Optimization
The Cache Hierarchy

Luca Tornatore - I.N.A.F.

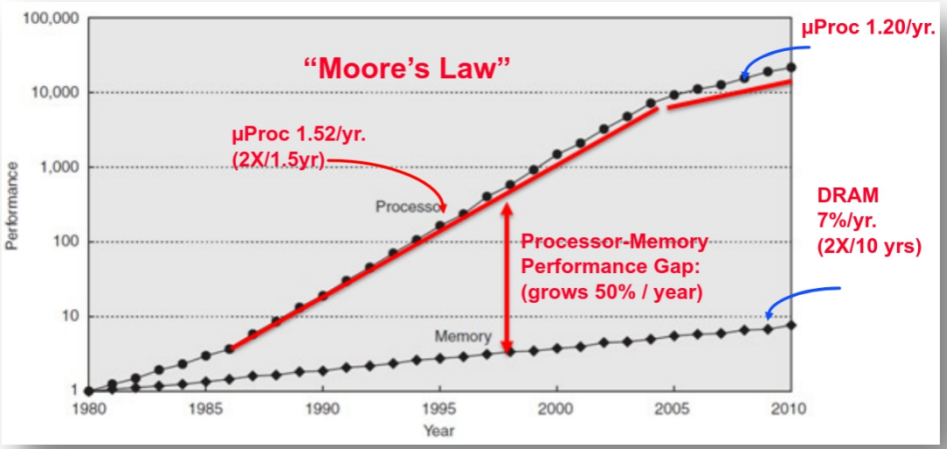
"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING

2021-2022 @ Università di Trieste

Let’s start with a small recap About memory and caches

Early 90s: CPU get faster than memory



Processor	Alpha 21164	
Machine	AlphaServer 8200	
Clock Rate	300 MHz	
Memory Performance	Latency	Bandwidth
I Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
D Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
L2 Cache (96KB on chip)	20 ns (6 clocks)	4800 MB/sec
L3 Cache (4MB off chip)	26 ns (8 clocks)	960 MB/sec
Main Memory Subsystem	253 ns (76 clocks)	1200 MB/sec
Single DRAM component	≈60ns (18 clocks)	≈30–100 MB/sec

Taken from a 1997 paper

The CPU may spend more time waiting for data coming from RAM than executing ops.
That is part of the so called "memory wall".
What is the solution ?

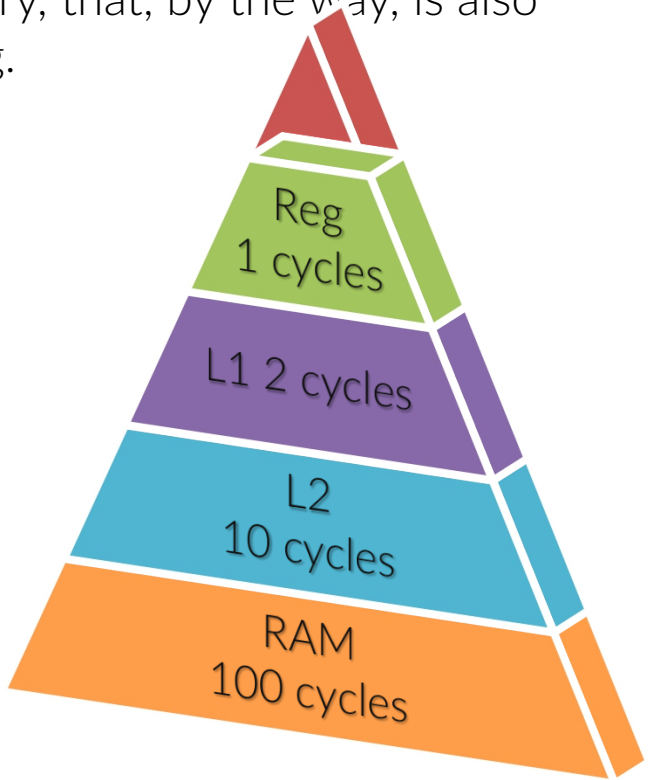
The cache memory

The solution is to equip your CPU with a *faster* memory, that, by the way, is also *way* more expensive and *way* more energy consuming.

Furthermore, to be faster it ought to be *extremely closer*.
All in all, the new memory that will be called ***cache***, will be much *smaller* than RAM.

The cache itself has a hierarchy:

- *Level-I* is inside each core.
- *Level-II* is also inside the core, or may be shared by more cores.
- *Level-III* is inside the CPU, shared by many cores.



The cache memory

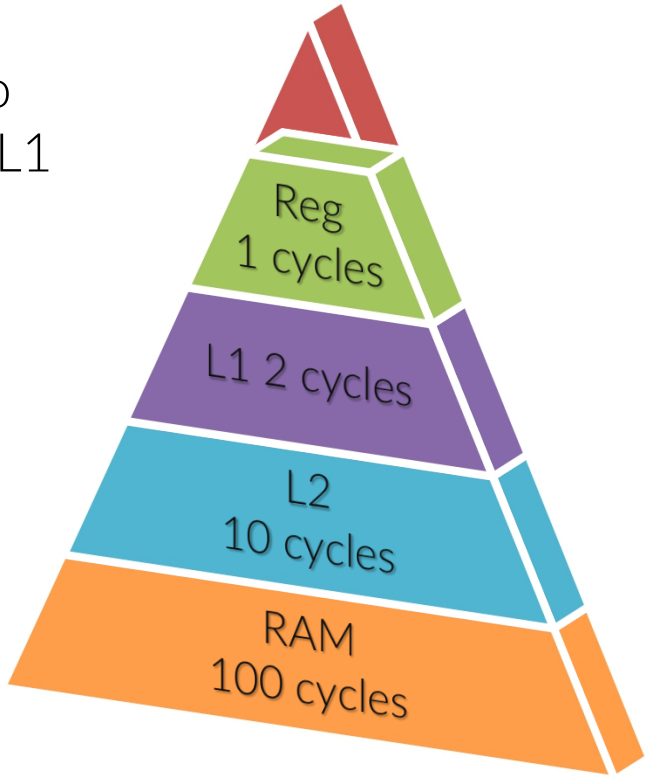
Let's suppose that we have only L1 and RAM. What would be the average cost, *in CPU cycles*, to retrieve the data a a function of how good is the L1 occupancy ?

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

} 50% to 150% slower



The cache memory

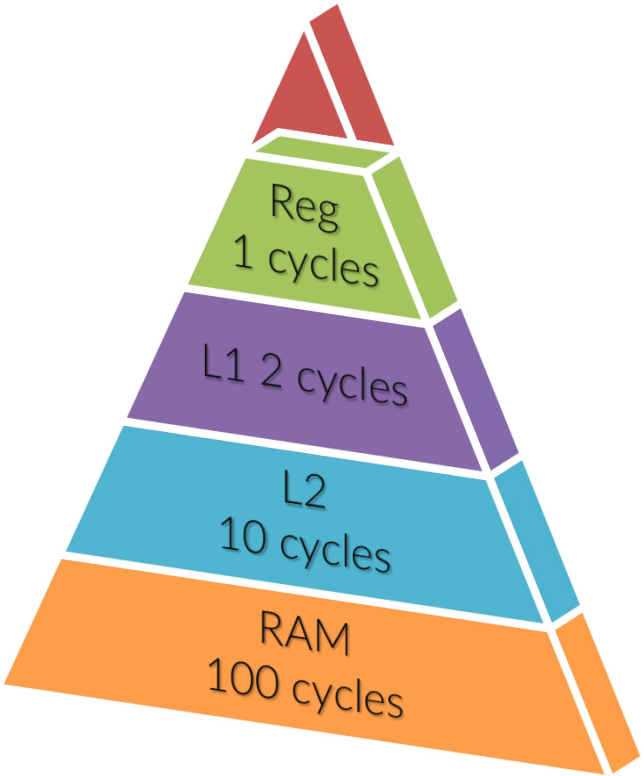
Let's now add a 2nd level of cache, L2:

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

The average cycles-per-load is much better now with an additional larger L2 (nowdays you find also an even larger L3)



The principle of locality

We are quite naturally led to the “principle of locality”:

> Data are defined “local” when they reside in a “small”portion of the address space that is accessed in some “short” period of time
>
> → local data are likely to be in the cache

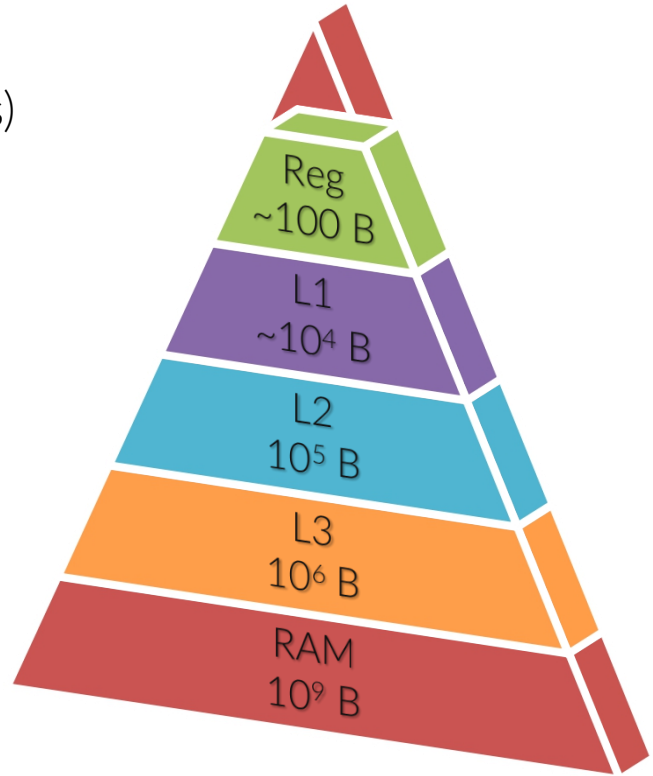
Temporal locality	if an address is referenced, it is likely to be referenced again soon
Spatial locality	if an address is referenced, close addresses are likely to be referenced soon

Memory→Cache mapping

Question:
The RAM contains ~10^9 bytes, while L1 contains ~10^4 bytes (32KB for data and 32KB for instructions)

So, how do you map the RAM in to a given level of cache, for instance L1, in an effective way?
The main problems are:

- *Where* to map an address
- *What if* the location in L1 is already occupied?



Memory→Cache mapping

Let's say that both the RAM and the cache are subdivided in blocks of equal size (for instance, 64B): you do not load just a byte in your cache but an entire block (normally called *line*)

RAM

Block number	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

cache

cache block number	0	1	2	3	4	5	6	7
memory block number	.	.	.	.	.	.	.	.
data								
valid bit	0	0	0	0	0	0	0	0
dirty bit	0	0	0	0	0	0	0	0

Each "`block`" here is 64B long

Memory→Cache mapping

Full mapping

Data can be placed in any free cache block

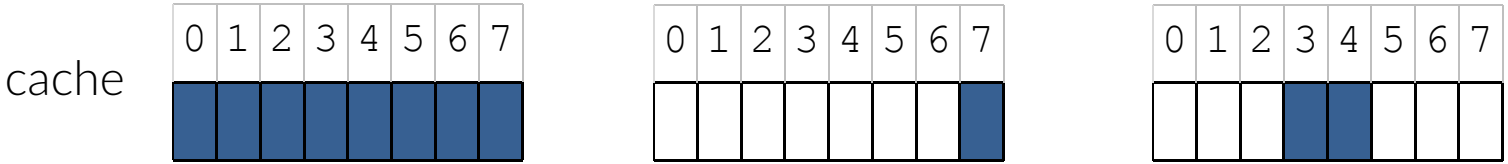
Direct mapping

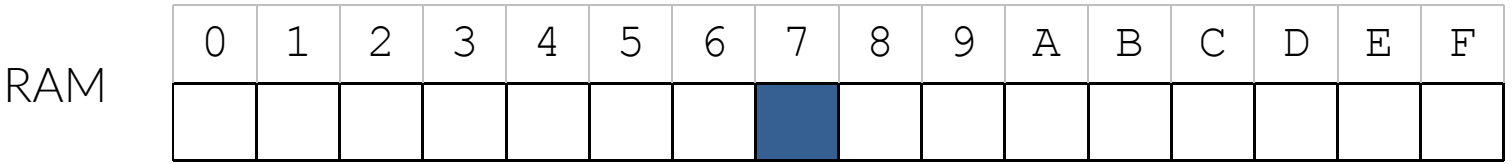
Data can be placed only in a given block i.e. **`block_num % blocks_in_cache`**

n-way associative

Data can be placed in few cache blocks i.e. **`block_num % (blocks_in_cache / n)`**

n=2





Memory → Cache mapping

Full mapping

Data can be placed in any free cache block.

Pros
Very efficient in writing (minimizes conflicts).

Cons
Very inefficient in reading (all the locations could contain the addressed data).

Direct mapping

Data can be placed only in a given block

Pros
Very efficient in writing (no search for available locations).

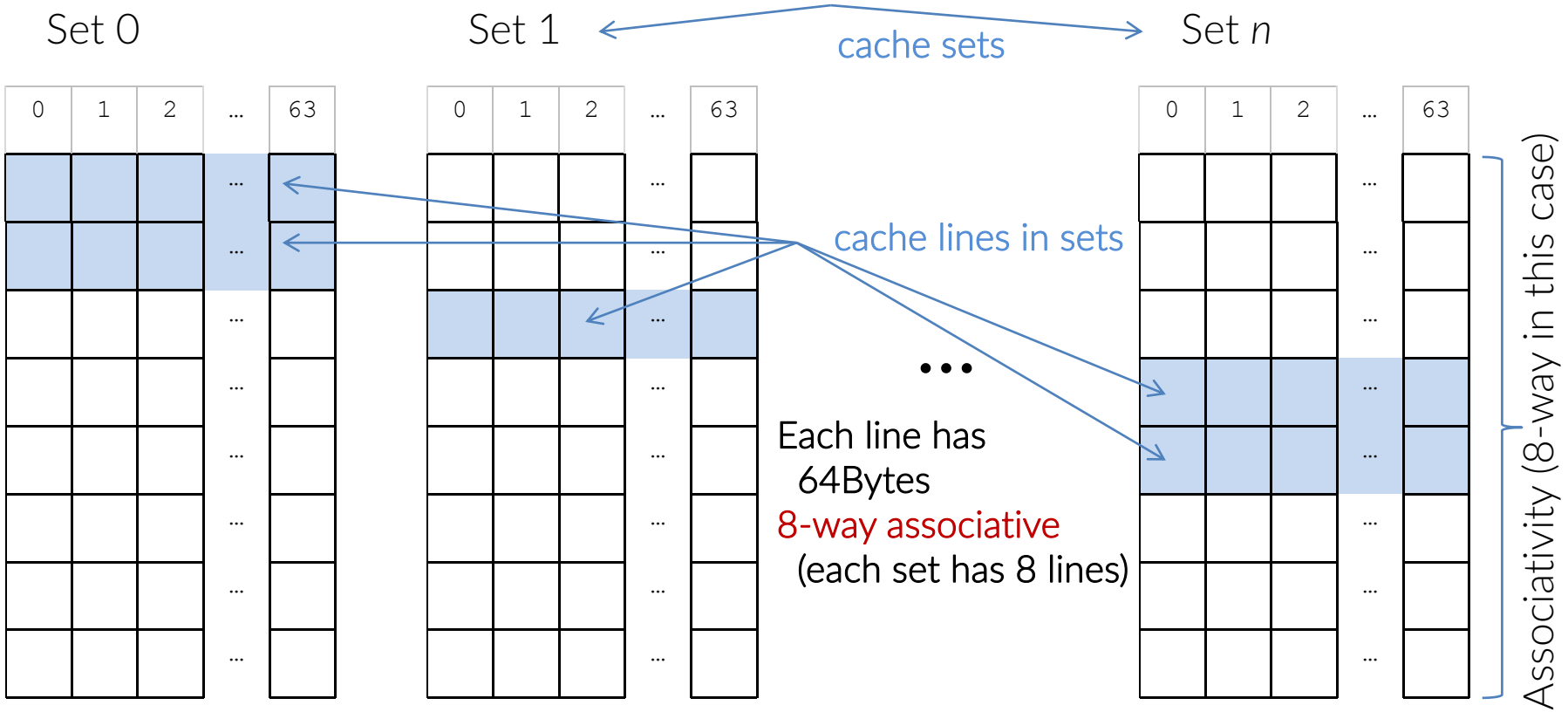
Cons
Maximizes the cache conflicts, for only 1 location is eligible for a vast amount of addresses.

n-way associative

Data can be placed in few cache blocks i.e.
A good trade-off

Pros
Very efficient in determining the set (only one per memory address).
Only a moderate amount of cache conflicts.

A typical today cache



Memory→ Cache mapping

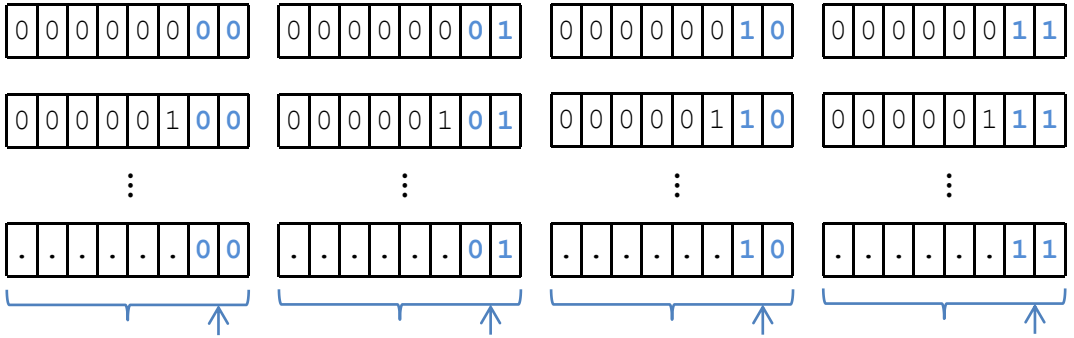
How a byte is actually mapped into a cache location ?

The “elemental” unit of the cache is a line, which normally today ha 64 bytes.
Then, 64 subsequent bytes in the RAM are mapped in the same line of he cache.

You can achieve that if the *position of the byte* in the target cache line is determined by the least significant bits of its address. Those bits are the fastest changing and, of course, they “cycle”.

Let’s build up an example →

Let’s suppose, for simplicity, that we have only 256Bytes of memory.
Then 8 bits, are sufficient to address our memory:



These bytes will always be at pos 0 in a cache line

These bytes will always be at pos 1 in a cache line

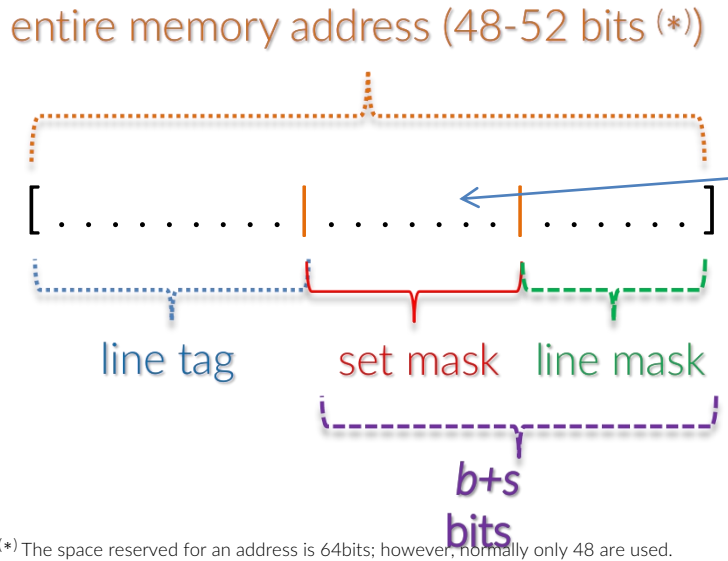
These bytes will always be at pos 2 in a cache line

These bytes will always be at pos 3 in a cache line

If you consider the first *n* bits, will have a cycle of 2^n bits; if *n=6*, the position will have a cycle of 64.
So, the least significant bits decides the position of a given address in a cache line, by group of 64.
But *what line exactly ?*

Memory → Cache mapping

How a byte is actually mapped into a cache location ?



(*) The space reserved for an address is 64bits; however, normally only 48 are used. In some architectures up to 52 bits are used to address memory.

If your cache size has C bytes in total and it is w-way associative with lines of size L bytes with

$$L = 64B = 2^b \text{ bytes}, \; w = 8, \; C = 32KB$$

then there are $C/(L \times w) = 128 = 2^s$ sets (where $s = 7$).

To map the memory addresses into the cache, the first $s+b$ bits are used to determine to what byte in a free line of what cache set that address will be stored.
The first b bits determines the position of the byte in the line, the second s bits determine ***uniquely*** the set to which that line belongs:

$$set = \texttt{set\_mask\_value} \; \% \; 2^s$$

- the set masking is like a “direct mapping”: 1-to-1 between a byte and a set
- the fact that each set has 8 possible lines is like the “fully associative” mapping, i.e. it gives you enough flexibility to cope with conflicts.

Eventually, to allow the recovering of the full address of a line stored in a cache line the final bits of the address are copied in a location, called *tag*, aside the line:

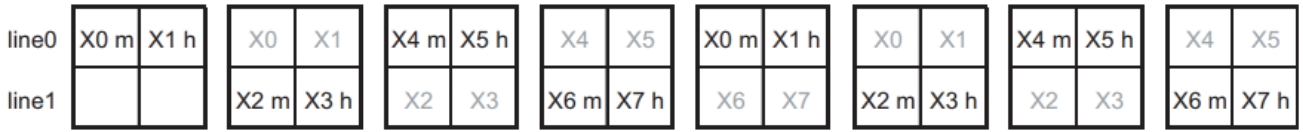
The memory access pattern

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 memory access pattern

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

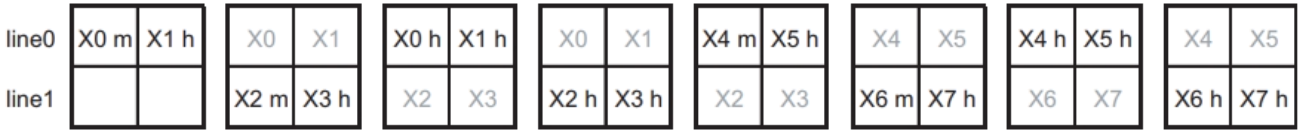
```
float X[8];
for(int j=0; j<2; j++)
   {
      for(int i=0; i<7; i+=2)
         access(X[i]);
      for(int i=1; i<8; i+=2)
         access(X[i]);
   }
```

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

The memory access pattern

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

```
for(int k = 0; k < 2; k++)
   for(int i = 0; i < 2; i++)
      for(int j = 4*k; j < (k+1)*4; 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

Cache recap in two slides

3C for the foes

- **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

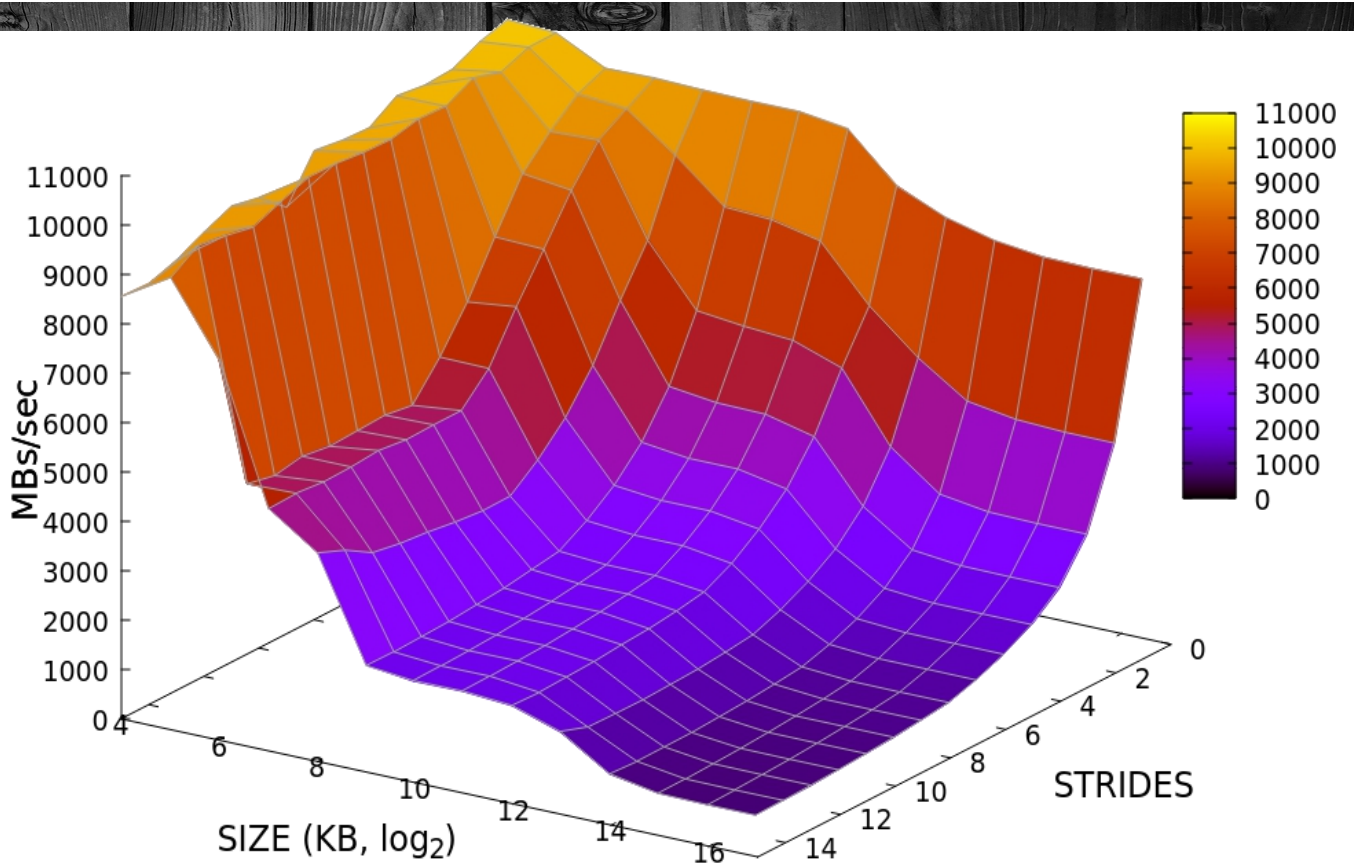
Cache recap in two slides

3R for the friends

- **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

The memory access pattern

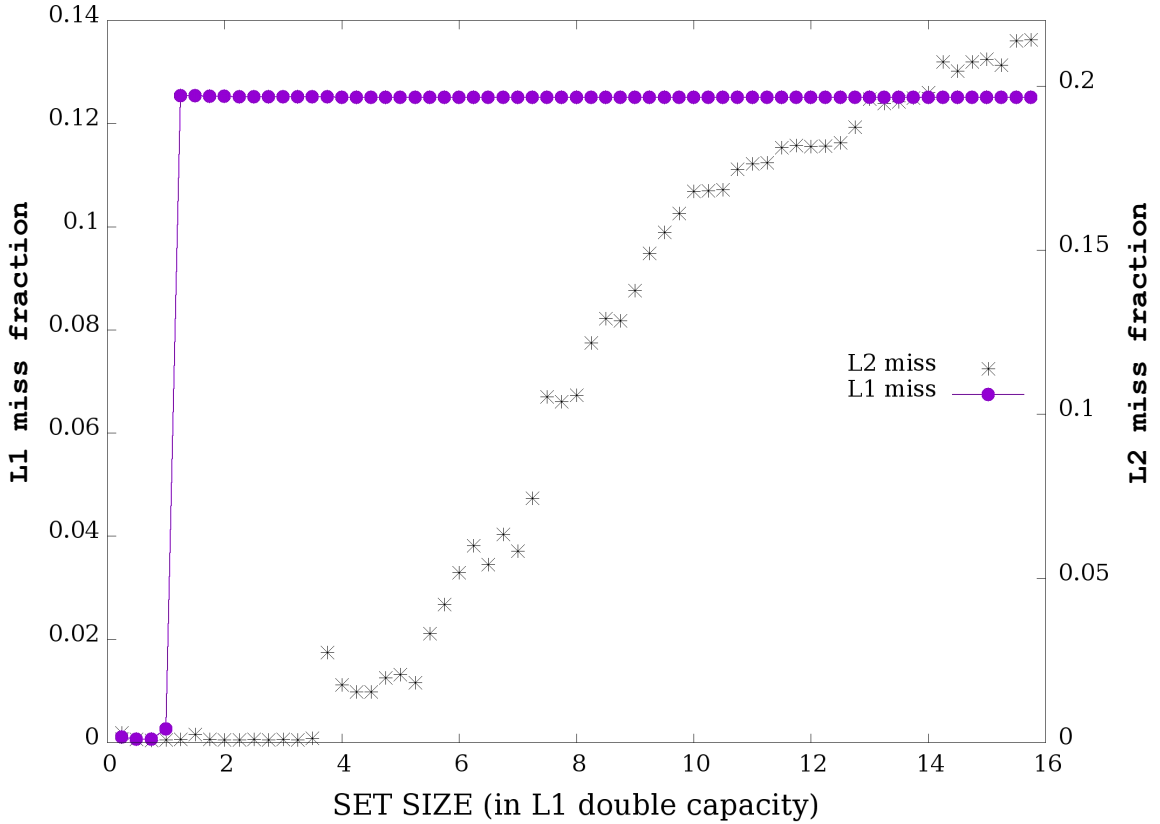
The result is..
the memory mountain



The cache-miss signature

Let's find out our cache size

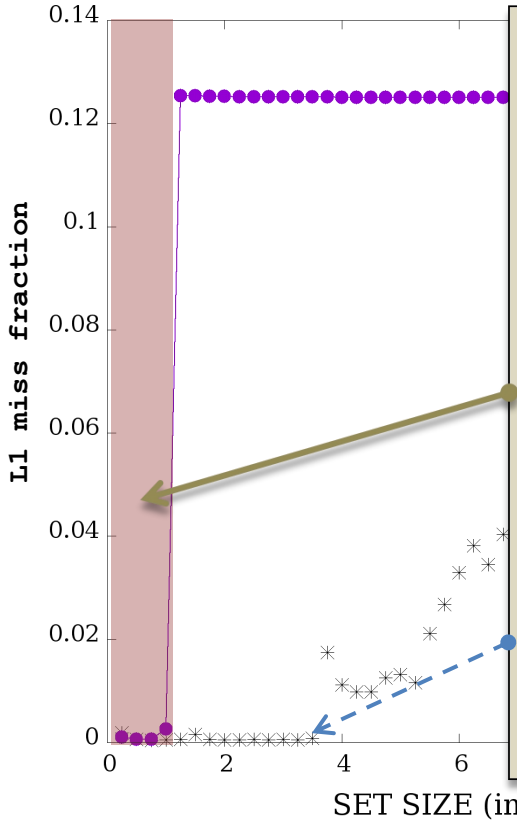
```
for (j=0; j < size; j++)
   array[j] =
     2.3*array[j]+1.2;
```



The cache-miss signature

Let's find out our cache size

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



Here the array has a size that is smaller than the L1, i.e. the array is entirely contained in the cache.

Then, there are basically no misses.

That is because the array was entirely load in the L1 in the warm-up phase (line 84 of cache_size.c), otherwise of course we would have obtained 1 compulsory miss at the begin of each line).

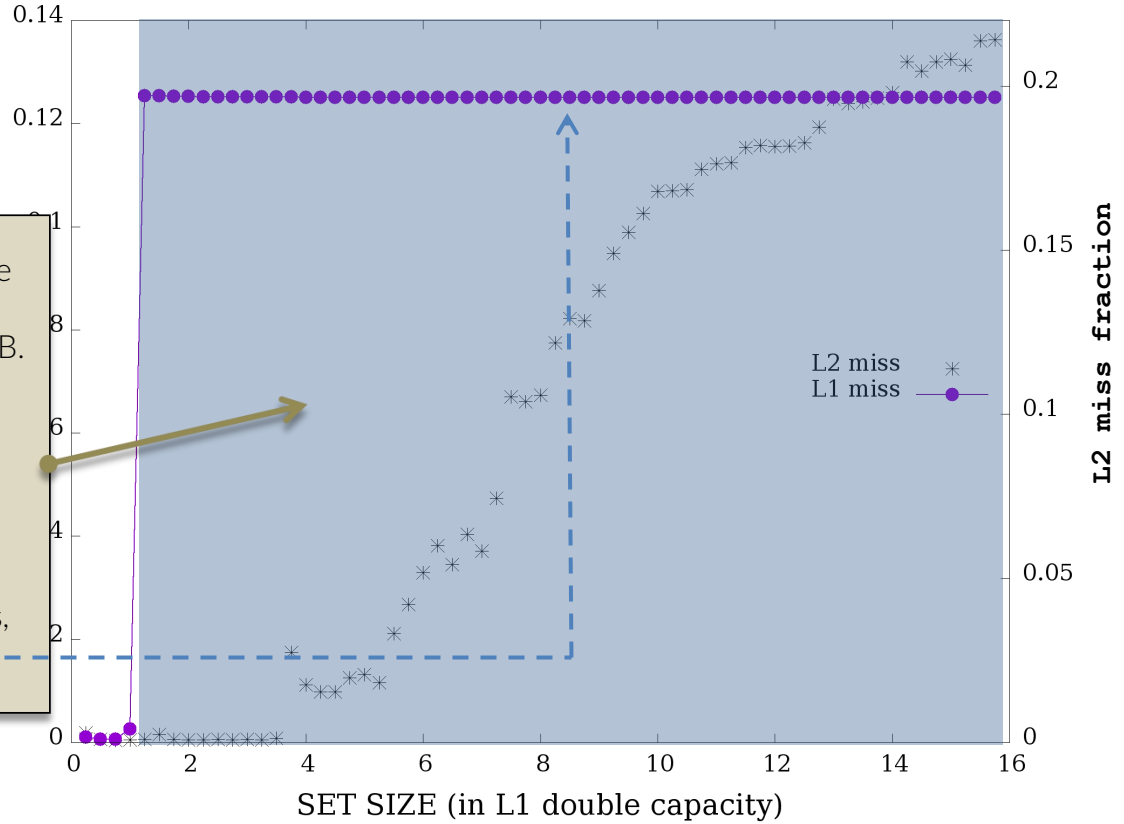
The same holds for L2, which has a size 4 times larger than L1.

The cache-miss signature

Here the array has a size that is larger than the L1, and so it can not be contained within it. The warm-up phase can only warm-up the TLB.

Then, every 8 double, the first access to the subsequent group of 8 doubl causes a compulsory miss *and* the upload of all the 8 double into a cache line.

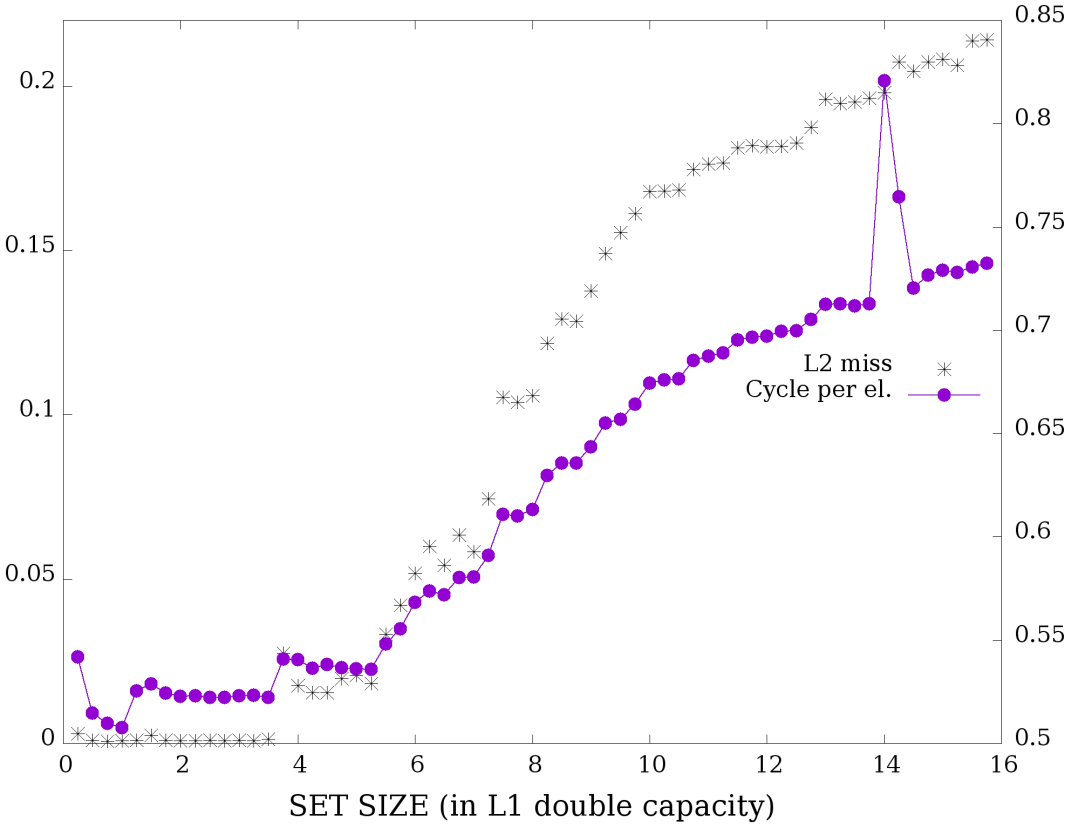
Hence, 1 every 8 accesses results in a L1 miss, i.e. a ratio of 12.5%.



The cache-miss signature

And the effect on cycles-per-operation metrics

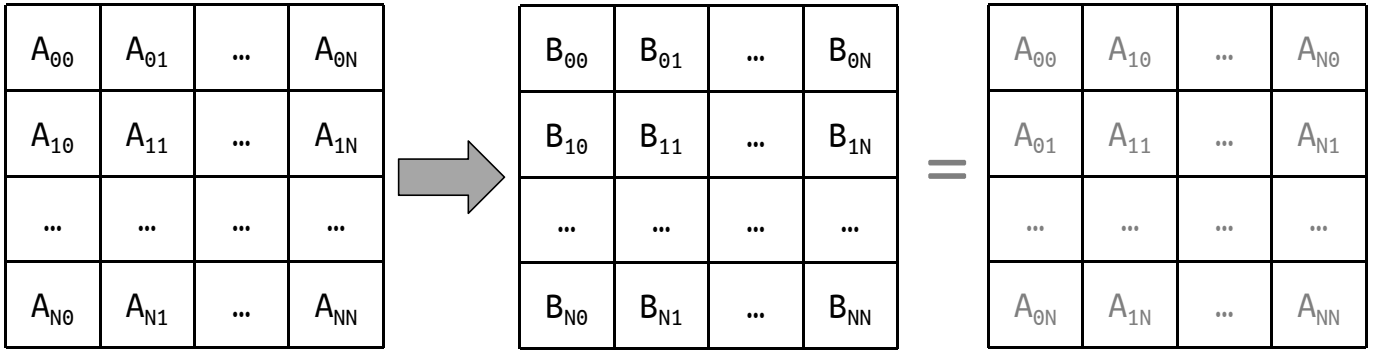
```
for (j=0; j<size; j++)
   array[j] =
     2.3*array[j]+1.2;
```



Strided access

Let's consider a quite common problem: the transpose of a matrix.

Matrix transpose



$$A^T_{ij} = A_{ji}$$

Strided access

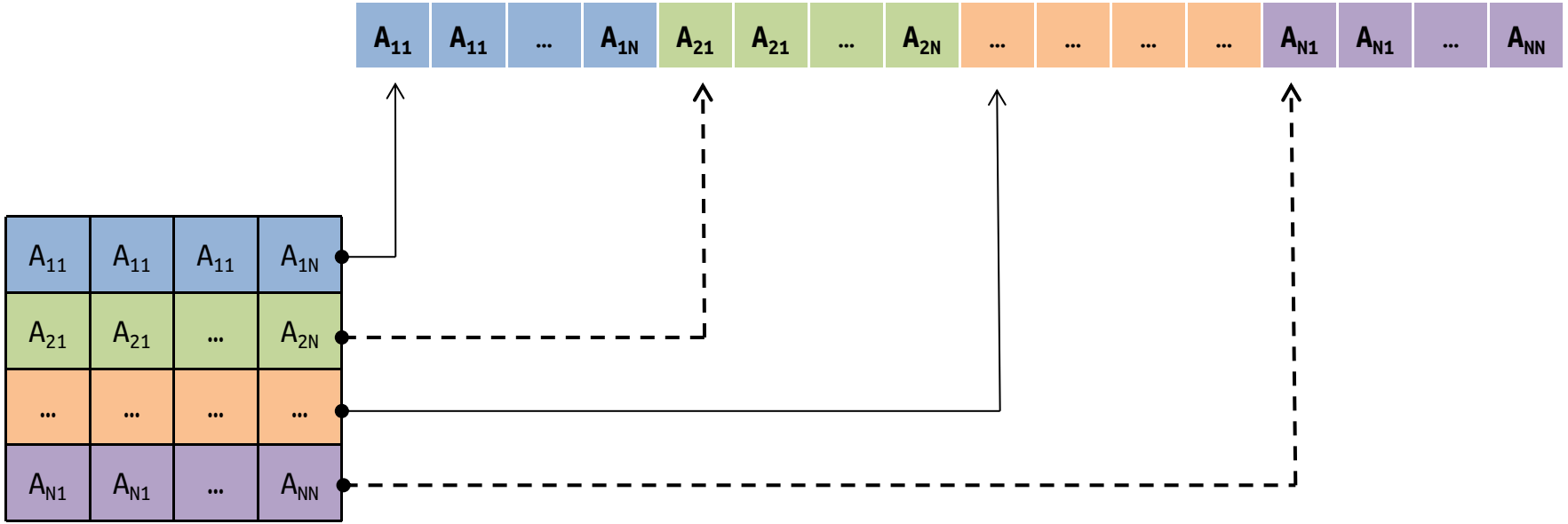
Matrix transpose

The very simple, straightforward implementation is:

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

Note: how a matrix is stored in memory

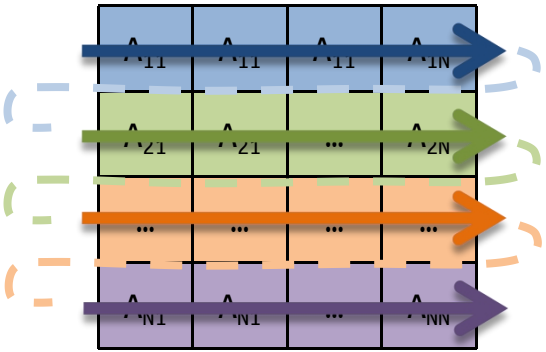
Remember the obvious fact that your memory is a continuous 1-dimensional stream of bytes.



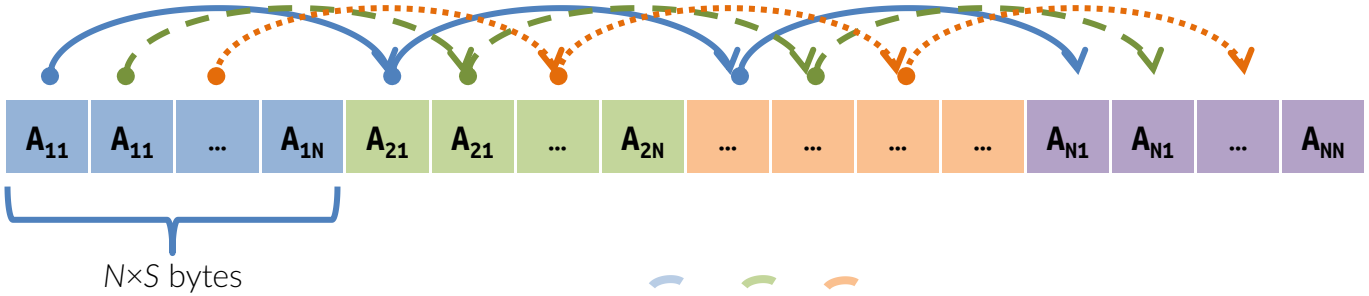
This convention is the C/C++ convention, which is labelled as *row-major order*. Note that the Fortran convention is opposite, with columns being contigous in memory (*column-major*).

Note: how a matrix is stored in memory

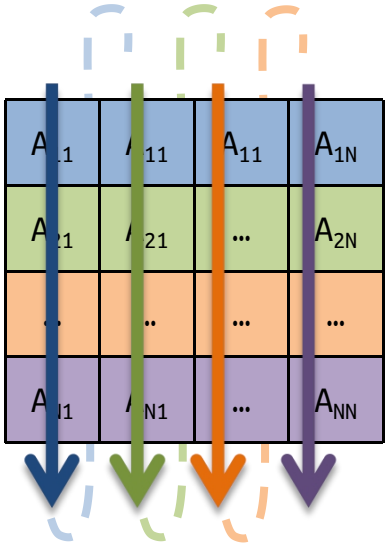
Then, traversing the matrix in the same row-major order amounts to traverse the memory in contiguous order.



Note: how a matrix is stored in memory



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



Strided access

Matrix transpose

The very simple, straightforward implementation is:

```
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* ?

Strided access

Matrix transpose

Naïve version:

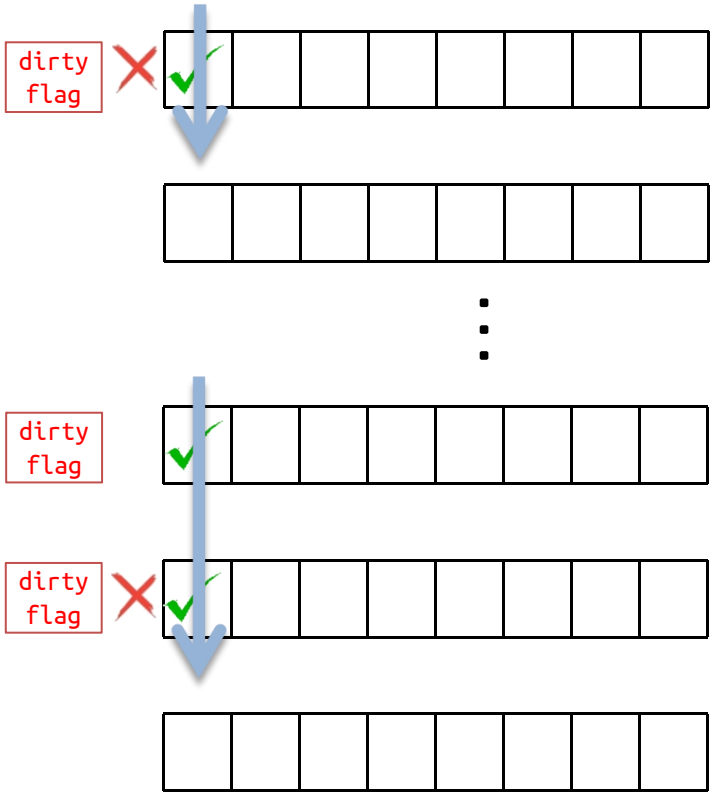
```
for(int row = 0; row < N; row++)
    for(col = 0; col < N; col++)
        A [ col*N + row ] = B [ row*N + col ];
```

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.

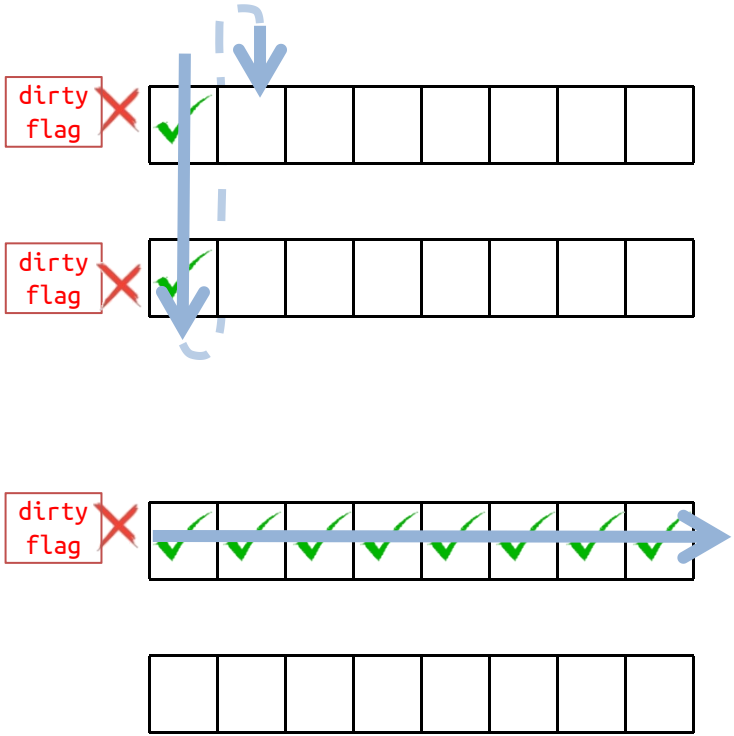
Strided access



As we have seen, as you traverse in column-major order, you are traversing also the memory in the in the same way; since we are considering a non-trivial case in which the matrix does not fit in the cache, this means that accessing the subsequent elements of a column you are accessing different lines in a cache.

When the content of the corresponding entry of the transposed matrix is written, the entire cache line is flagged as “dirty”. To enforce the memory-cache coherency, the line must then be flushed back in memory.

Strided access



Then, when you are back again on the first line, which is flagged dirty, and you continue the col-major on the next column. At this point, before you can access that location, the line is written back in memory and refreshed

If you wrote in row-major, instead, you would write an entire line of cache that is afterwards entirely written back into memory.
This greatly reduces the cache-memory transactions.

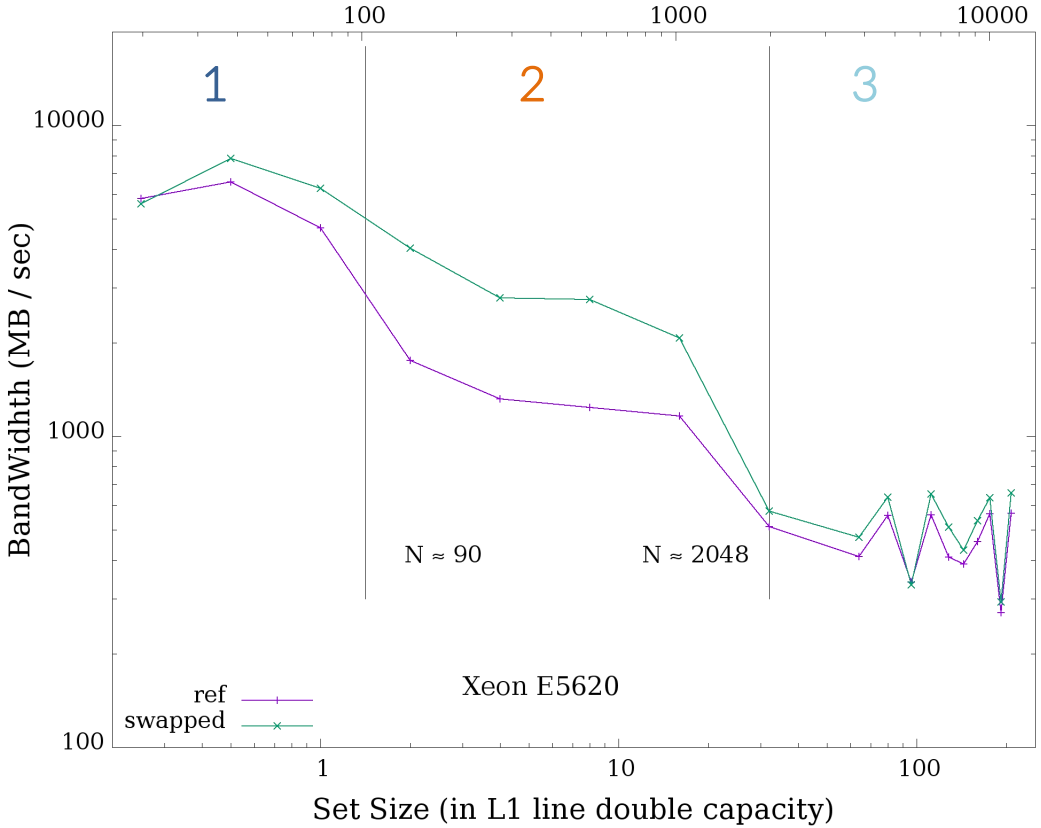
Strided access

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.



Strided access

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

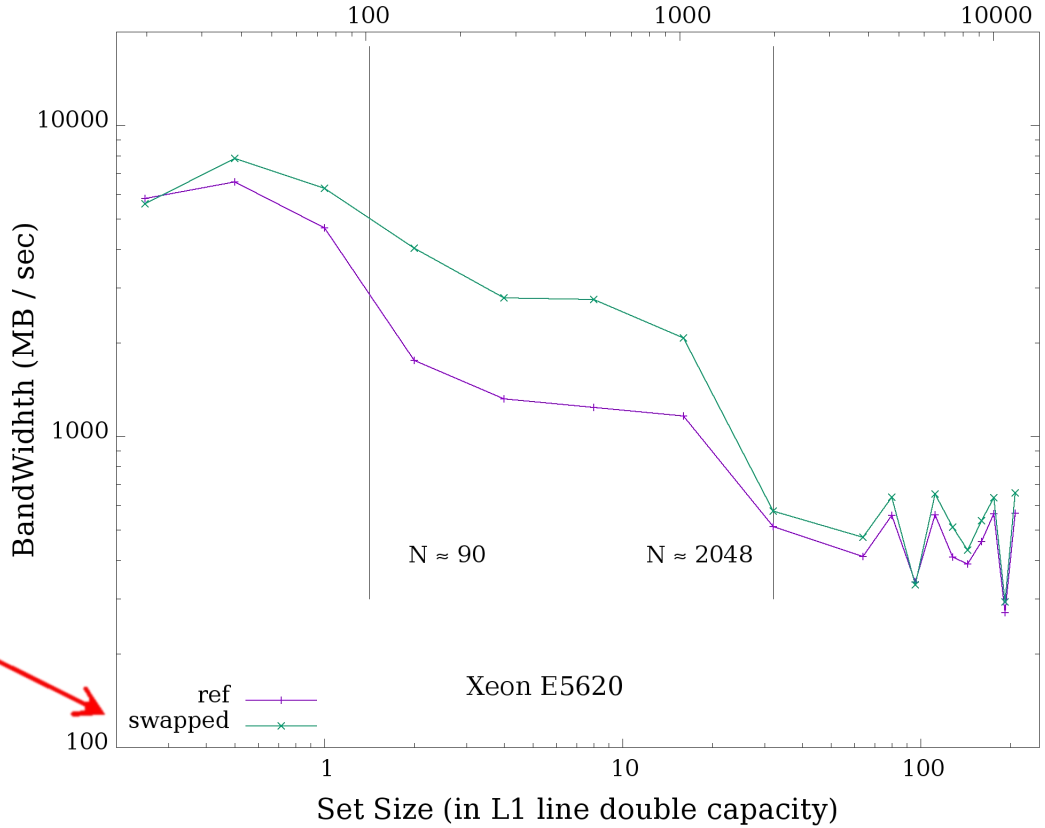
1. $2 \times N^2 < C$
   both matrices can fit in the cache,

Inner cycle flipped:

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

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.

Due to time constraints, we can't go in deeper details. Ask if interested, though.



Strided access

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

1. $2 \times$ both trave impa maxi

2. N $\times$ strid colu

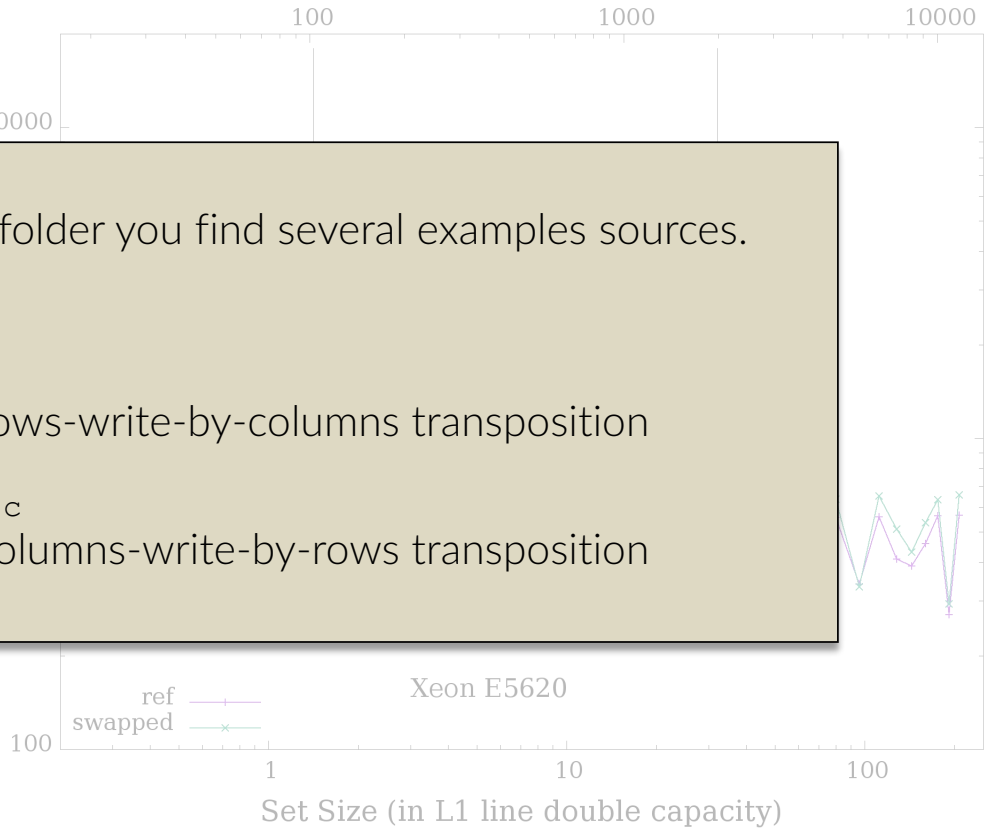
3. N > Each miss
   A sharp drop in performance is expected since basically only one entry per line will be used.

Due to time constraints, we can't go in deeper details. Ask if interested, though.

In the `day6/matrix_transpose` folder you find several examples sources. Among the others:

`matrix_transpose.c`
implements the read-by-rows-write-by-columns transposition

`matrix_transpose_swapped.c`
implements the read-by-columns-write-by-rows transposition



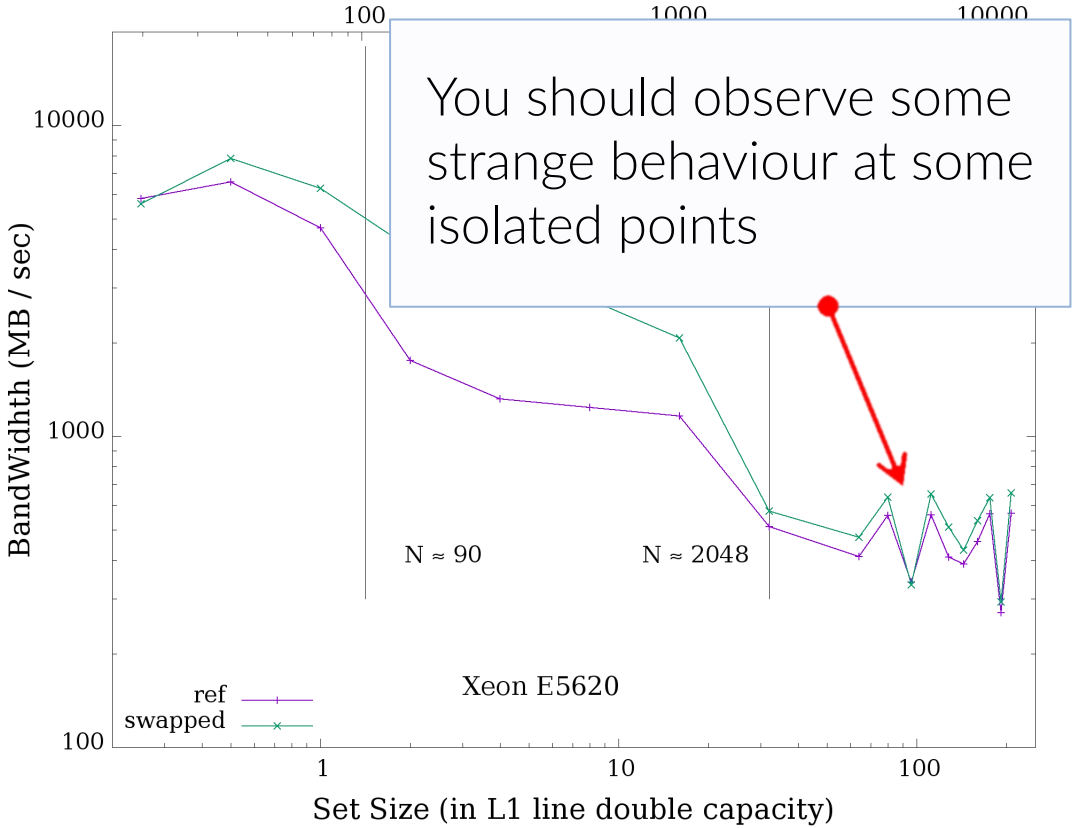
Cache-associativity conflicts

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)

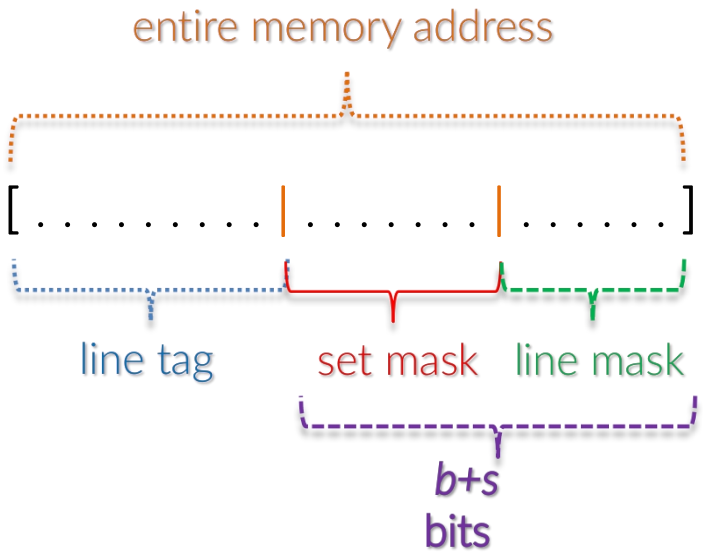
2. $N \times L_C < C$
   strided write is alleviated by fraction of column fitting in the cache

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.



Cache-associativity conflicts

Cache associativity conflicts ("cache resonance")



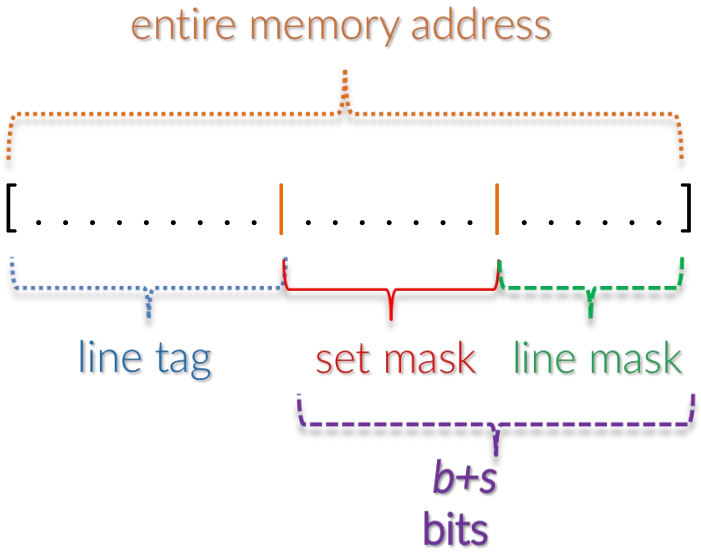
You know that your cache of size C bytes is w-way associative: it means that your cache is made up by lines of size L bytes, grouped in w-sized sets.

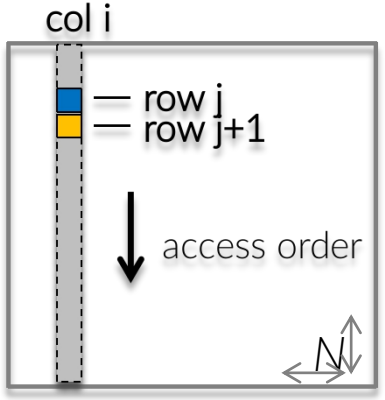
Tipical figures nowadays are: L = 64B (=2^b bytes), w = 4-8, C = 32KB (i.e. there are $C/(L \times w)$ = 64-128 (= 2^s sets).

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-associativity conflicts

Cache associativity conflicts ("cache resonance")





Matrix ^{T}

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).

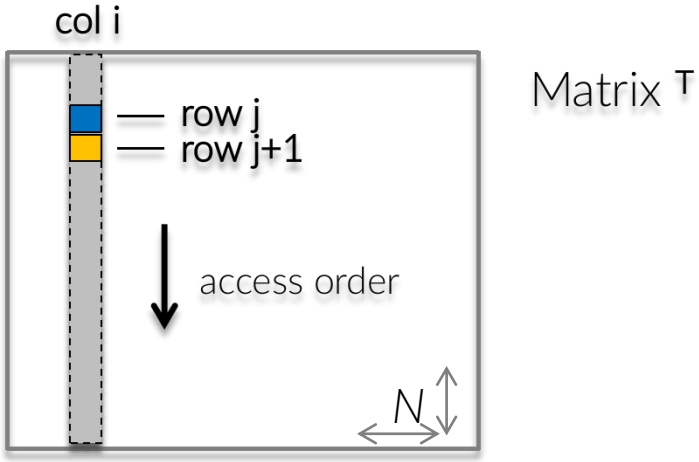
If N is a power of 2, $N = 2^n$; if $W = 2^d$ bytes.

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

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-associativity conflicts

Cache associativity conflicts ("cache resonance")



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

Adding one column to the transposed matrix, which then would be $(N+k) \times N$, 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,

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

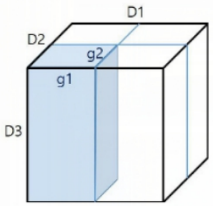
then if

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

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

Cache-associativity conflicts

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* $D_i \prod_{1\leq k\leq i-1} g_k$.

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

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 possibly very good return-on-investment)

In the `day6/matrix_transpose` folder you find also:

`matrix_transpose_padded.c`
implements `matrix_transpose.c` with padding

`matrix_transpose_swapped_padded.c`
implements `matrix_transpose_swapped.c` with padding

Main message about cache

How you place the data in the memory and how you access them is of paramount importance.

This impacts directly on the *data model* that you choose and implement, and on how you design your workflow.

We'll see more details in the next lectures, when we'll talk about pipelines and vectorization.

Hot & cold fields

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

Linked-list node

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

Linked-list traversal

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

Hot & cold fields

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

Since we are looking for a unique node in the list, the one that has the *key* we are looking for, all but one nodes are discarded in our search.

Then, the usual execution pattern is

```
while( p != NULL) {
        if( p->key == key ) {}
        p = p    next_node; }
```

Or, in other words, the **key** and **next_node** fields are temporary local, in that they are accessed one after the other.

Linked-list traversal

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

Hot & cold fields

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

However, there are 300 bytes in between **key** and **next_node**, which is not an optimal way of organizing the data because for sure they are not spatially local neither in memory nor in cache, while they are temporally local.

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

Hot & cold fields

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];
}
```

A first move is simply to swap the position of `my_data` and `next_node`.
That is a simple example, to clarify what it means to optimize the data layout for cache locality

Hot & cold fields

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

Still, the data bunch **my_data** is the real breaker.

A most significant move, is to separate the *metadata* **key** and **next_node** from the *data* **my_data** that are accessed only once the search in the linked list is successful.

During the traversal of the linked list, **key** and **next_node** fields are the *hot* fields, while the **my_data** is a *cold* field.

A good strategy in general, is to keep the hot fields (i.e. data temporally close) spatially close together as much as possible.

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

Hot & cold fields

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

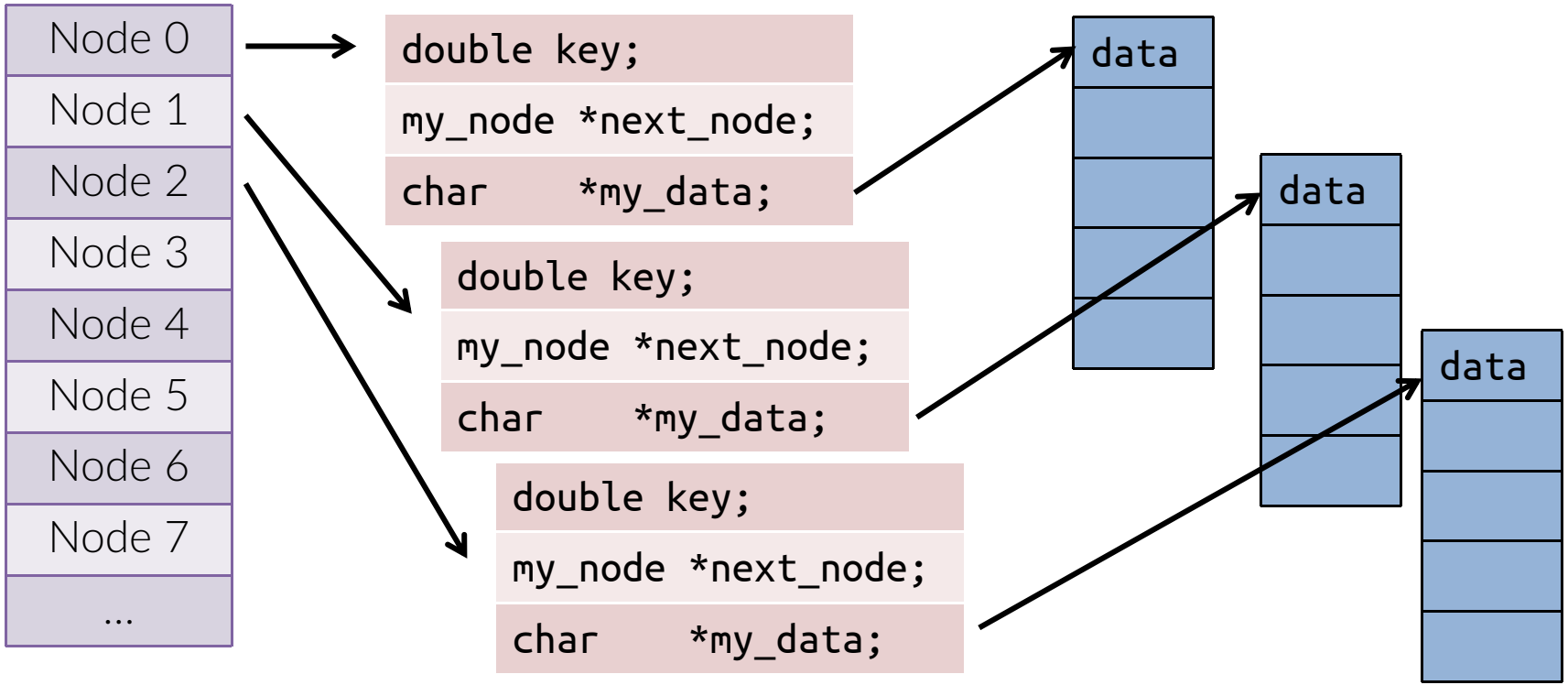
```
struct my_node
{
  double   key;
  char
my_data[300];
  my_node *next_node;
}
```

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

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

Hot & cold fields

Those are called *hot* and *cold* fields



Organizing data to enhance locality

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.

Organizing data to enhance locality

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

Organizing data to enhance locality

In the “space” the data live in – for instance our usual 3D space – there might be a metric that correlates with spatial coherence.
Generally, it is more probable to access in a short time lapse points that are also spatially close.

Then, two obvious strategies 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.

Organizing data to enhance locality

What is the “minimum” distance distortion ?

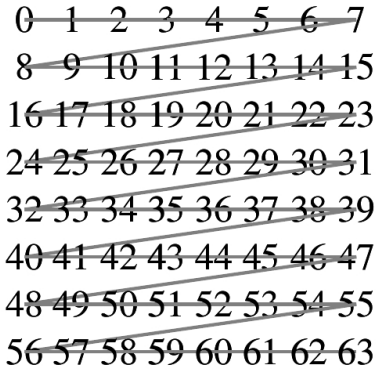
In 1-D the answer is trivial.

In 2-D the answer is less so, or not trivial at all, and it is increasingly less trivial as the number of dimensions grows.

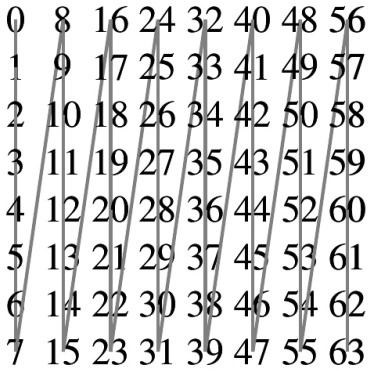
Organizing data to enhance locality

Scanline order *row-major*

(a)



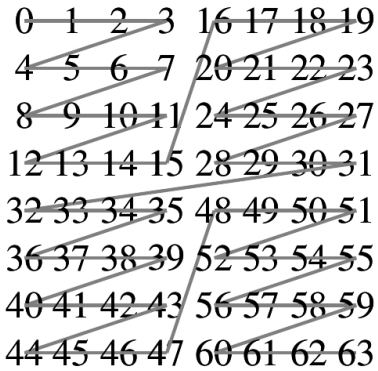
(b)



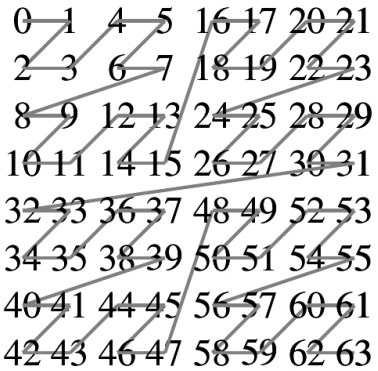
Scanline order *col-major*

Block order + scan sub-order

(c)



(d)



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

The z-Order

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

`0, 1, 2, 3, 4, 5, …, n`*`th`*

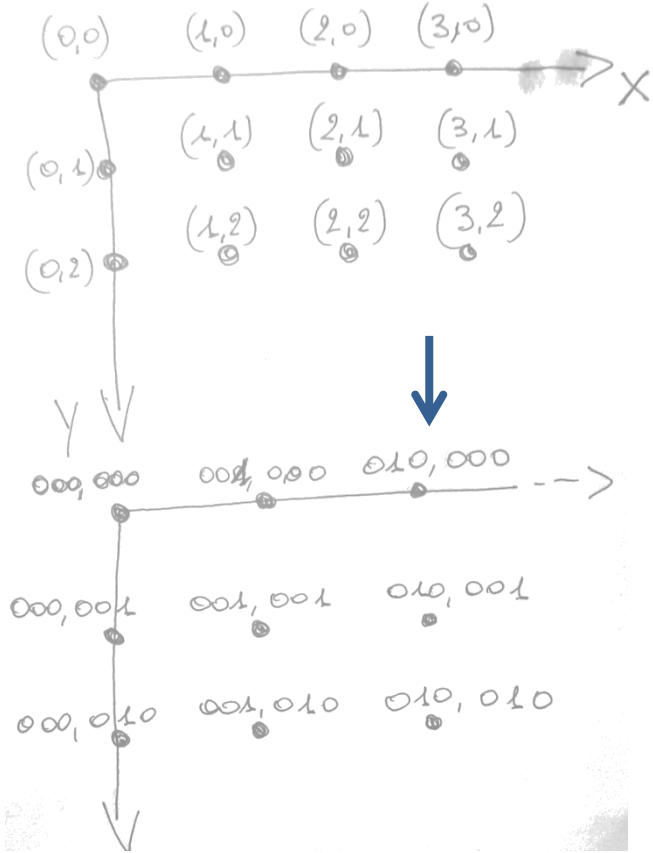
on some spatially-distributed data with integer coordinates.

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?

The z-Order



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

$x = x_i \dots x_2 x_1 x_0$

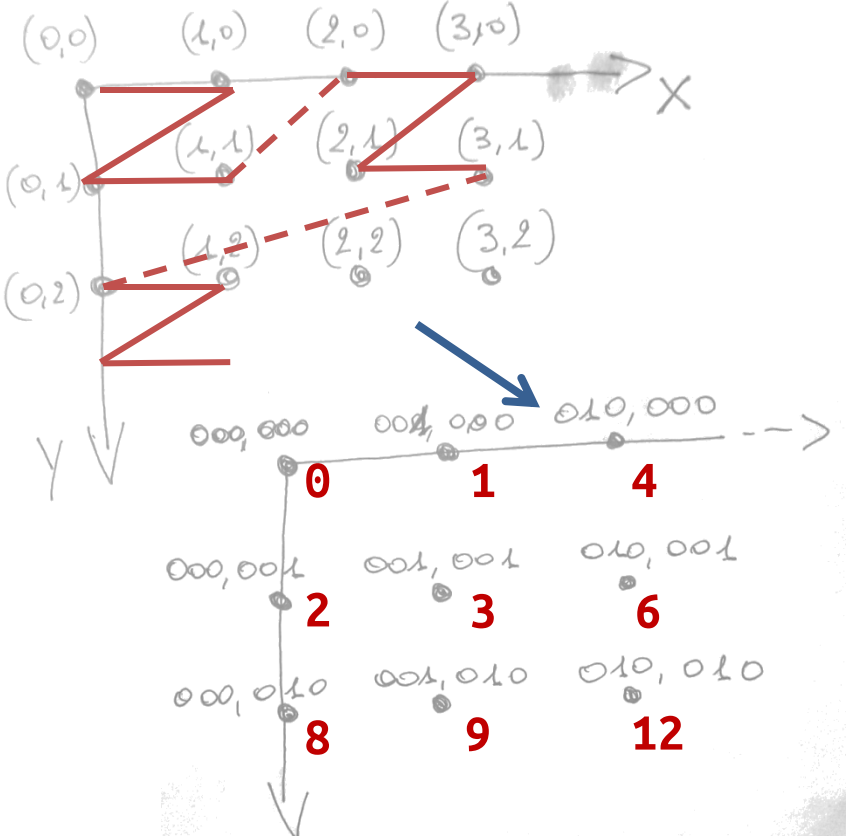
so that a couple *(x,y)* reads as:

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

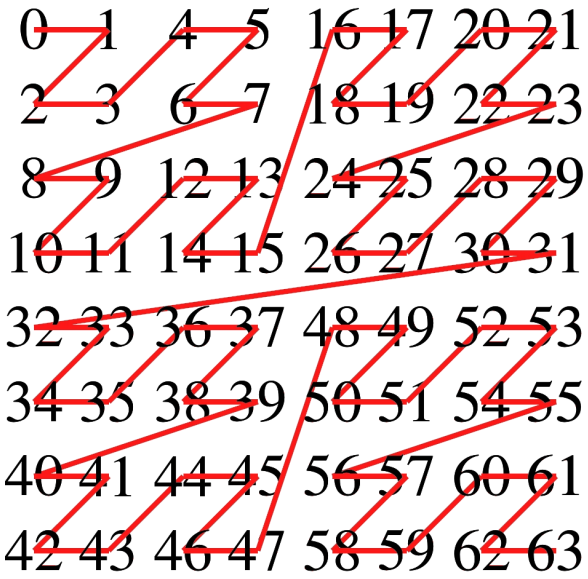
Then let's define the following reshuffle so to *interleave the bits*

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

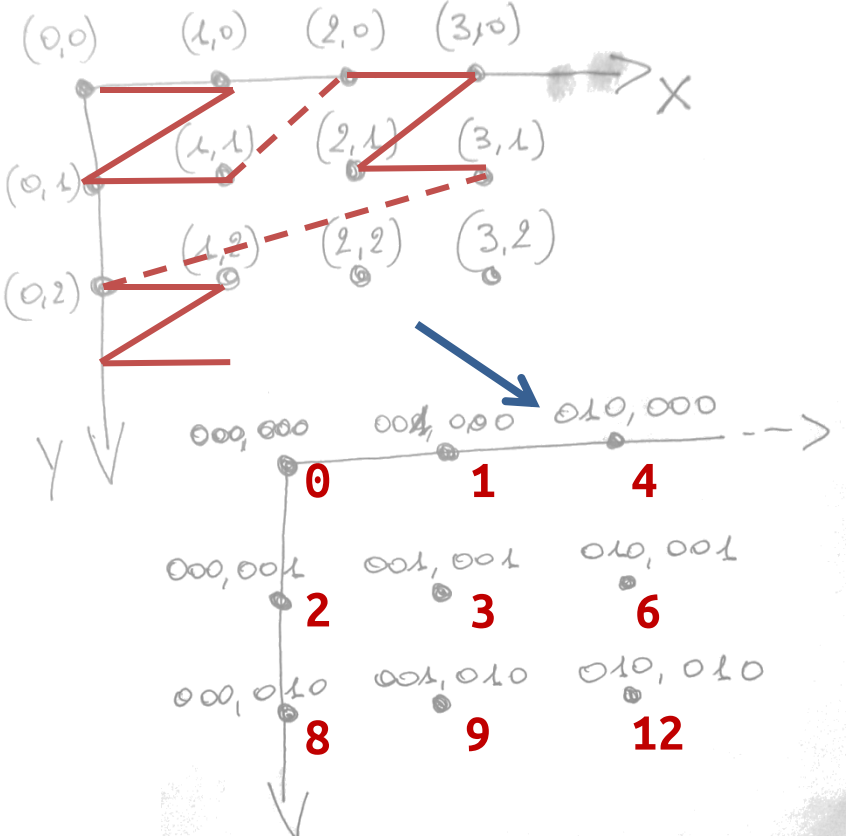
The z-Order



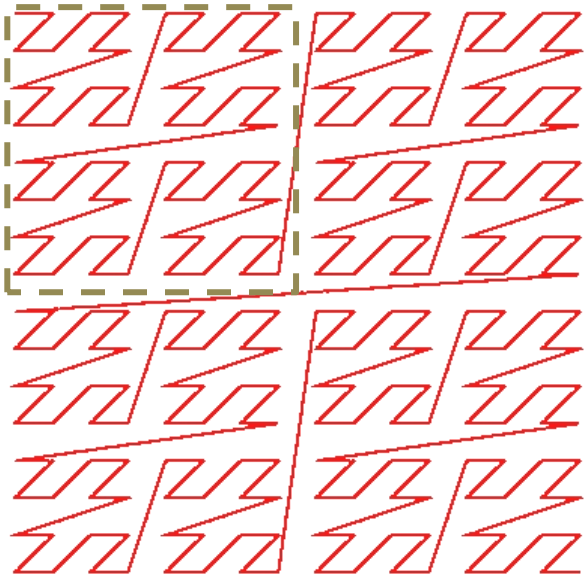
The result of this interleaving is a mapping from the 2-D plane to the 1-D line known as Z-line which is one of the **plane-filling curves** discovered by Peano



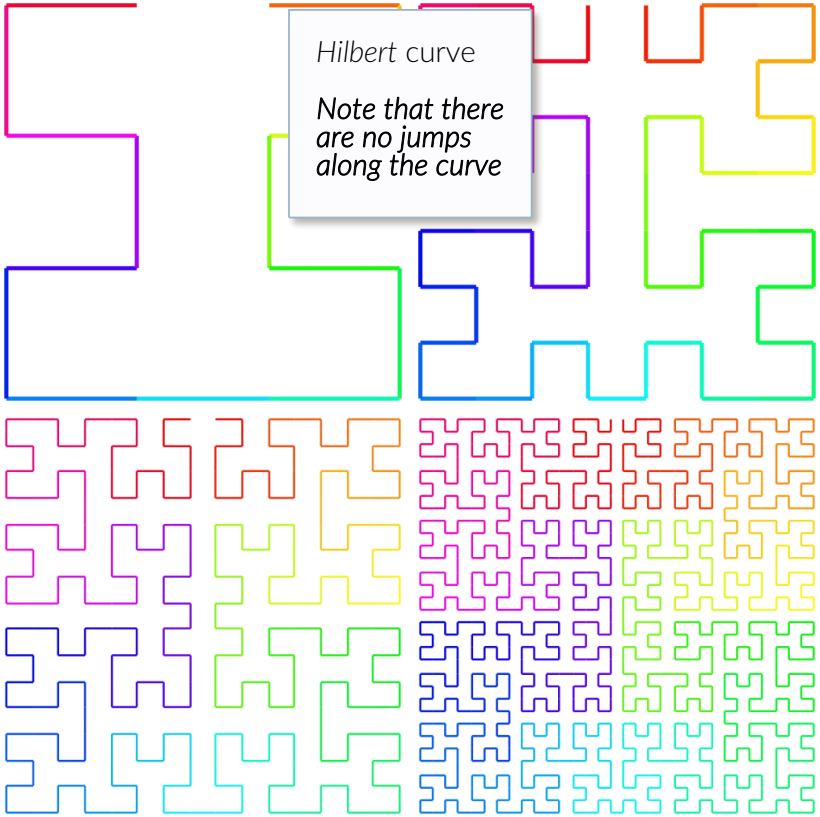
The z-Order

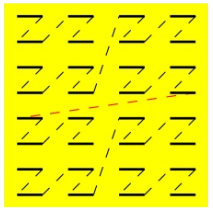


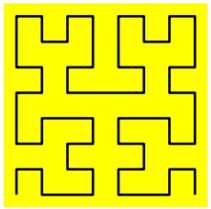
The result of this interleaving is a mapping from the 2-D plane to the 1-D line known as Z-line which is one of the **plane-filling curves** discovered by Peano

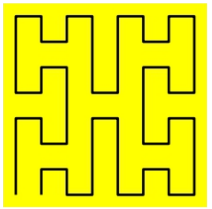


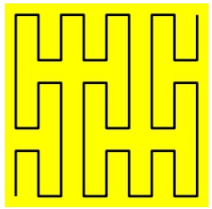
The space-filling curves

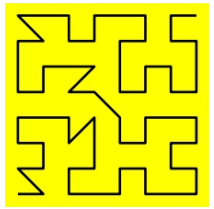


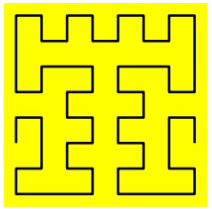

Z-order

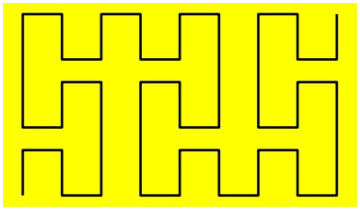

Hilbert curve


H-order


Peano's curve

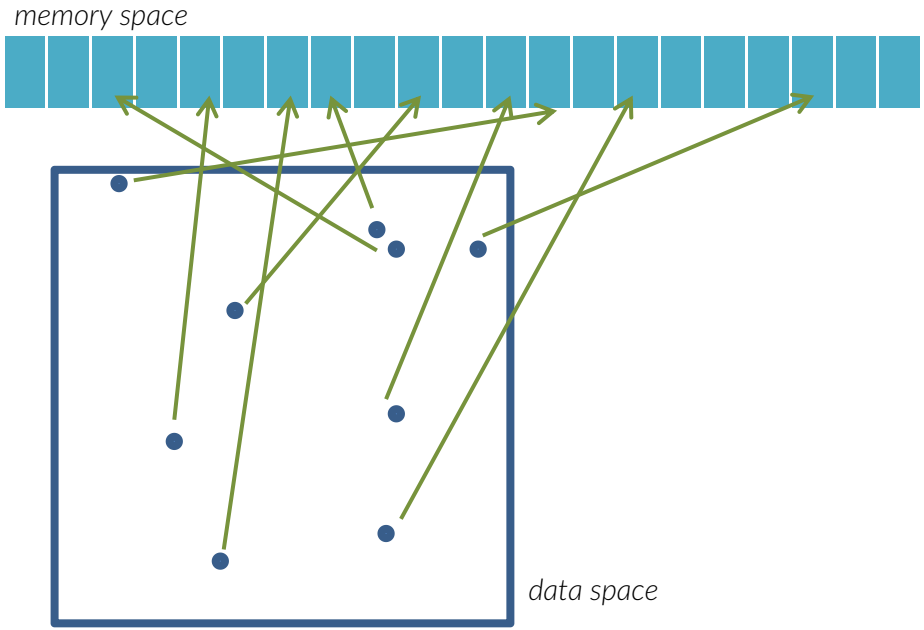

AR^2W^2-curve


$\beta\Omega$-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

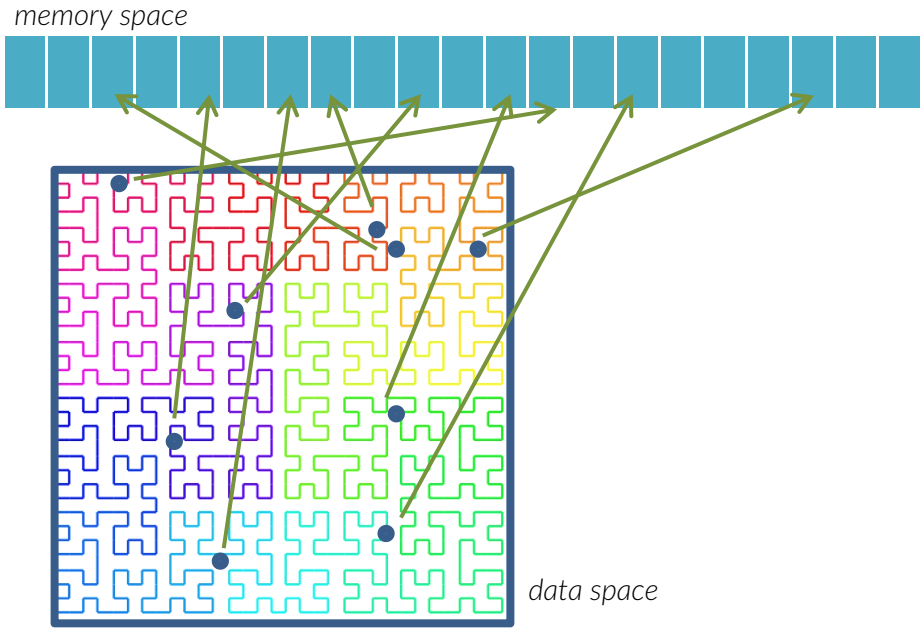
The space-filling curves



Let's say that your data space has 2D, like in the figure on the left, and that data live in memory in non particular order (meaning that there in no relation between the memory position and the position in data space.

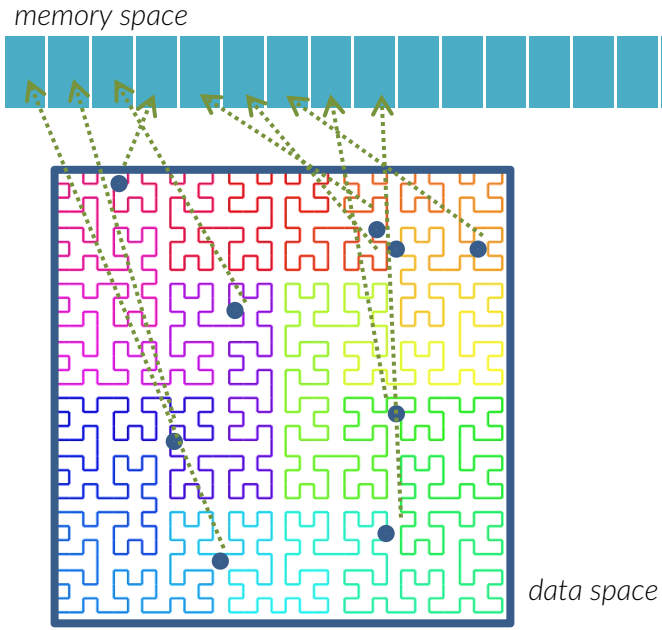
To re-order the data in memory so that to preserve their locality, a space-filling curve can be used.

The space-filling curves



STEP 1: calculate the 1D index of each data along the curve (in this case, a Peano-Hilbert curve)

The space-filling curves



STEP 1: calculate the 1D index of each data along the curve (in this case, a Peano-Hilbert curve).

STEP 2: sort data in memory according to their index.

The space-filling curves

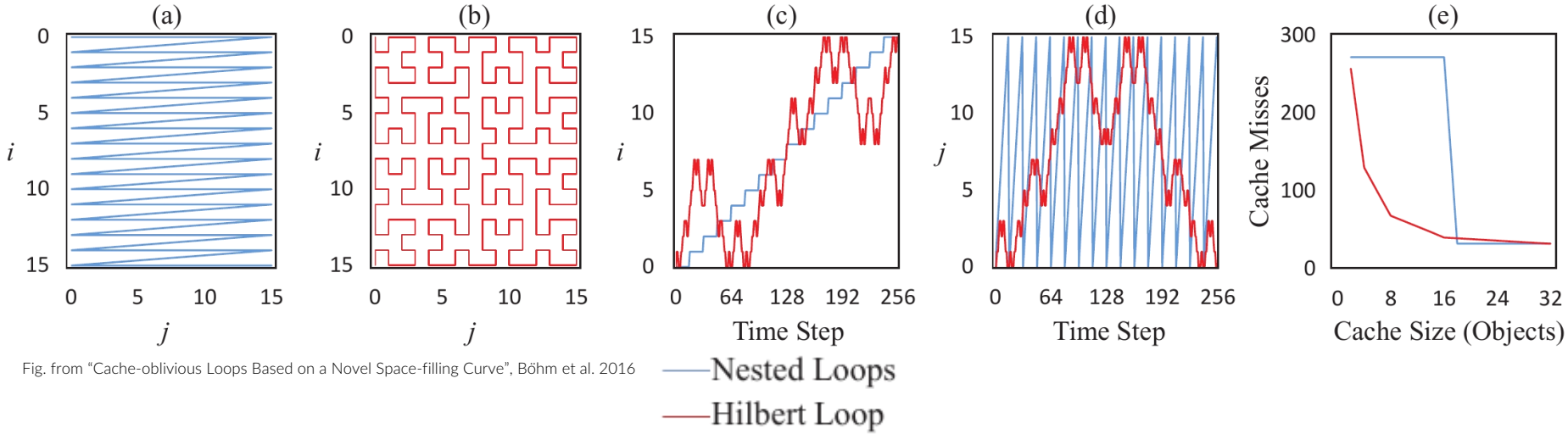


Fig. from "Cache-oblivious Loops Based on a Novel Space-filling Curve", Böhm et al. 2016

The space-filling curves can also be used to determine the traversal order of structured data; this plots report the traversal order for classic nested-loops and the Hilbert Curve.
Note the increased locality in *j* (panel *d*) and the highly reduced number of cache misses.

that’s all, have fun