CS 267 - HW1

Pedro Guimarães Martins* Dhanush Nanjunda Reddy†
Jun Tian‡

February 5, 2023

1 Methods

For matrix multiplication optimization, we have tried various strategies to improve the performance. Different methods have different effects on computational efficiency. It is worth mentioning that even though we did not end up adopting all of these methods, we still found them instructive.

1.1 Loop Reordering

In the basic code, there are 3 nested loops. The original order for the nested do-block function was from outer to inner order: i, j, k. We directly tested six different sorting situations and found that the loop order of j, k, i has the best performance. The reason for its effectiveness is it fully utilizes the advantage of spatial locality. Since the matrices in this homework are in column-major order, the loop order of j, k, i can help us visit the matrix elements in turn. Thus in the following parts, we will use j, k, i instead of i, j, k as the loop ordering. In later parts of the assignment, we implemented another set of three nested loops (see Blocking), and there we also took advantage of the same loop reordering approach.

There were two other instances in the code where the loop order was important. This is when we were padding the matrices and then trimming down the C matrix (see Padding). In this case, since C is stored in a column-major order fashion, if we want to access its elements in the most efficient way it is best to have an inner loop over its rows so that we can access data that is contiguous.

*Pedro (pedrogm@berkeley.edu) coded a significant part of the code in C, mainly implemented Micro-Kernel and Repacking, and helped with the report.

†Dhanush (dhanushn@berkeley.edu) also did a lot of programming, mainly implemented Blocking and Padding, and helped with writing the report.

‡Jun (jun_tian@berkeley.edu) wrote this report, generated graphs, evaluated loop reordering and performed some other coding tests.

1.2 Blocking

The original code only provides a single layer of blocking. However, our experiments found that using only one layer of blocking will limit the performance based on loop reordering. In order to fully utilize the L1 and L2 caches, we then implemented multi-level nested blocking in our code. We initially followed the algorithm in [1], which proposes two sets of blocking. One set of blocking is over the three dimensions, and another level is only on the rows of A and columns of B (similar to i and j only). Ultimately, they propose a sixth and last loop inside the kernel.

We first implemented a two level blocking algorithm with six loops. The outer three loops create larger blocks sized for the L2 cache, the middle two loops lead to further splitting to form the Macro-Kernel and the innermost loop is for the Micro-Kernel. However, in our tests, we found that with the two-tier loop for Macro-kernel, the performance was limited and could not exceed 30%. In the end, we found it is much better to use the same structure for L1 cache blocking as the blocking for L2 cache, which is including a further blocking along the columns of A/rows of B (similar to direction k).

Notice that in the final implementation we did not create a local copy of the matrices and explore the proper L1 and L2 cache memory. We tried implementing this approach to be able to actually access the cache, but creating the blocks in memory by making copies of the matrices made the performance worse. Most likely, the performance gains from bringing the blocks into cache memory does not compensate for the computational work required to bring the desired data to the cache in our current implementation.

In the most optimal implementation, we just use blocking to divide the matrix into smaller parts and make the indexing more efficient and easier to access. Nonetheless, even with only updating the pointers, we can get better performance since the loop indices now increment by n_c, for instance (or any other blocking size) rather than in units of 1. Ultimately, this requires fewer logical operations to check if the loop has ended. And the performance can now exceed 40%.

Ultimately, we also performed many tests on the blocking size, as our optimal one was somewhat smaller than we would have expected. Given that the cache of 1 core of the AMD EPYC 7713 CPU is structured to have 32 KB data in L1 cache, 512 KB in L2 cache, and 4 MB in L3 cache, we would expect to fit 4096 doubles in L1, 65536 doubles in L2, and around 524288 doubles in L3. In our case, we have 3 square matrices so to fit a part (block) of each one, the maximum block size can be 36 for the L1 cache, 147 for the L2 cache, and 418 for the L3 cache. Since we only implemented 2 levels of blocking we tried having blocks of size 147 in the first level of loops and blocks of size 36 in the second level of loops, so as to fit the blocks in the L2 and L1 caches respectively. However, the results for this was much worse than other block sizes we tried. After implementing the 8x1 micro-kernel, we experimented with block sizes for the first level that were multiples of 8, such as 24, 48, 72, 96, 120, while keeping the block size of the second level as 8x8. We found that the block size of 24x24

for the first level performed the best, which is surprising given the structure of the CPU cache. So, in the final version of our code, we used block sizes of 24x24 and 8x8 for the first and second levels respectively, as this combination yielded the optimal performance.

1.3 Padding

We implemented padding to make sure the blocking is executed properly without any errors. To do this, we created a minimal size matrix that would include all the given elements but would also fit our block size perfectly (from section 1.2). We did this by filling in the original values at the same positions and zeros at the remaining positions that were beyond the size of the original matrix. This was done for both matrices A and B. This trick might add a little bit of computational burden on a small matrix, but it has almost no effect on a large matrix. With this optimization, we can input a matrix of any size and still take advantage of blocking and our 8x8 kernel, as all input matrices will be transformed into a matrix size that is a multiple of our block size.

1.4 Micro-kernel (SIMD)

Micro-Kernel (with SIMD) is another very important technique that we implemented. The idea is to take advantage of hardware optimized computations, which will significantly increase the computing efficiency. To be more specific, we perform matrix multiplication on the smallest block size. For instance, this would be $m_r \times k_r$ in our implementation for A. Compared to the algorithm we took inspiration from in [1], this would be fitting this small block into the processor registers. Then, instead of using + and $\times$, we use the Intel intrinsic operations '__mm256_xxx', which perform the arithmetic computations in a way highly optimized for the hardware.

Considering that the Perlmutter's processors have a 256-bit vector width, meaning each instruction can operate on 4 64-bit data elements at a time, we implemented a micro-kernel in multiples of 4. Apart from this last constraint, we had many design choices to make.

For a micro-kernel of size $4n \times 1$ we perform outer products between columns of A with $4n$ elements and one element of B, thus populating a column of C with $4n$ elements. We started with a micro-kernel of size 8x1, in which we hard coded this outer product operation for a total of 8 elements. Then, we implemented two loops, one over each of the n_r columns of C/elements in a row of B as well as one over the k_r rows of B/columns of A. Ultimately, we can execute the matrix multiplication on blocks of size $m_r \times k_r$ for A times size $k_r \times n_r$ for B. For reference, here are some other design features we explored, even if ultimately the 8x1 micro-kernel was the best performing one (Further discussion and results under the Experiments section).

1.4.1 Micro-kernel of size $4n \times 1$

First, we optimized the row size in the micro-kernel, in multiples of 4: 4, 8, and 12. The full-performance analysis can be found in the Results section, but ultimately the 8×1 micro-kernel showed the most optimal results.

1.4.2 Micro-kernel of size $4n \times j$: Loop Unrolling

Now, a design feature proposed by [1] includes hard-coding the loop over j, that is, the loop over n_r columns of C/elements in a row of B. We implemented this approach and expanded the size of the micro-kernel hard-coded operations. Ultimately, this means we can achieve a kernel of size $4n \times 8$. Upon implementing this, we found a significant decrease in performance, and did not include the unrolling in our final code.

1.5 Repacking

As spatial locality is highly important for efficiency, we also implemented repacking of our matrices. This strategy consists of changing the order in which we read the elements in the matrix according to the order in which they are used in the algorithm. Since the matrices in this homework are all in column-major order, and an outer product requires a portion of a column of A and an element of B, there will be many non-contiguous memory accesses.

Therefore, we reordered the matrix entries according to [1] so that the data has greater spatial locality and is stored in a contiguous way. We implemented this by creating (this time in the cache) a reordered copy of the matrix, and updated all the following indices appropriately. Given the desired access pattern, we applied the same repacking algorithm to matrix A and the transpose of matrix B. Ultimately, despite performing well (close to 50%), repacking did not lead to further improvements and was slightly worse than doing no repacking at all, so it was not included in the best performing code. See further results and analysis in the Results section.

2 Experiments

2.1 Best performing algorithm

We compared our algorithms with Blas. **Ultimately, our submission can achieve an average performance of 55.28% on the overall benchmark**. Note that the performance is above 60% for large matrices, while it is lower on smaller matrices which brings the average down. This is because large matrices can better utilize the advantages of Blocking and the Micro-Kernel algorithm, while smaller matrices will suffer from the burden brought by some techniques like padding.

As can be seen from the curve in Figure 1, the performance of the algorithm generally improves as the matrix size increases. However, the curve is

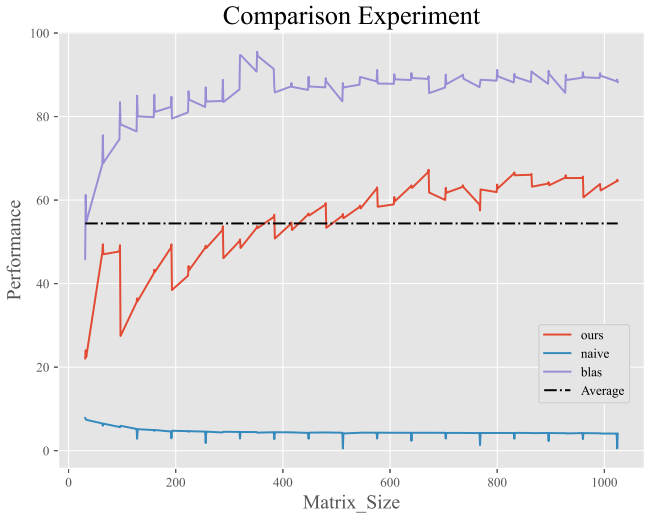

Figure 1: Comparison Results

not smooth and there are many dips. This is mainly caused by the padding algorithm. Since our larger block size is 24, if we have a matrix size of 23 for example, then we only need to pad one additional column and row and impact on the performance will be very little. However, if the matrix size is just beyond the block size, 25 for example, then we have to pad all the way to size 48 which significantly hurts performance. In addition, this decline in performance should be smoother as the matrix size increases, since padding a matrix with size 961 should result in very little additional work in the grand scheme of things, while going from 25 to 48 incurs a much more significant drop in performance.

2.2 SIMD Micro-kernel Design

2.2.1 Micro-kernel of size $4n \times 1$

First, we tested the number of row operations hard-coded in the micro-kernel, still within the structure of a double for loop (note that here we are testing only $4n \times 1$ sizes). While increasing the number of hard-coded operations might be beneficial as it would avoid further iterations over the inner for loops and logical operations involved, it also leads to data management with decreasing spatial locality inside the micro-kernel. While with a 4x1 and 12x1 micro-kernel we achieved 41.5% and 50.5% efficiency respectively (refer to Figure 2), the best performing design was the 8x1 micro-kernel, reaching 55% efficiency.

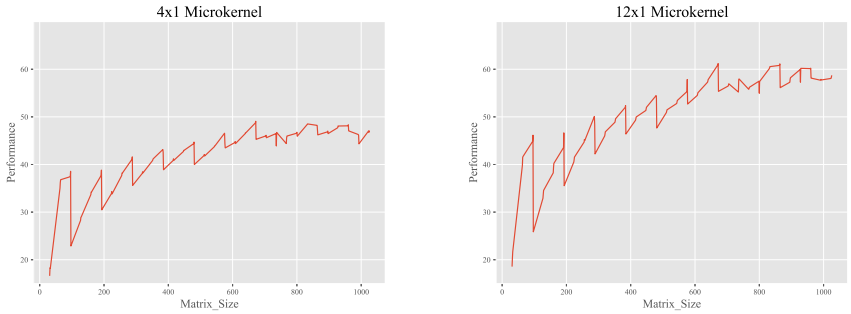

Figure 2: Micro-kernel Design Analysis

2.2.2 Micro-kernel of size $4n \times j$: Loop Unrolling

In Figure 3, we have the performance of the algorithm once we unrolled the loop over the elements in a row of B or the columns of c, which seriously hindered the performance. While unrolling the loop might be helpful, here it causes the data to be accessed in an even less contiguous way, as within the loop we can hop over places of closer proximity.

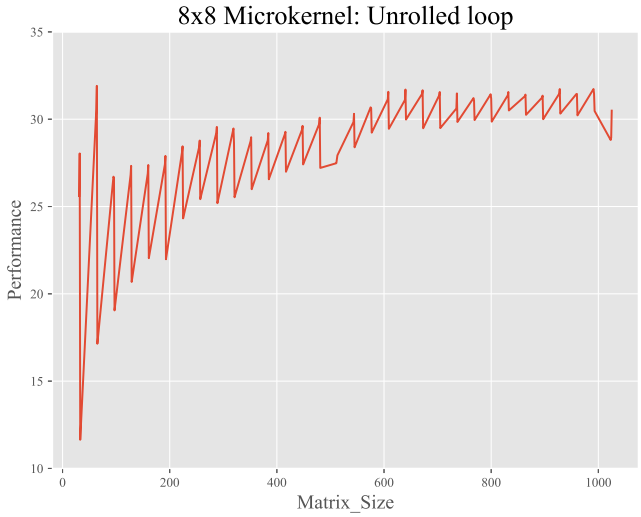

Figure 3: 8x8 Micro-kernel: Unrolled loop Analysis

2.3 Repacking Results

Now, even though repacking was ultimately not included in our best performing algorithm, we were able to implement repacking of A and B that allowed maximum data contiguity. These results are shown in the panel graph in Figure 4. We were able to achieve overall averages close to the approach without repacking, which was in the low 50%s. With this implementation of repacking, we can achieve even more computational efficiency as we retrieve data with greater locality in the micro-kernel. After packing, the column index is much smaller, so there is less hopping around in memory. Nonetheless, these gains are shadowed by the additional computational work to create a local copy of the repacked matrices, particularly for B as further computations are required to generate the transpose.

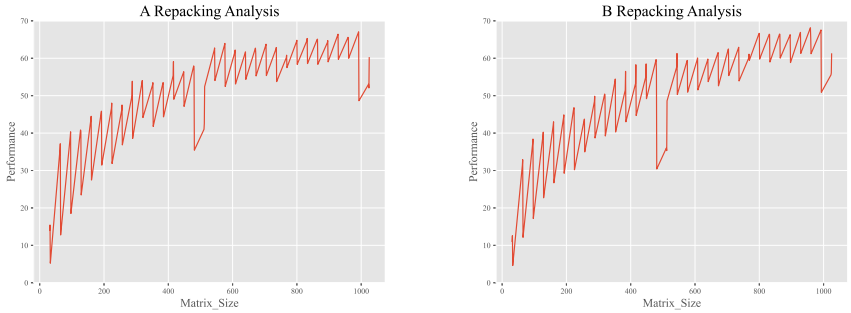

Figure 4: Repacking Analysis

One interesting consequence of repacking is that we see less continuity in the average efficiency over size, and the data has much stronger fluctuations. This is caused by the same phenomena that explained the original spikes in the curve. But now, we not only have significant losses in efficiency with sizes just above a multiple of 24 (due to padding), but also for sizes just above multiples of 8, since this is the base size for ordering in repacking (and the micro-kernel). As a consequence, we perceive further fluctuations as the size increases.

Another interesting feature is a major dip at size 481, which can be seen in both repacked A and B. Notice here that when repacking the matrices we create a local copy in the cache, and therefore, can take advantage of this faster memory access to perform computations. Nonetheless, the cache size is limited, and once we run out of space, the algorithm may encounter cache misses when retrieving data. At that point, the system has to recover local memory, which is considerably more expensive. Notice that this dip continues through the next few sizes, and only after size 513 the other strategies implemented here can make up for this additional data fetching cost and get back to efficiency higher than 55%. We also implemented repacking both matrices simultaneously, but this approach led to even worse results.

Upon data with greater spatial locality, we tested what would happen if we

unroll the micro-kernel in addition to repacking. The idea behind this strategy is that now, with greater spatial locality of the data, the micro-kernel can operate more efficiently without a for loop but rather sequential loading, as the data is already stored contiguously. The results of those tests are in Figure 5, and overall these tests achieved very low averages in the high 20%s. Ultimately, it seems that the implementation with a small micro-kernel 8x1 was able to retrieve information from memory more efficiently with the raw matrices.

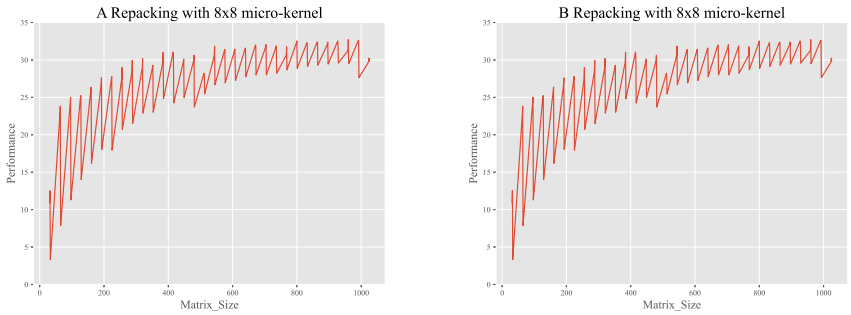

Figure 5: Repacking with 8x8 Micro-Kernel

3 Conclusion

In this homework, we explored the various optimization strategies of matrix multiplication. We leverage the multi-layer blocking algorithm with well-designed micro-kernel and many other techniques to help us achieve a better performance on the benchmark than the naive algorithm. Compared with the naive edition code, our framework not only performs better, but is also more robust for larger size matrices.

References

[1] Jianyu Huang, Robert A. van de Geijn. 2016. BLISlab.: A Sandbox for Optimizing GEMM.