Rohan Tiwari
AMATH 583
Final Exam

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?**

The time grows linearly as the number of images. 262781/500 = 525.562 so the time will be around 525 times more than what it is for 500 images. This comes out to be 5255.62 seconds which is about 87.59 minutes.

Rohan Tiwari
AMATH 583
Final Exam
Sequential

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

In the **'outer' method**, the following optimizations are applied:

- Switch order of the loops (A is stored as row-major) to reduce cache misses and increase the cache efficiency.
- Hoisted x(i) to reduce memory reads which are slow. This makes x(i) be stored in a local register which is must faster.
- Loop unrolling is implemented on the inner loop to improve instruction pipeline efficiency (reduce jumps)

```
// exchanging order of loops to get better cache efficiency as A is
stored row-major.
  // hoisting x(i) to reduce memory reads
  for (size_t i = 0; i < A.num_rows(); ++i) {
    double t = x(i);
    size_t innercond = A.num_cols() - A.num_cols() % 2;
    for (size_t j = 0; j < innercond; j += 2) { // unroll
      A(i, j) += t * x(j);
      A(i, j+1) += t * x(j+1);
    }
    if(innercond < A.num_cols()){ //check if we missed something!
      A(i, innercond) += t * x(innercond);
    }
  }
```

In the **'gen_covariance' method,** the following optimizations are applied:

1. Loop unrolling is implemented on the inner loop over C.num_cols() and on the single loop over z.size() to improve instruction pipeline efficiency (reduce jumps).

```
size_t sz = z[0].num_rows();
  Matrix C(sz, sz);

  size_t cond = z.size() - z.size() % 2;
```

```
  for (size_t k = 0; k < cond; k += 2) { // unroll
    outer(C, z[k]);
    outer(C, z[k+1]);
  }
  if(cond < z.size()){//check if we missed something!
    outer(C, z[cond]);
  }

  // Normalize
  for (size_t i = 0; i < C.num_rows(); ++i) {
    size_t innercond = C.num_cols() - C.num_cols() % 2;
    for (size_t j = 0; j < innercond; j += 2) { // unroll
      C(i,j) = C(i,j) / z.size();
      C(i,j+1) = C(i,j+1) / z.size();
    }
    if(innercond < C.num_cols()){ //check if we missed something!
      C(i,innercond) = C(i,innercond) / z.size();
    }
  }
```

Tried the optimizations over different input parameters. This is an example

Optimized:

```
[rtiwari6@klone1 eigenfaces]$ srun -A amath -p ckpt
./eigenfaces_opt.exe -i small.bin -n 500
# Face size 55 x 45
# Reading small.bin
# elapsed time [read_file]: 73 ms
# elapsed time [compute_mean]: 1 ms
# elapsed time [compute_shifted]: 0 ms
# elapsed time [compute_covariance]: 2819 ms
# Gflops/s = 2.17298
```

Rohan Tiwari
AMATH 583
Final Exam

Parallelization

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?**

`#pragma omp parallel for` directive is used in the 'outer' and the 'gen_covariance' methods. This directive divides loop iterations between spawned threads. Each thread can run in parallel.

In the 'outer' method, this directive divides rows of the A matrix between parallel threads. This is also the case with the nested for loop in the 'gen_covariance' method. For both these cases, the outer for loop is parallelized.

Tried the optimizations over different input parameters for small.bin. The overall GFlops/s and the time in milliseconds of compute_variance step is noted below.

This parallelized version includes the optimizations applied in eigenfaces_opt.

'outer' method:

```
// exchanging order of loops to get better cache efficiency as A is
stored row-major.
  // hoisting x(i) to reduce memory reads
  // add omp directive to parallelize
  #pragma omp parallel for
  for (size_t i = 0; i < A.num_rows(); ++i) {
    double t = x(i);
    size_t innercond = A.num_cols() - A.num_cols() % 2;
    for (size_t j = 0; j < innercond; j += 2) { // unroll
      A(i, j) += t * x(j);
      A(i, j+1) += t * x(j+1);
    }
    if(innercond < A.num_cols()){ //check if we missed something!
      A(i, innercond) += t * x(innercond);
    }
  }
```

'gen covariance' method:

```
size_t cond = z.size() - z.size() % 2;
  for (size_t k = 0; k < cond; k += 2) { // unroll
```

```
    outer(C, z[k]);
    outer(C, z[k+1]);
  }
  if(cond < z.size()){//check if we missed something!
    outer(C, z[cond]);
  }

  // Normalize
  // omp directive to parallelize the loop
  #pragma omp parallel for
  for (size_t i = 0; i < C.num_rows(); ++i) {
    size_t innercond = C.num_cols() - C.num_cols() % 2;
    for (size_t j = 0; j < innercond; j += 2) { // unroll
      C(i,j) = C(i,j) / z.size();
      C(i,j+1) = C(i,j+1) / z.size();
    }
    if(innercond < C.num_cols()){ //check if we missed something!
      C(i,innercond) = C(i,innercond) / z.size();
    }
  }
```

Let us next loop at performance numbers.

5000 faces (small.bin):

We note that the unoptimized version gives 0.98 GFlops/s overall and 62422ms for Compute_Covariance on small.bin for 5000 faces. Given below are some performance numbers for parallelized version.

Cores	Optimized (overall GFlops/s)	Optimized (Compute_covariance in ms)
1	2.24	27344
2	3.91	15702
4	9.45	6477
10	17.05	3592
20	25.92	2363

Comparing with unoptimized version: Comparing the run times of the compute_variance method, we see that 2 cores gives a speed up of 3.98, 4 cores gives a speedup of 9.64, 10 cores gives a speedup of 17.38 and 20 cores gives a speed up of 26.42 over unoptimized version.

Relative speedup comparison: Comparing the relative run times of the compute_covariance we see that 2 cores give a speed up of 1.75 over 1 core. 4 cores give a speed up of over 2.42 over 2 cores. 10 cores gives speedup of 1.80 compared to 4 cores. And 20 cores gives a speed up of 1.52 over 10 cores.

15000 faces (small.bin):

We note that the unoptimized version gives 0.78 GFlops/s overall and 234944ms for Compute_Covariance method on small.bin for 15000 faces. Given below are some performance numbers for parallelized version.

Cores	Optimized (overall GFlops/s)	Optimized (Compute_covariance in ms)
1	2.23	82047
2	5.31	34593
4	9.88	18593
10	16.64	11040
20	30.79	5967

Comparing with unoptimized version: Comparing the run times of the compute_variance method, we see that 2 cores gives a speed up of 6.79, 4 cores gives a speedup of 12.64, 10 cores gives a speedup of 21.28 and 20 cores gives a speed up of 39.38 over unoptimized version.

Relative speedup comparison: Comparing the relative run times of the compute_covariance, we see that 2 cores give a speed up of 2.66 over 1 core. 4 cores give a speed up of over 1.86 over 2 cores. 10 cores gives a speedup of 1.69 compared to 4 cores. And 20 cores gives a speed up of 1.85 over 10 cores.

Thus, we see that relative speedup stays around the same range for 15000 and 5000 faces for small.bin for optimized version. But we get much more speedup as compared to the unoptimized version for 15000 faces than we do for 5000 faces.

Rohan Tiwari
AMATH 583
Final Exam
Blocking for Cache

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?**

Blocking Approach: The matrix C is split into two halves and each half is operated upon independently as the result is symmetrical. The first half is computed in the initial part of the gen_covariance method by iterating on half of the matrix C and then the second half is computed using the first half. This reduces the overall work needed to be done. The loops use hoisting to reduce memory accesses that are slow. Also, the loop that iterates on rows of C is surrounded by `#pragma omp parallel for` directive to increase parallelism as different omp threads are spawned with the rows of C distributed among them.

'gen_covariance' method:

```
for (size_t k = 0; k < z.size(); ++k) {
    // using omp to parallelize the outer loop
    #pragma omp parallel for
    for (size_t i = 0; i < C.num_rows(); ++i){ // unroll
      double t = z[k](i);  // hoist - saves memory accesses which are
slow.
      // divide into half and compute
      for (size_t j = 0; j <= i; ++j) {
            C(i, j) += t * z[k](j);
      }
    }
  }
  // using omp to parallelize the outer loop
  #pragma omp parallel for
  //process the other half
  for (size_t i = 0; i < C.num_rows(); ++i){
    for (size_t j = i + 1; j < C.num_cols(); ++j){
      C(i, j) += C(j, i);
    }
  }
```

Let us look at performance numbers now.

We will compare the blocked version to non-blocked and unoptimized versions.

5000 faces (parallelized but without blocking) (small.bin):

Cores	Optimized (overall GFlops/s)	Optimized (Compute_covariance in ms)
1	2.24	27344
2	3.91	15702
4	9.45	6477
10	17.05	3592
20	25.92	2363

5000 faces (parallelized with blocking) (small.bin):

Cores	Optimized (overall GFlops/s)	Optimized (Compute_covariance in ms)
1	9.03	6780
2	12.43	4926
4	22.31	2745
10	47.30	1295
20	51.82	1182

We note that the unoptimized version gives 0.98 GFlops/s overall and 62422ms for Compute_Covariance on small.bin with 5000 faces.

Comparing with unoptimized version: For 10 cores, the speed up is 48.20 and for 20 cores the speed up is 52.81 over the unoptimized version.

Speedup over the non-blocked case: For 10 cores, the speed up is 2.77 and for 20 cores it is 1.99 for compute_variance method compared to non-blocking version.

Next, we look at 50000 faces.

50000 faces (parallelized but without blocking) (small.bin):

Cores	Optimized (overall GFlops/s)	Optimized (Compute_covariance in ms)
2	5.31	115251
4	9.76	62709
10	14.79	41396
20	27.88	21792

50000 faces (parallelized with blocking) (small.bin):

Cores	Optimized (overall GFlops/s)	Optimized (Compute_covariance in ms)
2	12.36	49563
4	21.49	28501
10	48.02	12754
20	66.69	9188

We note that the unoptimized version gives 0.37 GFlops/s overall and 1627920ms for Compute_Covariance on small.bin for 50000 faces.

We will compare the blocked version to non-blocked and unoptimized versions.

Comparing with unoptimized version: For 10 cores, the speed up is 127.64 and for 20 cores the speed up is 177.18 over the unoptimized version.

Speedup over the non-blocked case: For 10 cores, the speed up is 3.24 and for 20 cores it is 2.37 for compute_variance method compared to non-blocking. These numbers are better than what was seen for 5000 faces.

Also, overall, we see that blocking has been beneficial.

Using dgemm library call (Extra Credit)

5. **What single core performance do you get with eigenfaces_dgemm? How does your performance scale for 2, 5, 10, 20 cores? How does your performance scale with number of faces (500, 5k, 50k)? (You may even try 40 cores and 50k or more faces.)**

Call to Dgemm:

```
//copy into F
  #pragma omp parallel for
  for (size_t i=0; i < sz; ++i) {
    for (size_t j=0; j< z.size(); ++j) {
      F(i,j) = z[j](i);
    }
  }
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, sz /** row of c**/, sz/**col of c**/,
    z.size() /**col of F**/, 1.0 /**scale factor **/,
    &F(0,0), z.size() /**row major leading dim of F**/, &F(0,0), z.size() /**row major leading dim of Ft**/,
    0.0, &C(0,0), sz /**leading dim of c **/);
```

Let us look at performance numbers now.

500 faces (small.bin):

Given below are performance numbers for dgemm using 500 faces and small.bin.

Cores	Optimized (overall GFlops/s)	Optimized (Compute_covariance in ms)
1	44.39	138
2	67.31	91
5	92.81	66
10	125.04	49
20	109.39	56
40	70.41	87

Single core performance : Around 44.39 GFlops/s and the runtime of compute_variance is 138ms.

Relative performance comparison: These scales fast until 10 cores as the compute_variance times keep dropping quickly but then we see an increase in the time probably indicating that there is not enough

work for additional cores and communication costs slow it down. Comparing the run times of compute_variance for each configuration, we see that 2 cores is 1.52x faster than 1 core. 5 cores is 1.38x faster than 2 cores. 10 cores is 1.35x faster than 5 cores. 20 cores is 0.88x faster than 10 cores. 40 cores is 0.64 x as fast as 20 cores.

Let us also compare with the unoptimized version. We note that the unoptimized version gives 0.86 GFlops/s overall and 7116ms for Compute_Covariance on small.bin with 500 faces.

Comparing with unoptimized version: Comparing the run times of the compute_variance method, we see that 2 cores gives a speed up of 78.19, 5 cores gives a speedup of 107.82, 10 cores gives a speedup of 145.22 and 20 cores gives a speed up of 127.07 and 40 cores give a speed up of 81.79 over unoptimized version.

5000 faces (small.bin):

Given below are performance numbers for dgemm using 5000 faces and small.bin.

Cores	Optimized (overall GFlops/s)	Optimized (Compute_covariance in ms)
1	83.68	732
2	136.73	448
5	290.31	211
10	403.002	152
20	471.202	130
40	528.071	116

Single core performance : Single core performance is around 83.68 GFlops/s and the runtime of compute_variance is 732ms.

Relative performance comparison: The execution times for compute_Variance reduce well as we increase the number of cores so this scales well unlike 500 faces. Comparing the run times of compute_variance for each configuration, we see that 2 cores is 1.64x faster than 1 core. 5 cores is 2.12x faster than 2 cores. 10 cores is 1.39x faster than 5 cores. 20 cores plateau a bit and is 1.17x faster than 10 cores. 40 cores is 1.13x as fast as 20 cores.

Let us also compare with the unoptimized version. We note that the unoptimized version gives 0.98 GFlops/s overall and 62422ms for Compute_Covariance on small.bin with 5000 faces.

Comparing with unoptimized version: Comparing the run times of the compute_variance method, we see that 2 cores gives a speed up of 139.33, 5 cores gives a speedup of 295.84, 10 cores gives a speedup of 410.68 and 20 cores gives a speed up of 480.17 and 40 cores gives 538.12 speedup over unoptimized version.

50000 faces (small.bin):

Given below are performance numbers for dgemm using 50000 faces and small.bin.

Cores	Optimized (overall GFlops/s)	Optimized (Compute_covariance in ms)
1	8.29	73809
2	92.00	6658
5	216.53	2829
10	445.5	1375
20	545.96	1122
40	820.03	761

Single core performance : Single core performance is around 8.29 GFlops/s and the runtime of compute_variance is 73809ms.

Relative performance comparison: The execution times for compute_Variance reduce well as we increase the number of cores so this scales well unlike 500 faces. Comparing the run times of compute_variance for each configuration, we see that 2 cores is 11.08x faster than 1 core. 5 cores is 2.35x faster than 2 cores. 10 cores is 2.06x faster than 5 cores. 20 cores is 1.22x faster than 10 cores. 40 cores is 1.48x as fast as 20 cores.

Let us also compare with the unoptimized version. We note that the unoptimized version gives 0.37 GFlops/s overall and 1627920ms for Compute_Covariance on small.bin for 50000 faces.

Comparing with unoptimized version: Comparing the run times of the compute_variance method, we see that 2 cores gives a speed up of 244.50, 5 cores gives a speedup of 575.44, 10 cores gives a speedup of 1183.94 and 20 cores gives a speed up of 1450.91 and 40 cores gives 2139.19 speedup over unoptimized version.

Scaling with number of faces: Let us take an example for compute_variance method. 10 cores needed 49ms for 500 faces, 152ms for 5000 faces and 1375ms for 50000 faces. The ratio of runtimes between 5000 faces and 50000 faces is almost 9.05 (close to 10). So as we 10x number of faces, the execution times proportionately increases by around the same factor which is good. Increase in execution time from 500 to 5000 faces is around a factor of 3x which is great as well. This is because for 500 faces, the execution time includes perhaps a lot of communication costs between 10 cores (there may not be enough work to do).

We can compute a table for the jumps between execution times of compute_variance as below. It can be seen below that the execution time does not increase by a factor of 10x when going from 500 to 5000 faces indicating that all cores weren't getting utilized well on the smaller problem size but when we go from 5000 to 50000 we see an almost 10x jump in execution time till around 10 cores and after that it start plateauing again. As the number of cores is increased, the problem size should be increased proportionately to hide the communication costs between the cores.

cores	5000 faces: 500 faces	50000 faces : 5000 faces
1	5.30	100.83

2	4.92	14.87
5	3.20	13.41
10	3.10	9.05
20	2.32	8.63
40	1.33	6.57

MPI: eigenfaces_mpi

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

50000 faces on small.bin:

Given below are performance numbers for mpi using 50000 faces and small.bin.

Nodes (10 cores)	overall GFlops/s	Compute_covariance in ms
1	329.335	1860
2	637.422	961
4	1063.48	576
8	1642.26	373

This scales well as we increased number of nodes.

We will compare the millisecond runtimes of Compute_Variance for each configuration relatively. 2 Nodes has a speedup of about 1.54 over 1 Node. 4 Nodes has a speedup of about 1.45 over 2 Nodes. 8 Nodes has a speedup of about 2.45 over 4 Nodes. The jump is highest from 4 Nodes to 8 Nodes.

Let us also compare with the unoptimized version. We note that the unoptimized version gives 0.37 GFlops/s overall and 1627920ms for Compute_Covariance on small.bin for 50000 faces.

Comparing with unoptimized version: Comparing the run times of the compute_variance method, we see that 2 cores gives a speed up of 1693.99, 4 cores gives a speedup of 2826.25, 8 cores gives a speedup of 4364.39 over unoptimized version.

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

The following configurations were tried:

Size	Nodes	CPUs	overall GFlops/s	Compute_covariance Time in ms
med	4	16	1950.53	19550
small	4	20	1709	1883.79
Med	4	20	2046.31	18635
med	8	8	3073.75	12406
med	8	20	5646.81	6753
med	8	40	8007.76	4762

The best performance is highlighted. Max GFlops/s observed was 8007.76 GFlops/s and compute_variance time was 4762ms for this configuration of 8 nodes and 40 CPUs for medium sized problem.

I used the settings given in max.bash and only modified Nodes and MKL_NUM_THREADS.

```
# Face size 109 x 89
# elapsed time [read_file]: 14204 ms
# elapsed time [scattering]: 10835 ms
# elapsed time [compute_mean]: 212 ms
# elapsed time [compute_shifted]: 165 ms
# elapsed time [reduce]: 1192 ms
# elapsed time [compute_covariance]: 4762 ms
# Gflops/s = 8007.76
```