Homework 4

Josh Bradt

February 15, 2016

1 Performance estimates

1.1 All data remains in cache

In this case, all elements of each matrix need to be read once, and the output elements are all written once. There are 2 floating point operations in the inner loop. Thus,

$$\text{time} \approx 2cn^3 + rn^2 + rn^2 + wn^2 = 2cn^3 + (2r + w)n^2. \tag{1}$$

Using the provided values of $c = 0.2\,\text{ns/FLOP}$ and $r = w = 1\,\text{ns/double}$ for a matrix with $n = 10^3$, this equals

$$\text{time} \approx 2(0.2\,\text{ns})(10^3)^3 + 3(1\,\text{ns})(10^3)^2 = 0.4\,\text{s} + 3\,\text{ms} = 0.4\,\text{s}.$$

1.2 Matrix B does not fit in cache

In this case, the n^2 elements of matrix B must be re-read every time they are used. This moves their read times to the n^3 term:

$$\text{time} \approx 2cn^3 + rn^2 + rn^3 + wn^2 = (2c + r)n^3 + (r + w)n^2. \tag{2}$$

Plugging in the same values as before gives

$$\text{time} \approx (2(0.2\,\text{ns}) + (1\,\text{ns}))(10^3)^3 + 2(1\,\text{ns})(10^3)^2 = 1.4\,\text{s} + 2\,\text{ms} = 1.4\,\text{s}.$$

2 Performance measurements

For the comparison, I wrote a C function for blocked matrix multiply which is shown in Listing 1. I compared the time taken by this function to a C translation of the Fortran function written in the assignment. In both cases, the data was stored in row-major order.

I compiled the code with version 15.0.0 of the Intel C compiler on the `dev-intel14` node of the HPCC and ran it on one core of one node of the `intel14` cluster. I tried optimization flags `-O0`, `-O1`, `-O2`, and `-O3`, and ran the code for several dimensions up to $n = 2000$. This upper bound should be larger

```
#define ind(i,j) (i)*matSize+j
#define MIN(a,b) ((a) < (b) ? a : b)

void blockedMatrixMultiply(const double* matA, const double* matB,
                           double* out, const size_t matSize)
{
  const size_t bsize = 16;

  for (size_t i = 0; i < matSize; i += bsize) {
    for (size_t j = 0; j < matSize; j += bsize) {
      double sum = 0;
      for (size_t k = 0; k < matSize; k++) {
        for (size_t ii=i; ii < MIN(i+bsize,matSize); ii++) {
          for (size_t jj=j; jj < MIN(j+bsize,matSize); jj++) {
            sum += matA[ind(ii,k)] * matB[ind(k,jj)];
          }
        }
      }
      out[ind(i,j)] = sum;
    }
  }
}
```

Listing 1: Blocked matrix-matrix multiply function.

than the 25 MB L3 cache[1] of the Intel Xeon E5 2670 processor in the node since $\sqrt{(25 \times 10^6\,\mathrm{B})/(8\,\mathrm{B})} = 1767$.

The results of these runs are shown in Table 1 and plotted in Figure 1 and Figure 2.

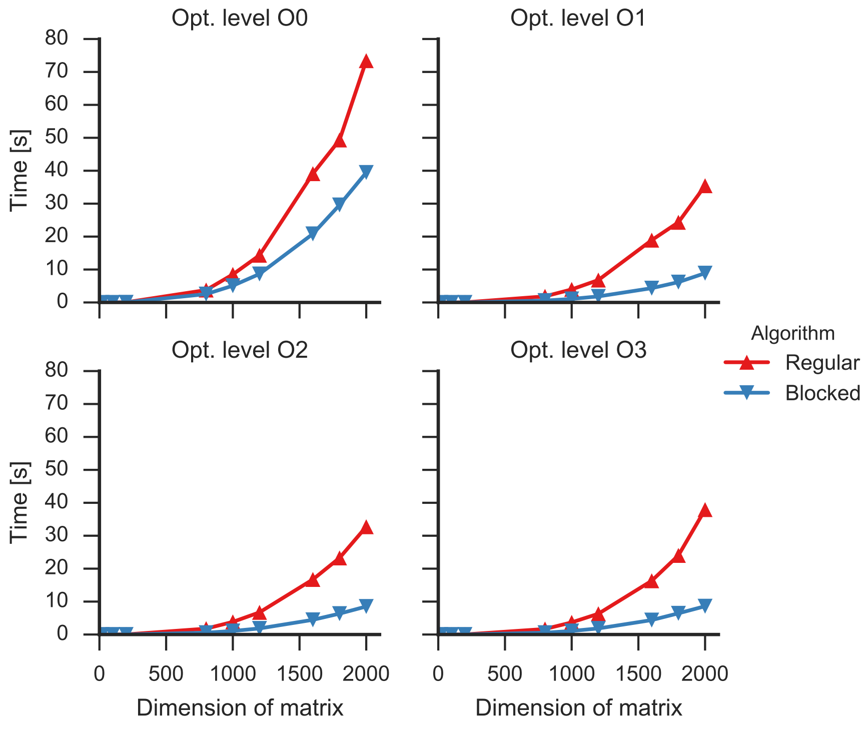
For large n, the blocked code performed significantly better, as expected. I had expected there to be an abrupt drop in performance around the point where the matrices would no longer fit in cache, but that is either not present or is hidden by my choices of n. This might be because there is a lot more computation happening in this example than there was when we were just copying arrays in different ways.

I was slightly surprised that increasing the optimization level beyond `-O1` didn't improve performance much, especially since I used a "fancy" compiler (`icc`) just for this reason. Most likely, though, my efforts to hide my intentions from the optimizer worked and prevented it from replacing my code entirely with something better (which it initially did before I inserted a call to a dummy function in my plain matrix-matrix multiply function).

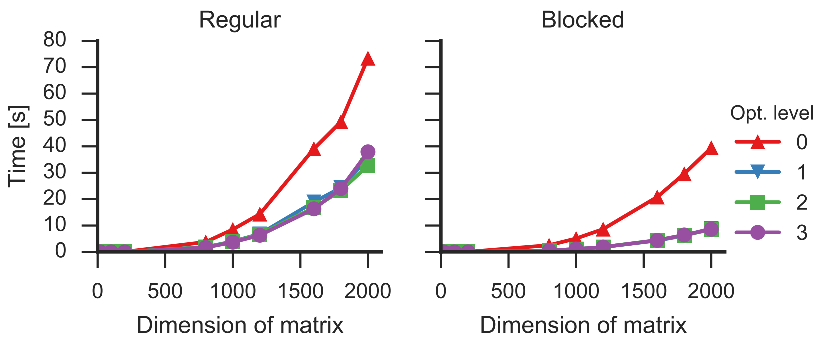
[1] It's not clear how this is shared amongst the 10 cores of the processor, but if the matrix is larger than the whole cache, then clearly it's larger than part of the cache.

Opt.	N	Blocked Time	Blocked Rate	Regular Time	Regular Rate
0	20	3.909×10^{-5}	0.409	5.387×10^{-5}	0.297
	100	4.913×10^{-3}	0.407	6.162×10^{-3}	0.325
	200	3.955×10^{-2}	0.405	5.062×10^{-2}	0.316
	800	2.540	0.403	3.769	0.272
	1000	5.084	0.393	8.507	0.235
	1200	8.647	0.400	1.433×10^{1}	0.241
	1600	2.082×10^{1}	0.393	3.911×10^{1}	0.209
	1800	2.960×10^{1}	0.394	4.934×10^{1}	0.236
	2000	3.947×10^{1}	0.405	7.342×10^{1}	0.218
1	20	8.508×10^{-6}	1.881	2.226×10^{-5}	0.719
	100	1.053×10^{-3}	1.899	2.915×10^{-3}	0.686
	200	8.321×10^{-3}	1.923	2.410×10^{-2}	0.664
	800	5.671×10^{-1}	1.806	1.848	0.554
	1000	1.068	1.872	3.960	0.505
	1200	1.833	1.886	6.803	0.508
	1600	4.334	1.890	1.892×10^{1}	0.433
	1800	6.172	1.890	2.433×10^{1}	0.479
	2000	8.888	1.800	3.545×10^{1}	0.451
2	20	8.307×10^{-6}	1.926	2.279×10^{-5}	0.702
	100	1.041×10^{-3}	1.922	2.937×10^{-3}	0.681
	200	8.491×10^{-3}	1.884	2.506×10^{-2}	0.639
	800	5.493×10^{-1}	1.864	1.788	0.573
	1000	1.080	1.853	3.839	0.521
	1200	1.848	1.870	6.729	0.514
	1600	4.462	1.836	1.673×10^{1}	0.490
	1800	6.344	1.839	2.326×10^{1}	0.501
	2000	8.510	1.880	3.273×10^{1}	0.489
3	20	8.567×10^{-6}	1.868	2.488×10^{-5}	0.643
	100	1.063×10^{-3}	1.881	2.924×10^{-3}	0.684
	200	8.564×10^{-3}	1.868	2.471×10^{-2}	0.648
	800	5.482×10^{-1}	1.868	1.675	0.611
	1000	1.079	1.854	3.686	0.543
	1200	1.849	1.869	6.310	0.548
	1600	4.387	1.867	1.632×10^{1}	0.502
	1800	6.438	1.812	2.403×10^{1}	0.485
	2000	8.591	1.862	3.794×10^{1}	0.422

Table 1: Timing and rate measurements for the two codes under a variety of conditions. All times are given in seconds, and all rates are given in GFLOPS.

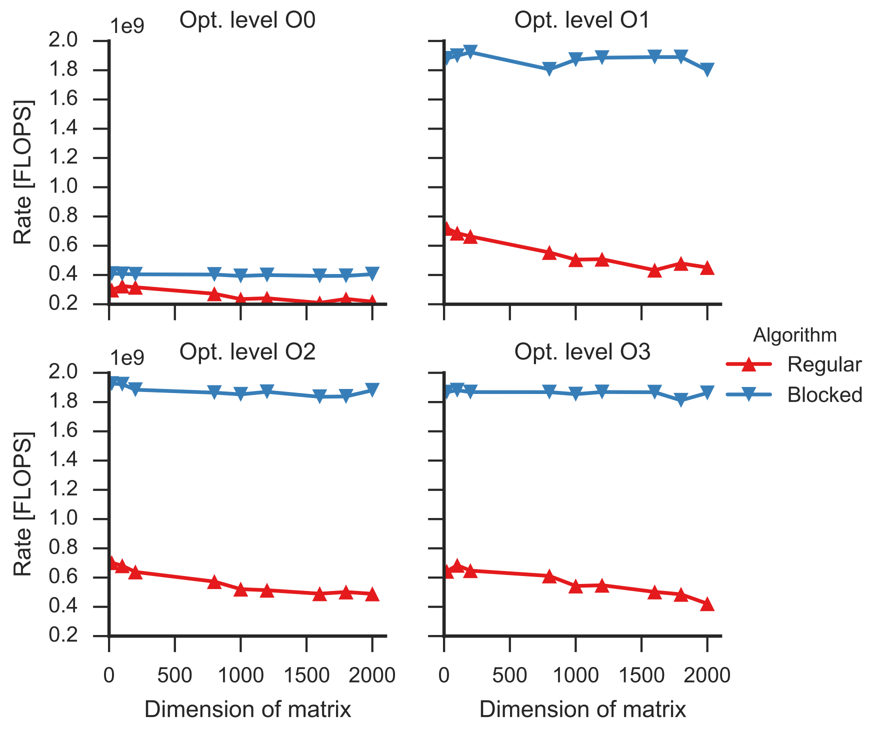


(a) Compared across algorithms

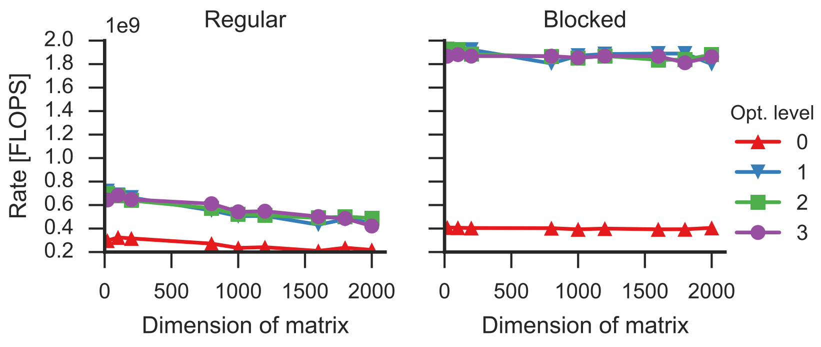


(b) Compared across optimization levels

Figure 1: The timing results, shown in two ways.



(a) Compared across algorithms



(b) Compared across optimization levels

Figure 2: The rate results, shown in two ways.