Matrix Multiplication

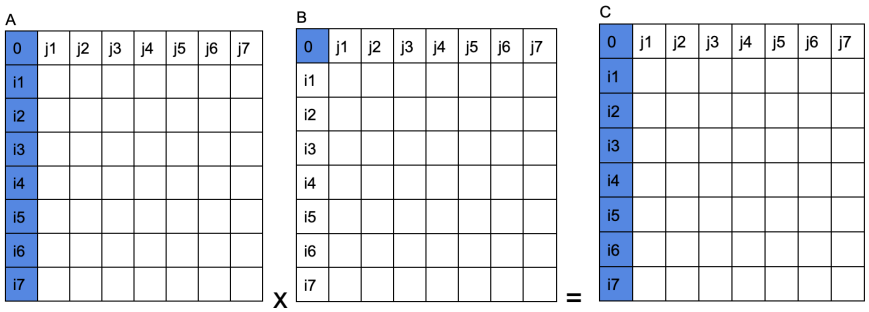
Group 64: Min-Chi, Jianglai, Lloyd

In homwork1, after implementing three versions of code, we finally achieved 31.58% (35.33% for all size) average peak performance. Below is the code's explanation of our best result.

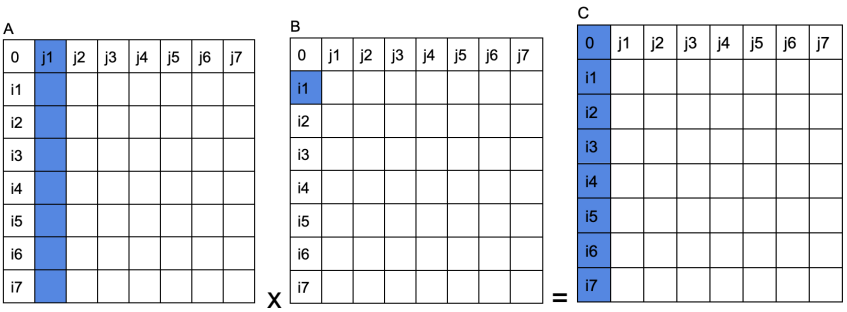
1. Rewrite Matrix Multiplication Algorithm + AVX512

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 second column in A matrix, multiply by B(1,0), and still accumulate back to result for 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 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 9% average peak performance for our program. We conclude that matrix multiplication is a program bounded by memory access, and thus memory alignment is a key point for optimization.

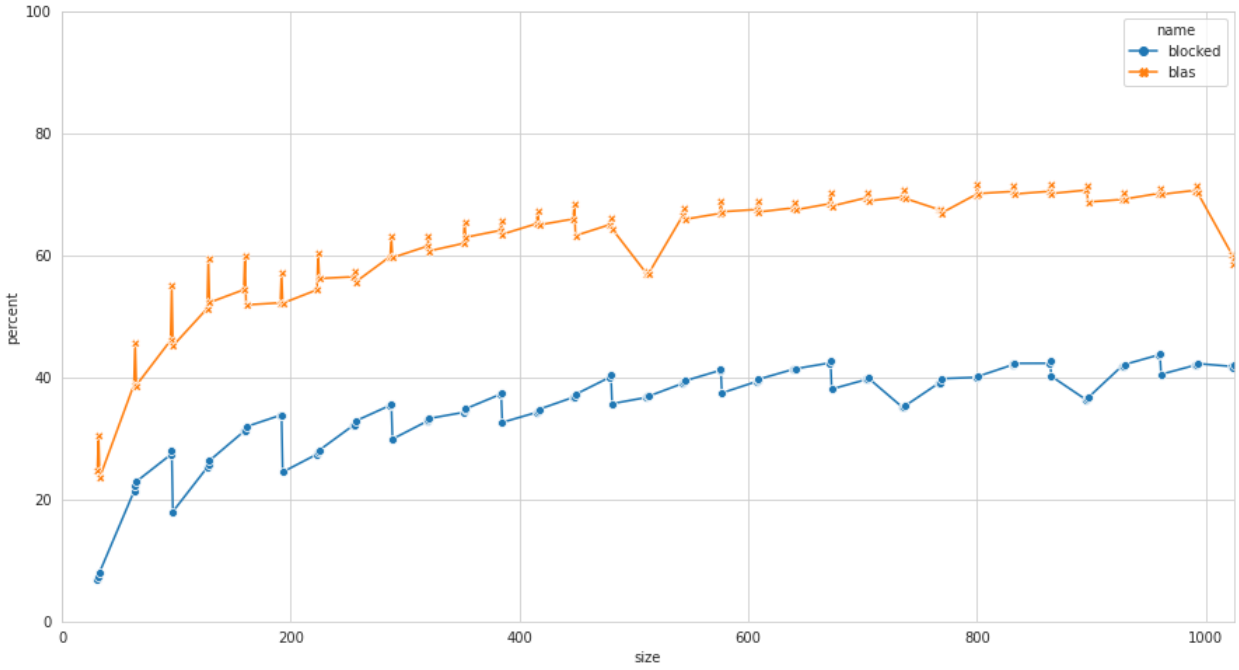
6. Multi-Level Blocking

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 slow down the program efficiency.

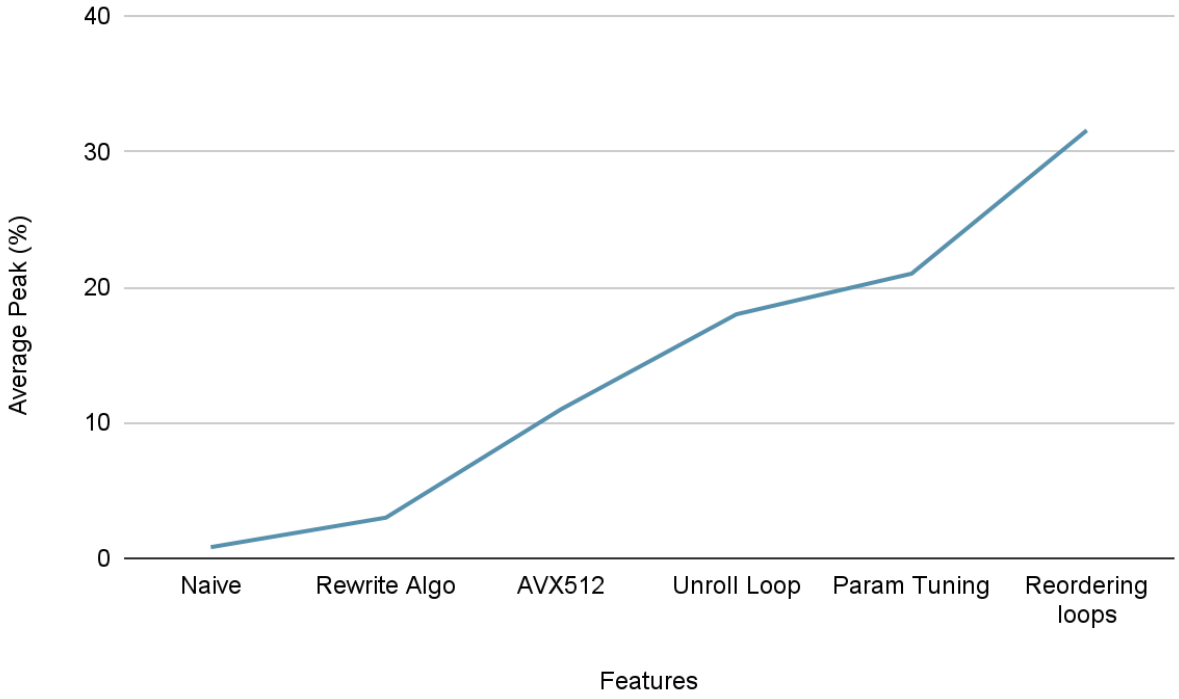
Benchmarking

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



As we can see, we alleviate the drop when the block size is equal to the power of 2. We think the main reason is that our code allows storing values for C matrix to be parallelized, which maximizes the overall average performance. However, there's always a drop when the block 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 our matrix. (For example, for size 97, they need to plus 23 to expand to 120. But for size 119, they only need to plus 1 to be 120.)

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



As we can see, the performance boost is caused by a series of features integration. For example, only implementing AVX512 on naive block matrix multiplication would only acquire 12% average peak. However, if we rewrite the multiplication algorithm first, parallelizing the writes operations for different elements in C matrix, we can gain roughly 16% average peak after implementing AVX512.

Contribution

Min-Chi Chiang: Implement all features mentioned above, majority of codes and finish the whole project write-up.
Jianglai Dai: Write python code to graph our performance
Lloyd Brown: Try to implement inline assembly implementation