Esther Matthew, Mengqi Zou, Karina Lee
CS267 Spring 2025
Homework 1
February 10th, 2025

Team Members and Contributions
The team is composed of Mengqi Zou, Esther Mathew, and Karina Lee. Mengqi contributed to code implementation for loop reordering, SIMD, and multi-level blocking. Esther worked on testing block sizes and implementing multilevel blocking in the dgemm_blocked code. She also wrote the Blocked (Tiled) Matrix Multiply and Testing Block Sizes and Multi-Level Blocking sections of the paper. Karina worked on implementing SIMD. She also contributed to the loop reordering, SIMD, and optimized code sections of the paper.

Blocked (Tiled) Matrix Multiply and Testing Block Sizes
The first optimization that we tried starting from the given dgemm_blocked code was to test different block sizes. Blocked or tiled matrix multiply optimizes memory access patterns and increases cache efficiency by working on smaller submatrices, or tiles, that fit into fast cache memory.

The naive approach to matrix multiplication involves three nested loops with time complexity O(N^3). Also, matrices are stored in row-major order, but accessing values of a matrix in the inner loop sometimes leads to cache misses due to non-contiguous memory access. For large matrices, the matrices may not fit in cache which leads to excessive memory transfers between different levels of memory hierarchy.

In blocked matrix multiply, each NxN matrix is broken down into smaller bxb tiles. These tiles are then multiplied independently. Instead of accessing entire rows and columns from large matrices and trying to put them into cache, the smaller tiles fit into cache which reduces cache misses. The same data that is in the tile gets reused multiple times before being evicted from cache, preventing data from being loaded multiple times from main memory and reducing memory traffic. Microkernels using SIMD benefit from working on tiles that fit in registers.

We start with a single level of blocking and try to find an optimal block size for blocked matrix multiply. Since this method involves fitting the matrices into levels of cache, we calculated the maximum block size to have each tile's data fit into L1 and L2 cache. We first break the problem into smaller matrix multiplication, or multiplication of mxk and kxn blocks shown in Figure 1.

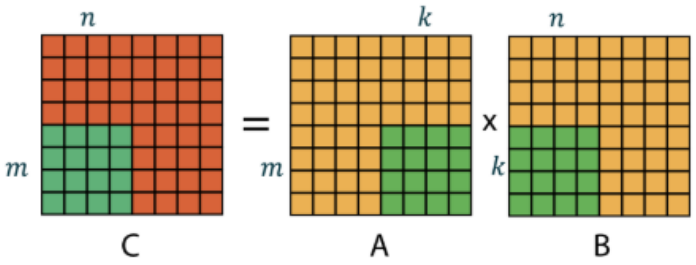

Figure 1. Dimensions of smaller tiles to determine block sizes for tiled matrix multiply.

We then choose a cache level to target (L2, L1) and compute the total number of data words *D* that it can hold. We want to choose block size such that *mn+mk+kn <= C.* The formulas below show how we calculated maximum block size for L2 and L1 cache. The L1 and L2 cache sizes are in the linked reference.

$$dataWordSize \;=\; 8 \;-->\; size\ of\ Double\ data\ type\ in\ bytes$$

Targeting L2 cache size (512 KiB), find total data words that L2 cache can hold:

$$512\ KiB \;=\; 524288\ bytes$$

$$D \;=\; 524288\ /8 \;=\; 65536$$

Assuming m, n, and k are equal (square matrices):

$$sqrt(65536\ /\ 3) \;=\; 147.8 \;-->\; max\ block\ size\ to\ target\ L2\ cache$$

Targeting L1 cache size (32 KiB), find total data words that L1 cache can hold:

$$32\ KiB \;=\; 32768\ bytes$$

$$D \;=\; 32768\ /8 \;=\; 4096$$

Assuming m, n, and k are equal (square matrices):

$$sqrt(4096\ /\ 3) \;=\; 36.95 \;-->\; max\ block\ size\ to\ target\ L1\ cache$$

This tells us that if we are targeting L1 or L2 cache, the block sizes that give the largest amount of data that can be contained in the caches are 147 and 36, respectively. Given these constraints, we tested block sizes from 16 to 128. The performance of each tested block size with varying matrix sizes is shown in Figure 2.

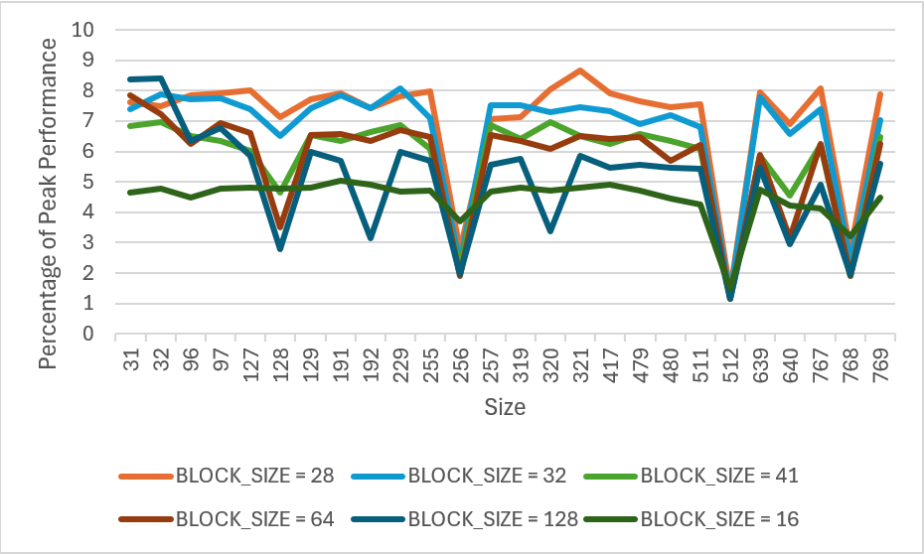

Figure 2. Performance of block sizes with single-level tiled matrix multiply.

We observed that block sizes of 28 and 32 had the best performance across matrix sizes. This aligns with our maximum block size when targeting L1 cache by being slightly less than 36. The larger block sizes like 64 and 128 have generally worse performance when compared to smaller block sizes, likely due to exceeding cache capacity and the resulting cache evictions. The block size of 16 had the worst performance. Since it is a lot smaller than the maximum L1 block size, it may not be effectively utilizing the cache which leads to excessive memory accesses.

There are sharp performance drops at certain matrix sizes like 256 and 512. This could be due to cache associativity limits where certain sizes cause more frequent cache conflicts and evictions. If the matrix size does not align well with the cache's associativity, or how memory addresses are mapped to cache lines, the rows or columns of the matrix could keep overwriting each other and lead to excessive memory accesses from main memory.

Repacking blocked portions into a new memory location in a contiguous order would help optimize the memory layout to align with memory access patterns. Currently, the matrices are stored in column-major order, but data is accessed row-by-row. Repacking the matrices to row-major order prevents the processor from having to access dispersed data, preventing cache misses. Our team attempted to implement repacking to achieve this end, but we could not implement it successfully before the submission deadline.

Multi-Level Blocking

Creating multiple levels of smaller tiles can help us take advantage of multiple types of caches, specifically L1 and L2. This method ensures that each cache level is utilized fully before accessing main memory. The inner block size should target the L1 cache and the outer block size should target the L2 cache. This is because the larger tile is further split into smaller tiles, and the

computations are done in a microkernel on the smaller tiles. The larger tiles would need a larger block size which means that we should target L2 cache in the outer level. Similarly, smaller tiles need a smaller block size which would mean targeting the L1 cache in the inner level.

We tested multilevel blocking with L2 blocking and L1 blocking where we target the L2 cache with a larger block size and the L1 cache with a smaller block size. As we can see in the SIMD and Optimized Code sections, using an inner L1 block size of 32 and an outer L2 block size of 128 resulted in the best performance.

Loop Reordering

Loop reordering was employed to increase performance in the do_block function. The original implementation iterates through the rows of A, columns of B, and the inner dimension k. This approach is inefficient and results in cache inefficiency. Accessing k in the inner loop results in larger jumps in memory for matrix A and disrupts cache locality. Furthermore, the variable in matrix C is loaded and stored with each interaction resulting in extra memory access overhead and increased latency. Our solution to improve performance is to reorder the loops such that the code iterates over the inner dimension k, followed by columns of B, and the rows of A. We are can do this because all the calculations occur in one step and do not have any loop dependencies. This increased performance from 6.23% to 20.18% (Figure 3). We attribute this performance increase to the spatial locality where the destination registry is different for each iteration, eliminating the data dependencies. This allows instructions to run in parallel and out of order. Additionally, a single row stays in the registers, while B is accessed contiguously, making the code more cache-friendly.

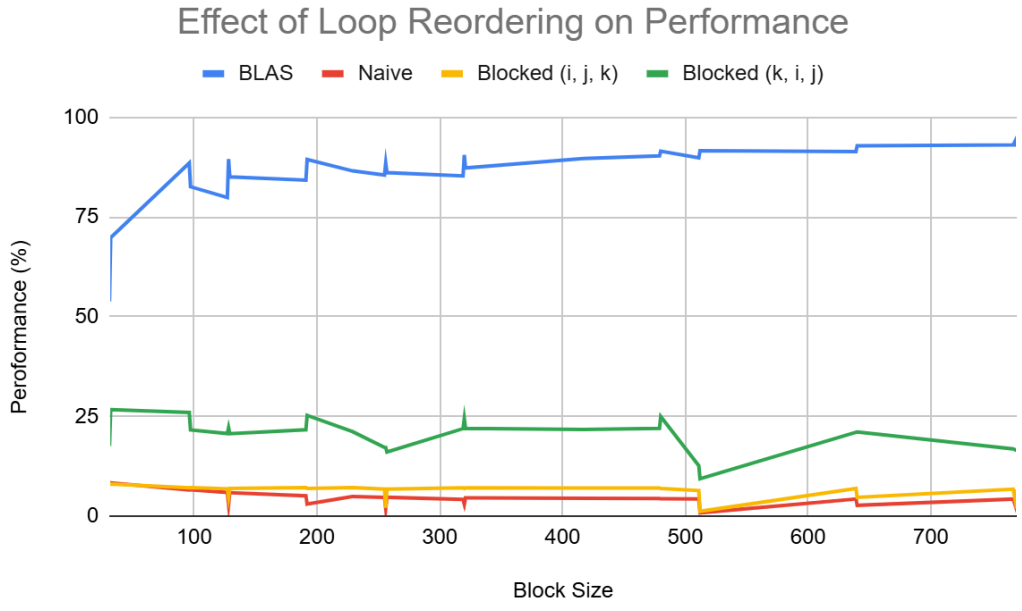

Figure 3. Effect of Loop Reordering on Blocked Matrix Multiplication

SIMD

Single Instruction, Multiple Data (SIMD) was employed to increase the performance of our blocked matrix multiplication on the Perlmutter machine using a 256-AVX instruction set. SIMD allows the processor to perform the same operation on multiple data elements in parallel and increase throughput. Initially, we introduced parallelism into the original code with a step size of 4 which achieved an average performance of 29.74% (Figure 4). We attribute the increase in performance to vectorization in which the SIMD instructions allow the processor to process 4 double precision floating point numbers simultaneously. This allows the CPU to perform more operations in a single cycle, significantly increasing the time needed to perform the matrix multiplication. Next, we increased the step size to 8 to further optimize memory access and observed an average performance of 41.10% (Figure 4). A step size of 8 allows each iteration to load a 256-bit block of data from memory which aligns the data access with SIMD registers. This ensures the data in the cache is continuous, improving spatial locality.

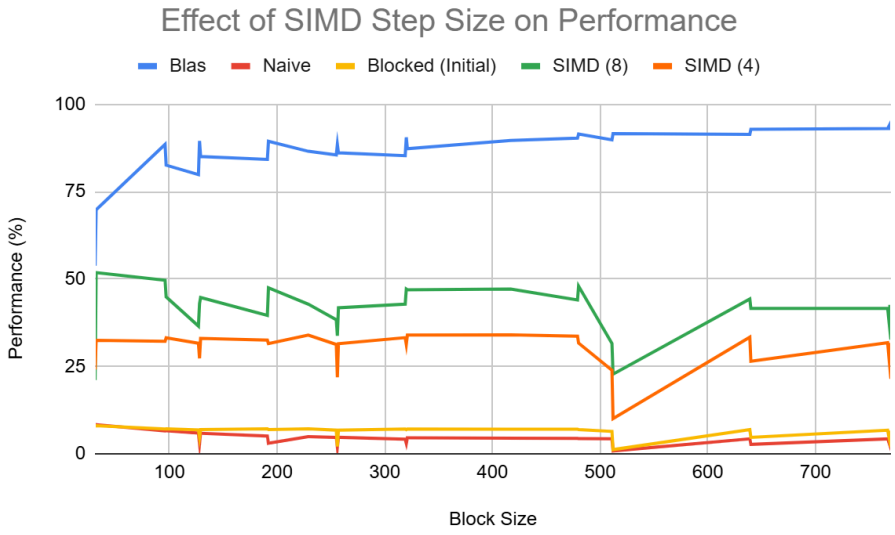

Figure 4. Effect of SIMD Step Size on Performance

Optimized Code

When we combined the SIMD instructions with a step size of 8 and combined with multilevel blocking we achieved an average performance of 45% (Figure 5). This approach leveraged multi-level blocking with block sizes of 32 and 128. This allows us to target L1 and L2 cache, ensuring the data can be reused efficiently and minimize memory latency. The 32-block size fits into the L1 cache, allowing frequent access while the 128-block size fits in the L2, increasing spacial locality and decreasing latency. The hierarchical structure of multi-level blocking ensures load balance among cache levels, eliminating unnecessary bottlenecks. Although true parallelism

is split across multiple cores, the SIMD vectorization and efficient cache usage increase throughput and performance.

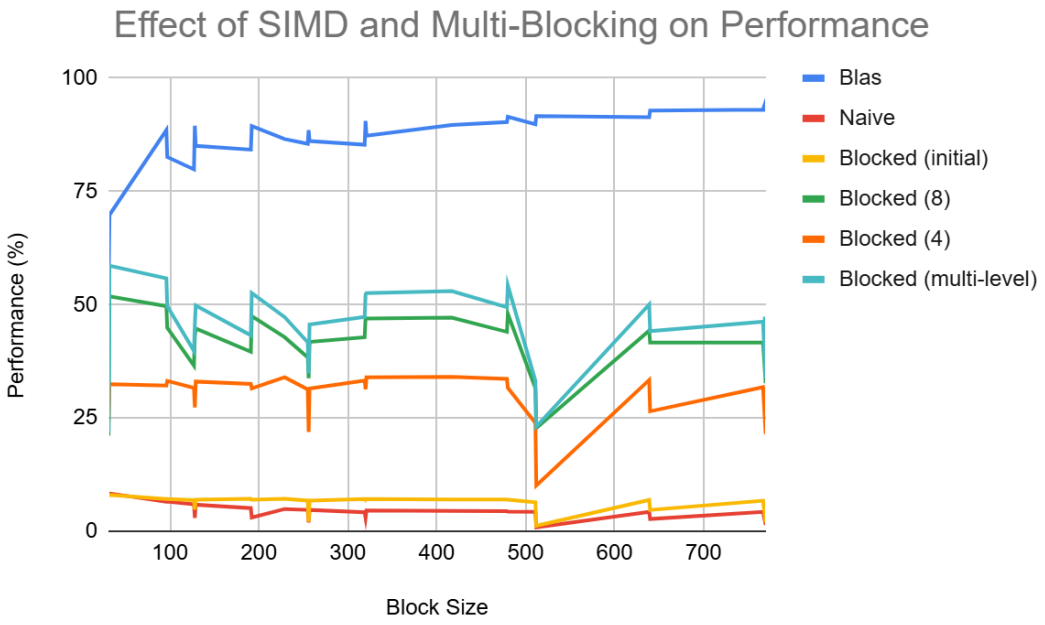

Figure 5. Effect of SIMD and Multi-blocking on Performance

Conclusion

Herein we demonstrate various optimization strategies for blocked matrix multiplication utilizing various block sizes, multi-level blocking, and single instruction, multiple data. The primary focus of our optimizations is spatial locality and efficient cache use. Loop reordering provided about 15% increase in performance, however, SIMD proved to be more useful. Our optimal code reached an average of 45% performance with multi-level blocking and SIMD optimization. SIMD instructions allowed us to introduce vectorized operations and increase throughput while tuning step size and multi-level blocking allowed efficient cache use. We anticipate further optimization such as repacking can increase performance by optimizing memory alignment and access.