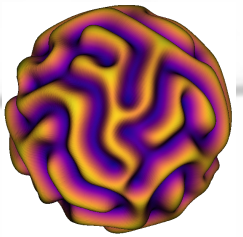
Performance, memory, and parallelism

December 4, 2023

Ian May

Department of Applied Mathematics
University of California Santa Cruz

Santa Cruz, CA



Introduction

So far in this course we haven't talked very much about how to write performant code.

Overview for today

- What performance means
- Hierarchy of parallelism
- Transparent parallelism
- Optimizing compilers
- Memory and cache
- Multithreading and message passing

Acknowledgments: Material partially adapted from Prof. LeVeque's lecture notes on scientific computing

What is performant code?

Expend the smallest time and energy to perform a task

- Problem dependent
- Hardware dependent
- Generally, fully utilizing resources is best
  - Keep the processor saturated

What is performant code?

Expend the smallest time and energy to perform a task

- Problem dependent
- Hardware dependent
- Generally, fully utilizing resources is best
  - Keep the processor saturated

Don’t optimize too early!!

Achieving better performance: The old way

Be patient and buy better hardware

Each generation transistors got smaller, and chips got denser. This had a few major impacts:

- Architecture upgrades came out frequently
- Caches (sometimes) got larger
- Distinct units on the chip were physically closer
- **Clock speeds got faster**
- Floating point units got more advanced

You could speed up your code by simply waiting for a new generation of hardware.

Power limits and clock speeds

Clock speeds have (mostly) stopped increasing. Two major bottlenecks are:

1. Electric signals can only travel a certain distance per clock cycle (speed of light actually matters)
2. Faster clocks require higher power draw
   - Hard to remove waste heat fast enough
   - Hard to design small chips capable of transmitting that power

Power limits and clock speeds

Clock speeds have (mostly) stopped increasing. Two major bottlenecks are:

1. Electric signals can only travel a certain distance per clock cycle (speed of light actually matters)
2. Faster clocks require higher power draw
   - Hard to remove waste heat fast enough
   - Hard to design small chips capable of transmitting that power

A new bottleneck – Memory

Despite stagnation in clock speed increase, most programs are really bottlenecked by the time required to interact with main memory (DRAM).

Power limits and clock speeds

Clock speeds have (mostly) stopped increasing. Two major bottlenecks are:

1. Electric signals can only travel a certain distance per clock cycle (speed of light actually matters)
2. Faster clocks require higher power draw
   - Hard to remove waste heat fast enough
   - Hard to design small chips capable of transmitting that power

A new bottleneck – Memory

Despite stagnation in clock speed increase, most programs are really bottlenecked by the time required to interact with main memory (DRAM).

Key observation: Quite a bit of performance can be gained without parallelism. Understanding the hardware can take you quite far.

Modern computers (can) exploit parallel computing in many ways

1. Bit level parallelism
2. Instruction pipelining
3. Superscalar processors and SIMD
4. (Simultaneous) Multithreading
5. Distributed computing

Some are totally transparent (1,2), mostly transparent (3), or fully explicit (4,5)

Hierarchy of parallelism
An incomplete list

Modern computers (can) exploit parallel computing in many ways

1. Bit level parallelism
2. Instruction pipelining
3. Superscalar processors and SIMD
4. (Simultaneous) Multithreading
5. Distributed computing

Some are totally transparent (1,2), mostly transparent (3), or fully explicit (4,5)

Other routes: Discrete co-processors, e.g. GPUs.

Instruction pipelining

What makes the processor do useful work?

Roughly a five step sequence:

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

Many instructions will require further steps (so-called micro-operations).

Memory access can add indeterminate overhead.

Hardware considerations

Cartoon of a modern CPU

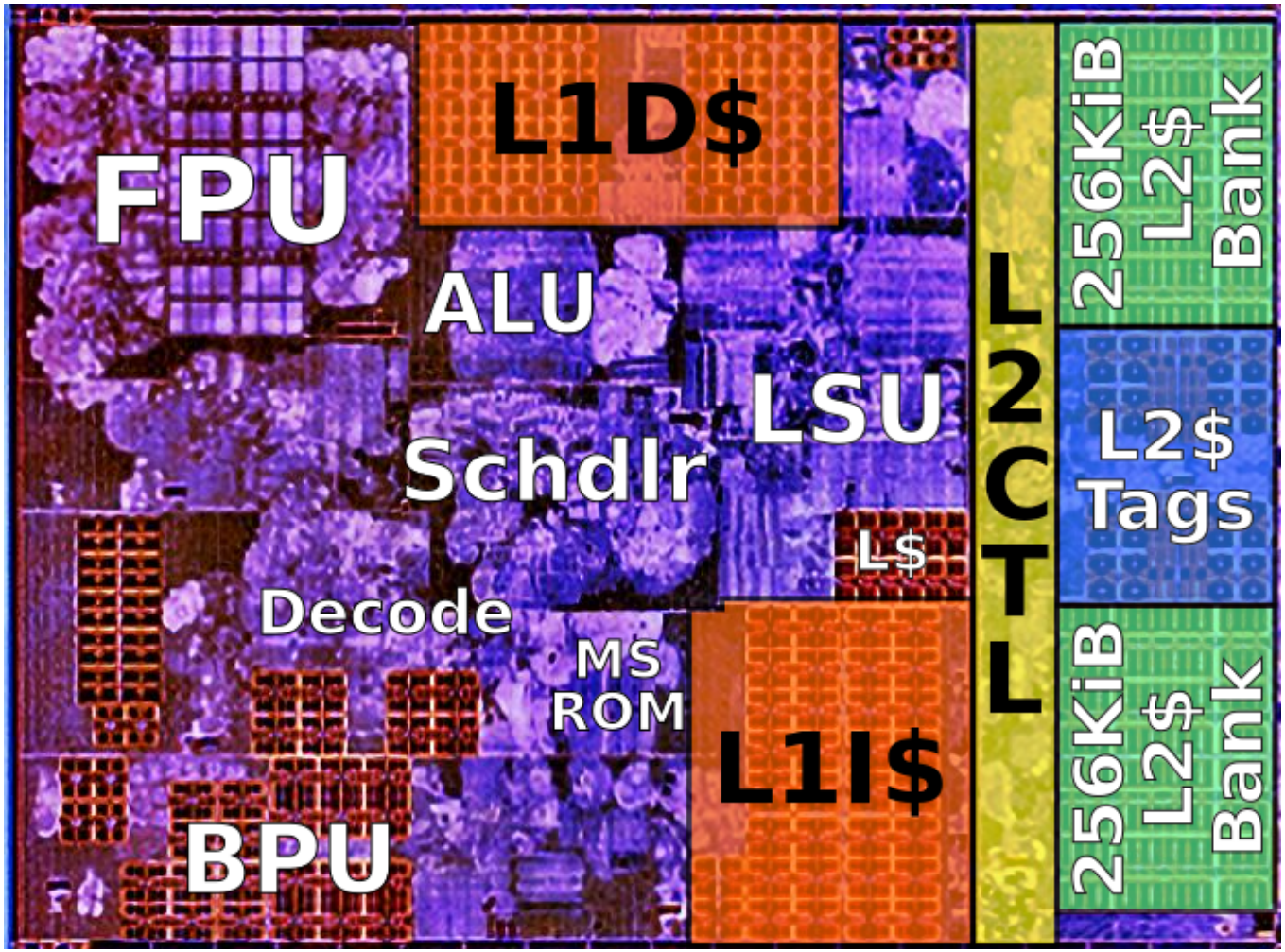


Figure: Labeled die shot of an AMD Zen core.

Instruction pipelining

The steps in the pipeline are processed by distinct units

Idea: Keep all units busy by processing the next instruction before the current one completes.

Some difficulties arise:

- How do you resolve instructions that require a different number of clock cycles?
- How do you handle wait time for memory accesses?
- How do you handle instructions that depend on each other?

There is a lot of tuning involved, but hardware manufacturers have mostly settled on what works well.

Instruction pipelining
Ideal execution

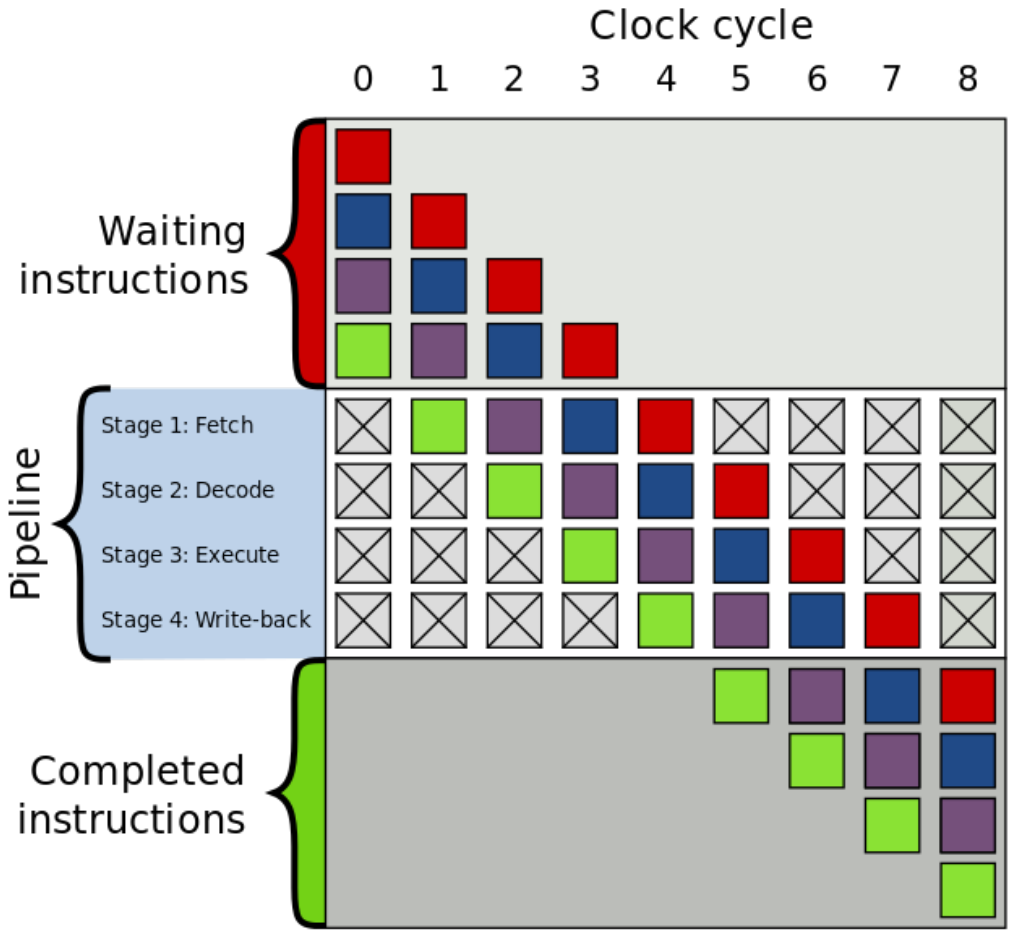


Figure: Schematic of the history of a pipelining processor. Each color corresponds to a distinct instruction. Courtesy of Wikipedia.

Instruction pipelining
Pipeline bubbles

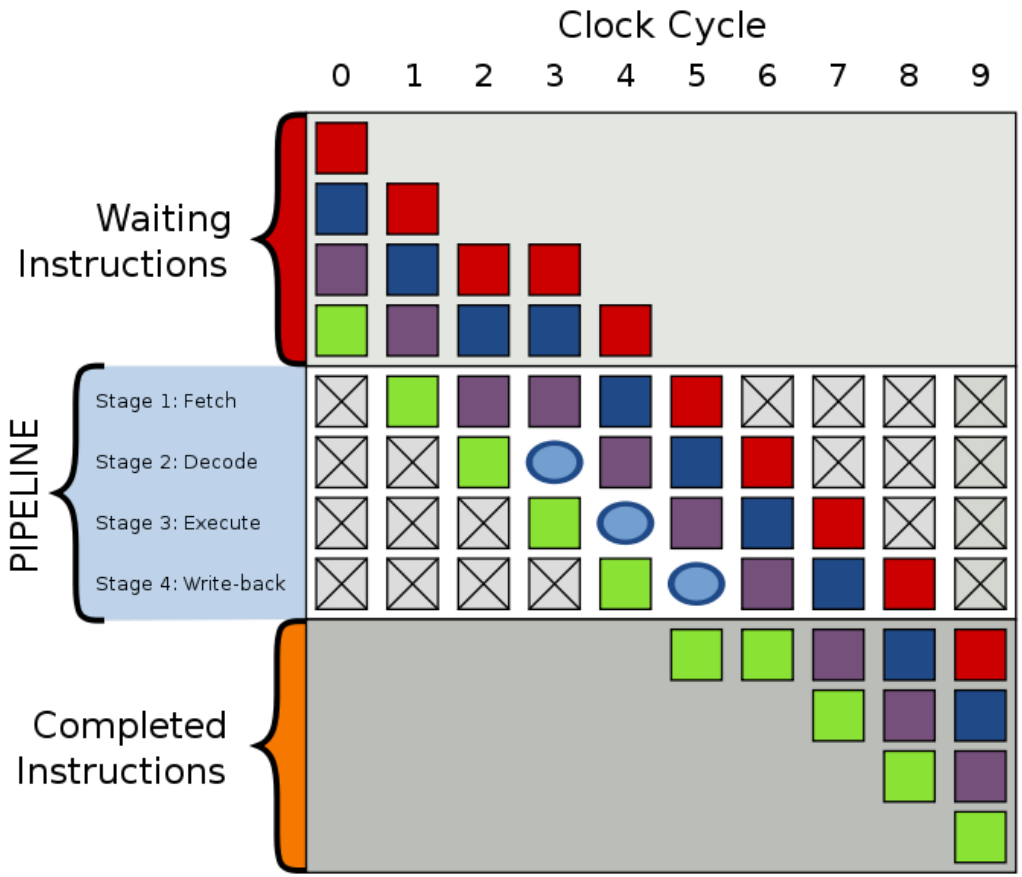


Figure: Pipeline history including a hazard. Courtesy of Wikipedia.

Superscalar processors and SIMD

Consider adding two vectors to each other.

- The same exact operations will be done many times over
- Pipelining and prefetching can hide some cost
- There ought to be a way to optimize this pattern further...

Superscalar processing

Notice that there is one instruction that needs to be applied to many different pieces of data.

SIMD: Single instruction multiple data

Throughput can be improved by packaging data together and acting on it simultaneously. This requires special hardware and instructions.

Linux provides a wealth of information

- The `/proc` directory holds interesting informational files
  - `/proc/cpuinfo` Information about your CPU
  - `/proc/meminfo` Information about your memory
- `lscpu` shows information similar to that in `/proc/cpuinfo`
- `lsmem` shows information similar to that in `/proc/meminfo`
- `lstopo` shows this information diagramatically

Vendor information pages are also incredibly useful.

Transforming human readable code into executable instructions

Optimizing compilers will try to generate better machine instructions without changing the function of a program.

Lots of transformations of the code can be performed automatically:

- Function inlining
- Loop unrolling
- Automatic vectorization (e.g. pack and issue SIMD instructions)
- Operation re-ordering
- Operation fusing (e.g. fused-multiply-add)

Some flags for GCC compilers (`gfortran`, `gcc`, `g++`)

- `-O2` Many generic optimizations, alignment, peepholes, some inlining
- Alternatively, `-O3`, Same as prior but also adds more loop optimizations (and other stuff)
- `-march=native -mtune=native` Optimize the code specifically for the architecture being compiled on. **Binaries will likely not be portable!**
- Try adding `-fopt-info-vec-all`. This will dump info in stderr, capture it with `2>`

Automatic vectorization

What ways can you think of to use SIMD?

We often need to perform the same operation many times over. Think of how many loops we have written that have fairly simple bodies.

Problem: Hard-coding SIMD operations is hard, and not portable. What can we do instead?

Solution: Let the compiler do it for us! Of course, we need to write code that the compiler can actually optimize...

Automatic vectorization

Part of what makes Fortran so fast

The restricted memory model that Fortran assumes is in place (partially) to make automatic vectorization really easy for the compiler.

How to make C similarly fast (gcc)

The auto-vectorization engine underneath `gcc` is the same as the one under `gfortran`. However, C allows much more general memory access.

To promote automatic vectorization you need to restrict C in the same way as in default Fortran. The relevant keyword is `restrict`.

Data is stored in many places on your machine

How many cycles are needed before the processor can interact with data stored in different places?

- Processor register: ~ 1 cycle
- L1 cache: ~ 5 cycles
- L2 cache: $\sim 10 - 20$ cycles
- L3 cache: $\sim 50 - 100$ cycles
- DRAM: ~ 1000 cycles
- SDD/HDD: More, probably...

Many of the optimizations are present to increase cache hit rate.

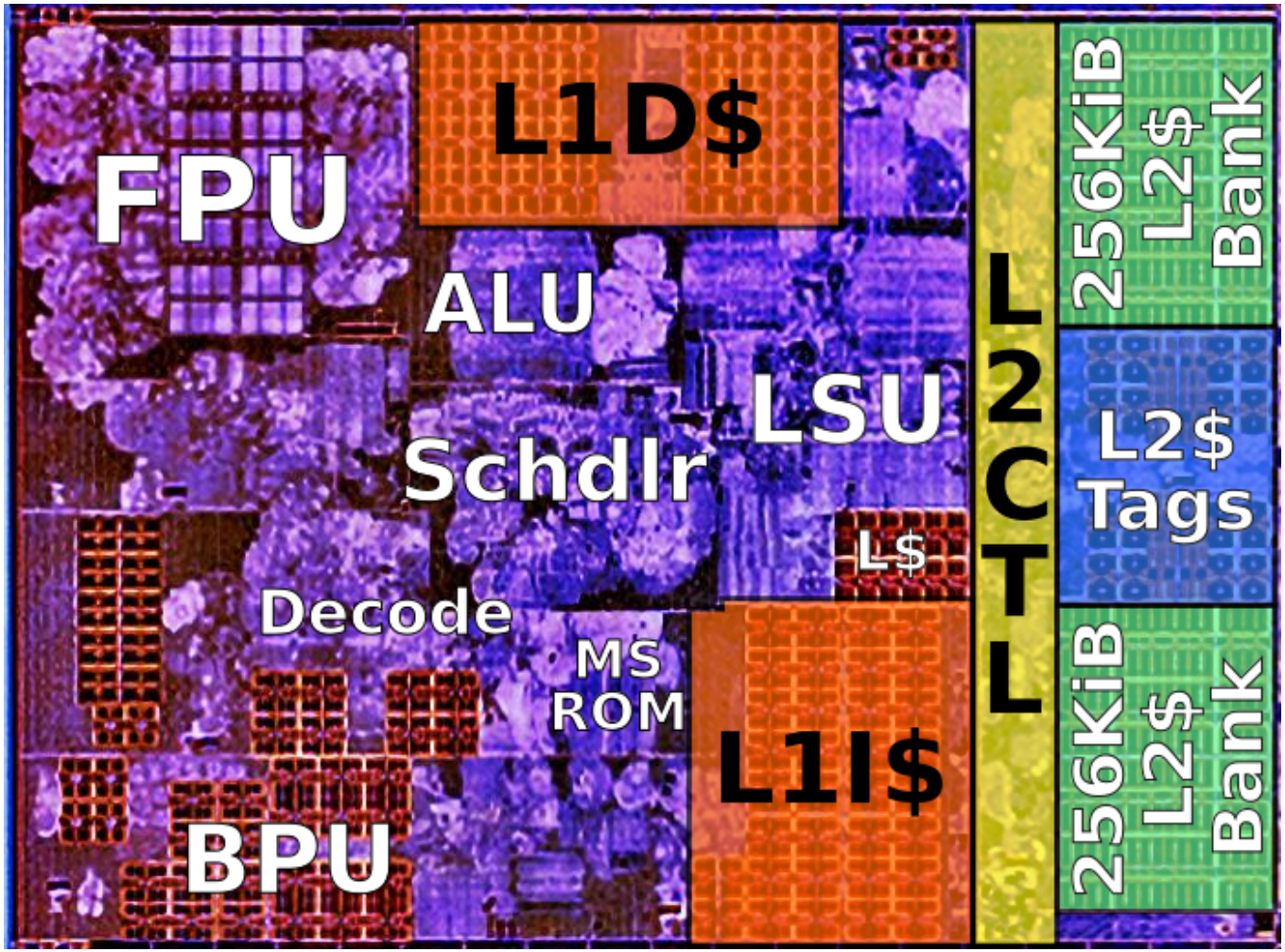


Figure: Labeled die shot of an AMD Zen core.

Being nice to your cache

Consider a sample program

Purpose: fill a (moderately) large matrix, and perform a matrix vector product.

Being nice to your cache

Consider a sample program

Purpose: fill a (moderately) large matrix, and perform a matrix vector product.

Pre-fetching and spatial locality

- Traversing the matrix in the same order as it is stored lets the hardware fetch memory very efficiently
- Traversing in the wrong order is bad
- Traversing in the wrong order with `N=8192` is **very** bad
  - Lots of cache collisions
- The optimizer is very effective, but performs best when the underlying code is reasonable

So far everything we've said applies to a single processing unit (core).
Threads: One (part of a) program running on one core is called a *thread of execution*

So far everything we've said applies to a single processing unit (core).
Threads: One (part of a) program running on one core is called a *thread of execution*

Hardware considerations

- Memory controller mediates access to main memory
- Private L1 caches need to remain *coherent*
- Need to snapshot thread state to handle switching and interrupts

So far everything we've said applies to a single processing unit (core).
Threads: One (part of a) program running on one core is called a *thread of execution*

Hardware considerations

- Memory controller mediates access to main memory
- Private L1 caches need to remain *coherent*
- Need to snapshot thread state to handle switching and interrupts

Software considerations

- OS needs to understand how to schedule threads
- Many serial algorithms require significant re-design to work in parallel
- Atomicity, mutual exclusion, and race conditions, oh my!

Many workloads can be split into distinct, independent, tasks.

OpenMP – Directive based parallelism

- Programmer responsible for dividing up workload and handling dependencies
- A software runtime batches tasks out to available threads/cores
- Pragmas (C/C++) or special comments (Fortran) indicate regions of parallel execution
- Works well for parallel loops, divide and conquer, reductions, etc.
- Memory access patterns more important than ever!

Using more machines
Distributed memory parallelism

Single machine, shared memory parallelism has its limitations.
Workloads can be spread, i.e. distributed, across multiple machines.

Single machine, shared memory parallelism has its limitations.
Workloads can be spread, i.e. distributed, across multiple machines.

Hardware considerations

- Machines need to communicate with each other
- File systems often also distributed
- Lots of complicated networking involved

Single machine, shared memory parallelism has its limitations.
Workloads can be spread, i.e. distributed, across multiple machines.

Hardware considerations

- Machines need to communicate with each other
- File systems often also distributed
- Lots of complicated networking involved

Software considerations

- Each machine needs a copy of the program
- Something needs to synchronize these programs
  - Manage communication
  - Propagate faults/failures
- Programmer needs to think in a fundamentally different way

Message passing interface (MPI)

By far, the most common way to handle these software complexities is the *Message passing interface*.

The programming model

- A software runtime coordinates all involved machines
- Runtime can send/receive messages between distinct machines
- The user doesn't need to know much about the HW implementation
- There is no concept of serial regions and parallel regions
- There are no shared variables between processes
- MPI is just a standard
  - Common implementations: MPICH, OpenMPI, MVAPICH

GPUs and Coprocessors

You've undoubtedly heard about GPU computing. What makes this approach so amazing?

GPUs and Coprocessors

You've undoubtedly heard about GPU computing. What makes this approach so amazing?

Heterogeneous computing

Key idea: Use different hardware for different tasks, allowing more aggressive optimization.
GPUs have a few distinct features:

- Massively parallel, 1000's of cores
- Substantially different memory architecture, often no enforced coherence
- Minimal instruction pipelining
- Hardware-based thread scheduler

GPUs can provide massive throughput, **but only for appropriate workloads**.

Where to learn more

Applied mathematics

- AM 250: An Introduction to High Performance Computing
  - Provides a nice introduction to thinking in terms of distributed memory parallelism
- AM 148: GPU Programming for Scientific Computations
  - A good introduction to the particularities of GPUs

Computer science and engineering

- CSE 113: Parallel Programming
  - I can't directly speak about this one
- CSE 226: Advanced Parallel Processing
  - Lot's of detail about hardware internals and their design