

CS 4414/5416: Systems Programming

Performance, ILP, and memory

David Bindel

2026-09-15

Logistics

- Class sheets: you might not like them, the course staff definitely don't! So:
 - PollEverywhere for attendance and in-class polling
 - Freebie this lecture to let you get set up
 - Ed threads for some longer responses
- Lab submissions will switch to just GradeScope
- Also, PDF slides are now available on the course page

- I know you have an exam!
 - Initial tasks are straightforward (blocking for performance)
 - Rest has straightforward baseline with room to play
- Will have timing targets on GCP nodes

- Practice exam up soon!
- Exam from 7:30-9:30 (target 7:30-9:00) on 9/22
 - NetID A-L: Olin 155; M-V: Olin 255; W-Z: Olin 165
 - Early exams (5:30) in Statler 396
- There is a conflict form if main time does not work
 - **Fill it out today**
 - SDS exam accommodations should also be sorted by now
- Covers the Rust material (first three weeks)

This module: fundamentals of *single-thread* performance

- Hardware effects (particularly memory)
- Performance modeling and measurement
- Using the compiler and tools effectively

Concurrency and multi-threaded performance comes later.

Rage Against the 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

Memory operations are *not* all the same!

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

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

Goal: Understand how to help the compiler.

Today, a play in two acts:

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

Act 1: Not So Serial

- 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

Pipelined execution:

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

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

- **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

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

Current versions are much more complicated!

We will be using AMD EPYC Turin (the **c4d-highcpu-2** GCP instances) for timing P2.

Fetch/decode or retire multiple ops at once

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

Support multiple HW threads/core

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

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

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

Different pipelines for different units

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

- 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)

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

Compiler understands CPU *in principle*

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

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

The goal:

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

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

Act 2: Memory Matters

- 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

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.

- 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*.

- 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

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.

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 .

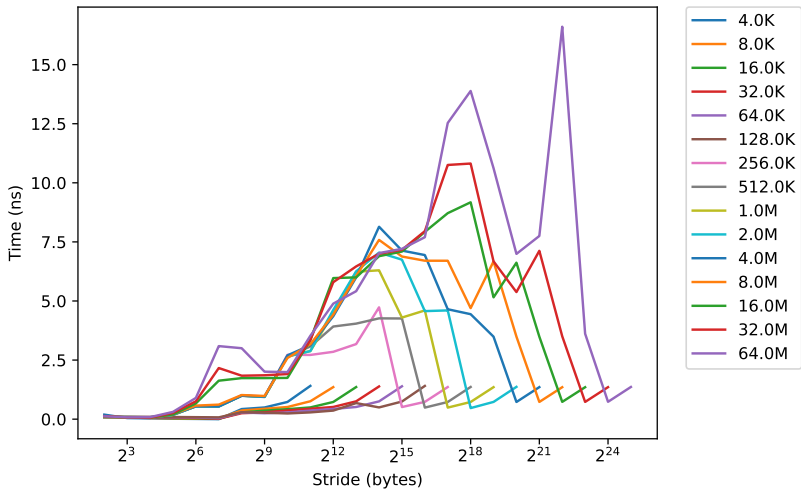
Apple M1 Pro (Firestorm core?)

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

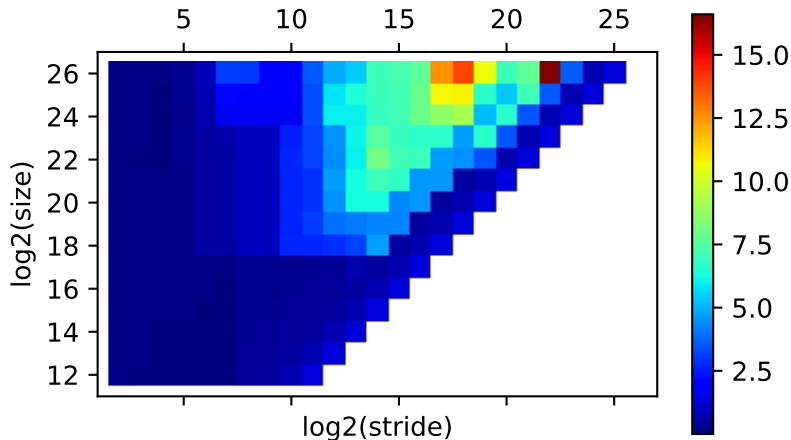
A memory benchmark

```
/* 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



- 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

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*)

- Ed Thread: Have you done any performance tuning elsewhere (courses, internships, personal projects)? Anything you are particularly proud of?