Intro to Computer Architecture for Computational Scientists

Edward Valeev

Department of Chemistry
Virginia Tech
Blacksburg, VA

Last Updated: June 25, 2015

Lecture Outline

- Why We Must Learn HPC
- The Past: Instruction-Level Parallelism
  - classic serial computer
  - evolution of a scalar processor
  - vector processors
- The Present: Parallel Instruction Streams
  - shared-memory multiprocessors
  - distributed-memory multiprocessors
- The Future: Parallelize or Perish
  - “think” parallelism from the start

Modern Computers: Overview

UPDATE

now

by 2017

your department/campus
~100000 CPU+GPU cores

national resources
10^7-10^8 cores

our department
1024 CPU cores

our campus
10^4 CPU cores
+ 10^5 GPU cores

Oak Ridge, TN
299,008 CPU cores
+ ~400000 GPU cores

my office
12 CPU cores
+ 56 GPU cores

likely to use massively parallel
SIMD hardware like GPUs

your lap
4 CPU cores
+ 24 GPU cores

Modern Processors: x86 family

Intel Haswell (2013)/
Broadwell (2015)

up to ~16 cores
64-bit ("x86-64")

AMD Jaguar (2013)

Intel Sandy Bridge (2011)/
Ivy Bridge (2012)

Intel Nehalem (2008-2011)

Intel Core (2006-2008)

....

AMD Bulldozer (2011)

AMD Barcelona (2007-2011)

AMD K8 (2003-2008)

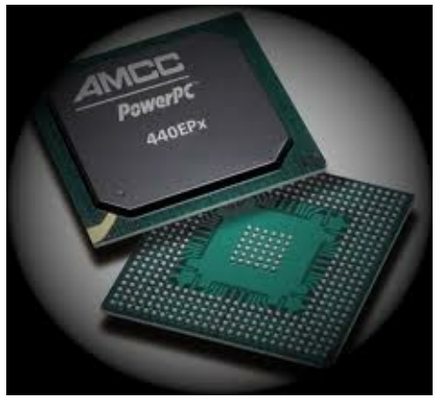
....

Intel 8086 (1978)

1 core
16-bit ("x86-16")

Modern Processors: non-x86 families

PowerPC (PPC)

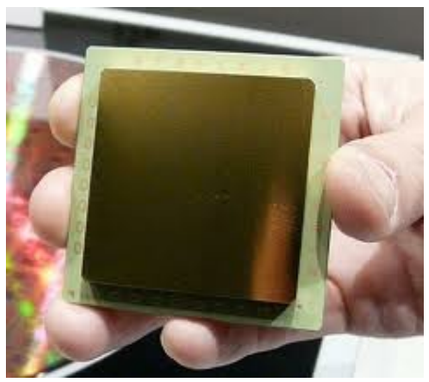


IBM Blue Gene and POWER servers

ARM

THE mobile/embedded

SPARC



Fujitsu K computer and Oracle servers

GPUs

accelerator cards

Modern Computers: Supercomputers

Cray XK7

AMD Bulldozer + Cray Interconnect + NVIDIA Tesla K20X GPU (XK7)

IBM Blue Gene Q

PowerPC + IBM Interconnect

Fujitsu K Computer

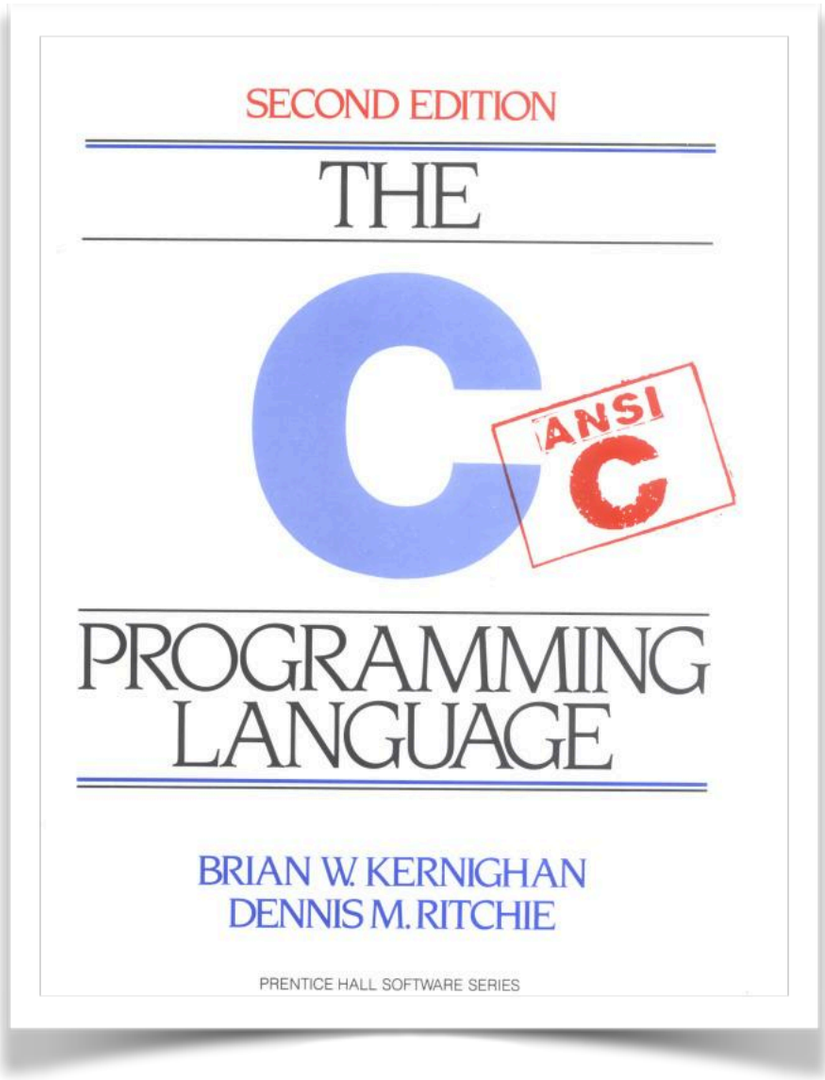
Fujitsu SPARC64 VIIIfx + Fujitsu Interconnect

Generic

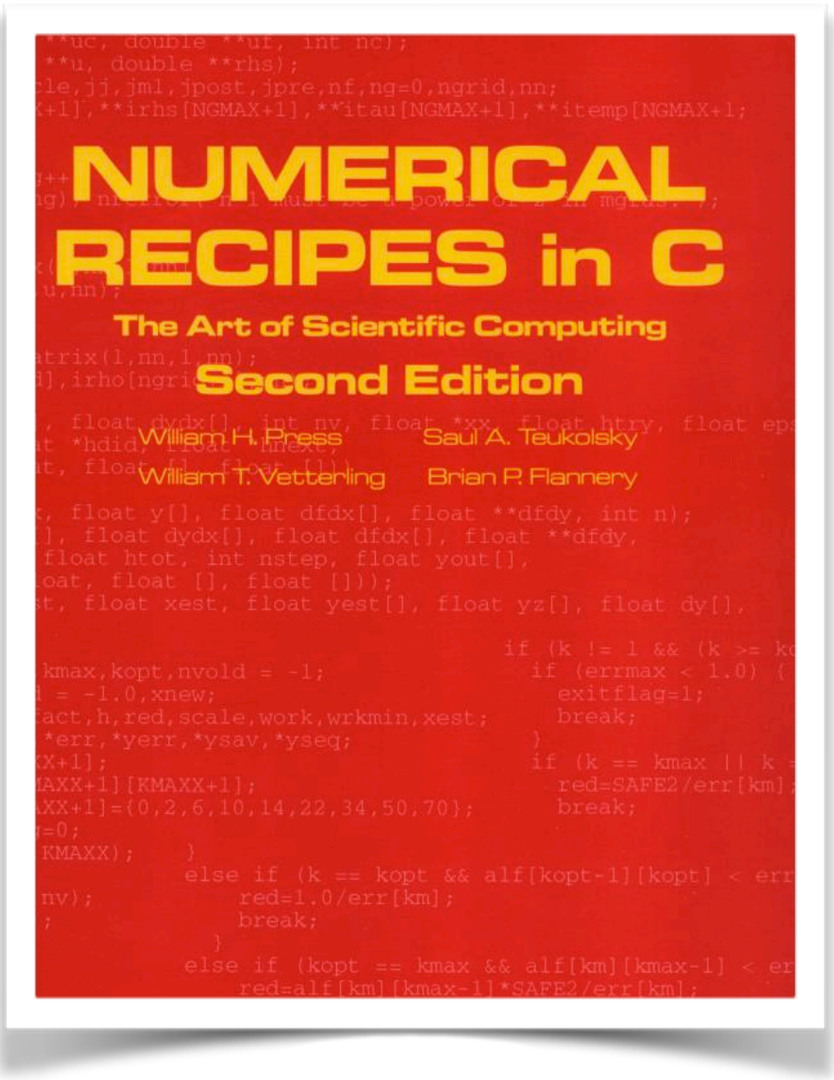
generic x86-64 + generic Infiniband interconnect + optional GPUs

Why We Must Spend Time Learning HPC

Is this not enough?



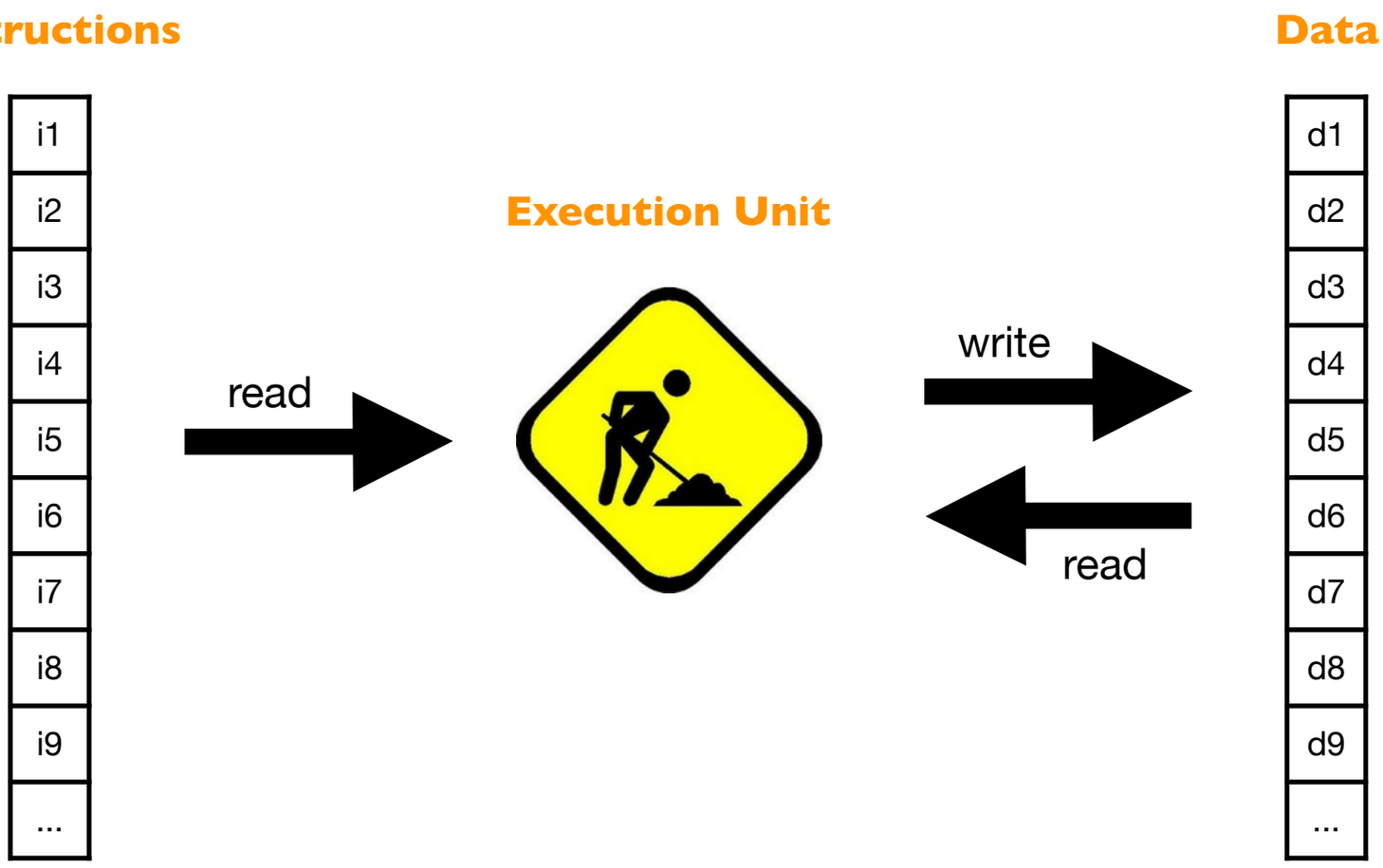
+



Why We Must Spend Time Learning HPC

- Computers are changing (a mini-revolution is underway) $\Longrightarrow$ textbook numerical recipes programs ...
    - ... now use small fraction of the computer's potential (typically 5% or less)
    - ... and this situation will become worse with time
- Parallel hardware gives us more freedom in computation to
    - ... investigate new ideas faster
    - ... treat more realistic systems

Idealized Serial Processor



for example:
i1 does “d8 := d7 + d2”
i2 does “d5 := d8 x d1”
....

- one instruction retired at a time
- each instruction acts on 1 set of data

Performance Model

start with very simple way to think of performance

$$t_{\text{total}} = t_{\text{read}} + t_{\text{execute}}$$

$$t_{\text{read}} = \alpha + \frac{d}{\beta}$$

latency

data size

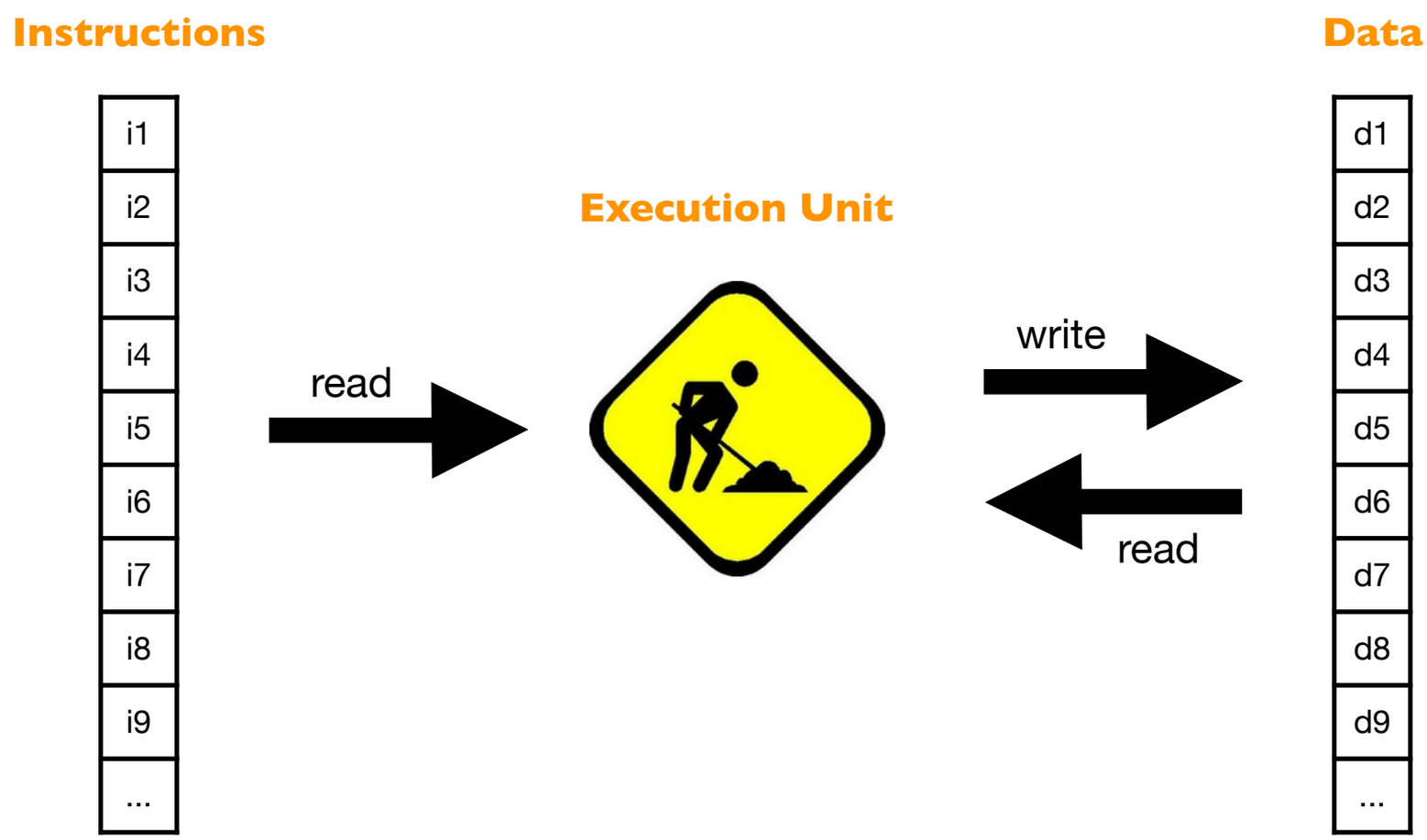
bandwidth

$$t_{\text{execute}} = \frac{1}{\gamma}$$

"instruction bandwidth"

will refine when we know more...

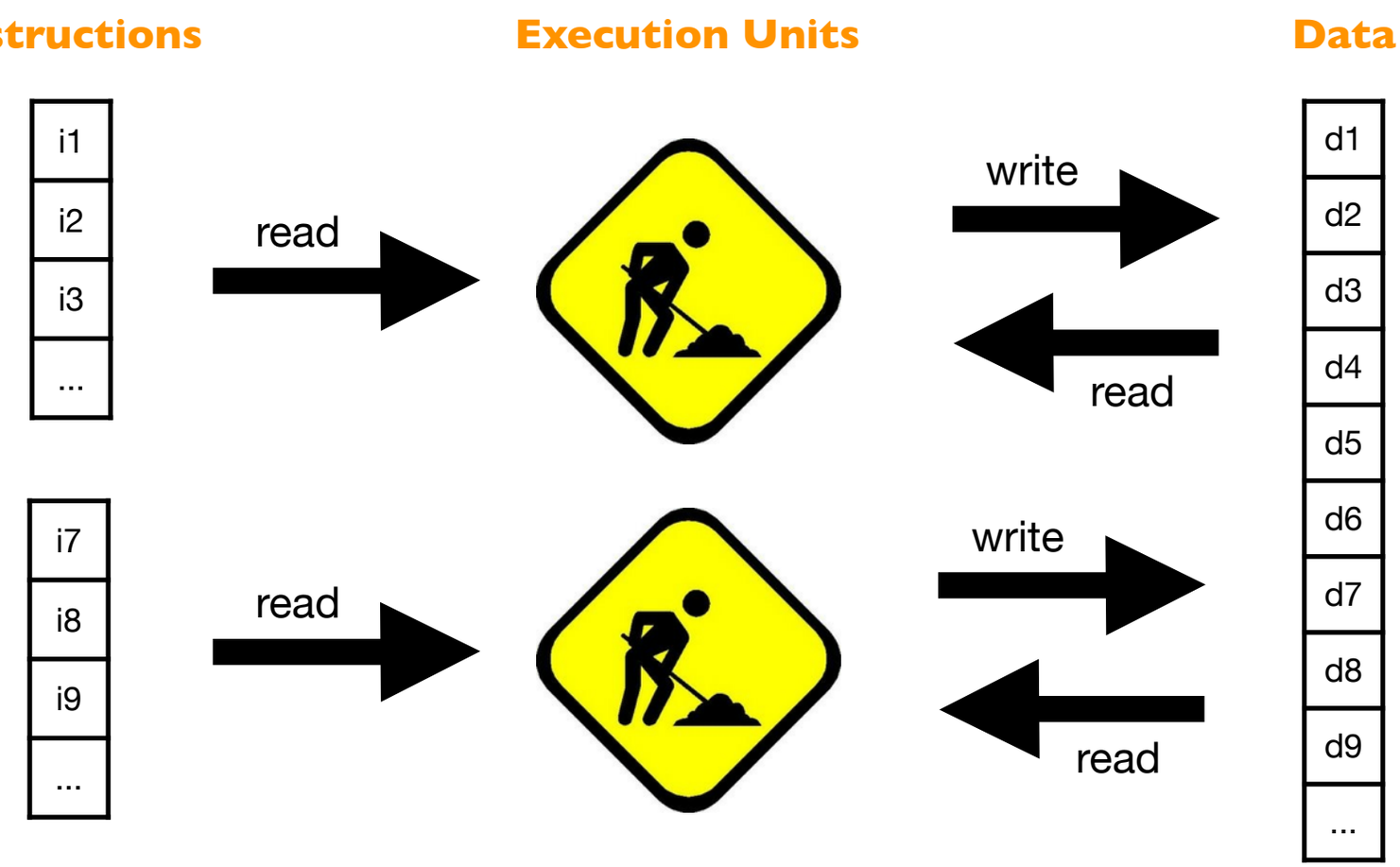
Idealized Data-Parallel Processor



for example:
i1 does “d8 := d6 + d3” and “d9 := d7 + d4”
i2 does “d4 := d11 x d1” and “d5 := d12 x d2”
....

- one instruction retired at a time
- each instruction acts on several (2-16) sets of data
- Single Instruction Multiple Data (SIMD)
- most processors of today and all processors of tomorrow

Idealized Instruction-Parallel Processor



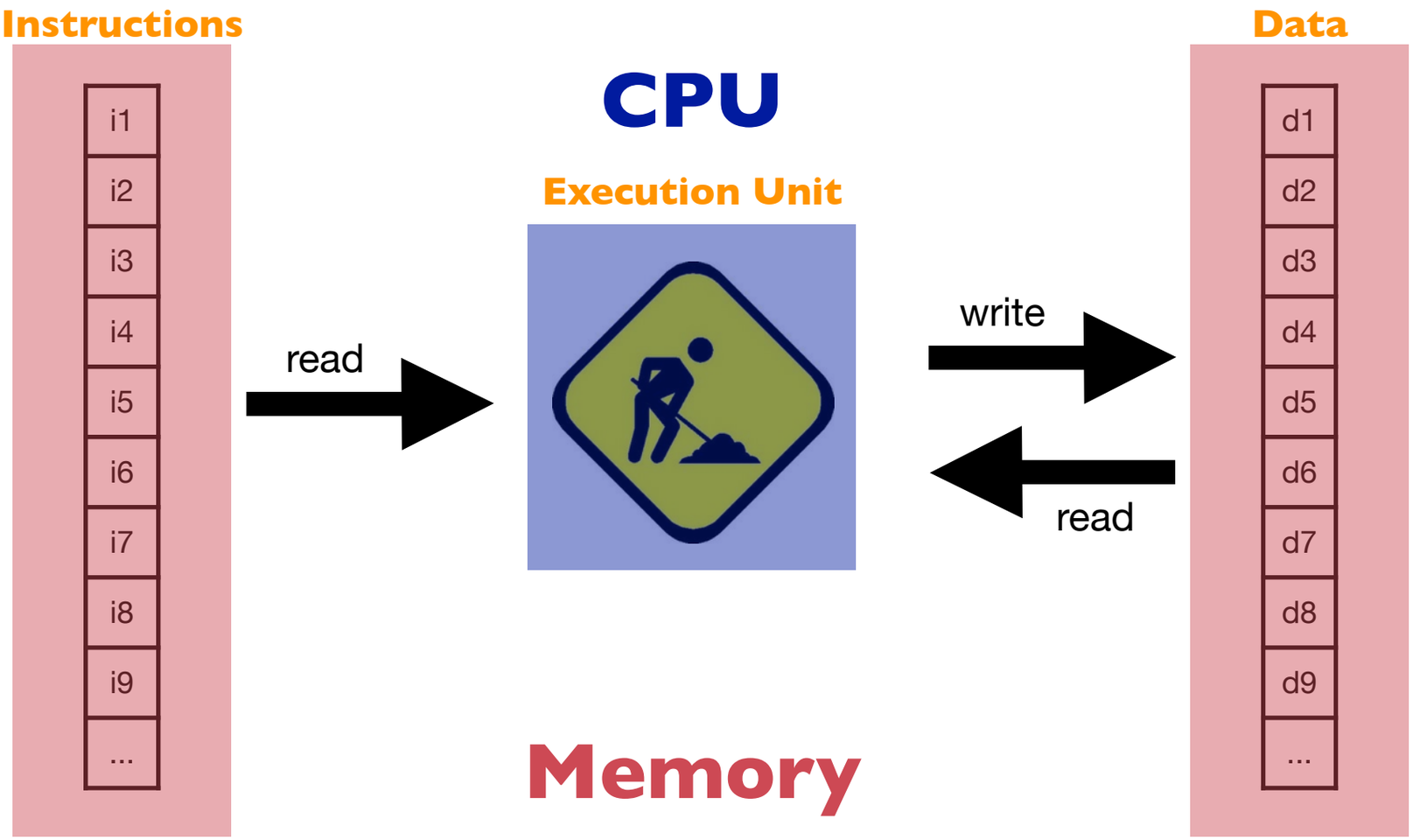
- several (2-32) instructions retired at a time
- each instruction acts on one or several (2-16) sets of data
- Multiple Instructions Multiple Data (MIMD)
- all processors of today and tomorrow

Challenges of Parallel Computing

- **MIMD**
  - load balancing: ensuring that all workers finish roughly at the same time
  - data races: when multiple workers access the same data and at least one access is a write, the result is unpredictable
  - non-deterministic execution: hard to reason about MIMD programs
- **SIMD**
  - data must be organized in a particular way (best when instructions process continuous properly-aligned data chunks)

programming parallel computers will be the main focus of this lecture series, but we must first understand how a computer works in more detail to understand its performance

Simplified Serial Processor



Memory Hierarchy for a “Classic” Computer

Memory Type	Size	Latency, cycles	Location
Registers	1000 B	1	on die
Level 1 Cache	64 KB	3	on die
Level 2 Cache	256 KB - 12 MB	10	on die
Level 3 Cache	4 MB - 18 MB	40	on/off die
Main Memory (DRAM)	256 MB - 4 GB	100	off/on die
Disk	1 TB	10^7	off die
Tape	20 TB	10^{10}	off die

Exercise

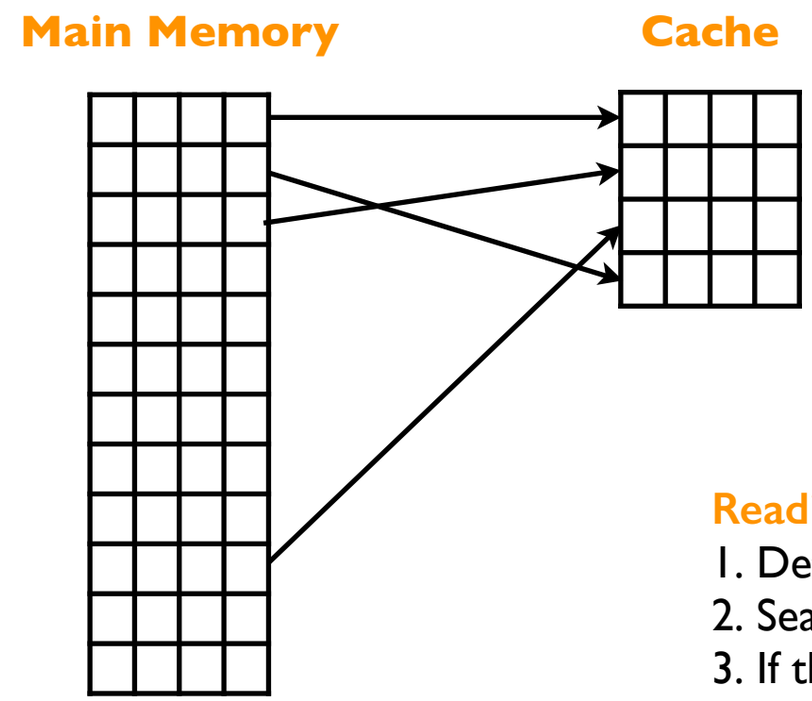
- estimate how quickly can you compute with data in L1, L2, and RAM

- then measure how quickly we can multiply 2 matrices

- analyze

Cache Structure and Cache Misses

Example:
Addresses 0-3
Addresses 4-7
Addresses 8-11
Addresses 12-15
....



Line 0	Tag 0
Line 1	Tag 2
Line 2	Tag 9
Line 3	Tag 1

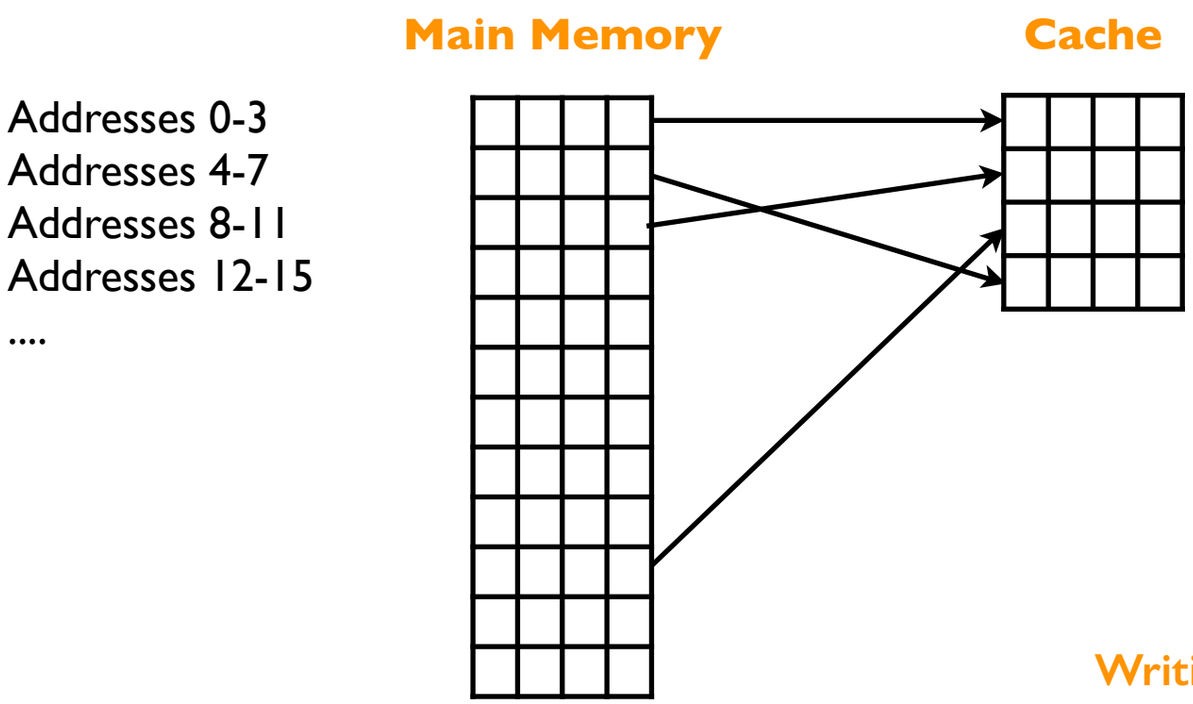
cache lines: 8 - 512 bytes
(Intel Sandy Bridge: 64 bytes)

Reading data

1. Determine the address
2. Search the address among the cache line *tags*
3. If the tag is found, data is in cache, read. DONE
4. If the tag is not found, *cache miss* occurred
5. If cache is full, evict a line (heuristic algorithm, e.g. LRU)
6. Load data into an empty cache line, read. DONE

L1 cache miss costs ~10 cycles
L2 cache miss costs ~100 cycles
....

Cache Structure and Cache Misses



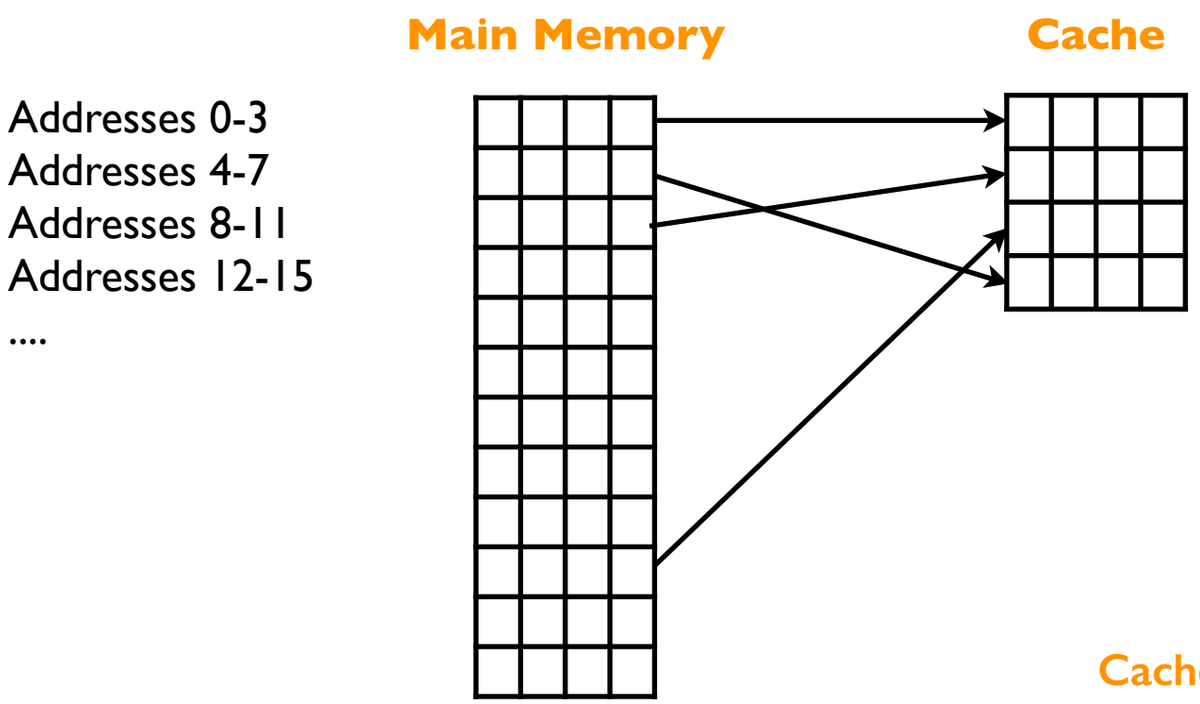
Line 0	Tag 0
Line 1	Tag 2
Line 2	Tag 9
Line 3	Tag 1

cache lines: 8 - 512 bytes
(Intel Sandy Bridge: 64 bytes)

Writing data

1. Similar to reading
2. Whether data in memory is updated immediately is determined by the *write policy: write-through* vs. *write-back*

Cache Structure and Cache Misses



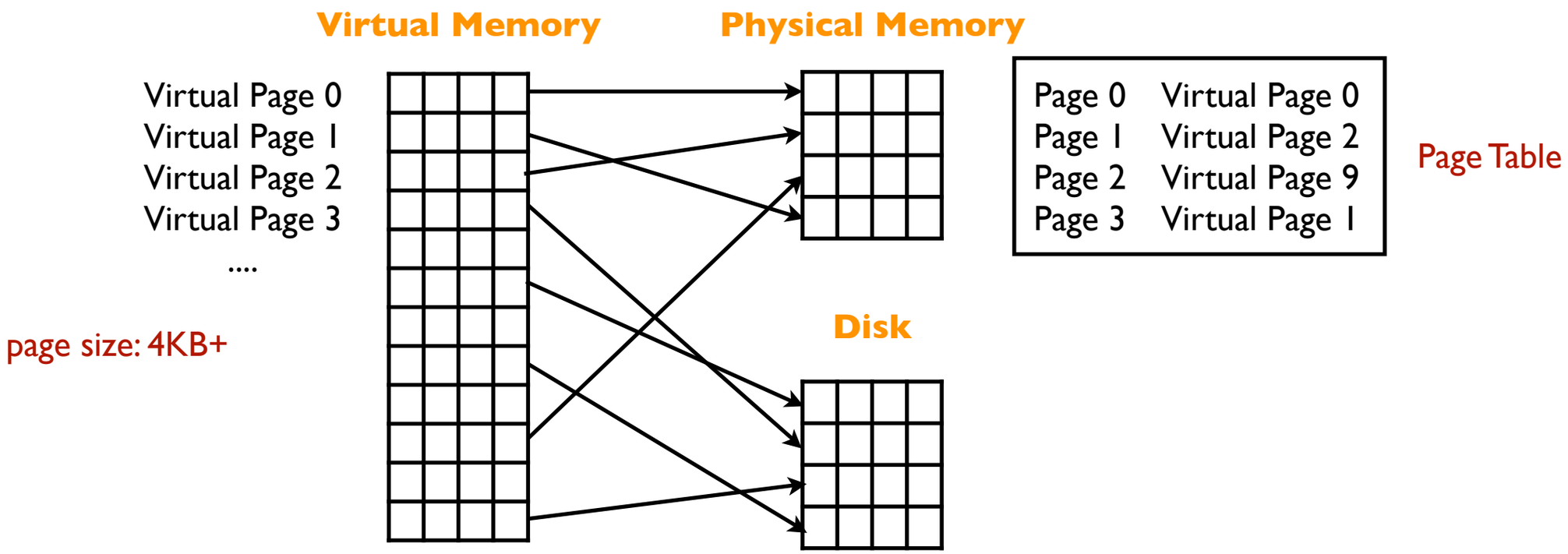
Line 0	Tag 0
Line 1	Tag 2
Line 2	Tag 9
Line 3	Tag 1

cache lines: 8 - 512 bytes
(Intel Sandy Bridge: 64 bytes)

Cache Associativity

1. fully associative: an address can map to any cache line
2. fully mapped: an address can map to one cache line
3. n-way associative: an address can map to n cache lines

Virtual Memory



Address Translation

1. Look up virtual page in *Translation Lookaside Buffer* (TLB, hardware cache for Page Table). If found, return physical address. DONE
2. Generate *Page Fault*. OS searches Page Table. If found, write to TLB. Go to step 1.
3. Load up the needed page from disk to physical memory. Update Page Table. Write to TLB. Go to step 1.

Page fault costs ~1000 cycles

Summary: Memory

- Memory access is the primary bottleneck in many scientific algorithms; some algorithms are inherently memory-bandwidth limited (e.g. Gaussian integrals) -- consider problem reformulation

- Minimize the amount of data your algorithm uses; best if all data fits into L1/L2 cache

- Once computed (hence in registers/cache), data should be used immediately for subsequent computation

- Do not rely on virtual memory mechanism

- Adjacent instructions should access adjacent memory locations

Arithmetic Logic Units (ALU)

Integer Unit
- Addition/subtraction
- Multiplication/division
- Logical operations (AND, OR, XOR)
- Bitshift
- ...

Floating-Point Unit
- Addition/subtraction
- Multiplication/division
- Square root
- ...

often separate units for scalar and SIMD processing

Example: Intel Sandy Bridge ALUs

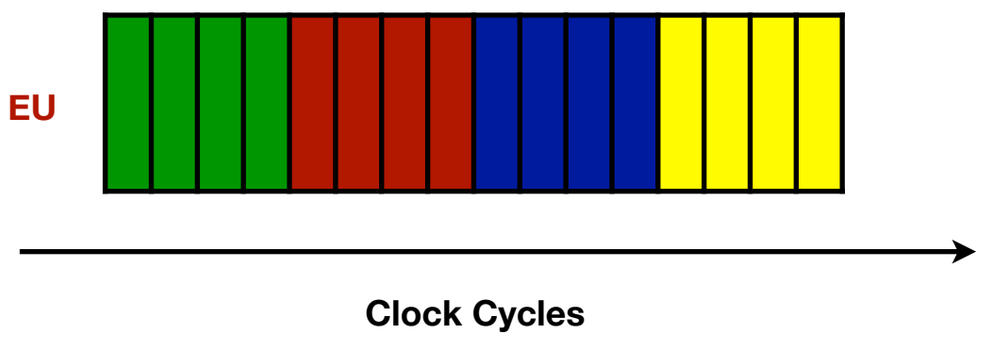
Instruction	Latency
add	1
mul	3-4
div	20-94
bitshift	1

Instruction	Latency
add	3
mul	5
div	21-45
sqrt	21-43

Pipelining

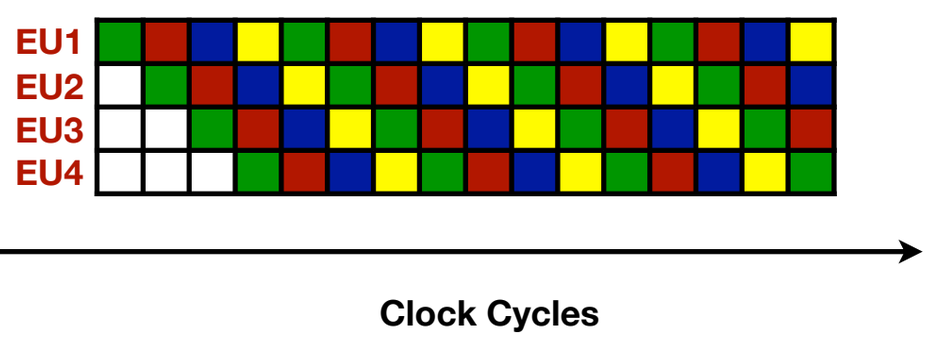
Example: executing four 4-cycle latency instructions in a loop

Non-pipelined execution



throughput = 4 instructions / 16 cycles = 0.25 ipc

Pipelined execution



throughput = 16 instructions / 16 cycles = 1 ipc

Benefits of pipelining

1. Increases throughput by recovering *instruction-level parallelism*
2. Each execution (sub)unit is simpler, hence can be run at higher clock speed

Avoid Pipeline Stalls

Reduced Instruction Set Computer (RISC) generic pipeline

1. Fetch instruction
2. Decode instruction
3. Fetch data
4. Execute instruction
5. Write back result

Branches (if, for, while, do statements) are trouble!
So are cache misses and page faults!
Both can cause pipeline stalls...

Branch prediction and speculative execution

- Specialized hardware (branch predictor) is *key* to preventing stalls!
- Uses heuristic techniques + branch predictor tables to guess whether each branch will be taken or not.
- Until prediction is verified, the code is executed *speculatively*!
- Branch misprediction results in a stall.
- The longer the pipeline -- the more costly is branch misprediction

Out-of-order execution

- Decoded instructions are queued in the *instruction buffer*
- Instructions are not *dispatched* until input data is available.
- Instructions can be dispatched ahead of other instructions in the instruction buffer as long as the sequential data dependencies are not violated.
- Allows to hide the latency of data access!
- The benefit grows with the pipeline length and the speed disparity between memory and processor

Branch prediction and out-of-order execution logic take up a significant portion of a modern CPU

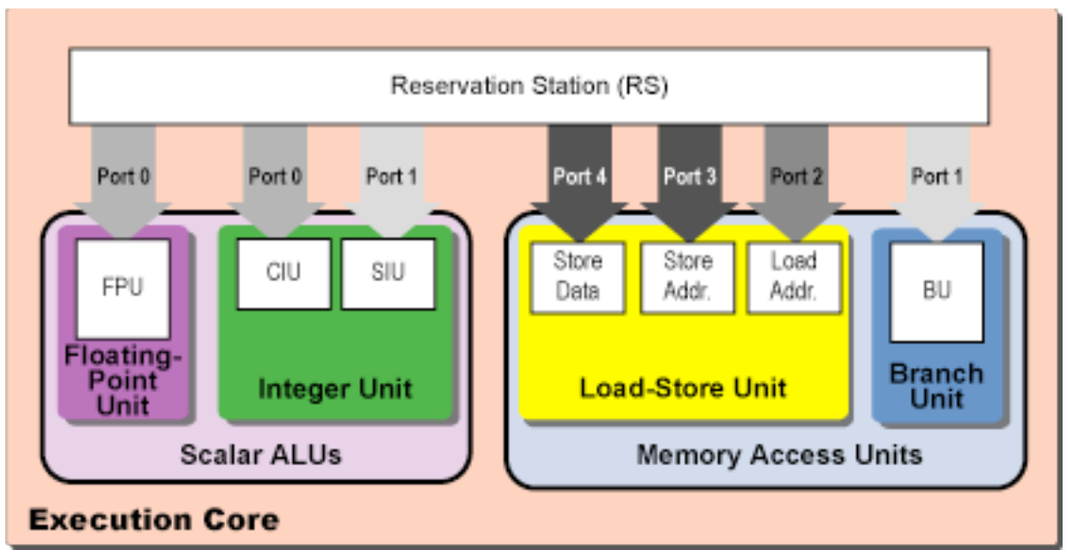
Multiple issue (superscalar) processors

exploit fine-grained parallelism by executing multiple operations simultaneously

```
double a = b+c;
int d = e+f;
int g = h*i;
```

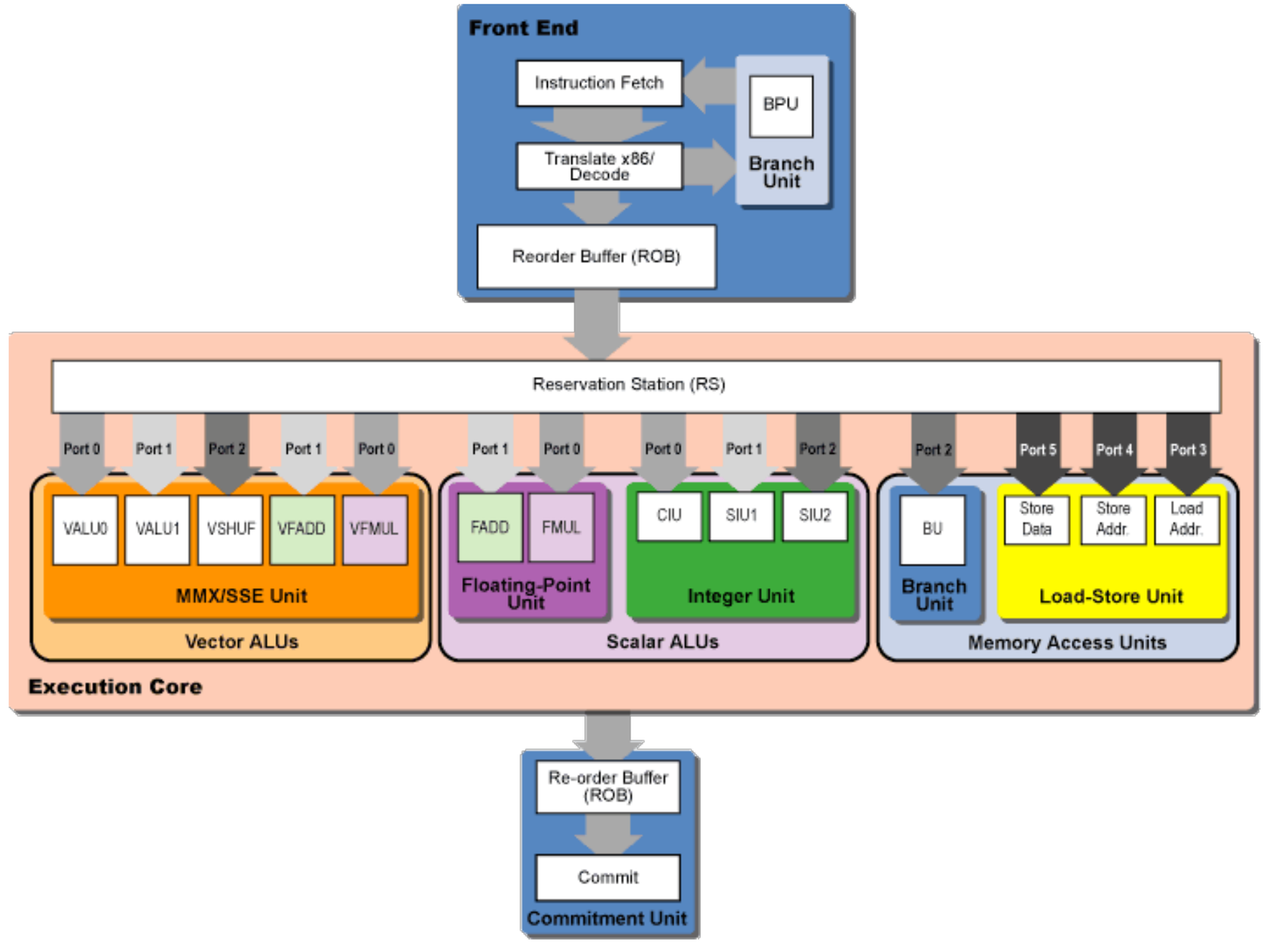
no data dependencies, can be executed simultaneously given enough ALUs

Intel Pentium Pro: superscalar out-of-order core



Multiple issue (superscalar) processors

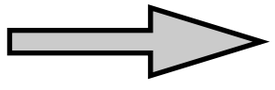
Intel Core: superscalar out-of-order core



Summary: ALU

- Avoid expensive operations (/, sqrt): e.g. i >> 1 better than i/2, or

```
for(int i=0; i<N; ++i) {
  a[i] = b[i] / d;
}
```



```
const double one_over_d = 1.0 / d;
for(int i=0; i<N; ++i) {
  a[i] = b[i] * one_over_d;
}
```

- Avoid branches (if, goto, short for and while loops) in the time-consuming parts of the code

- Avoid data dependencies between adjacent instructions

- Avoid code composed of one kind of instructions, e.g. all floating-point adds; instruction mixes perform better

Vector (data-parallel) processors

scalar code

```
for(int i=0; i<1024; ++i) {
  C[i] = A[i] * B[i];
}
```

vectorized code

```
for(int i=0; i<1024; i+=32) {
  vec_register vA = vec_load(&A[i]);
  vec_register vB = vec_load(&B[i]);
  vec_register vC = vA * vB;
  vec_store(vC, &C[i]);
}
```

Single Instruction Multiple Data (SIMD)

Benefits of vectorization

1. Increases throughput by decreasing the cost of instruction fetching/ decoding, address translation, etc.
2. Regular data access pattern makes caches more effective or redundant
3. Simple loops are easy to auto-vectorize by the compiler

Challenges of vectorization

1. Not all operations can be expressed in vector form
2. **Data must be aligned!!!**
3. May require new algorithms
4. Modern compilers still struggle to generating vector code except from simplest loops

Where vector processing is employed

1. Classic vector machines (Cray, NEC, Fujitsu): lots of vector registers, specialized memory hierarchies **obsoleted by high $cost$**
2. SIMD units on CPUs/GPUs (SSE/AVX on x86, QPX on PPC): fixed short vector lengths

x86 SIMD instruction sets

Streaming SIMD Extensions (SSE)

- 128-bit registers = 2 double-precision numbers
- 16 registers in 64-bit mode
- instruction pattern: a = a op b
- load/store packed data only (no scatter/gather)

introduced in Intel Pentium III; present across x86

Advanced Vector Extensions (AVX)

- 256-bit registers = 4 double-precision numbers
- 16 registers (shared with SSE)
- instruction pattern: c = a op b
- load/store packed data only
- have fused multiply-add (FMA) and gather now (AVX2)

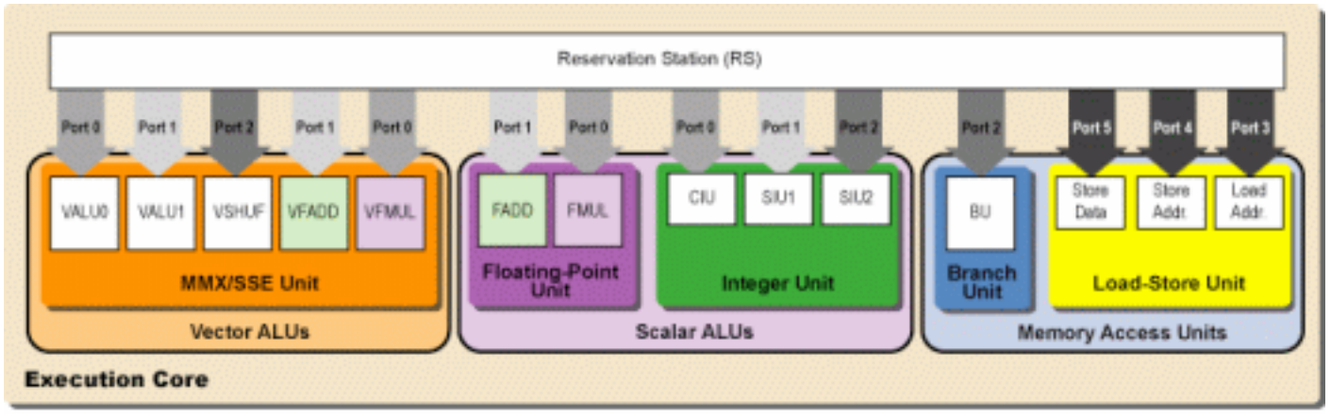
introduced in Intel Sandy Bridge and AMD Bulldozer

future SIMD extensions for x86 to involve 512-bit vector registers
now: Intel Xeon Phi **in 2016**: all x86 and next XeonPhi with AVX-512 instruction set

non-x86 architectures also have SIMD instructions (QPX on Blue Gene/Q, HPC-ACE on K computer, NEON on ARM)

FLOPS as performance metrics of CPU/code

Intel Core2 execution core



VALU0	VALU1	VSHUF	VFADD	VFMUL
-VALU	-VALU	-Vshuff	-VFadd	-VFmul
-Vmul	-Vshift	-Vrecip/ rsqrt	-Vmov	-VFdiv
		-Vmov		-Vsqrt
				-Vmov

FADD	FMUL
-Fadd	-Fmul
	-Fdiv

CIU	SIU1	SIU2
-ALU	-ALU	-ALU
-mul		
-shift		

Theoretical peaks (64 bit)

Scalar: 1 FP add + 1 FP mul every cycle = 2 FLOPs / cycle
Vector: 2 FP add + 2 FP mul every cycle = 4 FLOPs / cycle

DAXPY: Y[i] += a * X[i]

2 FLOPs for every 2 FP loads and 1 FP store
Can only load and store 2 FP numbers per cycle

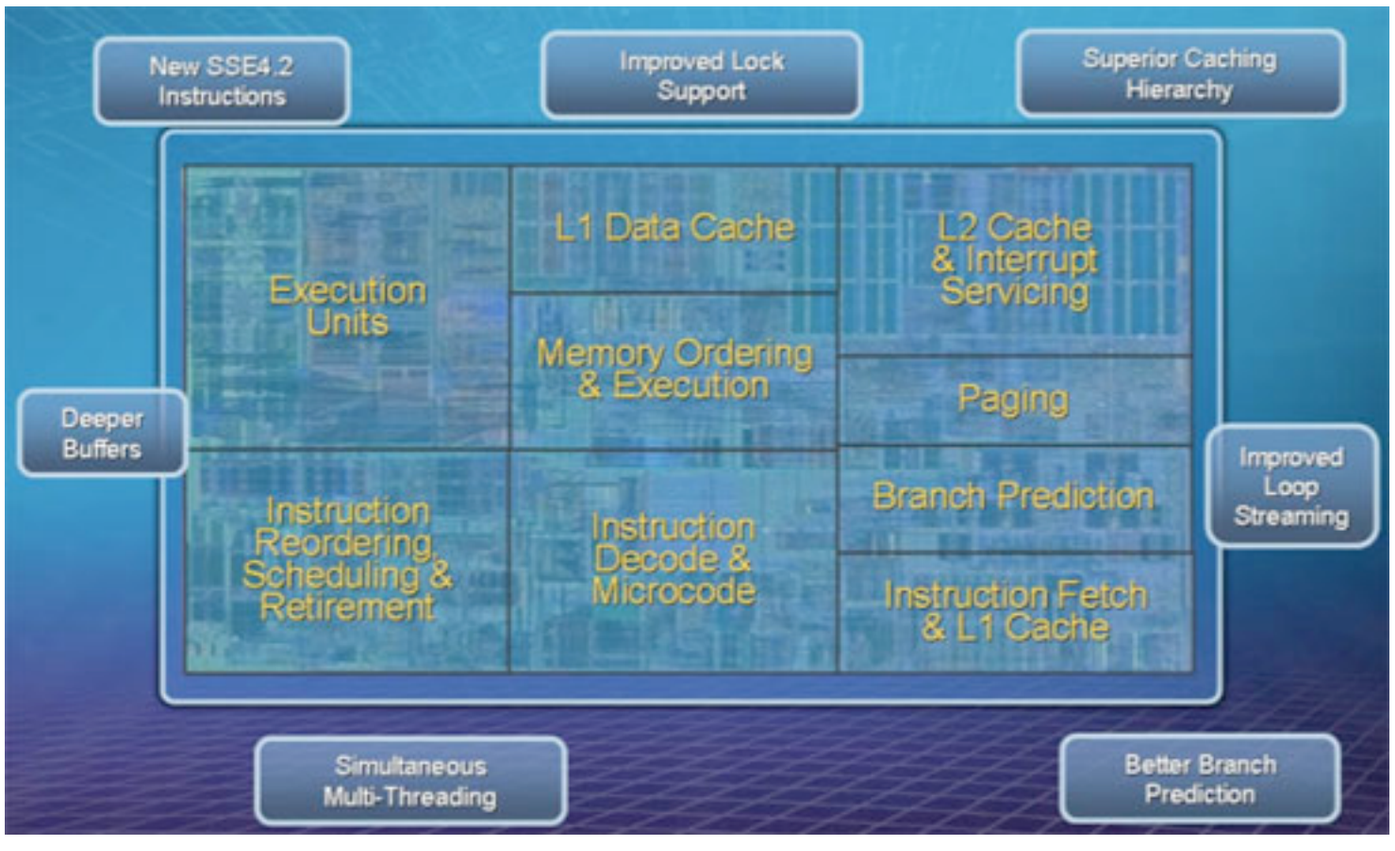
Can't feed the SSE units with enough data

DDOT? DGEMM?

Quiz: where are most transistors used?

Quiz: where are most transistors used?

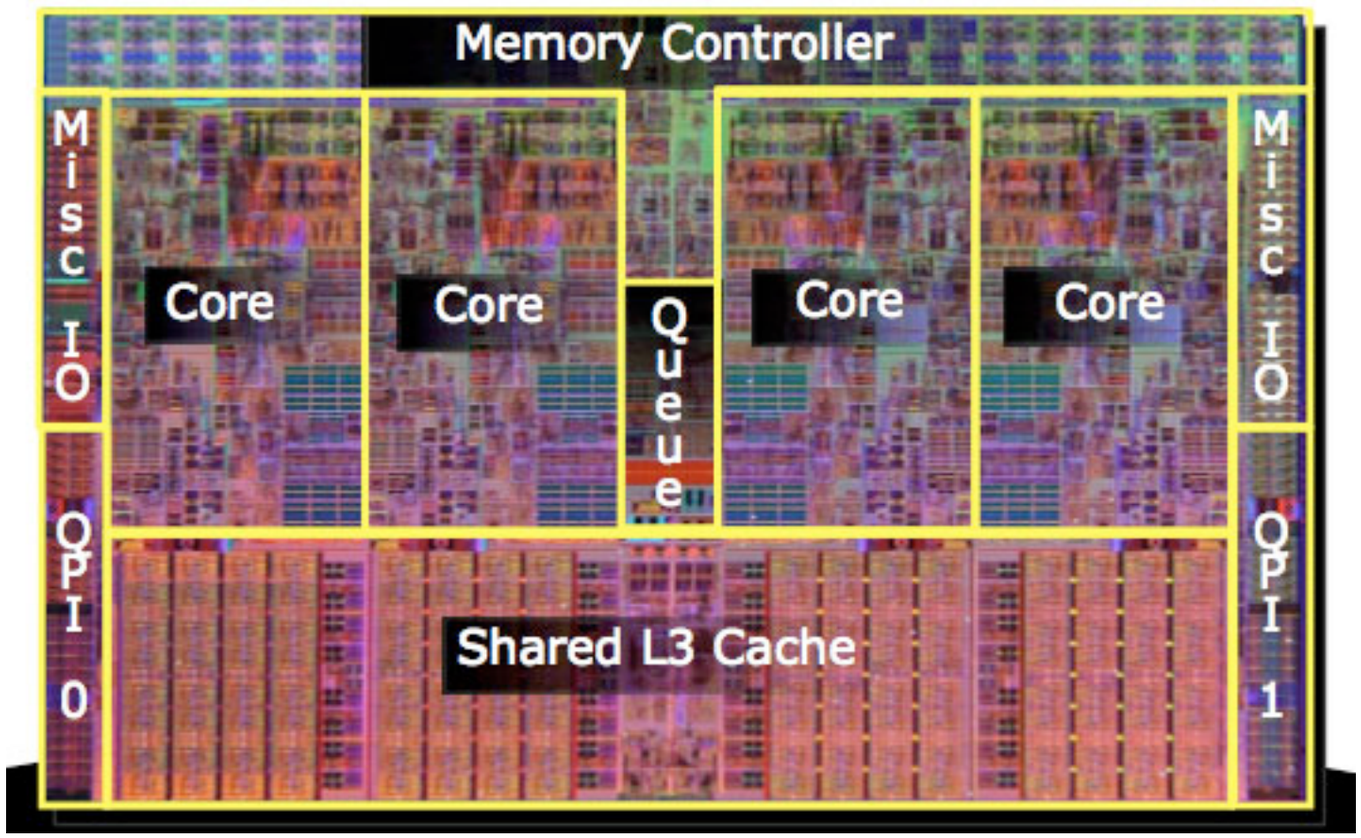
Intel Nehalem *execution core* dye



most transistors are located not in, but around the execution units

Quiz: where are most transistors used?

Intel Nehalem *chip* dye



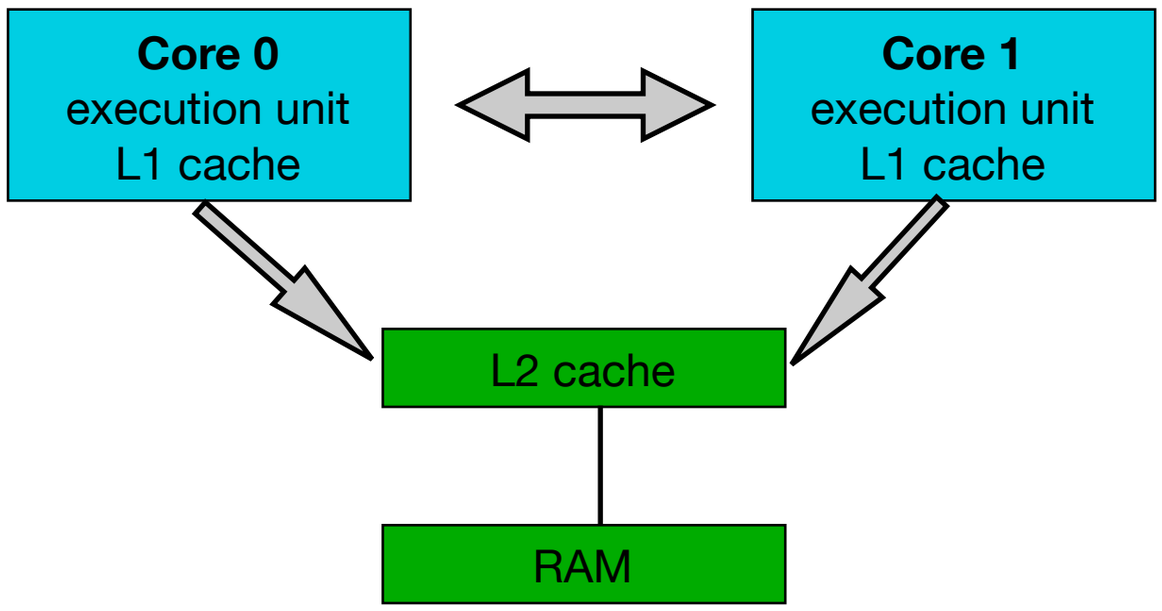
Summary I

- Modern general-purpose cores can execute instructions faster than the data can be fetched. Hardware designers employ a number of techniques to overcome this fundamental problem, such as hierarchical memory, pipelining, out-of-order and multiple-issue execution. Further improvements along this route are unlikely because the CPU lacks the context to optimize/parallelize further.

- Since most scientific computation is not irreducibly serial, further improvement requires optimization/parallelization by the programmer/ compiler duo, i.e. in the software.

Multicore processors

generic 2-core processor

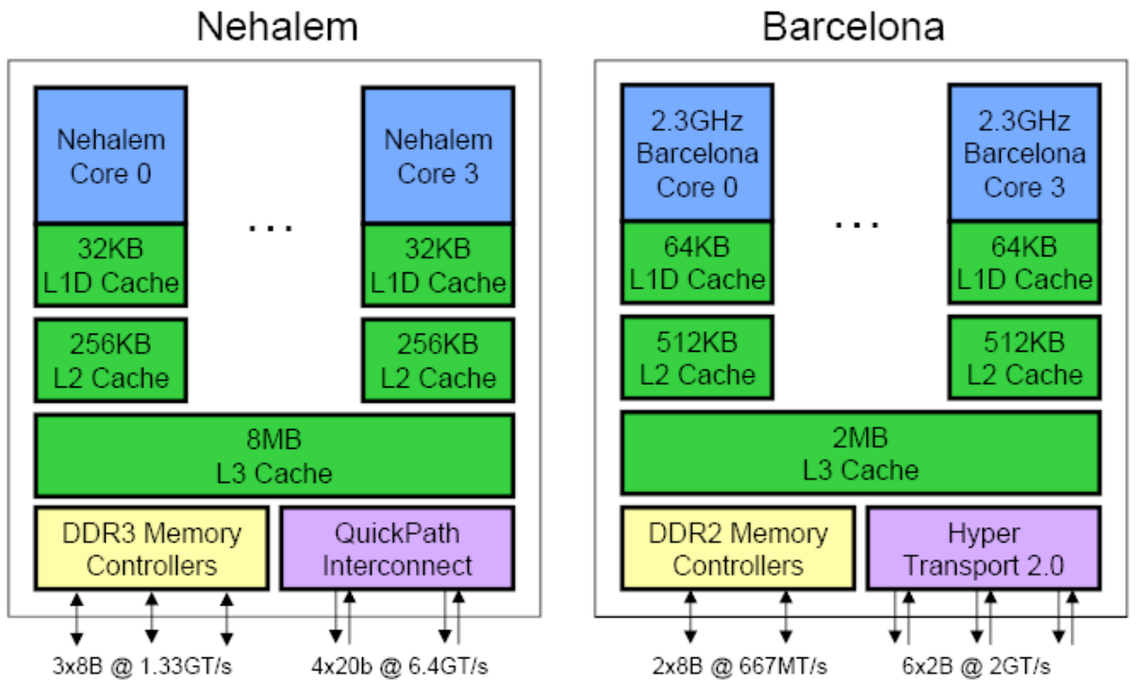


Cache coherence is an additional consideration

- without it memory is read-only
- *snooping coherence*: writes are broadcast to all caches which invalidate the touched cache lines
- *directory-based coherence*: all requests go through a table

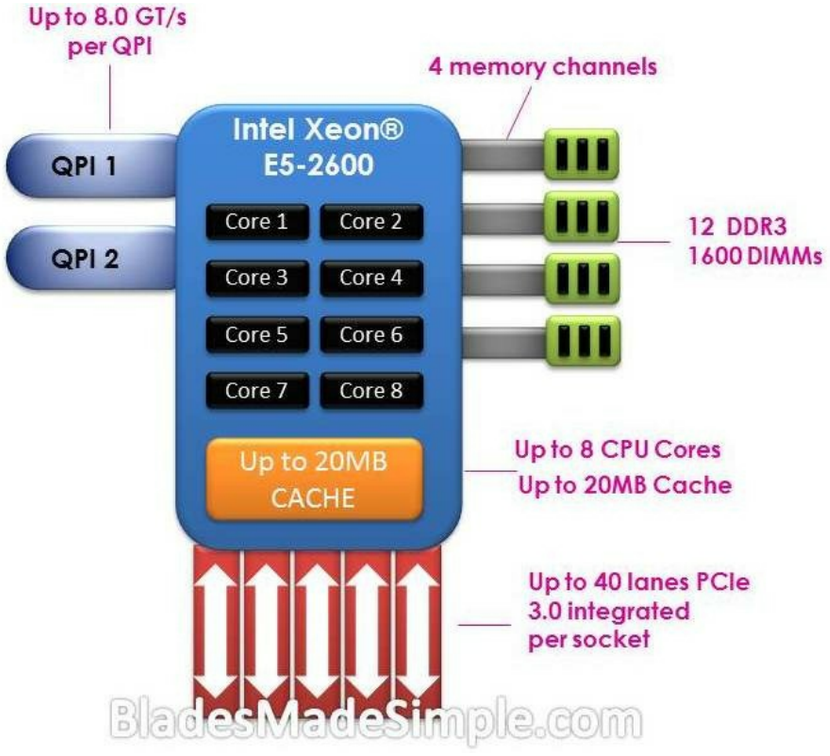
Multicore processors

Intel Nehalem



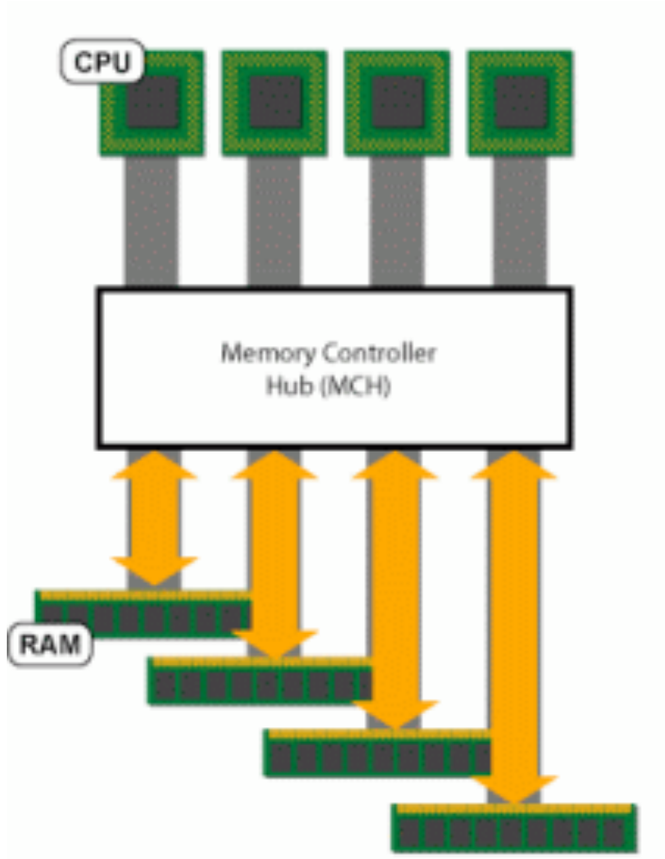
AMD Barcelona

Intel Sandy Bridge



Big lie: “shared-memory” multiprocessors

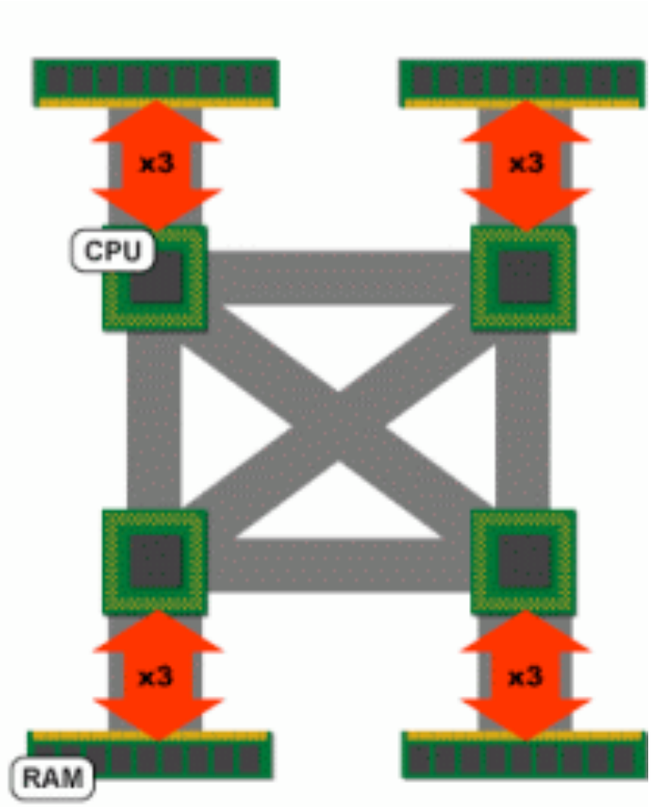
old multiprocessor architecture



INTEL’S CURRENT FOUR-SOCKET PLATFORM

Symmetric Multiprocessing (SMP) architecture

modern multiprocessor architecture

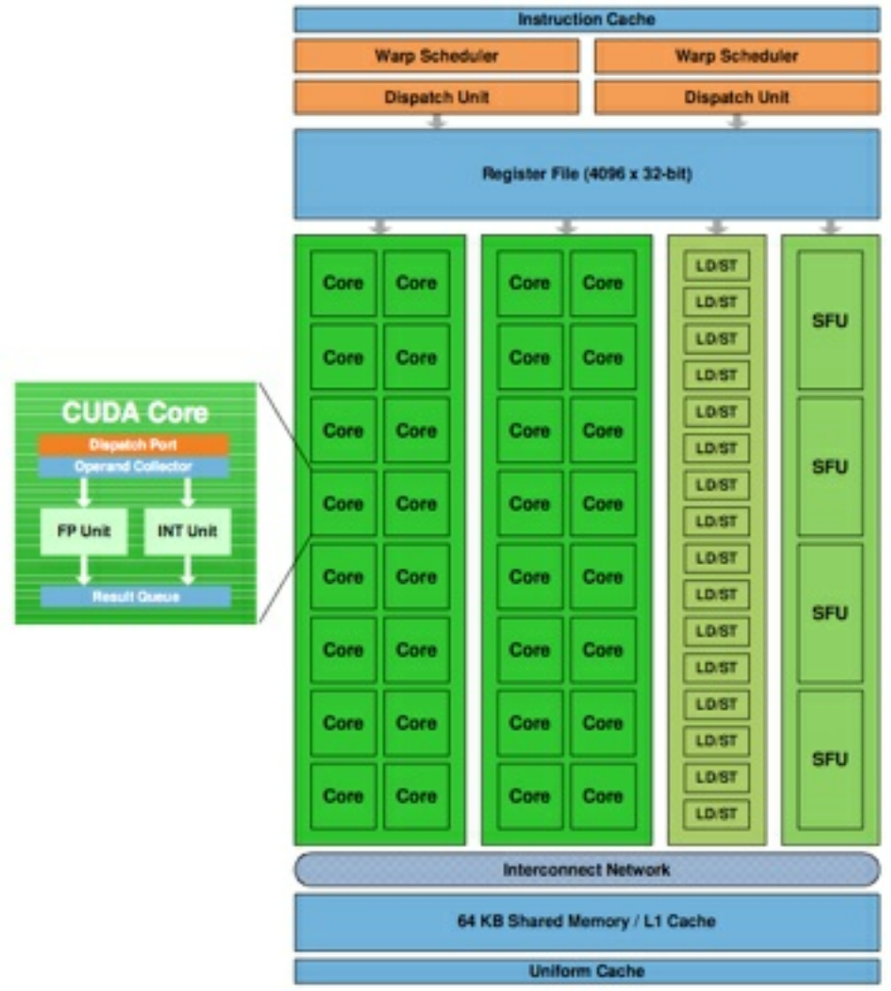


NEHALEM FOUR-SOCKET PLATFORM

Nonuniform Memory Access (NUMA) architecture

Massively many cores on a chip: NVIDIA Fermi architecture

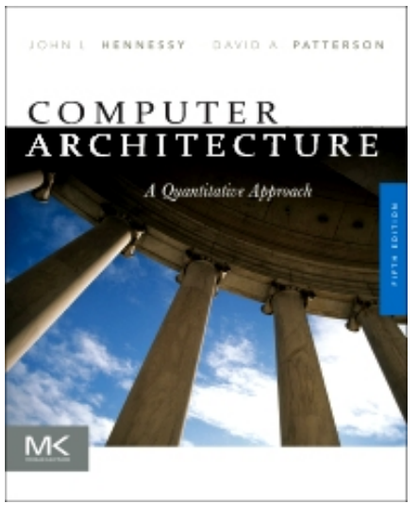
- many (up to 32) cores, ~1 GHz
- 16-wide SIMD units
- little cache relative to CPU analogs
- huge memory bandwidth: O(100 GB/s)
- memory latency hidden by having **many** (1000s) instructions in-flight at a time -- someone will always have work to do
- bandwidth to the main (host) computer is low -- must compute everything on the GPU card
- specialized programming models: CUDA (specific to NVIDIA) and OpenCL (cross-platform)



NVIDIA Kepler architecture is an improved version of this

Comparing GPU vs. CPU (around 2009-10)

Data from Hennessy & Patterson *Computer Architecture: A Quantitative Approach* (5th ed), Morgan Kaufmann (2011)



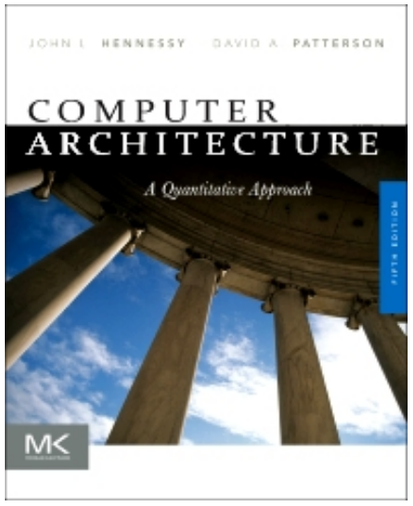
	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TSMC 65 nm	TSMC 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3030 M	2.0	4.4
Memory bandwidth (GBytes/sec)	32	141	177	4.4	5.5
Single-precision SIMD width	4	8	32	2.0	8.0
Double-precision SIMD width	2	1	16	0.5	8.0
Peak single-precision scalar FLOPS (GFLOP/Sec)	26	117	63	4.6	2.5
Peak single-precision SIMD FLOPS (GFLOP/Sec)	102	311 to 933	515 or 1344	3.0–9.1	6.6–13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused multiply-adds)	N.A.	(622)	(1344)	(6.1)	(13.1)
(Rare SP dual issue fused multiply-add and multiply)	N.A.	(933)	N.A.	(9.1)	--
Peak double-precision SIMD FLOPS (GFLOP/sec)	51	78	515	1.5	10.1

Figure 4.27 Intel Core i7-960, NVIDIA GTX 280, and GTX 480 specifications. The rightmost columns show the ratios of GTX 280 and GTX 480 to Core i7. For single-precision SIMD FLOPS on the GTX 280, the higher speed (933) comes from a very rare case of dual issuing of fused multiply-add and multiply. More reasonable is 622 for single fused multiply-adds. Although the case study is between the 280 and i7, we include the 480 to show its relationship to the 280 since it is described in this chapter. Note that these memory bandwidths are higher than in Figure 4.28 because these are DRAM pin bandwidths and those in Figure 4.28 are at the processors as measured by a benchmark program. (From Table 2 in Lee et al. [2010].)

GPUs have more (and more power-efficient) cores and higher memory bandwidth than CPUs
but programming GPUs is more difficult

Comparing GPU vs. CPU (around 2009-10)

Data from Hennessy & Patterson *Computer Architecture: A Quantitative Approach* (5th ed), Morgan Kaufmann (2011)



memory-bandwidth-limited algorithms (BLAS-1, e.g. DAXPY) closer to peak on GPUs than CPUs

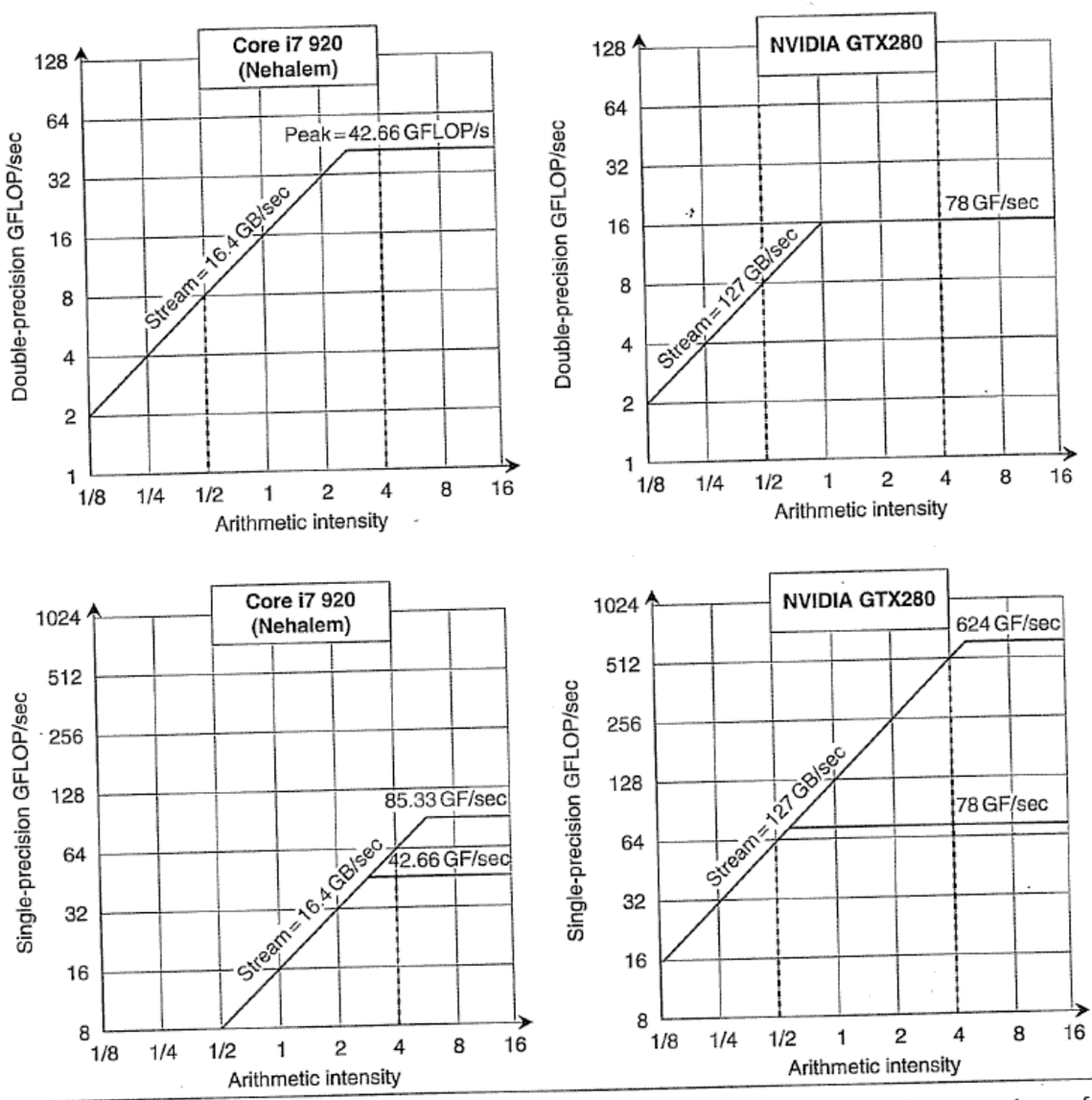
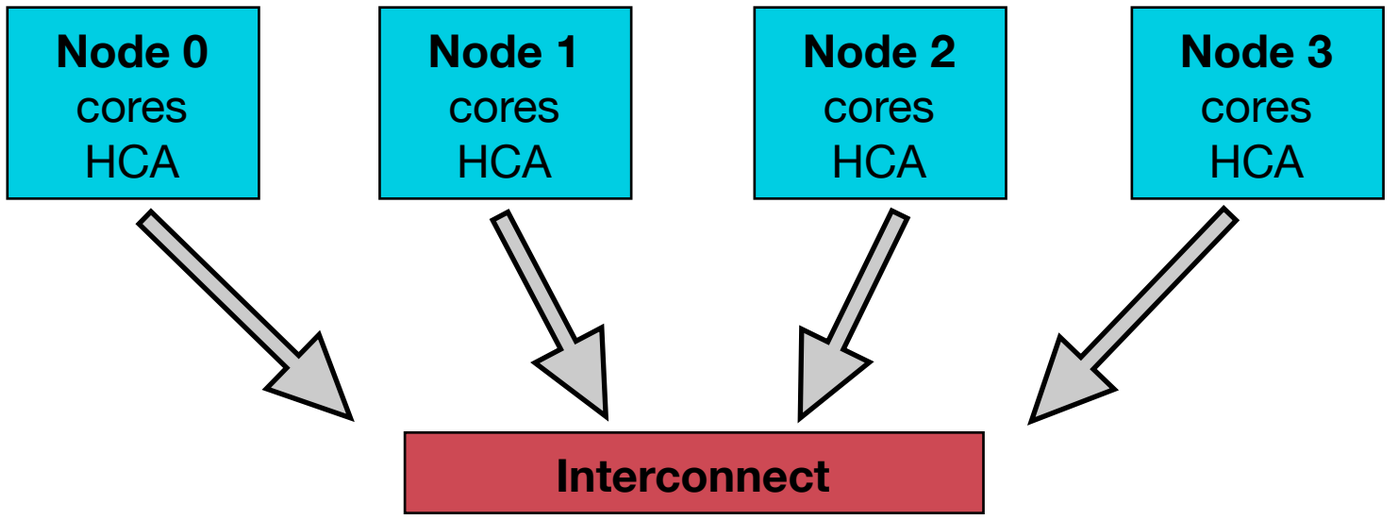


Figure 4.28 Roofline model [Williams et al. 2009]. These rooflines show double-precision floating-point performance in the top row and single-precision performance in the bottom row. (The DP FP performance ceiling is also in the bottom row to give perspective.) The Core i7 920 on the left has a peak DP FP performance of 42.66 GFLOP/sec, a SP FP peak of 85.33 GFLOP/sec, and a peak memory bandwidth of 16.4 GBytes/sec. The NVIDIA GTX 280 has a DP FP peak of 78 GFLOP/sec, SP FP peak of 624 GFLOP/sec, and 127 GBytes/sec of memory bandwidth. The dashed vertical line on the left represents an arithmetic intensity of 0.5 FLOP/byte. It is limited by memory bandwidth to no more than 8 DP GFLOP/sec or 8 SP GFLOP/sec on the Core i7. The dashed vertical line to the right has an arithmetic intensity of 4 FLOP/byte. It is limited only computationally to 42.66 DP GFLOP/sec and 64 SP GFLOP/sec on the Core i7 and 78 DP GFLOP/sec and 512 DP GFLOP/sec on the GTX 280. To hit the highest computation rate on the Core i7 you need to use all 4 cores and SSE instructions with an equal number of multiplies and adds. For the GTX 280, you need to use fused multiply-add instructions on all multithreaded SIMD processors. Guz et al. [2009] have an interesting

Distributed-memory multiprocessors

Generic distributed-memory multiprocessor

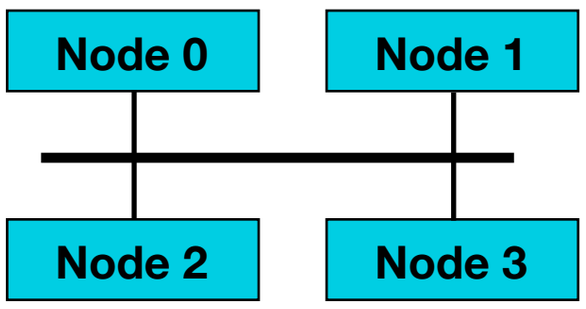


Distributed- vs. shared-memory

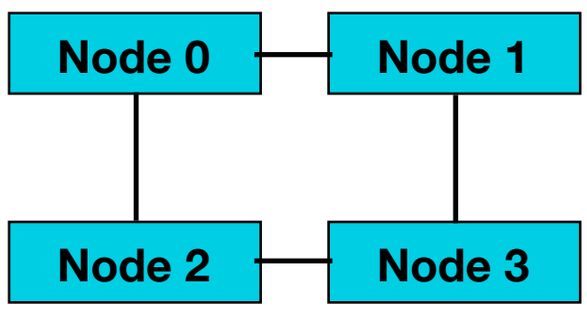
- easier to increase the size of a distributed-memory machine, no complicated memory coherence hardware necessary
- programming shared-memory machines can be deceptively similar to serial
- both bring up new issues in software development

Interconnect networks

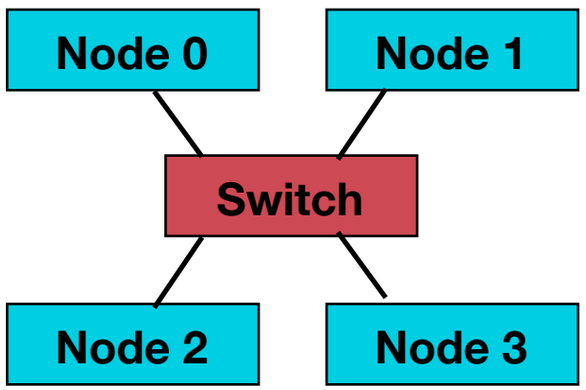
Bus



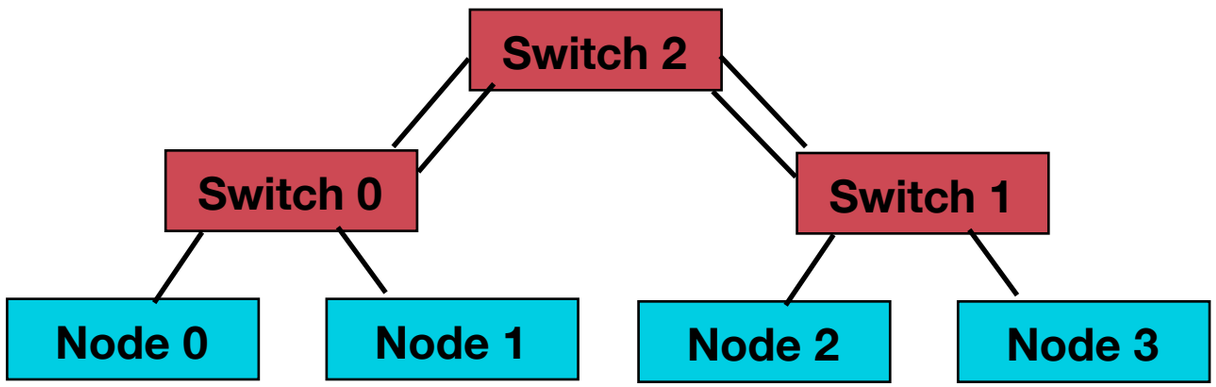
Ring



Crossbar



Fat tree



Modern interconnect technologies

Name	Bandwidth, MB/s	Latency, microsec	Cost
GigE	70-80	40-50	$
Infiniband	700-4000	2-5	$$
Cray Gemini	20000	1	$$$

Summary and Outlook

- Code optimization for even 1 “serial” core is nontrivial and best left to experts. Use vendor/expert-provided libraries (BLAS, MKL). Must be aware of general issues (memory hierarchy) to be at least competent.

- Limits of instruction-level parallelism push HPC towards data and instruction-stream parallelism. The responsibility for recovering this type of parallelism rests squarely on the programmer/compiler duo.

- Future outlook

    - more slower+simpler cores
    - more data per instruction (wider SIMD)
    - hybrid architectures with deep memory hierarchies

Questions