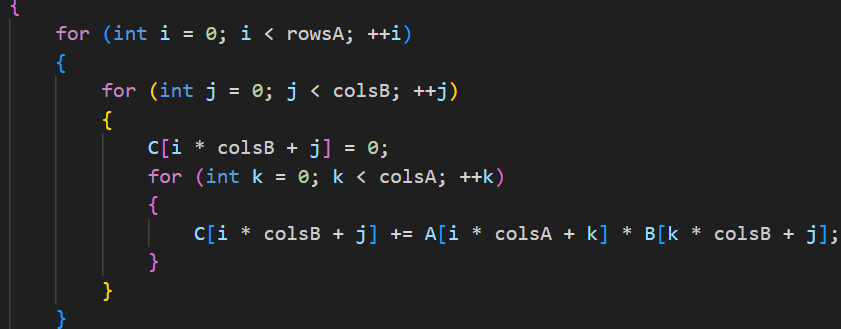
CUDA PROGRAMMING PROCESS :

INTRODUCTION :

This project focuses on comparing the performance of matrix multiplication on the CPU and GPU, using different approaches: Thrust for both host (CPU) and device (GPU) vector operations, and cuBLAS, an optimized GPU library from NVIDIA designed specifically for linear algebra operations. The main objectives of this project are twofold: to implement matrix multiplication using Thrust’s functors, ensuring compatibility with both CPU and GPU operations, and to compare the execution times of CPU-based and GPU-based matrix multiplications. Additionally, the project aims to explore the performance of cuBLAS for matrix multiplication on the GPU and compare it against Thrust’s GPU performance.

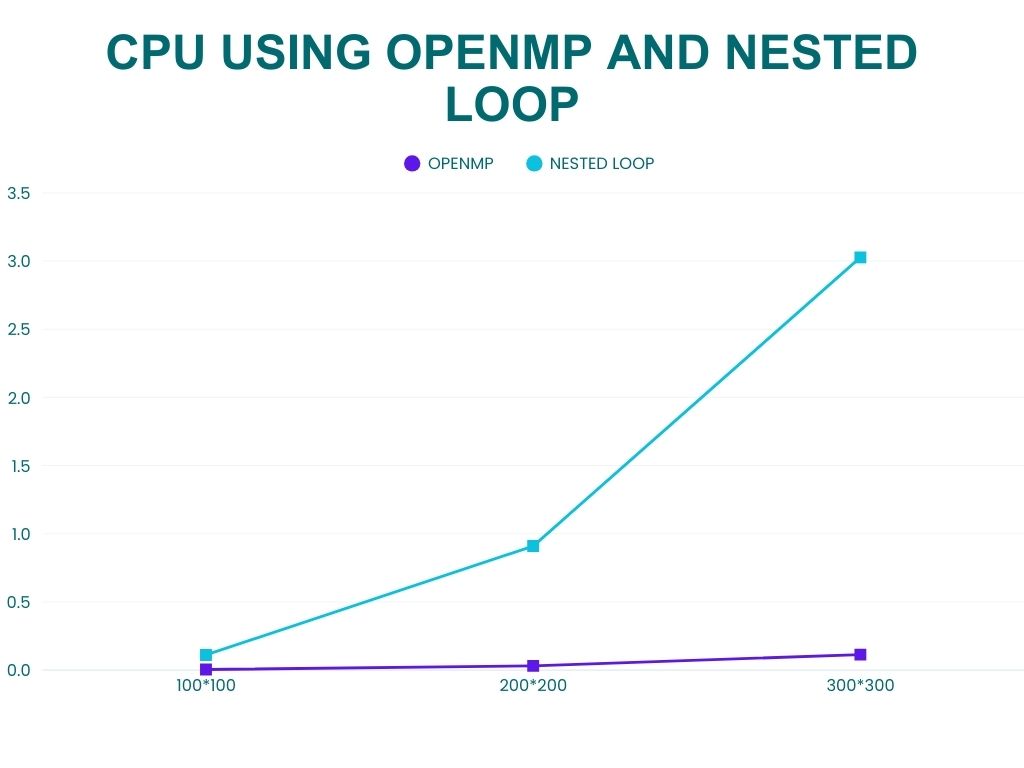
MY EXPERIENCES:

Initially, I implemented CPU-based matrix multiplication using the row-by-column method with nested for loops. I utilized thrust::host\_vector, which runs on the CPU, to execute the matrix multiplication algorithm. This took 33 minutes to compute for 2500x2500 matrices.

**Using a Friend’s Advice (CPU Parallel Execution):**

Using a friend’s advice, I implemented CPU parallel execution for matrix multiplication. This approach aimed to optimize the performance of the row-by-column method by utilizing parallelism on the CPU. As I searched this method, ChatGPT recommended using the OpenMP library for CPU parallel execution. OpenMP is a popular tool for parallel programming in C, C++, allowing for easy parallelization of loops and tasks to speed up the matrix multiplication process on the CPU.  
By using the OpenMP (Open Multi-Processing) library, the execution time for a 2500 × 2500 matrix was reduced to just 3 minutes, compared to 33 minutes with my initial implementation.

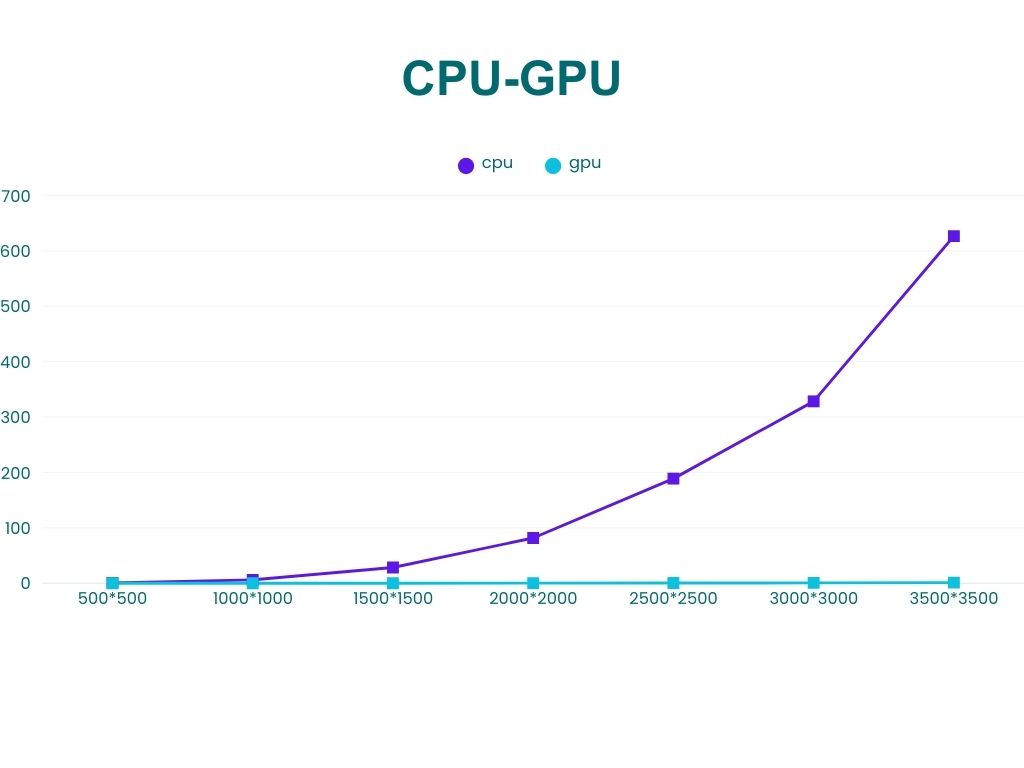
In the chart below, you can see the immense difference between CPU execution using a nested loop and CPU execution using the OpenMP library.



Problems with My First Code:

1. Using nested loops for CPU execution.
2. Using three different functors separately for CPU, GPU (Thrust-based), and GPU (cuBLAS).
3. Unexpectedly, cuBLAS performed slower than Thrust-based GPU implementations, which was incorrect since cuBLAS is expected to be faster for large-sized matrices (more than 1000\*1000).

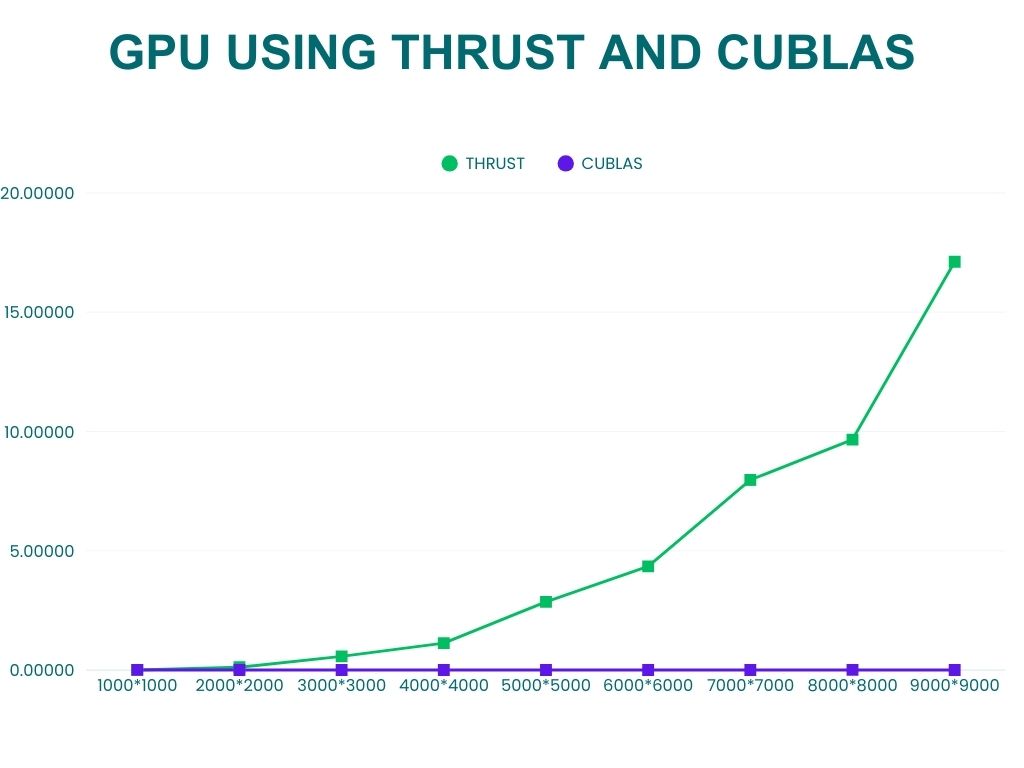
To fix this, I removed the error detection function and simplified the cuBLAS function code, leading to results that were finally as expected—with cuBLAS outperforming the other implementations.

In the second graph, you can see the difference in execution time between the CPU and GPU. For small-sized matrices, the CPU performs faster. However, as the matrix size increases, the gap between the lines becomes significantly larger. This indicates that the GPU outperforms the CPU for matrices of size 1000 × 1000 and larger.  


Overall, whether using nested loops or the OpenMP library, CPU execution time is significantly higher than GPU execution time.

Another approach I used was the cuBLAS library. Based on my search, cuBLAS includes a dedicated function for matrix multiplication (part of the Basic Linear Algebra Subprograms (BLAS) group), which is expected to be faster than GPU execution using Thrust.

By analyzing the graph, I was able to visualize this significant difference. However, based on my experience, for matrices smaller than 1000 × 1000, GPU execution using Thrust performed better and faster than cuBLAS. But as the matrix size increased, I finally obtained the expected results—cuBLAS outperformed Thrust, as anticipated.



In general, using different types of code, we ultimately get similar results. Whether programming the CPU using Thrust with nested loops or utilizing the OpenMP library for CPU parallelization, the CPU functionality tends to be slower than the GPU, whether using the Thrust library or not. The primary goal of CUDA programming is to leverage the GPU to achieve faster results. This confirms that the GPU is indeed much more efficient for large-scale computations. The new insight I gained was regarding the functionality of cuBLAS and its purpose for handling large-sized matrices.