**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.

**Sequential**

**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:

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

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).

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

**Parallelization**

**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.

The single for loop looping over z.size() in ‘gen\_covariance’ is parallelized such that each spawn thread get a chunk of the vector to run the calls to outer.

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.

The parallelized version includes the optimizations applied in eigenfaces\_opt.

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.

|  |  |  |
| --- | --- | --- |
| 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.

|  |  |  |
| --- | --- | --- |
| 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.

**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. 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.

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

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 |

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.

Next, we look at 50000 faces. We note that the unoptimized version gives 0.37 GFlops/s overall and 1627920ms for Compute\_Covariance on small.bin for 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 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.

**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.)**

5000 faces (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 is around 83.68 GFlops/s and the runtime of compute\_variance is 732ms.

This scales fast until 10 cores but after that not so much. 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 over unoptimized version.

**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:

|  |  |  |
| --- | --- | --- |
| 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. 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.

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

Maxed total number of CPUs (nodes \* Cpu\_per\_node) to 80. The following configurations were tried:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Size | Nodes | CPUs | overall GFlops/s | Time in ms |
| small | 4 | 8 | 2632.38 | 1223 |
| small | 4 | 16 | 3265.11 | 986 |
| small | 4 | 20 | 4298.26 | 749 |
| Small | 5 | 16 | 3312.14 | 972 |
| Med | 4 | 20 | 5853.1 | 6515 |
| small | 8 | 4 | 1880.49 | 1712 |
| Small | 8 | 8 | 3083.71 | 1044 |
| med | 10 | 8 | 3883.19 | 9820 |
| Small | 20 | 4 | 1955.89 | 1646 |

The best performance is highlighted. It was 5853.10 GFlops/s for 4 Nodes and 20 Cores for med.bin.