Comparing results

sequential
1024 - 3.431s - 0.627 GFLOPS
2048 - 60.290s - 0.285 GFLOPS
4096 - 726.072s - 0.189 GFLOPS
parallel
1024 - 0.0370s - 58 GFLOPS - 93x speedup
2048 - 0.282s - 61 GFLOPS - 214x speedup
4096 - 2.421s - 57 GFLOPS - 300x speedup
parallel-blocked
1024 - 0.00978s - 219 GFLOPS - 351x speedup
2048 - 0.0773s - 222 GFLOPS - 780x speedup
4096 - 0.592s - 232 GFLOPS - 1225x speedup

The parallel implementation doesn't scale perfectly, but I didn't put much time into optimizing it because that time would be better spent on the blocked one. The blocked version does scale with the size of the data seemingly around O(n) better than the sequential one. The blocking allows it to cache values so that they are reused multiple times with one memory access. There are also other (not exclusive to parallelization/blocking) optimizations that I talk about in the next section.

Impact of each optimization

My initial run only used loop interchange, and it got me to 0.5GFLOPs. Then introducing the blocking (128x128x128) was very effective and brought me to around 20GFLOPS. Then I introduced an array private to each thread to store the blocks it is working on, which brought me up to 80GFLOPS. I implemented 4x8x4 unroll-and-jam on the block-level loops, which brought me up to 200GFLOPS. The unroll and jam also let me get fewer += operations which was nice. Then I decided to try uneven (128x256x128) blocks, which got me to 220GFLOPS. Finally I used memcpy and memset when I wanted to move large chunks of data which bumped me up a little to 230ish GFLOPS.

Thread count scalability

1 - 51 GFLOP
2 - 95 GFLOP
4 - 191 GFLOP
8 - 233 GFLOP
16 - 226 GFLOP
23 - 221 GFLOP

The m5.x2large has 8 vCPUs so it makes sense that performance peaks at 8 threads. Past that there is just the overhead of dealing with more threads without being able to do extra work in parallel. The increases from 1->2->4 each fall just a bit short of doubling the performance, which means my parallelization is not bad. It is interesting that going 4->8 only results in about a 20% speed boost. I attribute this to there only being 4 physical cores with 2 threads per core. Since this program is mainly bottlenecked by memory, most of the benefit comes from utilizing caches effectively. Running ‘lscpu’ confirms that there are only 4 instances of L1 and L2 caches. Having 2 threads per core (8 total) mostly just speeds up computations and not memory access so it only helps a little.

Also note that the 1 thread version is substantially better than the sequential version (over 250x speedup). This shows that the speedup numbers from the first part are not fair for assessing the benefits of parallelism because we are not comparing it to the best sequential algorithm.

Results discussion

I thought this lab would be mostly about OpenMP, but that was not my experience. I tried a few things involving atomic, single, critical, barriers, etc. but what I found was that my code ran fastest if I just used an ‘omp parallel for’ on the top level loop and organized my code in a way that I didn’t need to micromanage the threads. Most of my performance gains came from optimizing for locality/caching which is not the same as parallelism. Loop interchange and tiling were the first obvious choices. But I was surprised by the improvements that came from unroll-and-jam and using a local variable to store the blocks each thread works with. I guess the compiler has trouble using SIMD instructions without unroll-and-jam (I checked the assembly to confirm it is doing SIMD), and without a local variable there might be some confusion about what info should be kept in the cache.