ETRO VUB-DEPARTMENT OF ELECTRONICS AND INFORMATICS

Vrije Universiteit Brussel

General introduction: GPUs and the realm of parallel architectures

GPU Computing Training

August 17-19th 2015

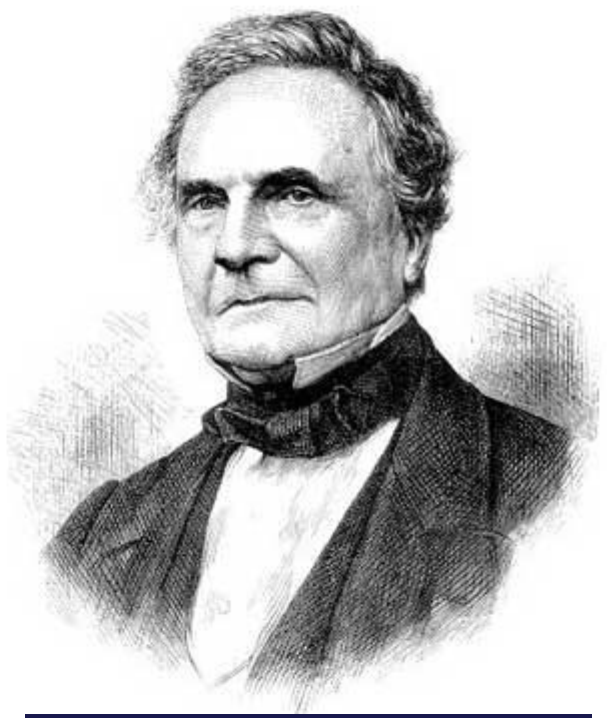
Jan Lemeire (jan.lemeire@vub.ac.be)

- Graduated as Engineer in **1994** at VUB
- Worked for **4 years** for 2 IT-consultancy companies
- **2000-2007**: PhD at the VUB while teaching as assistant
  - Subject: *probabilistic models for the performance analysis of parallel programs*
- **Since 2008**: postdoc en parttime professor at VUB, department of electronics and informatics (ETRO)
  - Teaching ‘Informatics’ for first-year bachelors; ‘parallel systems’ and ‘advanced computer architecture’ to masters
- **Since 2012**: also teaching for engineers industrial sciences (‘industrial engineers’)
- Projects, papers, phd students in *parallel processing* (performance analysis, GPU computing) & *data mining/machine learning* (probabilistic models, causality, learning algorithms)

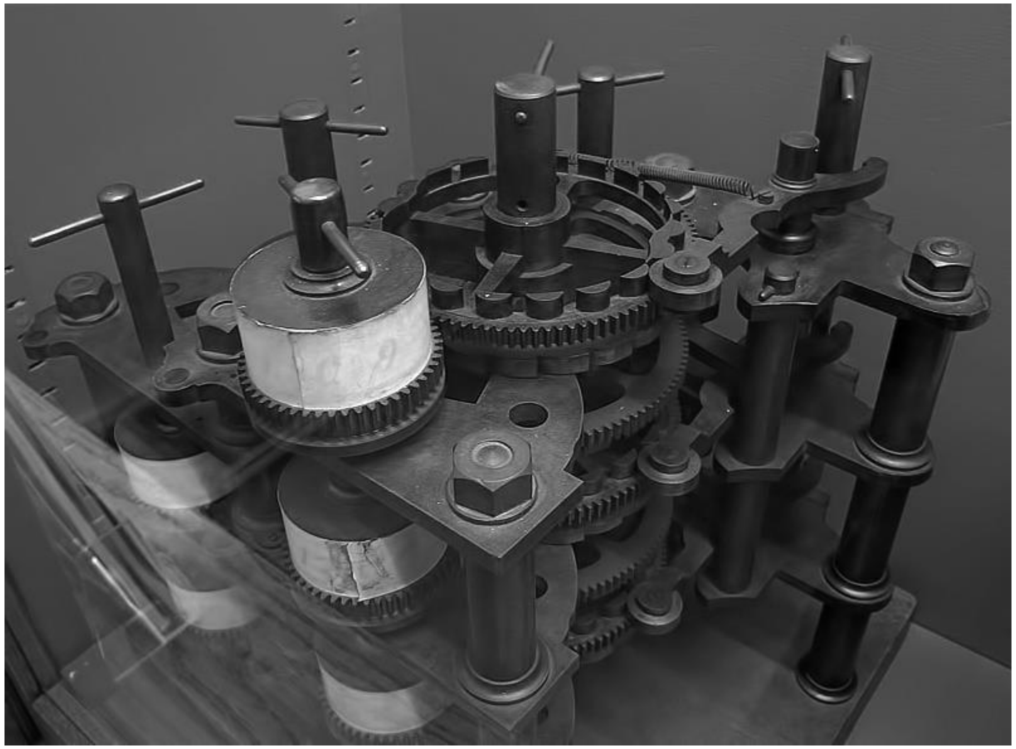
- http://parallel.vub.ac.be

A bit of History

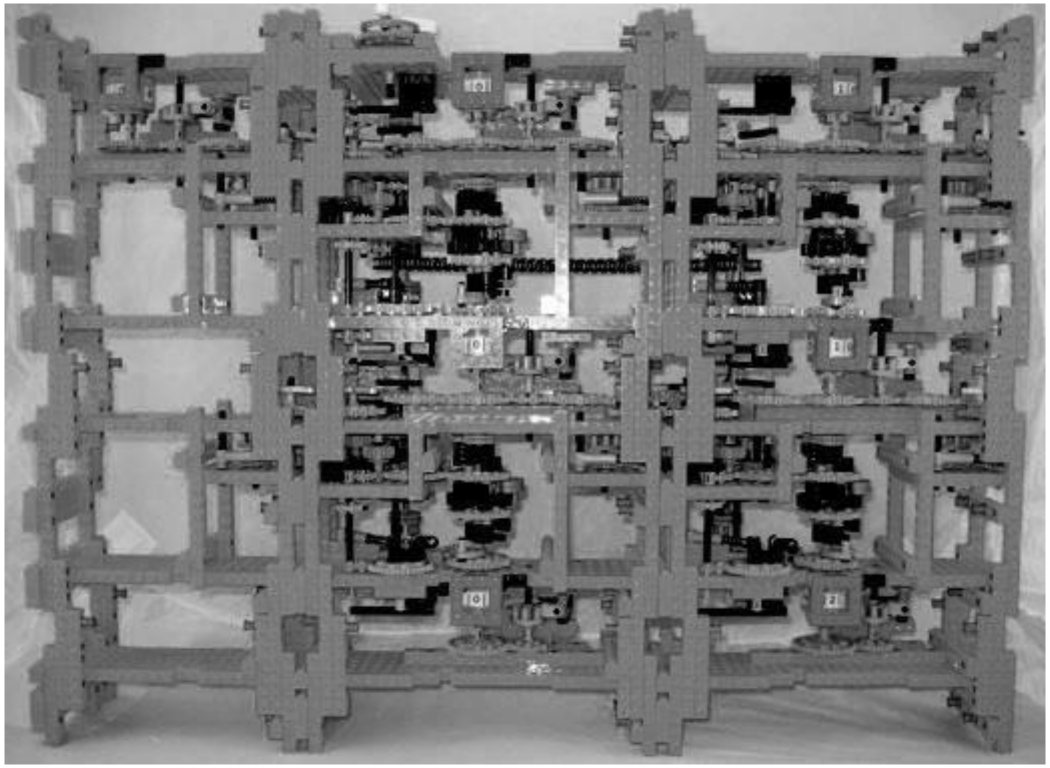
The first computer, mechanical



Charles Babbage
1791-1871



Babbage Difference Engine made with LEGO



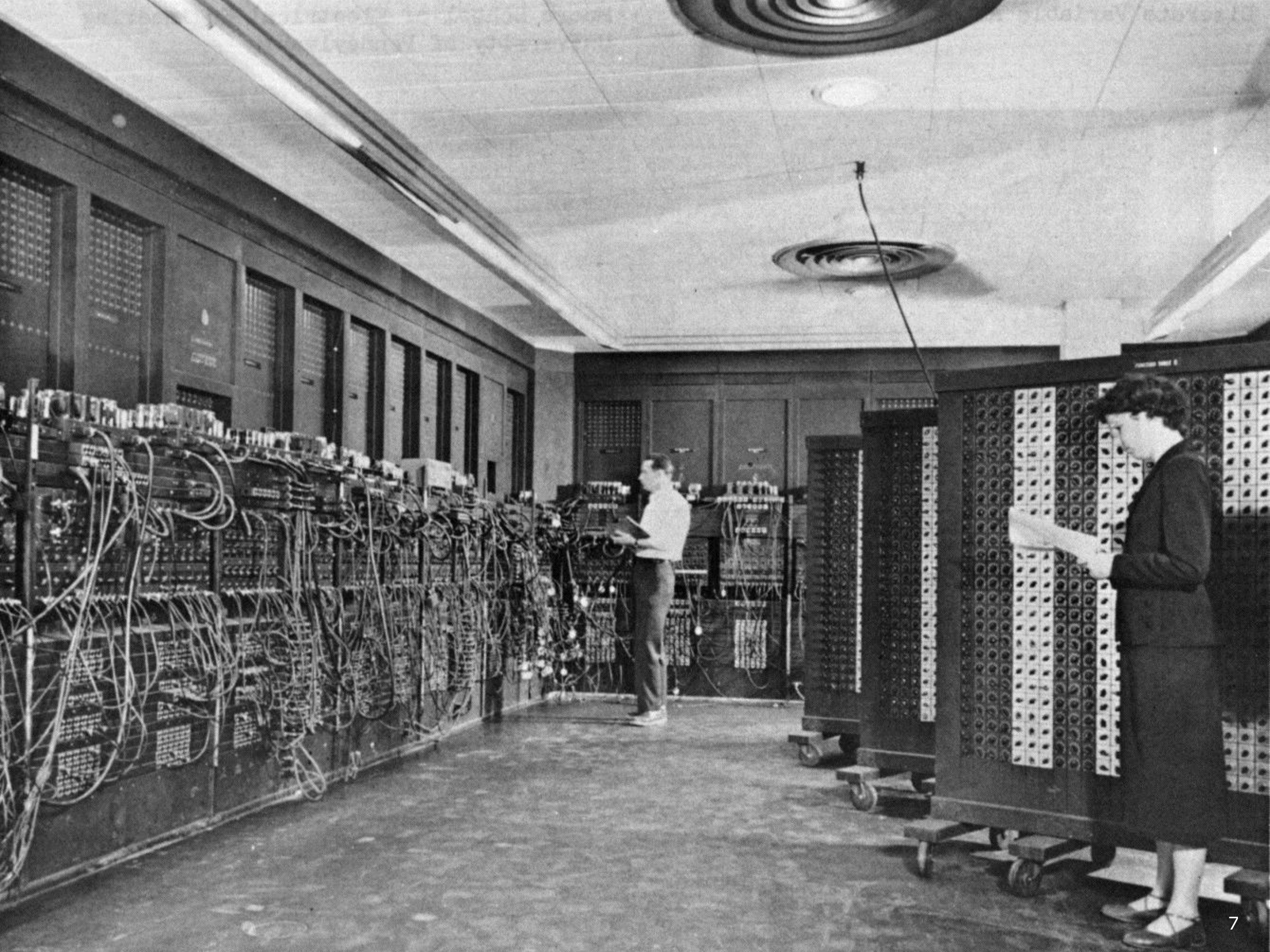
- http://acarol.woz.org/

This machine can evaluate polynomials of the form $Ax^2 + Bx + C$ for x=0, 1, 2, …n with 3 digit results.

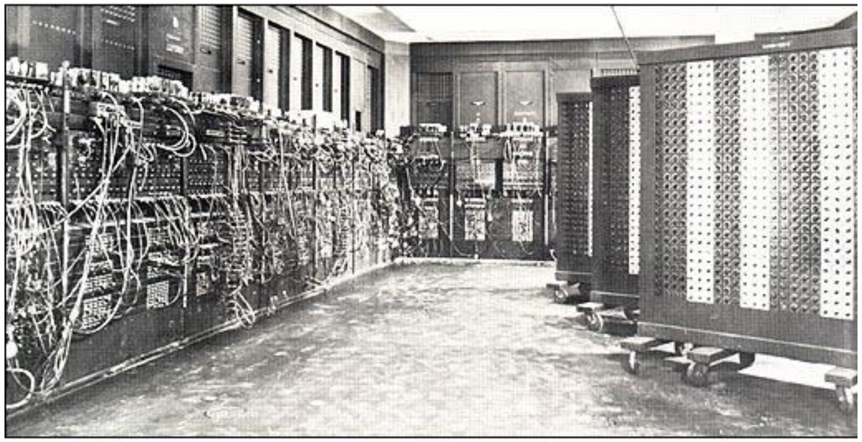
The first informatician

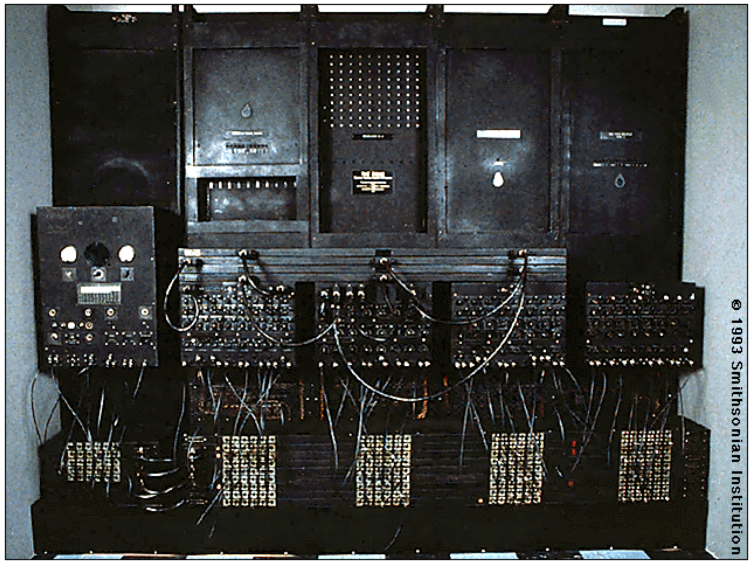
Ada Lovelace
1815 – 1852

- Describes what software is
  - She brings the insight that a computer goes beyond plain calculations
- She writes the first algorithm/program

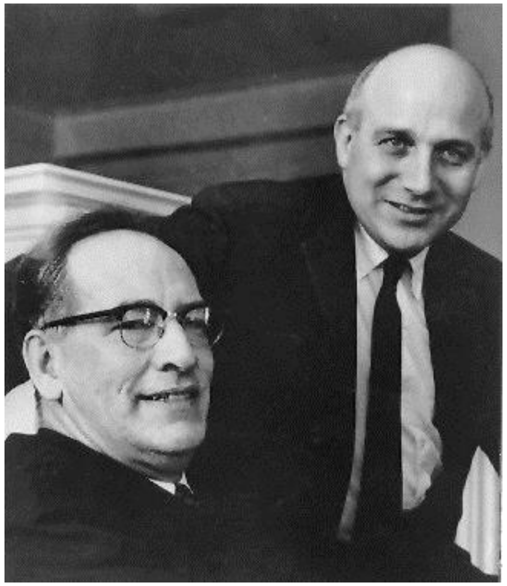


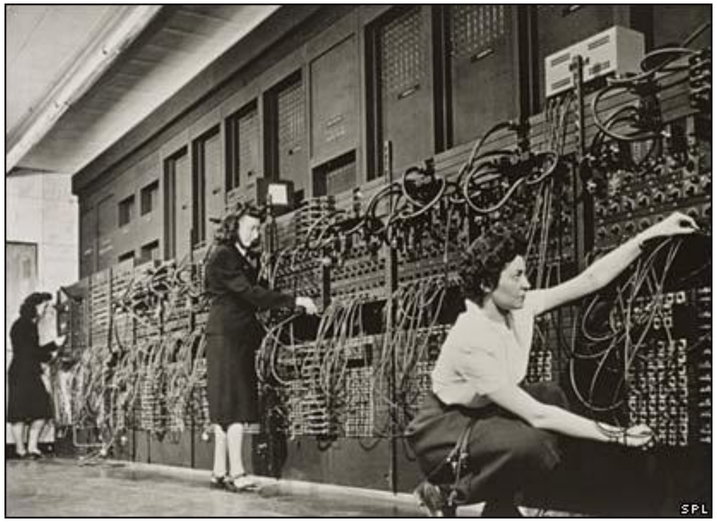
ENIAC



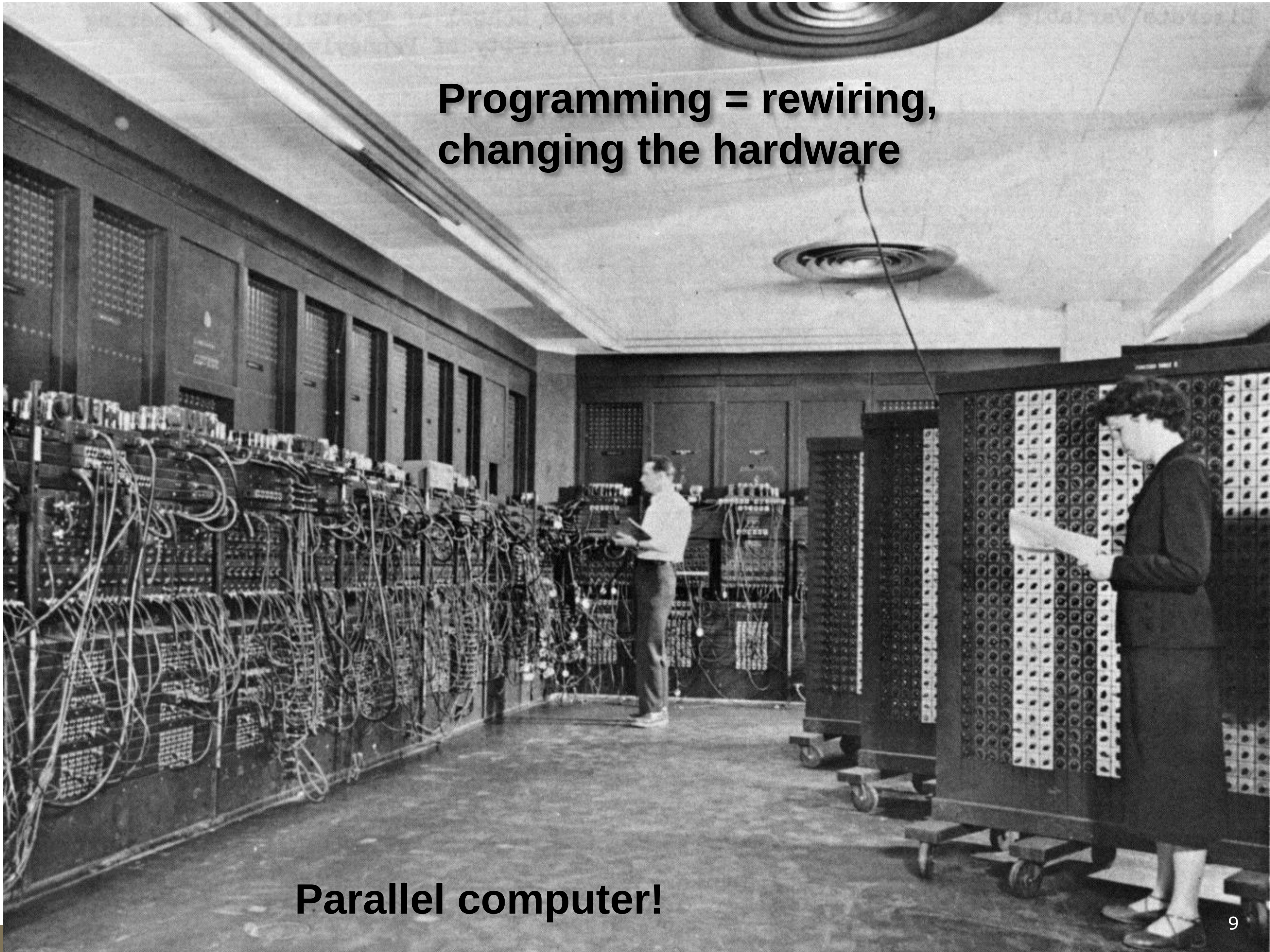


First computer: WWII





John Mauchly and John Eckert, 1945

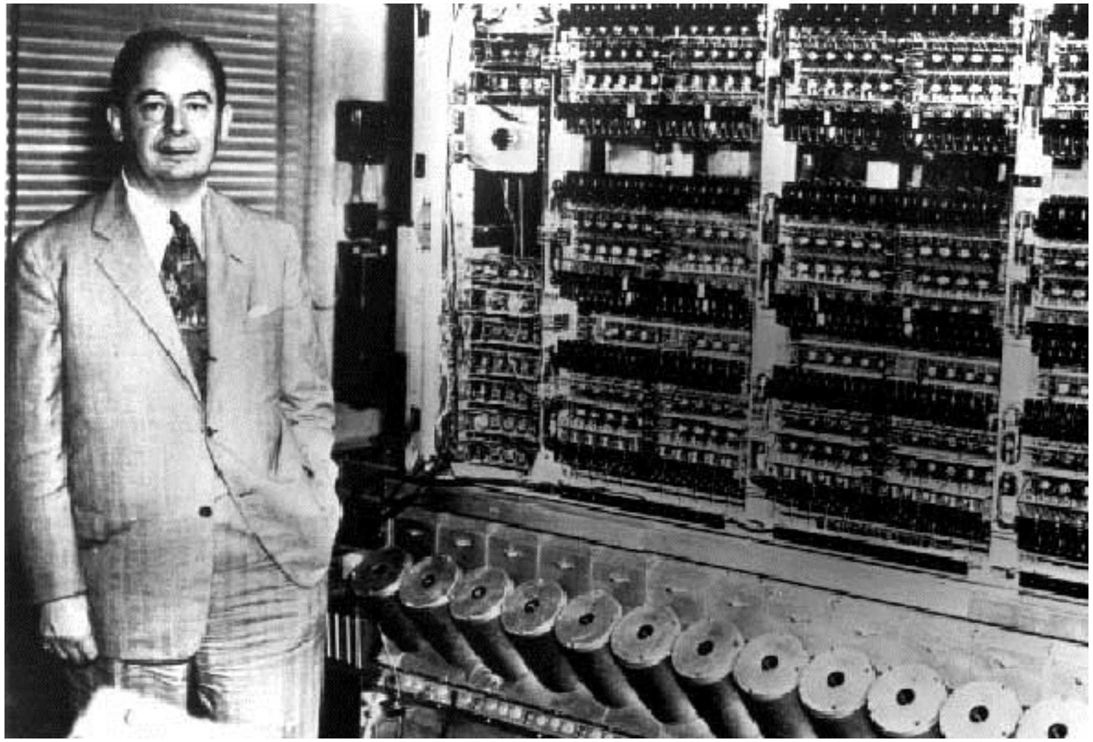


Programming = rewiring, changing the hardware

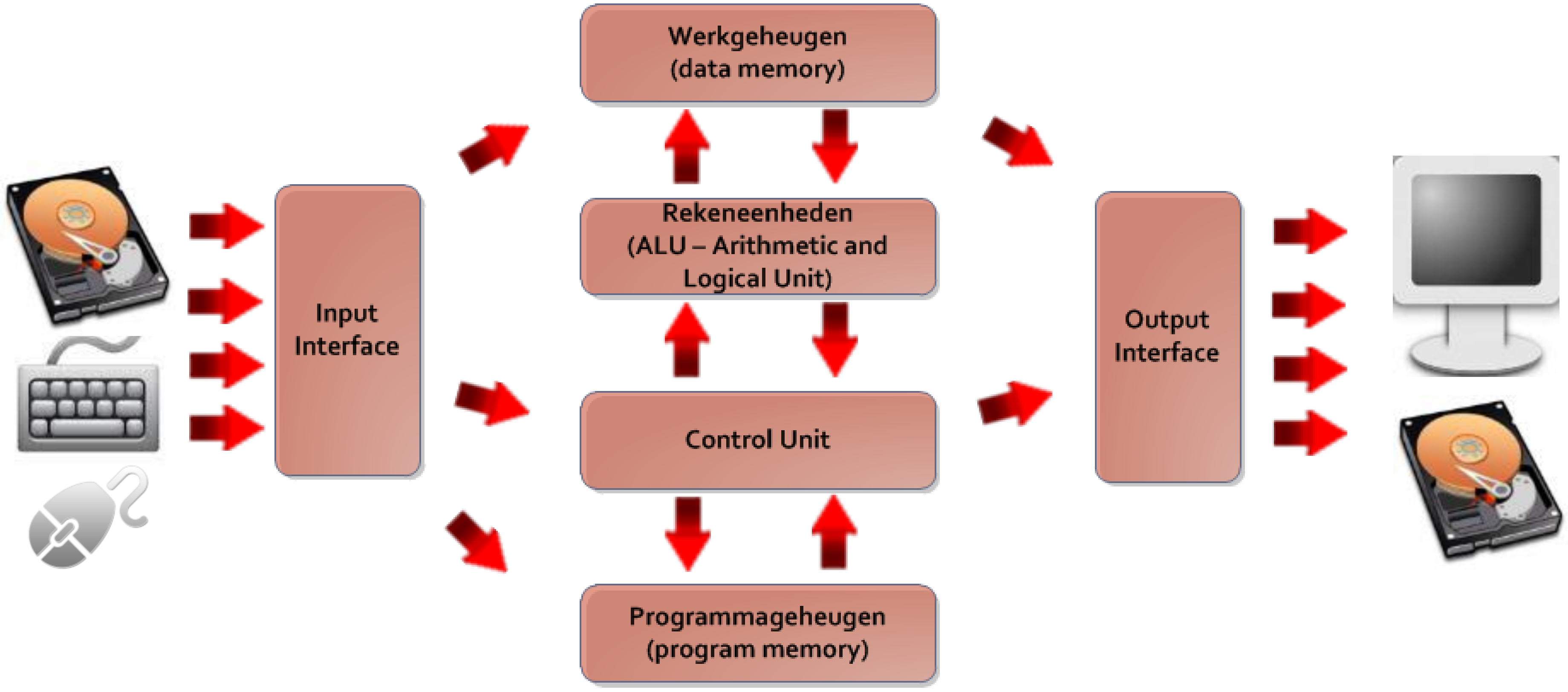
Parallel computer!

Von Neumann rethinks the computer

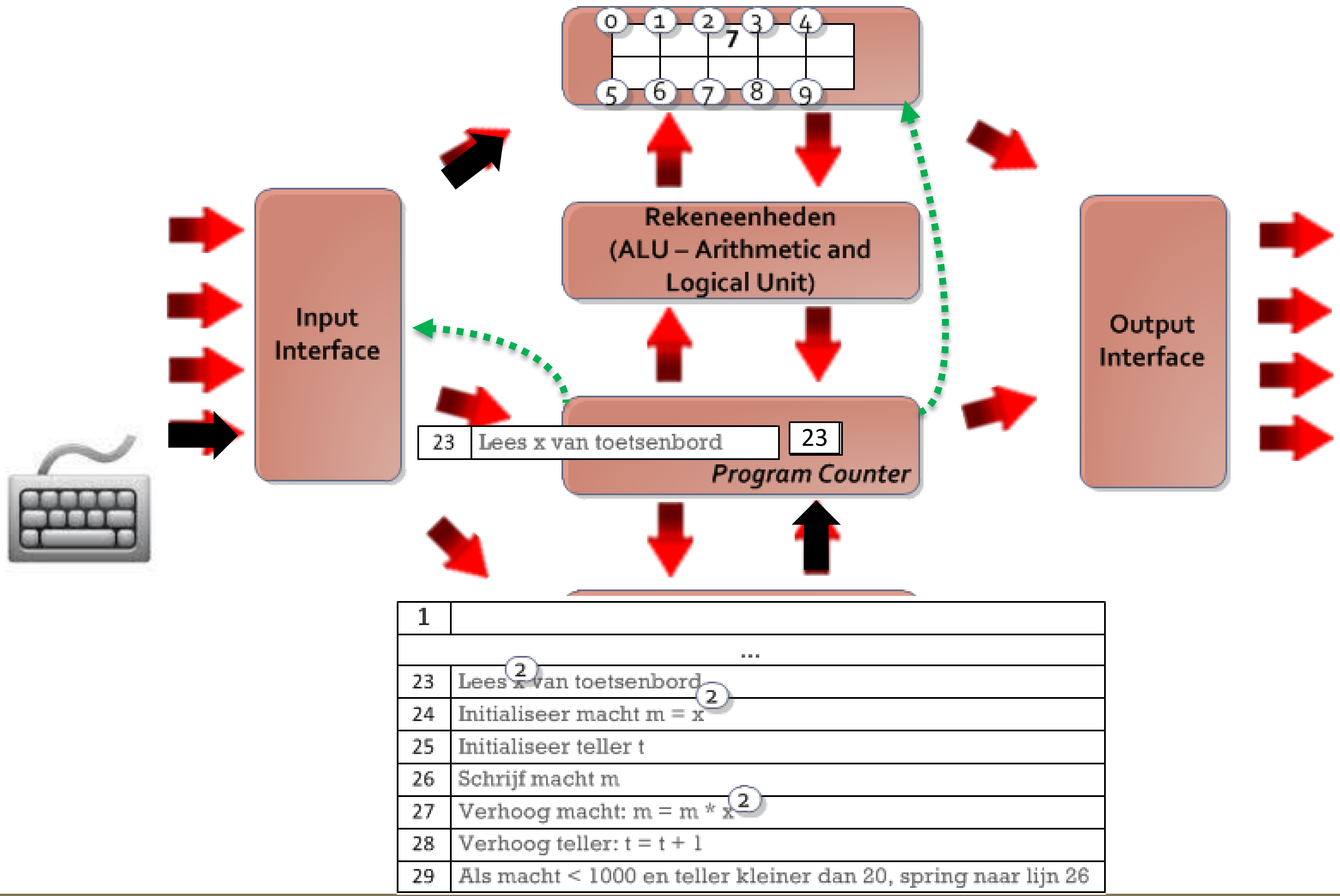
John Von Neumann



The Von Neumann-architecture



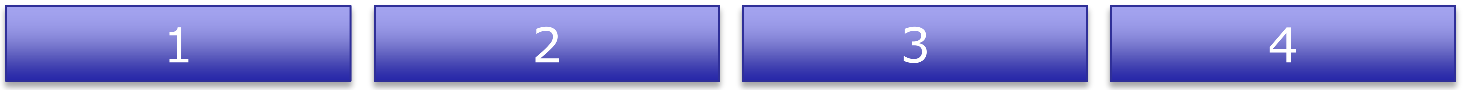
Execute program step-by-step



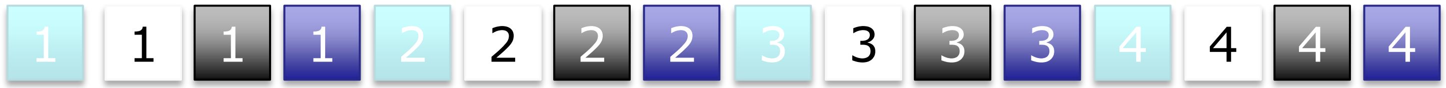
1	
...	
23	Lees x van toetsenbord
24	Initialiseer macht m = x
25	Initialiseer teller t
26	Schrijf macht m
27	Verhoog macht: m = m * x
28	Verhoog teller: t = t + 1
29	Als macht < 1000 en teller kleiner dan 20, spring naar lijn 26

For acceleration: **pipeline**

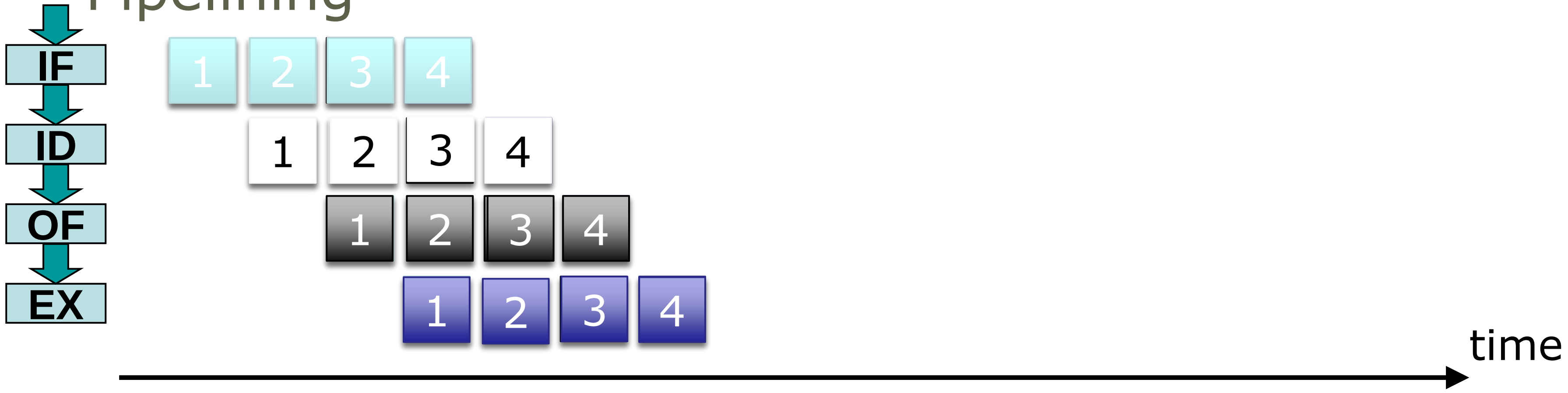
- Long operations



- Combination of short operations

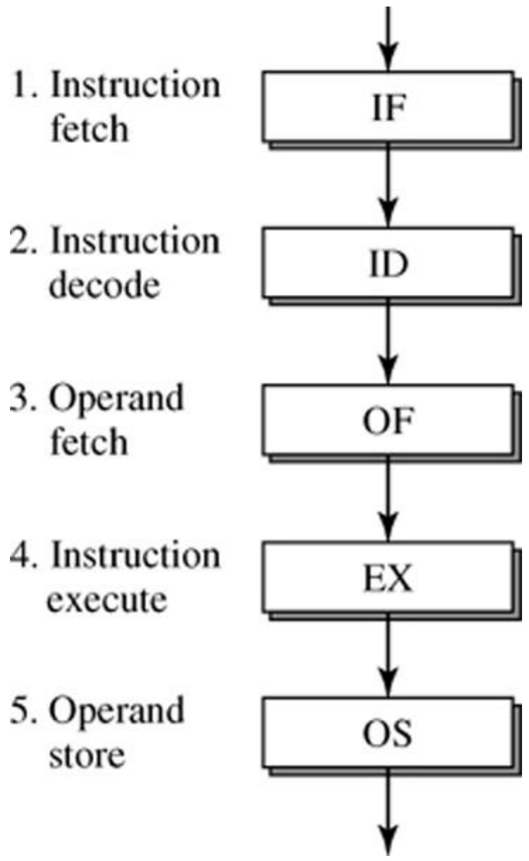


- Pipelining

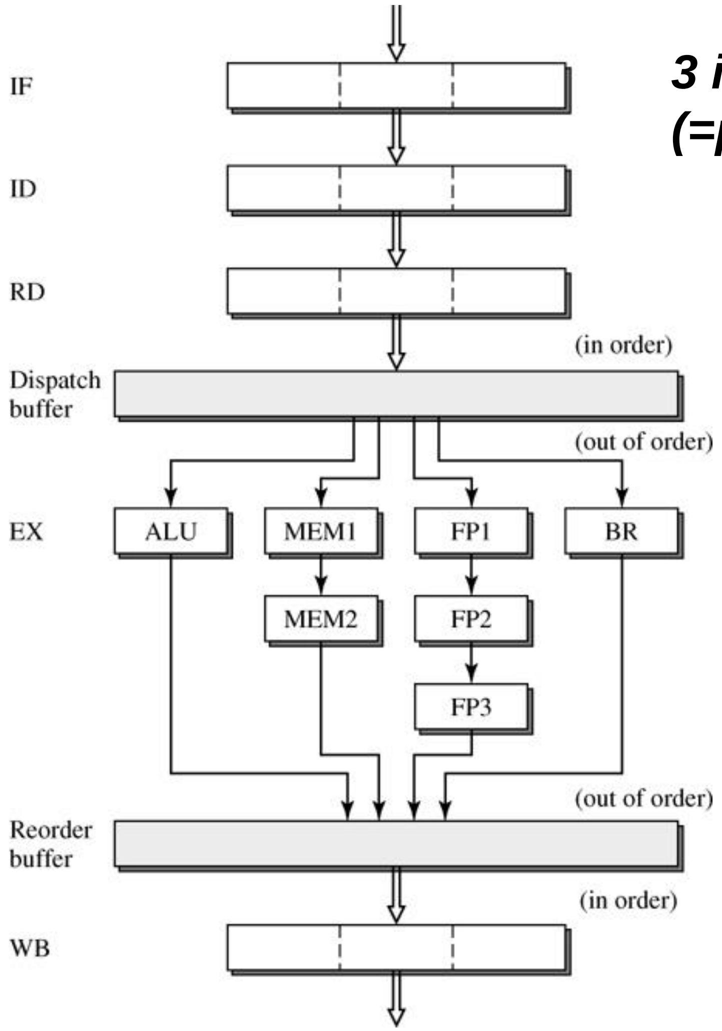


Pipeline Design

- Typically five tasks in instruction execution
  - IF: instruction fetch
  - ID: instruction decode
  - OF: operand fetch
  - EX: instruction execution
  - OS: operand store,
    often called write-back WB



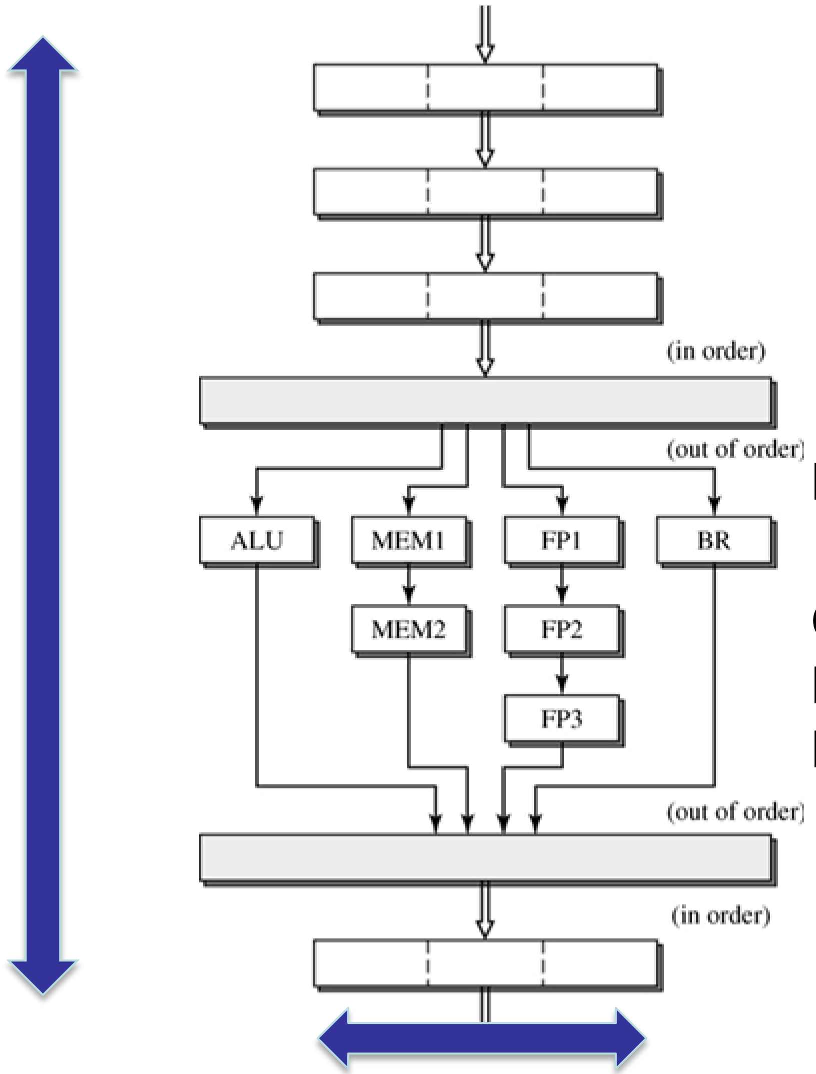
Superscalar out-of-order pipeline



3 instructions simultaneously (=pipeline width)

Independent instructions can overtake each other

‘Sequential’ processor: super-scalar out-of-order pipeline



Pipeline depth

Pipeline width

Different processing units

Out-of-order execution
Branch prediction
Register renaming
…

Now we are computing sequentially!

Parallel computing?

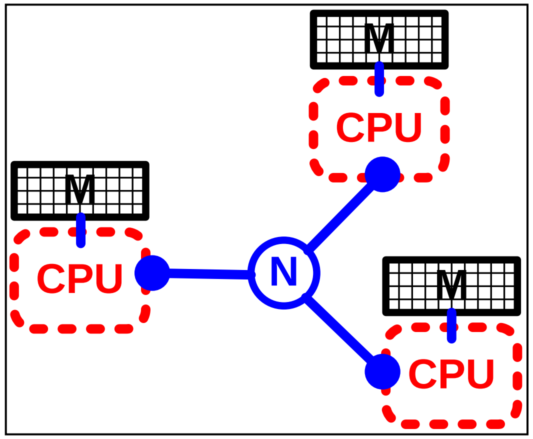
Super computer: BlueGene/L

- IBM 2007
- 65.536 dual core nodes
  - E.g. one processor dedicated to communication, other to computation
- Each 512 MB RAM
- No 8 in Top 500 Supercomputer list (2010)
  - www.top500.org

Clusters

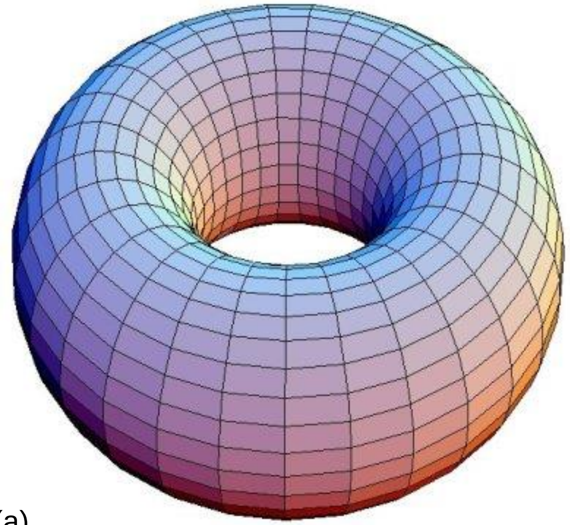
- Made from commodity parts
  - or blade servers
- Open-source software available

Distributed-Memory Architectures

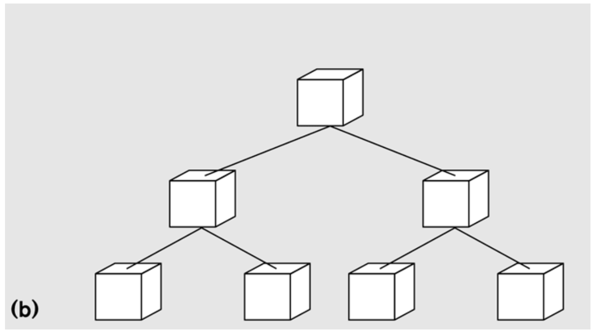


- Each process got his own local memory
- Communication through messages
- Process is in control

BlueGene/L communication networks



(a)



(a) 3D torus (64x32x32) for standard interprocessor data transfer
  - Cut-through routing (see later)

(b) collective network for fast evaluation of *reductions*.

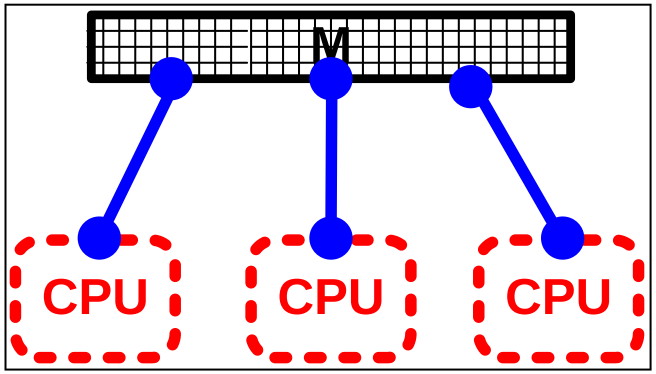
(c) Barrier network by a common wire

Message-passing

- The ability to send and receive messages is all we need
  - void send(message, destination)
  - message receive(source)
  - boolean probe(source)

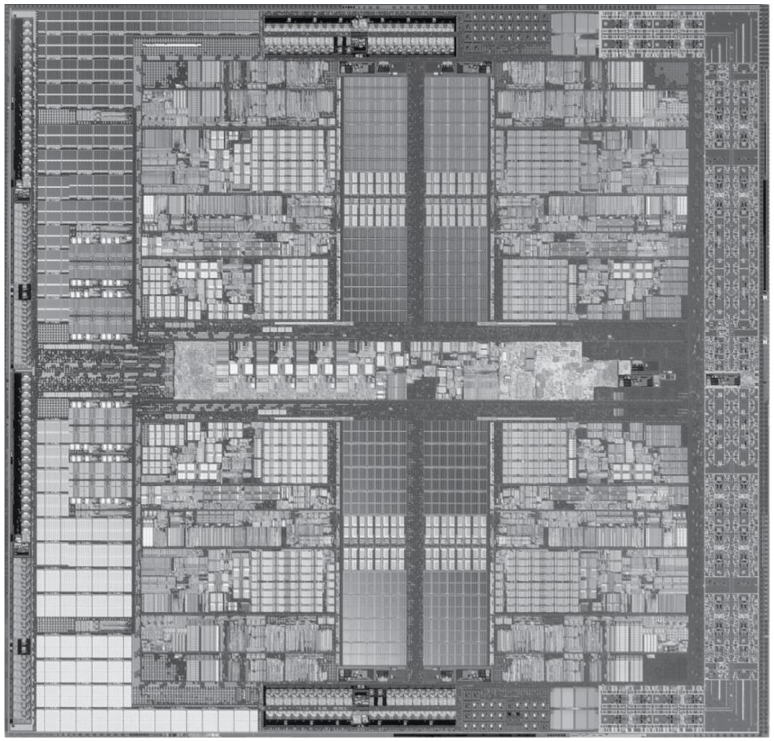
Multicore computing

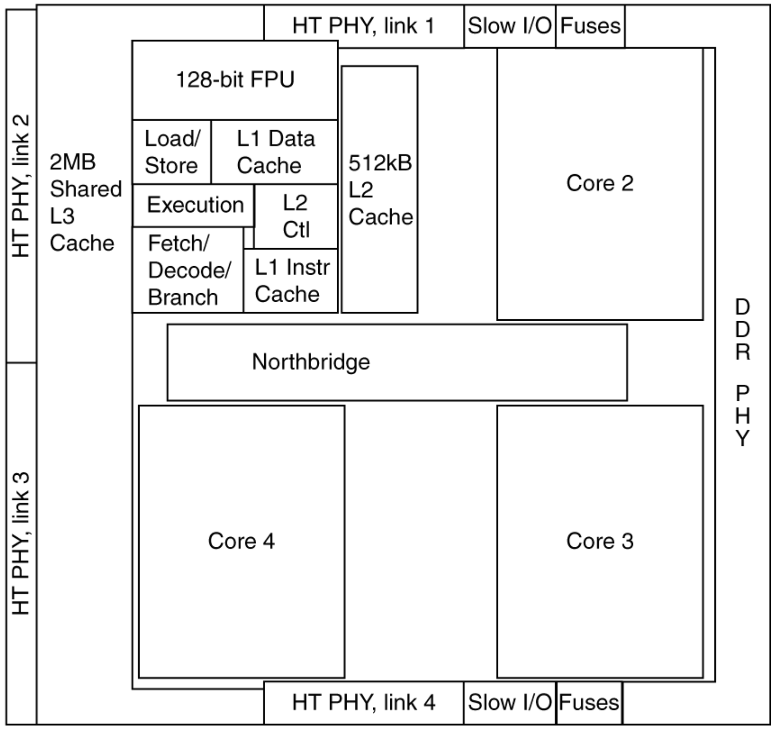
Shared Address-space Architectures



- Example: multiprocessors

AMD Barcelona: 4 processor cores





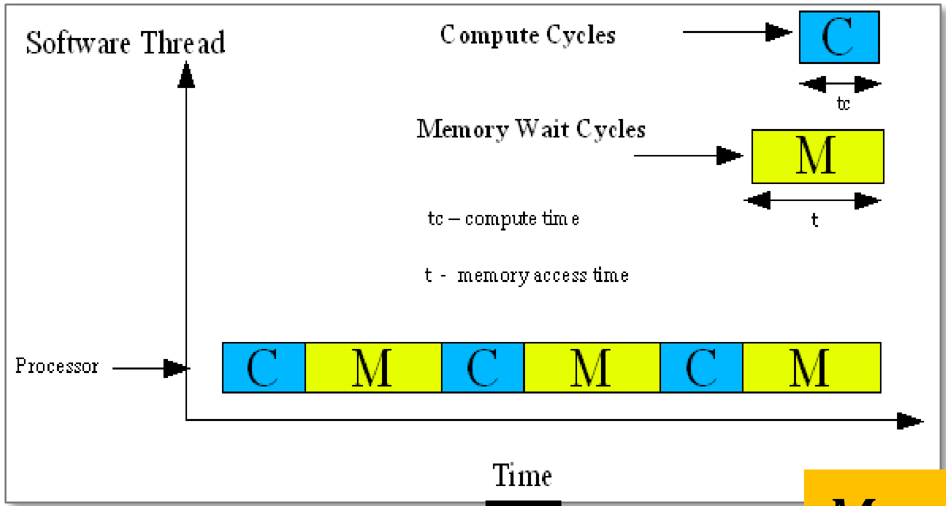
Thread / core

- A different thread per core
- Each thread can run independently

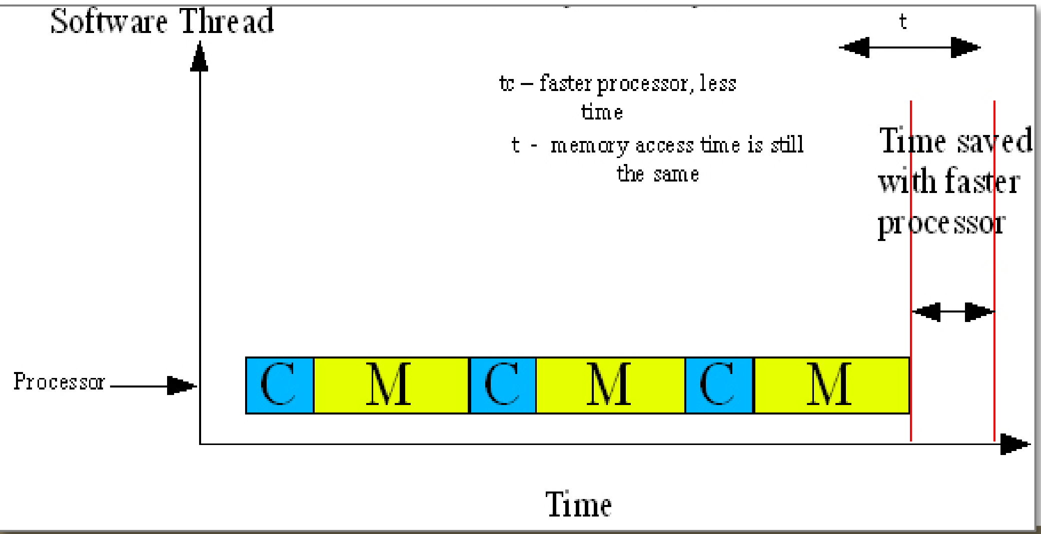
➡ Multi-threaded programming
- Thread synchronization necessary

- Multiple threads per core also possible
  - Context switches necessary
  - Hardware threads/hyperthreading

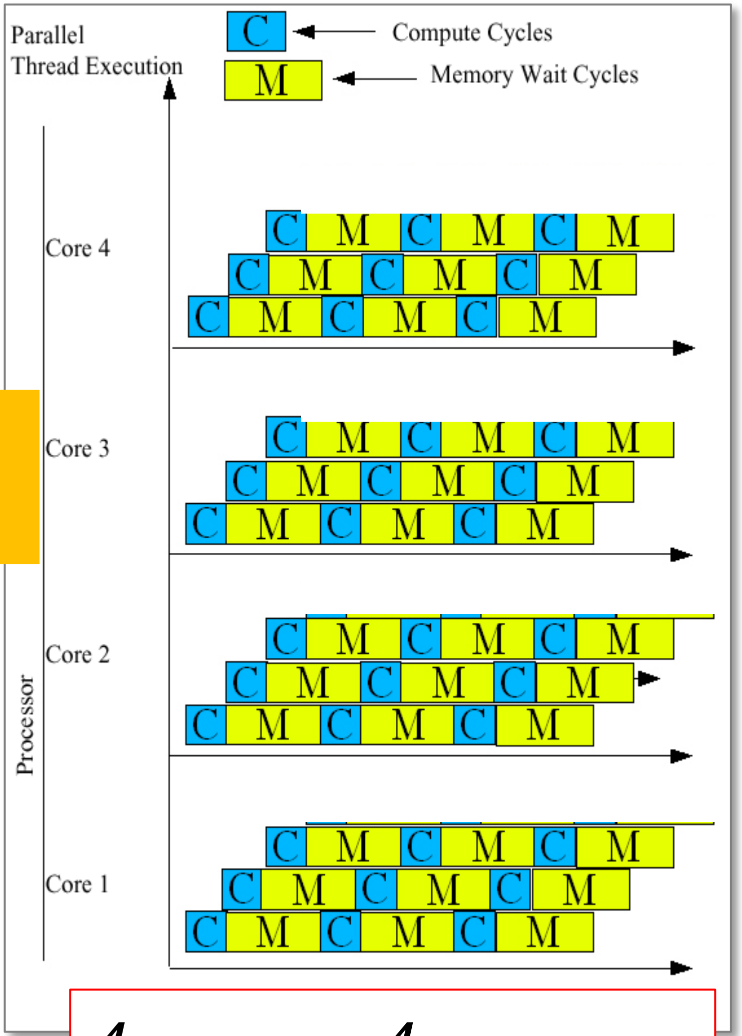
Latency Hiding



Faster CPU


Software Thread
t
tc – faster processor, less time
t - memory access time is still the same
Time saved with faster processor
Processor
C M C M C M
Time

More threads



4 cores: x4

Latency hiding: x3

Next level

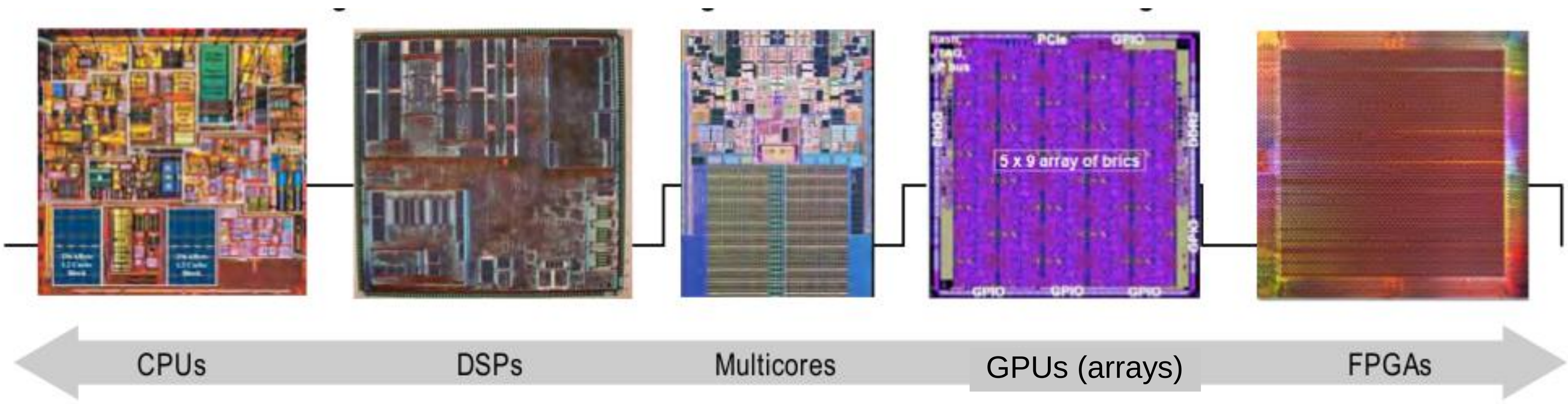
2010

350 Million triangles/second
3 Billion transistors GPU

1995

5,000 triangles/second
800,000 transistors GPU

Parallel processors



Single Cores

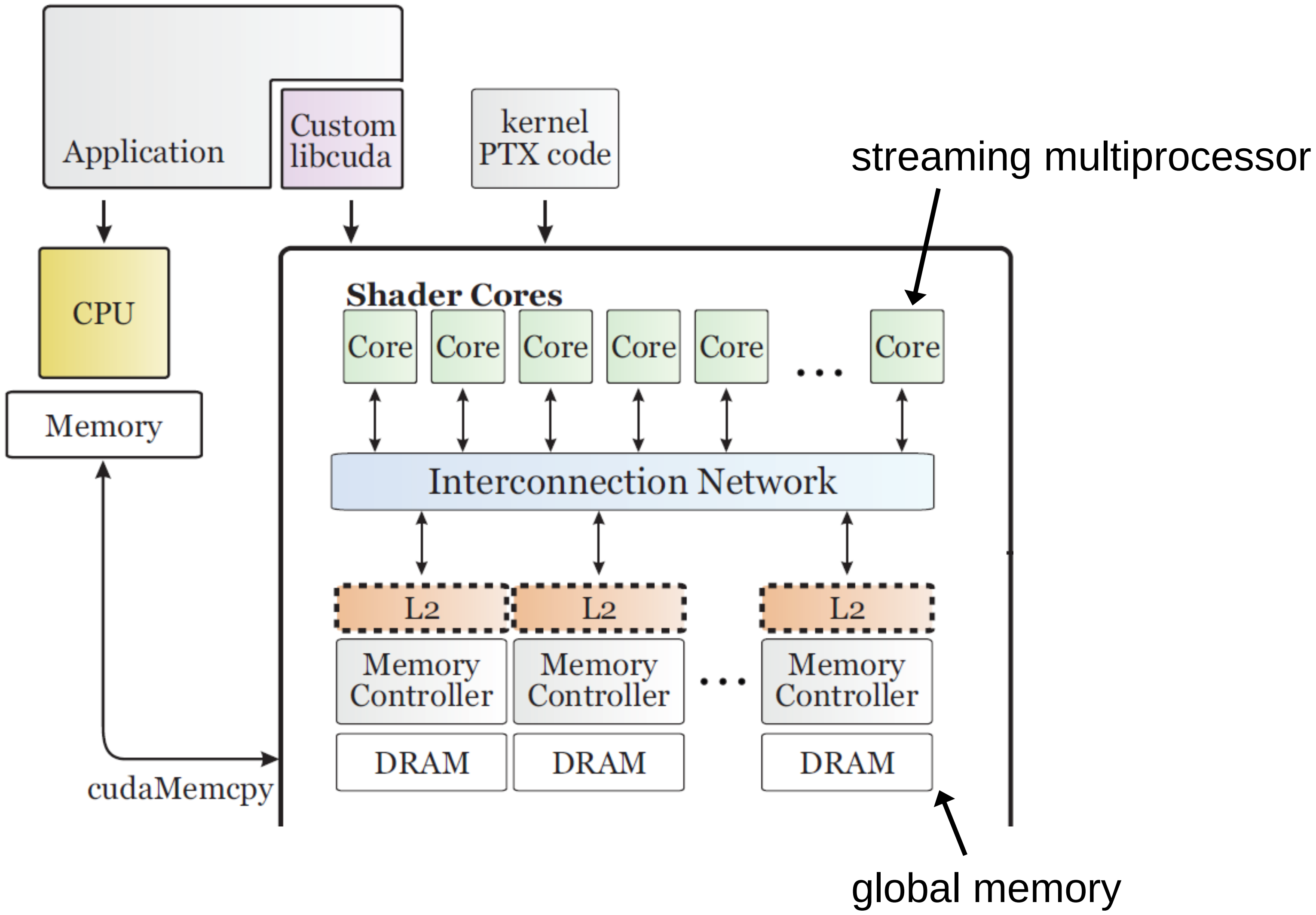
Multicores Coarse-Grained CPUs and DSPs

Coarse-Grained Massively Parallel Processor Arrays

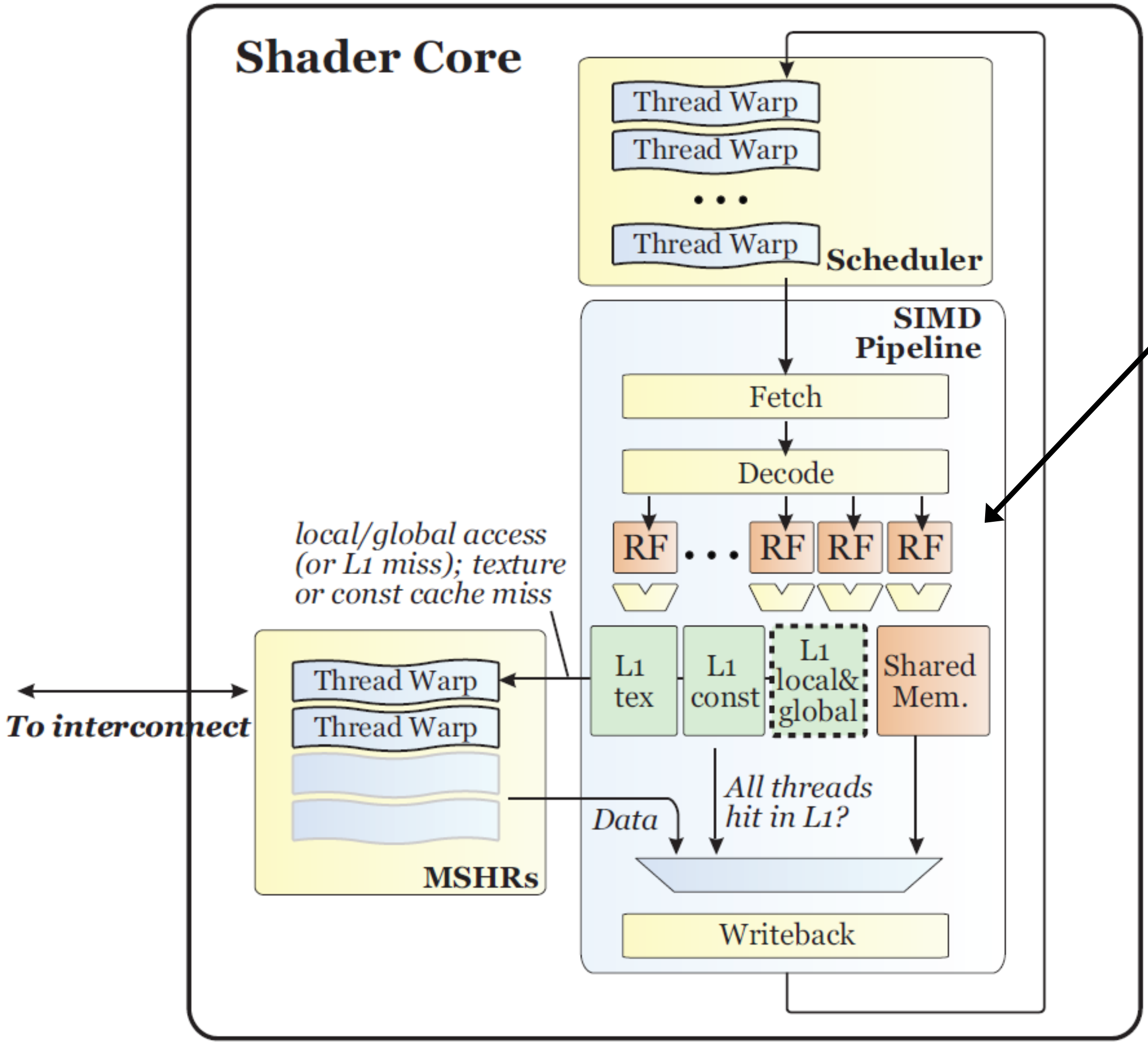
Fine-Grained Massively Parallel Arrays

Courtesy of ALTERA

GPU Architecture



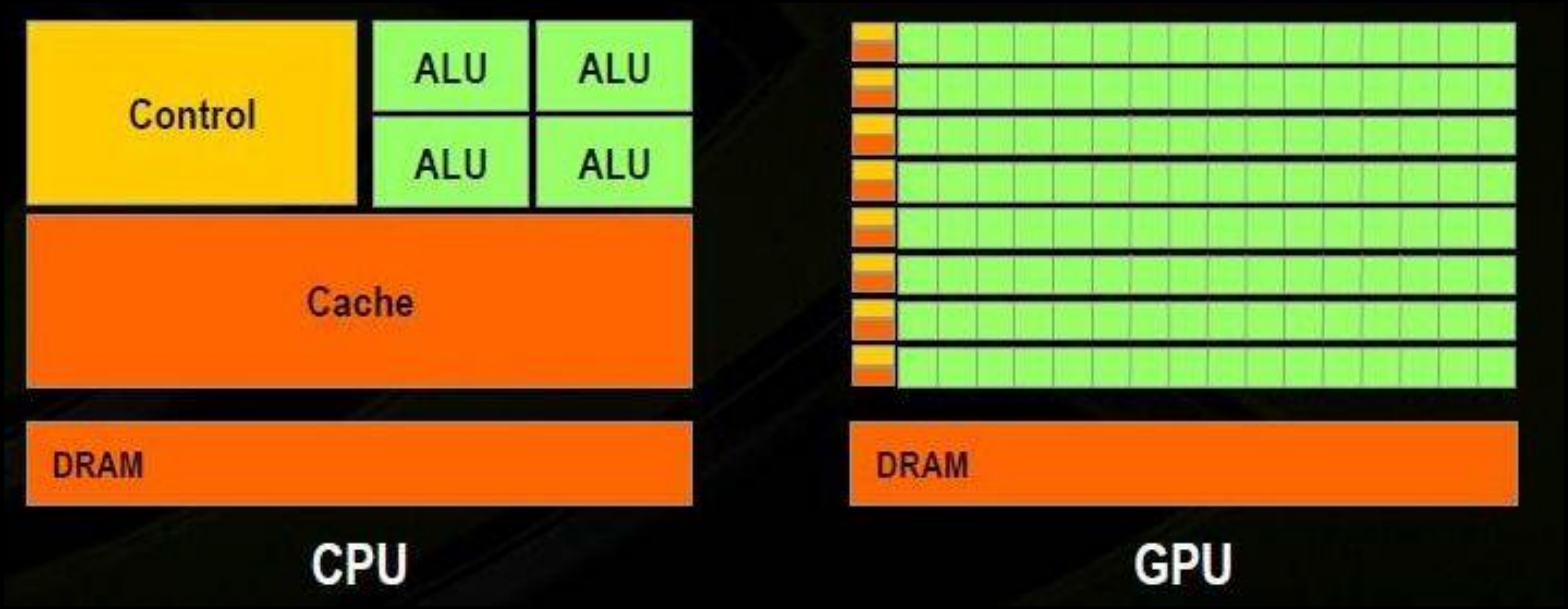
1 Streaming Multiprocessor



The Same Instruction is executed on Multiple Data (SIMD)

width of pipeline: 8 - 32

Why are GPUs faster?

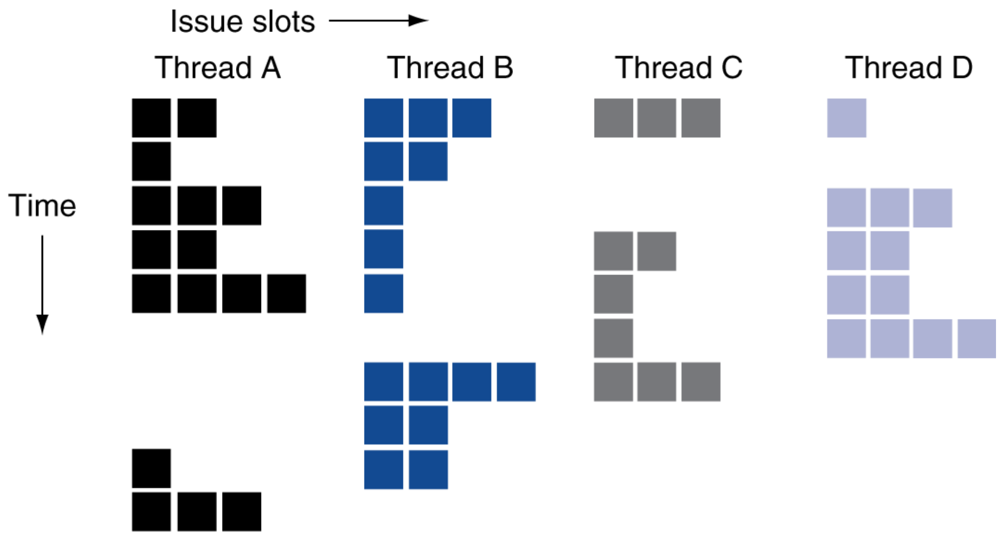


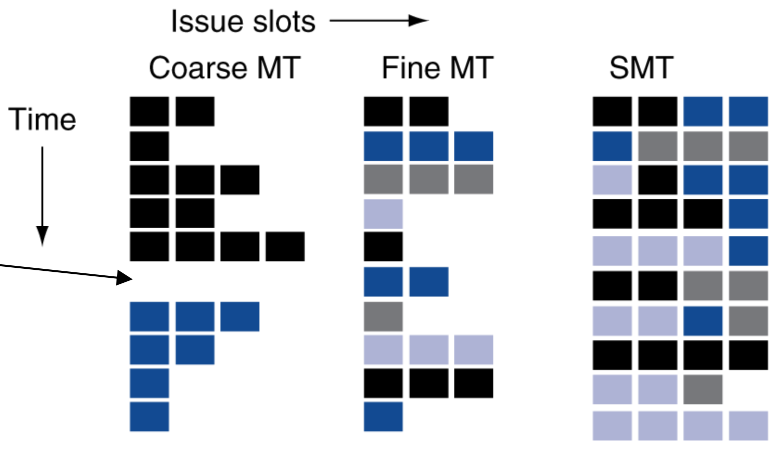
Devote transistors to… computation

GPU processor pipeline

- ±24 stages (old), now ± 8
- in-order execution!!
- no branch prediction!!
- no forwarding!!
- no register renaming!!
- Memory system:
  - relatively small
  - Until recently no caching
  - On the other hand: much more registers (see later)

Multi-Threading (MT) possibilities





Context switch

Processing power not for free

Obstacle 1

Hard(er) to implement

Obstacle 2

Hard(er) to get efficiency

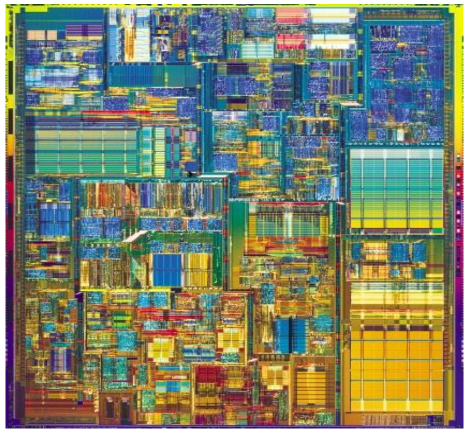
CPU computing

Write once
Run everywhere
efficiently!

manual

automatic

$$\begin{tikzcd}
\textbf{Algorithm} \arrow[d] \\
\textbf{Implementation} \arrow[d] \\
\textbf{Compiler} \arrow[d] \\
{}
\end{tikzcd}$$



Automatic optimization

⇩

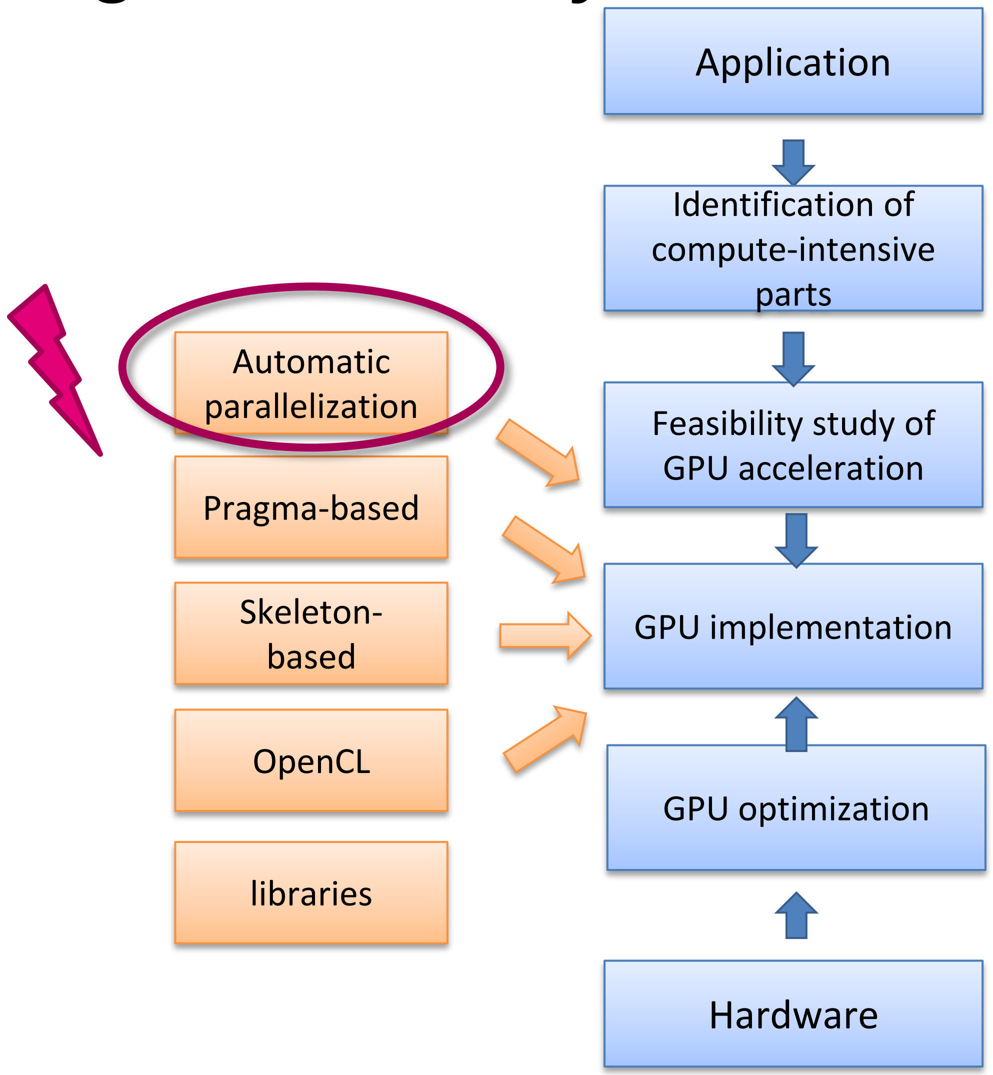
Low latency of each instruction!

Computer science

is not about

computers

Programmability solutions



Application
Identification of compute-intensive parts
Feasibility study of GPU acceleration
GPU implementation
GPU optimization
Hardware
Automatic parallelization
Pragma-based
Skeleton-based
OpenCL
libraries

Auto-parallelization

- Key requirements
  - A compiler must not alter the program semantics
  - If the compiler cannot determine all dependencies, it has to forego parallelization

- Compilers sometimes need to act very conservatively
  - Pointers make it hard for the compiler to deduce memory layout
  - Codes may produce overlapping arrays through pointer arithmetics
  - If the compiler can't tell, it does not parallelize

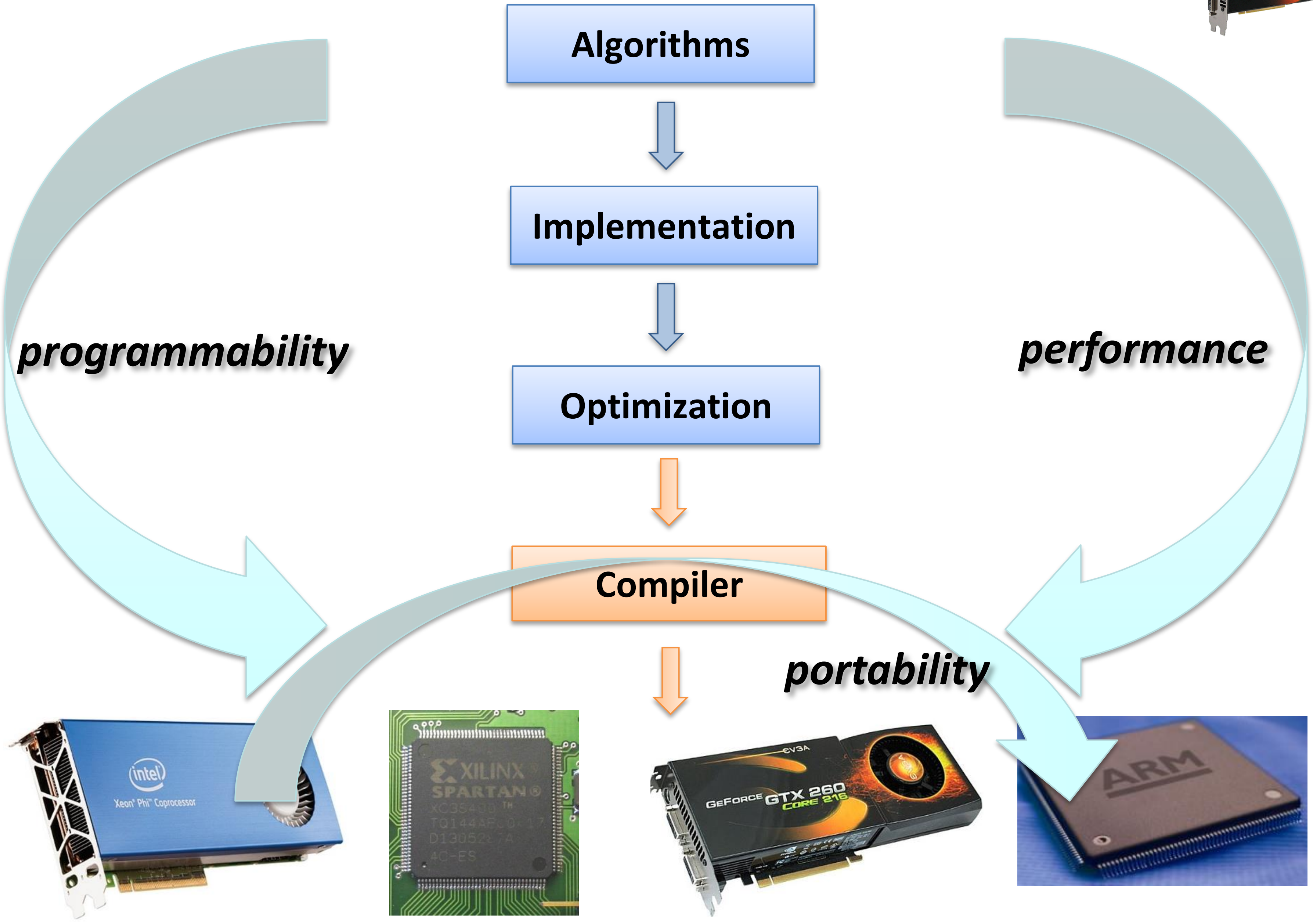
- Past 30 years have shown that auto-parallelization
  - is a tough problem in general
  - is only applicable to very regular loops
  - cannot take care of manual parallelization tasks

Computer science

is n~~ot~~ about

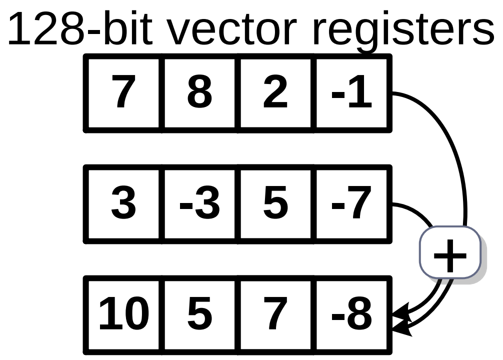
computers

Challenges of GPU computing



Intel Xeon Phi

Single Instruction Multiple Data (SIMD)



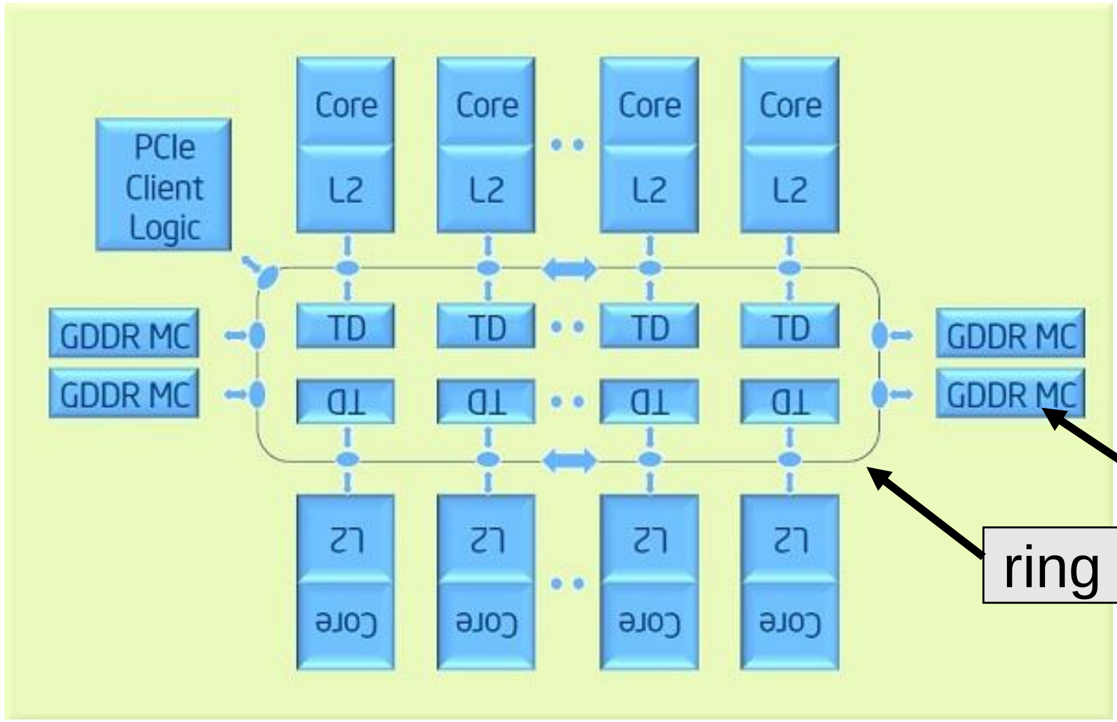
Instructions can be performed at once on all elements of vector registers

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
    - Data has to be moved explicitly to/from vector registers
- All processors execute the same instruction at the same time
  - Instruction has to be fetched only once

Vector processors (SIMD)

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- **Has long be viewed as the solution for high-performance computing**
  - Why always repeating the same instructions (on different data)? => just apply the instruction immediately on all data
- *However: difficult to program*
- Is SIMT (OpenCL) a better alternative??

Intel’s Xeon Phi coprocessor



RAM

ring network

Intel’s response to GPUs…

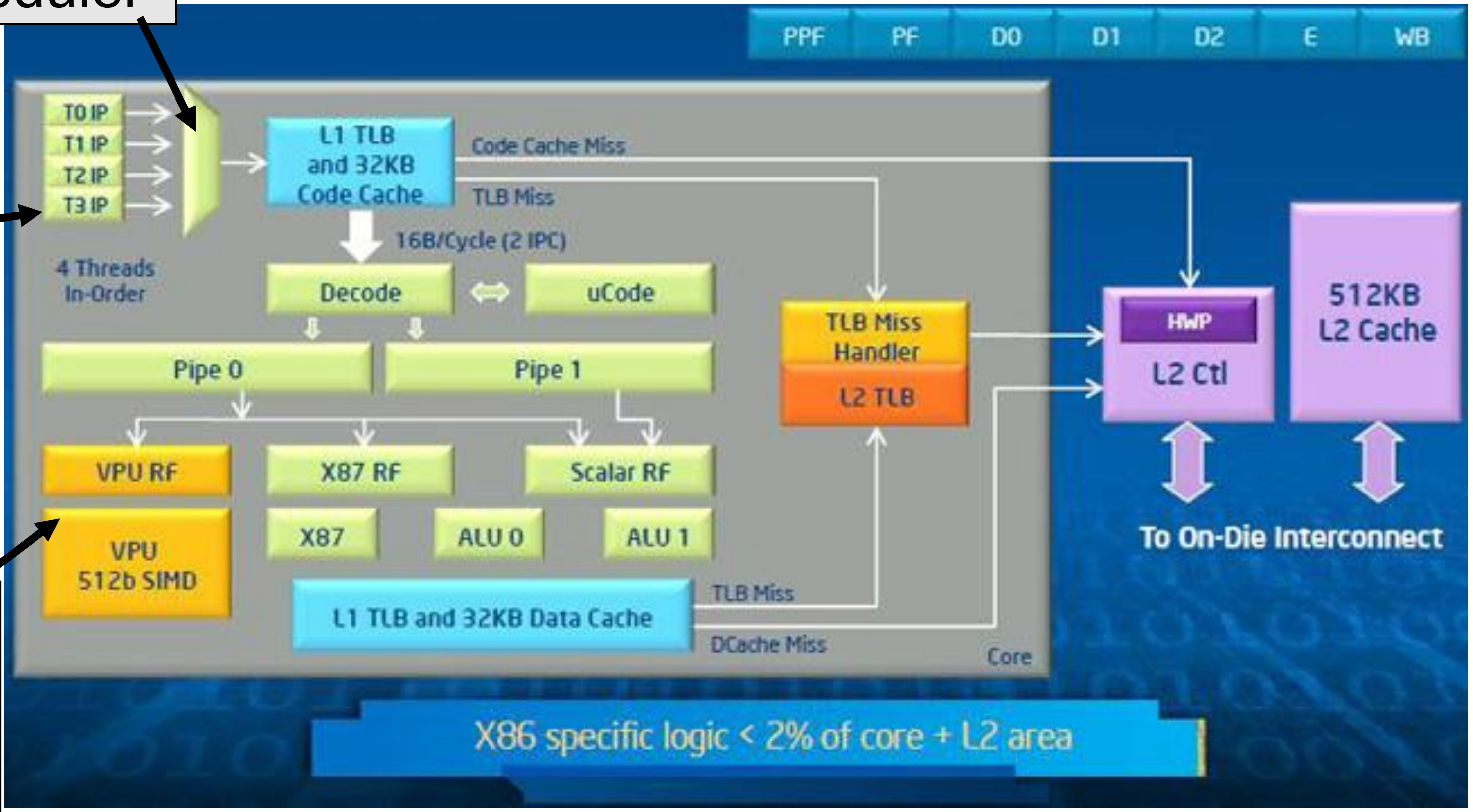
60 cores

Intel's Xeon Phi's core

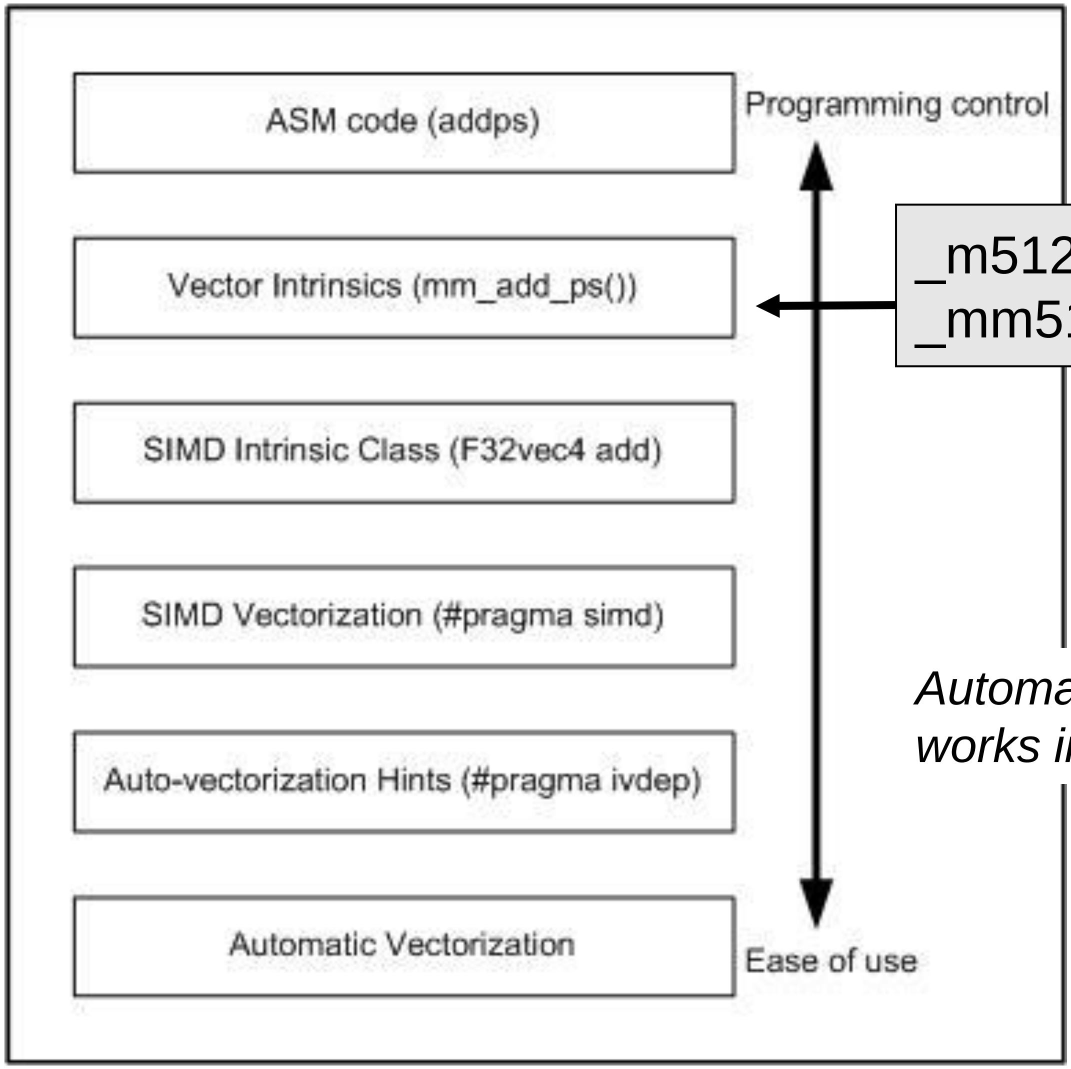
Thread scheduler

4 hardware threads

512-bit Vector unit (SIMD)



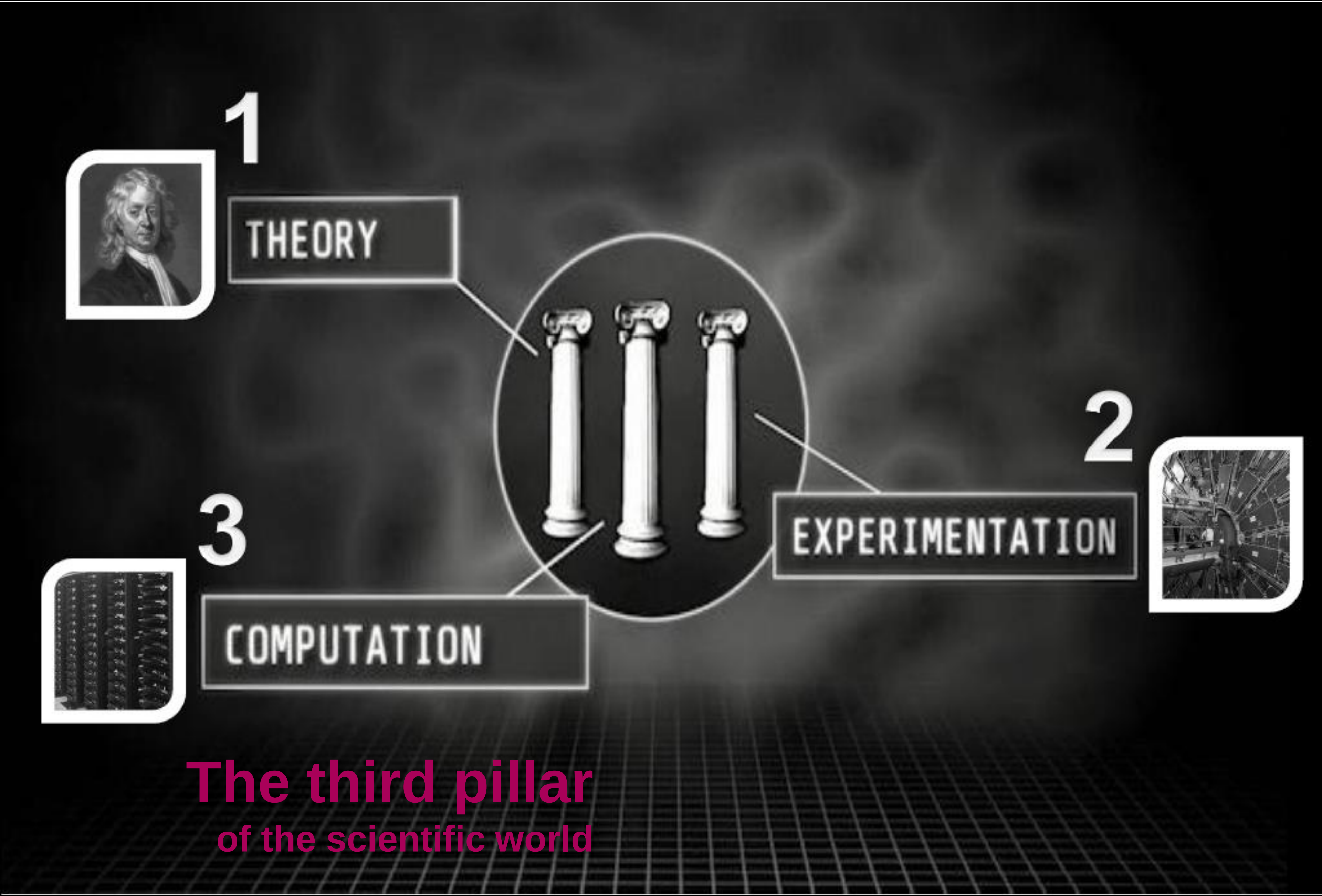
Vectorization needed for peak performance!!


ASM code (addps)
Vector Intrinsics (mm_add_ps())
SIMD Intrinsic Class (F32vec4 add)
SIMD Vectorization (#pragma simd)
Auto-vectorization Hints (#pragma ivdep)
Automatic Vectorization
Programming control
Ease of use

_m512 register
_mm512_add operations

Automatic vectorization only works in simple cases

Conclusions

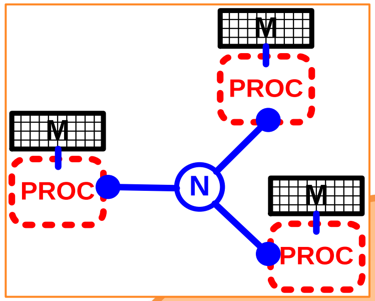


The third pillar

of the scientific world

Parallel Programming Paradigms

Distributed memory

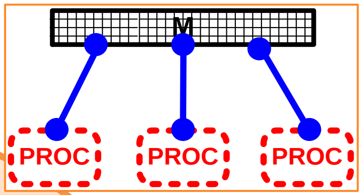


Message-passing MPI

coarse-grain parallelism

Shared memory

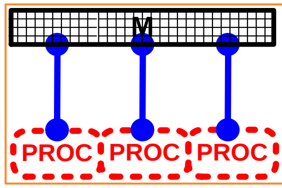
coarse-grain parallelism



Explicit multi-threading

OpenMP

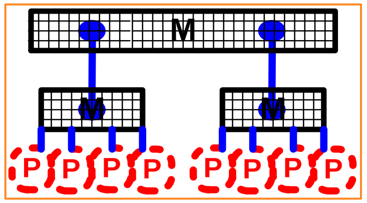
Explicit vector instructions



SIMD

OpenCL/CUDA

fine-grain parallelism



SIMT

Hardware? Software?

- You need to have insight into the hardware!
- No universal hardware/programming model (yet)
- **Intel-approach (SIMD)**
  - Intel sticks to x86 architecture
    - That’s what programmers know & they won’t change
    - Vectorization necessary
- **OpenCL-approach (SIMT)**
  - Will semi-abstract model remain valid?