Matrix Multiplication:

Native code (Not optimized):

This code performs matrix multiplication of two random matrices of size N x N, where N is defined as 1300 using three nested loops.

First, it initializes a random number generator using the current time, and then allocates memory for three matrices A, B, and C of size N x N using the malloc function.

Next, it fills matrices A and B with random numbers between 0 and 1 using the rand function, and then performs matrix multiplication of matrices A and B using a naive matrix multiplication algorithm with three nested loops. The result of the matrix multiplication is stored in matrix C.

The execution time of the matrix multiplication is measured using the clock function before and after the multiplication loop, and the elapsed time is printed to the console. Finally, the allocated memory is freed using the free function, and the program exits with a return value of 0.

native.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#define N 1300

int main()
{
    // Initialize random number generator
    srand(time(NULL));

    // Allocate memory for matrices A, B, and C
    double *A = (double *)malloc(N * N *
sizeof(double));
    double *B = (double *)malloc(N * N *
sizeof(double));
    double *C = (double *)malloc(N * N *
sizeof(double));

    // Initialize matrices A and B with random numbers
    for (int i = 0; i < N * N; i++)
    {
        A[i] = (double)rand() / RAND_MAX;
        B[i] = (double)rand() / RAND_MAX;
    }

    // Multiply matrices A and B and store the result
in matrix C
    printf("Multiplying...\n");
    clock_t start_time = clock();
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            double sum = 0;
            for (int k = 0; k < N; k++)
            {
                sum += A[i * N + k] * B[k * N + j];
            }
            C[i * N + j] = sum;
        }
    }
    clock_t end_time = clock();
    printf("Done\n");

    // Print execution time
    double execution_time = ((double)(end_time - 
start_time)) / CLOCKS_PER_SEC;
    printf("Execution time(Not-Optimized): %f 
seconds\n", execution_time);

    // Free memory
    free(A);
    free(B);
    free(C);

    return 0;
}
```

<u>Optimized code (optimized)</u>:

This code performs matrix multiplication of two matrices A and B of size N x N and stores the result in matrix C using AVX2 intrinsics for optimization.

The first part of the code is the same as the non-optimized version, where memory is allocated for matrices A, B, and C, and they are initialized with random numbers.

In the optimized version, the innermost loop for matrix multiplication is replaced with AVX2 intrinsics to improve performance.

Specifically, the loop uses the _mm256_loadu_pd function to load four double precision floating point numbers from the ith row of matrix A and the kth column of matrix B into 256-bit AVX registers a and b, respectively.

The _mm256_mul_pd function is then used to multiply the values in a and b and store the result in another AVX register c.

The _mm256_add_pd function is used to add the four values in c to an accumulator AVX register c_sum initialized to zero using _mm256_setzero_pd.

The loop increments k by 4 since four floating point numbers can be loaded into an AVX register at a time.

After the loop completes for a given (i, j) pair, the four values in c_sum are added together and stored in the corresponding entry of matrix C.

The optimized version prints out the execution time using the clock() function and CLOCKS_PER_SEC constant.

Finally, memory for matrices A, B, and C is freed and the program returns 0.

optimized.c

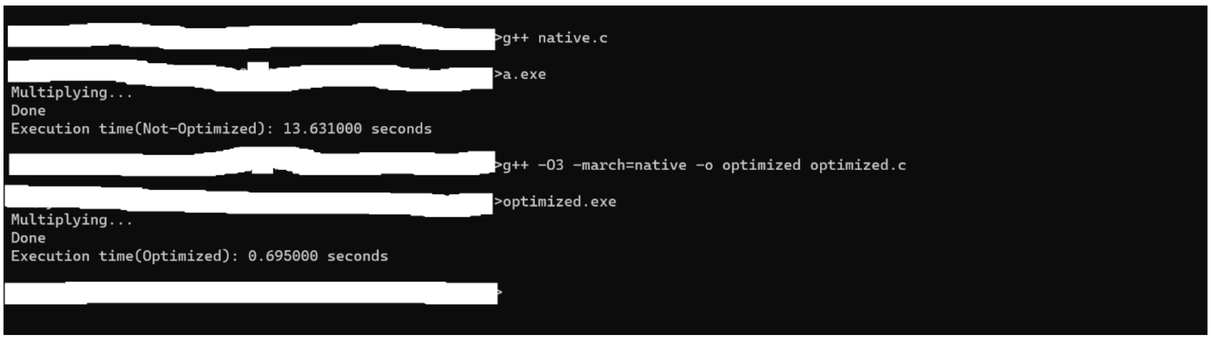
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <immintrin.h>

#define N 1300

int main()
{
    // Initialize random number generator
    srand(time(NULL));

    // Allocate memory for matrices A, B, and C
    double *A = (double *)malloc(N * N *
sizeof(double));
    double *B = (double *)malloc(N * N *
sizeof(double));
    double *C = (double *)malloc(N * N *
sizeof(double));

    // Initialize matrices A and B with random numbers
    for (int i = 0; i < N * N; i++)
    {
        A[i] = (double)rand() / RAND_MAX;
        B[i] = (double)rand() / RAND_MAX;
    }

    // Multiply matrices A and B and store the result
in matrix C
```

```
    printf("Multiplying...\n");
    clock_t start_time = clock();

    __m256d a, b, c, c_sum;

    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            c_sum = _mm256_setzero_pd();
            for (int k = 0; k < N; k += 4)
            {
                a = _mm256_loadu_pd(&A[i * N + k]);
                b = _mm256_loadu_pd(&B[k * N + j]);
                c = _mm256_mul_pd(a, b);
                c_sum = _mm256_add_pd(c_sum, c);
            }
            C[i * N + j] = c_sum[0] + c_sum[1] +
c_sum[2] + c_sum[3];
        }
    }

    clock_t end_time = clock();
    printf("Done\n");

    // Print execution time
    double execution_time = ((double)(end_time -
start_time)) / CLOCKS_PER_SEC;
    printf("Execution time(Optimized): %f seconds\n",
execution_time);

    // Free memory
    free(A);
    free(B);
```

```c
    free(C);

    return 0;
}
```

Output:

```
>g++ native.c

>a.exe

Multiplying...

Done

Execution time(Not-Optimized): 13.631000 seconds


>g++ -O3 -march=native -o optimized optimized.c

>optimized.exe

Multiplying...

Done

Execution time(Optimized): 0.695000 seconds
```



```
[redacted]>g++ native.c

[redacted]>a.exe
Multiplying...
Done
Execution time(Not-Optimized): 13.631000 seconds

[redacted]>g++ -O3 -march=native -o optimized optimized.c

[redacted]>optimized.exe
Multiplying...
Done
Execution time(Optimized): 0.695000 seconds

[redacted]>
```

In summary, we can say that optimizing the matrix multiplication code using SIMD instructions can significantly improve the performance. In this case, by using AVX2 intrinsic, the execution time reduced from 13 seconds to 0.6 seconds, which is over 20 times faster. The AVX2 intrinsic uses vectorization to perform the matrix multiplication on multiple elements at once, thus reducing the number of instructions required to perform the operation. This results in a much faster execution time.

References:

- https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics.html
- http://sci.tuomastonteri.fi/programming/sse