Optimizing Matrix Multiplication On NERSC's High Performance Computer Cori

Xuan Jiang[*], William Lin[†], Ansh Nagda[‡]

1 Introduction

We use a variety of techniques to optimize the single-threaded operation of:

$$C = C + A \cdot B$$

A number of optimizations techniques yielded significant speedups, including multi-level blocking, copy optimizations, and loop adjustments. However, we also tried a number of other optimizations that did not improve our program, including prefetching.

All of our team members contributed to writing/testing various optimizations, collecting results, and writing the report. More specifically, Xuan implemented prefetching with different distances and hints, loop unrolling, inline function, and attempted multi-level blocking and SIMD. William implemented single-level blocking, copy optimizations, an optimized 8x8 matrix multiply function taking advantage of instruction-level parallelism, and dynamic padding. Ansh implemented multi-level blocking, copy optimizations for multi-level blocking, and the SIMD micro-kernel.

In the remainder of this report, we describe each of our optimizations in our final submission, present results with evidence that they work, and describe attempted optimizations that did not noticeably improve overall our performance. Unless otherwise noted, all of our benchmarks operate on matrices of size 1024x1024.

2 Optimizations

2.1 Blocking

2.1.1 Single Level

We split up the matrix multiplication operation into a series of smaller matrix multiplications to optimize the cache usage(Lam et al., 1991). We divide the A, B, and C matrices into sub-matrices, such that three sub-matrices can fit in the L1 cache. When performing the small matrix multiplications on the sub-matrices, the computation avoids unnecessary accesses to the L2 cache or to main memory after initially bringing the elements into L1 cache. As a result, across the entire

[*]j503440616@berkeley.edu
[†]will.lin@berkeley.edu
[‡]anshnagda@berkeley.edu

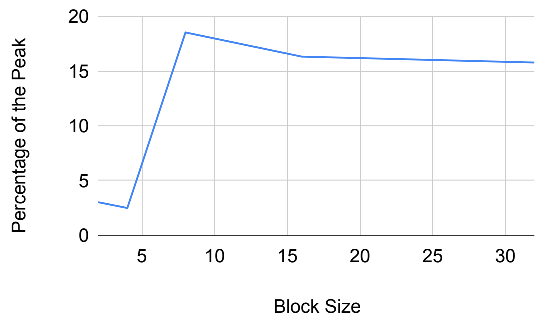

Figure 1: Single-level block matrix multiply on 128x128 matrices.

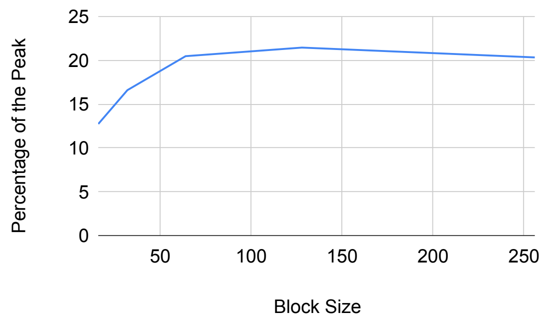

Figure 2: Multiple-level block matrix multiply with a small block size of 8x8.

matrix multiplication operation, we are able to better utilize the L1 cache compared to the naive matrix multiplication with three loops.

The size of the L1 cache on the CPU is 32KiB and each value in the matrix is 8B (for double precision numbers). For simplicity, we use square sub-matrices of size `dim` x `dim`. We compute the maximum size of each sub-matrix to ensure that three of them can fit in the L1 cache:

$$3 \cdot 8B \cdot \texttt{dim}^2 \leq 32 \cdot 2^{10}B$$
$$\texttt{dim}^2 \leq \frac{32}{24} \cdot 2^{10}$$
$$\texttt{dim} \leq 36$$

However, because the L1 cache can be filled up with other data, we do not want to operate at the theoretical limit to avoid unnecessary cache evictions. We expect the ideal block size to be around or less than the theoretical limit. We experimentally determine the optimal block size by performing tiled matrix multiplication operations using 128x128 matrices, while varying the size of the blocks. Within each block, we use a naive three-loop matrix multiply implementation. Figure 1 demonstrates the results of this experiment. Based on these findings, a block size of 8x8 performs the best.

2.1.2 Micro-Kernel

We further optimize the implementation of our 8x8 block matrix multiply with a SIMD register-blocked kernel. Our implementation takes inspiration from the following "outer product" representation of matrix multiplication, where A^i and B_j are the i^{th} column of A and j^{th} row of B respectively:

$$A \cdot B = \sum_{k=1}^{n} A^k B_k^T \tag{1}$$

For each k, we are able to compute a column of the outer product $A^k B_k^T$ using a single fused multiply-add AVX intrinsic. The CPU has two vector lanes and can process two vector instructions at the same time. By allocating different vector registers for each column, we can fully take advantage of instruction-level parallelism by placing the fused multiply-add instructions in different cache lines next to each other. By making explicit use of the vector registers, we are also able to better optimize their utilization, instead of relying on the compiler to do this automatically in the case of the naive fixed-size three-loop approach.

Since each vector register stores 8 doubles, our micro-kernel needs 8 vector registers for C. For each k, requires 1 vector register to store A^k, and 8 vector registers to store different entries of B_k^T. By reusing registers across different k, the total number of vector registers we need is $8 + 8 + 1 = 17$, which is less than the 32 vector registers on the machine. Figure 3 demonstrates the improvement the micro-kernel achieves over the naive three-loop implementation of the outer product representation in Eq. 1.

2.1.3 Multiple Levels

For larger matrix sizes, we use similar ideas of blocking to optimize accesses to the L2 cache by avoiding unnecessary accesses to main memory. We take a layered blocking approach: we divide the A, B, C matrices into large blocks, such that three blocks can fit in the L2 cache. We then split each of the large blocks into smaller blocks that fit into L1 cache, as described in the previous section. In accordance with the previous experimental results, we set these smaller blocks to be 8x8.

The size of the L2 cache on the CPU is 1MiB. We compute the theoretical maximum size of each block:

$$3 \cdot 8B \cdot \mathtt{dim}^2 \leq 2^{20}B$$
$$\mathtt{dim}^2 \leq \frac{2^{20}}{24}$$
$$\mathtt{dim} \leq 209$$

We again experimentally determine what block size to set to target the L2 cache and expect the ideal block size to be around or less than the theoretical limit. From Figure 2, we see that block sizes of 128x128 appear to achieve the highest performance for 1024x1024 matrices. However, we also observe that there is some flexibility in block sizes; the performances using block sizes of 64x64 and 256x256 are not significantly lower than that achieved by using a block size of 128x128. We will use this idea later on when addressing the problem of performing operations on matrices that cannot be cleanly divided into a pre-configured block size.

2.2 Copy Optimization

We copy and rearrange the A, B, and C matrices into temporary buffers to optimize the benefits of spatial locality. We perform the operations on the temporary buffers before copying the results back into the final output matrix. These copy operations add an overhead of $O(n^2)$ to a matrix multiplication operation costing $O(n^3)$, so the initial overheads are acceptable for large enough matrices.

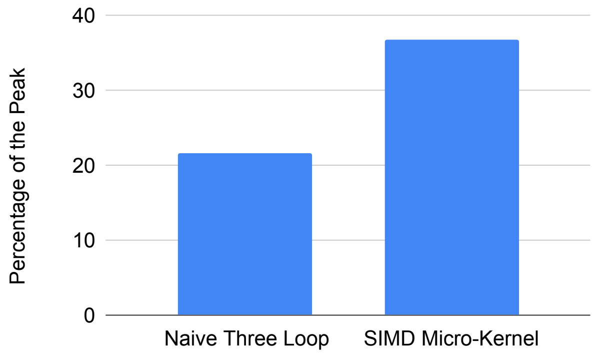

Figure 3: Performance comparison between naive three-loop matrix multiply vs SIMD micro-kernel.

2.2.1 Column-Major to Row-Major

Accesses to the A matrix are typically performed row-by-row. In the case of matrix tiling, elements within a block are accessed in a row-by-row manner. The original A matrix is in column-major order, meaning that accessing a row of A requires crossing a cache line per row element. By copying the A matrix into a new buffer to follow a row-major order, we more effectively exploit spatial locality. Accessing one element in a row of A will bring the next few row elements in the cache line. As a result, we make fewer accesses to slow memory.

2.2.2 Access-based Reordering

We can take the idea of spatial locality further by specializing it to our algorithm. The access pattern of multiple-level blocking is predictable. Specifically, each block of A is accessed in row-major order, while each block of B or C is accessed in column-major order. Similarly, each sub-block within a block of A is accessed in row-major order, while each sub-block within a block of B or C is accessed in column-major order. We rearrange the A, B, and C matrices such that large and small blocks are stored sequentially in the order they are accessed. The reordering allows use to address other factors that may negatively affect performance, like avoidable translation lookaside buffer (TLB) misses or early cache evictions due to cache conflicts between elements in a block.

2.2.3 Realignment to Cache Lines

We allocate the temporary buffers to be aligned on cache lines. The size of our cache line is 64B, which means that a single memory access can bring eight matrix elements into the cache. Because the size of our sub-blocks are 8x8 matrices and they are stored sequentially in memory, cache line alignment ensures that sub-blocks can be read into cache with at most eight accesses to slow memory. As a result, we are able to maximize the benefits of spatial locality. Alignment also enables us to use the faster aligned versions of SIMD instructions.

2.3 Padding

We pad our matrices to be a multiple of our block size, which allows us to cleanly divide operations with arbitrary matrix sizes into sub-operations. However, this results in an increase in the size of

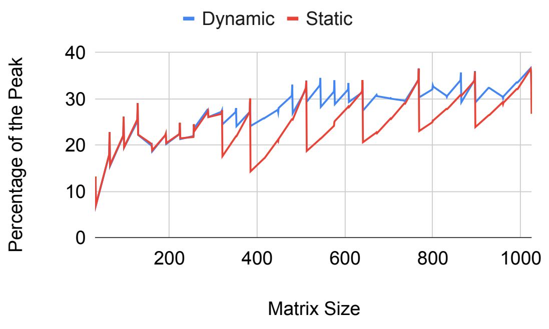

Figure 4: Performance comparison between matrix multiply using static or dynamic padding.

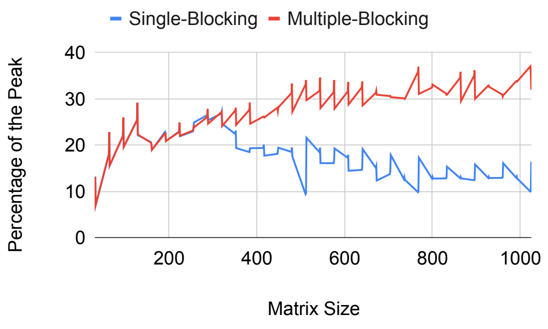

Figure 5: Performance comparison between single-level blocking and multiple-level blocking.

the matrix and necessarily impacts performance. We employ a number of techniques to reduce the amount of padding needed.

2.3.1 Sub-Block Padding

For smaller matrices, we do not pad to the large block sizes (128x128) but rather pad to the smaller sub-block size (8x8) and use single-level blocking. This minimizes the additional computation when padding smaller matrices, as the matrix dimensions is increased by at most 7 elements.

2.3.2 Dynamic Block Sizes

For larger matrix sizes, we use larger block sizes that target the L2 cache. However, the issue is that these block sizes can add significantly more computation that is not needed by the final output. To address this issue, we use our previous observation that the larger block size can be somewhat flexible without impacting performance too much. We can dynamically adjust the block size to minimize the amount of padding necessary and reduce unnecessary operations.

For instance, suppose we have matrices of size 513x513. We do not want to simply pad to the nearest static block size of 128, as doing so would change the computation to operate on 640x640 matrices. Instead, if we pad to a dynamic block size of 136, we only need to operate on 544x544 matrices, which is a significant improvement.

Figure 4 demonstrates the result of this optimization compared to static padding. We can see that the performance drops from padding is less significant with dynamic padding compared to static padding.

2.3.3 Switching Padding Behavior

We switch between sub-block padding (with single-level blocking) and dynamic block padding (with multiple-level blocking) to balance having fewer padded elements with the benefits of multiple-level blocking. To determine at what point we should change the behavior, we compare the performances of the two techniques on different matrix sizes. Figure 5 demonstrates the results of this experiment. We observe that after a block size of 320, multi-level blocking performs better than single-level

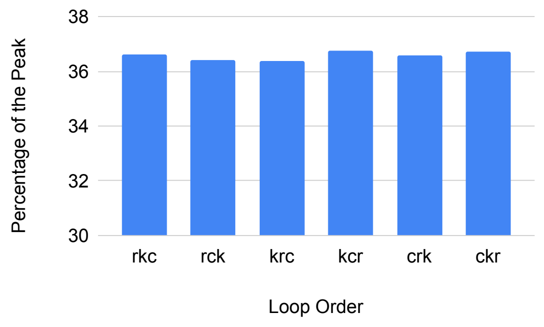

Figure 6: Performance comparison of outer loop orders.

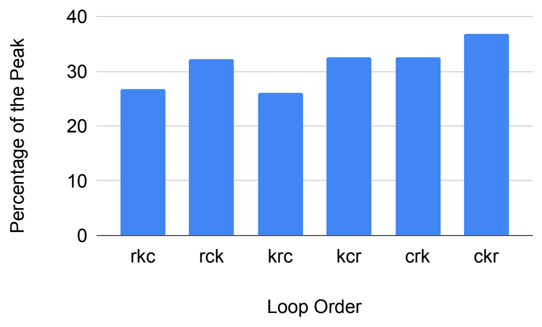

Figure 7: Performance comparison of L2 block loop orders.

blocking. In our final implementation, we switch between single-level and multi-level blocking at this threshold.

2.4 Optimizing Loops

2.4.1 Loop Reordering

We order the loops at each layer according to the permutation that gives us maximum performance. The blocks of matrices A are accessed in row-major order, while the blocks of matrices B and C are accessed in a column-major order. Similarly, sub-blocks of an A block are accessed in row-major order, while the sub-blocks of B and C blocks are accessed in column-major order.

We experimentally determine the ideal ordering of loop order to access the blocks. We denote a loop order by any permutation of the sequence "c-r-k", where r denotes the row of A and C, c denotes the column of B and C, and k denotes the current column or row of A or B respectively.

Figures 6 and 7 show how the performance varies when different loop orders are used. To maximize performance, we found that an outer loop ordering of k-c-r and a L2 block loop ordering of c-k-r performed the best. These ordering makes sense, as they tend to favor reusing blocks already in the L2 or L1 cache, rather than evicting them for another block.

2.4.2 Loop Unrolling

We use constants and macros in place of variables wherever possible to help the GCC compiler optimize our code. Doing so has allowed the compiler to add loop unrolling to our micro-kernel for 8x8 matrix multiply. We are able to avoid branching, despite using a single loop, without needing to explicitly unroll every loop ourselves. Figure 8 demonstrates how loop unrolling in our micro-kernel has helped with performance.

2.5 Miscellaneous

2.5.1 Separating Padding and Non-Padding Logic

The need to pad matrices adds additional overhead to the copy-optimization functions, because they may require additional branching or extra copy operations. For matrices that do not require

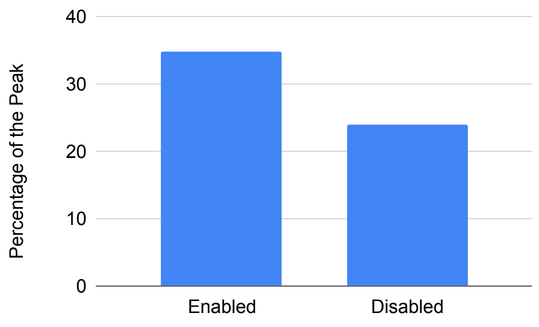

Figure 8: Performance comparison of enabling or disable loop unrolling in the SIMD micro-kernel.

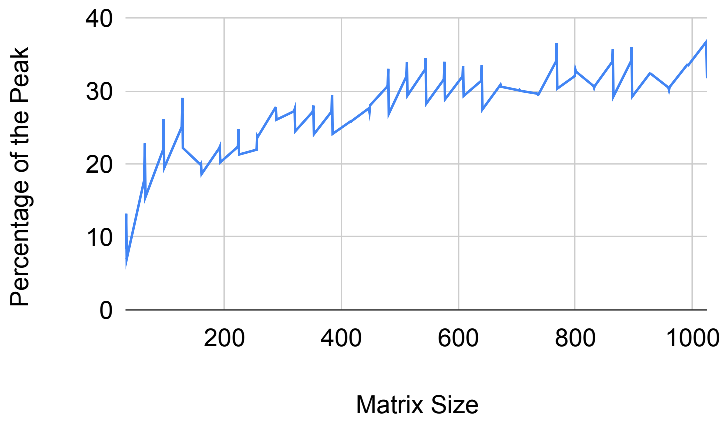

Figure 9: Performance of matrix multiplications after optimizations.

padding, these overheads add unnecessary performance losses. To avoid such situations, we create two versions of copy optimization functions, one to support padding and one that assumes padding is not necessary. We are able to see a slight performance bump for matrix sizes that do not require padding as a result.

3 Results

Running our matrix multiplication implementation on the provided test sizes, we achieve an average percentage of peak across all matrix sizes at around 28%. Figure 9 shows the results of our benchmarks.

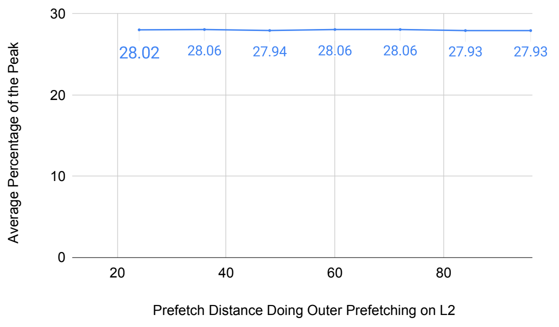

Figure 10: Performance with different prefetching sizes for outer blocks

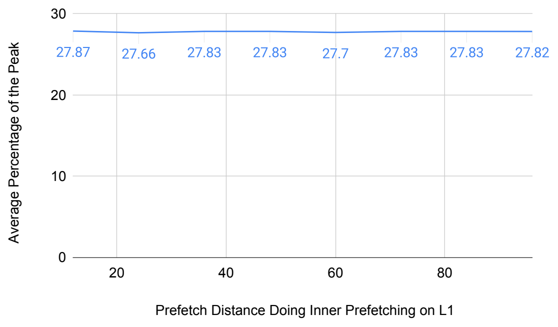

Figure 11: Performance with different prefetching sizes for inner blocks

3.1 Sawtooth Behavior

We observe a sawtooth behavior in our results. The reason for this behavior is that we are running the operations on matrix sizes $32n \pm 1$. A minimum requirement for a matrix to not need padding is that its dimension must be a multiple of 8, since our sub-block size is 8x8. For matrix sizes of $32n$, we *may* not need to do any padding, as it is already a multiple of 8. For matrix sizes of $32n - 1$, we will always require a padding of 1 element, so there will always be a drop in performance relative to matrix sizes of $32n$. For matrix sizes of $32n + 1$, we will must pad the matrix dimensions by at least 7 elements, so the drop in performance is more significant. This behavior causes the sawtooth behavior that we see throughout the performance graph.

4 Other Optimization Attempts

In this section, we describe our optimization attempts that did not make it into our final submission due to performance losses or lack of significant change in performance.

4.1 Prefetching

Ideally, proper prefetching should minimize the number of cache misses. It is essential to prefetch memory locations at the right distance – if the distance is too short, the program will be executed before the values arrive in the cache. If the distance is too long, it is likely that the values will be evicted from the cache before the program gets to them.

We attempt to add L1 cache prefetching and L2 cache prefetching into our inner and outer loops, while varying the prefetching distance. The results of doing so are shown in Figure 10 and Figure 11. The results indicate that for our code, prefetching does not seem to help much with performance and may at times hurt performance, so we have opted to not use prefetching.

4.2 Kernel using Alternate Formulation of Matrix Multiplication

We try single level blocking targeting the L2 cache with blocks of size 128x128. We use SIMD instructions to write a kernel inspired by the "inner product" formulation of matrix multiplication:

$$(A \cdot B)_{i,j} = A_i^T B^j,$$

where A_i and B^j are the i^{th} row of A and j^{th} column of B respectively. Then, the computation of $A_i^T B^j$ would involve sequentially doing a fused multiply-add on 8 pairs of vector registers, which might lead to performance increases through instruction level parallelism. This approach gives us a performance of 15% on 1024x1024 matrices, as opposed to 35% that we get from the "outer product"-based 8x8 micro-kernel implementation of Section 2.1.2.

5 Conclusion

In our attempts at optimizing matrix multiplication, we find that taking full advantage of spatial locality (through techniques like loop rearrangement or copy optimizations) and instruction-level parallelism greatly improves the performance of our operation. Furthermore, techniques like prefetching can be a double-edged sword; it can help performance when used correctly but can slow down performance as well. Finally, we find that the compiler adds optimizations out-of-the-box, like loop-unrolling, that help with performance underneath.

6 References

References

Lam, M. D., Rothberg, E. E., and Wolf, M. E. (1991). The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74.