# CS267 Homework 1: Optimizing Matrix Multiplication

Tzu-Chuan Lin, Yu Gai, Ian Kolaja

February 3rd, 2022

## 1 Introduction

This report describes the efforts to develop an optimized matrix multiplication designed to run on the NERSC Cori supercomputer. A single-threaded matrix multiply kernel was developed that runs on a single core. Several optimization techniques were applied, and the most effective combinations of these techniques were explored.

## 2 Methods of Optimization

This section describes the individual optimization techniques that were implemented.

### 2.1 SIMD

Based on the starter code logic, we adapt the AVX512 registers to apply SIMD into the logic.
Note that because we want to use cache aligned intrinsics (ex. _mm512_load_pd(...), not _mm512_load**u**_pd) to achieve the highest speed. We force our BLOCK_SIZE macro to be a multiple of 8. It is also worth noting that the original block-level "ijk" order will write to the same memory location, so that is why we first tried "jki" order (we will explain loop order in 2.3 in more detail) to issue a single instruction and write into multiple locations at the same time.

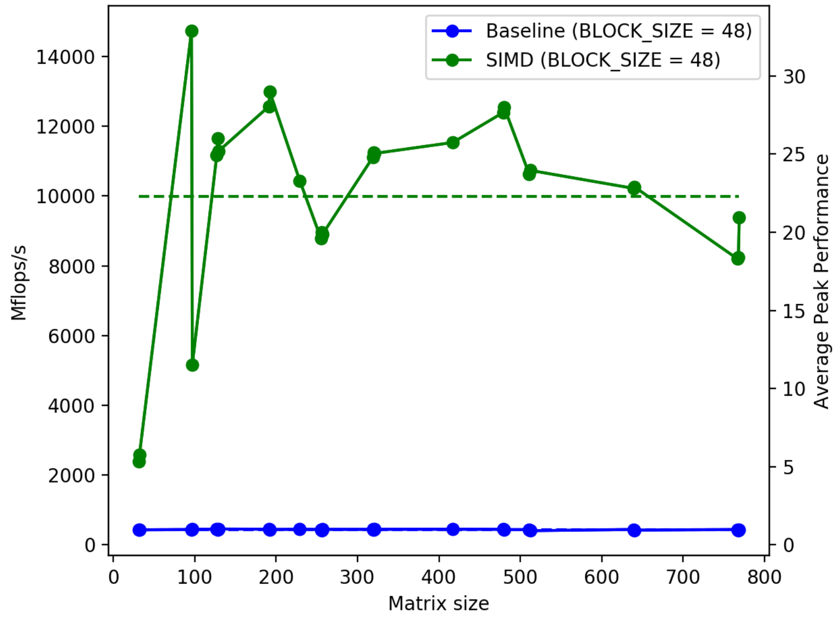

Figure 1. Performance of baseline implementation versus SIMD enhanced implementation with "jki" loop ordering

In Figure 1, you can see that after reordering the loop to "jki" and applying SIMD, the speed is boosted from 1% of the peak to ~23% of the peak when we fixed the BLOCK_SIZE to 48. (Checkout the code experiment 0 and 3 for more details).

## 2.2 Blocking

The blocking algorithm in the starter code provided in dgemm-blocked.c was used as a starting point for development. Smaller matrix multiplication operations are performed by iterating through blocks of the original matrices. This allows for data to be stored in the L1 or L2 caches and be utilized more quickly. It was necessary to determine the optimal block size to fit the necessary data into the L1 caches. This promotes greater spatial locality.

Based on section 2.1 outstanding result, we experimented different BLOCK_SIZE to see which BLOCK_SIZE will give us the best performance. Figure 2. shows the impact of block size on one implementation of the code. It is seen that a block size of 48 was optimal. Therefore, in the later sections, we will stick with BLOCK_SIZE = 48 throughout our experiments.

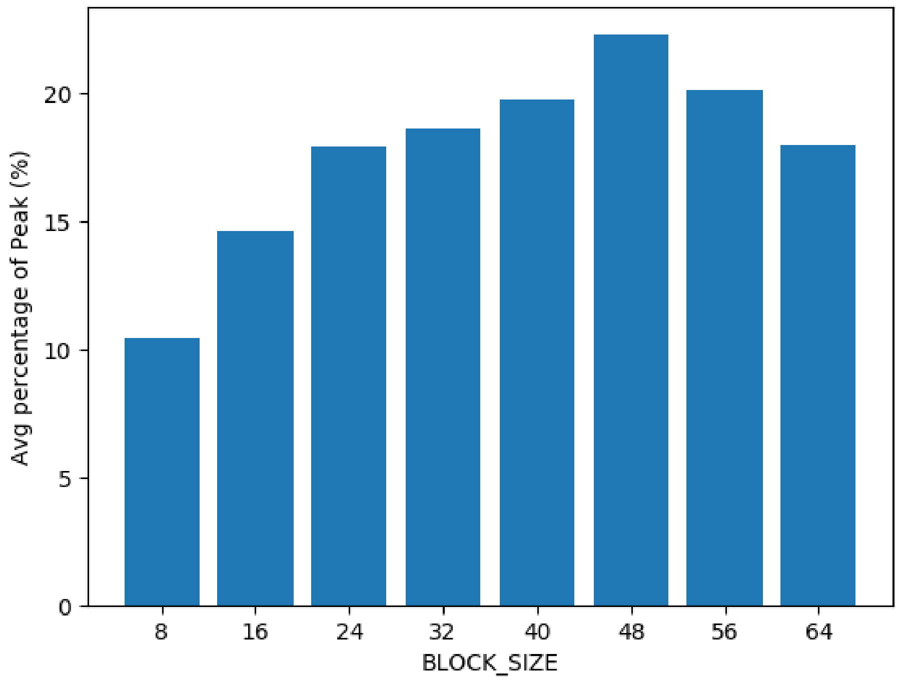

Figure 2. Impact of block size on performance.

Different block shapes were not explored. Recognizing that the input arrays are arranged in a column-major format, utilizing a rectangular block shape that has longer columns than rows would likely be marginally beneficial.

## 2.3 Loop Order Optimization

The original starter code of the blocking algorithm can be described as this pseudocode:

```
// global level
for(int i = 0; i < N; i += BLOCK_SIZE){
    for(int j = 0; j < N; j += BLOCK_SIZE){
        for(int k = 0; k < N; k += BLOCK_SIZE){
            // block level
            for(int ii = 0; ii < BLOCK_SIZE; ++ii){
                for(int jj = 0; jj < BLOCK_SIZE; ++jj){
                    for(int kk = 0; kk < BLOCK_SIZE; ++kk){
                        ...
```

We refer the outer three for loops as “**global-level**” and the inner three loops as “**block-level**”. Initially, the global-level loop order is “ijk” and the block-level loop order is “ijk” as well.
Note that because the block-level “ijk” or “jik” order will compute a (1 x BLOCK_SIZE) x (BLOCK_SIZE x 1) = (1 x 1) for each i, j, this **disable** SIMD the ability to simultaneously issue a single instruction to **write into multiple memory locations** which will be obviously slow compared to other loop order. Thus, we do **NOT** include “ijk” or “jik” block-level loop order into our experiment because it must be slow.

The pseudocode after SIMD is described as (only shows block-level “jki” order.):

```
// global level
for(int i = 0; i < N; i += BLOCK_SIZE){
    for(int j = 0; j < N; j += BLOCK_SIZE){
        for(int k = 0; k < N; k += BLOCK_SIZE){
            // block level
            for(int jj = 0; jj < BLOCK_SIZE; ++jj){
                for(int kk = 0; kk < BLOCK_SIZE; ++kk){
                    for(int ii = 0; ii < BLOCK_SIZE; ii += 8){
                        // perform SIMD
```

In Figure 3, we compare: **block-level** “jki” and “kji”. We do not include “ikj” or “kij” in the illustration because the last index “j” will **NOT** walk along the **column** (which is contiguous in the memory), thus generating more cache misses and **obviously slower.** We found out that “jki” is faster than “kji”, we argue that is because the cost of switching “j” is the most costly because when we jump from column j in B to column j+1, the program will likely need to fetch a new memory page and causing the CPU stall. That is why putting “j” to the most outer loop will boost the performance the most.

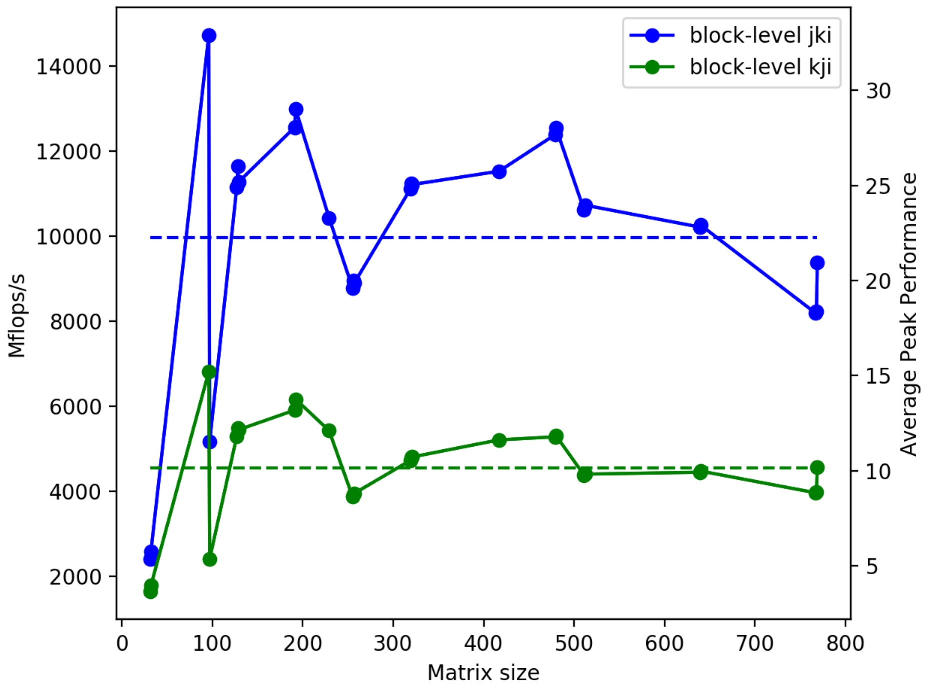

Figure 3. Comparison of block level loop order performance.

Similarly, we also tested different the global level loop order. Figure 4 shows that the “jki” loop order also performed better than “kji” at the global level.

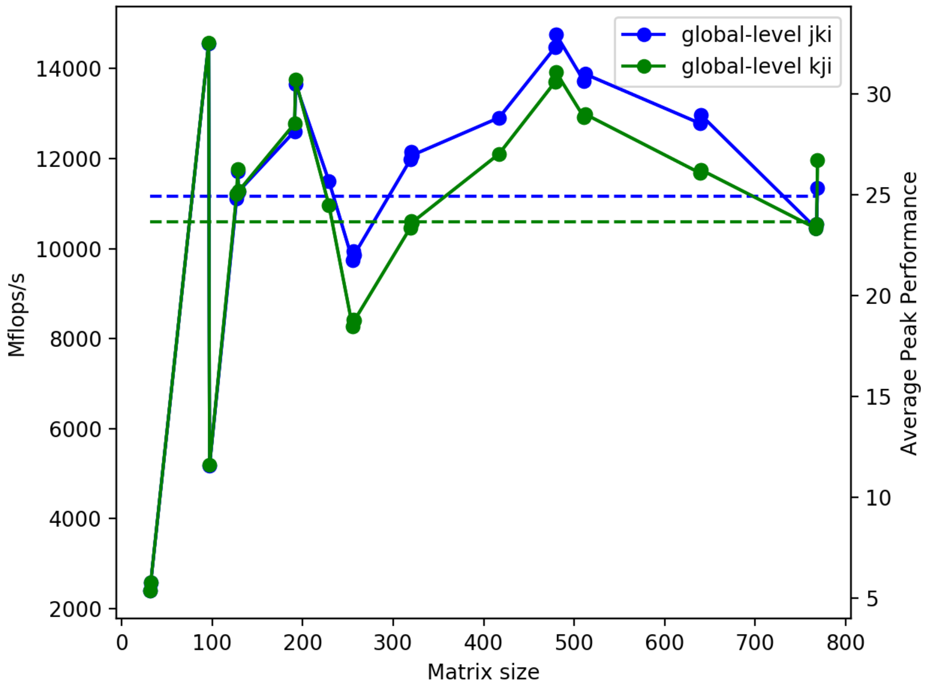

Figure 4. Comparison of globel level loop order performance.

Future sections utilize the “jki” loop order unless mentioned otherwise.

## 2.4 Matrix repacking

Matrix repacking was performed such that the “jki” loop order would access data in a contiguous manner, enabling the hardware to prefetch the next block for us. Therefore, we packed both the matrix A, C into global-level column major and block-level column major. We hope after the memory repacking, each block of the matrix during the computation will be fetched by hardware in advance and reduce the burden of virtual memory addressing. **However**, **no substantial improvement** was observed in our implementation from repacking. We are not sure why this does not enhance the performance. We will leave this exploration as future direction.
In Figure 5., we show the comparison between without packing and with packing performances.

```
#define BI(i) ((i) / BLOCK_SIZE)
#define BJ(j) ((j) / BLOCK_SIZE)
#define BII(i) ((i) % BLOCK_SIZE)
#define BJJ(j) ((j) % BLOCK_SIZE)
void cpy_and_pack(int N_pad, int N, double *from, double *to){
    int griddim = N_pad / BLOCK_SIZE;
    for(int j = 0; j < N; ++j){
        for(int i = 0; i < N; ++i){
            to[(BI(i) + BI(j) * griddim) * BLOCK_SIZE * BLOCK_SIZE + BII(i)
+ BJJ(j) * BLOCK_SIZE] = from[i + j * N];
        }
        for(int i = N; i < N_pad; ++i){
            to[(BI(i) + BI(j) * griddim) * BLOCK_SIZE * BLOCK_SIZE + BII(i)
+ BJJ(j) * BLOCK_SIZE] = 0;
        }
    }
    for(int j = N; j < N_pad; ++j){
        for(int i = 0; i < N_pad; ++i){
            to[(BI(i) + BI(j) * griddim) * BLOCK_SIZE * BLOCK_SIZE + BII(i)
+ BJJ(j) * BLOCK_SIZE] = 0;
        }
    }
}
```

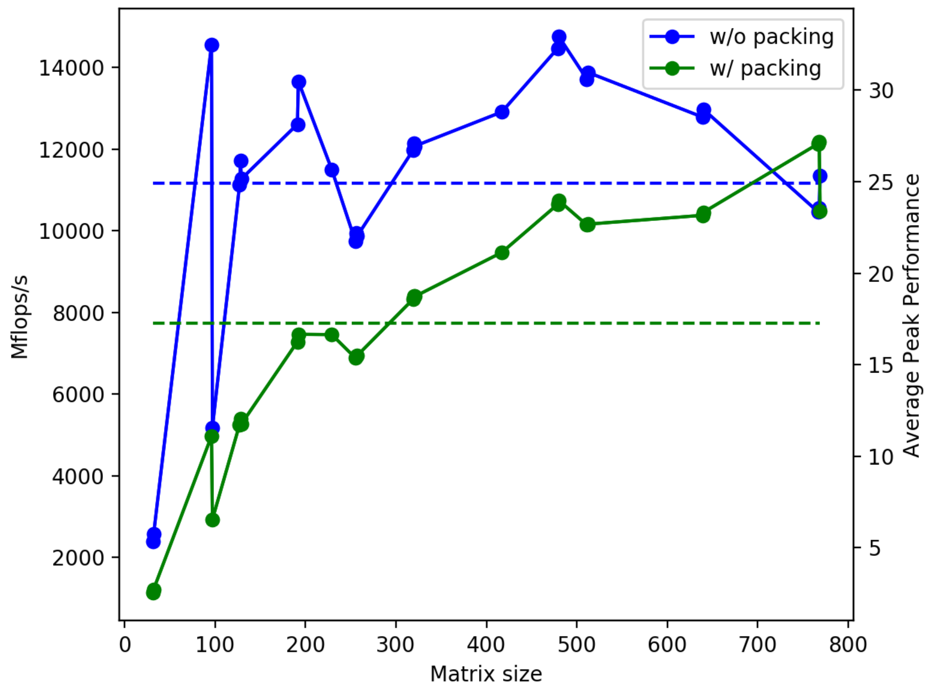

Figure 5. Comparison of performance with and without matrix repacking.

## 2.5 Two-level blocking

Due to the architecture of the KNL node, we implemented two-level blocking to aim to fit the larger block to L2 cache and the smaller block to L1 cache. A block size of 48 has already been shown to fit well in the L1 cache. The performance of different sizes for the larger block is shown in Figure 6. Two-level blocking lead to a slight decrease in performance and was not utilized in the final implementation.

## 2.6 Prefetching

We want to further test whether **explicitly** stating prefetching will affect the performance. It turns out that adding prefetching will slightly hurt the performance. We are not sure why and leave it to future direction. See “square_dgemm_jki_block_jki_prefetch” for more information.

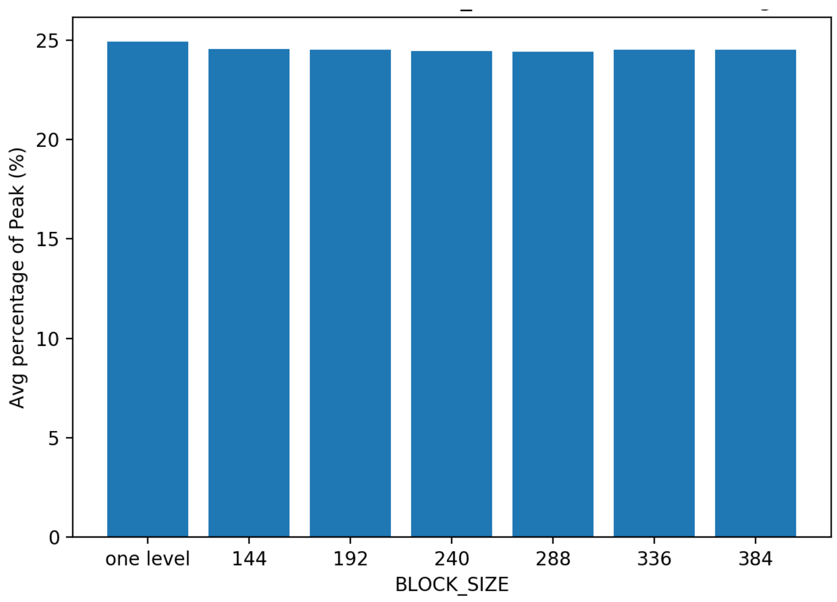

Figure 6. Impact of block size for two-level blocking implementation.

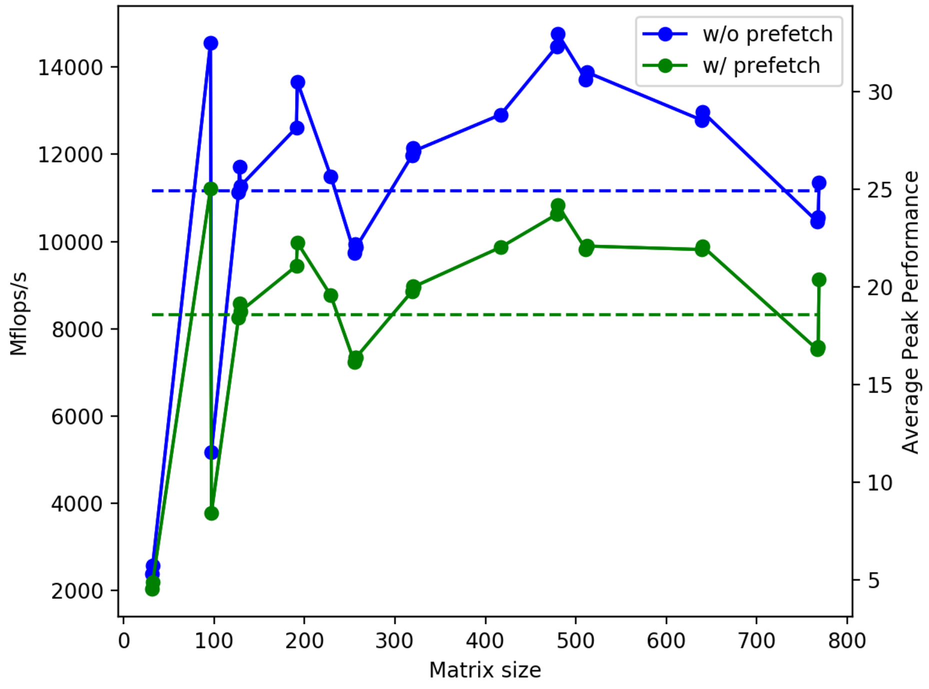

Figure 7. Impact of prefretching on performance.

## 2.7 Padding

In this section, we compare the performance between padding or not by fixing both global level and block level loop order to “jki”. We argue that the result of this huge difference might be the **block-level loop can be unrolled by compilers** when the loop condition is associated with a constant macro.

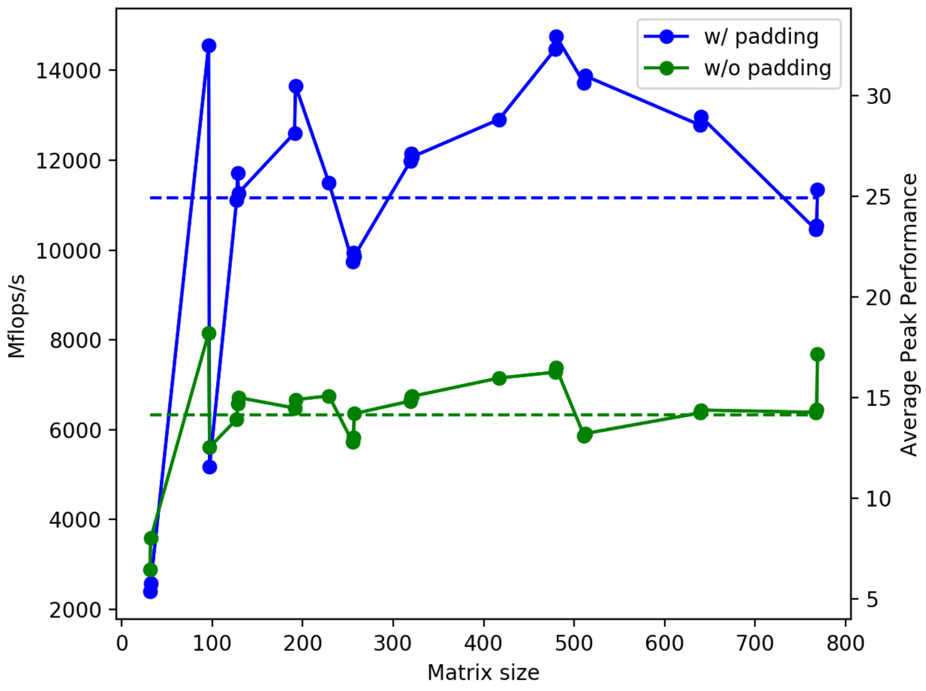

Figure 8. Impact of padding on performance.

## 2.8 Manual loop unrolling

Manual loop unrolling was implemented in order to reduce loop overhead and maximize instruction-level parallelism. A sequence of explicit operations were generated using Python's "cog" package. Surprisingly, the result between unrolling or not does not differ, as shown in Figure 9. This suggests that the compiler **already** did the loop unrolling **implicitly** when we gave the flag "-funroll-loops".

## 2.9 GotoBLAS

On the website, GotoBLAS method is mentioned in the reference section. To test the performance of this method, we also implemented this method follow the Fig 1. in "Anatomy of High-Performance Many-Threaded Matrix Multiplication". We tried out three variants: "jki" micro-kernel without packing, "kji" micro-kernel without packing and "kji" micro-kernel **with packing** in Figure 10. However, we do not achieve any speed up but a degradation. We do not know exactly why but we suspect maybe because the matrices A, B and C in our case are all square and GotoBLAS seems to be designed for long width matrix multiplication. In addition, the KNL node does not have L3 cache, which seems to be an essential part in GotoBLAS paper.

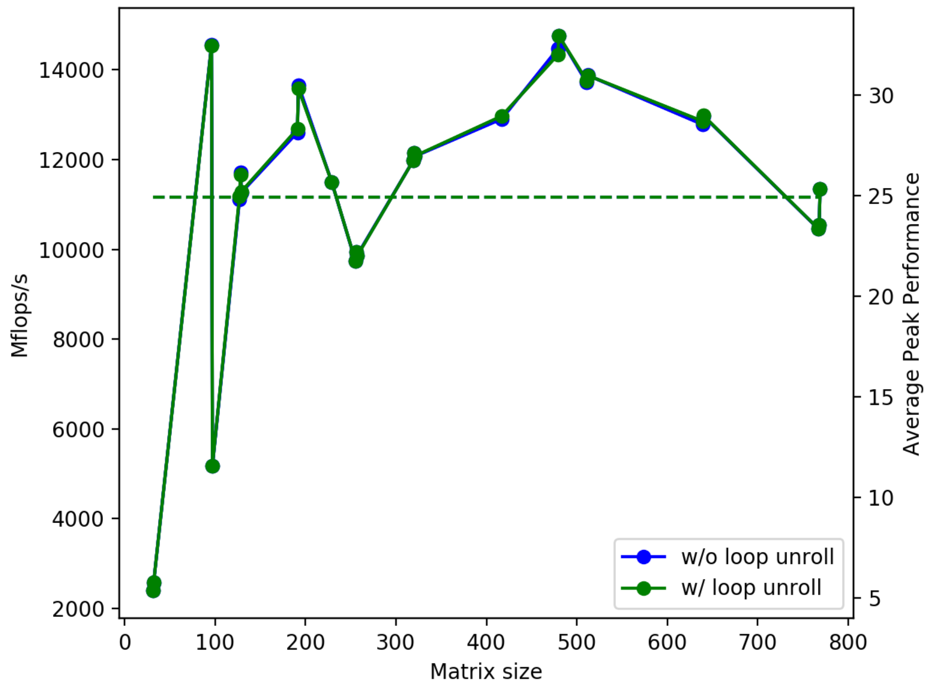

Figure 9. Impact of manual loop unrolling on performance.

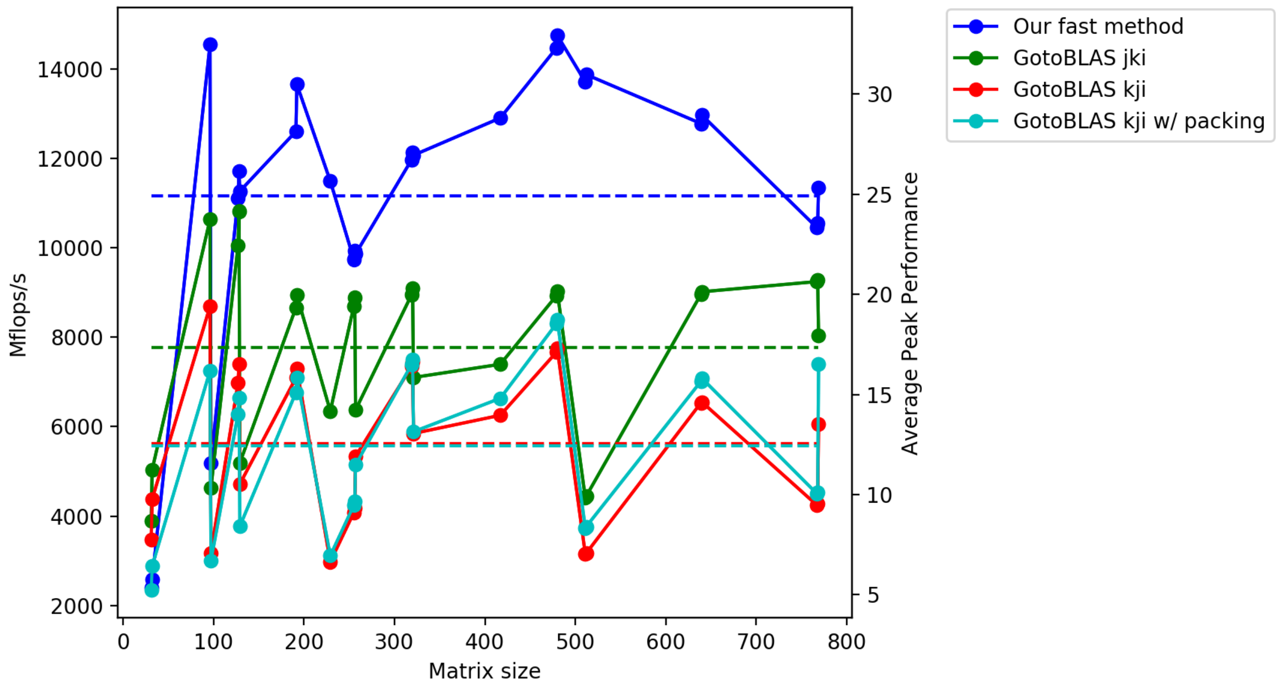

Figure 10. Comparison of different loop order GotoBLAS implementations.

### 2.10 Microkernel

Thanks to Mencher (Min-Chi) Chiang suggestions, we experimented a 24x8 kernel so that we can use 29 AVX512 registers (at most we have 32 AVX512 registers) in the most inner loop. The idea of this method is: in the block-level loop, we will have “ijk” loop order while “jki” for the global-level loop. We will compute a 24x8 small block in C, allowing us to declare (24/8) * 8 = 24 registers for C. Then we use 24/8 = 3 registers for a column in A and 2 registers for B. The reason that we choose only 2 registers for B is because KNL only have 2 vector lanes. This method achieves about **29%** of the peak performance and is used as the final result. It is worth mentioning that we choose BLOCK_SIZE = **168** for this method (to fit in 32K L1 cache) but due to the page limit we do not show the BLOCK_SIZE searching comparison.

## 3 Final Results

The final implementation employed blocking, padding, cacheline alignment and SIMD to achieve about **29%** average speed. Here is the fastest code (ignore copy function) compiled with “cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_C_FLAGS="-funroll-loops" ..”. (cog’s expansion code is ignored)

```
void square_dgemm_jki_microkernel(int N, double* A, double* B, double* C){
    int N_pad = (N + H - 1) / H * H;

    double *A_align = (double *)_mm_malloc(N_pad * N_pad * sizeof(double),
CACHELINE * sizeof(double));
    double *B_align = (double *)_mm_malloc(N_pad * N_pad * sizeof(double),
CACHELINE * sizeof(double));
    double *C_align = (double *)_mm_malloc(N_pad * N_pad * sizeof(double),
CACHELINE * sizeof(double));

    cpy(N_pad, N, A, A_align);
    cpy(N_pad, N, B, B_align);
    cpy(N_pad, N, C, C_align);

    for(int j = 0; j < N_pad; j += BLOCK_SIZE){
        for(int k = 0; k < N_pad; k += BLOCK_SIZE){
            for(int i = 0; i < N_pad; i += BLOCK_SIZE){
                const int M = min(N_pad - i, BLOCK_SIZE);
                const int N = min(N_pad - j, BLOCK_SIZE);
                const int K = min(N_pad - k, BLOCK_SIZE);

                double *A_block = A_align + i + k * N_pad;
                double *B_block = B_align + k + j * N_pad;
                double *C_block = C_align + i + j * N_pad;
                for(int ii = 0; ii < M; ii += H){
                    for(int jj = 0; jj < N; jj += W){
                        double *A_panel = A_block + ii;
                        double *B_panel = B_block + jj * N_pad;
                        double *C_block_small = C_block + ii + jj * N_pad;
/*[[[cog
import cog
H = 24
W = 8
NUM_B_REGISTERS = 2
CACHELINE = 8
# Declare registers for A
for i in range(H // CACHELINE):
    cog.out(
    """
    __m512d a{i};
    """.format(i=i)
    )
# Declare registers for B
for j in range(NUM_B_REGISTERS):
    cog.out(
    """
    __m512d b{j};
    """.format(j=j))
# Declare registers for C
for i in range(H // CACHELINE):
    for j in range(W):
        cog.out(
        """
        __m512d c{i}{j};
        c{i}{j} = _mm512_load_pd(C_block_small + {i} * {CACHELINE} + {j} *
N_pad);
        """.format(i=i, j=j, CACHELINE=CACHELINE)
        )
]]]*/

//[[[end]]]
                        for(int kk = 0; kk < K; ++kk){
                            double *A_col = A_panel + kk * N_pad;
                            double *B_row = B_panel + kk;
```

```
/*[[[cog
import cog
H = 24
W = 8
NUM_B_REGISTERS = 2
CACHELINE = 8
# Load A
for i in range(H // CACHELINE):
    cog.out(
    """
        a{i} = _mm512_load_pd(A_col + {i} * {CACHELINE});
    """.format(i=i, CACHELINE=CACHELINE)
    )
for i in range(H // CACHELINE):
    for j in range(W // NUM_B_REGISTERS):
        # Load B
        for k in range(NUM_B_REGISTERS):
            cog.out(
            """
            b{k} = _mm512_set1_pd(B_row[{j} * N_pad]);
            """.format(k=k, j=j*NUM_B_REGISTERS + k)
            )
        # Compute C and store C
        for k in range(NUM_B_REGISTERS):
            cog.out(
            """
            c{i}{j} = _mm512_fmadd_pd(a{i}, b{k}, c{i}{j});
            """.format(i=i, j=NUM_B_REGISTERS * j + k, k=k,
CACHELINE=CACHELINE)
            )
]]]*/
//[[[end]]]
                        }

/*[[[cog
import cog
H = 24
W = 8
CACHELINE = 8
# Store the C register back
for i in range(H // CACHELINE):
    for j in range(W):
        cog.out(
        """
        _mm512_store_pd(C_block_small + {i} * {CACHELINE} + {j} * N_pad, 
c{i}{j});
        """.format(i=i,j=j, CACHELINE=CACHELINE)
        )
]]]*/
//[[[end]]]
                    }
                }
            }
        }
    }

    for(int j = 0; j < N; j++){
        for(int i = 0; i < N; i++){
            C[i + j * N] = C_align[i + j * N_pad];
        }
    }

    _mm_free(A_align);
    _mm_free(B_align);
    _mm_free(C_align);
}
```

## 4 Conclusions

In this report, we demonstrate different techniques to optimize O(n^3) matrix multiplication. Several optimization techniques were implemented and experimented with, including two-level blocking, prefetching, memory repacking, and the GotoBlas implementation. Ultimately, loop reordering, utilizing Intel intrinsics SIMD, and optimizing the block size allowed us to achieve a peak performance of 29% in our final implementation.

## 5 Division of Work

Each group member independently attempted to write the most efficient code possible, and the code that produced the highest percentage of peak performance was utilized. Yu Gai first improved the starter code performance from 1% to 14% and Tzu-Chuan Lin's 29% code inspired by Mencher (Min-Chi) Chiang was ultimately used as the final result. The write up was a collaborative effort lead mostly by Ian Kolaja and Tzu-Chuan Lin.