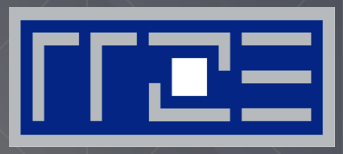
ERLANGEN REGIONAL COMPUTING CENTER



# MULTICORE ARCHITECTURES

Georg Hager, Jan Treibig, Gerhard Wellein

DIMACS Workshop on Multicore and Cryptography

July 21, 2014

Stevens Institute of Technology, Hoboken, NJ

# A conversation

From a student seminar on “Efficient programming of modern multi- and manycore processors”

**Student**: I have implemented this algorithm on the GPGPU, and it solves a system with 26546 unknowns is 0.12 seconds, so it is really fast.

**Me**: What makes you think that 0.12 seconds is fast?

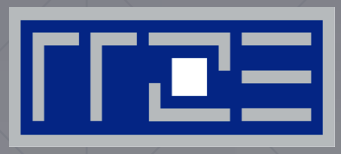
**Student** (very confident): It is fast because my baseline C++ code on the CPU is about 20 times slower.

# A statement

High performance computing is computing at a bottleneck

This does not mean that there is no faster way to solve the problem!

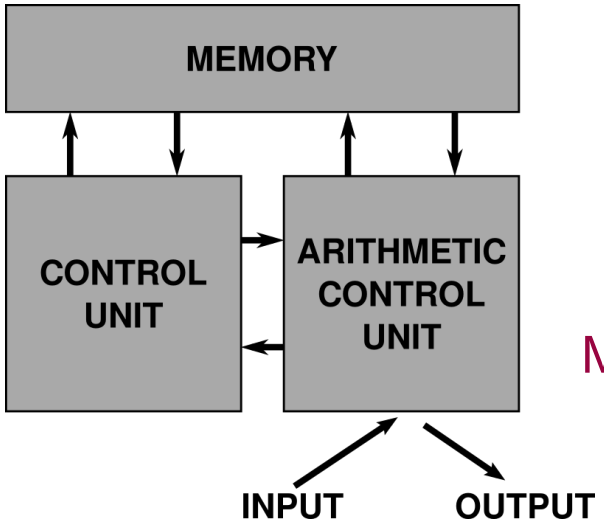
# INTRODUCTION: MODERN COMPUTER ARCHITECTURE



The stored program computer and its inherent bottlenecks

# Computer Architecture

*The evil of hardware optimizations*




Stored program computer:
Flexible, but optimization is hard!

Architect's view:
Make the common case fast !

**EDSAC 1949**

- Provide improvements for relevant software
  - What are the technical opportunities?
  - Economical concerns
  - Multi-way special purpose

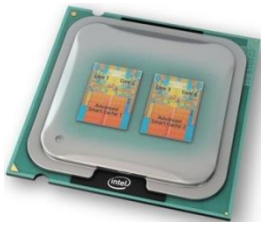
**What is your relevant aspect of the architecture?**

# Hardware-Software Co-Design?

*From algorithm to execution*

The machine view:

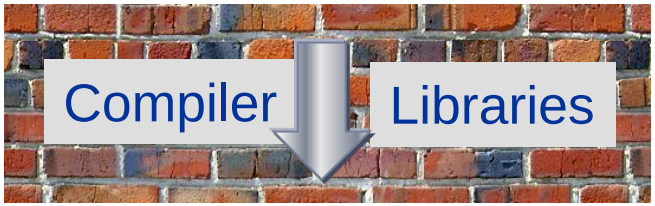
ISA (Machine code)



The user's view:

Algorithm

Programming language




**Hardware = Black Box**

# Basic Resources

*Instruction throughput and data movement*

1. Instruction execution

   This is the primary resource of the processor. All efforts in hardware design are targeted towards increasing the instruction throughput.

   Instructions are the concept of "work" as seen by processor designers.

   Not all instructions count as "work" as seen by application developers!

   Example: Adding two arrays

```
do i=1, N
  A(i) = A(i) + B(i)
enddo
```

Processor work:

```
LOAD r1 = A(i)
LOAD r2 = B(i)
ADD r1 = r1 + r2
STORE A(i) = r1
INCREMENT i
BRANCH → top if i<N
```

User work:

**N** ops (ADDs)

# Basic Resources

*Instruction throughput and data movement*

2. Data transfer

   Data transfers are a consequence of instruction execution and therefore a secondary resource. Maximum bandwidth is determined by the request rate of executed instructions and technical limitations (bus width, speed).

   Example: Adding two arrays

```
do i=1, N
  A(i) = A(i) + B(i)
enddo
```

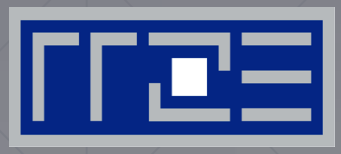
Data transfers:
8 byte: `LOAD r1 = A(i)`
8 byte: `LOAD r2 = B(i)`
8 byte: `STORE A(i) = r2`

Sum: **24 byte**

Crucial question: What is the bottleneck?

- Data transfer?
- Code execution?

# INTRODUCTION: MODERN COMPUTER ARCHITECTURE

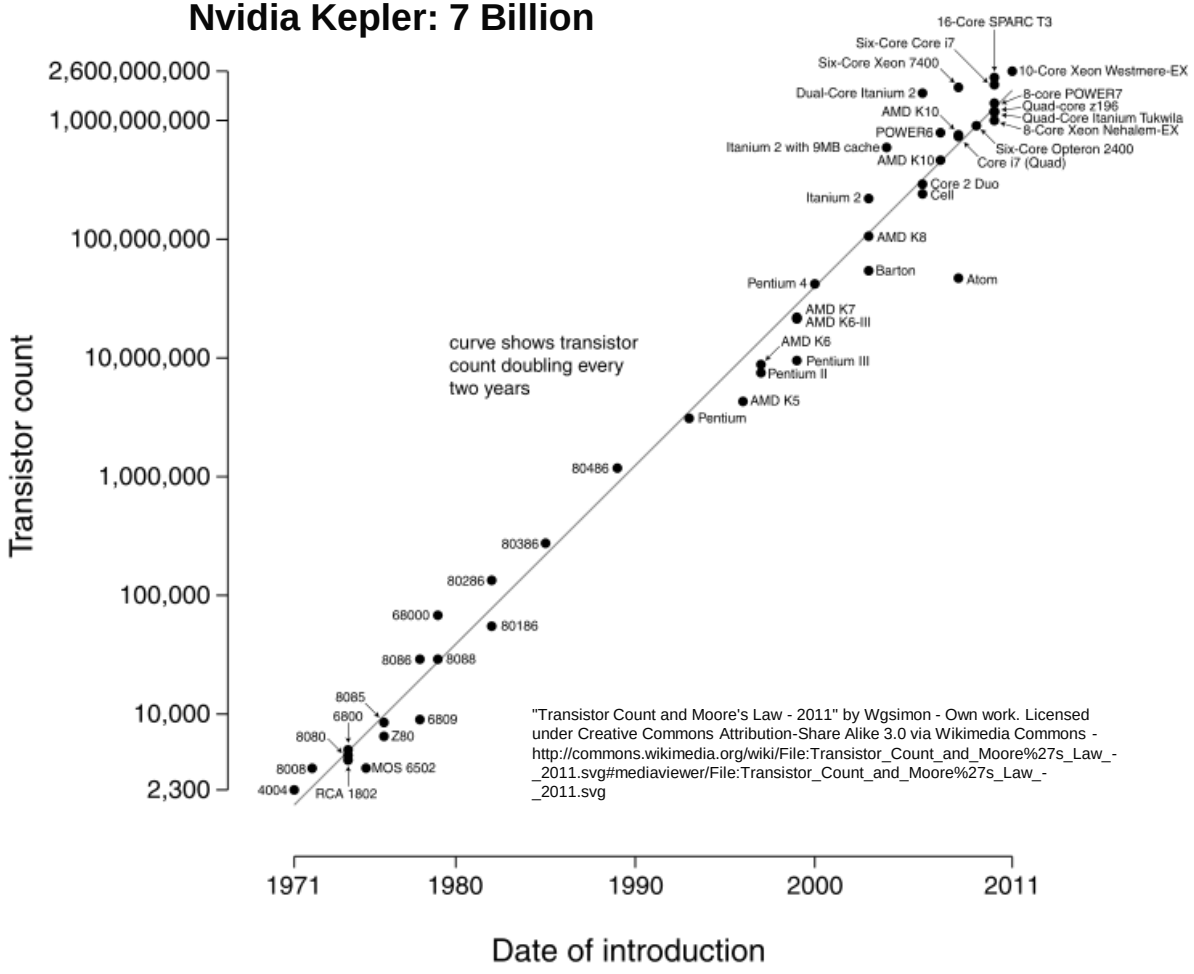


Multi-cores – where and why

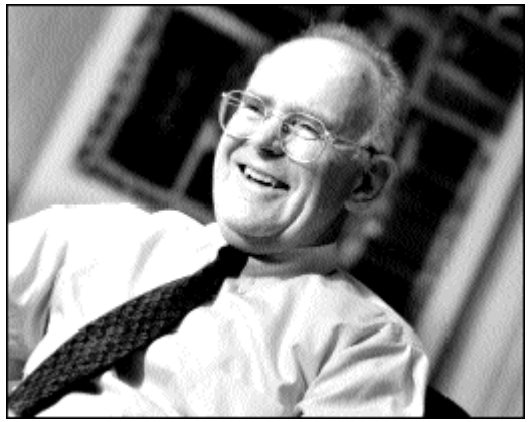
# Moore's law

**Intel Sandy Bridge EP: 2.3 Billion**

**Nvidia Kepler: 7 Billion**



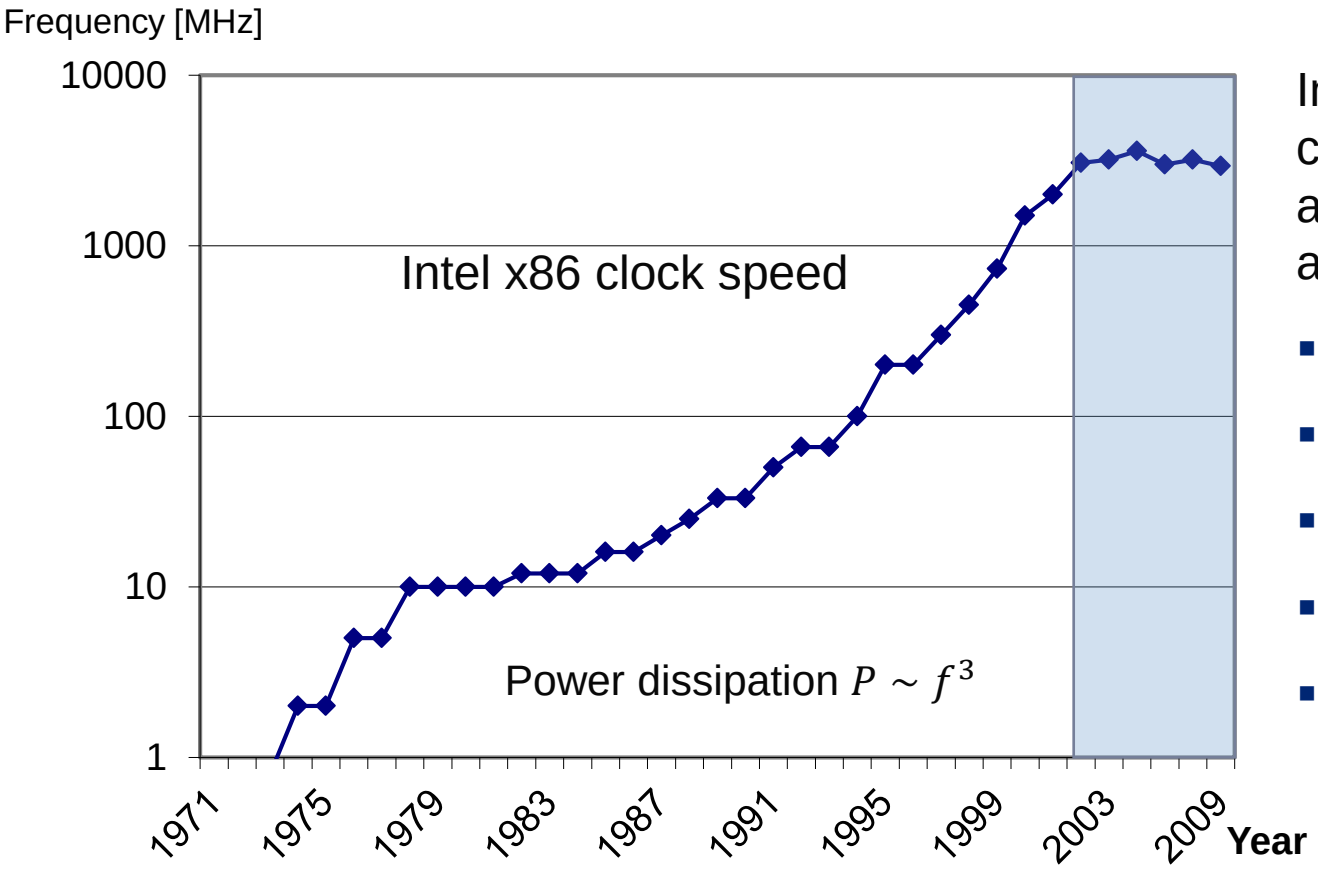

"Transistor Count and Moore's Law - 2011" by Wgsimon - Own work. Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg#mediaviewer/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg



1965: G. Moore claimed #transistors on "microchip" doubles every 12-24 months

# Moore's law: faster cycles and beyond

Moore's law → transistors are getting smaller → run them faster
Faster clock speed → Higher Throughput (Ops/s)




Increasing transistor count and clock speed allows / requires architectural changes:

- Pipelining
- Superscalarity
- SIMD / Vector ops
- Multi-Core/Threading
- Complex on-chip caches

# Multi-Core: Intel Xeon 2600 (2012)

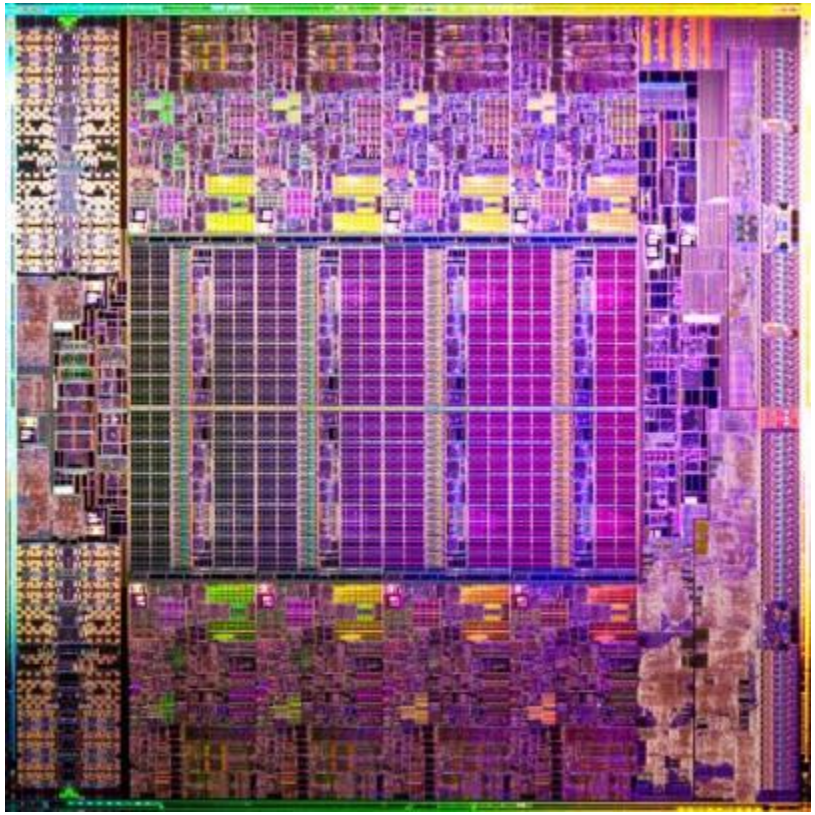
Xeon 2600 “Sandy Bridge EP”:
8 cores running at 2.7 GHz (max 3.2 GHz)

Simultaneous Multithreading
→ reports as 16-way chip

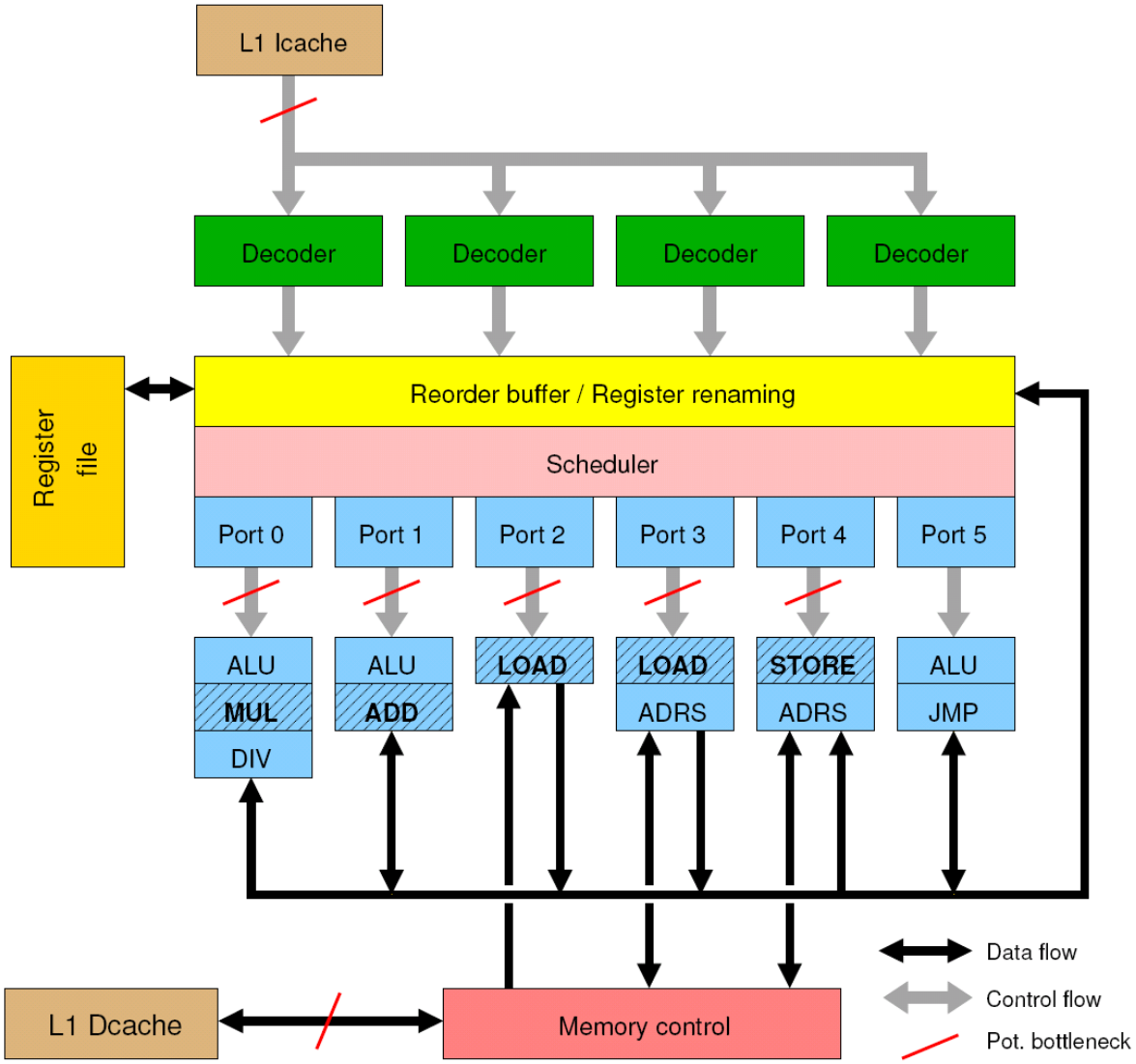
2.3 Billion Transistors / 32 nm

Die size: 435 mm$^2$


2-socket server

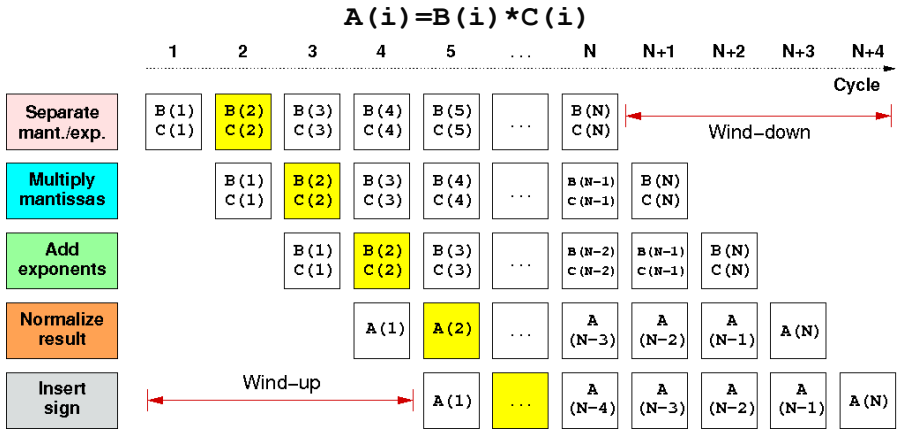


# In-core code execution



# Basics of superscalar pipelined execution

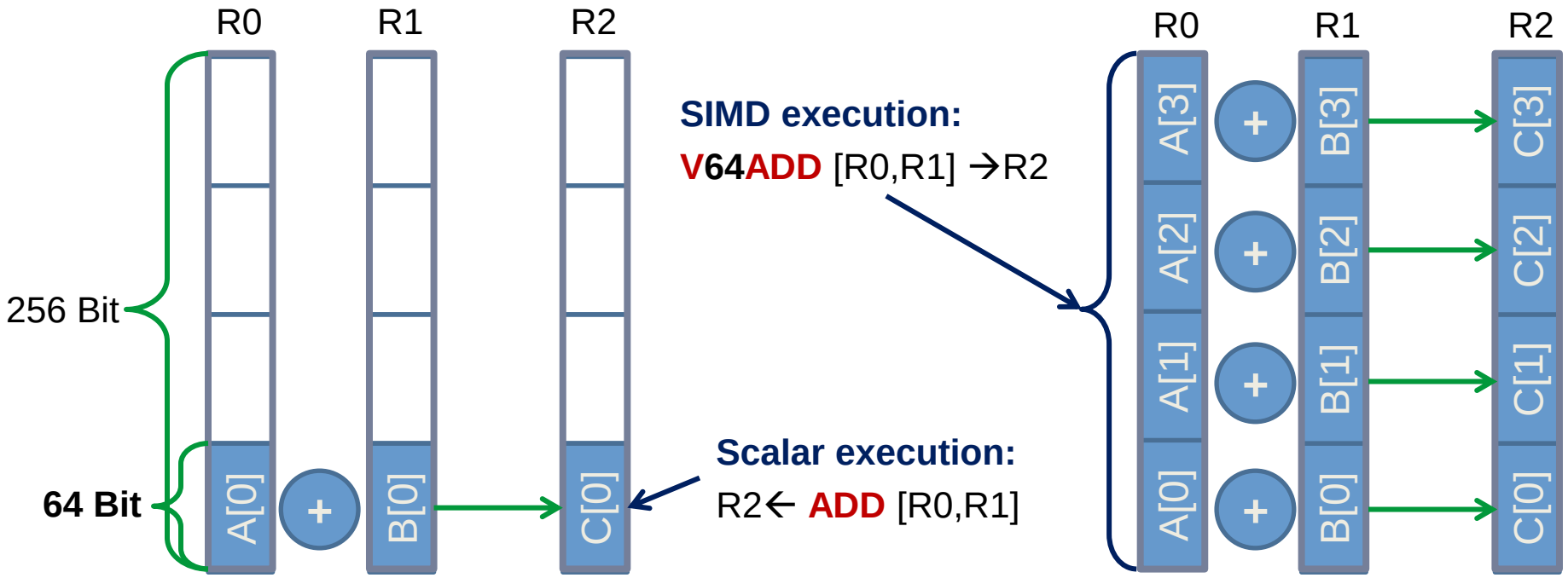
*Instruction-level parallelism (ILP)*




- (Almost) all execution units are pipelined
  - Throughput: minimum cycles per retired instruction
  - Latency: cycles for a single instruction end-to-end
  - Dependencies → stalls (“bubbles”)
- Multiple pipelines can work in parallel
  - “Superscalarity”
  - Maximum sustained throughput may be a bottleneck
- Out-of-order execution can automatically fill bubbles
  - Instructions executed when operands are available
- Hyperthreading (SMT) may do the same
  - Independent threads on same core may fill each other’s bubbles

# Core details: SIMD processing

- Single Instruction Multiple Data (SIMD) operations allow the concurrent execution of the same operation on “wide” registers
- x86 SIMD instruction sets: SSE (128 bit), AVX (256 bit)
- SIMD implements in-core data parallelism → fewer instructions for the same amount of work



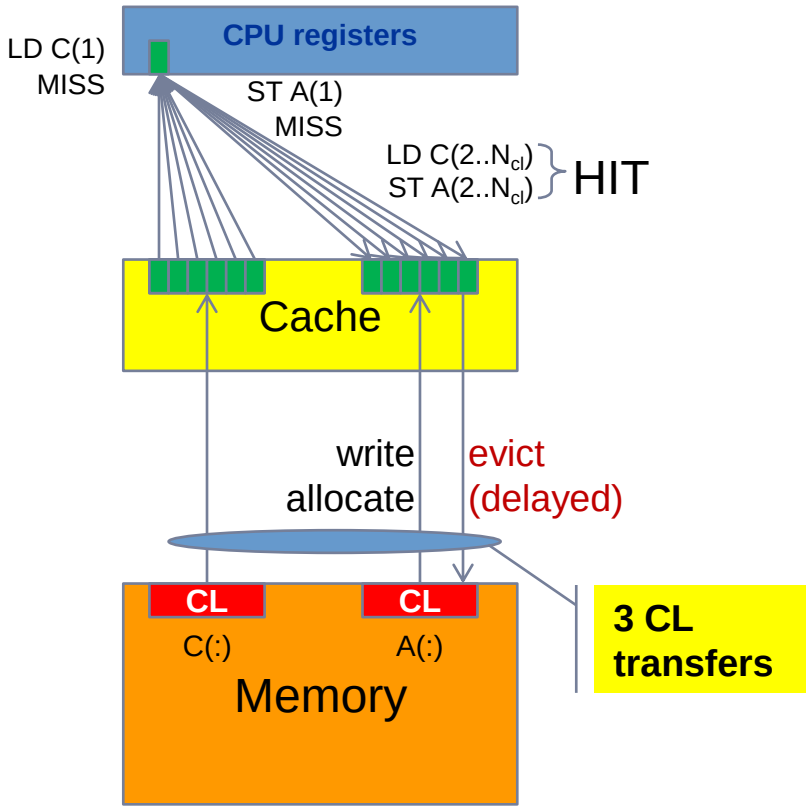
# Registers and caches:

*Data transfers in a memory hierarchy*

Caches help with getting instructions and data to the CPU “fast”

→ How does data travel from memory to the CPU and back?

- Remember: Caches are organized in cache lines (e.g., 64 bytes)
- Only complete cache lines are transferred between memory hierarchy levels (except registers)
- MISS: Load or store instruction does not find the data in a cache level
  → CL transfer required

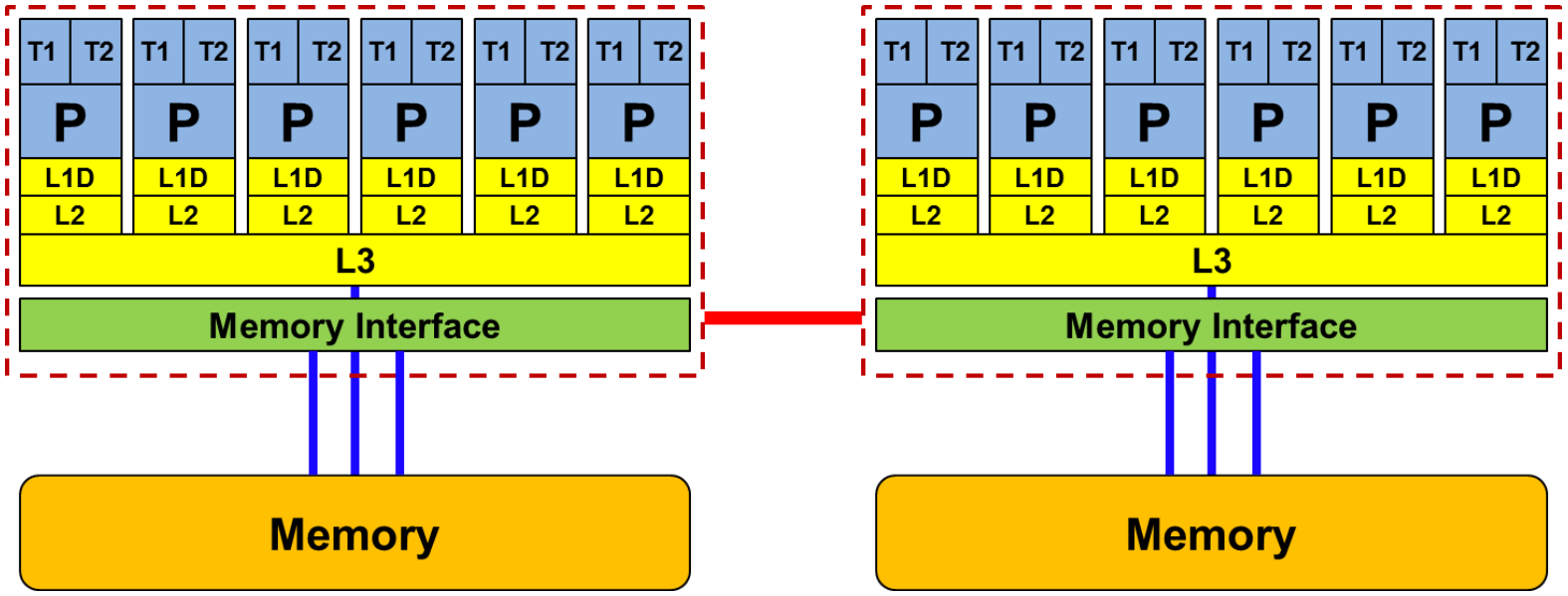



Example: Array copy `A(:)=C(:)`

# Multiple cores and the memory bottleneck

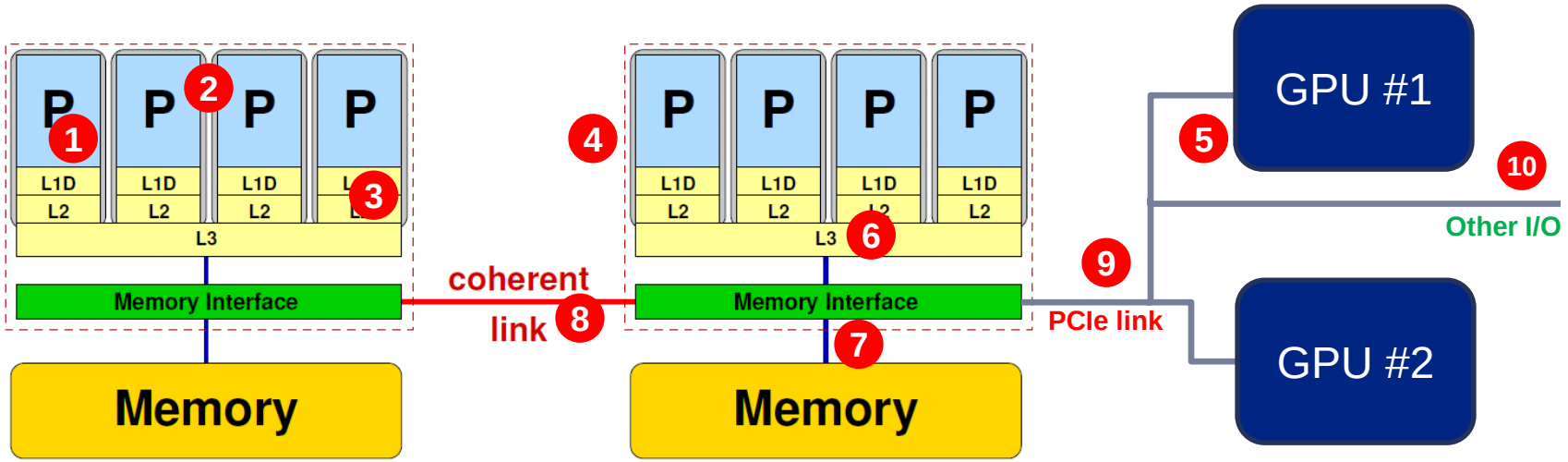
Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

Multi-socket servers: scalable bandwidth at the price of ccNUMA architectures → *Where does my data finally end up?*



# Parallelism in a modern compute node

## Parallel and shared resources within a shared-memory node




### Parallel resources:

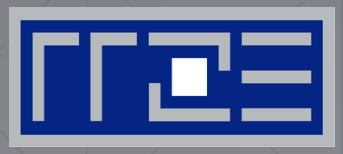
- Execution/SIMD units 1
- Cores 2
- Inner cache levels 3
- Sockets / ccNUMA domains 4
- Multiple accelerators 5

### Shared resources:

- Outer cache level per socket 6
- Memory bus per socket 7
- Intersocket link 8
- PCIe bus(es) 9
- Other I/O resources 10

**Which of these resources are critical for your code?**

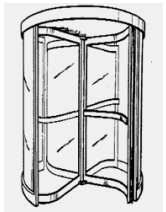
# PERFORMANCE MODELING



## The Roofline Model

# Prelude: Modeling customer dispatch in a bank

Revolving door throughput:
$b_S$ [customers/sec]



Processing capability:
$P_{max}$ [tasks/sec]

Intensity:
$I$ [tasks/customer]

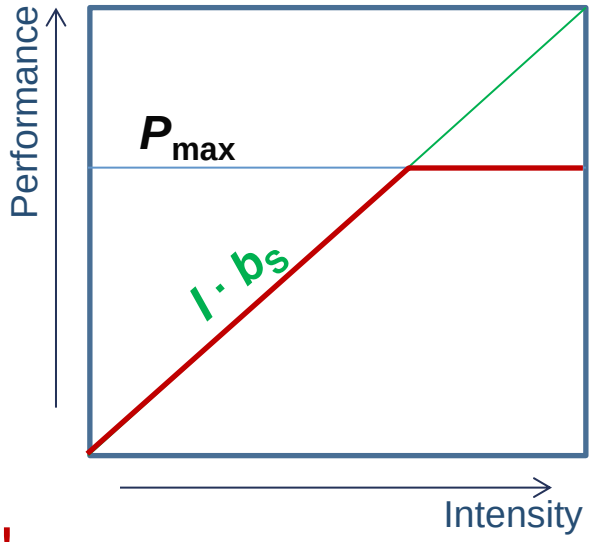
# Prelude: Modeling customer dispatch in a bank

How fast can tasks be processed? $P$ [tasks/sec]

The bottleneck is either

- The service desks (max. tasks/sec): $P_{\max}$
- The revolving door (max. customers/sec): $I \cdot b_S$

$$P = \min(P_{\max}, I \cdot b_S)$$




This is the “Roofline Model”

- High intensity: P limited by “execution”
- Low intensity: P limited by “bottleneck”

The model is **optimistic** – *P* is like “lightspeed”!

# The Roofline Model[1,2]

*Loop-based performance modeling*

1. $P_{max}$ = Applicable peak performance of a loop, assuming that data comes from L1 cache (this is not necessarily $P_{peak}$)

2. $I$ = Computational intensity ("work" per byte transferred) over the slowest data path utilized ("the bottleneck")
   - Code balance $B_C = I^{-1}$

3. $b_S$ = Applicable peak bandwidth of the slowest data path utilized

Expected performance:

$$P = \min(P_{\max}, I \cdot b_S)$$

[1] W. Schönauer: Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers. (2000)
[2] S. Williams: Auto-tuning Performance on Multicore Computers. UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)

# Aplying the Roofline Model

1. Identify the time-consuming loop constructs in your code (profiling)
2. Define a suitable metric for "work" and determine $P_{max}$
3. Answer the question "What part of the data comes from where?"
4. Identify the relevant data transfer bottleneck in the memory hierarchy & determine $I$
5. Apply $P = \min(P_{\max}, I \cdot b_S)$

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 70.45     5.14     5.14 26074562     0.00     0.00  substitute
 26.01     7.03     1.90  4000000     0.00     0.00  map
  3.72     7.30     0.27      100     2.71    73.03  shuffle
```

$$P_{\max} = \frac{16 \text{ substitutions}}{32 \text{ cy}} \cdot 3.0 \frac{\text{Gcy}}{\text{s}} = 1.5 \frac{\text{G subst.}}{\text{s}}$$

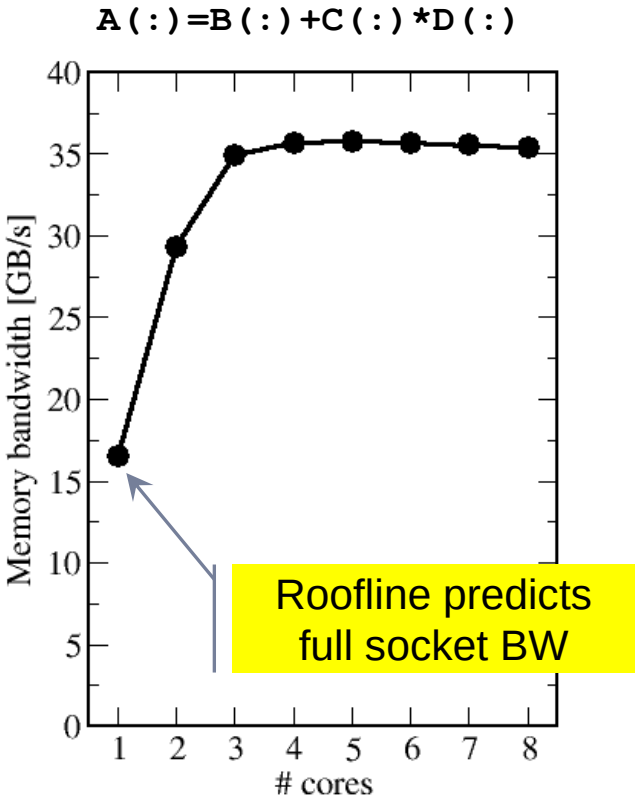
| Level | Bytes / subst. |
|---|---|
| L1 | 32+32 |
| L2 | 32 |
| L3 | 32 |
| **Memory** | **32** |

$$\rightarrow I = \frac{1}{32} \frac{\text{subst.}}{\text{byte}}$$

$$P = \min\left(1.5 \frac{\text{G subst.}}{\text{s}}, \frac{1}{32} \frac{\text{subst.}}{\text{byte}} \cdot 8 \frac{\text{GByte}}{\text{s}}\right) = \mathbf{0.25} \frac{\mathbf{G\ subst.}}{\mathbf{s}}$$

# Shortcomings and limitations of the Roofline Model

- All data accesses are assumed to come at no latency cost – bandwidth is the only limitation
  - Erratic/indexed data access may break this assumption
- Data transfers and computation overlap perfectly
  - Good assumption for multi-core, not true for single core
- Relevant data paths can be saturated (used with full bandwidth)
  - Good assumption for multi-core and main memory. Not so good for caches and single-core




G. Hager et al.: *Exploring performance and power properties of modern multicore chips via simple machine models*. Concurrency and Computation: Practice and Experience (2013). DOI: 10.1002/cpe.3180

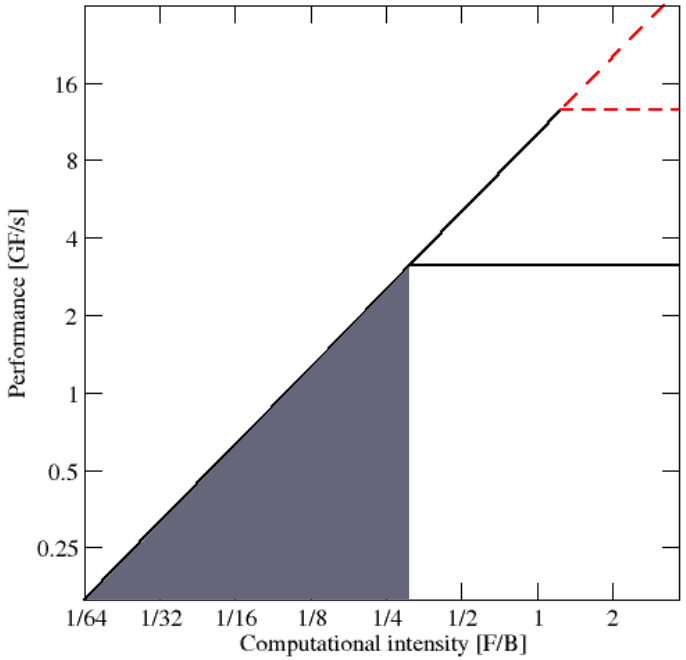
# Factors to consider in the Roofline Model

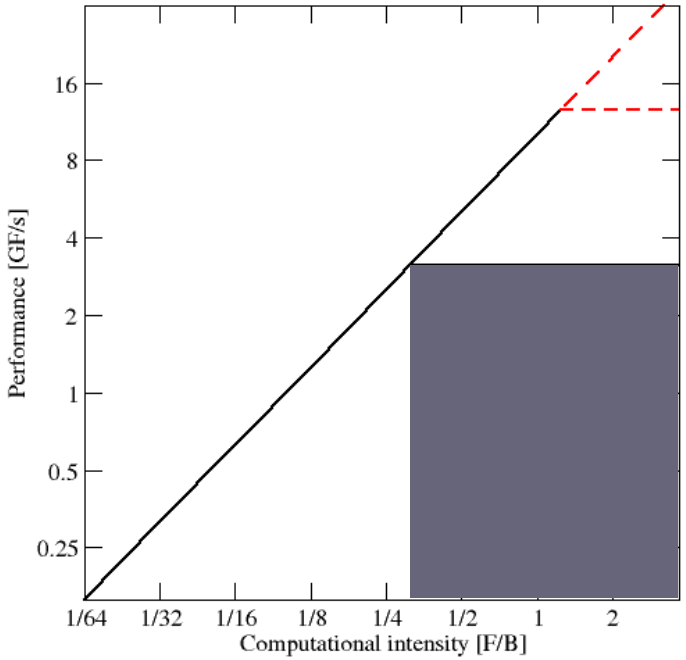
## Bandwidth-bound (simple case)

- Accurate traffic calculation (write-allocate, strided access, …)
- Practical ≠ theoretical BW limits
- Erratic access patterns

## Core-bound (may be complex)

- Multiple bottlenecks: LD/ST, arithmetic, pipelines, SIMD, execution ports
- Limit is linear in # of cores



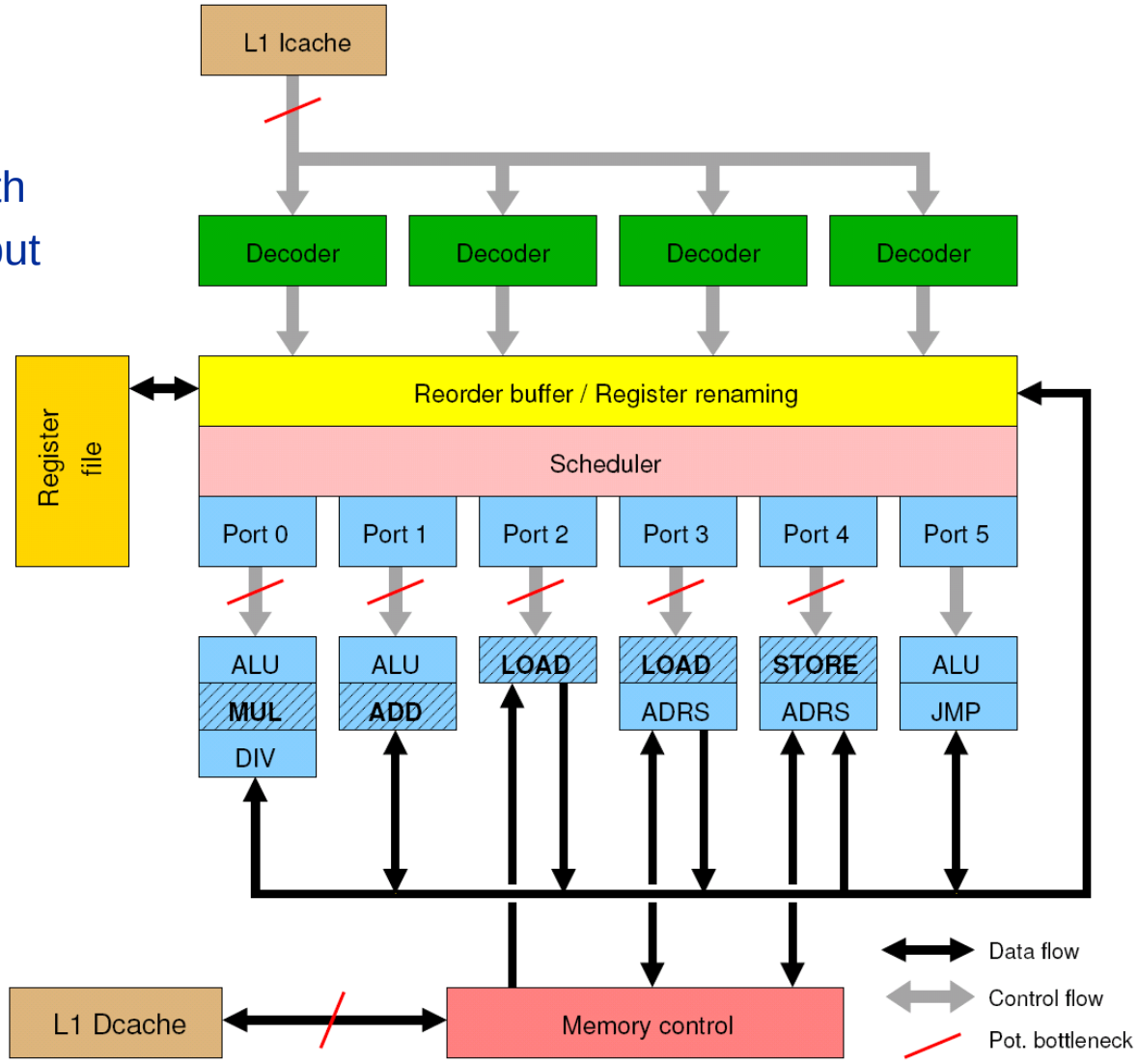



# Complexities of in-core execution

## Possible bottlenecks:

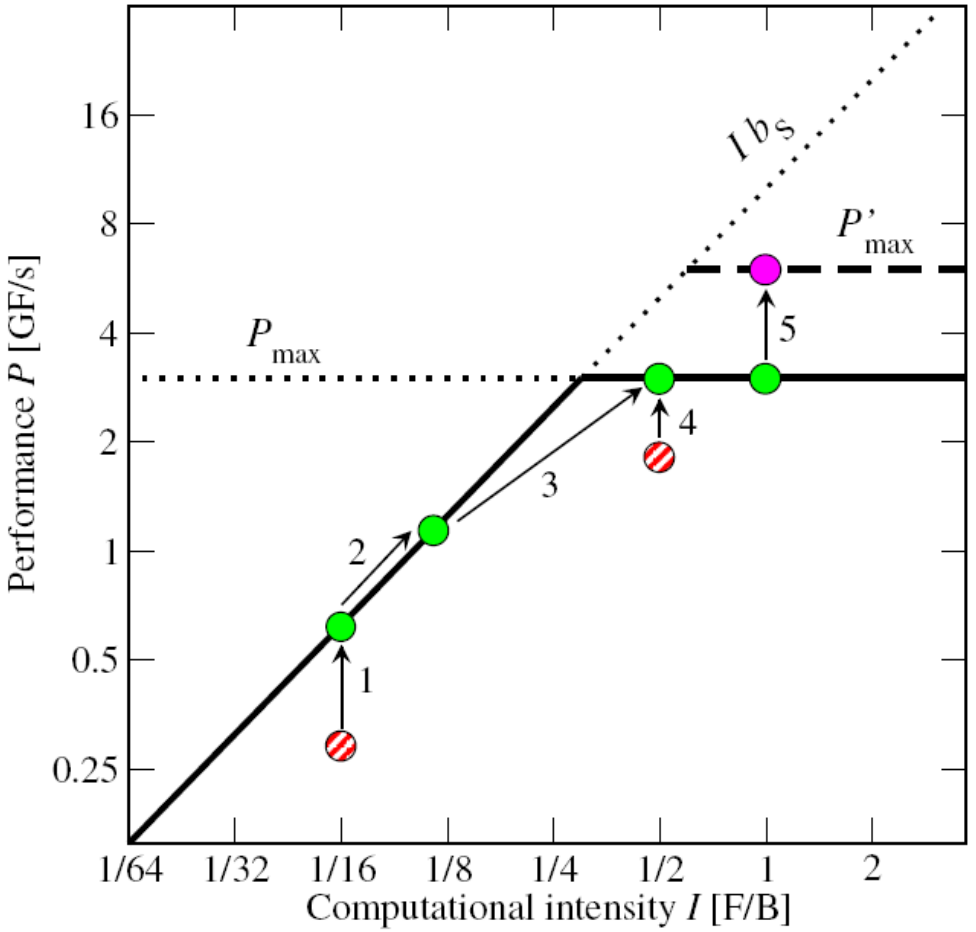
- L1 Icache (LD/ST) bandwidth
- Decode/retirement throughput
- Port contention (direct or indirect)
- Arithmetic pipeline stalls (dependencies)
- Overall pipeline stalls (branching)
- L1 Dcache bandwidth (LD/ST throughput)
- Scalar vs. SIMD execution
- …

- Register pressure
- Alignment issues



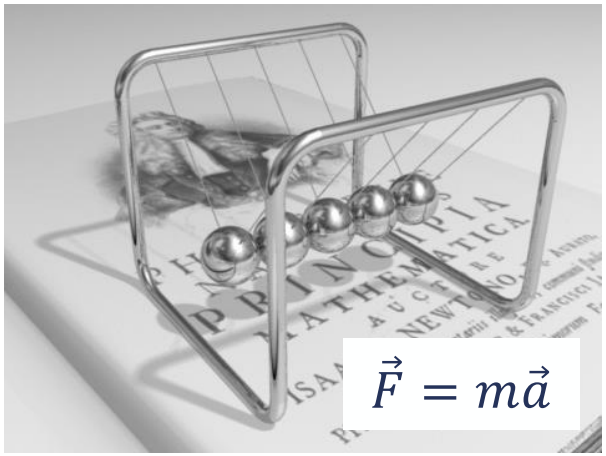
# Typical code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
2. Increase intensity to make better use of BW bottleneck
3. Increase intensity and go from memory-bound to core-bound
4. Hit the core bottleneck by good serial code
5. Shift $P_{max}$ by accessing additional hardware features (e.g., SIMD)



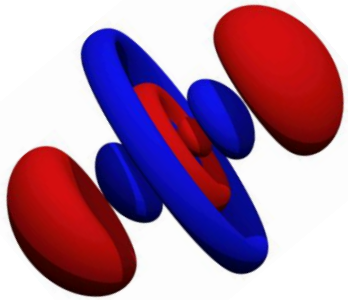
# Why building models? An example from physics

Newtonian mechanics



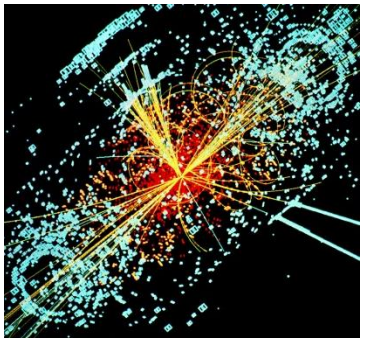

Fails @ small scales!

Nonrelativistic quantum mechanics



$$i\hbar\frac{\partial}{\partial t}\psi(\vec{r},t) = H\psi(\vec{r},t)$$

**Fails @ even smaller scales!**



**Relativistic quantum field theory**

$$U(1)_Y \otimes SU(2)_L \otimes SU(3)_c$$

**Consequences**

- If models fail, we learn more
- A simple model can get us very far before we need to refine

**Essentially, all models are wrong, but some are useful.**

Box, G. E. P., and Draper, N. R., (1987), *Empirical Model Building and Response Surfaces*, John Wiley & Sons, New York, NY.