Intro to Computer Architecture for Computational Scientists

Edward Valeev

Department of Chemistry
Virginia Tech
Blacksburg, VA

Last Updated: August 5, 2019

Lecture Outline

- Why Computational Scientists Need To Understand Computer Architecture
- Instruction-Level Parallelism
  - classic serial computer
  - evolution of a scalar processor
  - vector processors
- Parallel Instruction Streams
  - shared-memory multiprocessors
  - distributed-memory multiprocessors
- The Lesson: Parallelize or Perish
  - “think” parallelism from the start

Why We Care About Code Performance

and

are finite

Why We Care About Code Performance

- Faster code = faster checking of hypotheses, shorter research feedback loop, quicker path to Ph.D.
- Faster code = can test ideas on more realistic systems
- Faster code = makes new types of simulations feasible (e.g. ab initio dynamics)
- Faster code = aesthetically pleasing

Ability to reason about code performance is our main goal

Why Reason About Code Performance?

- “my calculation takes *too long*: is the problem the quality of implementation, an algorithmic choice, or the method itself?”

- “will the new method/algorithm be competitive with an existing method/ algorithm?”

- “how can I correctly compare the performance of my *pilot* implementation of an improved method/algorithm against a published benchmark of a production reference implementation”

- “how will changes in the hardware affect methods in my domain?”

- etc.

Projected Outcomes of This Lecture

- **Key lessons**:
  - Serial code performance on modern hardware depends on many factors (hardware characteristics, compiler, code traits)
  - To reason about code performance must build performance models
    - Modern hardware and compilers do pretty amazing things under the hood to create an illusion that the code executes on the idealized abstract serial machine.
- **Most reasonable reaction**: “I will stick to using high-performance kernels produced by experts: Modern hardware + compilers are just too complicated and I would rather do science.”
- **However**: even basic ability to reason about the code will make you a your code better and make you a better scientist! So pay attention.

How We Can Improve Code Performance

- **Measure**: use a specialized tool (profiler) or instrument the code for performance assessment (Mon AM = now)
- **Reason**: build and validate a performance model (Mon AM)
- Consider **alternative algorithms/methods**
- **Parallelize**
  - Intranode: SIMD (Mon PM), OpenMP (Tue AM), specialized extensions for accelerators (CUDA, OpenMP offload) (Wed)
  - Internode: Message Passing (MPI), etc. (Tue PM)
  - Hybrid = MPI+X (Tue PM)
- **Optimize serial (single-thread) performance** (Mon AM+PM)

Measuring Code Performance

- **Nonintrusive**
  - `time <cmd>` — overall execution time
  - profiling tools: GNU profiler (gprof), Intel VTune Amplifier, Xcode Instruments — stochastic sampling of execution time (per-function and even per-line)
- **Intrusive**
  - High-res timer around the code block
  - microbenchmarking harnesses:
    - Google Benchmark: https://github.com/google/benchmark
    - Celero: https://github.com/DigitalInBlue/Celero
    - picobench: https://github.com/iboB/picobench
  - hardware counters:
    - PAPI: https://icl.utk.edu/papi/

Measuring Code Performance: Exercise

- In `Summer-School-Materials/Intro`
  - simple build to warm up (assumes Intel stack): `make; ./run_axpy.pl`
  - profiling build: add `-pg -g` to `CXXFLAGS` and `-pg` to `LDFLAGS`
  - run: `./gemm 512 10000 blas` , this produces binary file `gmon.out` with the execution trace
- generate human-readable profile: `gprof gemm gmon.out`

Example Code Profile

here's the *flat profile*

```
[molssi000@sn-mem Intro]$ gprof ./gemm gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 86.37      0.19     0.19       20     9.50     9.50  Eigen::internal::gebp_kernel<double, double, long, 4, 4, false,
false>::operator()(double*, long, double const*, double const*, long, long, long, double, long, long, long, long, double*)
  9.09      0.21     0.02       10     2.00    22.00  Eigen::Map<Eigen::Matrix<double, -1, -1, 1, -1, -1>, 0, Eigen::Stride<0, 0>
>& Eigen::MatrixBase<Eigen::Map<Eigen::Matrix<double, -1, -1, 1, -1, -1>, 0, Eigen::Stride<0, 0> >
>::operator=<Eigen::GeneralProduct<Eigen::Map<Eigen::Matrix<double, -1, -1, 1, -1, -1> const, 0, Eigen::Stride<0, 0> >,
Eigen::Map<Eigen::Matrix<double, -1, -1, 1, -1, -1> const, 0, Eigen::Stride<0, 0> >, 5>
>(Eigen::DenseBase<Eigen::GeneralProduct<Eigen::Map<Eigen::Matrix<double, -1, -1, 1, -1, -1> const, 0, Eigen::Stride<0, 0> >,
Eigen::Map<Eigen::Matrix<double, -1, -1, 1, -1, -1> const, 0, Eigen::Stride<0, 0> >, 5> > const&)
  4.55      0.22     0.01       20     0.50     0.50  Eigen::internal::gemm_pack_lhs<double, long, 4, 2, 1, false,
false>::operator()(double*, double const*, long, long, long, long, long)
  0.00      0.22     0.00       20     0.00     0.00  Eigen::internal::gemm_pack_rhs<double, long, 4, 1, false,
false>::operator()(double*, double const*, long, long, long, long, long)
  0.00      0.22     0.00       10     0.00    20.00  Eigen::internal::general_matrix_matrix_product<long, double, 1, false,
double, 1, false, 0>::run(long, long, long, double const*, long, double const*, long, double*, long, double,
Eigen::internal::level3_blocking<double, double>&, Eigen::internal::GemmParallelInfo<long>*)
  0.00      0.22     0.00        2     0.00     0.00  Eigen::internal::queryCacheSizes_intel_direct(int&, int&, int&)
  0.00      0.22     0.00        1     0.00   220.02  dgemm_eigen(double const*, double const*, double*, unsigned long, unsigned
long)
  0.00      0.22     0.00        1     0.00     0.00  profile_dgemm(unsigned long, unsigned long,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
```

call graph also produced

Measuring Code Performance: C++ Chrono

read the clock before and after the kernel call

```
#include <iostream>
#include <chrono>

int main()
{
    auto start_time = chrono::high_resolution_clock::now();
    // your kernel goes here
    auto end_time = chrono::high_resolution_clock::now();
    cout << chrono::duration_cast<chrono::seconds>(end_time - start_time).count() << ":";
    cout << chrono::duration_cast<chrono::microseconds>(end_time - start_time).count() << ":";
    return 0;
}
```

issues: granularity, overhead, jitter, cold hardware, etc.

best to use specialized microbenchmarking libraries

Why We Must Understand Computer Architecture

- Hardware is the primary factor that determines performance of your code (compiler is another … but that's another story altogether)

- Knowledge of the hardware can help us develop better performance models and better reason about code's performance (and even correctness)

- Knowledge of the hardware can also help us write better code by choosing more performant data structures and algorithms, etc.

- Modern computer architectures are pretty amazing feats of human ingenuity (and so are the compilers …)

- etc.

Modern Computers: Overview

now

Oak Ridge National Lab
202752 CPU cores
+ 2322432 GPU "cores"

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

my office
64 CPU cores

your lap
2-8 CPU cores
+ ~20 GPU cores

~2021

Argonne National Lab

Oak Ridge National Lab

exascale computer near you
~10^7 cores?

your department/campus
~10^6 CPU+GPU cores

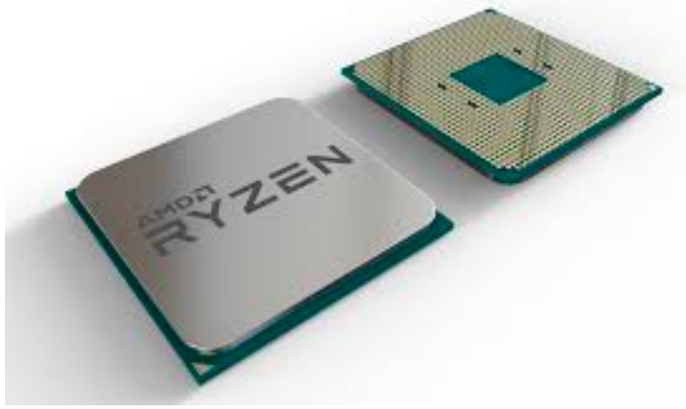
Modern Processors: x86 family

Intel Ice Lake (2019)

Intel Skylake (2015)/
Cannon Lake (2018)

up to ~32 cores
64-bit ("x86-64")

AMD Zen (2017)



Intel Haswell (2013)/
Broadwell (2015)

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)

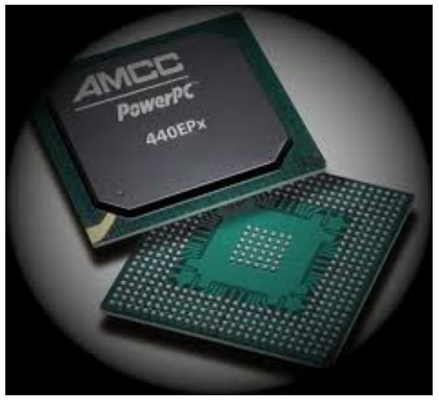
....

Intel 8086 (1978)

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

Modern Processors: non-x86 families

PowerPC (PPC)

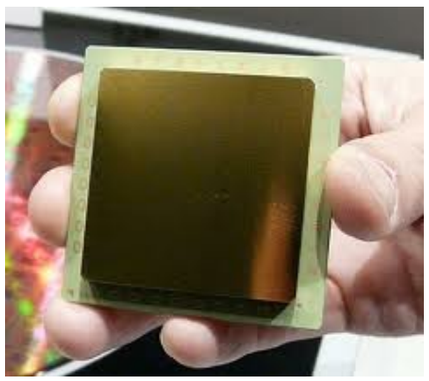


IBM Blue Gene and POWER servers

ARM

THE mobile/embedded

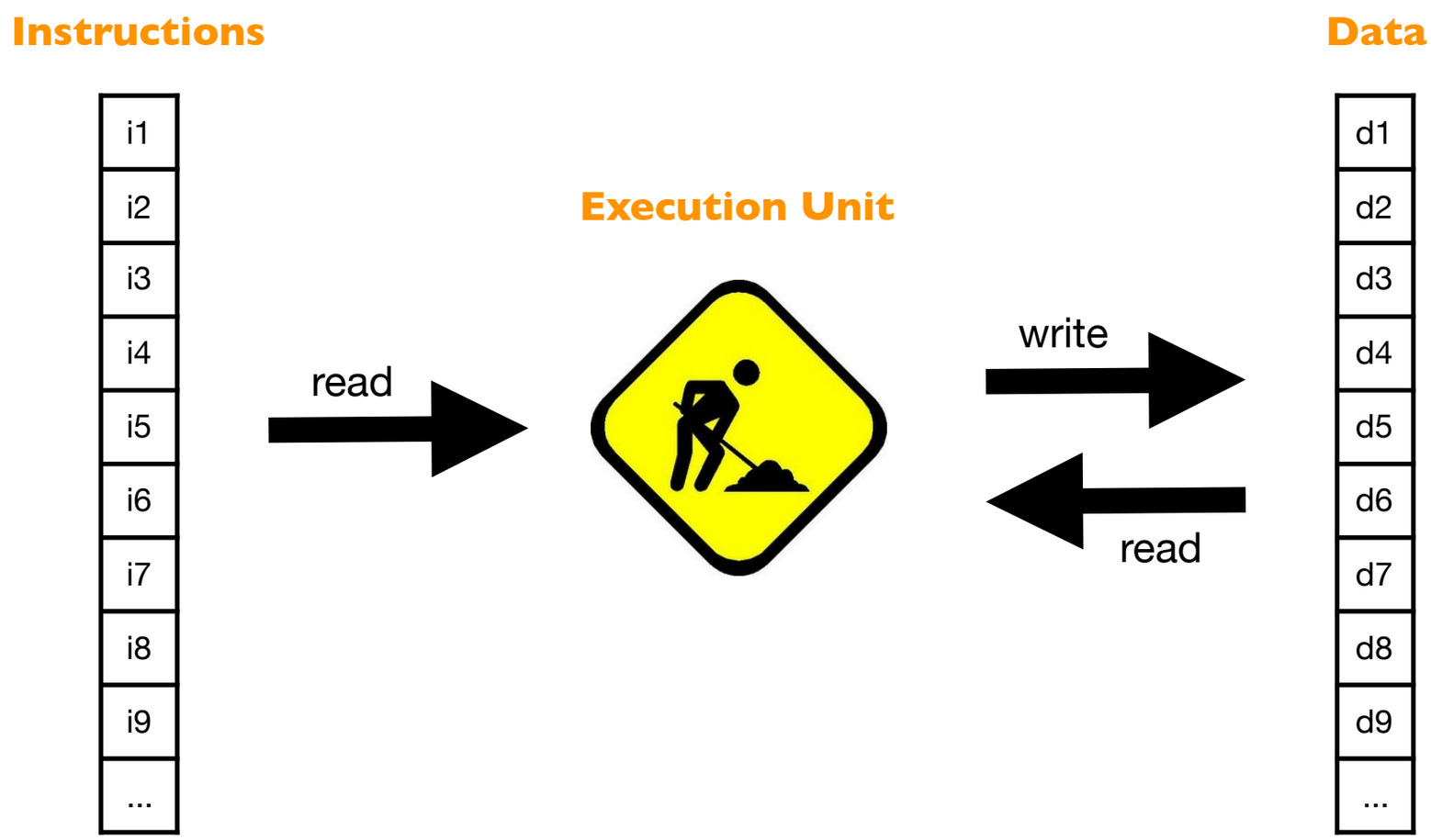
SPARC



Fujitsu K computer and Oracle servers

GPUs/accelerators

Abstract 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

How much time does processing one *Op* take?

Op = unit of work, i.e. instruction, kernel, etc.

naive $t_{\text{total}} = t_{\text{read-write}} + t_{\text{execute}}$

(perfect) streaming $t_{\text{total}} = \max(t_{\text{read-write}}, t_{\text{execute}})$

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

α: latency (0, if streaming)

β: data bandwidth

d: data size

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

"Op bandwidth"

Q: what are the units of α, β, γ, and d?

Performance Model

How much time does processing one *Op* take?

naive $t_{\text{total}} = t_{\text{read-write}} + t_{\text{execute}}$

(perfect) streaming $t_{\text{total}} = \max(t_{\text{read-write}}, t_{\text{execute}})$

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

α: latency (0, if streaming)

d: data size

β: data bandwidth

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

"Op bandwidth"

α, β, γ are **model parameters** determined by:

- type of work and data
- hardware
- compiler
- environment

Their idealized values can be obtained from hardware specs, but in practice you want to measure them

Performance Model

$$t_{\text{total}} = \max(t_{\text{read-write}}, t_{\text{execute}})$$

Overall performance can determined by memory or computation

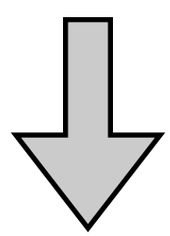
memory-bound	$t_{\text{read-write}} > t_{\text{execute}}$
compute-bound	$t_{\text{read-write}} < t_{\text{execute}}$

Most scientific applications are memory-bound

Performance Model

time / Op

$$t_{\text{total}} = \max(t_{\text{read-write}}, t_{\text{execute}})$$



Ops / time

$$\gamma_{\text{total}} \equiv \frac{1}{t_{\text{total}}} = \min\left(\frac{1}{t_{\text{read-write}}}, \frac{1}{t_{\text{execute}}}\right)$$

$$= \min\left(\frac{\beta}{d}, \gamma\right) \equiv \min(I \times \beta, \gamma)$$

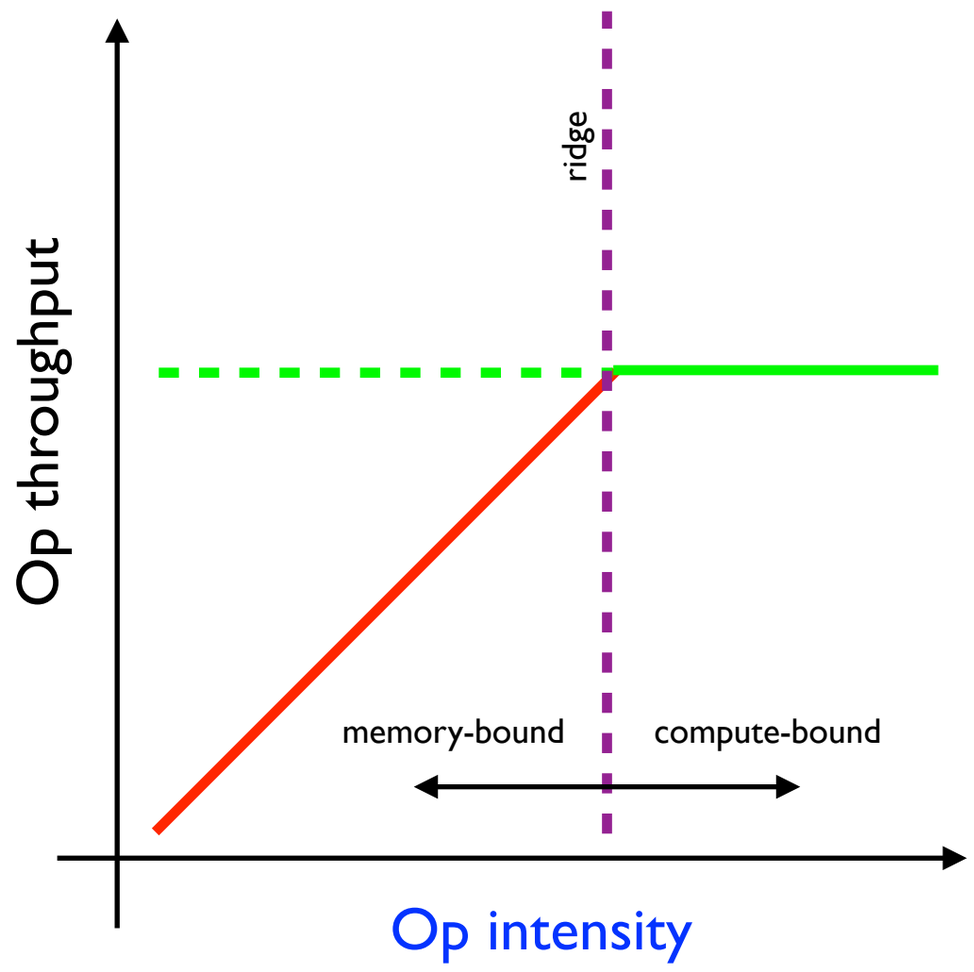
Ops / byte

$$I \equiv \frac{1}{d}$$

= operational (arithmetic) intensity

Performance Model: Roofline

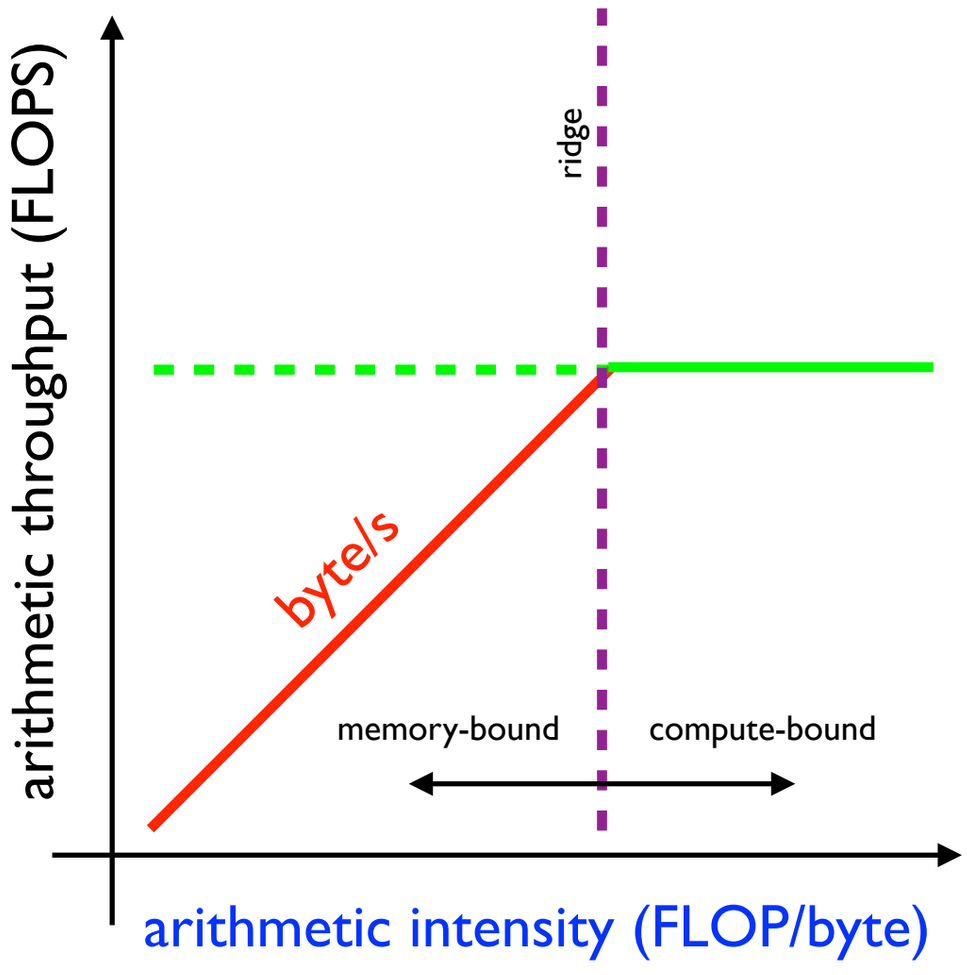
$$\gamma_{\text{total}} = \min(I \times \beta, \gamma)$$



recall: this is an **idealization**, assumes perfect streaming

Roofline for Floating-Point Instruction Streams

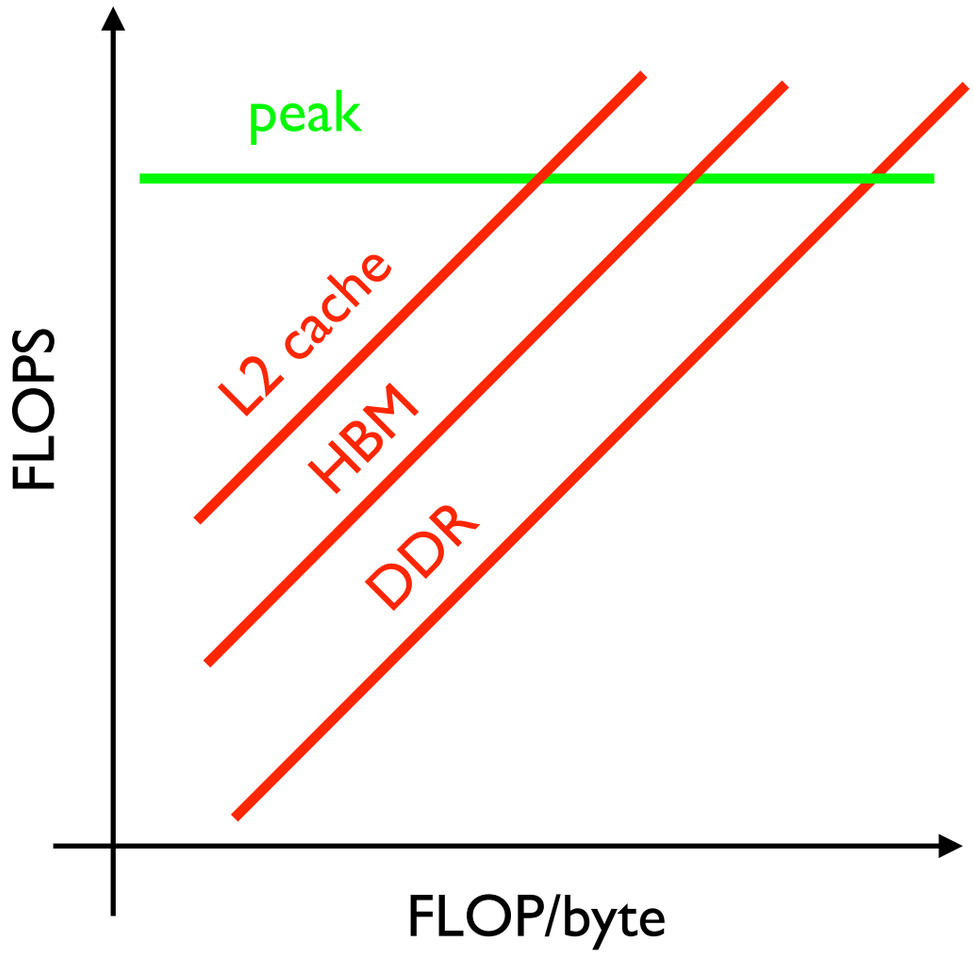
$$\gamma_{\text{total}} = \min(I \times \beta, \gamma)$$



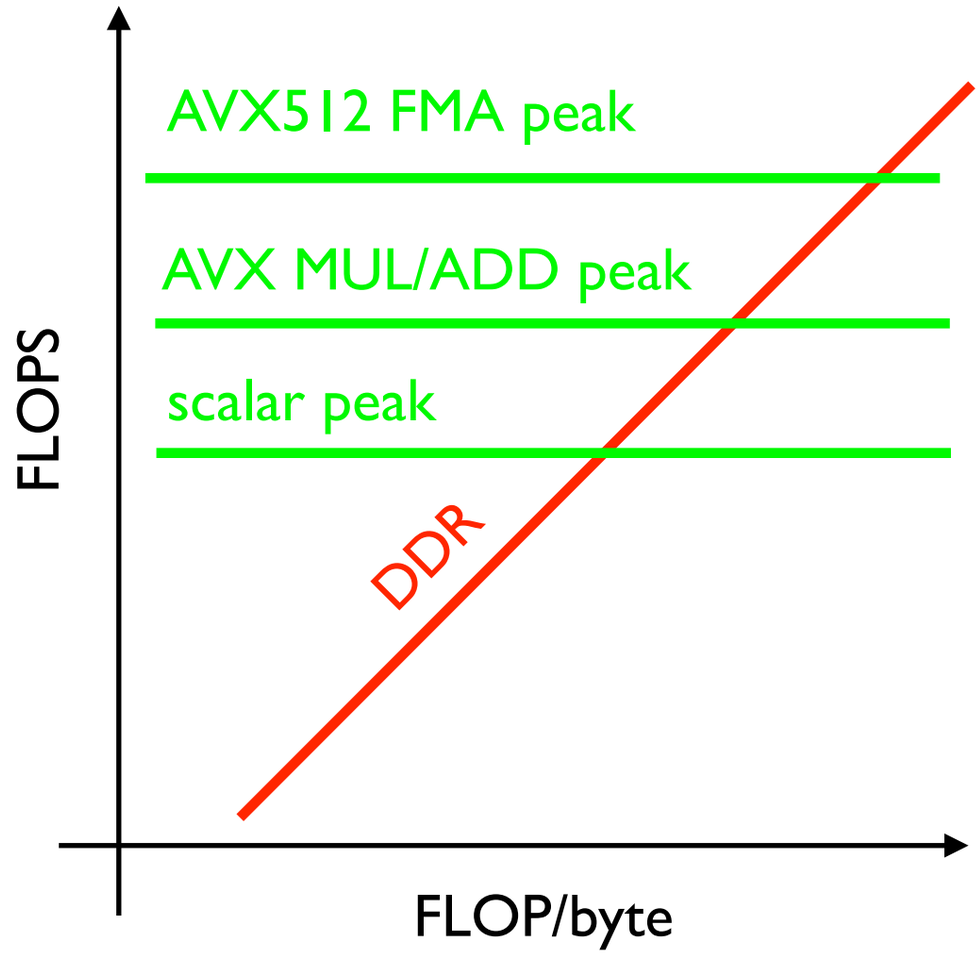
FLOP = floating point operation
FLOPs = plural of FLOP
FLOPS = FLOP/second
GFLOPS = 10^9 FLOPS
etc.

Roofline for Floating-Point Instruction Streams

further refinements for realism



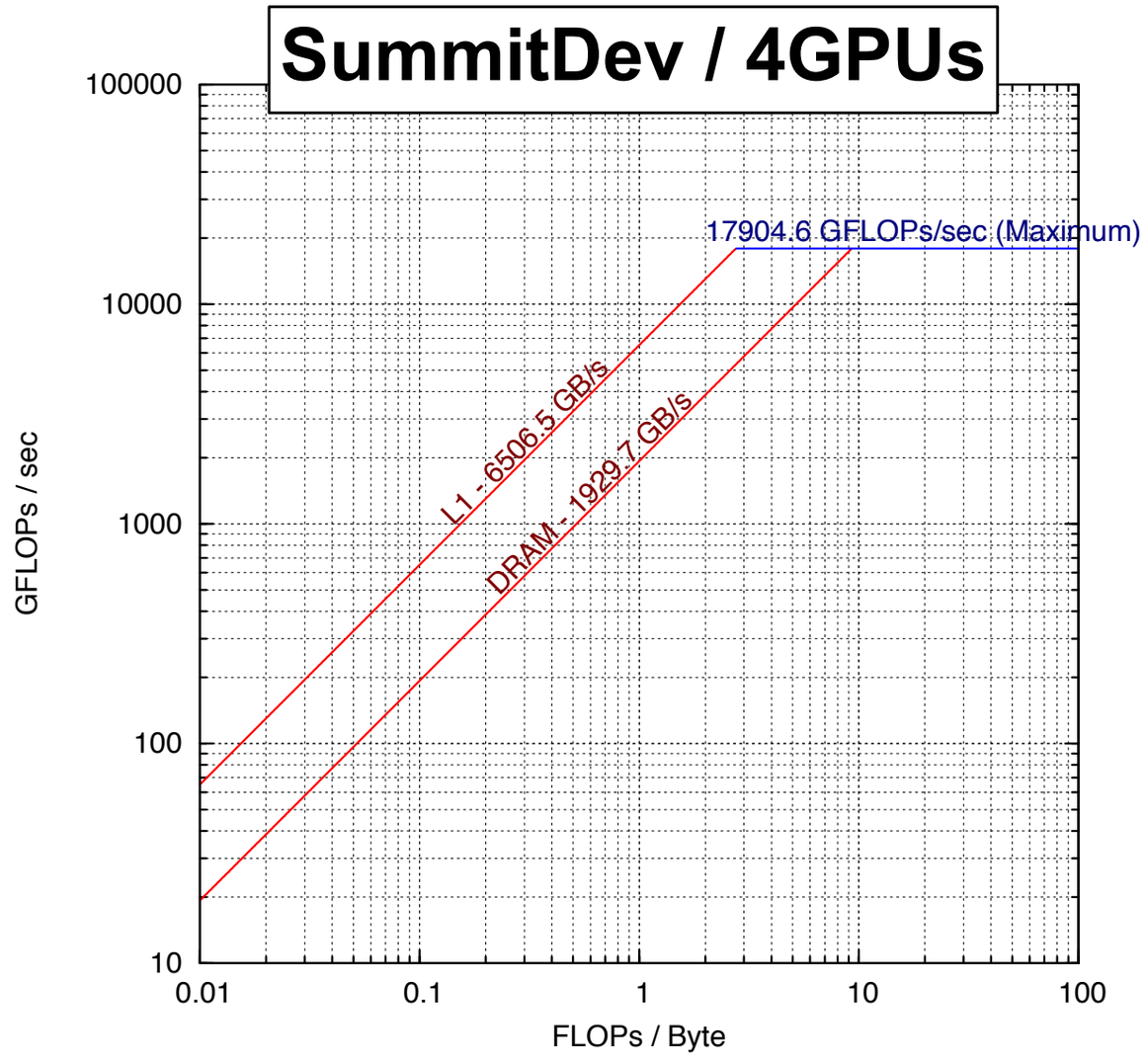
memory hierarchy



execution capability hierarchy

Roofline for Floating-Point Instruction Streams

roofline model for a real system



http://crd.lbl.gov/assets/Uploads/roofline-intro.pdf

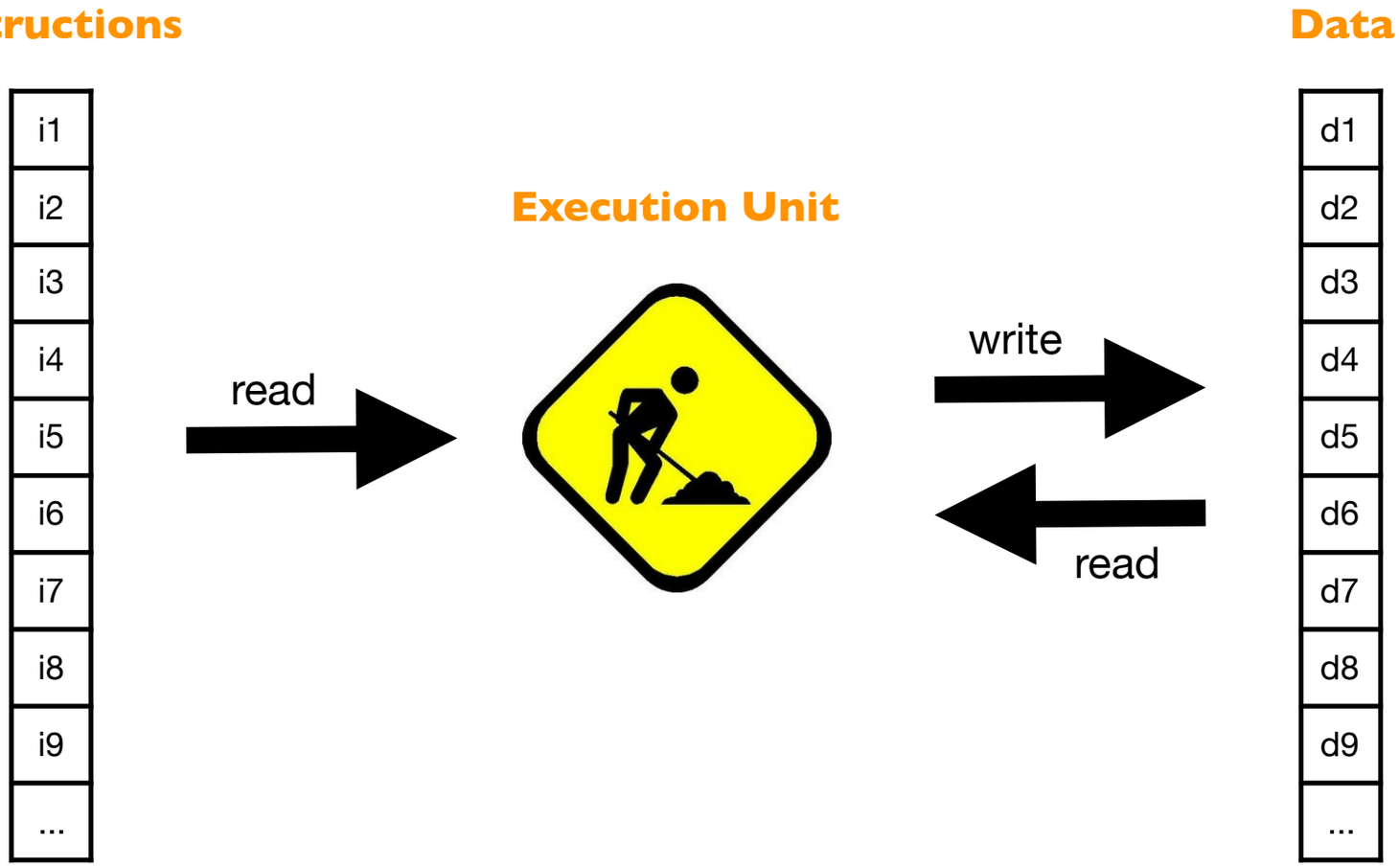
Burning Questions

- What determines the **memory bandwidth**?
- What determines the **operation bandwidth**?
- How can **latency** (of data access and instruction execution) be hidden?
- What is **vectorization**?
- How are **GPUs/accelerators** different from CPUs?
- How does the **compiler** play into this?

hardware + compiler work together to create the illusion that your code executes on an abstract model of real hardware (e.g. abstract serial processor)
(e.g. "C Is Not a Low-level Language" https://queue.acm.org/detail.cfm?id=3212479)

let's see what actually happens behind the scenes

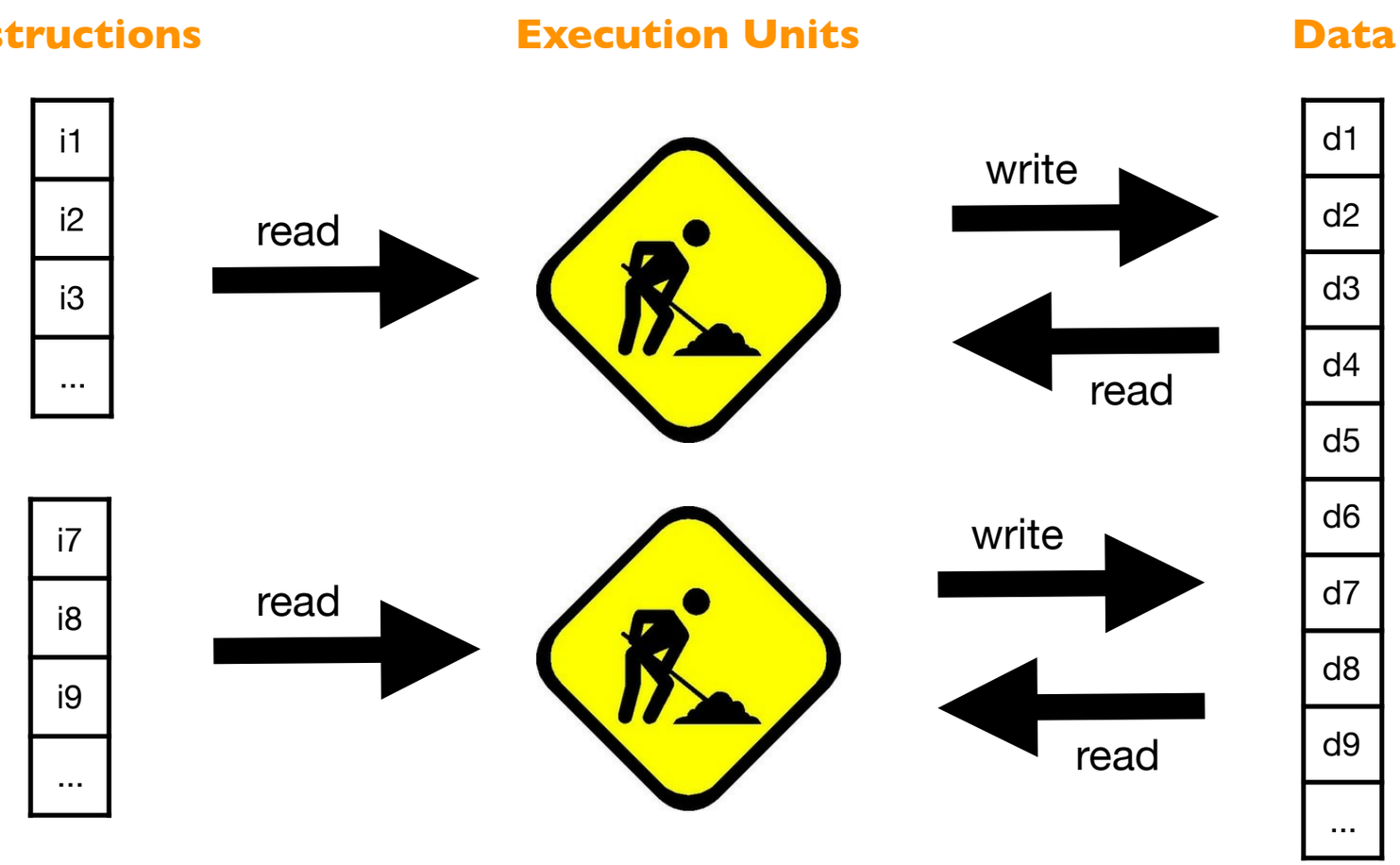
Abstract Data-Parallel Processor



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)
- All processors of today and tomorrow

Abstract Instruction-Parallel Processor

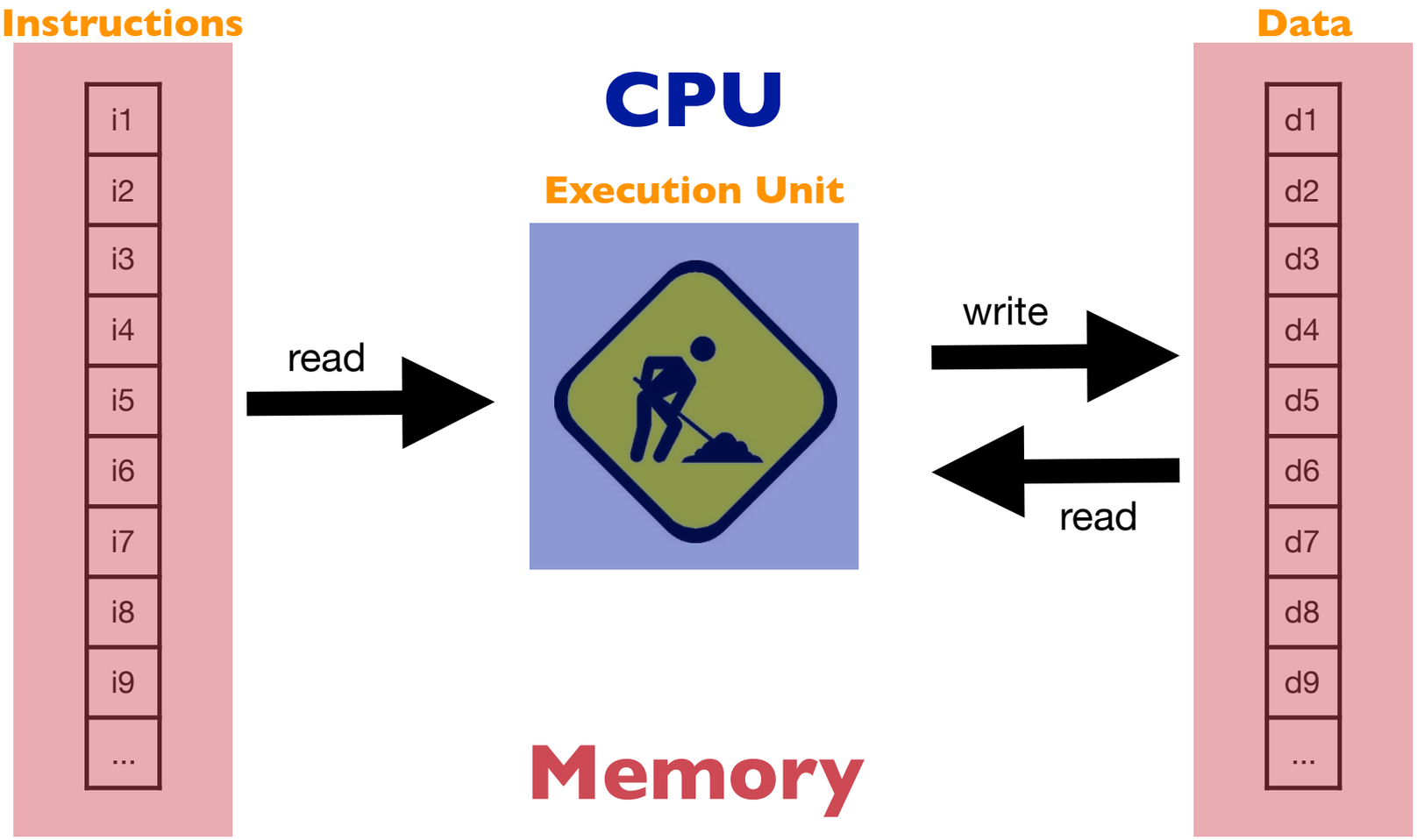


- several (2-32) instructions retired at a time
- each instruction acts on one or several (2-16) sets of data
- Multiple Instructions Multiple Data (MIMD)
- all processors of today and tomorrow

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)

Abstract Serial Processor

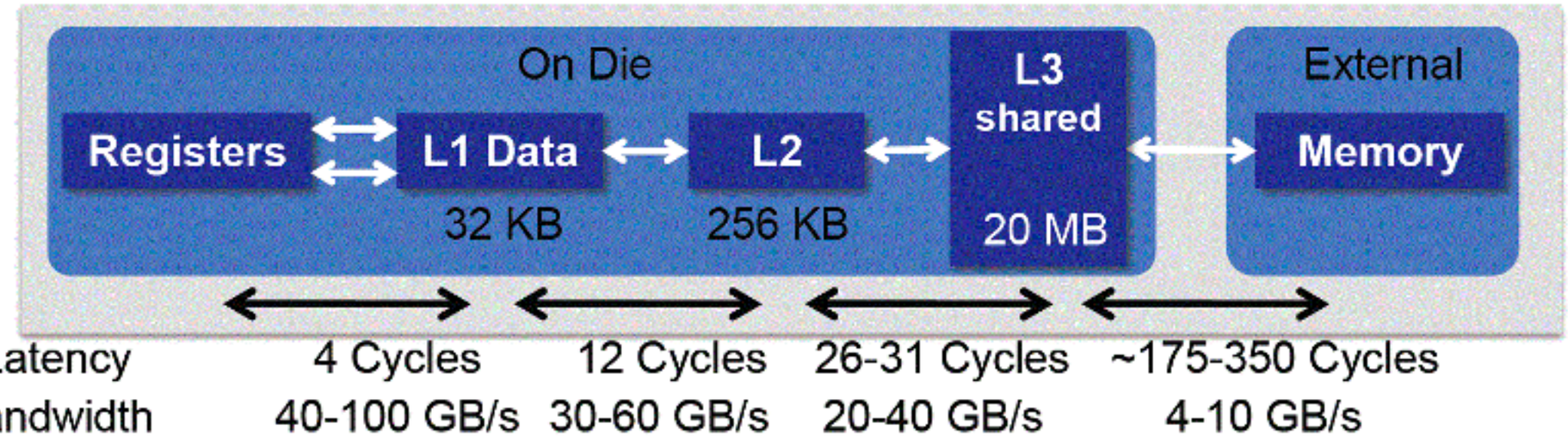


Memory Hierarchy for a “Classic” Computer

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 - 32 GB	100	off/on die
Disk	1 TB	10^7	off die
Tape	20 TB	10^{10}	off die

Memory Hierarchy for a “Classic” Computer

Example: Intel Xeon “Sandy Bridge” memory hierarchy



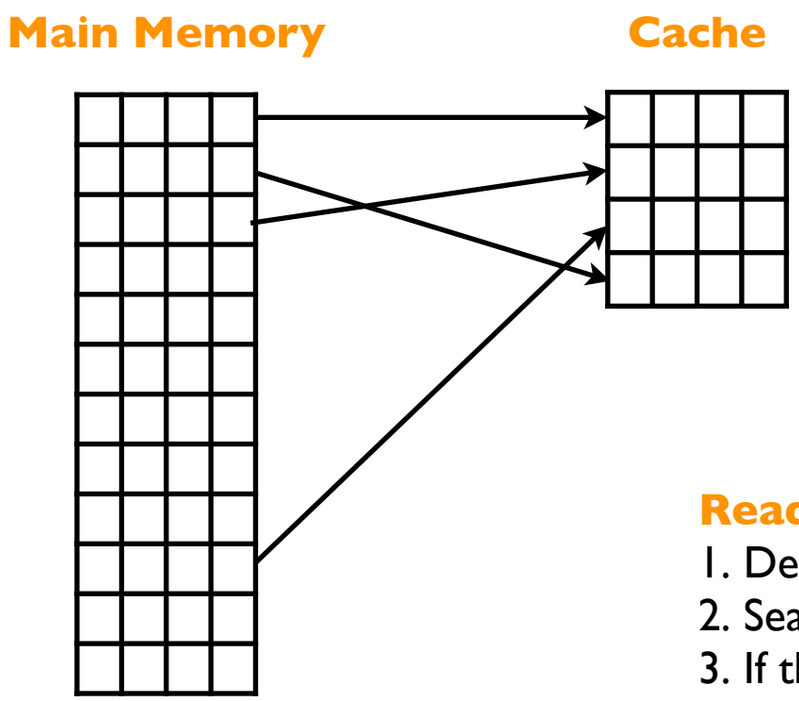
Numerical Algorithms: Linear Algebra

Performance of DAXPY vs. vector size (GFLOP/s)

	n	Intel Xeon E5645 “Nehalem” 2.4 GHz (1.3 GHz DRAM) SSE2 peak=9.6	Intel Core I7-3820QM “Ivy Bridge” 2.7 GHz (1.6 GHz DRAM) AVX peak=21.6
data in L1 cache	1024	4.7	11.7
	2048	4.6	11.6
data in L2 cache	4096	2.7	5.4
	8192	2.7	5.3
data in L3 cache	100000	1.8	3.8
data in main memory	10000000	1	1.6

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

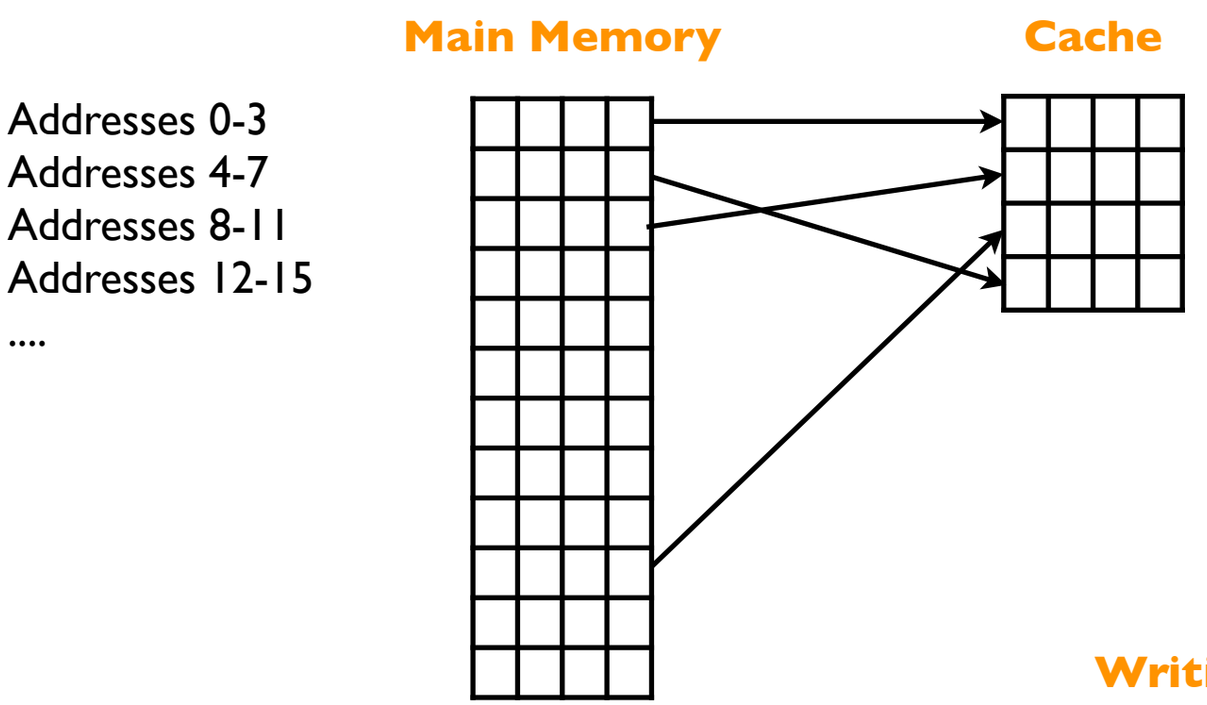
cache lines: 8 - 512 bytes
(x86: 64 bytes)

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
....
can avoid penalties by **prefetching**

Cache Structure and Cache Misses



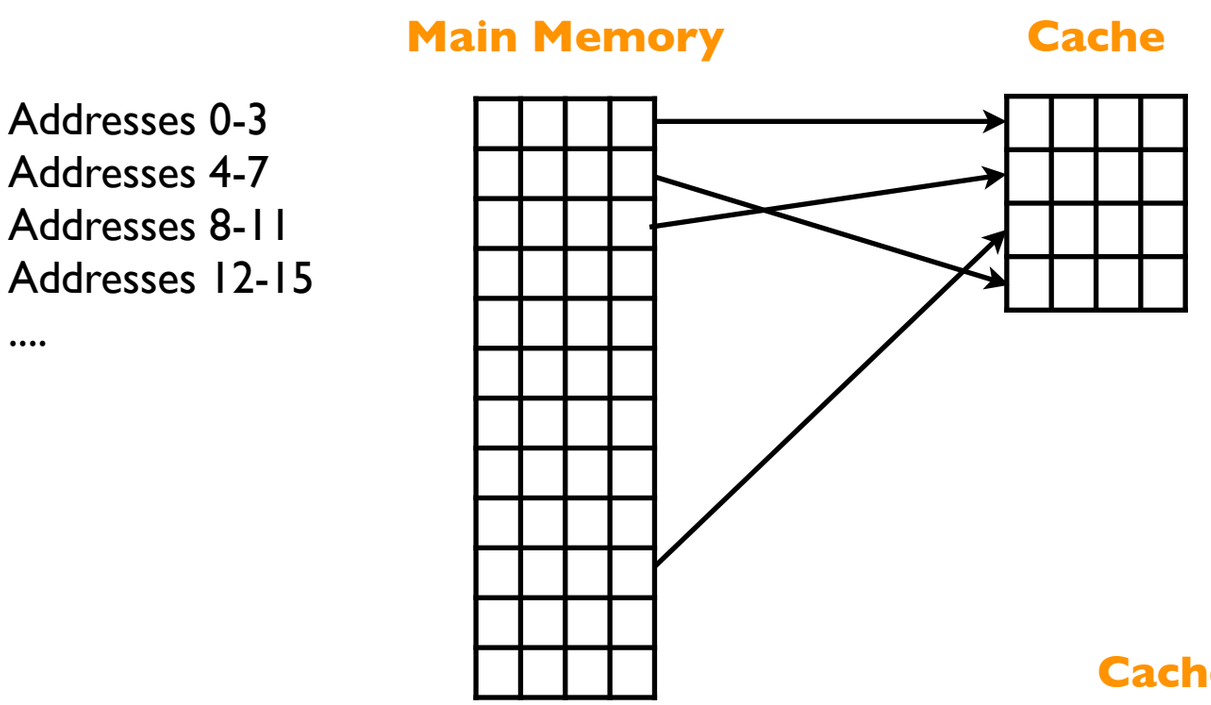
Line 0	Tag 0
Line 1	Tag 2
Line 2	Tag 9
Line 3	Tag 1

cache lines: 8 - 512 bytes
(x86: 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



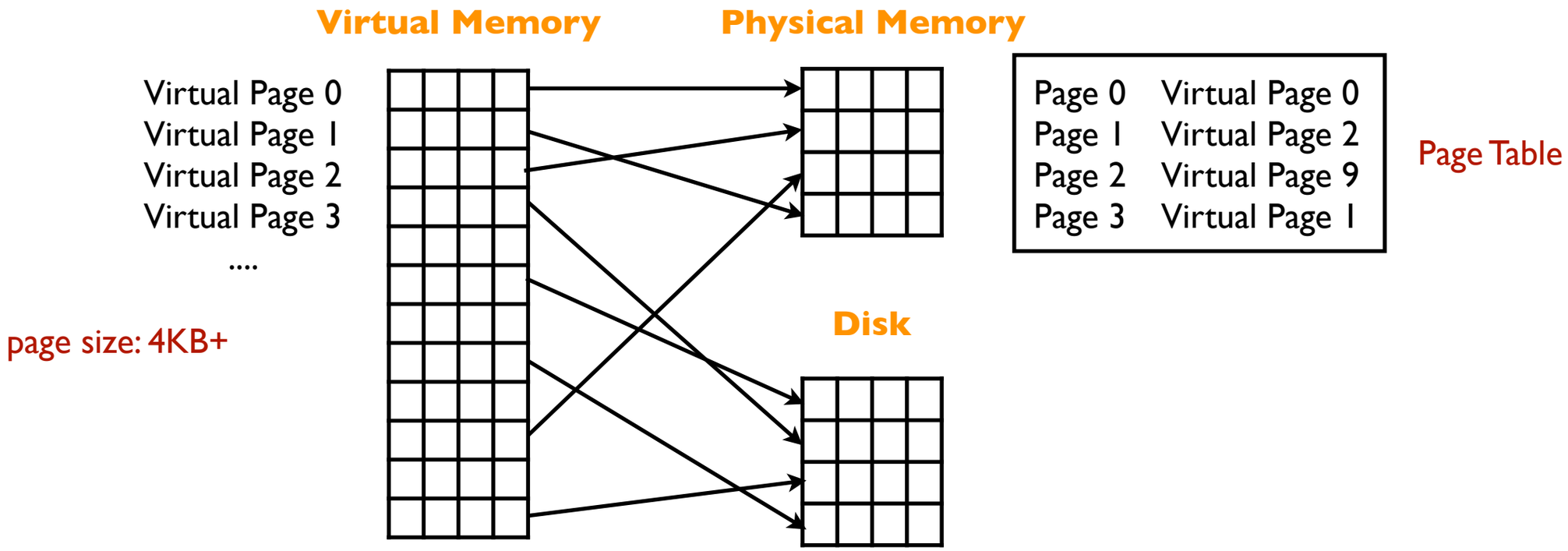
Line 0	Tag 0
Line 1	Tag 2
Line 2	Tag 9
Line 3	Tag 1

cache lines: 8 - 512 bytes
(x86: 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

Lesson: for high performance all data should fit into physical memory

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 (temporal locality)
- Do not rely on virtual memory mechanism! Fit all data in DRAM
- Adjacent instructions should access adjacent memory locations (spatial locality), otherwise bandwidth is wasted
- Unfortunate memory access patterns can result in reduced effective cache size

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

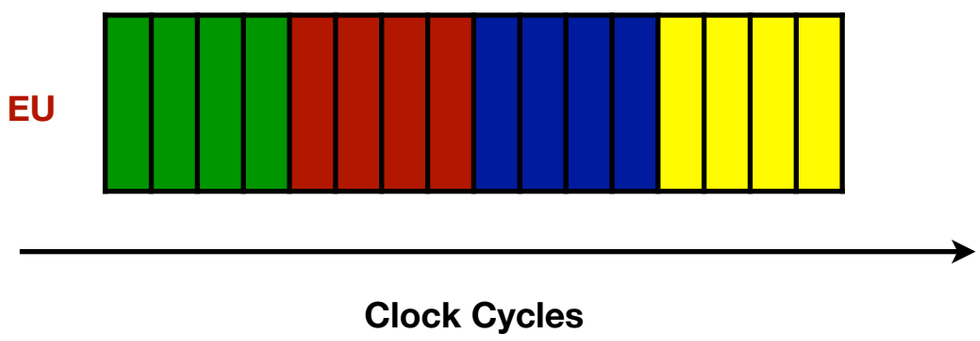
Instruction	Latency
add	3
mul	5
div	21-45
sqrt	21-43

latencies vary with microarchitecture and are not published anywhere
see Agner Fog's resources https://www.agner.org/optimize/ for more info

Pipelining

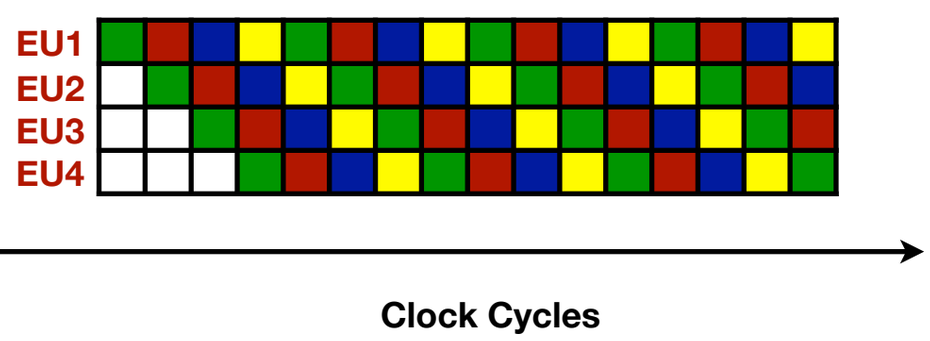
Example: executing four 4-cycle latency instructions in a loop

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

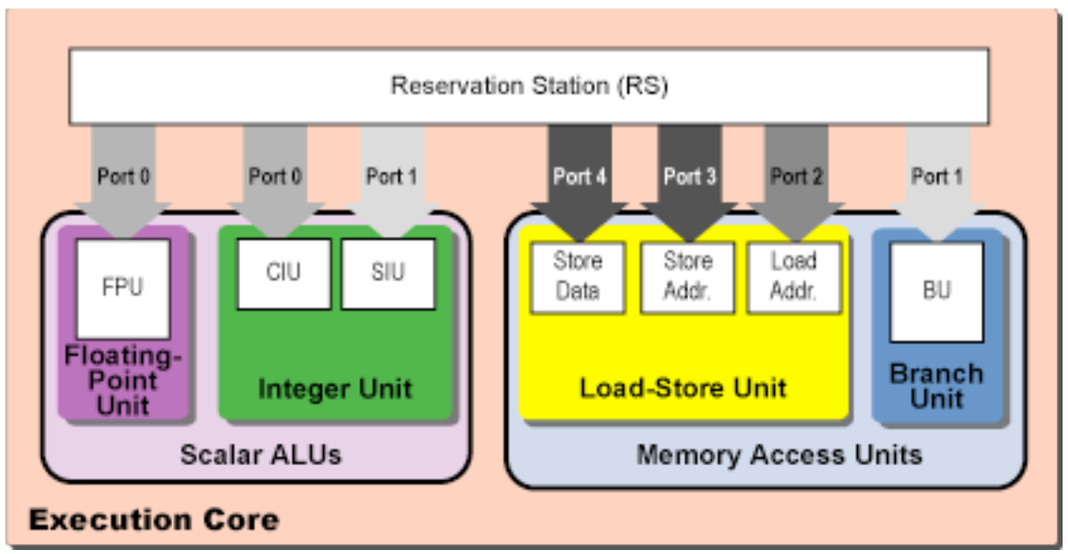
Multiple issue (superscalar) processors

exploit fine-grained parallelism by executing multiple operations simultaneously

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

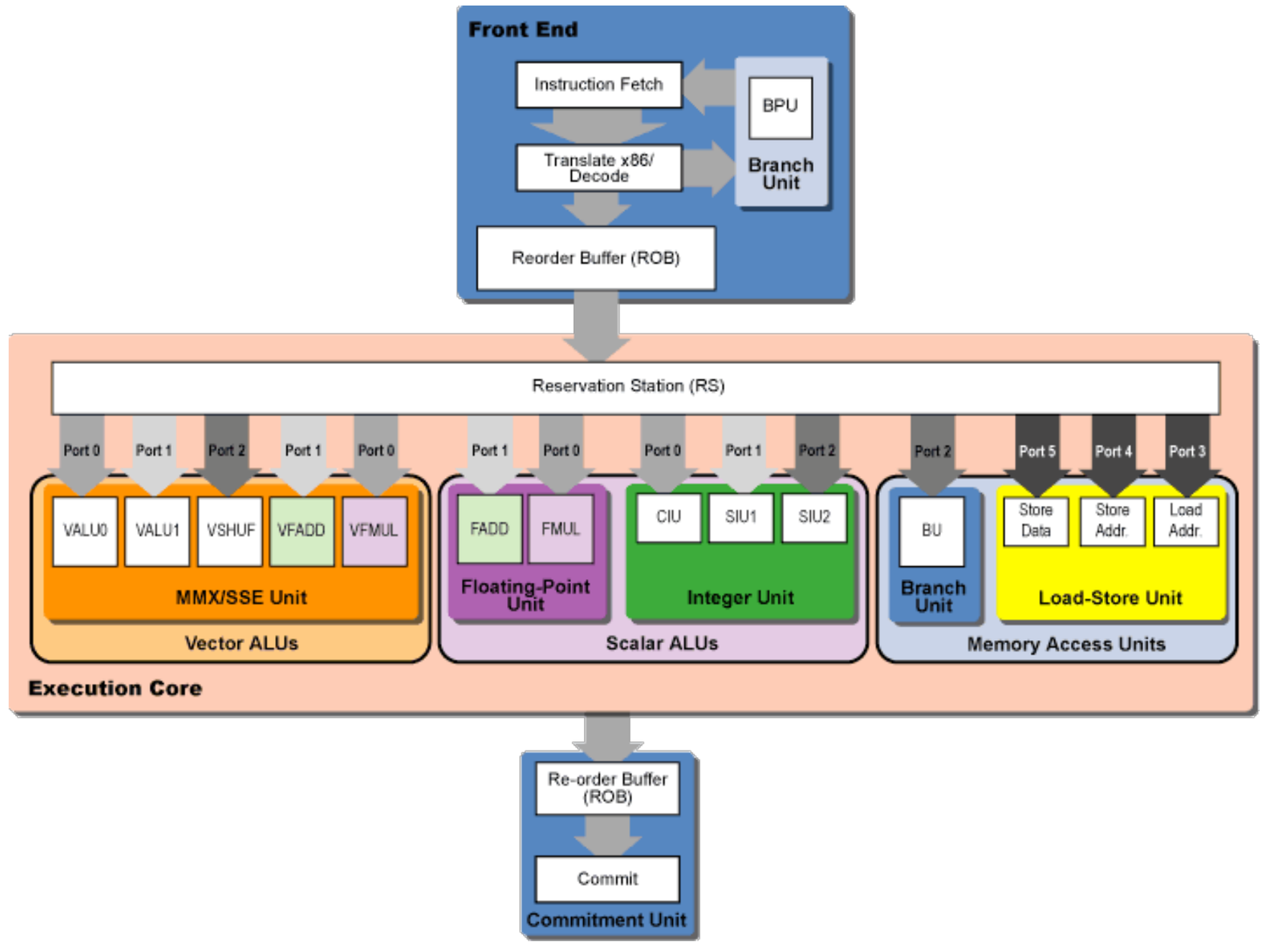
no data dependencies, can be executed simultaneously given enough ALUs

Intel Pentium Pro: superscalar out-of-order core



Multiple issue (superscalar) processors

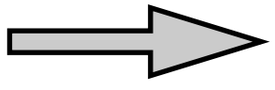
Intel Core: superscalar out-of-order core



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
- **AVX2** introduced fused multiply-add (FMA) and gather

introduced in Intel Sandy Bridge and AMD Bulldozer

AVX-512

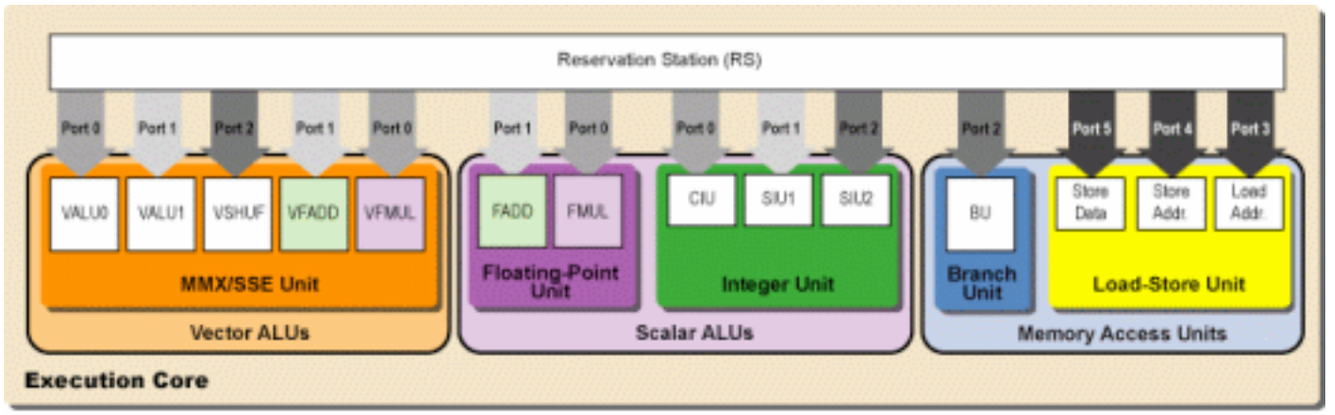
- 512-bit registers = 8 double-precision numbers
- Includes many extensions, not all of them supported by all hardware

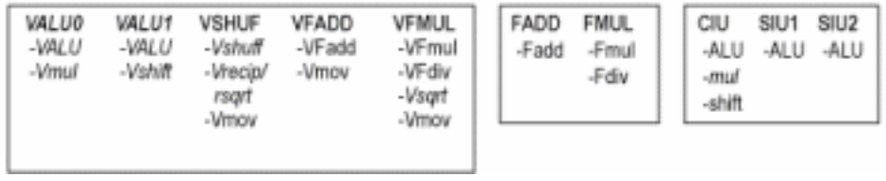
introduced in Intel KNL, now available in Skylake Xeon and Cannon Lake

non-x86 architectures also have SIMD instructions (VSX on POWER, HPC-ACE on K computer, NEON on ARM)

Example: Computing FLOPS for Linear Algebra

Intel Core2 execution core





Theoretical peaks (64 bit)

Scalar: 1 FP add + 1 FP mul every cycle = 2 FLOPs / cycle
SSE: 2 FP add + 2 FP mul every cycle = 4 FLOPs / cycle
AVX: 4 FP add + 4 FP mul every cycle = 8 FLOPs / cycle
AVX2: 4 FP multiply-add + 4 FP multiply-add = 16 FLOPs / cycle
AVX512: 8 FP multiply-add + 8 FP multiply-add = 32 FLOPs / cycle

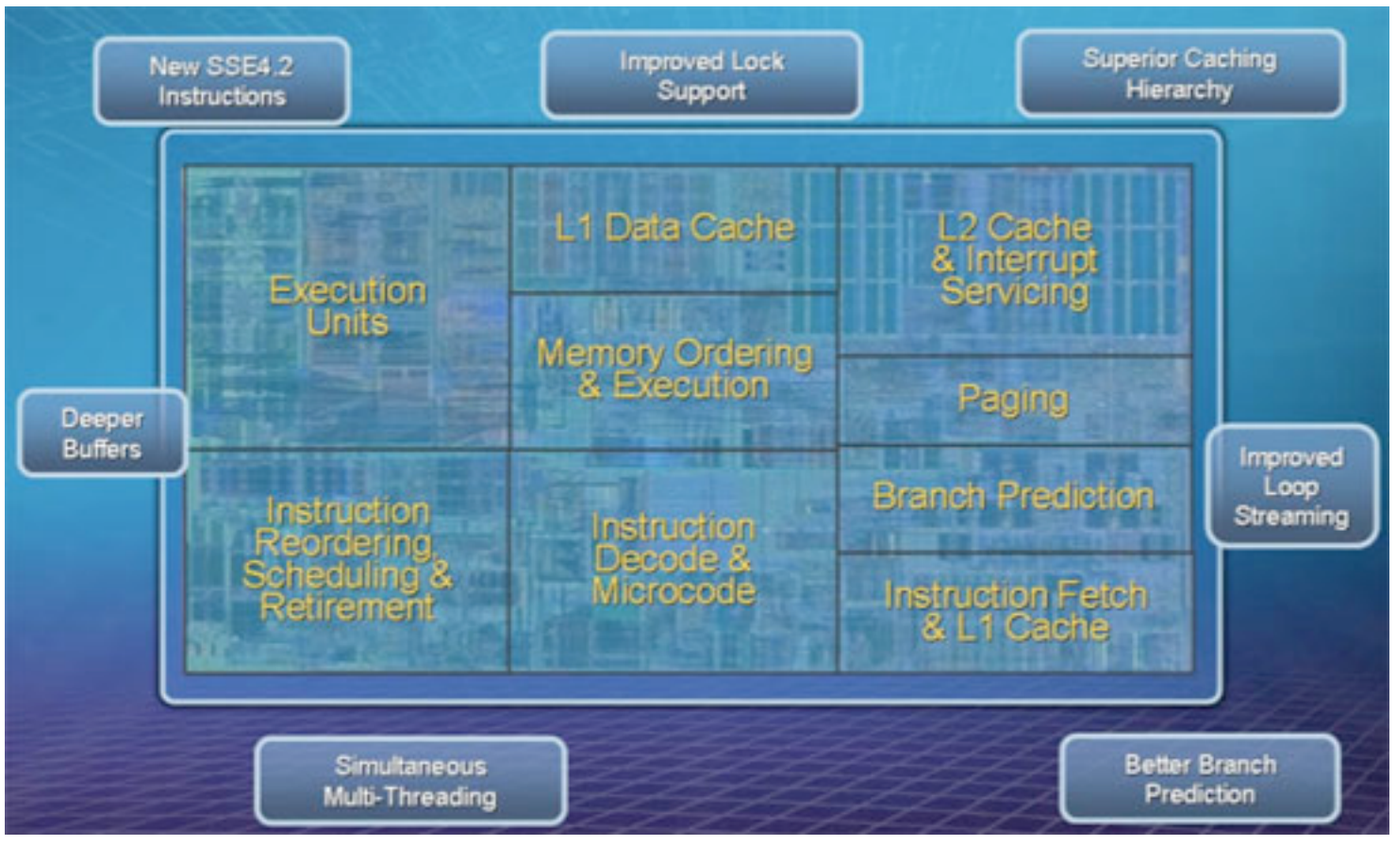
DAXPY: Y[i] += a * X[i]

- 2 vops for every 2 vloads and 1 vstore
- max 2 vloads/vstores / cycle!
  - Memory-bound even out of L1 cache!
- Roofline analysis for my laptop
  - 2133 DDR3 = 17GB/s = 2.1 billion FP64 / s
  - AI = 2 / 3 = 0.66
  - 2.1 billion FP64 / s x 0.66 = 1.4 GFLOPS
  - Much less than 46.4 GLOPS nominal peak!

DDOT? DGEMM?

Where are most transistors used? Not on ALU!

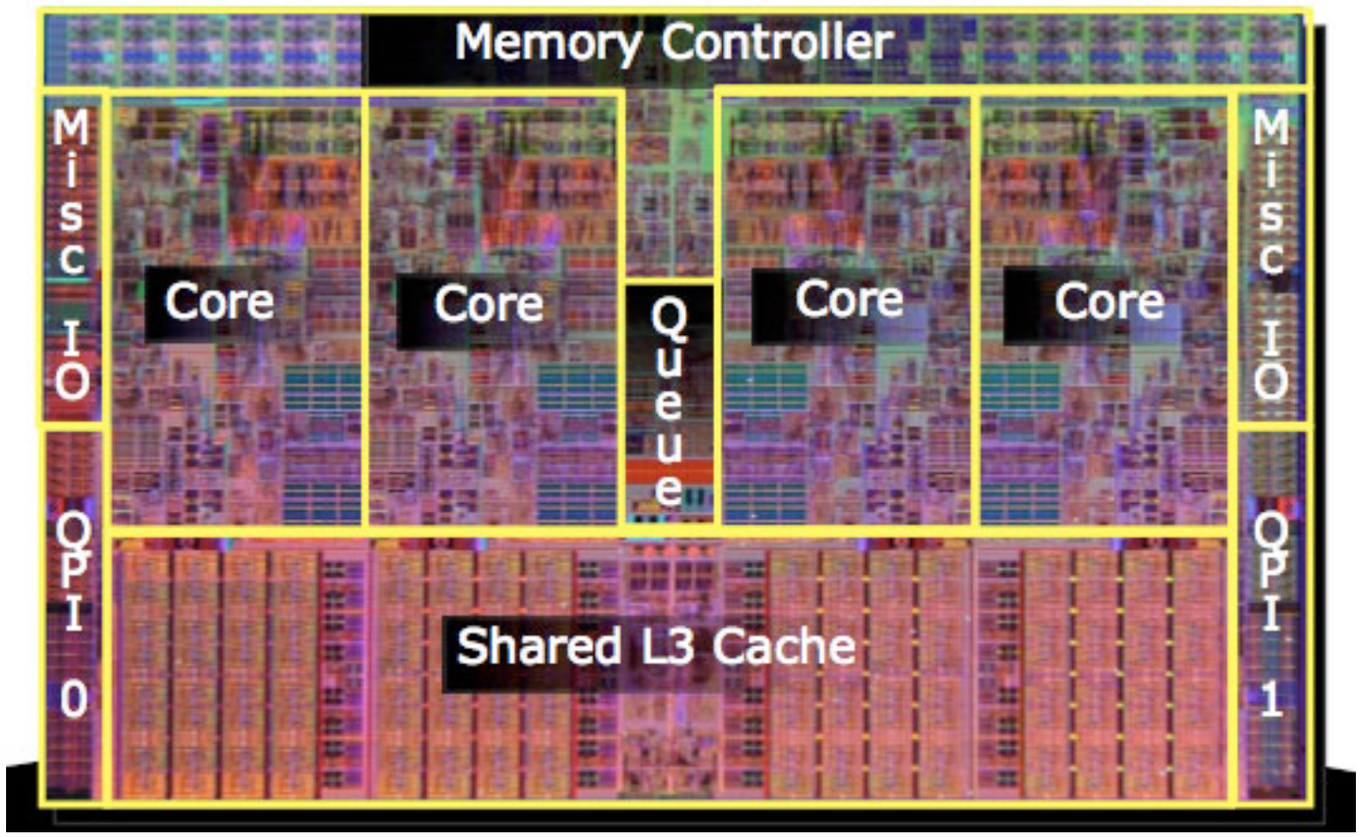
Intel Nehalem *execution core* dye



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

Where are most transistors used? Not on ALU!

Intel Nehalem *chip* dye



Power is another performance metric

High Cost of Data Movement

Operation	Energy consumed	Time needed
64-bit multiply-add	200 pJ	1 nsec
Read 64 bits from cache	800 pJ	3 nsec
Move 64 bits across chip	2000 pJ	5 nsec
Execute an instruction	7500 pJ	1 nsec
Read 64 bits from DRAM	*12000 pJ*	*70 nsec*

Notice that 12000 pJ @ 3 GHz = 36 watts!

Algorithms & Software: minimize data movement;
perform more work per unit data movement.

http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/SPRING-2015/lect14-1.pdf

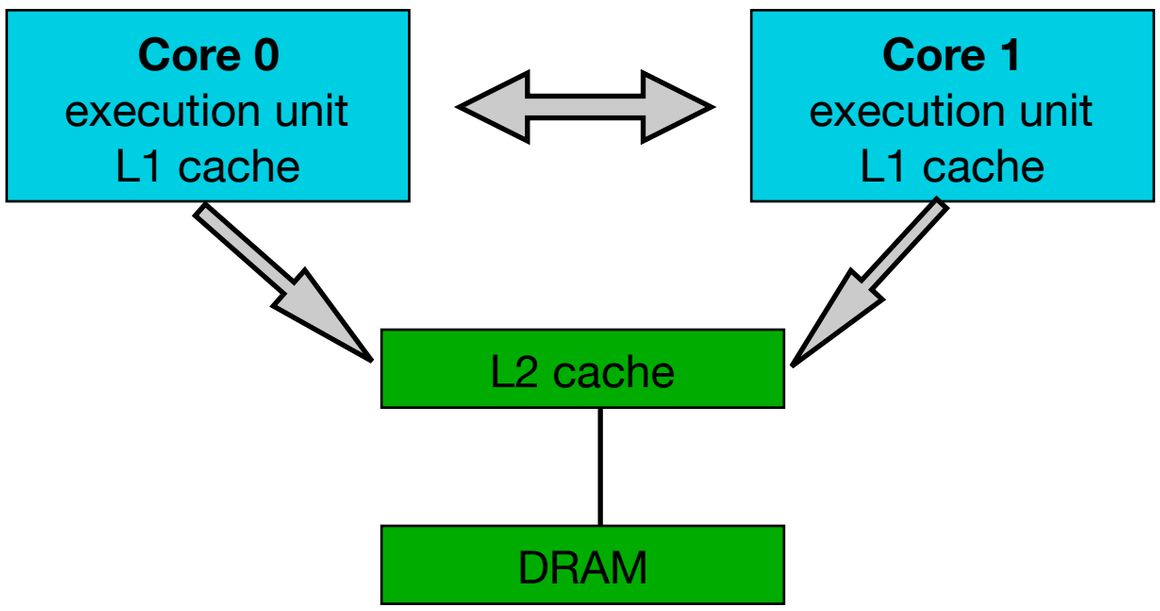
Summary

- 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

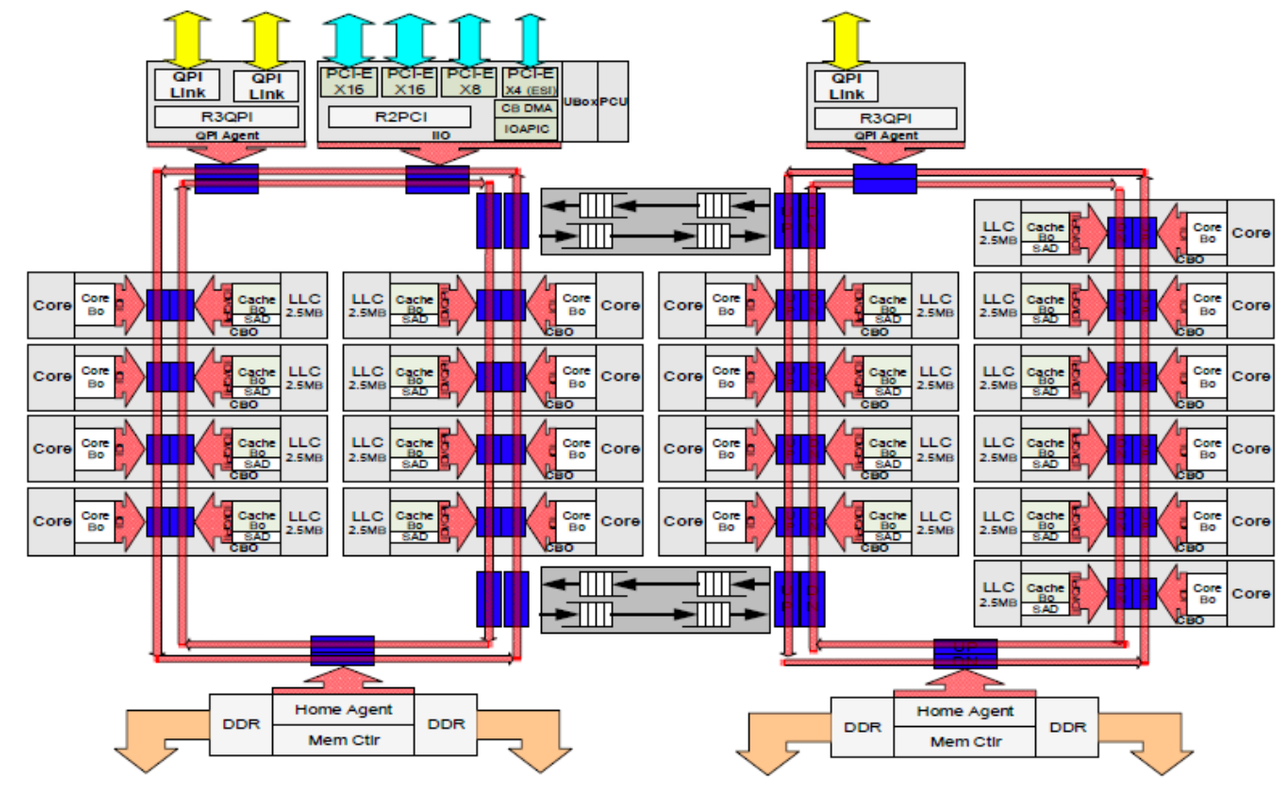


Cache coherence is an additional consideration

- 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
- coherence works with cache lines! Avoid *false sharing* which occurs if multiple cores write to distinct memory locations on same line

Multicore processors

Intel Xeon Haswell EX

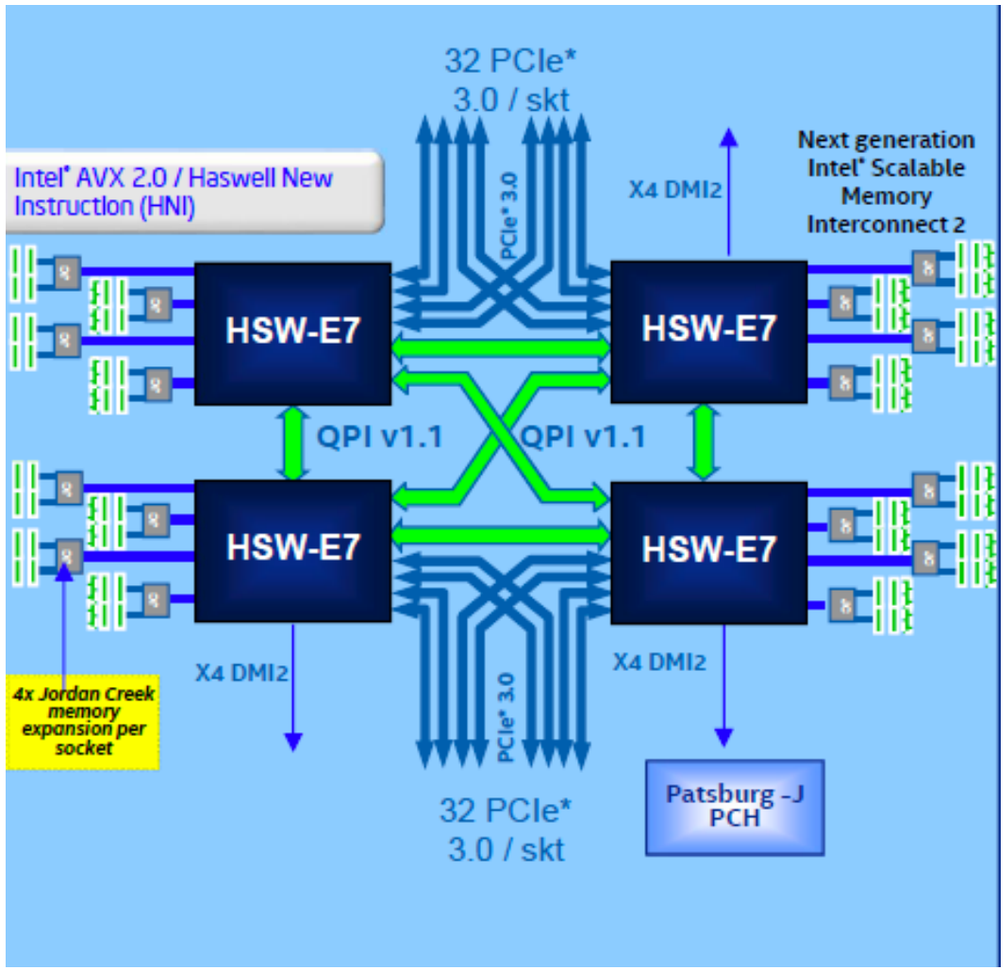


Chop	Columns
Columns	4
Home Agents	2
QPI links	3
Cores	4-18
Power (W)	115-165
Transistors (B)	5.7
Die Area (mm2)	662

- Each core has its own L1 and L2
- All cores share L3 (Last Level Cache, LLC)

“Shared-Memory” multiprocessors

Intel Haswell Xeon system

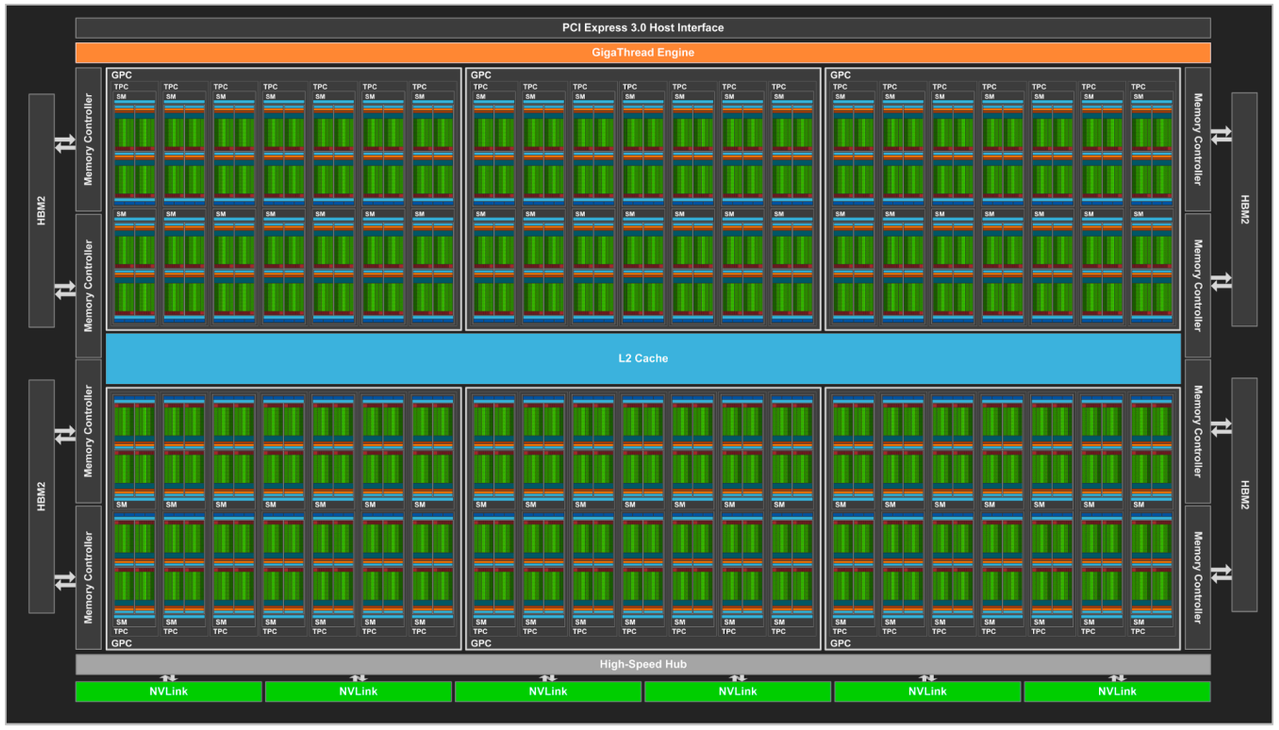


Nonuniform Memory Access (NUMA)

- Each socket is directly connected to a memory bank
- Sockets are connected by a high-speed interconnect (QPI)
- Accessing a remote memory bank involves QPI traffic
- To reach peak performance must carefully optimize for data locality

Massively many cores on a chip: NVIDIA GPGPUs

- 10s of Streaming Multiprocessors (think: CPU cores) running @ O(1 GHz)
- Each SM has multiple 16-wide SIMD units (but scheduled as 32-wide SIMD)
- little cache relative to CPU analogs
- huge memory bandwidth: O(1000 GB/s)
- bandwidth to the main (host) computer is relatively low, O(10 GB/s) -- must compute everything on the GPU card
- memory latency hidden by having many (1000s) instructions in-flight at a time -- someone will always have work to do; requires huge register file
- specialized programming models: CUDA (specific to NVIDIA), AMD ROCm (AMD), OpenCL (cross-platform C), emerging C++ standard proposals (+ Intel SYCL)



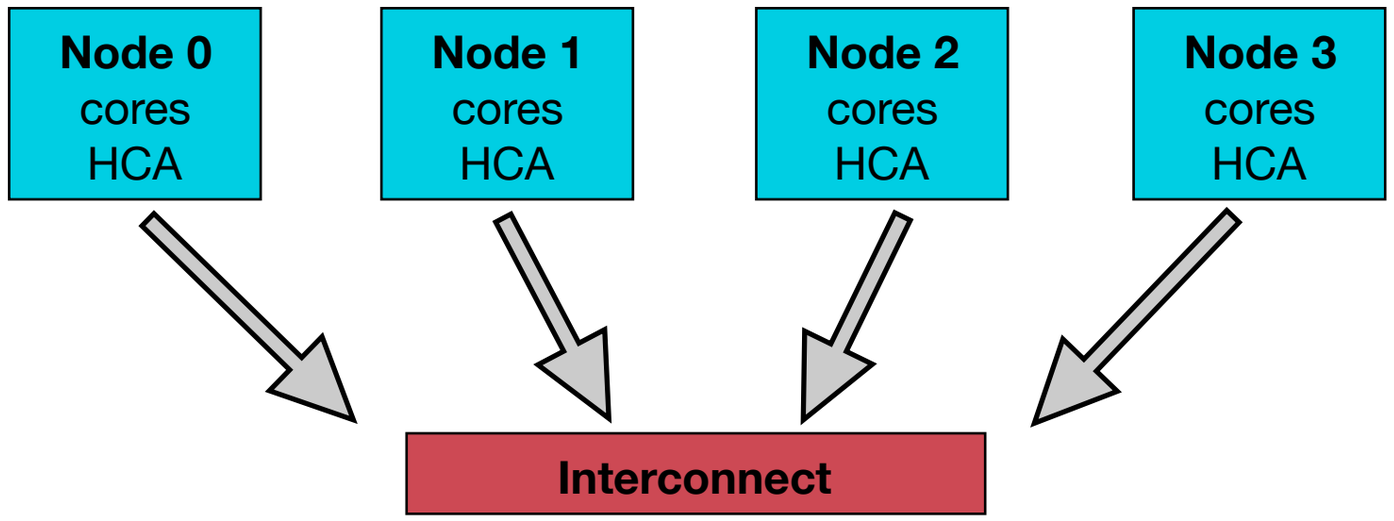
NVIDIA V100

84 SMs x 4 FP SIMD x ~1.4 GHz = 7.5 TFLOPs

AMD and Intel GPU architectures largely similar, but use different terminology

Distributed-memory multiprocessors

Generic distributed-memory multiprocessor

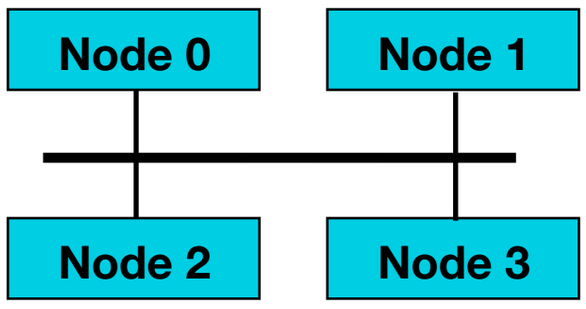


Distributed- vs. shared-memory

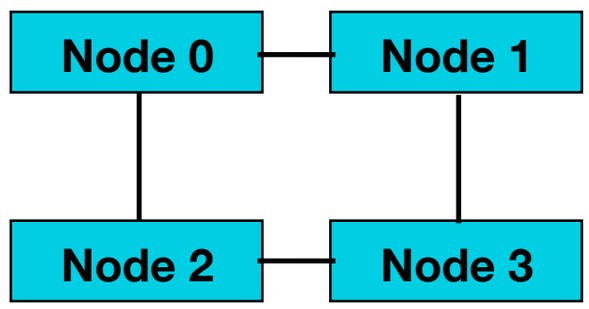
- easier to increase the size of a distributed-memory machine, no complicated memory coherence hardware necessary
- programming shared-memory machines can be deceptively similar to serial
- both bring up new issues in software development

Interconnect networks

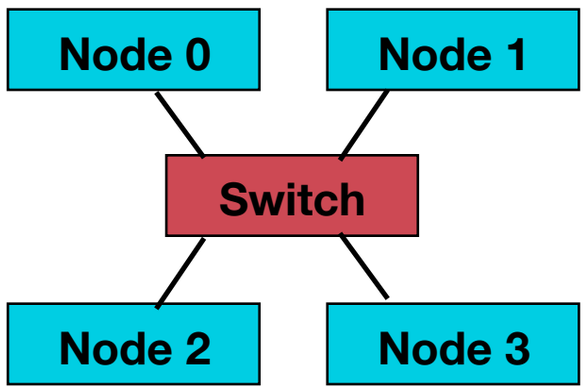
Bus



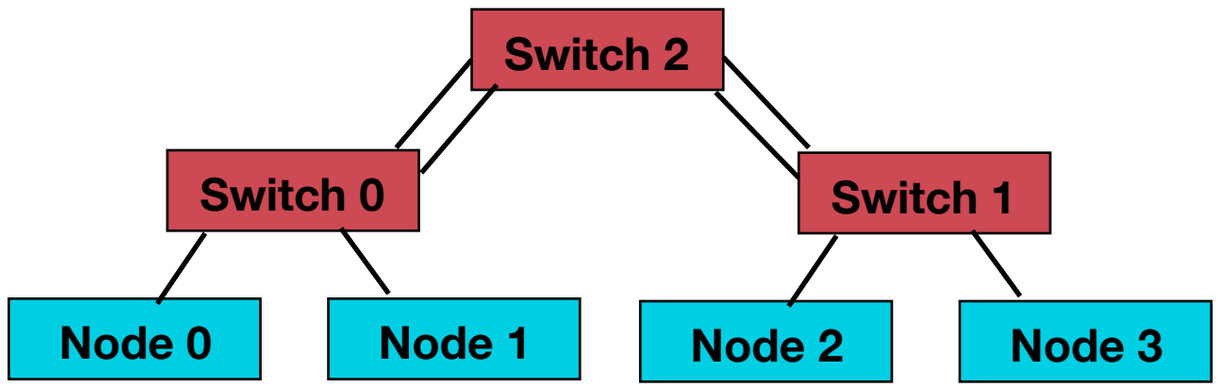
Ring



Crossbar



Fat tree



Modern interconnect technologies

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

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
  - *manycore* chips with wide SIMD units
  - hybrid architectures with deep memory hierarchies
  - special-purpose execution units (e.g. tensor cores for ML)
  - programmable silicon (FPGA)

Questions