CS 5220

Single Core Architecture

David Bindel

2024-09-03

Logistics

Office hours

- Bindel missing OH this week (new grad student social)
- Caroline and Evan will still have office hours
- Appointment and Ed are also options!

Enrollment

- Use ticket system for enrollment issues
- HW0 due Sep 5 via CMS
- Add deadline is Sep 9
- We are currently at 125!

CIS Partner Finding Social

Searching for a study buddy or partner? Looking to meet a new friend? Are you taking CS, INFO, or ORIE classes? If so, the CIS Partner Finding Social is for you! This is the PERFECT opportunity to find a partner and meet other students in your classes, so join us on September 11th at 5-7pm in Duffield Atrium!

Computing Systems

You will be using three systems (if enrolled!) – see email.

- Perlmutter
- Google Cloud Platform

Perlmutter and GCP are Unix environments. Recommended for local work:

- Mac: Terminal and homebrew
- Windows: Windows Subsystem for Linux

You can also develop remotely.

Computing Systems

- Perlmutter
  - Fill out the CMS survey for your login ID
  - Make sure you have an MFA token set up
- GCP
  - Request and redeem GCP coupon
  - Try the console on cloud.google.com

The Toolbox

You will want to know:

- Covered tools: Compilers, profilers, modules
- Demonstrated: git, make
- "You'll pick up": Unix shell, editors

Resources:

- Software Carpentry
- The Missing Semester
- Cornell virtual workshops

Just for Fun

Mythbusters pitch NVidia!

Rage Against the Machine

Idealized Machine

- Address space of named words
- Basic ops: register read/write, logic, arithmetic
- Everything runs in program order
- High-level language means “obvious” machine code
- All operations take about the same time

Real World

Memory operations are *not* all the same!

- Speeds vary (registers and caches)
- Memory layout dramatically affects performance

Real World

Instructions are non-obvious!

- Pipelining allows instructions to overlap
- Functional units run in parallel (and out of order)
- Instructions take different amounts of time
- Cost depends on order, instruction mix

Real World

Goal: Understand how to help the compiler.

Sketching Reality

Today, a play in two acts:

1. One core is not so serial
2. Memory matters

Act 1: Not So Serial

Laundry

- Three stages: wash, dry, fold
- Three loads: darks, lights, underwear
- How long?

Laundry

Serial execution:

1	2	3	4	5	6	7	8	9
wash	dry	fold						
			wash	dry	fold			
						wash	dry	fold

Laundry

Pipelined execution:

1	2	3	4	5
wash	dry	fold		
	wash	dry	fold	
		wash	dry	fold

RISC Pipeline

Classic five-stage pipeline (MIPS and company)

1	2	3	4	5	6	7	8	9
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

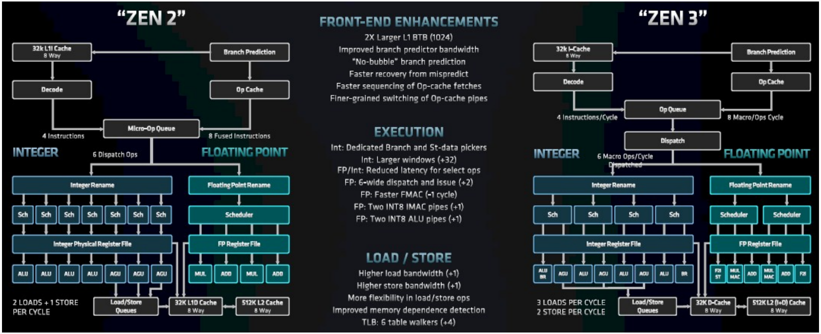
RISC Pipeline

- **Fetch** - read instruction and increment PC
- **Decode** - determine registers and addresses
- **Execute** - where the actual computation occurs
- **Memory** - any memory accesses
- **Writeback** - results into register file

Pipelining

- Improves *bandwidth*, not *latency*
- Potential speedup = number of stages
    - What if there’s a branch?

AMD Milan (Zen 3)



FRONT-END ENHANCEMENTS

2X Larger L1 BTB (1024)
Improved branch predictor bandwidth
"No-bubble" branch prediction
Faster recovery from mispredict
Faster sequencing of Op-cache fetches
Finer-grained switching of Op-cache pipes

EXECUTION

Int: Dedicated Branch and St-data pickers
Int: Larger windows (+32)
FP/Int: Reduced latency for select ops
FP: 6-wide dispatch and issue (+2)
FP: Faster FMAC (-1 cycle)
FP: Two INT8 IMAC pipes (+1)
FP: Two INT8 ALU pipes (+1)

LOAD / STORE

Higher load bandwidth (+1)
Higher store bandwidth (+1)
More flexibility in load/store ops
Improved memory dependence detection
TLB: 6 table walkers (+4)

Current versions are much more complicated!

Wide Front-End

Fetch/decode or retire multiple ops at once

- Limited by instruction mix
  - Different ops use different port
- NB: May dynamically translate to micro-ops

Hyperthreading

Support multiple HW threads/core

- Independent registers, program counter
- Shared functional units
- Helps feed core independent work

Out-of-Order Execution

- Internally reorder operations
- Have to *commit* results in order
- May discard uncommited results (speculative execution)
- Limited by data dependencies

SIMD (Vectorization)

- Single Instruction Multiple Data
- Cray-1 (1976): 8 registers × 64 words of 64 bits
- Resurgence in mid-late 90s (for graphics)
- Now short vectors (256-512 bit) are ubiquitous

Pipelining

Different pipelines for different units

- *Front-end* has a pipeline
- *Functional units* have their own pipelines
    - Example: FP adder, multiplier
    - Divider often *not* pipelined

All Together, Now...

- Front-end reads several ops at once
- Ops may act on vectors (SIMD)
- Break into mystery micro-ops (and cache)
- Out-of-order scheduling to functional units
- Pipelining within functional units
- In-order commit of finished ops
- Can discard before commit
  (speculative execution)

All Together, Now...

Modern single-core architecture is complex! Desiderata

- Maintain (mostly) serial semantics
    - In-order retirement, precise exceptions
- Make lots of latent parallelism available
    - Wide issue, SIMD, pipelining
- Help programmer/compiler manage complexity
    - Out-of-order execution

Punchline

Compiler understands CPU *in principle*

- Rearranges instructions to get a good mix
- Tries to use FMAs, SIMD instructions, etc

Punchline

Compiler needs help in practice

- Set optimization flags, pragmas, etc
- Make code obvious and predictable
- Expose local independent work
- Use special intrinsics or library routines
- Data layouts, algorithms to suit machine

Punchline

The goal:

- You handle high-level optimization
- Compiler handles low-level stuff

Note **memory layouts** are part of your job!

Act 2: Memory Matters

Basic Problem

- Memory *latency* = how long to get a requested item
- Memory *bandwidth* = steady-state rate
- Bandwidth improves faster than latency
- Inverse bandwidth remains worse the flop rate

Locality

Programs usually have *locality*:

- **Spatial locality**: things close to each other tend to be accessed consecutively.
- **Temporal locality**: we tend to use a “working set” of data repeatedly.

The *cache hierarchy* is built to take advantage of locality.

How Caches Help

- Hide memory costs by reusing data
    - Exploit temporal locality
- Use bandwidth to
    - Fetch by *cache line* (spatial locality)
    - Support multiple reads
    - Prefetch data

This is mostly automatic and *implicit*.

Cache Basics

- Organize in cache *lines* of several bytes
- Cache *hit* when copy of needed data in cache
- Cache *miss* otherwise. Basic types:
    - *Compulsory*: never used this data before
    - *Capacity*: cache full, working set too big
    - *Conflict*: insufficient associativity for access pattern

Cache Associativity

Where can data go in cache?

- Direct-mapped: Each address can go in only one location (e.g. store address xxxx1101 only at cache location 1101)
- n-way: Each address can go in one of n possible cache locations (store up to 16 words with addresses xxx1101 at cache location 1101).

Higher associativity costs more in hardware.

Teaser

We have $N = 10^6$ two-dimensional coordinates and want their centroid. Which of these is faster and why?

1. Store an array of (x_i, y_i) coordinates. Loop i and simultaneously sum the x_i and the y_i.
2. Store an array of (x_i, y_i) coordinates. Loop i and sum the x_i, then sum the y_i in a separate loop.
3. Store the x_i in one array, the y_i in a second array. Sum the x_i, then sum the y_i.

Caches on My Laptop

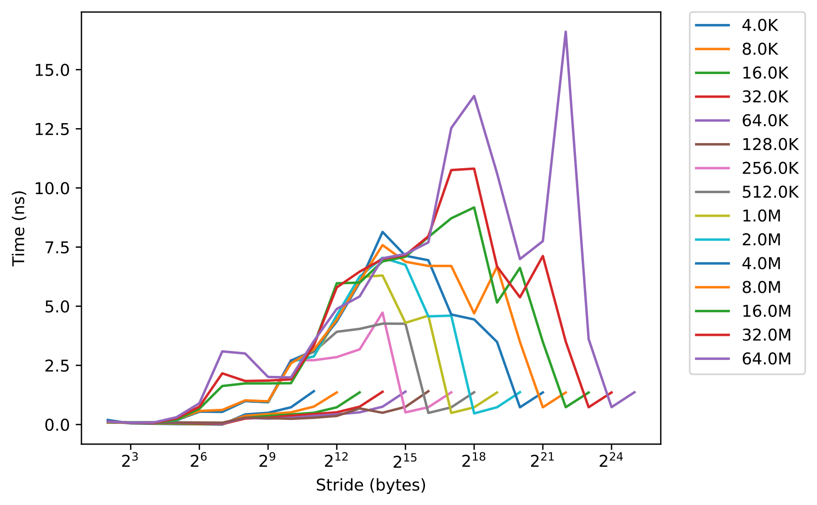
Apple M1 Pro (Firestorm core?)

- 128 KB L1 data cache
- 12 MB L2 cache (shared)
- 24 MB system level cache

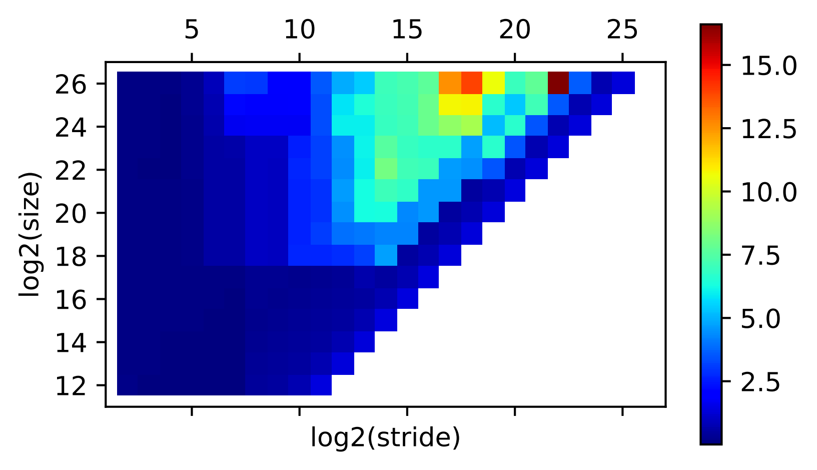
A memory benchmark (Membench)

```
/* Time the loop with strided access + loop overhead */
int steps = 0;
double start = omp_get_wtime();
do {
    for (int i = SAMPLE*stride; i != 0; i--)
        for (int index = 0; index < limit; index += stride)
            x[index]++;
    steps++;
    sec0 = omp_get_wtime()-start;
} while (sec0 < RTIME);
```

Membench on My Laptop



Membench on My Laptop



Features

- Vertical: 128 B cache lines (2^6), 16 KB pages (2^{14})
- Horizontal: 128 KB L1 (2^{17}), 12 MB L2 ($< 2^{24}$)
- Diagonal: 8-way set assoc, 256 page L1 TLB, 3072 page L2 TLB

The Moral

Even for simple programs, performance is a complicated function of architecture!

- Need to know a little to write fast programs
- Want simple models to understand efficiency
- Want tricks to help design fast codes
  - Example: *blocking* (also called *tiling*)