CS 5220 HW1

bc496, djf244, lmh95

September 2015

1 Introduction

We sought to test the effects of different optimization techniques on the performance of a square double-precision matrix multiplication (or "DGEMM"). Specifically, the DGEMM takes in matrices $A, B, C \in \mathbb{R}^{n \times n}$ and performs the operation,

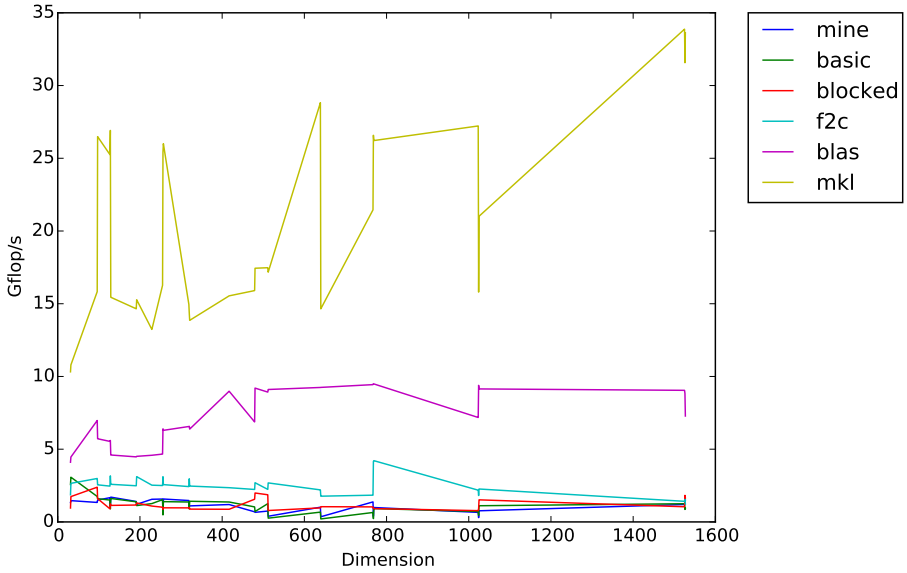
$$C \leftarrow C + AB$$

We hope to eventually use the insight gained from these experiments to combine the best-performing optimizations into a matrix multiplication implementation whose performance is at least of the same order of magnitude as MKL. The baseline implementations with which we compared our optimizations are a naive matrix multiplication, a blocked matrix multiplication, a FORTRAN implementation of a naive matrix multiplication, the BLAS implementation, and the MKL implementation. The performance of these baselines is plotted in Figure 1.

2 Loop Reordering

We experimented with the effects of loop reordering on the naive matrix multiplication implementation given in `dgemm_basic.c` to test its effect on performance. The hope here was that different orderings would improve performance by reducing the number of cache misses. The results of this experiment are given in Figure 2. Here a label such as "jki" refers to the order in which we iterate over the indices in the expression $C_{ij} = \sum_k A_{ik} B_{kj}$. It is immediately clear from this plot that selecting the optimal loop order for the problem can improve the performance as much as threefold over the default implementation depending on the input size. In particular the "jki" ordering achieves the best performance of all orderings across all input sizes. We hypothesize that this is because this ordering iterates over both C and A in column-major order and is thus well-suited to our memory layout. The downside of this loop order is that it iterates over B in row-major order, but the current B element being accessed only changes once for every column of A that we iterate over, and thus the number of cache misses due to this factor is relatively low.

Figure 1: Performance for baseline implementations.



3 Blocking

To understand the effects of block size on the performance of blocking implementations find a high-performing block size for our implementation, we began with the basic blocking implementation given in `dgemm_blocked.c` and timed its performance across a range of block sizes. The results are given in Figure 3.

It is clear that the block sizes at the low end of the range we tried (e.g. 16) generally achieve the worst performance, but the medium and large block sizes beat each other in different input dimension regimes. We suspected that this was due to the interaction between block size and cache misses due to loop ordering and matrix layout. In particular, the large block sizes such as 513 perform up to dimension 1000, at which point they drop off. We suspect that this is because cache misses due to matrix layout within each block multiplication become significant as blocks become large. To test this theory, we experimented with different block sizes using the best-performing loop ordering, "jki". We used the "jki" ordering within each block multiplication. We experimented with different orders for iteration over the blocks themselves but found that this yielded little improvement and ultimately used the "jki" ordering for this iteration as well. These results are located in Figure 4.

We can see from Figure 4 that the best-performing block size with the "jki" ordering is 513. We believe that for smaller block sizes, the overhead required by blocking negates the benefits due to blocking. The performance of the 513 block size is very similar to the performance of the "jki" ordering without blocking as

Figure 2: Performance for alternate loop orderings.

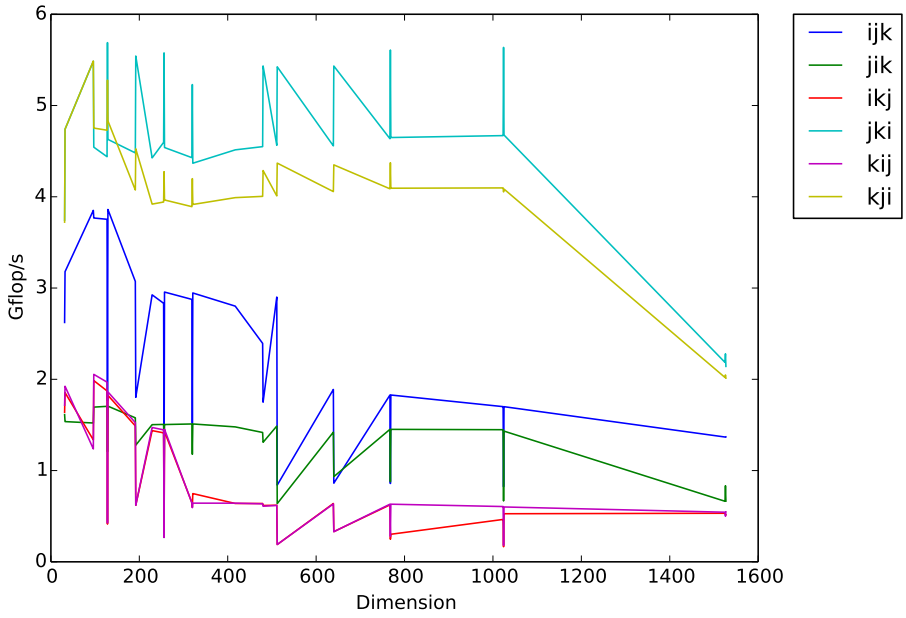


Figure 3: Performance of different block sizes with standard loop order.

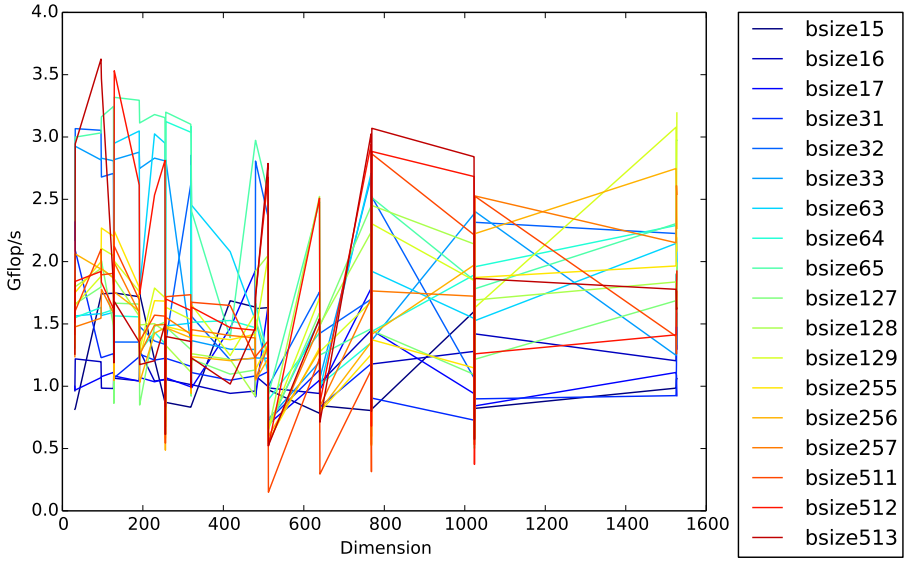
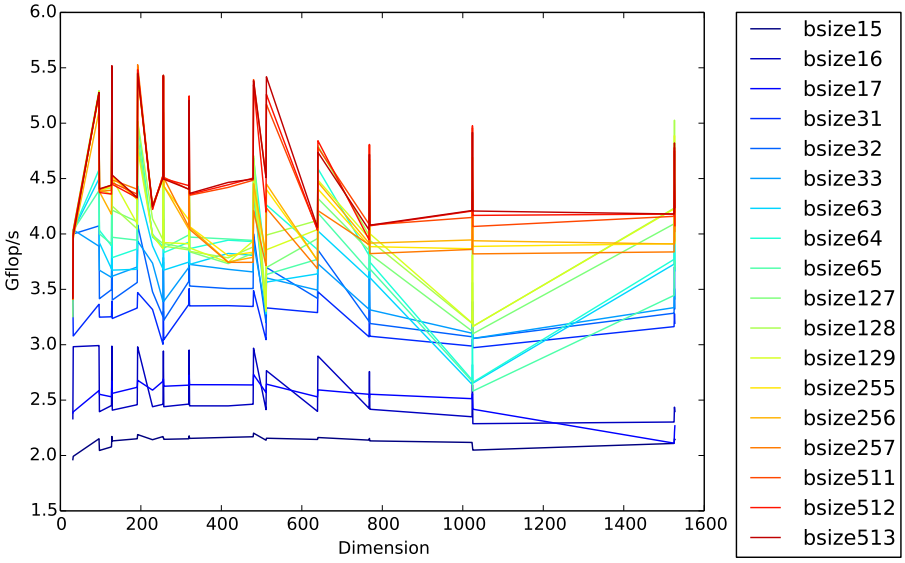


Figure 4: Performance of different block sizes with "jki" loop order.



in Figure 2, but is strictly better when the input matrix is large. We suspect that this is because the blocks are large enough for the overhead in blocking to be relatively small, and that they improve cache performance.

4 AVX and Memory Alignment

Use Intel Intrinsics, we can do multiple instructions at one time. As you can see from http://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2_40-GHz, the cluster supports AVX2.0 and FMA. Since the Totient machine only supports 256 bits each instruction, so every time we can do 4 double precision operations per time. The code to use jki loop order and AVX is shown in Listing1 in the appendix.

When you are using jki order to calculate C = A * B, basically you are selecting one column of A, denote it as a, multiply it by one number from B, denote it as b_0 and added back to the corresponding column of C, denote it as c. The formula then is $c = c + a * b_0 + c$. Since we can do four instructions per time, so in line 20 and line 21 of Listing 1, we first read in 4 doubles from A and C respectively and store then in ymm1 and ymm2. In order to use the AVX, we then use _mm256_broadcast_sd to broadcast b_0 in to a vector in line 18, which has length 4. In line 22, we use the _mm256_mul_pd to do the multiplication of a and b_0 and then in line 23, we add $a * b_0$ with c and store it back to c, which concludes our computation. In line 36-37, we free the memory which was allocated at the beginning. The detailed c code for this implementation is in

Listing 1.

To further speed up our code, we did the loop unrolling. The idea behind it is to consider the latency and throughput of each instructions[1], as you can find here https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX,AVX2&text=add&expand=100 for each instructions. With the loop unrolling, the code is shown in Listing 2 in the appendix. Here we define one additional variable called _col_reduced_16, it counts how many length 16 vectors are contained in one column of A. And now the inner for loop will again do the calculation for $c = a * b_0 + c$, but now both a and c will be a length 16 vectors. This will speed up the code almost twice.

Use aligned memory will further improve the speed of AVX. With the alignment memory, you can then use _mm256_store_pd and _mm256_load_pd instead of _mm256_storeu_pd and _mm256_loadu_pd, which will speed up the loading and storing. However, for each memory alignment, you also need to copy the original vector to the new vector and there is a trade off. The results suggest that we shouldn't do memory alignment for all A, B and C. Thus we only did the memory alignment for B and C, but not for A. See code line 9-10, 13-18 for details of Listing 1 for details.

Our last attempt with vectorization is to try FMA. With FMA, it can do $a * b_0 + c$ in one operation instead of first doing the multiplication and then followed by the addition. The detailed code is in Listing 3. See line 42-45 for details. This improves the speed of our code by 1 Gflop/s.

5 The `restrict` Keyword and Aliasing

We investigated using the `restrict` pointer qualifier [3] to improve compiler optimizations. Consider the following restricted pointer declaration:

```
double * restrict A;
```

This informs the compiler that no pointers reference the same memory location as A. To see how this can be useful, consider the `square_dgemm` function declaration:

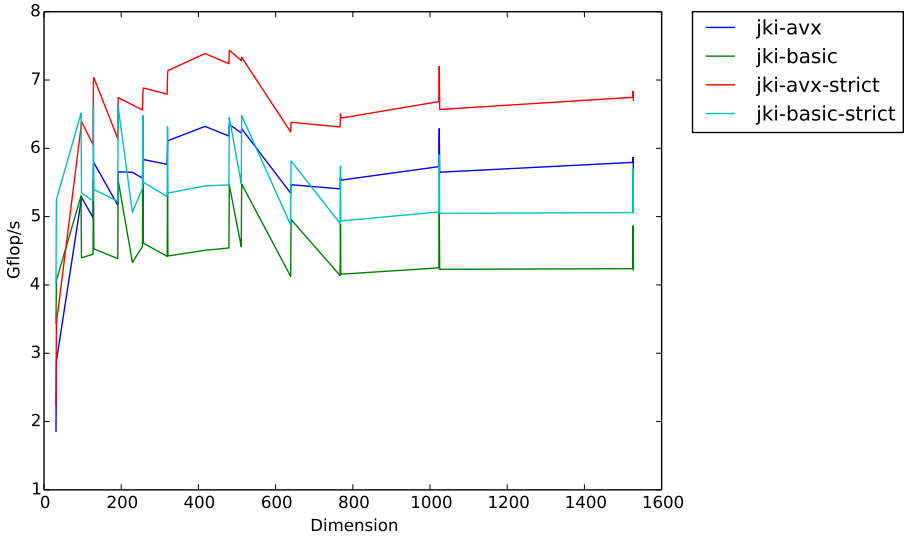
```
void square_dgemm(const int M, const double *A,
                  const double *B, double *C);
```

As far as the compiler knows, A, B, and C refer to the same location in memory. This means that every time we write to an element of C, we must reload A and B into the processor's registers. Since we know that C is not equal to A or B (the basic implementation would be incorrect if this were true), these load instructions are frivolous, and we can use the restrict keyword to eliminate them.

This sounds nice in theory, but we found that simply adding the `restrict` keyword to our `dgemm_X.c` files had no effect on performance. Adding `restrict` to the `matmul.c` file in addition to the DGEMM files had a very positive effect:

Performance was boosted by one GFlop/s uniformly across all of our implementations. This was informative but not acceptable for a final implementation since the DGEMM implementation should be self-contained. We sidestepped this issue by opting not to use the `restrict` keyword and instead using the `-fstrict-aliasing` compiler flag, which enables ANSI aliasing rules for optimization [2]. The performance boost from this flag was identical to the boost we achieved by uniformly applying the `restrict` keyword throughout our code, but required no code modifications. This test is plotted in Figure 5. Here "jki-basic" refers to our implementation with the best block size and loop order and "jki-avx" refers to the same code with a vectorized kernel. We see that adding the `-fstrict-aliasing` flag adds an extra gigaflop of performance for both implementations. For this reason we decided to include the `-fstrict-aliasing` flag in our final implementation.

Figure 5: Performance of different implementations with `-fstrict-aliasing` keyword.

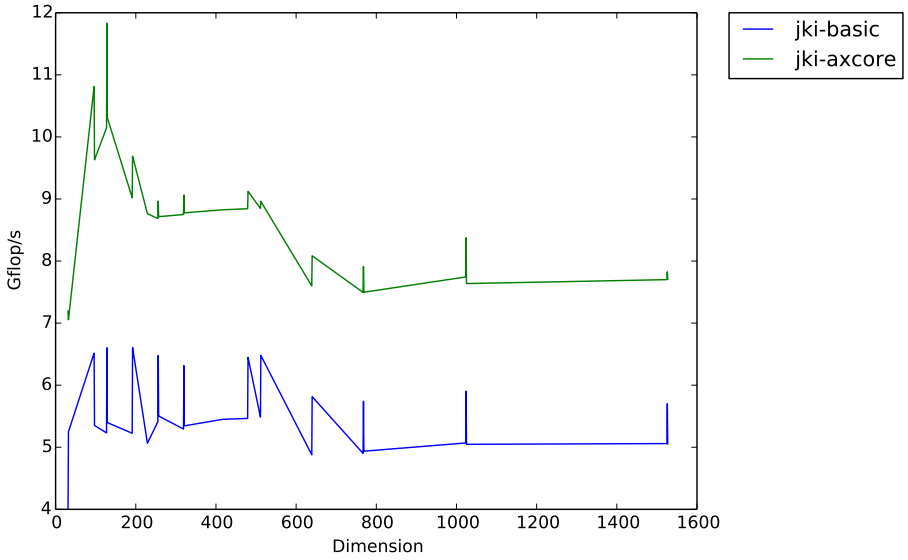


6 Automatic Vectorization

Instead of manually vectorizing our code using Intel intrinsics, we can have ICC attempt to vectorize with AVX2 instructions using the `-axCORE-AVX2` compiler flag [2]. We found that this can yield a massive performance boost for some DGEMM implementations. We experimented with this flag with the jki ordering with a block size of 512, with and without AVX intrinsics. Without intrinsics, the flag almost doubles performance, as shown in Figure 6. With

the manually-vectorized code, the flag makes no difference, which is depicted in Figure 7. This is not surprising, as the manually vectorized code is much more complicated than the basic blocked code, and thus more difficult for the compiler to optimize. It's also not clear what the compiler would change here, since the inner loop is already unrolled and vectorized. What's interesting is that the automatically-vectorized code basic implementation performs better than the manually-vectorized code. It seems that the Intel compiler has a better idea of how to optimize the code than we do.

Figure 6: Performance of basic blocked implementation with `-axCORE-AVX2` flag.

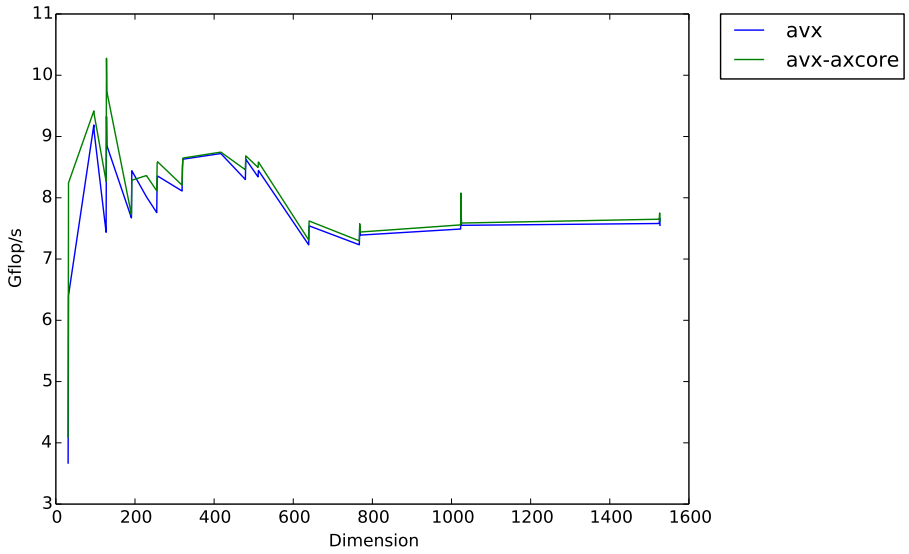


7 Compiler Flags

I tested the `-fast`, `-fstrict-aliasing`, `-axCore-AVX2 -qopt-prefetch=4`, and `-align` flags. I tested every combination of these flags on our two best dgemm's, one with manual AVX and one without manual AVX. The results are in the table in Figure 8.

Now that we have a significantly better implementation than before, the compiler flags don't make as much of a difference in terms of of actual time, but percentage-wise they make a similar impact. One of the interesting results was that we needed a combination of flags to make much of a difference - the right flag took the runtime of the non-avx implementation down to (and even lower than) the runtime of the avx implementation, but other than that, there was no flag that made a significant difference on its own. In the end, we decided to use the `-fast`, `-axCore-AVX2`, and `-fstrict-aliasing` with our non-avx

Figure 7: Performance of blocked, manually-vectorized implementation with `-axCORE-AVX2` flag.



implementation. The graph of Dimension vs. Gflops/s with these flags are is labeled as "jki" in Figure 9. We chose this combination of flags due to both the runtime and the fact that we got the best Gflops/s with these flags.

8 Matrix Transposition

We experimented with transposing our matrices in memory before performing the matrix multiplication. The reasoning behind this optimization is simple: The default (ijk) loop ordering iterates across the rows of A and C, but A and C are stored in column-major form. By switching these matrices to be row-major, we gain unit stride. Our implementation allocates new buffers for A and C, copies A and C in in row-major form, and then accesses them in row-major for subsequent operations. The performance of the ijk ordering with transposition ("copy") and the non-transposed jki ordering ("jki") with our best compiler flag combination is plotted in Figure 9. Ultimately the transposed ijk ordering yielded no improvement over the non-transposed jki loop ordering. The reason behind this phenomenon becomes clear when one thinks about the behavior of the jki ordering: This ordering loops over the columns of all three matrices and thus has unit stride, but this is achieved without the overhead of transposition. As the input size increases, the transposition overhead contributes a negligible amount to the performance and the jki ordering's performance approaches the transposed ijk performance.

Figure 8: Performance Given Different Combinations of Flags

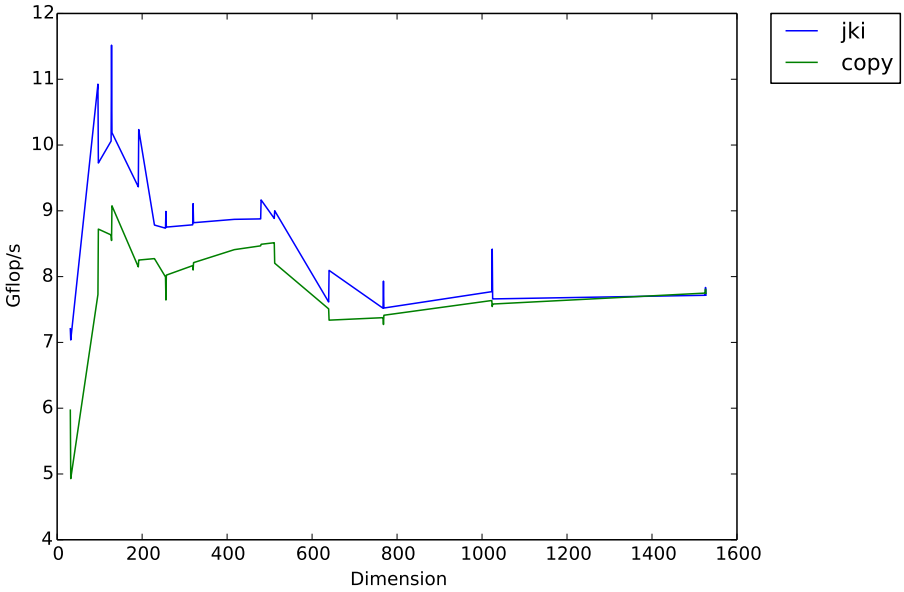
	non-avx	avx
default flags	00:01:16	00:01:02
`-fast`	00:01:15	00:01:02
`-fstrict-aliasing`	00:01:19	00:01:02
`-axCore-AVX2`	00:01:01	00:01:02
`-qopt-prefetch=4`	00:01:15	00:01:02
`-align`	00:01:15	00:01:02
`-fast` & `aliasing`	00:01:15	00:01:02
`-fast` & `AVX2`	00:01:02	00:01:01
`-fast` & `-qopt`	00:01:14	00:01:11
`-fast` & `-align`	00:01:15	00:01:02
`aliasing` & `AVX2`	00:01:02	00:01:02
`aliasing` & `-qopt`	00:01:15	00:01:02
`aliasing` & `-align`	00:00:57	00:00:57
`AVX2` & `-qopt`	00:01:15	00:01:02
`AVX2` & `-align`	00:00:58	00:00:57
`-qopt` & `-align`	00:01:05	00:00:56
`-fast` & `aliasing` & `AVX2`	00:00:57	00:00:57
`-fast` & `aliasing` & `-qopt`	00:01:14	00:01:15
`-fast` & `aliasing` & `-align`	00:01:04	00:00:56
`-fast` & `AVX` & `-qopt`	00:01:00	00:01:02
`-fast` & `AVX` & `-align`	00:00:57	00:00:56
`-fast` & `-qopt` & `-align`	00:01:03	00:01:03
`aliasing` & `AVX2` & `-qopt`	00:00:57	00:00:55
`aliasing` & `AVX2` & `-align`	00:01:01	00:00:55
`aliasing` & `-qopt` & `-align`	00:01:05	00:00:56
`AVX2` & `-qopt` & `-align`	00:00:57	00:00:55
not `-fast`	00:00:57	00:00:56
not `-fstrict-aliasing`	00:01:00	00:01:02
not `-axCore-AVX2`	00:01:03	00:01:03
not `-qopt-prefetch=4`	00:00:57	00:00:56
not `-align`	00:01:00	00:01:03
all flags	00:01:00	00:01:03

Ultimately we opted to leave transposition out of our final implementation and used the jki loop ordering in its place.

9 Conclusion

Our best-performing code is labeled as "jki" in Figure 9. This is a basic blocked implementation with the jki loop ordering with the `-fstrict-aliasing`, `-fast`, and `-axCORE-AVX2` compiler flags. Our approach of focusing on simple high-level

Figure 9: Performance of transposed ijk and non-transposed jki orderings.



optimizations and using carefully-selected compiler flags seems to have paid off, as the compiler does an excellent job of vectorizing our code, but our loop ordering and blocking optimization play a fundamental role in obtaining the performance on the plot. We're also happy with our best attempt at manually vectorizing the code, which is plotted in Figure 7. This attempt uses the same set of compiler flags, but features vectorization that we implemented ourselves using Intel intrinsics along with optimizations such as fused multiply-add and loop unrolling. This code did not perform quite as well as the simple automatically-vectorized code, but got close. Given more time, we would have experimented more with memory alignment and with multiple layers of blocking.

References

[1] https://thinkingandcomputing.com/2014/02/28/using-avx-instructions-in-matrix-multiplication/

[2] http://geco.mines.edu/guide/icc.html

[3] http://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.pdf

Appendix

Listing 1: Basic AVX Implementation

```
1  void basic_dgemm(const int lda, const int M, const int N, const int K,
2          const double *A, const double *B, double *C) {
3      int i, j, k;
4      const int col_reduced_4 = M - M % 4;
5      double *c = _mm_malloc(M * sizeof(double), 64);
6      double *b = _mm_malloc(K * sizeof(double), 64);
7      __m256d ymm0, ymm1, ymm2, ymm3;
8      for (j = 0; j < N; ++j) {
9          const int index1 = j * lda;
10         for (k = 0; k < K; ++k) {
11             b[k] = B[index1 + k];
12             }
13         for (i = 0; i < M; i++) {
14             c[i] = C[index1 + i];
15             }
16         for (k = 0; k < K; ++k) {
17             const int index2 = k * lda;
18             ymm0 = _mm256_broadcast_sd(&b[k]);
19             for (i = 0; i < col_reduced_4; i += 4) {
20                 ymm1 = _mm256_loadu_pd(&A[index2 + i]);
21                 ymm2 = _mm256_load_pd(&c[i]);
22                 ymm3 = _mm256_mul_pd(ymm1, ymm0);
23                 _mm256_store_pd(&c[i], _mm256_add_pd(ymm2, ymm3));
24             }
25             for (i = col_reduced_4; i < M; i++) {
26                 c[i] += A[index2 + i] * b[k];
27             }
28         }
29         for (i = 0; i < col_reduced_4; i += 4) {
30             _mm256_storeu_pd(&C[index1 + i], _mm256_loadu_pd(&c[i]));
31             }
32         for (i = col_reduced_4; i < M; i++) {
33             C[index1 + i] = c[i];}
34             }
35        }
36         _mm_free(b);
37         _mm_free(c);
38 }
```

Listing 2: AVX implementation with loop unrolling

```
1  void basic_dgemm(const int lda, const int M, const int N, const int K,
2          const double *A, const double *B, double *C)
3  {
4      int i, j, k;
5      const int col_reduced_4 = M - M % 4;
6      const int col_reduced_16 = M - M % 16;
7      __m256d ymm0, ymm1, ymm2, ymm3, ymm4, ymm5, ymm6, ymm7, ymm8, ymm9,
8              ymm10, ymm11, ymm12;
9      double *b = _mm_malloc(K * sizeof(double), 64);
10     double *c = _mm_malloc(M * sizeof(double), 64);
11     for (j = 0; j < N; ++j) {
12         const int index1 = j * lda;
13         for (k = 0; k < K; ++k) {
14             b[k] = B[index1 + k];}
15         for (i = 0; i < M; i++) {
16             c[i] = C[index1 + i];}
17         for (k = 0; k < K; ++k) {
18             const int index2 = k * lda;
19             ymm0 = _mm256_broadcast_sd(&b[k]);
20             for (i = 0; i < col_reduced_16; i += 16) {
21                 ymm1 = _mm256_loadu_pd(&A[index2 + i]);
22                 ymm2 = _mm256_loadu_pd(&A[index2 + i + 4]);
23                 ymm3 = _mm256_loadu_pd(&A[index2 + i + 8]);
24                 ymm4 = _mm256_loadu_pd(&A[index2 + i + 12]);
25                 ymm5 = _mm256_load_pd(&c[i]);
26                 ymm6 = _mm256_load_pd(&c[i + 4]);
27                 ymm7 = _mm256_load_pd(&c[i + 8]);
28                 ymm8 = _mm256_load_pd(&c[i + 12]);
29                 ymm9 = _mm256_mul_pd(ymm1, ymm0);
30                 ymm10 = _mm256_mul_pd(ymm2, ymm0);
31                 ymm11 = _mm256_mul_pd(ymm3, ymm0);
32                 ymm12 = _mm256_mul_pd(ymm4, ymm0);
33                 _mm256_store_pd(&c[i], _mm256_add_pd(ymm5, ymm9));
34                 _mm256_store_pd(&c[i + 4], _mm256_add_pd(ymm6, ymm10));
35                 _mm256_store_pd(&c[i + 8], _mm256_add_pd(ymm7, ymm11));
36                 _mm256_store_pd(&c[i + 12], _mm256_add_pd(ymm8, ymm12)); }
37             for (i = col_reduced_16; i < M; i++) {
38                 c[i] += A[index2 + i] * b[k];}
39         }
40         for (i = 0; i < col_reduced_4; i += 4) {
41             _mm256_storeu_pd(&C[index1 + i], _mm256_loadu_pd(&c[i]));}
42         for (i = col_reduced_4; i < M; i++) {
43             C[index1 + i] = c[i];
44         }
```

```
45      }
46      _mm_free(b);
47      _mm_free(c);
48  }
```

Listing 3: AVX implementation with FMA

```
1   void basic_dgemm(const int lda, const int M, const int N, const int K,
2           const double *A, const double *B, double *C)
3   {
4       int i, j, k;
5       const int col_reduced_4 = M - M % 4;
6       // This is used for loop unrolling
7       const int col_reduced_16 = M - M % 16;
8       __m256d ymm0, ymm1, ymm2, ymm3, ymm4, ymm5, ymm6, ymm7, ymm8;
9       // Memory alignment
10      double *b = _mm_malloc(K * sizeof(double), 64);
11      double *c = _mm_malloc(M * sizeof(double), 64);
12      // Try to do the computation in the jki order
13      for (j = 0; j < N; ++j) {
14          const int index1 = j * lda;
15          for (k = 0; k < K; k++) {
16              b[k] = B[index1 + k];
17          }
18          for (i = 0; i < M; i++) {
19              c[i] = C[index1 + i];
20          }
21          for (k = 0; k < K; ++k) {
22              const int index2 = k * lda;
23              // broadcast b[k] to 4 double digits that are stored in the
24              // 256 aligned memory
25              ymm0 = _mm256_broadcast_sd(&b[k]);
26              // Do the C[i] += A[index2 + i] * B[k] using loop unrolling
27              for (i = 0; i < col_reduced_16; i += 16) {
28                  // load in 16 digits of A, since A is not aligned,
29                  // we need to use loadu.
30                  ymm1 = _mm256_loadu_pd(&A[index2 + i]);
31                  ymm2 = _mm256_loadu_pd(&A[index2 + i + 4]);
32                  ymm3 = _mm256_loadu_pd(&A[index2 + i + 8]);
33                  ymm4 = _mm256_loadu_pd(&A[index2 + i + 12]);
34
35                  // load in 16 digits of C
36                  ymm5 = _mm256_load_pd(&c[i]);
37                  ymm6 = _mm256_load_pd(&c[i + 4]);
38                  ymm7 = _mm256_load_pd(&c[i + 8]);
39                  ymm8 = _mm256_load_pd(&c[i + 12]);
```

```
40
41                      // This line is doing C[i] += A[index2 + i] * B[k] using fma
42                      _mm256_store_pd(&c[i], _mm256_fmadd_pd(ymm1, ymm0, ymm5));
43                      _mm256_store_pd(&c[i + 4], _mm256_fmadd_pd(ymm2, ymm0, ymm6));
44                      _mm256_store_pd(&c[i + 8], _mm256_fmadd_pd(ymm3, ymm0, ymm7));
45                      _mm256_store_pd(&c[i + 12], _mm256_fmadd_pd(ymm4, ymm0, ymm8));
46                  }
47                  for (i = col_reduced_16; i < M; i++) {
48                      c[i] += A[index2 + i] * b[k];
49                  }
50              }
51              for (i = 0; i < col_reduced_4; i += 4) {
52                  _mm256_storeu_pd(&C[index1 + i], _mm256_loadu_pd(&c[i]));
53              }
54              for (i = col_reduced_4; i < M; i++) {
55                  C[index1 + i] = c[i];
56              }
57          }
58          _mm_free(b);
59          _mm_free(c);
60      }
```