An Introduction to computer engineering and the nature of programming languages

Tim Mattson

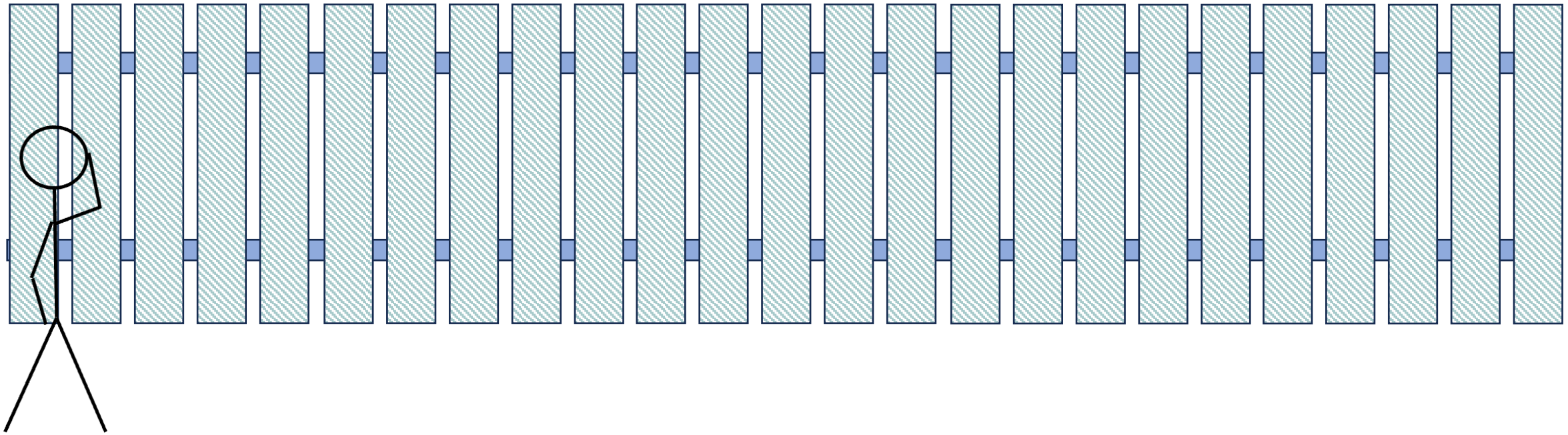
Tim demonstrates his new invention: Kayak snorkeling. Palawan Philippines, 2019

But first a brief digression …

What do we mean by the word *parallelism*

Painting a fence

- You must paint a fence. It has 25 planks.

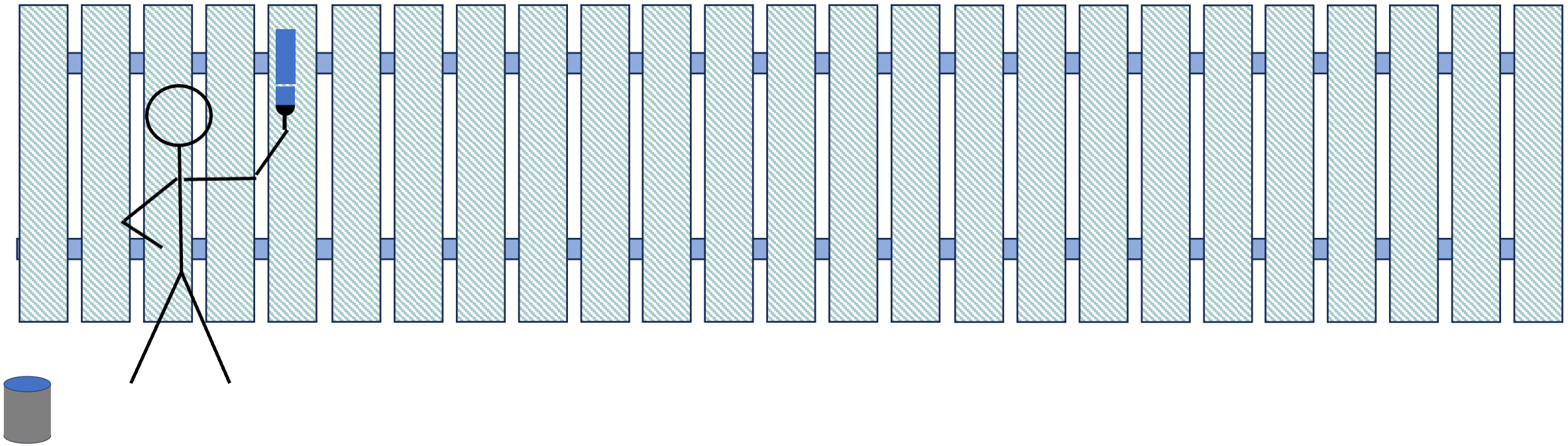


Painting a fence: Doing it as a serial process

- You must paint a fence. It has 25 planks. You are going to do this one plank at a time (**a serial process**)
- How long should it take?

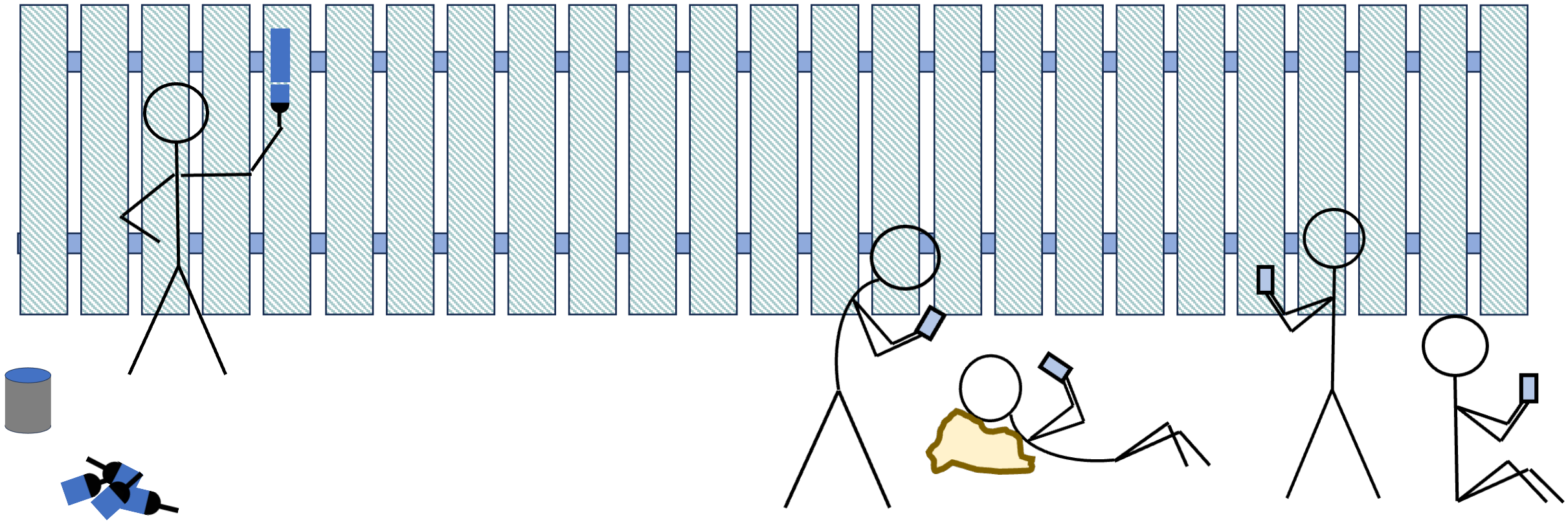
Ideal case

It takes T_p minutes to paint a plank. Therefore it takes $25*T_p$ minutes to paint the entire fence.



Painting a fence: Getting help to finish the job in less time

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take with the five of you working **in parallel**?

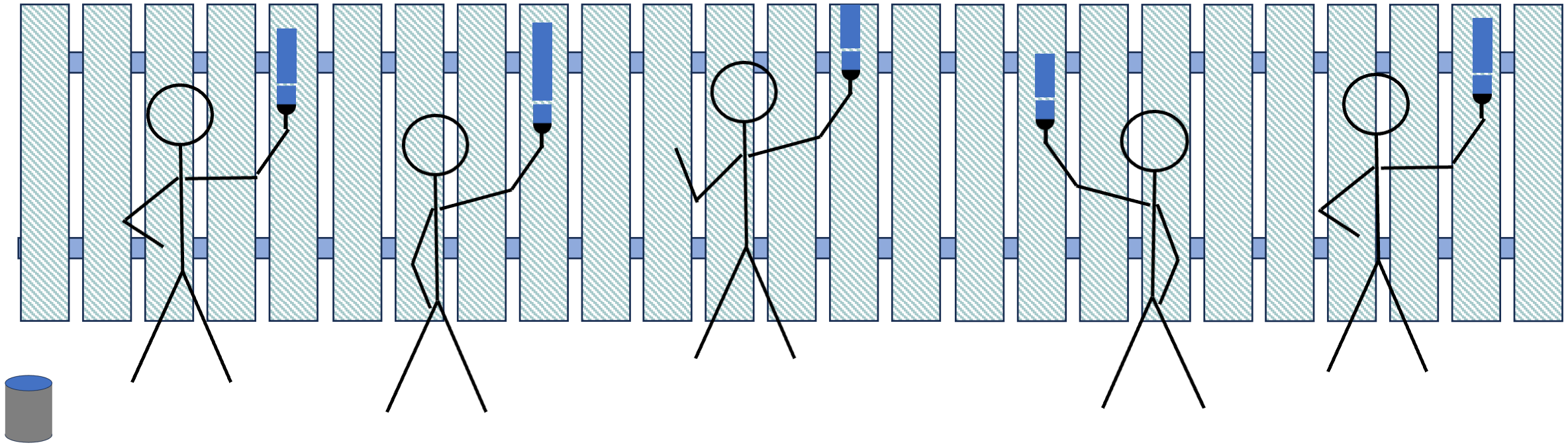


Painting a fence: Getting help to finish the job in less time

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take?

Ideal case

It takes $25*T_p$ minutes to paint the fence on your own ("**in serial**"). With N = 5 people each taking an equal chunk of the fence, then each person paints 25/N planks. If they do it all at the same time (that is, "**in parallel**"), the fence will be done in $(25*T_p)/N = 5*Tp$ minutes



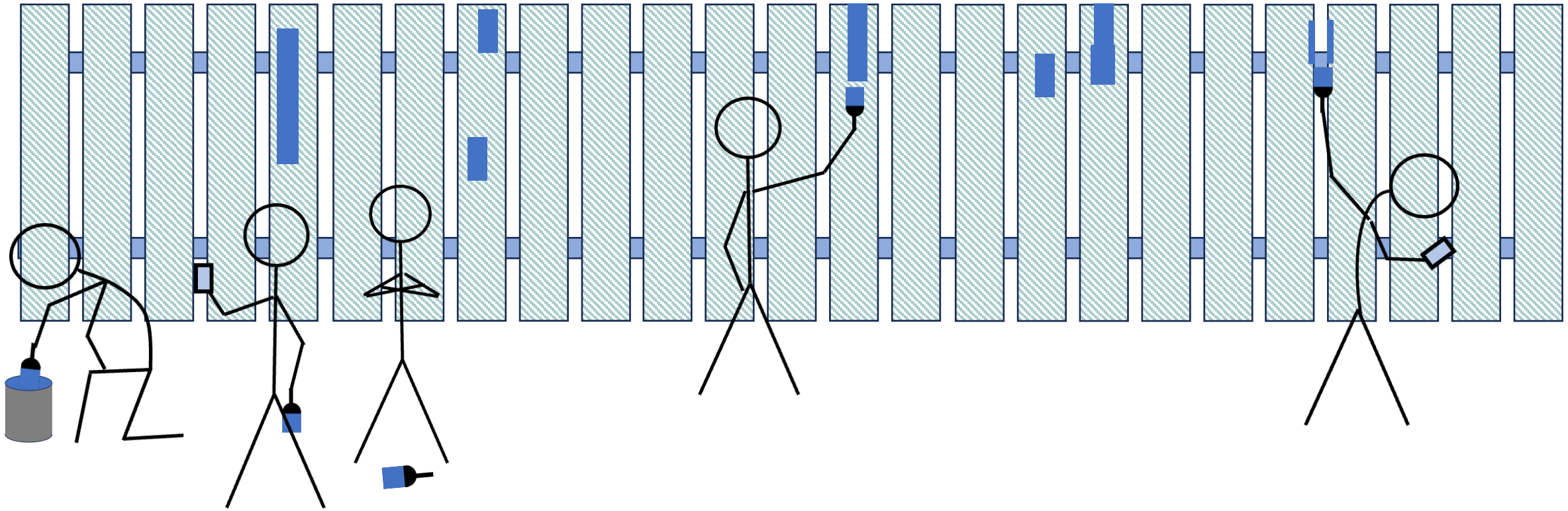
Speedup: How many times faster does your job complete when N people work in parallel? Ideally, Speedup = N.

Painting a fence: Sometimes, Reality Sucks

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take?

Reality

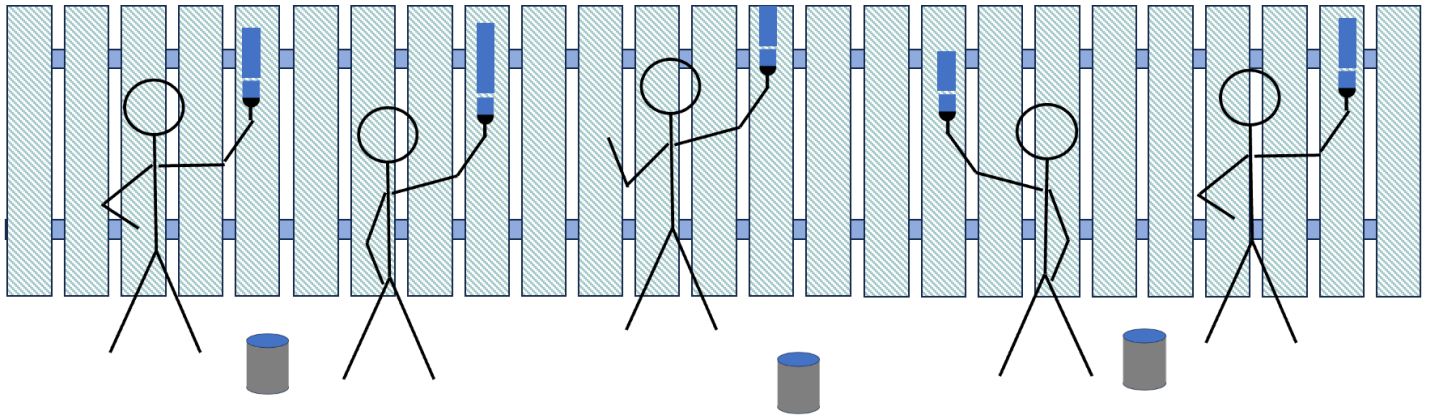
Not everyone works equally hard. More importantly with only one can of paint (a **shared resource**) people waste time waiting for their turn to dip their brush in the paint. These and other issues mean that you almost never achieve the ideal case.



Summary: the concept of parallelism

- This sequence of definitions explains the term “parallelism”
  - **Agent**: A person, process, thread, or other “unit of execution” that can work on a task.
  - **Problem Decomposition**: Break the problem into a collection of distinct tasks that are mostly (if not completely) independent.
  - **Serial**: When a single agent carries out a problem’s tasks one after the other.
  - **Parallel**: When the tasks execute and make forward progress at the same time.
  - **Parallelism**: the features of problem and its solution that support parallel execution.

- Assume you have multiple agents to carry out a set of tasks. You are not done until the last agent is done. It never goes as well as you hope.
  - Making the set of tasks (the work) for each agent **balanced** so they all finish at the same time is hard (**load balancing**).
  - Agents share resources (such as paint) and often waste time waiting for their turn for the shared resource (**contention**).
  - Coordinating the work of the multiple agents is extra work you wouldn’t have if you did the job in serial. This is called **parallel overhead**.
  - Recasting the problem into a collection of distinct and largely independent subproblems (tasks) can be difficult
  - There is almost always a small fraction of the work that cannot be done in parallel (**serial fraction**).
  - The serial fraction limits the number of agents that can productively help you complete the job



Now we can return to our originally planned lecture

Outline

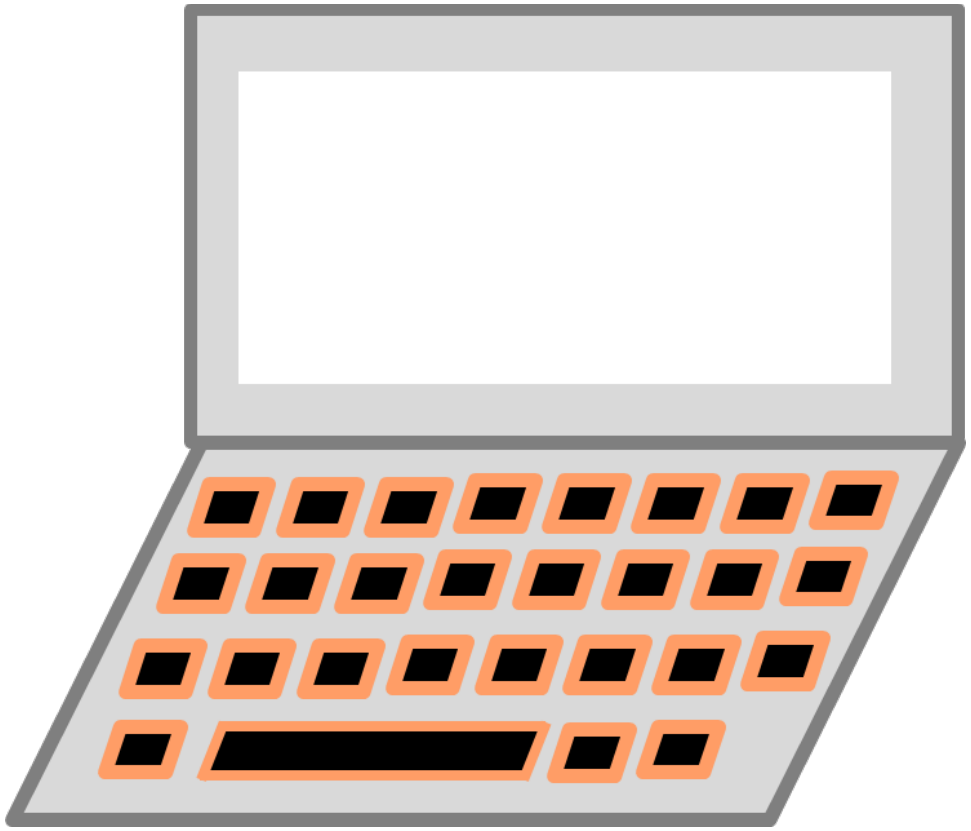
- Key concepts in computer architecture
- A brief history of supercomputing
- Programming languages and choice overload: Python, C/C++, Fortran

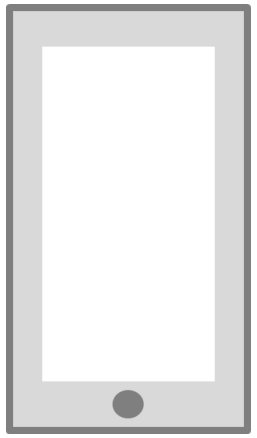
A quick assignment

- Find a scrap of paper and a pen … or use your imagination if you lack these objects

- You have 30 seconds …. Draw a picture of a computer

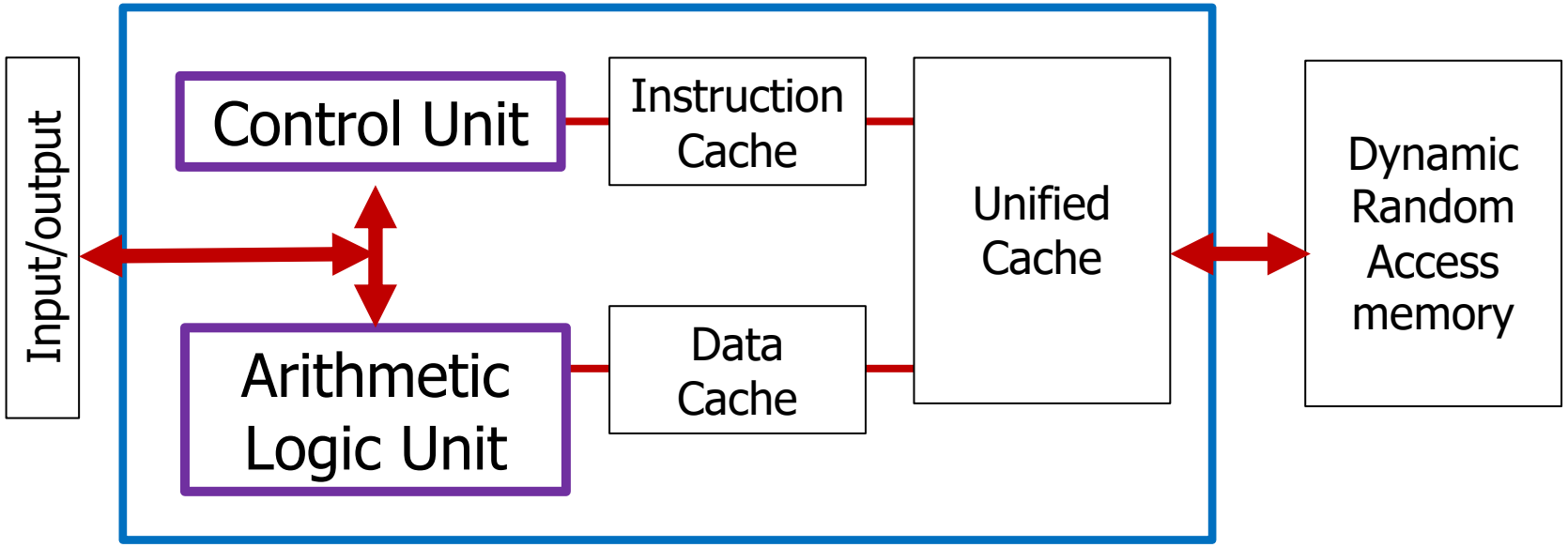
What did you draw?





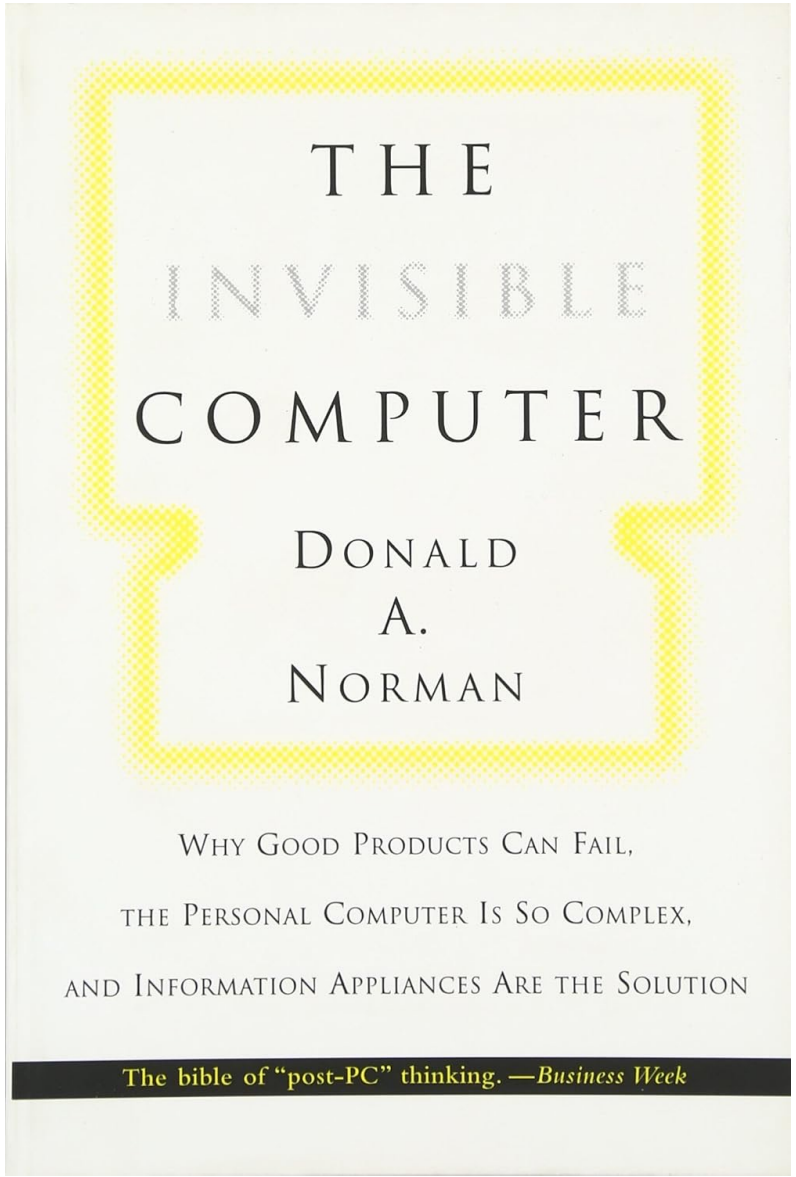
Most of the time I do this, people draw a laptop. Some draw a smart phone.

Tim Draws a computer



By the time we're done, you'll draw computers this way as well.

Computing for Humans



Don Norman, trained in Electrical Engineering and cognitive psychology, pioneered the idea of "Human centered design" in computer system design.

Joining Apple in 1993, he helped solidify the company's approach to design … taking existing technology (e.g. Apple did not invent the smart phone) and making it better by designing it around user experience.

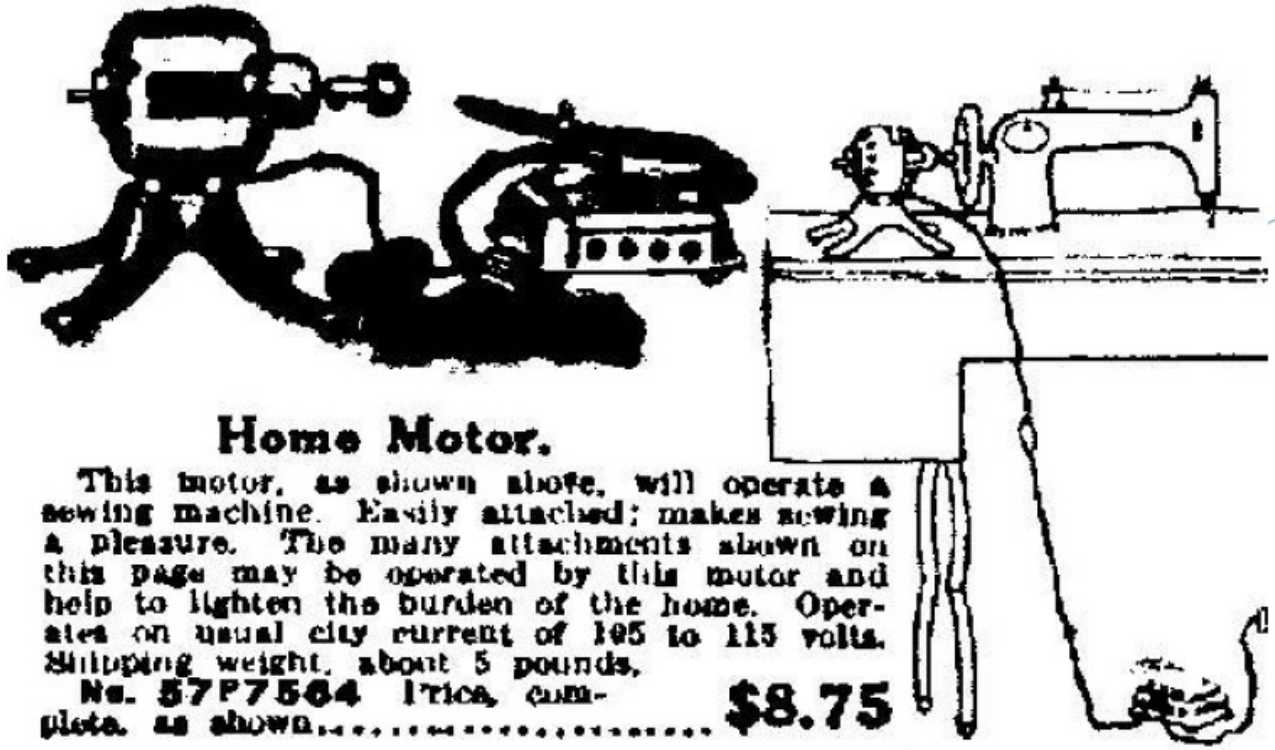
He summarizes his thinking in the famous book "The Invisible Computer" published in 1998.

An example from the book “The Invisible Computer”

How many **electric motors**. Have you purchased recently?

They used to be something you thought about and bought explicitly.

Now they are buried inside cars, appliances and other products … **they are invisible**



Home Motor.

This motor, as shown above, will operate a sewing machine. Easily attached; makes sewing a pleasure. The many attachments shown on this page may be operated by this motor and help to lighten the burden of the home. Operates on usual city current of 105 to 115 volts. Shipping weight, about 5 pounds.

No. 57F7564 Price, complete, as shown..... $8.75

Beater Attachment.

Whips cream and beats eggs, and many other uses will be found for these attachments when used in connection with the Home Motor. Parts include the stand, handle and the beater. Shipping weight, about 14 ounces.

No. 57F7585 Price..... $1.30

Churn and Mixer Attachment.

Used in connection with the Home Motor, makes a small churn and mixer for which you will find many uses. The attachments include the base, supports, mixer, handle and special cover for jar. Shipping weight, about 1¾ pounds.

No. 57F7582 Price..... $1.30

Fan Attachment.

Includes fan and guard which can be quickly attached to Home Motor, and will be a great comfort in hot weather. Shipping weight, about 14 ounces.

No. 57F6215 Price..... $1.30

A page from a Sears and Roebuck Catalog: 1918

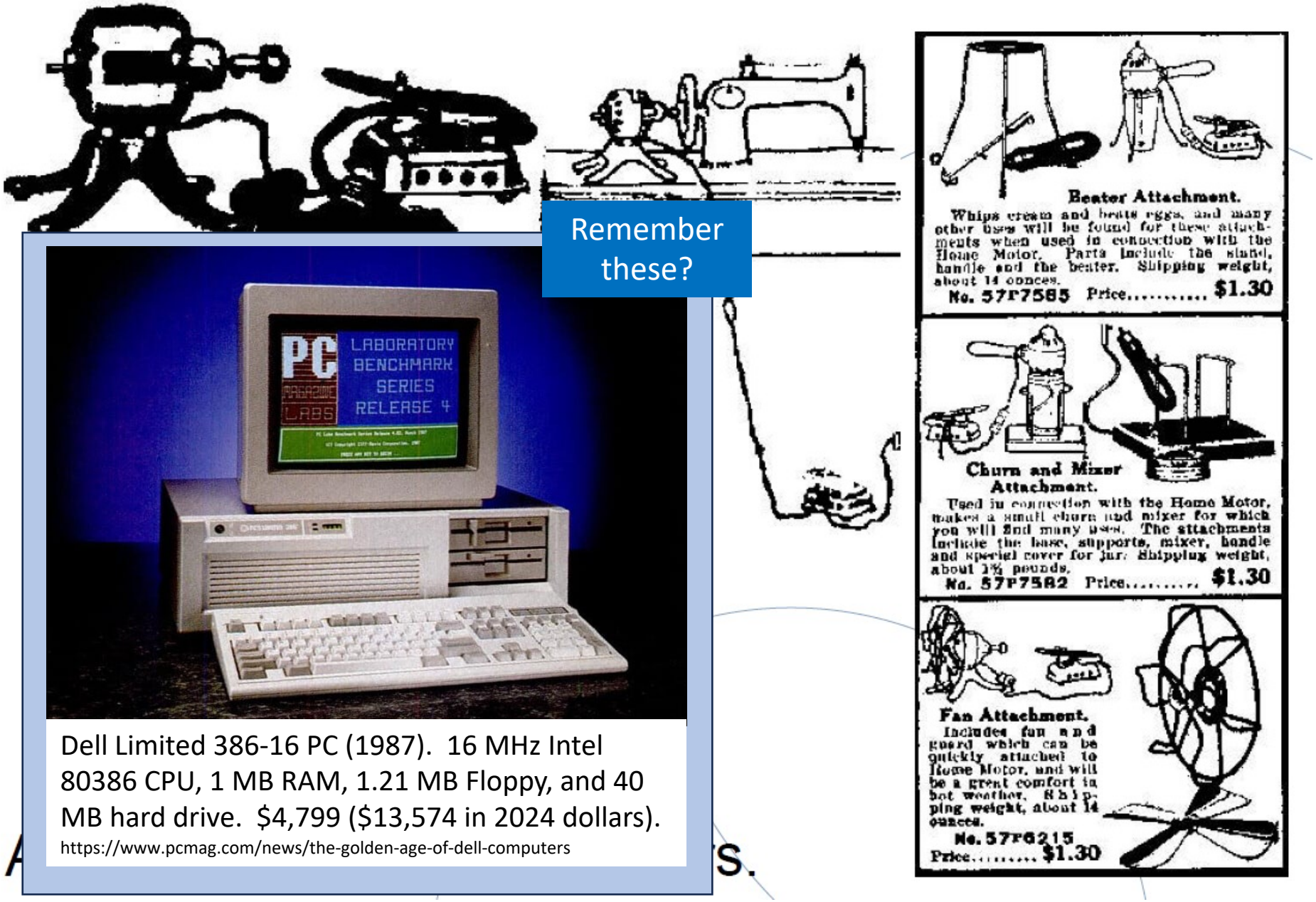
About $100 in today’s dollars.

Source: S. Edwards, C.S. Columbia Univ. http://www1.cs.columbia.edu/~sedwards/presentations/eitc2002.9up.pdf From Donald Norman, *The Invisible Computer, 1998.*

An example from the book “The Invisible Computer”

The PC is going the way electric motors.

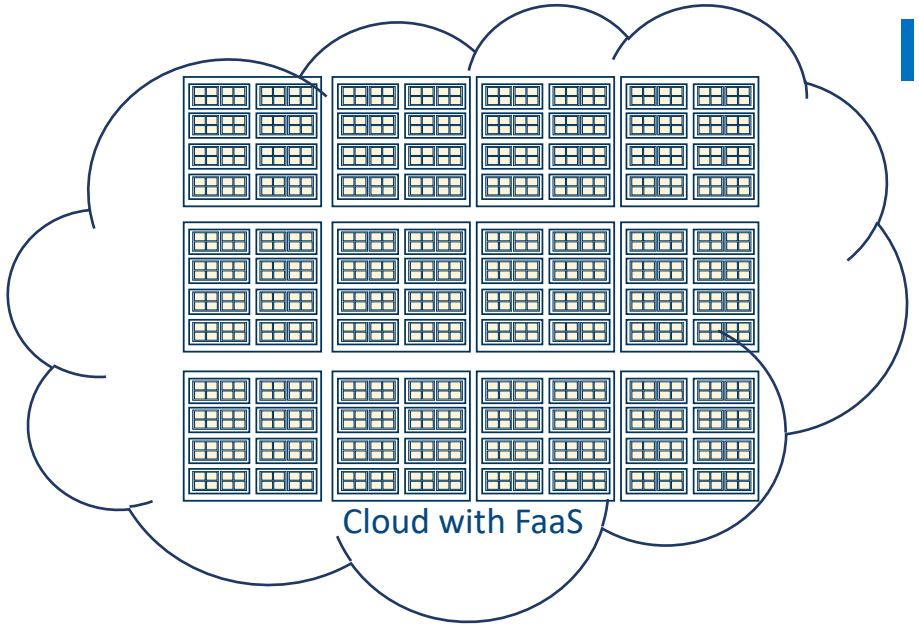
They are becoming invisible!



Dell Limited 386-16 PC (1987). 16 MHz Intel 80386 CPU, 1 MB RAM, 1.21 MB Floppy, and 40 MB hard drive. $4,799 ($13,574 in 2024 dollars).
https://www.pcmag.com/news/the-golden-age-of-dell-computers

Source: S. Edwards, C.S. Columbia Univ. http://www1.cs.columbia.edu/~sedwards/presentations/eitc2002.9up.pdf From Donald Norman, *The Invisible Computer, 1998.*

Invisible computers



FaaS: Function as a service. You see the function, not the hardware.

Nintendo Switch™ mobile gaming console. Nvidia Tegra X1 with 4 ARM Cortex A57 cores each with a NEON vector unit, 4 GB DRAM, and an Nvidia GM20B GPU (supports CUDA and OpenCL 1.2). 64 GB Memory.

Hello Barbie™, Uses AI in the cloud to generate conversation. Launched and terminated in 2015 due to privacy concerns over stored dialog by children.
Marvell 88MW300 ARM Cortex-M4F processor plus a 24 bit Nuvoton NAU8810 audio codec. 16 Mbit Gigadevice GD25Q16 Flash memory.
https://www.microcontrollertips.com/teardown-electronics-hello-barbie/

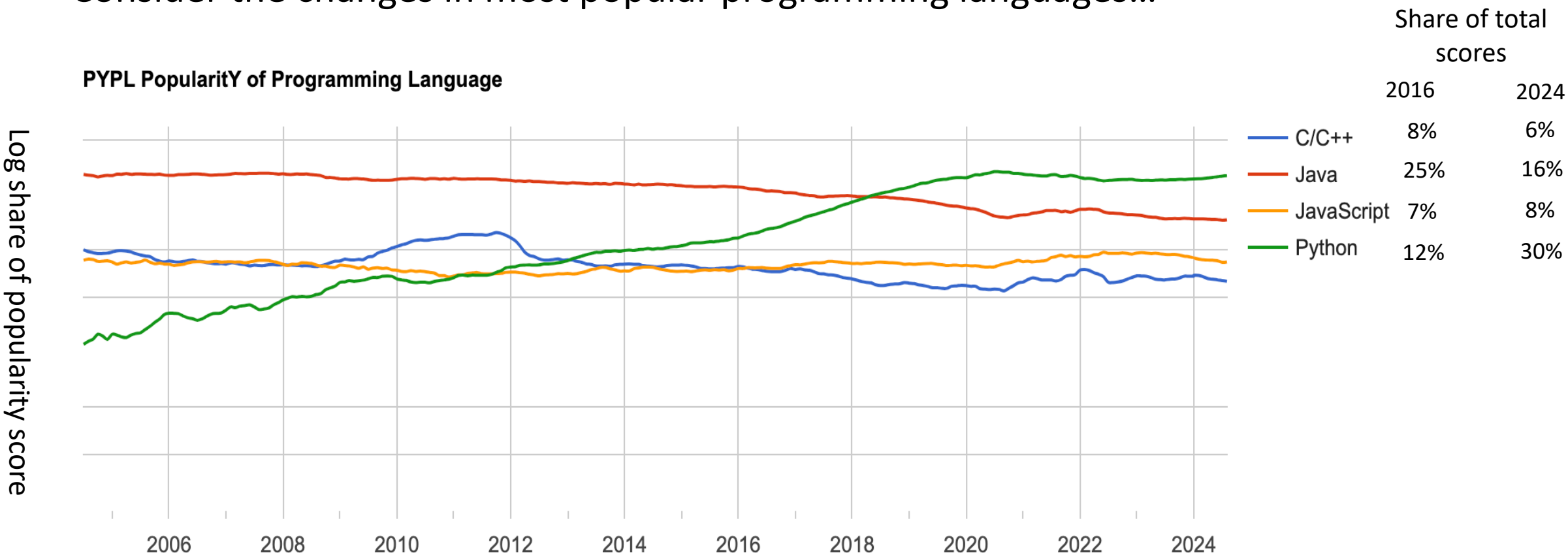
Mercedes-Benz SL roadster
http://www.extremetech.com/extreme/125621-mercedes-benz-over-the-air-car-updates

~50 computers on average in a car.
High-end cars have ~100

Roku streaming stick 4K, ARM Cortex A55, 1GB RAM
https://developer.roku.com/docs/specs/hardware.md

Python and friends are the languages of invisible computing

Consider the changes in most popular programming languages…



http://pypl.github.io/PYPL.html

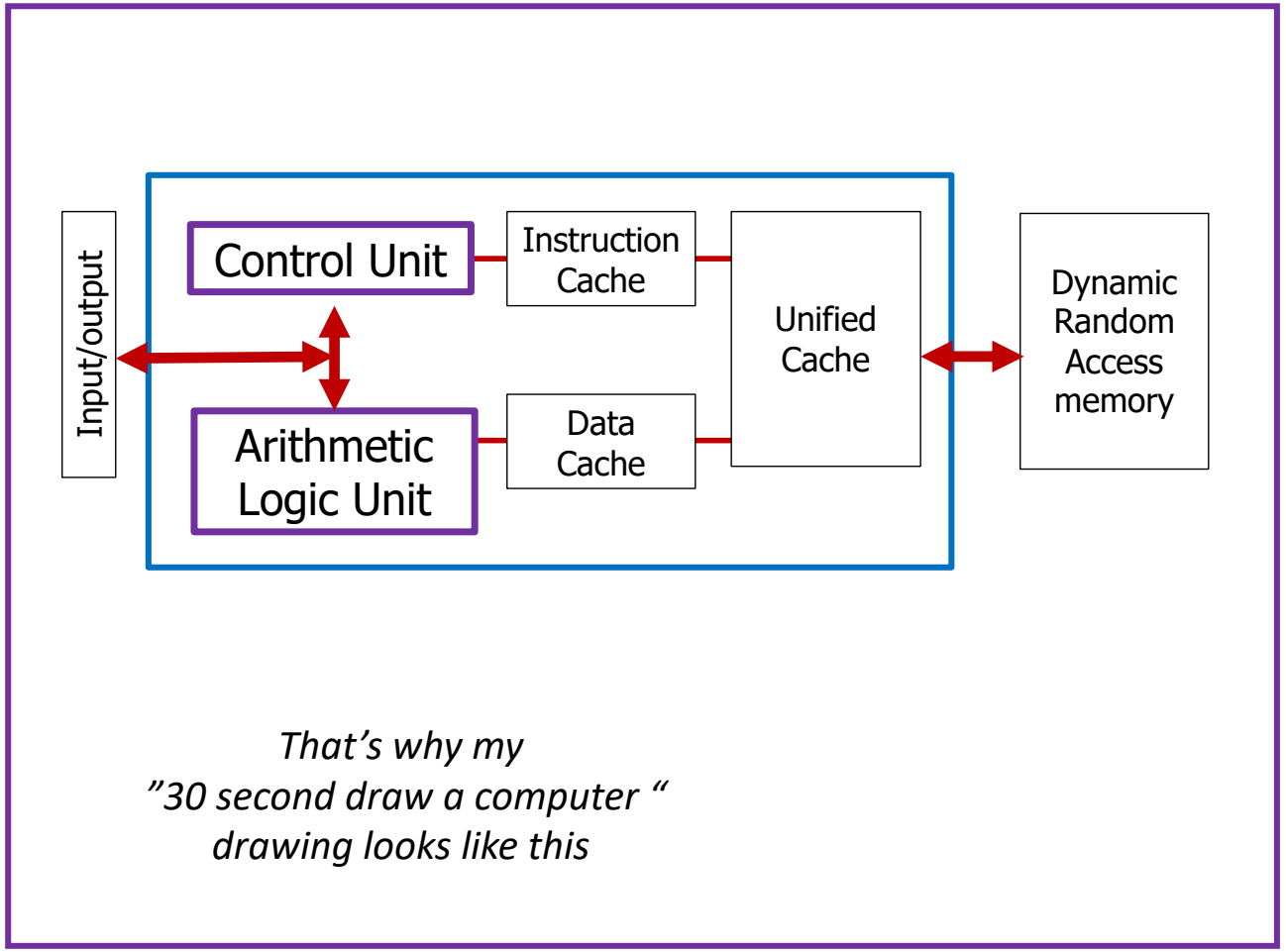
Invisible computers and scientific computing

- In scientific computing, we are often limited by performance or problem-size.

- We must understand what is happening inside our computers so we can design software that works well with the hardware and meets our needs.

We must make our computers visible

- And we must use programming languages that lets us directly deal with the hardware.
  - C, C++, and Fortran are the key programming languages of Scientific Computing.



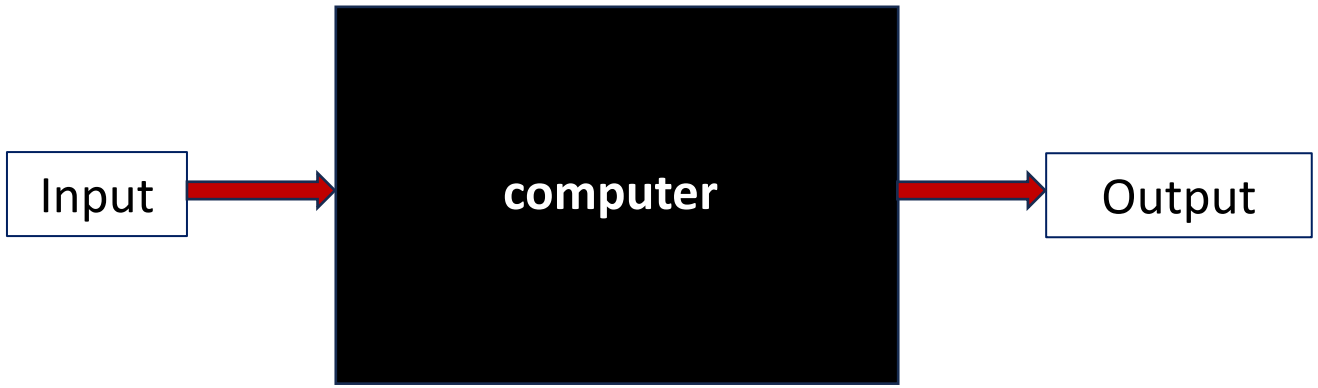
That's why my "30 second draw a computer " drawing looks like this

Let’s go back to basics ….

What is a computer?

What is a computer:

- **Computer:**
  - A machine that transforms *input values* into *output values*.

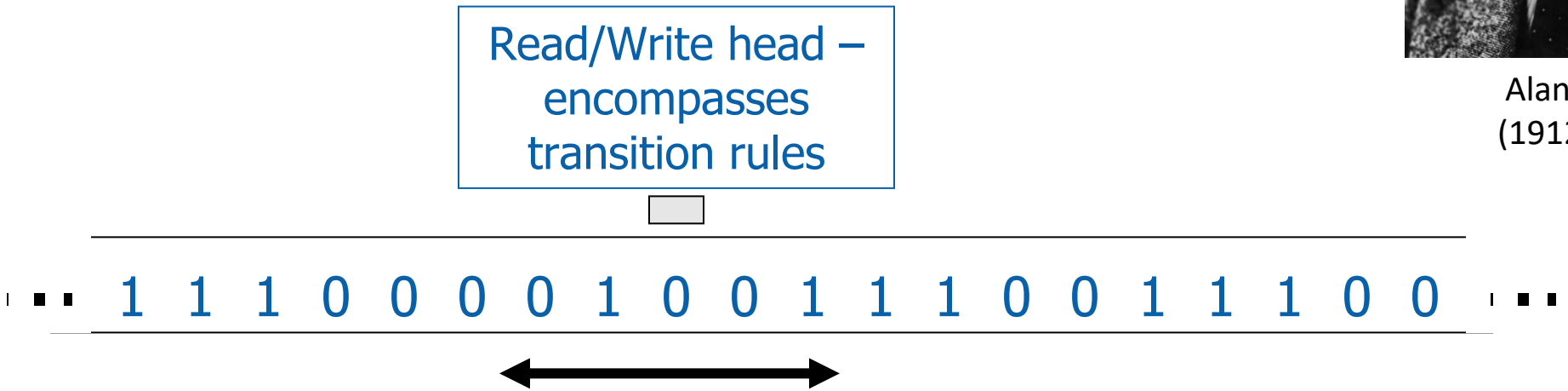


- The computer as a black-box is not very helpful. We need a bit more detail.

Computer models: Turing Machine

Alan Turing
(1912-1954)

- Alan Turing proposed a general model of a computer and showed that it was universal:



- Read an “infinite” tape of 1’s and 0’s. Based on the pattern of values, shift the tape, read values and write values. These are controlled by transition rules (i.e. a program)
- This was useful for proving mathematical theorems about computing, but not for actually working with computers.

Von Neumann or a “stored Program” Model

- John von Neumann proposed a more useful model where a computer consists of: (1) control unit, (2) Arithmetic-Logic unit (ALU), (3) registers that hold values close to the ALU, and (4) memory that holds both the data and the sequence of instructions(the program).

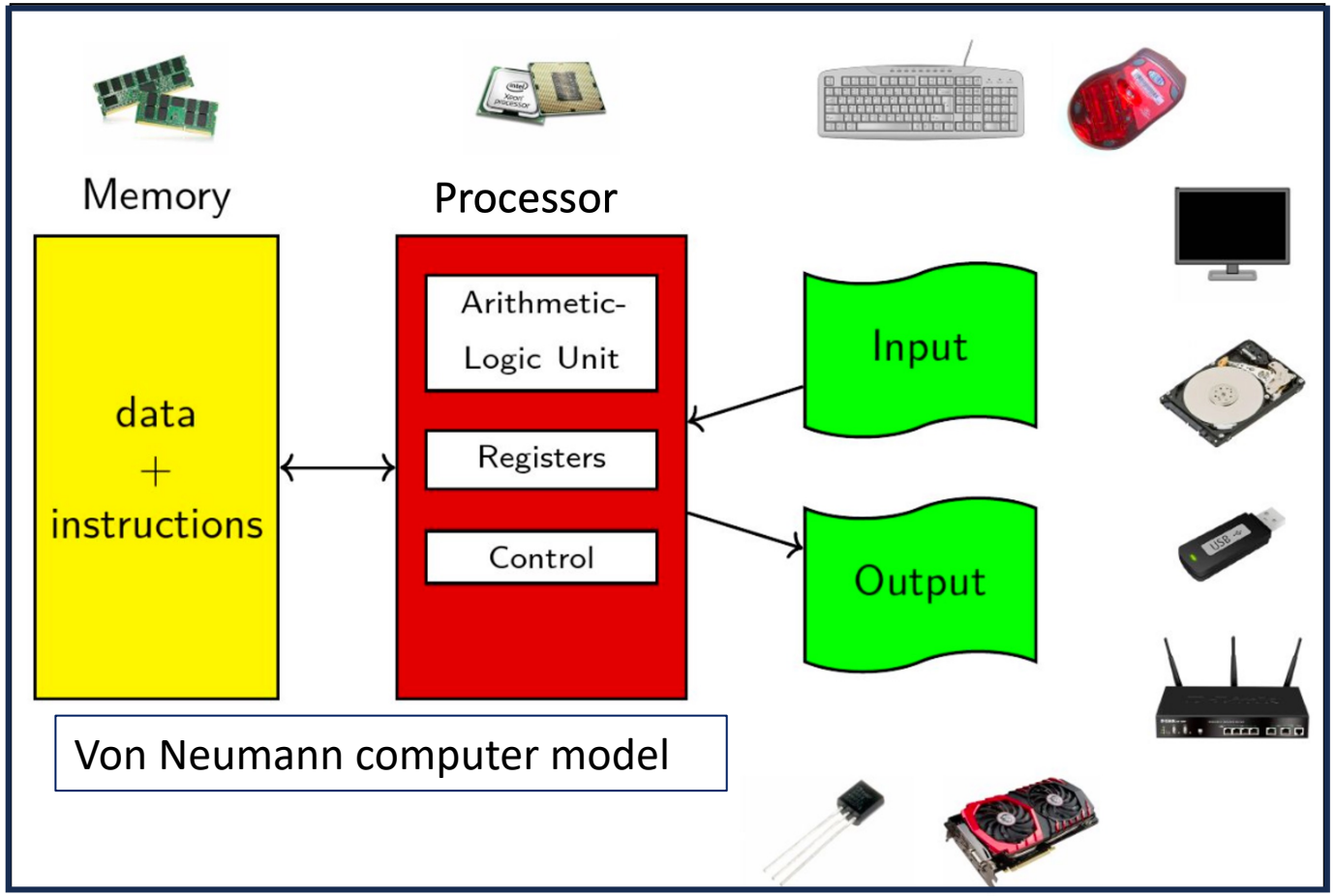
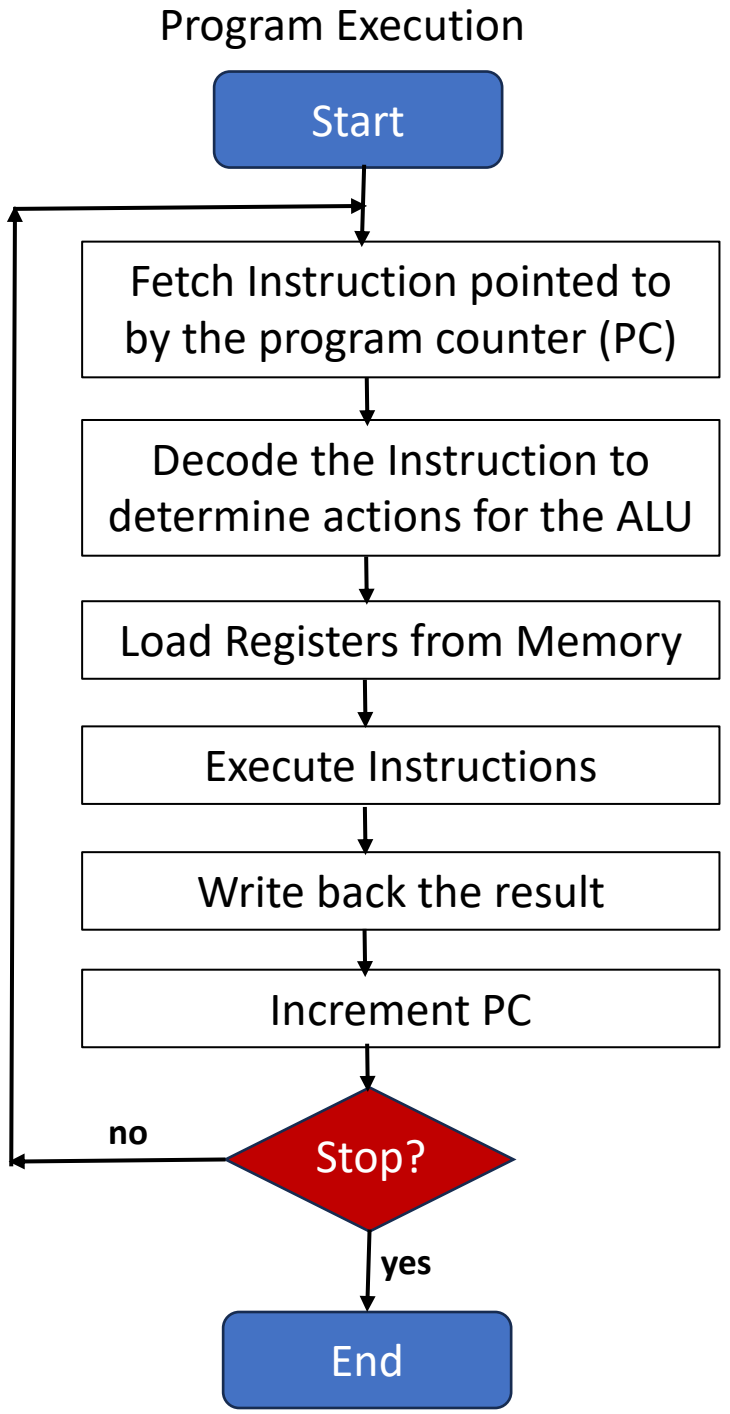


Image Source: Felice Pantaleo, CERN, ESC’23



Von Neumann or a "stored Program" Model

- John von Neumann proposed a more useful model where a computer consists of: (1) control unit, (2) Arithmetic-Logic unit (ALU), (3) registers that hold values for the ALU, and (4) memory that holds both the data and the program (the sequence of instructions).

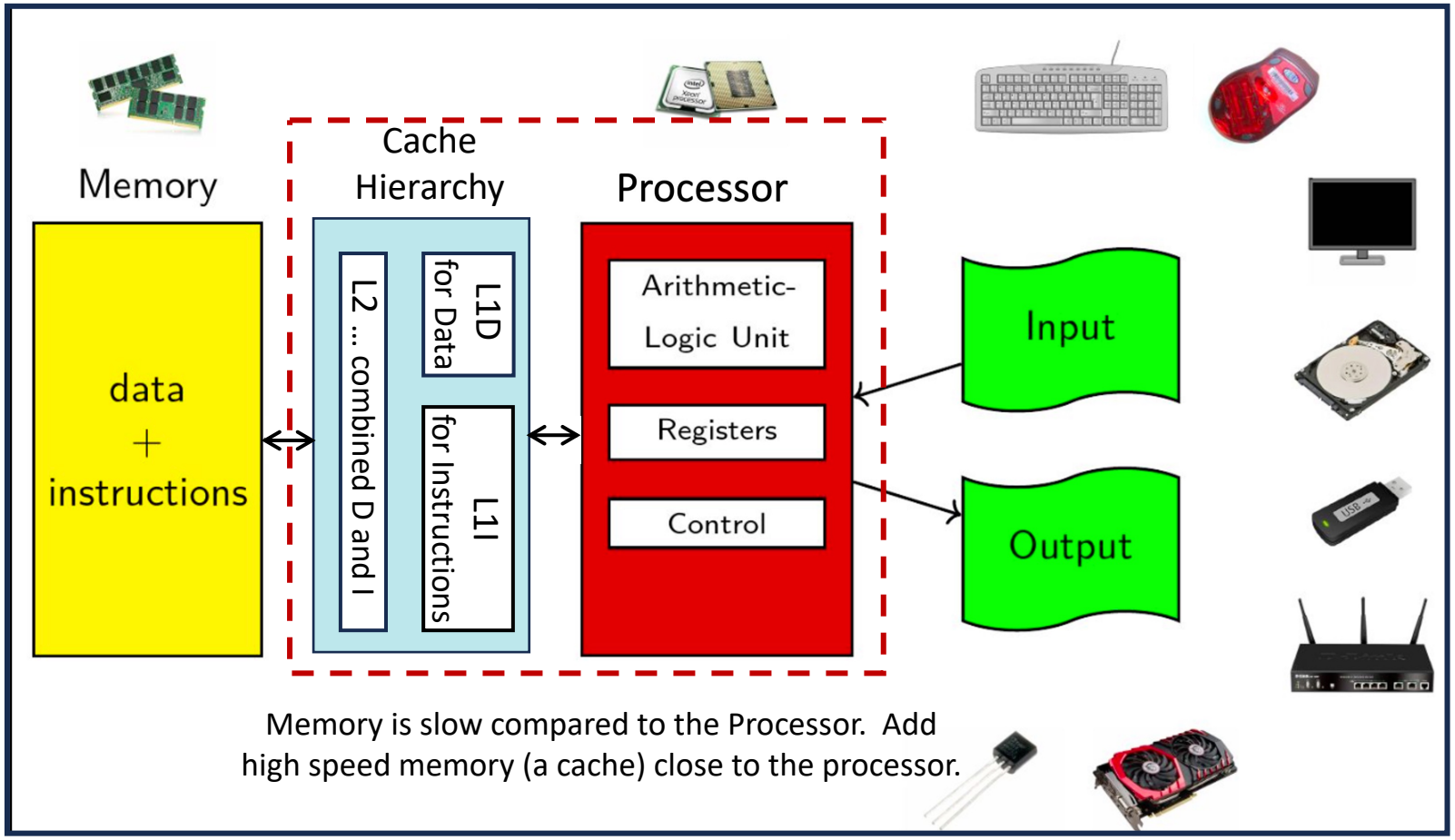
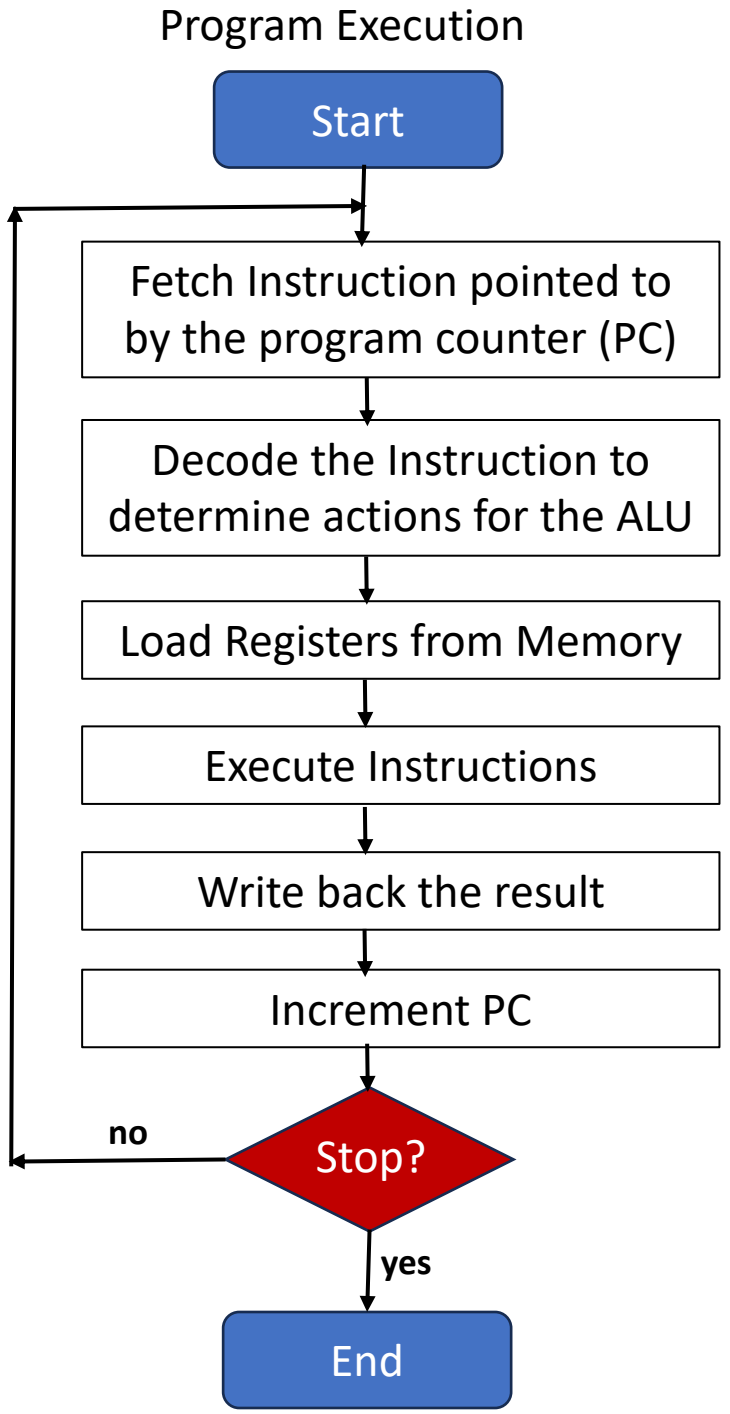


Image Source: Felice Pantaleo, CERN, ESC'23



Modern computers follow the von-Neuman model

To understand them more deeply, we need to look into computer architecture

Computer Architecture: Computer attributes visible to a user

- Computer architecture is composed of 3 topics

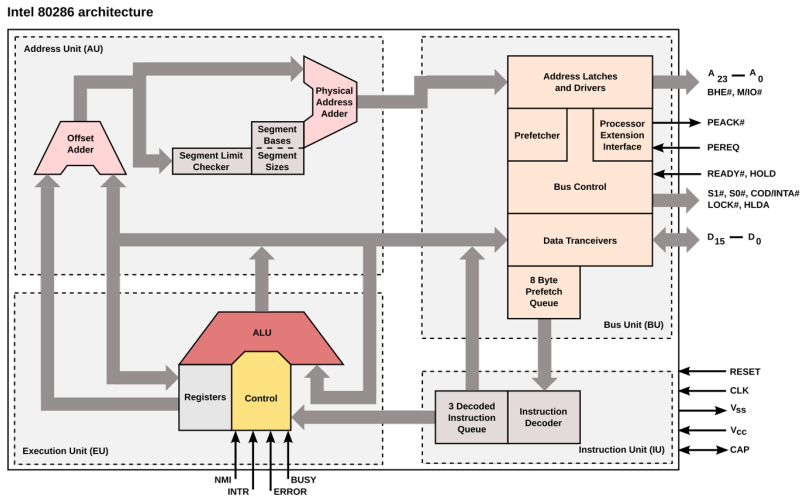
1. Instruction set architecture (ISA): the interface to the computer presented to a programmer

2. The microarchitecture: A design for how the ISA is implemented

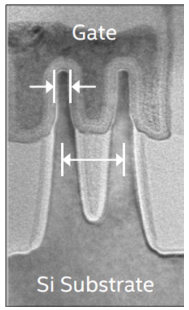
3. The hardware: The implementation of the computer itself (largely the silicon implementation)

```
12  .L3:
13          fcvt.d.w        fa5,a5
14          fcvt.d.s        fa4,fa4
15          addi    a5,a5,1
16          fadd.d  fa5,fa5,ft0
17          fmul.d  fa5,fa5,fa3
18          fcvt.s.d        fa5,fa5
19          fmul.s  fa5,fa5,fa5
20          fcvt.d.s        fa5,fa5
21          fadd.d  fa5,fa5,fa1
22          fdiv.d  fa5,fa2,fa5
23          fadd.d  fa5,fa5,fa4
24          fcvt.s.d        fa4,fa5
25          bne     a0,a5,.L3
26          fmul.s  fa0,fa0,fa4
27          ret
```

RISC-V assembly code for the “pi program loop” generated by gcc -O3



Intel 80286 microarchitecture



Electron microscope image of an Intel transistor for the 14 nm process technology

Transistor image source: https://www-ssl.intel.com/content/dam/www/public/us/en/documents/technology-briefs/bohr-14nm-idf-2014-brief.pdf
80286 image source: By Appaloosa - https://commons.wikimedia.org/w/index.php?curid=6902962
Assembly code from the compiler explorer at https://godbolt.org/

Instruction Set Architecture (ISA)

- The essence of computer architecture is the instruction set architecture (ISA), which is the interface to the hardware presented to a programmer … it is how software talks to hardware.
- Two major classes of ISA
  - CISC: **Complex instruction set Computer**. Large set of instructions to cover numerous special cases. Example: Intel® x86 ISA
  - RISC: **Reduced instruction set Computer**. Smaller set of instructions, easier to work with and implement. Example: ARMv8

ISA features	Intel x86* (CISC)	ARMv8 (RISC)
Class of ISA	Register memory ISA … operations can reference registers or memory.	Load-Store… can only access memory through load-store operations.
Memory address	Bytes addressing	Byte addressing, but objects must be aligned An object of size s bytes is aligned if A mod s = 0.
Registers exposed in architecture definition	16 general purpose and 16 floating point	31 general purpose 32 floating point registers
Encoding an ISA … Instruction widths	Variable length, ranging from 1 to 18 bytes. Can result smaller executables.	Fixed length, 4 byte Thumb instructions: 2-byte
Number of instructions	Exact count is difficult … over 3500	Base = 354, SIMD/FP = 404, SVE = 508 … total ~1266

ISA details are challenging to nail down. The Intel ISA manual is over 5000 pages.
Hence numbers on this slide convey a general sense of size and miss many details and special cases.

* these numbers are for the Intel® 64 x86 ISA. * SIMD: single instruction multiple data or vector instructions. FP: floating point SVE: Scalable vector instructions

Instruction sets: Complex (CISC) vs Reduced (RISC)

```
1  void add_abc(double *a, double *b, double *c)
2  {
3      *c = *a + *b;
4  }
```

Compare assembly code for a simple function for CISC (x86-64) and RISC (ARM) processors

COMPILER EXPLORER

https://godbolt.org/

CISC

Ops work on registers and addresses in memory.

Complex but extra options for aggressive optimization

x86-64 gcc 14.2 -O2

A Output... Filter... Libraries Overrides

```
1  add_abc(double*, double*, double*):
2          movsd   xmm0, QWORD PTR [rdi]
3          addsd   xmm0, QWORD PTR [rsi]
4          movsd   QWORD PTR [rdx], xmm0
5          ret
```

- Load double at address [rdi] into register xmm0
- Add double at address [rsi] to xmm0, put result in xmm0
- Store double in xmm0 to address [rdx]
- Branch to return address on the stack

RISC

All ops on registers

Consistency means smaller and simpler instruction set

ARM GCC 14.2.0 -O2

A Output... Filter... Libraries Overrides

```
1  add_abc(double*, double*, double*):
2          vldr.64 d16, [r0]
3          vldr.64 d17, [r1]
4          vadd.f64        d16, d16, d17
5          vstr.64 d16, [r2]
6          bx      lr
```

- Load double at address [r0] into register d16
- Load double at address [r1] into register d17
- Add double in d17 to double in d16, put result in d16
- Store double in d16 to address [r2]
- Branch to return address lr

Computer Architecture: CISC vs RISC

- Intel pioneered mass-market-computing through the IBM PC.

- Intel's x86 ISA is a CISC instruction set and that played a key role in the history of computing.
  - Starting with the Intel 8086 CPU in 1978 and continuing to today as the dominant architecture for servers and laptops.

- ARM: the dominant commercial RISC vendor starting with the ARM1.
  - ARM1, 1985, 25 thousand transistors compared to Intel's 1985 CPU (i386) with 275 thousand transistors.

- **Every new CPU ISA since 1980 has been based on a RISC ISA**.
  - As we'll see later, internal to a modern CISC CPU from Intel is a RISC execution engine.

The "golden handcuffs" of legacy applications will keep CICS/x86 around for many years. But in terms of innovative designs and the future, **RISC has "won"**.

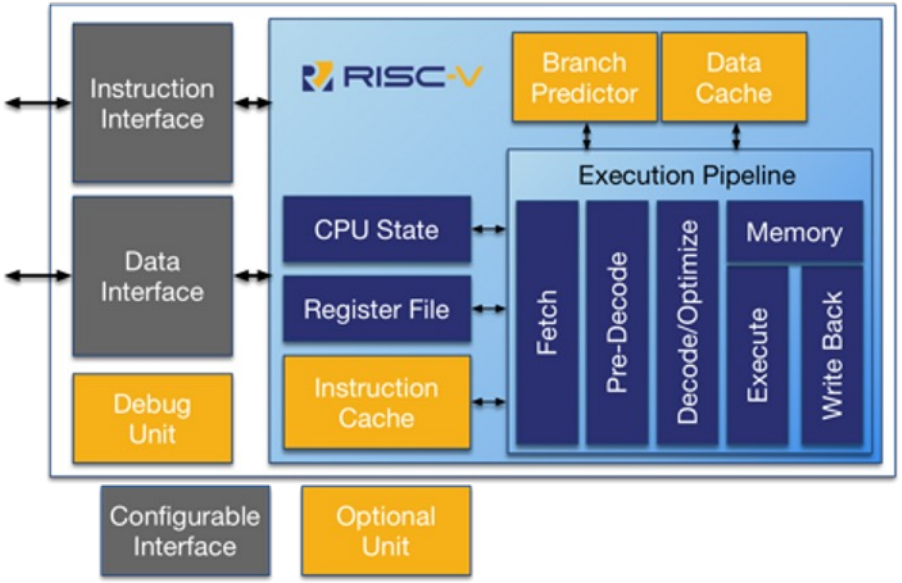
RISC across the computer industry

- ARM licenses CPU designs for others to implement.
  - Used extensively in cell-phones, tablets, Apple laptops, embedded processors, and other devices. The number one CPU by volume.
  - ARM is moving into Servers and HPC … For example, Nvidia is shipping chips for HPC using ARM (Nvidia® Hopper™)
  - ARM charges a royalty for each unit sold and vigorously protects it's monopoly over the ISA.

- Just as Open Source Software changed the nature of the software industry, an Open Source ISA will change the hardware industry.

- RISC-V (pronounced RISK-Five) is an open source ISA.
  - 2010: research project in the Computer Science Department at the University of California, Berkeley.
  - 2011: The first RISC-V specifications were released.
  - 2015: RISC-V International was established to promote adoption and standardization of the RISC-V ISA. Now has over 200 members.

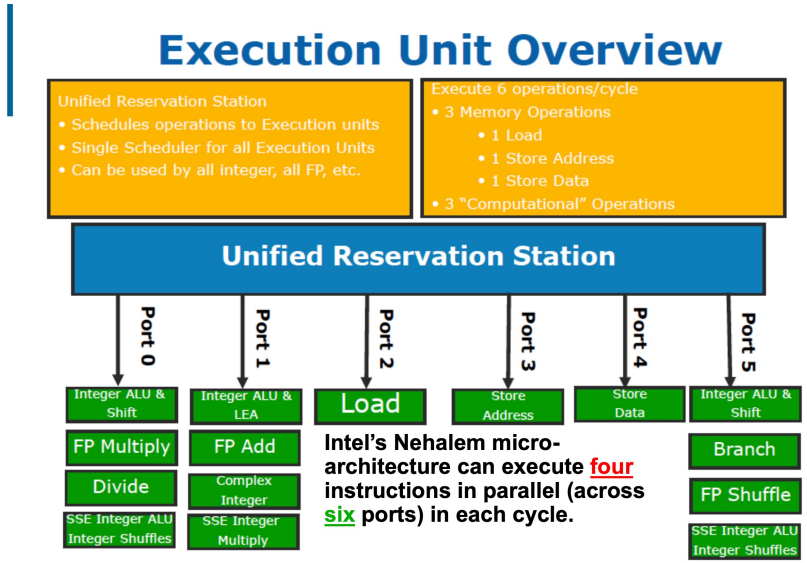
RISC-V block diagram



- Load-store ISA
- 32 bit instruction format
- RISC-V base ISA has only 50 instructions compared to 354 for ARM8 base ISA

Big companies like Apple and Google will tire of paying royalties per unit to ARM. The future is RISC-V

Microarchitecture … the details for how an architecture is implemented



A modern CPU

Intel® X86 architecture

Sandy Bridge Microarchitecture

By the time we are done, **most** of this will make sense to you.

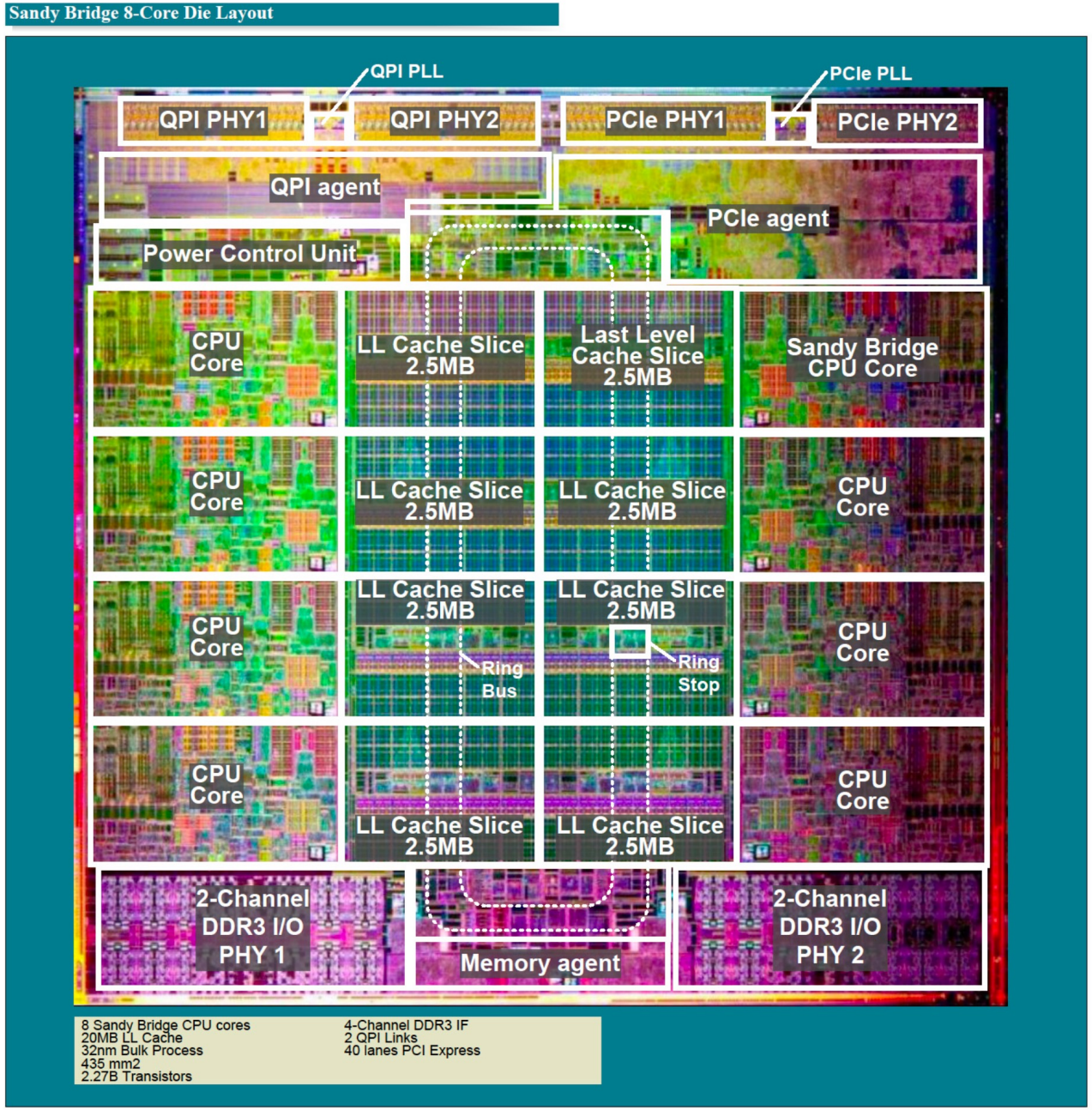


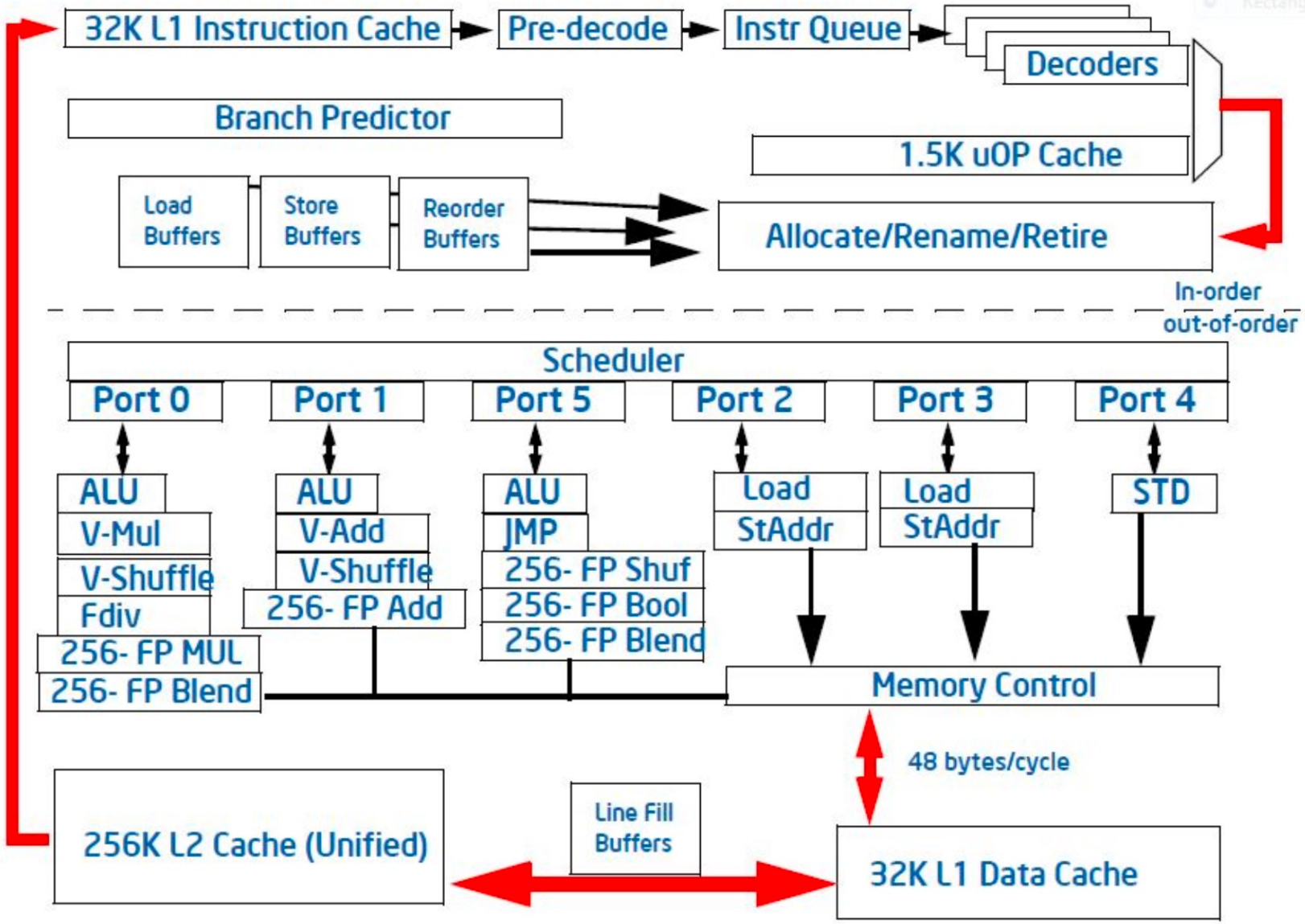
Image Source: Victor Eijkhout, processor architecture lecture, 2023

Structure of a Sandy Bridge CPU core

- The point of a microarchitecture is to support the architecture with aggressive optimization to achieve high performance.
- A modern microarchitecture can be extremely complex … both for CISC and RISC chips

By the time we are done, **much** of this will make sense to you.

Block Diagram of an Intel Sandy Bridge Core (used in Core i7, i5, i3 CPUs)



Launched 2011 and the core microarchitecture at Intel until 2013

The key to performance inside a CPU: Instruction level parallelism (ILP)

- The fundamental equation of quantitative architecture analysis:

$$Time_{CPU} = N_{instructions} * \frac{cycles}{Instruction} * \frac{seconds}{cycle}$$

$N_{instructions}$ ≡ Number of instructions in an executable

$$\frac{cycles}{Instruction} = \frac{Total\ number\ cycles\ to\ execute\ a\ program}{N_{instructions}} = \text{CPI}$$

- An architecture that lets multiple instructions make forward progress each cycle reduces the Cycles per Instruction (CPI) … if all goes well, we can design architectures where CPI < 1.

- We do this with Instruction Level Parallelism (ILP)

- Main ways to implement ILP in a design:
  - Pipelined execution: overlap execution across multiple stages
  - Superscalar execution: multiple-issue of independent instructions
  - branch prediction, speculative execution, prefetching
  - out-of-order execution

Let's go back to the late 80's and 90's to look at Instruction level parallelism through the lens of x86 CPUs

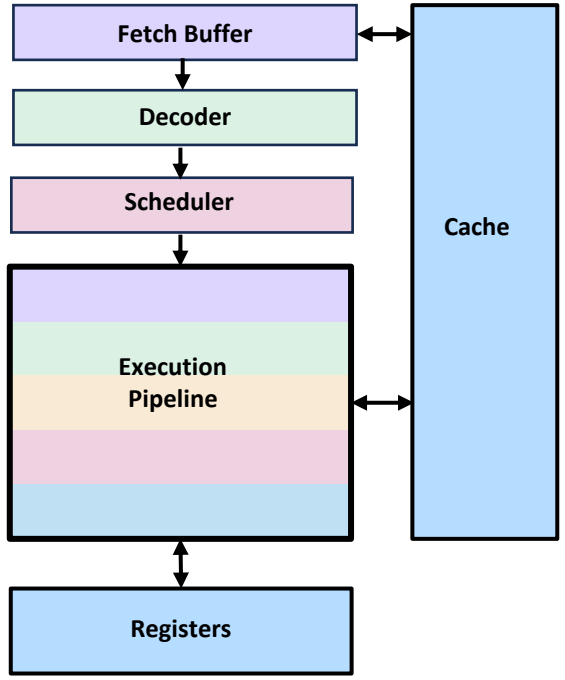
X86 … "an architecture that is difficult to explain and impossible to love"

Hennessy and Patterson, 2nd ed, page D-2

While we focus on x86 chips, all the techniques we'll discuss are used in RISC chips as well

Pipelining

i486™ 1.2 Million transistors, 50 Mhz



… plus an integrated floating point unit

- Intel added pipelined instructions to i486 in 1989
- More than doubled the performance compared to a i386 at the same clock rate.
- The five stage i4586 pipeline, one cycle per stage

F	Fetch an instruction from the instruction cache.
D1	Decode the instruction.
D2	Translate memory addresses and displacements for the instruction
EX	Execute the instruction.
WB	Retire the instruction, write results back to registers and/or memory.

Five instructions going through the pipeline

Cycles (each stage takes one cycle)

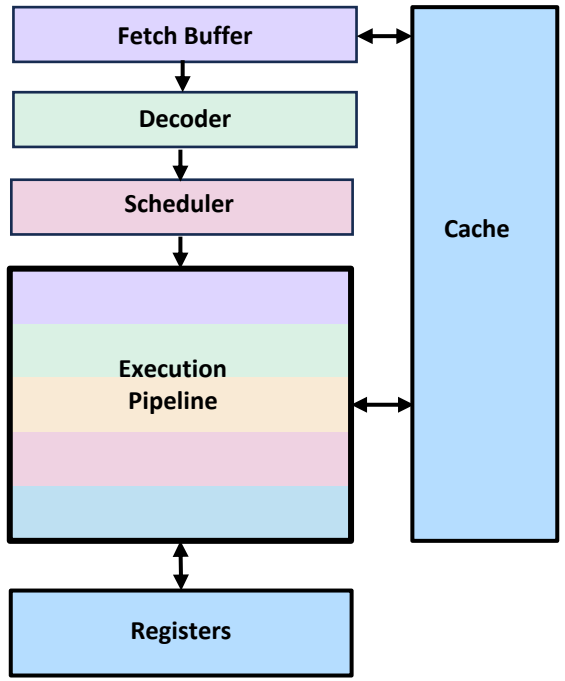
```
    1    2    3    4    5    6    7    8    9
1:  [F ][D1][D2][EX][WB]
2:       [F ][D1][D2][EX][WB]
3:            [F ][D1][D2][EX][WB]
4:                 [F ][D1][D2][EX][WB]
5:                      [F ][D1][D2][EX][WB]
```

Note: after five cycles, the pipeline is full and we get a result per cycle.

Pipelining Performance

1989

i486™ 1.2 Million transistors, 50 Mhz



… plus an integrated floating point unit

- Given the following definitions:

$$n = number\ of\ operations$$
$$\ell = number\ of\ stages$$
$$\tau = number\ of\ stages$$

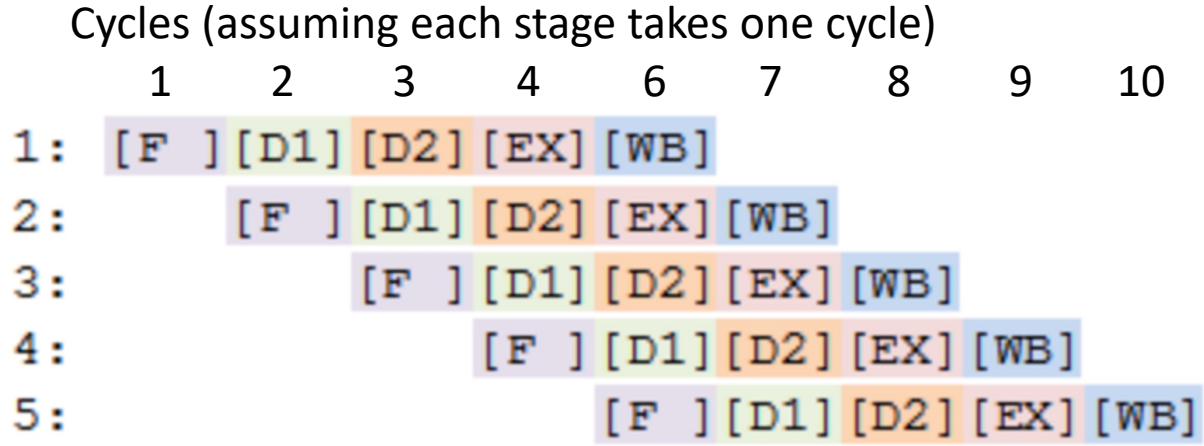
- Runtime without pipelining: $t(n) = n\ell\tau$

- With pipelining you need to setup the pipeline (costs s cycles) and fill the pipeline (costs ℓ cycles) at which point you have completed one instruction. Then for the next (n - 1) cycles you get one result per cycle. The runtime with pipelining is:

$$t(n) = (s + \ell + n\text{-}1)\,\tau$$

- For n much greater than $(s + \ell - 1)$, $t(n) \approx n\,\tau$, so the code runs ℓ times faster.

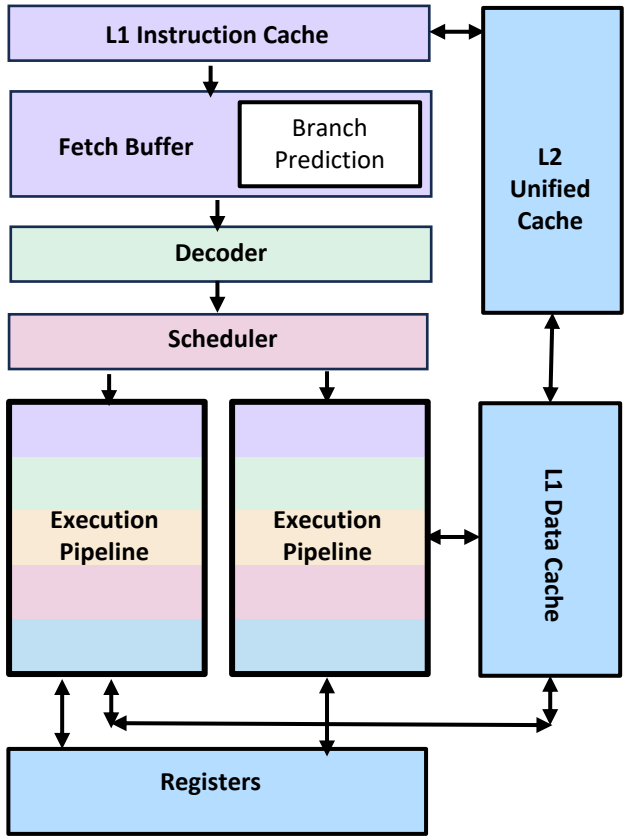
Five instructions going through the pipeline



Note: after five cycles, the pipeline is full and we get a result per cycle.

Superscaler + branch prediction

Pentium™ 3.1 Million transistors, 66 Mhz, 5 stage pipeline



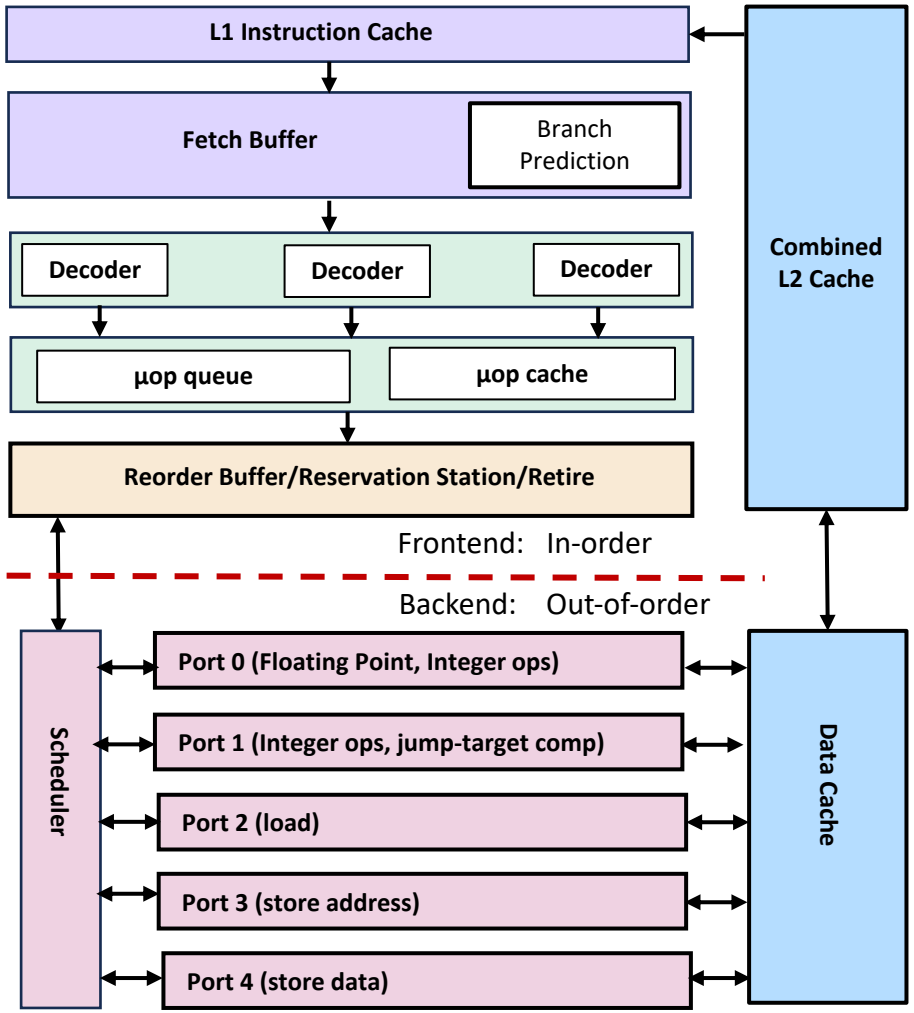
… plus an integrated floating point unit shared between pipelines

- The Pentium CPU was introduced in 1993 with major innovations
  - **Superscalar execution**: Added a second execution pipeline so it could keep two pipelined instruction streams in flight at the same time.
  - **Branch prediction**: Speculate on branches a program might take to load next instructions and get a head start should the branch be taken.
  - **Separate L1 caches for data and instructions**: A unified second level cache that holds data and instructions but separate L1 caches for data and instructions to reduce conflicts.

- Branch prediction is important. With two execution pipelines, its challenging to keep enough work in the execution units so they are fully occupied.

Out of Order (OOO) + speculation

Pentium Pro CPU, 5.5 Million transistors, 200 MHz,
14 stage pipeline

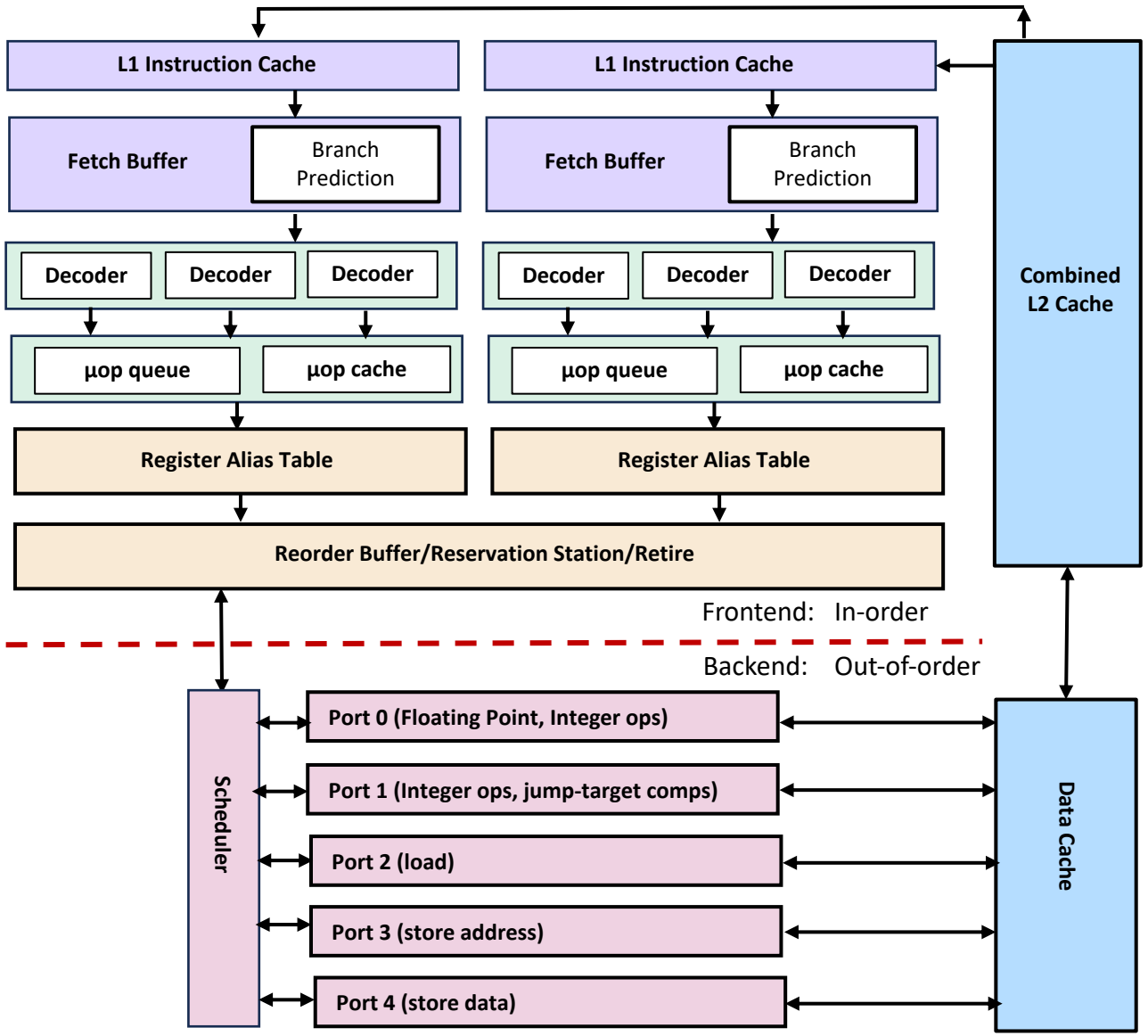


- The Pentium Pro CPU was a major performance upgrade and made Intel CPUs suitable for demanding technical computing workloads (and was used inside the computer ranked as the fastest computer in the world from 1996 to 2000).
- It added the following microarchitectural innovations:
  - Input CISC instructions were decoded into fixed-length, load/store micro-ops (µop). This made the internal execution engine inside the Intel CPU a RISC chip.
  - Micro-ops were reordered and executed based on availability of data … hence compared to input CISC instructions, they execute Out of Order (OOO)
  - Micro-ops complete in-order … i.e., they are retired in an order consistent with the input program
  - Register usage efficiency greatly enhanced by renaming them to avoid spurious conflicts due to register naming in code.
  - Dynamic speculative execution to generate enough work to fully occupy the execution units … a powerful capability but branch misprediction is very expensive as all involved pipelines must be flushed.

Simultaneous Multithreading … or the Intel Marketing term, hyperthreading

2002

Pentium 4 CPU, 125 Million transistors, 3.06 GHz, 20 stage pipeline



- Out of order execution of pipelined execution units creates so much opportunity for instruction level parallelism that it can be challenging to keep the resources fully occupied.
- Solution … replicate the in-order front end of the processor so two front-ends feed a single out-of-order backend.
- These two in-order front ends are managed as distinct threads by the OS typically with single cycle context switching overhead between them. We call these hardware threads.
- They are usually exposed to the operating system as an additional core … so the case hyperthreading case on this slide results in the OS treating the system having two cores.

Be careful with hyperthreading. If your work load can saturate the functional units on a CPU with a single thread, hyperthreading adds overhead and can slow down your code.

HPC programmers working highly optimized, compute bound codes often turn it off by default.

ILP is great, but we can get carried away

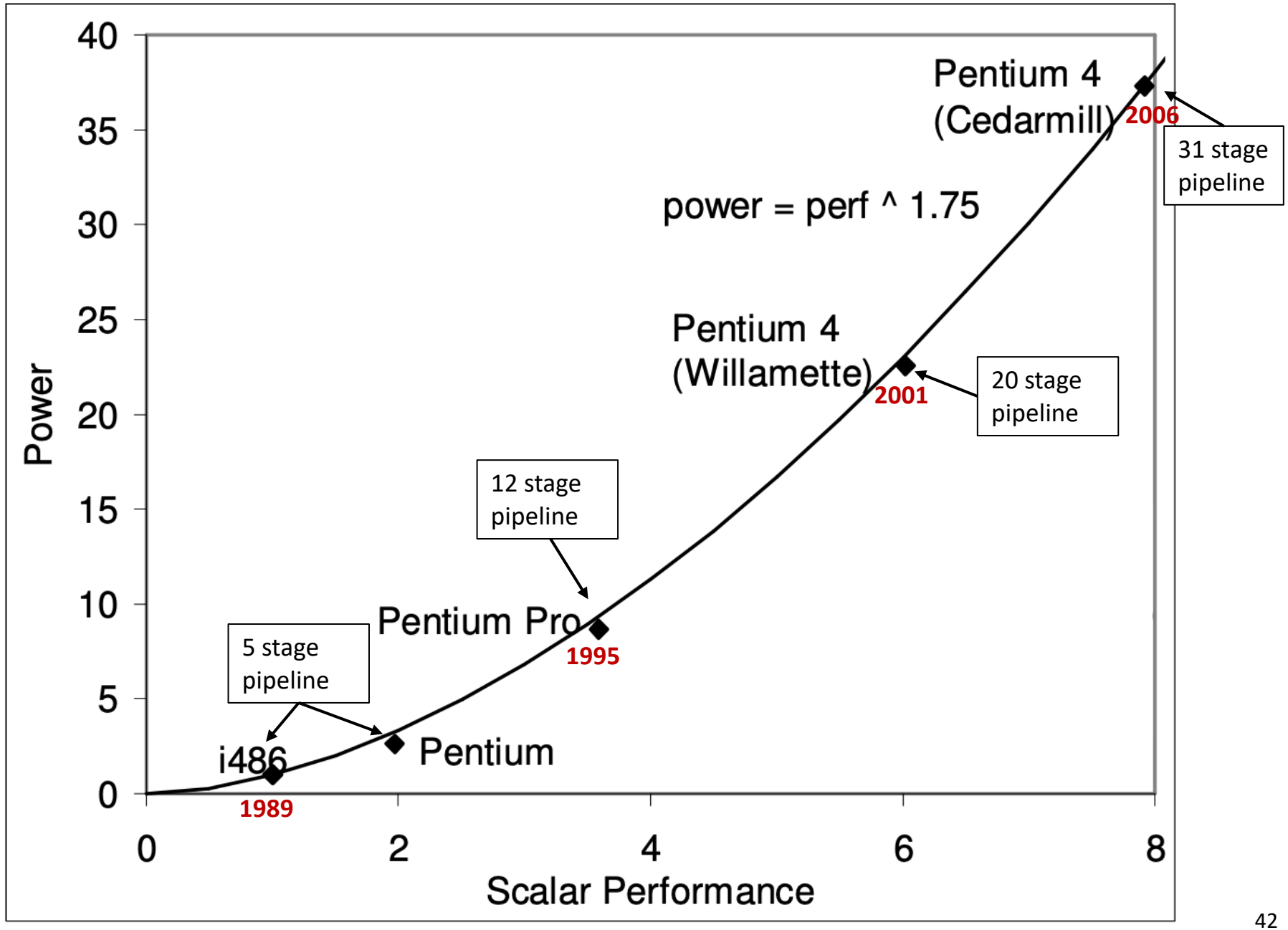
Normalized Power vs. scalar performance for Intel CPUs

Assume multiple generations of Intel CPUs using the same process technology as for i486.

Any changes are due to microarchitectural enhancements

This shows the unsustainable power demands of every deepening pipelines.

Power and Performance scaled to the i486 … e.g., Petium 4 is 6 times faster than an i486 but uses 22 times more power.



Energy per instruction Trends in Intel® Microprocessors, Ed Grochowski and Murali Annavaram, 2006

Normalized Power vs. scalar performance for Intel CPUs

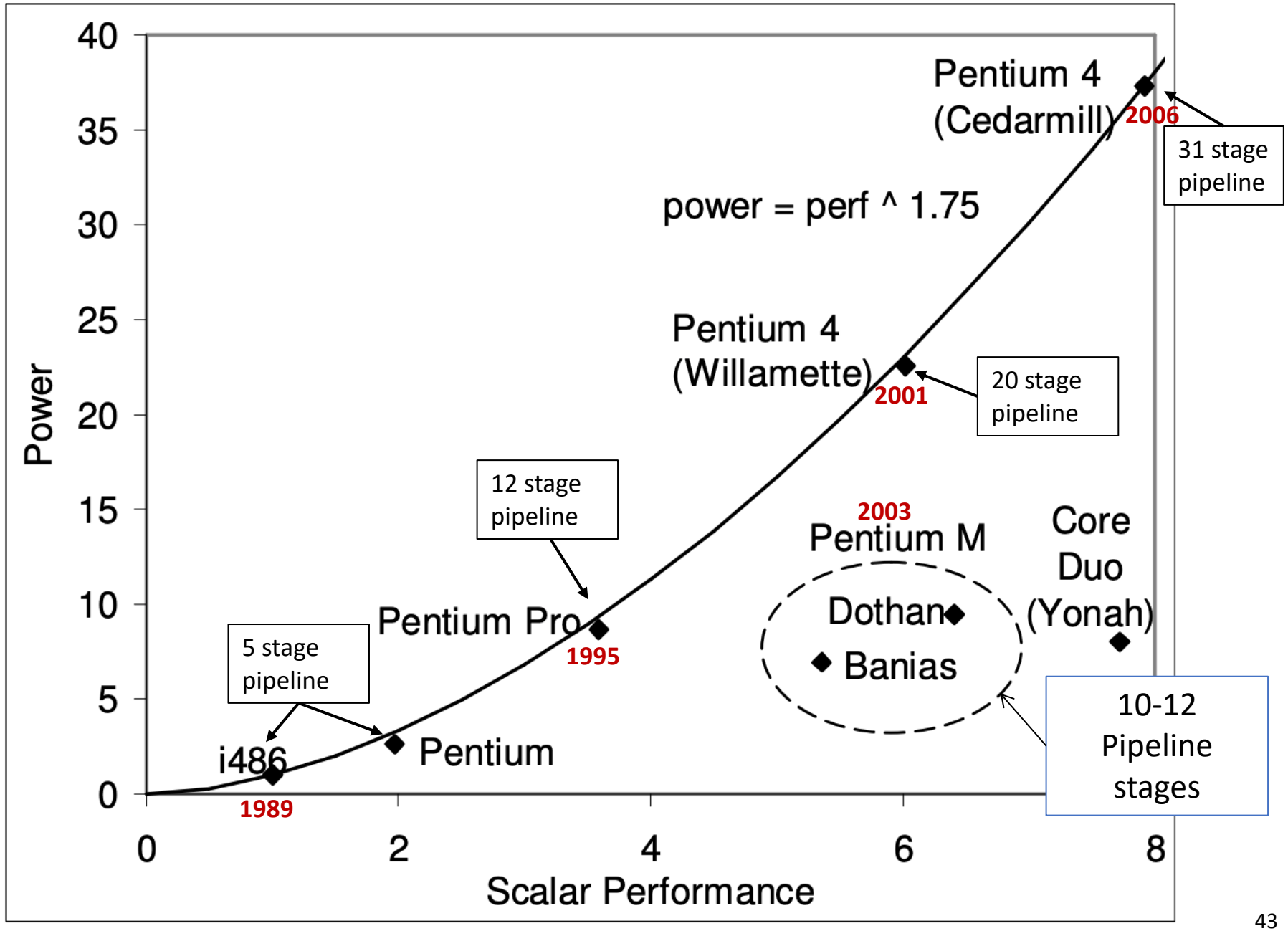
Assume multiple generations of Intel CPUs using the same process technology as for i486.

Any changes are due to microarchitectural enhancements

This shows the unsustainable power demands of every deepening pipelines.

This plot ended around 2006. More modern CPUs have pipeline depths of 14 to 20

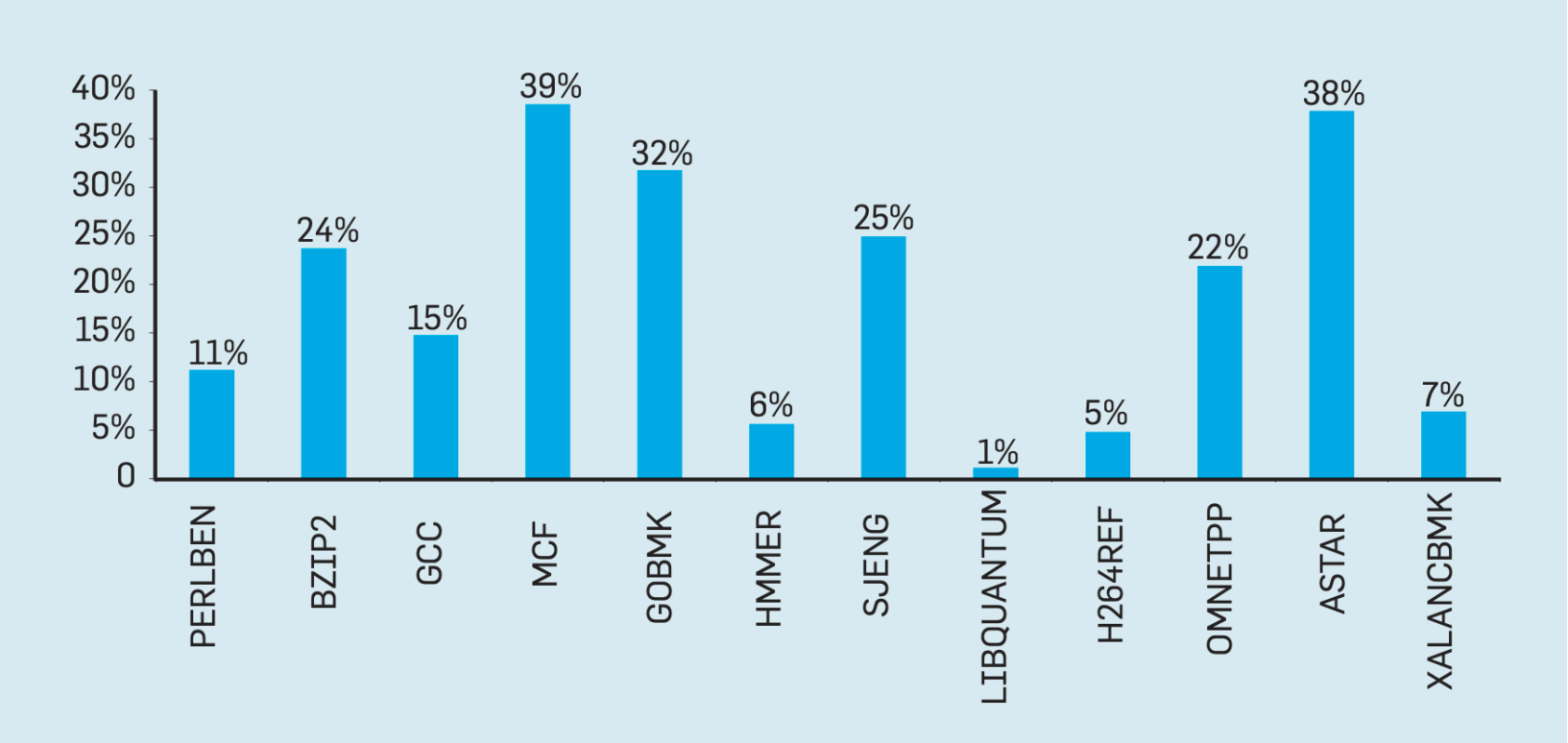
For example the Raptor Cove Intel® Xeon™ microarchitecture (2023) has 12 pipeline. stages



Energy per instruction Trends in Intel® Microprocessors, Ed Grochowski Murali Annavaram, 2006

Branch prediction

- Percentage of instructions wasted for SPEC integer benchmarks running on an Intel core i7. These are wasted due to incorrect branch predictions.



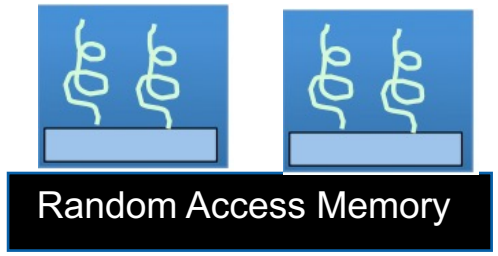
- On average, 19% of the instructions are wasted for these benchmarks on an Intel Core i7. The amount of wasted energy is greater, however, since the processor must use additional energy to restore the state when it speculates incorrectly.

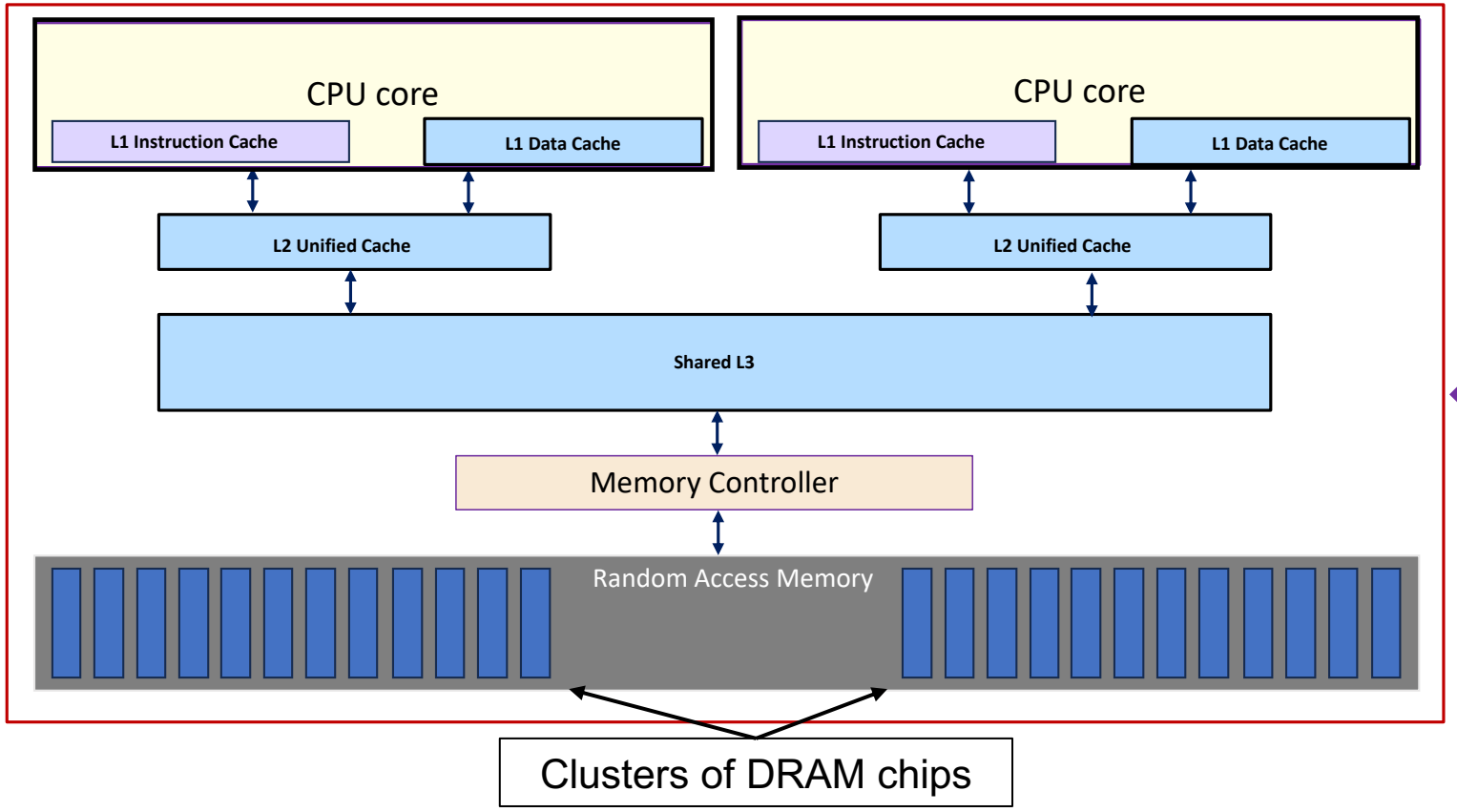
Image from "A New Golden Age for Computer Architecture", John Hennessy and David Patterson, CACM, 2019

Enough about the processors, what about the memory hierarchy

the Memory Hierarchy

We like to draw pictures like this



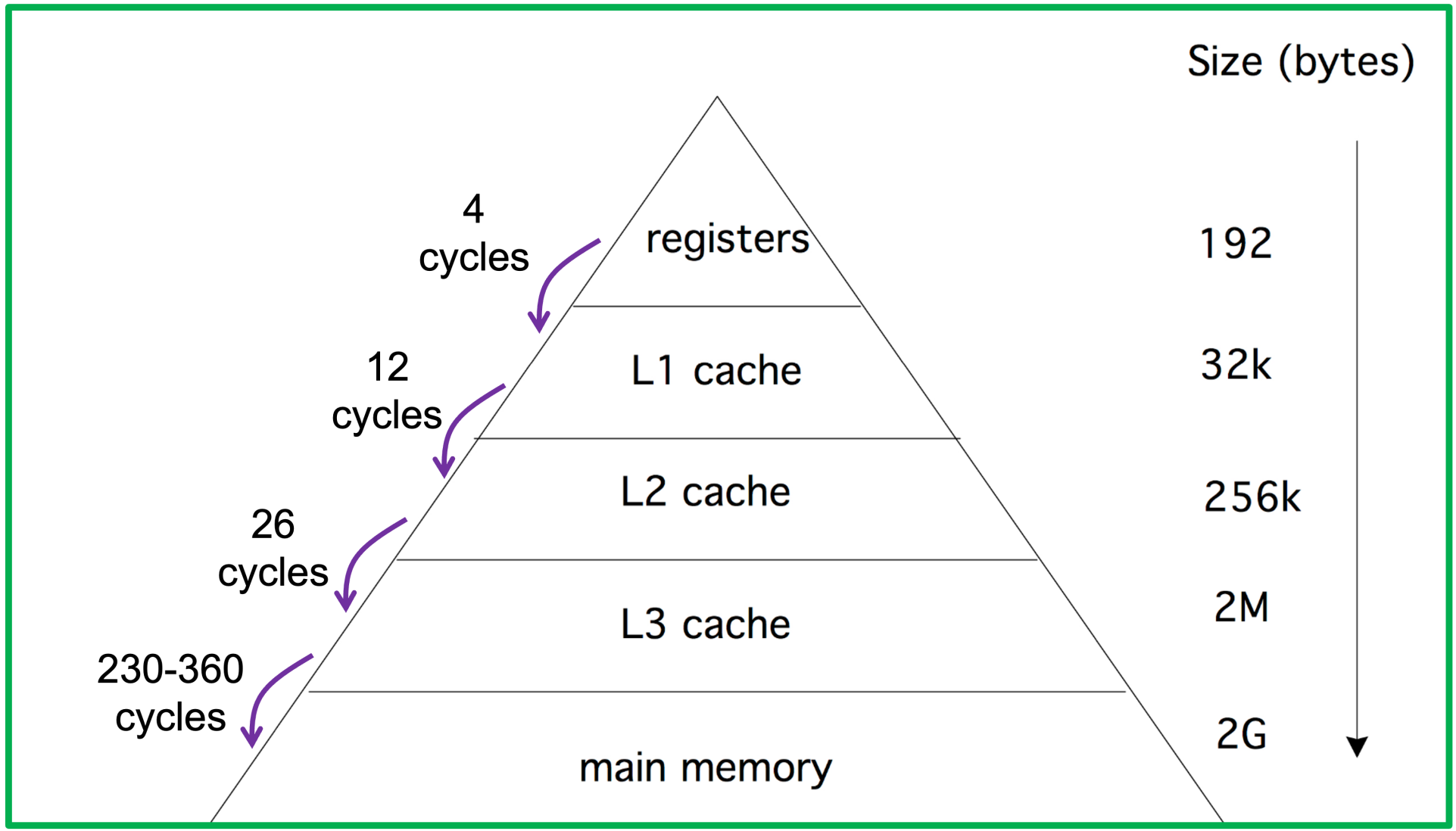


But reality is much more complex

DRAM: Dynamic Random Access Memory

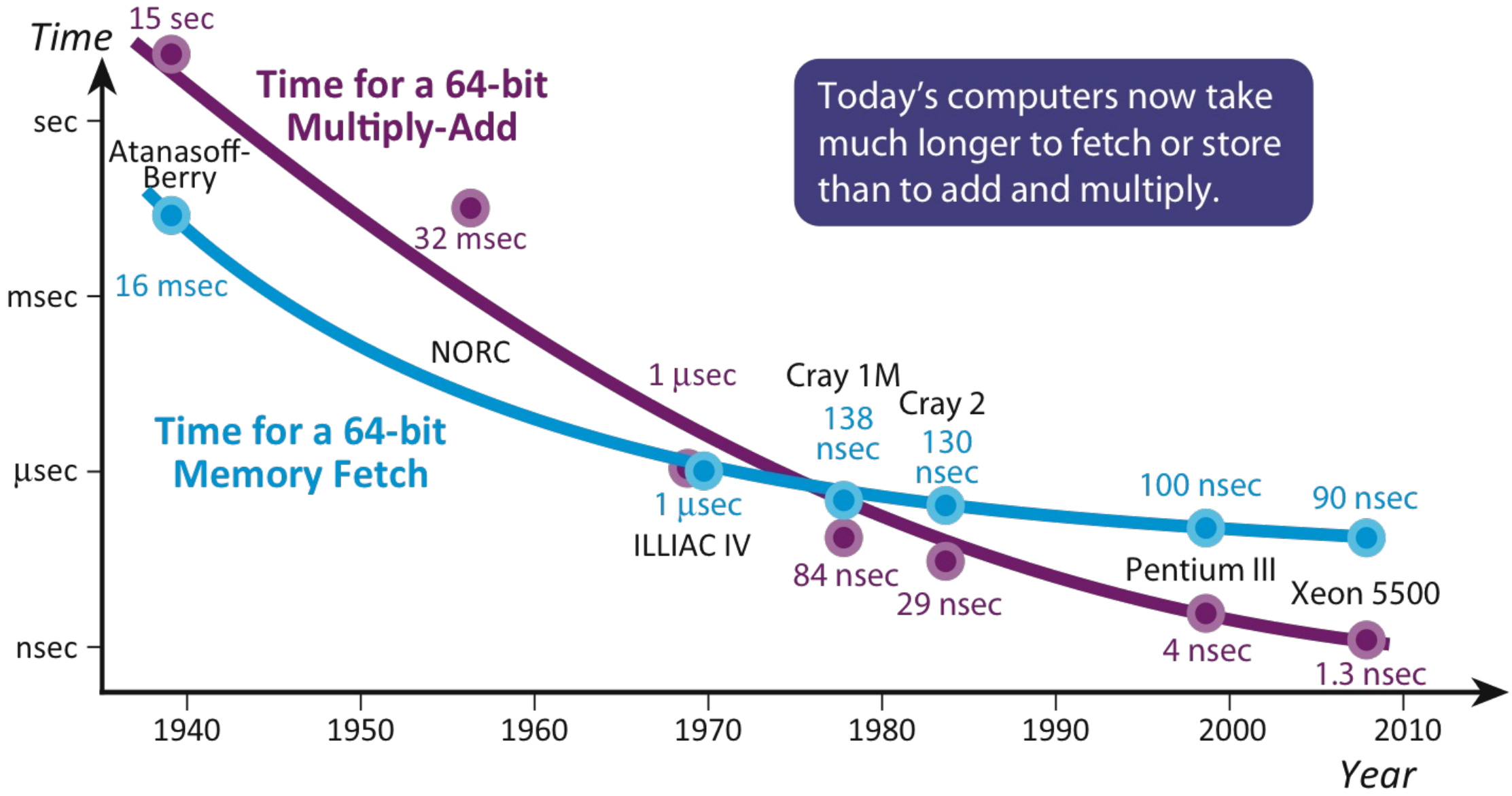
L1, L2, L3 pronounced Level 1, Level 2, Level 3. Also written sometimes as $L1, $L2, $L3

Latencies across the Memory Hierarchy



Source: Victor Eijkhout – Processor Architecture – Fall'2023

Latencies across the Memory Hierarchy



Memory access occurs as Cache Lines

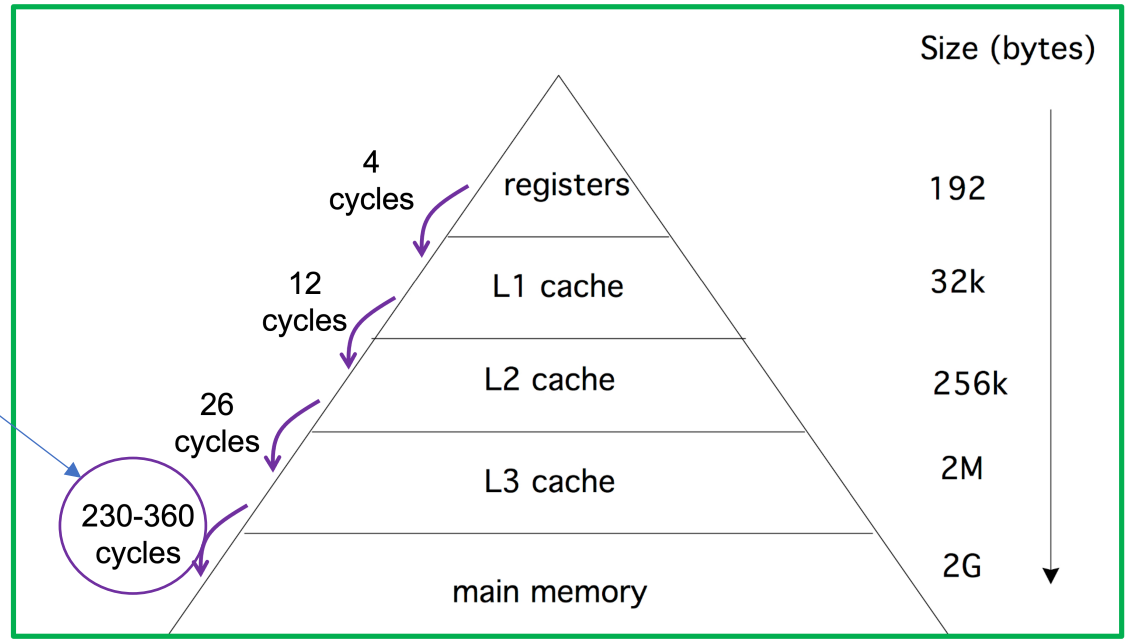
- Load/Store a data element to/from memory as a block of bytes in the level 1 cache ($L1). This is called a ***Cache line***.
- The size of an L1 cache line is a property of the architecture, but it is generally **64** bytes aligned in memory at 64 Bytes.
- Example, load element 17 of an array of doubles. You get the following cache line in the L1 data cache

A 32 KB $L1 holds 512 cache lines

Recall the Latencies across the memory hierarchy

Fetching a cache line all the way from memory costs hundreds of cycles.

Therefore, make sure you do as much as you can with any cache line you access



Use as much of a cache line as you can per access

- If you only use the one item you requested, you waste a huge amount of bandwidth to memory.

- **Spatial locality**: use all the elements in a line before fetching the next line.
  - Good Spatial locality … C/C++ use a row-major layout for multidimensional arrays, make the last index change the fastest.
  - Its easier to see what's happening if we represent the array through pointers (which is the most common case in C anyway).

```
float matrix[Nrows][Ncols];
for (int i = 0; i<Nrows; i++)
    for (int j= 0; j<Ncols; j++)
        matrix[i][j] += increment;
```

```
float *matrix;
matrix = (float*) malloc( Nrows*Ncols*sizeof(float);
for (int i = 0; i<Nrows; i++)
    for (int j= 0; j<Ncols; j++)
        *matrix(i*Nrows+j) += increment;
```

In most modern CPUs, the hardware will try to hide latency by prefetching cache lines before you need them.

  - Terrible Spatial locality … traverse a linked list (pointer chasing). Generate lots of "cache thrashing".

```
while (p != NULL) {
        DoWork(p);
        p = p->next;
}
```

- **Temporal locality**: Try to complete "all" work with data while you know its in the cache.

Use as much of a cache line as you can per access

- If you only use the one item you requested, you waste a huge amount of bandwidth to memory.

This is our first encounter with the C programming language

The variable **matrix** as an array (**matrix[Nrows][Ncols]**). Elements of the array are **matrix[i][j]**.

This code has pair of nested loops. The loops have loop-control indices (**i** and **j**) which run over a range of index values. For example, for the first loop we start at **i=0**, increment **i** by one at each iteration (**i++)** and keep going as long **i** < **Nrows**. So **i** goes from **0** to **(Nrows-1)**

```
float matrix[Nrows][Ncols];
for (int i = 0; i<Nrows; i++)
    for (int j= 0; j<Ncols; j++)
        matrix[i][j] += increment;
```

```
float *matrix;
matrix = (float*) malloc( Nrows*Ncols*sizeof(float);
for (int i = 0; i<Nrows; i++)
    for (int j= 0; j<Ncols; j++)
        *matrix(i*Nrows+j) += increment;
```

Experienced C programmers work with arrays through pointers to locations in memory and offsets. We allocate a block of space with a memory allocator (**malloc()**).

The array syntax **matrix[i][j]** is equivalent to the pointer syntax ***matrix(i*Nrows+j)**.

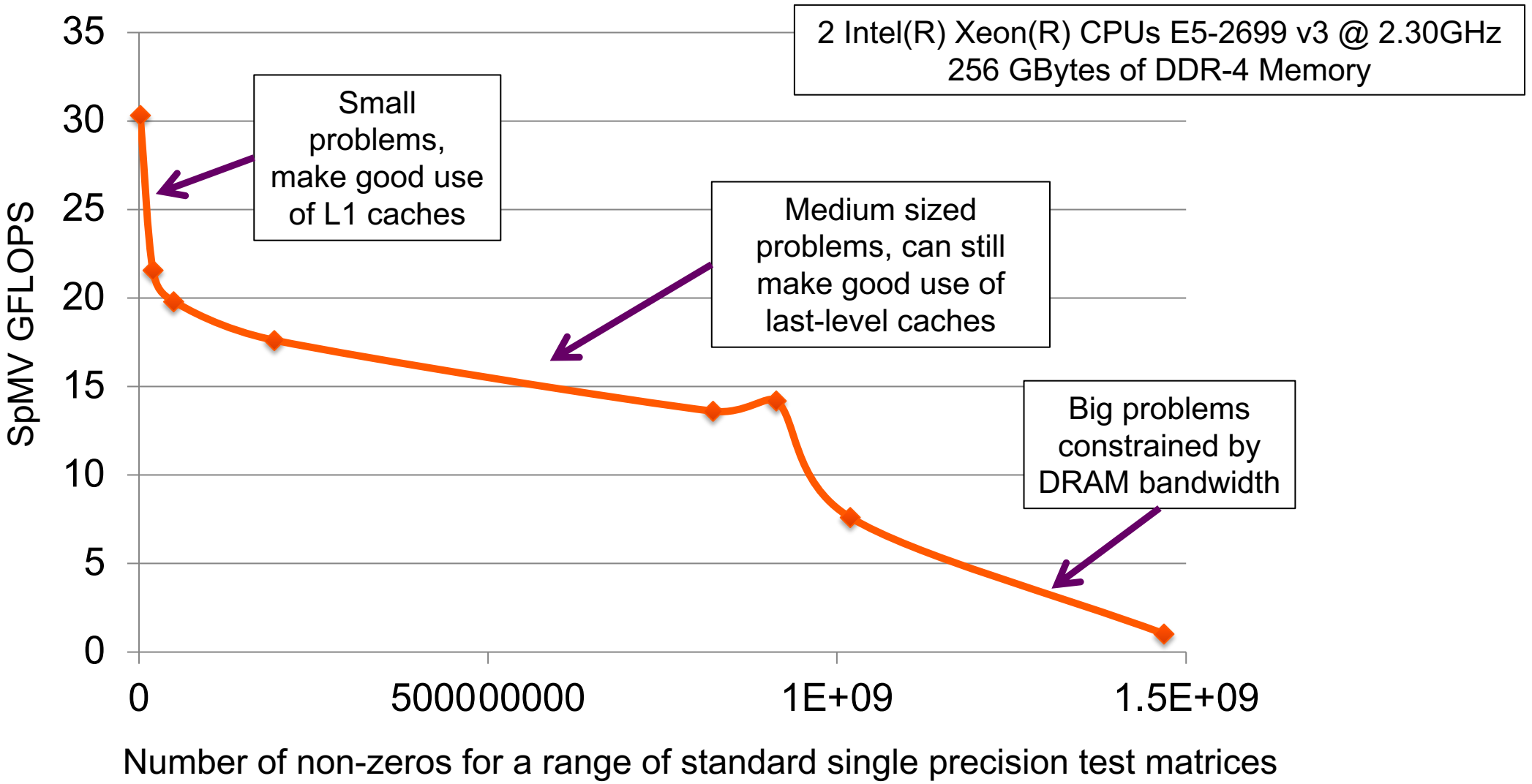
  - Terrible Spatial locality … traverse a linked list (pointer chasing). Generate lots of "cache thrashing

This is just another style of loop. The loop continues as long as the loop control condition (**p != NULL**) is true.

```
while (p != NULL) {
        DoWork(p);
        p = p->next;
}
```

- **Temporal locality**: Try to complete "all" work with data while you know its in the

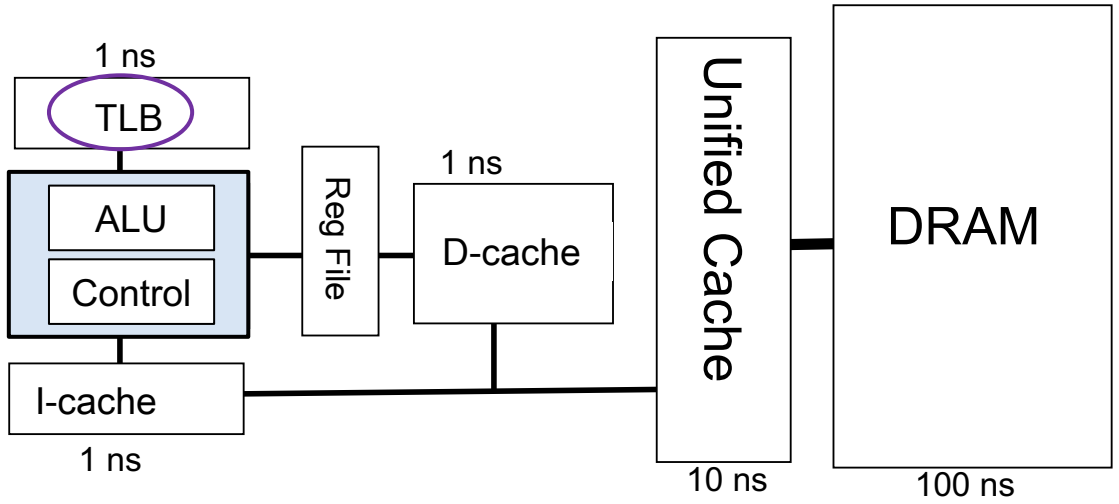
Sparse matrix vector multiplication (SpMV) is a good way to see the impact of caches across the memory hierarchy



Source: Intel® MKL™ 11.3, “mkl_cspblas_scsrgemv” with KMP_AFFINITY='verbose,granularity=fine,compact,1,0’

Memory Hierarchies

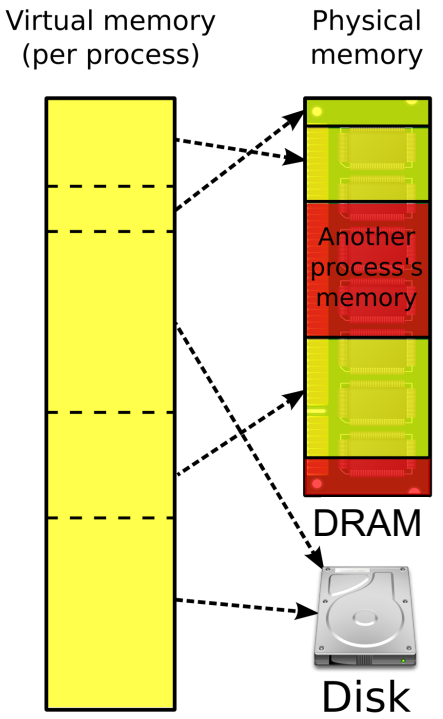
- A typical microprocessor memory hierarchy (with a simplified cache hierarchy)



- Consider applications with memory demands larger than the physical memory allocated to a process.
- We use virtual memory backed-up by the file system … bringing in pages of physical memory … to handle such problems
- The **TLB (Translation Lookaside Buffer)** implements virtual memory and caches the mapping from virtual addresses to physical addresses
  - If the entry is in the TLB page table, a small overhead of reading the entry and computed the physical address is incurred.
  - If the entry is not in the page table, a page fault exception will be raised. Requires expensive OS operations and stalls the CPU.

Just a bit of OS theory …

- A **process** is an instance of an executing program.
- An OS manages resources (such as threads and memory) for an executing program in terms of the process



OS: Operating system

Consider the “simple” Matrix Transpose

- Copy the transpose of A into a second matrix B.

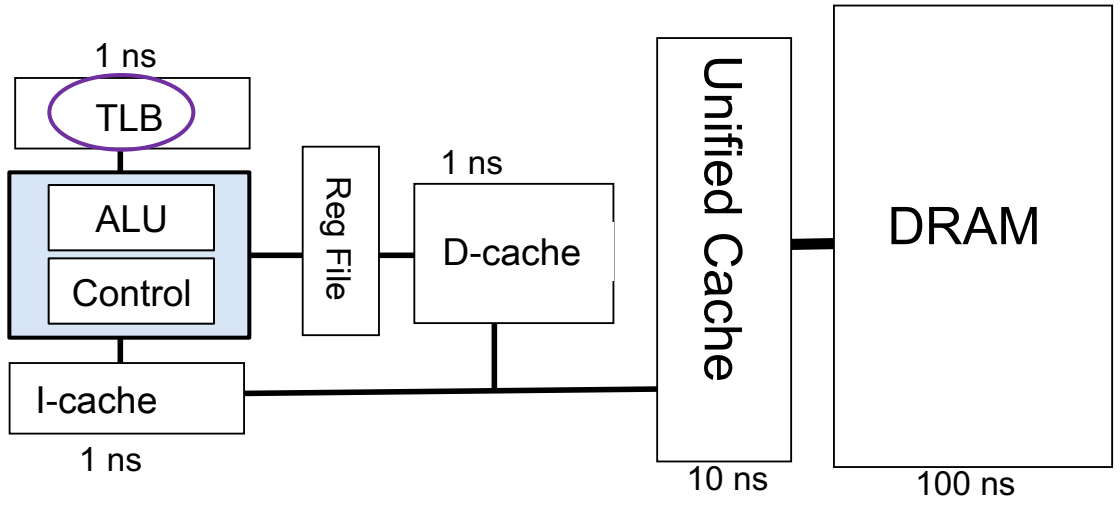
$A \in R^{NxN}$ $B \in R^{NxN}$

```
for (i=0;i<N; i++) {
    for (j=0;j<N;j++) {
        B[i+N*j] = A[j+N*i];
    }
}
```

column j of B

Row i of A

- Consider this operation and how it interacts with the TLB (Translation Lookaside Buffer).



For large N, as you march across addresses of A and B, you span multiple pages in memory … causes multiple page faults

Optimizing Matrix Transpose for the TLB

- Solution … break the loops into blocks so we reduce the number of page faults

```
for (i=0; i<N; i+=tile_size) {
    for (it=i; it<MIN(N,i+tile_size); it++){
  for (j=0; j<N; j+=tile_size) {
    for (jt=j; jt<MIN(N,j+tile_size);jt++){
            B[it+N*jt] = A[jt+N*it];
        }
      }
   }
}
```

Step 1: split the “i” and “j” loops into two … loop over tiles of a fixed size

Optimizing Matrix Transpose for the TLB

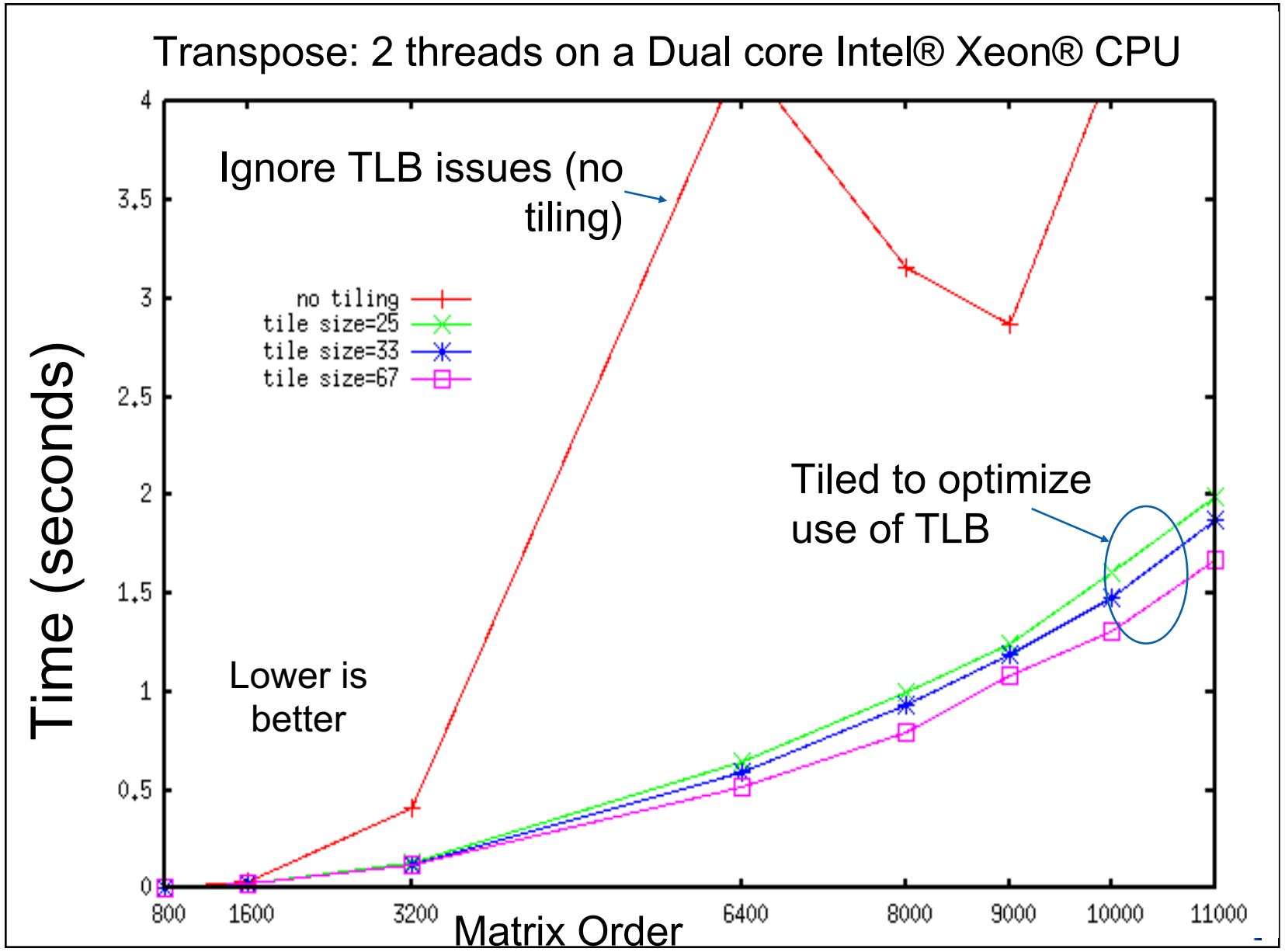
- Solution … break the loops into blocks so we reduce the number of page faults

```
for (i=0; i<N; i+=tile_size) {
  for (j=0; j<N; j+=tile_size) {
      for (it=i; it<MIN(N,i+tile_size); it++){
          for (jt=j; jt<MIN(N,j+tile_size);jt++){
              B[it+N*jt] = A[jt+N*it];
          }
      }
  }
}
```

Step 2: rearrange the loops. Move the loops over the tiles to the innermost loop-nest

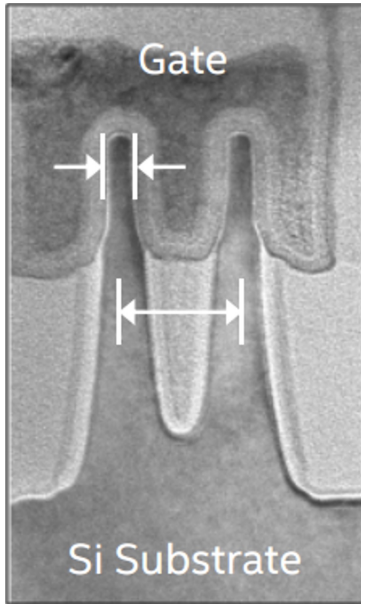
The result … you grab a tile, transpose that tile, then go to the next tile

Do you need to worry about the TLB?

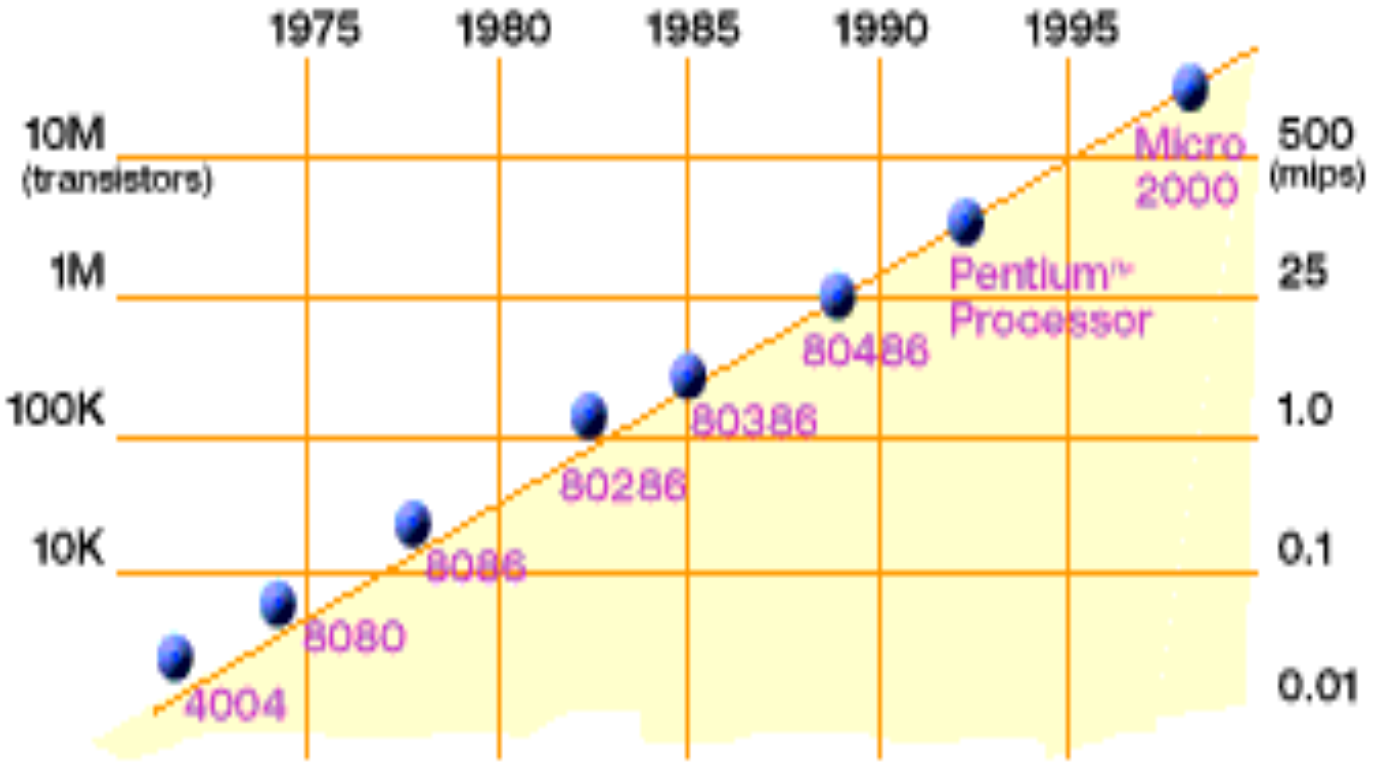


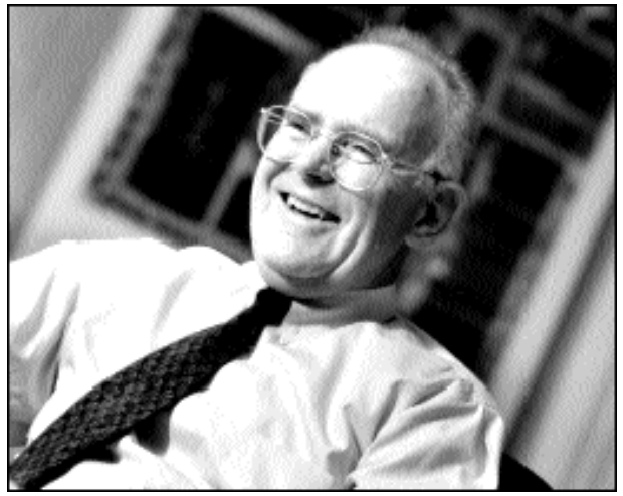
Source: M Frumkin, R. van de Wijngaart, T. G. Mattson, Intel

At last we get to the “final” topic in our discussion of computer architecture … hardware



Moore's Law





- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ***He was right!*** Over the last 50 years, transistor densities have increased as he predicted.

"Cramming more components onto integrated circuits", G.E. Moore, Electronics, 38(8), April 1965

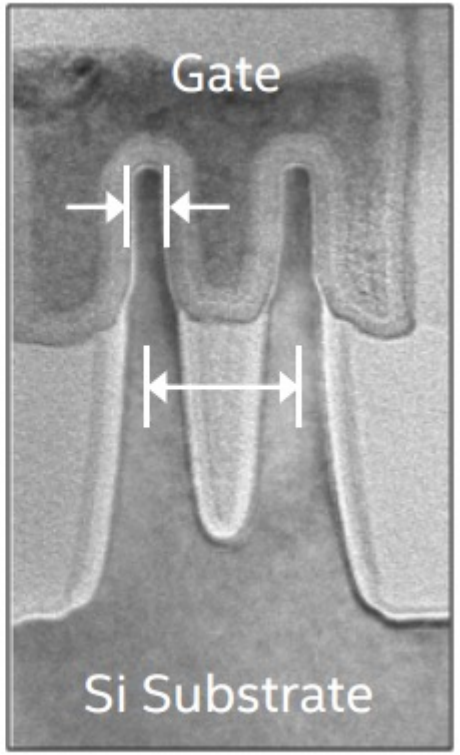
Slide source: UCB CS 194 Fall'2010

We've come a long way since Gordon Moore proposed his famous law

An electron microscope image of a single Intel transistor using the 14 nm process

8 nm Fin Width

42 nm Fin Pitch



We put billions of these transistors on a single chip

How small is a nm (nanometer)? One nm is 10^{-9} meters. Light travels about one foot in 10^{-9} seconds.

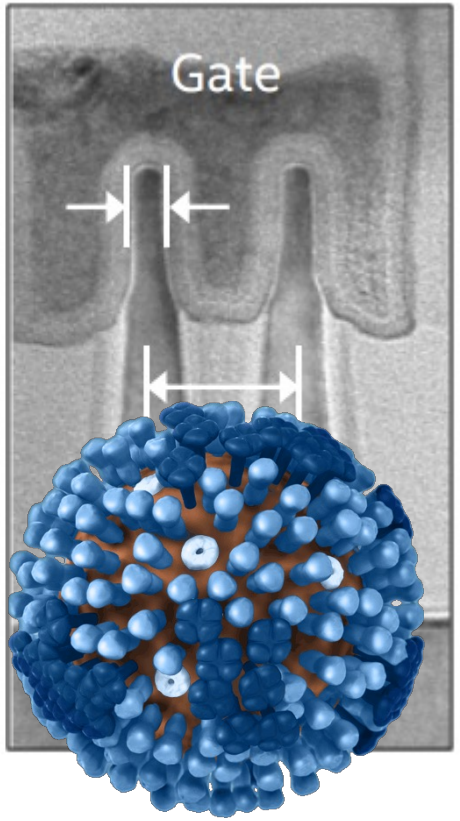
https://www-ssl.intel.com/content/dam/www/public/us/en/documents/technology-briefs/bohr-14nm-idf-2014-brief.pdf

We've come a long way since Gordon Moore proposed his famous law

An electron microscope image of a single Intel transistor using the 14 nm process

8 nm Fin Width

42 nm Fin Pitch



We put billions of these transistors on a single chip

An influenza virus is around 100 nm across!

http://www.cdc.gov/flu/images/h1n1/3D_Influenza/3D_Influenza_transparent_no_key_full_med2.gif

https://www-ssl.intel.com/content/dam/www/public/us/en/documents/technology-briefs/bohr-14nm-idf-2014-brief.pdf

How did those itty-bitty transistors impact performance?

The Linpack benchmark over time

Cray 1, 1977, first real supercomputer, 110 MFLOPS (Linpack 1000)

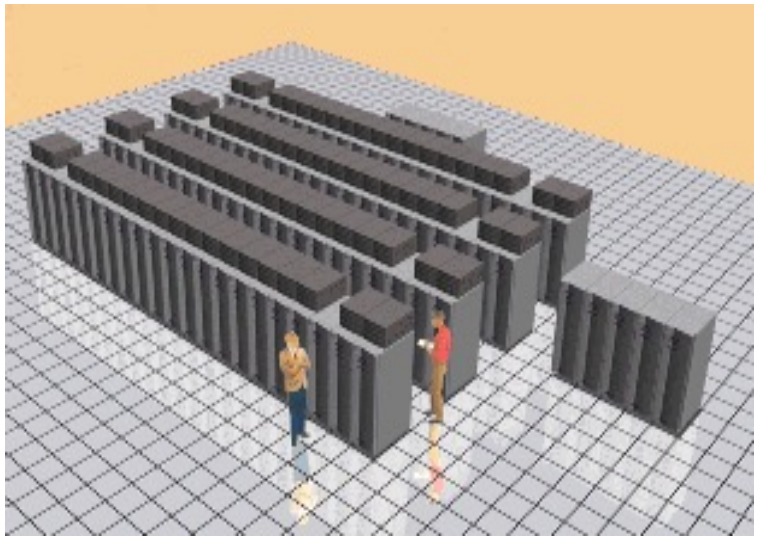
A MFLOPS is one million floating point operations per second (i.e. one million adds or multiplies per second).

Apple iPhone 6S, 2016, 1274 MFLOPS "in my pocket"

VAX 11/780, ~1980, 0.14 MFLOPS, this is the computer I used in my Ph.D. research in the early 1980's.

Moore's Law: A personal perspective

First TeraScale* computer: 1997



Intel's ASCI Option Red

Intel's ASCI Red Supercomputer

9000 CPUs

one megawatt of electricity.

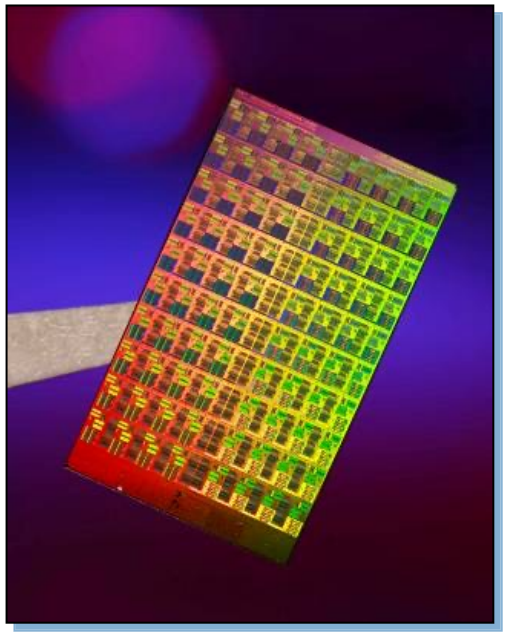
1600 square feet of floor space.

*Double Precision TFLOPS running MP-Linpack

A TeraFLOP in 1996: The ASCI TeraFLOP Supercomputer, Proceedings of the International Parallel Processing Symposium (1996), T.G. Mattson, D. Scott and S. Wheat.

10 years later

First TeraScale% chip: 2007



Intel's 80 core teraScale Chip

1 CPU

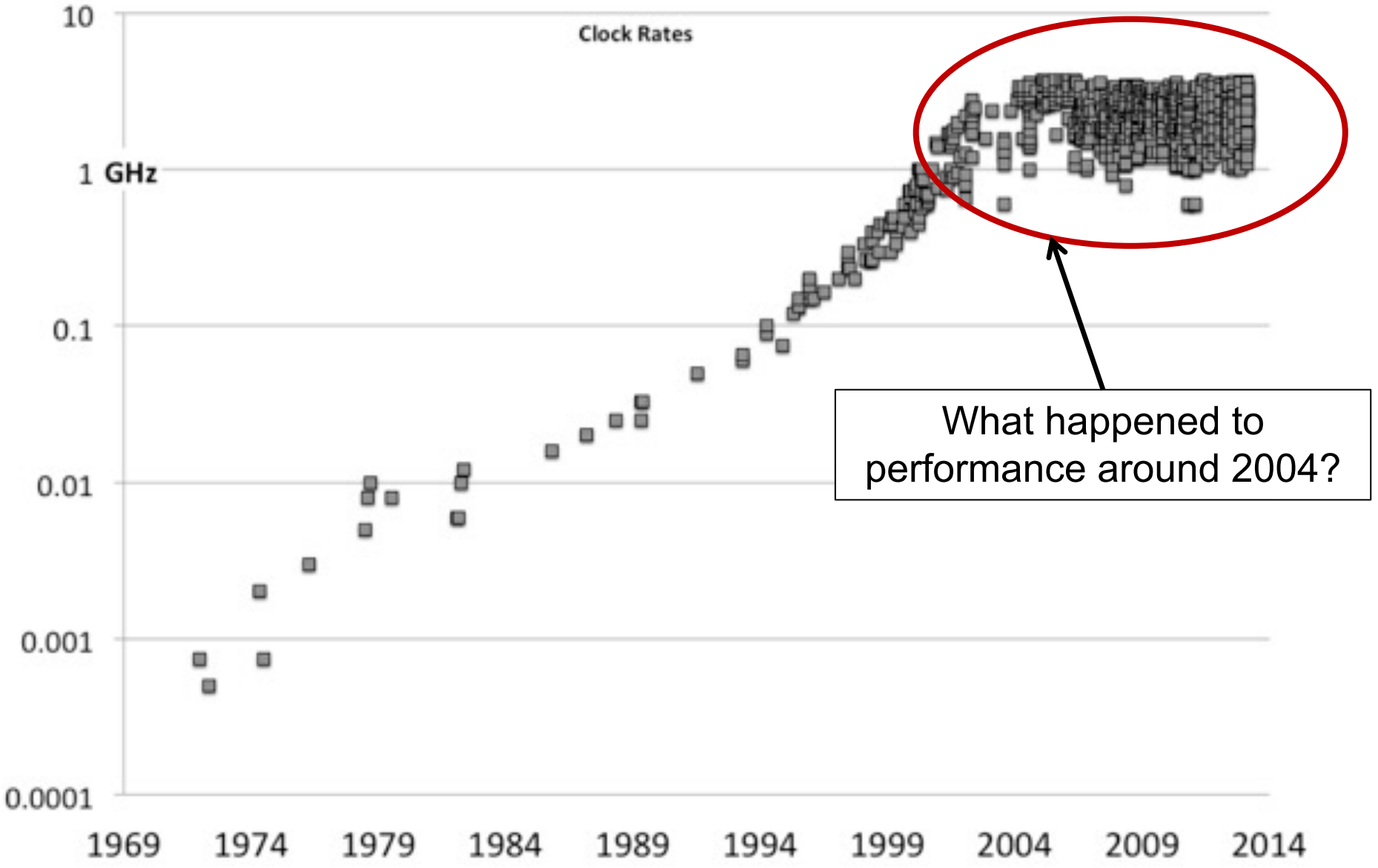
97 watt

275 mm2

%Single Precision TFLOPS running stencil

Programming Intel's 80 core terascale processor SC08, Austin Texas, Nov. 2008, Tim Mattson, Rob van der Wijngaart, Michael Frumkin

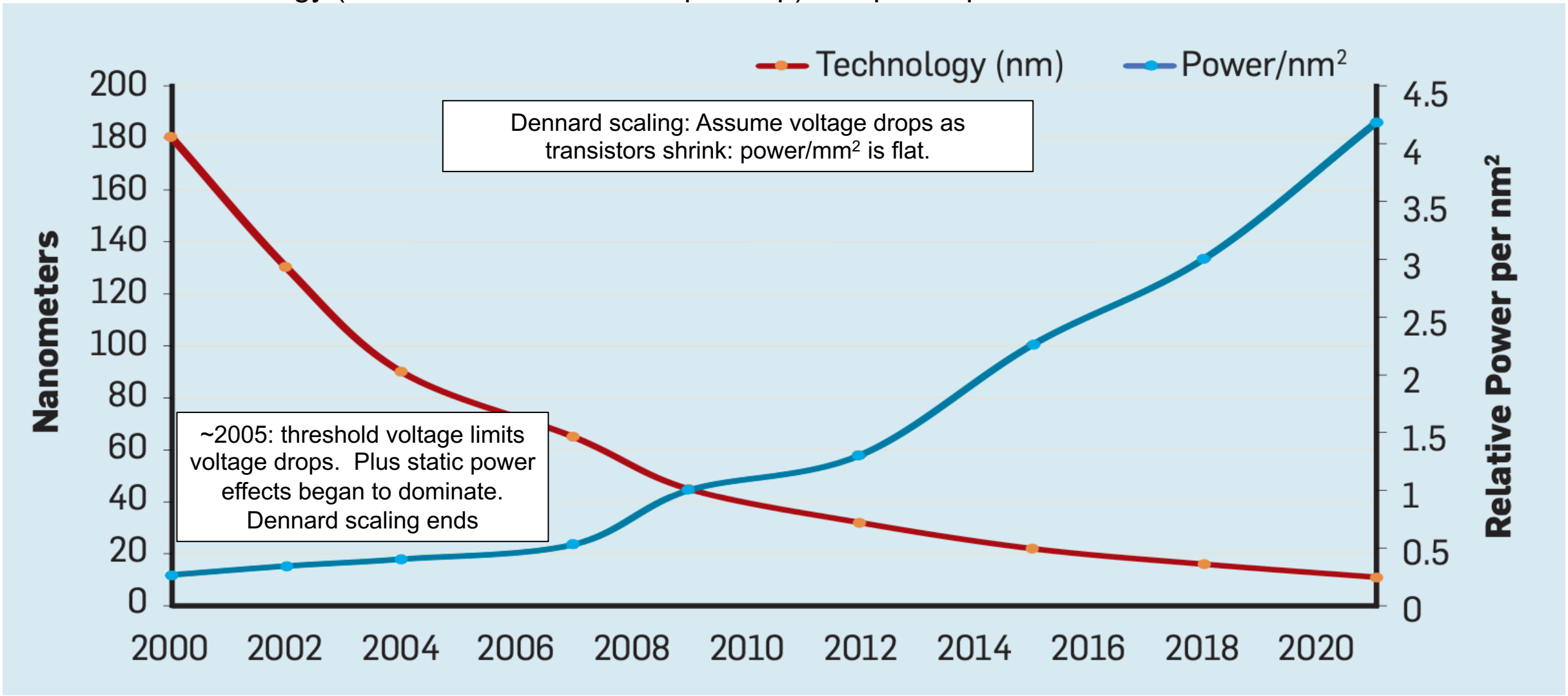
CPU Frequency (GHz) over time (years)



Source: James Reinders (from the book “structured parallel programming”)

Dennard Scaling

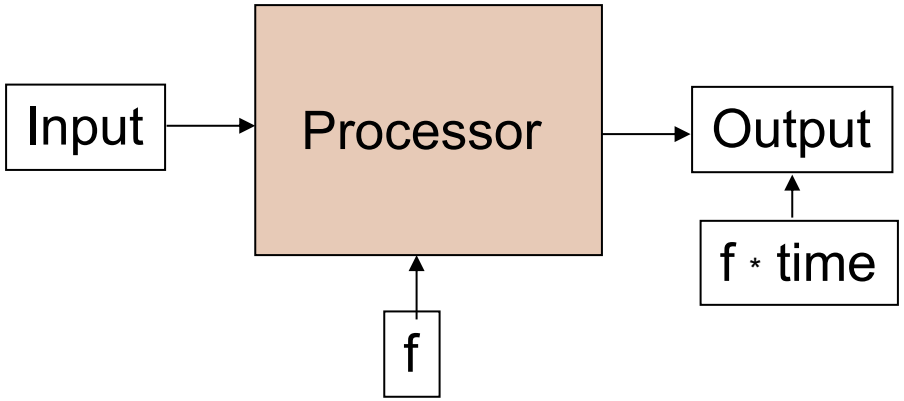
- Process technology (translates to Transistors per chip) and power per mm^2



Process technology nodes defined by the smallest feature on a chip (i.e. gate length in nm). After 22 nm, it's become a marketing term that doesn't map to a specific feature's length.

Image from "A New Golden Age for Computer Architecture", John Hennessy and David Patterson, CACM, 2019

Consider power in a chip …



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

C = capacitance … it measures the ability of a circuit to store energy:

$C = q/V \rightarrow \quad q = CV$

Work is pushing something (charge or q) across a "distance" … in electrostatic terms pushing q from 0 to V:

$V * q = W.$

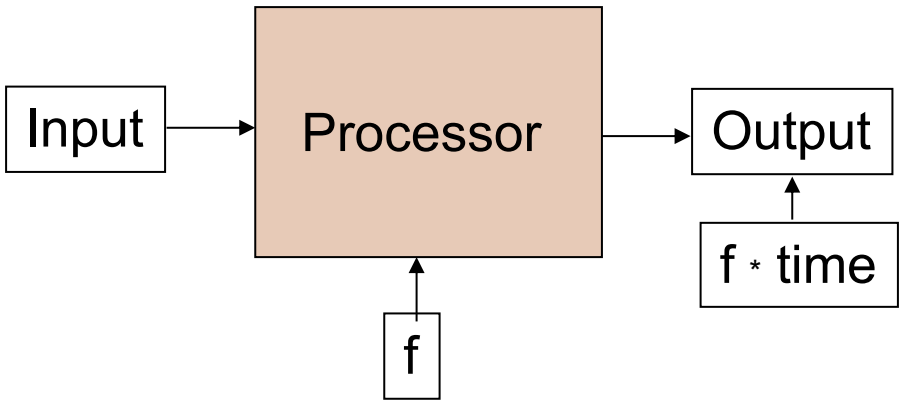
But for a circuit $q = CV$ so

$W = CV^2$

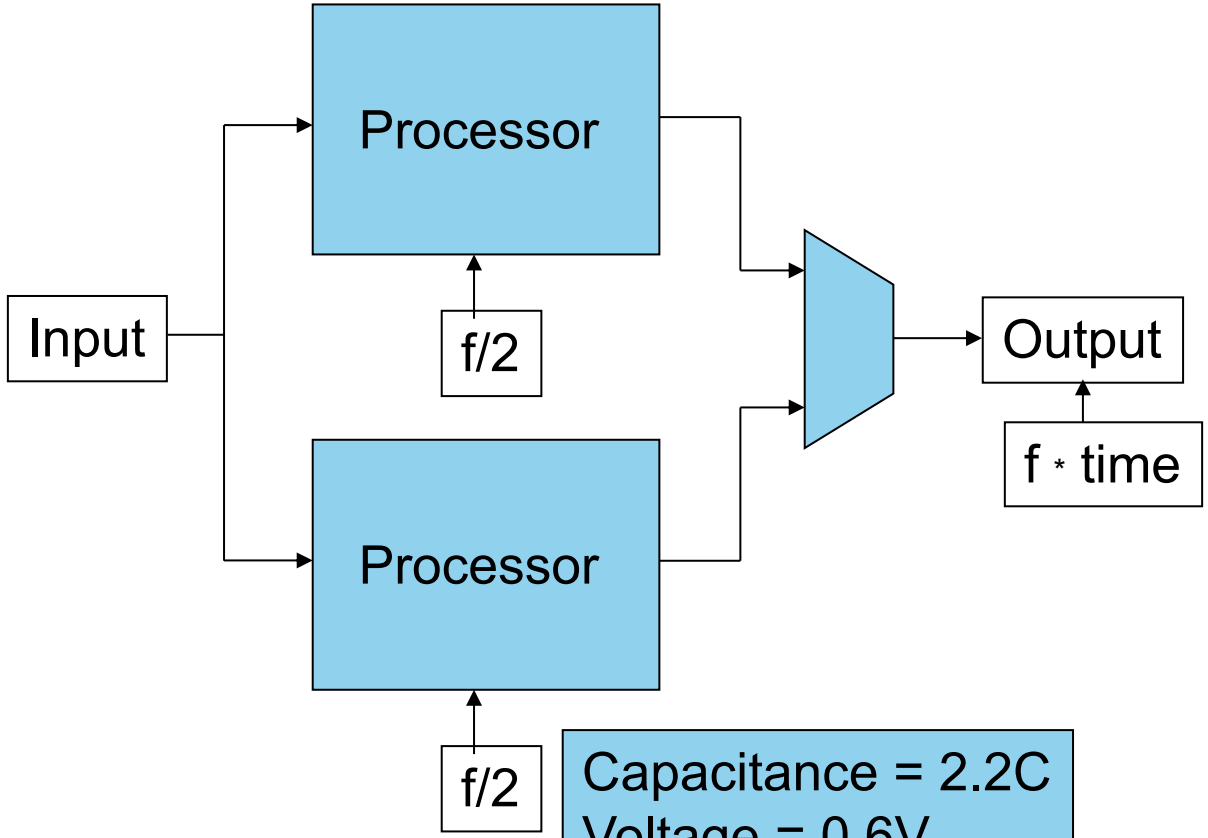
power is work over time … or how many times per second we oscillate the circuit

Power = W* F → Power = CV^2f

... Reduce power by adding cores



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

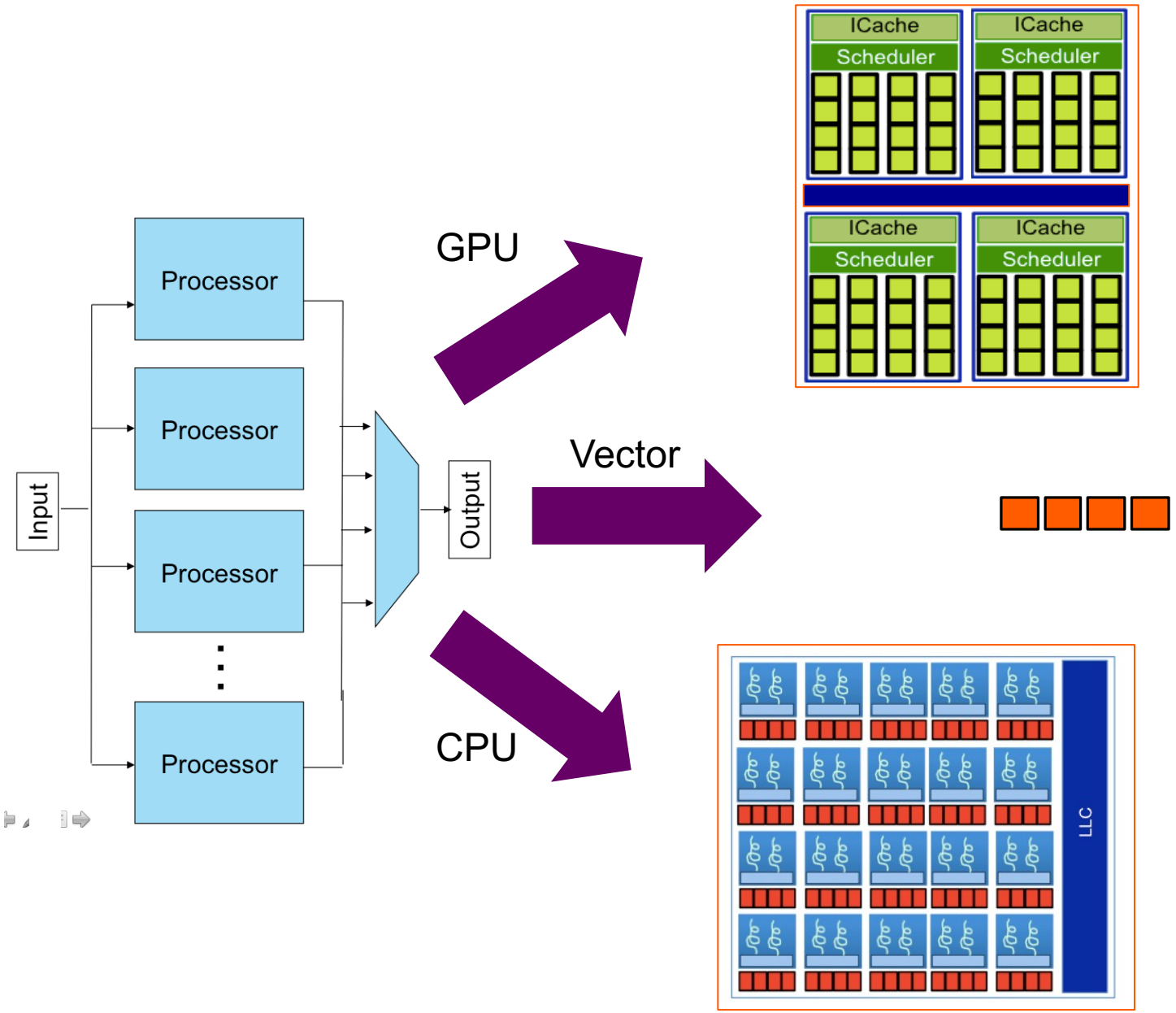


Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2f$

Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol.14, no.1, pp.12-31, Jan 1995

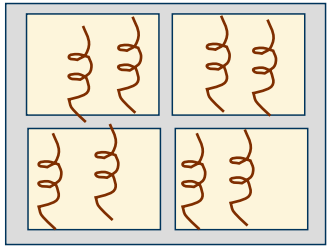
Source: Vishwani Agrawal

Manycore processors: three hardware options



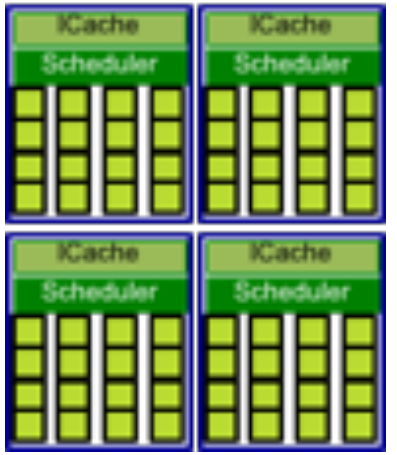
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

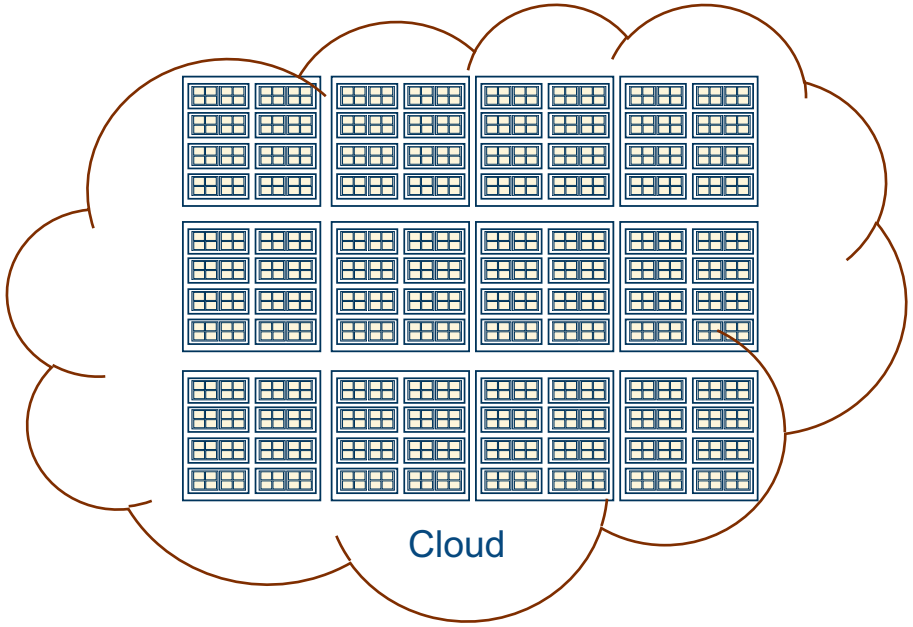


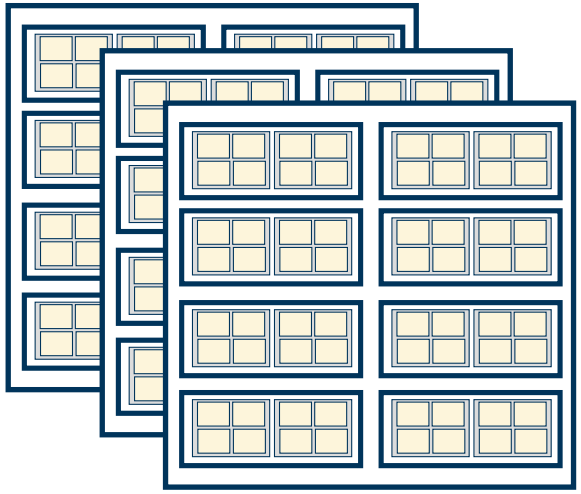
CPU

SIMD/Vector

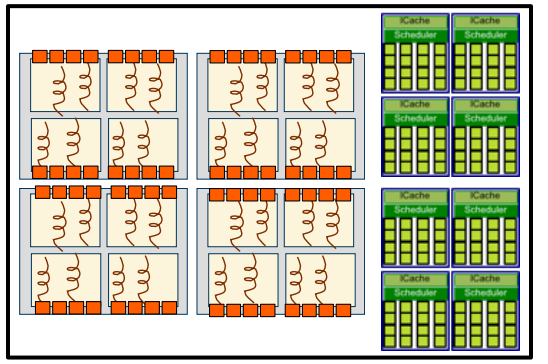


GPU





Cluster



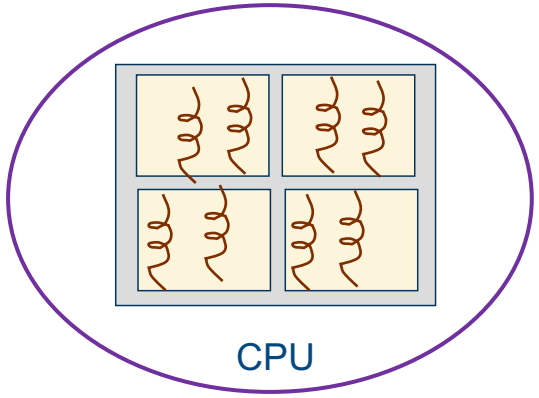
Heterogeneous node

Let’s quickly survey the key “on-node” parallel hardware approaches

By the term “on-node” I mean we are not going to discuss parallelism that comes from networking together large numbers of independent computer systems (as is done for cluster and cloud computing).

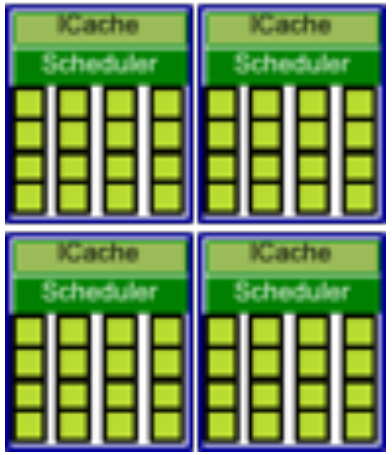
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

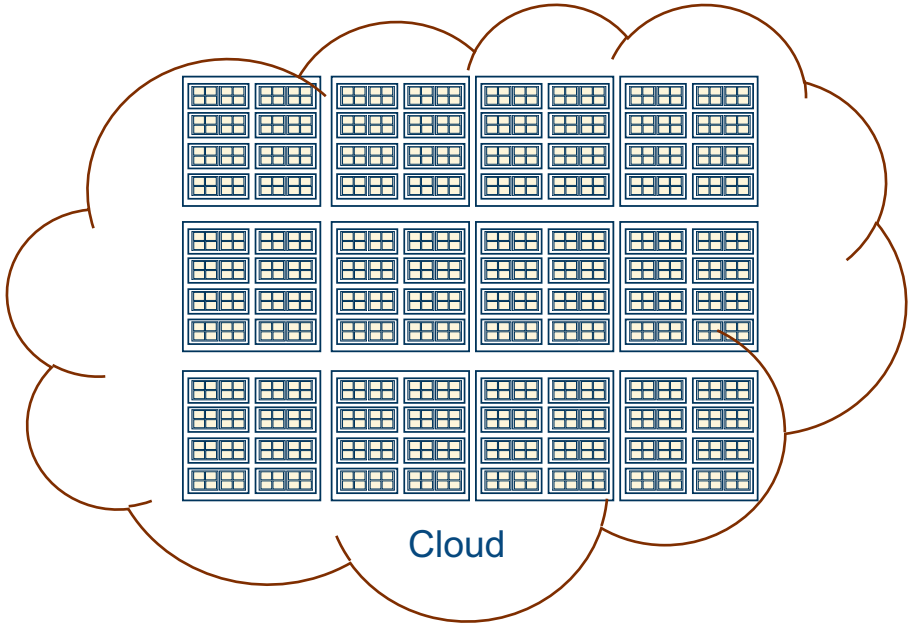


CPU

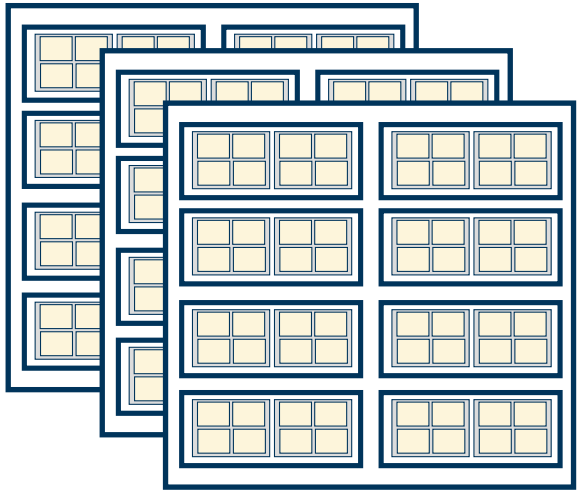
SIMD/Vector



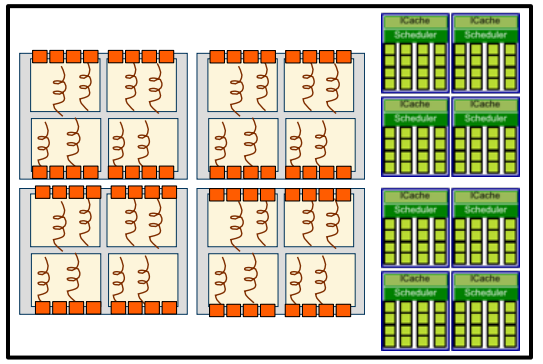
GPU



Cloud



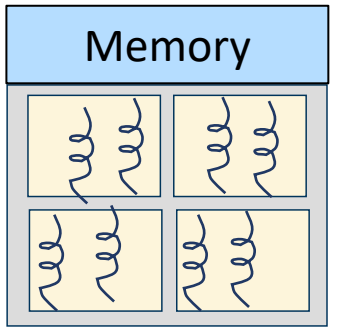
Cluster



Heterogeneous node

CPU parallelism: Multicore CPUs

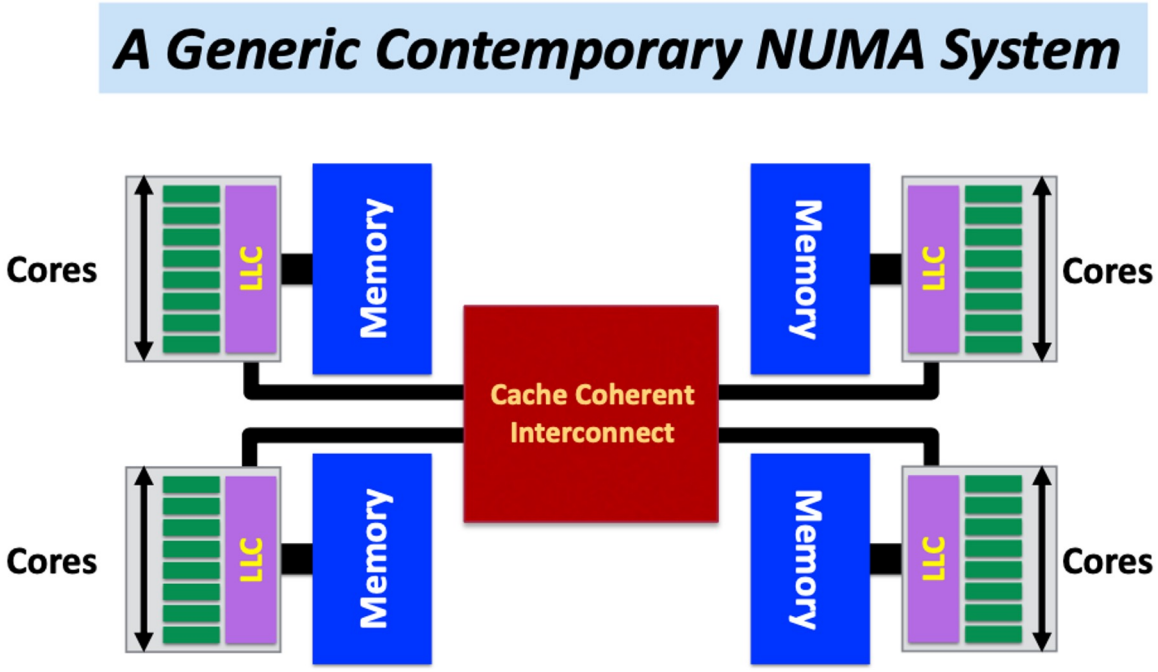
- A modern CPU optimized for performance will have multiple cores sharing a single memory hierarchy.
- The memory appears as a single address space.
- An instance of a program is a process.
- The process has a region of memory, system resources, and one or more threads.
- Parallelism is managed by the programmer as multiple threads mapped to the various cores.
- When every core is treated the same by the operating system (OS) and has an equal cost function to any location in memory, we call this a symmetric multiprocessor or SMP.



A four core CPU running a process with 8 threads mapped as 2 threads per core

CPU and Memory: a harsh does of reality

- Most systems today are Non-Uniform Memory Access (NUMA)
- Accessing memory in remote NUMA is slower than accessing memory in local NUMA
- Accessing High Bandwidth Memory is faster than DDR



Each core may have multiple hyperthreads

Also possible on-package high bandwidth memory on the node

Diagram courtesy Ruud van der Pas

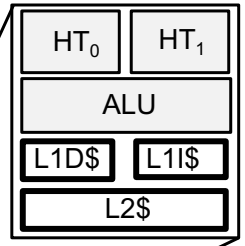
LLC: Last level cache

An even harsher does of reality when we look at an actual system

2 Intel® Xeon™ E5-2698 v3 CPUs (Haswell) per node (launched Q3'14)

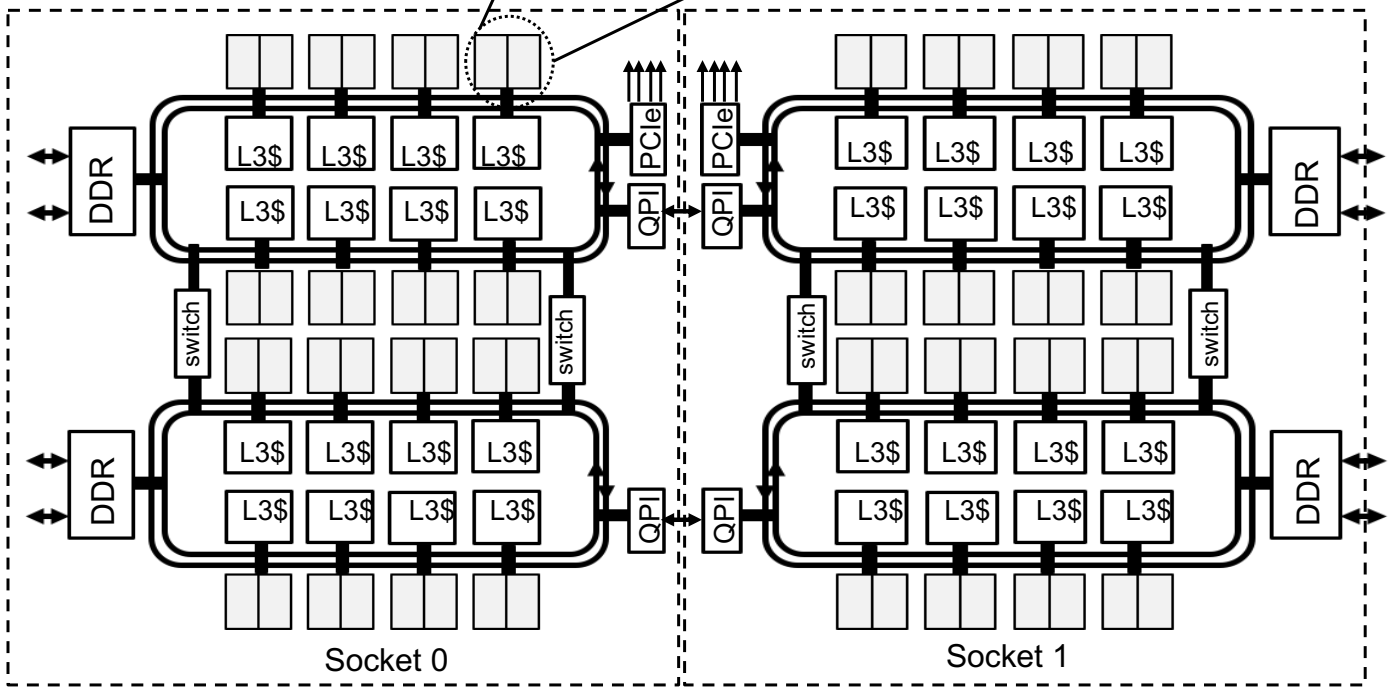
4 blocks of 8 core units connected by an on-chip-network with a DDR memory controller.

This constitutes a NUMA domain since memory access from a core to its "own" DDR is less expensive.



2 Hardware threads (HT) per core
Intel® AVX2 (256 bit Vector unit)
L1$ instruction and data: 32 KB
Unified L2$ 256 KB

40 MB shared L3$



DDR: Double Data Rate memory controller

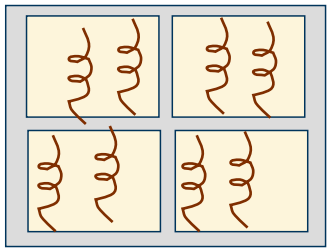
PCIe is the connection from the CPU to other devices in a node.

QPI: Quick Path Interconnect. A coherent interconnect between CPUs. Makes it easy to build multiple CPU nodes.

As configured for Cori at NERSC: CPUs at 2.3 GHz, 2 16 GB DIMMs per DDR memory controller, 16 cores per CPU. 2 CPUs connected by a high-speed interconnect (QPI)

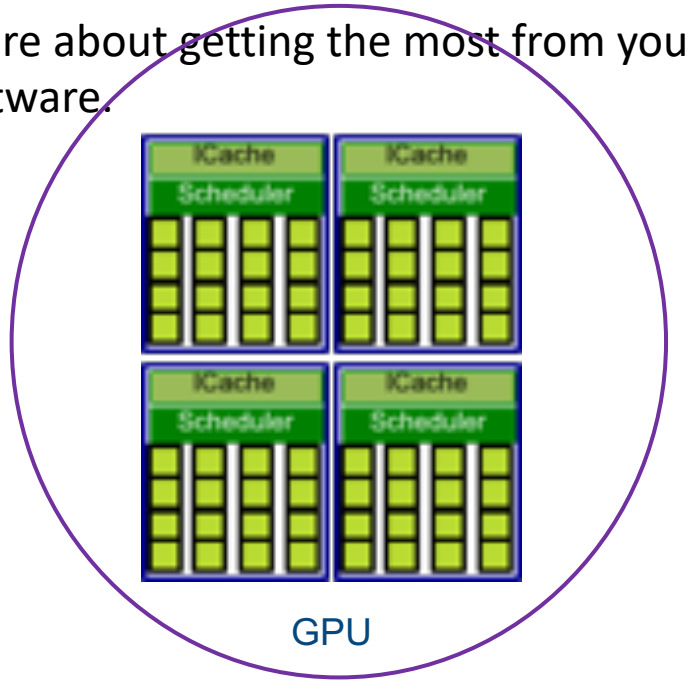
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

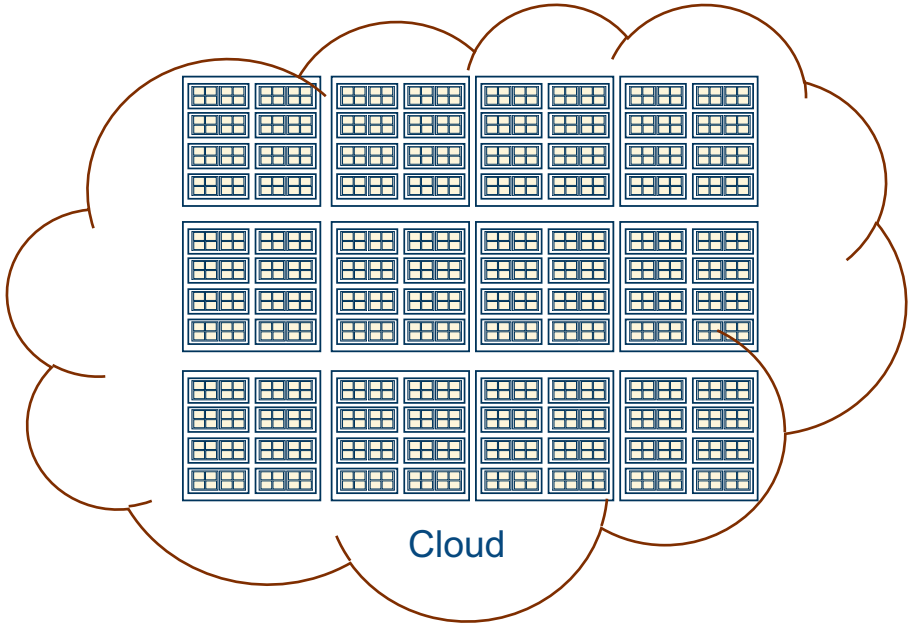


CPU

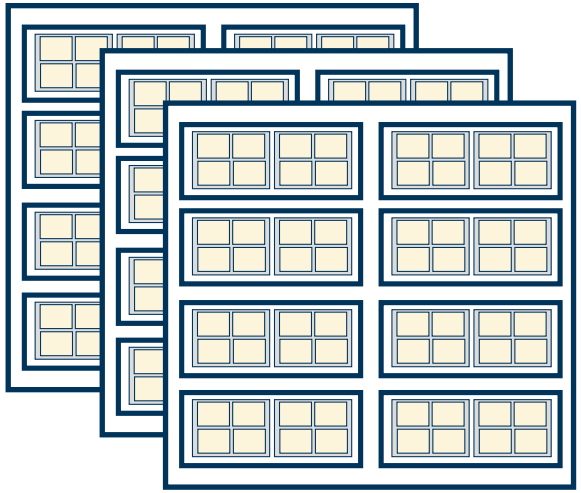
SIMD/Vector



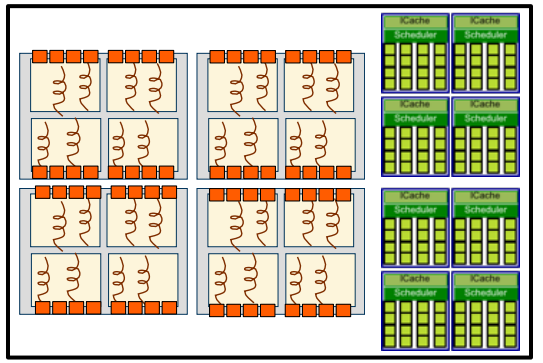
GPU



Cloud



Cluster

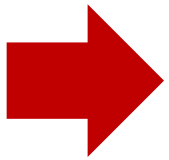


Heterogeneous node

The "BIG idea" Behind GPU programming

Traditional Loop based vector addition (vadd)

```
int main() {
   int N = . . . ;
   float *a, *b, *c;

   a* =(float *) malloc(N * sizeof(float));

   // ... allocate other arrays (b and c)
   // and fill with data

   for (int i=0;i<N; i++)
      c[i] = a[i] + b[i];

}
```



Data Parallel vadd with CUDA

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

Assume a GPU with unified shared memory … allocate on host, visible on device too

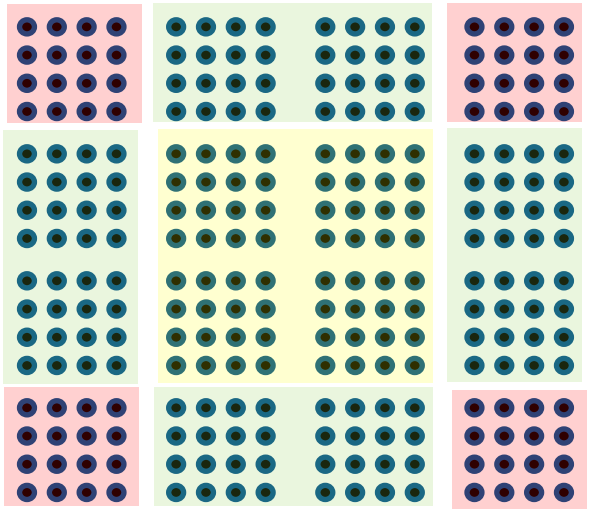
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn source code into a scalar work-item

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N)
{
    int i = blockIdx.x * blockDim.x +
threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c,
N);
}
```

This is CUDA code … the sort of code the OpenMP compiler generates on your behalf

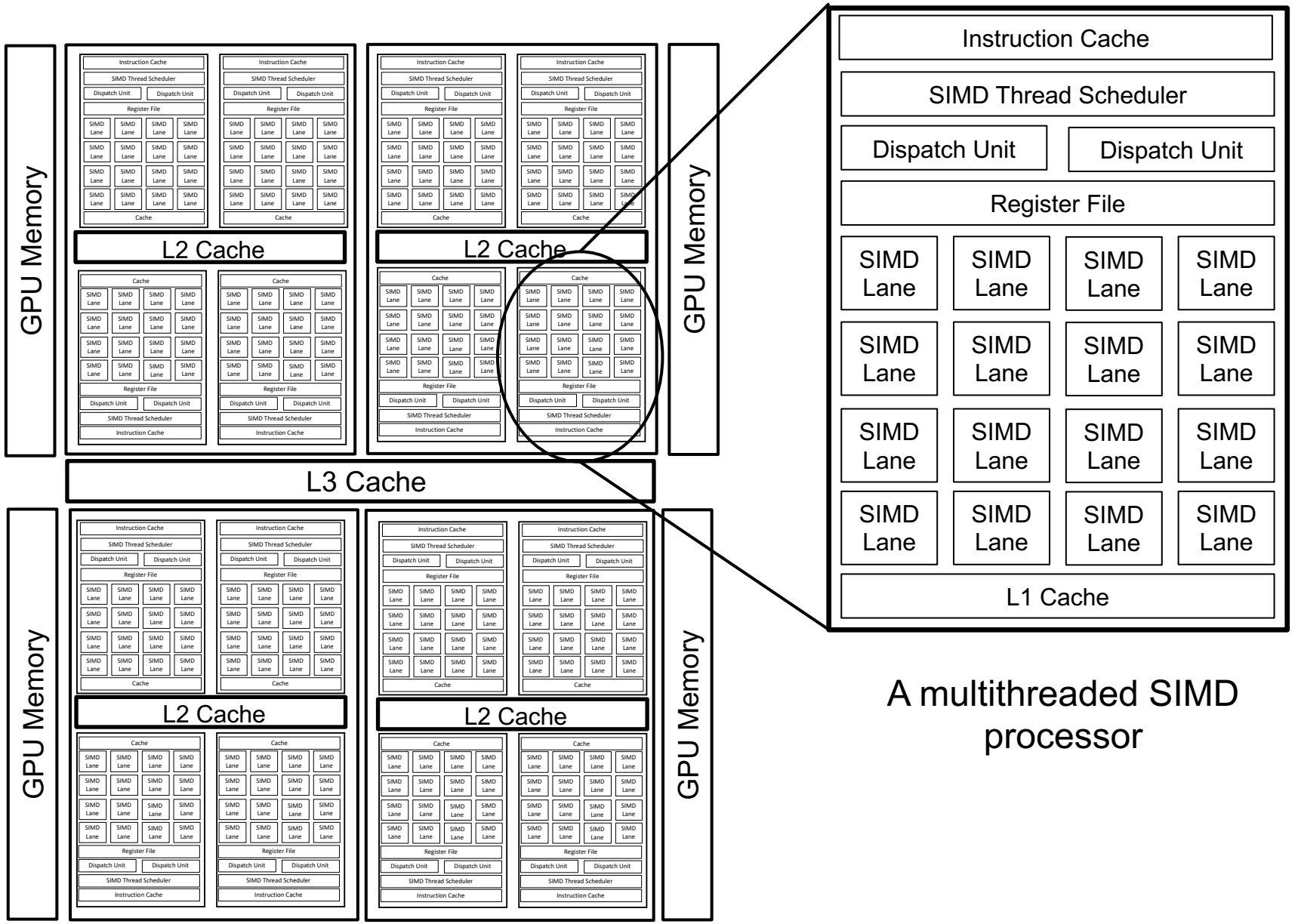
2. Map work-items onto an N dim index space.



3. Map data structures onto the same index space

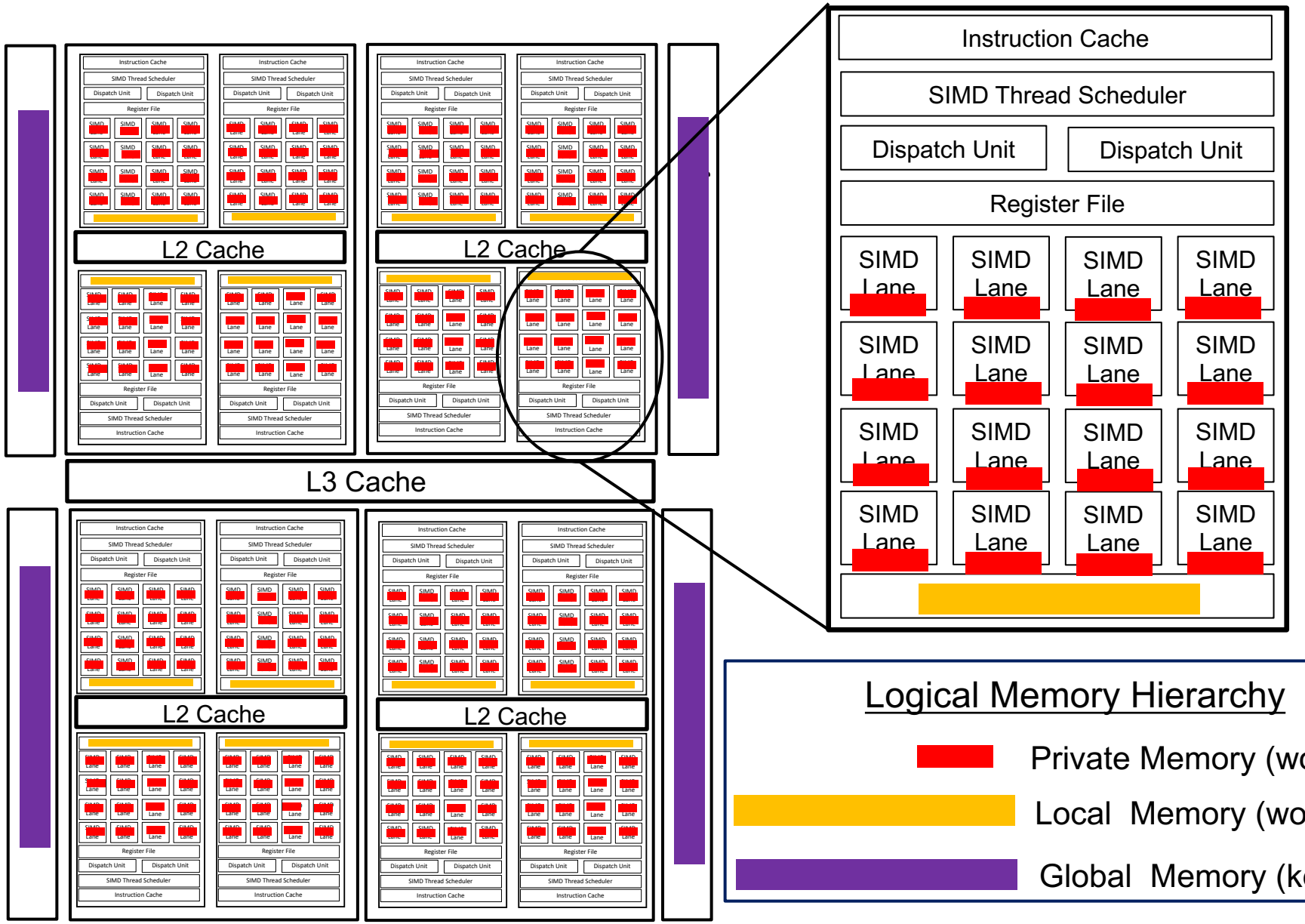
4. Run on hardware designed around the same SIMT execution model

A Generic GPU (following Hennessey and Patterson)



A multithreaded SIMD processor

A Generic GPU (following Hennessey and Patterson)

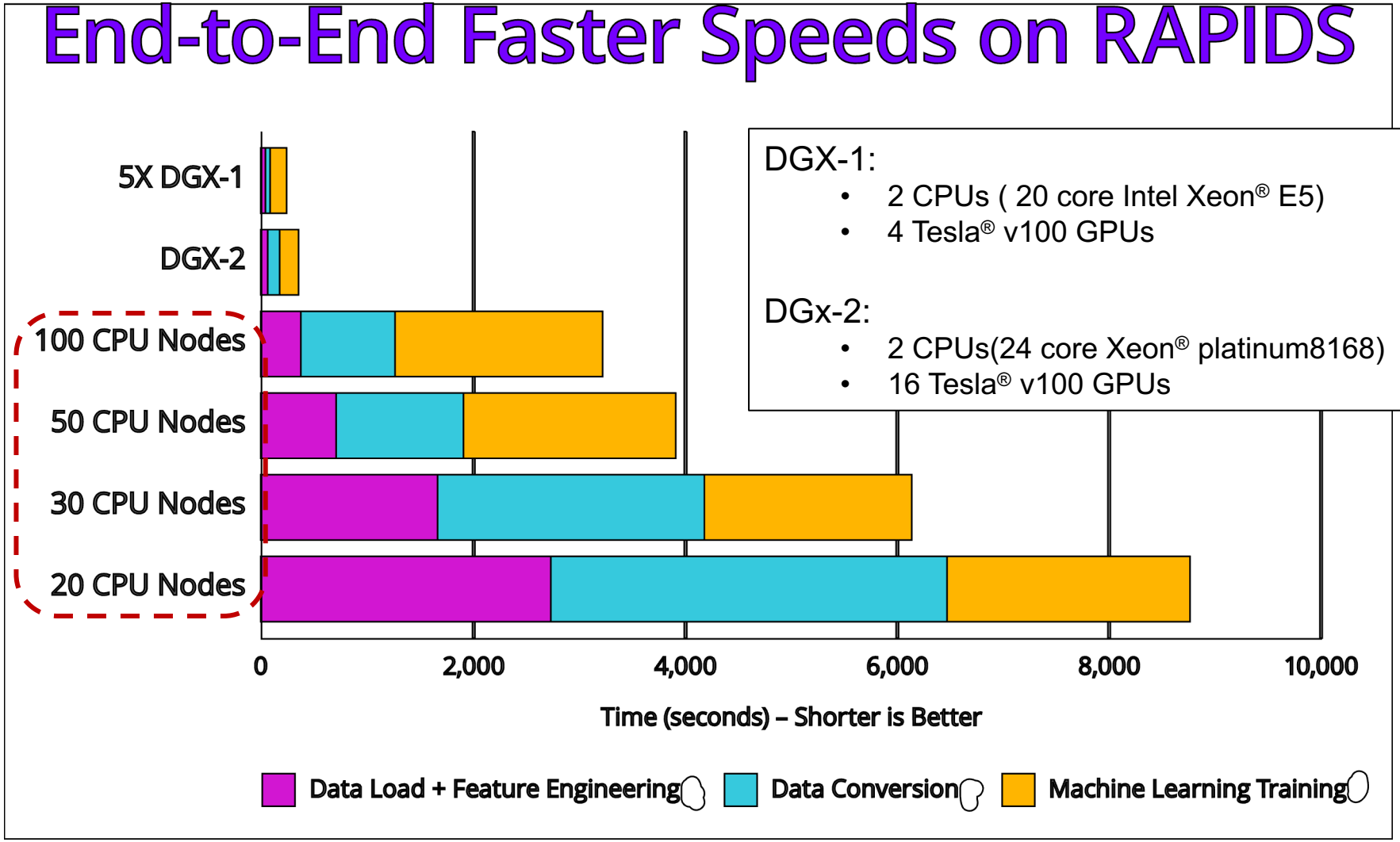


A brief aside …. Look out for dishonest GPU/CPU benchmarking

NVIDIA Performance claims

An Nvidia slide from CLSAC'18 talk

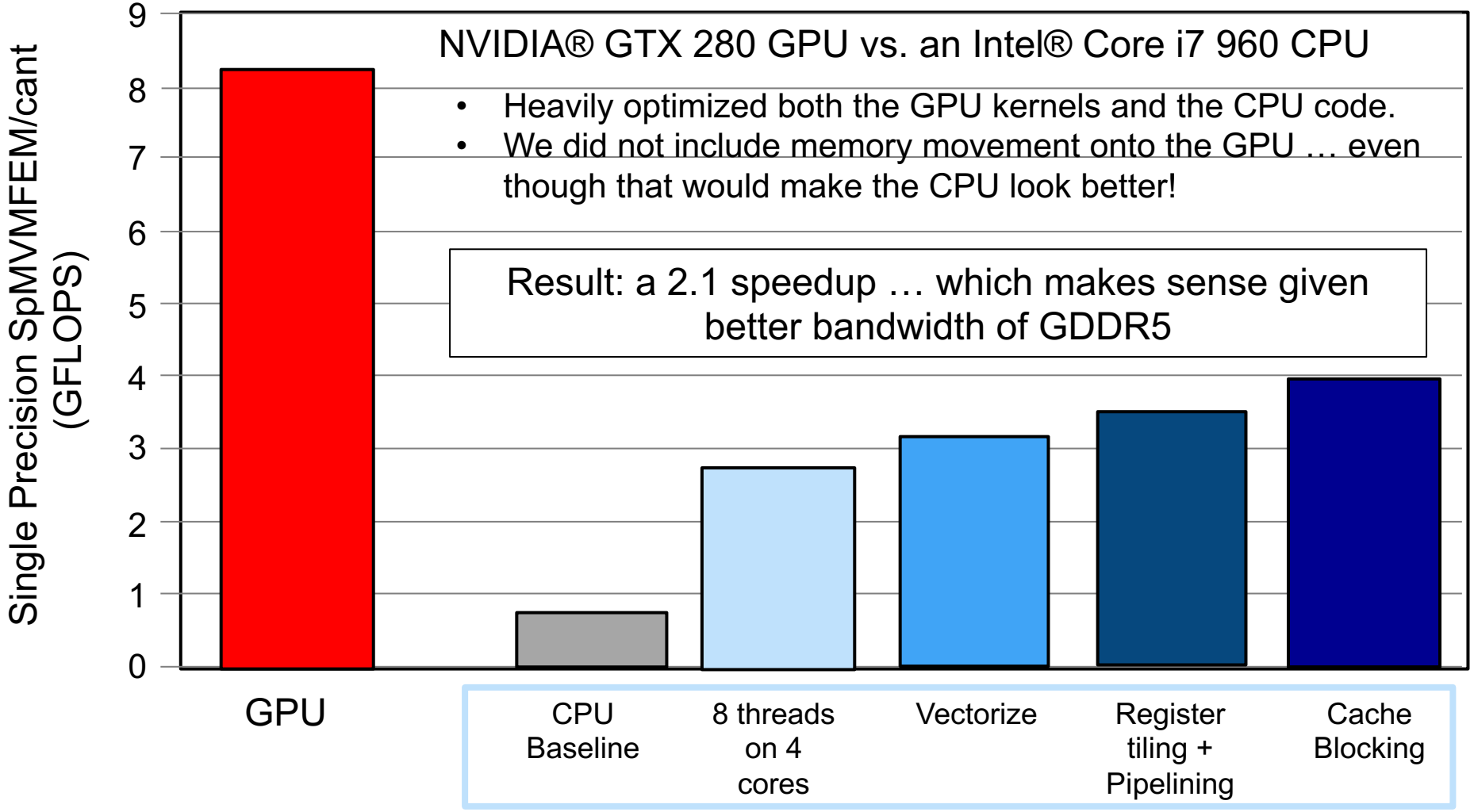
This is the most dishonest marketing I have EVER seen.



"CPU Node" = 1 AWS Broadwell Intel® Xeon® E5 hardware thread.

Sparse matrix vector product: GPU vs. CPU

- [Vazquez09]: reported a 51X speedup for an NVIDIA® GTX295 vs. a Core 2 Duo E8400 CPU … but they used an old CPU with unoptimized code



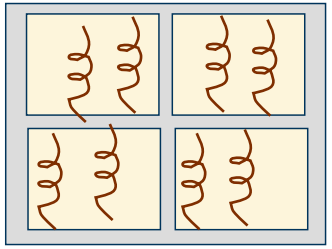
Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

… back to “on-node” parallel hardware

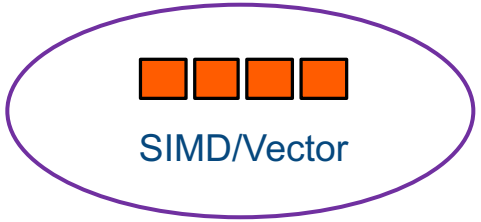
By the term “on-node” I mean we are not going to discuss parallelism that comes from networking together large numbers of independent computer systems (as is done for cluster and cloud computing).

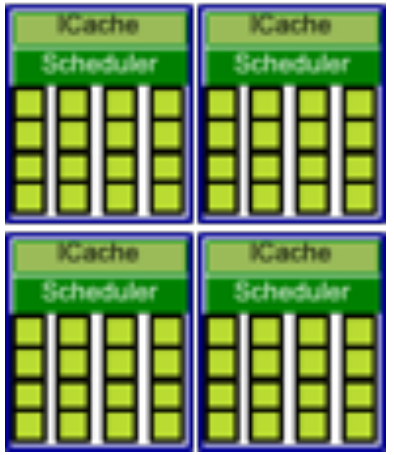
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

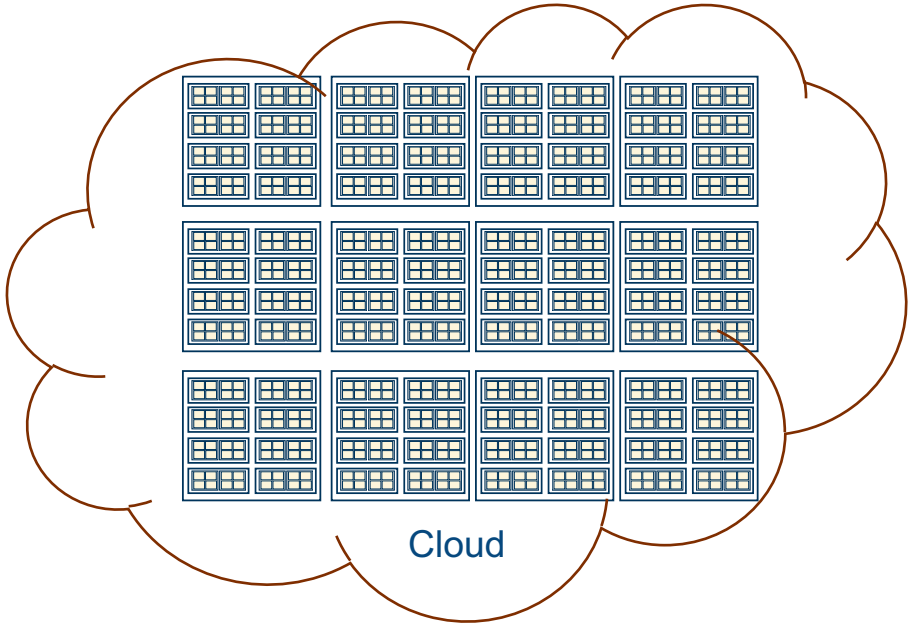


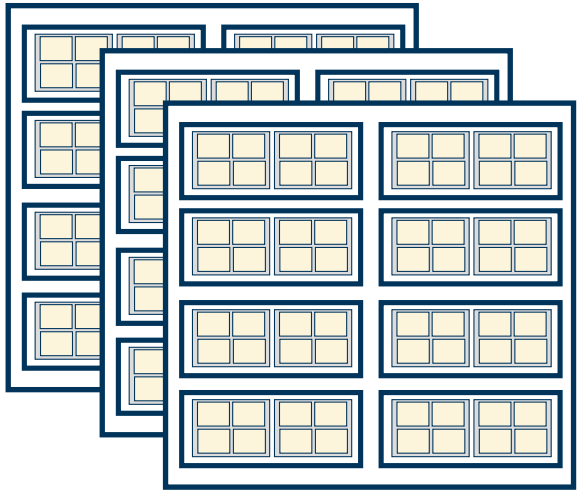
CPU



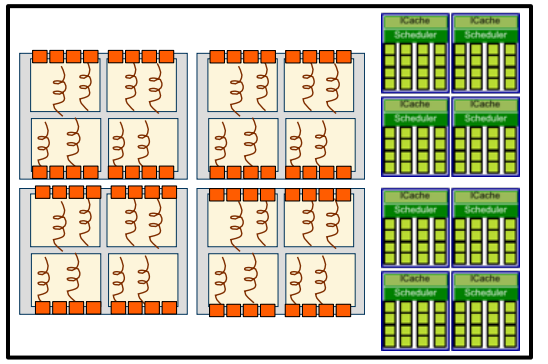


GPU



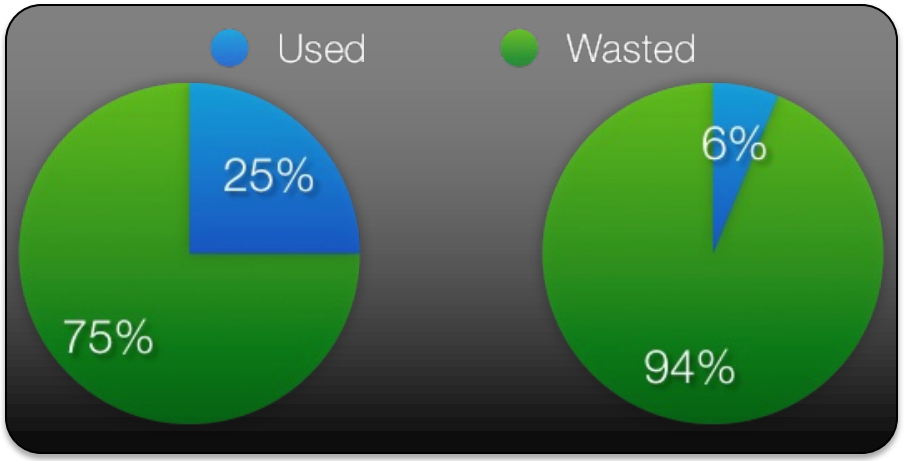


Cluster



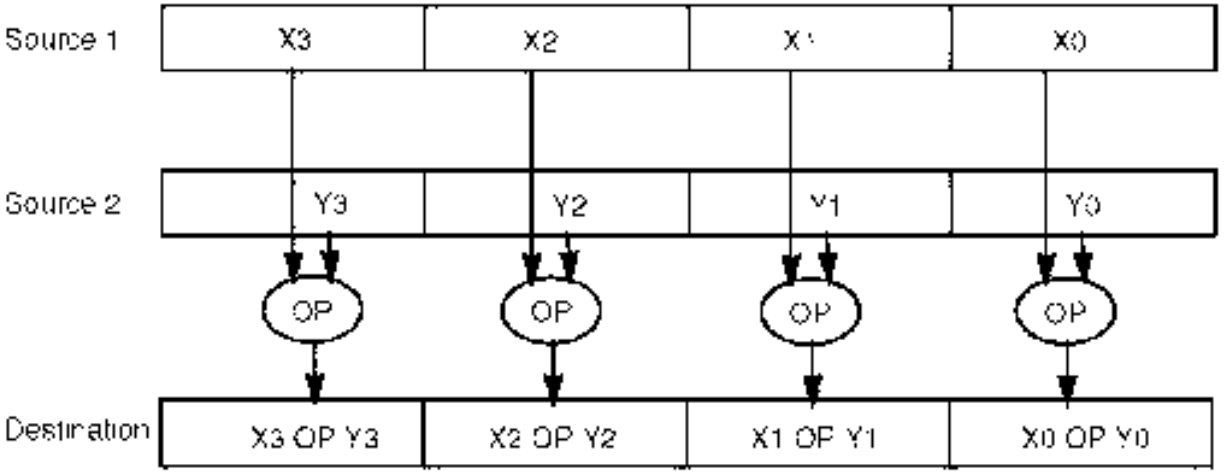
Heterogeneous node

Vector (SIMD) Programming



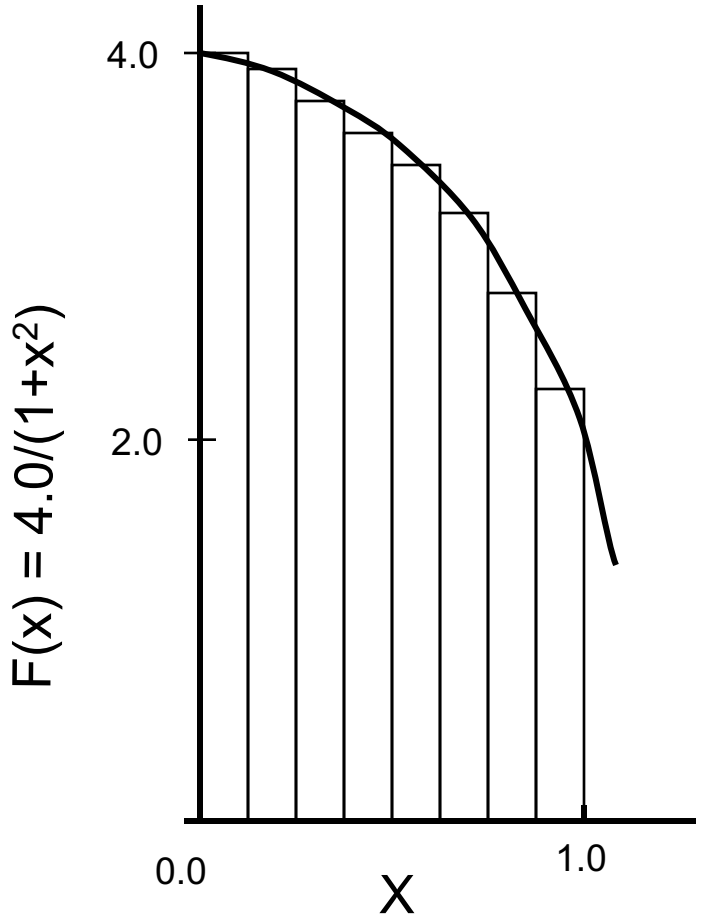
4 way SIMD (SSE)

16 way SIMD (Xeon™ PHI)



- Architects love vector units, since they permit space- and energy-efficient parallel implementations.
- However, standard SIMD instructions on CPUs are inflexible, and can be difficult to use.
- Options:
  - Let the compiler do the job
  - Assist the compiler with language level constructs for explicit vectorization.
  - Use intrinsics … an assembly level approach.

Example Problem: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI program

Literals as double (no-vec), 0.012 secs
Literals as Float (no-vec), 0.0042 secs

```
static long num_steps = 100000;
float step;
int main ()
{           int i;      float x, pi, sum = 0.0;

            step = 1.0/(float) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Normally, I'd use double types throughout to minimize roundoff errors especially on the accumulation into sum. But to maximize impact of vectorization for these exercise, we'll use float types.

Explicit vectorization of our Pi Program: Step 1 ... Unroll the loop

- We need one interation to fit in the vector unit

- What is the width of your vector unit?
- We'll use SSE which is 128 bits wide.
- A float in C is 32 bits wide … 4 floats fits in 128 bits

So unroll our loop by four

```
float pi_unroll(int  num_steps)
{
   float step, x0, x1, x2, x3, pi, sum = 0.0;

   step = 1.0f/(float) num_steps;

   for (int i=1;i<= num_steps; i=i+4){      //unroll by 4, assume num_steps%4 = 0
      x0 = (i-0.5f)*step;
      x1 = (i+0.5f)*step;
      x2 = (i+1.5f)*step;
      x3 = (i+2.5f)*step;
      sum += 4.0f*(1.0f/(1.0f+x0*x0) + 1.0f/(1.0f+x1*x1)  + 1.0f/(1.0f+x2*x2)  + 1.0f/(1.0f+x3*x3));
   }

     pi = step * sum;
     return pi;
}
```

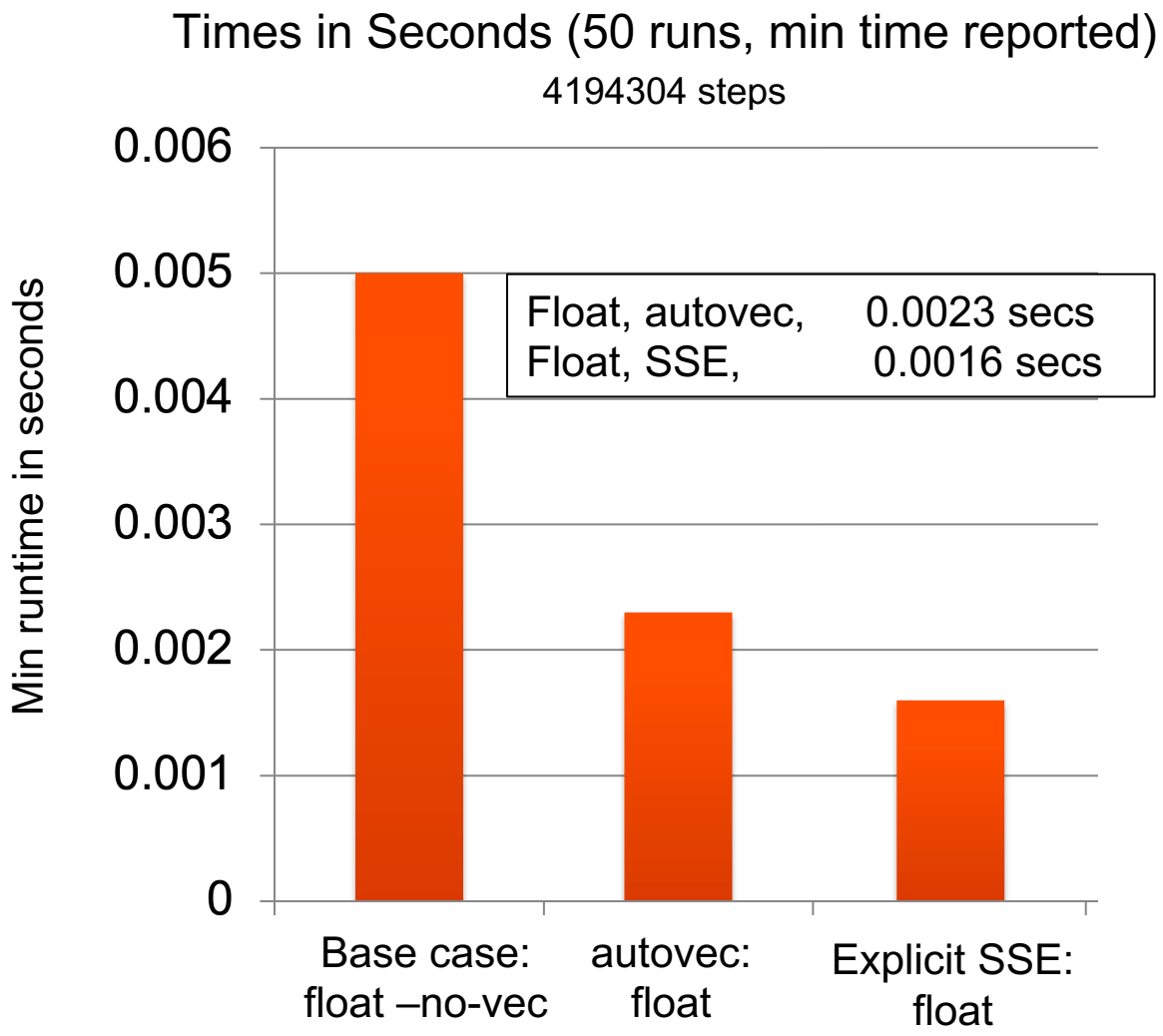
Explicit vectorization of our Pi Program: Step 2 … Add SSE intrinsics

```
#include <immintrin.h>
float pi_sse(int  num_steps)
 {
   float scalar_one =1.0, scalar_zero = 0.0,  ival, scalar_four =4.0, step, pi, vsum[4];
   step = 1.0/(float) num_steps;

   __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
   __m128 one    = _mm_load1_ps(&scalar_one);
   __m128 four   = _mm_load1_ps(&scalar_four);
   __m128 vstep  = _mm_load1_ps(&step);
   __m128 sum    = _mm_load1_ps(&scalar_zero);
   __m128 xvec;   __m128 denom;  __m128 eye;

  for (int i=0;i< num_steps; i=i+4){          // unroll loop 4 times
      ival       = (float)i;                  // and assume num_steps%4 = 0
      eye        = _mm_load1_ps(&ival);
      xvec       = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
      denom  = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
      sum      = _mm_add_ps(_mm_div_ps(four,denom),sum);
   }
   _mm_store_ps(&vsum[0],sum);
   pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
  return pi;
}
```

The vast majority of programmers never write explicitly vectorized code.

It is important to understand explicit vectorization so you appreciate what the compiler does to vectorize code for you

PI program Results:



- Intel Core i7, 2.2 Ghz, 8 GM 1600 MHz DDR3, Apple MacBook Air OS X 10.10.5.
- Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 15.0.3.187 Build 20150408

Helping the compiler to vectorize

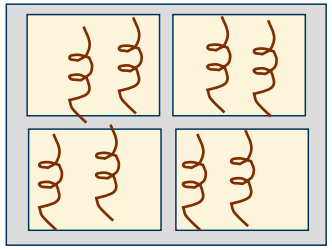
- Vectorization is enabled in gcc by the flags:
  - -ftree-vectorize
  - -O3
- Vectorizable:
  - Countable innermost loops
  - No variations in the control flow
  - Contiguous memory access
  - Independent memory access
- Avoid aliasing problems with restrict
- Use countable loops, with no side effects (break, continue, non-inlined function calls)
- Avoid indirect memory access (x[y[i]])

Source: Introduction to Architecture and Performance, Felice Pantaleo, ESC'23

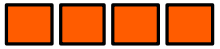
OK, let’s wrap up and conclude all this architecture stuff

Parallelism: we've covered the "on-node" cases

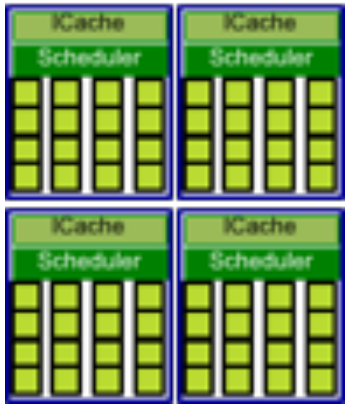
All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



CPU

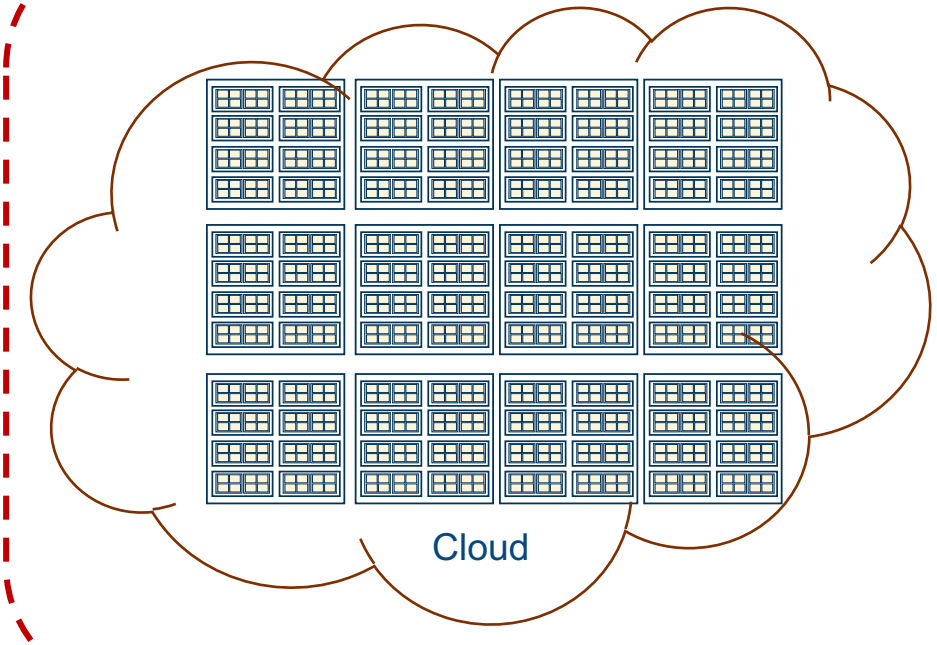


SIMD/Vector

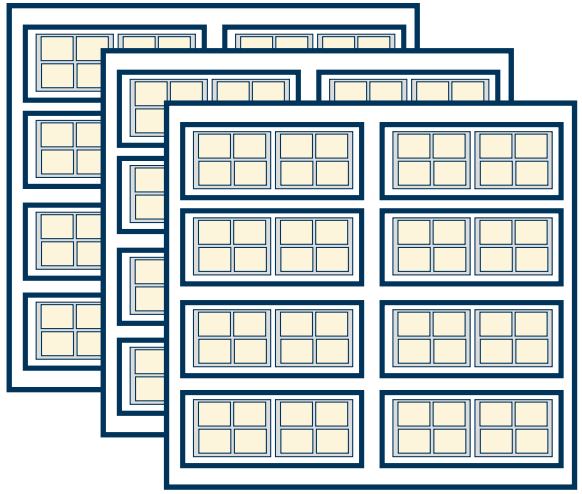


GPU

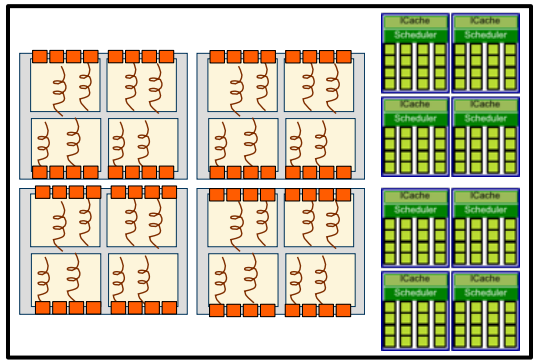
We'll discuss these later … after you understand how to write software for the "on-node" cases



Cloud



Cluster



Heterogeneous node

Summary

- We've covered more material in this one lecture than you can possibly absorb. Sorry. Use these slides as reference material and talk to me during out time together. Working together, you can master what you need.

- I'll give the closing statement to Sverre Jarp … an old friend from the Open Lab at CERN

Computer Architecture and Performance Tuning

So, what does it all mean?

- **Here is what we tried to say in these lectures:**
- **You must make sure your data and instructions come from caches (not main memory)**
- **You must parallelise across all "CPU slots"**
  - Hardware threads, Cores, Sockets — **10x – 100x**
- **You must get your code to use vectors** — **2x, 4x, 8x, 16x**
- **You must understand if your ILP is seriously limited by serial code, complex math functions, or other constructs** — **1x – 10x**

Sverre Jarp

Outline

- Key concepts in computer architecture
- A brief history of supercomputing
- Programming languages and choice overload: Python, C/C++, Fortran

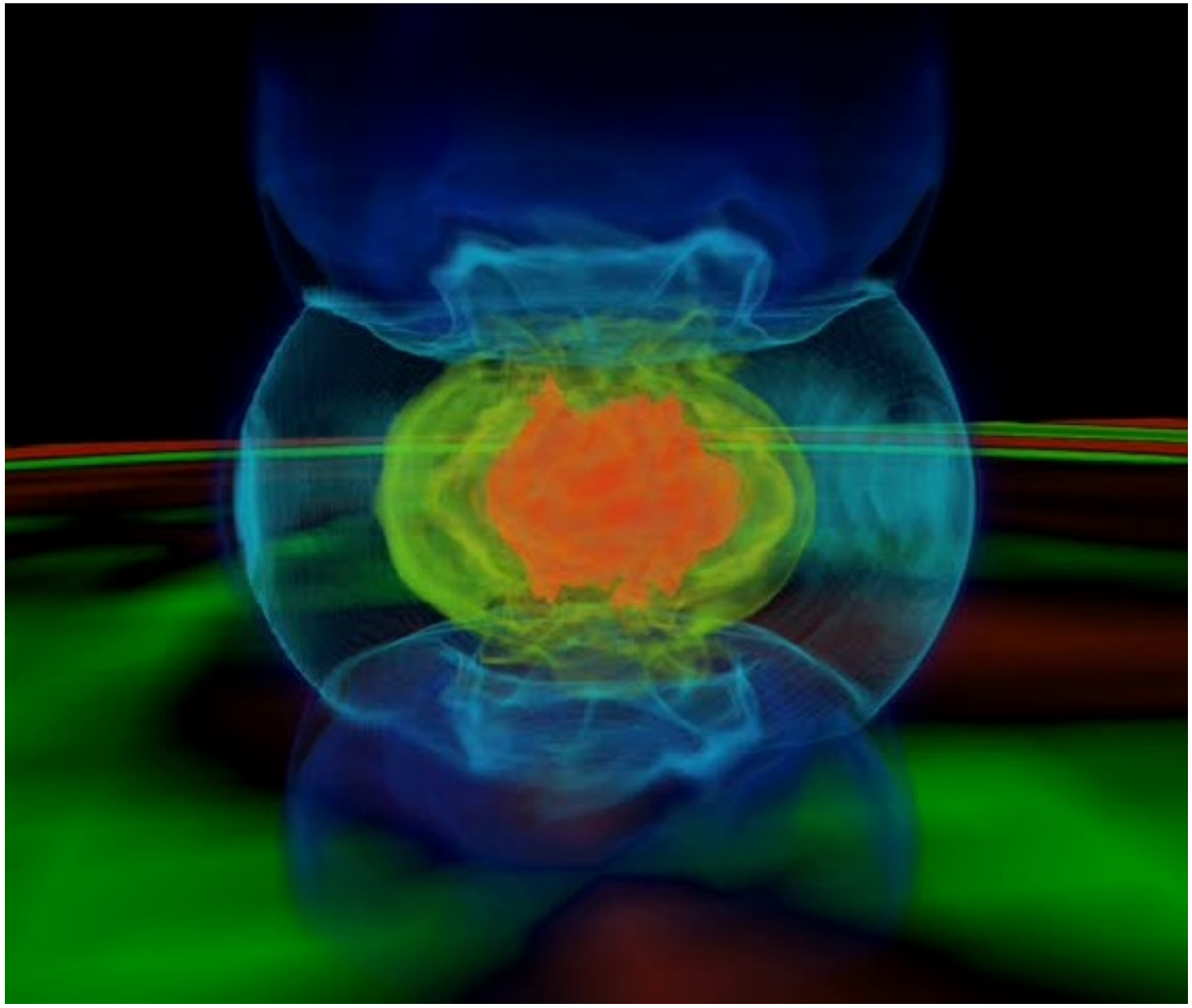
The Essence of supercomputing: Totally cool science

Gamma ray bursts from a core collapse supernova

Rotationally deformed protoneutron star, formed in a core collapse

[C. D. Ott; R. Kähler]

Scientific computing on big supercomputers is addictive. Once you wrap your brain around these sorts of problems, there is no going back.



Source: John Shalf, UCB CS294, Spring 2009

The birth of Supercomputing

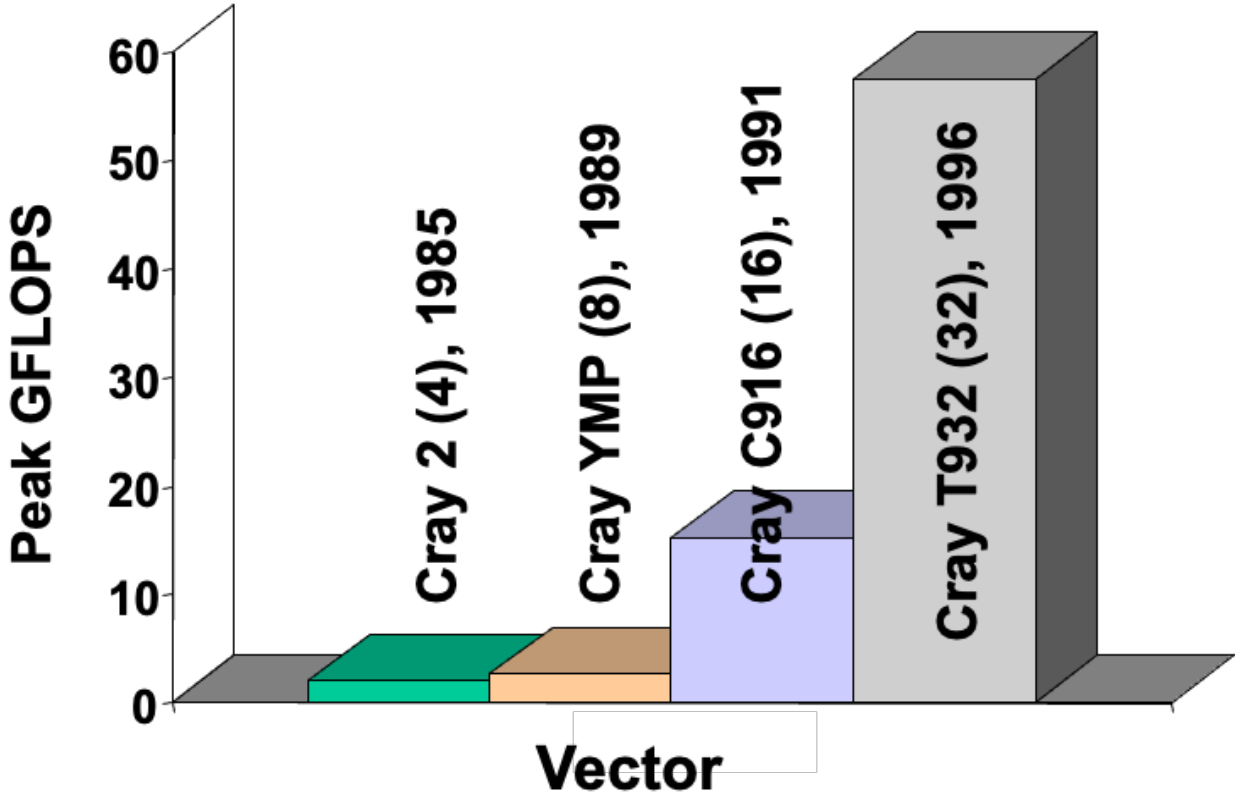
- On July 11, 1977, the CRAY-1A, serial number 3, was delivered to NCAR. The system cost was $8.86 million ($7.9 million plus $1 million for the disks).

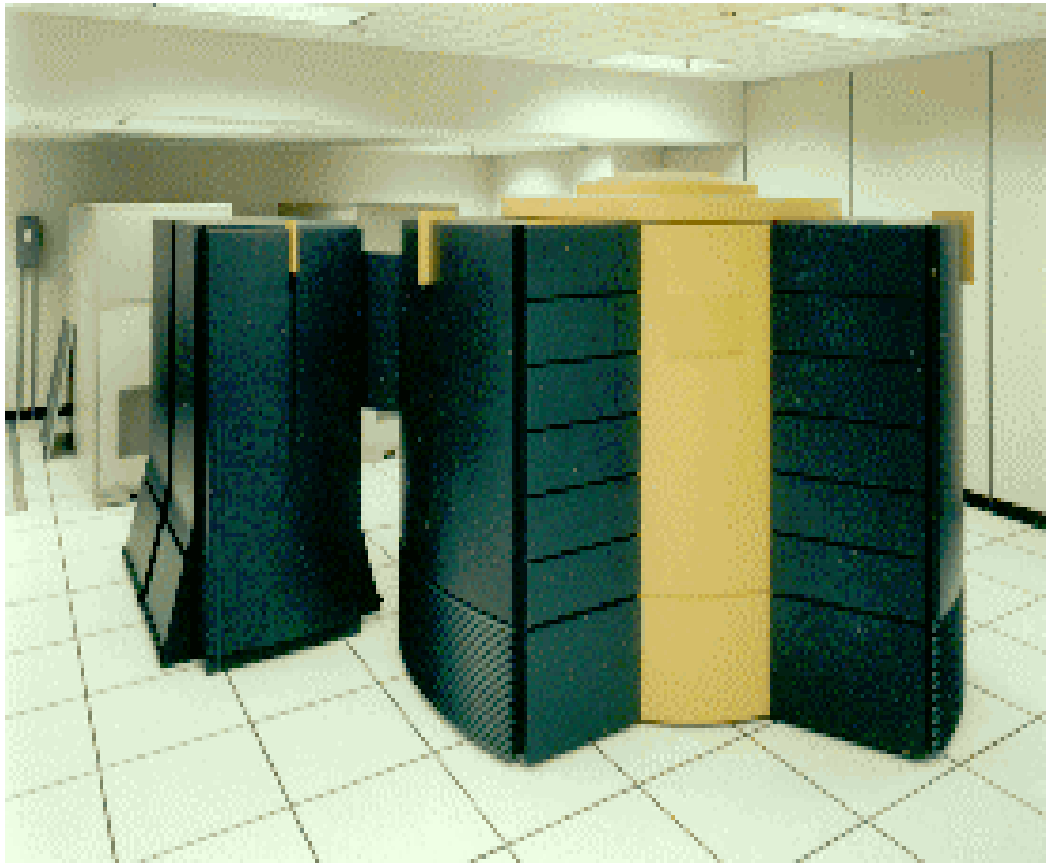
http://www.cisl.ucar.edu/computers/gallery/cray/cray1.jsp

- The CRAY-1A:
  - 2.5-nanosecond clock,
  - 64 vector registers,
  - 1 million 64-bit words of high-speed memory.
  - Peak speed:
    - 80 MFLOPS scalar.
    - 250 MFLOPS vector (but this was VERY hard to achieve)
- Cray software … by 1978
  - Cray Operating System (COS),
  - the first automatically vectorizing Fortran compiler (CFT),
  - Cray Assembler Language (CAL) were introduced.

History of Supercomputing: The Era of the Vector Supercomputer

- Large mainframes that operated on vectors of data
- Custom built, highly specialized hardware and software
- Multiple processors in an shared memory configuration
- Required modest changes to software (vectorization)





The Cray C916/512 at the Pittsburgh Supercomputer Center, 1991

GFLOPs = gigaFLOPS = Billion flops per second

The attack of the killer micros

Image source: http://calteches.library.caltech.edu/3419/1/Cubism.pdf

The cosmic cube, Charles Seitz, Communications of the ACM, Vol 28, number 1 January 1985, p. 22

- The **Caltech Cosmic Cube** developed by Charles Seitz and Geoffrey Fox in1981
- 64 Intel 8086/8087 processors
- 128kB of memory per processor
- 6-dimensional hypercube network

- Used off-the-shelf CPUs but the node-boards were custom designed around the needs of the network.

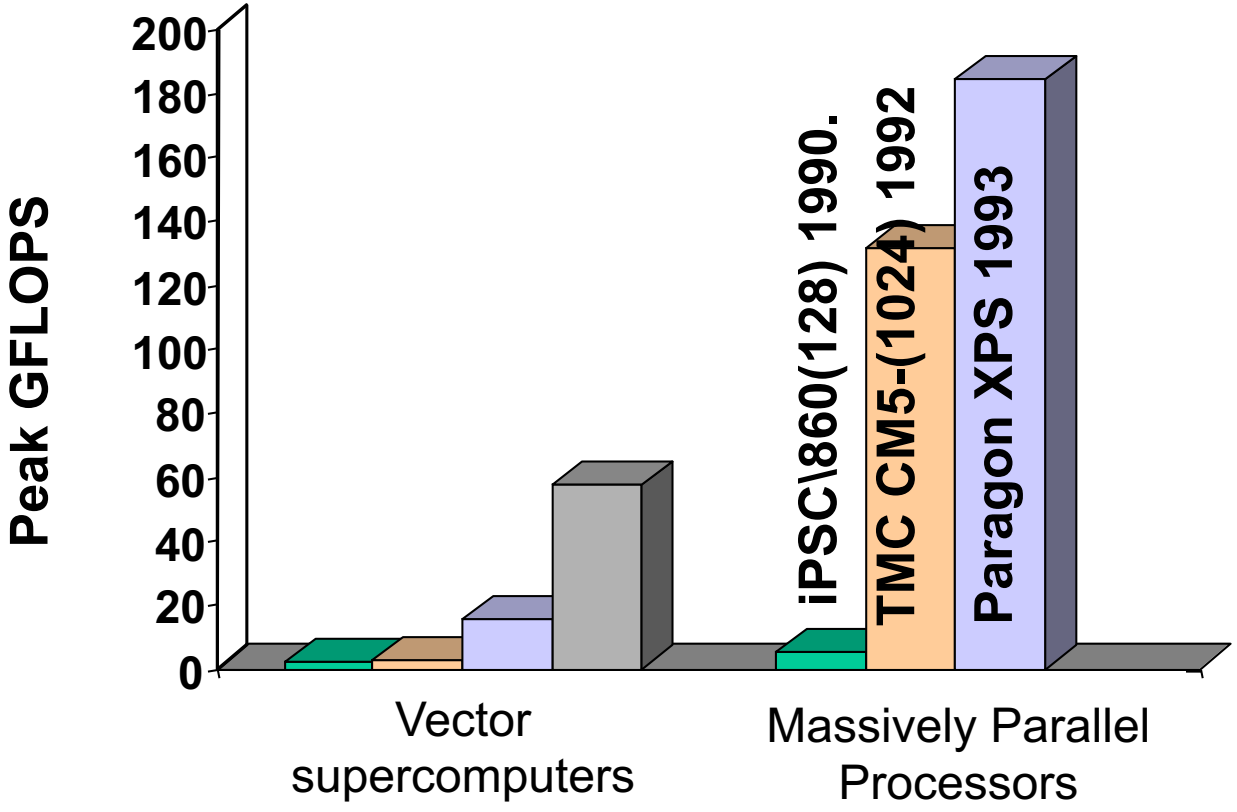
- We expected these to grow to huge sizes so we called these **Massively Parallel Processors** (**MPP**)

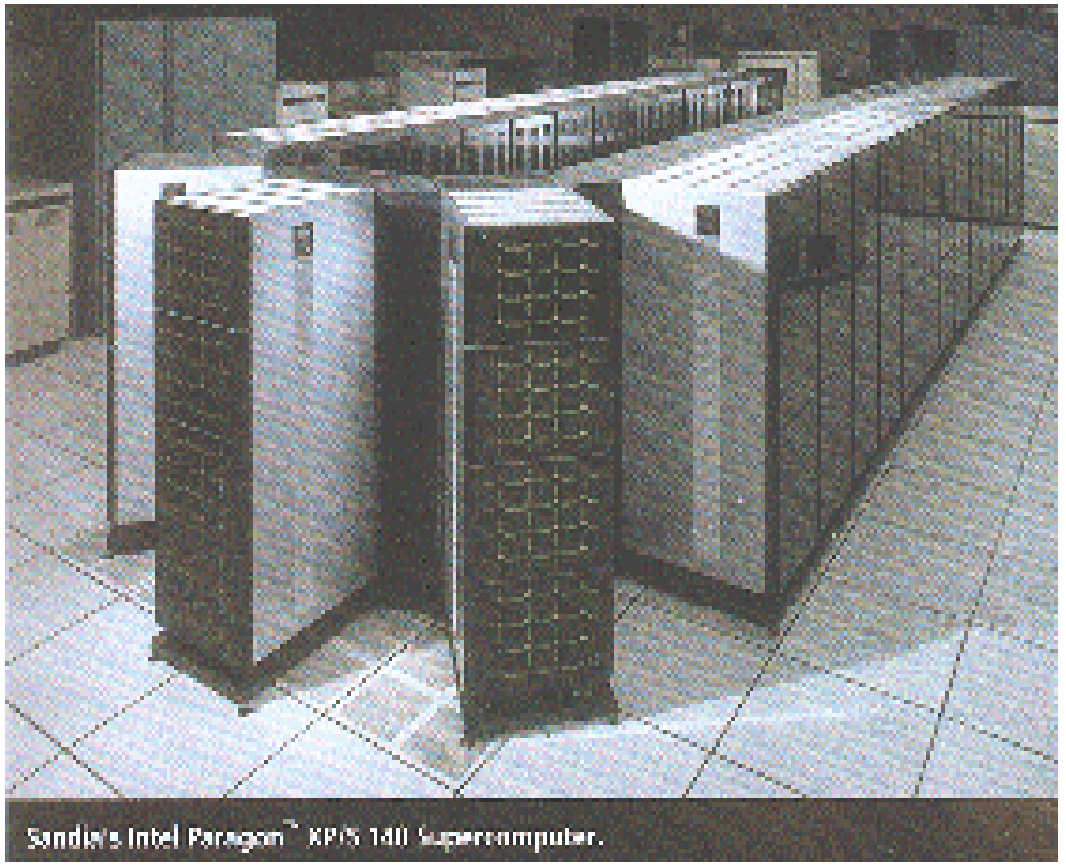
From the perspective of the Cray vector machines … The cosmic cube launched what came to be known as the “attack of the killer micros”
Eugene Brooks, SC’90

As a postdoc in chemical physics in 1985, this project is where I got my start in parallel computing

It took a while, but MPPs came to dominate supercomputing

- Parallel computers with large numbers of microprocessors
- High speed, low latency, scalable interconnection networks
- Lots of custom hardware to support scalability
- Required massive changes to software (parallelization)





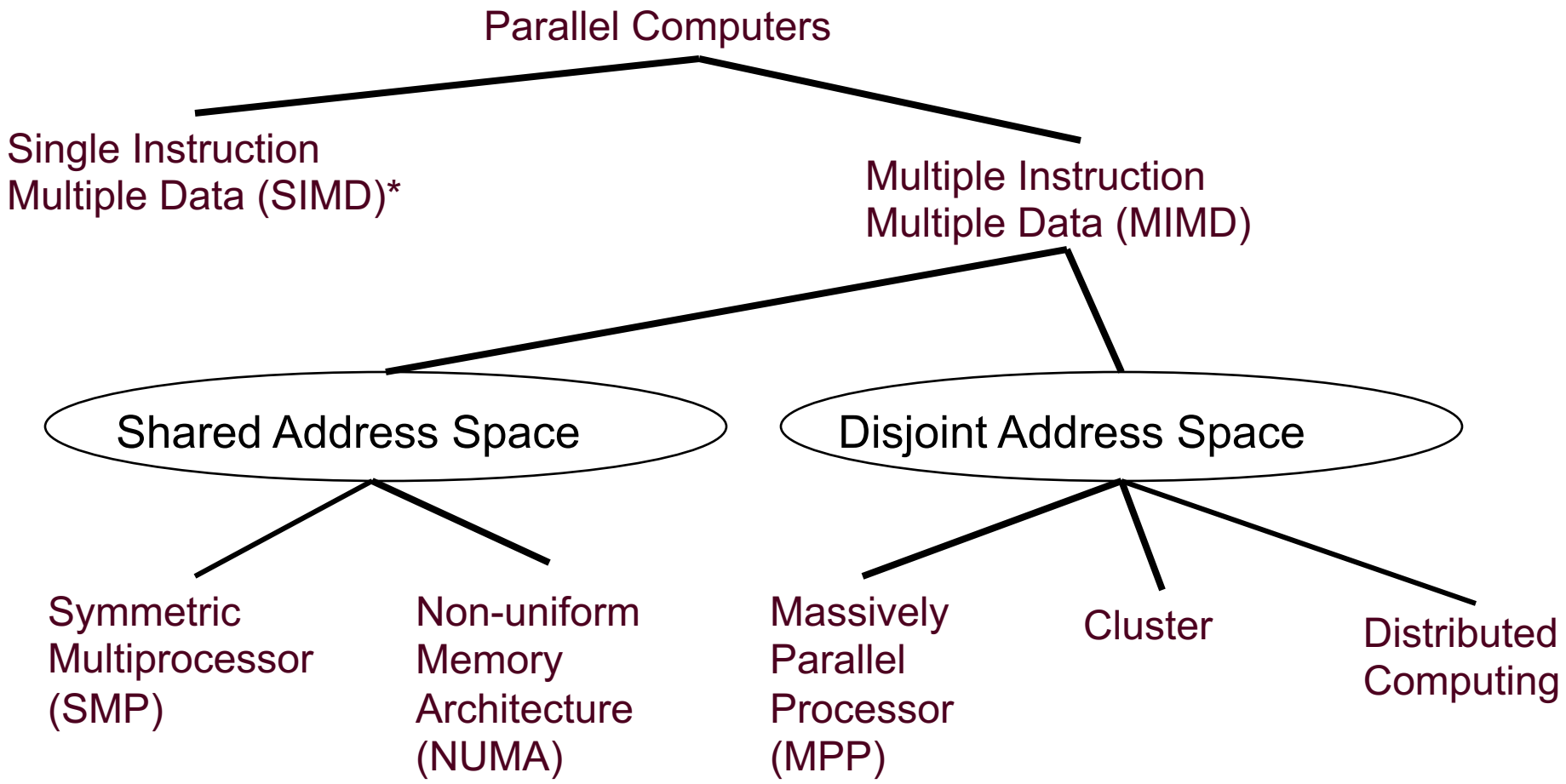
Paragon XPS-140 at Sandia National labs in Albuquerque NM, 1993

The MPP future looked bright … but then clusters took over

- A cluster is a collection of connected, independent computers that work in unison to solve a problem.
- Nothing is custom … motivated users could build cluster on their own

- First clusters appeared in the late 80's
- The Intel Pentium Pro in 1995 coupled with Linux made them competitive.
- NASA Goddard's Beowulf cluster demonstrated publically that high visibility science could be done on clusters.
- Clusters made it easier to bring the benefits due to Moore's law into working supercomputers

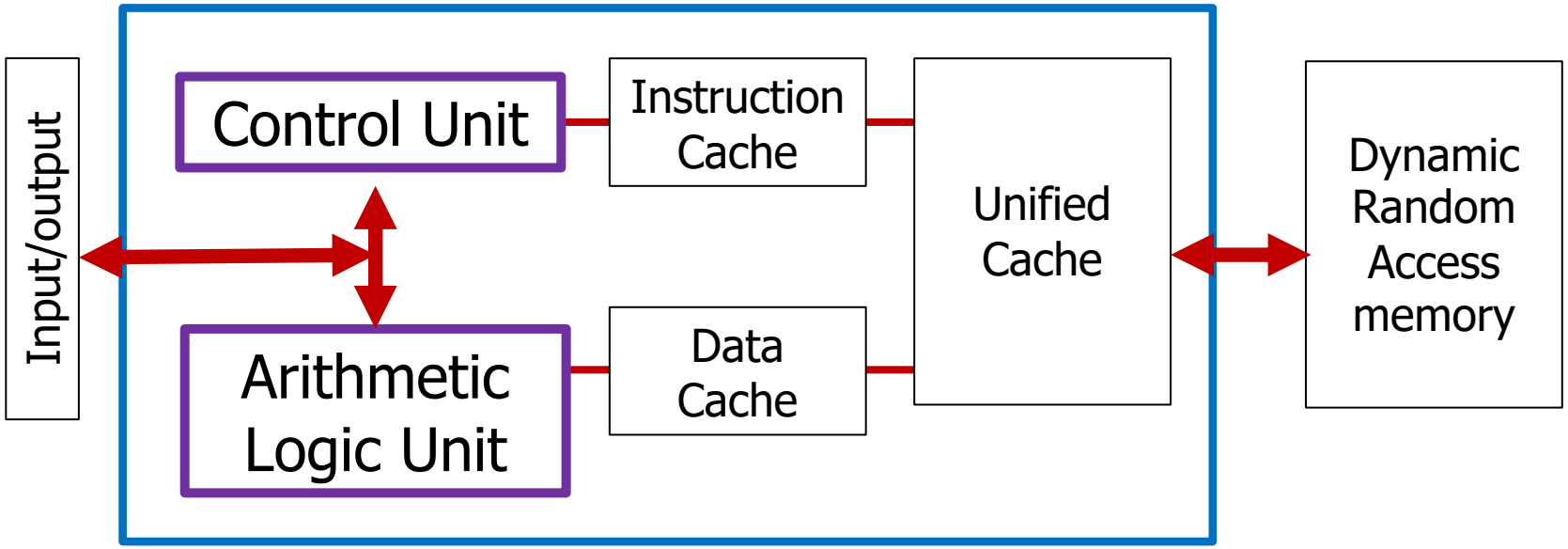
Hardware for High Performance Computing (HPC)



*SIMD has failed as a way to organize large scale computers with multiple processors. It has succeeded, however, as a mechanism to increase instruction level parallelism in modern microprocessors (MMX, SSE, AVX, etc.).

How do we build Supercomputers today?

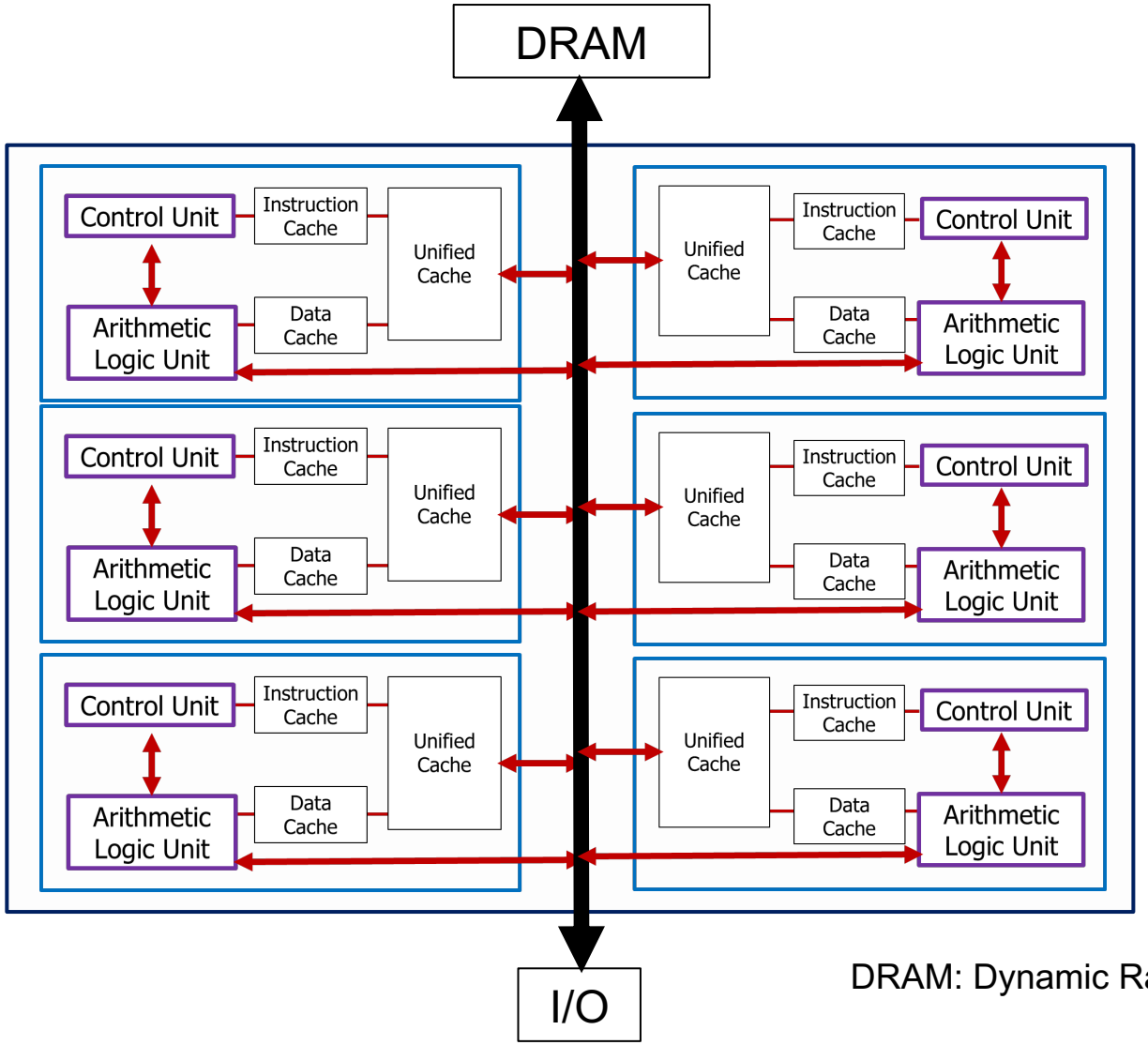
- Take advantage of Moore's law and the economy of scale.
- Start with a simple von-Neumann computer..



6 cores

How do we build Supercomputers today?

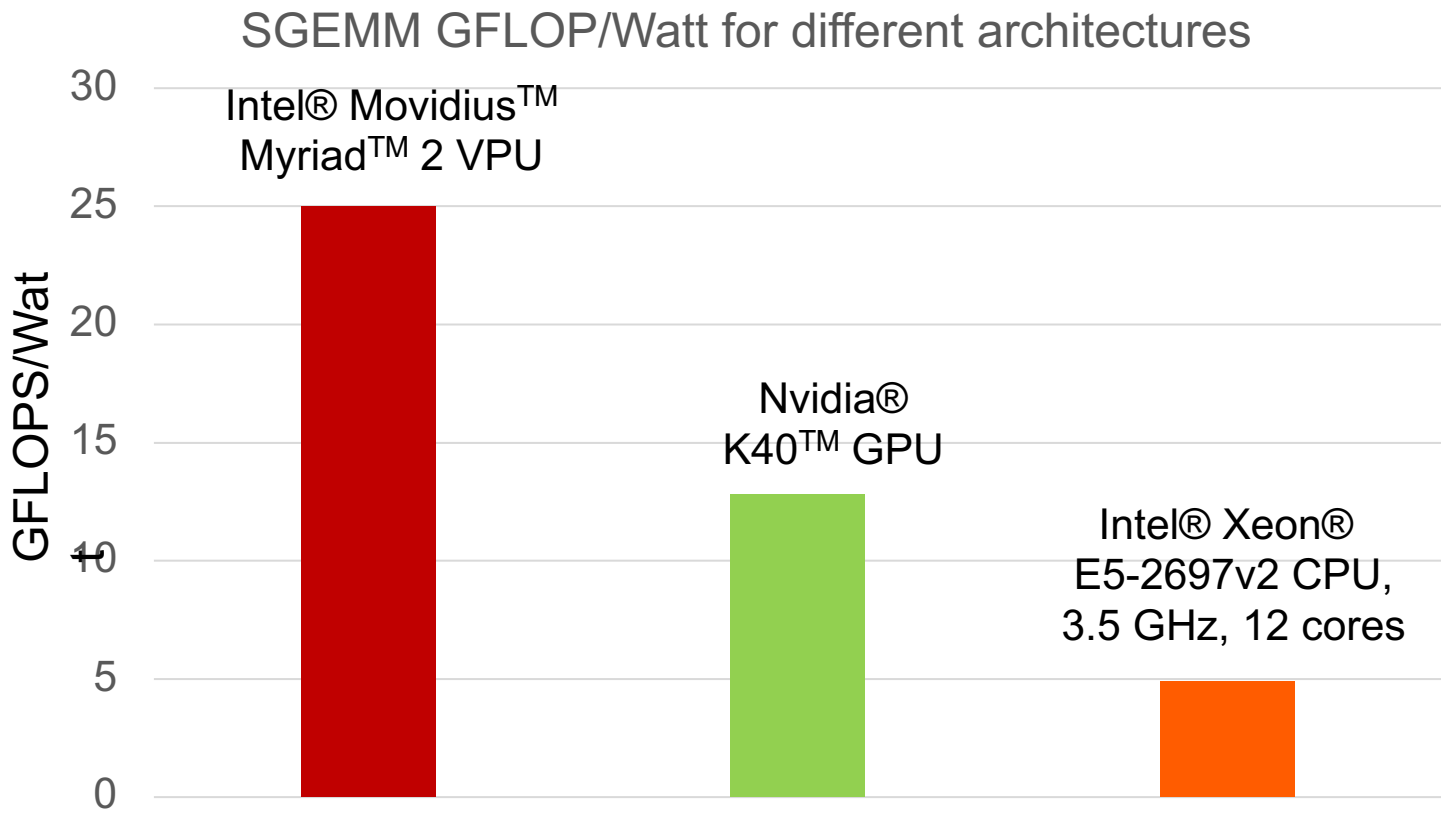
- Put a whole bunch of simple von-Neumann computers onto a single silicon chip (a multi-core chip).



DRAM: Dynamic Random Access Memory

If you care about power, the world is heterogeneous?

Specialized processors doing operations suited to their architecture are more efficient than general purpose processors.



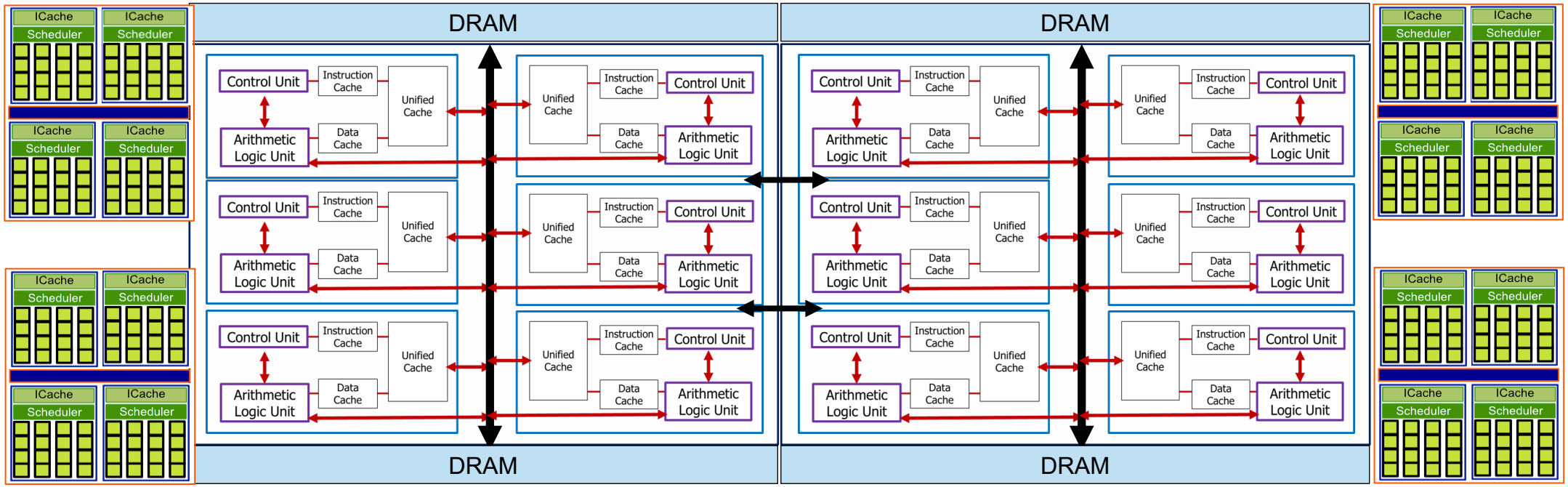
Hence, future systems will be increasingly heterogeneous … GPUs, CPUs, FPGAs, and a wide range of accelerators

Source: Suyash Bakshi and Lennart Johnsson, “A Highly Efficient SGEMM Implementation using DMA on the Intel/Movidius Myriad-2. IEEE International Symposium on Computer Architecture and High Performance Computing, 2020

6 cores

How do we build Supercomputers today?

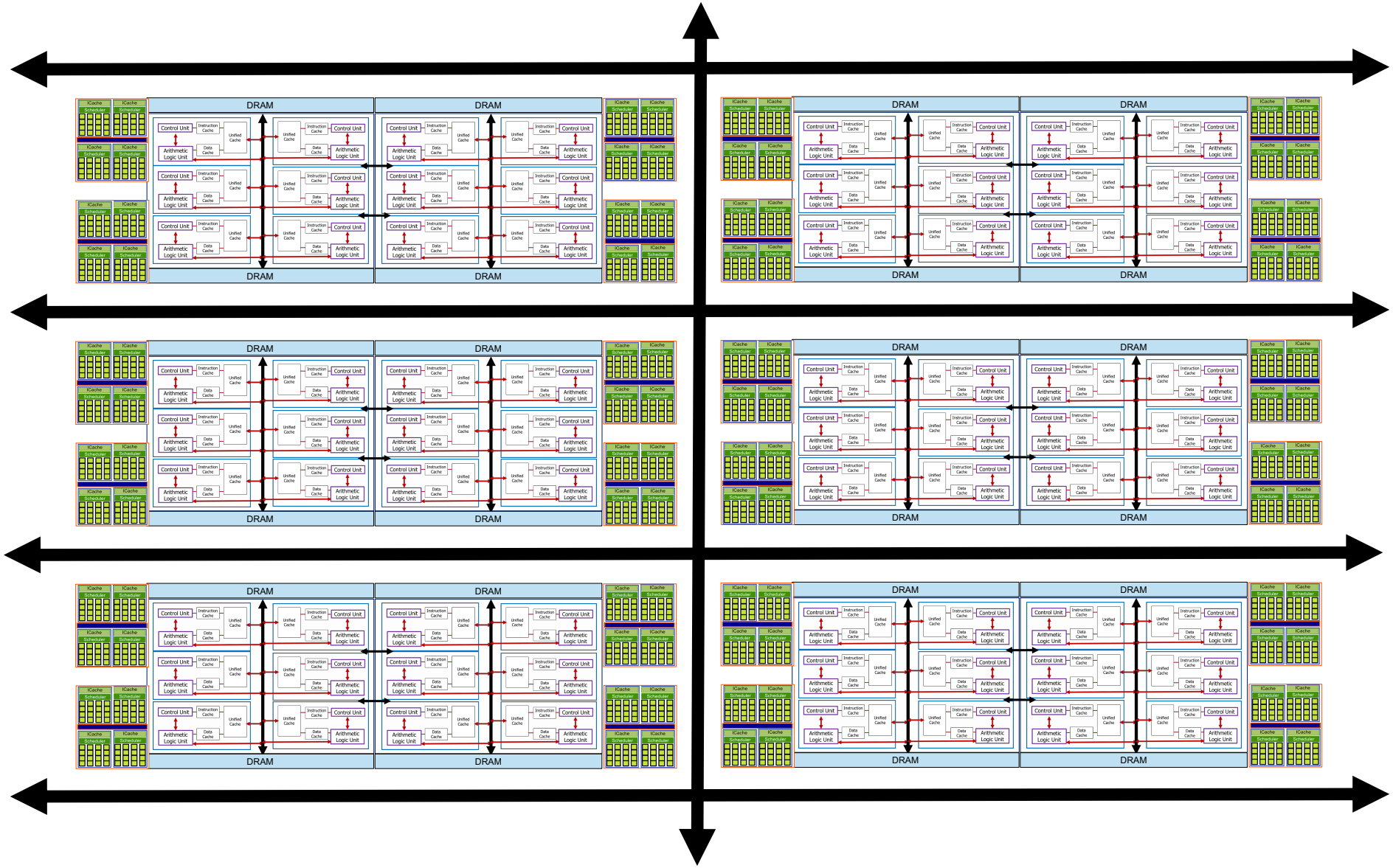
- So we combine multicore CPUs with several GPUs to build a node



72 cores

How do we build Supercomputers today?

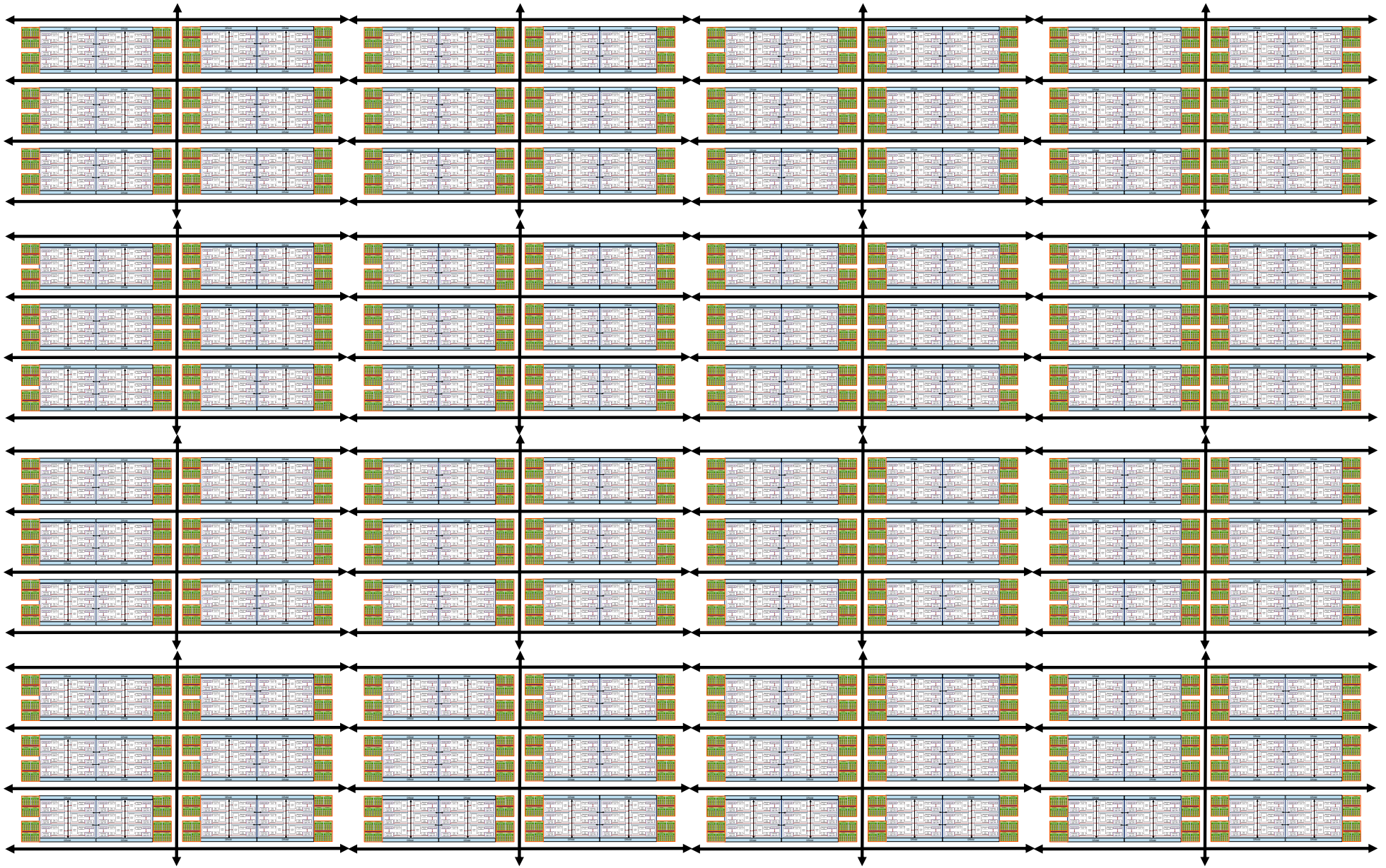
- Combine nodes to build a cluster



1152 cores

How do we build Supercomputers today?

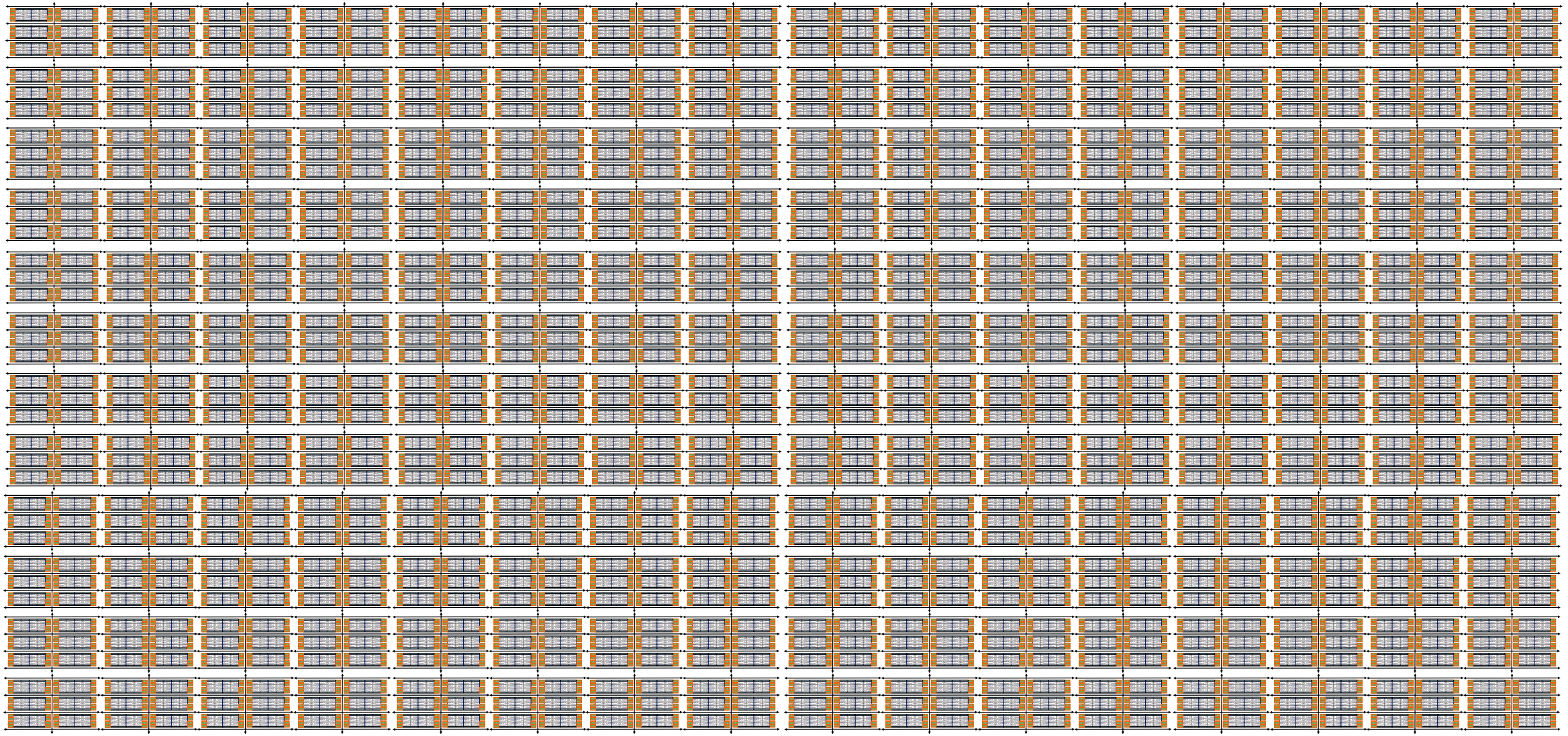
- Do this a whole bunch



10368 cores

How do we build Supercomputers today?

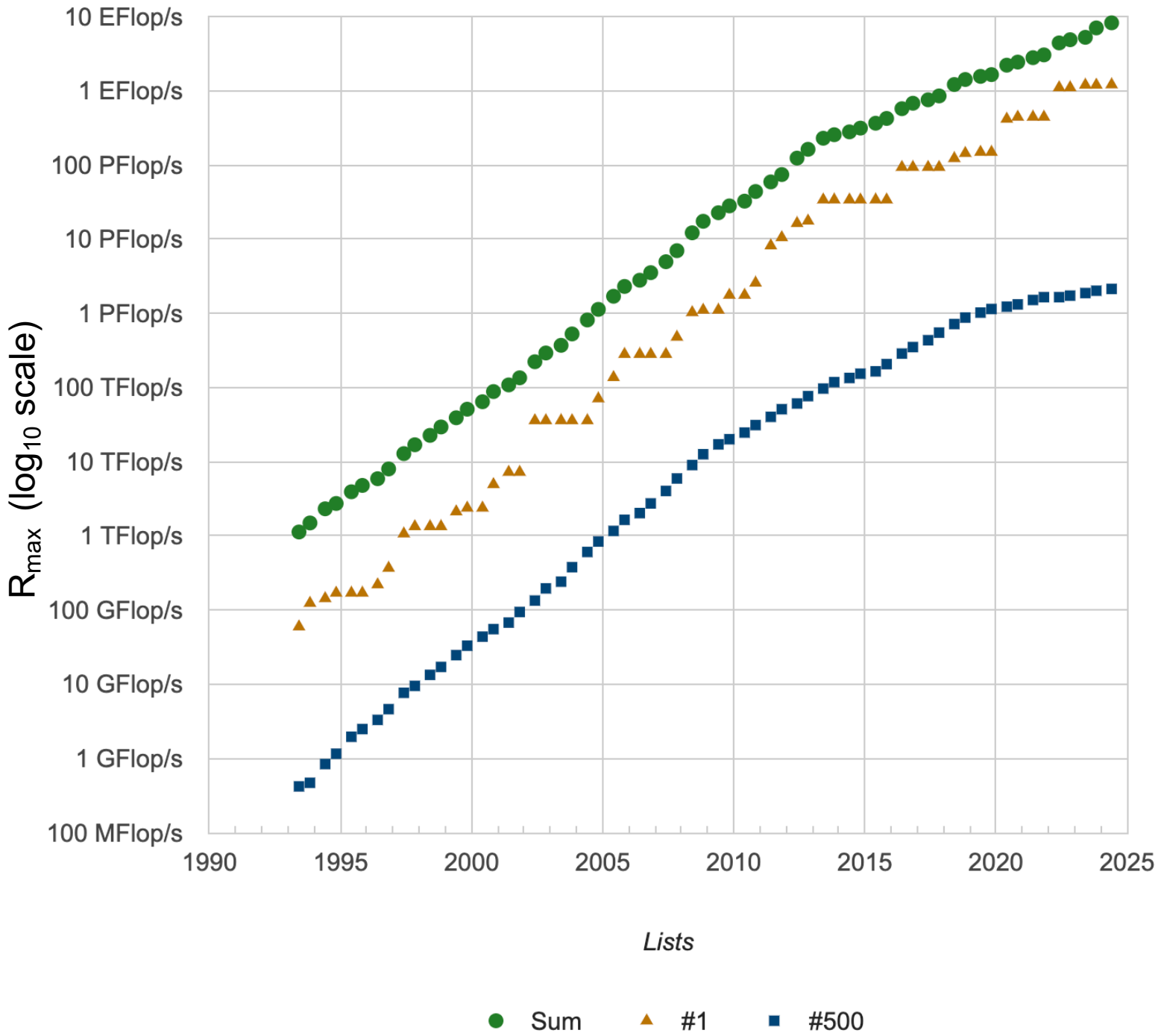
- Keep going until you run out of money (or can't afford the electricity bill)



The World's Fastest Supercomputers

- Top500: a list of the 500 fastest computers in the world
  - www.top500.org
- Computers ranked by solution to the MPLinpack benchmark:
  - Solve Ax=b problem for any order of A
- List released twice per year: in June and November
- List includes :

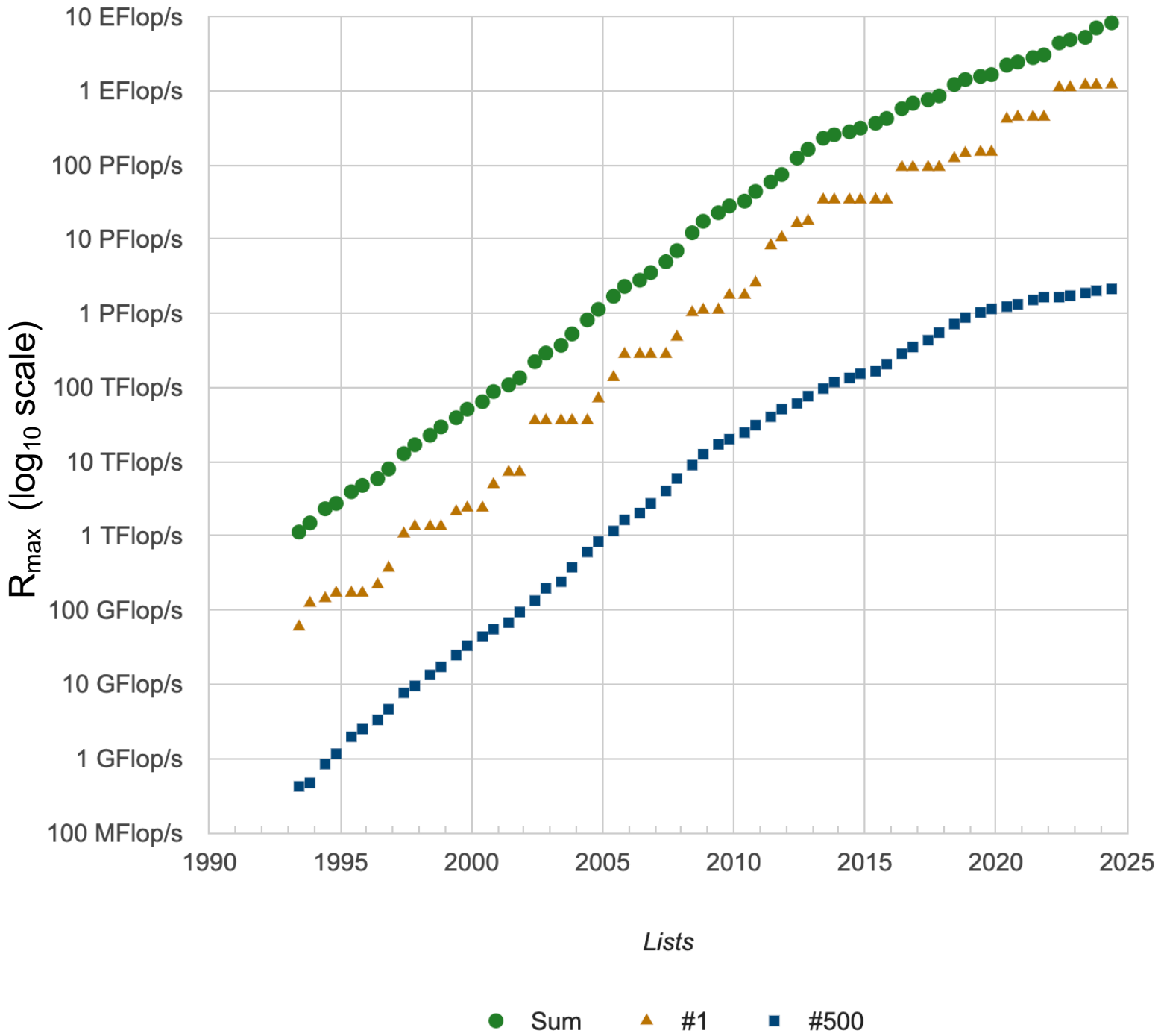
- **R_{max}** **Maximal LINPACK performance**
- **R_{peak}** **Theoretical peak performance**
- **N_{max}** **Problemsize for R_{max}**
- **$N_{1/2}$** **Problemsize for half of R_{max}**



The World's Fastest Supercomputers

- Major Milestones …
  - CRAY 2 ... 4 vector processor, SMP.

1st GFLOP
1985, Cray 2



FLOP: floating point Operations

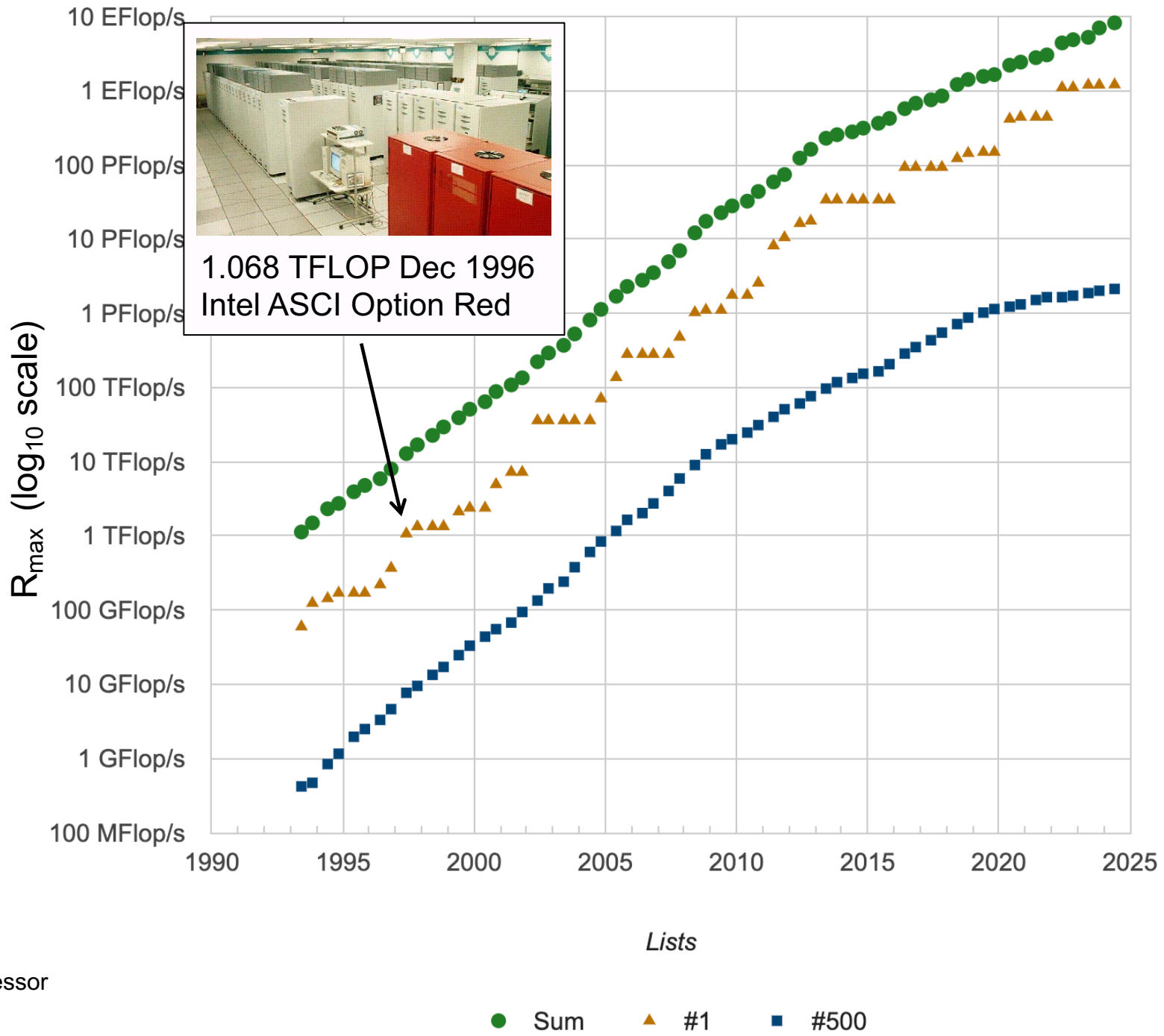
SMP: Symmetric Multiprocessor

GFLOP: 10^9 FLOPs

The World's Fastest Supercomputers

- Major Milestones …
  - CRAY 2 ... 4 vector processor, SMP.
  - Intel's ASCI Red … MPP with 9298 Pentium Pro CPUs.

1st GFLOP
1985, Cray 2



1.068 TFLOP Dec 1996
Intel ASCI Option Red

FLOP: floating point Operations

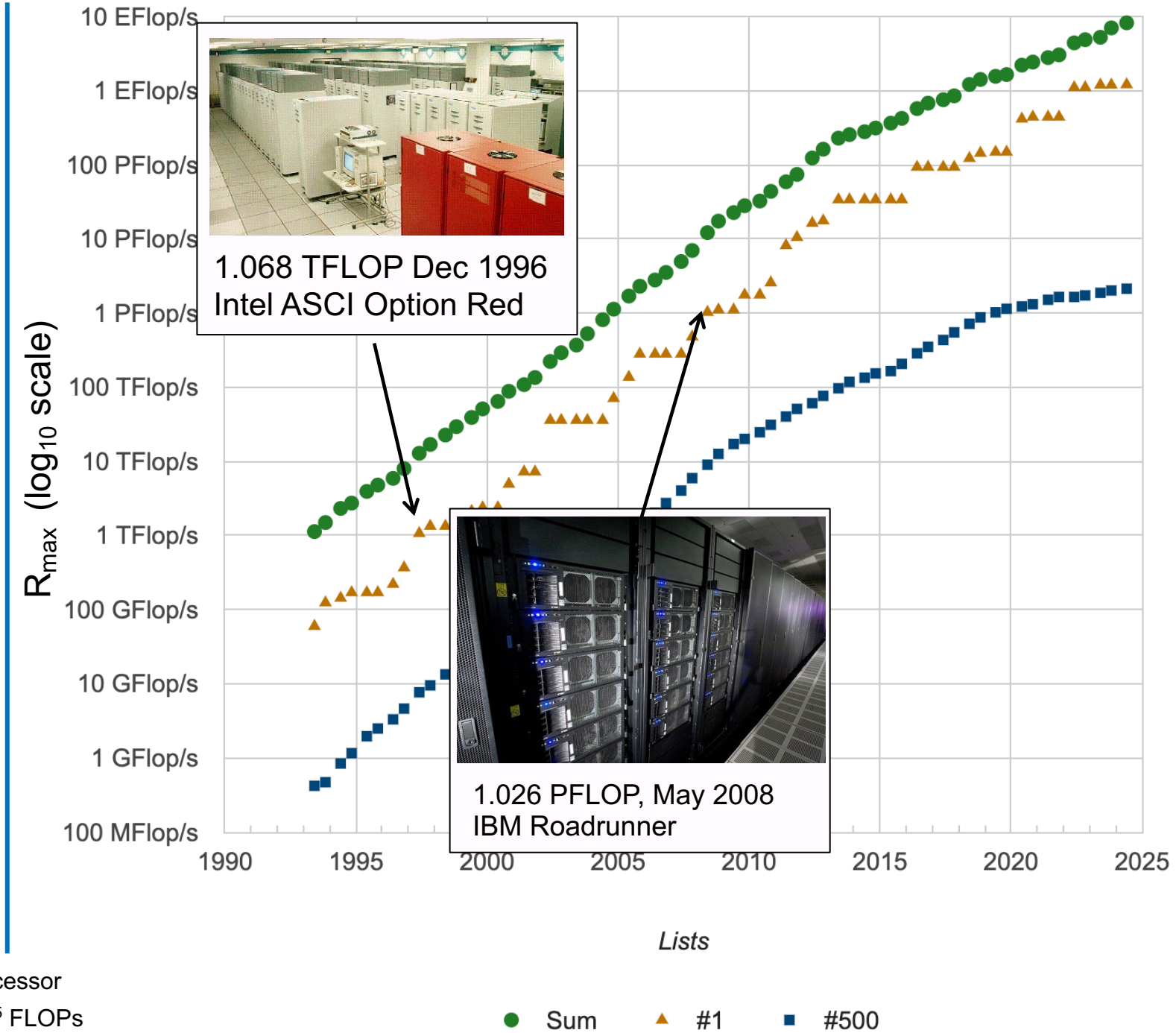
SMP: Symmetric Multiprocessor MPP: Massively Parallel Processor

GFLOP: 10^9 FLOPs TFLOP: 10^{12} FLOPs

The World's Fastest Supercomputers

- Major Milestones …
  - CRAY 2 ... 4 vector processor, SMP.
  - Intel's ASCI Red … MPP with 9298 Pentium Pro CPUs.
  - IBM Roadrunner ... MPP with heterogeneous nodes, 6,480 AMD dual core CPUs and 12,960 BIM PowerXCell 8i accelerators.

1st GFLOP
1985, Cray 2



FLOP: floating point Operations

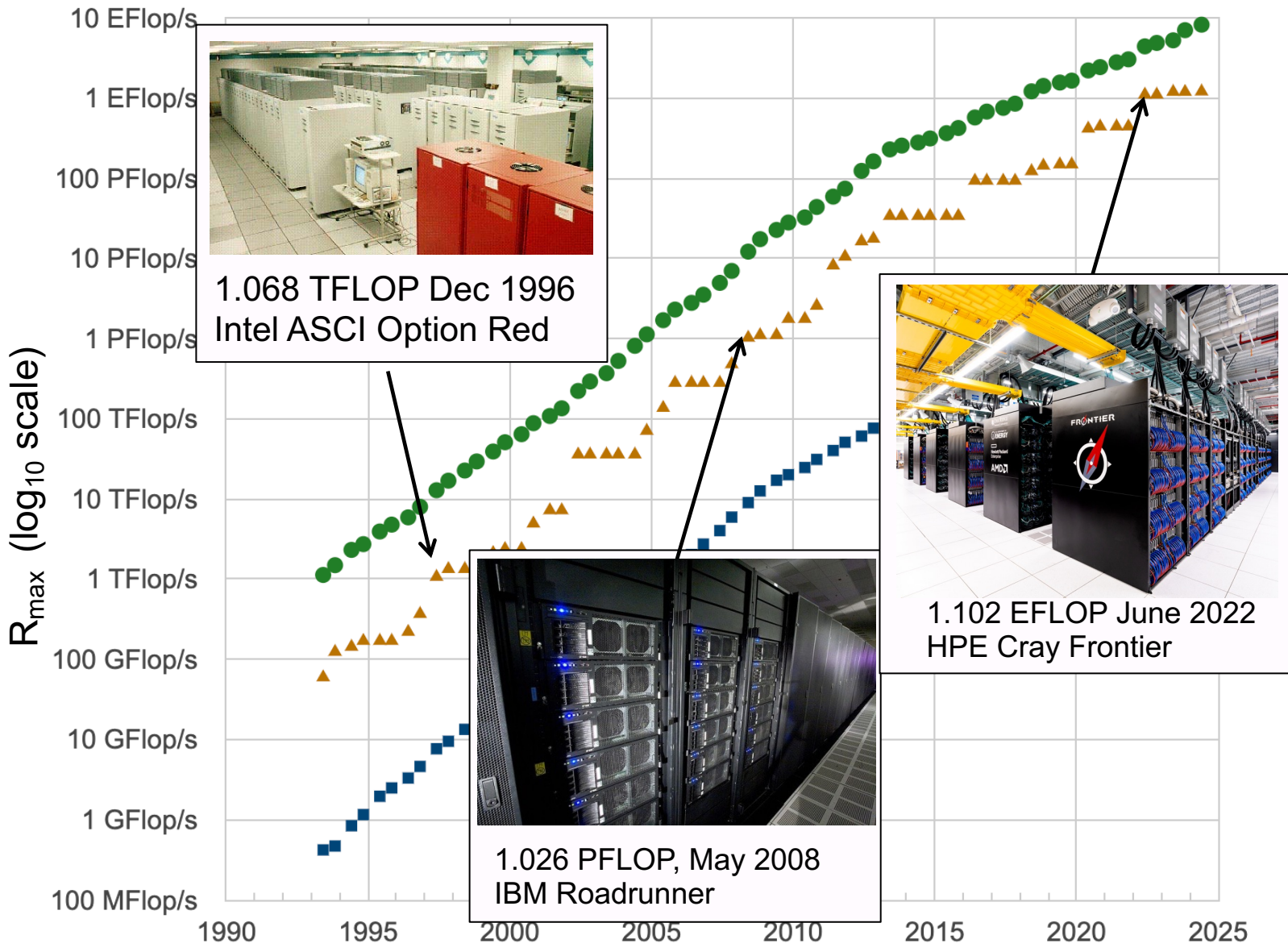
SMP: Symmetric Multiprocessor MPP: Massively Parallel Processor

GFLOP: 10^9 FLOPs TFLOP: 10^{12} FLOPs PFLOP: 10^{15} FLOPs

The World's Fastest Supercomputers

- Major Milestones …
  - CRAY 2 ... 4 vector processor, SMP.
  - Intel's ASCI Red … MPP with 9298 Pentium Pro CPUs.
  - IBM Roadrunner ... MPP with heterogeneous nodes, 6,480 AMD dual core CPUs and 12,960 BIM PowerXCell 8i accelerators.
  - Cray HPE Frontier … MPP with heterogenous nodes, 9472 AMD 64 core SPUS and 37,888 AMD M12590X GPUs.

1st GFLOP
1985, Cray 2



FLOP: floating point Operations

SMP: Symmetric Multiprocessor MPP: Massively Parallel Processor

GFLOP: 10^9 FLOPs TFLOP: 10^{12} FLOPs PFLOP: 10^{15} FLOPs EFLOP: 10^{18} FLOPs

The world's fastest supercomputer

- More details on the Frontier EFLOP supercomputer

- Frontier Cray HP Enterprise system,
- 9,472 AMD Epyc 7713 64-core CPUs @ 2 Ghz
- 37,888 AMD Instinct M1250X GPUs,
- HPE Slingshot-11 network,
- Cost: $600 Million,
- Oak Ridge National Laboratory, USA,
- 22.8 MW

The problem with the top500 list

- Top500 ranks machines by the MP-Linpack benchmark
  - Solve a dense linear system of equations (the Ax=b) problem for arbitrarily large A.
  - Memory access $O(N^2)$ and computation $O(N^3)$.

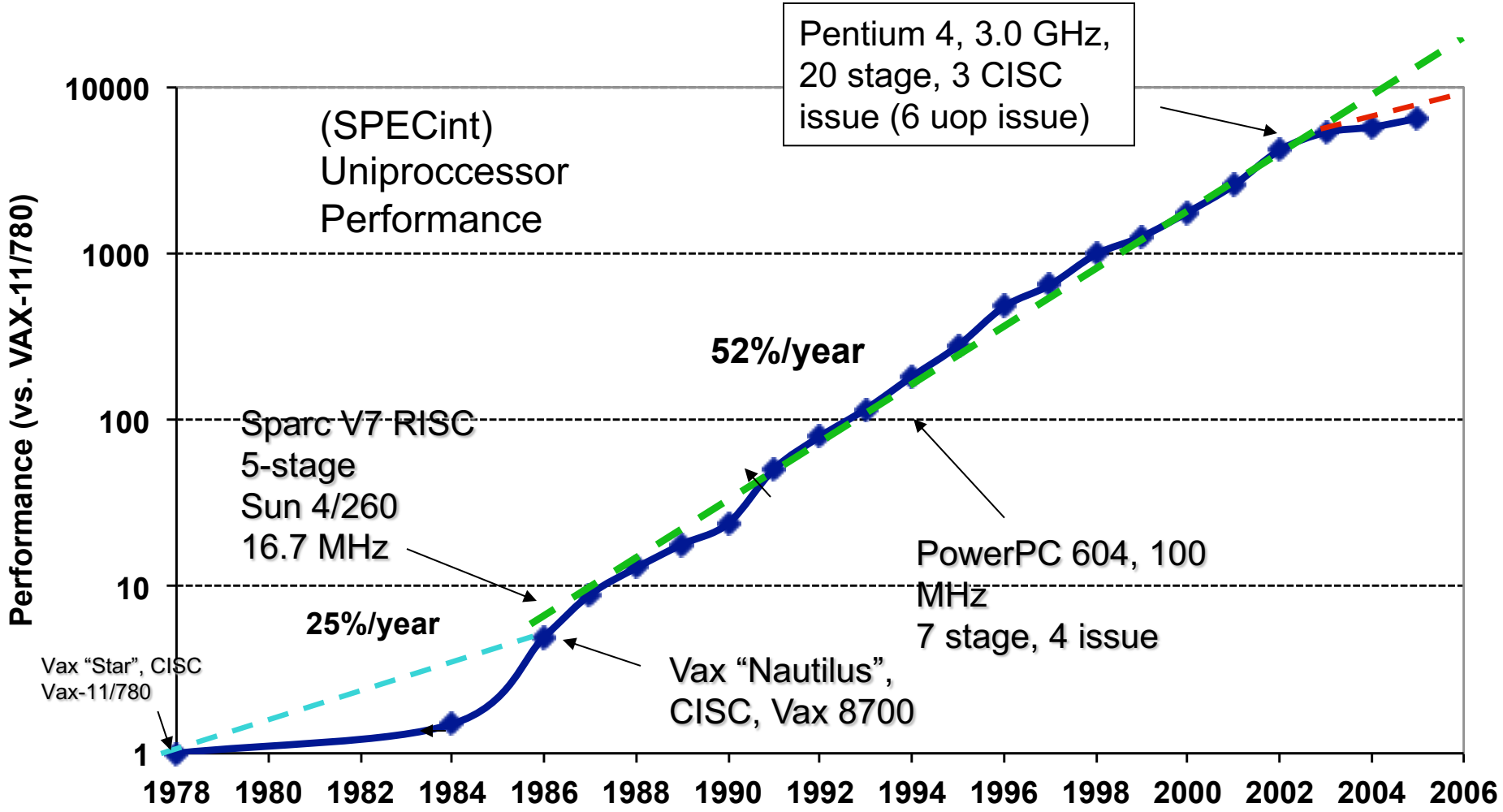
- The problem is that few (if any) real problems solve these sorts of problems

- Since everyone wants to brag about their positions on lists, ranking machines by MP-Linpack leads us to build computers poorly suited to the needs of key workloads:
  - Most workloads involve irregular memory access patterns and complex data structures
  - MP-Linpack has **high computational intensity** (lots of work per memory reference). Real work loads have modest amounts of work per memory reference.
  - Even problems that use Dense Linear algebra don't often work with matrices the sizes of those used in top-500 benchmarks (e.g., N = 24,330,240 for the number one machine on the list).

Outline

- Key concepts in computer architecture
- A brief history of supercomputing
- Programming languages and choice overload: Python, C/C++, Fortran

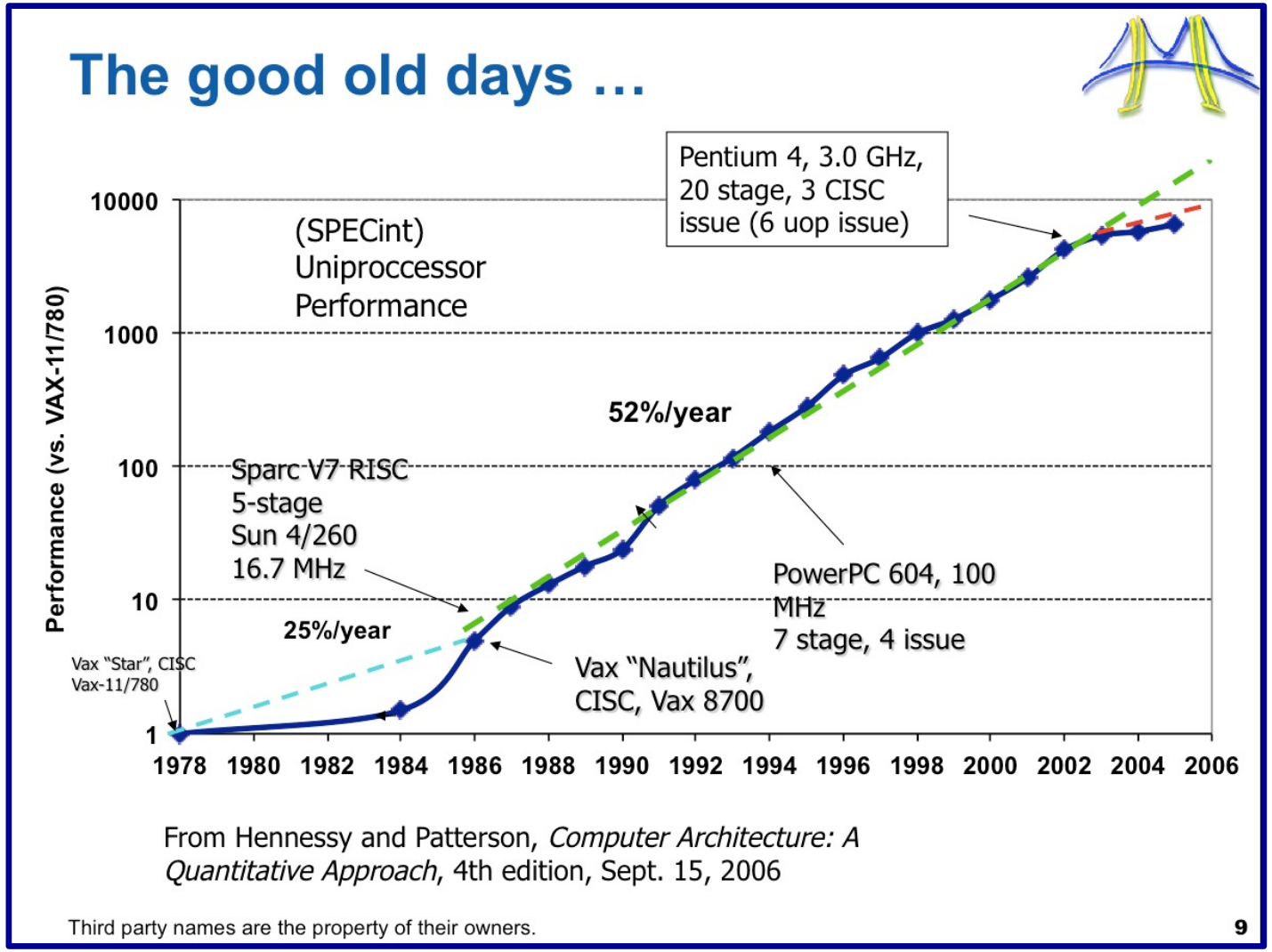
The good old days …



From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Sept. 15, 2006

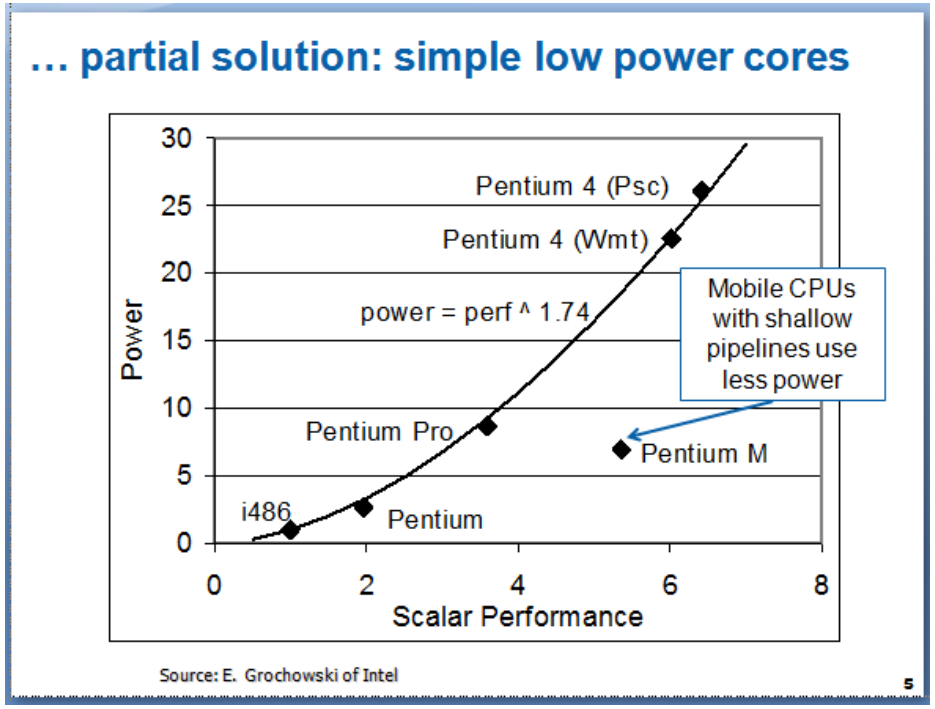
The *old* Hardware/Software contract

- Write your software as you choose and the HW-geniuses will take care of performance.

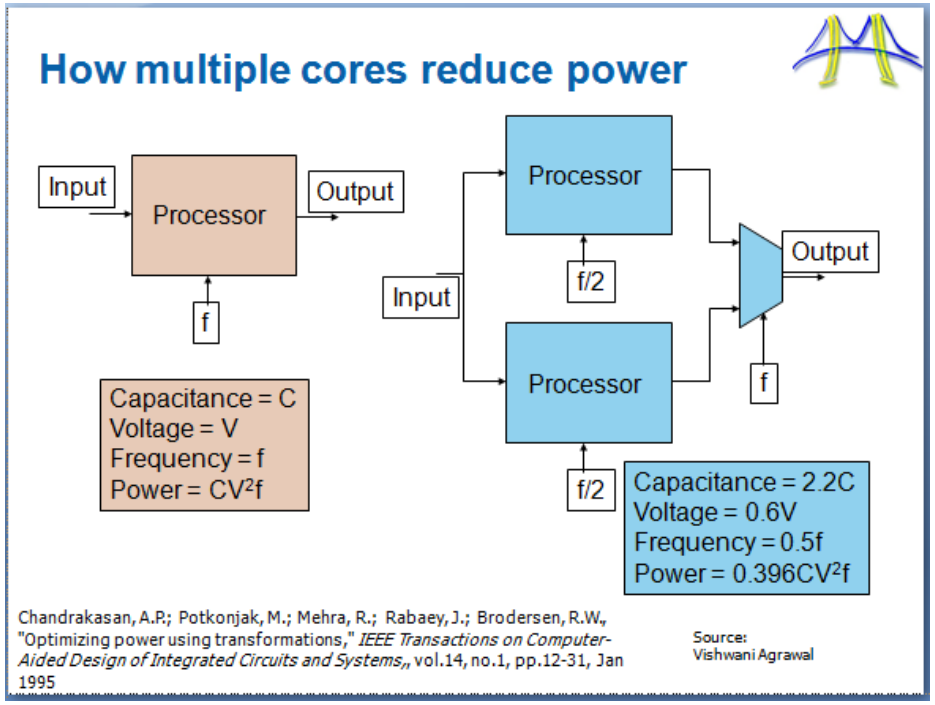


- The result: Generations of performance ignorant software engineers using performance-handicapped languages (such as Java) … which was OK since performance was a HW job.

The *new* Hardware/Software contract



+



= A new HW/SW contract … HW people will do what's natural for them (lots of cores) and optimization is up to SW people who will have to adapt (rewrite everything)

The problem is this was presented as an ultimatum … nobody asked us if we were OK with this new contract … which is kind of rude.

We tried to solve the programmability problem by searching for the right programming environment

Parallel programming environments in the 90's

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
HPF
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

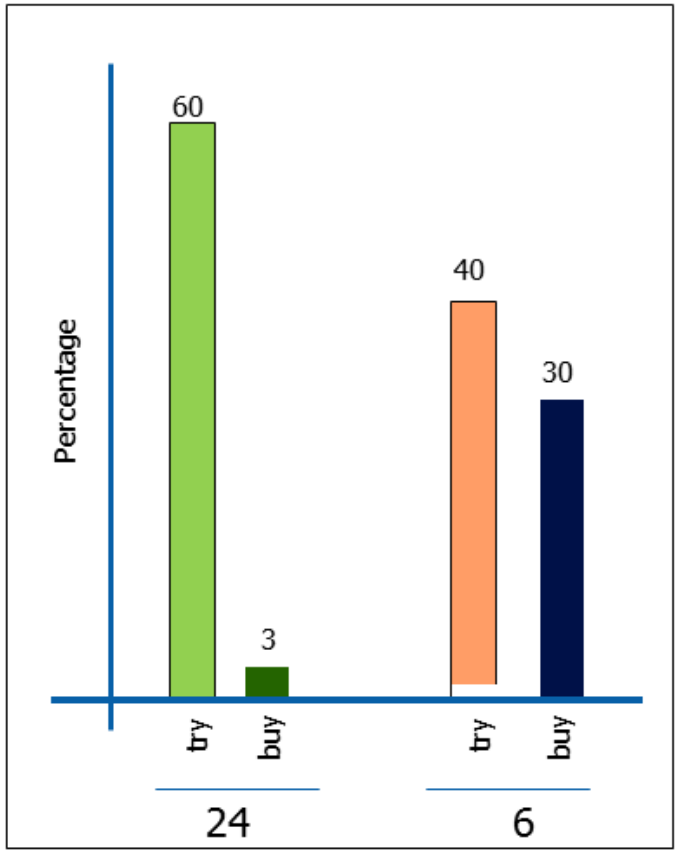
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

A warning I've been making for the last 20 years

Language obsessions: More isn't always better

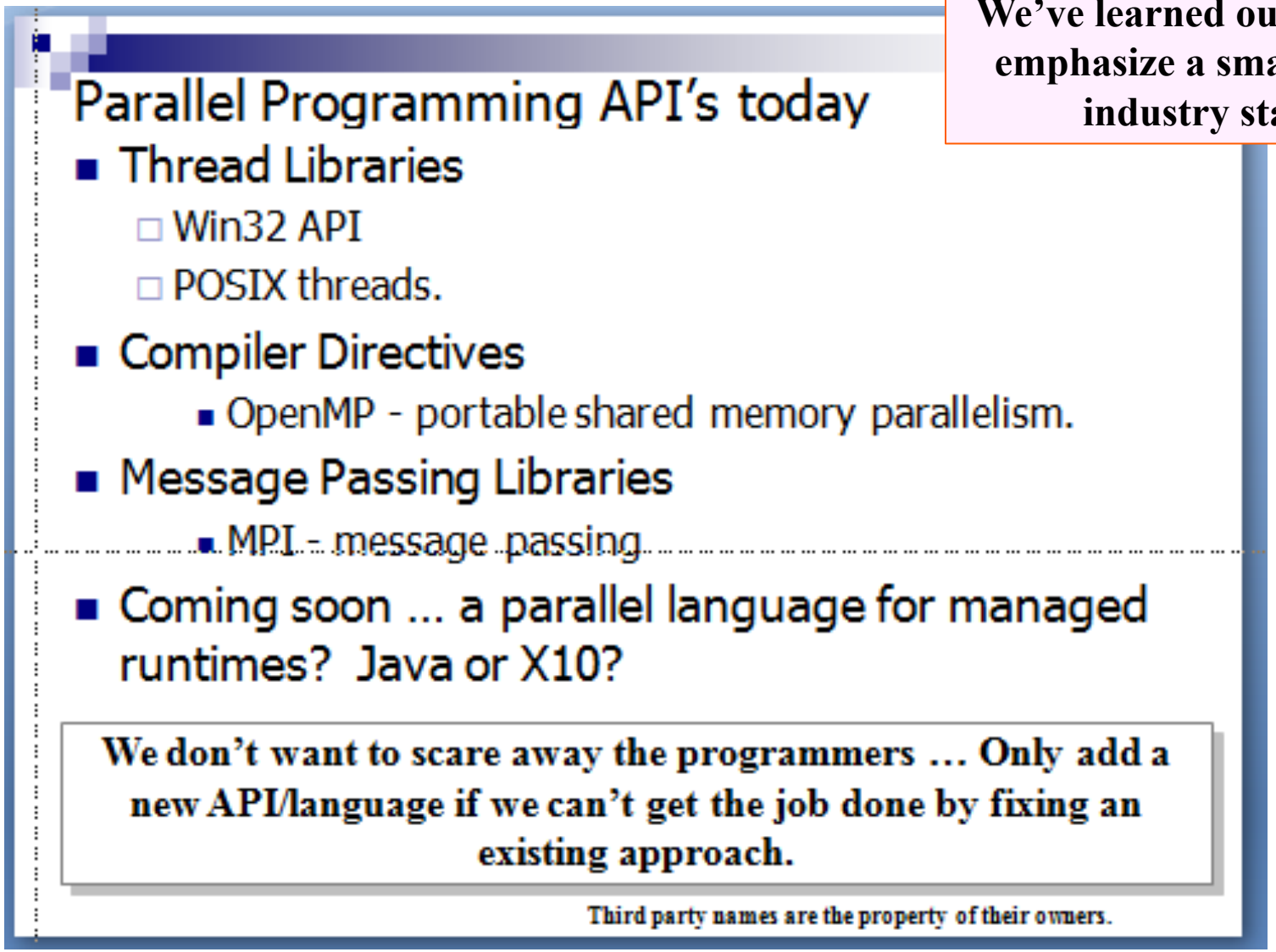
- The Draeger Grocery Store experiment consumer choice :
  - Two Jam-displays with coupon's for purchase discount.
    - 24 different Jam's
    - 6 different Jam's
  - How many stopped by to try samples at the display?
  - Of those who "tried", how many bought jam?



The findings from this study show that an extensive array of options can at first seem highly appealing to consumers, yet can reduce their subsequent motivation to purchase the product.

Iyengar, Sheena S., & Lepper, Mark (2000). When choice is demotivating: Can one desire too much of a good thing? *Journal of Personality and Social Psychology*, 76, 995-1006.

My optimistic view from 2005 …



We've learned our lesson … we emphasize a small number of industry standards

While those of us close to parallel architectures knew about GPUs for HPC computing, even as late as 2005 none of us (outside Nvidia) saw it coming as a mainstream HPC technology.

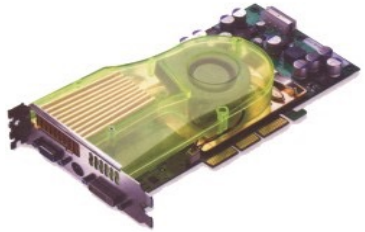
But CUDA hit us in 2006 and changed everything.

The emergence of “general purpose” computing on a GPU (GPGPU)

1st generation:
Voodoo 3dfx (1996)

2nd Generation:
GeForce 256/Radeon 7500 (1998)

3rd Generation:
GeForce3/Radeon 8500 (2001). The first GPU to allow a limited programmability in the vertex pipeline.



4th Generation: Radeon 9700/GeForce FX (2002): The first generation of “fully-programmable” graphics cards.

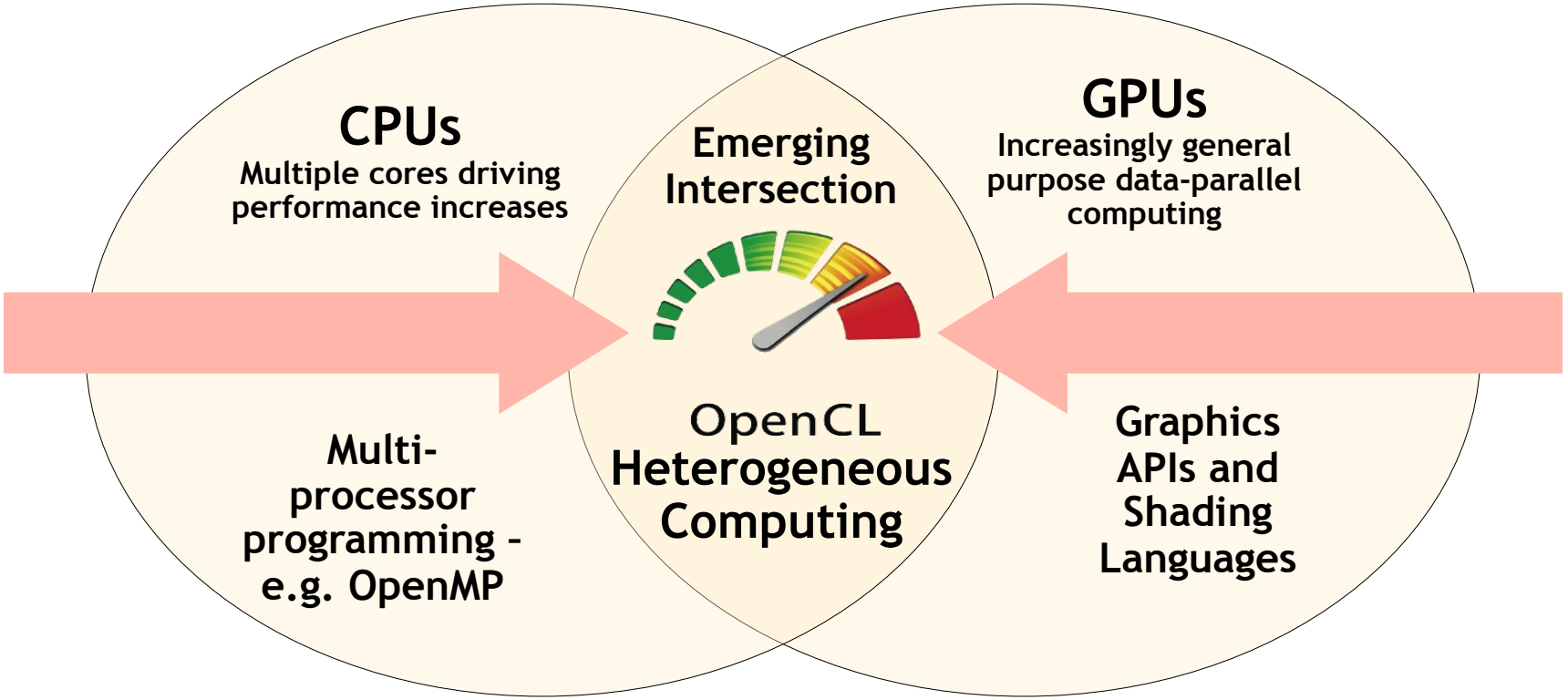
GPGPU arrives: 2006

- GeForce 8800/HD2900:
  - Ground-up GPU redesign
  - Support for Direct3D 10
  - Geometry Shaders
  - Stream out / transform-feedback
  - Unified shader processors
- ***Support for General GPU programming***

- Fortunately for NVIDIA, the academic community had been working on GPGPU programming for almost a decade by the time the GeForce 8800 came along.
  - Ian Buck in his Stanford PhD dissertation, "Stream computing on Graphics Hardware", described a language called "Brook".
  - He moved over to NVIDIA and led the effort to create CUDA.

CUDA has been extremely influential … With inspiration from CUDA, Late in 2008 Apple, AMD, Intel, NVIDIA, Imagination Technologies and several other companies released a vendor-neutral, portable standard for stream computing called OpenCL.

OpenCL – Open Compute Language



OpenCL has been very successful in the Gaming industry. It is very heavily used ... But not in HPC

Vendors didn't stand behind the specification and users let them "off the hook" by "selling their souls" to Nvidia and basing their work on CUDA® (an Nvidia proprietary technology).

The major parallel Programming systems in 2024 …

well at least we have our act together in two cases. ☹

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)
- Even if you don't plan to spend much time programming with these systems … a well rounded HPC programmer must know MPI, OpenMP, and at least one GPU language.

That wraps things up for now

Let’s talk about our next steps for the course

Starting with our next lecture, we will shift to hands on learning

- People learn by doing. The next time we get together we'll talk about our transition to hands-on learning.

- This means learning about Linux, the C programming language, working from a command line prompt and more.

- To prepare I want each of you to make sure your laptop has an actual Gnu compiler installed by our next time together.

- For information on how to accomplish this … see these useful guides, the first in English and the second covering the same material in Italian.
  https://github.com/giacomini/ebernburg2024/tree/main/guides
  https://github.com/Programmazione-per-la-Fisica/howto/tree/main/other-OSes

Note: these come from Francesco Giacomini at the University of Bologna. His course emphasized C++. We will focus on C. And he advises VSCode. That's OK, but I suggest vi (or vim) instead.