TRANSFORMERS: AGE OF PARALLEL MACHINES

(a biased introduction to computer architecture for supercomputing)

Ana Lucia Varbanescu, University of Twente, NL

a.l.varbanescu@utwente.nl

UNIVERSITY OF TWENTE.

Agenda (ambitious)

- ~~Part 1 : The anatomy of supercomputers~~
- ~~Part 2 : What's in a name node?~~
- Part 3 : Diversity in parallelism
- Part 4 : One more word about performance
- Part 5 : Summary and beyond
  - Famous last words …

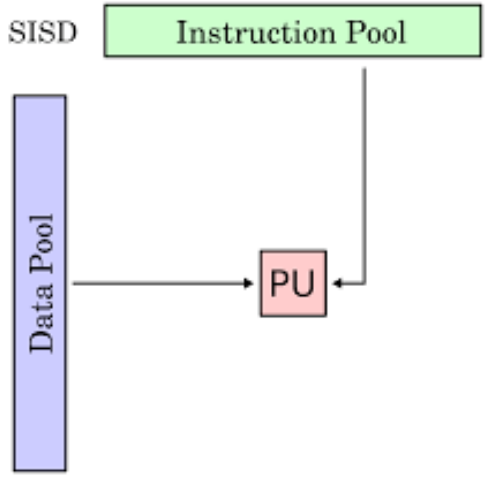
"Larry, do you remember where we buried our hidden agenda?"

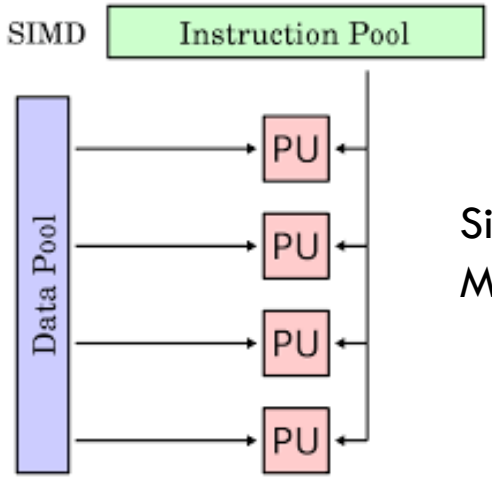
PART 3: PARALLELISM DIVERSITY

Different parallelism models from hardware to software

First taxonomy: Michael Flynn (1966)

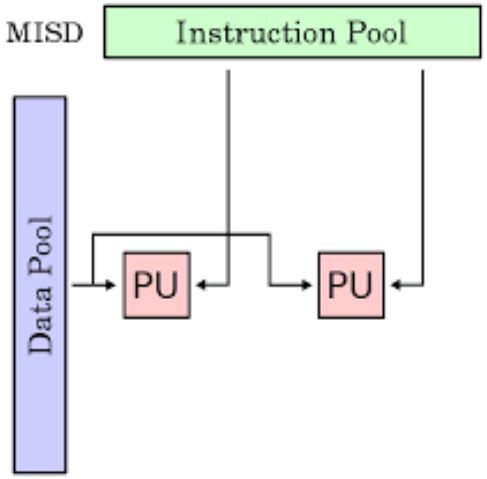
Single Instruction
Single Data

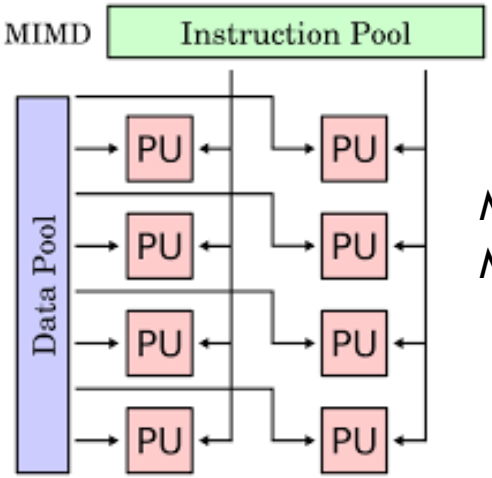




Single Instruction
Multiple Data

Multiple Instructions
Single Data



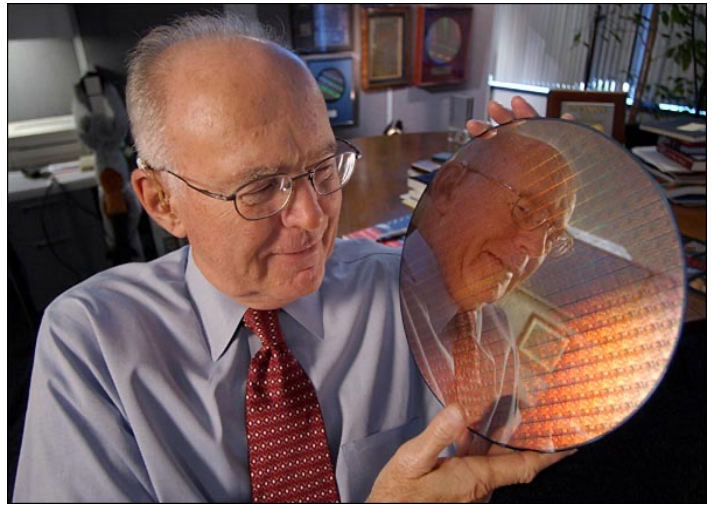


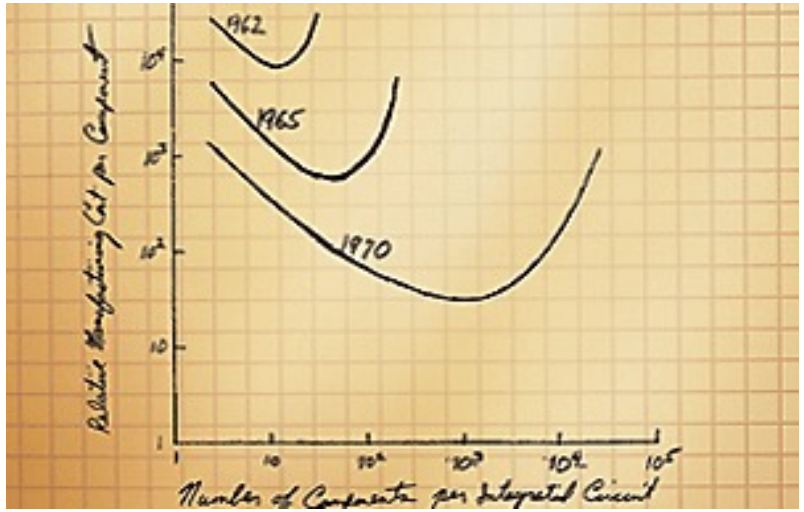
Multiple Instructions
Multiple Data

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.





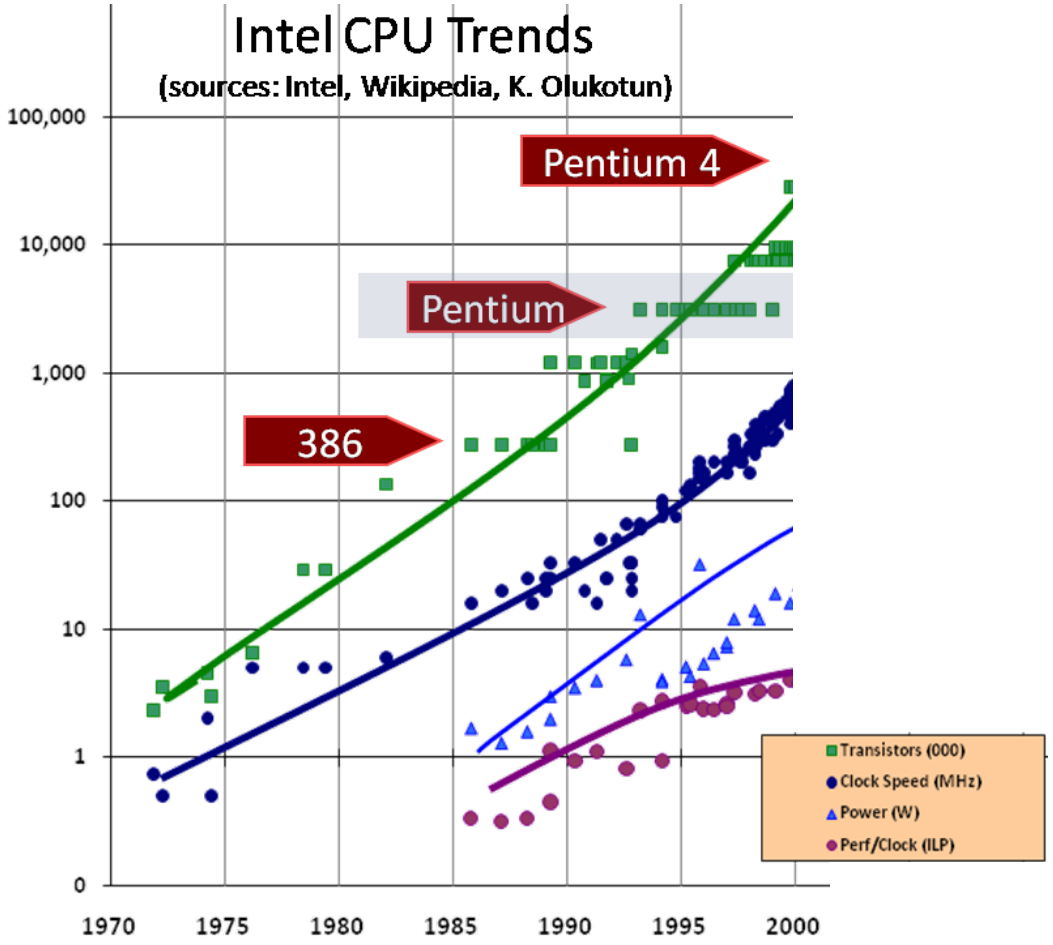
"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase...." Electronics Magazine 1965

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 …

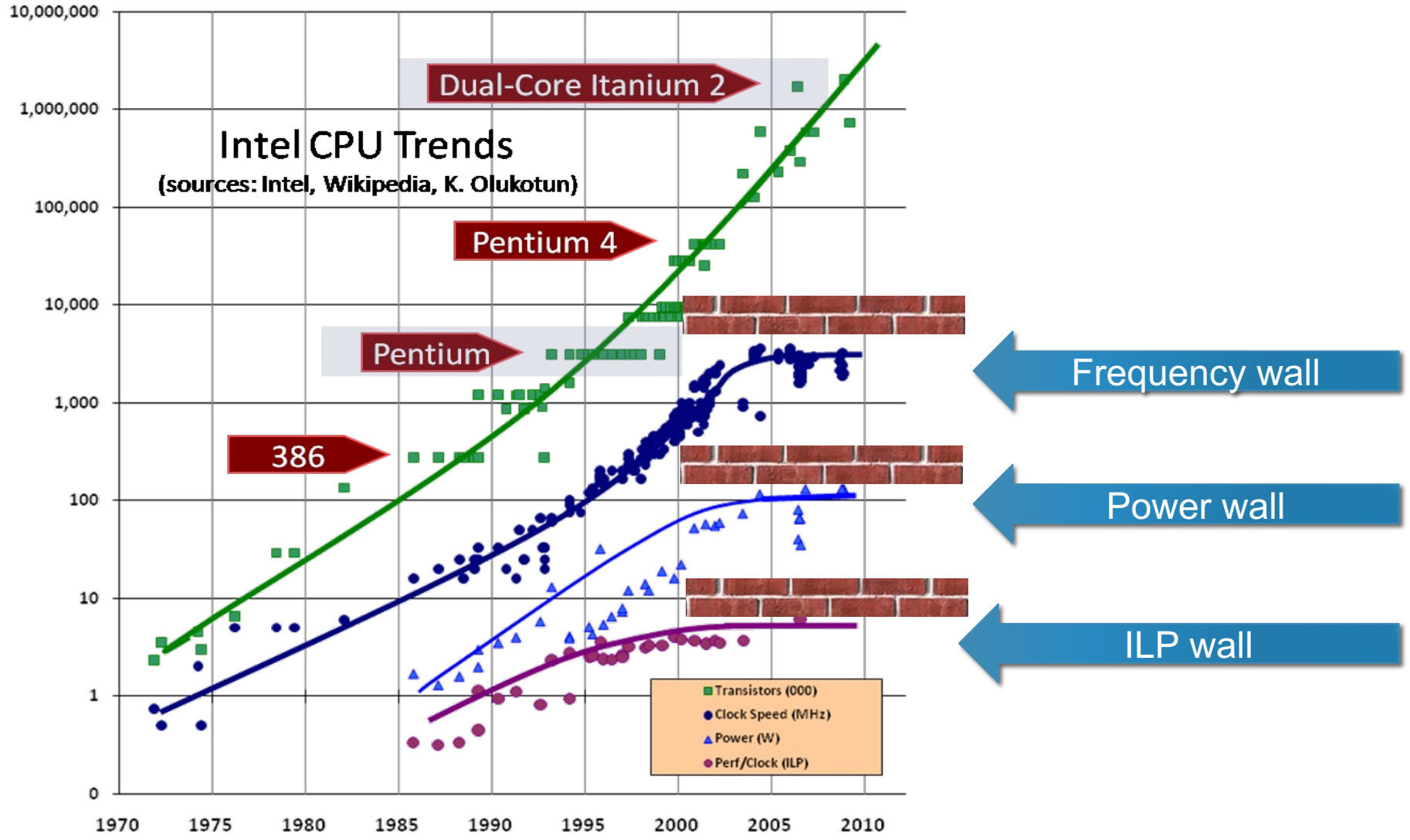
  - Ideally, we want power consumption to be low

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”

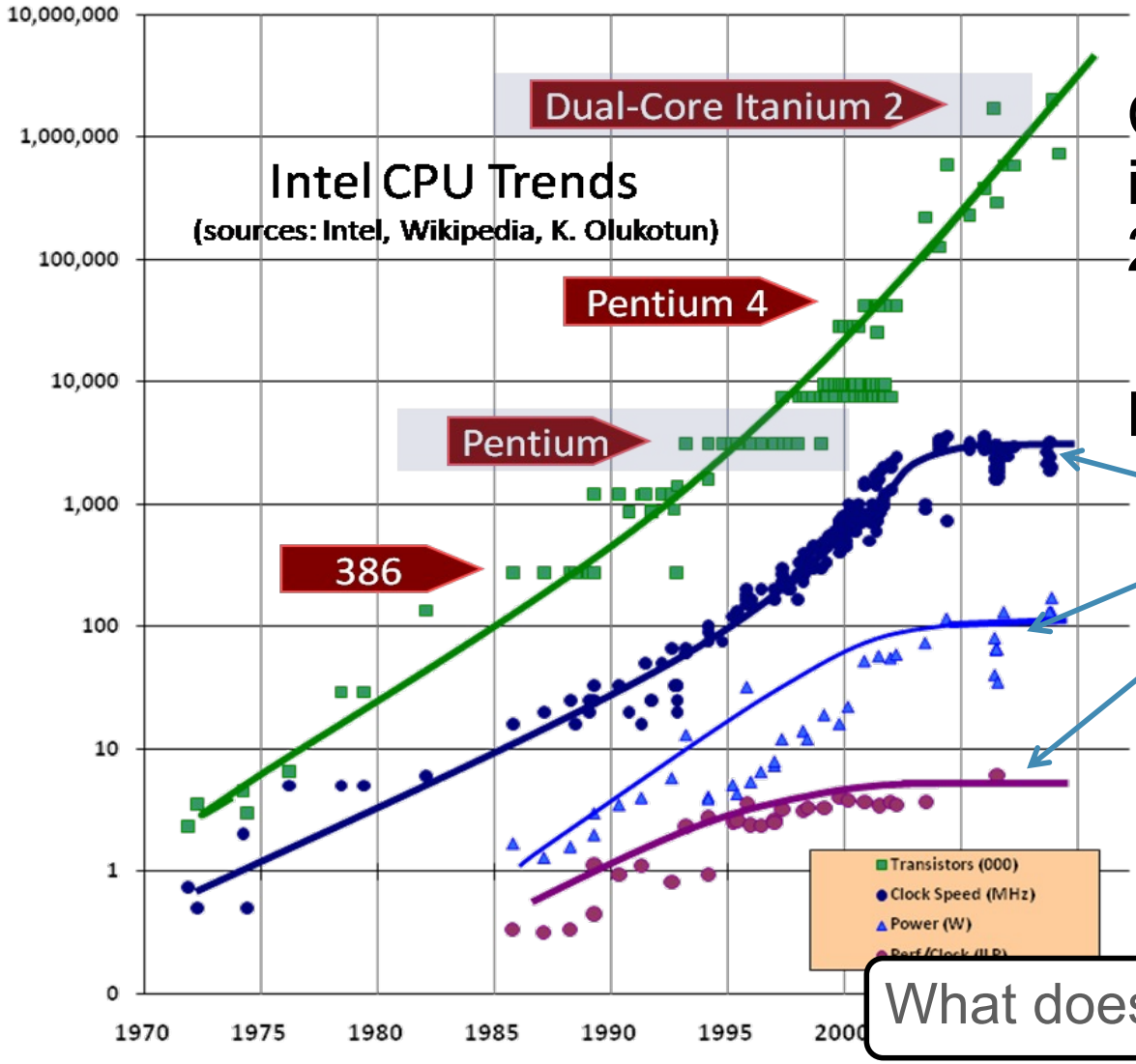


Single core performance scaling

- 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

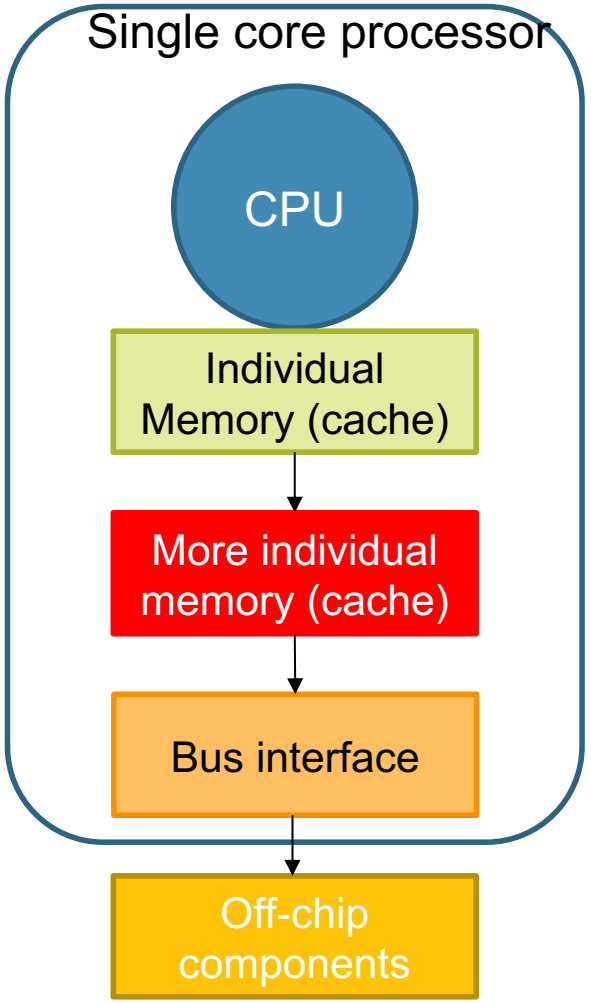
BUT

- **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.

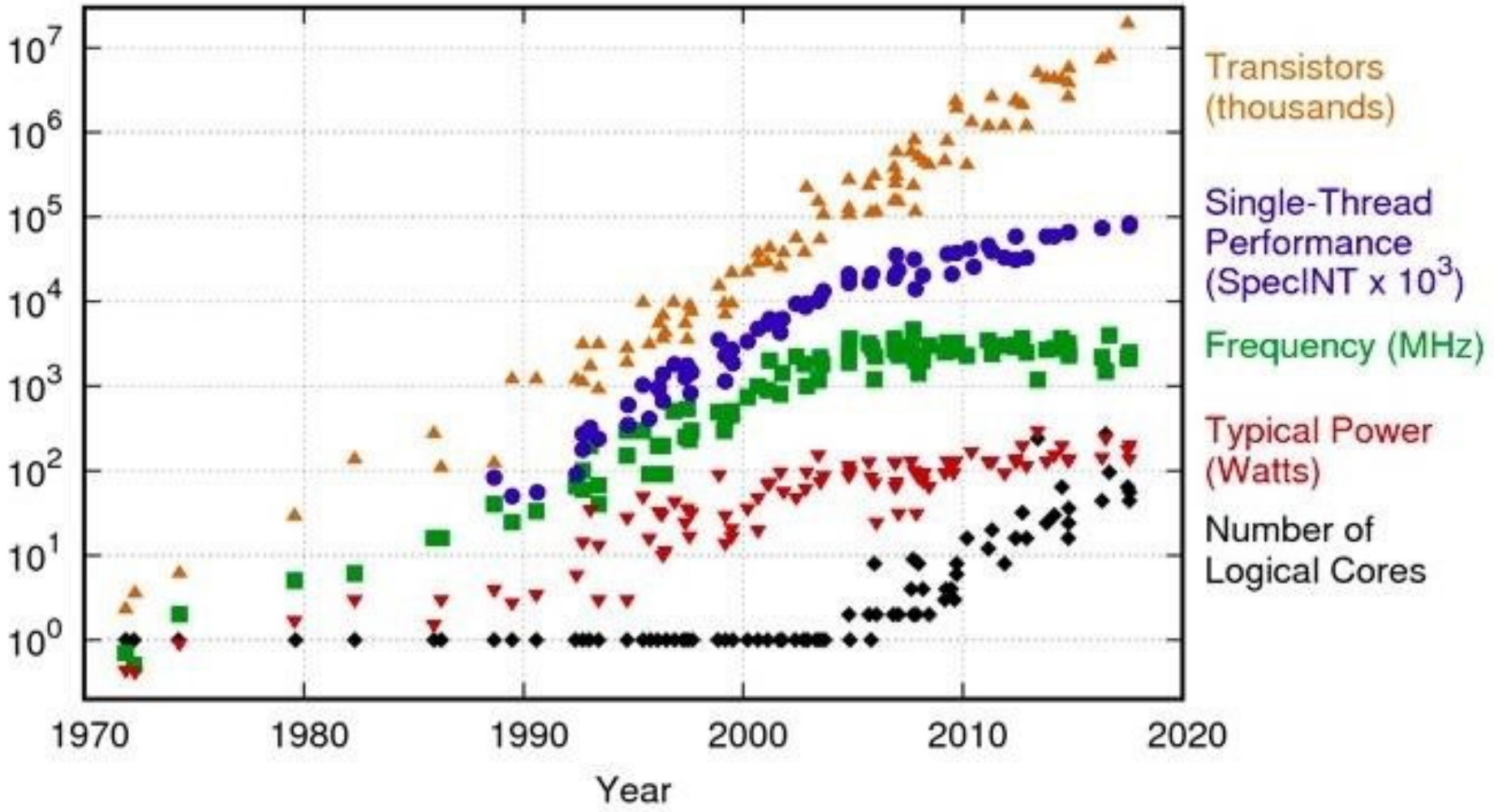


Not anymore!
We no longer gain performance by “growing” sequential processors …

New ways to use transistors

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

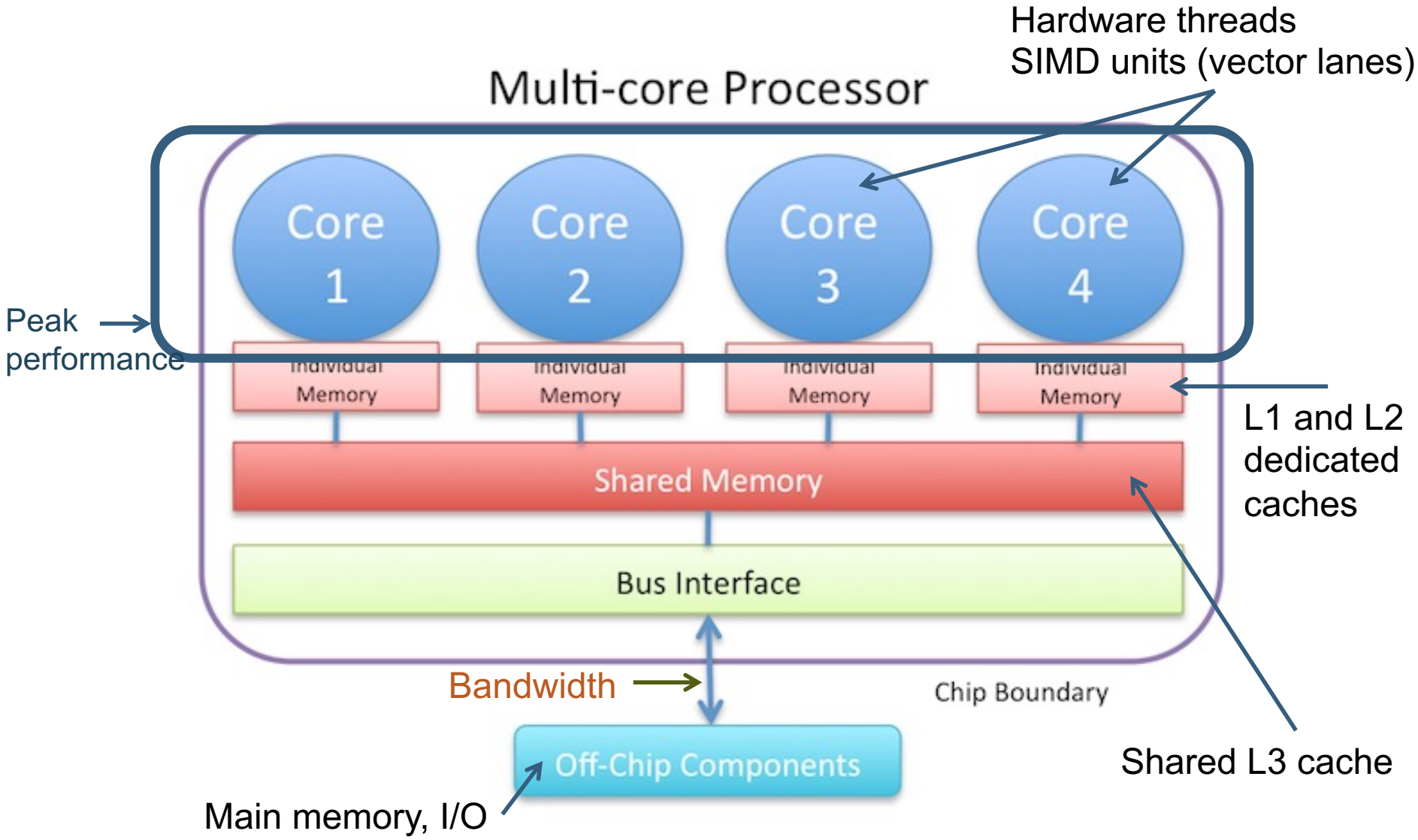
The shift to multi-core



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

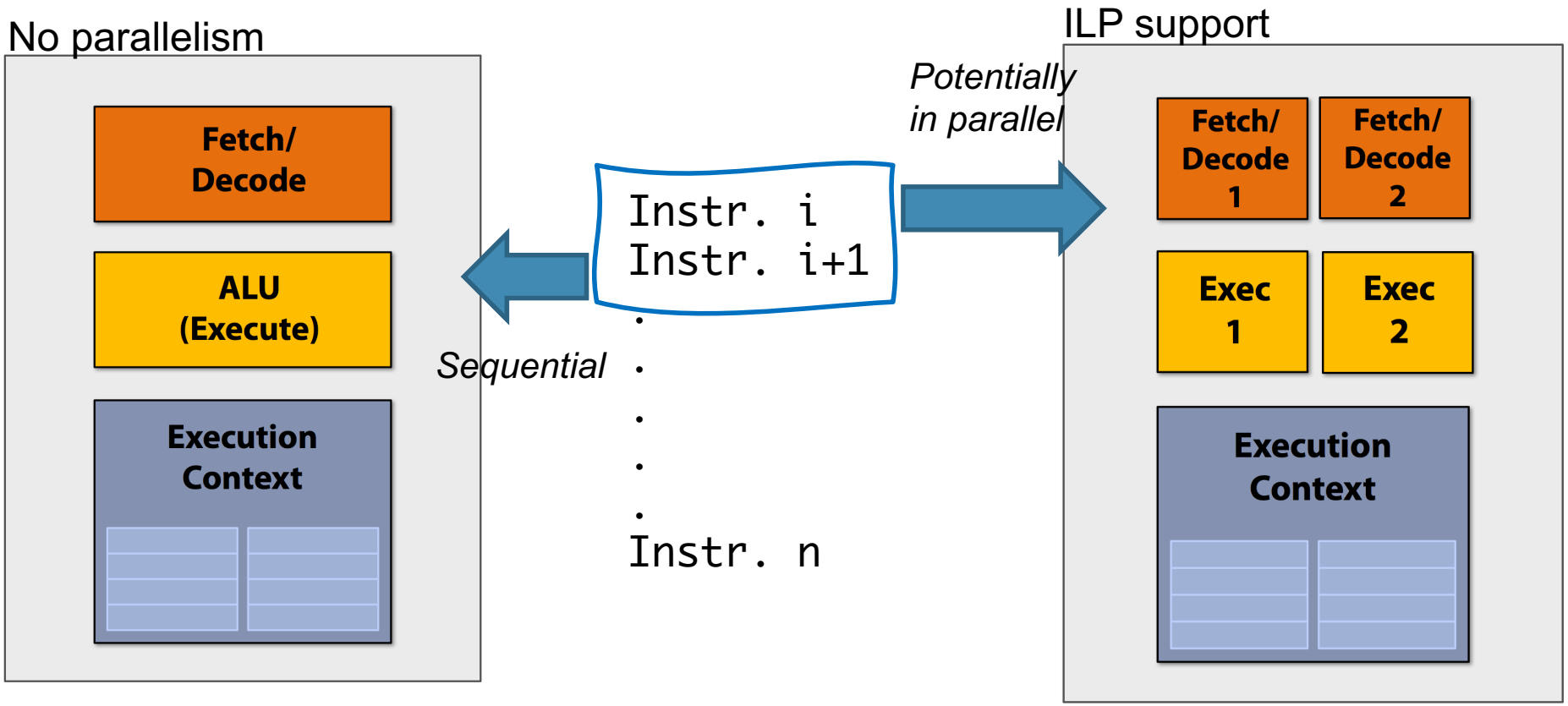


CPU levels of parallelism

- **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!

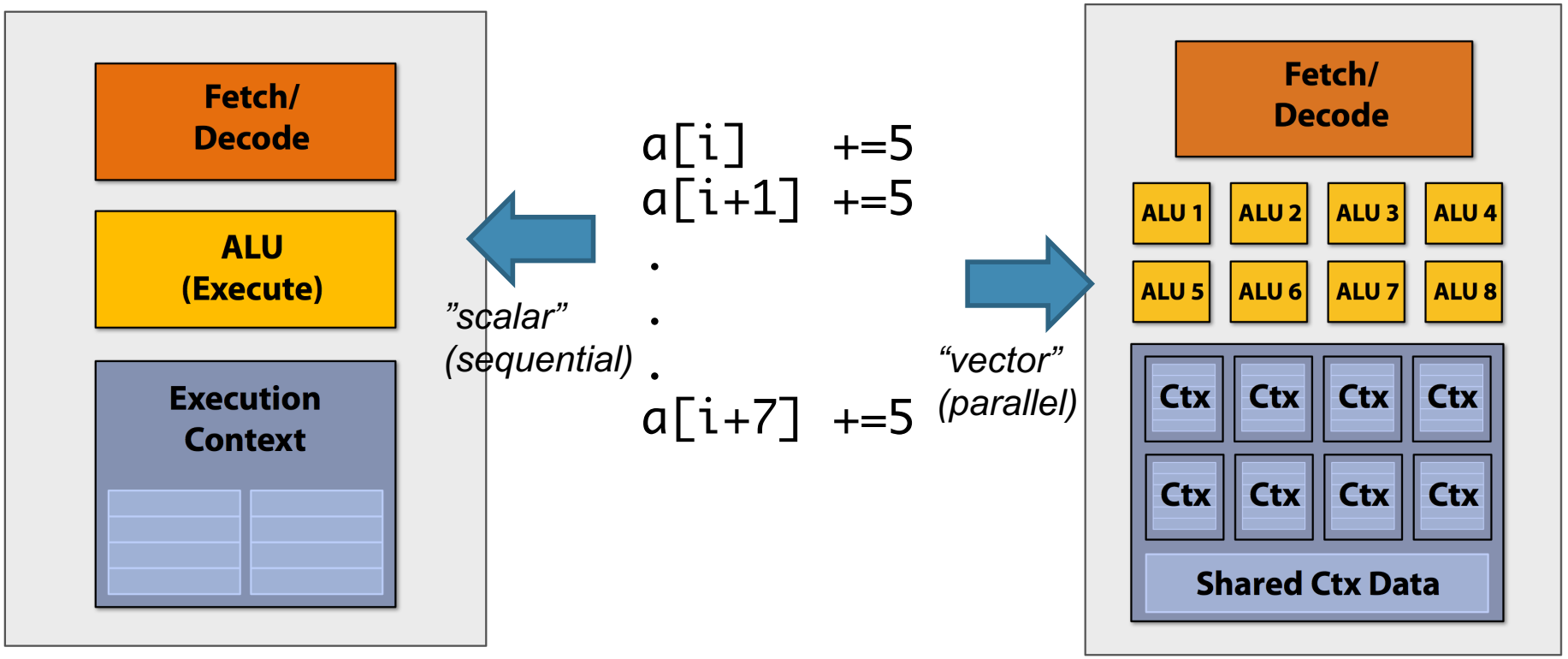
Implementing ILP

- 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

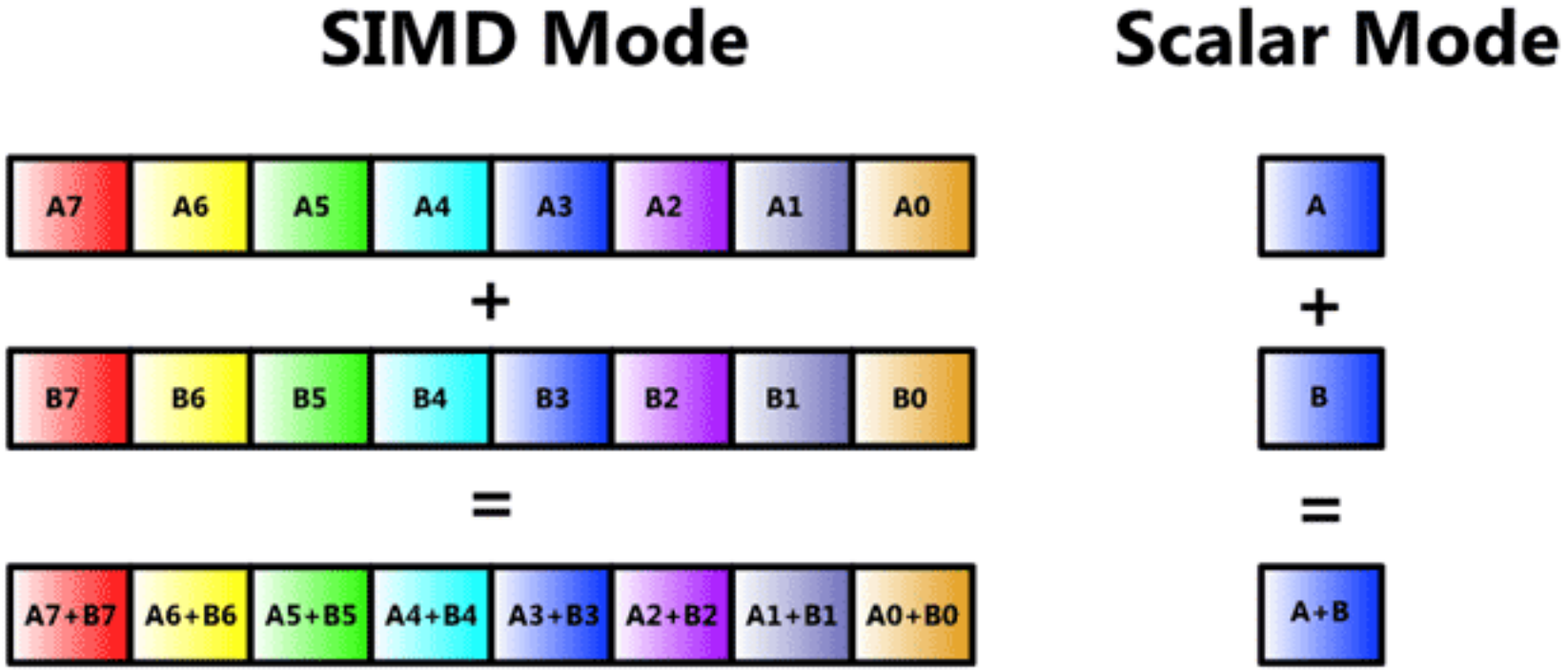
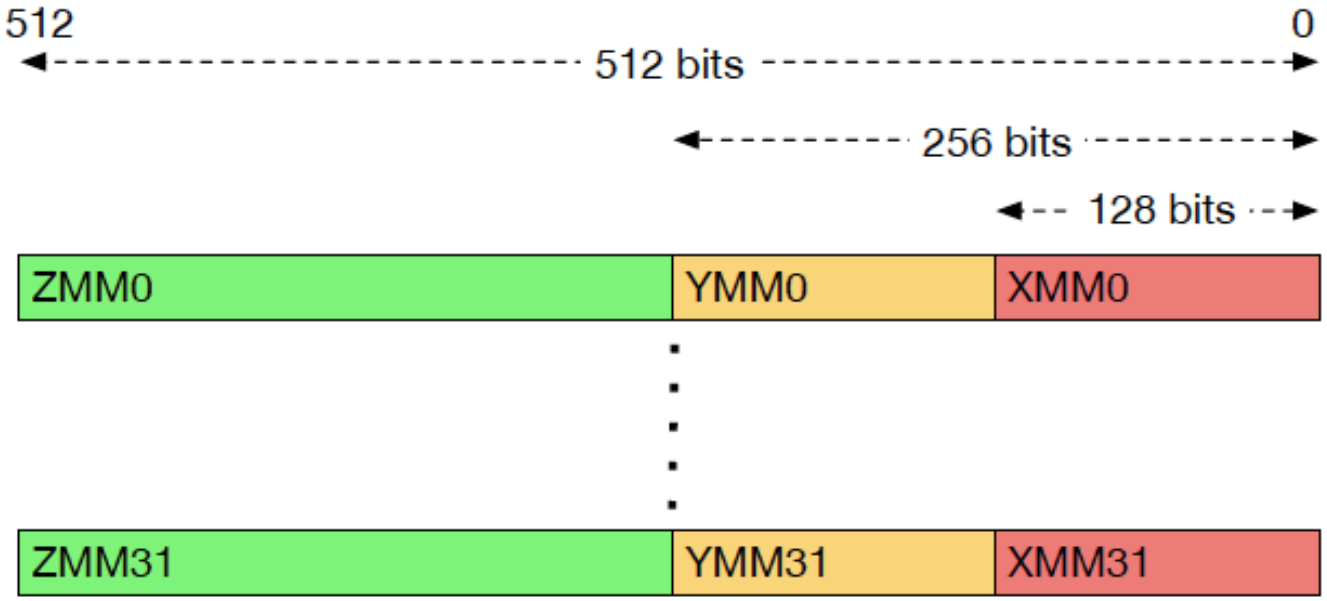


Image: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html

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)



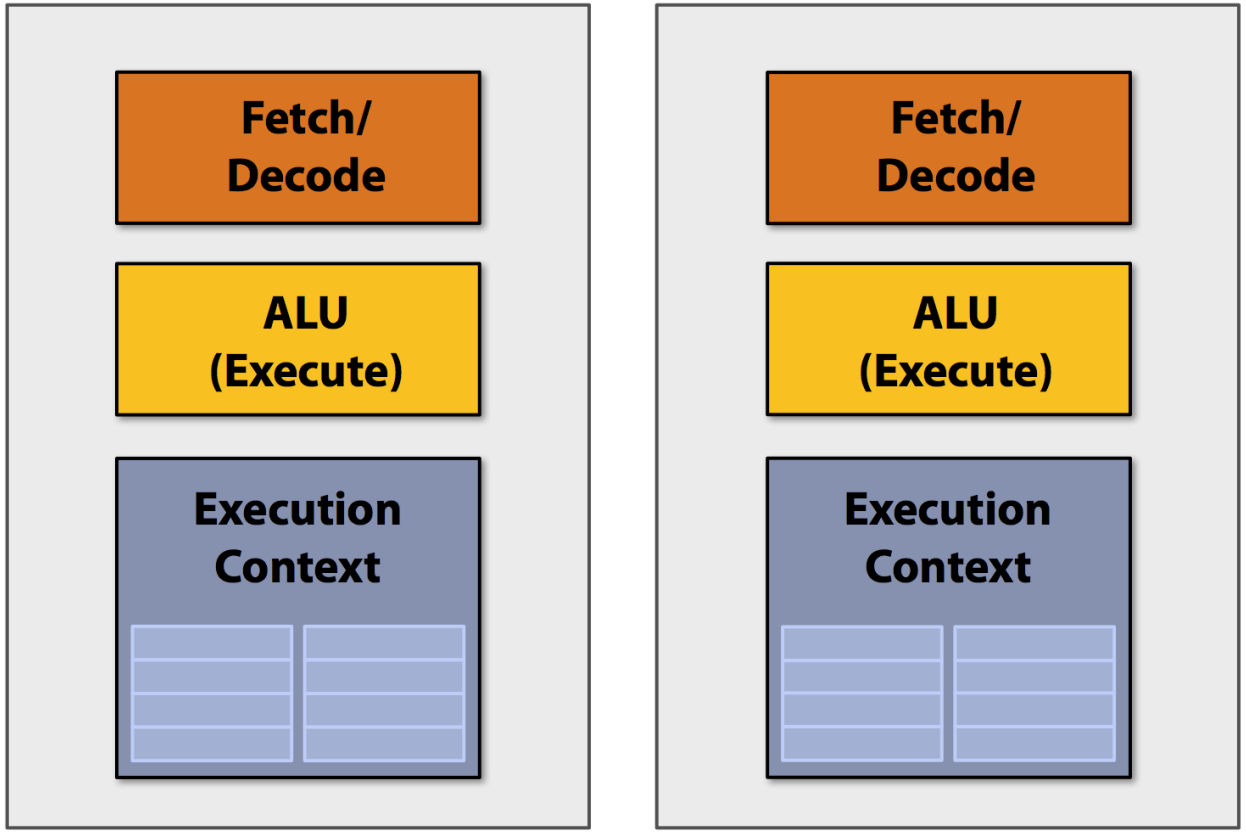
SIMD programmer intervention

- 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

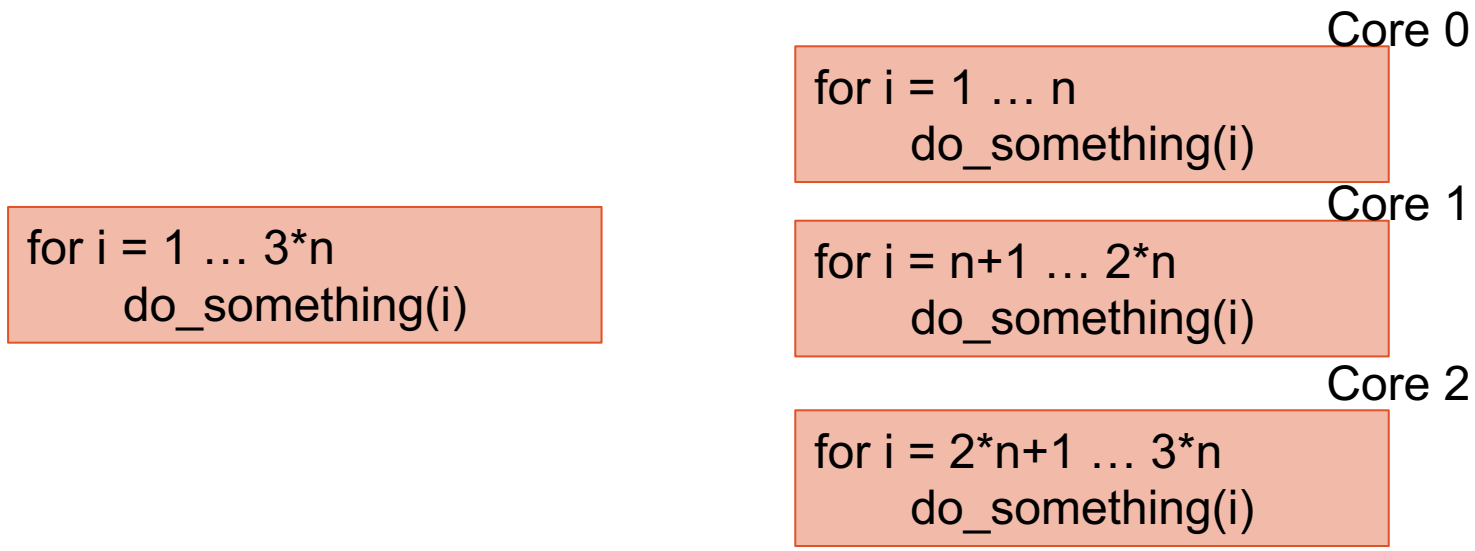
- Two (or more cores) to execute different streams of instructions.



*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



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

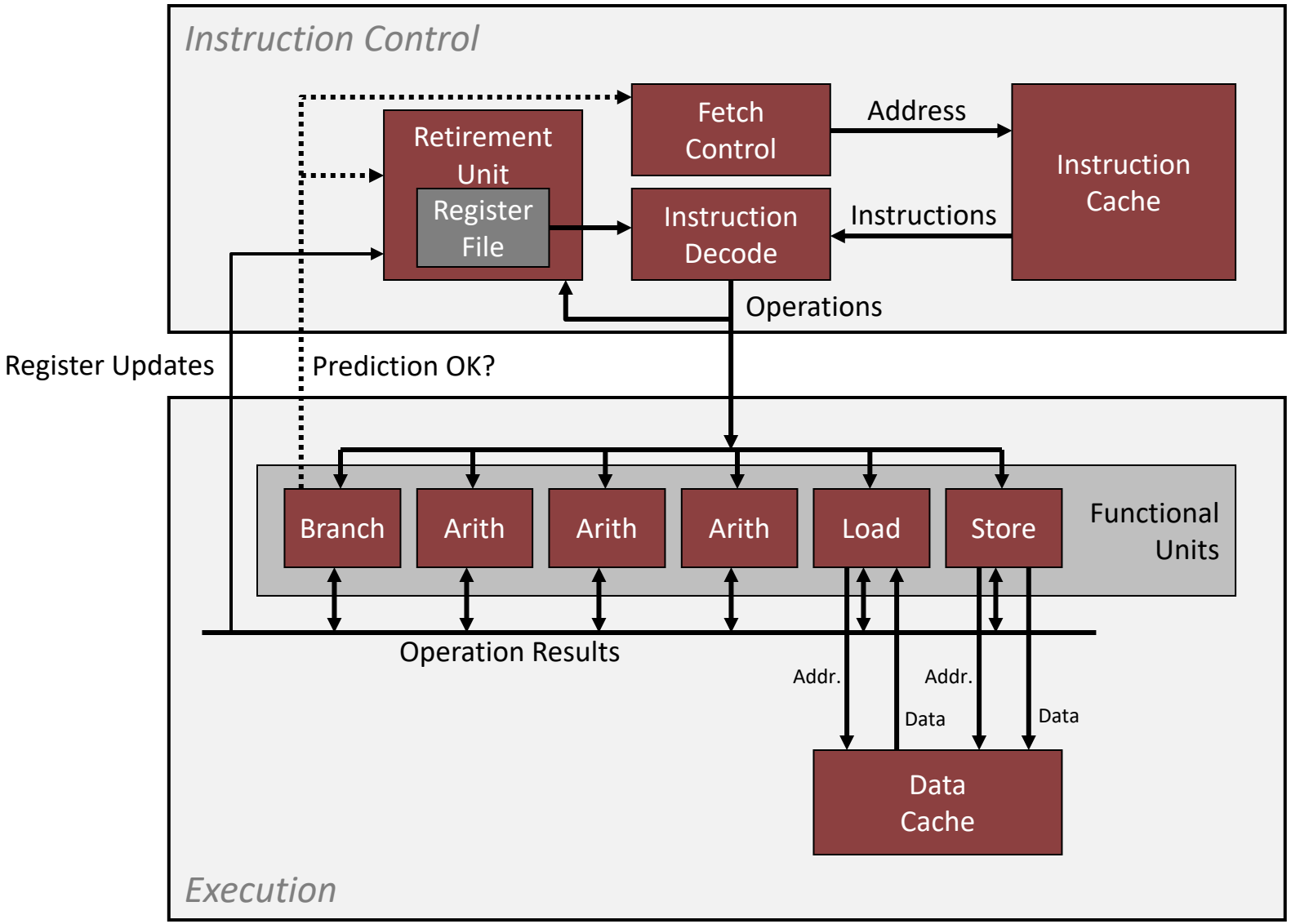
Superscalar, Out-of-order

- 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.

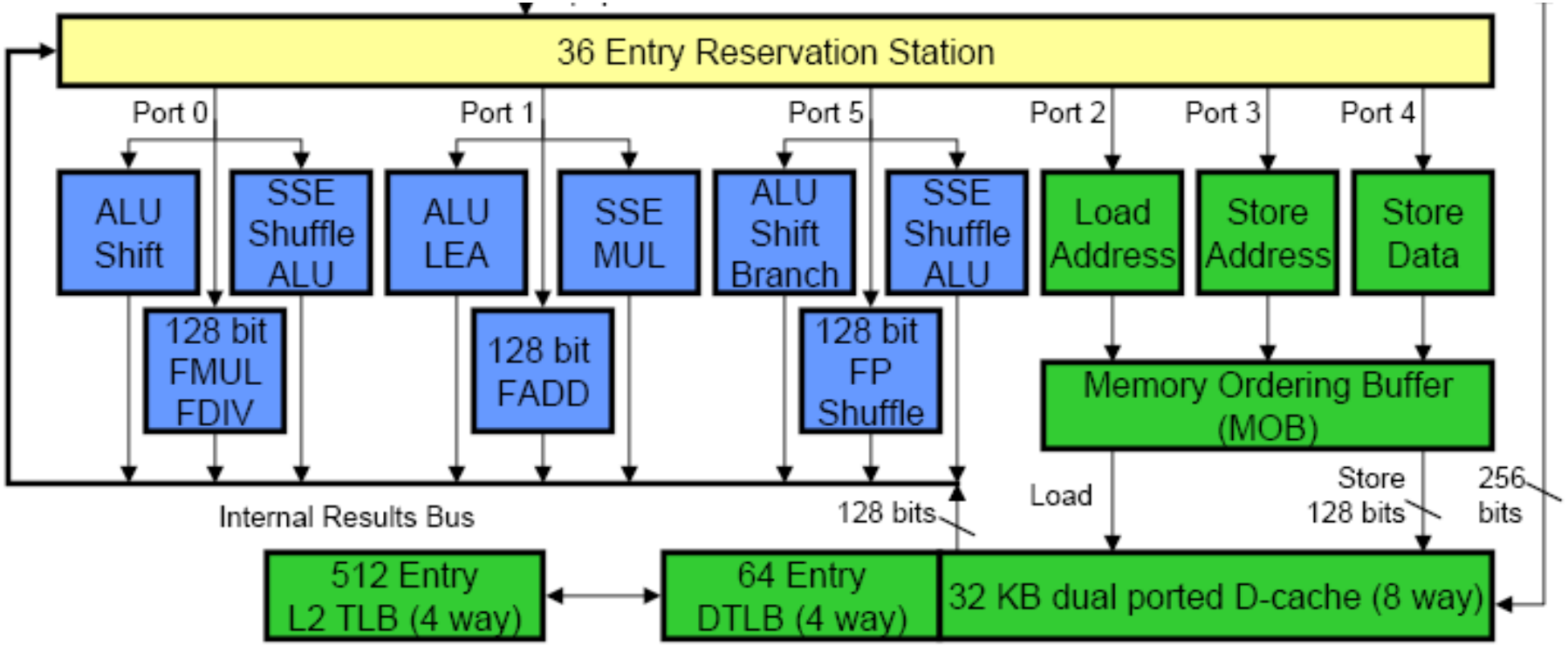
- Superscalar, out-of-order processors can take advantage of the *instruction level parallelism* that most programs have.

- 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

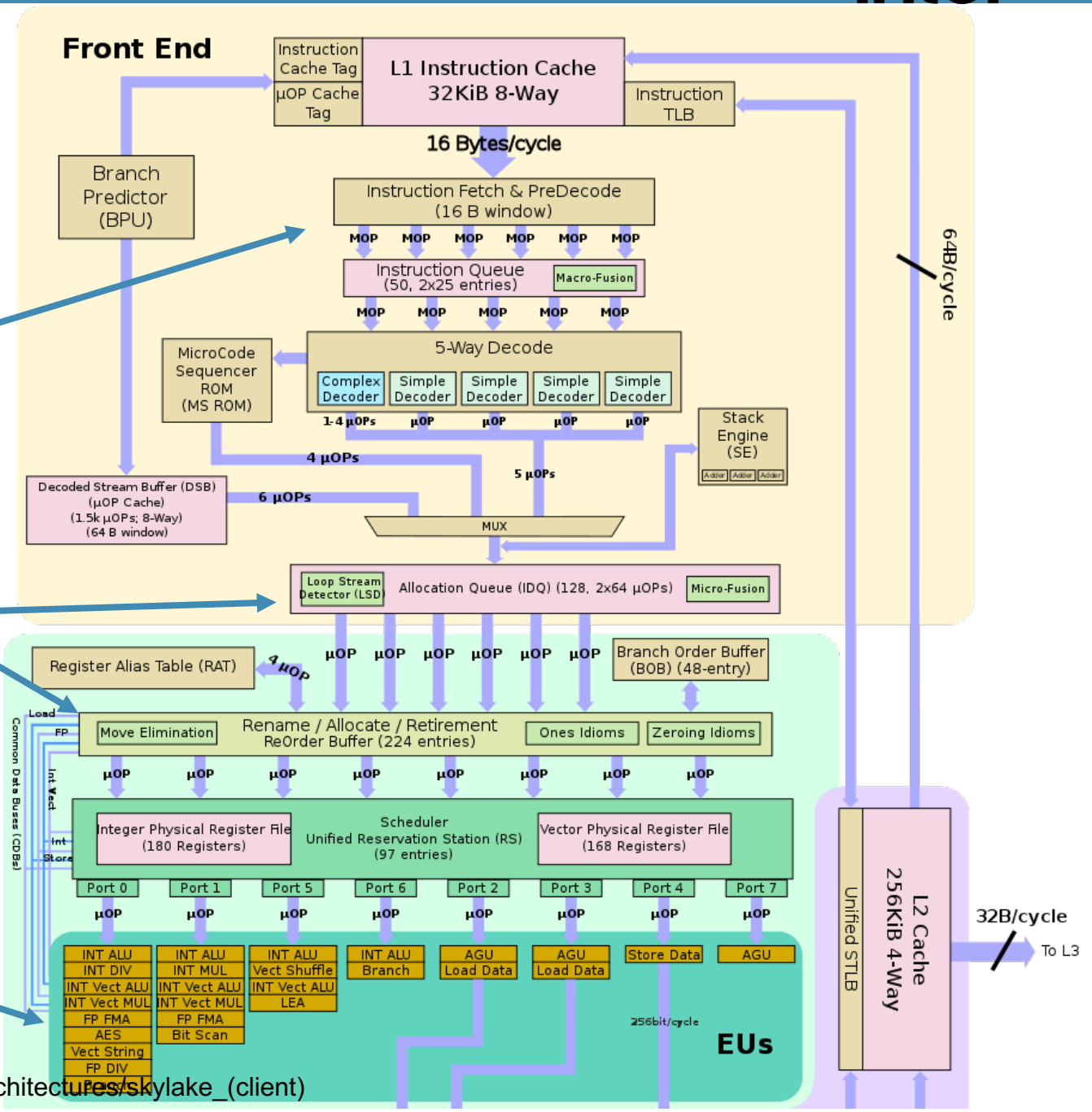


Image: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

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

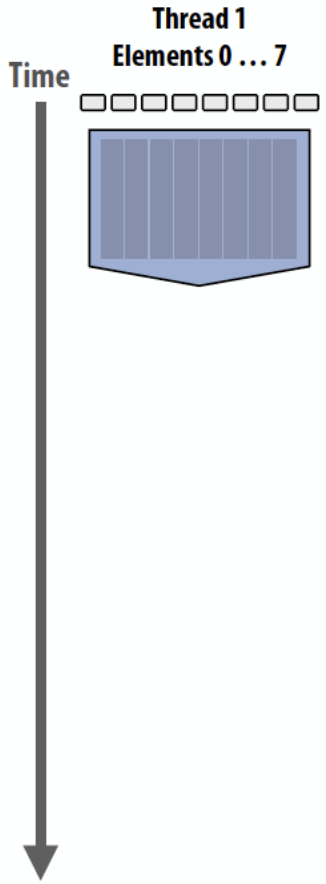
Why bother?

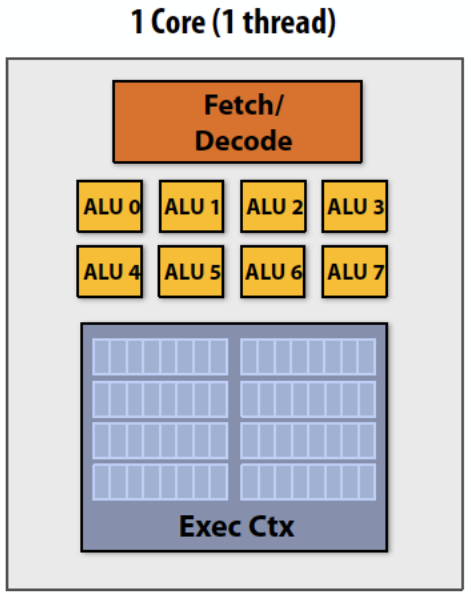
- Interleave the processing of multiple instruction streams on the same core to hide the latency of stalls

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

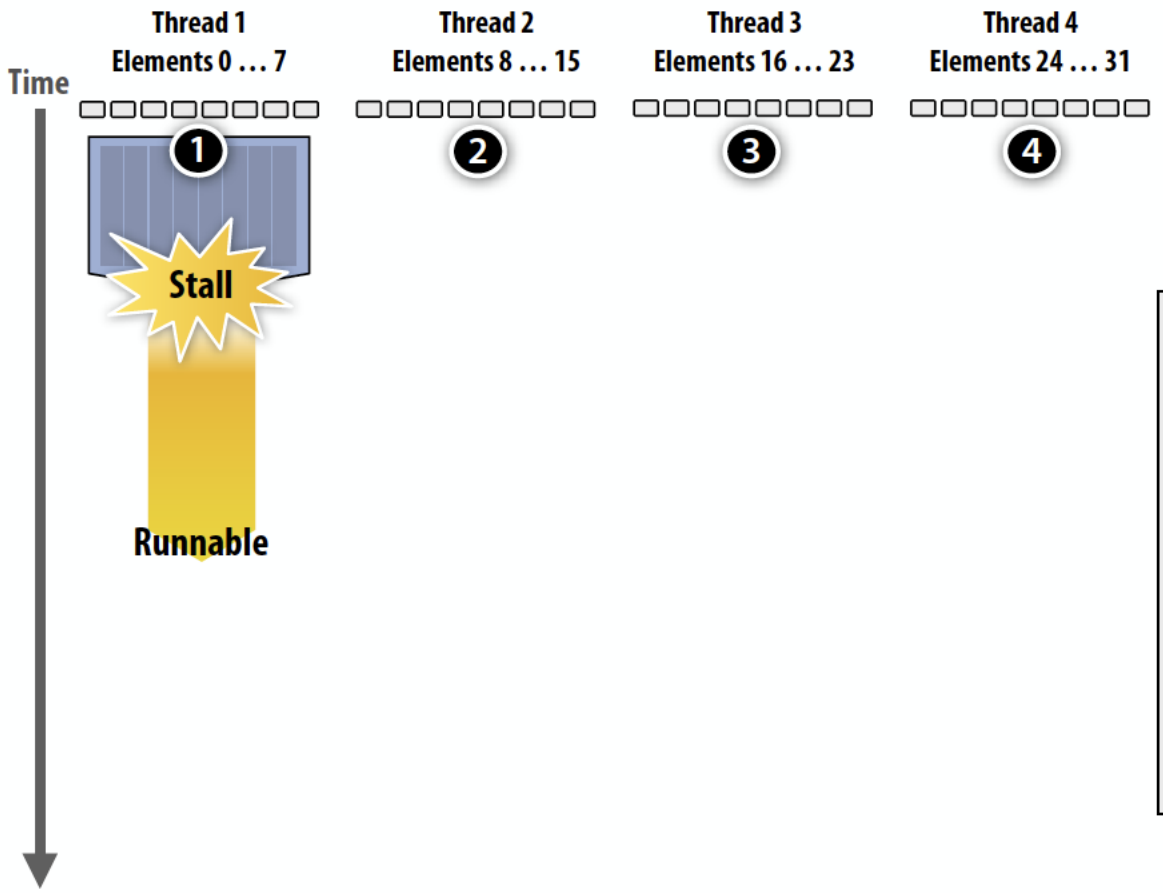
- *Performance improvement: higher throughput*

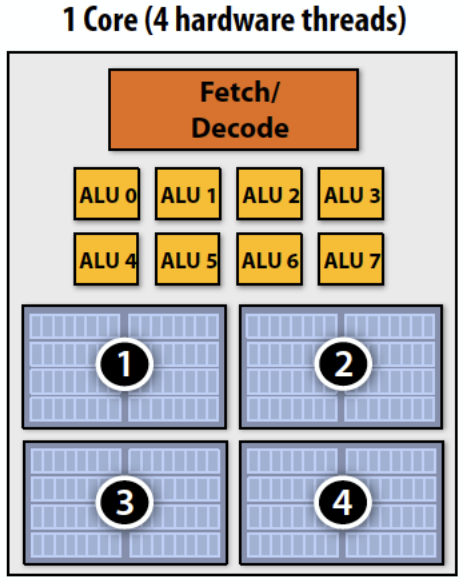
Advantage: increased throughput



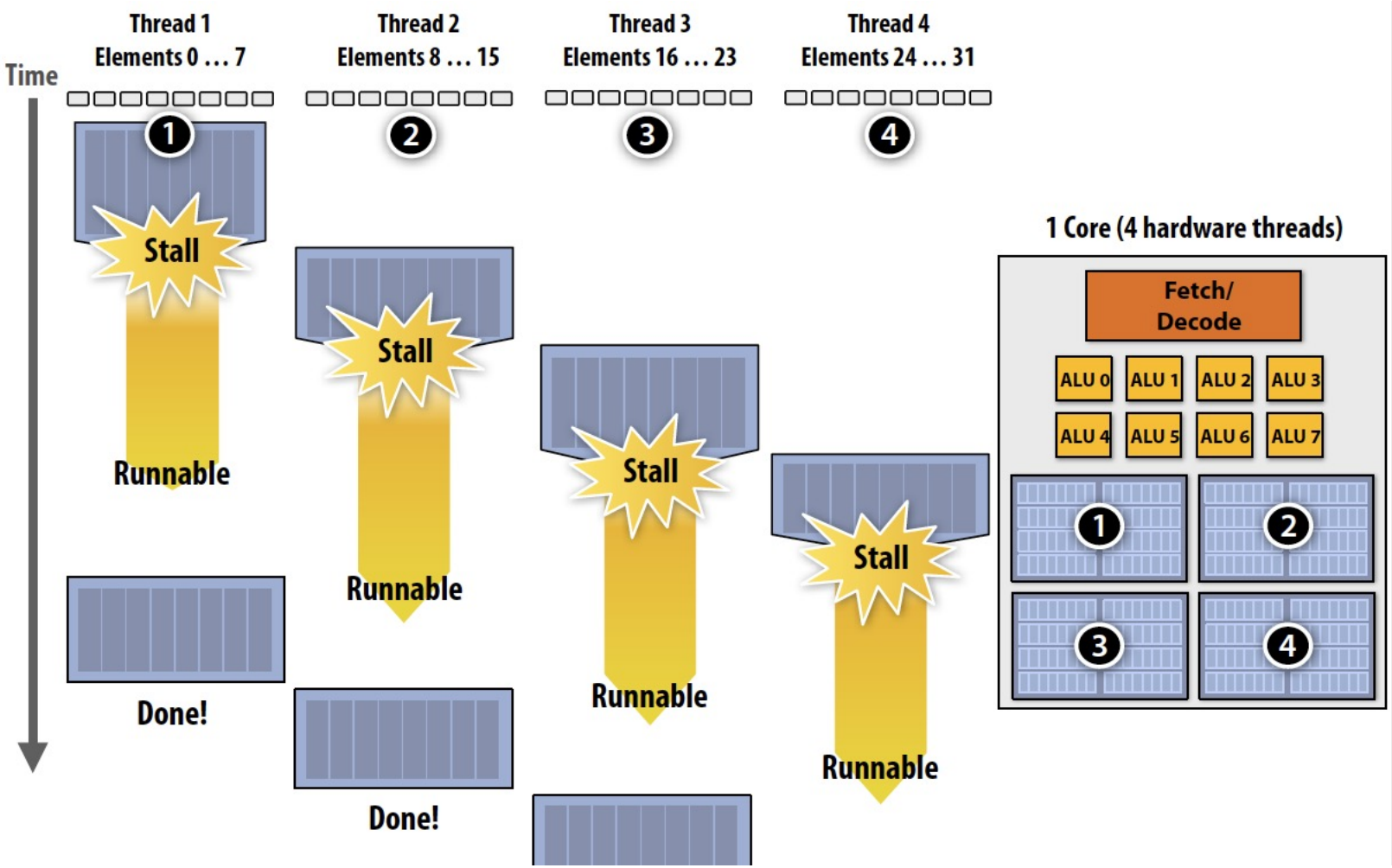


Advantage: increased throughput



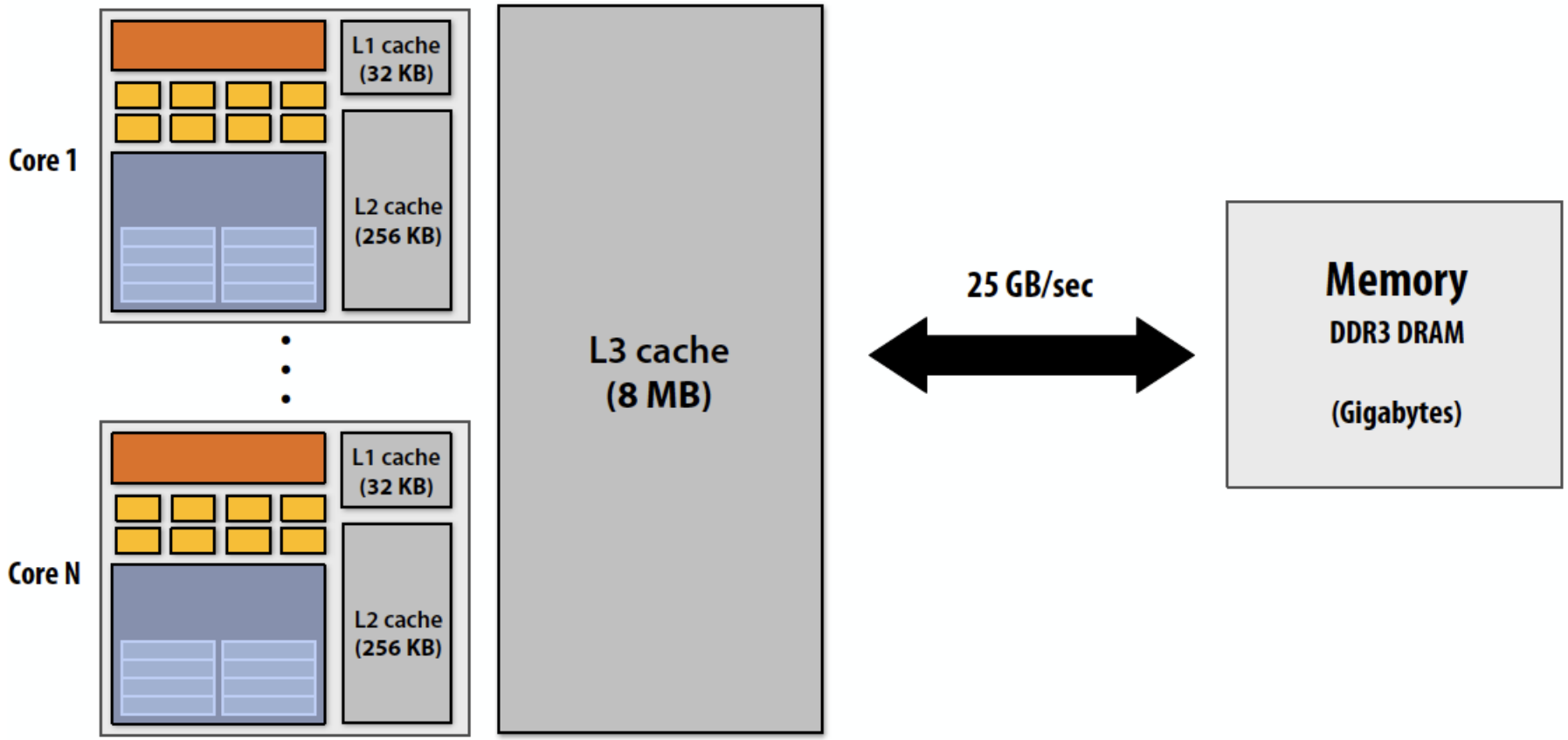


Advantage: increased throughput



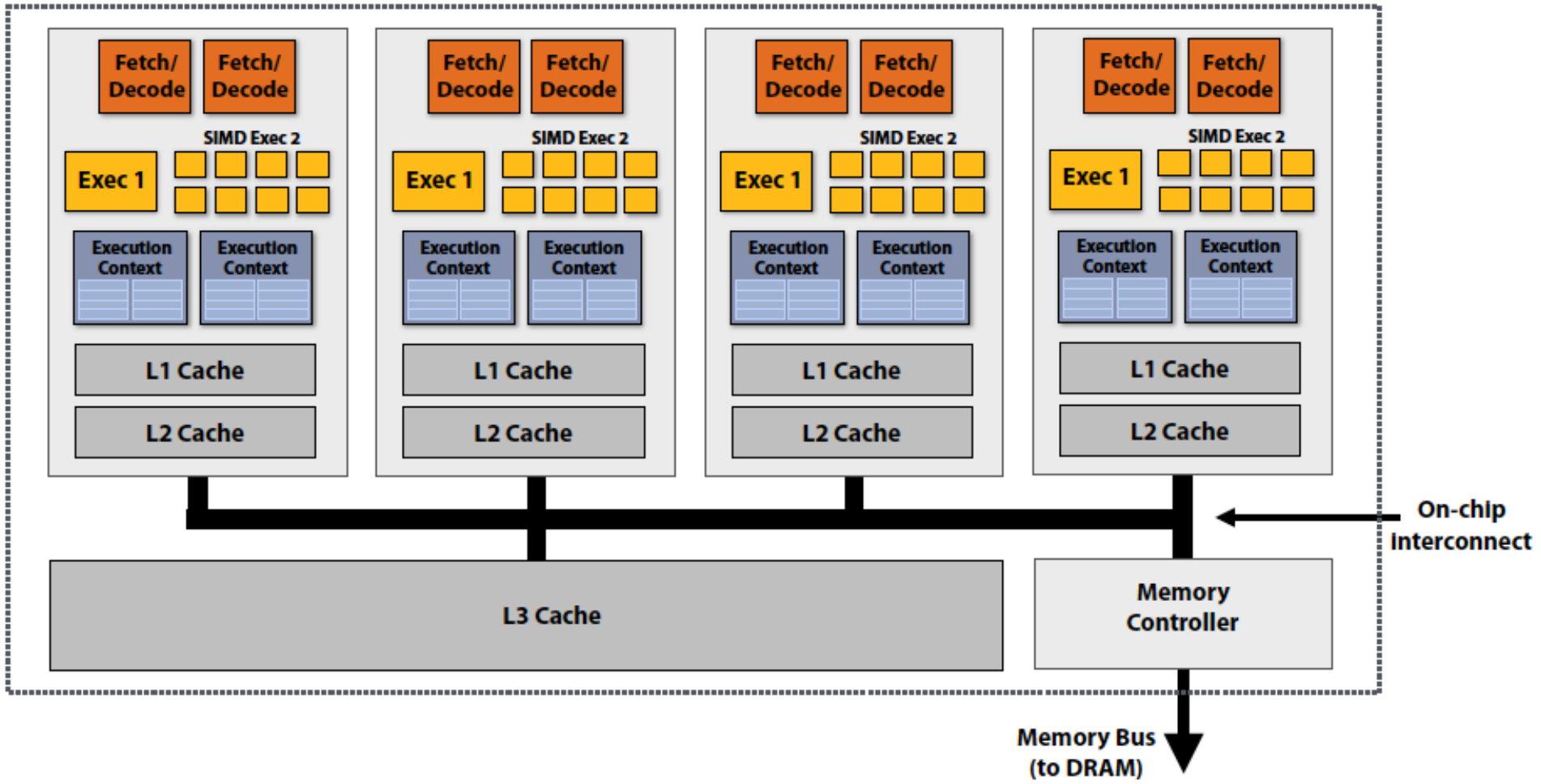
What about the memory?

- 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

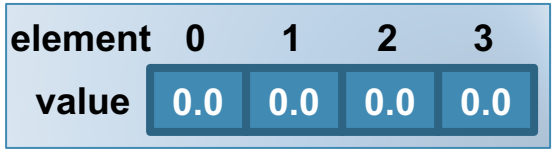
- Auto-vectorization
  - Both gcc and icc have support for it
  - Successful for simple loops and data structures
- Compiler directives
  - Both gcc and icc allow for specific pragma's to enable vectorization
  - Pragma's are used to "force" the compiler to vectorize
- C or C++: **intrinsics**
  - Declare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

Using intrinsics

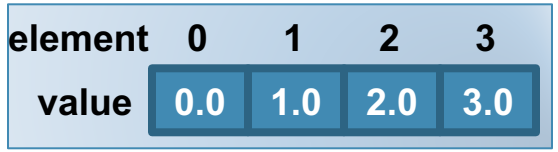
- https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions
- https://software.intel.com/sites/landingpage/IntrinsicsGuide/
- Requirements:
  - Using aligned data structures (aligned to the size of the vector)

Examples of intrinsics

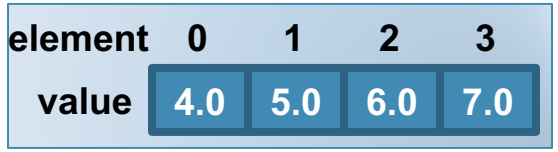
```
float data[1024];
// init: data[0] = 0.0, data[1] = 1.0, data[2] = 2.0, etc.
init(data);

// 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

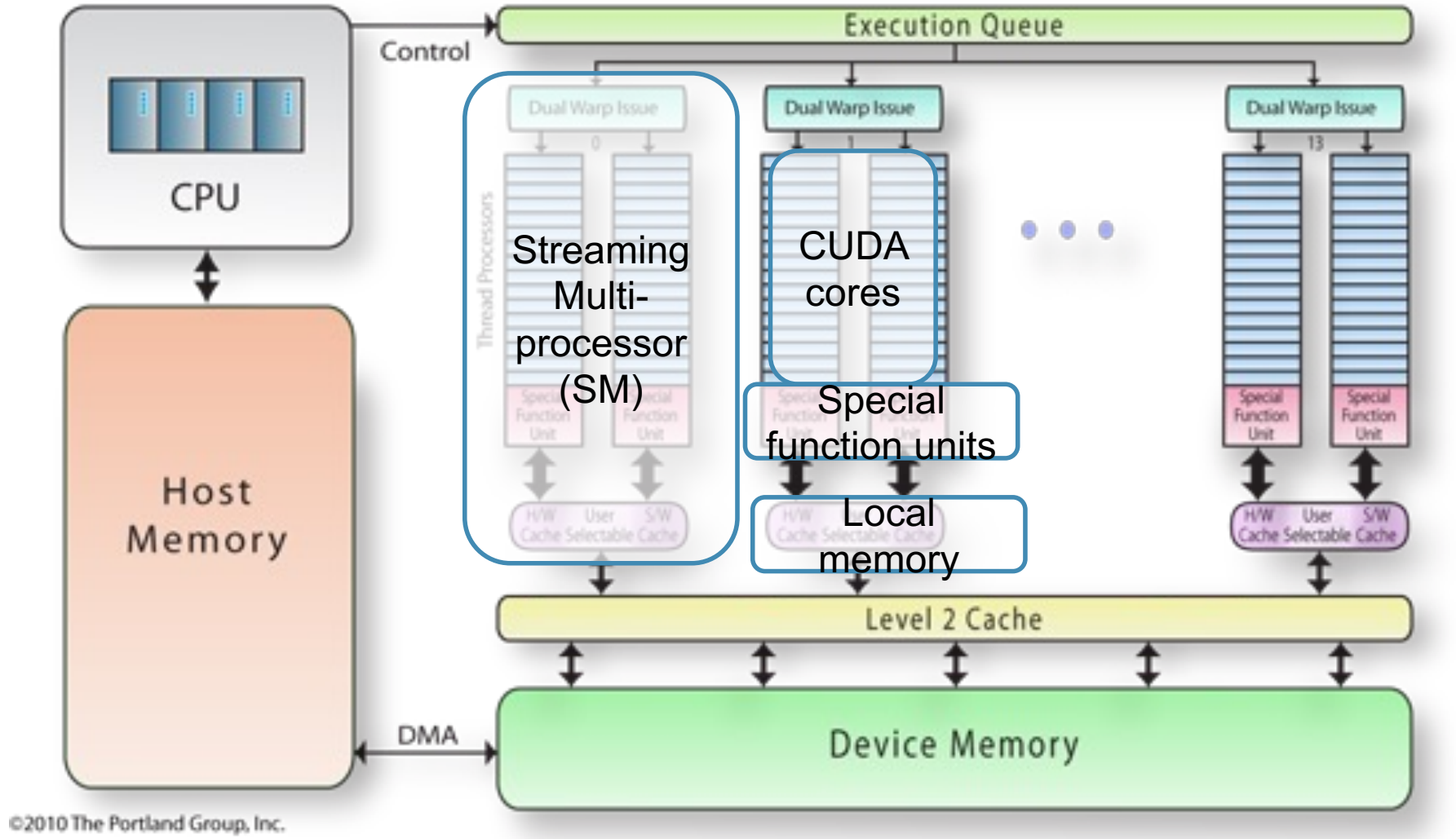
```
void vectorAdd(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i += 4) {
        c[i+0] = a[i+0] + b[i+0];
        c[i+1] = a[i+1] + b[i+1];
        c[i+2] = a[i+2] + b[i+2];
        c[i+3] = a[i+3] + b[i+3];
    }
}
```

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

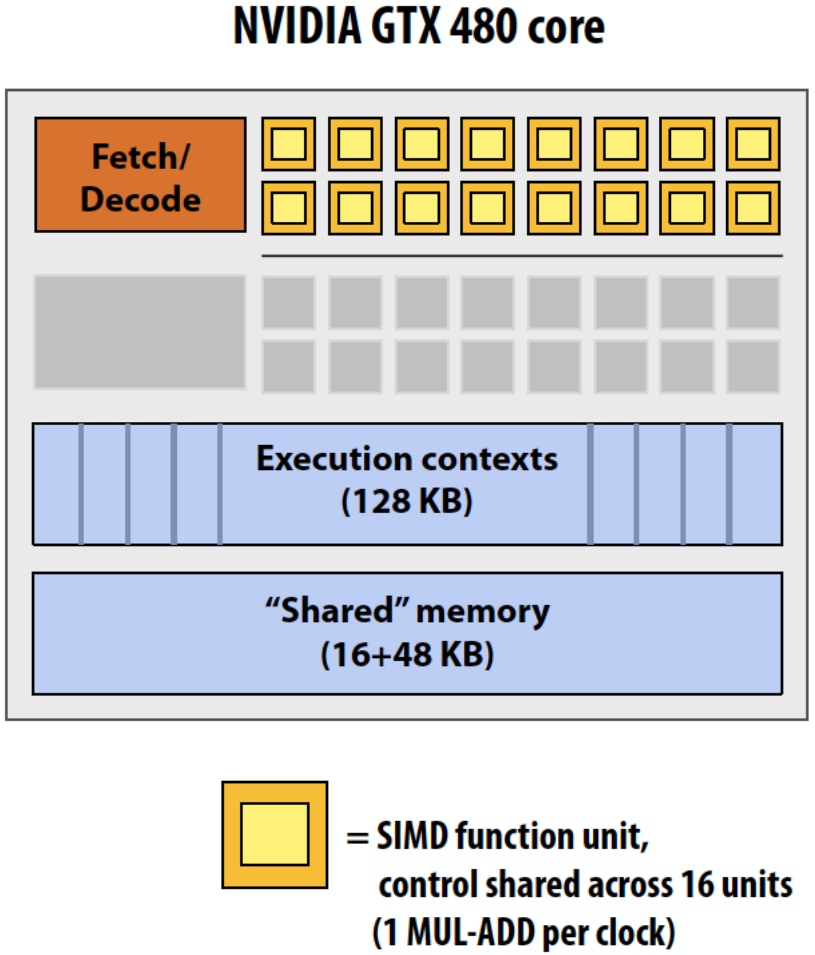
Generic GPU


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

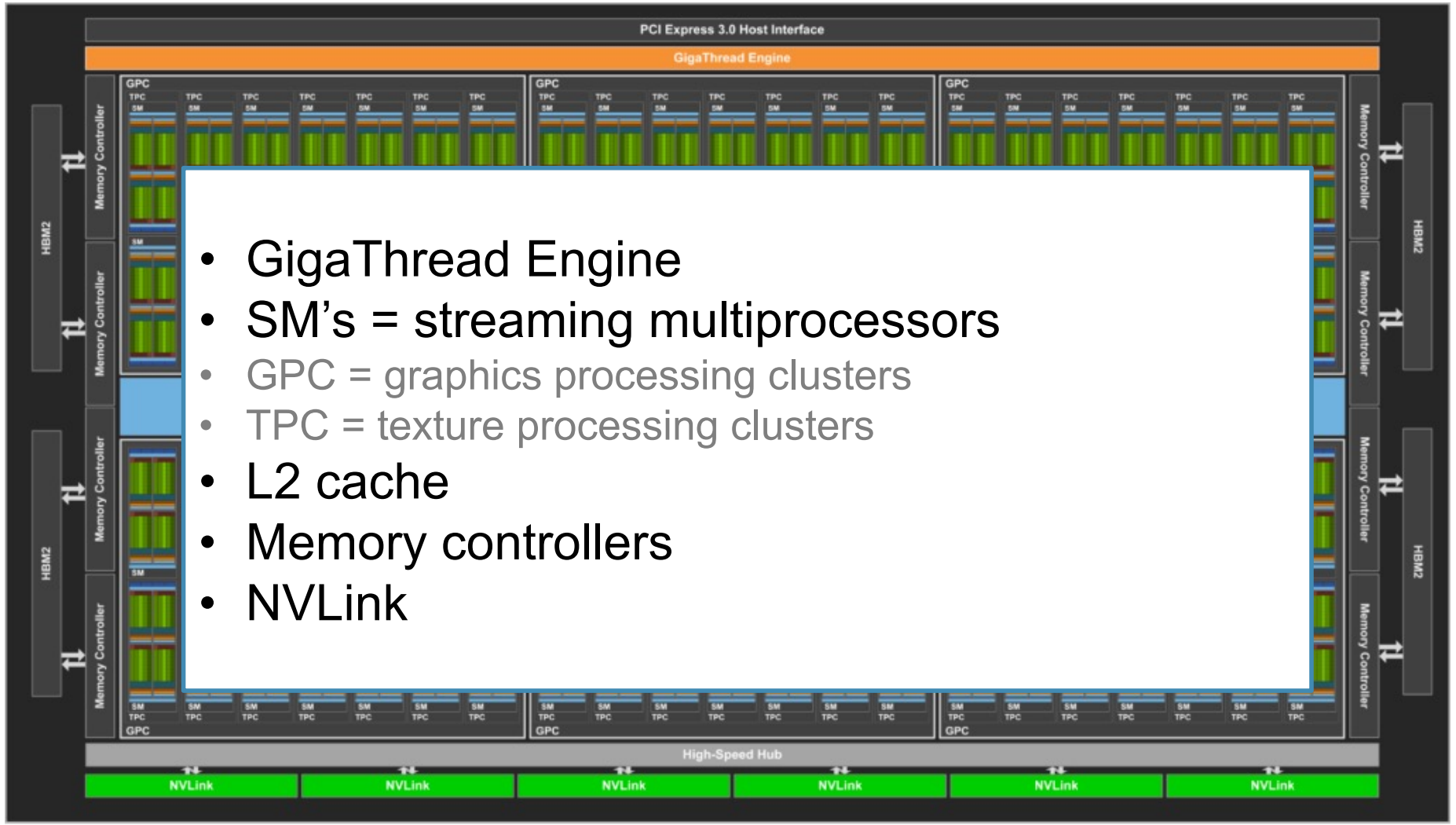
- Full board: 15 cores (SMs)!



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

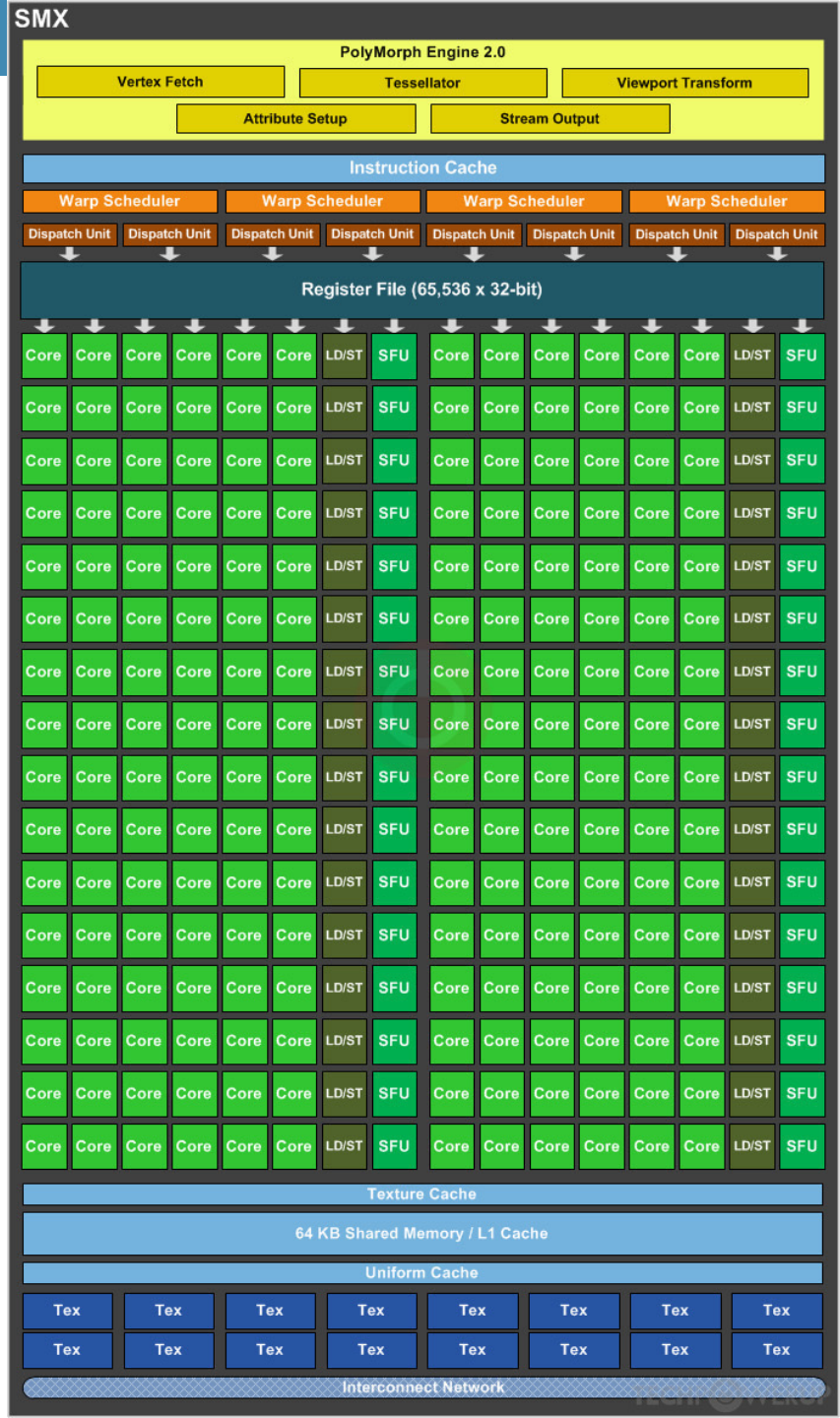
Inside an NVIDIA GPU architecture

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)

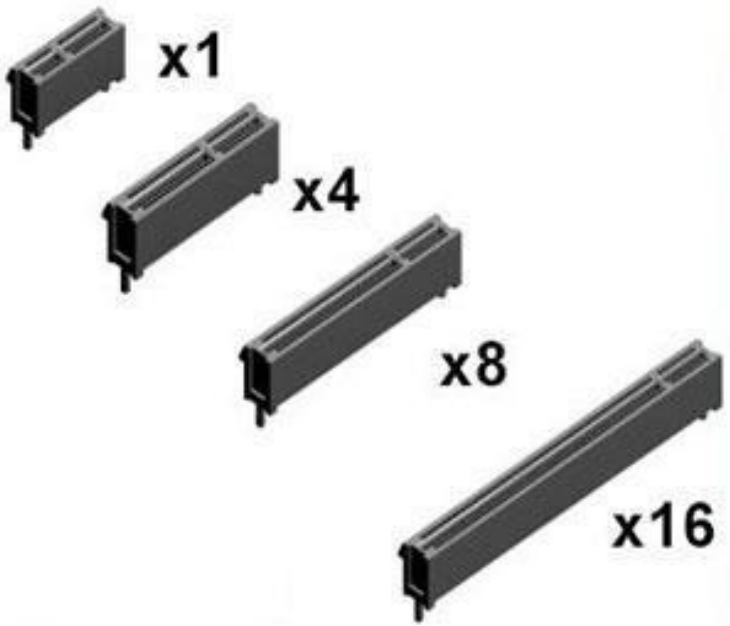
<Pascal

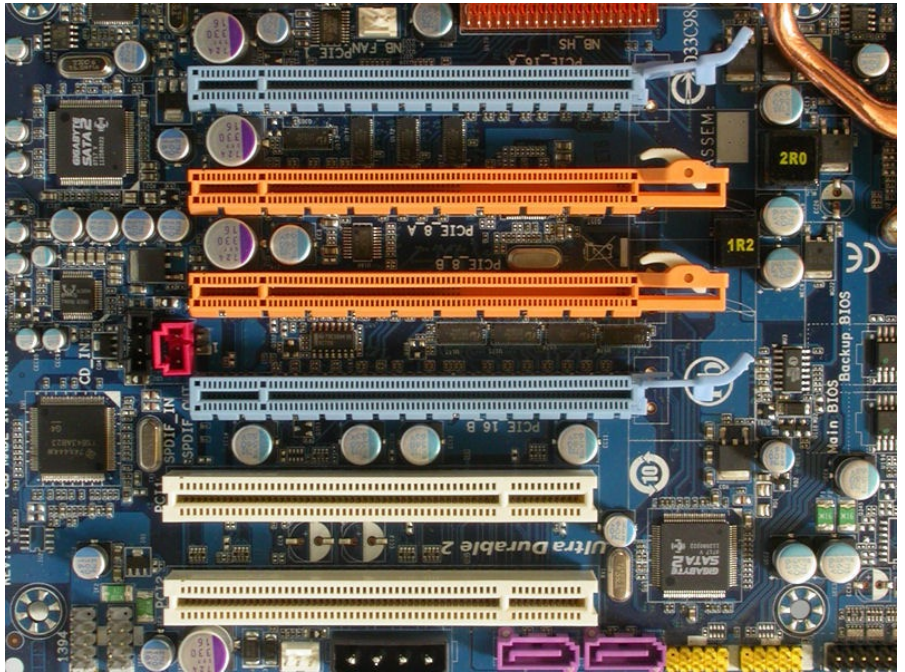
Volta>

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



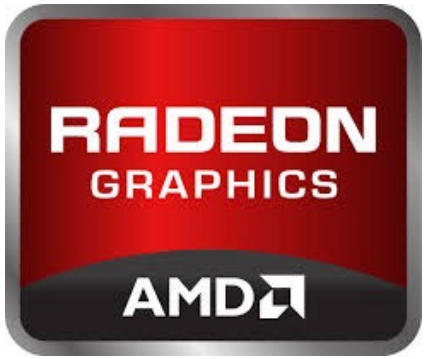


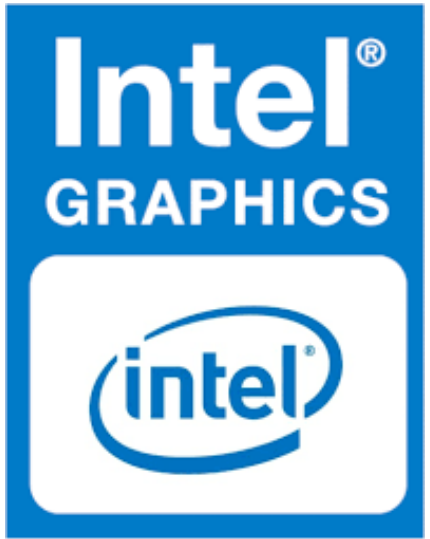
NVIDIA GPUs (8+ years)

	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 …

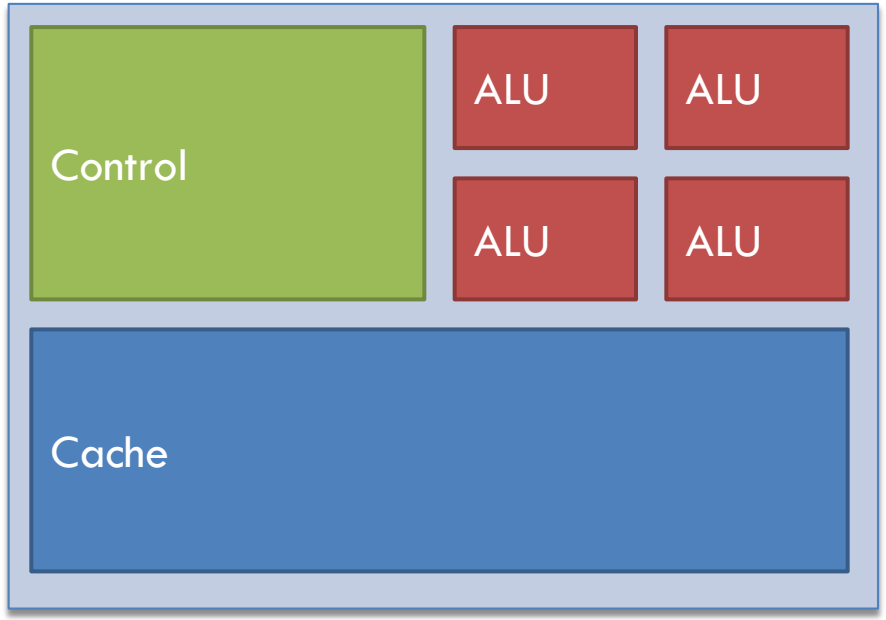
*http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

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

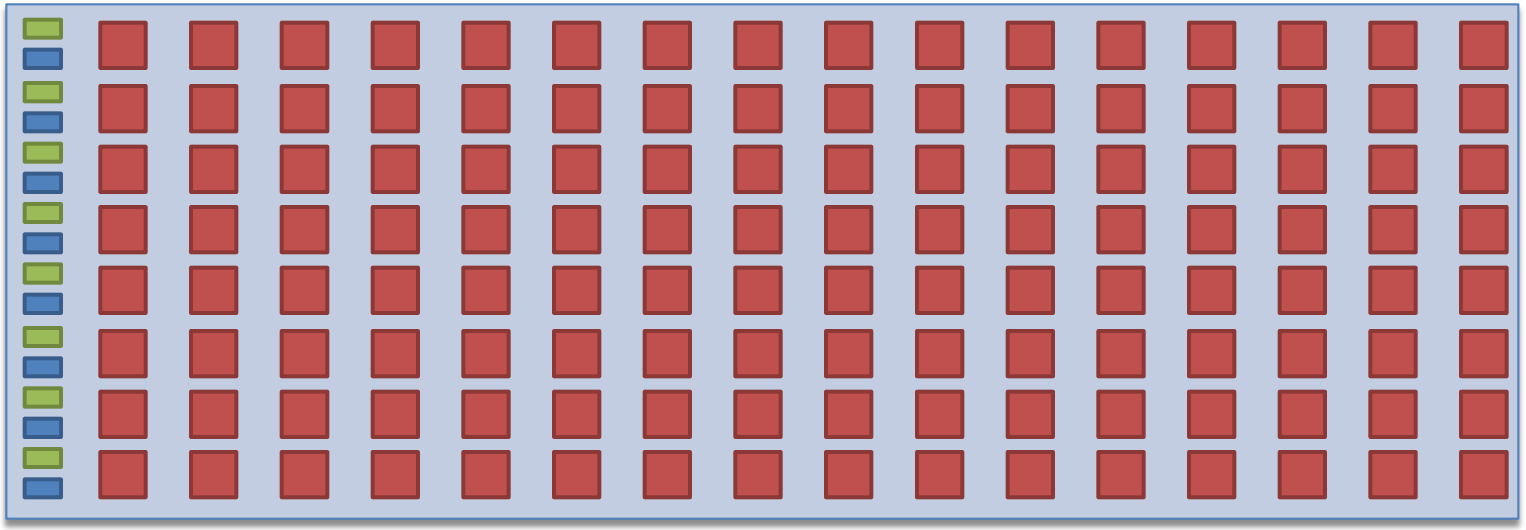


CPU

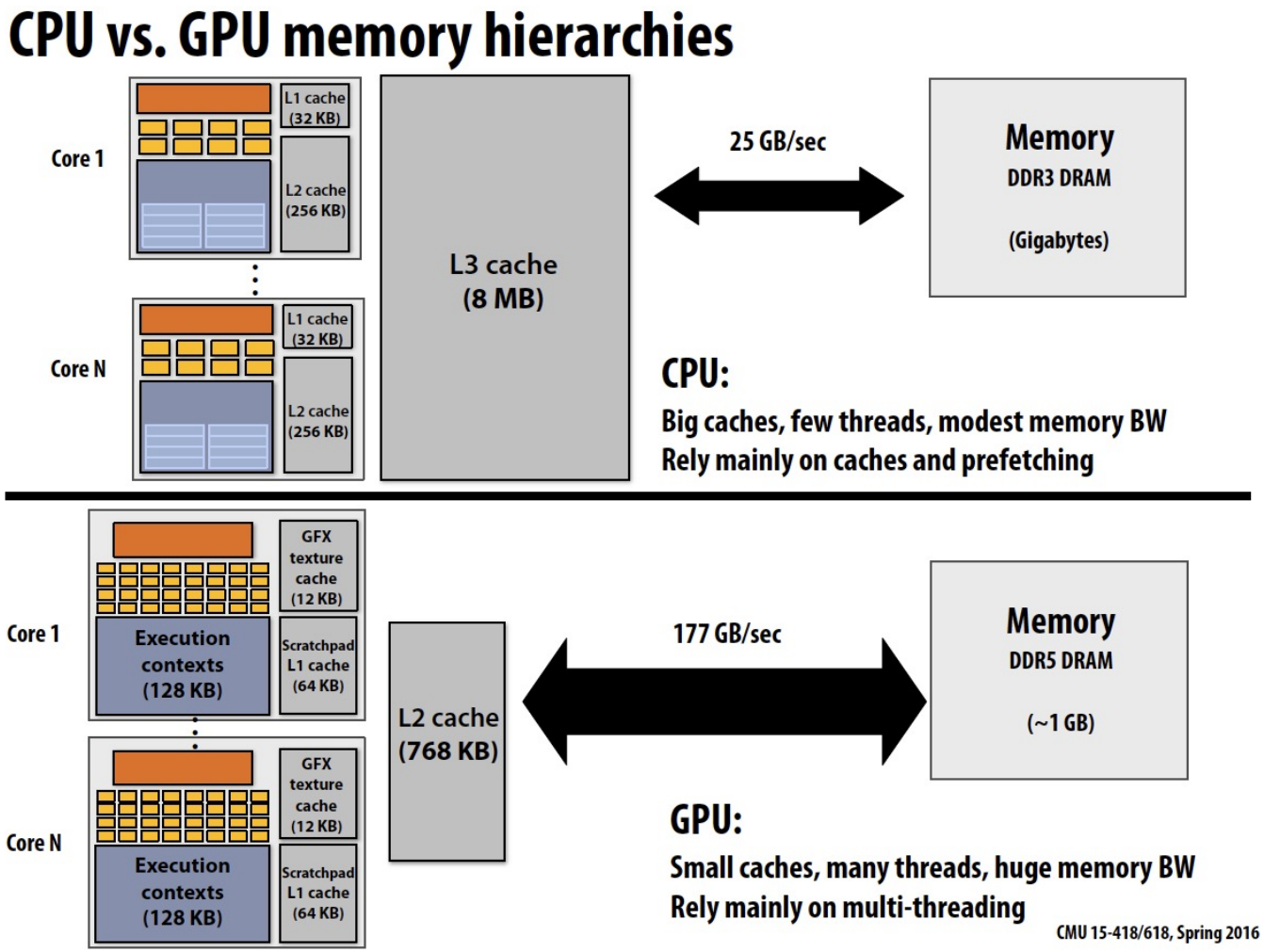
Low latency, high flexibility.
Excellent for irregular codes with limited parallelism.

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

PART 3: PERFORMANCE

Performance “metrics”

- Clock frequency [GHz] = absolute hardware speed
  - Memories, CPUs, interconnects
- **Operational speed [GFLOPs]**
  - Operations per second, **single/double/…** precision
- **Memory bandwidth [GB/s]**
  - Memory operations per second
  - Differs a lot between different memories on chip
- Derived metrics
  - FLOP/Byte, FLOP/Watt

Name	FLOPS
yottaFLOPS	10^{24}
zettaFLOPS	10^{21}
exaFLOPS	10^{18}
petaFLOPS	10^{15}
teraFLOPS	10^{12}
gigaFLOPS	10^{9}
megaFLOPS	10^{6}
kiloFLOPS	10^{3}

Theoretical peak performance

Throughput[GFLOP/s] = chips * cores * vectorWidth * FLOPs/cycle * clockFrequency

Bandwidth[GB/s] = memory bus frequency * bits per cycle * bus width

	Cores	Threads/ALUs	Throughput	Bandwidth
Intel Core i7	4	16	**85**	**25.6**
AMD Barcelona	4	8	**37**	**21.4**
AMD Istanbul	6	6	**62.4**	**25.6**
NVIDIA GTX 580	16	512	**1581**	**192**
NVIDIA GTX 680	8	1536	**3090**	**192**
AMD HD 6970	384	1536	**2703**	**176**
AMD HD 7970	32	2048	**3789**	**264**
Intel Xeon Phi 7120	61	240	**2417**	**352**

Why should we care?

- Peak performance indicates an absolute bound of the performance that can be achieved on a given machine
  - It is *application independent*

- Such performance is rarely* achievable in practice for real applications.
  - Applications rarely utilize all the machine features.

- The balance of an application must *consistently* match the balance of the machine to get anywhere near the peak…
- ... or else… different bottlenecks!

*Empirical studies show this reads as “almost never” .

https://gitlab.com/astron-misc/benchmark-intrinsics/-/tree/master

Hardware performance

Performance “metrics”

- Clock frequency [GHz] = absolute hardware speed
  - Memories, CPUs, interconnects
- **Operational speed [GFLOPs]**
  - Operations per second
  - **single** AND **double** precision
- **Memory bandwidth [GB/s]**
  - Memory operations per second
    - Can differ for read and write operations !
  - Differs a lot between different memories on chip
- Power [Watt]
  - The rate of consumption of energy
- Derived metrics
  - FLOP/Byte, FLOP/Watt

Name	FLOPS
yottaFLOPS	10^{24}
zettaFLOPS	10^{21}
exaFLOPS	10^{18}
petaFLOPS	10^{15}
teraFLOPS	10^{12}
gigaFLOPS	10^{9}
megaFLOPS	10^{6}
kiloFLOPS	10^{3}

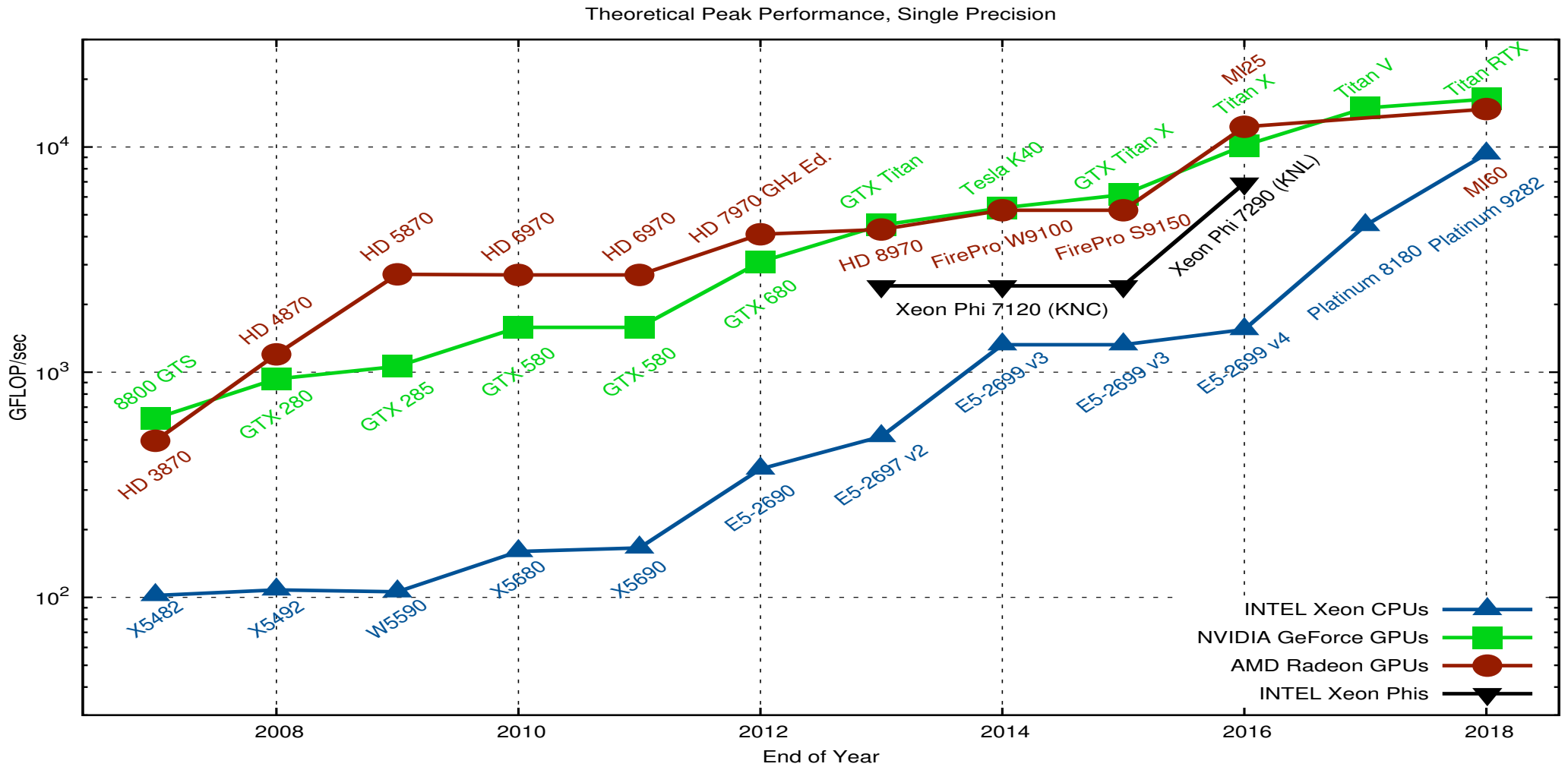
Theoretical peak performance

Throughput[GFLOP/s] = chips * cores * vectorWidth * FLOPs/cycle * clockFrequency

Bandwidth[GB/s] = memory bus frequency * bits per cycle * bus width

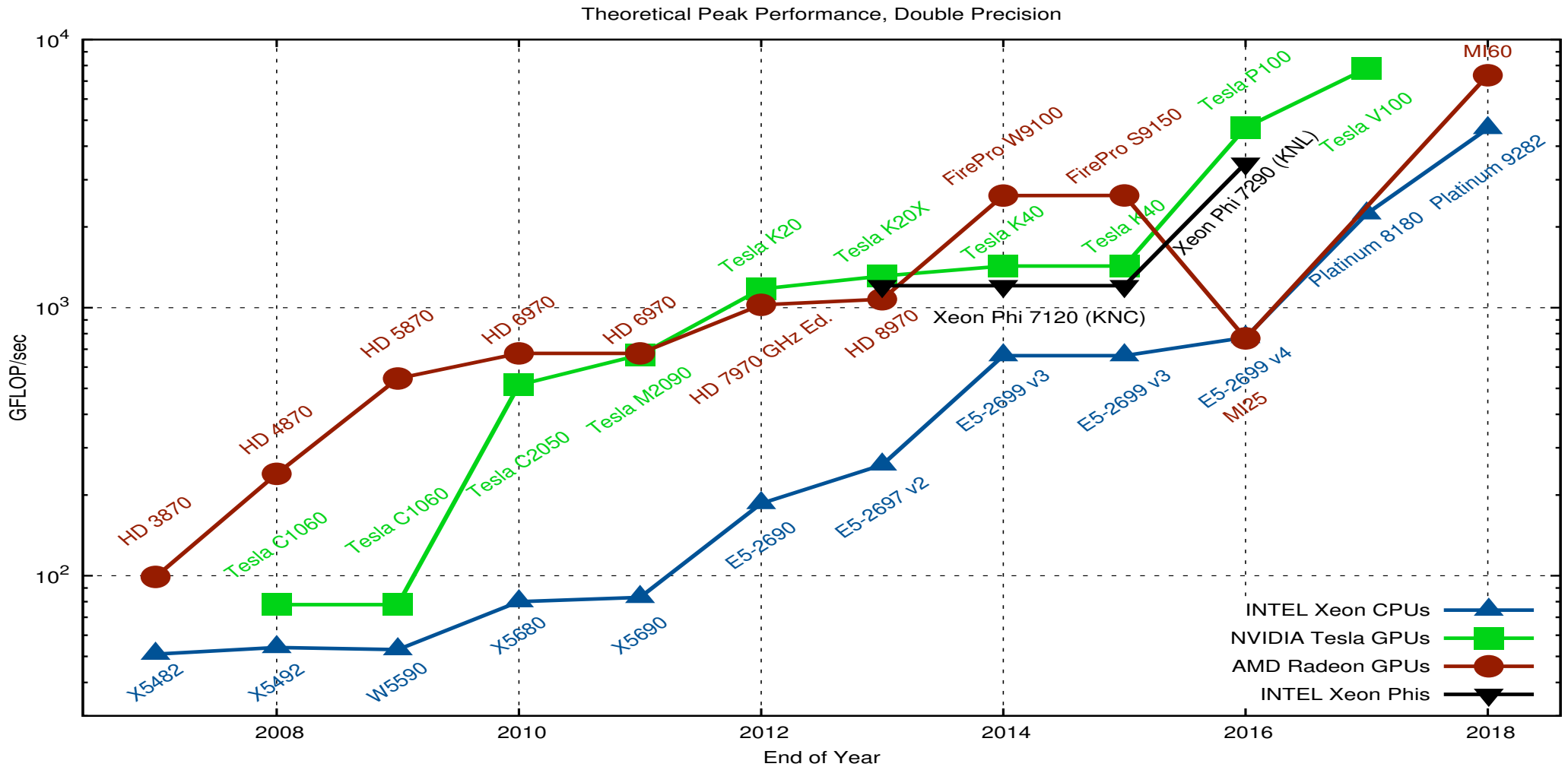
	Cores	Threads/ALUs	Throughput	Bandwidth
Intel Core i7	4	16	**85**	**25.6**
AMD Barcelona	4	8	**37**	**21.4**
AMD Istanbul	6	6	**62.4**	**25.6**
NVIDIA GTX 580	16	512	**1581**	**192**
NVIDIA GTX 680	8	1536	**3090**	**192**
AMD HD 6970	384	1536	**2703**	**176**
AMD HD 7970	32	2048	**3789**	**264**
Intel Xeon Phi 7120	61	240	**2417**	**352**

multi vs *many* cores (SP-FLOPs)



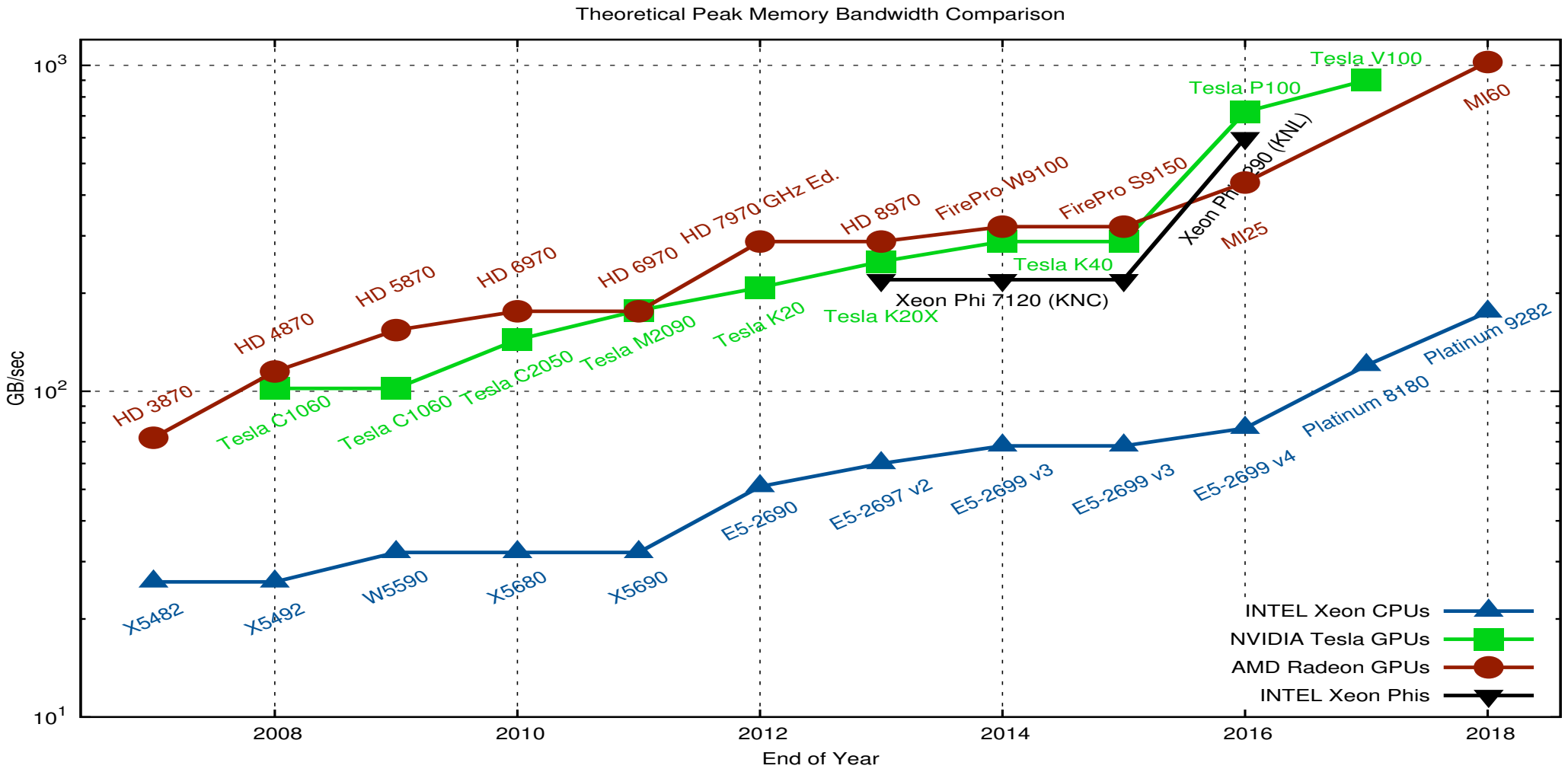
https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

multi vs *many* cores (DP-FLOPs)



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

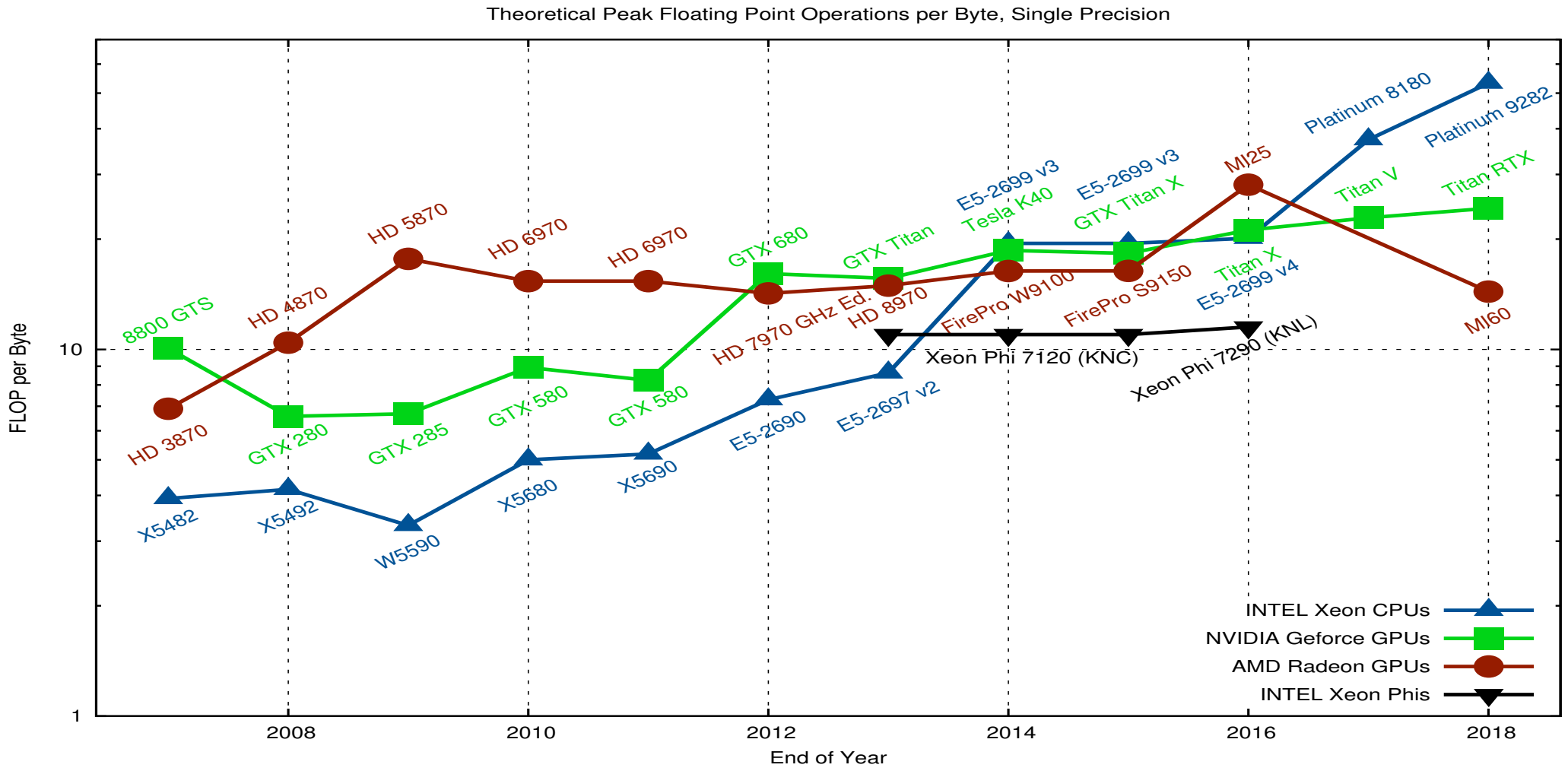
multi vs *many* cores (GB/s)



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Balance ?

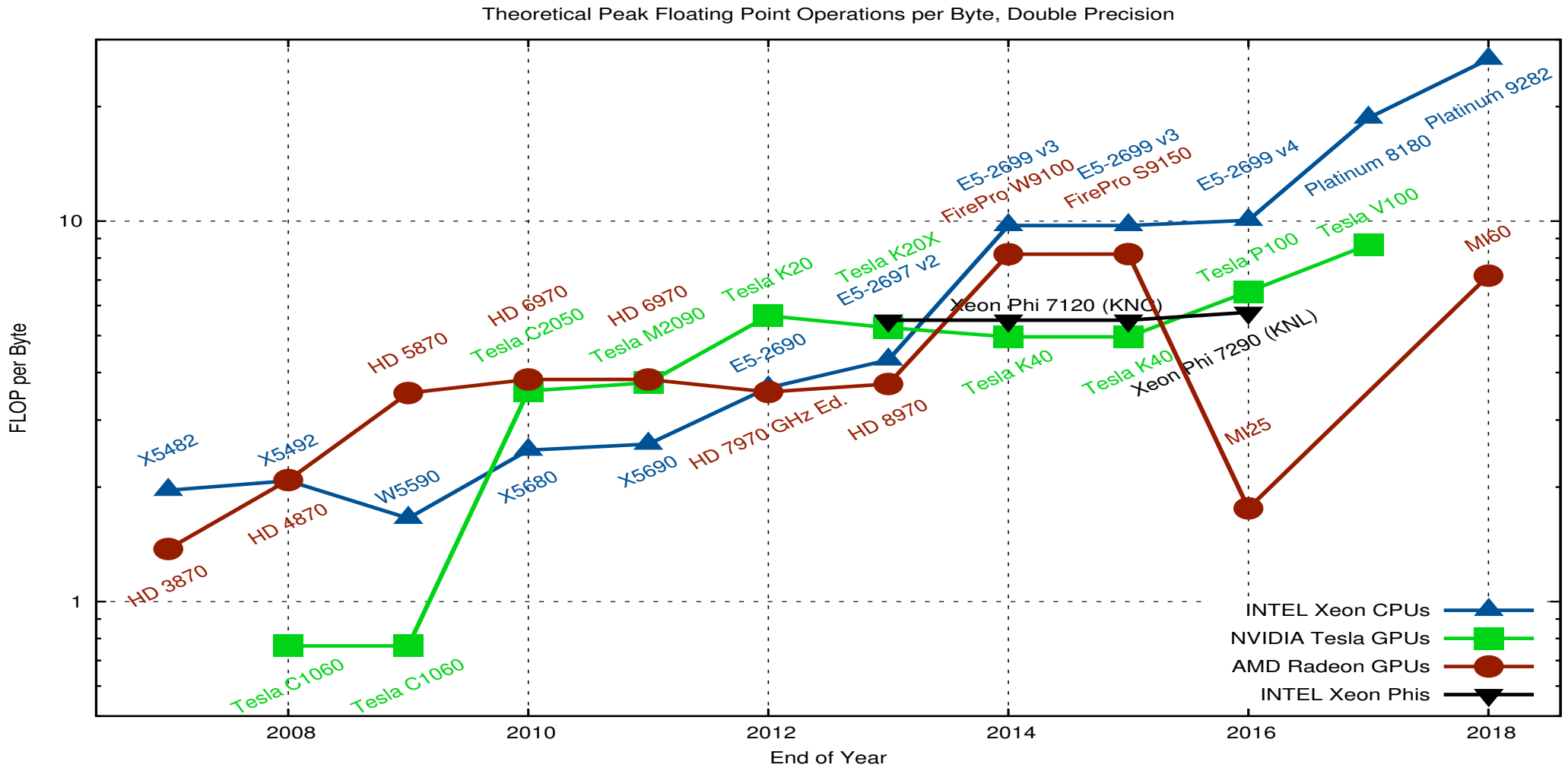
FLOPs/Byte (SP) !



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Balance ?

FLOPs/Byte (DP) !



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Why should we care?

- Peak performance indicates an absolute bound of the performance that can be achieved on a given machine
  - It is *application independent*
- Such performance is rarely* achievable in practice for real applications.
  - Applications rarely utilize all the machine features.
- The balance of an application must *consistently* match the balance of the machine to get anywhere near the peak…
- ... or else… different bottlenecks!

*Empirical studies show this reads as “almost never” .

https://gitlab.com/astron-misc/benchmark-intrinsics/-/tree/master

Measuring hardware performance

- Microbenchmarking*
  - Evaluates hardware features in isolation
  - Goal: find out the true limits of the hardware components
  - Platform-specific results
  - **Compared** with the theoretical peak, **per platform**.

- Benchmarking
  - Evaluates the FULL platform
  - Application-specific performance
    - Top500 – computation capability
    - Graph500 – graph processing capability
    - Green500 – energy consumption
  - **Compares platforms**

* Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, Andreas Moshovos. ”Demystifying GPU

Microbenchmarks

- Isolate specific features of the processing units and define specific stress tests for them.

- Compute operations
  - CPU: nanoBench, likwid-bench, …
  - GPU: MIPP, various papers, …

- Memory operations
  - ”memory mountain”
    - see Computer Systems: A Programmer’s Perspective
  - CPU: nanoBench, lmbench3, …
  - GPU: various papers

- Different compute and memory mixes
  - STREAM / BabelStream / …

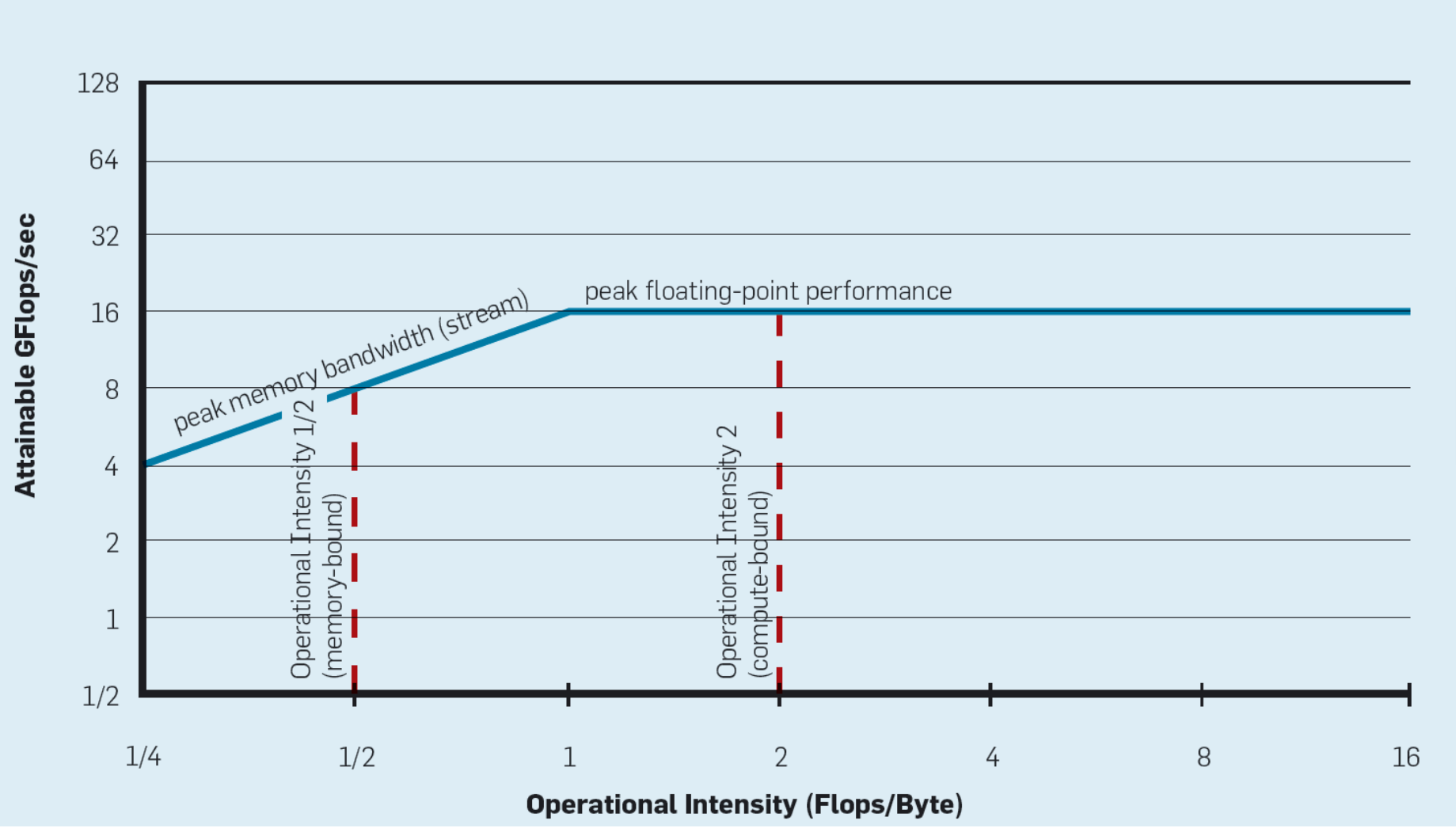
Benchmarking suites

- Collections of “representative” applications
- Allow testing processors in real-life conditions and compare them

- Application-specific benchmarking suites
  - Top500
  - Graph500
  - Green500

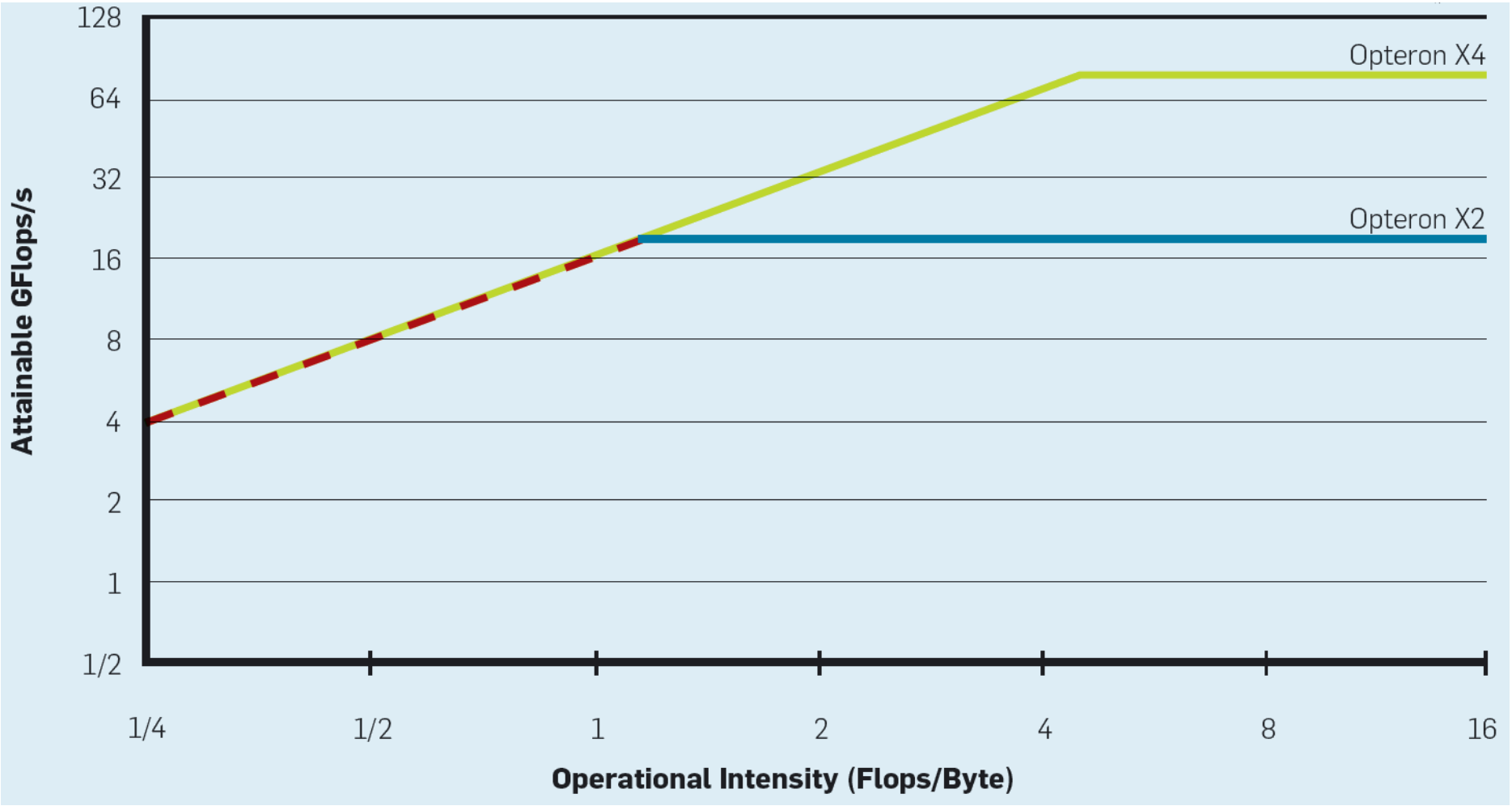
- Scientific benchmarking suites:
  - SPEC benchmarks
  - NAS parallel benchmarks
  - SPLASH-2
  - PARSEC
  - …

The Roofline model



AMD Opteron X2 (two cores): 17.6 gflops, 15 GB/s, ops/byte = 1.17

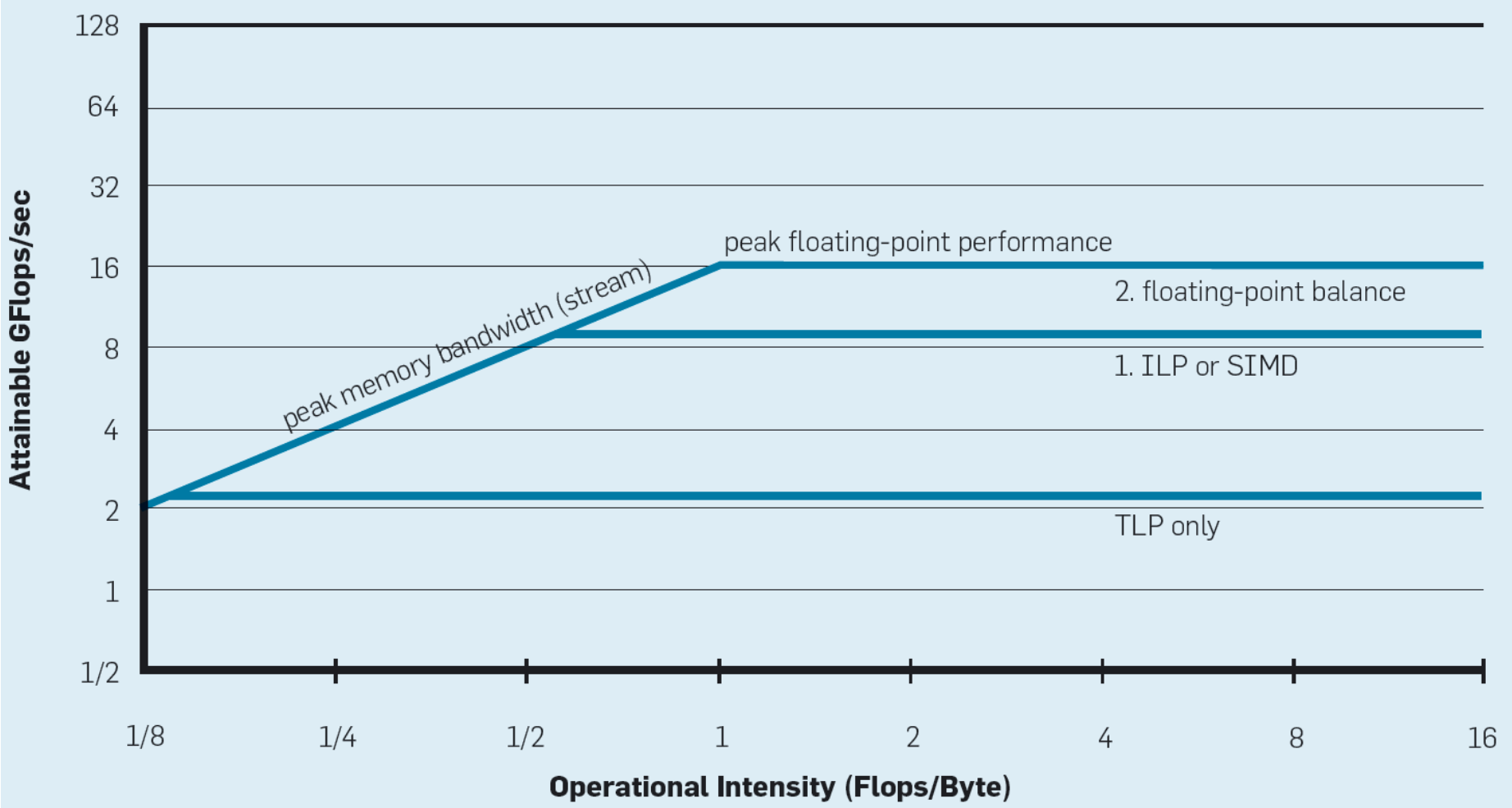
Roofline: comparing architectures



AMD Opteron X2: 17.6 gflops, 15 GB/s, ops/byte = 1.17

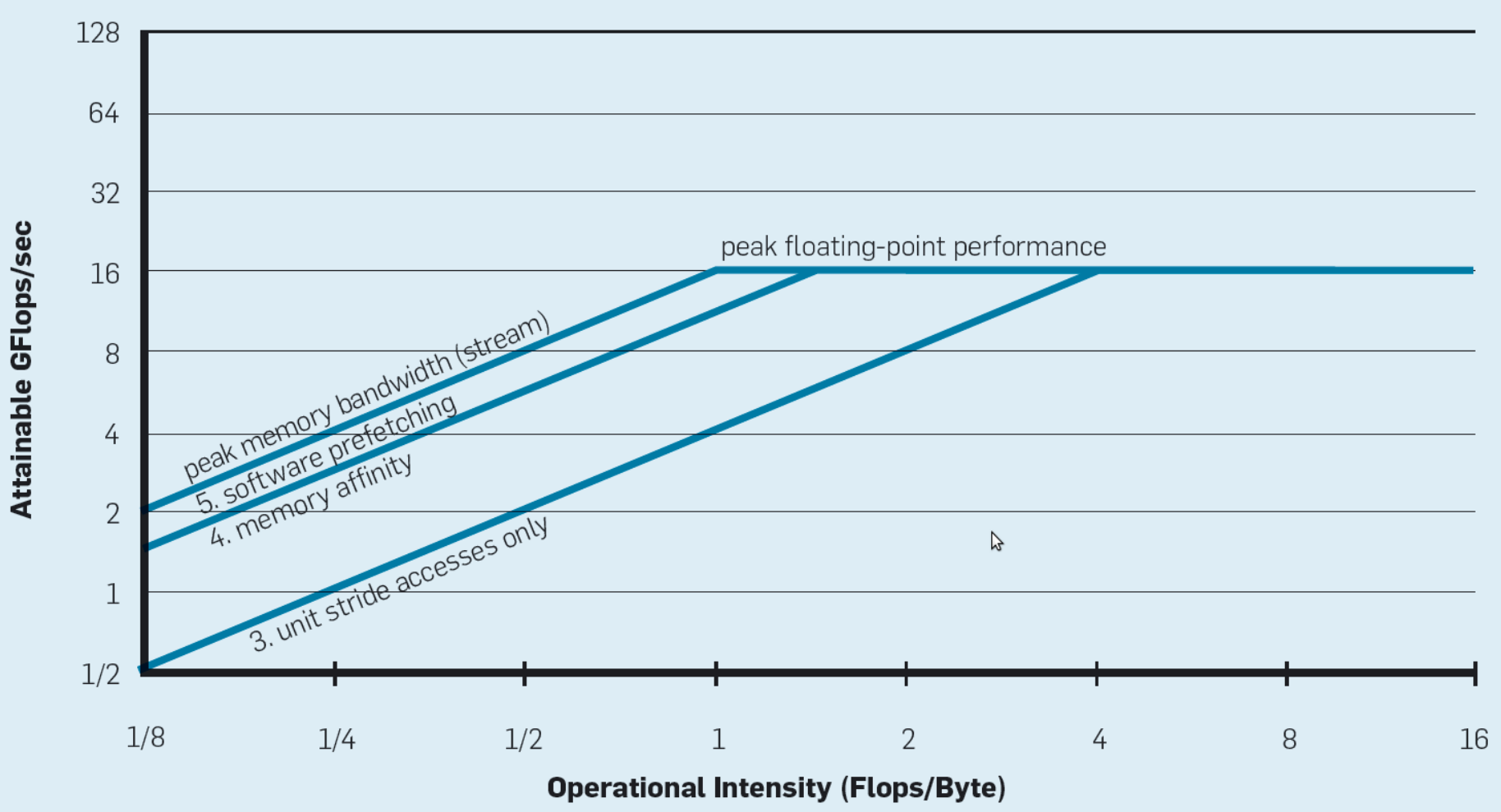
AMD Opteron X4: 73.6 gflops, 15 GB/s, ops/byte = 4.9

Roofline: computational ceilings



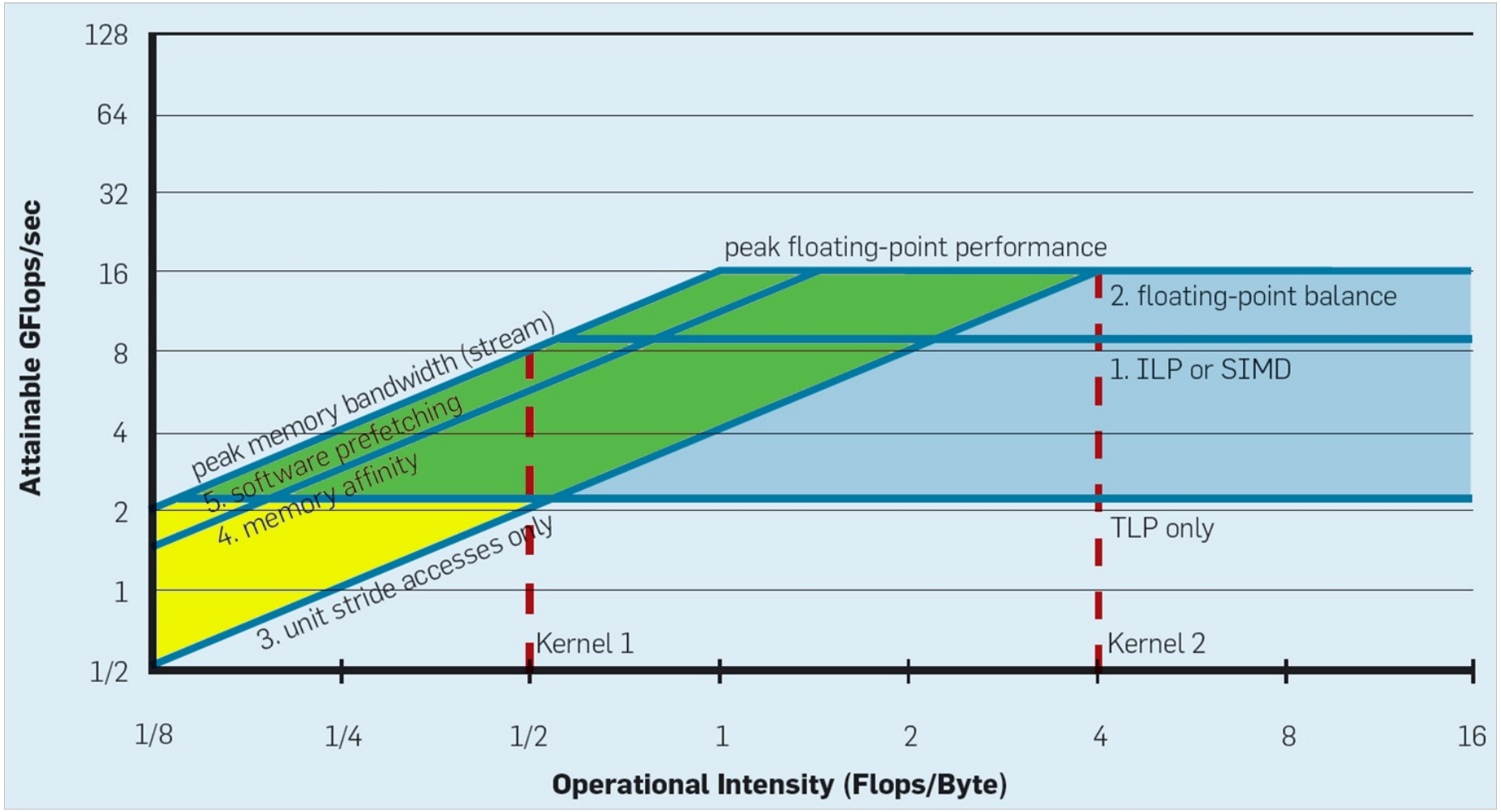
AMD Opteron X2 (two cores): 17.6 gflops, 15 GB/s, ops/byte = 1.17

Roofline: bandwidth ceilings



AMD Opteron X2 (two cores): 17.6 gflops, 15 GB/s, ops/byte = 1.17

Roofline: optimization regions



Use the Roofline model

- Determine what to do first to gain performance
  - Increase memory streaming rate
  - Apply in-core optimizations
  - Increase arithmetic intensity

- Read:

  Samuel Williams, Andrew Waterman, David Patterson

  “Roofline: an insightful visual performance model for multicore architectures”

Absolute hardware performance

- Only achieved in the optimal conditions:
  - Processing units 100% used
  - All parallelism 100% exploited
  - All data transfers at maximum bandwidth
- In real life
  - No application is like this
  - Can we reason about “real” performance?

“REAL” hardware performance

- Microbenchmarking*
  - Evaluates hardware features in isolation
  - Goal: find out the true limits of the hardware components
  - Platform-specific results
  - **Compared** with the theoretical peak, **per platform**.

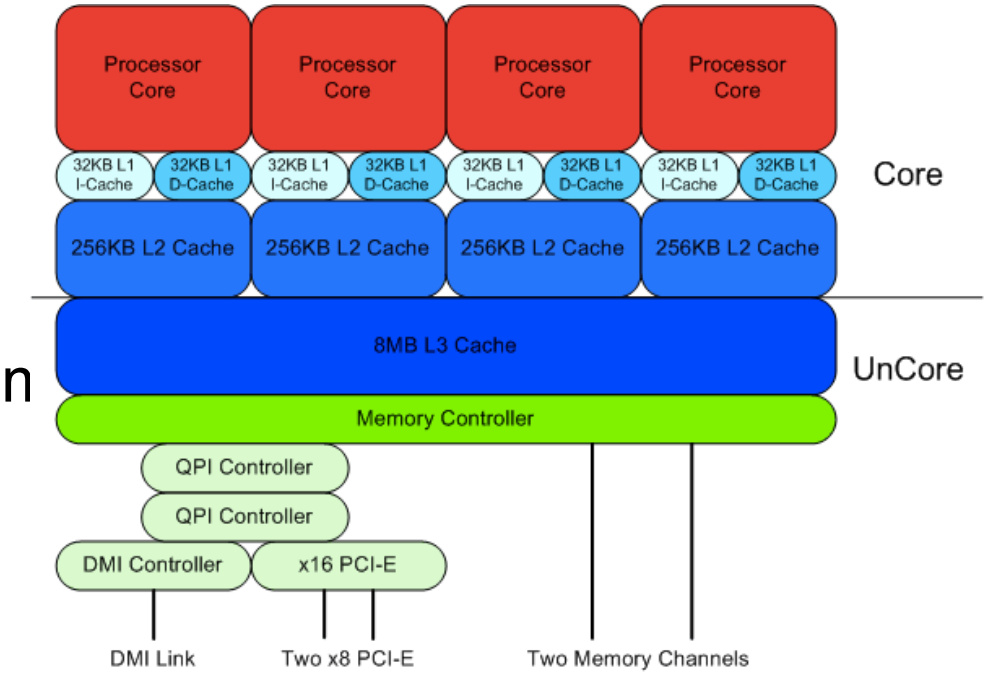
- Benchmarking
  - Evaluates the FULL platform
  - Application-specific performance
    - Top500 – computation capability
    - Graph500 – graph processing capability
    - Green500 – energy consumption
  - **Compares platforms**

* Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, Andreas Moshovos. ”Demystifying GPU

What if you want to know more?

Meet Hardware performance counters

- A set of special-purpose registers built into modern microprocessors
- Store the counts of hardware-related activities/events
- Counters = the actual registers
- Events = actual hardware events
  - Events / counters >> 1 => reprogramming the counters !
- Performance Monitoring Units: hardware units to n
  - Core: what happens at core level
  - Uncore: outside the core



Types of counters (examples)

Core-events

- instructions retired
- elapsed core clock ticks
- core frequency
- memory subsystem (L1, L2)

UnCore-events

- LLC
- Read/written bytes from/to memory controller(s)
- Data traffic transferred by the QPI links.

Literally hundreds and hundreds more

Warnings : complexity

- High-complexity of hardware => many different types of counters
  - It is rarely the case that a single event tells a complete story
- Intel splits events in *architectural* and *non-architectural*
  - i.e., processor independent vs. processor dependent
- Different generations => different *non-architectural* counters
  - Different names
  - Different meanings
- Typically used to confirm/infirm hypotheses
  - A counter on its own won't tell you much if you don't have any expectation …

Tools & methods

- Low-level assembly code
  - Set what needs to be counted …
  - … and keep reading the register!

- PAPI (http://icl.cs.utk.edu/papi/overview/index.html)
  - Portable interface across devices
  - Simple API to access most coun

- High-level tools
  - Intel VTune
  - AMD uProf
  - NVIDIA nsight/nvprof
  - **LIKWID**
    - Lots of tools to simplify collection

```
int event[NUM_EV]={PAPI_TOT_INS, PAPI_TOT_CYC, PAPI_L1_DCM };
long long values[NUM_EV];

/* Start counting events */
PAPI_start_counters(event, NUM_EV);
//call function
PAPI_read_counters(values, NUM_EV);

printf("Total instructions: %lld\n", values[0]);
/* Stop counting events */
PAPI_stop_counters(values, NUM_EVENTS)
```

Top500: the HPC pulse

HPC pulse

- TOP500 Project*
  - The 500 most powerful computers in the world
- Benchmark: Rmax of LINPACK
  - Solve the Ax=b linear system
    - dense problem
    - matrix A is random
  - Dominated by dense matrix-matrix multiply
- Metric: FLOPS/s
  - Computational throughput: number of floating point operations per second
- Updated twice a year: latest is June 2023

*Read more: www.top500.org

TOP5(00) JUNE 2023

1 Exascale machine

1 custom-built machine

3 energy-efficient machines: AMD, Intel+NVIDIA, IBM

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,824,768	238.70	304.47	7,404
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

Top500 June 2023

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,824,768	238.70	304.47	,404
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	0,096

Gap: 20 – 30%

Lots of **accelerators**.

Many many **many cores**.

High **peak** performance.

Large **gap** from peak to “real” performance.

Let’s talk about energy.

See more at: https://www.top500.org/

Green 500

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	Power (kW)	Energy Efficiency (GFlops/watts)
1	255	Henri - ThinkSystem SR670 V2, Intel Xeon Platinum 8362 32C 2.8GHz, NVIDIA H100 80GB PCIe, Infiniband HDR, Lenovo United States	8,288	2.88	44	65.396
2	34	Frontier TDS - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE United States	120,832	19.20	309	62.684
3	12	Adastra - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE France	319,072	46.10	921	58.021
4	17	Setonix – GPU - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Australia	181,248	27.16	477	56.983
5	77	Dardel GPU - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Sweden	52,864	8.26	146	56.491

PART 5: SUMMARY AND BEYOND

Where to?

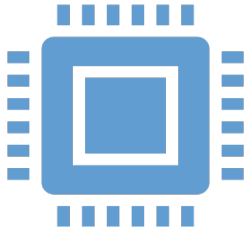
Today’s computing machines

- Parallel at different levels
  - Multi- or many-cores
  - Core-level parallelism
  - CPU-accelerator(s) parallelism
  - None-level parallelism
- Different performance and power “profiles” => different energy consumption => different energy efficiency envelops
- Hardware-level performance
  - FLOPs (or INTOPs) for computation
  - GB/s for memory bandwidth
  - FLOPS/Watt for energy efficiency
- Benchmarking machine performance
  - Micro-benchmarking vs. benchmarking
  - Diverse metrics => “performance counters”

Programming and programmability

- Diversity in hardware from nodes to system is challenging for programmers
- A multitude of programming models with different trade-offs between productivity/programmability and performance exist and emerge
- There exists important limitations in suppoting all models on all systems *effectively and efficiently*
  - … meaning choices (and therefore criteria for such choices) are needed
- System-level knowledge is difficult to acquire and use as programmer
  - But we should not stop trying.
- Tools for programming, debugging, modelling, analysis, benchmarking exist
  - But they could use further improvements.

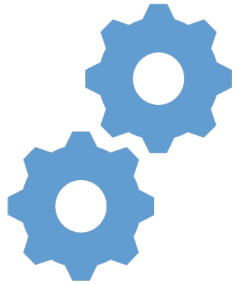
Performance/Efficiency will depend on …



Developers and users

Improve the energy efficiency of their own codes, making use of algorithmic, programming, and hardware tools

Design and implement applications able to adapt to the available system resources



System integrators

Offer the right mix of resources for the application developers and system operators.

Include efficient hardware to enable different application mixes.

System operators

Ensure efficient scheduling of workloads on system resources.

Harvest energy where resources/systems are massively underutilized.

Dolas, Sagar et al. “Making Scientific Research on Dutch National Supercomputer Energy Efficient.” *ERCIM News 131*, Oct. 2022
https://ercim-news.ercim.eu/en131/special/making-scientific-research-on-dutch-national-supercomputer-energy-efficient

Looking forward

- Larger and more complex systems
  - With a stronger focus on energy efficiency
- More heterogeneous systems
  - At node, blade, and rack level
  - At memory-system level
- More programming models, runtimes, and schedulers
  - Seamless integration possible, but unlikely
- Gradual performance shift from pure HPC towards efficiency and sustainability
  - Metrics and tools needed

From: https://image.shutterstock.com/image-vector/man-between-arrows-choice-past-260nw-1390413521.jpg
Adapted from: Jim Naylor at https://www.cartoonstock.com/cartoon?searchID=CS273198