Week 2: Monday, Aug 31

Logistics

1. Lecture 1 notes are posted. In general, lecture notes will be posted under the "lectures" page.
2. Homework 1 has been posted. The due date is Monday, September 21.
3. My official office hours are Tuesday and Wednesday, 2-3 pm. You can also drop by my office, assuming the door is open.

Blocked matrix multiply

Supposing we have very limited memory (e.g. only registers) available, let's consider where loads and stores might occur in the inner product version of square matrix-matrix multiply:

```
% To compute C = C + A*B
for i = 1:n
  for j = 1:n
    % load C(i,j)
    for k = 1:n
      % load A(i,j) and B(k,j)
      C(i,j) = C(i,j) + A(i,k)*B(k,j);
    end
    % store C(i,j)
  end
end
```

We can count that there are $2n^2 + 2n^3$ loads and stores in total – that is, we do roughly the same number of memory transfer operations as we do arithmetic operations. If memory transfers are much slower than arithmetic, this is trouble.

Now suppose that $n = Nb$, where b is a block size and N is the number of blocks. Suppose also that b is small enough that we can fit one block of A, B, and C in memory. Then we may write

```
for i = 1:N
  for j = 1:N
    % Make index ranges associated with blocks
    I = ((i-1)*N+1):i*N;
    J = ((j-1)*N+1):j*N;

    % load block C(I,J) into fast memory

    for k = 1:N
      % load blocks A(I,K) and B(K,J) into fast memory
      K = ((k-1)*N+1):k*N;
      C(I,J) = C(I,J) + A(I,K)*B(K,J);
    end

    % store block C(I,J)
  end
end
```

Following the previous analysis, we find there are $2N^2 + 2N^3$ loads and stores of blocks with this algorithm; since each block has size $b \times b$, this gives $(2N^2 + 2N^3)b^2 = 2n^2 + 2n^3/b$ element transfers into fast memory. That is, the number of transfers into fast memory is about $1/b$ times the total number of element transfers.

We could consider using different block shapes, but it turns out that roughly square blocks are roughly optimal. It also turns out that this is only the start of a very complicated story. The last time I taught a high-performance computing class, we worked on a tuned matrix-matrix multiplication routine as one of the first examples. My version roughly did the following:

1. Split the matrices into 56×56 blocks.

2. For each product of 56×56 matrices, copy the submatrices into a contiguous block of aligned memory.

3. Further subdivide the 56×56 submatrices into 8×8 submatrices, which are multiplied using a simple fixed-size basic matrix multiply with a few annotations so that the compiler can do lots of optimizations. If the overall matrix size is not a multiple of 8, pad it with zeros.

4. Add the product submatrix into the appropriate part of C.

By using the Intel compiler and choosing the optimizer flags with some care, I got good performance in the tiny $8 \times 8 \times 8$ matrix multiply kernel that did all the computation – it took advantage of vector instructions, unrolled loops to get a good instruction mix, etc. The blocking then gave me reasonably effective cache re-use. It took me a couple days to get this to my liking, and my code was still not as fast as the matrix multiplication routine that I got from a well-tuned package.

Matrix-vector multiply revisited

Recall from last time that I asked you to look at three organizations of matrix-vector multiply in MATLAB: multiplication with the built-in routine, with a column-oriented algorithm, and with a row-oriented algorithm. I coded up the timing myself:

```
for n= 4095:4097

  fprintf('-- %d --\n', n);
  A= rand(n);
  b= rand(n,1);

  tic;
  for trials = 1:20
    c1 = A*b;
  end
  fprintf('Standard multiply: %g ms\n', 50*toc);

  tic;
  for trials = 1:20
    c2 = zeros(n,1);
    for j = 1:n
      c2 = c2 + A(:,j)*b(j);
    end
  end
  fprintf('Column-oriented:   %g ms\n', 50*toc);
  tic;
  for trails = 1:20
    c3 = zeros(n,1);
    for j = 1:n
      c3(j) = A(j,:)*b;
    end
  end
  fprintf('Row-oriented:      %g ms\n', 50*toc);

end
```

This is what I saw:

```
-- 4095 --
Standard multiply: 21.2793 ms
Column-oriented:   96.799 ms
Row-oriented:      185.404 ms
-- 4096 --
Standard multiply: 22.7508 ms
Column-oriented:   97.1834 ms
Row-oriented:      379.368 ms
-- 4097 --
Standard multiply: 22.4621 ms
Column-oriented:   97.5955 ms
Row-oriented:      185.886 ms
```

Notice that the row-oriented algorithm is about twice as slow as the column-oriented algorithm for $n = 4095$, and that *it is twice as slow for* $n = 4096$ *as for* $n = 4095$! The reason again is the cache architecture. On my machine, the cache is 8-way set associative, so that each memory location can be mapped to only one of eight possible lines in the cache, and each row of A uses exactly the same set of eight lines when $n = 4096$. Therefore, we effectively get *no* cache re-use with $n = 4096$. With $n = 4095$ we don't have this conflict (or at least we suffer from it far less); and since each cache line holds two doubles, we effectively load two rows of A at a time.

If by now you're feeling too lazy to want to untangle the details of how the cache architecture will affect matrix-matrix and matrix-vector multiplies, let alone anything more complicated, don't worry. That's basically the point.

The lazy man's lesson

What's the moral of these stories? It is simply that there is a lot going on under the hood, much of which has to do with the organization of memory; and that we can get the best "bang for our buck" by taking the time to formulate our algorithms in block terms, so that we can spend most of our computation inside someone else's well-tuned matrix multiply routine (or something similar). There are several implementations of the Basic Linear Algebra Subroutines (BLAS), including some implementations provided by hardware vendors and some automatically generated by tools like ATLAS. The best BLAS library varies from platform to platform, but by using a good BLAS library and writing routines that spend a lot of time in *level 3* BLAS operations (operations that perform $O(n^3)$ computation on $O(n^2)$ data and can thus potentially get good cache re-use), we can hope to build linear algebra codes that get good performance across many platforms.

This is also a good reason to use MATLAB: it uses pretty good BLAS libraries, and so you can often get surprisingly good performance from it for the types of linear algebraic computations we will pursue.

Error analysis for matrix-vector multiply

We now switch gears for a little while to consider the sensitivity of matrix-vector multiplication. Roundoff and imprecise input values act perturb such problems, and so we need to study *perturbation theory* in order to understand how error is introduced and propagated. Since it may be cumbersome to write separate error bounds for every component, it will be very convenient to express our analysis in terms of *norms* and to develop the concept of *conditioning*. It will also be convenient to introduce the *singular value decomposition* as a way of understanding our analysis in the Euclidean norm.

We begin with the idea of norms. Recall that a norm is a scalar-valued function from a vector space into the real numbers with the following properties:

1. *Positive-definiteness*: For any vector x, $\|x\| \geq 0$; and $\|x\| = 0$ iff $x = 0$
2. *Triangle inequality*: For any vectors x and y, $\|x + y\| \leq \|x\| + \|y\|$
3. *Homogeneity*: For any scalar α and vector x, $\|\alpha x\| = |\alpha|\|x\|$

We will pay particular attention to three norms on $\mathbb{R}^n$ and $\mathbb{C}^n$:

$$\begin{aligned}
\|v\|_1 &= \sum_i |v_i| \\
\|v\|_\infty &= \max_i |v_i| \\
\|v\|_2 &= \sqrt{\sum_i |v_i|^2}
\end{aligned}$$

If A maps between two normed vector spaces $\mathcal{V}$ and $\mathcal{W}$, the *induced norm* on A is

$$\|A\|_{\mathcal{V},\mathcal{W}} = \sup_{v \neq 0} \frac{\|Av\|_{\mathcal{W}}}{\|v\|_{\mathcal{V}}}.$$