CS267 HW1: Optimizing Matrix Multiplication

Group 16
Tiffany Tang, Ishaq Aden-Ali, Patrick Neal

Here, we will discuss the various optimization techniques applied to matrix multiplication attempted by our group. The naive triply nested loop had an average percentage of peak performance of roughly 4.5%, which is our starting baseline.

We all initially started working with SIMD vectorized operations since this would be necessary for any well optimized implementation.Then, we split up and each experimented with a different set of additional strategies. First, we discuss the impact of simply adding vector instructions to the given dgemm-blocked.c code. All of the following techniques build further on top of this initial improvement.

Optimization Techniques and Outcomes

SIMD

The largest performance gain we saw through the whole homework was with our initial addition of SIMD instructions. Specifically, we load four doubles from A and C, broadcast a value of B, then perform a fused multiply-add. Finally, we store the updated vector c0 back in the original location in C.

```
__m256d a0 = _mm256_loadu_pd(A_local + i + (k + 0) * BLOCK_SIZE);
__m256d c0 = _mm256_loadu_pd(&C[i + j * lda]);
__m256d b0 = _mm256_broadcast_sd(B_local + (k + 0) + j * BLOCK_SIZE);
c0 = _mm256_fmadd_pd(a0, b0, c0);
_mm256_storeu_pd(&C[i + j * lda], c0);
```

This allows for processing four element chunks at a time. For matrices whose size are not divisible by 4, the remaining elements are processed with standard scalar operations. Working on chunks of 4 with the AVX instructions added to the provided blocked implementation gave an increase to 23% peak performance. Unless otherwise noted, this strategy of loading chunks of A and C, broadcasting an element of B, and performing a fused multiply add was used in all subsequent implements.

Loop Reordering

There are a number of ways to restructure the loops in matrix multiplication to improve cache locality. In our final approach, we found the (j, i, k) loop ordering to be most effective. But, in the process of experimenting one of the most bang-for-buck methods found surprisingly effective was to iterate through the rows and columns of B, broadcasting the value to the AVX register. Then the inner loop loads chunks of A and C's columns. This ensures that we are always accessing each matrix's underlying elements contiguously. Each element of B is accessed exactly once and each chunk of C that gets loaded is contiguous to the last chunk improving cache locality. The biggest source of cache misses then becomes re-fetching the start of matrix A once the previous iteration has hit the last column of A.

```
  for (int B_col=0; B_col < n; ++B_col) {
        for (int B_row=0; B_row < n; ++B_row) {
            int B_flat_idx = (B_col * n) + B_row;
            double B_val = B[B_flat_idx];
            __m256d B_val_broad = _mm256_set1_pd(B_val);

            for (int chunk=0; chunk < n_chunks; chunk++) {
                int A_flat_idx_col_start = (B_row * n) + (chunk * 4);
                __m256d A_col = _mm256_loadu_pd(&A[A_flat_idx_col_start]);
                int C_flat_idx_col_start = (B_col * n) + (chunk * 4);
                __m256d C_col = _mm256_loadu_pd(&C[C_flat_idx_col_start]);
                C_col = _mm256_fmadd_pd(B_val_broad, A_col, C_col);
                _mm256_storeu_pd(&C[C_flat_idx_col_start], C_col);
}}}
```

While this was fast, with an average percentage of peak of 26%, we favored the blocking approach with the (j, i, k) re-ordering since we were experimenting with the microkernel at first.

Multilevel Blocking

We implemented multilevel blocking with an outer block size of 128×128 and an inner block size of 8×8, expecting improved cache locality and better SIMD utilization. However, our results indicate that multilevel blocking does not provide a noticeable performance improvement over single-level blocking. Given the lack of substantial improvement, we chose to simplify our implementation by processing blocks directly.

Repacking

Another way to combat cache misses from striding is to repack an individual block of the matrix multiplication into a temporary buffer that is laid out for optimal access. But, repacking itself is not free. Allocating the buffer and copying the matrices can be expensive depending on the size. You ultimately end up paying for the cache misses upfront. For a small matrix, it is often not worth the cost. In our hands, repacking seemed to give at best marginal improvement. The matrices in the benchmark are not particularly large, and since we are doing just a single matrix multiplication there isn't a great way to re-use the repacked blocks to maximize the benefit from costly repacking.

We attempted to align the repacked buffers so that we could benefit from aligned AVX loads. However, the cases where the matrix size was not divisible by 32 presented issues, throwing off the alignment after the first loop iterations. We started an attempt to pad the blocks with zeros to alleviate this, but ran out of time to fully complete.

Lastly, we also experimented with a third repacking technique where matrix B's block is transposed so we could attempt an efficient dot product with a different set of SIMD instructions as shown below:

```
void double_dot_product(__m256d v1, __m256d v2, __m256d v3, __m256d v4, double* res1, double* res2) {
        // Multiply
        __m256d mul1 = _mm256_mul_pd(v1, v2);
        __m256d mul2 = _mm256_mul_pd(v3, v4);

        // Horizontal sum
        __m256d sum = _mm256_hadd_pd(mul1, mul2);
        __m128d sum_high = _mm256_extractf128_pd(sum, 1);
        __m128d result = _mm_add_pd(sum_high, _mm256_castpd256_pd128(sum));

        // Store in respective res's
        *res1 = _mm_cvtsd_f64(result);
        *res2 = _mm_cvtsd_f64(_mm_shuffle_pd(result, result, 1));
}
```

Ultimately, this implementation was only giving roughly 20% of peak performance, which was worse than the fused multiply-add methods, therefore we decided to stop exploring this avenue and returned to simply packing the blocks into a contiguous buffer with the fused multiply-add technique.

Microkernel

We initially implemented a 4×4 microkernel to optimize inner-block computations using AVX2 vectorization. However, this approach only achieved 25% of peak performance. To further optimize, we attempted an 8×8 microkernel, but encountered numerical precision errors and incorrect results, likely due to alignment issues and register spilling, unfortunately we weren't able to resolve these precision errors within the time frame.

```
static inline void microkernel_4x4(int lda, double* A, double* B, double* C) {
    __m256d c0 = _mm256_loadu_pd(&C[0 * lda]);  // Load C column 0, 1, 2, 3
    __m256d c1 = _mm256_loadu_pd(&C[1 * lda]);
    __m256d c2 = _mm256_loadu_pd(&C[2 * lda]);
    __m256d c3 = _mm256_loadu_pd(&C[3 * lda]);

    for (int k = 0; k < 4; ++k) {
        __m256d a0 = _mm256_loadu_pd(&A[k * lda]);  // Load 4 elements of A
        __m256d b0 = _mm256_broadcast_sd(&B[k + 0 * lda]);
        __m256d b1 = _mm256_broadcast_sd(&B[k + 1 * lda]);
        __m256d b2 = _mm256_broadcast_sd(&B[k + 2 * lda]);
        __m256d b3 = _mm256_broadcast_sd(&B[k + 3 * lda]);

        c0 = _mm256_fmadd_pd(a0, b0, c0);
        c1 = _mm256_fmadd_pd(a0, b1, c1);
        c2 = _mm256_fmadd_pd(a0, b2, c2);
        c3 = _mm256_fmadd_pd(a0, b3, c3);
    }

    _mm256_storeu_pd(&C[0 * lda], c0);
    _mm256_storeu_pd(&C[1 * lda], c1);
    _mm256_storeu_pd(&C[2 * lda], c2);
    _mm256_storeu_pd(&C[3 * lda], c3);
}
```

Given these challenges, we decided to process blocks directly, integrating memory loads within the computation loops. We will discuss the impact of this decision and our final combined optimizations in the next section, "Combined Final Performance".

Combined Final Performance

Our final implementation incorporates multiple performance optimization techniques that we explored:

Blocking for Cache Efficiency

This improves cache locality by operating on small blocks of matrices instead of the entire matrix at once. We found a block size of 32 to yield the best performance.

SIMD Vectorization with AVX2 Instructions

Our final code uses AVX2 intrinsics (_mm256_fmadd_pd, _mm256_loadu_pd, _mm256_storeu_pd) to vectorize the computation. We therefore perform operations on 4 floating-point values in parallel, further benefiting from fused multiply-add. We also leverage broadcasting to prevent loading B multiple times, instead sending a single value across all SIMD lanes using _mm256_broadcast_sd().

Preloading & Repacking Matrices

Instead of accessing memory inefficiently, we copy blocks of A and B into contiguous arrays (A_local, B_local). This improves data alignment and preloads data into cache, reducing translation lookaside buffer misses. Because our block is relatively small, it fits on the stack preventing a more expensive heap allocation.

```
double A_local[BLOCK_SIZE * BLOCK_SIZE];
for (int i = 0; i < M; i++) {
    for (int j = 0; j < K; j++) {
        A_local[i + j * BLOCK_SIZE] = A[i + j * lda];
 }}
```

Loop Ordering for Memory Locality

While we explored a few loop reorderings, we settled on (j, i, k) to work best in conjunction with our blocking and repacked memory layout.

```
for (int j = 0; j < N4; j += 4) {
    for (int i = 0; i < M4; i += 4) {
        for (int k = 0; k < K4; k++) {
            // SIMD operations...
 }}}
```

Eliminating Microkernel Overhead

We initially implemented a 4×4 microkernel, but it only reached 25% of peak performance. Instead of a dedicated microkernel function, we process blocks directly and mix memory loads with computation loops.

```
for (int j = 0; j < N4; j+=4) {
        for (int i = 0; i < M4; i += 4) {
            // load 4 double-precision floating point values, u - unaligned memory
            __m256d c0 = _mm256_loadu_pd(&C[i + (j+0) * lda]);
            __m256d c1 = _mm256_loadu_pd(&C[i + (j+1) * lda]);
            __m256d c2 = _mm256_loadu_pd(&C[i + (j+2) * lda]);
            __m256d c3 = _mm256_loadu_pd(&C[i + (j+3) * lda]);

            // for (int k = 0; k < K4; k += 4) {
            for (int k = 0; k < K4; k++) {
                __m256d a0 = _mm256_loadu_pd(A_local + i + k * BLOCK_SIZE);

                // duplicates a single double value across all 4 lanes
                // used when multiplying a row of A with a single value from B, more efficient than
loading the same value 4 times
```

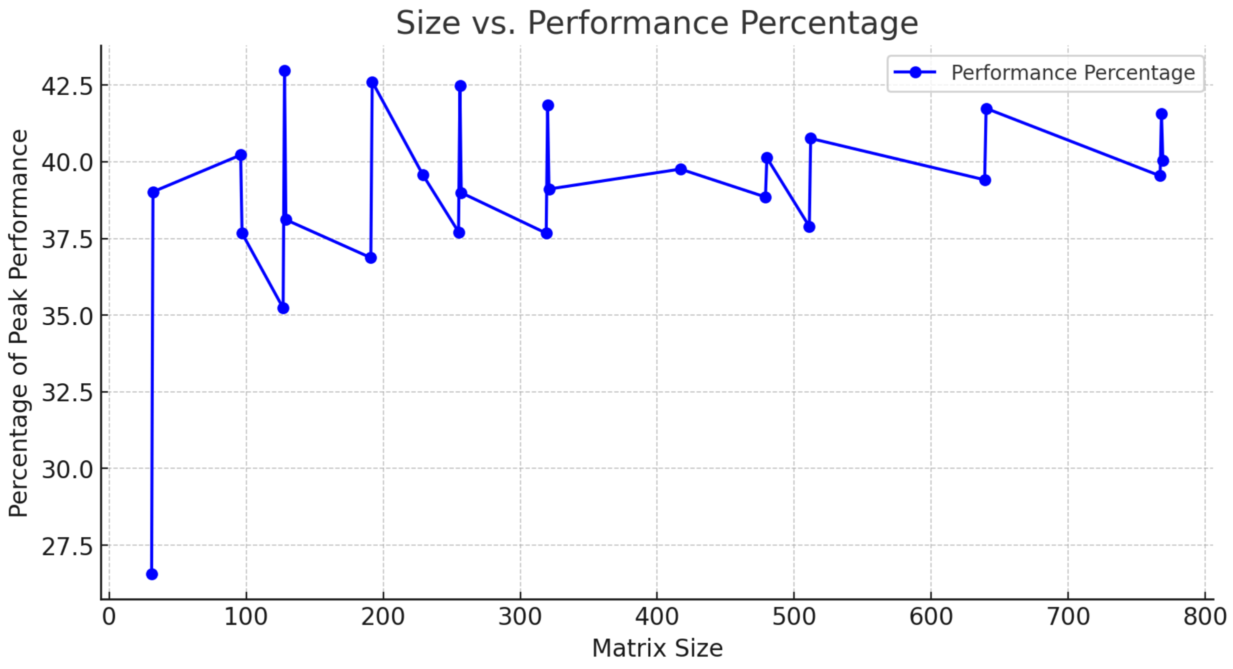
```
                __m256d b0 = _mm256_broadcast_sd(B_local + k + j * BLOCK_SIZE);
                __m256d b1 = _mm256_broadcast_sd(B_local + k + (j+1) * BLOCK_SIZE);
                __m256d b2 = _mm256_broadcast_sd(B_local + k + (j+2) * BLOCK_SIZE);
                __m256d b3 = _mm256_broadcast_sd(B_local + k + (j+3) * BLOCK_SIZE);

                // perform fused multiply-add, (A * B) + C
                c0 = _mm256_fmadd_pd(a0, b0, c0);
                c1 = _mm256_fmadd_pd(a0, b1, c1);
                c2 = _mm256_fmadd_pd(a0, b2, c2);
                c3 = _mm256_fmadd_pd(a0, b3, c3);
            }
            // Handle remaining elements if K is not a multiple of 4
            for (int k_rem = K4; k_rem < K; ++k_rem) {
                __m256d a0 = _mm256_loadu_pd(&A[i + k_rem * lda]);
                __m256d b0 = _mm256_broadcast_sd(&B[k_rem + j * lda]);
                __m256d b1 = _mm256_broadcast_sd(&B[k_rem + (j+1) * lda]);
                __m256d b2 = _mm256_broadcast_sd(&B[k_rem + (j+2) * lda]);
                __m256d b3 = _mm256_broadcast_sd(&B[k_rem + (j+3) * lda]);
                c0 = _mm256_fmadd_pd(a0, b0, c0);
                c1 = _mm256_fmadd_pd(a0, b1, c1);
                c2 = _mm256_fmadd_pd(a0, b2, c2);
                c3 = _mm256_fmadd_pd(a0, b3, c3);
        }
            // stores 4 double-precision floating-point values back into memory
            _mm256_storeu_pd(&C[i + j * lda], c0);
            _mm256_storeu_pd(&C[i + (j+1) * lda], c1);
            _mm256_storeu_pd(&C[i + (j+2) * lda], c2);
            _mm256_storeu_pd(&C[i + (j+3) * lda], c3);
        }}
```

Handling Edge Cases Efficiently

Handles cases where M, N, or K are not multiples of 4 by using scalar fallback loops. Maintains high performance for aligned cases while handling irregular sizes effectively. We found this more effective than our unfinished attempts to work on blocks padded with zeros.

```
// Handle Remaining Columns (N % 4 != 0)
    for (int j = N4; j < N; ++j) {
        for (int i = 0; i < M4; i += 4) {
            __m256d c0 = _mm256_loadu_pd(&C[i + j * lda]);
            for (int k = 0; k < K; ++k) {
                __m256d a0 = _mm256_loadu_pd(&A[i + k * lda]);
                __m256d b0 = _mm256_broadcast_sd(&B[k + j * lda]);
                c0 = _mm256_fmadd_pd(a0, b0, c0);
            }
            _mm256_storeu_pd(&C[i + j * lda], c0);
        }}
    // Handle Remaining Rows (M % 4 != 0)
    for (int j = 0; j < N; ++j) {
        for (int i = M4; i < M; ++i) {
            double cij = C[i + j * lda];
            for (int k = 0; k < K; ++k) {
                cij += A[i + k * lda] * B[k + j * lda];
            }
            C[i + j * lda] = cij;
        }
    }
```

This combined implementation results in an average performance of **39.09% of peak**, a substantial improvement over our initial attempts.



The notable dips in performance correlate with input sizes that are not multiples of our block size. This is due to falling back on scalar operations for the remaining elements. However, the performance is relatively even across the range of n=100-800 suggesting our implementation is not heavily biased toward small or large matrices at this scale.

Areas for Further Improvement

1. **Improved Multi-Level Blocking**

While we attempted multi-level blocking with a 128×128 outer block and 8×8 inner block, it did not provide a noticeable improvement over single-level blocking. Future work could explore adaptive blocking strategies that dynamically adjust block sizes based on cache hierarchy profiling.

2. **Higher-Dimension Microkernels**

We initially implemented a 4×4 microkernel, which yielded only 25% of peak performance. Our attempt to expand to an 8×8 microkernel resulted in numerical precision errors, possibly due to alignment issues and register spilling. Refining the microkernel implementation, particularly handling register usage and loop unrolling, could further improve performance.

3. **Optimized Packing Strategy**

We experimented with matrix repacking to improve cache locality, but the additional overhead from memory allocation and copying offset much of the benefit. A more efficient packing strategy, such as revisiting transposed blocks of B, ensuring our packed arrays have 32byte alignment for faster AVX loads, or streaming stores for A and C, could reduce the cost while improving performance.

4. **Handling Remainders More Efficiently**

While our implementation correctly handles cases where M, N, or K are not multiples of 4, it does so using separate scalar fallback loops. An alternative approach is to use masked AVX2 operations to handle irregular sizes without switching to scalar operations, which could further reduce performance degradation in edge cases.

Conclusions

In this project, we explored various optimization strategies for matrix multiplication using SIMD, loop reordering, blocking, repacking, and microkernels.

1. Our **SIMD vectorized blocking** approach using **AVX2 intrinsics** led to the largest performance gains, reaching **39.09%** of peak performance.
2. **Loop ordering** improvements helped optimize cache locality, reducing redundant memory accesses.
3. **Repacking and preloading data** into contiguous memory blocks helped improve alignment but did not yield significant benefits given the problem size.
4. **Microkernel optimizations** were partially successful, with a 4×4 microkernel achieving 25% performance.
5. **Multi-level blocking** provided no significant improvement, suggesting that our single-level blocking strategy was close to optimal.

Ultimately, we decided to process blocks directly, integrating memory loads within computation loops. This approach balanced performance and numerical stability while ensuring efficient SIMD utilization. While our implementation significantly improves upon the baseline naive approach, further optimizations in parallelization, packing, and microkernel refinement could push performance even closer to the hardware’s peak capabilities.