Efficient Matrix Multiplication

Po-Yen Chen, Sebastian Prillo, Numi Sveinsson

1 Introduction

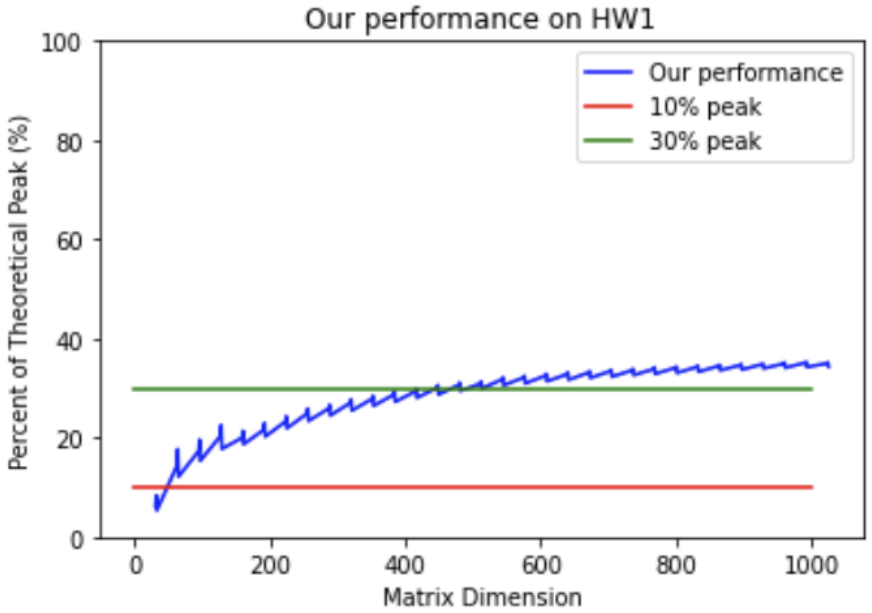

Figure 1: Our performance on HW1 averages $\sim 28.6\%$ peak.

In this project, we set out to implement an efficient, single-threaded implementation of matrix multiplication. Concretely, given matrices A, B, C of dimension `lda` $\times$ `lda`, we want to implement the update $C \mathrel{+}= A \times B$. The three matrices are stored in column-major order. A and B may alias each other, but C is not aliased. Originally provided naive and blocked implementations of matrix multiplication both achieve $< 1\%$ peak. Our goal is to obtain at least $10\% - 30\%$ peak.

We were finally able to achieve $\sim 28.6\%$ peak when averaged over all test sizes. Our performance for the largest block sizes reaches $\sim 35\%$. We will describe the steps we took to get there. Figure 1 shows our performance across all input sizes.

2 Strategy

In this work, we explored the following optimizations; we will describe our journey implementing these later:

- Allowing from very generic, non-square block sizes. More specifically, the blocks of A will be `M_BLOCK_SIZE`$\times$ `K_BLOCK_SIZE` and the blocks of B will be of size `K_BLOCK_SIZE` $\times$ `M_BLOCK_SIZE`.
- **Repacking** suitable parts of A and B to optimize our direct-mapped L1/L2 cache performance (i.e. minimize communication between main memory and cache).

- **SIMD microkernels** for vector and matrix multiplication, which furthermore introduce a second level of blocking.
- **Pre-fetching** within the microkernel to accelerate the microkernel.

Let us start by motivating *why* the above optimization strategies are relevant:

First, some preliminaries are important. As we have seen in lecture (and becomes even more apparent when implementing HW1), reading and writing data in main memory are the most expensive operations for matrix multiplication. Therefore, it is important to (1) minimize the number of such operations, and (2) maximize memory bandwidth by issuing all such operations at the same time if possible. Technically, this is called *instruction level parallelism* (ILP), because operations are executed by the processor in parallel, also known as *pipelining*.

Blocked matrix multiply precisely looks to achieve the two goals in the previous paragraph by loading blocks of A, B and C to the cache, *then* performing their matrix product, and *then* writing the result back to the block of C. By doing this, the computational intensity of matrix multiply compared to the naive implementation is potentially B times larger, where B is the block size. One would be tempted to choose B as large as possible, but this is not so simple, because the blocks must **fit in the cache**. This means that finding the right block sizes is a non-trivial task, which requires carefully doing manual computations and/or brute-forcing block sizes (ideally, both). But in particular, it means that we want to be flexible about the block sizes we use, which motivates the first strategy we will use: **allowing from very generic, non-square block sizes**. To make things worse, there are two levels of caching, and each cache has different sizes (L2 cache is 1 MB direct-mapped, and L1 cache is 32KB direct-mapped).

Another important aspect is the fact that our caches are **direct-mapped**. This means that positions in main memory with the same address modulo the cache size will compete with each other. Therefore, the blocks of A and B which we are working with should be arranged to minimize cache collisions. This is achieved by the technique of **repacking** - the second strategy in our list above - wherein the blocks of A and B that we are working with are *first* copied into contiguous memory locations, and *then* multiplied together.

Third, even though main memory reads and writes are the most costly operations, provided that they have been minimized, a significant amount of compute is still needed to perform the matrix product of the blocks of A and B. This can be optimized by leveraging SIMD operations. The architecture we are working on offers 32 AVX512 registers capable of storing 8 doubles, as well as the associated FMA operation (among others), and 2 vector lanes. Therefore, we can expect speedups by leveraging SIMD. Concretely, we set out to write **SIMD microkernels** to speed up multiplying the block of A against blocks of B. This is our third strategy. Since our microkernel updates small (concretely, they will be 8×16) blocks of C *within* the original block of C, this corresponds to a **second level of blocking**.

Finally, software prefetching can provide additional boost in performance by making sure that data is present in the cache before it is used by the SIMD operations. We implement software prefetching similarly to how it was introduced to us in the recitation class, within the microkernel inner loop. This introduces a new hyperparameter `PFETCH_DIST` that we further need to tune.

Other strategies are possible to speed up matrix multiplication, but due to time constraints and experience we only focused on the three above. Moreover, even when only using our four strategies above, further improvements can be expected by better tuning block size hyperparameters, writing better microkernels, more clever prefetching, etc. It is thus no wonder that there is more room for improvement for our ~ 28.6 average peak. During our work, we referred to [1, 2] for ideas on repacking and logic of SIMD microkernels. We referred to recitation 1 for ideas on prefetching.

Having motivated our three main strategies, we proceed to describe how we implemented them, what limitations we found (particularly in our initial SIMD matrix multiplication microkernel), and how we tried to improve upon them (particularly, how we implemented a better SIMD matrix multiplication microkernel).

3 SIMD dot product microkernel (peak perf ~ 4.0)

Early on, we implemented a SIMD dot product microkernel and used it within the naive triple-loop implementation. Some considerations are in place:

1. Since we are taking the dot product of *rows* of A versus *columns* of B, then it is most convenient to store A in *row-major* order, and B in *column-major* order. Since A is not already stored in row-major order, we need to do this ourselves at the very beginning of the program. This optimization alone slightly improves the performance of the naive triple loop (to slightly above ~ 1.0), since elements of A are now accessed contiguously in the inner loop.

2. Since we are using SIMD operations, which operate on 8 floats at a time, we **pad** the inner dimensions of A and B to be a multiple of 8 bytes. Moreover, we **align** A and B to 64 bytes in memory, which allows us to use the faster `_mm512_load_pd` operation. Padding is a recurring theme throughout our work. Later on, when we explore matrix microkernels of size 8×16, we will pad **both** dimensions of A and B, to allow us to perform matrix multiplication with just a 8×16 microkernel.

By leveraging this dot product microkernel, we were able to obtain ~ 4.0 of peak, around a $4\times$ speedup over the naive code. This already shows how useful microkernels will be.

The following shows our SIMD dot product microkernel. We implemented this following Sebastian's previous experience in using SIMD to implement dot products and some googling to figure out the final reduction. (The most annoying part is reducing the final 8 doubles stored in an AVX512 register into their sum.)

```
/* This function computes the dot product of two vectors X and Y
 * whose length is len.
 * IMPORTANT NOTE: It is assumed that len is a multiple of 8.
 * This function will fail if len is not a multiple of 8!
 */
double dot_product(double* X, double* Y, int len){
    // We accumulate the dot product with fused-multiply-add (FMA)
    __m512d Xr, Yr, accumulator;
    accumulator = _mm512_setzero_pd();
    for(int u = 0; u < len; u += 8){
        Xr = _mm512_loadu_pd(X + u);
        Yr = _mm512_loadu_pd(Y + u);
        accumulator = _mm512_fmadd_pd(Xr, Yr, accumulator);
    }

    // Finally, we need to add up the parallel lanes of the accumulator register.
    // I've always hated the operations needed to do this! It's so hard to get right!
    // Add 8 values into 4
    const __m256d r4 = _mm256_add_pd( _mm512_castpd512_pd256( accumulator ),
        _mm512_extractf64x4_pd( accumulator, 1 ) );
    // Add 4 values into 2
    const __m128d r2 = _mm_add_pd( _mm256_castpd256_pd128( r4 ), _mm256_extractf128_pd( r4, 1 ) );
    // Add 2 lower values into the final result
    const __m128d r1 = _mm_add_sd( r2, _mm_unpackhi_pd( r2, r2 ) );

    return _mm_cvtsd_f64( r1 );
}
```

The following resource was used to figure out operations needed for the final reduction:
`https://stackoverflow.com/questions/59494745/avx2-computing-dot-product-of-512-float-arrays`

4 SIMD matrix multiplication microkernel via dot products (peak perf ~ 21.0)

We next explored extending our SIMD dot product microkernel to multiply small matrices. Concretely, when computing the matrix product XY, we perform the dot products between all rows of X and columns of Y at the same time. For example, multiplying a $2 \times K$ matrix $X = \begin{bmatrix} X0 \\ X1 \end{bmatrix}$ by a $K \times 2$ matrix $Y = [Y0|Y1]$ looks like this:

```
void matmul_2_by_2_microkernel(
    double* X0, double* X1, double* Y0, double* Y1, int K, double* res
){
    __m512d Xr0, Xr1, Yr0, Yr1, acum_0_0, acum_0_1, acum_1_0, acum_1_1;
    acum_0_0 = _mm512_setzero_pd();
    acum_0_1 = _mm512_setzero_pd();
    acum_1_0 = _mm512_setzero_pd();
    acum_1_1 = _mm512_setzero_pd();
    for(int u = 0; u < K; u += 8){
        Xr0 = _mm512_load_pd(X0 + u);
        Xr1 = _mm512_load_pd(X1 + u);
        Yr0 = _mm512_load_pd(Y0 + u);
        Yr1 = _mm512_load_pd(Y1 + u);
        acum_0_0 = _mm512_fmadd_pd(Xr0, Yr0, acum_0_0);
        acum_0_1 = _mm512_fmadd_pd(Xr0, Yr1, acum_0_1);
        acum_1_0 = _mm512_fmadd_pd(Xr1, Yr0, acum_1_0);
        acum_1_1 = _mm512_fmadd_pd(Xr1, Yr1, acum_1_1);
    }
    // Reduce accumulators
    ...
}
```

The accumulator `acum_i_j` accumulates the inner product of the i-th row of X and the j-th column of Y. In this approach, we wrote code to generate matrix multiplication microkernels of diverse sizes. We took the blocked implementation provided in the starter code and modified it to:

1. Allow from very generic, non-square block sizes. More specifically, the blocks of A will be `M_BLOCK_SIZE`$\times$ `K_BLOCK_SIZE` and the blocks of B will be of size `K_BLOCK_SIZE` $\times$ `M_BLOCK_SIZE`.
2. As before, store A in row-major order.
3. As before, pad the inner dimension of A and B to be a multiple of 8.

For each block of A and each block of B, we compute the product of these two blocks by multiplying thin slices of A by thin slices of B. The larger the slices, the better, because reads from main memory are reduced. The largest slices we were able to multiply profitably were $2 \times K$ by $K \times 6$, i.e. a 2×6 matrix multiplication microkernel. We strided the microkernel over the block of C, and used smaller matrices at the border. In other words, this corresponds to **a second level of blocking**. The code to stride over the block of C was automatically generated. With this, we were able to achieve an average peak performance of ~ 21.0.

Overall, we concluded that using SIMD matrix multiplication microkernels together with blocking provides significant performance improvements, and together place us within the target $10\% - 30\%$ peak perf.

Of course, it cannot be understated how important it was to **tune the block sizes**. We ended up using blocks of sizes 32×128 for A and 128×32 for B.

5 SIMD matrix multiplication microkernel via outer products (peak perf ~ 21.4)

Unfortunately, the main issue with the approach in Section 4 is that it requires *too many* AVX512 registers when reducing the 8 numbers in the accumulator into 1. Indeed, it has one accumulator per entry in the resulting matrix! This prohibits us to increase the kernel size to increase parallelism, since there are not enough registers. Because of this, we next looked to improve our microkernel. Concretely, instead of computing the matrix product of X and Y by evaluating all inner products at the same time, we looked into accumulating the **outer products** of X and Y, as shown in Figure 2 from [2].

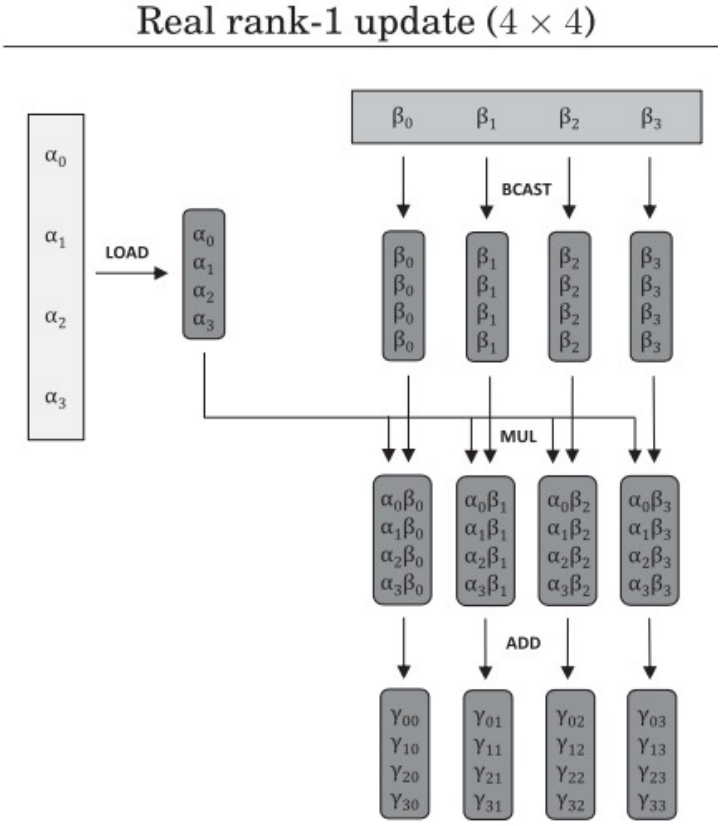

Figure 2: Outer product calculation logic

This approach enables us to increase the kernel size. The optimal kernel size we found for this approach is 8×5, which is more than three times as large as the 6×2 kernel we used in Section 4. The code of the microkernel of this approach is as follows:

```
void matmul_8_by_5_microkernel(
    double* A_topleft,
    double* B_topleft,
    int K,
    int A_nrows,
    int A_ncols,
    int B_nrows,
```

```
    int B_ncols,
    double* res
    ){
    __m512d A, acum_0, acum_1, acum_2, acum_3, acum_4, B_0, B_1, B_2, B_3, B_4;
    acum_0 = _mm512_load_pd(res + 0 * 8);
    acum_1 = _mm512_load_pd(res + 1 * 8);
    acum_2 = _mm512_load_pd(res + 2 * 8);
    acum_3 = _mm512_load_pd(res + 3 * 8);
    acum_4 = _mm512_load_pd(res + 4 * 8);

    for(int k = 0; k < K; k += 1){
        A = _mm512_load_pd(A_topleft + k * 8);
        B_0 = _mm512_set1_pd(*(B_topleft + L_KERNEL_SIZE * k + 0));
        B_1 = _mm512_set1_pd(*(B_topleft + L_KERNEL_SIZE * k + 1));
        B_2 = _mm512_set1_pd(*(B_topleft + L_KERNEL_SIZE * k + 2));
        B_3 = _mm512_set1_pd(*(B_topleft + L_KERNEL_SIZE * k + 3));
        B_4 = _mm512_set1_pd(*(B_topleft + L_KERNEL_SIZE * k + 4));
        acum_0 = _mm512_fmadd_pd(A, B_0, acum_0);
        acum_1 = _mm512_fmadd_pd(A, B_1, acum_1);
        acum_2 = _mm512_fmadd_pd(A, B_2, acum_2);
        acum_3 = _mm512_fmadd_pd(A, B_3, acum_3);
        acum_4 = _mm512_fmadd_pd(A, B_4, acum_4);
    }
    _mm512_store_pd(res + 0 * 8, acum_0);
    _mm512_store_pd(res + 1 * 8, acum_1);
    _mm512_store_pd(res + 2 * 8, acum_2);
    _mm512_store_pd(res + 3 * 8, acum_3);
    _mm512_store_pd(res + 4 * 8, acum_4);
}
```

With this approach, we were able to obtain a peak perf of ~ 21.4, which is just a bit better than our previous small 2×6 microkernel. What might be going on? We figured that our new microkernel traverses the data in A and B in an order that is not convenient for column-major order. **Therefore, we next looked to repacking.**

6 Repacking (peak perf ~ 23.7)

In Section 5, all matrices are stored in column-major format, which does not align with the order which the elements are accessed in the algorithm. In order to make the algorithm more **cache friendly**, we repack the cells of matrix A and B according the the order they were accessed in the algorithm. Figure 3 from [2] illustrates the repacking order of matrix A and B. Also, we make sure to **align the cells to cache line boundaries** by calling `_mm_malloc(<bytes we want>, 64)`, which makes around 1% difference in performance.

With this approach, we were able to obtain a good improvement, going up to ~ 23.7.

7 Software Pre-Fetching (peak perf ~ 25.4)

Usually, the cache is transparent to the programmer. Software prefetching allows the programmer to have some control over the cache, in particular, to give a hint that a certain position in memory will be accessed soon. We followed the example from **recitation 1** and added software pre-fetching to our code in the microkernel. Because we are computing an 8×16 matrix, our code looks like this:

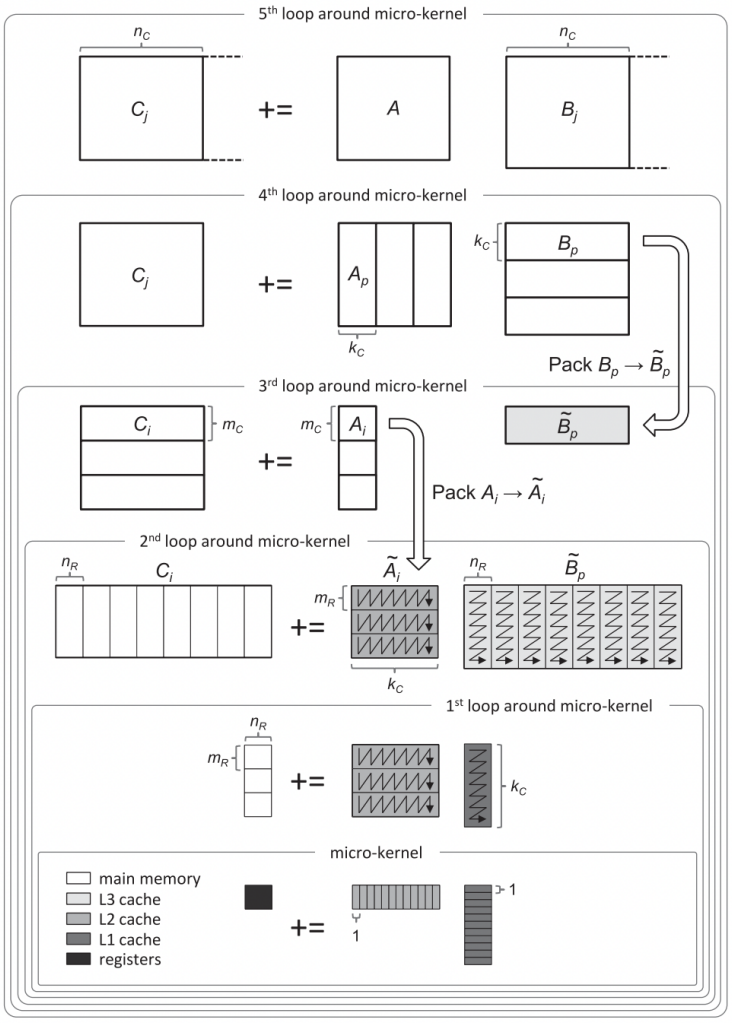

Figure 3: Rapacking logic

```
void matmul_8_by_16_microkernel(
    ...
    ){
    // Create AVX 512 accumulators, etc.
    ...
    for(int k = 0; k < K; k += 1){
        #define PFETCH_DIST 12
        _mm_prefetch(A_topleft + (k + PFETCH_DIST) * 8, _MM_HINT_T0);
        _mm_prefetch(B_topleft + L_KERNEL_SIZE * (k + PFETCH_DIST) + 0, _MM_HINT_T0);
        _mm_prefetch(B_topleft + L_KERNEL_SIZE * (k + PFETCH_DIST) + 8, _MM_HINT_T0);
        // Load A and B, broadcast, crunch data with FMA
        ...
    }
    // Store accumulators in res buffer
    ...
}
```

We tunes the `PFETCH_DIST` manually, and found 12 to be optimal. As mentioned in lecture, `PFETCH_DIST` is

a very sensitive parameter, and it must be neither too small nor too big. Using prefetching, we were able to bump our performance by almost 2.0 points, up to 25.4.

8 Final Optimizations (peak perf ~ 28.6)

After implementing our fast matrix multiplication microkernel, repacking, prefetching, we set out to retune our block sizes. Surprisingly, we found that choosing a very large `M_BLOCK_SIZE`, `N_BLOCK_SIZE`, and `K_BLOCK_SIZE` led to best performance. We set them all to 2048, which means that we completely removed the outer level of blocking, using only the **microkernel level of blocking** (which is striding over C in matrices of sizes 8×16). With this, we reached ~ 26.8. This means that we could expect further improvements by better leveraging the L1/L2 cache. Finally, we noted that we could **read and write from C directly to registers**: instead of using a buffer and `memcpy`, we read and wrote to AVX512 registers with `_mm512_loadu_pd` and `_mm512_storeu_pd`, bumping us up to 28.6. Note that we need to use the unaligned versions because C is not necessarily aligned. We also needed to be extra careful at the border of C, where we could potentially segfault. Because of this, on the border of C (that is to say, when we are actually updating a mini block of C smaller than 8×16), we just used our previous buffer and `memcpy` strategy. This final boost we achieved reinforces the importance of minimizing memory reads and writes, in this case accomplished by avoiding our intermediate buffer completely, and instead reading and writing C from AVX512 registers directly.

9 Conclusion

In this work, we learnt that there are many different ways to leverage parallelism at the single-thread level, and that they can all be combined to produce substantial speedups at the task of matrix multiplication. We explored multiple levels of blocking, SIMD microkernels, repacking, and software prefetching. By building on all these, we obtained an average peak perf of ~ 28.6. Our biggest learning is that **minimizing the number of slow reads and writes from main memory (i.e. cache misses) is of fundamental importance**. All our strategies can be viewed as centered on this idea. Even the SIMD microkernel: at first glance it seems like its sole purpose is to accelerate arithmetic operations, but this is only half true. Indeed, **AVX512 registers also allow us to hold small chunks of A and B in registers at the same time to perform matrix multiplication, eliminating memory accesses by working directly in the registers**.

10 Member Contributions

Po-Yen implemented the faster SIMD matrix multiplication microkernel together with Numi. Po-Yen implemented repacking. Po-Yen tuned the block sizes. Po-Yen proposed using large matrix multiplication microkernels of size 8×16, found by manual tuning. Po-Yen peer-reviewed Sebastian's code.

Sebastian implemented the initial SIMD dot product microkernel and the first version of the SIMD matrix multiplication microkernel. Sebastian implemented pre-fetching and tuned the pre-fetch distance. Sebastian performed the final hyperparmater optimization.

Numi implemented the faster SIMD matrix multiplication microkernel together with Po-Yen. Numi peer-reviewed all of Sebastian's and Po-Yen's code. Numi proposed repacking.

References

[1] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), may 2008.

[2] Field G. Van Zee and Tyler M. Smith. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Trans. Math. Softw.*, 44(1), jul 2017.