CS 267: HW 1

Yisen Wang, Amit Jain

February 6, 2022

1 Introduction

The objective of this assignment is to optimize matrix multiplication on NERSC's Cori supercomputer. Our job is to write an optimized single-threaded matrix multiply kernel and tune the code to run efficiently on Cori's processors. The focus will mainly be on reducing data movement, better utilizing the memory hierarchy, rearranging the computing logic such as the loop order and implementing SIMD Micro-Kernel to increase single core parallelism.

The matrix multiplication algorithm will be benchmarked on the Intel Xeon Phi KNL processor with the following characteristics:

1. CPU operating frequency at 1.4GHz.
2. Two vector lanes
3. Registers: 32 AVX512 vector registers holding 8 doubles each
4. L1 Data Cache: 32 KB direct-mapped
5. L2 Cache: 1 MB direct-mapped

The following section will give a brief introduction to the given naive implementation using three for loops and one layer tiling with default block size - 41. Our successive performance-improvement techniques are mainly built on the blocked method. Section 3 gives the detailed performance improvement process and analysis. Finally, we will visualize the performance change compared to the benchmark at different phases and present the final conclusion.

2 Naive Implementation

In this section, we run the given reference code and visualize the performance comparison (Figure 1). For the naive three non-blocked loop implementation, we read each column of B n times: n^3, read each row of A once: n^2 and r/w each element of C once: $2n^2$.At this point, we don't take advantage of memory hierarchy - L2, L1 and Register. There will be high data transfer latency for each multiply and add instruction. In the code provided to use, one layer of blocking was also implemented on top of the naive solution. However, this method still has several deficiencies such as high loop control overhead . Problems of heavy data transferring and inefficient cache memory access also exist. In addition, through our next experiment, solely adding the layer of blocking cannot bring that much performance improvement. Only if we comprehensively consider the feature of memory distribution and fully utilize the SIMD feature by using vector computation, can we witness significant improvement.

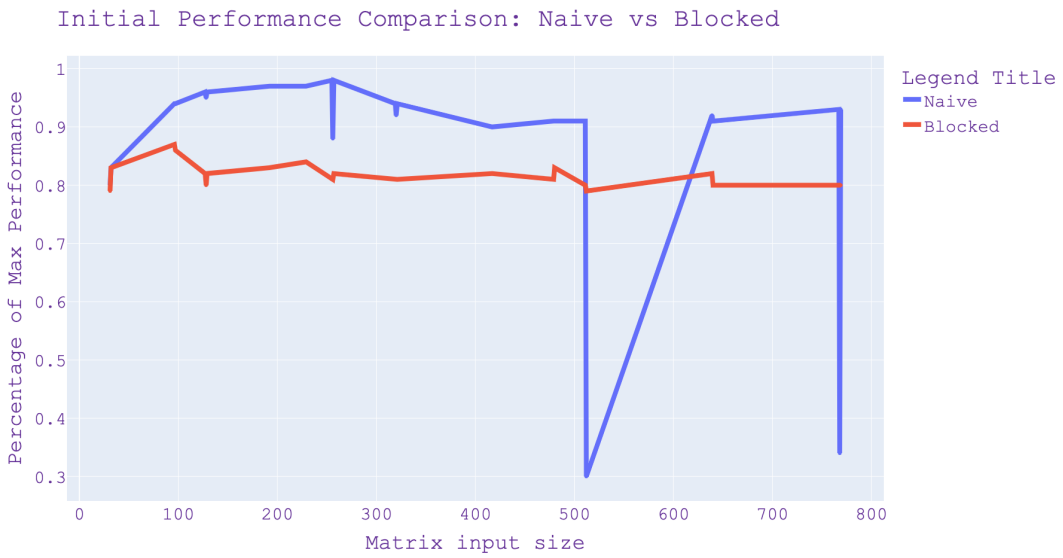

Figure 1. Initial Performance Comparison: Naive vs Blocked

3 Optimization and Analysis

3.1 Overview of optimizations used

1. Blocked Matrix Multiply: 3 Levels of blocking for L2, L1 and Registers.
2. Optimizing Loop Order and Block size at each level: Final block sizes were 1024, 256 and 96. The loop order was kept as j, k, i from outer loop to inner loop across all levels of blocking
3. Tiling for Registers: We used a 8x4 Tile for C computed by multiplying a 8x4 A-Tile by 4x8 B-Tile. The tiling above gives us the following register usage: 4x AVX512 Registers for C, 4x AVX512 Registers for A, 16 AVX512 Registers for B (set1_pd was used)
4. Loop Unrolling: The above register tiling allows us to loop unroll minimizing the branching overhead
5. SIMD Microkernel with AVX Intrinsics for data level parallelism. Since the Tile is small enough we can store and employ vectorization with AVX512 registers, acting on 2x8 doubles in parallel.
6. Instruction Level Parallelism: Microkernel was rewritten twice to provide the optimal instruction level parallelism. Greater ILP was achieved by removing dependencies between adjacent instructions

3.2 Blocked Matrix Multiply

By using a technique known as matrix blocking, we can leverage the computers memory hierarchy for efficiently storing data into different levels of cache and exploit the benefits of spatial locality. In cache blocking, we are essentially partitioning the input matrix into smaller matrices correlated to the sizes of the different memory sectors in our system. With appropriate partitioning of the input matrix, we can target specific caches such as L1 and L2 as well as the CPU registers. The following image shows the CPU memory hierarchy for the Cori Supercomputer's Intel Xeon Phi KNL processor:

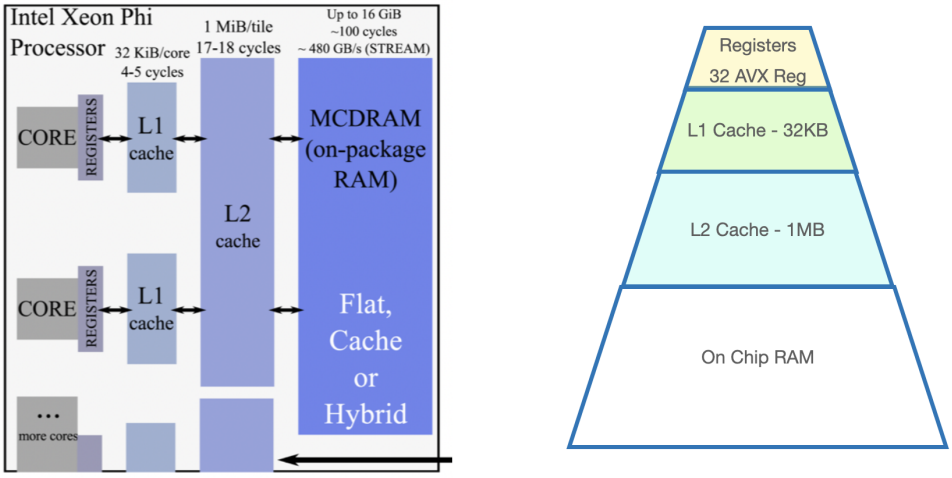

Figure 2. Memory hierarchy for Intel Xenon Phi KNL (Knights Landing) CPU

We ran an experiment to see how many levels of blocking would provide the optimal performance. Results shown below in Figure 2 were collected with the optimized loop order as well as the optimized block size.

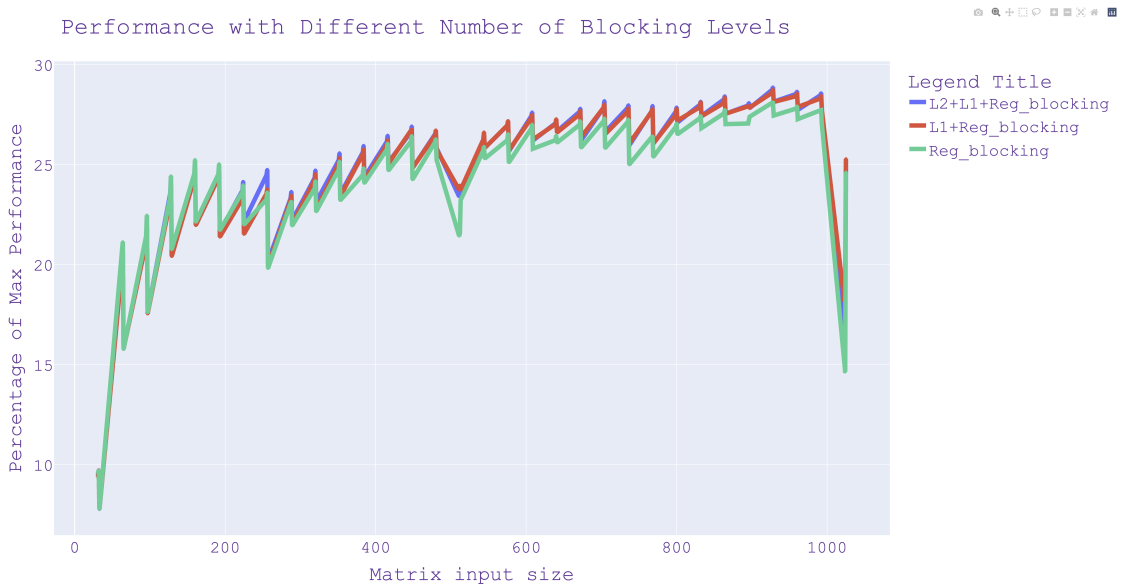

Figure 3. Comparison of Different Number of Blocking Levels

As can be noticed above, different levels of blocking didn't provide a significant performance enhance-

ment, but enabling at least one additional level of blocking on top of the register blocking did yield a 1% improvement. The minor performance improvement comes from reaping the benefits of data locality in multiple memory sectors as opposed to just one. We can also see the dips are less pronounced, although still present, with an additional level of blocking. The dips are an artifact of cache conflicts, and if we split up subsequent matrix entries that map to the same cache line into different blocks, we can effectively avoid those cache conflicts. Later on we will discuss a better method to avoid these large performance dips due to cache conflicts. We chose 3 levels of blocking here as the final implementation as it provided the best overall performance.

3.3 Block size and Loop Order

In our first attempt to find the optimal block size, we first figure out the ideal block size for each level of cache, based on the formula given

$$\text{Blocking Size} = \sqrt{\frac{\text{Cache size}}{3 * 8Byte}}$$

The numerator here will be the size of the cache we are targeting in a blocking level. The three in the denominator comes from the requirement to fit three matrices in cache, one for A, B and C. A double precision number is stored in each entry of the matrix, hence we also divide by 8. Finally we take the square root to get the length of the edge for each matrix A, B and C.
Therefore, we get the following ideal block sizes:

$$L2_{size} = \sqrt{\frac{\text{1MB}}{3 * 8Byte}} = 204,\ L1_{size} = \sqrt{\frac{\text{32KB}}{3 * 8Byte}} = 37,\ Reg_{size} = \sqrt{\frac{\text{32*8*8B}}{3 * 8Byte}} = 9$$

We then scaled these block sizes down by a factor of 1/4 and used that as a seed for determining the optimal block sizes. Ultimately we found the following set of blocking sizes gave us the highest performance: outer-block = 1024; middle-block = 256; inner-block = 96. These ended up being much higher than what our original calculations produced. We think that we were actually targeting one memory level above what we had anticipated. So our inner-block wasn't targeting register level but rather L1 cache. Similarly, the outer block wasn't targeting L2 but perhaps MCDRAM. The following graph shows a comparison of different block sizes:

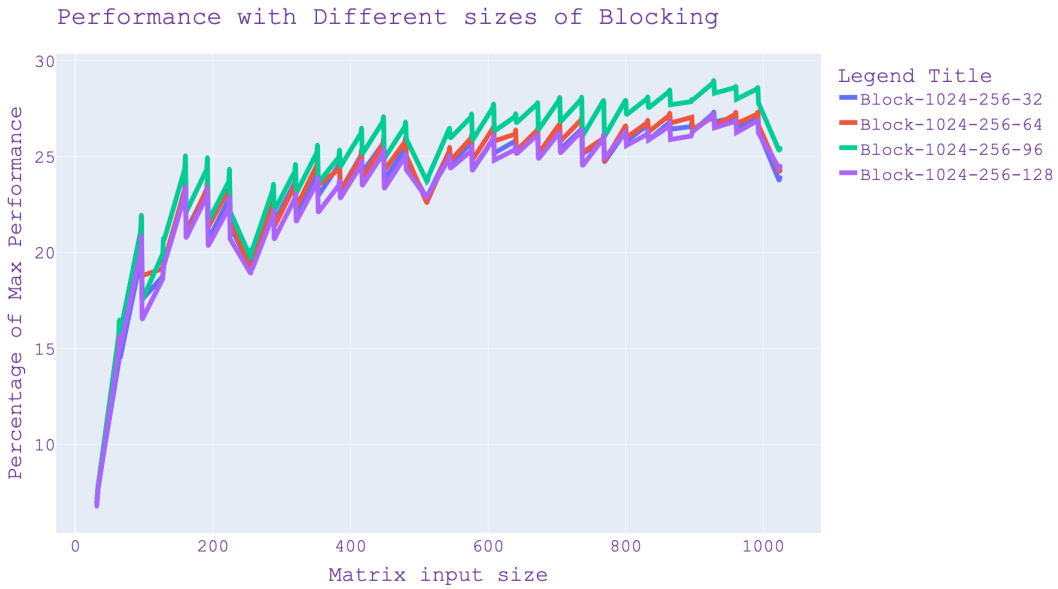

Figure 4. Performance With Different Size of Blocking

For determining the optimal loop order, let's first take a look at the Naive implementation . The three simple for-loop implementations is as follows:

```
void square_dgemm (int n, double* A, double* B, double* C) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      for(int k = 0; k < n; k++)
        C[i+j*n] += A[i+k*n] * B[k+j*n];
}}}
```

If we rearrange the loop order for Matrix Multiplication, we will get 6 potential orders. The one that provided us the optimal performance was JKI loop order.

```
void square_dgemm (int n, double* A, double* B, double* C) {
  for (int j = 0; j < n; ++i) {
    for (int k = 0; k < n; ++j) {
      for(int i = 0; i < n; k++)
        C[i+j*n] += A[i+k*n] * B[k+j*n];
}}}
```

We believe JKI loop order is the optimal loop order in our implementation because of how the matrix elements are packed in our algorithm. We decided to keep all matrices, at all blocking levels, in column major order due to how our micro kernel was written and also to avoid the overhead of transposing the matrices. The following is an explanation for why JKI loop order is the optimal computing logic for column major distribution of memory. First, let's focus on the inner for-loop. Here, we have i traversing for 0 to n-1, which is in a line with the column major distribution. Therefore, the address pointer could move fast for Matrix A, also for Matrix C. This is because, for C, we are actually updating the same column of C in our inner two layers of for-loop, which is k from 0 to n-1 and i from 0 to n-1. As we can see, each column of C will be reused many times. So, we end up caching this column and reduce the unnecessary data fetching. The process is also illustrated in the following image:

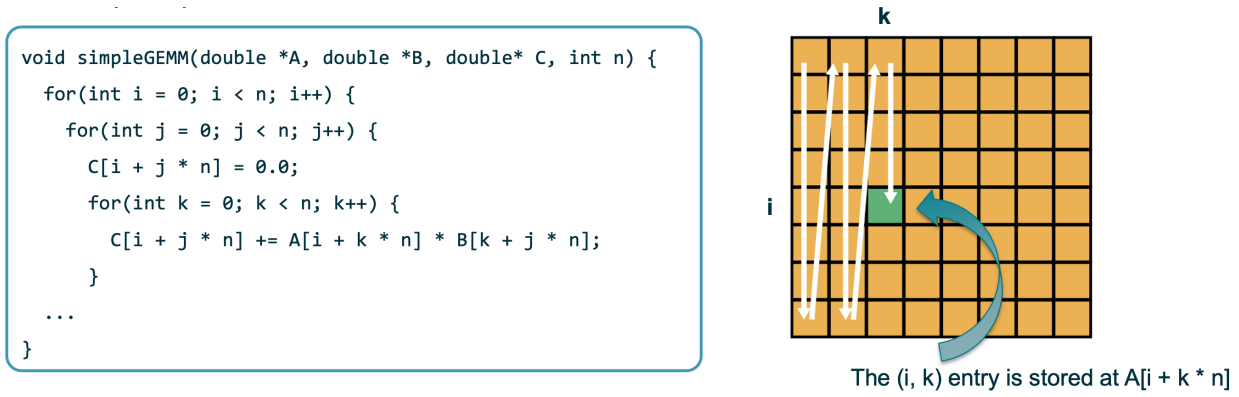

Figure 5. Memory Accessing Process

3.4 Tiling for Register and Loop Unrolling

Even with accurate branch prediction there can be quite a bit of overhead with control statements. By enforcing loop unrolling, we can minimize the number of branch statements. The unrolling takes place in the inner loop and the amount of unrolling is governed by the tiling size selected.

In the investigation for the appropriate register tile size, we looked at three different combinations of matrix sizes for C, B and A. Initially we studied $C_{8x8} = [A_{8x8}][B_{8x8}]$. This required a significant amount of loop-unrolling, but once implemented we immediately saw the benefits over the naive implementation with no-unrolling. We then switched the unrolled instructions to AVX512 instructions to leverage the CPU

registers. However with this tile size we were using 70 registers, which means we'd be replacing some register values that we intended to keep for reuse. We explored a smaller tile size $C_{8x8} = [A_{8x4}][B_{4x8}]$ which required only 44 registers, but this did not seem to improve performance much, and was still exceeding the 32 registers available. As a final step, we went down to using a tile size of $C_{8x4} = [A_{8x4}][B_{4x4}]$ which required only 24 AVX registers, and here we saw a performance boost of 1%. We decided to use this tile size in our final implementation. Another benefit of loop unrolling is not only minimizing the control overhead but also exposing opportunities for Instruction-Level Parallelism, which will be discussed in the next section.

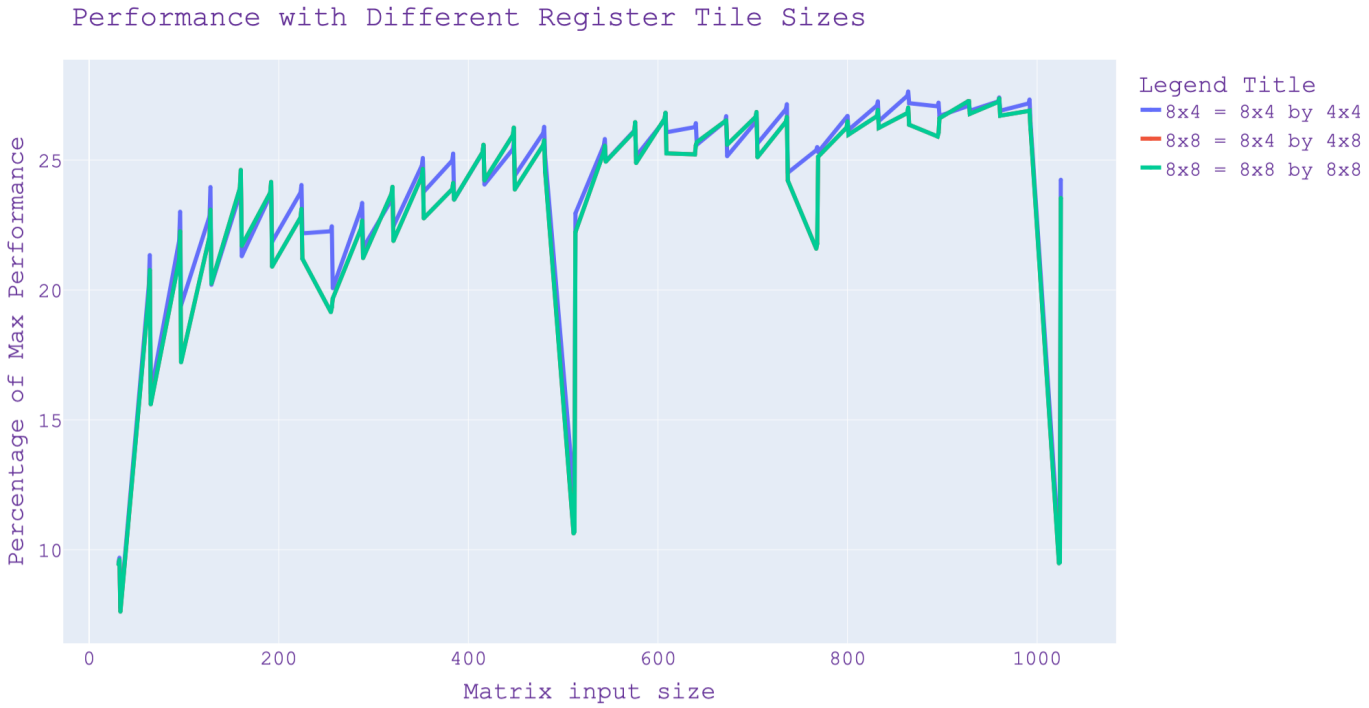

Figure 6. Comparing Different Tiling Sizes

3.5 SIMD Microkernel with AVX512 Intrinsics

At this stage, we considered using a micro-kernel to manually calculate four columns Matrix C with a sequence of explicit instructions. The benefits of using the Micro-kernel with explicit instructions is it will allow us to enforce CPU register usage and also give us more fine-grained control on instruction level parallelism. First, we extracted the 4x4 small block from B and broadcast each entry to a 8*Double long vector in 16 AVX registers. This is done by using the set1_pd intrinsic. Then, based on column-major memory structure, we extracted an 8x4 small block of A, using 4 AVX registers for each column. We also saved the corresponding 4 columns of Matrix C into AVX registers. Finally, we need to execute the FMA instruction to update Matrix C. The computation is illustrated below in Figure 7. The columns and rows are shaded with the same color to indicate which elements are being multiplied together. The next computation that would be applied from the one shown in Figure 7, in our sequence of instructions, would be with the next column of C (C01), next column of A (A01), and the element below in B (10).

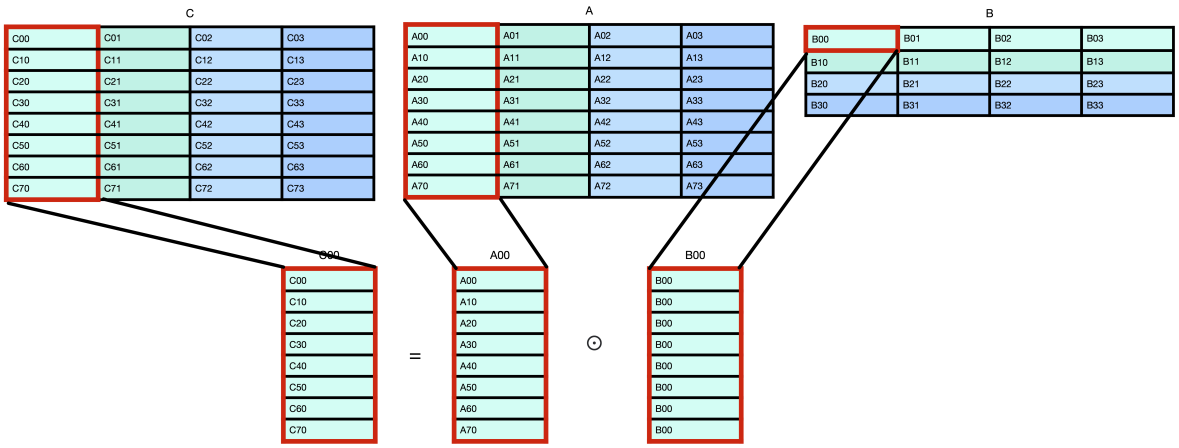

Figure 7. AVX Register Vector Multiplcation Scheme

When we originally wrote the micro-kernel we were calculating each column of C before moving onto the next column. This meant every column in A was being multiplied by the corresponding element in B00's column to produce the C00 column. We would repeat this for every column of C. The process would look as follows:

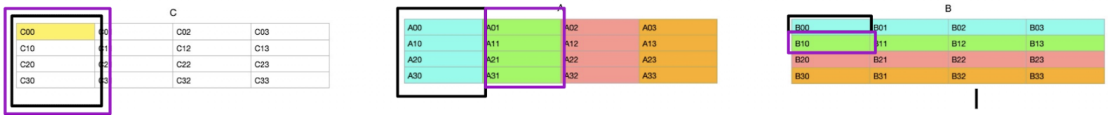

Figure 8. Old microkernel technique

The downside of this sequence is that we would be waiting for the previous instruction to calculate the value of C00 before executing the current one. In other words, we were introducing dependencies between subsequent instructions and not fully utilizing the CPU's pipeline. Once we re-organized the micro kernel to improve the instruction level parallelism we saw close to a 3% performance improvement. The graph shows the comparison between the two microkernels.

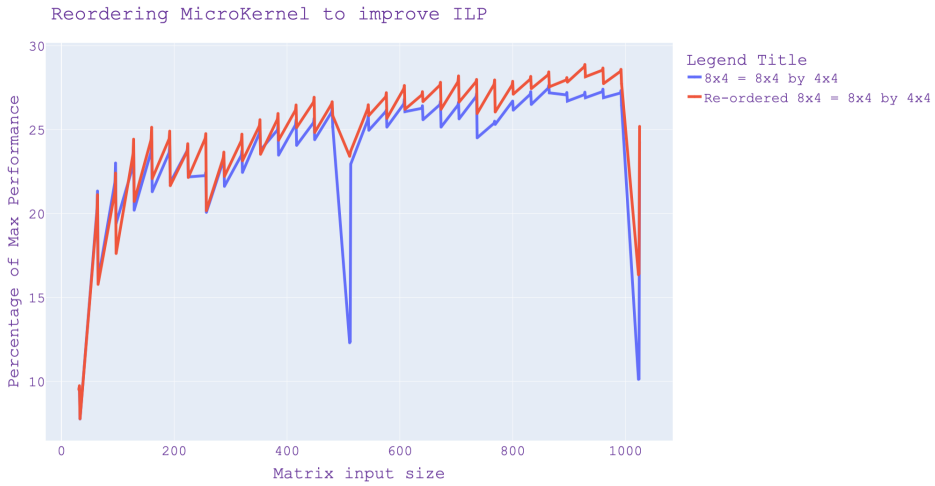

Figure 9. Improving performance with Instruction Level Parallelism

3.6 Other optimizations attempted

One of the other optimizations we tried was software prefetching, but we weren't able to find an appropriate prefetch distance that showed any noticeable performance improvement. The idea with software prefetching is that you can explicitly tell which segment of memory (row or column) to pull into L1 cache so that its ready to load in the registers once the computation on it begins. In most attempts, software prefetching would actually lead to a significant degradation, likely because we were clogging up the pipeline and hardware prefetching was already doing a sufficient job.

4 Performance Analysis

4.1 Performance Overview

The average performance that our algorithm yielded was 24.8 %. This is within the target range of 10-30 % and significantly higher than the original performance of 0.8 % with the provided implementation. The graph below illustrates a comparison of our algorithm (in blue), with the naive implementation (in red) and the BLAS implementation (in green).

Further examining the graphs, we can notice that the smaller matrices don't reach as high of performance as the larger matrices. This is simply because the smaller matrices don't get to exploit the benefits of temporal and spatial locality as much as the larger matrices do. The computational intensity of smaller matrices is lower than that of larger matrices because the number of accesses to slow memory relative to the total number of floating point operations is higher. Once we reach a matrix size of 200x200, we can see that the performance stays fairly constant with a variation of +-0.5%. The performance dips seen with our blocked implementation were less severe than those experienced with the BLAS implementation. In the following section we will take a closer look at why these performance dips occur and how we can potentially mitigate them.

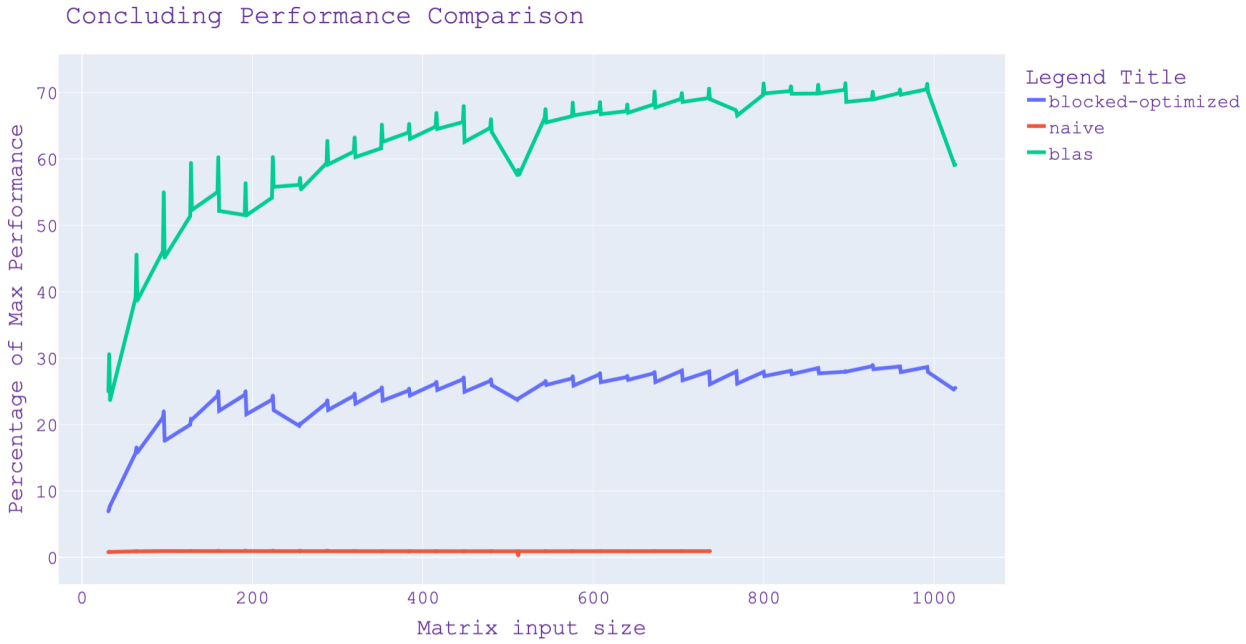

Figure 10. Final Performance Results and Comparison with other Algorithms

4.2 Performance Anomalies

There were a couple odd behaviours we observed during our performance testing, but perhaps the most obvious being the large dips in performance particularly at matrix sizes of the power of 2. At 1024x1024 we

saw close to 8% performance degradation. After examining why this could happen, we came to understand that its likely because of cache conflicts and TLB misses. At powers of 2, we will start getting data wrapping around and getting stored in the same 64-byte cache line causing nearly immediate eviction of other relevant data. A hack that we had devised in order to mitigate this was to implement additional padding for the matrix sizes that were a power of 2. This way we can ensure that same cache lines aren't being used. The red trace in the following graph shows the implementation of the additional padding technique.

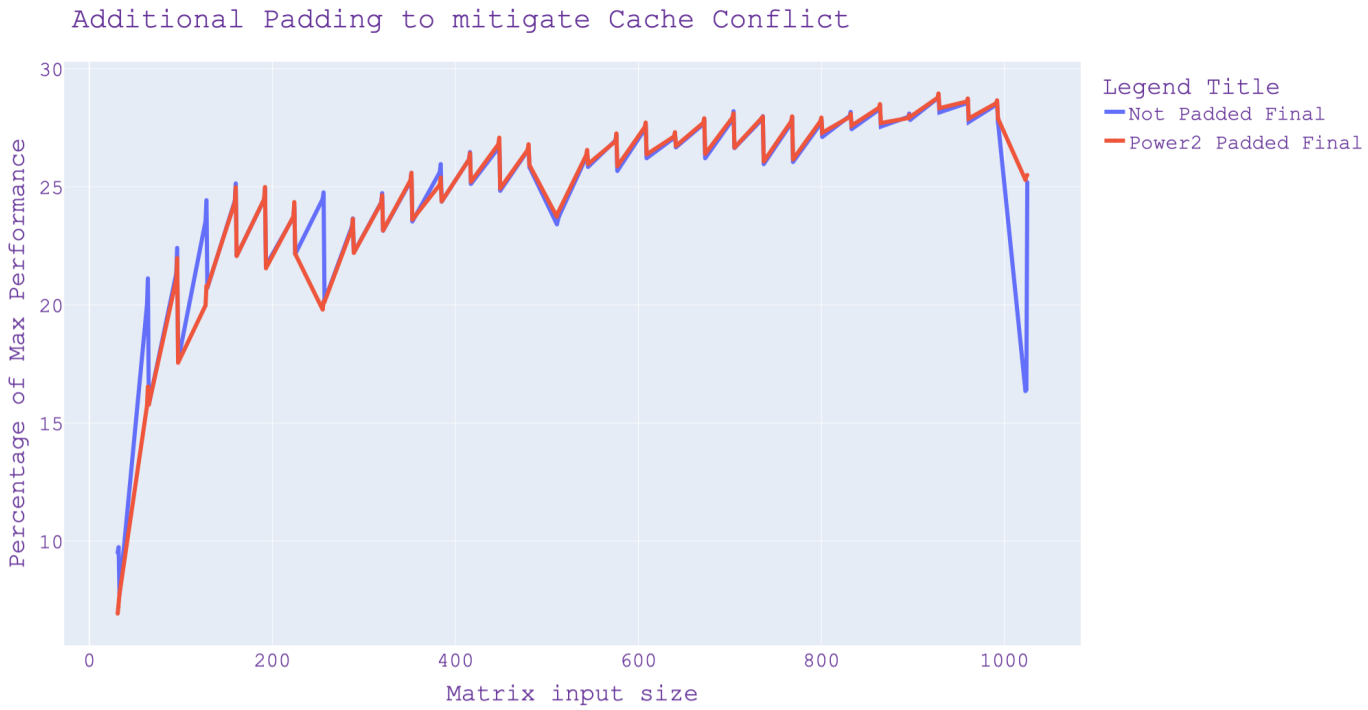

Figure 11. Comparing Padded vs non-padded

5 Division of Work

Amit:

Did initial set of optimizations, including blocking, finding optimal block sizes, determining correct loop order and 1st version of SIMD Microkernel. This got us from 0.8% to 12%

Wrote the final version of SIMD microkernel with new register tiling and enhanced ILP, taking performance from 20% to 25%.

Owned majority of the report. Contributed significantly to all sections of the report, provided all the figures and graphs apart from Figure 4/5

Yisen:

Design and implemented 2nd version of SIMD Microkernel, using AVX512 : C(8x8) = A(8x8) X B(8x8), laying the foundation for our successive optimization. This increased the performance from 12% to 20%.

Contributed to the writeup for Section 1, Section 2, Section3.3 (lock size and Loop Order) and give the mathematical explanation for SIMD AVX512 Operation in Section3.5. Collected graphs showing difference in block sizes for L1 cache.