CS267 Assignment 1: Optimizing Matrix Multiplication

Cheol Jun Cho, Grace Wei, Yinjun Zheng

Spring 2023

1. Introduction

In this work, we implemented an optimized matrix multiplication function on NERSC's Perlmutter supercomputer. The theoretical peak of Perlmutter's Milan nodes is 56 GFlops/s. Our goal was to write optimized single-threaded matrix multiply kernels, and reach a value as close as possible to this theoretical peak. Using four different methods - loop reordering, multi-level blocking, microkernel SIMD, and repacking - we were able to optimize the function to obtain 48% of the theoretical peak. In this report, we discuss each optimization separately alongside incremental performance increases, and then present the final result obtained via additional optimization of parameters.

2. Methods for Optimization

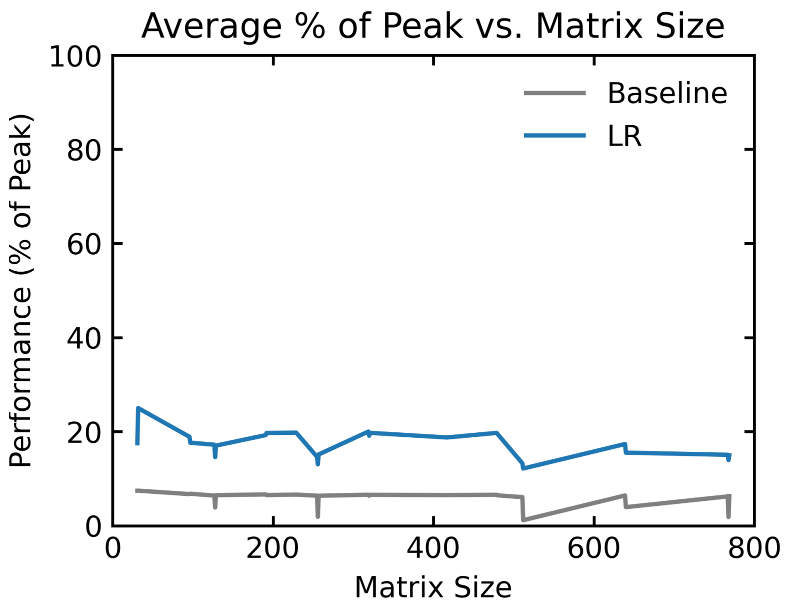
In this section, we will show the four methods we have tested, and present the results obtained for each method.

2.1 Loop Reordering (LR)

In the original code, successive matrix computations C[i+j*n] += A[i+k*n] * B[k+j*n] are performed by traversing the outer loop *i*, followed by the middle loop *j* and then the inner loop *k*.

However, this ordering is vastly inefficient. First, we observe that the destination register C[i+j*n] is independent of *k*, meaning that instructions must be executed serially. Thus, we can take *k* to be the outer loop, meaning the inner *i,j* loops can be executed in parallel, leading to code speedup. Additionally, to optimize the middle and inner loop order, we wish for the inner loop to access data that is contiguous in memory; this would allow us to maximize cache hits and minimize cache misses. This is because the processor will access a portion of the array at a time, so accessing contiguous data preserves spatial locality. The matrices are stored in column-major order, which means *i* should be the innermost loop.

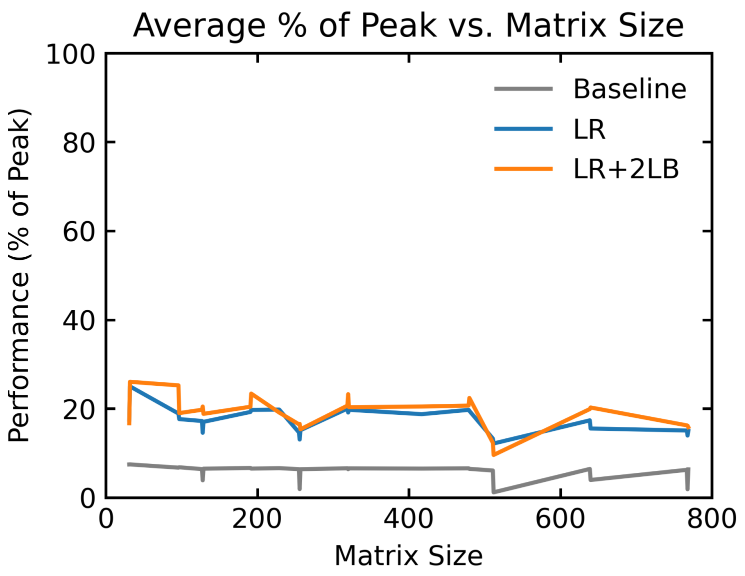
Thus, our first optimization was to re-order the loop with *k* as the outer loop, *j* as the middle loop, and *i* as the inner loop. This allowed us to reach an average performance of 17.2% of the peak performance, a large improvement from the unoptimized block matrix code given, which only reached 5.7% of the peak. Below, we plot the two performance percentages as a function of matrix size.



2.2 Two-level Blocking (2LB)

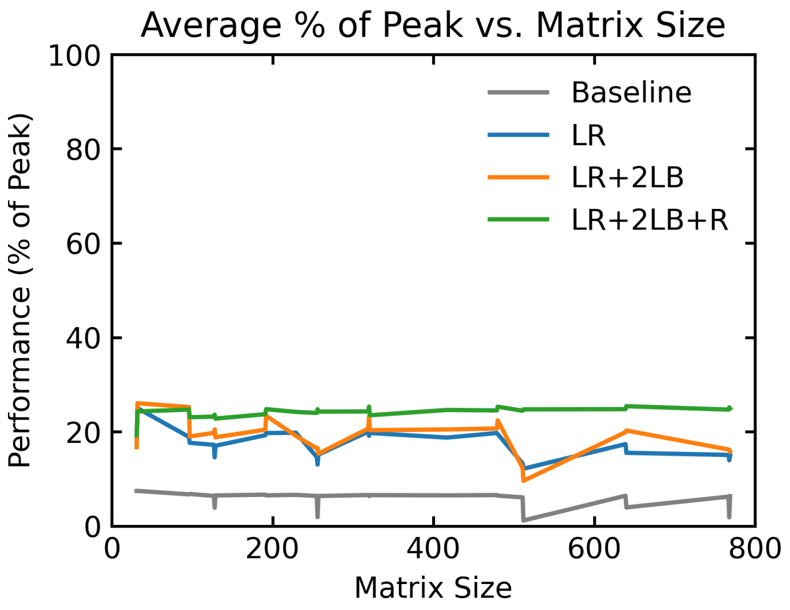
The second optimization we perform is the addition of a second layer of blocking to the function. We divide the original matrix into smaller matrix blocks, and then divide these blocks into further blocks where we then do smaller matrix operations by iterating through these blocks. Our goal is to target L2 for the first level, and target L1 for the second level. In this way, we can fully utilize the L1 and L2 cache to make retrieving data faster.

We test different block sizes for the level one and level two blocking and choose the best result with outer block size = 64 and inner lock size = 32. The resulting optimized function with multi-level blocking and loop reordering achieves average performance 19% of peak performance. This is plotted alongside the loop reordering optimization below.



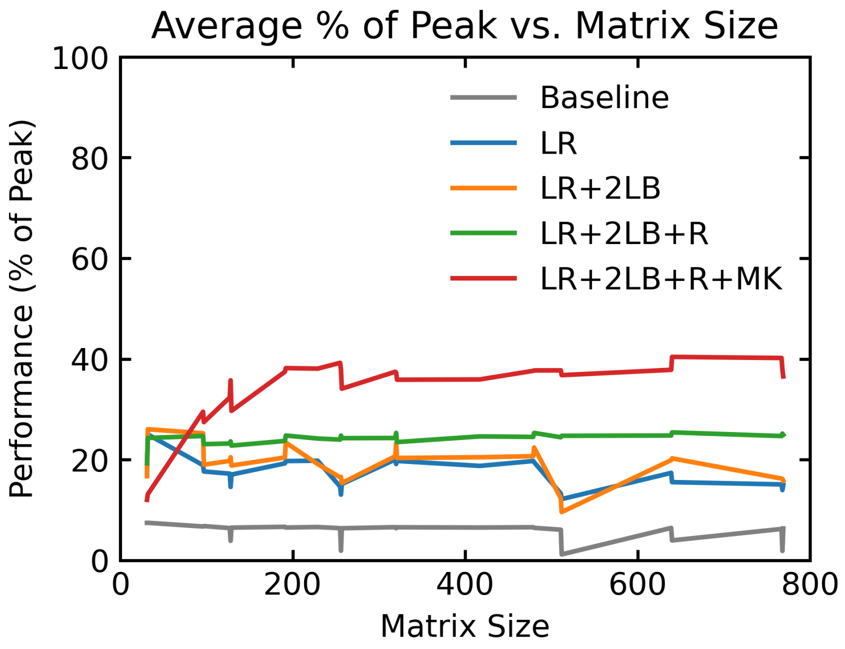
2.3 Repacking (R)

When we block the matrix into smaller blocks, the memory access is not contiguous, which leads to cache misses. Thus, our third optimization is to repack the blocked portions of the matrix into new memory locations such that the computation then accesses the values of A and B contiguously. Additionally, we store the results in C in contiguous memory locations and copy them into column-major order after the computation. After repacking the blocks, our code achieved an average performance of 25% of peak. Notice that many of the dips in performance were resolved due to the decrease in cache misses and cache conflicts due to the repacking.



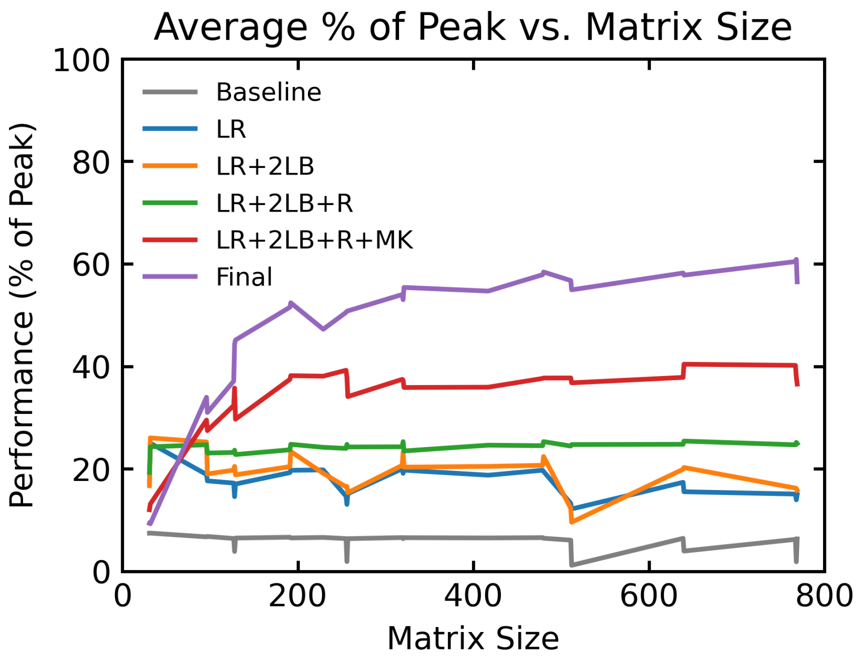
2.4 Micro-Kernel (MK)

Our fourth optimization is the addition of a micro-kernel with SIMD instructions. Currently, each multiply-and-add operation performed during matrix multiplication has a loop counter increment, which is a costly check that lowers performance. Thus, a micro-kernel can be used to unroll the loop. With our micro-kernel implementation, the loops increment by the microkernel size, which we set here as 4, meaning the processor only has to check whether the loop bound has been exceeded N/4 times for each loop, greatly improving performance. Furthermore, performing several operations next to one another in the kernel allows for increased instruction-level parallelism. Additionally, we utilize 256-bit AVX intrinsics to make use of hardware vector registers. Because vector intrinsics are implemented directly in compilers, the CPU is able to perform them much faster than library functions. Our micro-kernel makes use of vector intrinsics to perform a 4xMK_SIZE matrix multiplication. Below, we observe that the use of a 4x4 microkernel increases the average code performance across all matrix sizes to 34%. To implement the microkernel, we also had to rewrite the repacking method to be consistent with the kernel.



2.5 Final Tuning and Result

With all four of our optimizations, we perform final tuning of the parameters and implement small changes to the code to improve performance. We align all the matrix blocks to the cache boundary such that the SIMD instructions can work on contiguous segments of 4 double words in memory. We also increase our microkernel size so that the microkernel performs a 4x12 operation. We increase the block size of L1 to 192 and the block size of L2 to 768. With these additional steps, our code reaches an average performance of 48% of theoretical peak.



3. Odd Behaviors

In this section, we will discuss two anomalies in performance, namely 1) dips in performance and 2) low performance at small matrix sizes.

In many of the curves, we observe noticeable dips in performance when the matrix size is at particular powers of two (2^7=128, 2^8=256, and 2^8=512). This is because of cache associativity. The CPU we are working with has 8-way set associative L1 and L2 cache organization, and 16-way set associative L3 cache organization, which means each data block is mapped to 8 or 16 cache lines. The mapping of these cache lines is straightforward and determined by the memory address. In fact, the cache line index is simply the “middle portion” of the memory address. This means that memory addresses with the same middle part will map to the same cache lines. Thus, at higher powers of two, we are utilizing addresses that likely will map to the same cache line, which reduces the effective size of the cache. This helps to explain the large performance dips for many of the optimizations. Additionally, we notice that these dips are less prominent after repacking is implemented. This is likely because repacking alters the memory locations of the matrices, thus mitigating cache conflict.

Furthermore, we observe that upon implementation of the micro-kernel, our matrix performance for smaller matrices with sizes less than 100 x 100 drops significantly. This is because for smaller matrices, we are not reusing the vector registers as much as for larger matrices, which makes it inefficient to load them explicitly via our code. Additionally, because our block sizes are larger to accommodate the increased microkernel size, we essentially perform naive matrix multiplication for smaller matrices. These two reasons are explanations for the low performance of smaller matrices upon addition of the microkernel.