Assignment 1:
Matrix Multiplication

Lara Backer
Unmukt Gupta
Edward Tremel

1 Overview

Matrix multiplication is one of the most fundamental computations required to solve linear systems, and as such is of vital importance in numerous computational applications. Therefore, reducing computation time for matrix multiplication calculations is an obvious method for increasing the efficiency of many solvers.

In this paper, we will be discussing the implementation and results from multiple methods for optimizing matrix multiplication of equation 1, with matrix dimensions M, N, and K. Results are compared to often-used matrix multiplication routines, such as those present in BLAS, a basic blocked multiplication method, and a naive matrix multiplication routine.

$$C (M x N) = A (M x K) * B (K x N) \tag{1}$$

2 Optimization Techniques

2.1 Blocking

Blocking techniques take advantage of the temporal locality of inner loops. Each chunk, or 'block', of a matrix, is chosen so that the code loads the necessary block into cache for each loop, discarding the block when the loop is finished. Matrix multiplication is performed block by block. By knowing the cache level sizes, the block sizes can be chosen to fit within the cache and avoid cache misses, which will decrease code performance.

Each compute node on the totient cluster used to run this code has an Intel Xeon E5-2620 v3 processor. We first computed the cache sizes to try to figure out the optimal block sizes, as shown in the next subsections. We initially used the results from these calculations for block sizes, but as we will see later, further testing revealed that other block sizes performed even better.

2.1.1 Level 1 Cache

The processor level 1 cache size is 32KB. To fit three blocks for A, B, and C for a square matrix and double matrix values (8 bytes), the formula to compute the optimal desired block size is:

$$3*(\text{block size})^2*(\text{double size}) = \text{cache size} \tag{2}$$

Thus, the maximum size for this blocking and for square matrices is 36x36.

2.1.2 Level 2 cache

For Xeon boards, the processor level 2 cache size is 256KB. In order to optimize for this cache as well, we performed two stages of blocking, one for the level 1 cache (smaller blocks), and one for level 2 (larger blocks).

Using the same formula as for the level 1 cache, the maximum size for the level 2 blocking is 103.

The first three lines of Figure 1 show the improvement gained from adding one and two levels of blocking, respectively. We were surprised to see that the second level of blocking appeared to make little difference, but kept it for the small gains it produced at some sizes.
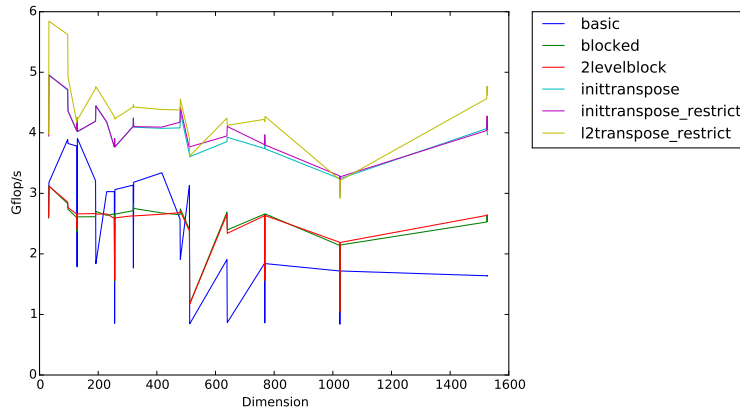

Figure 1: Performance of a series of optimization techniques. Blocking was performed with 36x36 small blocks and 103x103 large blocks

2.2 Copy Optimization

Copy optimization improves spatial locality and reduces cache conflicts by copying fixed size blocks and storing them elsewhere for the computation. Due to the additional data storage, it primarily optimizes code for large matrix computations. We used copy optimization on matrix A, since we realized that the pattern of iteration over matrix A would be much more cache-efficient if it was stored in row-major rather than column-major order.

We first tried copying the entire input matrix A into a row-major format before iterating over it in blocks. The results of this optimization are the fourth line on Figure 1. As an alternative, to improve spatial locality for the block iterations, we tried copying each large block of matrix A to a row-major format before splitting it into small blocks. This did improve performance compared to transposing the entire input matrix at once, as shown on the last line of Figure 1.

2.3 Compiler Hints

C contains some special keywords that can be used to provide hints to the compiler about the layout of memory and allow it to use more aggressive optimizations. We tested the `restrict` keyword, which specifies that the memory region referenced by a pointer will not overlap with a memory region referenced by any other pointer in scope. This keyword could be applied to all pointers to matrices A, B, and C, since they are allocated in distinct sections of memory and the pointers are never used for any matrix other than their own.

The fifth line on Figure 1 shows the results of adding the `restrict` keyword to all applicable pointers in our code that implemented the copy optimization with initial transposing. Unfortunately, this seemed to make little difference, but we kept it in future iterations of the code (such as the copy optimization at the large-block level).

2.4 Index Order

After implementing copy optimization, we tested all index orders for loops in the smallest blocked matrix multiplication routine (`dgemm_basic`). We found that the default i,j,k ordering was already optimal for the code and blocking, as it was written.

2.5 Block Size Tuning

Although our calculations showed that block sizes of 36x36 and 103x103 should be optimal for the cache sizes of the cluster's Xeon boards, we experimented with a few other block sizes both smaller and larger than these, after implementing the copy optimization.

The results of different small- and large-block sizes for the implementation that copies matrix A at the large-block level are shown in Figure 2. Surprisingly, it appears that the largest two block sizes, 64x64 for the small blocks and 256x256 for the large blocks, resulted in the best performance.
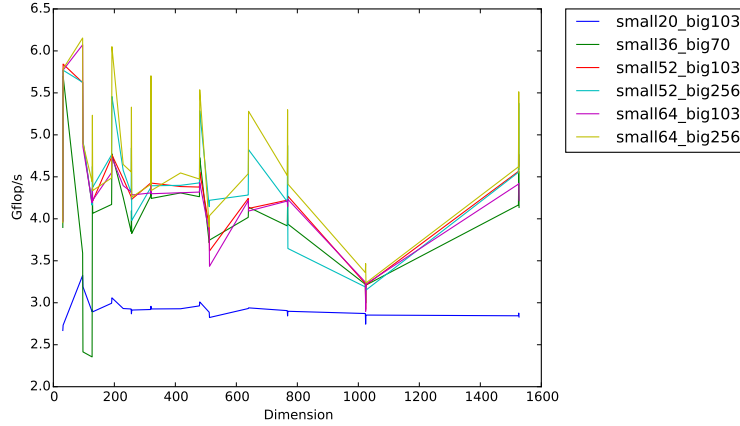

Figure 2: Comparison of Different Blocking Sizes, with Copy Transpose at Large Blocks

2.6 SSE Directives

We hoped to be able to use SSE instructions to force matrix C to be written without being cached, since caching C does not improve performance and only takes up cache space that could be used by A and B. We tested a set of Intel intrinsic functions that are documented to use non-temporal storage instructions. We modified dgemm_basic as follows:

```
void basic_dgemm(const int lda, const int M, const int N, const int K,
                 const double* restrict A, const double* restrict B, double* restrict C)
{
    int i, j, k;
        __m128d atemp,btemp,ctemp,cij
    for (i = 0; i < M; ++i) {
        for (j = 0; j < N; ++j) {
            cij = _mm_load_sd(&C[j*lda+i]);
                for(k = 0;k<K;++k) {
                        atemp = _mm_load_sd(&A[k*lda+i]);
                        btemp = _mm_load_sd(&B[j*lda+k]);
                        ctemp = _mm_mul_sd(atemp,btemp);
                        cij = _mm_add_sd(cij,ctemp);
            }
            _mm_stream_pd(&C[j*lda+i],cij);
        }
    }
}
```
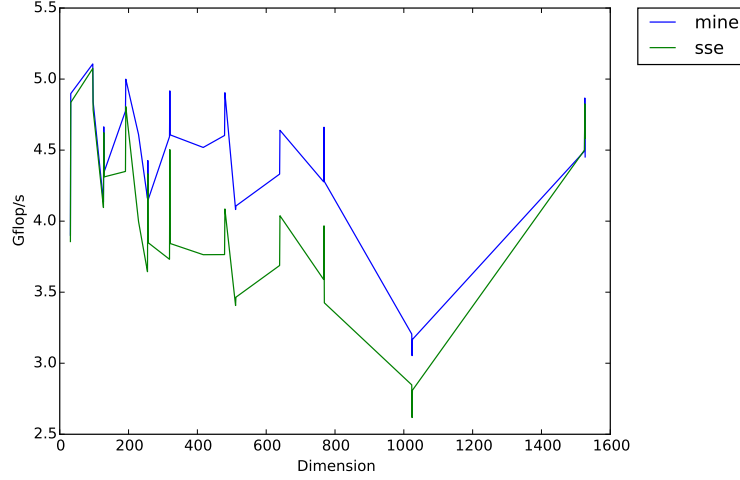
Figure 3: SSE Comparison Graph

As can be seen in Figure 3, the timings for the modified SSE code were actually worse when compared to the basic blocked dgemm_mine code.

2.7   Compiler Flags

Compiler flags that were found to improve performance were for this code are listed as follows:

- -O2: Maximizes speed, enables vectorization. Basic optimize flag.

- -mfpmath=sse: Enables XMM registers in floating point operations

- -Ofast: Enables high level aggressive optimizations -O3, -no-prec-div, and -fp-model fast=2.

Additional options were tested, such as -ipo, -flto, and -march=native. However, none of the flags were actually shown to increase efficiency. A comparison of the timings for options with all bulleted flags (flags), just the -O3 flag (o3), and no flags used (noflags) is shown in Figure 4.
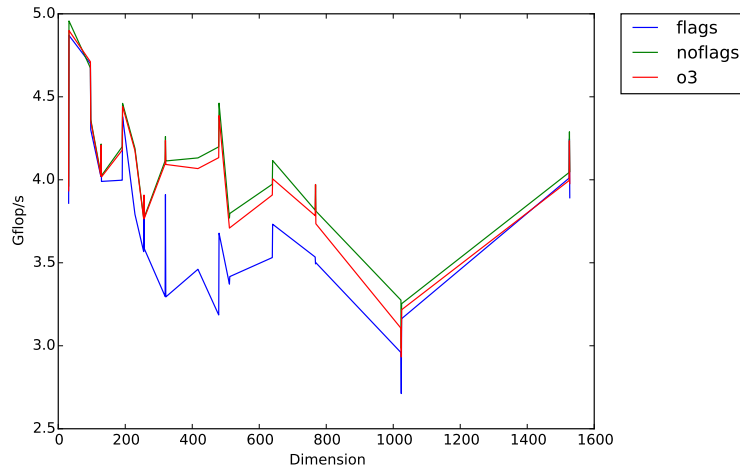

Figure 4: Flag Comparison Graph

3   Conclusion

After implementing all of the optimizations we found to improve performance, we were able to achieve a performance level a little less than halfway between the naive implementation of matrix multiply and the finely-tuned BLAS library. We also reduced the variation in speed with matrix dimension compared to the naive implementation, smoothing out many of the performance dips at powers of 2. The final timing graph for our code is shown in Figure 5, compared to these two benchmarks.
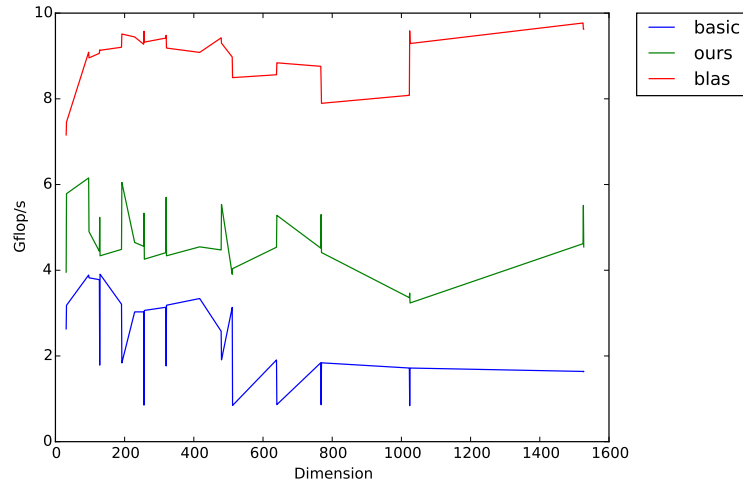

Figure 5: Our optimized implementation compared to the naive matrix multiply and the BLAS library code.