

CS 4414/5416: Systems Programming

Tuning Code and Data

David Bindel

2026-09-22

Logistics

- Prelim tonight! 7:30-9:30:
 - NetID A-L: Olin 155
 - NetID M-V: Olin 255
 - NetID W-Z: Olin 165
- Early makeup (5:30-7:30) in Statler 396
- ATP starts at 5:30

- Lab 4 and P2 check-in extended to 9/23
 - *Do* turn off your VM when not in use
- Lab 5 is Friday: inspecting assembly, FFIs
 - This may be useful for P2 as well
- P2 final extended to 9/30
 - Gradescope will check basic correctness
 - Timing targets on **c4d** are posted

Note re Github

Workflow recommendation (keep your stuff separate!):

```
# Clone the reference repo, then re-clone for solution version
git clone git@github.coecis.cornell.edu:cs4414-f26/p2_edit.git
git clone p2_edit p2_edit_mine

# In the solution version, rename origin (allow future pulls)
cd p2_edit_mine
git remote rename origin upstream

# Add origin
git remote add origin git@github.coecis.cornell.edu:dsb253/p2_edit.git
git branch -M main
git push -u origin main
```

NB: Can't accidentally push to **upstream** (not bare)

- Read the materials
- Come by office hours (or post on Ed) for help
- *Ask a question*

Recap

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.

- Single core exhibits lots of ILP
 - Compiler is good at helping out
 - Helps to code so independent ops are obvious
- But often memory is the bottleneck
 - Favor compact data, limit working sets, regular access

Combine HW context, tricks, and principles:

- Think before you write (Thu)
- Stand on the shoulders of giants (Thu)
- Time before you tune (Thu)
- Tune your data structures (today)
- Help tools help you (today/Thu)

A certain computation repeatedly access the first word of each element of a large array in which each element type is `[usize; 1024]`. Cache re-use in this setting is terrible! Is this because of

- Compulsary misses?
- Capacity misses?
- Conflict misses?

Tuning Project

P2: Mean Hamming distances in a dictionary

- $d(s_i, s_j)$ is swaps to convert s_i to s_j .
- Goal: $\bar{d}_i = N^{-1} \sum_{j=1}^N d(s_i, s_j)$
- $O(N^2)$ computation on $O(N)$ data

Tuning knocks a *lot* of time off the base code!

- Blocked version (improve cache use) - check-in version
- Packed version (reduce pointer chasing)
- SWAR version (bit twiddling trickery for parallelism)
- Fast version (for me: SWAR + symmetry + care with memory)

- SIMD = Single Instruction, Multiple Data
 - Do *same operation* across a vector of data
- Explicit vector ops for standard sizes on most HW
- SIMD Within a Register for non-standard sizes

We use two primitives:

- Test equality within field
- Count nonzero fields

Ex: Compare two bit numbers (1, 3).

- Store with one extra bit: `0b001`, `0b011`
- Exclusive or zero iff same numbers: `0b010`
- Add to max two-bit value: `0b010 + 0b011 = 0b101`
 - Generates carry iff other term is nonzero
- Mask the carry bit: `0b101 & 0b100 = 0b100`
- Shift carry down by two bits: `0b001`

- Consider (1,0,1) in three five-bit fields
- Encoding fits in a `u16`:
 - 1 in 1s place
 - 0 in $2^5 = 32$ place
 - 1 in $2^{10} = 1024$ place
- Note: $2^5 = 2^{10} = 1 \pmod{31}$
- Sum: $2^{10} + 1 = 1 + 1 = 2 \pmod{31}$

Hamming distance between tuples of three four-bit numbers:

Tuple	Fields
$x = (1, 5, 10)$	0b00001_00101_00101
$y = (2, 5, 7)$	0b00010_00101_00111
$d = x \wedge y$	0b00011_00000_00010
$s = d + 0b01111 \text{ (x3)}$	0b10010_01111_10110
$c = s \& 0b10000 \text{ (x3)}$	0b10000_00000_10000
$r = c \gg 4$	0b00001_00000_00001
$z = r \% 31$	2

Platform	Dictionary	Naive	Blocked	Packed	SWAR	Fast
M1	popular	5.3s	4.1s	2.9s	0.95ms	0.54s
M1	enable1	259.0s	222.0s	94.5s	43.3s	24.3s
c4d	popular	4.3s	2.9s	2.7s	1.3s	0.64s
c4d	enable1	221.7s	160.1s	126.4s	61.7s	33.3s

SWAR/Fast: 2ns (10 cycle at 5GHz boost) per comparison.

Performance Rubrics

Performance scoring thresholds (C4D and **enable**)

Score (of 2)	Blocked	Packed	SWAR
3	175s	140s	70s
2	185s	150s	80s
1	195s	160s	90s

No credit for fast-but-broken.

- Also one point each for **WRITEUP.md** answers (eight Q)!
- Also one point for a crate that compiles!

Ask Me a Question!

Tune Your Data Structures

For compulsory misses:

$$T_{\text{data}} \text{ (s)} \geq \frac{\text{data required (bytes)}}{\text{peak BW (bytes/s)}}$$

Possible optimizations:

- Shrink working sets to fit in cache (pay this once)
- Use simple unit-stride access patterns

Reality is more complicated...

Access is not the only cost!

- Allocation/de-allocation also costs something
- So does GC (where supported)
- Beware hidden allocation costs (e.g. on resize)
- Often bites naive library users

Two thoughts to consider:

- Preallocation (avoid repeated alloc/free)
- Lazy allocation (if alloc will often not be needed)

Desiderata:

- Compact (fits lots into cache)
- Traverse with simple access patterns
- Avoids pointer chasing

- SoA: Structure of Arrays
 - Friendly to vectorization on field
 - Poor locality to access all of one item
- AoS: Array of Structs
 - Great for computing on all fields per item
 - Not SIMD friendly on single fields

Can use **zip** to iterate over SoA like AoS.

Can copy between formats to accelerate, e.g.

- Copy piece of AoS to SoA format
- Perform vector operations on SoA data
- Copy back out

Performance gains > copy costs?

Plays great with tiling!

Can get (some) programmer control over

- Pre-fetching
- Uncached memory stores

But usually best left to compiler / HW.

- Things we love in memory layouts
 - Small working sets – fit in cache!
 - Short types – more per cache line!
 - Regular stride – for cache and vectorization
- Don't love
 - Pointer chasing
 - Unclear bounds
 - Extra allocations/drops

Consider computing longest common prefixes over pairs of strings. For this computation, would it be more reasonable to store this as a structure of arrays or an array of structures?

- Structure of Arrays
- Array of Structures

Help Tools Help You

- Linters (like clippy)
- Profilers (like samplify)
- Benchmark tools (like criterion)
- Static analysis (like llvm-mca)
- Dynamic analysis (like cachegrind)
- And, of course, the compiler...

How Do Compilers Help?

In decreasing order of effectiveness:

- Local optimization
 - Especially restricted to a “basic block”
 - More generally, in “simple” functions
- Loop optimizations
- Global (cross-function) optimizations

- Compiler can almost always inline!
- Global optimization across full crate
- Lots of type information is available to compiler
- Can safely make local copy from shared reference

- Register allocation: compiler > human
- Instruction scheduling: compiler > human
- Branch joins and jump elim: compiler > human?
- Constant folding and propagation: humans OK
- Common subexpression elimination: humans OK
- Algebraic reductions: humans definitely help

Mostly leave these to modern compilers

- Loop invariant code motion
- Loop unrolling
- Loop fusion
- Induction variable substitution
- Software pipelining (mostly)
- Vectorization (with caveats)

Aside: Software Pipelining

```
fn do_laundry(v: &mut [Laundry]) {  
    for x in v.iter_mut() {  
        x.wash();  
        x.dry();  
        x.fold();  
    }  
}
```

This one is worth knowing for later...

Aside: Software Pipelining

```
// Assume v is long enough
fn do_laundry(v: &mut [Laundry]) {
    let i = v.iter_mut();
    let mut x0 = i.next();
    let mut x1 = i.next();
    x0.wash();
    x0.dry();
    x1.wash();
    for x2 in i {
        x2.wash();
        x1.dry();
        x0.fold();
        x0 = x1;
        x2 = x2;
    }
}
```

Obstacles for the Compiler

- Long dependency chains
- Excessive branching
- Aliasing (Rust beats C!)
- Complex loop logic
- Cross-module optimization

Obstacles for the Compiler

- Function pointers and virtual functions
- Pointer chasing
- Unexpected FP costs
- Missed algebraic reductions
- Lack of instruction diversity

Let's look at a few...

Long Dependency Chains

Sometimes these can be decoupled. Ex:

```
fn sum1(arr: &[f32]) -> f32 {  
    arr.into_iter().sum()  
}
```

Apparent linear dependency chain:

```
let t0 = arr[0];  
let t1 = t0 + arr[1];  
let t2 = t1 + arr[2];  
// etc
```

(Yes, Rust 1.98 algebraic add might auto-optimize.)

Long Dependency Chains

```
fn sum2(arr: &[f32]) -> f32 {  
    const B: usize=4;  
    let mut results = [0f32; B];  
    let l = (arr.len()/B)*B;  
    for i in (0..l).step_by(B) {  
        for k in 0..B {  
            results[k] += arr[i+k];  
        }  
    }  
    for i in l..arr.len() {  
        results[0] += arr[i];  
    }  
    results.into_iter().sum()  
}
```

Auto-vectorizer probably kicks in here...

Why can this not vectorize easily?

```
void add_vecs(int n, double* result, double* a, double* b)
{
    for (int i = 0; i < n; ++i)
        result[i] = a[i] + b[i];
}
```

Q: What if `result` overlaps `a` or `b`?

- C/C++ allow pointer aliasing
 - C `restrict` promises no aliasing
 - C++ doesn't have standardized version
- Fortran forbids aliasing!
- Rust borrow rules outlaw problem
 - Issue: What if memory underlying `&x` changes?
 - Solution: No writers allowed in this case!

“Black Box” Calls

Compiler assumes arbitrary wackiness:

```
void foo(double* restrict x)
{
    double y = *x; // Load x once
    bar();         // Assume bar is a 'black box' fn
    y += *x;       // Must reload x
    return y;
}
```

Rust borrow checker again prevents this!

A Remaining Gotcha

```
fn matvecs(a: &[f64], x: &[f64], n: usize) -> Vec<f64> {  
    let result = vec![0f64; n];  
    for i in 0..n {  
        result[i] = a[i*n+i] * x[i];  
        for j in i+1..n {  
            let z = a[i*n+j] * x[j];  
            result[i] += z;  
            result[j] += z;  
        }  
    }  
}
```

Harder to manage successive writes to same array!

Several possible optimizations:

- Use different precisions
- Use more/less accurate special function routines
- Underflow as flush-to-zero vs gradual

But these change semantics! Often needs a human.

Wrap-Up

Hardware context, some tricks, and some principles:

- Think before you write (last week)
- Stand on shoulders of giants (last week)
- Time before you tune (last week)
- Tune your data structures
- Help your tools help you (continues Thu)

- Lab 4 and P2 check-ins delayed to Weds (9/23)
- P2 submission also delayed, due 9/30
 - Performance thresholds are on **c4d** nodes

No outro Q today, good luck on the exam!