CS 5220: Optimization basics

David Bindel

2017-08-31

Reminder: Modern processors

- Modern CPUs are
  - Wide: start / retire multiple instructions per cycle
  - Pipelined: overlap instruction executions
  - Out-of-order: dynamically schedule instructions
- Lots of opportunities for instruction-level parallelism (ILP)
- Complicated! Want the compiler to handle the details
- Implication: we should give the compiler
  - Good instruction mixes
  - Independent operations
  - Vectorizable operations

Reminder: Memory systems

- Memory access are expensive!
- Flop time $\ll$ bandwidth^{-1} $\ll$ latency
- Caches provide intermediate cost/capacity points
- Cache benefits from
  - Spatial locality (regular local access)
  - Temporal locality (small working sets)

Goal: (Trans)portable performance

- Attention to detail has orders-of-magnitude impact
- Different systems = different micro-architectures, caches
- Want (trans)portable performance across HW
- Need *principles* for high-perf code along with tricks

Basic principles

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

Think before you write

Premature optimization

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

– Don Knuth

Premature optimization

Wrong reading: “Performance doesn’t matter”

> *We should forget about small efficiencies, say about 97% of the time: premature* **optimization is the root of all evil**.
>
> *– Don Knuth*

Premature optimization

What he actually said (with my emphasis)

> *We should forget about* **small** *efficiencies, say* **about 97%** *of the time:* **premature** *optimization is the root of all evil.*
>
> *– Don Knuth*

- Don't forget the big efficiencies!
- Don't forget the 3%!
- Your code is not premature forever!

Don't sweat the small stuff

- Speed-up from tuning ϵ of code $< (1-\epsilon)^{-1} \approx 1+\epsilon$
- OK to write high-level stuff in Matlab or Python
- OK if configuration file reader is un-tuned
- OK if $O(n^2)$ prelude to $O(n^3)$ algorithm is not hyper-tuned?

Lay-of-the-land thinking

```
for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j)
    for (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 the 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?

How big is n?

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

- Meaning: $\exists C, N : \forall n \geq N, T_n \leq Cf(n)$.
- Says *nothing* about constant factors: $O(10n) = O(n)$
- Ignores lower order term: $O(n^3 + 1000n^2) = O(n^3)$
- Behavior at small n may not match behavior at large n!

Beware asymptotic complexity arguments about small-n codes!

Avoid work

```
bool any_negative1(int* x, int n)
{
  bool result = false;
  for (int i = 0; i < n; ++i)
    result = (result || x[i] < 0);
  return result;
}

bool any_negative2(int* x, int n)
{
  for (int i = 0; i < n; ++i)
    if (x[i] < 0)
      return false;
  return true;
}
```

Be cheap

Fast enough, right enough $\implies$
Approximate when you can get away with it.

Do more with less (data)

Want lots of work relative to data loads:

- Keep data compact to fit in cache
- Use short data types for better vectorization
- But be aware of tradeoffs!
  - For integers: may want 64-bit ints sometimes!
  - For floating-point: will discuss in detail 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

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 eliminate
    - Cute coinage: “de-slugging”

Practical timing

Need to worry about:

- System timer resolutions
- 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 it several times with “characteristic” data
- Accumulate the total time spent

Caveats: Effects from repetition, “characteristic” data

Manual instrumentation

- Hard to get *portable* high-resolution wall-clock time!
- Solution: `omp_get_wtime()`
- Requires OpenMP support (still not CLang)

Types of profiling tools

- Sampling vs instrumenting
    - Sampling: Interrupt every t_{profile} cycles
    - Instrumenting: Rewrite code to insert timers
    - Instrument at binary or source level
- Function level or line-by-line
    - Function: Inlining can cause mis-attribution
    - Line-by-line: Usually requires debugging symbols (`-g`)
- Context information?
    - Distinguish full call stack or not?
- Time full run, or just part?

Hardware counters

- Counters track cache misses, instruction counts, etc
- Present on most modern chips
- May require significant permissions to access...

Automated analysis tools

- Examples: MAQAO and IACA
- Symbolic execution of *model* of a code segment
- Usually only practical for short segments
- But 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 a list

Kernel trade-offs

- Critical to get *properly tuned* kernels
    - Kernel *interface* is consistent across HW types
    - Kernel *implementation* varies according to arch details
- General kernels *may* leave performance on the table
    - Ex: General matrix-matrix multiply for structured matrices
- Overheads may be an issue for small n cases
    - Ex: Usefulness of batched BLAS extensions
- But: Ideally, someone else writes the kernel!
    - Or it may be automatically tuned

Help your tools help you

What can your compiler do for you?

In decreasing order of effectiveness:

- Local optimization
  - Especially 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 optimizations

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

Let's look at a few...

Ex: Long dependency chains

Sometimes these can be decoupled (e.g. reduction loops)

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

Apparent linear dependency chain. Compilers might handle this, but let's try ourselves…

Ex: Long dependency chains

Key: Break up chains to expose parallel opportunities

```
// Version 1
float s[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)
  s[j] += x[i+j];

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

Ex: 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`?

Ex: Pointer aliasing

C99: Use `restrict` keyword

```
void add_vecs(int n, double* restrict result,
              double* restrict a, double* restrict b);
```

Implicit promise: these point to different things in memory.

Fortran forbids aliasing — part of why naive Fortran speed beats naive C speed!

Ex: “Black box” function calls

Compiler must assume arbitrary wackiness from “black box” function calls

```
double 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;
}
```

Ex: Floating point issues

Several possible optimizations available:

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

Problem: This changes semantics!

- A daring compiler will pretend floats are reals and hope
- This will break some of my codes!
- Human intervention is indicated

Optimization flags

- `-O[0123]` (no optimization – aggressive optimization)
  - `-O2` is usually the default
  - `-O3` is useful, but might break FP codes (for example)
- Architecture targets
  - Usually a “native” mode targets current architecture
  - Not always the right choice (e.g. consider Totient head/compute)
- Specialized optimization flags
  - Turn on/off specific optimization features
  - Often the basic `-Ox` has reasonable defaults

Auto-vectorization and compiler reports

- Good compilers try to vectorize for you
    - Intel is 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 workflow:

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

Helps compiler optimize branches based on observations.

Data layout matters

“Speed-of-light” analysis

For compulsory misses to load cache:

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

Possible optimizations:

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

Reality is generally more complicated...

When and how to allocate

Why is this an $O(n^2)$ loop?

```
x = [];
for i = 1:n
  x(i) = i;
end
```

When and how to allocate

- Access is not the only cost!
    - Allocation / de-allocation also costs something
    - So does garbage collection (where supported)
    - Beware hidden allocation costs (e.g. on resize)
    - Often bites naive library users
- Two thoughts to consider
    - Pre-allocation (avoid repeated alloc/free)
    - Lazy allocation (if alloc will often not be needed)

Storage layout

Desiderata:

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

Multi-dimensional arrays

Two standard formats:

- Col-major (Fortran): Each column stored consecutively
- Row-major (C/C++): Each row stored consecutively

Ideally, traverse arrays with unit stride! Layout affects choice.

More sophisticated multi-dim array layouts may be useful...

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!

Data alignment and vectorization

- Vector load/stores faster if *aligned* (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!

Data alignment and cache contention

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 type to start on *b*-byte memory boundary.
- Compiler may pad structures to enforce this.
- Advice: arrange structure fields in decreasing size order.

SoA vs AoS

```c
// Struct of Arrays (parallel arrays)
typedef struct {
  double* x;
  double* y;
} aos_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
- AoS: Array of Structs
  - Naturally supported default
  - Not very SIMD-friendly
- Possible to combine the two...

Copy optimizations

Copy between formats to accelerate computations, 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.

Matrix multiplication

- This was a lot of stuff in a short time!
- Best way to digest it is try some things out
- First project: tune matrix-matrix multiply
- Due Sep 12 (about two weeks)
    - Gives enough time to play with some ideas
    - Not enough time for obsessive tuning to ruin lives
- We encourage partners – try to cross disciplines!

Recommended strategy

- Start with a small “kernel” multiply
    - Maybe odd sizes, strange layouts – just go fast!
    - Intel compiler may do fine with simple-looking code
    - Deserves its own timing rig
- Use blocking to build up larger multiplies
- Will have to do something reasonable with edge blocks…

References

- My serial tuning notes.
- Ulrich Drepper, *What Every Programmer Should Know About Memory*
- Intel Optimization Manual
- Hager and Wellein, *Intro to HPC for Scientists and Engineers*
- Goedecker and Hoisie, *Performance Optimization of Numerically Intensive Codes*
- Agner Fog's Software Optimization Manuals