Jason Lai
204995126
DIS 1A

CS133 Lab 1 Report

The performance measurements for this lab are taken from an arbitrary instance of running a version of the `gemm` program. Since AWS EC2 instances have shared tenancy, performance of the different versions of the program can vary drastically. Therefore, in each performance report, the respective version of the code was run multiple times, and an appropriate median value was selected.

Performance Comparison

Matrix Size / Version	Sequential	Parallel	Parallel-Blocked
1024^3	0.544273 GFlops 3.9456 s	109.818 GFlops 0.019555 s	176.472 GFlops 0.012169 s
2048^3	0.254113 GFlops 67.6073 s	128.765 GFlops 0.13342 s	174.589 GFlops 0.098402 s
4096^3	0.190301 GFlops 722.219 s	53.3434 GFlops 2.57649 s	176.444 GFlops 0.778938 s

Comparing the three different versions of `gemm`, the results above show that the sequential version of the matrix multiplication algorithm is by far the slowest for all three sizes, followed by the parallel version, with the parallelized and blocked version being the fastest. Evidently, dividing the work of the matrix multiplication into subtasks for threads when parallelizing results in significant improvements to the performance. However, for the sequential version and even the non-blocked parallelized version, we still see dramatically different speedup numbers for different sizes of matrices, with the greatest discrepancy being the parallelized version's poor performance on the 4096^3 size. This can likely be attributed to the poor memory efficiency of the sequential and parallelized versions of the program. In those versions, the use of spatial and temporal locality isn't very good, so the resulting cache misses begin to accumulate and impact performance negatively, which is especially apparent for large matrix sizes like 4096^3.

It should be noted that the discrepancy in execution performance for the non-blocked parallelized version of the program at the 4096^3 size may also be due to the AWS m5.2xlarge instance as well. I noticed that running the program across different sessions would sometimes give me values in the 80s and 90s of GFlops consistently in a particular session, while other sessions where I test would only yield performance values in the 50s and 60s.

Optimizations: `omp-blocked` at 4096^3

For each optimization I performed, I show the performance with and without the optimization in their respective tables, wherein each scenario is executed in three consecutive trials. The meaning of "with" and "without" in regards to an optimization is the most optimized version of the `omp-blocked` code, except we test with the specific optimization being examined as the only thing added or removed.

Blocked Matrix Multiplication / Loop Tiling:

Trial	1	2	3
With	183.305 GFlops 0.749783 s	182.656 GFlops 0.752447 s	174.882 GFlops 0.785896 s
Without	53.8344 GFlops 2.55299 s	54.7969 GFlops 2.50815 s	52.1176 GFlops 2.63709 s

For this optimization, I found that using loop tiling on all three loops: `i`, `j`, and `k`, and different sizes of blocks for each loop yielded the best performance, specifically the sizes: 32, 512, 128 for the `i`, `j`, and `k` loops respectively. The code that was used as the baseline without the use of tiling was simply the non-blocked parallel version of `gemm`. From the results above, we see that the use of tiling creates significant performance boosts, where the blocked version performs nearly 3x more floating point operations per second than the non-blocked version. This is most likely due to more efficient use of memory locality when performing parallelized matrix multiplication.

Loop Unrolling:

Trial	1	2	3
With	169.327 GFlops 0.811678 s	171.035 GFlops 0.803573 s	172.139 GFlops 0.798418 s
Without	82.0425 GFlops 1.67522 s	82.4092 GFlops 1.66776 s	83.0076 GFlops 1.65574 s

Another optimization I used is the loop unrolling transformation on the tiled k loop, and I found that unrolling 4 iterations produced the greatest performance improvement. From the above, we see that this optimization nets about 2x more floating point operations per second than the unrolled version of the program. This is probably due to the reduction in control/scheduling overhead that loop unrolling can cause, as discussed in lecture.

Reduce Writes to c[][]:

Trial	1	2	3
With	175.499 GFlops 0.783133 s	176.003 GFlops 0.780889 s	172.52 GFlops 0.796656 s
Without	119.839 GFlops 1.14686 s	118.118 GFlops 1.16358 s	121.381 GFlops 1.1323 s

In this optimization, I consolidated all writes to `c[ii][jj]` into a single line of code. That is, I initially followed the Loop Unrolling example in the lecture slides, which should that each iteration that was unrolled would have its code executed on separate lines:

```
    c[ii][jj] += a[ii][kk] * b[kk][jj];
    c[ii][jj] += a[ii][kk+1] * b[kk+1][jj];
```

However, I instead combined these into a single line:

```
    c[ii][jj] += (a[ii][kk] * b[kk][jj]) + (a[ii][kk+1] * b[kk+1][jj]);
```

so that the write to `c[ii][jj]` is done only once per iteration. The results above show that this optimization improves the performance significantly. I suspect that this is because we reduce the number of writes into `c[ii][jj]`, which reduces the use of slow memory that writing uses as discussed in class.

Loop Permutation:

Trial	1	2	3
With	165.752 GFlops 0.829184 s	180.947 GFlops 0.759552 s	180.325 GFlops 0.762174 s
Without	2.79917 GFlops 49.0999 s	2.81097 GFlops 48.8938 s	2.79528 GFlops 49.1683 s

For this optimization, I used loop permutations where the order of loops that I found to have the greatest performance is `i`, `k`, `j`, `kk`, `ii`, `jj` from the outermost to the innermost loop. This is done in order to manipulate the order in which data is read from the matrices being multiplied, so that the use of memory locality is improved. In C language, arrays are row-based, so they are laid out in memory contiguously row-by-row, so if the dimensions of the 2D array are large, this makes reading values down a column (as in `b[][]`'s case) potentially memory inefficient. This is because when a row is large, it takes up more space when a cache line is read into memory, so the next element in a column may not be in the cache, resulting in a cache miss. Loop permutation can help remedy this, and from the above results, we see that performance is improved greatly with this optimization, compared to when the loops are not permuted.

Parallelization:

Trial	1	2	3
With	176.433 GFlops 0.778986 s	174.138 GFlops 0.789251 s	173.605 GFlops 0.791675 s
Without	42.2889 GFlops 3.25 s	41.588 GFlops 3.30478 s	41.4141 GFlops 3.31865 s

Finally, the last optimization I tested is parallelization; using different thread counts, I found that spawning 4 or 8 threads were similar in performance, but I felt that using 8 threads was giving me greater maximum values in performance. Compared to the single-threaded scenario, it's evident that the performance is much better when parallelization/multi-threading is used. This is due to the fact that more work can be done at the same time, when each individual thread is given a portion of the work to do simultaneously as the other threads, effectively increasing our throughput in calculations.

Scalability: `omp-blocked` at 4096^3

Thread Count / Trial	1	2	3
1	40.4666 GFlops 3.39635 s	41.2968 GFlops 3.32808 s	40.9243 GFlops 3.35837 s
2	86.4351 GFlops 1.59008 s	87.0654 GFlops 1.57857 s	87.2512 GFlops 1.57521 s
4	178.061 GFlops 0.771865 s	178.777 GFlops 0.768774 s	178.487 GFlops 0.770021 s
8	180.846 GFlops 0.759977 s	175.978 GFlops 0.780999 s	178.124 GFlops 0.771593 s
16	156.911 GFlops 0.875904 s	159.941 GFlops 0.85931 s	163.432 GFlops 0.840954 s
32	157.863 GFlops 0.870624 s	156.289 GFlops 0.879391 s	157.093 GFlops 0.87489 s

From the results when testing different thread counts as shown above, we see that there are drastic performance gains when increasing the thread counts from 1 to 2 to 4, so the optimized `omp-blocked` code scales relatively evenly. This can be attributed to the threads all performing pertinent work, with each increase in thread count contributing to the amount of work and therefore the throughput of the program. However, when increasing the thread count to 8, I noticed that the performance is less consistent than when using a thread count of 4, where the performance may be slightly less, on par, or much better than when only using 4 threads. I believe this could be due to issues with load balancing at higher thread counts, wherein some threads are not being fully utilized during some executions, and the overhead that they contribute is effectively “cancelling out” their performance contributions.

I also observed a considerable performance dropoff when spawning more than 8 threads, as seen in the results when using 16 and 32 threads. This leads me to believe that the m5.2xlarge AWS instance only has 8 threads allocated to users. The professor confirmed this in lecture, noting that the machines used for the m5.2xlarge instances have 24 core CPUs, however tenants are only allocated 4 cores, which has 8 threads. This would mean that the degradation in performance when spawning more than 8 threads is likely due to the overhead that results from the system having to process more threads than are physically available on the system. However, thread counts larger than 8 have relatively similar performance, and this is likely due to OpenMP limiting the amount of tasks created based on the iteration count when using `#pragma omp parallel for`.

Discussion of Results:

Overall, the results that we observed from this lab show that memory efficiency/locality and parallelization can improve the performance of a program outstandingly. In particular, performing loop transformations such as permutations and tiling can help better utilize temporal and spatial locality, which reduces the amount of cache misses and the overhead caused by them. Parallelization can greatly improve a program’s performance by doing more work at once, but there are diminishing returns when increasing the thread count, as observed in the scalability of the optimized `omp-blocked` code. However, the biggest takeaway for me from these results is that despite having a sequential program that runs exceedingly poorly, as seen in the execution time of the sequential `gemm` which took minutes to finish, we can apply various optimizations that can improve the performance to such a degree that the same program can now be done in fewer than 5 seconds.