Assignment 1: Matrix Multiplication Optimization

Group 25 members: Ben Shulman, Eric Lee, Wensi Wu

Optimization Methods

Multi-level Blocking

We started our optimization by working off the basic blocking DGEMM. The basic blocking DGEMM divides the matrix up into smaller blocks which can then be multiplied using an optimized inner kernel. We built on top of this DGEMM by adding in multiple levels of blocking. In total we created 3 levels of blocking.

Each level was designed such that the two blocks (one for A, one for B) could both fit in the cache corresponding to that block level. We partitioned both matrices A and B into big blocks that fit in L3 cache, which in turn are partitioned into smaller blocks that fit in L2 cache, which in turn are partitioned into tiny blocks that fit into the L1 cache. We then do all computation on the tiny blocks.

In order to make full use of each cache we calculated the optimal block size for each cache (L1, L2, L3) such that 2 blocks could fit into each cache to maximize our use of the caches. To calculate the maximum (square) matrix dimension for each cache we did the following: $N = \sqrt{Size/16}$. This equation is derived from the fact that we are storing doubles in each cache (8 bytes per double) and have to store 2 matrices of size N^2.

The L1 cache is 32KB, L2 is 256KB, and L3 is 15MB shared among 6 cores (2.5MB per core)[1]. For L1 this give us $N \approx 44$, L2: $N \approx 126$, L3: $N \approx 395$. We rounded each of these N down slightly to account for other data which the processor might bring into each cache. This gave us a big block size of $N = 360$, smaller block size of $N = 120$, and tiny block size of $N = 40$.

[1] http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2620%20v3.html

Copy Optimization

After implementing multi-level blocking we moved onto implementing copy optimization to help the processor take better advantage of the memory layout. To do this, the tiny blocks for A and B are copied into pre-allocated buffers before we run our kernel function. The kernel function then operates on these buffers, storing the results directly into the C (result) matrix.

We do not create a pre-allocated buffer for C. We chose to do this because C will have to be frequently written to meaning we would not only have to copy the current values of C into a pre-allocated buffer, we would also have to write the new values back to C which is even more overhead.

Compiler Flag Optimization

We also spent some time trying to optimize compiler performance by trying various flags. In particular we tried using the flags -axAVX and -xCORE-AVX2 to tell the compiler to use vector instructions (AVX and SSE) where possible.[2] We also tried other suggested flags such as -funroll-loops, plus others suggested by Intel.[3] None of these flags improved our performance, and in some cases slowed our code down. We believe this is likely because we are currently using the naive inner kernel that does standard matrix multiplication without any loop unrolling or explicit use of Intel's AVX instructions.

Loop Unrolling

We have briefly tried loop unrolling by taking the inner loop of our naive kernel and having it advance each time by 4 steps rather than 1, and by doing the 4 multiplies and additions inside the for loop explicitly (including handling edge cases of less than 4 steps remaining). We found that this did not increase performance, probably because the Intel compiler was not able to convert the inner loop into vectorized instructions without explicit use of AVX instructions. Also performance may not have improved due to memory misalignment.

---

[2]https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations

[3] https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler#s6

Results

Our current optimized square DGEMM which uses multi-level blocking and copy optimization to speed up performance has improved performance dramatically. For most matrix sizes, particularly as dimensions get larger, our performance is now 4.5-4.7GFlop/s while the naive basic DGEMM is close to 3 times slower as matrix size increases. Table 1 at the end of this section provides a comparison of DGEMM_mine's computation performance before and after tuning.

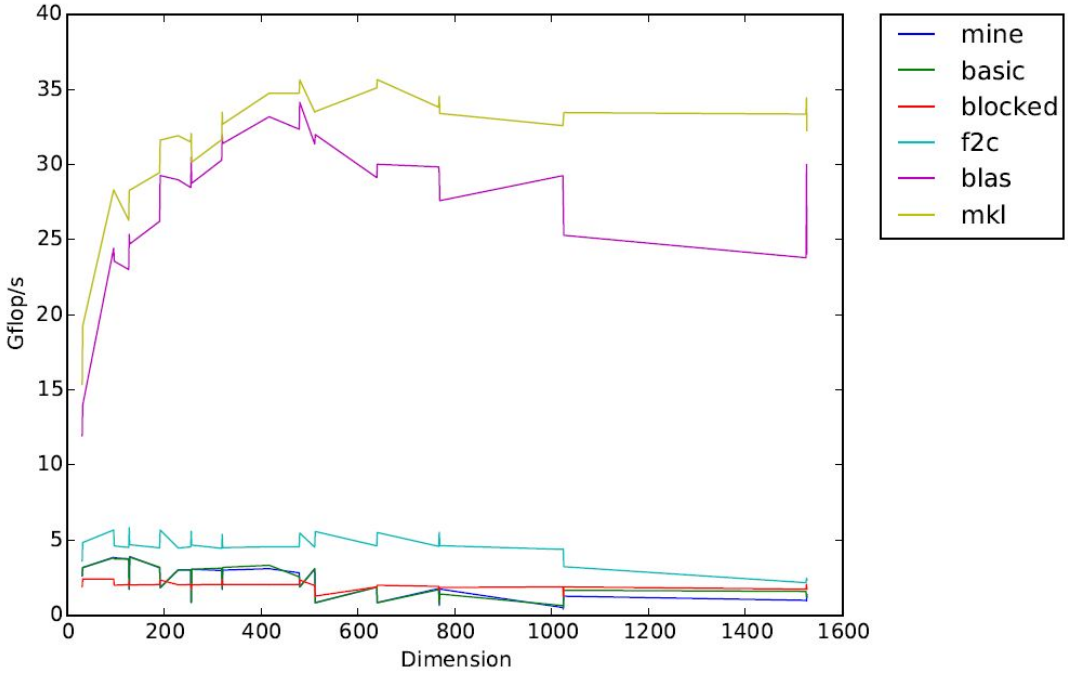

Figure 1. DGEMM_mine Gflop/s vs. Matrix Dimension before Tuning

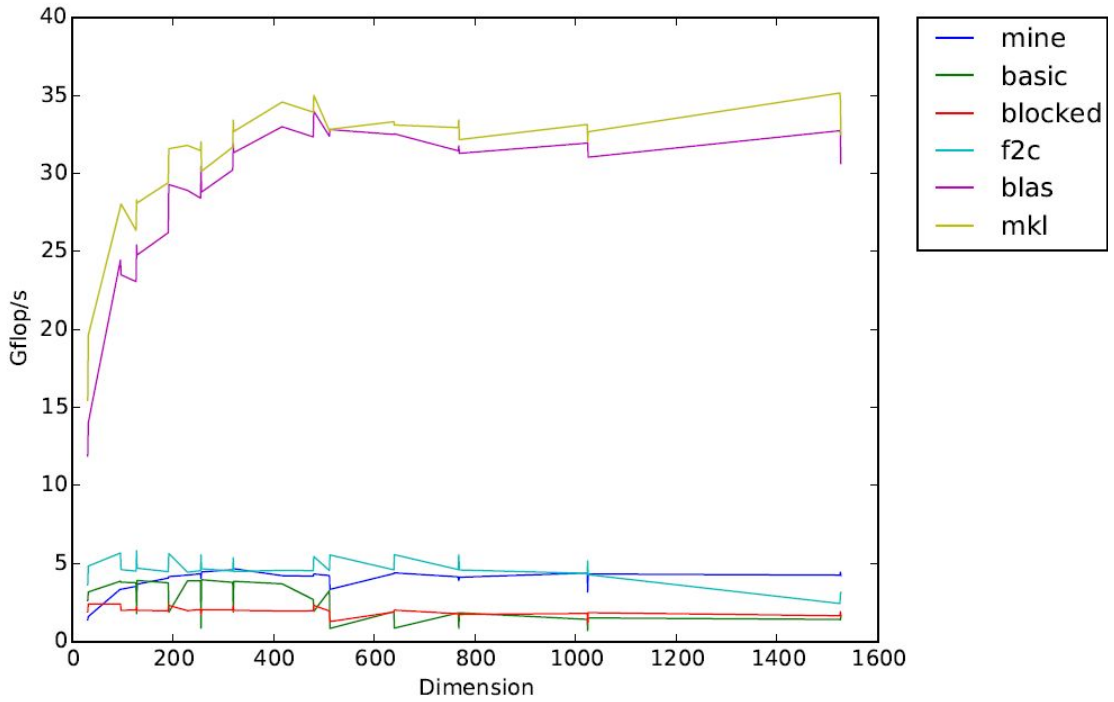

Figure 2. DGEMM_mine Gflop/s vs. Matrix Dimension after Tuning

Figures 1 and 2 provide a visual representation of the performance of DGEMM_mine compares to other matrix-matrix operation program. DGEMM_mine was one of the slowest matrix-matrix operation programs before tuning. However, after we implemented the blocking technique, DGEMM_mine performs almost as good as the DGEMM_f2c.

Matrix Dimension	Before Tuning (Mflop/s)	After Tuning (Mflop/s)
31	1240	2670
32	2213	3119
96	3448	3847
97	3765	3874
127	3890	3756
128	1759	1800
129	4028	4028
191	3550	4039
192	1828	4186
229	3459	4294
255	3425	4294
256	839	3950
257	3458	4475

319	2976	4688
320	1449	4767
321	3015	4764
417	3188	4330
479	2490	4436
480	1862	4532
511	3025	4415
512	808	3391
639	1853	4720
640	643	4649
767	1521	4527
768	608	4074
769	1517	4541
1023	1034	4585
1024	481	3300
1025	703	4616
1525	1642	4639
1526	1629	4628
1527	1639	4634

Table 1. DGEMM_mine's Computation Performance Comparison

Future Work

Kernel Tuning

Thus far we are using a completely naive kernel for doing matrix operations. We plan on optimizing the kernel by using loop unrolling as well as Intel's provided AVX instructions.[4] This will allow us to take full advantage of the vector capabilities of the processor which we aren't taking advantage at the moment.

Automatic Parameter Tuning

We also plan on tuning our parameters such as possible compiler flags, block sizes and AVX instruction tuning using automatic tuning which will sweep over a large number of the possible combinations to find which work best. We do not plan on tuning too much manually

[4] https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

because tuning on many combinations of parameters manually is inefficient particularly when parameters may not be independent.