SUSTAINABLE COMPUTING

Part2: systems, apps, tools

Ana-Lucia Varbanescu
a.l.varbanescu@utwente.nl

UNIVERSITY OF TWENTE.

Agenda

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

Part 1

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

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

Sustainable computing ⇔ zero-waste computing

Raise awareness

- Monitor (energy) efficiency
- Quantify waste

Performance analysis

Performance modeling

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 optimization

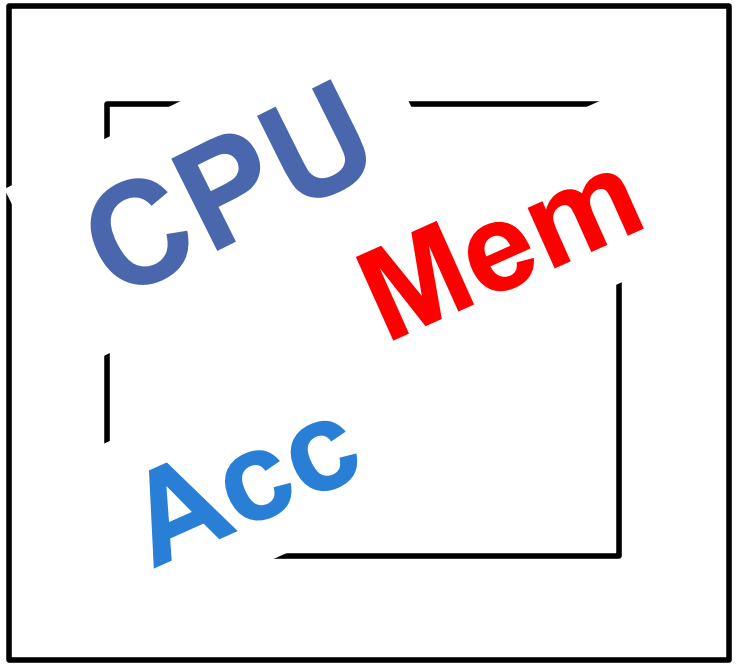
Efficient scheduling and resource sharing

Application-centric system design

??

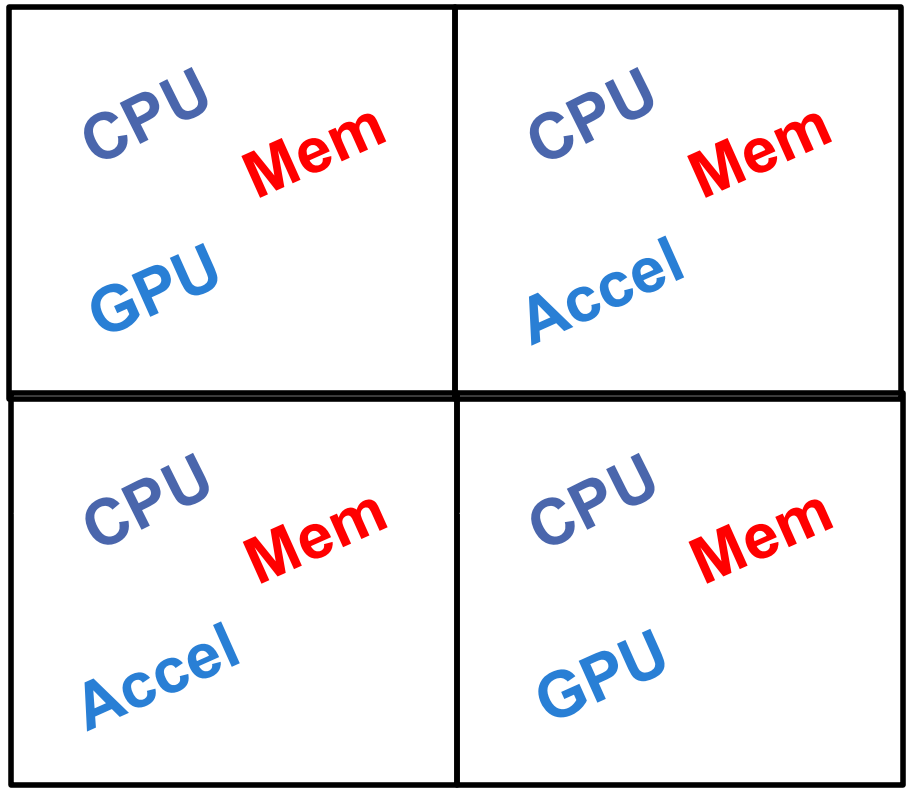
“Systems at hand”

Making faster/larger systems



Scale-up:
Bigger machine …

and



Scale-out:
More machines …

Sustainability at scale

- Programmer exposes parallelism at application level
  - Job = application + dataset
  - Application = set of tasks
  - Tasks = execute in some sequential order and/or in parallel

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

Knowledge of node architecture is essential for effective optimization.

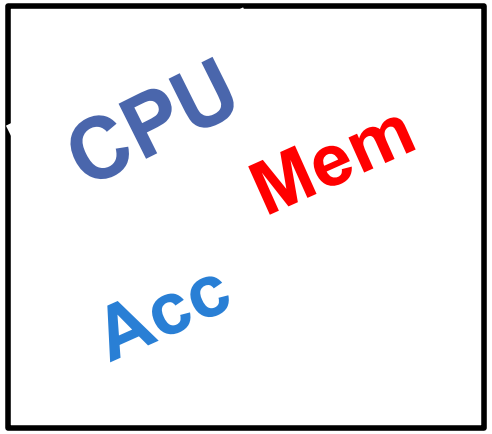
- Runtime/OS map the tasks on resources
  - In both space and time
  - Possibly with programmer's restrictions

What runs where and when?

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

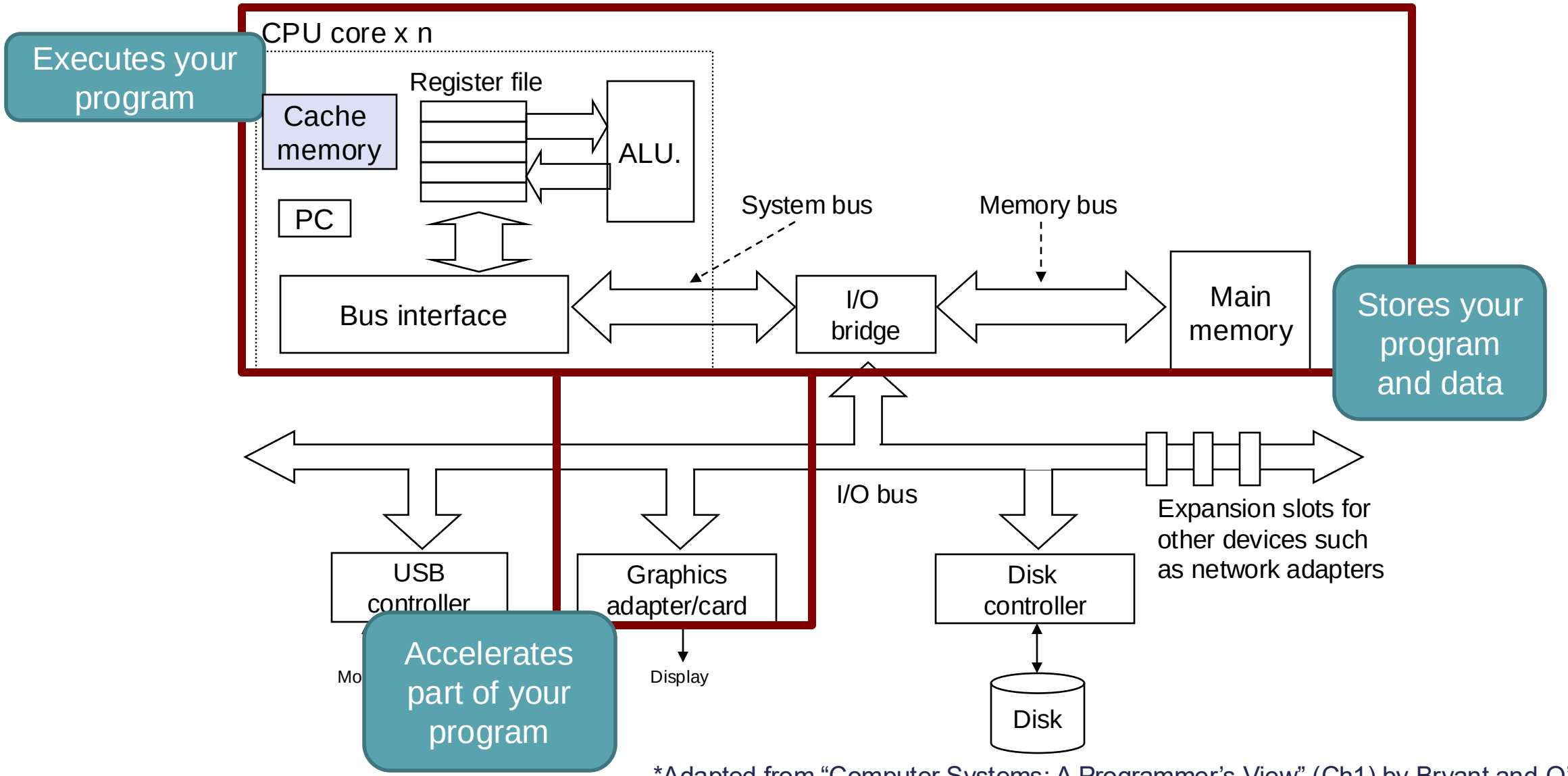
- (Job) Scheduler en
  resources
  - Ideally sufficient an

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



Systems SOTA: Inside the node

Inside the node*



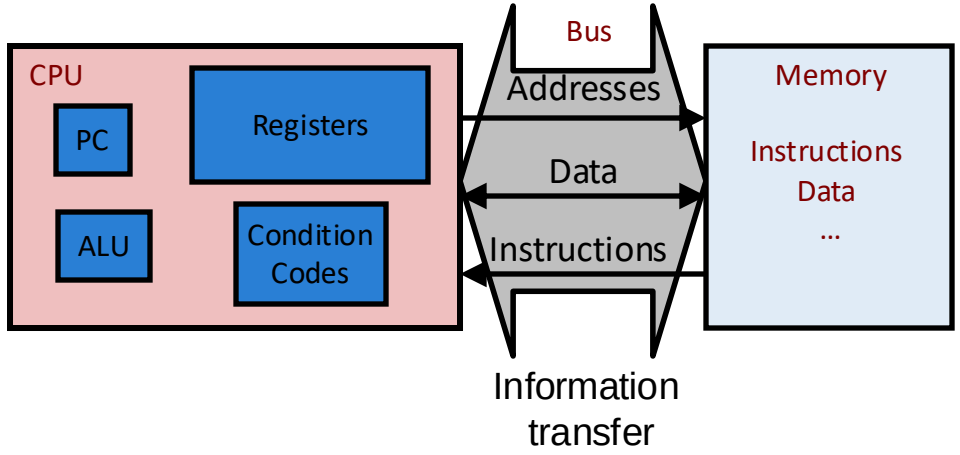
*Adapted from “Computer Systems: A Programmer’s View” (Ch1) by Bryant and O’Hallaron

Systems 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}

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
- Scale-up 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!

CPU levels of parallelism

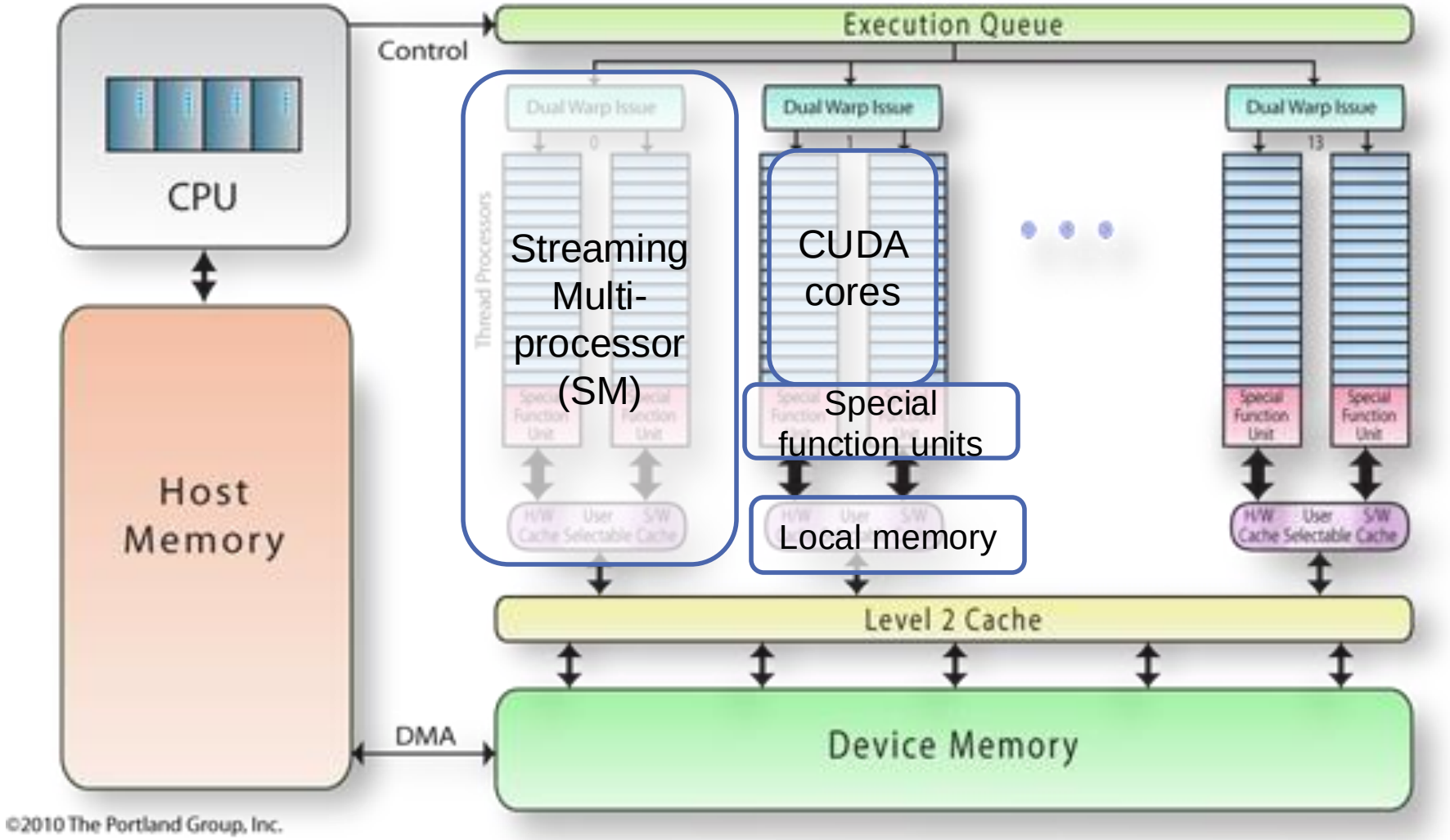
- **Instruction-level** parallelism (e.g., superscalar processors) (fine)
  - Multiple operations of different kinds per cycle
  - Implemented/supported by the instruction scheduler
    - typically in hardware
- **SIMD** parallelism = data parallelism (fine)
  - Multiple operations *of the same kind*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

Programer's intervention and OS support

Acc: a generic GPU



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!

Programer's intervention!

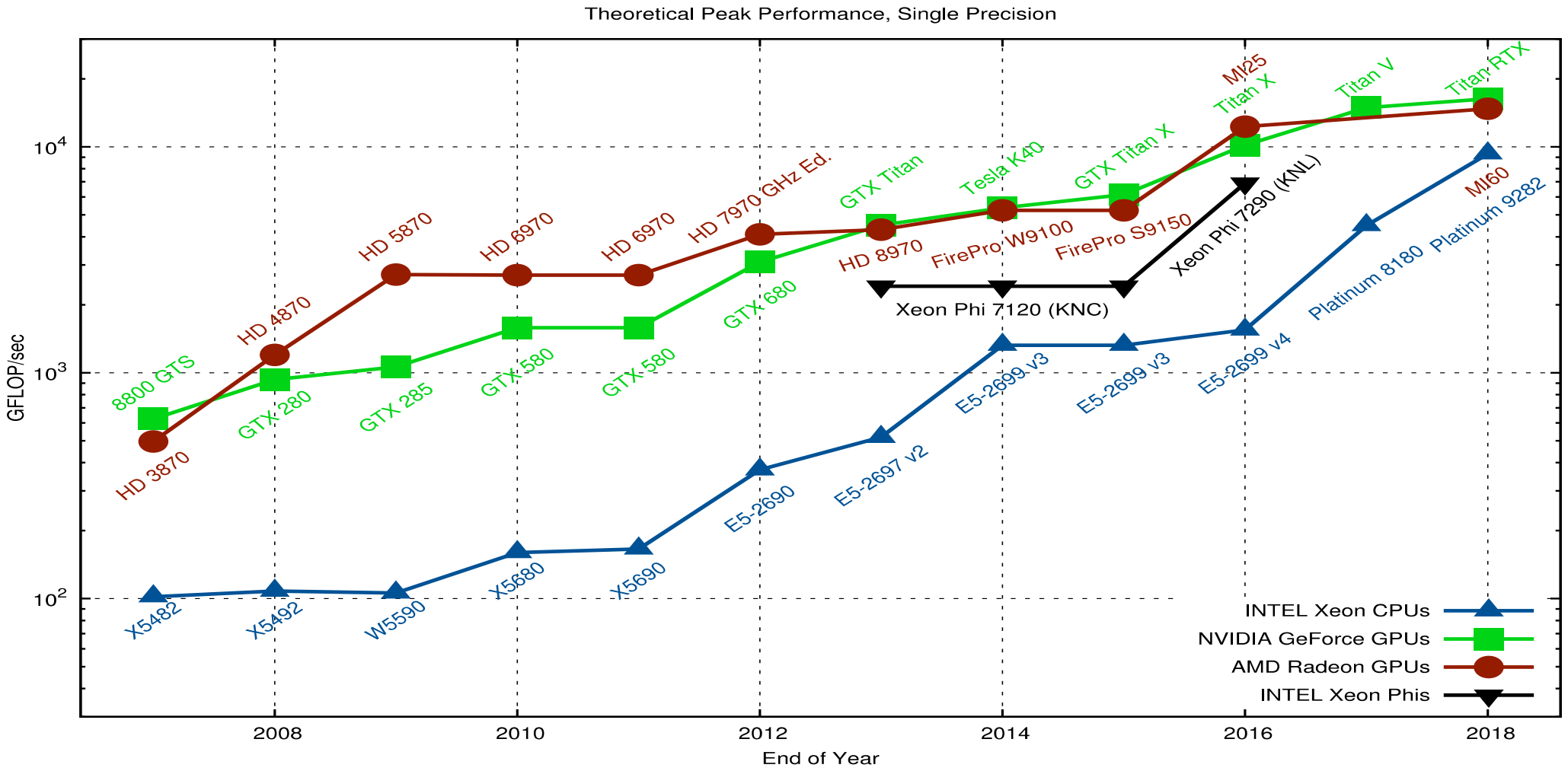
*http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

Theoretical peak performance

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

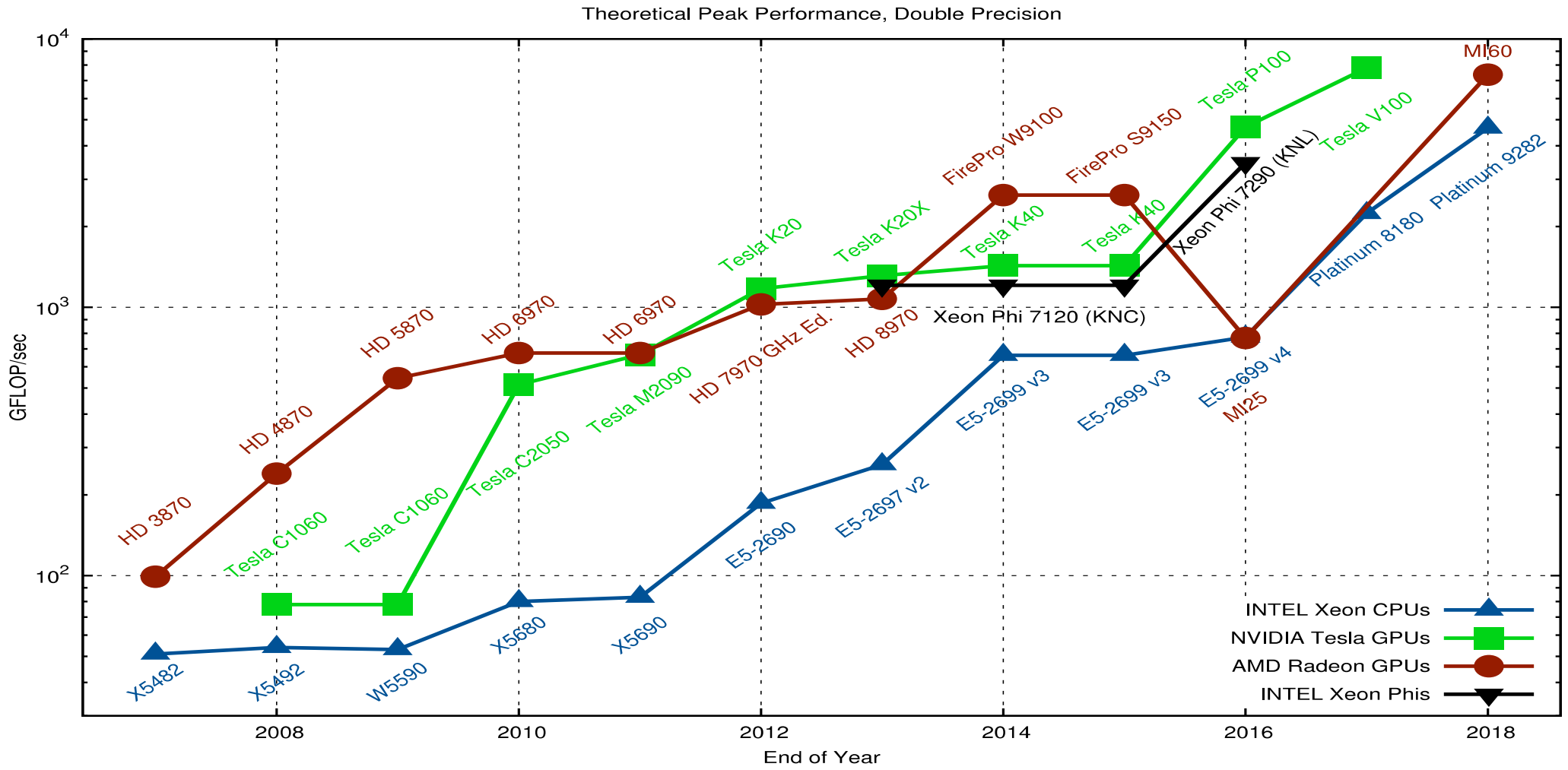
	Cores	Threads/ALUs	Throughput	Bandwidth
Intel Core i7	4	16	**85**	
AMD Barcelona	4	8	**37**	
AMD Istanbul	6	6	**62.4**	
NVIDIA GTX 580	16	512	**1581**	
NVIDIA GTX 680	8	1536	**3090**	
AMD HD 6970	384	1536	**2703**	
AMD HD 7970	32	2048	**3789**	
Intel Xeon Phi 7120	61	240	**2417**	

multi vs *many* cores (SP-FLOPs)



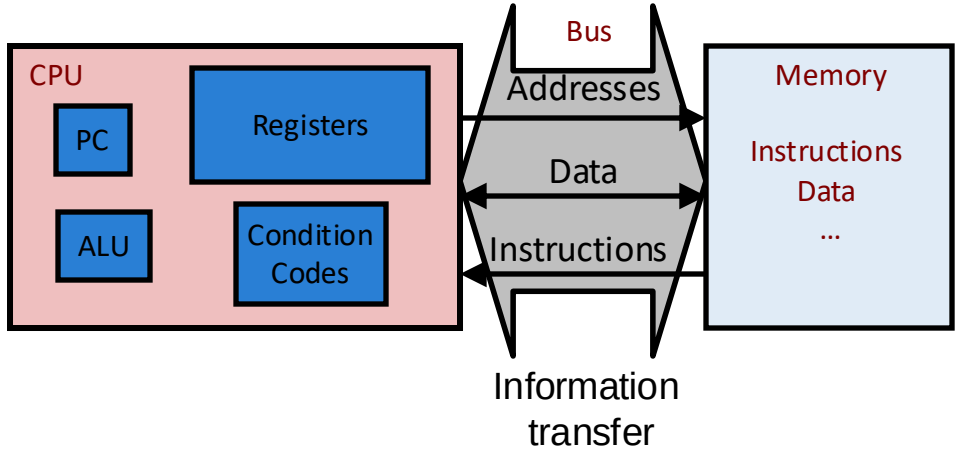
https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

multi vs *many* cores (DP-FLOPs)



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

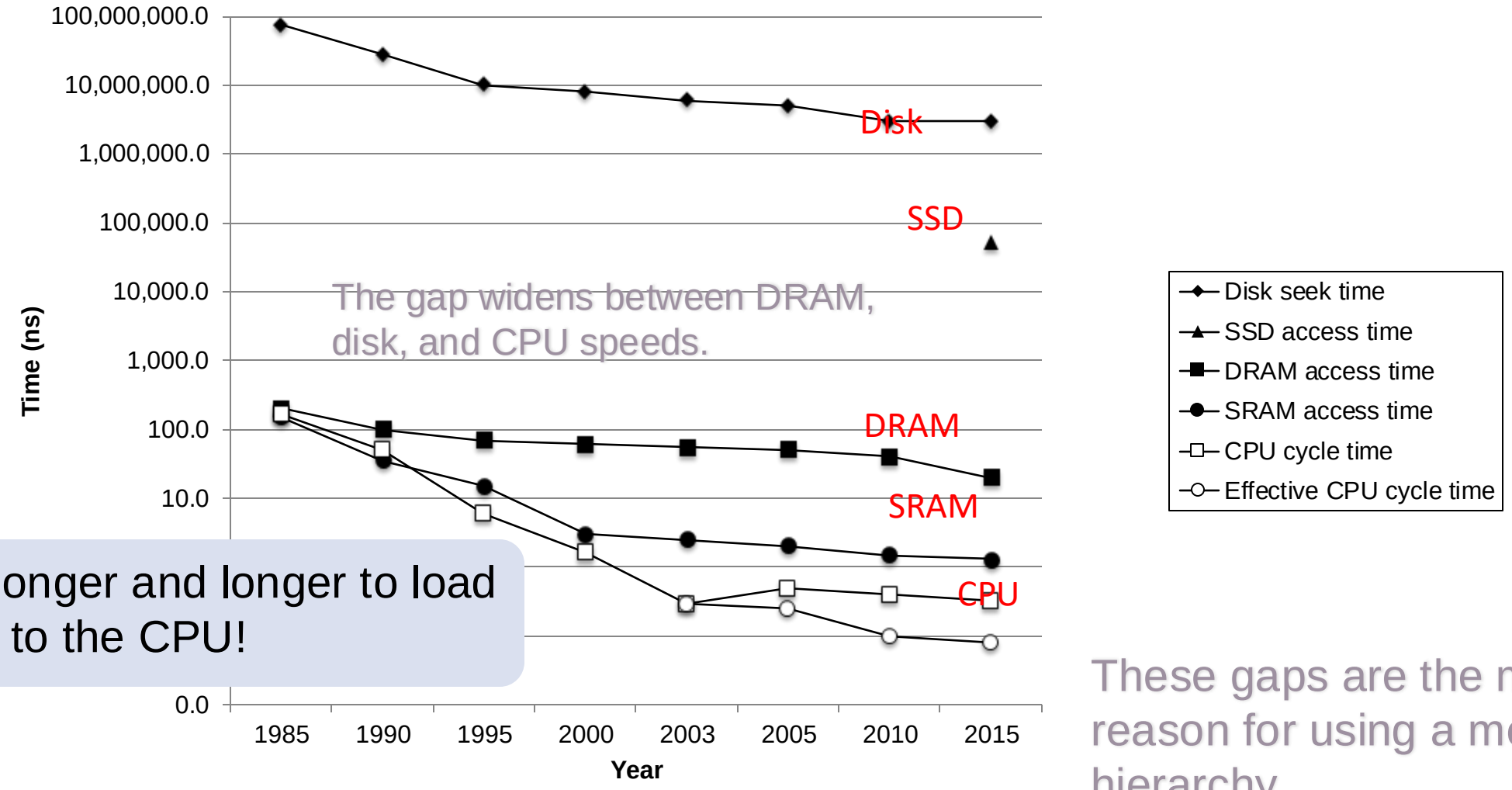
Memory



- Typically organized as linear spaces
  - Some word-size granularity
- App's code and data are stored in memory
  - Memory layout = **how** code/data is laid-out in memory
  - Access pattern(s) = **what code/data** is accessed and **when**
- 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.

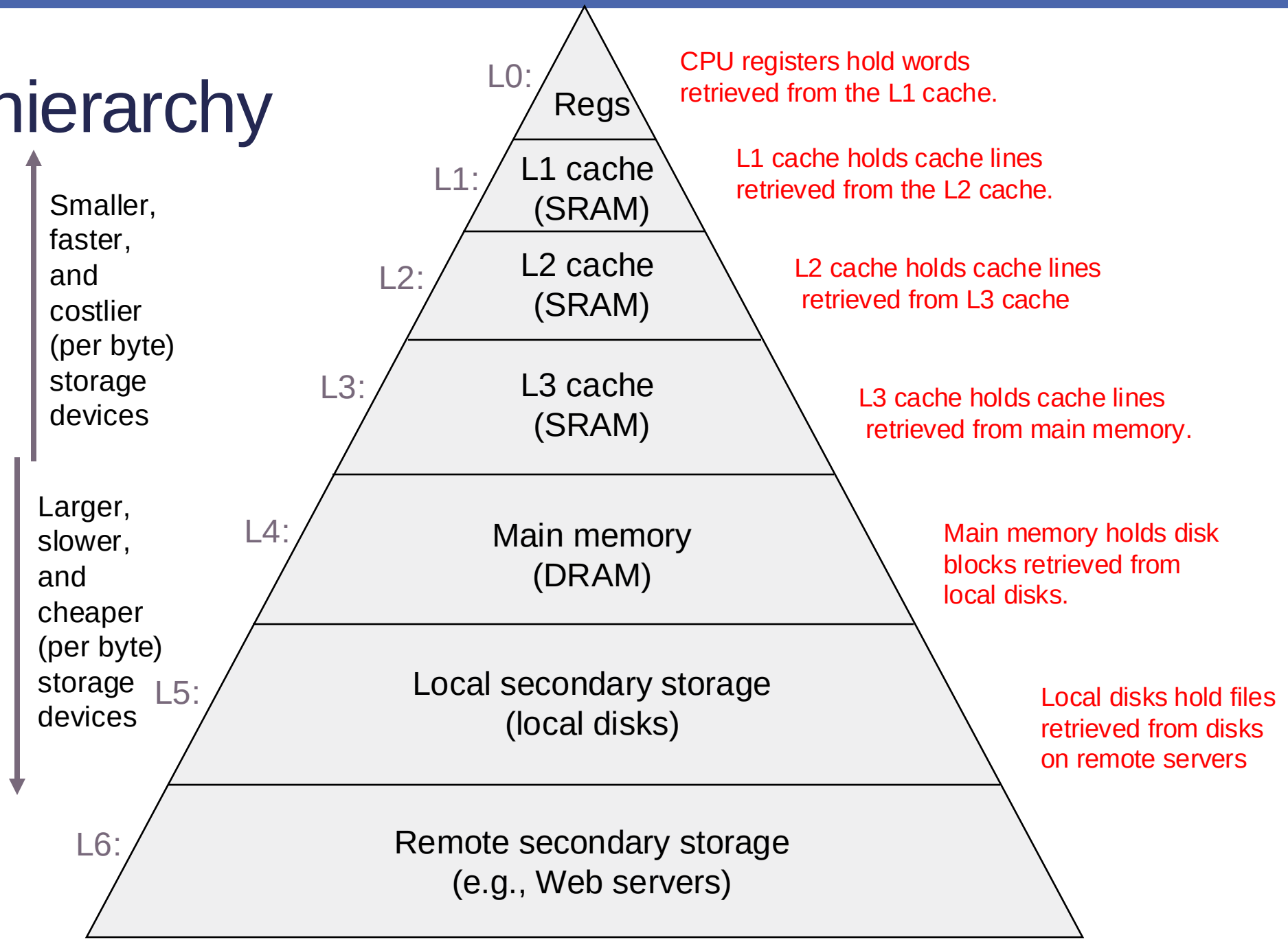
Memory hierarchy and caches

- **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.*

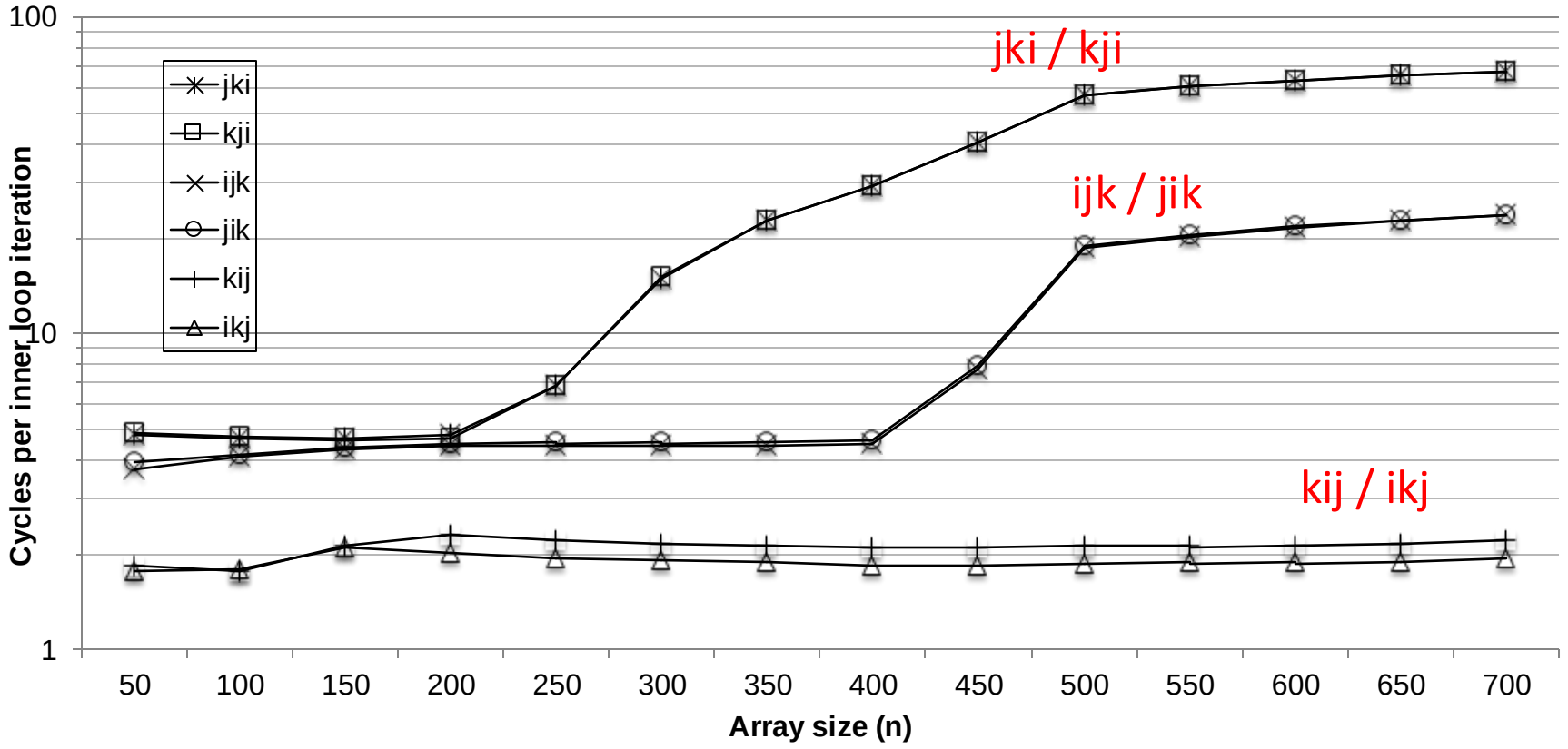
- How/why do memory hierarchies work?
  - Predict what the app uses next & **prefetch ⇔ principle of locality**
  - Locality <=> data at level k is used more often than data at level k+1.
    - Level k+1 can be slower, and thus larger and cheaper.

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

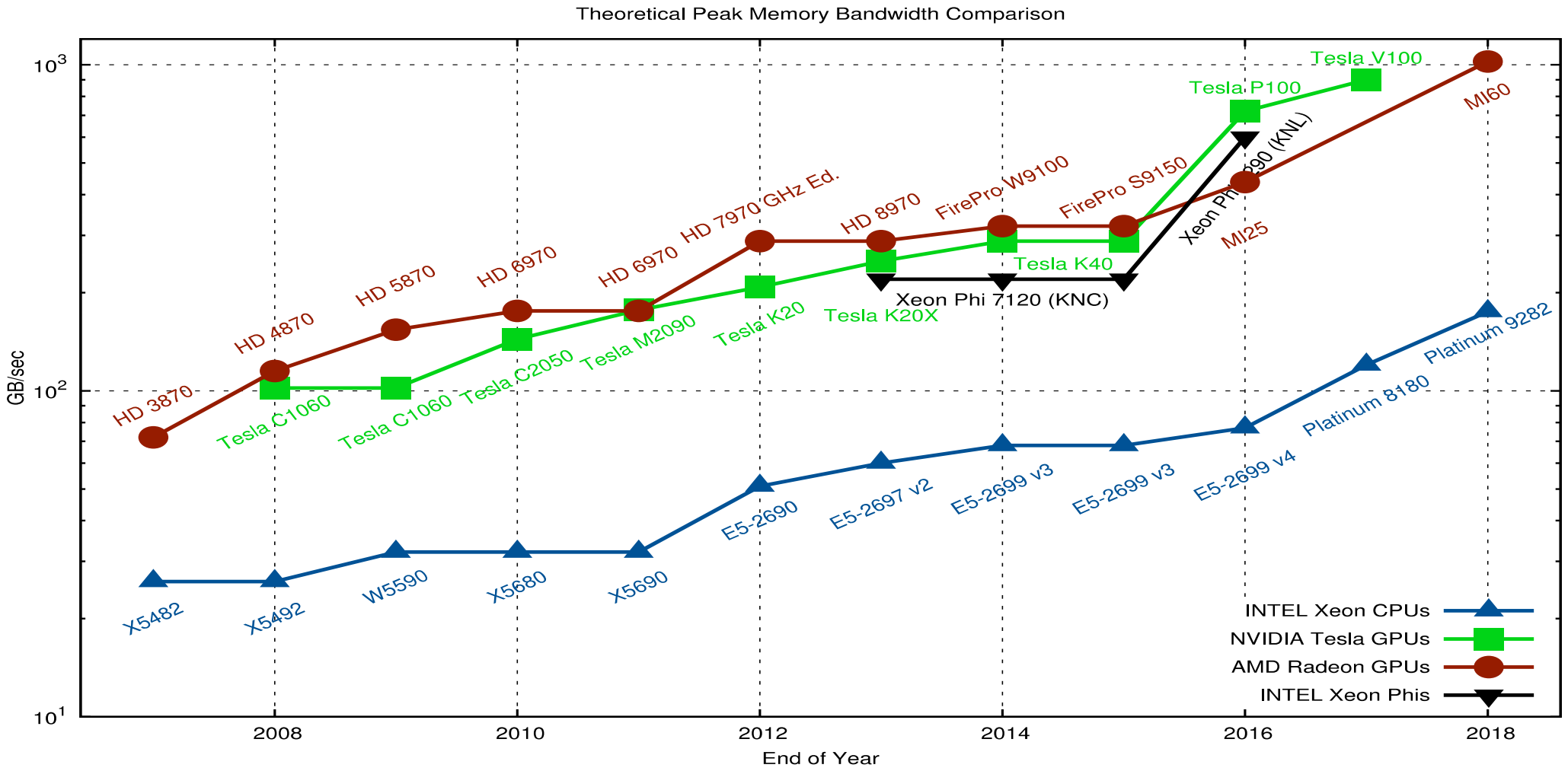
Theoretical peak performance

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)



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

What about energy?

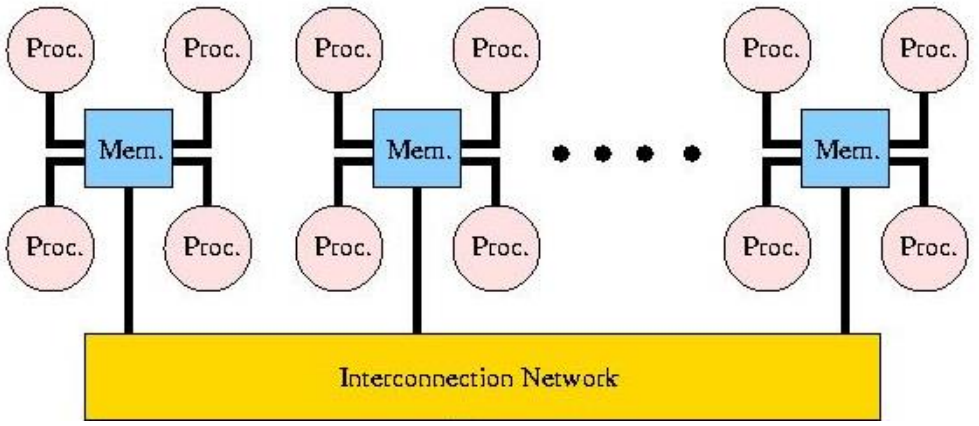
- Multi-core CPU
  - Multi-core energy consumption != N * energy/core
    - Complex architecture, different clocks, shared resources
  - Various ways to implements DVFS + power reduction techniques
  - Non-trivial correlation with performance

- GPU
  - Power is significantly impacted by the type of workload and occupancy
  - Always check the power cap, too!

- Heterogeneous computing
  - Sum of energy by components works well

- Multi-node computing:
  - Sum of energy works OK
  - Missing networking energy

Hybrid



CPU C-states/P-states

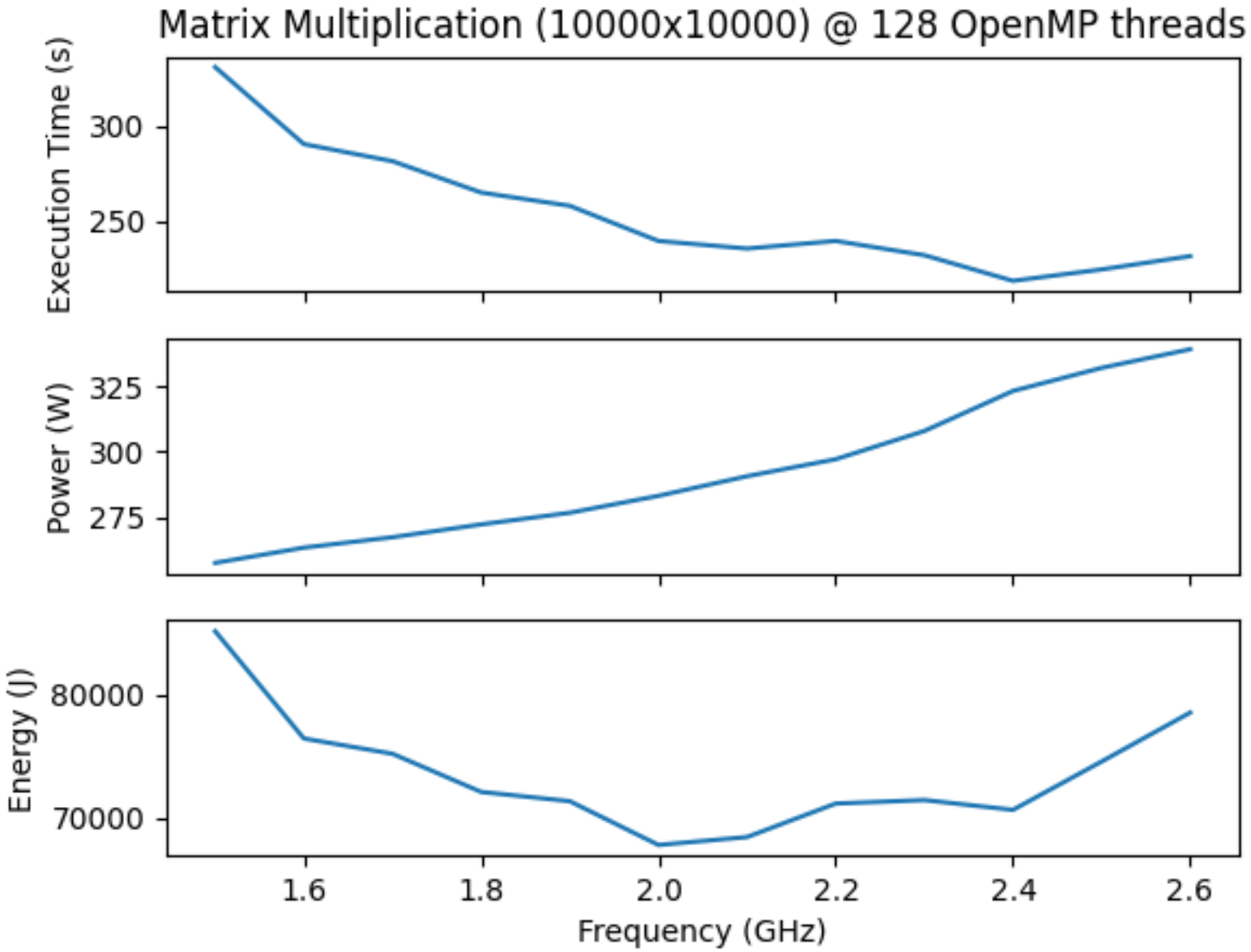
- P-states = power performance states
  - scale the frequency and voltage at which the processor runs
    - reduce the power consumption of the CPU.
  - Number of available P-states can be different for each model of CPU

- C-states = idle sleep states
  - reduced or turned off selected functions
    - Higher is … more => more of the CPU is shut-off
  - Number of available C-states can be different for each model of CPU

https://hardwaresecrets.com/everything-you-need-to-know-about-the-cpu-c-states-power-saving-modes/

An example

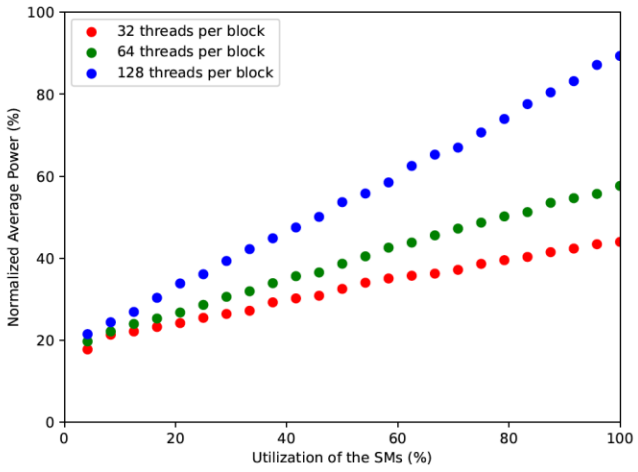
- AMD EPYC CPU
- Running SGEMM
  - Different frequencies (P-states)



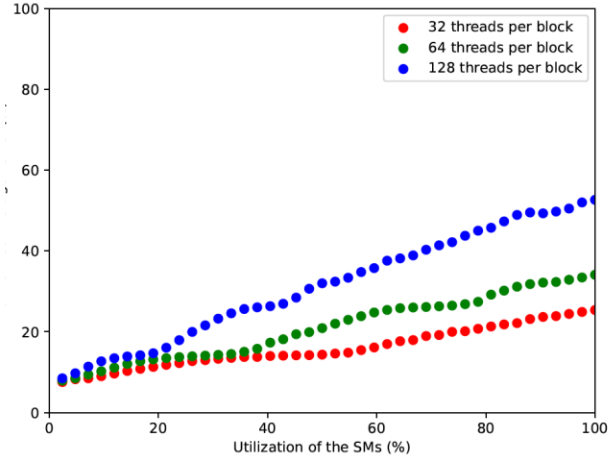
An example:

GPU	SMs	Cores/SM	Total Cores	Max Power [W]	Idle Power [W]
A4000	48	128	6144	140	39.5
A6000	84	128	10572	300	71.5
A2	10	128	1280	60	18.1
A100	108	64	6912	250	37.9

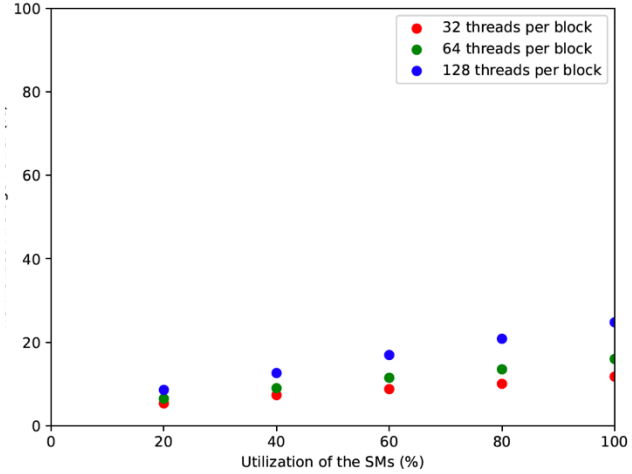
A4000



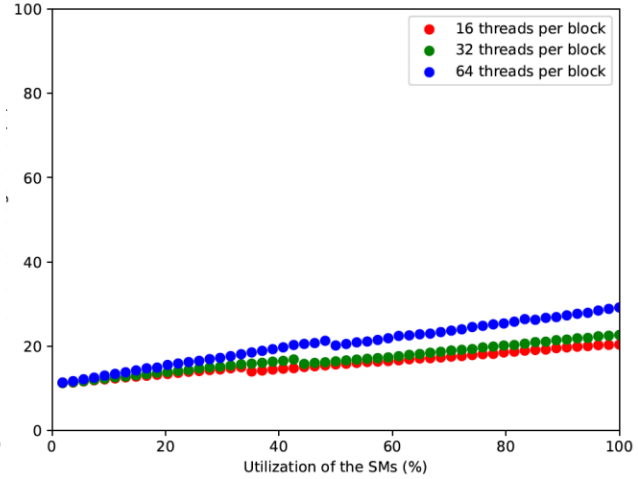
A6000



A2

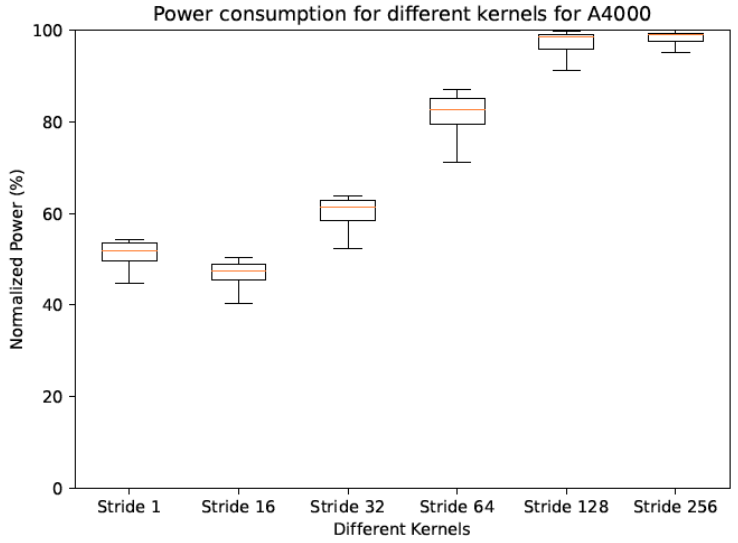


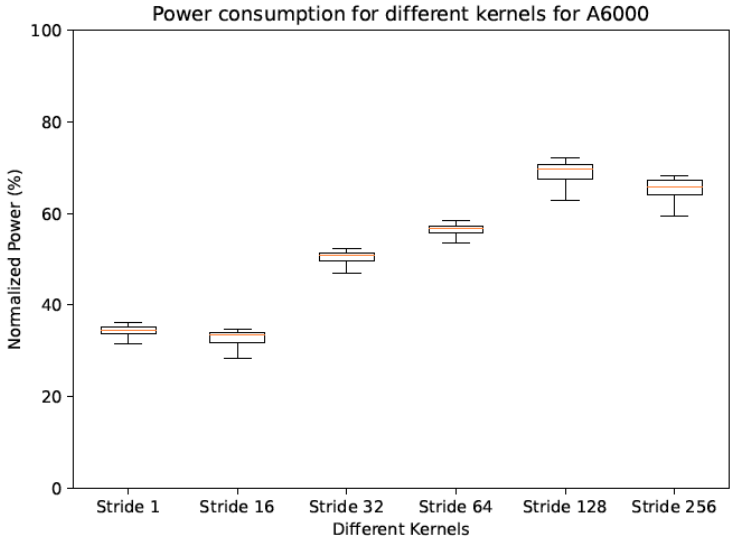
A100

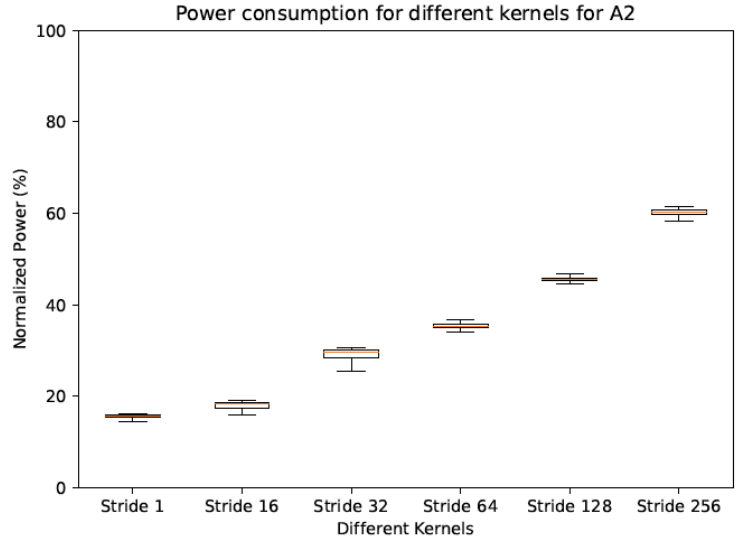


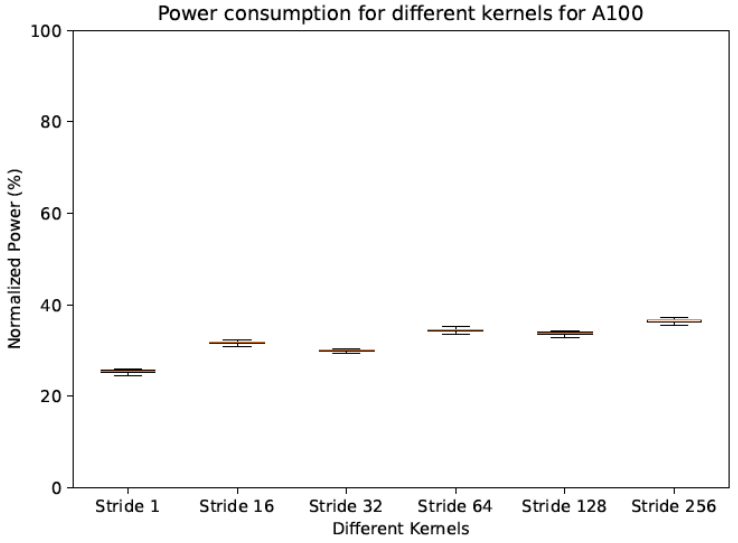
Same example

- Caching patterns make a significant difference
- Compute vs memory intensive – mem consumes more.
- Memory coalescing, negligible
- Data types & instruction mix show some differences









Performance vs. sustainability

- Low performance ➔ waste in computing
  - We power resources that are not needed
- High performance ➔ faster execution ➔ less energy consumed
  - Max energy efficiency ⇔ max performance (i.e., lowest runtime…)
    - Strong assumption that power is constant
- DVFS is a technique to reduce the impact of underutilized hardware
  - Non-linear effects on performance
- In the context of multi-core and heterogeneous systems, maximizing execution time might not guarantee lowest energy consumption.

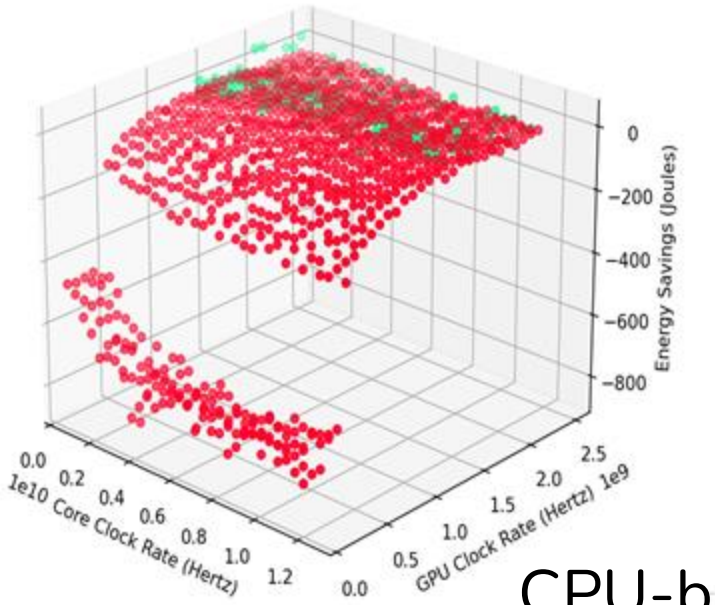
Goel and McKee – "A Methodology for Modeling Dynamic and Static Power Consumption for Multicore Processors"

Example 1: Energy harvesting

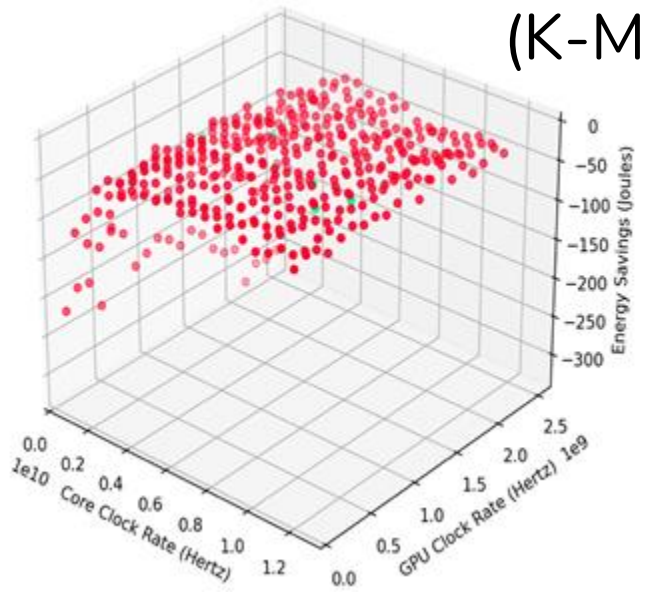
- Basic assumptions
  - Tasks run on different processors
  - Idle processors waste energy
  - Higher/lower operating frequencies
    - => more/less power respectively
    - => reduce or increase runtime respectively

- Opportunities
  - Dynamic Voltage and **Frequency** Scaling (DVFS)
  - Reducing operating frequencies in idle states may save energ
    - No active task => no runtime increase
  - Increasing operating frequencies in busy states may save ene
    - Lower runtime => less time to consume energy

GPU-bound
(Matrix Multiply)



CPU-bound
(K-Means)



Results

Applications	Best Policy					
	Single Core			Multi Core		
	Name	Energy	Time	Name	Energy	Time
BFS	Scaled MinMax	-0.5%	0.2%	Ranked MinMax	0.9%	1.2%
LavaMD	Maximum Frequency	-0.7%	-0.1%	MinMax	-6.6%	1.0%
NW	Ranked MinMax	4.8%	4.4%	Ranked MinMax	-7.9%	21.0%
Particlefilter-float	Ranked MinMax	-0.0	1.5%	Ranked * MinMax	-10.2%	14.8%
Kmeans	Ranked MinMax	3.7%	0.6%	Ranked MinMax	-3.8%	4.1%
Bandwidth	Maximum Frequency	-2.3%	0.1%	Maximum* Frequency	-2.7%	1.2%
UnifiedMemoryPerf	MinMax	-1.5%	-3.8%	Scaled MinMax	-16.2%	-2.8%
matrixMul	Maximum Frequency	3.5%	-0.0%	Maximum Frequency	8.5%	-0.2%
Jacobi unoptimized	MinMax	-3.5%	-7.4%	Maximum Frequency	-26.8%	-7.7%
Jacobi optimized	MinMax	-2.7%	-9.4%	Maximum Frequency	-34.8%	-10.0%

Example 2: Smaller GPUs, anyone?

Applications:

- 5 Rodinia kernels:
  - **Compute**-bound: hotspot, k-means (2)
  - **Memory**-bound: k-means (1)
    backpropagation (1), backpropagation (2)

Systems:

- Baseline: RTX 2060 Super
- Variables:
  - **SMs**: 25, 30, …., 40
  - **Core clock**: 1000, 1150, …., 1900
  - **Memory clock**: 800, 1250, …, 3500

Simulation run-time ≈ 24-40 hours

Simulated with:

https://github.com/romnn/gpucachesim

Varying SMs

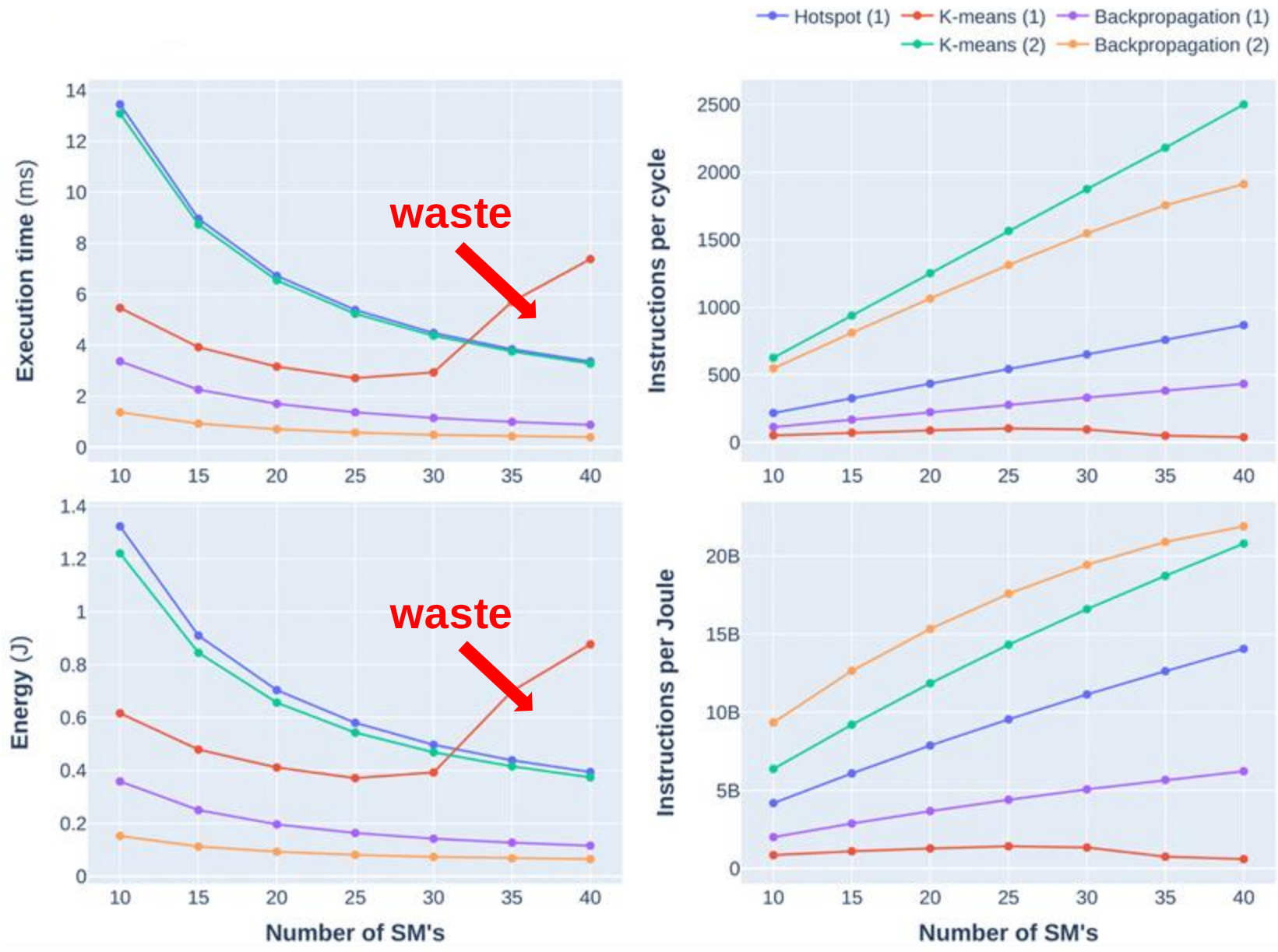
Compute-bound:

- Hotspot
- K-means (2)

Memory-bound:

- K-means (1) ←
- Backprop (1)
- Backprop (2)

More resources ≠ better performance



Core clock

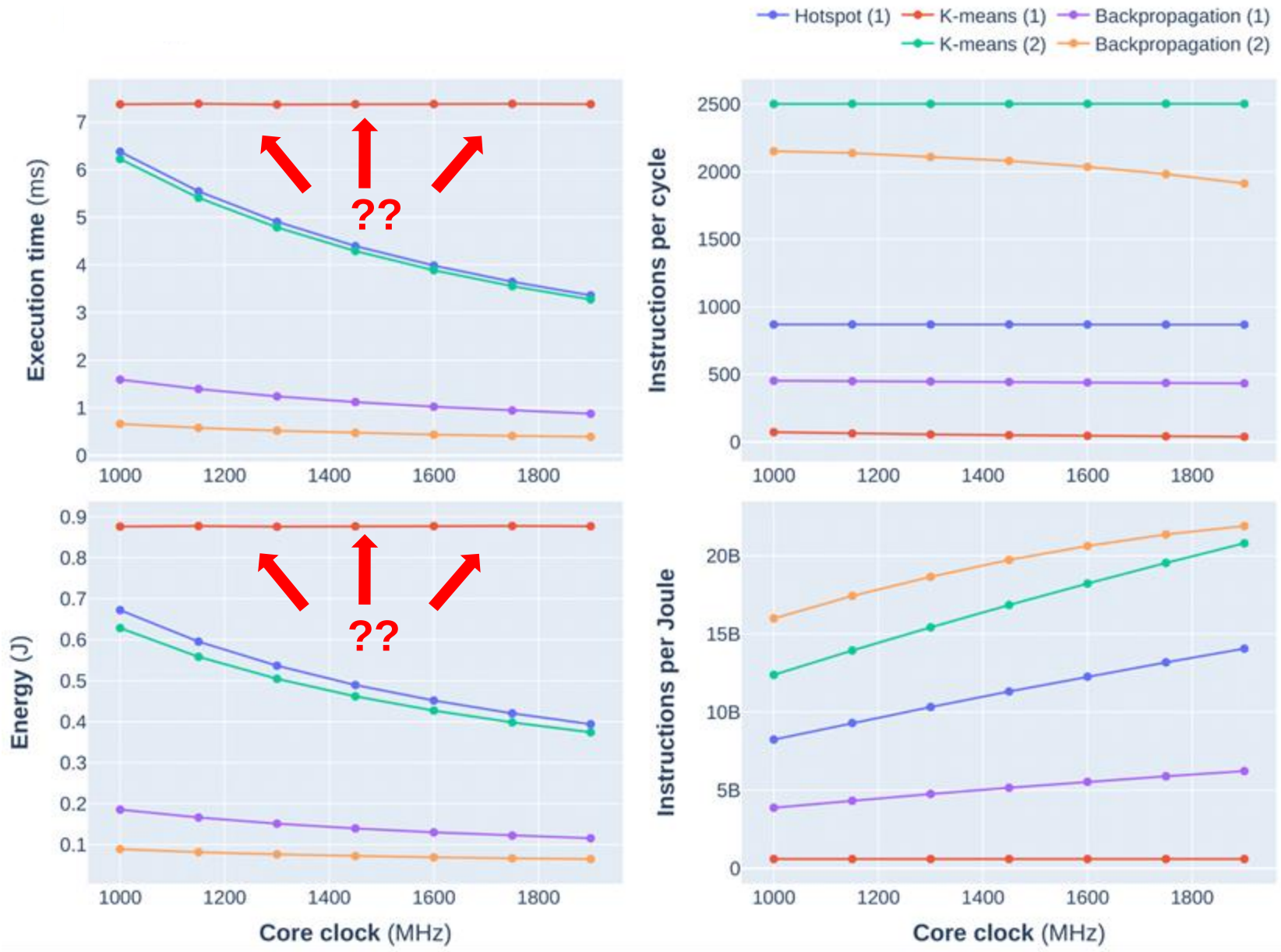
Compute-bound:

- Hotspot
- K-means (2)

Memory-bound:

- K-means (1) ⟵
- Backprop (1)
- Backprop (2)

The core-level static energy model in AccelSim seems to be broken.



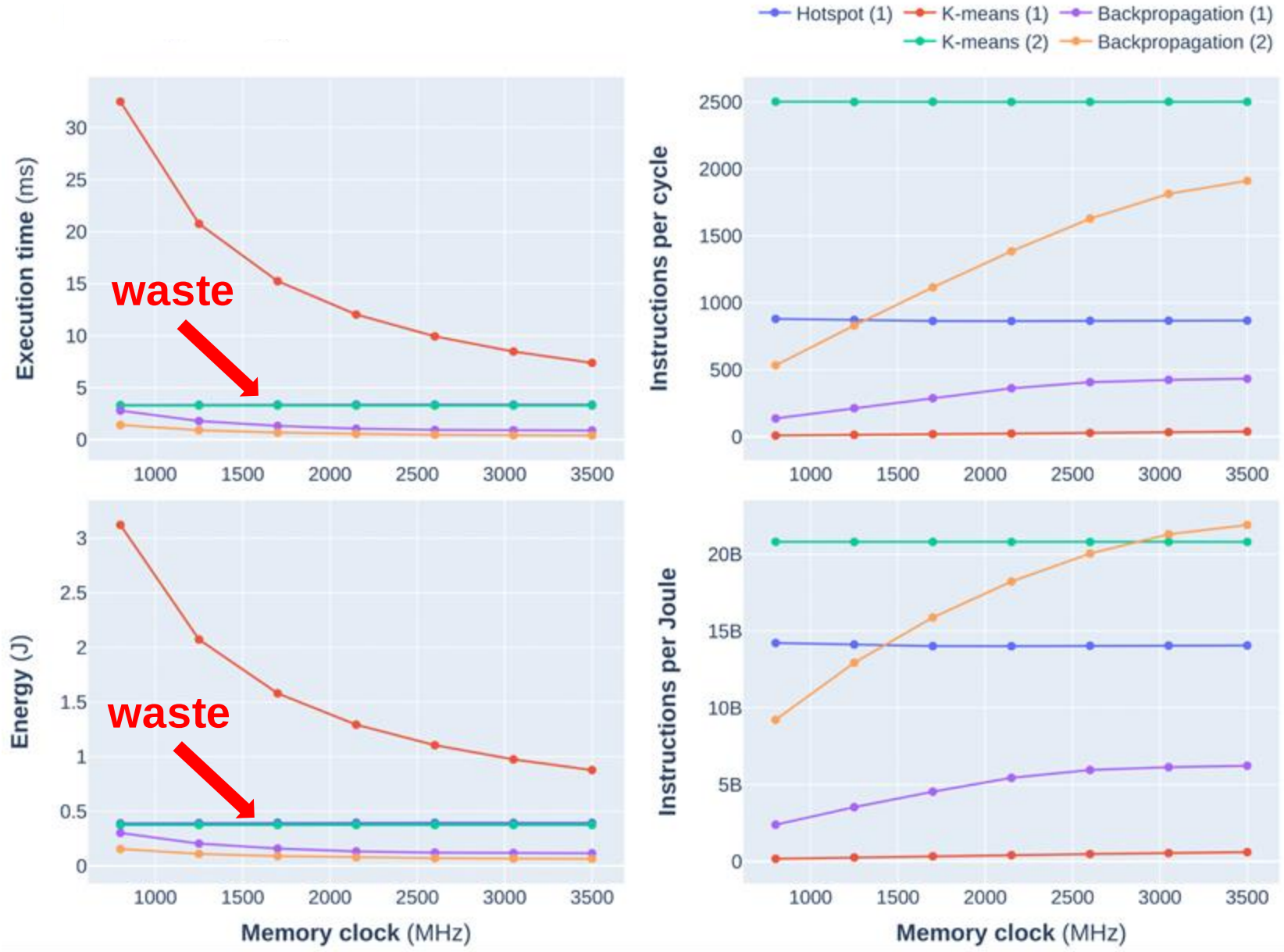
Memory clock

Compute-bound:

- Hotspot
- K-means (2)

Memory-bound:

- K-means (1)
- Backprop (1)
- Backprop (2)



Sustainability TODO's

- Use all CPUs capabilities
  - Maximize parallelism
  - Use SIMD & ILP
- Use accelerators – if needed!
  - Maximize parallelism
  - Efficient mapping
  - Heterogeneous computing
- Use all memory capabilities
  - Maximize bandwidth
  - Use caching / improve locality

Limit the impact of low utilization

Reduce impact of unused cores/chips

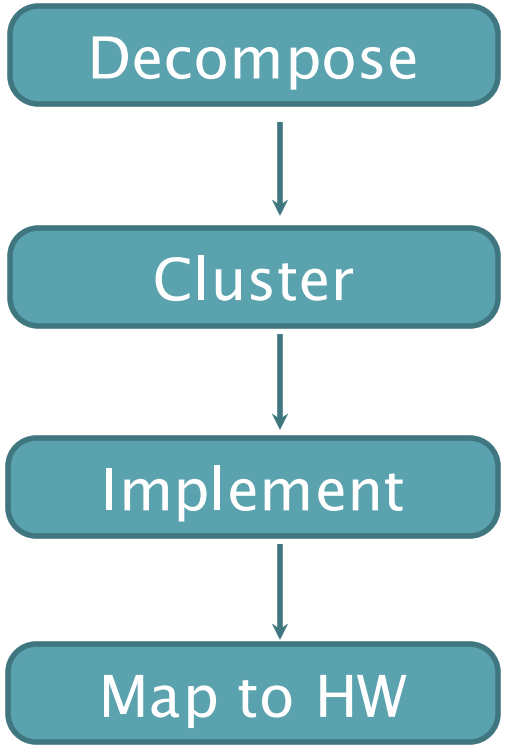
Reduce impact of unused bandwidth**

** non trivial …

Applications

A Method for Parallel Application Design*

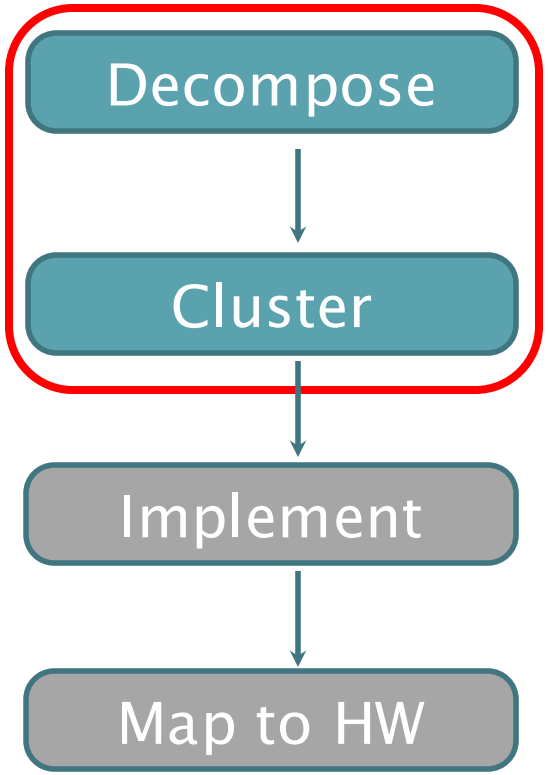
- Decompose (partition)
  - What is the computation? Which data?
- Cluster (communicate & agglomerate)
  - What granularity ?
- Implement in programming model
  - Which model ?
  - Implement tasks as processes or threads
  - Add communication and synchronization constructs
- Map units of work to processors
  - Might be transparent.



*In Foster - ”Designing and Building Parallel Programs”
http://www.mcs.anl.gov/dbpp

Models of Parallel Computation

- Conceptual level : **defining tasks and data interactions**
  - Recipes that typically cover “decompose” and “cluster”
- (may) Provide
  - More parallelism in the application
  - Better load-balancing
  - Systematic performance analysis
- Examples
  - Trivially/embarrassingly parallel
  - Data parallelism
  - Task parallelism
  - Farmer/worker
  - Divide and conquer
  - Bulk synchronous



Implementation matters

- Efficient parallelization
- Memory bandwidth utilization
- Programming models/languages
- Compilers
- Libraries and additional tools

The programming language

- C/C++ remain faster than most alternatives
  - And benefits from most low-level tools
- Python is slower, but libraries and ecosystem around it is improving
- Julia/Go/Rust picking up, but limited support with libraries and tools

n-body

source	secs	mem	gz	cpu secs
C++ g++ #0	2.15	19,736	1933	2.15
C++ g++ #7	3.96	19,736	1815	3.96
C++ g++ #2	4.20	19,736	1884	4.20
C++ g++ #3	5.31	19,736	1402	5.31
C++ g++ #9	5.44	19,736	1536	5.44
C++ g++ #4	5.45	19,736	1434	5.45
C++ g++ #5	6.02	19,736	1544	6.01
C++ g++	6.50	19,736	1666	6.50
C++ g++ #8	6.54	19,736	1524	6.54
C++ g++ #6	6.91	19,736	1674	6.91
Python 3	354.45	19,652	1201	354.43
Python 3 #2	360.53	19,652	1247	360.52
Python 3 #8	577.27	19,440	1202	577.25

https://benchmarksgame-team.pages.debian.net/benchmarksgame/

What about energy?

- Same experiment, with energy consumption metrics
- In most cases …
  - Correlation between time and energy
  - Depends a lot on the characteristics of applications

https://sites.google.com/view/energy-efficiency-languages/updated-functional-results-2020?authuser=0

n-body				
	Energy	Time	Ratio	Mb
(c) Rust ⇓9	37.79	3328	0.011	6
(c) Fortran	50.34	3581	0.014	1
(c) Ada ↓1 ⇓5	51.79	4098	0.013	3
(c) C++ ↑1	57.30	3770	0.015	2
(c) C ⇑2	59.45	4190	0.014	2
(c) Chapel ⇓18	60.16	5203	0.012	42
(c) Pascal ⇑6	64.87	5702	0.011	1
(v) Java ⇓14	65.15	5839	0.011	30
(c) Ocaml ⇑3	65.75	5857	0.011	2
(c) Go ⇑5	67.23	5899	0.011	2
(v) C# ↓1 ⇓8	68.16	6117	0.011	26
(c) Swift ↑1	73.06	6036	0.012	7
(v) Lisp ⇓3	75.25	6685	0.011	15
(i) JavaScript ⇓6	78.74	6763	0.012	27
(v) F# ↓2 ⇓8	79.41	7105	0.011	32
(i) TypeScript ⇓5	80.11	6861	0.012	28
(i) Dart ↑2 ⇓8	84.33	6827	0.012	47
(c) Haskell ⇑7	127.82	10037	0.013	6
(v) Racket ⇑1	281.87	22260	0.013	21
(i) Jruby ⇓7	1,889.85	98407	0.019	689
(v) Erlang ⇑4	1,998.66	150698	0.013	18
(i) PHP ↓1 ⇑7	2,468.30	179360	0.014	14
(i) Hack ↓1 ⇓3	3,399.70	243356	0.014	123
(i) Ruby ↓1 ⇑11	3,870.67	281470	0.014	8
(i) Lua ↑3 ⇑18	3,981.88	177251	0.022	2
(i) Perl ⇑17	4,663.48	335391	0.014	5
(i) Python ⇑13	7,879.92	559214	0.014	8

Adding parallelism, acceleration, distribution

- Pthreads/CUDA/OpenCL/SyCL are relevant low-level options
  - Challenging to code, high-performance
- OpenMP is mucht more programmer friendly
  - Support for accelerators
  - Support for SIMD
- OpenACC and alternative directive-based tools are still limited for complex applications
- Kokkos / Raja / others focus on portability and performance
  - Popular for scientific applications

- Distributed computing => often some form of lock-in
  - De-facto standard remains MPI
  - Alternative models exist, but steep learning curve

What about (energy) efficiency?

- How well do we utilize the system for parallel applications?
  - Amdahl's law
  - **Roofline model**
  - Performance modelling
    - Analytical
    - Machine learning
  - (micro)Benchmarking
  - Simulators

- How well do we utilize the system for distributed applications?
  - Analytical models (LogP)
  - Tools for performance estimation (Scalasca, ScoreP, Vampir, …)

Balance

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

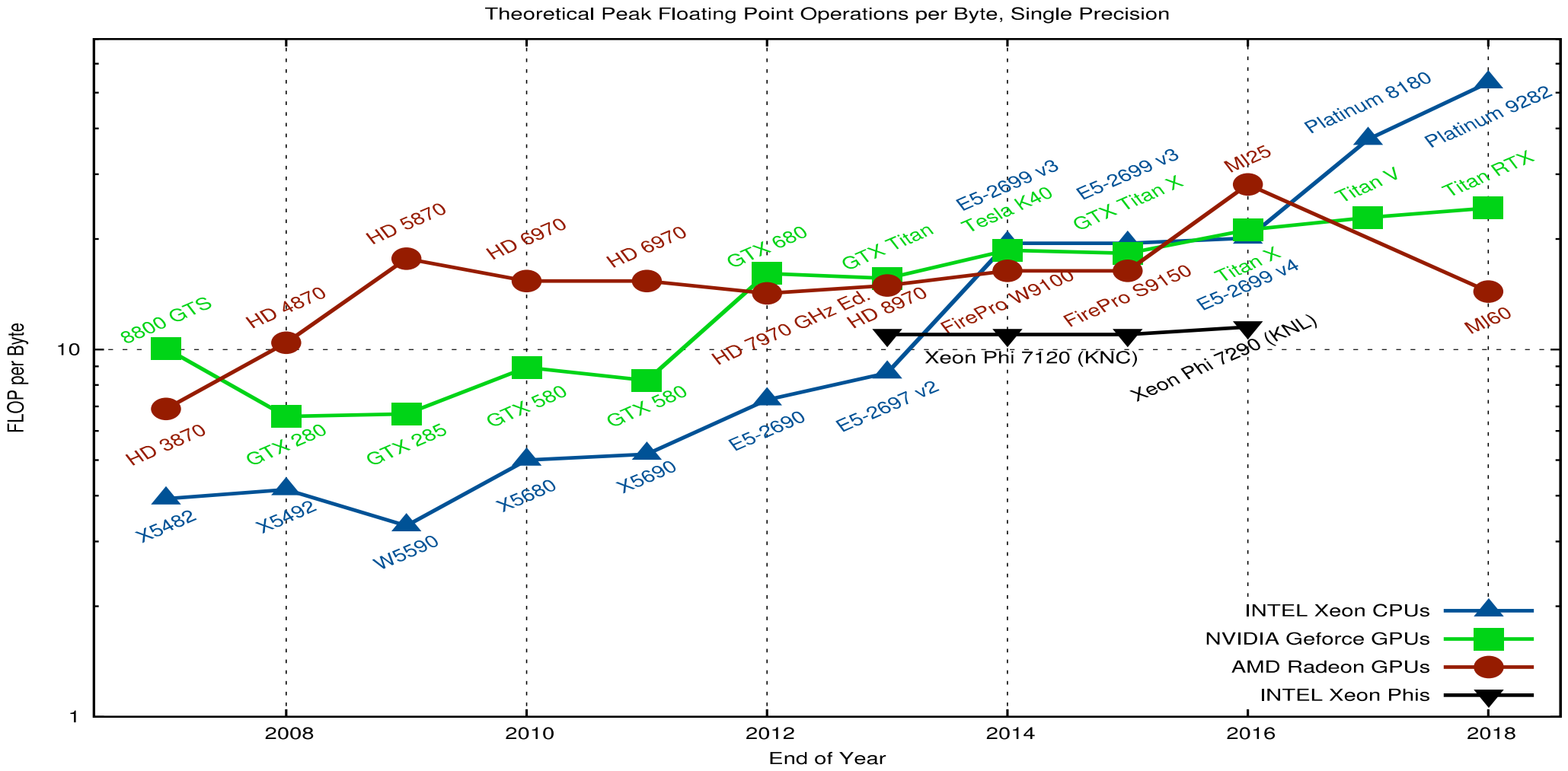
Bandwidth[GB/s] = memory bus frequency * bits per cycle * bus width

Bandwidth[FLOP/byte] = throughput / bandwidth

	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 ?

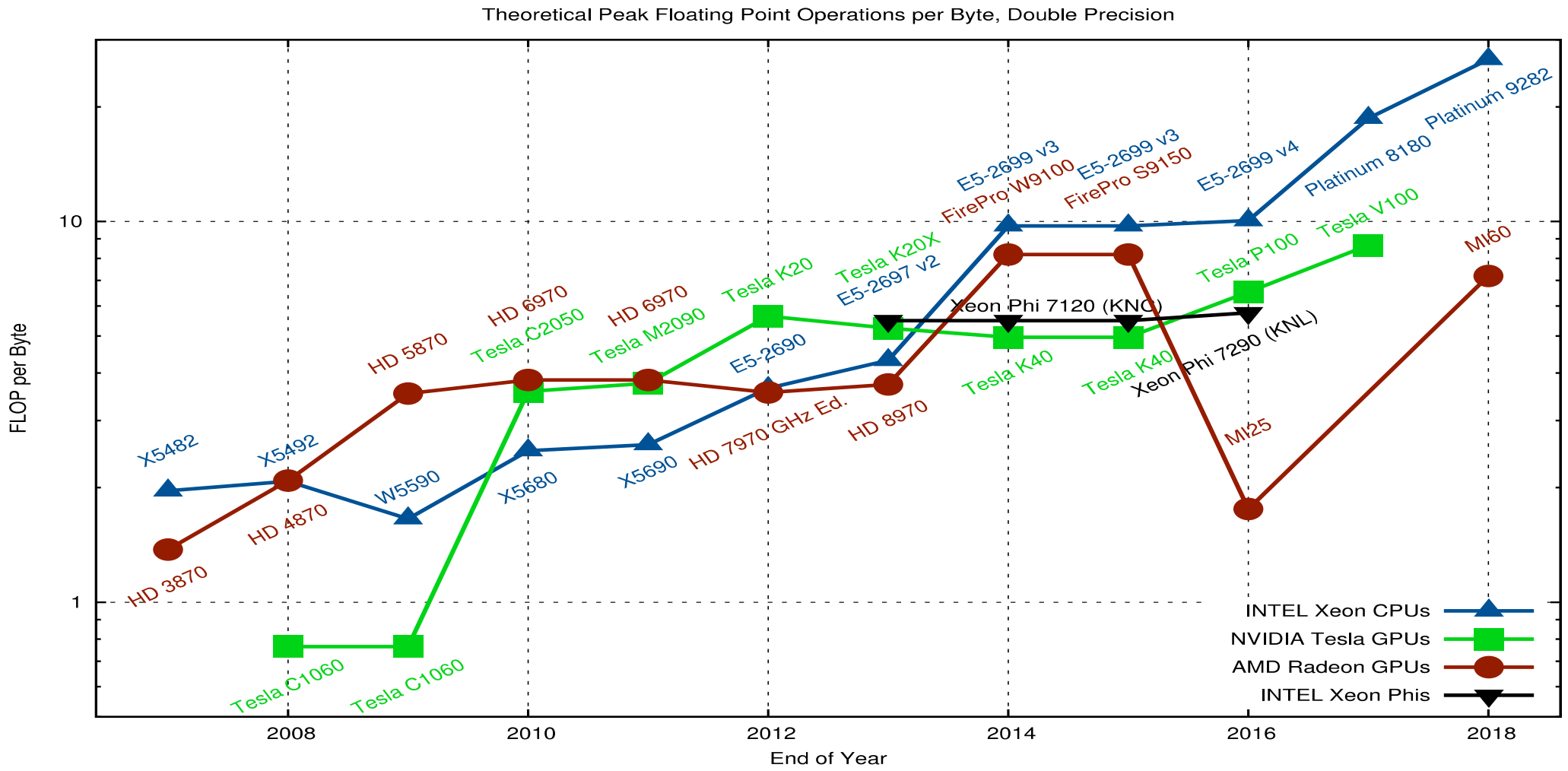
FLOPs/Byte (SP) !



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/

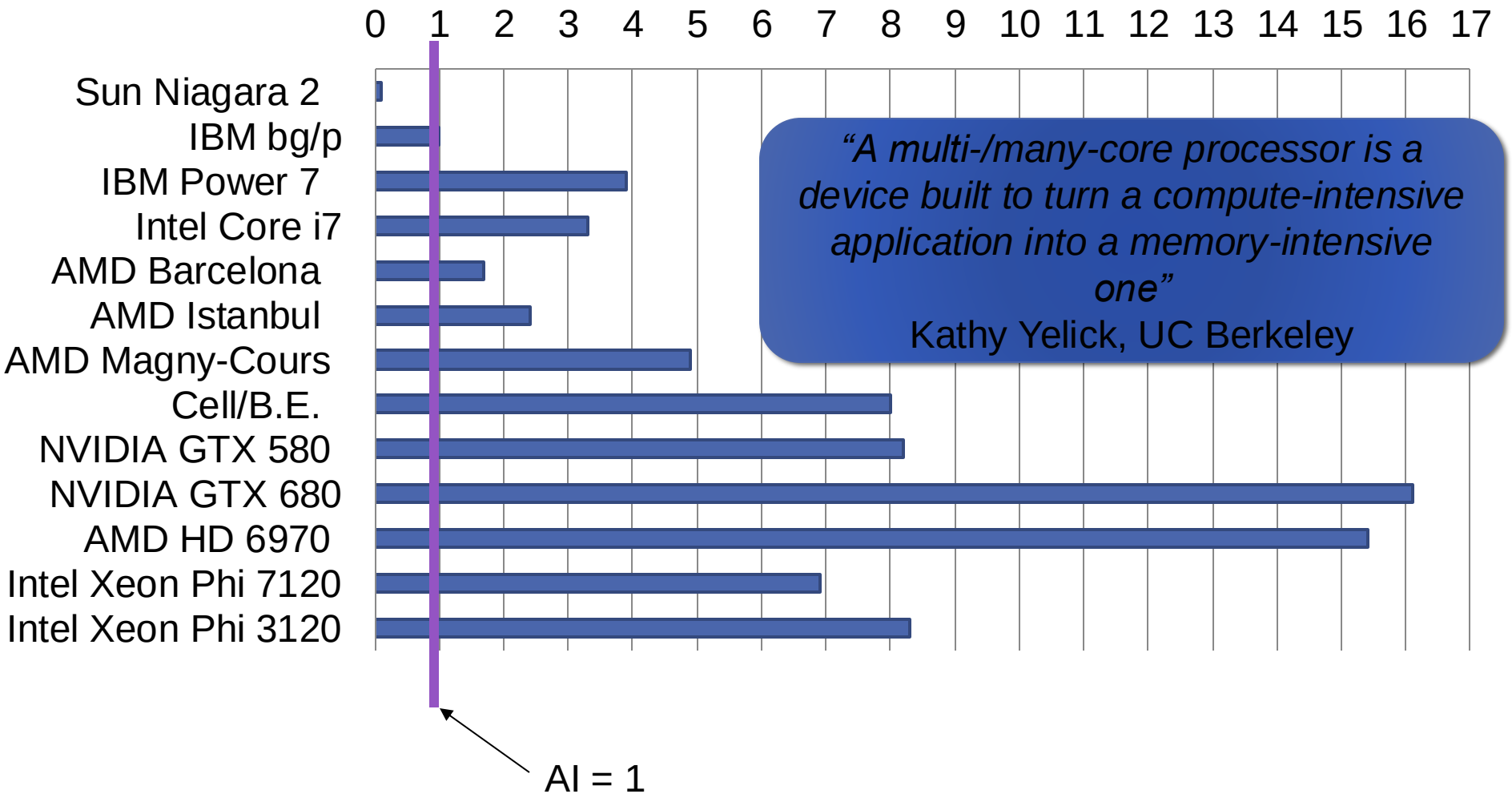
Application arithmetic (operational) intensity

- The number of arithmetic (floating point) operations per byte of memory that is accessed
  - Compute-intensive?
    - i.e., adding more compute power increases performance.
  - Memory-intensive?
    - i.e., adding more memory bandwidth increases performance.

- It is an application characteristic!

- To measure:
  - Count useful operations
  - Ignore “overheads”
    - Loop counters
    - Array index calculations
    - Branches

Compute or memory intensive?



"A multi-/many-core processor is a device built to turn a compute-intensive application into a memory-intensive one"
Kathy Yelick, UC Berkeley

Attainable performance

- Attainable GFlops/sec

  = min(Peak Floating-Point Performance,

  Peak Memory Bandwidth * Arithmetic Intensity)

Compute intensive

Memory intensive

- Peak iff AI_app ≥ PeakFLOPs / PeakBW
  - Compute-intensive iff AI_app ≥ (FLOPs/Byte)platform
  - Memory-intensive iff AI_app < (FLOPs/Byte)platform

Attainable performance (cont'd)

- Typical case: application A runs on platform X in Texec_A :

```
PeakCompute(X) = maxFLOP GLOPS/s          (catalogue)
PeakBW(X) = maxBW GB/s                    (catalogue)
RooflineCompute(A,X) = min(AI(A)*maxBW, maxFLOP)  (model)
AchievedCompute(A,X) = FLOPs(A)/Texec_A     (real execution)
AchievedBW(A,X) = MemOPs(A)/Texec_A       (real execution)
UtilizationCompute = AchievedCompute(A,X)/PeakCompute(X) < 1
UtilizationBW = AchievedBW(A,X)/PeakBW(X) ? 1
```

- Rules of thumb:

$$Peak_{Compute} >= Roofline > Achieved_{Compute}$$

$$Peak_{BW} \ ? \ Achieved_{BW}$$

Note: AchievedBW > PeakBW <=> faster memories on the chip play a role.

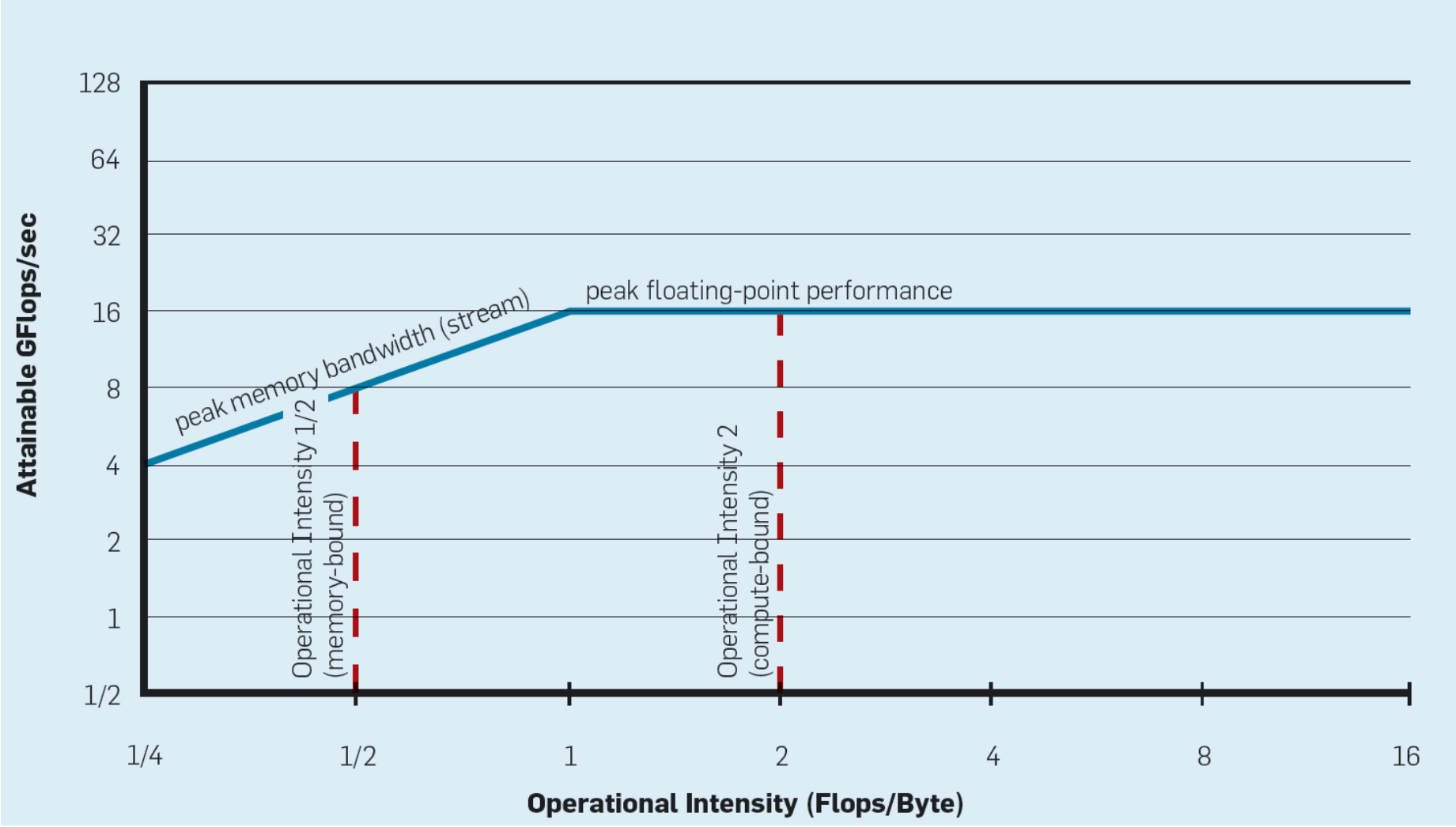
The Roofline model

- Takes the application into account
  - Via Operational intensity
- Takes the platform into account
  - Via hardware specifications

- Determines the performance bounds of an application when executed on different processors.

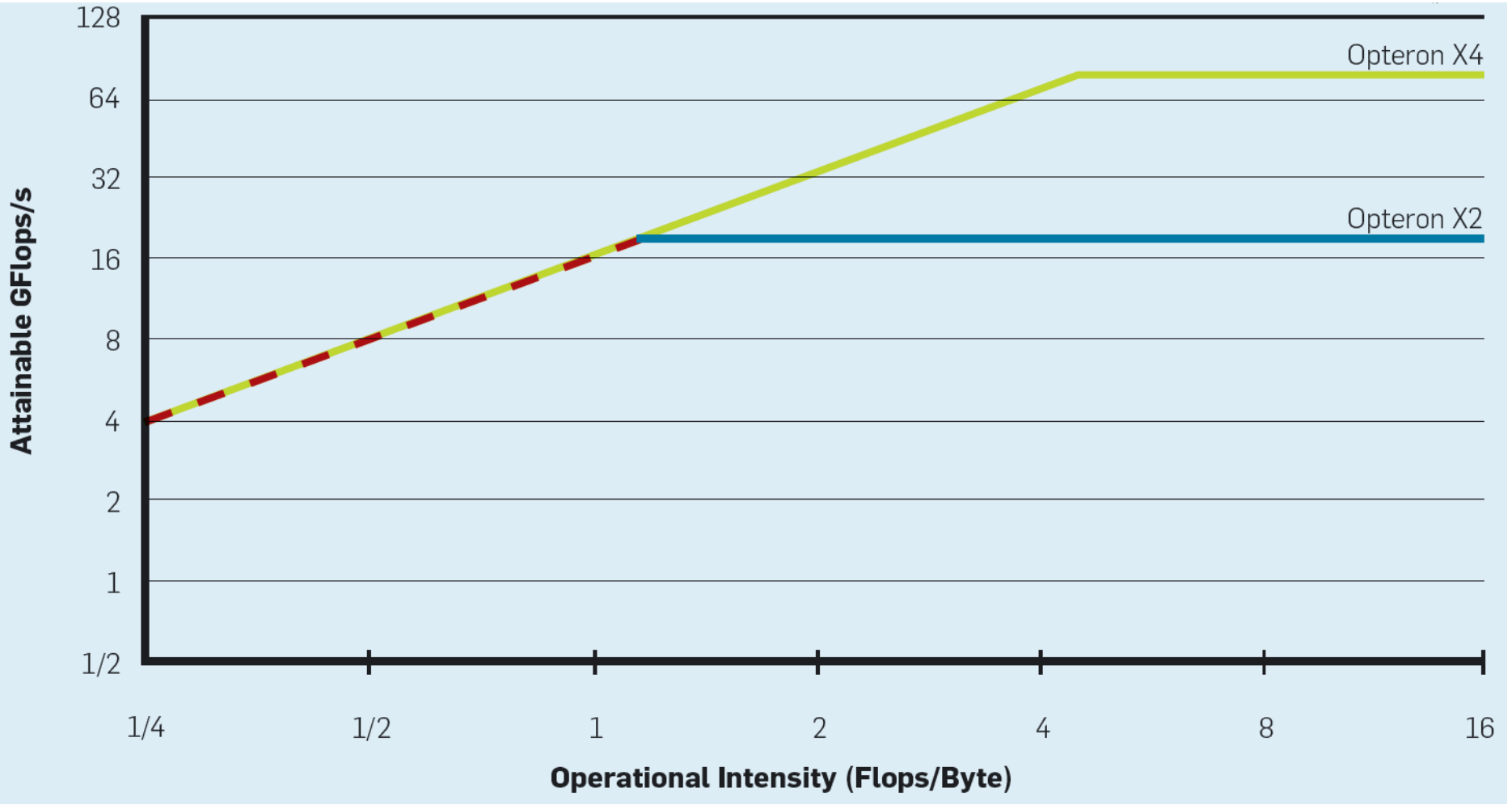
- Hints to optimization strategies.

The Roofline model



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

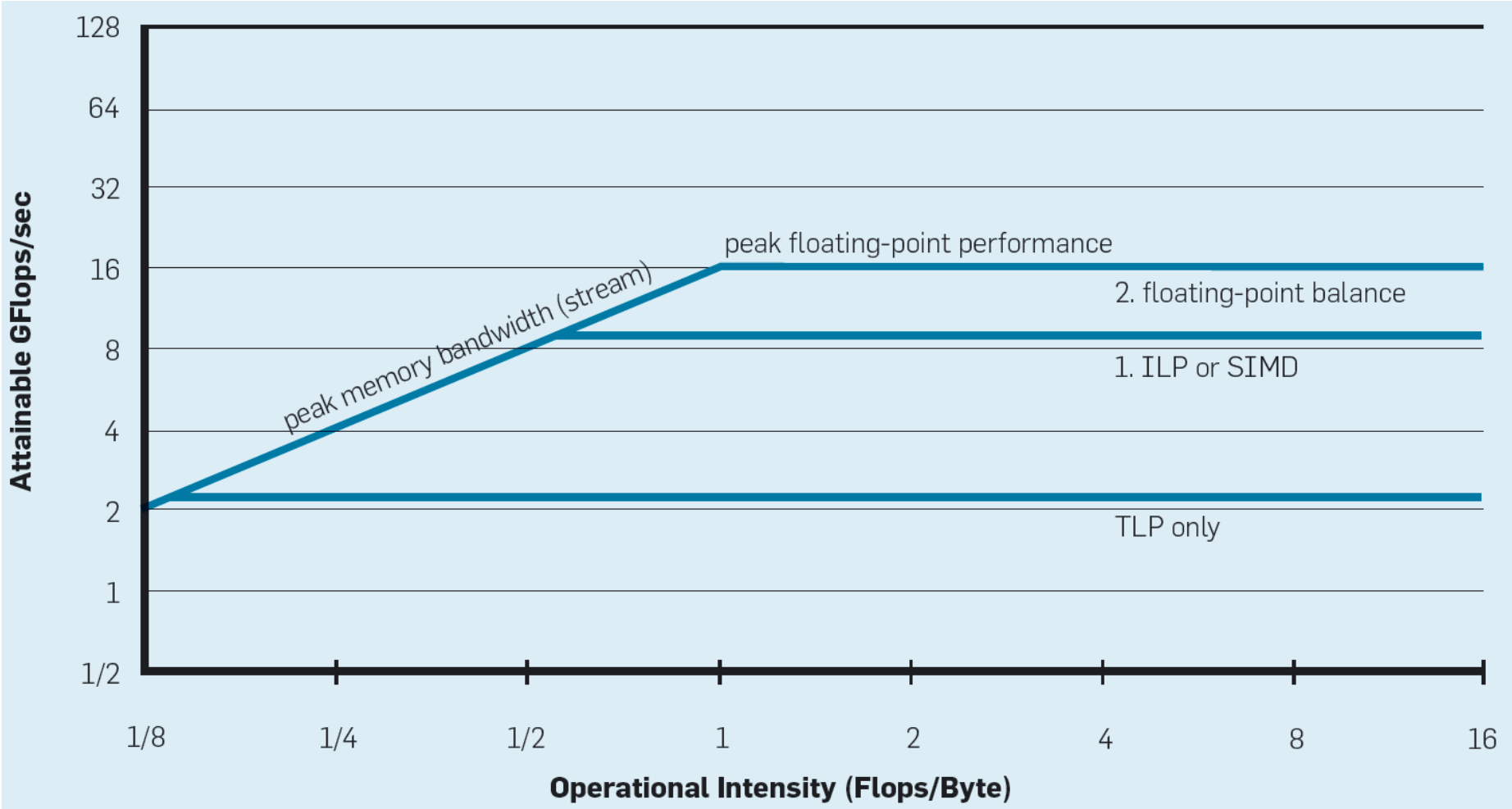
Roofline: comparing architectures



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

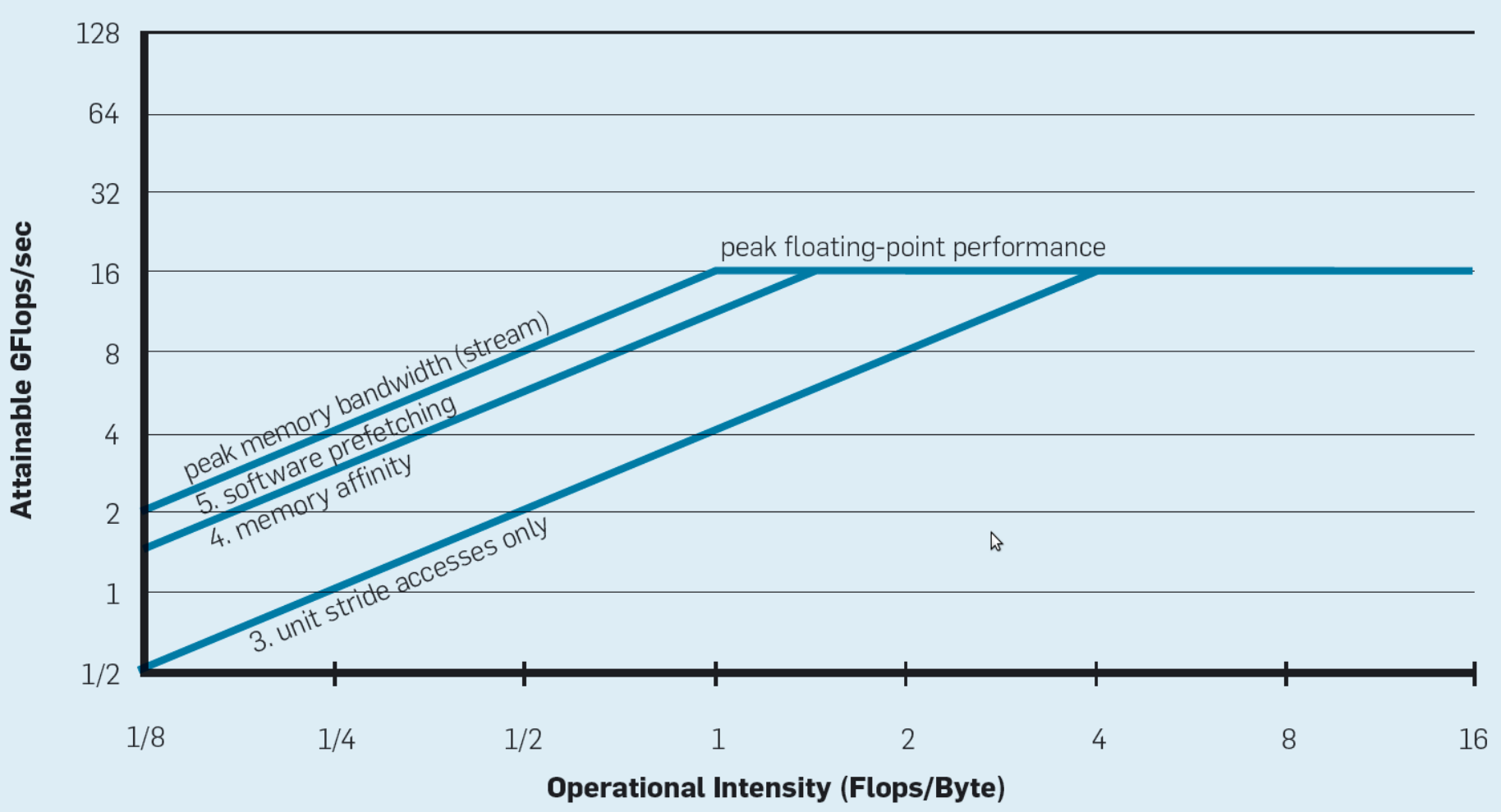
AMD Opteron X4: 73.6 gflops, 15 GB/s, ops/byte = 4.9

Roofline: computational ceilings



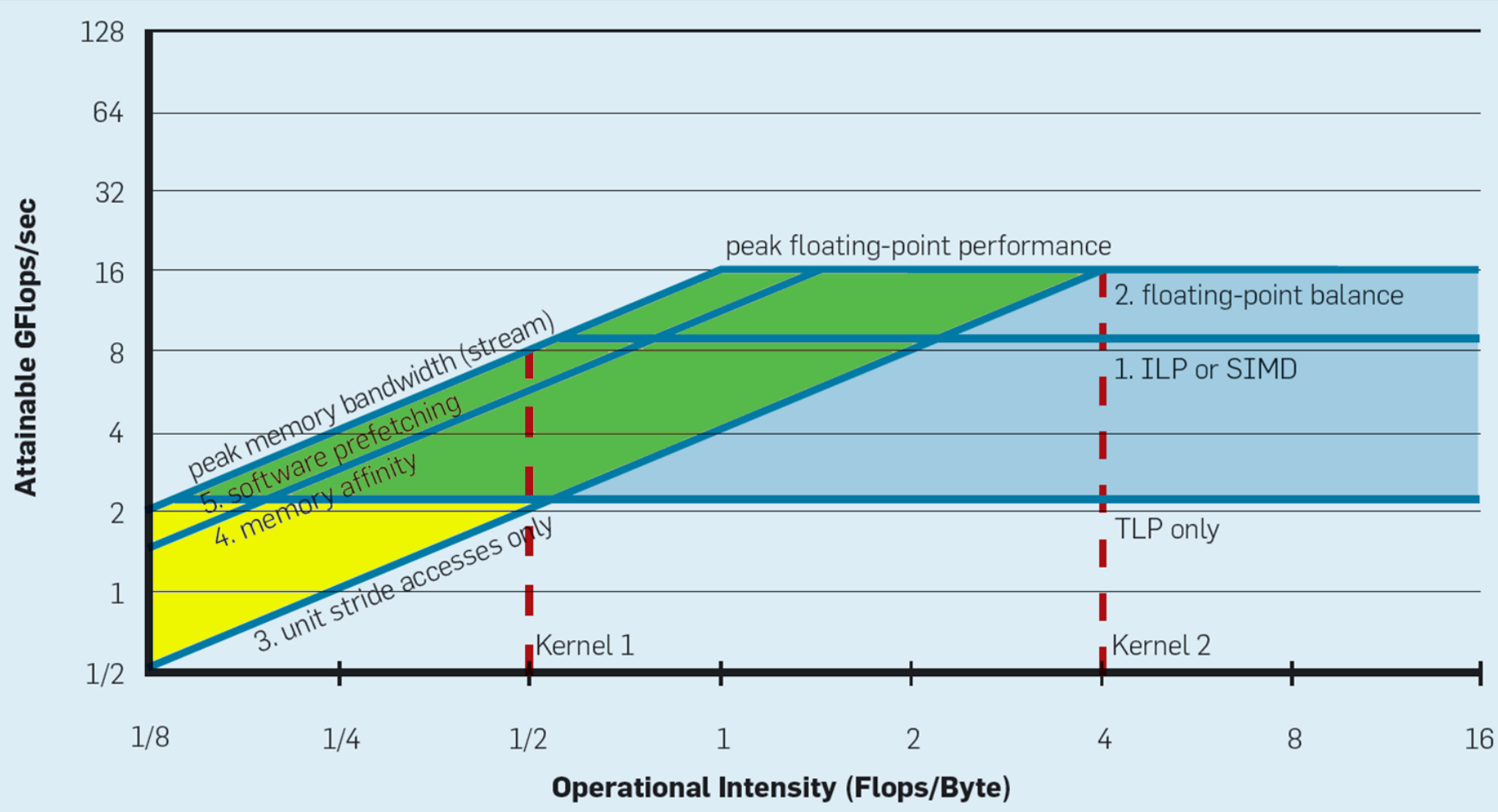
AMD Opteron X2 (two cores): 17.6 gflops, 15 GB/s, ops/byte = 1.17

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
- Showcases limitations in utilization
  - Can indicate upper bounds of energy efficiency as well
- Read:

  Samuel Williams, Andrew Waterman, David Patterson

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

Hands-on: try them out!

Methods and tools

- Understanding/Monitoring the hardware
  - hwloc, scpu
  - cpupower -c 0 frequency-info
  - Nvidia: nvidia-smi (and upgrades), nsight systems
  - Intel: Intel Vtune, various RAPL interfaces
  - AMD: uProf (AMDuProfCLI)
- Performance counters and derivatives
  - PAPI
  - LIKWID
  - Tau
  - … many others
- Larger-scale systems
  - Vampir, Scalasca, Tau, …
  - Various simulators

https://inria.hal.science/hal-04030223/file/_CCGrid23__An_experimental_comparison_of_software_based_power_meters__from_CPU_to_GPU.pdf

In summary

Take ~~home~~ to-the-office message [1]

- Computing demands increasing significantly
  - We build more efficient machines
  - We use them more and more (the rebound effect, Jevon's paradox, …)
- Systems scale-up and scale-out
  - Scale-up: more machinery in the box
    - More complex to use efficiently, more difficult to estimate their performance
    - Require multiple levels of parallelism & heterogeneity
  - Scale-out: increasingly large (possibly distributed) clusters
    - Significant carbon footpring, embodied and operational
    - High utilization is mandatory for efficiency
      - Schedulers and resource managers help
- Efficient applications with understandable performance are fundamental for the efficiency of these systems.

Take ~~home~~ to-the-office message [2]

- Assess what is feasible in terms of performance, efficiency, energy consumption

- To reduce energy consumption
  - Power-off what is not needed
  - “donate” what is not needed

- To increase efficiency
  - Assess feasible performance and aim to reach it
  - Maximize utilization of hardware

- Always **select** algorithms, implementation tools, and platforms

Take ~~home~~ to-the-office message [3]

- Most tool targets …
  - Benchmarking
  - Performance analysis & prediction
  - Upper bounds estimates
  - Bottlenecks
- Emerging tools for sustainability
  - Various LCA models
  - Various carbon footprint models
  - Still rely on very basic utilization models
- We need to work on bridging the gap ☺