Computer Science 133, Lab 1

Michael Wu
UID: 404751542

January 24th, 2019

1 Speedup Summary

The following table summarizes the results of my implementation.

Problem Size	1024^3	2048^3	4096^3
Sequential Speed (GFlops)	0.595169	0.283576	0.195333
Parallel Speed (GFlops)	21.4397	21.965	19.6744
Parallel Speedup	36.02x	77.46x	100.72x
Parallel-Blocked Speed (GFlops)	64.2536	83.1584	94.6579
Parallel-Blocked Speedup	107.96x	293.25x	484.60x

The speedup differences are mainly due to the sequential version performing worse as the problem size increases, since there are more cache misses in the sequential version. The non-blocked parallel implementation began to drop off slightly as the problem size increased due to cache misses as well. The blocked version seemed to increase in performance as the problem size increased, since blocking works better when used on a larger array.

2 Parallel Blocked Optimizations

There seemed to be a range of block size values that could lead to good performance, as long as they were not too large or too small. The following table summarizes the performance of my blocked parallel implementation in relation to block size on the 4096^3 problem size. I chose to use a block size of 128 since it led to good performance in all three problem sizes of 1024^3, 2048^3, and 4096^3.

Block Size	Speed (GFlops)
32	60.5134
64	76.7047
128	94.7986
256	95.8581
512	65.7190

My blocked implementation uses SIMD instructions that multiply 16 floats at a time in order to achieve high performance. Additionally, I permuted the loops so that the j index would be on the inside in order to better utilize the cache with SIMD instructions. This gave me approximately 7 to 10 GFlops more speed.

Additional optimizations I tried performing on the regular parallel version were loop unrolling and transposing the b matrix so that the innermost loop traversed the rows of b instead of the columns. Loop unrolling gave me approximately twice the speed, and transposing the matrix gave me approximately ten times the speed. Note that loop unrolling led to small rounding errors on the order of 10^{-7} as the array size increased, as the floating point addition might have been performed in a different order. These optimizations only worked on the regular parallel implementation since the blocked version uses SIMD instructions which makes these optimizations redundant.

3 Parallel Blocked Thread Performance

The `m5.2xlarge` instance supports up to 8 threads, since there are 8 cores on the machine. The following summarizes the scalability of my blocked implementation with different thread numbers on the 4096^3 problem size.

Threads	Speed (GFlops)
1	21.8415
2	42.7087
4	80.9481
8	94.7224

4 Results

In my implementation, it seems that the blocked parallel implementation becomes faster as the array size increases. Note that as array size increases,

the speedup over the sequential version increases as well because of cache efficiency. The regular parallel version seems to drop off due to cache efficiency as well. Some of the speed I obtained from blocking is not dependent on multithreading, because even with a single thread the optimizations can lead to a speed of 21.8415 GFlops. With additional threads, the blocked implementation can become over four times faster.

5 Challenges

Two main challenges I faced were related to loop unrolling and testing my implementations. Loop unrolling caused small rounding errors that led me to believe my implementation was incorrect. I only was able to fix this after reading on Piazza that the TAs would raise the error threshold to the order of 10^{-5}, so these small rounding errors would be acceptable. Testing my implementation was also a challenge because I initially made sure that my implementations worked for arbitrary problem sizes, which reduced my ability to use SIMD for my blocking implementation. Without SIMD, for most problem sizes the parallel implementation performed better than the blocked parallel implementation. I was considering submitting my parallel implementation since I was confused on which problem size my implementation would be tested on. There were so many tuning parameters that I wasted a lot of time fiddling with. After the TAs mentioned that they would only test dimensions that were a multiple of 1024, it became much easier to use SIMD. In the end I chose to optimize for the 4096^3 problem size, as that is what the TAs gave their benchmarks for.