PERFORMANCE ENGINEERING

Lecture 1: Introduction

April 4th, 2022

Ana Lucia Varbanescu
a.l.varbanescu@uva.nl
(room C3.109)

To Do

- ~~Course structure~~

- What's in a name?
- Prerequisites
  - To know or to learn …
- Our first performance model

About performance engineering

Performance

- The accomplishment of a **given task** measured in given conditions against preset known **standards** of **accuracy**, **completeness**, **cost**, and/or **speed**.

Why care about performance ?

- As a user …
  - Your application is not responsive
  - Your simulation is not ready in time
  - Your data is not fully processed
  - …
- As a mindful citizen
  - Every time your application is not responding someone else takes a decision to *buy* or *use* more hardware
    - Higher energy consumption
    - Lower efficiency

Performance engineering provides a better answer than more hardware! Instead, let's make better use of the resources you already have!

Why is performance challenging?

- It is a *nonfunctional* requirement
  - Like efficiency, scalability, …
- There are a lot of myths around it
  - It’s someone else’s problem
  - It’s just a matter of money
    - More hardware
    - More people
    - More time
  - It’s really easy to fix later
  - It’s “just engineering”
- There’s little glory or fame in it
- **And it’s really really hard to do!**

Performance engineering

Software Performance Engineering (SPE) is a **systematic**, **quantitative** approach to the cost-effective development of software systems to **meet performance requirements**.

- SPE is a software-oriented approach that focuses on **architecture**, **design**, and **implementation** choices.
- SPE gives you the information you need to **build software** that **meets performance requirements** within **budget**.

Performance "engineering" requires multi-disciplinary research and thorough knowledge in multiple fields!

Systematic approach

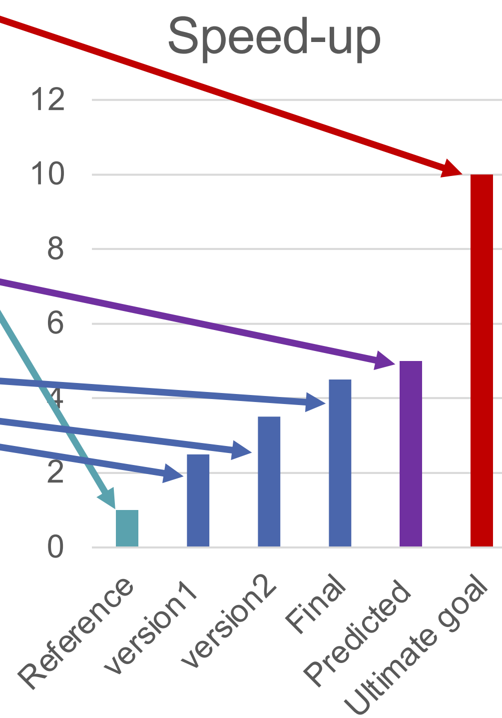
Systematic approach

Given an application …

1. Understand (user) **requirements**
2. Understand **current performance**
3. **Can** it be done?
4. **How** can it be done?
5. **Tuning**

    Not there yet? => back to 2

6. Analyze & document the result



Running example

- Calculate the head dissipation (2D stencil operation, iterative) in a metal cylinder.

1. Requirements

- Collect and process performance requirements

- Examples:
  - Real-time performance
  - Best possible performance
  - *N* times faster than reference implementation
  - *X* % or more hardware utilization
  - Linear scaling
  - …

Calculate heat dissipation for a 10K x 10K cylinder in 1ms.
15 ops per point x $(10K)^2$ = 1.5GFLOP => Throughput = 1.5 / 0.001 = 1500 GFLOPS

2. Current performance

- Analyze current performance
  - Decide on metrics (latency, throughput, efficiency, …)
    - Ideally related to requirements!!!
  - Collect/infer relevant use-**scenarios**
    - Input data included !
  - Per **scenario**:
    - Profile the code => **identify hot-spots**
    - Measure performance in detail **=> identify bottlenecks**

Current speed for a 10K x 10K cylinder is 1s => We need 1000x improvement!

3. Can it be done?

- Feasibility analysis based on modeling.
  - Model the performance to …
    - Determine best-case/worst-case performance
    - Determine theoretical lower/upper performance bounds
    - Determine scalability
    - …
  - If not feasible => revise requirements

Maybe 1500 GFLOPs is too much?

CPU peak performance = 100 GFLOPs << 1500 GFLOPs => CPU not feasible!
GPU peak performance = 2000 GFLOPs > 1500 GFLOPs => GPU feasible … with 75% utilization! => maybe …

4. How can it be done?

- Select the methods and tools for tuning.
  - Identify feasible actions
    - Better hardware/OS/…
    - Tuning – parameters, compiler-options, etc.
    - Implementation – better constructs, more efficient data structures
    - Restructuring / refactoring – better algorithms, methods, parallelization, …
  - Rank options in terms of gain **using performance models**
  - Select the best ones
    - Always handle the performance bottleneck first
    - Assess/rank gain, effort, …

Key challenge: accurate models!!

Code optimization: Apply SIMD => 2x , Improve caching => 1.5x, …
Different algorithms: Handle boundary conditions, aggressive recalculation, …
Better hardware: Use a GPU! => 20x

5. Tuning

- Implement the selected tuning methods
  - Apply one action at a time
  - Re-evaluate after each step
    - Performance => “update” models
    - Tuning steps => “update” plan
  - Are you there yet? If not: continue.

Mostly implementation, benchmarking, and model refinement.
Reorder optimizations depending on current results.

6. Analyze the result

- Document design options & choices
- Document models and benchmarks
- Reflect on the results
  - Cost, effort, sustainability
- Document future steps, and their requirements
  - Cost, effort

Selected algorithm A because … => see model!
Used a GPU because the CPU was not feasible => see model!
Further implementation to make … => see model!

Systematic approach

Project

1. Understand requirements
2. Understand current performance
3. Can it be done? (modeling)
4. How can it be done? (some options)
5. Tuning

   Not there yet? => back to 2
6. Analyze the result

Lectures

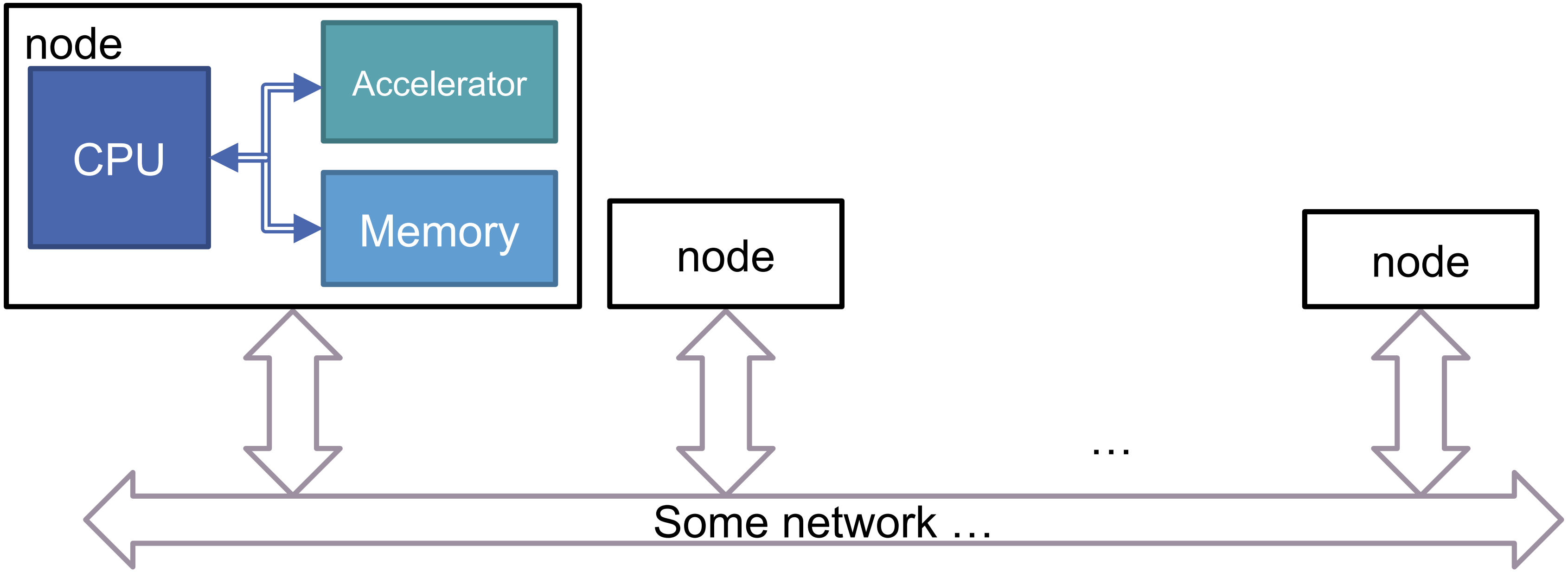
Programming for performance ~~is~~ can be **ugly**.

- Data-structures are kept simple for memory access patterns changes
- If statements are replaced with various constructs
  - Conditional assignment
  - Additional computation
- Trade-offs are made between compute and store
- Best algorithms may be (a bit) forgotten
  - More operations can be better for parallel systems

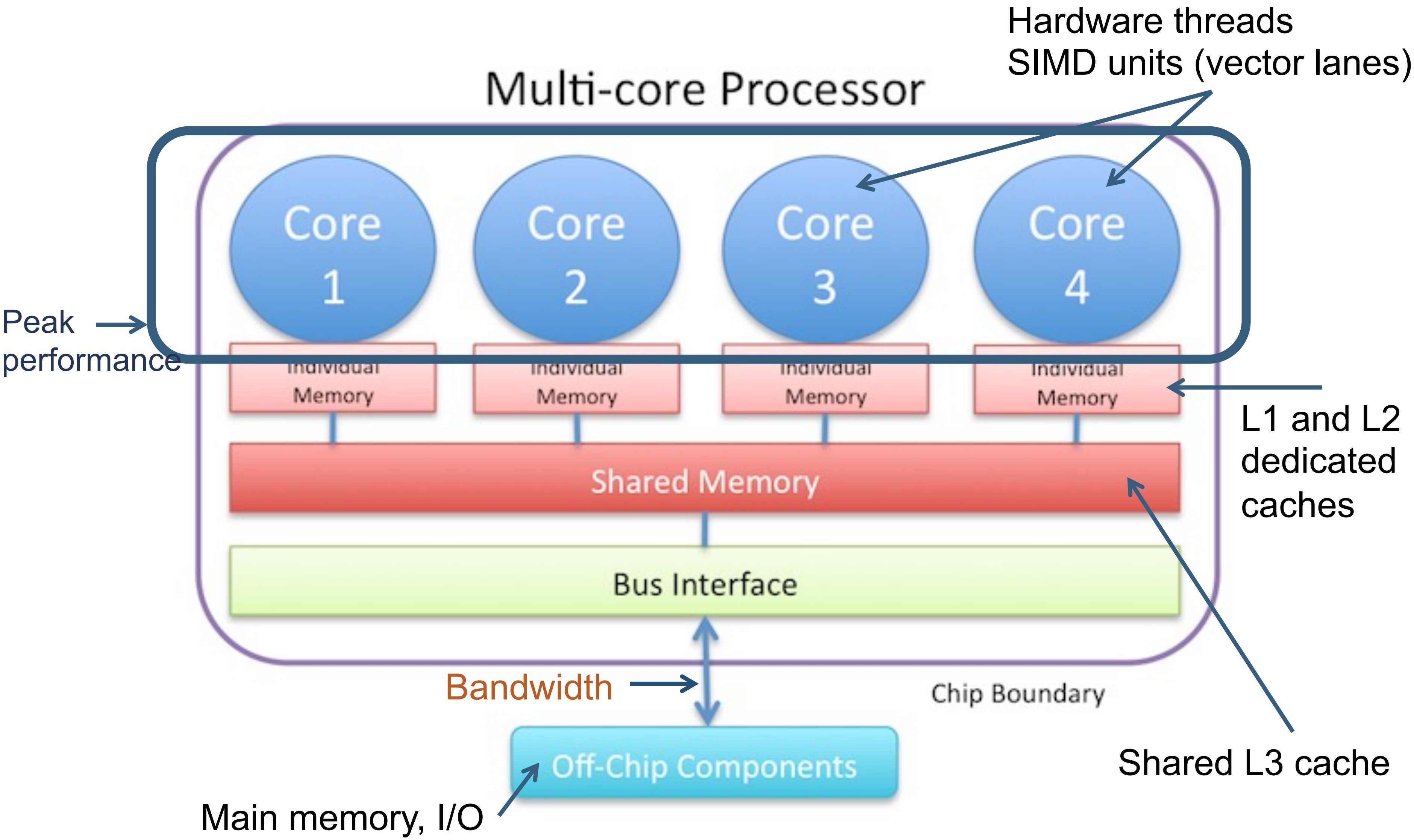
Prerequisites: computer systems

Generic view

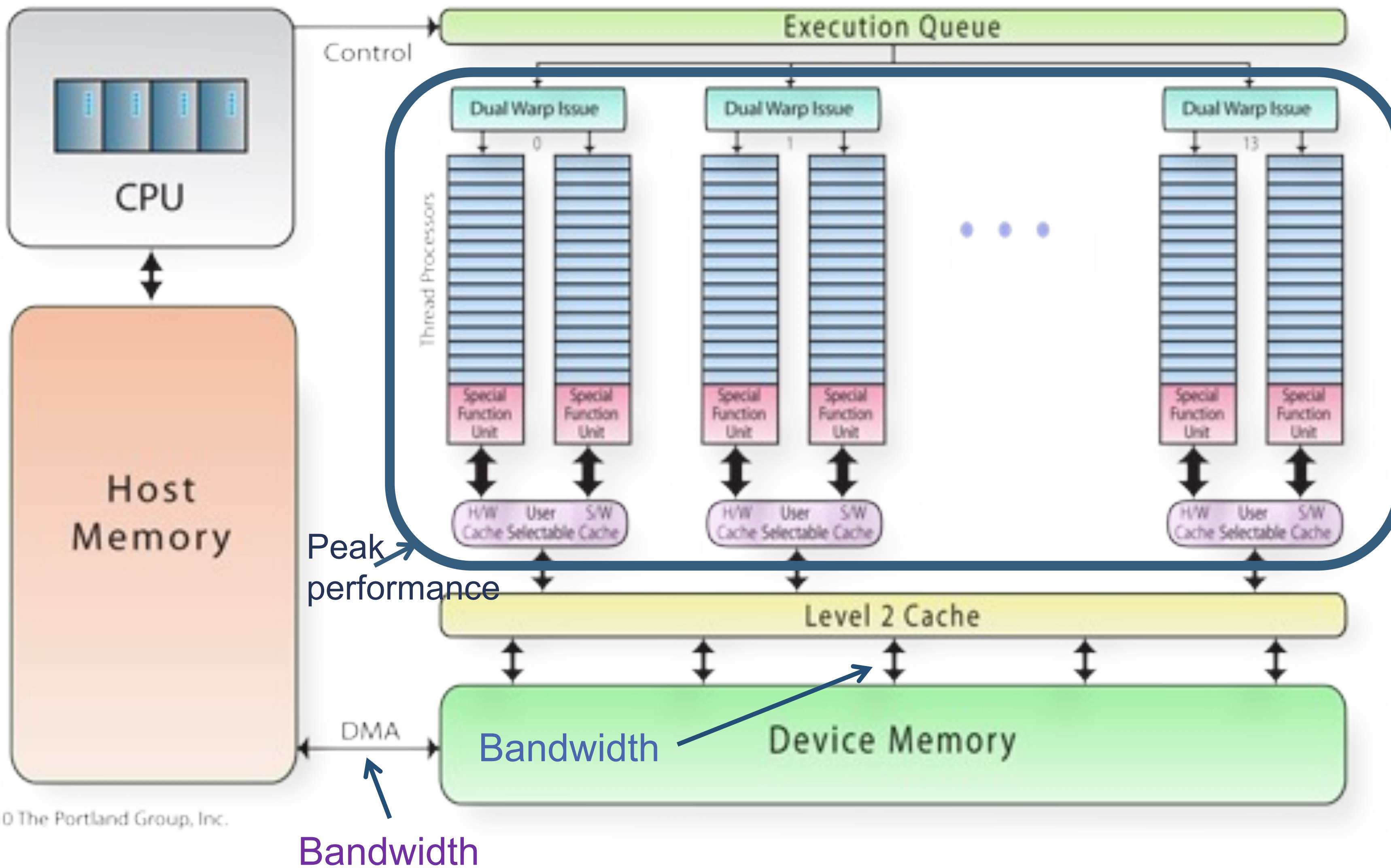
- Heterogeneous, parallel & distributed systems
  - Single-node performance
  - Aggregate performance



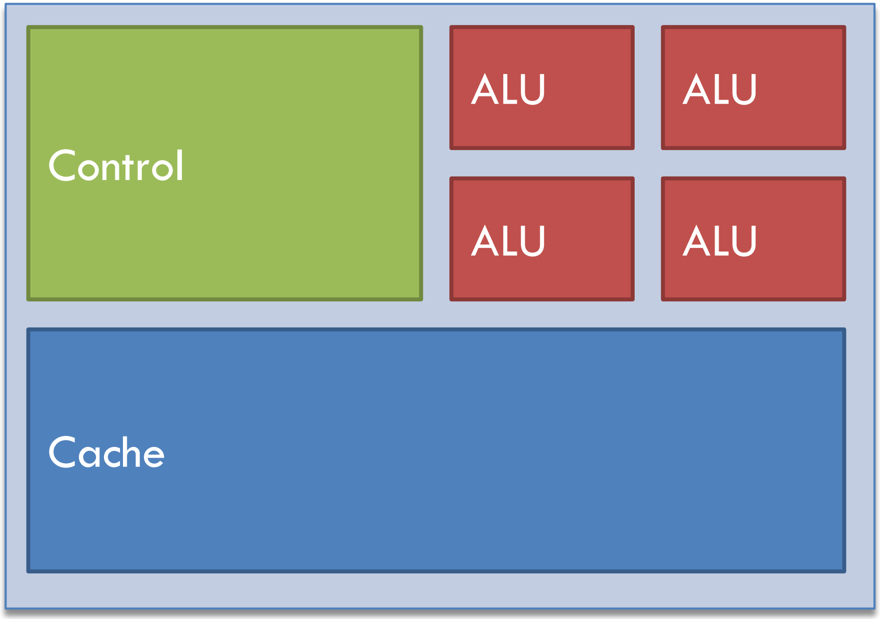
Generic multi-core CPU



Generic GPU


CPU
Control
Execution Queue
Dual Warp Issue
Dual Warp Issue
Dual Warp Issue
0
1
13
Thread Processors
Special Function Unit
H/W Cache
User Selectable
S/W Cache
Host Memory
Peak performance
Level 2 Cache
DMA
Bandwidth
Device Memory
Bandwidth

CPU vs. Accelerator (GPU)

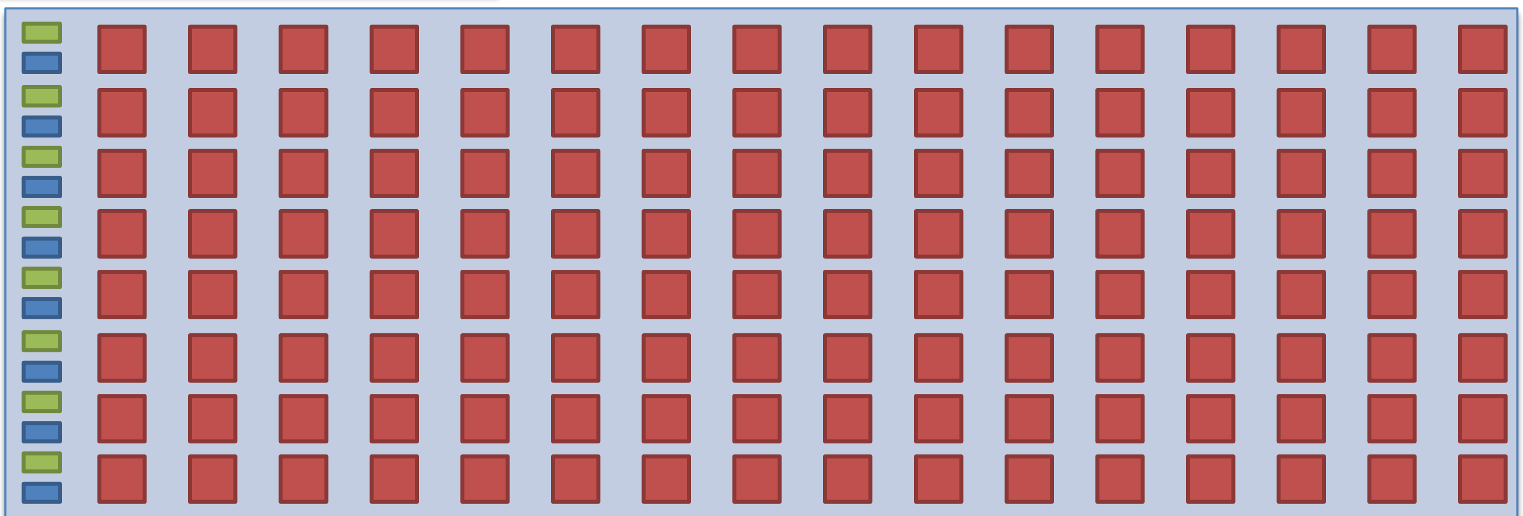


CPU

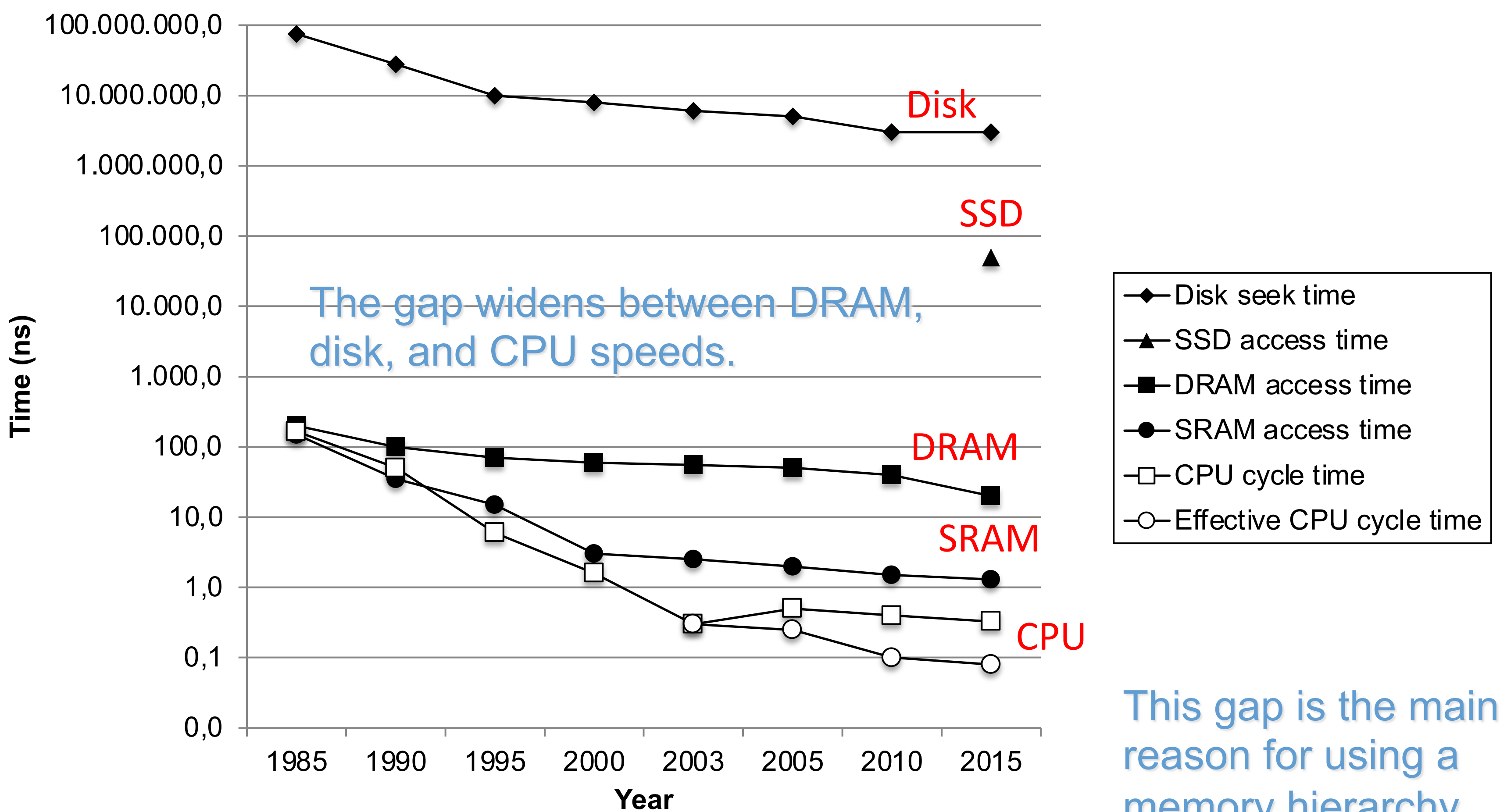
Low latency, high flexibility.
Excellent for irregular codes with limited parallelism.

GPU

High throughput.
Excellent for massively parallel workloads



The *PU-Memory Gap



This gap is the main reason for using a memory hierarchy.

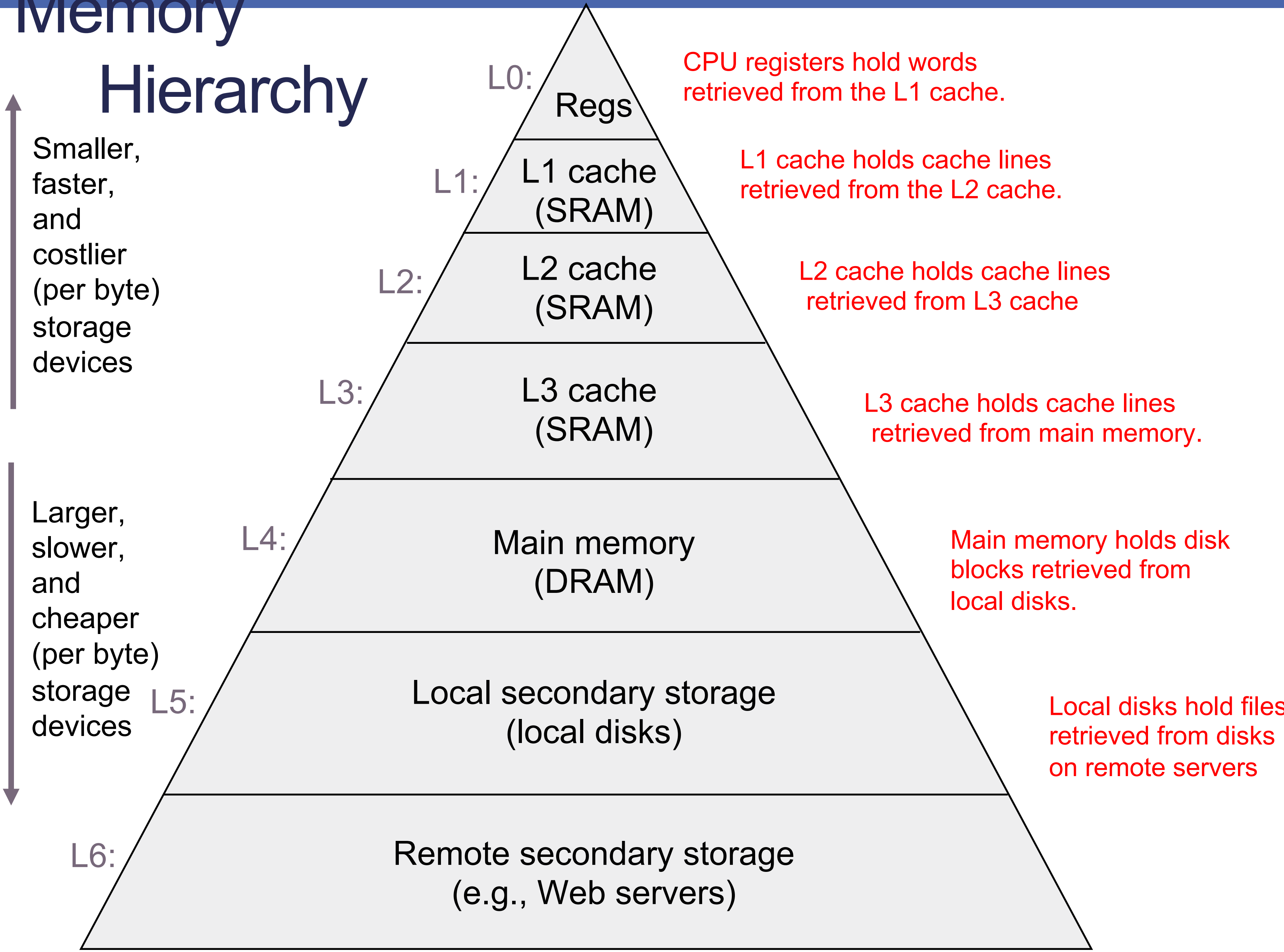
Caches

- **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



Prerequisites: performance metrics

(Types of) Performance metrics

- Latency/delay
  - The time for one operation (instruction) to finish, L
  - To improve: minimize L
    - Lower is better
  - Examples ?
- Throughput
  - The number of operations (instructions) per time unit, T
  - To improve: maximize T
    - Higher is better
    - Thus, time per instruction decreases, on average
  - Examples?

Hardware performance

Hardware 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 [GFLOPs] = chips * cores * SIMD_Width * FLOPs/cycle * clockFrequency

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

	Cores	Threads/ALUs	**GFLOPS**	**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**

Application performance

About performance

- Measure performance
  - Observe the performance
    - For every (application, machine, dataset) instance.
- Evaluate & analyze performance
  - Reason about performance causes and limitations
  - Application-centric : understand more about the application => understand more about its performance
- Model & predict performance
  - Predict further performance behavior
    - For different configurations
    - For different machines
    - For different applications

Measured metric: execution time

- Applications have “stages”
  - Sequential stage
  - Parallel stage
  - Communication stage
- Wall-clock time: the time it takes the full application to execute
  - Essential for end-users
- Stage execution times
  - Essential for detailed analysis

- Alternative metrics
  - Number of cycles (counter)
  - Number of executed instructions (counter)
  - IPC = Instructions per Cycle (computed)
  - CPI = Cycle per instruction (computed)

Derived metrics

- Speed-up: $$S(p) = \frac{T(1)}{T(p)}$$
  - T(1) = sequential execution
    - Ideally, the best known one
    - In practice, avoid as many overheads as possible
- Efficiency: $$E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \times T(p)}$$

Any application information here?!

Execution time
MUST include all machine specification
Cycles
SHOULD include machine (micro)architecture specification
Speed-up and Efficiency
MUST use the best possible sequential execution time

More derived metrics

- Take application into account?

- Achieved (comp.) throughput: App_GFLOPs = #FLOPs / T
  - **Compute efficiency**: Ec = App_GFLOPs/peak * 100

- Achieved bandwidth: App_BW = #(RD+WR) [bytes] / T
  - **Bandwidth efficiency**: Ebw = App_BW/peak * 100

- Achieved bandwidth and throughput can be used to compare *different* algorithms.
- Efficiency can be used to compare *different* (application, platform) combinations.

Quiz – 10p, 7 min

- Calculate:
  - Achieved throughput in GFLOPs, running in 1s
  - Achieved bandwidth for this loop
  - Compute efficiency: 16-core 2-way SIMD CPU, 256GFLOPs.
  - Bandwidth efficiency: 100 GB/s

```
struct complex {float re; float im;};
int sum_array_3d(complex a[N][N][N])
{
    int i, j, k,
    float partial[N], sum = 0;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++) {
          partial[i] += a[k][i][j].re
          sum += a[k][i][j].re;
        }
    return sum;
}
```

Are these metrics enough?

- Goal: understand achieved vs. theoretical performance
  - Understand bottlenecks
  - Perform correct optimizations
  - … decide when to stop fiddling with code!!!

- Realistic theoretical limits*
  - Use theoretical peak limits => low accuracy
  - Use **application characteristics**
  - Use **platform characteristics**

*unsolved challenge in parallel performance analysis

Arithmetic/Operational intensity

- The number of arithmetic (floating point) operations per byte of memory that is accessed

- It's an application characteristic!

- Ignore "overheads"
  - Loop counters
  - Array index calculations
  - Branches

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

- Attainable GFlops/sec

  = min(Peak Floating-Point Performance,

  Peak Memory Bandwidth * Arithmetic Intensity)

Compute intensive

Memory intensive

- Example: RGB-to-Gray
  - AI = 1.25
  - NVIDIA GTX680
    - P = min (3090, 1.25 * 192) = 240 GFLOPs
      - Only **7.8%** of the peak
  - Intel MIC = Intel Xeon Phi
    - P = min (2417, 1.25 * 352) = 440 GFLOPs
      - Only **18.2%** of the peak

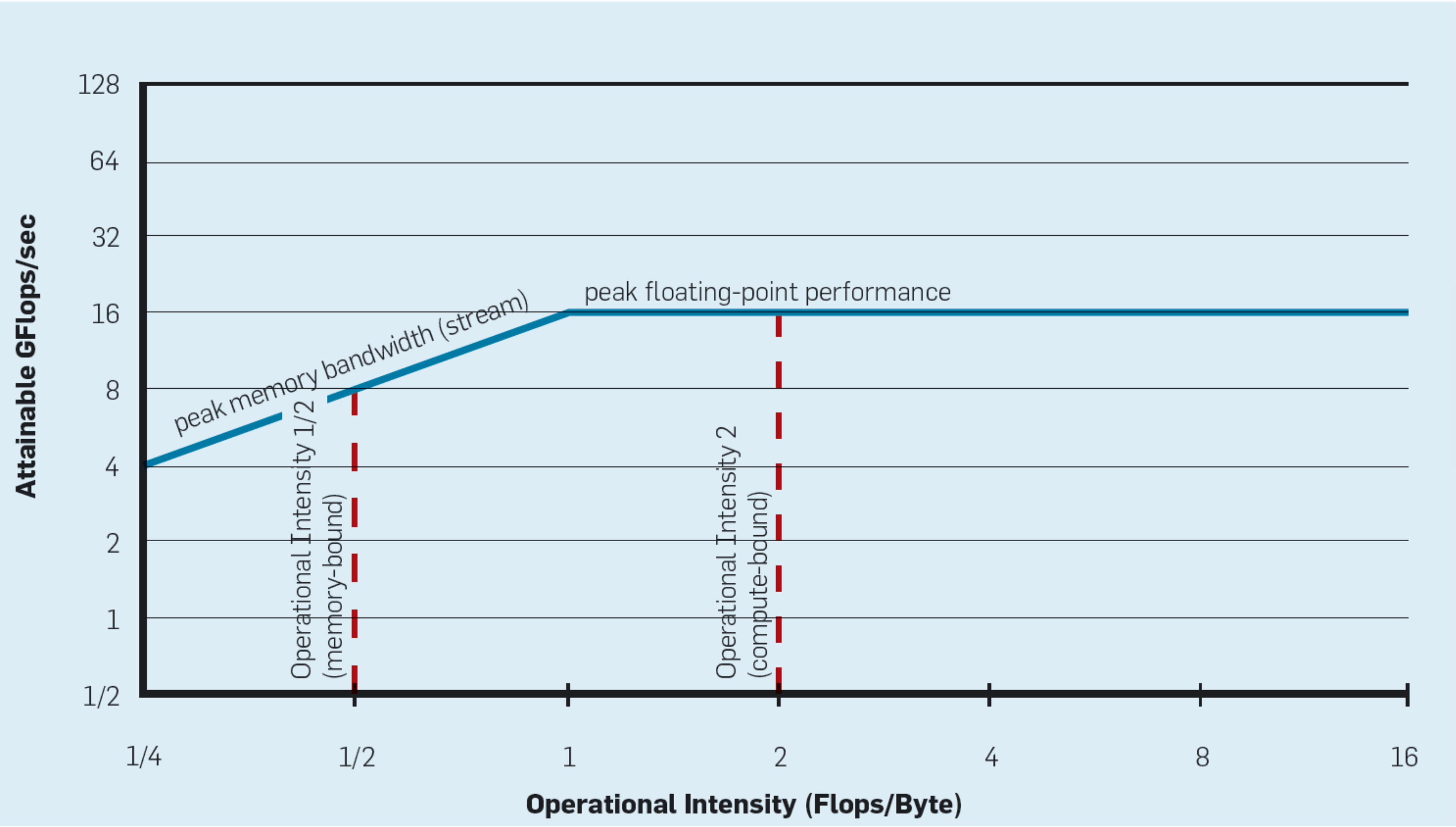
The Roofline model (**one** example!)

- 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

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”