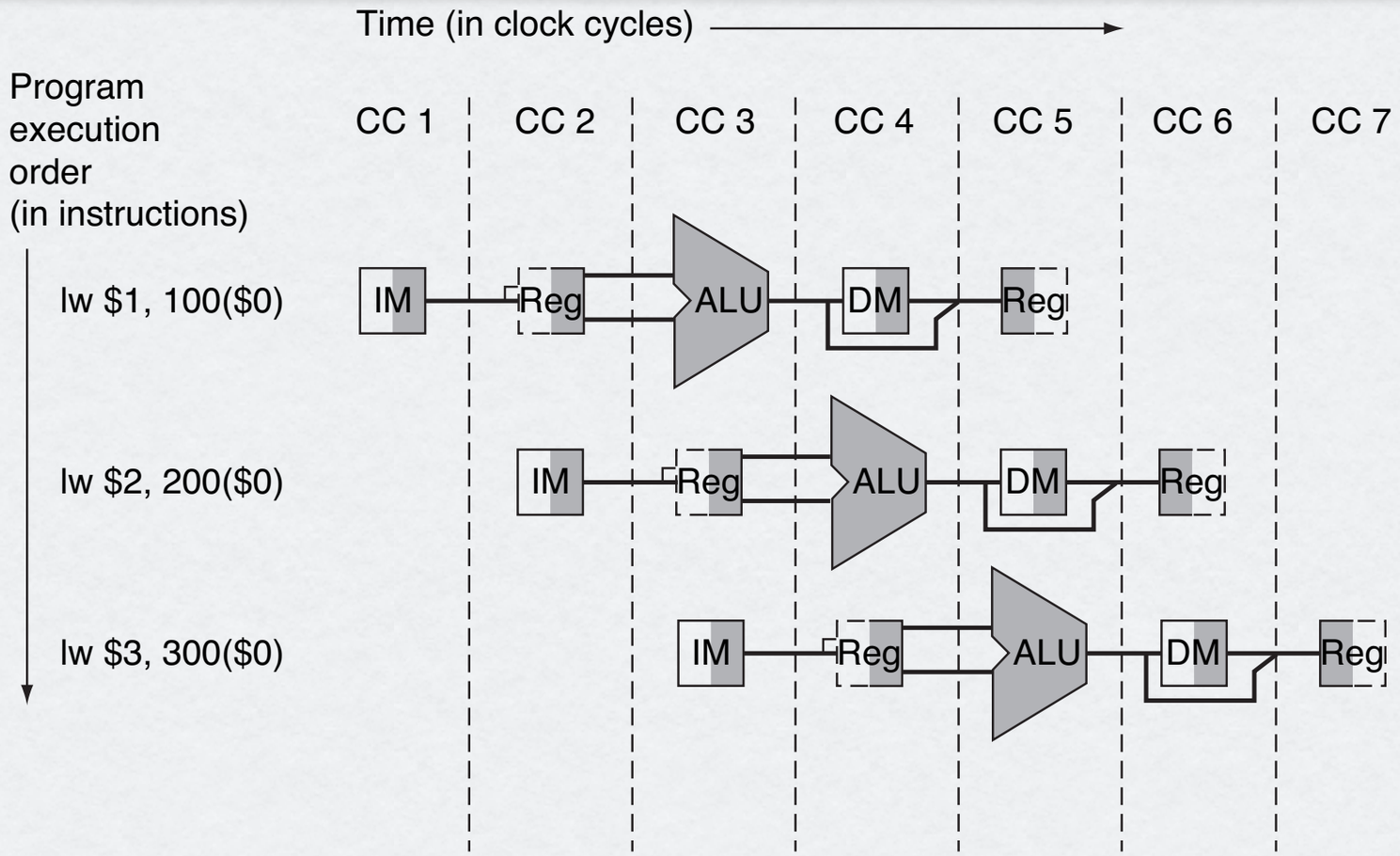
Instructions being executed

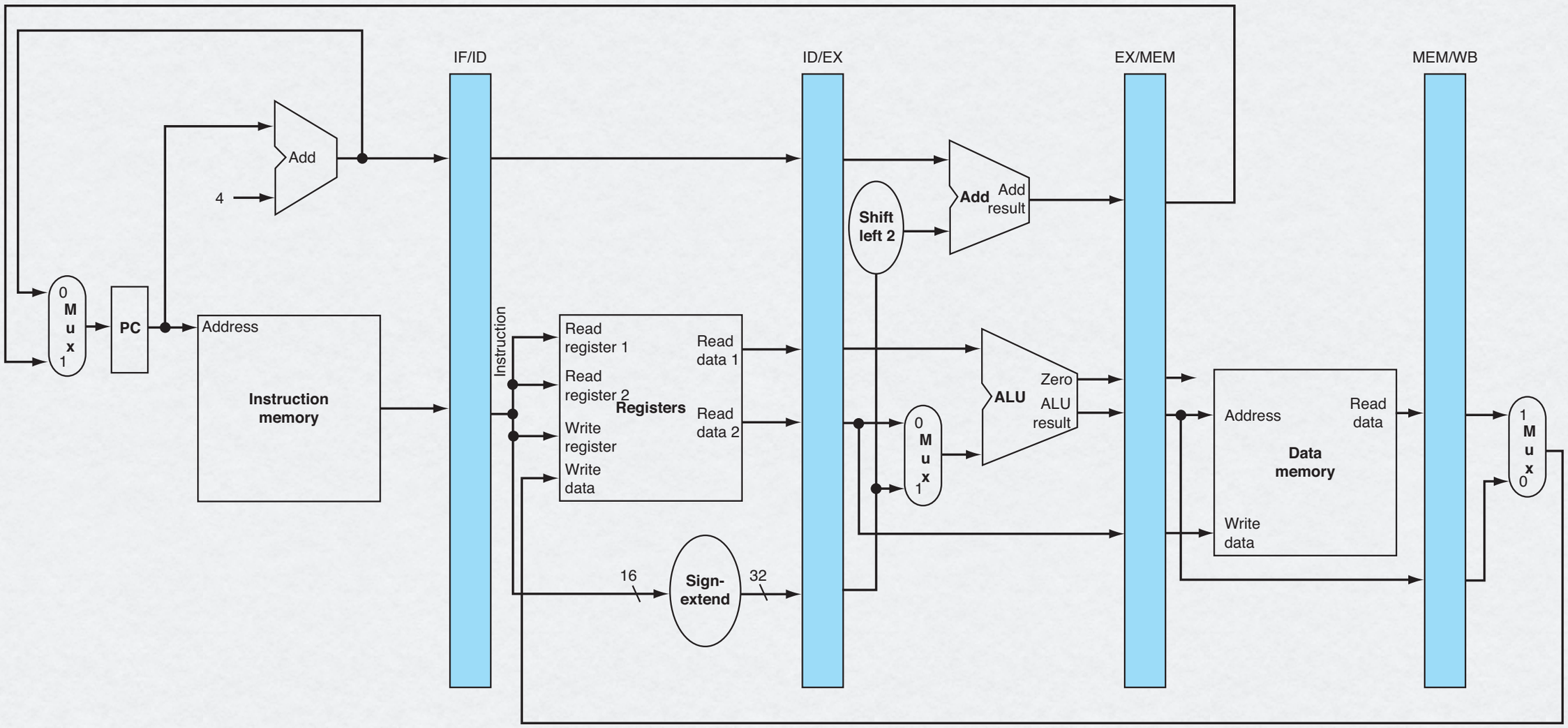


This figure pretends that each instruction has its own datapath, and shades each portion according to use.

Each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.33. *IM* represents the instruction memory and the PC in the instruction fetch stage, *Reg* stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on.

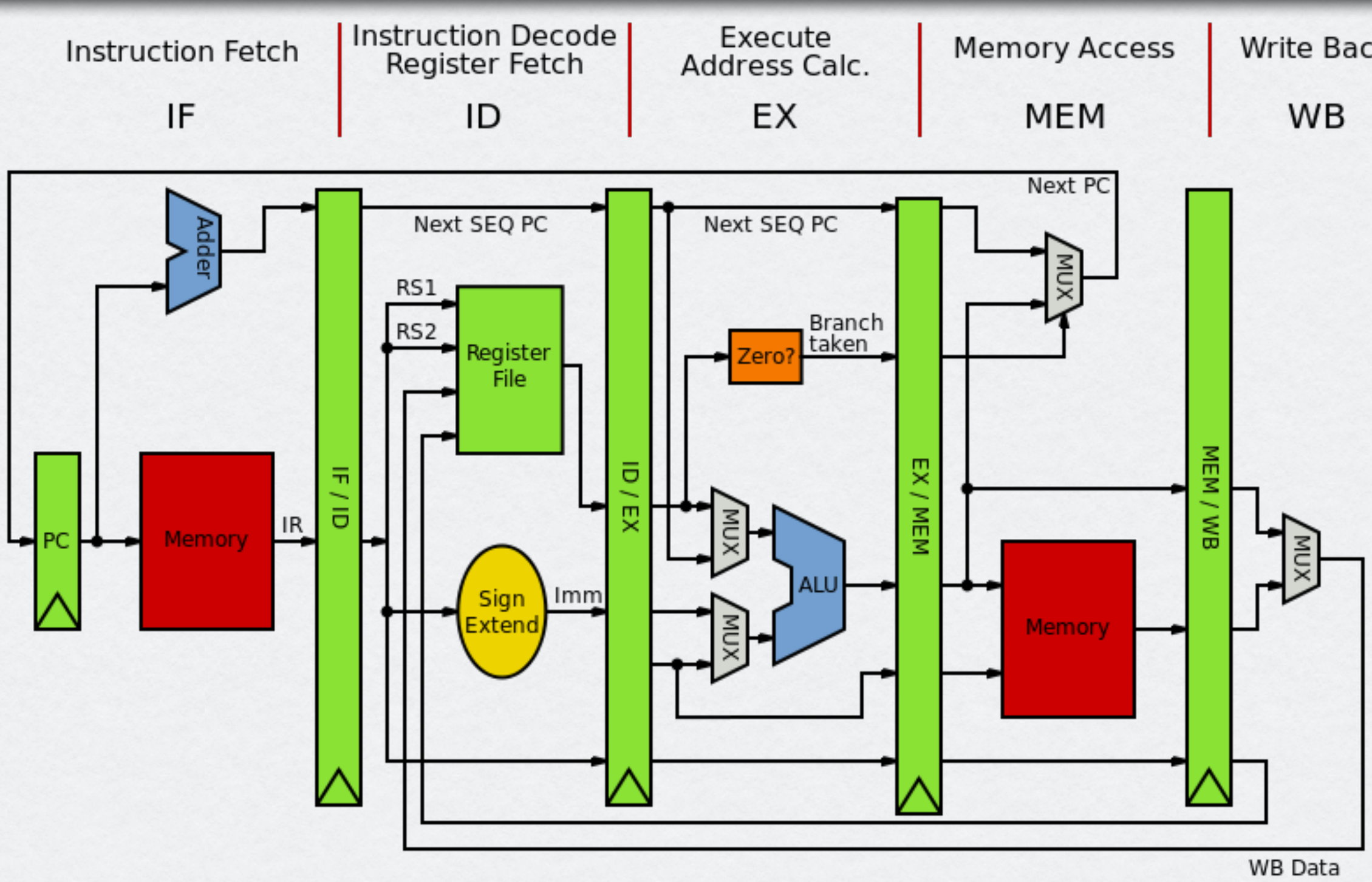
To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

Pipeline



The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages.

MIPS Architecture Pipeline



The stage-by-stage architecture of a MIPS microprocessor with a pipeline. Although the memory is shown twice for clarity of the pipeline, MIPS architectures have only one memory bank (i.e. von Neumann architecture)

Pipelined MIPS, showing the five stages (instruction fetch, instruction decode, execute, memory access and write back).

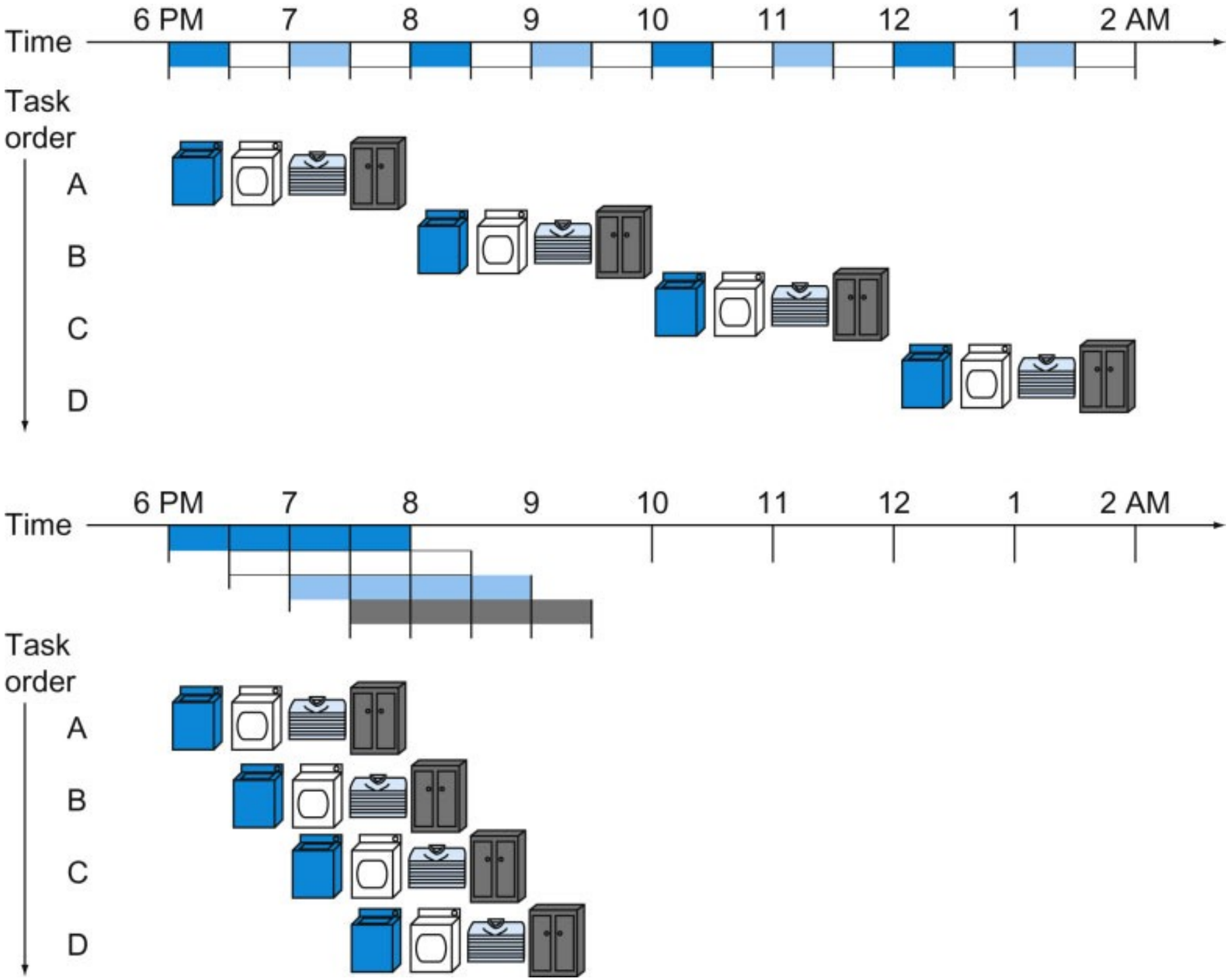
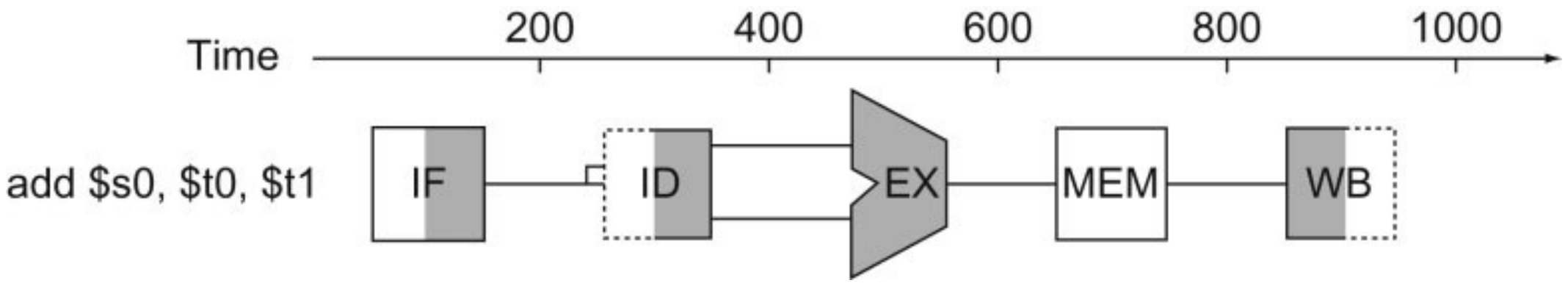


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, “folder,” and “storer” each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.

FIGURE 4.28 (*)
Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.25.



Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter.

The symbols for the five stages:

IF **for the instruction fetch stage**, with the box representing instruction memory;

ID **for the instruction decode/register file read stage**, with the drawing showing the register file being read;

EX **for the execution stage**, with the drawing representing the ALU;

MEM **for the memory access stage**, with the box representing data memory; and

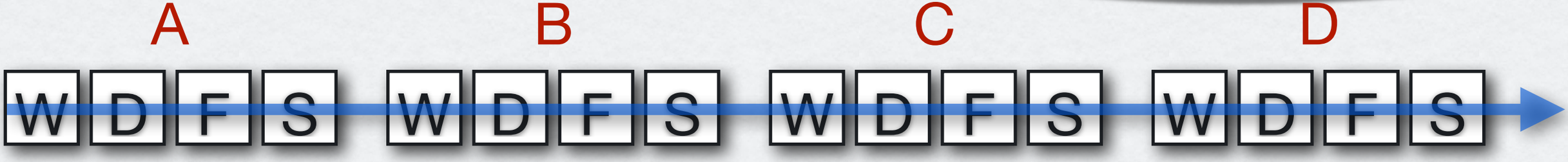
WB **for the write-back stage**, with the drawing showing the register file being written.

The shading indicates the element is used by the instruction. **Hence, MEM has a white background because add does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage.** Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

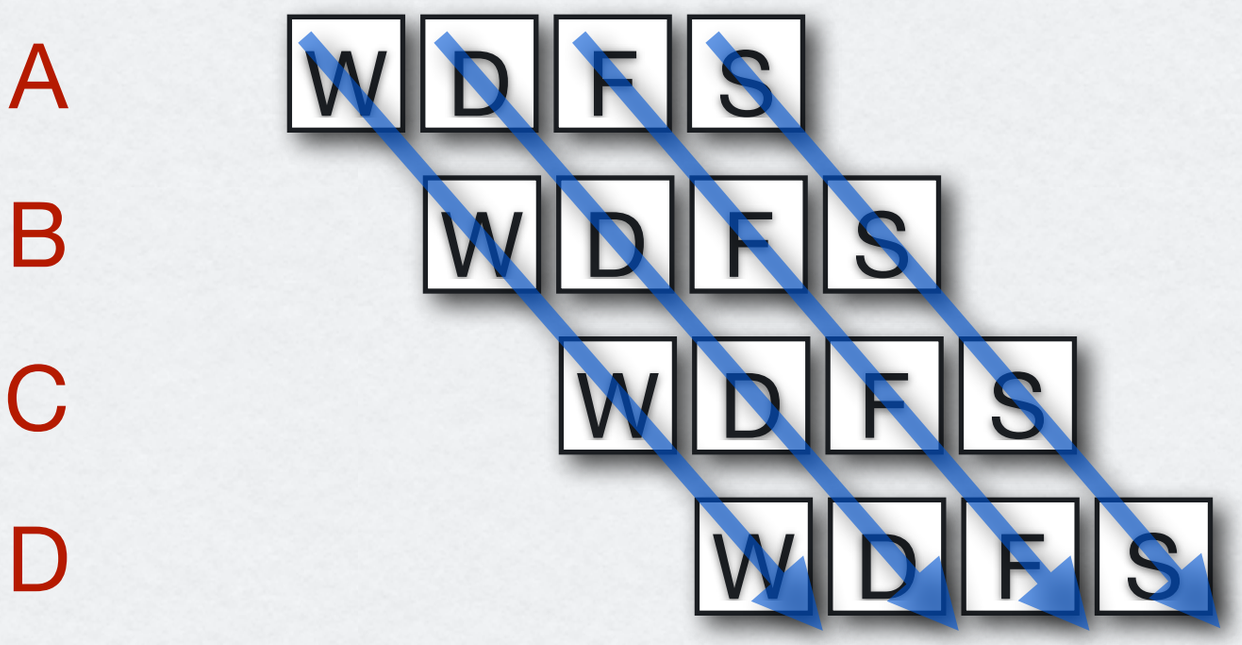
Pipelining - The Laundry

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution
- **The laundry example: A, B, C, D** have laundry and must **W**ash, **D**ry, **F**old, **S**tore.
  Each operation takes 0.5 hrs (same for all)

Sequential: 8 hrs total



Alternative:



Total – 3.5 hrs

Pipelining

- Pipelining improves **throughput.** It does not decrease the time to do one load of laundry, but it decreases the time when we have many loads of laundry

- Note loss of time at the beginning and end

- If the stages take about the same time and there is enough work to do then the speed-up due to the pipelining is equal to the number of stages in the pipeline (in this case 4)

Pipelining

- The same principles apply to the processors where we pipeline instruction executions

- MIPS instruction takes 5 steps:

1. **Fetch** instruction from memory
2. **Read** registers while decoding the instruction (simultaneous)
3. **Execute** an operation or calculate an address
4. **Access** on operand in data memory
5. **Write** the result into a register

Pipelining (example)

Limit to 8 instructions: load word (`lw`), store word (`sw`), add (`add`), substitute (`sub`), AND (`and`), OR (`or`), set less than (`slt`), branch on equal (`beq`)

Instruction class	Instruction fetch (ps)	Register read (ps)	ALU compute (ps)	Data access (ps)	Register write (ps)	Total (ps)
lw	200	100	200	200	100	800
sw	200	100	200	200		700
add, sub, and, or, slt	200	100	200		100	600
beq	200	100	200			500

Picosecond
10^{-12} second

- Single cycle model – every instruction takes 1 cycle
- As the pipeline stages take a single clock cycle in the **non-pipelined** design the time before the first and fourth instruction is 3 x 800 ps = **2400 ps** ← Worst case
- For the **pipelined** design we take the worst case ~200 ps.
  So between first and fourth command is 3 x 200 ps = 600 ps

Pipelining

- Assuming ideal conditions – perfectly balanced stages:

$$\text{Time between instructions}_{\text{Pipelined}} = \frac{\text{Time between instructions}_{\text{Sequential}}}{\text{Number of pipe stages}}$$

- For the large number of instructions the speed up is ~ number of stages (note that some instructions are not balanced + overhead)

- **CHECK**: Pipelining improves the **performance** by:

Increasing instruction **throughput**

Decreasing ~~**execution time**~~ for a single instruction

Pipelining hazards

- Structural hazards: Hardware cannot support the combination of instructions that we want to execute in the same clock cycle (eg. washer/dryer combo or roommate busy elsewhere)

- Data hazards: Pipeline must be stalled because one step is waiting for another to complete (eg. folding up, looking for a missing sock)

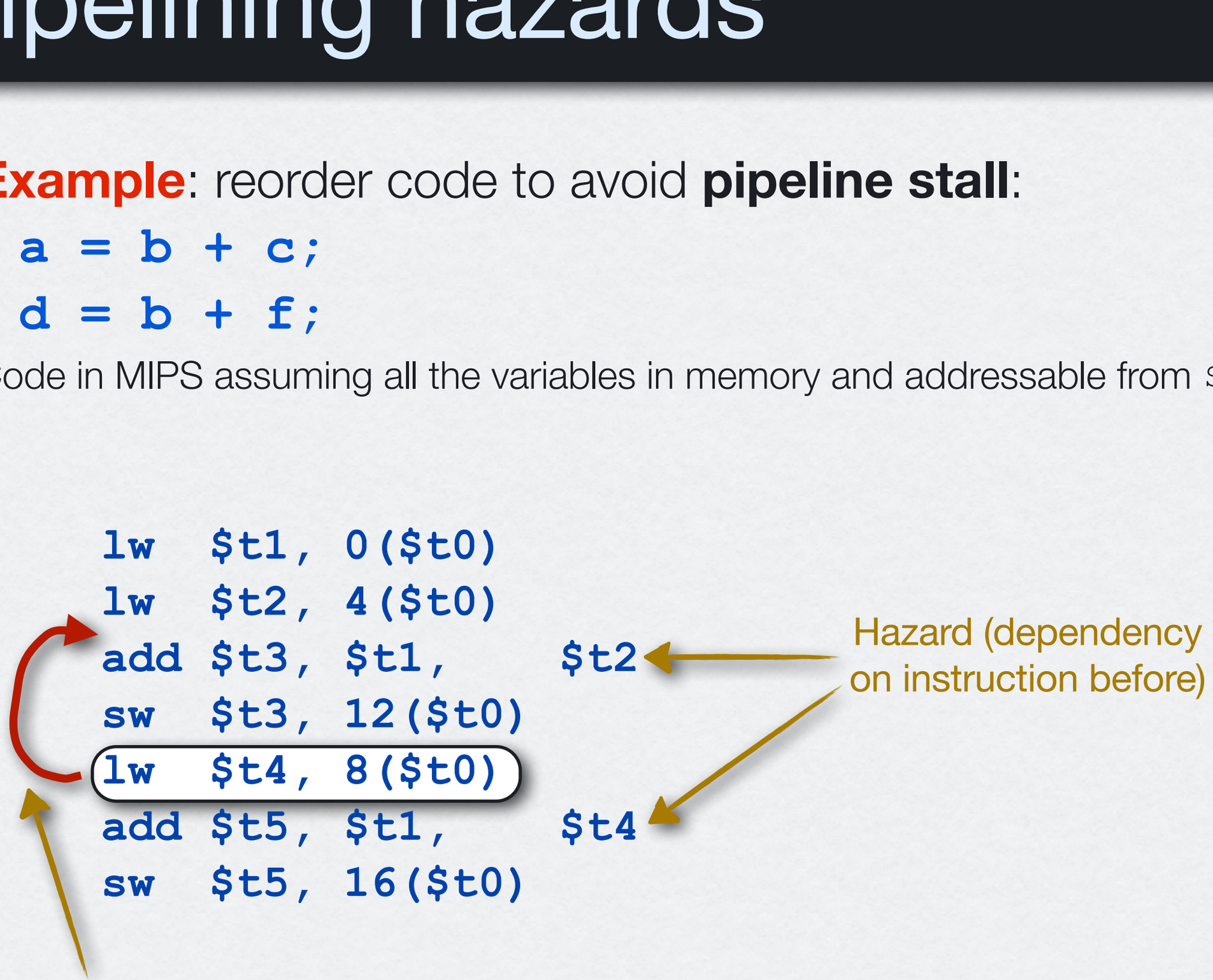
- Control hazards: Need to make a decision based on the results of one instruction while others are executing (e.g. not clear what to do next if a sock not washed). Branch prediction and forwarding help make a computer fast while still getting the right answer

Pipelining hazards

Example: reorder code to avoid **pipeline stall**:

```
a = b + c;
d = b + f;
```

Code in MIPS assuming all the variables in memory and addressable from `$t0`



Pipelining: the BIG picture

- Pipelining increase the number of simultaneously executing instructions and the rate at which instructions are started and completed

- Pipelining **does not** reduce the time it takes to complete on individual instruction (latency)

- Pipelining increases throughput rather than latency

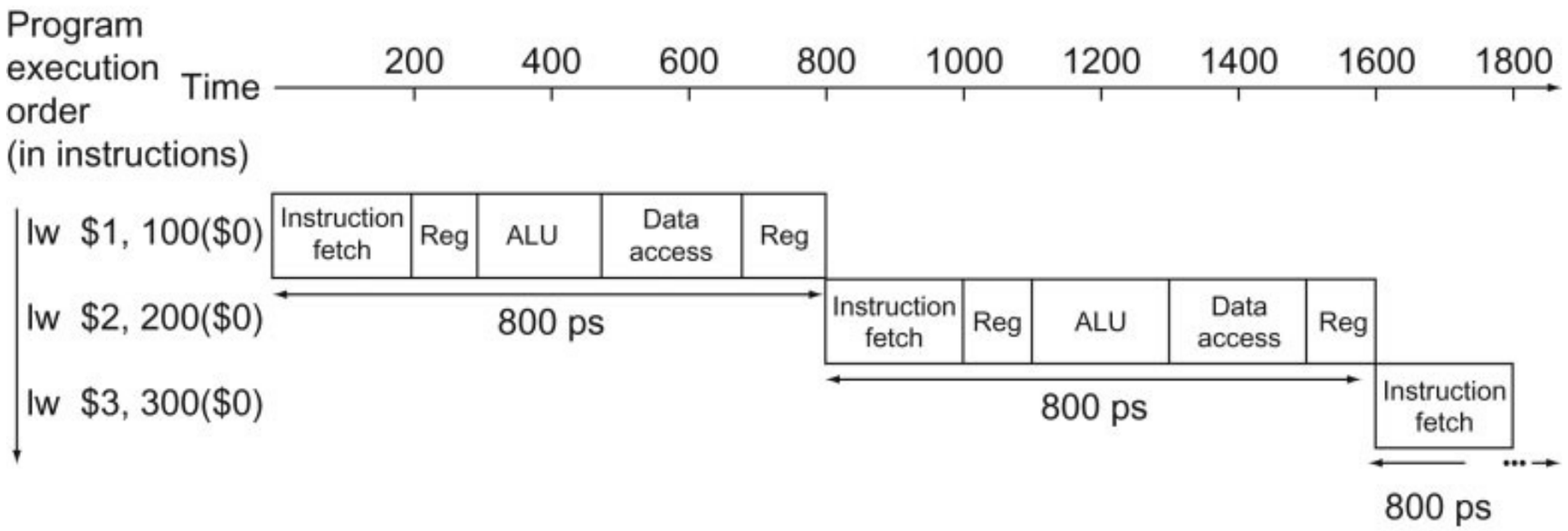
- Pipelining has structural, data, control hazards (important for designers)

(*)

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

FIGURE 4.26 Total time for each instruction calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

(*)



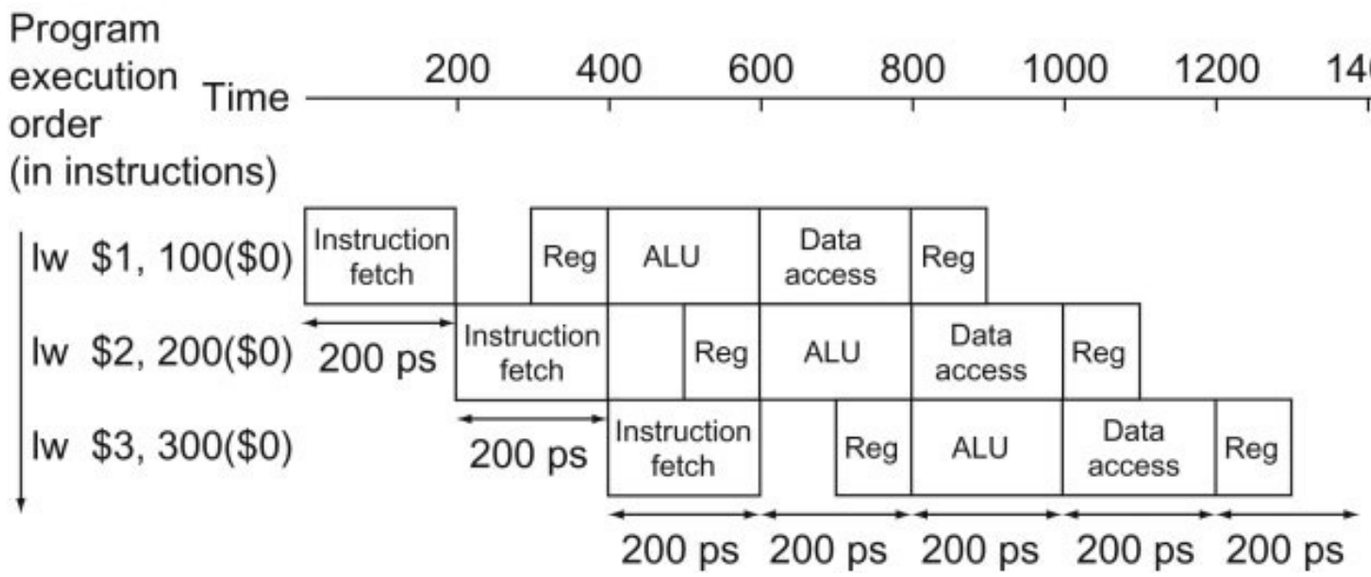


FIGURE 4.27 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 4.26. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.25. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

(*)

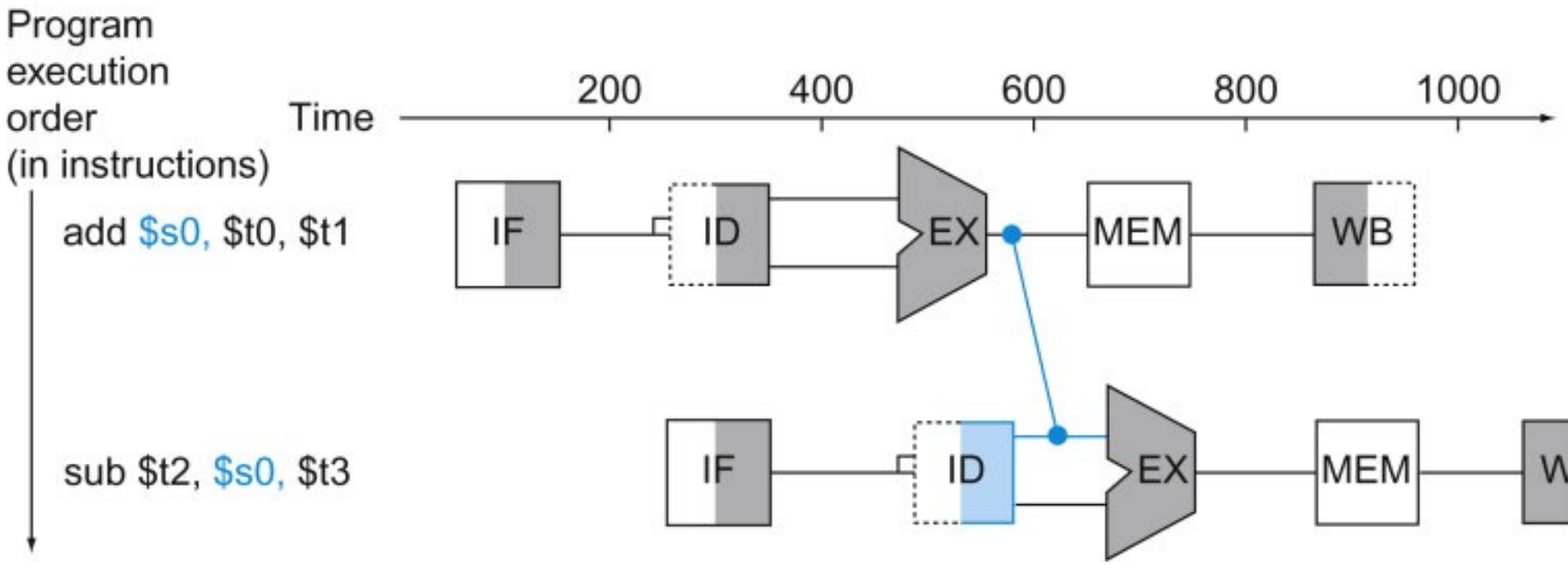


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register $s0 read in the second stage of sub.

(*)

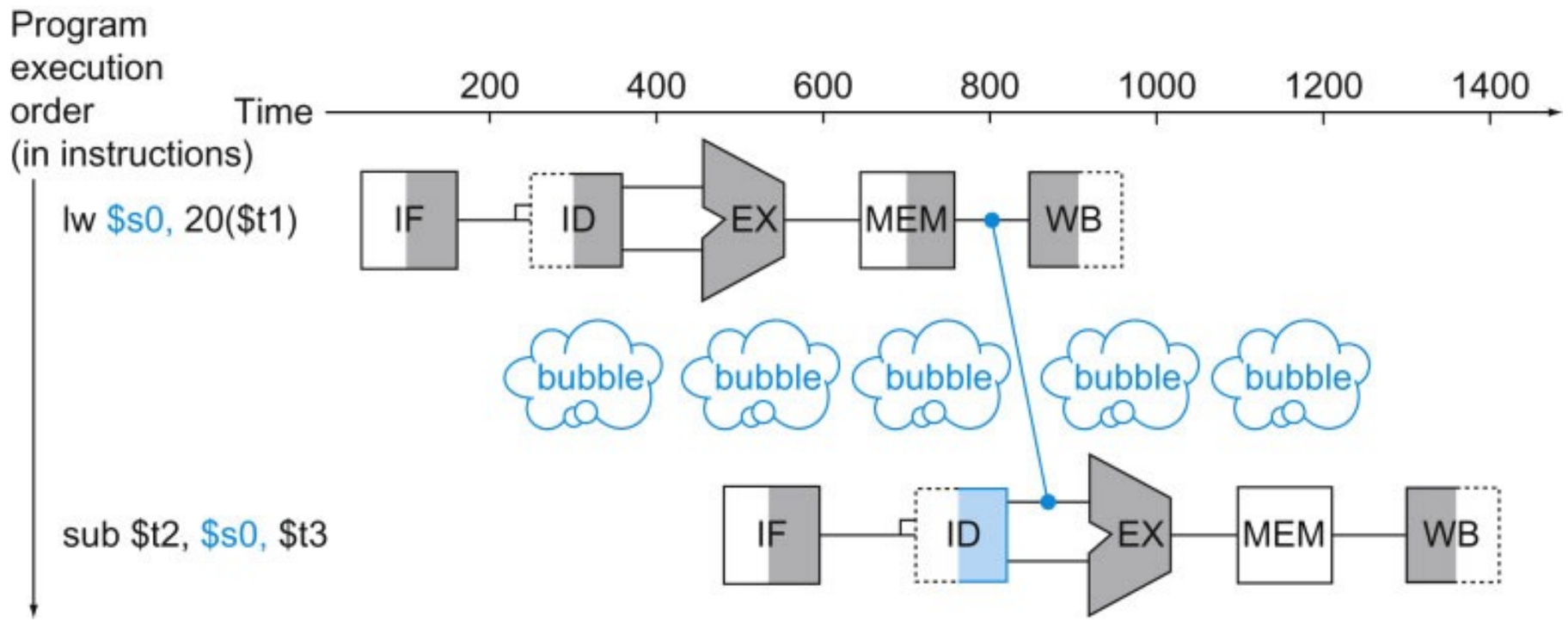


FIGURE 4.30 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 4.7 shows the details of what really happens in the case of a hazard.

(*)

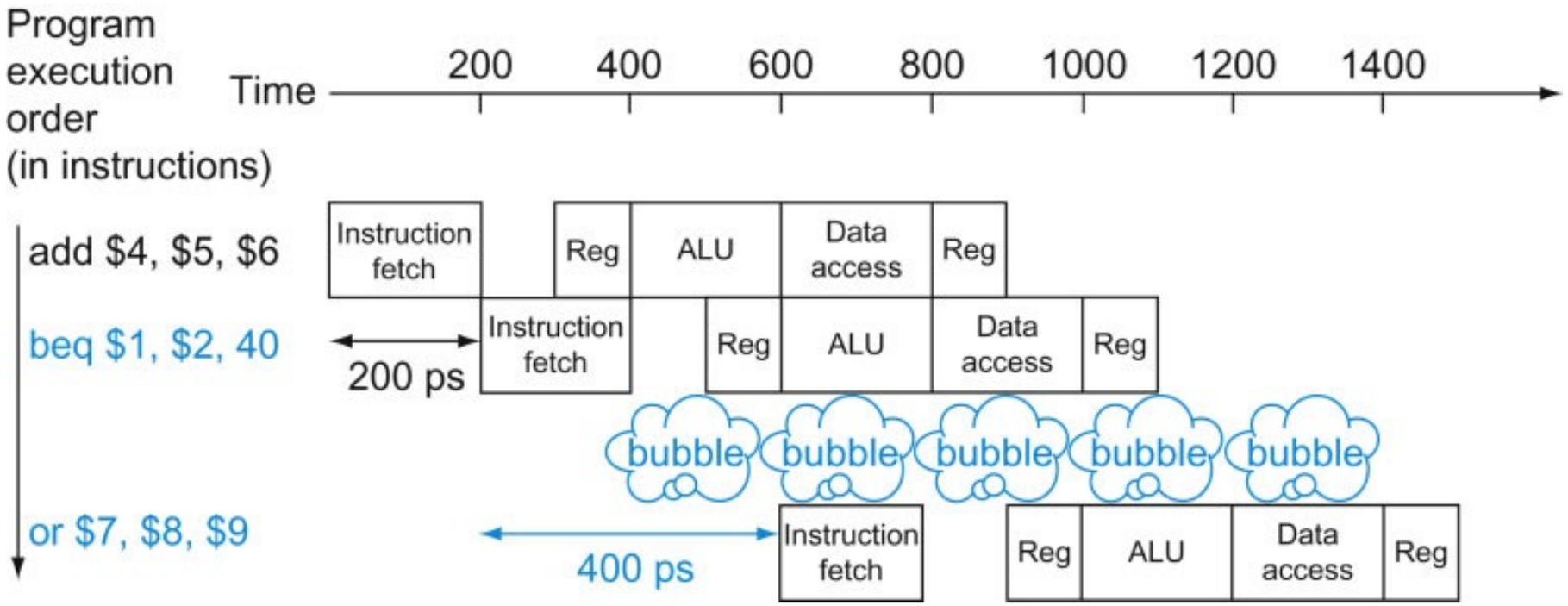


FIGURE 4.31 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.8. The effect on performance, however, is the same as would occur if a bubble were inserted.

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

- Layers of software/hardware
  - Compiler, assembler, hardware

- MIPS: typical of RISC ISAs
  - c.f. x86

Parallelism and computer arithmetic: Associativity

- Integer addition is associative (same result with 1 or 10^6 processor)

- Because FP are approximation of real numbers and because computer arithmetic has limited precision, it does no hold for FPs.

$$x + (y + z) = -1.5 \cdot 10^{38} + \left(1.5 \cdot 10^{38} + 1.0\right) = -1.5 \cdot 10^{38} + 1.5 \cdot 10^{38} = 0.0$$

$$\neq$$

$$(x + y) + z = \left(-1.5 \cdot 10^{38} + 1.5 \cdot 10^{38}\right) + 1.0 = 1.0$$

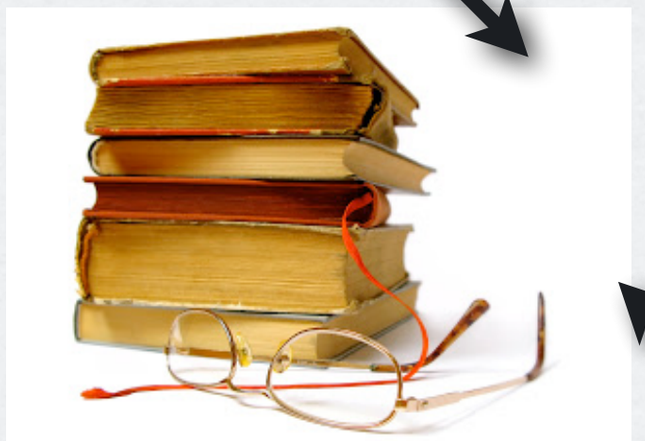
- Bit patterns have no inherent meaning. The same pattern may represent a signed integer, unsigned integer, FP number, instruction and so on. It is the instruction that operates on the word that determines its meaning

- FP arithmetic is not pencil and paper. Numerical analysis is necessary to find the fastest algorithms for parallel computers that still achieves a correct result.

Memory

Memory

Library Paradigm: think of writing an important review on computer hardware.



You go to the library and collect a few books that you put on your desk to consult.

This is faster than taking one book at a time and returning to bookcases every time you need something.

Locality

Same principle allows us to create an illusion of a large memory that we can access as fast as very small memory

➡ HAVE NEAR YOU ONLY WHAT YOU NEED

Principle of LOCALITY

- Temporal locality:

  if an item is referenced, it will need to be referenced again soon

  if you recently brought a book to your desk chances are you need it again

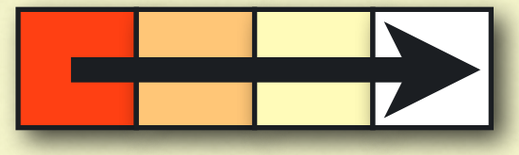
- Spatial locality:

  if an item is referenced, items whose addresses are close by, will tend to be referenced soon

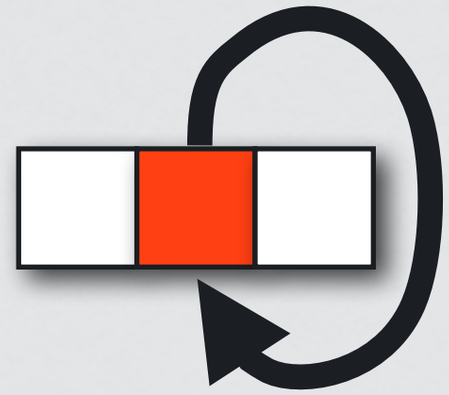
  libraries put books on similar topics nearby to increase spatial locality

Locality examples

- **DATA ACCESS**
  - ➡ sequential access to elements of an array
  - ➡ SPATIAL LOCALITY



- **LOOPS:**
  - ➡ instructions and data accessed repeatedly
  - ➡ TEMPORAL LOCALITY

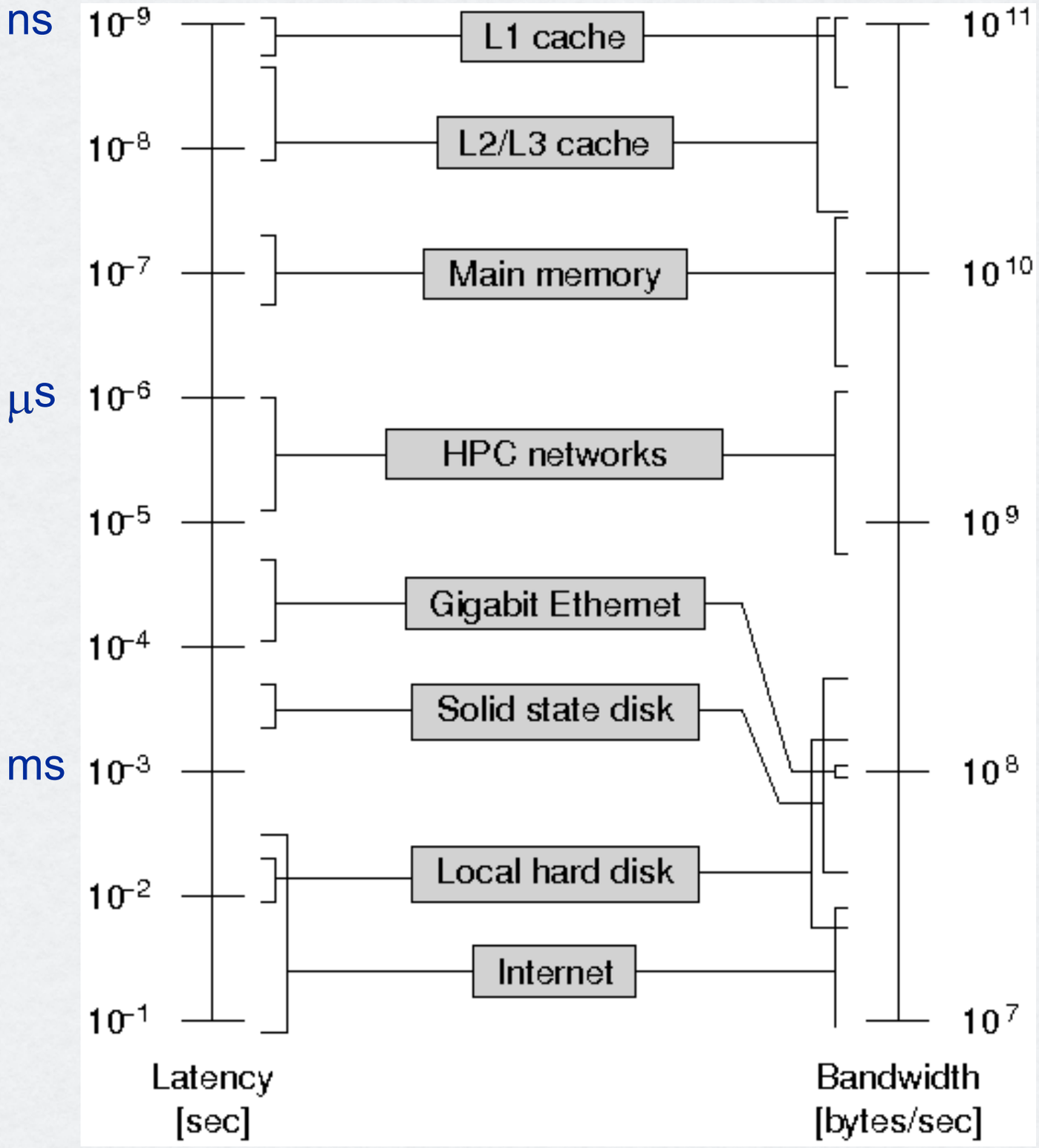


Memory Hierarchy

Multiple levels of memory with different speeds and sizes.

Memory	Access time (ns)	$ per GB (2008)	Library Example
SRAM	0.5 - 2.5	2000 - 5000	*Books on desk*
DRAM	50 - 70	20 - 75	*Books in library*
Magnetic Disk	5×10^6 - 20×10^6	0.2 - 2	*Libraries on campus*

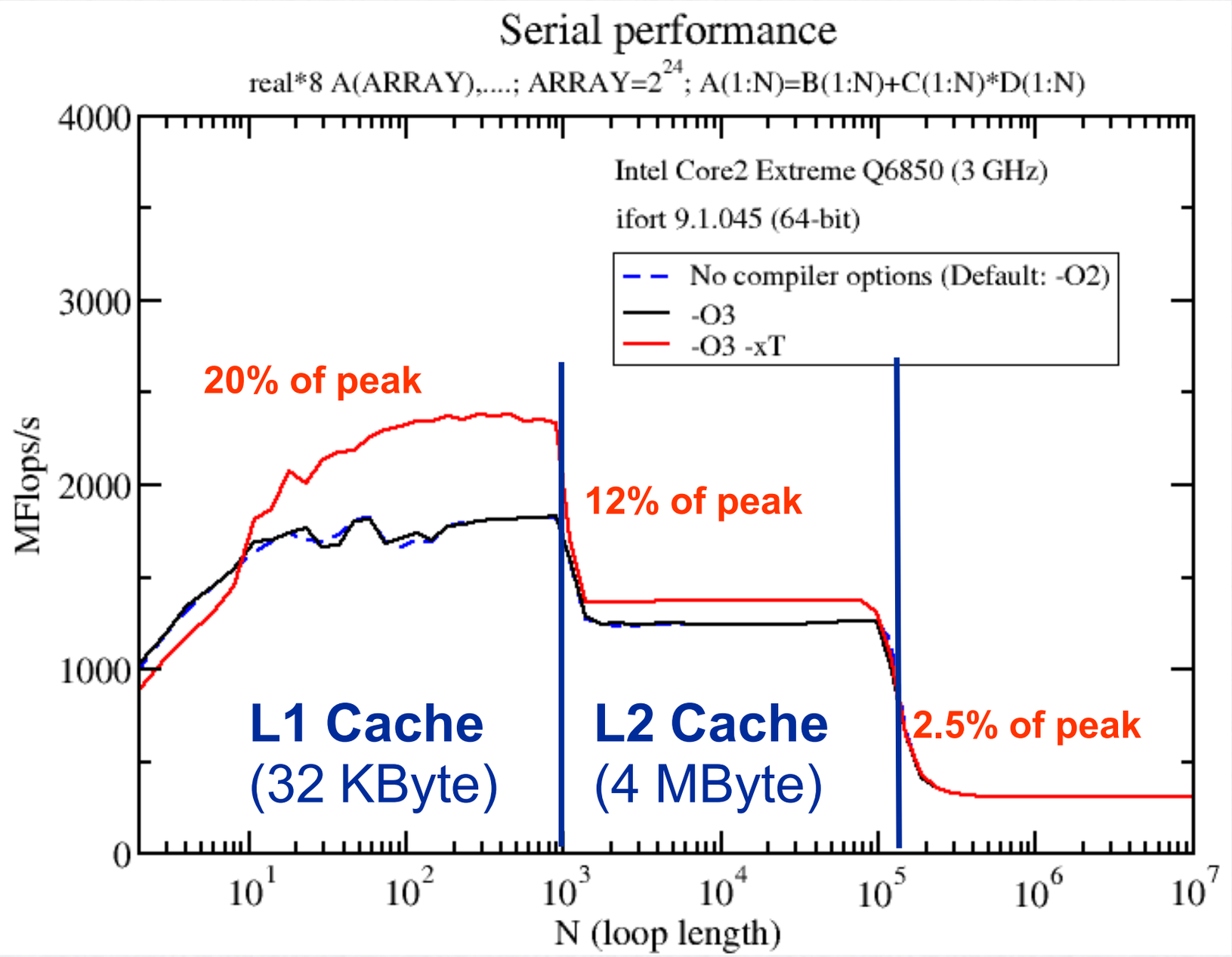
Latency and bandwidth in modern computer environments



Characterization of Memory Hierarchies: Intel Core2 Q6850

Peak Performance for 1 core of Intel Core2 Q6850 (DP):

3 GHz * (2 Flops (DP-Add) + 2 Flops (DP-Mult)) = **12 GFlops/s**



Performance decreases if data set exceeds cache size

`-xT` : Enables vectorization & improves in-cache performance: Packed SSE instructions

Cache: Terminology

- **Block/Line**: minimum unit of information that can be present or not present in a cache *(a book on the desk)*

- **Hit**: data request by the processor, encountered in some block in the *upper (closer to processor)* level of memory hierarchy. If the data is not found, it is a **miss**. Then a *lower (further from processor)* level is accessed to retrieve the data *(you go from your desk to the shelves to find a book)*.

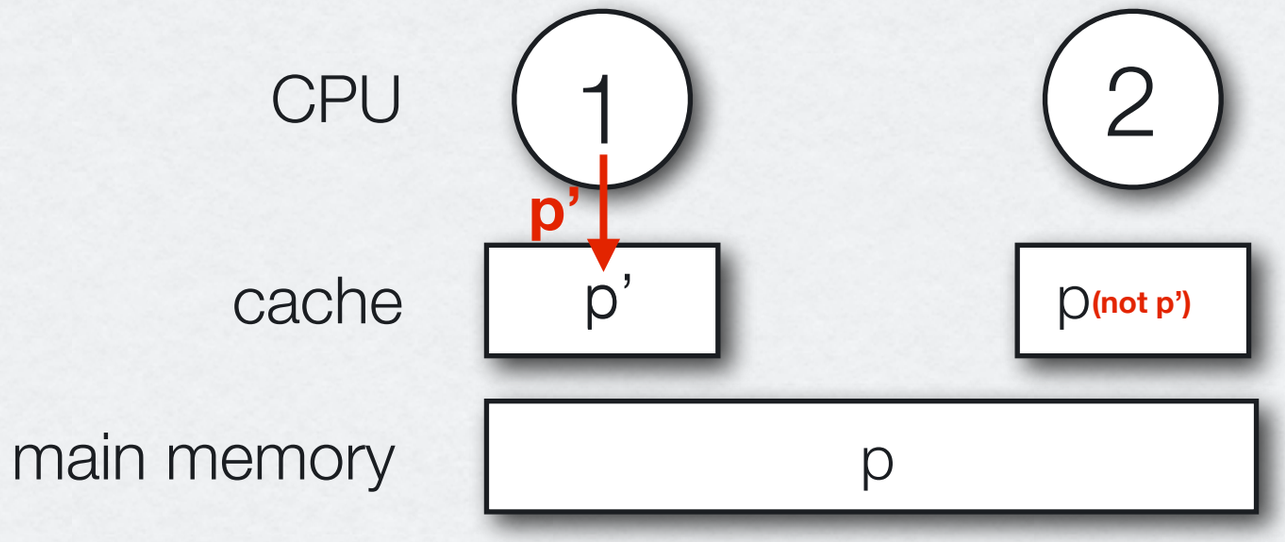
- **Hit rate**: fraction of memory accesses found in the upper level.

- **Hit time**: time to access upper level including time to determine if it is a hit or a miss.

- **Miss penalty**: time to replace a block in the upper level with the corresponding block from the lower level.

Parallelism and Memory Hierarchies

- **Multicore multiprocessor** = multiple processors on a single chip
- Processors (most likely) share a common physical address space
- ***Caching shared data:*** view of memory for each processor through their individual caches so it differs if changes are made.
- CAREFUL: 2 different processors can have 2 different values for the same location **->** cache coherence problem



Cache Coherency

A memory system is coherent if:

1. A read by processor P to location X, that follows a write by P to X, with no writes to X by another processor occurring between the write and read by P, always returns the value written by P.

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occurs between the 2 accesses. ⇒ need controller

3. Writes to the same location are serialized: that is 2 writes to the same location by any 2 processors are seen in the same order by all processors.

Enforcing Coherence

- **Memory latency** refers to delays in transmitting data between the CPU and Memory. Latency is often measured in memory bus clock cycles. However, the CPU operates faster than the memory, so it must wait while the proper segment of memory is located and read, before the data can be sent back. This also adds to the total Memory latency. (*source: Wikipedia*) ***(it's the time required to go to the library and back.)***

- **Migration**: a data item can be moved to a *local cache* and used there in a transparent fashion. Reduces latency access of remote data and bandwidth of shared memory. ***(bring a book to the desk)***

- **Replication**: when shared data are being read simultaneously, the caches make a copy of the data item in the local cache. Reduces access latency and contention for a read in shared data items.

Enforcing Coherence

Protocols are maintained for cache coherence by **tracking the state of any sharing** of a data block.

- Example -> **Snooping protocols**: every cache with a copy of the data from a block of physical memory, also has a copy of the sharing status of the block, but no centralized state is kept.

- The caches are all accessible via some broadcast medium (bus or network) and all cache controllers monitor (snoop) on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access.

Memory Usage: Remarks

- The difficulty of building a memory system to keep pace with faster processors is underscored by the fact that the raw material for main memory, DRAM, is essentially the **same** for fastest as well as for cheapest and slowest computers.

- Principle of Locality: gives chance to overcome long latency of memory access.

Memory Usage: Remarks

- **Multilevel caches** make it possible to use more cache optimization, more easily:
  - design parameters of a lower-level cache are different from a first level cache **(as lower level cache is larger, then we use larger block sizes).**
  - lower level cache is not constantly used by processor, as a first level cache **(allow lower level cache to prevent future misses)**.

Memory Usage: Remarks

- **Software** for improved memory usage. Compilers to transform programs.
  - recognize program to **enhance its spatial and temporal locality** (loop-oriented programs, using large arrays as the major data structure; e.g. large linear algebra problems) by restructuring the loops (to improve locality) large cache performance

- **Prefetching**: a block of data is brought to cache before it is referenced. Hardware to predict accesses that may not be detected by software.

- Cache-aware instructions to optimize memory transfer.

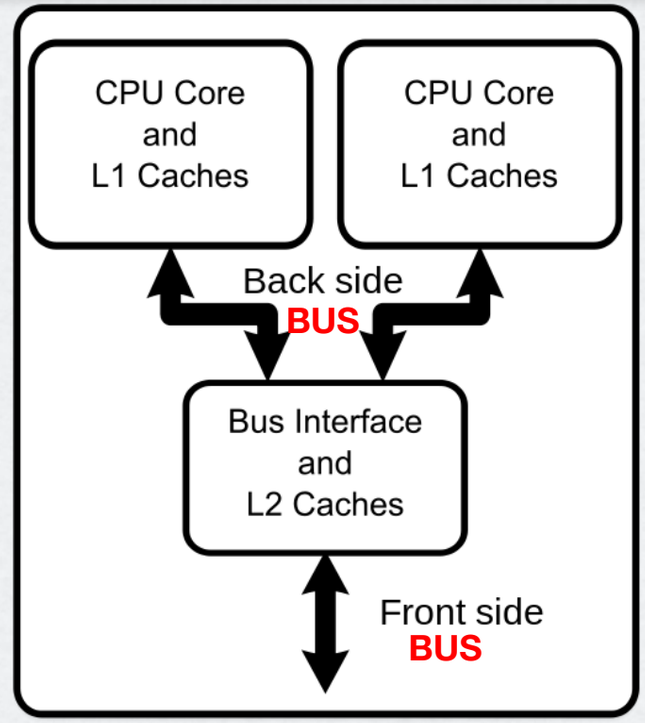
CONNECTING: Processors, Memory and I/O Devices

- **BUS:**
  - A shared communication link, which uses one set of wires to connect multiple subsystems.
  - Used for communication between memory, I/O and processors.

- **versatility**: since we have a single connection scheme, new devices can be added
- **simplicity**: single set of wires is shared in multiple ways
- **communication bottleneck**: limiting the I/O throughput as all information passes a single wire

CONNECTING: Processors, Memory and I/O Devices

- **Processor-Memory Bus**: bus that connects processor and memory, and that is short, high speed and matched to the memory system so as to *maximize memory-processor bandwidth*.



- **I/O Buses**: longer wires and many devices attached and have a wide range of data bandwidth. They do not interface the memory but they go through processor or the backplane bus.

- **Backplane Bus**: a bus designed so that processor, memory and I/O devices can coexist on a single bus.

BUS LIMITS: the length of the bus and the number of devices

Multicore, Multiprocessor & Clusters

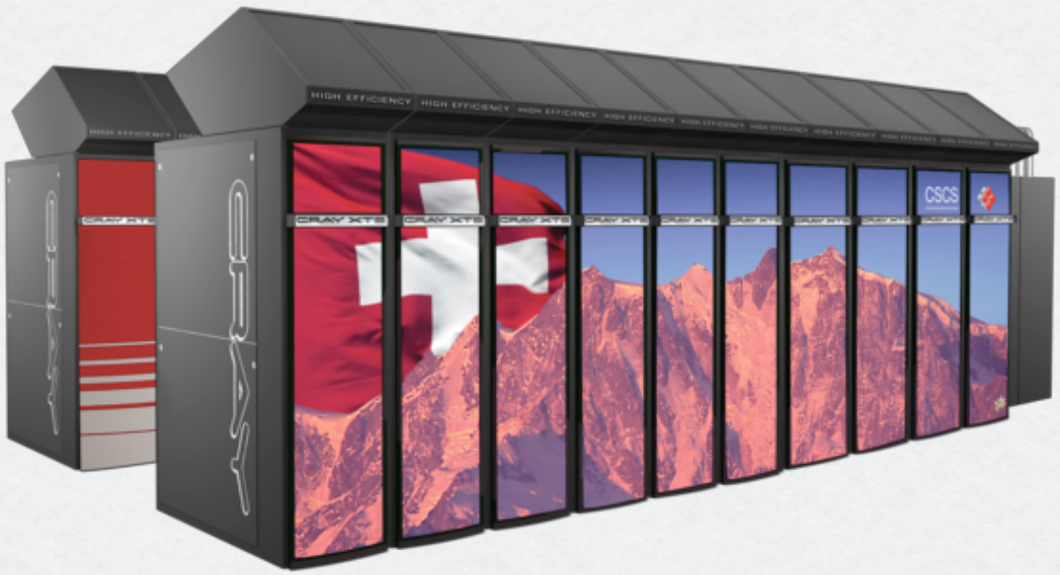
The El-Dorado of Computer Design:

Creating powerful computers simply by connecting many existing smaller ones.

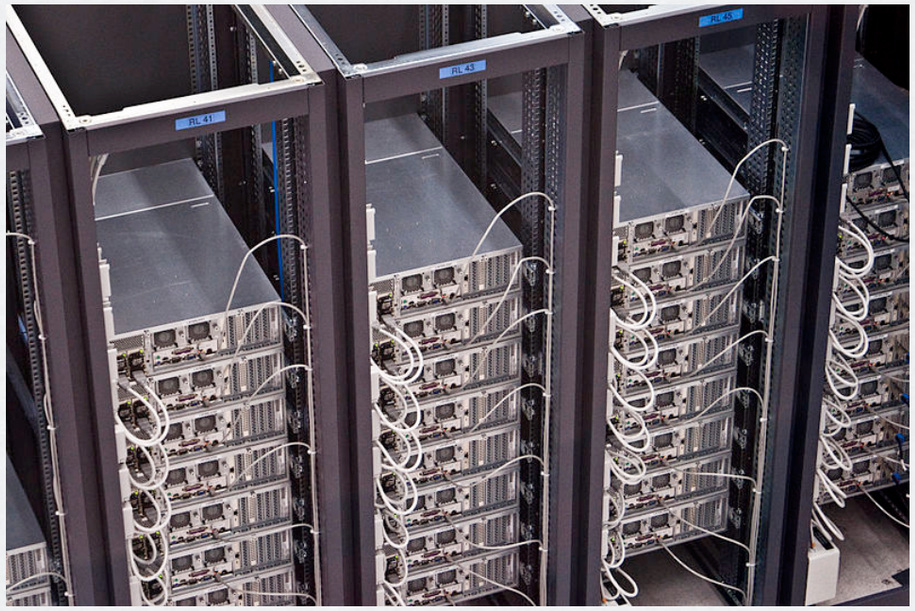
- **Why Smaller computers:**
  - better performance/watt
  - scalability

Cluster

- A set of computers connected over a Local Area Network (LAN) that functions as a single large multiprocessor.
- Performance comes from more processors per chip rather than higher clock



Cray XE6 – Monte Rosa (CSCS)

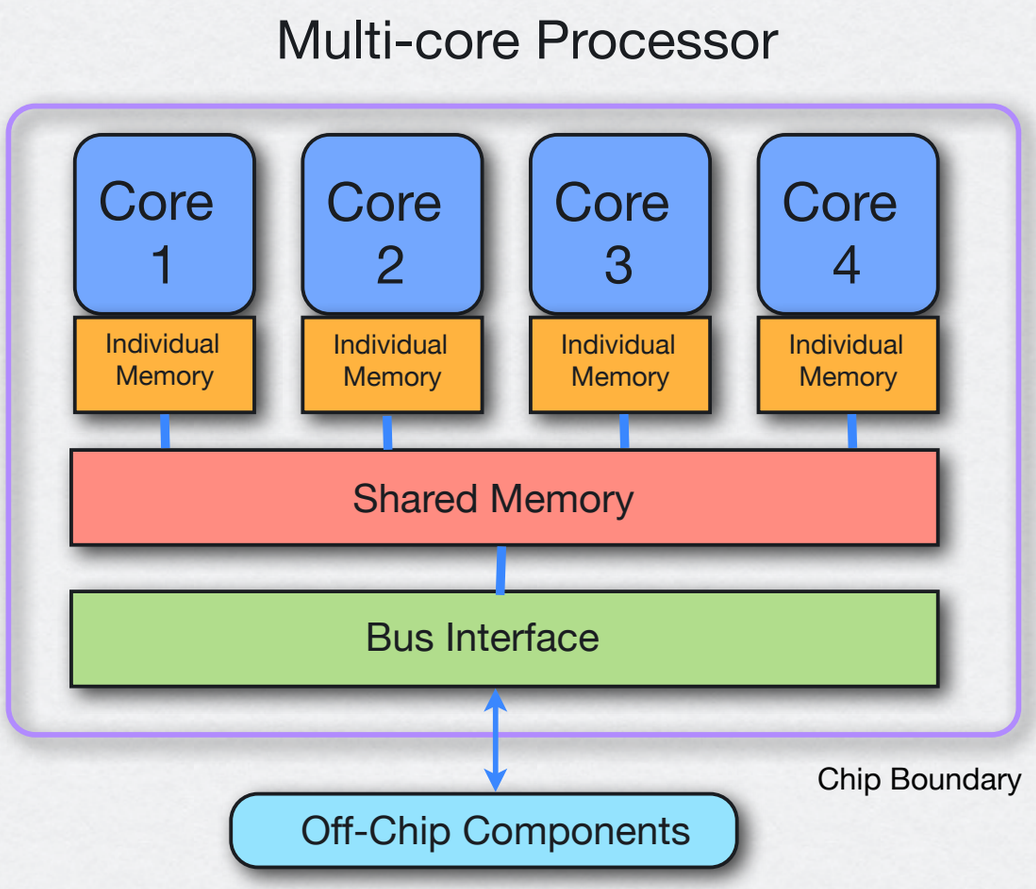


CERN Server

Multicores, Multiprocessors

- multiple processors per chip
- processors are referred to as cores

→ number of cores per chip is expected to double per year



An Intel Core 2 Duo E6750 dual-core processor.

Multicore, Multiprocessor & Clusters

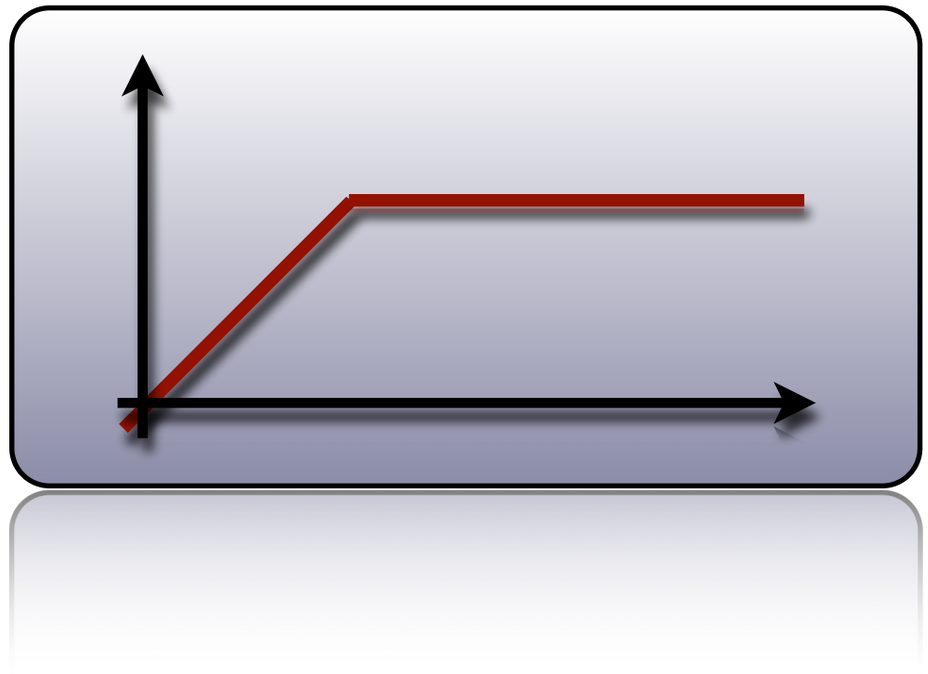
- Job/Process level parallelism:
  - high throughput for *independent* jobs
  - one of the most popular levels of parallelism

- Parallel processing program:
  - a program that runs on multiple processors simultaneously

MEASURING PERFORMANCE

The roofline model

Outline



- Motivation
- 3C model
- The roofline model
- Roofline-based software optimization
- Study case: grid-based simulations on CPU/GPU
- Conclusions

Information taken from:
[1] Roofline: an insightful visual performance model for multicore architectures,
Williams, S. and Waterman, A. and Patterson, D., Communication to ACM, 2009
[2] Intel 64 and IA-32 Architectures Optimization Reference Manual:
http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
[3] Software Optimization Guide for AMD Family 15h Processors:
http://support.amd.com/us/Processor_TechDocs/47414.pdf

Motivation (1)

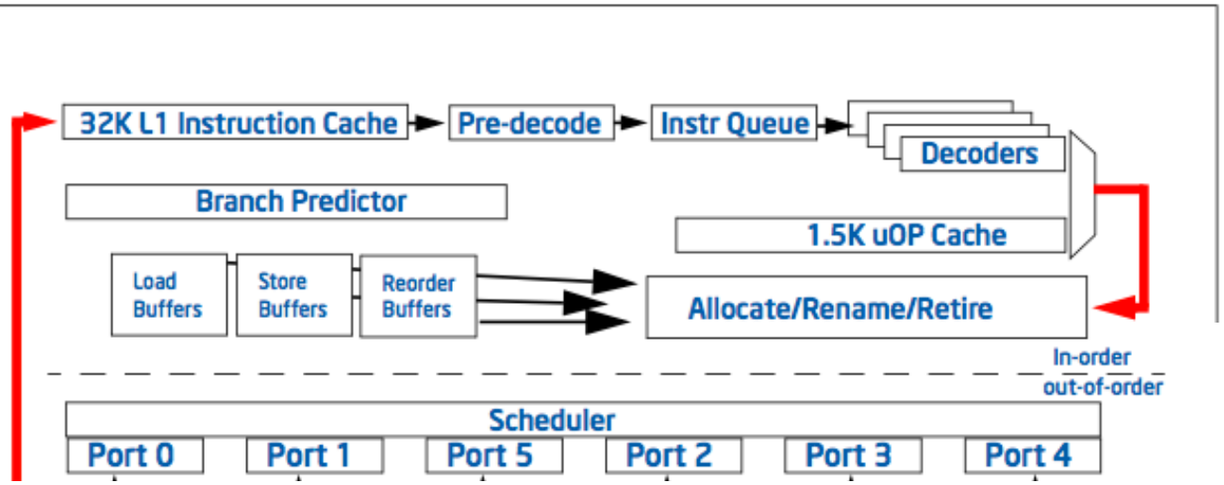
Industry switch to multi/many-core platforms:

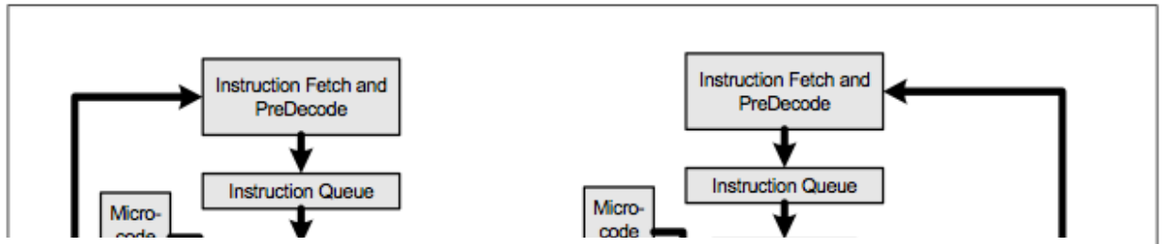
- No conventional wisdom has yet emerged

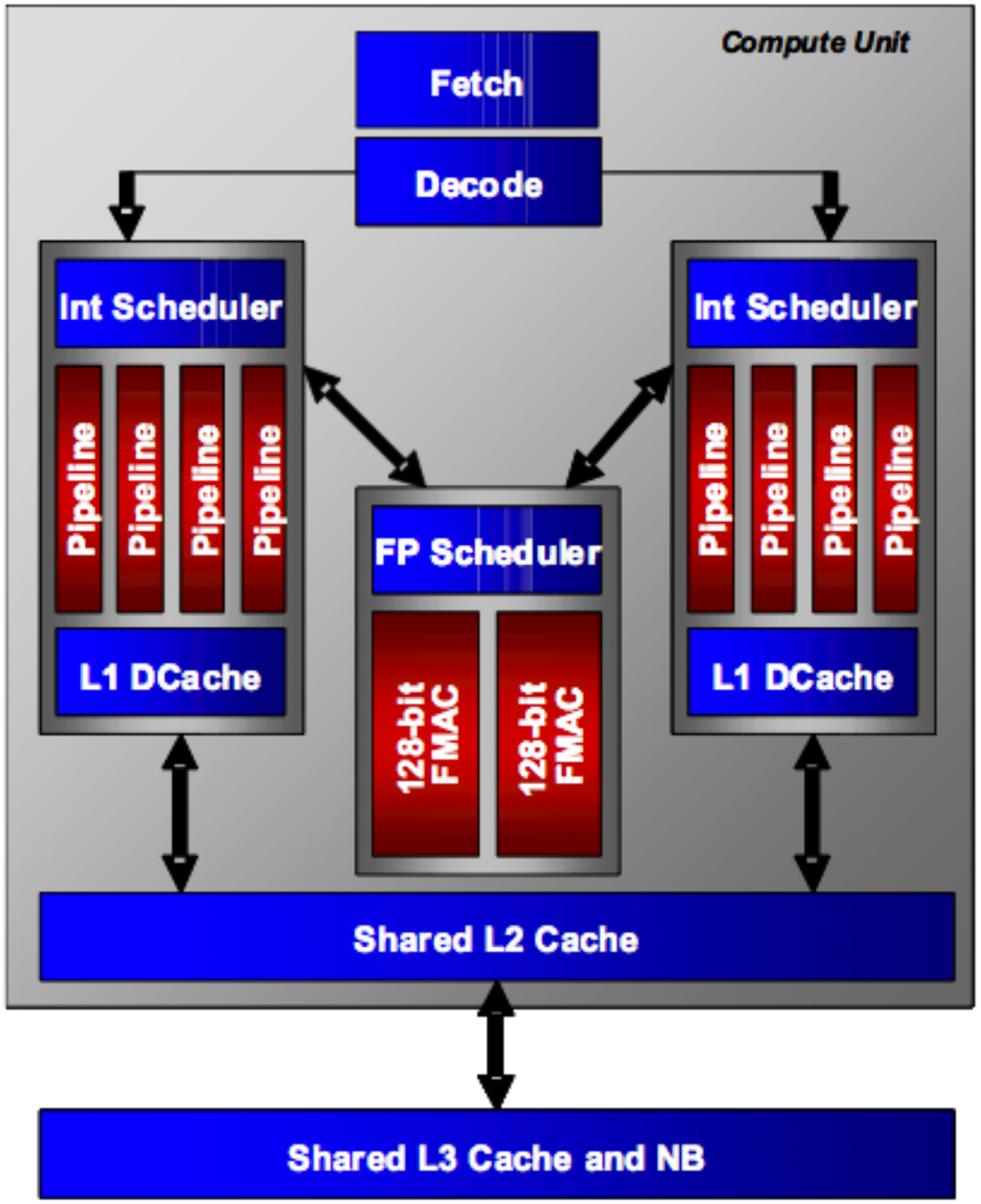
x Micro processors are diverse

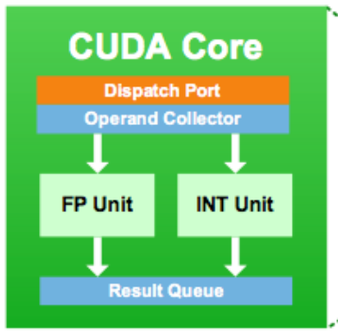
- Differences in:

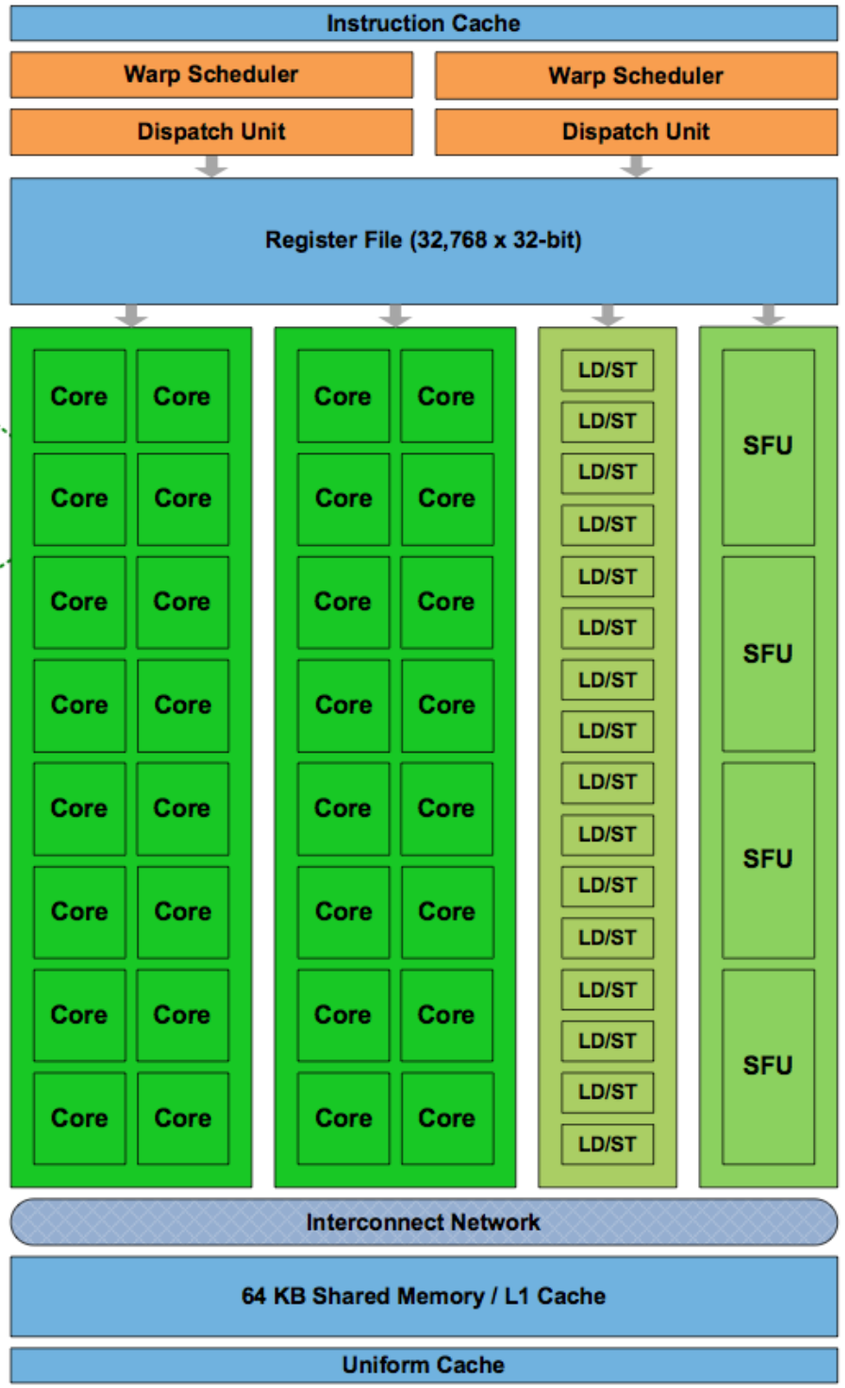
  1. Number of processing elements
  2. Number pipeline stages/execution ports
  3. Instruction issues/scheduling
  4. Memory/cache hierarchies











AMD Family 15h Processor Block Diagram

Motivation (2)

Are these platforms converging into a single one?

- Difficult to say, but it is unlikely (manufacturers perspective):
  - ➡ Moore's law (number of cores will increase)
  - ➡ Vendors want to cover many price-performance points
- Hard times for software developers:
  - ➡ Are there general performance guidelines?
  - ➡ Are there insightful performance models?

Performance model: 3C

"3C" stands for cache misses:

- Compulsory
- Capacity
- Conflict

Used for software optimization [1]:

- Popular for nearly 20 years
- Insightful model
- Not perfect (ambiguity in the cache misses types)

It focuses on:

- Take full advantage of **cache hierarchies**
- Maximize **data reuse**

[1] Hill, M. And Smith, A. Evaluating Associativity In CPU Caches. IEEE Transactions On Computers 38, 1989

The roofline model

Recently proposed by Williams, Waterman and Patterson [1]:

- Crucial in performance predictions
- Helpful for software optimization

"Bound-and-bottleneck" analysis:

- Provides valuable *performance insight*
- Focuses on the *primary performance factors*
- Main *system bottleneck* is highlited and quantified

[1] Roofline: an insightful visual performance model for multicore architectures, Williams and Waterman and Patterson, Communication to ACM, 2009

Transfer-Computation *overlap*

On CPU:

- Superscalar execution (multiple instructions per cycle)
- *In principle*: automatic overlap (balanced instructions)
- ✓ *In practice*: enforced through software prefetching

On GPU:

- Superscalar execution (VLIW for AMD GPUs)
- ✓ Achieved by high occupancies (automatic ILP increase)
- ✓ Achieved by manually improving the ILP [1]
  (by avoiding RAW/WAR conflicts, no renaming on GPU)

[1] V. Volkov. Better performance at lower occupancy, GTC 2010.

The roofline model

Main assumptions/issues:

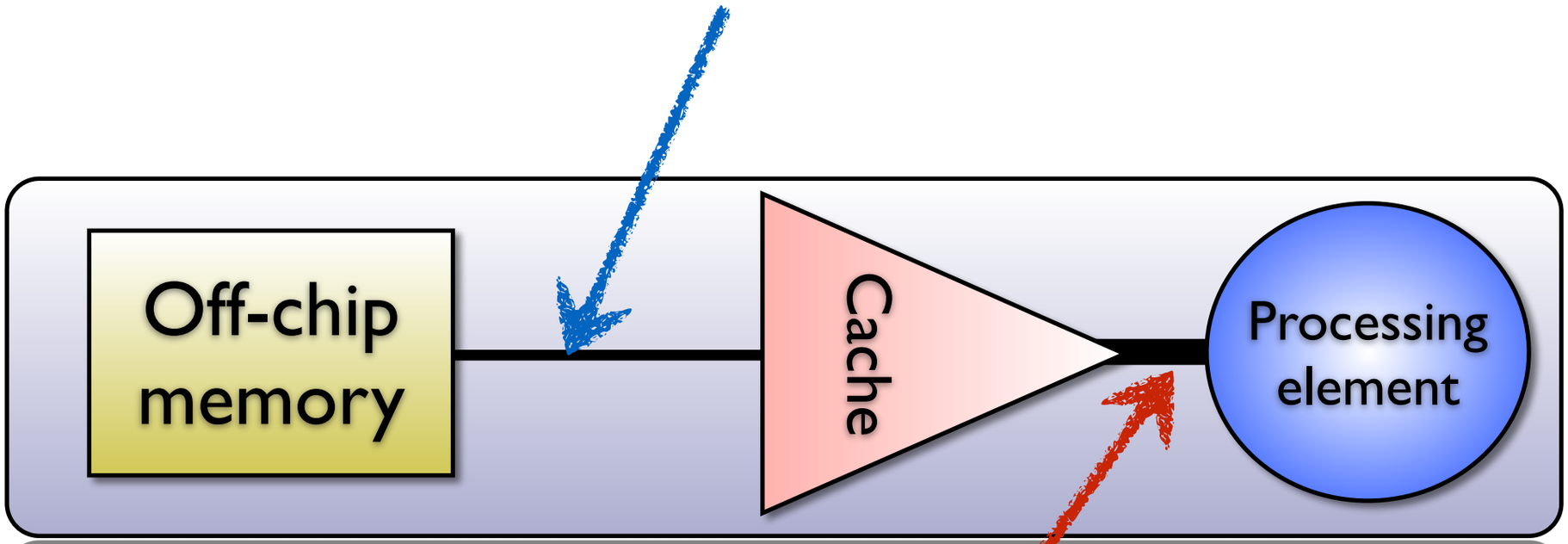
- The memory bandwidth is the constraining resource *(Off-chip system memory)*
- Transfer-computation overlap
- Memory footprint does not fit in the cache

We want a model that relates:

- Computing performance [GFLOP/s]
- Off-chip memory traffic [GB]

➡ New concept: the operational intensity [FLOP/Byte]

Operational intensity



- Not equivalent to arithmetic intensity [1], machine balance [2] (which refer to traffic between the processor and the cache)
- Not forcedly bound to FLOP/Bytes (e.g. Comparison/Byte)

[1] Harris, M. Mapping Computational Concepts To Gpus. In ACM SIGGRAPH Courses, 2005.
[2] Callahan, D., Cocke, J., And Kennedy, K. Estimating Interlock And Improving Balance For Pipelined Machines (1988)

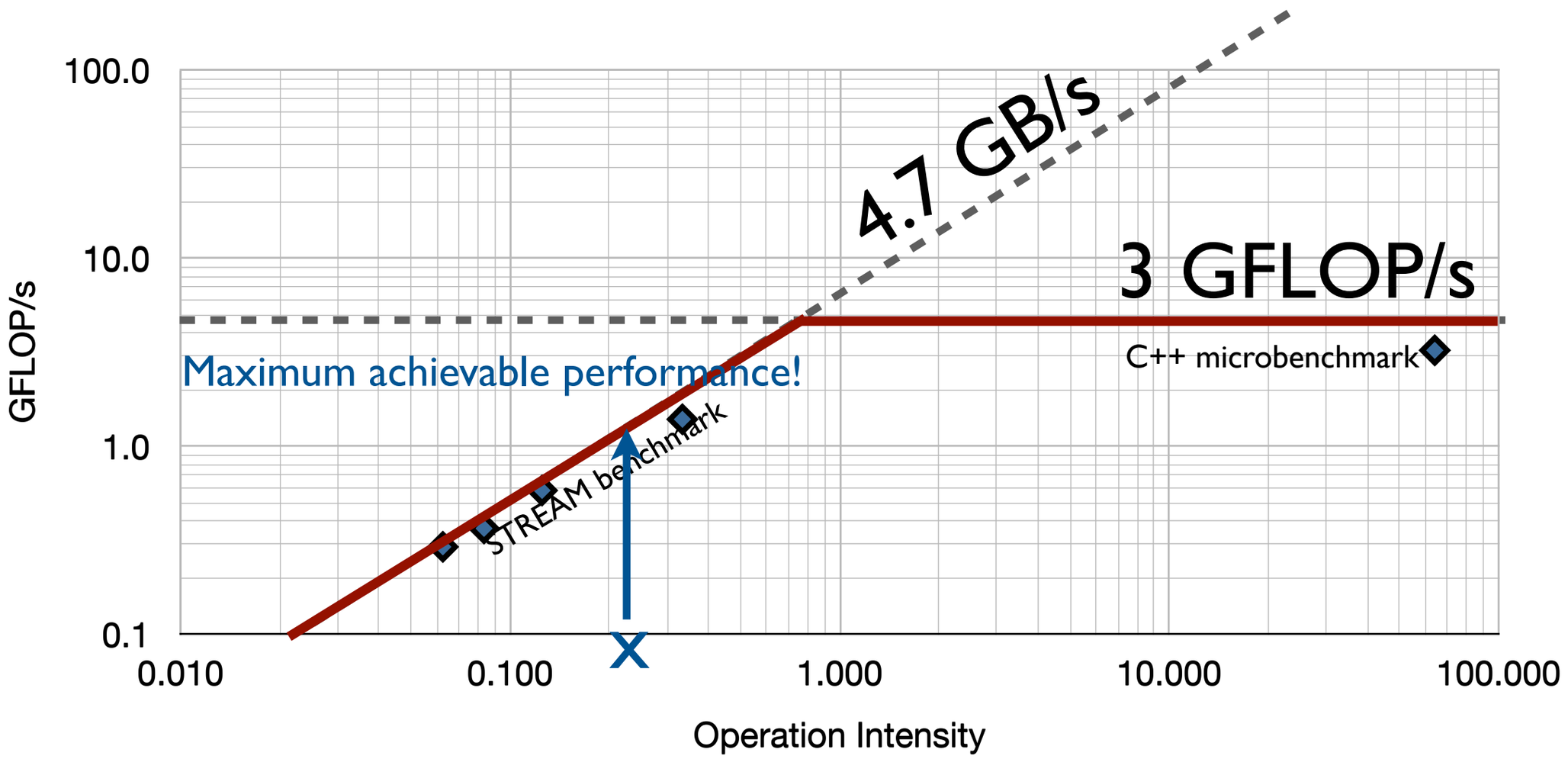
The roofline model

- The roofline is a ***log-log plot***
- It relates:
  - Performance f [FLOP/s] with
  - Operational intensity r [FLOP/Byte]
- Two theoretical regimes for a kernel k:
  1. Performance of k is limited by the DRAM bandwidth:

     → $f(r_k) = r_k \; b_{peak}$
  2. Performance of k is limited by the compute power:

     → $f(r_k) = f_{peak}$

The roofline model

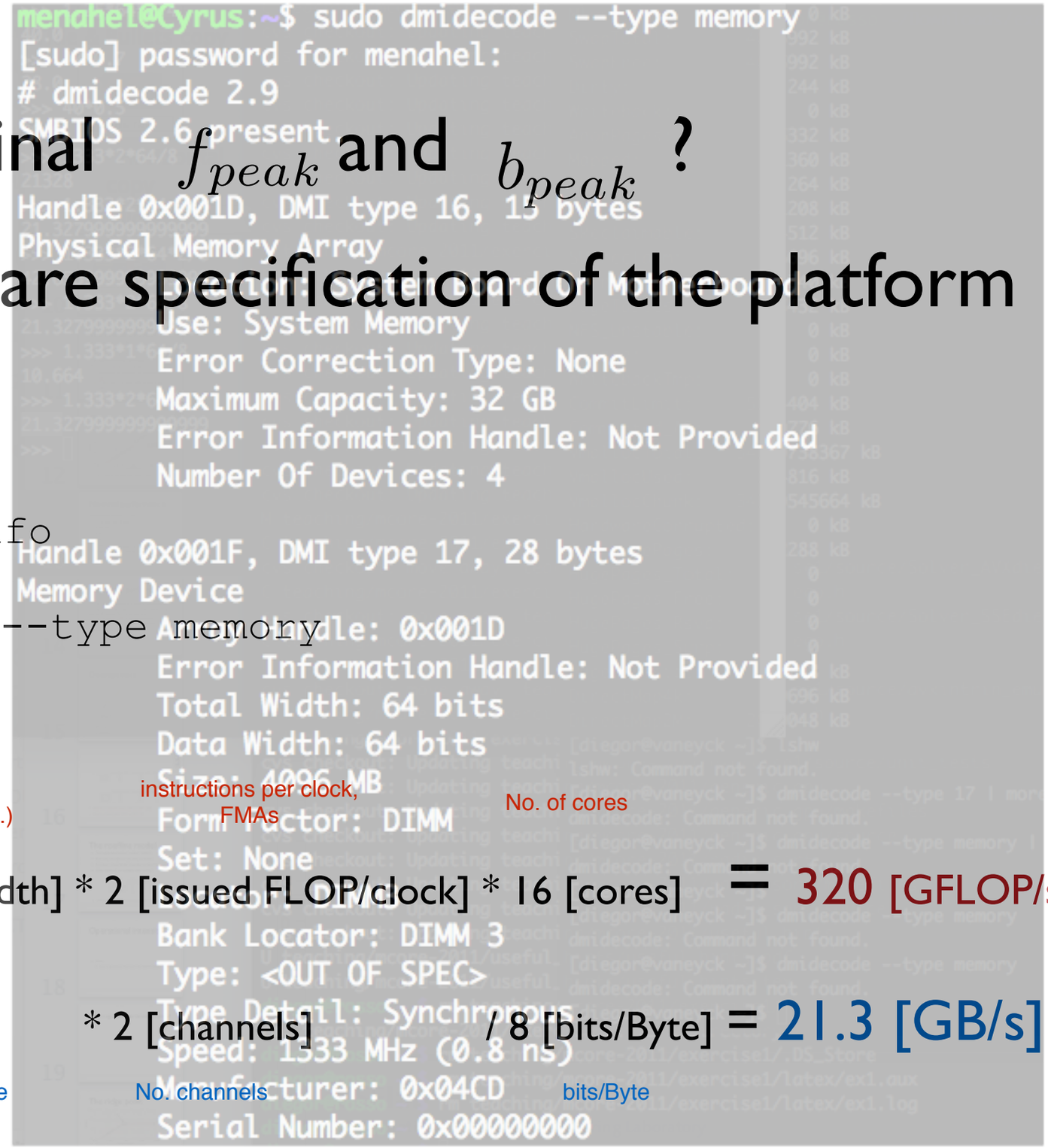


◆ 4x Quad-Core AMD Opteron 8380 @ 2.5GHz - 1 Thread - C++

Nominal performance

How to estimate nominal f_{peak} and b_{peak} ?

- From the hardware specification of the platform
- On Linux
  - **type** `cat /proc/cpuinfo`
  - And `sudo dmidecode --type memory`



Examples

Processor Clock/sec — Vector size ((SSE, AVX,..) — instructions per clock, FMAs — No. of cores

➡ **PP**: 2.5 [Ghz] * 4 [SIMD-width] * 2 [issued FLOP/clock] * 16 [cores] = 320 [GFLOP/s]

➡ **PB**: 1.3 [Ghz] * 64 [bits] * 2 [channels] / 8 [bits/Byte] = 21.3 [GB/s]

Memory Clock/sec — Channel size — No. channels — bits/Byte

Measured performance

Microbenchmarks:

- STREAM benchmark or similar
- Code: `computepower.cpp, bandwidth.cpp`

Expected discrepancy from nominal quantities:

✓ FLOP/s: 90-100% of nominal performance

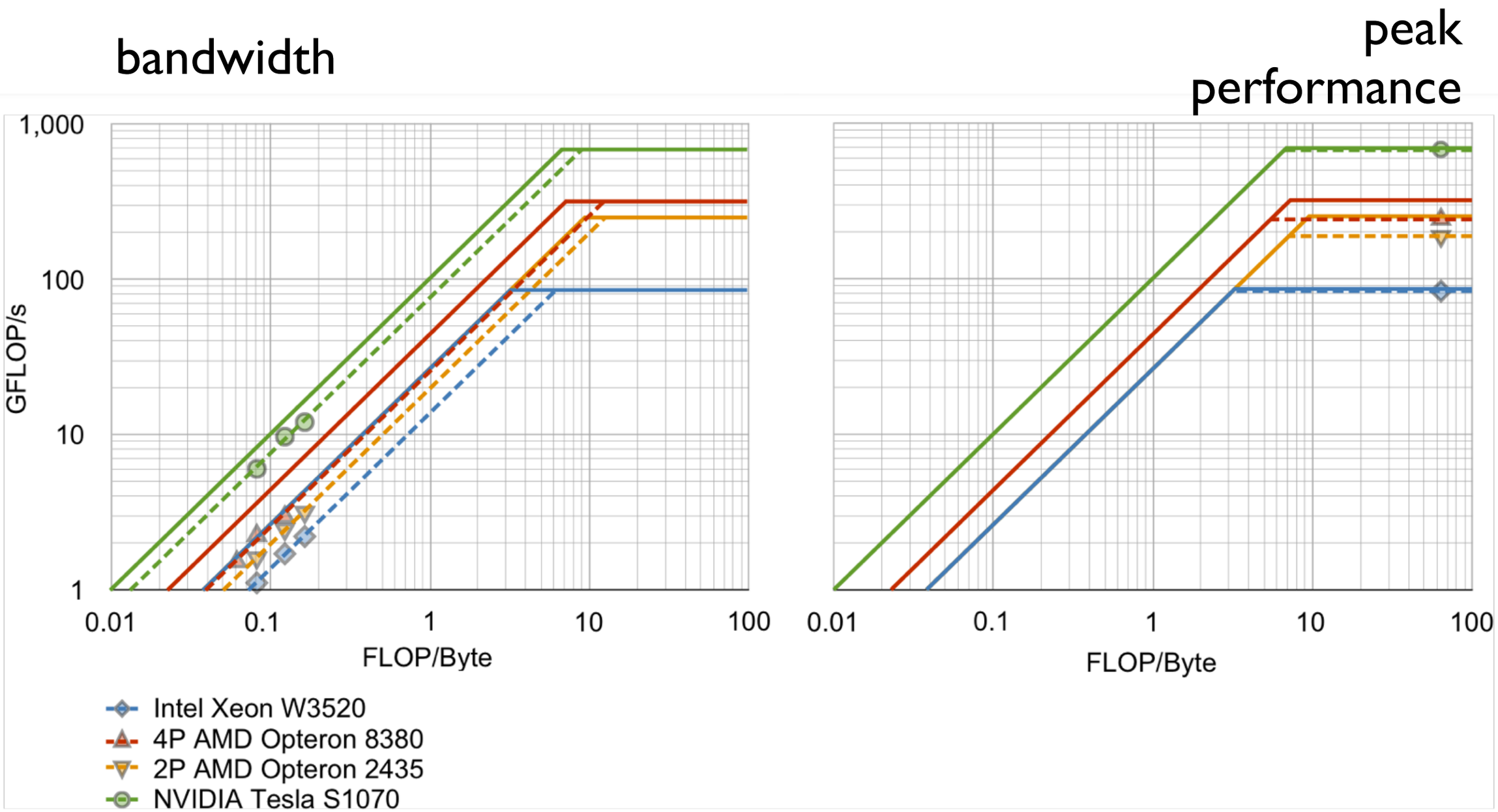
x GByte/s : 50-70% of nominal performance

Discrepancies reveal:

➡ *Programming skills in extracting system performance*

➡ *Best case scenario for more complex kernels*

Discrepancies (measured(dashed lines) vs nominal (solid lines))



- Real scenario: brutus cluster node
- 16 CPU cores (max), 1 Tesla GPU
- Bandwidth discrepancies more pronounced

GPU gain vs. CPU

	Observed Peak [GFLOP/s]	Theoretical Peak [GFLOP/s]	Efficiency [%]
Xeon W3520	83.2	85.4	97
Opteron 8380	237.2	320.0	74
Opteron 2435	184.9	249.6	74
Tesla S1070	684.0	691.2	99

	Observed Peak [GB/s]	Theoretical Peak [GB/s]	Efficiency [%]
Xeon W3520	13.4	25.5	52
Opteron 8380	24.5	42.5	58
Opteron 2435	19.2	25.6	75
Tesla S1070	73.8	100.2	74

GPUs are faster than CPUs

- **GFLOP/s: 2X-8X**
- **GB/s: 1.7X-5.5X**

The roofline model

✓ Run once per platform, not once per kernel

x Estimation of operational intensities (Flops/byte) can be tricky

x What happens if you compute them wrong?

	add $z_i = x_i + y_i$	scale $z_i = \alpha x_i$	triad $z_i = \alpha x_i + y_i$
Intel Xeon W3520	$\frac{1}{12}$ 2 read (x,y) 1 write(z)	$\frac{1}{8}$ 1 read (x) 1 write(z)	$\frac{2}{12}$ 2 read (x,y) 1 write(z)
4P AMD Opteron 8380	$\frac{1}{16}$ 3 read (x,y, z) 1 write(z)	$\frac{1}{12}$ 2 read (x,z) 1 write(z)	$\frac{2}{16}$ 3 read (x,y,z) 1 write(z)
2P AMD Opteron 2435	$\frac{1}{12}$	$\frac{1}{8}$	$\frac{2}{12}$
NVIDIA Tesla S1070	$\frac{1}{12}$	$\frac{1}{8}$	$\frac{2}{12}$

NOTE:
Cache Dependent Numbers

➡ *What is the operational intensity of your algorithm steps?*

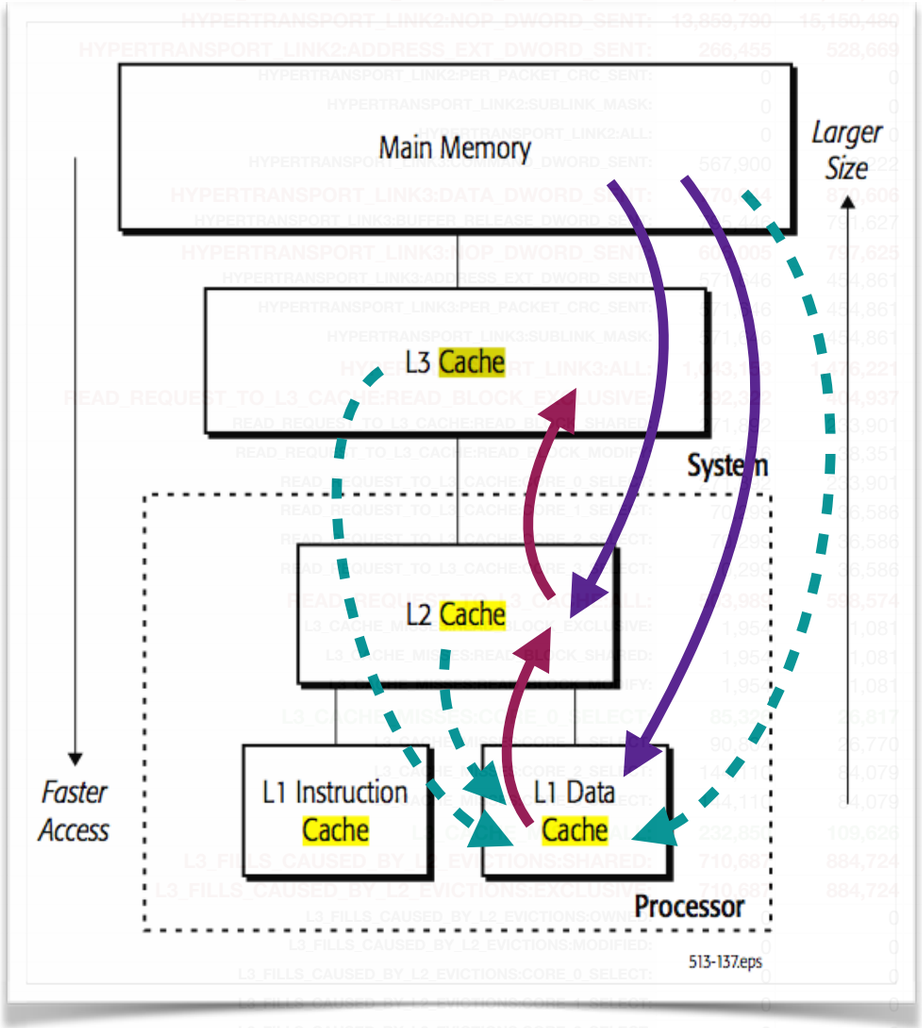
Operational intensity

How to determine the DRAM traffic?

1. By hand (difficult)

- Without cache hierarchy: $t^{write} = t^{read}$
- With cache hierarchy:
  - Write-through: $t^{write} = t^{read}$
  - Write-back(*): $t^{read} = 2\ t^{write}$

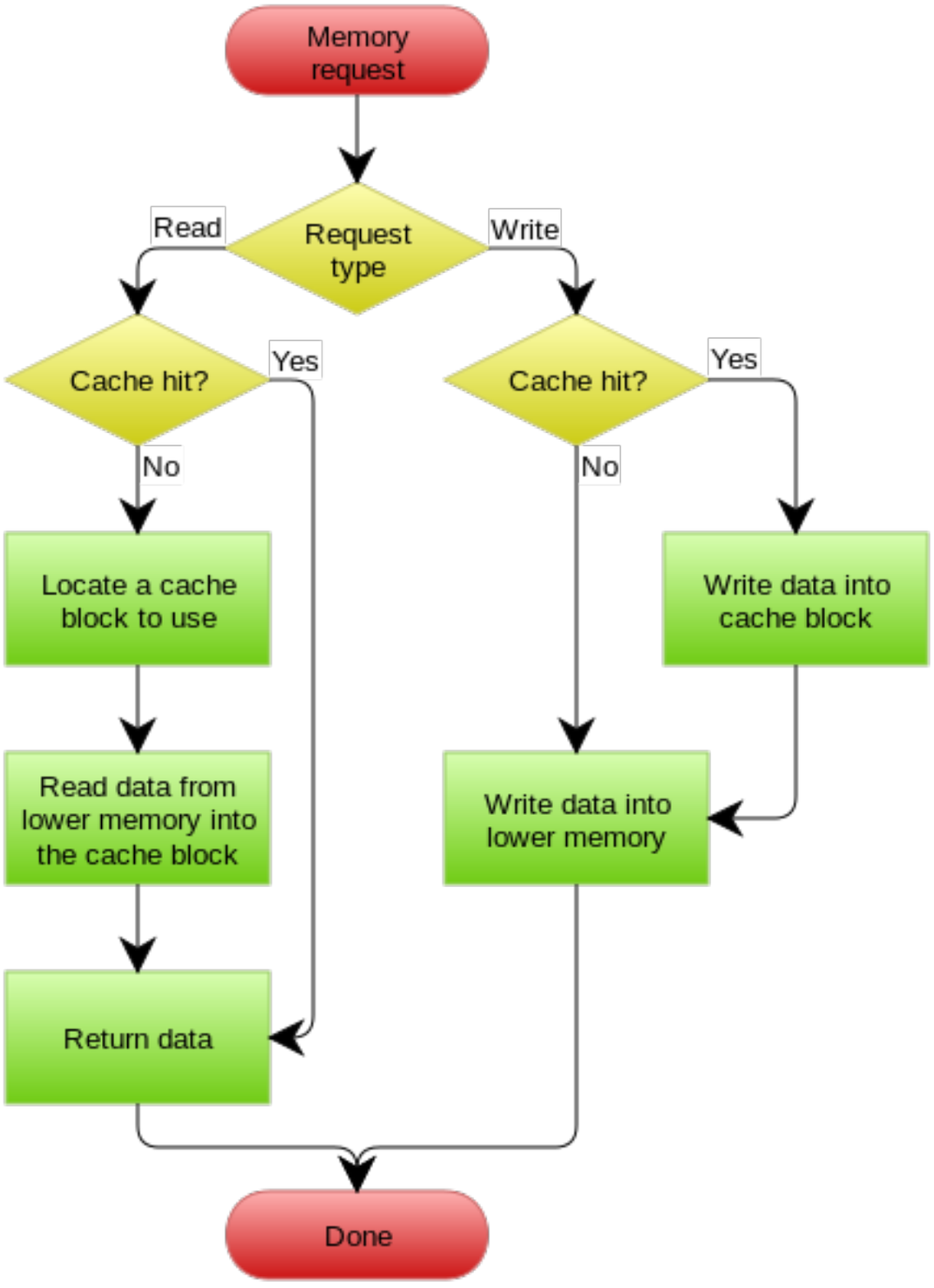
2. With perfomance counters (difficult)



(*)For write back : a read cache miss requires 2 memory accesses: one is to write back to the memory the data and one to read another piece of data

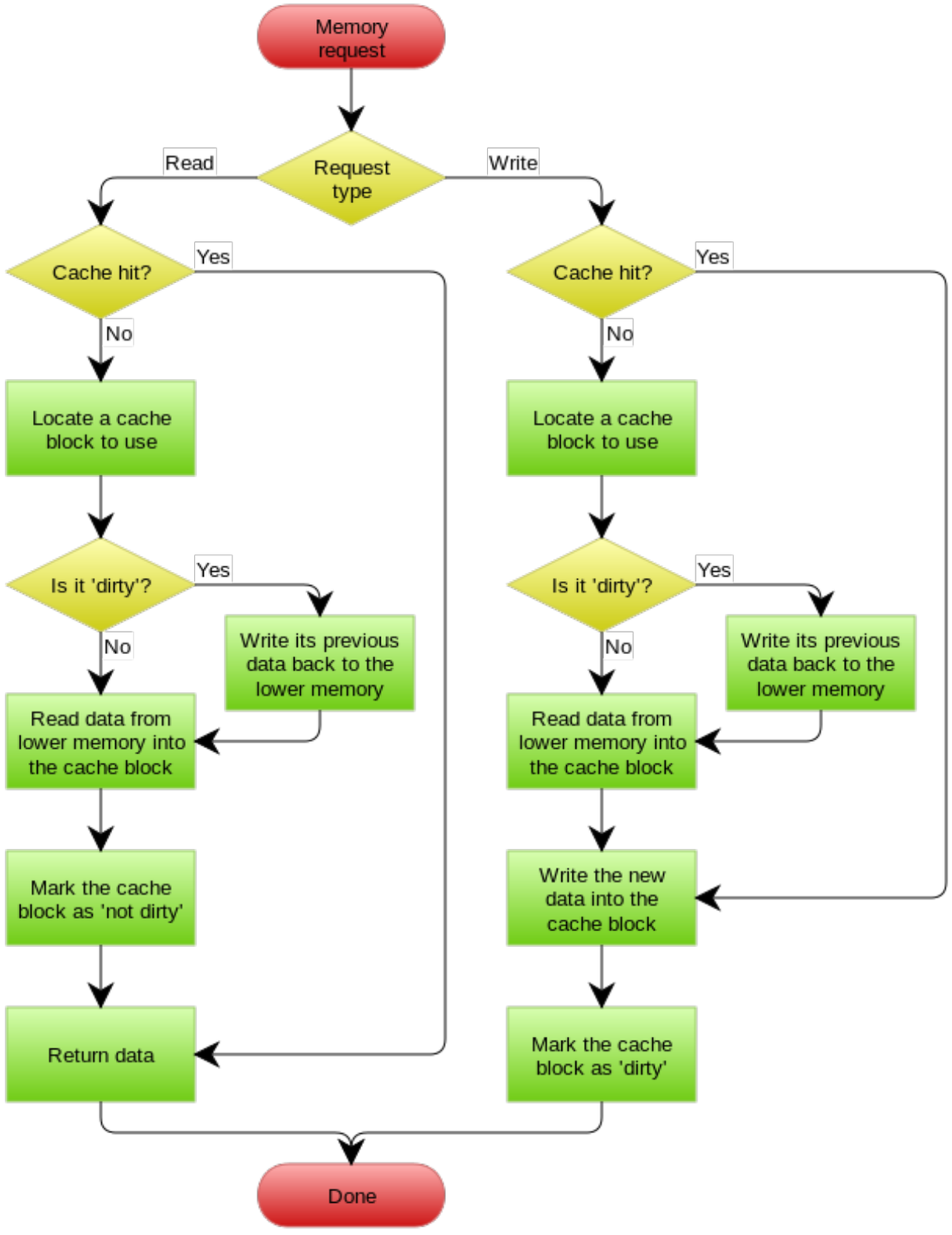
WRITE THROUGH CACHE

update different levels of cache and main memory



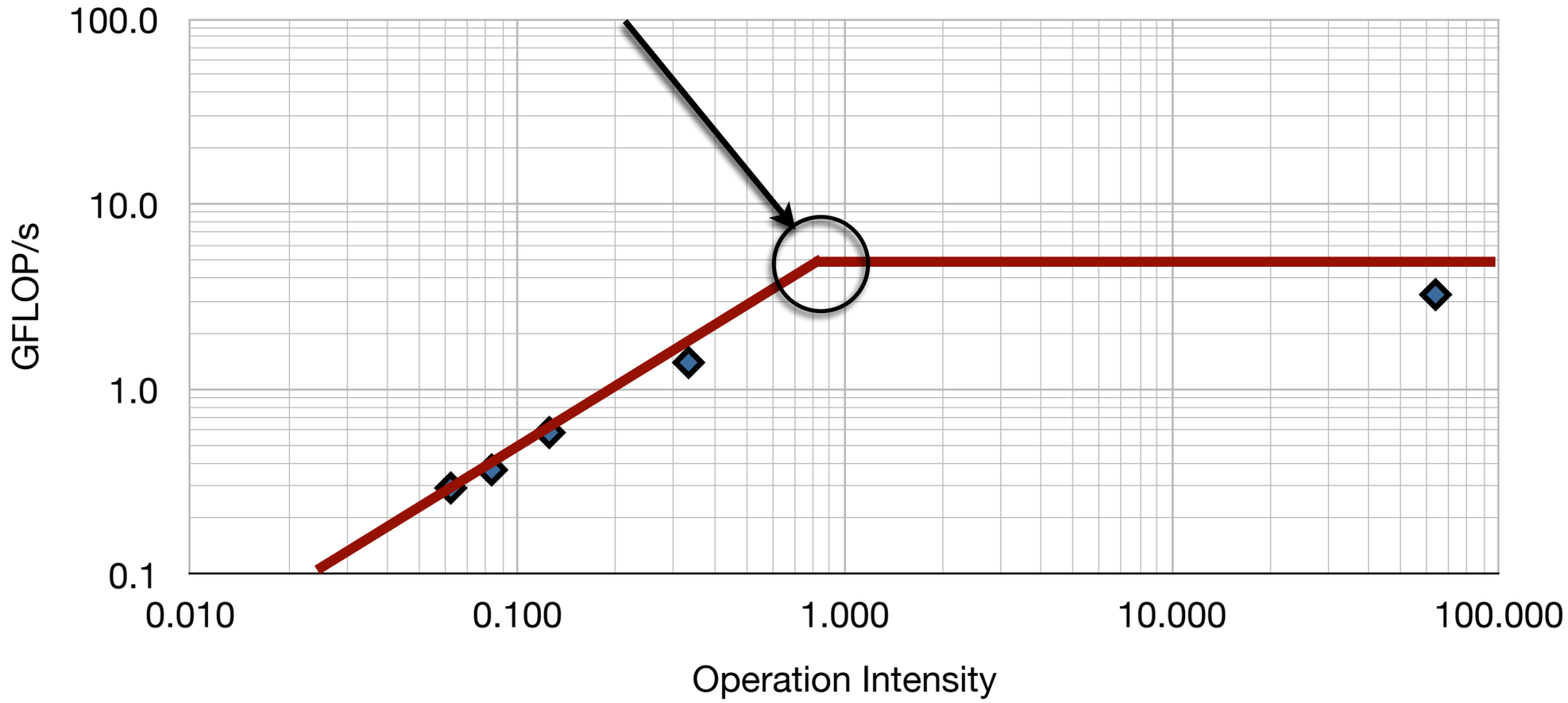
WRITE BACK CACHE

write on main memory only when you need to clean some level of cache



http://www.computerweekly.com/feature/Write-through-write-around-write-back-Cache-explained

The ridge point



- Ridge point **characterize the overall machine performance**

➡ Ridge point "to the left": it is relatively easy to get peak performance

➡ Ridge point "to the right": it is difficult to get peak performance

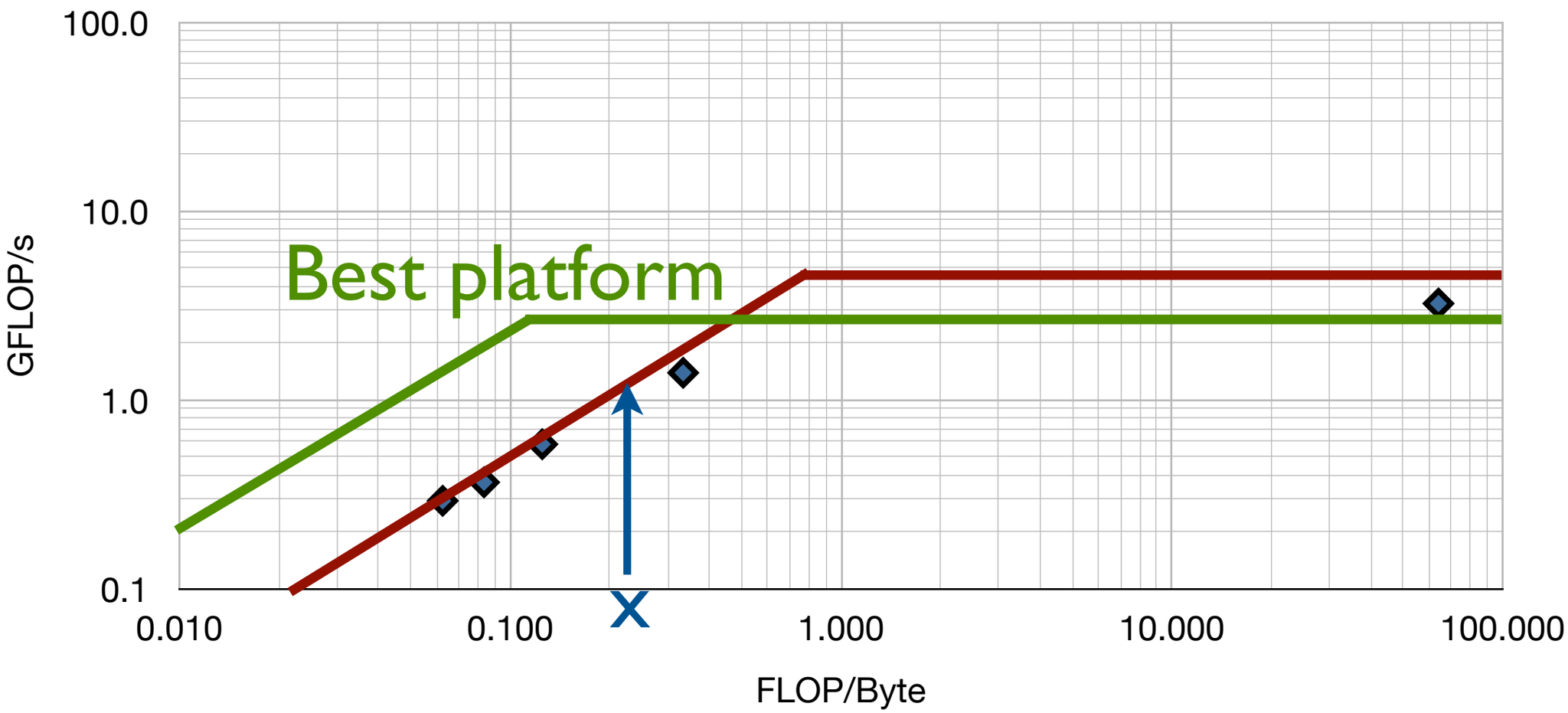
What does it mean "a ridge point to the right" anyway?

Production software

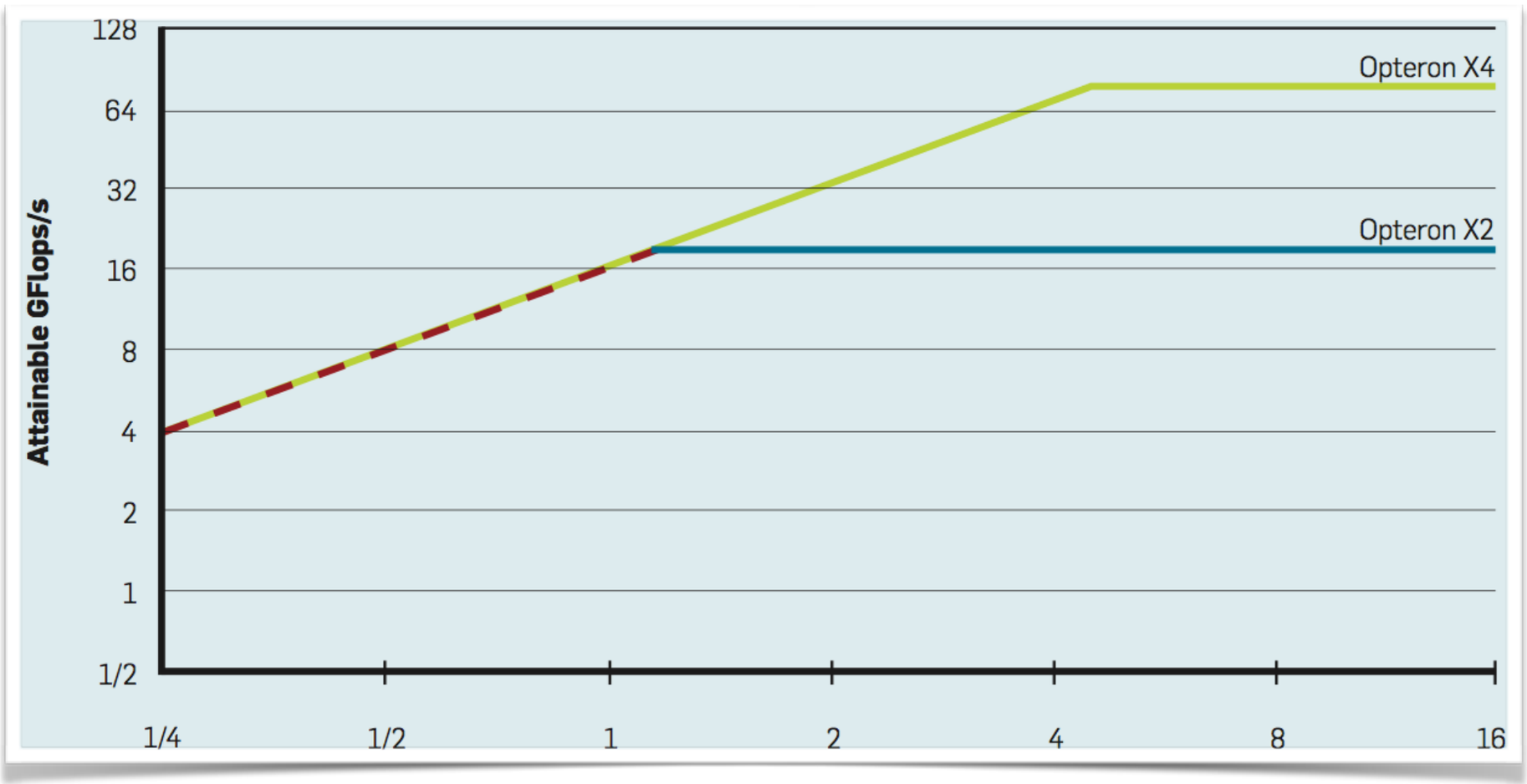
Assumption: production-ready software

- Limited set of algorithms
- Fixed set of kernels
- Fixed operational intensities

} ➡ Best hardware solution?



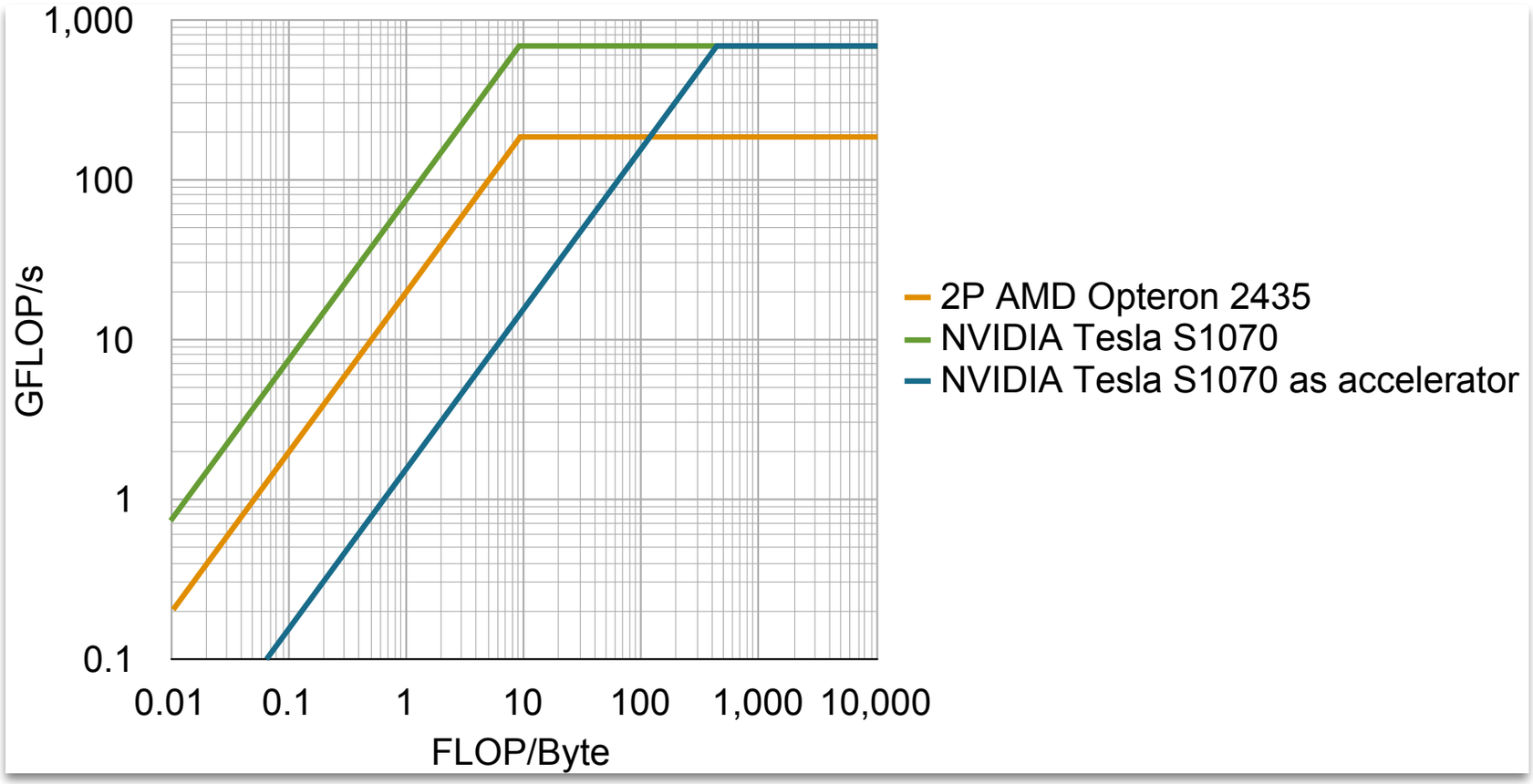
Is Moore worth?



It depends:

- On the ridge point
- On the operational intensity of the considered kernels

GPU as an "accelerator"



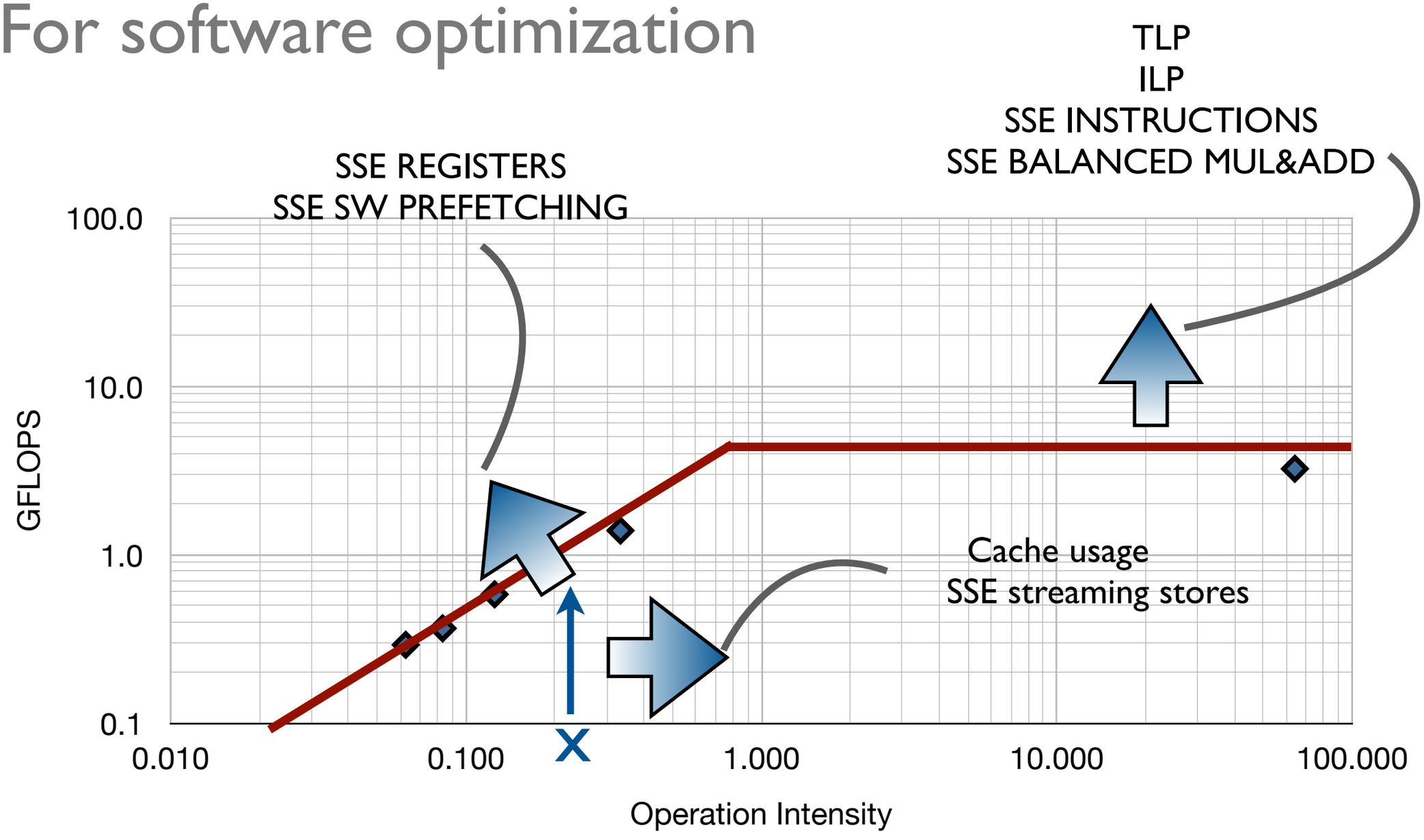
- Ridge point at 400 FLOP/Byte
- Almost no kernels can approach this value

➡ Move all data to the GPU (bad idea in general)

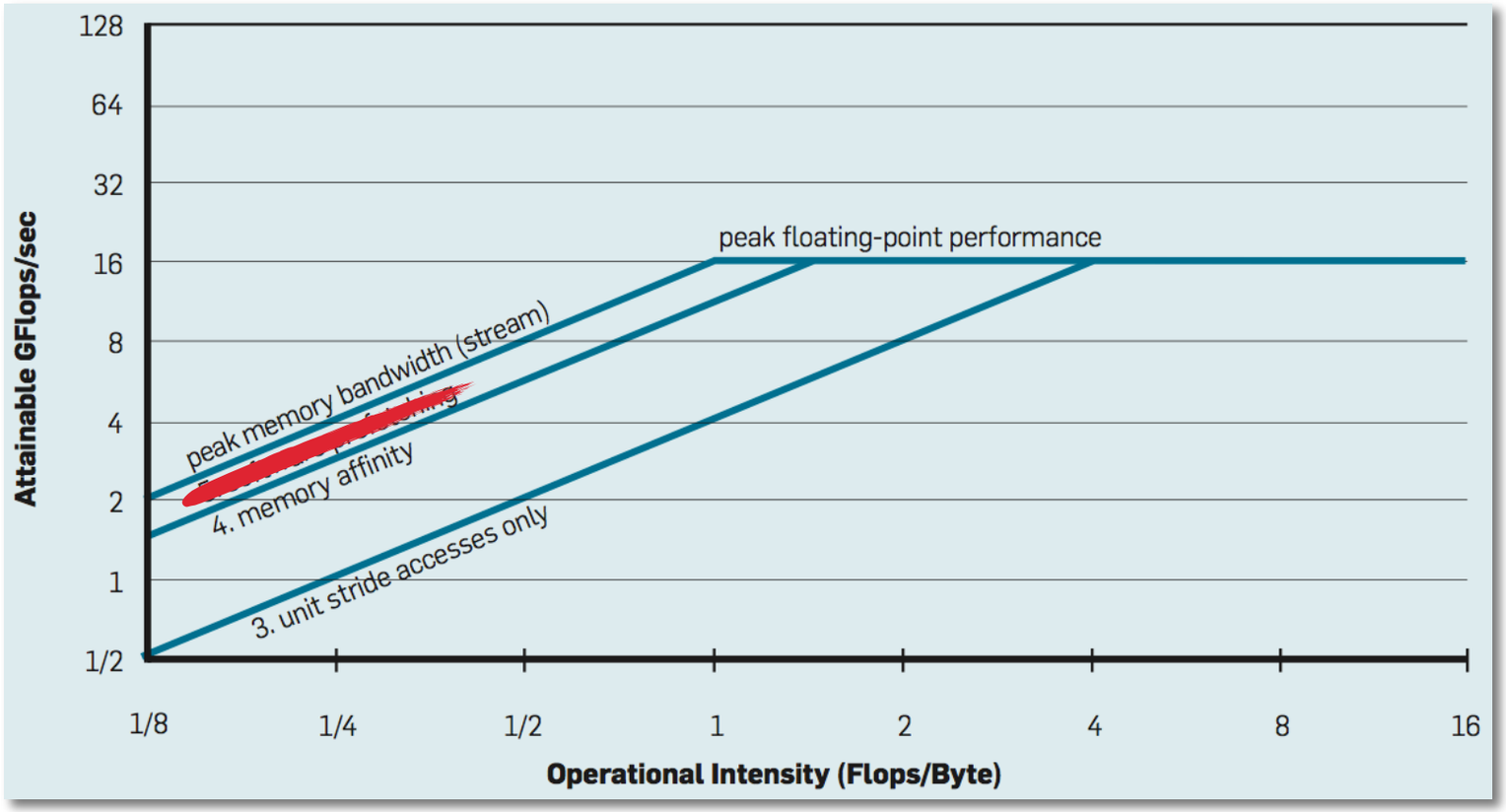
➡ Hide data transfers with GPU computation

The roofline model

For software optimization



Bandwidth ceilings



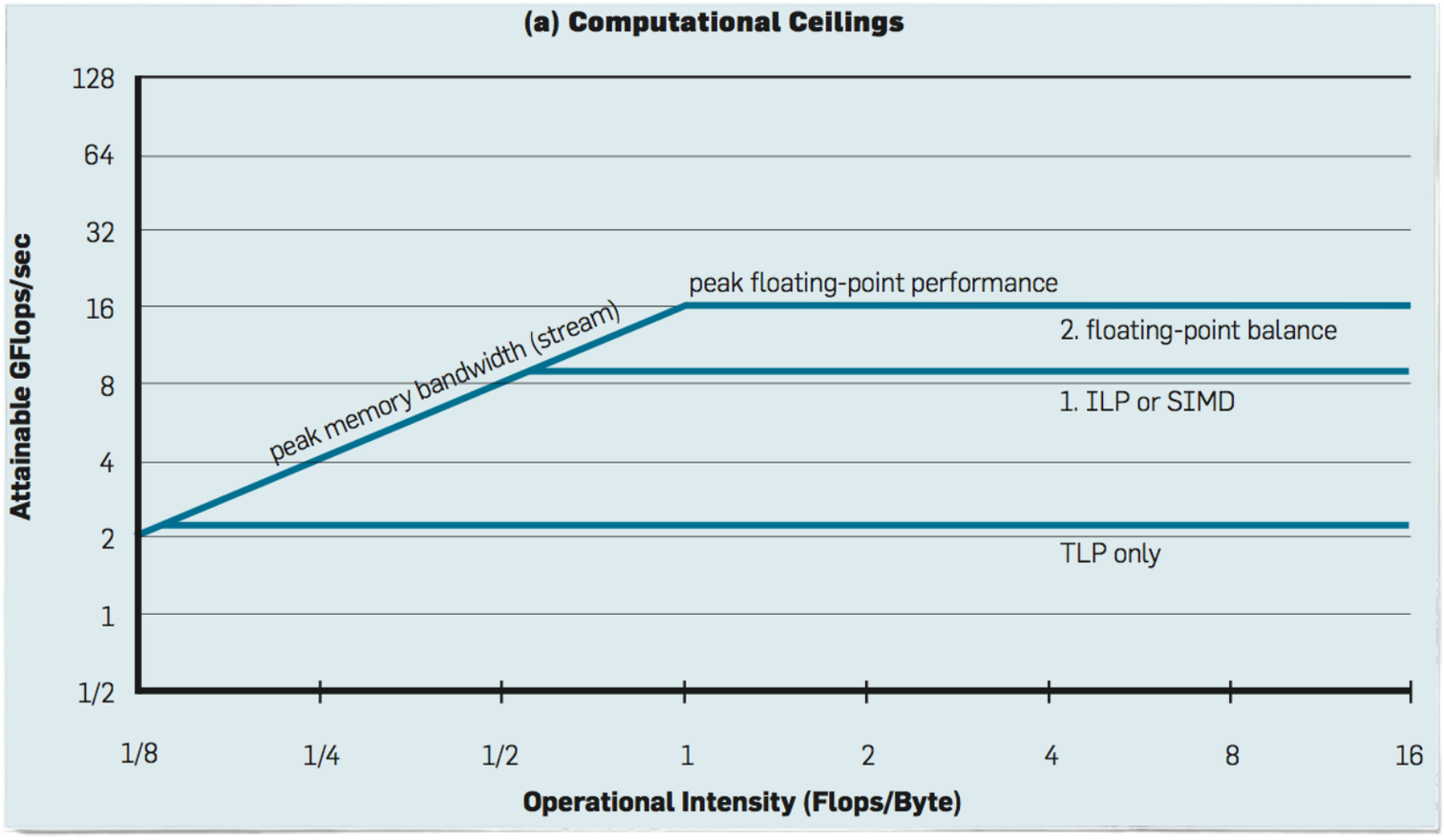
GPU:

- Coalesced access
- Non-temporal writes
- Use of textures

CPU:

- Unit-stride access
- NUMA-awareness
- Non-temporal writes

Performance ceilings



GPU optimization:

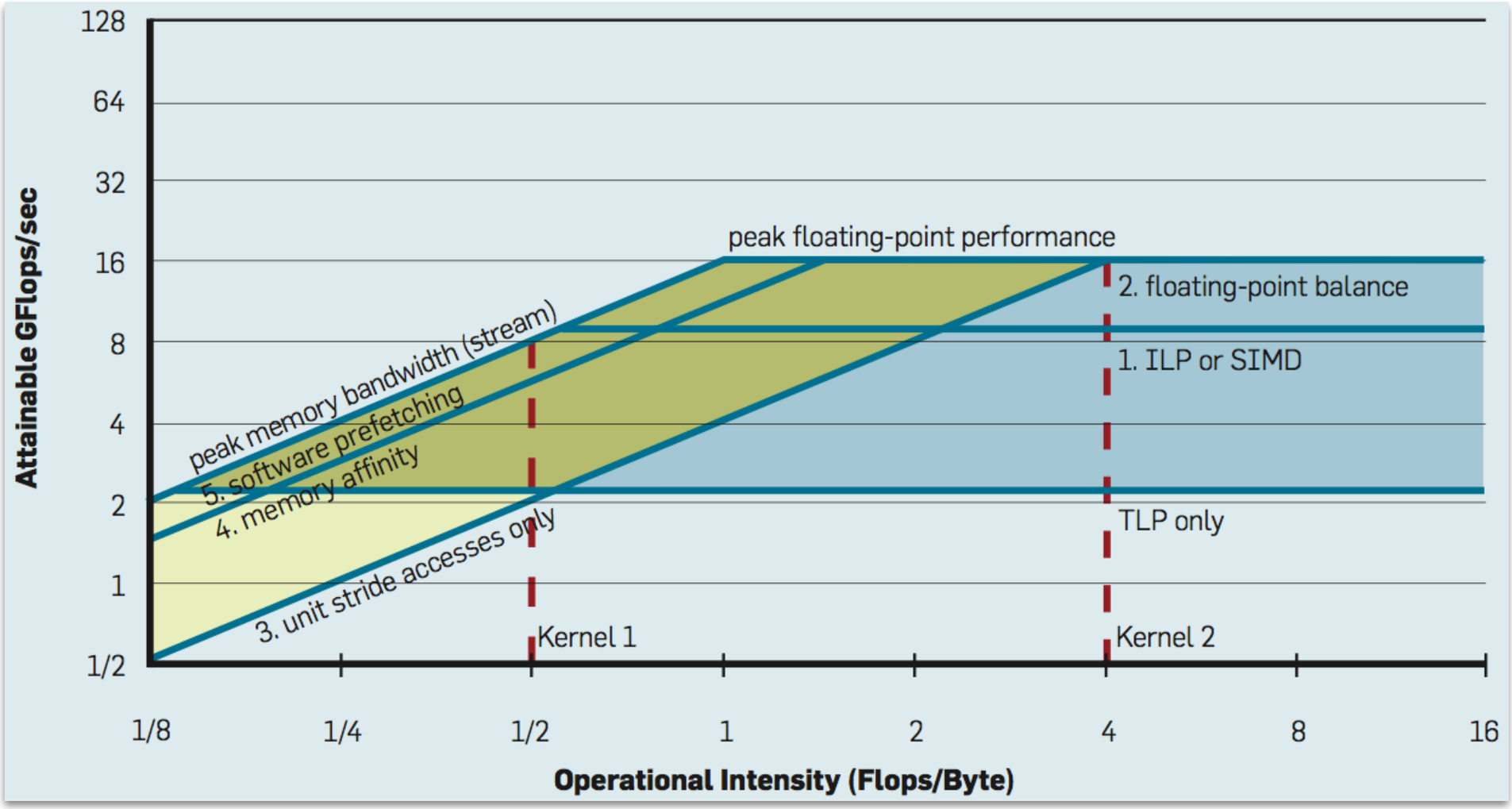
- Increase DLP
- Increase ILP
- Loop unrolling
- Use FMAD

- The roofline is an upper bound to performance
- Optimization steps should break performance ceilings

➡ What software optimization should we perform?

➡ In what order?

In what order?



- Some optimizations are not orthogonal
- Break the next performance ceiling
- Ceilings are ranked bottom-to-top
  - Bottom: little effort
  - Top: high effort OR inherently lacking

The first optimization step

For a kernel at the left of the ridge point:

- First step: ignore ceilings, maximize reuse

➡ Increased operational intensity

How to achieve this?

1. Minimize conflict cache misses (data reordering)
2. Minimize capacity cache misses
    - Computation reordering
    - Non-temporal writes/streaming stores

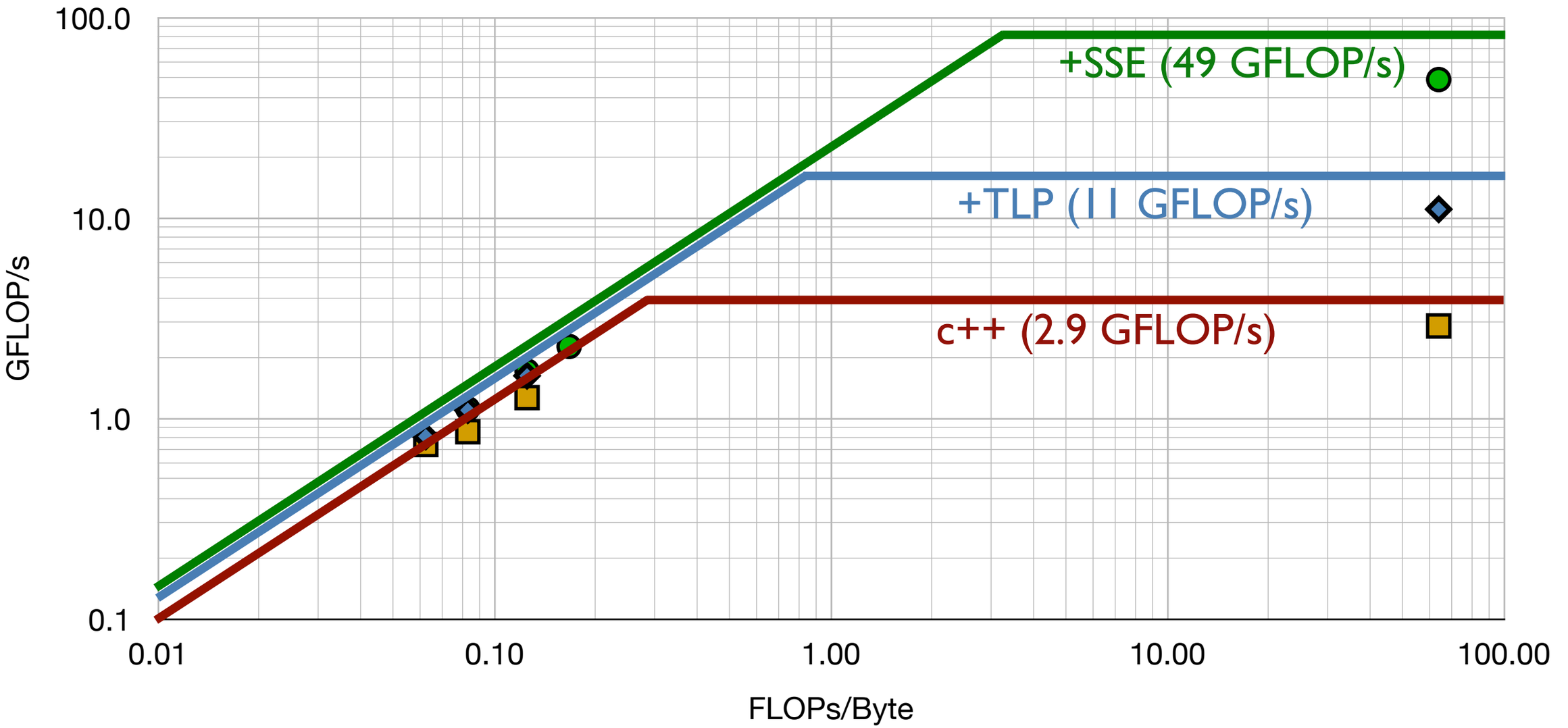
How to achieve reordering?

- Tiling
- Blocking
- Space filling curves

What is the upper bound?

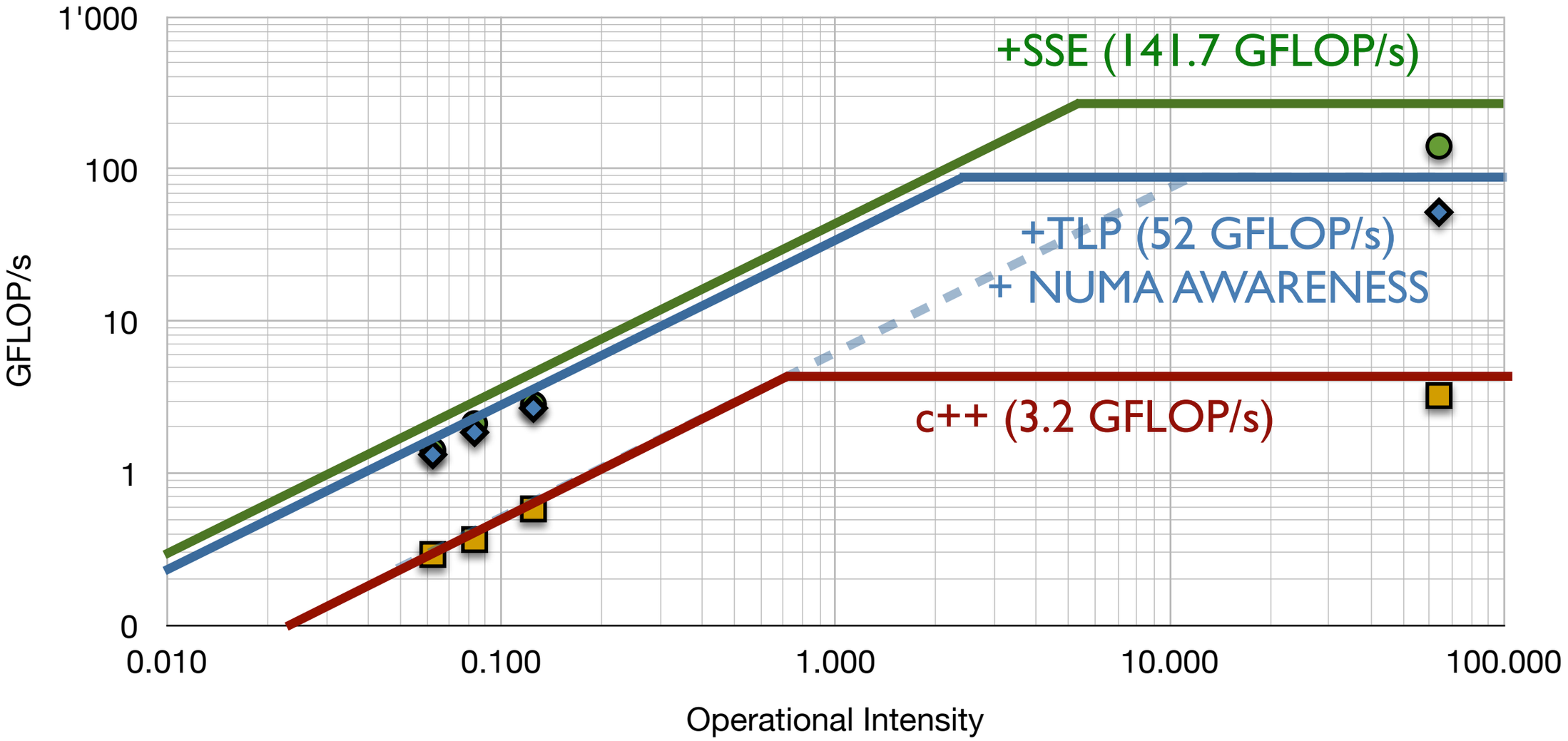
➡ Compulsory cache misses

Performance ceilings



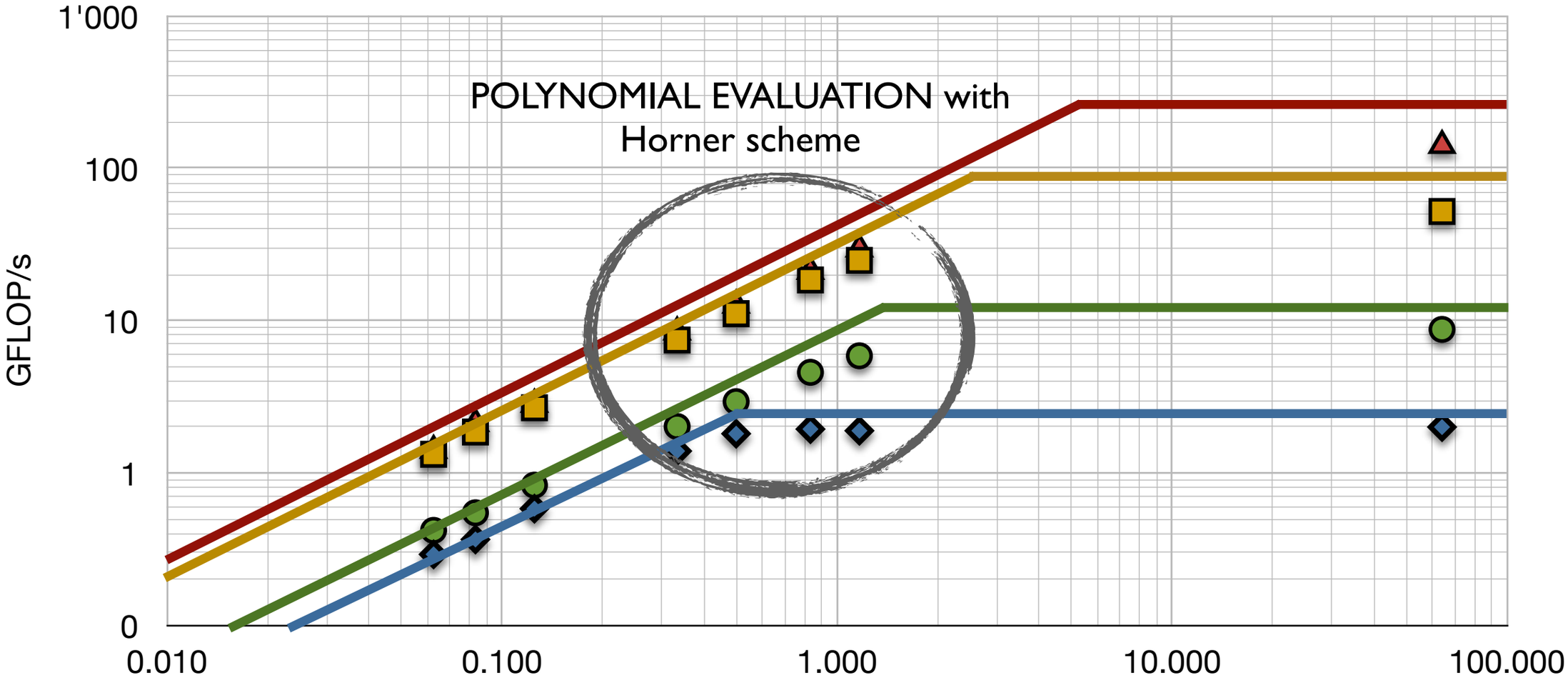
PERF. GAIN:17X

Performance ceilings



PERF. GAIN: 44X

Confirming the roofline



The roofline is there.
(for real)

Quiz (1)

- Does the roofline model account for caches and advanced memory types (eg. caches, textures, non-temporal writes)?

- When the operational intensity is low:
  by doubling the cache size, does the operational intensity increase?

- Does the model take into account long memory latencies?

Quiz (2)

- Does the roofline model have little to do with multi/ many-core computing?

- Does the FLOP/s roofline, ignoring the integer computation, possibly produce a wrong estimation of the performance?

- Is the roofline model limited to easily optimized kernels that never hit the cache?

Quiz (3)

- Is the roofline model limited to the performance analysis of kernels performing floating-point computations?

- Is the roofline model forcedly bound to the off-chip memory traffic?

Study case

Computational settings:

- Grid-based simulation of the diffusion process
- Two-dimensional problem
- Structured grid, uniform grid spacing
- Explicit time discretization

Computer settings:

- Grid data large enough to not fit in the cache
- Floating point arithmetic in single precision

$$u : [0,T] \times \Omega^2 \rightarrow R$$

$$\partial_t u - (\partial_{xx} + \partial_{yy})u = 0$$

$$u(t,x) = u_0(x), \;\; x \in \Gamma$$

$$\begin{aligned} \text{tmp} &:= \Delta\omega^n \\ \omega^{*,n} &:= \omega^n + \frac{\delta t}{2}\,\text{tmp} \\ \text{tmp} &:= \Delta\omega^{*,n} \\ \omega^{n+1} &:= \omega^n + \delta t\,\text{tmp}. \end{aligned}$$

Step 1:

$$\text{tmp}_{i,j} = a_0 \cdot u^n_{i,j} + a_1 \cdot (u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j-1} + u^n_{i,j+1})$$

Step 2:

$$u^{n+1}_{i,j} = u^n_{i,j} + a_2 \cdot (\text{tmp}^n_{i+1,j} + \text{tmp}^n_{i-1,j} + \text{tmp}^n_{i,j-1} + \text{tmp}^n_{i,j+1} - 4\text{tmp}_{i,j})$$

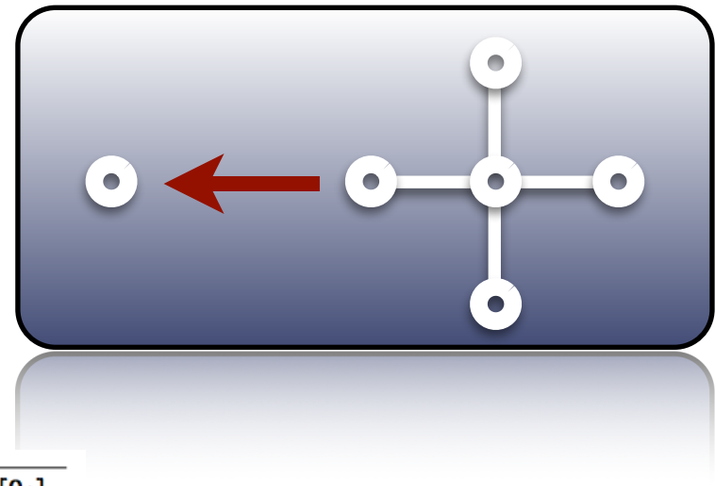
$$a_0 = 1 - 2\frac{dt}{h^2} \qquad a_1 = \frac{dt}{2h^2} \qquad a_2 = \frac{dt}{h^2}$$

Operational intensities

NO WB-WRITE ALLOCATE CACHE MISSES

NAIVE, STEP1: R = 6 FLOPS/(4*4 BYTES) = 0.375

TILED, STEP 1: R = (3/4)*(256/272) = 0.47



	Observed Peak [GB/s]	Theoretical Peak [GB/s]	Efficiency [%]
Xeon W3520	13.4	25.5	52
Opteron 8380	24.5	42.5	58
Opteron 2435	19.2	25.6	75
Tesla S1070	73.8	100.2	74

EXPECTED PERFORMANCE (SIMPLE MEM-LAYOUT):

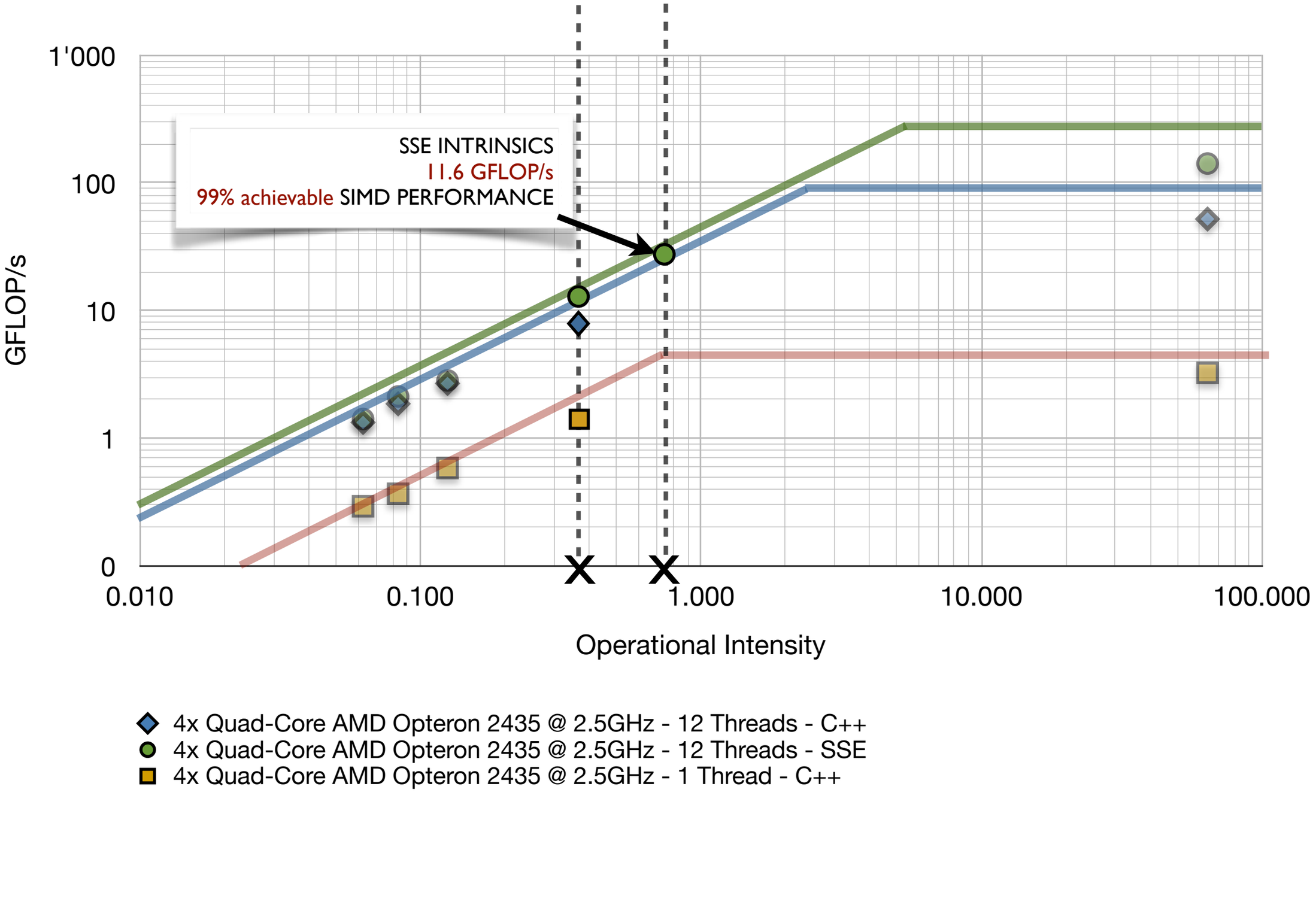
OPTERON 2435: 19.2*0.375 = **7.2** [GFLOP/S]

TESLA S1070: 73.8*0.375 = **27.7** [GFLOP/S]

TILING, EXPECTED PERFORMANCE:

OPTERON 2435: 19.2*0.47 = **9** [GFLOP/S]

CPU (similar) results



Results

EXPECTED PERFORMANCE (SIMPLE MEM-LAYOUT):

OPTERON 2435: 19.2*0.375 = 7.2 GFLOPS

TESLA S1070: 73.8*0.375 = 27.7 GFLOPS

CPU-TILING, EXPECTED PERFORMANCE:

OPTERON 2435: 19.2*0.47 = 9 GFLOPS

OBSERVED PERFORMANCE (SIMPLE MEM-LAYOUT):

OPTERON 2435: **6.6 [GFLOP/S]** (**91%** EFF.)

TESLA S1070: **30 [GFLOP/S]** (~**100%**, CACHING EFFECTS)

CPU-TILING, OBSERVED PERFORMANCE:

OPTERON 2435: **7.8 [GFLOP/S]** (**86%** EFF.), MORE COMPLEX CODE!

COMPILER OPTIMIZATIONS: LESS EFFECTIVE

Conclusions

- When is the roofline model useless?
  - x When you discuss performance in terms to time-to-solution.
- When is the roofline model crucial?
  - ➡ When you want to optimize your code (*data reuse, ceilings*)
  - ➡ To predict maximum achievable performance (*roofline, ridgepoint*)
  - ➡ To systematically assess your performance (*roofline, op. int.*)
- What do you do if all your kernels have a bad op. int.?
  - ~ Either live with it
  - ✓ Go back to equations, pick better discretizations/algorithms (leading to a higher op. int.)
  - ✓ Wanted: less simulation steps, but more costly (High order schemes)