Computer Architecture

MIT Department of Electrical Engineering and Computer Science

Announcements

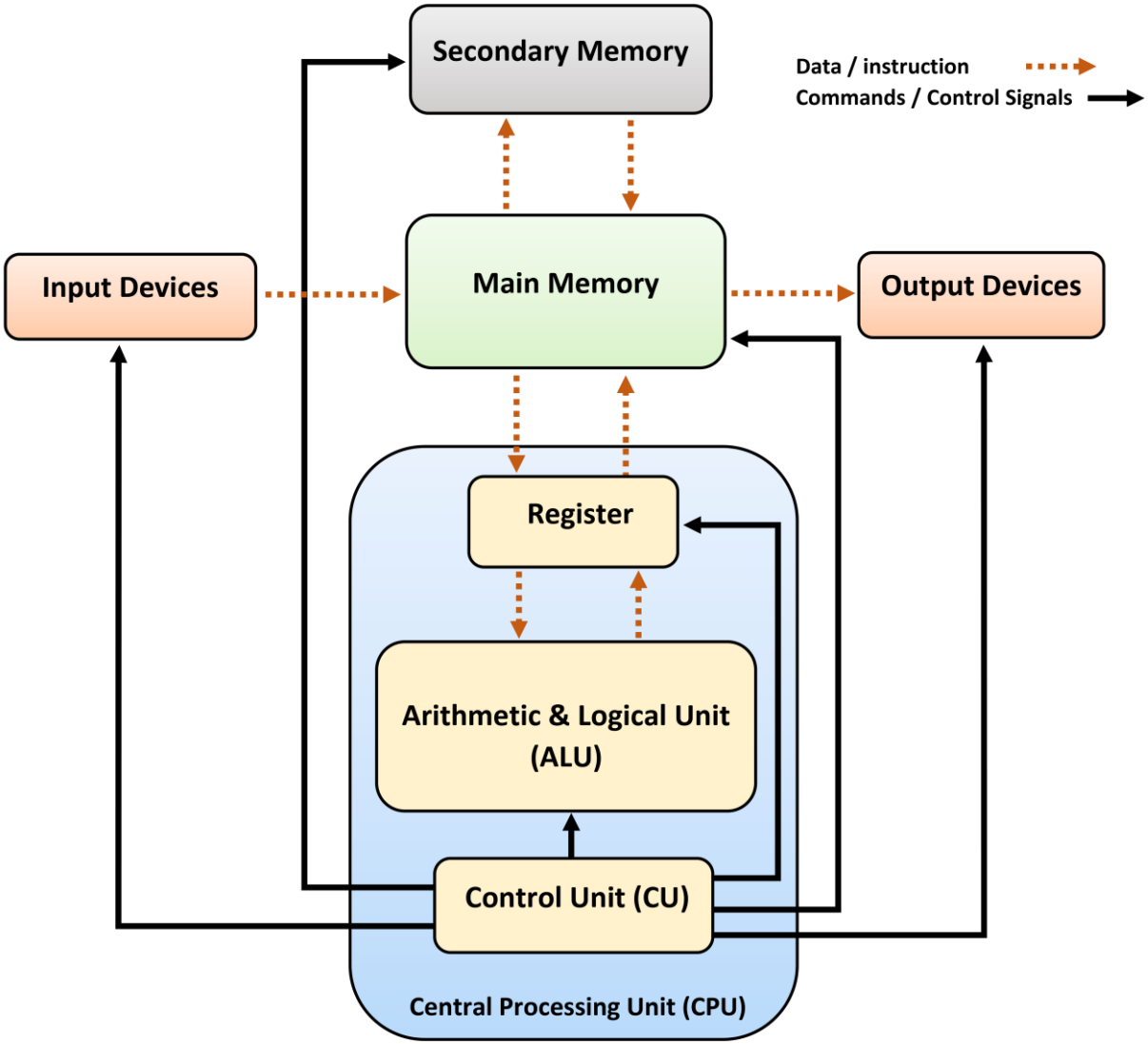
- Pset 6 checkoff is due tomorrow evening
- Tomorrow is the last day of regular office hours
- Andrew will have office hours this Thursday 1–3 pm in 38-370 (usual office hours room, not his office)

- Please fill out course evaluations by Monday, Dec 15 at 9 am!
  - https://eduapps.mit.edu/subjeval/studenthome.htm
  - Current response rate is 11%

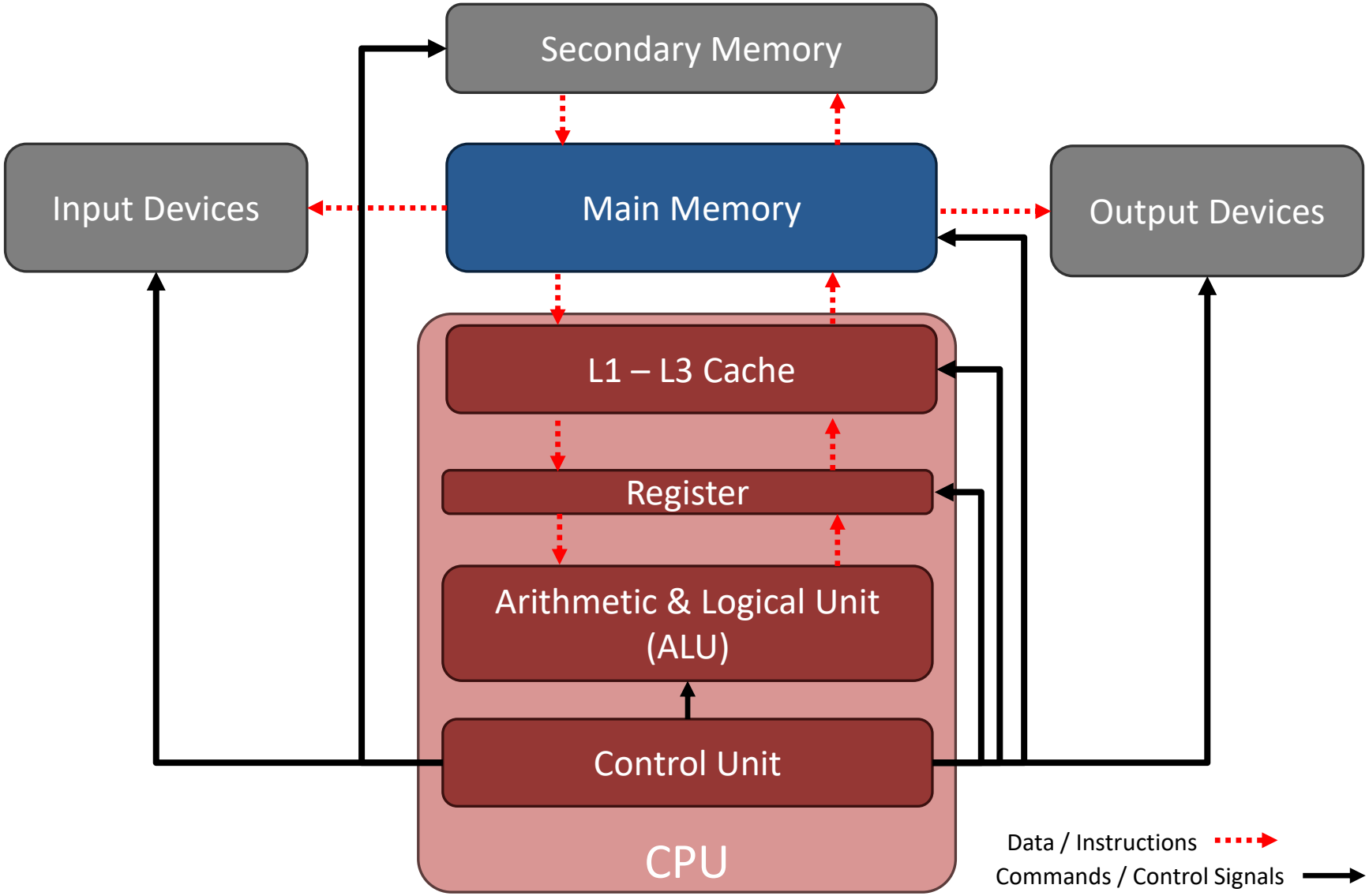
Today's Lecture

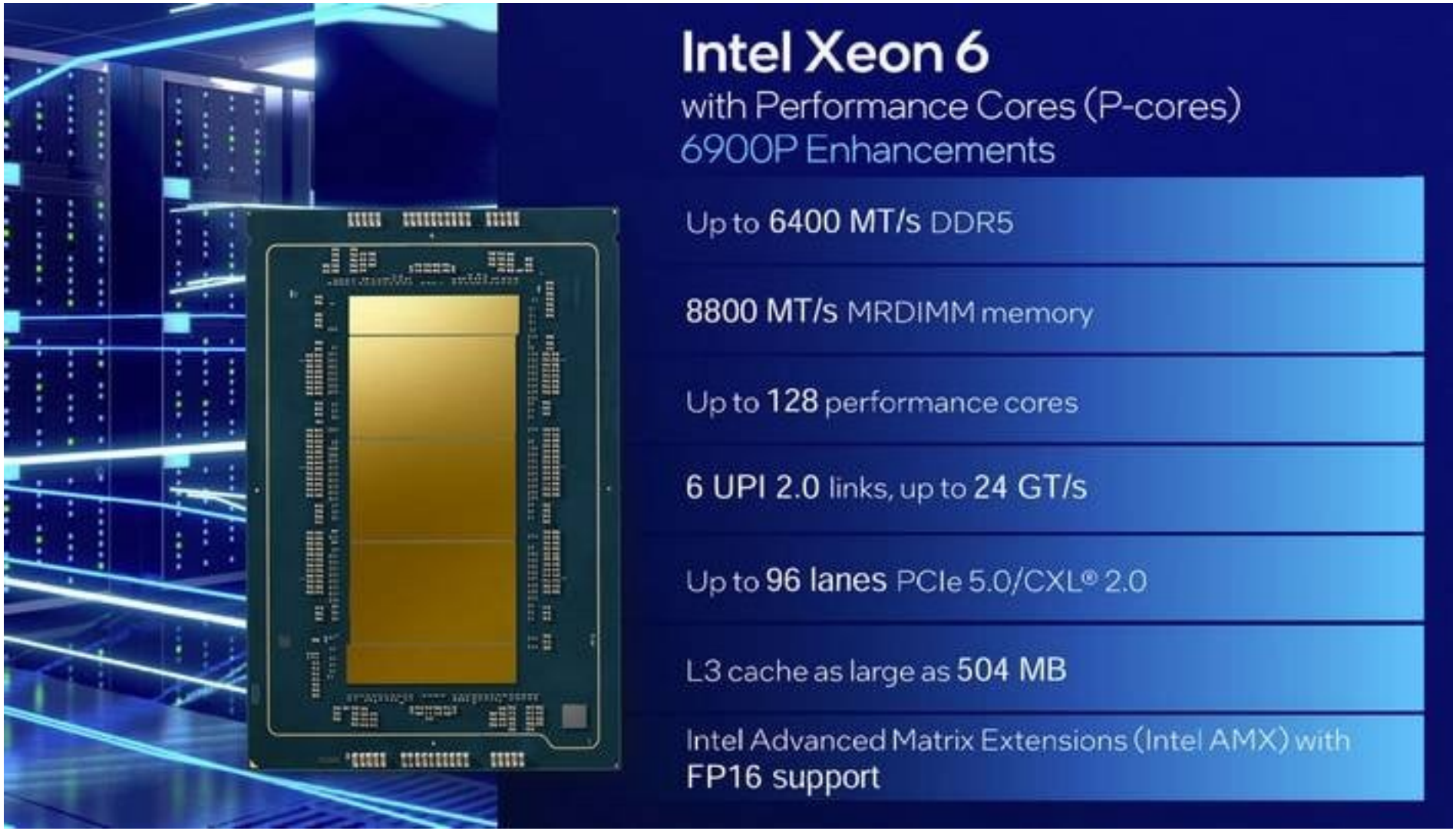
- Computer Architecture
- Efficiency of Python

Conceptual Computer Architecture

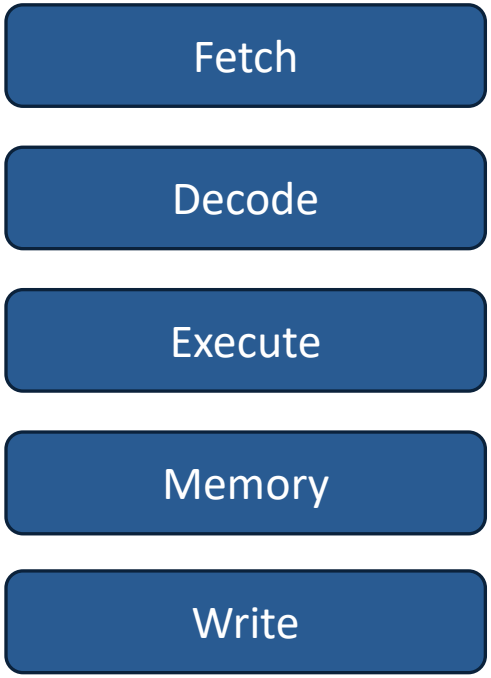


Conceptual Computer Architecture




Intel Xeon 6
with Performance Cores (P-cores)
6900P Enhancements
Up to 6400 MT/s DDR5
8800 MT/s MRDIMM memory
Up to 128 performance cores
6 UPI 2.0 links, up to 24 GT/s
Up to 96 lanes PCIe 5.0/CXL® 2.0
L3 cache as large as 504 MB
Intel Advanced Matrix Extensions (Intel AMX) with FP16 support

Phases of a single CPU instruction



Fetch data from memory/cache into registers

Decode instruction (e.g., add/multiply/etc)

Execute instruction

Memory Access

Write result back to memory

Every stage takes typically 1 cycle
but some operations might take more (e.g., multiply)

Note: The stages vary from CPU to CPU and sometimes more or less stages are used to explain the pipeline concept

Example Instruction ADD

1. IF (Instruction Fetch)

- Action: The instruction is fetched from Instruction Memory using the address stored in the Program Counter (PC).
- Registers Used: The instruction is stored in the Instruction Register (IR). The PC is automatically incremented to point to the next instruction in sequence.

2. ID (Instruction Decode and Register Fetch)

- Action: The instruction in the IR register is decoded by the Control Unit to determine it's an ADD operation. Simultaneously, the values of the two source registers are read from the Register File and placed into temporary pipeline registers, preparing them for the next stage.

3. EX (Execute)

- Action: The Arithmetic Logic Unit (ALU) performs the addition adding the two values from the register. Result is stored in ALU Output Register

4. MEM (Memory Access)

- Action: For an ADD instruction this is a no-op

5. WB (Write-Back)

- Action: The final result of the addition, which was calculated in the EX stage, is written back to the Register File into the specified destination register. Result becomes visible

Example Instruction LOAD

1. IF (Instruction Fetch)

- Action: The instruction is fetched from Instruction Memory using the address stored in the Program Counter (PC).
- Registers Used: The instruction is stored in the Instruction Register (IR). The PC is automatically incremented to point to the next instruction in sequence.

2. ID (Instruction Decode and Register Fetch)

- Action: The instruction is decoded. The value of the base register is read from the Register File. The offset value is extracted and sign-extended to 32 bits, preparing it for address calculation.

3. EX (Execute)

- Action: The **Arithmetic Logic Unit (ALU)** is used to calculate the final, **effective memory address**. Effective Address = Rs + offset

4. MEM (Memory Access)

- Action: The CPU sends the Effective Address to the Data Memory (or cache) and issues a read signal. The data stored at that exact address is read back.

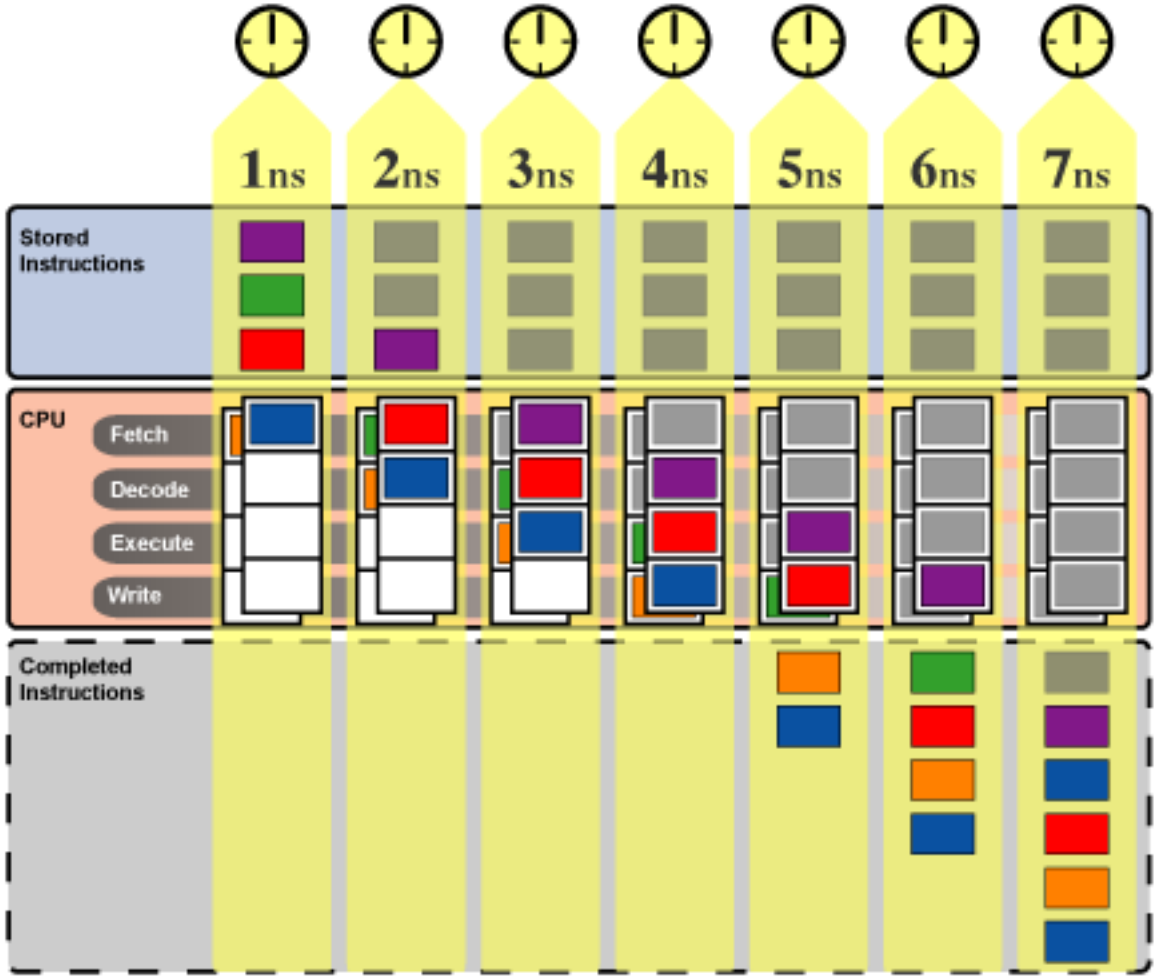
5. WB (Write-Back)

- Action: The data value retrieved from memory in the MEM stage is written into the destination register Rt in the Register File.

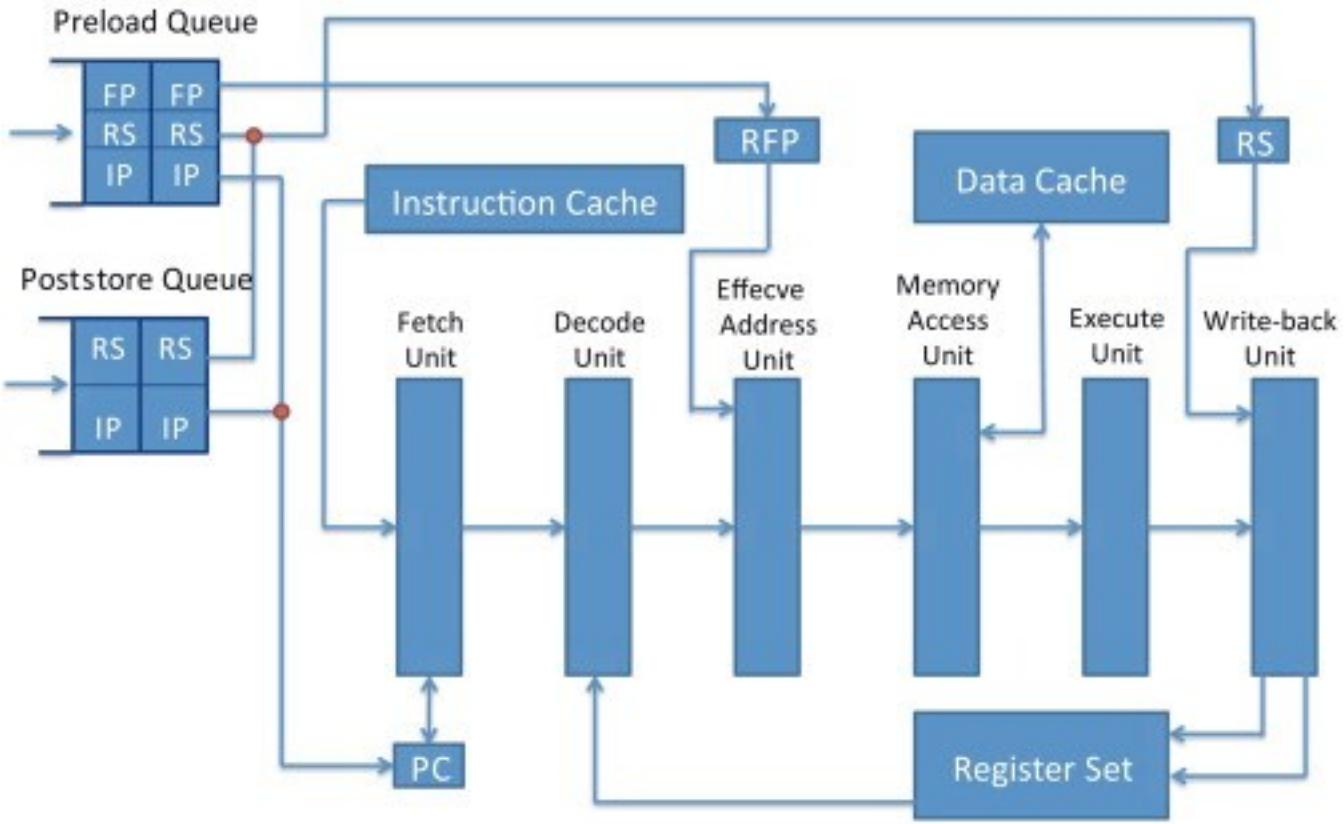
How many instruction can a core process?

- *Every instruction takes roughly 5 steps*
- *Every stage takes typically 1 cycle*
  but some operations might take more (e.g., multiply)
- *One cycle on a modern CPU with 2GHz takes roughly* $T = 1/f = 1 / 2\ GHz = 0.5\ ns = 0.5 * 10^{-9}$ *seconds*
- *In other words, a single core (theoretically) can roughly do 0.8 billion instructions per second.*

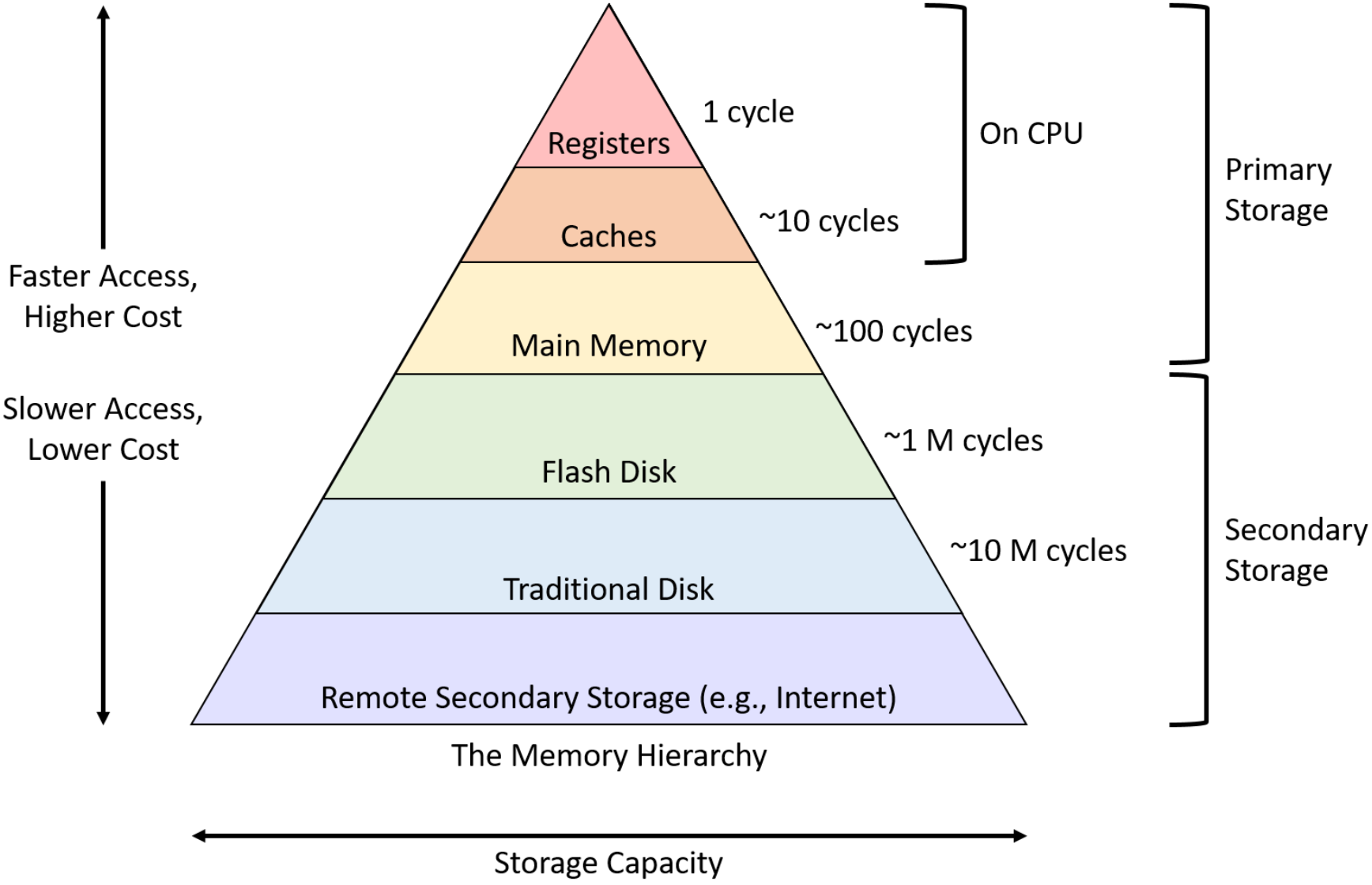
Can you think of a technique to do even more instructions per second?


1ns
2ns
3ns
4ns
5ns
6ns
7ns
Stored Instructions
CPU
Fetch
Decode
Execute
Write
Completed Instructions

A different view



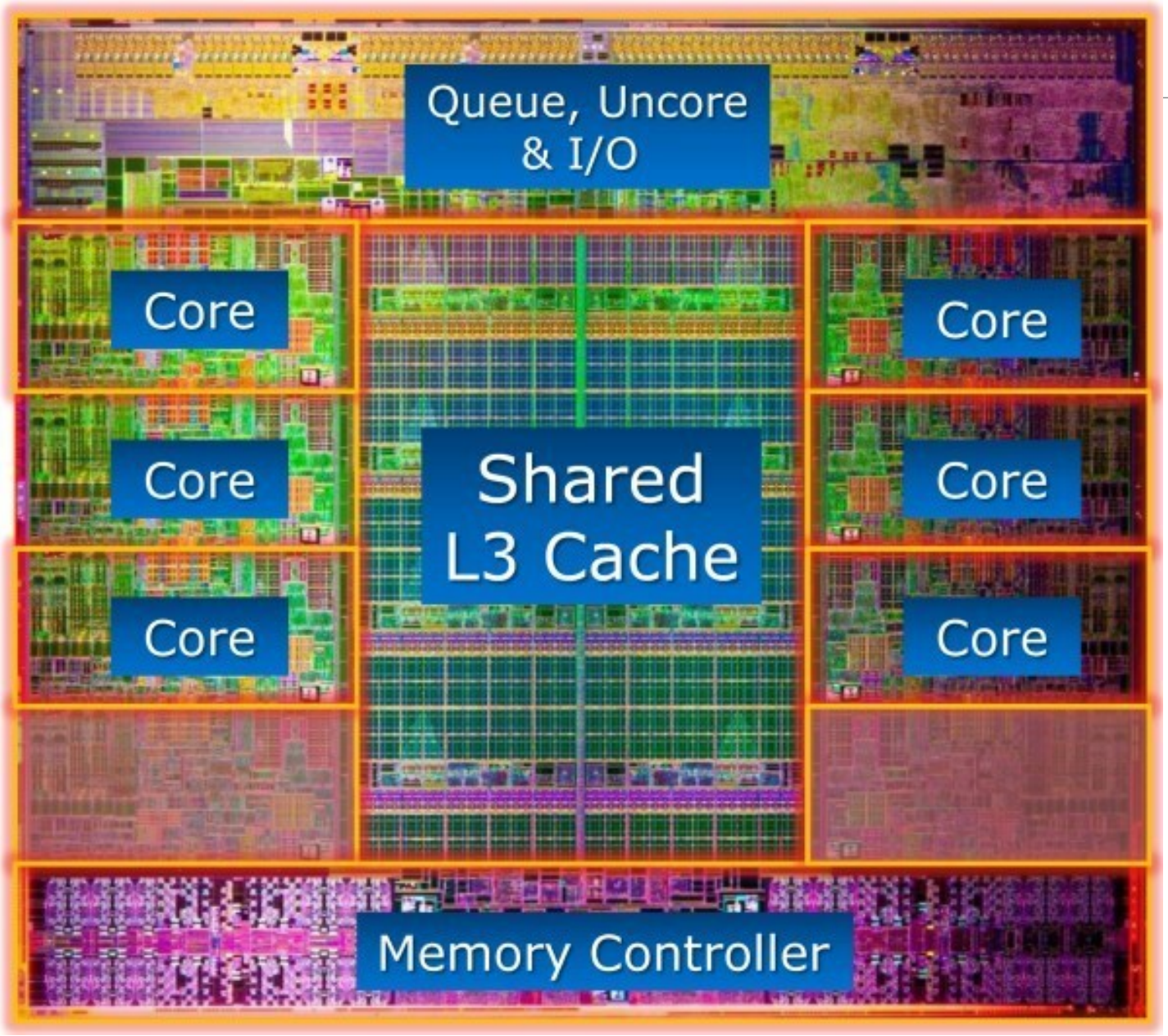
Storage Latencies

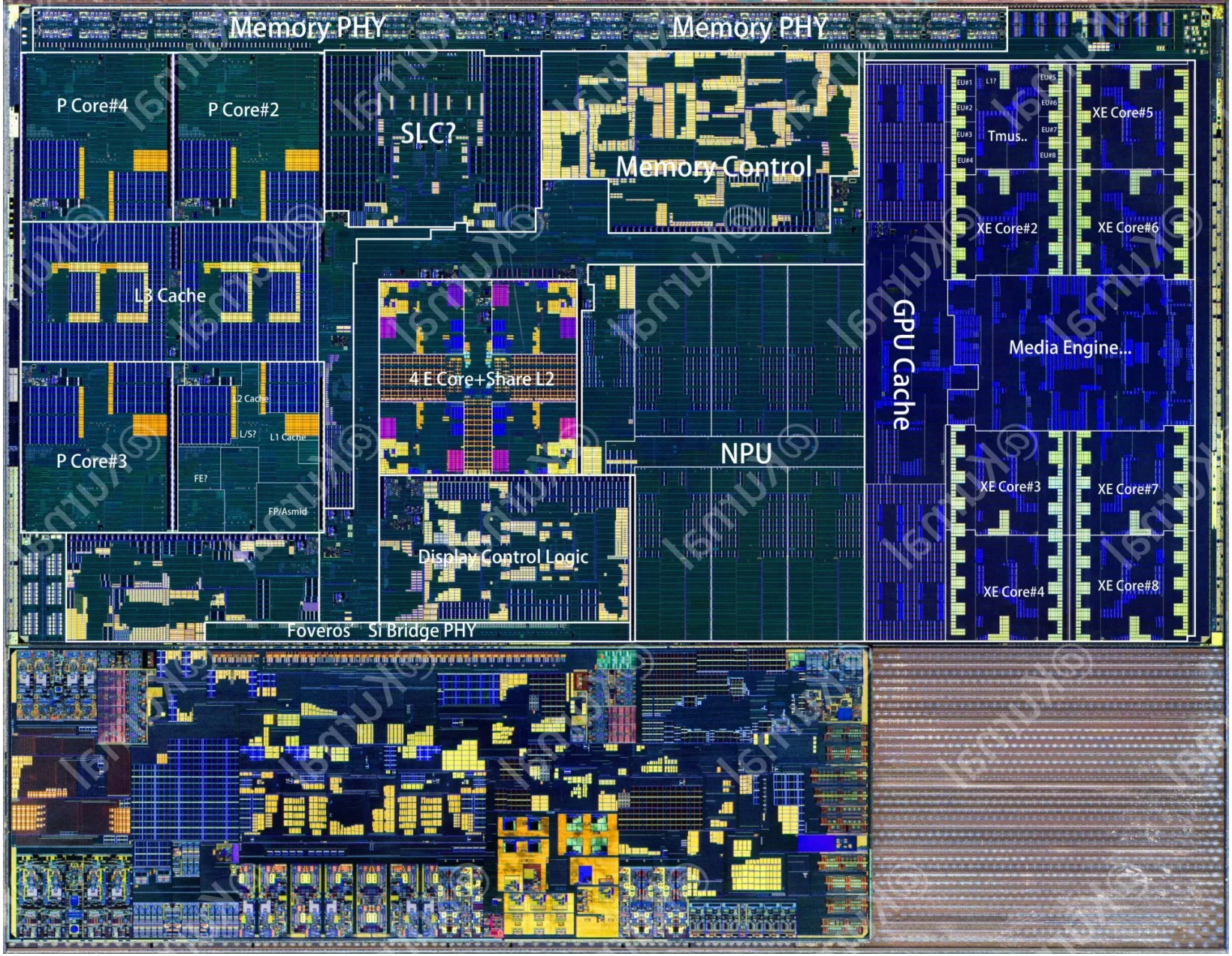


L1 to HDD

Component	**Typical Latency Range**	**Typical Throughput Range**	Typical Capacity Range	**Notes**
Registers	0 cycles	Nn/a	A few hundred Bytes	
L1 Cache	0.5–1 ns	50–100 GB/s	32 KB to 128 KB per core	Fastest; typically divided into Instruction/Data cache.
L2 Cache	3–15 ns	10–50 GB/s	256 KB to 1 MB per core	Larger than L1; may be private or shared per core.
L3 Cache	10–40 ns	10–30 GB/s	8 MB to 64 MB (shared)	Largest cache; shared across all CPU cores.
Main Memory (RAM)	50–100 ns	20–400 GB/s	8 GB to 128 GB	Latency is ~100x that of L1 cache. Throughput depends on memory channels (e.g., DDR5).
NVMe SSD	20–100 µs	1–13 GB/s	256 GB to 8 TB	Connects via PCIe, offering superior performance.
SATA SSD	50–500 µs	300–600 MB/s	128 GB to 4 TB	Limited by the older SATA interface.
HDD	1–10 ms	50–200 MB/s	500 GB to 20 TB	Latency dominated by mechanical seek time; several orders of magnitude slower than SSDs.
Local Network	0.1–1 ms	100 Mi Megabits/s to 80 Gigabits/s	n/a	Note: Gigabits not bytes. Local Network can be faster than disk, particular with technology like InfiniBand and RDMA
Long-Distance Network	10–100 ms	Varies significantly	n/a	Latency is dominated by the physical distance (speed of light).

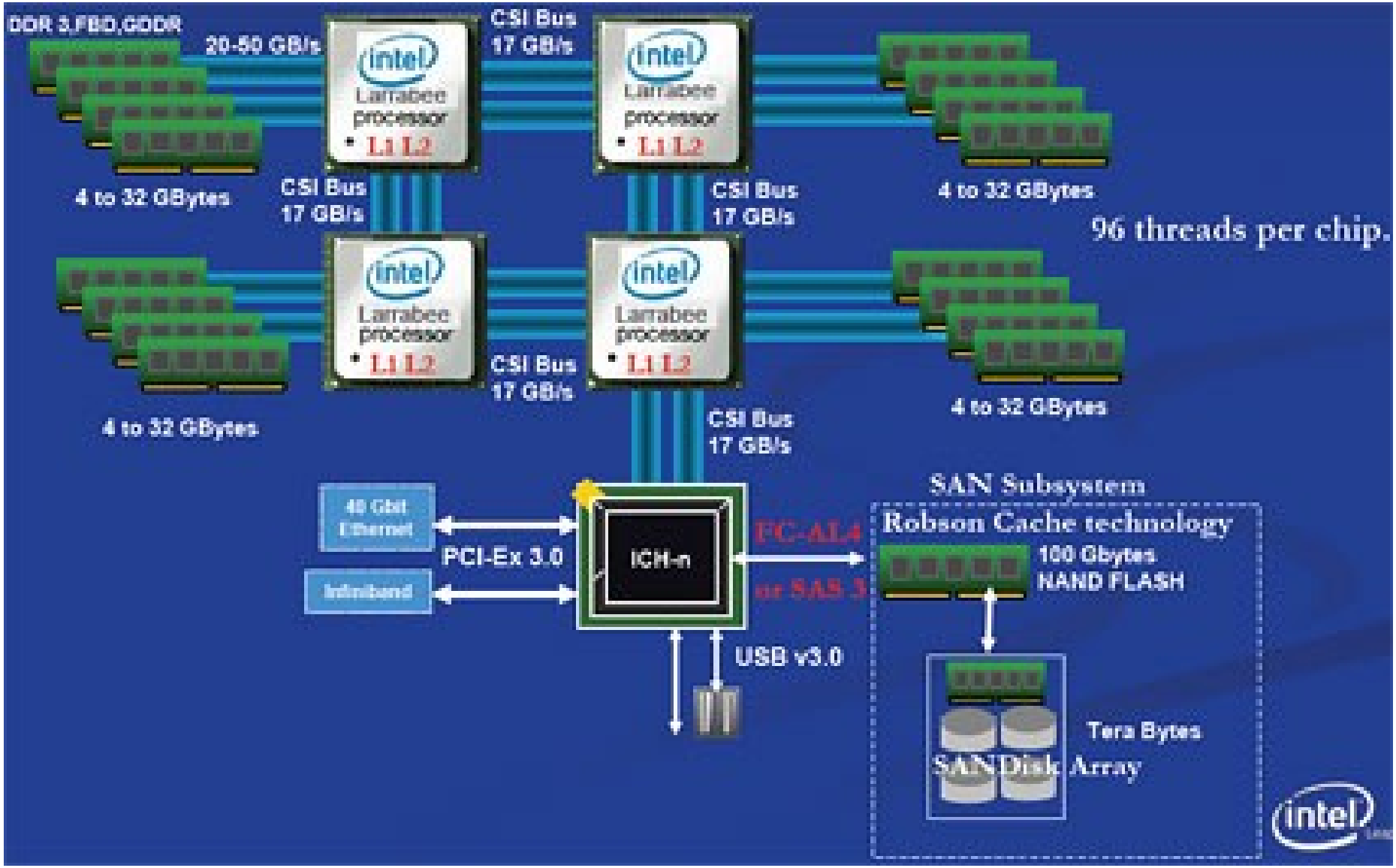
Modern CPUs are Distributed Systems


Queue, Uncore
& I/O
Core
Core
Core
Core
Core
Core
Shared
L3 Cache
Memory Controller



https://www.guru3d.com/story/intel-lunar-lake-processor-architecture-die-and-pch-annotated/

https://www.allaboutcircuits.com/news/intel-makes-way-ai-pc-era-new-lunar-lake-architecture/


DDR 3,FBD,GDDR
20-50 GB/s
CSI Bus
17 GB/s
Larrabee processor
L1 L2
4 to 32 GBytes
96 threads per chip.
40 Gbit Ethernet
Infiniband
PCI-Ex 3.0
ICH-n
USB v3.0
SAN Subsystem
Robson Cache technology
100 Gbytes NAND FLASH
Tera Bytes
SAN Disk Array
intel

Let's test how many operations Python can do per second

Task: Multiply two 1000x1000 matrix

How many operations?

- 1000^2 elements need to be computed
- Calculate a single cell requires to multiply 1000 pairs and sum the numbers ≈2000 operations
- In addition, we need to read the 1000 pairs from main memory and store them into main memory. So another ≈3000 operations
- We also need to create random numbers for both matrix → however, as this is "only" requires 2M random numbers, we ignore it
- Total: 5000 * 1000^2 ≈ 5 billion

On a 3.5GHz laptop this should take around 1.5s

Result

Python:

```
time python3 matrix_multiply.py

Checksum: 20260447223

python3 matrix_multiply.py  54.01s
user 0.33s system 98% cpu 55.428 total
```

Result Python

```
time python3 matrix_multiply.py

Checksum: 20260447223

python3 matrix_multiply.py  54.01s
user 0.33s system 98% cpu 55.428 total
```

Result C++

```
time ./matrix_multiply

Checksum: 20255776602

./matrix_multiply  2.52s user 0.02s
system 98% cpu 2.585 total
```

Result C++

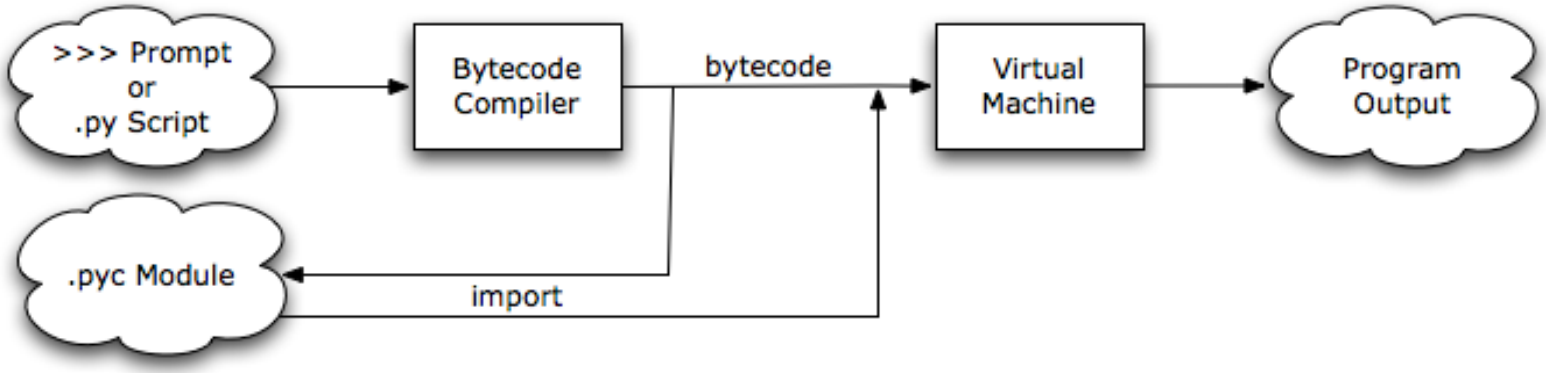
```
time python3 matrix_multiply_numpy.py

NumPy Checksum (don't time this):
20219073247

python3 matrix_multiply_numpy.py
1.45s user 0.75s system 338% cpu 0.650
total
```

Why????

Why is Python So Slow



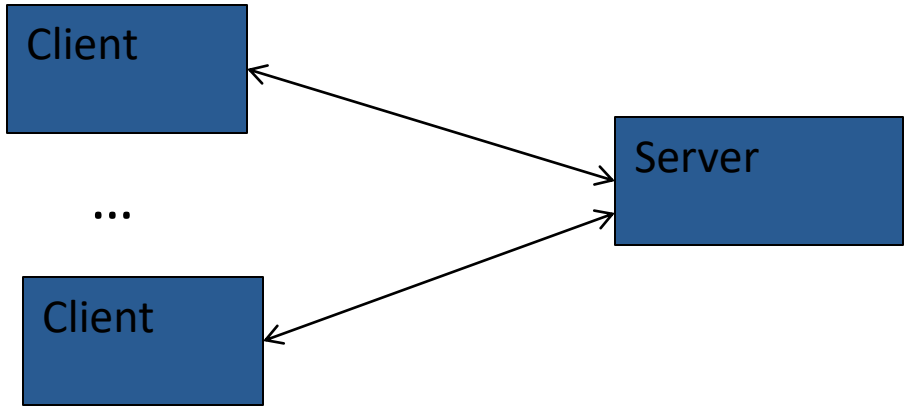
Virtual machine (VM) implementation is a loop that reads an instruction, and jumps to the code to execute the instruction

On modern CPUs this is very inefficient, because it results in many branch misses and poor processor cache locality

What if we need to tune performance?

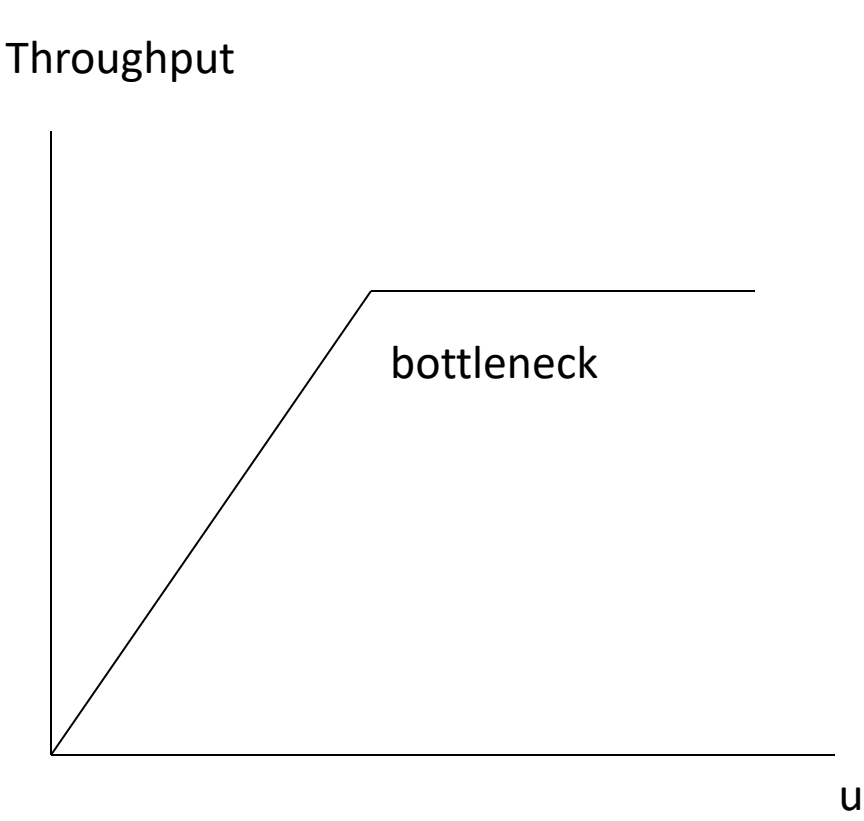
- High level tools like Python are fine for many problems but may be too slow, especially as you scale up problem size
- Typically requires optimization and redesign
- Some strategies
  - Buy more hardware
  - Use a different runtime
  - Improve implementation
- Today we will focus on some simple data-oriented improvements; parallelism and algorithmic tricks in later lectures

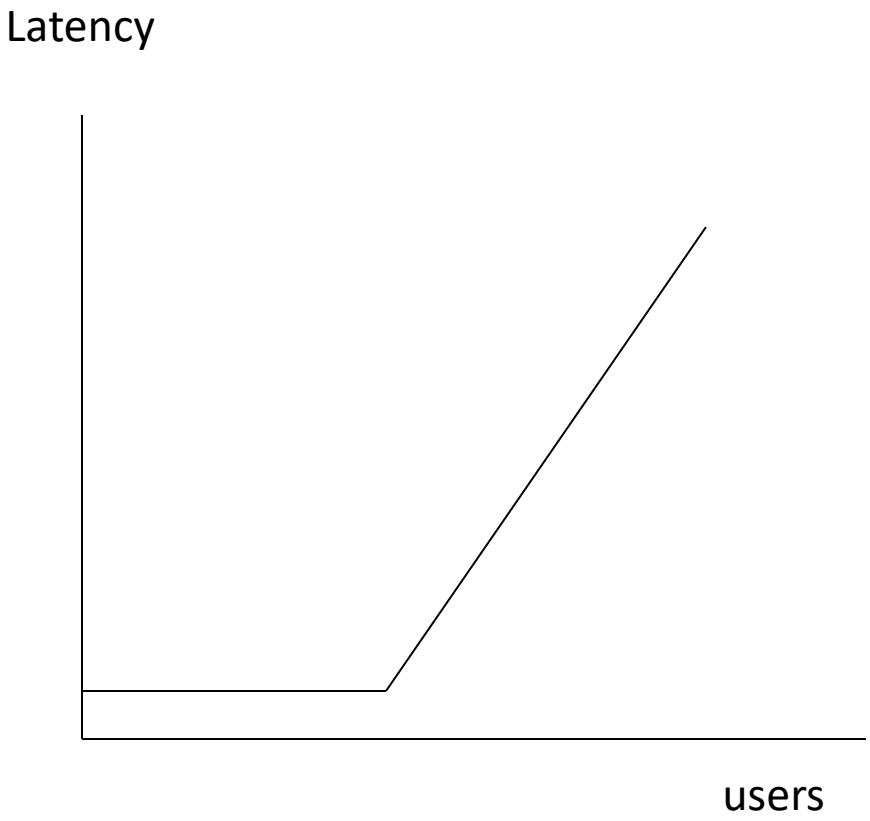
Performance metrics



- Performance metrics:
  - **Throughput**: request/time for many requests
  - **Latency**: time / request for single request
- Latency = 1/throughput?
  - Often not; e.g., server may have two CPUs

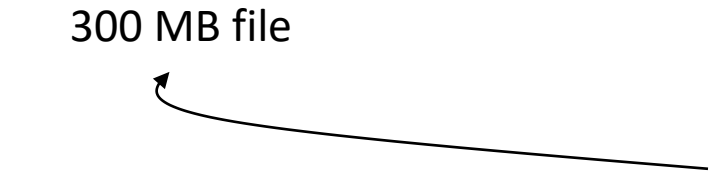
Heavily-loaded systems





- Once system busy, requests queue up

Approaches to finding bottleneck




```
df = pd.read_csv(PATH, delimiter='|',
     header=None, names=header)
print df[df['NAME'].str.contains("MADDEN")
```

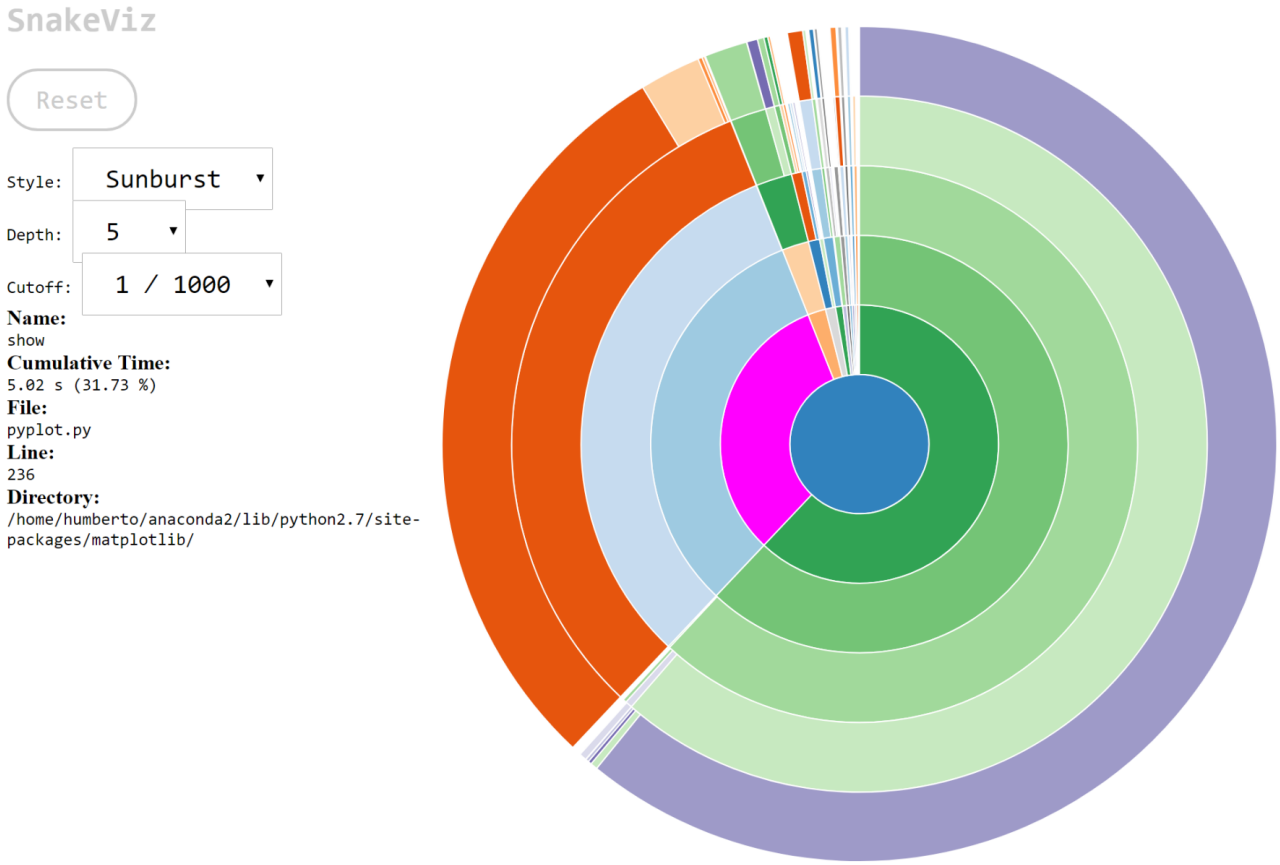
- Measure utilization of each resource
  - CPU is 100% busy, disk is 20% busy
  - CPU is 50% busy, disk is 50% busy, alternating
- Model performance of your approach
  - What performance do you expect?
- Guess, check, and iterate
  - Don't prematurely optimize

Some Tools

- print statements / timing
- top / system profilers
- code profilers

Python code profile

python –m cProfile script.py
snakeviz script.py

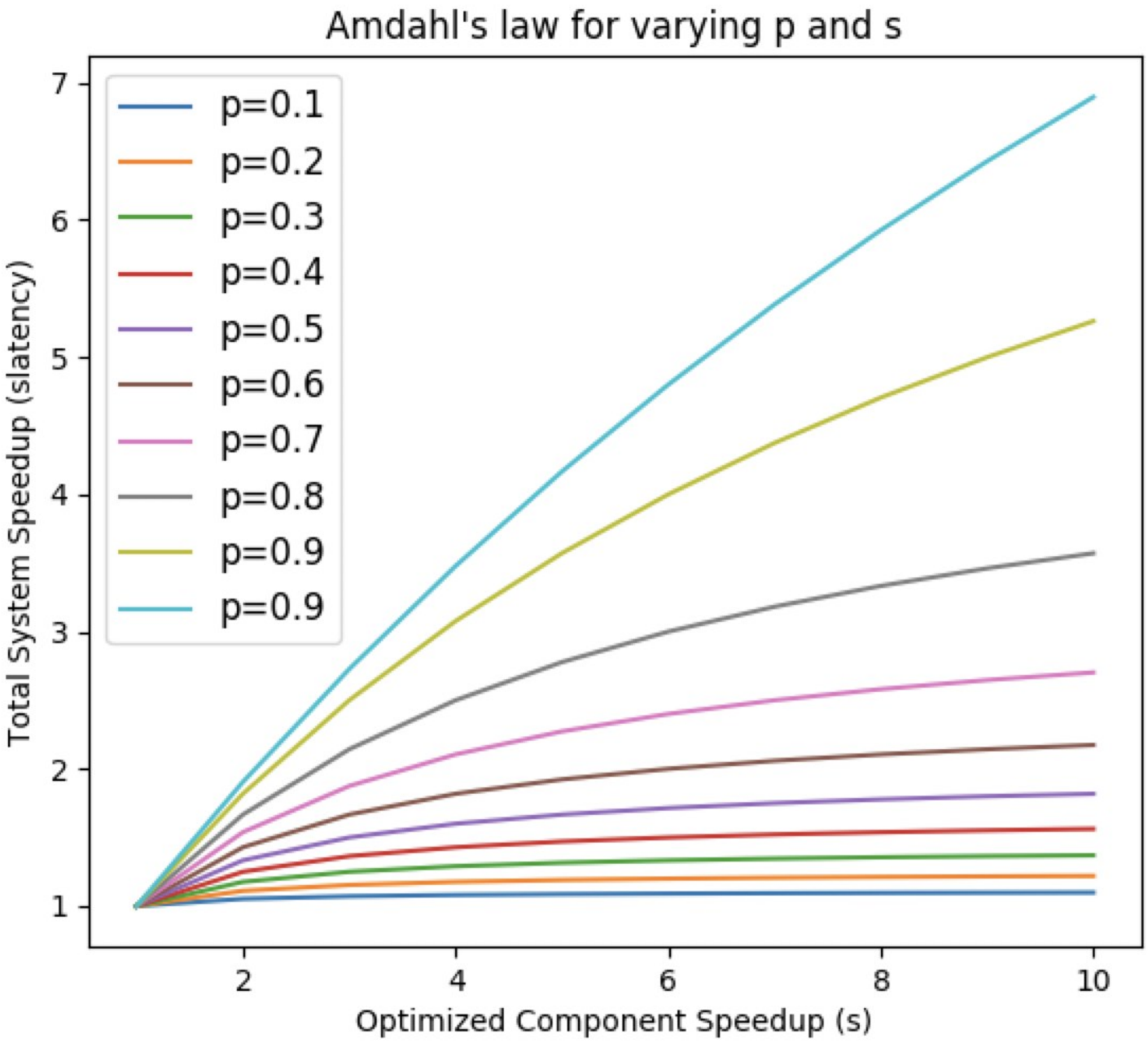


What Improvement Can We Expect

- Always keep Amdahl's law in mind

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

$S_{latency}$ is the over all speedup in all stages of a task
s is the speedup on a stage of the task that we optimize
p is the original proportion of the stage that was optimized


Amdahl's law for varying p and s
Total System Speedup (slatency)
Optimized Component Speedup (s)
p=0.1
p=0.2
p=0.3
p=0.4
p=0.5
p=0.6
p=0.7
p=0.8
p=0.9
p=0.9
1
2
3
4
5
6
7
2
4
6
8
10

Fixing a bottleneck

- Get better hardware
- Use better execution environment
- Find better algorithm
- Write better implementation; strategies
  - Indexing
  - Predicate push down
  - Early projection
  - Caching
  - Efficient joins
  - Partitioning & parallelism -- not today

What about using multiple cores in Python?

- **Python Is (Mostly) Single-Threaded**
- **CPython uses the Global Interpreter Lock (GIL)**
  - Only *one* thread can execute Python bytecode at a time
  - Prevents race conditions in memory management
- **Multiple threads exist**, but **only one runs Python code simultaneously (threading module)**
- Threads **don't speed up CPU-bound tasks** (e.g., computation-heavy operations)
- Threads **can** improve performance for **I/O-bound tasks** (waiting on network, disk, APIs)

Solution

1. **threading module (I/O-bound only)**
   - Lightweight threads managed inside one process
   - Still restricted by the GIL for CPU work

2. **multiprocessing module**
   - Spawns multiple Python processes → no shared GIL
   - Best option for CPU-bound parallelism
   - Uses inter-process communication (IPC)

3**. concurrent.futures**
   - High-level API over threads or processes
   - ThreadPoolExecutor → I/O-bound tasks
   - ProcessPoolExecutor → CPU-bound tasks

4. **Async I/O (asyncio)**
   - Not multithreading, but cooperative concurrency
   - Great for high-scale I/O, networking, servers
   - Single-threaded but highly parallel in behavior for I/O

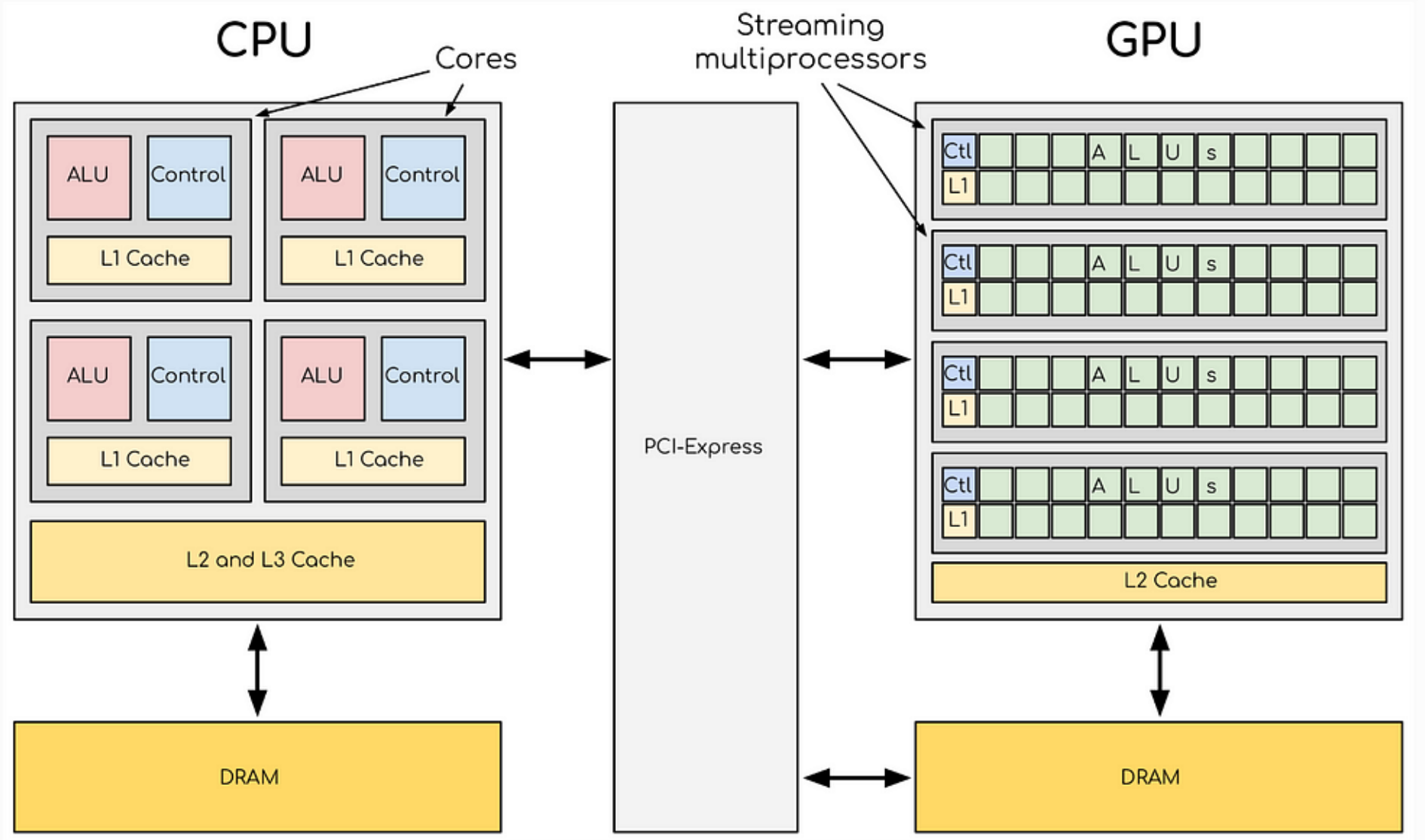
5. C **Extensions / NumPy / Numba**
   - Heavy compute pushed into C/C++ layers
   - Bypass the GIL for numeric or vectorized operations
   - True parallelism inside compiled native code

6. **Alternative Interpreters**
   - Jython, IronPython → No GIL (but different ecosystems)
   - PyPy STM → Experimental Software Transactional Memory

Difference Process vs Thread

Aspect	Process	Thread
Definition	A running instance of a program with its own dedicated memory space and resources.	A segment of a process; the smallest unit of execution that can be managed by the OS scheduler.
Memory	Has its own separate virtual address space, isolating it from other processes.	Shares the memory space, code, and OS resources (like open files) of its parent process.
Overhead	"Heavyweight"; creation, termination, and context switching are resource-intensive and slow.	"Lightweight"; creation, termination, and context switching are faster and consume fewer resources.
Communication	Communication between processes (Inter-Process Communication or IPC) is more complex and requires specific system mechanisms (e.g., pipes, sockets).	Communication is faster and simpler as threads can directly access and modify shared data within the same memory space.
Isolation	Highly isolated; a crash in one process generally does not affect others, enhancing stability.	Not isolated within the process; a crash or error in one thread can potentially bring down the entire process.

What about GPUs?



Summary

- Python is often slow
- Identifying performance bottlenecks is an art
  - Figure out if you have an I/O or CPU problem
  - Estimate expected performance
  - Remember Amdahl's law!
- Rewriting in low level languages can help
- Using more efficient data accesses can help