Task 1

```
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

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++;
    }
  }
}
```

Blocking/Tiling

```
void mulMatBlocked(const double *mat1, const double *mat2T, double *result) {
  const int BLOCK_SIZE = 64; // 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 = result[i * C2 + j];
```

```
            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;
          }
        }
      }
    }
  }
}
```

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

SIMD instructions -> `AVX512`

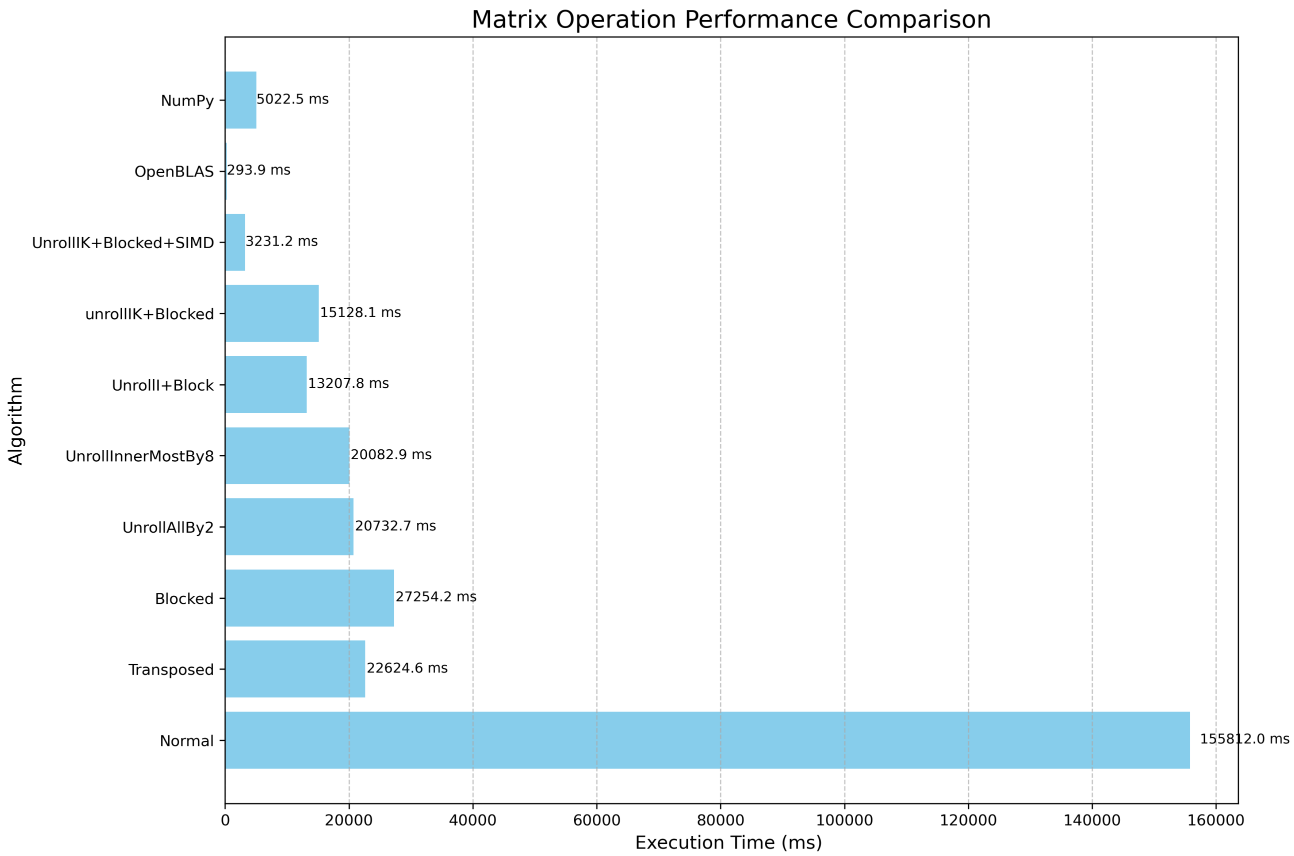
```
void mulMatWithUnrolledBlockedI(const double *mat1, const double *mat2T, double *result) {
  const int BLOCK_SIZE = 128;
  const int UNROLL = 8; // 8 * double = 512

  memset(result, 0, sizeof(double) * R1 * C2);

  // 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
        int iLimit = std::min(i0 + BLOCK_SIZE, R1 - UNROLL + 1);
```

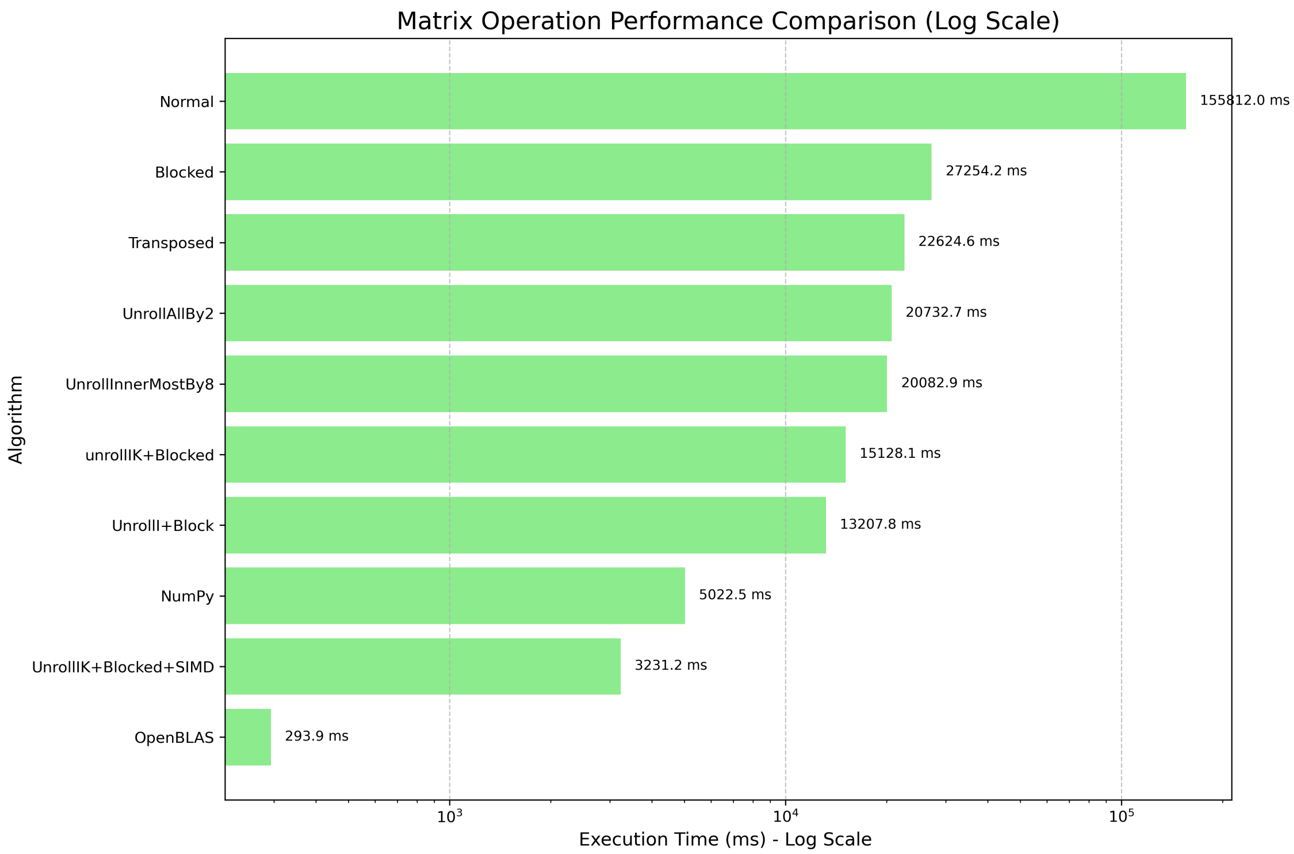
```
      int jLimit = std::min(j0 + BLOCK_SIZE, C2);
      int kLimit = std::min(k0 + BLOCK_SIZE, C1 - UNROLL + 1);

      for (int i = i0; i < iLimit; i += UNROLL) {
        for (int j = j0; j < jLimit; 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 < kLimit; 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);

            // Handle remaining k values that couldn't be vectorized
            int kLimit = ROUND_DOWN(std::min(k0 + BLOCK_SIZE, C1), UNROLL);
            for (int k = kLimit; k < std::min(k0 + BLOCK_SIZE, C1); k++) {
              double m2_val = mat2T[j * C1 + k];

              hsum0 += mat1[i * C1 + k] * m2_val;
              hsum1 += mat1[(i + 1) * C1 + k] * m2_val;
              hsum2 += mat1[(i + 2) * C1 + k] * m2_val;
              hsum3 += mat1[(i + 3) * C1 + k] * m2_val;
              hsum4 += mat1[(i + 4) * C1 + k] * m2_val;
              hsum5 += mat1[(i + 5) * C1 + k] * m2_val;
              hsum6 += mat1[(i + 6) * C1 + k] * m2_val;
              hsum7 += mat1[(i + 7) * C1 + k] * m2_val;
            }

            // Add to existing result values
            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;
          }
        }

        // Handle remaining rows that couldn't be unrolled
        for (int i = ROUND_DOWN(std::min(i0 + BLOCK_SIZE, R1), UNROLL); i < std::min(i0 +
BLOCK_SIZE, R1); i++) {
          for (int j = j0; j < std::min(j0 + BLOCK_SIZE, C2); j++) {
            // Use scalar operations for remaining rows
            double sum = result[i * C2 + j];

            // Try to vectorize k loops even for remaining rows
            for (int k = k0; k < std::min(k0 + BLOCK_SIZE, C1 - UNROLL + 1); k += UNROLL)
{
              __m512d m1 = _mm512_loadu_pd(&mat1[i * C1 + k]);
              __m512d m2 = _mm512_loadu_pd(&mat2T[j * C1 + k]);
              __m512d prod = _mm512_mul_pd(m1, m2);
              sum += _mm512_reduce_add_pd(prod);
            }

            // Handle remaining k values
            int k_limit = ROUND_DOWN(std::min(k0 + BLOCK_SIZE, C1), UNROLL);
            for (int k = k_limit; k < std::min(k0 + BLOCK_SIZE, C1); k++) {
              sum += mat1[i * C1 + k] * mat2T[j * C1 + k];
            }
```

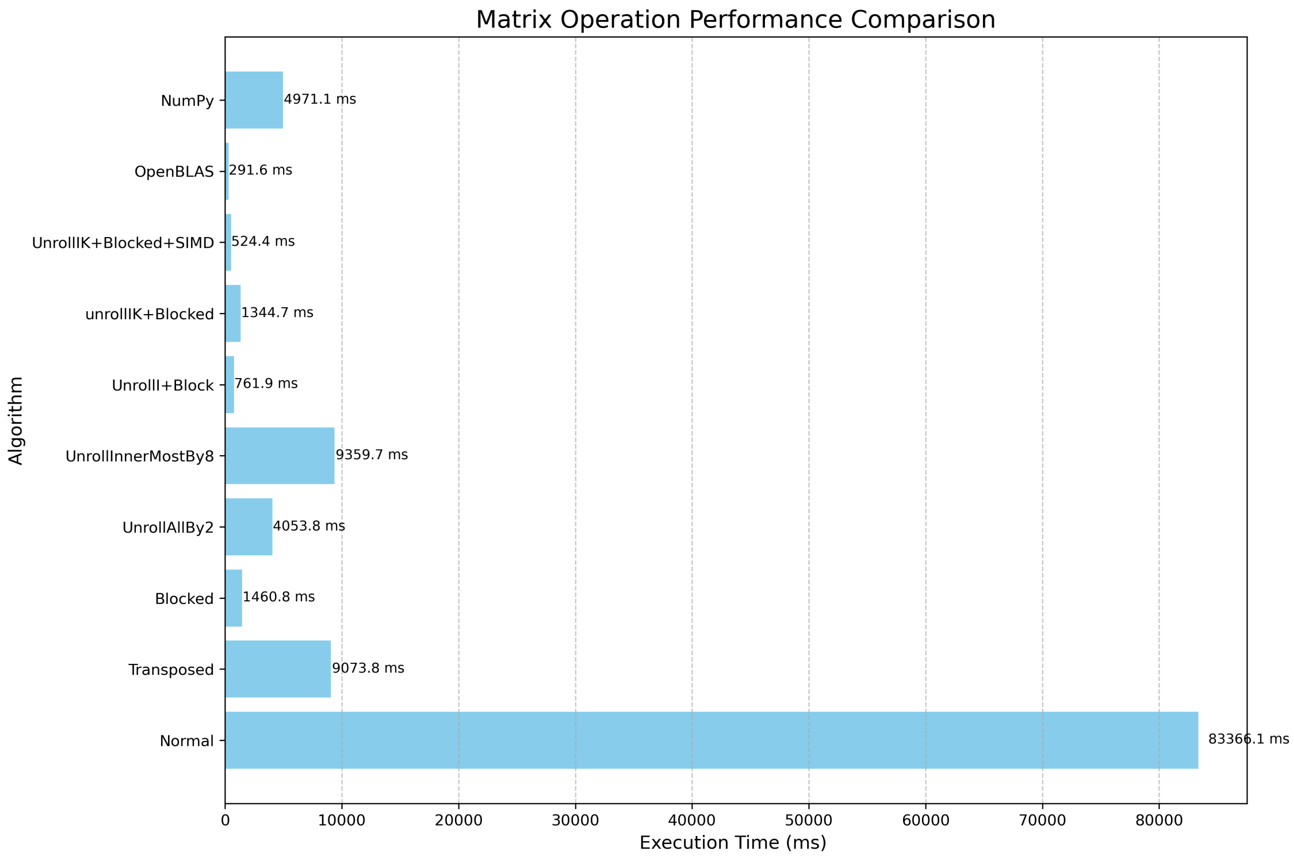
```
            result[i * C2 + j] = sum;
          }
        }
      }
    }
  }
}
```

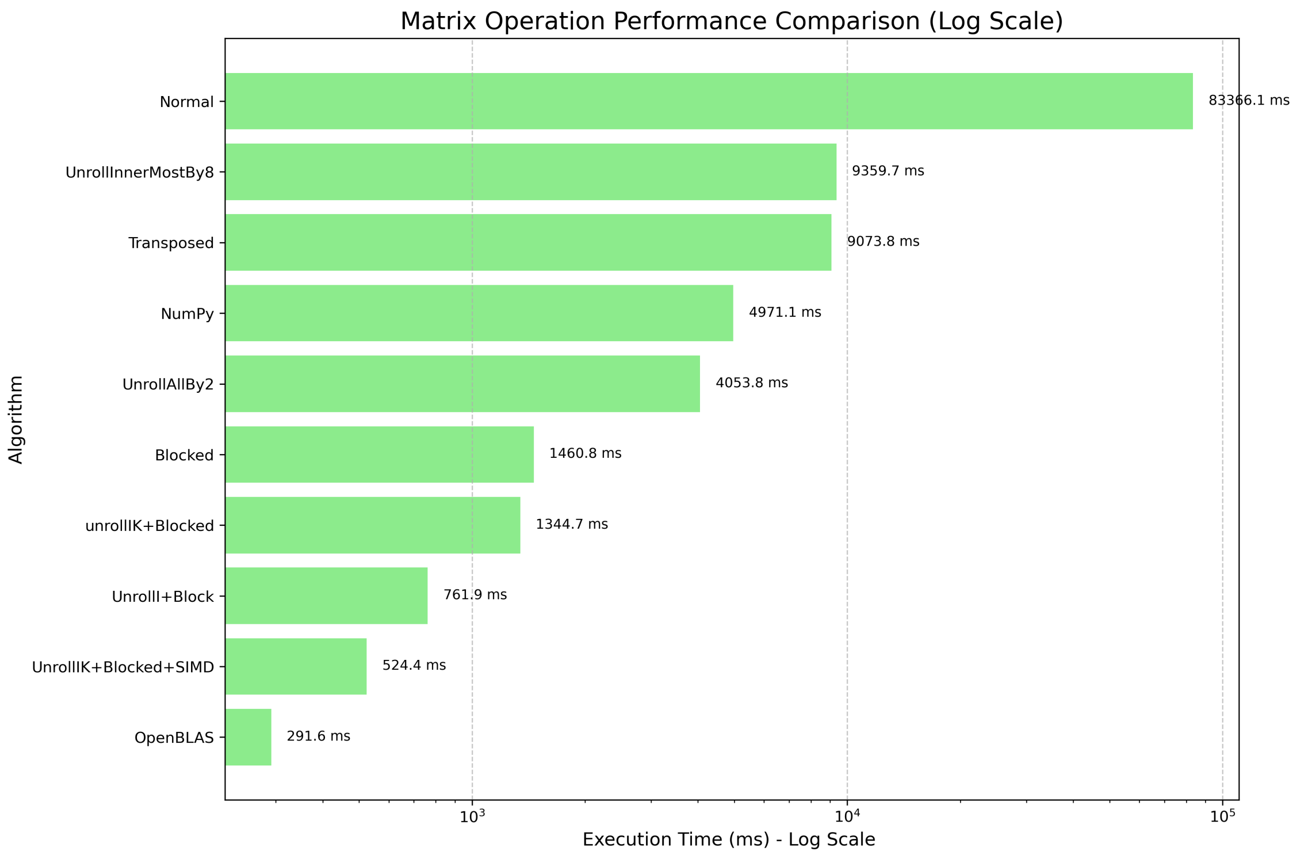
Matrix Multiplication Performance Comparison





Enable → `-Ofast`





Implementation	Time (ms)	Performance (GFLOPS/s)
Numpy	5011.99	3.43
Normal	84,415.80	0.20
Transposed	9,129.56	1.88

Implementation	Time (ms)	Performance (GFLOPS/s)
Blocked	1,514.02	11.35
Unrolled All	4,143.10	4.15
Unrolled Innermost	9,023.19	1.90
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

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

Rough Estimation of the theoretical maximum performance:

FMADD => 0.5 cycles / instruction => 2 inst / cycle

$$\underbrace{2}_{\text{(add, mult)}} * \underbrace{8}_{\text{(doubles)}} * \underbrace{2}_{\text{(TP)}} = 32 * \underbrace{2.4}_{\text{clock speed}} = 76.8 \text{ Gflops}$$

Task 3

Implementation	Time (ms)	Performance (GFLOPS/s)
Unrolled I,K + Blocked + SIMD	223.854	76.746
OpenBLAS	90.2454	190.368

```
#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) {
```