Computer architectures

Matthieu Haefele, Maison de la Simulation

Bastien Di Pierro, LMFA Université Claude Bernard Lyon 1

Saclay, May 2019

Previously in Optim workshop...

```
a[i] = b[i] + c[i];
```

$\hspace{3cm} \downarrow$ Compiler / Interpreter / JIT

```
LOAD r1, b[i]
LOAD r2, c[i]
ADD  r3, r1, r2
STORE a[i], r3
```

Debunking ideas

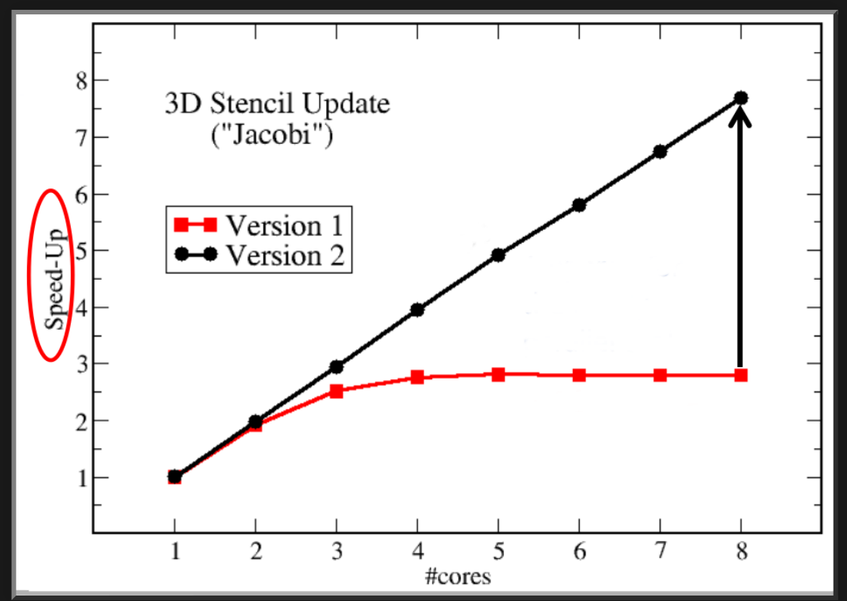
Lore1

In a world of highly parallel computer architectures only highly scalable codes will survive

Single core performance no longer matters since we have so many of them and use scalable codes

Debunking ideas

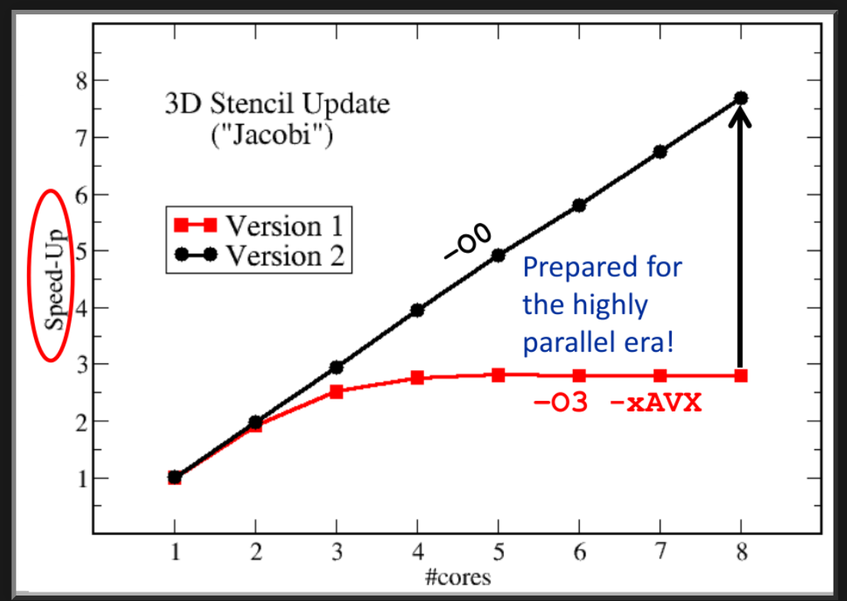
```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj
    do i = 1 , Ni
       y(i,j,k)= b*( x(i-1,j,k)+ x(i+1,j,k)+ x(i,j-1,k)+
                     x(i,j+1,k)+ x(i,j,k-1)+ x(i,j,k+1))
    enddo
  enddo
enddo
```




3.2

Debunking ideas

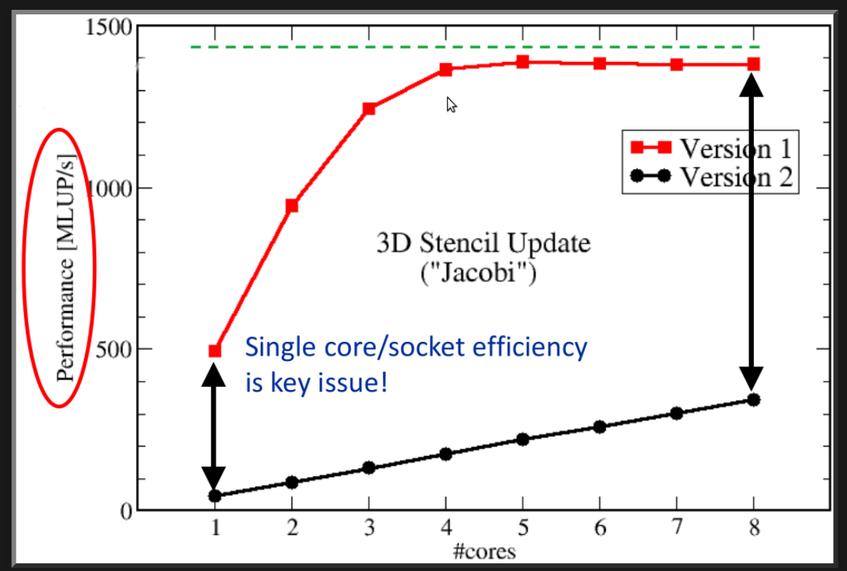
```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj
    do i = 1 , Ni
       y(i,j,k)= b*( x(i-1,j,k)+ x(i+1,j,k)+ x(i,j-1,k)+
                     x(i,j+1,k)+ x(i,j,k-1)+ x(i,j,k+1))
    enddo
  enddo
enddo
```




3.3

Debunking ideas

```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj
    do i = 1 , Ni
       y(i,j,k)= b*( x(i-1,j,k)+ x(i+1,j,k)+ x(i,j-1,k)+
                     x(i,j+1,k)+ x(i,j,k-1)+ x(i,j,k+1))
    enddo
  enddo
enddo
```




3.4

Debunking ideas

HPC **is not only** about scalability !

HPC and more generally optimisation **is about** running at the **bottleneck of the hardware** !

Outline

- Introduction
- CPU core
- CPU socket and node
- CPU based cluster
- Scalability
- Some words on GPUs and FPGAs

Some definitions and units

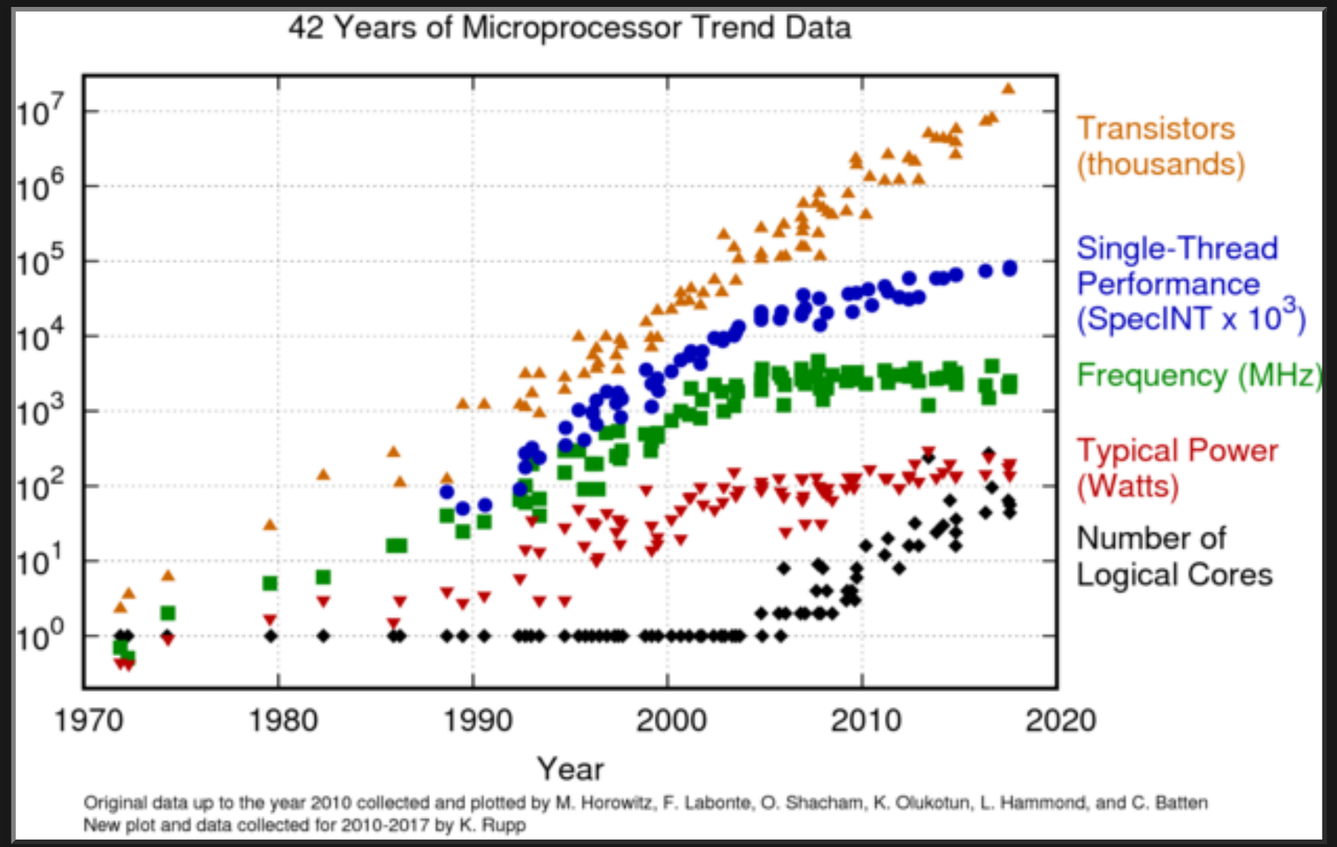
Data size [B]: 1B is able to store the value of 8 bits

Latency [s]: round trip time for an 0-sized information to flow from one point to another

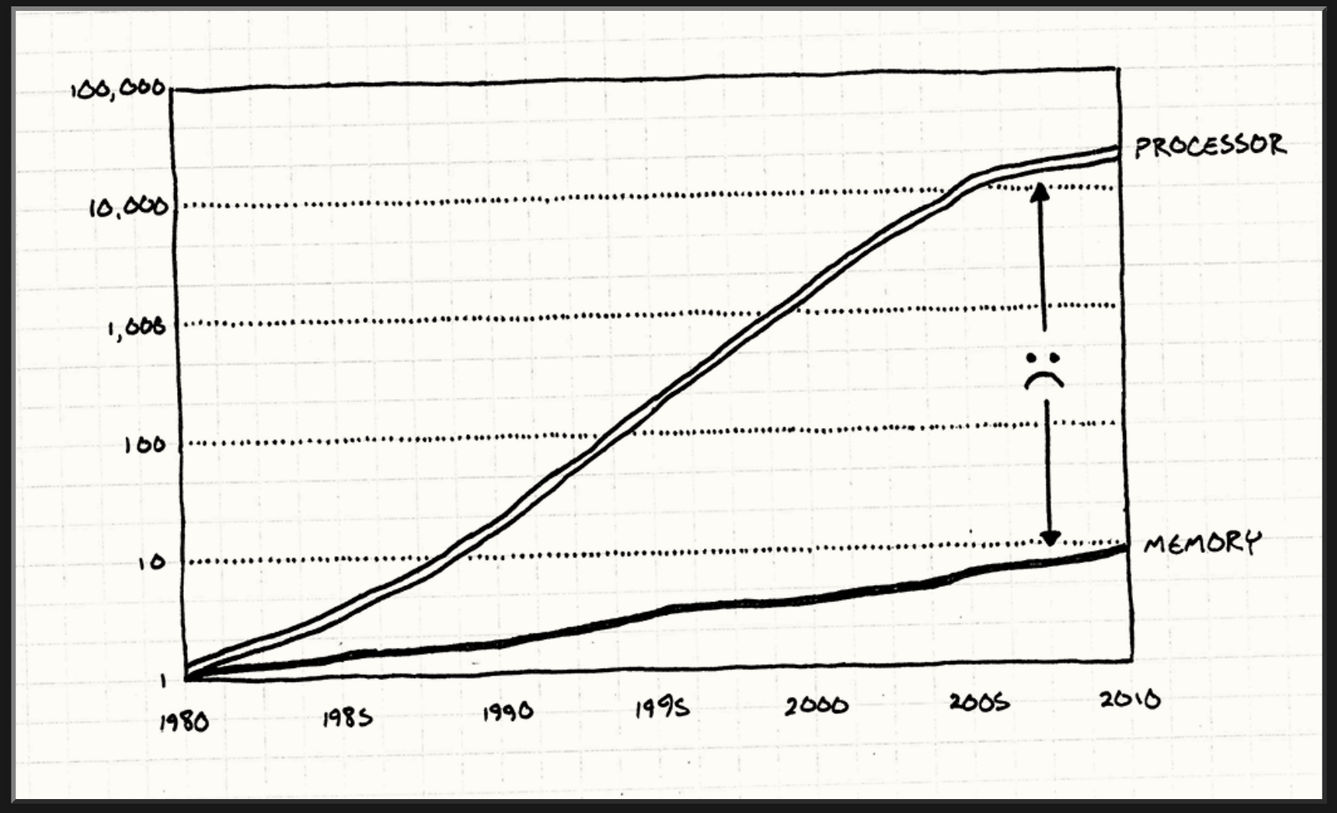
Throughput [B/s]: information transfer rate of a medium

Performance [Flops/s] (one def among others...): number of Floating point operations performed per unit of time

Moore’s law and parallelism

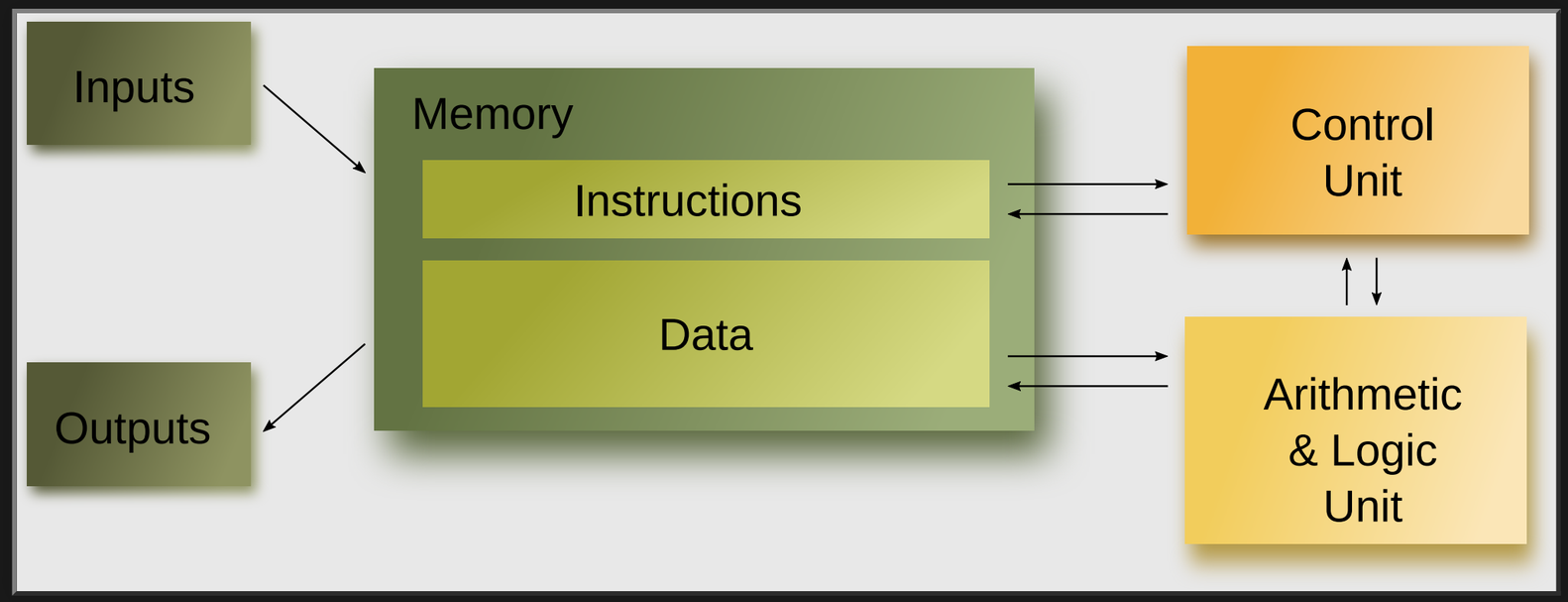


Memory wall

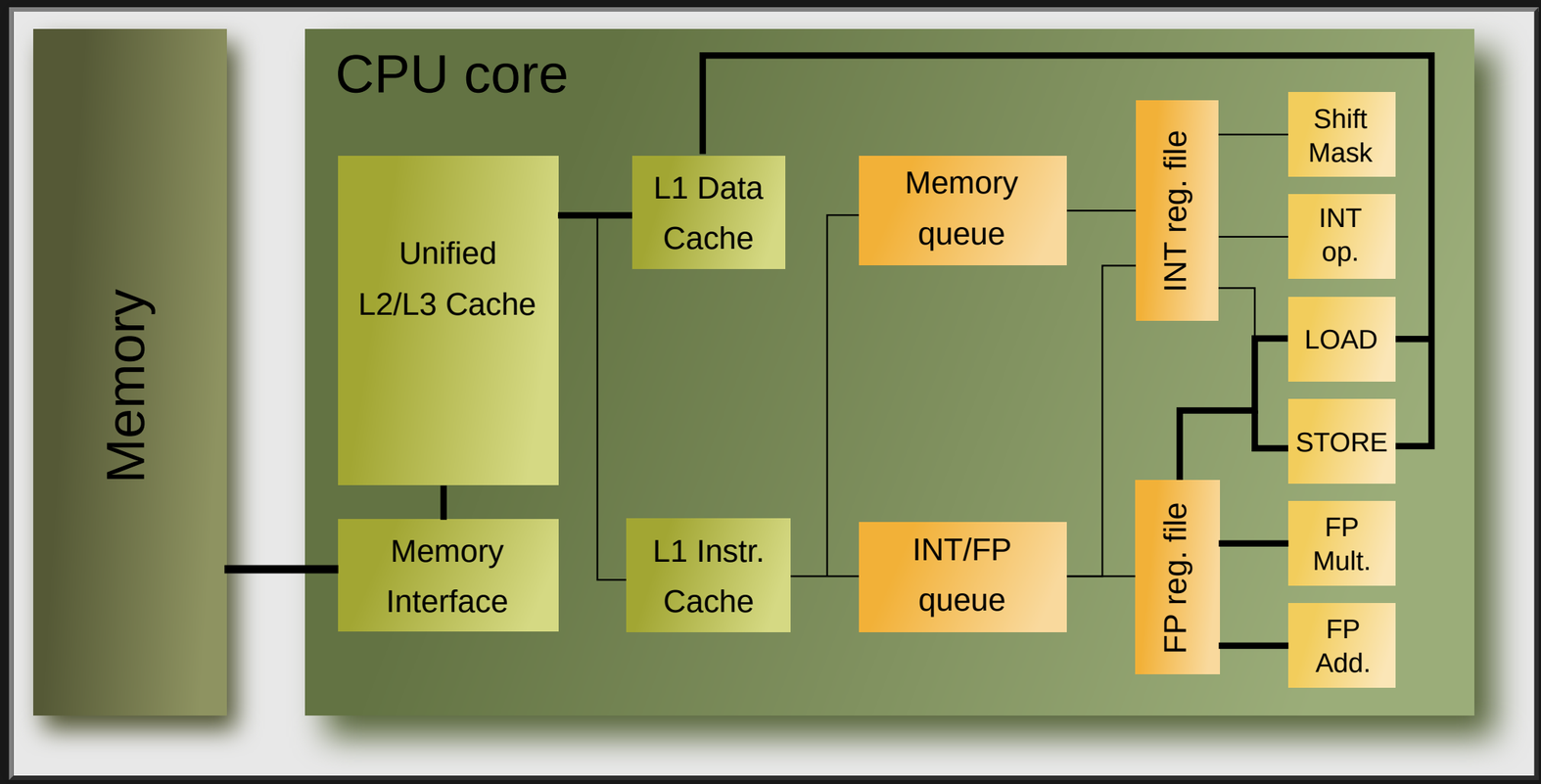


Courtesy to http://gameprogrammingpatterns.com/data-locality.html

Von neuman architecture



General architecture of a cache based processor

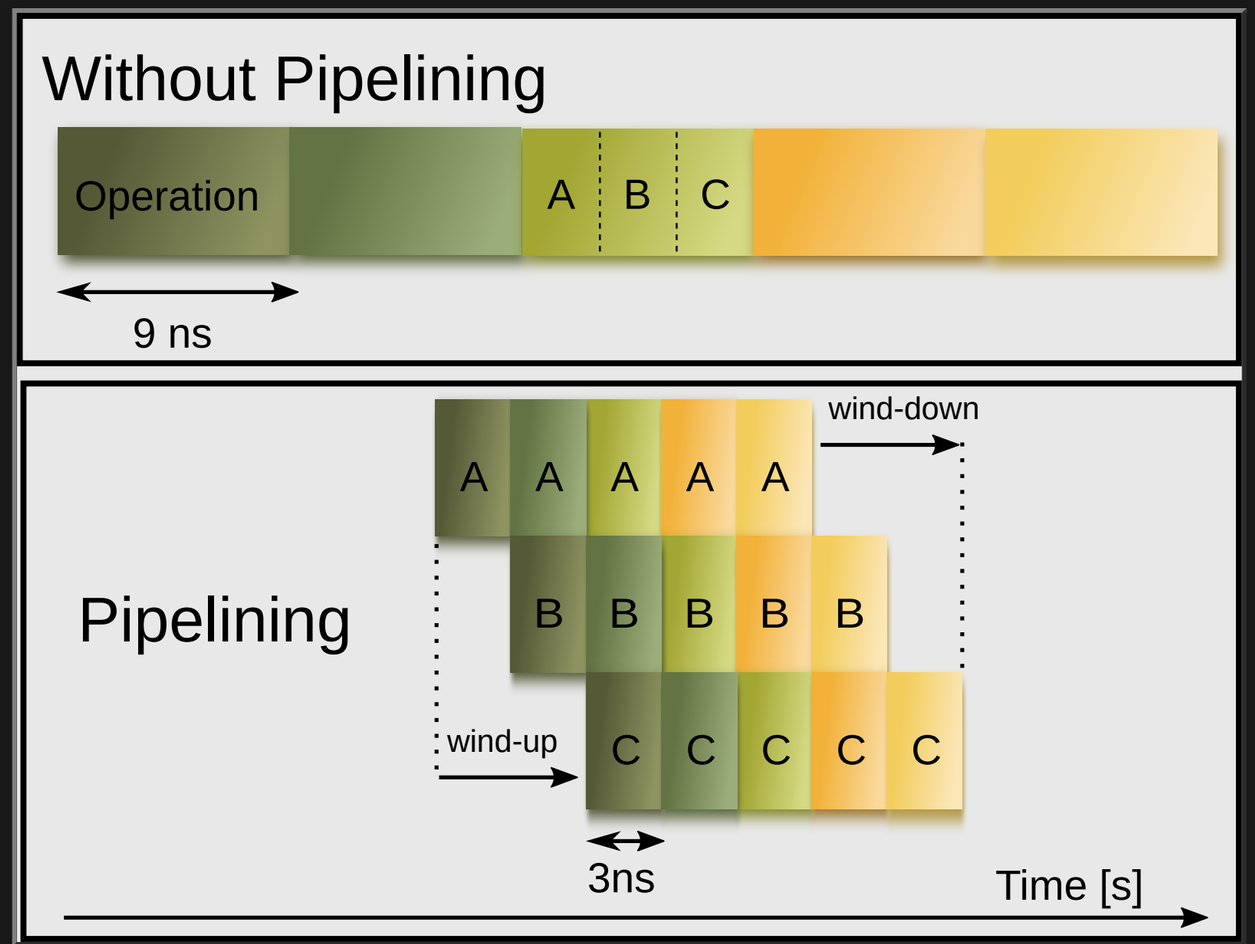


Most of the control unit, e.g. instruction fetch/decode, branch prediction,... not shown


7.2

CPU core

Pipelines


Without Pipelining
Operation
A
B
C
9 ns
Pipelining
wind-down
A A A A A
B B B B B
C C C C C
wind-up
3ns
Time [s]

Pipelines

- Splitting of complex operations in stages
- Stages executed concurrently
- Clock frequency increased
- **Operations troughput increase**

However

- Latency of operation is the same
- Bubbles in the pipeline or pipeline flush events reduce this throughput

Example of pipelines

- Instruction execution, 3 stages:
    - Fetch
    - Decode
    - Execute
- Floating point operations, Mult 5 stages:
    - Separate mantissa / exponent
    - Multiply mantissa
    - Add exponents
    - Normalize results
    - Insert sign

Superscalar processors

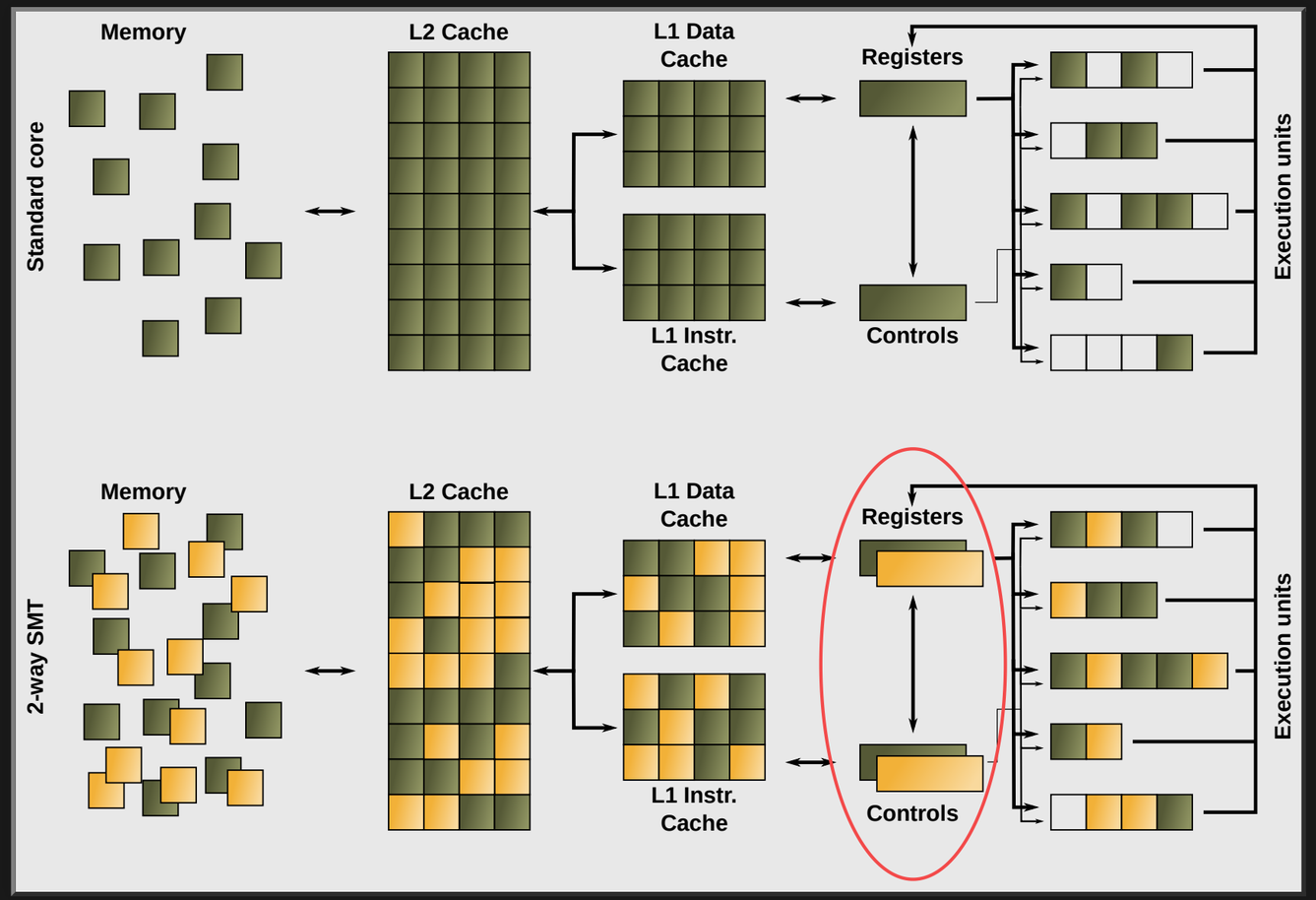
A bit of history

- Scalar processor vs vector processor: floating operation on a single scalar instead of a vector
- Single instruction executed at a time

Multiplying control units

- Multiple control units enable the use of Instruction Level Parallelism (ILP):
- Instruction stream is "parallelized" on the fly
- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors are 3- to 6-way superscalar & can perform 2 floating point instructions per cycles

Simultaneous multi-threading (SMT)



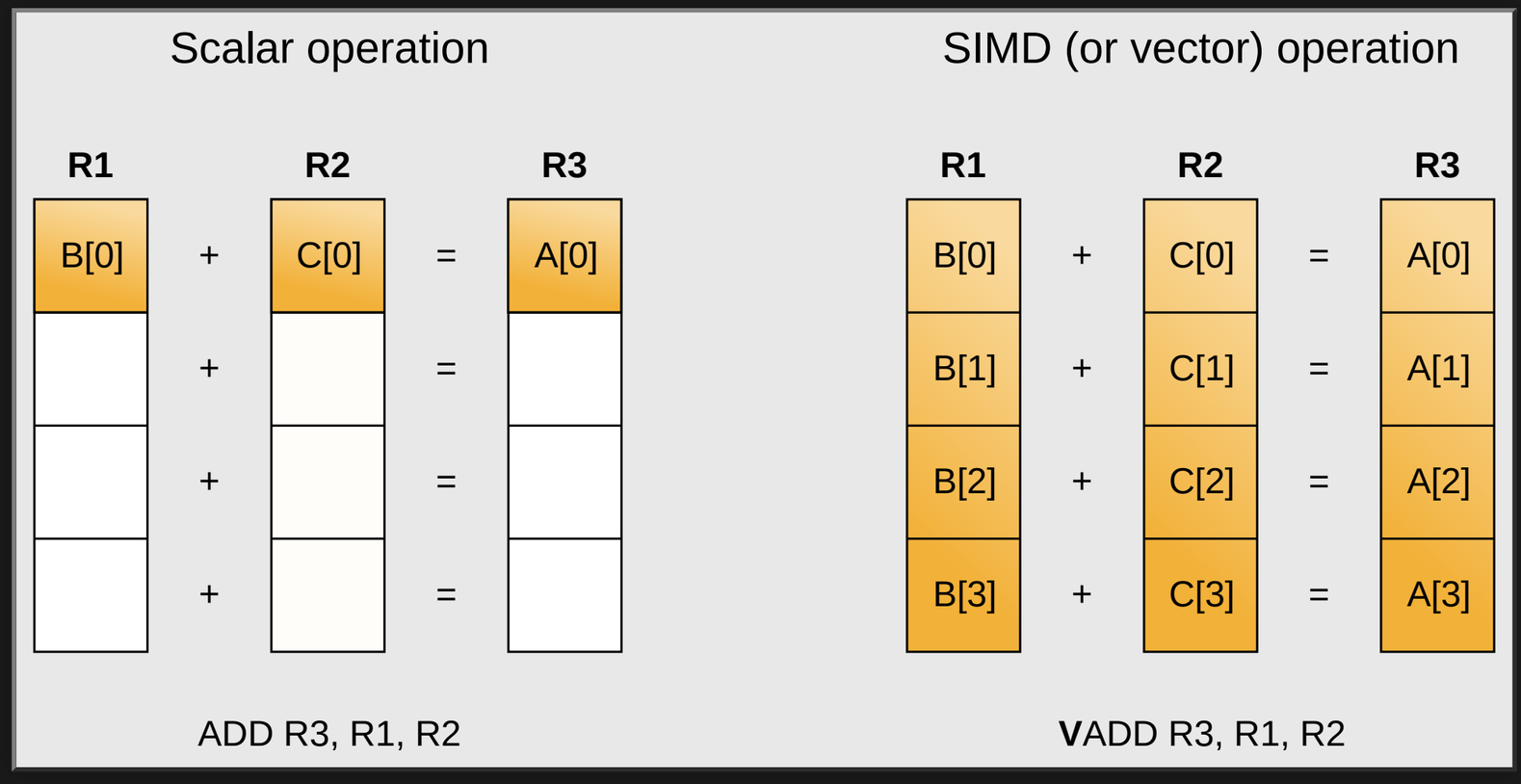
Simultaneous multi-threading (SMT)

- Control and register hardware replicated for each core
- Enables fast context change between threads running on the same core
- Potential **performance increase** (up to 25%) because pipelines better filled

But

- Potential performance decrease because cache size per thread reduced
- Performance potentially limited because twice the intra-node scalability is required

Single Instruction Multiple Data (SIMD)



Single Instruction Multiple Data (SIMD)

- Vectorisation: 1 instruction = N time the same operations on different data
- Increase performance because more operations per cycle
- Increase performance because less instructions for the same amount of operations
- Register size for AVX512: 512 bits $\rightarrow$ 8 values in Double Precision (or 16 SP)

Auto vectorisation by the compiler

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

$\hspace{3cm} \downarrow$ Code transformation by the compiler

```
for(int i=0; i < N; i+=4)
{
  a[i + 0] = b[i + 0] + c[i + 0];
  a[i + 1] = b[i + 1] + c[i + 1];
  a[i + 2] = b[i + 2] + c[i + 2];
  a[i + 3] = b[i + 3] + c[i + 3];
}
```

$\hspace{3cm} \downarrow$ Compiler

```
VLOAD r1, b[i]
VLOAD r2, c[i]
VADD  r3, r1, r2
VSTORE a[i], r3
```

Consequences on programming

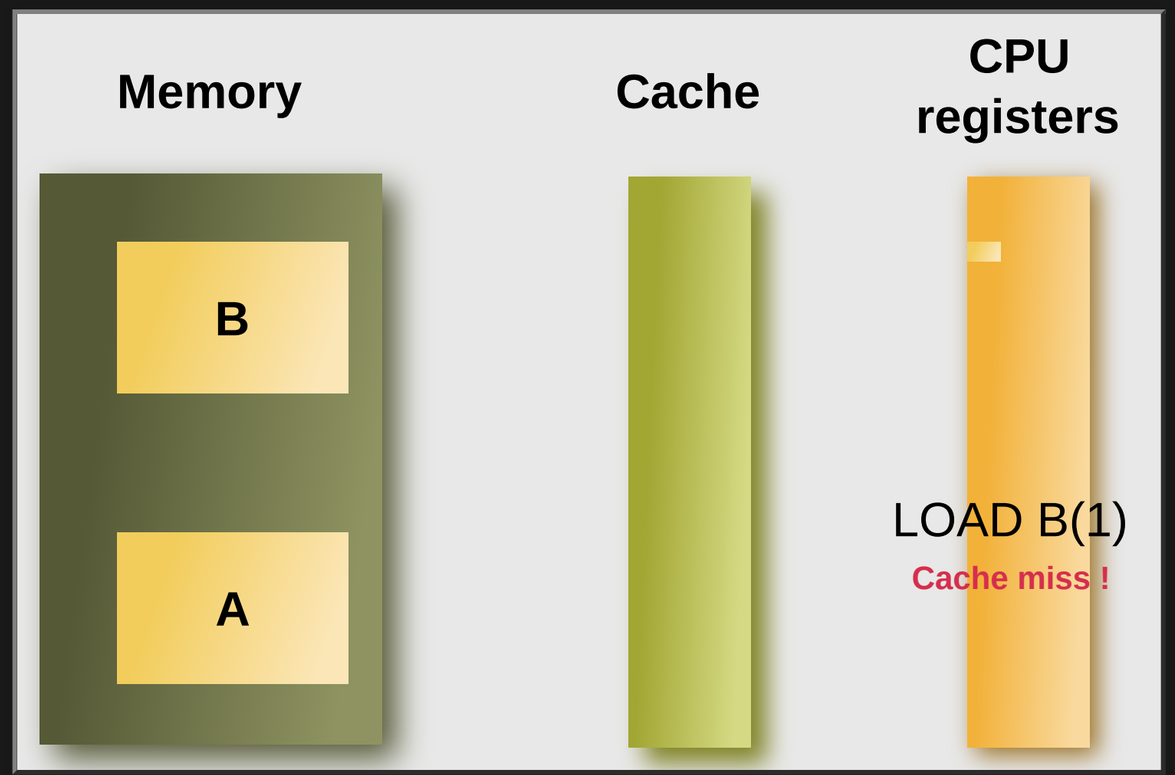
- Check out compiler optimisation reports !
- Avoid *if* statements in loop bodies
- Avoid data dependencies between loop iteration

CPU socket and node

Cache mechanism

Executing an array copy:

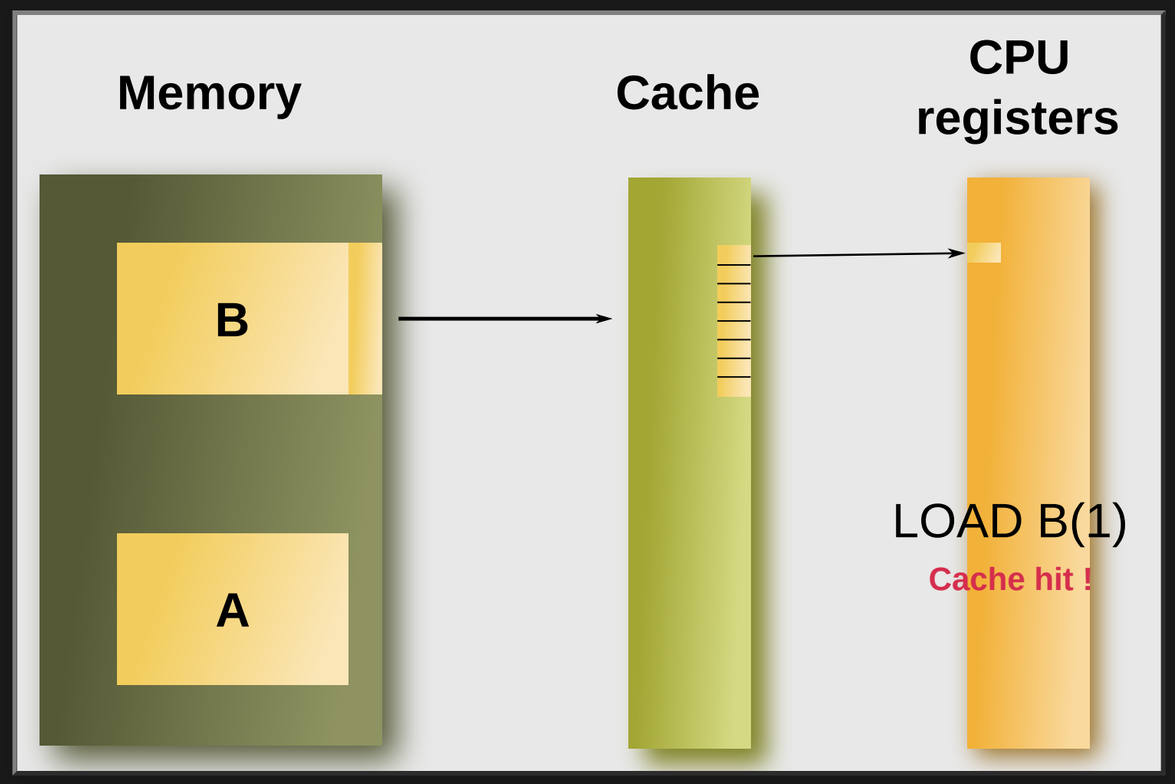
```
A(:) = B(:)
```



Cache mechanism

Executing an array copy:

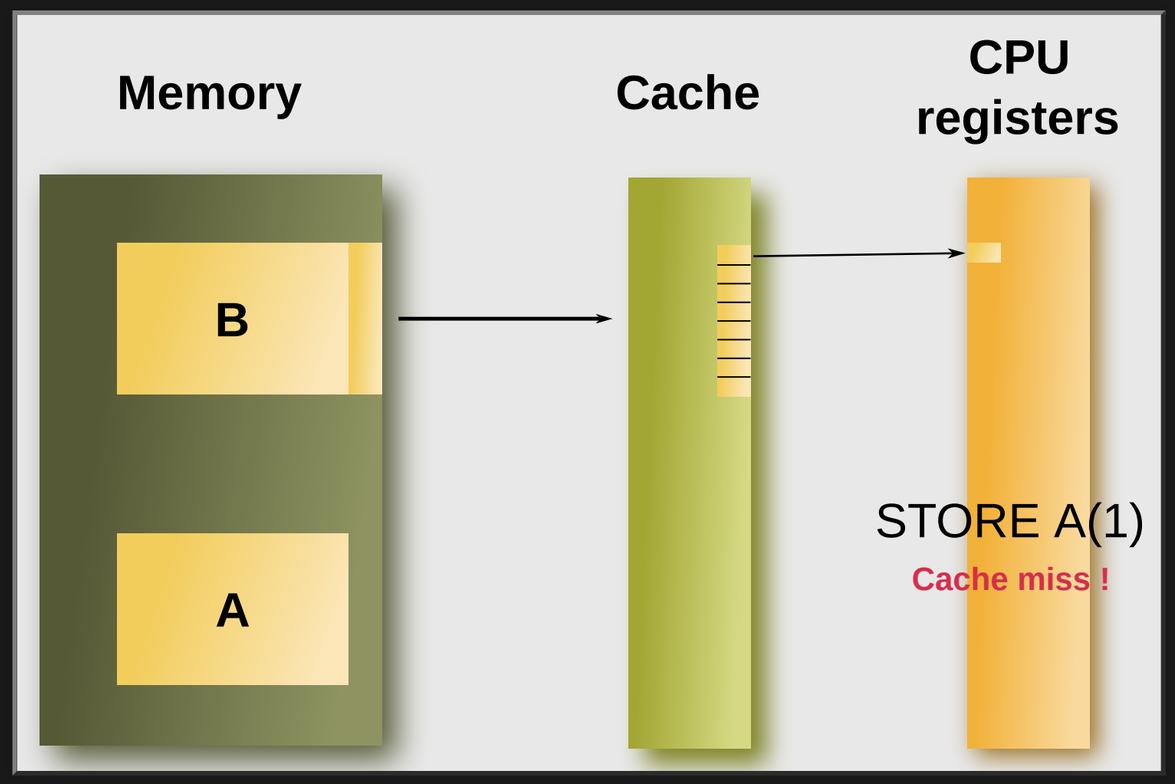
```
A(:) = B(:)
```



Cache mechanism

Executing an array copy:

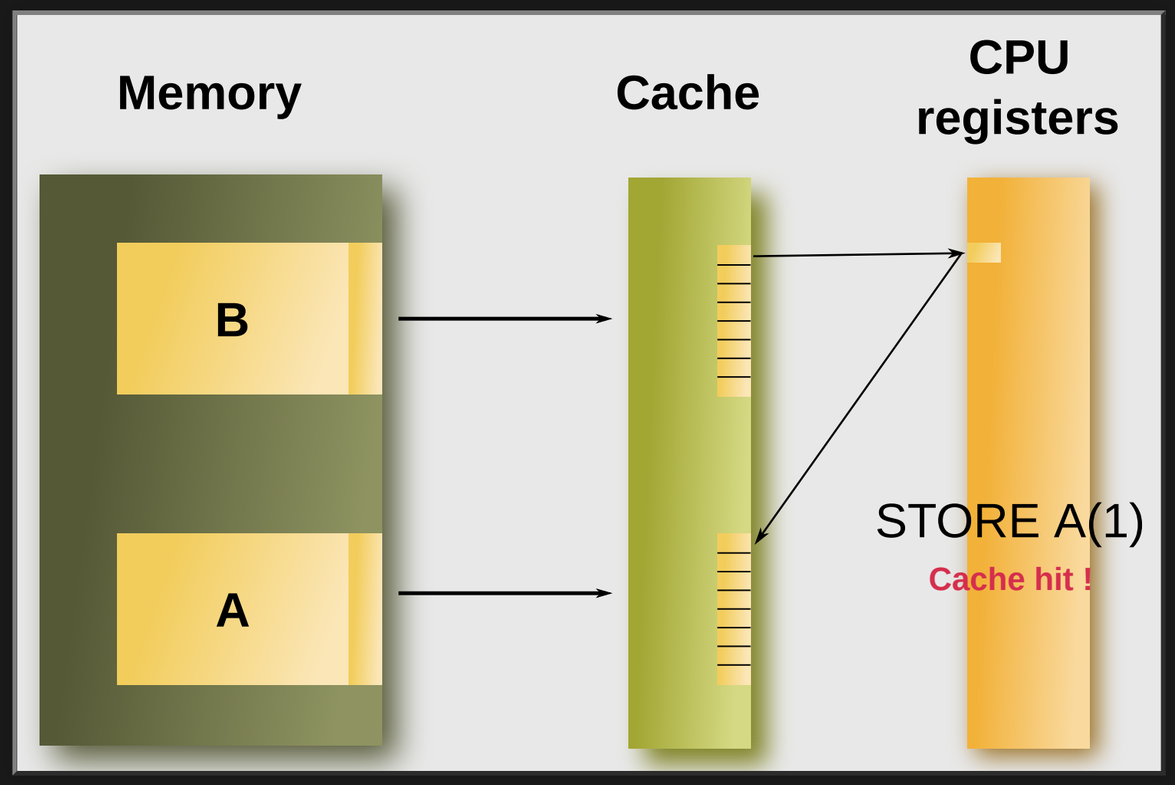
```
A(:) = B(:)
```



Cache mechanism

Executing an array copy:

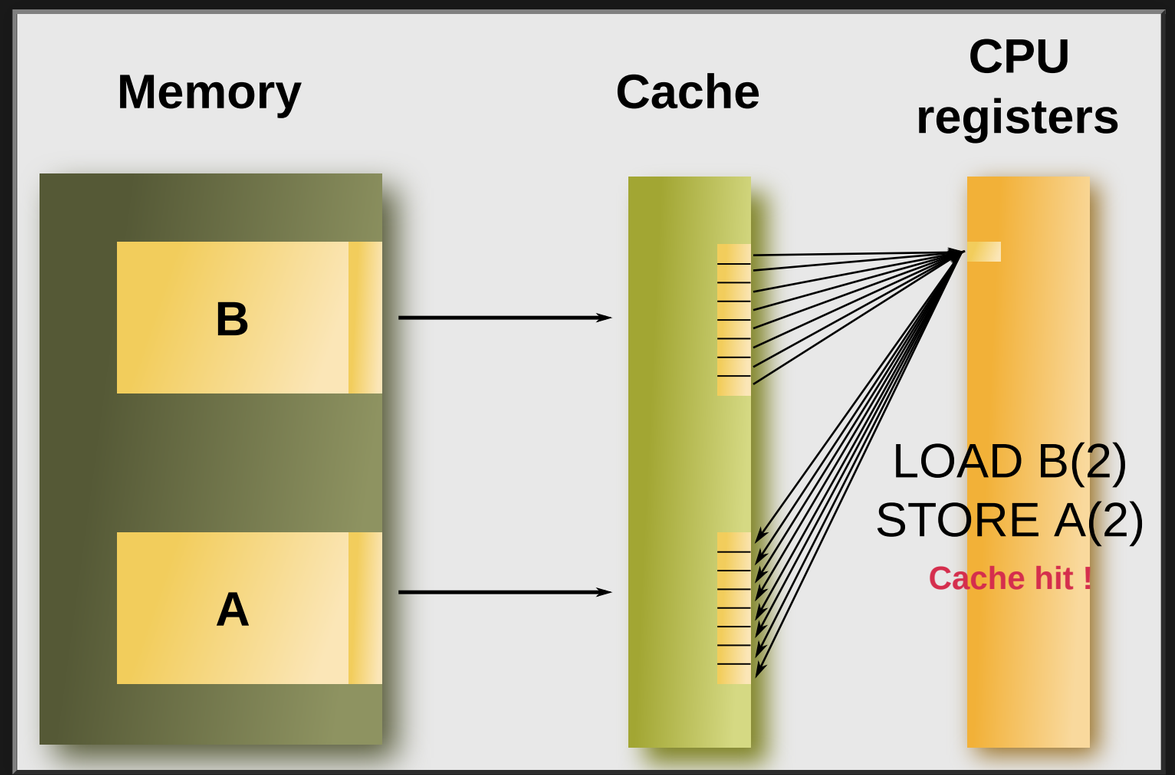
```
A(:) = B(:)
```



Cache mechanism

Executing an array copy:

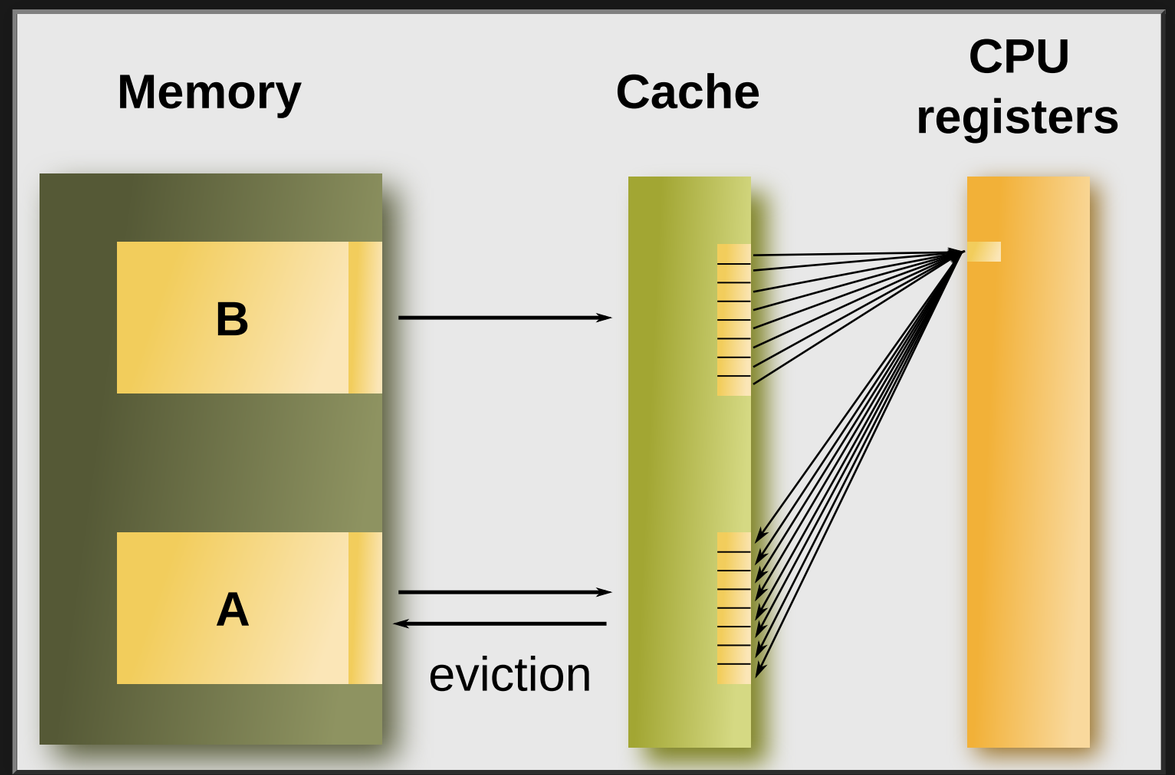
```
A(:) = B(:)
```



Cache mechanism

Executing an array copy:

```
A(:) = B(:)
```



Cache mechanism

- Cache introduced to hide memory latencies
- Cache line: atomic element transferred on the memory bus and between cache levels
- Size typically 64B:
- Cache lines movements are triggered by LOAD and STORE instructions
- Worth to be noted: 3 cache lines transferred on the memory bus to do the copy (Write-Allocate mechanism)

Consequences on programming

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

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

Consequences on programming

Good **spatial** locality

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

Bad **spatial** locality

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

Consequences on programming

```
for(int i=0; i < N; ++i)
{
  A[i] = scale * B[i];
  A[i] += g(i);
}
```

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

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

Consequences on programming

Good **temporal** (and spatial) locality

```
for(int i=0; i < N; ++i)
{
  A[i] = scale * B[i];
  A[i] += g(i);
}
```

Bad **temporal** (but good spatial) locality

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

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

However

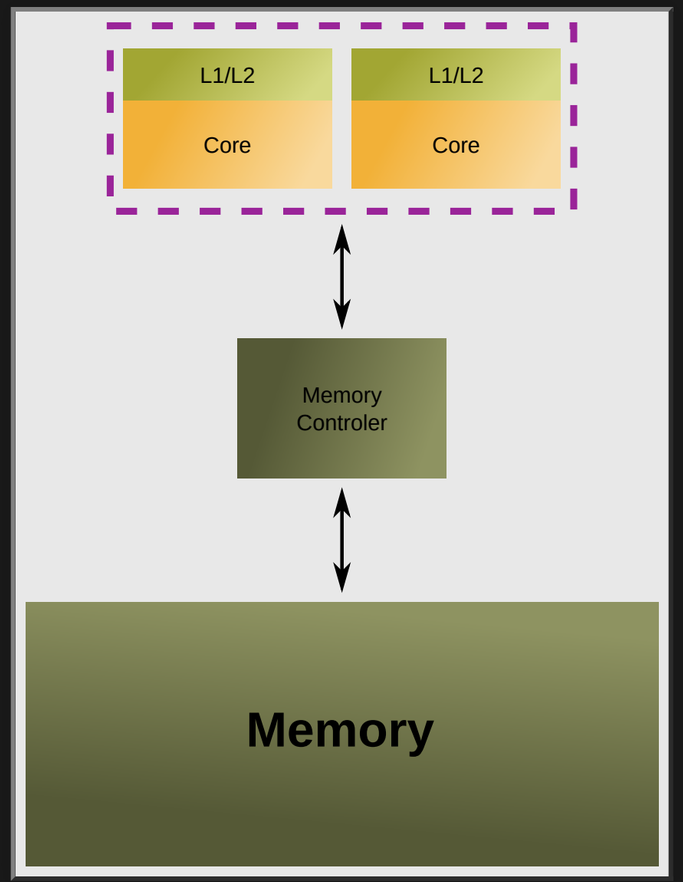
- Compiler able to merge the two loops of the second version
- Difficult to split the first version if loop body too large

UMA vs cc-NUMA

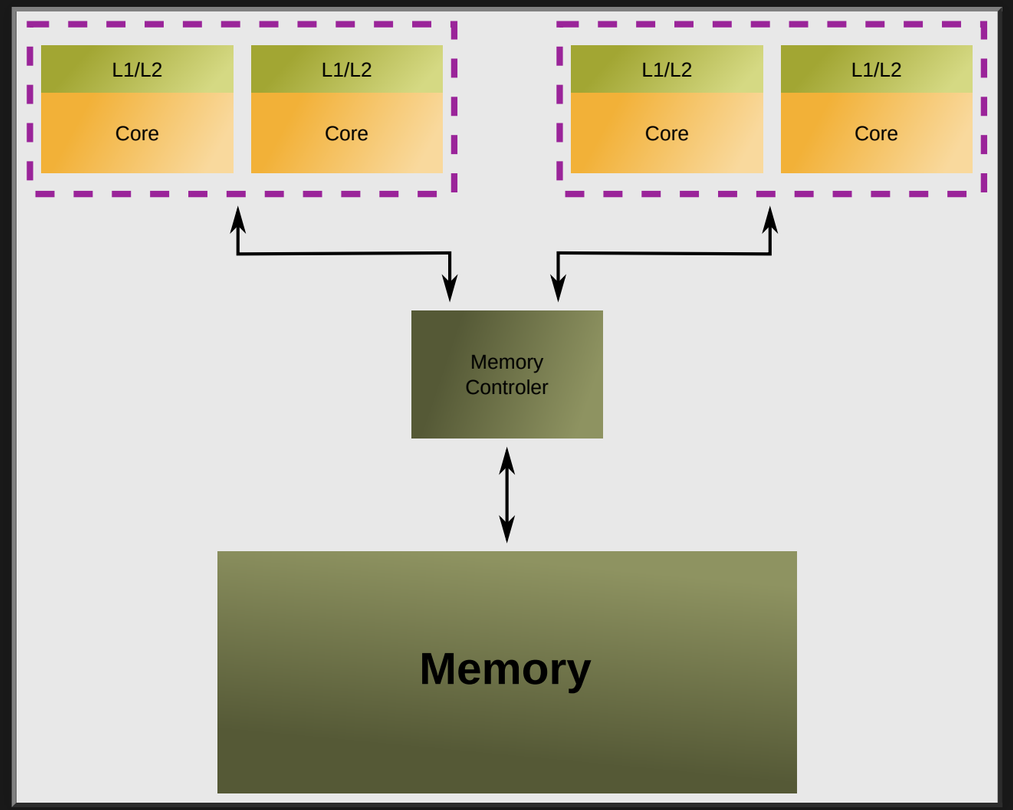
Shared memory systems requires coherent memory accesses between threads.

cc-NUMA is a consequence of hardware evolution to maintain coherent shared memory access at the hardware level.

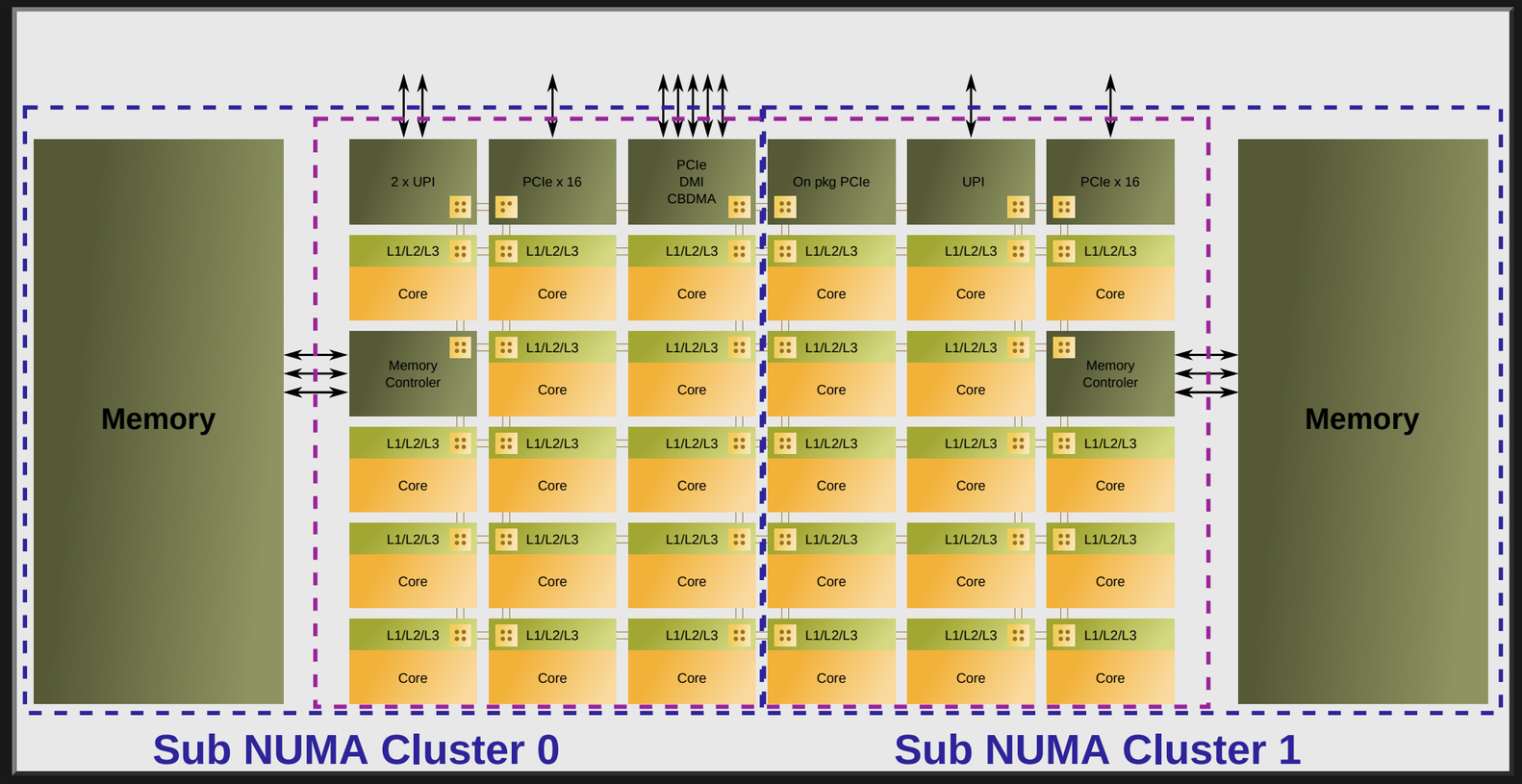
Intel Xeon Core2 (2006)



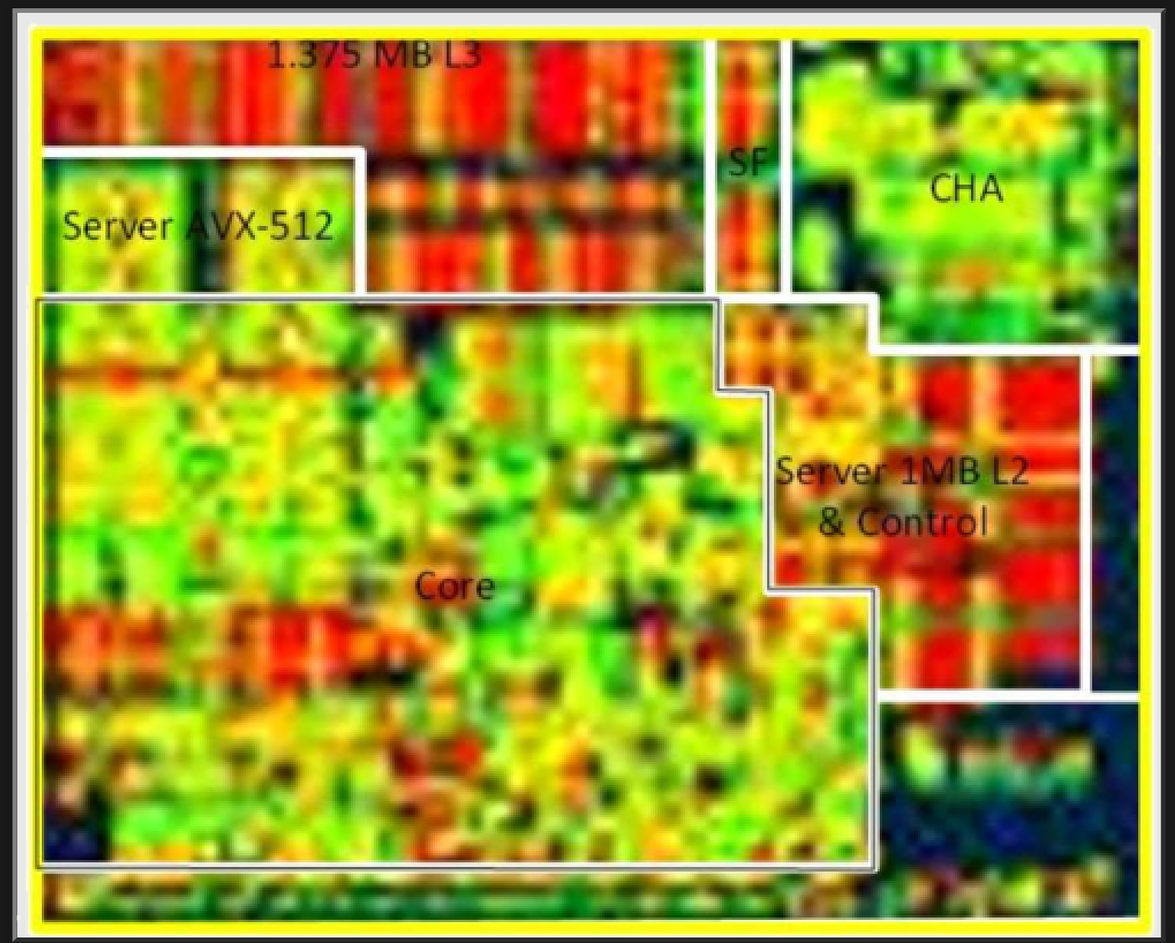
2x Intel Xeon Core2 (2006) UMA



Intel Xeon SkyLake Platinum (2017)

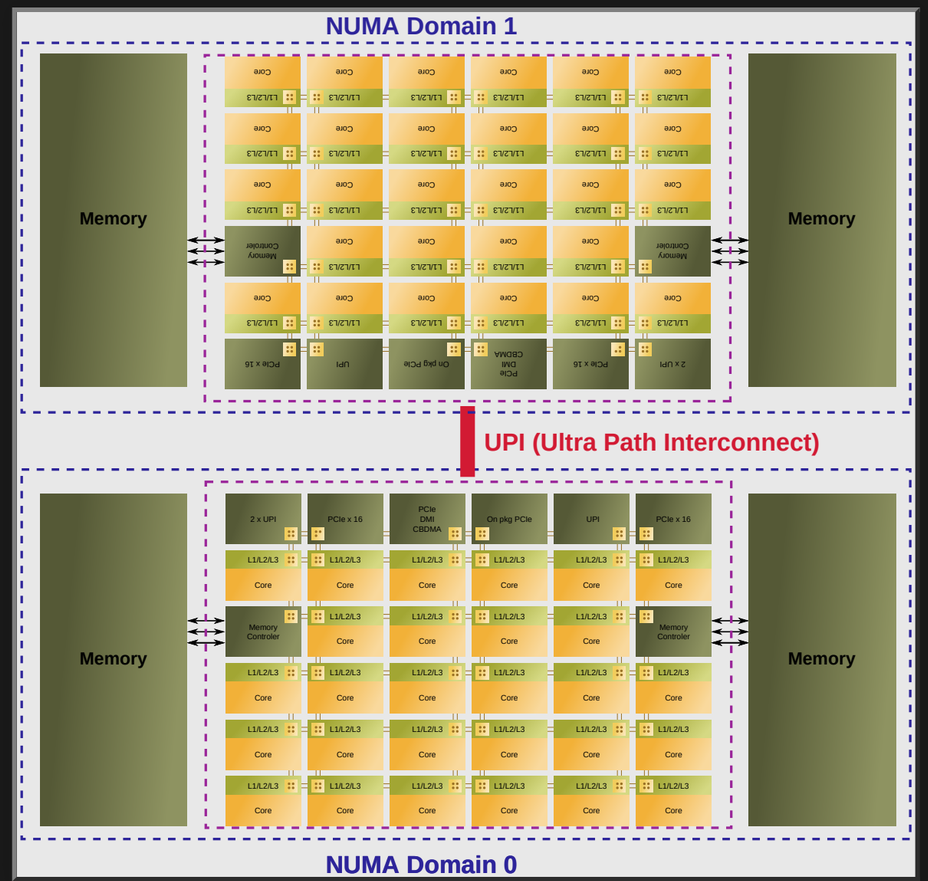


Intel Xeon SkyLake Core



Intel Xeon SkyLake full chip layout

2 x Intel Xeon SkyLake cc-NUMA



UMA vs cc-NUMA

UMA: Uniform Memory Access

- Memory latency and bandwidth similar for all threads whatever they access
- Coherency achieved thanks to a single memory controller

cc-NUMA: cache coherent Non Uniform Memory Access

- Memory latency and bandwidth different for all threads
- Coherency achieved thanks to a cache coherency mechanisms (several memory controllers)
- Depends on where they run on the processor and what they access
- Memory is organised in so-called NUMA domains
- NUMA parameters can be set with *numa-ctl* command line tool on linux

Consequences on programming

Thread pinning matters

```
export OMP_PROC_BIND=close/scatter/True
```

- Packs, scatters or simply binds threads on cores when a multi-thread application is launched
- Likwid, hwloc tools can give more control on thread pinning

Consequences on programming

Fisrt touch page policy

- Memory pages are allocated on the closest NUMA node of the core the thread is being executed on, at the time it touches the memory for the first time
- Shared arrays $\Rightarrow$ parallel initialisation

```
#pragma OMP PARALLEL FOR
for(int i=0; i < N; ++i)
{
  A[i] = 0.0;
  B[i] = 0.0;
}

read_data("input.txt", A, B);

#pragma OMP PARALLEL FOR
for(int i=0; i < N; ++i)
  A[i] = scale * B[i];
```

Roofline performance model

Three ingredients

- P_{max}: Applicable peak performance of the loop (Architecture)
- b_S: Applicable peak bandwidth of the slowest data path utilized (Architecture)
- I: Computational intensity, i.e. the "work" or Flops per byte transferred over the slowest data path utilized (your code)

The result

Expected performance P: $$P = min(P_{max}, I\times b_S)$$

Peak performance

For Intel Xeon Skylake platinum

- Frequency: $\nu = 2.7$ GHz
- Operations per instruction (2 x AVX512): $V = 2 \times 8$ (DP)
- Flop / operation (FMA: 1 Mul + 1 Add): $F = 2$
- Number of cores: $n_{core} = 24$
- Peak Performance: $P = \nu \times V \times F \times n_{core}$ Flop/s

$$P = 2072 ; GFlop/s$$

But for a serial code that does not vectorise

$$P = 10.8 ; GFlop/s$$

By the way, Top1 supercomputer in 1998 had 1100 GFlop/s...

Roofline performance model

```
A(:) = B(:) + C(:) * D(:)
```

- b_{S} = 100 GB/s
- I = 2 Flops / (4+1) Words = 0.4 F/W = 0.05 F/B
- $I \times b_S$ = 5.0 GF/s
- $P_{max} \simeq$ Peak Performance = 2072 GFlop/s
- $P = min(P_{max}, I\times b_S)$ = 5 GFlop/s

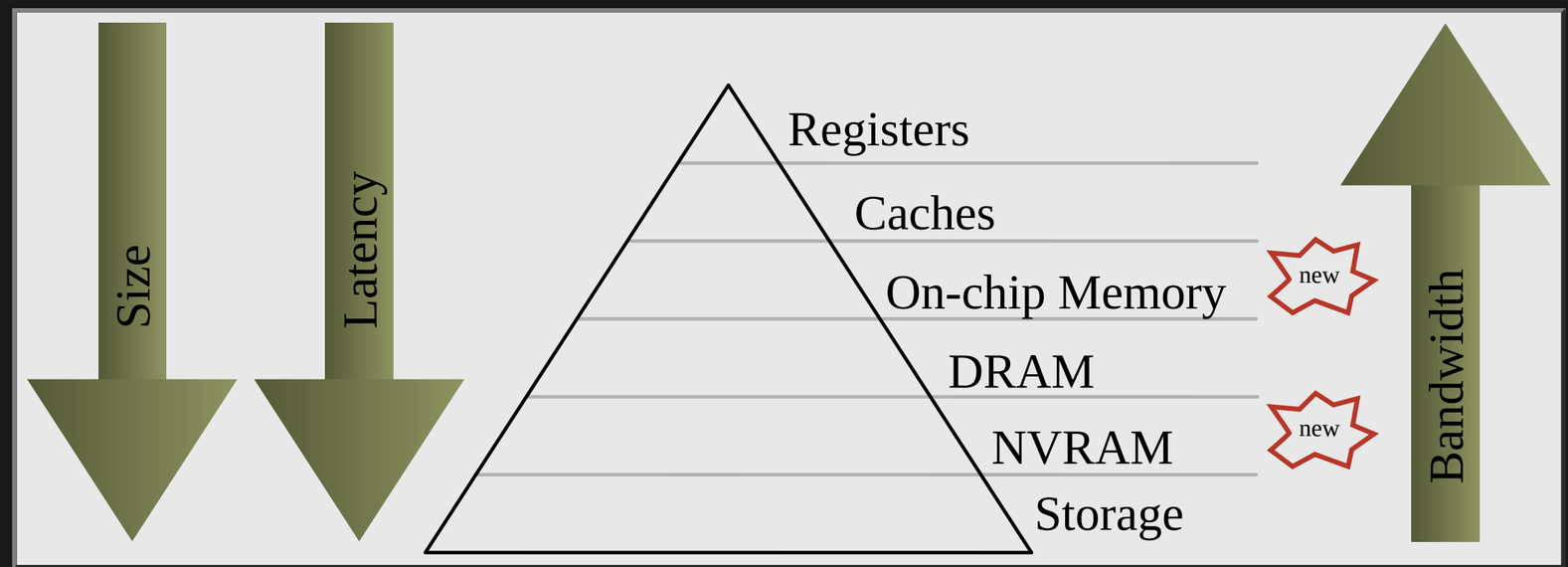
Kernel running at 0.25% of the peak performance

To keep in mind

Computing kernels are:

- **Memory bound**: $\Rightarrow$ Improve computational intensity (cache blocking, recompute data instead of storing it, ...)
- **Compute bound**: $\Rightarrow$ Improve core computing (vectorisation, SMT, ...)

Memory hierarchy



Memory hierarchy

- Two new levels of memory in the hierarchy
- Fully handled by the hardware ? Not clear…

CPU based cluster

Shared vs dist. memory systems

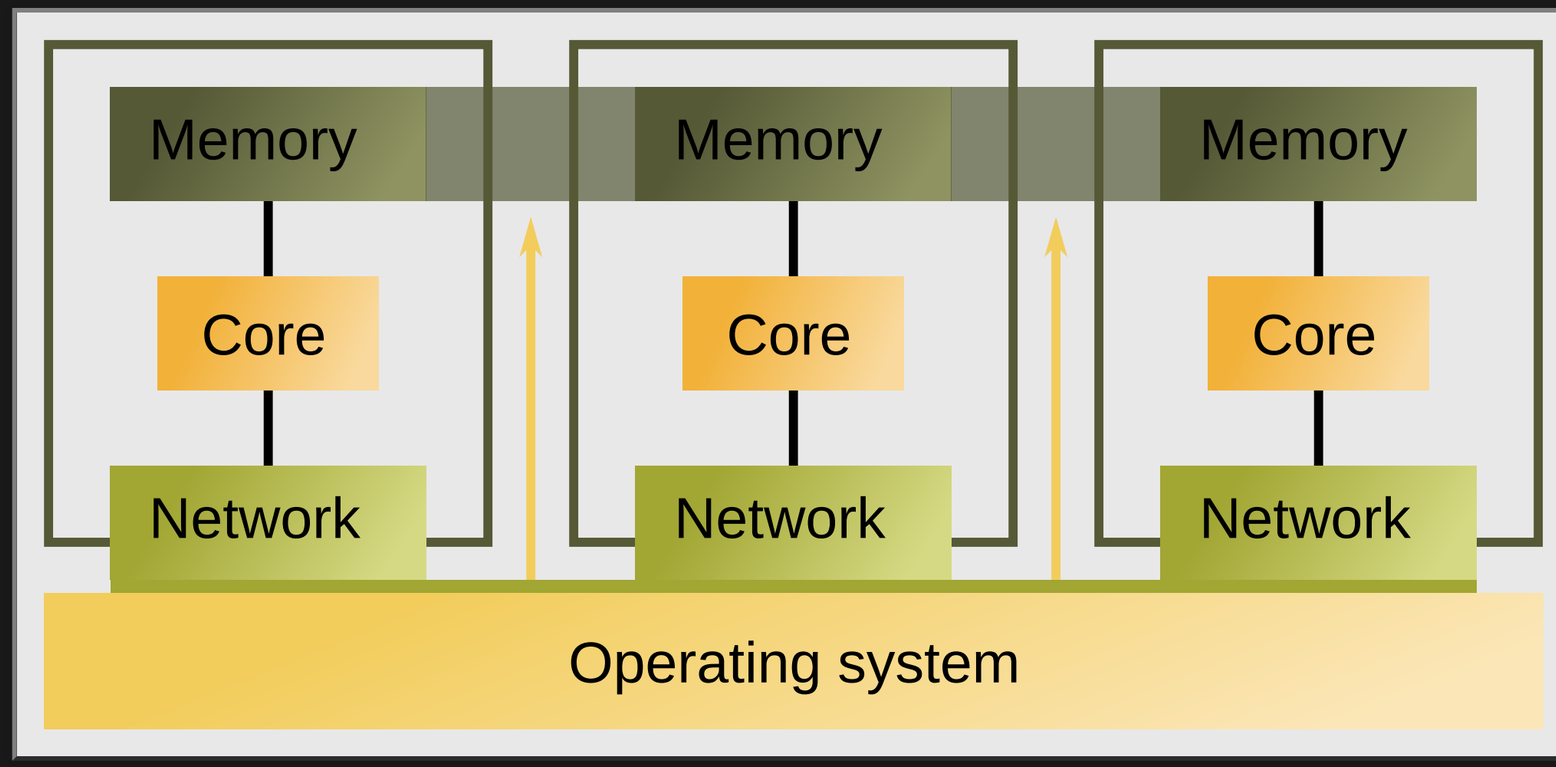
Shared memory systems: memory address space accessible by all threads running on the system

$\Rightarrow$ communication via shared arrays

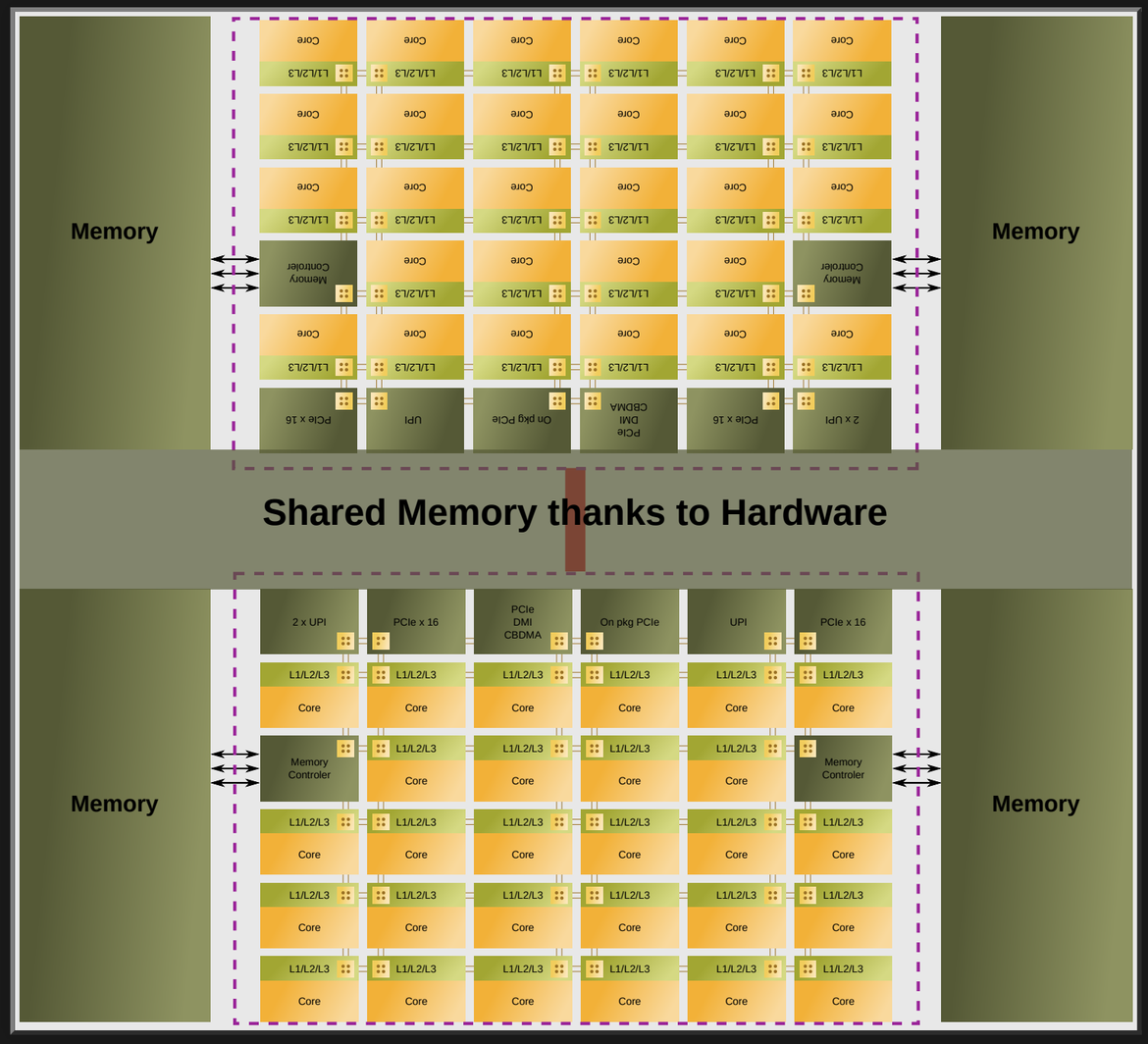
Distributed memory systems: memory address space split on different operating systems

$\Rightarrow$ communication via network messages

Shared memory systems (2002)

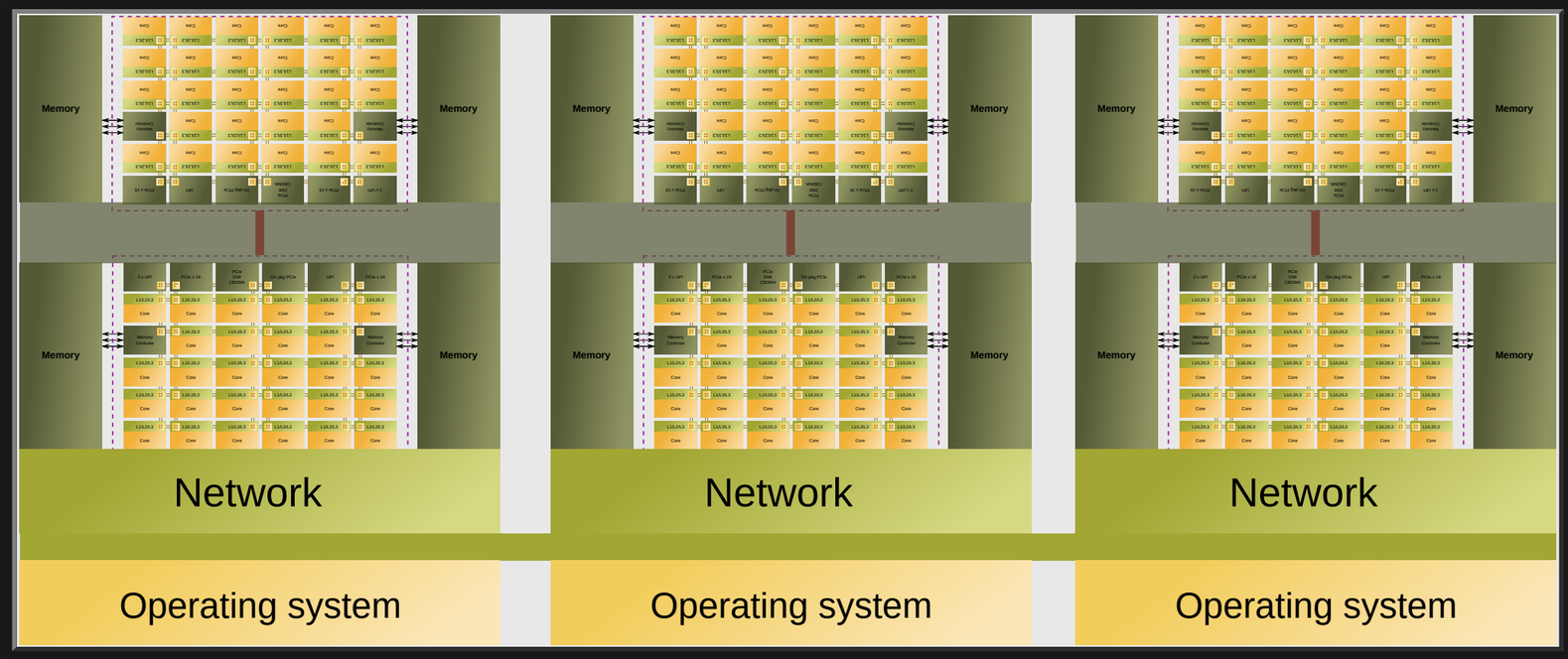


Intra-node shared memory (2017)




22.3

Distributed memory systems



Consequences on programming

- Threading model to benefit from shared memory (e.g. OpenMP)
- Network communication to benefit from both distributed and shared memory (e.g. MPI)

Network topology

Types of topology

- Non-blocking fat tree
- Pruned tree
- Torus

Consequences on programming

- Impacts machine vendors and MPI developers
- Few programming consequences, maybe some application deployment constraints

File systems

- Shared resources $\Rightarrow$ your job can be affected by another job running next to it
- Different file systems on a single HPC platform
    - Home: back-uped, low performance
    - Work: back-uped or not, medium-high performance
    - Scratch: no back-up, high performance
- ASCII vs binary output
- Serial vs parallel IO
- File system bandwidth reachable vs total bandwidth

File systems

- Total memory of irene supercomputer at CEA TGCC: 392 TB
- Total space of the scratch file system: 5 PB at 300GB/s
- Ratio: 78x the memory space, requires 21 min to perform a full memory dump

Impossible to export all raw data generated

Consequences on programming

- To know in advance the type of analysis to perform
- In-situ or in-transit post-processing starts to become mandatory
- Separating IO, post-processing and computation concerns with interfaces like PDI

Clouds vs HPC systems

Clouds:

- Solve 1000 independent problems of size 1
- Elasticity
- Redundancy

HPC systems:

- solve 1 problem of size 1000
- CPU performance
- Low latency networks required because tightly coupled problems

Scalability: definition

Answers the question: if N resources are used instead of 1, is the execution time t divided by N ?

Speedup $$S(N)=\frac{t(1)}{t(N)}$$

Relative efficiency $$E(N)=\frac{S(N)}{N} = \frac{t(1)}{Nt(N)}$$

- $S(N) \sim N$ or $E(N) \sim 100% \Rightarrow$ Application scales
- $S(N) < N/2$ or $E(N) < 50% \Rightarrow$ Application does not scale

Scalability: Amdhal's law

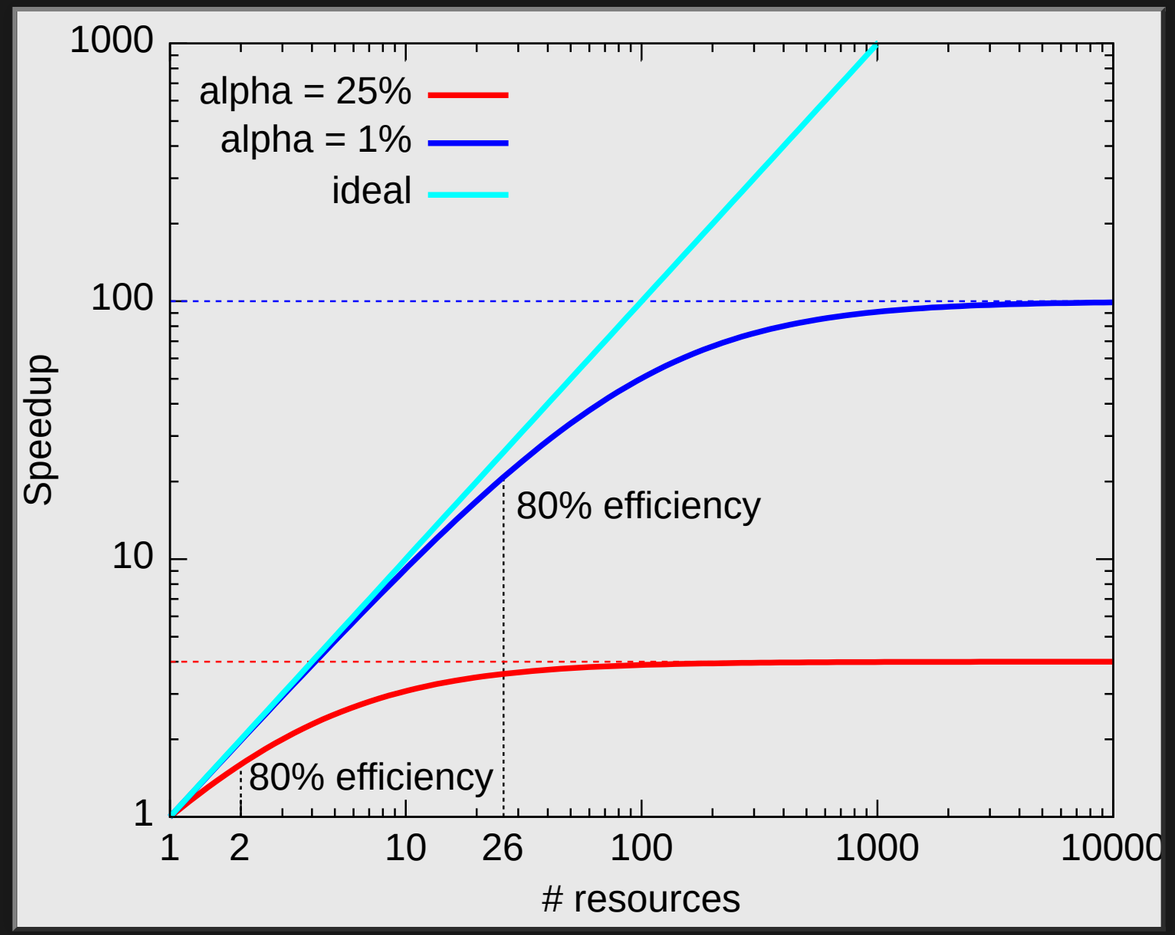
Serial α_s and parallel α_p fractions of the **source code**
$$t(1)= (\alpha_s + \alpha_p)t(1)$$

Assuming a **perfect scaling** of the parallel fraction $$t(N) = (\alpha_s +\alpha_p/N)t(1)$$

The speedup reads $$S(N)=\frac{t(1)}{t(N)}=\frac{1}{\alpha_s + \alpha_p/N}$$

Assuming a **perfect and unlimited parallel computer**
$$\lim_{N\rightarrow \infty} S(N)=\frac{1}{\alpha_s}$$

Scalability: Amdhal's law



Scalability: Strong vs Weak scaling

Strong scaling

- same global problem size when resources $\nearrow$
- problem size per resource $\searrow$ when resources $\nearrow$

Weak scaling

- global problem size $\nearrow$ with resources
- same problem size per resource when resources $\nearrow$

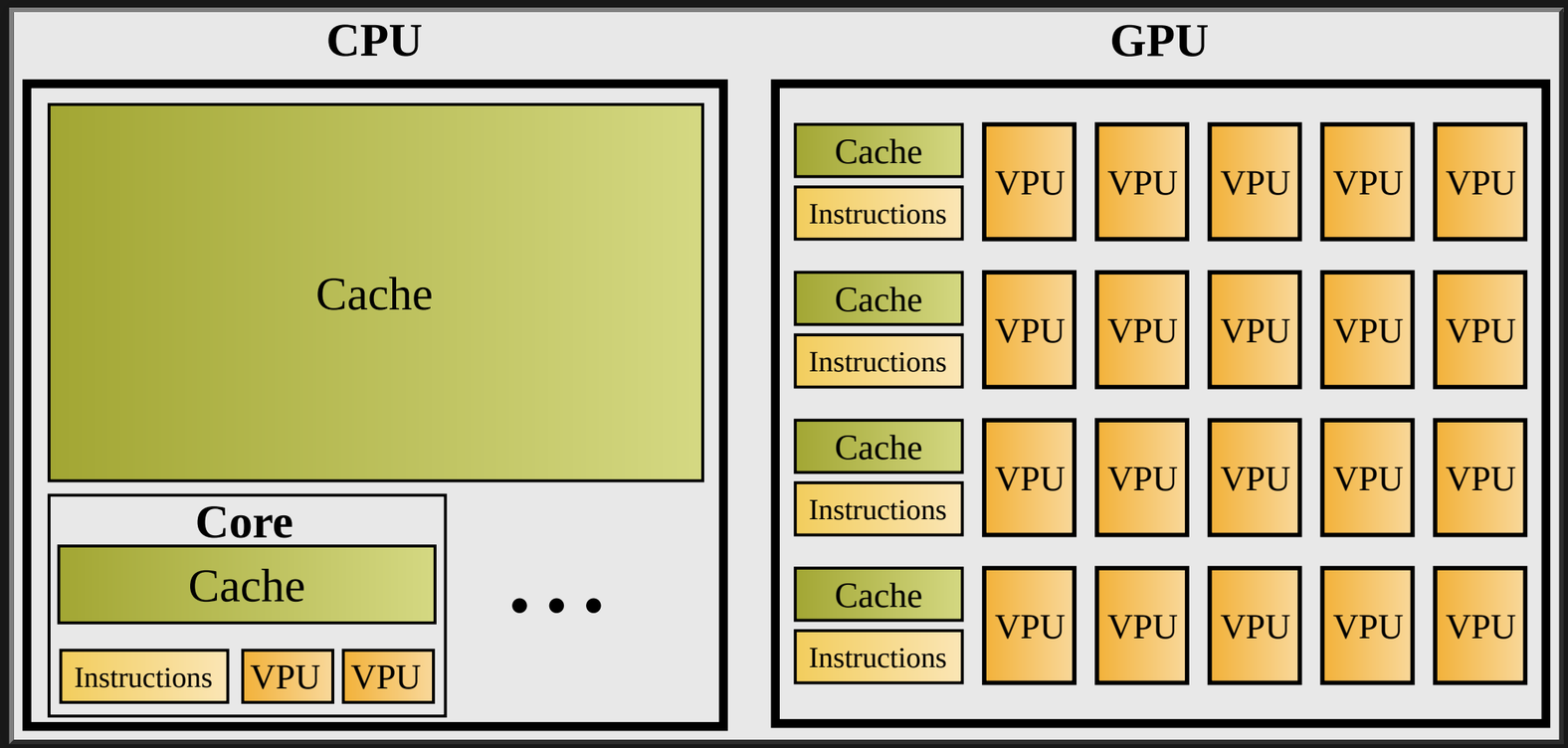
Consequences on programming

- Load imbalance
- Parallelization overhead
- Number and size of communications
- Data synchronisation

Scalability: presentation

- Present **performance oriented metrics** rather than speedups
  - GFlop/s
  - Simulated time / seconds of simulation time
  - Number of convergence iterations / seconds of simulation time
- Enable to exhibit single core or single node optimization
- Separate intra-node from inter-node scalability

GPUs compared to CPU



GPUs require massive and finer grain parallelism

GPUs compared to CPU

CPUs designed for high serial performance

- focus on capturing temporal locality in a single thread
- sophisticated (i.e. large chip area) control logic for instruction-level parallelism (branch prediction, out-of-order instruction, etc...)
- CPU have large cache memory to reduce the instruction and data access latency

GPUs compared to CPU

GPUs designed to maximize the throughput of **ALL** threads

- High latency tolerance
- Focus on capturing spatial locality, shared data among multiple threads
- # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
- Multithreading can hide latency $\Rightarrow$ no need for big caches
- Share control logic across many threads

GPUs compared to CPU

- GPUs are accelerators (data transfers)
- Low level programming: cuda
- High level programming: OpenACC, OpenMP

You said FPGA ?

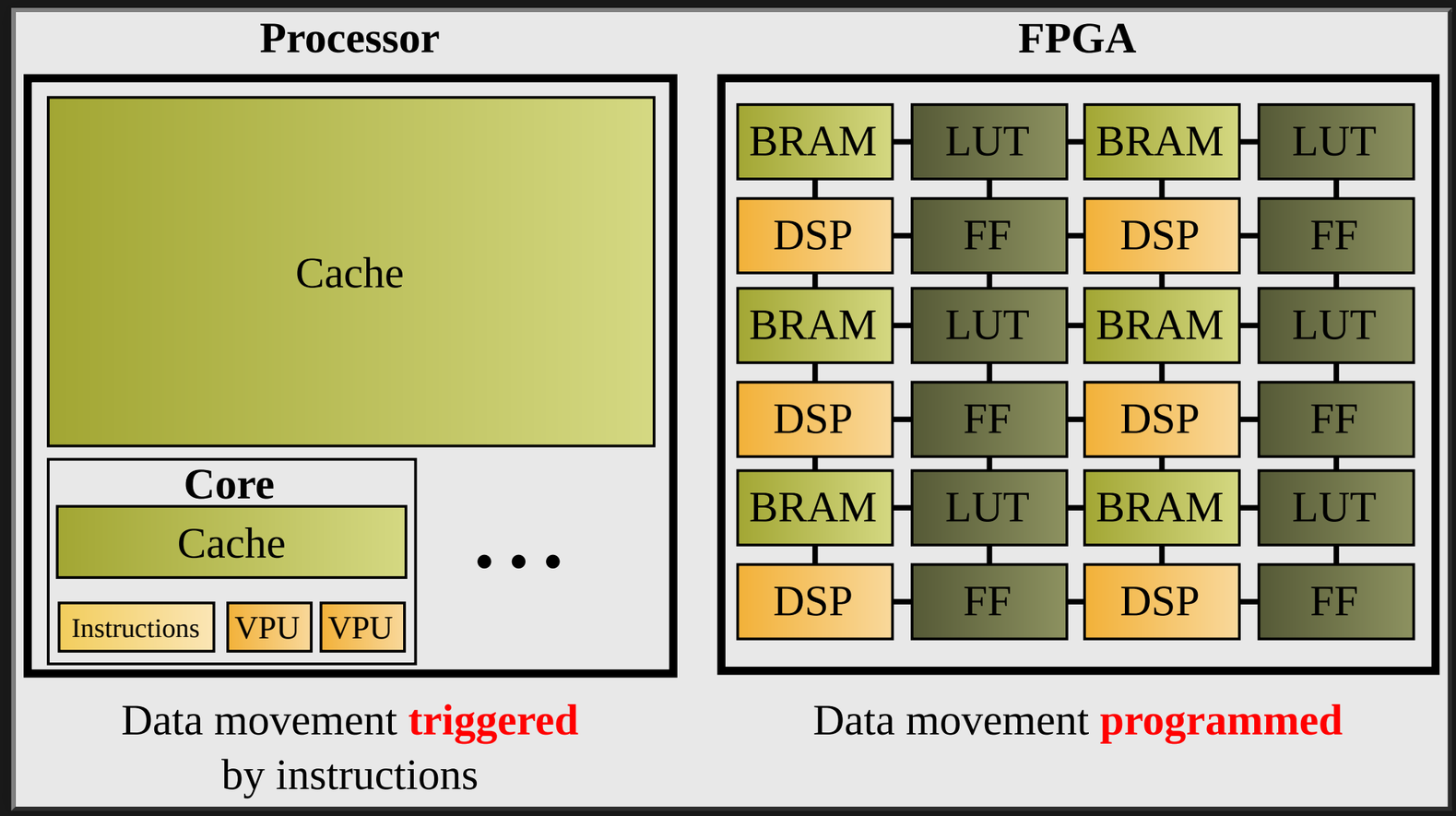
Some acronyms

- FPGA: Field Programmable Gate Array
- LUT: Look Up Table (boolean logic function)
- FF: Flip-Flop (circuit to store one bit of information)
- BRAM: 4KB blocks of RAM
- DSP: Digital Signal Processing (versatile arithmetic unit)

But what is it ?

- Reconfigurable logic
- Algorithm "hard wired" in the silicon
- Computations offloaded as for a GPU accelerator

An FPGA is NOT a processor



How to design a circuit for FPGA ?

Available languages

- VHDL, the standard: very low level for electronic people
- XilinX HLS: Pragmas for C
- CycL or Intel OneAPI ??: C++ framework
- Maxeler MaxJ: DSL based on Java

Challenges

- Algorithm reformulation for a streaming implementation
- Reductions are your enemies
- Significant space on silicon can be saved with smaller precision arithmetic

Architecture comparison

Chip	2x Intel SkyLake	Intel KNL	NVidia Pascal	Xilinx XCVU9P
Techno.	14nm	14nm	16nm	16nm
Power	410W	215W	300W	**< 50W**
Freq.	2.7GHz	1.4GHz	1.5GHz	0.1-0.5GHz
#cores	48	72	N.A.	N.A.
cache	2x57MiB	34 MiB	18 MiB	62 MiB
On Chip Memory	0	16GB	16GB	0
DRAM	128-768 GB	384 GB	0	48GB
Peak perf. (DB)	4 TF/s	3 TF/s	5.3 TF/s	**0.5 TF/s**

Some orders of magnitude

Data transfer

Device	Latency	Throughput
NVLink 2.0	5 μs	400 Gb/s
PCIe Gen3 x 8	5 μs	50 Gb/s
SATA 3.0	20 μs	5 Gb/s
USB 3.0	30 μs	10 Gb/s
Gb Ethernet	20 μs	1 Gb/s
10Gb Ethernet	20 μs	10 Gb/s
Infiniband	2 μs	200 Gb/s

Some orders of magnitude

Data storage

Device	Latency	Throughput
Register	1 cy	N.A
L1 Cache	4 cy	N.A
L2 Cache	10-20 cy	N.A
L3 Cache	30-60 cy	N.A
HBM (Pascal)	100 ns	700 GB/s
MCDRAM (KNL)	100 ns	500 GB/s
DRAM DDR4	100 ns	100 GB/s

Some orders of magnitude

Data storage (cont.)

Device	Latency	Throughput
NVDIMM	3 μs	5GB/s ?
NVe NVRAM	10 μs	5GB/s ?
SSD	100 μs	500 MB/s
HDD	5 ms	50 MB/s

Architecture trends

- GPU share in HPC still growing
- Processors integrating reconfigurable logic (Intel Xeon-Arria)
- FPGA integrating specialised blocks (XilinX Versal ACAP)
- Convergence ?

Join us in September !

- Action Nationale de Formation PerfEvalHPC
- Du 16 au 20 Septembre 2019 à l'Observatoire de Haute Provence
- Deux outils :
  - Scalasca
  - Paraver
- **Hébergement et repas pris en charge**

Venez avec votre code et repartez avec une feuille de route d'optimisation !

Il reste encore des places, inscrivez vous vite.