CS 5220

Basic Code Optimization

David Bindel

2024-09-05

Reminder

- Modern CPUs are wide, pipelined, out-of-order
  - Want good instruction mixes, independent operations
  - Want vectorizable operations
- Communication (including with memory) is slow
  - Caches provide intermediate cost/capacity points
  - Designed for spatial and temporal locality

(Trans)portable Performance

- Details have orders-of-magnitude impacts
- But systems differ in micro-arch, caches, etc
- Want *transportable* performance across HW
- Need *principles* for high-perf code (+ tricks)

Principles

- Think before you write
- Time before you tune
- Stand on shoulders of giants
- Help your tools help you
- Tune your data structures

Think Before You Write

We should forget about small efficiencies, say 97% of the time: premature optimization is the root of all evil.
- Knuth, Structured programming with go to statements, Computing Surveys (4), 1974.

Premature Optimization

... Yet we should not pass up our opportunities in that critical 3%.
- Knuth, Structured programming with go to statements, Computing Surveys (4), 1974.

Premature Optimization

- At design time, think *big* efficiencies
- Don’t forget the 3%!
- And the time is not premature forever!

Functionality First

No prize for speed of wrong answers.

Lay-of-the-Land Thinking

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        for (int k = 0; k < n; ++k)
            C[i+j*n] += A[i+k*n] * B[k+j*n];
```

- What are the “big computations” in my code?
- What are natural algorithmic variants?
  - Vary loop orders? Different interpretations!
  - Lower complexity algorithm (Strassen?)
- Should I rule out some options in advance?
- How can I code so it is easy to experiment?

Don't Sweat the Small Stuff

- Fine to have high-level logic in Python and company
- Probably fine not to tune configuration file readers
- Maybe OK not to tune $O(n^2)$ prelude to $O(n^3)$ algorithm?
    - Depending on n and on the constants!

How Big?

Typical analysis: time is $O(f(n))$

- Meaning: $\exists C, N : \forall n \geq N, T_n \leq Cf(n)$
- Says *nothing* about constants: $O(10n) = O(n)$
- Ignores lower-order term: $O(n^3 + 1000n^2) = O(n^3)$

Beware asymptotic complexity analysis for small n!

Avoid Work

Asymptotic complexity is not everything, but:

- Quicksort beats bubble sort for modest n
- Counting sort even faster for modest key space
- No time at all if data is already sorted!

Pick algorithmic approaches thoughtfully.

Be Cheap

Our motto: Fast enough, right enough

- Want: time saved in compute $\gg$ time taken in tuning
  - Your time costs more than compute cycles
  - No shame in a slow workhorse that gets the job done
- Maybe an approximation is good enough?
  - Depends on application context
  - Answer usually requires error analysis, too

Do More with Less (Data)

Want lots of work relative to data loads:

- Keep data compact to fit in cache
- Short data types for better vectorization
- But be aware of tradeoffs!
  - For integers: May want 64-bit ints sometimes!
  - For floating point: More in other lectures

Remember the I/O

Example: Explicit PDE time stepper on 256^2 mesh

- 0.25 MB per frame (three fit in L3 cache)
- Constant work per element (a few flops)
- Time to write to disk $\approx$ 5 ms

If I write once every 100 frames, how much time is I/O?

Time Before You Tune

Back to Knuth

It is often a mistake to make a priori judgements about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail.
- Knuth, Structured programming with go to statements, Computing Surveys (4), 1974.

Hot Spots and Bottlenecks

- Often a little bit of code takes most of the time
- Usually called a “hot spot” or bottleneck
- Goal: Find and remove (“de-slugging”)

Practical Timing

Things to consider:

- Want *high-resolution* timers
- Wall-clock time vs CPU time
- Size of data collected vs how informative it is
- Cross-interference with other tasks
- Cache warm-start on repeated timings
- Overlooked issues from too-small timings

Manual Instrumentation

Basic picture:

- Identify stretch of code to be timed
- Run several times with “characteristic” data
- Accumulate time spent

Caveats: Effects from repetition, “characteristic” data

Manual Instrumentation

- Was hard to get *portable* high-resolution wall-clock time!
    - Things have improved some...
- If OpenMP available: `omp_get_wtime()`
- C11 `timespec_get`
- C++ `std::chrono::high_resolution_clock`

Profiling Tools

- **Sampling**: Interrupt every t_{profile} cycles
- **Instrumenting**: Rewrite code to insert timers
    - May happen at binary or source level

Time Attribution

May time at *function level* or *line-by-line*

- *Function*: Can still get mis-attribution from inlinining
- *Line-by-line*: Attribution is harder, need debug symbols (`-g`)

More Profiling Details

- Distinguish full call stack or not?
- Time full run, or just part?
- Just timing, or get other info as well?

Hardware Counters

- Counters track cache misses, instruction counts, etc
- Present on most modern chips
- *But* may require significant permissions to access

Symbolic Execution

- Main current example: `llvm-mca`
- Symbolically execute assembly on *model* of core
- Usually only practical for short segments
- Can give detailed feedback on (assembly) quality

Shoulders of Giants

What Makes a Good Kernel?

Computational kernels are

- Small and simple to describe
- General building blocks (amortize tuning work)
- Ideally high arithmetic intensity
  - Arithmetic intensity = flops/byte
  - Amortizes memory costs

Case Study: BLAS

Basic Linear Algebra Subroutines

- Level 1: $O(n)$ work on $O(n)$ data
- Level 2: $O(n^2)$ work on $O(n^2)$ data
- Level 3: $O(n^3)$ work on $O(n^2)$ data

Level 3 BLAS are key for high-perf transportable LA.

Other Common Kernels

- Apply sparse matrix (or sparse matrix powers)
- Compute an FFT
- Sort an array

Kernel Tradeoffs

- Critical to get *properly tuned* kernels
- *Interface* is consistent across HW types
- *Implementation* varies by archiecture
- General kernels *may* leave performance on table
    - Ex: General matrix ops for structured matrices
- Overheads may be an issue for small n cases

Kernel Tradeoffs

Building on kernel functionality is not perfect –

But: Ideally, someone else writes the kernel!

(Or it may be automatically tuned)

Help Tools Help You

How can Compiler Help?

In decreasing order of effectiveness:

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

Local Optimizations

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

Loop Optimization

Mostly leave these to modern compilers

- Loop invariant code motion
- Loop unrolling
- Loop fusion
- Software pipelining
- Vectorization
- Induction variable substitution

Obstacles for the Compiler

- Long dependency chains
- Excessive branching
- Pointer aliasing
- Complex loop logic
- Cross-module optimization

Obstacles for the Compiler

- Function pointers and virtual functions
- 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:

```
// Version 0
float s = 0;
for (int i = 0; i < n; ++i)
    s += x[i];
```

Apparently linear dependency chain.

```c
// Version 1
float ss[4] = {0, 0, 0, 0};
int i;

// Sum start of list in four independent sub-sums
for (i = 0; i < n-3; i += 4)
    for (int j = 0; j < 4; ++j)
        ss[j] += x[i+j];

// Combine sub-sums, handle trailing elements
float s = (ss[0] + ss[1]) + (ss[2] + ss[3]);
for (; i < n; ++i)
    s += x[i];
```

Pointer Aliasing

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`?

Pointer Aliasing

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

- C `restrict` promise: no overlaps in access
- Many C++ compilers have `__restrict__`
- Fortran forbids aliasing – part of why naive Fortran speed often beats naive C speed!

“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;
}
```

Floating Point

Several possible optimizations:

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

But these change semantics! Needs a human.

Optimization Flags

`-O0123`: no optimization – aggressive optimization

- `-O2` is usually the default
- `-O3` is useful, but might break FP codes (for example)

Optimization Flags

Architectural targets

- “Native” mode targets current architecture
- Not always the right choice (e.g. head/compute)

Optimization Flags

Specialized flags

- Turn on/off specific optimization features
- Often the basic `-Ox` has reasonable defaults

Auto-Vectorizations Reports

- Good compilers try to vectorize for you
    - Vendors are pretty good at this
    - GCC / CLang are OK, not as strong
- Can get reports about what prevents vectorization
    - Not necessarily by default!
    - Helps a lot for tuning

Profile-Guided Optimization

Basic workload

- Compile code with optimizations
- Run in a profiler
- Compile again, provide profiler results

Helps with branch optimization.

Data Layout Matters

“Speed-of-Light”

For compulsory misses:

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

Possible optimizations:

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

Reality is more complicated…

When and How to Allocate

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

When and How to Allocate

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

Two standard formats:

- Column major (Fortran): Store columns consecutively
- Row major (C/C++?): Store rows consecutively

Ideally, traverse with unit stride! Layout affects choice.

Can use more sophisticated multi-dim array layouts...

Blocking / Tiling

Classic example: matrix multiply

- Load $b \times b$ block of A
- Load $b \times b$ block of B
- Compute product of blocks
- Accumulate into $b \times b$ block of C

Have $O(b^3)$ work for $O(b^2)$ memory references!

Alignment and Vectorization

- Vector load/stores faster if *aligned* (e.g. start at memory addresses that are multiples of 64 or 256)
- Can ask for aligned blocks of memory from allocator
- Then want aligned offsets into aligned blocks
- Have to help compiler recognize aligned pointers!

Cache Conflicts

Issue: What if strided access causes conflict misses?

- Example: Walk across row of col-major matrix
- Example: Parallel arrays of large-power-of-2 size

Not the most common problem, but one to watch for

Structure Layouts

- Want b-byte types on b-byte memory boundaries
- Compiler may pad structures to enforce this
- Arrange structure fields in decreasing size order

SOA vs AOS

```c
// Structure of arrays (parallel arrays)
typedef struct {
    double* x;
    double* y;
} soa_points_t;

// Array of structs
typedef struct {
    double x;
    double y;
} point_t;
typedef point_t* soa_points_t;
```

SOA vs AOS

SoA: Structure of Arrays

- Friendly to vectorization
- Poor locality to access all of one item
- Awkward for lots of libraries and programs

SOA vs AOS

AoS: Array of Structs

- Naturally supported default
- Not very SIMD-friendly

Can use C++ `zip_view` to iterate over SOA like AOS.

Copy Optimizations

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!

For the Control Freak

Can get (some) programmer control over

- Pre-fetching
- Uncached memory stores

But usually best left to compiler / HW.

Summary

Strategy

- Think some about performance *before* writing
- After coding, time to identify what needs tuning
- Tune data layouts and access patterns together
- Work with compiler on low-level optimizations