

SUSTAINABLE COMPUTING

Part1: Introduction & systems at hand



**UNIVERSITY
OF TWENTE.**

Ana-Lucia Varbanescu

a.l.varbanescu@utwente.nl

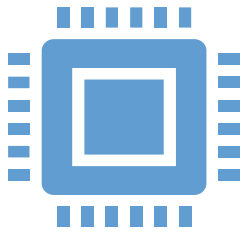
Computing is everywhere ... and it's not free!

- Top 10 videos on YouTube* consumed as much as 600-700 EU persons per year (or about 400 North America persons)
- Training Alpha-Zero for a new game consumes as much as 100 EU persons per year
- A mid-size datacenter alone consumes as much energy as a small town
 - And that is not considering purchasing and secondary operational costs (e.g., cooling)
- In 2011, the United States alone consumed more energy than any other nation
 - And that is not considering purchasing and secondary operational costs (e.g., cooling)
- The ICT sector is predicted to reach 21% of the global energy consumption by 2030

The energy consumption of computing is substantial and constantly increasing!

*https://en.wikipedia.org/wiki/List_of_most-viewed_YouTube_videos#Top_videos

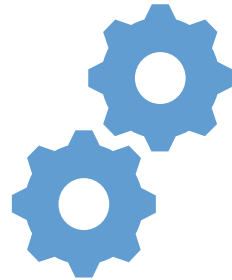
Three types of stakeholders



Developers and users

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

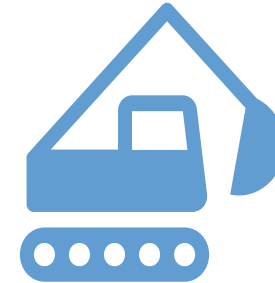
Design and implement applications able to adapt to the available system resources



System integrators

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

Include efficient hardware to enable different application mixes.

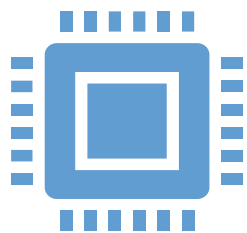


System operators

Ensure efficient scheduling of workloads on system resources.

Harvest energy where resources/systems are massively underutilized.

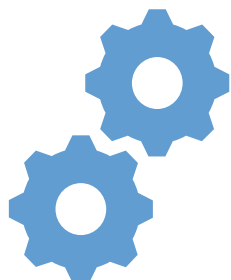
Three types of stakeholders



Developers and users

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

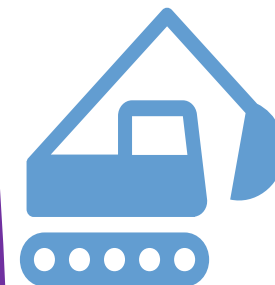
Design and implement applications able to adapt to the available system resources



System integrators

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

Include efficient hardware to enable different application mixes.



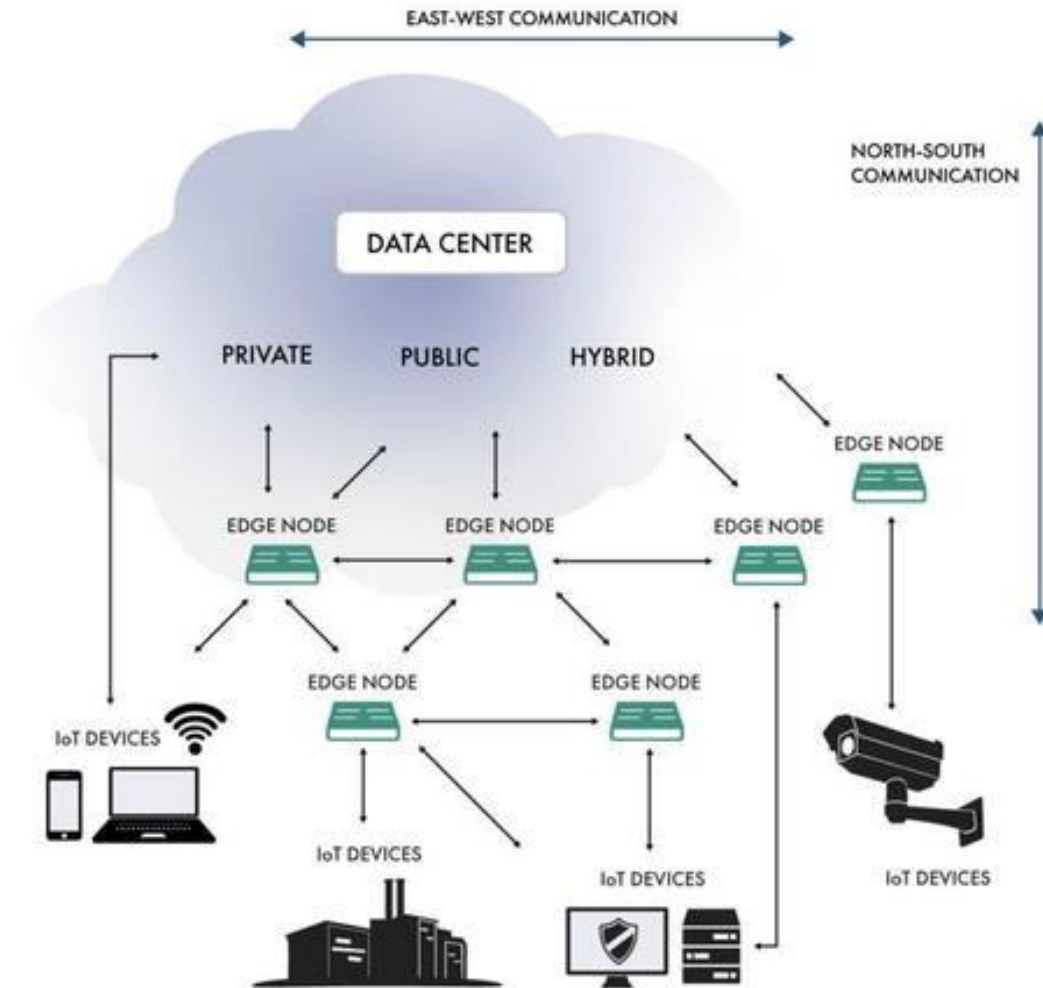
System operators

Ensure efficient scheduling of workloads on system resources.

Harvest energy where resources/systems are massively underutilized.

Systems

- On-premise hardware
 - Flexible, yet often limited in resources
 - Good for development
 - Limited value for production
- Supercomputers
 - Massive machinery, high-performance
 - Partially shared
 - Less flexible in terms of infra and programming
- Datacenters & Cloud computing
 - Scale-by-credit card
 - Excellent efficiency
 - Possible limitations in terms of performance (SLA)
- Computing continuum
 - New development in distributed computing
 - Unclear for scientific computing
 - Relevant for complete data analysis (sensor-to-result)



Supercomputing/Data-centers

- Supercomputing is extremely high in carbon emissions, mainly due to scale.
- **Embodied carbon:** Indirect emissions, e.g., production, shipping, and disposal of system components.
- **Operational carbon:** Electricity, heating, cooling, etc. for the site operation.



Fugaku
0.44Exa @30MW



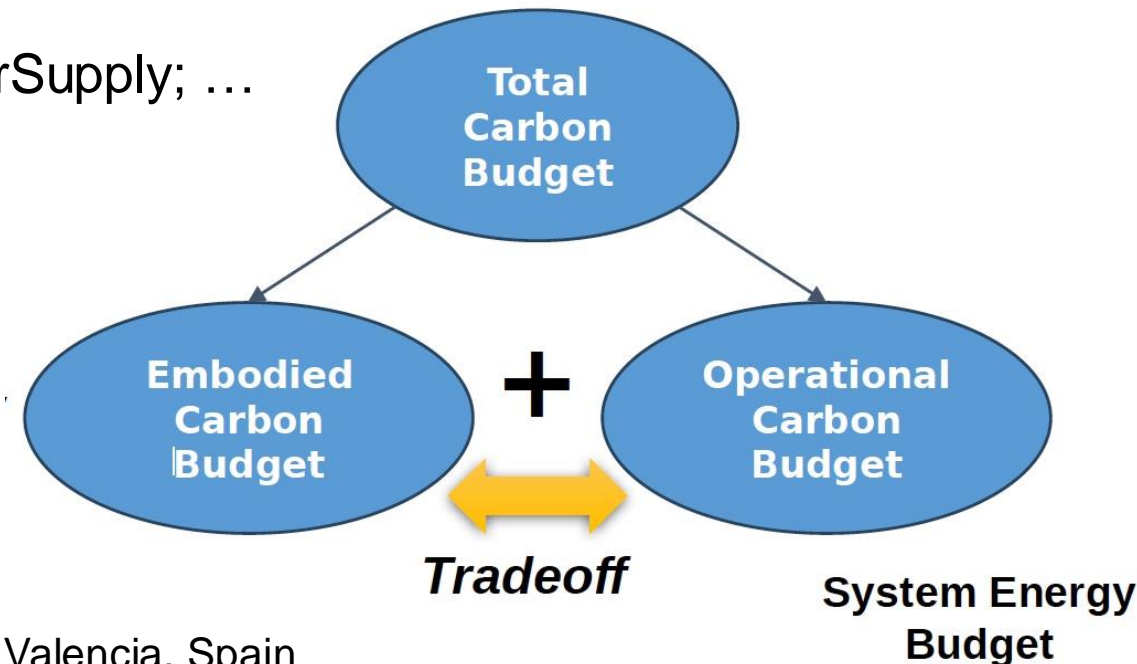
Frontier
1.2Exa @23MW



Aurora
2Exa? @60MW?

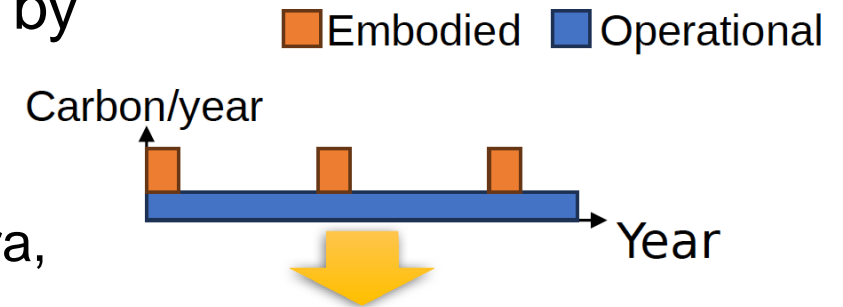
Acquisition

- New lifecycle assessment & procurement procedures
- **Current Goal:** Maximize Throughput (Workloads)
- **Constraints:**
 - Budget = Machine Cost + Electricity;
 - System Footprint/Weight; CoolingCapacity; PowerSupply; ...
- **New Constraint:** Carbon Budget



Extend lifetime

- Extend Lifetime, Reuse, and Recycle
- **System Lifetime:** Typically 4-6 years
 - Extended lifetime -> embodied carbon reduction.
- **Reuse & Recycle:** Reduce carbon emissions caused by disposal & production
 - Reuse: e.g., LRZ offers decommissioned machines for free.
 - Recycling: accelerators, DRAM chips, heat pipes, cooling infra,
...



Lifetime Extension
Reuse & Recycle

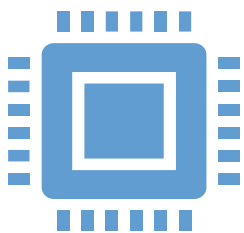


Operation

- Efficient operation
 - Schedulers
 - Automated tools
 - Support/education for users

- Additional opportunities
 - Shared resources
 - Location shifting
 - Time/Peak shifting

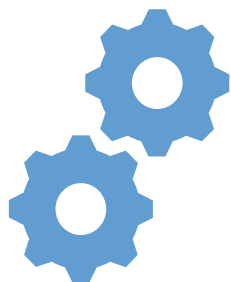
Three types of stakeholders



Developers and users

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

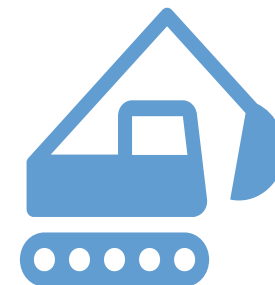
Design and implement applications able to adapt to the available system resources



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.

Developers & users

- Measure/Quantify
- Select the right systems
- Select the right implementation tools
- Select the right algorithms
- Tweak and tune ... and iterate

Agenda

- Different views on performance
 - Towards **zero-waste computing**
- Understand systems

- Understand applications
 - Performance engineering
- Methods and tools for sustainable computing
 - Energy consumption and efficiency
 - Beyond energy
- Take home message

Part 1

Part 2



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

Context

- Modern (and future) systems are parallel and heterogeneous
 - In many dimensions
- Systems are characterized by peak performance (with various “roofs”)
- All applications want more performance
 - Applications must enable parallelism
- One Application => n algorithms => $n*m$ implementations
 - Algorithms: characterized by complexity
 - Algorithms/implementations: characterized by arithmetic/operation intensity – ops/byte

We want more ...

- More speed => “higher performance”

- More pixels => “better resolution”

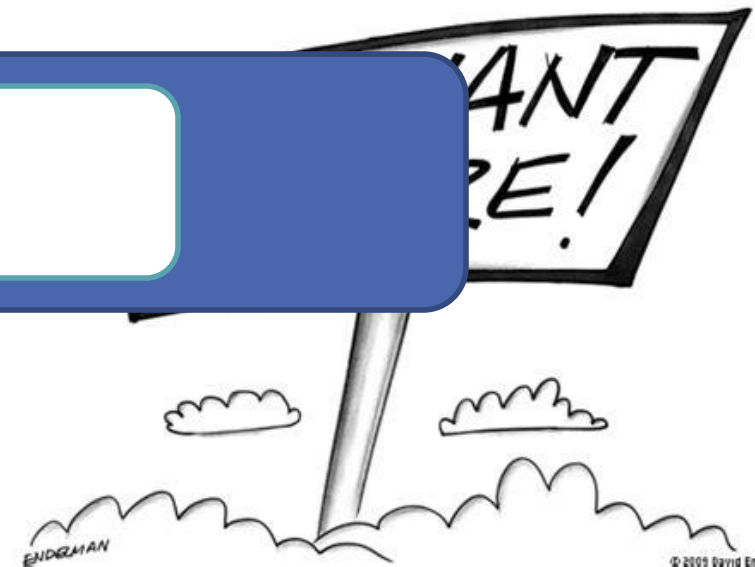
- More functio

We need want more compute !

- More ac

This is (potentially) inefficient!

- More re



Some relevant performance metrics*

- Speed-up: how much faster do we get with new machines, algorithms, ...

$$S(\text{workload}) = \text{Perf}(\text{Old}) / \text{Perf}(\text{New})$$

High-performance computing

- Efficiency: how efficient are we in getting performance

$$E(\text{workload}) = \text{Perf} / \text{Resources}$$

- Energy efficiency: how energy efficient are we in getting performance

$$EE(\text{workload}) = \text{Perf} / \text{Energy}$$

High-efficiency computing

- Utilization: how efficient are we utilizing our resources

$$U(\text{resource}) = \text{Achieved} / \text{Peak}$$

**please accept the naïve notation and pseudo-definitions*

Waste in computing

Unnecessary time (or energy) spent in (inefficient) computing is **compute waste**.

We **all can and must** improve software and hardware ***efficiency*** to minimize waste in computing!

To reduce compute waste, we must shift from time-to-solution towards **efficiency-to-solution**

Why is compute efficiency challenging?

It is a **nonfunctional** requirement

Focuses on user-“irrelevant” issues like resource utilization, scalability, ...

We all make a lot of excuses

It's so ... **and** new applications and new computing systems

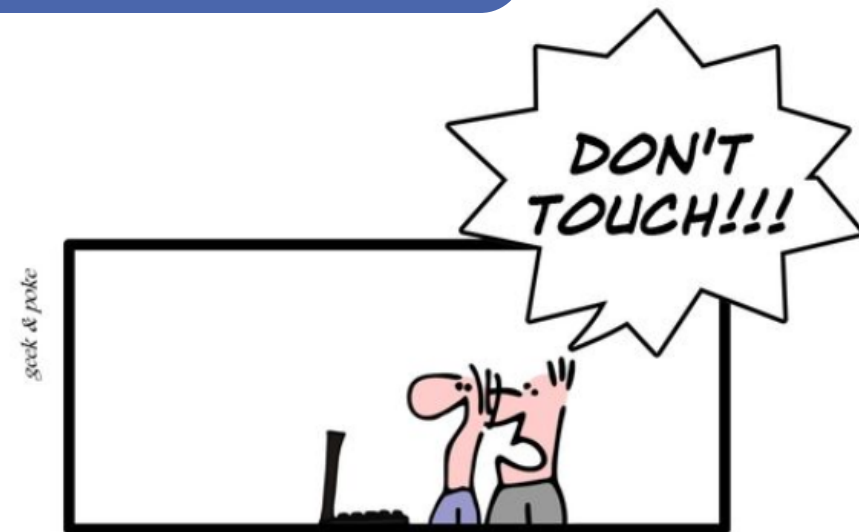
It's just ... emerge monthly ...

- More

It's easy to fix later

It's “just engineering”

Requires effort,
and there's (often) little glory in it.



Detecting and reducing waste

- We assume computing waste is a consequence of underutilized resources.

- Informally, assume:

`system1` > `system2`

`P1` = performance(algorithm, workload, `system1`)

`P2` = performance(algorithm, workload, `system2`)

- “Strict” definition:

if (`P1` == `P2`) => waste

- “Relaxed” definition:

if (abs (`P1` - `P2`) < T) => waste

with T = threshold for performance loss

Challenges in both
efficiency
quantification and
improvement.

Reducing waste in computing

Raise awareness

- Monitor (energy) efficiency
- Quantify waste

Improve efficiency

- Improve applications for the **systems at hand**
 - Make applications more efficient
 - Make applications share systems
- Improve systems for the **applications at hand**
- Co-design applications and systems

Performance analysis

Performance modeling

Performance optimization

Efficient scheduling and
resource sharing

Application-centric system
design

??

Systems SOTA: Supercomputers

Hardware developments: parallel & heterogeneous

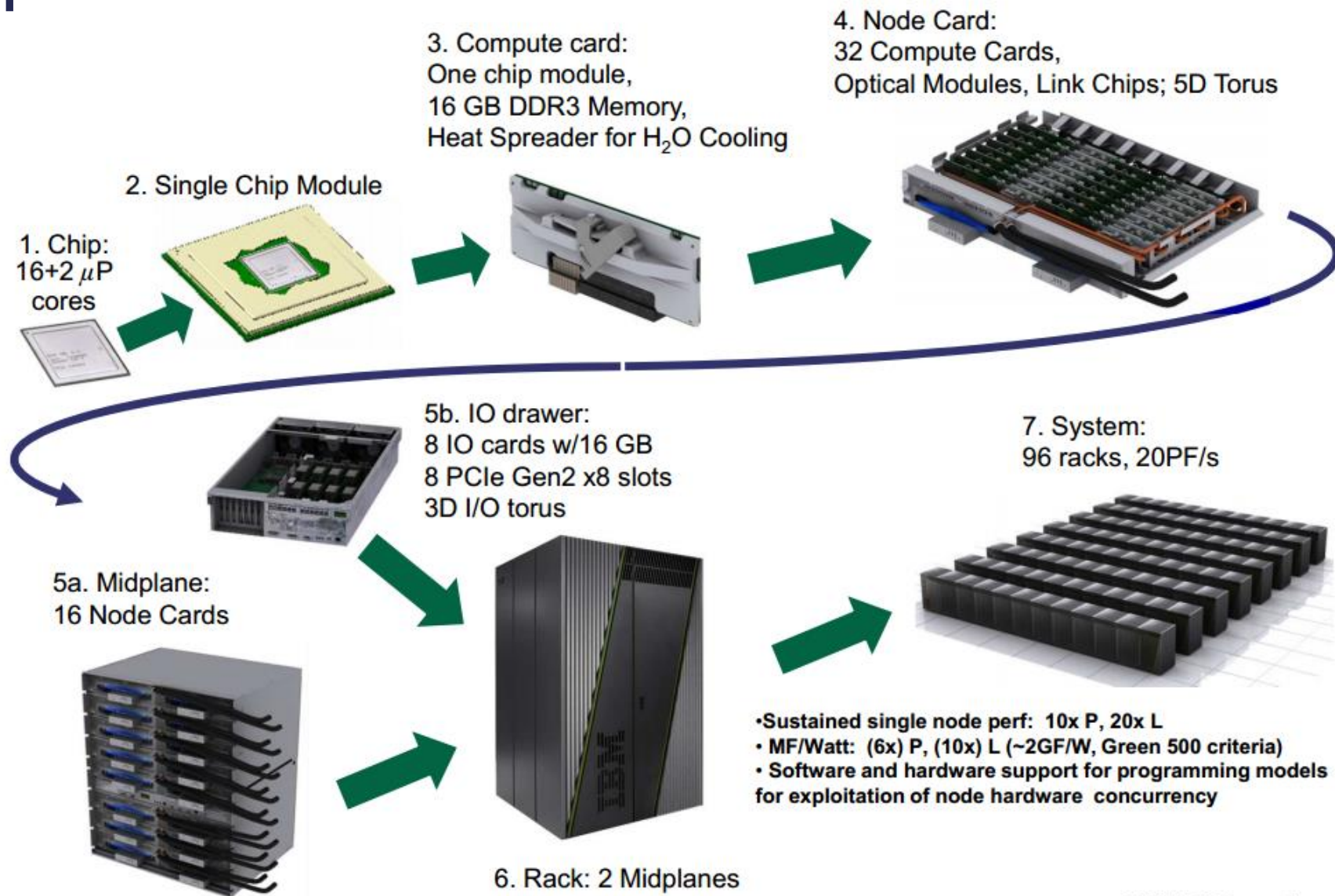
- More complex CPUs & GPUs
 - See Fugaku, Sunway TaihuLight
- GPUs from different vendors and different capabilities
- DPUs
 - Integrated CPUs+GPUs+DPUs (and other forms of HW)
- AI accelerators
 - Google's TPUS
- New machines for AI
 - GraphCore, Cerebra
- Don't forget FPGAs

And then put them together in supercomputers ...
Or datacenters ...

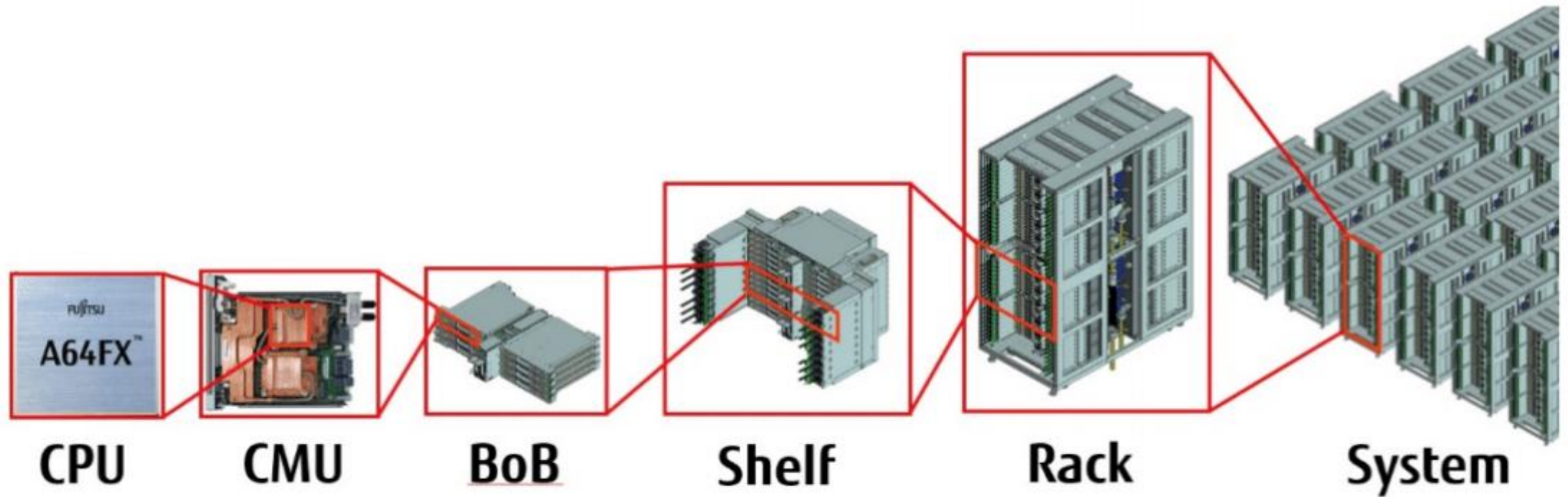
Supercomputers

- **Powerful computers** used for science, technology/engineering and AI
- Built as extremely **large computer systems**, with 100K's "basic" components and many billion transistors
 - "replicate" architectural patterns from nodes to blades to racks/cabinets.
 - interconnect each of these components with fast and/or efficient networks.
- All the processors in a supercomputer can perform computations at the same time => parallel computing.
 - Faster progress than sequential systems ...
 - ...iff parallel code exists.

Example: IBM's BlueGene/Q



Example: FUGAKU



Example: SUMMIT

Summit Overview



Components

IBM POWER9

- 22 Cores
- 4 Threads/core
- 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

NVIDIA GV100

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



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



Top500

@June'24

1 Exascale machine

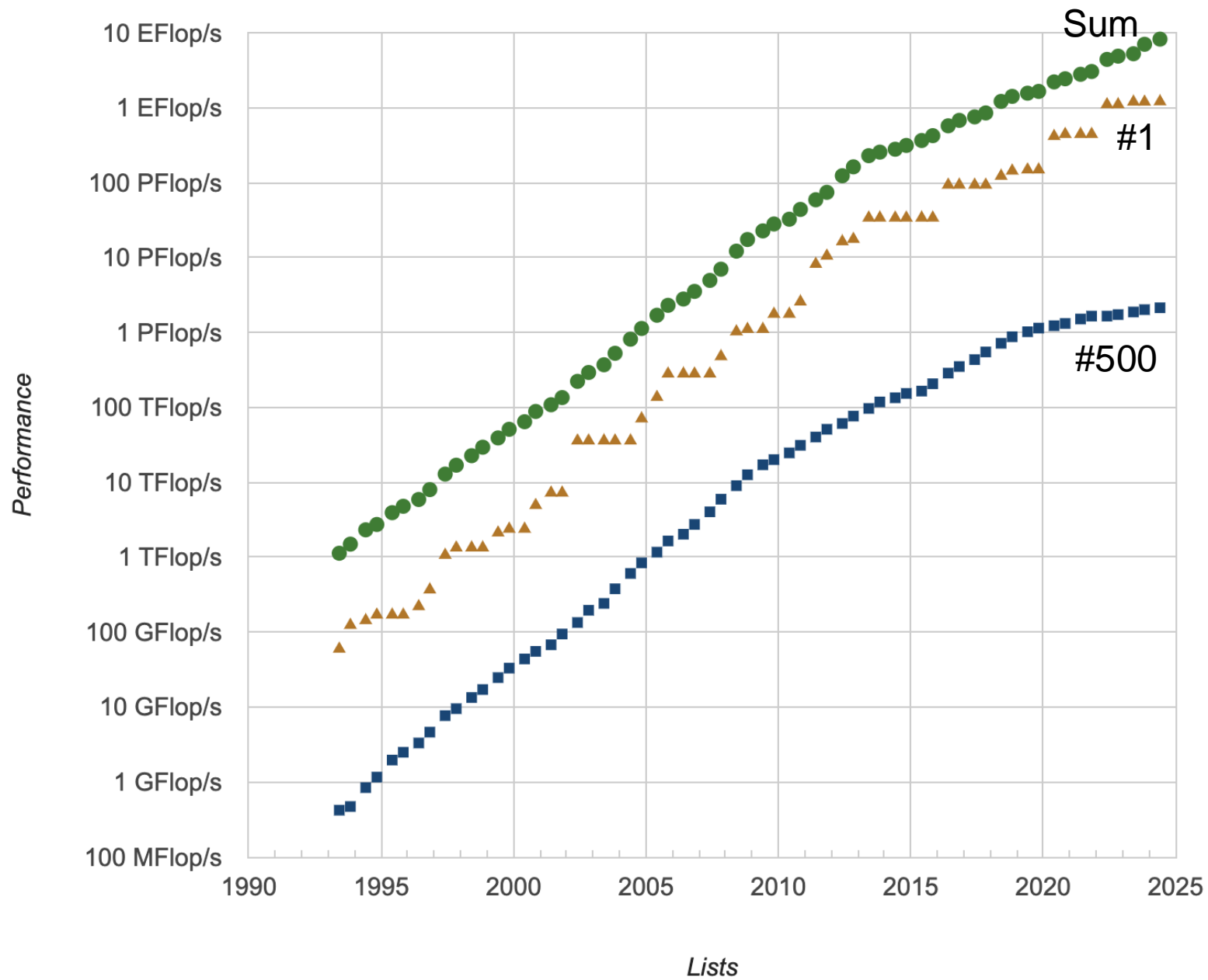
Heterogeneous
AMD, Intel+NVIDIA

1 custom-built machine

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	1,206.00	1,714.8	22,786
2	Aurora - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	Eagle - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.2	29,899
5	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.5	7,107

Gap: 20 – 30%

Performance Development



Green 500

@June 2024

None of the top5 ...

LUMI is #12

Aurora, Fugaku > #50

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	Power (kW)	Energy Efficiency (GFlops/watts)
1	189	JEDI - BullSequana XH3000, Grace Hopper Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, ParTec/EVIDEN EuroHPC/FZJ Germany	19,584	4.50	67	72.733
2	128	Isambard-AI phase 1 - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE University of Bristol United Kingdom	34,272	7.42	117	68.835
3	55	Helios GPU - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cyfronet Poland	89,760	19.14	317	66.948
4	328	Henri - ThinkSystem SR670 V2, Intel Xeon Platinum 8362 32C 2.8GHz, NVIDIA H100 80GB PCIe, Infiniband HDR, Lenovo Flatiron Institute United States	8,288	2.88	44	65.396
5	71	preAlps - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Swiss National Supercomputing Centre (CSCS)	81,600	15.47	240	64.381

Parallel System Models

- Shared Memory

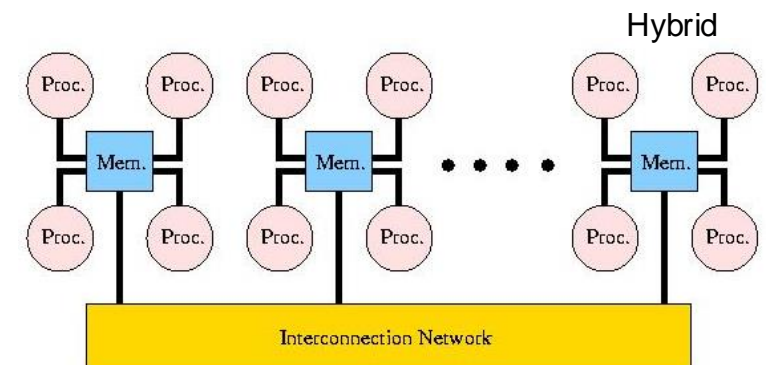
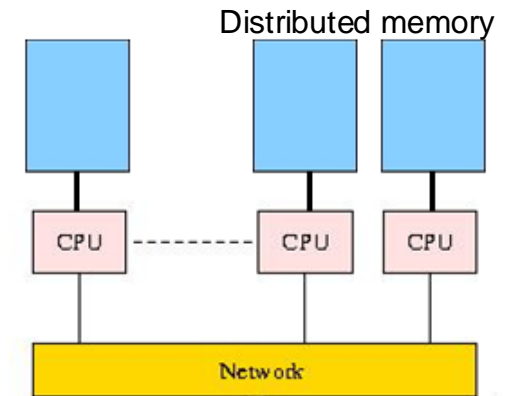
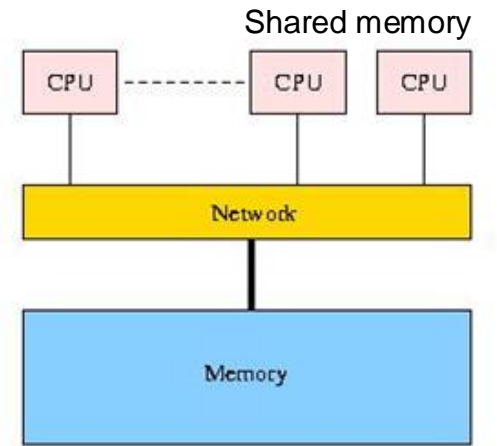
- Multiple compute nodes
- One single shared address space
- Typical example: multi-cores

- 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

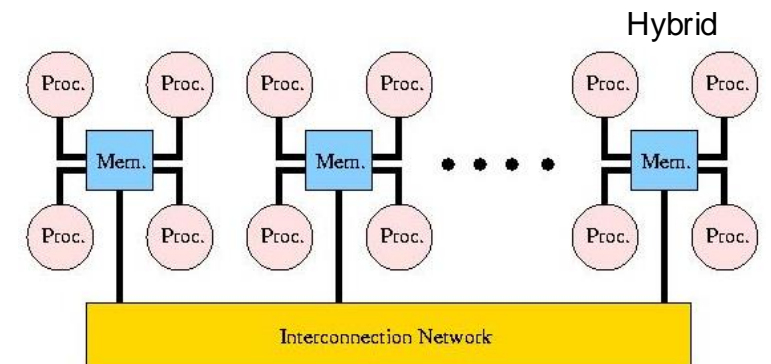
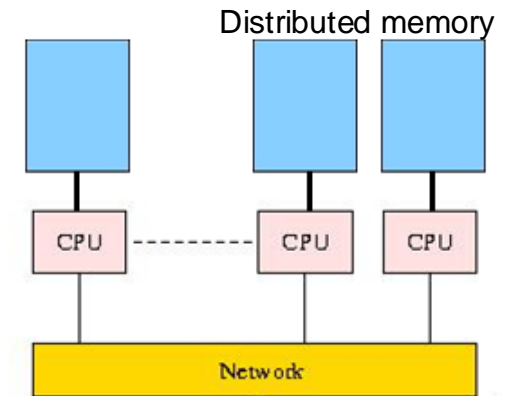
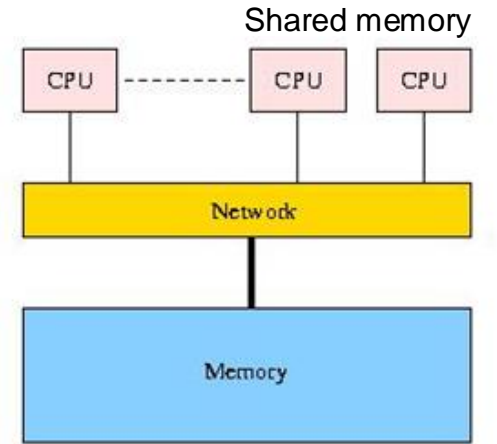
Programming: multi-threading
Programming models: OpenMP, pthreads, TBB, ...

- Distributed Memory

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

- Hybrids

Programming: very diverse, depending on the hardware configuration

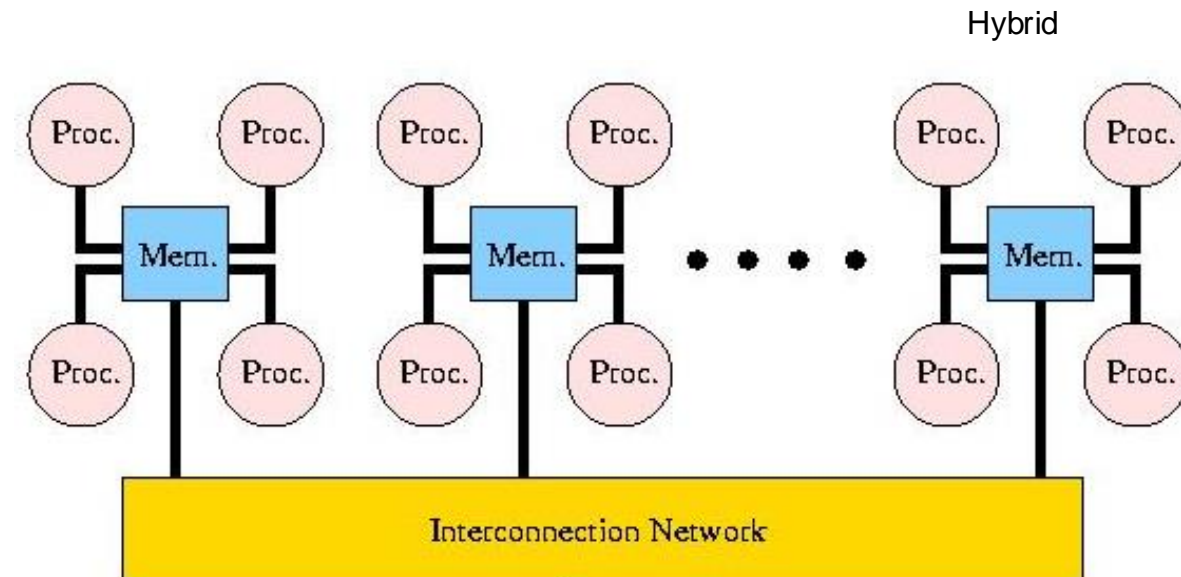


es" shared memory

ory

Main challenge: scaling to ExaFLOPS and beyond

- Peak performance = sum of capabilities of all machines
 - E.g.: 100 nodes x $\underbrace{128 \text{ cores} \times 100 \text{GFLOPs/core}}$
- Scaling options:
 - More nodes = scale out
 - More powerful nodes = scale up (or acceleration/heterogeneity)
- Limitations to actual performance
 - Memory, I/O, networking bottlenecks
 - Load-imbalance
 - Non-uniform behaviour
 - Programmability



How to scale-up/-out?

- Shared Memory model \Leftarrow typical for scale-up, limited for scale-out
 - Interconnect scalability problems & uniform accesses
 - Programming challenge: RD/WR Conflicts
- Distributed Memory model \Leftarrow typical for scale-out, inefficient for scale-up
 - Data distribution is mandatory
 - Programming challenge: remote accesses, consistency
- Virtual Shared Memory model \Leftarrow increased programmability and overhead
 - Significant virtualization overhead
 - Easier programming
- Hybrid models \Leftarrow trade-offs at different levels!
 - Local/remote data more difficult to trace

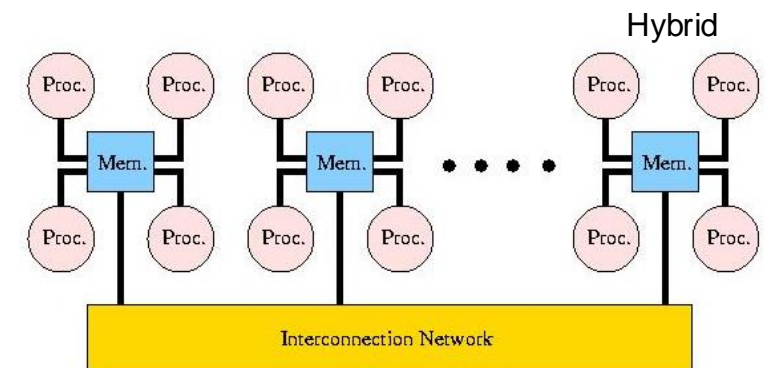
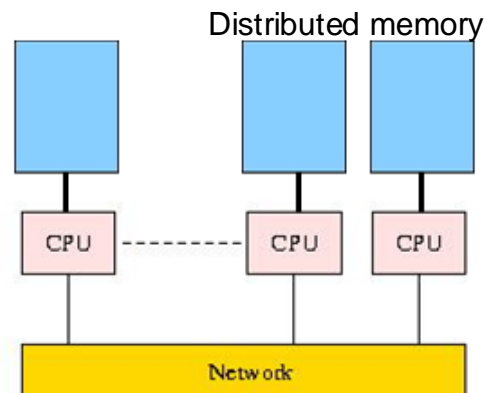
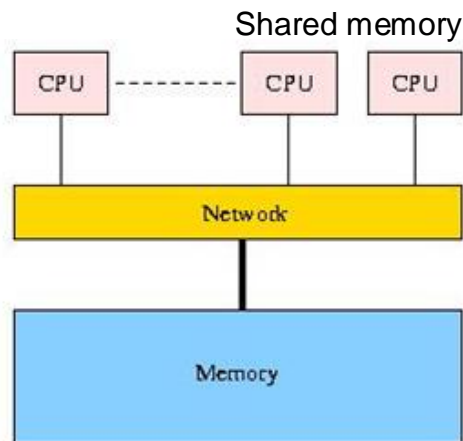
What about sustainability?

- **Efficient use of the system**

- Low performance => Low utilization => Low efficiency
- Inefficient resource reservation => Low efficiency

- Application scaling + programming \Leftrightarrow system architecture

- E.g.: calculate the histogram of a very large dataset on a (small) number of bins.



Programmer vs. runtime/OS vs. job scheduler

- Programmer exposes parallelism at application level
 - Job = application + dataset
 - Application = set of tasks
 - Tasks = execute in some sequential order and/or in parallel
- Runtime/OS map the tasks on resources
 - In both space and time
 - Possibly with programmer's restrictions
- (Job) Scheduler enforces constraints on resources
 - Ideally sufficient and

How to split and program the tasks? How is data accessed?

Knowledge of node architecture is essential for effective optimization.

What runs where and when?

Decisions by a runtime system and/or OS; require deep knowledge system architecture.

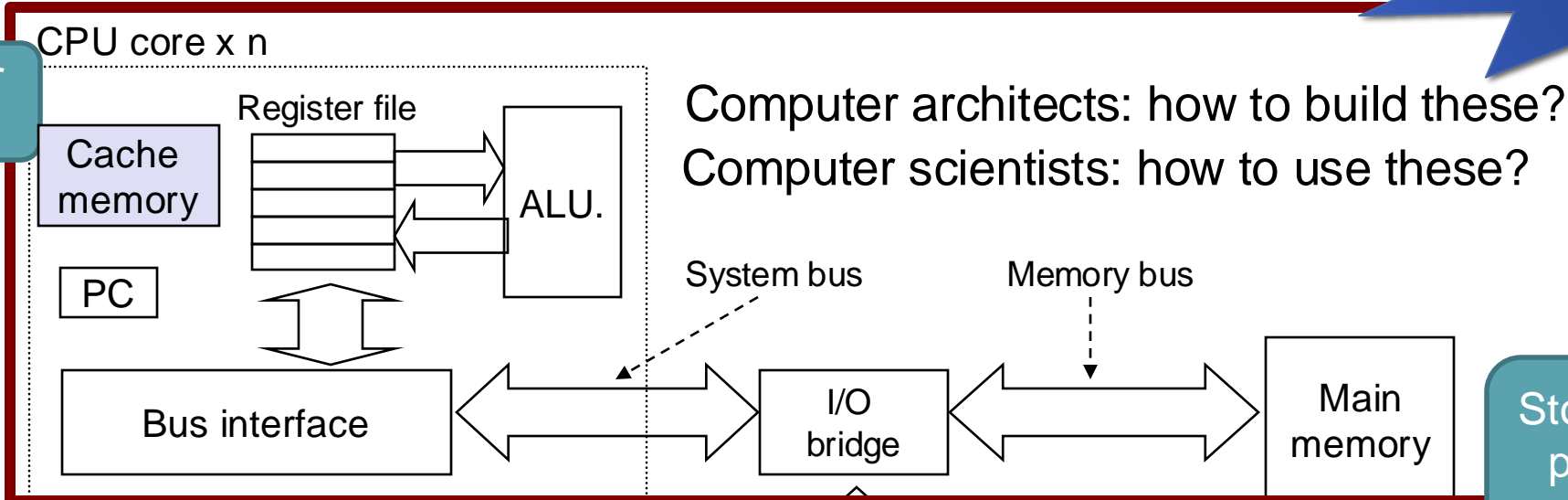
Inefficient parallel application design affects all levels in terms of efficiency and sustainability!

Systems SOTA: Inside the node

Inside the node*

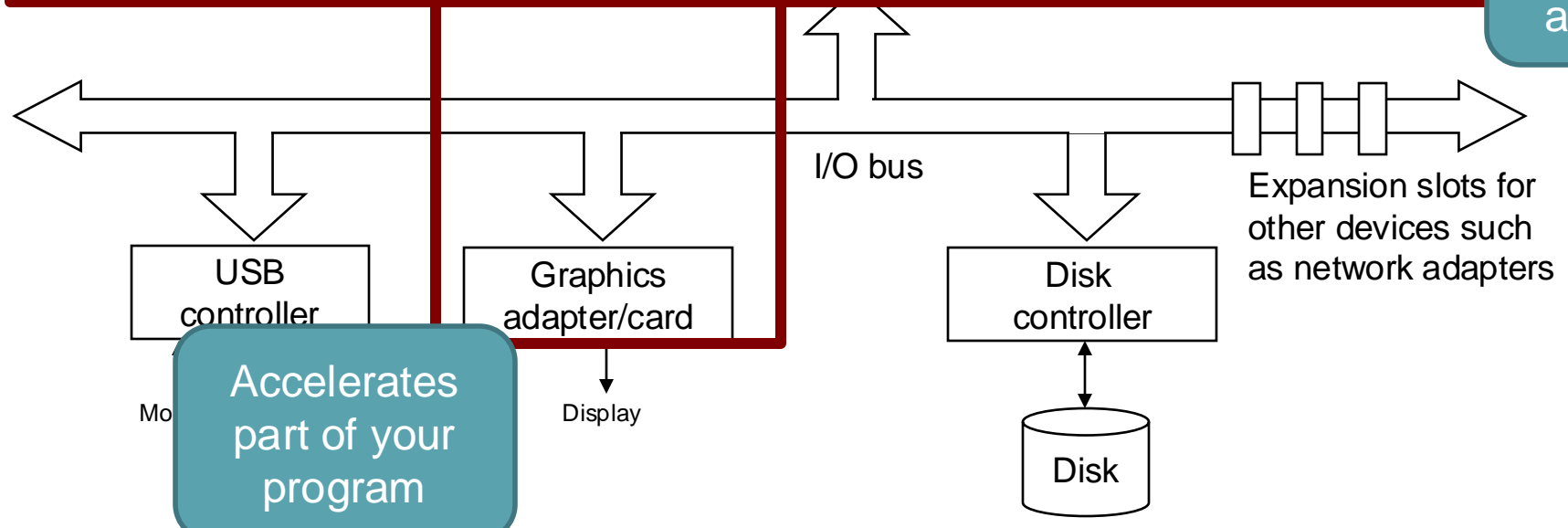
But the borders are blurrier every year.

Executes your program



Computer architects: how to build these?
Computer scientists: how to use these?

Stores your program and data



Accelerates part of your program

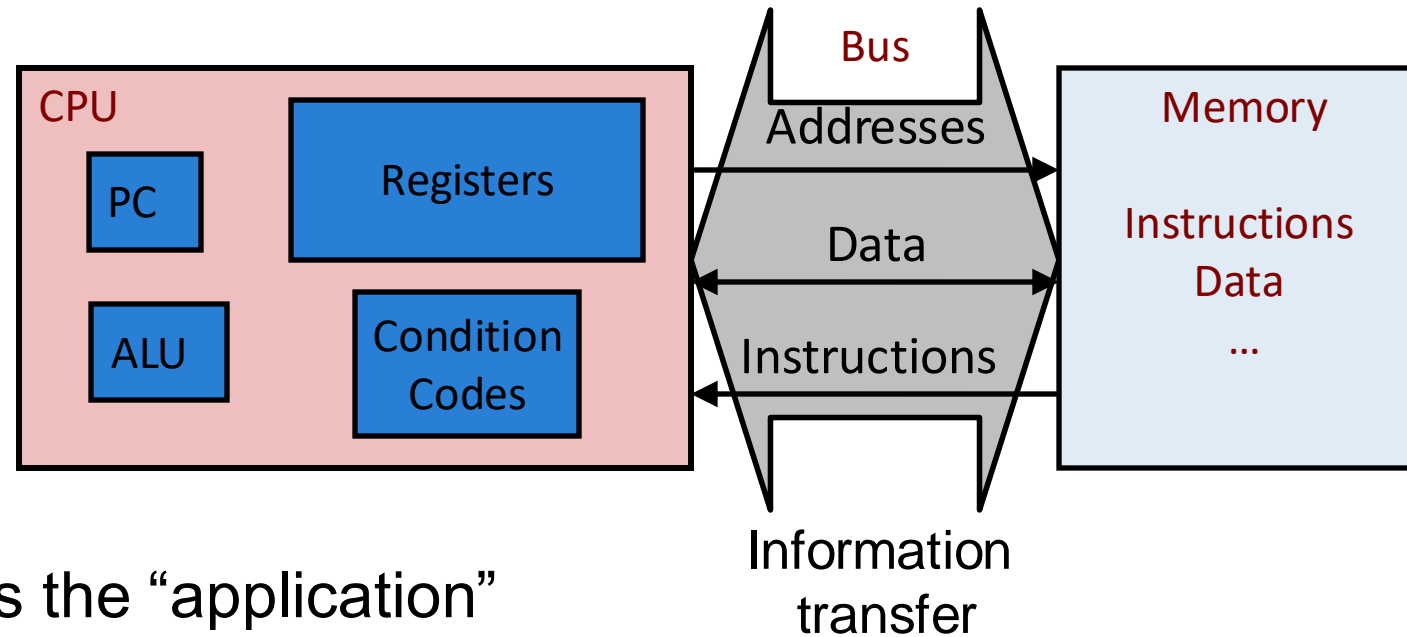
*Adapted from "Computer Systems: A Programmer's View" (Ch1) by Bryant and O'Hallaron

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

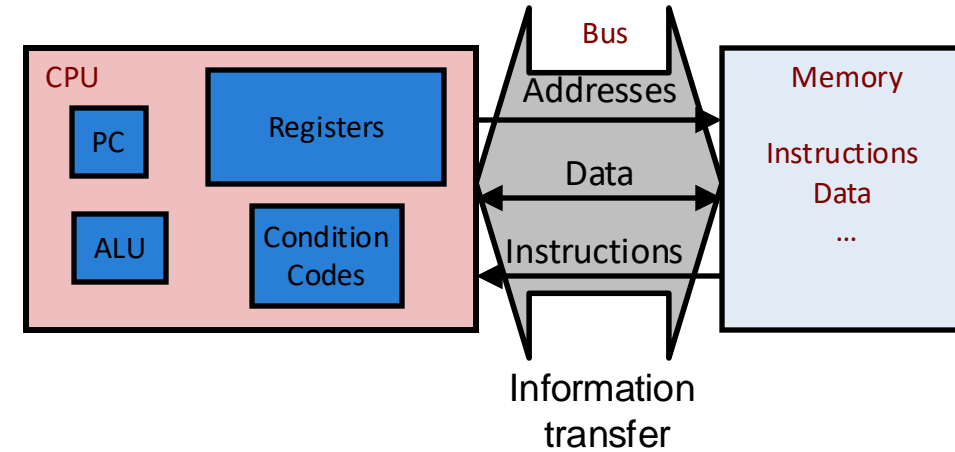
A processor's inner workings



- CPU = executes the “application”
 - Manages the execution progress (PC)
 - Fetches needed instructions and data (addresses)
 - Executes (ALU) operations and manages results
- Memory = stores the executable code of the application and the data
 - Receives request + address, replies with data (a bit vector)
- Bus = facilitates information (=bits) movement

The CPU

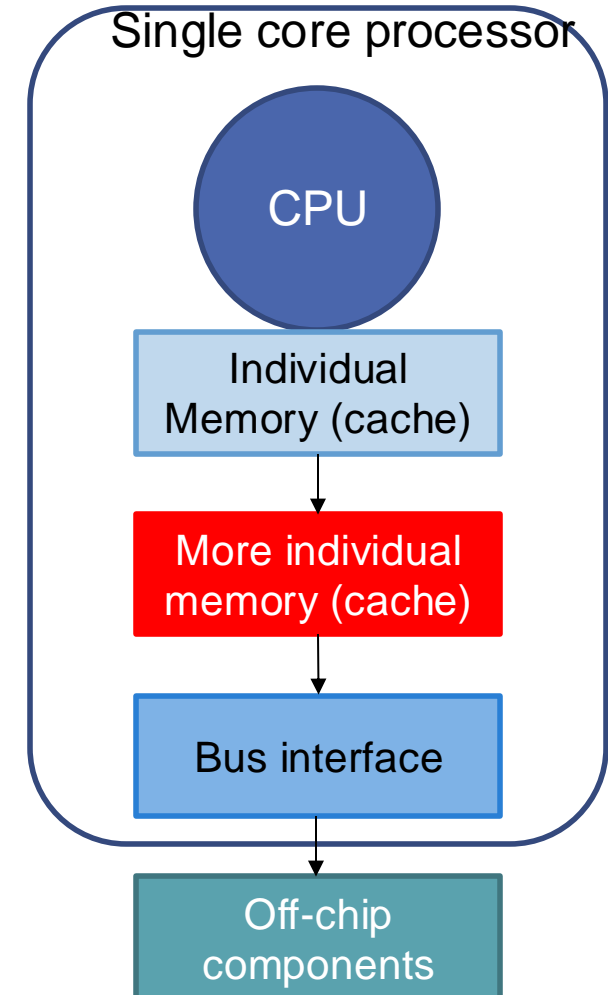
- Computations are executed by the ALU
 - Integer, single/double precision arithmetic, ...
 - Comparisons, logical operations, ...
- ALU runs at its own “clock speed” / frequency
 - Defines how many cycles/s can be executed by the CPU
 - Each operation takes 1 or more cycles
- Higher performance CPUs
 - Make a faster/smarter ALU
 - More operations per cycle
 - Make faster CPUs
 - More cycles/s
 - Multiple cores
 - Even more operations per cycle!



All these are powered!
Unused => low efficiency => waste!

Traditionally ... single core CPUs

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



Not anymore!

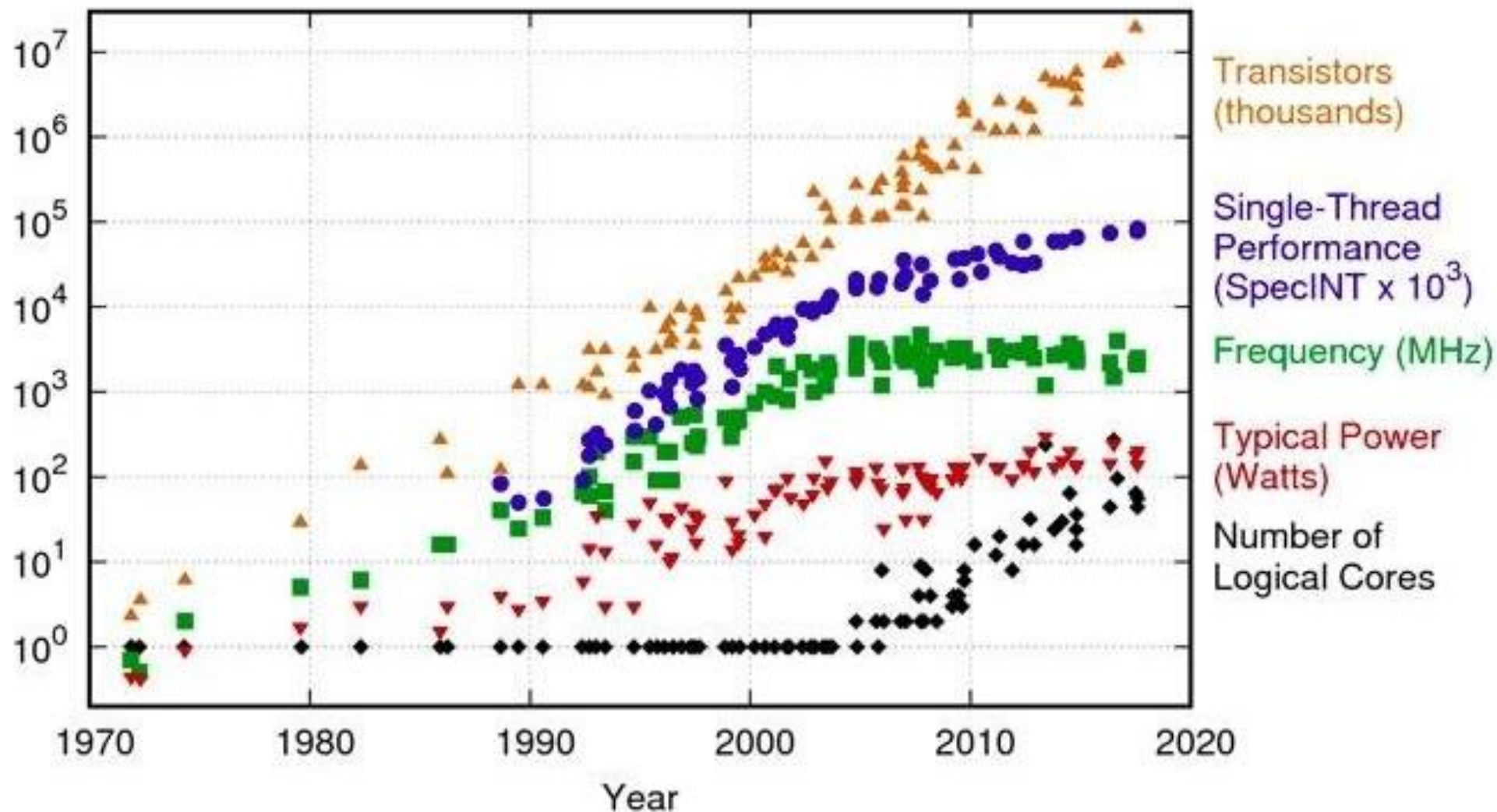
We no longer gain performance by “growing” sequential processors ...

New ways to use transistors

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

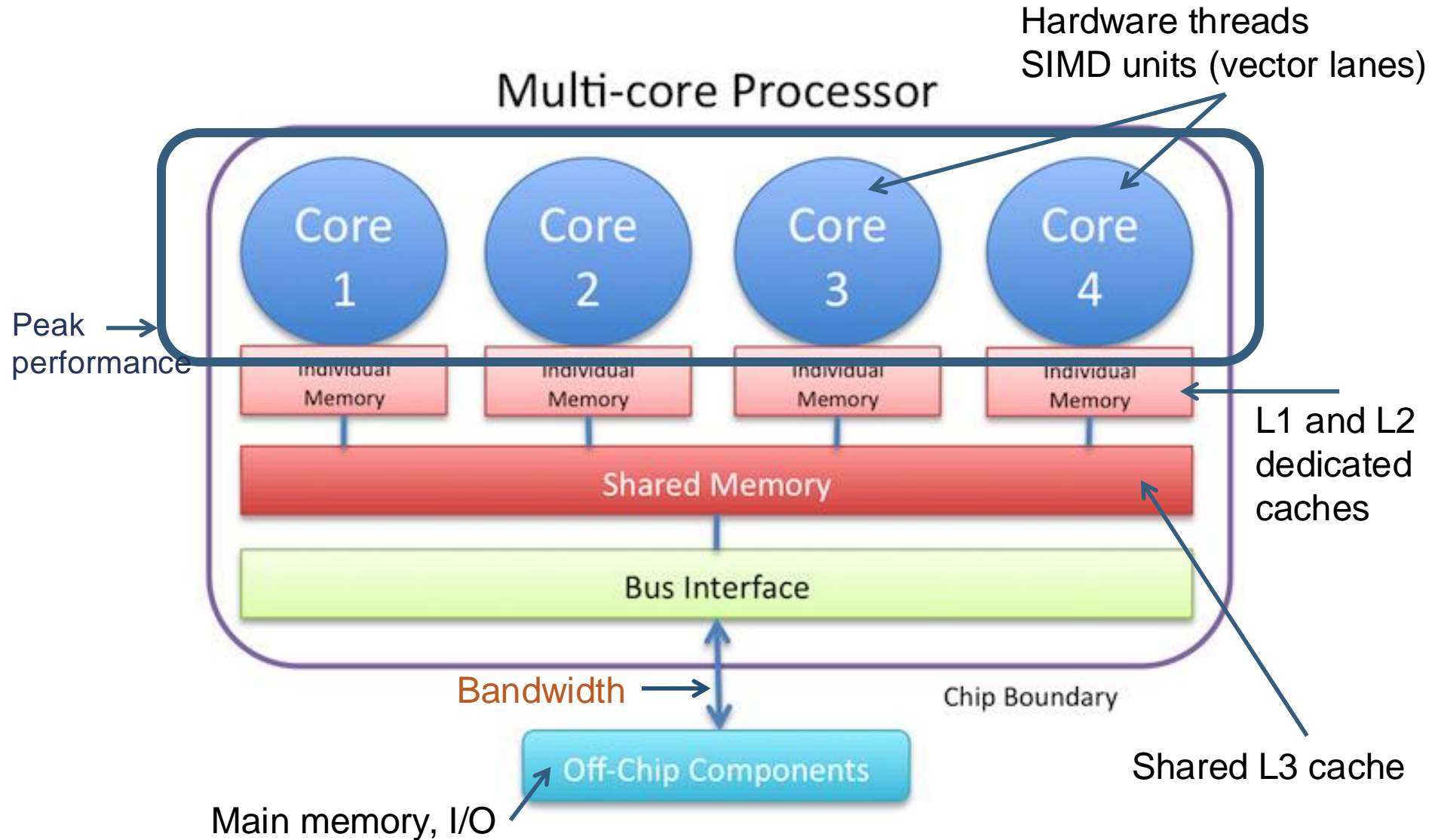


The shift to multi-core



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

Generic multi-core CPU



CPU levels of parallelism

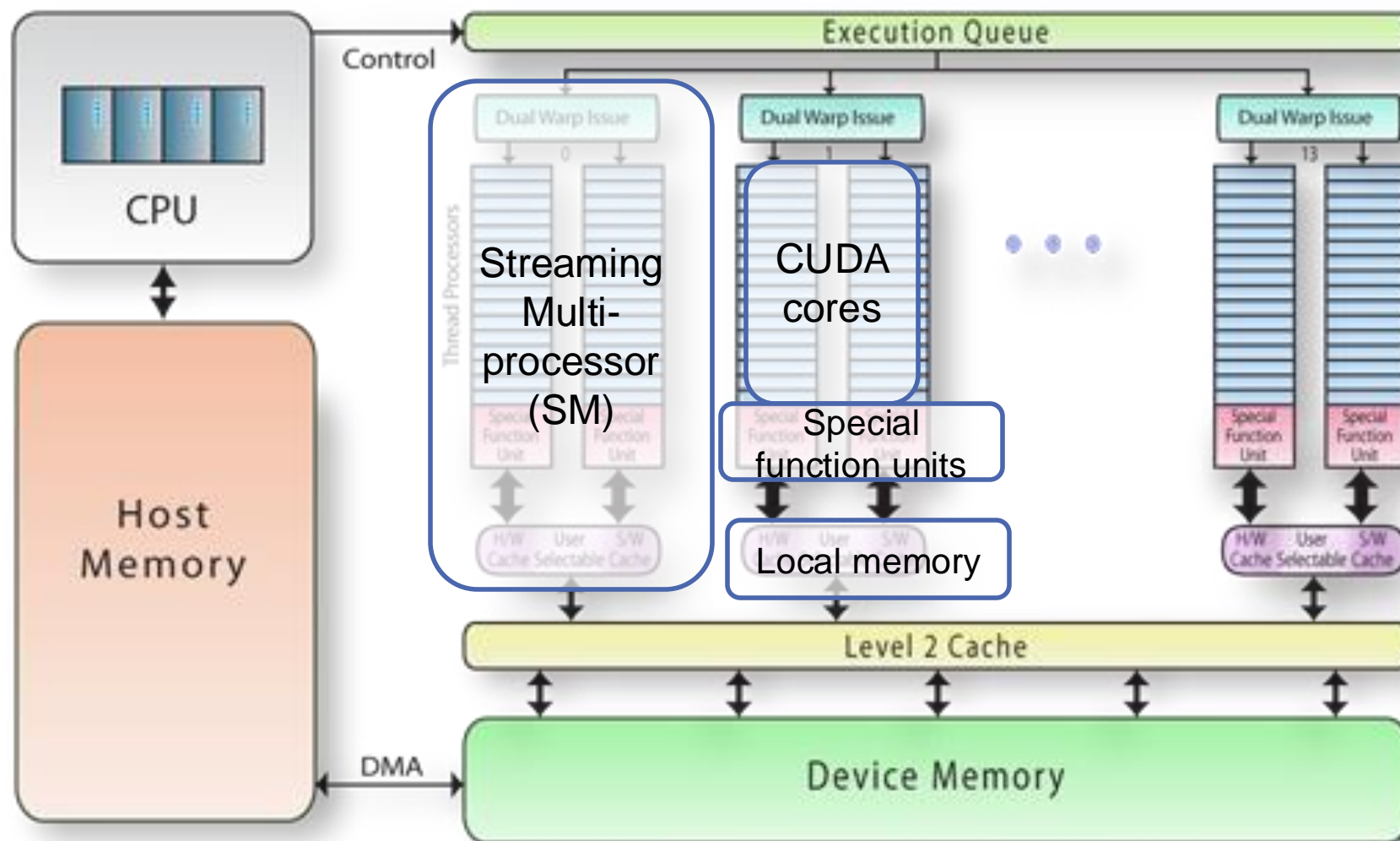
- **Instruction-level** parallelism (e.g., superscalar processors) (fine)
 - Multiple operations of different kinds per cycle
 - Implemented/supported by the instruction scheduler
 - typically in hardware
- **SIMD** parallelism = **data parallelism** (fine)
 - Multiple operations *of the same kind* per 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**

No programmer's intervention!

Some programmer/compiler intervention!

Programmer's intervention and OS support

Accelerators: a generic GPU

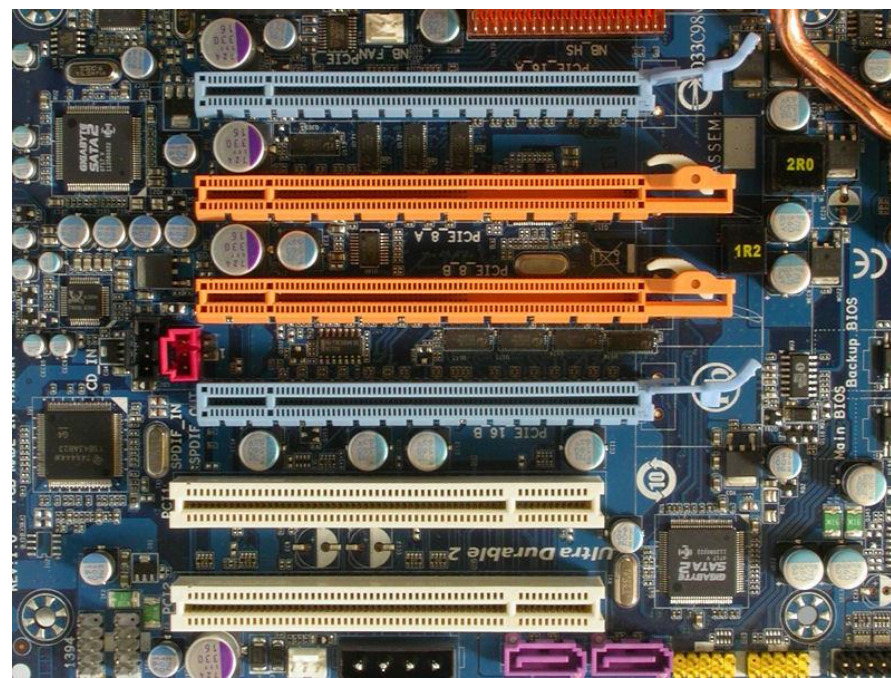
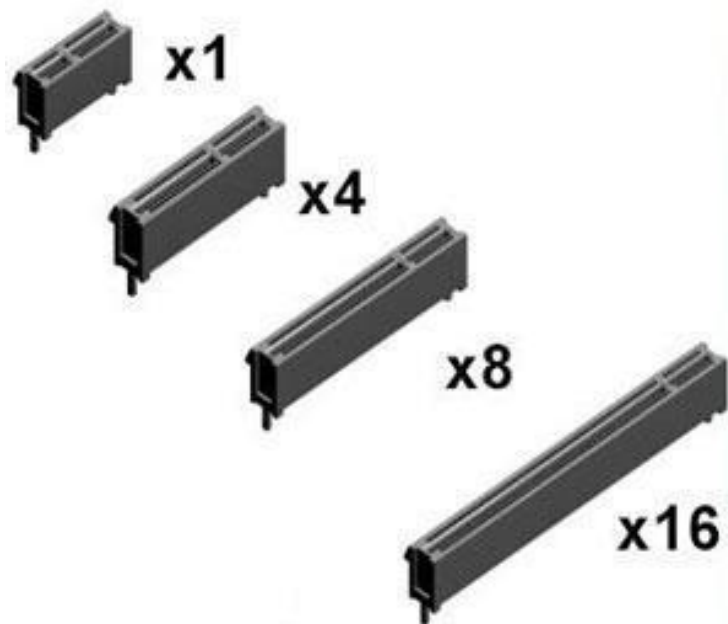


Inside an NVIDIA GPU architecture



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



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 ...

Programmer's (or some compiler's) intervention!

Programmer's intervention!

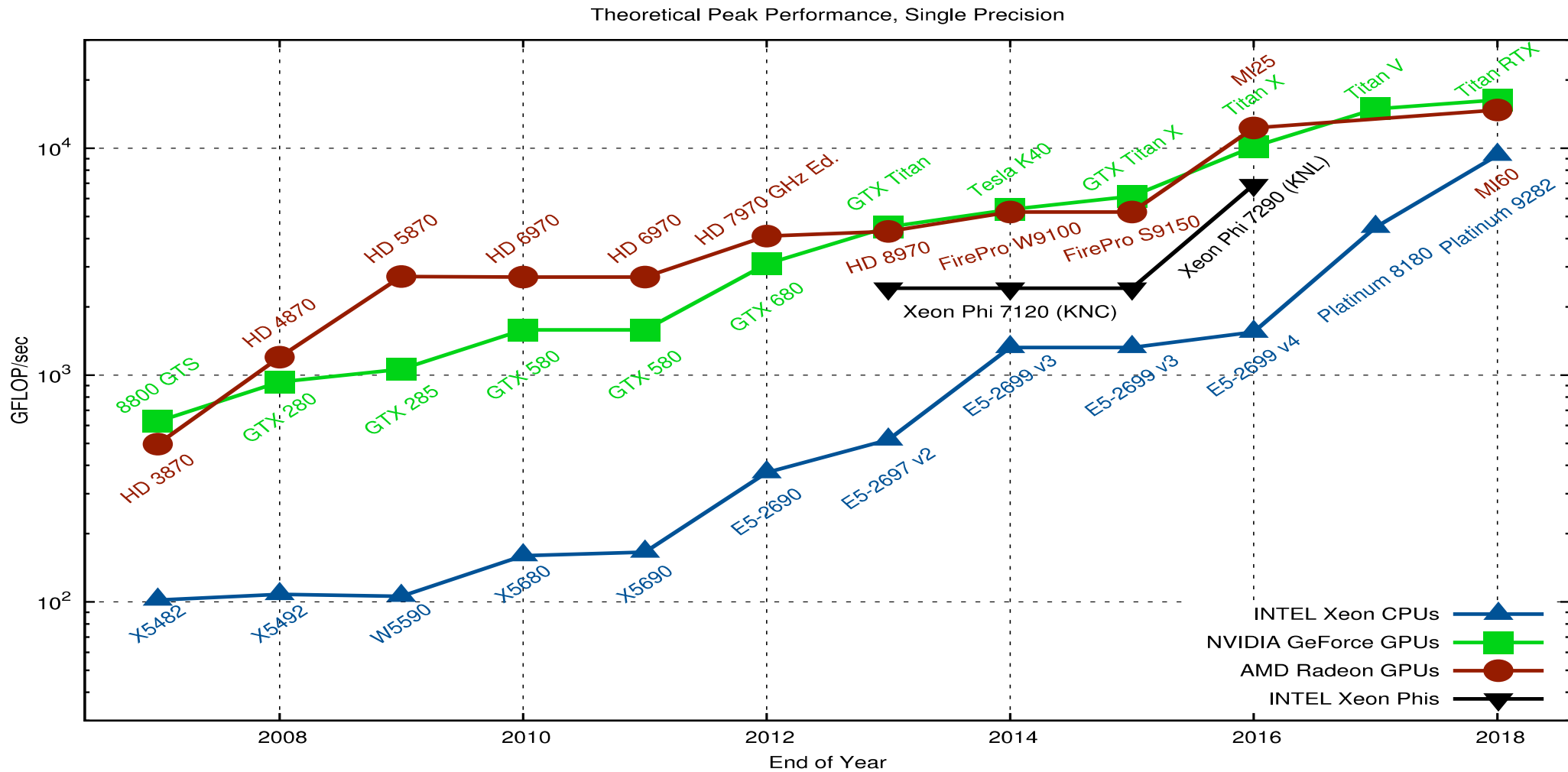
Programmer's intervention!

Theoretical peak performance

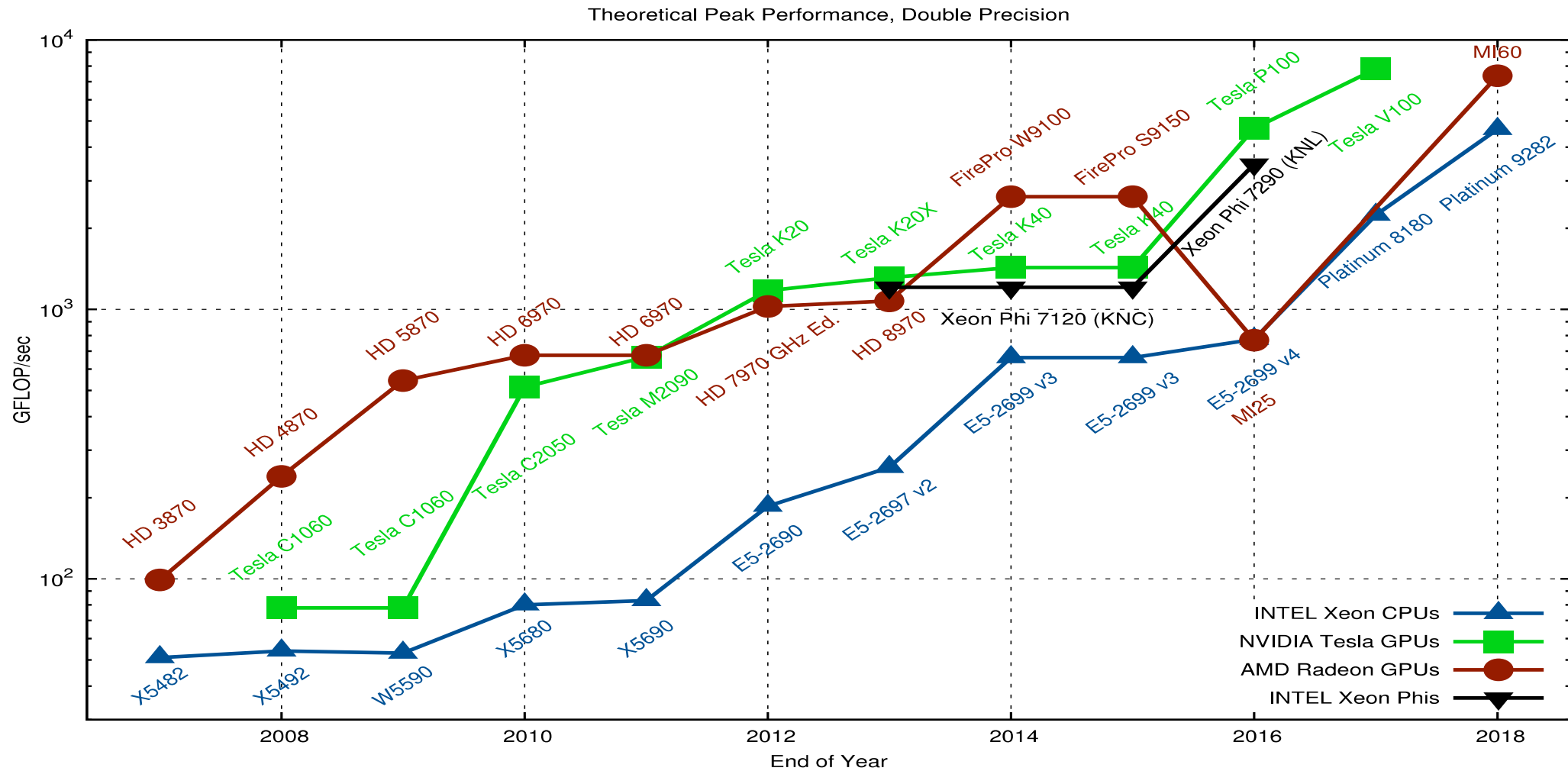
$$\text{Throughput [GFLOP/s]} = \text{chips} * \text{cores} * \text{vectorWidth} * \text{FLOPs/cycle} * \text{clockFrequency}$$

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

multi vs *many* cores (SP-FLOPs)

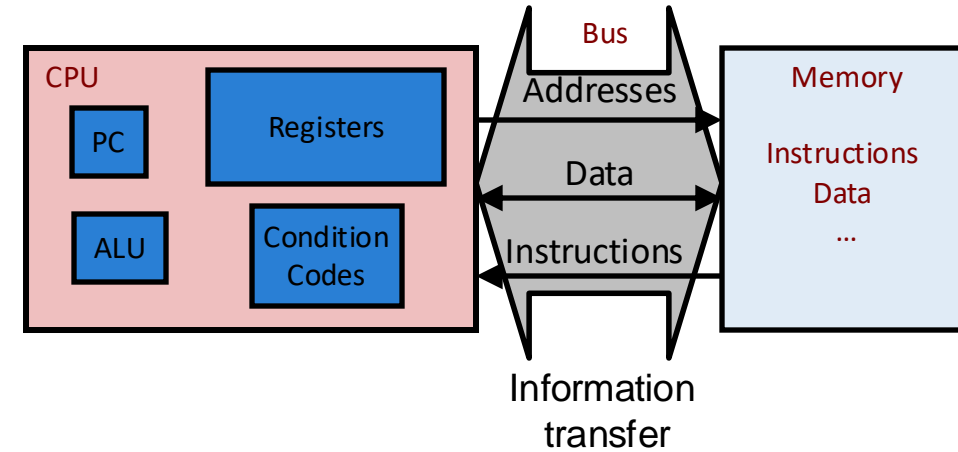


multi vs *many* cores (DP-FLOPs)



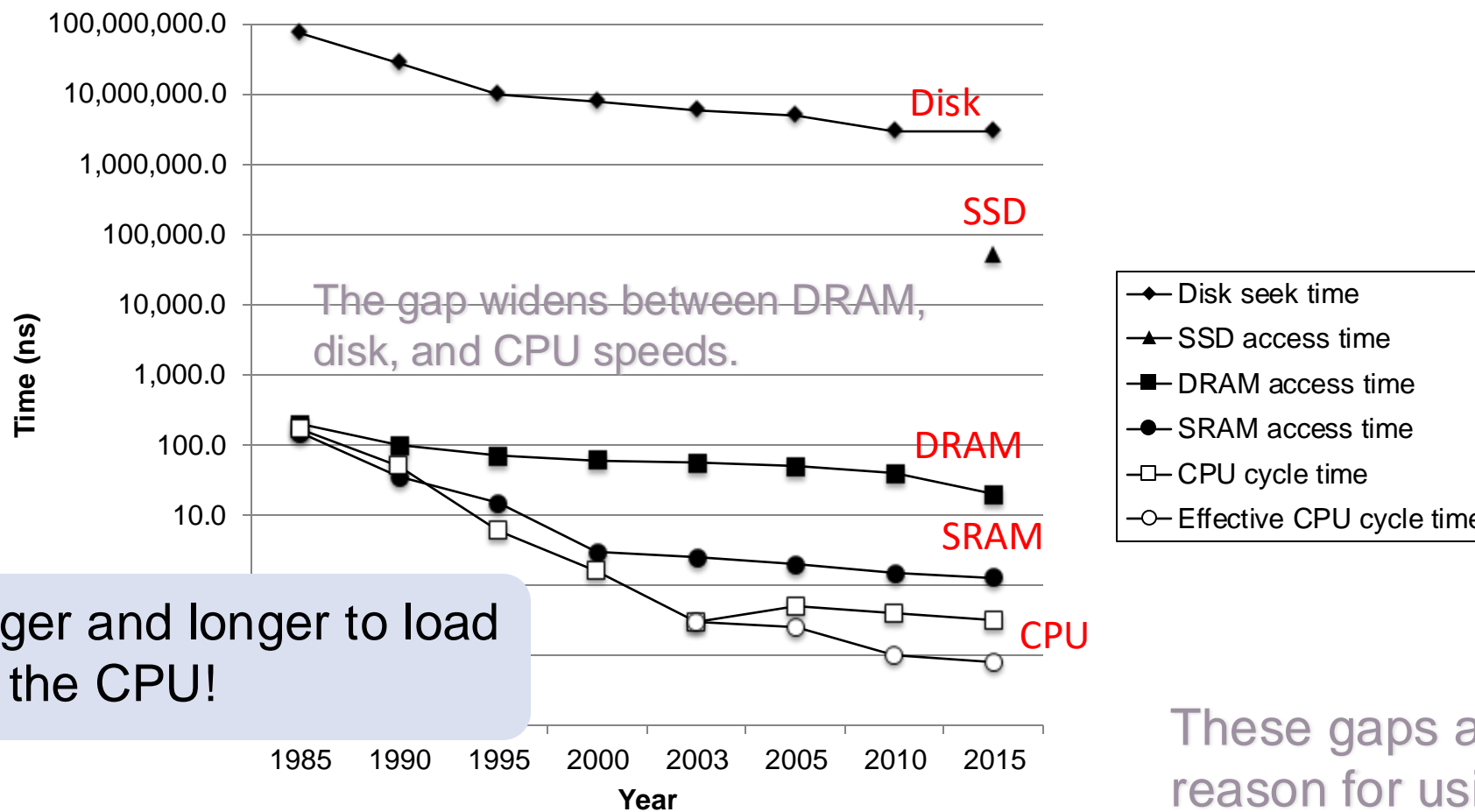
The memory

- Typically organized as linear spaces
 - Some word-size granularity
- Code and data are stored in memory
- Everything that lives in memory has an “address”
 - Visible at assembly level
 - Accessible via pointers/variable names/... from the program itself
- Memory operations are slow!
 - Off-chip
 - Request read/write
 - Search and find



Lots of memory traffic => CPUs idle
=> waste!

The CPU-Memories Gap

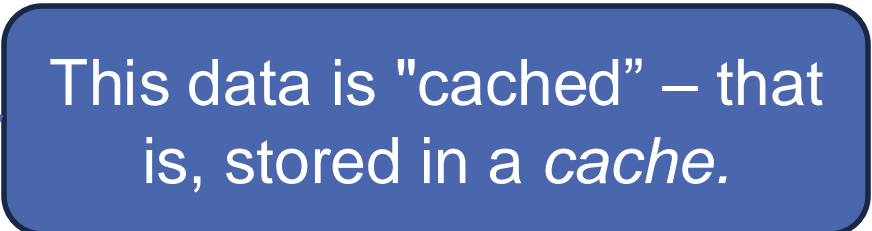


Data takes longer and longer to load to the CPU!

These gaps are the main reason for using a memory hierarchy.

Memory hierarchy

- A single memory for the entire system is not efficient!
- Several memory spaces
 - Large size, low cost, high latency – main memory
 - Small size, high cost, low latency – caches / registers
- Main idea: Bring **some of the data closer to the processor**
 - Smaller latency => faster access
 - Smaller capacity => not all data fits!
- Who can benefit?
 - Applications with **locality** in their data accesses
 - Spatial locality
 - Temporal locality

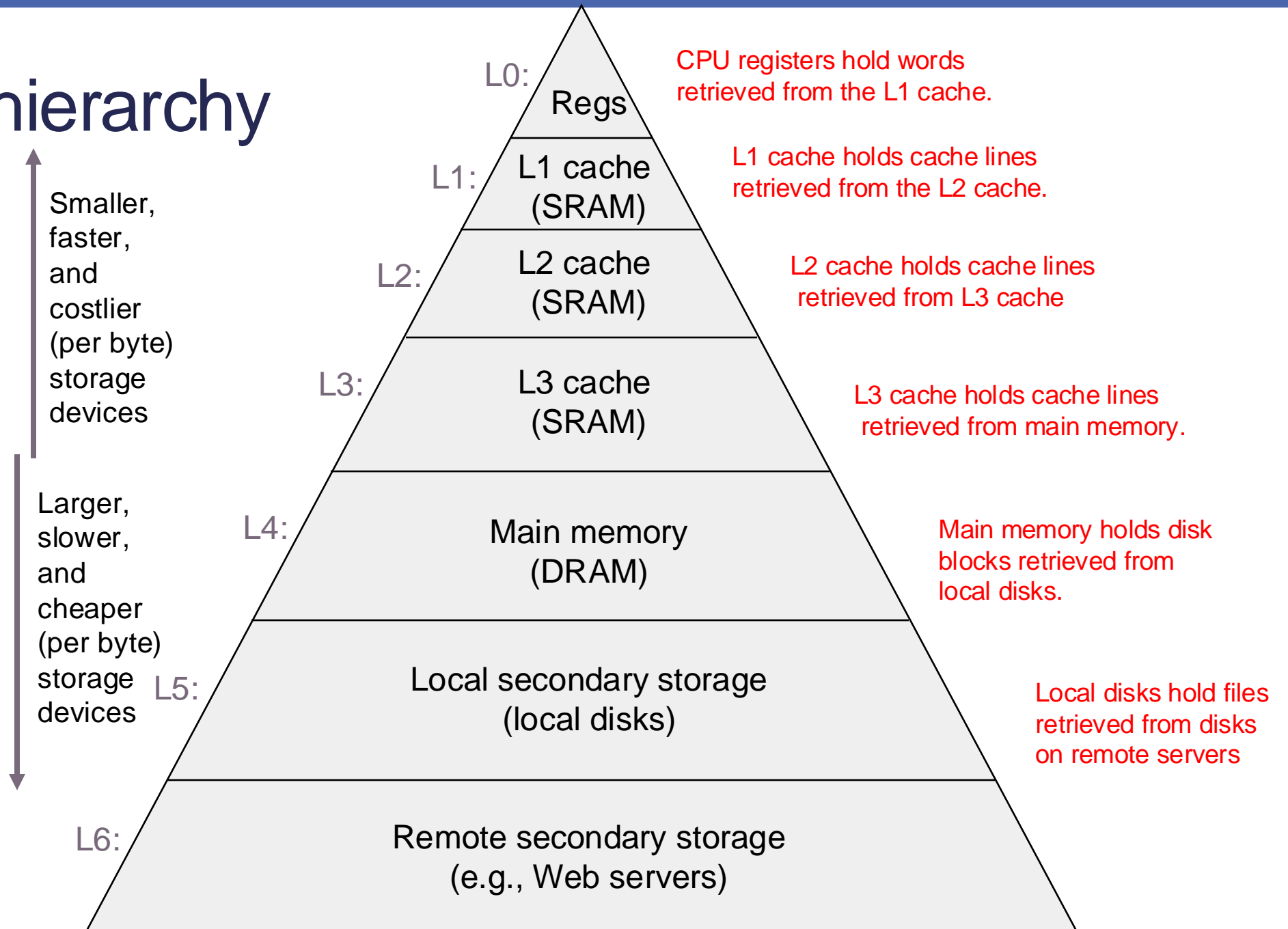


This data is "cached" – that is, stored in a *cache*.

Memory hierarchy and caches

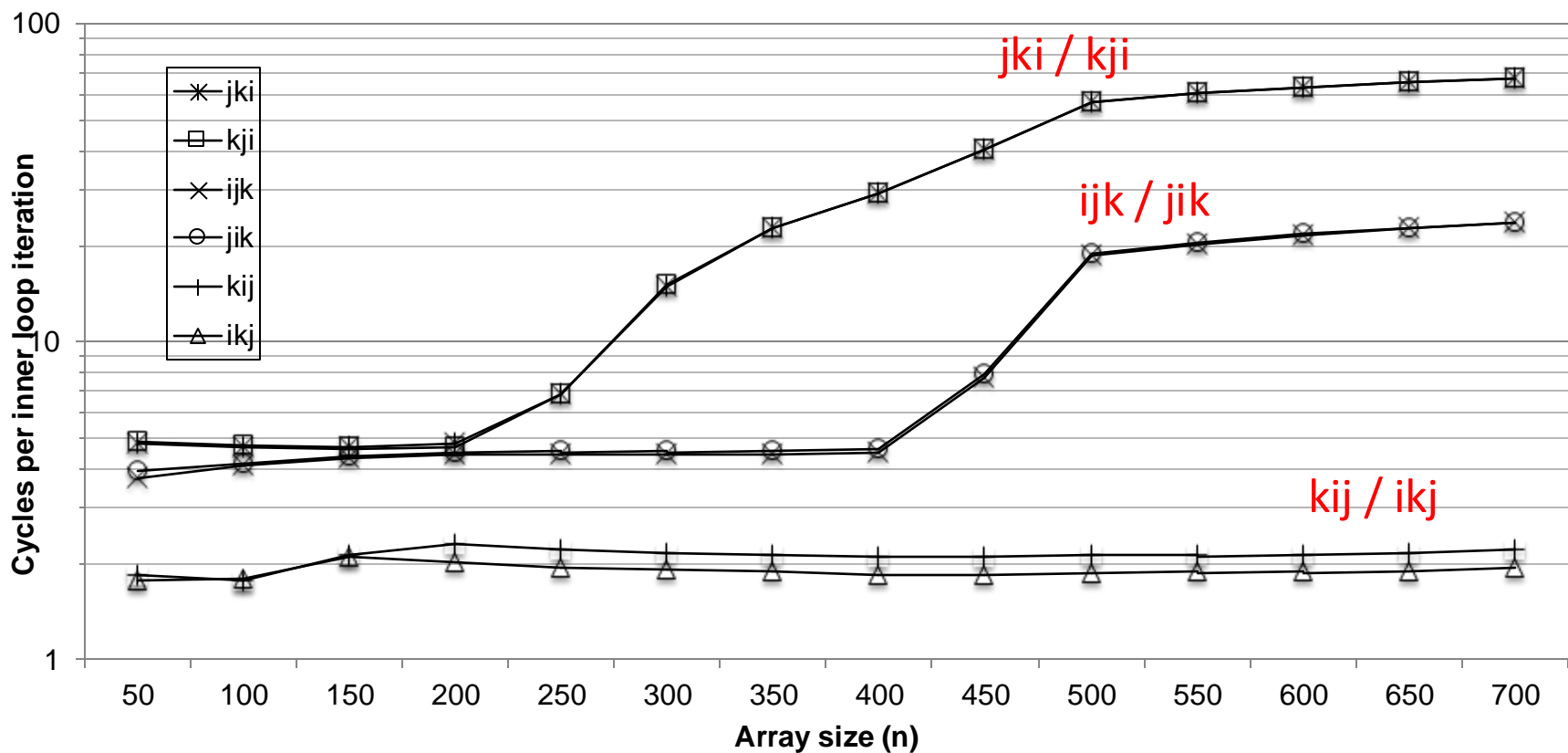
- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Memory hierarchy
 - Multiple layers of memory, from **small & fast** (lower levels) to **large & slow** (higher levels)
 - *For each k , the faster, smaller device at level k is a **cache** for the larger, slower device at level $k+1$.*
- How/why do memory hierarchies work?
 - **Locality** => data at level k is used more often than data at level $k+1$.
 - Level $k+1$ can be slower, and thus larger and cheaper.

Memory hierarchy



Matrix Multiplication

Good vs bad locality / caching ...



```
for (i=0; i<n; i++) {  
  for (j=0; j<n; j++) {  
    sum = 0.0;  
    for (k=0; k<n; k++)  
      sum += a[i][k] * b[k][j];  
    c[i][j] = sum;  
  }  
}
```

ijk / jik

```
for (k=0; k<n; k++) {  
  for (i=0; i<n; i++) {  
    r = a[i][k];  
    for (j=0; j<n; j++)  
      c[i][j] += r * b[k][j];  
  }  
}
```

kij / ikj

```
for (j=0; j<n; j++) {  
  for (k=0; k<n; k++) {  
    r = b[k][j];  
    for (i=0; i<n; i++)  
      c[i][j] += a[i][k] * r;  
  }  
}
```

jki / kji

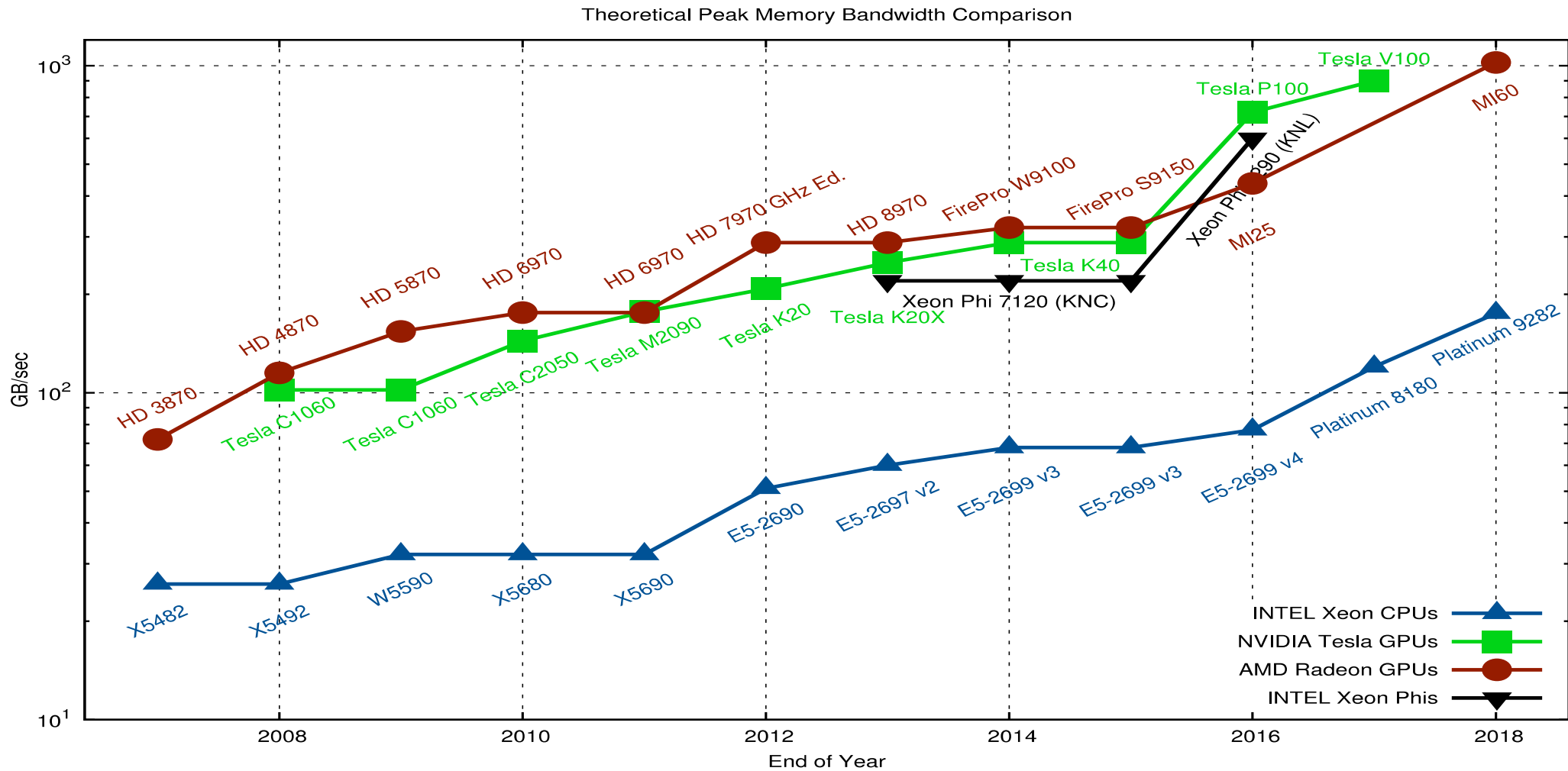
Theoretical peak performance

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 (GB/s)



Balance

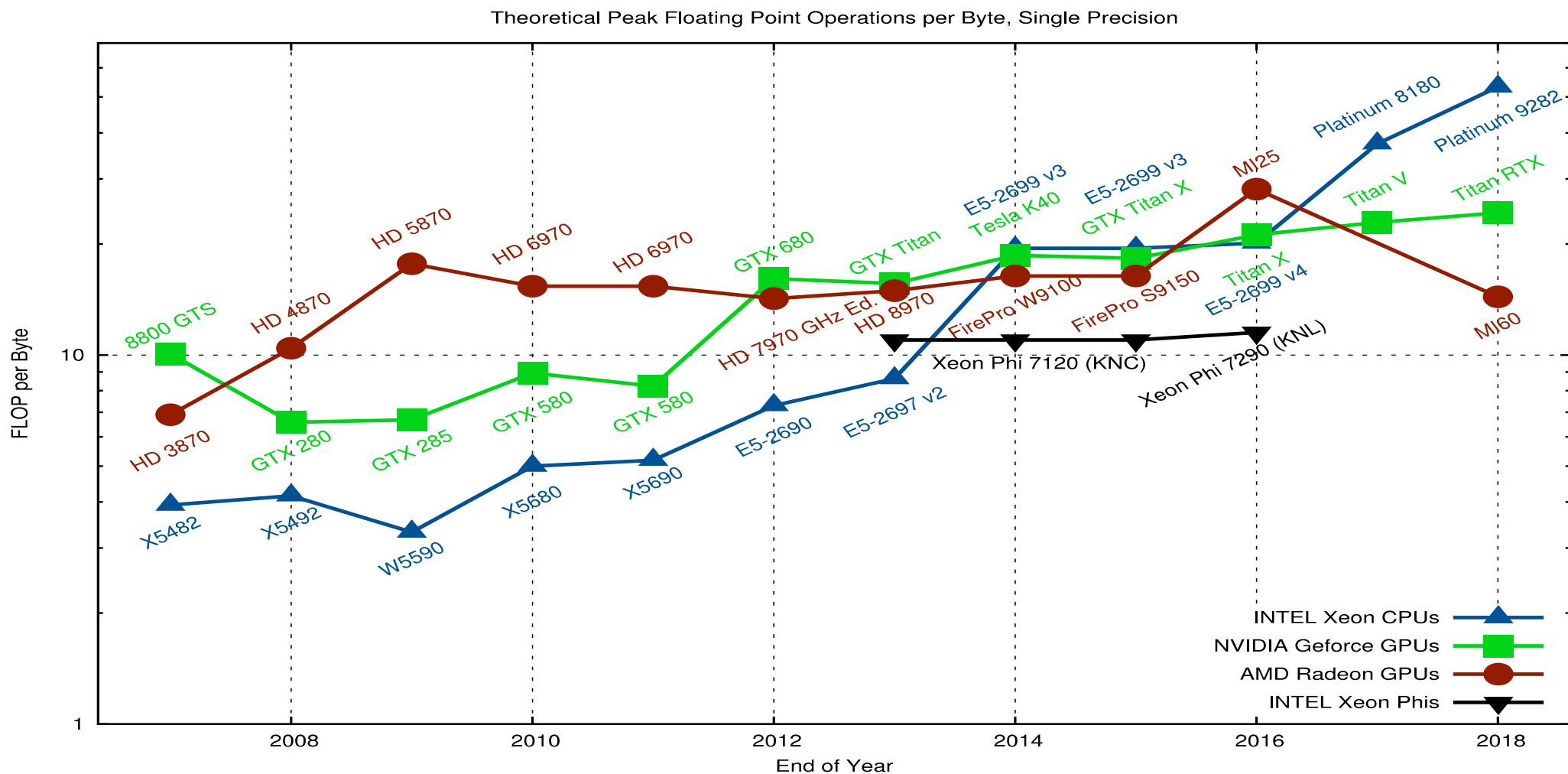
Throughput [GFLOP/s] = chips * cores * vectorWidth *
FLOPs/cycle * clockFrequency

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

	Cores	Threads/ALUs	FLOPS/s	Byte/s	FLOPS/Byte
Intel Core i7	4	16	85	25.6	3.3
AMD Barcelona	4	8	37	21.4	1.7
AMD Istanbul	6	6	62.4	25.6	2.4
NVIDIA GTX 580	16	512	1581	192	8.2
NVIDIA GTX 680	8	1536	3090	192	16.1
AMD HD 6970	384	1536	2703	176	15.4
AMD HD 7970	32	2048	3789	264	14.4
Intel Xeon Phi 7120	61	240	2417	352	6.9

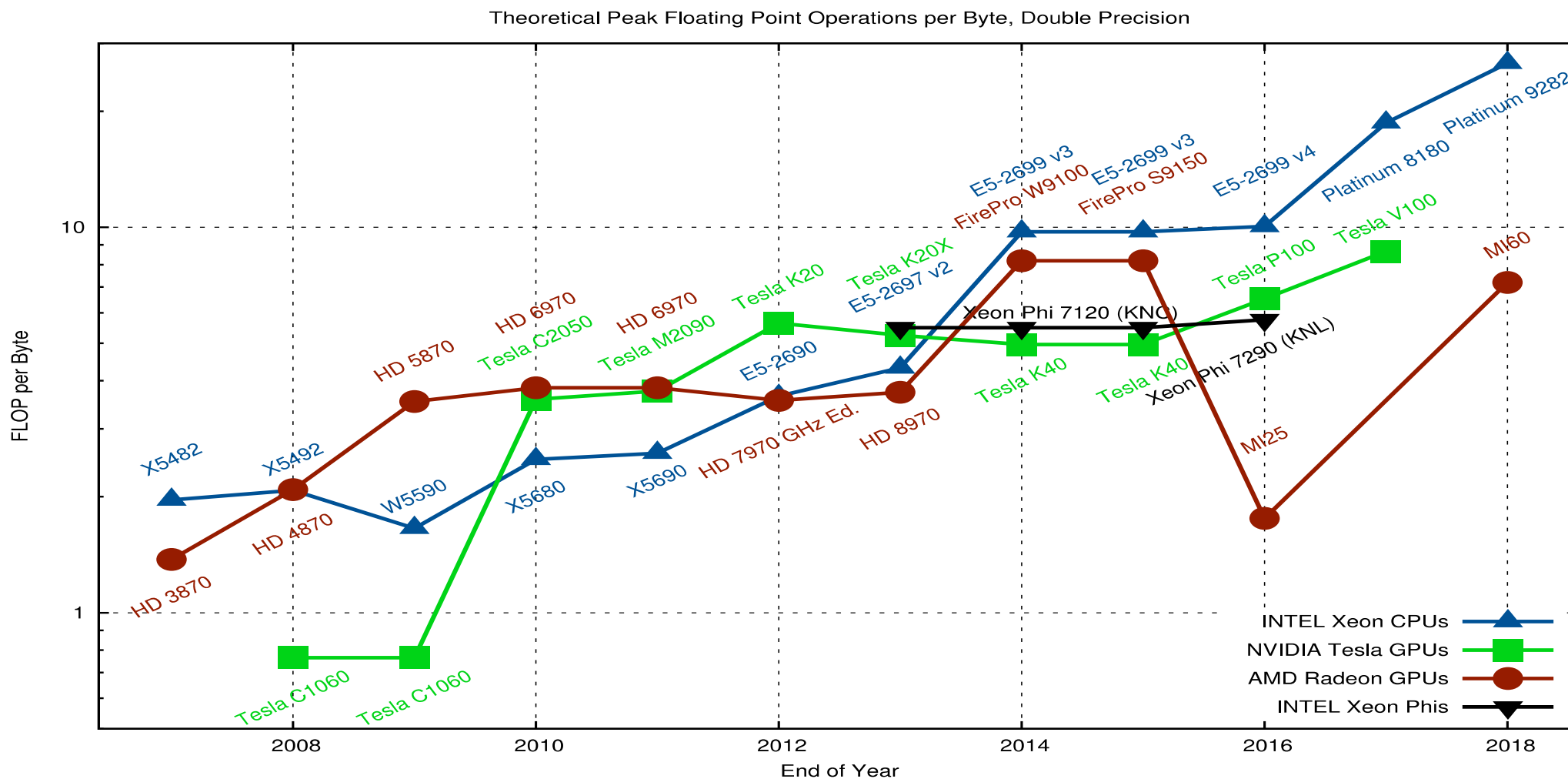
Balance ?

FLOPs/Byte (SP) !



Balance ?

FLOPs/Byte (DP) !



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>

Sustainability TODO's

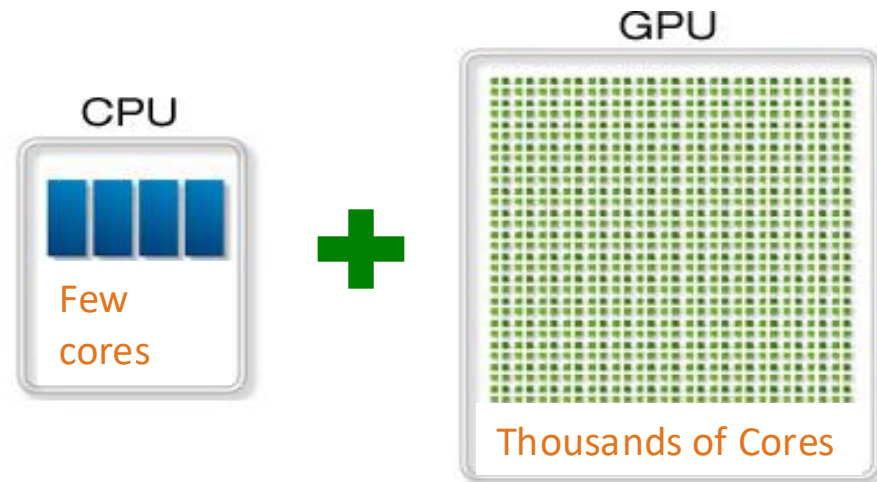
- Use all CPUs capabilities
 - Maximize parallelism
 - Use SIMD & ILP
- Use accelerators
 - Maximize parallelism
 - Correct mapping of application to GPU
 - Heterogeneous computing
- Use all memory capabilities
 - Maximize bandwidth
 - Use caching / improve locality

Limit the impact of low utilization
Reduce impact of unused cores/chips
Reduce impact of unused bandwidth**

** non trivial ...

Heterogeneous computing?

- A heterogeneous platform = a CPU + a GPU (the starting point)
- An application workload = an application + its input dataset
- Workload partitioning = workload distribution among the processing units of a heterogeneous system



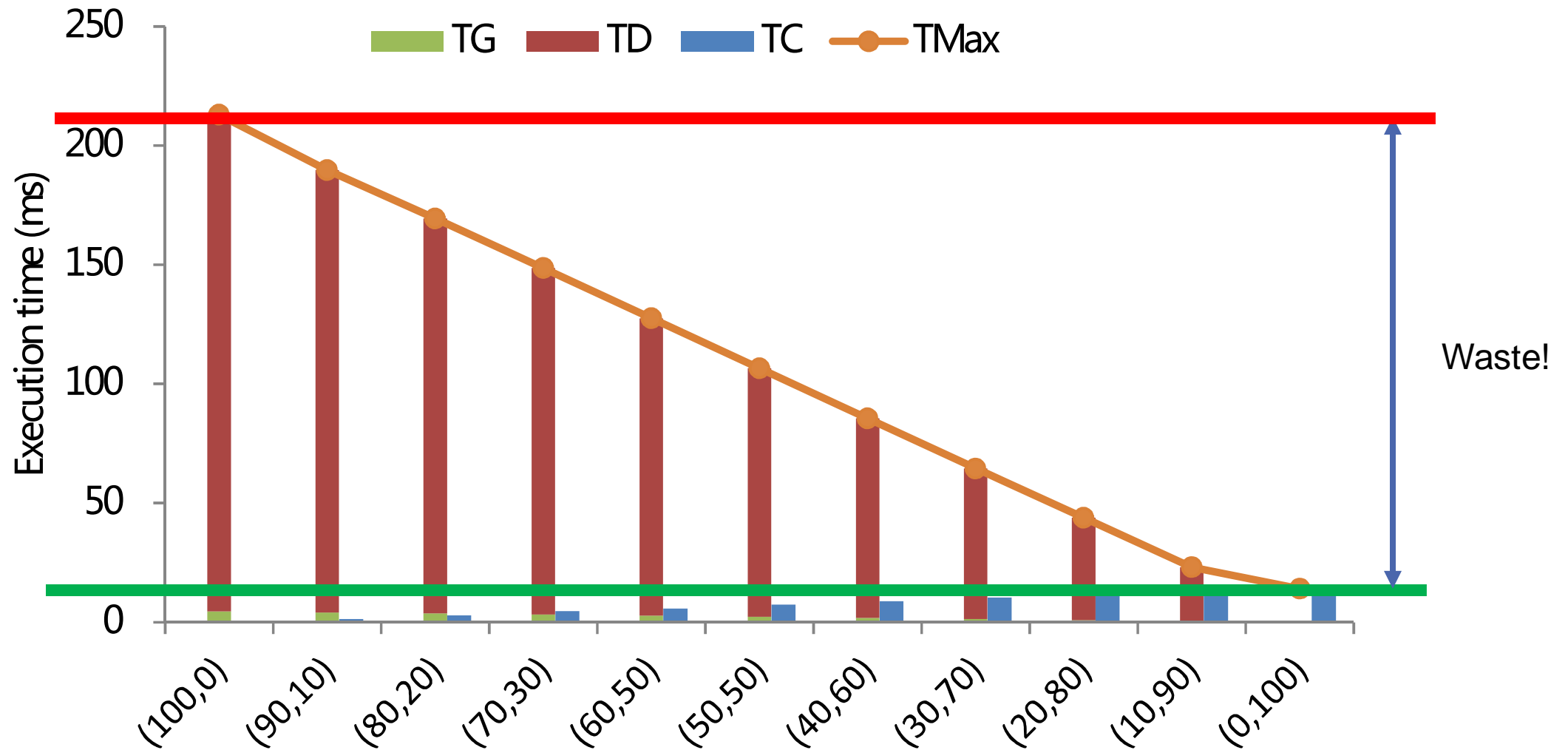
Example 1: dot product

- Dot product
 - Compute the dot product of 2 (1D) arrays
- Performance
 - T_G = execution time on GPU
 - T_C = execution time on CPU
 - T_D = data transfer time CPU-GPU
- GPU best or CPU best?

"Dot Product"

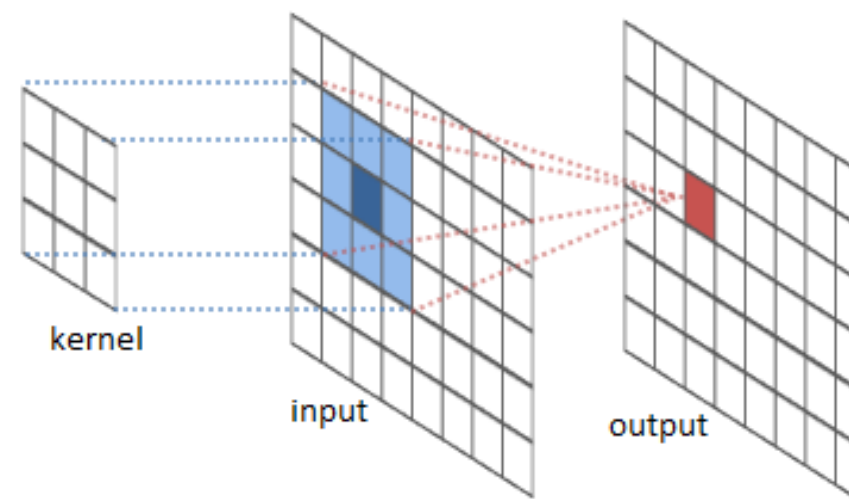
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \end{bmatrix}$$

Example 1: dot product

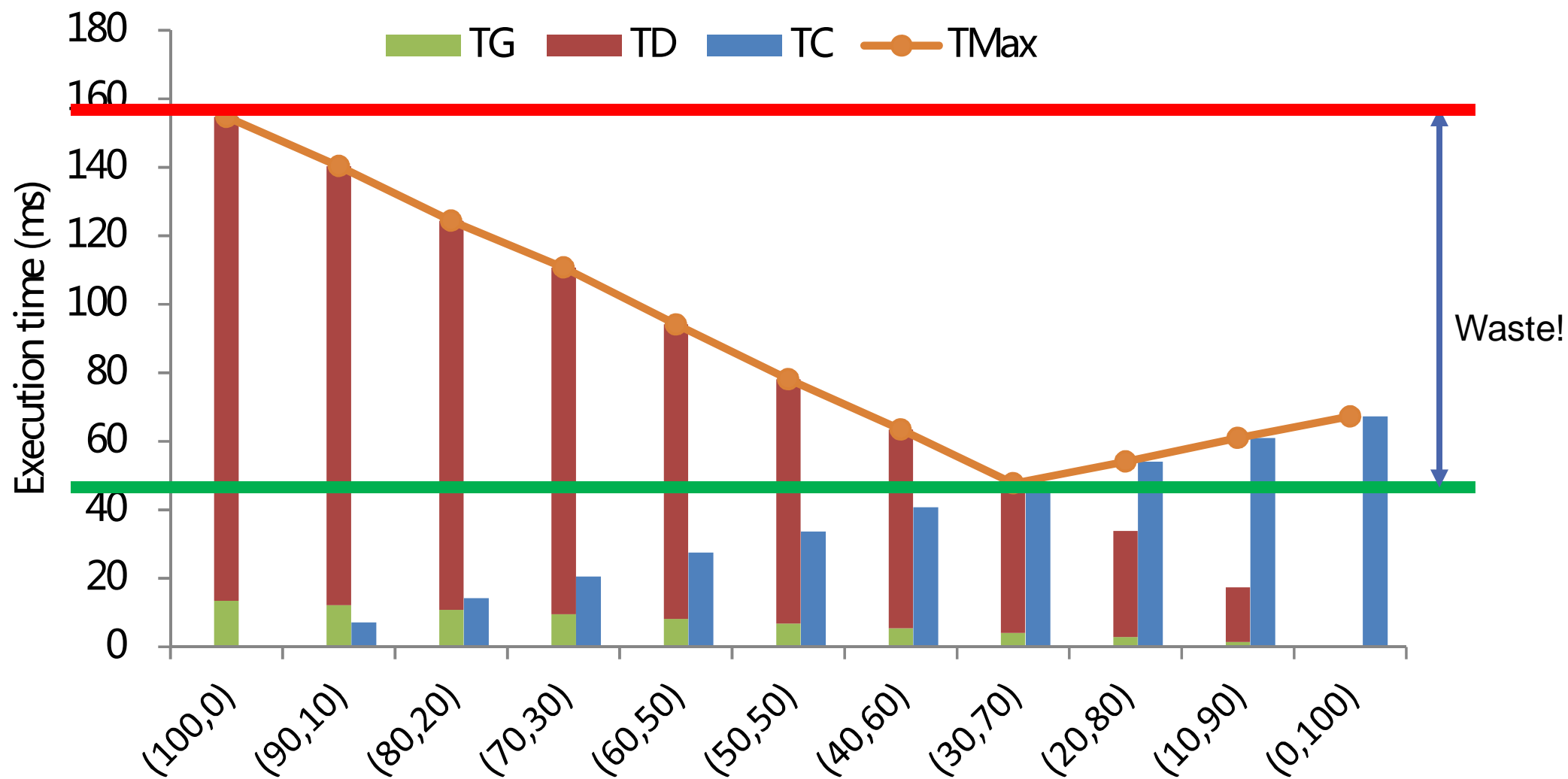


Example 2: separable convolution

- Separable convolution (CUDA SDK)
 - Apply a convolution filter (kernel) on a large image.
 - Separable kernel allows applying
 - Horizontal first
 - Vertical second
- Performance
 - T_G = execution time on GPU
 - T_C = execution time on CPU
 - T_D = data transfer time
- GPU best or CPU best?

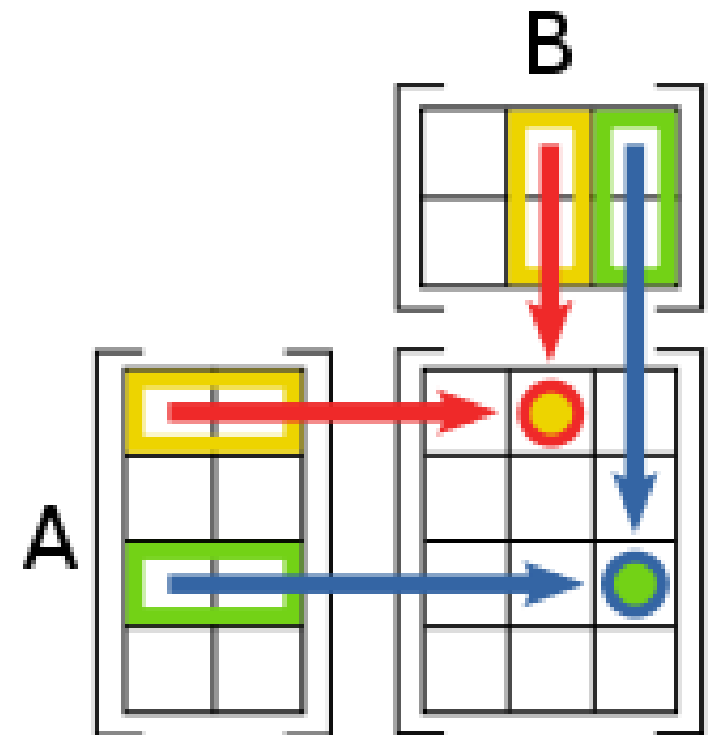


Example 2: separable convolution

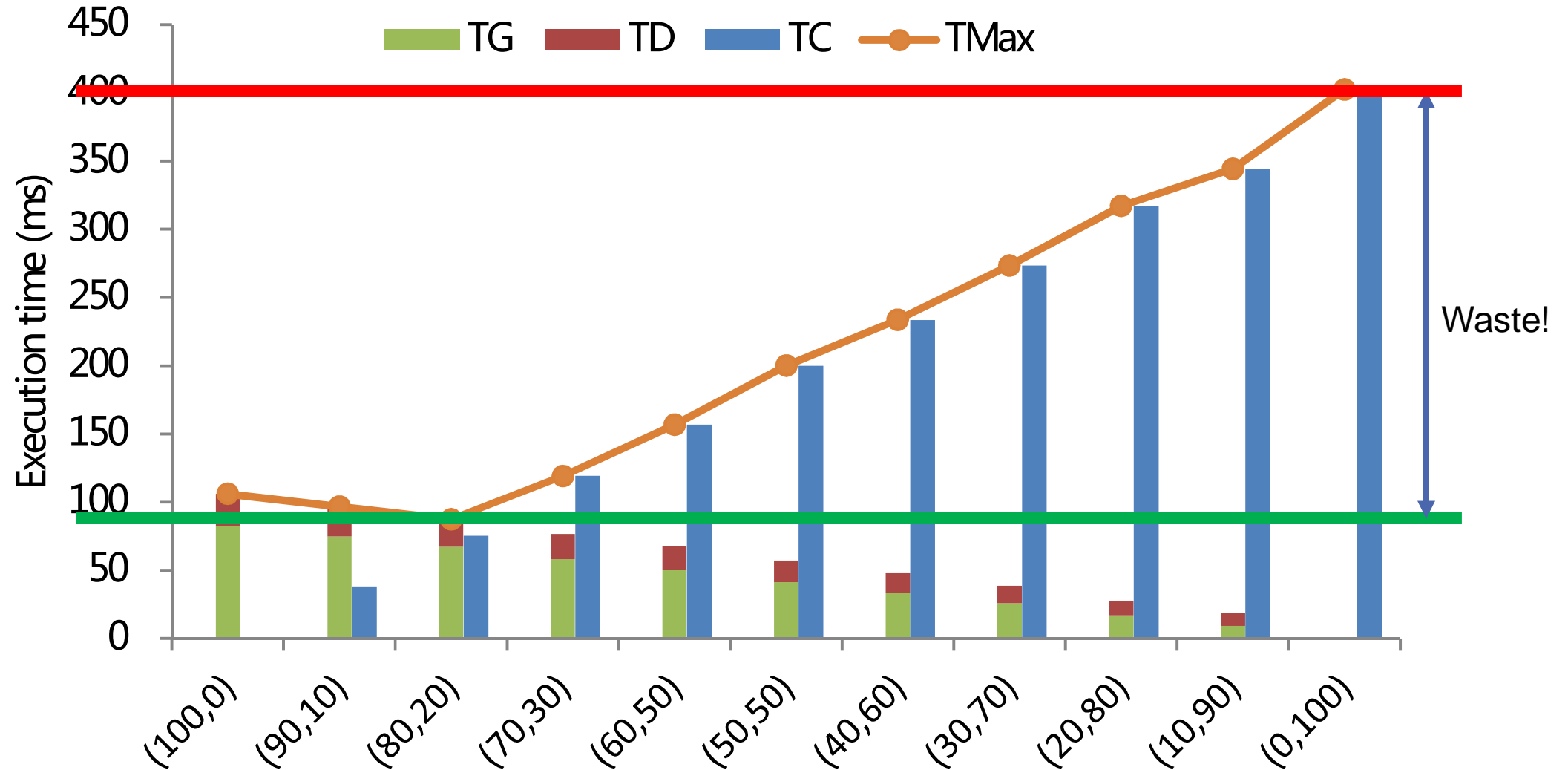


Example 3: matrix multiply

- Matrix multiply
 - Compute the product of 2 matrices
- Performance
 - T_G = execution time on GPU
 - T_C = execution time on CPU
 - T_D = data transfer time CPU-GPU
- GPU best or CPU best?

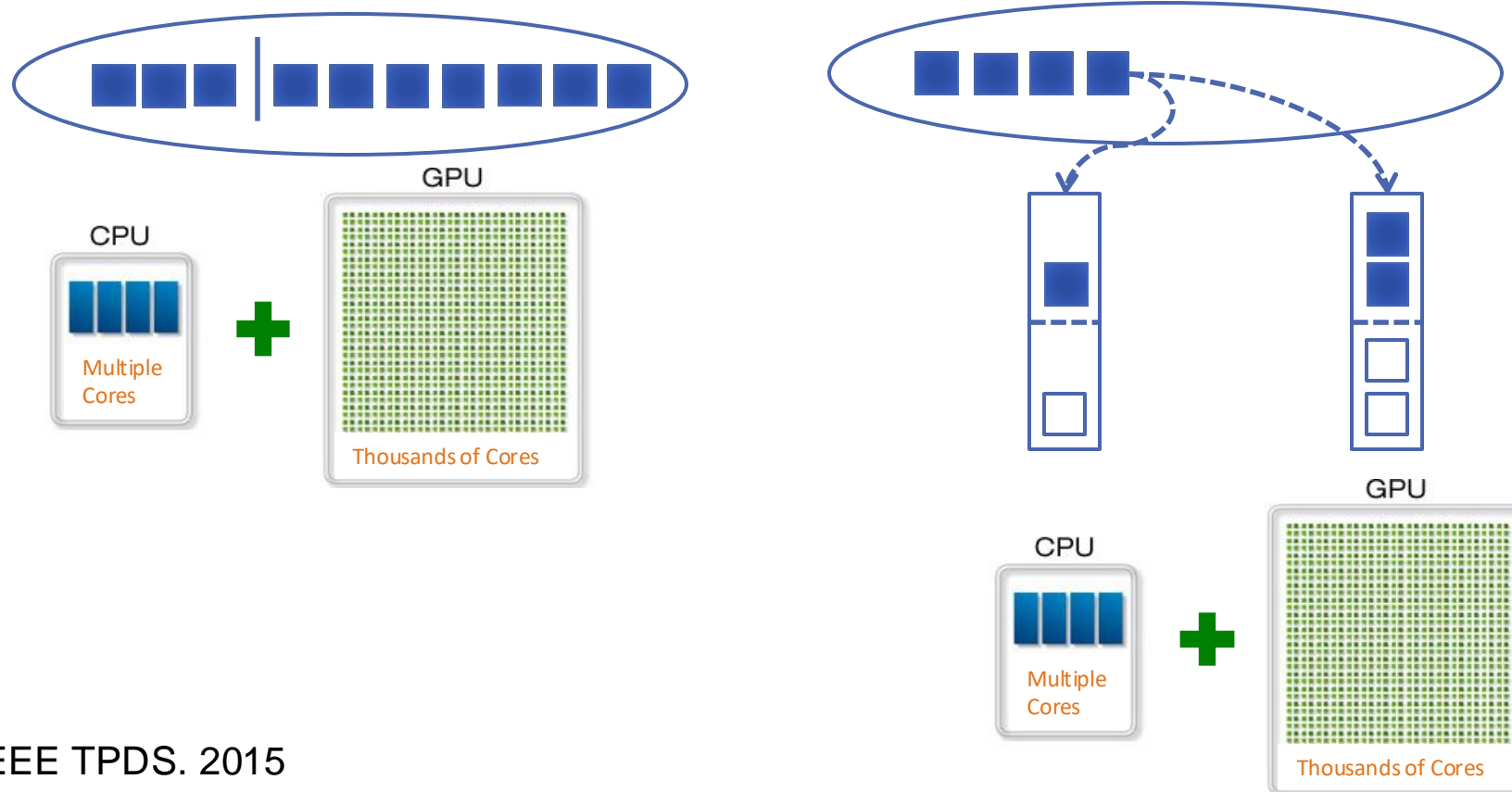


Example 3: matrix multiply



Determining the partition

- Static partitioning (SP) vs. Dynamic partitioning (DP)



In summary ...

- Understand the system you use ...
 - And match applications to hardware
- Increase utilization
 - Re-design application/algorithm
- Decrease impact of low utilization
 - Share resources
 - DVFS

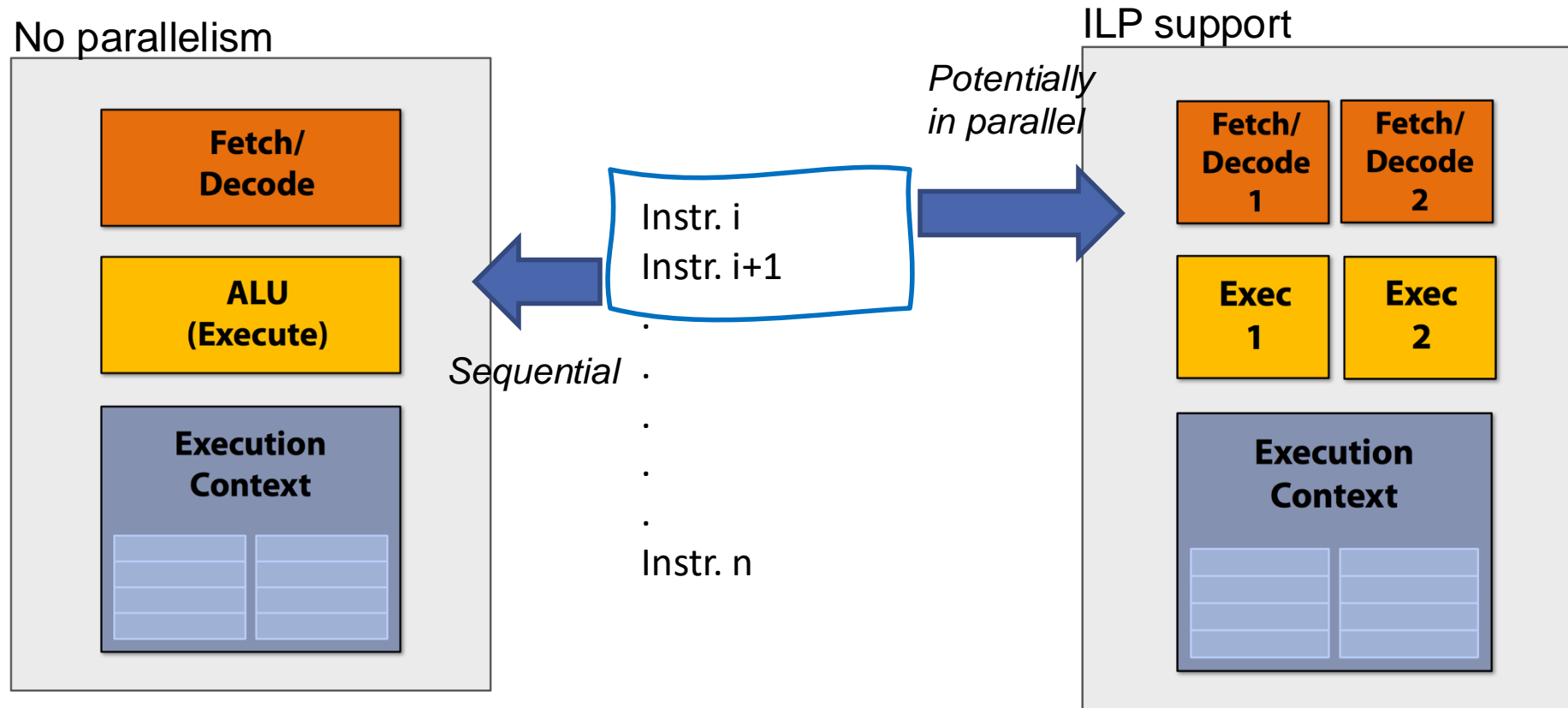
Next time: PART 2 – applications & tools

- Understand applications
- Tools to quantify and improve sustainability

Backup slides

(1) ILP (Instruction level parallelism)

- Multiple instructions issued & executed in the same cycle



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

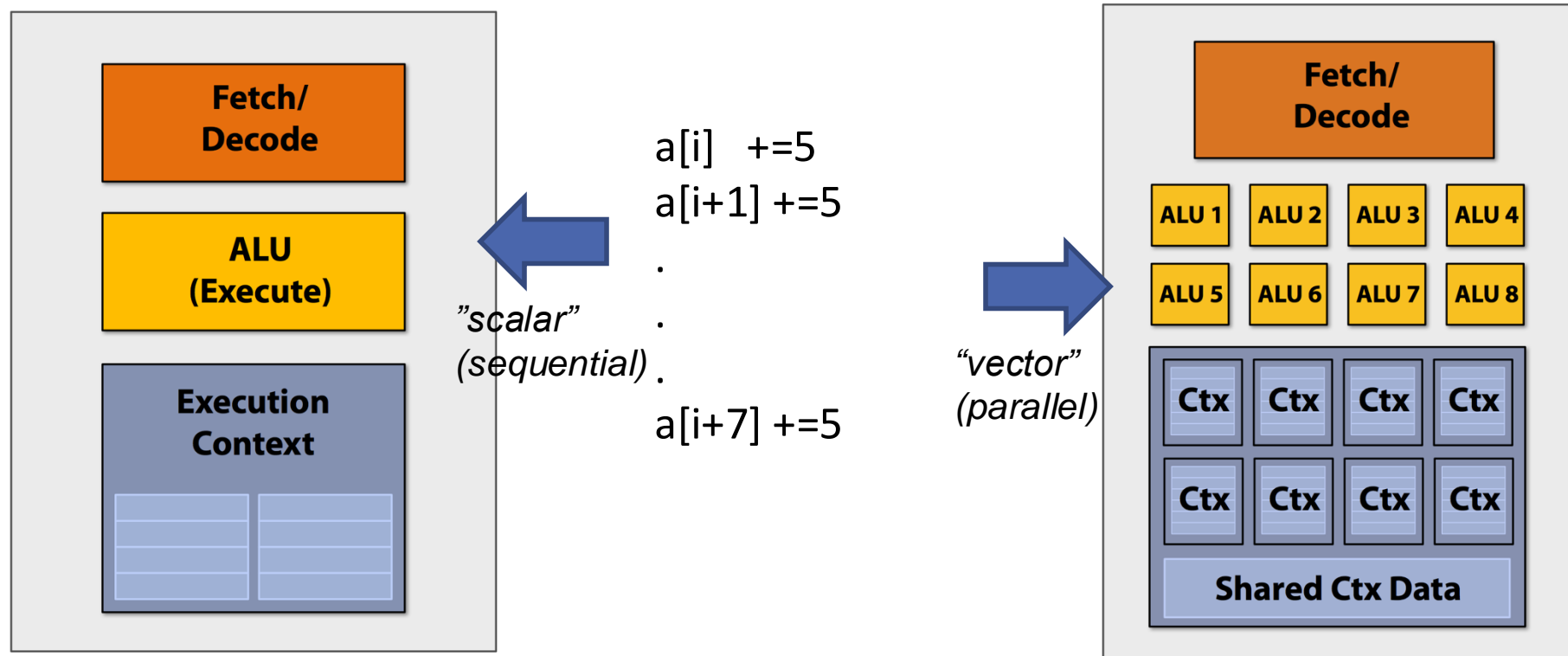
<http://15418.courses.cs.cmu.edu/spring2016/lectures>

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

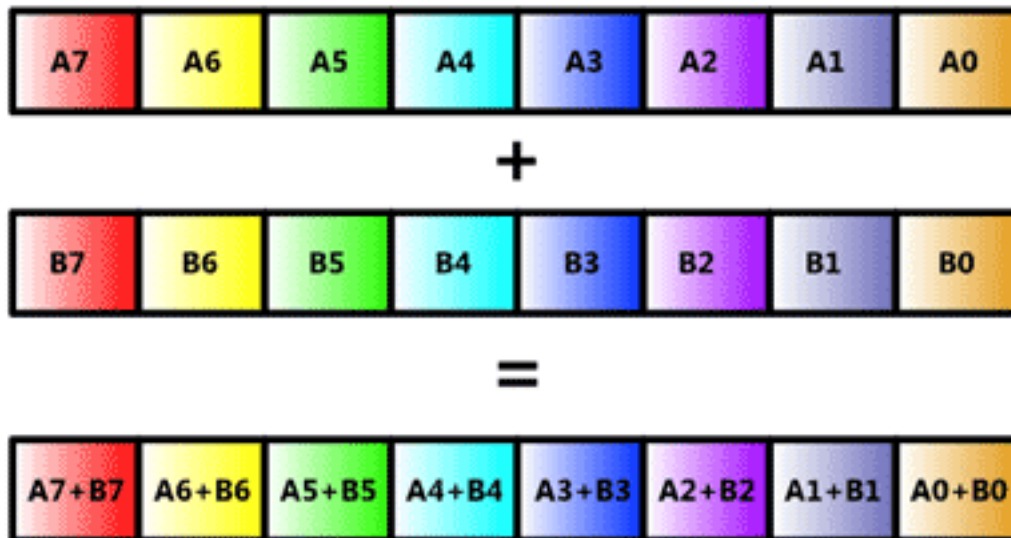
(2) SIMD (single instruction, multiple data)

- Same instruction executed on multiple data items

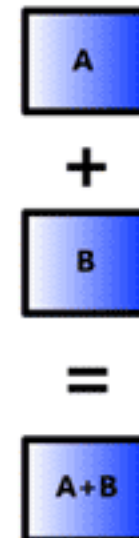


Scalar vs SIMD operations

SIMD Mode



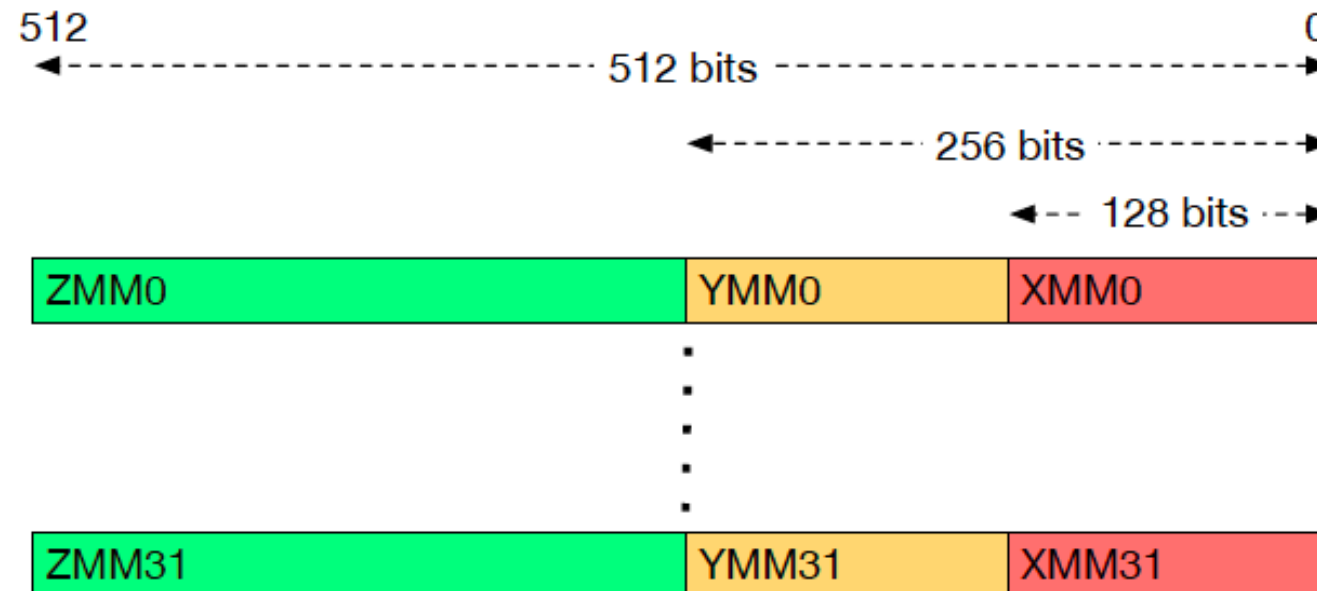
Scalar Mode



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)



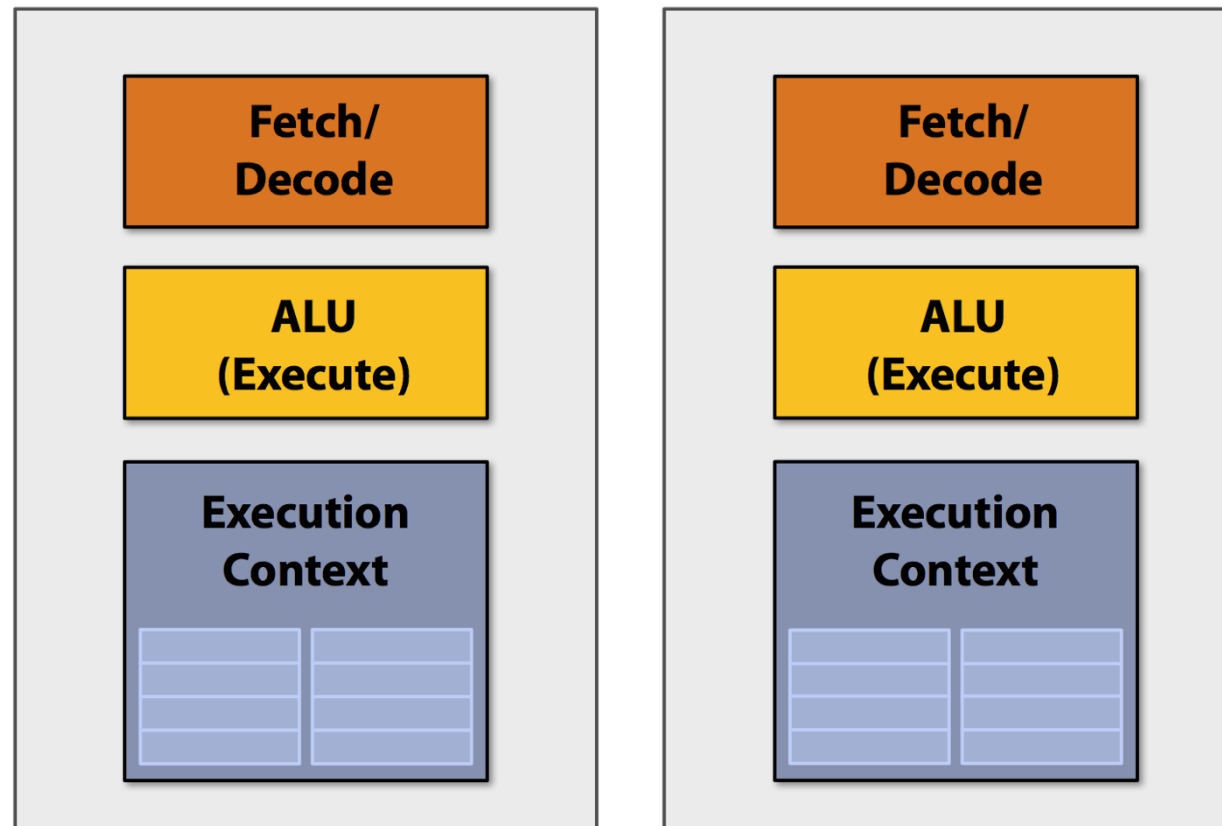
SIMD programmer intervention

- 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

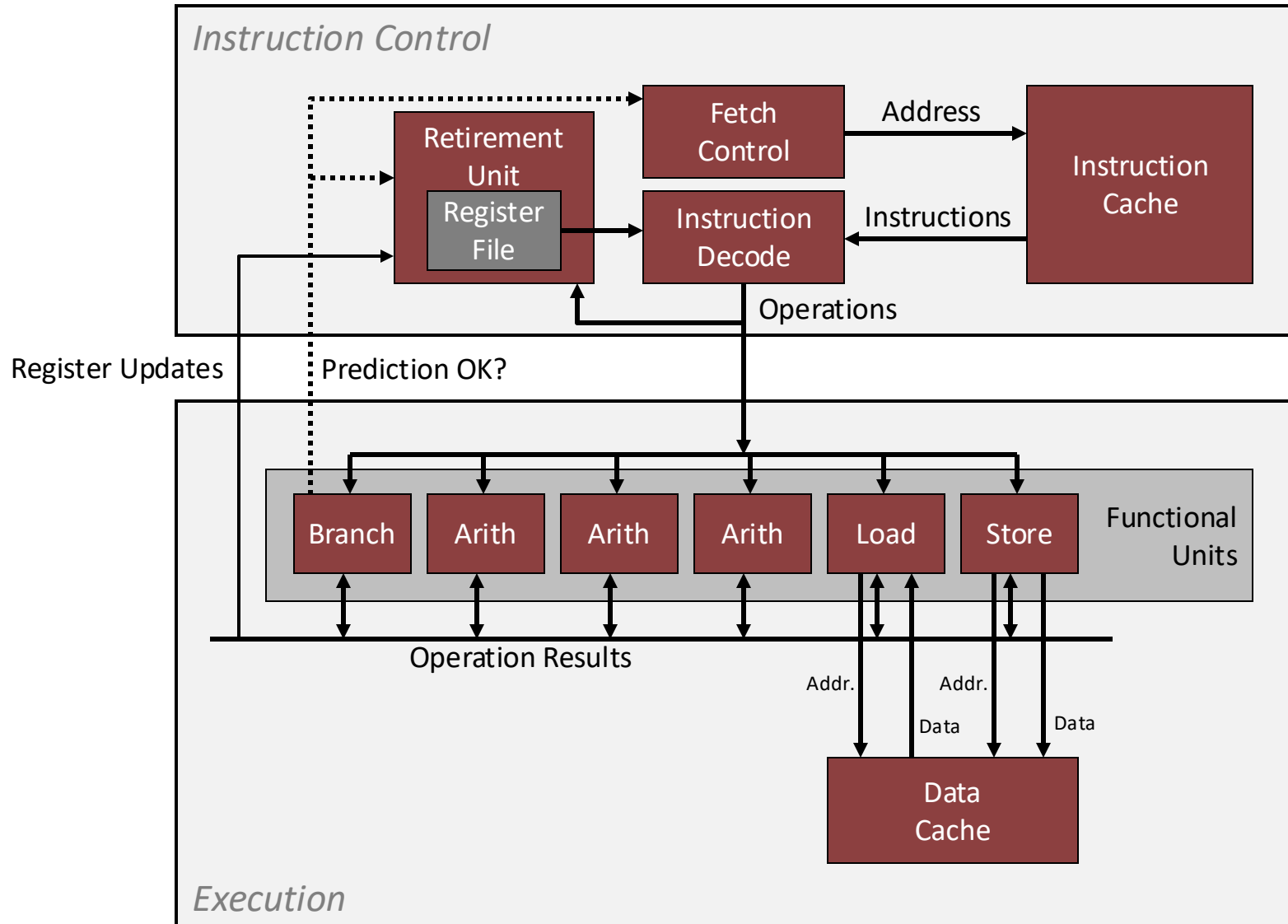
CPU features for ILP

- 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

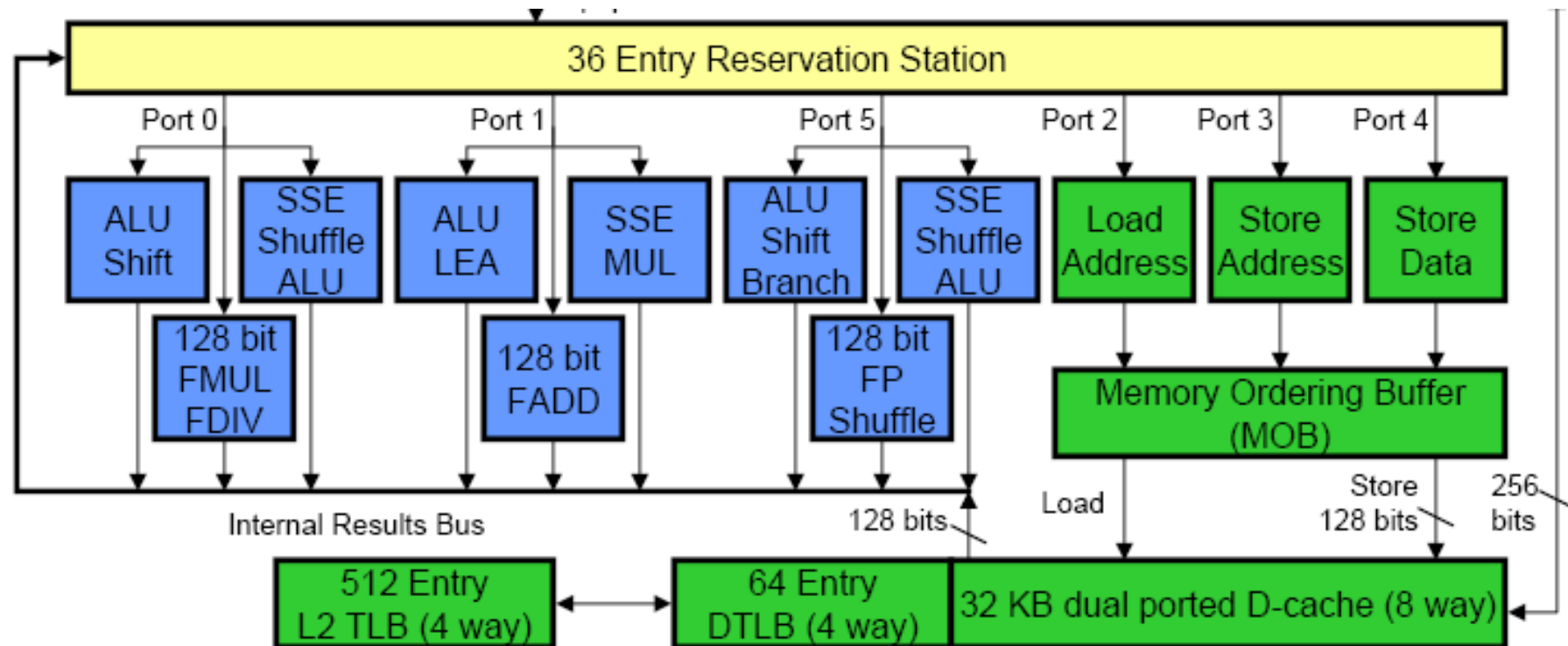
Superscalar, Out-of-order

- 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 ...

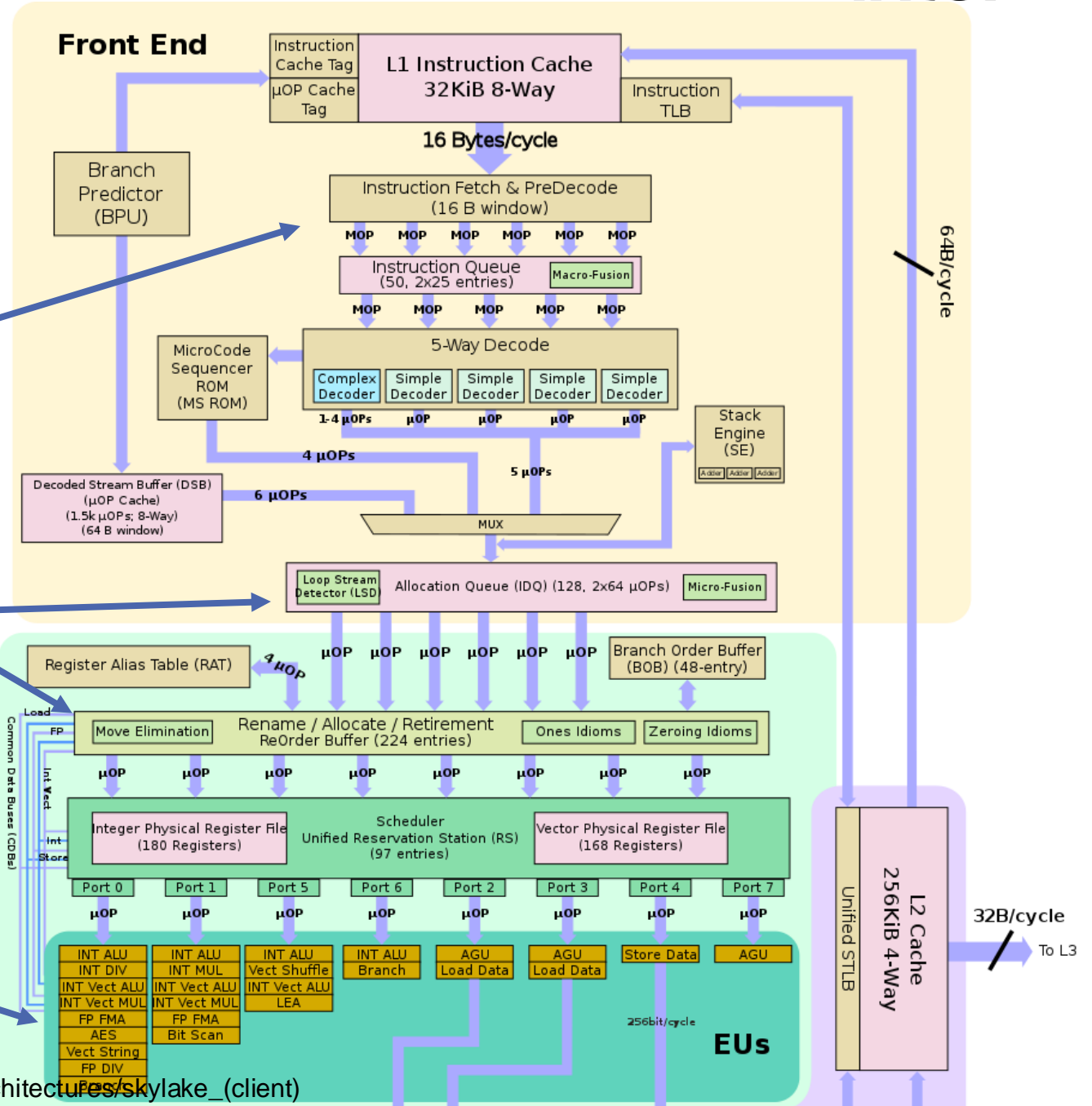


SkyLake®

Fetch & decode, producing multiple uOps

Optimize, reorder, schedule uOps

Multiple execution units, some SIMD





BONUS!

Hardware multi-threading (or hyperthreading®)

”Are there hardware threads?!”

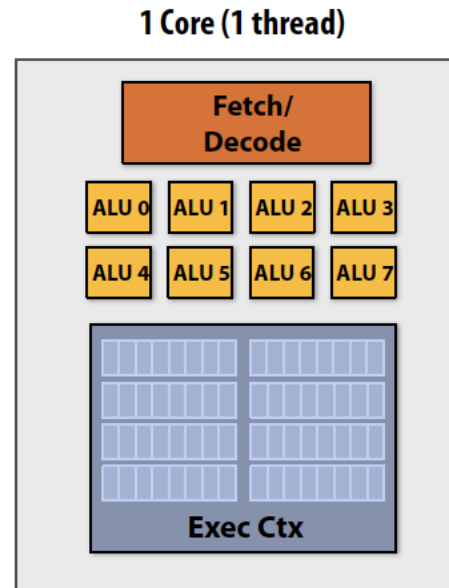
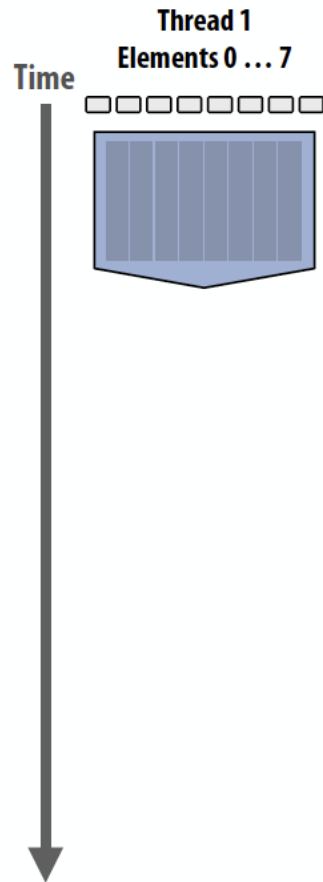
- Hardware (supported) multi-threading
 - Core manages thread context
 - Interleaved (temporal multi-threading) – employed in GPUs
 - Simultaneous (co-located execution) – e.g., Intel Hyperthreading



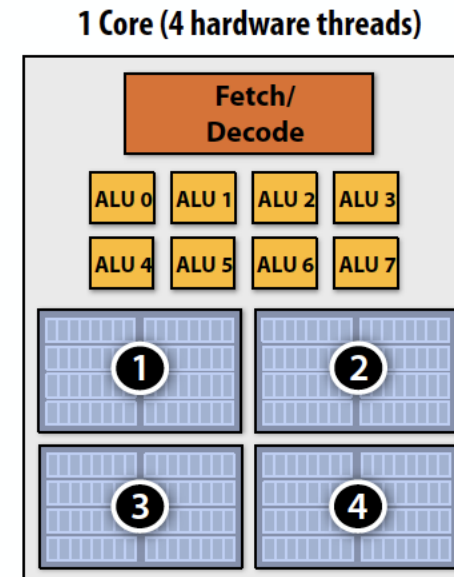
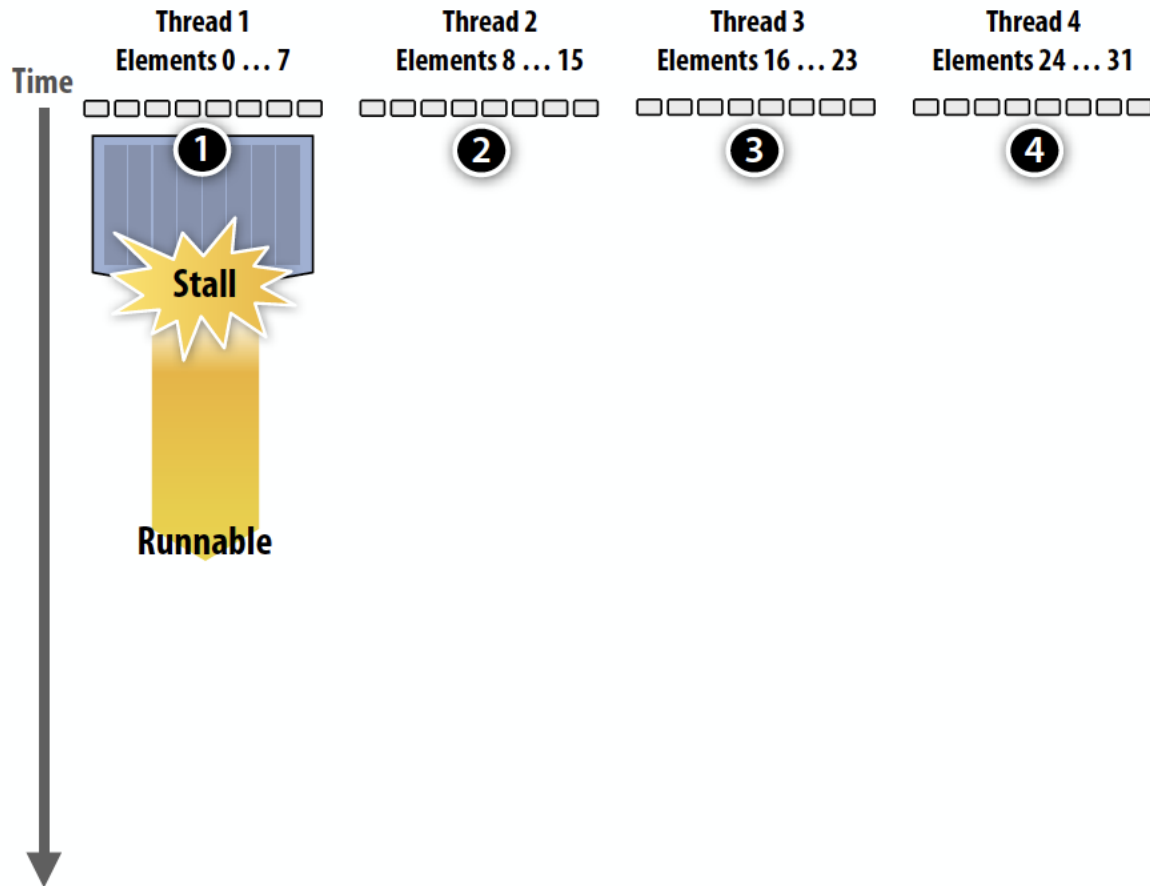
Why bother?

- 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*

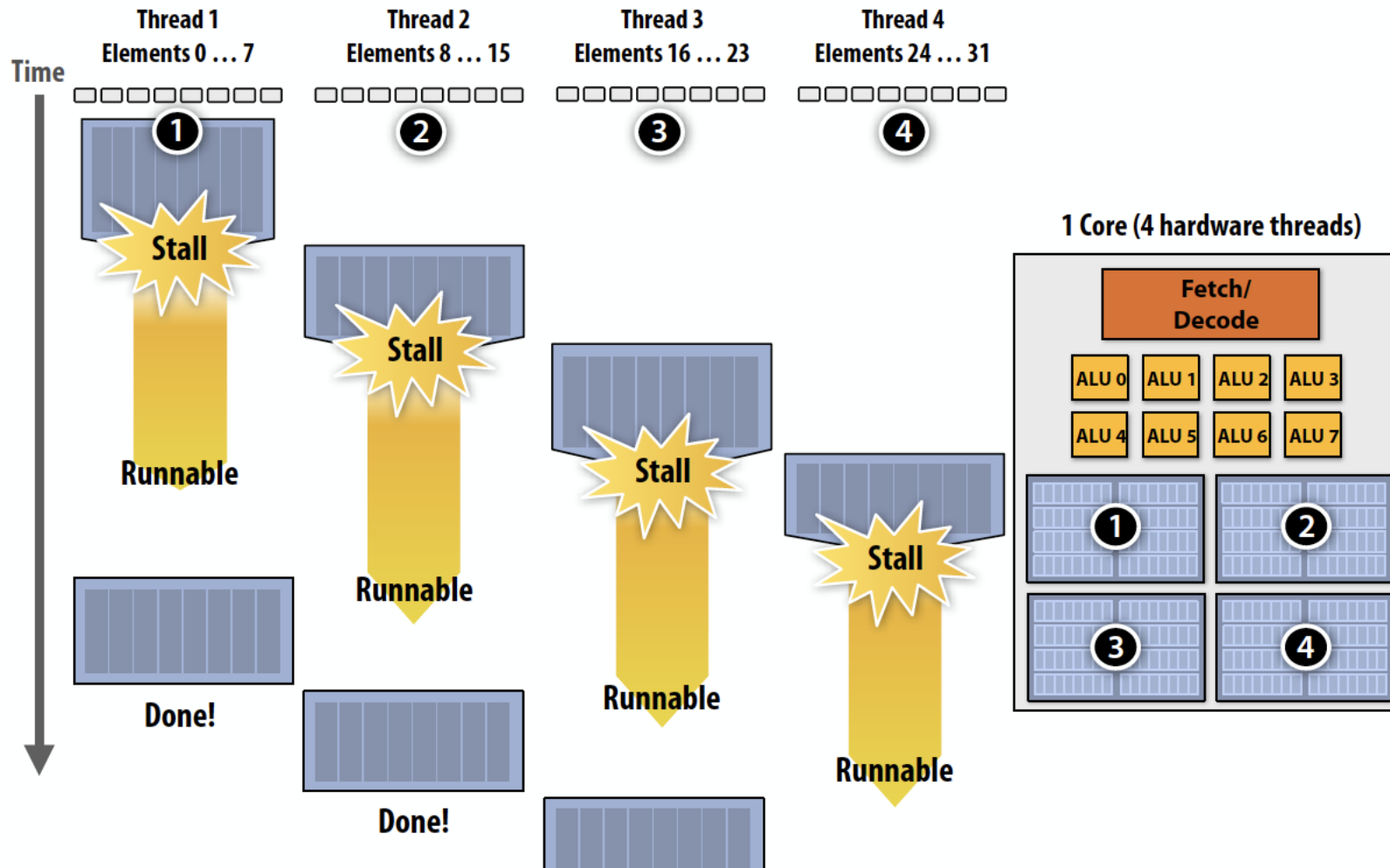
Advantage: increased throughput



Advantage: increased throughput

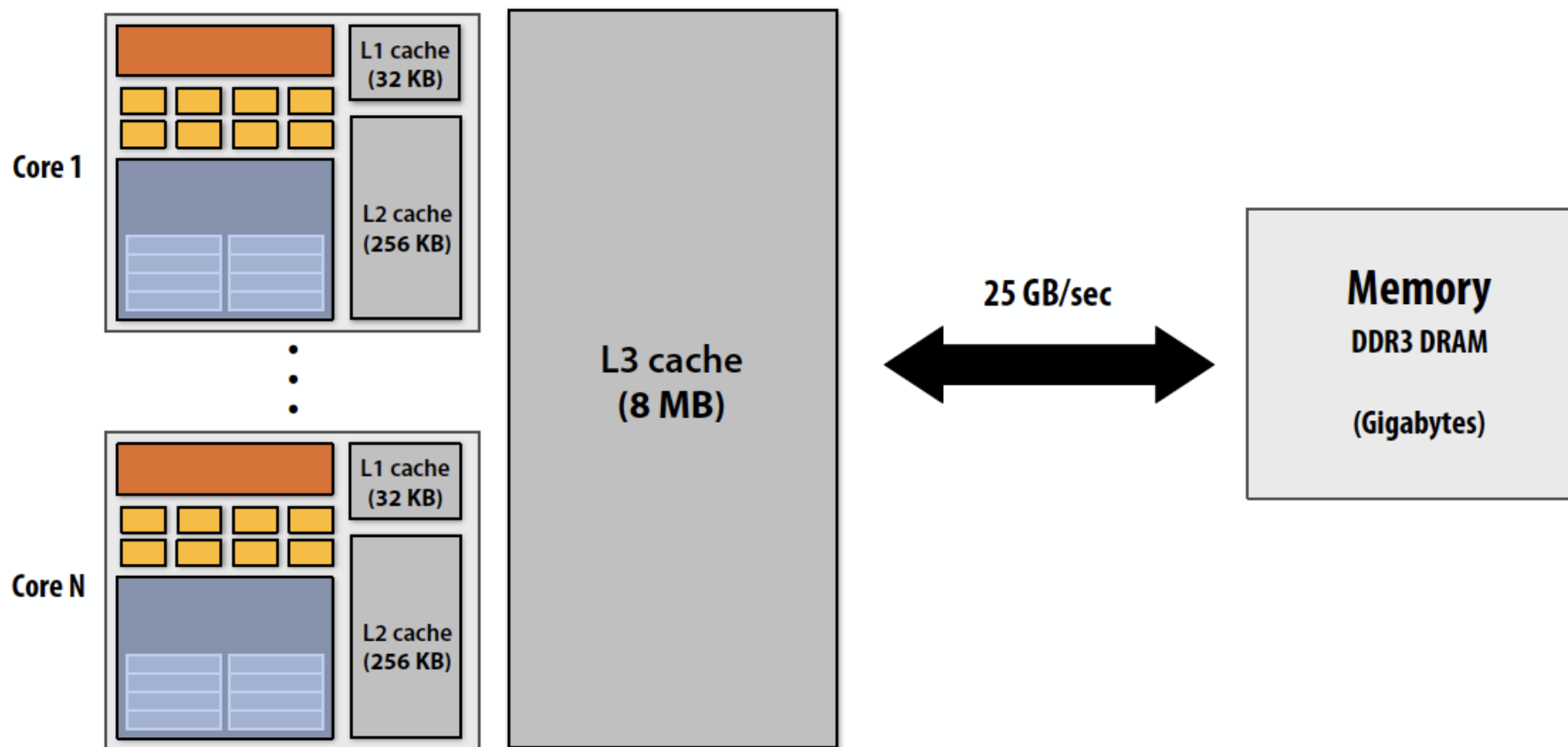


Advantage: increased throughput



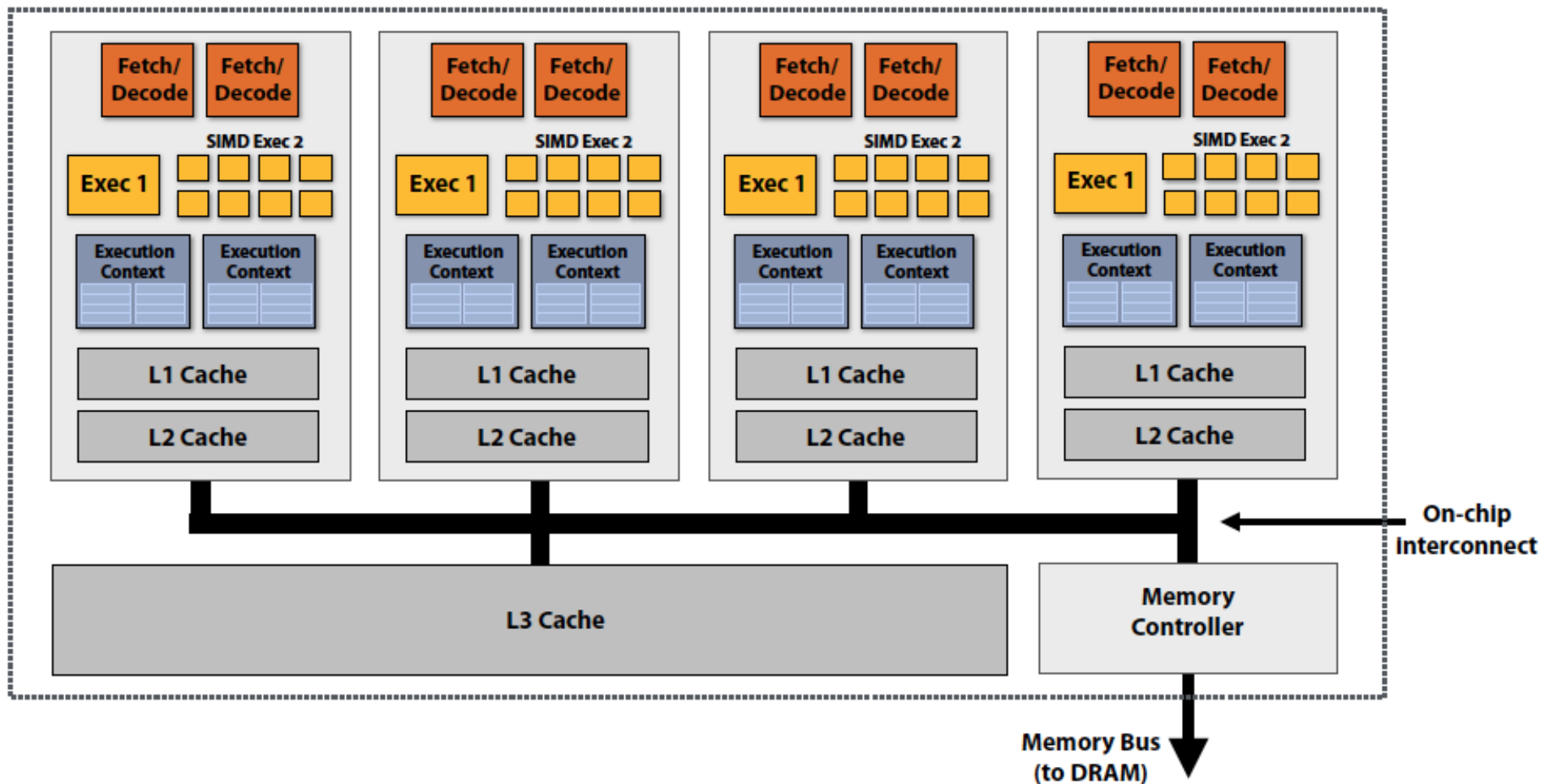
What about the memory?

- 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 ();
```

element	0	1	2	3
value	0.0	0.0	0.0	0.0

```
// Load the first 4 elements of the array into my vector.
```

```
__m128 myVector1 = __mm_load_ps (data);
```

element	0	1	2	3
value	0.0	1.0	2.0	3.0

```
// Load the second 4 elements of the array into my vector.
```

```
__m128 myVector2 = __mm_load_ps (data+4);
```

element	0	1	2	3
value	4.0	5.0	6.0	7.0

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  
    }  
}
```