A1 Report

Task 1

Simple matrix multiplication.

```
void mulMatWithCleanMemory(const double *mat1, const double *mat2, double *result) {
  memset(result, 0, sizeof(double) * R1 * C2);
  for (int i = 0; i < R1; i++)
    for (int j = 0; j < C2; j++)
      for (int k = 0; k < C1; k++)
        result[i * C2 + j] += mat1[i * C1 + k] * mat2[k * C2 + j];
}
```

Task 2

When improving the previous implementation, we first modified the memory access pattern. The following line causes the main issue because access to matrix 2 has a stride pattern - as opposed to contiguous access, which would be much more efficient:

```
result[i * C2 + j] += mat1[i * C1 + k] * mat2[k * C2 + j];
```

Our solution was to transpose matrix 2 to have a row-major order. Now, when accessing both matrices, the 'k' variable dictates the advancement - eliminating the need to jump by 'k*C2' in every iteration.

Transposing

```
void transpose(double *ogMat, double *tpMat) {
  memset(tpMat, 0, sizeof(double) * R2 * C2);
  int cnt = 0;
  for (int i = 0; i < C2; i++)
    for (int j = 0; j < R2; j++) {
      tpMat[cnt] = ogMat[C2 * j + i];
      cnt++;
    }
}
```

The transposing method creates a new matrix from matrix 2 by converting each column into a row. In all of the following experiments, the transposed matrix will be used.

Blocking/Tiling

```
void mulMatBlocked(const double *mat1, const double *mat2T, double *result) {
  const int BLOCK_SIZE = 256; // Cache size
  memset(result, 0, sizeof(double) * R1 * C2);

  for (int i0 = 0; i0 < R1; i0 += BLOCK_SIZE)
    for (int j0 = 0; j0 < C2; j0 += BLOCK_SIZE)
      for (int k0 = 0; k0 < C1; k0 += BLOCK_SIZE)

        for (int i = i0; i < std::min(i0 + BLOCK_SIZE, R1); i++)
          for (int j = j0; j < std::min(j0 + BLOCK_SIZE, C2); j++) {
            double sum = 0;
            for (int k = k0; k < std::min(k0 + BLOCK_SIZE, C1); k++)
              sum += mat1[i * C1 + k] * mat2T[j * C1 + k];
            result[i * C2 + j] += sum;
```

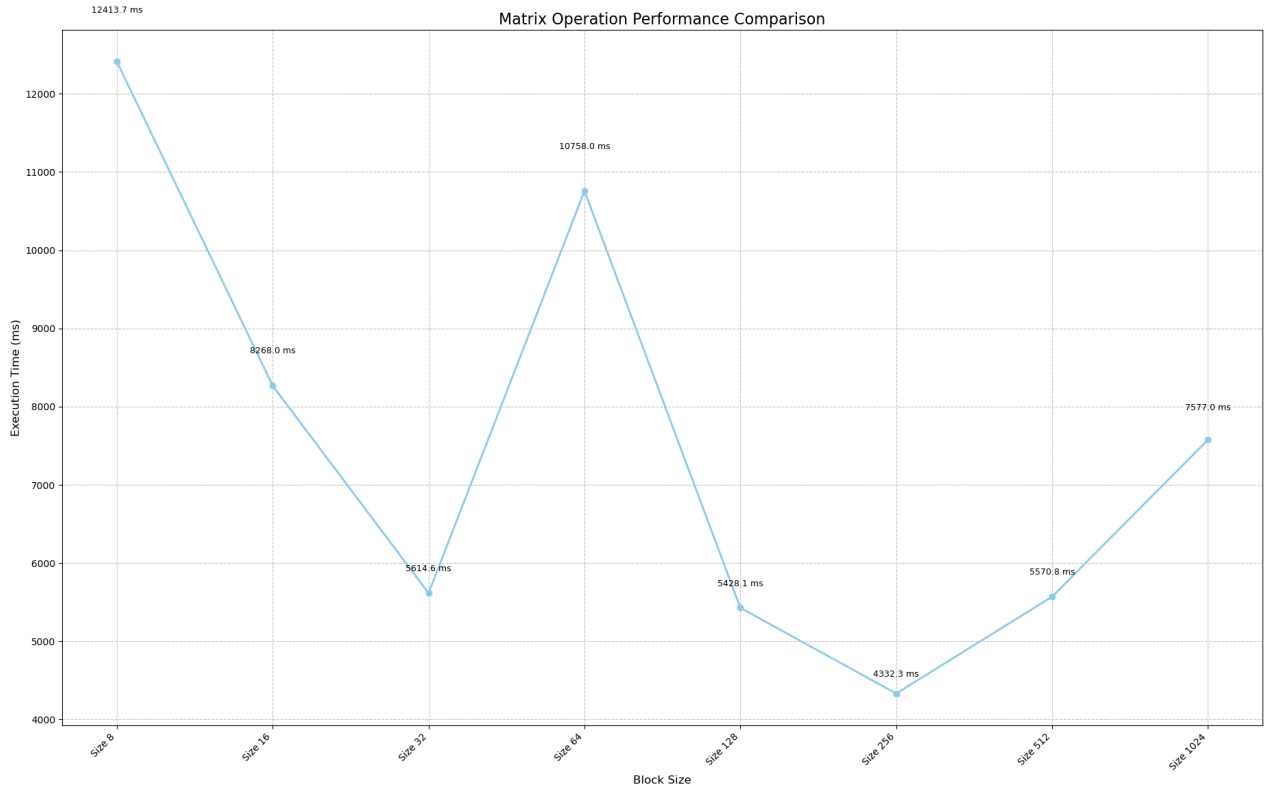
```
            }
}
```

Next, we implemented Blocking to decrease cache misses. With the appropriate size, all three sub-matrices can fit inside the cache - decreasing trips to main memory. The test system has an i5 1135G with the following cache configuration:

```
Caches (sum of all):
  L1d:                    192 KiB (4 instances)
  L1i:                    128 KiB (4 instances)
  L2:                     5 MiB (4 instances)
  L3:                     8 MiB (1 instance)
```

Through testing, we found that a 256 block size yields the best performance. With simple calculations, we observed that this block size adds up to roughly 1.5MiB (`256 * 256 * 3 * 64bit`), which fits comfortably inside our L3 cache. The L1 and L2 caches are too small for consideration. Specifically, since they are private caches dedicated to each thread, when running on a single thread, only 1/4th of the total cache is available.

However, we found it interesting that this leaves a rather large unused space of roughly 6.5MiB in L3, yet increasing the block size only made performance worse. We assume this is because L3 is a uniform memory structure that needs to reserve space for instructions as well. Additionally, there is the matter of Cache Associativity - from our research, this CPU's L3 is 16-way associative. This means each set of L3 is 512KiB, the exact same size as one of our blocks (`256*256*64bit`). We believe this matching of set size allowed the algorithm to execute at its best, outperforming more apparently reasonable block sizes like 32.



Unrolling

```
void mulMatWithUnrolledK(const double *mat1, const double *mat2T, double *result) {
  int UNROLL = 8;
  memset(result, 0, sizeof(double) * R1 * C2);
  for (int i = 0; i < R1; i++)
    for (int j = 0; j < C2; j++) {
```

```
      double sum = 0;
      for (int k = 0; k < C1; k += UNROLL) {
        sum += mat1[i * C1 + k] * mat2T[j * C1 + k];
        sum += mat1[i * C1 + k + 1] * mat2T[j * C1 + k + 1];
        sum += mat1[i * C1 + k + 2] * mat2T[j * C1 + k + 2];
        sum += mat1[i * C1 + k + 3] * mat2T[j * C1 + k + 3];
        sum += mat1[i * C1 + k + 4] * mat2T[j * C1 + k + 4];
        sum += mat1[i * C1 + k + 5] * mat2T[j * C1 + k + 5];
        sum += mat1[i * C1 + k + 6] * mat2T[j * C1 + k + 6];
        sum += mat1[i * C1 + k + 7] * mat2T[j * C1 + k + 7];
      }
      result[i * C2 + j] = sum;
    }
  }
```

Initially, we weren't expecting much from loop unrolling. In essence, it only allows us to decrease the number of conditional checks the loop needs to perform. In the above example, we unrolled by 8, so the number of conditional checks needed is 1/8 of the original quantity. However, one of the main reasons loop unrolling is implemented is to make implementing AVX512 instructions easier, since we can store and operate on all 8 (`512=8*64`) values in one iteration.

SIMD instructions: `AVX512`

Our final optimization implemented SIMD instructions from the AVX512 family. Specifically, we used `FMADD`, which is a combined method for multiply and add operations. When three vectors are provided, it multiplies two of them and adds the results to the third - perfect for our use case. To extract the final scalar value, we used a reduction by sum method. As you can see, we unrolled the inner-most loop by 8, since a 512-bit vector can contain 8 doubles. This way we can efficiently process all unrolled values at once, as if it were a normal scalar iteration - except for the reduction step.

Synopsis

```
__m512d _mm512_fmadd_pd (__m512d a, __m512d b, __m512d c)
#include <immintrin.h>
Instruction: vfmadd132pd zmm, zmm, zmm
             vfmadd213pd zmm, zmm, zmm
             vfmadd231pd zmm, zmm, zmm
CPUID Flags: AVX512F
```

Description

Multiply packed double-precision (64-bit) floating-point elements in a and b, add the intermediate result to packed elements in c, and store the results in dst.

Operation

```
FOR j := 0 to 7
        i := j*64
        dst[i+63:i] := (a[i+63:i] * b[i+63:i]) + c[i+63:i]
ENDFOR
dst[MAX:512] := 0
```

Latency and Throughput

Architecture	Latency	Throughput (CPI)
Icelake Intel Core	4	1
Icelake Xeon	4	0.5
Sapphire Rapids	4	0.5
Skylake	4	0.5

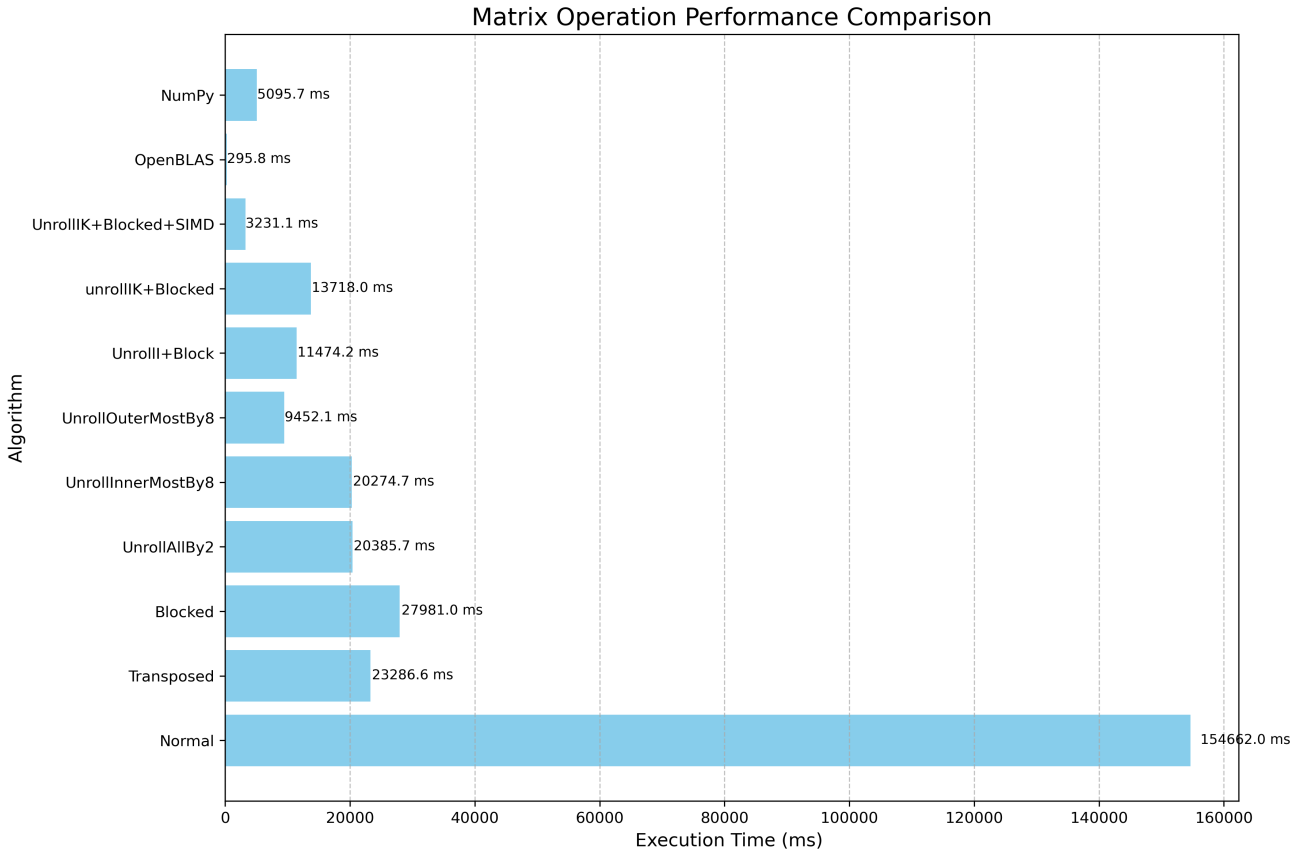
```
  // BLOCK
  for (int i0 = 0; i0 < R1; i0 += BLOCK_SIZE)
    for (int j0 = 0; j0 < C2; j0 += BLOCK_SIZE)
      for (int k0 = 0; k0 < C1; k0 += BLOCK_SIZE)
        // UNROLL
```

```
        for (int i = i0; i < i0 + BLOCK_SIZE; i += UNROLL)
          for (int j = j0; j < j0 + BLOCK_SIZE; j++) {
            __m512d sum0 = _mm512_set1_pd(0.0);
            __m512d sum1 = _mm512_set1_pd(0.0);
            __m512d sum2 = _mm512_set1_pd(0.0);
            __m512d sum3 = _mm512_set1_pd(0.0);
            __m512d sum4 = _mm512_set1_pd(0.0);
            __m512d sum5 = _mm512_set1_pd(0.0);
            __m512d sum6 = _mm512_set1_pd(0.0);
            __m512d sum7 = _mm512_set1_pd(0.0);

            for (int k = k0; k < k0 + BLOCK_SIZE; k += UNROLL) {
              __m512d m2 = _mm512_loadu_pd(&mat2T[j * C1 + k]);

              __m512d m1_0 = _mm512_loadu_pd(&mat1[i * C1 + k]);
              sum0 = _mm512_fmadd_pd(m1_0, m2, sum0);

              __m512d m1_1 = _mm512_loadu_pd(&mat1[(i + 1) * C1 + k]);
              sum1 = _mm512_fmadd_pd(m1_1, m2, sum1);

              __m512d m1_2 = _mm512_loadu_pd(&mat1[(i + 2) * C1 + k]);
              sum2 = _mm512_fmadd_pd(m1_2, m2, sum2);

              __m512d m1_3 = _mm512_loadu_pd(&mat1[(i + 3) * C1 + k]);
              sum3 = _mm512_fmadd_pd(m1_3, m2, sum3);

              __m512d m1_4 = _mm512_loadu_pd(&mat1[(i + 4) * C1 + k]);
              sum4 = _mm512_fmadd_pd(m1_4, m2, sum4);

              __m512d m1_5 = _mm512_loadu_pd(&mat1[(i + 5) * C1 + k]);
              sum5 = _mm512_fmadd_pd(m1_5, m2, sum5);

              __m512d m1_6 = _mm512_loadu_pd(&mat1[(i + 6) * C1 + k]);
              sum6 = _mm512_fmadd_pd(m1_6, m2, sum6);

              __m512d m1_7 = _mm512_loadu_pd(&mat1[(i + 7) * C1 + k]);
              sum7 = _mm512_fmadd_pd(m1_7, m2, sum7);
            }

            // Vector -> Scalar
            double hsum0 = _mm512_reduce_add_pd(sum0);
            double hsum1 = _mm512_reduce_add_pd(sum1);
            double hsum2 = _mm512_reduce_add_pd(sum2);
            double hsum3 = _mm512_reduce_add_pd(sum3);
            double hsum4 = _mm512_reduce_add_pd(sum4);
            double hsum5 = _mm512_reduce_add_pd(sum5);
            double hsum6 = _mm512_reduce_add_pd(sum6);
            double hsum7 = _mm512_reduce_add_pd(sum7);

            result[i * C2 + j] += hsum0;
            result[(i + 1) * C2 + j] += hsum1;
            result[(i + 2) * C2 + j] += hsum2;
            result[(i + 3) * C2 + j] += hsum3;
            result[(i + 4) * C2 + j] += hsum4;
            result[(i + 5) * C2 + j] += hsum5;
            result[(i + 6) * C2 + j] += hsum6;
            result[(i + 7) * C2 + j] += hsum7;
          }
```

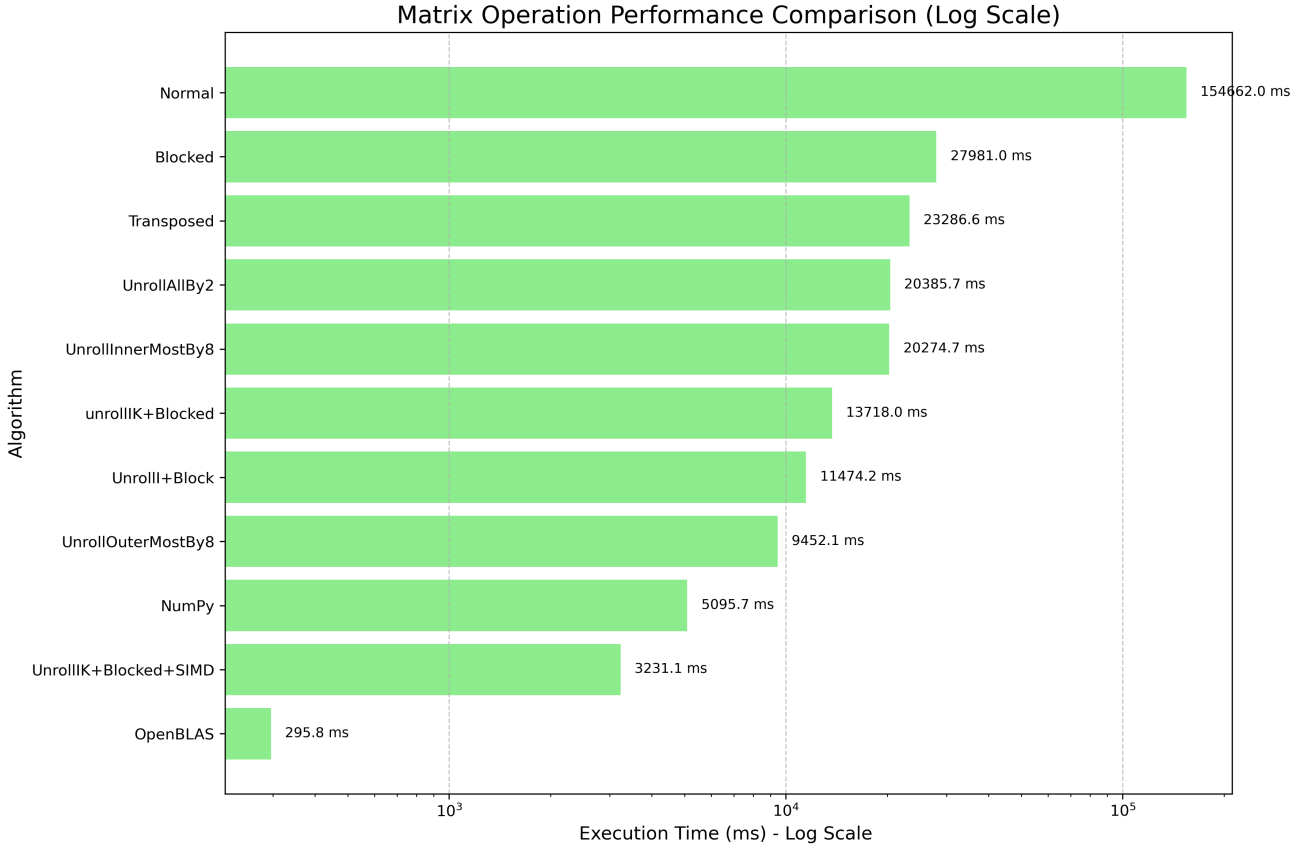
Matrix Multiplication Performance Comparison

The test was conducted on 2048 by 2048 matrices containing randomly generated double precision floating point numbers ranging from 0 to 2048.

Initially, we ran our algorithm without utilizing GCC compiler's optimization flag (-O).

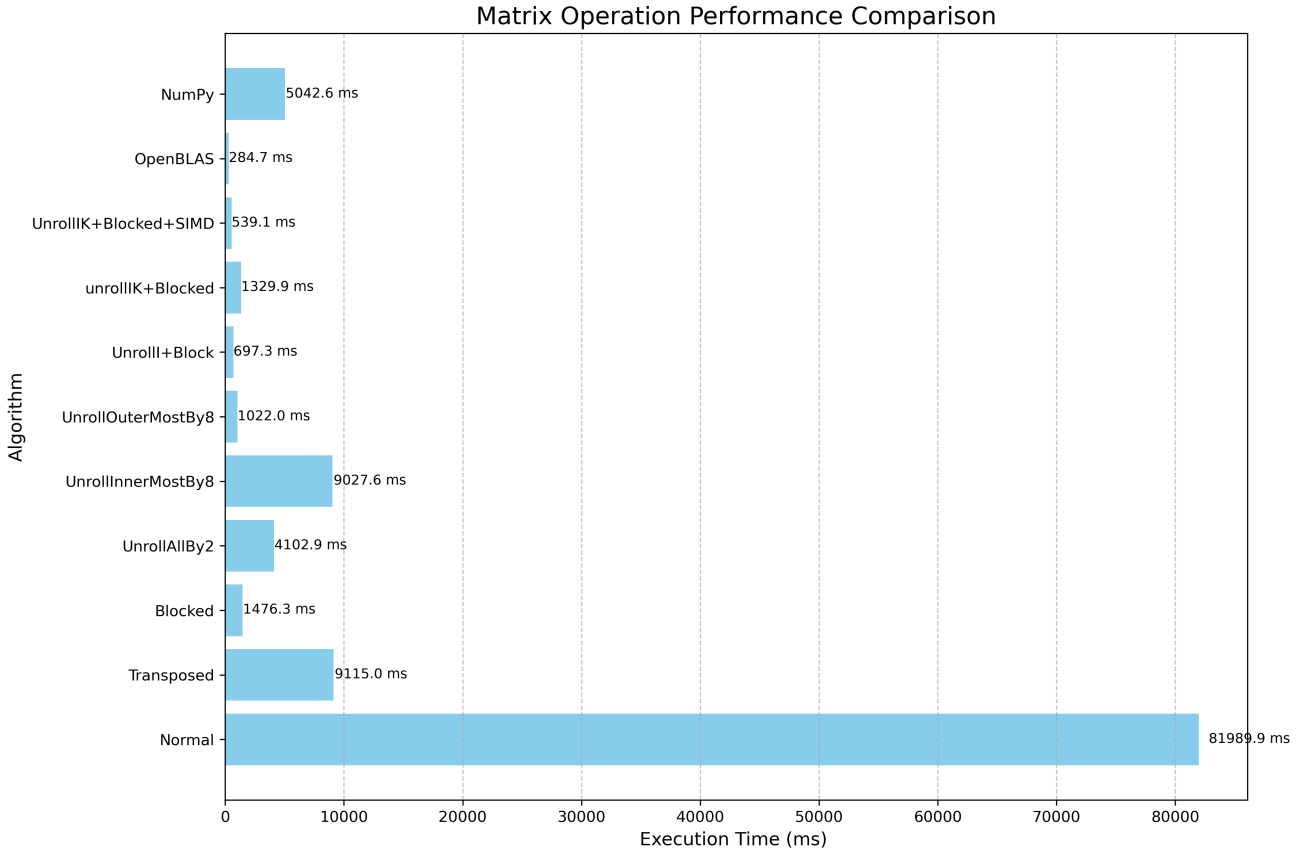


Our best effort (UnrollIK+Blocked+SIMD) completes in a little over 3 seconds, while NumPy takes around 5 seconds. This indicates the significance of SIMD instructions, since none of our other implementations came close to this performance. Additionally, we can see how much transposing the matrix impacts performance. Just by transposing, the same algorithm sped up by 130 seconds, reducing execution time from 154 seconds. Unfortunately, every time Blocking was implemented without optimization flags, it only introduced more overhead - leading to performance degradation.



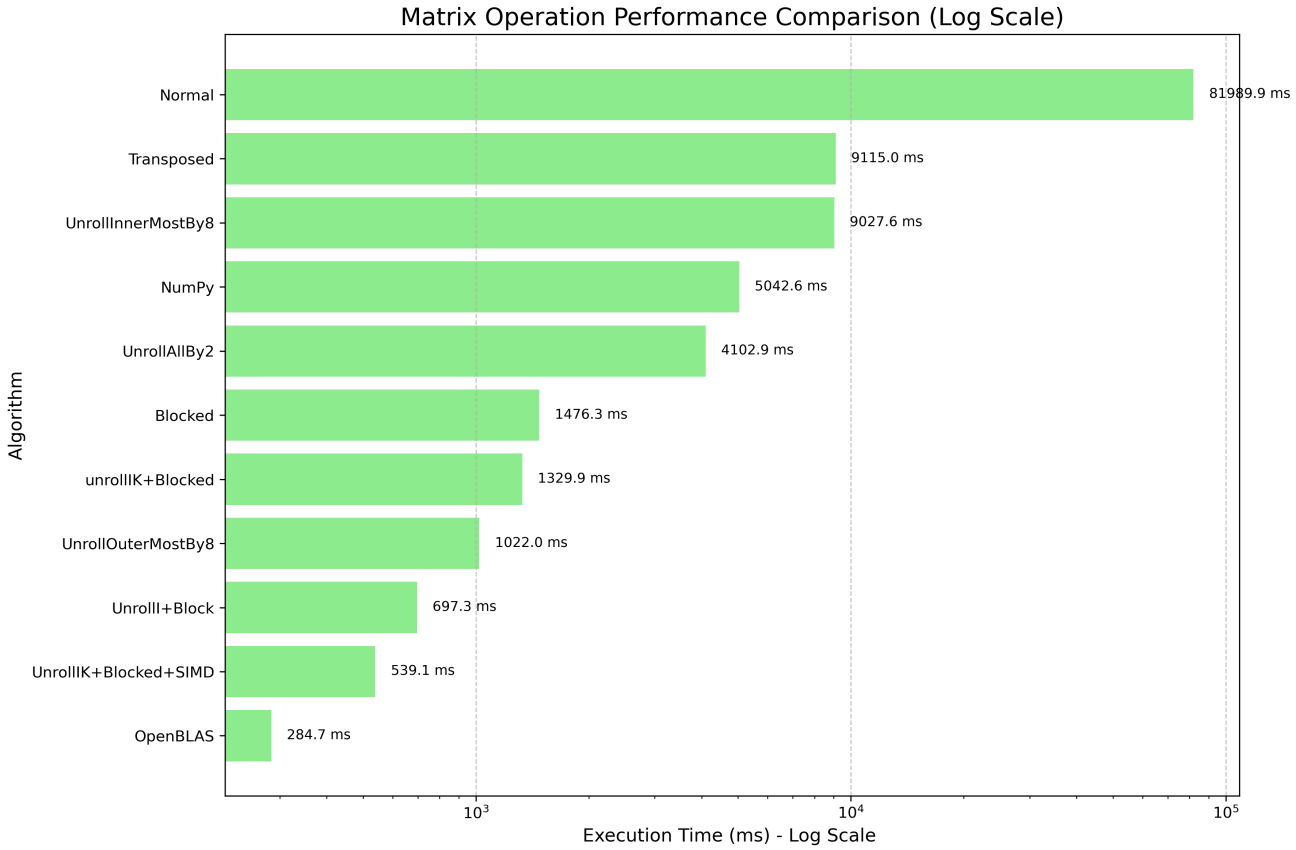
Every algorithm was executed on transposed matrix 2, except for the "Normal" implementation.

The following results were gathered after enabling the optimizer flag. The flag was set to `-Ofast`.

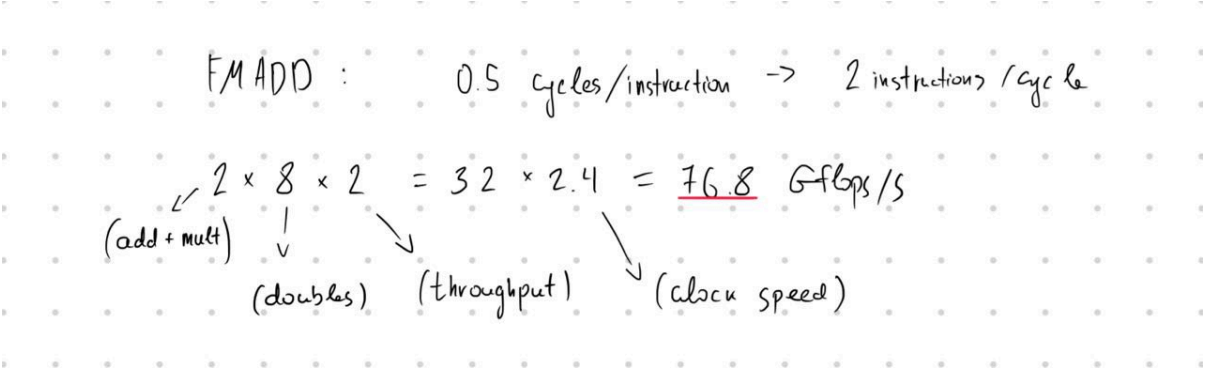


Every implementation gained a sizable speedup. More importantly, now the Blocking introduces huge improvements. Previously, implementing Blocking on Transposed matrices only slowed it down, but now it drastically cuts down execution time from 9 seconds to 1.4 seconds.

Our best implementation now approaches the performance of OpenBLAS - a dedicated native library. Albeit, twice as slow.



However, we began to wonder how good OpenBLAS really is. Are there more improvements to be made? We made a rough estimation of the theoretical maximum performance achievable on our test machine.



This is an approximation; in real-world scenarios, numerous factors affect the performance of the machine. For starters, the test machine utilizes dynamic clock speed depending on processor load, heat, and other factors.

Implementation	Time (ms)	Performance (GFLOPS/s)
NumPy	5011.99	3.43
Normal	84,415.80	0.20
Transposed	9,129.56	1.88
Blocked	1,514.02	11.35
Unrolled All	4,143.10	4.15
Unrolled Innermost	9,023.19	1.90
Unrolled Outermost	1008.35	17.04
Unrolled I + Blocked	774.97	22.17
Unrolled I,K + Blocked	1,343.03	12.79
Unrolled I,K + Blocked + SIMD	552.81	31.08
OpenBLAS	286.17	60.03

From our testing, we can see OpenBLAS is approaching the limit of our test machine. The possible improvements utilized to reach such high performance may include memory alignment - In our implementation we are using `loadu_pd` which allows us to load values into unaligned memory address. It is convenient but noticeably slower than its aligned counterpart `load_pd`. Additionally, we believe OpenBLAS is using more flexible block sizes/shapes. While our implementation only works with square block sizes e.g., 256 by 256, they could use rectangle block - possibly using the cache line size to the depth of the cache(e.g., 64 by 128). Which will make use of the cache much better.

Task 3

The multi-threading process used OpenMP API. We collapsed the all three for loops into one large loop which then will be divided into the available 4 thread.

```
#pragma omp parallel for collapse(3)
for (int i0 = 0; i0 < R1; i0 += BLOCK_SIZE) {
  for (int j0 = 0; j0 < C2; j0 += BLOCK_SIZE) {
    for (int k0 = 0; k0 < C1; k0 += BLOCK_SIZE) {
```

Our implementation achieved a little over double the speed of the single-threaded version. While, OpenBLAS performed little over triple its performance. Additionally, we tried collapsing only two loops which resulted noticeable degradation in performance.

Multi-threaded Implementation	Time (ms)	Performance (GFLOPS/s)
Unrolled I,K + Blocked + SIMD	223.854	76.746
OpenBLAS(Collapse = 3)	90.269	190.368
OpenBLAS(Collapse = 2)	107.078	160.443

Written and tested by:

- Marios Gkikopouli
- Batjigdrel Bataa