CS 5220 HW1

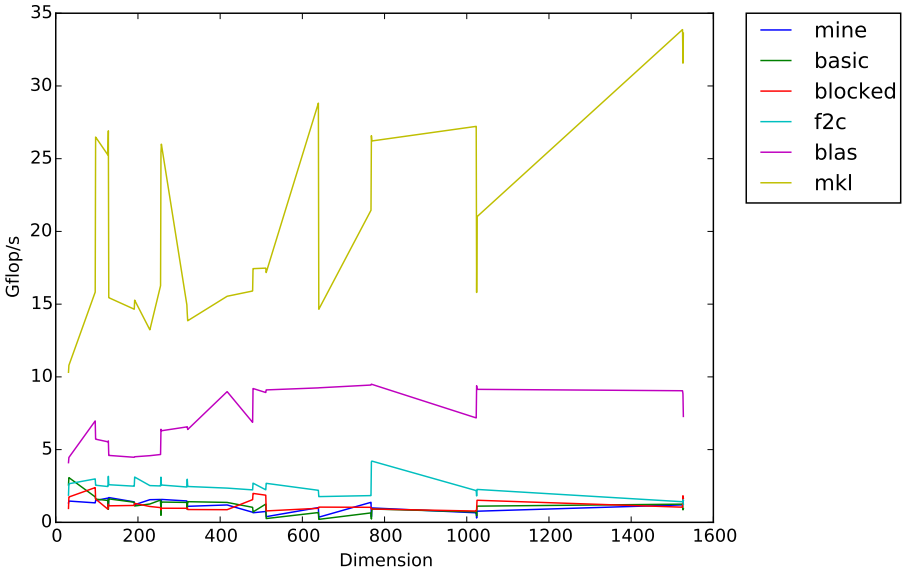
bc496, djf244, lmh95

September 2015

1 Introduction

We sought to test the effects of different optimization techniques on the performance of a square double-precision matrix multiplication. 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.

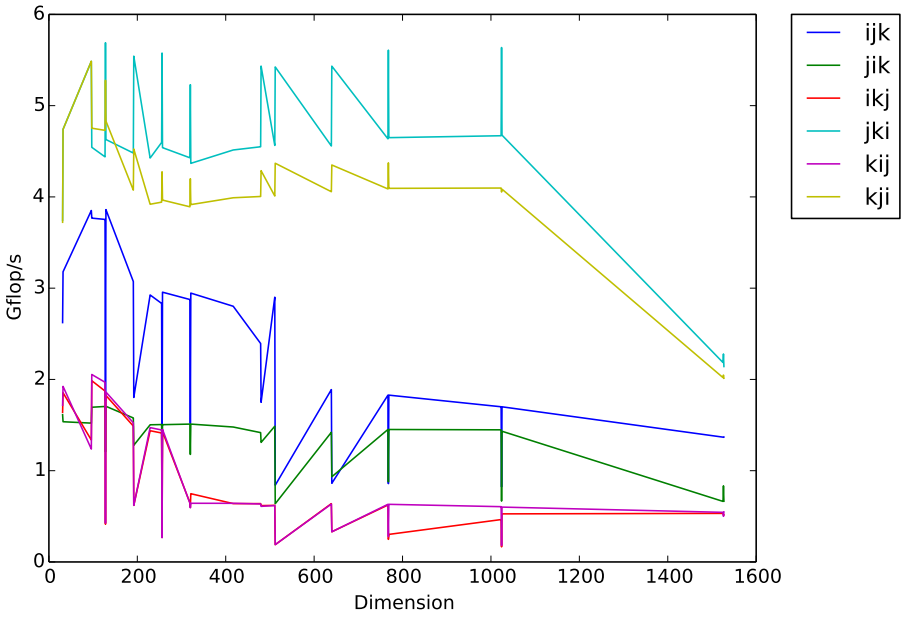
Figure 1: Performance for baseline implementations.



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.

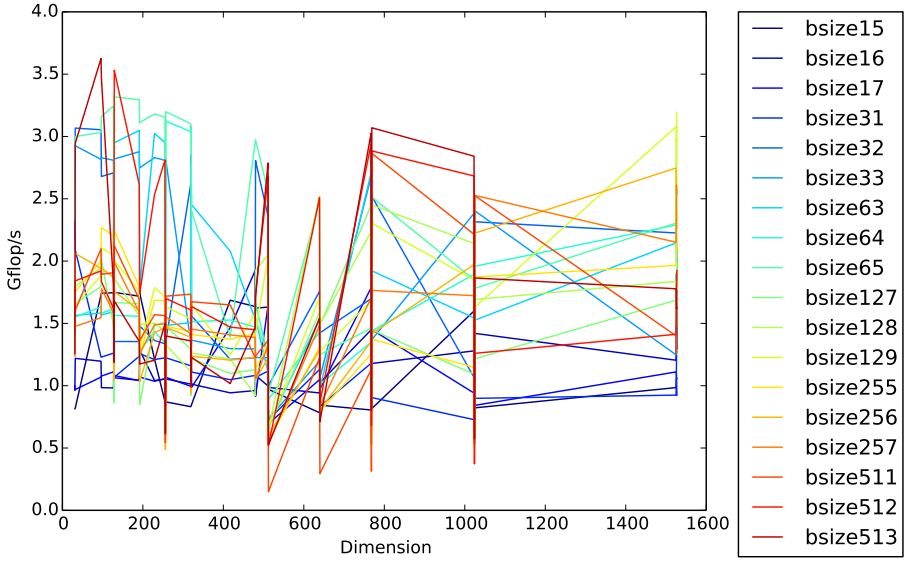
Figure 2: Performance for alternate loop orderings.



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.

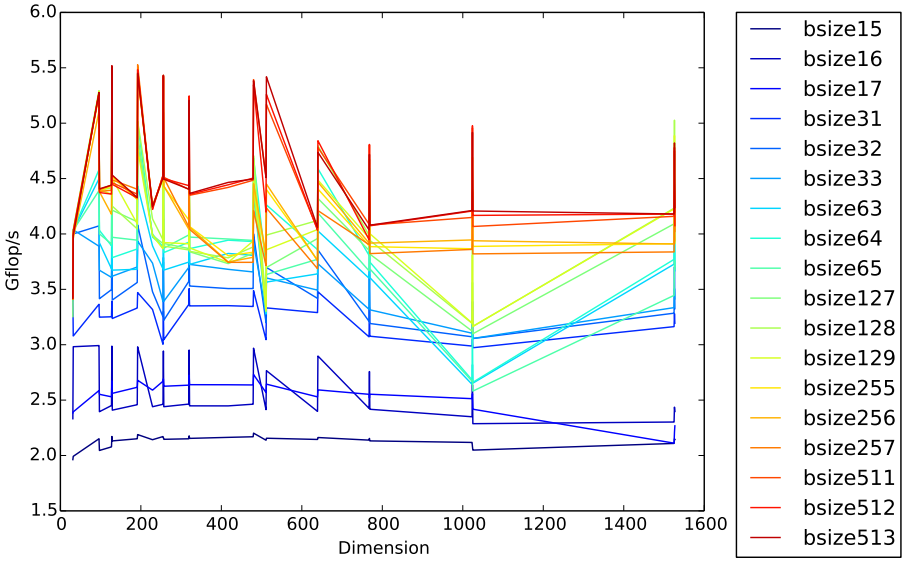
Figure 3: Performance of different block sizes with standard loop order.



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 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

This reference of this code is `https://thinkingandcomputing.com/2014/02/28/using-avx-instructions-in-matrix-multiplication/`. Basically, it did the unrolling and vectorization. I changed the gcc intrinsics to icc intrinsics. The result is summarized in the 4. With AVX, the baseline algorithm performs better than basic and blocked algorithm and when the dimension is large, it performs competitively with respect to f2c. But there is a significant drop when the dimension reach 1000.

5 Memory Alignment

Use contiguous memory will further improve the speed of AVX. However without blocking, it is impossible to use _mm_malloc to align the memory for the whole matrix. Here I just use _mm_malloc to align the memory for one single column. The result is summarized in the 6. In the future, we plan to tune the block size so that each block can fit into the memory. In this case, we could align the memory to make it faster.

Figure 5: Performance of AVX without alignment

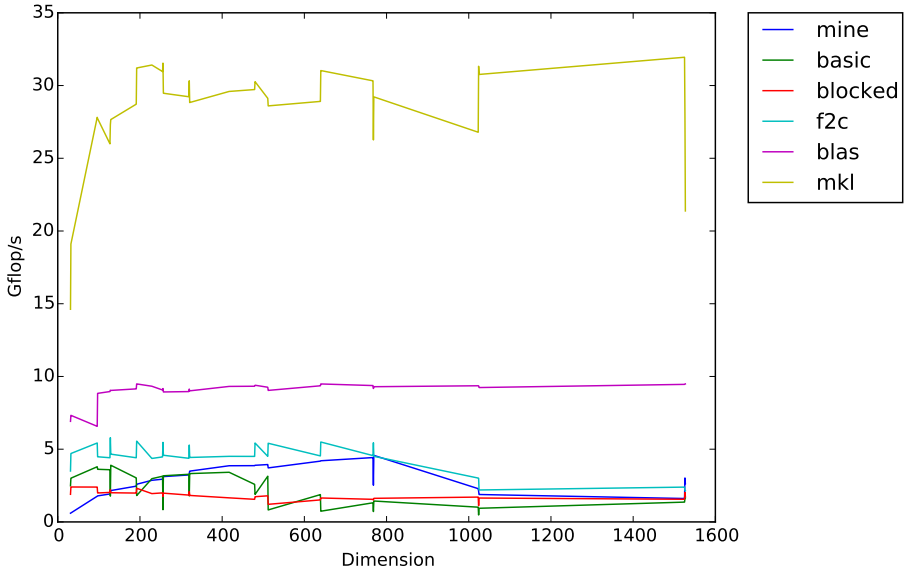


Figure 6: Performance of AVX with alignment

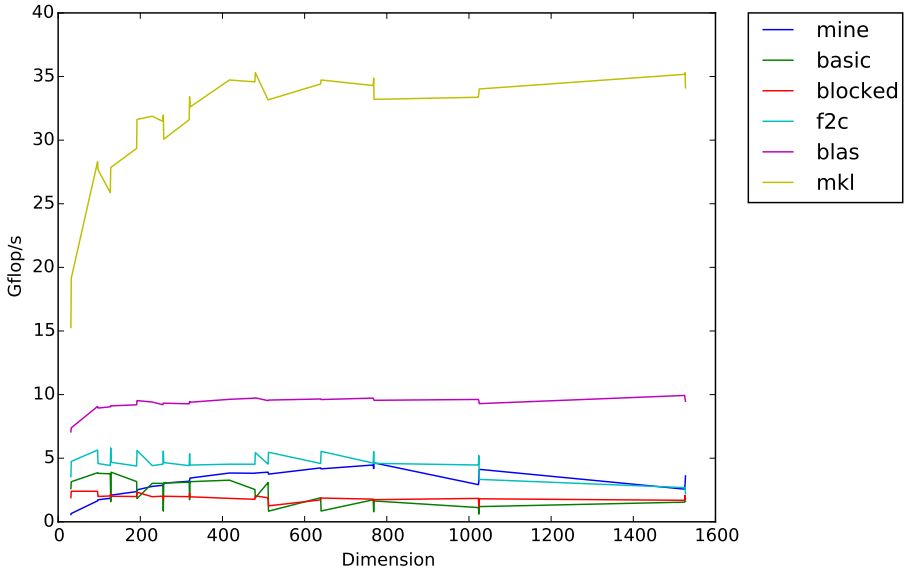
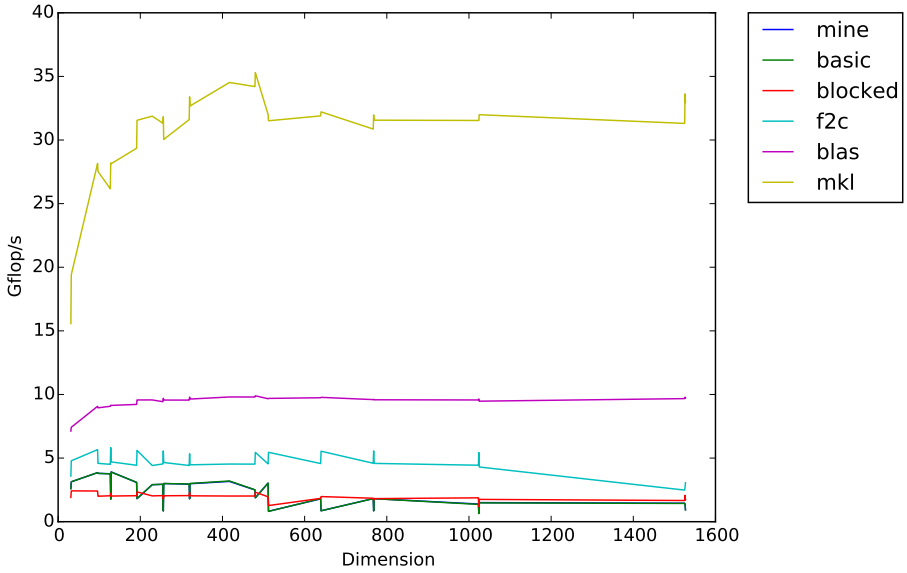


Figure 7: Default flags



6 Compiler flags

I tested the `-xHost`, `-mtune=core-avx2`, `-unroll-aggressive`, and `-qopt-prefetch=4` flags. Since `-unroll-agressive` made the performance worse, I did not test it in conjunction with the others, and since `-qopt-prefetch=4` only helped with blocked and basic, We decided to wait to test it with our final dgemm. Here is a table showing the overall speedup on each `matmul-*`:

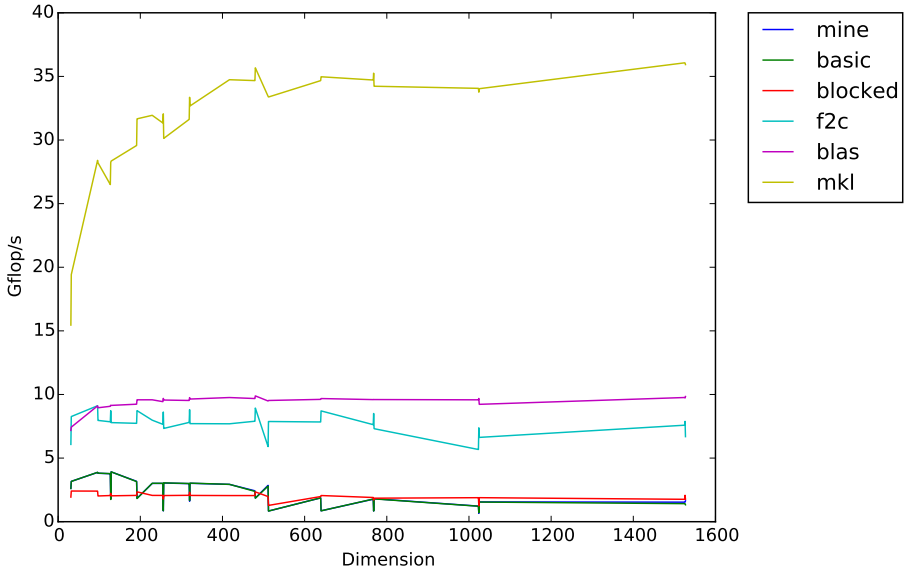
	basic	blocked	f2c	blas	mkl
default flags	00:03:40	00:02:51	00:01:54	00:01:32	00:01:07
`-xHost`	00:02:40	00:02:09	00:01:02	00:01:02	00:00:50
`-mtune`	00:02:51	00:02:42	00:01:43	00:01:27	00:00:54
`-unroll`	00:06:08	00:04:12	00:02:08	00:02:18	00:01:48
`-qopt`	00:02:47	00:02:09	00:02:25	00:01:07	00:01:12
`-xHost` & `-mtune`	00:02:50	00:02:53	00:01:15	00:01:14	00:01:20

The graphs of dimension vs. Gflops/s for the default Makefile and the Makefile with the best flag (`-xHost`) are in Figure 7 and Figure 8, respectively.

The `-xHost` flag made the most difference for every `matmul-*`, so we will be using it with every test from now on. This makes sense because it uses the highest instruction set and processor available on our machine, leading to more efficient code.

The `-qopt` flag made a lot of difference with the basic and blocked dgemms, so we will test it with our finialized implementation which will likely include

Figure 8: Default flags plus `-xHost`



blocking. It increases the prefetch optimization, so it makes sense that it would help with routines that aren't as implicitly good at memory management.

The `-mtune` flag does similar optimizations to the `-xHost` flag, but it appears to not do them as well for our architecture.

I was surprised at how poorly the `-unroll` flag performed - it both greatly increased the runtime and decreased the Gflop/s. Since basic unrolling is part of `-O3` (which worked a bit better than `-O2` when I did preliminary tests), I would have thought that more aggressive unrolling would be even better.

I left out of this analysis flags that made little difference to the implementations we currently have, but may still test with our final implementation, so that the table did not become bloated and unreadable.

7 Future Work

To create our final implementation, we will optimally combine our results from this analysis. For example, as mentioned above, we will use the `-qopt-prefetch=4` flag on our own implementation if it primarily uses blocking.

We will try to build a kernel that can do fixed size matrix multiplication using the best possible loop order, blocking size, AVX and aligned memory. For the whole matrix, we will pad it with zeros if necessary so that it can be divided into small chunks and do calculation using the kernel. We will also do some autotuning with respect to compiler flags to get the best possible speed with our implementation.

Additionally, we will test copy optimization on our already-tuned code, since its form and effectiveness both depend on the framework already there (and it would be a waste of human time to try to tune this before we have that framework).

Appendix

Code for memory aligned AVX

```
1  void square_dgemm(const int M, const double *A, const double *B, double *C) {
2      double *scratchpad = _mm_malloc(4 * sizeof(double), 64);
3      int i,j,k,s;
4      double *D = (double*) malloc(M * M * sizeof(double));
5      for (i = 0; i < M; i++) {
6          for (j = 0; j < M; j++) {
7              D[i * M + j] = A[j * M + i];
8          }
9      }
10     for (i = 0; i < M; i++) {
11         const int col_reduced = M - M%32;
12         const int col_reduced_16 = M - M%16;
13         __m256d ymm0, ymm1, ymm2, ymm3, ymm4, ymm5, ymm6, ymm7,
14                 ymm8, ymm9, ymm10, ymm11, ymm12, ymm13, ymm14, ymm15;
15         const double *x = _mm_malloc(M * sizeof(double), 64);
16         x = &B[i * M];
17         for (j = 0; j < M; j ++) {
18             double res = 0;
19             const int index = j * M;
20             for (k = 0; k < col_reduced; k += 32) {
21                 ymm8 = _mm256_load_pd(&x[k]);
22                 ymm9 = _mm256_load_pd(&x[k + 4]);
23                 ymm10 = _mm256_load_pd(&x[k + 8]);
24                 ymm11 = _mm256_load_pd(&x[k + 12]);
25                 ymm12 = _mm256_load_pd(&x[k + 16]);
26                 ymm13 = _mm256_load_pd(&x[k + 20]);
27                 ymm14 = _mm256_load_pd(&x[k + 24]);
28                 ymm15 = _mm256_load_pd(&x[k + 28]);
29
30                 ymm0 = _mm256_loadu_pd(&D[index + k]);
31                 ymm1 = _mm256_loadu_pd(&D[index + k + 4]);
32                 ymm2 = _mm256_loadu_pd(&D[index + k + 8]);
33                 ymm3 = _mm256_loadu_pd(&D[index + k + 12]);
34                 ymm4 = _mm256_loadu_pd(&D[index + k + 16]);
35                 ymm5 = _mm256_loadu_pd(&D[index + k + 20]);
36                 ymm6 = _mm256_loadu_pd(&D[index + k + 24]);
37                 ymm7 = _mm256_loadu_pd(&D[index + k + 28]);
```

```
38
39                  ymm0 = _mm256_mul_pd(ymm0, ymm8 );
40                  ymm1 = _mm256_mul_pd(ymm1, ymm9 );
41                  ymm2 = _mm256_mul_pd(ymm2, ymm10);
42                  ymm3 = _mm256_mul_pd(ymm3, ymm11);
43                  ymm4 = _mm256_mul_pd(ymm4, ymm12);
44                  ymm5 = _mm256_mul_pd(ymm5, ymm13);
45                  ymm6 = _mm256_mul_pd(ymm6, ymm14);
46                  ymm7 = _mm256_mul_pd(ymm7, ymm15);
47
48                  ymm0 = _mm256_add_pd(ymm0, ymm1);
49                  ymm2 = _mm256_add_pd(ymm2, ymm3);
50                  ymm4 = _mm256_add_pd(ymm4, ymm5);
51                  ymm6 = _mm256_add_pd(ymm6, ymm7);
52                  ymm0 = _mm256_add_pd(ymm0, ymm2);
53                  ymm4 = _mm256_add_pd(ymm4, ymm6);
54                  ymm0 = _mm256_add_pd(ymm0, ymm4);
55
56                  _mm256_store_pd(scratchpad, ymm0);
57                  for (s = 0; s < 4; s++)
58                      res += scratchpad[s];
59              }
60              for (k = col_reduced; k < col_reduced_16; k += 16) {
61                  ymm8 = _mm256_load_pd(&x[k]);
62                  ymm9 = _mm256_load_pd(&x[k + 4]);
63                  ymm10 = _mm256_load_pd(&x[k + 8]);
64                  ymm11 = _mm256_load_pd(&x[k + 12]);
65
66                  ymm0 = _mm256_loadu_pd(&D[index + k]);
67                  ymm1 = _mm256_loadu_pd(&D[index + k + 4]);
68                  ymm2 = _mm256_loadu_pd(&D[index + k + 8]);
69                  ymm3 = _mm256_loadu_pd(&D[index + k + 12]);
70
71                  ymm0 = _mm256_mul_pd(ymm0, ymm8 );
72                  ymm1 = _mm256_mul_pd(ymm1, ymm9 );
73                  ymm2 = _mm256_mul_pd(ymm2, ymm10);
74                  ymm3 = _mm256_mul_pd(ymm3, ymm11);
75
76                  ymm0 = _mm256_add_pd(ymm0, ymm1);
77                  ymm2 = _mm256_add_pd(ymm2, ymm3);
78                  ymm0 = _mm256_add_pd(ymm0, ymm2);
79
80                  _mm256_store_pd(scratchpad, ymm0);
81                  for (s = 0; s < 4; s++)
82                      res += scratchpad[s];
83              }
```

```
84                 for (k = col_reduced_16; k < M; k++) {
85                     res += D[index + k] * x[k];
86                 }
87                 C[j + i * M] = res;
88             }
89         }
90 }
```