TRANSFORMERS: AGE OF PARALLEL MACHINES

(a biased introduction to Computer architecture)

Ana Lucia Varbanescu, University of Twente, NL

A.L.Varbanescu@utwente.nl

Agenda (ambitious)

- ~~Part 1 : Introduction to computer systems~~
  - ~~CPUs, Memory, Caching, Accelerators~~

- Part 2 : Parallelism and parallel machines
  - Flynn's taxonomy, SIMD/vectorization, multi-core/many-core
  - Alternative architectures (FPGAs, AI-based, ...)

- Part 3 : Performance and tools
  - Basic metrics, models, counters …

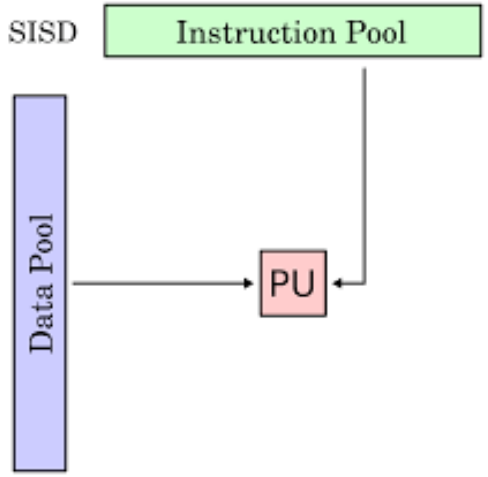
- Part 4 : Where to ?
  - Famous last words …

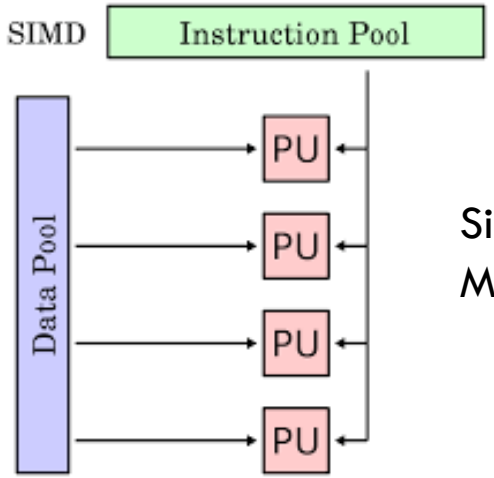
"Larry, do you remember where we buried our hidden agenda?"

PART 2: PARALLELISM & MACHINES

First taxonomy: Michael Flynn (1966)

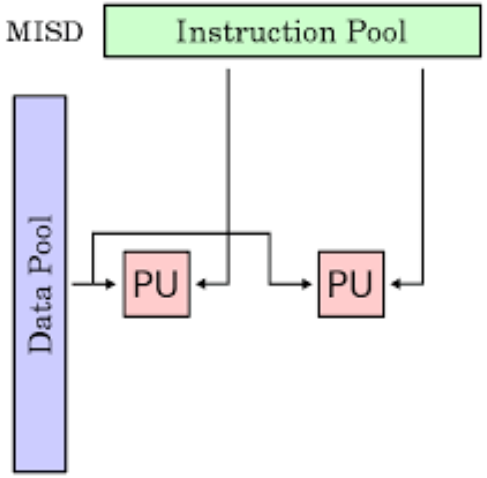
Single Instruction
Single Data

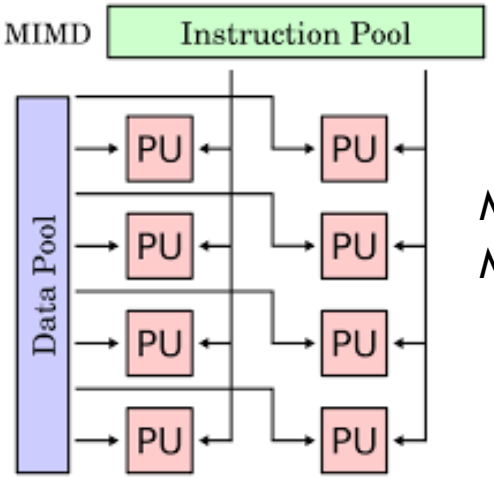




Single Instruction
Multiple Data

Multiple Instructions
Single Data



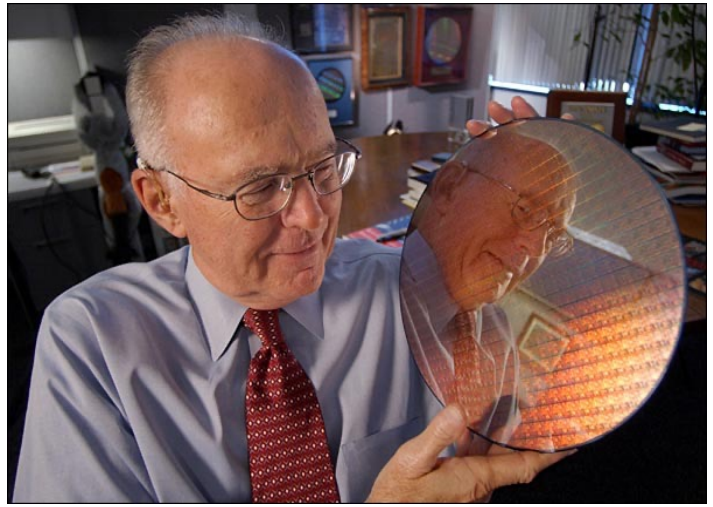


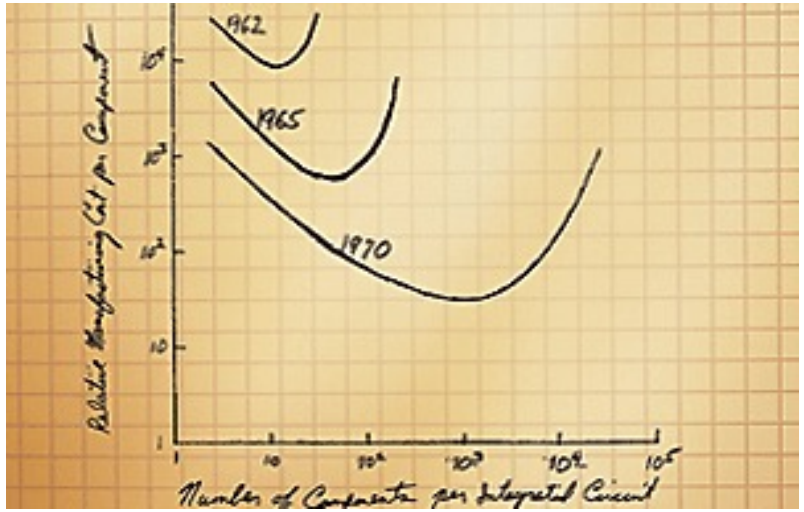
Multiple Instructions
Multiple Data

Before 2005: technology push

Moore's Law

- Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.





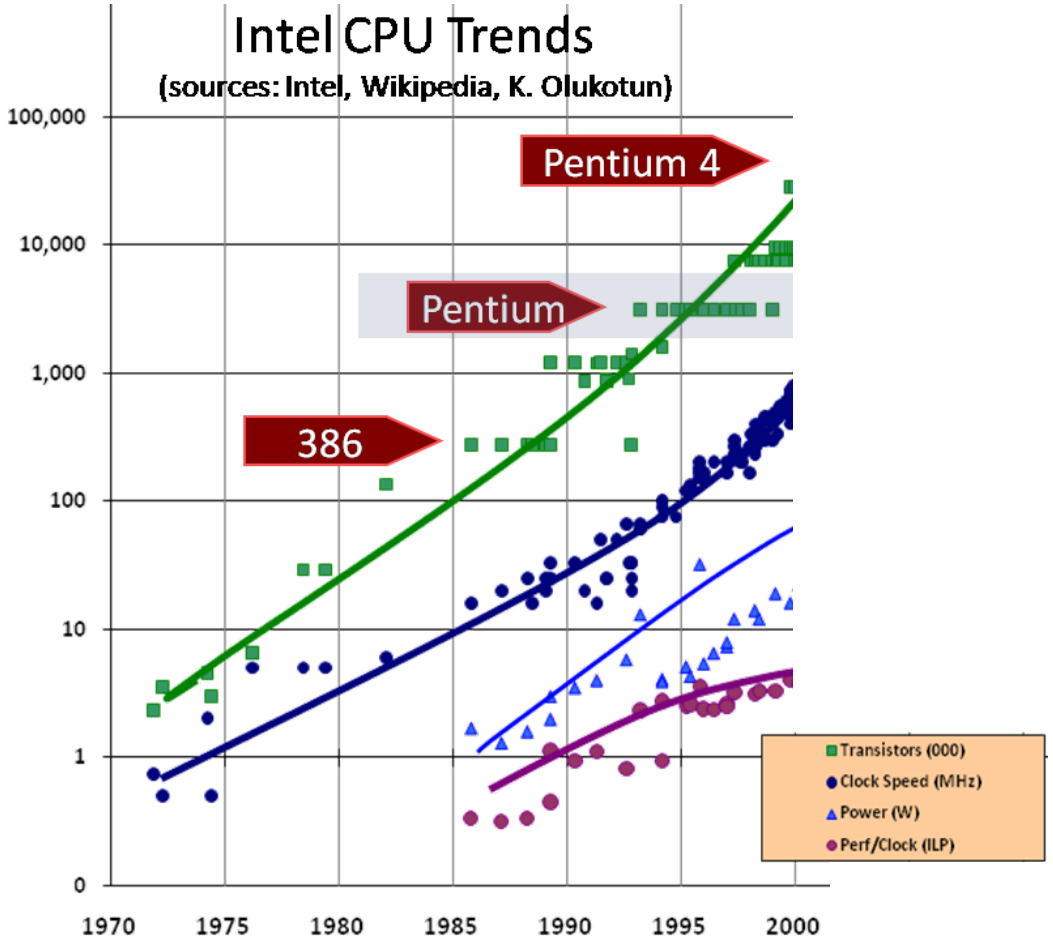
"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase...." Electronics Magazine 1965

Until early 2000s …

More transistors = more performance

Thus, every 18 months,
we had better and faster
processors.

- Higher clock-speed
- Higher perf/cycle
- Same power



Wait … why do I care?

- More transistors = … ?

= more functionality

  - Think more functional units, more complex units, etc….

- Higher perf/clock (aka, higher ILP) = … ?

= more operations per cycle

  - Faster overall applications (when they have different operations…)

- Higher clock frequency = …?

= more operations per time unit

  - Faster instructions => faster overall application

- Higher power = … ?

= global warming …

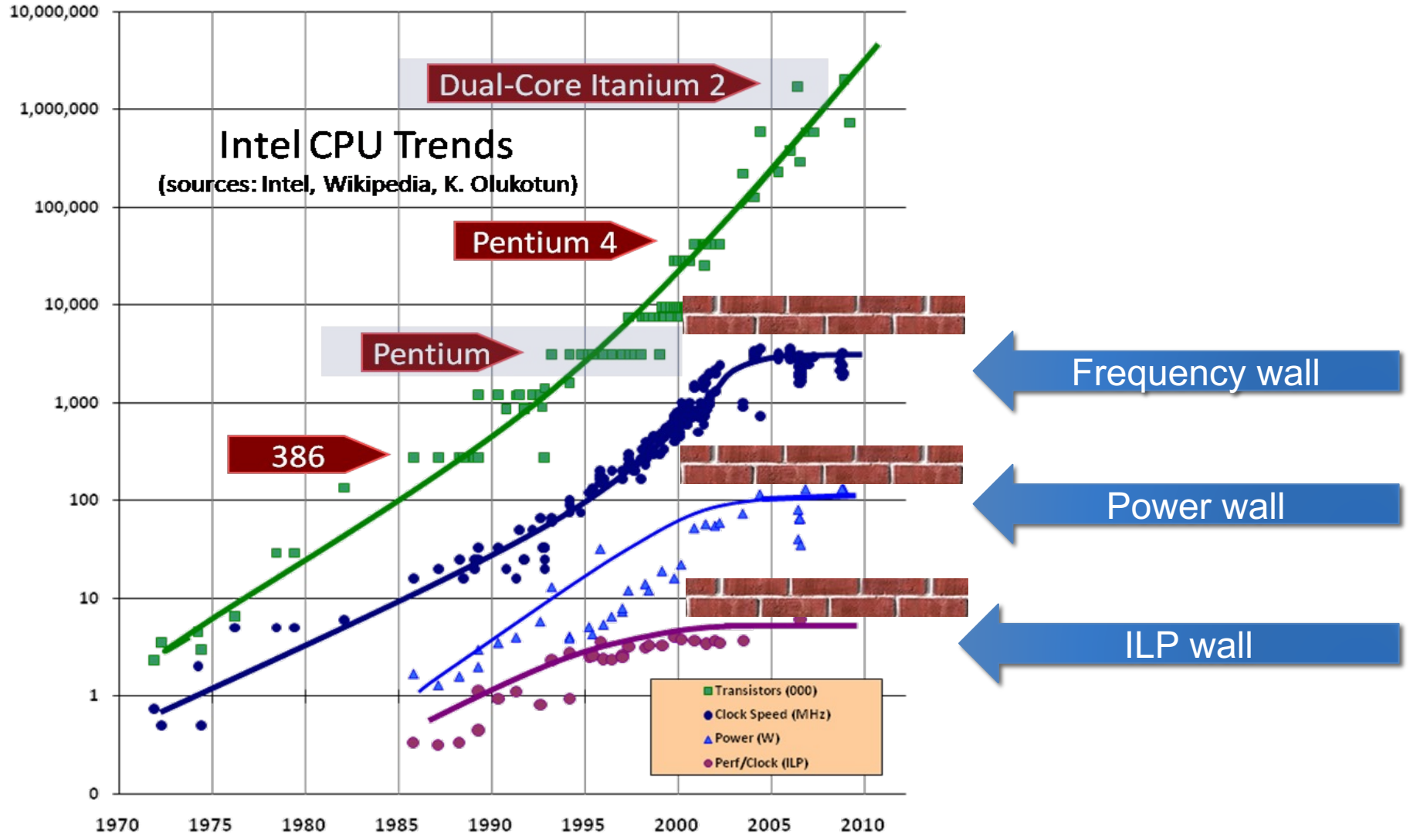
  - Ideally, we want power consumption to be low

Until early 2000s …

Parallelism = interesting and “quirky”, but not main-stream

- **Pro:** Better performance than frequency scaling would provide.
- **Con:** Parallelizing code was not always worth the effort
  - Do nothing: the performance will double ~ every 18 months

Around 2005: “hitting the walls”

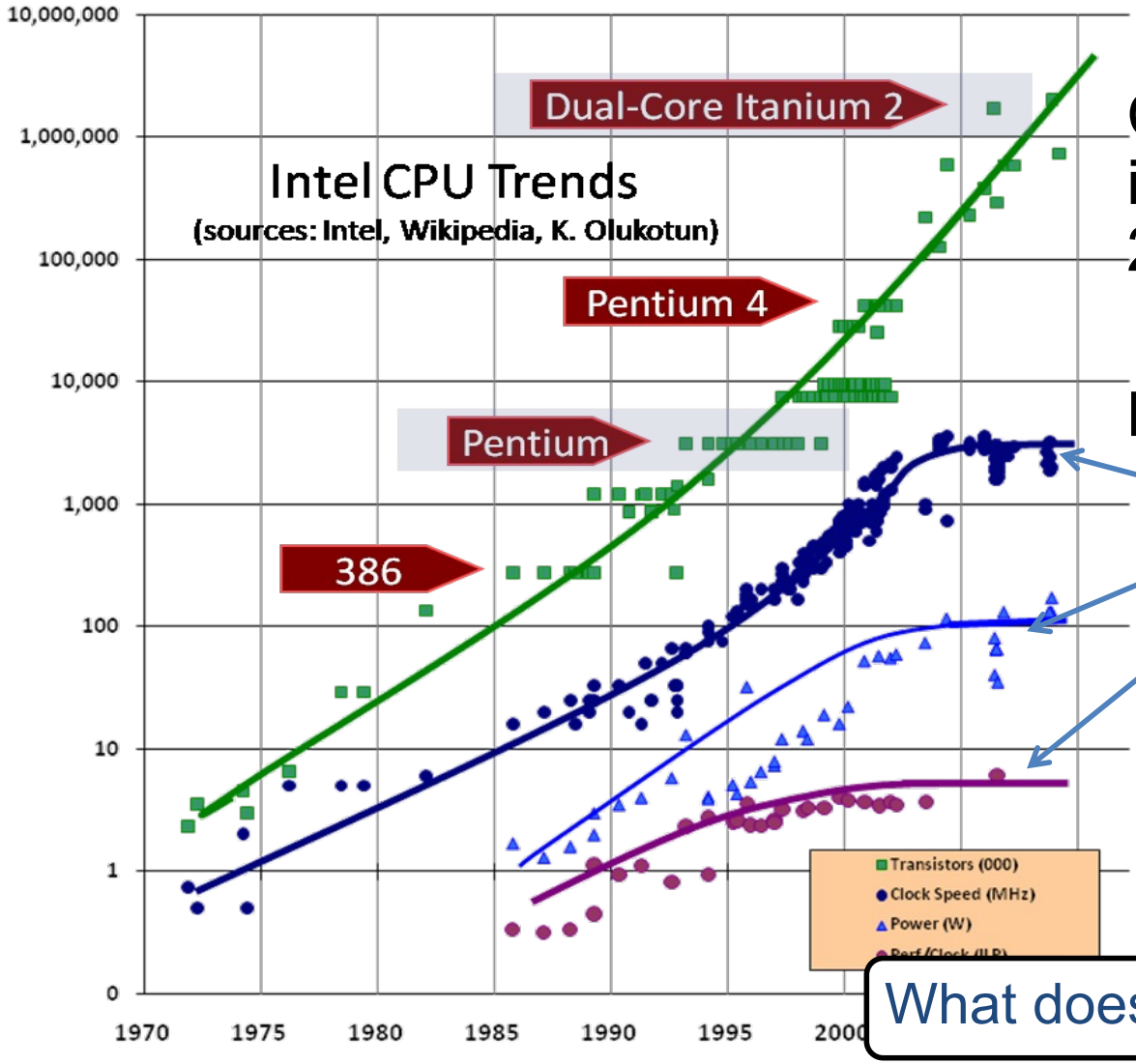


Single core performance scaling

- The rate of single core performance scaling has significantly decreased (essentially, to 0)
  - Frequency scaling limited by power
  - ILP scaling tapped out
  - Design complexity posing serious limitations

- No more free lunch for software developers!
  - No more dramatic increase of software performance for free.

So what?



Chip density can still increase about 2x every 2 years

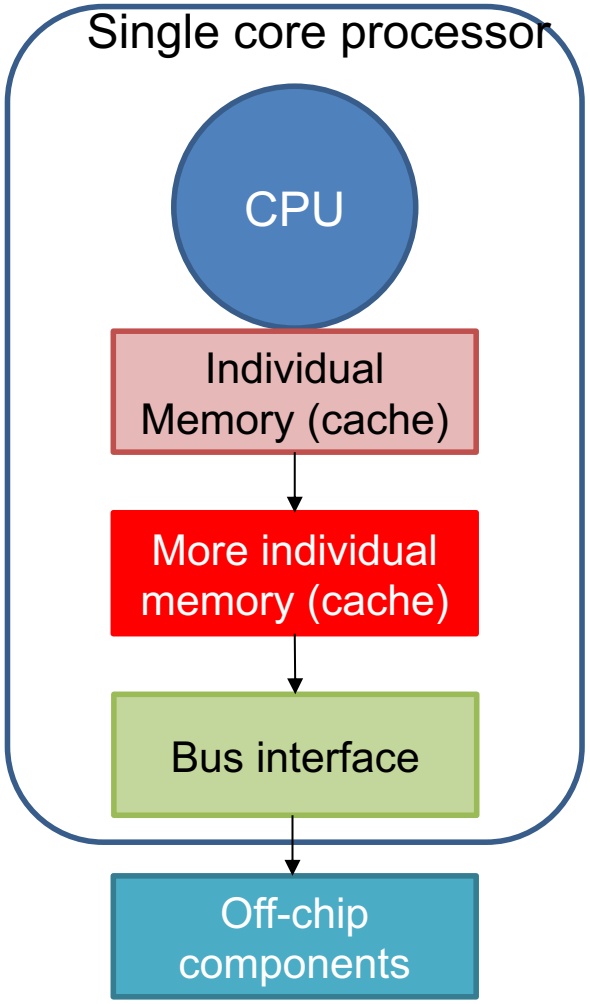
BUT

- **Clock speed is not**
- **Power is not**
- **Instruction Level Parallelism is not**

What does this mean in practice?

Traditionally … single core CPUs

- More transistors = more functionality
- Improved technology = faster clocks = more speed
- Every 18 months => better and faster processors.

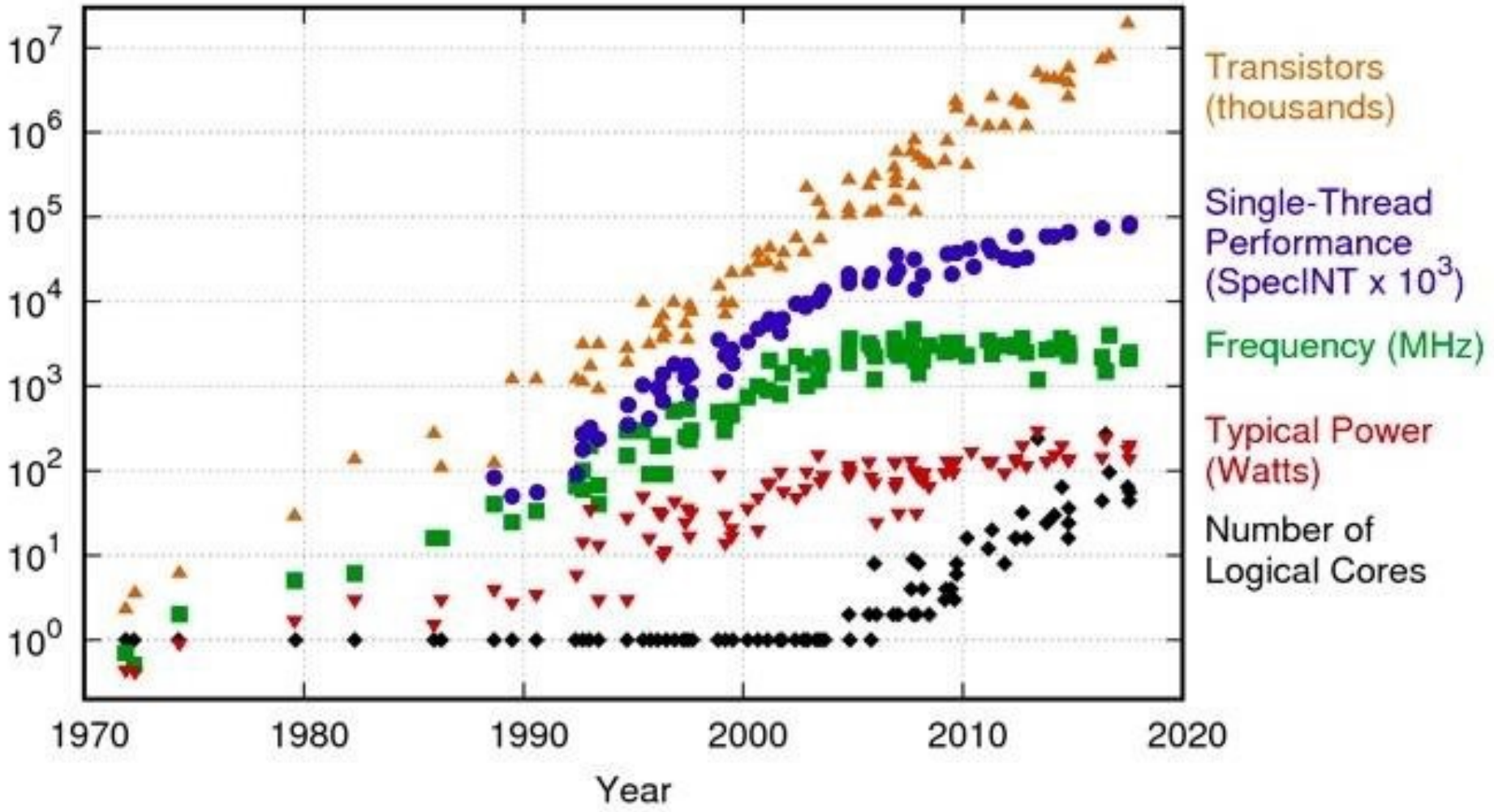


Not anymore!
We no longer gain performance by “growing” sequential processors …

New ways to use transistors

Improve PERFORMANCE by using parallelism on-chip: multi-core (CPUs) and many-core processors (GPUs).

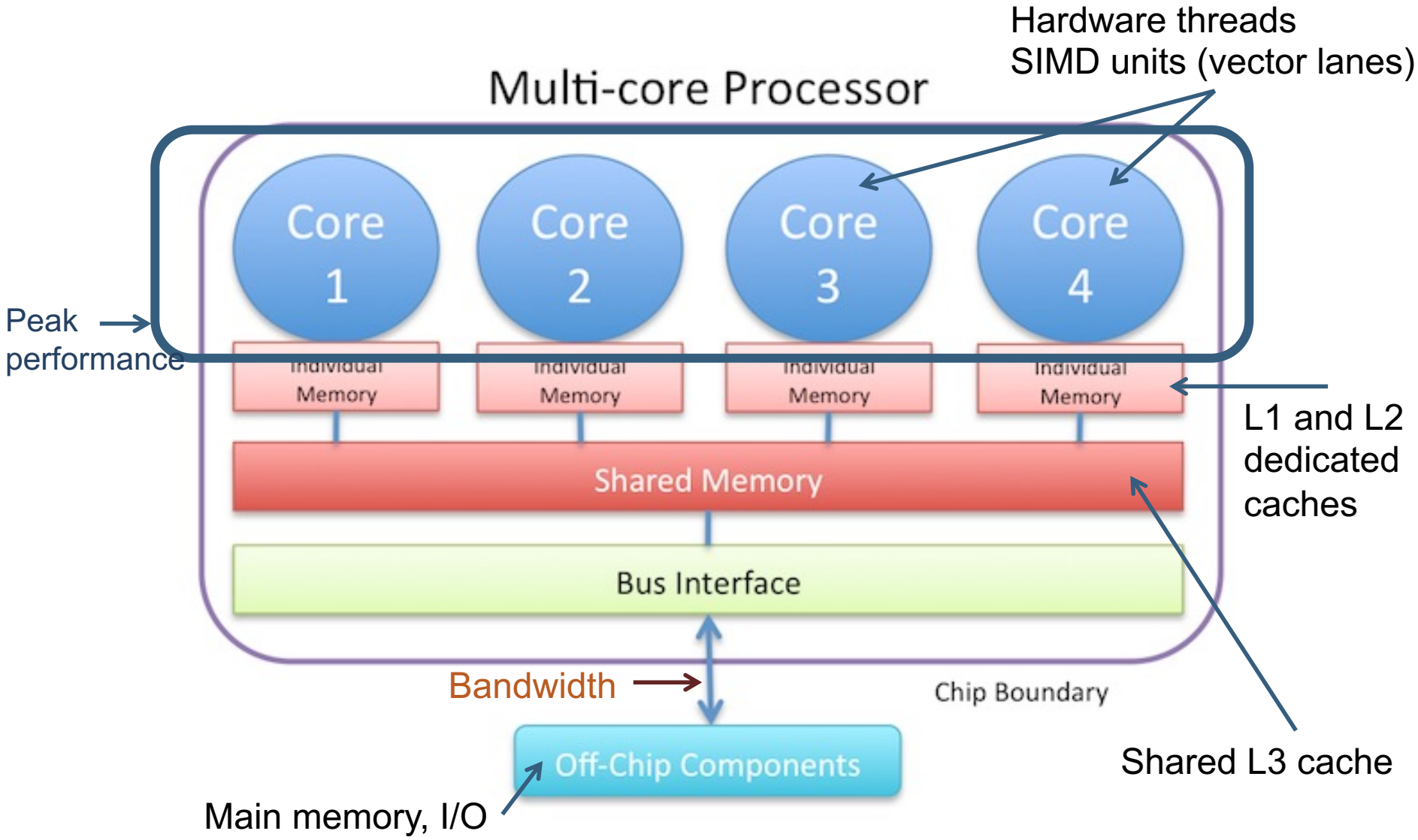
The shift to multi-core



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 2010-2017 by K. Rupp

Multi-core CPUs

Generic multi-core CPU

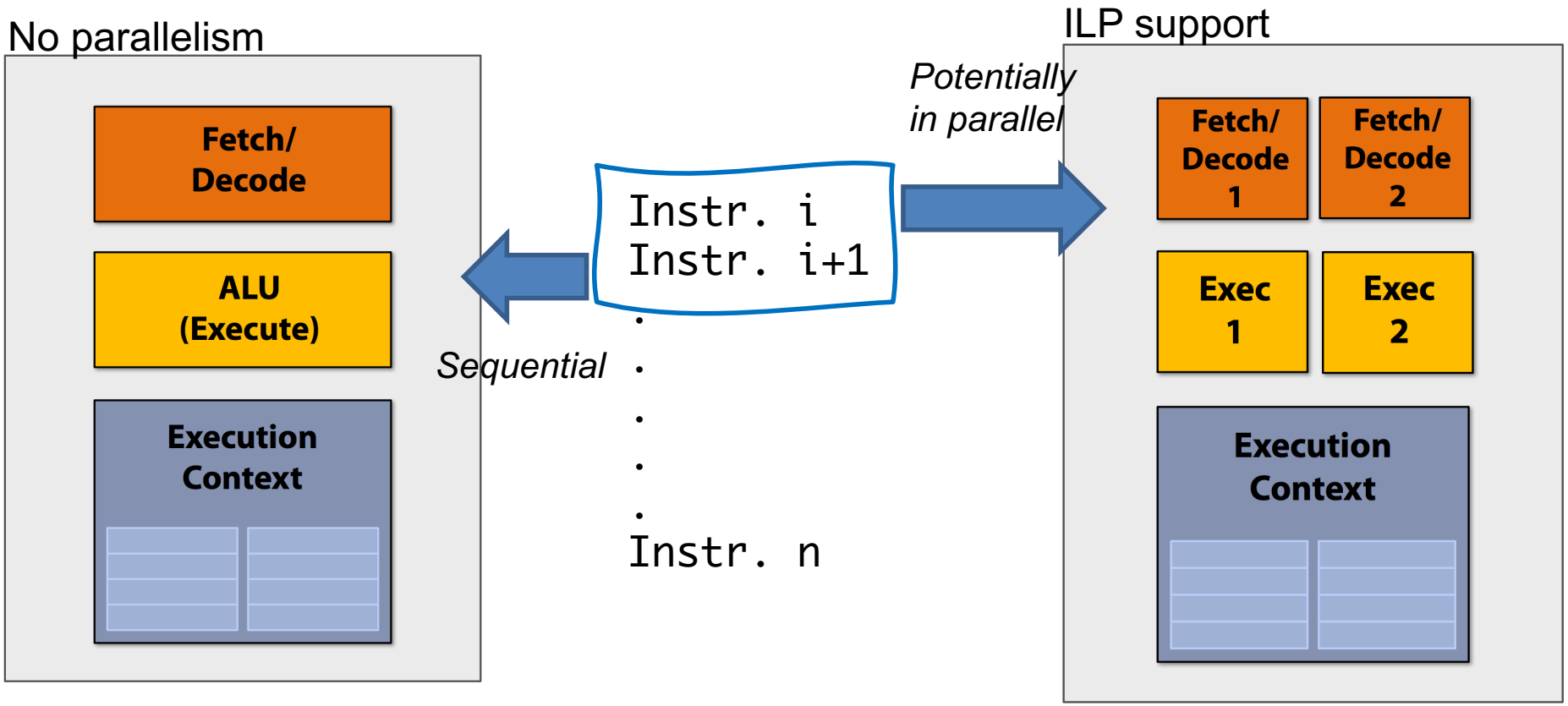


CPU levels of parallelism

- **Instruction-level** parallelism (e.g., superscalar processors) (fine)
  - Multiple operations of different kinds per cycle
  - Implemented/supported by the instruction scheduler
    - typically in hardware
- **SIMD** parallelism = data parallelism (fine)
  - Multiple operations *of the same kind* per cycle
  - Run same instruction on vector data
  - Sensitive to divergence
  - Implemented by programmer OR compiler
- **Multi-Core** parallelism ~ task/data parallelism (coarse)
  - 10s of powerful cores
    - Hardware hyperthreading (2x)
  - Local caches
  - Symmetrical or asymmetrical threading model
  - Implemented by programmer

(1) ILP (Instruction level parallelism)

- Multiple instructions issued & executed in the same cycle



*Diagrams adapted from CMU's course "Parallel Computer Architecture and Programming" – http://15418.courses.cs.cmu.edu/spring2016/lectures

No programmer's intervention!

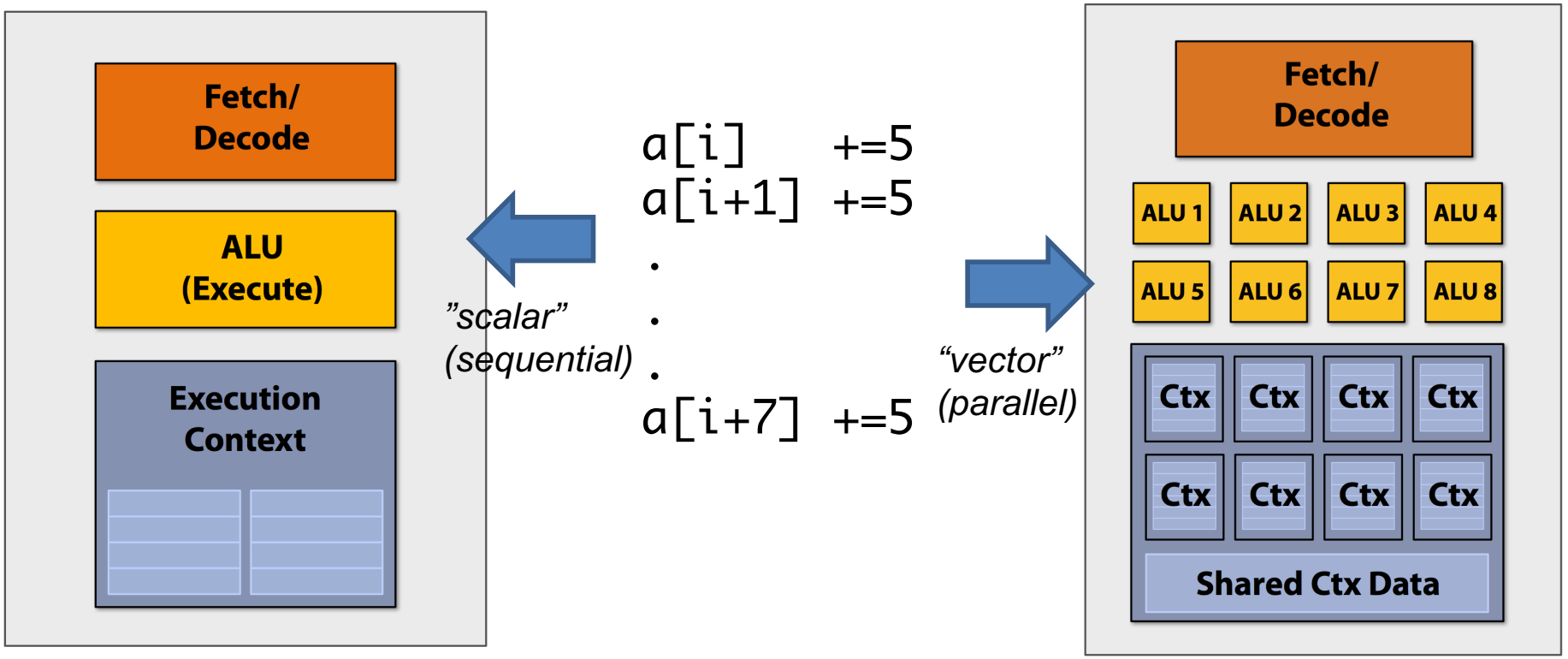
Implementing ILP

- Super-scalar processors
  - "dynamic scheduling": instruction reordering and scheduling happens in hardware
    - More complex hardware
      - More area, more power …
  - Adopted in most high-end CPUs today

- VLIW processors
  - "static scheduling": instruction reordering and scheduling is done by the compiler
    - Simpler hardware
      - Less area, less power
  - Adopted in most GPUs and embedded CPUs

(2) SIMD (single instruction, multiple data)

- Same instruction executed on multiple data items



*Diagrams adapted from CMU’s course "Parallel Computer Architecture and Programming” –

Scalar vs SIMD operations

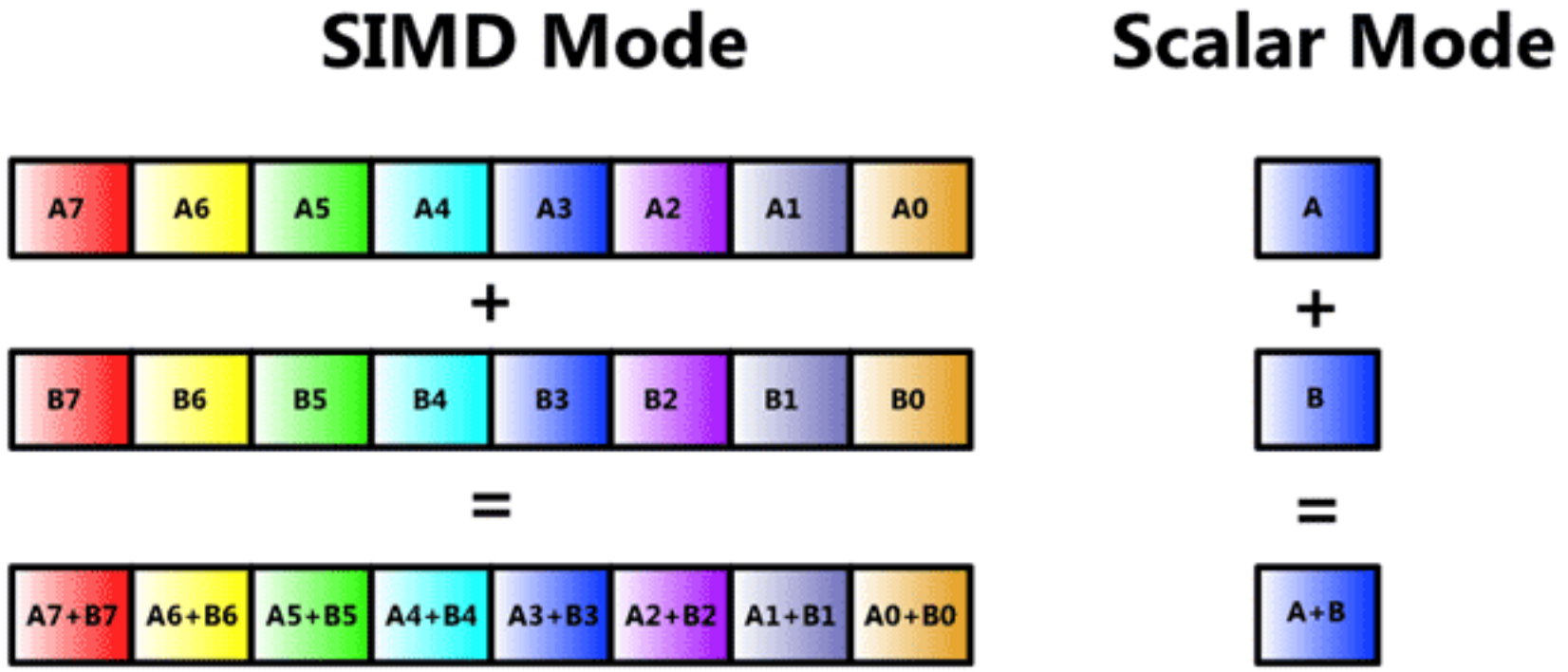
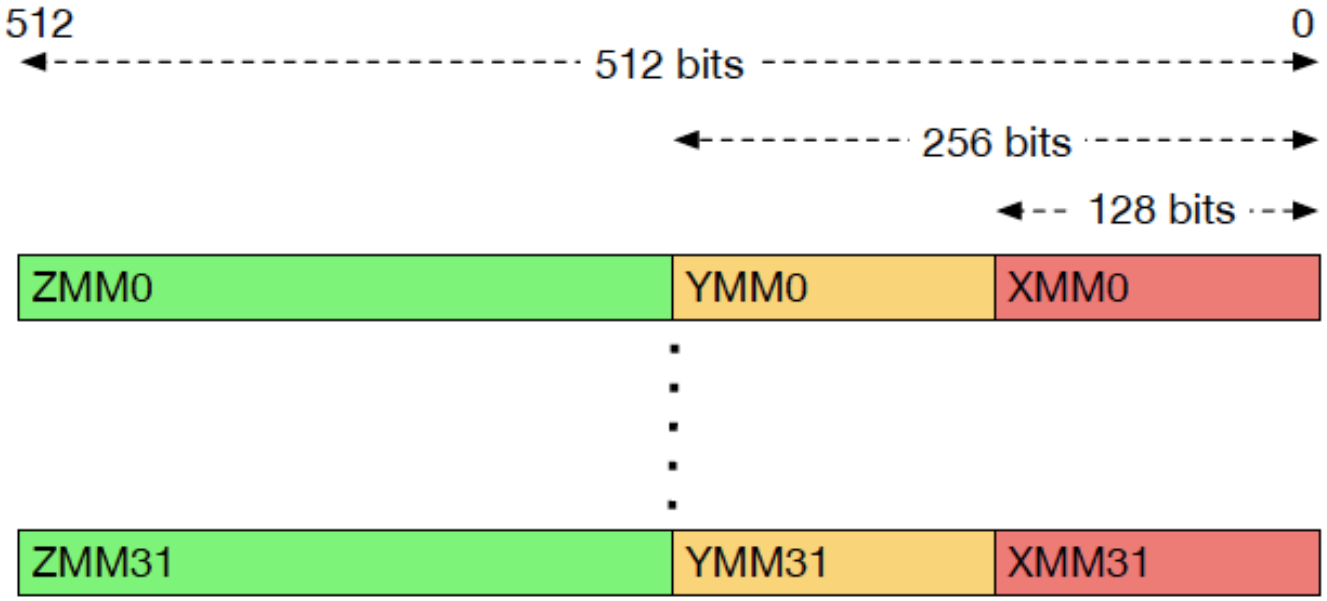


Image: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html

Requires programmer's (or compiler's) intervention!

Implementing SIMD

- SIMD extensions: special registers and functional units
- Multiple generations of SIMD extensions
  - SSE4.x = 128 bits
  - AVX / AVX2 = 256 bits (most available CPUs, DAS-5 included)
  - AVX-512 = 512 bits (Intel Xeon Phi, partial in most recent CPUs)



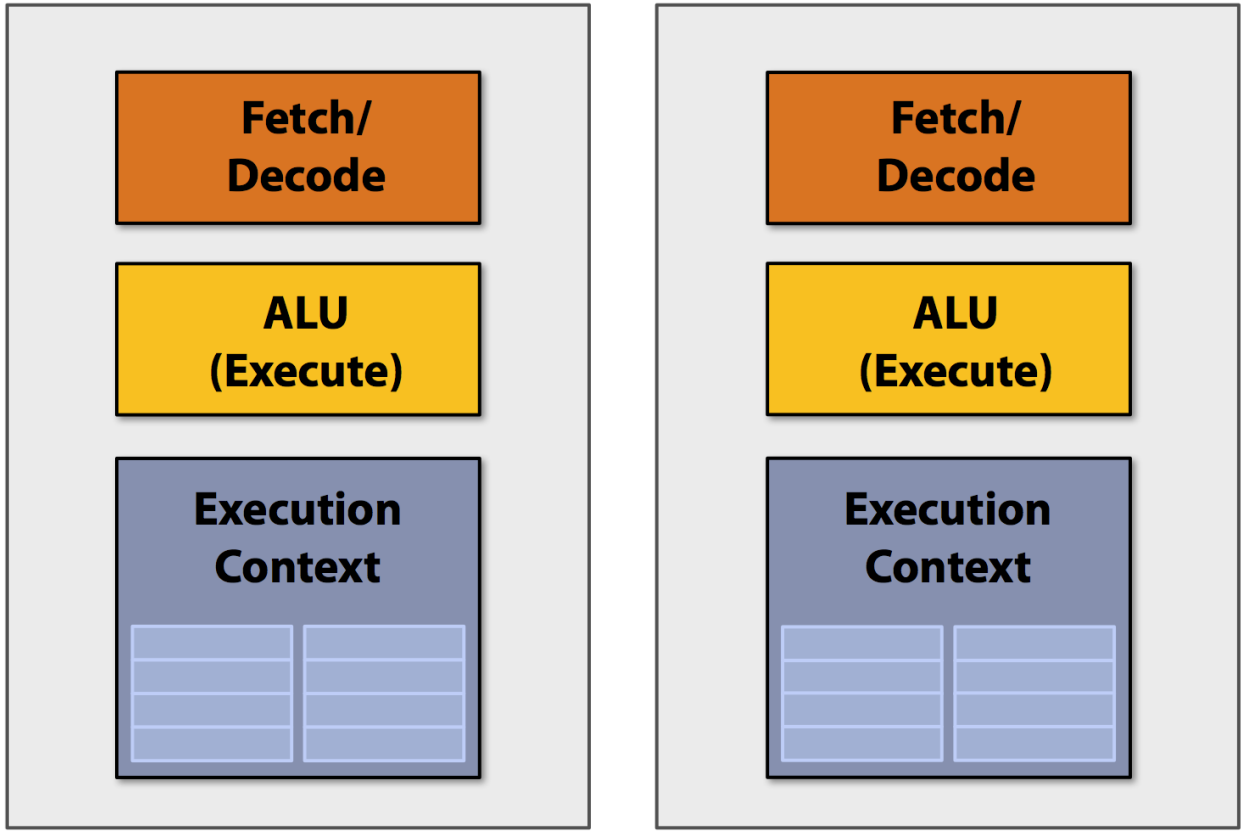
SIMD programmer intervention

- Auto-vectorization
  - Typically enabled with “-O” compiler flags
- Compiler directives
  - Specifically add directives in the code to ~~force~~ persuade the compiler to vectorize code
- C or C++ **intrinsics**
  - Wrappers around ASM instructions
    - Declare vector variables
    - Name instruction
    - Work on variables, not registers
- Assembly instructions
  - Can write assembly to target SIMD

Requires programmer's (or compiler's) intervention and OS (operating system) support!

(3) Multi-core parallelism

- Two (or more cores) to execute different streams of instructions.



*Diagrams adapted from CMU's course "Parallel Computer Architecture and Programming" – http://15418.courses.cs.cmu.edu/spring2016/lectures

Multi-core programmer intervention

- Must define *concurrent tasks* to be executed in parallel
  - Typically called (software) threads
- Threads are executed per core
  - Under the OS scheduling
    - Some control can be exercised with additional programmer intervention

```
for i = 1 … 3*n
        do_something(i)
```

Core 0

```
for i = 1 … n
        do_something(i)
```

Core 1

```
for i = n+1 … 2*n
        do_something(i)
```

Core 2

```
for i = 2*n+1 … 3*n
        do_something(i)
```

Computer architecture talk

CPU features for ILP

- Instruction pipelining
  - Multiple instructions “in-flight”
- Superscalar execution
  - Multiple execution units
- Out-of-order execution
  - Any order that does not violate data dependencies
- Branch prediction
- Speculative execution

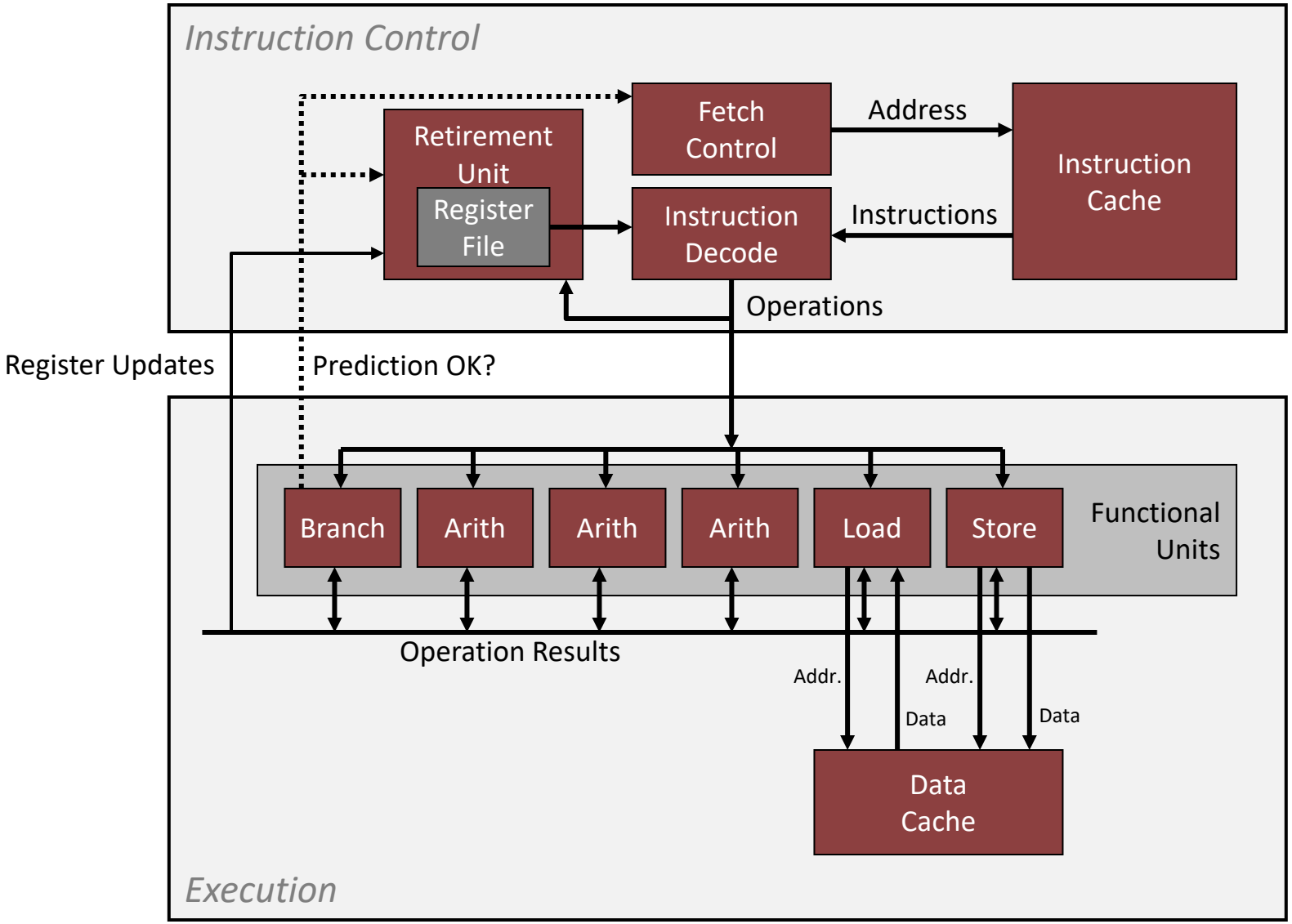
Superscalar, Out-of-order

- A superscalar processor can issue and execute *multiple instructions in one cycle*.
  - The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- An out-of-order processor can reorder the execution of operations in hardware.

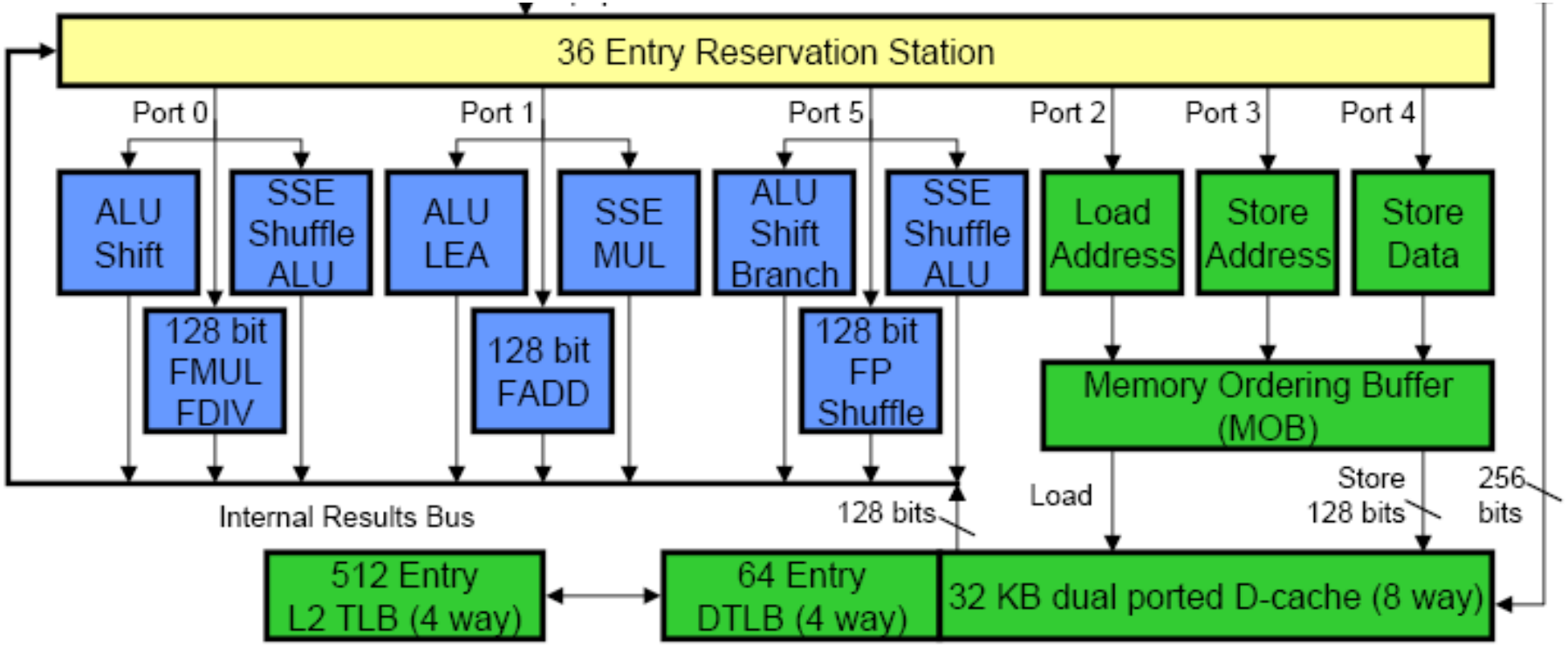
- Superscalar, out-of-order processors can take advantage of the *instruction level parallelism* that most programs have.

- Most modern CPUs are superscalar and out-of-order.
- Intel: since Pentium (1993)

Modern CPU Design



A real CPU …


36 Entry Reservation Station
Port 0
Port 1
Port 5
Port 2
Port 3
Port 4
ALU Shift
SSE Shuffle ALU
ALU LEA
SSE MUL
ALU Shift Branch
SSE Shuffle ALU
Load Address
Store Address
Store Data
128 bit FMUL FDIV
128 bit FADD
128 bit FP Shuffle
Memory Ordering Buffer (MOB)
Internal Results Bus
128 bits
Load
Store 128 bits
256 bits
512 Entry L2 TLB (4 way)
64 Entry DTLB (4 way)
32 KB dual ported D-cache (8 way)

SkyLake ®

Fetch & decode, producing multiple uOps

Optimize, reorder, schedule uOps

Multiple execution units, some SIMD

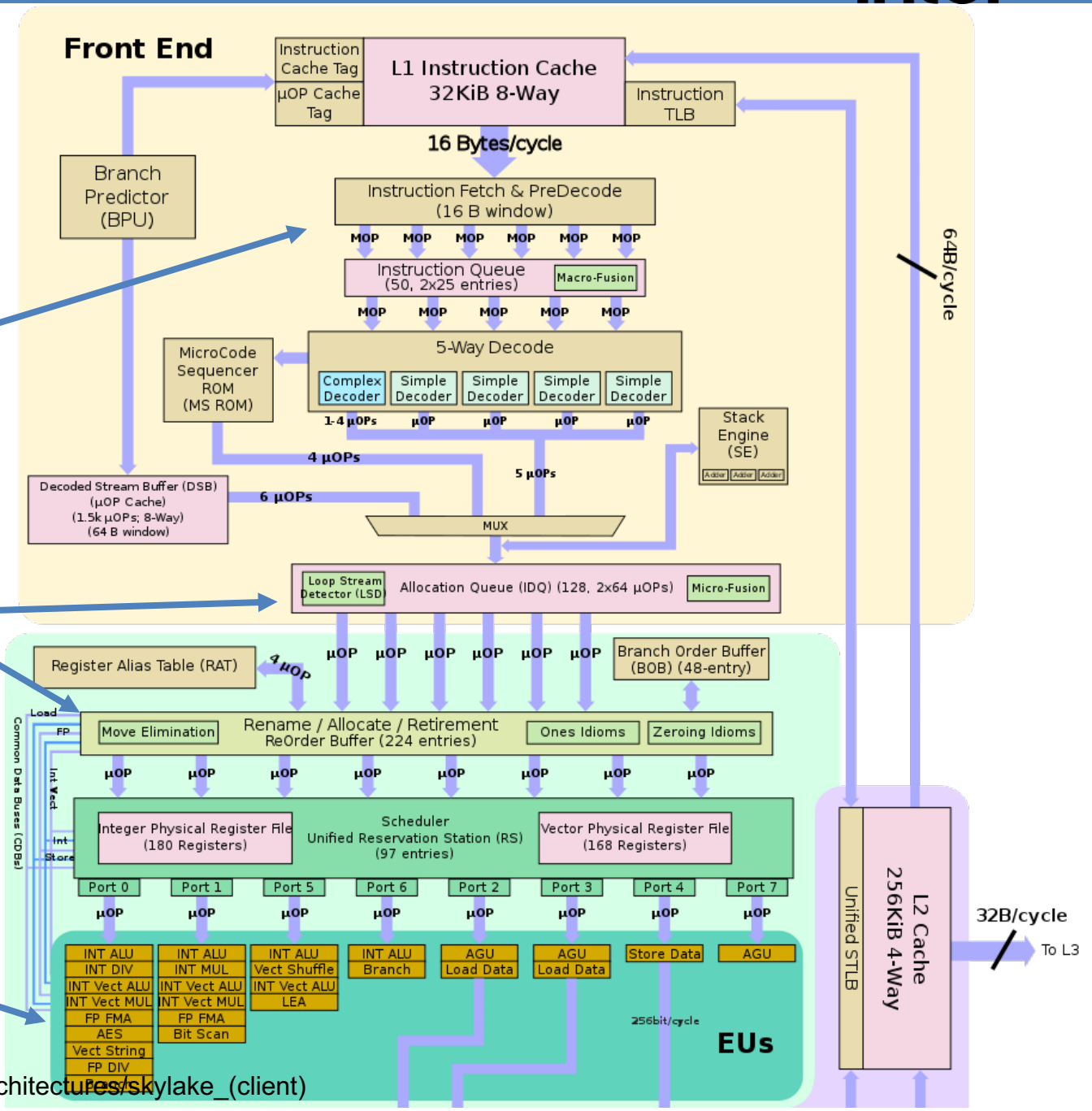


Image: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

Hardware multi-threading (or hyperthreading®)

”Are there hardware threads?!”

- Hardware (supported) multi-threading
  - Core manages thread context
    - Interleaved (temporal multi-threading) – employed in GPUs
    - Simultaneous (co-located execution) – e.g., Intel Hyperthreading

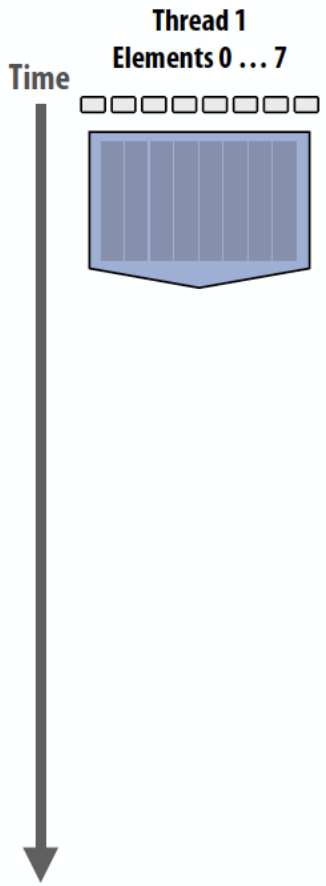
Why bother?

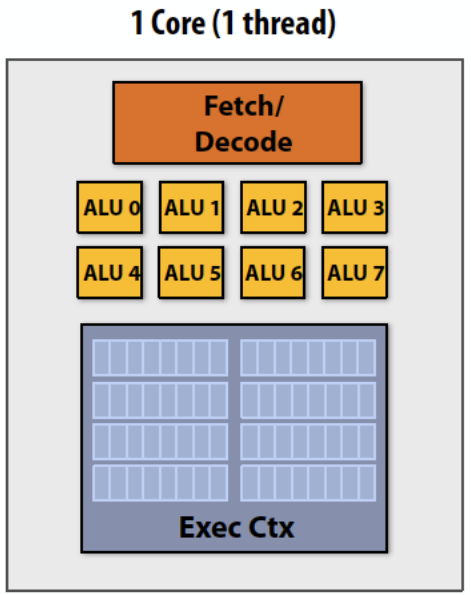
- Interleave the processing of multiple instruction streams on the same core to hide the latency of stalls

- Requires replication of hardware resources
  - Each thread uses its own PC to execute the instruction stream
  - Requires replication of register file

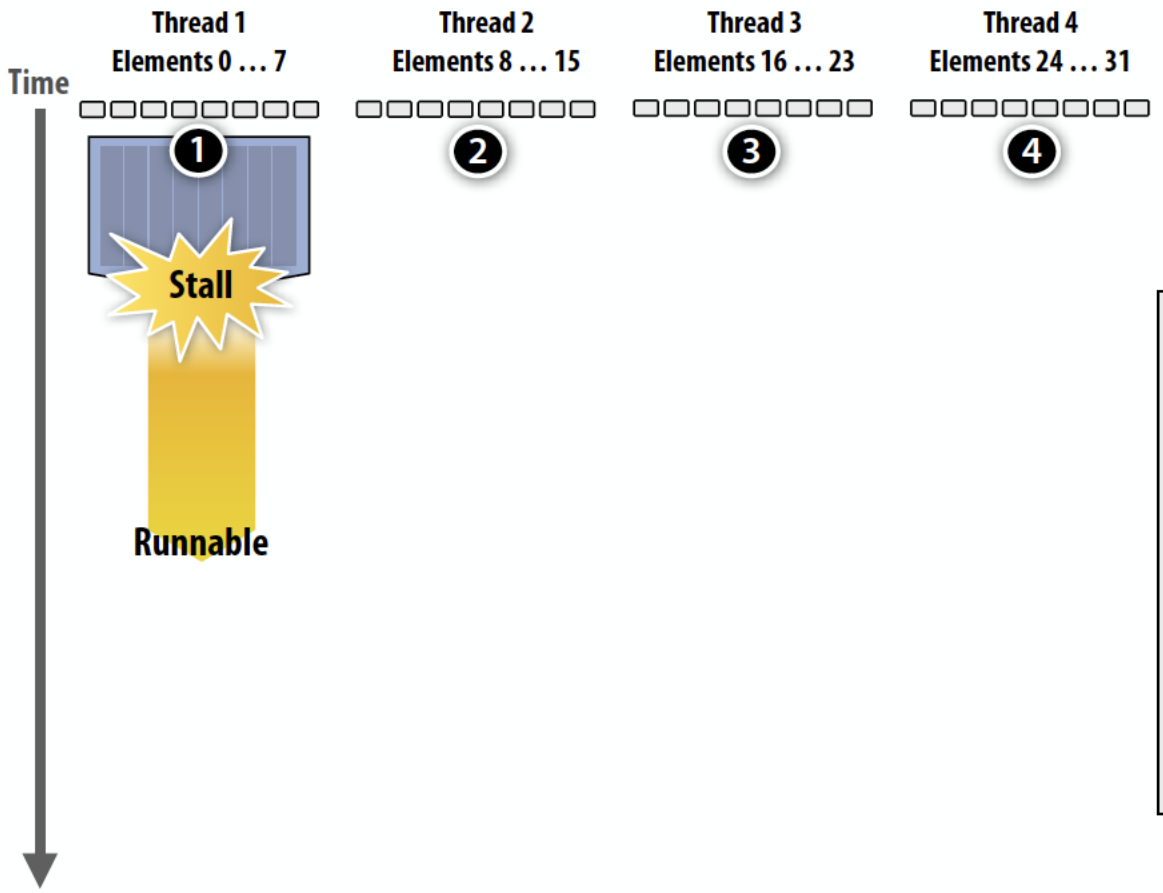
- *Performance improvement: higher throughput*

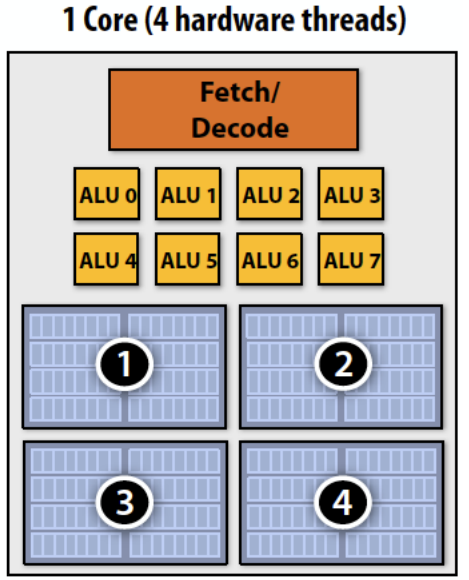
Advantage: increased throughput



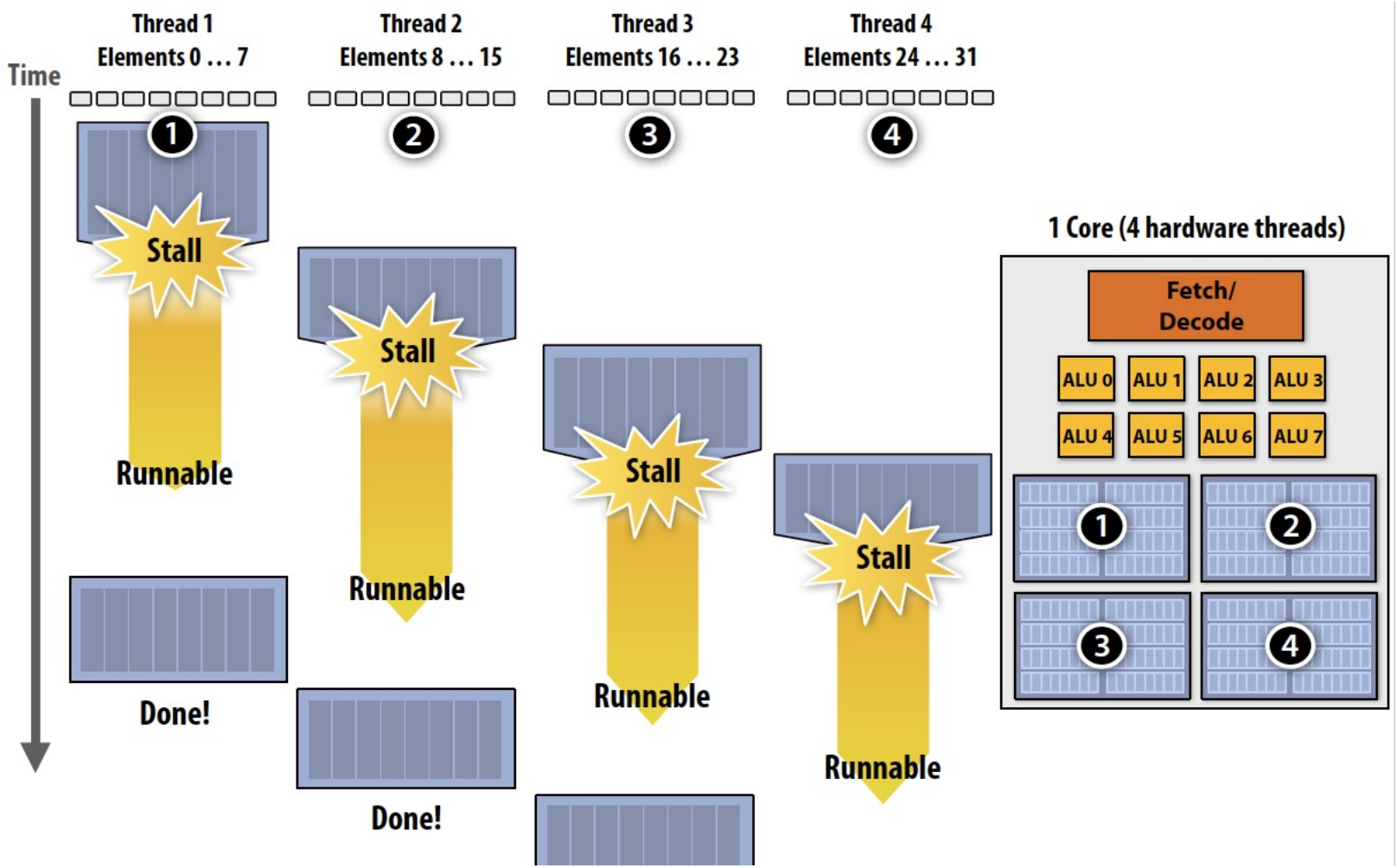


Advantage: increased throughput



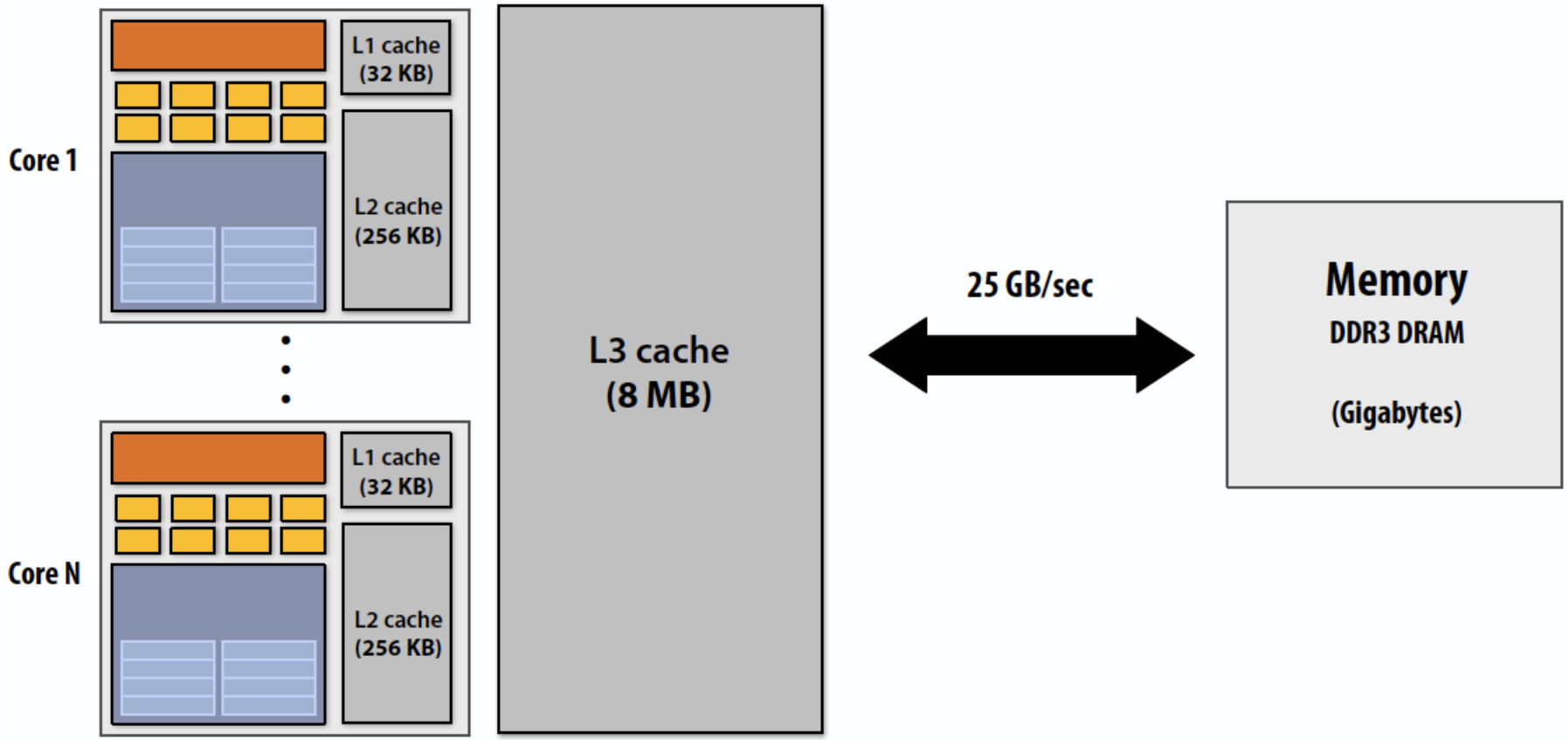


Advantage: increased throughput



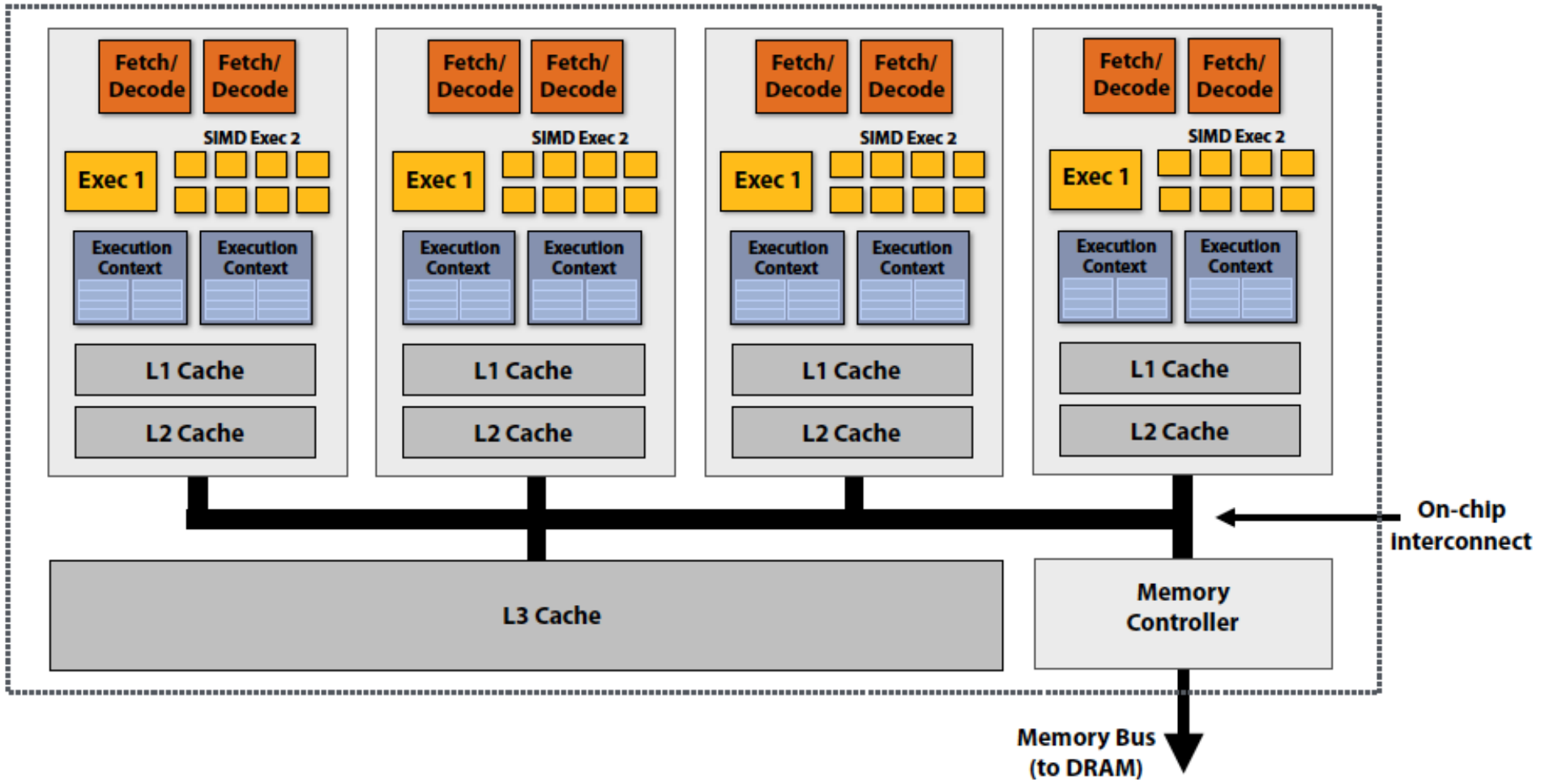
What about the memory?

- Three levels of cache: L1 (separate I$ and D$, per-core), L2 (per-core), L3 (=LLC, shared)



Putting it all together

- A modern CPU has a mix of all these features…



SIMD programming

Vectorization/SIMD options

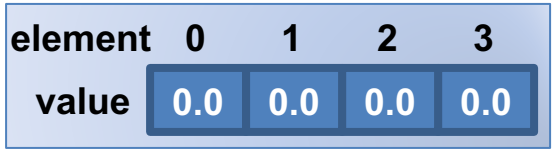
- Auto-vectorization
  - Both gcc and icc have support for it
  - Successful for simple loops and data structures
- Compiler directives
  - Both gcc and icc allow for specific pragma's to enable vectorization
  - Pragma's are used to "force" the compiler to vectorize
- C or C++: **intrinsics**
  - Declare vector variables
  - Name instruction
  - Work on variables, not registers
- Assembly instructions
  - Execute on vector registers

Using intrinsics

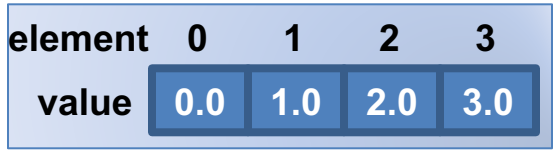
- https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions
- https://software.intel.com/sites/landingpage/IntrinsicsGuide/
- Requirements:
  - Using aligned data structures (aligned to the size of the vector)

Examples of intrinsics

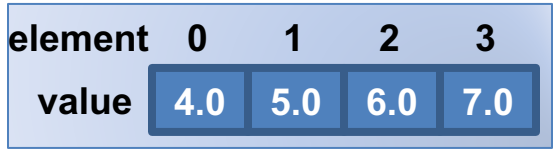
```
float data[1024];
// init: data[0] = 0.0, data[1] = 1.0, data[2] = 2.0, etc.
init(data);

// Set all elements in my vector to zero.
__m128 myVector0 = _mm_setzero_ps();
```




```
// Load the first 4 elements of the array into my vector.
__m128 myVector1 = _mm_load_ps(data);
```

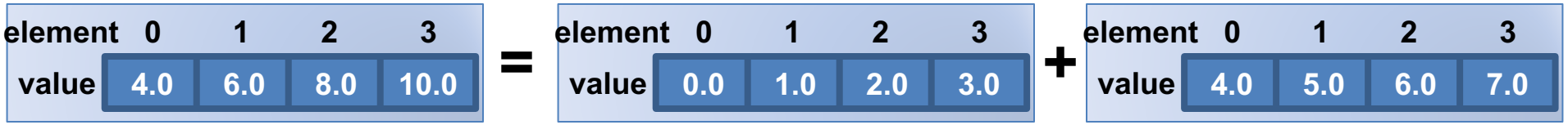



```
// Load the second 4 elements of the array into my vector.
__m128 myVector2 = _mm_load_ps(data+4);
```



Examples of intrinsics

```
// Add vectors 1 and 2; instruction performs 4 FLOP.
__m128 myVector3 = _mm_add_ps(myVector1, myVector2);
```




```
// Multiply vectors 1 and 2; instruction performs 4 FLOP.
__m128 myVector4 = _mm_mul_ps(myVector1, myVector2);
```


```
// _MM_SHUFFLE(w,x,y,z) selects w&x from vec1 and y&z from vec2.
__m128 myVector5 = _mm_shuffle_ps(myVector1, myVector2,
                                  _MM_SHUFFLE(2, 3, 0, 1));
```

Steps for vectorization

- Identify (loop) to vectorize
- Unroll (by the intended SIMD width)
- Use the correct intrinsics to vectorize computation
- Move data from arrays to vectors

Vector add

```
void vectorAdd(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Vector add with SSE: unroll loop

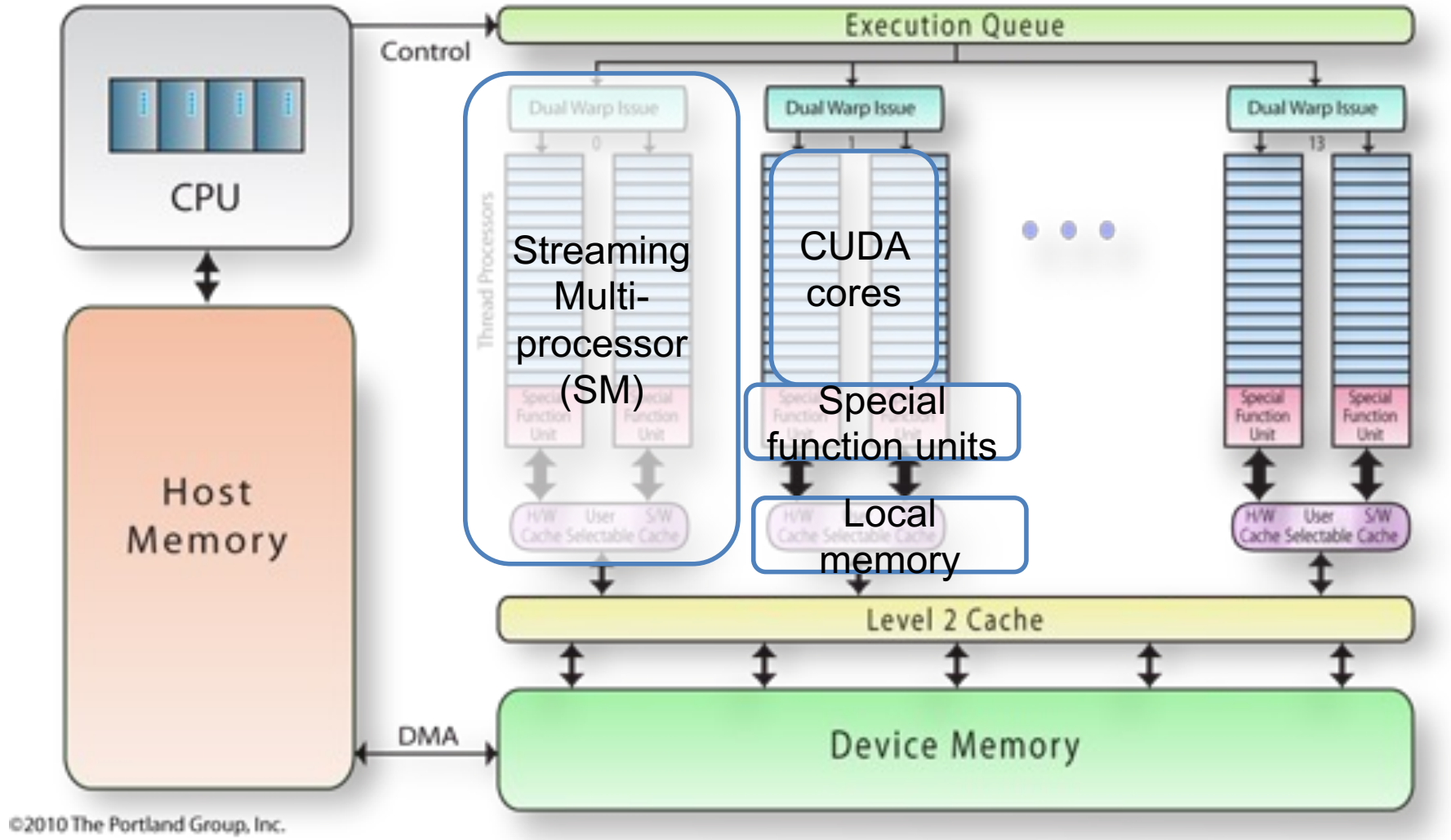
```
void vectorAdd(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i += 4) {
        c[i+0] = a[i+0] + b[i+0];
        c[i+1] = a[i+1] + b[i+1];
        c[i+2] = a[i+2] + b[i+2];
        c[i+3] = a[i+3] + b[i+3];
    }
}
```

Vector add with SSE: vectorize loop

```
void vectorAdd(int size, float* a, float* b, float* c) {
  for(int i=0; i<size; i += 4) {
      __m128 vecA = _mm_load_ps(a + i); // load 4 elts from a
      __m128 vecB = _mm_load_ps(b + i); // load 4 elts from b
      __m128 vecC = _mm_add_ps(vecA, vecB); // add four elts
     _mm_store_ps(c + i, vecC); // store four elts
  }
}
```

Many-core GPUs

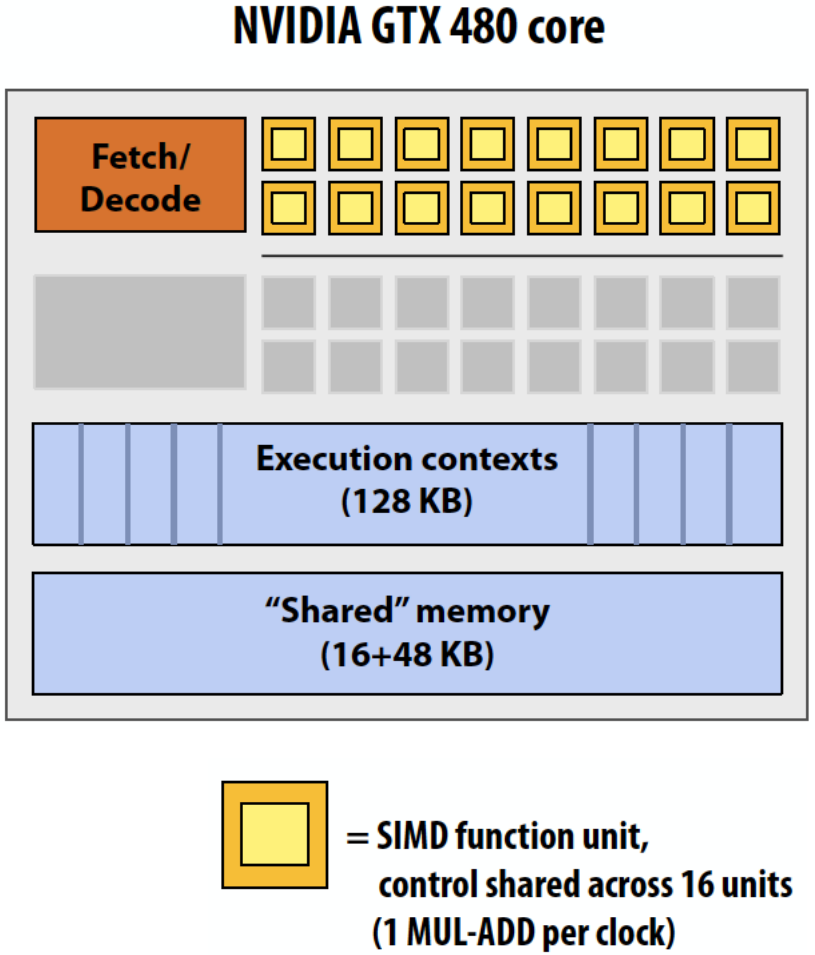
Generic GPU


CPU
Control
Execution Queue
Host Memory
Dual Warp Issue
Dual Warp Issue
Dual Warp Issue
Streaming Multi-processor (SM)
CUDA cores
Special function units
Local memory
Special Function Unit
H/W Cache
User/S/W Selectable Cache
Level 2 Cache
DMA
Device Memory
©2010 The Portland Group, Inc.

… or, using our CPU “symbols”

- Instructions operate on 32 pieces of data at a time (called “warps”).
  - Warp = thread issuing 32-wide vector instructions
- Up to 48 warps are simultaneously interleaved
- Over 1500 elements can be processed concurrently by a core

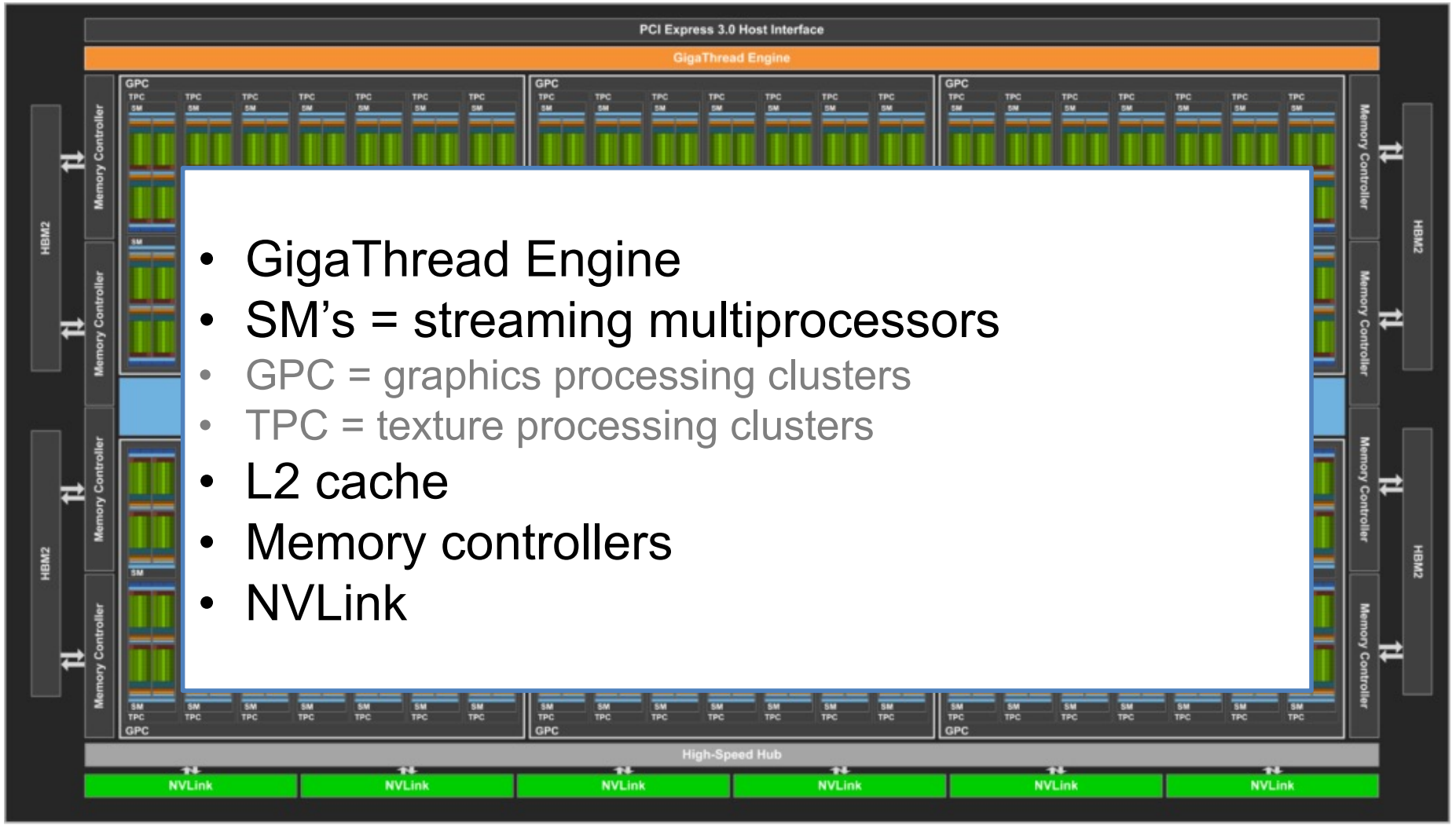
- Full board: 15 cores (SMs)!



*Diagrams adapted from CMU’s course "Parallel Computer Architecture and Programming” – http://15418.courses.cs.cmu.edu/spring2016/lectures

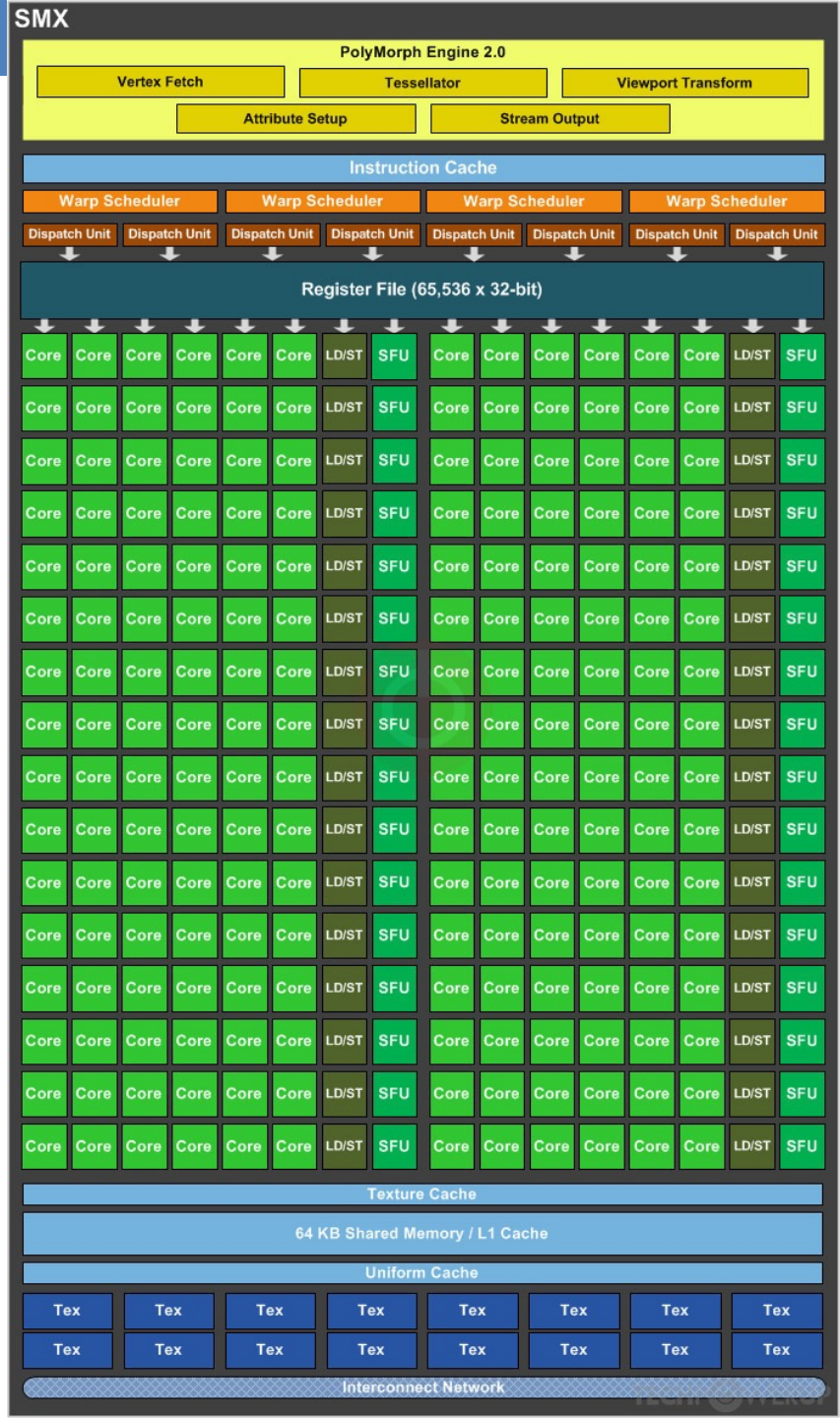
Inside an NVIDIA GPU architecture

Inside an NVIDIA GPU architecture



Inside a Streaming Multiprocessor

- Different types of cores
  - CUDA Cores (INT/FP32)
  - LD/ST
  - Special function units
- Register file
- Warp scheduler
- Data caches
- Instruction buffers/caches
- Texture units



Maxwell

More features …

- Different types of cores
  - Adding: DP Units (Pascal)
  - Adding: Tensor units (Volta)

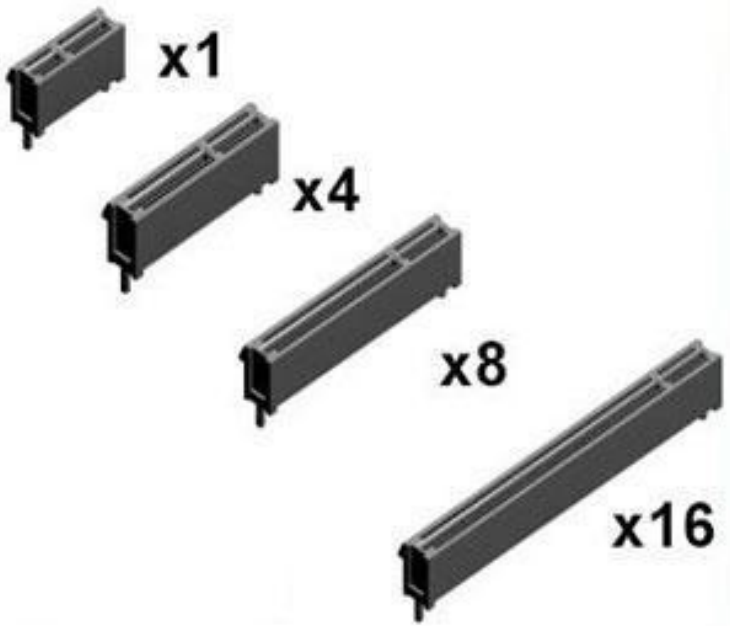
<Pascal

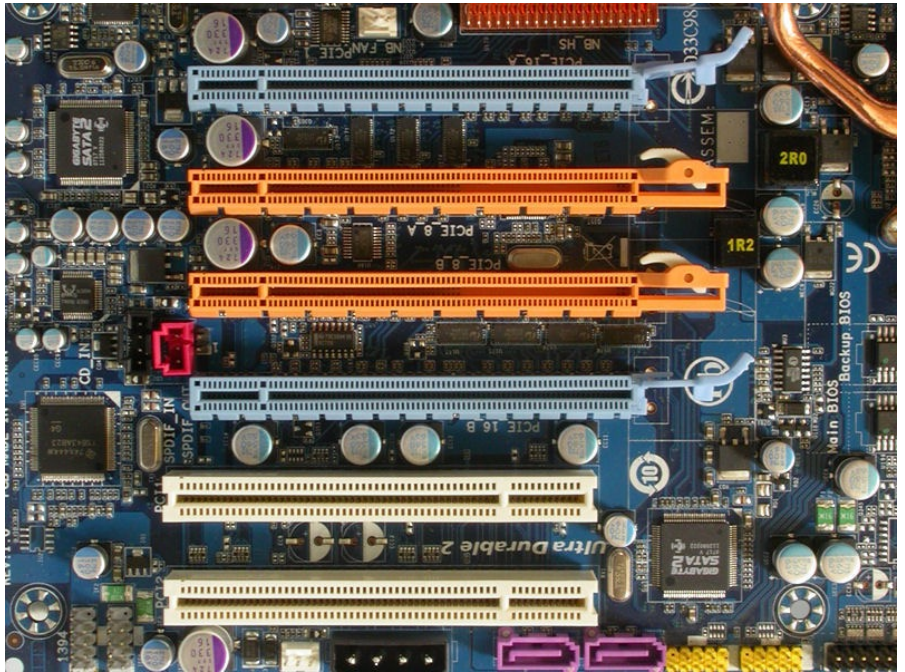
Volta>

GPU Integration into the host system

- Typically based on a PCI Express bus
- Transfer speed (effectively, CPU-to-GPU):

  16 GT/s per lane x 16 lanes
- Can be NVLink (~10x faster) for specialized motherboards



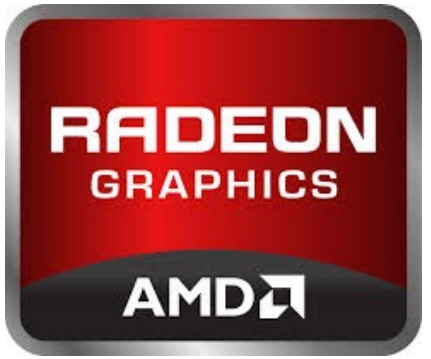


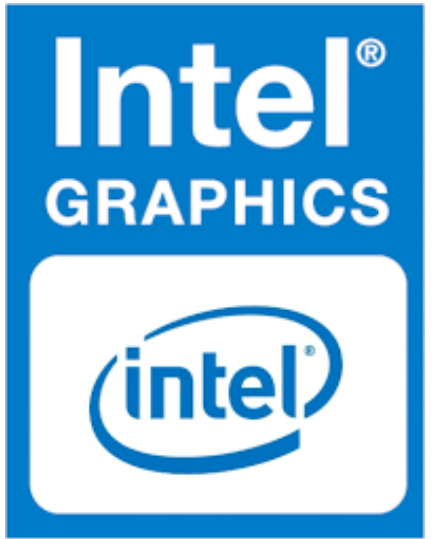
NVIDIA GPUs (8+ years)

	Fermi	Kepler	Maxwell	Pascal	Volta
GPU	GTX480	GK180	GM200	GP100	GV100
Compute capability (CC)	2.x	3.5	5.2	6.0	7.0
SMs	16	15	24	56	80
TPCs	16	15	24	28	40
FP32 Cores / SM	32	192	128	64	64
FP64 “Cores” / SM	4	64	4	32	32
Clock[MHz]	700	875	1114	1480	1530
Peak FP32 [TFLOPs]	1.35	5.04	6.8	10.6	15.7
Peak FP64 [TFLOPs]	0.168	1.68	.21	5.3	7.8

Other players on the market

- **AMD** (former ATI)
  - Much better performance
  - Programmed using OpenCL (standard!)
  - Poorer software drivers and infrastructure (so far)
  - A lot less libraries and tools
  - Much smaller community effort
- **arm** (formerly ARM ☺)
  - Low-power devices (mobile platforms mostly)
  - Programmed using OpenCL
  - Lower performance than ATI and Intel, by choice
- **Intel**
  - To support own CPUs with integrated graphics
  - Programmed using OpenCL





All GPUs …

- Have a similar architecture
  - Massively parallel
  - Simple cores
  - Complex memory system
- Are programmed in a similar way
  - Fine-grain (SIMD/SIMT) parallelism
- Programming models ?
  - OpenCL is the de-facto standard for GPU programming
  - Lots of efforts for C++
  - Many other libraries and models on top of CUDA / OpenCL

GPU Levels or Parallelism

- **Data parallelism** (fine-grain)
  - Write 1 thread, instantiate a lot of them
  - SIMT (Single Instruction Multiple Thread) execution
    - Many threads execute concurrently
      - Same instruction
      - Different data elements
      - HW automatically handles divergence
    - Not same as SIMD because of multiple register sets, addresses, and flow paths*
  - Hardware multithreading
    - HW resource allocation & thread scheduling
      - Excess of threads to hide latency
      - Context switching is (basically) free
- **Task parallelism is "emulated"** (coarse-grain)
  - Hardware mechanisms exist
  - Specific programming constructs to execute multiple tasks.
- **Heterogeneous** computing
  - CPU is always present …

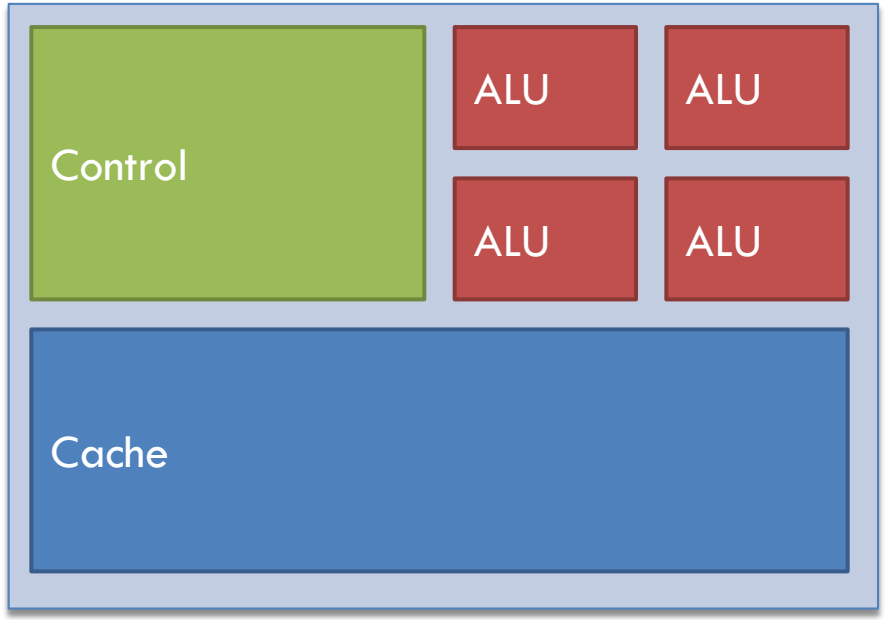
*http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

GPUs vs CPUs

Why so different?

- Different goals produce different designs!
  - CPU must be good at everything
  - GPUs focus on massive parallelism
    - Less flexible, more specialized
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, etc.)
  - multithreading can hide latency => no big caches
  - share control logic across many threads

CPU vs. GPU

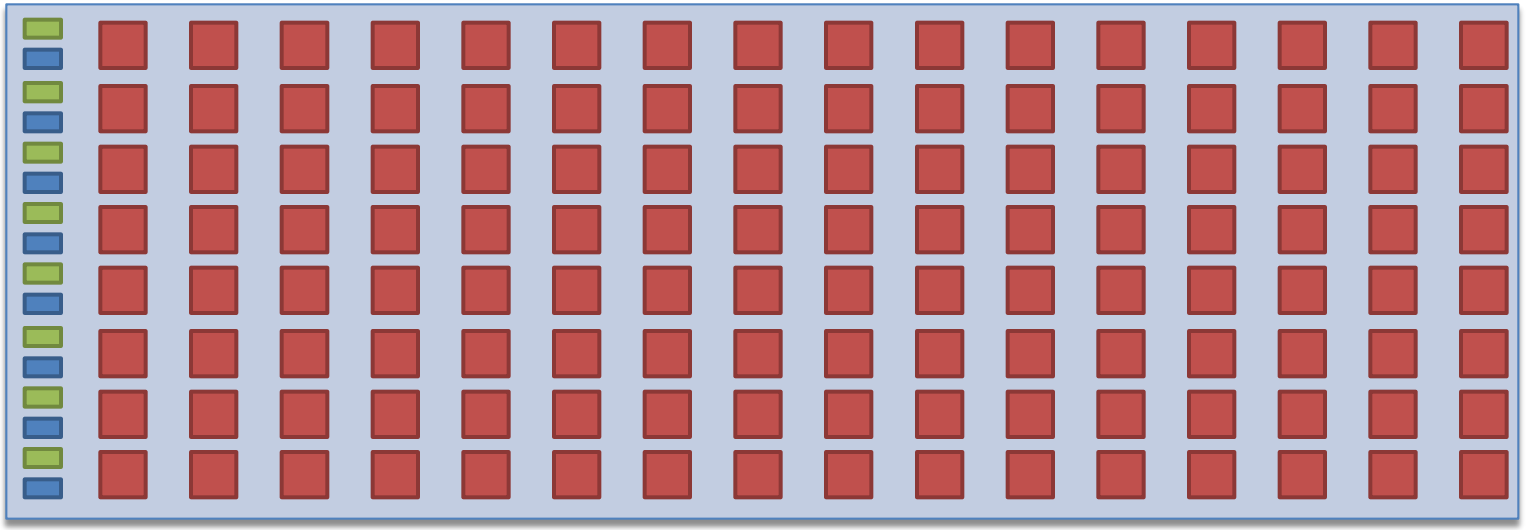


CPU

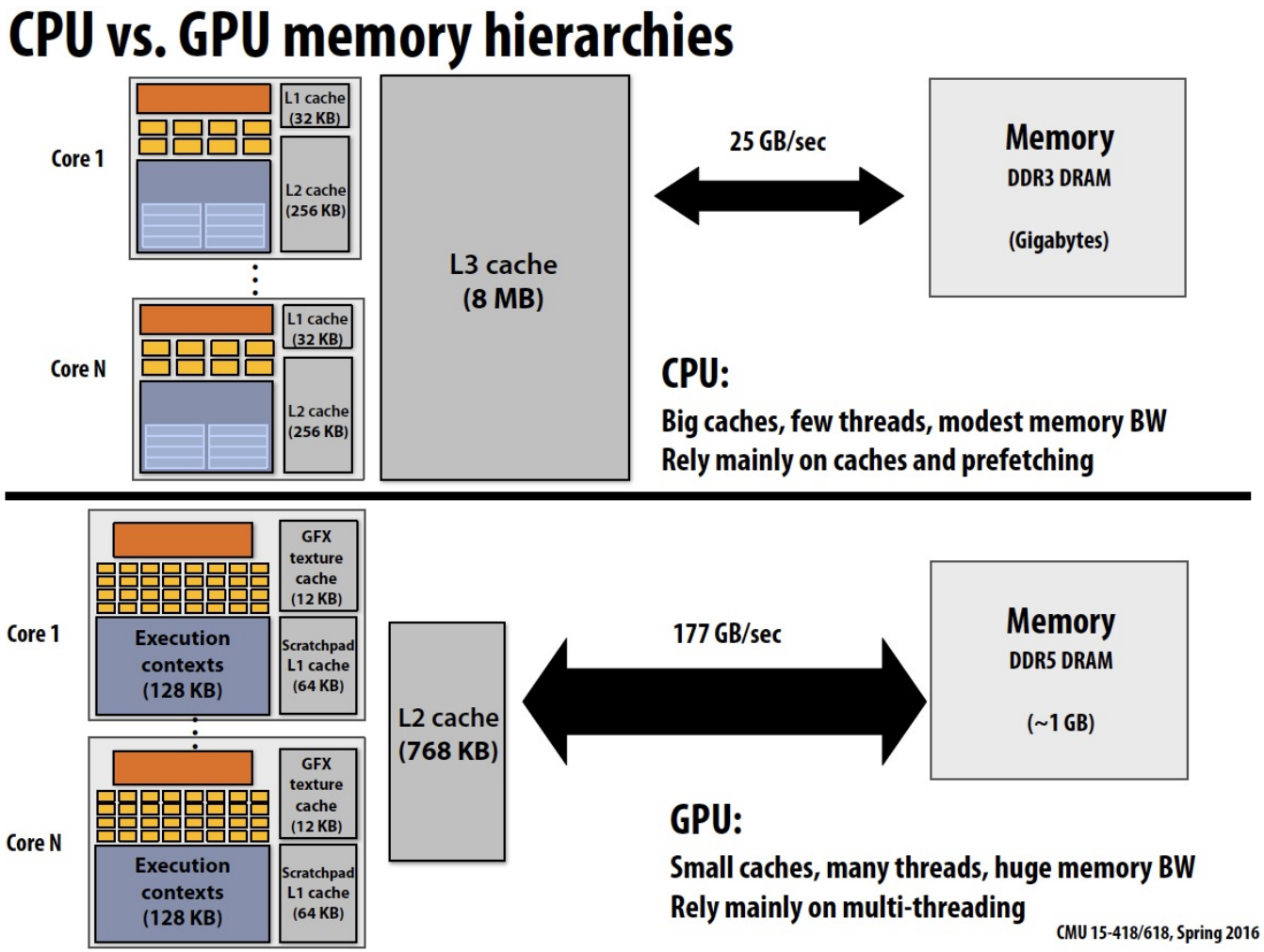
Low latency, high flexibility.
Excellent for irregular codes with limited parallelism.

GPU

High throughput.
Excellent for massively parallel workloads.



CPU vs GPU


CPU vs. GPU memory hierarchies
Core 1
L1 cache (32 KB)
L2 cache (256 KB)
Core N
L1 cache (32 KB)
L2 cache (256 KB)
L3 cache (8 MB)
25 GB/sec
Memory
DDR3 DRAM
(Gigabytes)
CPU:
Big caches, few threads, modest memory BW
Rely mainly on caches and prefetching
Core 1
GFX texture cache (12 KB)
Execution contexts (128 KB)
Scratchpad L1 cache (64 KB)
Core N
GFX texture cache (12 KB)
Execution contexts (128 KB)
Scratchpad L1 cache (64 KB)
L2 cache (768 KB)
177 GB/sec
Memory
DDR5 DRAM
(~1 GB)
GPU:
Small caches, many threads, huge memory BW
Rely mainly on multi-threading
CMU 15-418/618, Spring 2016

CPU vs. GPU: the movie

- The Mythbusters
  - Jamie Hyneman & Adam Savage
  - Discovery Channel
- Appearance at NVIDIA's NVISION 2008:

https://www.youtube.com/watch?v=-P28LKWTzrI

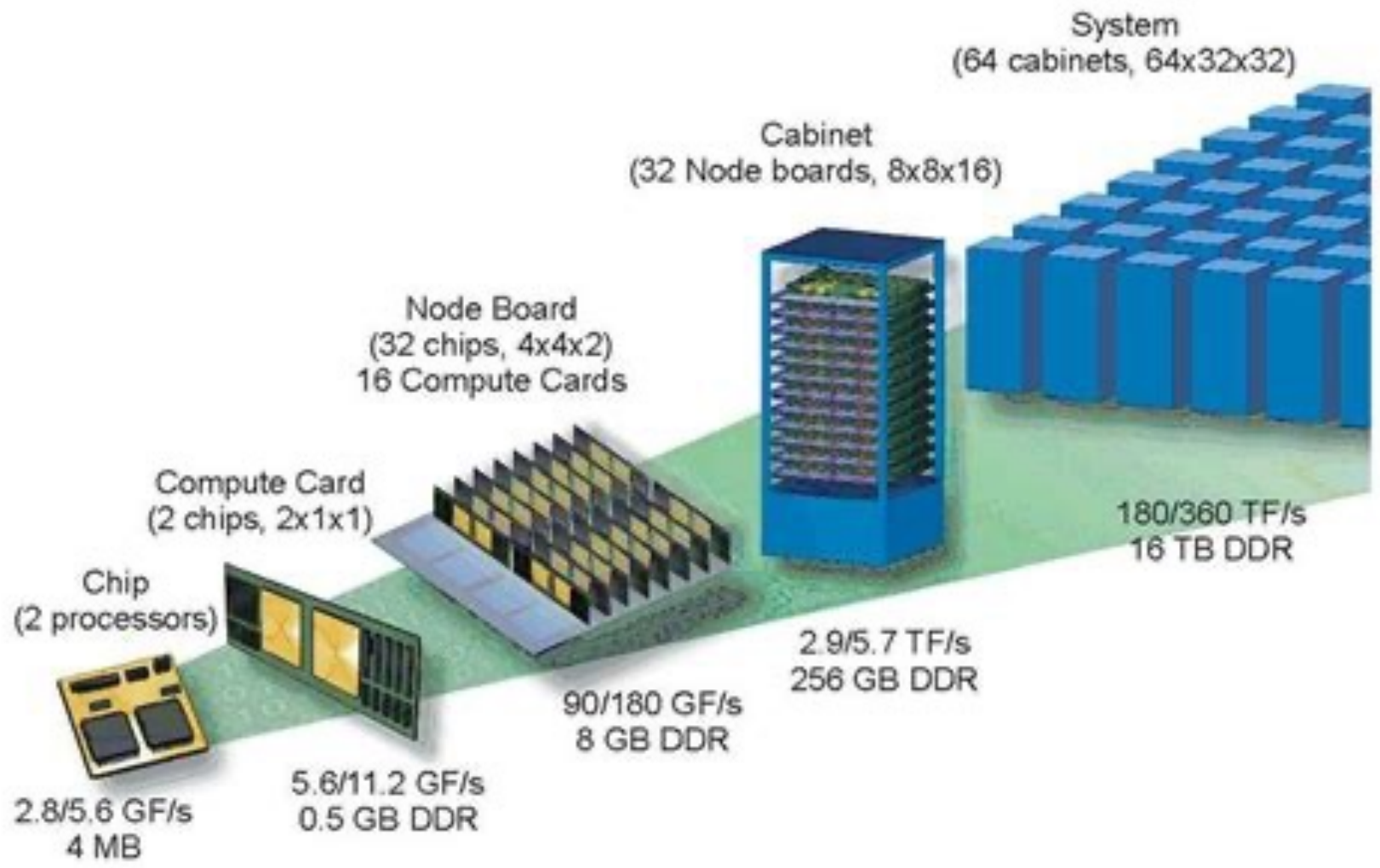
Scaling out: Multi-node systems => supercomputers

Putting it all together

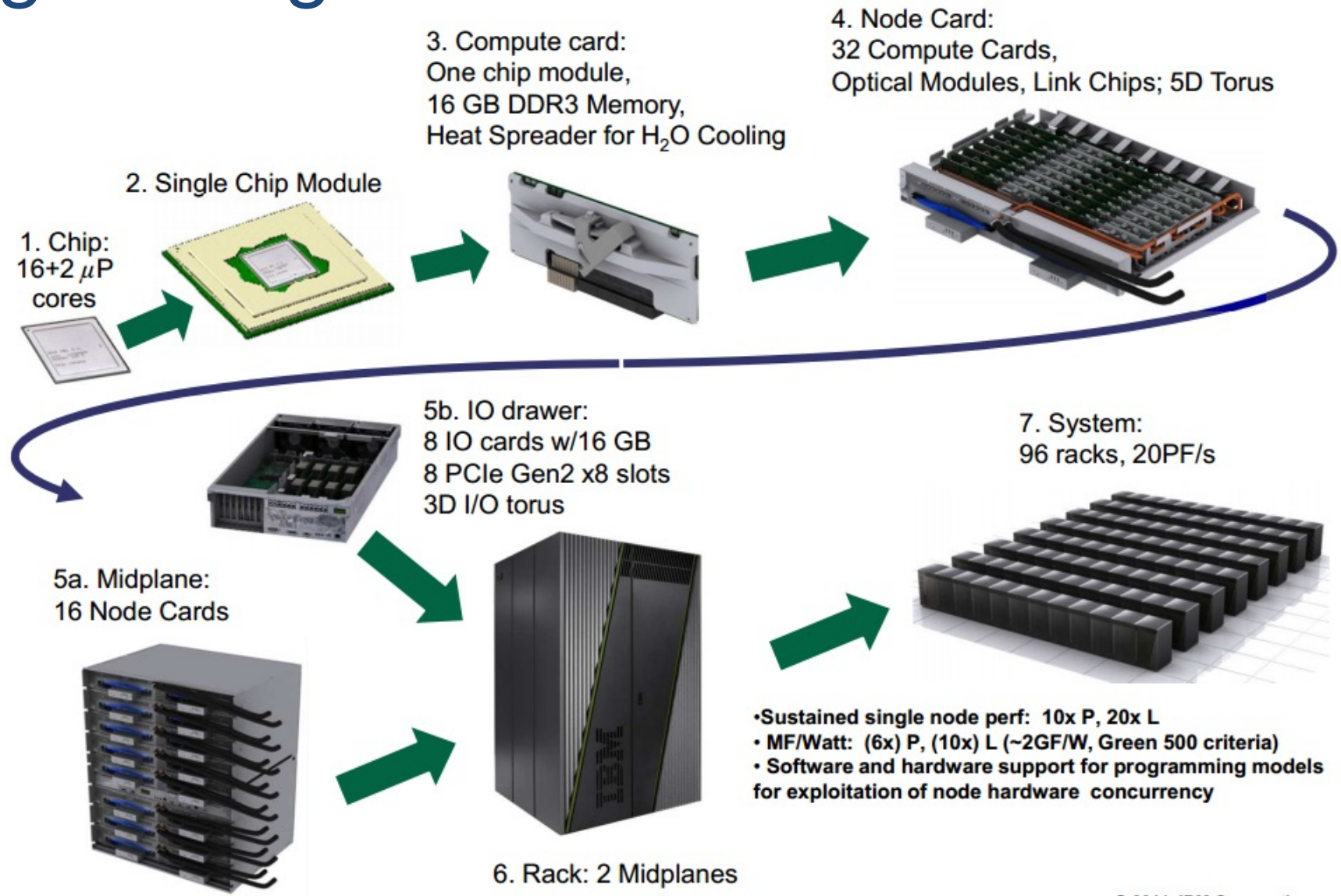
- Multiple nodes
  - Potentially grouped/clustered in islands
- Communication network
  - Latency & throughput differences compared to intra-node
- Homogeneous vs. Heterogeneous

- Peak Performance: summing all up
- Energy consumption: summing it all up

Putting it all together: IBM's BLUGENE/L



Putting it all together: IBM's BlueGene/Q



Putting it all together: FUGAKU

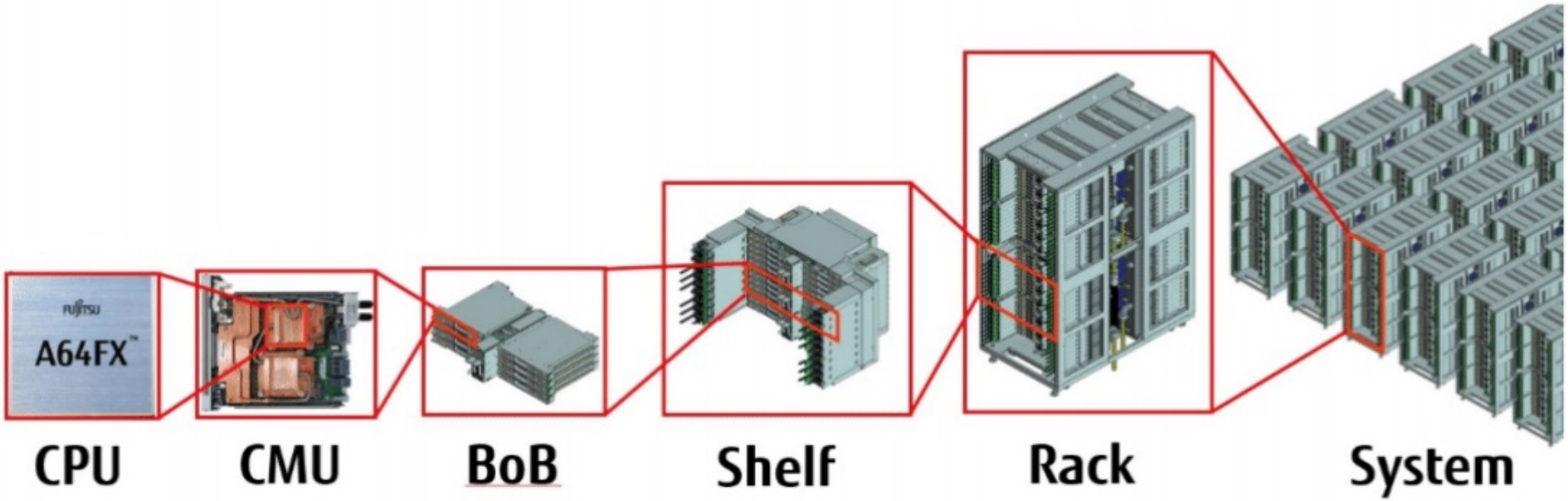


Figure 4. System Configuration

Putting it all together: SUMMIT

Summit Overview

OpenPOWER

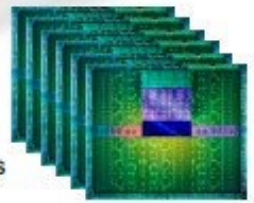
Components

IBM POWER9

- 22 Cores
- 4 Threads/core
- NVLink

NVIDIA GV100

- 7 TF
- 16 GB @ 0.9 TB/s
- NVLink



Compute Node

2 x POWER9
6 x NVIDIA GV100
NVMe-compatible PCIe 1600 GB SSD

25 GB/s EDR IB- (2 ports)
512 GB DRAM- (DDR4)
96 GB HBM- (3D Stacked)
Coherent Shared Memory

Compute Rack

18 Compute Servers
Warm water (70°F direct-cooled components)
RDHX for air-cooled components

39.7 TB Memory/rack
55 KW max power/rack

Compute System

10.2 PB Total Memory
256 compute racks
4,608 compute nodes
Mellanox EDR IB fabric
200 PFLOPS
~13 MW

GPFS File System

250 PB storage
2.5 TB/s read, 2.5 TB/s write

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Zoom-out: Parallel machine models

(Parallel) Systems Models

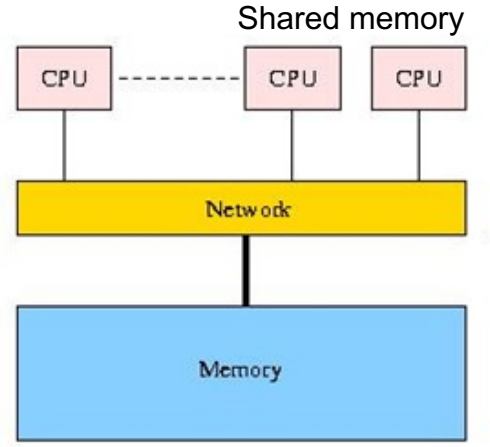
- Why do we need parallel system models?
  - Provide an abstraction of the real machine
  - Dictate the properties of “dedicated” programming models
  - Enable the selection of an appropriate programming model

- Organization-based classification
  - Shared Memory
  - Distributed Memory
  - Virtual shared Memory
  - Hybrids

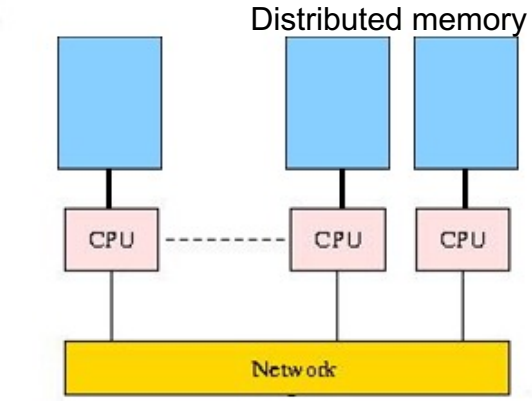
- Processing-based classification
  - Single/Multi Instruction, Single/Multi Data (items)

Parallel Machine Models

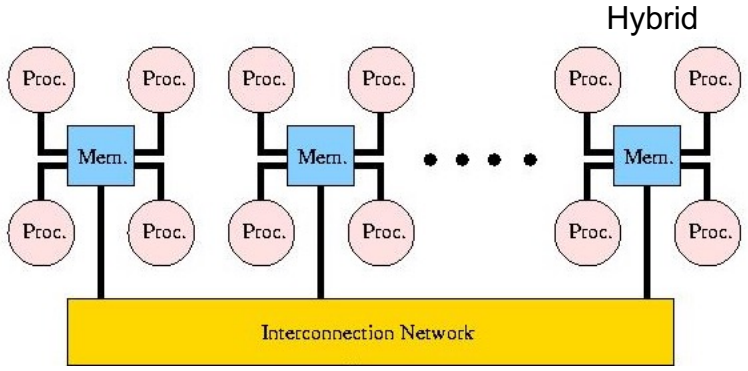
- Shared Memory
  - Multiple compute nodes
  - One single shared address space
  - Typical example: multi-cores



- Distributed Memory
  - Multiple compute nodes
  - Multiple, local (disjoint) address spaces
  - **Virtual shared memory:** software/hardware layer “emulates” shared memory
  - Typical example: clusters



- Hybrids
  - Multiple compute nodes, typically heterogeneous
  - Mixed address space(s), some shared, some global memory
  - Typical example: supercomputers



Parallel Machine Models

- Shared Memory

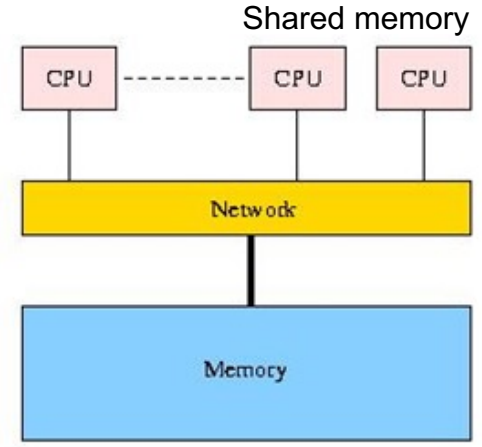
  Programming: multi-threading
  Programming models: OpenMP, pthreads, TBB, …

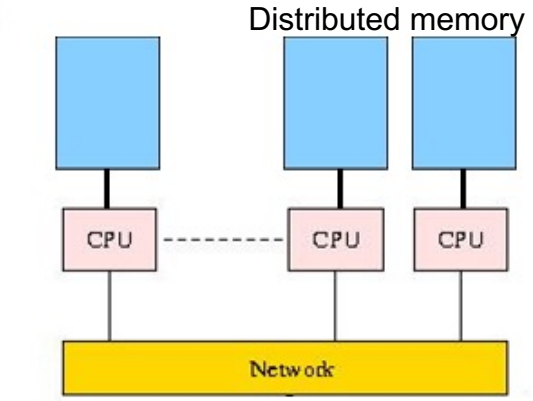
- Distributed Memory

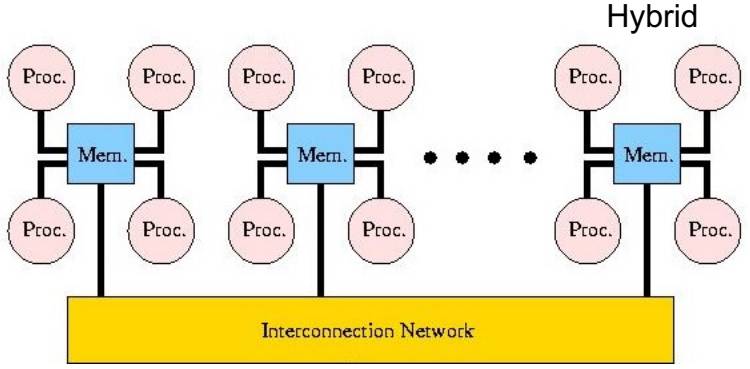
  Programming: message passing
  Programming models: MPI, Big-data models, …

- Hybrids

  Programming: very diverse, depending on the hardware configuration







Examples

- Multi-core CPUs ?
  - Shared memory with respect to system memory
  - Hybrid when taking caches into account
- Clusters ?
  - Distributed memory
  - Could be shared if middleware for virtual shared space is provided
- Supercomputers ?
  - Usually hybrid
- GPUs ?
- Architectures with GPUs?
  - Distributed for traditional, off-chip GPUs
  - Shared for new APUs

Major issues

- Shared Memory model
  - Scalability problems (interconnect)
  - Programming challenge: RD/WR Conflicts
- Distributed Memory model
  - Data distribution is mandatory
  - Programming challenge: remote accesses, consistency
- Virtual Shared Memory model
  - Significant virtualization overhead
  - Easier programming
- Hybrid models
  - Local/remote data more difficult to trace

PART 2: IN SUMMARY…

Parallelism in Computer Systems

- Multi-core and many-cores are the current building blocks of supercomputers
- Accelerators are here to stay
  - … and annoy us during programming …
- Supercomputers are distributed combinations (see previous talks) of multi- and many-cores

- Programming these systems is a mix of programming models
- For efficiency, we need to understand their archtiectures