AMATH 583 FINAL
Ben Francis

QUESTION 1

Given that forming the covariance matrix with 500 images takes approximately 10 seconds, how long (approximately) would forming the covariance matrix take if we used all 262,781 images in the data file?

Answer

The time complexity of the covariance matrix is the vector sized squared times the number of vectors. That is, it is linear with the number of images/vectors. If 500 images take 10 seconds, then 1 image takes 0.02 seconds. Then 262,781 would take 262,781 * 0.02 = 5255.62 seconds, or approximately 87 minutes, or over an hour.

QUESTION 2

What optimizations did you apply to improve eigenfaces_opt? Explain why these optimizations would result in better performance.

Answer

Optimization 1:
(In "outer')
First, we switch the order of the loops. Presuming A to be row-major ordering, we want to go through all of the column elements in the first row, then all of the columns in the second row, etc., not the other way around (all of the row elements in a column). These accesses are done in *A(i,j)*.

Optimization 2:
(In "outer")
Second, we hoist out *x(i)*, so we access it once per row of *A*, not once per element in *A*.

For good measure, though I wouldn't go so far as to call it an "optimization", I got the size of *z* and put it in a new variable *z_size*, so that I wouldn't have to call *z.size()* each time. I am not sure the mechanics of the *size* call; I am sure it is *O(1)*, but the referencing of the vector each time to call its method, versus the storing of a local variable that could be kept in a register, felt like a good idea.

With these improvements, I was able to get 2,285 ms, in the range of what the professor reported.

QUESTION 3

How did you parallelize your code? On 5000 faces, how much speedup were you able to get on 2, 4, 10, and 20 cores (cpus-per-task)? Does the speedup improve if you load 15000 faces rather than 5000 faces?

Answer
I used *#pragma omp parallel for*. We are here work sharing via loop parallelization, where we are splitting work by rows. Then each processor is effectively computing a dot product with one row of *A* and the vector *v*. As such, the pragma is placed around the outside for loop, such that it parallelizes by row.

I will now inventory its rates on different values of cores (evaluating the compute covariance time), on a problem size of 5000, to evaluate the speedup.

CORES	MS
1	22,436
2	12,109
4	7,622
10	4,435
20	2,018

From 1 to 2 this is a speedup of about 1.85.
From 1 to 4 this is a speedup of about 2.94.
From 1 to 4 this is a speedup of about 5.05.
From 1 to 4 this is a speedup of about 11.18.

The speedup for 15,000 is:

CORES	MS
1	69,504
2	36,458
4	22,909
10	13,594
20	6,679

From 1 to 2 this is a speedup of about 1.91.
From 1 to 4 this is a speedup of about 3.03.
From 1 to 4 this is a speedup of about 5.11.
From 1 to 4 this is a speedup of about 10.4.

The returns are about the same.

QUESTION 4
Explain your blocking approach. On 5000 faces and 10 (or 20) cores (say), how much speedup were you able to get over the non-blocked case? How much speedup were you able to get over the non-blocked case for 50000 face?

ANSWER

First, we realize that the covariance matrix C can be computed in blocks, since each sub-square of C depends only on a certain range of each *z[k]*. As such, we use an outer loop with new indices *ii* and *jj* that step along in blocks. The block sizes have a maximum size of 32, though can be as small as the size of *C* if *C* is less than 32 in width/height.

We find the modulus of the problem size with the block size, which we call *diff*, to find out the size of the fringe. Then we iterate the blocks up until the matrix size *minus diff*, so that we don't get a seg fault error. Now things get a little sneaky; there are three "fringe regions" we must cover. There is a "strip" of fringe on the right and bottom, and then a tiny fringe "square" in the bottom right. We iterate by blocks in one dimension and by the fringe length in the other to capture the strips, and by the fringe length in both dimensions to get the square. This is commented in the code, of which I hope it is clear.

The following reports in units of ms:

# CPUs	**10 CPUs**		**20 CPUs**	
Problem Size	**5,000**	**50,000**	**5,000**	**50,000**
Non-blocked	4,555	43,765	2,405	25,692
Blocked	1,098	9,240	2,033	18,386

As such, the speedups are:
10 CPUs : 5,000 problem size: 4.15
10 CPUs : 50,000 problem size: 4.74
20 CPUs : 5,000 problem size: 1.18
20 CPUs : 50,000 problem size: 1.40

The fact it performed better with problem size 50,000 with 10 CPUs than with 20 CPUs deserves mention, though I'm not sure what quite to make of it. My guess is that these are due to communication costs. It is also worth noting that the non-blocked case had a similar difference.

QUESTION 5 (Extra Credit)
What single core performance do you get with eigenfaces_dgemm? How does your performance scale for 2, 5, 10, 20 cores? (You may even try 40 cores.)

ANSWER
I am sure that there is a simple mistake in my code, but I will say I am quite uncertain why this is not working. My *cblas_dgemm* call is:

```
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, sz, z_size, sz, 1.0, &F(0,0),
z_size, &F(0,0), sz, 0.0, &C(0,0), sz);
```

Where *F* has been allocated like:

```
size_t sz = z[0].num_rows();
  Matrix C(sz, sz);
  size_t z_size = z.size();
  Matrix F(sz, z_size);
  // Turn it into a matrix
  for (size_t i=0; i < sz; ++i) {
    for (size_t j=0; j< z_size; ++j) {
      F(i,j) = z[j](i);
    }
  }
```

This is more of a strange curiosity for me, at this point; what *is* wrong? F and C are both row-major oriented, so the first argument should be CblasMajorRow. We would like FF^T, so the next two arguments should be: CblasNoTrans and CblasTrans. The matrix shapes are sz, z_size, and z_size, sz, so the next three arguments should be: sz, z_size, sz. We don't want any multiplier, so we set the next argument to 1.0. Then we pass in a pointer/reference to the first element of the matrix, &F(0,0). Since it is row-major, the leading dimension is its number of columns, which is z_size. This is repeated for the next two arguments, since op(A) = op(B) here, except the applied transpose. Now the leading dimension is sz, however, since it is to be transposed (according to the documentation, but I also tried it both ways). We don't want to add C, but we do still need to specify it since this is where it will be read too. So the next argument is &C(0,0). Then the last parameter is the leading dimension of C, which is sz. In case I mixed up any rows/columns, I also tried every permutation of sz versus z_size. No matter the try, I always got:

Intel MKL ERROR: Parameter 14 was incorrect on entry to cblas_dgemm.

Anyhow, I know I was not able to get this, but I am saying this out of curiosity over *why* this didn't work. Egah! Probably something small that I overlooked. Alas, such is life.

QUESTION 6
How does the performance of your eigenfaces_mpi.exe scale for 1, 2, 4, 8 nodes? (With 10 cores.)

ANSWER
The following results are all from testing with a problem size of 50,000. I will be reporting the *[compute_covariance]* value.

Nodes	Time
1	11,384
2	5,985

4	3,151
8	2,151

Speedup from each of:
1 → 2: 1.90
1 → 4: 3.61
1 → 8: 5.29

QUESTION 7
What configuration of nodes and cores per node gives you the very best performance? (And what was your best performance?)

ANSWER

Size	Nodes	CPUs	Time	Performance (GF/s)
Small	8	40	15,607	206.27
Small	16	20	5,475	588.02
Small	32	10	9,514	338.38
Med	8	40	131,321	314.31
Med	16	20	70,467	541.15
Med	32	10	128,783	296.10

My best performance was on the small picture dataset with 16 nodes and 20 CPUs, with a performance of 588 GF/s.

In my reported *max.txt* file I have 471 GF/s, as I ran this after the above trials. Electrons are fickle beasts, and so this is lower than the mentioned 588.02.