Introduction to High-Performance Computing

Giorgio Amati
Alessandro Ceci

Corso di dottorato in Ingegneria Aeronautica e Spaziale 2025

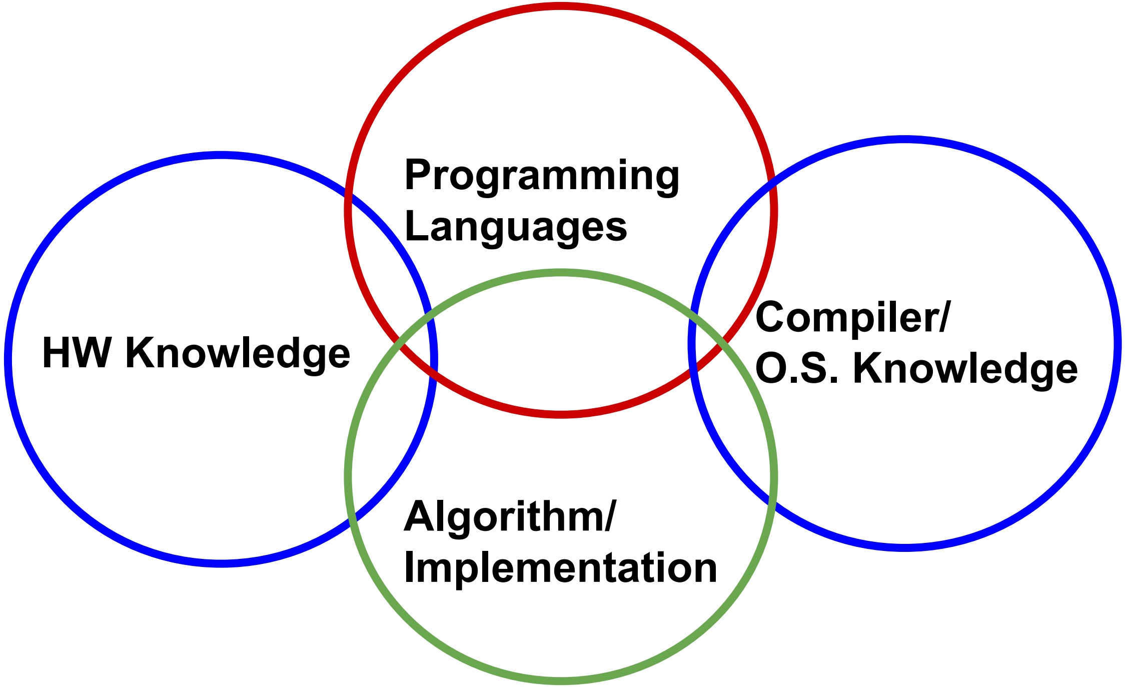
g.amati@cineca.it / g.amaticode@gmail.com
alessandro.ceci@uniroma1.it

Agenda

- ✓ **HPC: What it is?**
- ✓ **Hardware: how it works**
- ✓ Algorithm vs. Implementation
- ✓ Compiler
- ✓ Parallel Paradigm
- ✓ Conclusions & Comments

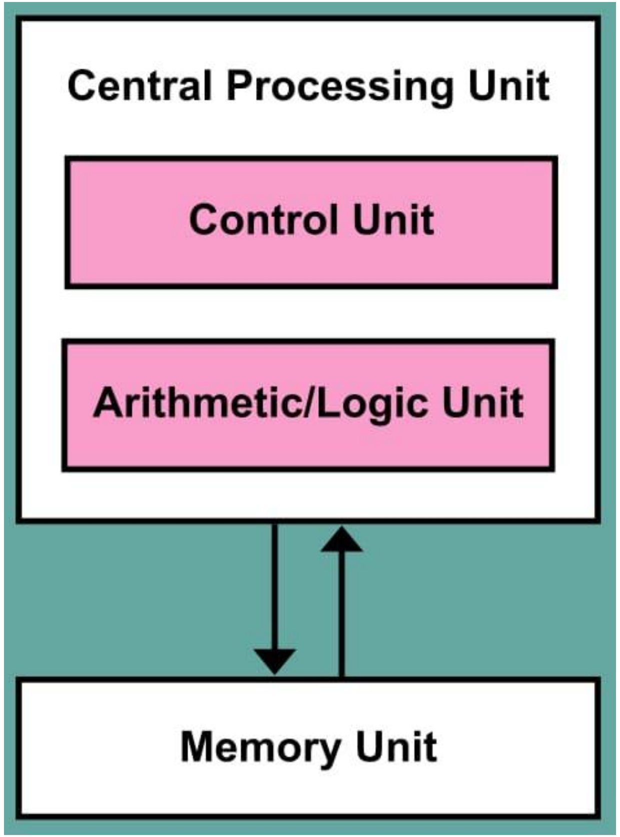
HPC: what it is?

- ✓ These are the main skills for efficient HPC



Von Neumann’s model

- ✓ Independent Systems that talk to each other exchanging data
  - ■ Moving data/instructions
  - ■ Performing operations
- ✓ Data and instructions are stored in the same area
- ✓ Different, and independent, units for instructions (Control Units) and data (ALU units)

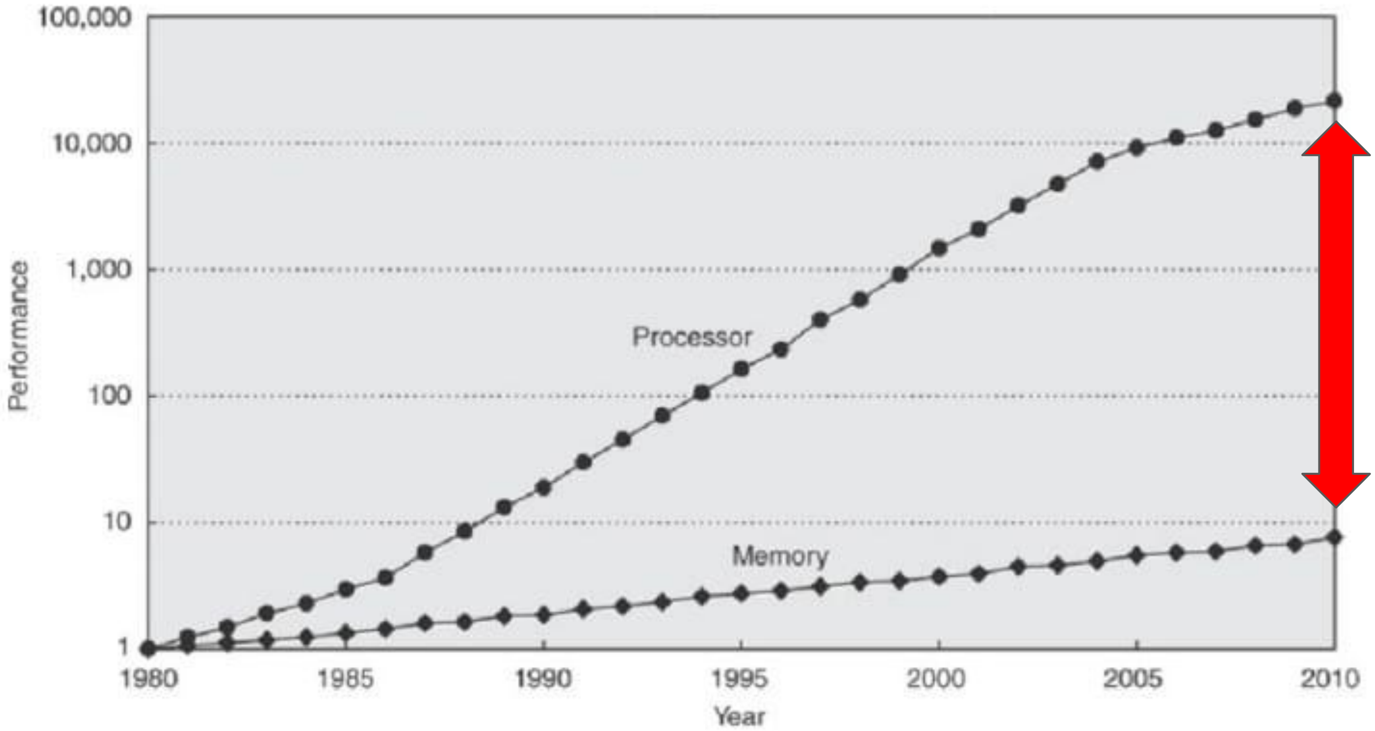


Two key points

- ✓ Data moved back and forth from main memory
- ✓ Instructions (e.g. Flops) performed from Memory

BW & Flops: evolution

✓ Hic sunt leones! (From Hennessy & Patterson, 2011)



Two metrics…

Latency:

- ✓ Time needed to complete one operation (once all the input data are available)
- ✓ Lower is better

Throughput:

- ✓ How many operations can be completed within a time interval, usually clock cycle
- ✓ Higher is better

Real life example:

- ✓ Elevator: **low latency, low throughput**
- ✓ Escalator: **high latency**, **high throughput**

Floating point units (vanilla version)

The big increase in Performance (or Flops) was due to:

- ✓ optimization of the Floating Point Unit (FPU)
- ✓ a big number of FPU/Cores in a CPU (thanks to Moore’s Law)

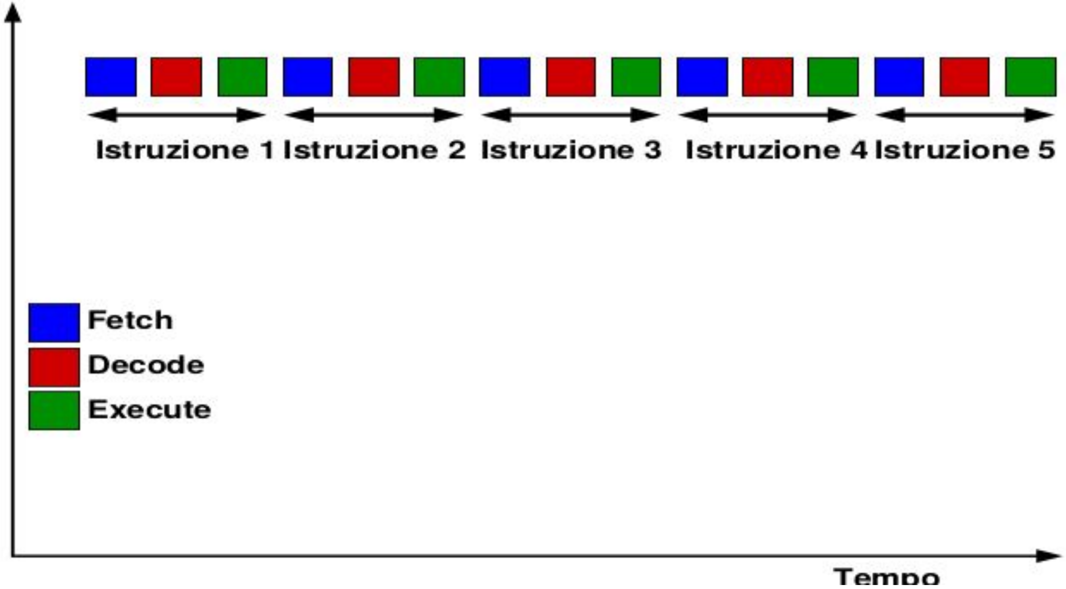
A generic operation can be splitted in different (independent) stages: e.g (but it’s a simplification):

1. Fetch
2. Decode
3. Execute

- ✓ **Let’s assume that each stage can be completed in a single clock cycle**
- ✓ **Let’s assume that the operation is a floating point operation (e.g. a sum or a product)**

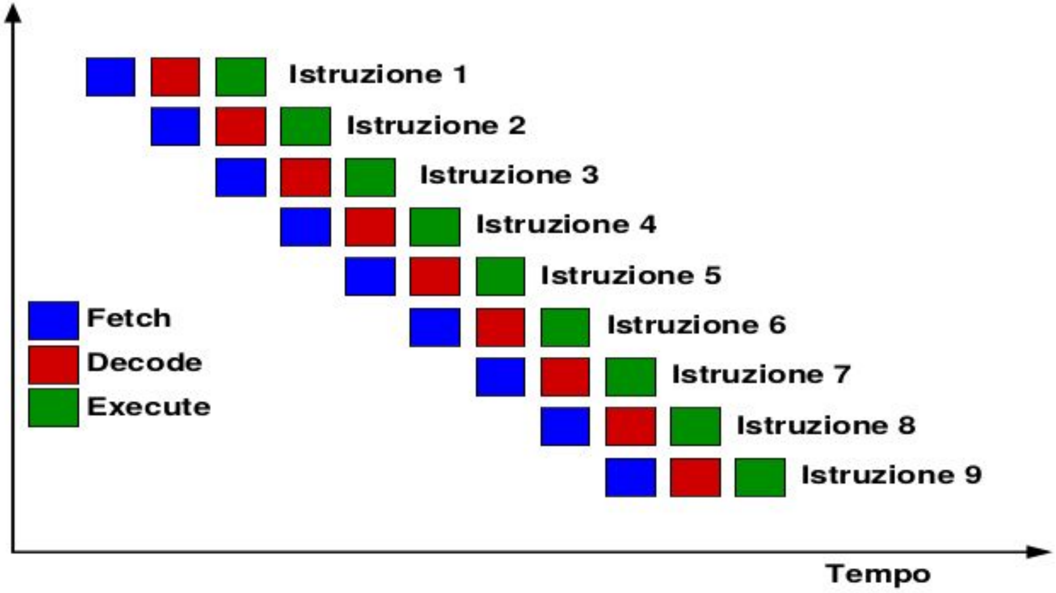
Floating point unit

- ✓ **Simple FPU**: an instruction completed every 3 cycles
- ✓ 1 Flop every 3 cycle: we need to have 2 load+1store every 3 cycles
- ✓ **Latency = 3 cycles**
- ✓ **Throughput = ⅓**
- ✓ **clock=1**



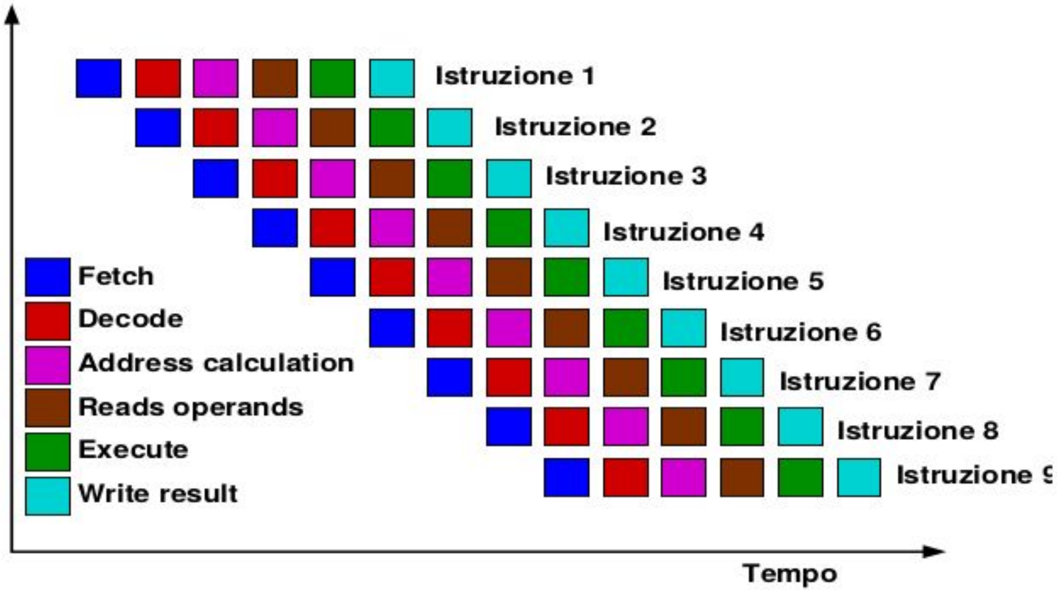
FPU Pipelining

- ✓ Pipelined FPU: all the three stages are independent (different transistors)
- ✓ 1 Flop computed every cycle: we need 2 load+1 store every cycle
- ✓ at least 3 independent instructions (intr #1, #2, #3)
- ✓ **Latency=3 cycles**
- ✓ **Throughput = 1**
- ✓ **clock=1**
- ✓ **Increase=3x**



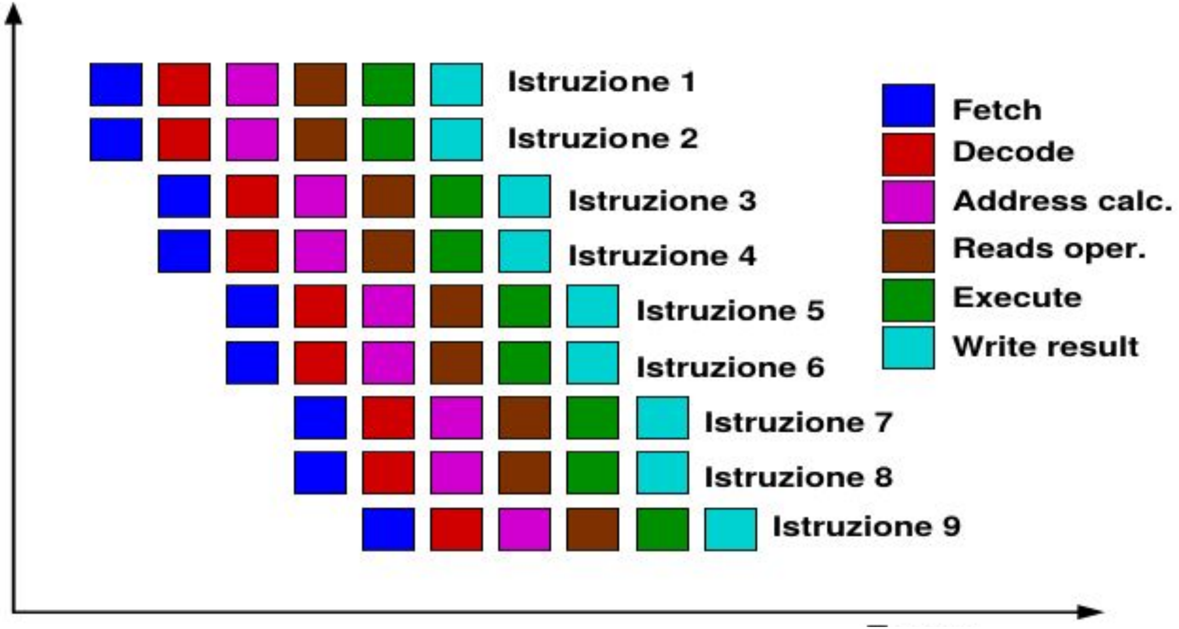
Superpipelining

- ✓ Superpipelined FPU: Further decomposition in more stages
- ✓ We can (ideally) increase by a factor 2 the frequency
- ✓ 1 Flop computed every cycle: we need 2 load+1 store every cycle (but doubled frequency)
- ✓ **Latency = 6 cycles**
- ✓ **Throughput = 1**
- ✓ **clock=1/2**
- ✓ **Increase=6x**



Superscalar

- ✓ More transistor → More (independent) FPU
- ✓ 2 Flop computed every cycle: we need 4 load+2 store every cycle
- ✓ 2 independent instructions **every cycle**
- ✓ **Latency = 6 cycles**
- ✓ **Throughput = 2**
- ✓ **clock=1/2**
- ✓ **Increase=12x**



Out Of Order Execution

✓ “On the fly” reordered instruction in order to avoid “bubbles” in the pipeline

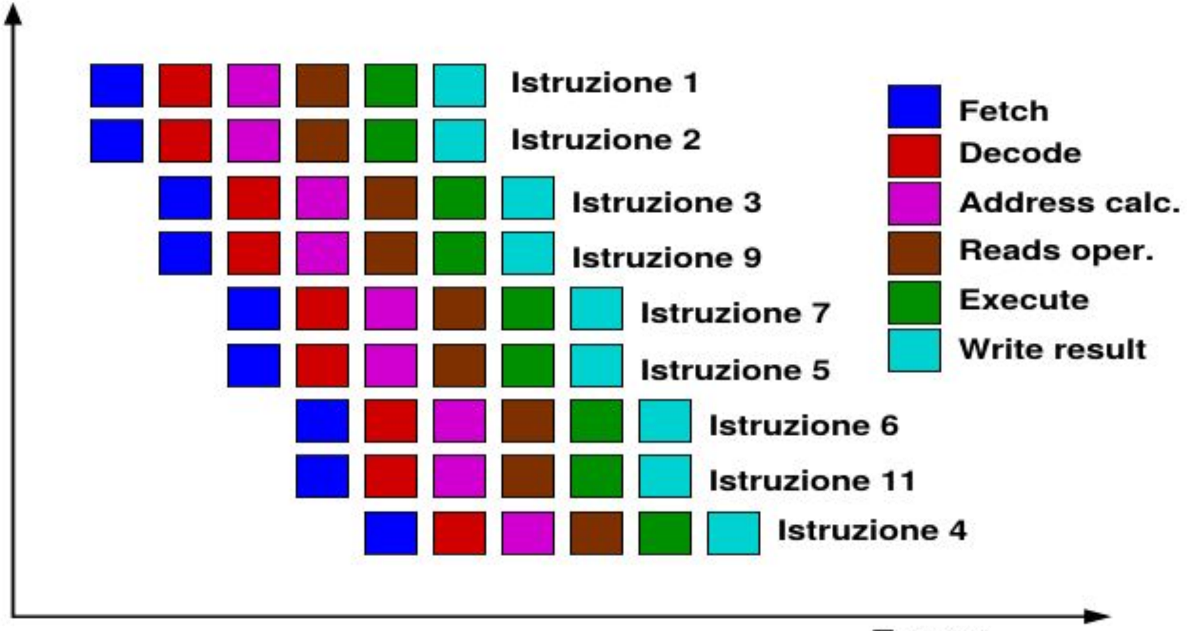
✓ Same request of Superscalar but better performance

✓ **Latency = 6 cycles**

✓ **Throughput = 2**

✓ **clock=1/2**

✓ **Increase>12x**



Pipeline Recap

With pipelining we see a big increase in performance (e.g. #Flops) but we need

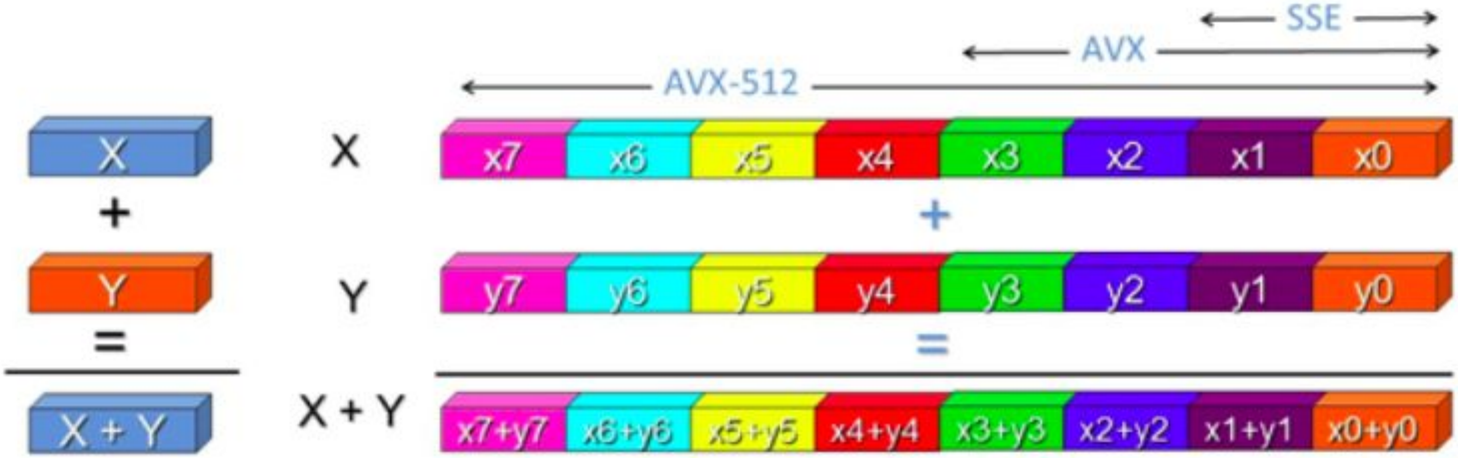
- ✓ for Superscalar FPU
  - ■ 12 times more independent instructions respect a non-pipelined FPU
  - ■ 12 times more data to perform all requested operations
- ✓ Both Mem subsystem & Implementation can help

	FPU	Pipelined	Superpipelined	Superscalar	OOO
stages	1	3	6	6	6
frequency	1	1	1/2	1/2	1/2
performance	1	3	6	12	12
Requested BW	1/3	1	2	4	4

FPU: Vector Unit

Further improvement: operations (e.g. sum) are performed using a vector of 512 byte (8 Double precision or 16 Single precision, AVX-512)

- ✓ 8x improvement respect single scalar unit
- ✓ 8x request of independent instruction
- ✓ 8x more BW requirement



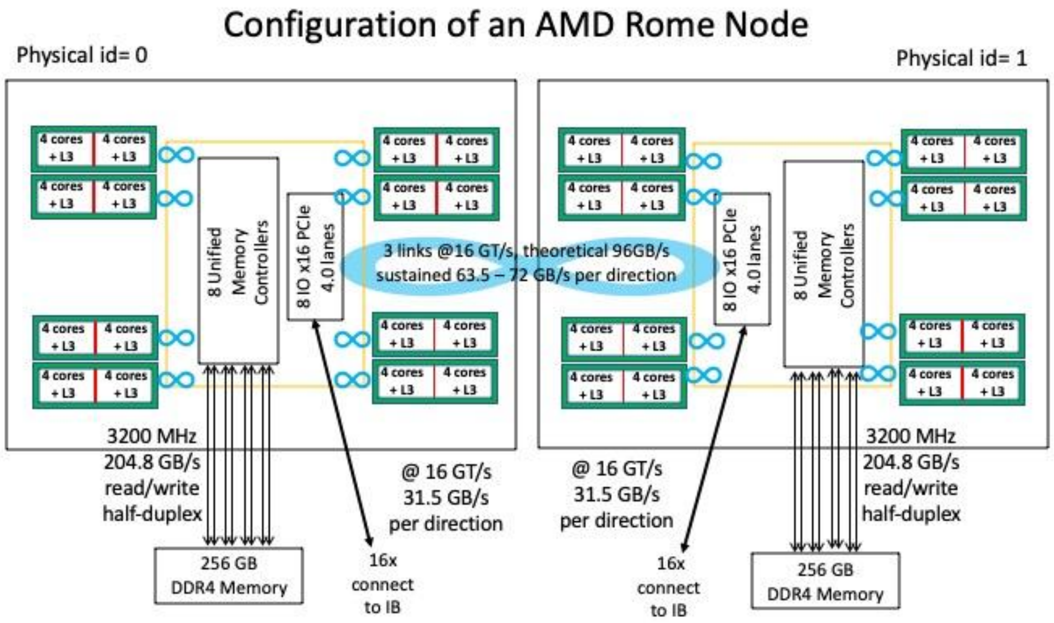
FPU:Further improvement

Other improvements

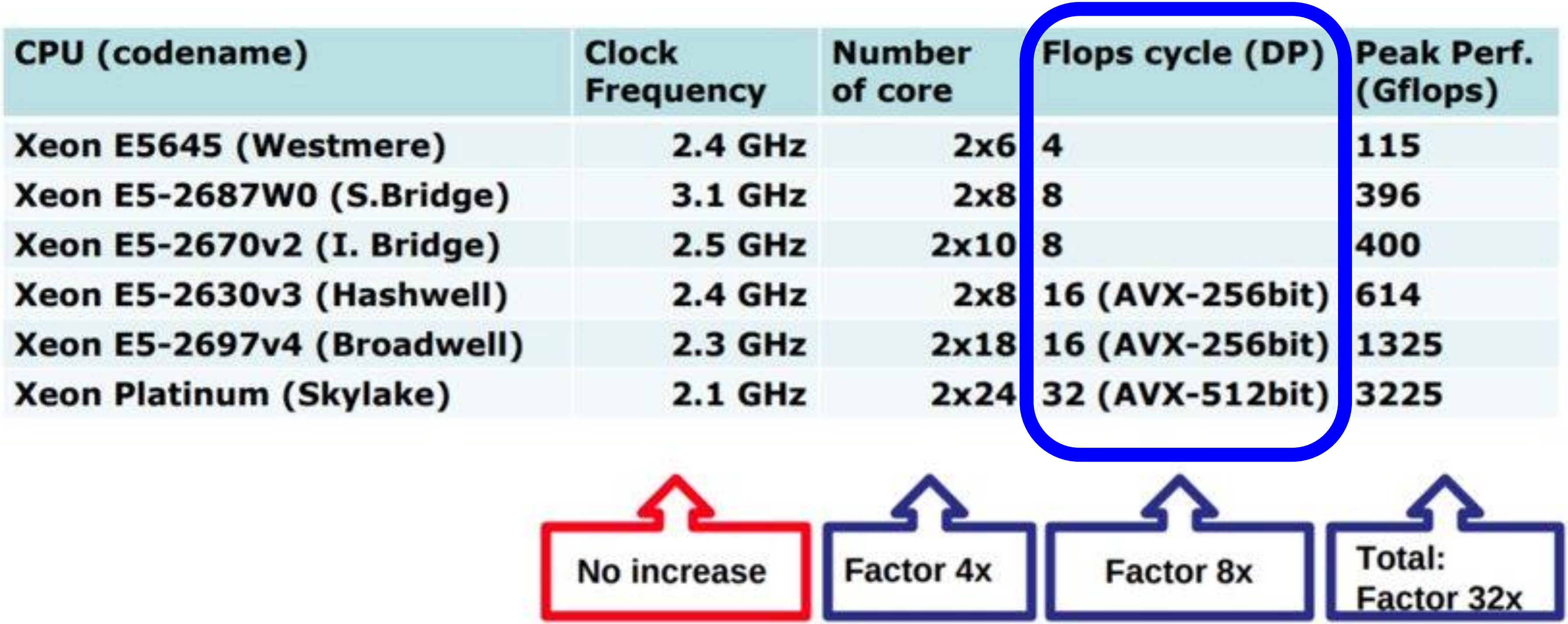
- ✓ FMA
  - ■ **F**used **M**ultiply and **A**dd: FPU able to perform `c=a*x+b` in one cycle
  - ■ what if you don’t have this kind of flops?
- ✓ VLIW: very long instruction word (intel Itanium)
  - ■ https://it.wikipedia.org/wiki/Very_long_instruction_word
  - ■ ILP
  - ■ performance heavily depends (too much) from the compiler
- ✓ Vector register
  - ■ Vector machine use vector register, i.e. 128 vector long register, similar to vector units
  - ■ Used in the “CRAY” period

A today CPU

- ✓ https://www.nas.nasa.gov/hecc/support/kb/amd-rome-processors_658.html
- ✓ **Order of 40 Billion Transistors**
- ✓ 64 Cores: each core
  - ■ **2 FMA units at 256-bit**
  - ■ **4 x86 instructions per cycle**
  - ■ **4 flops per cycle**



Performance evolution/Intel



CPU (codename)	Clock Frequency	Number of core	Flops cycle (DP)	Peak Perf. (Gflops)
Xeon E5645 (Westmere)	2.4 GHz	2x6	4	115
Xeon E5-2687W0 (S.Bridge)	3.1 GHz	2x8	8	396
Xeon E5-2670v2 (I. Bridge)	2.5 GHz	2x10	8	400
Xeon E5-2630v3 (Hashwell)	2.4 GHz	2x8	16 (AVX-256bit)	614
Xeon E5-2697v4 (Broadwell)	2.3 GHz	2x18	16 (AVX-256bit)	1325
Xeon Platinum (Skylake)	2.1 GHz	2x24	32 (AVX-512bit)	3225

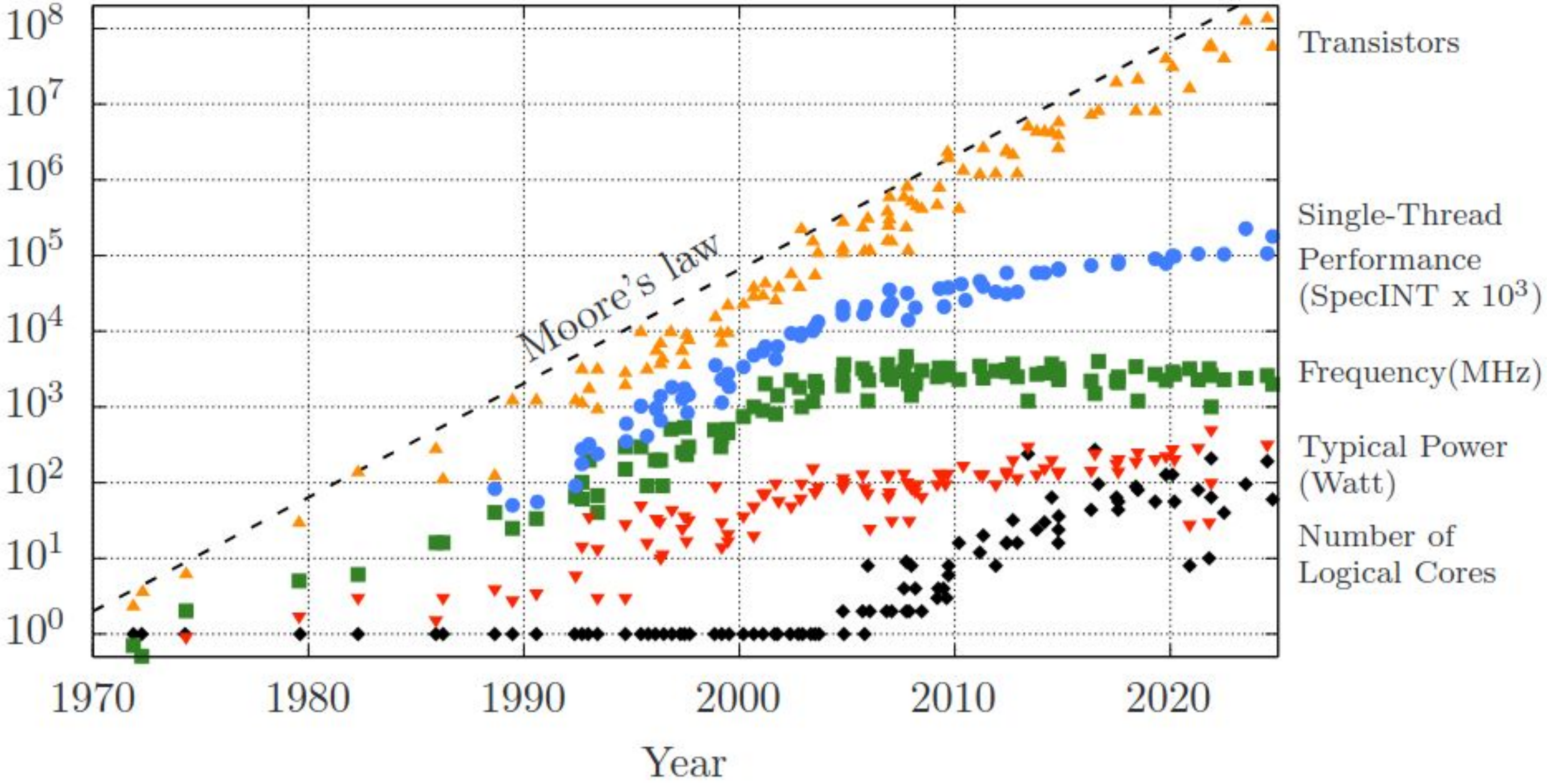
No increase

Factor 4x

Factor 8x

Total:
Factor 32x

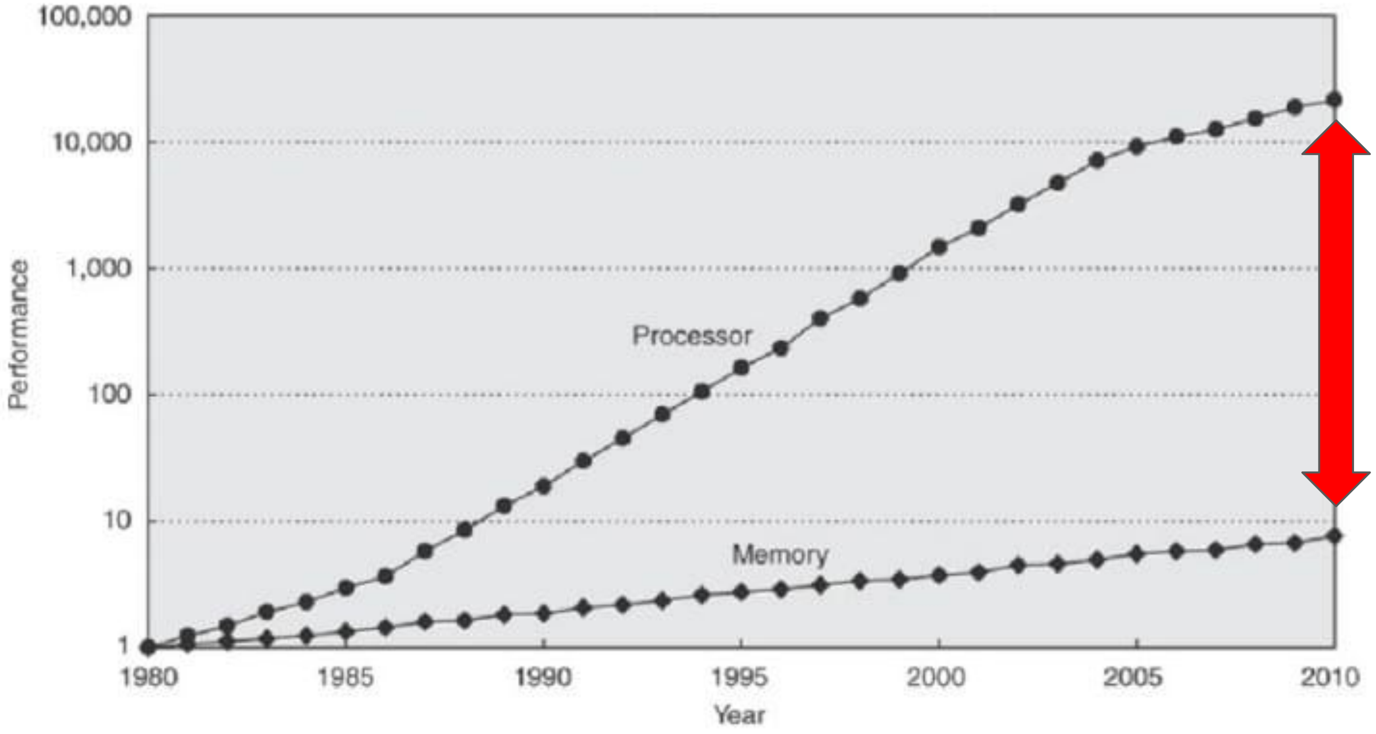
Moore's Law



10^8
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Year
Moore's law
Transistors
Single-Thread Performance (SpecINT x 10^3)
Frequency(MHz)
Typical Power (Watt)
Number of Logical Cores

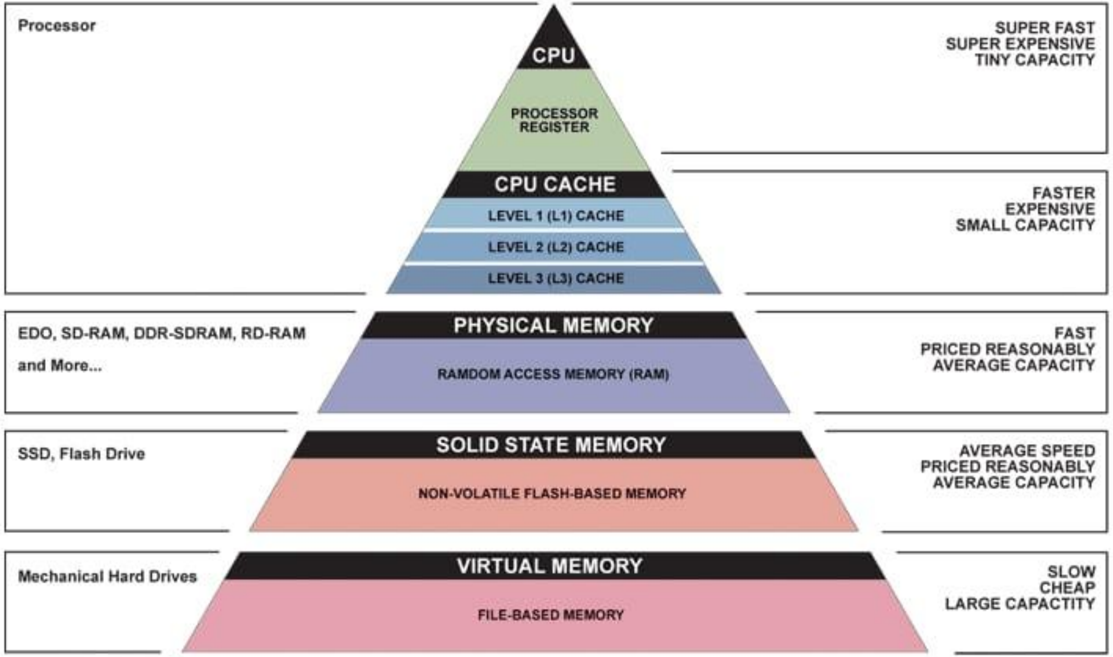
BW & Flops: evolution

✓ Hic sunt leones! (From Hennessy & Patterson, 2011)



Memory Subsystem

Memory subsystem: hierarchical structure



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

Memory Subsystem: velocity vs Size

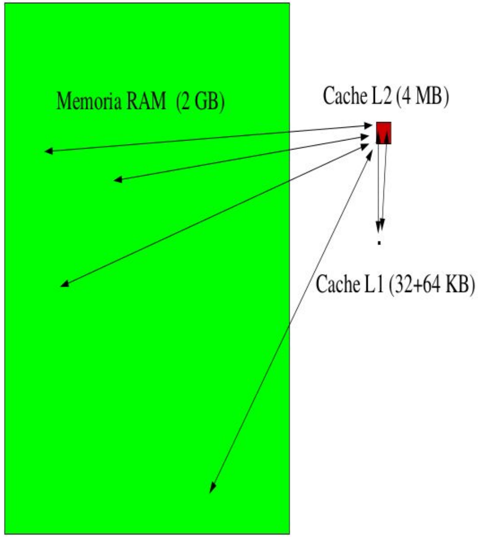
https://www.nas.nasa.gov/hecc/support/kb/amd-rome-processors_658.html

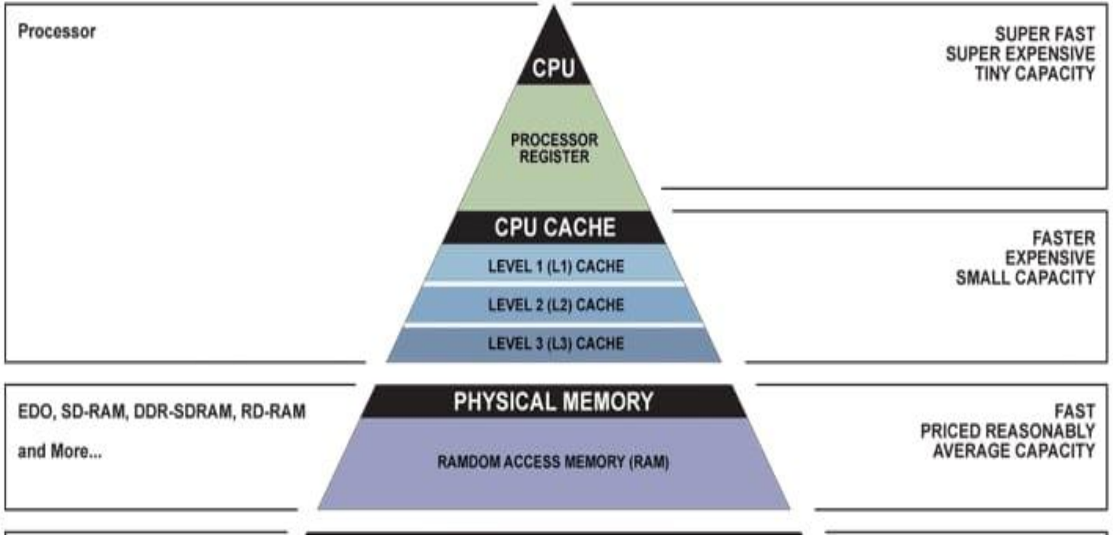
Unit	Size	Latency
Register	~32/128	~1 Cycles
Cache Level 1 (Data)	32 KB	~7 Cycles
Cache Level 1 (istr.)	32 KB	~7 Cycles
Cache Level 2	512 KB (per Core)	~12 Cycles
Cache Level 3	16MB (per 4Core)	~36 Cycles
HMB	~64 GB	~100/1000 Cycles
RAM	> 100 GB	~ 1000/10000 Cycles

Memory Subsystem: velocity

Data movement is expensive: RAM latency is in the order of 1’000 of cycles

- ✓ Intermediate (cache) levels to hide latency
- ✓ All data are stored in RAM memory. In cache we store intermediate results or copies





Principle of Locality

- ✓ Locality is the key point of the memory subsystem structure.
- ✓ Locality is crucial to achieve performance. Without exploiting locality performance are very very low

locality is the tendency of a processor to access the same set of memory locations repetitively over a short period of time

- ✓ Two different localities are implemented:

Spatial (or data) locality

Temporal locality

https://en.wikipedia.org/wiki/Locality_of_reference

Space Locality

Usually a program uses data that are “close” to each other.

- ✓ If at `t`, I need data in location `A`
- ✓ at time `t+1`, I’ll look for a data in location `B` “close” to `A`

Solution

- ✓ When I’ve to “access” to location `A`, I can try to “access” also to location `B`, to hide data access latency (prefetch)

Temporal locality

Usually a program re-use data in a short time-frame

- ✓ If at time `t`, I need data in location `A`
- ✓ at time `t+1, t+2, t+3`,.... I’ll need again the data stored in location `A`

Solution:

- ✓ Store the data in location `A` “somewhere” before writing it back to RAM

keyword

Cache hit:

✓ when a “needed” data is found in a generic cache level
✓ Low latency (< 100 Cycles)

Cache miss:

✓ when a “needed” data is **NOT** found in a generic cache level
✓ It must be “retrieved” from RAM
✓ High latency (~> 1000 Cycles)

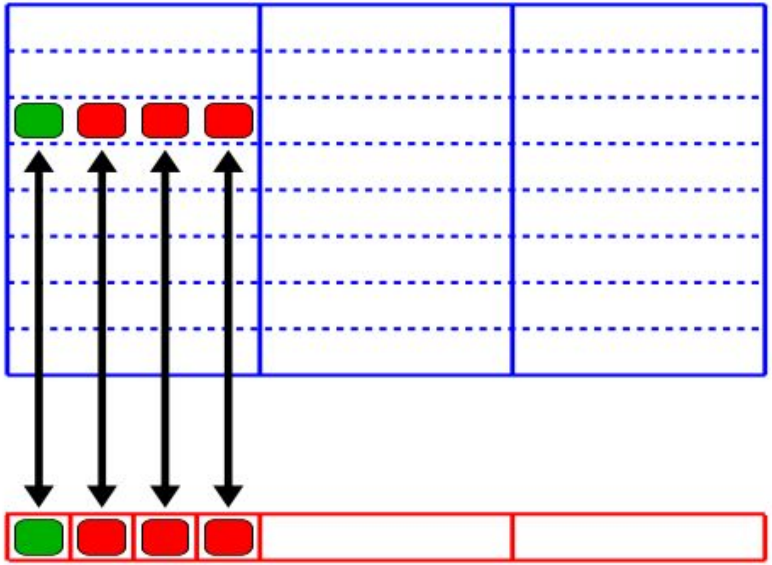
Cache Thrashing:

✓ Because Cache << RAM When to make room to new data I have to throw away “useful” data
✓ It depends from cache-ram associativity

Exploiting locality: Cache line

Data are moved from RAM memory to cache in a cache-line: usually 64 byte wide.

- ✓ RAM latency is order of 1’000 Cycles
- ✓ Cache latency is order of 10 Cycles



- ✓ To exploit Spatial locality
- ✓ FPU “sees” the Cache latency not the RAM one

Temporal locality

Data used many times are stored in cache instead of being copied (every time) back into RAM.

This applies in particular to

✓ Intermediate Results
✓ Constants

Remember: **data in cache are copies of the "original" data in RAM, the OS is "obliged" to synchronize between cache and RAM. e.g arrays**

Cache friendly vs. cache unfriendly

Cache unfriendly

1. Look for A(1)
2. cache miss
3. load from RAM to cache
4. move data to register
5. perform the operation
6. write back the result
7. look for A(2)
8. cache miss
9. load from RAM to cache
10. move data to register
11. perform the operation
12. write back the result
13. look for A(3)
14. cache miss
15. ….

Cache friendly

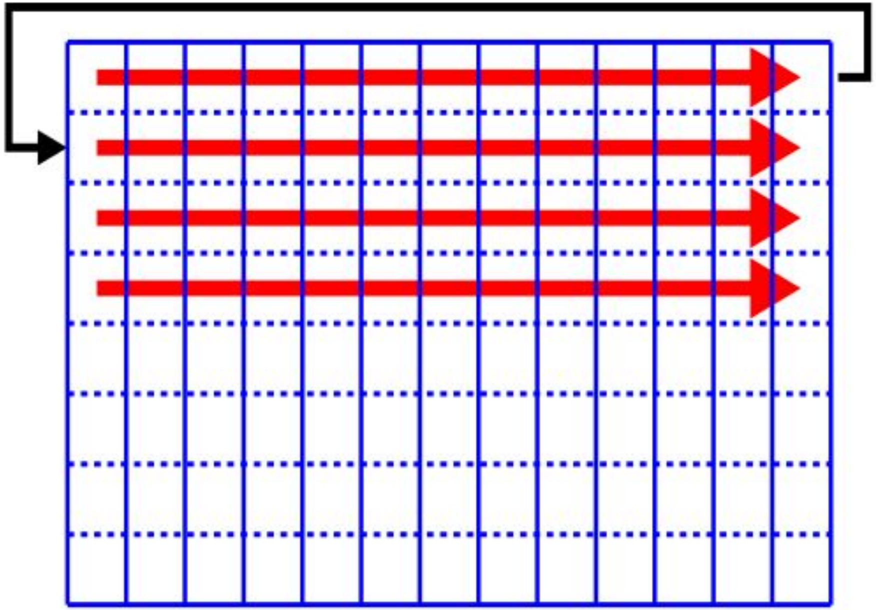
1. Look for A(1)
2. cache miss
3. load from RAM to cache
4. move data to register
5. perform the operation
6. write back the result
7. look for A(2)
8. cache hit
9. move data to register
10. perform the operation
11. write back the result
12. look for A(3)
13. cache hit
14. move data to register
15. ….

Data allocation: C

- ✓ Programmer style/languages could have an impact in the performance
- ✓ Memory is a linear Array: how to map matrix `A[i][j]`?

- ✓ C is row oriented
- ✓ In C a matrix is a “vector of vectors”

- ✓ `A[1][1], A[1][2] → stride=1`
- ✓ `A[1][1], A[2][1] → stride=n`

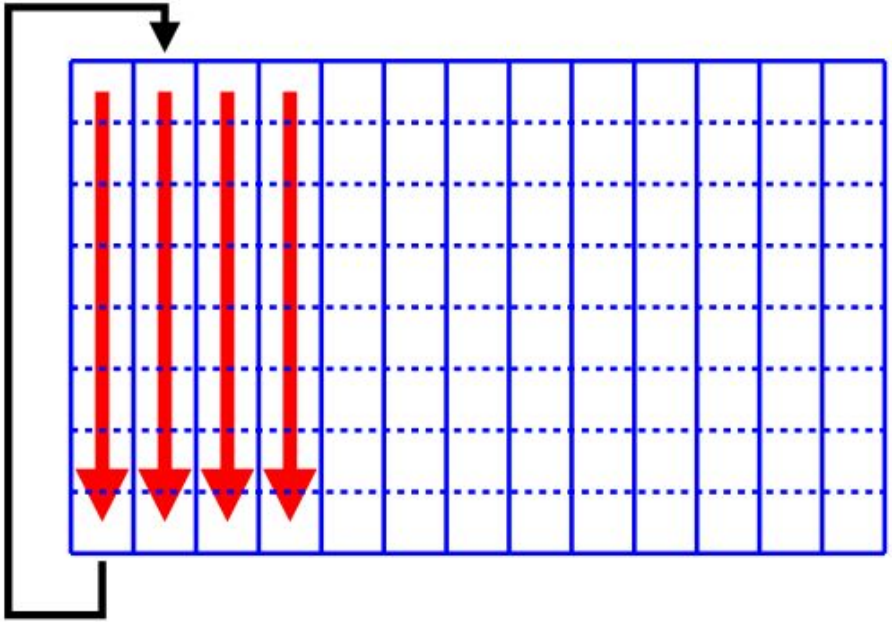


Data allocation: Fortran

- ✓ Programmer style/languages could have an impact in the performance
- ✓ Memory is a linear Array: how to map matrix `A(i,j)`?

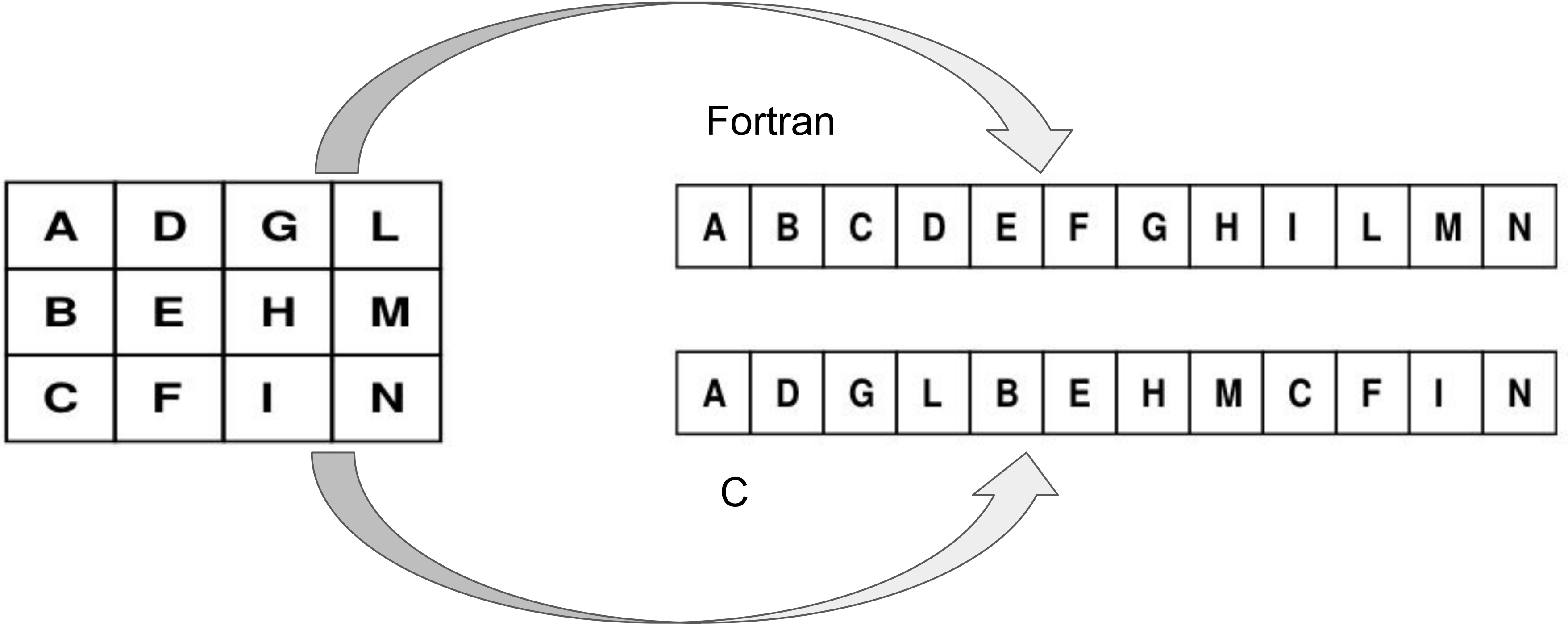
- ✓ Fortran is column oriented

- ✓ `A(1,1), A(1,2) → stride=n`
- ✓ `A(1,1), A(2,1) → stride=1`



Data allocation: effects

The matrix (left) is allocated in a completed different way between Fortran and C



A	D	G	L
B	E	H	M
C	F	I	N

A	B	C	D	E	F	G	H	I	L	M	N

A	D	G	L	B	E	H	M	C	F	I	N

matrix-matrix multiplication (Fortran)

Both codes are correct : but which one has unitary stride?

```
do j = 1, n
   do k = 1, n
      do i = 1, n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
      enddo
   enddo
enddo
```

```
do i = 1, n
   do k = 1, n
      do j = 1, n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
      enddo
   enddo
enddo
```

matrix-matrix multiplication (C)

Both codes are correct : but which one has unitary stride?

```
for (i = 0; i < nn; i++)
  for (k = 0; k < nn; k++)
    for (j = 0; j < nn; j++)
      c[i][j]=c[i][j]+a[i][k]*b[k][j];
```

```
for (j = 0; j < nn; j++)
  for (k = 0; k < nn; k++)
    for (i = 0; i < nn; i++)
      c[i][j]=c[i][j]+a[i][k]*b[k][j];
```

Some figure

Performance in Mflops (higher is better)

- ✓ HW: Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz
- ✓ Compiler intel 2021.5
  - `ifort -O1`
  - `icc -O1`
- ✓ Matrix size: 2048 (96 MB)

Index order	Language	Mflops
i,k,j	Fortran	**2870**
j,k,i	Fortran	**129**
i,k,j	C	**181**
j,k,i	C	**2373**

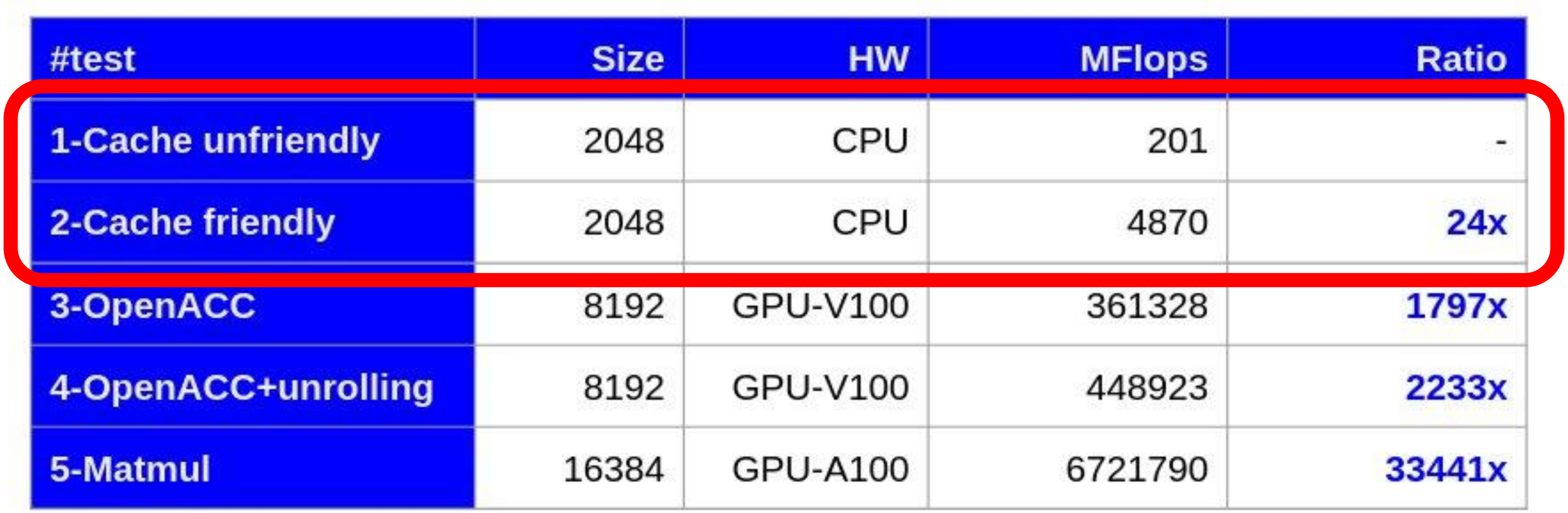
Example: matrix-matrix multiplication

✓ Performance can be really different, depending on HW, implementation etc….

✓ Improvement in performance can be really high….

✓ ….or you can easily “depress” performance

✓ Performance in Mflops: the higher the better



#test	Size	HW	MFlops	Ratio
1-Cache unfriendly	2048	CPU	201	-
2-Cache friendly	2048	CPU	4870	24x
3-OpenACC	8192	GPU-V100	361328	1797x
4-OpenACC+unrolling	8192	GPU-V100	448923	2233x
5-Matmul	16384	GPU-A100	6721790	33441x

SIze matters!!!!

✓ Performance, in MFlops, for different problem sizes: from `n=100^2` to `n=4000^2`

- using cache friendly access (stride=1)
- using cache un-friendly access (stride=n)

	256	512	769	1024	1536	2048
`stride=n`	268	367	335	113	71	49
`stride=1`	1073	2643	2828	2290	2273	1683
Ratio	**4.0x**	**7.2x**	**8.4x**	**20x**	**32x**	**34x**

Data allocation: SoA or AoS?

The way we allocate data force the way we access to them!

e.g.: Velocity for a cartesian grid

- ✓ **Structure of Arrays (SoA)**: each element of the structure is an array
- ✓ **Arrays of Structures (AoS):** each element of the array is a structure

```
…
! SoA
! vel=1,2,3 where 1=vx, 2=vy, 3=vz
a(i,j,k,vel) =  …
…
```

```
…
! AoS
! vel=1,2,3 where 1=vx, 2=vy, 3=vz
a(vel,i,j,k) =  …
…
```

- ✓ Which is faster?

Exploiting Cache usage: blocking

Matrix Transpose

✓ Or load or store with not unitary stride

✓ Split in different blocks, little enough to fit in cache

```
do j = 1, n
   do i = 1, n
      B(i,j) = A(j,i)
   enddo
enddo
```

```
do jj = 1, n, step
   do ii = 1, n, step
      do j = jj, jj+step-1
         do i = ii, ii+step-1
            B(i,j) = A(j,i)
         enddo
      enddo
   enddo
enddo
```

Exploiting performance: unrolling

Unrolling exploits instruction independence

- ✓ Allows to increase the number of data streams
- ✓ be careful to index extremes
- ✓ can introduce systematic cache trashing

```
do j = 1, n
  do k = 1, n
    do i = 1, n
       c(i,j) = c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

```
do j = 1, n, 4
  do k = 1, n
    do i = 1, n
       c(i,j+0)=c(i,j+0)+a(i,k)*b(k,j+0)
       c(i,j+1)=c(i,j+1)+a(i,k)*b(k,j+1)
       c(i,j+2)=c(i,j+2)+a(i,k)*b(k,j+2)
       c(i,j+3)=c(i,j+3)+a(i,k)*b(k,j+3)
    enddo
  enddo
enddo
```

Exploiting performance: unrolling

Unrolling exploits instruction independence

- ✓ Allows to increase the number of data streams
- ✓ be careful to index
- ✓ can introduce systematic cache trashing

```
do j = 1, n
  do k = 1, n, 2
     do i = 1, n
       c(i,j)=c(i,j)             &
              +a(i,k+0)*b(k+0,j) &
              +a(i,k+1)*b(k+1,j) &
     enddo
  enddo
enddo
```

```
do j = 1, n
  do k = 1, n
    do i = 1, n, 4
       c(i+0,j)=c(i+0,j)+a(i+0,k)*b(k,j)
       c(i+1,j)=c(i+1,j)+a(i+1,k)*b(k,j)
       c(i+2,j)=c(i+2,j)+a(i+2,k)*b(k,j)
       c(i+3,j)=c(i+3,j)+a(i+3,k)*b(k,j)
    enddo
  enddo
enddo
```

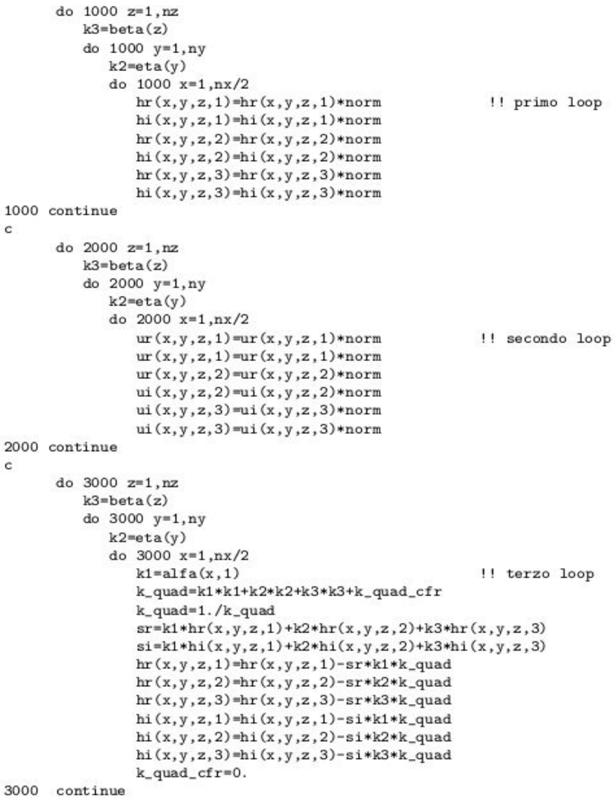
Some further figure

Performance in Mflops (higher is better)

- ✓ HW: Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz
- ✓ Compiler intel 2021.5
  - `ifort -O1`
- ✓ Matrix size: 2048 (96 MB)

Optimization	Mflops
simple	2210
blocking (32)	2928
unrolling x 4 (j)	3267
unrolling x 4 (i)	2530
unrolling x 2 (k)	3420
All together	5653

Exploiting performance: loop merging

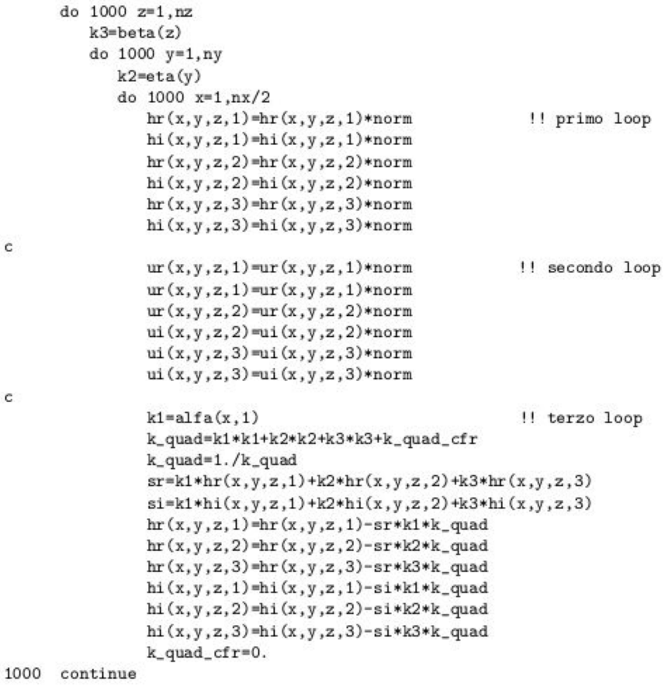


```
      do 1000 z=1,nz
         k3=beta(z)
         do 1000 y=1,ny
            k2=eta(y)
            do 1000 x=1,nx/2
               hr(x,y,z,1)=hr(x,y,z,1)*norm             !! primo loop
               hi(x,y,z,1)=hi(x,y,z,1)*norm
               hr(x,y,z,2)=hr(x,y,z,2)*norm
               hi(x,y,z,2)=hi(x,y,z,2)*norm
               hr(x,y,z,3)=hr(x,y,z,3)*norm
               hi(x,y,z,3)=hi(x,y,z,3)*norm
1000 continue
c
      do 2000 z=1,nz
         k3=beta(z)
         do 2000 y=1,ny
            k2=eta(y)
            do 2000 x=1,nx/2
               ur(x,y,z,1)=ur(x,y,z,1)*norm            !! secondo loop
               ur(x,y,z,1)=ur(x,y,z,1)*norm
               ur(x,y,z,2)=ur(x,y,z,2)*norm
               ui(x,y,z,2)=ui(x,y,z,2)*norm
               ui(x,y,z,3)=ui(x,y,z,3)*norm
               ui(x,y,z,3)=ui(x,y,z,3)*norm
2000 continue
c
      do 3000 z=1,nz
         k3=beta(z)
         do 3000 y=1,ny
            k2=eta(y)
            do 3000 x=1,nx/2
               k1=alfa(x,1)                            !! terzo loop
               k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
               k_quad=1./k_quad
               sr=k1*hr(x,y,z,1)+k2*hr(x,y,z,2)+k3*hr(x,y,z,3)
               si=k1*hi(x,y,z,1)+k2*hi(x,y,z,2)+k3*hi(x,y,z,3)
               hr(x,y,z,1)=hr(x,y,z,1)-sr*k1*k_quad
               hr(x,y,z,2)=hr(x,y,z,2)-sr*k2*k_quad
               hr(x,y,z,3)=hr(x,y,z,3)-sr*k3*k_quad
               hi(x,y,z,1)=hi(x,y,z,1)-si*k1*k_quad
               hi(x,y,z,2)=hi(x,y,z,2)-si*k2*k_quad
               hi(x,y,z,3)=hi(x,y,z,3)-si*k3*k_quad
               k_quad_cfr=0.
3000  continue
```

FPU needs a lot of data to be “full”.

Sometimes a single loop has not enough instruction to be efficient

It increases the number on independent instructions



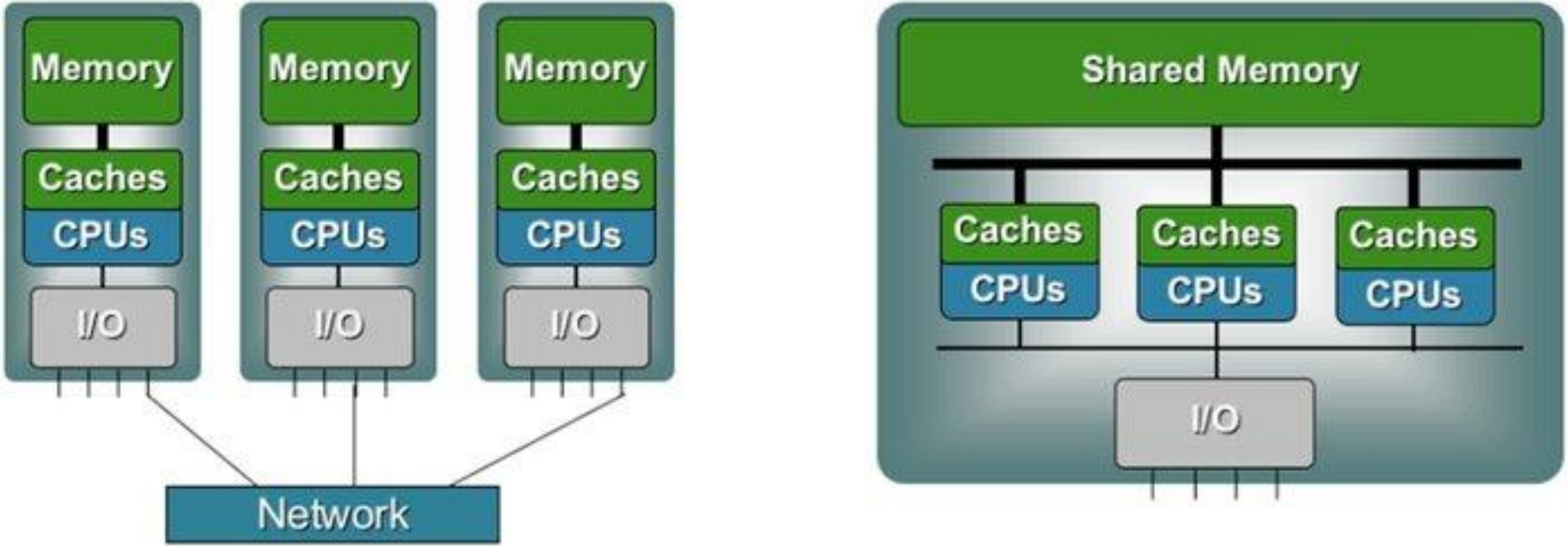
```
      do 1000 z=1,nz
         k3=beta(z)
         do 1000 y=1,ny
            k2=eta(y)
            do 1000 x=1,nx/2
               hr(x,y,z,1)=hr(x,y,z,1)*norm             !! primo loop
               hi(x,y,z,1)=hi(x,y,z,1)*norm
               hr(x,y,z,2)=hr(x,y,z,2)*norm
               hi(x,y,z,2)=hi(x,y,z,2)*norm
               hr(x,y,z,3)=hr(x,y,z,3)*norm
               hi(x,y,z,3)=hi(x,y,z,3)*norm
c
               ur(x,y,z,1)=ur(x,y,z,1)*norm            !! secondo loop
               ur(x,y,z,1)=ur(x,y,z,1)*norm
               ur(x,y,z,2)=ur(x,y,z,2)*norm
               ui(x,y,z,2)=ui(x,y,z,2)*norm
               ui(x,y,z,3)=ui(x,y,z,3)*norm
               ui(x,y,z,3)=ui(x,y,z,3)*norm
c
               k1=alfa(x,1)                            !! terzo loop
               k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
               k_quad=1./k_quad
               sr=k1*hr(x,y,z,1)+k2*hr(x,y,z,2)+k3*hr(x,y,z,3)
               si=k1*hi(x,y,z,1)+k2*hi(x,y,z,2)+k3*hi(x,y,z,3)
               hr(x,y,z,1)=hr(x,y,z,1)-sr*k1*k_quad
               hr(x,y,z,2)=hr(x,y,z,2)-sr*k2*k_quad
               hr(x,y,z,3)=hr(x,y,z,3)-sr*k3*k_quad
               hi(x,y,z,1)=hi(x,y,z,1)-si*k1*k_quad
               hi(x,y,z,2)=hi(x,y,z,2)-si*k2*k_quad
               hi(x,y,z,3)=hi(x,y,z,3)-si*k3*k_quad
               k_quad_cfr=0.
1000  continue
```

Exploiting performance: loop splitting

The opposite of loop merging: sometimes a loop is so “fat” that the compiler is not able to efficiently fetch enough independent instructions: It works on a finite window of lines and cannot have a complete vision of the operation to be performed

- ✓ Vectorization
- ✓ It’s a “try and error” procedure

Spoiler: Distributed & Shared Memory systems



(some) Keywords

- ✓ **Bandwidth**: the rate of data transfer from/to CPU and Memory Subsystem
- ✓ **FMA FPU**: Fused Multiply and ADD FPU
- ✓ **Latency**: time delay between the start and the end of the operation (usually in cycles)
- ✓ **Cache:** is a hardware or software component that stores data so that future requests for that data can be served faster
- ✓ **Cache hit/miss:** when a data is found/not found in che cache
- ✓ **Cache Trashing:** systematic trashing of a cache line due to different data stream insisting on the same cache line
- ✓ **Stride:** distance in memory between two different consecutive access
- ✓ **Spatial locality (also data locality):** data/instruction elements within relatively close storage locations
- ✓ **Temporal locality:** data/instruction elements that will be used again soon

RECAP

- ✓ Today HW is very performing if, and only if, you perform
  - ■ an efficient use of the FPUs
  - ■ an efficient use of the Memory sub-system

- ✓ From user point of view
  - ■ Take care of data allocation
  - ■ Access at unitary stride if possible
  - ■ Exploit independent instructions
  - ■ Exploit data streams
  - ■ **A program can be written in many different way, all correct but with very different performance figures!**

Take home message

- ✓ HW is developed to reach high performance but in particular configuration
  - ■ high data reuse
  - ■ many independent stream of data
  - ■ Instruction level parallelism ILP
- ✓ Programmer can exploit or depress performance in many different ways
  - ■ coding in a not-cache-friendly way
  - ■ not exploiting compiler options
  - ■ implementing “not suitable” algorithm
  - ■ exploiting data parallelism

Topic not covered here/1

Virtual memory, or **virtual storage**, is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine which "creates the illusion to users of a very large (main) memory". The computer's operating system, using a combination of hardware and software, maps memory addresses used by a program, called *virtual addresses*, into *physical addresses* in computer memory. Main storage, as seen by a process or task, appears as a contiguous address space or collection of contiguous segments. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. Address translation hardware in the CPU, often referred to as a memory management unit (MMU), automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities, utilizing, e.g., disk storage, to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer….

https://en.wikipedia.org/wiki/Virtual_memory

Topic not covered here/2

Cache Associativity: Associativity is the size of these sets, or, in other words, *how many different cache lines each data block can be mapped to*.

https://en.algorithmica.org/hpc/cpu-cache/associativity/

In computer science, **thrashing** occurs in a system with virtual memory when a computer's real storage resources are *overcommitted*, leading to a constant state of paging and page faults, slowing most application-level processing.[1] This causes the performance of the computer to degrade or even collapse. The situation can continue indefinitely until the user closes some running applications or the active processes free up additional virtual memory resources.

.

https://en.wikipedia.org/wiki/Thrashing_(computer_science)

Some Books/Reference

✓ Charles Severance; Kevin Dowd “High Performance Computing”, O’Reilly, ISBN 13:9781565923126

✓ John L. Hennessy, David A. Patterson , “Computer Architecture: A Quantitative Approach” Morgan Kaufmann; ISBN-10 : 0128119055

✓ John L. Hennessy, David A. Patterson , “Computer Organization and Design: The Hardware/Software Interface”, Morgan Kaufmann;

✓ D. Goldberg, ”What Every Computer Scientist Should Know About Floating-Point Arithmetic”

✓ U. Drepper: “What Every Programmer Should Know About Memory”