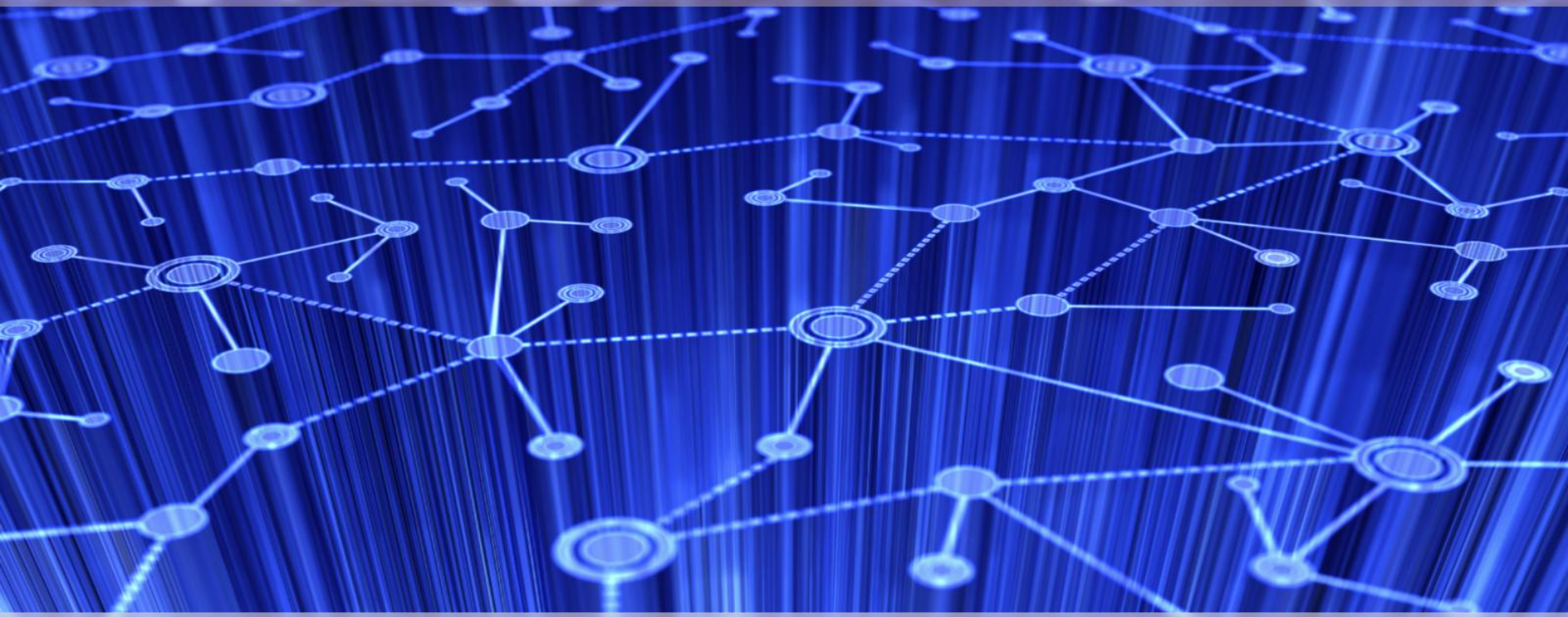


Foundations of High Performance Computing

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.

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

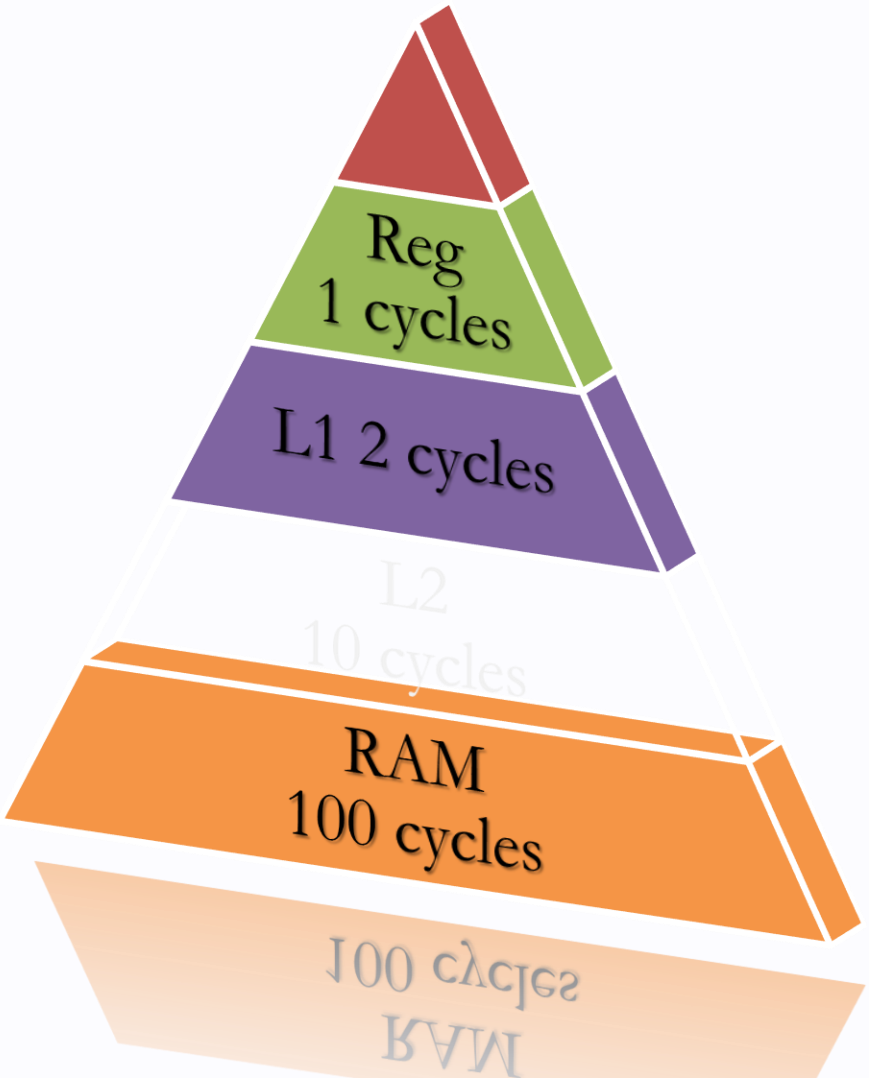
- **Memory** optimizations
  - > the importance of memory access
  - > the importance of the cache
  - > cache access optimization in loops
  - > cache-oblivious algorithms
  - > cache-oblivious data structures
- **Loops** optimizations
- Other optimizations

The memory access

Again on cache

The importance of memory access

Impact of cache misses on average memory access time



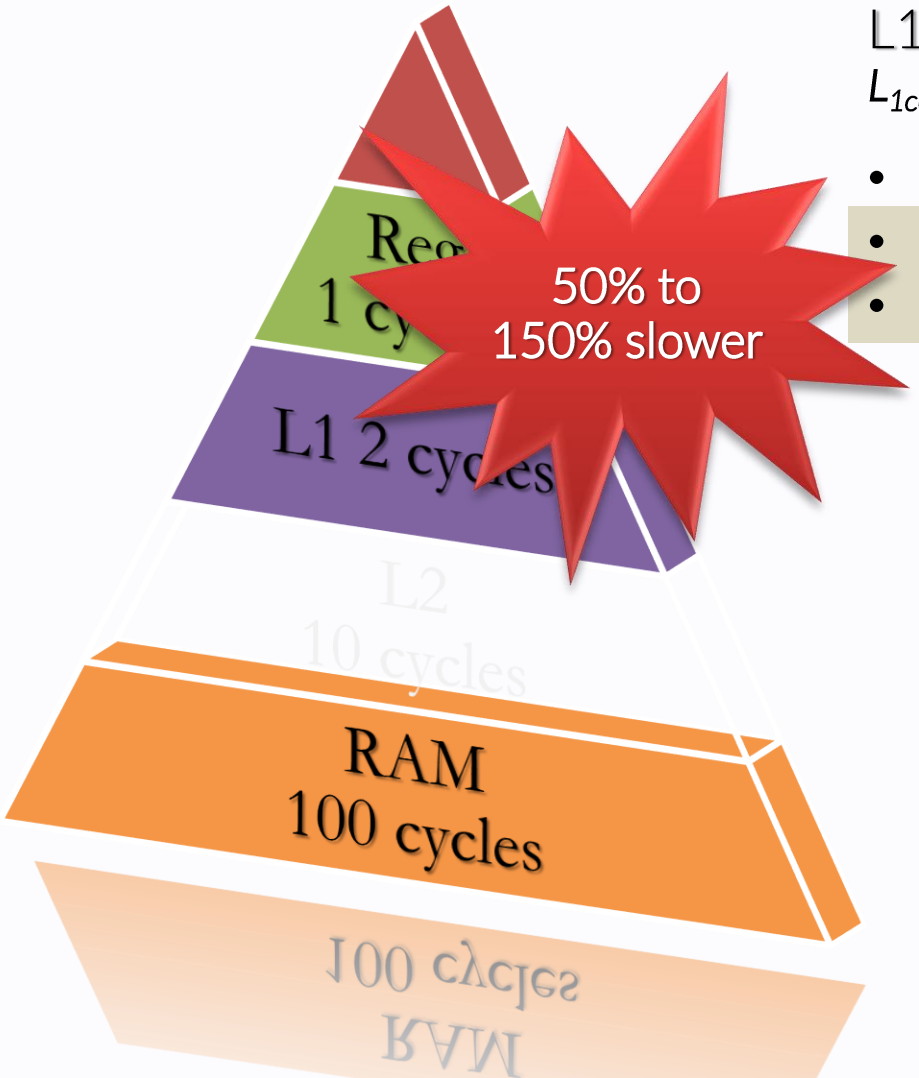
L1 cache + RAM

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

- 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



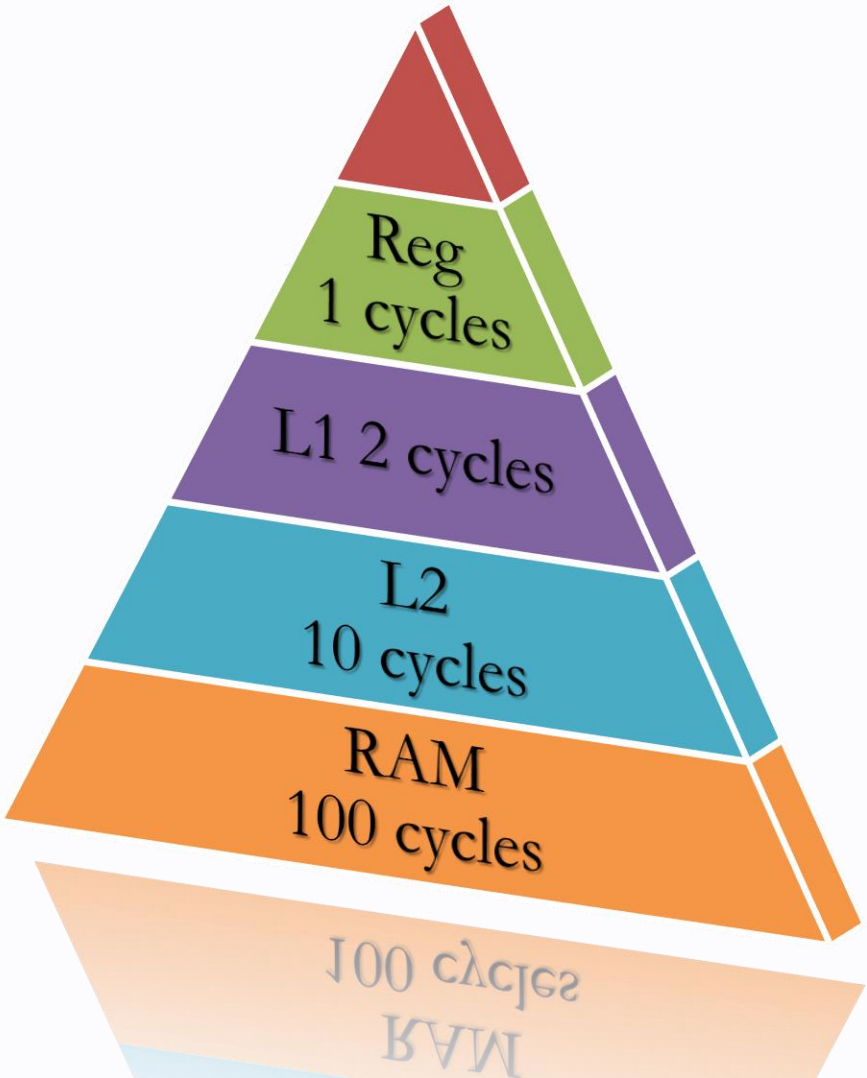
L1 cache + RAM

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

- 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

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

- 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?

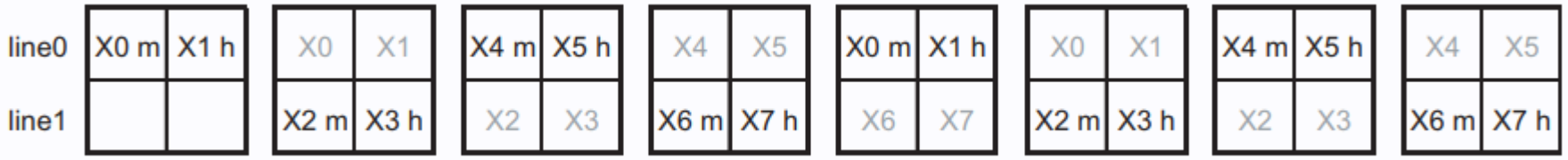
The importance of cache

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]);
```

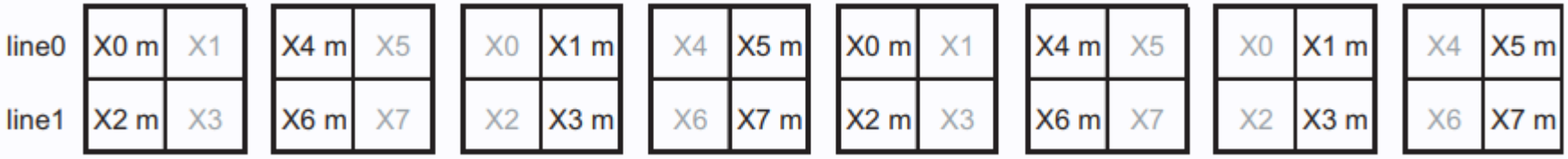


The hit-miss pattern is : MH MH MH MH MH MH MH MH, that is miss-rate is 50%

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]);
   }
```

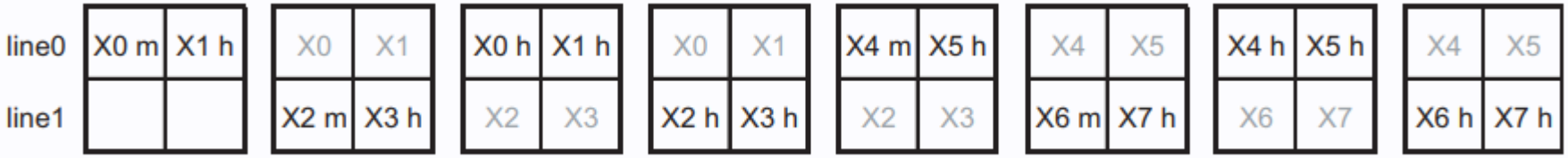


The hit-miss pattern now is : MM MM MM MM MM MM MM MM, that is miss-rate is 100%

The importance of cache

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

```
for(int i = 0; i < 2; i++)
   for(k = 0; k < 2; k++)
      for(int j = 0; j < 7; j += 2)
         access(X[ j + i×4 ]);
```

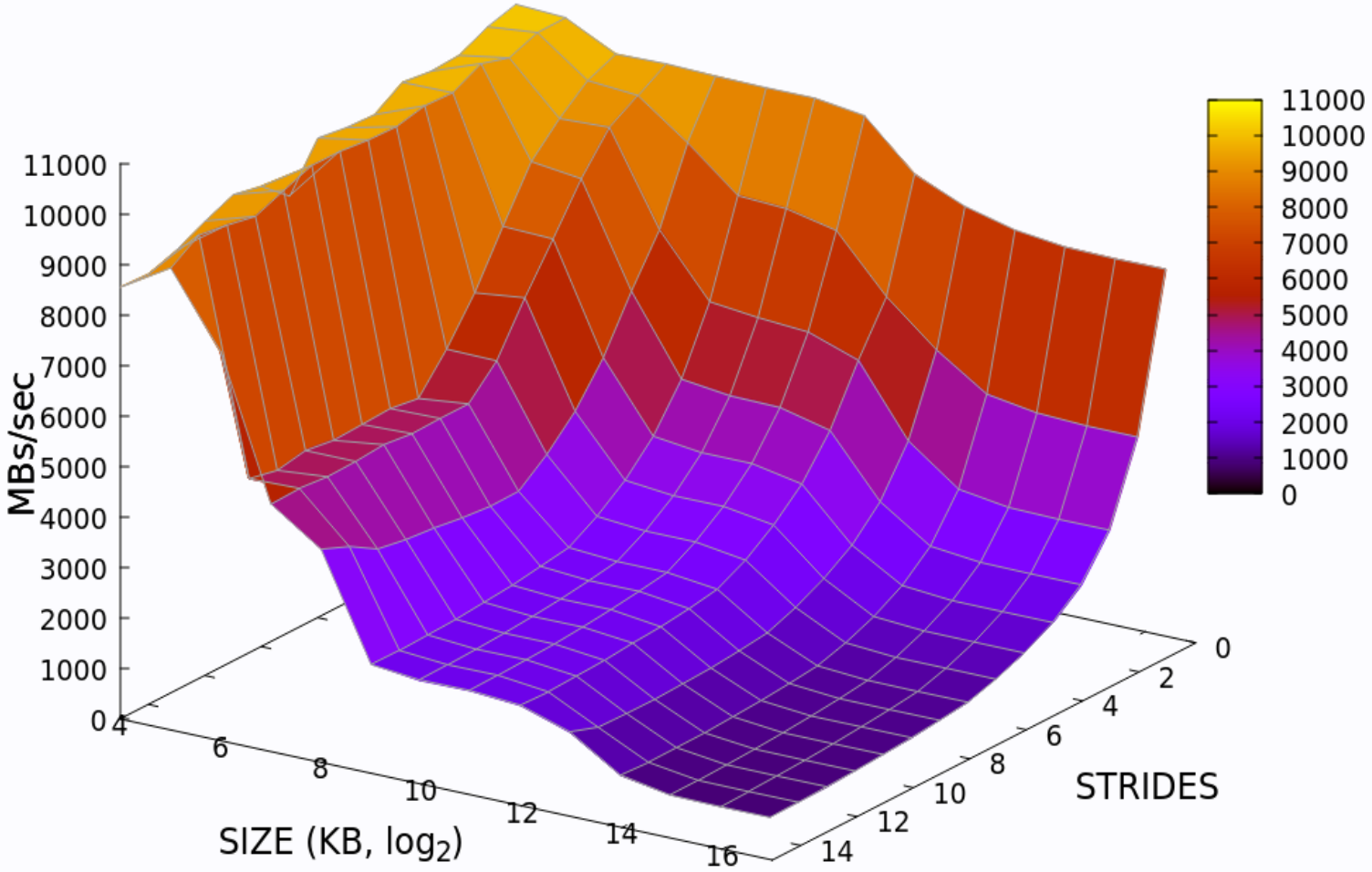


The hit-miss pattern now is : MH MH HH HH MH MH HH HH, that is miss-rate is 25%

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

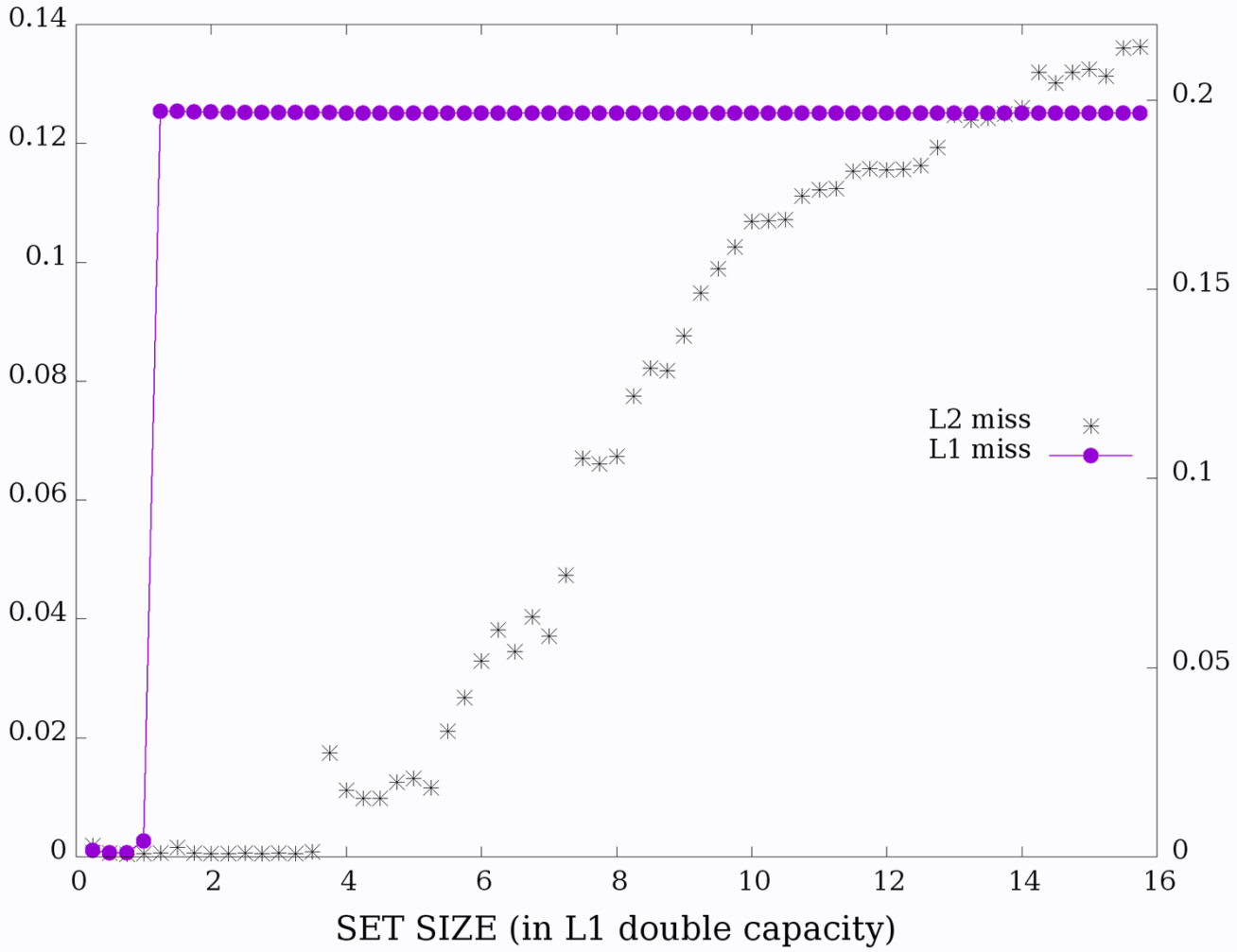
The importance of cache

The result is.. the memory mountain. You already know it ☺



The importance of memory access pattern

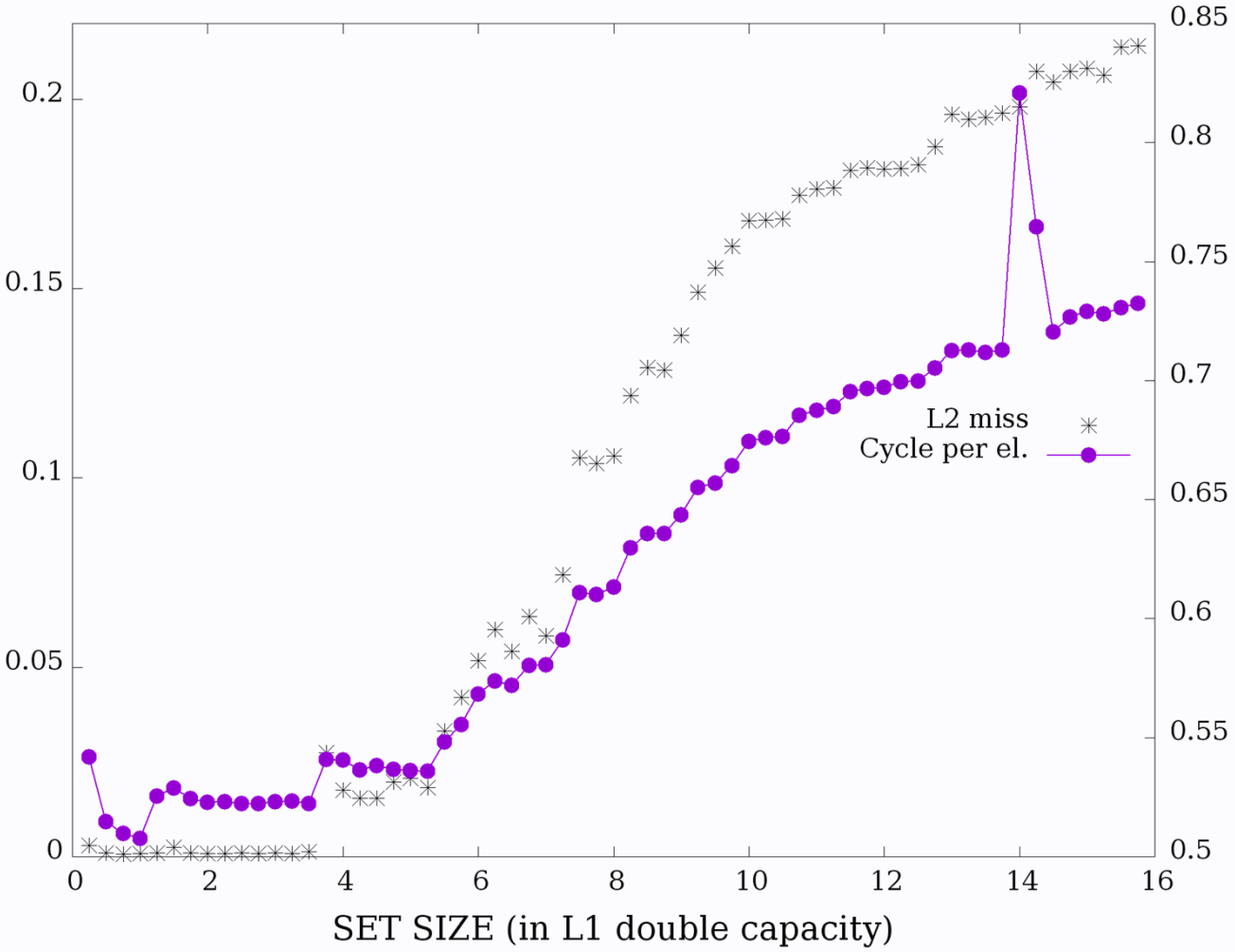
Let's find our L1 – and L2– cache sizes




```
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:

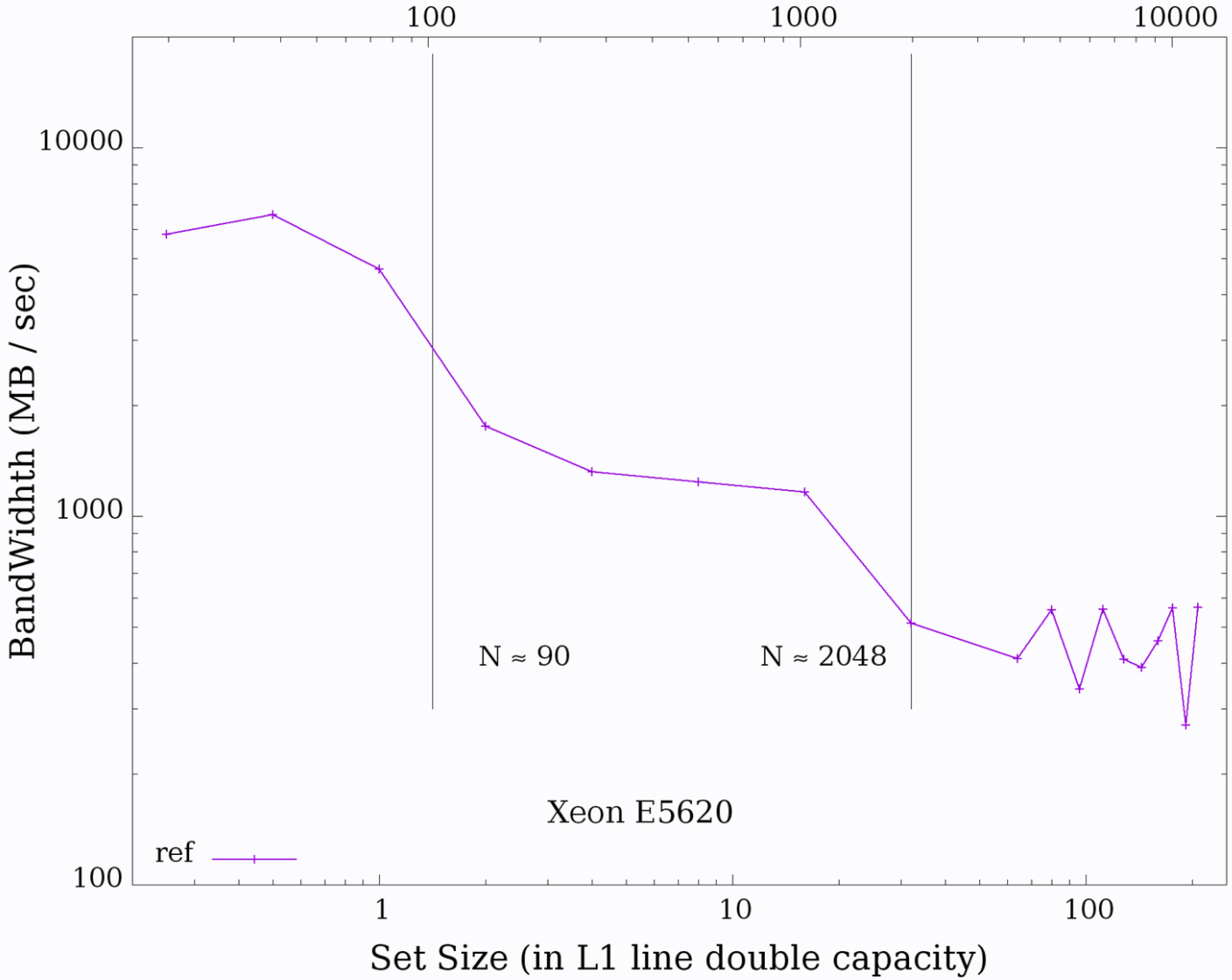
1. $2 \times N^2 < C$
   both matrices can fit in the cache, traversal order and locality does not impact on performance (bandwidth ~ maximum)

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.

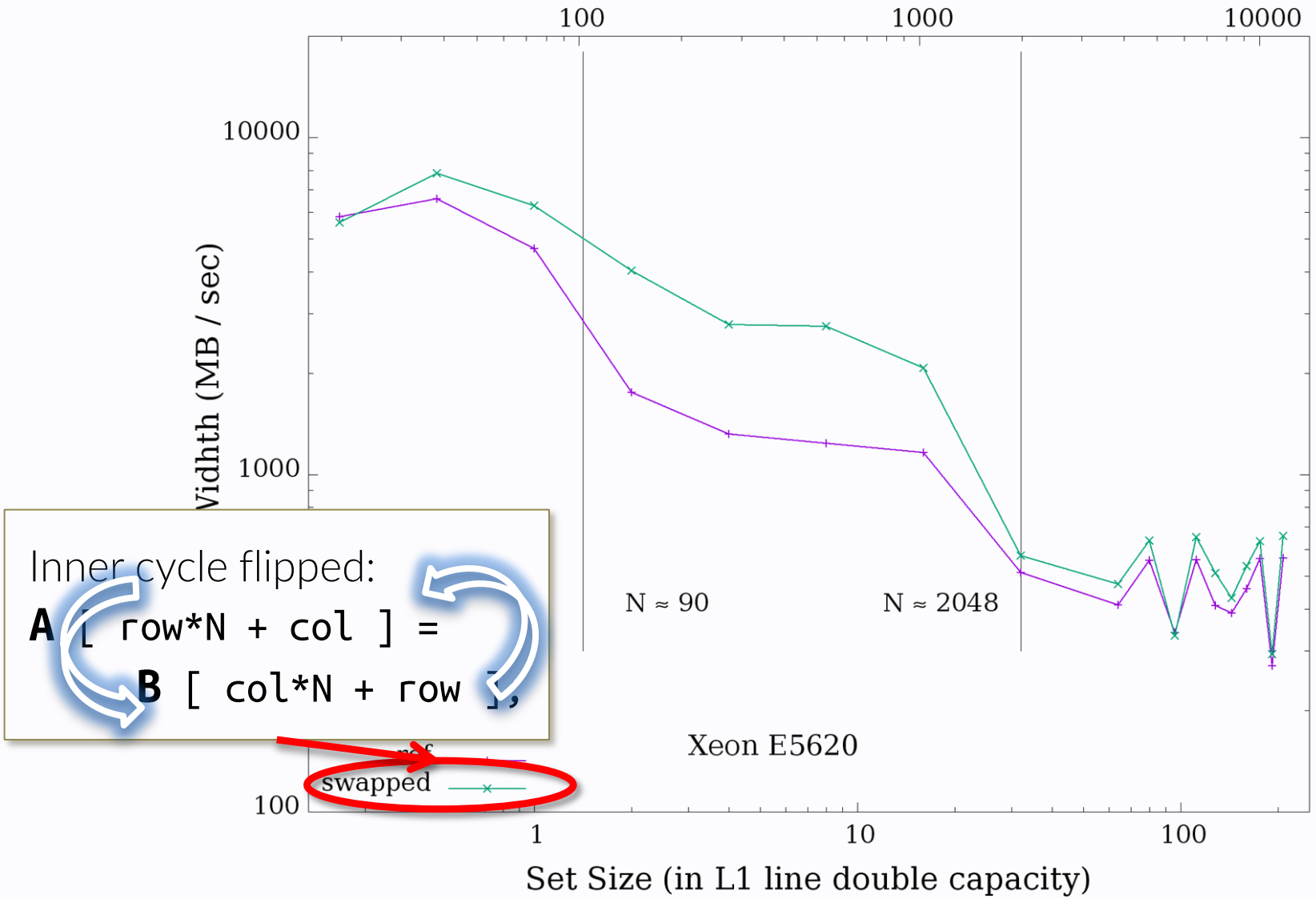
The importance of memory access pattern

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



The importance of memory access pattern

Matrix transpose: avoiding strided access on write

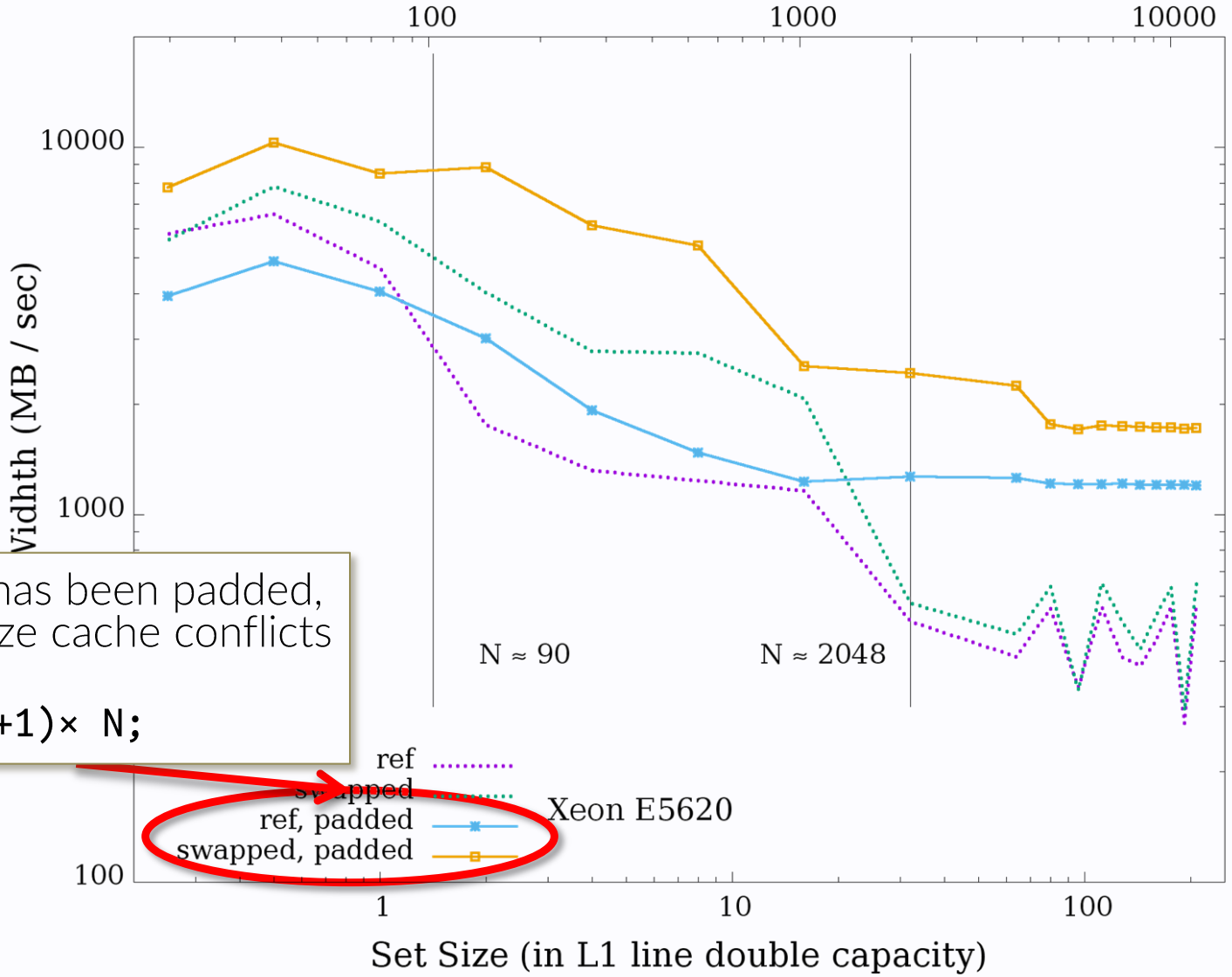


Inner cycle flipped:

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

The importance of memory access pattern

Matrix transpose: avoiding cache associativity conflicts

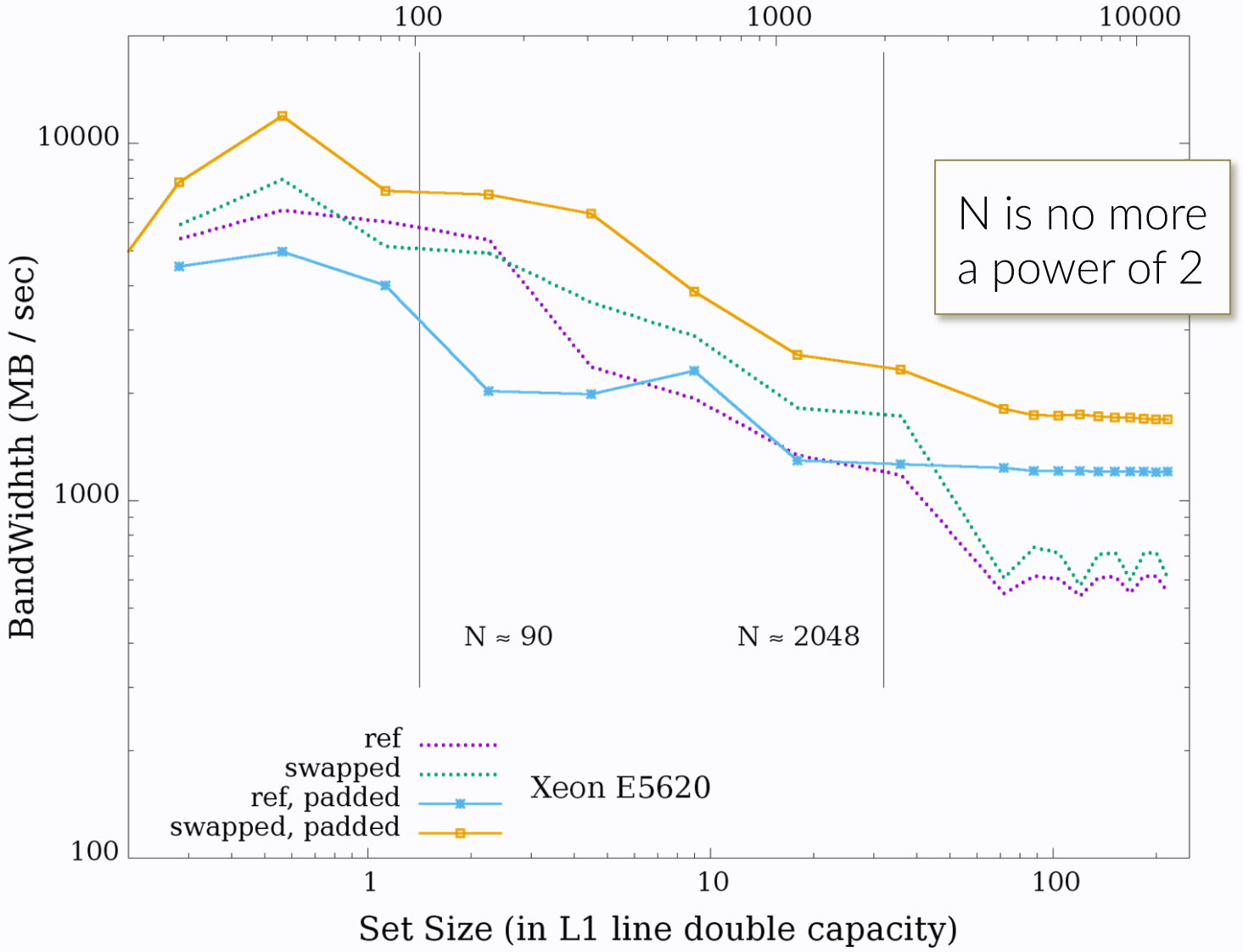


B layout has been padded, to minimize cache conflicts with A

```
B = (N+1)× N;
```

The importance of memory access pattern

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

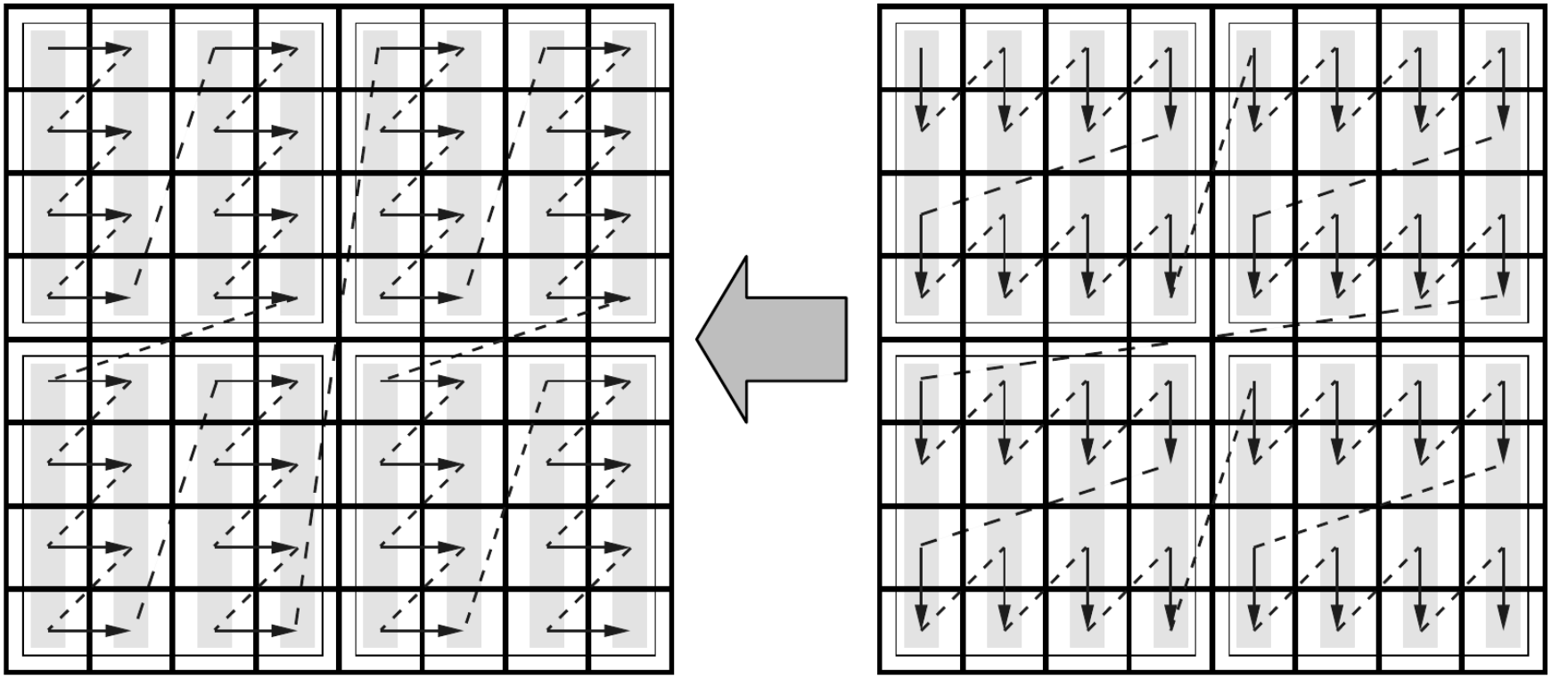
Sometimes no magic wand can cure the fact that you have to access N^2 memory locations.
For instance: in matrix transpose you have to access all the source matric and all the destination matrix once.

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.**

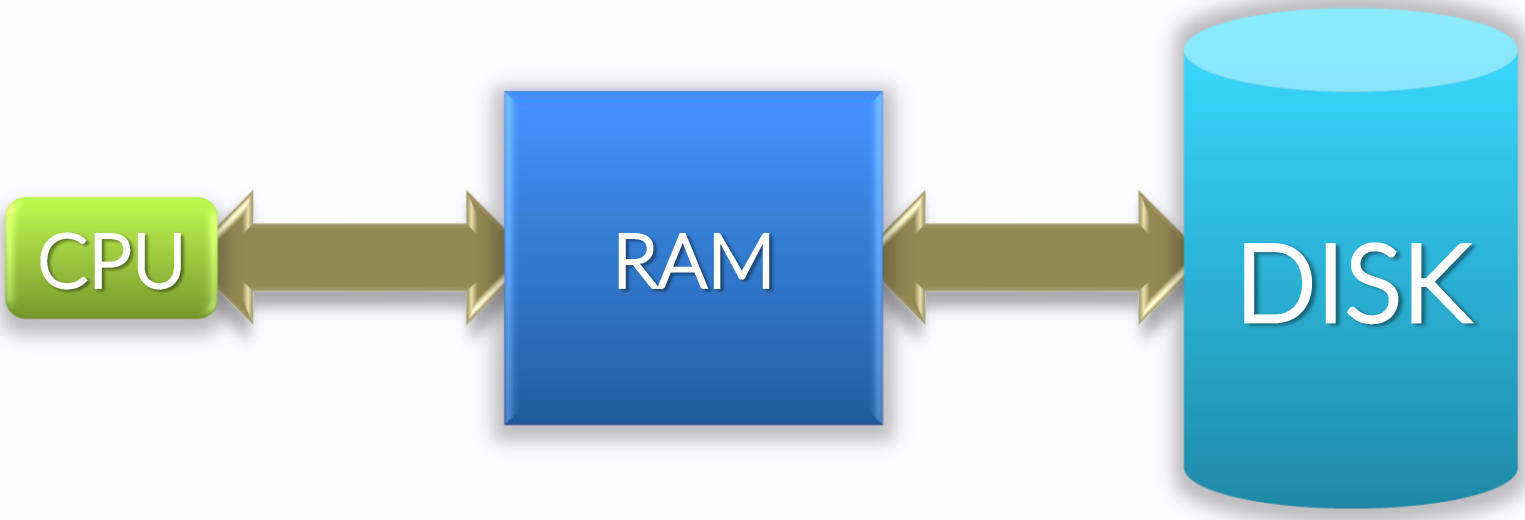
Cache-consciousness

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.

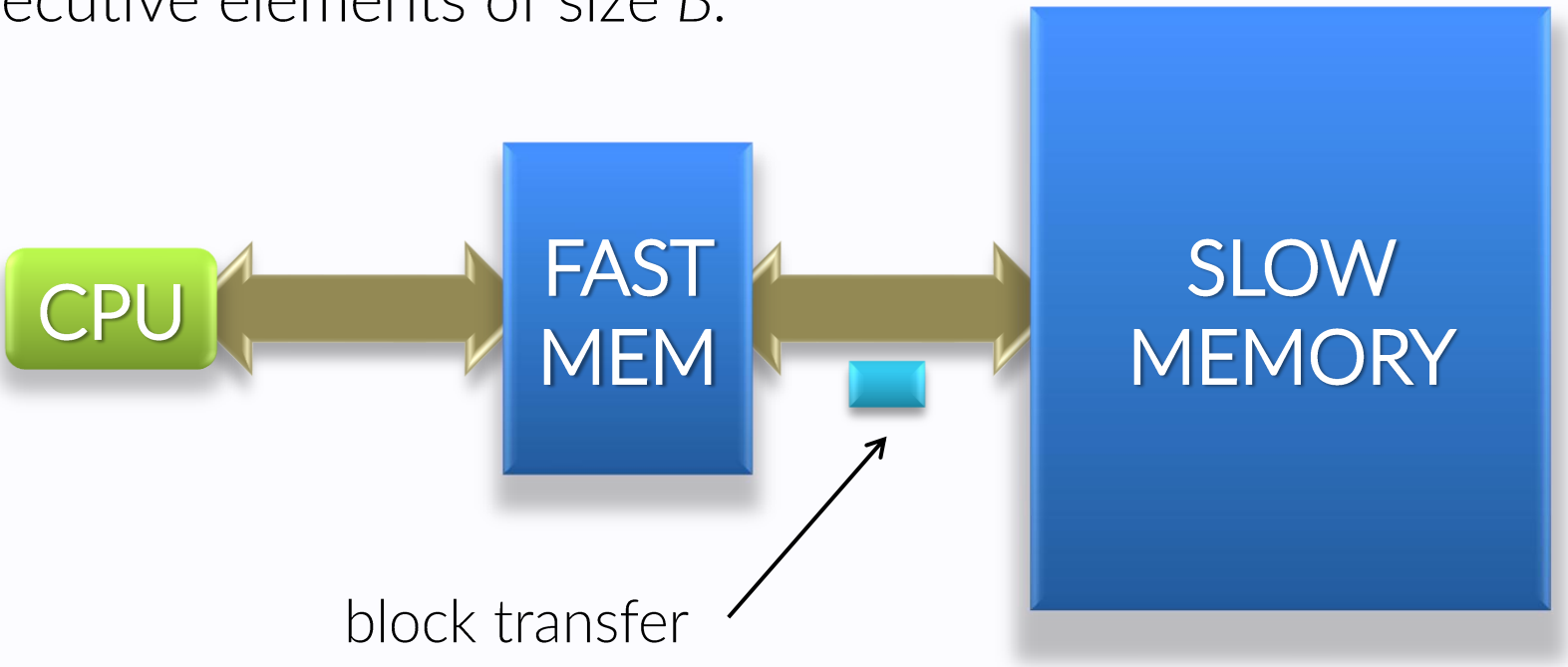
The **RAM MODEL** was more than enough to design and analyze algorithms, based on complexity measured as the *instructions* – the work *W(n)* – needed for an input problem of a given size *n*.



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.

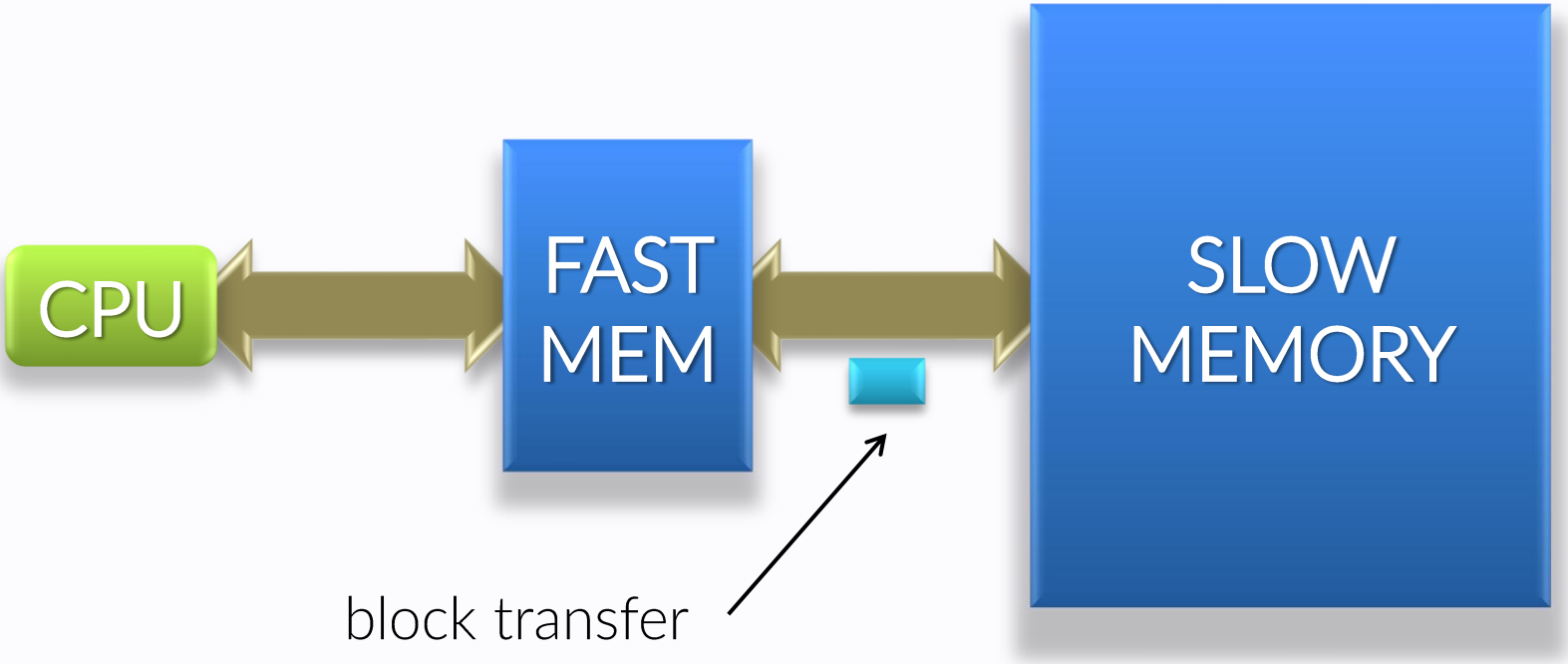
The hierarchy is assumed to consist in a fast memory of size *M* that exchange data with a slower infinite memory in bunches of consecutive elements of size *B*.



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];
}
```

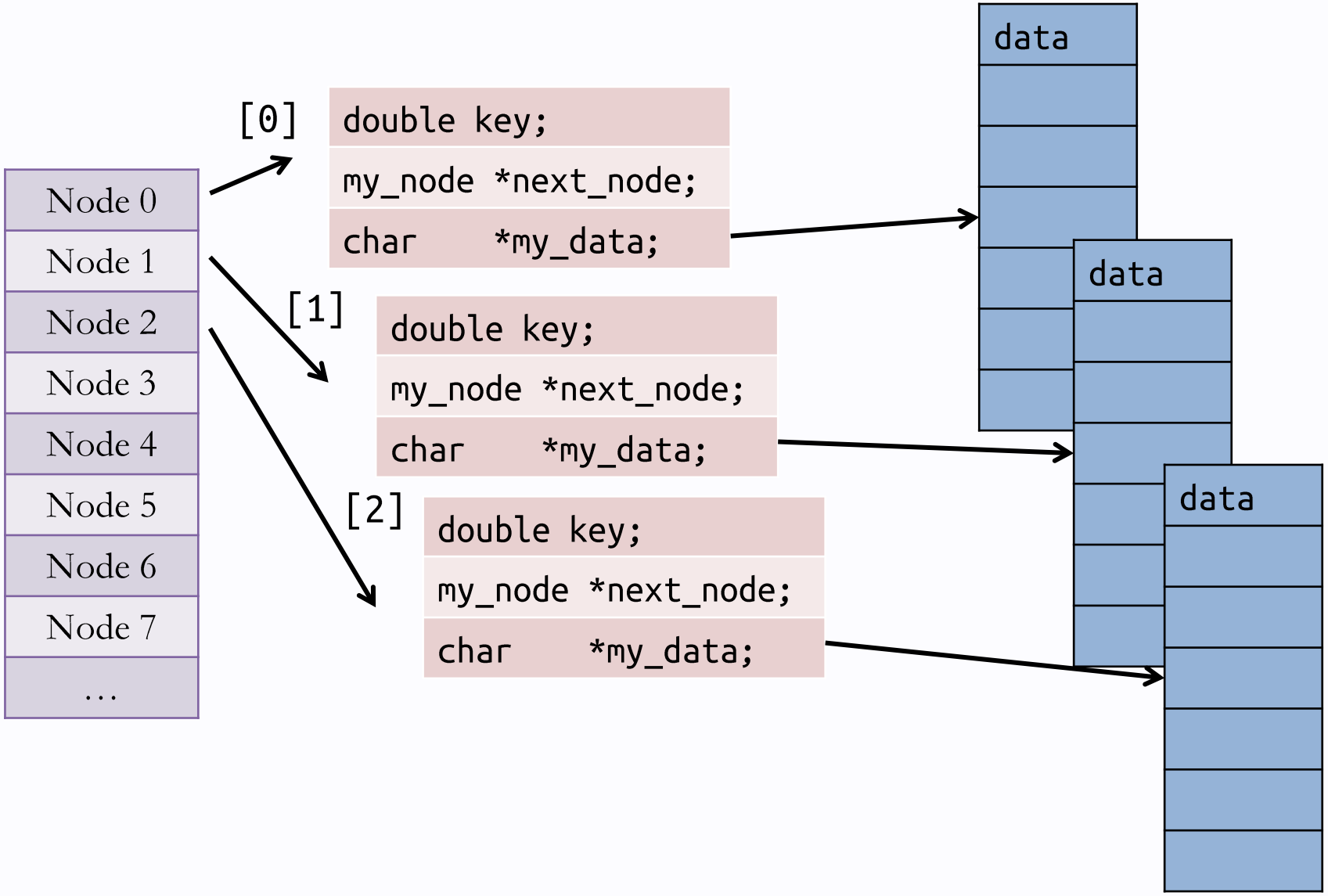
Field split: hot and cold

Split fields so that to keep consecutive fields used sequentially

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

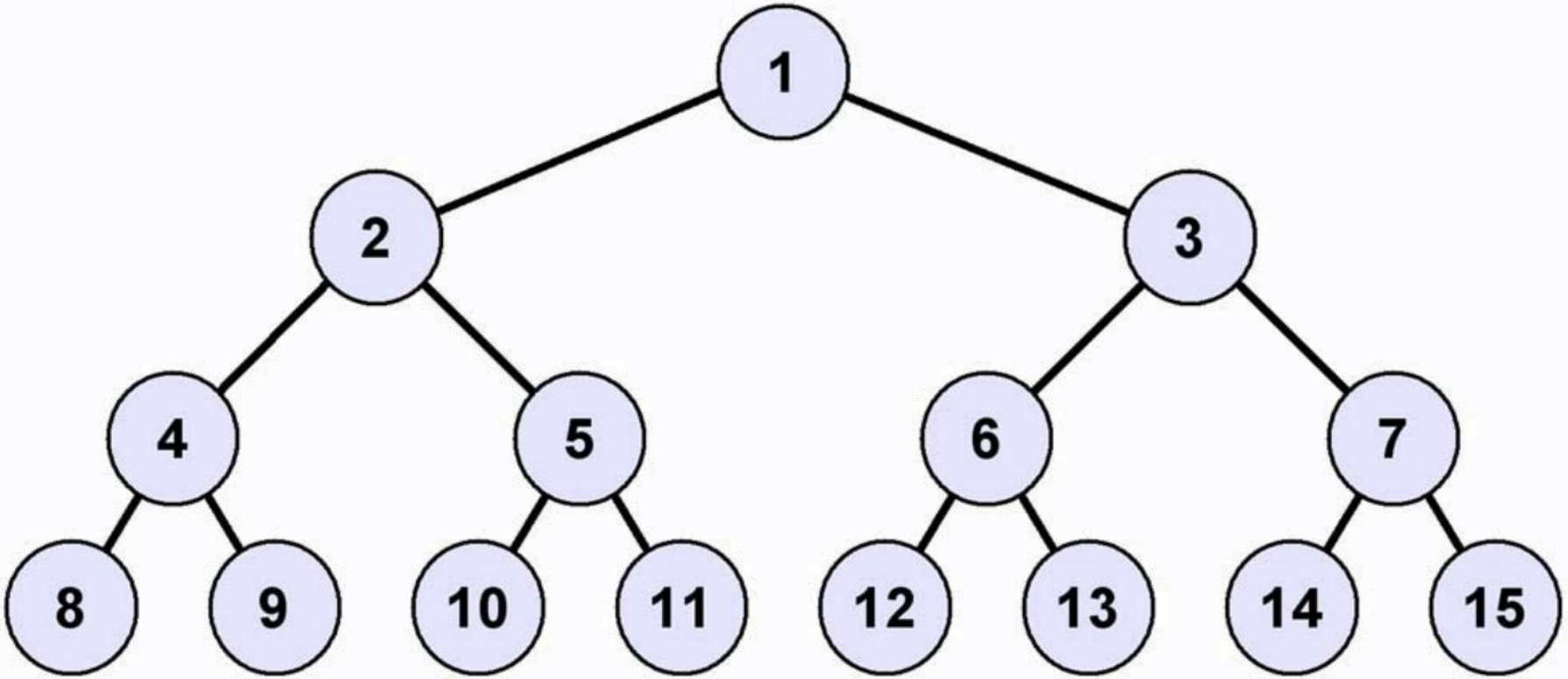
```
struct my_node
{
  double   key;
  my_node *next_node;
  void    *my_data;
}

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

Field split: hot and cold



Data structure reordering

Example: TREEs – depth-first order

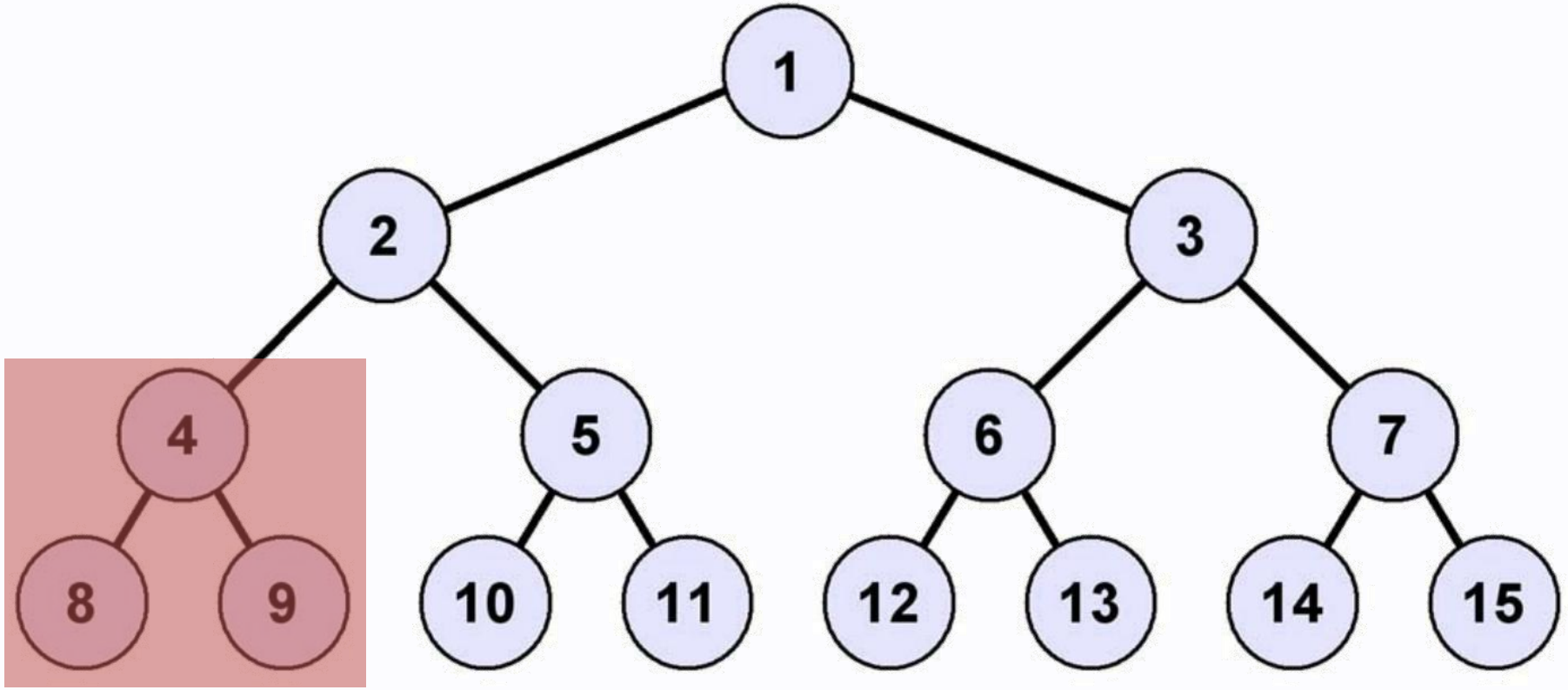


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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

Data structure reordering

Example: TREEs – depth-first order

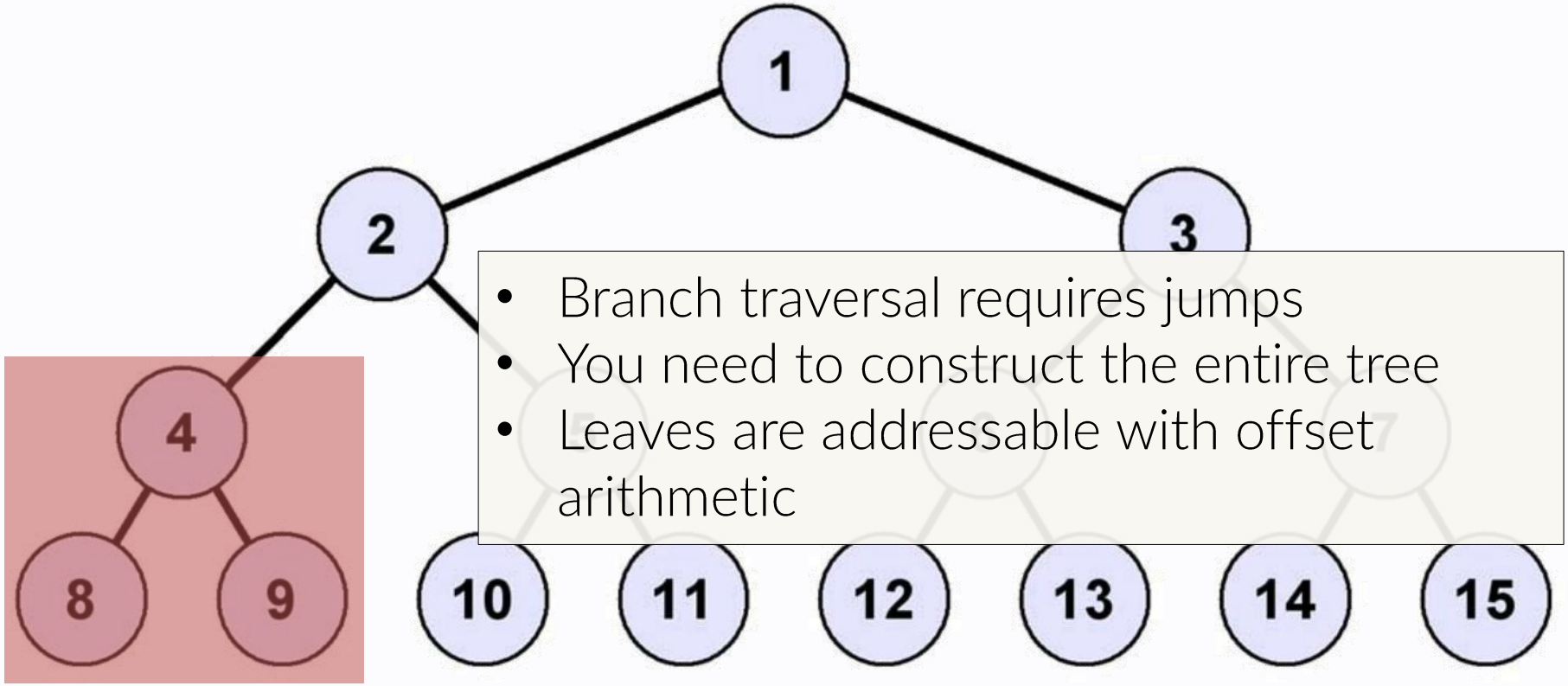


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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

Data structure reordering

Example: TREEs – depth-first order

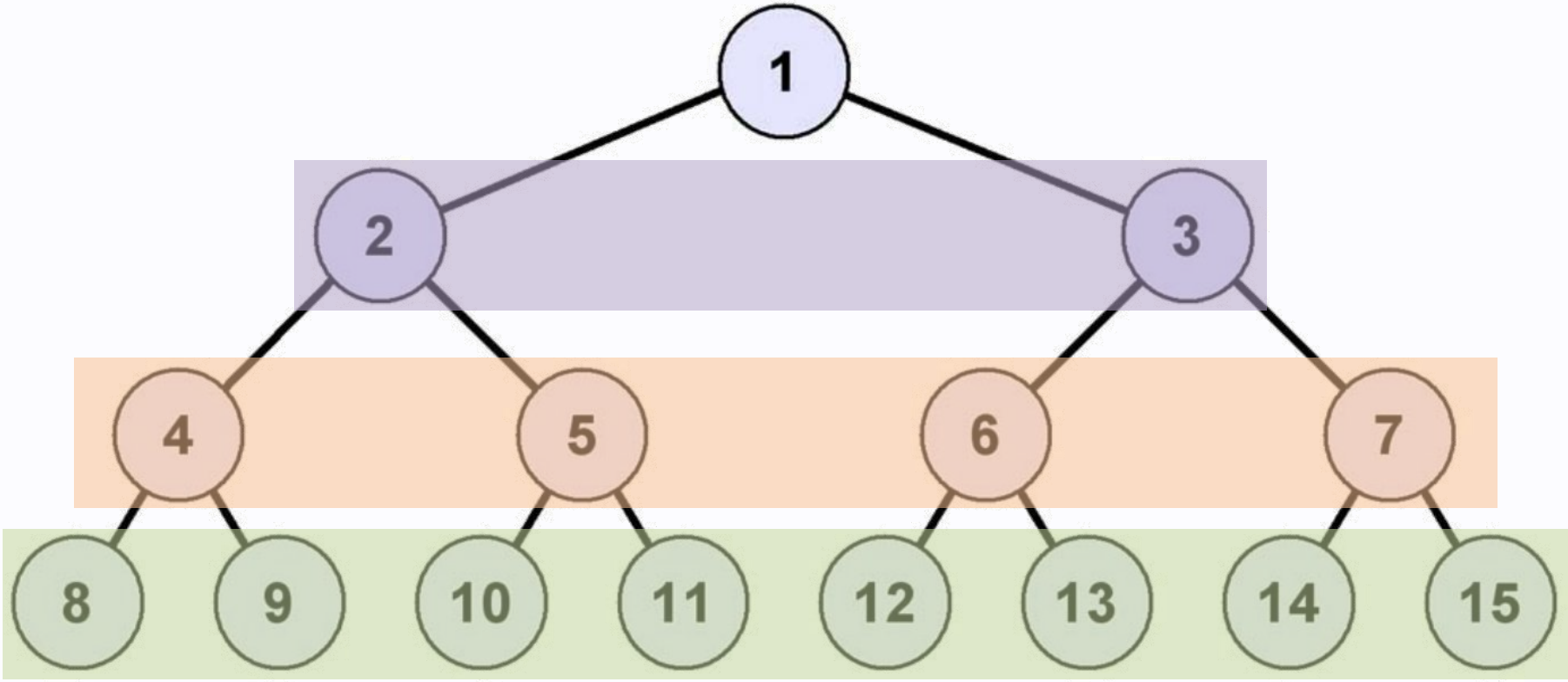


- Branch traversal requires jumps
- You need to construct the entire tree
- Leaves are addressable with offset arithmetic

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

Data structure reordering

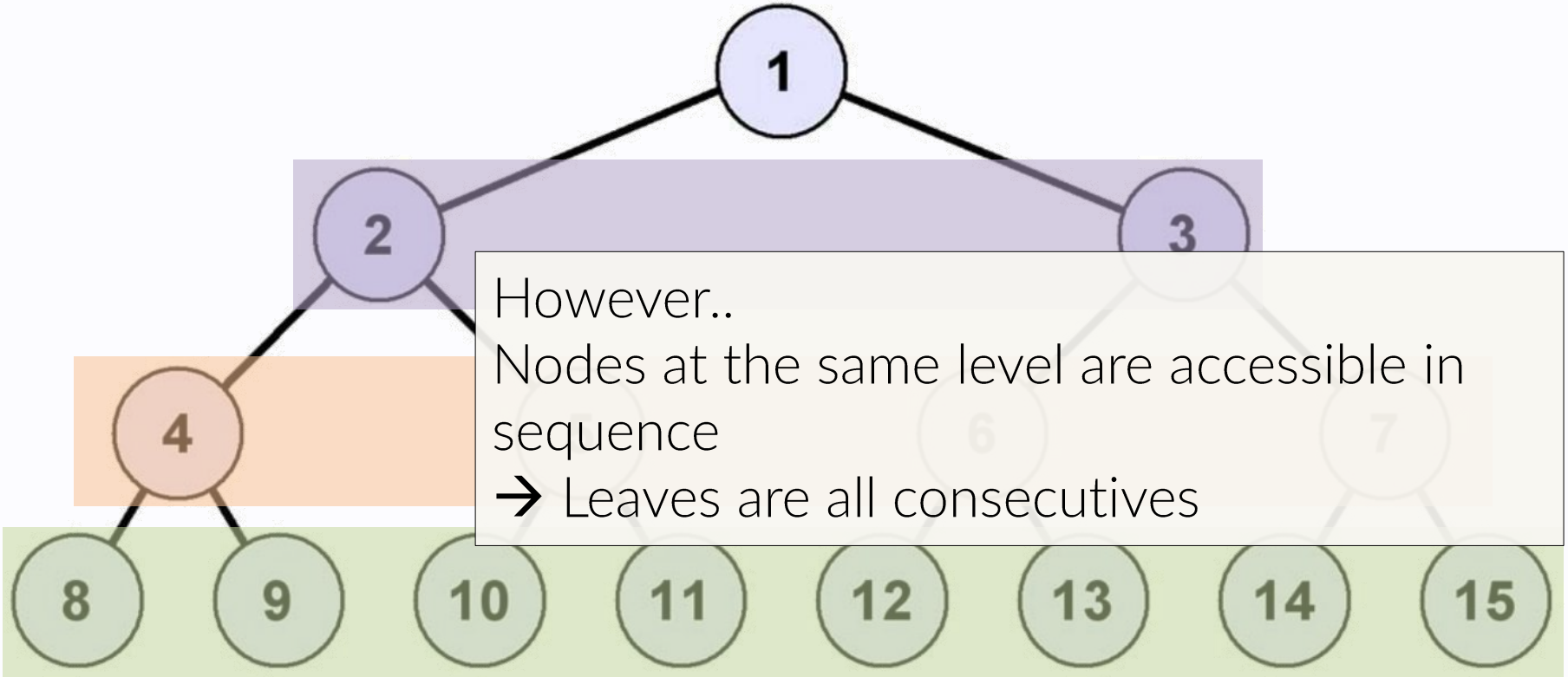
Example: TREEs – depth-first order



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

Data structure reordering

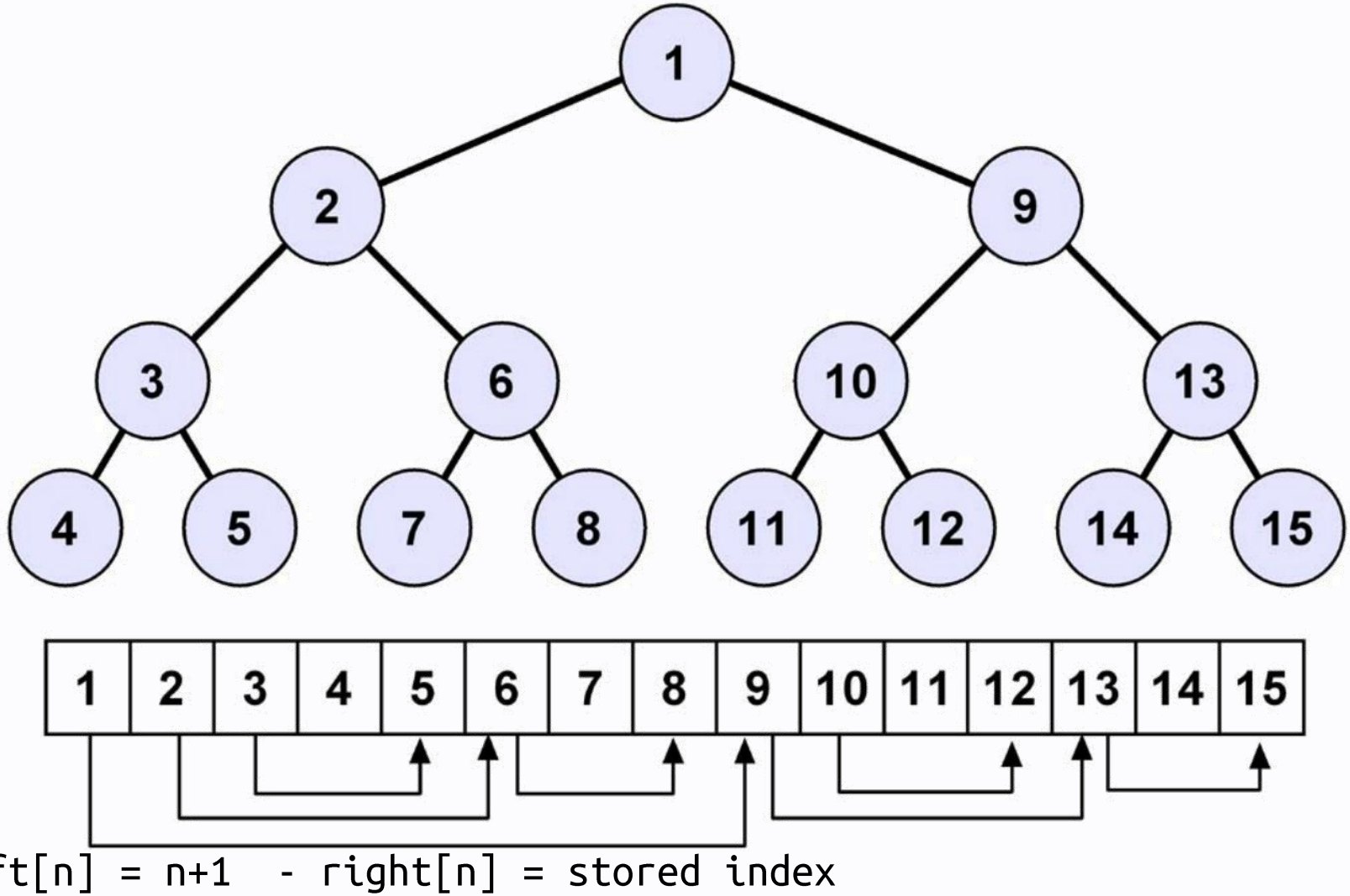
Example: TREEs – depth-first order



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

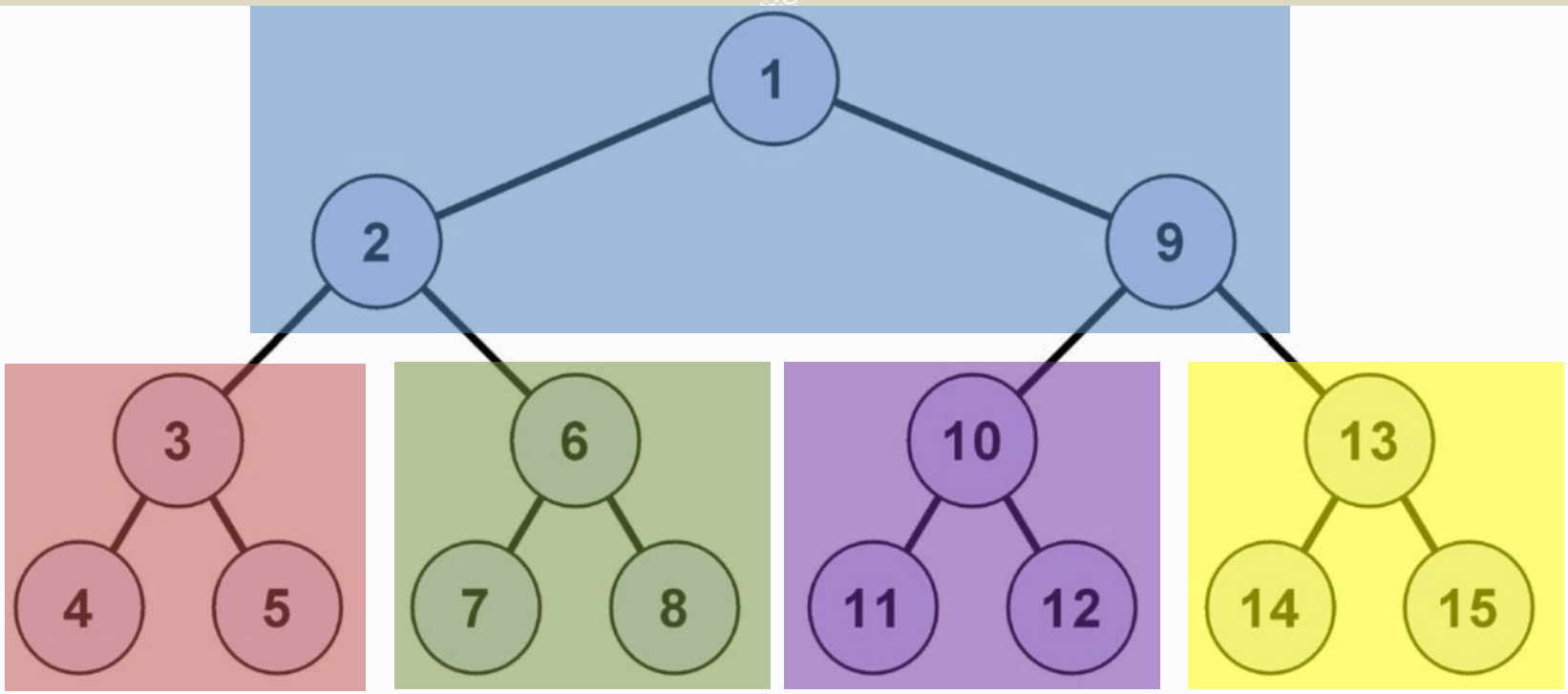
Data structure reordering

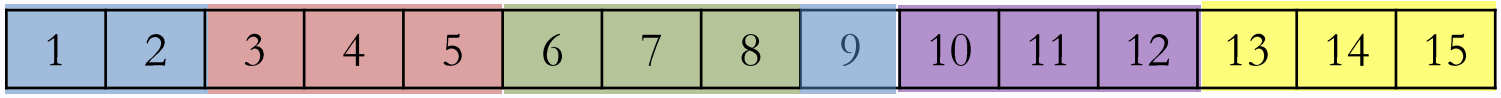
Example: TREEs – breadth-first order



Data structure reordering

Example: TREEs – breadth-first order

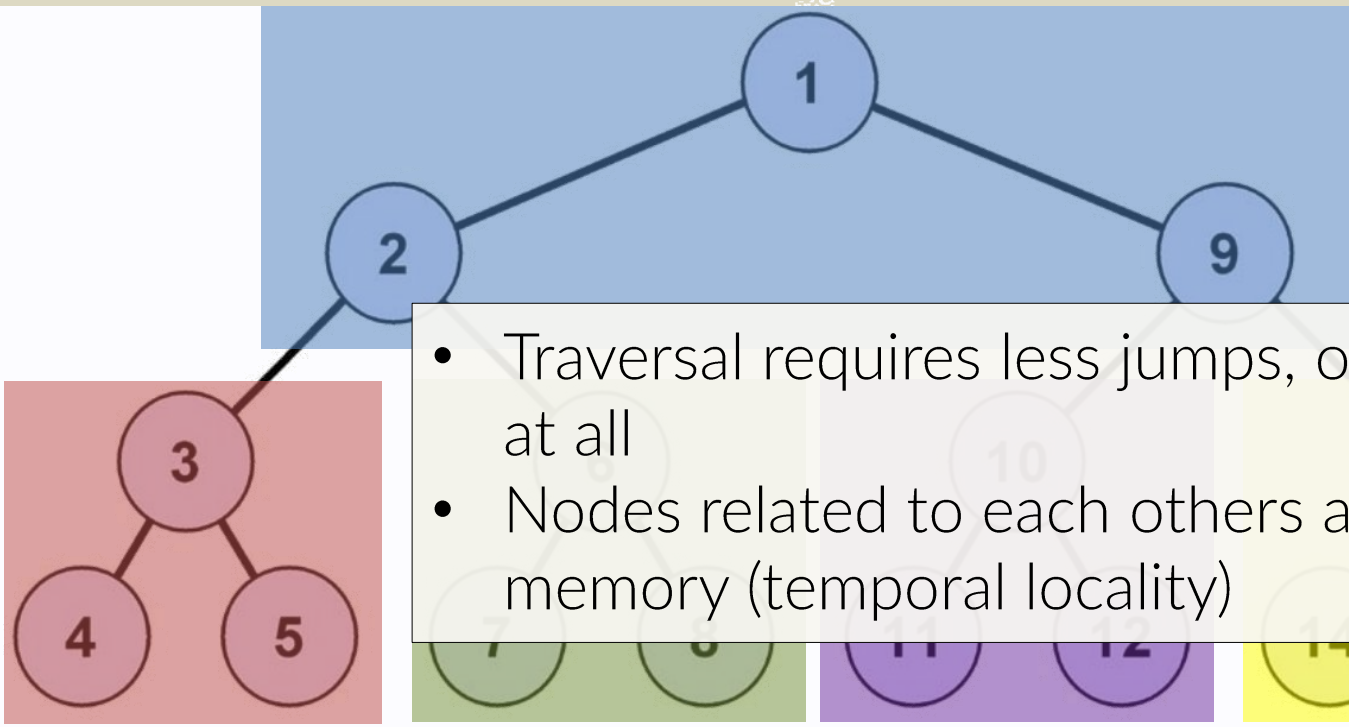




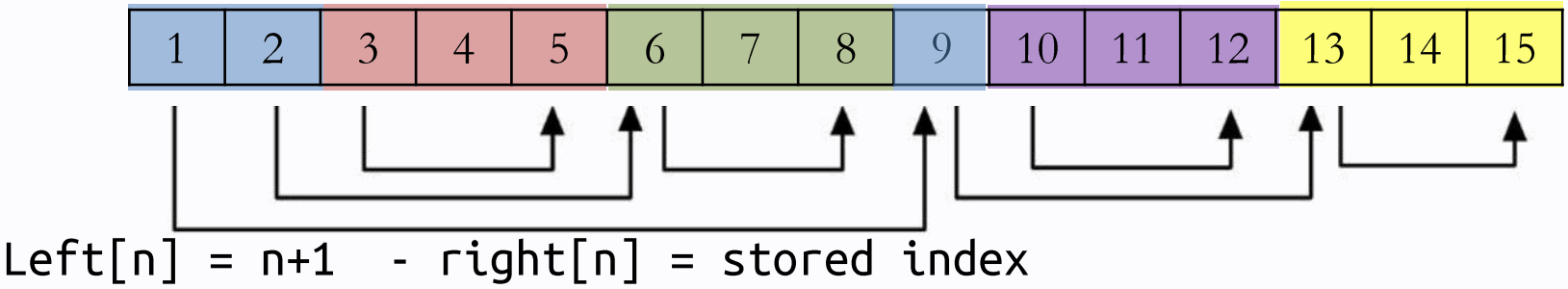
Left[n] = n+1 - right[n] = stored index

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.

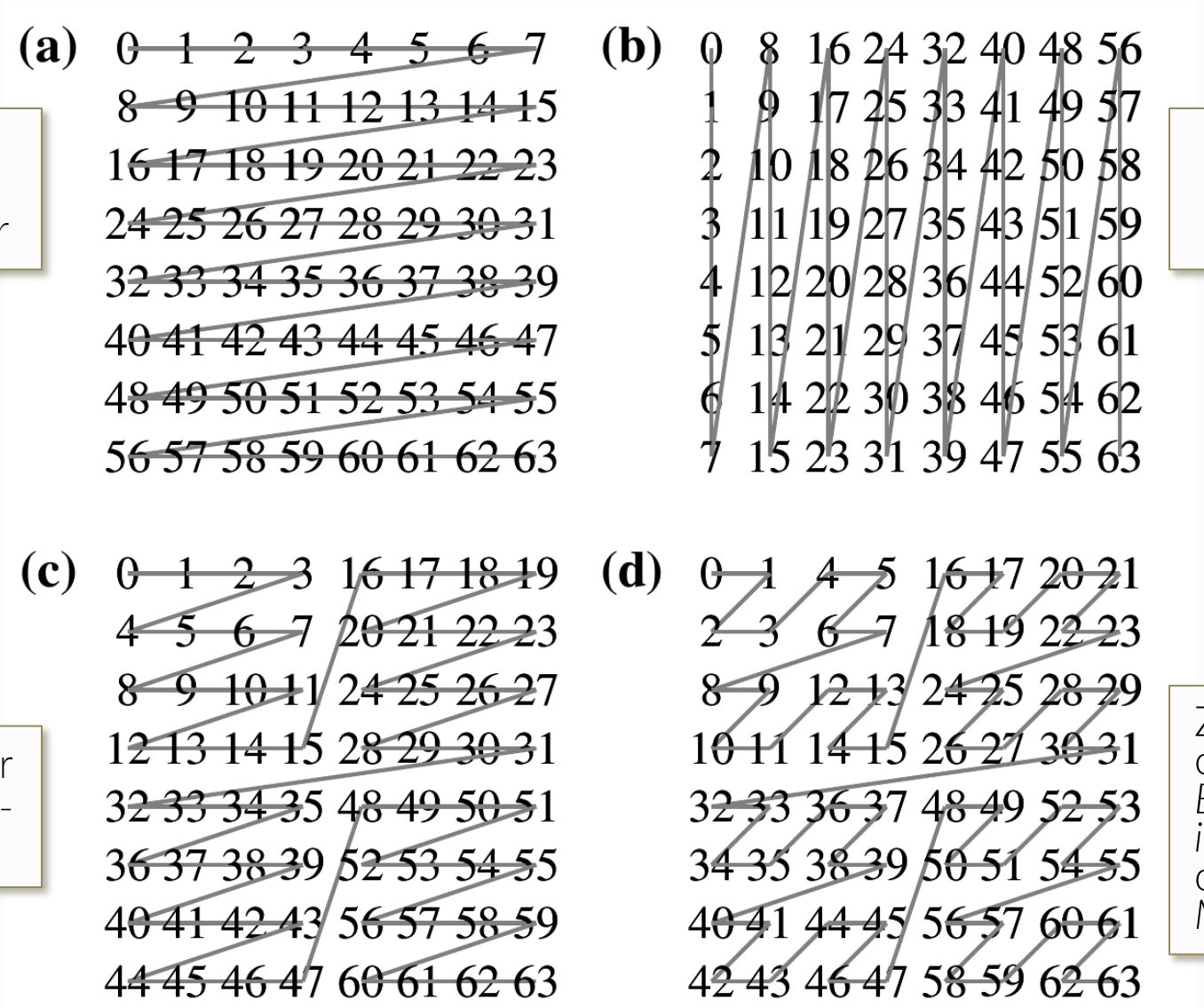
Data reordering

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`*

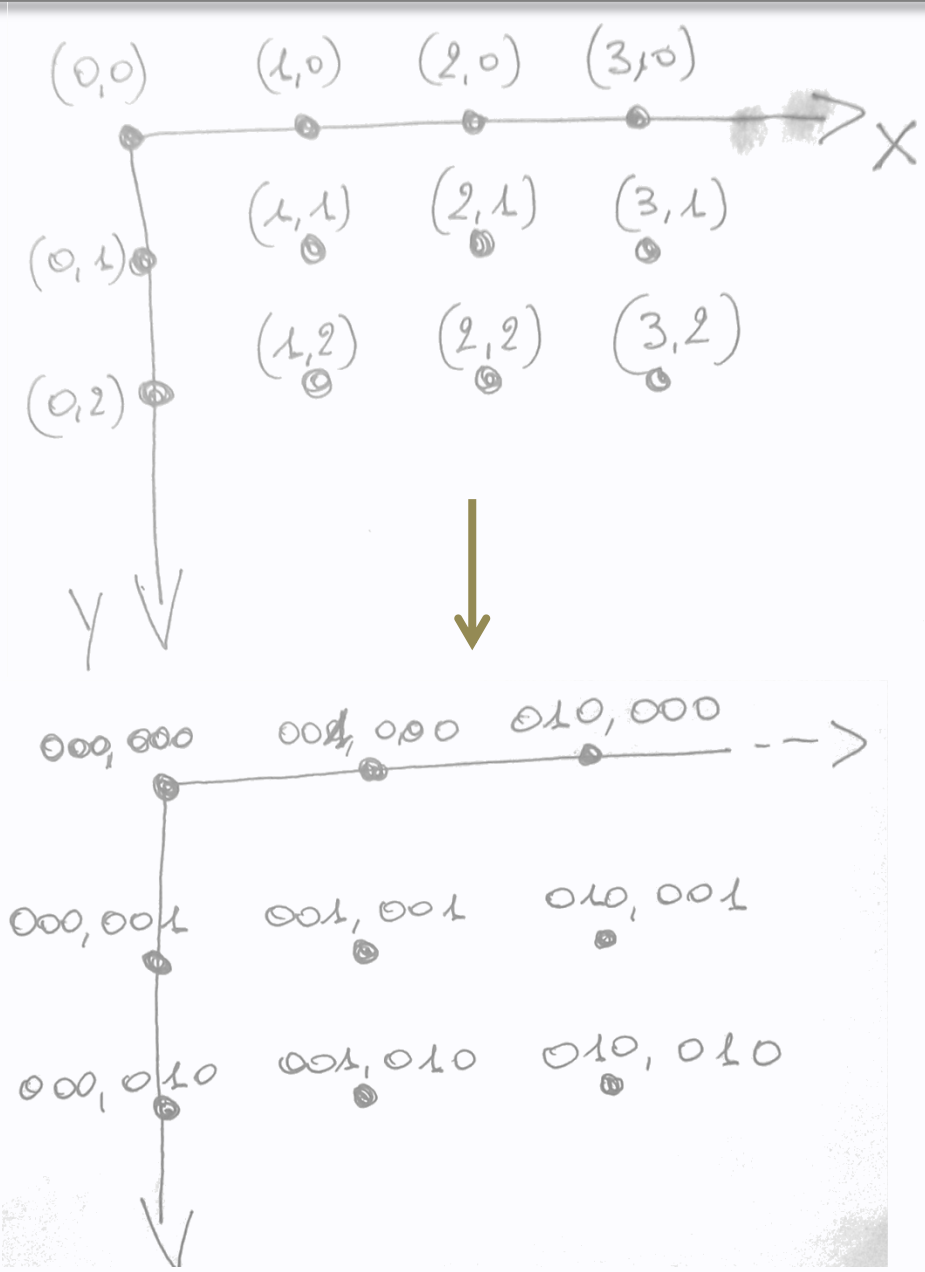
on some spatially-distributed data with integer coordinates.

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* :

$x = x_i \dots x_2 x_1 x_0$

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

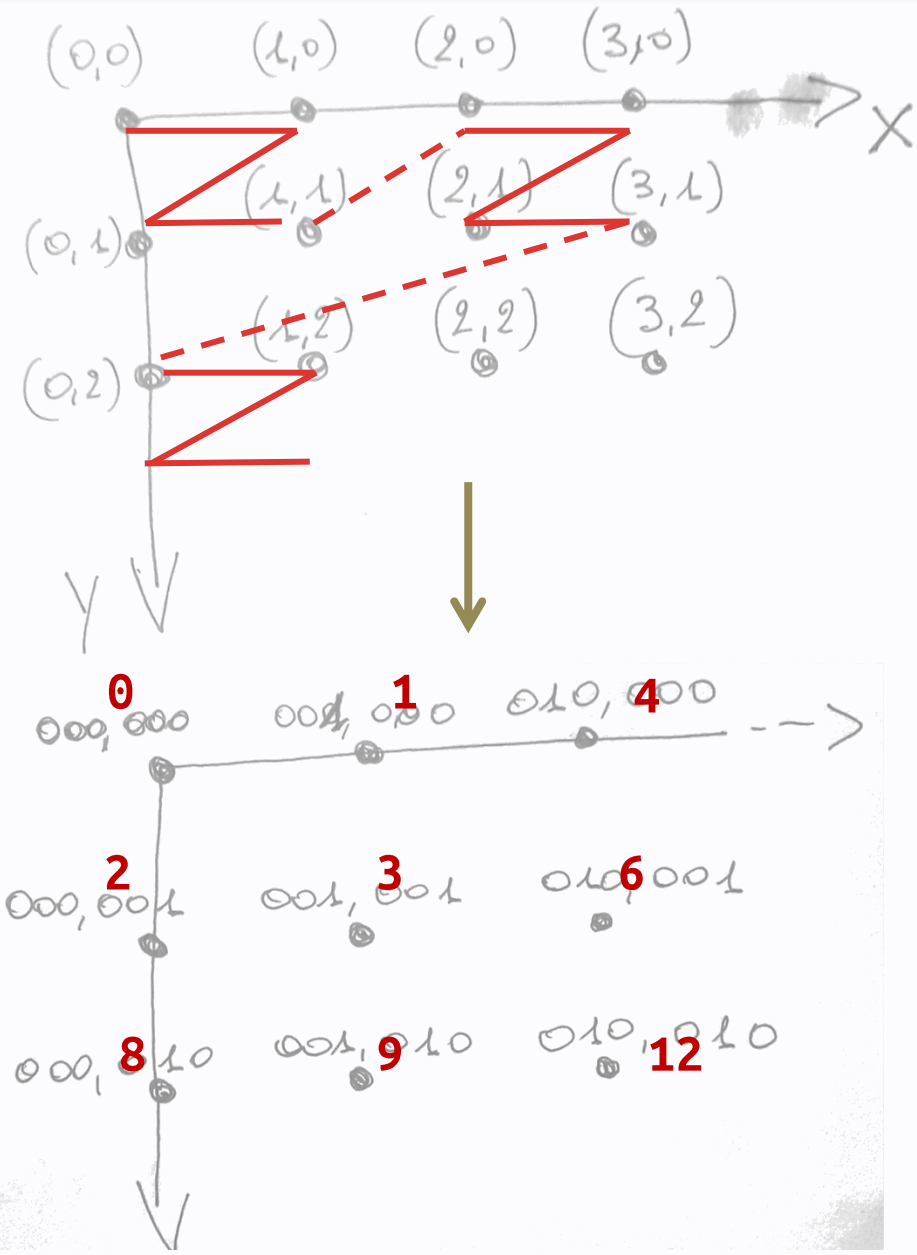
$(x_i \dots x_2 x_1 x_0 , y_i \dots y_2 y_1 y_0)$

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

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

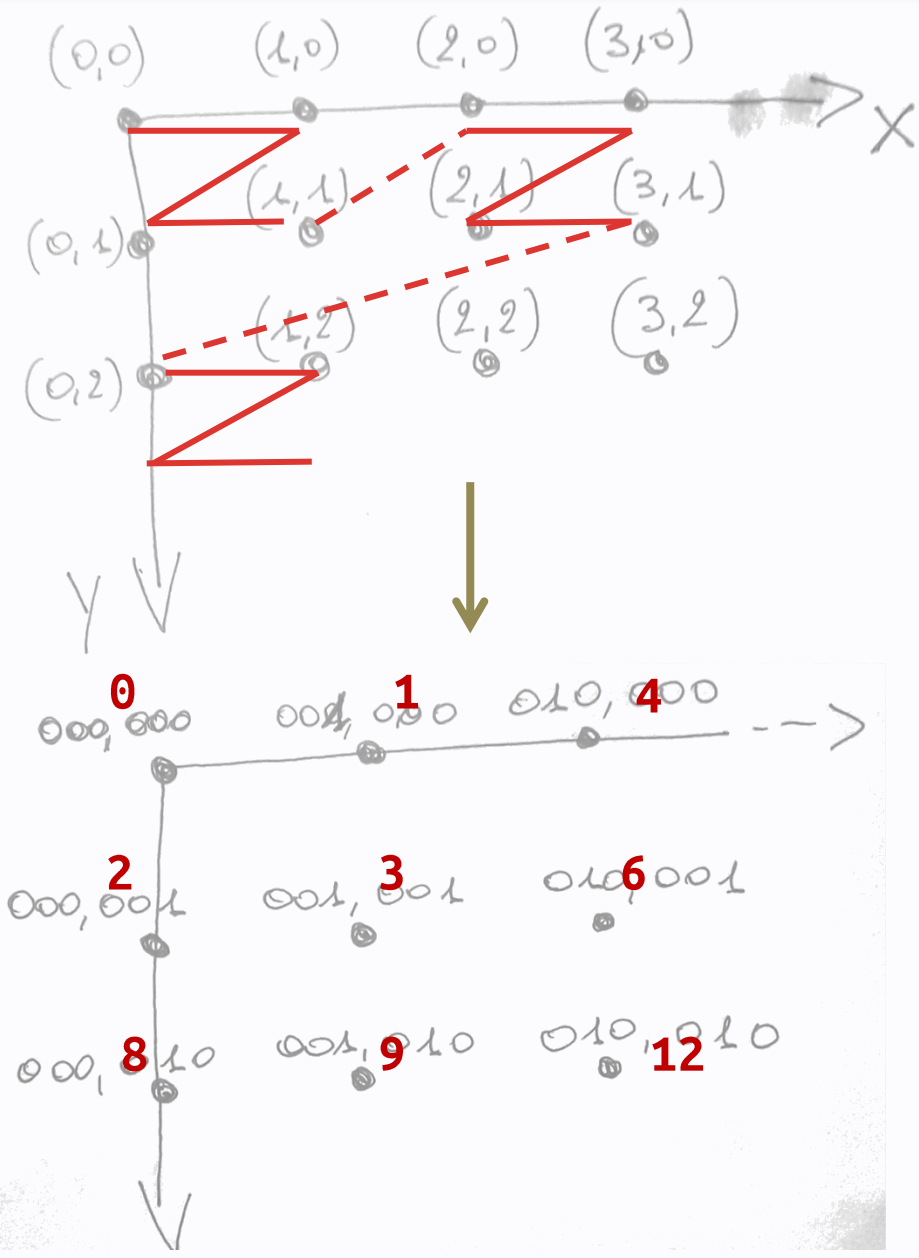
(0,0) → 00 00 = 0
(1,0) → 00 01 = 1
(2,0) → 01 00 = 4
(0,1) → 00 10 = 2
(1,1) → 00 11 = 3
…

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

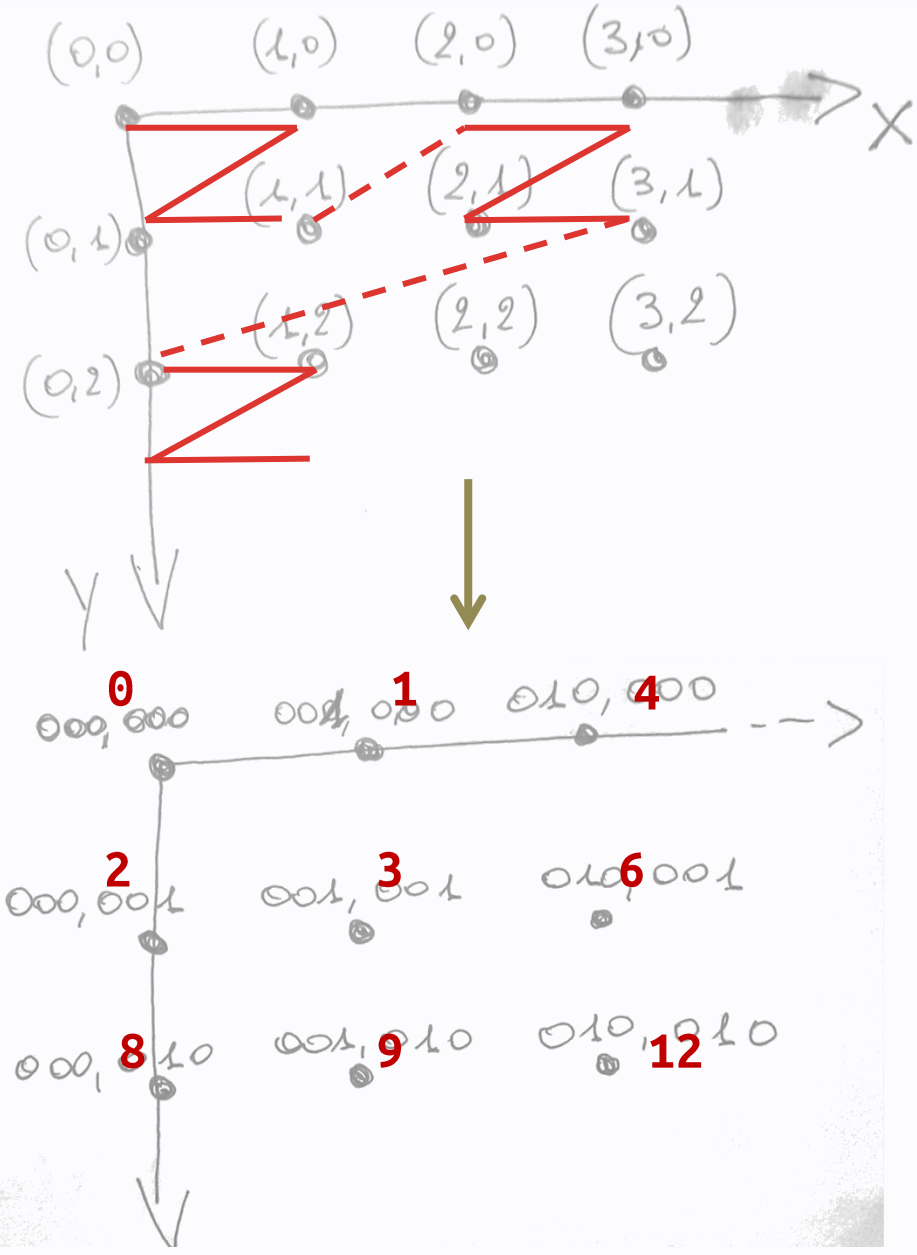
Data reordering



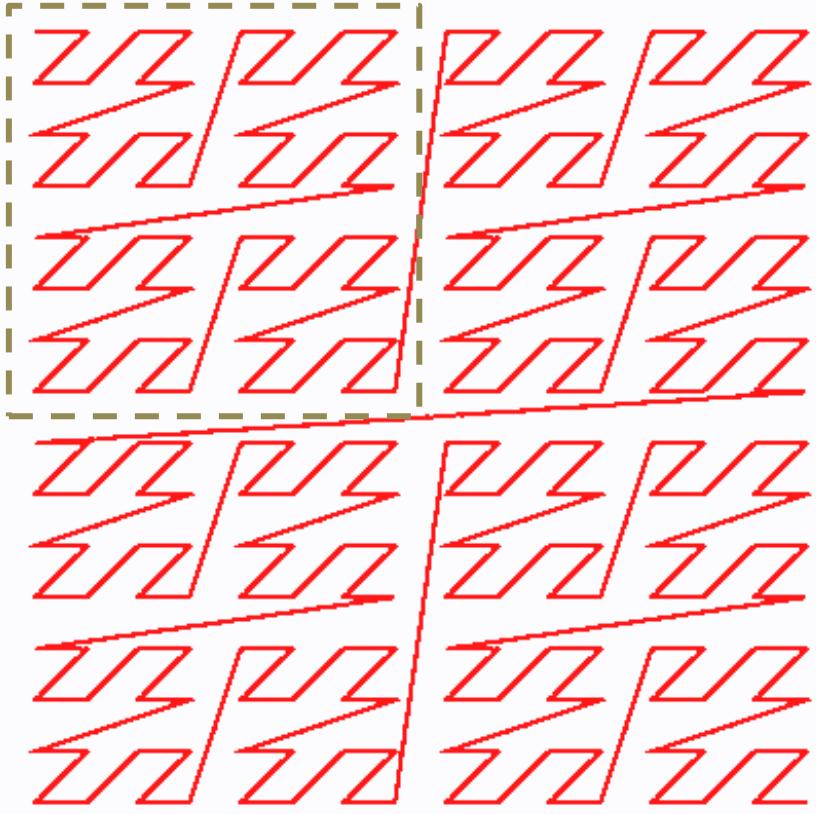
The result of this interleaving is a mapping from the 2-D plane to the 1-D line known as Z-line which is one of the **plane-filling curves** discovered by Peano

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

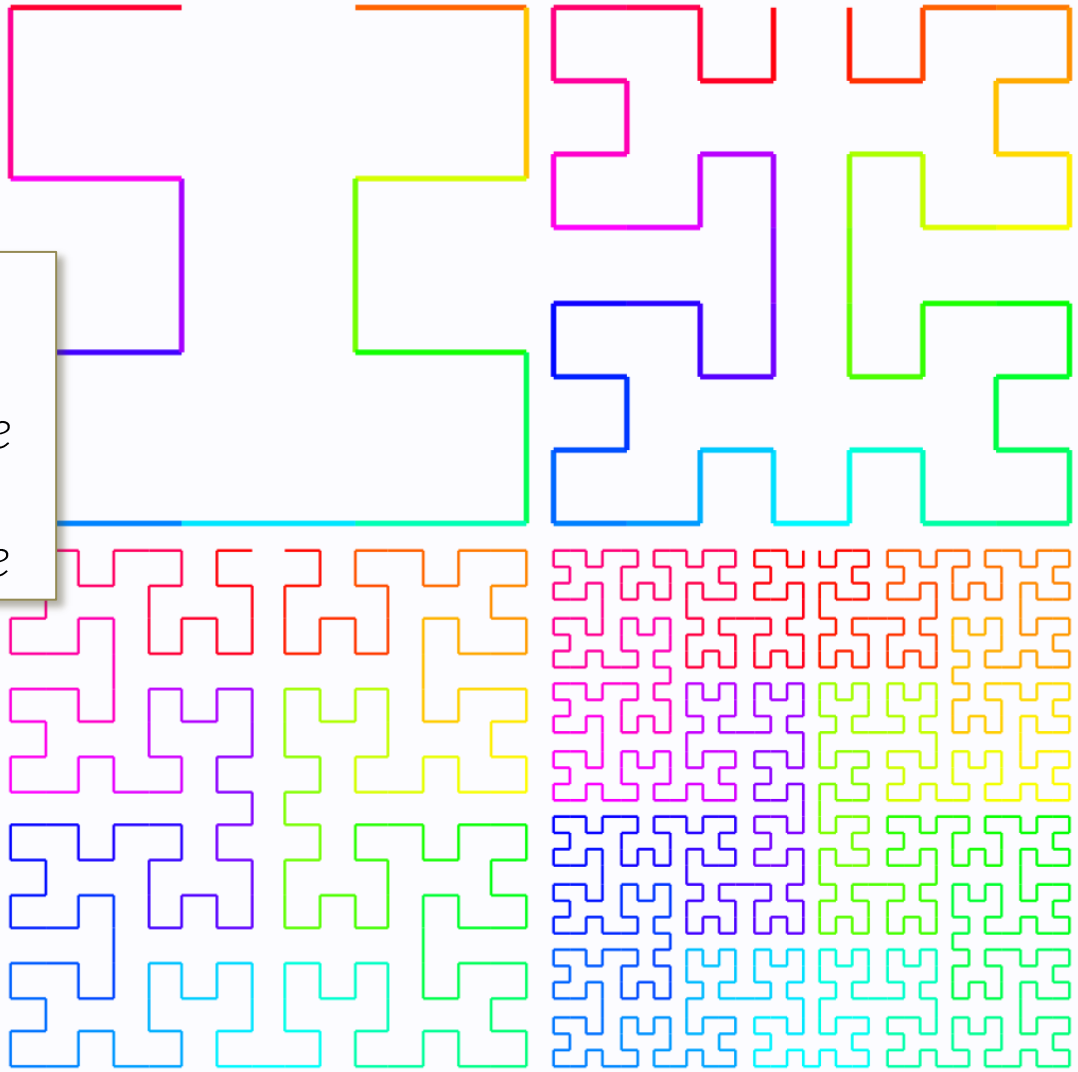


The wonderful land of *Peano*-curves

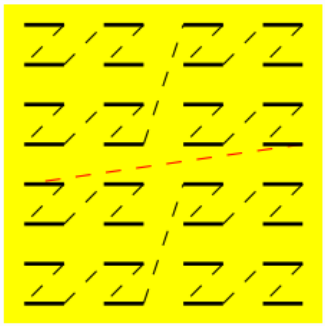
There are many of these curves

Hilbert curve

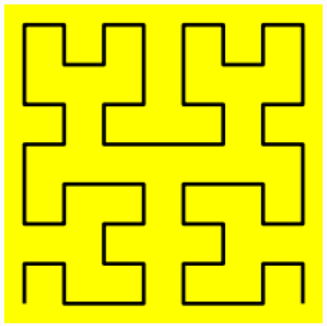
Note that there are no jumps along the curve



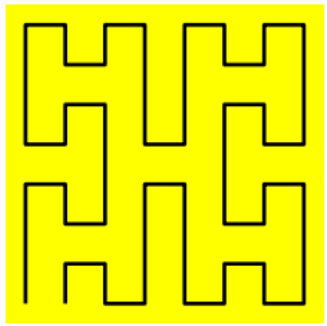
The wonderful land of *Peano*-curves



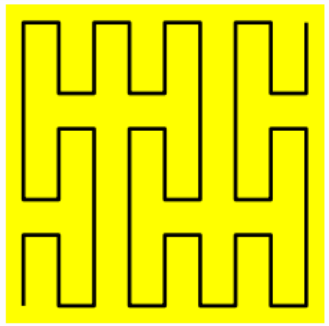
Z-order



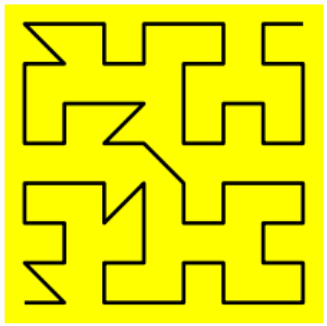
Hilbert curve



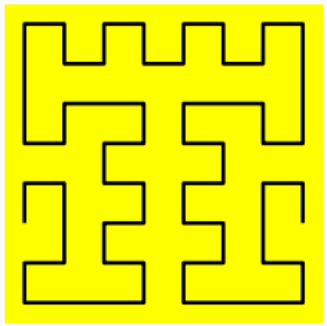
H-order



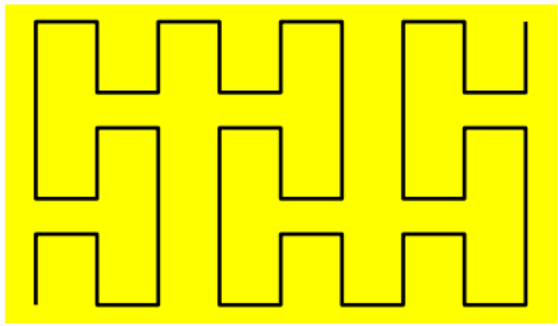
Peano's curve



AR^2W^2-curve



$\beta\Omega$-curve

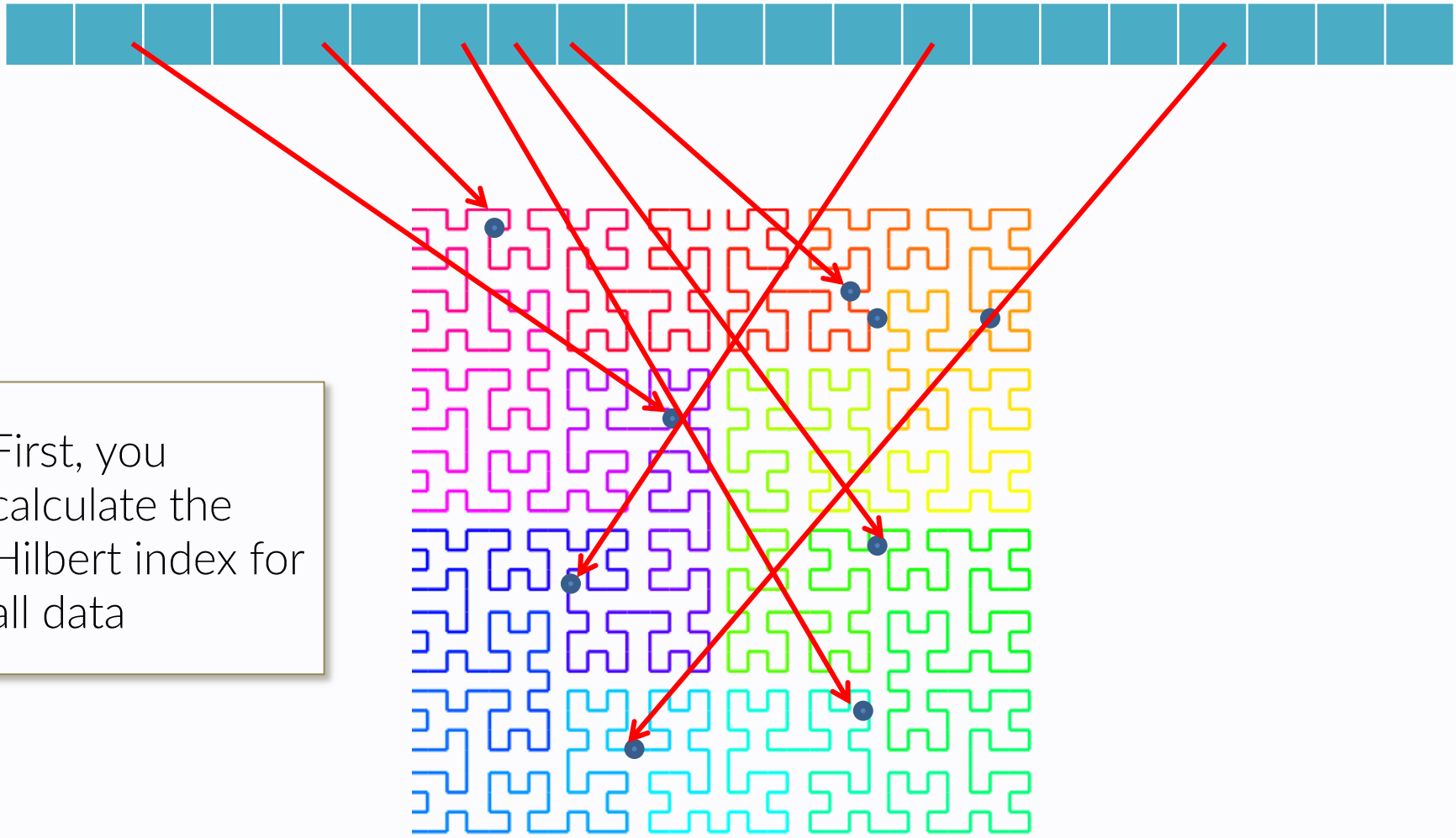


Balanced Peano

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

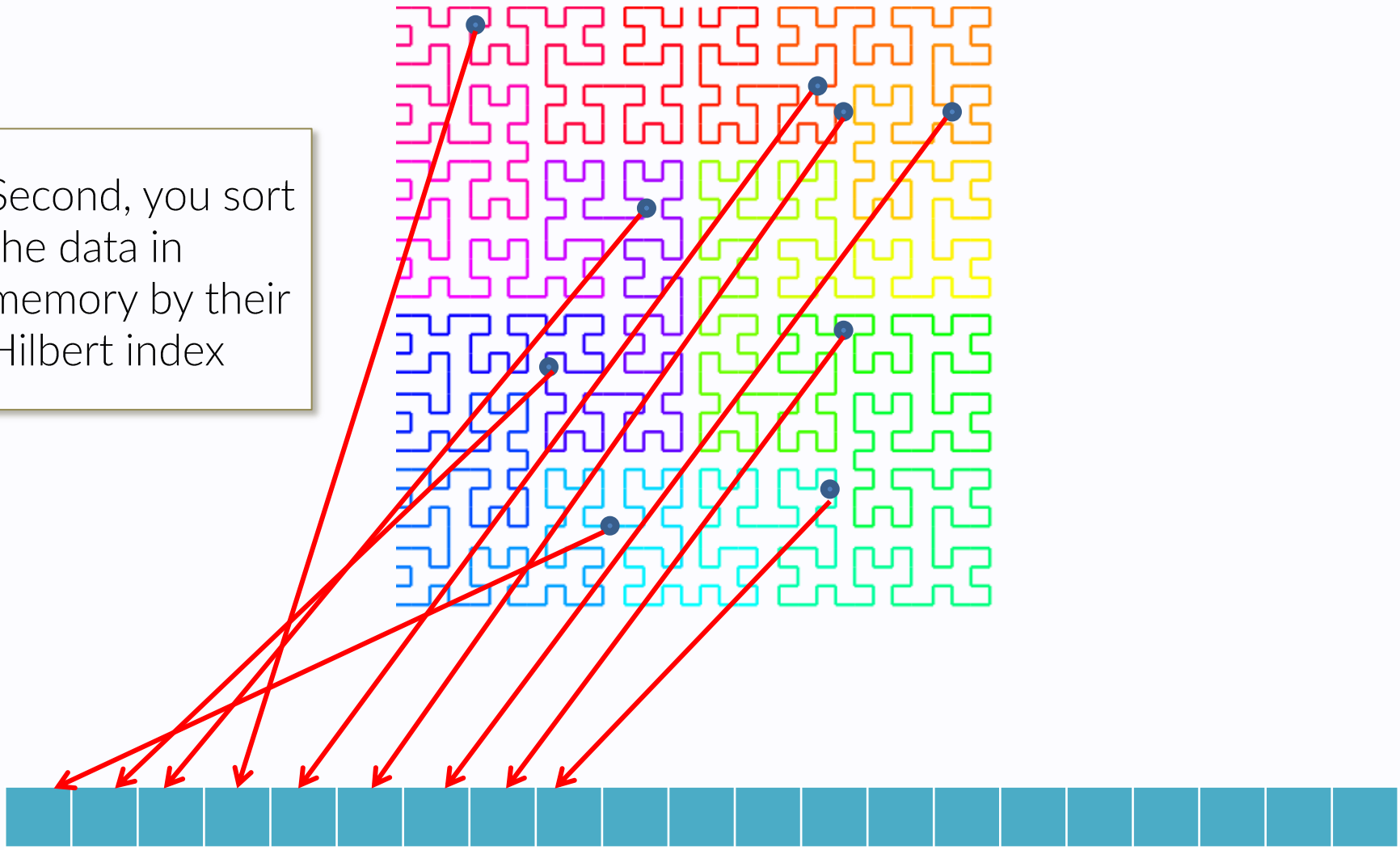


First, you calculate the Hilbert index for all data

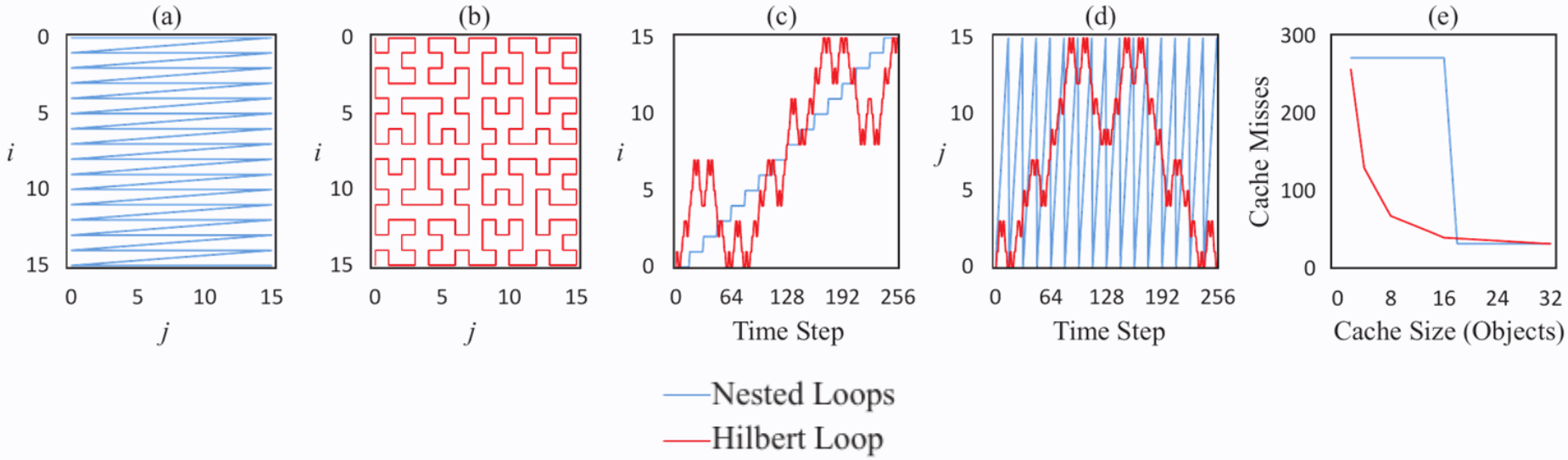
The wonderful land of *Peano*-curves

Re-ordering data in memory

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



The wonderful land of *Peano*-curves

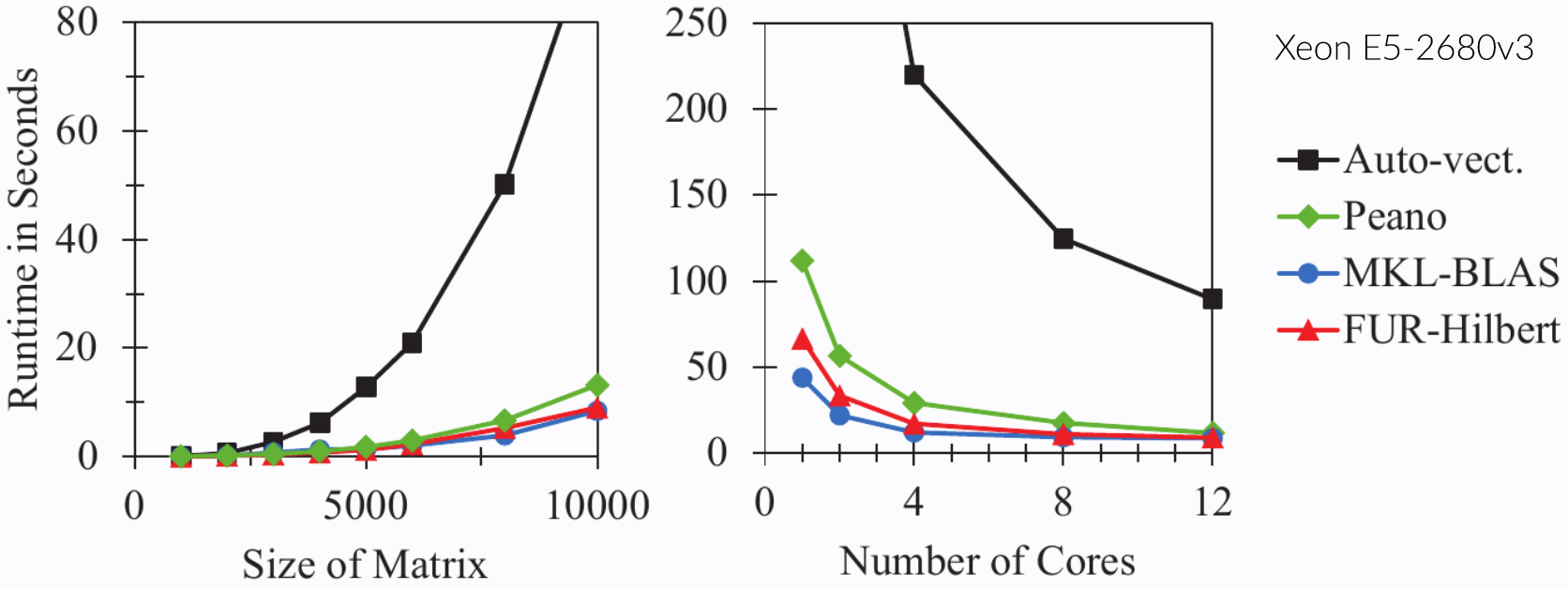


Cache-oblivious at work : traversal order

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

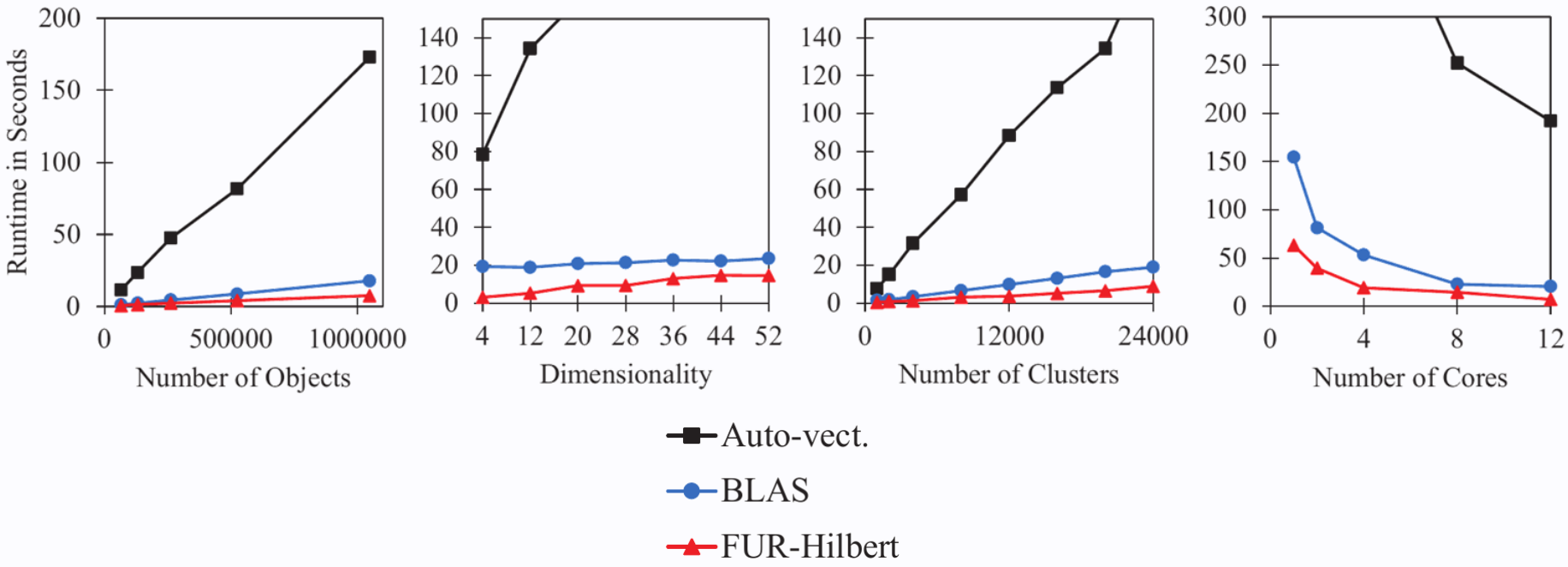
The wonderful land of *Peano*-curves



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];
  }
```

➔ ...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.O3

sum is 983597794767, elapsed seconds: 0.660148

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.O3

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

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.O3

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.O3n

sum is 983597794767, elapsed seconds 0.217864

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.O3n

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

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.O3n

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 vectorIzed!!`

Loops optimizations: branches

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

gcc -O3

```
93 0082 660FEFC0          pxor    xmm0, xmm0
94 0086 4989D8            mov     r8, rbx
95 0089 49C1E802          shr     r8, 2
96 008d 660FEFC9          pxor    xmm1, xmm1
97 0091 49F7D8            neg     r8
```

gcc -O3 -march=native

```
93 0090 4989D8            mov     r8, rbx
94 0093 C5F157C9          vxorpd  xmm1, xmm1, xmm1
95 0097 C5F957C0          vxorpd  xmm0, xmm0, xmm0
96 009b 49C1E802          shr     r8, 2
97 009f C4E1F32A          vcvtsi2sdq      xmm1, xmm1, QWORD PTR [rbp-72]
```

Pipelining

Loops optimizations: pipelining

Modern processor may be able to perform more than one operations at a time, or to pipeline operations

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

This way S is both read and written and the FP pipeline is difficult to be exploited

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]; }
```

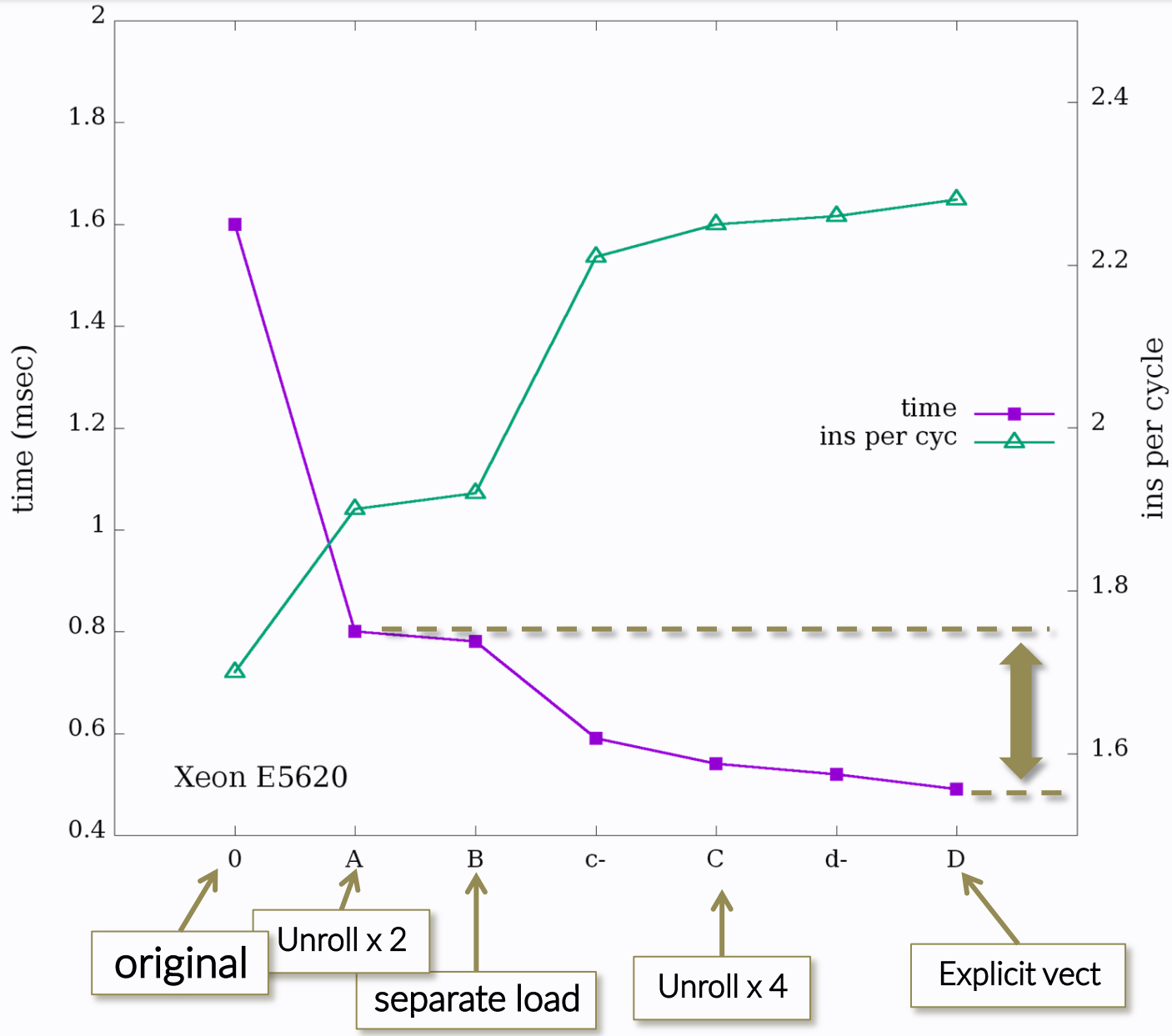
v. C

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



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
```

```
    return -1;
}
```

Prefetching

Prefetching by preloading

```
elem a = elements[0]
for ( i = 0; i < 4*N_4; i+= 4 )
 {
      elem e = elem[i+4]; // cache-miss non-block
      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;
 }
```

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

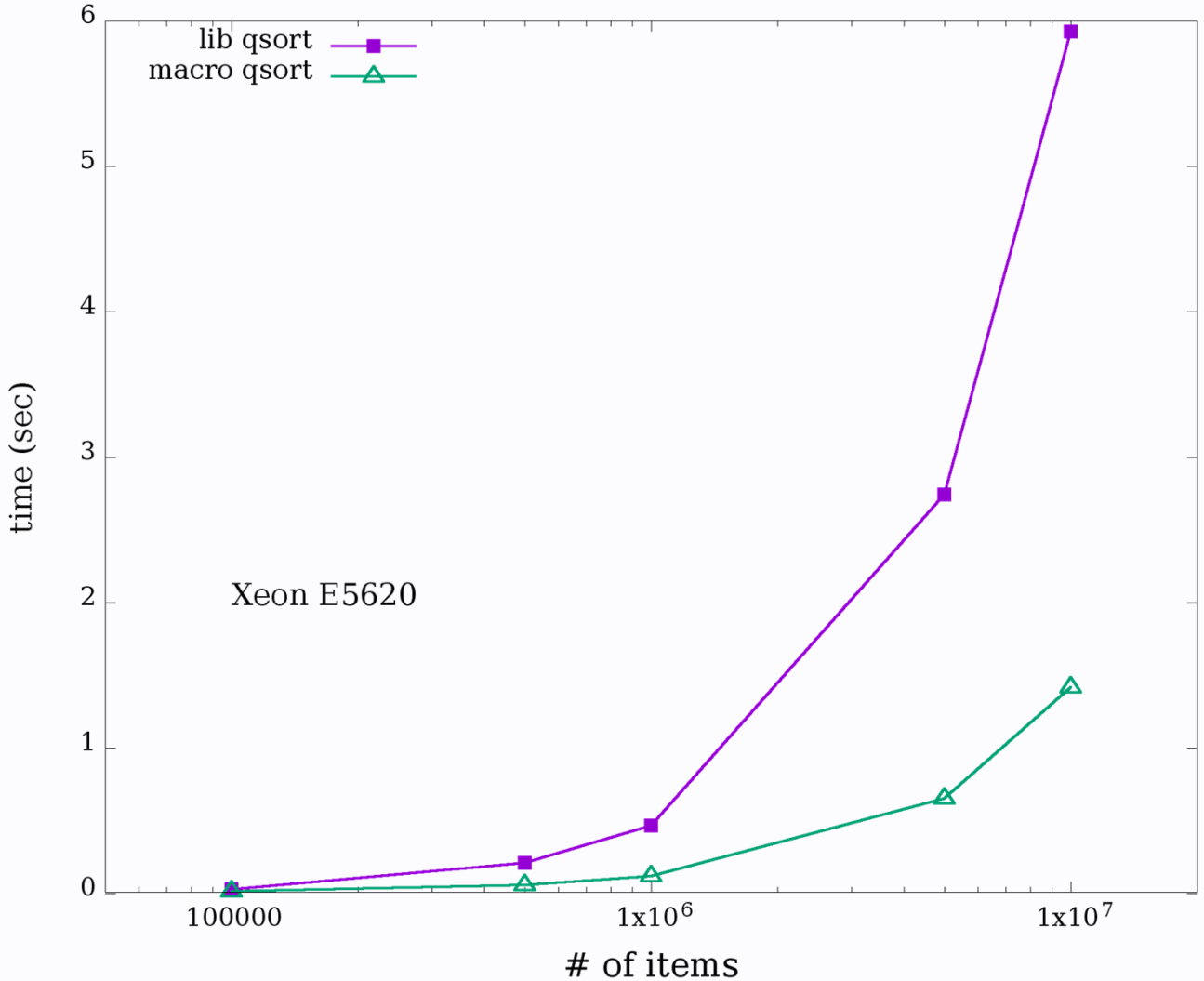
And you define MY_DATA_TYPE and DEALER and whatever else you need just before including the .h file

```
#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 **]**