Matrix Multiplication Optimization

Eric Lee, Ben Shulman, Wensi Wu

Introduction

Matrix Multiplication is an algorithm for which the naive implementation is extremely simple, but also extremely inefficient when it comes to cpu and memory utilization. This makes it a prime target for optimization as its simplicity allows us to focus on optimizations instead of corner cases and it lends itself easily to being broken down and optimized.

There are many possible optimization techniques that we tried and most increased performance as compared to the base performance before the optimization. We compared each optimization to a base performance which may not be the basic DGEMM. For instance methods which make the most sense when they build on top of other methods (such as copy optimization building on top of blocking, or memory alignment building on top of copy optimization) are reported against the method they build against.

We report our findings for each technique we tried as well as how we combined them to create our most successful DGEMMs. Further for each optimization we try to shed light on why the optimization improves performance. As a note all results in this report used an optimized level of 3 (-O3).

Optimization Techniques

Loop Ordering

One of the first and easiest things to do is to take the basic matrix multiplication algorithm and permute the ordering. The default ordering of ijk is not necessarily the best for optimizing cache use. To that end we took the basic implementation of matrix multiplication and tried all permutations of loop ordering.

Results

The speed of each loop ordering is shown in Figure 1. We can see that the best two loop orderings are jki and kji. jki and kji being fastest is because both result in unit stride over A and C. Orderings ending in j have no matrices with unit stride and thus are the slowest as this is the worst access pattern for the cache. Orderings ending in k have unit stride for only B, which is not as good as having both A and C be unit stride as cache performance is much better with 2 arrays having unit stride rather than 1.

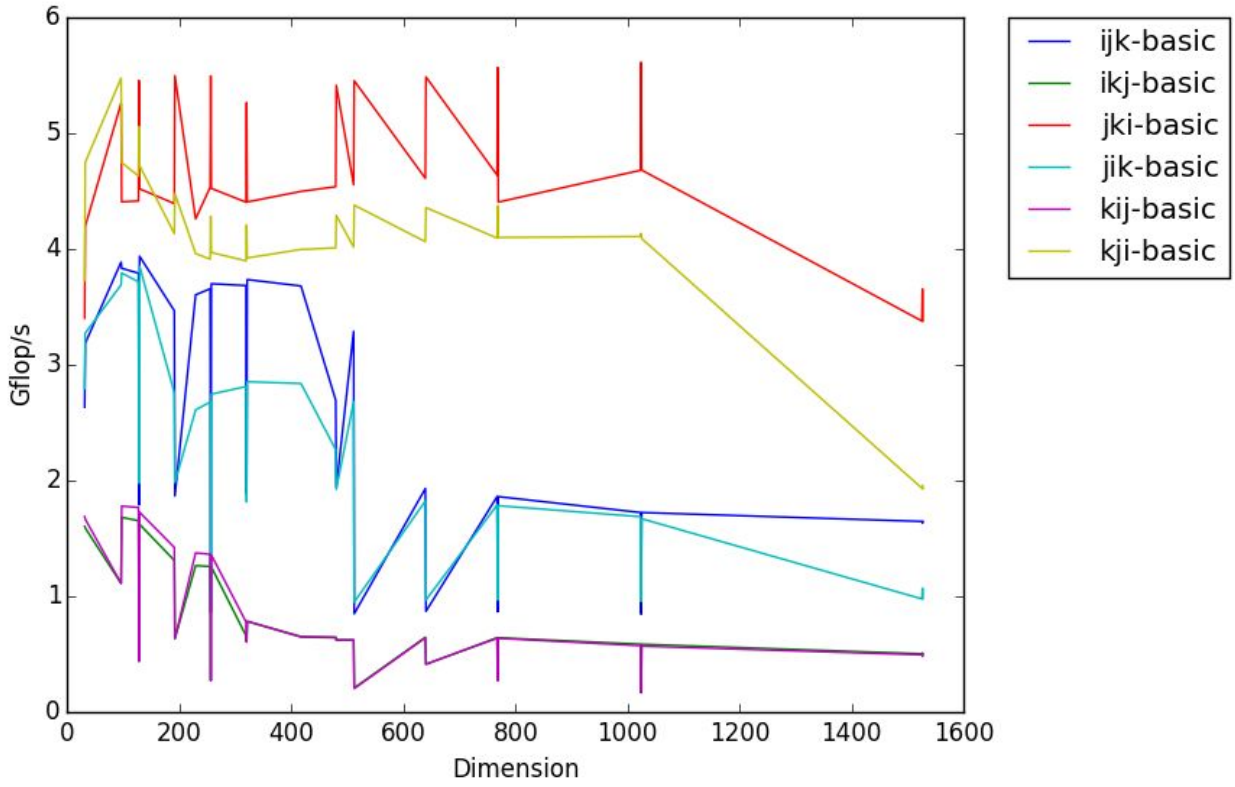

Figure 1: Speeds of various version of the basic dgemm with the loop order permuted

Loop Unrolling

Loop unrolling can be an effective and simple way of getting better performance out of code by reducing the number of branches (jumps) that the CPU must take. This reduces dependencies and increases CPU utilization. Unrolling is essentially taking a loop and doing multiple steps within the loop to reduce the number of times the loop is executed. For example:

```
for (k = 0; k < K; ++k) {
        cij += k;
}
```

can be re-written instead as:

```
double cij = 0;
double K = 10;
for (k = 0; k < K/4*4; k+=4) {
        cij += k;
        cij += k+1;
        cij += k+2;
```

```
        cij += k+3;
}
for (k = K/4*4; k < K; ++k) {
        cij += k;
}
```

This loop unrolling can help the compiler optimize loops and take advantage of vectorization. With these improvements in mind we took the basic 3-for-loop implementation and added loop unrolling (size 4) to try and improve speed and compiler performance.

Results

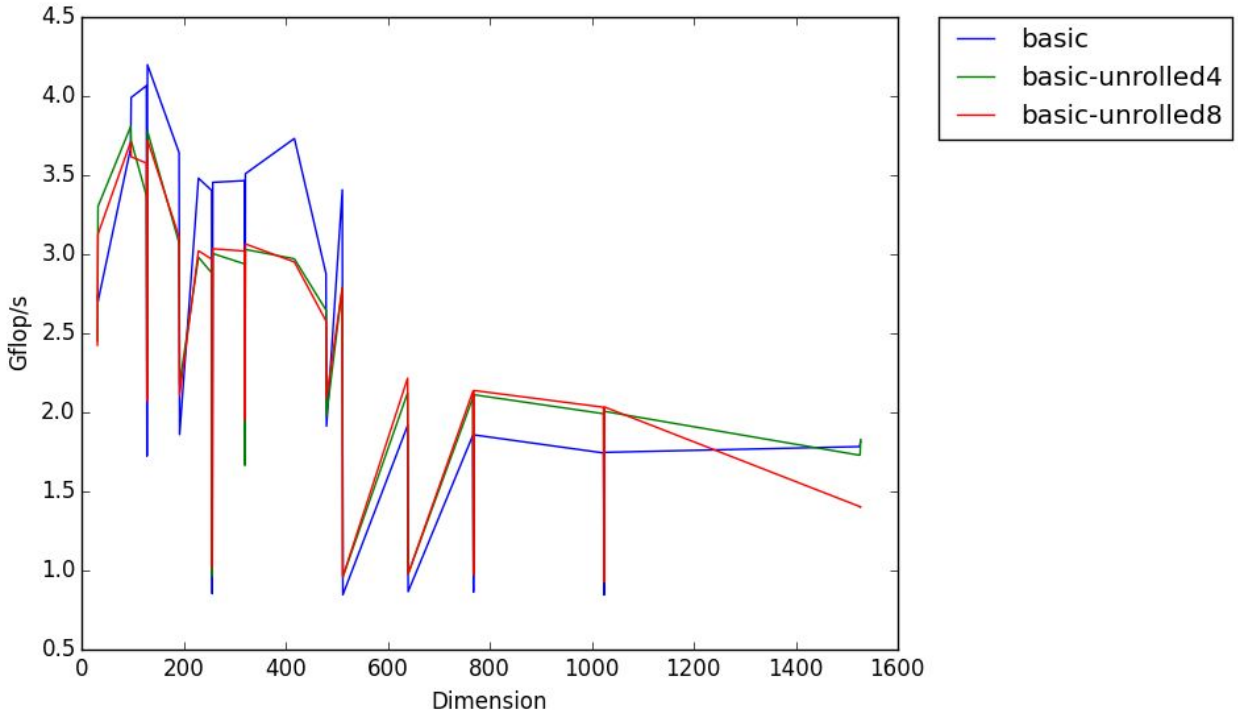

Figure 2: Timing of the basic DGEMM versus the basic with loop unrolling factor 4 and 8

Figure 2 shows the timings for the basic DGEMM versus when you unroll the loop to 4 instructions or 8. We find that unrolling provides higher performance at larger dimensions, but loop factor 4 performs better than 8. However, in neither case is performance much higher than basic. Basic performance is also higher for dimensions smaller than 512. These results suggest that the compiler is not performing much more vectorization than for the basic version, and results in worse performance on smaller matrices.

Blocking

The naive implementation of matrix multiplication loops over one entire dimension, then over the next, etc. With large matrices this may result in poor cache coherence and thus is unlikely to take full advantage of the cache. Instead it may be more efficient to break each matrix into a series of blocks, and then for each block of matrix A and B multiply these smaller matrices together using an optimized inner kernel. The size of these sub-blocks can be determined such that they will fit well into a chosen cache level.

Multi-level Blocking

The idea of blocking can be extended being a single level of blocks smaller than the whole matrix. Instead of a single block size which takes advantage of a single cache size, multiple levels of blocks with varying sizes can be built to fit into each cache level, or a subset of the caches.

In particular the smallest blocks can be made small enough that fit into the L1 cache, the medium blocks made to fit into L2 cache, and the large blocks into (some portion of) the L3 cache. So exactly how large should these blocks be to ideally fit into the caches? For each block we must fit 3 double matrices of size $N \times N$ into the cache. The total space taken by these matrices is thus $24N^2$.

Thus for each cache we have the following:[1]

$$\text{L1: } 24N^2 = 32,000 \rightarrow N \approx 36$$

$$\text{L2: } 24N^2 = 256,000 \rightarrow N \approx 103$$

$$\text{L3: } 24N^2 = 2,500,000 \rightarrow N \approx 322$$

We use 2.5MB for the L3 cache instead of the 15MB reported for an E5 2620 v3 because the 15MB is actually one L3 cache for each core of size 2.5MB that is accessible by the other processors.[2] Thus each core owns one 2.5MB L3 cache each.

Results

Figure 3 shows our experiments with a single level blocked DGEMM for various sizes as it compares to the basic DGEMM. In all cases, the inner block was calculated using a naive (basic) kernel. The basic DGEMM performs better than blocked dgemms on small matrices as the blocking overhead is not worth it when the blocks are larger or only slightly smaller than the matrix as there will be only a few blocks at most in each matrix.

---

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

[2] Architecture from: http://wccftech.com/intel-xeon-e52600-v3-haswellep-workstation-server-processors-unleashed-highperformance-computing/ (2600, not 2620, but close enough)

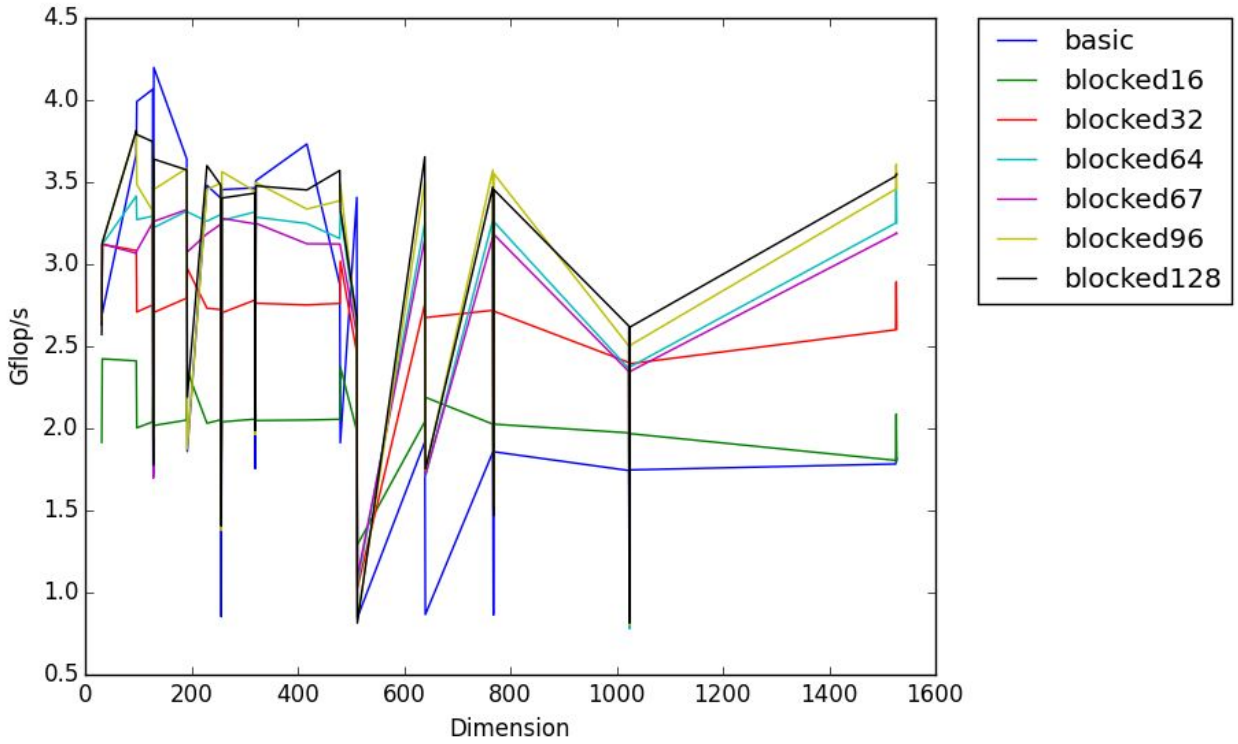

Figure 3: Timing of various block sizes as compared to the basic DGEMM

In general we found that performance goes up as block size increases, with the fastest blocks being size 96 and 128. This is contradictory to the idea that blocks should be designed to fit into the lower caches in order to increase speed. The fastest block size was 128, for which all three matrices blocks cannot fit into the L2 cache. This is interesting and indicates that blocking is advantageous but block performance is not necessarily correlated with cache size.

In Figure 4 we compare single level blocking and the basic DGEMM to a multi-level blocking DGEMM of various sizes. Once again, the inner kernel was the basic naive matrix multiplication algorithm. What is interesting here is that adding levels to try and take advantage of multiple cache sizes does not result in better performance. All levels of blocking perform similarly to a size 64 single blocking DGEMM with the two multi-level DGEMMs with the smallest block size being 48 performing worst of all blocking DGEMMs compared here.

This indicates that adding levels of blocking is likely not worth it, and does not result in the CPU taking better advantage of its caches.

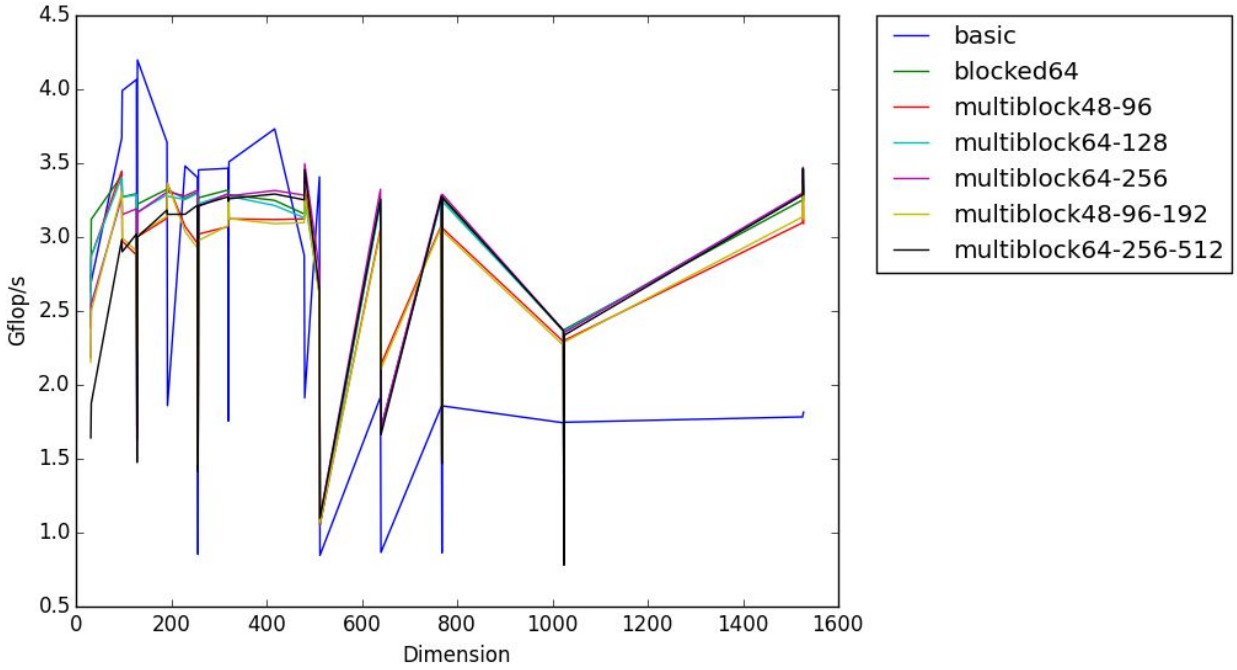

Figure 4: Timing of various multi-block sizes versus basic and 64 size block DGEMMs

Copy Optimization

Though blocking alone may be an effective way of breaking down a large matrix multiplication into smaller blocks it may be able to be improved upon by using copy optimization. Copy optimization is an optimization technique where the matrices being multiplied are copied into special memory locations (arrays) which can help the processor and compiler optimize computation and cache access. In particular this is effective for the inner kernel of a blocking matrix multiplication algorithm. By copying the sub-blocks of A and B into special memory locations, these blocks can be accessed and used more effectively.

It is important to note that copy optimization does not necessarily make sense in terms of the result matrix C as copying C into a special buffer does not make sense because the results must then also be copied back into C. This doubles the cost of copy optimization for C, potentially wiping out any gains.

Results

In Figure 4 we see that copy optimization adds large benefits over a basic DGEMM or a single-level blocked DGEMM. In particular using special memory buffers reduces performance drops on powers of 2 and multiples of 8 where cache line accesses are inefficient without specialized copying. Copying blocks into specially allocated buffers before performing a naive inner kernel seems to improve performance by allowing the CPU and compiler to optimize memory accesses to the buffers.

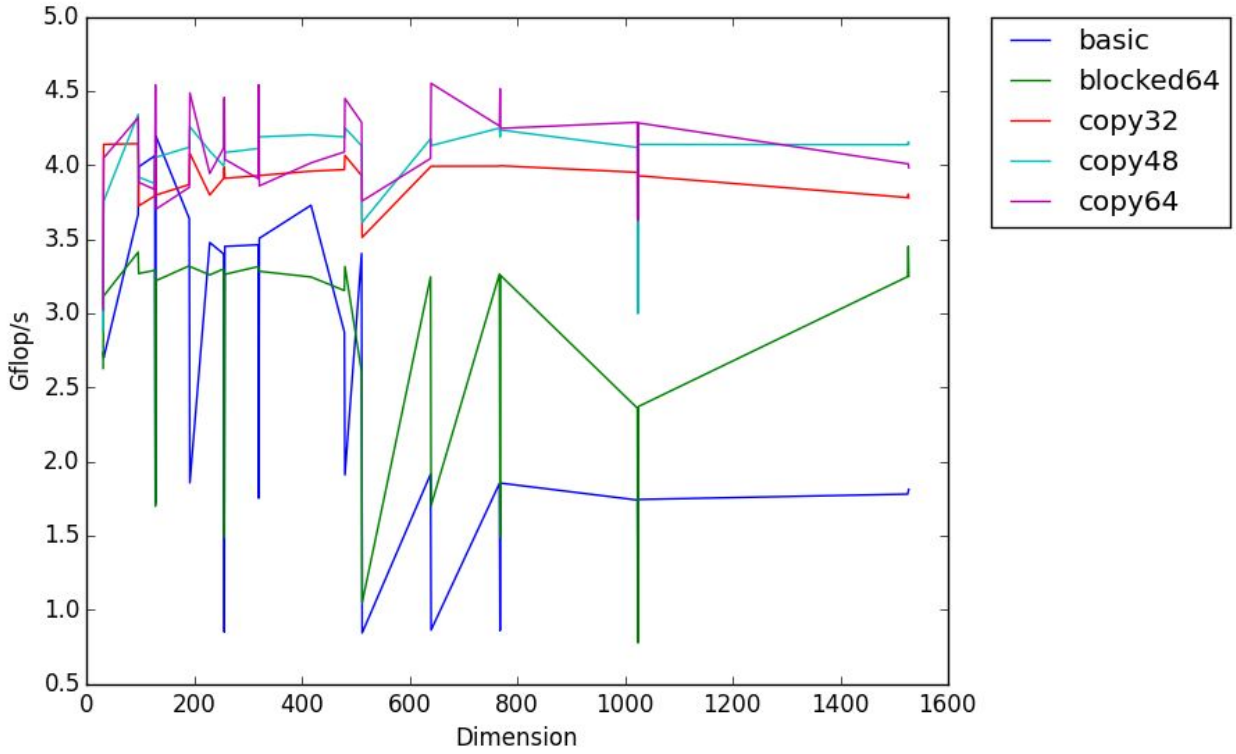

Figure 5: Timing of basic versus 64-size blocked versus various copy optimized block sizes

Memory Alignment

Memory alignment is important for the compiler because if memory is address aligned properly then the compiler can take advantage of that to have optimized memory accesses which can boost memory performance significantly and help increase memory efficiency which improves overall speed when bottlenecked by memory bandwidth.

Memory can be specifically aligned when being allocated using special attributes or intrinsics.[3] Static arrays can be aligned along memory boundaries using `__attribute__((aligned(64)))` while dynamically allocated arrays can be memory aligned using the `_mm_alloc` and `_mm_free` instructions. The compiler can then be told that these dynamic arrays are aligned using `__assume_aligned(a, 64)` before loops which access aligned array *a* - the 64 indicates that a is aligned along a 64-byte boundary.

Because we do not control whether the matrices A, B, and C are aligned before they are passed to `square_dgemm` this optimization only makes sense in the context of the previously discussed copy optimization.

[3] https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

Results

In Figure 6 we see that aligning along 16- or 64-byte the special copy buffers increases performance further beyond simple copy optimization. In particular performance was higher for 2 level blocking with the smaller block being copied into memory aligned buffers which is interesting as previously having multiple levels did not provide a performance boost for standard multi-level blocking without copy optimization. It is also important to note that non-multiple of 8 buffer dimension has much lower performance, this is likely because the memory alignment helps optimize for buffers which are also a multiple of 8.

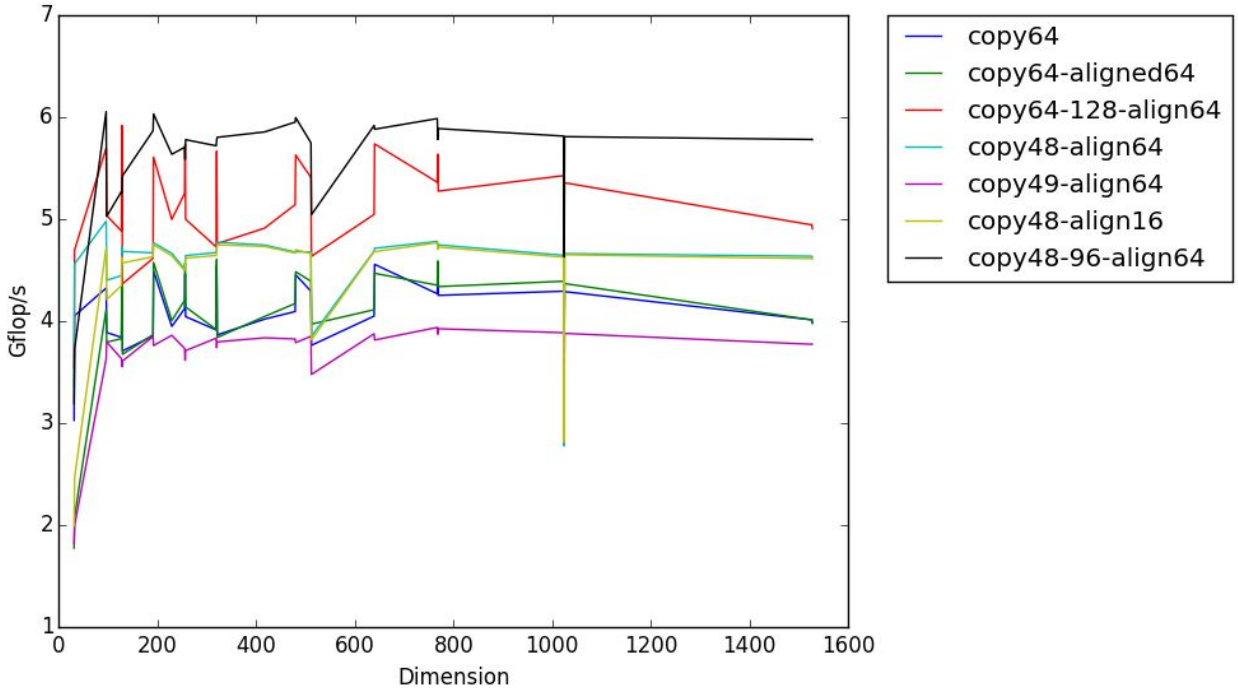

Figure 6: Standard copy optimization blocking versus aligned copy optimization

Matrix Transposition

Naive matrix multiplication accesses a column of matrix B and a row of matrix A at a time when computing a single entry for the result matrix C. If both matrix A and matrix B are stored in the same format (row- or column-major) then one matrix will be accessed using unit stride while the other will be accessed using stride equal to the dimension of the matrix when computing a single entry of matrix C. This non-unit stride can result in poor cache access patterns. For instance the stride might be such that for each cache line that is read in only the first

entry is accessed and then the next access is at the start of the next line, etc. This results in a significant number of cache misses and a major cost to performance.

This stride problem can be alleviated by storing each matrix in alternate formats, one in row-major and one in column-major. Because both matrices start out stored in column-major format, then matrix B is accessed with unit stride will matrix A is accessed with dimension stride. By converting matrix A into a row-major format it can then be accessed with unit stride when calculating a single entry of C. This format conversion can also be thought of as transposing matrix A and then taking dot products of columns of A and B rather than doing standard matrix multiplication.

Using this new format for matrix A, both matrices can be accessed using unit stride which improves cache performance and also allows the compiler to make better using of the sequential memory and hopefully use faster vector instructions. Of course this transposition comes at a cost: to transpose the matrix must be completely re-written coming at a cost of $O(N^2)$. However, this $O(N^2)$ may be worth it if the cache and vector instruction improvements save more time. In particular this technique makes the most sense when copy optimization is used. As copy optimization already copies parts of A and B to special arrays in which case there is no cost as part of A must be copied anyways.

Results

Transposing copied buffers provides a significant performance boost over non-transposed buffers. In Figure 7 we see that a 64 by 64 copied buffer which is 64-byte aligned is significantly outperformed by various versions which transpose the buffer for matrix A. In particular the 64 x 64 copied buffer aligned along 64-byte boundaries and transposed outperforms the high performing two level blocking with the inner 48 x 48 block being copied and aligned (red line).

Figure 7 also shows that the highest performing version was a two-level blocking DGEMM where the smaller block of size 64 x 64 is copied into a 64-byte aligned buffer and transposed. Transposition in general seemed to increase performance largely because it allows unit stride along both A and B which improves cache performance and vectorization. Further it is better to have unit stride for A and B as they are used more intensively and not written to, unlike C.

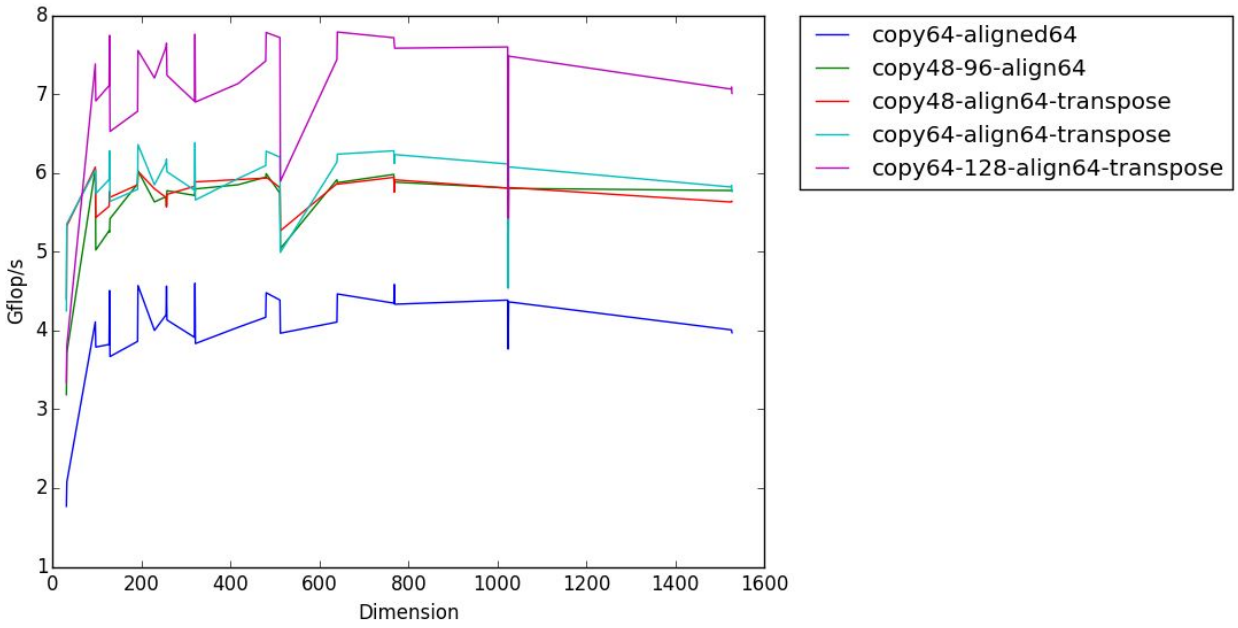

Figure 7: Normal aligned copy optimization versus transposed aligned copy optimization

Vector Instructions

Though the compiler will often vectorize loops when it can, it may not do so as effectively as having the code explicitly use the instructions. In particular if the compiler recognizes dependencies between loops it may not vectorize in a way that could be done if vector instructions could be used explicitly.

The E5 2620 v3's we are working with support both AVX and SSE instructions which can be used to compute vector operations such as additions and multiplications. For instance the naive matrix multiplication with an assumed row-order matrix A and an assumed column-order matrix B can be written as:

```
int i, j, k
for (i = 0; i < M; ++i) {
    for (j = 0; j < M; ++j) {
        double cij = C[j*M+i];
        for (k = 0; k < M; ++k)
            cij += A[i*M+k] * B[j*M+k];
        C[j*M+i] = cij;
    }
}
```

This naive matrix multiplication can be loop unrolled and re-written using explicit vector instructions to become (assuming A, B are aligned):

```
int i, j, k
double temp[4] __attribute__((aligned(64)));
for (i = 0; i < M; ++i) {
    for (j = 0; j < M; ++j) {
        double cij = C[j*M+i];
        __m256d ymm2 = _mm256_setzero_pd();
        for (k = 0; k < M/4*4; k+=4)
            __m256d ymm0 = _mm256_load_pd(A+i*M+k);
            __m256d ymm1 = _mm256_load_pd(B+j*M+k);
            __m256d ymm3 = _mm256_mul_pd(ymm0, ymm1);
            ymm2 = _mm256_add_pd(ymm2, ymm3);
        for (k = M/4*4; k < M; ++k)
            cij += A[i*M+k] * B[j*M+k];
        _mm256_store_pd(temp, ymm2);
        cij += temp[0] + temp[1] + temp[2] + temp[3];
        C[j*M+i] = cij;
    }
}
```

The vectorization and unrolling of the inner for loop may produce benefits beyond what the compiler could do for the original naive code.

Results

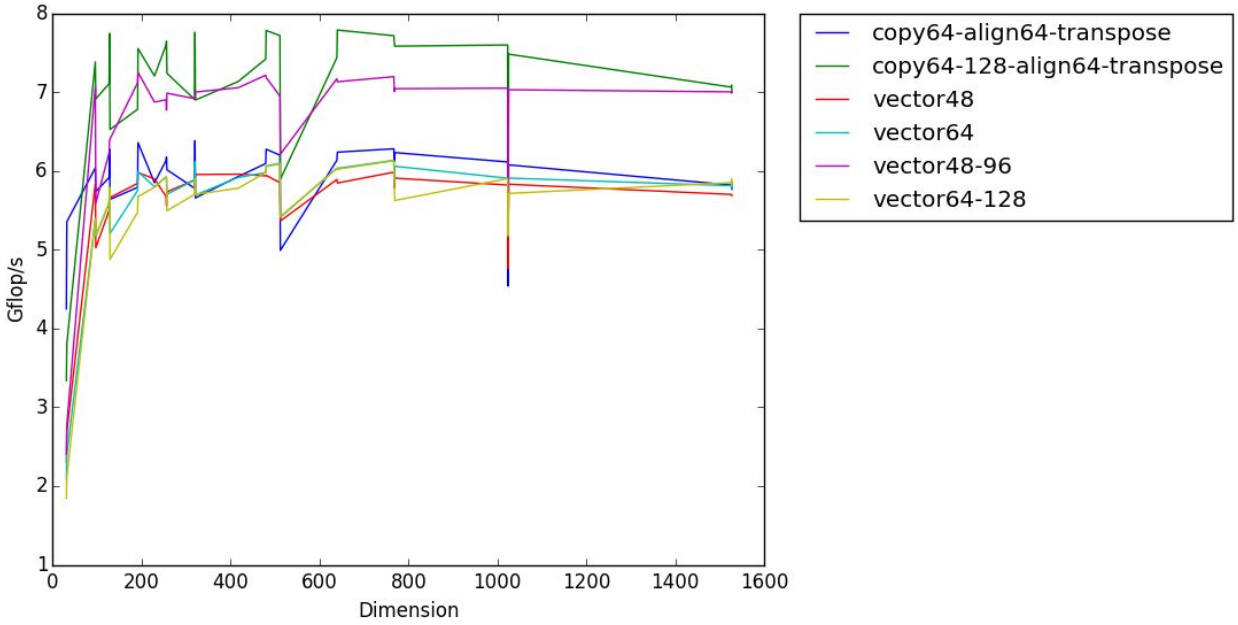

Figure 8: transposed aligned copy optimization versus the vectorized versions

Figure 8 Shows the results of using explicit vector intrinsics versus using a naive kernel to perform the inner block matrix multiplication. As can be seen in Figure 8 the performance of the explicit vector intrinsics (implemented exactly as described in the earlier code section) *reduces* performance over the best naive kernel from the previous section. Note that vectorX

DGEMMs use memory aligned copy buffers, (multi-level) blocking, and transposition from the previous sections. The reduction in performance when using explicit vector intrinsics ourselves is likely due to the compiler being able to better vectorize the inner kernel than we are able to do by hand. This means that when we added the vector instructions it interfered with the compiler's ability to vectorize the loops in our inner kernel, thus reducing performance.

Optimization Flags and Compiler Hints

Though the compiler will do certain things by default, especially when the optimization flag (-O) is set to 2 or 3, there are many other options that can be turned on. By turning on certain options that compiler may be able to take much better advantage of the code and add vectorization, improved memory accesses and other optimizations if it is told that it should try to do these optimizations. There are many possible flags that could be potentially useful for the ICC compiler to be given. Intel provides a useful guide for how to optimize using the compiler.[4] The `restrict` keyword can also be useful for indicating to the compiler that the arrays are not overlapping.

Results

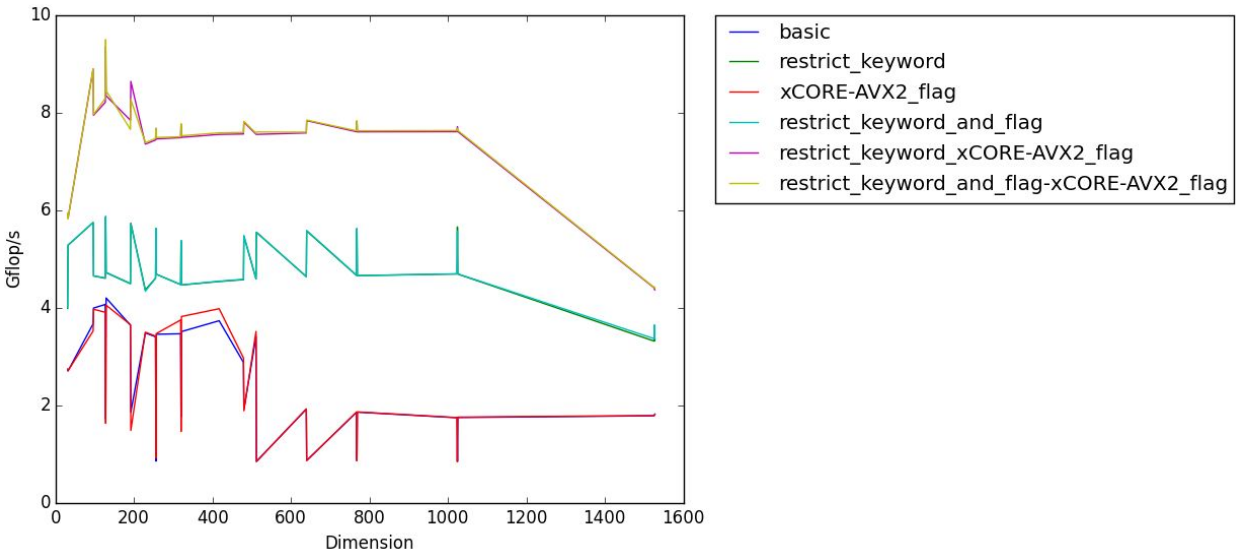

Figure 9: Basic DGEMM with various combinations of the restrict keyword, -restrict flag, and -xCORE-AVX2 flag

Adding the restrict keyword alone or -xCORE-AVX2 flag did not result in higher performance as the restrict keyword is ignored unless the proper switches are used and the AVX flag doesn't seem to help vectorization without the restrict keyword (Figure 9). By adding both

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

the restrict keyword, and the -restrict flag, restricted arrays were no longer ignored and the compiler could make many more optimizations. Adding the -xCORE-AVX2 flag with the restrict keyword (optionally adding the -restrict flag) allowed the basic DGEMM to achieve incredibly high performance. This increased performance is probably due to the usage of the restrict keyword allowing increased parallelization (when not ignored) through vectorization of the DGEMM. Using the -xCORE-AVX2 flag in conjunction with the restrict keyword allows the compiler to not only vectorize, but to vectorize using the best vectorization intrinsics supported by the Xeon E5 2620 v3s in our cluster furthering performance.

Putting it all together

To build up our final optimized matrix multiplication DGEMM we worked off of what we had built up for for (multi-level) blocking, copy optimization, memory alignment, and transposition. We tried various block sizes and number of blocks and added the flags and restrict keyword that we tried in the previous section. The results in Figure 10, show that by adding the flag to a single level copy optimized block of size 64x64 (or 256x256) which is memory aligned and which transposes buffer A can achieve maximum speeds over 9GFlops while having very consistent performance across matrix dimension sizes. We also tried taking a single level copy optimized block of size 64x64 with buffer A transposed using explicit vector instructions plus the flags and restrict keyword from the previous section. Finally we tried a different loop ordering which worked best in the loop ordering tests we did on the basic DGEMM.

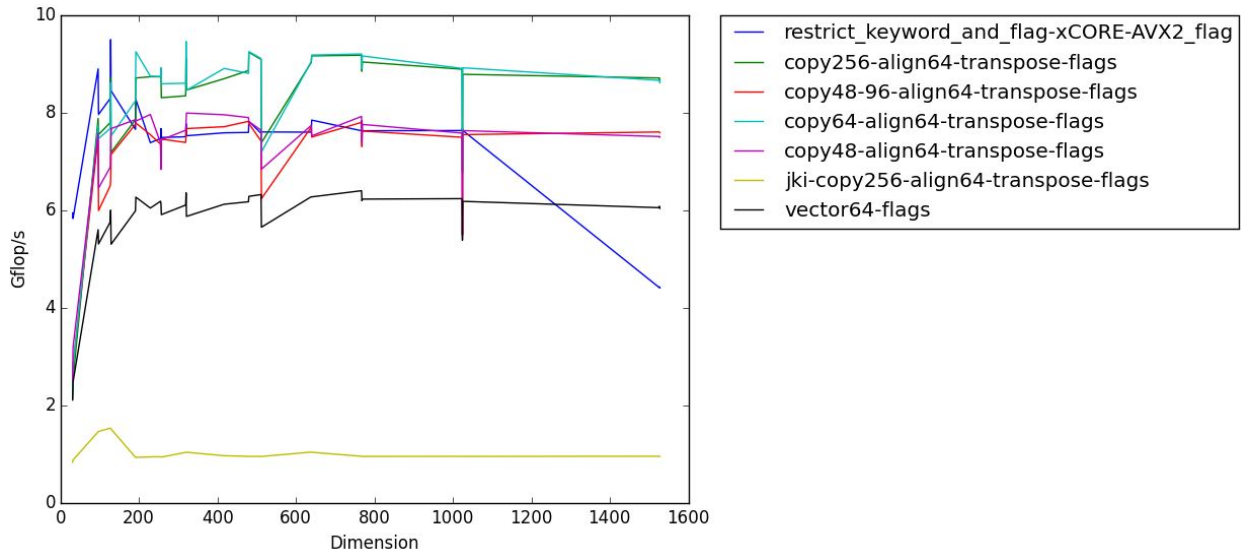

Figure 10: Various combinations of optimizations, flags refers to the restrict keyword, -restrict flag and -xCORE-AVX2 flags being used.

In figure 10 we can see that changing the loop ordering to JKI for a single-level blocked copy optimized, aligned DGEMM which transposes buffer A does not perform well as the loop ordering removes the unit stride in the inner loop which had previously allowed for lots of vectorization. The explicit use of vector instructions also did not reach the levels of the other DGEMMs, even with the flags that we found successful added as our own vectorization was simply not as good as the compilers.

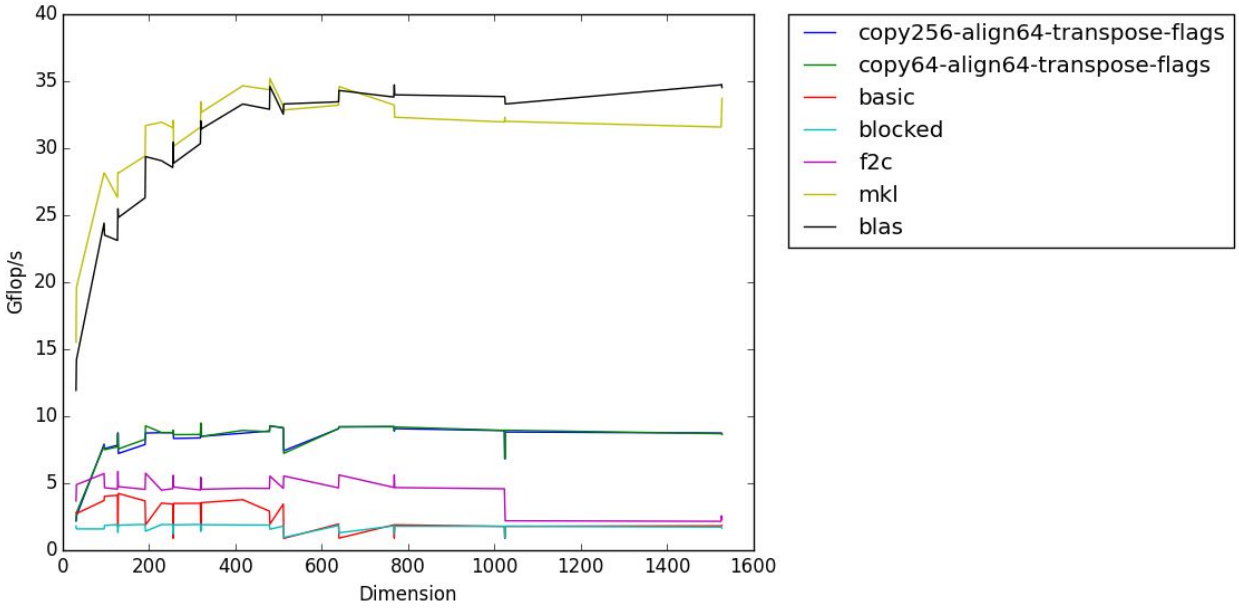

Figure 11: The best two DGEMMs from Figure 10 as compared to the baseline DGEMMs

In Figure 11 we show our best two DGEMMs from Figure 10 and compare it with the baseline algorithms. Our DGEMMs perform well, just under 10GFlops and sustain that performance across most matrix sizes. Both of our DGEMMs are easily outperformed by MKL and BLAS, but do achieve more than 25% performance of MKL and BLAS. Both of our best DGEMMS outperform all the other baseline algorithms by a wide margin.

After our development and tuning that resulted in these two high-performance DGEMMs we tried adding a few more flags as suggested by Intel.[5] In particular we tried the -no-prec-div and -ipo flags in conjunction with our other flags and restrict keyword that we found to work before. Adding -ipo did not seem to increase the performance of our DGEMMs, but adding -no-prec-div added approximately another 2GFlop/s to our peak performance.

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

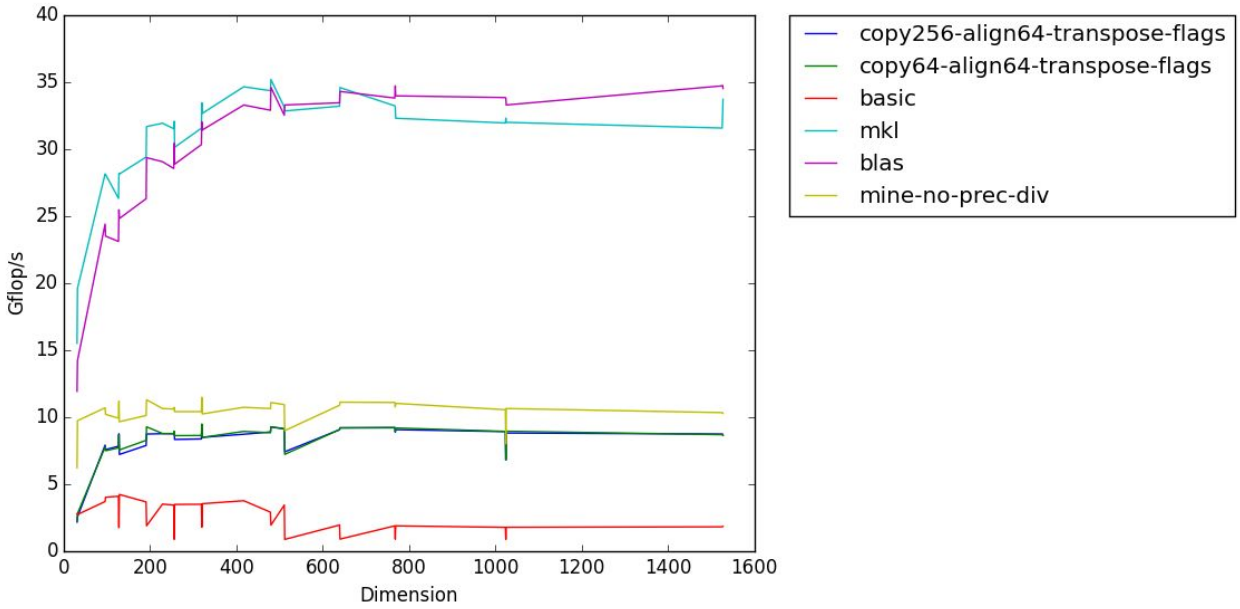

Figure 12: Figure 11 plus the size 64 copy aligned transposed buffer with the previous flags and -no-prec-div

In Figure 12 we show this new DGEMM (mine-no-prec-div) which is a single-level blocking DGEMM with a size 64x64 block which are copied into 64-byte memory aligned buffers and the buffer for A is transposed. The DGEMM uses the restrict keyword for all array arguments, the -xCORE-AVX2, -restrict, and -no-prec-div flags as well as -O3. This DGEMM achieved a peak performance of 11.2638 GFlop/s (matrix size 192), which is approximately 33% of the performance of BLAS/MKL, and nearly 2GFlop/s more than our previous best DGEMMs.

Conclusion

In our optimization of matrix multiplication we tried many different methods: loop ordering, loop unrolling, blocking, copy optimization, memory alignment, matrix transposition, vector instructions, and compiler optimization. We also tuned our DGEMMs parameters, primarily by trying multiple levels of blocking and varying block sizes.

We found that all of these methods provide some measure of improvement, though do so to varying levels and do not necessarily all work in concert together. In particular we found that loop unrolling, vector instructions and loop ordering in some cases may have been useful, were not useful when used in conjunction with most of the other optimizations and often detracted from the compiler's ability to optimize our code.

In the end we found that using a single-level of blocking and copying the blocks into buffers for matrices A and B was most effective. When copying the blocks into buffers, we forced the buffers to be memory aligned along 64-byte boundaries and transposing the contents

of A as they were copied into A's buffer was most effective. Blocks were then computed using a naive inner kernel which used an IJK ordering which provided unit stride to both the A buffer and B buffer at the innermost level. This allow the compiler to vectorize many of our loops, including the inner, most important level.

We tried various flags and found that the -restrict, -xCORE-AVX2 and -no-prec-div flags were most effective, especially in concert with the restrict keyword. We found that a block size of 64 was best. At the end we achieved a peak performance of 11.3GFlop/s, approximately 33% the performance of MKL/BLAS.