Tuning on a single core

1 From models to practice

In lecture 2, we discussed features such as instruction-level parallelism and cache hierarchies that we need to understand in order to have a reasonable mental model of performance of a single-processor code. Our goal in these notes is to turn our new-found understanding of these details into practical strategies for making single-core codes run fast, using the matrix multiplication project as a running example.

Note that these notes are an extended version of the start of lecture 3, and they contain information that is not in the lecture 3 slides.

2 A cautionary notes

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
– C. A. R. Hoare (quoted by Donald Knuth)

Performance tuning is like catnip for programmers. Most of us like the idea of writing blazingly fast code and reveling in our own cleverness[1]. But while computer cycles have become astonishingly cheap in terms of both money and time, the same is not true for programmer cycles. It is easy to fall into the trap of lovingly optimizing performance details at the cost of increased development time (including time for testing and debugging), higher maintenance overhead, and decreased robustness. A pragmatic approach is usually best. Worry about things that will have a big impact on performance, like the choice of appropriate algorithms, data structures, and memory layouts. Use fast libraries, and turn on the appropriate compiler optimizations. Make sure that the code works correctly, and set up a test framework so that you know immediately if you do something to break the code. Then, if it makes sense to spend more time thinking about performance, use a profiler to find bottlenecks where the code is spending most of its time, and concentrate your efforts there.

[1]At least, *I* like reveling in my own cleverness.

```
const char* dgemm_desc = "Basic, three-loop dgemm.";

void square_dgemm(const int M,
                  const double *A, const double *B, double *C)
{
    int i, j, k;
    for (i = 0; i < M; ++i) {
        for (j = 0; j < M; ++j) {
            double cij = C[j*M+i];
            for (k = 0; k < M; ++k)
                cij += A[k*M+i] * B[j*M+k];
            C[j*M+i] = cij;
        }
    }
}
```

Figure 1: Naive matrix-matrix multiplication code in C.

3 Matrix multiplication

The *D*ouble precision *Ge*neral *M*atrix-*M*atrix multiplication routine (DGEMM) is one of the Basic Linear Algebra Subroutines (BLAS). The BLAS interface is a standard library interface for linear algebra building blocks like dot products, matrix-vector products, and matrix-matrix multiplication. Libraries like LAPACK get good portable performance[2] by organizing the work they do so that most of the computation is done in calls to BLAS libraries; that way, if someone has designed a fast implementation of the BLAS for a particular machine, LAPACK can run on that machine. The matrix-matrix multiplication routine is one of the most important of the BLAS routines, because it is a useful building block that can be made to run fast on modern machines with deep cache hierarchies.

The BLAS DGEMM routine can actually do several different things, so let's consider a slightly simplified routine that computes $C = C + AB$ where A, B, and C are square matrices of dimension M (Figure 1). In terms of

[2]I may have to change this line in a couple years, since the LAPACK team is looking at new organizations for multicore architectures that go beyond the performance of the old BLAS-based blocked codes.

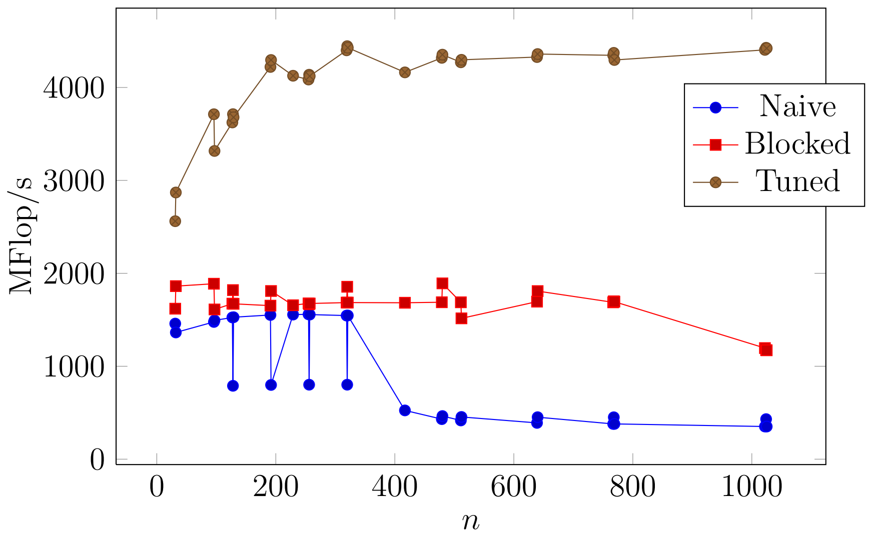

Figure 2: Performance of naive, blocked, and tuned codes on one core of an Intel Core 2 laptop.

matrix elements, we want to compute

$$c_{ij}^{\text{new}} = c_{ij} + \sum_{k=1}^{M} a_{ik} b_{kj}.$$

Note that in the C code, we assume the matrices are in *column-major* order: the first M elements of the `C` array represent the first column of the matrix C, then the next M elements represent the second column, and so forth. That is, the matrix element c_{ij} (row i, column j) appears at the array location `C[i+j*M]`, and the elements of A and B are accessed similarly. This is different from the *row-major* order that C uses for its native two-dimensional arrays. But the native two-dimensional array facility in C is quite limited, and we generally will ignore it.

The arithmetic cost of the `square_dgemm` routine is M^2 multiplies and M^2 add, for a total of $2M^3$ floating point operations. For any matrix size M, we can take the number of flops and divide by the time required to execute those flops in order to get the performance of our matrix-multiplication in flop/s. Usually, we scale by a million or a billion, and report Mflop/s or Gslop/s instead of flop/s. The theoretical peak flop rate on my laptop is 5

Gflop/s per core, or 10 Gflop/s if I use both cores. The actual flop rates for three versions of `square_dgemm` on this machine are shown in Figure 2. For large matrices, the performance of the naive code from Figure 1 is only 330 MFlop/s, or less than 7% of the performance on a single core. In contrast, with a little tuning[3], we can reach nearly 90% of the peak performance on this system. How do we do it?

4 DGEMM tuning ideas

4.1 Blocking

In class, we discussed the idea of blocking. For example, consider the matrix-matrix product

$$\begin{bmatrix} 650 & 762 & 874 & 986 \\ 740 & 868 & 996 & 1124 \\ 830 & 974 & 1118 & 1262 \\ 920 & 1080 & 1240 & 1400 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix} \begin{bmatrix} 21 & 25 & 29 & 33 \\ 22 & 26 & 30 & 34 \\ 23 & 27 & 31 & 35 \\ 24 & 28 & 32 & 36 \end{bmatrix}.$$

We can think of the $(1, 1)$ entry of the product as an inner product of the first row and the first column of the two matrices in the product:

$$650 = 1 \cdot 21 + 5 \cdot 22 + 9 \cdot 23 + 13 \cdot 24.$$

Similarly, we can think of partitioning all three matrices into 2-by-2 blocks, and writing the first block of the product in terms of the first block row and block column of the two terms in the product:

$$\begin{bmatrix} 650 & 762 \\ 740 & 868 \end{bmatrix} = \begin{bmatrix} 1 & 5 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 21 & 25 \\ 22 & 26 \end{bmatrix} + \begin{bmatrix} 9 & 13 \\ 10 & 14 \end{bmatrix} \begin{bmatrix} 23 & 27 \\ 24 & 28 \end{bmatrix}.$$

The advantage of thinking about matrix multiplication in this way is that even if our original matrices don't fit into cache, the little blocks will; and if we break the matrices into b-by-b blocks, then each block multiplication involves $2b^3$ flops and $2b^2$ data. Note that the idea of blocking can be applied recursively: a large matrix might be partitioned into big blocks that fit in L2 cache, which in turn are partitioned into smaller blocks that fit in L1 cache, which in turn are partitioned into tiny blocks that fit into the registers.

[3]For an embarrassingly large value of "little."

4.2 Loop orders

In the naive code, we loop over i, then j, then k. We could equally well loop over i, k, j, or j, i, k, or something else. In the i, j, k order, we have to go across a row of A in the inner loop, which is a non-unit-stride access. The reference Fortran DGEMM does better with the j, k, i ordering. You may want to think about other alternatives. Note that if you use blocking, you can potentially use different loop orders over blocks than within blocks.

4.3 Copy optimization

Even if we block and choose careful loop orderings, we might run into problems with conflict misses (associated with cache associativity) when the matrix sizes are evenly divisible by a large power of two. One way around this problem is to explicitly copy blocks into a contiguous block of local storage before multiplying them. Often, the extra $O(b^2)$ cost of copying the data into a temporary location is more than compensated by the improved performance of the subsequent multiplication.

Apart from cache associativity issues, there is another reason why it might make sense to copy blocks into separate storage before multiplying them. The short vector instructions (SSE instructions) care about data alignment, and we usually can't enforce that alignment if we're just handed general arrays by the user.

Another side benefit of copy optimization is that you can use it to gracefully deal with fringe blocks. If you copy "fringe" blocks of your matrices into temporary storage and fill the rest of the temporary storage with zeros, then you can always pretend the fringe blocks are a little bigger than they actually are in order to conveniently fit your lower-level blocking. You may end up doing some pointless multiplications by zero; but if you do a little extra work and improve the regularity of your data access, you may still end up getting better performance in the end.

4.4 Compiler flags and annotations

Modern compilers do some types of optimizations much better than people do. Even a mediocre compiler will usually do a reasonable job at scheduling instructions and figuring out which registers should store which variables.

But it's worthwhile playing with the flags that control the compiler optimizations. Some flags to try with GCC are

- `-O3`: Aggressive optimization
- `-march=core2`: Tuning for a specific architecture (my laptop is a Core 2; you should figure out the appropriate architecture flag for the cluster yourself).
- `-ftree-vectorize`: Automatically use the SSE units intelligently (the Intel compiler is still much better at this)
- `-funroll-loops`: Unroll loops (basic loop unrolling is automatic with `-O3`).
- `-ffast-math`: Allow risky floating point optimizations.

In addition to fiddling with compiler flags, you can help the compiler find opportunities for optimization by adding a few annotations to the code. For example:

- The `aligned (16)` attribute in GCC makes sure that some things are aligned in memory on 16-byte boundaries. That's useful with the SSE instructions, some of which only work with aligned data.
- The `restrict` modifier for pointers in C is a promise to the compiler that there will be no *aliasing* between different points. For example, consider the following function:

  ```
  void add_vecs(int n, double* restrict x, double* restrict y)
  {
      int i;
      for (i = 0; i < n; ++i) {
        x[i] += y[i];
      }
  }
  ```

  Without the `restrict` keyword, the compiler has to worry about the possibility that the data array pointed to by `x` overlaps the data array pointed to by `y`.

Note that this level of tuning is usually only worthwhile *after* you have worried about issues of memory layout and figured out a good set of optimization flags. A profiler can help guide you to places where such detailed tuning might help.

4.5 Vector instructions

Unfortunately, GCC is still not very good at generating code that takes full advantage of the SSE instructions. I hope in a couple years it will improve to the point where I never again have to write these instructions directly (the Intel compiler is already at this point). For the moment, though, it may make sense to use the SSE intrinsics defined in `xmmintrin.h`.

There is a reasonably good overview of SSE programming on the Apple developer web site:

`http://developer.apple.com/hardwaredrivers/ve/sse.html`

Though the topic of this page is nominally about SSE on OS X, most of the information is not specific to Macs, and applies equally well to Intel systems running other operating systems. A Google search for the keywords “SSE” and “xmmintrin” will bring you to many other resources for SSE programming.

4.6 Auto-tuning

Most of the tuning ideas described in this section involve some choice. What loop orders should we use? What blocking factors make sense? Should we do a copy optimization, and if so, at which levels? Which compiler flags should we try? When the answers to these questions are not obvious, it may make sense to do an experiment: try several possible parameter settings, and see what works best. But there’s no need for you to be directly involved in all of these experiments – you can just as well write a script to have the computer do it for you! For example, this strategy is used very successfully in the ATLAS project (Automatically Tuned Linear Algebra Software)