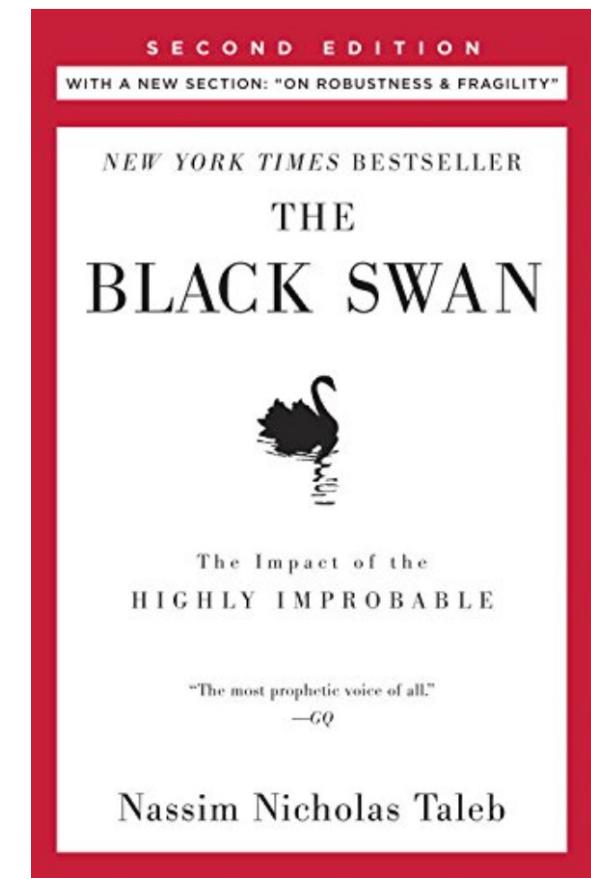
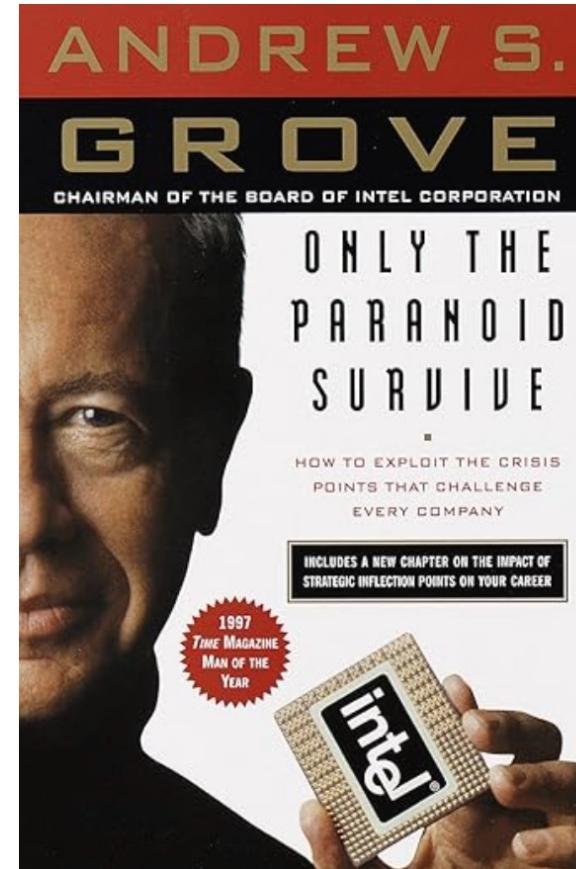
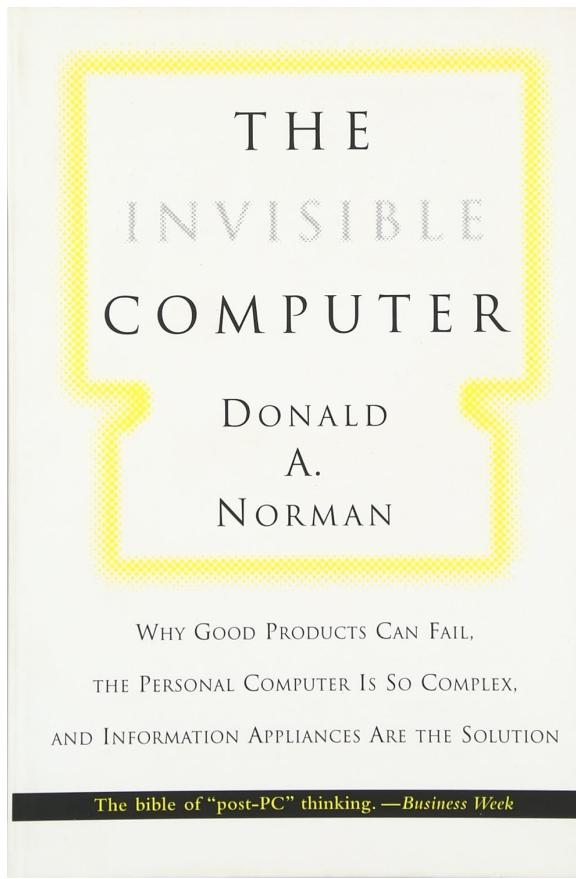
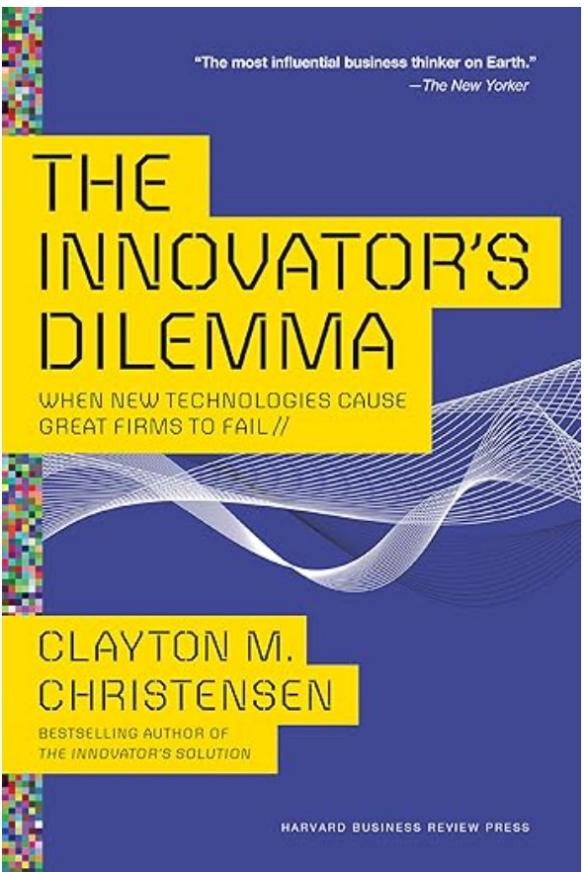


# **Understanding computers and how their design impacts performance**

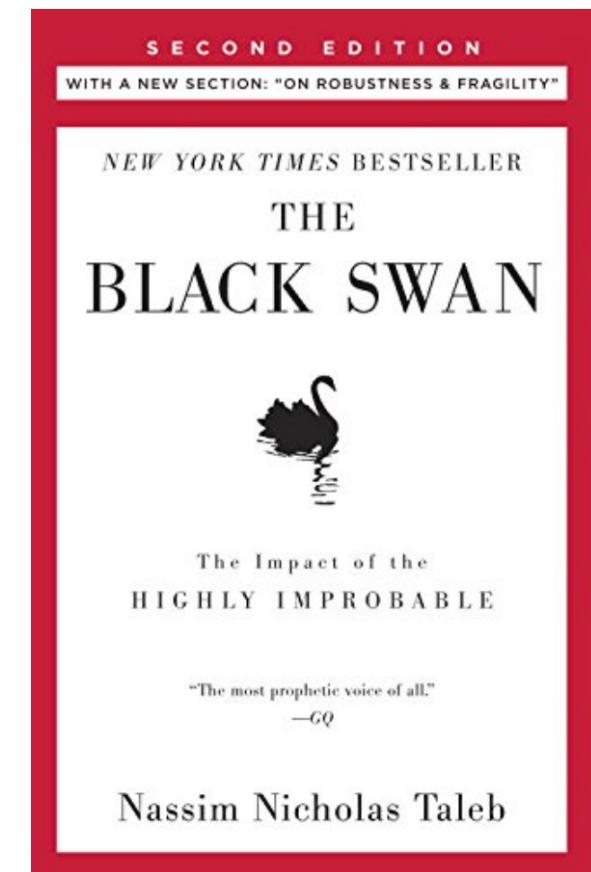
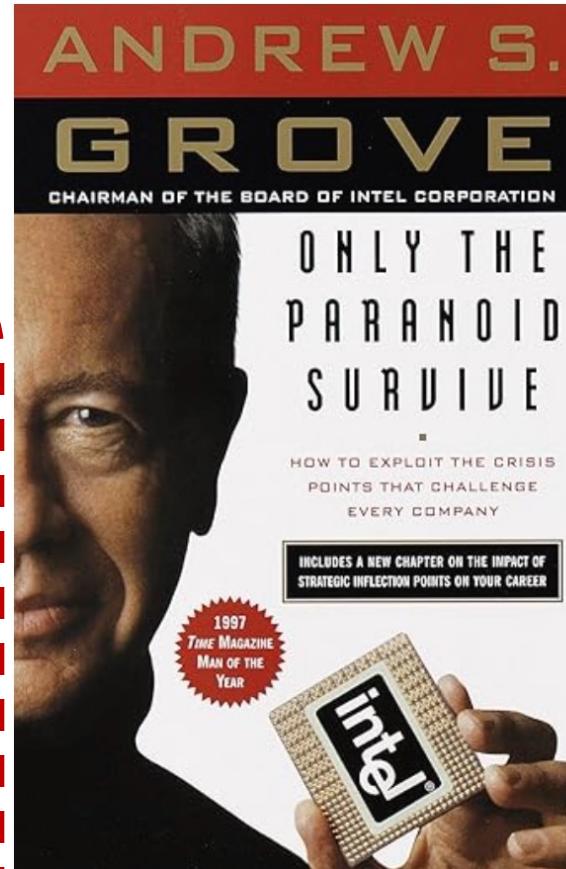
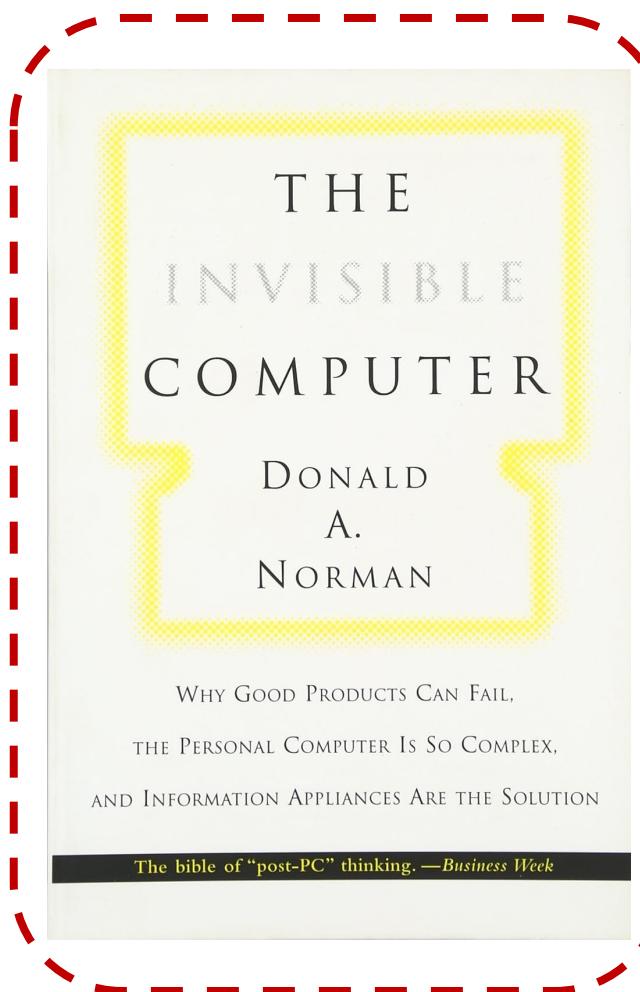
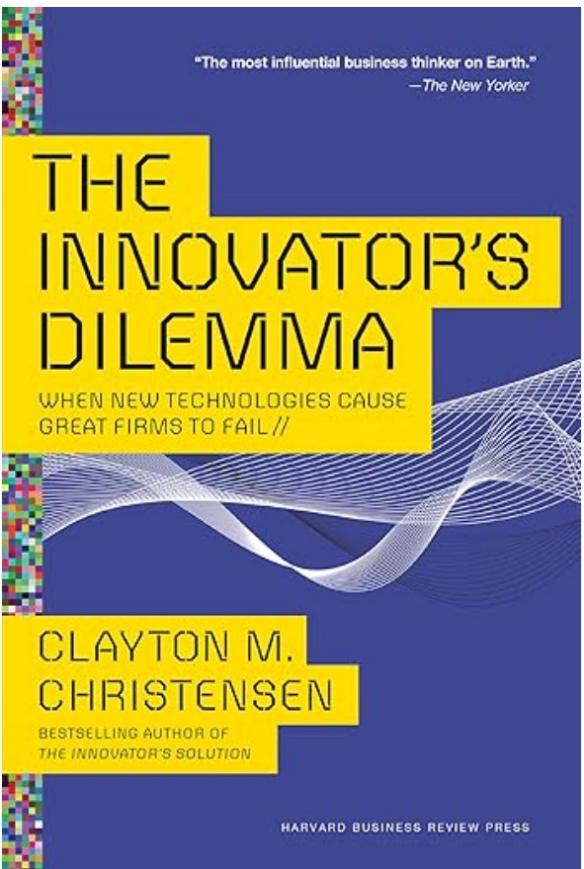
**Tim Mattson**

**Human Learning Group**

# 4 books to make sense of the computer industry

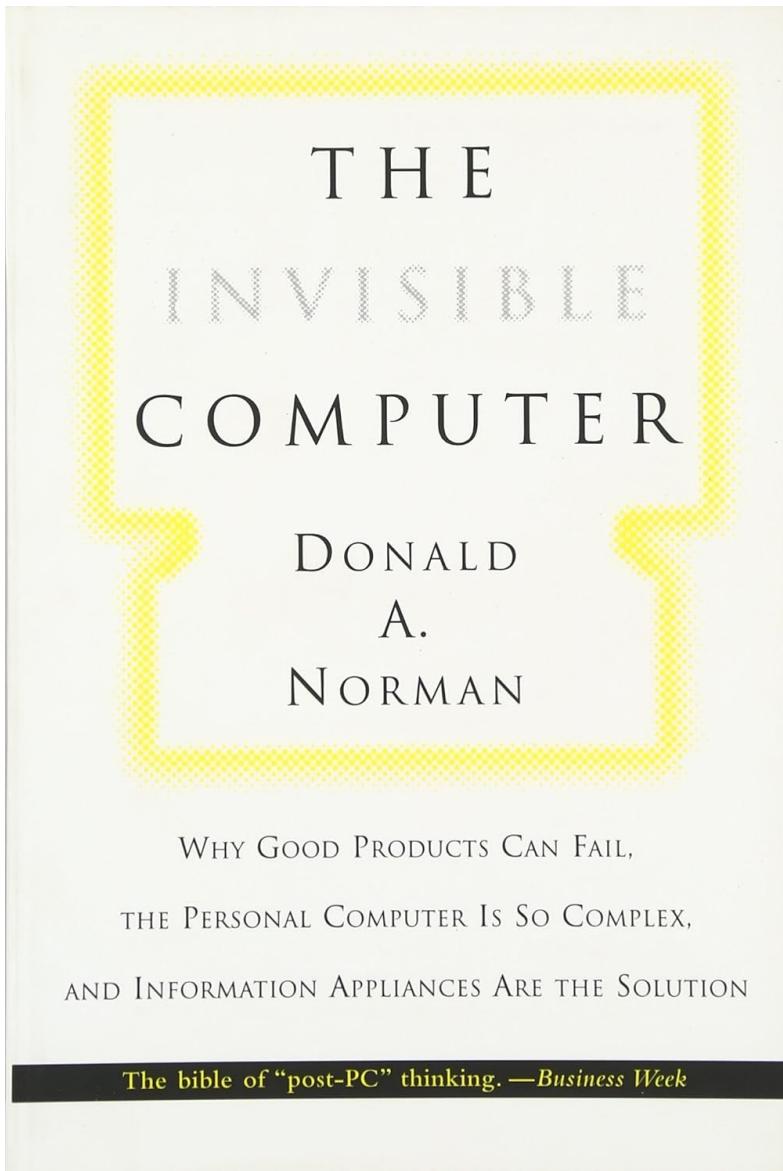


# 4 books to make sense of the computer industry



They are all good  
and worth reading,  
but we'll focus on  
this one for now

# Computing for Humans



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

Joined 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.

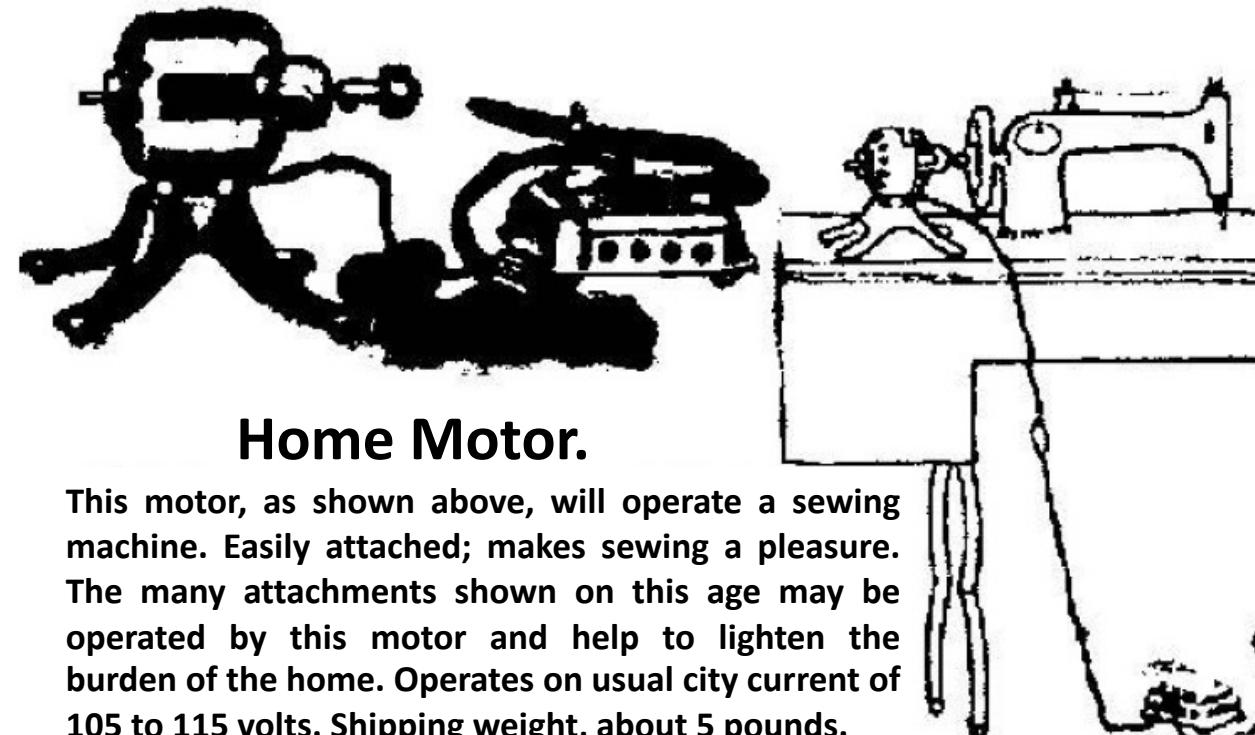
**How many electric motors have you purchased in the last few years?**

# 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 ... **motors 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 57F764. 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

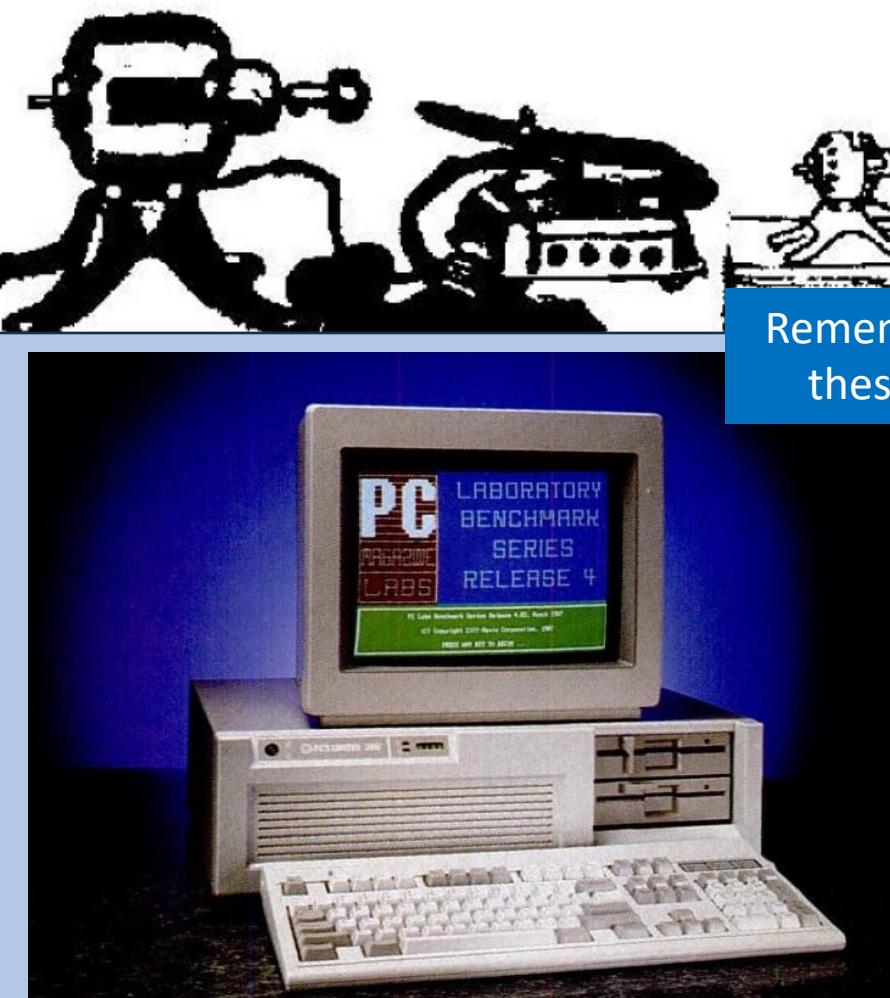
**Churn and Mixer Attachment.**  
Used in connection with the Home Motor, makes a small churn and mixer for which you will find many uses. The attachments include the base, supports, mixer, handle and special cover for jar. Shipping weight, about 3½ pounds.  
No. 57F7582 Price..... \$1.30

**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

About \$100 in today's dollars.

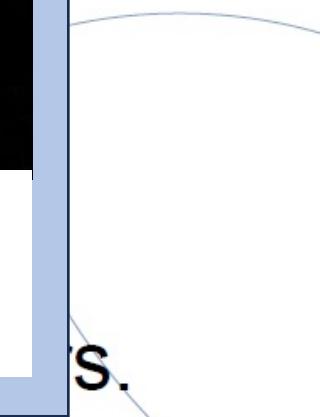
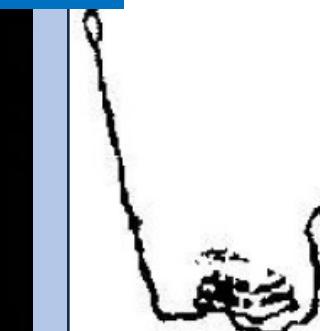
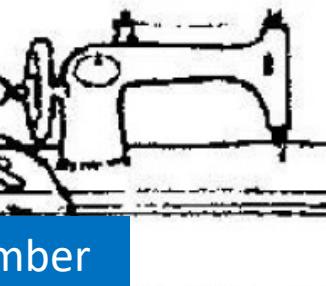
# An example from the book “The Invisible Computer”

The PC is going the way electric motors.

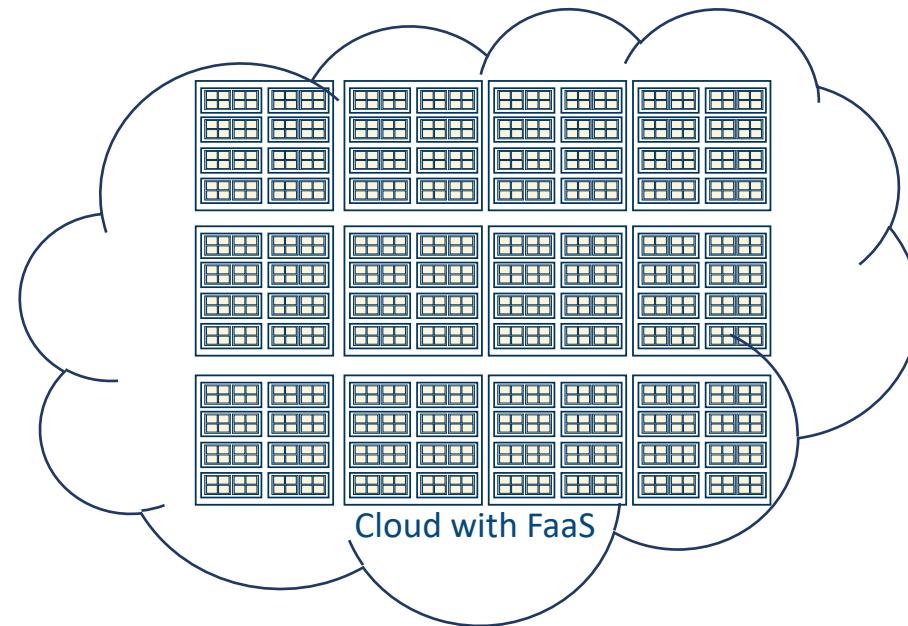


Dell Limited 386-16 PC (1987). 16 MHz Intel 80386 CPU, 1 MB RAM, 1.21 MB Floppy, and 40 MB hard drive. \$4,799 (\$13,574 in 2024 dollars).  
<https://www.pcmag.com/news/the-golden-age-of-dell-computers>

Remember  
these?



# Invisible computers



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



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



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.



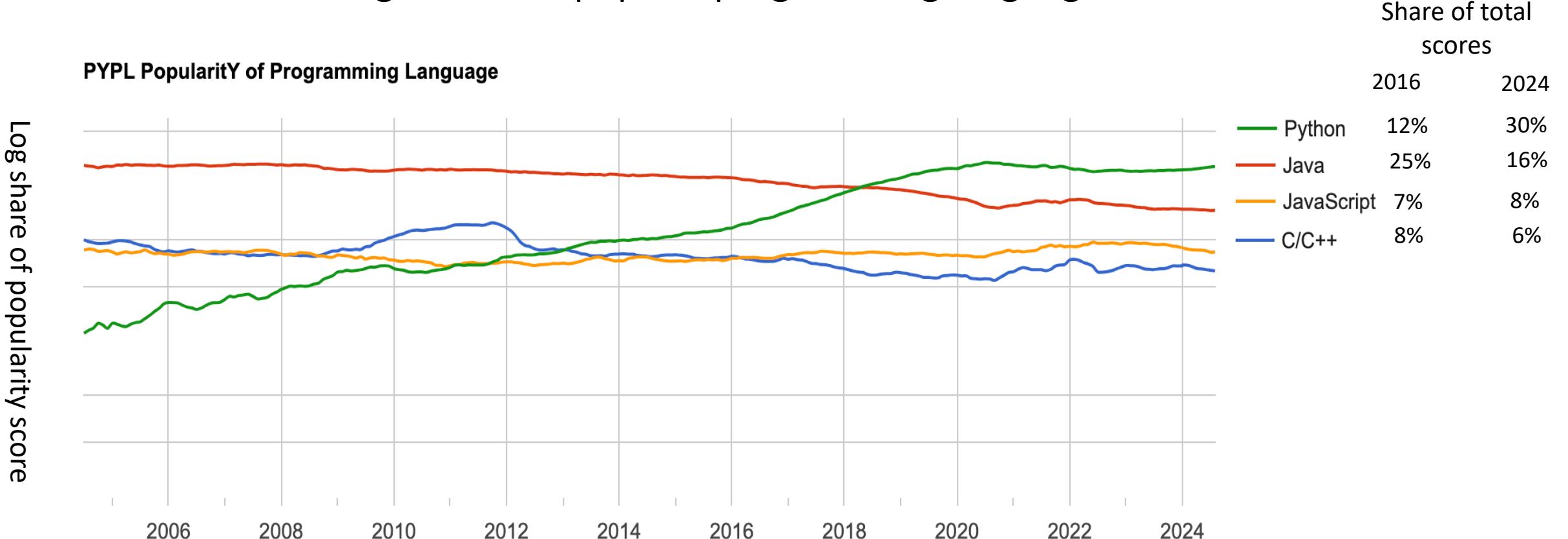
Roku streaming stick 4K, ARM Cortex A55, 1GB RAM  
<https://developer.roku.com/docs/specs/hardware.md>



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/>

# Python is the language of invisible computing

Consider the changes in most popular programming languages...

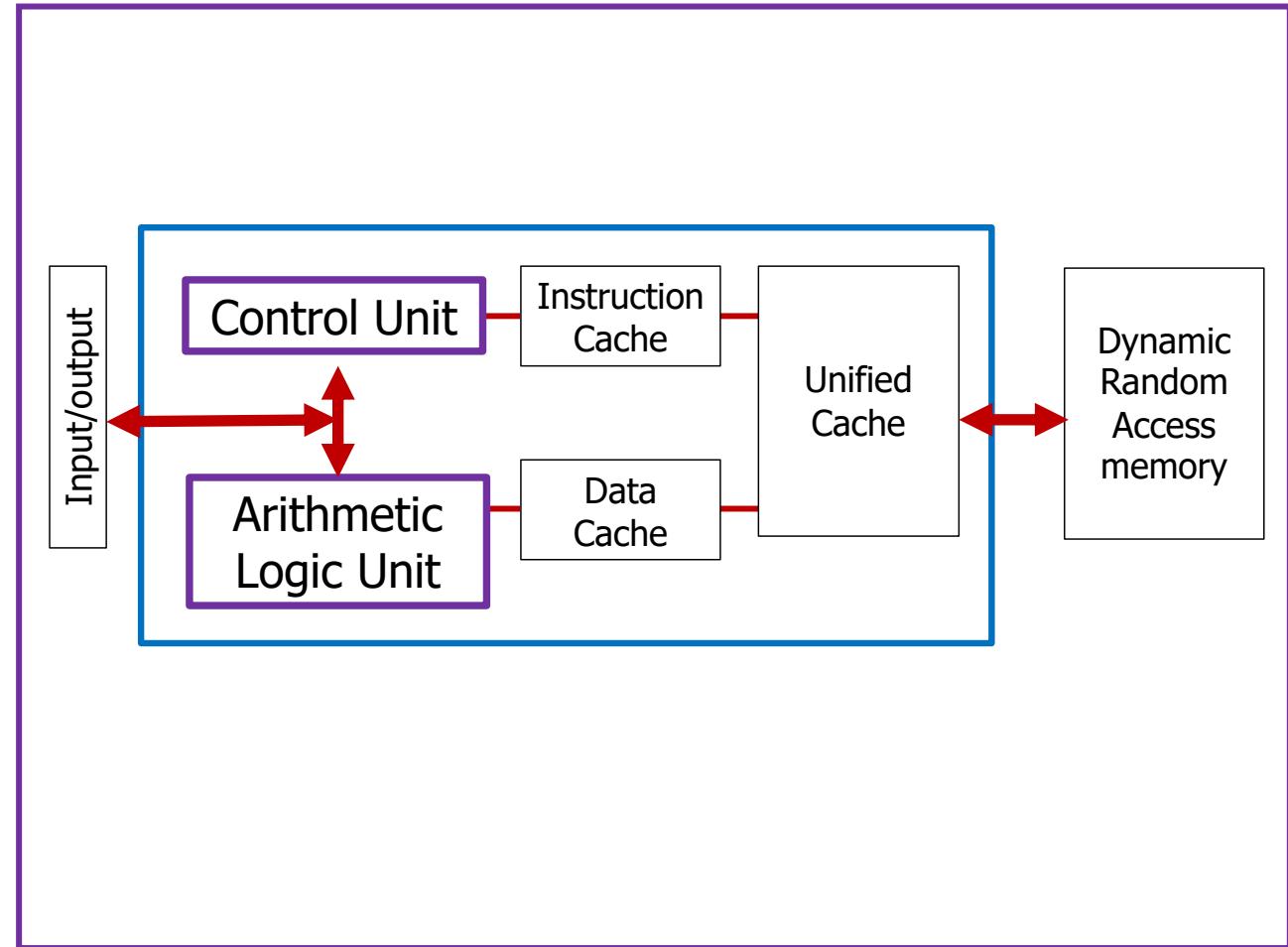


# Invisible computers and scientific computing

- 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 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.

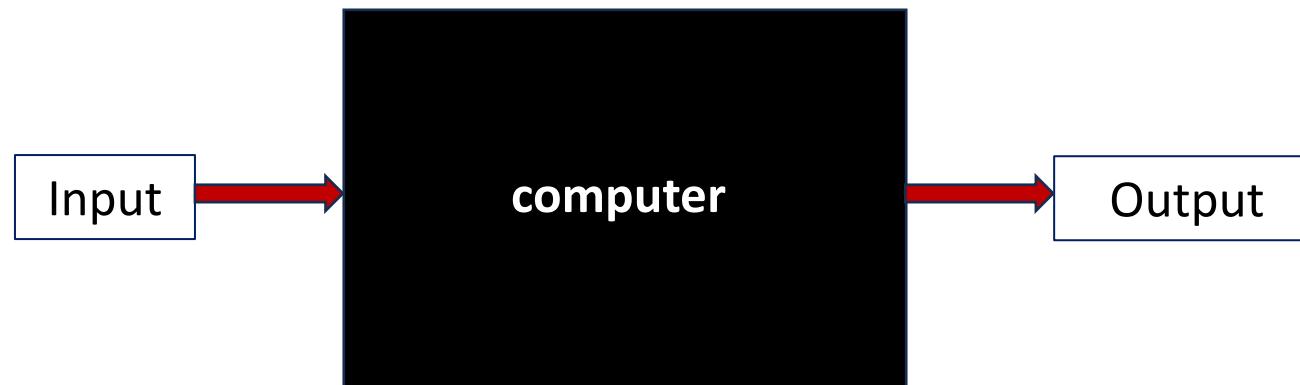


**Let's go back to basics ....**

**What is a computer?**

# What is a computer:

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



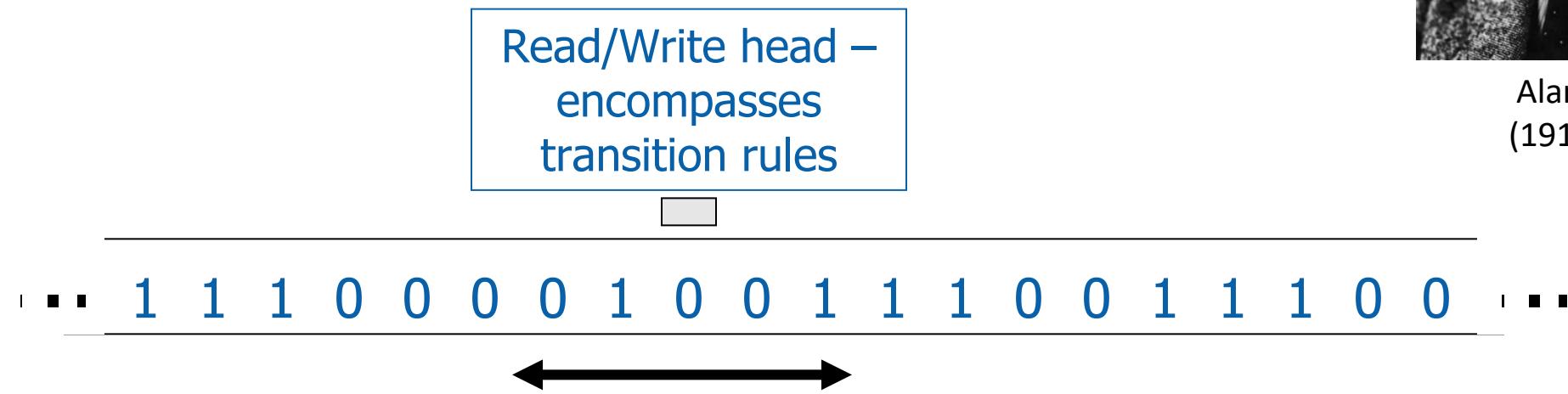
- The computer as a black-box is not very helpful. We need a bit more detail.

# Computer models: Turing Machine

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



Alan Turing  
(1912-1954)



- 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

- 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).

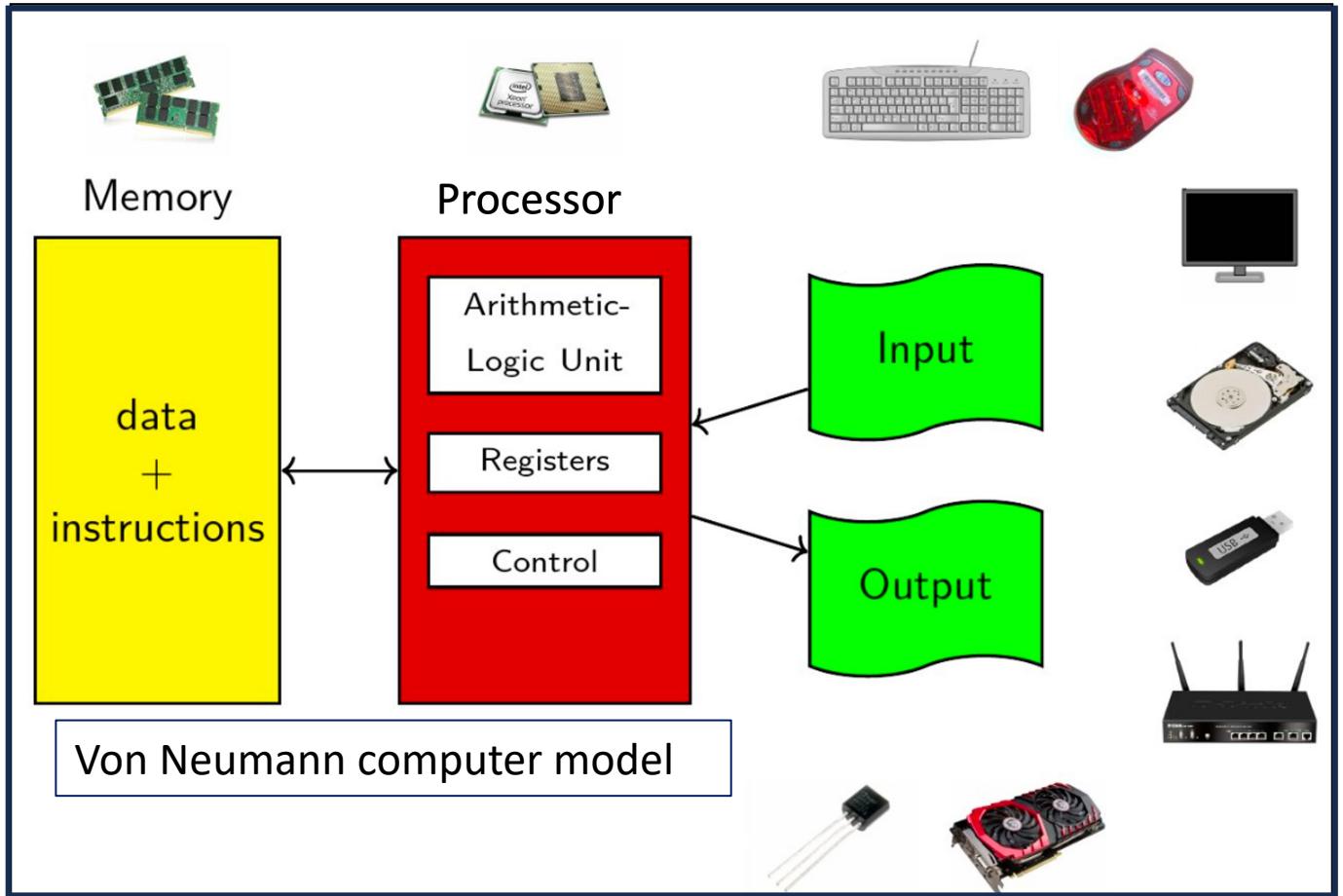
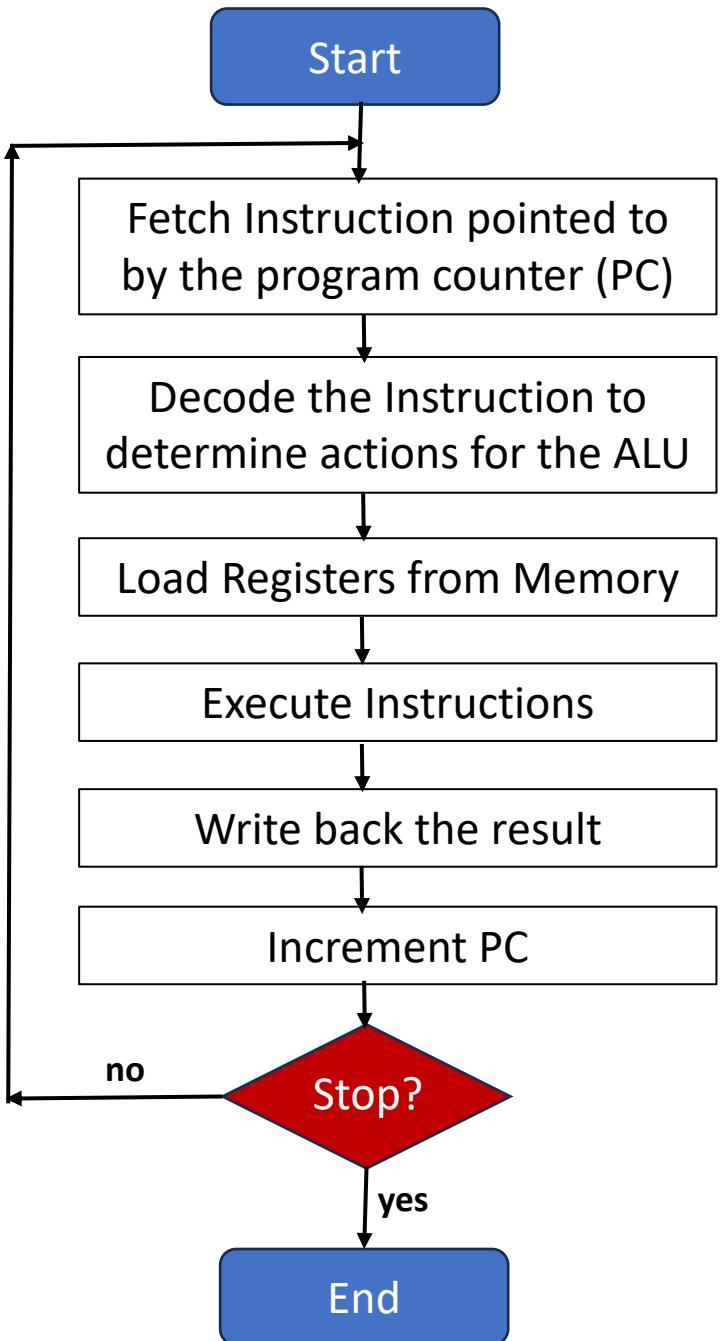


Image Source: Felice Pantaleo, CERN, ESC'23



# Von Neumann or a “stored Program” Model

- 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).

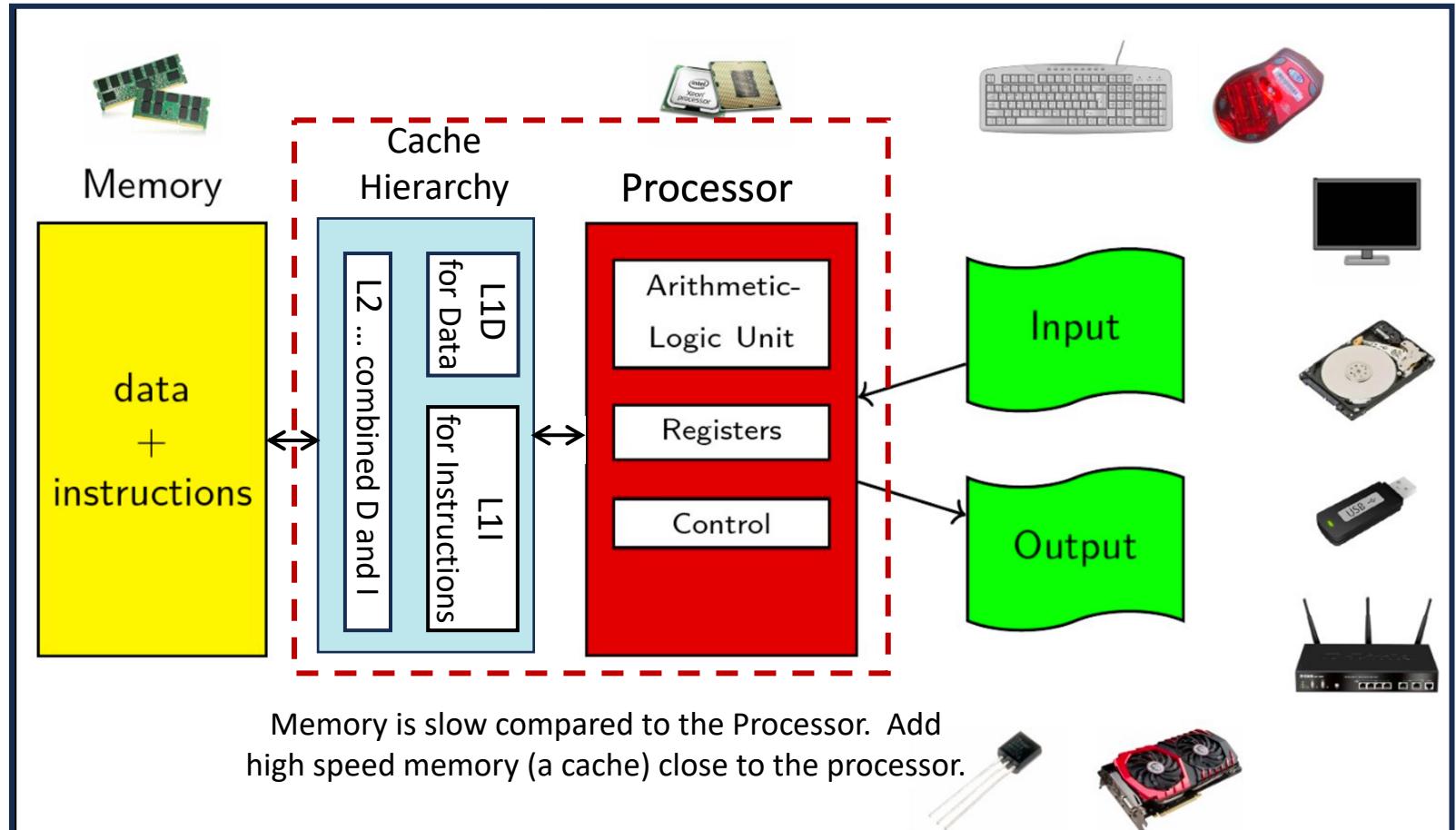
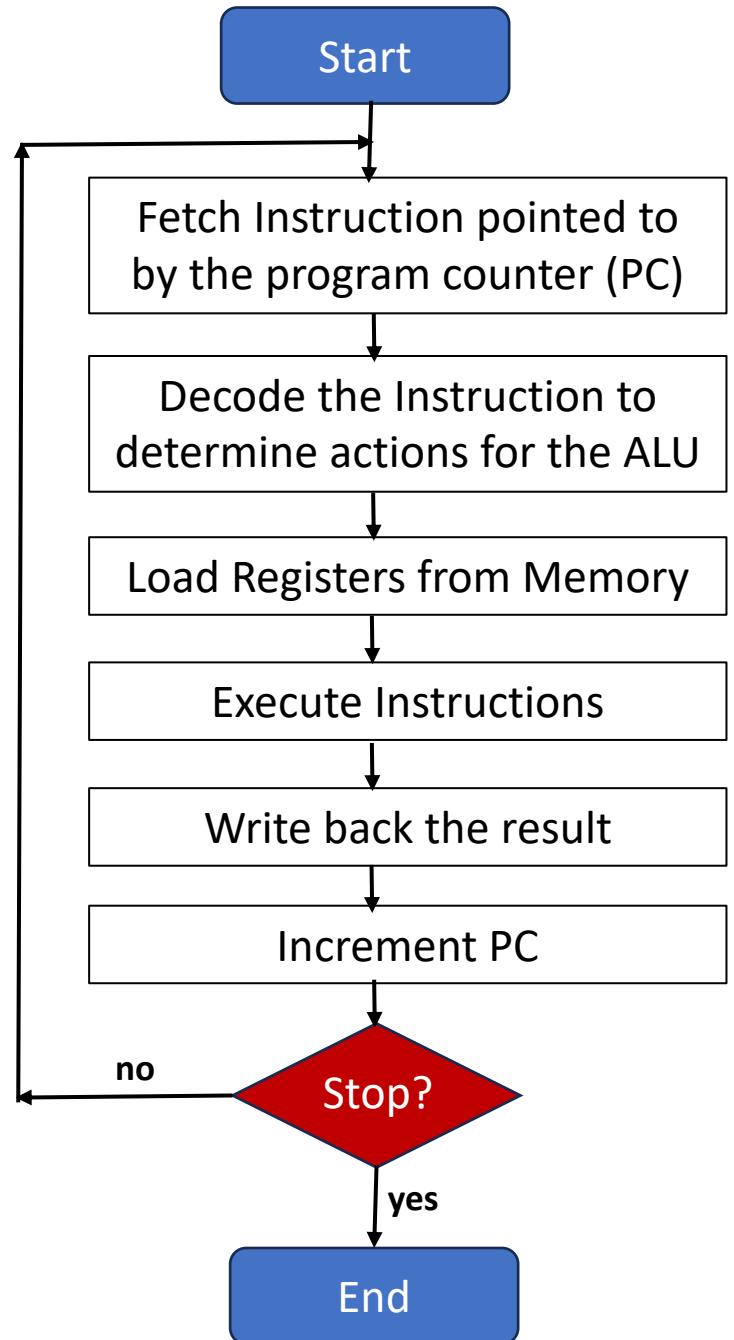


Image Source: Felice Pantaleo, CERN, ESC'23



**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

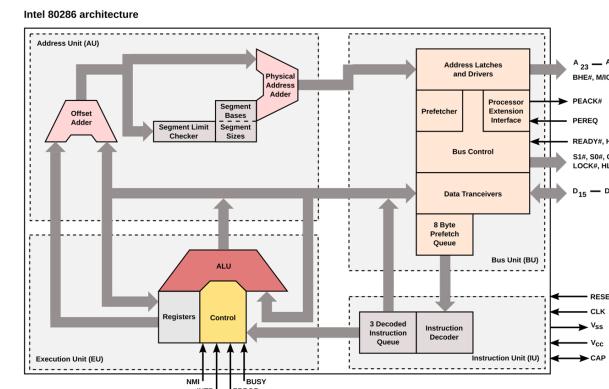
- Computer architecture is composed of 3 topics

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

```
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     fmuls.s 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     fmuls.s fa0,fa0,fa4  
27     ret
```

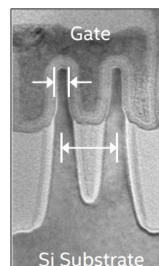
RISC-V assembly code for the “pi program loop” generated by gcc -O3

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



Intel 80286 microarchitecture

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



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

# Instruction Set Architecture (ISA)

- Instruction set architecture (ISA): **the interface to the hardware presented to a programmer**
- Two major classes of ISA
  - **CISC**: Complex instruction set. Large set of instructions to cover numerous special cases. Example: Intel® x86 ISA
  - **RISC**: Reduced instruction set. 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 (A) of size s bytes is aligned if $\&A \bmod s = 0$ .
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.

# 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



<https://godbolt.org/>

CISC  
Ops work on registers and addresses in memory.

Complex but extra options for aggressive optimization

x86-64 gcc 14.2

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

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

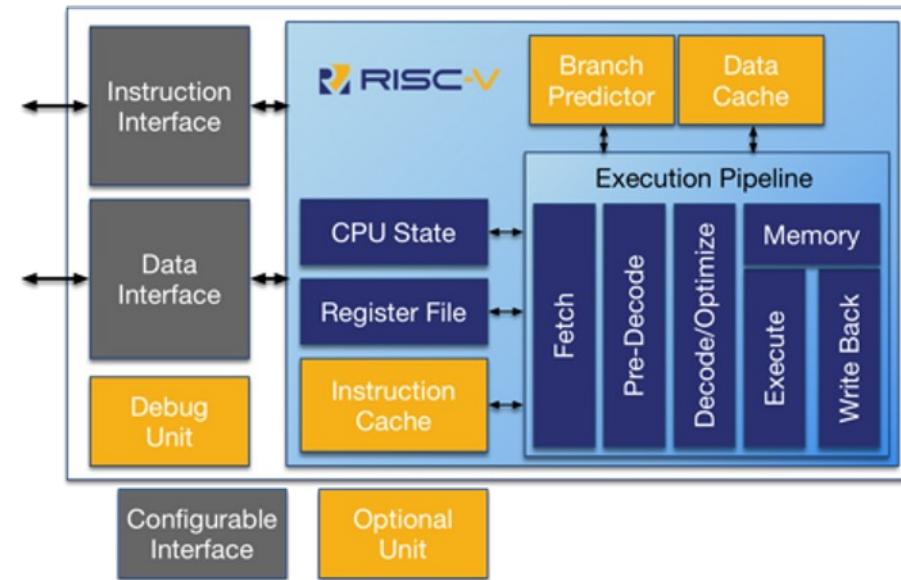
- Intel pioneered processors for mass-market-computing through the IBM PC.
- Intel's x86 ISA is a CISC instruction set ... it played a key role in the history of computing.
  - Starting with the Intel 8086 CPU in 1978 and continuing to today, it is the dominant architecture for servers and laptops.
- 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 its 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.

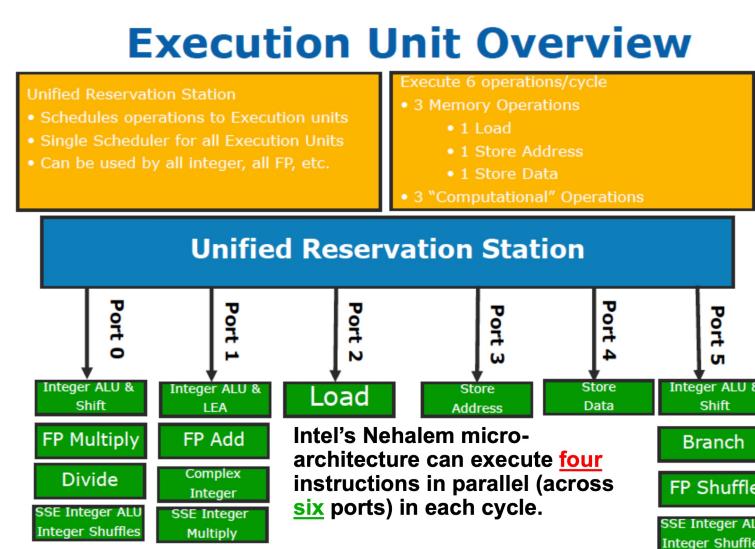
RISC-V block diagram



- 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

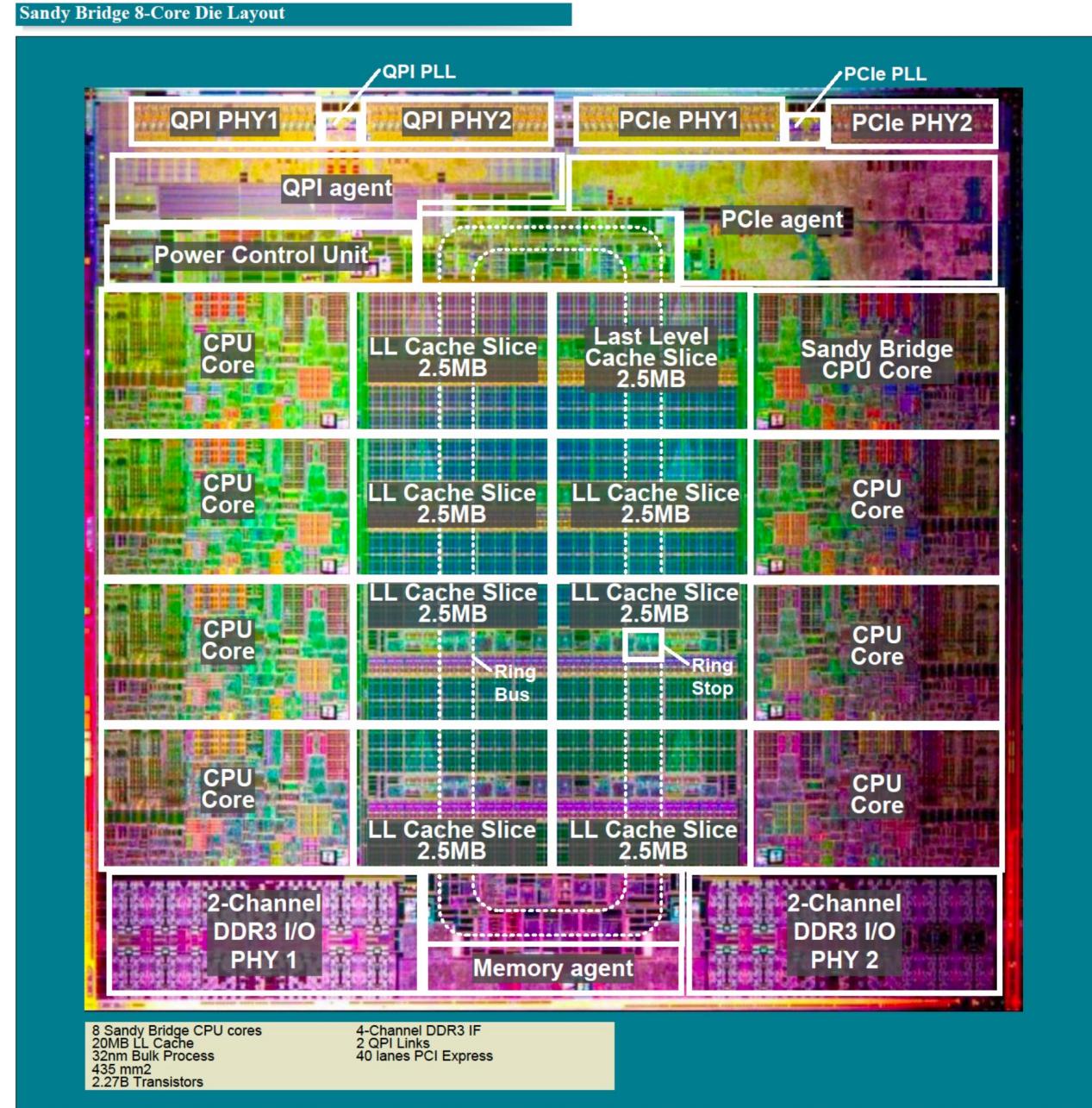


# A modern CPU

## Intel® X86 architecture

### Sandy Bridge Microarchitecture

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

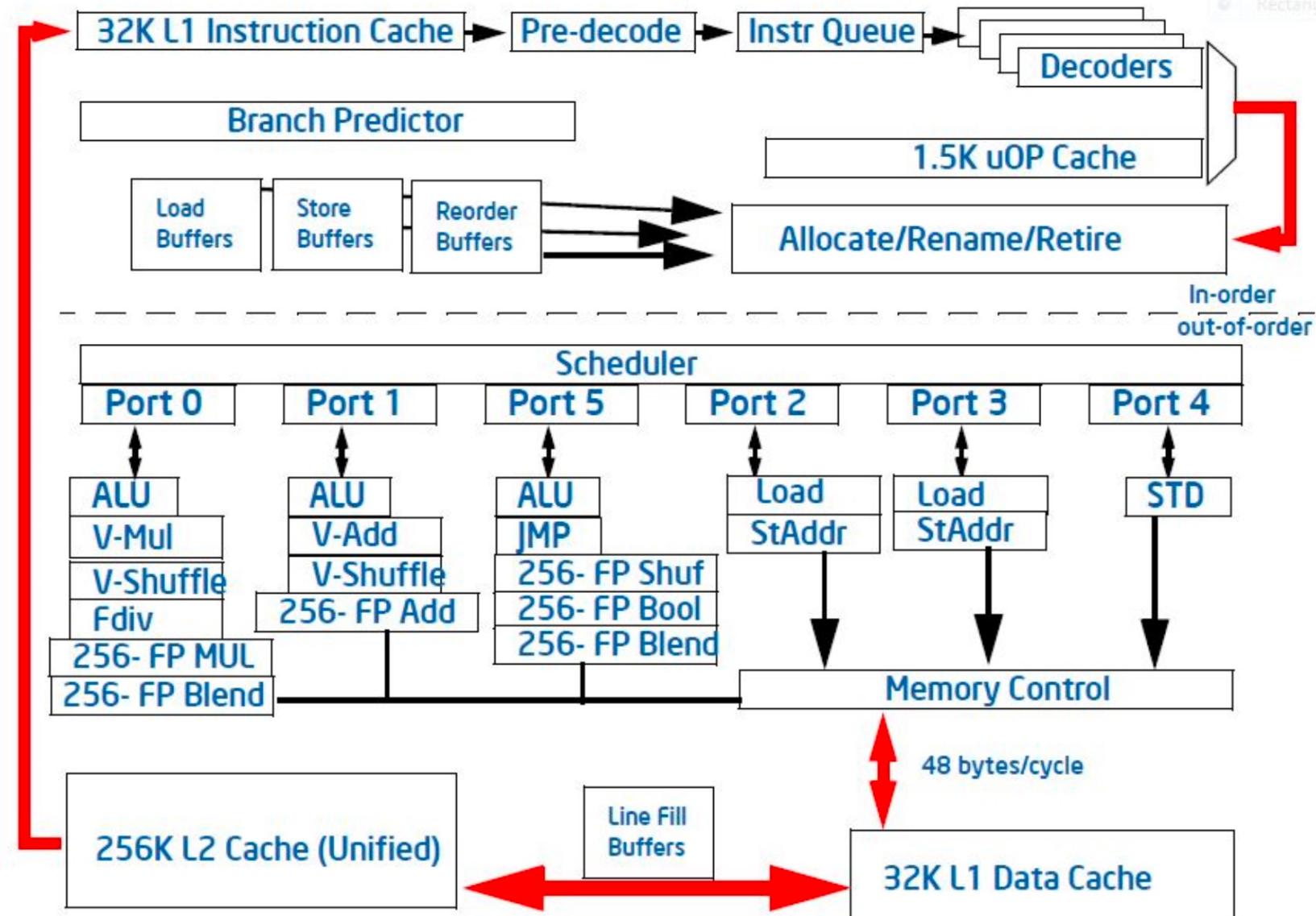


# Structure of a Sandy Bridge CPU core

- A microarchitecture supports 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

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

- 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 look at evolution of Instruction level parallelism through the lens of the history of x86 CPUs**

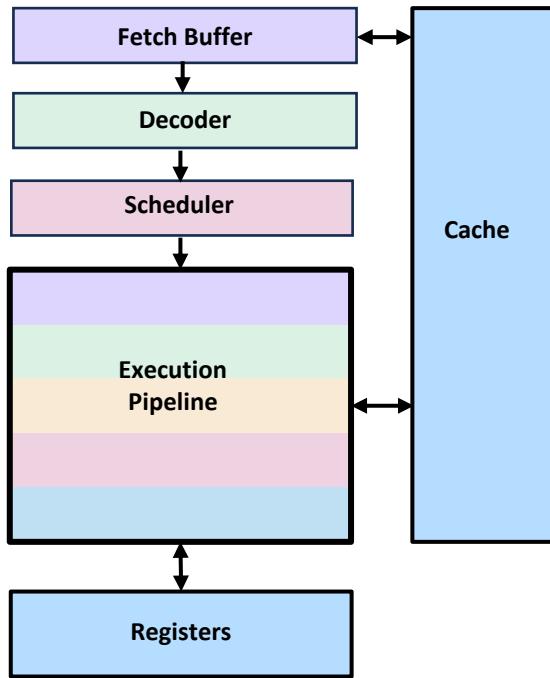
“X86 ... an architecture that is difficult to explain and impossible to love”

Hennessy and Patterson, 2<sup>nd</sup> ed, page D-2

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

# Pipelining

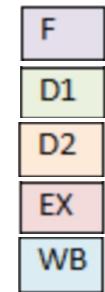
i486™ 1.2 Million transistors, 50 Mhz



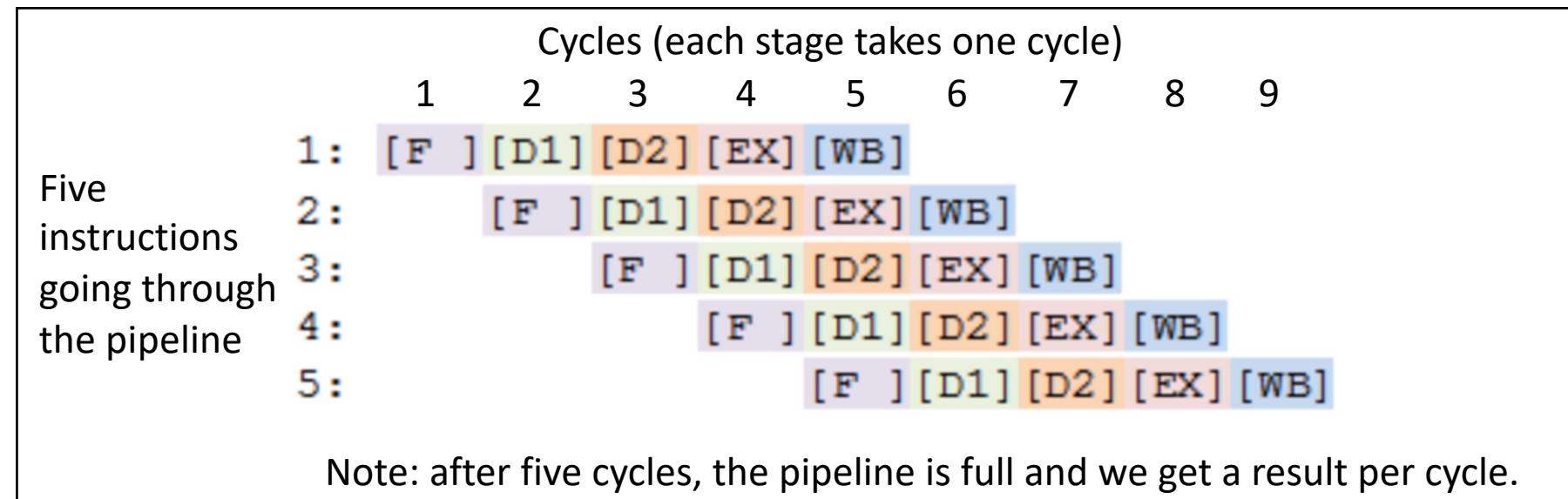
... plus an integrated floating point unit

- 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

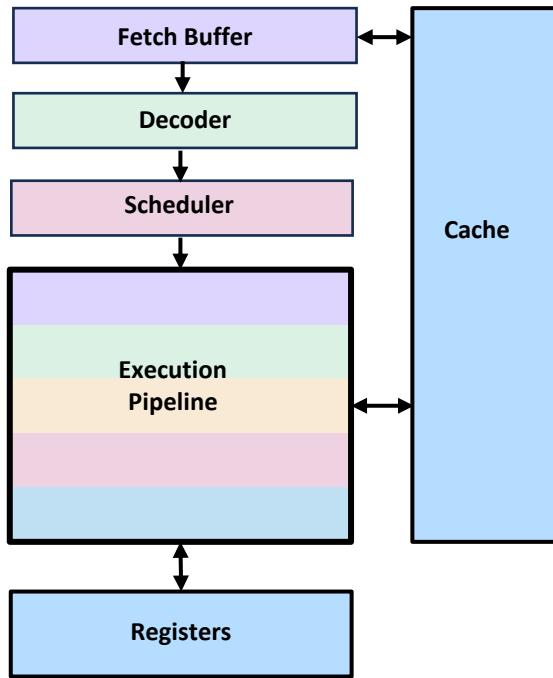


- Fetch an instruction from the instruction cache.
- Decode the instruction.
- Translate memory addresses and displacements for the instruction
- Execute the instruction.
- Retire the instruction, write results back to registers and/or memory.



# Pipelining Performance

i486™ 1.2 Million transistors, 50 Mhz



... plus an integrated floating point unit

- Given the following definitions:

$n = \text{number of operations}$

$\ell = \text{number of stages}$

$\tau = \text{time per stage}$

- Runtime without pipelining:  $t(n) = n\ell\tau$
  - With pipelining you need to setup the pipeline (costs  $s$  cycles) and fill the pipeline (costs  $\ell$  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:
- $$t(n) = (s + \ell + n-1) \tau$$
- For  $n$  much greater than  $(s + \ell - 1)$ ,  $t(n) \approx n \tau$ , so the code runs  $\ell$  times faster.

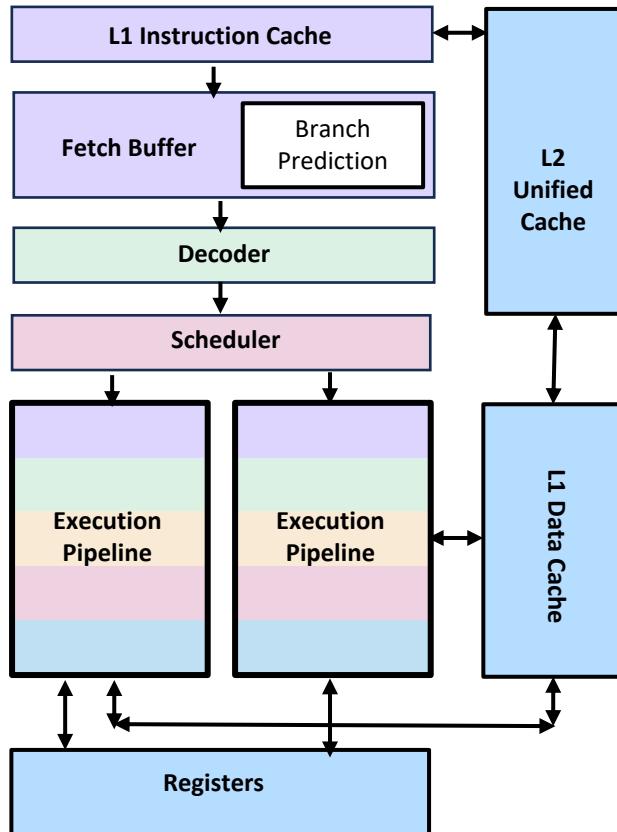
Cycles (assuming each stage takes one cycle)									
	1	2	3	4	6	7	8	9	10
1:	[F]	[D1]	[D2]	[EX]	[WB]				
2:		[F]	[D1]	[D2]	[EX]	[WB]			
3:			[F]	[D1]	[D2]	[EX]	[WB]		
4:				[F]	[D1]	[D2]	[EX]	[WB]	
5:					[F]	[D1]	[D2]	[EX]	[WB]

Five instructions going through the pipeline

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

# Superscalar + branch prediction

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



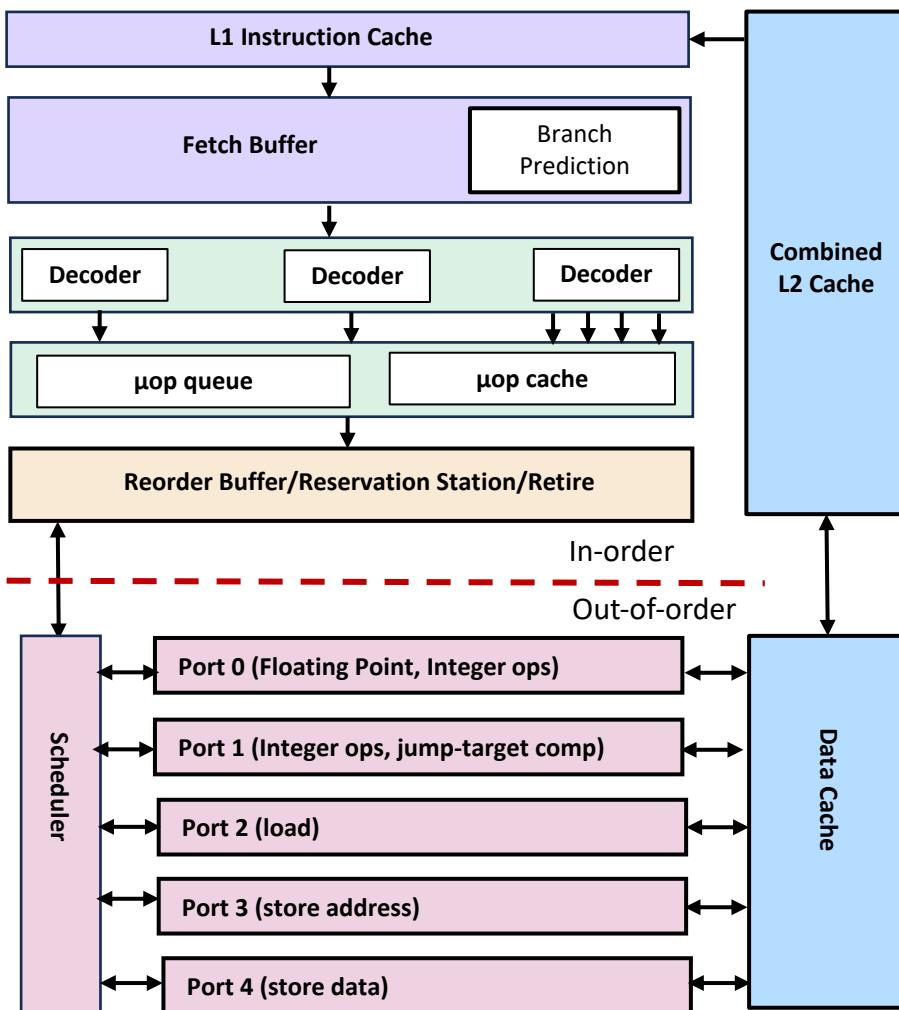
- The Pentium CPU was introduced in 1993 with major innovations
  - **Superscalar execution:** Added a second execution pipeline so it could keep two pipelined instruction streams in flight at the same time.
  - **Branch prediction:** Speculate on branches a program might take to load next instructions 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.

... plus an integrated floating point unit shared between pipelines

# Out of Order (OOO) + speculation

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

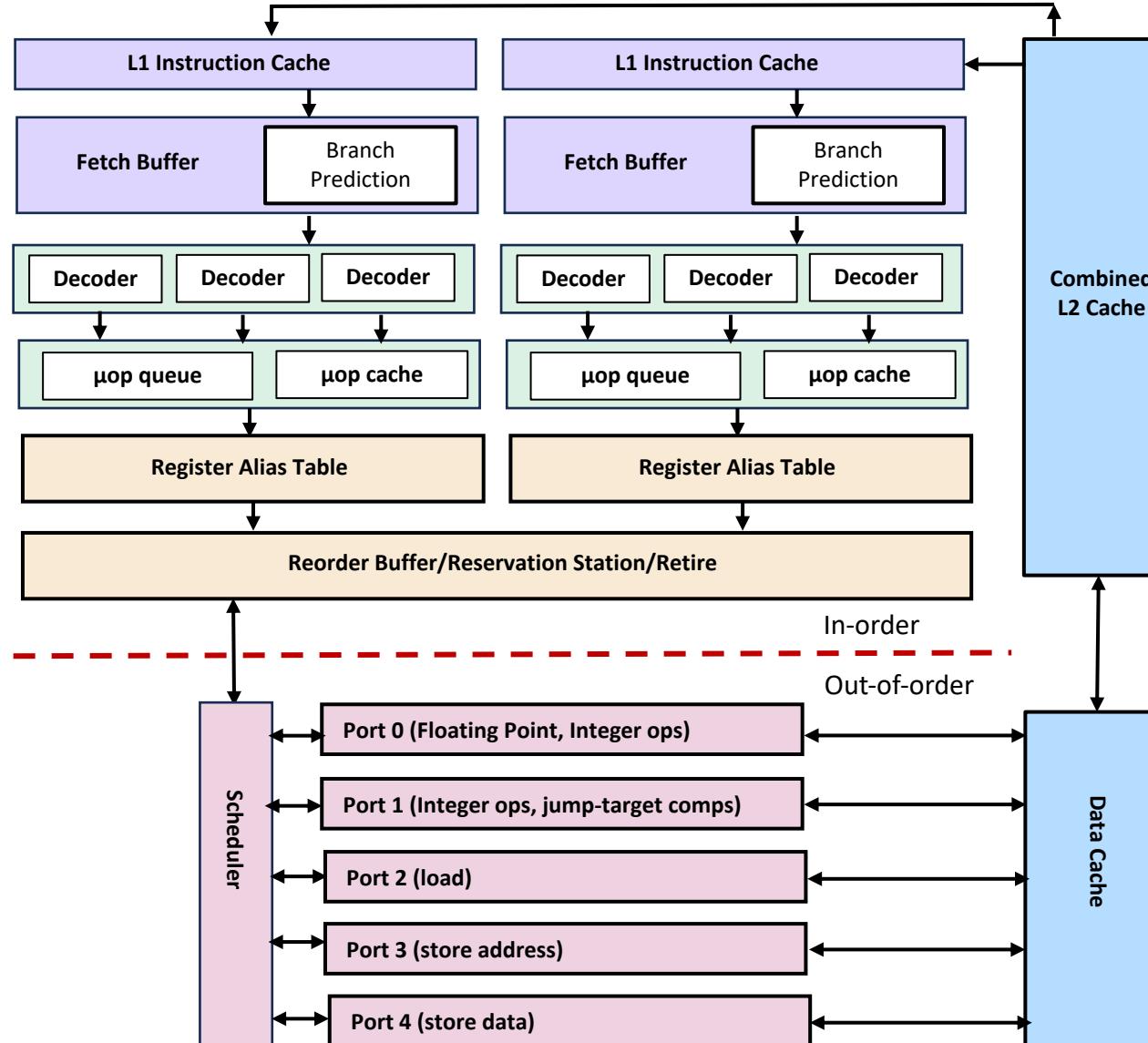


- The Pentium Pro CPU was a major performance upgrade and made Intel CPUs suitable for demanding technical computing workloads (and was used inside the computer ranked as the fastest computer in the world from 1996 to 2000).
- It added the following microarchitectural innovations:
  - Input CISC instructions were decoded into fixed-length, load/store micro-ops ( $\mu$ op). 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**

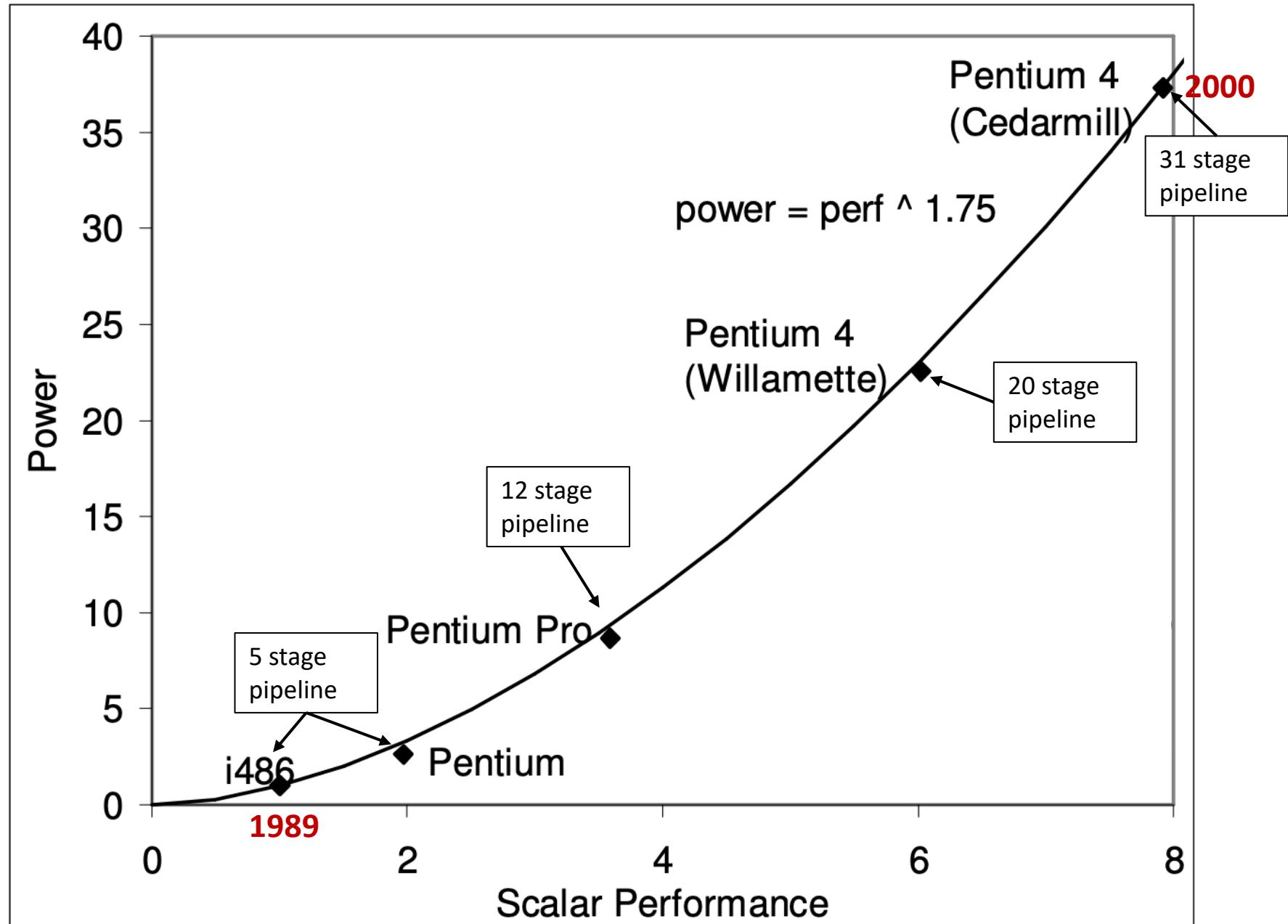
# Normalized Power vs. scalar performance for Intel CPUs

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., Pentium 4 is 6 times faster than an i486 but uses 22 times more power.



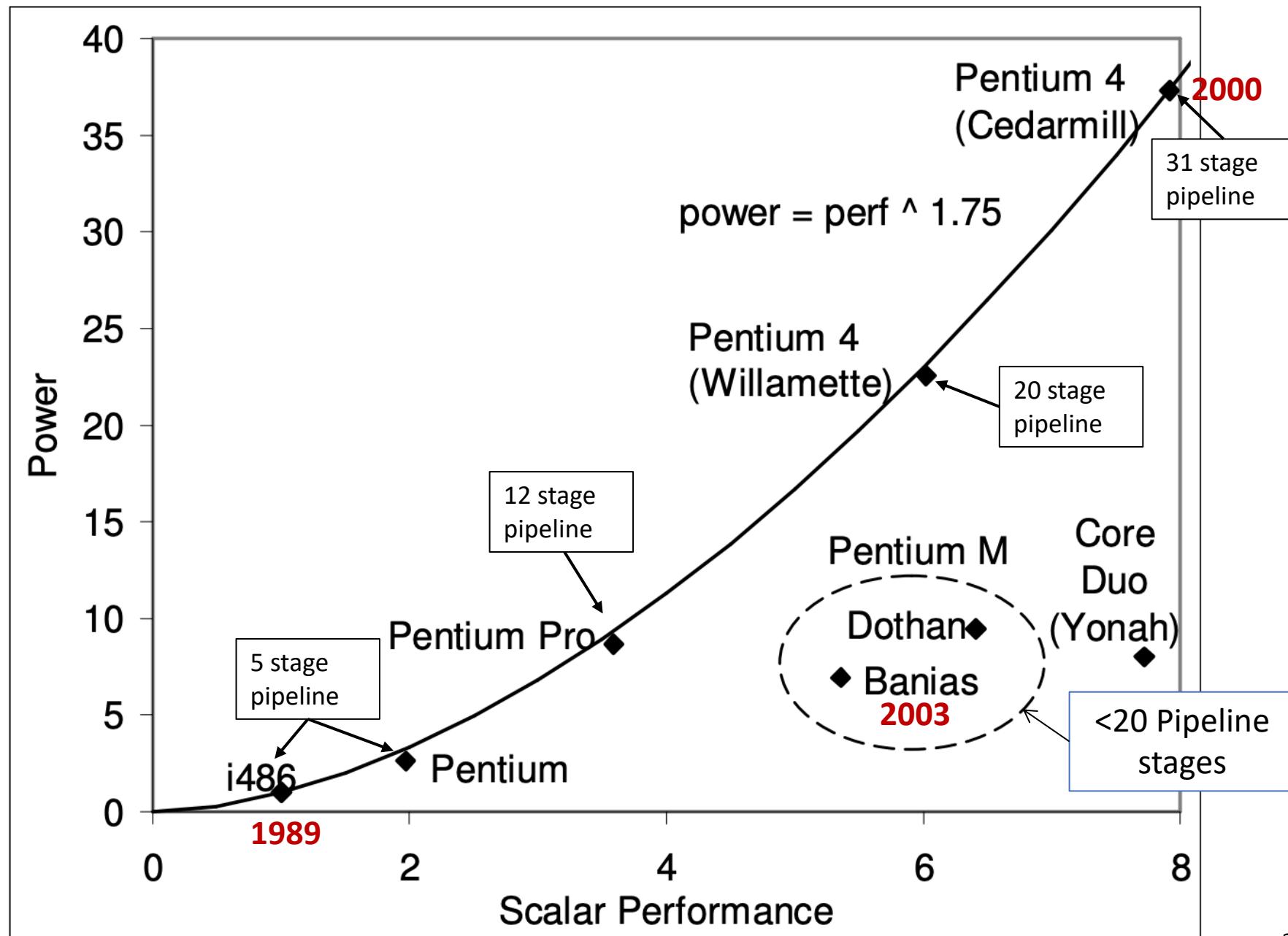
# Normalized Power vs. scalar performance for Intel CPUs

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

Any changes are due to microarchitectural enhancements

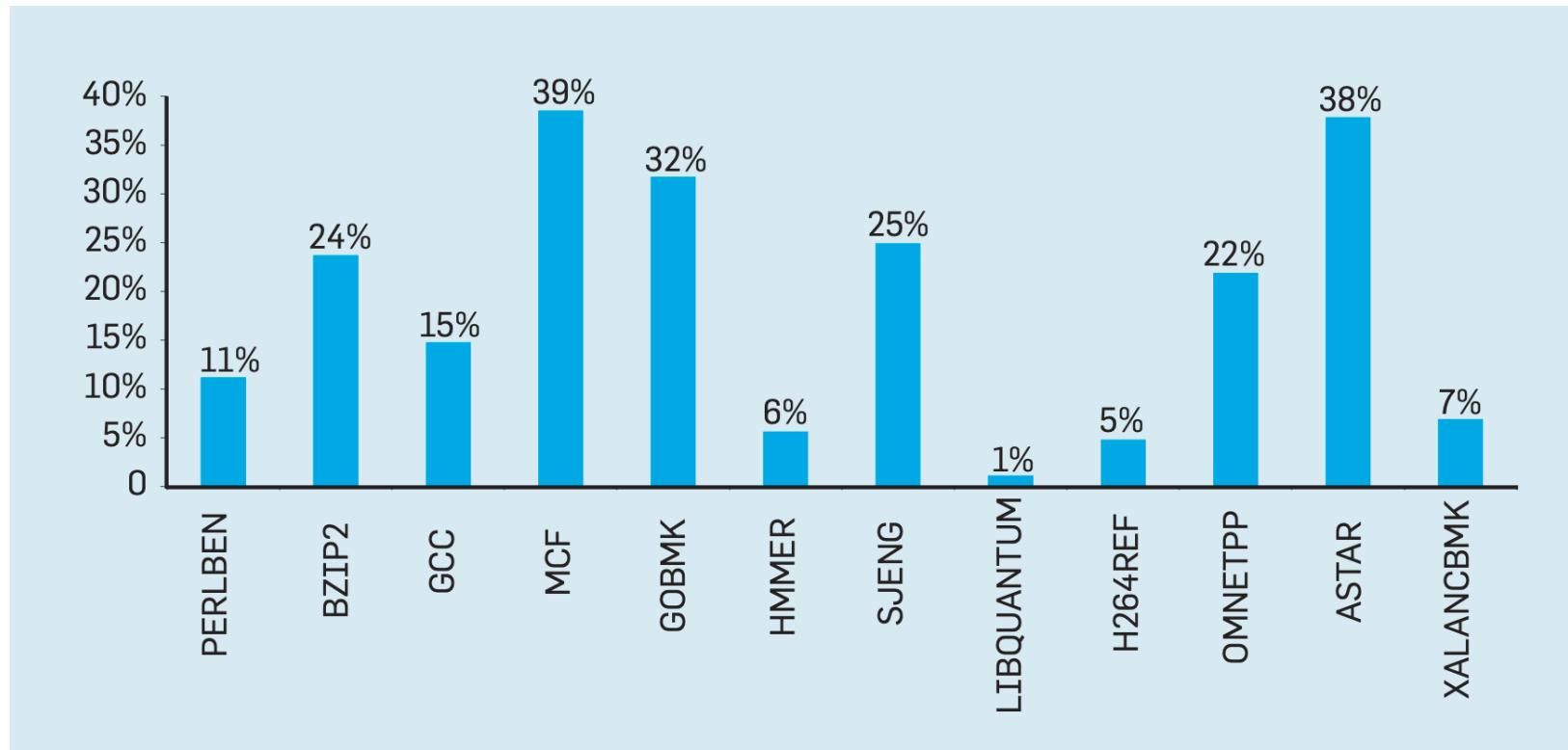
This shows the unsustainable power demands of every deepening pipelines.

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



# Branch prediction

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

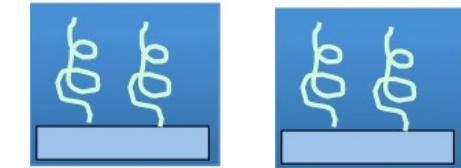


- On average, 19% of the instructions are wasted for these benchmarks on an Intel Core i7. The amount of wasted energy is greater, however, since the processor must use additional energy to restore the state when it speculates incorrectly.

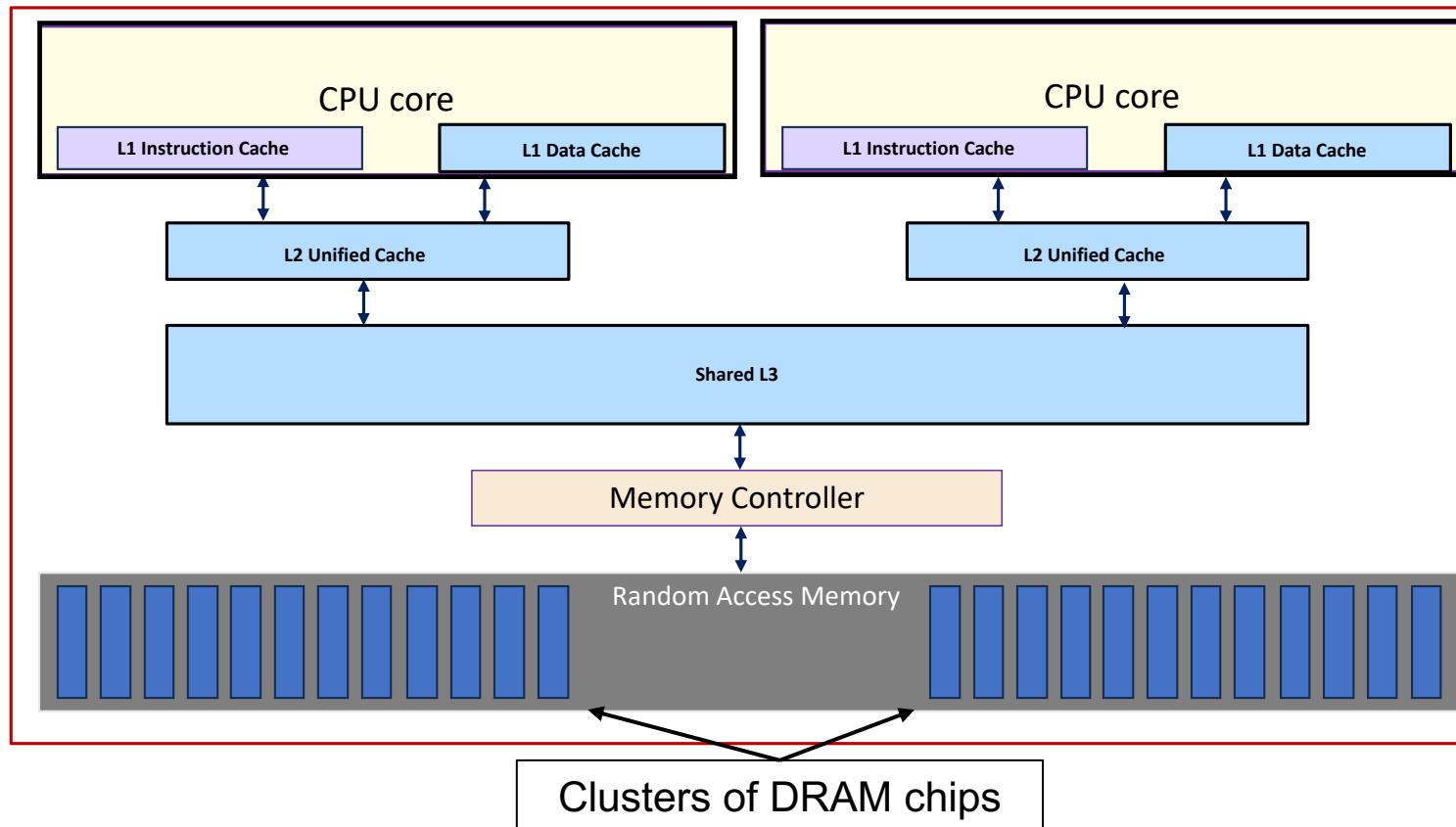
**Enough about the processors, what about the  
memory hierarchy**

# the Memory Hierarchy

We like to draw pictures like this

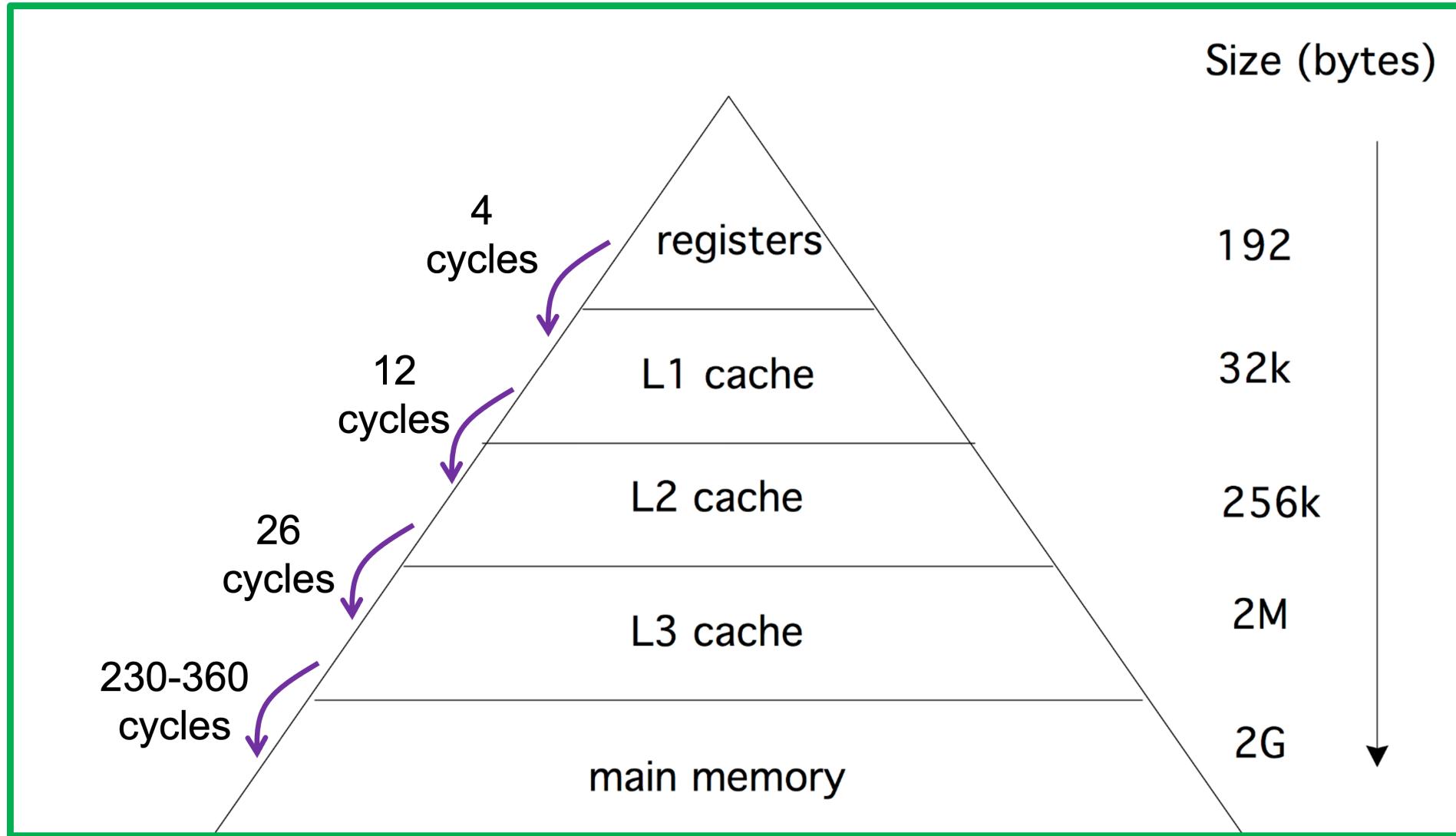


Random Access Memory

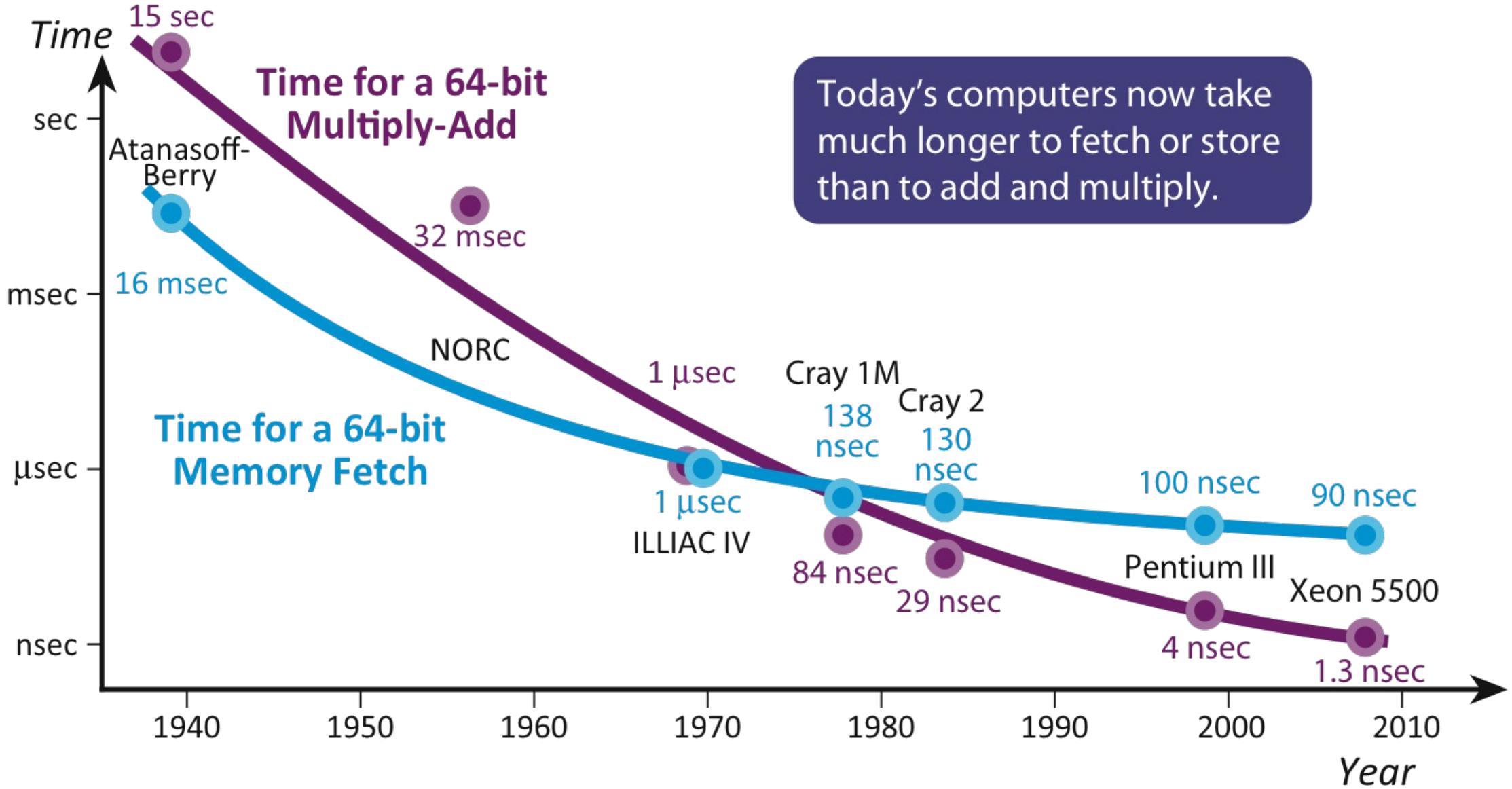


But reality is much more complex

# Latencies across the Memory Hierarchy

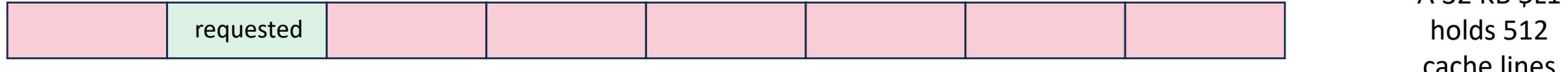


# Latencies across the Memory Hierarchy

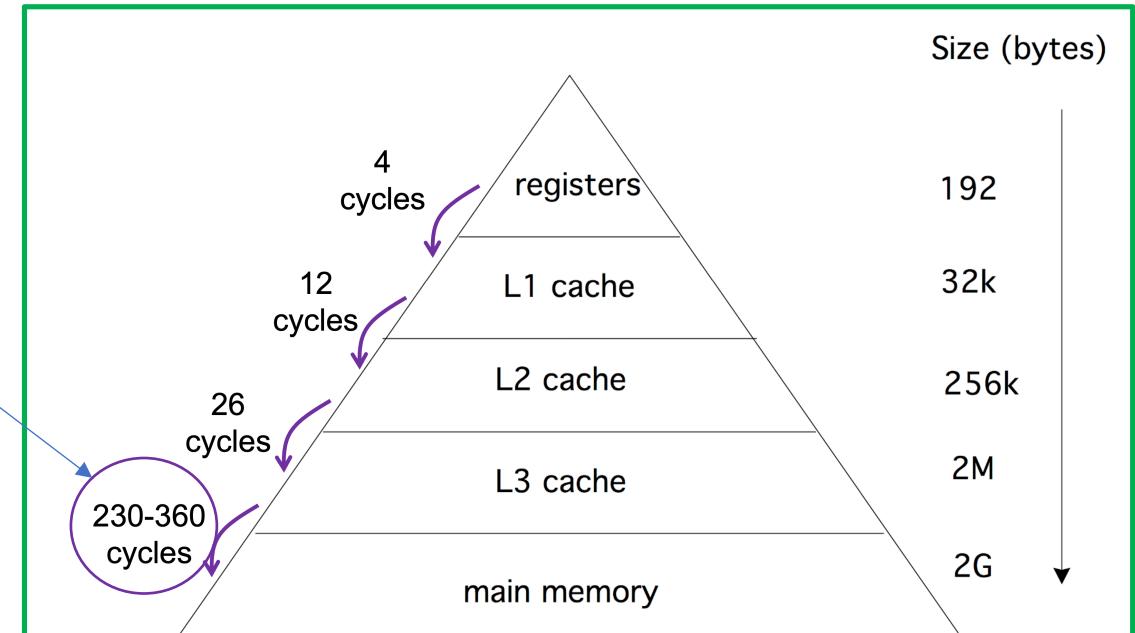


# Memory access occurs as Cache Lines

- 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



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;
```

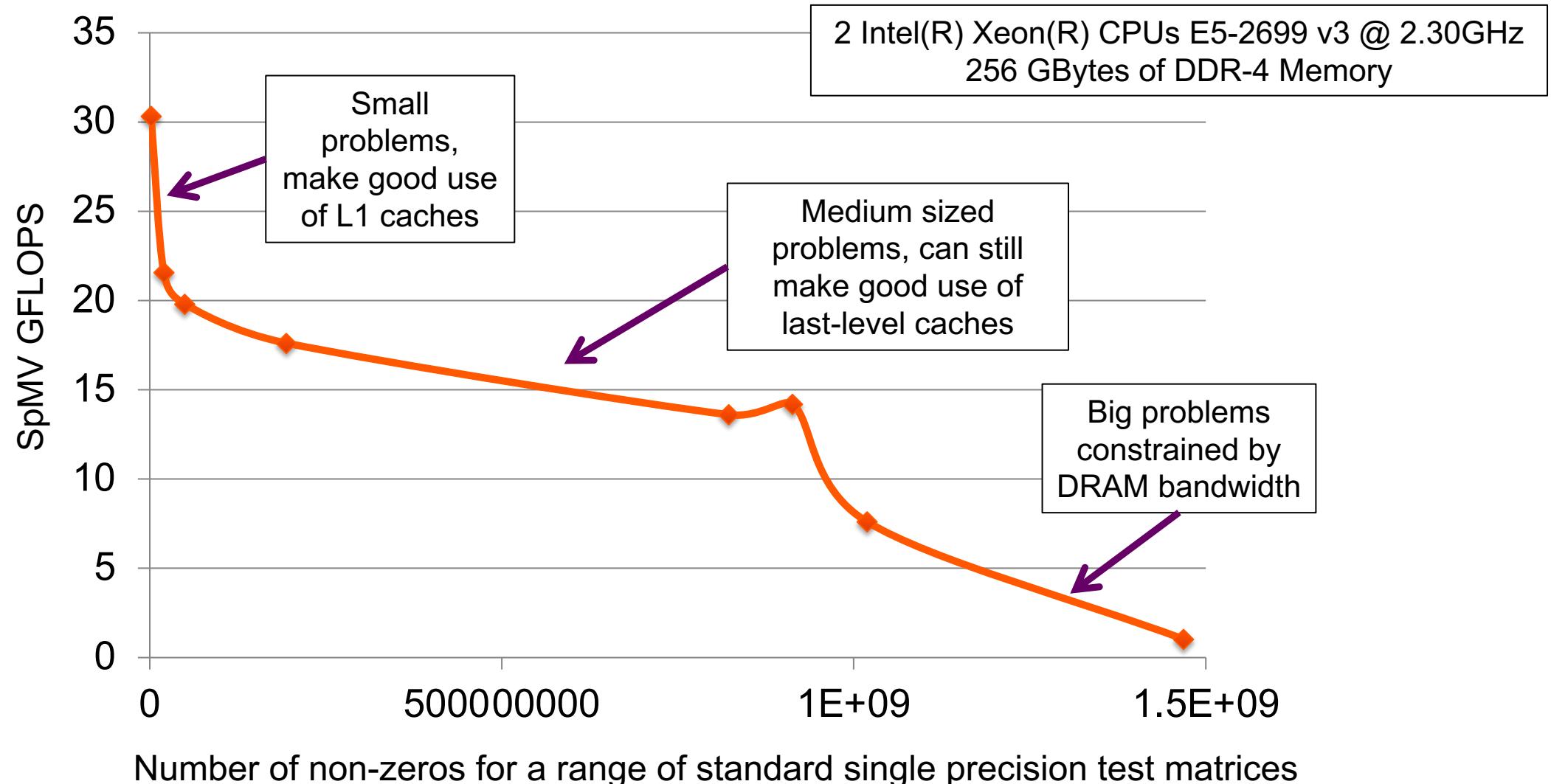
- Terrible Spatial locality ... traverse a linked list (pointer chasing). Generate lots of “cache thrashing”.

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

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

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

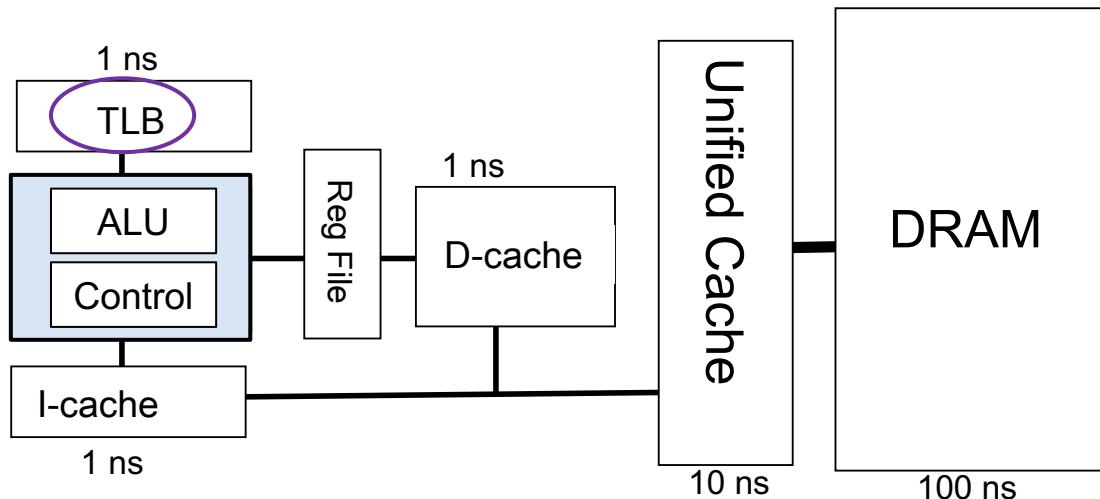
# Sparse matrix vector multiplication (SpMV) is a good way to see the impact of caches across the memory hierarchy



Source: Intel® MKL™ 11.3, “mkl\_cspblas\_scsrgemv” with KMP\_AFFINITY='verbose,granularity=fine,compact,1,0'

# Memory Hierarchies

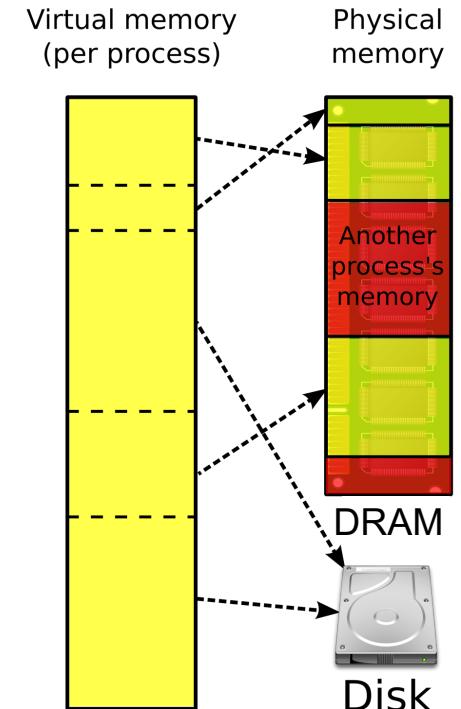
- A typical microprocessor memory hierarchy (with a simplified cache hierarchy)



- 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



# Consider the “simple” Matrix Transpose

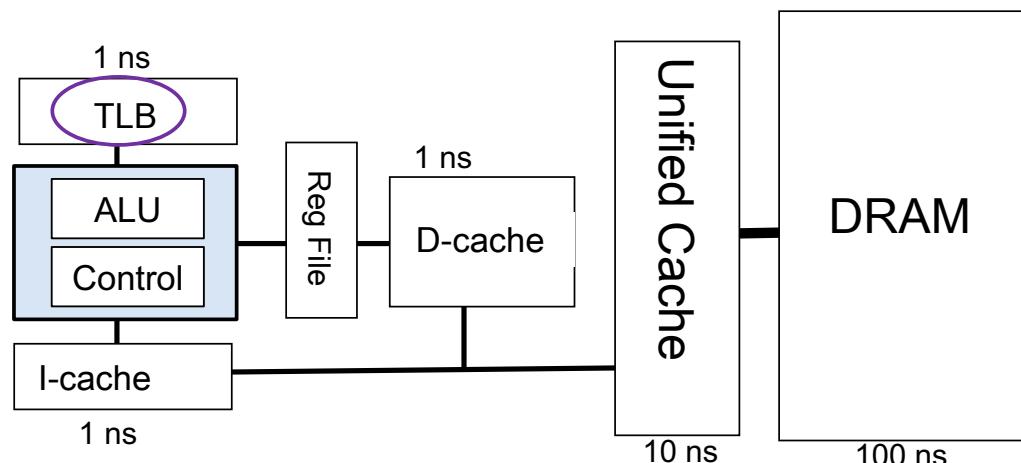
- Copy the transpose of A into a second matrix B.

$$A \in \mathbb{R}^{N \times N}$$

$$B \in \mathbb{R}^{N \times N}$$

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

- 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

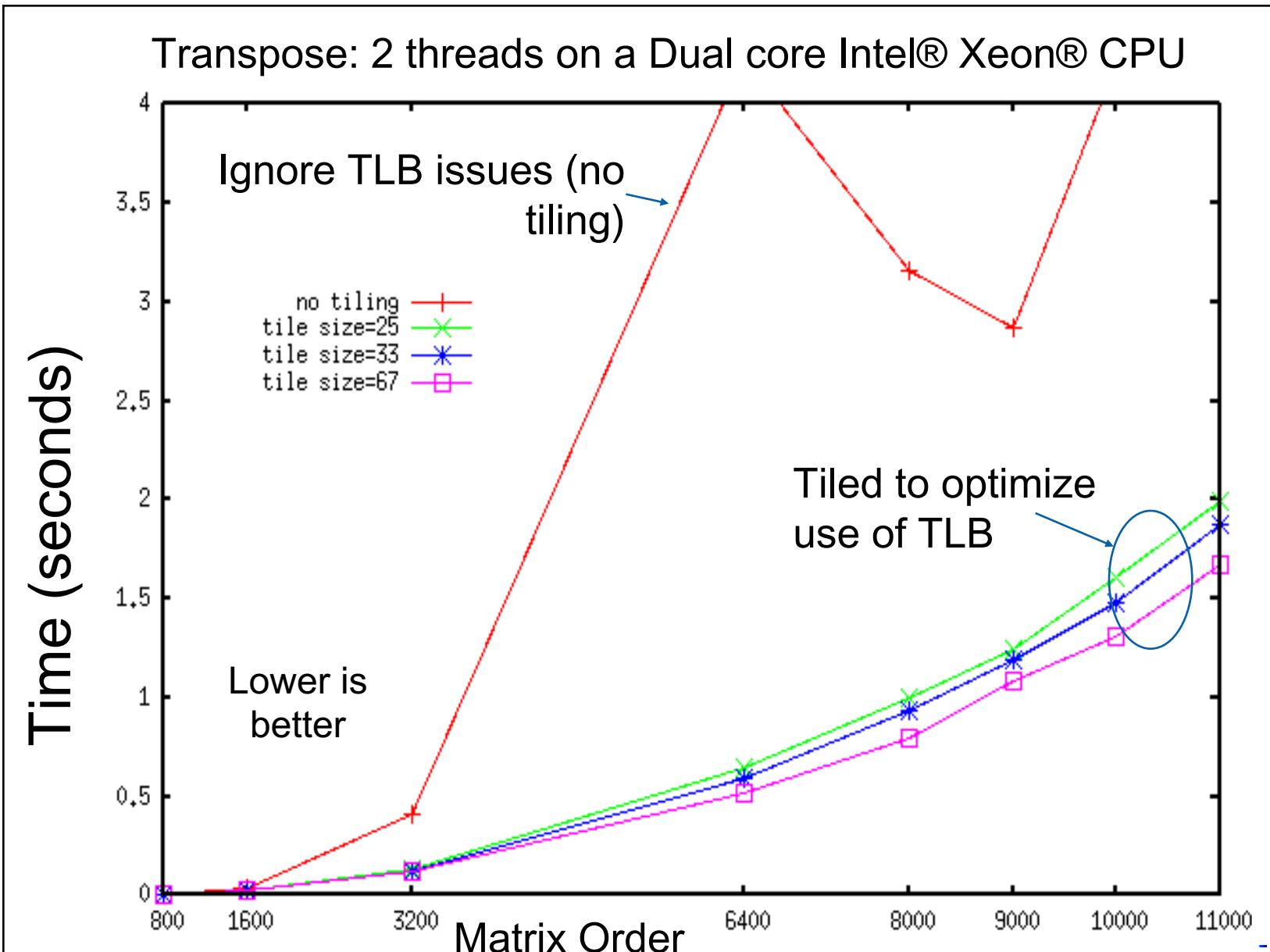
- Solution ... break the loops into blocks so we reduce the number of page faults

```
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];  
            }  
        }  
    }  
}
```

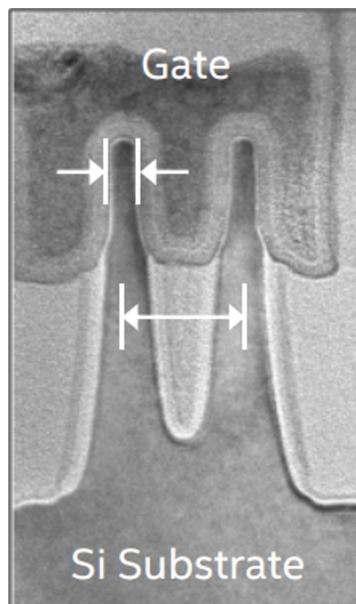
Step 2: rearrange the loops. Move the loops over the tiles to the innermost loop-nest

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

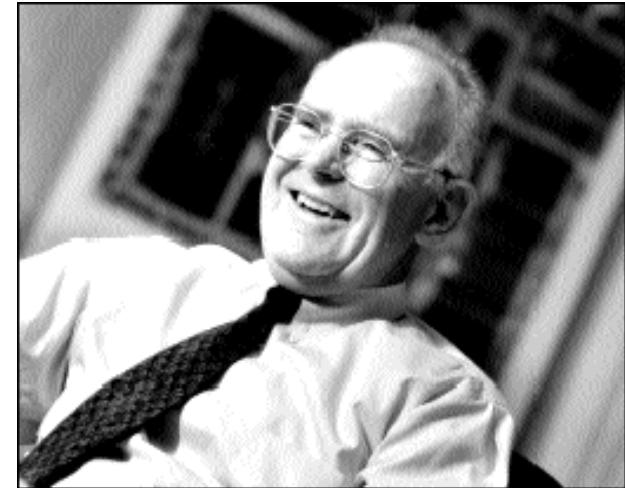
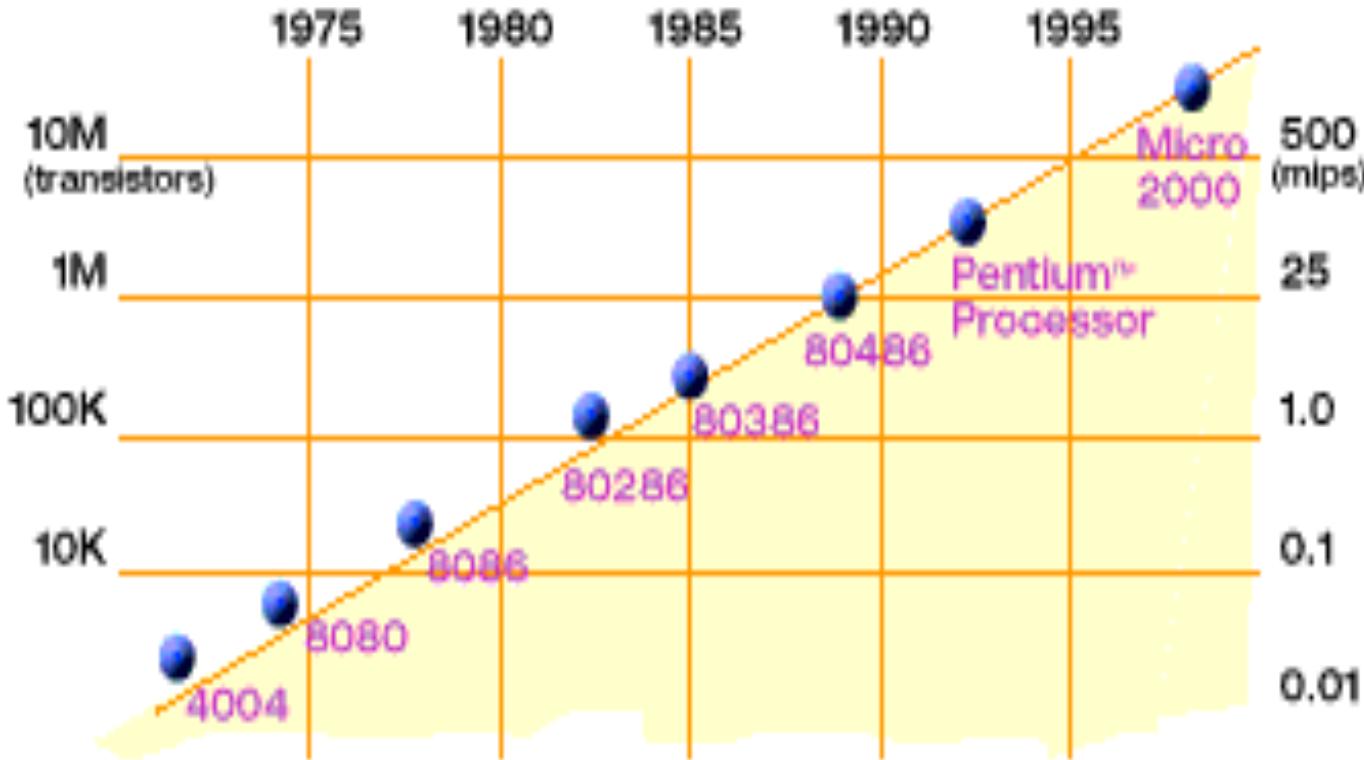
# Do you need to worry about the TLB?



**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.

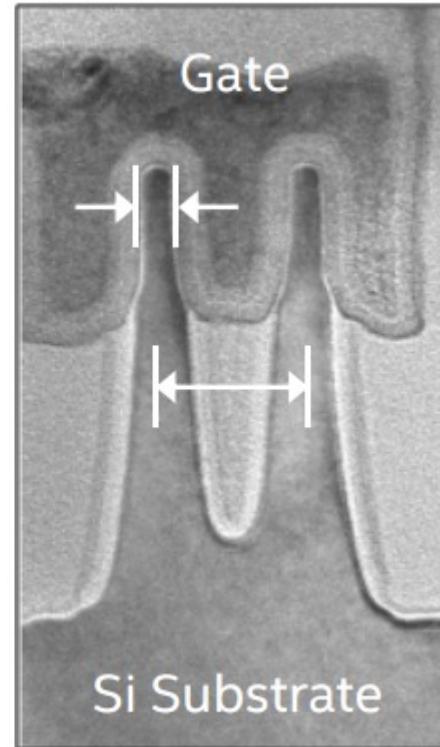
“Cramming more components onto integrated circuits”, G.E. Moore, Electronics, 38(8), April 1965

# 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

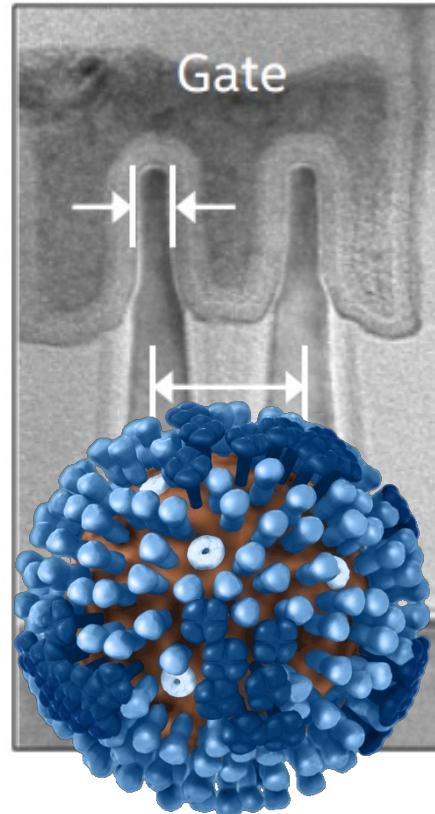
How small is a nm (nanometer)? One nm is  $10^{-9}$  meters. Light travels about one foot in  $10^{-9}$  seconds.

# 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](http://www.cdc.gov/flu/images/h1n1/3D_Influenza/3D_Influenza_transparent_no_key_full_med2.gif)

# How did those itty-bitty transistors impact performance?

## The Linpack benchmark over time



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

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



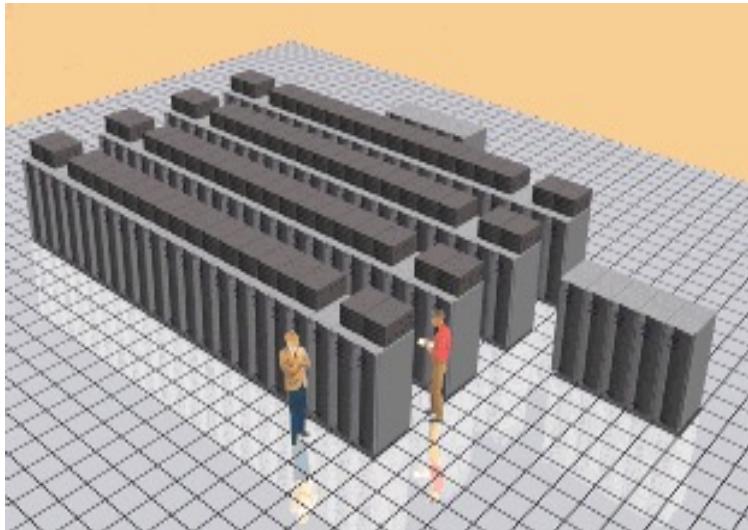
Apple iPhone 6S,  
2016, 1274 MFLOPS  
“in my pocket”



VAX 11/780, ~1980,  
0.14 the computer I  
used in my Ph.D.  
research

# Moore's Law: A personal perspective

First TeraScale\* computer: 1997



Intel's ASCI Option Red

## Intel's ASCI Red Supercomputer

9000 CPUs

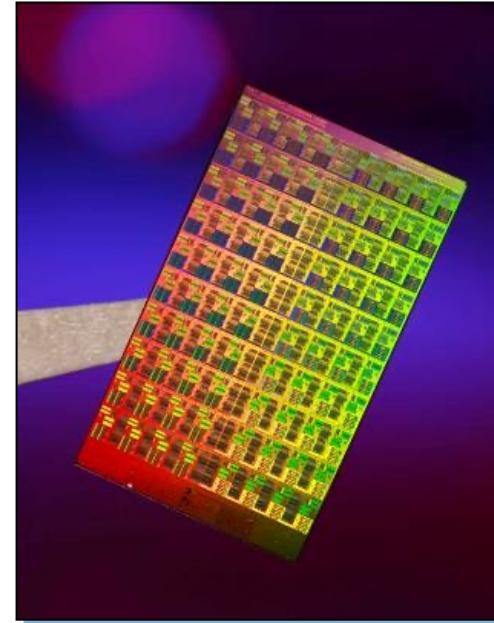
one megawatt of electricity.

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.

First TeraScale% chip: 2007



## Intel's 80 core teraScale Chip

1 CPU

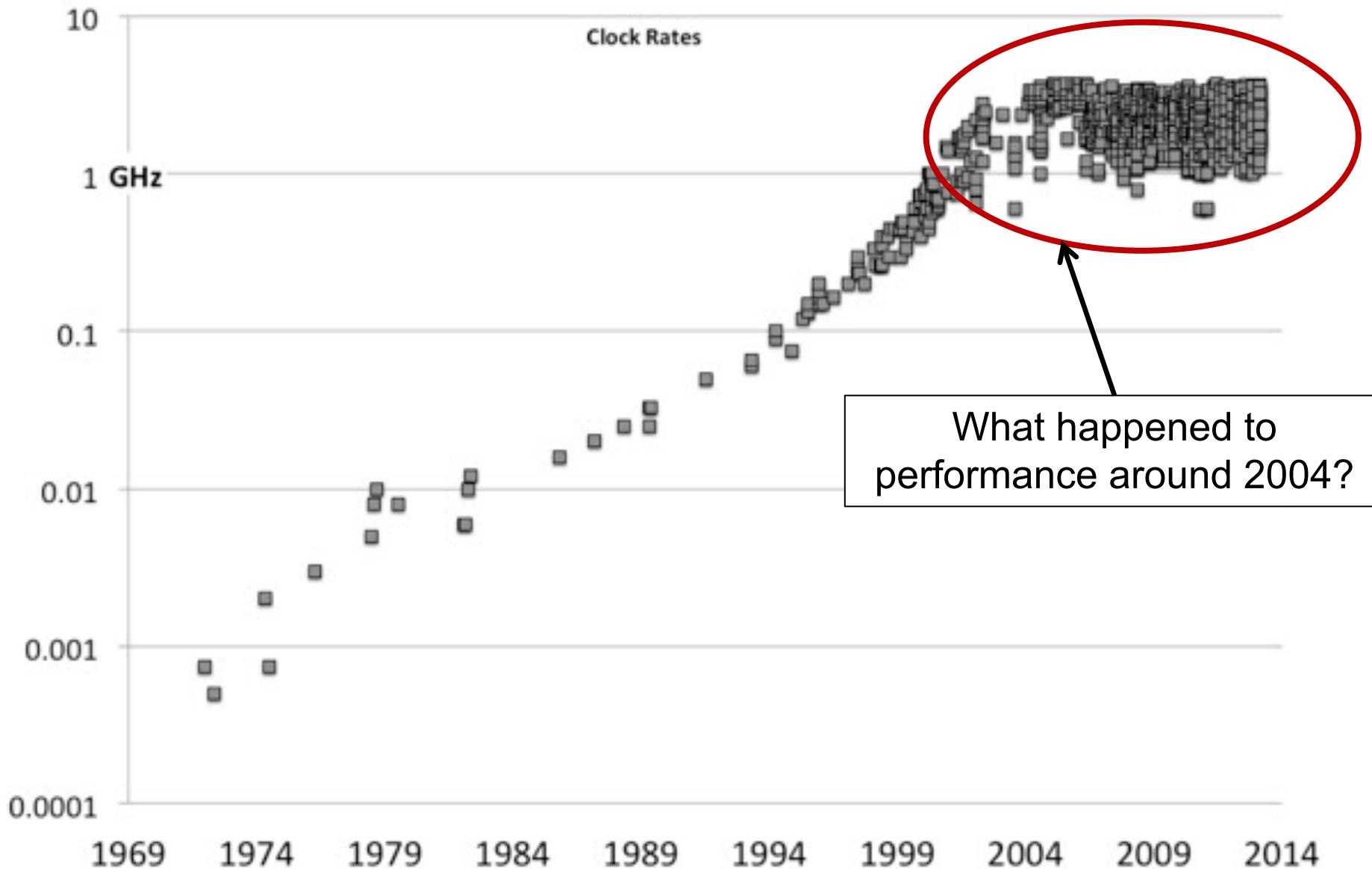
97 watt

275 mm<sup>2</sup>

%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

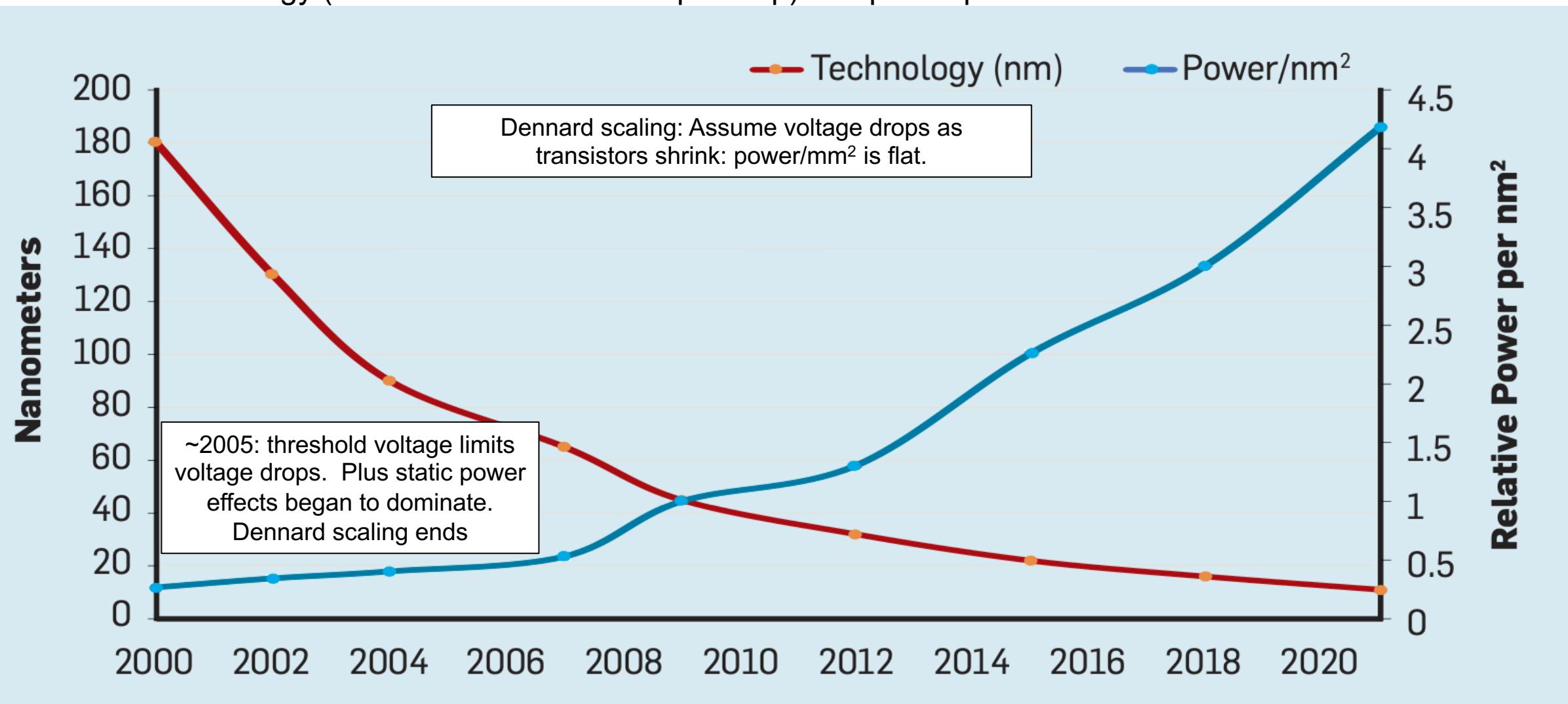
# CPU Frequency (GHz) over time (years)



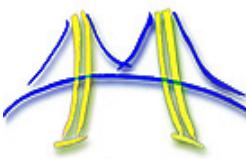
Source: James Reinders (from the book “structured parallel programming”)

# Dennard Scaling

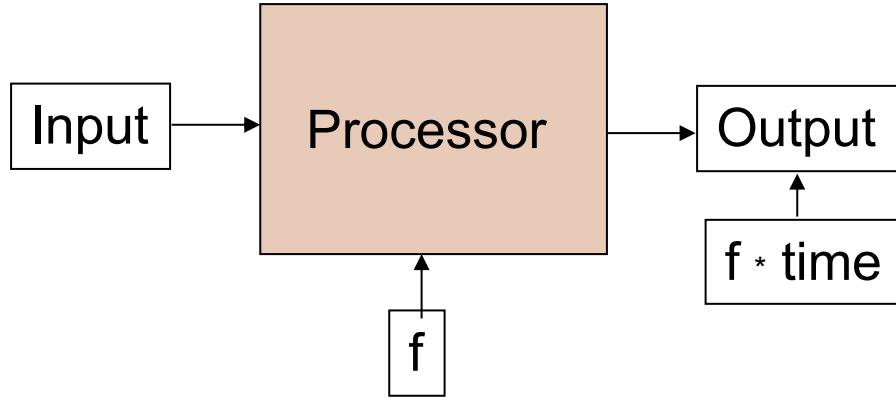
- Process technology (translates to Transistors per chip) and power per mm<sup>2</sup>



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.



# 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:

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

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

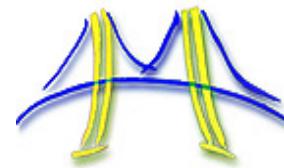
$$V * q = W.$$

But for a circuit  $q = CV$  so

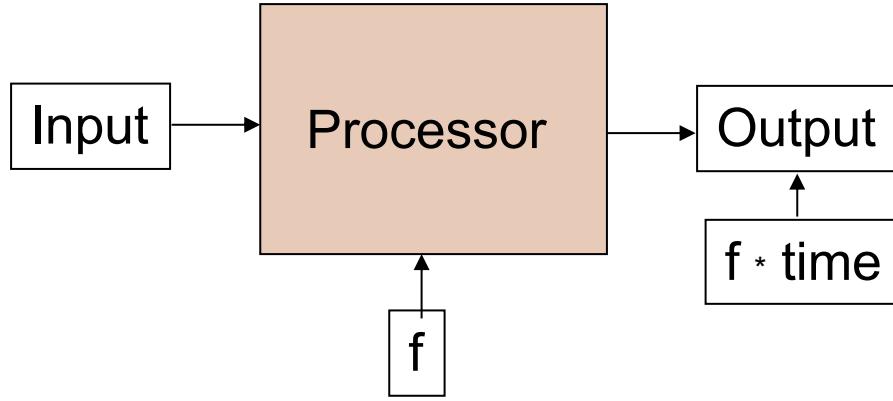
$$W = CV^2$$

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

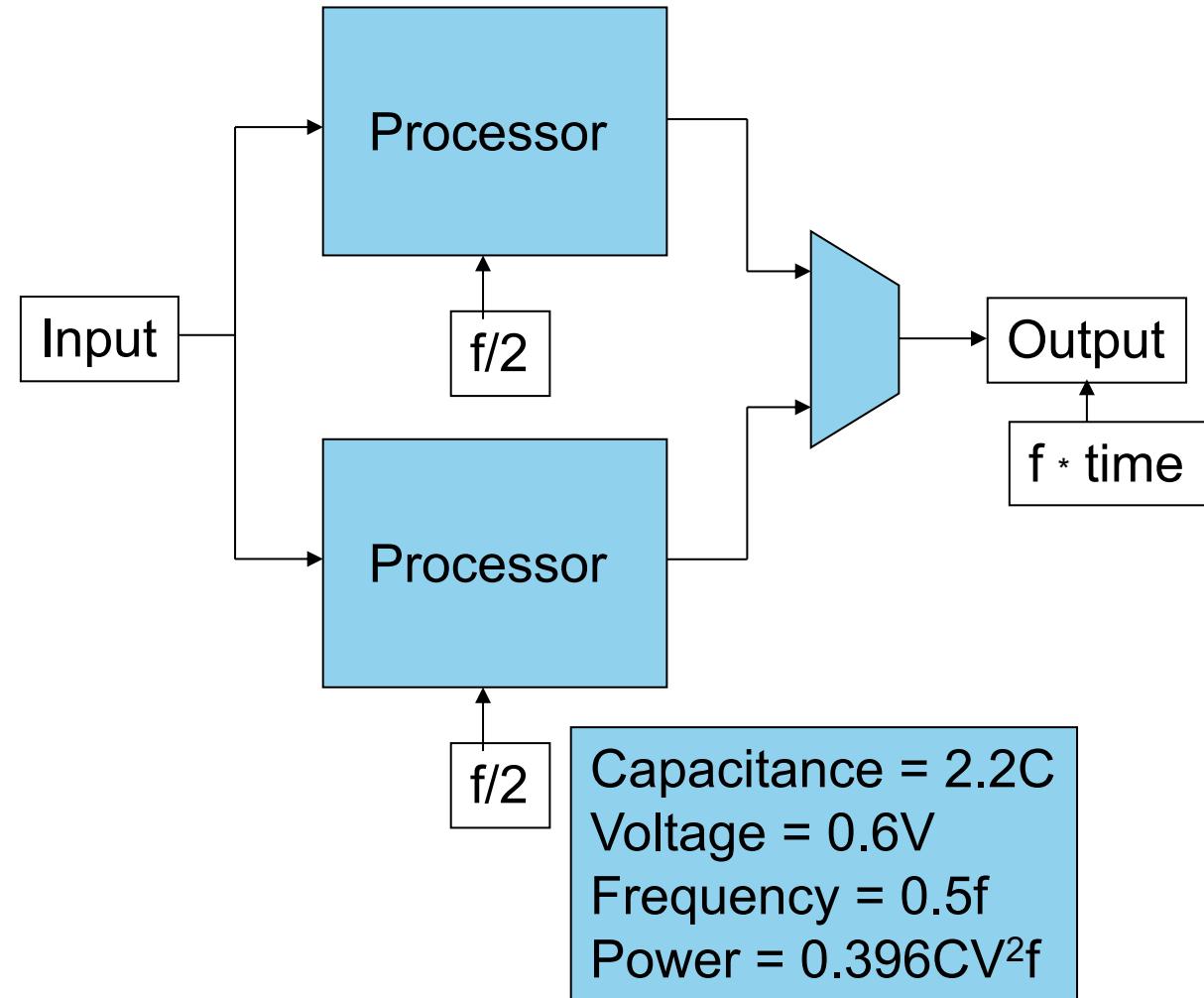
$$\text{Power} = W * F \rightarrow \text{Power} = CV^2f$$



# ... Reduce power by adding cores

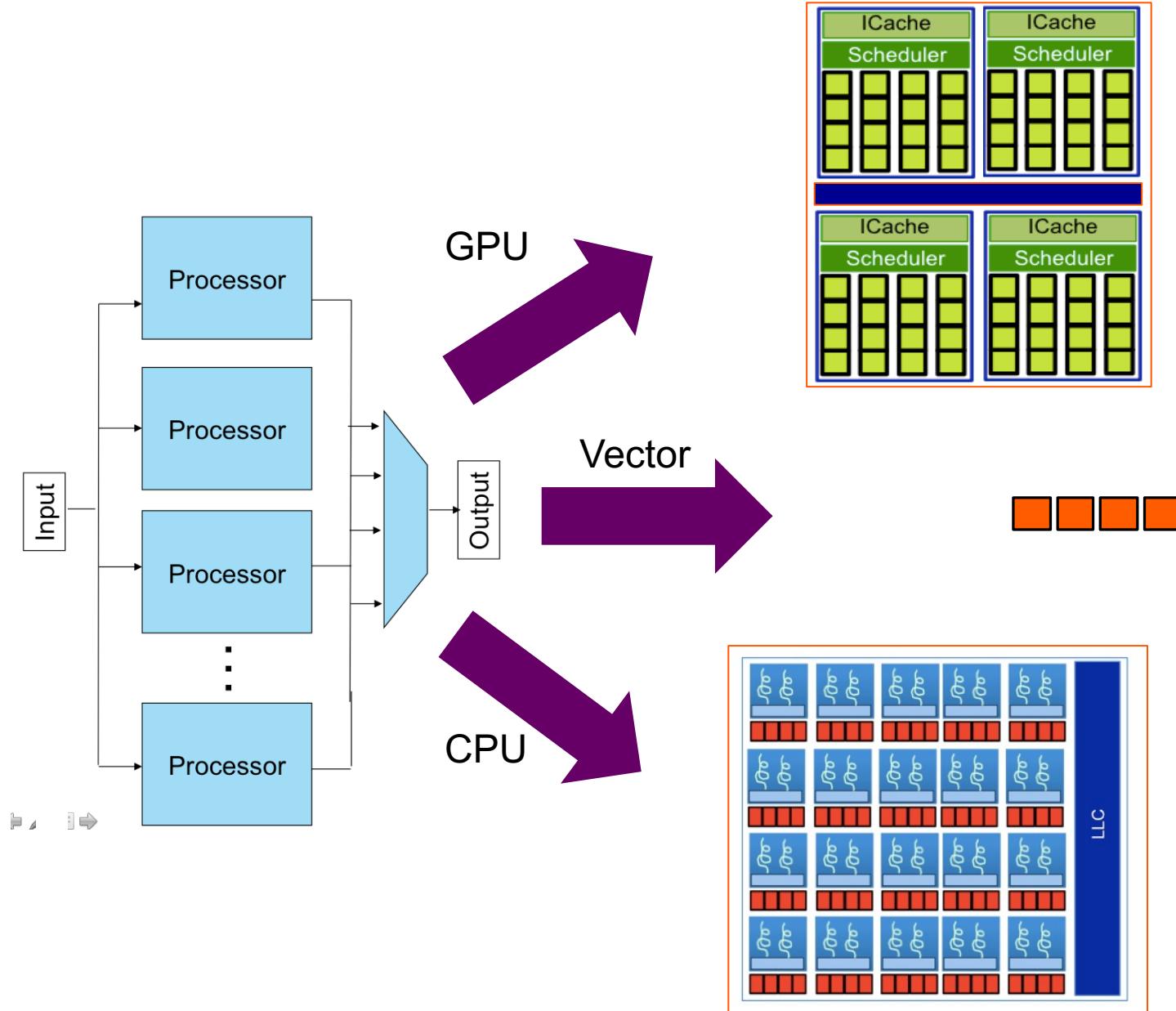


Capacitance = C  
Voltage = V  
Frequency = f  
Power =  $CV^2f$



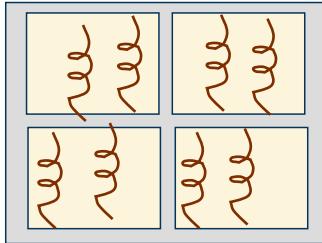
Capacitance =  $2.2C$   
Voltage =  $0.6V$   
Frequency =  $0.5f$   
Power =  $0.396CV^2f$

# Manycore processors: three hardware options



# For hardware ... parallelism is the path to performance

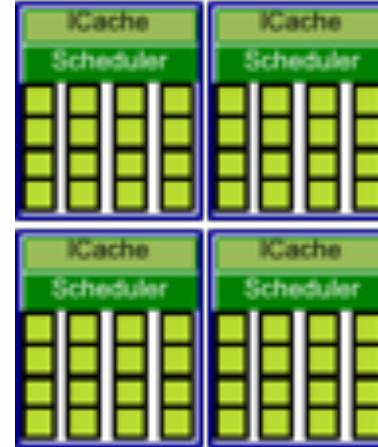
All hardware vendors are in the game ... parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



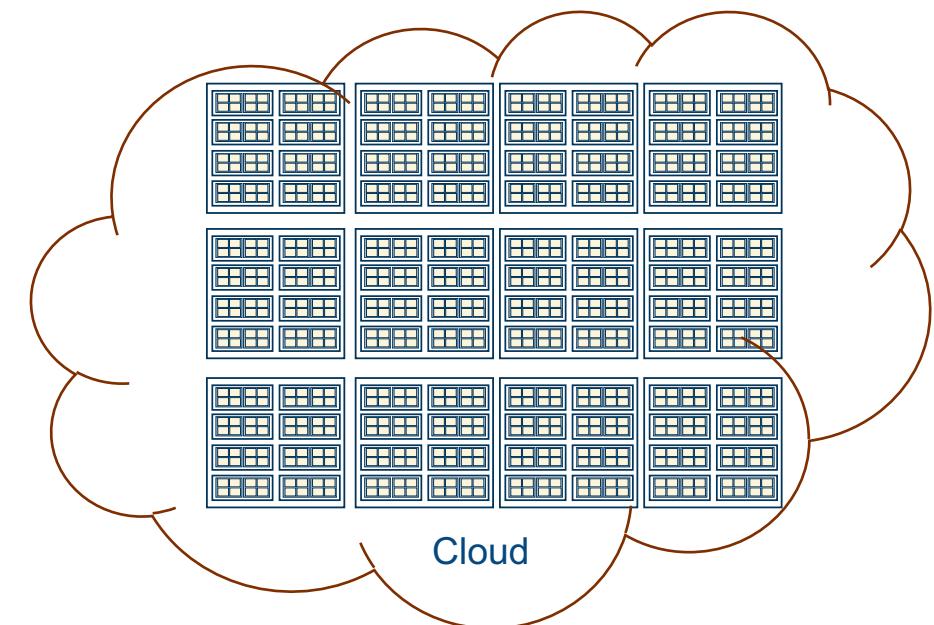
CPU



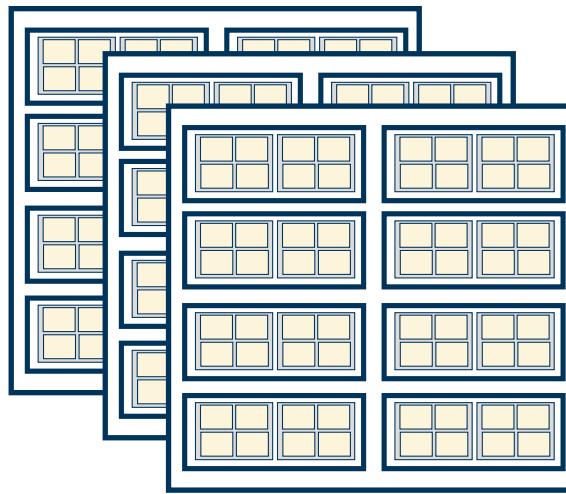
SIMD/Vector



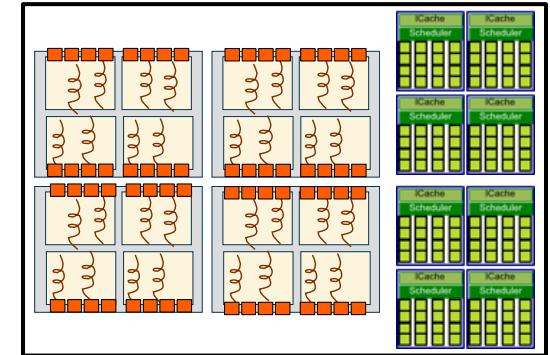
GPU



Cloud



Cluster



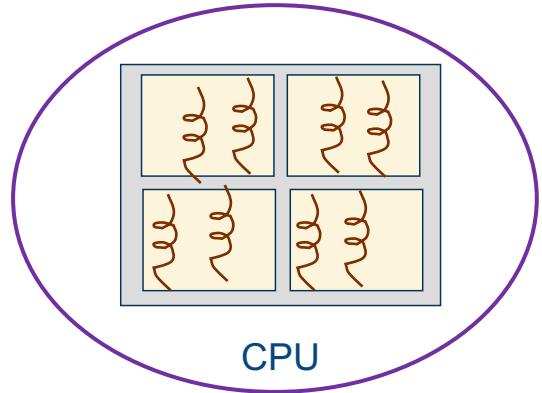
Heterogeneous node

## **Let's quickly survey the key “on-node” parallel hardware approaches**

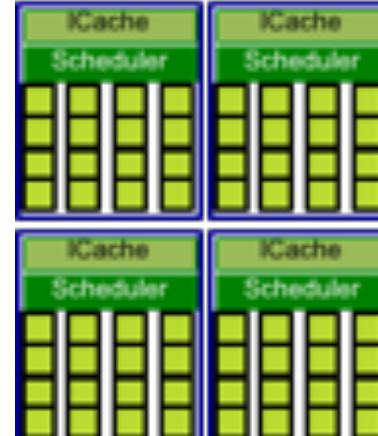
By the term “on-node” I mean we are not going to discuss parallelism that comes from networking together large numbers of independent computer systems (as is done for cluster and cloud computing).

# For hardware ... parallelism is the path to performance

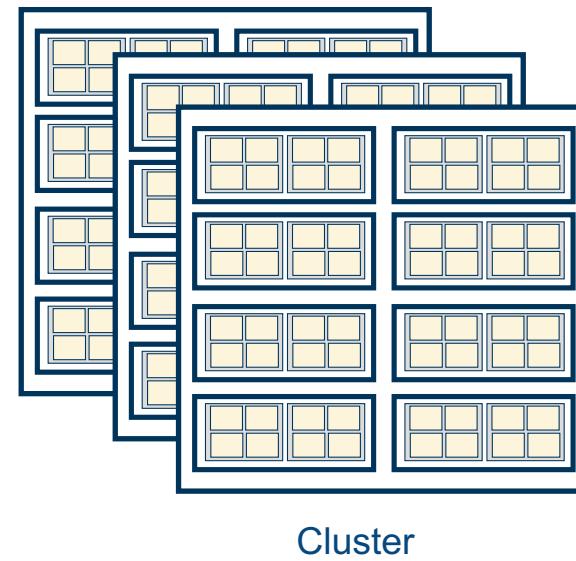
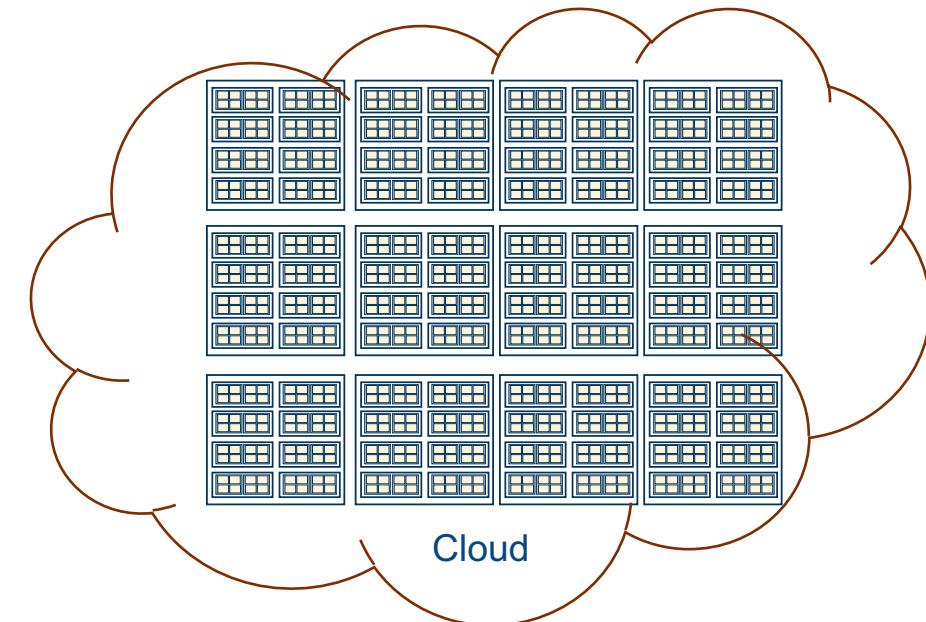
All hardware vendors are in the game ... parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



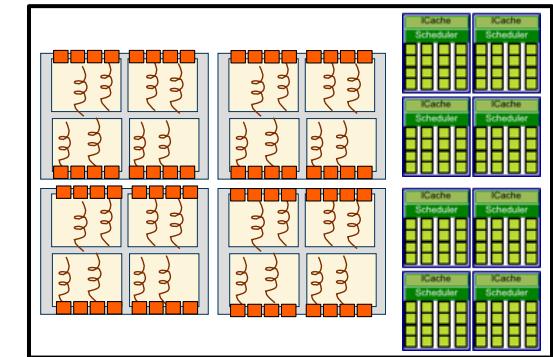
SIMD/Vector



GPU



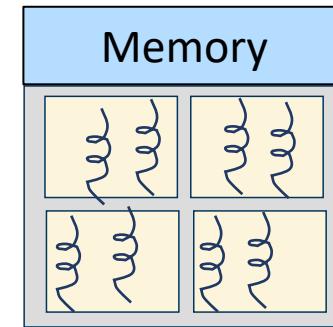
Cluster



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

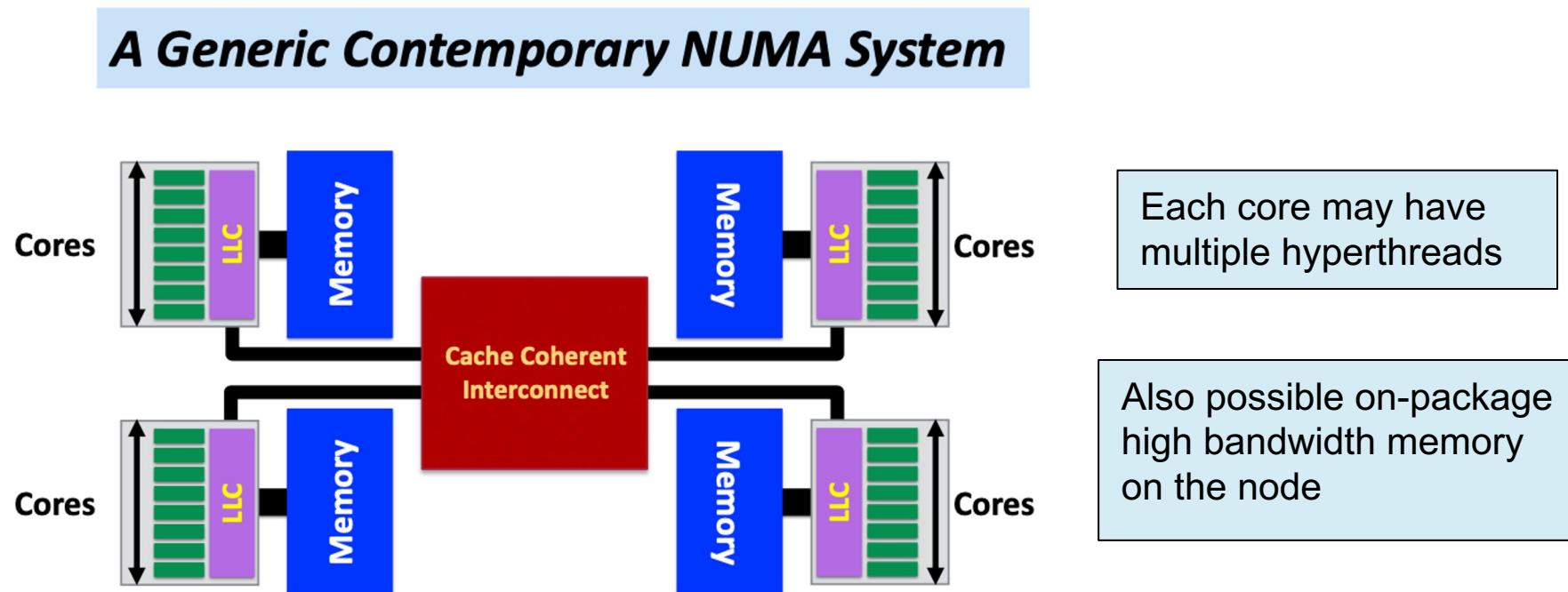


Diagram courtesy Ruud van der Pas

LLC: Last level cache

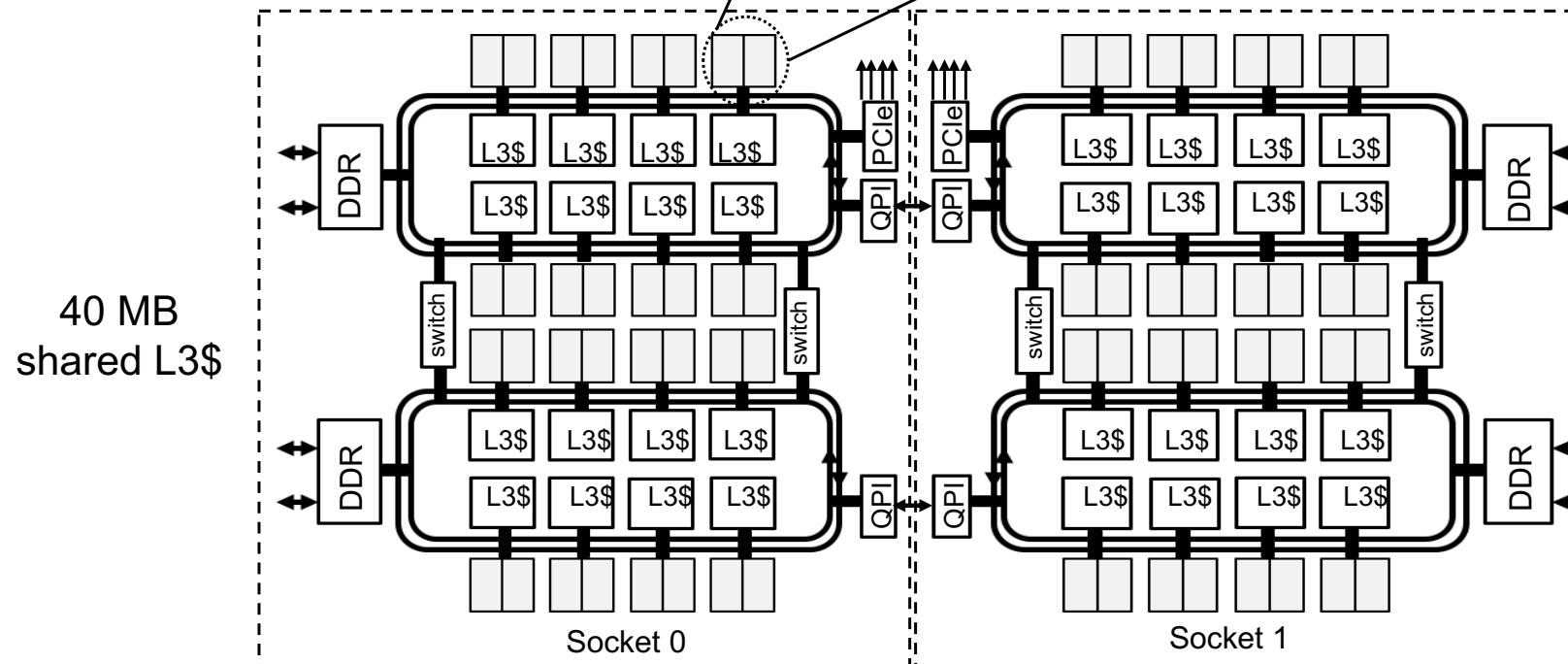
# CPU and Memory: a harsh does of reality

## 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



DDR: Double Data Rate memory controller

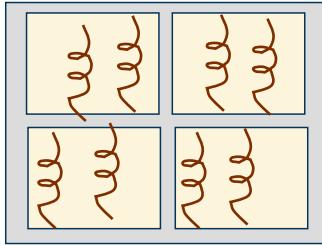
PCIe is the connection from the CPU to other devices in a node.

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

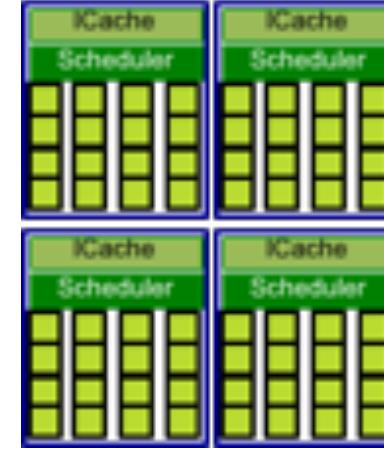
All hardware vendors are in the game ... parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



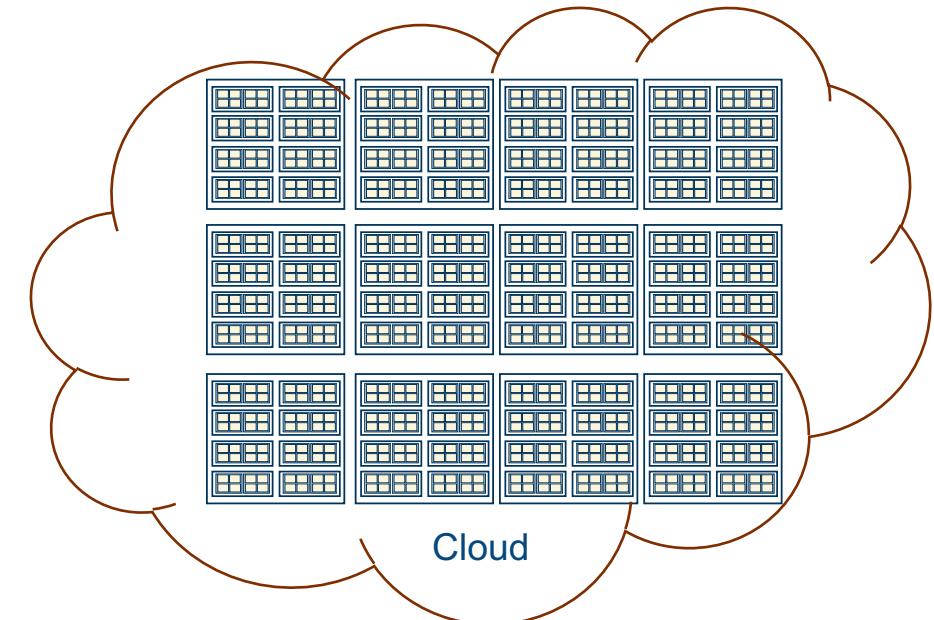
CPU



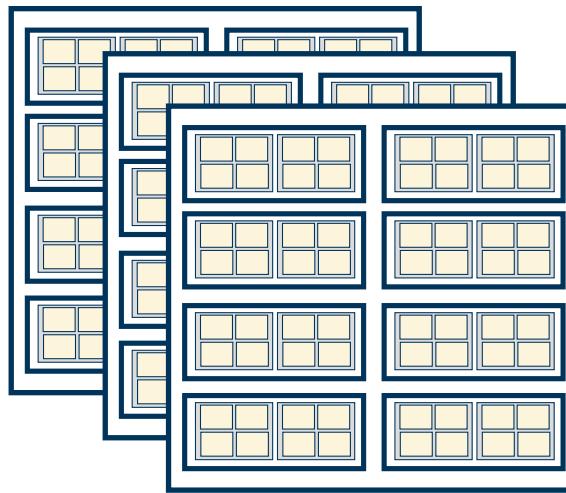
SIMD/Vector



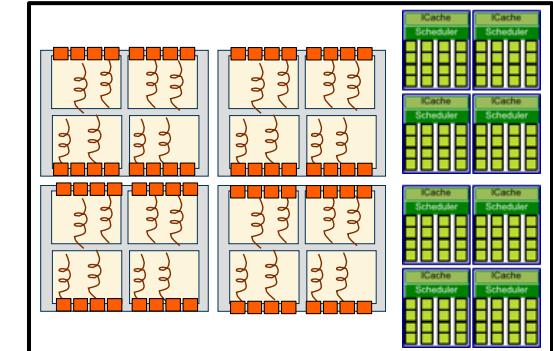
GPU



Cloud



Cluster

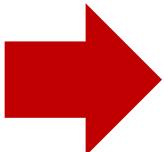


Heterogeneous node

# 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)

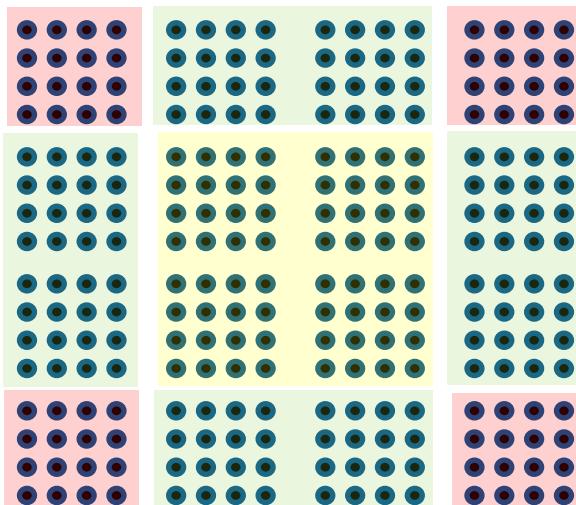
1. Turn source code into a scalar work-item

```
// 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);
}
```

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



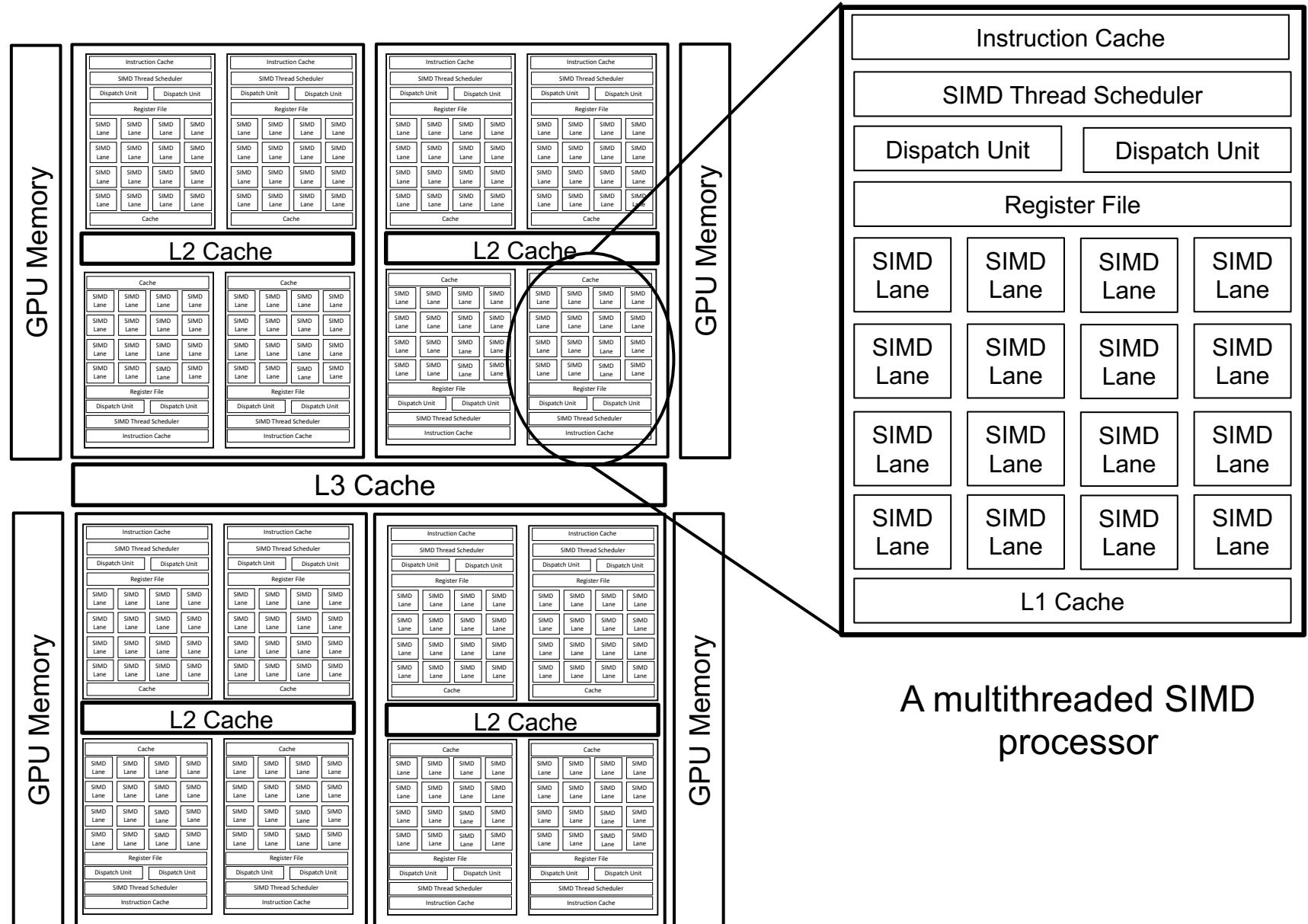
4. Run on hardware designed around the same SIMT execution model



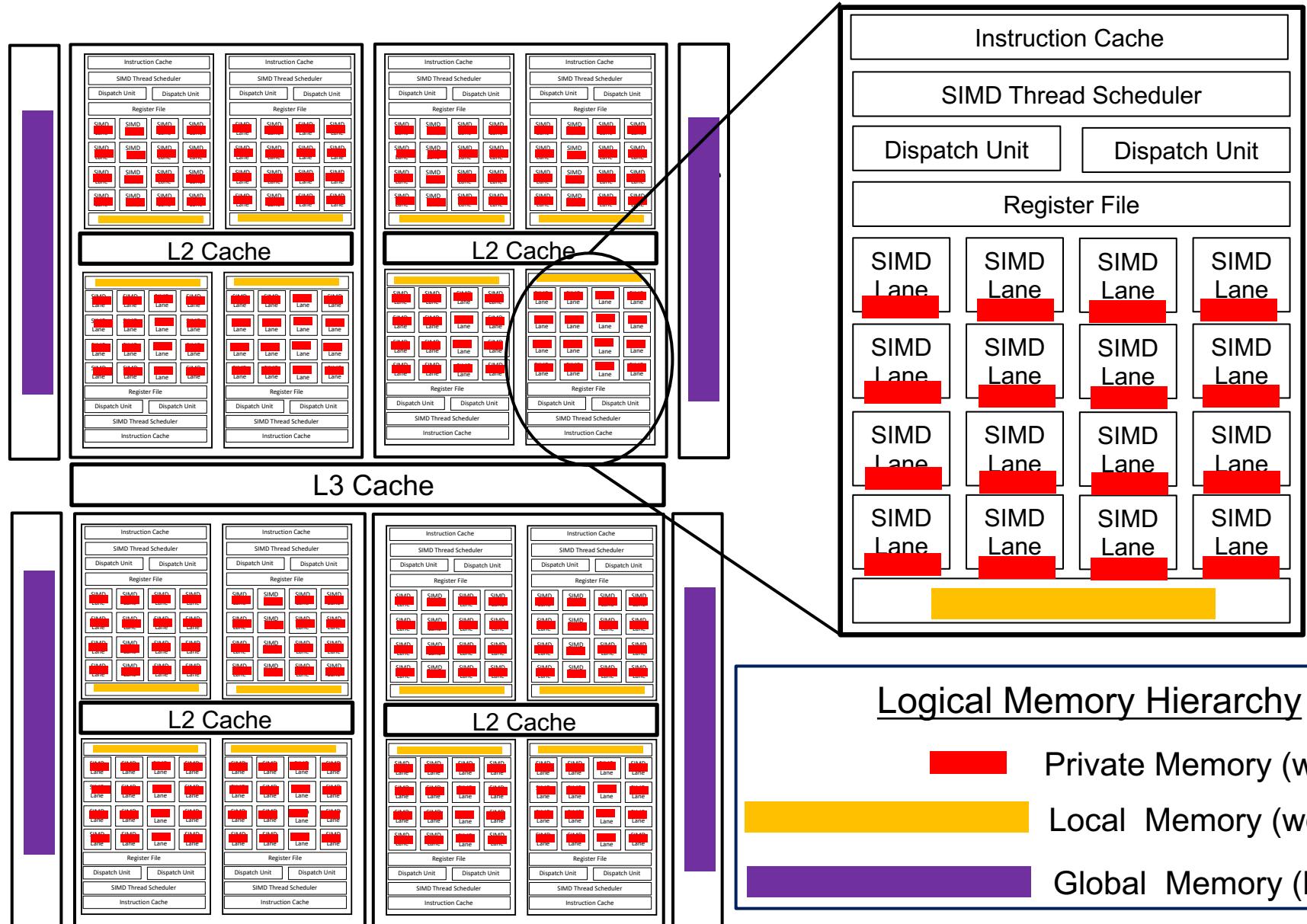
3. Map data structures onto the same index space

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

# A Generic GPU (following Hennessy and Patterson)



# A Generic GPU (following Hennessy and Patterson)



**Look out for dishonest GPU/CPU  
benchmarking**

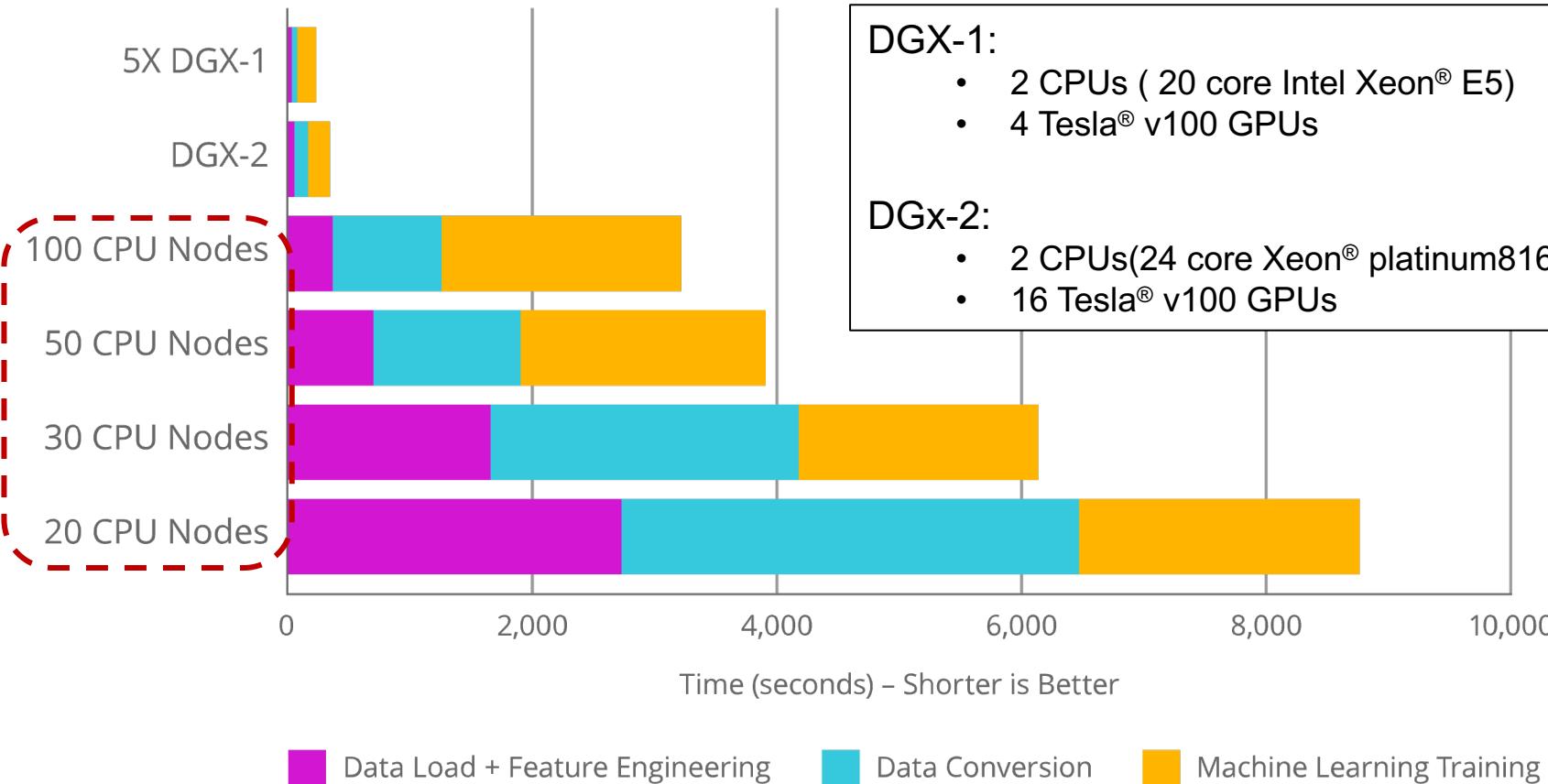
# NVIDIA Performance claims

An Nvidia slide from CLSAC'18 talk

This is the most dishonest marketing I have EVER seen.

## End-to-End Faster Speeds on RAPIDS

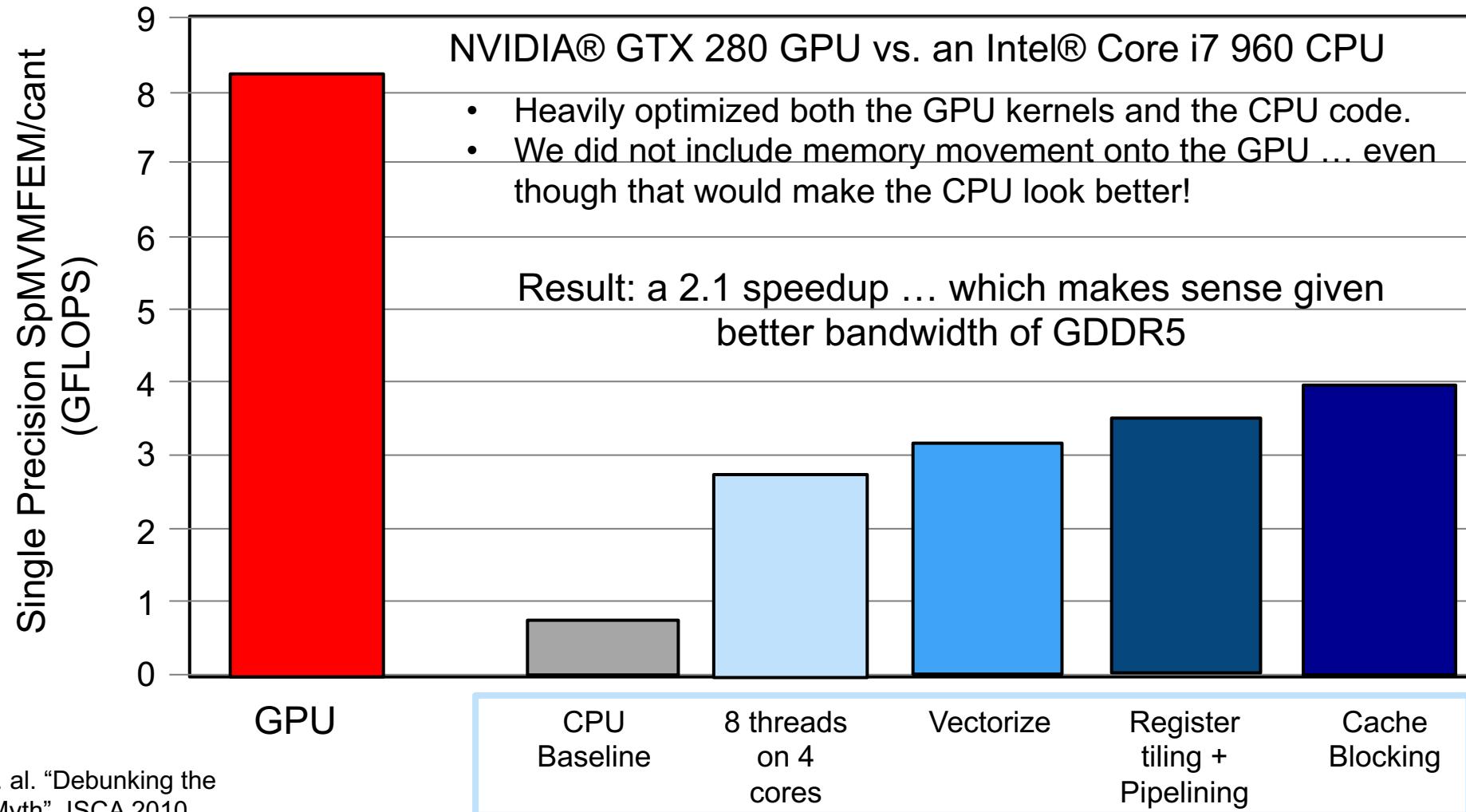
**"CPU Node"** = 1 AWS Broadwell Intel® Xeon® E5 hardware thread.  
Their "100 CPU nodes" is equivalent to three Intel® Xeon® E5 processors.



Plus they are comparing unoptimized, generic Python to highly optimized, custom CUDA code!

# 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

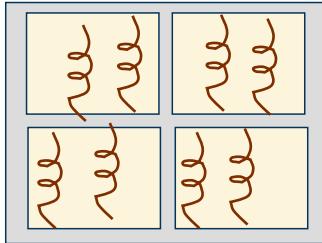


Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

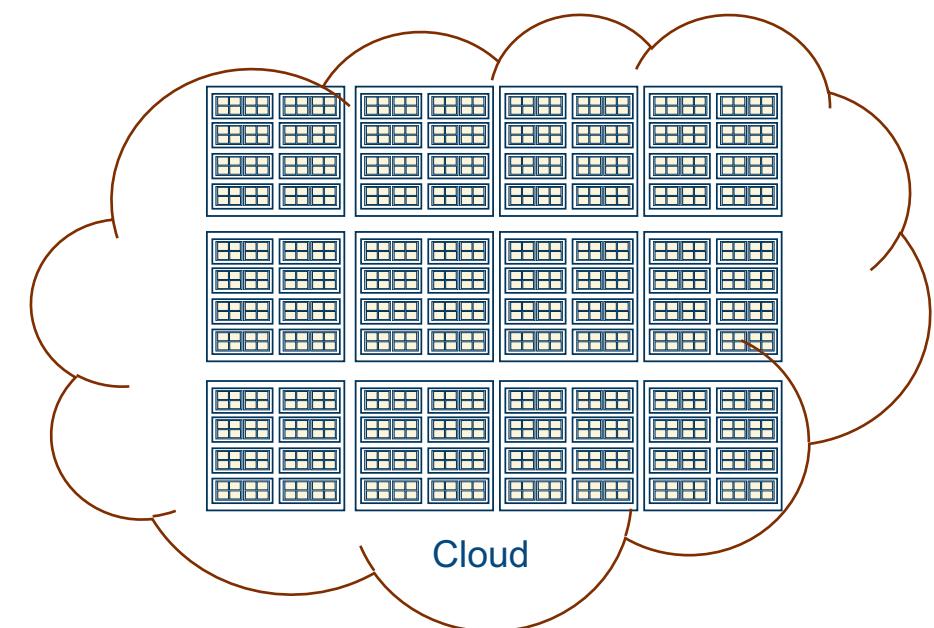
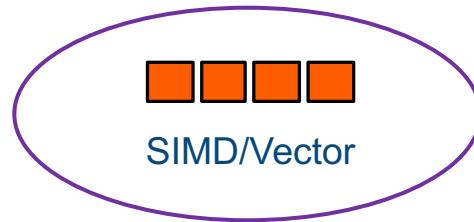
[Vazquez09]: F. Vazquez, E. M. Garzon, J.A.Martinez, and J.J.Fernandez. The sparse matrix vector product on GPUs. Technical report, University of Almeria, June 2009.

# For hardware ... parallelism is the path to performance

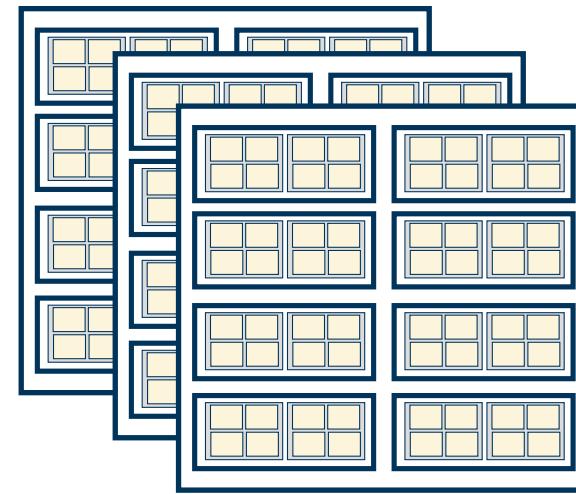
All hardware vendors are in the game ... parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



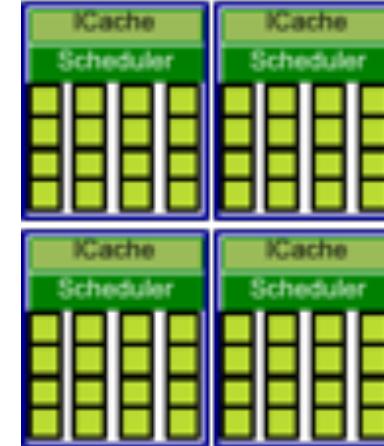
CPU



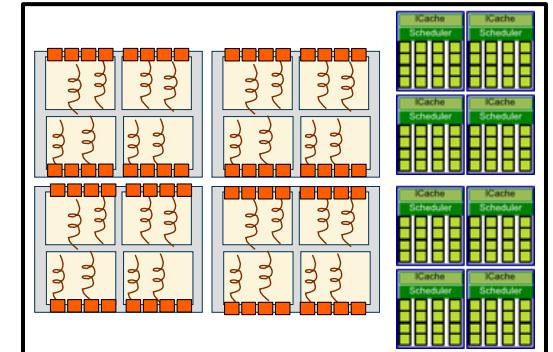
Cloud



Cluster

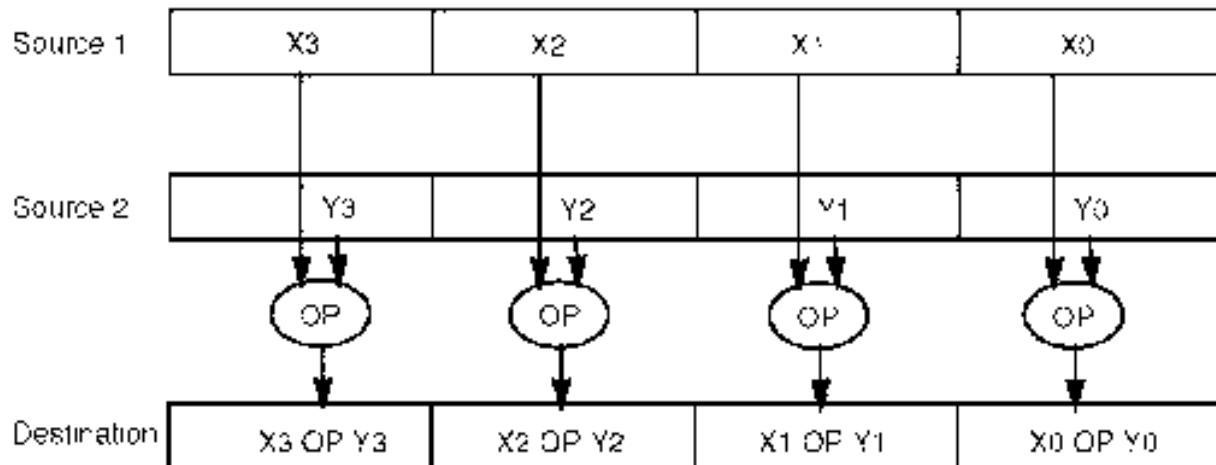
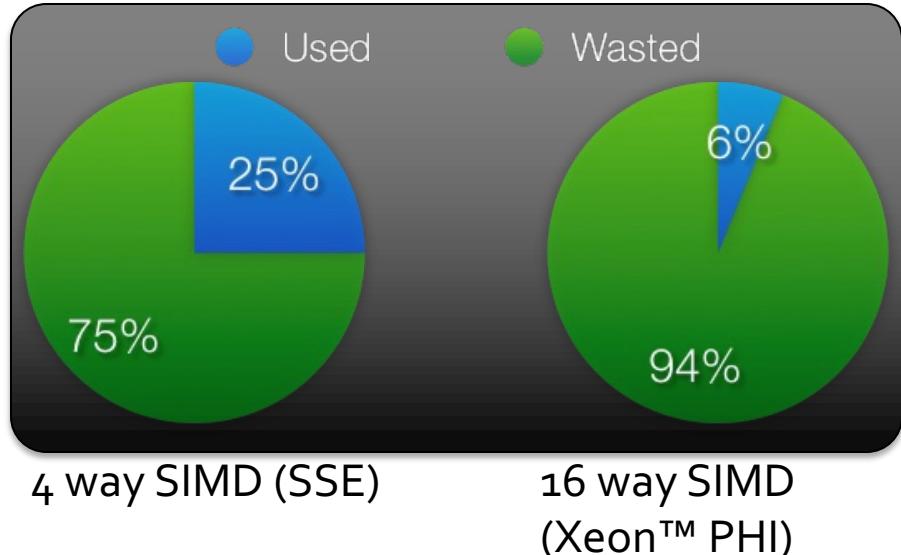


GPU



Heterogeneous node

# Vector (SIMD) Programming

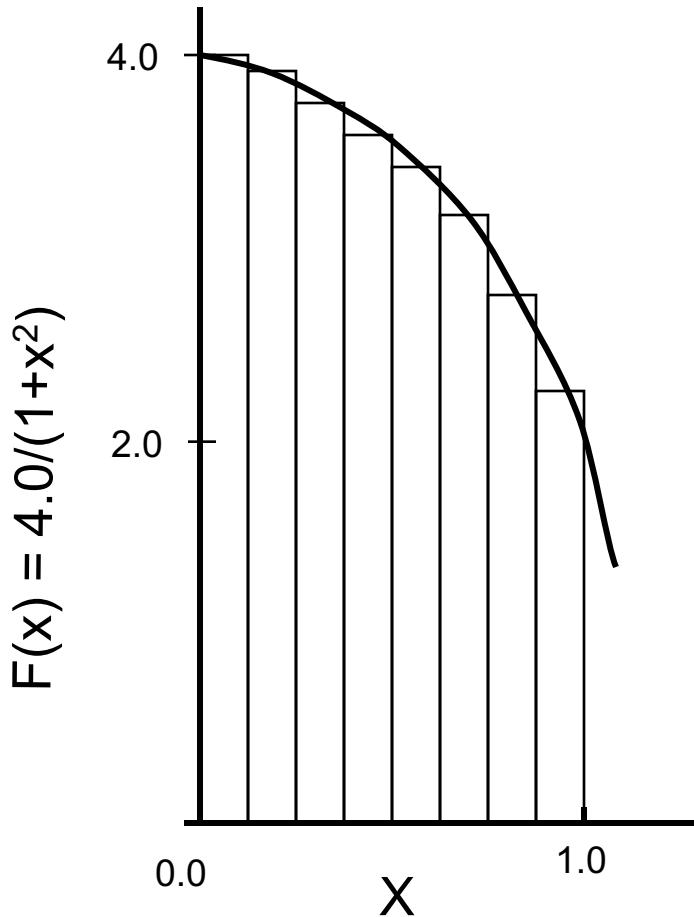


- Architects love vector units, since they permit space- and energy-efficient parallel implementations.
- However, standard SIMD instructions on CPUs are inflexible, and can be difficult to use.
- Options:
  - Let the compiler do the job
  - Assist the compiler with language level constructs for explicit 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  $\Delta x$  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

```
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;
}
```

- We need one iteration 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

# 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;

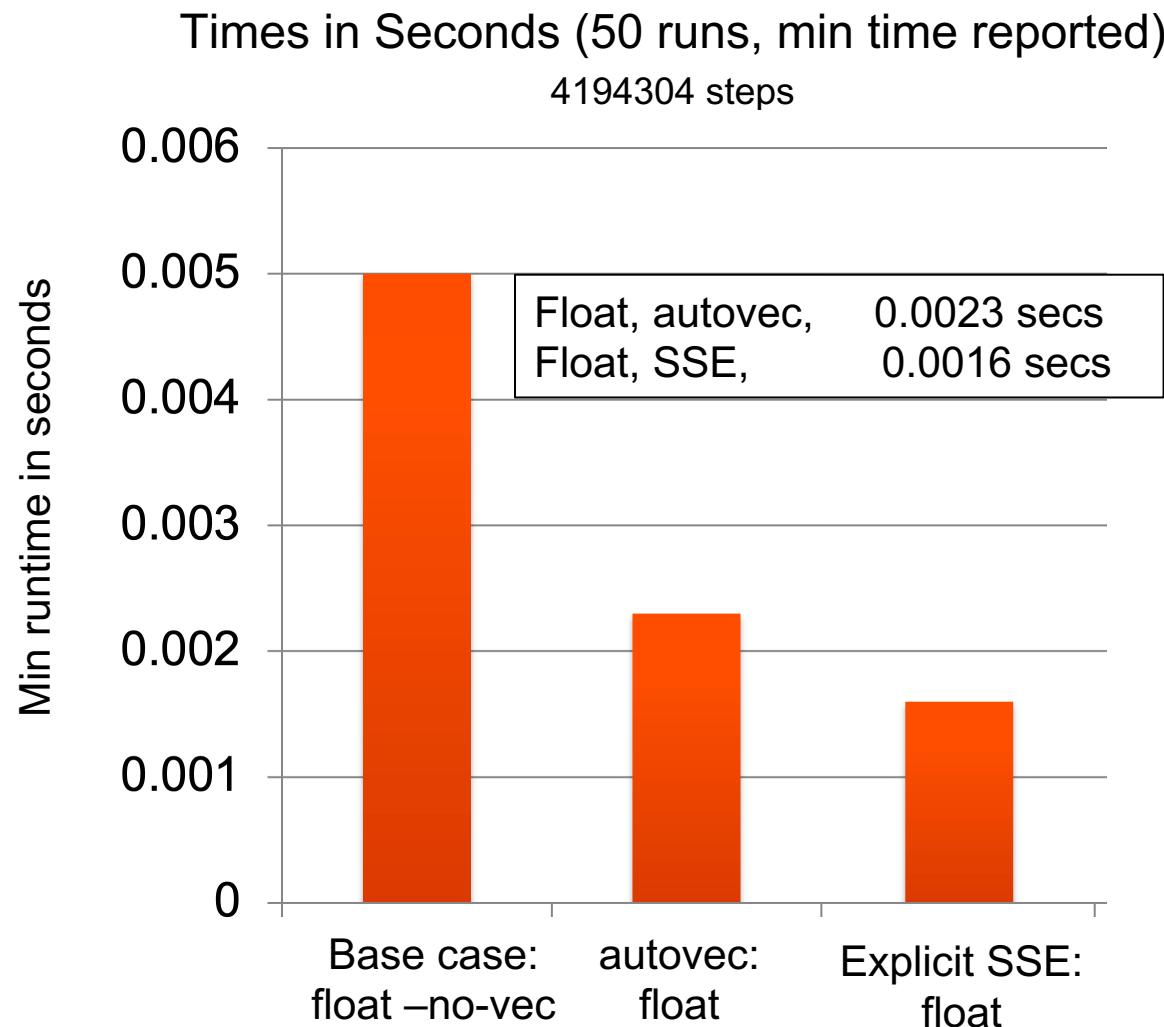
    __m128 ramp = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
    __m128 one = _mm_load1_ps(&scalar_one);
    __m128 four = _mm_load1_ps(&scalar_four);
    __m128 vstep = _mm_load1_ps(&step);
    __m128 sum = _mm_load1_ps(&scalar_zero);
    __m128 xvec; __m128 denom; __m128 eye;

    for (int i=0;i< num_steps; i+=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

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

# The key to high performance

We didn't say much about GPUs, but I want to include these comments here to highlight similarities in optimization for GPUs and CPUs.

## CPU

### Occupancy: keep the back-end busy

- Have enough work to keep all the functional units fully engaged ... order dozens per core
- Use inline-functions and short loop-bodies to keep the Micro-op cache full.
- Minimize expensive ops (such as divide and square root)  
Write code the compiler can vectorize (see next slide).

## GPU

- Have enough work to keep all the functional units fully engaged ... order thousand per GPU
- Algorithms optimized for throughput computing ... Data parallelism replicating blocks of code (kernels) as work-items (CUDA-threads) per GPU

### Memory optimization is the number one priority in maximizing performance

- Organize computation around the cache hierarchy
- Bring data into a cache, and do as much work as you can before it is evicted.
- Avoid indirect addressing (such as  $\text{tmp} = \text{x}[\text{a}[\text{j}+\text{k}]]$ )

- Organize computation around the cache hierarchy ... block work so work-items exploit private memory and work-groups exploit local memory (CUDA shared-memory).
- Memory coalescence: if SIMD-lane  $j$  works on  $\text{a}[\text{j}]$  then SIMD-lane  $(j+1)$  should work on  $\text{a}[\text{j}+1]$ .

### Regular and predictable flows of instructions

- Avoid complex branching. Reduce iCache and iTLB misses.
- Make branches predictable ... Misprediction is expensive.

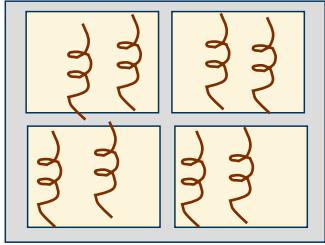
- Branches lead to diverged execution and wastes resources. Aggressively structure algorithms to avoid them. The goal ... converged execution flow

# 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]]`)

# Parallelism: we've covered the “on-node” cases

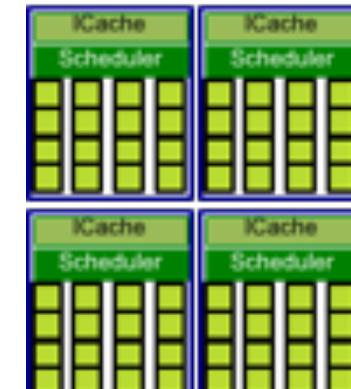
All hardware vendors are in the game ... parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



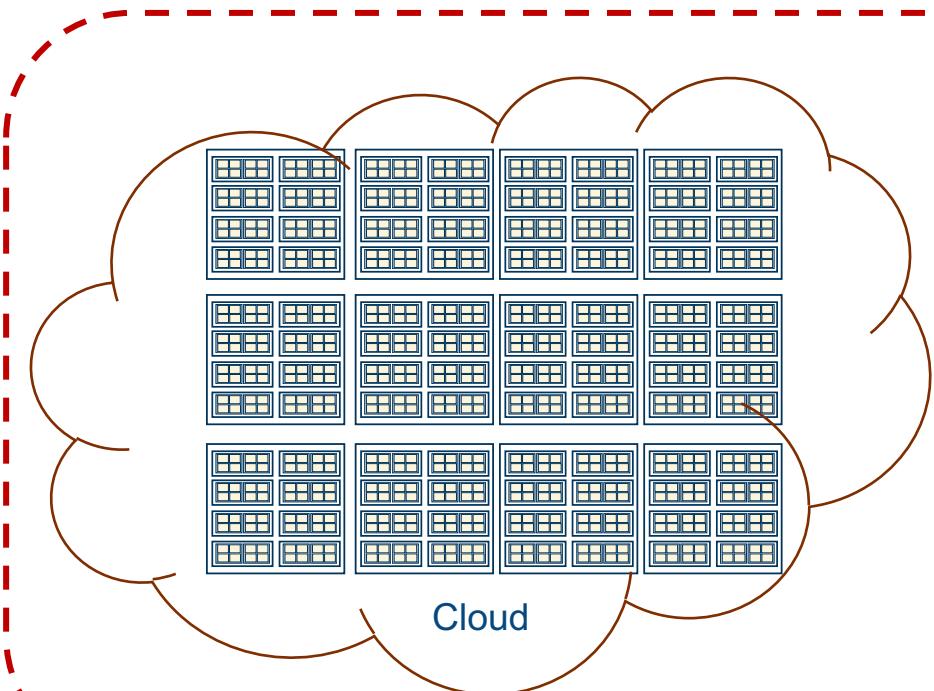
CPU



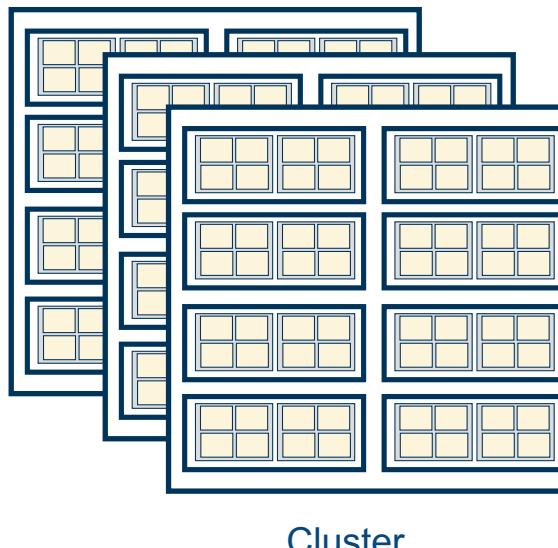
SIMD/Vector



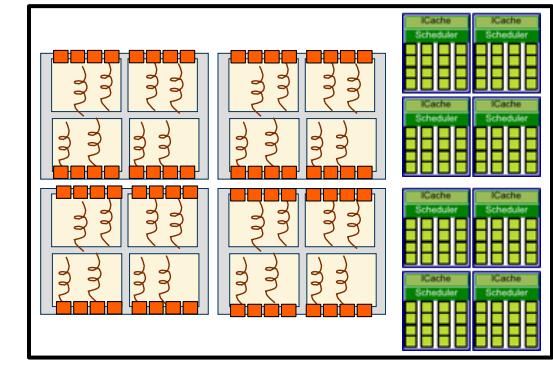
GPU



We'll discuss these later ... after you understand how to write software for the “on-node” cases



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 our time together. Working together, you can master what you need.
- I'll give the closing statement to Sverre Jarp ... an old friend of ESC

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
- **You must get your code to use vectors**
- **You must understand if your ILP is seriously limited by serial code, complex math functions, or other constructs**

10x – 100x

2x, 4x, 8x, 16x

1x – 10x