PARALLEL PROCESSING

…A RECAP

Modern computing for physics

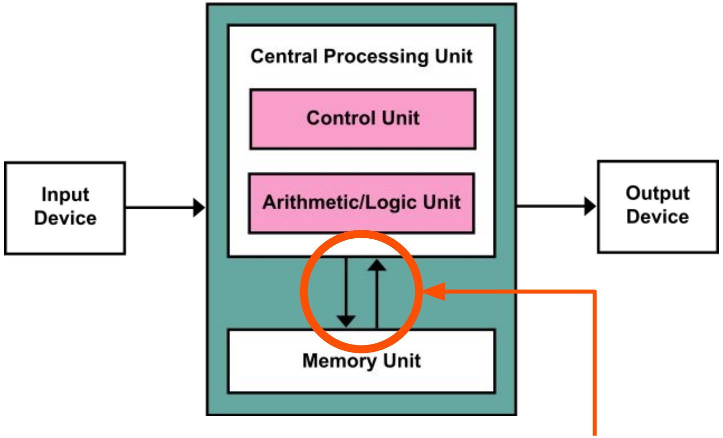
J.Pazzini
Padova University

Physics of Data
AA 2024-2025

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

A brief history of computing processing

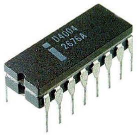
- 1940s: John Von Neumann proposed the *stored*-program computer model.
  This model proposed a computer as composed by three separate components:
  - **Memory**: for storing data and instructions
  - **Central Processing Unit (CPU)**: for decoding/executing instructions
  - **Inputs/Outputs (I/O)**: set of inputs and output interfaces (including storage devices)
- 1950s: advent of semiconductor-based transistor technology
- 1960s: transistors are used in the production of computers; invention of the integrated circuit (IC)
- 1970s: microprocessors were introduced

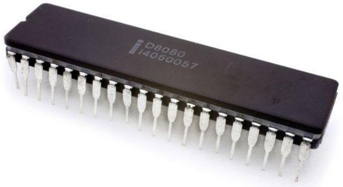


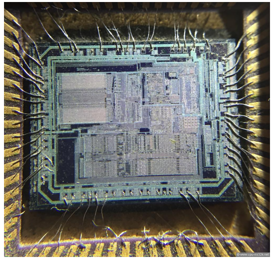
CPU ↔ Memory

- data
- instructions

Since then, **Modern CPUs** are microprocessors contained on a single integrated circuit chip.

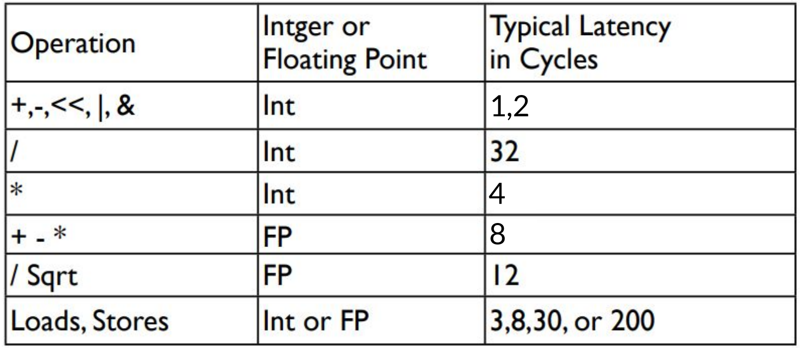




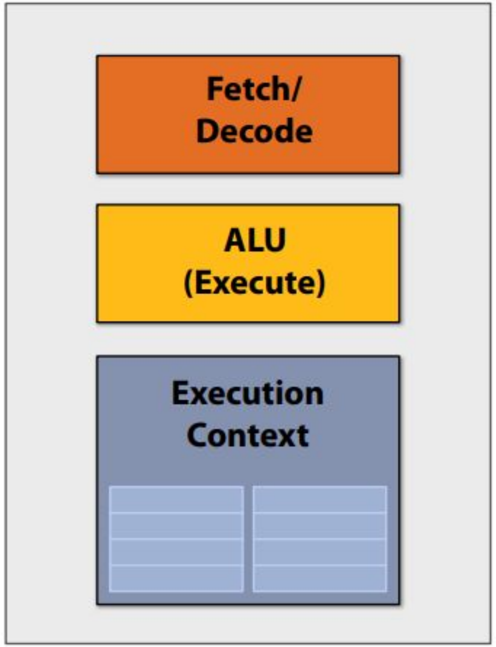


Smarter CPUs

- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
  - Development of dedicated ISAs (Instruction Set Architectures)
    - Reduced vs Complex Instruction (RISC vs CISC)
  - Instruction pipelining (Instruction Level Parallelism)
  - OoO (Out of Order) execution
  - Branch prediction

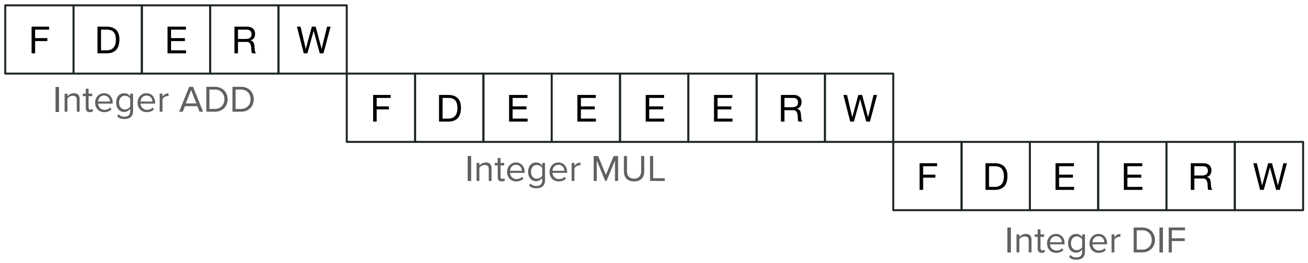


Operation	Intger or Floating Point	Typical Latency in Cycles
+,-,<<, |, &	Int	1,2
/	Int	32
*	Int	4
+ - *	FP	8
/ Sqrt	FP	12
Loads, Stores	Int or FP	3,8,30, or 200



Smarter CPUs

- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
    - Development of dedicated ISAs (Instruction Set Architectures)
        - Reduced vs Complex Instruction (RISC vs CISC)
    - Instruction pipelining (Instruction Level Parallelism)
    - OoO (Out of Order) execution
    - Branch prediction



F D E R W

Integer ADD

F D E E E E R W

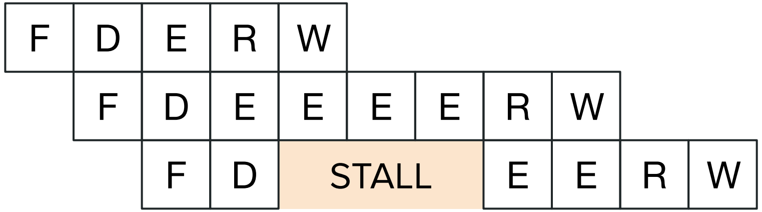
Integer MUL

F D E E R W

Integer DIF

Smarter CPUs

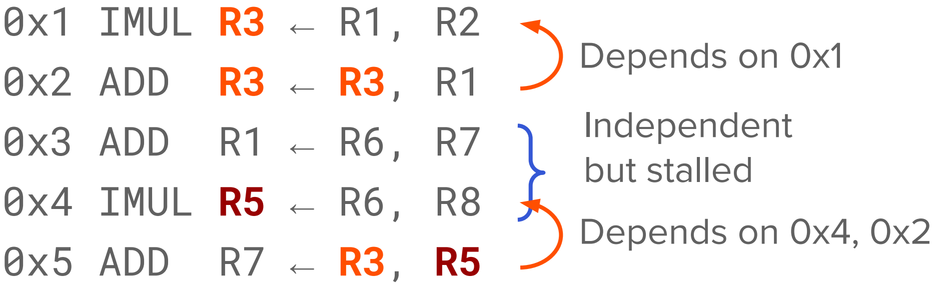
- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
  - Development of dedicated ISAs (Instruction Set Architectures)
    - Reduced vs Complex Instruction (RISC vs CISC)
  - Instruction pipelining (Instruction Level Parallelism)
  - OoO (Out of Order) execution
  - Branch prediction



F	D	E	R	W						
	F	D	E	E	E	E	R	W		
		F	D	STALL			E	E	R	W

Smarter CPUs

- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
    - Development of dedicated ISAs (Instruction Set Architectures)
        - Reduced vs Complex Instruction (RISC vs CISC)
    - Instruction pipelining (Instruction Level Parallelism)
    - OoO (Out of Order) execution
    - Branch prediction



```
0x1 IMUL R3 ← R1, R2
0x2 ADD  R3 ← R3, R1
0x3 ADD  R1 ← R6, R7
0x4 IMUL R5 ← R6, R8
0x5 ADD  R7 ← R3, R5
```

Depends on 0x1

Independent
but stalled

Depends on 0x4, 0x2

Smarter CPUs

- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
  - Development of dedicated ISAs (Instruction Set Architectures)
    - Reduced vs Complex Instruction (RISC vs CISC)
  - Instruction pipelining (Instruction Level Parallelism)
  - OoO (Out of Order) execution
  - Branch prediction

```
0x1 IMUL R3 ← R1, R2
0x2 ADD  R3 ← R3, R1
0x3 ADD  R1 ← R6, R7
0x4 IMUL R5 ← R6, R8
0x5 ADD  R7 ← R3, R5
```

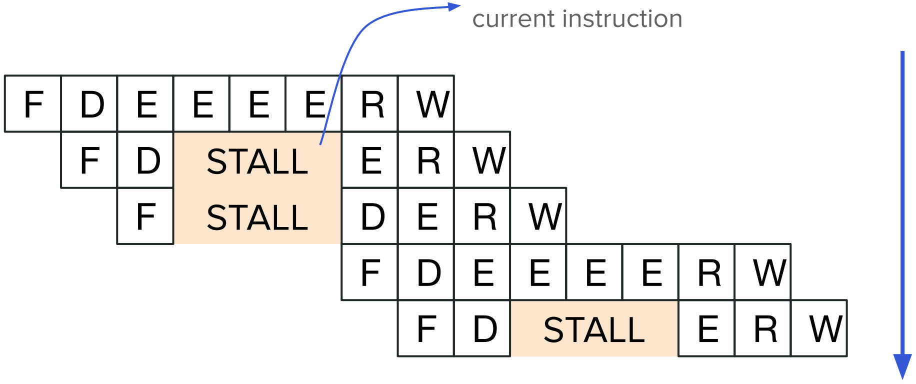
F	D	E	E	E	E	R	W							
	F	D	STALL			E	R	W						
		F	STALL			D	E	R	W					
						F	D	E	E	E	E	R	W	
							F	D	STALL			E	R	W

Assuming 1 adder & 1 multiplier that can work independently

Smarter CPUs

- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
  - Development of dedicated ISAs (Instruction Set Architectures)
    - Reduced vs Complex Instruction (RISC vs CISC)
  - Instruction pipelining (Instruction Level Parallelism)
  - OoO (Out of Order) execution
  - Branch prediction

```
0x1 IMUL R3 ← R1, R2
0x2 ADD  R3 ← R3, R1
0x3 ADD  R1 ← R6, R7
0x4 IMUL R5 ← R6, R8
0x5 ADD  R7 ← R3, R5
```



CPU unit blocked by current instruction

F	D	E	E	E	E	R	W							
	F	D	STALL			E	R	W						
		F	STALL			D	E	R	W					
						F	D	E	E	E	E	R	W	
							F	D	STALL			E	R	W

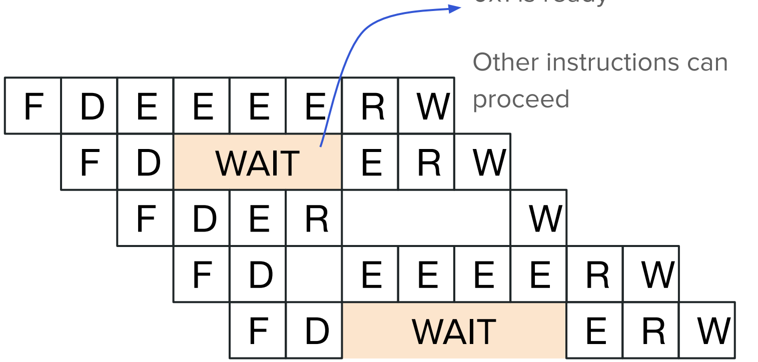
Scheduling instructions in order

(in-order dispatch)

Smarter CPUs

- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
  - Development of dedicated ISAs (Instruction Set Architectures)
    - Reduced vs Complex Instruction (RISC vs CISC)
  - Instruction pipelining (Instruction Level Parallelism)
  - OoO (Out of Order) execution
  - Branch prediction

```
0x1 IMUL R3 ← R1, R2
0x2 ADD  R3 ← R3, R1
0x3 ADD  R1 ← R6, R7
0x4 IMUL R5 ← R6, R8
0x5 ADD  R7 ← R3, R5
```



F	D	E	E	E	E	R	W					
	F	D	WAIT			E	R	W				
		F	D	E	R			W				
			F	D		E	E	E	E	R	W	
				F	D	WAIT				E	R	W

Instruction in a "waiting buffer" until the result of 0x1 is ready

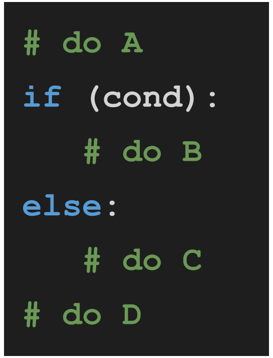
Other instructions can proceed

Scheduling instructions out-of-order

(dynamic dispatch)

Smarter CPUs

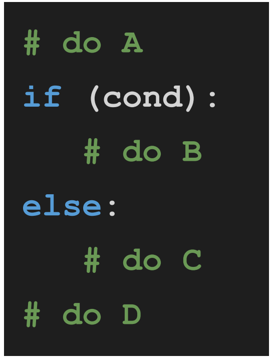
- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
    - Development of dedicated ISAs (Instruction Set Architectures)
        - Reduced vs Complex Instruction (RISC vs CISC)
    - Instruction pipelining (Instruction Level Parallelism)
    - OoO (Out of Order) execution
    - Branch prediction



```
# do A
if (cond):
    # do B
else:
    # do C
# do D
```

Smarter CPUs

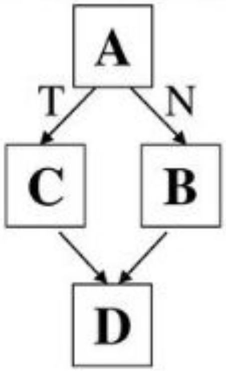
- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
    - Development of dedicated ISAs (Instruction Set Architectures)
        - Reduced vs Complex Instruction (RISC vs CISC)
    - Instruction pipelining (Instruction Level Parallelism)
    - OoO (Out of Order) execution
    - Branch prediction



```
# do A
if (cond):
    # do B
else:
    # do C
# do D
```

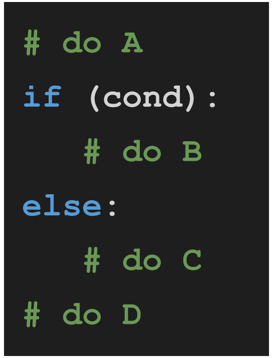
`branches/jumps`

➔ potentially similar effect to a STALL in the instruction pipeline



Smarter CPUs

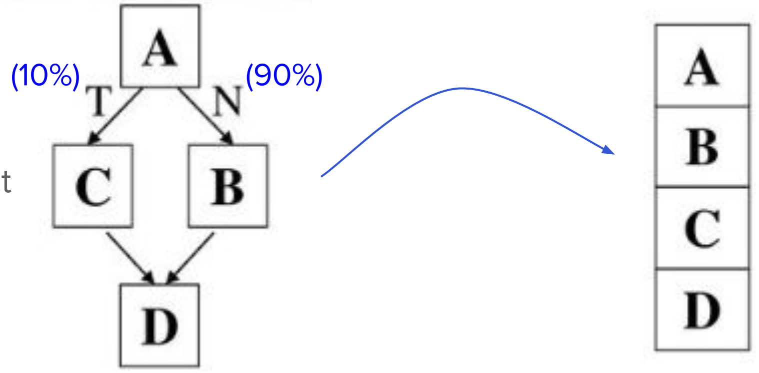
- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
    - Development of dedicated ISAs (Instruction Set Architectures)
        - Reduced vs Complex Instruction (RISC vs CISC)
    - Instruction pipelining (Instruction Level Parallelism)
    - OoO (Out of Order) execution
    - Branch prediction



```
# do A
if (cond):
    # do B
else:
    # do C
# do D
```

`branches/jumps`

→ potentially similar effect to a STALL in the instruction pipeline



“Squash” instructions following the prediction of the branch outcome

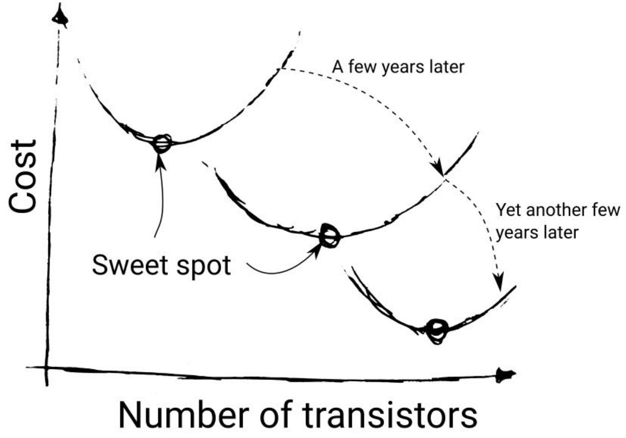
if OK → faster execution

if NOT → paying a price

Several different techniques (static/dynamic/...) for branch prediction currently available in CPUs

Moore’s law and Dennard scaling

- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
  - Development of dedicated ISAs (Instruction Set Architectures)
    - Reduced vs Complex Instruction (RISC vs CISC)
  - Instruction pipelining (Instruction Level Parallelism)
  - OoO (Out of Order) execution
  - Branch prediction
- But most importantly… by the increase of:
  - **Transistor area density** ⇒ Moore’s Law
  - CPU Clock Frequency ⇒ BUT… Dennard Scaling!

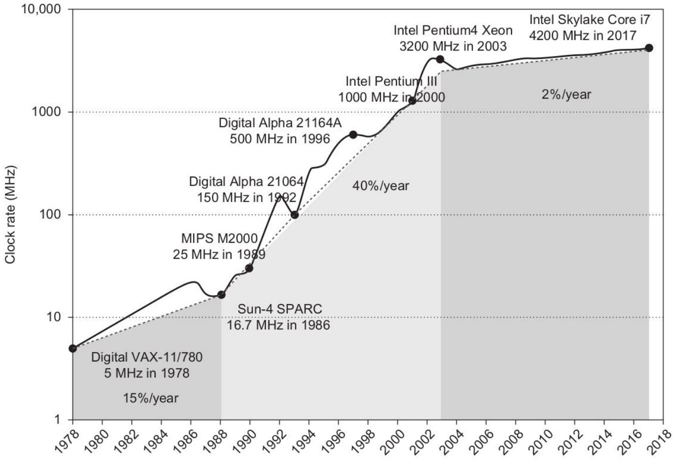


Due to technological manufacturing processes, the cost of “squeezing” transistors to a chip has a sweet spot. This changes over the years with technological improvements.

Moore's Law predicts a ~ exponential growth in optimal transistor numbers, guiding design trends, resulting in ~ doubling in density of transistors per year

Moore’s law and Dennard scaling

- Up to the early 2000s, the increased single-core performance of processors has been the main contributor towards throughput improvements of CPUs
- This was lead by multiple technological advancements in the logic of the CPUs:
  - Development of dedicated ISAs (Instruction Set Architectures)
    - Reduced vs Complex Instruction (RISC vs CISC)
  - Instruction pipelining (Instruction Level Parallelism)
  - OoO (Out of Order) execution
  - Branch prediction
- But most importantly… by the increase of:
  - Transistor area density ⇒ Moore’s Law
  - **CPU Clock Frequency** ⇒ BUT… Dennard Scaling!

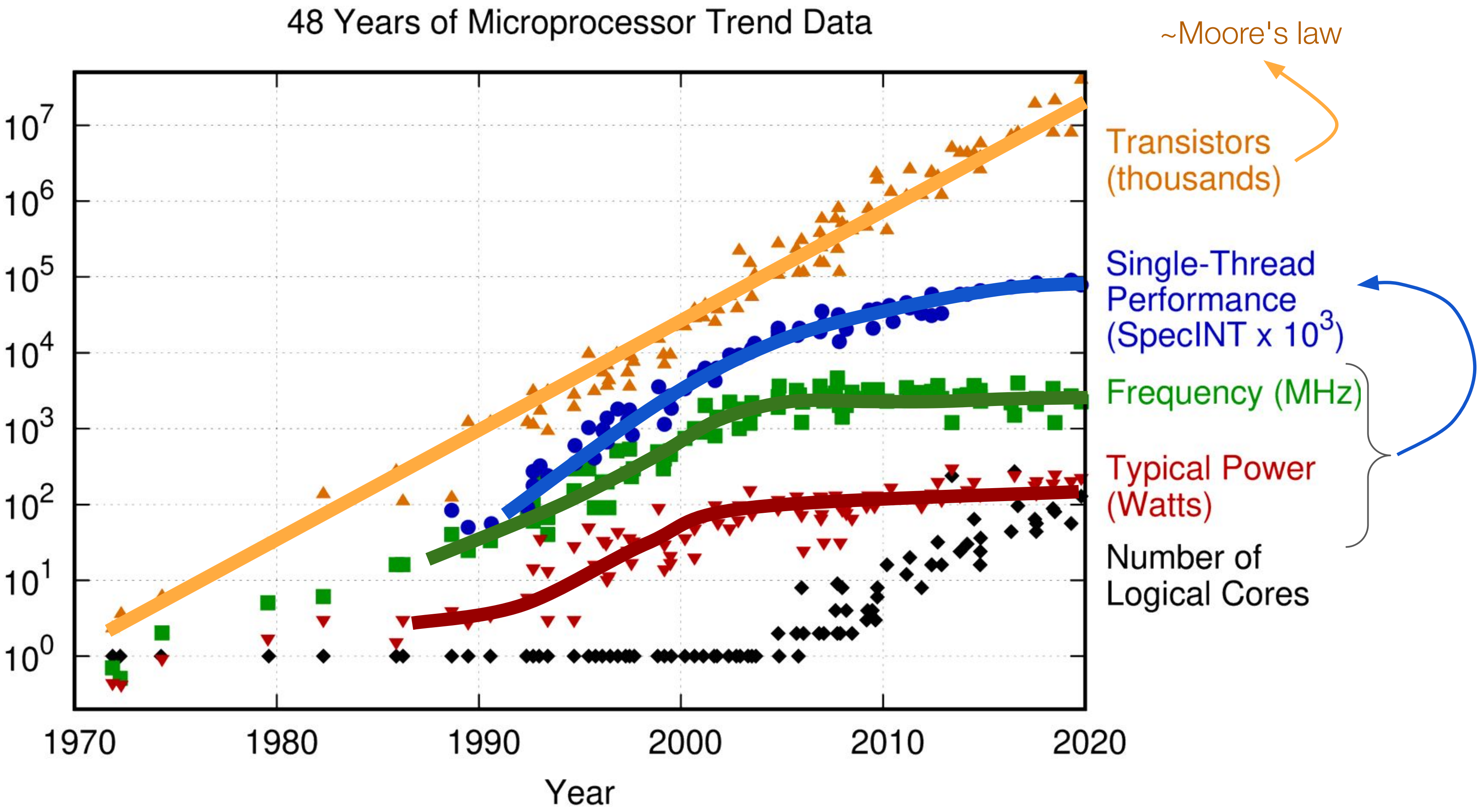


$$E_{\text{transistor}} \sim \tfrac{1}{2} C V^2$$

$$\Rightarrow \quad P_{\text{transistor}} \sim \tfrac{1}{2} C V^2 f + I V_{\text{latency}}$$

(the original Dennard scaling ignores that we also have some leakage…)

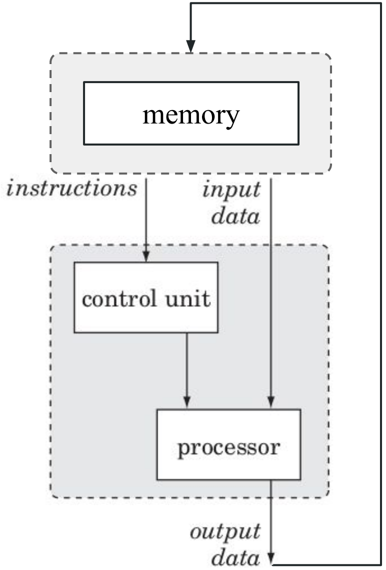
- Reduce voltage → reduces power consumption but risks unreliable switching if too low, already near the limit in modern chips
- Reduce transistor size → improves efficiency but is limited by physical constraints, increasing costs and potential manufacturing errors
- Reduce frequency → reduce voltage and save power, but it slows down the chip's performance.

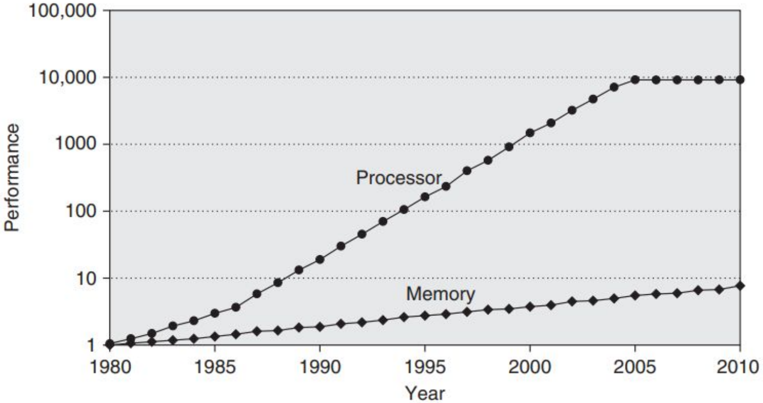


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

Memory and Von Neumann's bottleneck

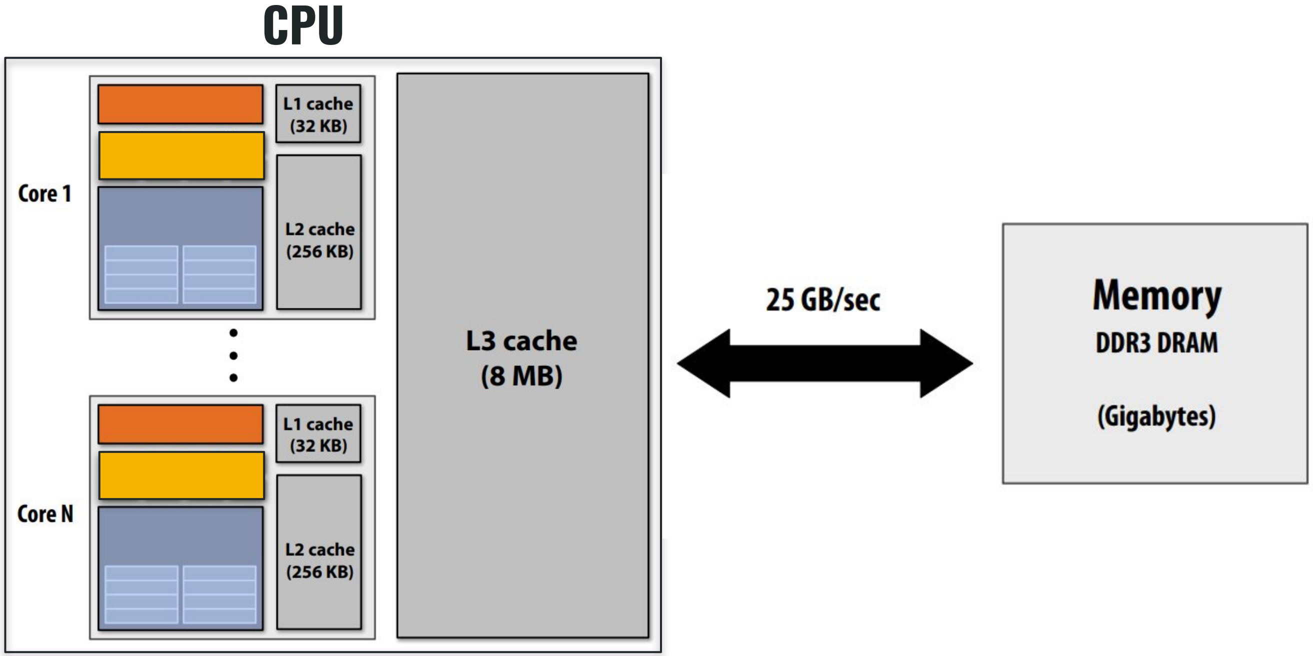
- In classical von Neumann architectures processors are directly connected to main memory
  - Fetch instructions
  - Fetch data
- Memory (usually DRAM) access latency is "small"... but "how small"?
  - Overtime, computation performance grew faster than main memory access speed
  - Modern processors process data much faster than reading data from main memory (*von Neumann bottleneck*)
- Latency is one thing, but there is another side of the medal
  - 1 (for now) CPU continuously accessing memory for instructions and data
  - Continuous data movement across the system BUS between Memory and CPU ⇒ High Bandwidth/Throughput is key!
- Unfortunately, memory performance increase has been less than the one of the processors





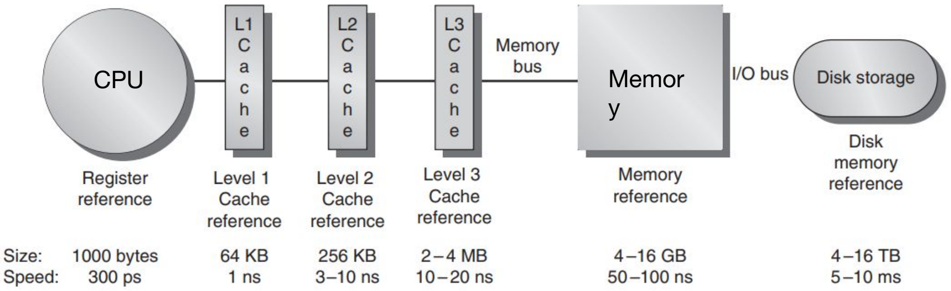
Multiprocessor architectures

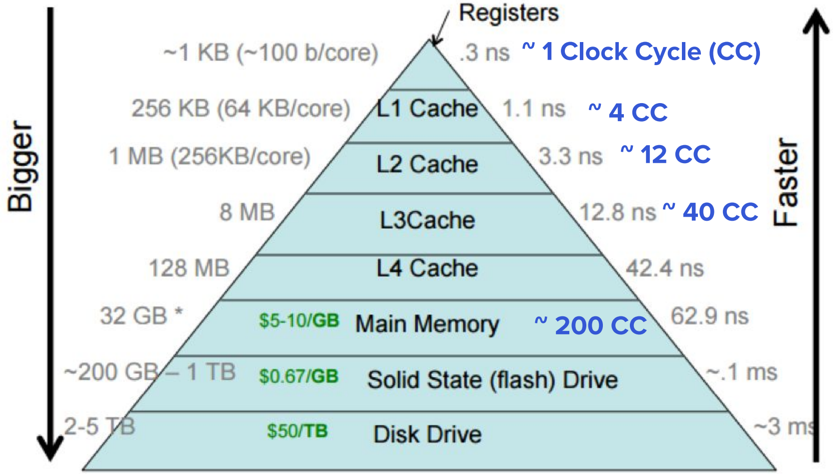
- Since around 2003 a trend in expanding on the number of processing units (cores) in CPUs to exploit threading and multiprogramming
- Multi-core CPUs and Multi-CPUs systems have now become the norm
- L1 cache and L2 cache layers assigned to each CPU core, and L3 cache shared across multiple cores of the same CPU “package”



Memory and Von Neumann’s bottleneck

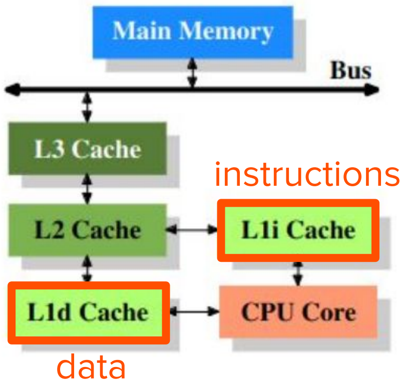
- What to do, then?
  - Create “memory layers” (cache) between CPU registers and main memory
  - Embed on a single chip (System on Chip, SoC) CPU, registers, and cache
  - Separate paths for data and instructions
  - Potentially, allow dedicated memory access per CPU core (more on that later)
- Caching means having data stored in a “preferred location”
  - Fast access, but must be (relatively) sure to cache only what is necessary
  - Cache hit/miss logic implemented on CPUs to aim at cache when possible, and revert to main memory in the case of a cache miss

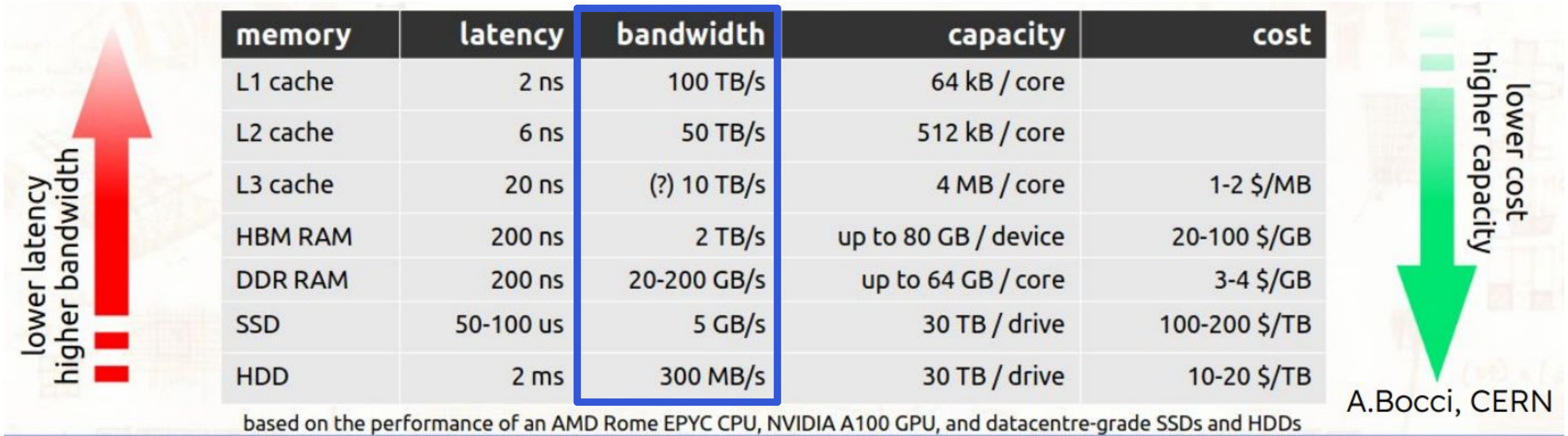




Memory and Von Neumann's bottleneck

- What to do, then?
  - Create "memory layers" (cache) between CPU registers and main memory
  - Embed on a single chip (System on Chip, SoC) CPU, registers, and cache
  - Separate paths for data and instructions
  - Potentially, allow dedicated memory access per CPU core (more on that later)
- Caching means having data stored in a "preferred location"
  - Fast access, but must be (relatively) sure to cache only what is necessary
  - Cache hit/miss logic implemented on CPUs to aim at cache when possible, and revert to main memory in the case of a cache miss





lower latency
higher bandwidth
lower cost
higher capacity

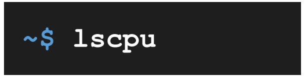
memory	latency	bandwidth	capacity	cost
L1 cache	2 ns	100 TB/s	64 kB / core	
L2 cache	6 ns	50 TB/s	512 kB / core	
L3 cache	20 ns	(?) 10 TB/s	4 MB / core	1-2 $/MB
HBM RAM	200 ns	2 TB/s	up to 80 GB / device	20-100 $/GB
DDR RAM	200 ns	20-200 GB/s	up to 64 GB / core	3-4 $/GB
SSD	50-100 us	5 GB/s	30 TB / drive	100-200 $/TB
HDD	2 ms	300 MB/s	30 TB / drive	10-20 $/TB

based on the performance of an AMD Rome EPYC CPU, NVIDIA A100 GPU, and datacentre-grade SSDs and HDDs

A.Bocci, CERN

Multiprocessor architectures

- Since around 2003 a trend in expanding on the number of processing units (cores) in CPUs to exploit threading and multiprogramming
- Multi-core CPUs and Multi-CPUs systems have now become the norm
- L1 cache and L2 cache layers assigned to each CPU core, and L3 cache shared across multiple cores of the same CPU “package”



```
~$ lscpu
```

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
Stepping:                        12
CPU MHz:                         2300.000
CPU max MHz:                     4900,0000
CPU min MHz:                     400,0000
BogoMIPS:                        4599.93
Virtualization:                  VT-x
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        8 MiB
```

Multiprocessor architectures

- Since around 2003 a trend in expanding on the number of processing units (cores) in CPUs to exploit threading and multiprogramming
- Multi-core CPUs and Multi-CPUs systems have now become the norm
- L1 cache and L2 cache layers assigned to each CPU core, and L3 cache shared across multiple cores of the same CPU “package”

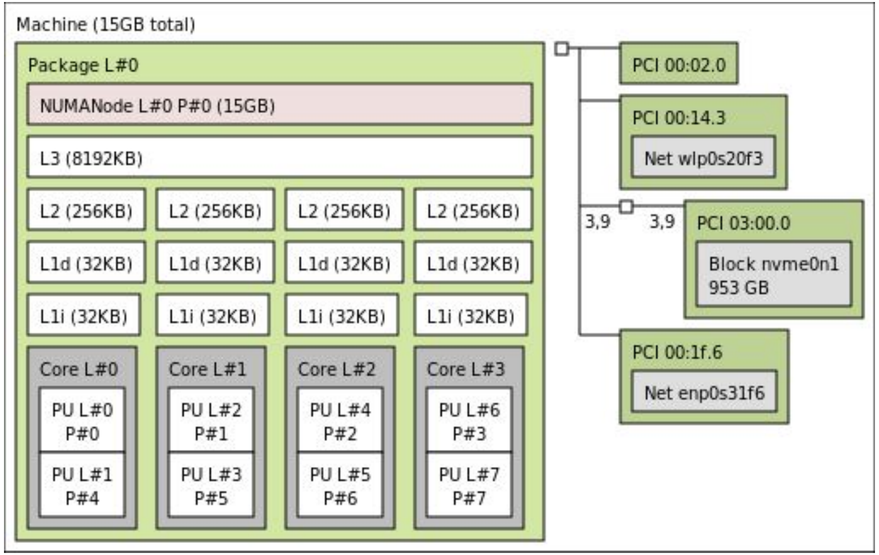
```
~$ lstopo
```

Much larger (8 MB) L3 cache, shared across all cores

Larger (256 KB) L2 cache

Small (32 KB) L1 cache, split into instruction and data cache

2 logical cores per physical one (hyper-threading)



Machine (15GB total)

Package L#0

NUMANode L#0 P#0 (15GB)

L3 (8192KB)

L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)

L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)

L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB)

Core L#0: PU L#0 P#0, PU L#1 P#4

Core L#1: PU L#2 P#1, PU L#3 P#5

Core L#2: PU L#4 P#2, PU L#5 P#6

Core L#3: PU L#6 P#3, PU L#7 P#7

PCI 00:02.0

PCI 00:14.3

Net wlp0s20f3

3,9 3,9

PCI 03:00.0

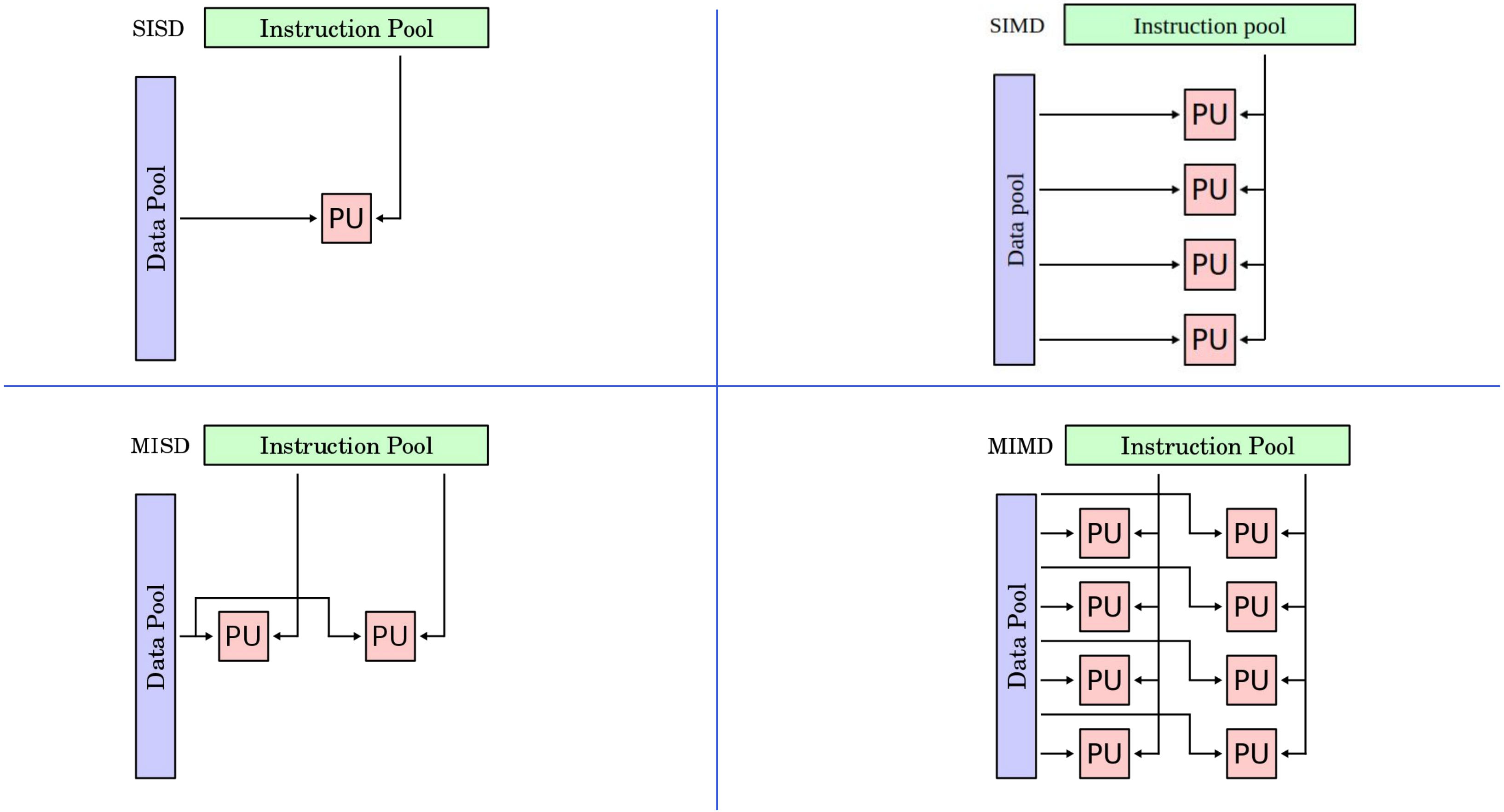
Block nvme0n1 953 GB

PCI 00:1f.6

Net enp0s31f6

4 physical cores

Flynn’s Taxonomy



Flynn’s Taxonomy

Single **sequential** processing units (PU) operates on a single stream of data. (traditional von Neumann architecture)

The **same operation (instruction!)** performed on multiple data items **simultaneously**.

- Specific CPUs instruction
- Dedicated Vector machines

Multiple PUs execute **different instructions simultaneously on a single stream of data**.

Multiple PUs execute **different instructions on different data streams**.

- Separate PUs run on different data
- Most flexible but more complex
- Multicore CPUs parallelism

Flynn’s Taxonomy

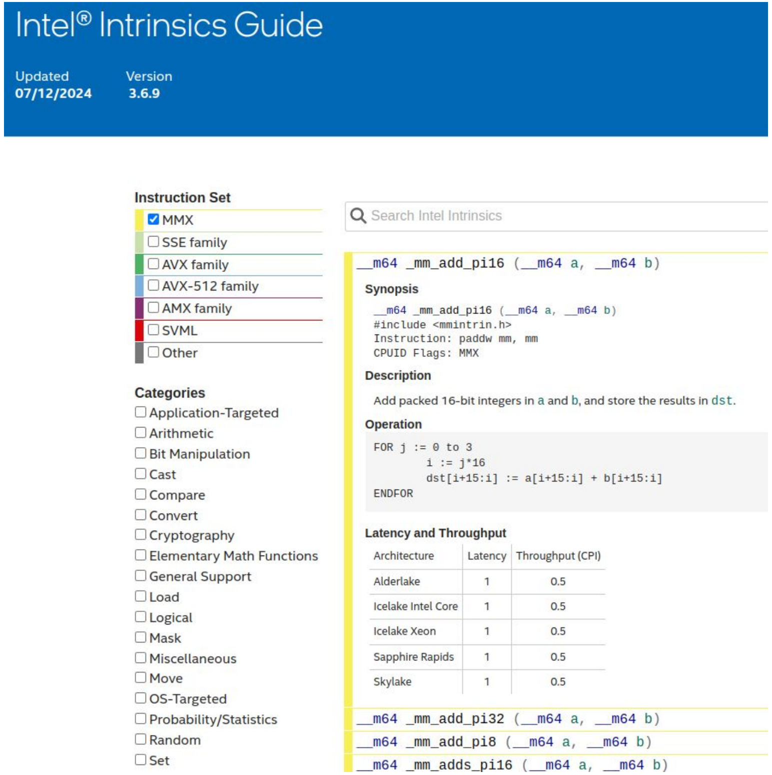
- Modern multi-core CPUs can work as MIMD but also contain SIMD instructions for vectorized operations

```
~$ lscpu
```

```
Vendor ID:                 GenuineIntel
  Model name:              Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
    CPU family:            6
    Model:                 142
    Thread(s) per core:    2
    Core(s) per socket:    4
    Socket(s):             1
    Stepping:              12
    CPU max MHz:           4900,0000
    CPU min MHz:           400,0000
    BogoMIPS:              4599.93
    Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
                           pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
                           pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
                           rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq
                           dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid
                           sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
                           f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single
                           ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid
                           ept_ad fsgsbase tsc_adjust sgx bmi1 avx2 smep bmi2 erms invpcid mpx
                           rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves
                           dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp vnmi
                           md_clear flush_l1d arch_capabilities
```

Flynn’s Taxonomy

- Modern multi-core CPUs can work as MIMD but also contain SIMD instructions for vectorized operations



Intel® Intrinsics Guide

Updated 07/12/2024 Version 3.6.9

Instruction Set

☑ MMX
□ SSE family
□ AVX family
□ AVX-512 family
□ AMX family
□ SVML
□ Other

Categories

□ Application-Targeted
□ Arithmetic
□ Bit Manipulation
□ Cast
□ Compare
□ Convert
□ Cryptography
□ Elementary Math Functions
□ General Support
□ Load
□ Logical
□ Mask
□ Miscellaneous
□ Move
□ OS-Targeted
□ Probability/Statistics
□ Random
□ Set

```
__m64 _mm_add_pi16 (__m64 a, __m64 b)
```

Synopsis

```
__m64 _mm_add_pi16 (__m64 a, __m64 b)
#include <mmintrin.h>
Instruction: paddw mm, mm
CPUID Flags: MMX
```

Description

Add packed 16-bit integers in a and b, and store the results in dst.

Operation

```
FOR j := 0 to 3
        i := j*16
        dst[i+15:i] := a[i+15:i] + b[i+15:i]
ENDFOR
```

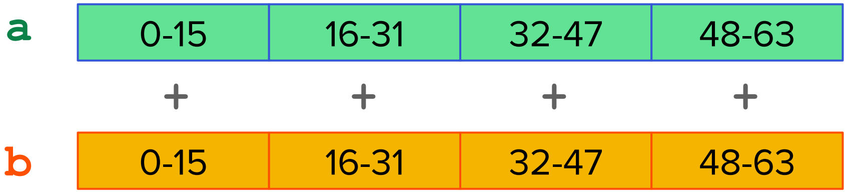
Latency and Throughput

Architecture	Latency	Throughput (CPI)
Alderlake	1	0.5
Icelake Intel Core	1	0.5
Icelake Xeon	1	0.5
Sapphire Rapids	1	0.5
Skylake	1	0.5

```
__m64 _mm_add_pi32 (__m64 a, __m64 b)
__m64 _mm_add_pi8 (__m64 a, __m64 b)
__m64 _mm_adds_pi16 (__m64 a, __m64 b)
```

E.g. MMX’s `_mm_add_pi16`:

Adds 2 operands `a, b` composed by 4x 16 bits INT (packaged integers)

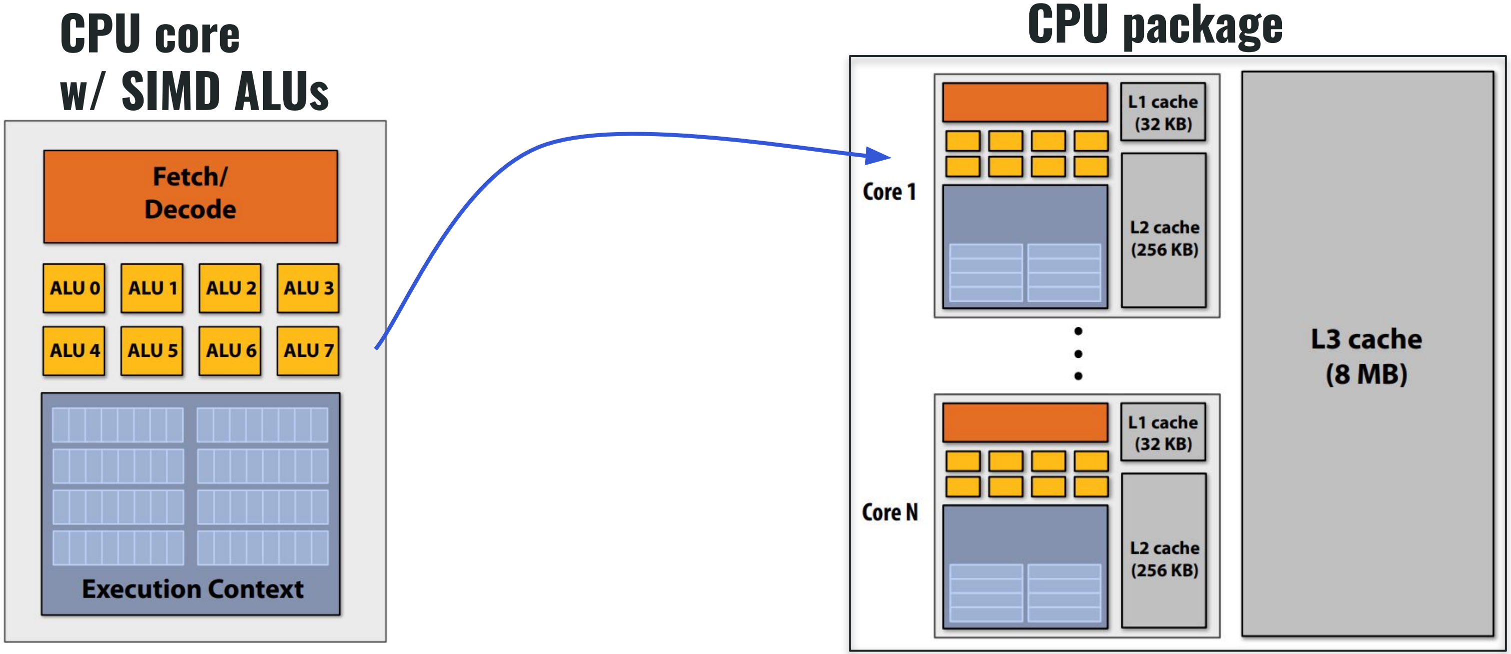


a	0-15	16-31	32-47	48-63
	+	+	+	+
b	0-15	16-31	32-47	48-63

Effectively… vector sum of 4 elements at a time with 1 clock cycle latency (instead of 4 w/ pipelining)

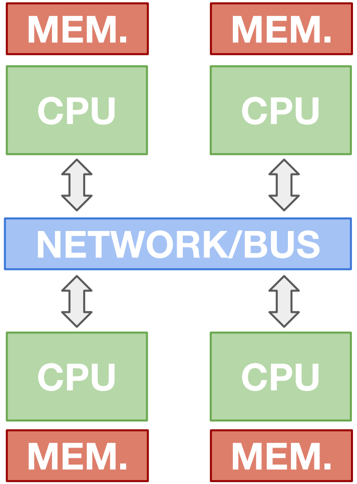
Flynn’s Taxonomy

- Modern multi-core CPUs can work as MIMD but also contain SIMD instructions for vectorized operations

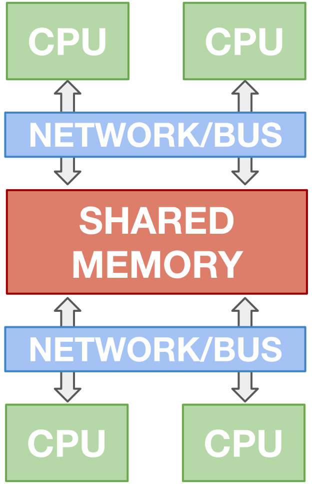


SPMD (Same Program Multiple Data)

- Not one of the original categories of Flynn's Taxonomy, but a very widely adopted extension of it
- Extremely viable parallelization approach for Data-Parallel tasks
  - One single task running on a pool of processes/threads, each one acting on a different data element/file/stream
- Each individual process in execution on a CPU can leverage from SIMD-like instructions, Pipelining, OoO Execution, etc
- Mostly useful in distributed memory systems, but still relatively common also in shared memory ones



Distributed memory



Shared memory

Scaling - Speedup - Efficiency

- Measure the capacity of a parallel system to deliver increasing performance with the number of processors used.
- It reflects the ability of hardware (and software!) to effectively exploit larger computational resources
- Scaling of the performance of processing with the number of processing units is often measured via the **Speedup** and **Efficiency** factors

Scaling - Speedup - Efficiency

- Measure the capacity of a parallel system to deliver increasing performance with the number of processors used.
- It reflects the ability of hardware (and software!) to effectively exploit larger computational resources
- Scaling of the performance of processing with the number of processing units is often measured via the **Speedup** and **Efficiency** factors

$$S(n) = \frac{T_1}{T_n}$$

Ideally, the **Speedup** would be linear, where the execution time scales perfectly with the number of processor elements.

Scaling - Speedup - Efficiency

- Measure the capacity of a parallel system to deliver increasing performance with the number of processors used.
- It reflects the ability of hardware (and software!) to effectively exploit larger computational resources
- Scaling of the performance of processing with the number of processing units is often measured via the **Speedup** and **Efficiency** factors

$$S(n) = \frac{T_1}{T_n}$$

$$E(n) = \frac{S(n)}{n}$$

Ideally, the **Speedup** would be linear, where the execution time scales perfectly with the number of processor elements.

Ideally, the **Efficiency** would be 1 if the Speedup is linear, meaning that all n PUs could perfectly take exactly 1/n of the computation.

Scaling - Speedup - Efficiency

- Measure the capacity of a parallel system to deliver increasing performance with the number of processors used.
- It reflects the ability of hardware (and software!) to effectively exploit larger computational resources
- Scaling of the performance of processing with the number of processing units is often measured via the **Speedup** and **Efficiency** factors

$$S(n) = \frac{T_1}{T_n}$$

$$E(n) = \frac{S(n)}{n}$$

Ideally, the **Speedup** would be linear, where the execution time scales perfectly with the number of processor elements.

Ideally, the **Efficiency** would be 1 if the Speedup is linear, meaning that all n PUs could perfectly take exactly 1/n of the computation.

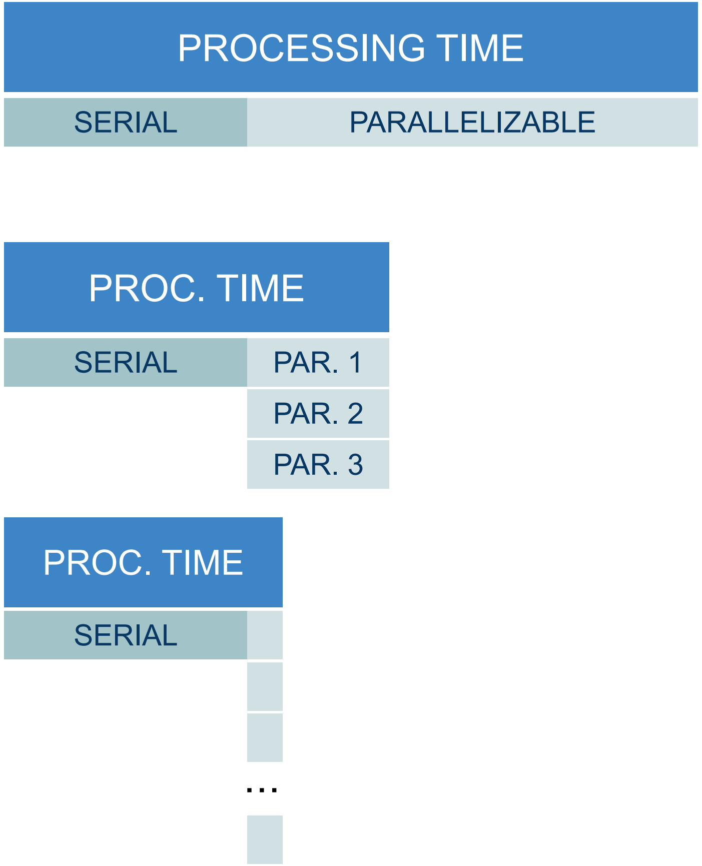
However, neither of these conditions is almost ever the case due to initialization costs, communication overheads, limited parallel fraction of the process

Strong and Weak Scaling

- Given a **fixed problem size** with a given fraction *p* being parallelizable and the rest *(1-p)* being serial/sequential
- The Scaleup is dependent on the amount of processing units *n* we can parallelize the task on
- The number of processors is increased while the problem size remains constant
- Mostly related to CPU-bound applications to find the best configuration for a fixed problem provided an increasing number of PUs

→ The scaleup is described by the Amdahl's law

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

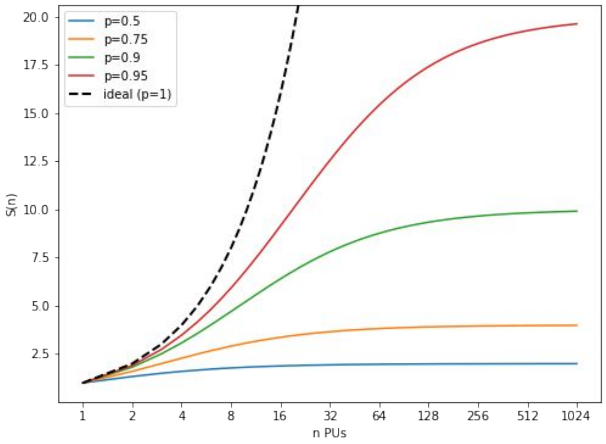


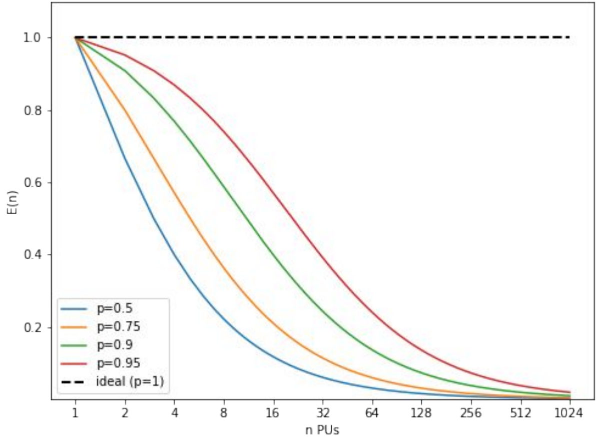
Strong and Weak Scaling

- Given a **fixed problem size** with a given fraction *p* being parallelizable and the rest *(1-p)* being serial/sequential
- The Scaleup is dependent on the amount of processing units *n* we can parallelize the task on
- The number of processors is increased while the problem size remains constant
- Mostly related to CPU-bound applications to find the best configuration for a fixed problem provided an increasing number of PUs

→ The scaleup is described by the Amdahl's law

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$





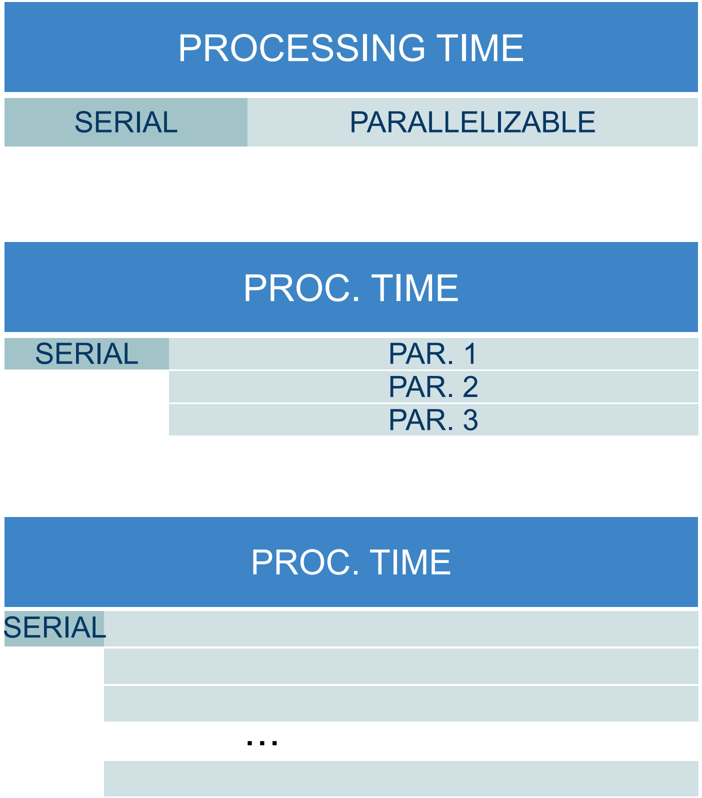
E.g.: Given a fixed dataset, can perform a given data analysis task in shorter times with increasing computational power

Strong and Weak Scaling

- Given a **non-fixed problem size** with a given fraction *p* being parallelizable and the rest *(1-p)* being serial/sequential
- The Scaleup is dependent on the amount of processing units *n* we can parallelize the task on
- **But also** the problem size can increase as the number of processors is increased
- Mostly related to Memory-bound applications to maintain high computing efficiency

→ The scaleup is described by the Gustafson’s law

$$S(n) = (1 - p) + p\,n$$

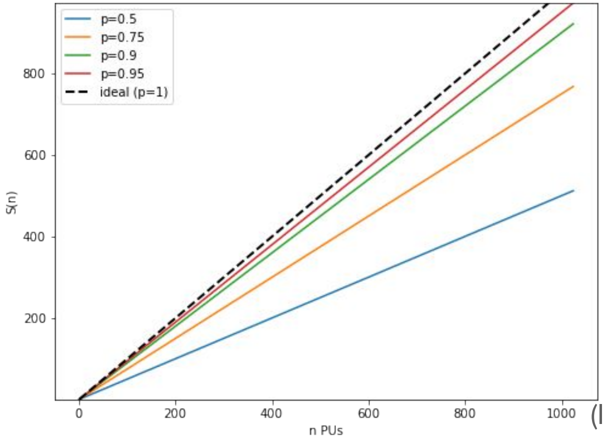


Strong and Weak Scaling

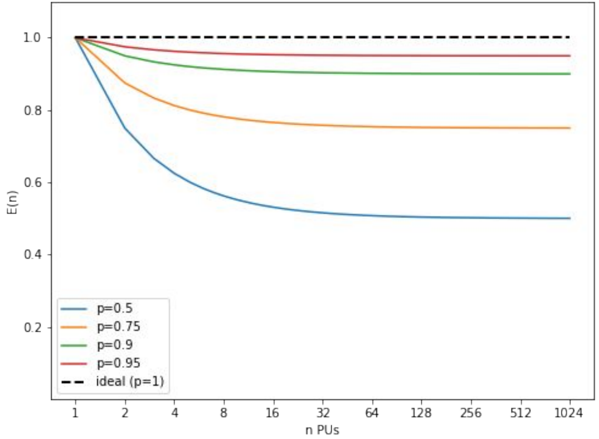
- Given a **non-fixed problem size** with a given fraction *p* being parallelizable and the rest *(1-p)* being serial/sequential
- The Scaleup is dependent on the amount of processing units *n* we can parallelize the task on
- **But also** the problem size can increase as the number of processors is increased
- Mostly related to Memory-bound applications to maintain high computing efficiency

→ The scaleup is described by the Gustafson's law

$$S(n) = (1 - p) + p\,n$$

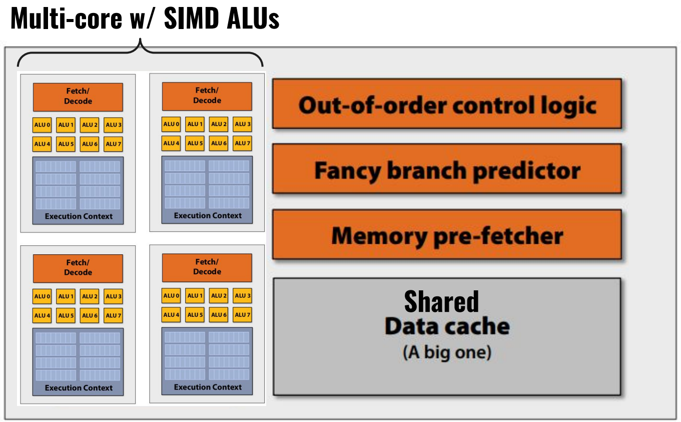


(linear scale)



E.g.: Given a weather forecasting problem, can compute higher accuracy models with increasing computational power

Overall...



- Modern CPUs embody most of the computational “tricks” we came up with throughout the years
  - High clock frequency
  - Branch-prediction
  - OoO execution
- CPU Registers ↔ Memory transactions are facilitated by layers of (relatively) large, high-bandwidth cache
  - L1i/d, L2, L3(shared)
- Multi-core architectures and complex ILAs allow to exploit some (limited) parallelism
  - SIMD instructions
  - MIMD-like multi-core CPU packages

⇒ **The CPUs are workhorses, dedicated to complex and (mostly) serial computation, with (some) multi-core capabilities**

What does any of this have to do with GPUs…?

The big picture

- Understanding what is the CPU made for, and what are its strengths and weaknesses, it’s paramount before embarking on studying other computing architectures
- It is important to avoid the pitfall of believing that “a GPU will provide 10x / 100x / 1000x performance boost over a CPU” just because…

- What is a GPU architecture, actually?
- How does it compare with the CPU?
- Can a GPU withstand all computation we may throw at it, or are there limitations?
- How can we exploit it at best?
- Is there anything we can do to make it perform better?
- Can we combine CPU and GPU in our computation?