Graham Wing

UID: 004738350

Lab 1 Report

Results:

	1024^3 (Time(s), Perf(GFlops))	2048^3 (Time(s), Perf(GFlops))	4096^3 (Time(s), Perf(GFlops))
Sequential	3.49124, 0.615106	68.8984, 0.249351	720.259, 0.190819
Parallel	0.019571, 109.728	0.178856, 96.0542	3.51188, 39.1354
Parallel Blocked	0.01113, 192.946	0.094598, 181.609	0.743338, 184.894

Testing performed on m5.2xlarge aws server: 4 core (8 thread) CPU @ 2.5 GHz with 32MiB of L3 cache.

Optimizations:

Starting with the base sequential code provided, the first optimization I applied was parallelizing it. Using $\#pragma\ omp\ parallel\ for$ the performance improved from ~0.2 GFlop to ~1.0 GFlops. Following this I permuted the loops so that the order of k and j were swapped, improving cache usage of matrix B; this loop ordering improved temporal locality due to utilizing row-major ordering. This massively improved performance, reaching ~40 GFlops. This achieves the same performance as the parallel function found in $omp.cpp$.

Next, I added blocking in both i and k. The effect is to have each thread work in a block of each matrix defined by i and k. By only using i rows in matrix A and k rows in matrix B, thrashing is mitigated. For instance, when iterating through all kK columns of A only one row is needed in cache due to spatial locality; however, iterating through kK rows of B requires a read for every single value due to the length of each row and the limited size of cache. By limiting the movement in the k value, the rows of B in use can be held in cache, greatly improving performance. The same effect is true for blocking in i. There is no need to block j due to only ever being used to index within a single row. Blocking i and k (with a block size of 8 for each) roughly doubled the performance. By replacing a function call to *std::min* with a ternary operation, I further improved the performance to ~100GFlops.

Following blocking, I optimized the block size to further improve performance. I knew the reasonable limit for a single block is 1024 elements due to the size of the L3 cache being 32MB. Through trial and error, I found the best performance to be ~107 GFlops using i block size of 1024 and k block size of 32.

Trying to squeeze more performance out of it, I iteratively tested different loop unrolling. I found unrolling in k was much more beneficial than unrolling in j (which was detrimental) due to it increasing the number of arithmetic operations for every write performed. An unrolling of 8 provided the best performance presumably due to higher unrollings causing too much register pressure. This improved the performance to 130 GFlops.

After unrolling, I found that more performance could be extracted by reducing the block size. The final values were i and k block sizes of 64 and 32 respectively, resulting in overall performance of 185 GFlops.

<u>Thread Comparisons:</u>

Number of threads on system: 8

Num Threads	4096^3 (Time(s), Perf(GFlops))
1	4.16136, 33.0274
2	2.08104, 66.0435
4	1.03379, 132.946
8	0.744162, 184.69
16	1.37054, 100.281

As indicated by the table above, my code scales as expected. On a hyperthreaded quad core system, performance scales linearly when the number of threads is less than or equal to the number of physical cores. When utilizing all the system's threads (8 in this case) performance improves, but not linearly due to hyperthreading's limitations compared to multiple physical cores. Increasing the number of threads beyond this however results in a drop off in performance due to the added overhead of context switching between threads.

<u>Discussion:</u>

The process and results of this lab indicate the benefits of thoroughly optimizing sequential code before considering parallelization. As indicated by the results above, a single optimized thread can nearly achieve (6 GFlop separation) the performance of poorly optimized multithreaded (8 threads) code. This also indicates the necessity of understanding the system the code is being deployed on. My optimized code is fine tuned specifically for a system with similar cache behavior as the server I tested it on. Although my code scales well with regards to threads, its performance suffers when run on systems with much smaller caches.

Additionally, this lab exemplified the need to understand cache (and memory) effects when trying to write fast code. For large problems, the process of optimization is not as simple as throwing more threads at the problem. Every read from memory must have as much locality extracted from it as possible to best improve performance.