Matrix Multiplication

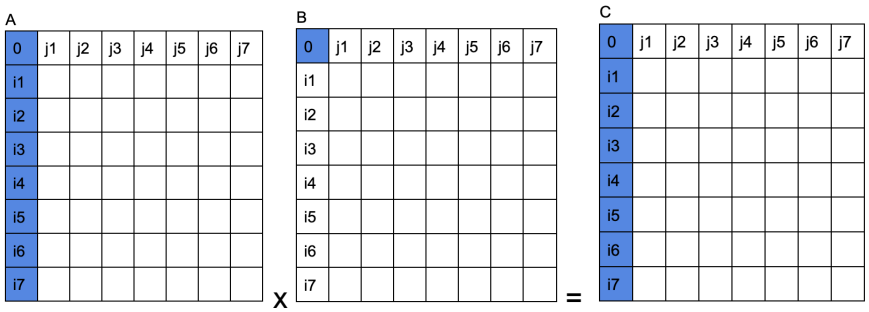
Group 64: Min-Chi, Jianglai, Lloyd

In homework1, after implementing three versions of code, we finally achieved 31.58% (35.33% for all sizes) average peak performance. Below is the code's explanation of our best result.

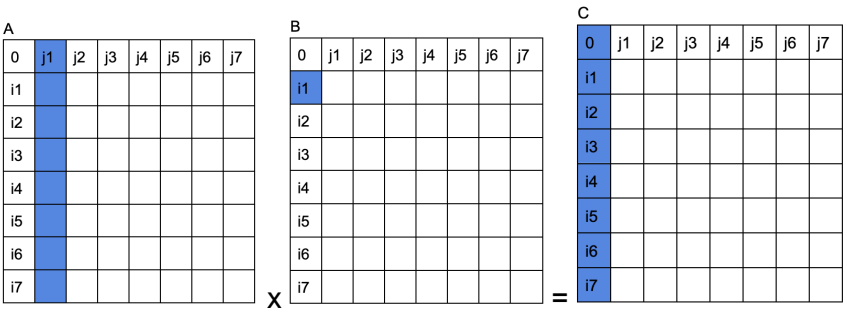
1. Rewrite Matrix Multiplication Algorithm + AVX512

For the original algorithm, element (i,j) of C is a dot product between row i of A and column j of B. With this algorithm, we'll encounter a huge overhead that we'll only be able to proceed to the next element in C after completing the previous one, which diminishes the possibility of parallelism.

We reference the paper *Anatomy of High-Performance Matrix Multiplication* and rewrite the algorithm to enable parallelism. The sample idea is the following:



For every inner k loop, we'll extract the continuous 8 elements at first, multiply by B(0,0), and generate the result for C([0:8], 0) elements, and so on.



Next, we extract the continuous 8 elements in the second column in A matrix, multiply them by B(1,0), and add them back on the C([0:8], 0) elements.

After completing the whole k rounds, we'll get results for the first 8 elements in C. In this way, we can parallelize the calculations without dependencies between elements in the

C matrix, and successfully maximize the instruction-level parallelism without reordering the for-loop.

In order to speed up, we employ the AVX512 intrinsics (`_mm512_loadu_pd`, `_mm512_set1_pd`, `_mm512_fmadd_pd`, `_mm512_store_pd`) to process all multiplication calculations in parallel, which significantly accelerate the calculation.

```
for (int i=0; i<M; i+=8) {
       for (int j=0; j<N; j++) {
           cij = _mm512_loadu_pd(C + (i + j * lda));

           for (int k=0; k<K; k++) {
               aik = _mm512_loadu_pd(A + (i + k * lda));
               bkj = _mm512_set1_pd(B[k + j * lda]);
               cij = _mm512_fmadd_pd(aik, bkj, cij);
           }
           _mm512_store_pd(C + i + j * lda, cij);
       }
   }
```

2. Loop Unrolling

For-loop will decrease the execution efficiency since the processor needs to keep checking whether the loop bounds have been exceeded. Therefore, eliminating expensive branches would be our priority.

According to the recitation, only 32 AVX512 vector registers are available, and we only have 2 FMA operations that can execute at one time. As a result, we decide to expand i to unroll 3 times (3*8 = 24) and j to unroll 8 times. In this way, we'll have to reserve 24 registers for the C matrix. Limited by FMA constraints, only 3 registers for A matrix and 2 registers for B matrix are required to perform FMA execution. We try our best to use up all available registers and concurrent FMA operations, and finally use 29 registers, which is below the 32 constraints.

```
   for (int i=0; i<M; i+=24) {
       for (int j=0; j<N; j+=8) {
           cij = _mm512_loadu_pd(C + (i + j * lda));
           cij1 = _mm512_loadu_pd(C + (i + (j+1) * lda));
           …
           ci8j = _mm512_loadu_pd(C + (i + 8 + j * lda));
           ci8j1 = _mm512_loadu_pd(C + (i + 8 + (j+1) * lda));
           …
```

```
            ci16j = _mm512_loadu_pd(C + (i + 16 + j * lda));
            ci8j1 = _mm512_loadu_pd(C + (i + 16 + (j+1) * lda));
            …

            for (int k=0; k<K; k++) {
                aik = _mm512_loadu_pd(A + (i + k * lda));
                ai8k = _mm512_loadu_pd(A + (i + 8 + k * lda));
                ai16k = _mm512_loadu_pd(A + (i + 16 + k * lda));

                bkj = _mm512_set1_pd(B[k + j * lda]);
                bkj1 = _mm512_set1_pd(B[k + (j+1) * lda]);

                cij = _mm512_fmadd_pd(aik, bkj, cij);
                cij1 = _mm512_fmadd_pd(aik, bkj1, cij1);

                ci8j = _mm512_fmadd_pd(ai8k, bkj, ci8j);
                ci8j1 = _mm512_fmadd_pd(ai8k, bkj1, ci8j1);

                ci16j = _mm512_fmadd_pd(ai16k, bkj, ci16j);
                ci16j1 = _mm512_fmadd_pd(ai16k, bkj1, ci16j1);
                …

            }
            _mm512_store_pd(C + i + j * lda, cij);
            …
        }
    }
```

3. Repack & Realign: Cache-align + Padding

SIMD instructions often access contiguous segments of 8 double words in memory. In order to make sure memory spaces for double elements align with the 64-byte cache boundary, we reallocate the memory space for A, B, C matrices by calling "`_mm_malloc`", and copy the content to a new matrix. Meanwhile, our unroll size for do_block function is 24. In order to make element access easier when performing matrix multiplication, we pad the matrix to be the multiple of 24, and fill created elements with 0.

4. Parameter Tuning

Block Size Fine-Tuning

We tested several BLOCK_SIZE parameters (e.g., 64, 128, 168, 192, 240, 256), and we found the optimal value for block size is 168.

Function Attributes
We tested several function attributes, and we finally added the “unroll-loop” flag into the do_block function, which successfully boosted the performance by 4%.

5. Column-Major Ordering + For-Loop Reordering

In order to make memory access continuous, we change our memory access to be Column-Wise Ordering. For example, since we reallocate the memory at the beginning, we use a column-major way to copy the content of the matrix.

```
for (int j = 0; j < n; j++) {
        for (int i = 0; i < n; i++) {
                A_cpy[i + n * j] = A[i + lda * j];
                B_cpy[i + n * j] = B[i + lda * j];
        }
    }
```

Meanwhile, we also reorder the for-loops from (i, j, k) to (j, k, i) since more memory access aligns with k & i layers.

```
    for (int j = 0; j < n; j += BLOCK_SIZE)
        for (int k = 0; k < n; k += BLOCK_SIZE)
            for (int i = 0; i < n; i += BLOCK_SIZE)
                do_block(...);
```

After implementation, we significantly increase average peak performance for our program by 9%. We conclude that matrix multiplication is a program bounded by memory access, and thus memory alignment is a key point for optimization.

6. Multi-Level Blocking

In order to fully utilize L1 and L2 cache, we implemented multi-level nested blocking for our code. For L1, since there are only 32 KB available memory, we set 36*36 block size for the first layer. For L2, under the constraints to align with 36 & 1MB available memory space, we set 180*180 block size for the second layer.

After implementing the feature, we found performance is only slightly improved. We discussed one of the possibilities causing this bottleneck is that we spend too much time on index transitioning and memory copying, which significantly slows down the program efficiency.

7. Assembly

To further improve performance we attempted writing a micro-kernel in assembly. We wanted to quickly generate assembly code and foolproof our implementation from small mistakes that naturally result when writing many lines of assembly. To do so we created a python parser that would take our existing SIMD micro-kernel and map it to assembly.

Our parser walks through each instruction and does two things: translates the instruction into assembly, and handling the registers used in each instruction. Translating the code assembly required peeling out the operands used in each instruction of the SIMD micro-kernel, finding the matching assembly instruction, and accounting for any differences in the instruction's behavior. We had separate logic to parse loads, stores, and our fused multiply-adds. Translating loads and stores was relatively straightforward as the intrinsic and assembly instructions were quite similar. Translating the fused multiply-adds required more care. Our intrinsic implementation relied on a set and an fmadd, but our assembly implementation needed to use the embedded broadcast available in the vfmadd231pd instruction. Each of our pair of set and fmadd intrinsics were turned into a single vfmadd instruction.

Below is our code to parse each instruction:

```
def parse_store(line):
    value_line = line[line.find("_mm512_store_pd(")+len("_mm512_store_pd("):]
    values = value_line[:value_line.rfind(")")]
    values = values.split(", ")
    memory_location = values[0].replace(" ", "")
    register = values[1].replace(" ", "")
    memory_location = array_and_index_to_variable(memory_location)
```

```
def parse_load(line):
    register_to_load_into = line[:line.find(" =")]
    value_line = line[line.find("_mm512_loadu_pd(")+len("_mm512_loadu_pd("):]
    value_to_add = value_line[:value_line.rfind(")")]
    value_variable = array_and_index_to_variable(value_to_add.replace(" ",""))
    asm_line = "\"vmovapd (%[{}]), %[{}]\\n\\t\"".format(value_variable, register_to_load_into)
    assignment_line = "double* {} = {};".format(value_variable, value_to_add)
    return asm_line, value_variable, register_to_load_into, assignment_line
```

```python
def parse_set_and_adds(lines):
    reg_to_broadcast_line = lines[0][lines[0].find("_mm512_set1_pd(")+len("_mm512_set1_pd("):]
    reg_to_broadcast_with_offset = reg_to_broadcast_line[:reg_to_broadcast_line.rfind(")")]
    reg_to_broadcast = array_and_index_to_variable(reg_to_broadcast_with_offset)
    assignment_line = "double* {} = {};".format(reg_to_broadcast, reg_to_broadcast_with_offset.replace("[", " + ").replace("]", ""))

    def gen_vfmadd_line(registers):
        return "\"vfmadd231pd (%[{}])%{{1to8%}}, %[{}], %[{}]\\n\\t\"".format(registers[1], registers[0], registers[2])

    def parse_fmadd_line(line, register_to_broadcast):
        registers_to_add_line = line[line.find("_mm512_fmadd_pd(")+len("_mm512_fmadd_pd("):]
        registers_to_add = registers_to_add_line[:registers_to_add_line.rfind(")")].split(", ")
        registers_to_add[1] = register_to_broadcast
        line = gen_vfmadd_line(registers_to_add)
        registers_to_add.remove(register_to_broadcast)
        return line, registers_to_add

    line_one, registers_to_add_one = parse_fmadd_line(lines[1], reg_to_broadcast)
    line_two, registers_to_add_two = parse_fmadd_line(lines[2], reg_to_broadcast)

    return line_one, line_two, registers_to_add_one + registers_to_add_two, reg_to_broadcast, assignment_line
```

After this step our instructions would be translated into valid assembly, but we wouldn't have told the assembler the registers we touch and operands relying on pointers and offsets would not be valid. The first fix was simple, as we generate assembly we keep track of registers and memory locations we use and append these to the instruction list at the end. The second problem required more thought. Although we knew the offset to use when loading the values in from the matrices the offset depended on the dimensions of the original matrix, and was not constant preventing us from generating valid assembly. This was a result of our original micro-kernel using the initial arrays when loading in values. To fix this we added an intermediary step where we would migrate the blocks we were considering in each matrix into arrays that contained only those values. This allowed our assembly micro-kernel to assume the size of the array and use constant offsets when performing load, stores, and fused multiply-adds.

Below is our step to create individual blocks, and load the result:

```c
// Initialize blocks starting at point (i,j,k)
for (int y = 0; y < MICROKERNEL_SIZE; y++) {
    for (int x = 0; x < MICROKERNEL_SIZE; x++) {
        block_A[x + y * MICROKERNEL_SIZE] = A[i + k * n + x + y * n];
        block_B[x + y * MICROKERNEL_SIZE] = B[k + j * n + x + y * n];
        block_C[x + y * MICROKERNEL_SIZE] = C[i + j * n + x + y * n];
    }
}
// Perform individual block dgemm
microkernel_8by8(block_A, block_B, block_C);
// Read our block c values into original c.
for (int y = 0; y < MICROKERNEL_SIZE; y++) {
    for (int x = 0; x < MICROKERNEL_SIZE; x++) {
        C[i + j * n + x + y * n] = block_C[x + y * MICROKERNEL_SIZE];
    }
}
```

Below are the loading, and fused multiply-add assembly instructions (stores omitted):

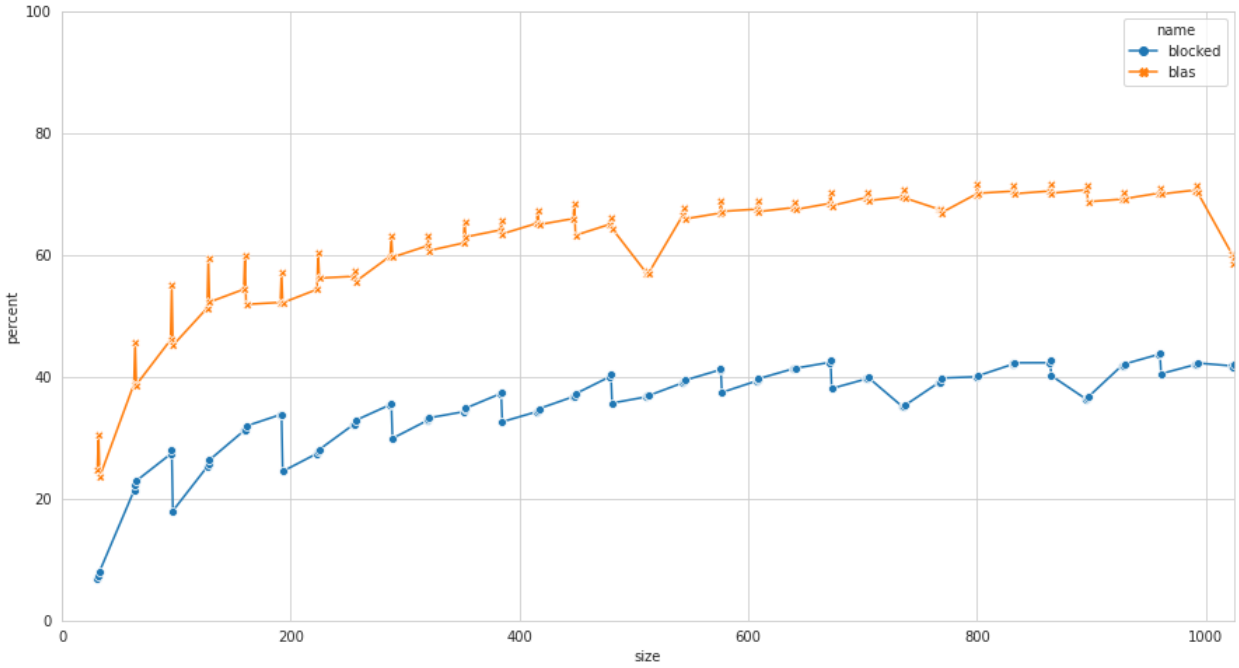
```
asm volatile (
    "vmovapd 0(%[C]), %[cij]\n\t"
    "vmovapd 64(%[C]), %[cij1]\n\t"
    "vmovapd 128(%[C]), %[cij2]\n\t"
    "vmovapd 192(%[C]), %[cij3]\n\t"
    "vmovapd 256(%[C]), %[cij4]\n\t"
    "vmovapd 320(%[C]), %[cij5]\n\t"
    "vmovapd 384(%[C]), %[cij6]\n\t"
    "vmovapd 448(%[C]), %[cij7]\n\t"
    : [cij] "+v" (cij),
    [cij1] "+v" (cij1),
    [cij2] "+v" (cij2),
    [cij3] "+v" (cij3),
    [cij4] "+v" (cij4),
    [cij5] "+v" (cij5),
    [cij6] "+v" (cij6),
    [cij7] "+v" (cij7)
    : [C] "r" (C)
    : "memory"
);
```

```
double* Aik = A + (k * MICROKERNEL_SIZE);
double* Bkj = B + k;
asm volatile (
    "vmovapd (%[Aik]), %[aik]\n\t"
    "vfmadd231pd 0(%[Bkj])%{1to8%}, %[aik], %[cij]\n\t"
    "vfmadd231pd 64(%[Bkj])%{1to8%}, %[aik], %[cij1]\n\t"
    "vfmadd231pd 128(%[Bkj])%{1to8%}, %[aik], %[cij2]\n\t"
    "vfmadd231pd 192(%[Bkj])%{1to8%}, %[aik], %[cij3]\n\t"
    "vfmadd231pd 256(%[Bkj])%{1to8%}, %[aik], %[cij4]\n\t"
    "vfmadd231pd 320(%[Bkj])%{1to8%}, %[aik], %[cij5]\n\t"
    "vfmadd231pd 384(%[Bkj])%{1to8%}, %[aik], %[cij6]\n\t"
    "vfmadd231pd 448(%[Bkj])%{1to8%}, %[aik], %[cij7]\n\t"
    : [aik] "+v" (aik),
    [cij] "+v" (cij),
    [cij6] "+v" (cij6),
    [cij5] "+v" (cij5),
    [cij4] "+v" (cij4),
    [cij2] "+v" (cij2),
    [cij3] "+v" (cij3),
    [cij7] "+v" (cij7),
    [cij1] "+v" (cij1)
    : [Aik] "r" (Aik),
    [Bkj] "r" (Bkj)
    : "memory"
);
```

After implementing this we found that our performance was around half of our original SIMD micro-kernel. We can think of a few reasons why this was the case but were unable to optimize the assembly version of our code before the deadline. We theorized that copying blocks was our biggest bottleneck and hoped that a larger block size would solve this problem. However, after experimenting with much larger blocking our performance worsened (likely as a result of conditional statements to determine what block size to use at each part of the matrix. We also believe that there might be a more efficient way to create our blocks such as using memcpy. Ultimately, we stuck with our SIMD micro-kernel.

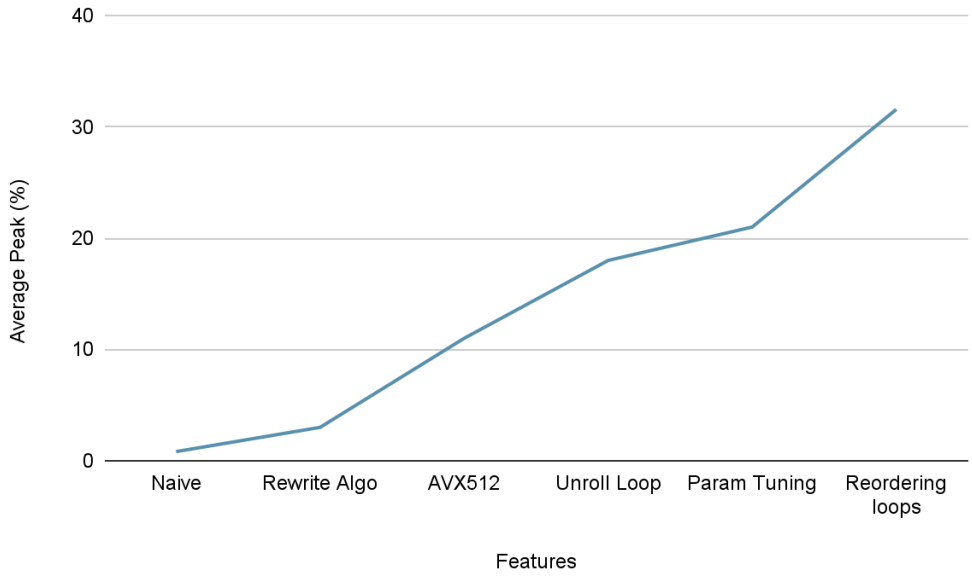
Benchmarking

After implementing all features, we finally got the average performance of peak 31.58% (35.33% for all sizes). The below graph is the result comparison with blas.

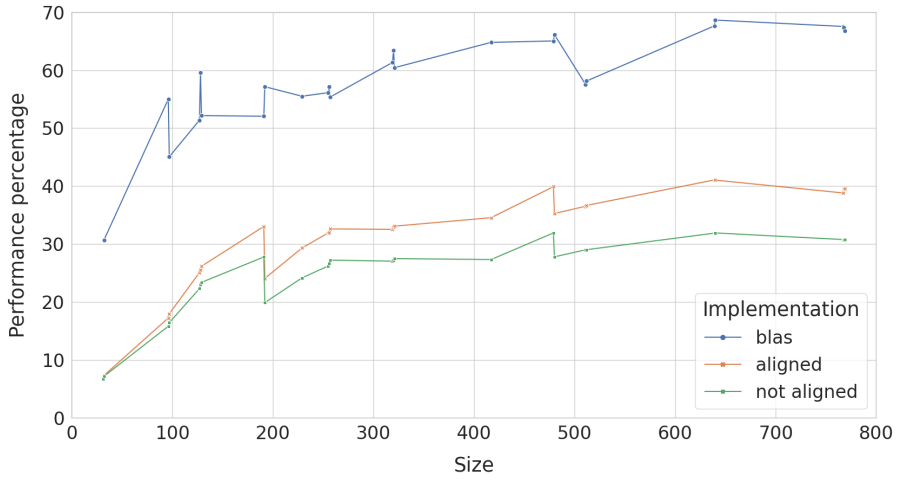


As we can see, we alleviate the drop when the matrix size is equal to the power of 2. We think the main reason is that our code allows storing values for the C matrix to be parallelized, which maximizes the overall average performance. However, there's always a drop when the matrix size is slightly above the multiple of 24 (24 is the size we pad our matrix). We think the main reason is that these matrices are required to pad more than other matrices. (For example, for size 97, they need another 23 empty elements to get expanded to 120. But for size 119, they only need an extra 1 to be 120.)

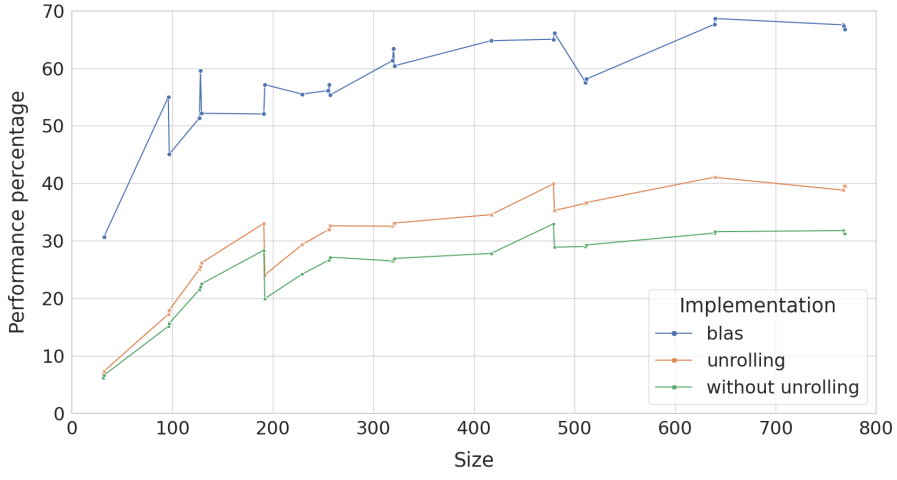
Below are the specific details of performance improvement after implementing features.



In our early attempts, we found that memory realignment is crucial to improve the overall performance since apparently the SIMD instructions would not be able to realize its full potential without target memory location being aligned with the cache boundary. To be specific, implementing memory alignment can boost the performance by around 5% - 6% based on our test.

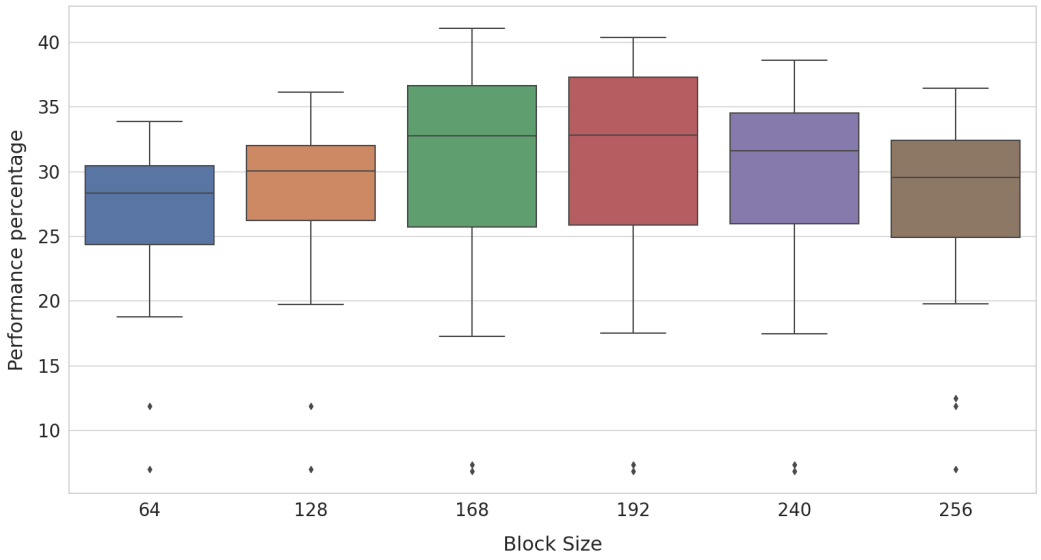


After getting a substantial performance improvement with SIMD instructions in our code, we also found that unrolling loops might contribute to the program efficiency at the compiler's level. For GCC, it is possible to either using #progma directives or adding a proper function attribute. We eventually added the function attribute explicitly in its signature to instruct the compiler to unroll the loops in it. It improved the overall performance by about 4% - 5%.



In order to get the best performance, we also did tests on various block sizes and checked the performance variance. It is noteworthy that we need to adjust the matrix padding parameter and step size in our loops accordingly and slightly modify the algorithm with unrolling loops. For

example, 168, 192 and 240 are divisible by 24, while 64, 128, and 256 are not, which leads us to use 24 as the padding / step size for the former group and 16 for the latter. Based on our experiment results, the blocks divisible by larger step size work better in general. Eventually, we chose 168 since it resulted in the best overall performance on all the matrix sizes. We visualize the performance distribution below, which was tested under different input matrix sizes.



To sum up, the performance boost is realized by a series of features. For example, only implementing AVX512 on naive block matrix multiplication would only reach a 12% average peak. However, if we rewrite the multiplication algorithm first, parallelizing the writes operations for different elements in the C matrix, we can gain roughly 16% average peak after implementing AVX512. We eventually achieve our final result with successful integration of all these features.

Contribution

Min-Chi Chiang: Finished majority of features, codes, and project write-up.
Jianglai Dai: Tried multi-level blocking & implemented visualization script
Lloyd Brown: Attempted to implement inline assembly