Modern Architectures & Optimization

Luca Tornatore - I.N.A.F.

"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING
2019-2020 @ Università di Trieste

Outline

Introduction

Modern Architecture

Optimization

Some preliminary concepts

High Performance Computing requires, by the name itself, to squeeze the maximum effectiveness from your code on the machine you run it.

“Optimizing” is, obviously, a key step in this process.

However, both “*optimizing*” and “*effectiveness*” have to be defined.
Let’s start with some general comments.

There may be no such thing as an “optimal” code *tout court*

Just try to define “optimal”..

MEMORY CONSTRAINTS
memory imprint (data & code)

ENERGY CONSTRAINTS

I/O CONSTRAINTS
carefully crafting I/O
(disk, memory, network,..)

"OPTIMIZATION"

TIME CONSTRAINTS
time-to-solution
(worst case? optimal case? "average case"?)

ROBUSTNESS/RESILIENCE
mission-critical

Premature optimization is the root of all evil

D. Knuth

You'd better start thinking in terms of "***improved***" code.

- You don't want to hurt your code that honestly does its job: a faster code that gives incorrect results is not optimal in *any* way
- You most probably will have to re-use that code after some time. And you will need it to be *readable*, *maintainable* and *well-designed*.
- *Others* most probably will have to read, understand and re-use that code: a clean, understandable, non-obscure, non-redundant code may not be "improved" in some way, but actually improves the quality of your life.

First steps to consider

To have a **cleaner code**

To **re-design** its fundamental architecture

To **improve** the workflow, the memory imprint, the resource usage, the time-to-solution, …

Dryness

DRY: Don't Repeat Yourself

Do neither add **unnecessary** code nor **duplicate** code

Unnecessary code increases the amount of needed work

- to maintain the code, either debugging or updating it
- to extend its functionalities

Duplicated code increases your *bad technical debt*, that already has a large enough number of sources:

- copy-and-paste programming style
- too much *agile* approach
- hard coding, quick-and-dirty fixes, cargo-cult, sloppiness

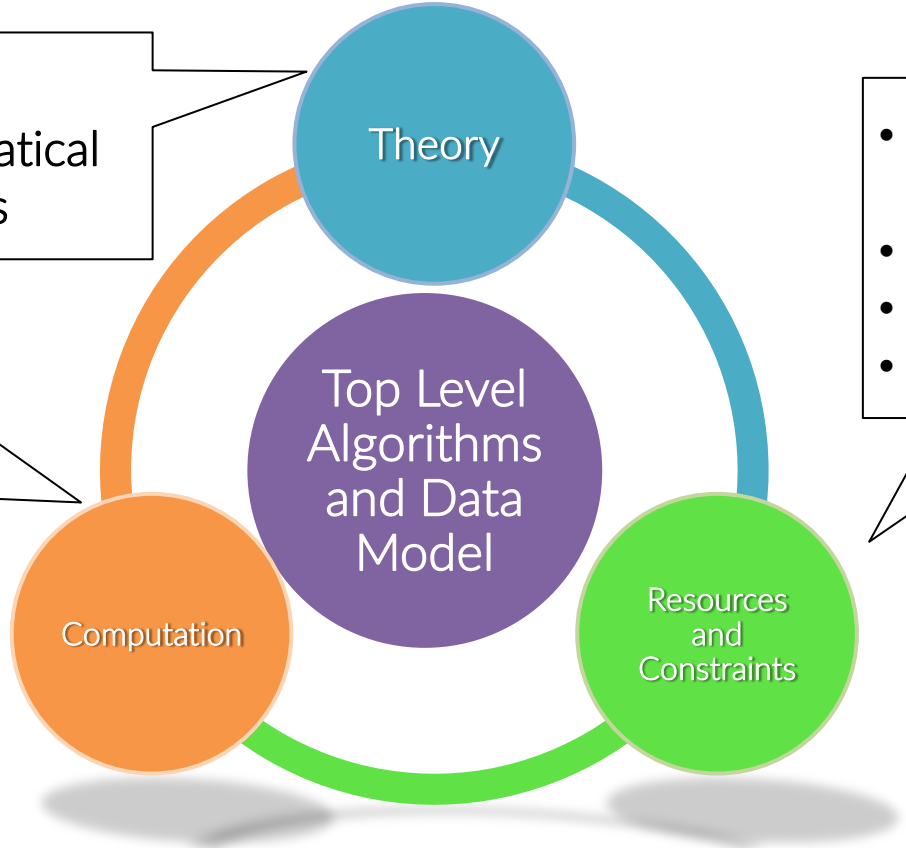
Readability is a pillar of **maintainability**.
Maintainability means that software can be extended, upgraded, debugged, fixed.

- **Baseline**: *write comments*
- **Advanced**: *WRITE COMMENTS !* (possibly, use doxygen-like tools)
- **X-advanced**: avoid stupid, obvious, useless, confusing comments

Keep in mind: "*the code is the ultimate man page*"

Design

- Consult literature
- Understand mathematical and scientific aspects



- Concurrency & parallelism
- Memory constrs.
- I/O constrs.
- Energy constrs.

You're not at the blackboard, but on a digital discrete system. Equations have to *translated*.

Design

ARCHITECTURAL DESIGN

Highest abstract picture:
- Interacting multi-components
- Top-level DATA MODEL

HIGH-LEVEL DESIGN

Sub-systems and modules Communication strategies

Modules as producers of complex tasks

DETAILS DESIGN

Implementation details
- many algorithms
- concurrency

Logical structure of modules

interactions

Very low-level coding and details

ABSTRACTION

CODING DETAIL

TESTING IS part of the design

- Unit test
  separately stress each unit of the code
- Integration test
  stress the integrated behavior of all the units together
- System test

VALIDATION and VERIFICATION ARE part of the design

- Validation ensures that the code does what it was meant to do
  (all modules are designed accordingly to design specifications)
- Verification ensures that the codes does what it does *correctly*
  (black-box testing against test-cases, …)

Improve the code

Improving the workflow, the memory imprint, the resource usage, the time-to-solution, …

Well, that is the focus of the next lectures

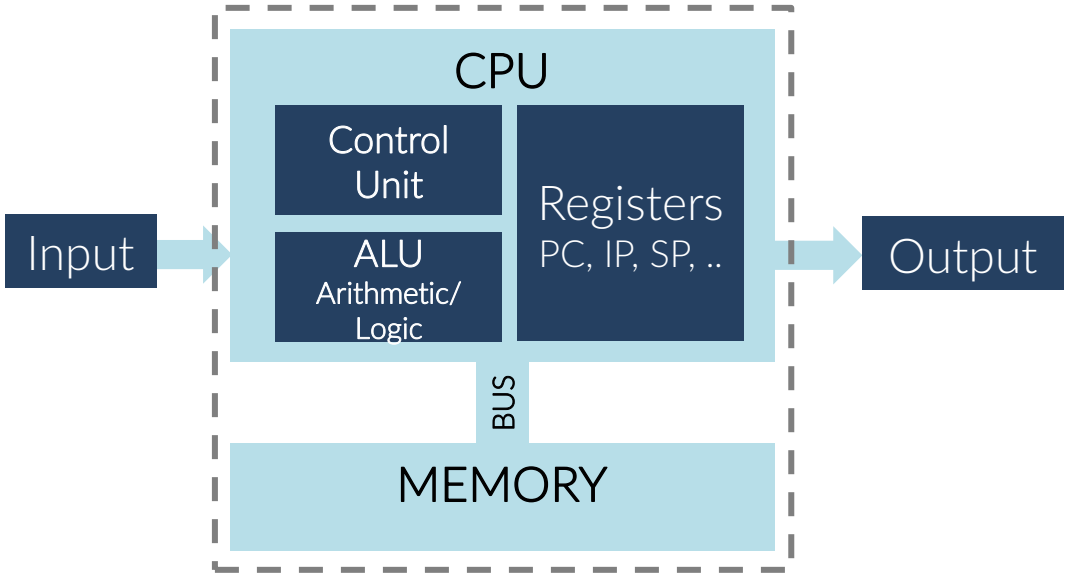
Outline

Modern Architecture

Optimization

Once upon a time..

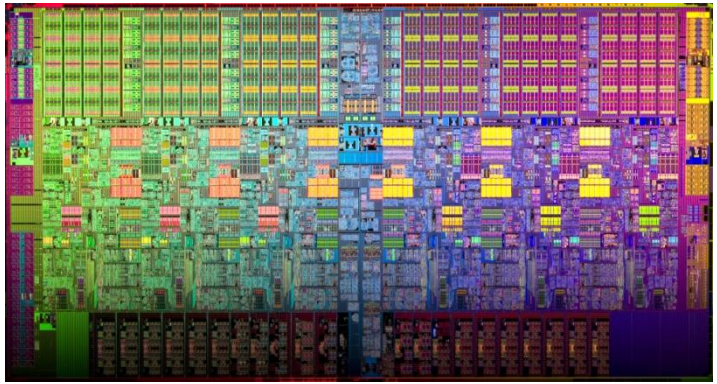
.. The computer architecture was much simpler than today.
In the lecture “The Free lunch is over”, we have seen the reasons why it was so – or, better, why nowadays it is much more complex.



In the Von Neumann architecture (still taught as the fundamental model)

- there is only 1 processing unit
- 1 instructions is executed at a time
- memory is “flat”:
  - access on any location has always the same cost
  - access to memory has the same cost than op execution

While today…



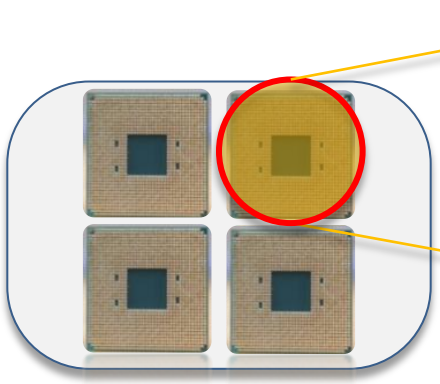
NOT an extreme example: 6-cores Intel Xeon e5600

(*) "cost" is in the CPU-cycles currency

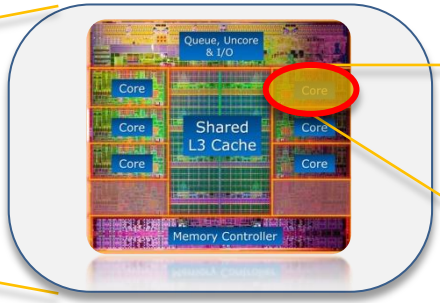
- there are *many* processing units
- many instructions can be executed at a time
- many data can be processed at a time
- "instructions" are internally broken down in many simpler µops that are pipelined
  - different instructions could have quite different cost (*)
- memory is strongly *not* "flat":
  - there is a strong memory hierarchy (*)
  - access memory can have *very* different costs (*) depending on the location
  - accessing RAM is *way* more costy than performing a µop or reading an internal register

In the next slides we'll go through all this features in chronological order, as they developed in time

Recap (you've seen more details in other lectures)



sockets



Many cores per socket

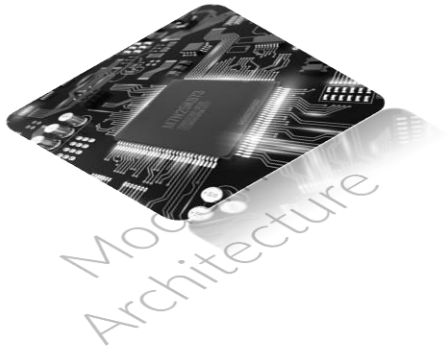
Many threads

Vectorisation

Pipelining

Superscalarity

Outline

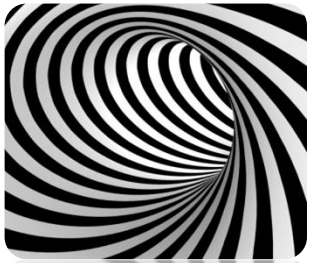


Optimization

Outline

First things first

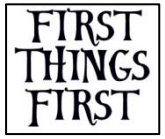
Cache & Memory



Loops

Branches

Pipelines



First things first

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

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

Top-level design plays a key role, as well as algorithm choice and implementation details.

Sometimes, but only sometimes, even a single line of code – for instance a prefetching – may have brilliant consequences

...

FIRST THINGS FIRST

Let the compiler do his job

As a general guideline just keep in mind that “optimization” reads

“let the compiler squeeze the maximum from your code”

Compilers are quite good indeed, and have a deep insight on the hardware they are running on.

So, as first, just learn how to :

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

FIRST THINGS FIRST

| Write non-obfuscated code

- write non-obfuscated code
  → avoid memory aliasing
  → make it clear what a variable is used for and when
  → take care of your loops
  → keep your conditional branches under control

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

FIRST THINGS FIRST

| Your data are the red pill

- write non-obfuscated code
- design a good data structure layout
  → be cache-friendly (but oblivious)
  → what is used together, stays together
  → be NUMA-conscious
  → avoid false-sharing in multi-threaded cores..
- design a “good” workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

FIRST THINGS FIRST

Let it flow

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

FIRST THINGS FIRST

May the force be with you

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

FIRST THINGS FIRST

Some C-specific hints

Optimization level: O*n*

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

Take into accounts that modern compilers allow for local specific optimizations or compilation flags.

In gcc for instance:

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

FIRST THINGS FIRST

Some C-specific hints

Optimization level: native

The compiler knows the architecture it is compiling on, of course. However, it will generate a *portable* code , i.e. a code that can run on *any* cpu belonging to that class of architecture.
Example: x86_64, x86_32, ARM, POWER8, are all classes of architecture.
Besides a general set of instructions that all the cpus of a given class can understand, specific models have specific different ISA that are not compatible with others (normally you have back-compatibility).
Using the switch `-march=native` it will optimize for exactly the specific cpu it's running on, much probably producing a more performant code for it.

FIRST THINGS FIRST

Some C-specific hint

Storage classes

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

FIRST THINGS FIRST

Some C-specific hints

Variable qualifiers

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

FIRST THINGS FIRST

Some C-specific hints

Focus on the *restrict* qualifier

→ code snippet ::

`memory_aliasing_1/`
`memory_aliasing_2/`

It's time to play a bit

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

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

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

Focus on the memory aliasing

What happens on your laptops ?

Without compiler opt.

With compiler opt.

```
[ltornatore@hp11 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) CPU E5-4627 v3 @ 2.60GHz
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0295128 (sigma^2: 1.27277e-10)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294134 (sigma^2: 3.97265e-12)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.031259 (sigma^2: 1.79355e-10)
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a.03
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00923895 (sigma^2: 6.87404e-11)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b.03
aliasing will be explicitly excluded
Ù
 - averaging best 10 of 30 iterations - 0.00932974 (sigma^2: 3.42719e-11)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00629554 (sigma^2: 3.2011e-09)
[ltornatore@hp11 aliasing]$
```

```
[ltornatore@gen10-01 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294641 (sigma^2: 3.63278e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0292034 (sigma^2: 7.0978e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0323442 (sigma^2: 9.72394e-08)
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a.03
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00825699 (sigma^2: 5.46725e-10)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00830632 (sigma^2: 2.81407e-11)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.005908 (sigma^2: 1.04317e-09)
[ltornatore@gen10-01 aliasing]$
```

Focus on the memory aliasing

What happens on your laptops ?

Memory aliasing not excluded

Memory aliasing excluded explicitly

Memory aliasing excluded explicitly and memory aligned explicitly

```
[ltornatore@hp11 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) CPU E5-4627 v3 @ 2.60GHz
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0295128 (sigma^2: 1.27277e-10)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294134 (sigma^2: 3.97265e-12)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.031259 (sigma^2: 1.79355e-10)
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a.03
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00923895 (sigma^2: 6.87404e-11)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b.03
aliasing will be explicitly excluded
ù
 - averaging best 10 of 30 iterations - 0.00932974 (sigma^2: 3.42719e-11)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00629554 (sigma^2: 3.2011e-09)
[ltornatore@hp11 aliasing]$
```

```
[ltornatore@gen10-01 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294641 (sigma^2: 3.63278e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0292034 (sigma^2: 7.0978e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0323442 (sigma^2: 9.72394e-08)
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a.03
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00825699 (sigma^2: 5.46725e-10)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.00830632 (sigma^2: 2.81407e-11)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c.03
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.005908 (sigma^2: 1.04317e-09)
[ltornatore@gen10-01 aliasing]$
```

Focus on the memory aliasing

What happens on your laptops ? While the first example goes smoothly, all the other versions, or at least the first two, perform equally.

```
[ltornatore@hp11 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) CPU E5-4627 v3 @ 2.60GHz
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0295128 (sigma^2: 1.27277e-10)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294134 (sigma^2: 3.97265e-12)
[ltornatore@hp11 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.031259 (sigma^2: 1.79355e-10)
[ltornatore@hp11 aliasing]$
```

```
[ltornatore@gen10-01 aliasing]$ cat /proc/cpuinfo | grep -m1 "model name"
model name      : Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_a
aliasing won't be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0294641 (sigma^2: 3.63278e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_b
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0292034 (sigma^2: 7.0978e-07)
[ltornatore@gen10-01 aliasing]$ ./pointers_aliasing_c
aliasing will be explicitly excluded

 - averaging best 10 of 30 iterations - 0.0323442 (sigma^2: 9.72394e-08)
[ltornatore@gen10-01 aliasing]$
```

Let's check the generated assembly to understand this behaviour

```
 - averaging best 10 of 30 iterations - 0.00629554 (sigma^2: 3.2011e-09)
[ltornatore@hp11 aliasing]$
```

```
 - averaging best 10 of 30 iterations - 0.005908 (sigma^2: 1.04317e-09)
[ltornatore@gen10-01 aliasing]$
```

FIRST THINGS FIRST

Focus on the memory aliasing

```
937                         .globl  add_float_array
939                     add_float_array:
947                     # pointers_aliasing_a.c:129:   for ( int i = 0; i < N; i++ )
129:pointers_aliasing_a.c ****     C[ i ] += A[ i ] + B[ i ];
949 0060 85FF                  test    edi, edi        # N
950 0062 0F8E1801              jle     .L36    #,
951 0068 4C8D4110              lea     r8, 16[rcx]     # tmp156,
952 006c 4C8D5610              lea     r10, 16[rsi]    # _31,
953 0070 4C39C6                cmp     rsi, r8 # C, tmp156
954 0073 8D47FF                lea     eax, -1[rdi]    # _33,
955 0076 410F93C1              setnb   r9b     #, tmp158
956 007a 4C39D1                cmp     rcx, r10        # B, _31
957 007d 410F93C0              setnb   r8b     #, tmp160
958 0081 4509C1                or      r9d, r8d        # tmp161, tmp160
959 0084 4C8D4210              lea     r8, 16[rdx]     # tmp162,
960 0088 4C39C6                cmp     rsi, r8 # C, tmp162
961 008b 410F93C0              setnb   r8b     #, tmp164
962 008f 4C39D2                cmp     rdx, r10        # A, _31
963 0092 410F93C2              setnb   r10b    #, tmp166
964 0096 4509D0                or      r8d, r10d       # tmp167, tmp166
965 0099 4584C1                test    r9b, r8b        # tmp161, tmp167
966 009c 0F84AE00              je      .L38    #,
967 00a2 83F802                cmp     eax, 2  # _33,
968 00a5 0F86A500              jbe     .L38    #,
969 00ab 4189F8                mov     r8d, edi        # bnd.78, N
970 00ae 31C0                  xor     eax, eax        # ivtmp.105
971 00b0 41C1E802              shr     r8d, 2  #,
972 00b4 49C1E004              sal     r8, 4   # _110,

976                     .L39:
980 00c0 0F100402              movups  xmm0, XMMWORD PTR [rdx+rax]     # MEM[base: A_16(D)]
981 00c4 0F100C01              movups  xmm1, XMMWORD PTR [rcx+rax]     # MEM[base: B_17(D)]
984 00c8 0F101406              movups  xmm2, XMMWORD PTR [rsi+rax]     # MEM[base: C_15(D)]
```

The compiler is good enough to understand that it could generate 2 different loops: one for the case in which there is memory overlap and a different one for the case in which there is not.

The second loop is very similar to what it generates if you tell him so through the *restrict* keyword.

FIRST THINGS FIRST

Some C-specific hints

Memory allocation

We have seen this in detail in the lecture about memory allocation.

Try to allocate **contiguous memory** and to **re-use it efficiently** avoiding fragmentation

FIRST THINGS FIRST

Some C-specific hints

Profile-guided optimization

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

For **gcc**:

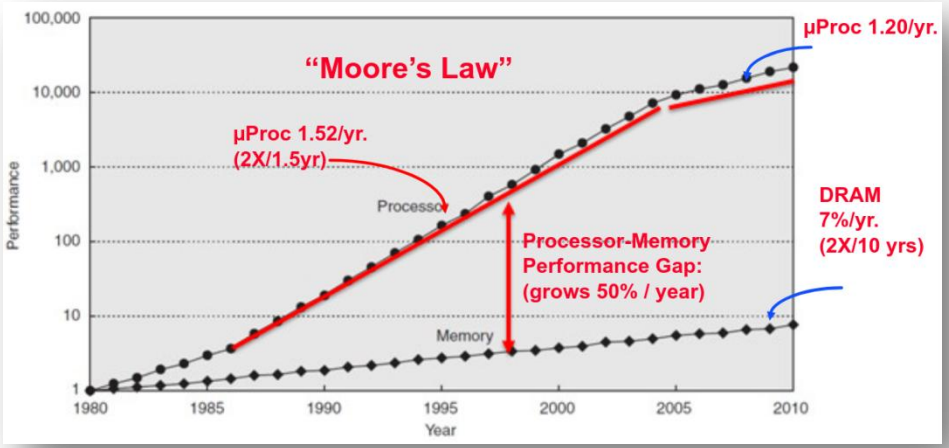
```
gcc -fprofile-arcs
< … run … >
gcc -fbranch-probabilities
```

Specific for branch prediction

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

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

Early 90s: CPU get faster than memory

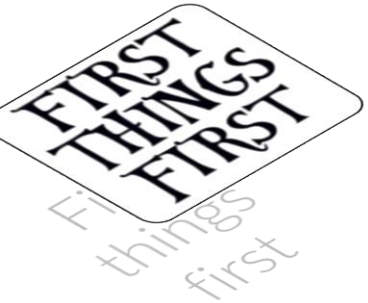


Processor	Alpha 21164	
Machine	AlphaServer 8200	
Clock Rate	300 MHz	
Memory Performance	Latency	Bandwidth
I Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
D Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
L2 Cache (96KB on chip)	20 ns (6 clocks)	4800 MB/sec
L3 Cache (4MB off chip)	26 ns (8 clocks)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

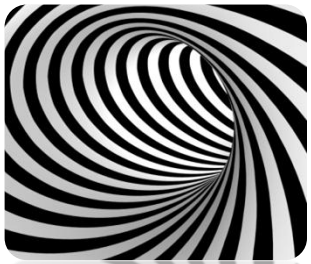
Taken from a 1997 paper

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 ?

Outline



Cache & Memory



Loops

Pipelines

Branches

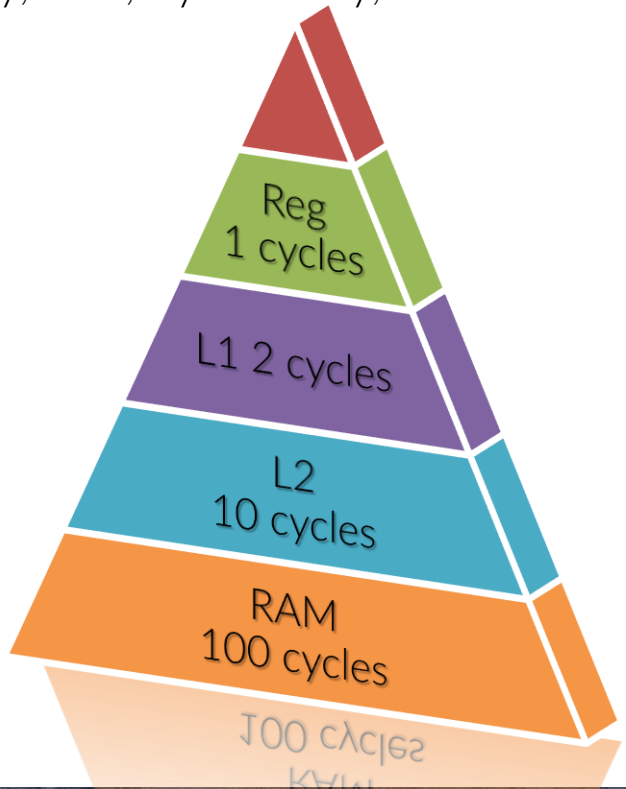
The cache memory

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.



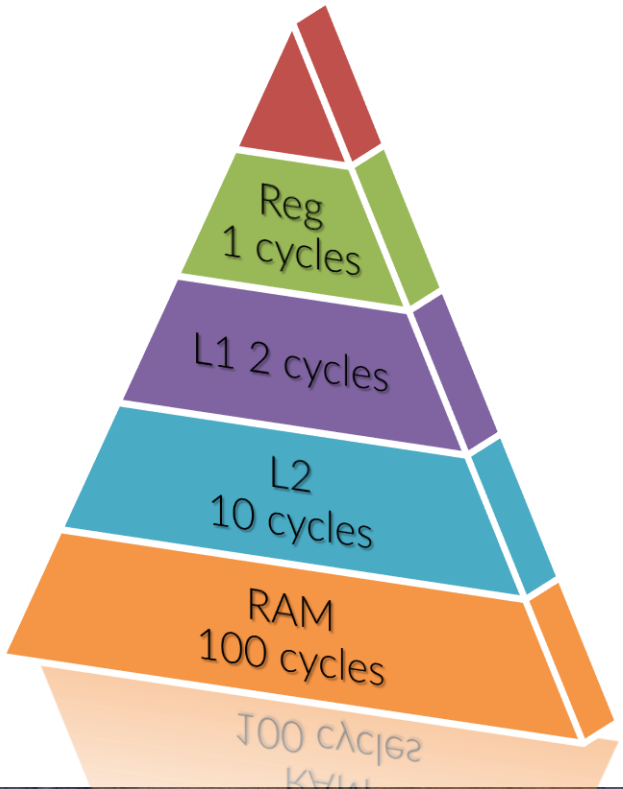
The cache memory

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

L1 cache + RAM

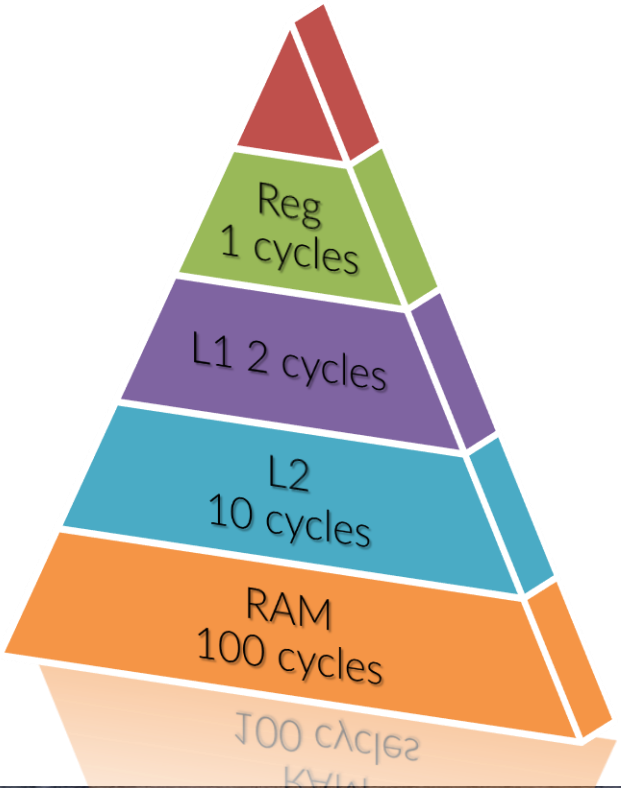
$L_{1cost} + Miss_1 \times RAM_{cost}$

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

L1 cache + L2 cache + RAM

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

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



The cache memory

L1 cache + RAM

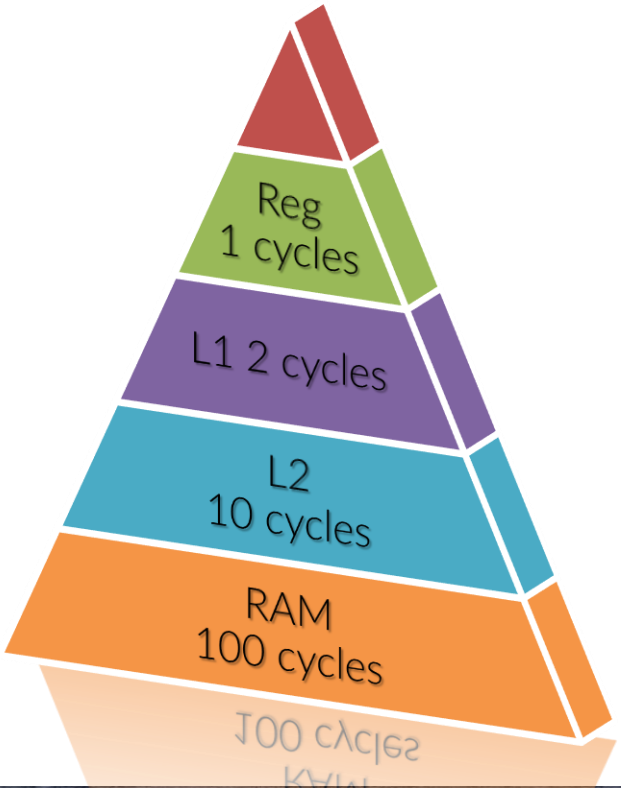
$L_{1cost} + Miss_1 \times RAM_{cost}$

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

L1 cache + L2 cache + RAM

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

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



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

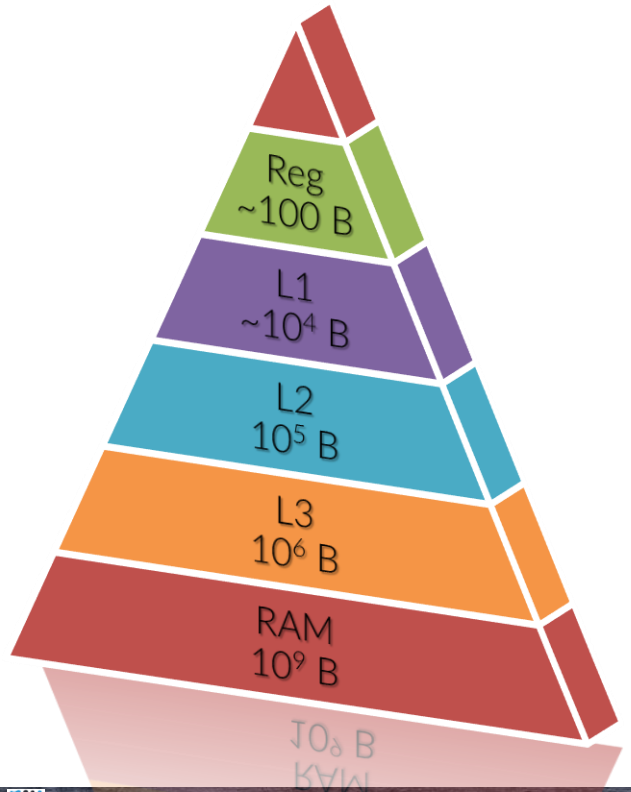
Temporal locality	if an address is referenced, it is likely to be referenced again soon
Spatial locality	if an address is referenced, close addresses are likely to be referenced soon

Cache mapping

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

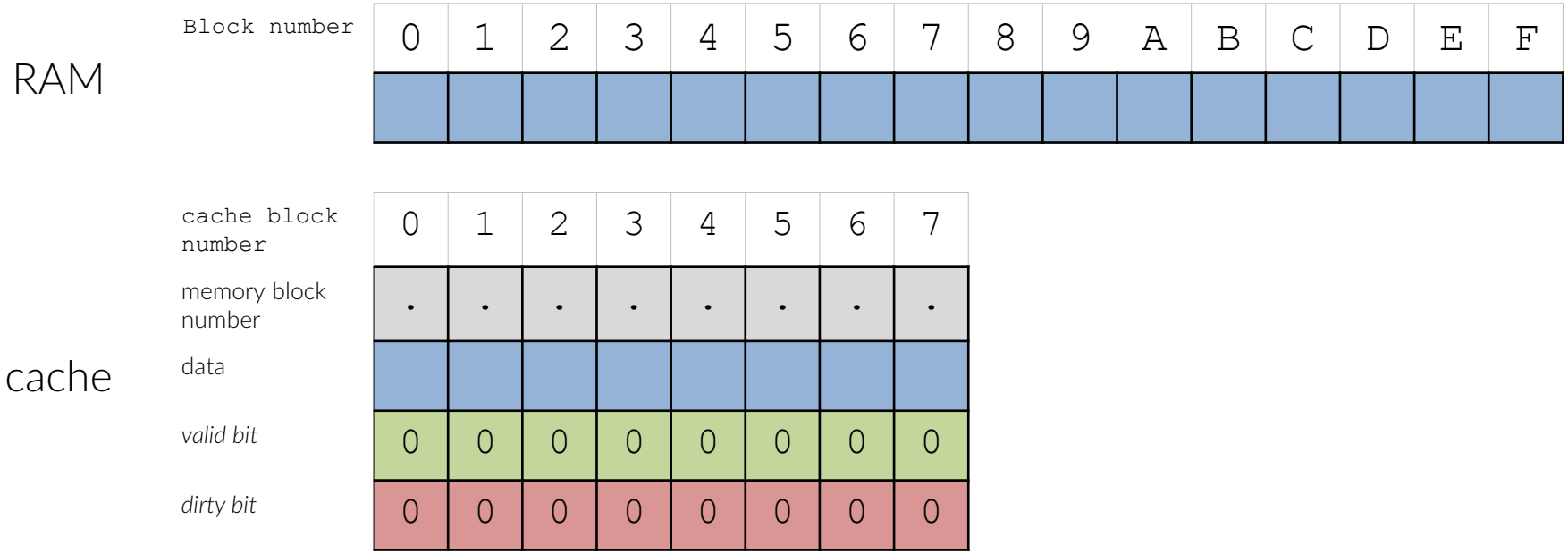
So, how do you map the RAM in to a given level of cache, for instance L1, in an effective way?
The main problems are:

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



Cache mapping

Let's say that both the RAM and the cache are subdivided in blocks of equal size (for instance, 64B),



Cache mapping

Full mapping

Data can be placed in any free cache block

Direct mapping

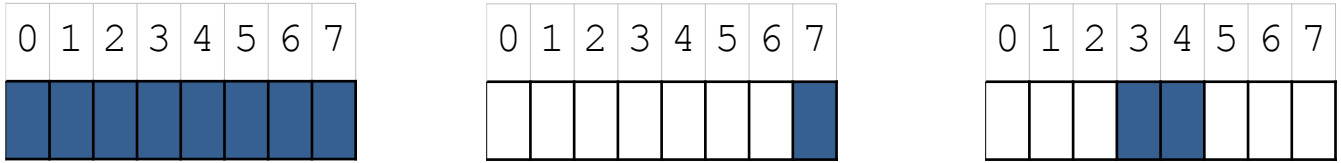
Data can be placed only in a given block i.e. **`block_num % blocks_in_cache`**

n-way associative

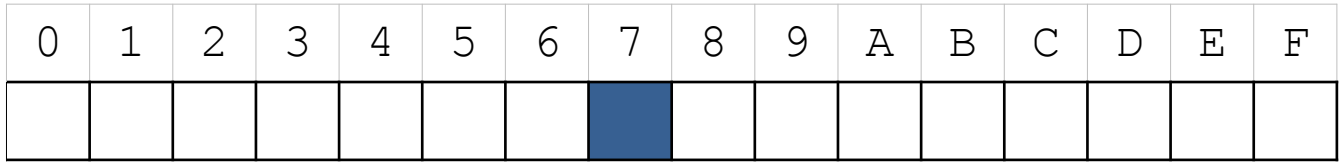
Data can be placed in few cache blocks i.e. **`block_num % (blocks_in_cache / n)`**

n=2

cache



RAM



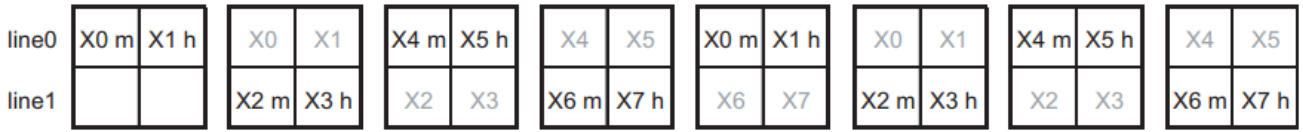
The memory access pattern

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]);
```

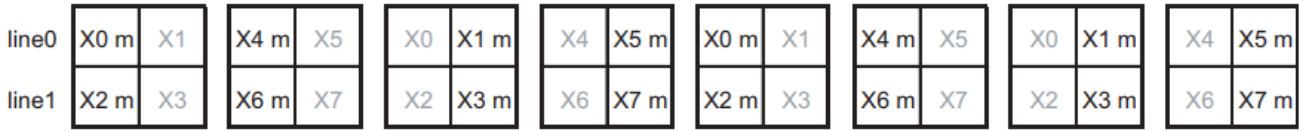


The hit-miss pattern is : MH MH MH MH MH MH MH MH,
the miss-rate is 50% (the first miss is compulsory miss).

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 ]);
```

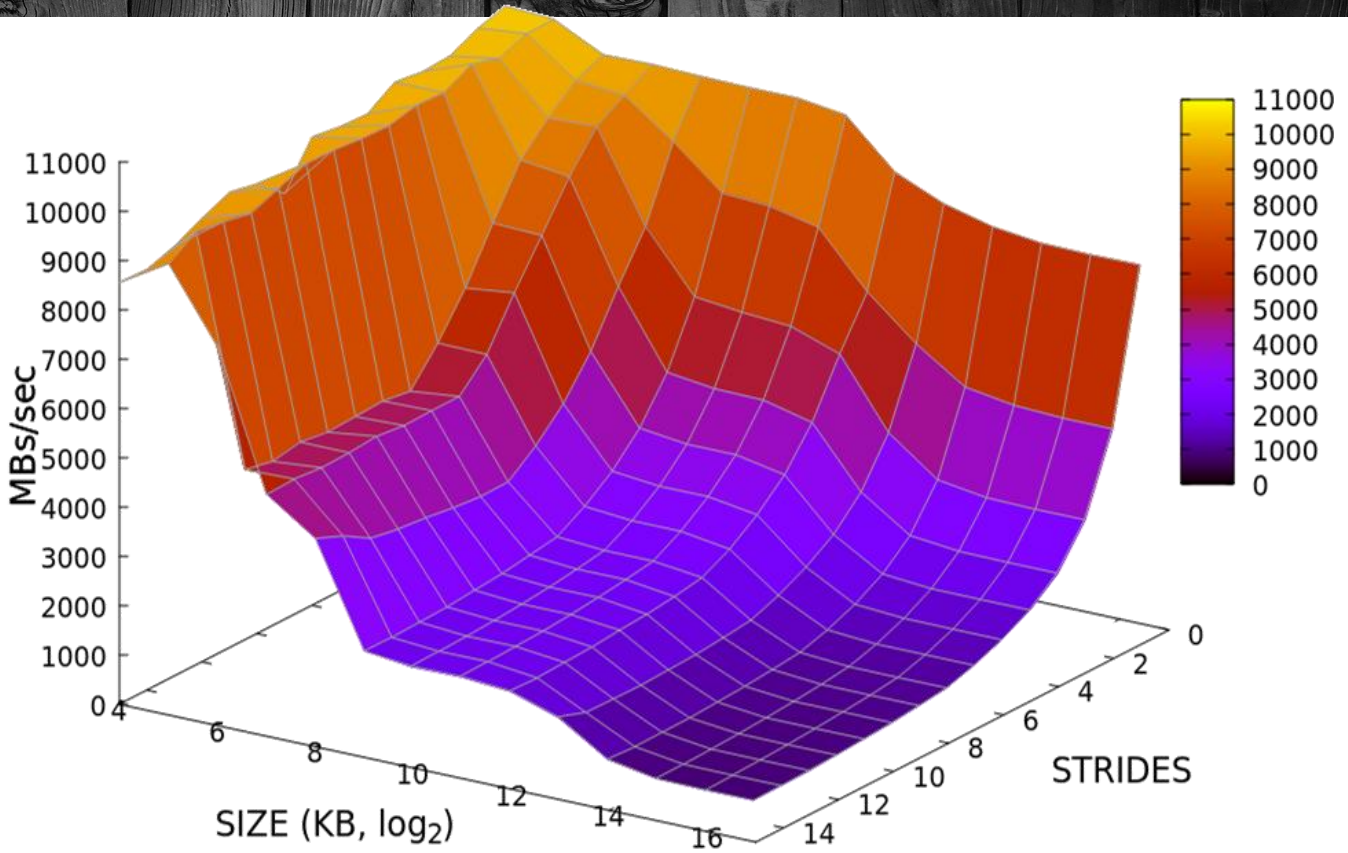
line0	X0 m	X1 h	X0	X1	X0 h	X1 h	X0	X1	X4 m	X5 h	X4	X5	X4 h	X5 h	X4	X5
line1			X2 m	X3 h	X2	X3	X2 h	X3 h	X2	X3	X6 m	X7 h	X6	X7	X6 h	X7 h

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

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

The memory access pattern

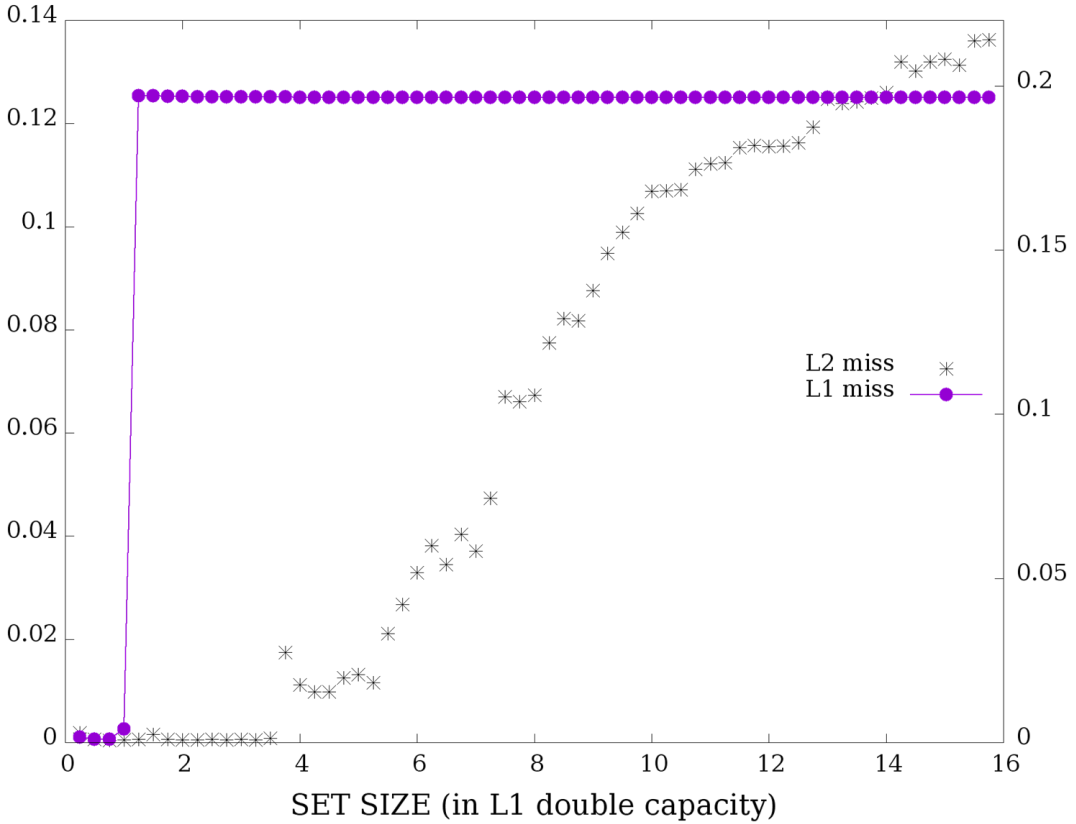
The result is..
the memory mountain



The cache-miss signature

Let's find out our cache size

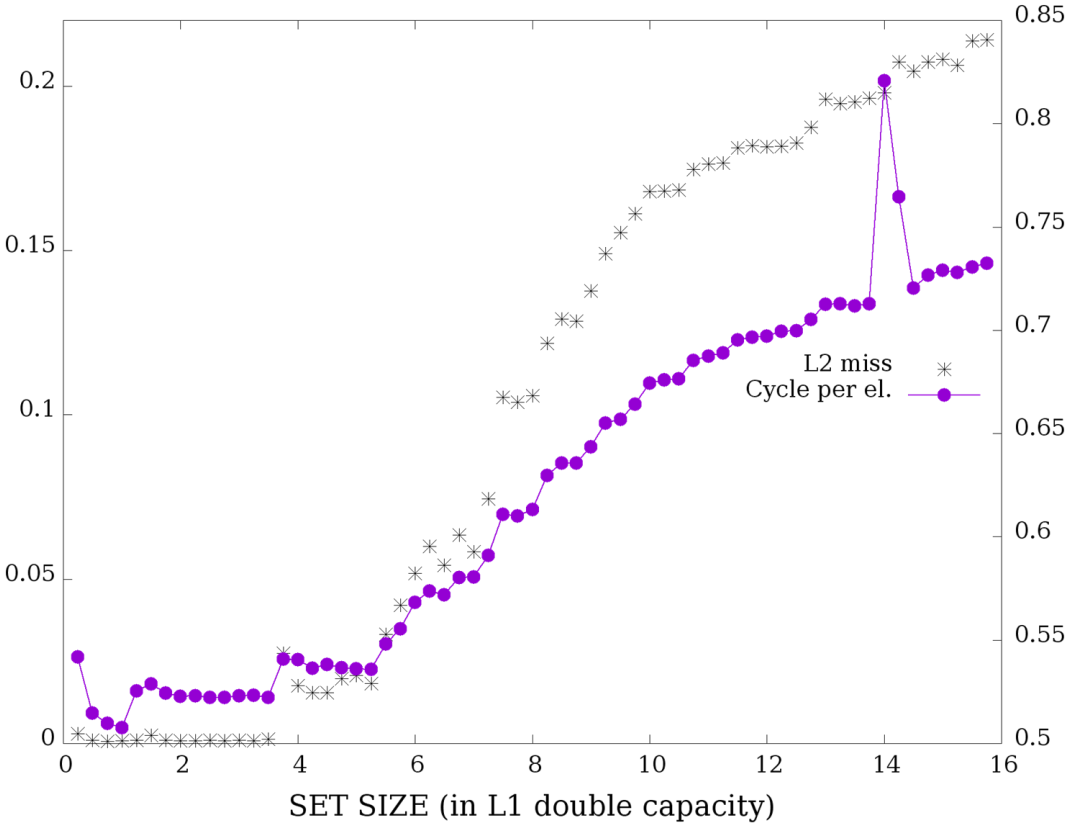
```
for (j=0; j < size; j++)
   array[j] =
     2.3*array[j]+1.2;
```



The cache-miss signature

And the effect on cycles-per-operation metrics

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



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* ?

Strided access

Matrix transpose

Naïve version:

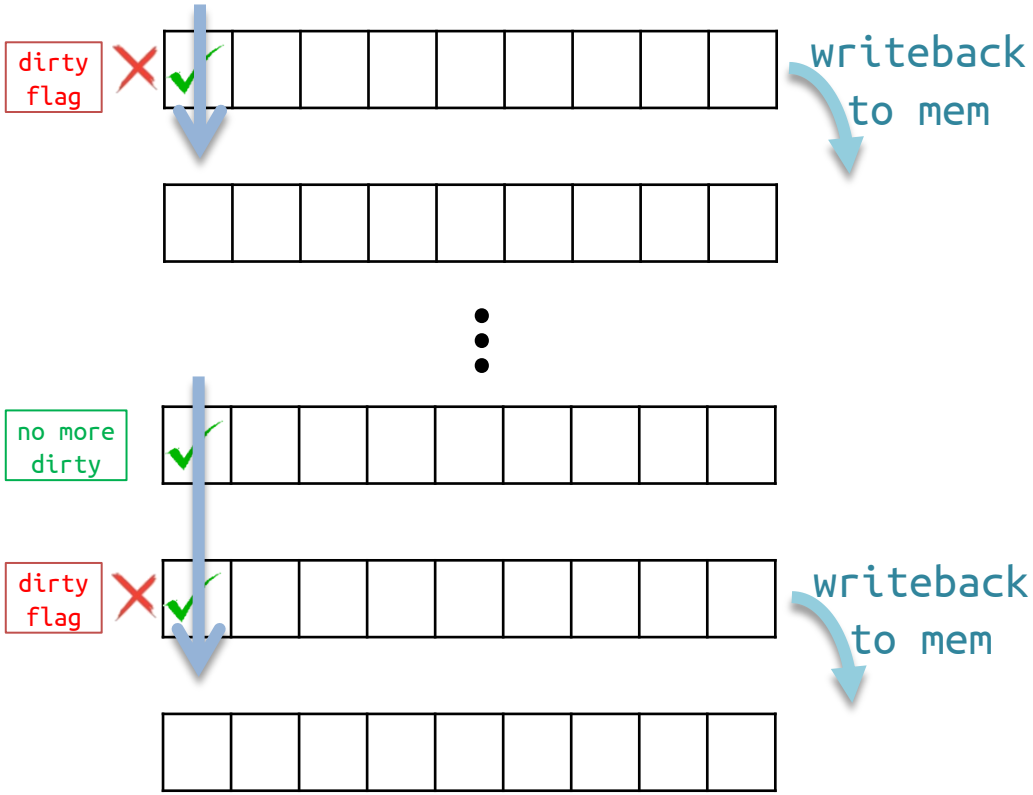
```
for(int row = 0; row < N; row++)
    for(col = 0; col < N; col++)
        A [ col*N + row ] = B [ row*N + col ];
```

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

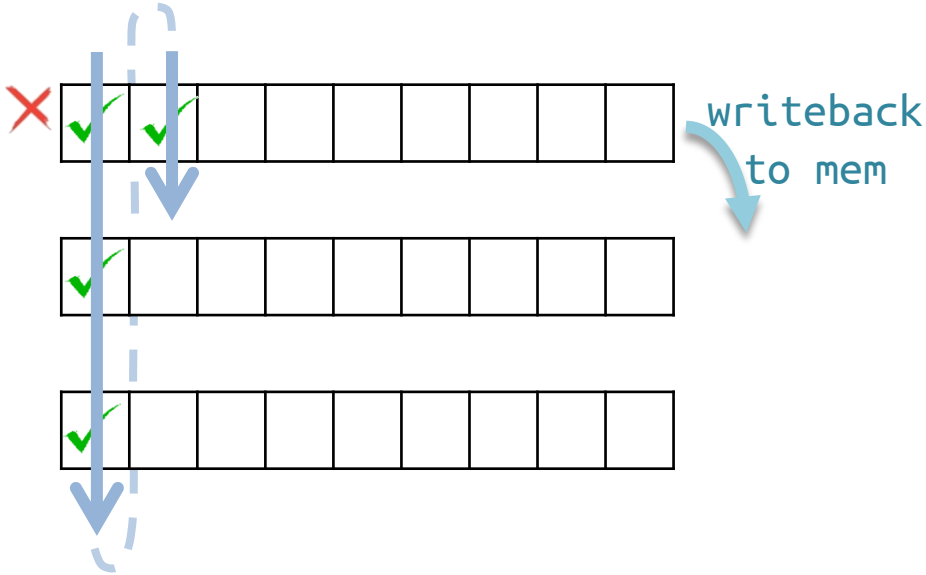


When you traverse in column-major order, you are traversing also the cache in the same way.

It means that you write only a single position in an entire line, which however is flagged “dirty”.

When you pass to the following line, the previous “dirty” one is written back to memory, and so on.

Strided access



Then, when you are back again on the first line and you continue the col-major on the next column, all the cache lines will be again re-written back into memory, and so on.

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

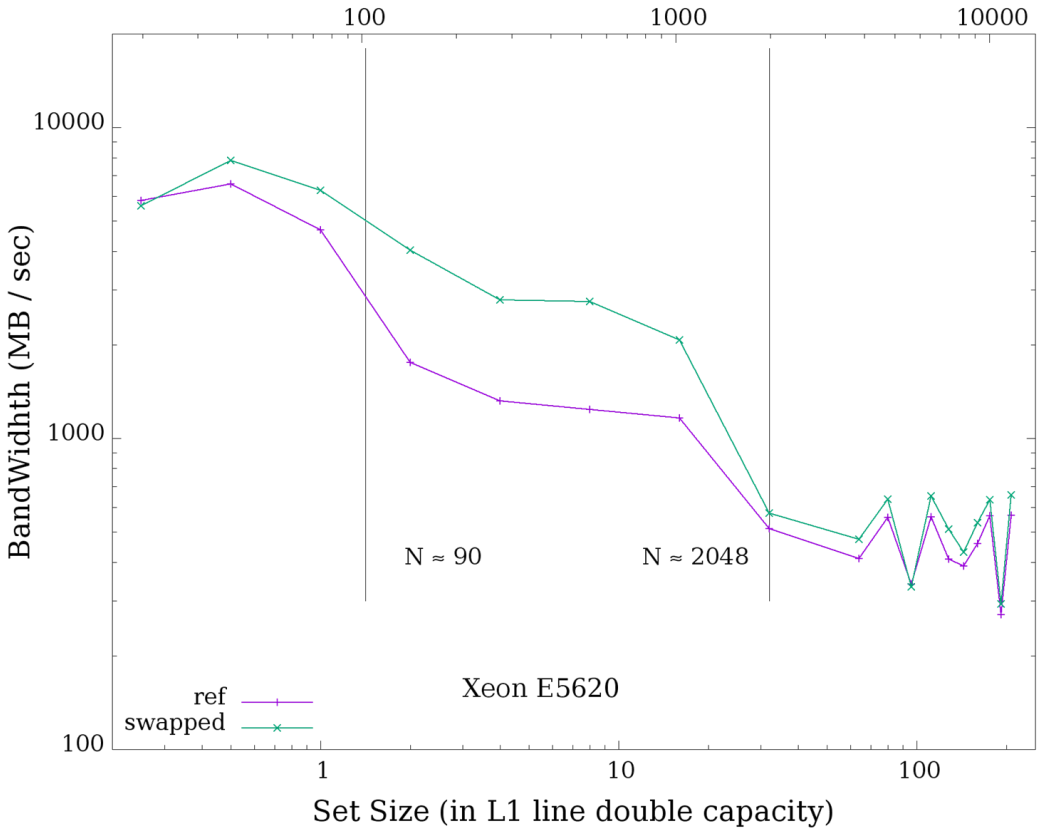
Strided access

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:

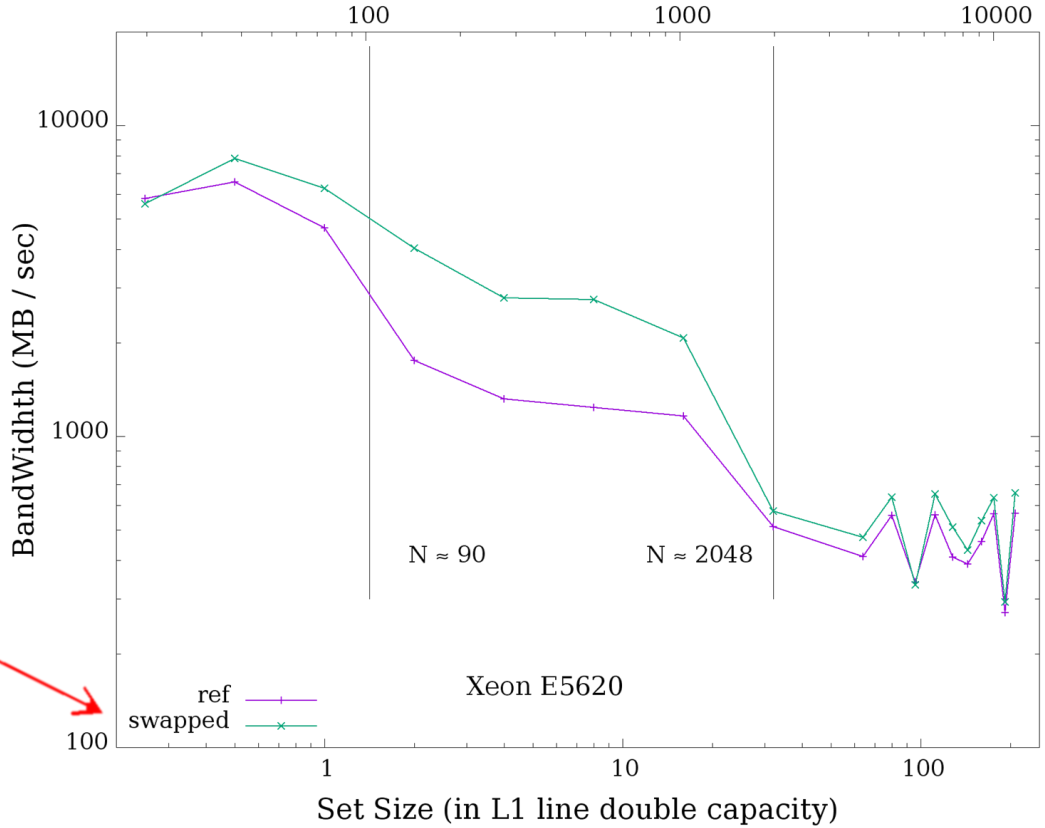
1. $2 \times N^2 < C$
   both matrices can fit in the cache.

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.



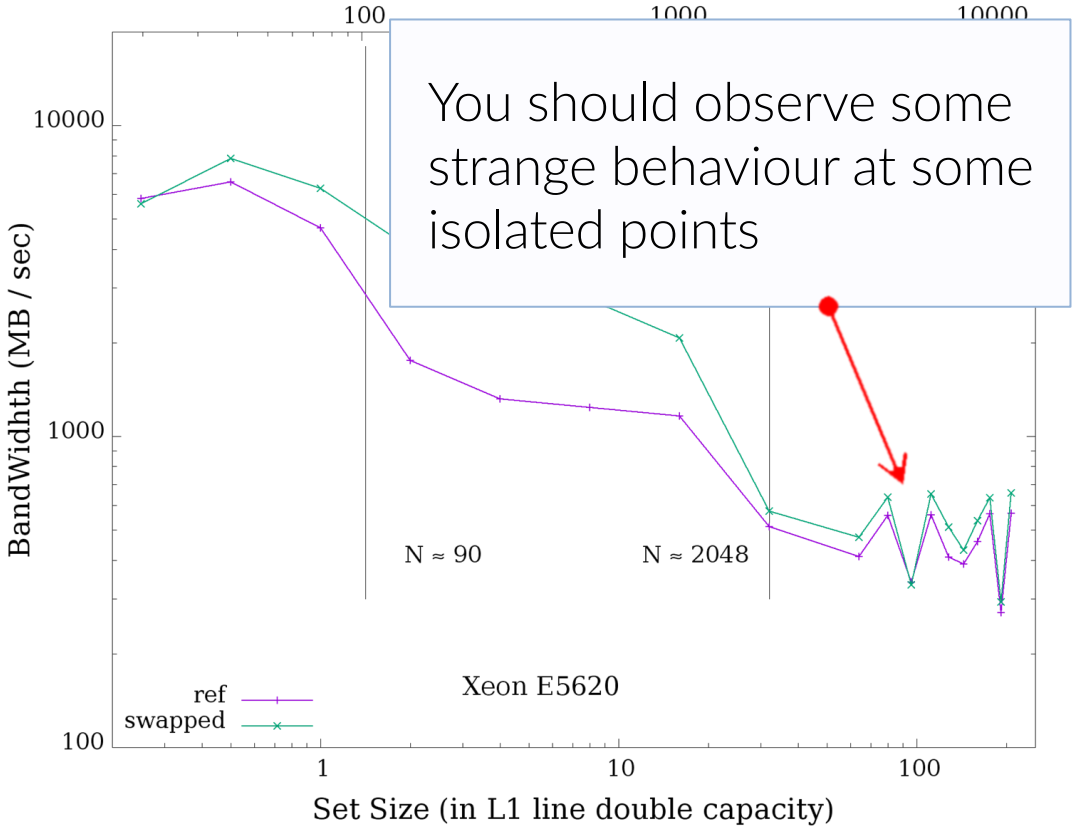
Cache-associativity conflicts

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)

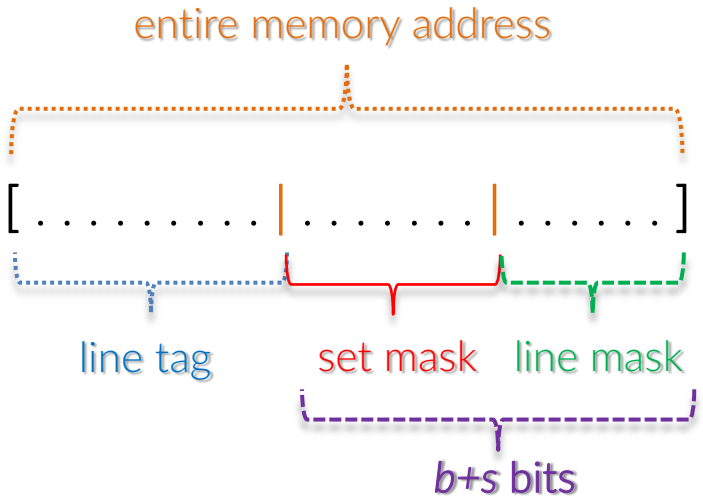
2. $N \times L_C < C$
   strided write is alleviated by fraction of column fitting in the cache

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.



Cache-associativity conflicts

Cache associativity conflicts ("cache resonance")



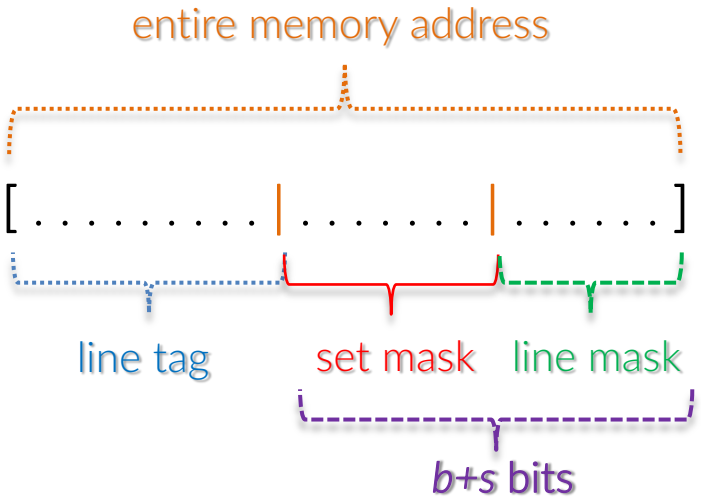
You know that your cache of size C bytes is w-way associative: it means that your cache is made up by lines of size L bytes, grouped in w-sized sets.

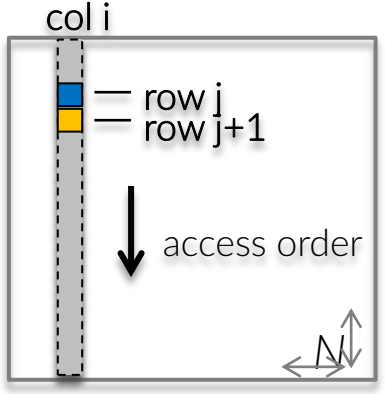
Tipical figures nowadays are: L = 64B (=2^b bytes), w = 4-8, C = 32KB (i.e. there are $C/(L \times w)$ = 64-128 (= 2^s sets).

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).

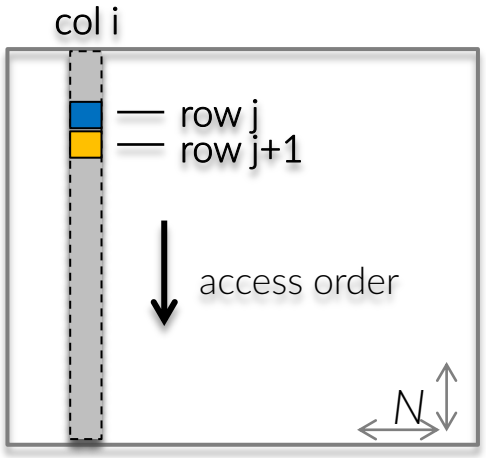
If N is a power of 2, $N = 2^n$; if $W = 2^d$ bytes.

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

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")



Matrix ^{T}

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

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

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

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

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

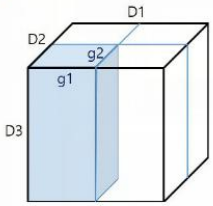
then if

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

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

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 very good return-on-investment)

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

```
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];
}
```

Hot & cold fields

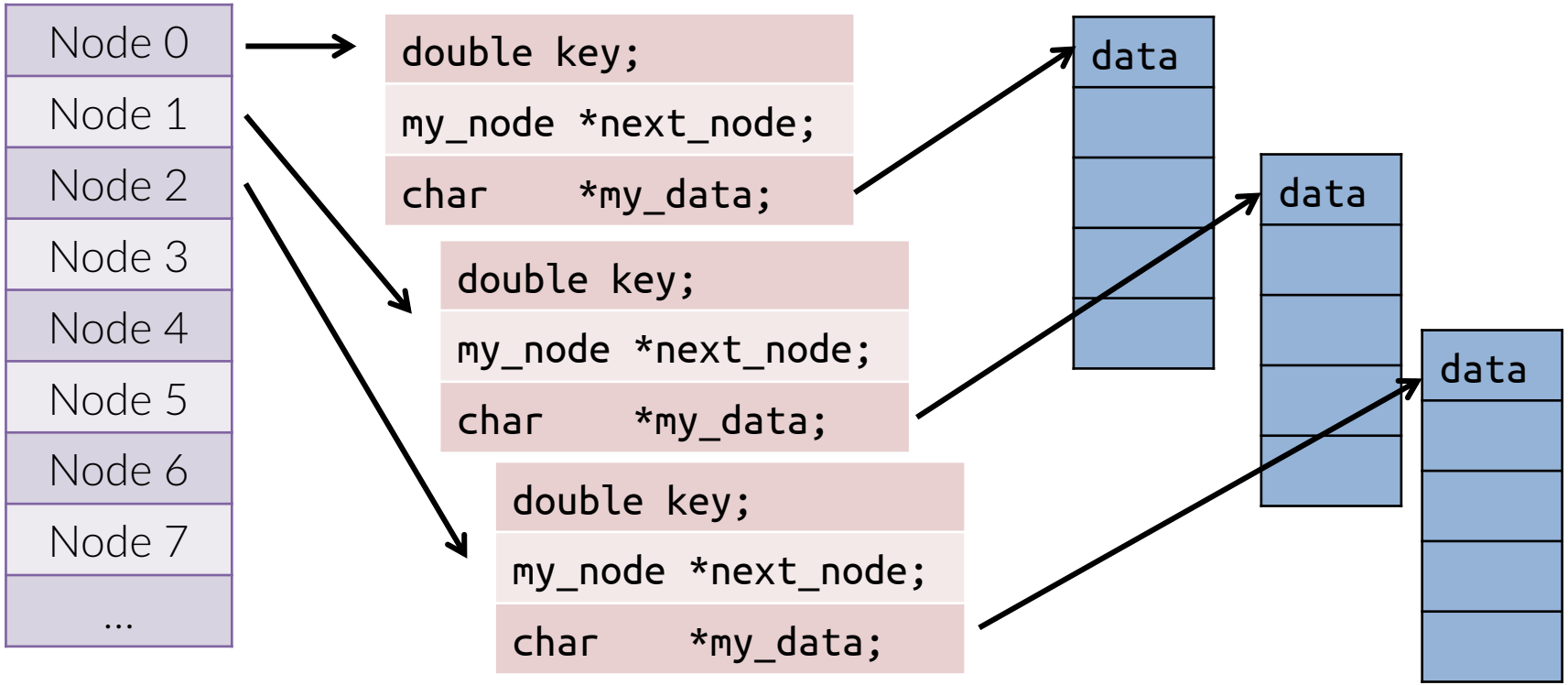
Split fields so that to keep consecutive the fields that are used sequentially

```
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

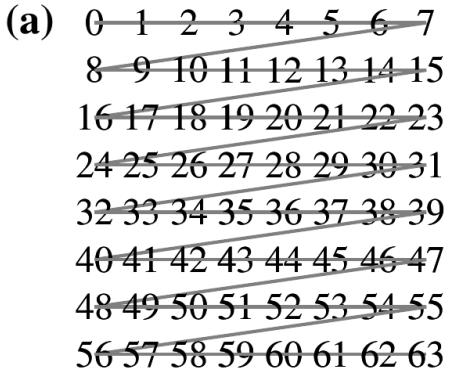
What is the “minimum” distance distortion ?

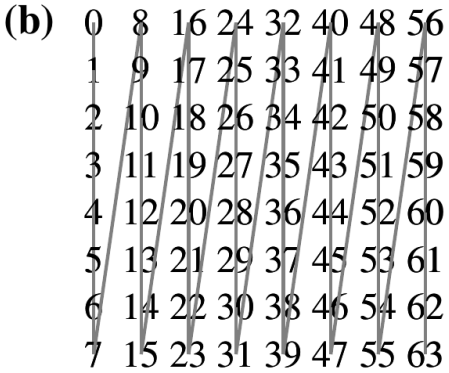
In 1-D the answer is trivial.

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

Organizing data to enhance locality

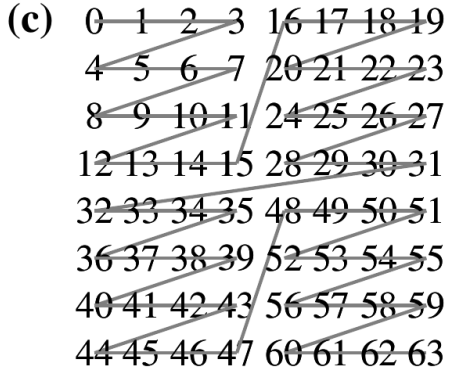
Scanline order *row-major*

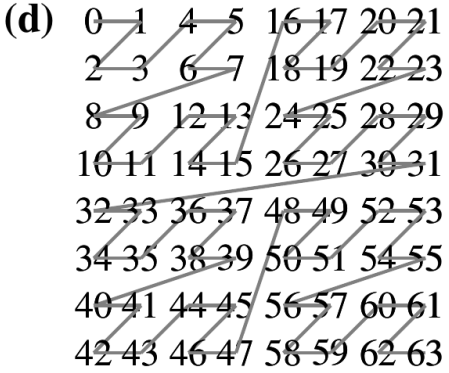




Scanline order *col-major*

Block order + scan sub-order





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

The z-Order

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

`0, 1, 2, 3, 4, 5, …, n`*th*

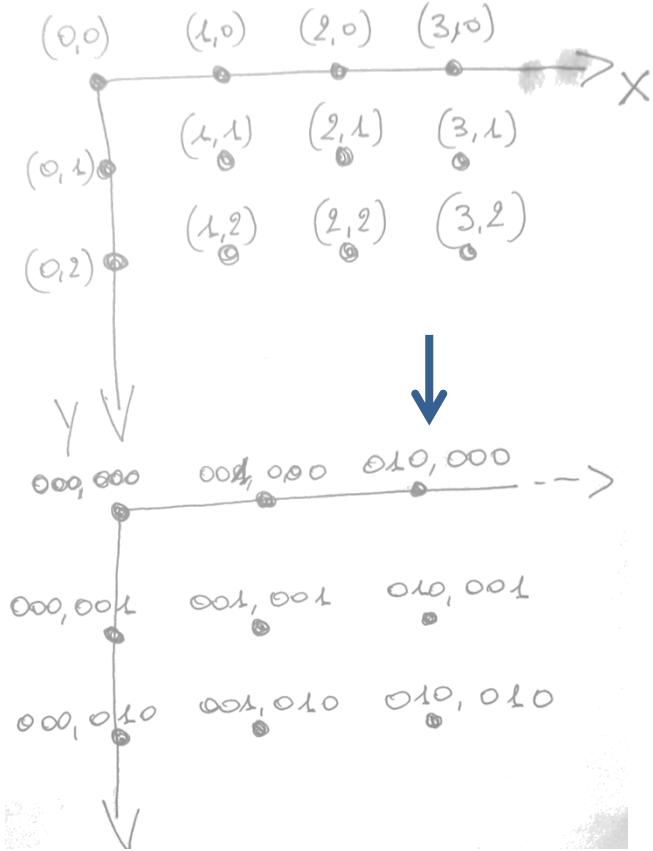
on some spatially-distributed data with integer coordinates.

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:

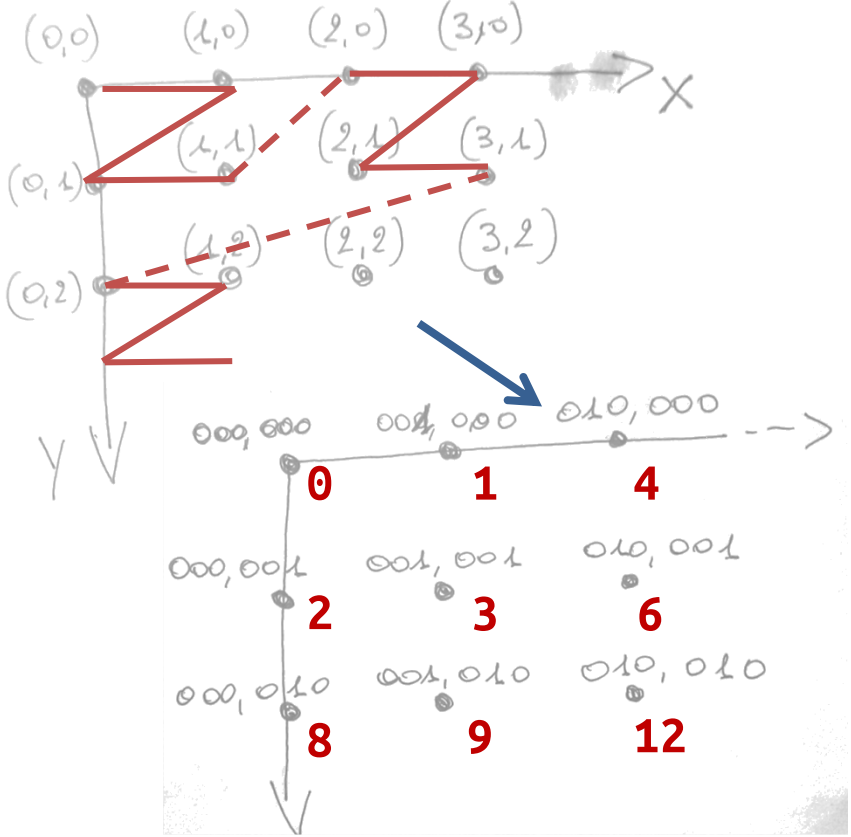
$(x_i \dots x_2 x_1 x_0 , y_i \dots y_2 y_1 y_0)$

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

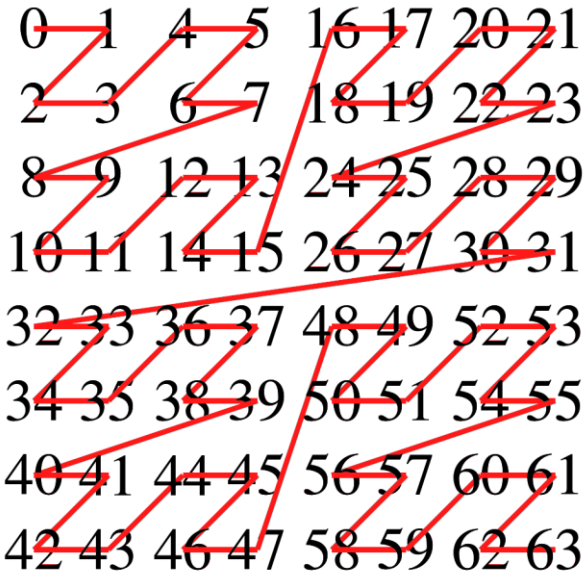
$(x,y) \rightarrow (y_i x_i \dots y_2 x_2 y_1 x_1 y_0 x_0)$

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

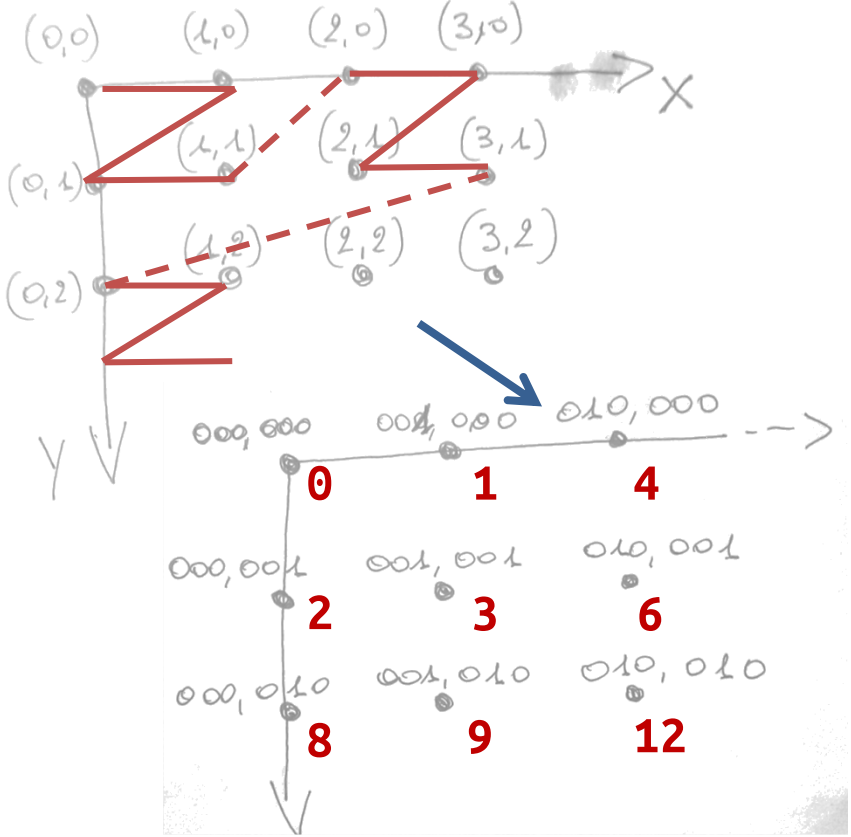
The z-Order



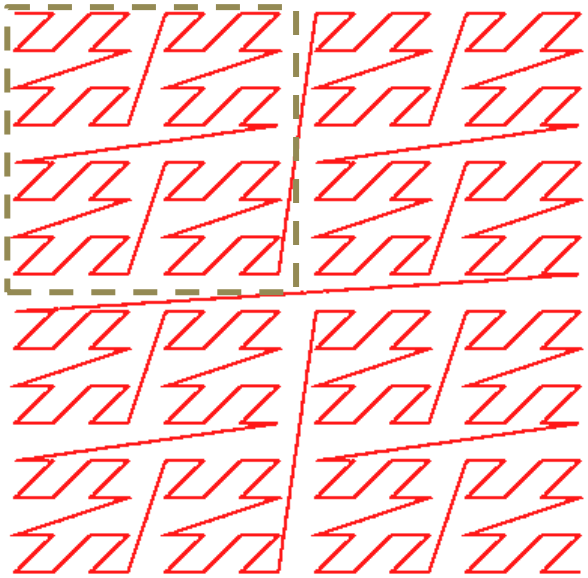
The result of this interleaving is a mapping from the 2-D plane to the 1-D line known as Z-line which is one of the plane-filling curves discovered by Peano



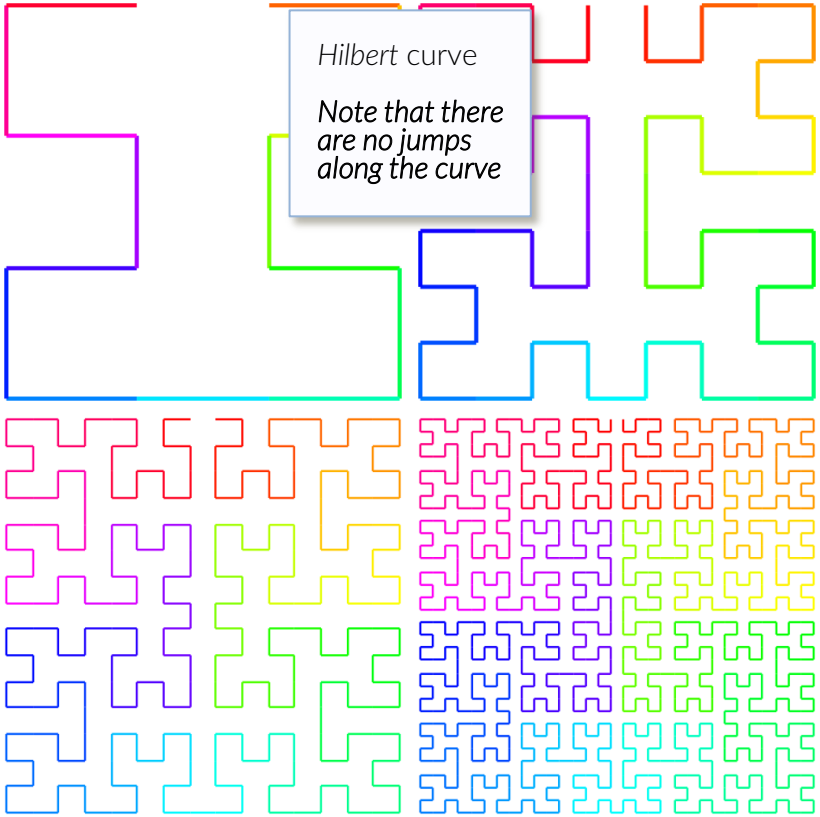
The z-Order

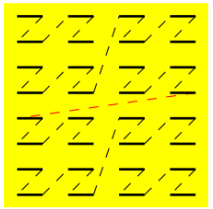


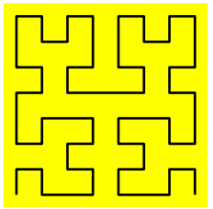
The result of this interleaving is a mapping from the 2-D plane to the 1-D line known as Z-line which is one of the **plane-filling curves** discovered by Peano

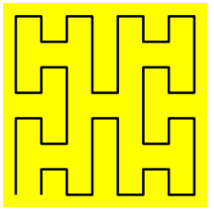


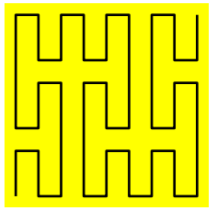
The space-filling curves

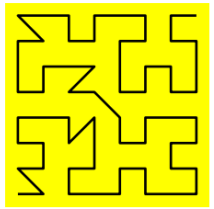


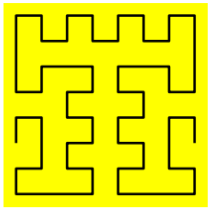

Z-order

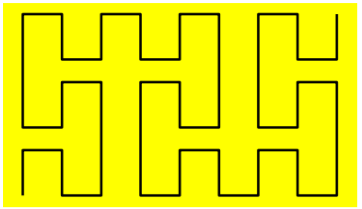

Hilbert curve


H-order


Peano's curve

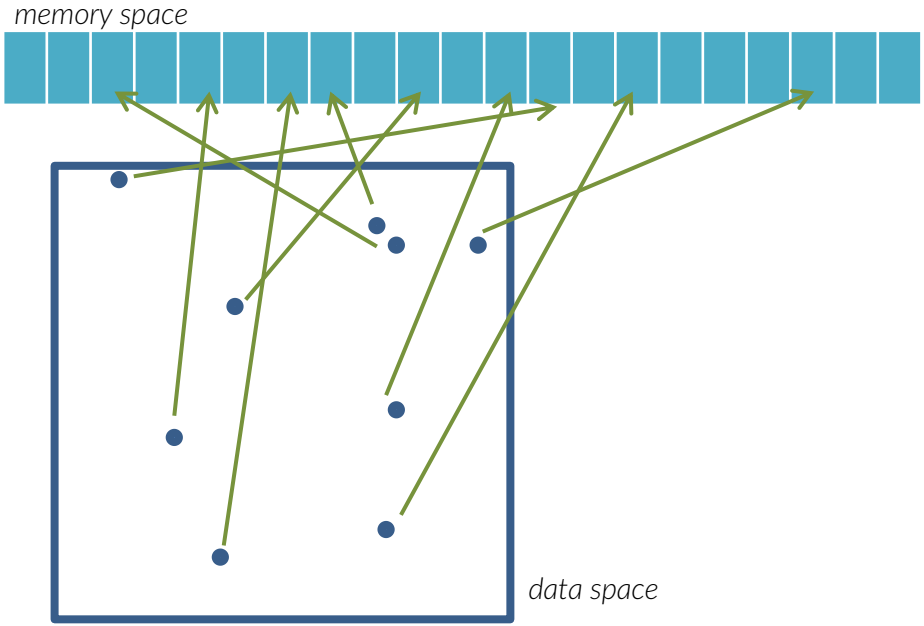

AR^2W^2-curve


$\beta\Omega$-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

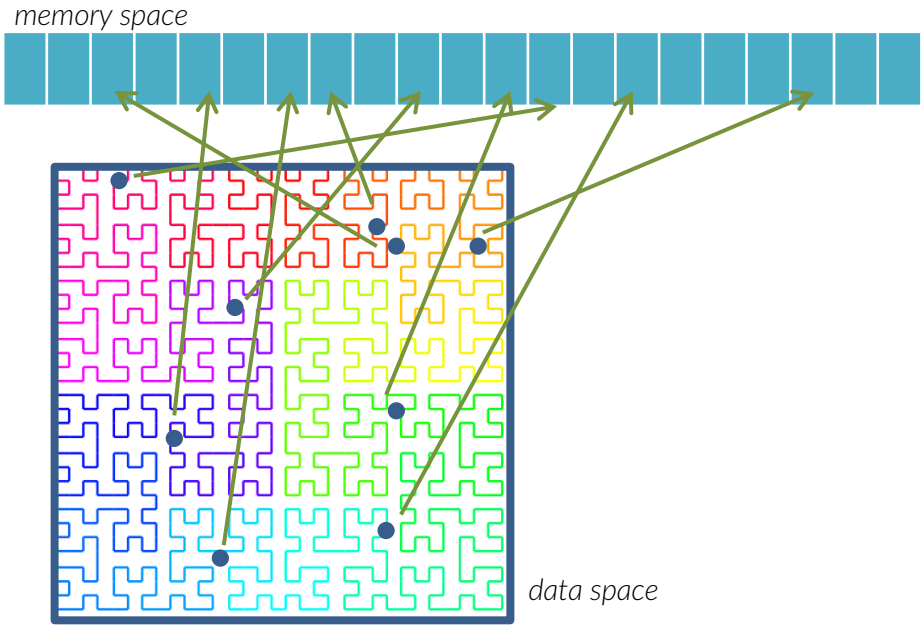
The space-filling curves



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.

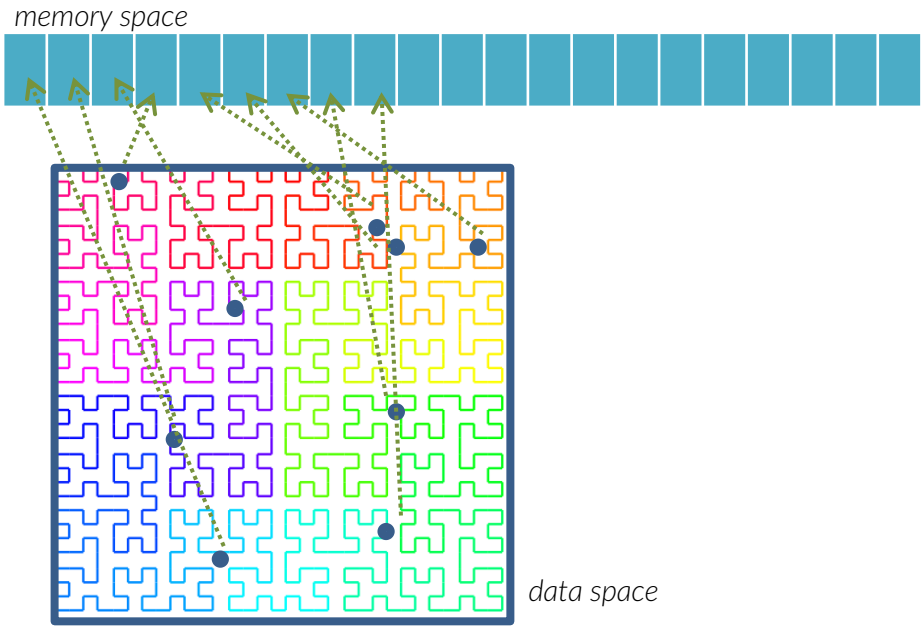
To re-order the data in memory so that to preserve their locality, a space-filling curve can be used.

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

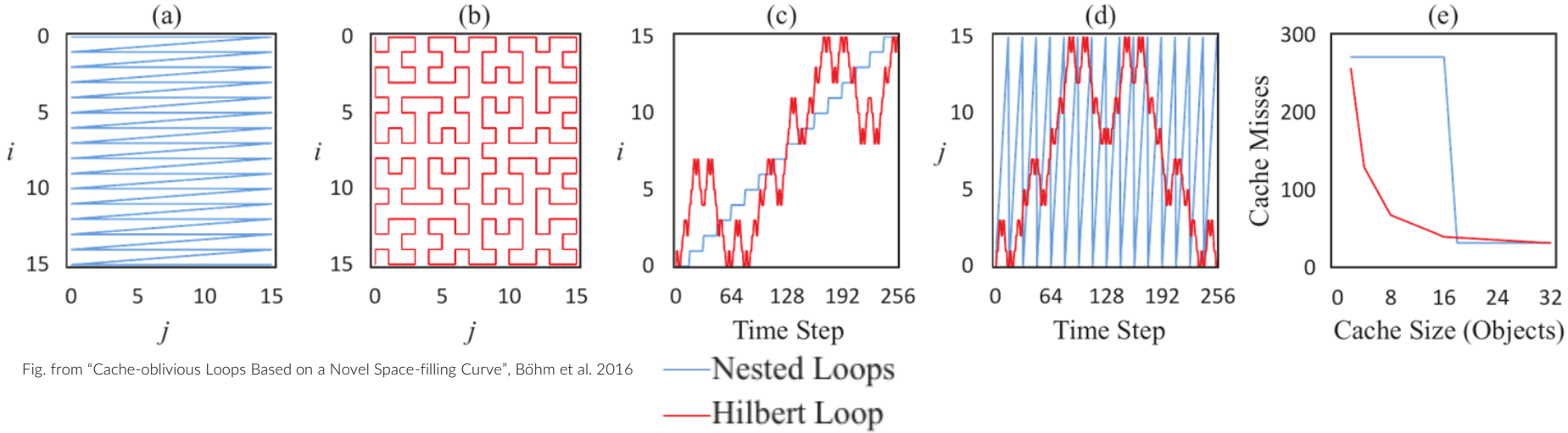


Fig. from "Cache-oblivious Loops Based on a Novel Space-filling Curve", Böhm et al. 2016

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.

Cache recap in two slides

3C for the foes

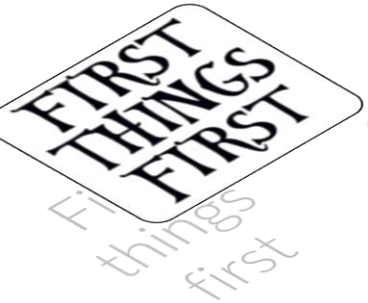
- **Compulsory misses**
  Unavoidable misses when data are read for the first time
- **Capacity misses**
  - Not enough space to hold all data
  - Too much data accessed in between successive use
- **Conflict misses**
  Cache trashing due to data mapping to same cache lines

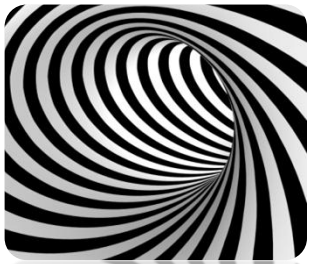
Cache recap in two slides

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

Outline





Loops

Pipelines

Branches

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)
   optimization potential limited
2. $O(N^2)$ / $O(N^2)$
   some more opportunities for opt.
3. $O(N^3)$ / $O(N^2)$
   significant optimization potential

Cache 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

[check the room for loops fusion]

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]
  }
```

In the version on the right, B is recalled from memory only once.

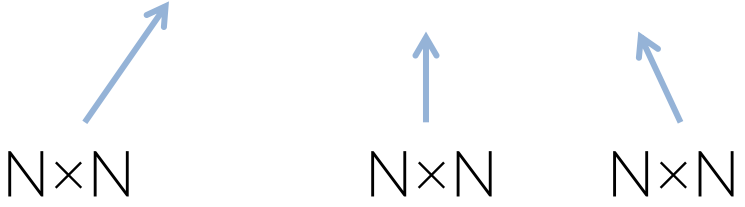
Cache access in loops: $O(N^2)/O(N^2)$

Example

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

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

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



→ $3 \times N^2$

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

→ $2 \times N^2 + N$

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

Note: unrolling and register spill

Using a too large m in the previous example while the target CPU does not have enough registers to keep all the needed operands results in a "code bloating".
In this case, the CPU has to spill registers' content to cache and viceversa, slowing down the computation.

→ learn to inspect the compiler's *log*

A too much involute and obscure loop body may hamper the compiler to effectively perform *unroll & jam* optimizations targeted to the CPU it runs on.

→ hand code effort to clarify the code
→ hints / directives to the compiler
(directives are generally not portable across different compilers)

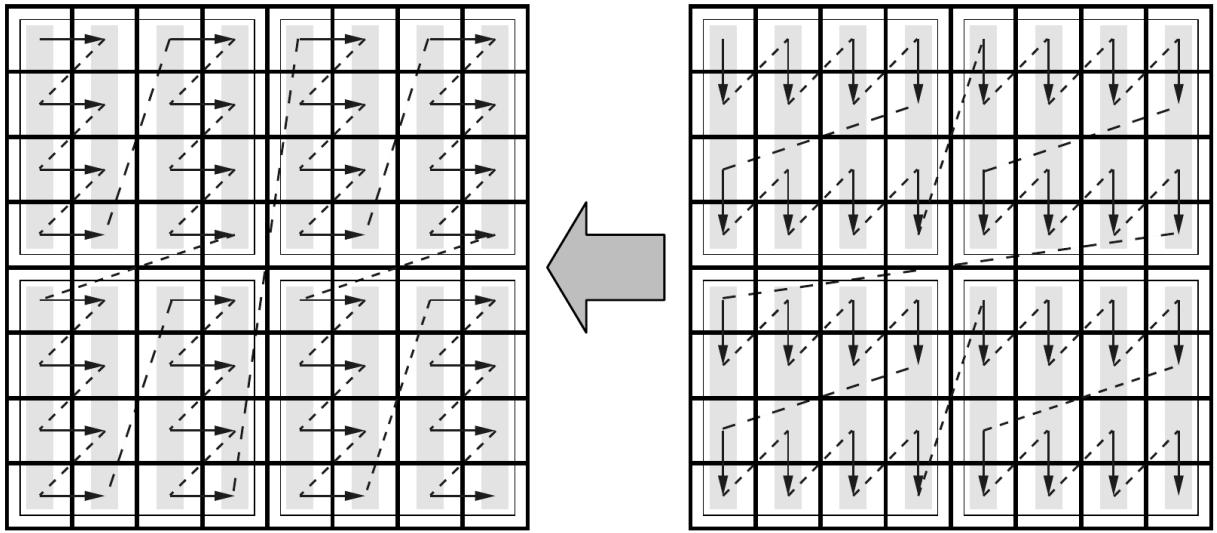
Cache access in loops: $O(N^2)/O(N^2)$

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

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

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

Cache 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

Fig. taken from: introuction to HPC for scientists and engineers

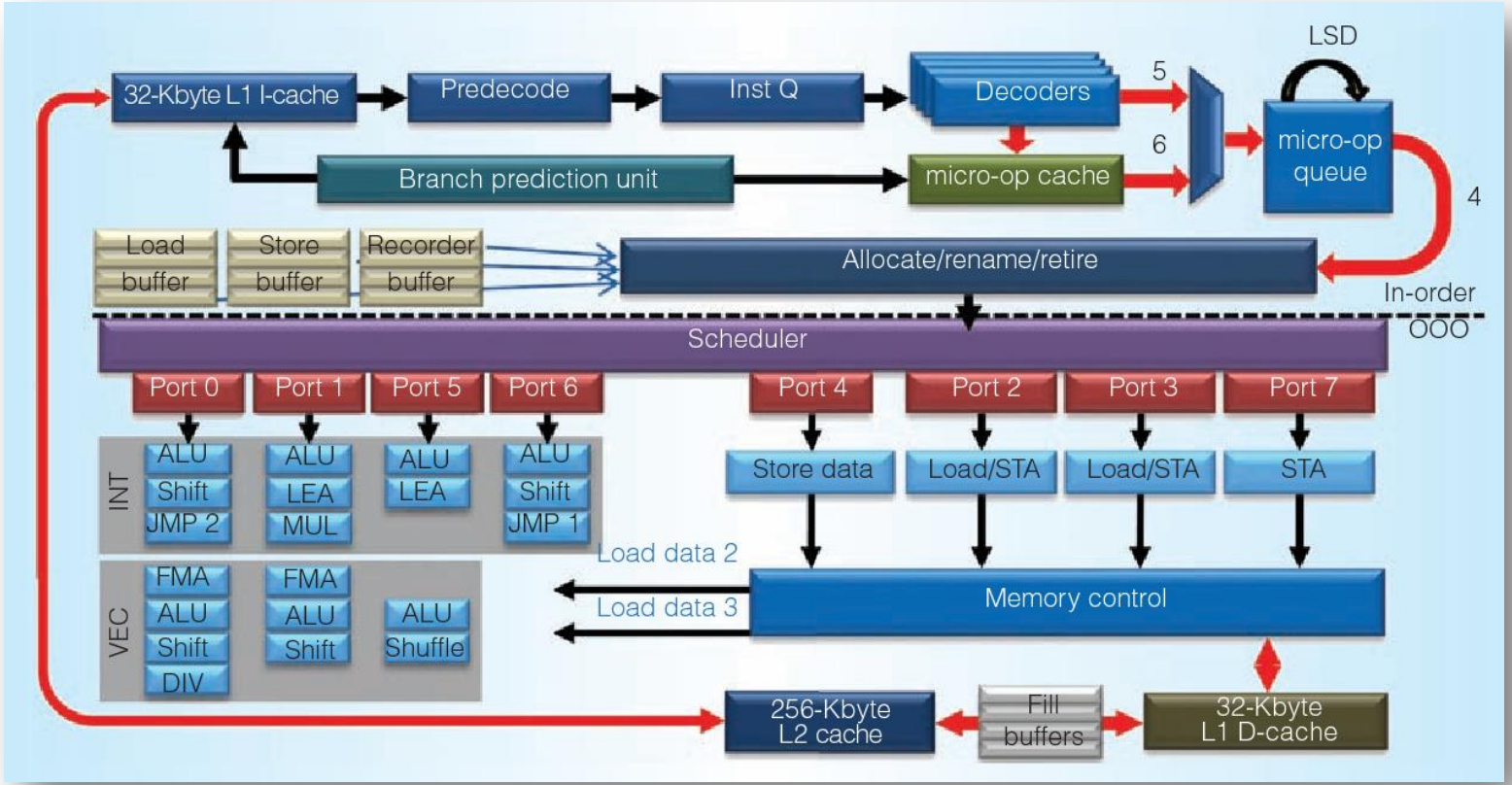
Cache access in loops: $O(N^3)/O(N^2)$

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`).

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

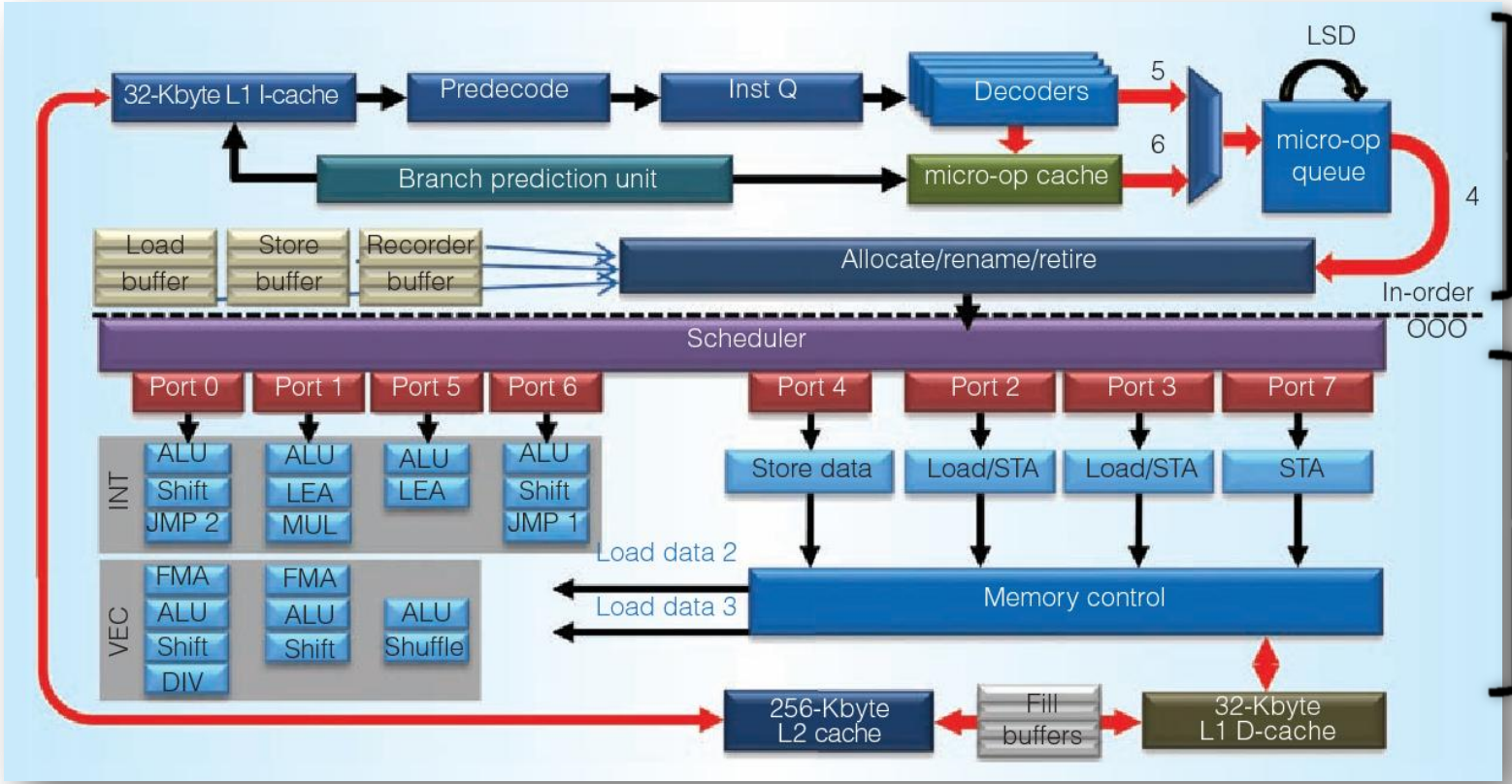
→ It is a brilliant idea to link those library instead of developing your own algorithm, unless some very special needs must be met.

Mid 90s: superscalar and out-of-order CPUs



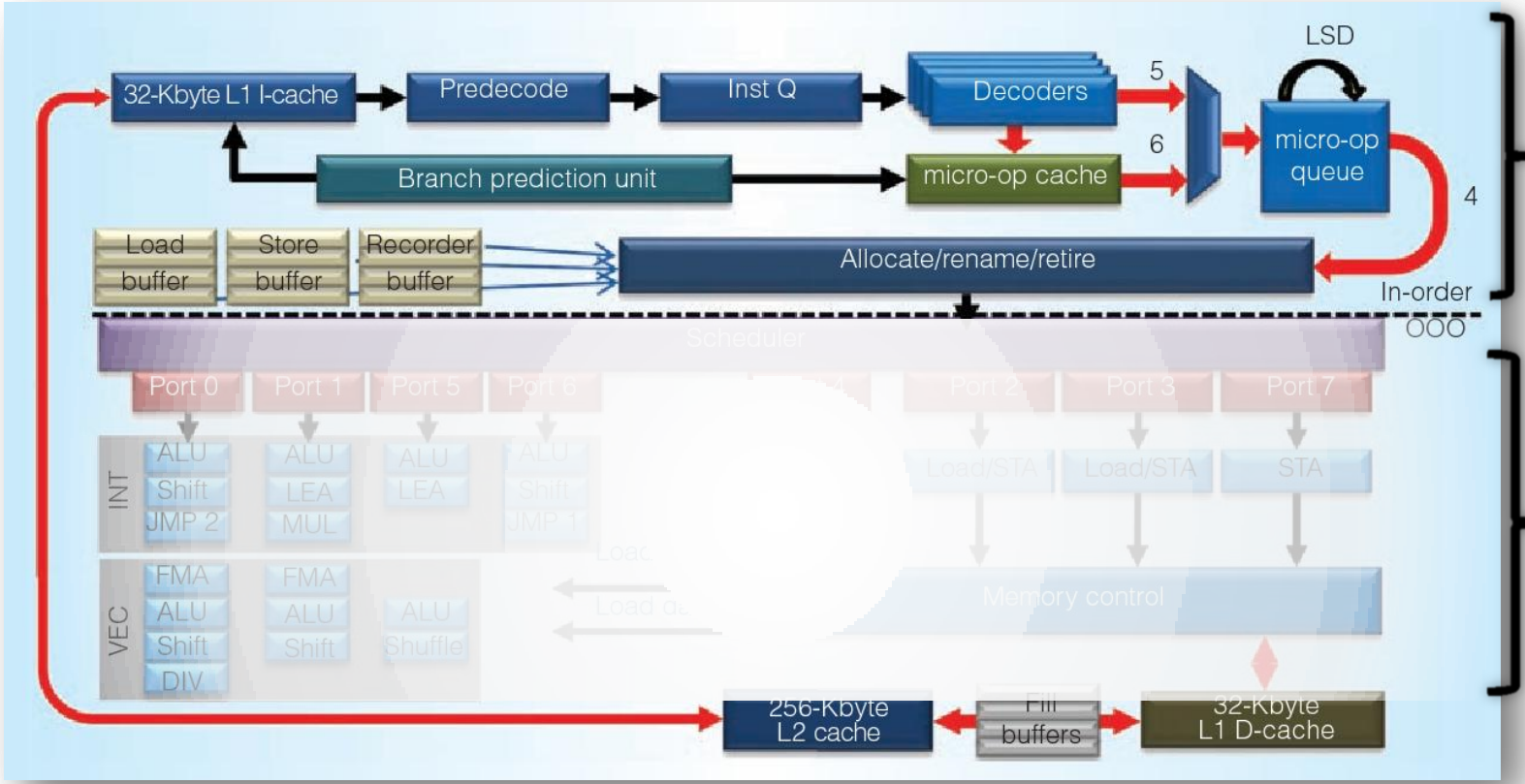
6th generation SkyLake micro-arch.

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



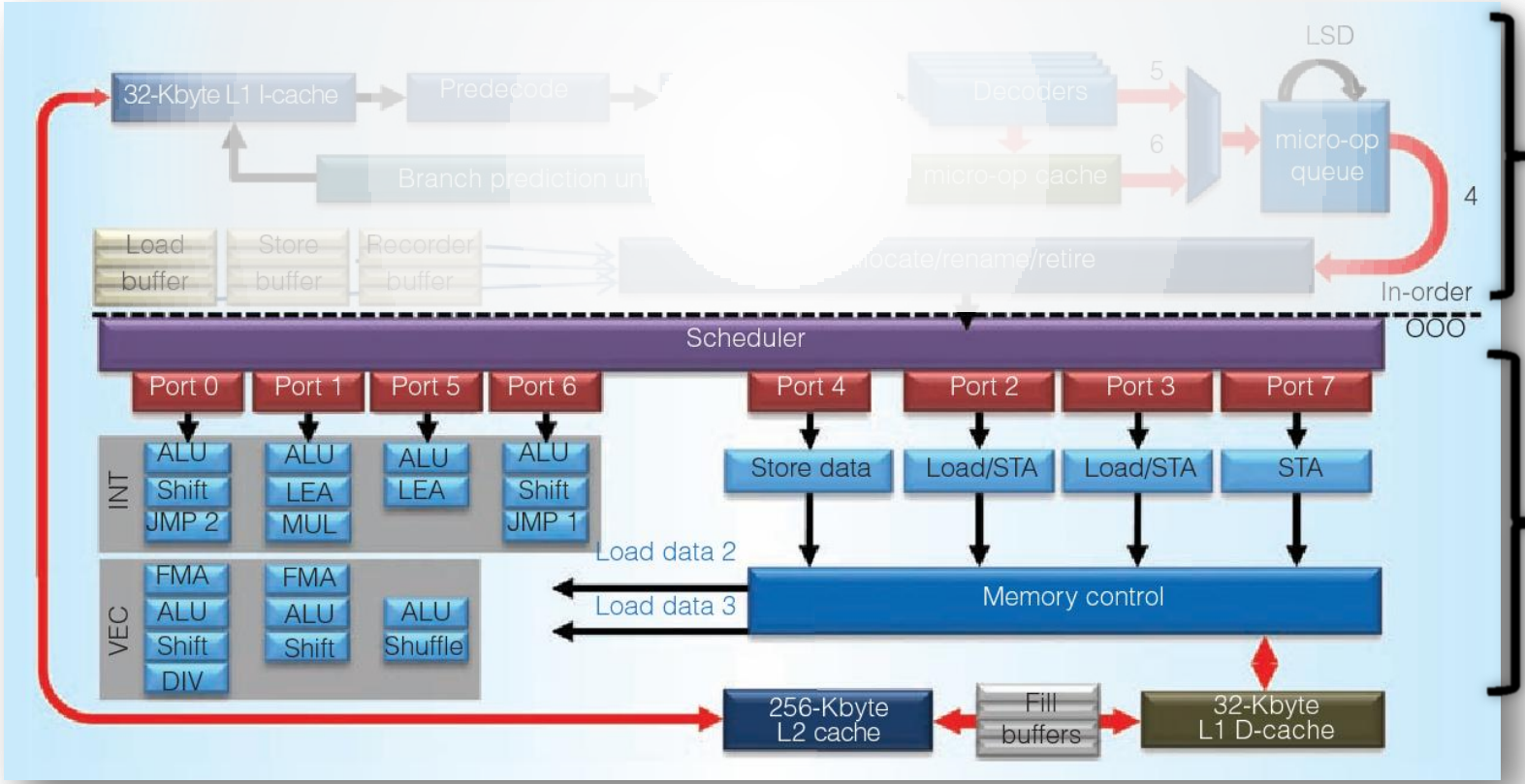
6th generation SkyLake micro-arch.

The *Front-End* basically fetches instructions and the data they operate on from instruction and data caches, decodes instructions, predicts branches and dispatches the instructions to different ports



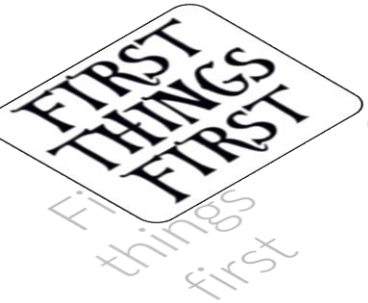
6th generation SkyLake micro-arch.

The *Back-End* is responsible for the actual instructions execution and for the back-writing of results in memory locations. It is responsible also for orchestrating out-of-order ops execution depending on their instructions/data dependencies.



6th generation SkyLake micro-arch.

Outline



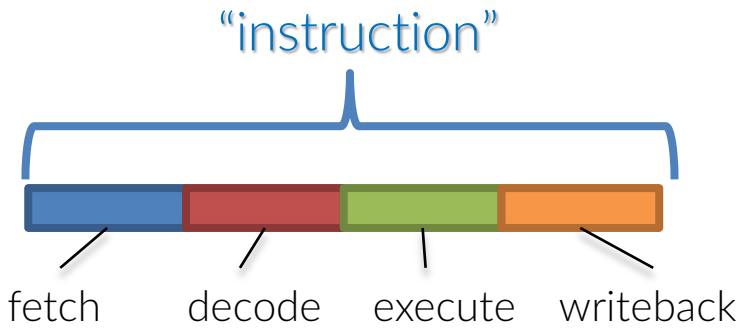
Pipelines

Branches

Pipelines

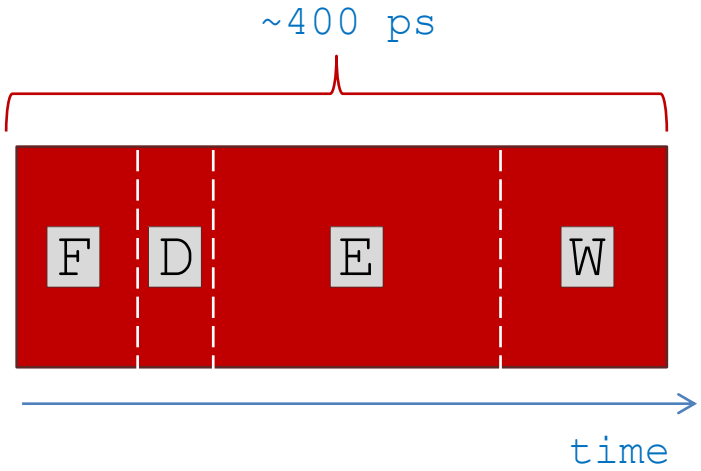
It would be obvious to think that an "instruction" is a kind of *atomic* operation that the CPU perform as a whole. Indeed that was true until the mid of the 80s.

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



1. *Fetching*
   it must be recalled from memory/Icache
2. *Deconding:*
   it must be "understood and interpreted"
3. *Execution*
4. *Writeback :*
   the result must be accounted in memory ()

Pipelines

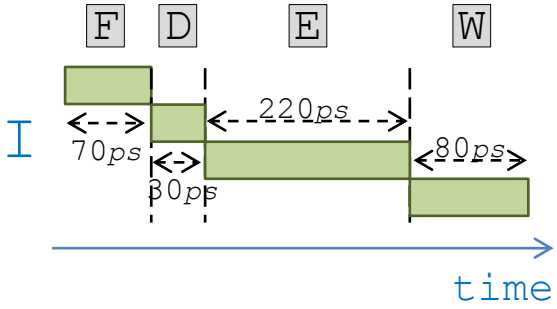


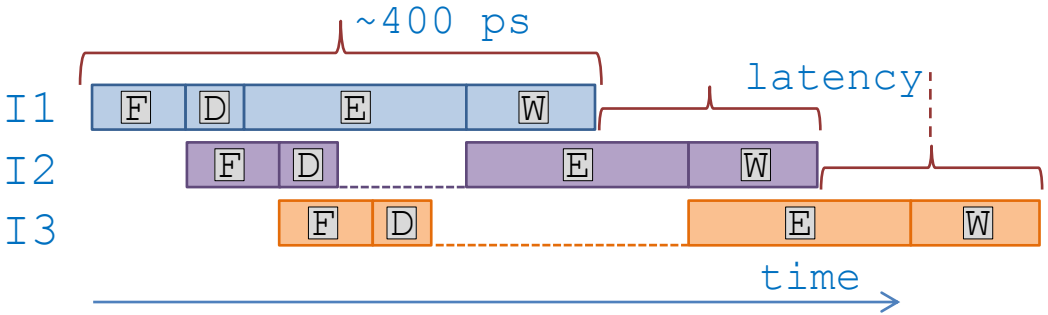
If all the four stages take ~400ps, we then would obtain a **throughput** of 2.5GIPS (giga-instructions per second).
400ps is also the total time required to get a result from an instruction, and so it is the **latency** of the instruction.

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

Pipelines

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



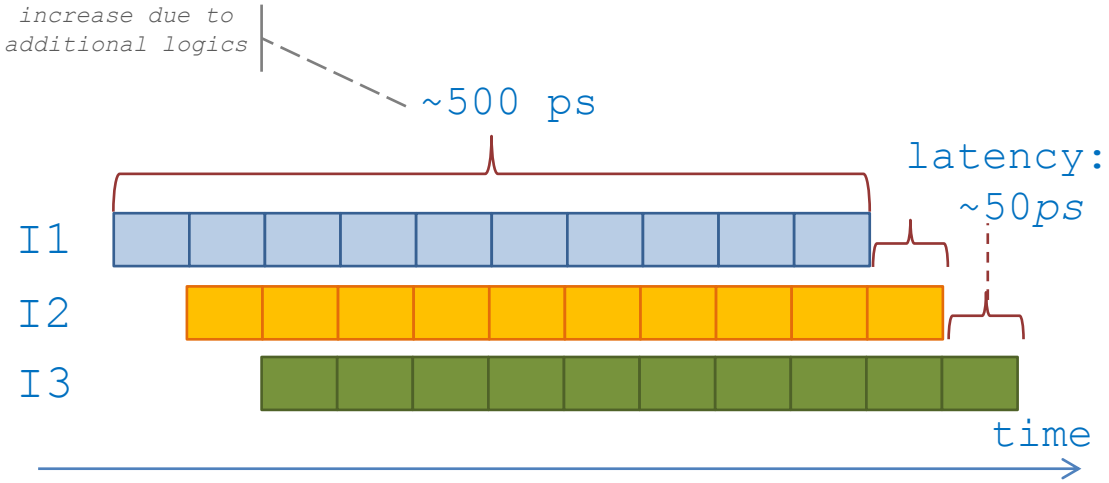


If many independent logical units exist to perform each step, they could operate subsequently on **different instructions**:

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

Pipelines

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



Now the throughput of our system has increased to 1 instruction retired every 50*ps*, i.e. 20GIPS

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

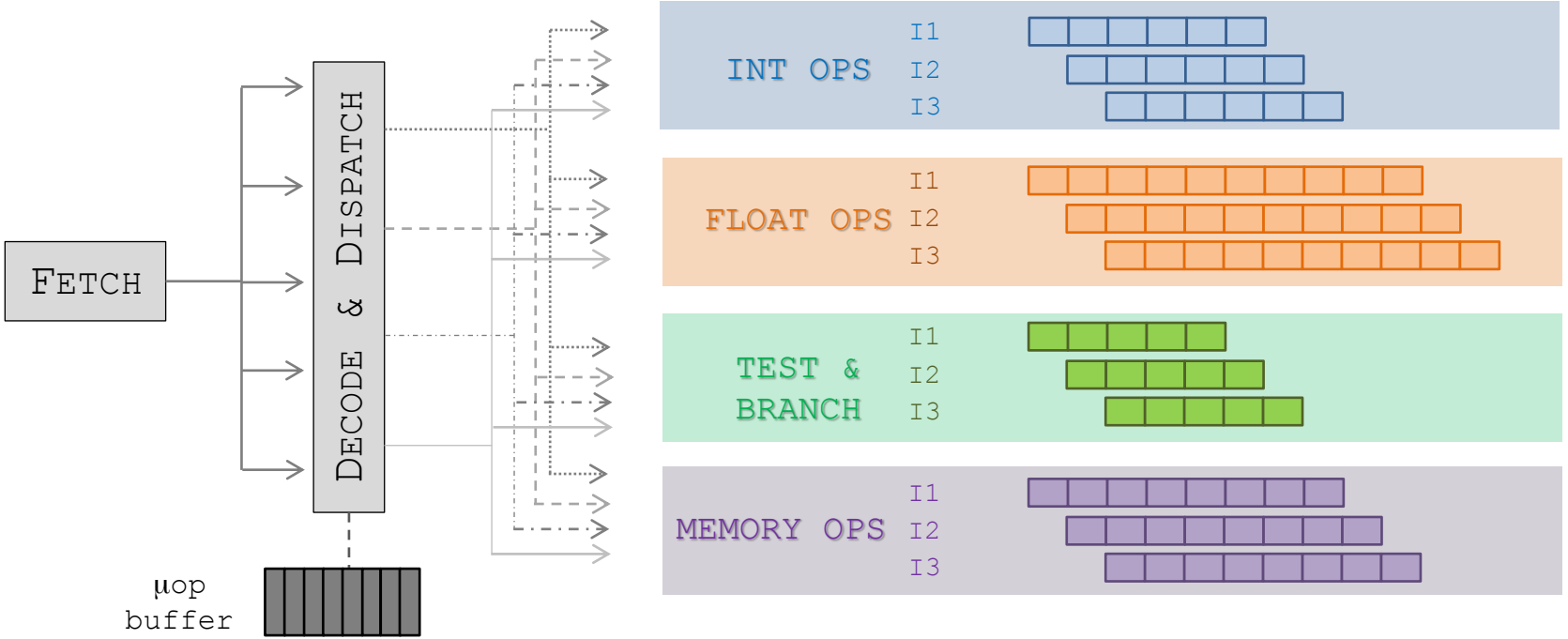
Pipelines are about throughput

Pipelining is then about increasing the throughput of a system, and as we have seen it could be really effective.

However, there is more that can be done working on the **execution stage** that, of course, encompasses a large number of different ***functional units***, performing a **different and independent tasks**.

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

Multiple pipelines



Pipelines latency

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

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

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

Pipelines hazards

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

There are two forms of dependency:

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

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

Pipelines hazards

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

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

The cost of branch mis-prediction

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

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

Example

Let's say we have 140 instructions on the 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%

Pipelines hazards

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

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

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

And the reason why branching is a performance-killer, too.

Loops optimization through pipeline optimization

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

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

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

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

Loops optimization through pipeline optimization

Make things easier for the compiler and the CPU

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

Unrolling make it easier for the CPU to saturate the pipelines.

v. A

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

Separate load and multiply from addition

v. B

Loops optimization through pipeline optimization

Make things easier for the compiler and the CPU

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

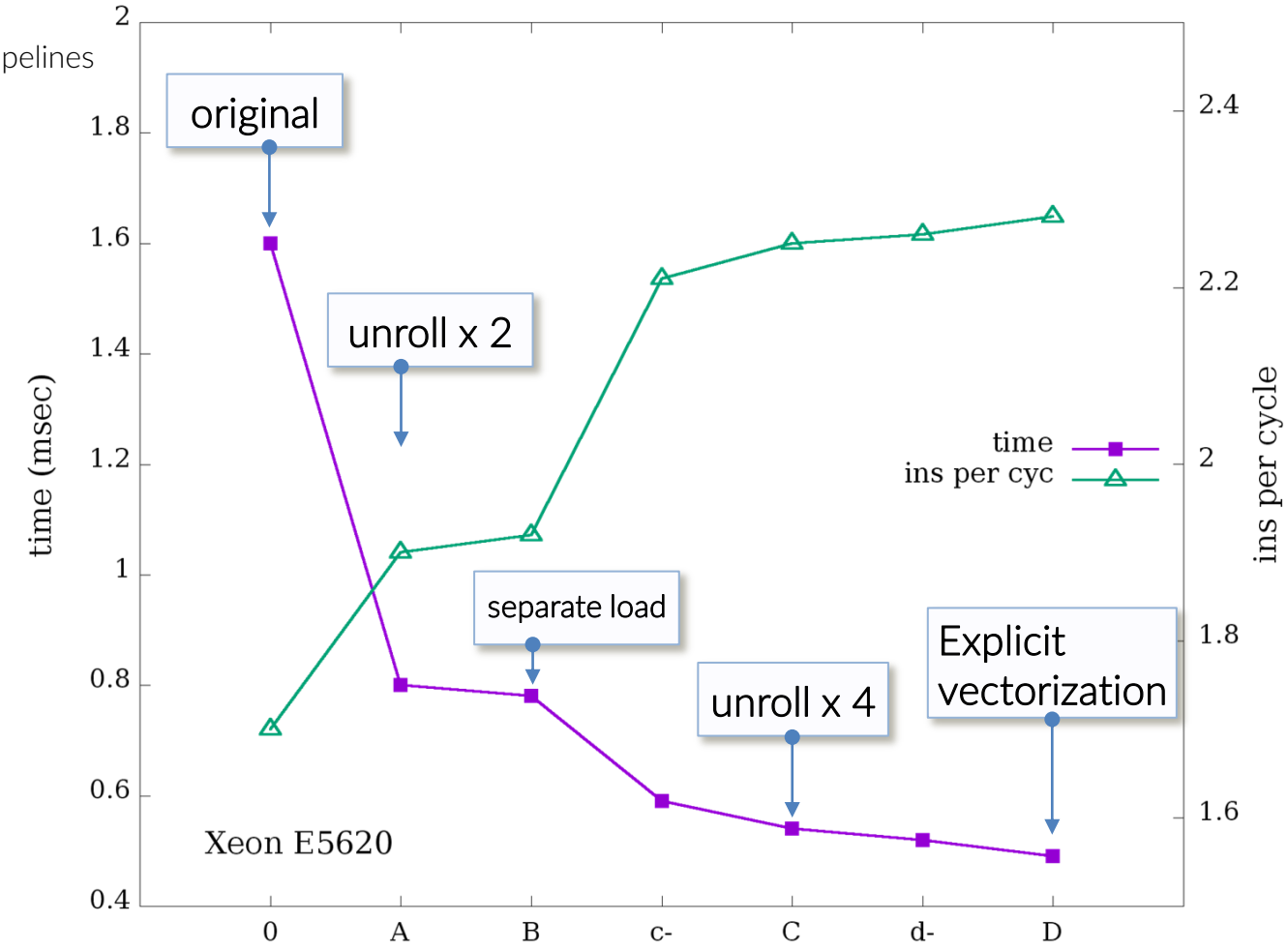
More unrolling may work even better

v. C

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

Explicit vectorization

v. D


original
unroll x 2
separate load
unroll x 4
Explicit vectorization
time
ins per cyc
Xeon E5620
time (msec)
ins per cycle
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
2.4
2.2
2
1.8
1.6
0
A
B
c-
C
d-
D

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

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

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

Hands-on

v0

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

Comparison btw 2 compilers

v0

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

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

        vmovsd  -32(%rbp), %xmm1

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

        vmovsd  %xmm0, -32(%rbp)


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

PGI 2018.10

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

Gcc 8.2.0

Comparison btw 2 compilers

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

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

        vmovsd  -32(%rbp), %xmm1

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

        vmovsd  %xmm0, -32(%rbp)


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

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

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

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

Note: no optimization has been used in both cases

Comments on the previous slides

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

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

Both compiler could be induced to generate different and more efficient code playing with their options (the default behaviour can differ among different compilers).
The result is more or less equivalent from the point of view of the run-time due to the fact that the code generated by `gcc` is larger (more instructions) but it is dispatched to more pipelines (larger IPC).

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

Turning on the optimization

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

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

-O0

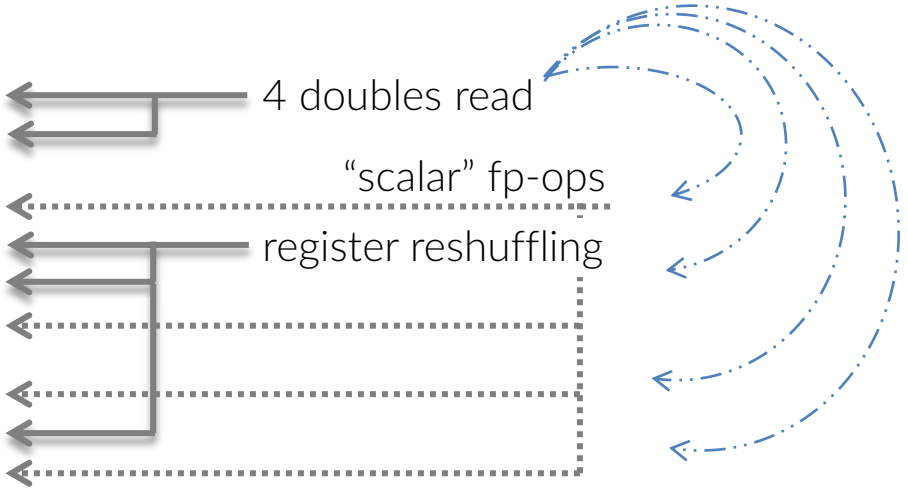
```
.L8:
# v0.c:185:     sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax,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

-O3 -march=native

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

Turning on the optimization

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



v0

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

gcc 8.2 on SkyLake
-O3 -march=native

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

Comments on the previous slides

Looking at the assembler generated by the compiler may be useful to understand where to direct optimization efforts.

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

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

(*)you should remember what alignment is from the lecture about memory allocation

V* profile – 00

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

v0

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

v1

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

v3

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

v6

V* profile – O3

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

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

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

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

V* profile – O3 + vectorisation

There's a lot of improvement in such a simple kernel, but why IPC even decreased?

Hint: the code becomes memory-bound (look at the memory transfer rate).

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

Comparison: v0 – v1

v0

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

v1

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

Comparison:

v0

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

v1

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

Comparison:

v0

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

v1

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

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

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

Comparison: v0 – v1

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

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

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

Refining v1

v1

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

v1b

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

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

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

Refining v1

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

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

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

Refining: v3

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

prefetching

v3

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

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

v3b

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

          a = b;
        }

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

Refining: vectorisation

With the following statement you can define a vector type in `gcc`

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

By creating this union, you can access the single doubles within the vector

This is the actual vector, made up by 4 doubles. You can not access the single elements.

Refining: vectorisation

What is this strange stuff about ?

The loop is re-written here in its simplest form.

Now, however, the variables are vectors.

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

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

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

With the following statements you prepare the ground to use *intrinsics*, both types and functions.

Vector type with 256bits (AVX)

You can initialize it as an array

The FMA in vector AVX flavour

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

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

Intrinsics are an API, exposed by the compiler(*), that provide direct access to vector instructions and types.

Why am I defining those macros ?

(*) they are no compiler-dependent: look at https://software.intel.com/sites/landingpage/IntrinsicsGuide/ for Intel's intrinsics

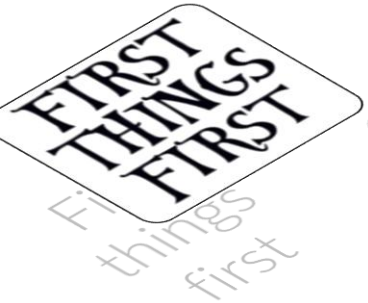
Refining: vectorisation with intrinsics

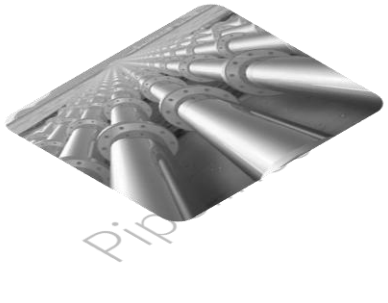
A before, the loop is re-written here in its simplest form.
It also looks very similar to the previous one.
The variables are still vectors.

Note the usage of macros: the code is still valid if you use AVX512 or SSEx

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

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

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

      tstart = CPU_TIME;

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

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

Outline





Branches

Branch prediction

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

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

Branch prediction

ex 1: Taking decisions before and outside the loop

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

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

Branch prediction

If you do not trust your compiler to perform the *loop hoisting* for you,

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

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

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

Branch prediction

However:

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

- "Unrolling" the if-tree outside the for – then having multiple for loops may be highly unpractical if the branches are big piece of code.
  There's no really a Swiss-knife recipe.

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

Branch prediction

ex 2: code restructuring

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];
  }
```

Branch prediction

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

Branch prediction

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++)
  {
     t = (data[ii] - PIVOT -1) >> 31;
     sum += ~t & data[ii];
  }
```

Branch prediction

-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%
```

Branch prediction

-O3

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

sum is 983597794767, elapsed seconds: 0.660148

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

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

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

sum is 983597794767, elapsed seconds: 0.679286

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

-O3
-march=native

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

sum is 983597794767, elapsed seconds: 0.217864

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

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

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

sum is 983597794767, elapsed seconds: 0.224288
```

Branch prediction

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

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

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

compare

if flag     → mov BX in AX
```

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

Branch prediction

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

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

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

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

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

12 instructions to process 4 **`int`**

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

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

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

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

9 instructions to process 8 **`int`**

Branch prediction

We can do slightly better:

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

can be changed to

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

Branch prediction

Let's look to another practical example

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

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

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

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

Branch prediction

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;
   }
```

Branch prediction

```
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);
```

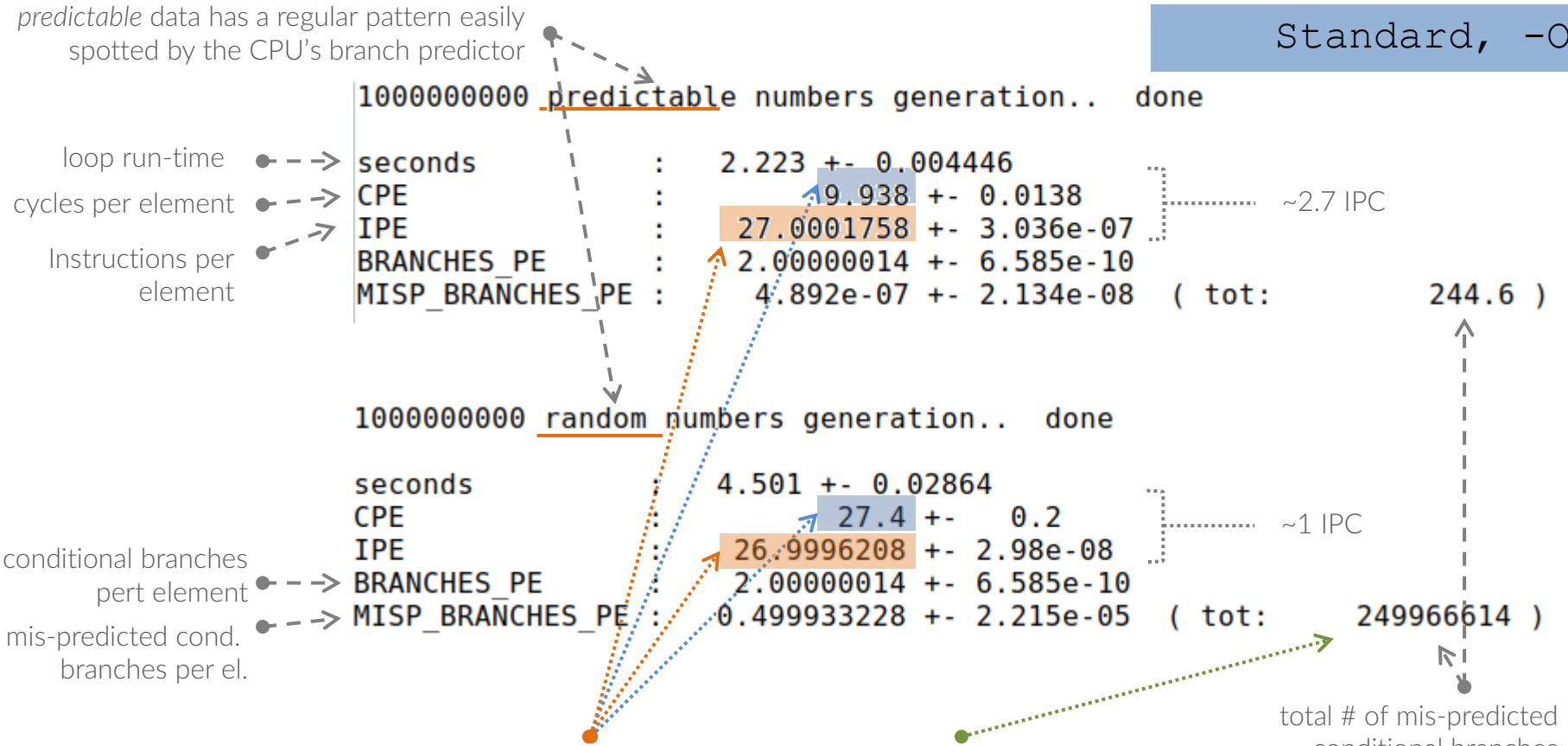
smart2

```
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
1000000000 predictable numbers generation.. done
loop run-time
cycles per element
Instructions per element
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)
~2.7 IPC
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)
~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)

Branch prediction

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

Branch prediction

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

Branch prediction

smart3, -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

Comments on the previous slides

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

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

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

`std` implementation, `-O0`

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

1 `cmp` instr. with a following jump

2 `cmov` instr. can use 2 pipelines

`smart` implementation, `-O0`

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

Comparison with predictable data

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

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

However, at `-O3` its behaviour is the worst one, with CPE larger by ~75% and IPE larger by a factor of ~2. Only with very aggressive optimization the compiler can generate a code comparable with the smart*X* ones

Comparison with non-predictable data

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

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

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

Conditional branches with pgi compiler (v 18.10-0)

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

Comments on the previous slides

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

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

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

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

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

Changing the point of view

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

Just changing point of view sometimes may help:

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

Changing the point of view

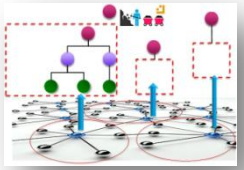
Can easily be re-written with no conditional evaluations at all:

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

       matrix[j][i] = 0.0;

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

A word of caution

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

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



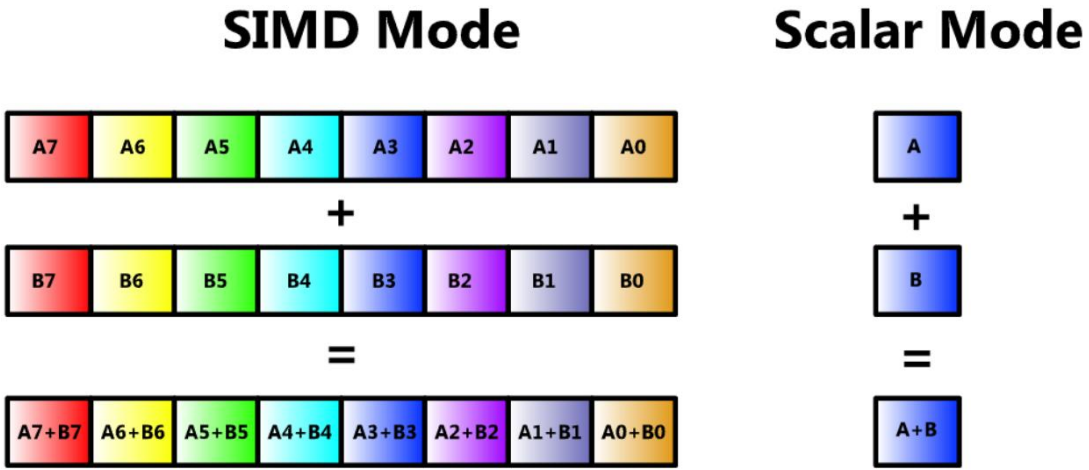
The algorithms that you choose

The data model you design

The overall quality, cleanness and robustness of your code

Late 90s: vector capabilities become mainstream

https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf

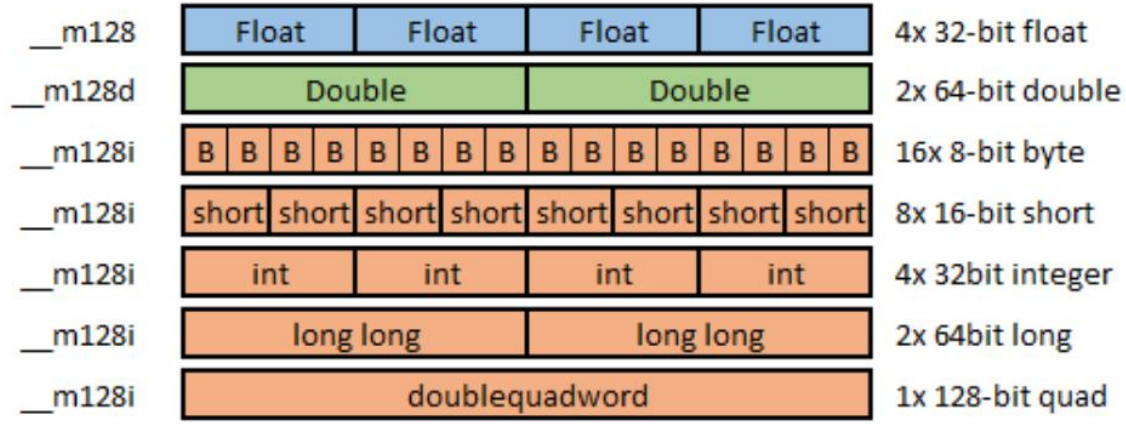


Vector registers are large special registers in the CPU that can be considered subdivided in smaller independent chunks over which the same operation can be performed:

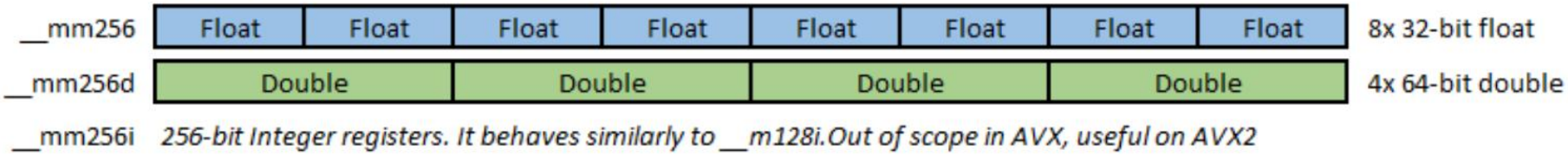
SIMD: Single Instruction
Multiple Data

Vector registers size

SSE Data Types (16 XMM Registers)



AVX Data Types (16 YMM Registers)



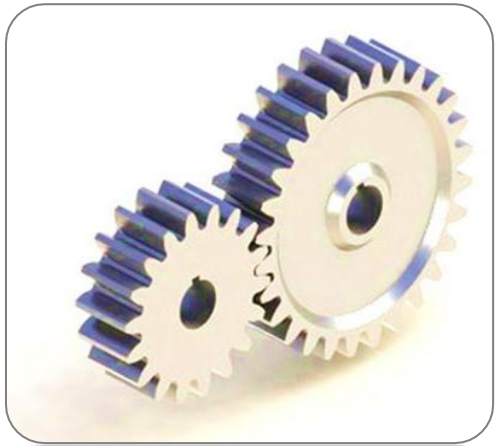
Early 00s: multi-core becomes mainstream

Late 00s : accelerators become mainstream
Early 10s: many-cores + heterogeneous computation become mainstream

That is the focus of the course on OpenMP/MPI

Some more sparse material

Prefetching



Macros as Templates

NUMA locality

At the right moment, at the right place

We know that waiting for data and instructions is a major performance killer.

Modern CPUs have the capability of pre-emptively bring from memory into cache levels data that **will be needed shortly afterwards**.
They can do that following some speculative algorithm based on the current execution flow and assuming spatial locality and temporal locality.

Both *data* and *instructions* can be pre-fetched.

Pre-fetching may be both hardware-based and software-based (tipically the compiler insert pre-fetching instructions at compile-time).

At the right moment, at the right place

From the point of view of the programmer, there are 2 possible ways to deal with prefetching:

EXPLICIT — you explicitly insert a pre-fetching directive.
Very difficult to be achieved effectively: the directive must be inserted timely but not too early (data eviction) or too late (load latency).

INDUCED — you consciously arrange data layout and execution flow so that to make it obvious to the compiler what to prefetch.

Explicit prefetching

This is a standard binary search implementation.

Find the median element

Define the next search

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

   while(low <= high) {
     mid = (low + high) / 2;

     if(data[mid] < Key)
       low = mid + 1;
     else if(data[mid] > Key)
       high = mid-1;
     else
       return mid;
   }
   return -1;
 }
```

Explicit prefetching

We can make it better by simply making sure that the element to be compared for (the `mid`) is in the cache when requested

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

   while(low <= high) {
     mid = (low + high) / 2;

     if(data[mid] < Key)
       low = mid + 1;
     else if(data[mid] > Key)
       high = mid-1;
     else
       return mid;
   }
   return -1;
 }
```

Explicit prefetching

We can make it better by simply making sure that the element to be compared for (the `mid`) is in the cache when requested

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

   while(low <= high) {
     mid = (low + high) / 2;
      __builtin_prefetch (&data[(mid + 1 + high)/2], 0, 3);
      __builtin_prefetch (&data[(low + mid - 1)/2], 0, 3);

     if(data[mid] < Key)
       low = mid + 1;
     else if(data[mid] > Key)
       high = mid-1;
     else
       return mid;
   }   return -1; }
```

Explicit prefetching

```
luca@GGG:~/code/HPC_LECTURES/prefetching% ./prefetching_off
performing 13421772 lookups on 134217728 data..
set-up data.. set-up lookups..
start cycle.. time elapsed: 20.7534
luca@GGG:~/code/HPC_LECTURES/prefetching% ./prefetching_on
performing 13421772 lookups on 134217728 data with prefetching enabled..
set-up data.. set-up lookups..
start cycle.. time elapsed: 12.6204
```

Explicit prefetching

```
Samples: 71K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 13901140
Overhead       Samples  Memory access
  71,08%         42196  Local RAM hit
  24,14%         17022  LFB hit
   4,11%         10967  L3 hit
   0,63%          1714  L1 hit
   0,02%            75  L2 hit
   0,01%            15  L3 miss
   0,00%             1  Uncached hit
```

```
Samples: 61K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 11720387
Overhead       Samples  Memory access
  68,74%         29450  LFB hit
  27,04%         28208  L1 hit
   2,72%           909  Local RAM hit
   1,29%          2983  L3 hit
   0,20%           346  L2 hit
```

Explicit prefetching

Usage of direct prefetching directive is highly uncertain, since it is difficult to spot the exact point – both in the code and in the execution – where to place them (also because your C code is different than the generated assembly code).

Moreover, the “exact point” is very likely dependent on the system you run on, and then it is susceptible to change significantly.

It is normally much safer to re-organize your code so to have **prefetching by pre-loading**.

Prefetching by moral suasion

Let's discuss together this very simple example before putting the hands on the code you find in the `git`

```
elem a = elements[0]
for ( i = 0; i < 4*N_4; i+= 4 )
 {
        elem e = elem[i+4]; // non-blocking miss
        elem b = elem[i+1]; // possible cache-hit
        elem c = elem[i+2]; // possible cache-hit
        elem d = elem[i+3]; // possible cache-hit
        Elaborate(a);
        Elaborate(b);
        Elaborate(c);
        Elaborate(d);
        a = e;
 }
```

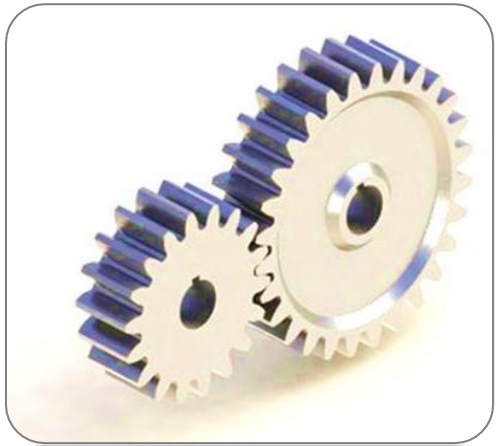
Hands-on

You find code snippets with different flavours of prefetching-by-preloading technique on our GitHub, with some comments about compilation.

Compile and run them with different options (and possibly different compilers) and try to understand what happens on your laptop and/or on Ulisse facility.

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

Some more sparse material



Macros as Templates

NUMA locality

Macros as templates

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

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

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

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

Macros as templates

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

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

```
int MY_WONDER(A, B, C, …) { \

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

Macros as templates

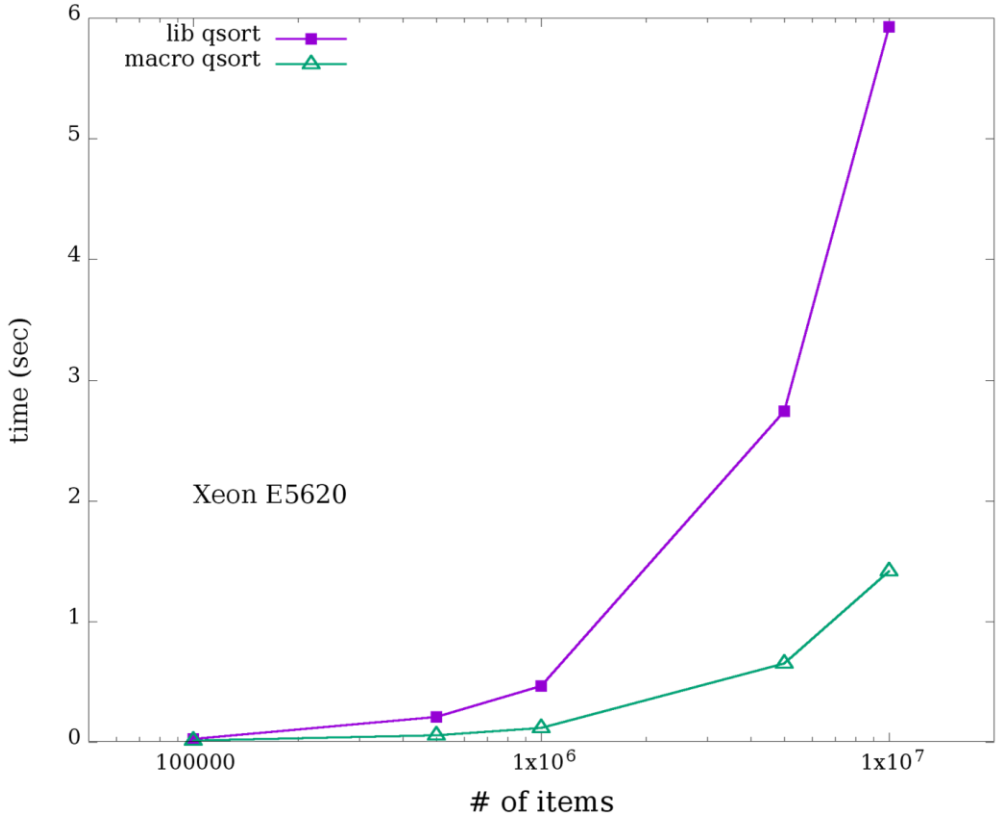
And you define MY_DATA_TYPE and DEALER and whatever else you need just before including the .h file

```
#define MY_DATA_TYPE some_type

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

#include “my_routines.h”
```

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

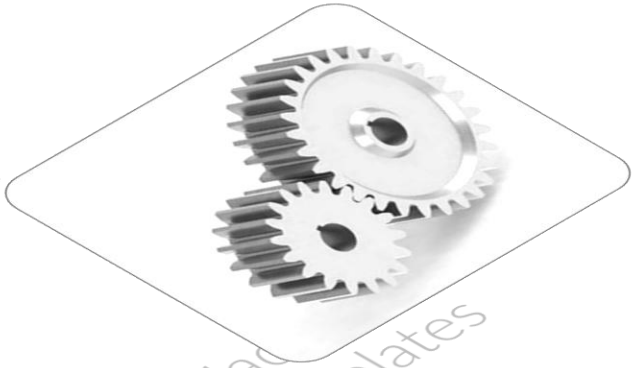
Macros as templates

Going practical, let's try a quicksort on a structure of few hundreds bytes



Some more sparse material

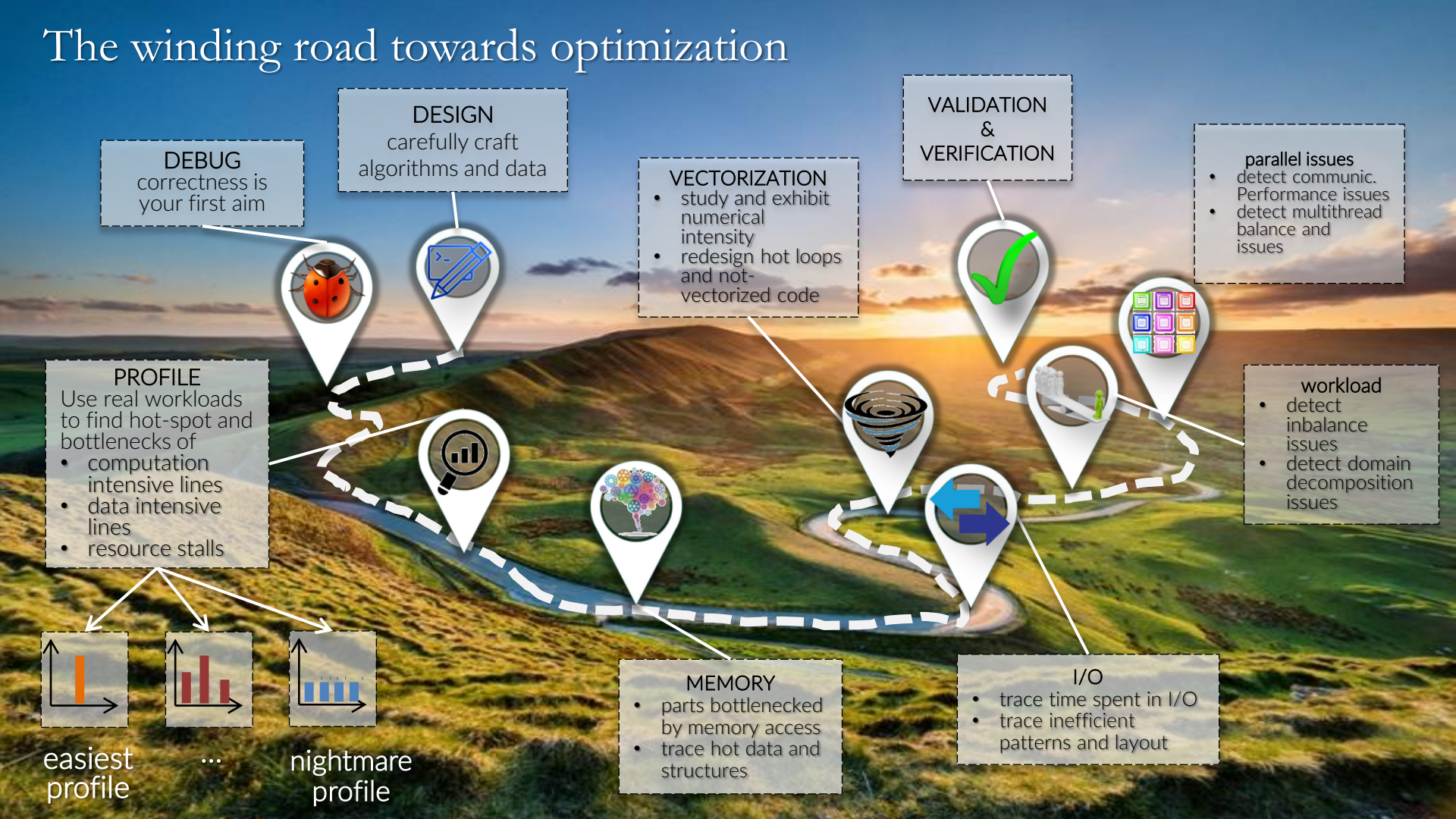
Prefe



Templates

NUMA
locality

The winding road towards optimization



that's all, have fun