

# Introduction to Parallel and Distributed Systems

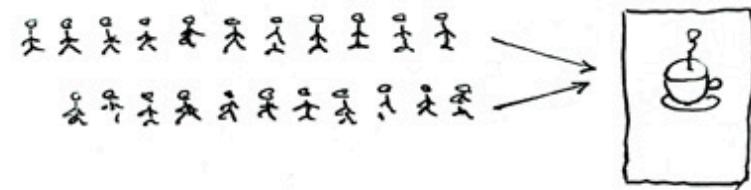
# Definitions

? set of operations

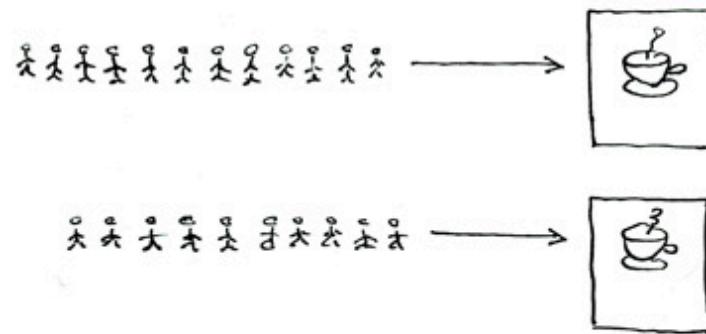
- Sequential execution : one or more tasks are executed one after the other.
  - “Usual” mode of execution of most programs
  - Tasks can be instructions, methods/functions...
- Parallel execution : one or more tasks are executed simultaneously.
  - Becoming the norm now
  - Requires special hardware
- Concurrent execution : one or more tasks are in progress at the same time
  - They don’t necessarily execute at the same time
  - From an outsider point of view, they progress
  - Usually implemented in programming languages and libraries

Tasks  
Parallel  
Concurrent

Concurrent = Two Queues One Coffee Machine



Parallel = Two Queues Two Coffee Machines



© Joe Armstrong 2013

# What is parallelism useful for?

- Going faster
  - Straightforward application of the definitions
  - Speed-up the execution
- Going larger
  - Manage much larger datasets
- Not necessarily the same techniques/tools for both

# Hardware support

Flynn's taxonomy

# Flynn's taxonomy (1966)

→ So old!

- Based on the 2 objectives of parallelism
- From a computer/program point of view, what does faster or larger mean?
  - Faster : more instructions per unit of time
  - Larger : processing more data per unit of time

# Flynn's taxonomy

	Single Instruction	Multiple Instructions	
Single Data	SISD	SIMD	MISD
Multiple Data	MISD	MISD	MISD

*1 data / time unit*

*1 instruction / time unit*

*SI ND*

*MISD*

*SIMD MISD*

- There are 4 types of parallel machines
  - Depends on how the manage instructions and data

# SISD

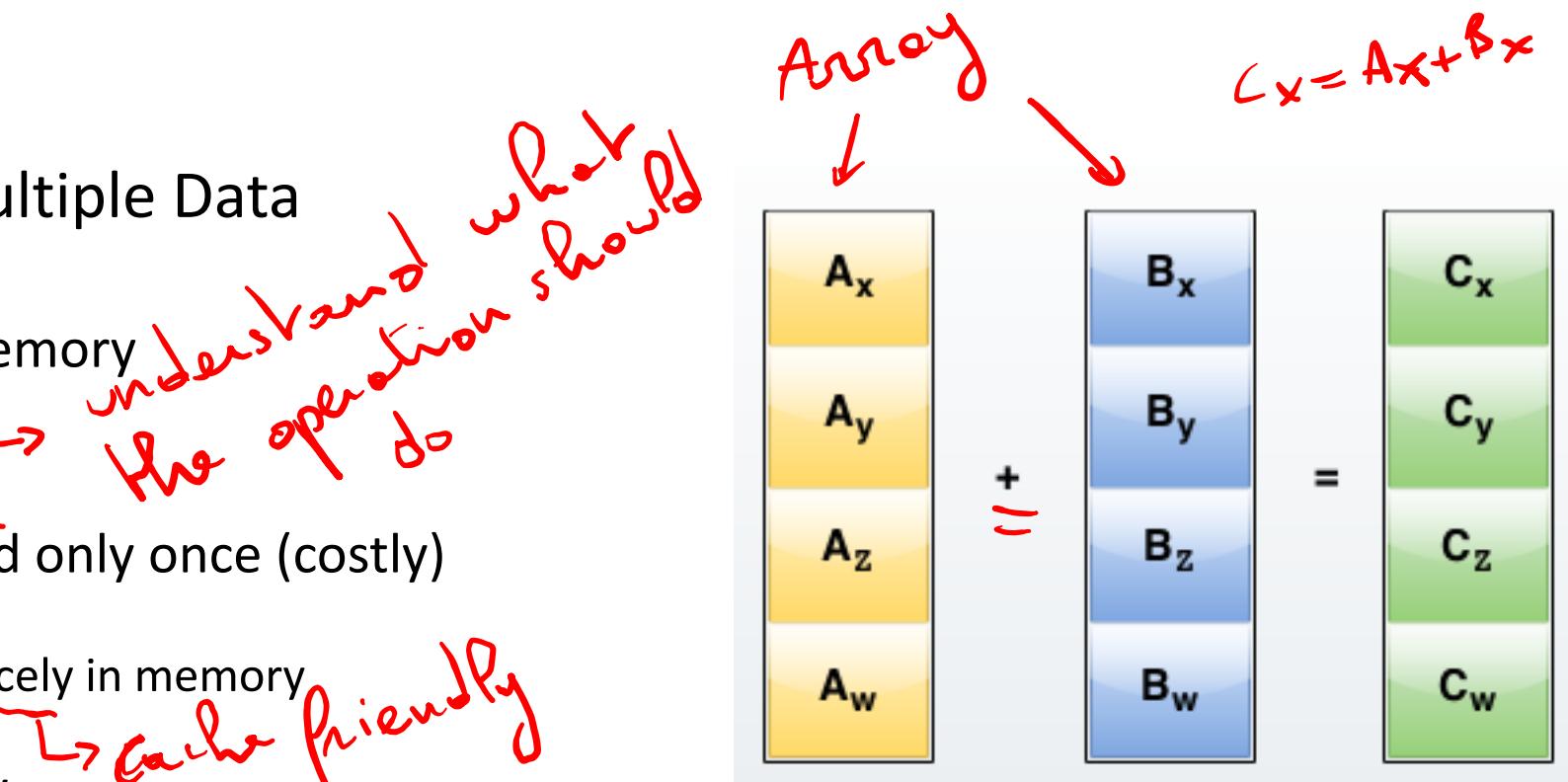
- Single Instruction, Single Data
- Takes
  - 1 instruction from memory
  - 1 data (a variable, a memory address...) ↴
- Executes, then do it again
- Example :
  - Very basic processor

( mono - core )

instruction == Number

# SIMD

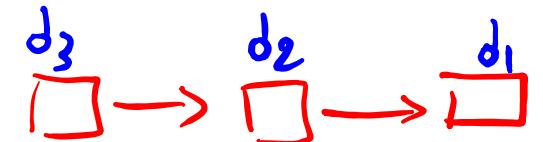
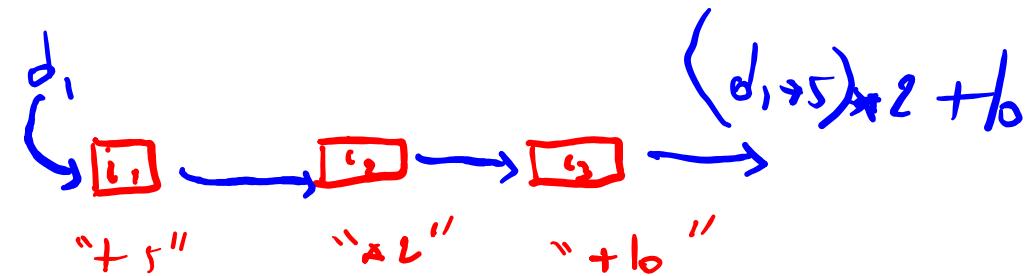
- Single Instruction, Multiple Data
- Takes
  - 1 instruction from memory
  - Multiple data
- Benefits
  - Instruction is decoded only once (costly)
  - Data
    - Can be organized nicely in memory
    - Loaded at once
  - Overall cache friendly
- Example:
  - Vector processors
  - Vector instructions in processors



Source: <http://man.hubwiz.com/> JavaScript SIMD API

# MISD

- Multiple Instructions, Single Data
- Principle
  - 1 data from memory
  - Apply multiple instructions on it
- The data “travels” from one instruction to another
  - It’s a pipeline
- Example:
  - Any modern processor has a pipeline for instructions (Fetch-Decode-Execute-Write back)
    - An instruction is processed in steps by a pipeline

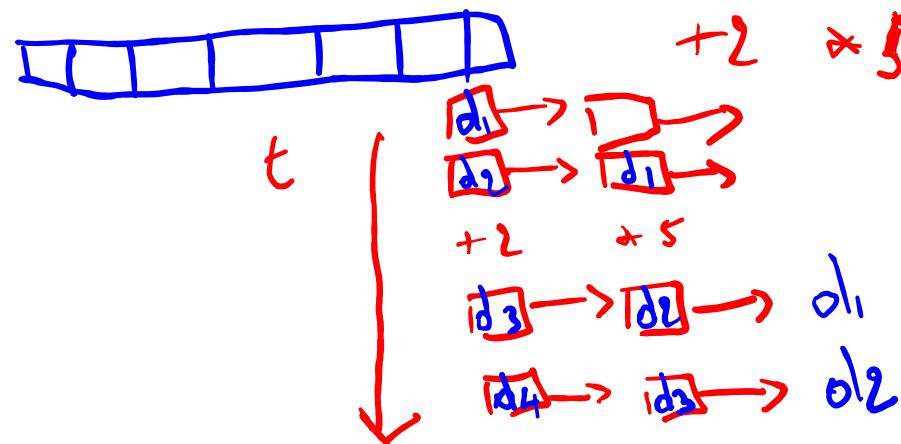
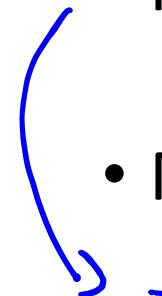


# MIMD

- Multiple Instructions, Multiple Data
- Basically multiple programs working on different data
- Examples:
  - Multiple computers
  - Multiple processors
  - Multi-cores

# Flynn's taxonomy and parallelism

- SISD: Not a parallel machine
- SIMD :
  - Process each data with a different processing unit
- MISD : (pipeline)
  - As soon as an instruction is finished, load next data
- MIMD : Naturally parallel

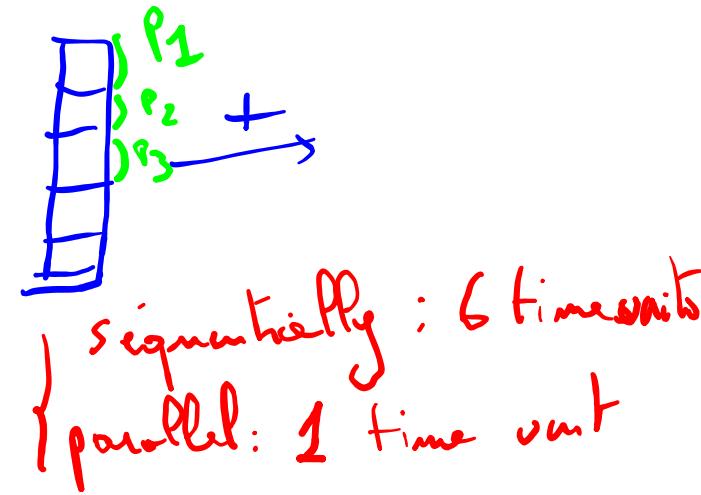


+2  $\times 5$  on each cell

{ sequentially : 7 time unit  
parallel :

- first data finishes after 2.
- Next data finishes at +1

$$\sum = 2 + 6 = 8 \text{ time unit}$$

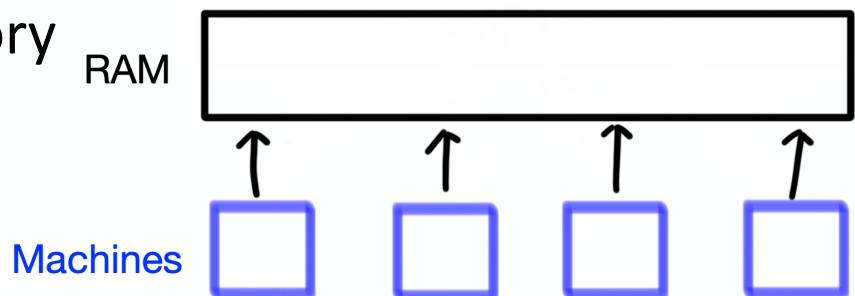


in theory

# Other taxonomy

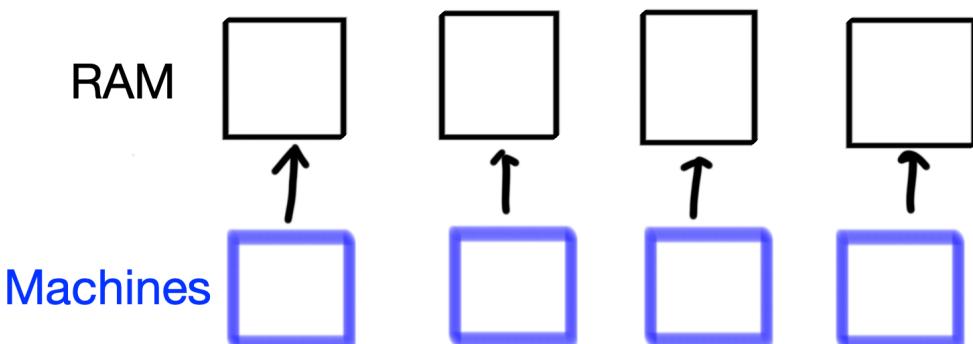
- Based on how memory is shared among processing units
- Shared memory

- All processing units can access the same memory



- Private memory (aka Distributed Systems)

- Each processing unit has some private memory invisible to others



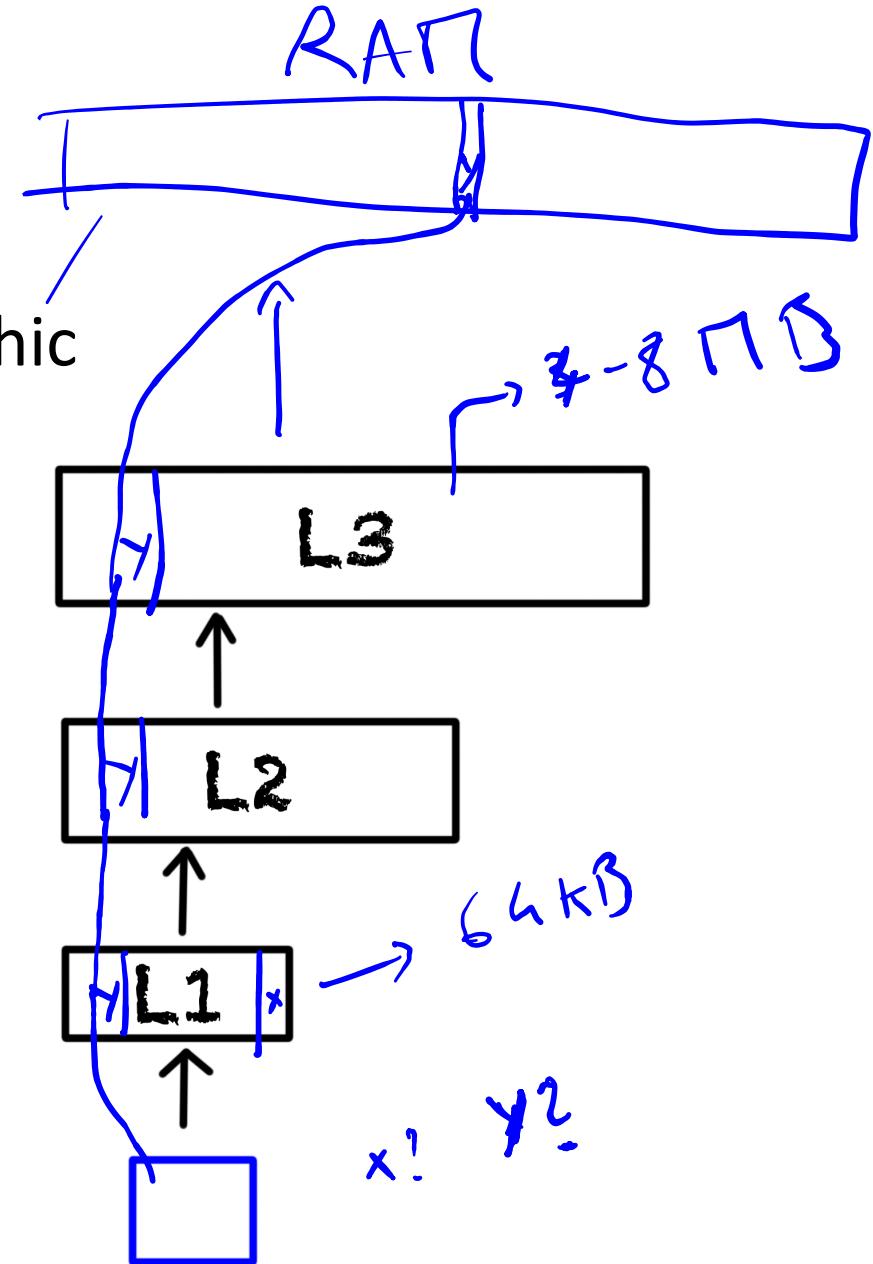
# Hardware support

From single core to multicores

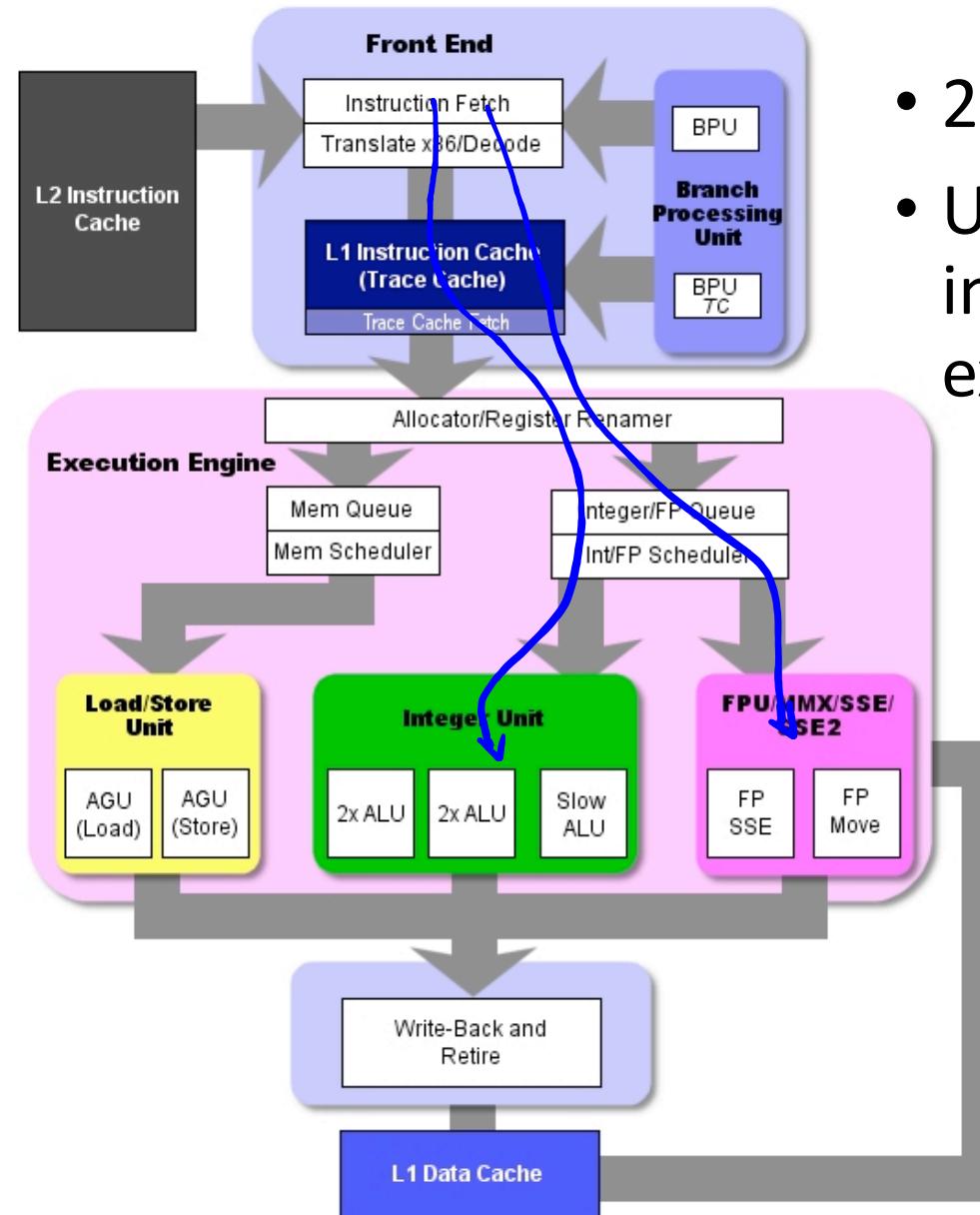
# Inside processors

- Processors are complex and not monolithic
- Many different functional units
  - Integer computation, floating point...
- Complex memory hierarchy
  - Different cache levels (L1, L2, L3)
  - Fast caches are small

*small but very  
fast memory where  
you store recently used data*



# Pentium 4

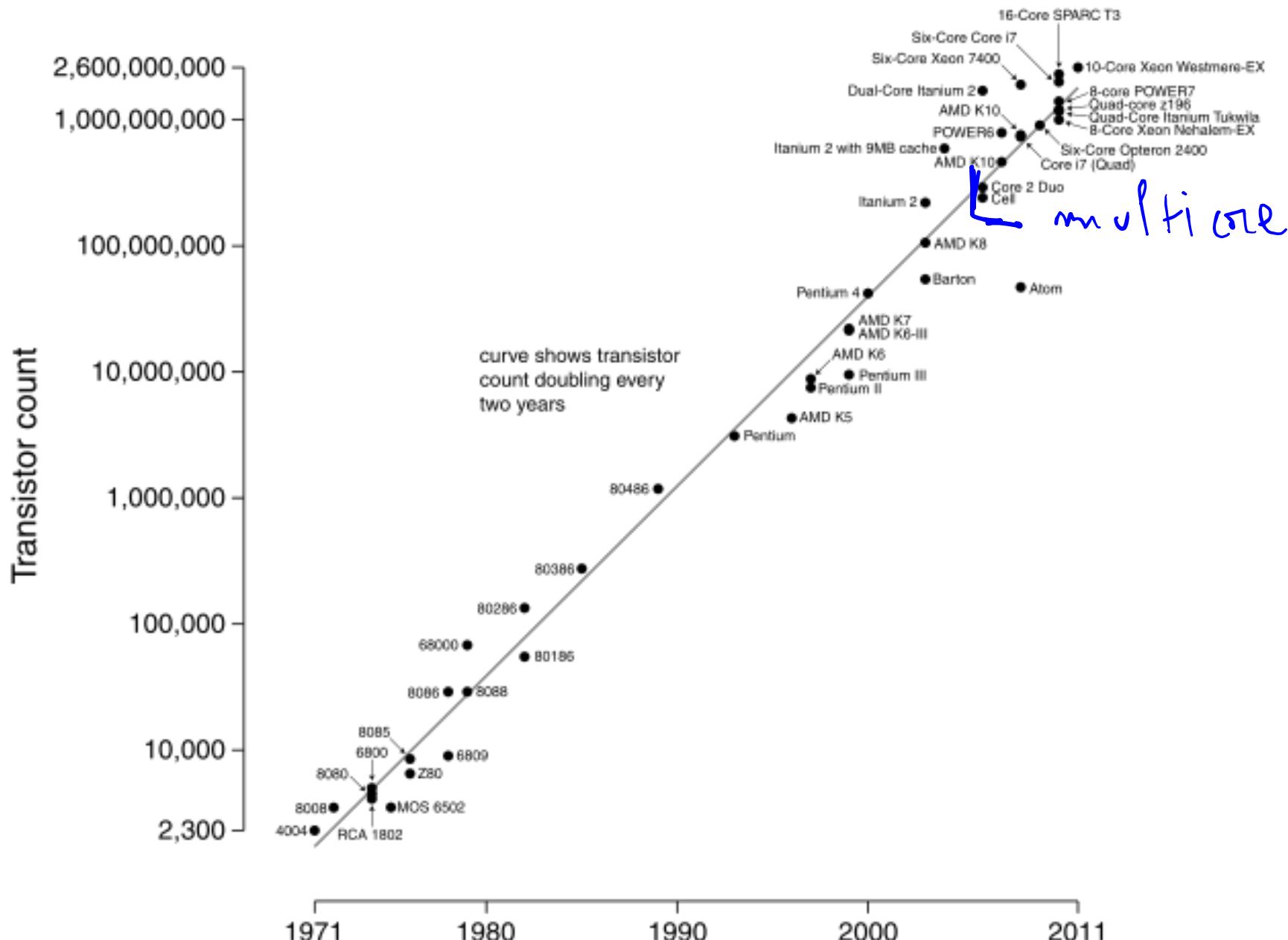


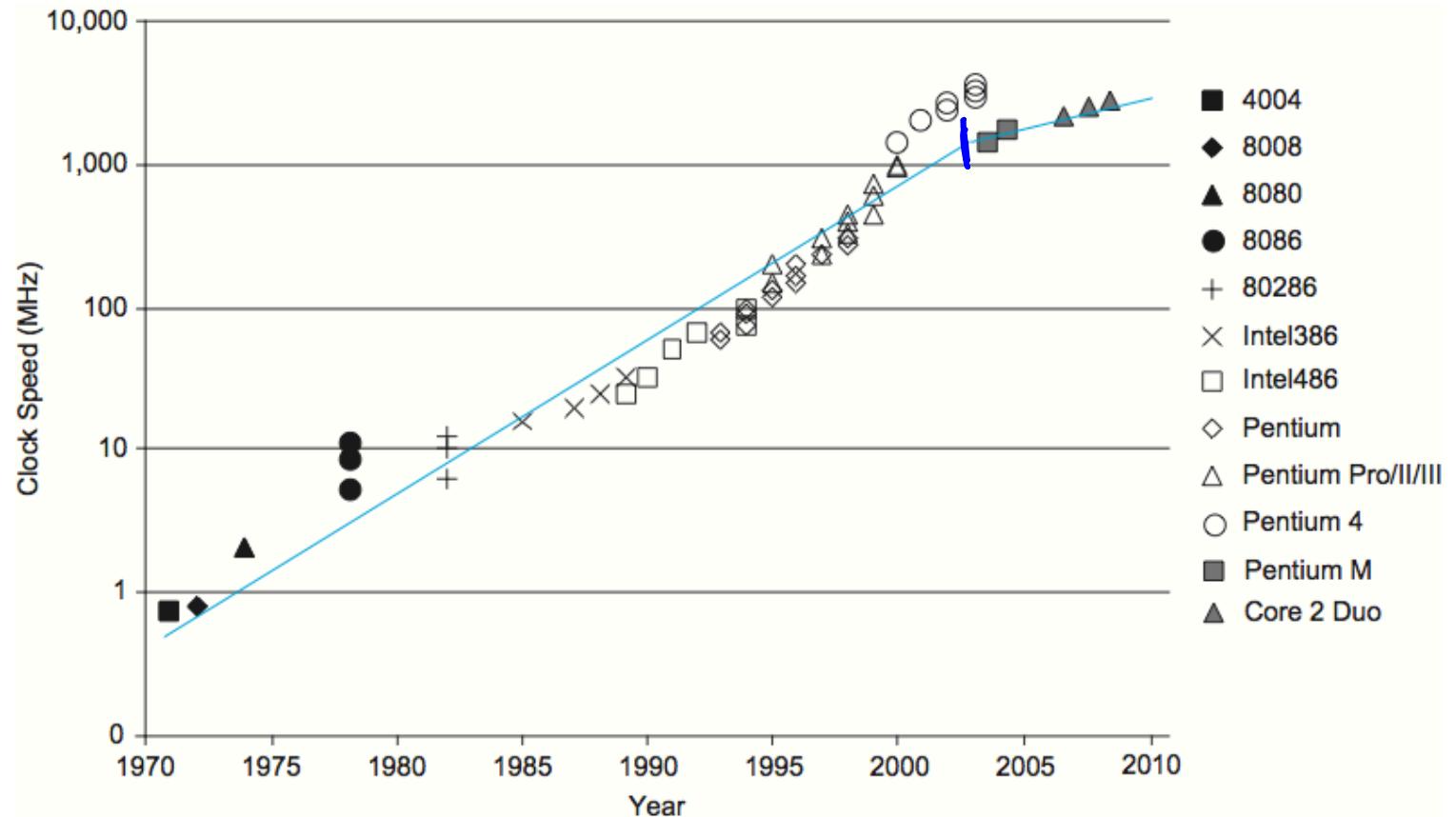
- 20 stages pipeline
- Up to 126 instructions executed in parallel

# Free speedup

- Moore's Law (1975):
  - Every 24 months, the number of transistors in a CPU can be doubled
- Number of transistors approximates performance
- If you have a program to run faster, just wait for next generation
  - It's free!

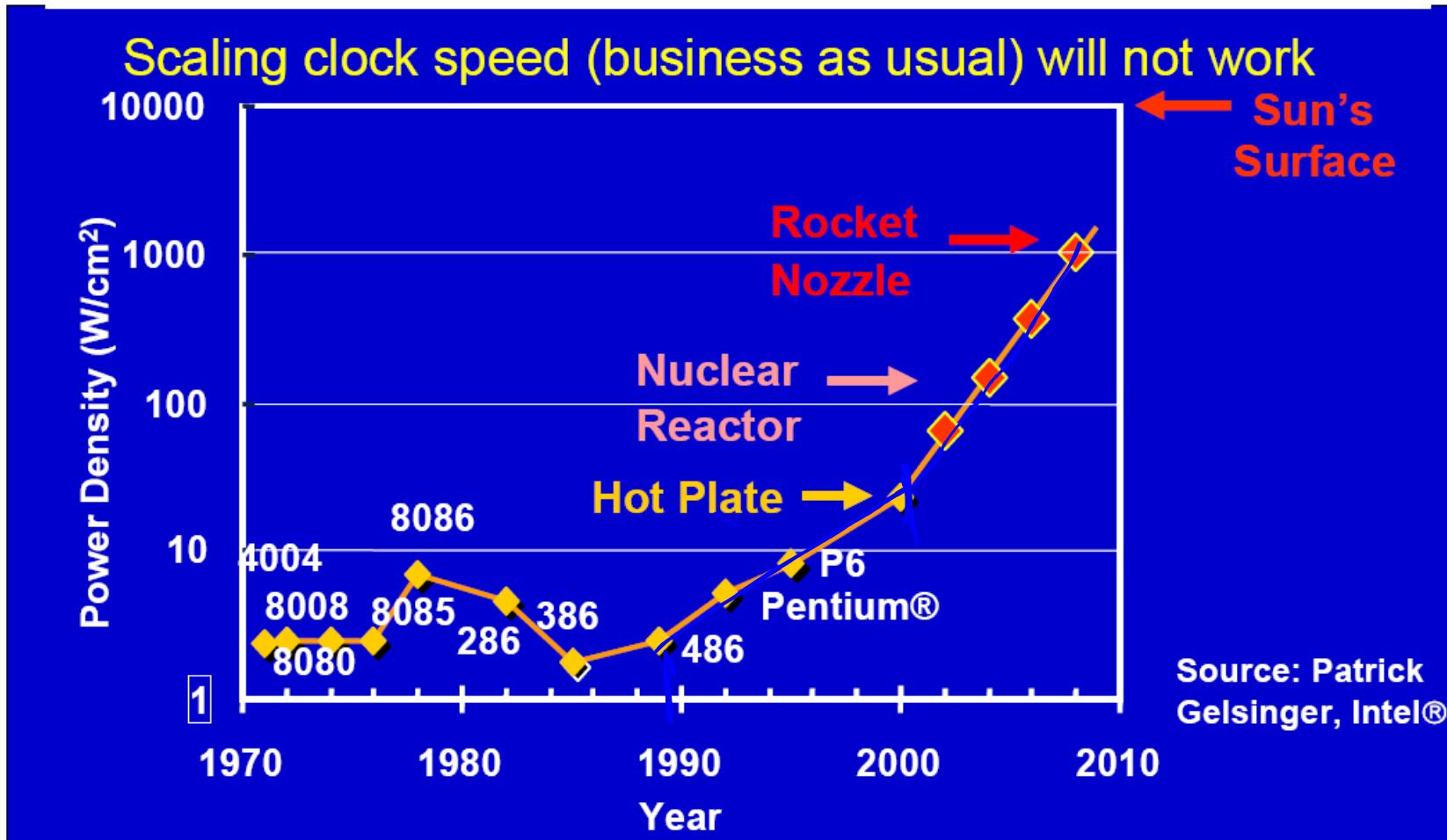
# Microprocessor Transistor Counts 1971-2011 & Moore's Law





**FIGURE 1.5** Clock frequencies of Intel microprocessors  
From : CMOS VLSI Design, Weste and Harris.

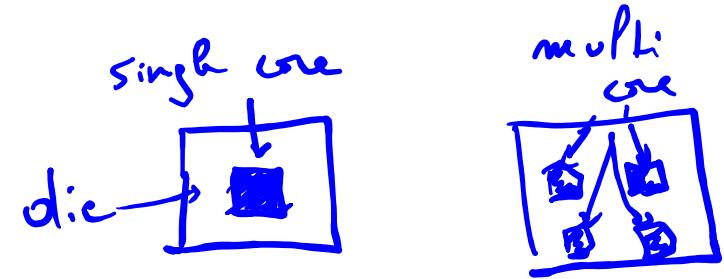
# Moore's Law limitations



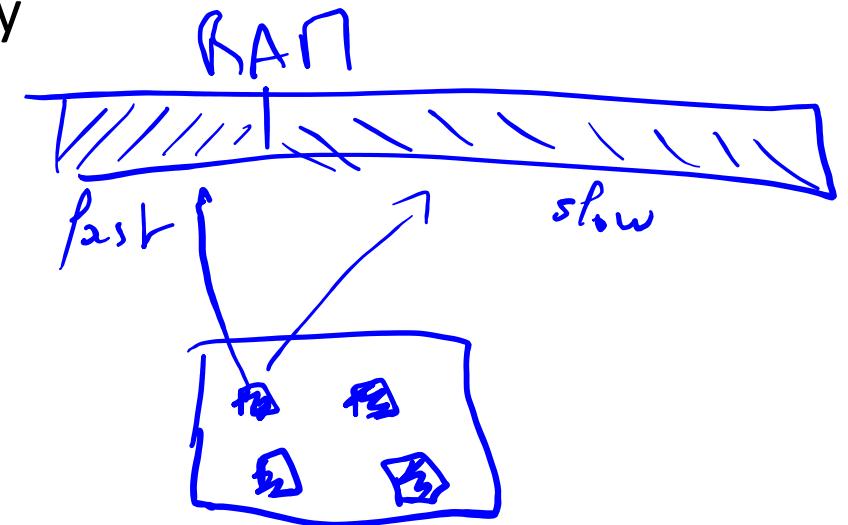
# Limits to Moore's Law

- Moore's Law is still valid
  - But no more free lunch
- What went wrong?
  - Physics did
- Limits on single-core
  - Smaller transistors
  - Or larger die
- Need to pack more CPUs together
  - Multi-core

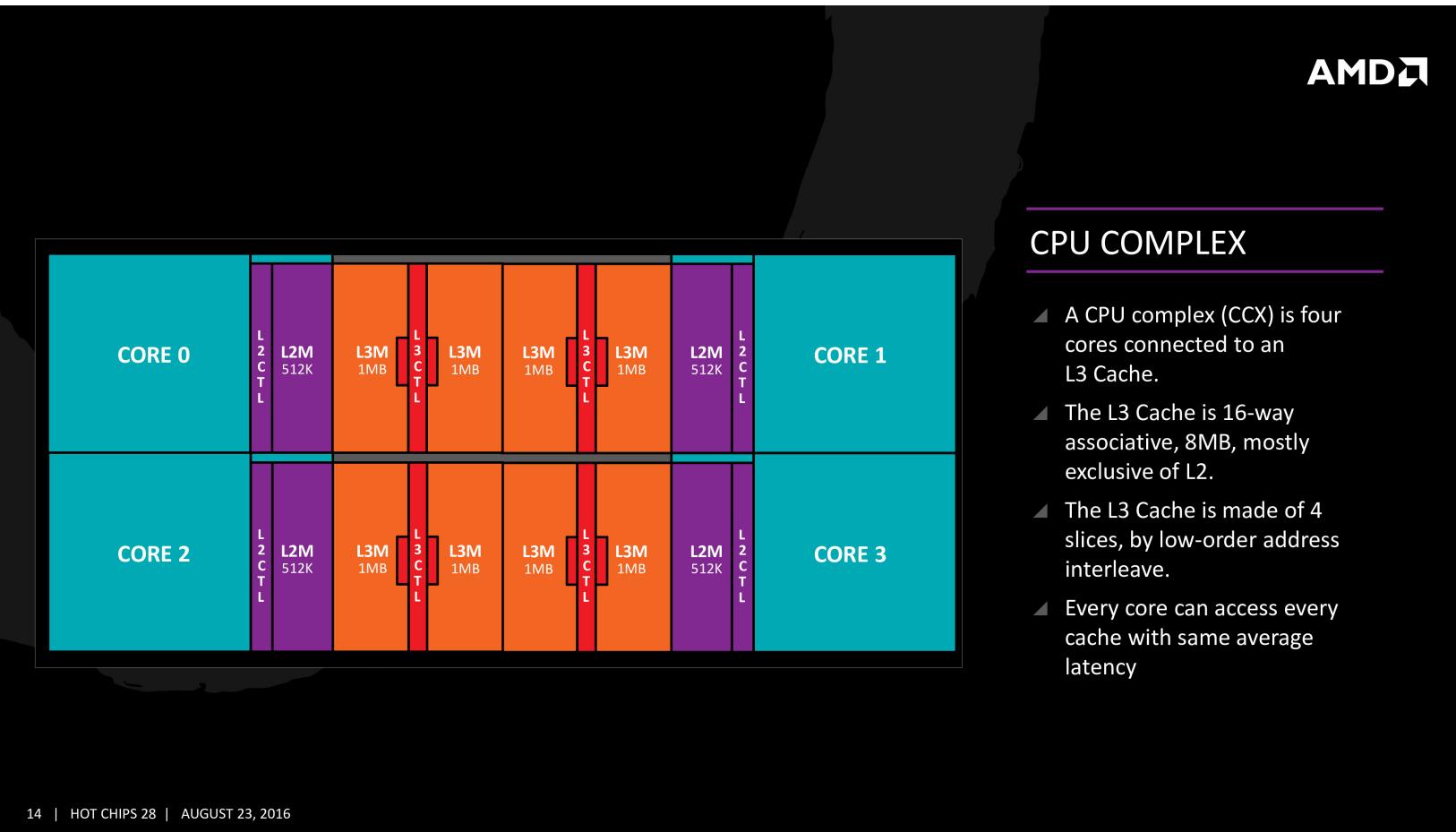
# Multi-core CPUs



- Multiple Processing Units (core) on the same die
  - Share some cache level
  - Have priority access to part of the physical memory
    - NUMA : Non Uniform Memory Access
- Core can be simple or complex
  - Complex cores form an MIMD machine
  - Simple cores usually SIMD



# AMD Ryzen 1 & 2

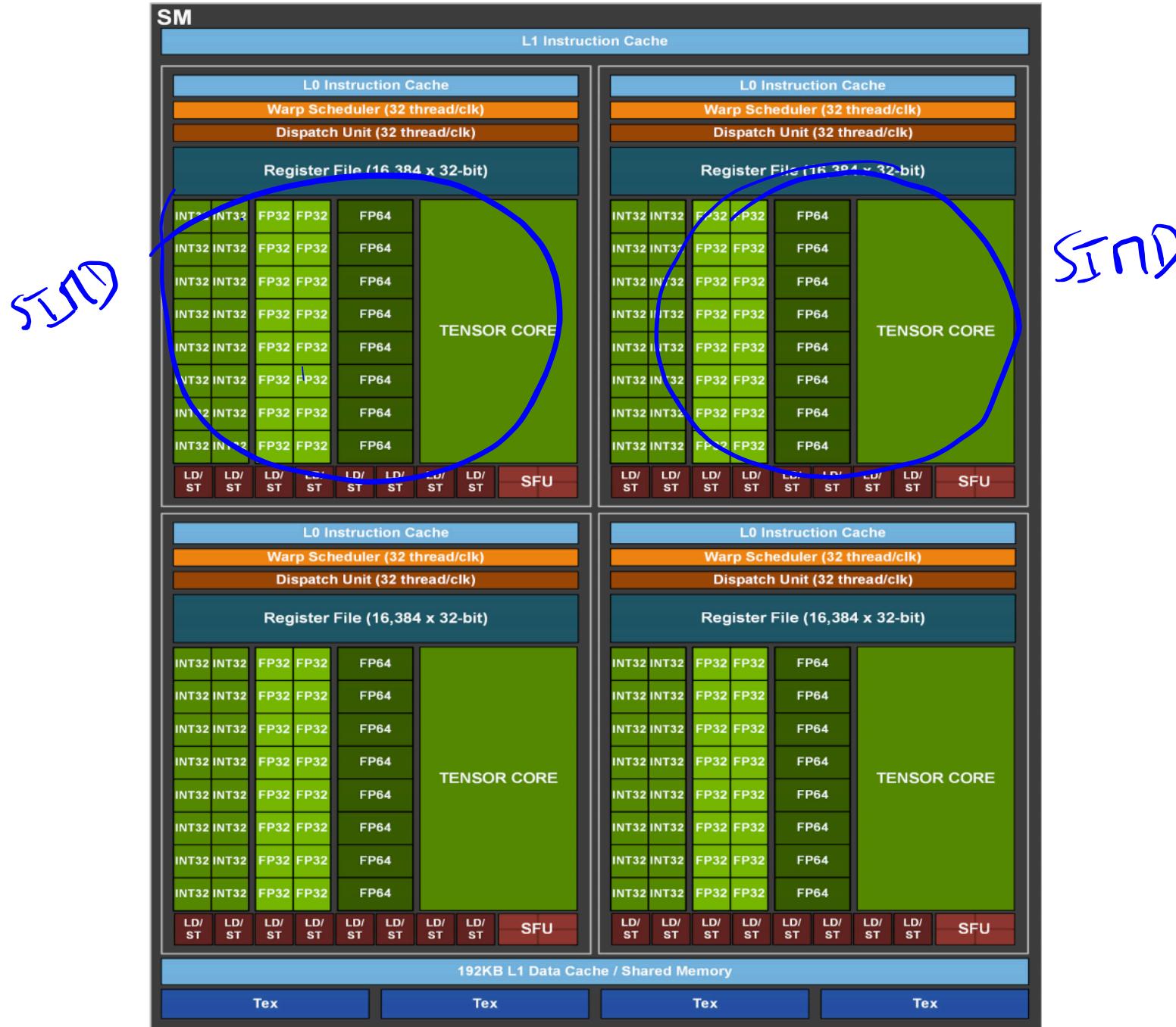


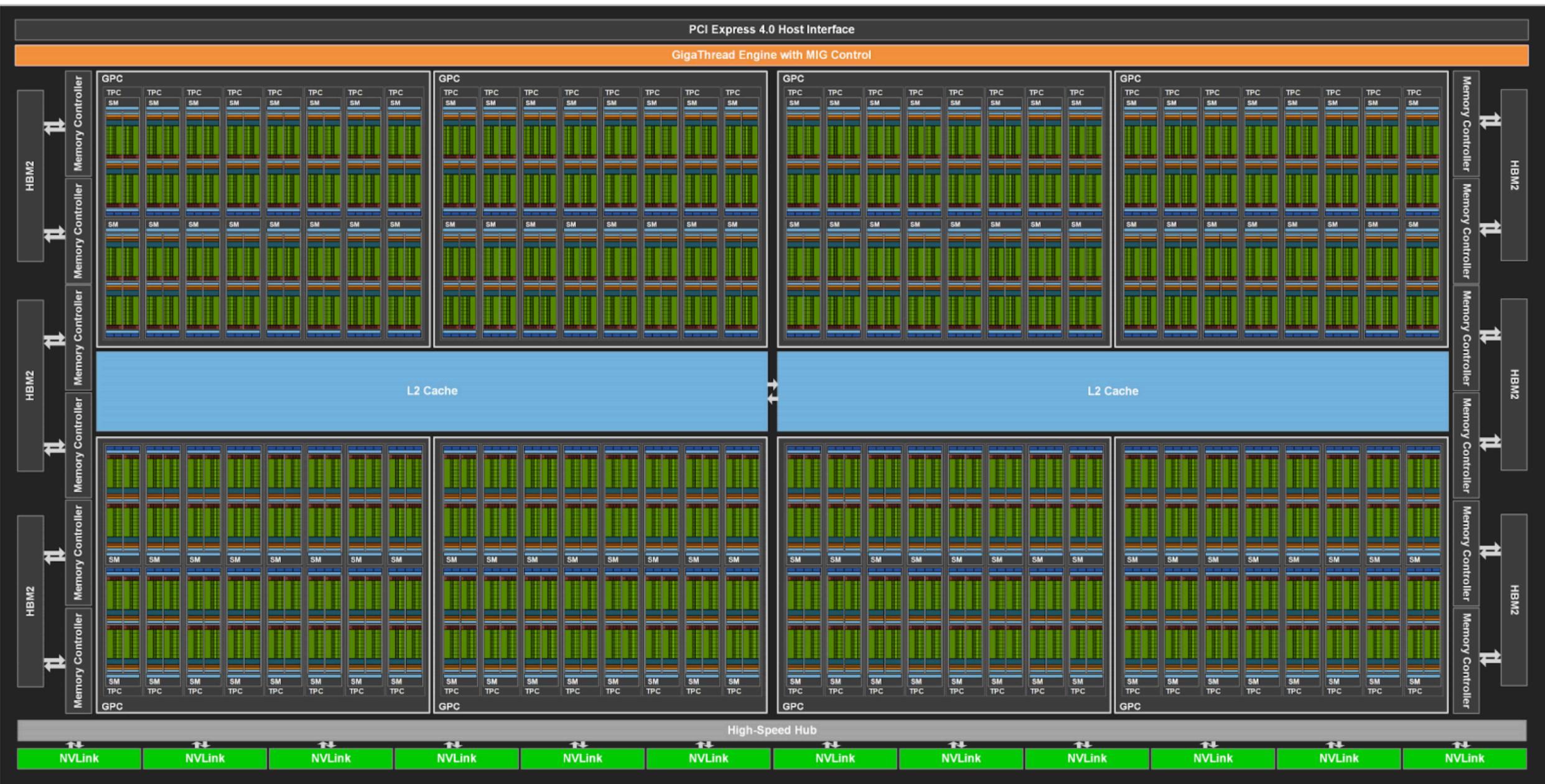
- Cores are grouped by 4 in a CCX
- Two CCX are grouped a CCD (Core Chiplet Die)
- Very modular architecture, increase CCD to have more cores
- Overall a MIMD

# GPGPU

- General Purpose Graphics Processing Unit (GPGPU)
  - Very efficient for FP computations
  - Massively parallel
- Hierarchical organization (NVIDIA)
  - GPU Processing Clusters
    - Streaming Multiprocessors
      - CUDA Cores
- Hybrid model
  - MIMD (Overall) & SIMD (SM)
- NVIDIA Ampere (GA100)
  - 128 Streaming Multiprocessors
  - 8192 Cuda Cores
  - 512 Tensor Cores → Matrix operations

Responsible for  
the rise of  
Deep Learning





# Software support

Processes and threads



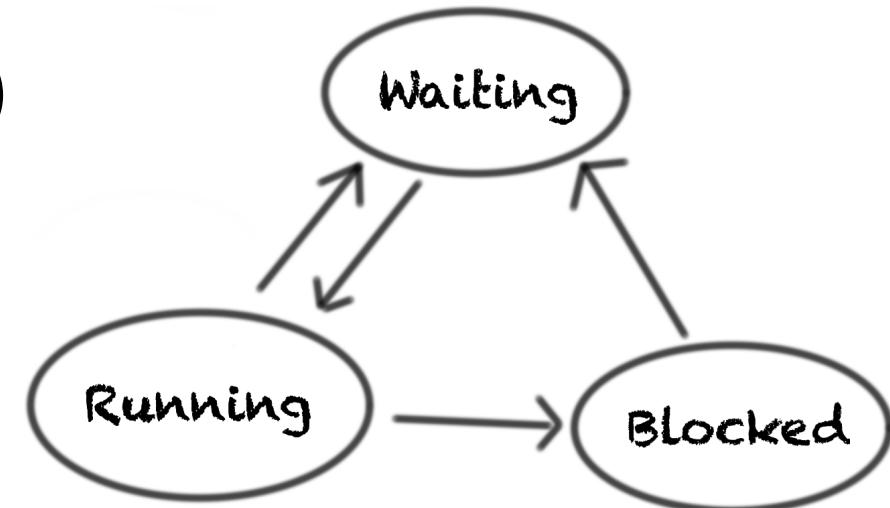
# Processes

- A process is a program being executed
- Created by the operating system (OS)
  - Linux
  - Windows
  - iOS/Android
- Contains all necessary information
  - Memory addresses used
  - Open files
  - Current instructions
- Provides isolation
  - Cannot see the memory of another process (SEGFAULT)
  - Sharing info between processes is a service of the OS
    - Inter Process Communication (IPC) through shared memory

e.g. copy paste

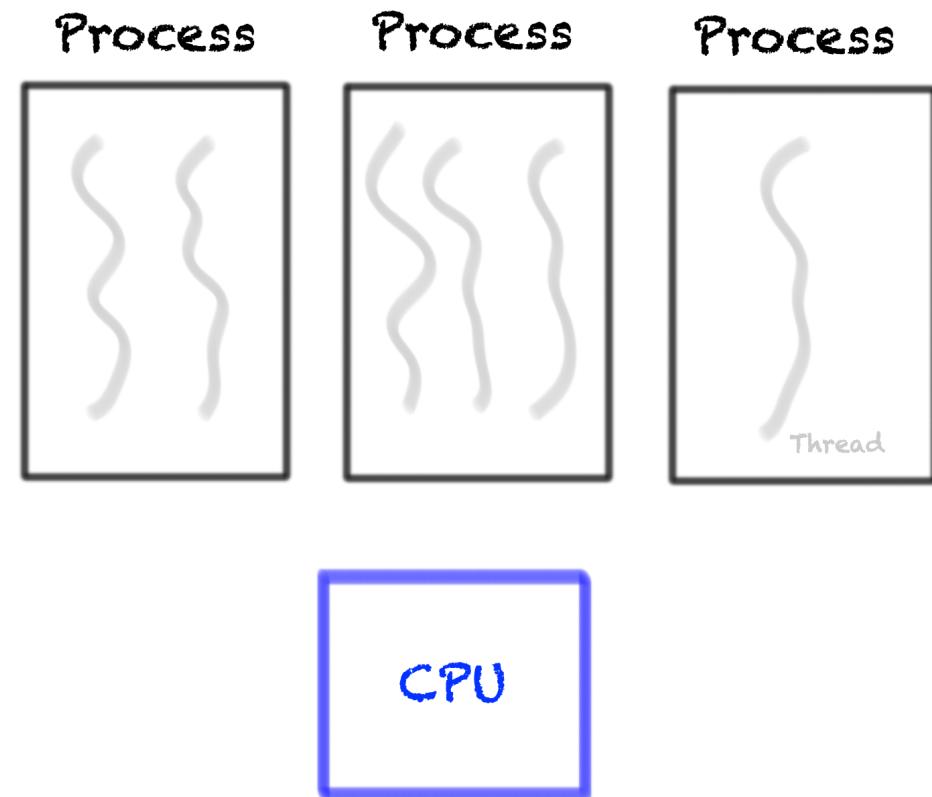
# Multi-tasking

- Processes share the CPU
  - The OS also needs the CPU
- Each process is given a timeslice (5-15ms)
  - Maximum duration on CPU
- A process loses the CPU if
  - It runs for the whole timeslice (waiting)
  - It does an I/O (blocked)  
*Slowwww!*
- The OS replaces the process on the CPU
  - Decides which process to run next (Scheduling)
  - Use of a queue with priority



# Context switch

- Changing which process runs is called context switch
- Context switches are costly
  - Lots of CPU register to change
  - Possible invalidation of caches
  - Execution of OS' scheduler
- Needs for a lighter alternative
  - Threads, aka lightweight processes.



# Threads

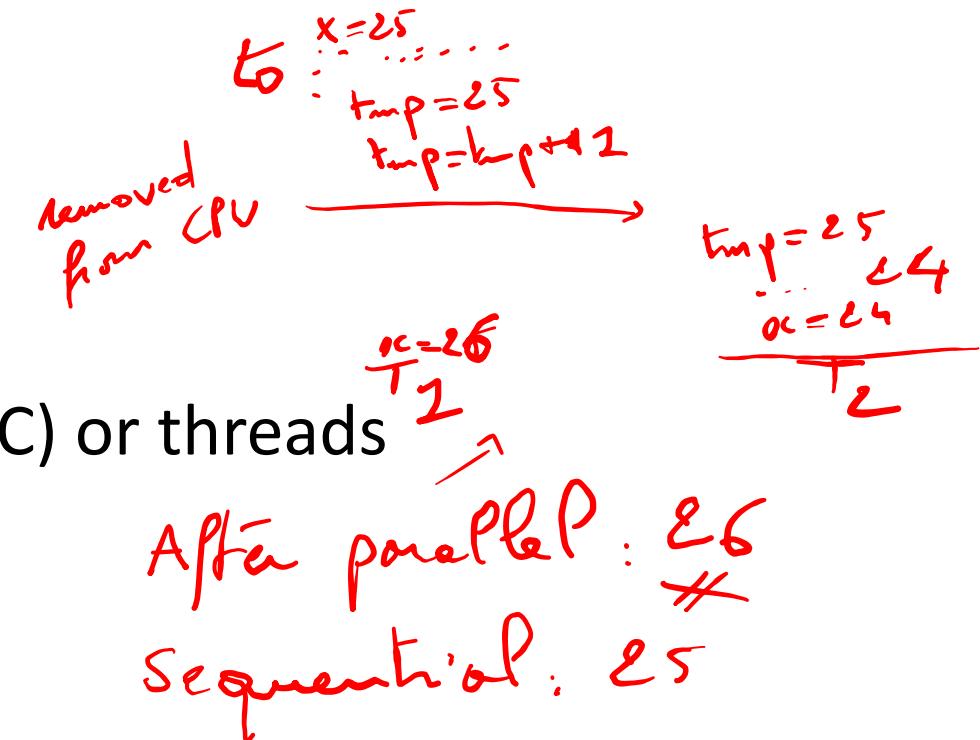
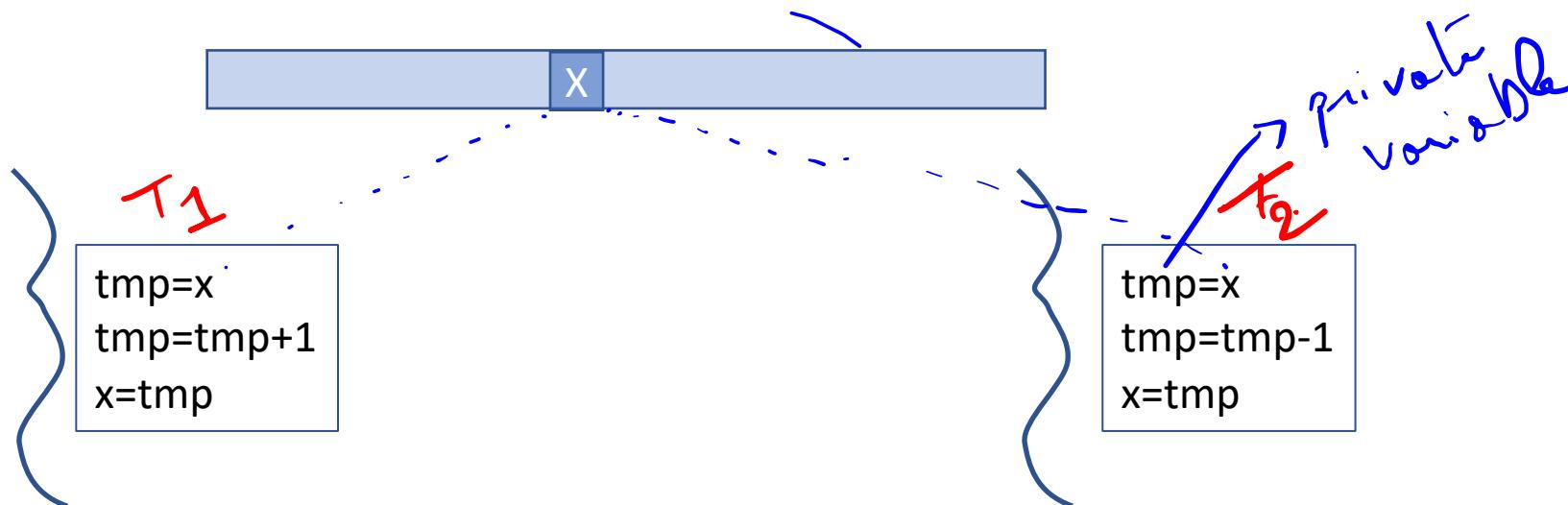
- A process contains multiple threads
- Threads of the same process share everything
  - Memory, open files...
  - Can have local (private) variables
- Scheduling another thread of the same process has very little cost
- Threads are provided by the OS
  - Usually accessible through APIs (pthreads in C, Thread in Java, threading in Python)

# Threads

- When and why use threads?
- Computational intensive application
  - Each thread computes part of the result
  - Use as many threads as core
  - Depends on the API used : not working in Python
- Applications with I/O
  - Use threads for performing I/O
  - Eg : app with GUI

# Race conditions

- Data can be shared between processes (IPC) or threads
  - What could go wrong?
- Example :
  - 2 threads using the same variable
  - 1 thread increasing it, the other decreasing it



# Protection against race conditions

- Avoid sharing!
  - Depends on your algorithm/problem
  - Easier with some programing models
- Protect the shared data
  - Notion of critical section (no 2 threads in this part of the code)
  - Test & set instructions
  - Low level tools : semaphores, monitors...

*the way to write  
the code*

# Deadlocks

2 or more process/thread which cannot process because they are waiting on each other

Example: lock(...): take a variable for exclusive access  
unlock(...) or wait until it's available

Case 1

incl:

lock(a)

...+  
unlock(x)

deck)

lock(x)

- - - → |

unlock(x)

Case 2

lock(a)

lock(b)

a+b

unlock(b)

unlock(a)

lock(b)

lock(a)

a-b

...

.

# Software support

Middleware and libraries

# Low level vs High level

- Low vs High
  - Usually how close you are to the hardware
- Example : C (low) vs Python (High)
- In parallel/distributed systems
  - Low : you manage threads, semaphores, sockets...
  - High : abstractions hide the low level details
- As data scientists
  - Know that low level exists
  - Focus on high level

# Libraries

- Libraries are high level tools
  - Provide function, methods to use in your code
- Example :
  - TensorFlow, Pandas
  - PyTorch : scientific computing with or without GPGPU

```
if torch.cuda.is_available():
    dev = "cuda:0"
else:
    dev = "cpu"

device = torch.device(dev)
a = torch.zeros(4,3)
a = a.to(device)
```

<https://medium.com/ai%C2%B3-theory-practice-business/use-gpu-in-your-pytorch-code-676a67faed09>

# Frameworks

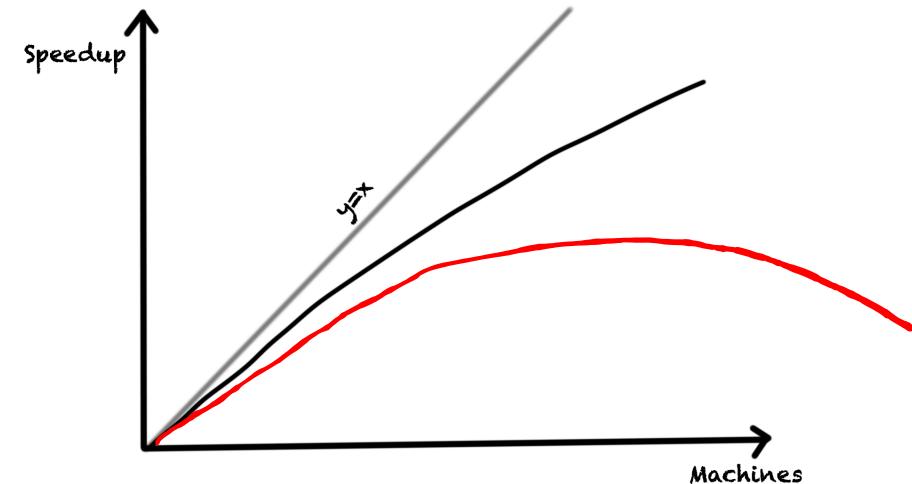
- Frameworks are also high level
  - Provides some default functionalities
  - Your code adds/modifies functionalities
  - The framework uses your code
- Example :
  - Hadoop, Spark, Flink...
- Frameworks are very common in distributed systems

# Concepts of Distributed Systems

Scalability

# Speedup

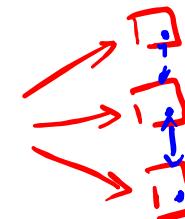
- Measure of the relative performance
  - Sequential vs parallel
- Example
  - $T_s = 200$  seconds
  - $T_p = 100$  seconds on 2 cores/machines
  - Speedup = 2 (i.e the parallel version is twice as fast as the sequential one)
- Speedup varies
  - With the problem
  - With the number of resources



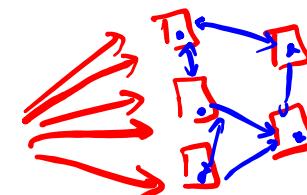
# Scalability

- Capacity of a system to manage varying load through adaptation
  - Add (resp. remove) resources when load increases (resp. decreases)
- Scaling up (resp. down)
  - Increase the resources of a single machine
    - Add memory, increase CPU frequency...
  - Limited and complicated but transparent to the application
- Scaling out (resp. in)
  - Add more machines
  - Easy if the application is stateless.

$T_0$  :



$T_1$  :



# Auto vs Manual scaling

- Manual scaling
  - Decided by the user or administrator
  - Could be based on expected load (e.g. time of day)
  - Slow, error prone
- Auto scaling
  - The application (framework) decides based on measured metrics
  - Fast but can have unexpected results

# Concepts of Distributed Systems

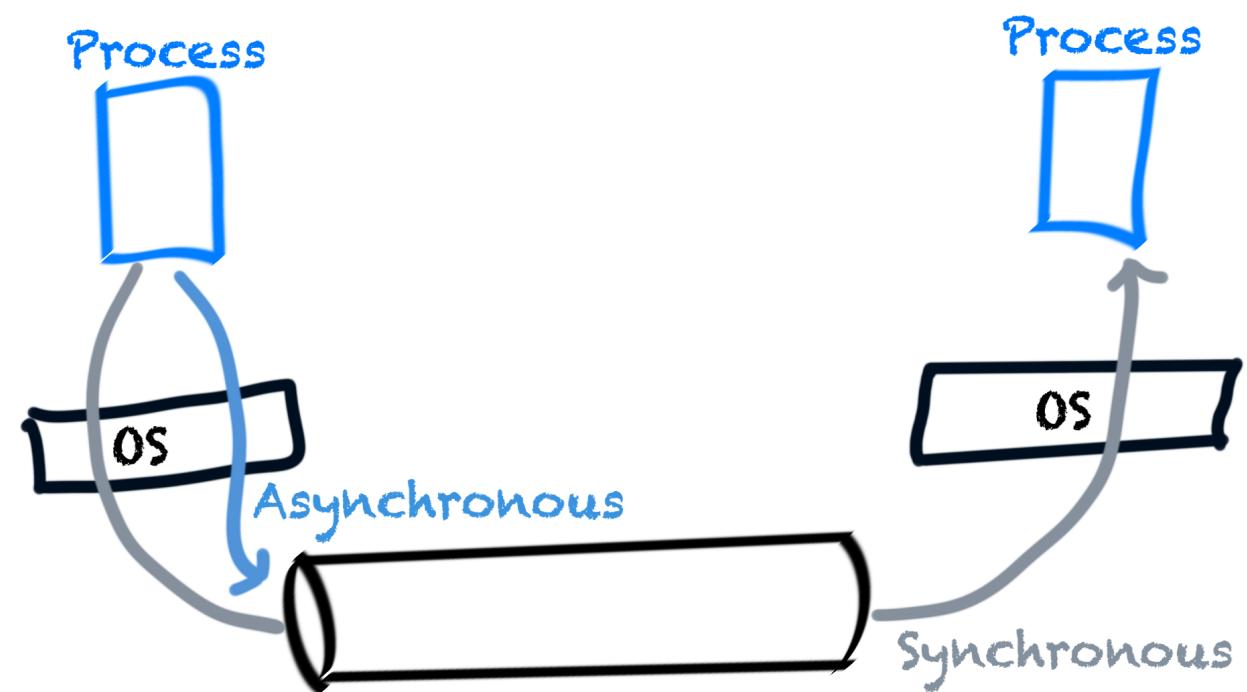
Communications

# Communication

- Low level : sockets
- End points for sending/receiving data
  - FIFO, handle bytes
- Needs IP address and port of remote machine
  - Lack of transparency
- Sending complex data structures requires transformation
  - Serialization/Deserialization
- High level : messages, Remote Procedure Call (RPC)

# Synchronous vs Asynchronous

- Synchronous
  - Sender is blocked until message is processed by receiver
- Asynchronous
  - Sender is blocked during sending
  - Usually relies on a message queue



# Latency

- Any communication takes time
  - Base latency + bandwidth from network
  - Multiple software layers
- Avoid them if possible
  - But latency does not come only from network
  - Even memory access add latency

# Latency Comparison Numbers

fun ↗

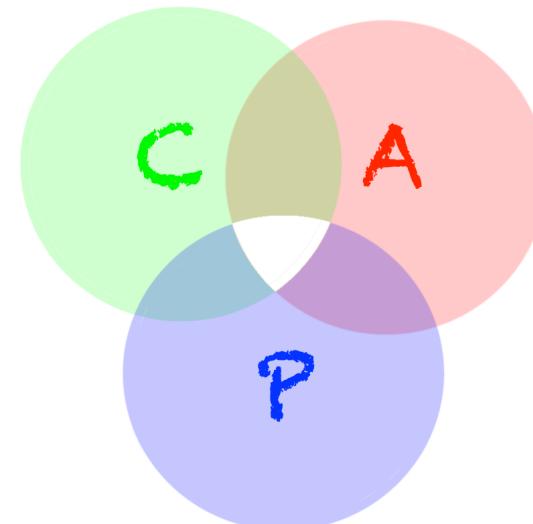
L1 cache reference .....	0.5 ns
Branch mispredict .....	5 ns
L2 cache reference .....	7 ns
Mutex lock/unlock .....	25 ns
Main memory reference .....	100 ns
Compress 1K bytes with Zippy .....	3,000 ns = 3 µs
Send 2K bytes over 1 Gbps network .....	20,000 ns = 20 µs
SSD random read .....	150,000 ns = 150 µs
Read 1 MB sequentially from memory .....	250,000 ns = 250 µs
Round trip within same datacenter .....	500,000 ns = 0.5 ms
Read 1 MB sequentially from SSD* .....	1,000,000 ns = 1 ms
Disk seek .....	10,000,000 ns = 10 ms
Read 1 MB sequentially from disk ....	20,000,000 ns = 20 ms
Send packet CA->Netherlands->CA ....	150,000,000 ns = 150 ms

# Concepts of Distributed Systems

Limits to distributed systems

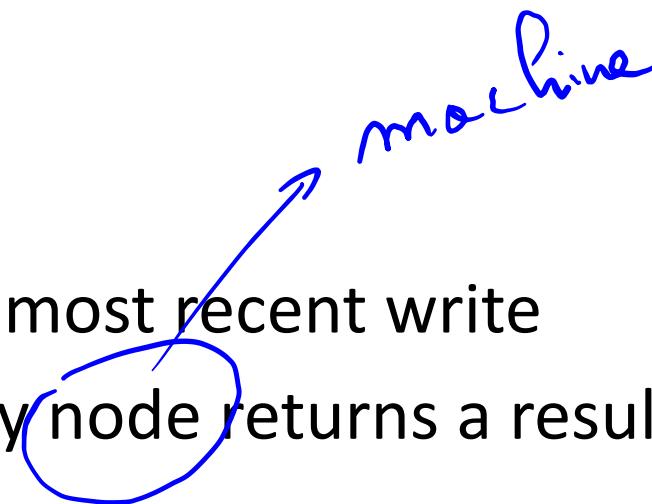
# CAP theorem

- Theorem from Eric Brewer (2000)
- A distributed system with read/write cannot provide more than 2 of the following guarantees
  - Consistency
  - Availability
  - Partition Tolerance



# CAP theorem

- Consistency : a read returns the most recent write
- Availability : every request to any node returns a result (not necessarily correct)
  - Partitions WILL occur
- Partition Tolerance : the system still works in presence of network failures.
- In practice, the network is NOT reliable
  - Partitions WILL occur



# CAP theorem

- So CAP theorem can be rephrased
  - In presence of network failure, you have to choose between Availability and Consistency
- A-P : When there is a partition, the system continues to work but might answer outdated/inconsistent results. (e.g Twitter, Facebook)
- C-P : When there is partition, some nodes might stop answering.

↳ Database for bank

# Failures

- Software can crash
- The network is not reliable
- But so are the machines!
  - A single machine can be reliable
  - But 10s of them?

# Typical first year for a new google cluster

- ~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
- ~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
- ~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
- ~1 network rewiring (rolling ~5% of machines down over 2-day span)
- ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
- ~5 racks go wonky (40-80 machines see 50% packetloss)
- ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
- ~12 router reloads (takes out DNS and external vips for a couple minutes)
- ~3 router failures (have to immediately pull traffic for an hour)
- ~dozens of minor 30-second blips for dns
- ~1000 individual machine failures
- ~thousands of hard drive failures
- slow disks, bad memory, misconfigured machines, flaky machines, etc.
- Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

# Conclusion

Take home message

# Conclusion

- Parallelism requires specific hardware
- Modern hardware is parallel
  - Multicore
  - GPGPUs
- High level libraries/frameworks hide the complexity