Introduction to Single-Core Optimization

Luca Tornatore, I.N.A.F.

2025 INAF Course on HPC

Outline

Avoid the avoidable inefficiencies

Cache & Memory

Loops

Branches

Pipelines

Unleash the Compiler

Outline

Avoid the avoidable inefficiencies

Cache & Memory

Loops

Branches

Pipelines

Unleash the Compiler

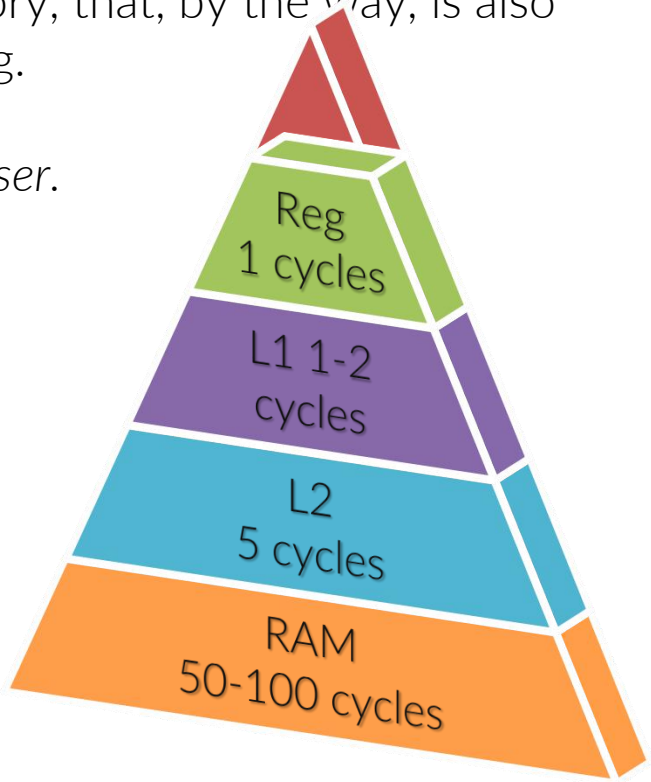
The cache memory

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.



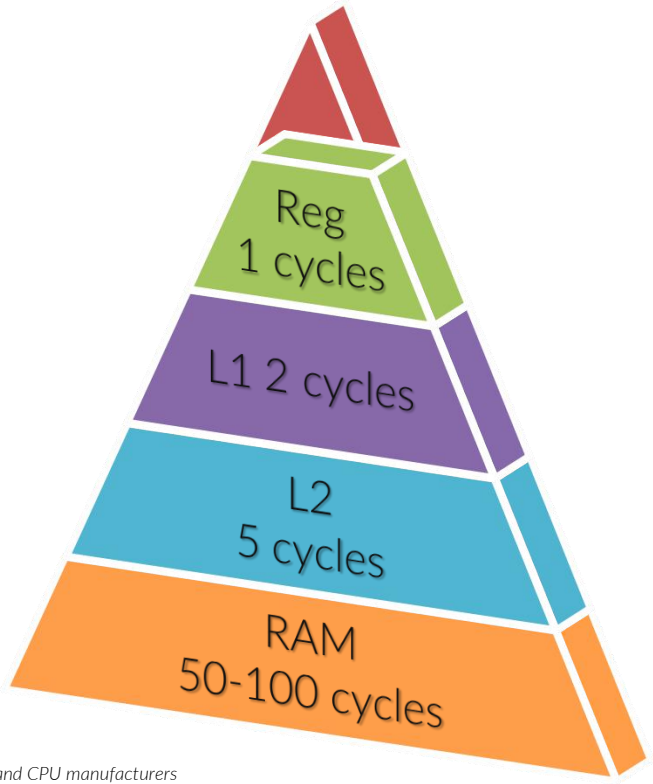
The cache memory

The cost of memory access

L1 cache + RAM

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

- 100% L1 hit → 1-2 cycles
- 99% L1 hit → 2.5 - 3
- 97% L1 hit → 3.5 - 5 cycles



These figures are not "exact", they are just representative. True ones depend on the details of the system, DDRx CAS latency and other characteristics, and CPU manufacturers

The cache memory

L1 cache + RAM

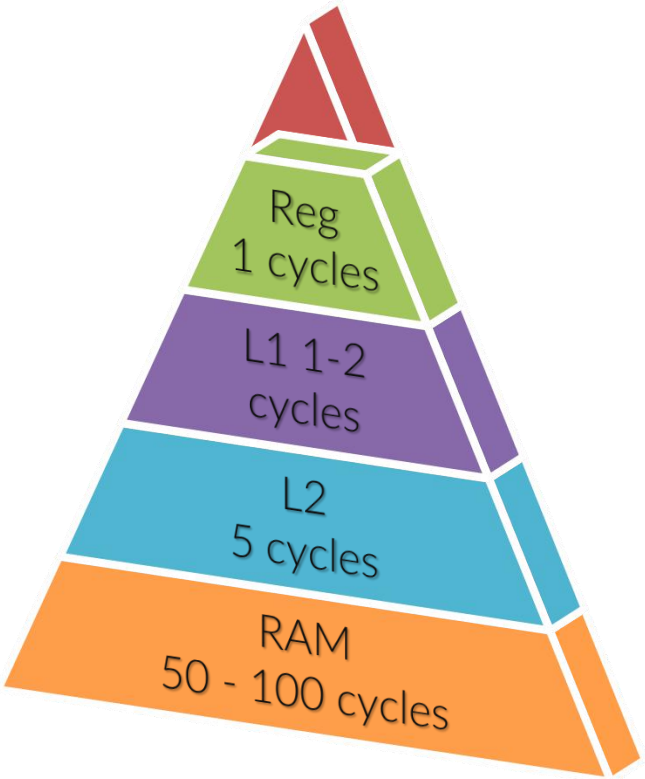
$L_{1cost} + Miss_1 \times RAM_{cost}$

- 100% L1 hit → 1-2 cycles
- 99% L1 hit → 2.5 - 3 cycles
- 97% L1 hit → 3.5 - 5 cycles

L1 cache + L2 cache + RAM

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

- 100% L1 hit → 1-2 cycles
- 99% L1 hit, 100% L2 hit → 1.04 - 2 cycles
- 97% L1 hit, 100% L2 hit → 1.12 - 2.1cycles
- 90% L1 hit, 97% L2 hit → 1.6 - 2.6 cycles



These figures are not "exact", they are just representative. True ones depend on the details of the system, DDRx CAS latency and other characteristics, and CPU manufacturers

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

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

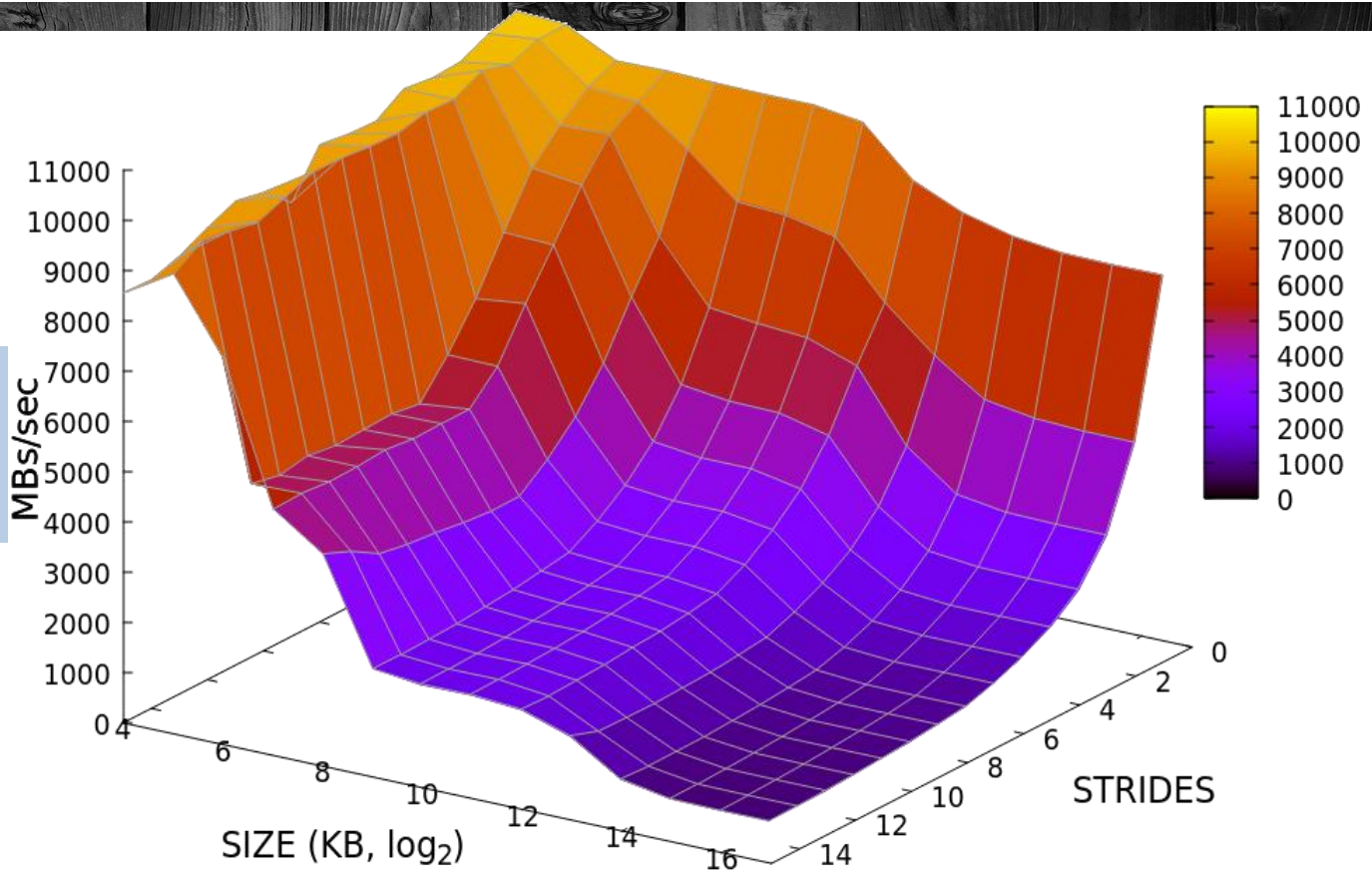
Cache recap in two slides

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



Just a preview;
more details in the next lecture

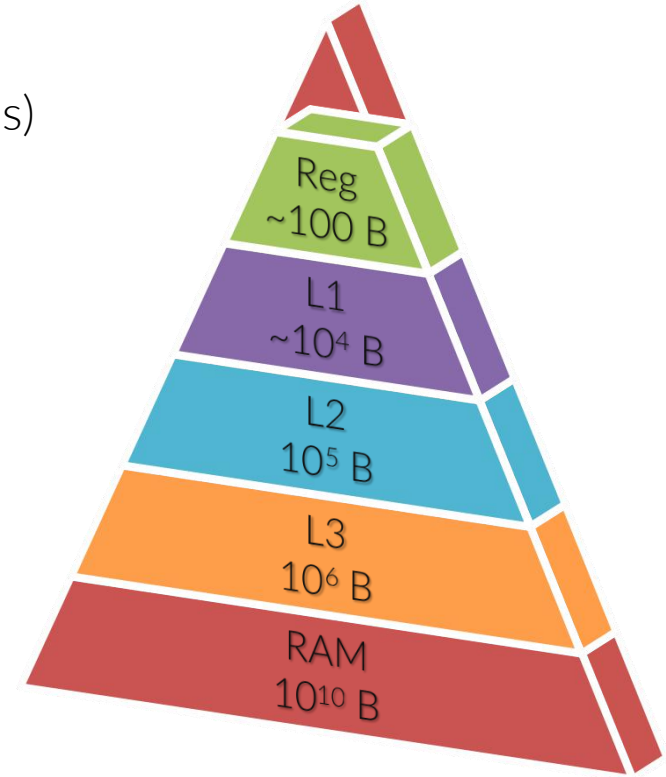
Cache mapping

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:

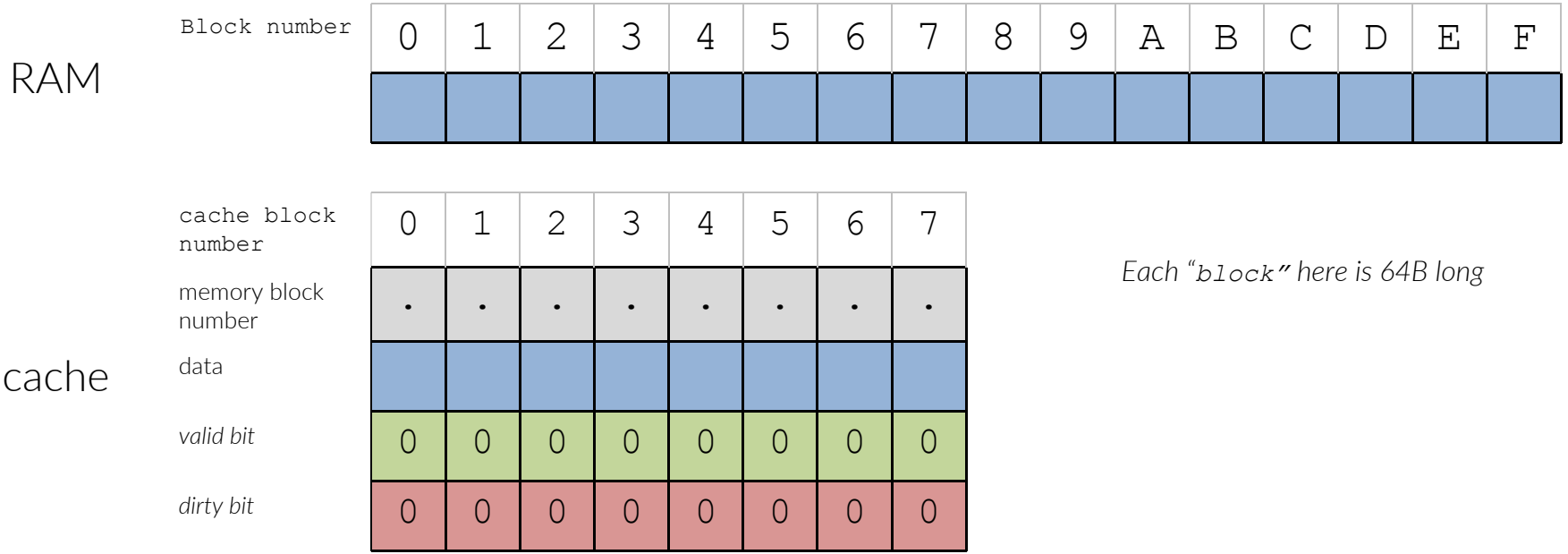
- *Where* to map an address
- *What if* the location in L1 is already occupied?

…we'll see some details



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*)



Each "`block`" here is 64B long

Memory→Cache mapping

Full mapping

Data can be placed in any free cache block

Direct mapping

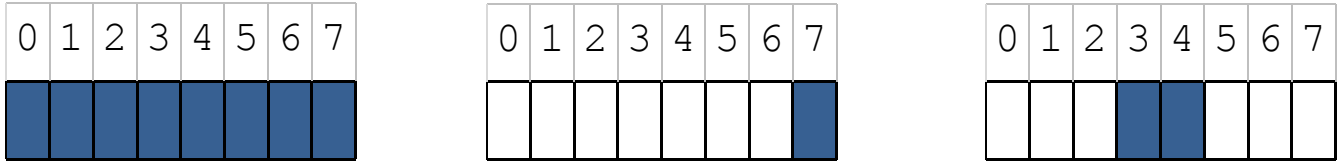
Data can be placed only in a given block i.e. **`block_num % blocks_in_cache`**

n-way associative

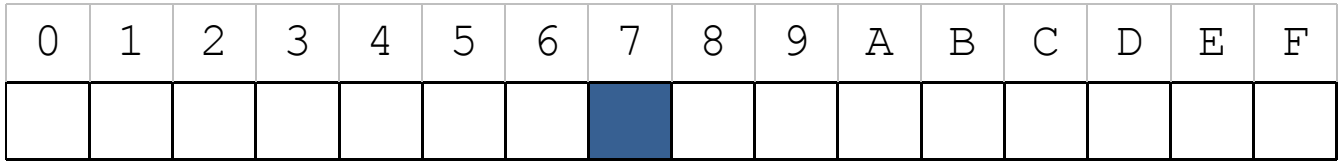
Data can be placed in few cache blocks i.e. **`block_num % (blocks_in_cache / n)`**

n=2

cache



RAM



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

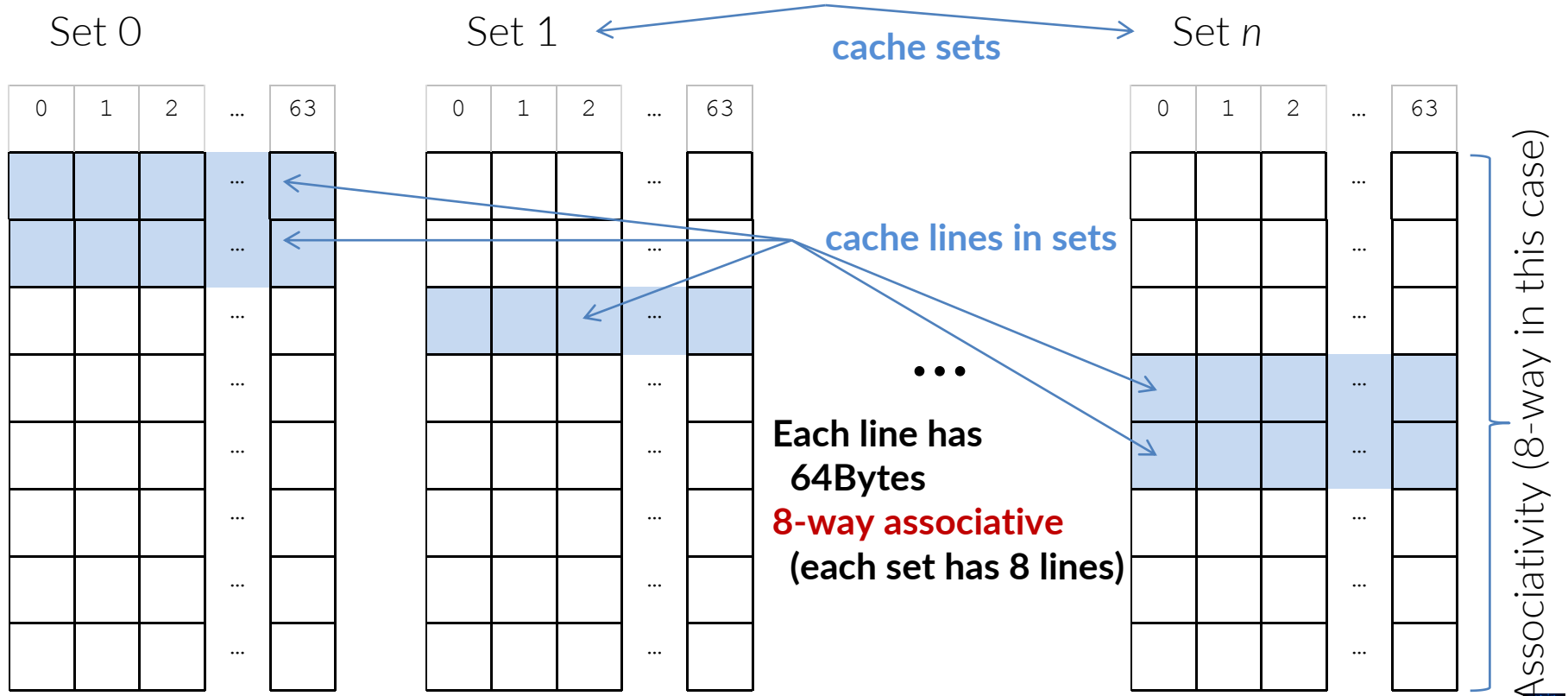
Pros

Very efficient in determining the set (only one per memory address). Smaller rate amount of cache conflicts.

Cons

More complicated logics; larger number of operations for the access.

A typical today cache



Memory→ Cache mapping

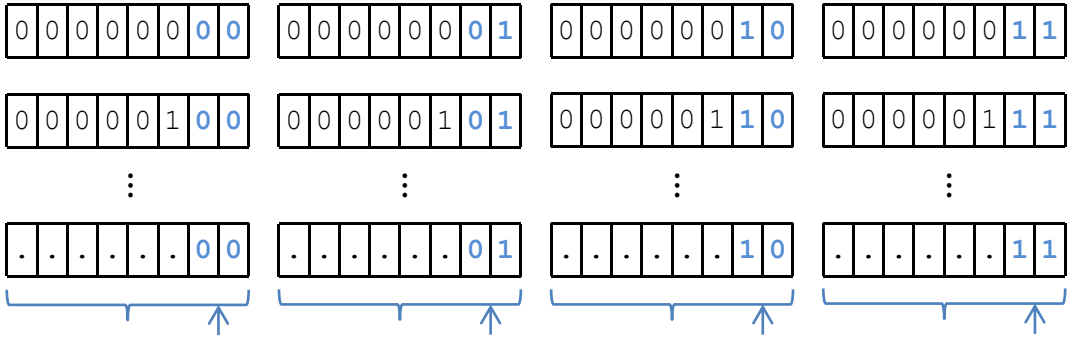
How a byte is actually mapped into a cache location ?

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

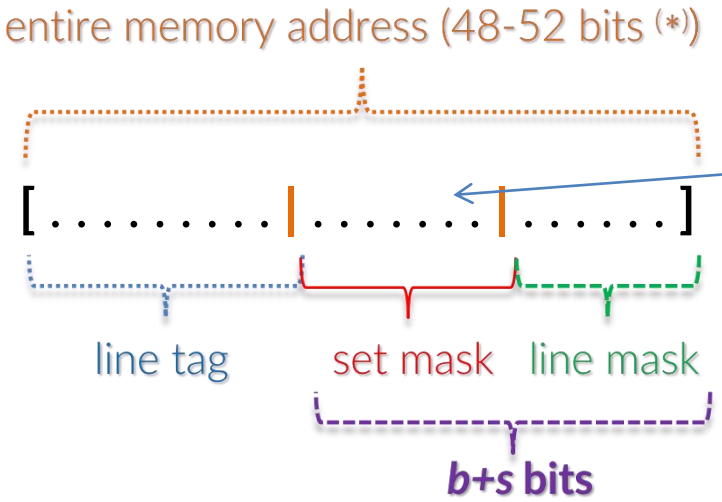
These bytes will always be at pos 2 in a cache line

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

$L = 64B = 2^b$ 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:

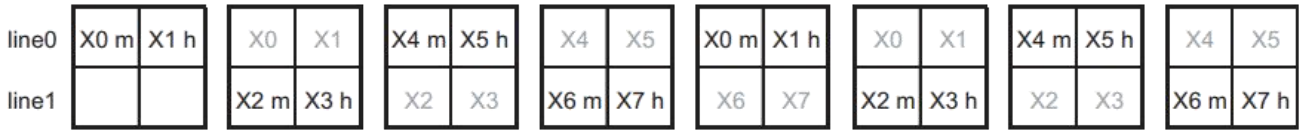
The memory access pattern

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]);
```

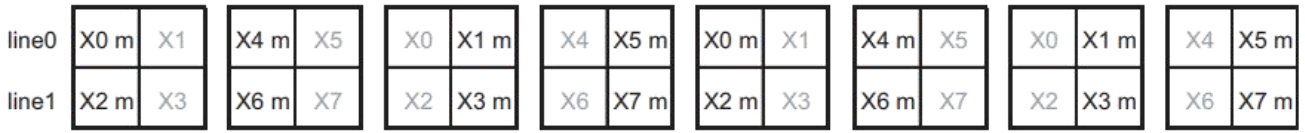


The hit-miss pattern is : MH MH MH MH MH MH MH MH,
the miss-rate is 50% (the first miss is compulsory miss).

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]);
   }
```

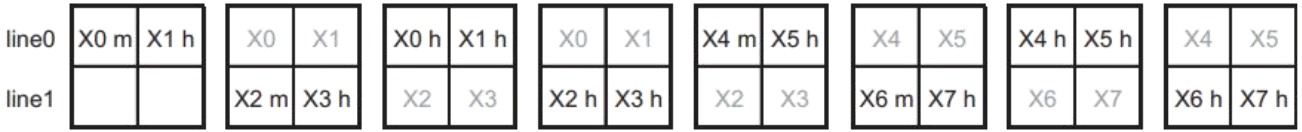


The hit-miss pattern now is : MM MM MM MM MM MM MM MM,
the miss-rate is 100%

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

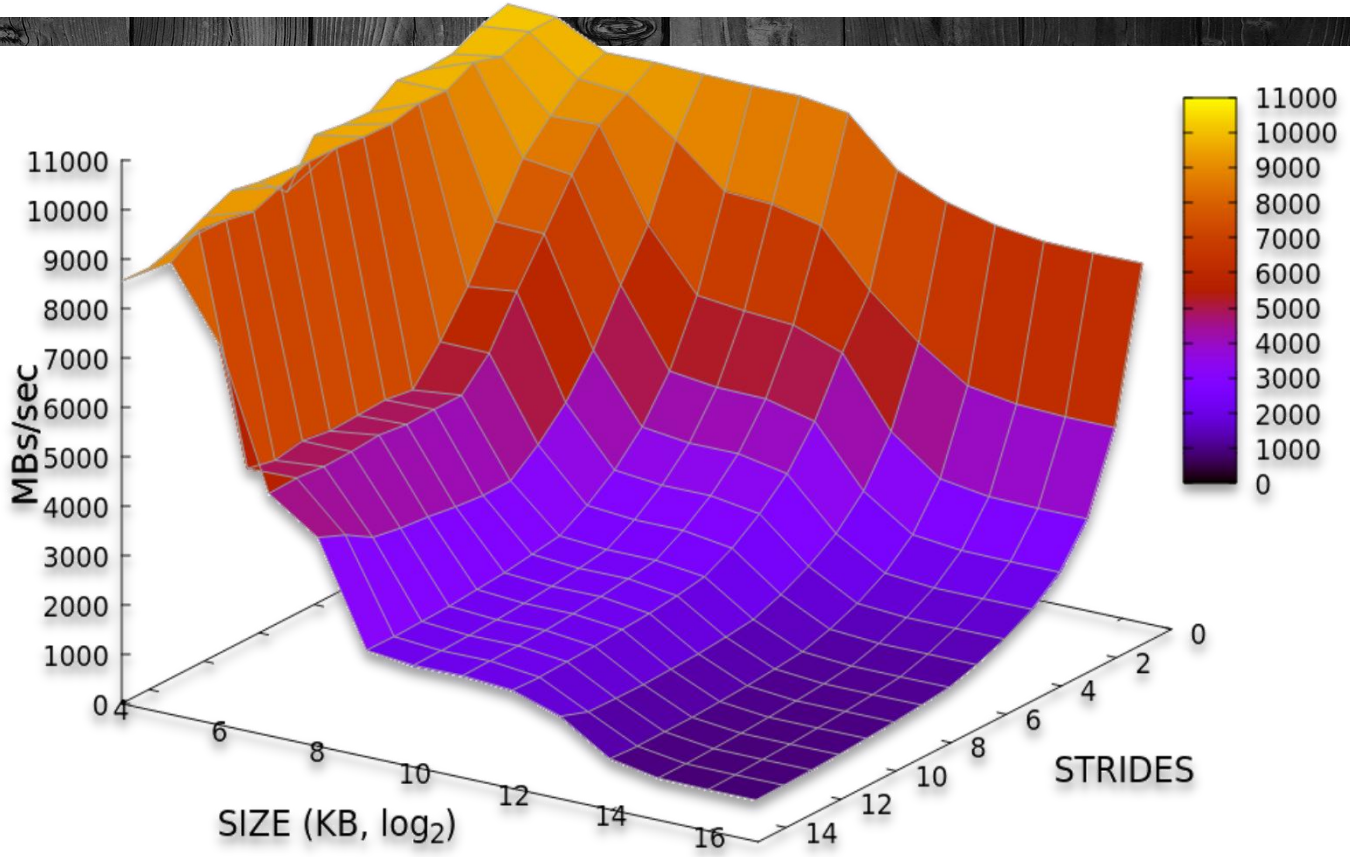
Cache recap in two slides

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

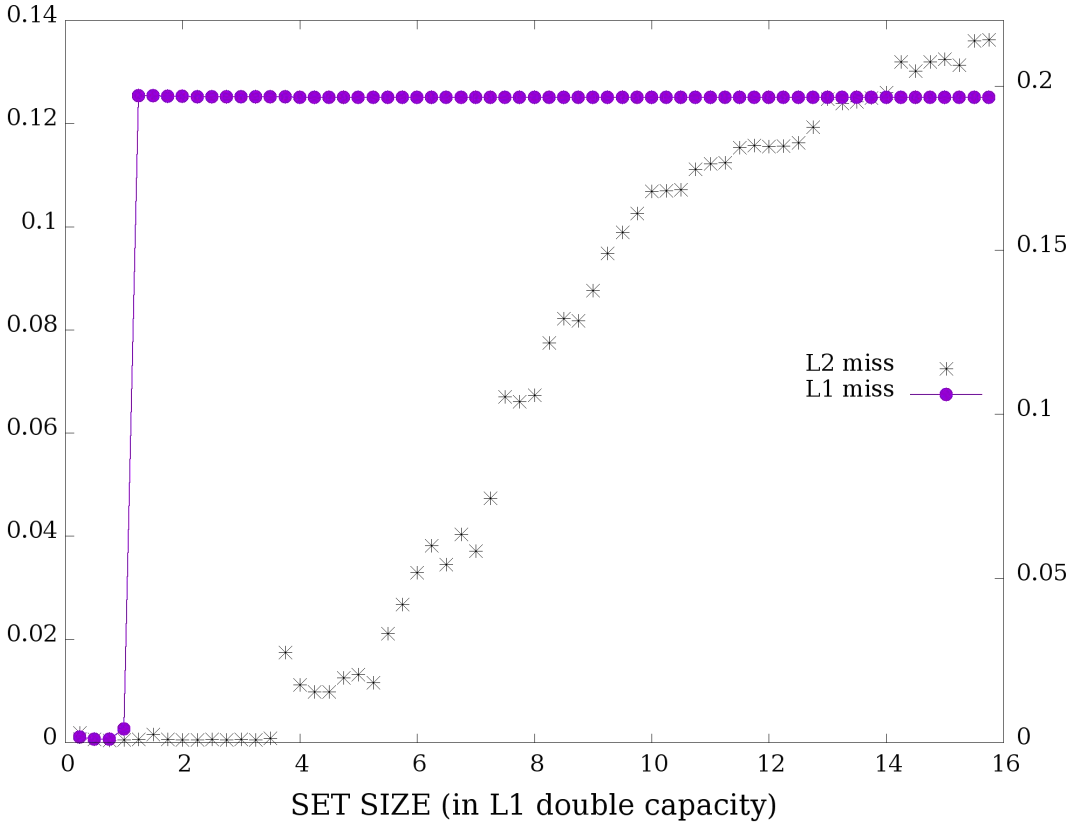


SCO/Examples_on_cache/
memory_mountain/mountain.c

The cache-miss signature

Let's find out our cache size

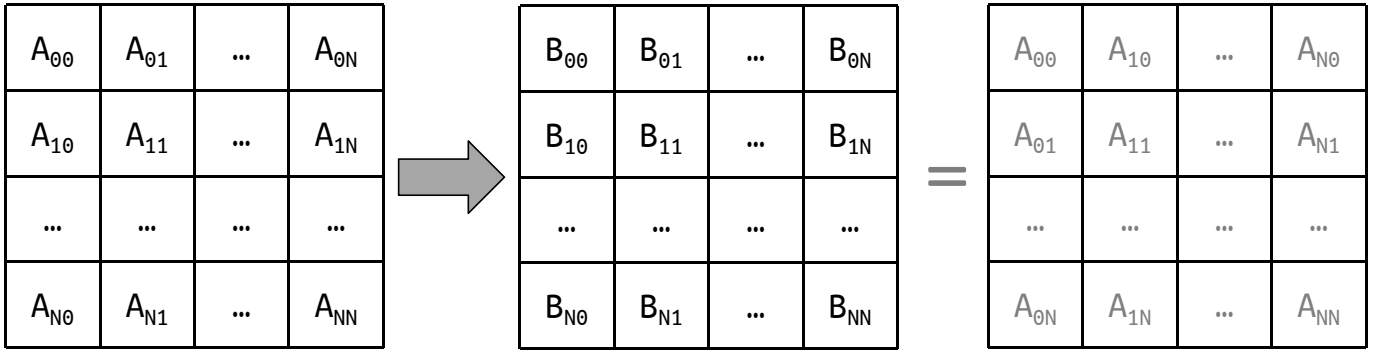
```
for (j=0; j < size; j++)
   array[j] =
     2.3*array[j]+1.2;
```



Strided access

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

Matrix transpose



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

Strided access

The very simple, straightforward implementation is:

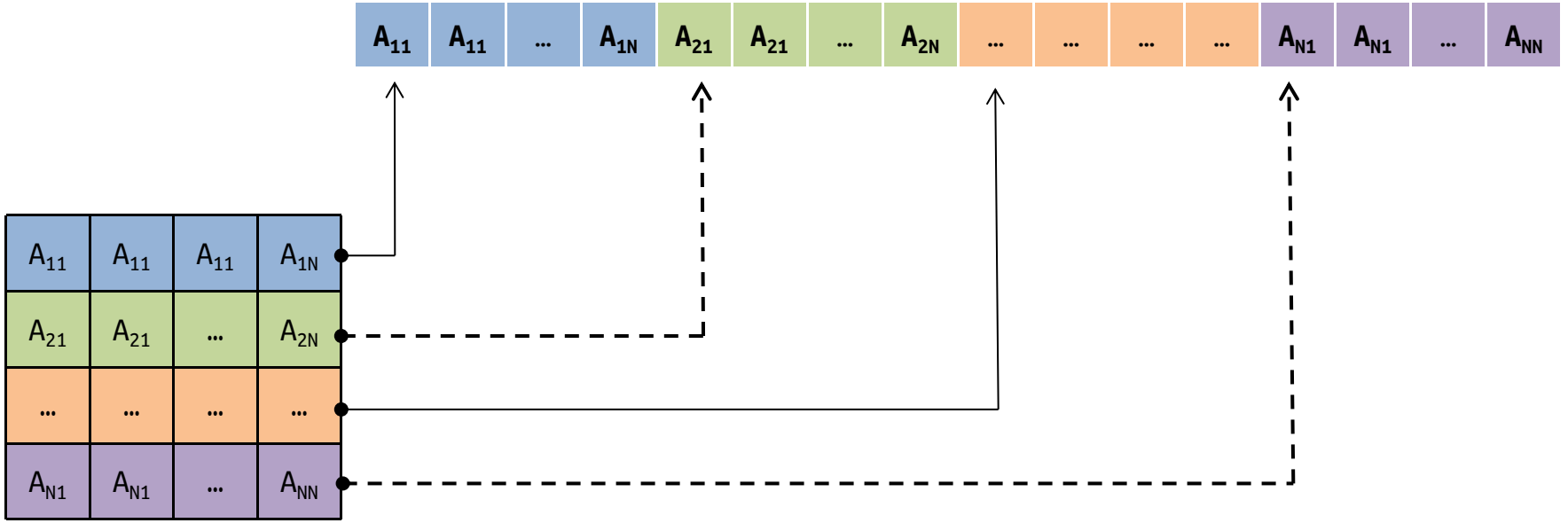
```
for(int row = 0; row < N; row++)
    for(col = 0; col < N; col++)
        A [ col*N + row ] = B [ row*N + col ];
```

Matrix transpose

SCO/Examples_on_cache/
matrix_transpose/matrix_transpose.c

Note: how a matrix is stored in memory

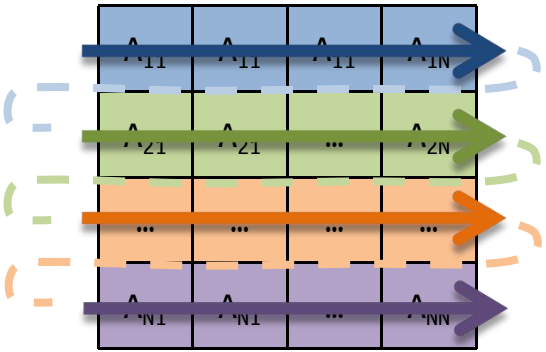
Remember the obvious fact that your memory is a continuous 1-dimensional stream of bytes. A 2D matrix is stored in memory by rows:



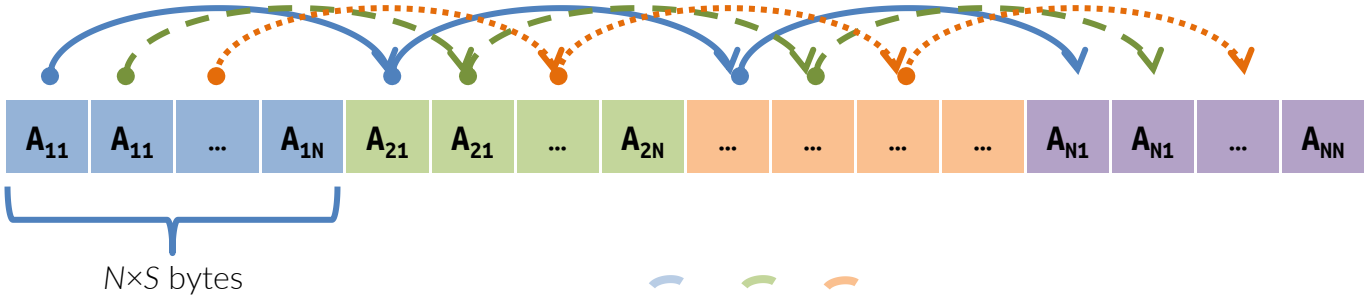
This convention is the C/C++ convention, which is labelled as ***row-major order***. Note that the Fortran convention is opposite, with columns being contigous in memory (*column-major*).

Note: how a matrix is stored in memory

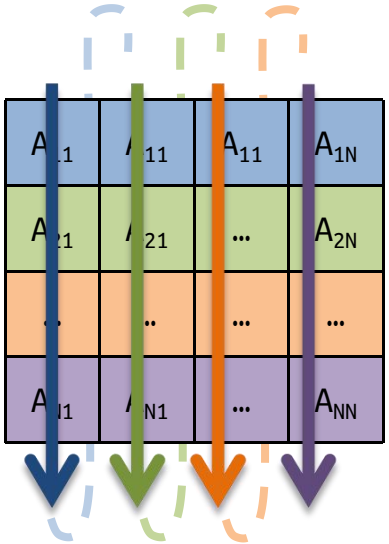
Then, traversing the matrix in the same row-major order corresponds to traverse the memory in contiguous order.



Note: how a matrix is stored in memory



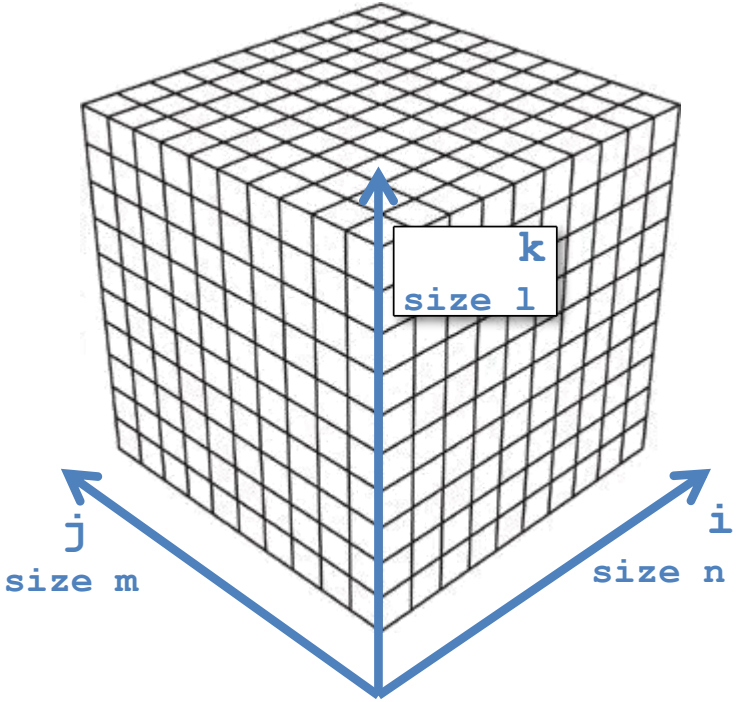
Whereas, traversing the matrix in the opposite column-major order amounts to jump in memory by *N* positions, i.e. *N*×*S* bytes, where *S* is the size of each element.



Note: how a matrix is stored in memory

How a 3D matrix, indexed as `[i][j][k]`, is stored in memory ?

`double M[n][m][l]`

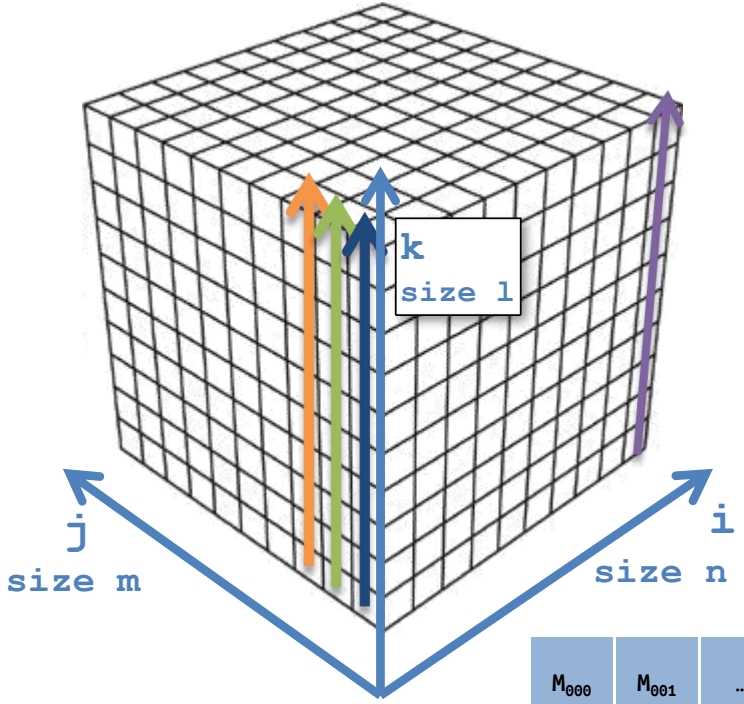


The innermost index is the one that varies at the fastest pace, and traces the memory contiguity.
Hence `M[i][j][k]` is contiguous to `M[i][j][k+1]`.

Note: how a matrix is stored in memory

How a 3D matrix, indexed as `[i][j][k]`, is stored in memory ?

`double M[n][m][l]`

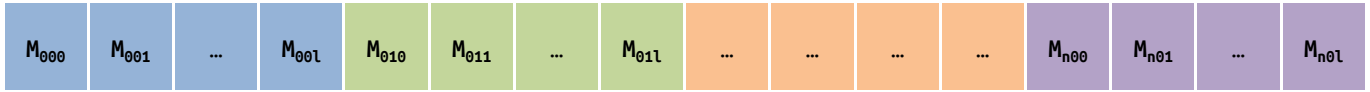


The innermost index is the one that varies at the fastest pace, and traces the memory contiguity.
Hence `M[i][j][k]` is contiguous to `M[i][j][k+1]`.

In the choice that we made here, the 3D matrix is then stored along the z direction, which is our `k` index.
Then by slices along the y direction, which is our `j` index.
And finally along the x direction, which is our `i` index, the slowest in changing.

The, to directly reference the entry `[i][j][k]` in M one should use

```
M + i*(m*l) + j*l + k
```



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* ?

Dealing with writes

When you modify the content of a variable, that change amounts to overwrites a memory location. When the variable is loaded in to the cache memory, what is loaded is a *copy* of the variable, that maintains its original location in the main memory.

So, if we modify its value, shall we modify it in the cache only or in the main memory too?

there are 2 possible policies:

1) **write-through**
memory and cache are kept consistent, so the value is immediately written back in memory

2) **write-back**
the new value is stored only in the cache, and it is propagated to lower levels when the cache row is replaced.

Dealing with writes

Writing complicates things in the cache a lot, depending on what policy the cache implements.
If there is a *write-miss* in a write-through cache, you may either allocate a block in the cache (*write-allocate*) or not and directly write in the main memory (*non-write-allocate*). In the latter case, no data fetch are necessary. Non-write-allocate is useful for burst writing operations.

If in a write-back cache you , you first have to write back the block in memory

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 ];
```

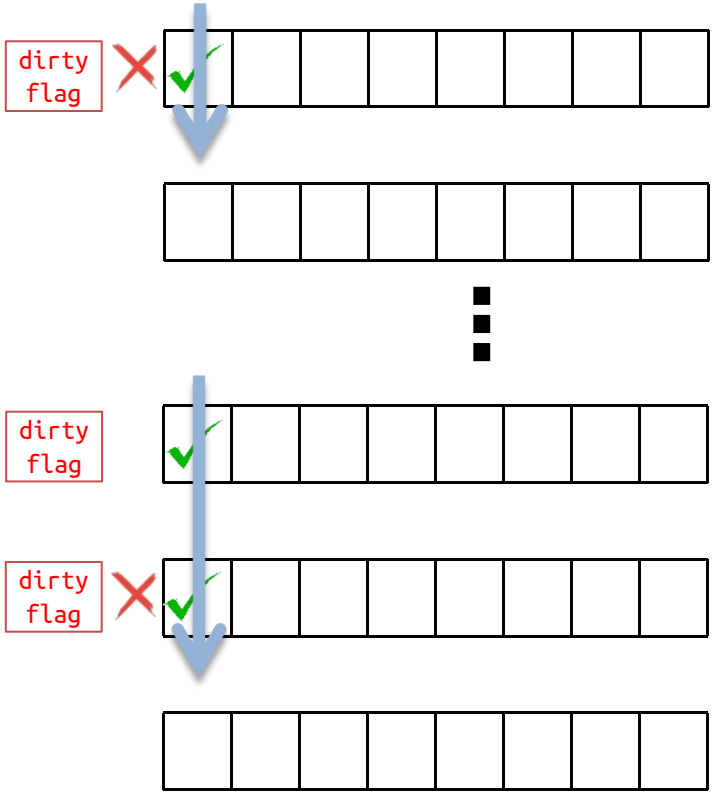
NOTE: strided access to either **A** or **B** is unavoidable.

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.

SCO/Examples_on_cache/
matrix_transpose/matrix_transpose_swapped.c

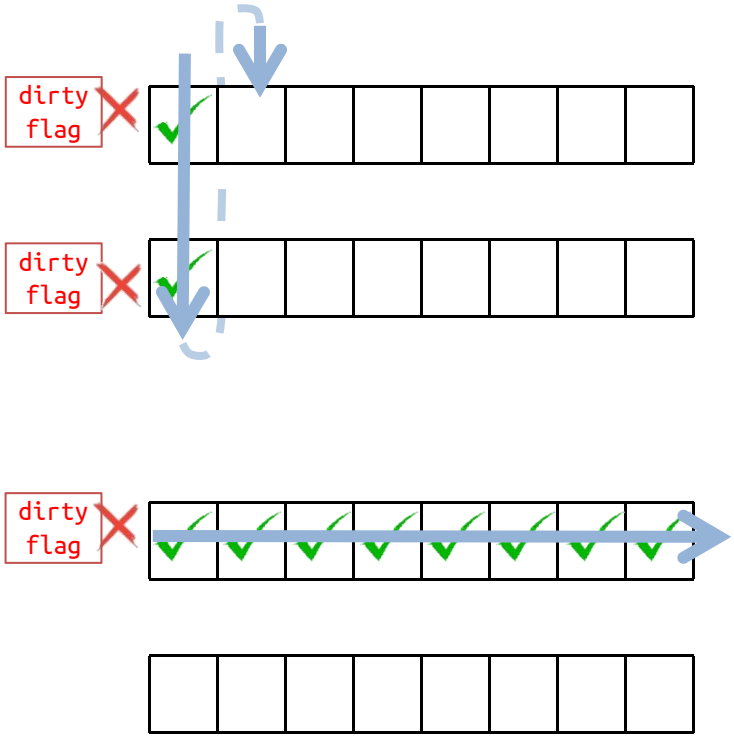
Strided access



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 if it is replaced.

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.

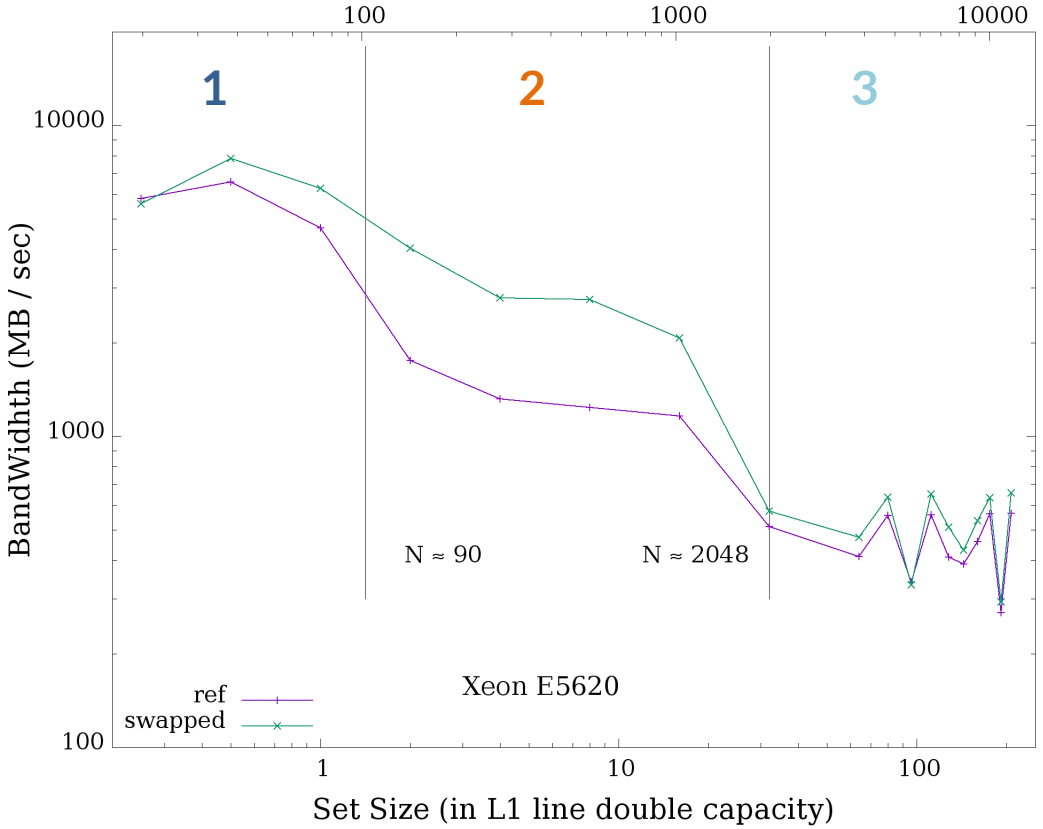
Strided access

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:

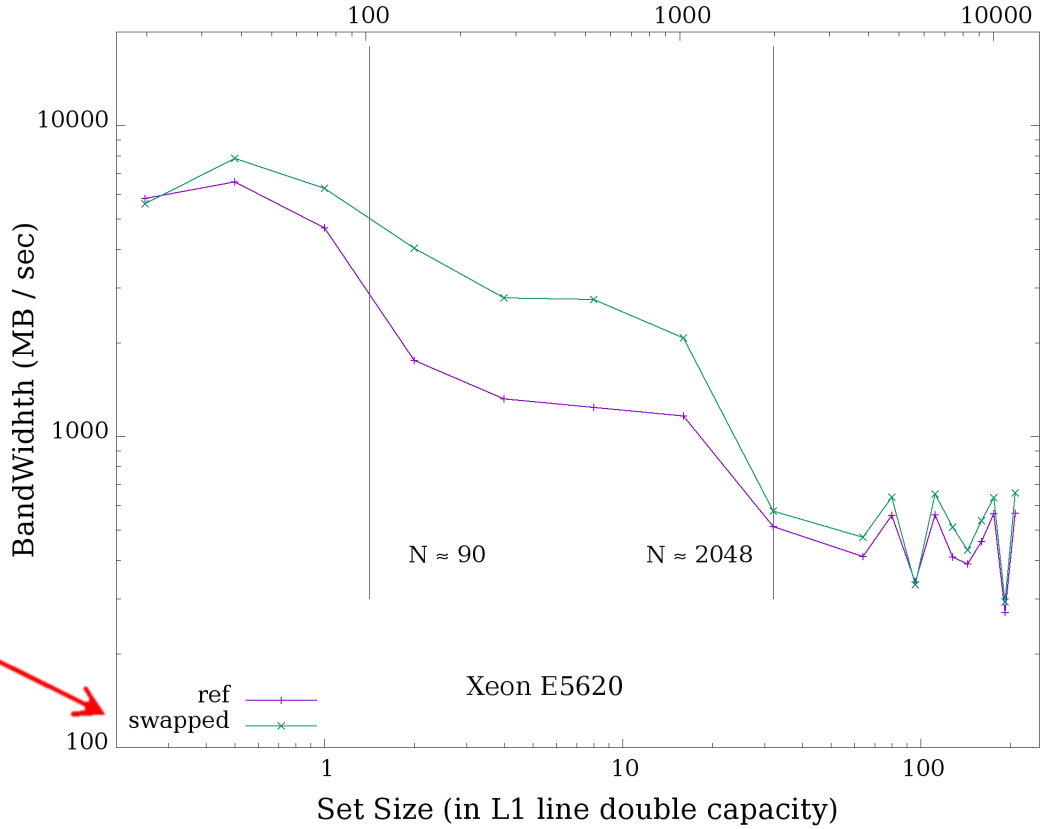
1. $2 \times N^2 < C$
   both matrices can fit in the cache.

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.



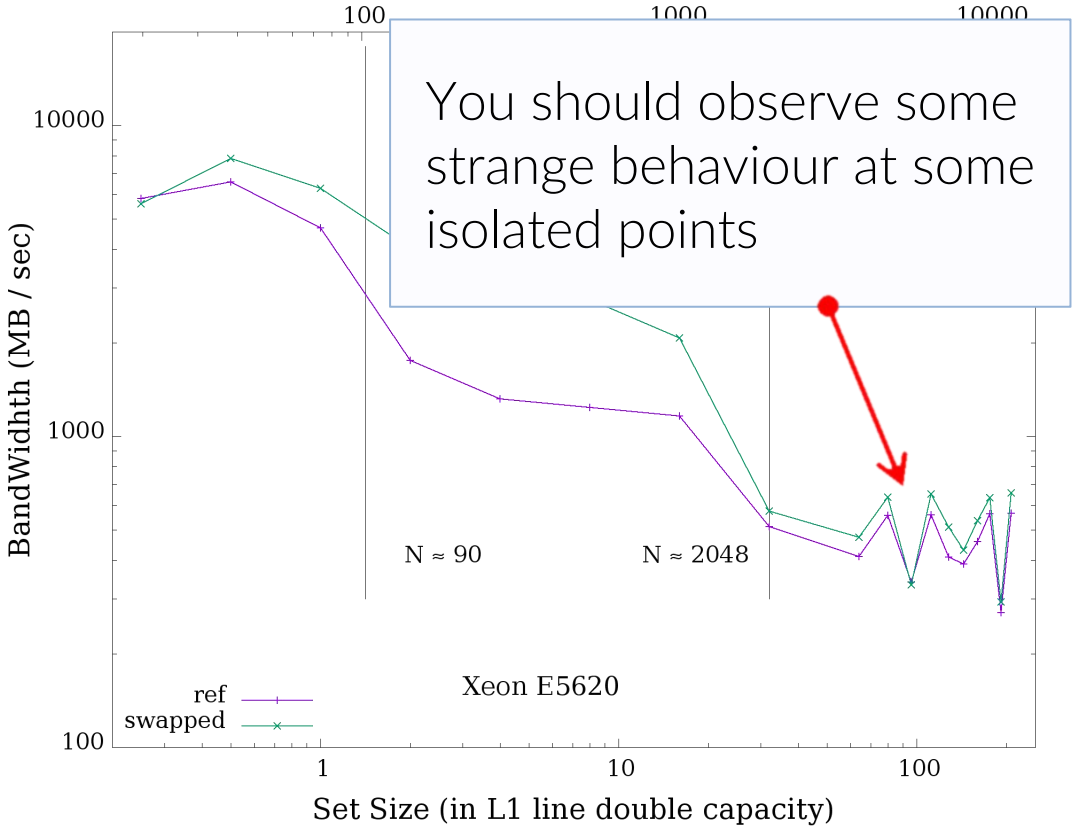
Cache-associativity conflicts

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.



Enhancing the locality

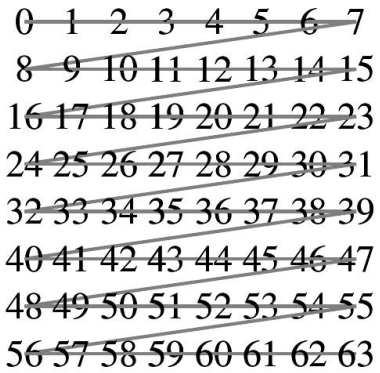
1) by how you *go through* the data

2) by how you *organize* the data

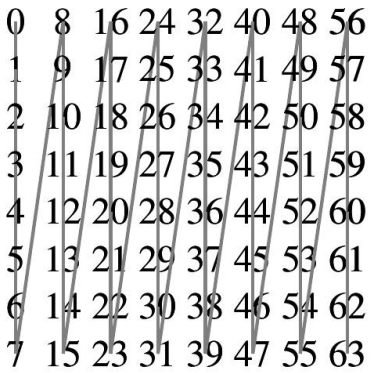
Getting through the data to enhance locality

Scanline order *row-major*

(a)



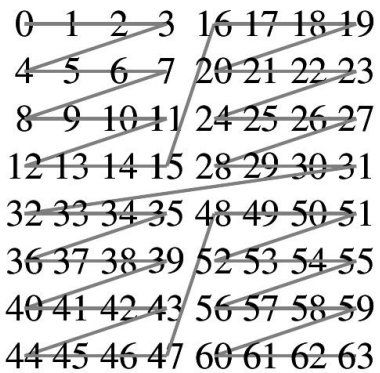
(b)



Scanline order *col-major*

Block order + scan sub-order

(c)

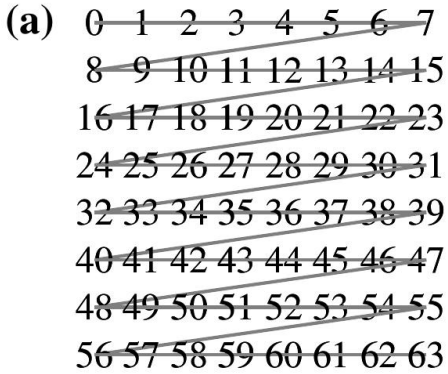


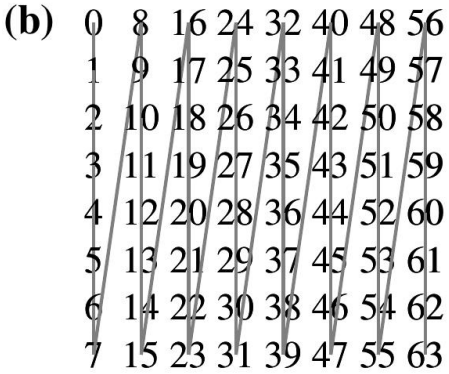
That will be an exercise

What if you could generate the same by-blocks pattern “naturally”, without hard-coding it into the code, which brings a fine-tuning of a parameter (the value of the block size) on a given platform?
The performance may be not as optimal as with a fine-tuning but the average cperformance on all platforms may result to be very good.

Getting through the data to enhance locality

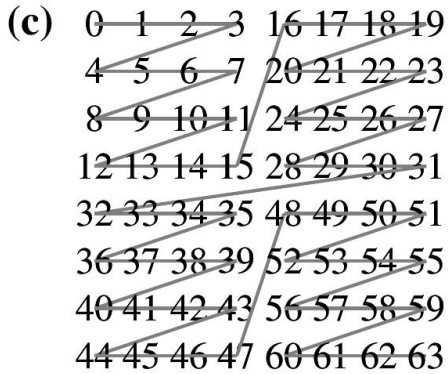
Scanline order *row-major*

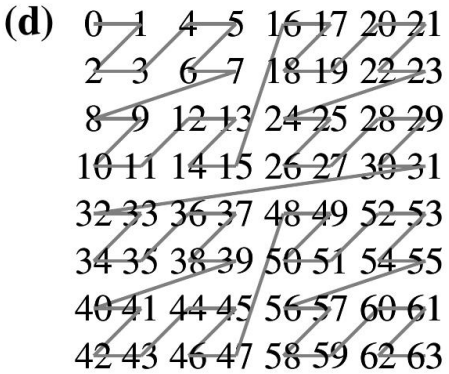




Scanline order *col-major*

Block order + scan sub-order





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

That will be an exercise

Organizing data to enhance locality

When the memory bandwidth is “limited” – which may be the case for highly parallel + multicore systems with a strong 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.

Hot & cold fields

Reorder the 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 the 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 the 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 the 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 the 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

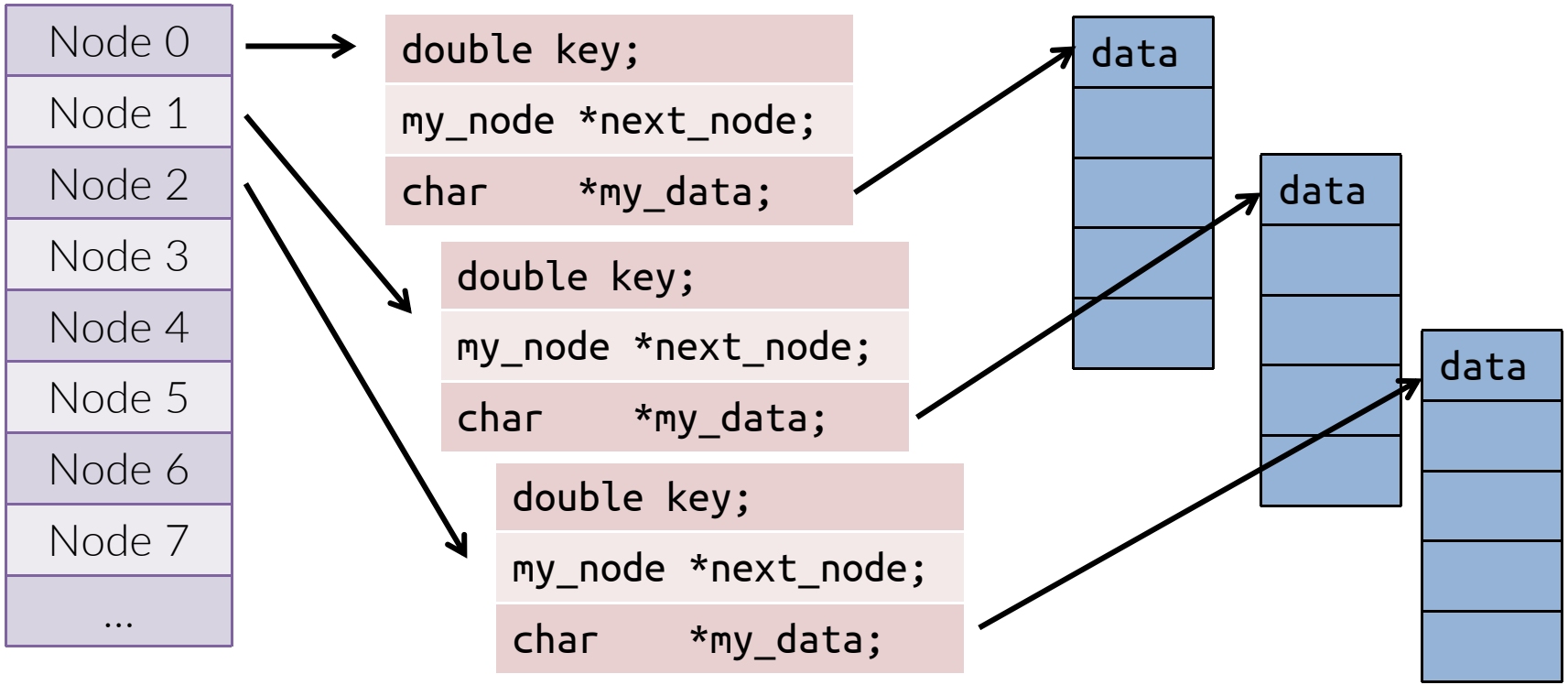
Split the fields so that to keep consecutive the fields that are used sequentially

```
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



Outline

Avoid the avoidable inefficiencies

Cache & Memory

Loops

Branches

Pipelines

Unleash the Compiler

Optimizing cache access in loops

Loop classification

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

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

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

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

Example

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

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

O(N) / O(N)

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

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

➔

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

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

[check the possibility for loops fusion]

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

Example

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

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

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

N×N N×N N×N

→ $3 \times N^2$ memory accesses

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

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

→

Step 1: Avoid unecessary loads /stores

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

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

→ $2\times N^2+N$ memory accesses

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

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

→ $N^2 \times (1+1/m) + N$

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

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

N×N/m

N×N

N

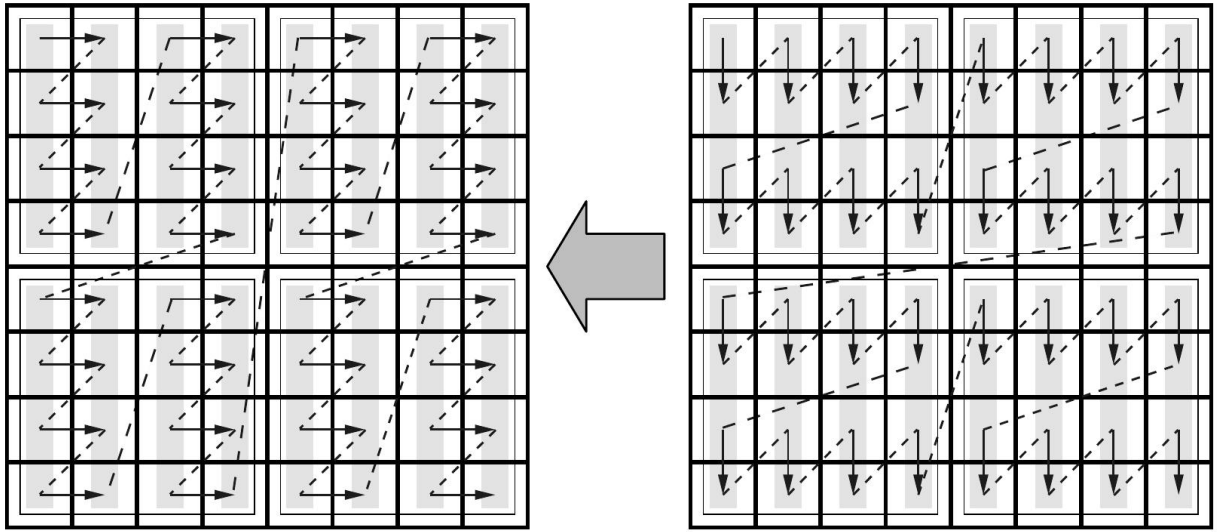
Cache access in loops: $O(N^2)/O(N^2)$

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

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

Loop tailing (or blocking) is a good strategy that does not save memory loads but increase dramatically the cache hit ratio.
We have mentioned it in the past, we'll se it in more detail at the end of this lecture.

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



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

Image taken from: introduction to HPC for scientists and engineers

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

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

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

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

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

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

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

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

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

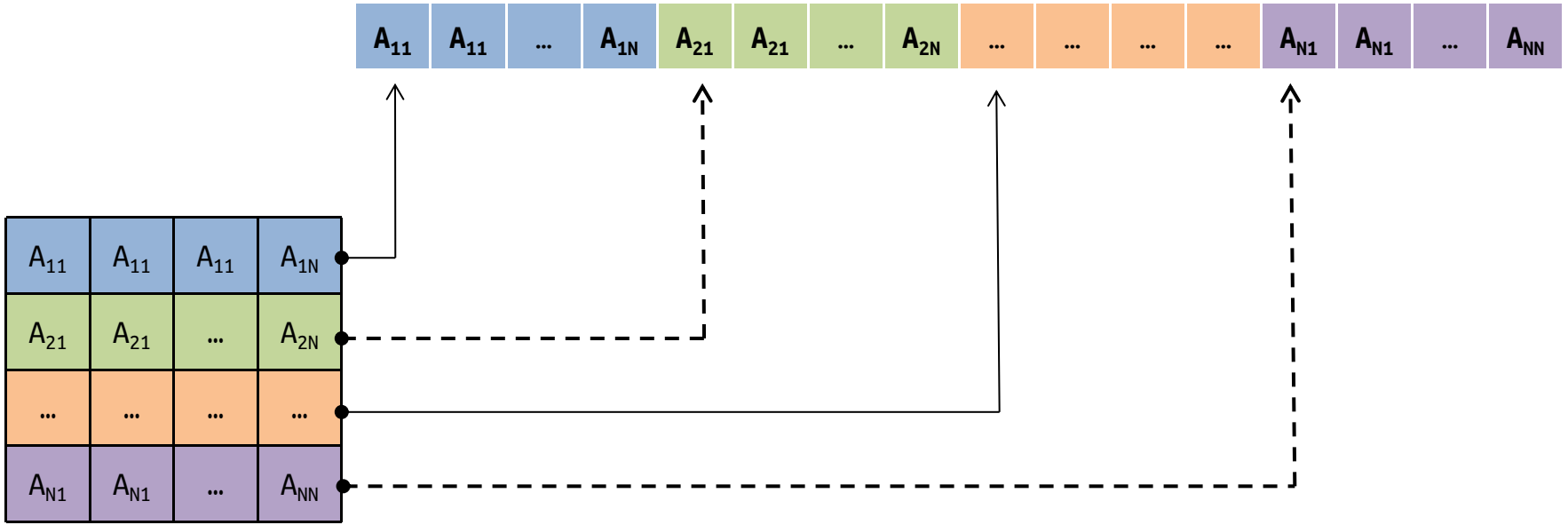
$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} \times B_{k,j}$$

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

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

Remind: how a matrix is stored in memory

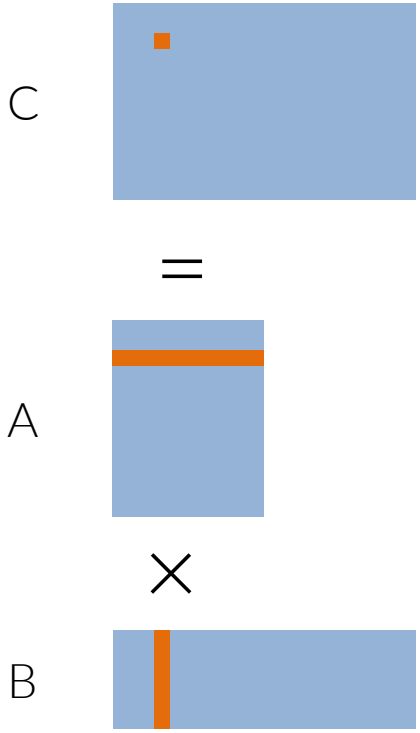
Remember the obvious fact that your memory is a continuous 1-dimensional stream of bytes.



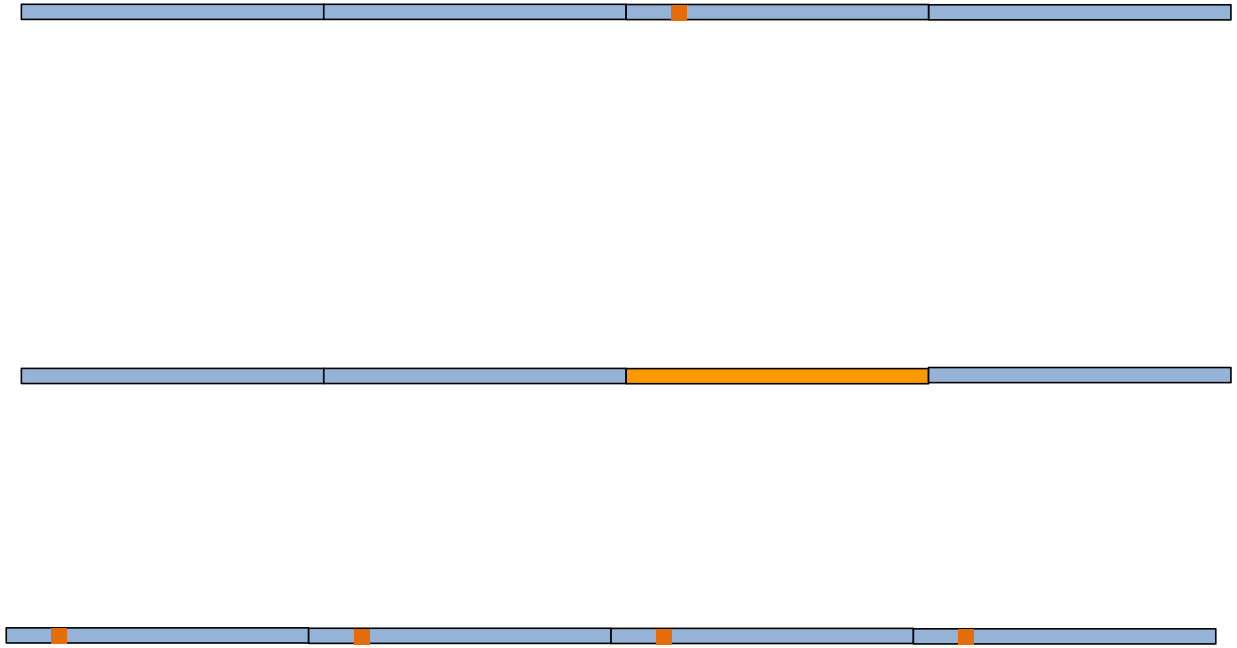
This convention is the C/C++ convention, which is labelled as *row-major order*. Note that the Fortran convention is opposite, with columns being contigous in memory (*column-major*).

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

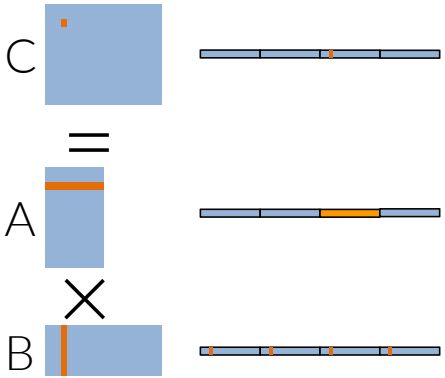
Matrix representation



Memory representation



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



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

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

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

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

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

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

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

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

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

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

Now we expect to have

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

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

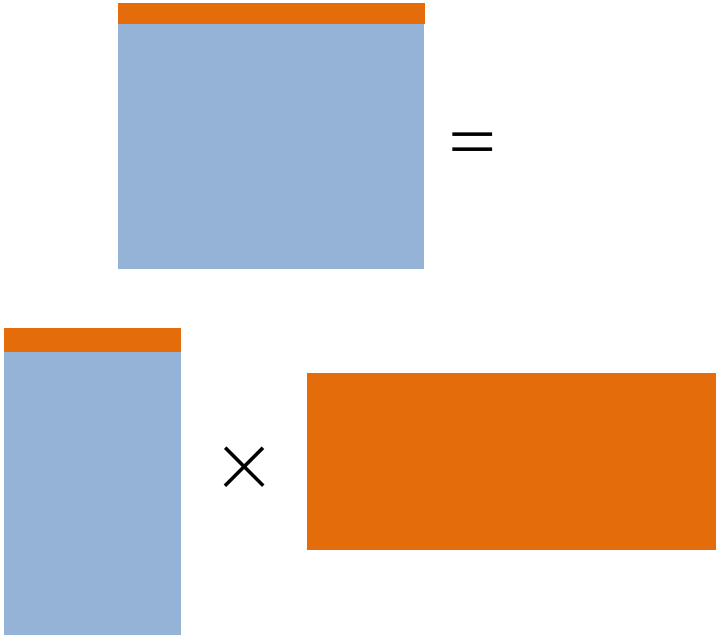
~ mnp(L -2)

less cache misses.

SCO/examples_on_pipelines/matrix_multiplication

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

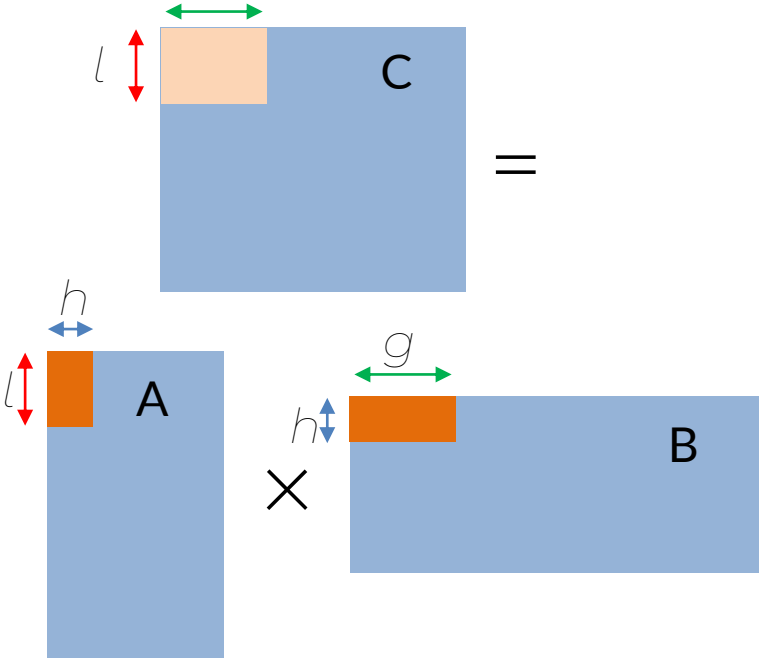
We can do even better by optimizing both the memory accesses and the data contiguity by *tailing* the loops:



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

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

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



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

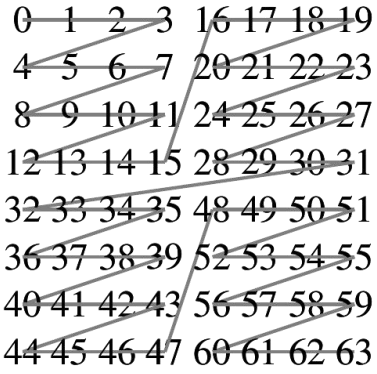
Traversing A and B by *blocks* as in the figure allows to accumulate partial results in the corresponding C's area while decreasing the number of cache misses by a factor

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

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

Traversing matrices by blocks

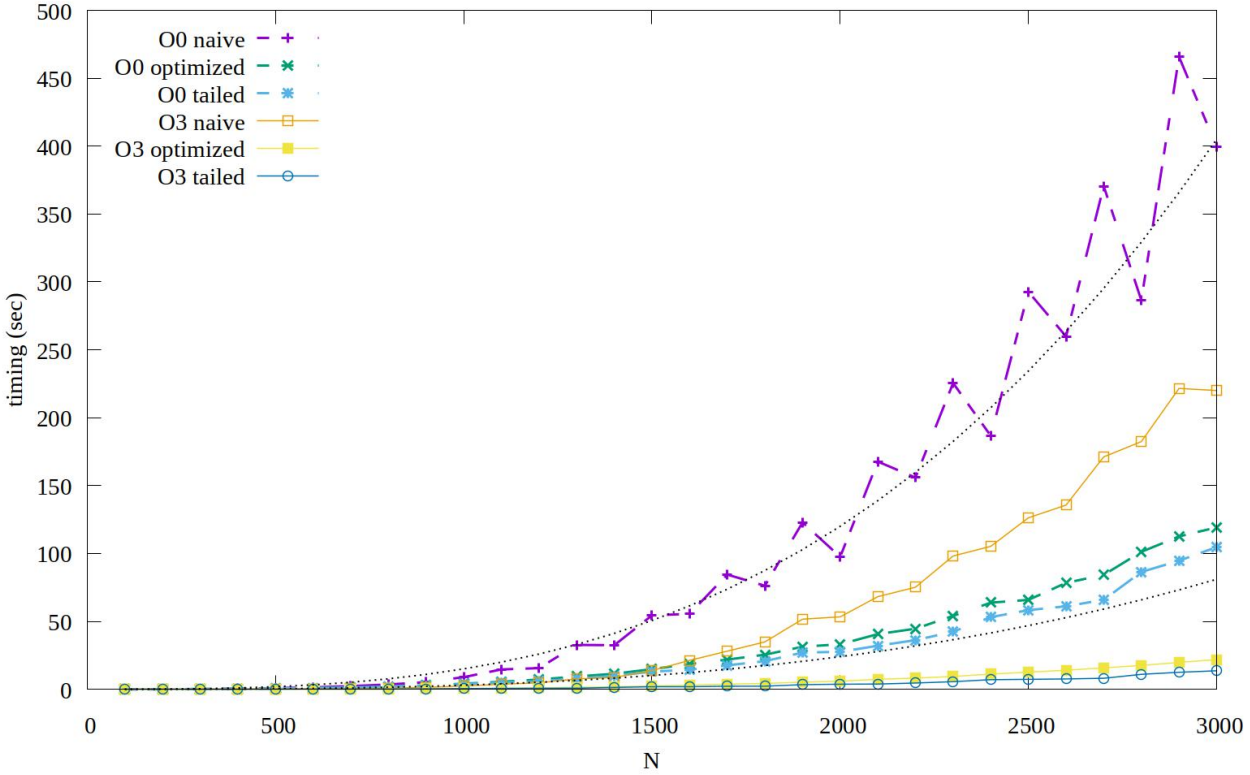
A common technique in matrix-matrix multiplication A×B = C, or in matrix inversion is to process the matrices by blocks instead of traversing entire rows/columns.
Obviously, if the block size is choosen wisely, that enhances the possibility that all the 3 blocks (from A, B and C) are hosted in L1




```c
void mblock (int n, int bsize,
             double *A, double *B, double *C)
{
  for (int ii = 0; ii < bsize; ++ii)
    for (int jj = 0; jj < bsize; ++jj) {
      int    jjn = jj*n;
      double cij = C[jjn + ii];           /* cij = C[i][j] */

      for( int kk = 0; kk < bsize; kk++ )
        cij += A[ii+kk*n] * B[kk+jjn];    /* cij += A[i][k]*B[k][j] */

      C[ii+jjn] = cij; }                  /* C[i][j] = cij */
}

void dgemm (int n, int bsize, double* A, double* B, double* C)
/* C_ij = SUM A_ik * B_kj */
{
  for ( int j = 0; j < n; j += bsize )
    for ( int i = 0; i < n; i += bsize )
      for ( int k = 0; k < n; k += bsize ) {
        double *AA = A + k*n + i;
        double *BB = B + j*n + k;
        double *CC = C + i*n + j;
        mblock(n, bsize, AA, BB, CC); }
}
```

note: this snippet does not take into account the case n%bsize != 0

Matrix multiplication results



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

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

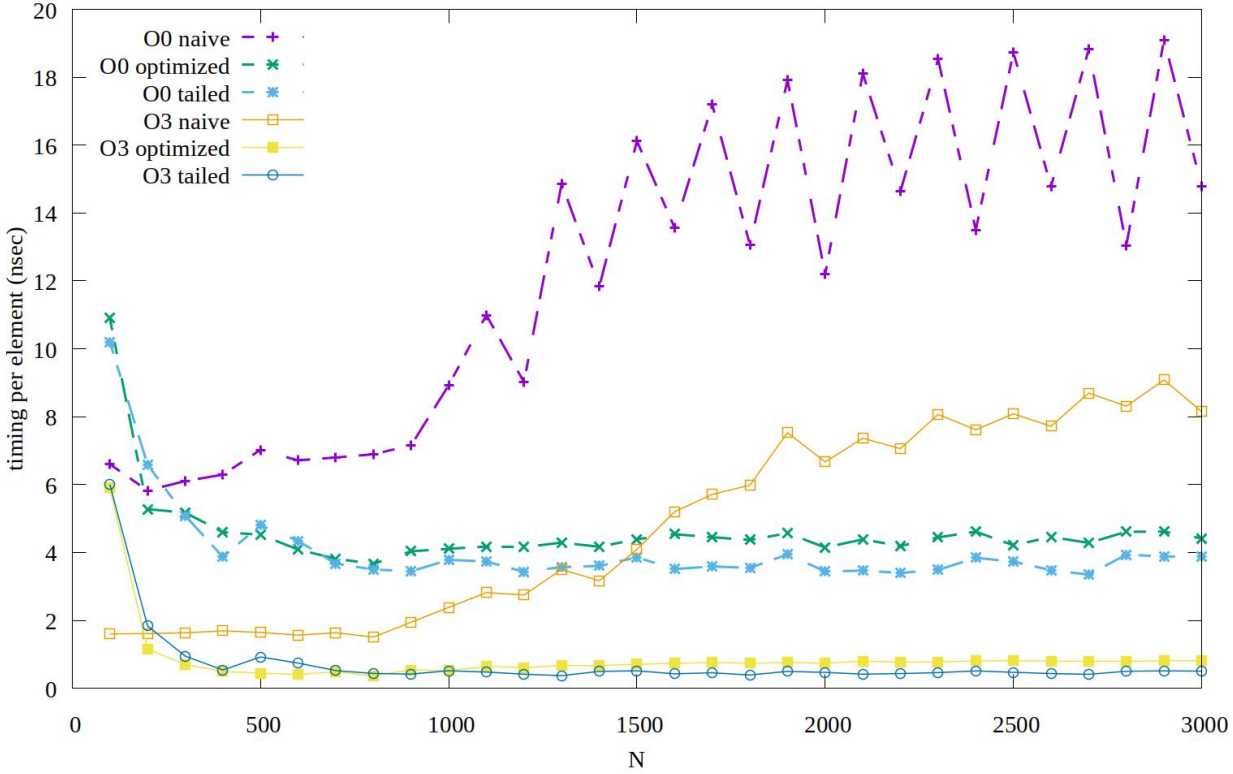
Naïve: the schoolbook's implementation

Optimized: inner loops swapped

Tailed: M-M by blocks

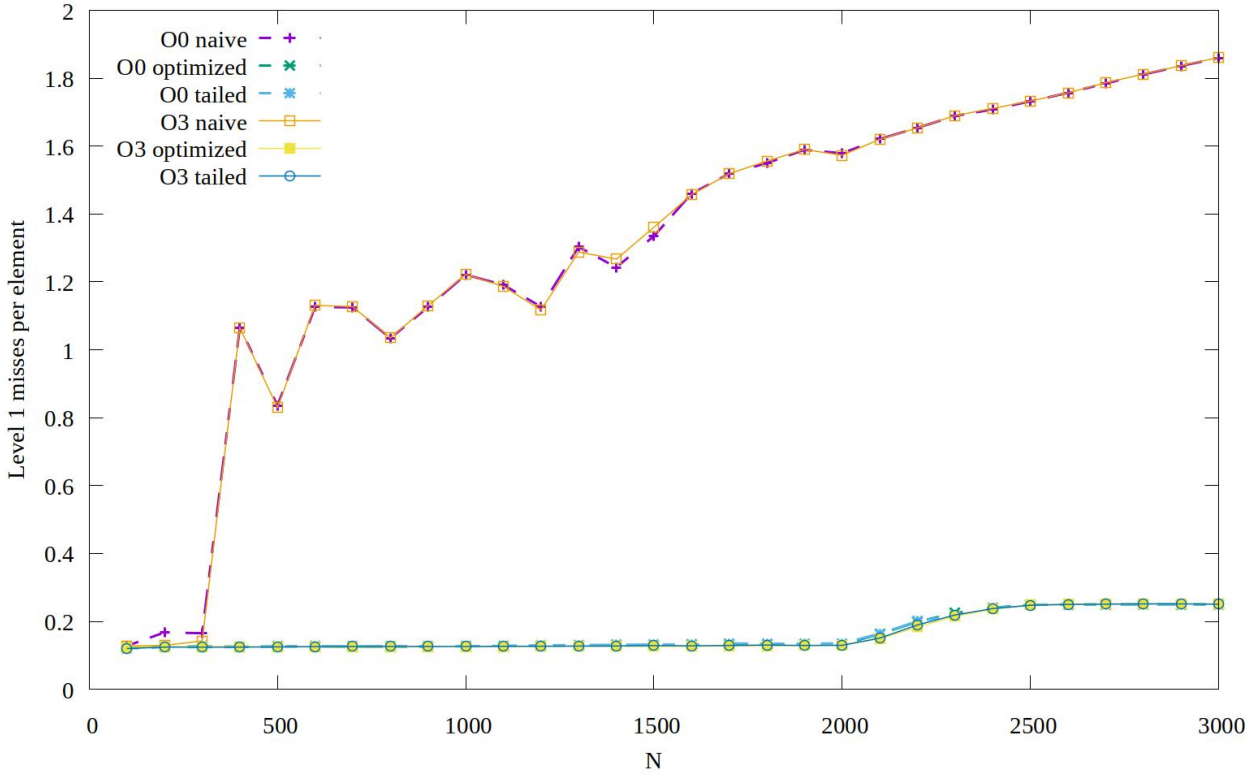
In this plot and in the following ones dashed lines are for `-O0` and solid line for `-O3 -march=native` (only `gcc` has been used)

Matrix multiplication results



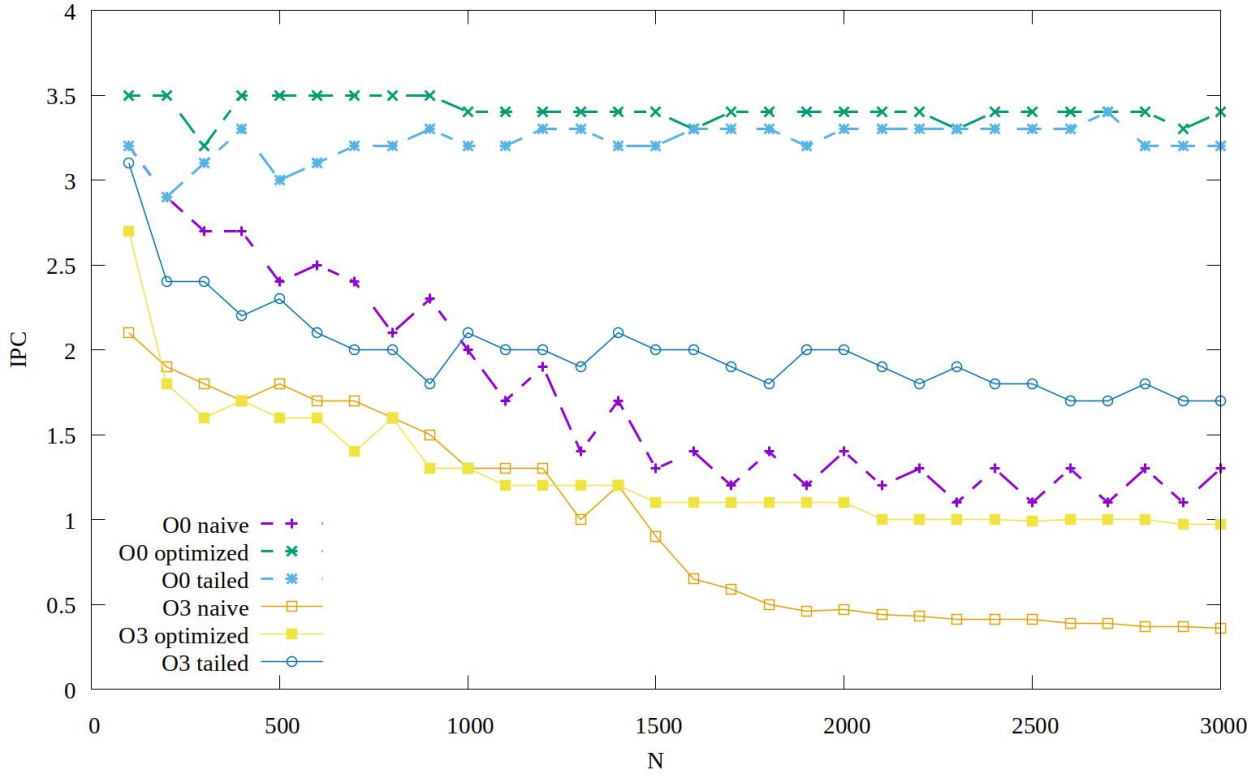
Time per element accessed (N^3)

Matrix multiplication results



Level 1 Data cache misses per element

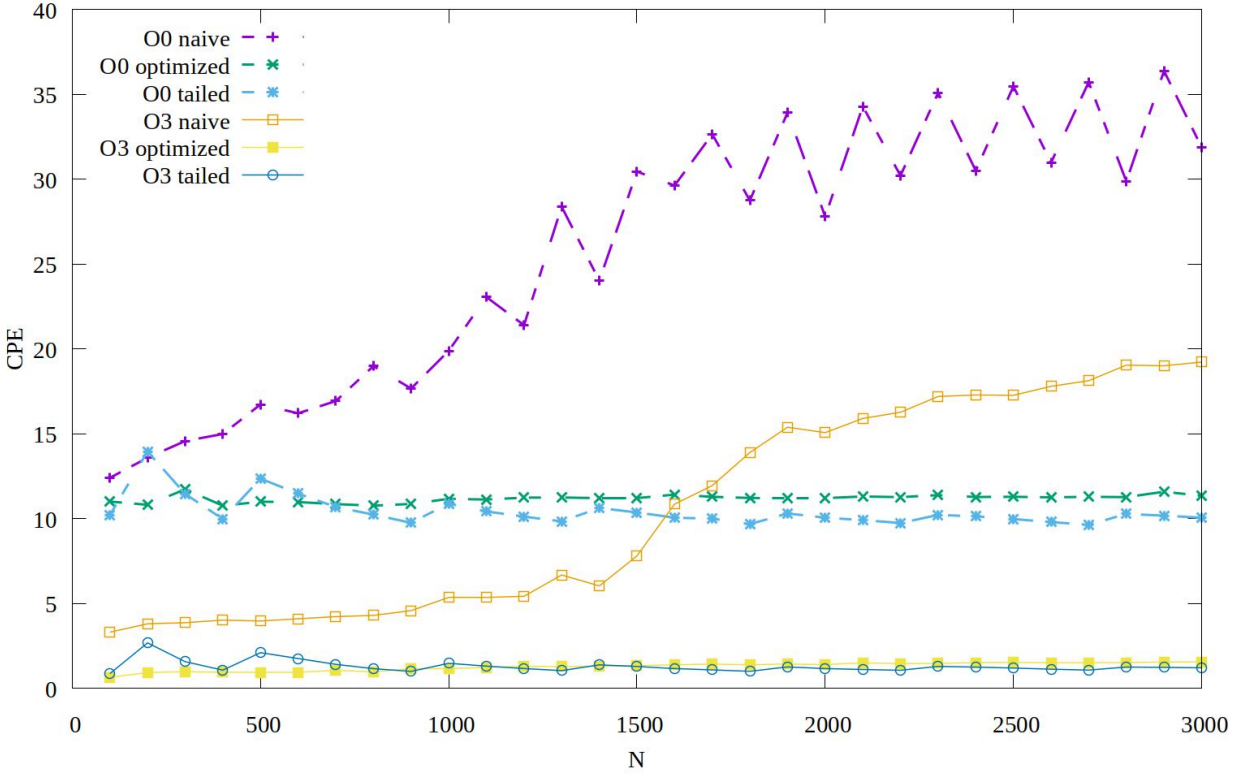
Matrix multiplication results



Instructions per cycle

(the larger the better)

Matrix multiplication results



Cycles per element

(the smaller the better)

Loop unrolling

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

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

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

Loop unrolling

Let's examine this simple *reduction*:

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

-O0

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

-O3 -march=native

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

        ret
```

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

Loop unrolling

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

`-O0`

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

`-O3 -march=native`

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

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

Loop unrolling

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

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

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

Loop unrolling

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

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

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

SCO/examples_on_pipelines
reduction

Step 1: unrolling 2×1

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

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

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

Step 1: unrolling 2×1

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

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

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

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

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

Step 1: unrolling 2×1

NOTE:

The unrolling is expressed in general as ***n*×*m*** (in this slide *n=2*, *m= 2*; in the previous slides, *n=2*, *m=1*).
n refers to the number of iterations that are unrolled
m refers to the number of accumulators that are being used

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

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

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

Step 1: unrolling 2×1

The compiler generates (*)
the following assembly code:

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

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

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

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

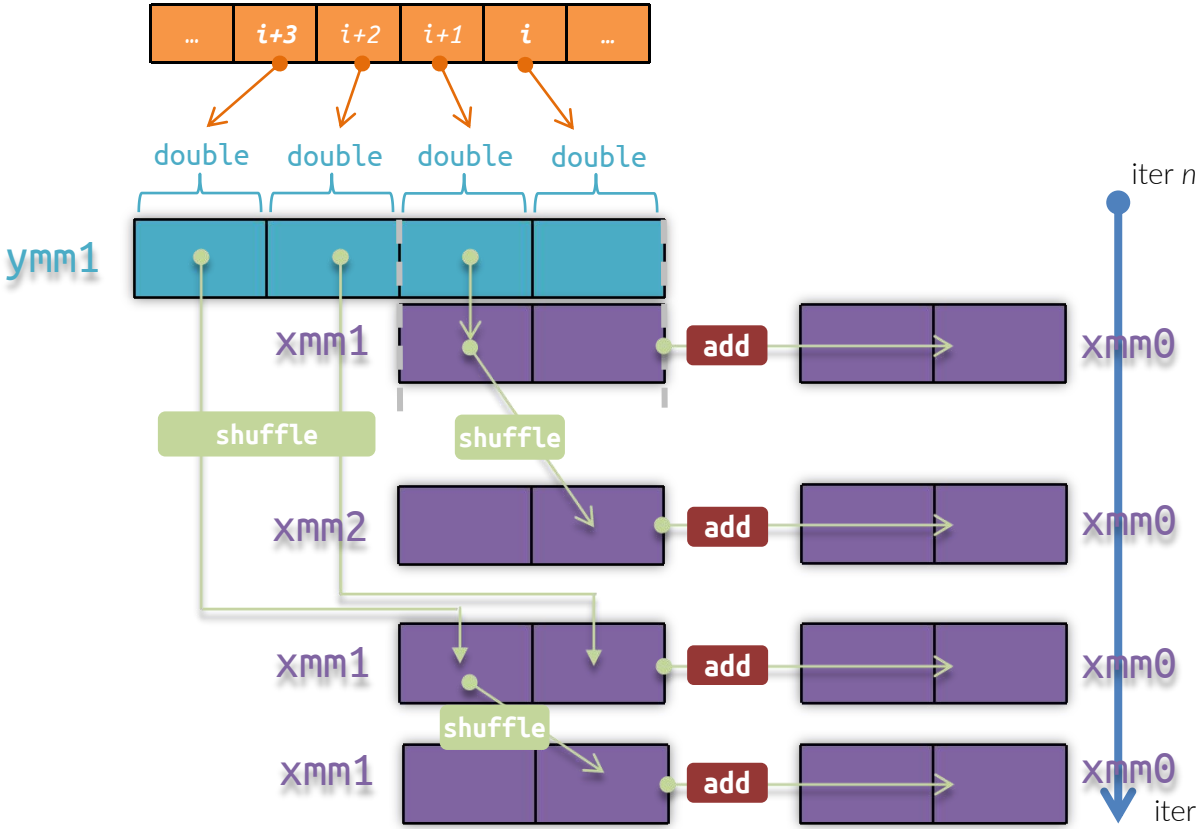
all these instructions sum with, and store in, **xmm0**

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

Step 1: unrolling 2×1

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

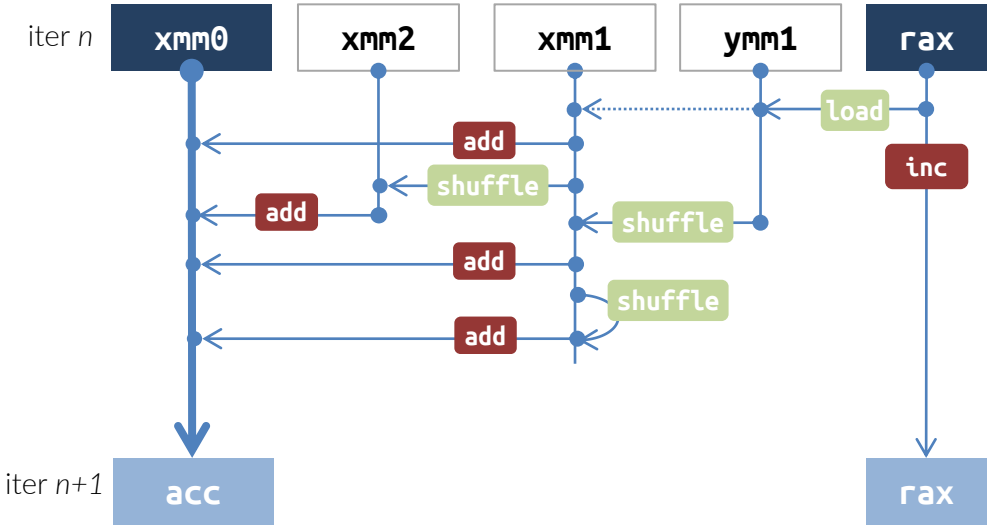
```
.L17:
        vmovupd ymm1, YMMWORD PTR [rax]
        add     rax, 32
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd xmm2, xmm1, xmm1
        vextractf128    xmm1, ymm1, 0x1
        vaddsd  xmm0, xmm0, xmm2
        vaddsd  xmm0, xmm0, xmm1
        vunpckhpd xmm1, xmm1, xmm1
        vaddsd  xmm0,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



Step 1: unrolling 2×1

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

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



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

Step 2: unrolling 2×1+ reshuffle

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

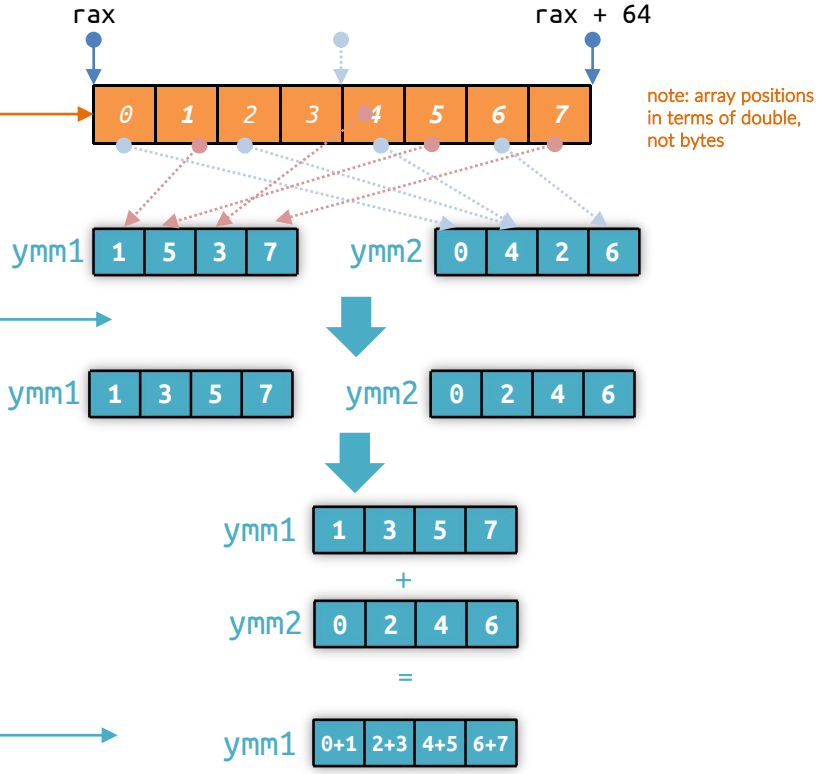
```
for ( int i=0; i<N-2; i+=2 )
  S = (S OP A[i]) OP A[i+1] ;
```

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

Step 2: unrolling 2×1+ reshuffle

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

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



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

4 summation of subsequent elements in the array!

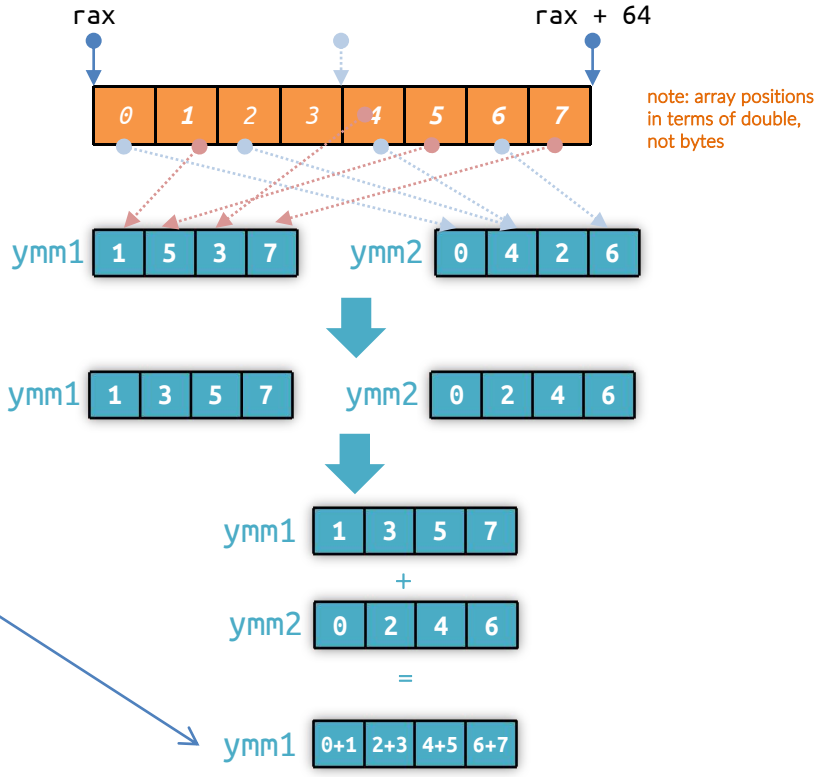
Step 2: unrolling 2×1+ reshuffle

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

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

```
Sum= ((([0]+[1]) + ([2]+[3])) + ([4]+[5])) +
([6]+[7])…
```

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



4 summation of subsequent elements in the array!

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

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

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

$$[A] \rightarrow [[[[0+a_0]+a_1]+a_2]+a_3]+\cdots\$\$$$
$$S_0 = a_0$$
$$S_1 = S_0 + a_1$$
$$S_2 = S_1 + a_2$$
$$\cdots$$
$$S_i = S_{i-1} + A_i$$

$$[B] \rightarrow [[[0+(a_1+a_2)]+(a_3+a_4)]+(a_5+a_6)]+\cdots$$
$$S_0 = 0 + (a_0 + a_1)$$
$$S_1 = S_0 + (a_2 + a_3)$$
$$S_3 = S_2 + (a_4 + a_5)$$
$$\cdots$$
$$S_i = S_{i-1} + (a_i + a_{i+1})$$

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

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

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

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

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

Step 3: unrolling 2×2

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

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

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

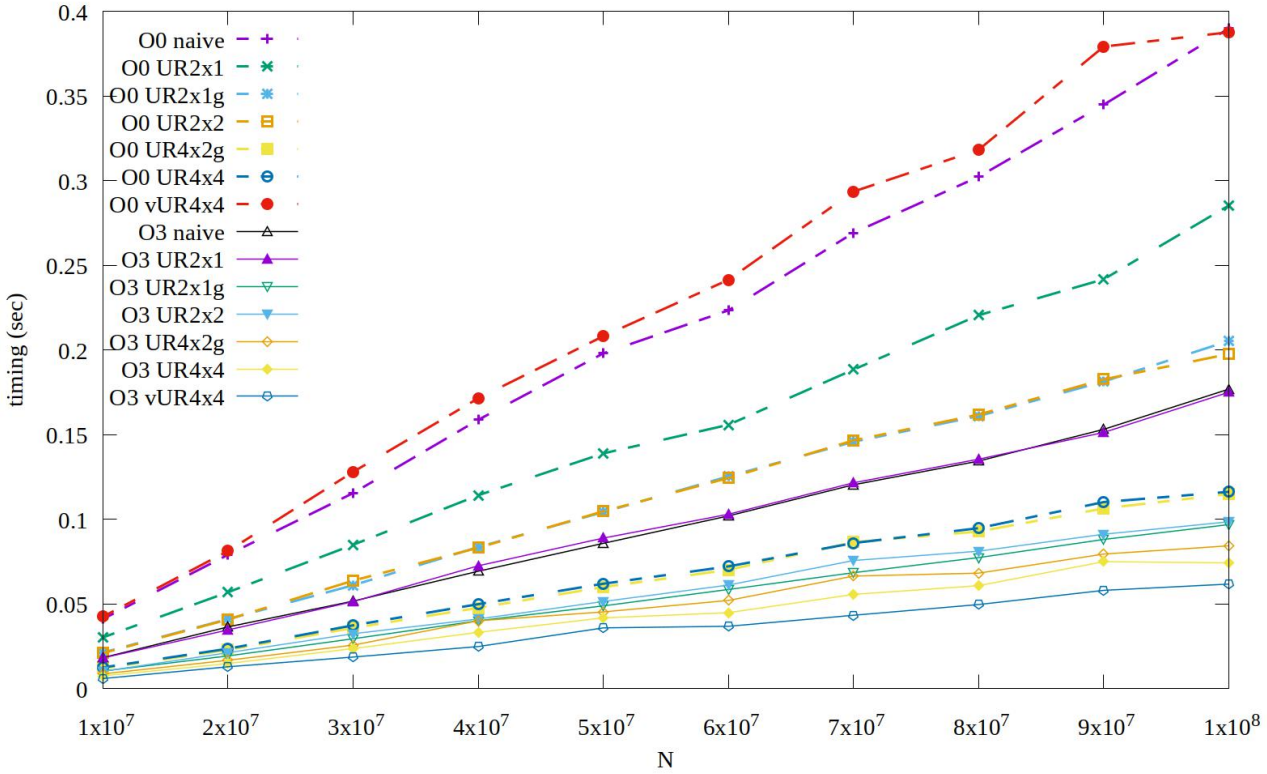
Step 2: unrolling 2×2

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

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

        cmp             rax, rcx
        jne     .L49
```

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

Reduction: results (timing)



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

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

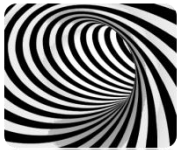
vUR4x4: UR4x4 with explicit vectorization.

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

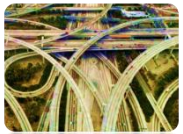
Outline

Avoid the avoidable inefficiencies

Cache & Memory



Loops



Branches

Pipelines

Unleash the Compiler

Don’t loose control

Whenever either *(i)* the sequence of operations that must be executed or *(ii)* the sequence of data to be processed depends on some condition, i.e. on the outcome of a test performed on some data or result, we have a *conditional execution*.

Modern architecture offer 2 distinct low-level instructions to implement a conditional execution upon a test:

- modifying the *control flow* → **data-dependent execution-flow**
- modifying the *data flow* → **data-dependent data-flow**

Let's see the conditional execution flow as first.

At machine level, the way to alter the execution flow is through a **jump instruction,** that causes the control to be passed to a different code section.

The jump instruction can be *conditional*, when its execution depends on the outcome of some operation (a test), or *unconditional* if it is not.

A function call is a jump instruction of particular type, in which we are not interested here.

Low-level control

`jmp` is the only *unconditional* instruction; it accepts either a *direct* destination (specified by a label) or an *indirect* destination (specified through an address in a register or in memory).

`je`, **`jne`** and the others, are *conditional* instructions: i.e. their execution depends on a condition.
These instructions access the values stored in the bits of the flag register, a special register where the CPU inscribes some characteristics outcomes of the last arithmetic or logical operation

`CF`	CARRY FLAG; a carry out of the msb has been generated, signaling an overflow in unsigned op
`ZF`	ZERO FLAG; the most recent op resulted in a zero
`SF`	SIGN FLAG; the most recent operation ended in a negative result
`DF`	OVERFLOW FLAG; a two's-complement overflow, either negative or positive.

This table shows some of the low-level jump instructions routinely available on modern CPUs.

`jmp`	*Label* **Operand*	direct jump indirect jump
`je`	*Label*	jump if equal / zero
`jne`	*Label*	jump if not equal / zero
`js`	*Label*	jump if negative
`jns`	*Label*	jump if not negative
`jg`	*Label*	jump if greater
`jge`	*Label*	jump if greater or equal
`jl`	*Label*	jump if less
...	...	...

Low-level example: for loop

Let's inspect how simple for cycle translates in assembler:

```
for ( int i = 0; i < 10; i++ )
  array[i] = 0;
```

```
.L2:
        mov     DWORD PTR [rax], 0
        add     rax, 4
        cmp     rax, rbp
        jne     .L2
```

`AX` contains the address of `array`

Direct memory access at *i*-th element of `array`

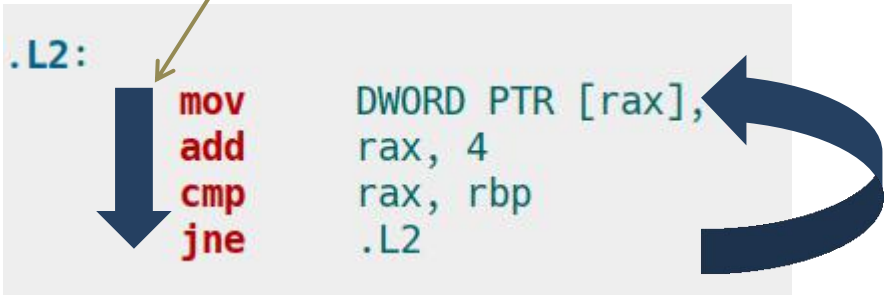
The address to be referenced is increased (equivalent to increase the counter `i`)

The comparison operator; it amounts to `rax-rbp` and sets the flag `ZF`
(note: `rbp` contains the termination value)

`jump if not equal`, i.e. jumps to .L2 if the `ZF` flag is not set

Low-level example: for loop

This is the linear flow of execution in the cycle body



Until `rax <= rbp` (i.e. `i < 10`) there is a *backward jump* and the body is executed again.

When the jump is *not* executed and the control flow continues afterwards.

Low-level example: if statement

```
if ( a < b )
  c = a+b;
else
  c = a-b;
```

```
        mov     eax, DWORD PTR -8[rbp]
        cmp     eax, DWORD PTR -4[rbp]
        jge     .L2
        mov     edx, DWORD PTR -8[rbp]
        mov     eax, DWORD PTR -4[rbp]
        add     eax, edx
        mov     DWORD PTR -12[rbp], eax
        jmp     .L3
.L2:
        mov     eax, DWORD PTR -8[rbp]
        sub     eax, DWORD PTR -4[rbp]
        mov     DWORD PTR -12[rbp], eax
.L3:
```

Note: compiled without optimization, though

Low-level example: if statement

move a in AX

cmp a with b
(amounts to a-b)

check ZF and SF
(zero and sign flag)

```
if ( a < b )
  c = a+b;
else
  c = a-b;
```

```
        mov     eax, DWORD PTR -8[rbp]
        cmp     eax, DWORD PTR -4[rbp]
        jge     .L2
        mov     edx, DWORD PTR -8[rbp]
        mov     eax, DWORD PTR -4[rbp]
        add     eax, edx
        mov     DWORD PTR -12[rbp], eax
        jmp     .L3
.L2:
        mov     eax, DWORD PTR -8[rbp]
        sub     eax, DWORD PTR -4[rbp]
        mov     DWORD PTR -12[rbp], eax
.L3:
```

Jump if a > b

Unconditional jump after the branch a < b

Note: compiled without optimization, though

Low-level example: if statement

Note:

The true branch is the closest to the test condition, while the `false` branch is reached upon a jump.

→ when coding, if possible pay attention to what is most likely to be true, to preserve the **code locality**.
(it is possible to suggest to compiler which branch will most probably be true)

```
  c = a+b;
else
  c = a-b;
```

```
        mov     eax, DWORD PTR -8[rbp]
        cmp     eax, DWORD PTR -4[rbp]
        jge     .L2
        mov     edx, DWORD PTR -8[rbp]
        mov     eax, DWORD PTR -4[rbp]
        add     eax, edx
        mov     DWORD PTR -12[rbp], eax
        jmp     .L3
.L2:
        mov     eax, DWORD PTR -8[rbp]
        sub     eax, DWORD PTR -4[rbp]
        mov     DWORD PTR -12[rbp], eax
.L3:
```

Note: compiled without optimization, though

Conditional data flow

We have seen some details about the conditional transfer of the *control flow* through the simple jump mechanism.
However, that could be quite inefficient in modern CPUs (we'll se more details on that when dealing with the pipelines).

A different mechanism is to **conditionally change the *data flow***, which is the second mechanism for conditional execution that we mentioned.

Conditional data flow

The conditional transfer of data flow yields a very high performance but is possible only on a small subset of cases;

basically those are when simple values are involved.

A typical example is, for instance, the absolute value of a result, or something alike where a single-valued outcome is expected.

```
if ( a < b )
  c = a+b;
else
  c = a-b;
```

Here `c` holds the result of a very simple arithmetic operation between `a` and `b`.

Conditional data flow

```
if ( a < b )
  c = a+b;
else
  c = a-b;
```

Compiled with
`gcc -O3 -march=native`

note:
ebx = a
eax = b

```
mov     edx, ebx
lea     ecx, [rbx+rax]
sub     edx, eax
cmp     eax, ebx
cmovg   edx, ecx
```

A **much** shorter and efficient code!

Conditional data flow

	eax	ebx	ecx	edx
`eax = b, ebx = a`	**b**	**a**	**-**	**-**
`mov edx, ebx`	**b**	**a**	**-**	**a**
`lea ecx, [rbx+rax]`	**b**	**a**	**a+b**	**a**
`sub edx, eax`	**b**	**a**	**a+b**	**a-b**
`cmp eax, ebx`	*compares a and b; sets ZF and CF (zero and carry flag)*			
`cmovg edx, ecx`	move ecx's value to edx if the condition (greater than) is satisfied			

The strategy is as follows:

(i) reg ecx contains a+b while reg edx contains a-b

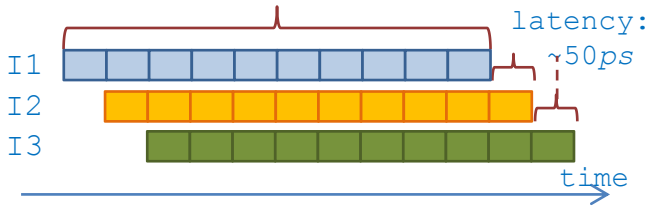
(ii) the conditional move checks the result of cmp a, b (i.e. the value of a-b)

(iii) the content of edx is changed into ecx's just in case.

No jump instructions have been issued.

The perils of conditional branches

As we have seen in the lecture about modern architecture, **modern processors achieve great performance thanks to the *pipelines* and *out-of-order execution***, i.e. by decomposing complex instructions in simpler steps and mixing the execution of those sub-steps for different instructions (up to tens of instructions at the same time may be “on-the-fly” in modern CPUs).



We’ll see more about pipelines in forthcoming lectures

That, however, **requires the pipelines to be always full;** if not so, the toll of great penalties in terms of wasted cpu cycles is to be payed.
To achieve this goal, it is in turn mandatory for the scheduler to be able to **predict** in advance what will be the sequence of instructions to be executed.
How can that be in a world full of possibilities and branches ?

The perils of conditional branches

In order to predict what the execution flow will be, modern cpus feature a **branch predictor,** that is an internal unit of highly sophisticated logic that guesses whether a jump instruction will succeed or not.

Best branch predictors are as good as 95% of accuracy; nonetheless, the branch mis-prediction, or branch miss, determines a huge performance loss.
Typical figures for penalty are 10-30 cycles!

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

Example
Let's say we have 140 instructions in flight, and 1 every 7 is a branching instructions. What is the probability that the pipelines shall ***not*** *be flushed with 95% correct branch predictor? And with a 90% one?*

answer: ~36% and ~12%

The perils of conditional branches

In order to predict what the execution flow will be, modern cpus feature a **branch predictor,** that is an inter
a jump instruction will su

Best branch predictors are a
prediction, or branch miss, c
penalty are 10-30 cycles!

That is because the longer t
the flow, the more difficult i

Example
Let's say we have 140 instruction
probability that the pipelines shall
one?

That is why the 2nd strategy we have seen, the **conditional change of data flow** is preferable whenever possible and the compiler will try to use it as much as possible:

it generates no jump instructions and the execution flow is linear and perfectly predictable.

However, as we said, it applies only on a limited sub-set of case.

ex. 1: about the fact that design and simplicity are the best move

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];
  }
```

Revise your code

Consider the following code snippet(*)

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

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

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

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

Revise your code

You 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++)
  {
     sum += ( data[ii]>PIVOT ) * data[ii];
   //sum += ( data[ii]>PIVOT ? data[ii] : 0);
  }
```

Revise your code

-O0

```
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%
```

-O3

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

sum is 983597794767, elapsed seconds: 0.660148

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

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

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

sum is 983597794767, elapsed seconds: 0.679286

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

Revise your code

-O3

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

sum is 983597794767, elapsed seconds: 0.660148

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

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

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

sum is 983597794767, elapsed seconds: 0.679286

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

-O3
-march=native

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

sum is 983597794767, elapsed seconds: 0.217864

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

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

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

sum is 983597794767, elapsed seconds: 0.224288
```

ex 2: code restructuring for sorting two arrays

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

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

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

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

Revise your code

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

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

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

Revise your code

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

standard

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

smart2

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

smart

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

smart3

predictable data has a regular pattern easily spotted by the CPU's branch predictor

Standard, -O0

```
1000000000 predictable numbers generation..  done

seconds           :    2.223 +- 0.004446
CPE               :          9.938 +- 0.0138
IPE               :     27.0001758 +- 3.036e-07
BRANCHES_PE       :    2.00000014 +- 6.585e-10
MISP_BRANCHES_PE  :     4.892e-07 +- 2.134e-08  ( tot:          244.6 )


1000000000 random numbers generation..  done

seconds           :    4.501 +- 0.02864
CPE               :         27.4 +-   0.2
IPE               :    26.9996208 +- 2.98e-08
BRANCHES_PE       :     2.00000014 +- 6.585e-10
MISP_BRANCHES_PE  :    0.499933228 +- 2.215e-05  ( tot:     249966614 )
```

loop run-time

cycles per element

Instructions per element

~2.7 IPC

~1 IPC

conditional branches pert element

mis-predicted cond. branches per el.

total # of mis-predicted conditional branches

same number of IPE but almost 3 times as many CPE when data pattern is not predictable

in fact, there is 1 mis-predicted branch every 2 (arrays are 500,000,000 long)

Revise your code

smart, -O0

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

~3 IPC

predictable

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

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

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

~3 IPC

random

Revise your code

smart2, -O0

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

~3 IPC

predictable

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

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

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

~3 IPC

random

Change the point of view

ex. 3: about the fact that design and simplicity are the best choice

Just changing point of view sometimes may help:

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

Change the point of view

Can easily be re-written with no conditional evaluations at all:

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

       matrix[j][i] = 0.0;

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

that's all, have fun