
```

# The importance of memory access pattern

A Worked example you have already seen: dense MATRIX TRANSPOSE

Naïve version:

```
for(int row = 0; row < N; row++)
    for(col = 0; col < N; col++)
        A [ col*N + row ] = B [ row*N + col ];
```

NOTE: strided access to either **A** or **B** is unavoidable.

However: is it better to have it either on *read* or on *write* ?

# The importance of memory access pattern

A Worked example you have already seen: dense MATRIX TRANSPOSE

Naïve version:

```
for(int row = 0; row < N; row++)
    for(col = 0; col < N; col++)
        A [ col*N + row ] = B [ row*N + col ];
```

NOTE: strided access to either **A** or **B** is unavoidable.

However: is it better to have it either on *read* or on *write* ?
Due to write-allocate transactions in the cache, strided writes are more expensive than strided loads.

# The importance of memory access pattern

## A Worked example: dense MATRIX TRANSPOSE

Let $C$ be the cache size and $L_C$ the cache line size, we should expect 3 different regimes:

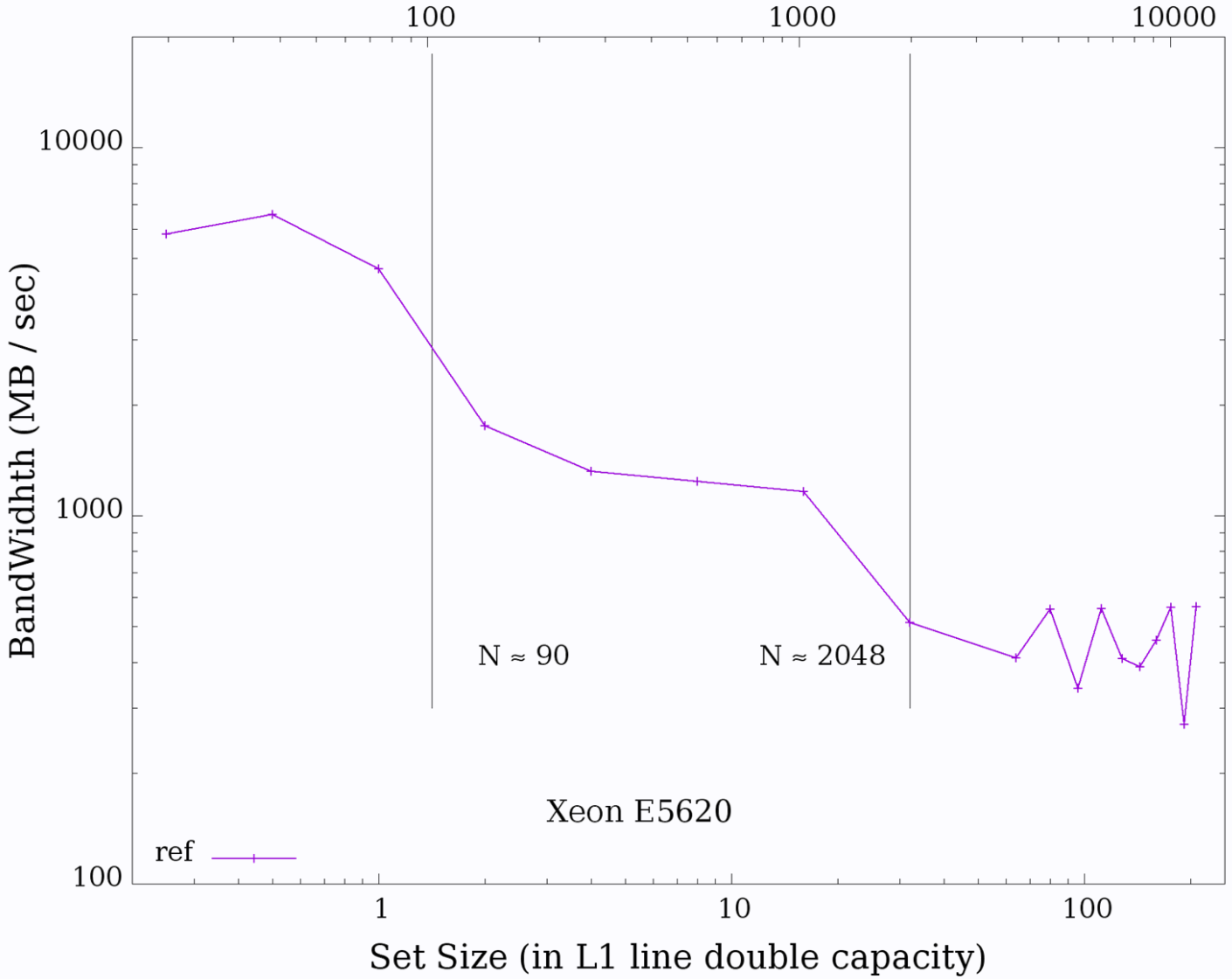
1. $2 \times N^2 < C$
   both matrices can fit in the cache, traversal order and locality does not impact on performance (bandwidth ~ maximum)

2. $N \times L_C < C$
   strided write is alleviated by fraction of column fitting in the cache

3. $N > C \times L_C$
   Each access to A determines a cache miss and a *write-allocate*.
   A sharp drop in performance is expected since basically only one entry per line will be used.

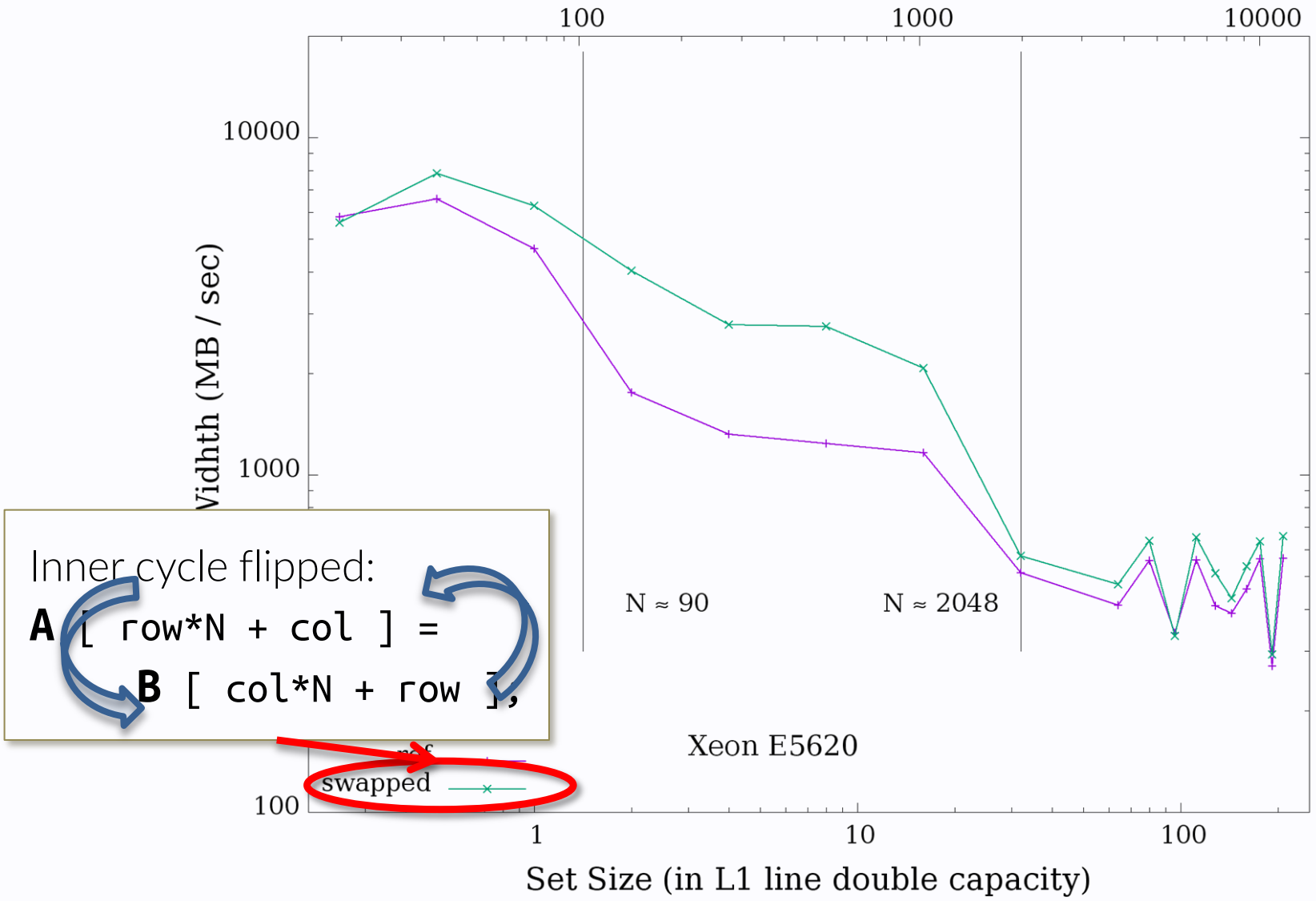
# The importance of memory access pattern

..more or less, you already know from previous lectures..



# The importance of memory access pattern

## Matrix transpose: avoiding strided access on write

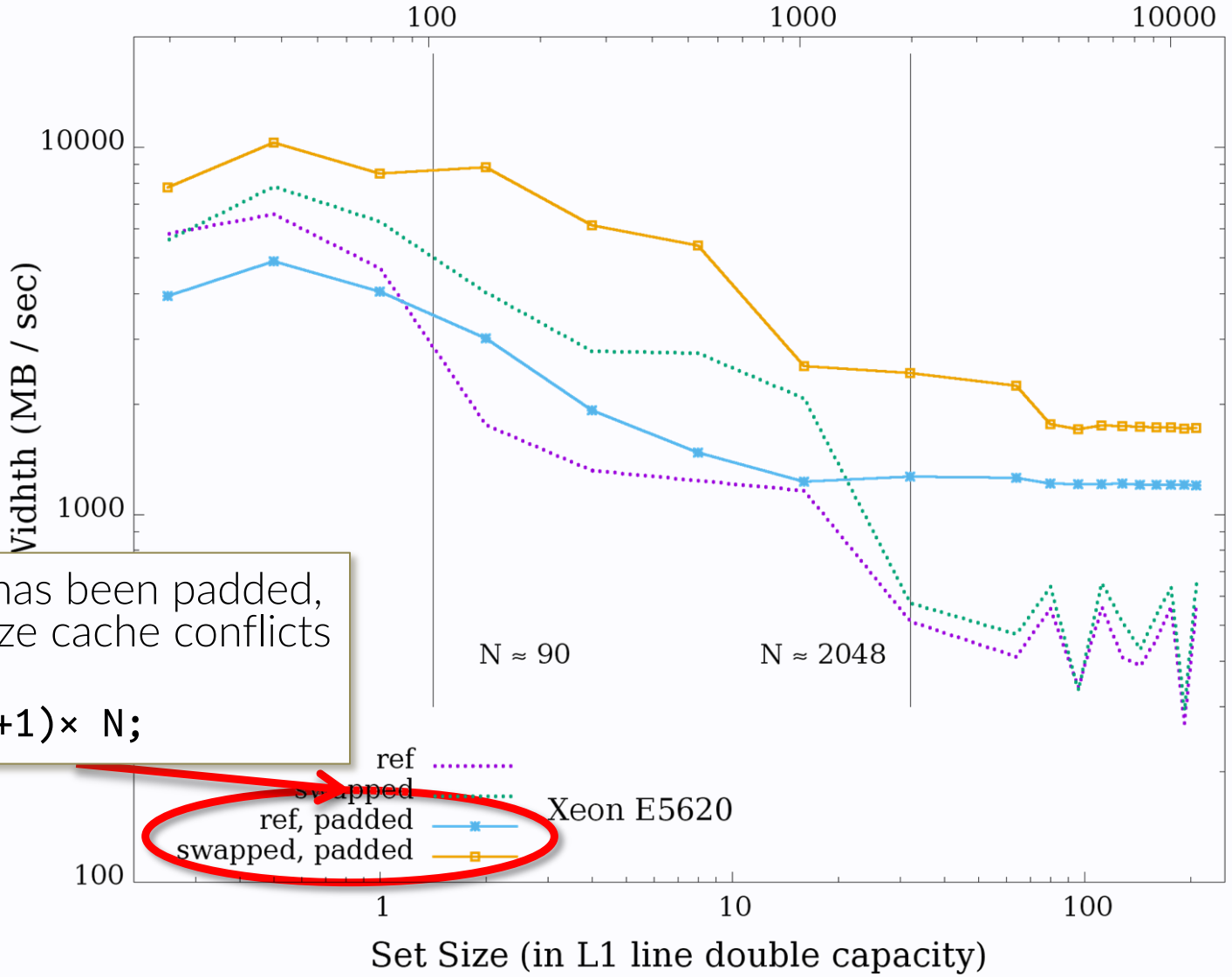



Inner cycle flipped:

```
A [ row*N + col ] =
    B [ col*N + row ];
```

# The importance of memory access pattern

## Matrix transpose: avoiding cache associativity conflicts




B layout has been padded, to minimize cache conflicts with A

```
B = (N+1)× N;
```

# Cache resonance

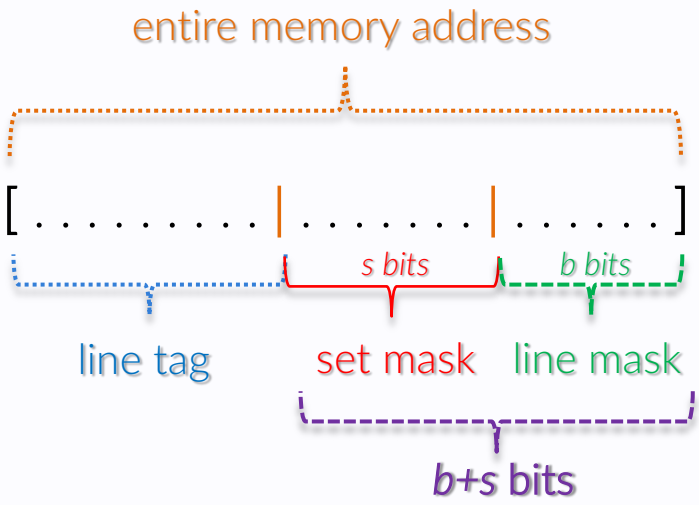
## Cache associativity conflicts (or “resonances”): what are they ?

You know that your cache of size $C$ bytes is $W$-way associative: it means that your cache is made up by lines of size $L$ bytes, grouped in $W$-sized sets.

Tipical figures nowadays are: $L$ = 64B (=$2^b$ bytes), $W$ = 4-8, $C$ = 32KB (i.e. there are $C/(L \times W)$ = 64-128 (= $2^s$ sets).

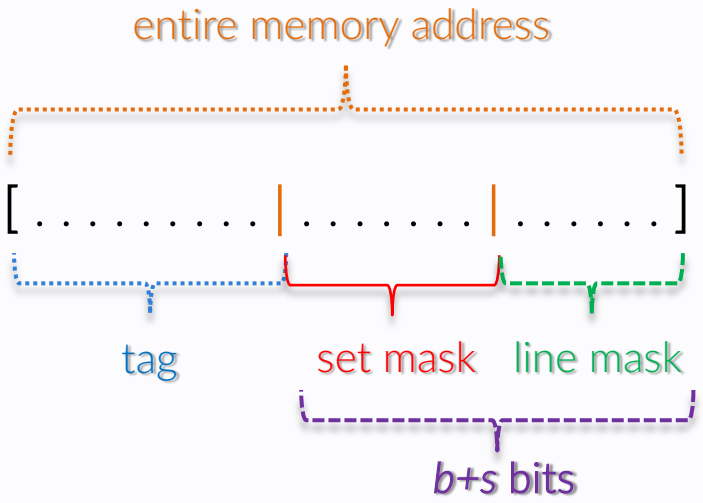
“ways” of associativity

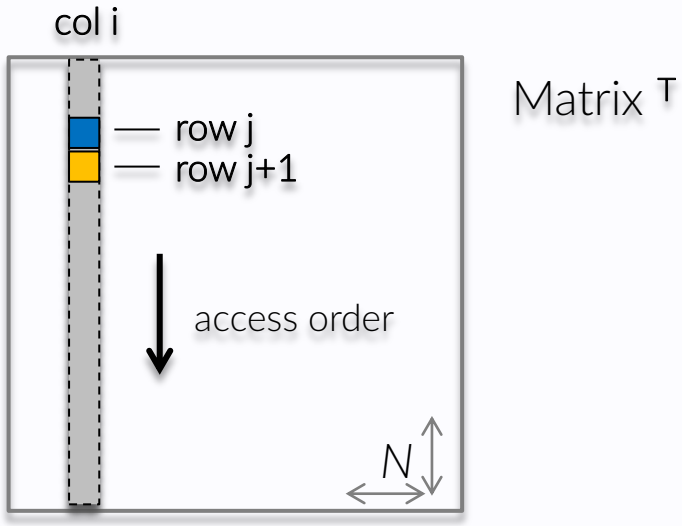
To map the memory addresses into the cache, the first $b+s$ bits are used to determine to what byte in a free line of what cache set that address will be stored.



# Cache resonance

## Cache associativity conflicts (or "resonances"): what are they ?




Accessing the element *i, j+1* of the transposed matrix, the stride with respect to previously accessed element *i, j* will amount to:

$$offset = N \times W \text{ bytes}$$

where *N* is the matrix size and W is the type size (e.g. double, 8 bytes).

If *N* is a power of 2, $N = 2^n$; if $W = 2^d$ bytes.

$$offset = 2^{n+d} \text{ bytes}$$

then if $n+d > b+s$, the address of *i, j+1* is mapped in the same set than *i, j* and then (since we assume $N \gg w$) at least every *w* accesses there is a cache conflict.

# Cache resonance: padding

Padding is a simple but effective cure for the issue of cache resonance.

Adding one column to the transposed matrix, which then is $(N+k) \times N$, means that when the element $i, j+1$ is accessed, the stride with respect to previously accessed element $i, j$ amounts to:

$$offset = (N+k) \times W \text{ bytes} = 2^{n+d} + k2^{d} \text{ bytes}$$

If $N$ is a power of 2, $N = 2^{n}$, with $W = 2^{d}$ bytes,

$$offset = 2^{n+d} + k2^{d} \text{ bytes}$$

then if

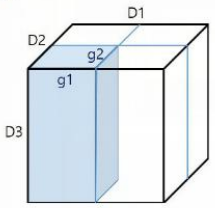
$$2^{b+s} < 2^{n+d} + k2^{d}$$

the address of $i, j+1$ is mapped on a different cache set, alleviating the cache conflicts

# The importance of data layout

## Cache associativity conflicts (or “resonances”): what are they ?

### 3.2 Padding for Set-associative Caches




To extend this result to the set-associative case for all $i$, $1 \leq i \leq d-1$, we introduce the characteristic number $g_i$ of dimension $i$ with respect to the cache size. Intuitively, as depicted (square at right) for $A = 4$, we will establish that if the enclosed tile $g_1 \times \cdots \times g_{d-1} \times D_d$ is free of self-interference conflicts in a direct-mapped cache, then the $A$-times larger tile $D_1 D_2 \cdots D_d$ is free of self-interference conflicts in an $A$-associative cache of the same capacity.

**Theorem 2** (Associative cache). *Consider a set-associative cache of capacity $C = SAB$. For all $1 \leq i \leq d-1$, let $g_i = \gcd(S/\prod_{1 \leq k \leq i-1} g_k, N_i)$. A loop nest whose tiles have a $d$-dimensional array footprint can fully utilize the cache and remain free of self-interference if and only if the following conditions are met:*

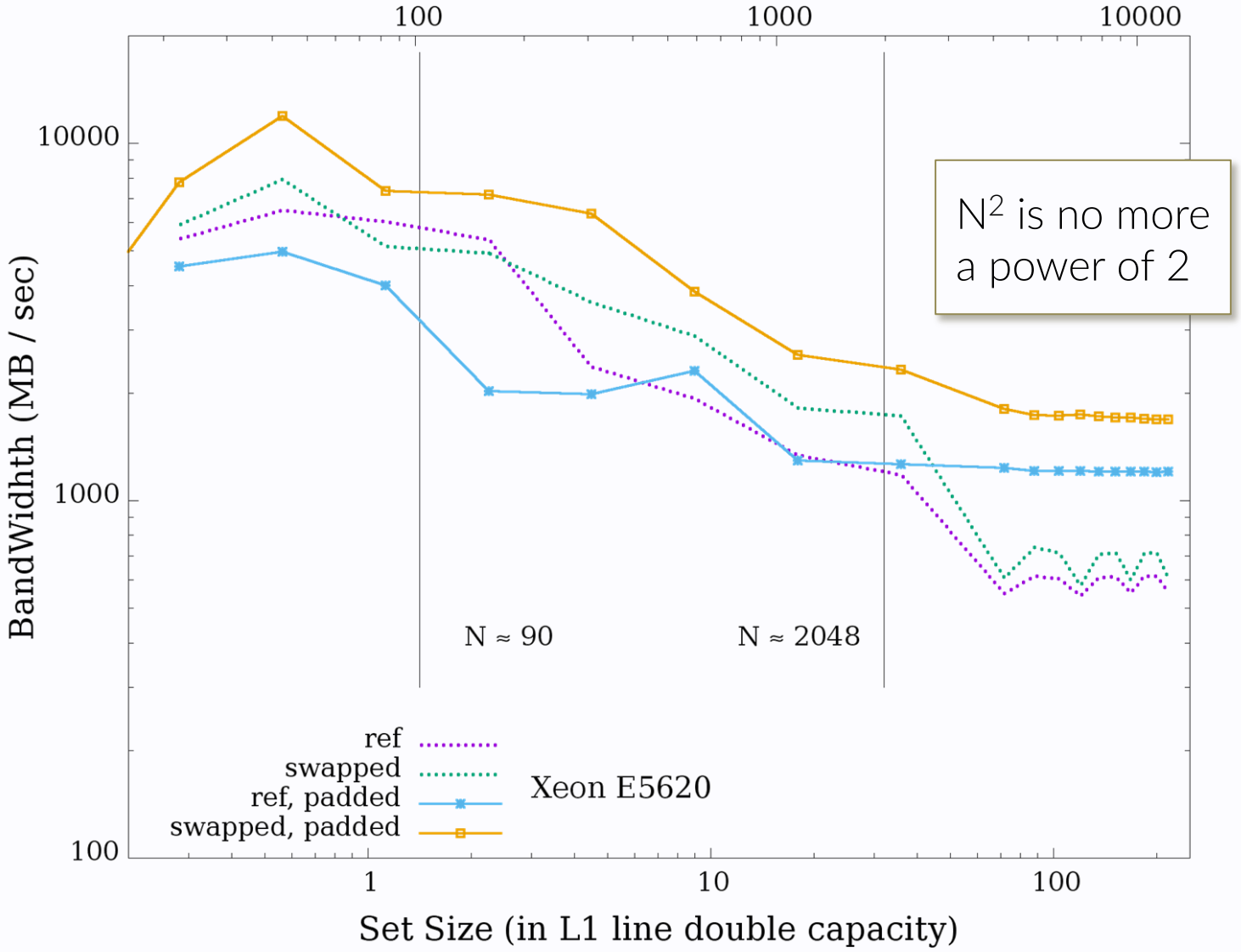
1. $\forall i, 1 \leq i \leq d-1$, $\exists j, 1 \leq j \leq i$, $\prod_{1 \leq k \leq i} g_k$ *divides* $D_j \prod_{1 \leq i \leq j-1} g_i$.
2. $\exists i, 1 \leq i \leq d$, $S$ *divides* $D_i \prod_{1 \leq k \leq i-1} g_k$.

People are doing a lot of impressive stuff and research about this issue.

The previous one was a simplistic explanation and a *very* naive solution of the problem
*(with, however, a very good return-on-investment)*

# The importance of data layout

## Matrix transpose: avoiding cache associativity conflicts



# Optimization of cache access in loops

# Cache access optimization in loops

## Loop classification

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

*Arithmetic Intensity*: the **ratio** between the **number of perfomend operations** and the **amount of the requested 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 optimization: *O(N) / O(N)* loops

Ex 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*, increasing **data reuse**

```
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 ]*

# Cache optimization: $O(N^2)$ / $O(N^2)$ loops

Ex 2-level loops on *N*: dense matrix-vector mul, matrix transpos., matrix add, …

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

Example of **unrolling + fusion**, starting from the following code:

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

N×N  $N^2$  N×N  → $3 \times N^2$

# Cache optimization: $O(N^2)$ / $O(N^2)$ loops

STEP 1:
Avoid unnecessary load/store operations

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

→ $2 \times N^2 + N$

# Cache optimization: $O(N^2)$ / $O(N^2)$ loops

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

```
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;
    }
```

→ $N^2 \times (1+1/m) + N$

N $N^2$ $N \times N / m$

```
/* NOTE: remainder loop ignored here */
```

# Cache optimization: $O(N^2)$ / $O(N^2)$ loops

**NOTE :: UNROLLING AND REGISTER SPILL**

If m in the previous example is too large it results in a code bloating if the target CPU does not have enough registers to keep all the needed operands.

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 optimization: $O(N^2)$ / $O(N^2)$ loops

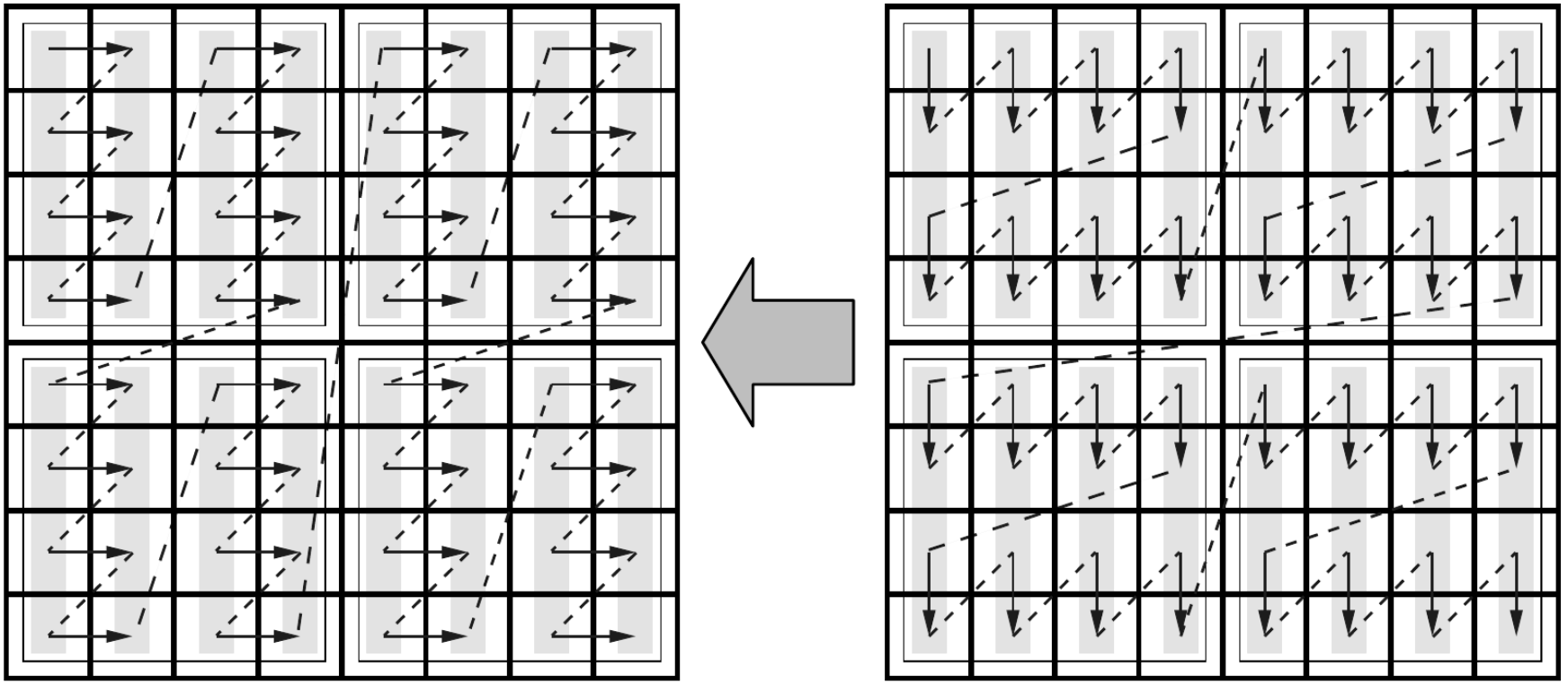
Sometimes no magic wand can cure the fact that you have to access $N^2$ memory locations.
For instance: in matrix transpose you have to access all the source matrix and all the destination matrix once.

*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 blocking*** is a good strategy that does not save memory loads but increase dramatically the cache hit ratio

# Cache optimization: $O(N^2)$ / $O(N^2)$ loops

STEP 3:
Exploit full locality of referenced data, cut TLB misses by accessing 2D arrays by blocks



*Taken from: introuction to HPC for scientists and engineers*

# Cache optimization: $O(N^3)$ / $O(N^2)$ loops

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

Blocking, 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.

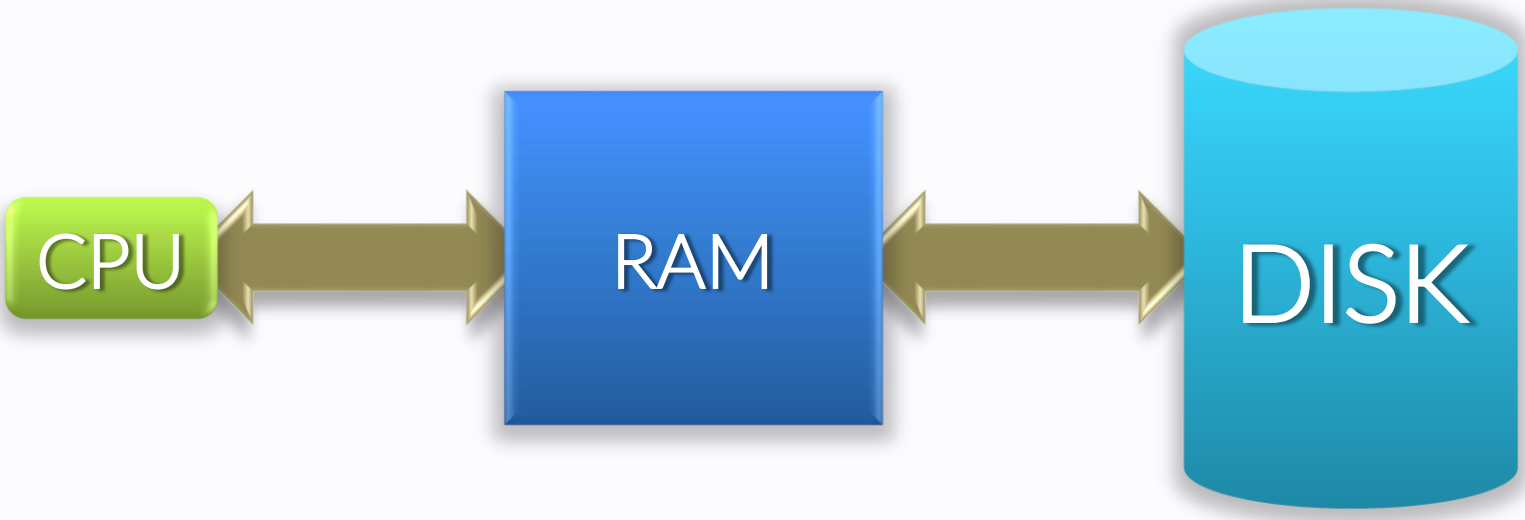
# Cache-consciousness

*cache-aware vs cache-oblivious*

# Cache – consciousness

In the good old days, computers were much simpler and much closer to a lovable *Von Neumann* architecture.

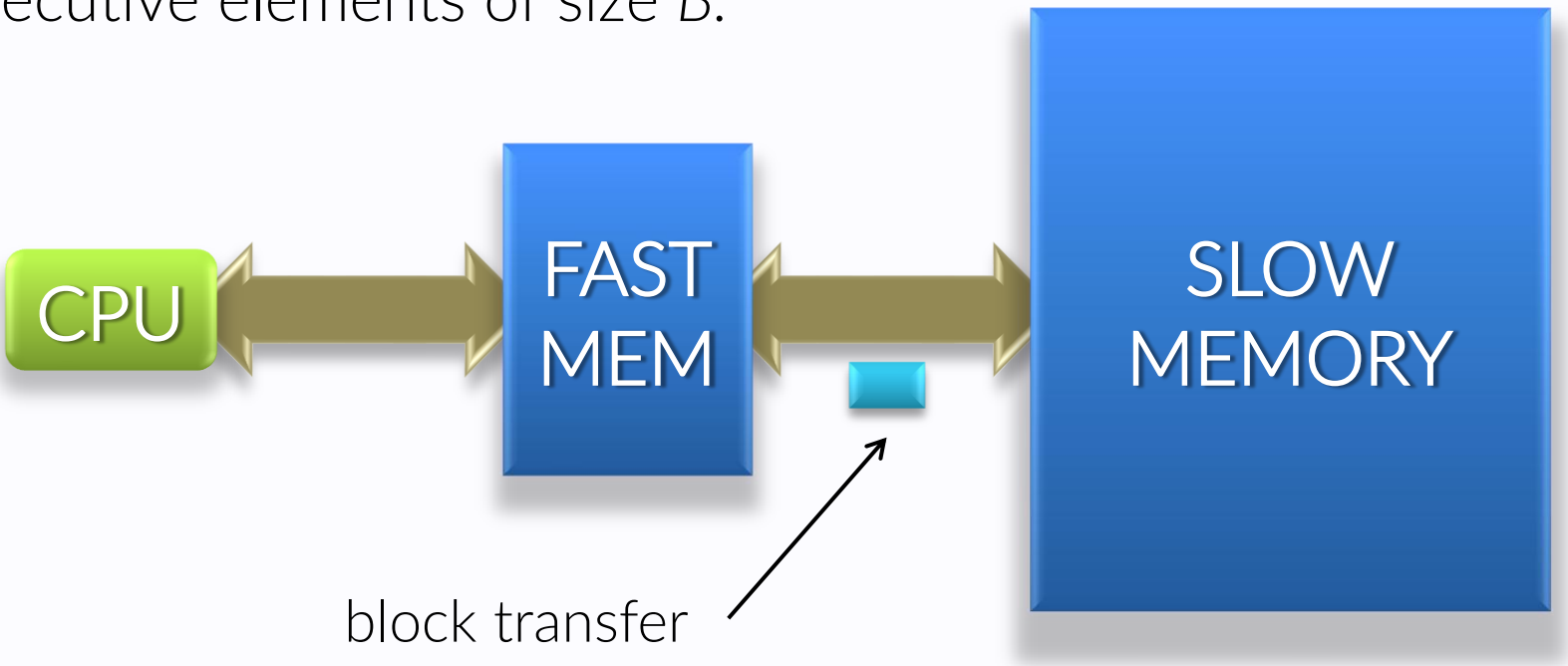
The **RAM MODEL** was more than enough to design and analyze algorithms, based on complexity measured as the *instructions* – the work *W(n)* – needed for an input problem of a given size *n*.



# Cache – consciousness

More realistic models became soon mandatory, and the most successful one was the simple two-level I/O MODEL, which was firstly shaped with memory-disk operations in mind.

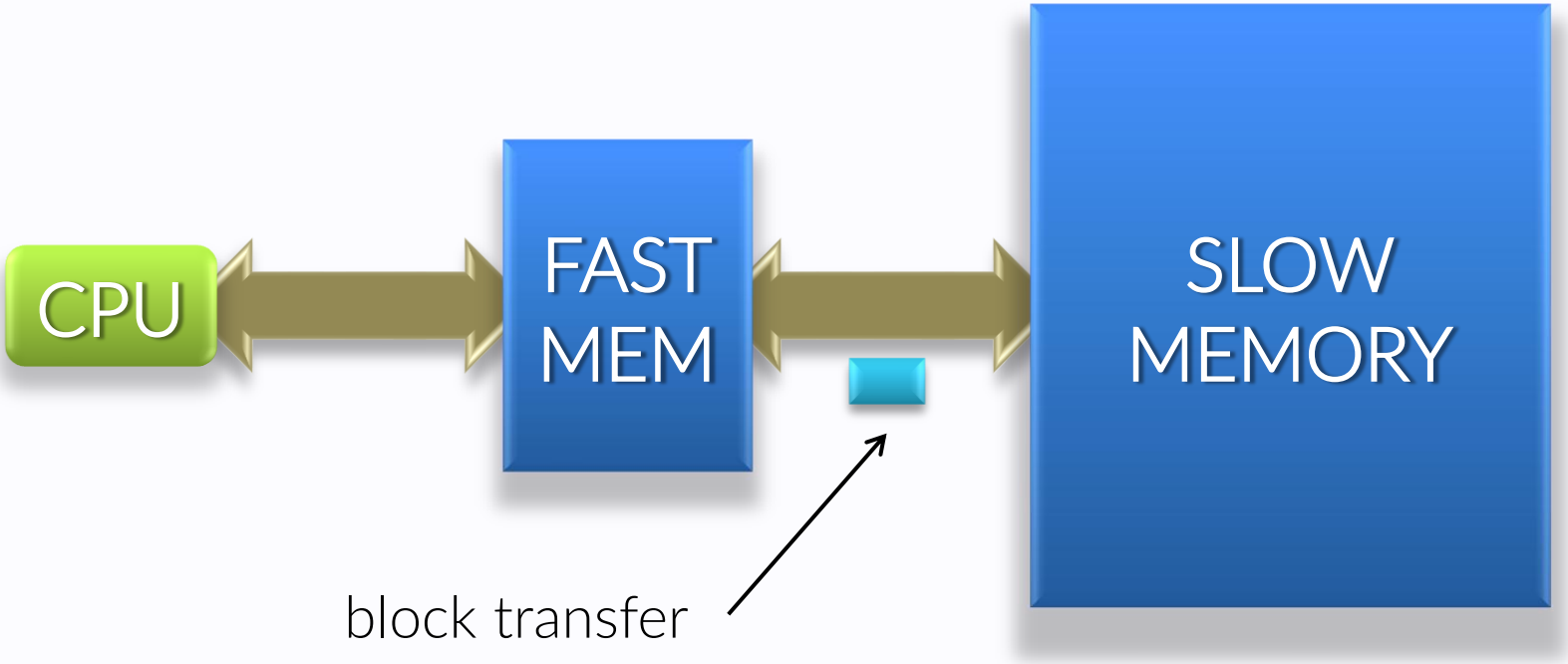
The hierarchy is assumed to consist in a fast memory of size *M* that exchange data with a slower infinite memory in bunches of consecutive elements of size *B*.



# Cache – consciousness

The basic idea is to evaluate the code efficiency by the memory transfers it need to operate. It captures particularly well the case of problems whose size *n* largely exceeds the RAM size. When tuned to specific *(M,B)*, it is called **cache-aware** model.

*Ex: comparison-based* $Sort_{M,B}(N) = \Theta\left(\frac{N}{B} log_{M/B} \frac{N}{B}\right)$ *memory transfers*



# Cache – oblivious model

In 1999, Frigo et al. proposed the ***cache-oblivious model*** which did **not required a precise knowledge of the memory hierarchy**.

Basically, a cache-oblivious algorithm is an algorithm formulated for the RAM model but analyzed with the I/O model, with the requirement that the analysis holds for any (M ,B) pair under the only assumption of the *tall memory:* $M = \Omega(B^2)$

The I/O-model analysis is valid for any block B and memory M, and then is valid for *all levels* of the cache hierarchy.
Hence, optimizing an algorithm, or a data structure for an unknown level of the hierarchy, it well behaves in the whole hierarchy.

# Cache – oblivious model

The cache-oblivious algorithms outperform the old-fashioned RAM algorithms and quite often are close or equivalent to algorithms tuned to a specific memory hierarchy.

However, they well behave at all level of the hierarchy and are much more portable (with good performance) and much more robust against change in problem sizes than algorithms designed for a very specific architecture.

# Cache – oblivious model

Several cache-oblivious algorithms and data structures have been found:

- Sorting
- Searching
- Static / dynamic B+-trees
- FFT
- Matrix transpose
- Matrix-Matrix multiply
- Computational geometry
- Priority queues
- Graph algorithms
- < … >

We are not going into detail here, just listing few basic concepts

# Some basic notes on data structures

# Field reordering

Reorder fields in structures so that what is used together stays together

```
struct my_node {
   double   key;
   char     my_data[300];
   my_node *next_node;
}
```

Linked-list node

```
void myfunc(my_node *p, double key, <…>) {
   while( p != NULL) {
         if( p->key == key ) {
               do_something( <…> );
               break;
         }
         p = p → next_node;
   }
```

Linked-list traversal

# Field reordering

Reorder fields in structures so that what is used together stays together

```
struct my_node
{
  double    key;
  char      my_data[300];
  my_node *next_node;
}
```

```
struct my_node
{
  double    key;
  my_node *next_node;
  char      my_data[300];
}
```

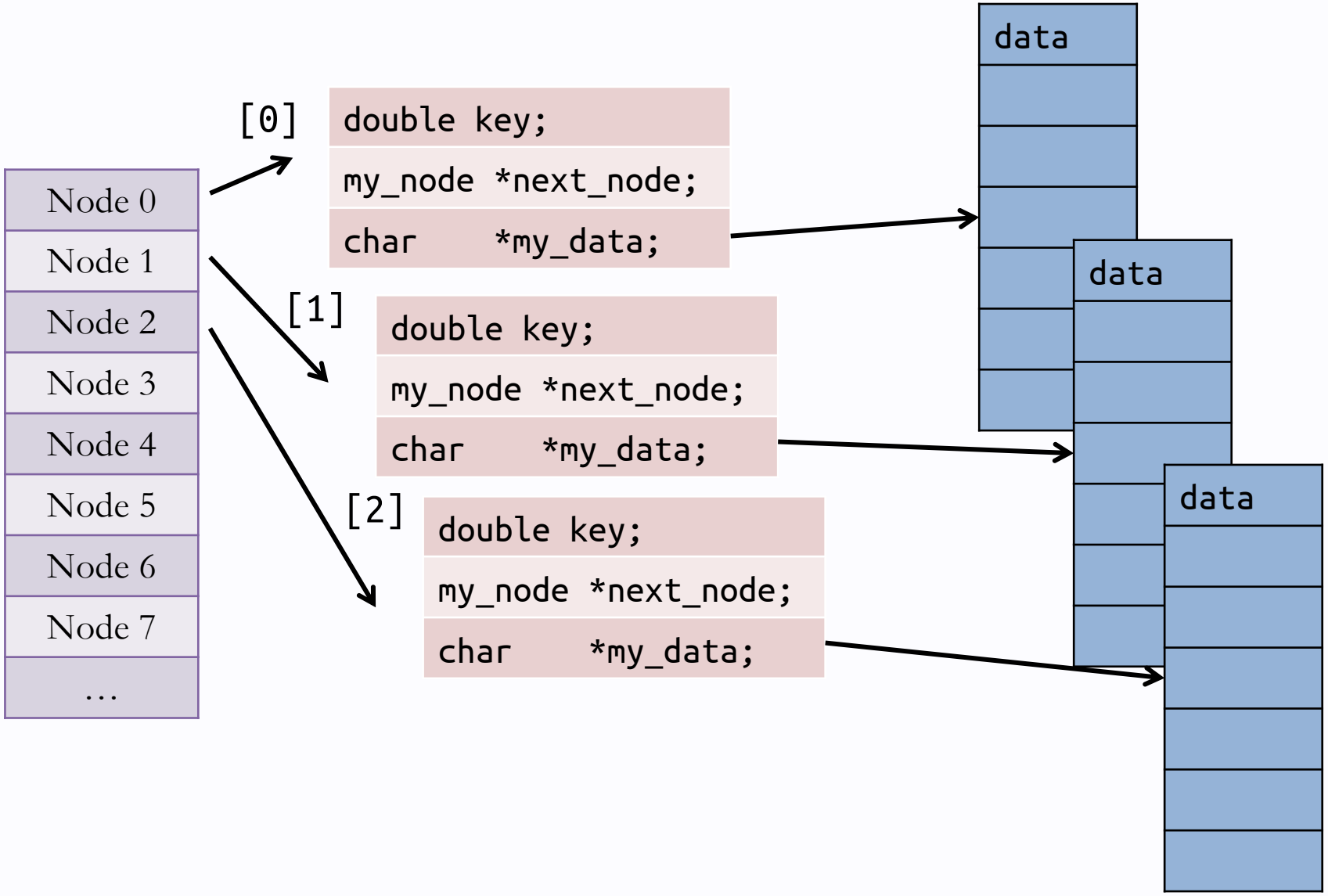
# Field split: hot and cold

Split fields so that to keep consecutive the fields that are used sequentially

```
struct my_node
{
  double   key;
  my_node *next_node;
  char     my_data[300];
}
```

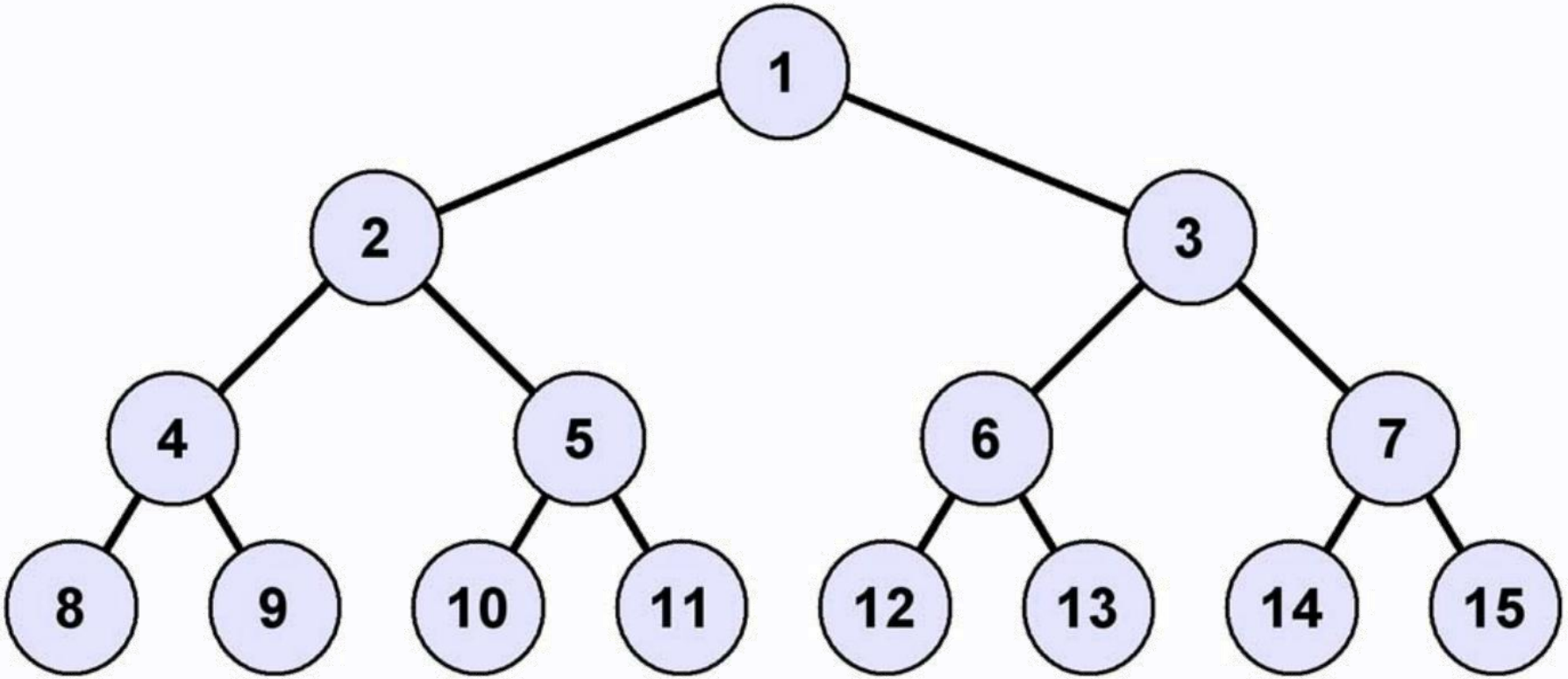
```
struct my_node
{
  double   key;
  my_node *next_node;
  void    *my_data;
}

struct my_data
{
  char data[300];
}
```

# Field split: hot and cold



# Data structure reordering

Example: TREEs – depth-first order

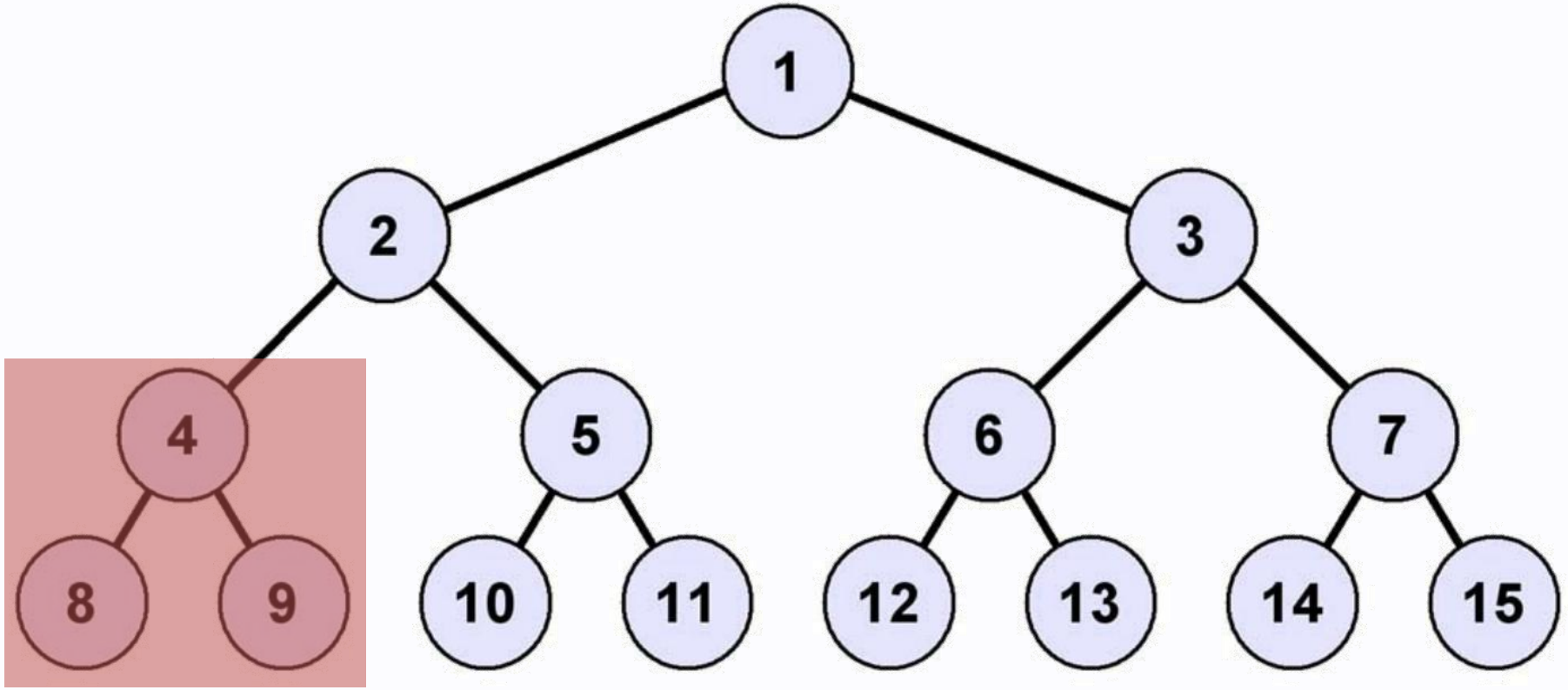



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

```
Left[n] = 2n  - right[n] = 2n+1
```

# Data structure reordering

Example: TREEs – depth-first order

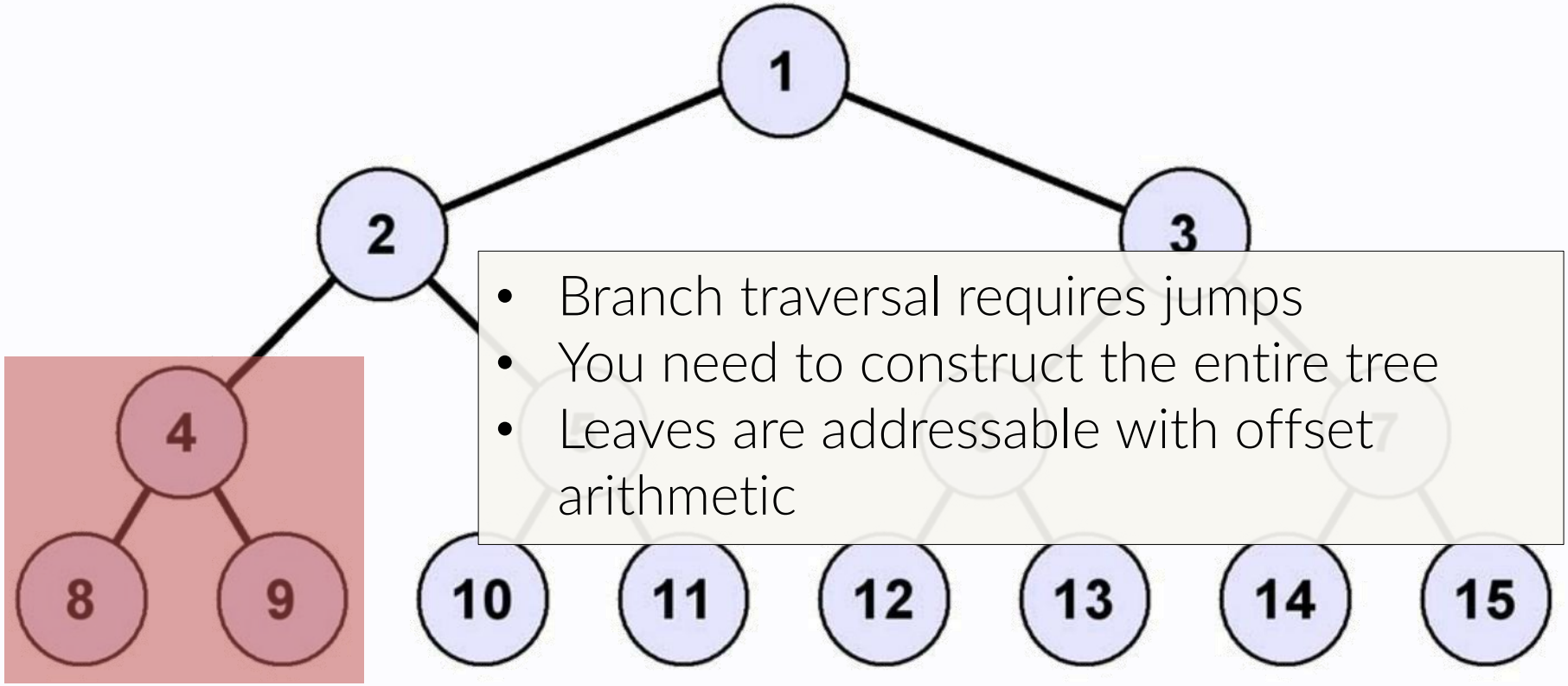



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

```
Left[n] = 2n  - right[n] = 2n+1
```

# Data structure reordering

Example: TREEs – depth-first order

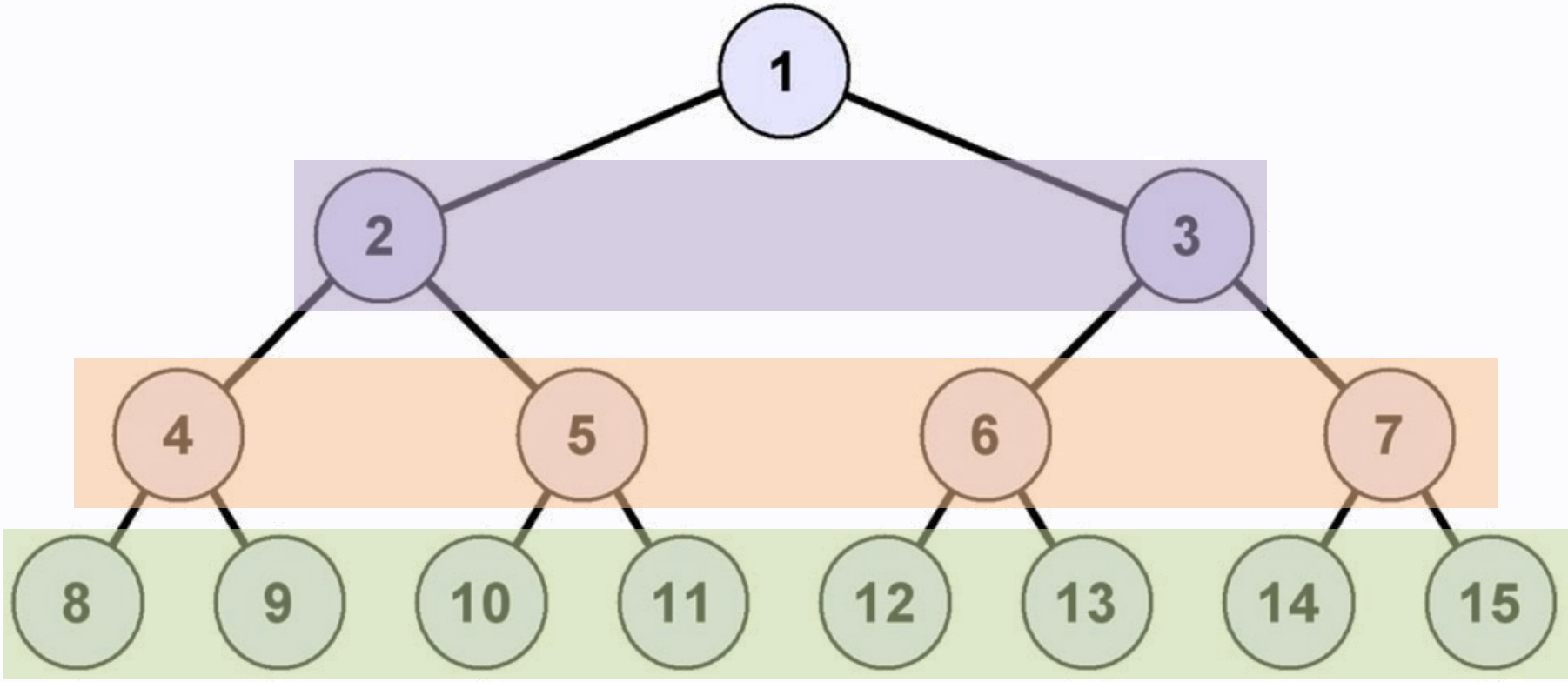



- Branch traversal requires jumps
- You need to construct the entire tree
- Leaves are addressable with offset arithmetic


```
Left[n] = 2n  - right[n] = 2n+1
```

# Data structure reordering

Example: TREEs – depth-first order

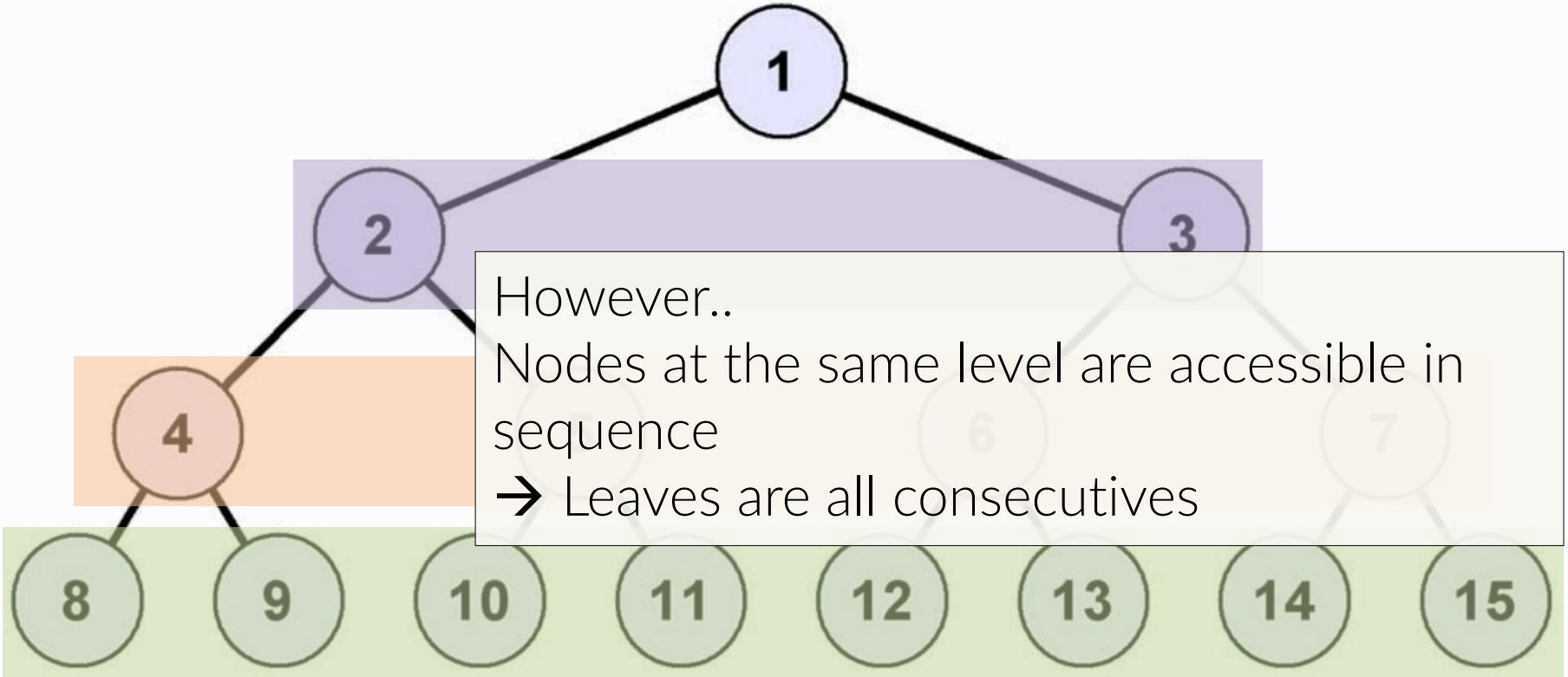




```
Left[n] = 2n  - right[n] = 2n+1
```

# Data structure reordering

Example: TREEs – depth-first order

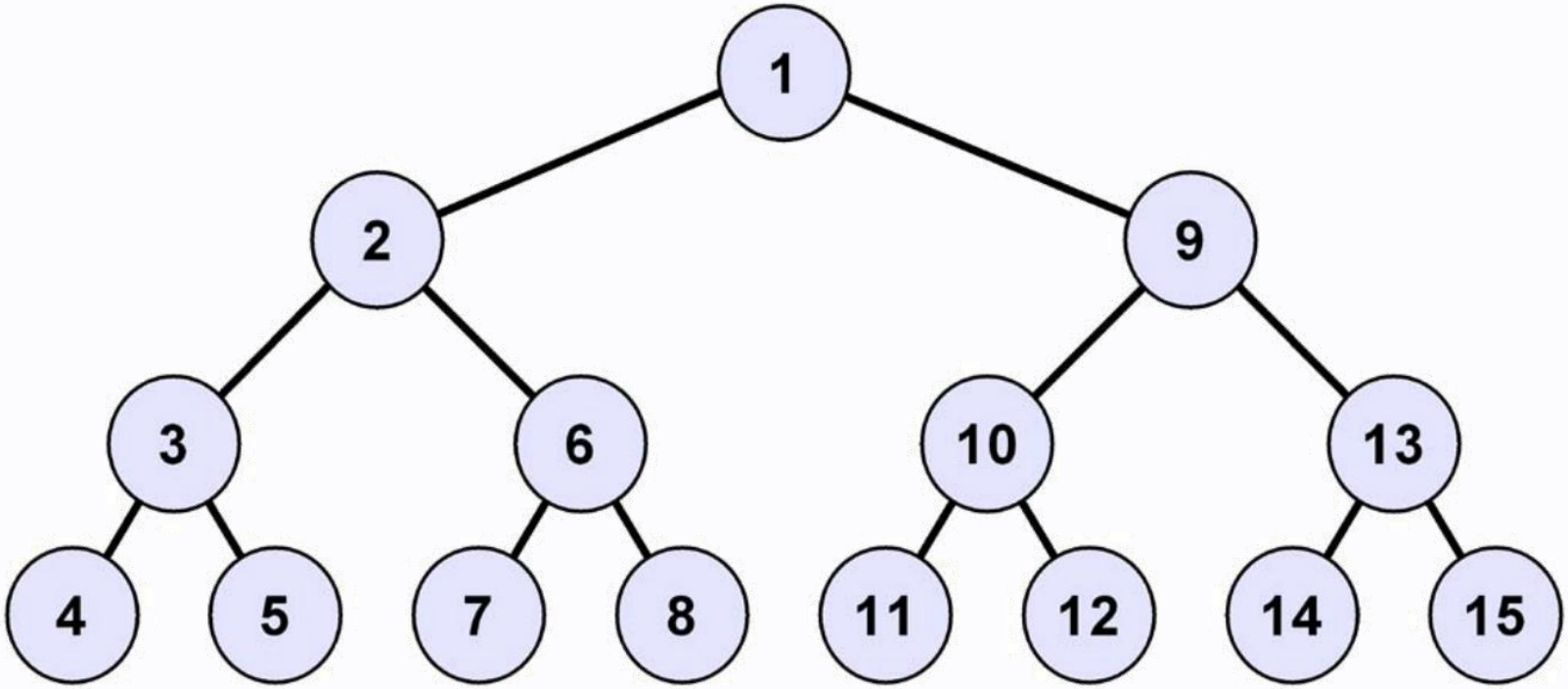


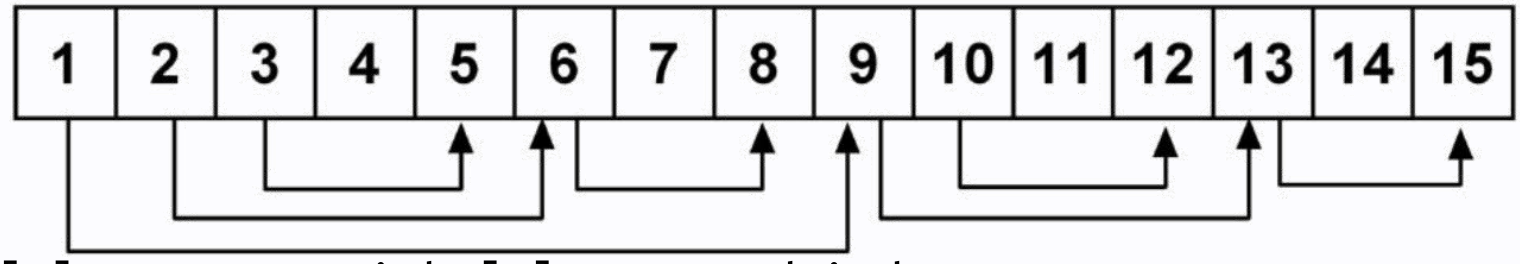


`Left[n] = 2n  - right[n] = 2n+1`

# Data structure reordering

Example: TREEs – breadth-first order

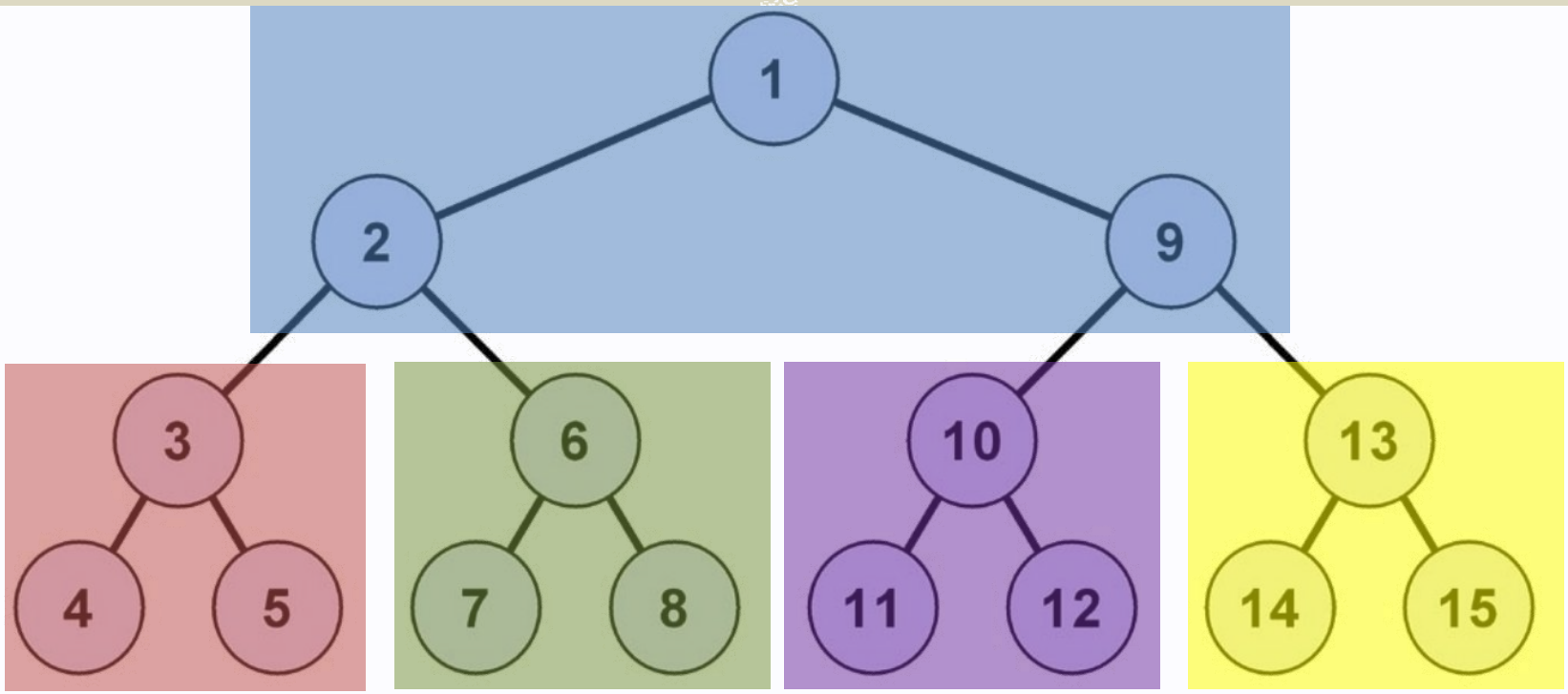


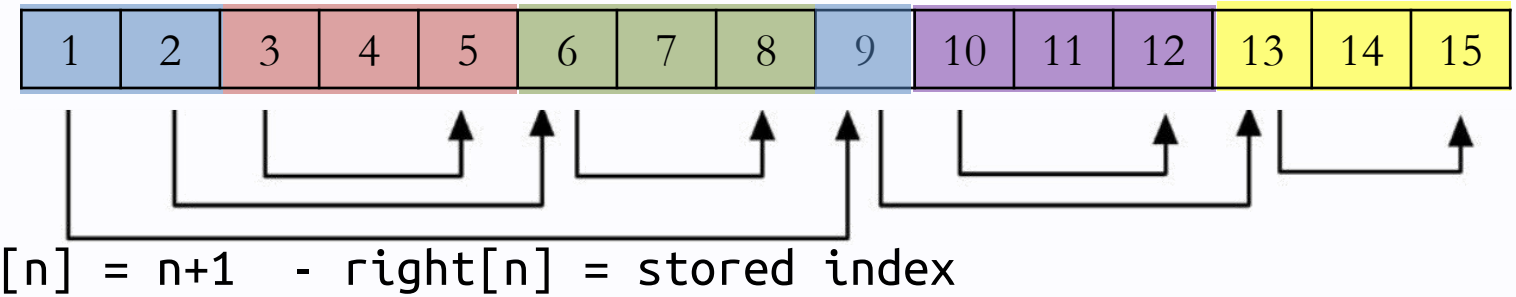




`Left[n] = n+1  - right[n] = stored index`

# Data structure reordering

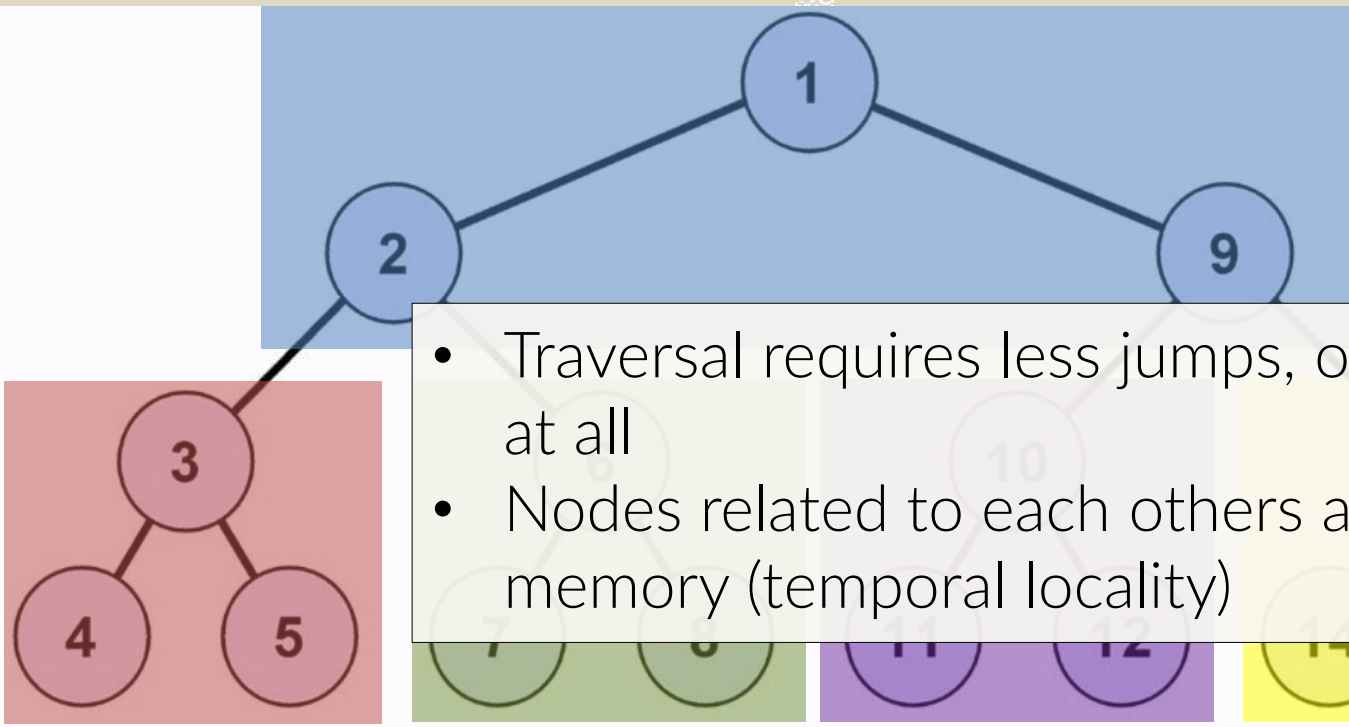
Example: TREEs – breadth-first order



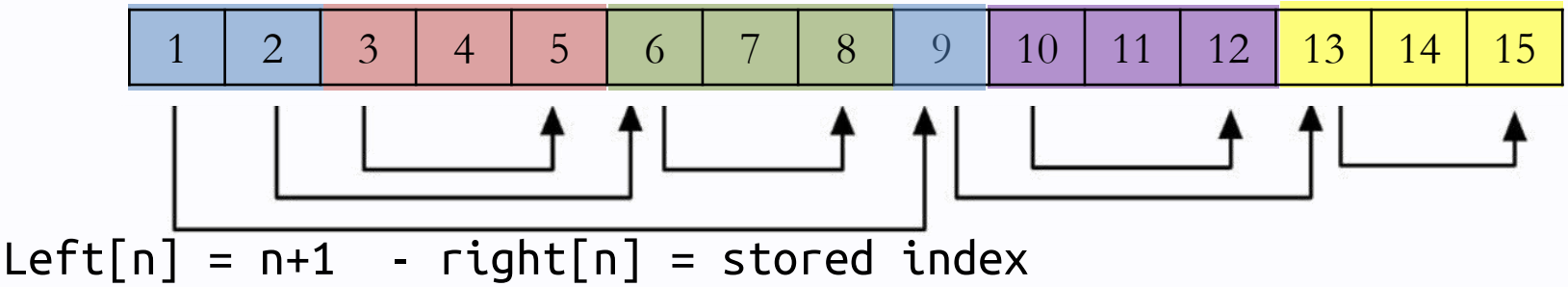



# Data structure reordering

Example: TREEs – breadth-first order




- Traversal requires less jumps, or no jumps at all
- Nodes related to each others are close in memory (temporal locality)



# Organizing data for locality

# Data reordering

As we have seen, there are several approaches to the cache-optimization problem:

- *Cache –aware* or *–oblivious* algorithm design
- Embedded euristics and policies in the CPU
- Prefetching
  - hardware
  - software (explicit or “induced”)
- SO level (page mapping)
- Fields reordering in data structures

and …

→ DATA REORDERING

# Data reordering

When the memory bandwidth is “limited” – which may be the case for highly parallel + multicore systems with a string NUMA hierarchy, **data locality optimization** can play a strong role.

Re-organizing data in “space” (whichever is their *n*-dimensional space) so that the access pattern is optimal for a given algorithm is related to such locality optimization.

Ex. 1: data pattern might be trivial, as in matrix transpose/mul

→ very specific ordering or pattern design

Ex. 2: data pattern may be spatially-coherent but unknown before it happens. For instance, in radiative transfer

→ optimization of data needed for a general case

# Data reordering

## Optimizing data locality for a general case

In the “space” the data live in – for instance our usual 3D space – there might be a metric that correlates with spatial coherence (in our 3D space there is the Euclidean metric, for instance).

Generally, it is more probable to access in a short time lapse points that are also spatially close.

Then, two popular measure to exploit this are

- Minimizing the distance distortion
- Preserve the locality

i.e. keeping close in 1D memory world points that are close in *n*-dimensions enhances the probability of using neighbouring memory locations while they are still in the cache.

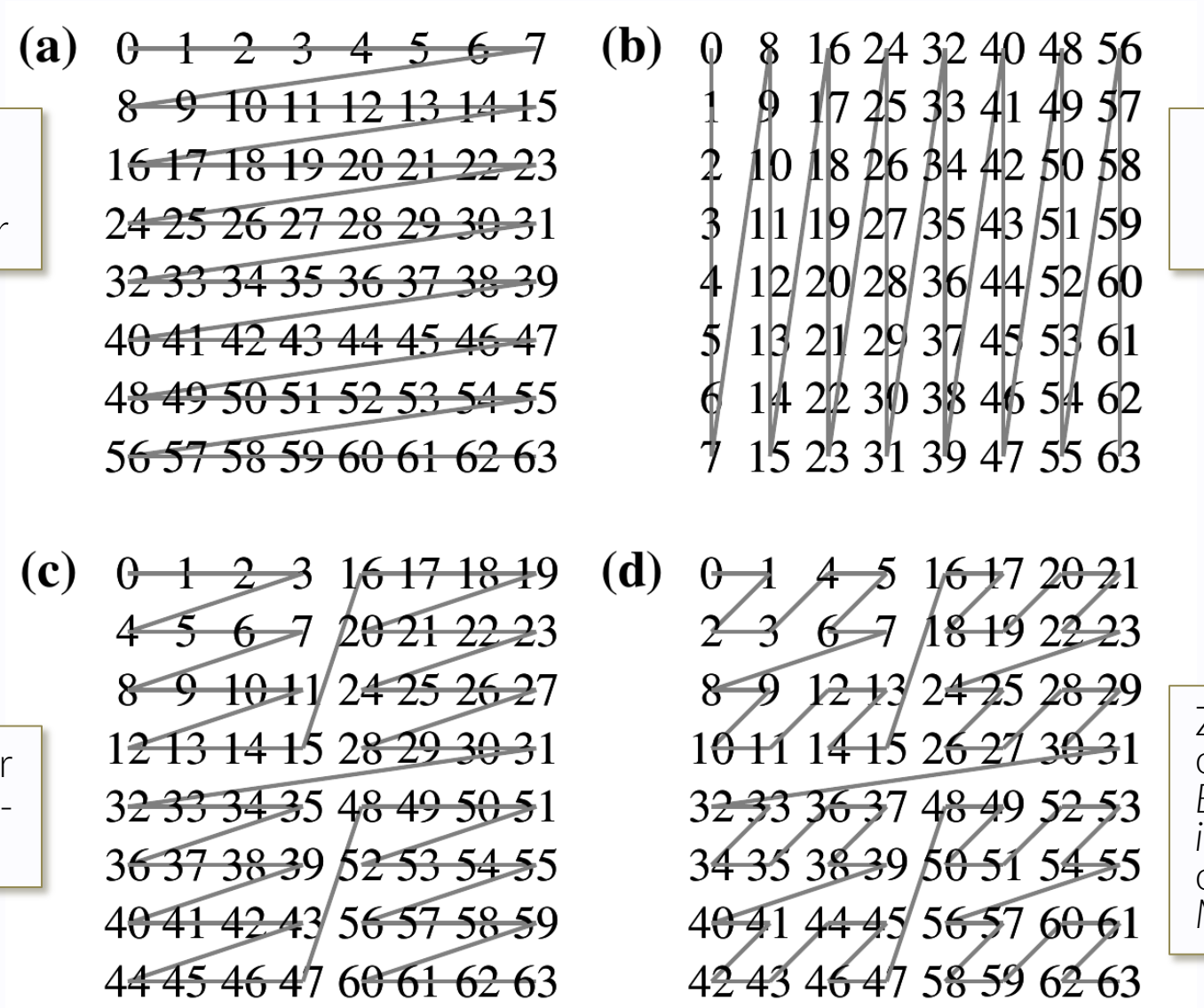
# Data reordering

What is the “minimum distance distortion” ?

In 1-D the answer is trivial.

In 2-D is less trivial, and it is increasingly less trivial while the number of dimensions rises.

# Data reordering




Scanline order *row-major*

Scanline order *col-major*

Block order + scan sub-order

Z- order or *Bit-interleaved* or *Morton order*

Fig. from *Frigo et al. 1999*

# Data reordering

Let's go in deeper detail about that "Z-order"

Let's say you want to map a linear access order

`0, 1, 2, 3, 4, 5, …, n`*th*

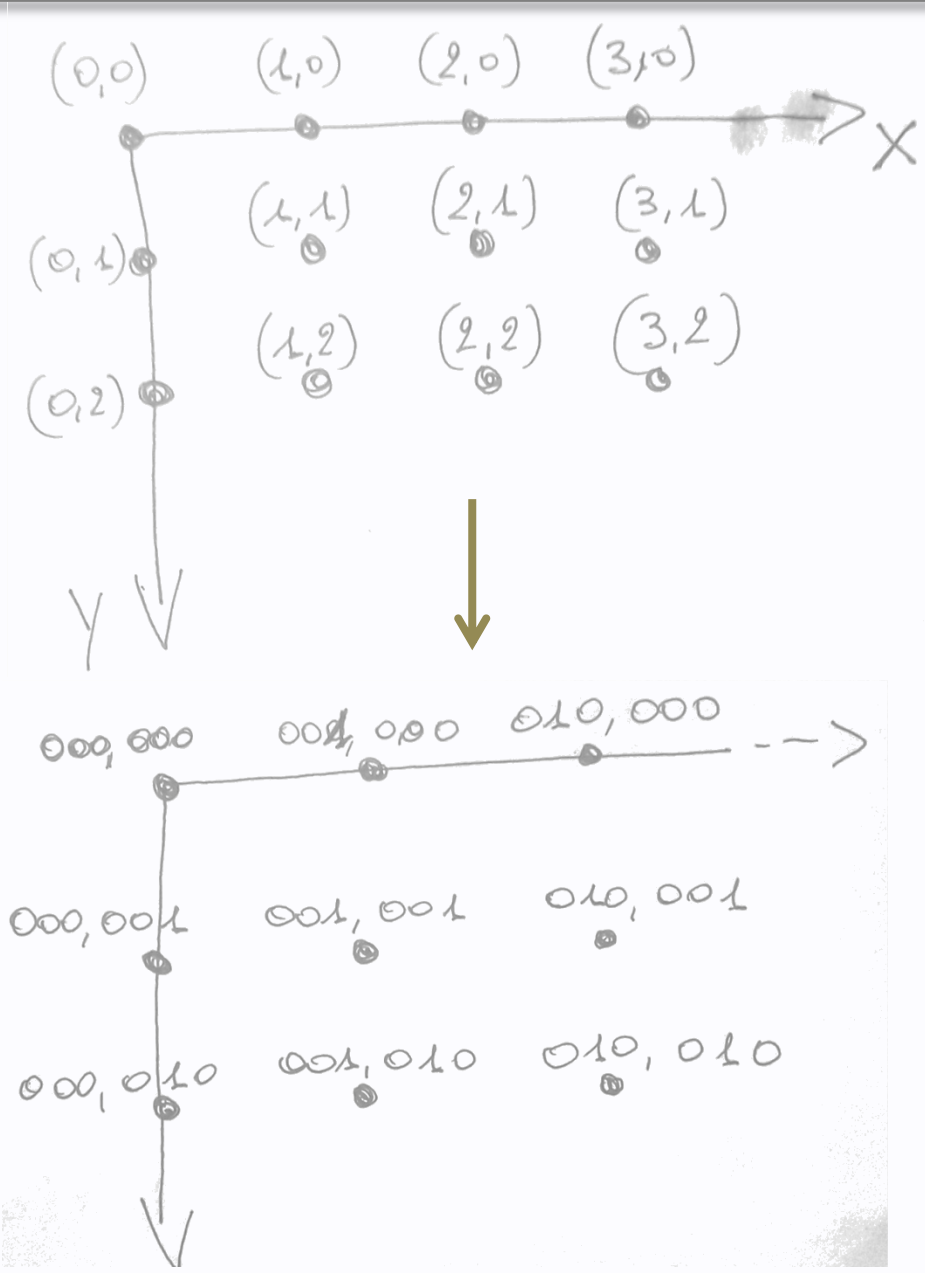
on some spatially-distributed data with integer coordinates.

Now, let's rewrite the linear access in base 2:

`0000, 0001, 0010, 0011, 0100, 0101, …`

What happens if the *bits* of the indexes of our traversal order are taken from the *bits* of the spatial coordinates of our points with a peculiar reshuffling?

# Data reordering




Let's define the binary representation of an integer number *x* :

$x = x_i \ldots x_2 x_1 x_0$

so that a couple *(x,y)* reads as:

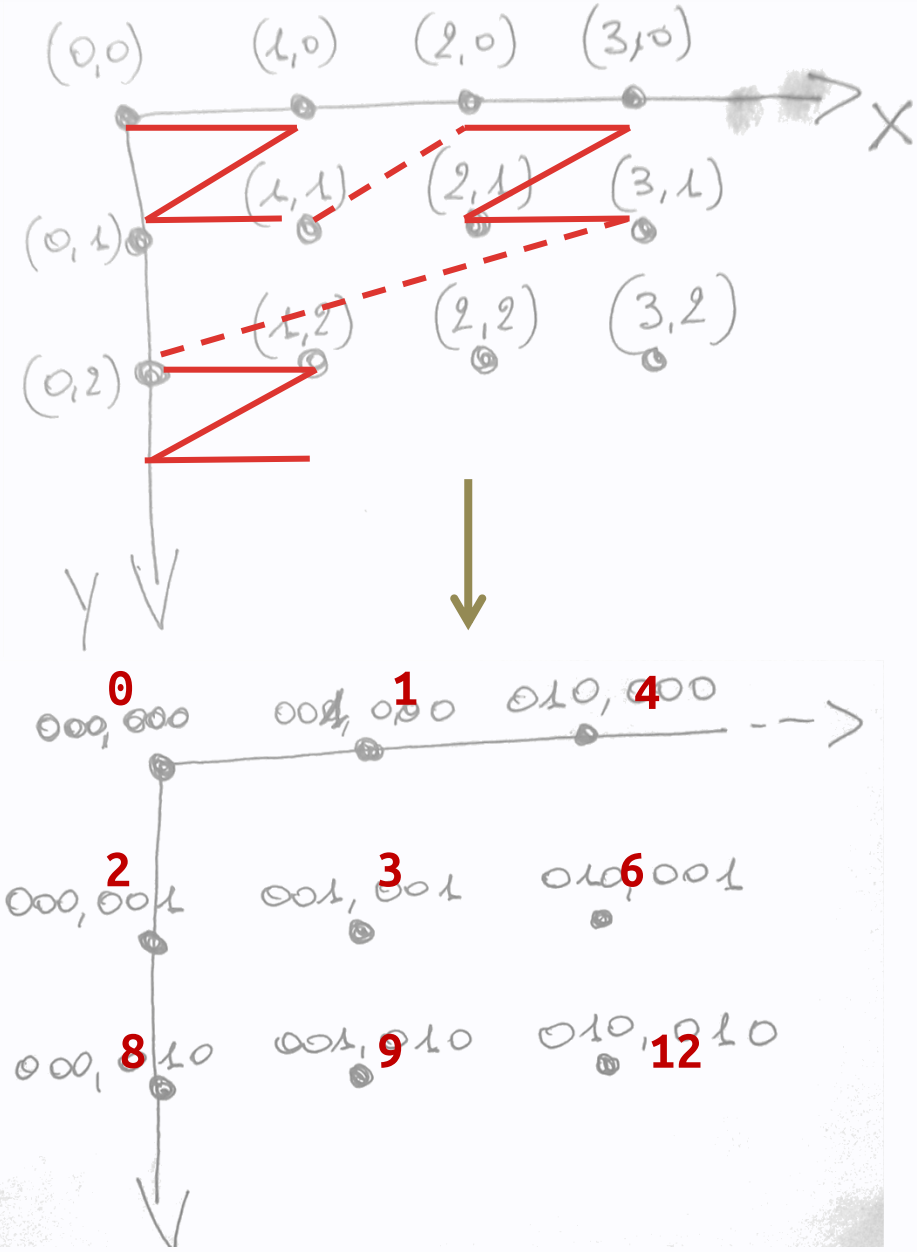
$( x_i \ldots x_2 x_1 x_0 , y_i \ldots y_2 y_1 y_0 )$

Then let's define the following reshuffle so to *interleave the bits*

$(x,y) \rightarrow (y_i x_i \ldots y_2 x_2 y_1 x_1 y_0 x_0)$

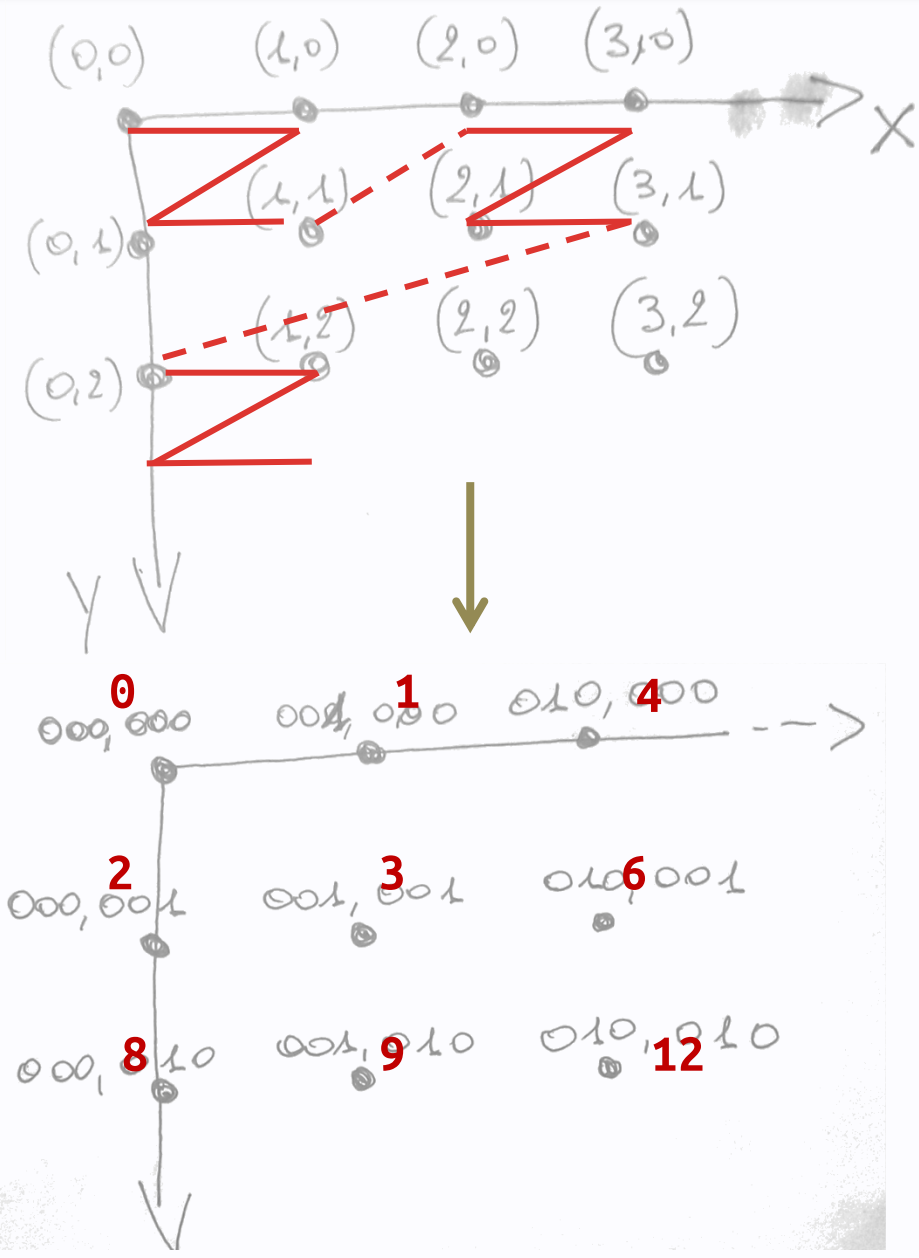
(0,0) → 00 00 = 0
(1,0) → 00 01 = 1
(2,0) → 01 00 = 4
(0,1) → 00 10 = 2
(1,1) → 00 11 = 3
…

# Data reordering




The result of this interleaving is a mapping from the 2-D plane to the 1-D line known as Z-line which is one of the **plane-filling curves** discovered by Peano

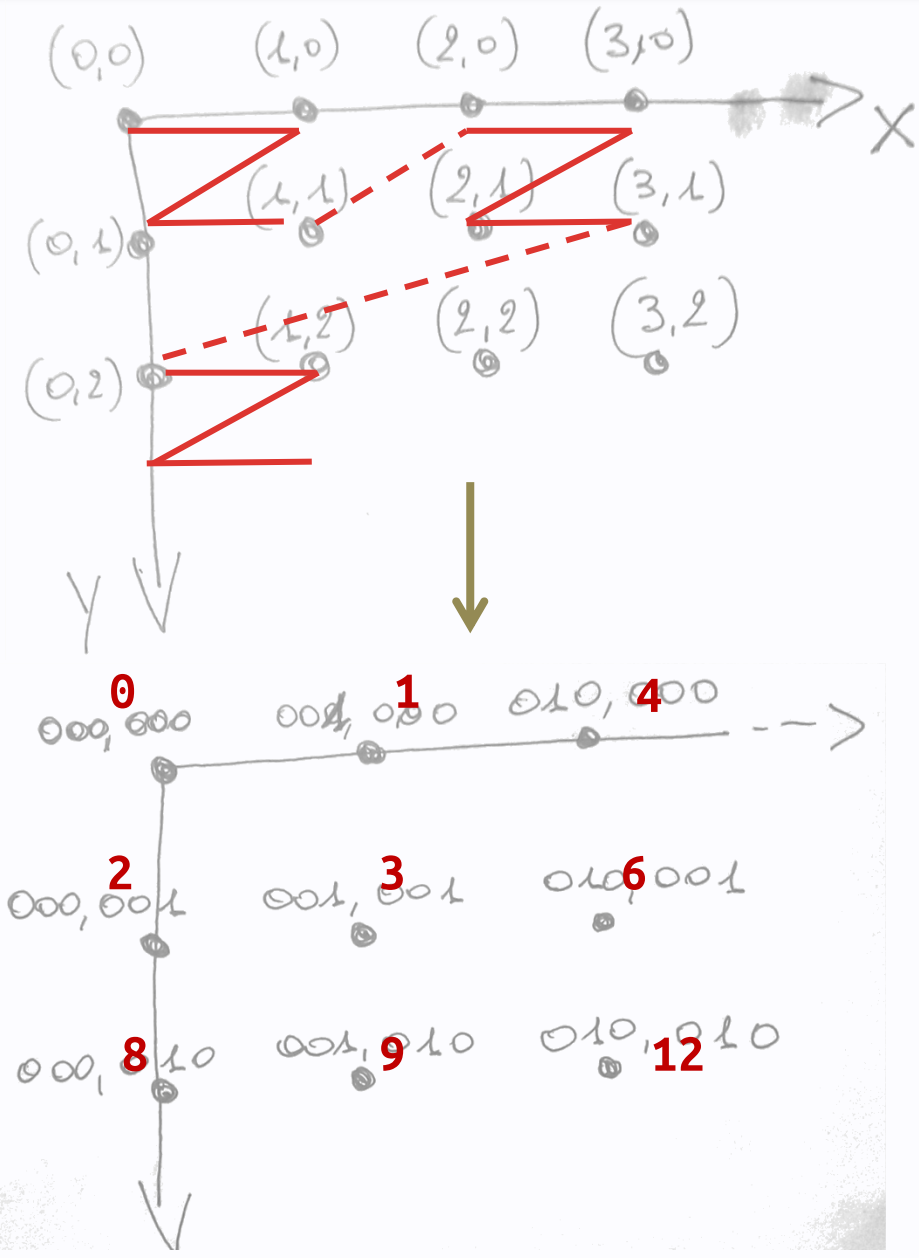
# Data reordering



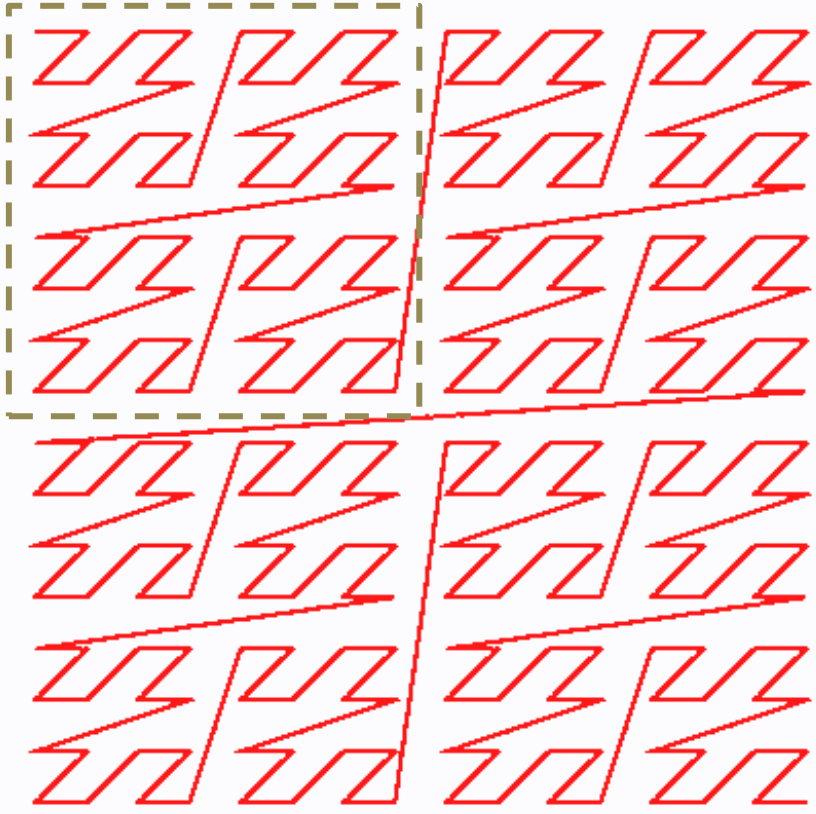

The result of this interleaving is a mapping from the 2-D plane to the 1-D line known as Z-line which is one of the **plane-filling curves** discovered by Peano

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 4 | 5 | 16 | 17 | 20 | 21 |
| 2 | 3 | 6 | 7 | 18 | 19 | 22 | 23 |
| 8 | 9 | 12 | 13 | 24 | 25 | 28 | 29 |
| 10 | 11 | 14 | 15 | 26 | 27 | 30 | 31 |
| 32 | 33 | 36 | 37 | 48 | 49 | 52 | 53 |
| 34 | 35 | 38 | 39 | 50 | 51 | 54 | 55 |
| 40 | 41 | 44 | 45 | 56 | 57 | 60 | 61 |
| 42 | 43 | 46 | 47 | 58 | 59 | 62 | 63 |

# Data reordering




The result of this interleaving is a mapping from the 2-D plane to the 1-D line known as Z-line which is one of the **plane-filling curves** discovered by Peano

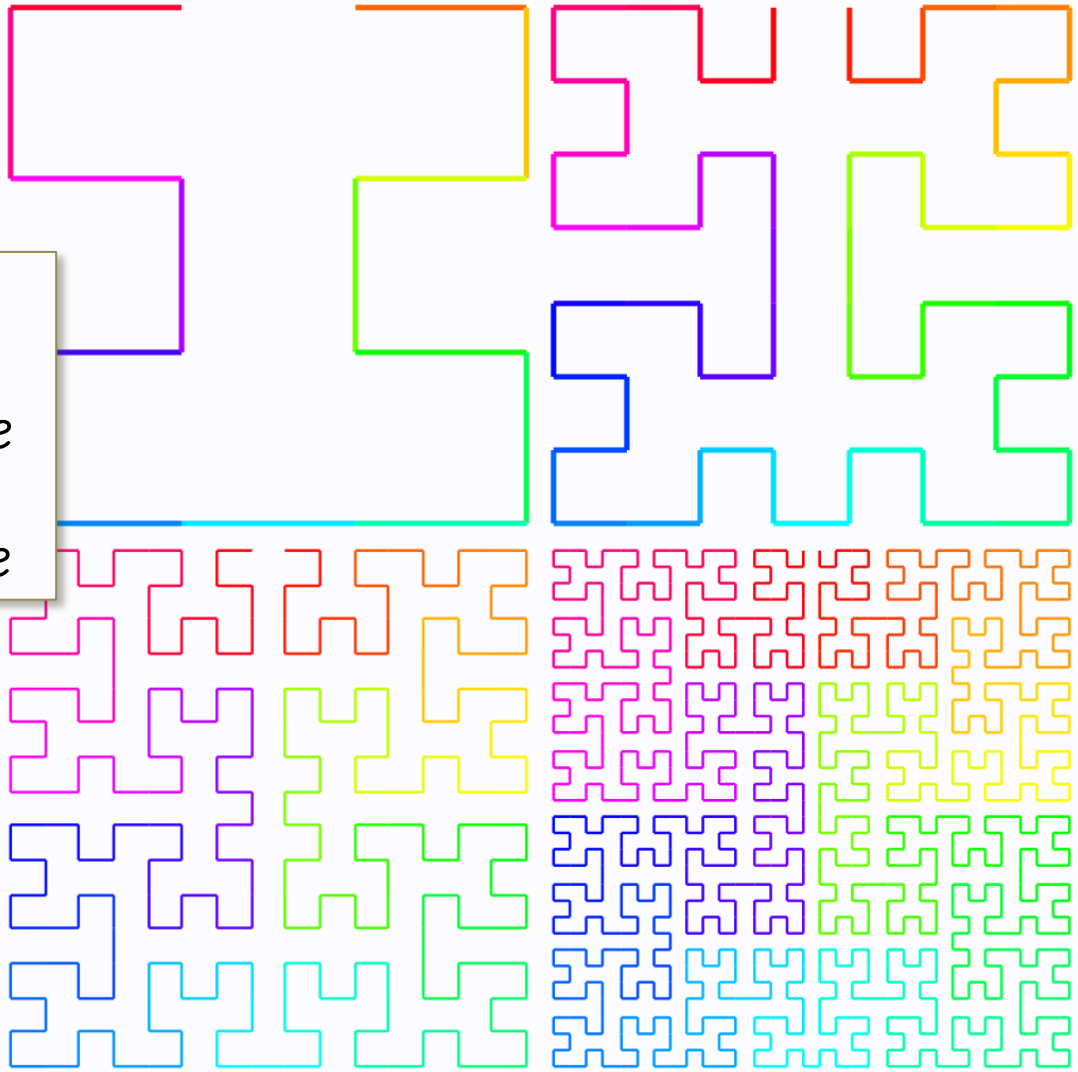


# The wonderful land of *Peano*-curves

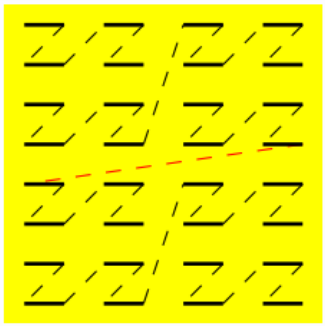
There are many of these curves

*Hilbert* curve

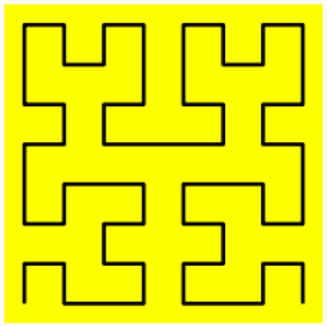
*Note that there are no jumps along the curve*



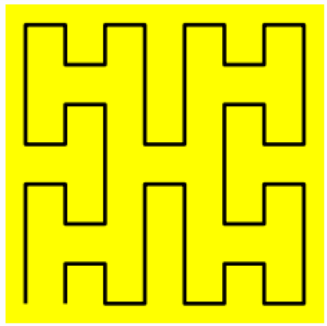
# The wonderful land of *Peano*-curves



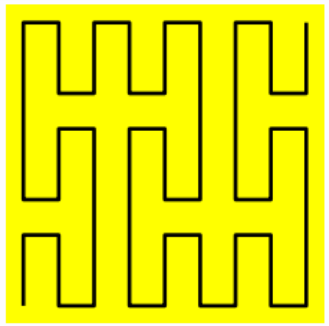
Z-order



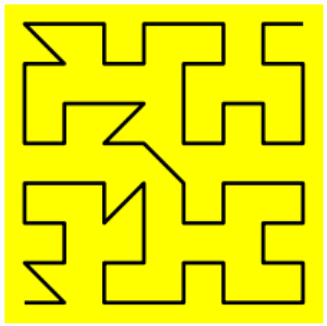
Hilbert curve



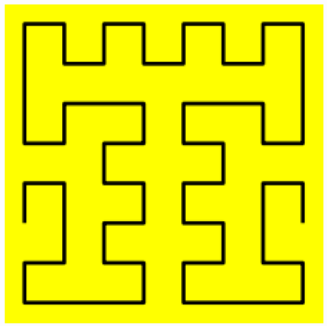
H-order



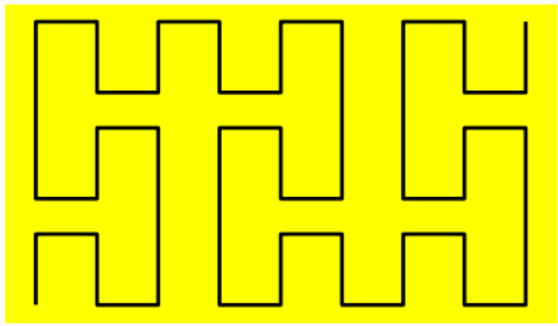
Peano's curve



$AR^2W^2$-curve



$\beta\Omega$-curve

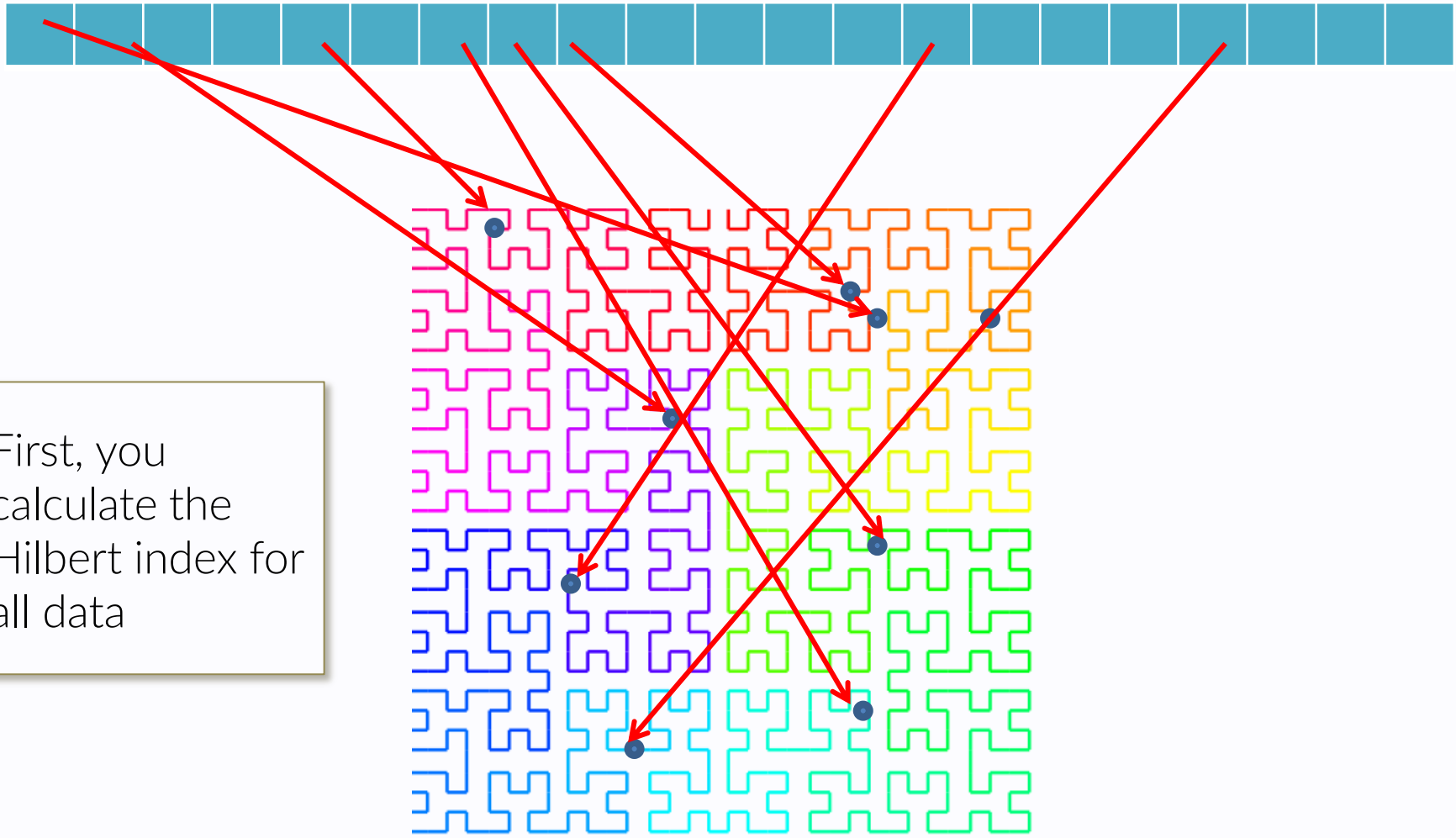


Balanced Peano

Each curve has some peculiar properties which reflects in the distance distortion they provide, i.e. on their performance in keeping "locality" in different situations

# The wonderful land of *Peano*-curves

## Re-ordering data in memory

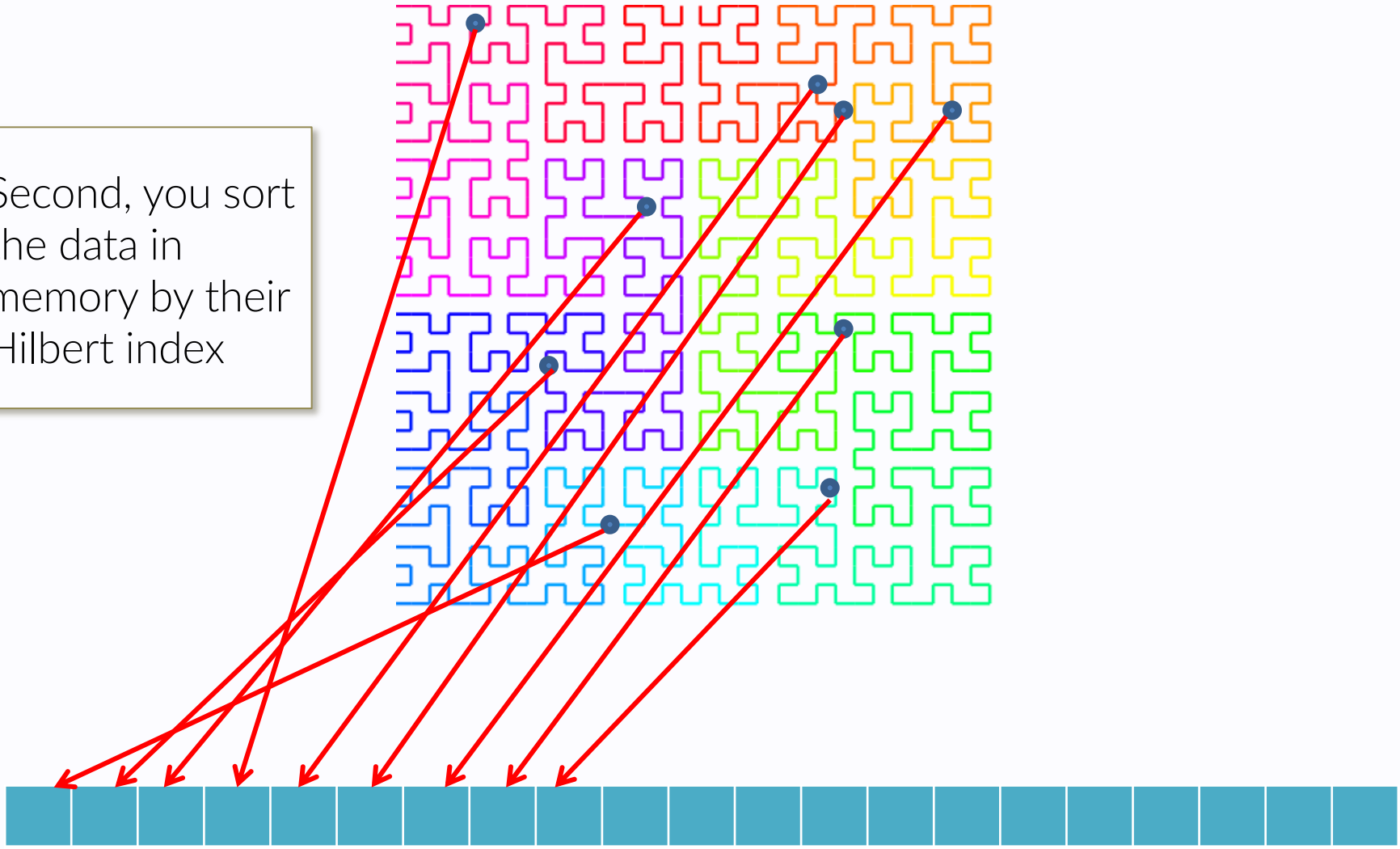


First, you calculate the Hilbert index for all data

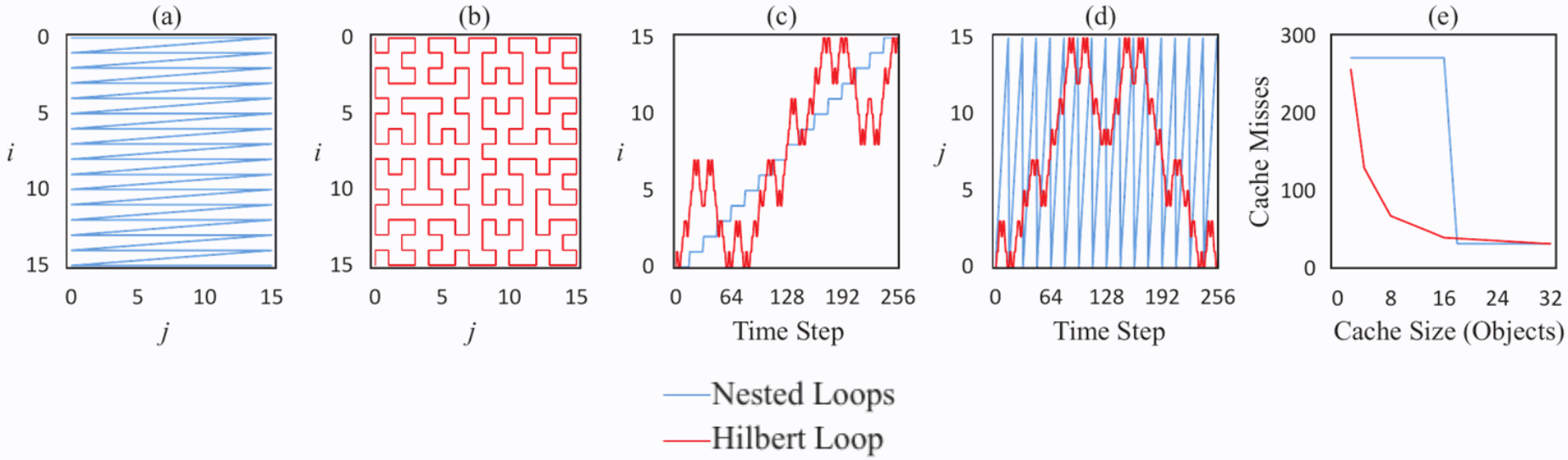
# The wonderful land of *Peano*-curves

## Re-ordering data in memory

Second, you sort the data in memory by their Hilbert index



# The wonderful land of *Peano*-curves

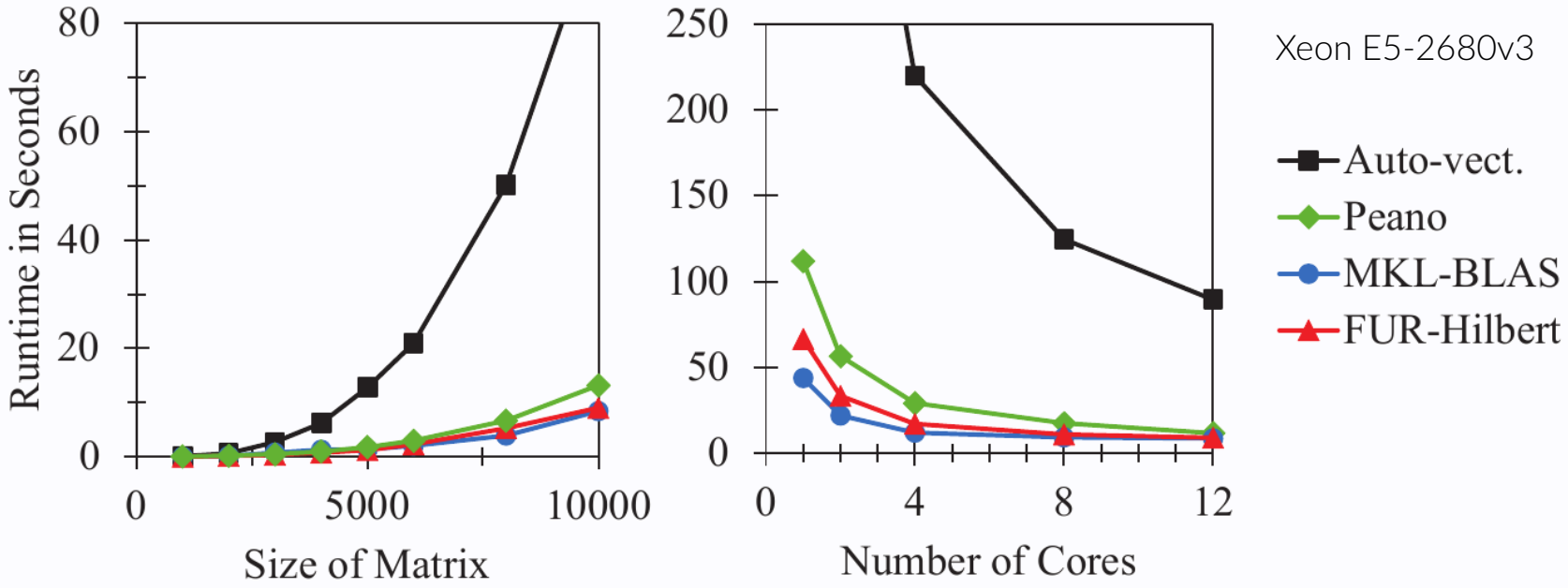



## Cache-oblivious at work : traversal order

Comparison of the traversal order for classic nested-loops and Hilbert curve.
Note the increased locality in *j* and the highly reduced numbers of L1 miss

Fig. from “Cache-oblivious Loops Based on a Novel Space-filling Curve”, Böhm et al. 2016

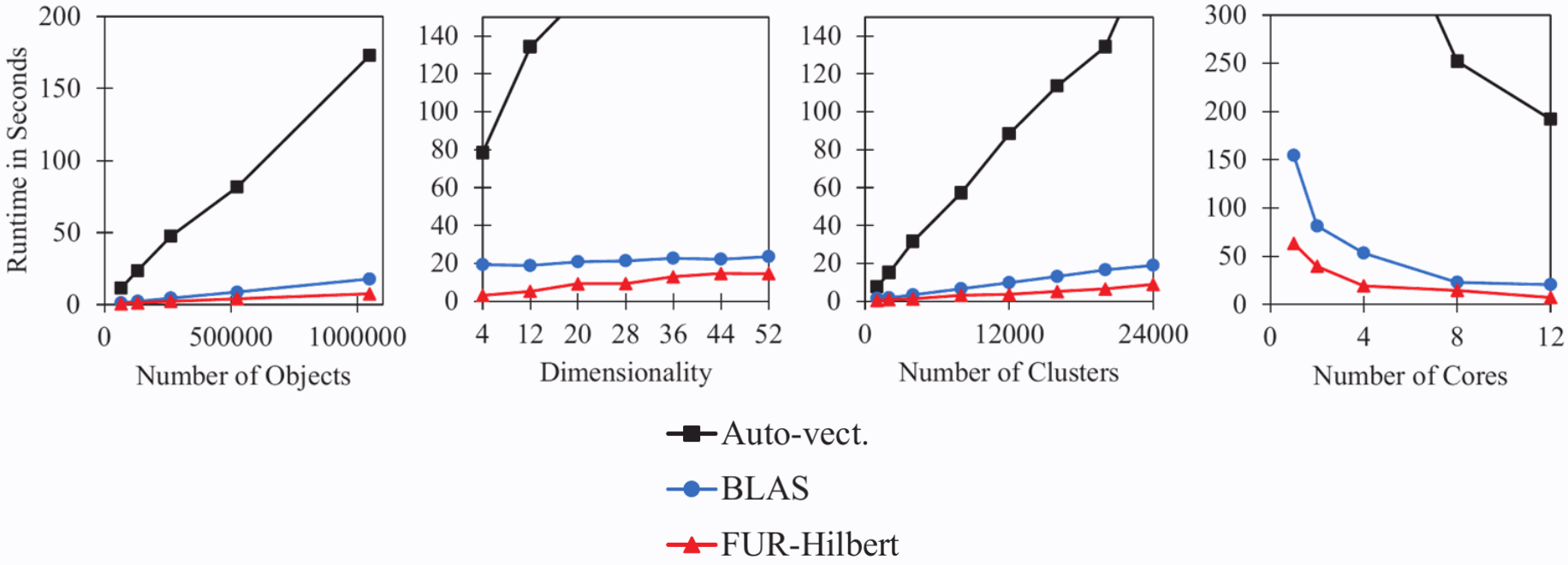
# The wonderful land of *Peano*-curves




Cache-oblivious at work : matrix multiplication

Fig. from “Cache-oblivious Loops Based on a Novel Space-filling Curve”, Böhm et al. 2016

# The wonderful land of *Peano*-curves




## Cache-oblivious at work : *k*-means

Fig. from “Cache-oblivious Loops Based on a Novel Space-filling Curve”, Böhm et al. 2016

# Cache recap

*In two slides*

# RECAP – Foes: 3 C's of cache misses

- **Compulsory misses**
  Unavoidable misses when data are read for the first time

- **Capacity misses**
  - Not enough space to hold all data
  - Too much data accessed in between successive use

- **Conflict misses**
  Cache trashing due to data mapping to same cache lines

# RECAP – Friends: 3 R's of cache hits

- **Rearrange ( code & data )**
  Design layout to improve temporal & spatial locality

- **Reduce ( size )**
  - Smaller data size – smaller chunks accessed
  - Fewer instructions

- **Reuse ( cache lines )**
  Increase spatial & temporal locality – keep resident data for more operations

# Outline

- Memory optimizations
  - > the importance of memory access
  - > the importance of the cache
  - > cache access optimization in loops
- **Loops** optimizations
- Other optimizations

Recap: a friend we've already seen

*Avoid the avoidable..*

# Loops optimization

Is there something wrong in this simple nested 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 < Rmax)
              dummy += dist;
          }
```

...yes, I know that you know

# Loops optimization

OPT 1: avoid expensive functions (like **`sqrt( )`**)

```
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 < Rmax2)
              dummy += sqrt(dist);
          }
```

# Loops optimization

OPT 1: avoid expensive functions (like **pow( )** )

```
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 < Rmax2)
              dummy += sqrt(dist);
          }
```

# Loops optimization

OPT 1: avoid expensive operations (like **floating point division** )

```
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 < Rmax2)
              dummy += sqrt(dist);
          }
```

# Loops optimization

OPT 2: avoid the avoidable + manual hoisting

```
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)
              dummy += sqrt(dist);     } } }
```

# Loops optimization

OPT 3: be clear with the compiler, use *local variables*

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

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

         for(int k = 0; k < Ng; k++) {
            double dz = z[p] - (double)k * …;
            double dist2 = dist2_xy + dz*dz;
            if(dist2 < Rmax2)
              dummy += sqrt(dist);     } } }
```

# Loops optimization

OPT 4: be clear with the compiler , also through **`register`** directive

```
double register Ng_inv = 1.0 / Ng;
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;
         register dist2_xy = dx2 + dy2;
         for(k = 0; k < Ng; k++) {
            register dz = z[p] - (double)k * …;

            register dist2 = dist2_xy + dz*dz;
            if(dist2 < Rmax2)
              dummy += sqrt(dist);     } } }
```

# Loops optimization

Compilers (at −O2,3) are very good in *loop hoisting*, that is to extract invariant operations/quantities from within the loops.

However, their speculation is limited by the obscurity of the code (for instance: aliasing of memory locations).

Ex. from previous code snippet:

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

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

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

# Loops optimization

Also, limit the use of **global variables** or variables with a **wide scope**, and use local variable with limited scope every time it is possible (and it often is).

That helps the compiler in optimizing the registers and cache use.

Trivial ex:

```
int    i;
double temp;
<…>
for(i = 0; i < Ng; i++){
   temp = <…>
}
```

➔

```
<…>
for(int i = 0; i < Ng;i++){
   double temp = <…>
}
```

# Loops and branches

# Loops optimizations: branches

Conditional branches should be avoided as much as possible inside loops:

- moving them outside the loop and writing specialized loops
- performing variables/quantities set-up pree-emptively outside the loop
- using pointers to functions instead of selecting functions inside the loop
- substituting conditional branches with different operations

# Loops optimizations: branches

ex 1: Taking decisions before and outside the loop

```
for(i = 1; i < top; i++)
   {
     if(case1 == 0) {
       if(case2 == 0) {
          if(case3 == 0)
            result += i;
          else
            result -= i;
          }
          else {
            if(case3 == 0)
              result *= i;
            else
              result /= i;
          }
      }
```

```
else {
  if(case2 == 0) {
     if(case3 == 0)
       result += log10((double)i);
     else
       result -= log10((double)i);
     }
  else {
    if(case3 == 0)
      result *= sin((double)i);
    else
      result /= (sin((double)i) +
                 cos((double)i));
     }
  }
```

# Loops optimizations: branches

ex. 1 : Taking decisions before and outside the loop

- define a specialized function for each case
- **before** and **outside** the loop set a function pointer to the right function

```
void (*func)(double *, int);
<here make func pointing to the right place>

double temp   = 0;
double result = 0;
for(i = 1; i < top; i++)
   {
       func( & temp, i);
       result = temp;
   }
```

➔ …just have a look at the live results..

# Loops optimizations: branches

However:

- Using function pointers you may incur in additional overhead due to function call.
  If the code snippets in different if-branches (or at least the most executed ones) are large/expensive, it might well be pointless (in modern CPUs).

- "Unrolling" the if-tree outside the for – then having multiple for loops :

```
if (case1 == 0) {
  if (case2 == 0) {
     if (case3 == 0) {
        for(i = 1; i < top; i++)
           result += i;
     else
        for(i = 1; i < top; i++)
           result -= i;
     }
```

may be highly unpractical if the branches are big piece of code.
There's no really a Swiss-knife recipe.

➔ …just have a look at the live results..

# Loops optimizations: branches

## ex. 2 : restructuring code

Consider the following code snippet

```
// generate random numbers
for (cc = 0; cc < SIZE; cc++)
  data[cc] = rand() % TOP;

// take action depending on their value
for (ii = 0; ii < SIZE; ii++)
  {
    if (data[ii] > PIVOT)
       sum += data[ii];
  }
```

# Loops optimizations: branches

## ex. 2 : restructuring code

Consider the following code snippet (*)

```
// generate random numbers
for (cc = 0; cc < SIZE; cc++)
  data[cc] = rand() % TOP;

qsort(data, SIZE, sizeof(int), compare);

// take action depending on their value
for (ii = 0; ii < SIZE; ii++)
  {
    if (data[ii] > PIVOT)
       sum += data[ii];
  }
```

(*) of course, you are adding an overhead due to the sorting routine, so the total running time may be even larger. Moreover, you should have all the values available so that does not work for real-time streamings. However, the point here is to focus on how – in general – it is better to avoid conditionals inside loop, with any possible trick or change in workflow

➔ ...just have a look at the live results..

# Loops optimizations: branches

ex. 2 : restructuring code

We can do even better, without adding operations

```
// generate random numbers
for (cc = 0; cc < SIZE; cc++)
  data[cc] = rand() % TOP;

// take action depending on their value
for (ii = 0; ii < SIZE; ii++)
  {
     t = (data[ii] - PIVOT -1) >> 31;
     sum += ~t & data[ii];
  }
```

➔ ...just have a look at the live results..

# Loops optimizations: branches

## gcc -o

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred

sum is 983597794767, elapsed seconds: 5.40445

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow

sum is 983597794767, elapsed seconds: 2.23186
(in total: 2.44473 seconds)

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart

sum is 983597794767, elapsed seconds: 2.8878

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2%
```

## gcc -03

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.03

sum is 983597794767, elapsed seconds: 0.660148

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.03

sum is 983597794767, elapsed seconds: 0.650005
(in total: 0.795181 seconds)

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.03

sum is 983597794767, elapsed seconds: 0.679286

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2%
```

# Loops optimizations: branches

## gcc -O3

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.03

sum is 983597794767, elapsed seconds: 0.660148

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.03

sum is 983597794767, elapsed seconds: 0.650005
(in total: 0.795181 seconds)

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.03

sum is 983597794767, elapsed seconds: 0.679286

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2%
```

## gcc -O3 -march=native

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.03n

sum is 983597794767, elapsed seconds 0.217864

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.03n

sum is 983597794767, elapsed seconds: 0.215645
(in total: 0.355377 seconds)

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.03n

sum is 983597794767, elapsed seconds: 0.224288
```

# Loops optimizations: branches

What changes in the base version with –O3 ? → conditional move

Modern CPUs have the capability of performing *conditional move*, i.e to execute **concurrently** both branches of a conditional – **if they are "simple enough"** – and to select the right result upon the evaluation of the conditional

```
perform op1 → res in AX
perform op2 → res in BX

compare

if flag     → mov BX in AX
```

HOWEVER: loops with conditionals **can not be fully vectorized !!**

# Loops optimizations: branches

Why the difference in the base v. between **-O3** and **-O3 -march=native** ?

## gcc -O3

```
.L8:
        movdqu  xmm0, XMMWORD PTR [rax]
        movdqu  xmm6, XMMWORD PTR [rax]
        movdqa  xmm2, xmm4
        add     rax, 16
        pcmpgtd xmm0, xmm5
        pand    xmm0, xmm6
        pcmpgtd xmm2, xmm0
        movdqa  xmm3, xmm0
        punpckldq       xmm3, xmm2
        punpckhdq       xmm0, xmm2
        paddq   xmm1, xmm3
        paddq   xmm1, xmm0
        cmp     rax, rcx
        jne     .L8
```

compare **4 integers** at a time using xmmX registers, that are common to x86_64 architectures.

bytes reshuffling to add one `int` at a time. These are SSE 128-bits instructions

increase the counter by **4 `int`**

12 instructions to process 4 **`int`** (3 ins/element)

## gcc -O3 -march=native

```
.L8:
        vmovdqu         ymm2, YMMWORD PTR [rax]
        add             rax, 32
        vpcmpgtd        ymm0, ymm2, ymm3
        vpand           ymm0, ymm0, ymm2
        vpmovsxdq       ymm2, xmm0
        vextracti128    xmm0, ymm0, 0x1
        vpaddq          ymm1, ymm2, ymm1
        vpmovsxdq       ymm0, xmm0
        vpaddq          ymm1, ymm0, ymm1
        cmp             rax, rcx
        jne             .L8
```

bytes reshuffling inside regs to add one `int` at a time
The v prefix tells you these are AVX2 256-bits instr.

compare **8 integers** at a time using ymmX registers. This requires AVX2 that is set on by `-march=native` for this CPU

increase the counter by **8 `int`**

9 instructions to process 8 **`int`** (1.12 ins/element)

# Take-home message

## Comments on the previous slides

Your branch predictor is quite good – as good as being right about ~95% of times in real codes – but still can not be right all the times. And penalties in this cases are quite severe (*).

Compiler's optimization in this simple case is able to convert the conditional in a `cmov`-style *when it is available*, which *may* makes the code as efficient as with the non-branching versions (or even more, because it is simpler).

Letting the compiler using the best instructions for the CPU also translate in partial vectorization, that further push performance.

(*) refer to the section on pipelines to understand why in more detail.

# Take-home message

The example before was purposely built..

Normally it does not happen that you have cases so perfectly suited for some kind of bitwise trickery.
Then, provided that you should always catch the opportunity, there is another, more general, way.

That amounts to re-define your if condition in a different way: instead of having a branch that depends on the condition evaluation, express it in two different values to be used in the subsequent lines of code.

# Loops optimizations: branches

```
for (ii = 0; ii < SIZE; ii++)
  {
    if (data[ii] > PIVOT)
       sum += data[ii];
  }
```

Can be rephrased as

```
for (ii = 0; ii < SIZE; ii++)
  {
     acc = ( data[ii]>PIVOT )? data[ii] : 0;
     sum += acc;
  }
```

➔ …just have a look at the live results..

# Loops optimizations: branches

Let's look to another practical example

You have 2 arrays, A and B, and you want to swap their elements so that

$$A[i] \geq B[i]$$

for all *i*.
A straightforward implementation would be:

```
for (i = 0; i < SIZE; i++)
  {
       if ( A[i] < B[i] )
          {
              t = B[i];
              B[i] = A[i];
              A[i] = t; }
          }
```

# Loops optimizations: branches

However, that implementation suffers exactly of the same problem we have just discussed.
An alternative way to write the same code, but in a more effective style is:

```
for (i = 0; i < SIZE; i++)
  {
     int min = A[i] > B[i] ? B[i] : A[i];
     int max = A[i] >= B[i] ? A[i] : B[i];

      A[i] = max;
      B[i] = min;
   }
```

➔ …just have a look at the live results..

# Loops optimizations: branches

Different forms of the conditional statement we are considering. They may seem awkward, but let us look at them in more detail in the following slides.

**standard**

```
for (uint ii = 0; ii < SIZE; ii++)
  {
    if ( B[ii] > A[ii] )
      {
        int t = A[ii];
        A[ii] = B[ii];
        B[ii] = t;
      }
```

**smart2**

```
for (uint ii = 0; ii < SIZE; ii++)
  {
    int register t = -(A[ii]<B[ii]);
    int register x = A[ii]^B[ii];
    A[ii] = A[ii]^(x & t);
    B[ii] = B[ii]^(x & t);
```

**smart**

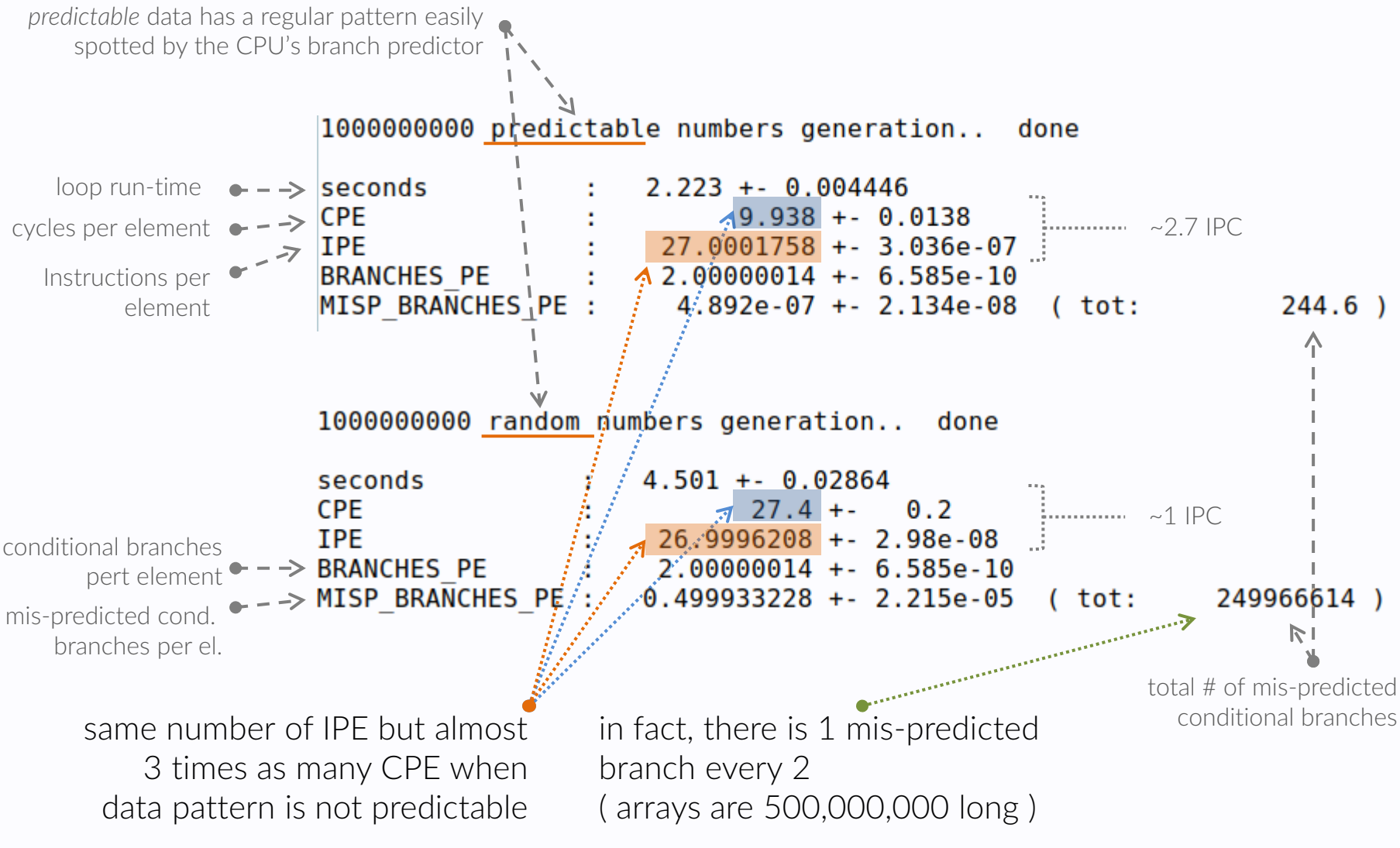
```
for (uint ii = 0; ii < SIZE; ii++)
  {
    int max = (A[ii]>B[ii])? A[ii]:B[ii];
    int min = (A[ii]>B[ii])? B[ii]:A[ii];
    A[ii] = max;
    B[ii] = min;
```

**smart3**

```
for (uint ii = 0; ii < SIZE; ii++)
  {
    int d = A[ii]-B[ii];
    d &= (d >> 31);
    A[ii] = A[ii] - d;
    B[ii] = B[ii] + d;
```

# Loops optimizations: branches

## Standard implementation, -O0



# Loops optimizations: branches

**`smart` implementation, `-O0`**

**`predictable`**

```
seconds           :    1.824 +- 0.05734
CPE               :       11.12 +- 0.341
IPE               :    34.0000018 +- 4.344e-08
BRANCHES_PE       :    1.00000014 +- 3.293e-10
MISP_BRANCHES_PE  :     9.94e-07 +- 3.207e-08   ( tot:          497 )
```

~3 IPC

number of IPE is larger than for standard case, but the CPE is stable !

mis-predicted branches are very few and comparable in both cases

**`random`**

```
seconds           :    1.857 +- 0.001762
CPE               :       11.32 +- 0.00192
IPE               :    34.0000018 +- 2.581e-08
BRANCHES_PE       :    1.00000014 +- 3.293e-10
MISP_BRANCHES_PE  :     9.512e-07 +- 4.987e-08   ( tot:        475.6 )
```

~3 IPC

# Loops optimizations: branches

**`smart2` implementation, `-O0`**

**`predictable`**

```
seconds           :    1.811 +- 0.002471
CPE               :          11.04 +- 0.00246
IPE               :     37.0000018 +- 3.799e-08
BRANCHES_PE       :     1.00000014 +- 3.293e-10
MISP_BRANCHES_PE  :     1.0236e-06 +- 7.194e-08  ( tot:          511.8 )
```

~3.4 IPC

number of IPE is [larger|smaller] than for [stdrd|smart] case, and the CPE is stable !

mis-predicted branches are still very few and comparable in both cases

**`random`**

```
seconds           :    1.745 +- 0.05743
CPE               :          10.63 +- 0.344
IPE               :     37.0000018 +- 4.344e-08
BRANCHES_PE       :     1.00000014 +- 3.293e-10
MISP_BRANCHES_PE  :      1.074e-06 +- 5.584e-08  ( tot:            537 )
```

~3.5 IPC

# Loops optimizations: branches

**`smart3` implementation, `-O0`**

**`predictable`**

```
seconds           :    1.628 +- 0.01015
CPE               :        9.993 +- 0.0464
IPE               :    32.0000017 +- 2.356e-08
BRANCHES_PE       :    1.00000014 +- 3.293e-10
MISP_BRANCHES_PE  :     3.728e-07 +- 8.898e-08   ( tot:          186.4 )
```

~3.2 IPC

number of IPE as smart case, larger than for stdrd, the CPE is stable

mis-predicted branches are still very few and comparable in both cases

**`random`**

```
seconds           :    1.688 +- 0.03554
CPE               :        10.36 +- 0.211
IPE               :    32.0000017 +- 2.98e-08
BRANCHES_PE       :    1.00000014 +- 3.293e-10
MISP_BRANCHES_PE  :       3.1e-07 +- 5.183e-08   ( tot:            155 )
```

~3.1 IPC

# Take-home message

## Comments on the previous slides

The standard implementation relies on the ability of your CPU's branch predictor to guess the correct data pattern.
When it is successful, it is **really** so (the lowest CPE and IPE).

However, whenever the data pattern is not guessable by the branch predictor things quickly become really weird.

Writing the code differently may make you loosing something in terms of CPE/IPE (with respect to the best possible standard case) but not really in terms of time-to-solution.
And, above all, the code behaviour is **stable** with both predictable and unpredictable patterns.

# A more detailed view

## `std` implementation, `-O0`

```
.L20:
        mov     eax, DWORD PTR -88[rbp] #  ii
        lea     rdx, 0[0+rax*4]
        mov     rax, QWORD PTR -56[rbp] #  B
        add     rax, rdx
        mov     edx, DWORD PTR [rax]
        mov     eax, DWORD PTR -88[rbp] #  ii
        lea     rcx, 0[0+rax*4]
        mov     rax, QWORD PTR -64[rbp] #  A
        add     rax, rcx
        mov     eax, DWORD PTR [rax]
# branchpred2.c:116:             if ( B[ii] > A[ii] )
        cmp     edx, eax
        jle     .L19
# branchpred2.c:118:             int t = A[ii];
        mov     eax, DWORD PTR -88[rbp] #  ii
        lea     rdx, 0[0+rax*4]
        mov     rax, QWORD PTR -64[rbp] #  A
        add     rax, rdx
# branchpred2.c:118:             int t = A[ii];
        mov     eax, DWORD PTR [rax]
        mov     DWORD PTR -84[rbp], eax # t
# branchpred2.c:119:             A[ii] = B[ii];
        mov     eax, DWORD PTR -88[rbp] # ii
        lea     rdx, 0[0+rax*4]
        mov     rax, QWORD PTR -56[rbp] #  B
        add     rax, rdx        #
# branchpred2.c:119:             A[ii] = B[ii];
        mov     edx, DWORD PTR -88[rbp] # ii
        lea     rcx, 0[0+rdx*4]
        mov     rdx, QWORD PTR -64[rbp] # A
        add     rdx, rcx
# branchpred2.c:119:             A[ii] = B[ii];
        mov     eax, DWORD PTR [rax]
# branchpred2.c:119:             A[ii] = B[ii];
        mov     DWORD PTR [rdx], eax
# branchpred2.c:120:             B[ii] = t;
        mov     eax, DWORD PTR -88[rbp] # ii
        lea     rdx, 0[0+rax*4] ,
        mov     rax, QWORD PTR -56[rbp] # B
        add     rdx, rax        #
# branchpred2.c:120:             B[ii] = t;
        mov     eax, DWORD PTR -84[rbp] # t
        mov     DWORD PTR [rdx], eax
.L19:
# branchpred2.c:112:         for (uint ii = 0; ii < SIZE; ii++)
        add     DWORD PTR -88[rbp], 1   # ii,
.L18:
# branchpred2.c:112:         for (uint ii = 0; ii < SIZE; ii++)
        mov     eax, DWORD PTR -88[rbp] # ii
        cmp     eax, DWORD PTR -120[rbp]        # SIZE
        jb      .L20
```

1 `cmp` instr. with a following jump

2 `cmov` instr. can use 2 pipelines

## `smart` implementation, `-O0`

```
.L19:
# branchpred2.c:122:         max = A[ii] > B[ii] ? A[ii] : B[ii];
        mov     eax, DWORD PTR -92[rbp] # tmp229, ii
        cdqe
        lea     rdx, 0[0+rax*4]
        mov     rax, QWORD PTR -56[rbp] #  B
        add     rax, rdx
        mov     edx, DWORD PTR [rax]
        mov     eax, DWORD PTR -92[rbp] #  ii
        cdqe
        lea     rcx, 0[0+rax*4] # _51,
        mov     rax, QWORD PTR -64[rbp] #  A
        add     rax, rcx
        mov     eax, DWORD PTR [rax]
# branchpred2.c:122:         max = A[ii] > B[ii] ? A[ii] : B[ii];
        cmp     edx, eax
        cmovge  eax, edx
        mov     DWORD PTR -88[rbp], eax # max
# branchpred2.c:123:         min = A[ii] < B[ii] ? A[ii] : B[ii];
        mov     eax, DWORD PTR -92[rbp] # ii
        cdqe
        lea     rdx, 0[0+rax*4] # _55,
        mov     rax, QWORD PTR -56[rbp] #  B
        add     rax, rdx        # _56, _55
        mov     edx, DWORD PTR [rax]
# branchpred2.c:123:         min = A[ii] < B[ii] ? A[ii] : B[ii];
        mov     eax, DWORD PTR -92[rbp] #  ii
        cdqe
        lea     rcx, 0[0+rax*4] # _59,
        mov     rax, QWORD PTR -64[rbp] #  A
        add     rax, rcx
        mov     eax, DWORD PTR [rax]
# branchpred2.c:123:         min = A[ii] < B[ii] ? A[ii] : B[ii];
        cmp     edx, eax
        cmovle  eax, edx
        mov     DWORD PTR -84[rbp], eax # min
# branchpred2.c:131:         A[ii] = max;
        mov     eax, DWORD PTR -92[rbp] #  ii
        cdqe
        lea     rdx, 0[0+rax*4]
        mov     rax, QWORD PTR -64[rbp] #  A
        add     rdx, rax
# branchpred2.c:131:         A[ii] = max;
        mov     eax, DWORD PTR -88[rbp] #  max
        mov     DWORD PTR [rdx], eax
# branchpred2.c:132:         B[ii] = min;
        mov     eax, DWORD PTR -92[rbp] #  ii
        cdqe
        lea     rdx, 0[0+rax*4] # _66,
        mov     rax, QWORD PTR -56[rbp] #  B
        add     rdx, rax
# branchpred2.c:132:         B[ii] = min;
        mov     eax, DWORD PTR -84[rbp] #  min
        mov     DWORD PTR [rdx], eax
# branchpred2.c:107:         for (ii = 0; ii < SIZE; ii++)
        add     DWORD PTR -92[rbp], 1   # ii,
.L18:
# branchpred2.c:107:         for (ii = 0; ii < SIZE; ii++)
        mov     eax, DWORD PTR -92[rbp] #  ii
        cmp     eax, DWORD PTR -104[rbp] #  SIZE
        jl      .L19
```

# Loops optimizations: branches

All optimizations at a glance, with *predictable* data

| opt | smart | data | time +- | err | CPE +- | err | INS +- | err | BRE |
|---|---|---|---|---|---|---|---|---|---|
| O0 | N | P | 1.697e+00 | 2.876e-03 | 1.041e+01 | 6.740e-04 | 2.700e+01 | 3.650e-08 | 2.000e+00 |
| O0 | smart | P | 1.859e+00 | 5.434e-02 | 1.118e+01 | 3.320e-01 | 3.400e+01 | 3.406e-07 | 1.000e+00 |
| O0 | smart2 | P | 1.815e+00 | 2.680e-03 | 1.114e+01 | 9.030e-03 | 3.700e+01 | 3.942e-08 | 1.000e+00 |
| O0 | smart3 | P | 1.628e+00 | 1.015e-02 | 9.993e+00 | 4.640e-02 | 3.200e+01 | 2.356e-08 | 1.000e+00 |
| O3 | N | P | 6.448e-01 | 3.248e-03 | 3.578e+00 | 2.550e-03 | 8.000e+00 | 2.849e-08 | 2.000e+00 |
| O3 | smart | P | 4.548e-01 | 7.833e-03 | 2.064e+00 | 2.040e-02 | 4.250e+00 | 3.953e-08 | 2.500e-01 |
| O3 | smart2 | P | 4.506e-01 | 1.419e-03 | 2.109e+00 | 1.650e-02 | 4.500e+00 | 3.723e-08 | 2.500e-01 |
| O3 | smart3 | P | 4.332e-01 | 2.937e-03 | 1.905e+00 | 6.180e-03 | 3.500e+00 | 2.542e-08 | 2.500e-01 |
| O3n | N | P | 3.518e-01 | 1.096e-02 | 1.357e+00 | 1.410e-02 | 1.125e+00 | 4.850e-09 | 2.500e-01 |
| O3n | smart | P | 3.905e-01 | 2.258e-03 | 1.367e+00 | 1.030e-02 | 1.125e+00 | 2.643e-08 | 1.250e-01 |
| O3n | smart2 | P | 4.114e-01 | 1.941e-02 | 1.671e+00 | 5.650e-02 | 1.750e+00 | 3.118e-08 | 1.250e-01 |
| O3n | smart3 | P | 4.119e-01 | 3.449e-03 | 1.523e+00 | 1.500e-02 | 1.500e+00 | 1.613e-09 | 1.250e-01 |

The standard implementation (label "N" in the table) exhibits a better behaviour at `-O0` considering CPE and above all IPE (label "INS" in the table), whereas run times are comparable among different variants (std, smart, smart2 and smart3).

However, at `-O3` its behaviour is the worst one, with CPE larger by ~75% and IPE larger by a factor of ~2.

Only with very aggressive optimization the compiler can generate a code comparable with the smart*X* ones

# Loops optimizations: branches

All optimizations at a glance, with *non predictable* data

| opt | smart | data | time +- | err | CPE +- | err | INS +- | err | BRE |
|---|---|---|---|---|---|---|---|---|---|
| O0 | N | R | 4.426e+00 | 2.966e-02 | 2.725e+01 | 1.830e-01 | 2.700e+01 | 2.788e-08 | 2.000e+00 |
| O0 | smart | R | 1.855e+00 | 1.822e-03 | 1.139e+01 | 1.860e-03 | 3.400e+01 | 3.161e-08 | 1.000e+00 |
| O0 | smart2 | R | 1.718e+00 | 5.001e-02 | 1.055e+01 | 2.940e-01 | 3.700e+01 | 3.942e-08 | 1.000e+00 |
| O0 | smart3 | R | 1.688e+00 | 3.554e-02 | 1.036e+01 | 2.110e-01 | 3.200e+01 | 2.980e-08 | 1.000e+00 |
| O3 | N | R | 2.306e+00 | 1.549e-02 | 1.418e+01 | 8.650e-02 | 8.000e+00 | 4.292e-08 | 2.000e+00 |
| O3 | smart | R | 4.178e-01 | 3.808e-02 | 2.138e+00 | 7.510e-02 | 4.250e+00 | 3.838e-08 | 2.500e-01 |
| O3 | smart2 | R | 4.517e-01 | 1.640e-03 | 2.098e+00 | 1.450e-02 | 4.500e+00 | 2.644e-08 | 2.500e-01 |
| O3 | smart3 | R | 4.321e-01 | 2.602e-03 | 1.910e+00 | 1.260e-02 | 3.500e+00 | 2.710e-08 | 2.500e-01 |
| O3n | N | R | 4.178e-01 | 3.130e-03 | 1.600e+00 | 6.140e-03 | 1.249e+00 | 2.661e-08 | 2.500e-01 |
| O3n | smart | R | 3.918e-01 | 1.255e-03 | 1.363e+00 | 1.260e-02 | 1.125e+00 | 2.602e-08 | 1.250e-01 |
| O3n | smart2 | R | 4.107e-01 | 1.860e-02 | 1.653e+00 | 3.830e-02 | 1.750e+00 | 9.429e-09 | 1.250e-01 |
| O3n | smart3 | R | 4.131e-01 | 1.929e-03 | 1.516e+00 | 9.290e-03 | 1.500e+00 | 1.397e-09 | 1.250e-01 |

When dealing with random data, the difference between std implementation and the other ones is even more obvious up to `-O3`, with the CPE being larger by a factor of ~3 and ~4 than in predictable data case at `-O0` and `-O3` respectively.

With very aggressive optimization the compiler can generate a code comparable with the smart*X* ones (for this very simple code snippet).

# Loops optimizations: branches

Using the PGI compiler..

| opt | smart | data | time | +- | err | CPE | +- | err | INS | +- | err | BRE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| O0 | N | P | 9.380e-01 | +- | 5.053e-03 | 5.443e+00 | +- | 4.480e-03 | 1.700e+01 | +- | 3.373e-08 | 2.000e+00 |
| O0 | N | R | 3.263e+00 | +- | 1.724e-02 | 1.982e+01 | +- | 2.920e-02 | 1.700e+01 | +- | 7.580e-08 | 2.000e+00 |
| O0 | smart | P | 1.671e+00 | +- | 5.070e-02 | 1.016e+01 | +- | 3.010e-01 | 3.300e+01 | +- | 2.788e-08 | 3.000e+00 |
| O0 | smart | R | 4.132e+00 | +- | 4.019e-02 | 2.525e+01 | +- | 2.350e-01 | 3.300e+01 | +- | 4.344e-08 | 3.000e+00 |
| O0 | smart2 | P | 1.848e+00 | +- | 3.021e-03 | 1.126e+01 | +- | 1.530e-02 | 3.000e+01 | +- | 2.471e-08 | 1.000e+00 |
| O0 | smart2 | R | 1.775e+00 | +- | 6.319e-02 | 1.082e+01 | +- | 3.790e-01 | 3.000e+01 | +- | 6.909e-08 | 1.000e+00 |
| O0 | smart3 | P | 1.362e+00 | +- | 6.870e-02 | 8.283e+00 | +- | 4.140e-01 | 2.300e+01 | +- | 3.573e-08 | 1.000e+00 |
| O0 | smart3 | R | 1.462e+00 | +- | 2.043e-03 | 8.873e+00 | +- | 8.040e-03 | 2.300e+01 | +- | 4.012e-08 | 1.000e+00 |
| O3 | N | P | 5.530e-01 | +- | 1.608e-03 | 3.010e+00 | +- | 6.270e-03 | 7.500e+00 | +- | 5.952e-08 | 2.000e+00 |
| O3 | N | R | 2.343e+00 | +- | 1.076e-02 | 1.428e+01 | +- | 6.810e-02 | 7.500e+00 | +- | 2.998e-08 | 2.000e+00 |
| O3 | smart | P | 3.788e-01 | +- | 2.156e-03 | 1.772e+00 | +- | 4.020e-02 | 1.875e+00 | +- | 1.867e-08 | 2.500e-01 |
| O3 | smart | R | 3.780e-01 | +- | 1.939e-03 | 1.769e+00 | +- | 2.440e-02 | 1.875e+00 | +- | 1.073e-08 | 2.500e-01 |
| O3 | smart2 | P | 4.013e-01 | +- | 1.917e-03 | 2.210e+00 | +- | 4.540e-03 | 3.125e+00 | +- | 4.165e-09 | 2.500e-01 |
| O3 | smart2 | R | 4.011e-01 | +- | 1.736e-03 | 2.212e+00 | +- | 6.520e-03 | 3.125e+00 | +- | 1.863e-09 | 2.500e-01 |
| O3 | smart3 | P | 3.862e-01 | +- | 4.548e-03 | 2.080e+00 | +- | 2.630e-02 | 2.375e+00 | +- | 9.701e-09 | 2.500e-01 |
| O3 | smart3 | R | 3.873e-01 | +- | 2.408e-03 | 2.067e+00 | +- | 9.120e-03 | 2.375e+00 | +- | 2.734e-08 | 2.500e-01 |
| O3n | N | P | 5.538e-01 | +- | 2.179e-03 | 3.012e+00 | +- | 5.910e-03 | 7.500e+00 | +- | 1.070e-08 | 2.000e+00 |
| O3n | N | R | 2.403e+00 | +- | 1.996e-02 | 1.464e+01 | +- | 1.010e-01 | 7.500e+00 | +- | 3.822e-08 | 2.000e+00 |
| O3n | smart | P | 3.759e-01 | +- | 2.942e-03 | 1.776e+00 | +- | 2.770e-02 | 1.875e+00 | +- | 3.132e-08 | 2.500e-01 |
| O3n | smart | R | 3.831e-01 | +- | 2.355e-03 | 1.746e+00 | +- | 1.320e-02 | 1.875e+00 | +- | 7.552e-09 | 2.500e-01 |
| O3n | smart2 | P | 4.047e-01 | +- | 4.673e-03 | 2.225e+00 | +- | 9.760e-03 | 3.125e+00 | +- | 1.050e-08 | 2.500e-01 |
| O3n | smart2 | R | 4.019e-01 | +- | 1.069e-03 | 2.214e+00 | +- | 7.920e-03 | 3.125e+00 | +- | 2.281e-09 | 2.500e-01 |
| O3n | smart3 | P | 3.848e-01 | +- | 3.987e-03 | 2.054e+00 | +- | 1.290e-02 | 2.375e+00 | +- | 3.029e-08 | 2.500e-01 |
| O3n | smart3 | R | 3.836e-01 | +- | 1.914e-03 | 2.048e+00 | +- | 3.770e-03 | 2.375e+00 | +- | 2.684e-08 | 2.500e-01 |

# Take-home message

## Comments on the previous slides

As we have seen, the `gcc` compiler can generate a code comparable to what it does with smart*X* variants only with very aggressive optimization and using AVX 256-bits instructions.

When random data are used, the standard implementation exhibits a behaviour that is much worse than with data with a predictable pattern, whereas trying not to use conditional branches generates a more stable code.

In this case, `pgi` compiler proves to be less able than gcc to generate optimal code, with –O3n level (*) still having a pronounced spike in CPE for random data.

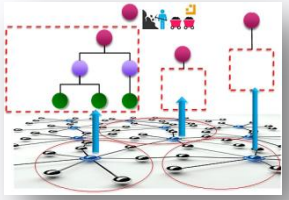
Bottom-line is: **do not take it for granted that you lit up the compiler's optimization and everything will go seamlessly towards a triumph**.

(*) actually it is `-O4 -fast -tp haswell -Mvect=simd:256`

# A word of caution

It may be really easy to get lost in "optmization", in hunting every single line wondering why some incredible trick that – you're convinced – should work, actually does not.

"Optimization" includes also optimizing your effort and your time, so always **remember that the far most important ingredients** are:



The algorithms that you choose

The data model you design

The overall quality, cleanness and robustness of your code

That said, being really conscious of what happen at low-level helps you in writing an high-quality, clean and robust code.

# Loops optimizations: branches

A last example.. about the fact that design and simplicity are the best move

Just changing point of view sometimes may help:

```
for ( j = 0; j < N; j++ )
    for ( i = 0; i < M; i++ )
        {
            if ( i > j )
               matrix[j][i] = 1.0;
            else if ( i < j )
               matrix[j][i] = -1.0;
            else
               matrix[j][i] = 0.0;
         }
```

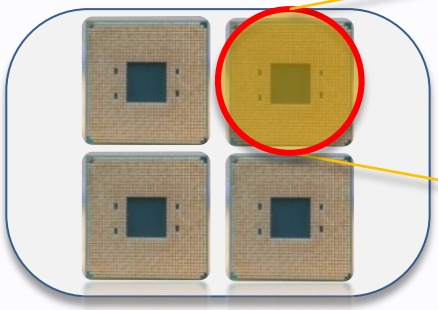
# Loops optimizations: branches

Can be easily re-written with no conditional evaluations

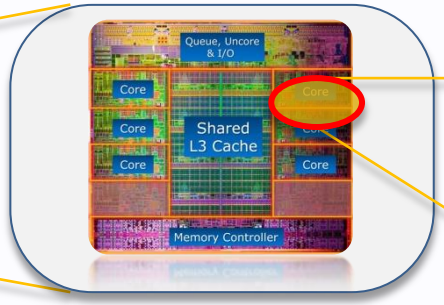
```
for ( j = 0; j < N; j++ )
   {
       for ( i = 0; i < j; i++ )
           matrix[j][i] = -1.0;

       matrix[j][i] = 0.0;

       for ( i = j+1; i < M; i++ )
            matrix[j][i] = 1.0;
   }
```

# Pipelining

# Architectural hierarchy of modern CPUs

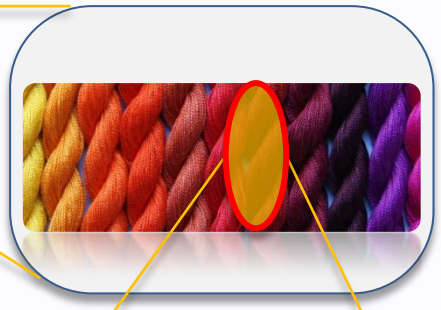


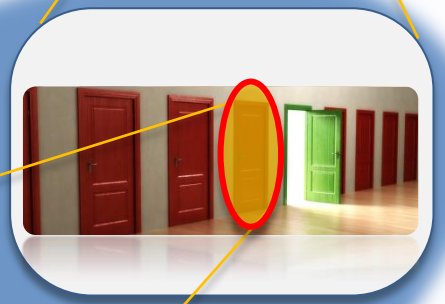
sockets level




Many cores per socket

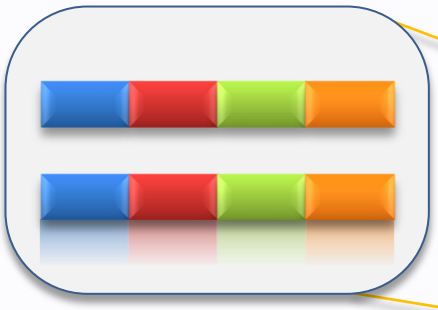
Many threads





Pipelining capacity

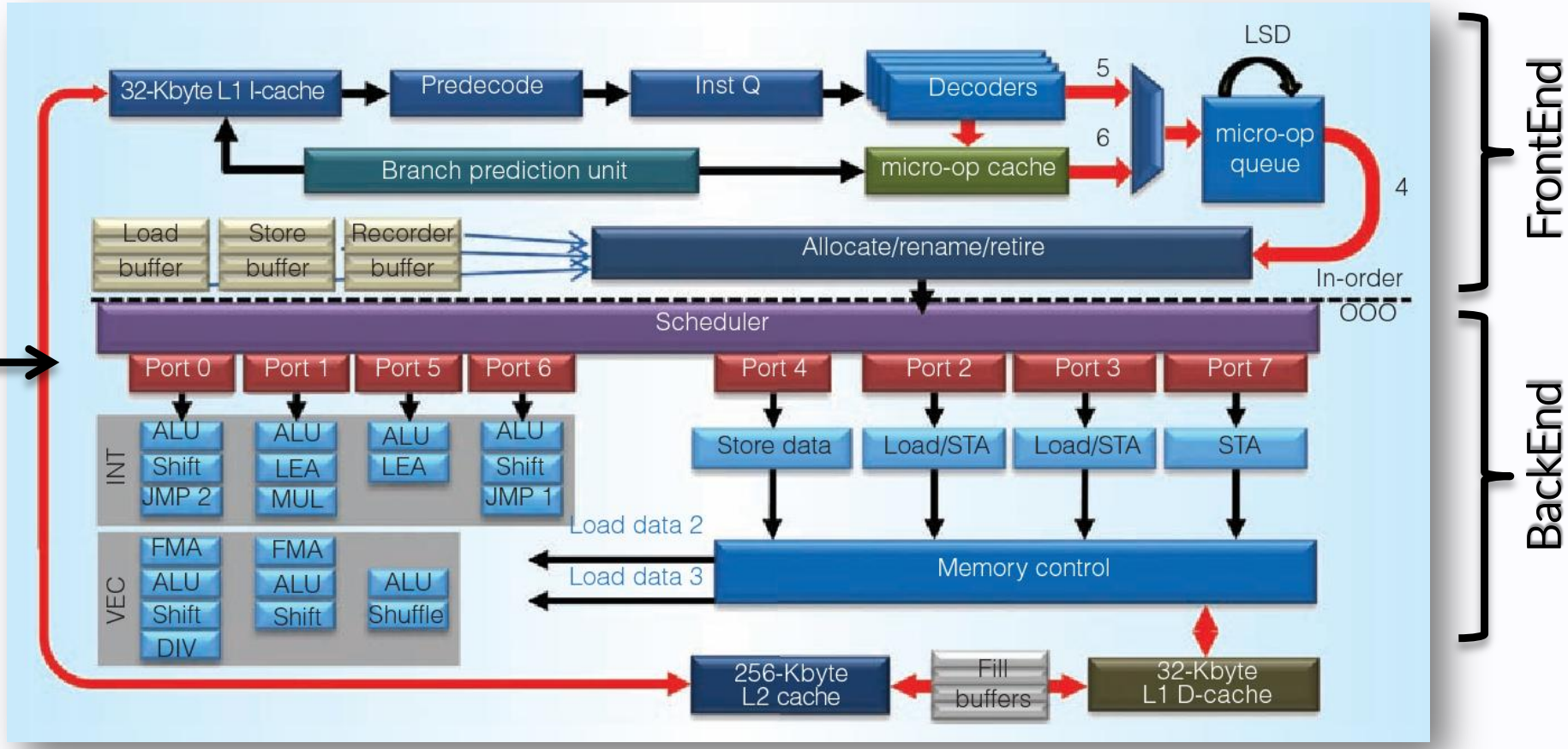
Superscalar capacity



Vectorisation capacity

# Architectural hierarchy of modern Cores

More than 1 *port* is available to execute CPU instructions, although different units have different specializations (ALU, LEA, SHIFT, FMA, … ) : that is superscalar capacity, i.e. the capacity of executing more than 1 instructions per cycle.



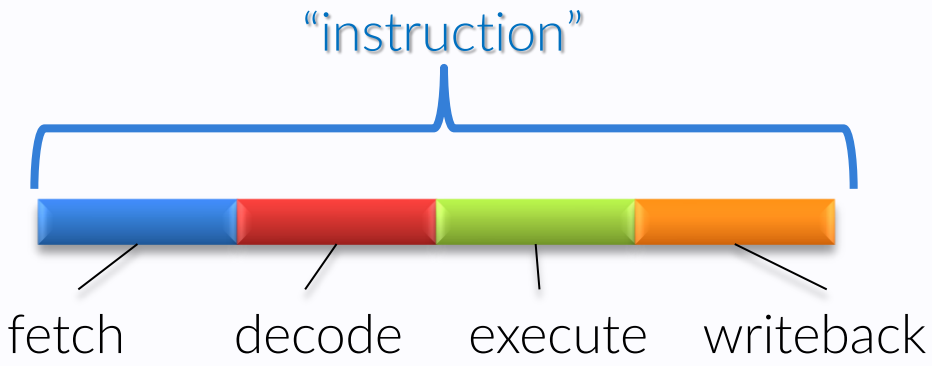

6th generation SkyLake core micro-arch.

# Pipelines

It would be obvious to think that an "instruction" is a kind of *atomic* operation that the CPU perform as a whole. Indeed that was true until the mid of the 80s.

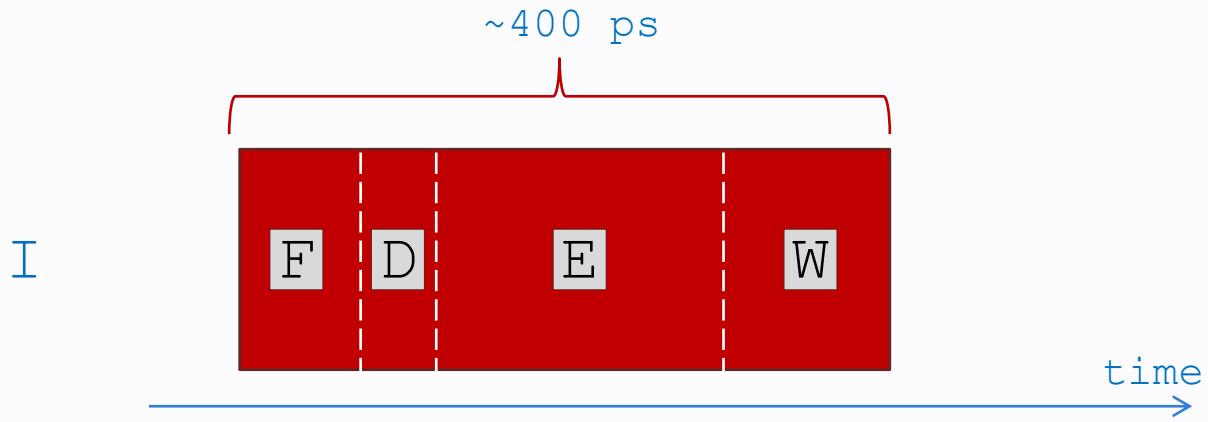
If you think carefully about it, it is easy to understand that actually an "instruction" involves at least the following ***independent*** steps:

1. It must be recalled from memory/Icache (***fetching***)
2. It must be "understood and interpreted" by the CPU (***decoding***)
3. It must be ***executed***
4. The result must be accounted in memory (***writeback***)



# Pipelines

Let us consider the instruction as a whole:

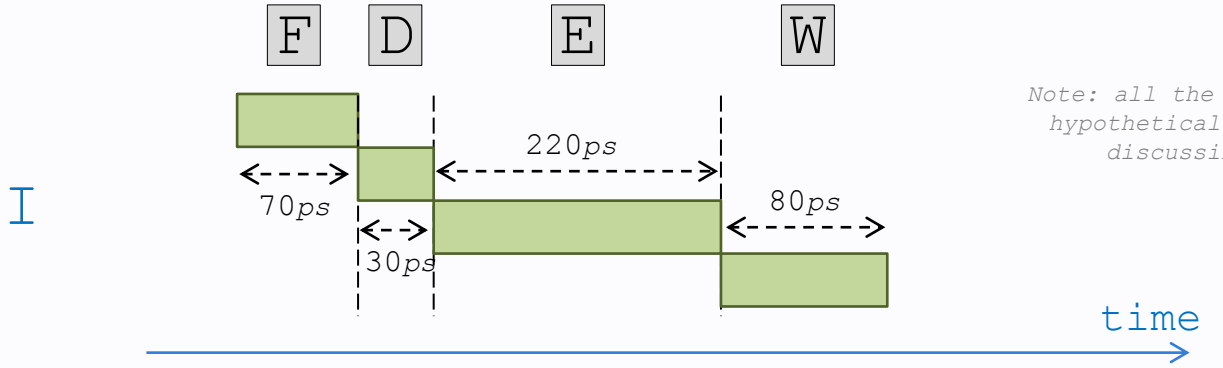



If all the four stages take ~400ps, we then would obtain a **throughput** of 2.5GIPS (giga-instructions per second).
400ps is also the total time required to get a result from an instruction, and so it is the **latency** of the instruction.

However, if we were able to "detach" the four stages, we could organize things differently, like in a car-building chain, or even at the mensa of the university.

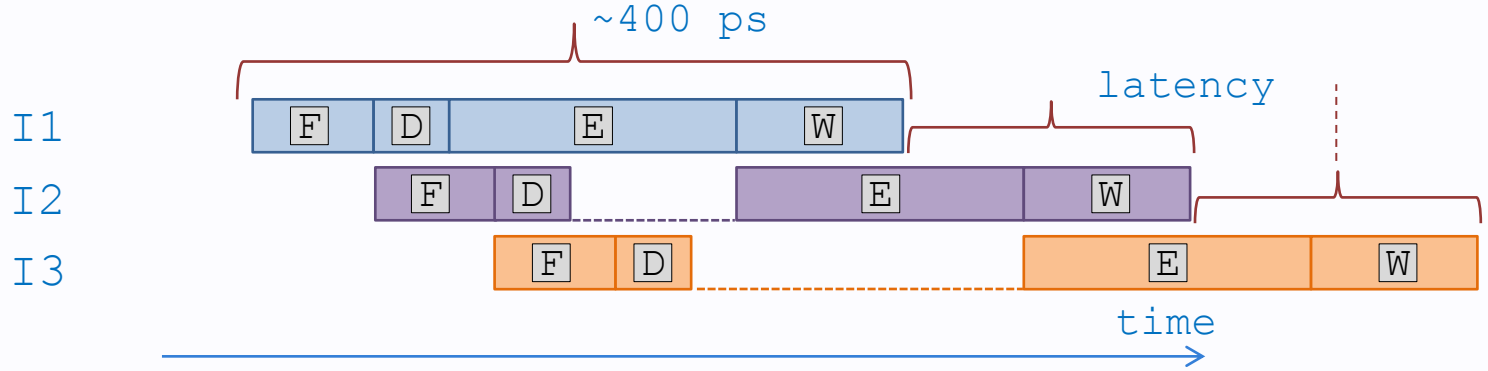
# Pipelines

If different and independent logical units of the CPU are in charge of each step:




*Note: all the timing estimates are hypothetical for the purpose of discussing the concepts*

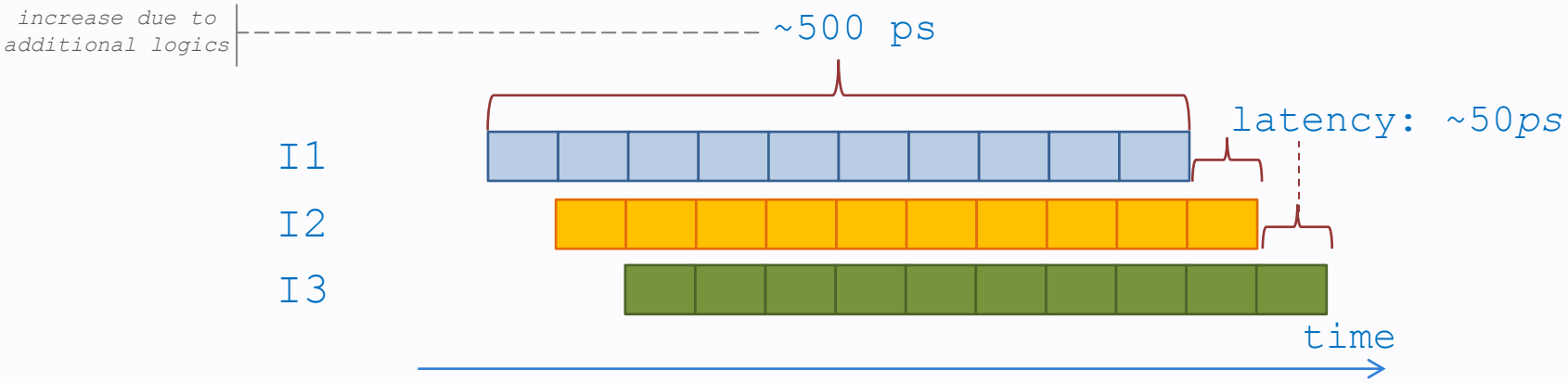
they actually could operate subsequently on **different instructions**:




If the stage delays are not uniform, the throughput is limited by the latency `F+(D+E)-(F+D) = E` ~ 220*ps*, which means we have a throughput of ~**4.5**GIPS just because of logic units separation.

# Pipelines

Therefore, introducing the instructions pipelining, we can increase the **throughput** of our system by a large factor.
However, the efficiency of the pipelines is limited by its longest stage: the better option would be to have all equal stages, for instance further subdividing each stage – especially the most demanding ones (*) :



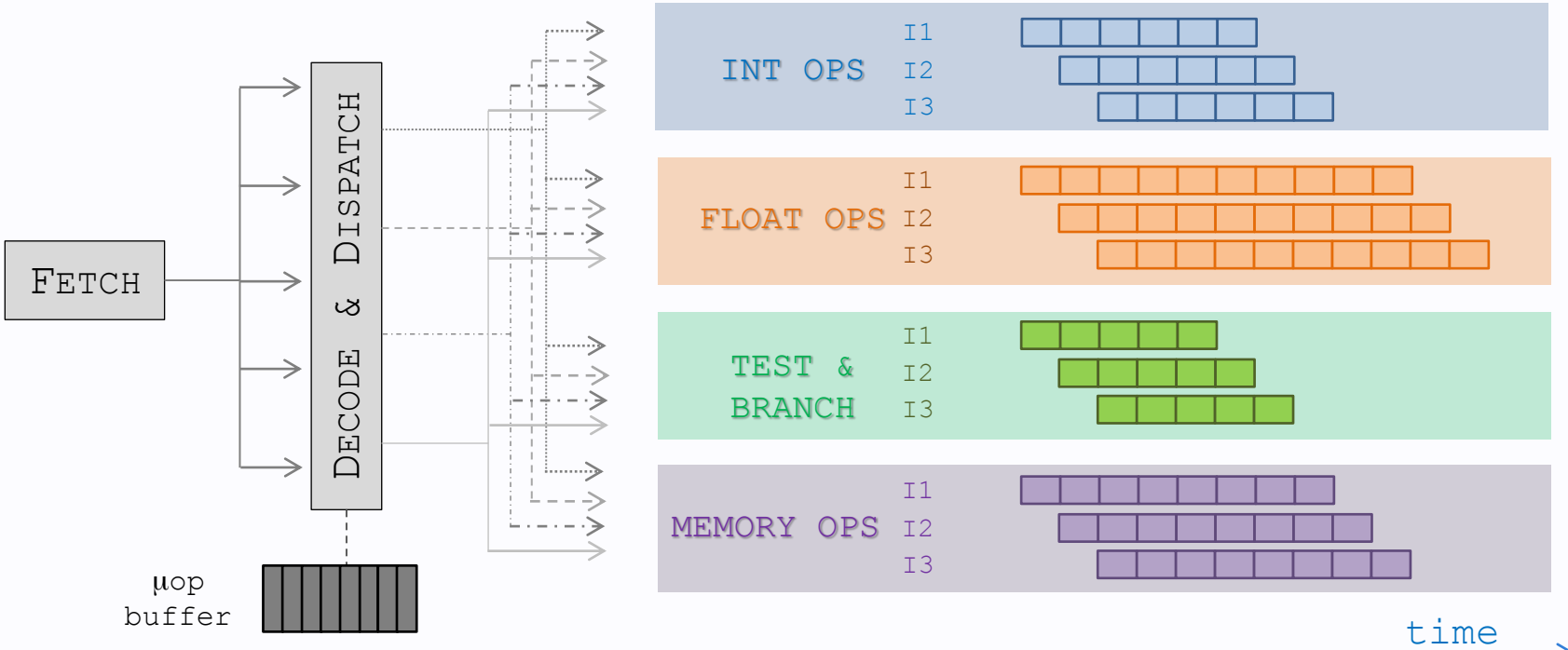

Now the throughput of our system has increased to 1 instruction retired every 50*ps*, i.e. 20GIPS ( *note: these figures are not real, they have only an illustrative purpose*).

(*) this is called *superpipelining*: modern CPUs may have 10-20 stages per pipeline.

# Pipelines and superscalar processors

Pipelining is then about increasing the throughput of a system, and as we have seen it could be really effective. However, there is more that can be done working on the **execution stage** that, of course, encompasses a large number of different ***functional units***, performing a **different and independent tasks**.

With an enhancement of the decode/dispatch stage, we can address multiple pipelines that can be active at the same time:



# Pipeline hazards

Pipelining a system with *feedbacks* (i.e.: in which a result from an instruction can be the input of a following instruction, or can change the workflow) can expose the system to *hazards* when there are dependencies between successive instructions.

There are two forms of dependency:

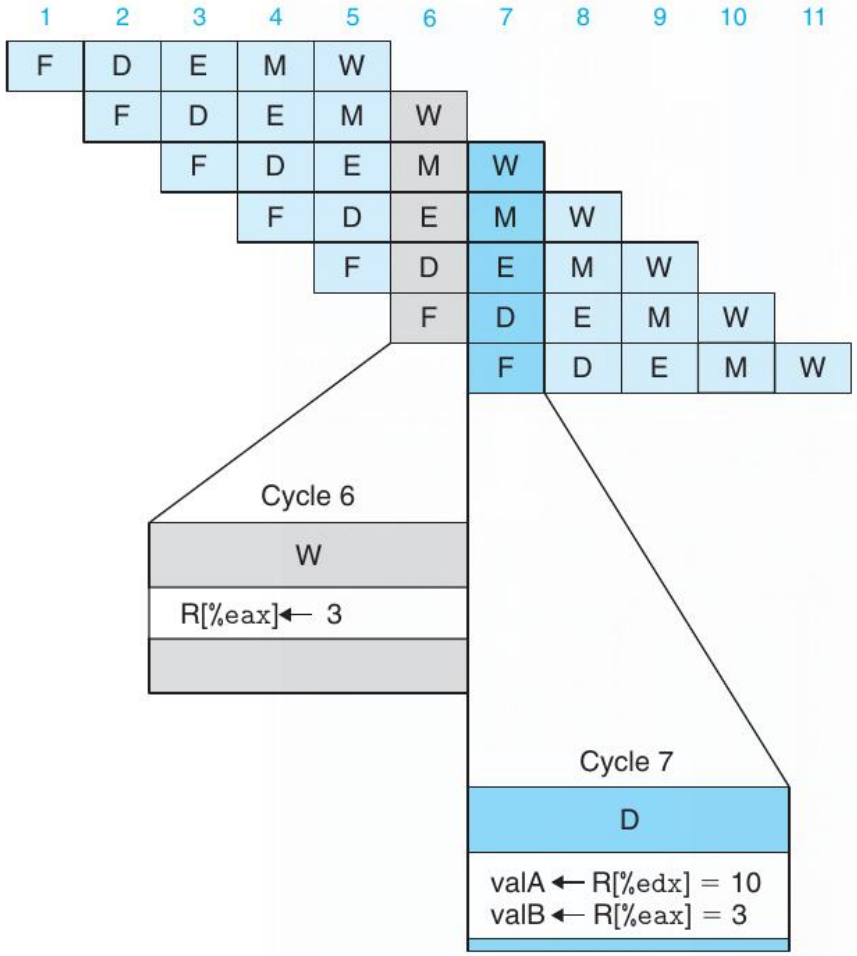
1. *data dependency*: the result computed by the `i`*th* instruction are used by one or more following instructions;
2. *control dependency*: the result of an instruction determines the value of the `pc`, i.e. the location of the next instruction to be executed.

When a dependency can lead to a wrong result, it is called an *hazard*.

# Pipeline hazards

It is not of your interest in this course to discuss the details of how to implement the pipelines and logics in CPUs.
However, let's just discuss an example (*)

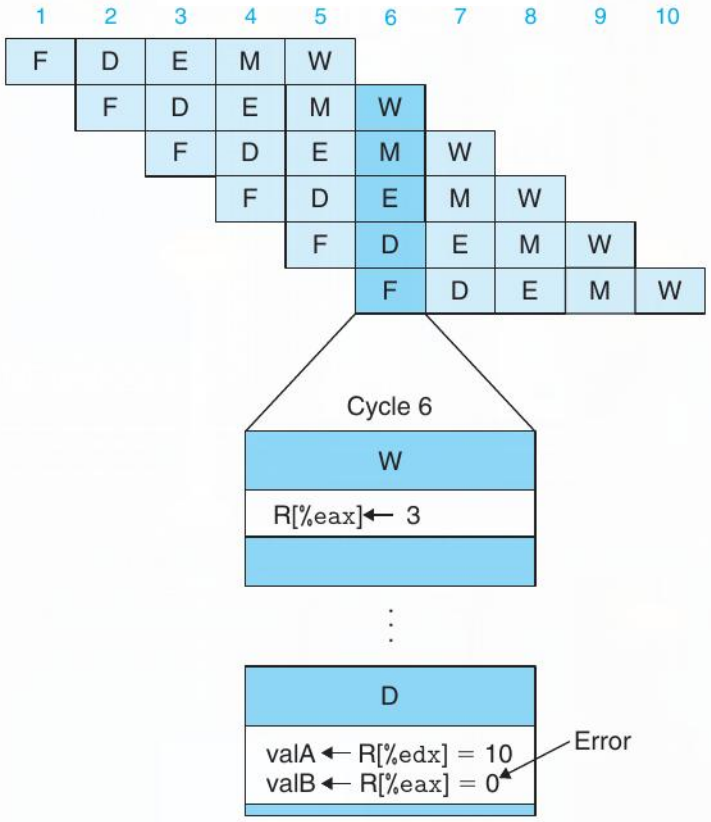
```
# progl
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl %edx,%eax
0x011: halt
```



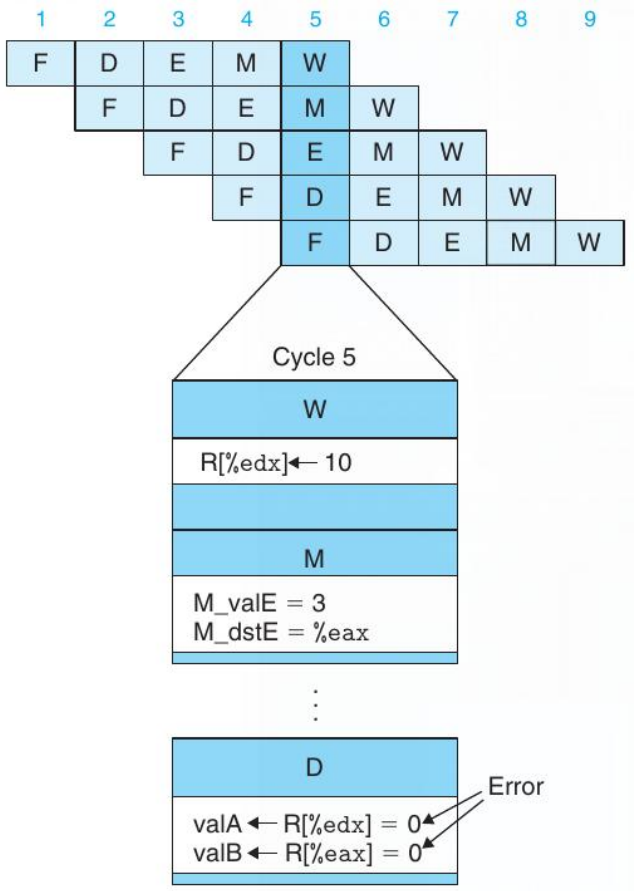

(*) taken from *computer systems: a programmers perspective, Pearson, 2015*

# Pipeline hazards

```
# prog2
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```




```
# prog3
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: addl %edx,%eax
0x00f: halt
```



# Pipeline hazards

So, whenever there is a data dependency hazard, the processors injects *bubbles* in the pipeline, during which no work gets done.

The number of cycles between when an instruction' execute stage begins and when its result is available for other instructions usage, is the *latency* of the instruction and grows with the pipeline deepness (the number of stages).

Typical latencies in modern processors (read the manual of yours) range from 1cyc for integer ops, to 3-6cyc for add and mul flop to ≥10-20 for div flop, ~50 for trig flop.

Latency for memory loads can be severely troublesome, because they are highly unpredictable and they block all other operations making it difficult to fill the delay with some other ops.

# Pipeline hazards

If a control hazard arise, the entire pipeline content has to be flushed away, and a new instruction has to start from the beginning, then wasting a lot of CPU cycles.

This is one of the main cause of performance loss in modern superscalar-superpipeline-out-of-order CPUs. The logics for the runtime branch predictor occupies a large space on processor chips but it definitely is worth it.

Best branch predictors are as good as 95% of accuracy: nonetheless, the branch mis-prediction, or branch miss, determines a huge performance loss: typical values for penalty are 10-20 cycles!

That is because the longer the pipeline, the further in the future you have to scrutinize the flow, the more difficult it is and the larger will be the misprediction penalty.

# Pipeline hazards

A very long pipeline, then, is not much more effective than a shorter one due to the real intrinsic nature of the codes that run on the CPUs.

The hazards we have just described are actually very common (very rarely a program is a stream of totally independent instructions with no jumps) and so the full exploitation of superscalarity+superpipelining is never reached.

Basically, that's the reason why we do not have 100-stage deep pipelines.

# Loops optimizations: pipelining

As we have seen, modern processor may be able to “perform more than one operations at a time”, through super-pipelining operations.

As for what concerns the programmer’s perspective, codes must be written so to make the pipelines saturation as effective as possible.

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

This way S is both read and written (the S as input in iteration *i* depends on S as output of iteration *i-1* ) and the FP pipeline is difficult to be exploited (iteration *i* must unavoidably wait for iteration *i-1* to end, which takes ~3-6 cycles at least)

# Loops optimizations: pipelining

Make things easier for the compiler and the CPU

```
for (i = 0; i < N; i += 2)  {
    sum0 += a[i] * b[i];
    sum1 += a[i+1] * b[i+1];  }
```

v. A

Unrolling make it easier for the CPU to saturate the pipelines.
Just try to separate load / multiply from addition:

```
for (i = 0; i < N; i += 2)   {
    double tmp0 = a[i] * b[i];
    double tmp1 = a[i+1] * b[i+1];
    sum0 += tmp0;
    sum1 += tmp1; }
```

v. B

# Loops optimizations: pipelining

Make things easier for the compiler and the CPU

More unrolling may work better, it depends on the CPU

```
for (i = 0; i < N; i += 4)   {
     sum0 += a[i] * b[i];
     sum1 += a[i+1] * b[i+1];
     sum2 += a[i+2] * b[i+2];
     sum3 += a[i+3] * b[i+3]; }
```

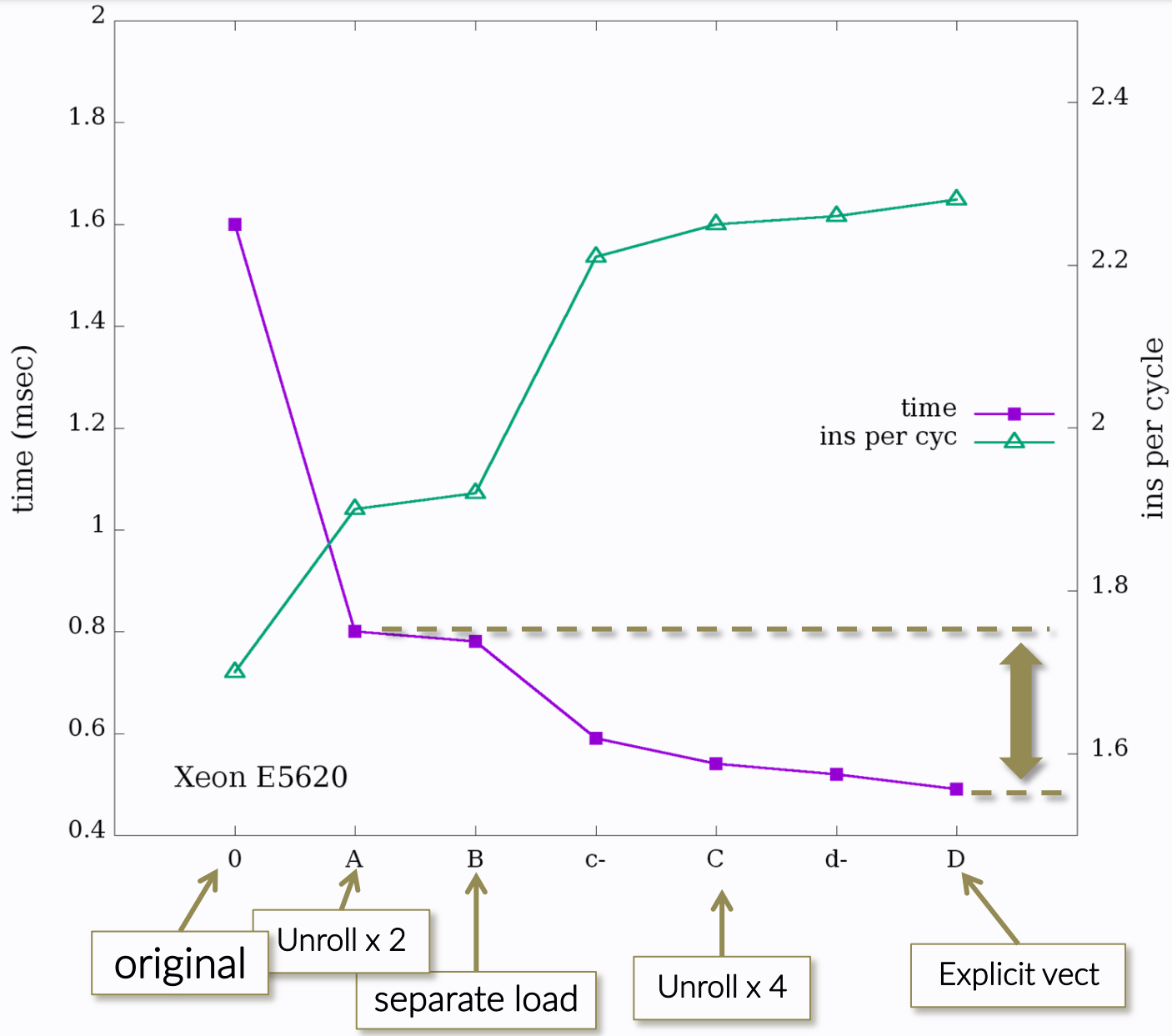
v. C

Or, eventually, let the explicit vectorization to step in

```
typedef double v4df __attribute__ ((vector_size (4*sizeof(double))));
v4df array1, array2;
v4df sum;
for (i = 0; i < N/4; i ++)
   sum += array1[i] * array2[i];
```

v. D

# Loops optimizations: pipelining



# Pipeline: hands-on

Let us now look in more detail to what happens at assembler level, so to understand in deeper detail why some implementations are more effective than others.

We will use –O0, for the usual reason: with such simple kernels, compilers are very good (not all equally good..) in optimizing the code, and differences among improved version are more vagues.

…that **does NOT** relieve you of knowing how to write not-that-bad code..

# Pipeline: hands-on

v0

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

# differences btw 2 compilers

v0

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

```
.LB3365:
##       for ( int i = 0; i < N; i++ )
        cmpl    %ebx, %r14d
        jge     .LB3366
##  lineno: 185
..LN20:

        movslq  %r14d, %rax
        movq    -72(%rbp), %rcx
        vmovsd  (%rcx,%rax,8), %xmm0

        vmovsd  -32(%rbp), %xmm1

        vfmadd132sd     (%r12,%rax,8), %xmm1, %xmm0

        vmovsd  %xmm0, -32(%rbp)


        addl    $1, %r14d
        jmp     .LB3365
```

```
.L8:
# v0.c:185:     sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163, sum
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp162
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:       for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164, i
        cmp     eax, DWORD PTR -148[rbp]        # tmp164, N
        jl      .L8
```

PGI 2018.10 | Gcc 8.2.0

# differences btw 2 compilers

```
.LB3365:
##      for ( int i = 0; i < N; i++ )
        cmpl    %ebx, %r14d
        jge     .LB3366
##  lineno: 185
..LN20:

        movslq  %r14d, %rax
        movq    -72(%rbp), %rcx
        vmovsd  (%rcx,%rax,8), %xmm0

        vmovsd  -32(%rbp), %xmm1

        vfmadd132sd     (%r12,%rax,8), %xmm1, %xmm0

        vmovsd  %xmm0, -32(%rbp)


        addl    $1, %r14d
        jmp     .LB3365
```

Note: no optimization has been switched on in both cases

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% pgi/v0_papi_O0 50000000
generating 100000000 numbers..done
sum is 1.24993e+07
time is 0.176009 (min 0.175734, std_dev 0.00054682, all 1.76643)
transfer rate was    4.23 GB/sec (  28.34% of theoretical max that is    15 GB/sec)
        IPC:    1
        [ time:  0.176sec   ins:     5e+08 ]
```

```
.L8:
# v0.c:185:     sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163, sum
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp162
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:       for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164, i
        cmp     eax, DWORD PTR -148[rbp]        # tmp164, N
        jl      .L8
```

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v0_papi_O0 50000000
generating 100000000 numbers..done
sum is 1.24993e+07
time is 0.190236 (min 0.189567, std_dev 0.000538453, all 1.90762)
transfer rate was    3.92 GB/sec (  26.22% of theoretical max that is    15 GB/sec)
        IPC:  1.9
        [ time: 0.1902sec   ins: 1.00158e+09 ]
```

# differences btw 2 compilers

## Short comment on the previous slides

The PGI compiler has opted for using specialised fuse multiply-addition instruction (`fmadd`) that allows for a more compact code, but an IPC of 1 (which basically means that the code occupies 1 pipeline at a time).

The gcc compiler, instead, opts for a redundant code (the 2 subsequent blocks `mov | cdqe | lea | mov | add | vmosd` ) that can involve 2 pipelines, and separate multiplication and addition.

Both compiler could be induced to generate different and more efficient code playing withtheir options (the default behaviour can differ among different compilers).

The result is more or less equivalent from the point of view of the run-time due to the fact that the code generated by gcc is longer (more instructions) but it is dispatched to more pipelines (larger IPC).

NOTE: the IPC is a good metric, but must be understood in a larger context. I.e. a larger IPC is good, but *how* good is the code depends on what instructions are being executed.

# turning on the optimization…

Let's have a preview of how the generated code changes..

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

-O0

```
.L8:
# v0.c:185:     sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163,
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:       for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164,
        cmp     eax, DWORD PTR -148[rbp]        # tmp164,
        jl      .L8
```

-O3 -march=native

```
.L8:
# v0.c:55:      sum += array1[i] * array2[i];
        vmovupd ymm5, YMMWORD PTR 0[r13+rax]
        vmulpd  ymm0, ymm5, YMMWORD PTR [r15+rax]
        add     rax, 32 # ivtmp.18,
        vaddsd  xmm4, xmm0, xmm4
        vunpckhpd       xmm1, xmm0, xmm0
        vextractf128    xmm0, ymm0, 0x1
        vaddsd  xmm1, xmm1, xmm4
# v0.c:55:      sum += array1[i] * array2[i];
        vaddsd  xmm1, xmm0, xmm1
        vunpckhpd       xmm0, xmm0, xmm0
        vaddsd  xmm4, xmm1, xmm0
        cmp     rax, r12
        jne     .L8     #,
```

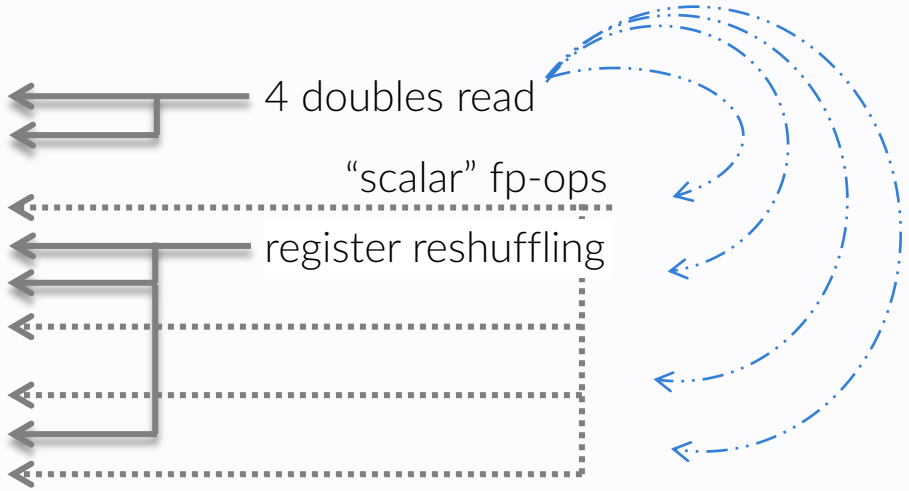
# turning on the optimization…

Let's have a preview of how the generated code changes..

v0

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

```
.L8:
# v0.c:55:      sum += array1[i] * array2[i];
      vmovupd ymm5, YMMWORD PTR 0[r13+rax]
      vmulpd  ymm0, ymm5, YMMWORD PTR [r15+rax]
      add     rax, 32 # ivtmp.18,
      vaddsd  xmm4, xmm0, xmm4
      vunpckhpd       xmm1, xmm0, xmm0
      vextractf128    xmm0, ymm0, 0x1
      vaddsd  xmm1, xmm1, xmm4
# v0.c:55:      sum += array1[i] * array2[i];
      vaddsd  xmm1, xmm0, xmm1
      vunpckhpd       xmm0, xmm0, xmm0
      vaddsd  xmm4, xmm1, xmm0
      cmp     rax, r12
      jne     .L8     #,
```




*gcc 8.2 on SkyLake*
*-O3 -march=native*

HINT:
data are fetched with vector instructions, then processed in a mixed way

# turning on the optimization…

## Short comment on the previous slide

Looking at the assembler generated by the compiler is often useful to understand wher to direct optimization efforts.

In the previous slide, the compiler (with `-O3 -march=native` ) chooses to fetch data from memory with a vector instruction, the loading 4 doubles at a time (note that it uses the instruction for *unaligned memory*[*]: `vmov`**`u`**`pd`).
But however can not really exploit the ILP (Instruction-Level parallelism) due to the data dependency intrinsic in how we wrote the loop.

That's why the best first step is to exhibit the possible parallelism, for instance either by unrolling or by a different accumulation (v1 and v3 in the following slides).

[*]you should remember what alignment is from the lecture about memory allocation

# v* profile – 00

```
time is :0.217358 (min 0.216092, std_dev 0.000667496)
transfer rate was   3.43 GB/sec (  22.95% of theoretical max that is    15 GB/sec)
       503,469,659      cycles:.                                        ( +- 12.99% )
       185,027,735      instructions:.           #    0.37  insn per cycle  ( +- 17.44% )
```

v0

```
time is :0.139794 (min 0.139124, std_dev 0.000733397)
transfer rate was   5.33 GB/sec (  35.68% of theoretical max that is    15 GB/sec)
       394,225,210      cycles:.                                        ( +-  9.57% )
       204,046,712      instructions:.           #    0.52  insn per cycle  ( +- 27.62% )
```

v1

```
time is :0.119491 (min 0.117614, std_dev 0.00116417)
transfer rate was   6.24 GB/sec (  41.75% of theoretical max that is    15 GB/sec)
     1,183,361,839      cycles:.                                        ( +-  6.59% )
       824,904,000      instructions:.           #    0.70  insn per cycle  ( +-  8.66% )
```

v3

```
time is :0.0805418 (min 0.0798115, std_dev 0.00128884)
transfer rate was   9.25 GB/sec (  61.93% of theoretical max that is    15 GB/sec)
       288,211,800      cycles:.                                        ( +-  1.80% )
        92,540,296      instructions:.           #    0.32  insn per cycle  ( +-  3.80% )
```

v6

# v* profile – 03

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v0_papi_O3n 50000000
generating 100000000 numbers..done
sum is 1.24993e+07
time is :0.104058 (min 0.103508, std_dev 0.000375562, all 1.04728)
transfer rate was   7.16 GB/sec (  47.94% of theoretical max that is    15 GB/sec)
        IPC: 0.65
        [ time: 0.1041sec  -  ins: 1.50004e+08 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v1_papi_O3n 50000000
generating 100000000 numbers..done (  1.27)
pipeline demonstrator, step 1:
- unroll 2 times

sum is 1.24993e+07
time is :0.0755591 (min 0.0750261, std_dev 0.000291917, all 0.762576)
transfer rate was   9.86 GB/sec (  66.02% of theoretical max that is    15 GB/sec)
        IPC:  1.2
        [ time: 0.07556sec  -  ins: 1.93753e+08 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v3_papi_O3n 50000000
generating 100000000 numbers..done (in   1.08sec)
pipeline demonstrator, step 3:

sum is 1.24993e+07
time is :0.0545231 (min 0.0544507, std_dev 8.46002e-05, all 0.545244)
transfer rate was   13.7 GB/sec (  91.49% of theoretical max that is    15 GB/sec)
        IPC:  1.1
        [ time: 0.0545sec  -  ins:   1.6e+08 ]%
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v3c_papi_O3n 50000000
generating 100000000 numbers..done (in   1.27sec)
pipeline demonstrator, step 3:
- unroll 2 times +
- separate mul and sum
- separate accumulations

sum is 1.24993e+07
time is :0.0527944 (min 0.0526688, std_dev 0.000107074, all 0.53314)
transfer rate was   14.1 GB/sec (  94.49% of theoretical max that is    15 GB/sec)
        IPC:  1.2
        [ time: 0.0528sec  -  ins:  1.75e+08 ]
```

# v* profile – O3 + vectorisation

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v6_papi_O3n 50000000
generating 100000000 numbers..done (in   1.27sec)
sum is 1.24993e+07
time is: 0.0647406 (min 0.0642672, std_dev 0.00017916, all 0.654759)
transfer rate was   11.5 GB/sec (  77.05% of theoretical max that is    15 GB/sec)
        IPC:  0.8
        [ time: 0.06475sec  -  ins: 1.00003e+08 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v6_intrinsics_papi_O3n 50000000
generating 100000000 numbers..done (in    1.1sec
sum is 1.24993e+07
time is: 0.0504149 (min 0.0468492, std_dev 0.00132376, all 0.509429)
transfer rate was   14.8 GB/sec (  98.95% of theoretical max that is    15 GB/sec)
        IPC: 0.67
        [ time: 0.05042sec  -  ins: 7.50003e+07 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v8_papi_O3n 50000000
generating 100000000 numbers..done (in   1.11sec)
sum is 1.24993e+07
time is: 0.0499751 (min 0.0466076, std_dev 0.00124928, all 0.499767)
transfer rate was   14.9 GB/sec (  99.82% of theoretical max that is    15 GB/sec)
        IPC: 0.56
        [ time: 0.04998sec  -  ins: 6.25002e+07 ]
```

There's a lot of improvement in such a simple kernel, but why IPC even decreased? Hint: the code becomes memory-bound (look at the memory transfer rate).

# comparison: v0 – v1

v0

```
for ( int i = 0; i < N; i++ )
  sum += array1[i] * array2[i];
```

v1

```
for ( int i = 0; i < N-1; i+=2 )
  // simply unrolling 2 times, exposes the fact that at least
  // 2 elements of the array can be processed independently
  {
    sum1 += array1[   i ] * array2[   i ];
    sum2 += array1[ i+1 ] * array2[ i+1 ];
  }
if ( N % 2 )
  sum = array1[ N-1 ] * array2[ N-1 ];
```

# comparison: v0 – v1 [-O0]

**v0**

```
.L8:
# v0.c:185:      sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163, sum
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp162
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:        for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164, i
        cmp     eax, DWORD PTR -148[rbp]        # tmp164, N
        jl      .L8
```

**v1**

```
.L10:
# v1_aligned.c:91:        sum1 += array1[   i ] * array2[   i ];
        movl    -148(%rbp), %eax        # i, tmp186
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -96(%rbp), %rax # array1, tmp187
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm1
        movl    -148(%rbp), %eax        # i, tmp188
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -88(%rbp), %rax # array2, tmp189
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm0
        vmulsd  %xmm0, %xmm1, %xmm0
        vmovsd  -112(%rbp), %xmm1       # sum1, tmp191
        vaddsd  %xmm0, %xmm1, %xmm0
        vmovsd  %xmm0, -112(%rbp)       # tmp190, sum1
        movl    -148(%rbp), %eax        # i, tmp192
        cltq
        incq    %rax
        leaq    0(,%rax,8), %rdx
        movq    -96(%rbp), %rax # array1, tmp193
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm1
        movl    -148(%rbp), %eax        # i, tmp194
        cltq
        incq    %rax
        leaq    0(,%rax,8), %rdx
        movq    -88(%rbp), %rax # array2, tmp195
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm0
        vmulsd  %xmm0, %xmm1, %xmm0
        vmovsd  -104(%rbp), %xmm1       # sum2, tmp197
        vaddsd  %xmm0, %xmm1, %xmm0
        vmovsd  %xmm0, -104(%rbp)
        addl    $2, -148(%rbp)  #, i
.L9:
# v1_aligned.c:87:        for ( int i = 0; i < N-1; i+=2 )
        movl    -164(%rbp), %eax        # N, tmp198
        decl    %eax    # _42
        cmpl    %eax, -148(%rbp)        # i
        jl      .L10
```

# comparison: v0 – v1 [-O0]

**v0**

```
.L8:
# v0.c:185:      sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163, sum
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp162
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:        for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164, i
        cmp     eax, DWORD PTR -148[rbp]        # tmp164, N
        jl      .L8
```

**v1**

```
.L10:
# v1_aligned.c:91:        sum1 += array1[   i ] * array2[   i ];
        movl    -148(%rbp), %eax        # i, tmp186
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -96(%rbp), %rax # array1, tmp187
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm1
        movl    -148(%rbp), %eax        # i, tmp188
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -88(%rbp), %rax # array2, tmp189
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm0
        vmulsd  %xmm0, %xmm1, %xmm0
        vmovsd  -112(%rbp), %xmm1       # sum1, tmp191
        vaddsd  %xmm0, %xmm1, %xmm0
        vmovsd  %xmm0, -112(%rbp)       # tmp190, sum1
        movl    -148(%rbp), %eax        # i, tmp192
        cltq
        incq    %rax
        leaq    0(,%rax,8), %rdx
        movq    -96(%rbp), %rax # array1, tmp193
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm1
        movl    -148(%rbp), %eax        # i, tmp194
        cltq
        incq    %rax
        leaq    0(,%rax,8), %rdx
        movq    -88(%rbp), %rax # array2, tmp195
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm0
        vmulsd  %xmm0, %xmm1, %xmm0
        vmovsd  -104(%rbp), %xmm1       # sum2, tmp197
        vaddsd  %xmm0, %xmm1, %xmm0
        vmovsd  %xmm0, -104(%rbp)
        addl    $2, -148(%rbp)  #, i
.L9:
# v1_aligned.c:87:        for ( int i = 0; i < N-1; i+=2 )
        movl    -164(%rbp), %eax        # N, tmp198
        decl    %eax    # _42
        cmpl    %eax, -148(%rbp)        # i
        jl      .L10
```

The `v1` version looks twice as longer (i.e.: more instructions to be executed), actually it really looks like the same code was copied & pasted below the original segment.

How does it come that the result is a faster execution?

# comparison: v0 – v1 [-O0]

It is clearer considering the output of the PAPI instrumented code, that returns the IPC.
Clearly, the 2fold unroll makes the CPU able to better exploit more than 1 logical units at a time.

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v0_papi_O0 50000000
generating 100000000 numbers..done
sum is 1.24993e+07
time is :0.191208 (min 0.190616, std_dev 0.00053432, all 1.91787)
transfer rate was    3.9 GB/sec (  26.09% of theoretical max that is    15 GB/sec)
        IPC:  1.9
        [ time: 0.1912sec  -  ins: 1.00158e+09 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v1_papi_O0 50000000
generating 100000000 numbers..done ( 0.965)
pipeline demonstrator, step 1:
- unroll 2 times

sum is 1.24993e+07
time is :0.125749 (min 0.12396, std_dev 0.00142263, all 1.26349)
transfer rate was   5.92 GB/sec (  39.67% of theoretical max that is    15 GB/sec)
        IPC:  2.8
        [ time: 0.1258sec  -  ins: 9.76578e+08 ]
```

# refining v1

v1

```
for ( int i = 0; i < N-1; i+=2 )
  // simply unrolling 2 times, exposes the fact that at least
  // 2 elements of the array can be processed independently
  {
    sum1 += array1[   i ] * array2[   i ];
    sum2 += array1[ i+1 ] * array2[ i+1 ];
  }
if ( N % 2 )
  sum = array1[ N-1 ] * array2[ N-1 ];
```

v1b

```
      double register sum0 = 0;
      double register sum1 = 0;

      double register * restrict a1 = __builtin_assume_aligned(array1, 32);
      double register * restrict a2 = __builtin_assume_aligned(array2, 32);

      tstart = CPU_TIME;
#pragma ivdep
      for( int i = 0 ; i < N-1; i+=2 )
        {
          sum0 += *(a1++) * *(a2++);
          sum1 += *(a1++) * *(a2++);
        }
      if( N%2 )
        sum = *a1 * *a2;
```

# refining v1

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v1_papi_O0 50000000
generating 100000000 numbers..done ( 0.973)
pipeline demonstrator, step 1:
- unroll 2 times

sum is 1.24993e+07
time is :0.125593 (min 0.125287, std_dev 0.000476174, all 1.26231)
transfer rate was   5.93 GB/sec (  39.72% of theoretical max that is    15 GB/sec)
        IPC:  2.8
        [ time: 0.1256sec  -  ins: 9.76578e+08 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v1b_papi_O0 50000000
generating 100000000 numbers.. done (in  0.973sec)
pipeline demonstrator, step 1b:
- unroll 2 times +
- align memory   +
- minimize ptr arithmetic

sum is 1.24993e+07
time is :0.0977905 (min 0.0972945, std_dev 0.000712555, all 0.977921)
transfer rate was   7.62 GB/sec (  51.01% of theoretical max that is    15 GB/sec)
        IPC:  2.1
        [ time: 0.0978sec  -  ins: 5.76578e+08 ]
```

# refining: v3

Let's go further in trying to make it clear for the CPU about the possible ILP

v3

```
#pragma ivdep
      for ( int i = 0; i < N_4; i+=4 )
        {
          sum += array1[i] * array2[i] +
            array1[i+1] * array2[i+1] +
            array1[i+2] * array2[i+2] +
            array1[i+3] * array2[i+3];
        }

      for ( int i = N_4; i < N; i++ )
        sum += array1[i] * array2[i];
```

prefetching

v3b

```
      double register a;
      double register b;
      a = array1[0] * array2[0];
#pragma ivdep
      for ( int i = 0; i < N_4; i+=4 )
        {
          DO_NOT_OPTIMIZE;
          b = array1[i+4] * array2[i+4];
          DO_NOT_OPTIMIZE;
          sum += a +
            array1[i+1] * array2[i+1] +
            array1[i+2] * array2[i+2] +
            array1[i+3] * array2[i+3];

          a = b;
        }

      for ( int i = N_4; i < N; i++ )
        sum += array1[i] * array2[i];
```

# refining: vectorization

```
typedef double v4df __attribute__ ((vector_size (4*sizeof(double))));
typedef union {
  v4df   V;
  double v[4];
}v4df_u;
```

```
#ifdef _GNU_SOURCE
      v4df sum_ = {0, 0, 0, 0};
#else
      v4df sum_ = {0};
#endif
      v4df register mytmp;
      v4df register tmp  = *((v4df*)&array1[0]) * *((v4df*)&array2[0]);

      int N__4 = N/4;
      int N_4 = N__4*4;
      tstart = CPU_TIME;
#pragma ivdep
      for( int i = 1; i <= N__4; i++)
        {
          _DO_NOT_OPTIMIZE_BEGIN;
          mytmp = *((v4df*)array1 + i) * *((v4df*)array2 + i);
          _DO_NOT_OPTIMIZE_END;
          sum_ += tmp;
          tmp   = mytmp;
        }

      for ( int i = N_4; i < N; i++ )
        sum += array1[ i ] * array2[ i ];
```

# refining: vectorization with intrinsics

```
#elif defined ( __AVX__ ) || defined ( __AVX2__ )

#define V_DSIZE 4
typedef __m256d vd;
vd _dzero_ = {0, 0, 0, 0};
#define MUL_ADD( A1, A2, R ) _mm256_fmadd_pd( (A1), (A2), (R) )
```

```
      vd sum_ __attribute__ ((aligned(64)));
      sum_ = _dzero_;

      vd * restrict A1 __attribute__ ((aligned(64)));
      vd * restrict A2 __attribute__ ((aligned(64)));
      A1  = (vd*)array1 ;
      A2  = (vd*)array2 ;

      int N__ = N / V_DSIZE;
      int N_  = N__ * V_DSIZE;

      tstart = CPU_TIME;

#pragma ivdep
      for( int i = 0; i < N__; i++)
        sum_ = MUL_ADD( A1[i], A2[i], sum_ );

      for ( int i = N_; i < N; i++ )
        sum += array1[ i ] * array2[ i ];
```

# Pipeline: hands-on

## Codes

You find the code snippets presented here 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 Ulisse facility.

# Outline

- Memory optimizations
- Loops optimizations
- Other optimizations
  - > prefetching
  - > macros as templates
  - > SIMD instructions
  - > access locality in NUMA systems

# Prefetching

# Prefetching

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).

# Prefetching

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

# Prefetching

## Explicit prefetching: a simple example

```
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;
 }
```

# Prefetching

## Explicit prefetching: a simple example

```
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;
 }
```

# Prefetching

## Explicit prefetching: a simple example

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

```
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
```

```
     high = mid-1;
   else
     return mid;
 }

 return -1;
}
```

# Prefetching

Explicit prefetching: a simple example

```
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
```

```
__builtin_prefetch (&data[(mid + 1 + high)/2], 0, 3);
__builtin_prefetch (&data[(low + mid - 1)/2], 0, 3);
```

```
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
```

```
}
```

# 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.

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

Prefetching by preloading

```
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]; // cache-hit
      elem c = elem[i+2]; // cache-hit
      elem d = elem[i+3]; // cache-hit
      Elaborate(a);
      Elaborate(b);
      Elaborate(c);
      Elaborate(d);
      a = e;
 }
```

# Prefetching: hands-on

A basic example for practicing with **preloading**

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

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 Ulisse facility.

# Macros as templates

# MACROS as templates

Standerd **`libc`** has a lot of very good well optimized routines for many general purposes.

However, many routines are meant to work on a general structure without any precise knowledge of its internals. Typically you call a **`libc`** routine like that;

```
libc_routine( void* base, size_t size, size_t n,
              int (*)dealer(void *, …) );
```

However, in this way the compiler can not switch on several optimizations because the data structure is opaque.

# MACROS as templates

A possibile cure for this is (in C) to write a MACRO that behaves like a template.

For instance, if you put the following code in a .h file

```
int MY_WONDER(A, B, C, …) { \

      MY_DATA_TYPE doing something
      ..some stuff here..
      DEALER(A, B, C, …)
      ..some other stuff.. }
```

# MACROS as templates

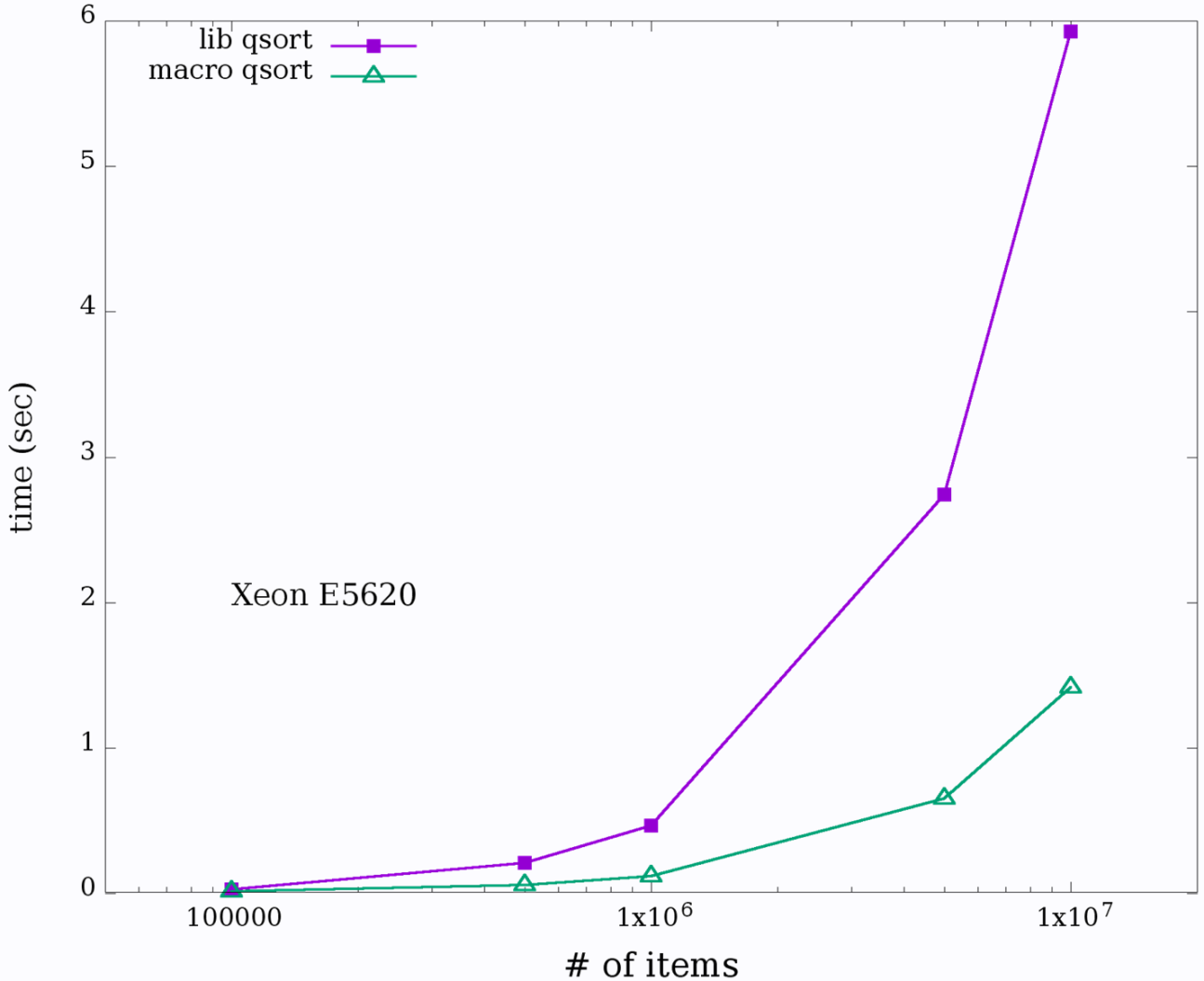
And you define MY_DATA_TYPE and DEALER  and whatever else you need just before including the .h file

```
#define MY_DATA_TYPE some_type

#define DEALER(A, B, C, …) { \
      ..some other stuff.. }

#include “my_routines.h”
```

You get something that resembles a template that the compiler can optimize.

# MACROS as templates

Going practical, let's try a quicksort (live):



# Access locality in NUMA sys

# Access locality in ccNUMA systems

Let's start from a code snippet of the classic triad benchmark:

```
// initialization of arrays
B[i] = … ; C[i] = …; D[i] = …;

#pragma omp parallel for
for (cc = 0; cc < SIZE; cc++)
   A[i] = B[i] + C[i] × D[i];
```

The initialization happens in a non-parallel region, while the loop where threads actually access data is in the **omp** region.

That means that **data will be placed in the memory/cache of the LD0** (first-touch policy) and threads from other LDs will all have to queue and wait – very likely determining severe cache inefficiencies – to access the memory on LD0.

# Access locality in ccNUMA systems

A good policy is then to parallelize the data initialization section as well:

```
#pragma omp parallel
B[i] = … ; C[i] = …; D[i] = …;

#pragma omp parallel for
for (cc = 0; cc < SIZE; cc++)
   A[i] = B[i] + C[i] × D[i];
```

or, more in general, to initialize/read the data first-touching them in the same way they will be accessed.

[ tech note: all that is true if the array is large enough so that each thread chunk is bigger than a page size. For smaller data set, consider to make them thread-private, creating local copies for each thread ]

# The winding road towards optimization