Università di Genova

Computer Engineerig

Project Exam

High Performance Computing

Martin Martuccio and Lorenzo Mesi

September, 2024

Contents

1 Introduction **4**

2 Methodology **4**

2.1 Computational Environment 4

2.2 GPU Specifications 5

2.3 Tools and Technologies 5

2.3.1 OpenMP 5

2.3.2 CUDA 6

3 Matrix multiplication algorithm **6**

3.1 Hotspot identification 8

4 Basic algorithm **9**

5 First Optimization **10**

5.1 Implementation 10

5.2 Performance Analysis 11

6 OpenMP implementation **13**

6.1 Basic implementation 13

6.2 SIMD implementation 13

6.3 Loop tiling implementation 14

6.4 Final implementation 15

6.5 Performance Analysis 16

6.6 Results explanation 17

7 OpenMP threads **17**

7.1 Performance Analysis 18

8 The Speedup **20**

9 Efficiency respect to the number of threads **21**

10 CUDA implementation **22**

10.1 Introduction to Cuda 22

10.2 Cuda Memory Spaces 23

10.3 Implementation 24

10.3.1 **Global Memory Implementation:** 24
10.3.2 **Pitched Memory Implementation:** 25
10.3.3 **Dimensionality Constraints of CUDA Implementations** 26
10.3.4 **Shared Memory Implementation** 27
10.4 **Code Correctness Verification** 31
10.5 Performance Analysis 31
10.5.1 Preliminary Tests (*Completeness and curiosity*) 32
10.5.2 Choosing tile size 36
10.6 The Speedup 38
10.7 The Efficiency respect Cuda cores 39

11 Comparison and Results **39**

1 Introduction

Matrix multiplication is a key operation in various scientific computations. Improving its performance can significantly impact the efficiency of applications ranging from machine learning to physics simulations. This report discusses methods to accelerate matrix multiplication using OpenMP and CUDA for NVIDIA GPUs.

2 Methodology

We implement the matrix multiplication in C++, enhanced with OpenMP and CUDA to utilize parallel processing capabilities of modern computing architectures. The following subsections describe the computational environment, tools, and the matrix multiplication algorithm.

2.1 Computational Environment

The laboratory is equipped with high-end computing systems configured for performance-intensive computational tasks. The main specifications are as follows:

- **Processor:** 12th Gen Intel(R) Core(TM) i9-12900K
- **CPU Specs:**
  - **Address sizes:** 46 bits physical, 48 bits virtual
  - **CPU(s):** 24
  - **Core(s) per socket:** 16
  - **Socket(s):** 1
  - **Base Frequency:** 3.2 GHz
  - **Max Turbo Frequency:** 5.2 GHz
- **Memory:**
  - **Architecture:** x86_64
  - **Byte Order:** Little Endian
  - **Virtualization:** Intel VT-x enabled

  - **Cache:** L1d - 384 KiB, L1i - 256 KiB, L2 - 10 MiB

- **Other CPU features:** Hyper-Threading, Turbo Boost, Virtualization (VT-x), instruction set AVX2 (Advanced Vector Extension).

2.2 GPU Specifications

The laboratory utilizes NVIDIA GPUs to facilitate GPU-accelerated computing tasks:

- **Model:** NVIDIA T400
- **CUDA Cores:** 384
- **Memory:** 2 GB GDDR6
- **Interface:** 64-bit
- **Bandwidth:** Up to 80 GB/s

2.3 Tools and Technologies

For the matrix multiplication project, two main parallel computing technologies were employed to optimize performance: OpenMP and CUDA. Each technology leverages different aspects of the computing hardware to maximize efficiency.

2.3.1 OpenMP

OpenMP (Open Multi-Processing) is utilized for optimizing performance on CPU-level operations. Key benefits of using OpenMP in this project include:

- **Hyper-Threading:** By exploiting hyper-threading technology of the Intel Core i9-12900K, OpenMP can execute multiple threads on a single core simultaneously, effectively doubling the computation capacity of each core.

- **Vectorization:** allows for the simultaneous execution of operations on multiple data elements within a single CPU instruction, fully leveraging the capabilities of vector registers and SIMD (Single Instruction, Multiple Data) instructions. In our case, OpenMP enables automatic vectorization of the code, which allows the CPU to process multiple data points with a single instruction (using AVX2 registers), thus accelerating the arithmetic computations significantly.

- **Parallelization:** is the technique of dividing a computational task into subtasks that can be performed simultaneously on several computing units. OpenMP directives were added to the matrix multiplication loops to parallelize operations that can be performed independently, maximizing the utilization of available CPU cores and threads.

2.3.2 CUDA

CUDA (Compute Unified Device Architecture) by NVIDIA was chosen to exploit the power of GPU computing. The key aspects of CUDA in this project are:

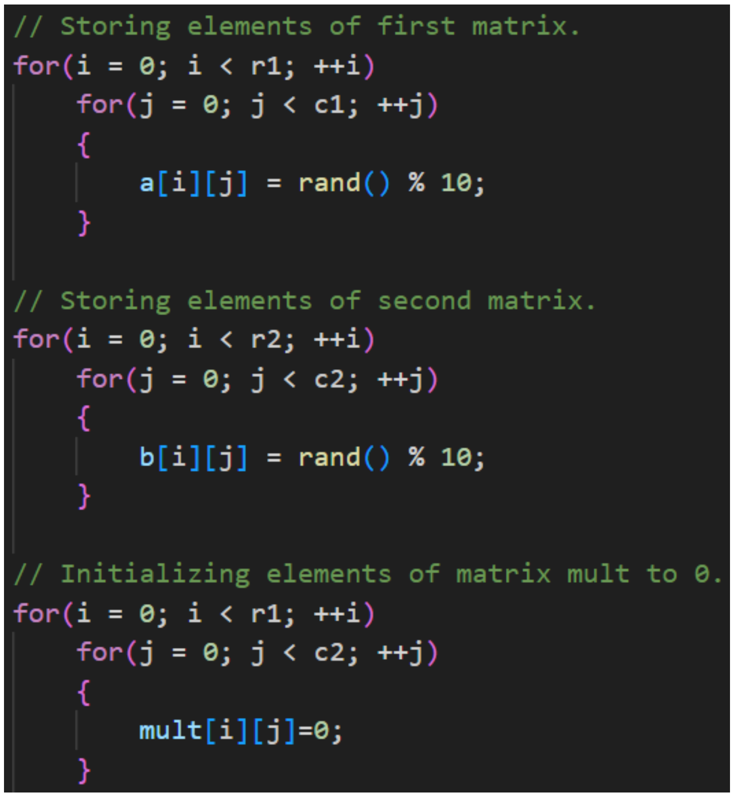
- **Co-processor Utilization:** The NVIDIA T400 GPU acts as a co-processor, taking over intensive computation tasks from the CPU to speed up the overall processing time.
- **Parallel Threads:** CUDA excels in managing thousands of threads simultaneously, making it ideal for executing multiple instances of matrix multiplication in parallel across its 384 CUDA cores, significantly reducing computation time.

CUDA's parallel computing capabilities were exploited to perform matrix multiplication, where each element of the resultant matrix is computed concurrently, demonstrating substantial reductions in processing time compared to serial implementations.

3 Matrix multiplication algorithm

Matrix Initialization

The matrices `a`, `b`, and `mult` are initialized using nested loops. Matrices `a` and `b` are filled with random integers between 0 and 9, while `mult` is initialized to zero, preparing it for storing the result of the multiplication.

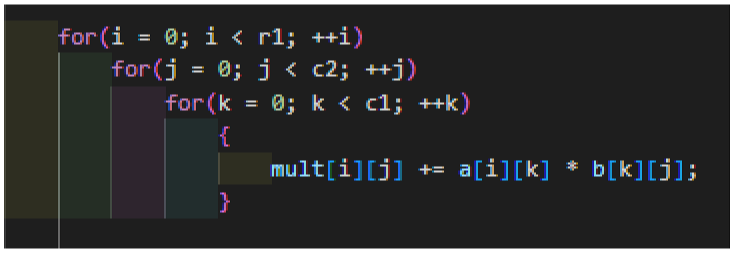


```
// Storing elements of first matrix.
for(i = 0; i < r1; ++i)
    for(j = 0; j < c1; ++j)
    {
        a[i][j] = rand() % 10;
    }

// Storing elements of second matrix.
for(i = 0; i < r2; ++i)
    for(j = 0; j < c2; ++j)
    {
        b[i][j] = rand() % 10;
    }

// Initializing elements of matrix mult to 0.
for(i = 0; i < r1; ++i)
    for(j = 0; j < c2; ++j)
    {
        mult[i][j]=0;
    }
```

Figure 1: Matrix initialization

Matrix Multiplication Process

The core of the program is the matrix multiplication loop, where each element of the resultant matrix `mult` is computed as the dot product of rows of `a` and columns of `b`.



```
for(i = 0; i < r1; ++i)
    for(j = 0; j < c2; ++j)
        for(k = 0; k < c1; ++k)
        {
            mult[i][j] += a[i][k] * b[k][j];
        }
```

Figure 2: Matrix Multiplication

Explanation

This code section demonstrates fundamental operations involved in matrix manipulation, initialization, computation.

3.1 Hotspot identification

In the matrix multiplication program, the hotspot where the most compute intensive operations occur is located in the nested loop structure used for the multiplication of the two matrices. This section of code is important as it directly determines the efficiency and performance of the entire computation.

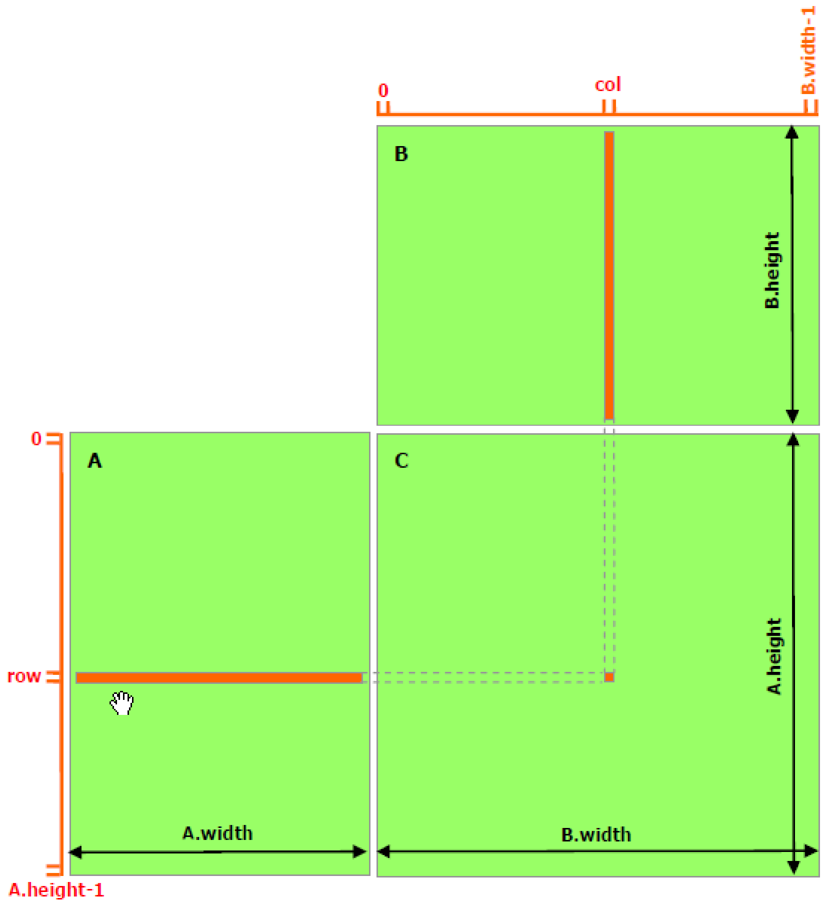


Figure 3: Matrix Multiplication

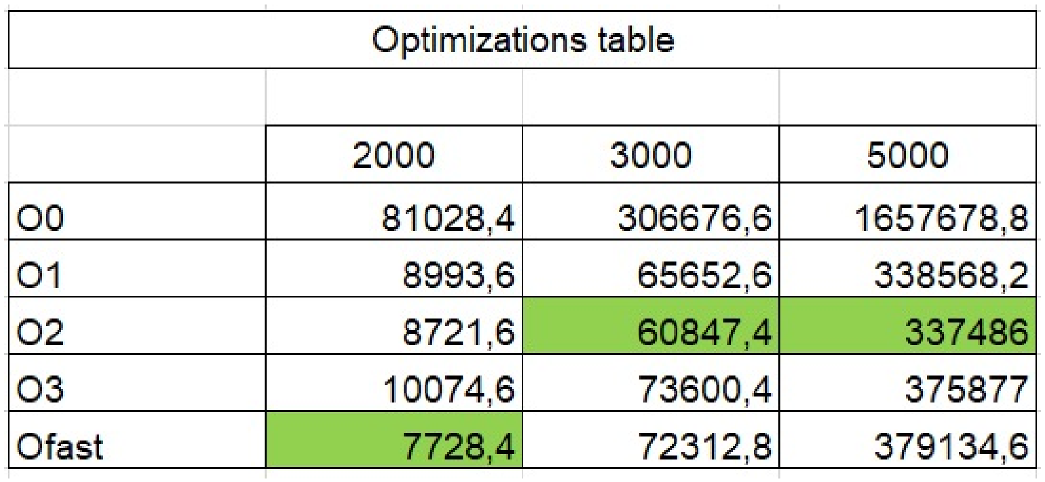
Explanation of the Hotspot: The reason this part of the code (see Figure : 2) is identified as the hotspot is due to its time complexity of $O(N^3)$, where

N represents the dimension of the square matrices involved. The complexity arises because the algorithm must execute the innermost statement a multiplication followed by an addition—for every combination of indices i, j, and k, which scales cubically with the size of the matrices.

This **cubic complexity** is characteristic of the standard matrix multiplication algorithm and is a **primary target for optimization** in high-performance computing.

4 Basic algorithm

As first step of our project we decided to run the sequential algorithm analysing the execution times of that version using different optimization levels. To select the best optimization level to use to compile the next different versions of the program, we run 15 times the basic version of the program for each optimization level and calculate the average time. Subsequently we obtained the optimization level of our interest, choosing the lowest average execution time.



Optimizations table

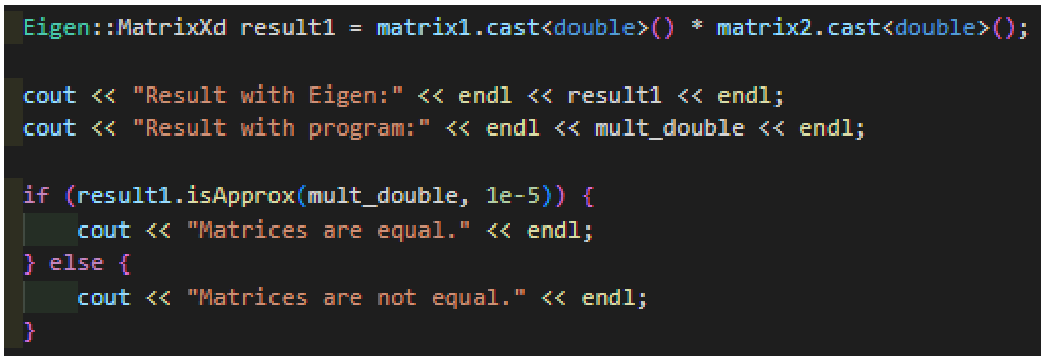
	2000	3000	5000
O0	81028,4	306676,6	1657678,8
O1	8993,6	65652,6	338568,2
O2	8721,6	60847,4	337486
O3	10074,6	73600,4	375877
Ofast	7728,4	72312,8	379134,6

Figure 4: Optimizations table (ms)

According with the results obtained, we decided to use the O2 level since in the executions with 3000x3000 and 5000x5000 matrices have taken a lower execution time with respect to the others optimization levels.

Result Verification using Eigen Library

To verify the correctness of the results obtained on all the different implementation (both with OpenMP and Cuda), we compare them with results computed using the Eigen library. This ensures that all the operation done on the matrices don't return wrong results.



```
Eigen::MatrixXd result1 = matrix1.cast<double>() * matrix2.cast<double>();

cout << "Result with Eigen:" << endl << result1 << endl;
cout << "Result with program:" << endl << mult_double << endl;

if (result1.isApprox(mult_double, 1e-5)) {
    cout << "Matrices are equal." << endl;
} else {
    cout << "Matrices are not equal." << endl;
}
```

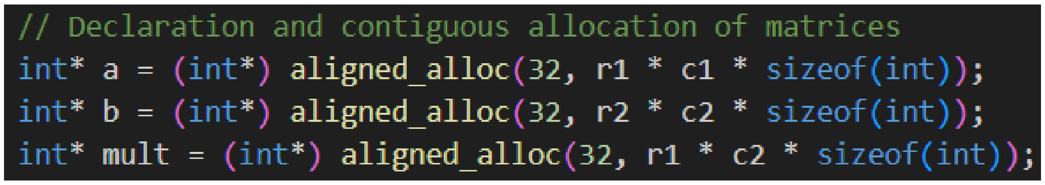
Figure 5: Result Comparison using Eigen Library

5 First Optimization

5.1 Implementation

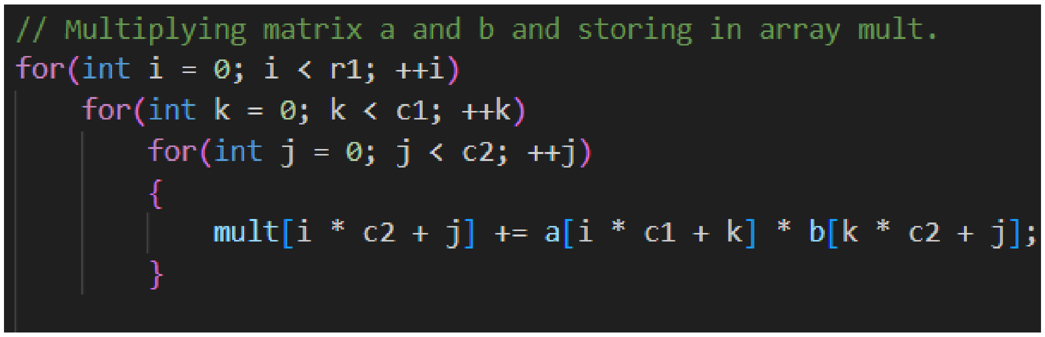
As first optimization we decided to modify the code in order to exploit better the hardware of the computer.

In fact we decided to apply some techniques that allow to perform operations on data in a faster way, like loop swapping and contiguous allocation of vectors in memory in order to improve the cache efficiency and vector operations. In particular the loop swapping allows to access all the three matrices (a, b and mult) by rows which exploits the spatial locality of data; while the contiguous allocation is useful to reduce cache miss.



```
// Declaration and contiguous allocation of matrices
int* a = (int*) aligned_alloc(32, r1 * c1 * sizeof(int));
int* b = (int*) aligned_alloc(32, r2 * c2 * sizeof(int));
int* mult = (int*) aligned_alloc(32, r1 * c2 * sizeof(int));
```

Figure 6: Contiguous allocation



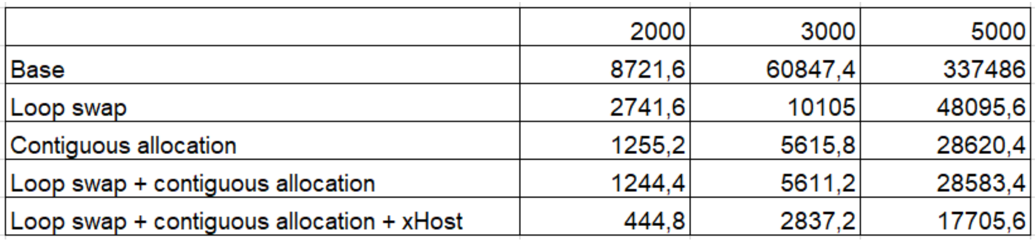
```
// Multiplying matrix a and b and storing in array mult.
for(int i = 0; i < r1; ++i)
    for(int k = 0; k < c1; ++k)
        for(int j = 0; j < c2; ++j)
        {
            mult[i * c2 + j] += a[i * c1 + k] * b[k * c2 + j];
        }
```

Figure 7: Loop swapping

Then we compiled the code using the *-xHost* flag which tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.

5.2 Performance Analysis

With these simple implementations we obtained satisfactory results in terms of reduction of the execution time.



	2000	3000	5000
Base	8721,6	60847,4	337486
Loop swap	2741,6	10105	48095,6
Contiguous allocation	1255,2	5615,8	28620,4
Loop swap + contiguous allocation	1244,4	5611,2	28583,4
Loop swap + contiguous allocation + xHost	444,8	2837,2	17705,6

Figure 8: Execution times (ms) with basic implementation

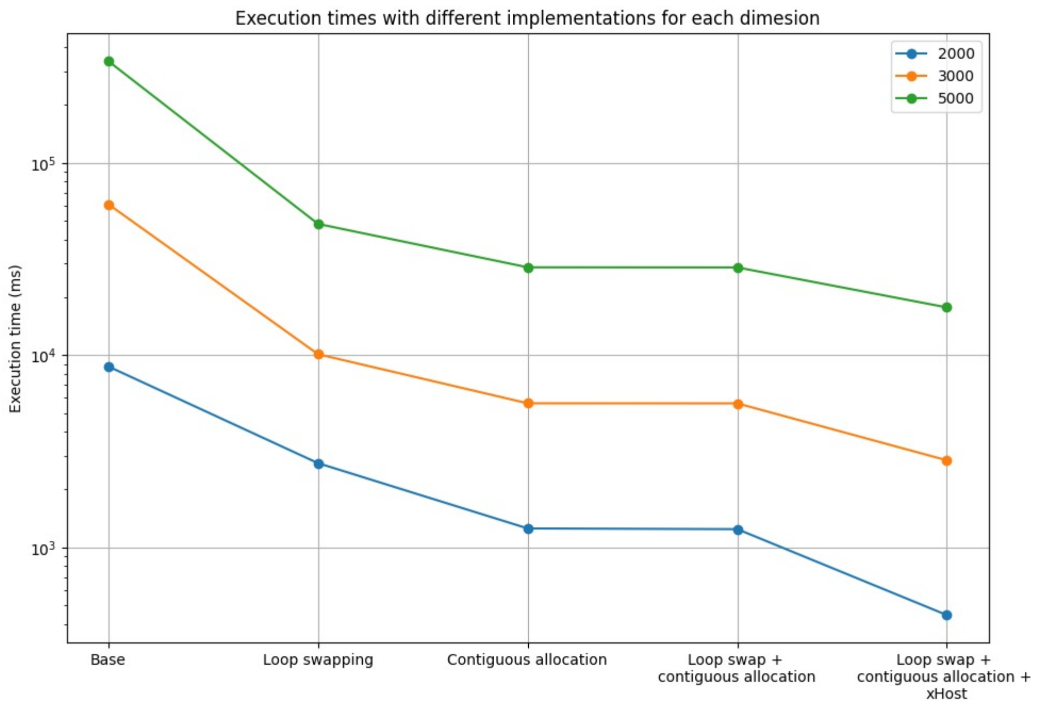
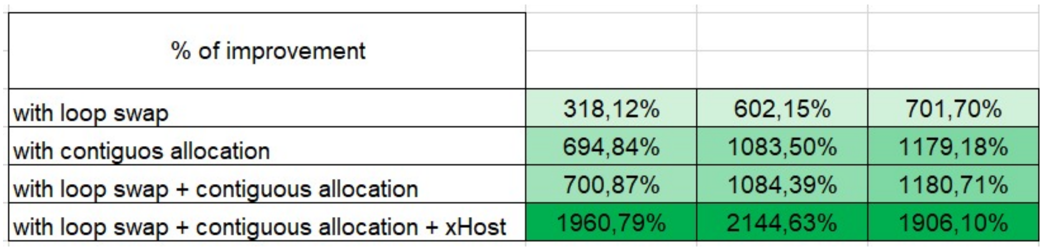


Figure 9: Basic implementations graph

We also computed the percentages of improvements with respect to the execution time required for running the basic version of the program.



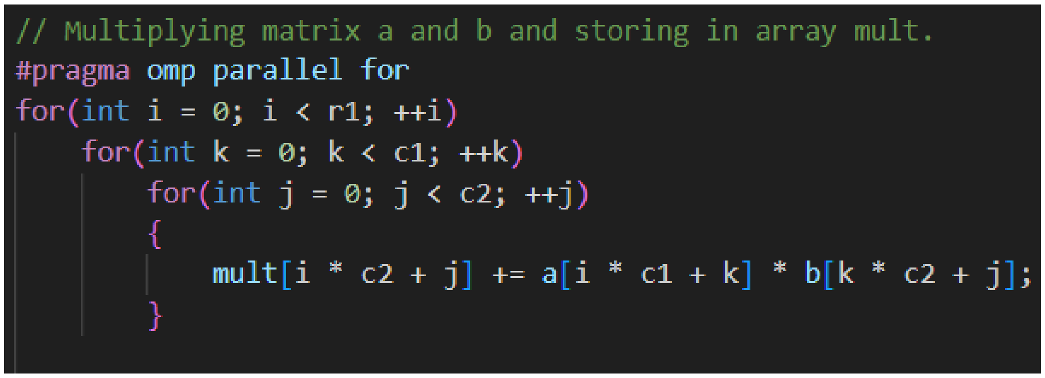
% of improvement			
with loop swap	318,12%	602,15%	701,70%
with contiguos allocation	694,84%	1083,50%	1179,18%
with loop swap + contiguous allocation	700,87%	1084,39%	1180,71%
with loop swap + contiguous allocation + xHost	1960,79%	2144,63%	1906,10%

Figure 10: Basic implementations improvements

6 OpenMP implementation

6.1 Basic implementation

With the aim to improve the performance of the program, starting from the best version of the program we obtained until now, i.e. the version with the loop swapping, the contiguous allocation and xHost, to which we implemented the OpenMP directives. We started adding the *#pragma omp parallel for* directive on the hotspot which allows to parallelize the inner-most loop.

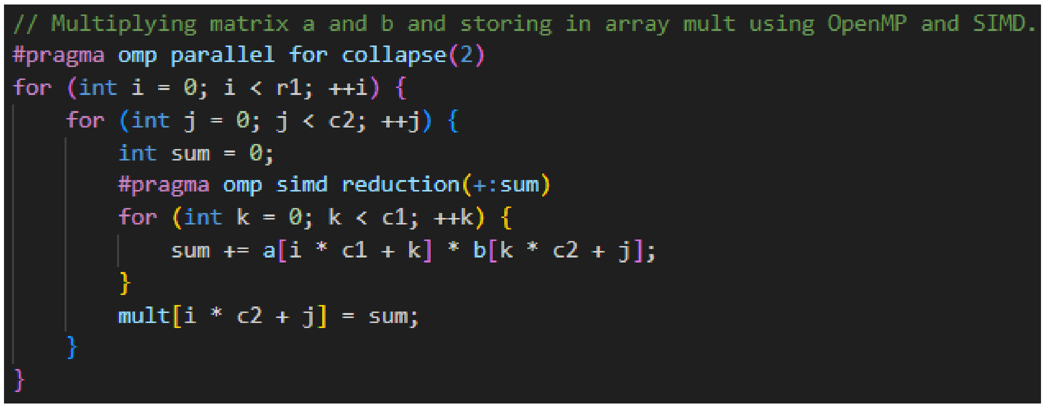


```
// Multiplying matrix a and b and storing in array mult.
#pragma omp parallel for
for(int i = 0; i < r1; ++i)
    for(int k = 0; k < c1; ++k)
        for(int j = 0; j < c2; ++j)
        {
            mult[i * c2 + j] += a[i * c1 + k] * b[k * c2 + j];
        }
```

Figure 11: OpenMP basic implementation

6.2 SIMD implementation

Then we introduced the SIMD (Single Input Multiple Data) directive adding *#pragma omp parallel for collapse(2)* and *#pragma omp simd reduction (+:sum)*. The *collapse* clause specifies that two nested loops must be treated as one loop for parallelization. The *simd* directive instructs the compiler to vectorize the cycle that follows, using SIMD instructions, this means that multiple iterations of the loop will be executed simultaneously using the vectorizing capabilities of the processor; while the *reduction(+:sum)* clause provides that a local copy the variable ”sum” is made and initialized for each thread which does its local accumulation first, then a global reduction, in particular a sum of the ”partial” results, is done at the end.

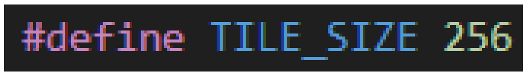


```
// Multiplying matrix a and b and storing in array mult using OpenMP and SIMD.
#pragma omp parallel for collapse(2)
for (int i = 0; i < r1; ++i) {
    for (int j = 0; j < c2; ++j) {
        int sum = 0;
        #pragma omp simd reduction(+:sum)
        for (int k = 0; k < c1; ++k) {
            sum += a[i * c1 + k] * b[k * c2 + j];
        }
        mult[i * c2 + j] = sum;
    }
}
```

Figure 12: SIMD implementation

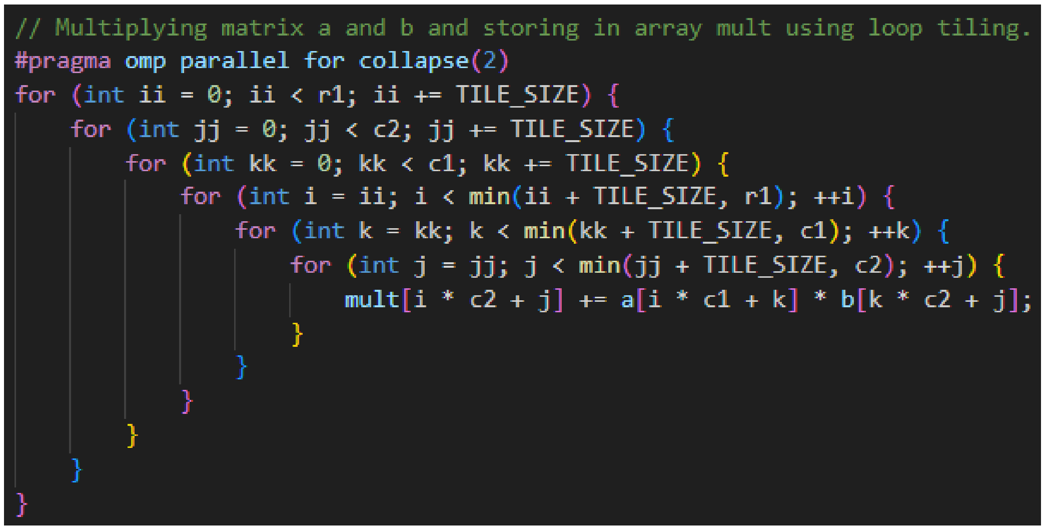
6.3 Loop tiling implementation

Loop tiling (or loop blocking) is an optimisation technique used to improve cache locality. The idea is to divide large loops into smaller blocks (or tiles) that can be contained in the cache, thus reducing cache misses and increasing cache efficiency. In particular, loops ii, jj and kk divide loops i, j and k respectively into blocks of size *TILE_SIZE*. This means that instead of iterating over all rows, columns or elements at once, the loop iterates over blocks of size *TILE_SIZE*. To select the best dimension for the *TILE_SIZE* we run the same program multiple times varying the tile size using powers of 2 dimension, until we obtained that best dimension for our problem and architecture is 256.



```
#define TILE_SIZE 256
```

Figure 13: TILE SIZE

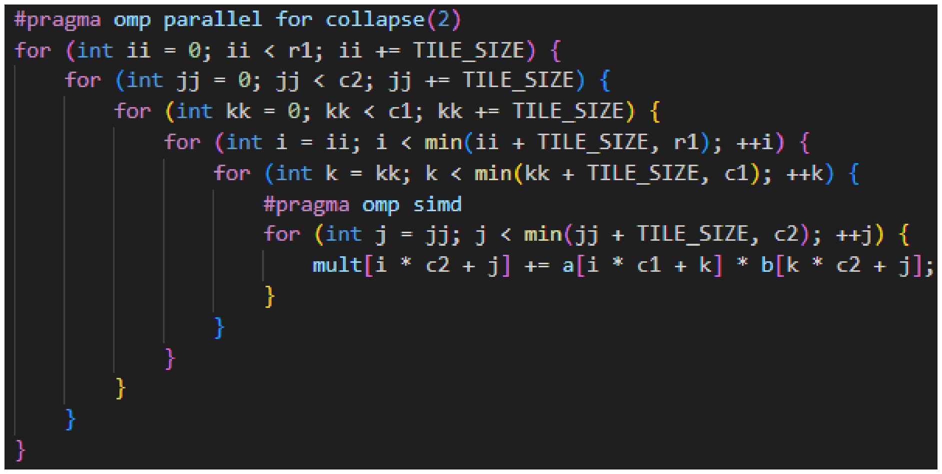


```
// Multiplying matrix a and b and storing in array mult using loop tiling.
#pragma omp parallel for collapse(2)
for (int ii = 0; ii < r1; ii += TILE_SIZE) {
    for (int jj = 0; jj < c2; jj += TILE_SIZE) {
        for (int kk = 0; kk < c1; kk += TILE_SIZE) {
            for (int i = ii; i < min(ii + TILE_SIZE, r1); ++i) {
                for (int k = kk; k < min(kk + TILE_SIZE, c1); ++k) {
                    for (int j = jj; j < min(jj + TILE_SIZE, c2); ++j) {
                        mult[i * c2 + j] += a[i * c1 + k] * b[k * c2 + j];
                    }
                }
            }
        }
    }
}
```

Figure 14: Loop tiling implementation

6.4 Final implementation

As final implementation we decided to apply both SIMD and loop tiling directives to the program hotspot so that each thread has the possibility to exploit the cache locality reducing the cache misses. With loop tiling, the variable *sum* and the *reduction(+:sum)* clause are not necessary because the update of the varaible *mult* occurs directly in a parallel and safe manner due to block splitting and efficient cache management.



```
#pragma omp parallel for collapse(2)
for (int ii = 0; ii < r1; ii += TILE_SIZE) {
    for (int jj = 0; jj < c2; jj += TILE_SIZE) {
        for (int kk = 0; kk < c1; kk += TILE_SIZE) {
            for (int i = ii; i < min(ii + TILE_SIZE, r1); ++i) {
                for (int k = kk; k < min(kk + TILE_SIZE, c1); ++k) {
                    #pragma omp simd
                    for (int j = jj; j < min(jj + TILE_SIZE, c2); ++j) {
                        mult[i * c2 + j] += a[i * c1 + k] * b[k * c2 + j];
                    }
                }
            }
        }
    }
}
```

Figure 15: SIMD and loop tiling implementation

6.5 Performance Analysis

Now we'll show a graph representing the execution times obtained implementing the directives described in the previous paragraph.

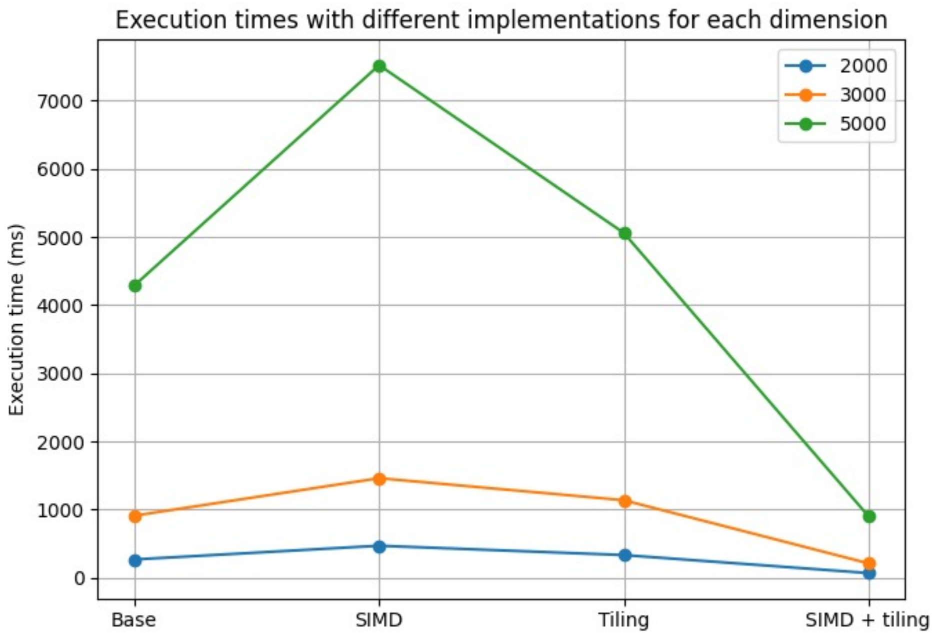
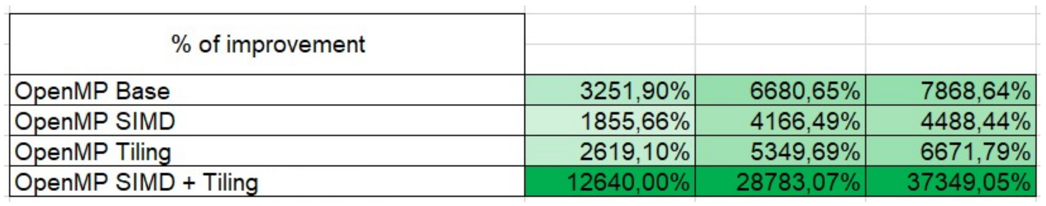


Figure 16: SIMD and loop tiling graph

Next we computed the percentages of improvements obtained using the OpenMP directives with respect to the execution time required for running the basic version of the program.



% of improvement			
OpenMP Base	3251,90%	6680,65%	7868,64%
OpenMP SIMD	1855,66%	4166,49%	4488,44%
OpenMP Tiling	2619,10%	5349,69%	6671,79%
OpenMP SIMD + Tiling	12640,00%	28783,07%	37349,05%

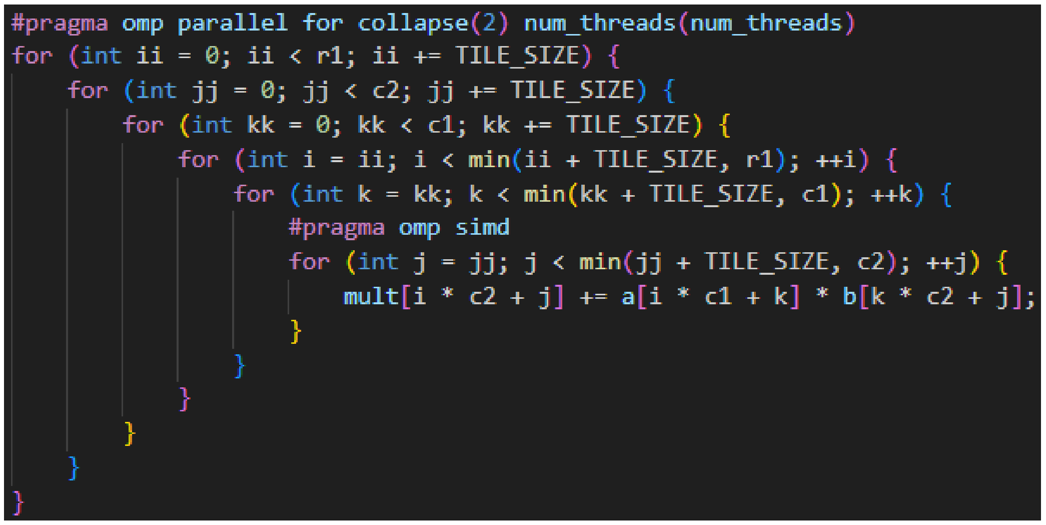
Figure 17: SIMD and loop tiling improvements

6.6 Results explanation

The analysis of execution times showed that the explicit use of OpenMP's SIMD directives did not produce the expected improvements over the basic version of the code; on the contrary, it worsened performance. This can be attributed to the fact that the starting code, already optimised to take advantage of the CPU's native SIMD instructions, had probably already vectorised some loops efficiently. The additional introduction of SIMD directives may therefore have led to over-optimisation, forcing vectorisation in situations where memory organisation was not optimal, thus compromising performance. However, when combined with tiling, which organises and prepares data more efficiently, SIMD can operate under ideal conditions, leading to significant performance improvements.

7 OpenMP threads

As last OpenMP implementation we tried to run the program making explicit the number of threads. We introduced the *#pragma omp parallel for num_threads(number_of_threads)* directive to allow effective distribution of the workload between the threads.



```
#pragma omp parallel for collapse(2) num_threads(num_threads)
for (int ii = 0; ii < r1; ii += TILE_SIZE) {
    for (int jj = 0; jj < c2; jj += TILE_SIZE) {
        for (int kk = 0; kk < c1; kk += TILE_SIZE) {
            for (int i = ii; i < min(ii + TILE_SIZE, r1); ++i) {
                for (int k = kk; k < min(kk + TILE_SIZE, c1); ++k) {
                    #pragma omp simd
                    for (int j = jj; j < min(jj + TILE_SIZE, c2); ++j) {
                        mult[i * c2 + j] += a[i * c1 + k] * b[k * c2 + j];
                    }
                }
            }
        }
    }
}
```

Figure 18: OpenMP threads implementation

7.1 Performance Analysis

Since our architecture can support at most 24 different threads (the architecture we used has 24 logical CPUs, then the maximum number of threads supported is equal to the total number of logical CPUs available) and thanks to an external script we run the code progressively increasing the number of threads:

$$num\_threads = 2^i \text{ con } i = 1, \ldots, 5$$

with a maximum of 32 threads (following the binary basis), and collected the performance data.

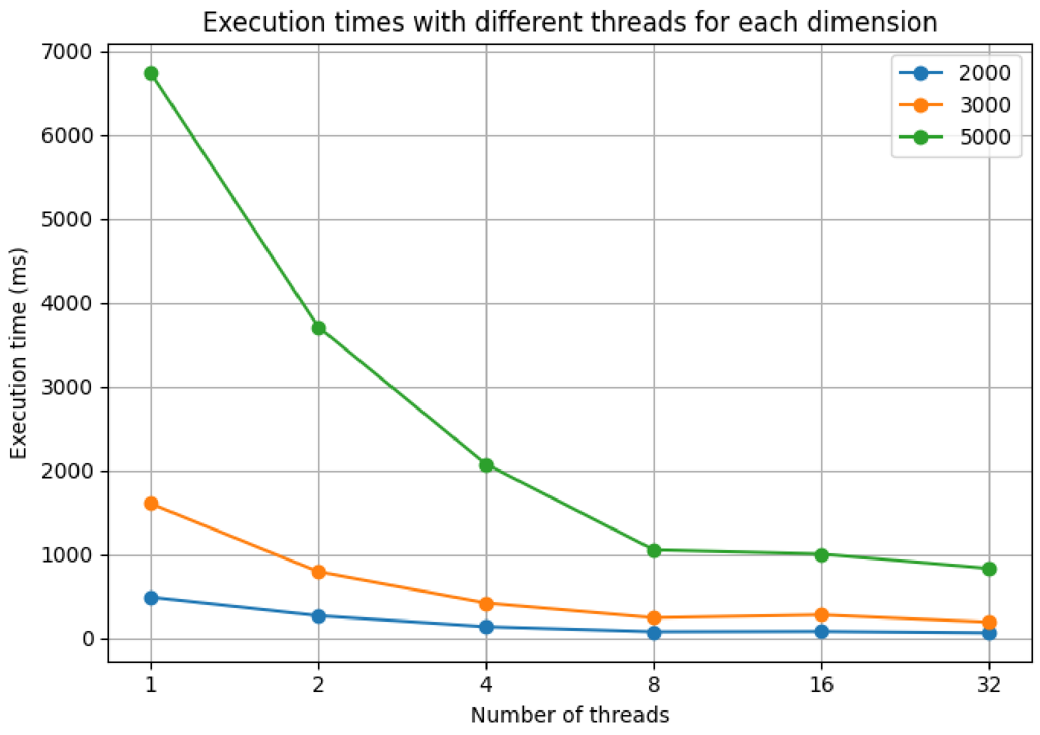
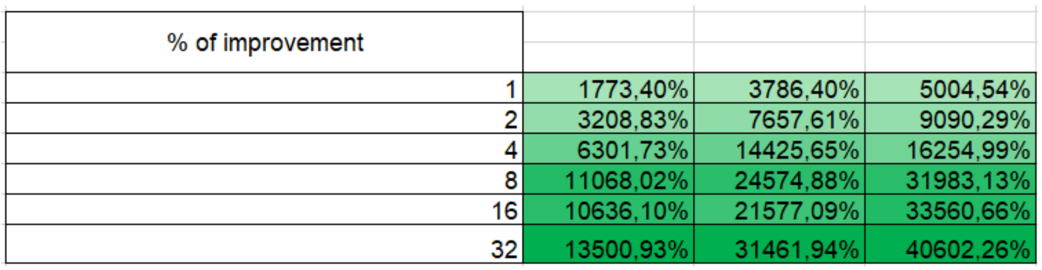


Figure 19: Execution times with different number of threads

We also created a table where we computed the percentage ratio between the execution time of the basic version and the execution times of the best version of the program run with different number of threads.



% of improvement			
1	1773,40%	3786,40%	5004,54%
2	3208,83%	7657,61%	9090,29%
4	6301,73%	14425,65%	16254,99%
8	11068,02%	24574,88%	31983,13%
16	10636,10%	21577,09%	33560,66%
32	13500,93%	31461,94%	40602,26%

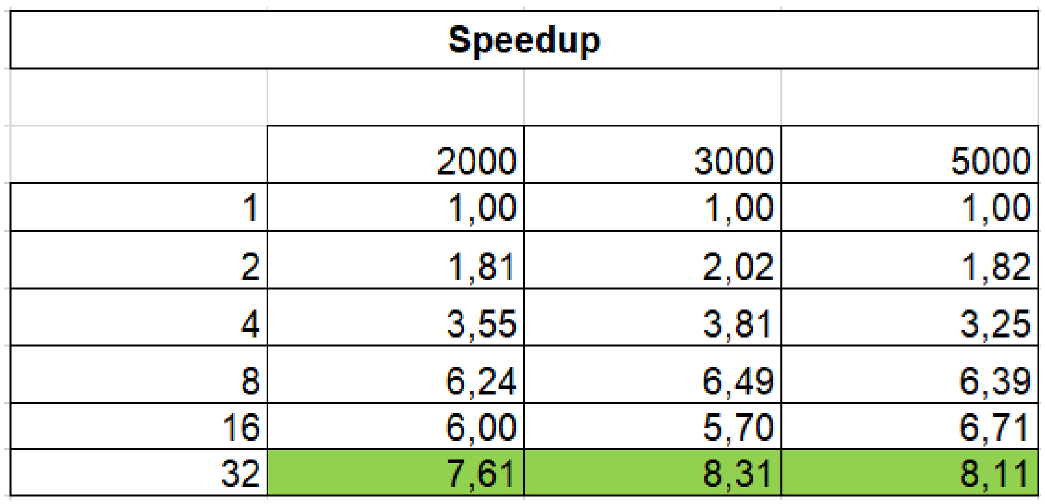
Figure 20: Improvements with different number of threads

8 The Speedup

Speedup tells us how much a program runs faster, using many processors at once, instead of using just one. If the speedup number is bigger, it means the program is using all the processors well. By looking at the speedup, we can figure out the best way to run the program so that it's fast and doesn't waste resources.

The speedup can be computed in this way:

$$\text{Speedup} = \frac{\text{Time on single processor}}{\text{Time on parallel system}}$$



Speedup

	2000	3000	5000
1	1,00	1,00	1,00
2	1,81	2,02	1,82
4	3,55	3,81	3,25
8	6,24	6,49	6,39
16	6,00	5,70	6,71
32	7,61	8,31	8,11

Figure 21: Speedup table

The next graph helps to analyze better the results we obtained computing the speedup.

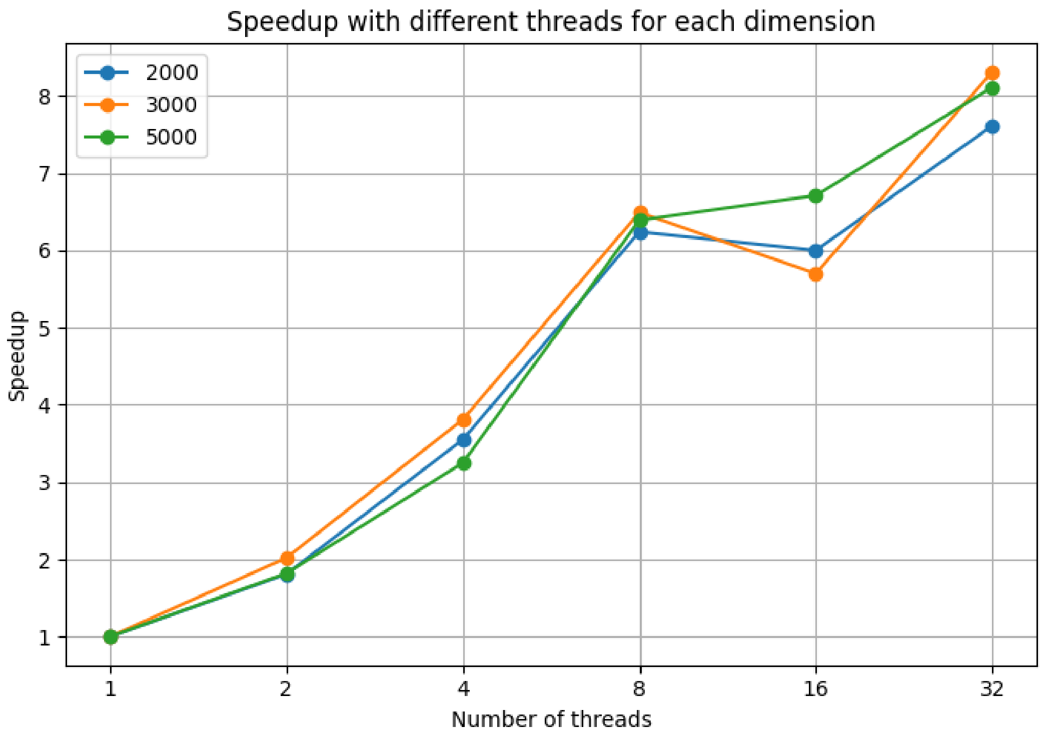


Figure 22: Speedup graph

The analysis revealed good scalability up to 8 threads, with speedups approaching the number of threads employed. However, beyond 8 threads, the performance increase does not grow linearly, indicating the presence of bottlenecks due, for example, to thread management overhead.

Larger matrices benefit more from parallelism, with higher speedups than smaller matrices, suggesting that larger workloads make better use of available parallelism. Nevertheless, with 16 and 32 threads, the overall efficiency of parallelism decreases, suggesting that management overhead and communication among threads become significant.

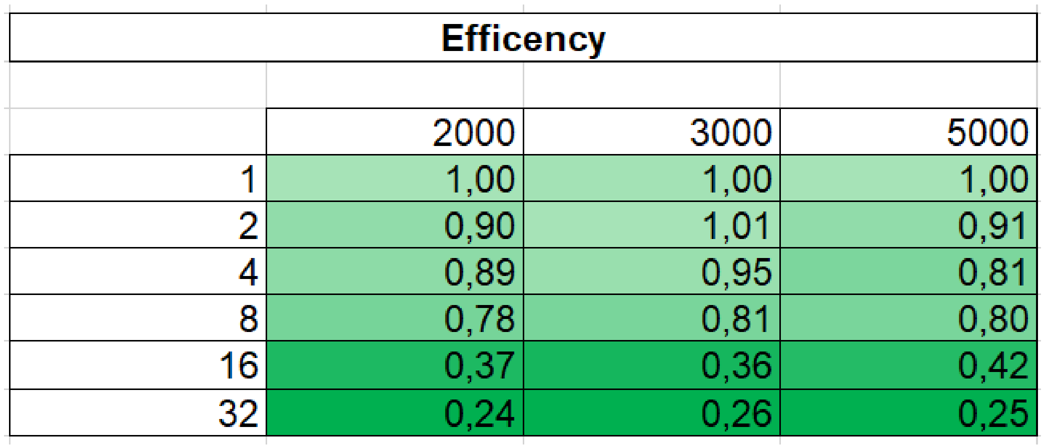
9 Efficiency respect to the number of threads

Efficiency is a metric that builds on top of speedup by adding awareness of the underlying hardware. [ref]

The Efficiency can be computed in this way:

$$E = \frac{\text{Speedup}}{\text{Number of Threads}}$$

The efficiency of a parallel program is a measure of how effectively additional threads improve performance. It assesses how well parallel resources (threads or processors) are utilised compared to sequential execution. When the efficiency is 1 (or 100%), it means that the parallel program uses all available resources optimally. This is the ideal, but rarely reachable. In practice, efficiency tends to decrease as the number of threads increases due to the overhead introduced by parallelism management (e.g. synchronisation, communication between threads).



Efficency

	2000	3000	5000
1	1,00	1,00	1,00
2	0,90	1,01	0,91
4	0,89	0,95	0,81
8	0,78	0,81	0,80
16	0,37	0,36	0,42
32	0,24	0,26	0,25

Figure 23: Efficiency table

10 CUDA implementation

10.1 Introduction to Cuda

As mentioned earlier in the section on CUDA (see 2.3.2), CUDA is a parallel computing platform and application programming interface (API) model by NVIDIA that significantly speeds up computations by breaking down tasks into parallel operations.

10.2 Cuda Memory Spaces

CUDA threads may access data from multiple memory spaces during their execution:

- **Global Memory**: Accessible by all threads and has the largest storage capacity but also the highest latency.
- **Shared Memory**: Shared among threads in the same block, much faster than global memory.
- **Local Memory**: Private to each thread.
- **Constant and Texture Memory**: Read-only memory spaces that are cached and can be accessed by all threads. They are beneficial for data that does not change throughout the kernel execution.
- **Unified Memory**: Allows the CPU and GPU to share the same memory space, simplifying memory management.

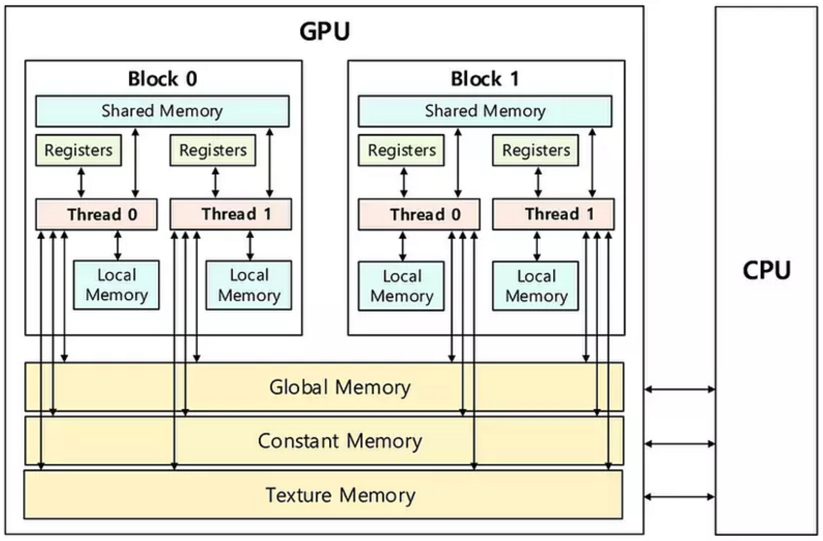


Figure 24: Memory Hierarchy

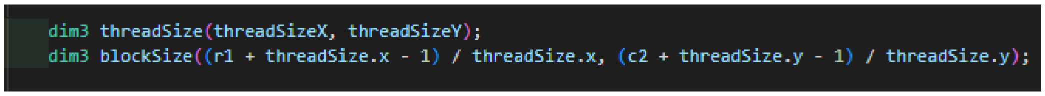
10.3 Implementation

The project includes different implementations of matrix multiplication using CUDA:

10.3.1 Global Memory Implementation:

In this implementation, the matrix is stored in a one-dimensional array and init with random values.

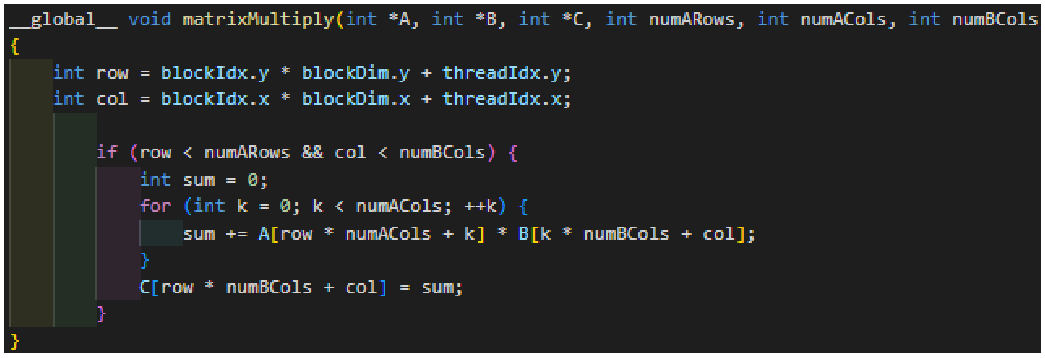
Each element of the resultant matrix is computed by a thread, where threads and blocks are organized in a two-dimensional space.



```
    dim3 threadSize(threadSizeX, threadSizeY);
    dim3 blockSize((r1 + threadSize.x - 1) / threadSize.x, (c2 + threadSize.y - 1) / threadSize.y);
```

Figure 25: Threads and Blocks Configuration

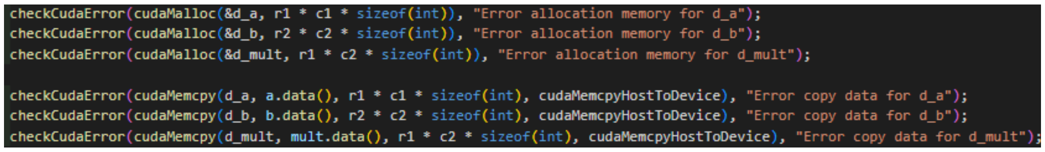
This involves iterating over the elements of the corresponding row of matrix A and the column of matrix B, multiplying them and summing up the products. The computed value is then stored in the appropriate position in the matrix C.



```
__global__ void matrixMultiply(int *A, int *B, int *C, int numARows, int numACols, int numBCols
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

        if (row < numARows && col < numBCols) {
            int sum = 0;
            for (int k = 0; k < numACols; ++k) {
                sum += A[row * numACols + k] * B[k * numBCols + col];
            }
            C[row * numBCols + col] = sum;
        }
}
```

Figure 26: Global Memory Kernel Function

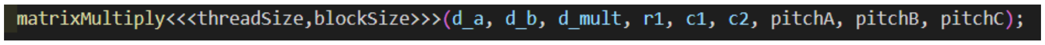
Data is transferred from the host (**CPU**) to the device (**GPU**) using '*cudaMalloc*' to allocate memory on the device and '*cudaMemcpy*' to copy the data.



```
checkCudaError(cudaMalloc(&d_a, r1 * c1 * sizeof(int)), "Error allocation memory for d_a");
checkCudaError(cudaMalloc(&d_b, r2 * c2 * sizeof(int)), "Error allocation memory for d_b");
checkCudaError(cudaMalloc(&d_mult, r1 * c2 * sizeof(int)), "Error allocation memory for d_mult");

checkCudaError(cudaMemcpy(d_a, a.data(), r1 * c1 * sizeof(int), cudaMemcpyHostToDevice), "Error copy data for d_a");
checkCudaError(cudaMemcpy(d_b, b.data(), r2 * c2 * sizeof(int), cudaMemcpyHostToDevice), "Error copy data for d_b");
checkCudaError(cudaMemcpy(d_mult, mult.data(), r1 * c2 * sizeof(int), cudaMemcpyHostToDevice), "Error copy data for d_mult");
```

Figure 27: Allocation in the Global Memory

To launch the kernel function, we run the kernel function as follows:



```
matrixMultiply<<<threadSize,blockSize>>>(d_a, d_b, d_mult, r1, c1, c2, pitchA, pitchB, pitchC);
```

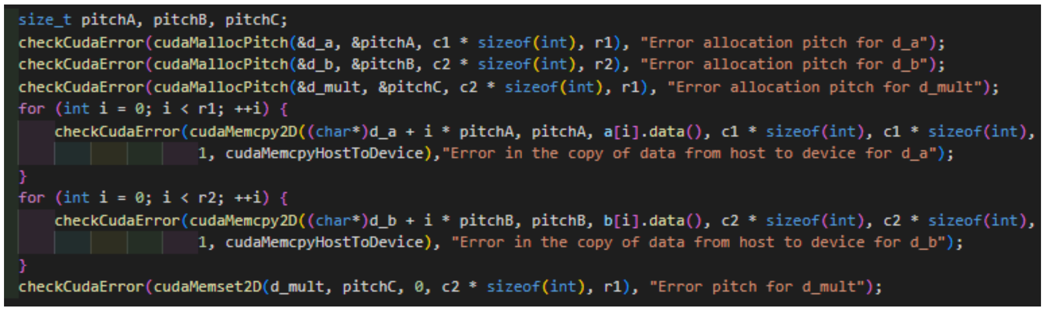
Figure 28: Kernel Launch Configuration

Note that 'pitch' has a default value when called by implementations that do not explicitly use it.

The results are copied back (using the same function) from the device to the host, and the allocated memory on the device is freed.

10.3.2 Pitched Memory Implementation:

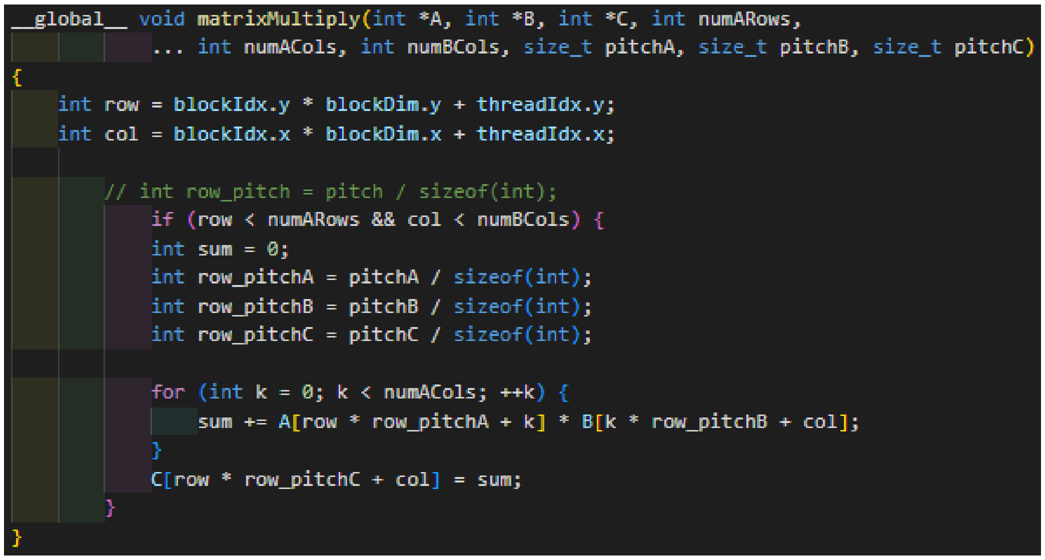
In this implementation, the matrix is stored in a two-dimensional array using pitched memory, which ensures that rows of the matrix are aligned, reducing memory transaction overhead. Data is transferred from **CPU** to **GPU** using '*cudaMemcpy2D*', which optimizes the memory layout.



```
size_t pitchA, pitchB, pitchC;
checkCudaError(cudaMallocPitch(&d_a, &pitchA, c1 * sizeof(int), r1), "Error allocation pitch for d_a");
checkCudaError(cudaMallocPitch(&d_b, &pitchB, c2 * sizeof(int), r2), "Error allocation pitch for d_b");
checkCudaError(cudaMallocPitch(&d_mult, &pitchC, c2 * sizeof(int), r1), "Error allocation pitch for d_mult");
for (int i = 0; i < r1; ++i) {
    checkCudaError(cudaMemcpy2D((char*)d_a + i * pitchA, pitchA, a[i].data(), c1 * sizeof(int), c1 * sizeof(int),
                1, cudaMemcpyHostToDevice),"Error in the copy of data from host to device for d_a");
}
for (int i = 0; i < r2; ++i) {
    checkCudaError(cudaMemcpy2D((char*)d_b + i * pitchB, pitchB, b[i].data(), c2 * sizeof(int), c2 * sizeof(int),
                1, cudaMemcpyHostToDevice), "Error in the copy of data from host to device for d_b");
}
checkCudaError(cudaMemset2D(d_mult, pitchC, 0, c2 * sizeof(int), r1), "Error pitch for d_mult");
```

Figure 29: Memory Allocation on the GPU with Pitched Memory

Each element of the resultant matrix is computed by a thread, where threads and blocks are organized in a two-dimensional space, similar to the global memory implementation (see Figure 28). Then, the results are copied back from the device to the host, and the allocated memory on the device is freed.



```
__global__ void matrixMultiply(int *A, int *B, int *C, int numARows,
        ... int numACols, int numBCols, size_t pitchA, size_t pitchB, size_t pitchC)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

        // int row_pitch = pitch / sizeof(int);
            if (row < numARows && col < numBCols) {
            int sum = 0;
            int row_pitchA = pitchA / sizeof(int);
            int row_pitchB = pitchB / sizeof(int);
            int row_pitchC = pitchC / sizeof(int);

            for (int k = 0; k < numACols; ++k) {
                sum += A[row * row_pitchA + k] * B[k * row_pitchB + col];
            }
            C[row * row_pitchC + col] = sum;
        }
}
```

Figure 30: Pitched Memory Kernel Function

10.3.3 Dimensionality Constraints of CUDA Implementations

While CUDA provides significant speedup for matrix multiplication through parallel computation, there are inherent limitations related to the dimensionality of the problem. These limitations arise due to the constraints on the number of threads and blocks that can be launched on a GPU. In our implementations, each thread computes a single element of the resultant matrix. The GPU architecture we are using can launch a maximum of 1024 threads per block and 1024 blocks per grid. Consequently, the total number of elements that can be computed simultaneously is limited to 1024 $\times$ 1024, which equals 1,048,576 elements.

Note: *For completeness and curiosity, we have conducted tests with this implementations but reducing the dimensions of matrices to 512 x 512 and 1024 x 1024. These results can be found in the files attached to this document or in the section 10.5.1.*

Possible Solutions

To address these limitations, the following solutions can be considered:

1. **Hardware Upgrade:** One solution is to upgrade the hardware.
2. **Tiling into Sub-matrices:** This approach involves dividing the matrix into smaller sub-matrices or tiles that fit within the GPU's thread and block limits.

10.3.4 Shared Memory Implementation

Shared memory is used as a cache (like an additional and faster memory) to reduce global/unified memory accesses, enhancing performance. This approach allows us to have a number of threads that can compute on smaller portions of the matrix and calculate the final result, without exceeding the hardware limit of our GPU's cores. The matrix is divided into tiles, and each tile is loaded into shared memory for fast access. It is the similar logic explained in the section : 6.3

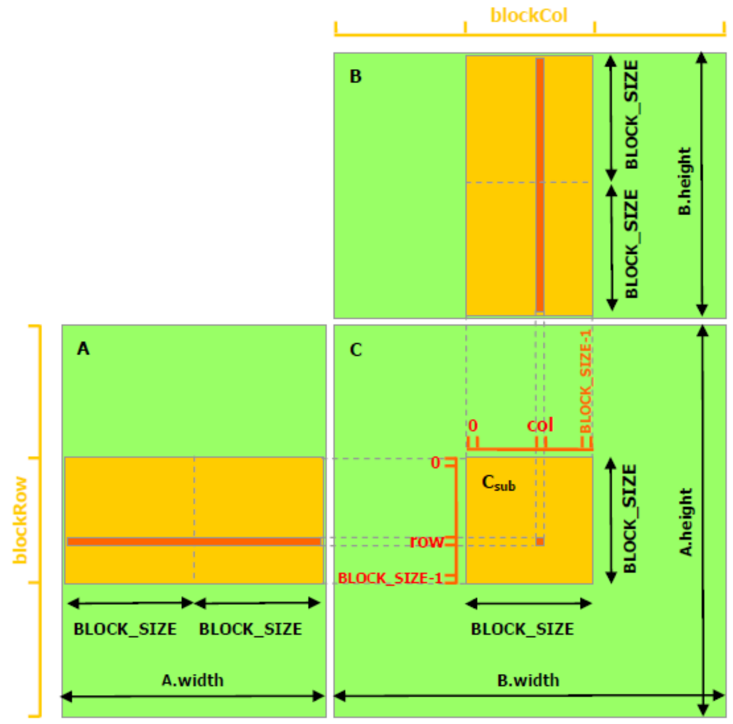
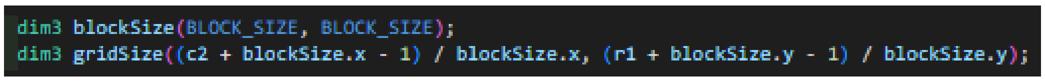


Figure 31: Matrix Multiplication with Shared Memory

Threads perform calculations on these tiles, minimizing the number of global/unified memory accesses and to have enough threads to compute the calculations.

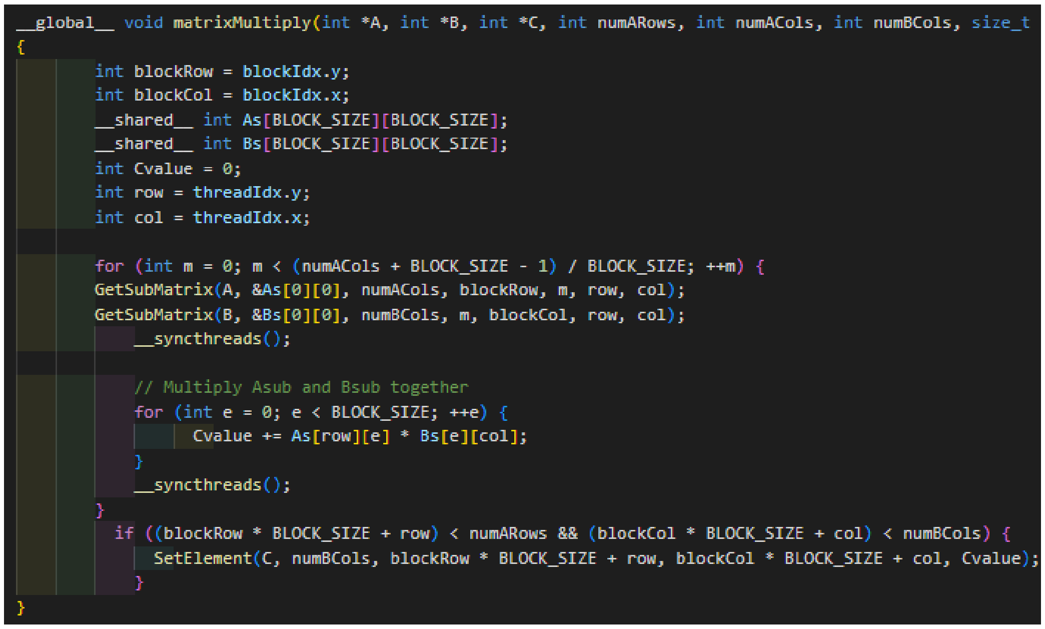


```
dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
dim3 gridSize((c2 + blockSize.x - 1) / blockSize.x, (r1 + blockSize.y - 1) / blockSize.y);
```

Figure 32: Configuration Block and Grid

Note: the variables ‘TILE_SIZE‘ and ‘BLOCK_SIZE‘ have the same meaning. The first is used for the OpenMP implementation, the second for Cuda one.

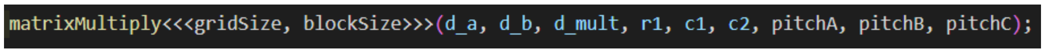
This approach leverages the speed of shared memory, which is much faster than global/unified memory. The computed results from shared memory are then accumulated and written back to memory.



```
__global__ void matrixMultiply(int *A, int *B, int *C, int numARows, int numACols, int numBCols, size_t
{
        int blockRow = blockIdx.y;
        int blockCol = blockIdx.x;
        __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];
        int Cvalue = 0;
        int row = threadIdx.y;
        int col = threadIdx.x;

        for (int m = 0; m < (numACols + BLOCK_SIZE - 1) / BLOCK_SIZE; ++m) {
        GetSubMatrix(A, &As[0][0], numACols, blockRow, m, row, col);
        GetSubMatrix(B, &Bs[0][0], numBCols, m, blockCol, row, col);
            __syncthreads();

            // Multiply Asub and Bsub together
            for (int e = 0; e < BLOCK_SIZE; ++e) {
                  Cvalue += As[row][e] * Bs[e][col];
            }
            __syncthreads();
        }
          if ((blockRow * BLOCK_SIZE + row) < numARows && (blockCol * BLOCK_SIZE + col) < numBCols) {
             SetElement(C, numBCols, blockRow * BLOCK_SIZE + row, blockCol * BLOCK_SIZE + col, Cvalue);
            }
}
```

Figure 33: Matrix Multiplication with Shared Memory

To launch the kernel function, we run the kernel function as follows:



```
matrixMultiply<<<gridSize, blockSize>>>(d_a, d_b, d_mult, r1, c1, c2, pitchA, pitchB, pitchC);
```

Figure 34: Running kernel function

We launch the kernel function by configuring blocks and grids, unlike the previous implementation where we configured threads and calculated the necessary blocks (see Figure 28). This approach ensures that a defined number of threads share memory together, optimizing access to shared memory and improving computation efficiency.

Implementation Approaches

We explored two main approaches for managing memory in our shared memory implementation: using global memory and using unified memory.

1. Global Memory Approach

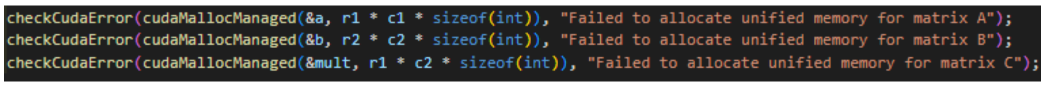
Here is how the process is structured:

1. **Tile Division**: The matrix is divided into smaller sub-matrices (tiles).
2. **Loading Tiles into Shared Memory**: Threads in a block cooperatively load a tile from global memory into shared memory.
3. **Performing Calculations**: Once the tile is in shared memory, threads perform the necessary calculations on it.
4. **Accumulating Results**: Intermediate results are stored in shared memory.
5. **Writing Back to Global Memory**: The final results are written back to global memory from shared memory.

2. Unified Memory Approach

The process is similar to the global memory approach but with some key differences:

1. **Unified Memory Management**: The matrix is managed using unified memory, allowing both the CPU and GPU to access the same memory space (see Figure 35).
2. **Tile Division and Loading**: The matrix is divided into tiles, which are loaded into shared memory directly from unified memory.
3. **Performing Calculations**: Threads perform the necessary calculations on the tiles in shared memory.
4. **Accumulating and Writing Results**: Intermediate results are accumulated in shared memory and then written back to unified memory.



```
checkCudaError(cudaMallocManaged(&a, r1 * c1 * sizeof(int)), "Failed to allocate unified memory for matrix A");
checkCudaError(cudaMallocManaged(&b, r2 * c2 * sizeof(int)), "Failed to allocate unified memory for matrix B");
checkCudaError(cudaMallocManaged(&mult, r1 * c2 * sizeof(int)), "Failed to allocate unified memory for matrix C");
```

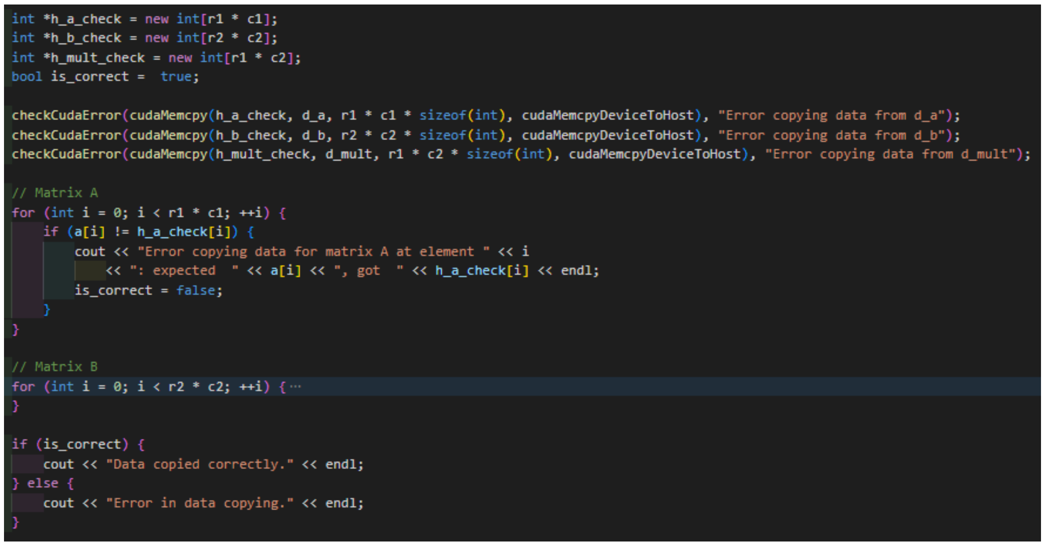
Figure 35: Unified Memory declaration

10.4 Code Correctness Verification

In this section, we discuss the various methods used to ensure the correctness of the CUDA implementations for matrix multiplication. Ensuring the correctness of the code is important for validating the results and confirming that the parallel computation is executed properly. The following techniques were implemented:

Post-Copy Verification

After transferring data from the host (**CPU**) to the device (**GPU**), the data is copied back to the host to verify that it has been correctly transferred. This step ensures that the data integrity is maintained during the transfer process.



```
int *h_a_check = new int[r1 * c1];
int *h_b_check = new int[r2 * c2];
int *h_mult_check = new int[r1 * c2];
bool is_correct =  true;

checkCudaError(cudaMemcpy(h_a_check, d_a, r1 * c1 * sizeof(int), cudaMemcpyDeviceToHost), "Error copying data from d_a");
checkCudaError(cudaMemcpy(h_b_check, d_b, r2 * c2 * sizeof(int), cudaMemcpyDeviceToHost), "Error copying data from d_b");
checkCudaError(cudaMemcpy(h_mult_check, d_mult, r1 * c2 * sizeof(int), cudaMemcpyDeviceToHost), "Error copying data from d_mult");

// Matrix A
for (int i = 0; i < r1 * c1; ++i) {
    if (a[i] != h_a_check[i]) {
        cout << "Error copying data for matrix A at element " << i
            << ": expected  " << a[i] << ", got  " << h_a_check[i] << endl;
        is_correct = false;
    }
}

// Matrix B
for (int i = 0; i < r2 * c2; ++i) {⋯
}

if (is_correct) {
    cout << "Data copied correctly." << endl;
} else {
    cout << "Error in data copying." << endl;
}
```

Figure 36: Post-Copy Verification Process

10.5 Performance Analysis

Now we'll discuss the performance analysis of the different CUDA implementations for matrix multiplication.

The primary metric for comparison is execution time, measured in milliseconds. Each configuration is tested 15 times, and the average execution time is reported. Problem sizes: 2000 x 2000, 3000 x 3000, and 5000 x 5000 matrices.

10.5.1 Preliminary Tests (*Completeness and curiosity*)

These tests aimed to explore how the program behaves under the dimensionality constraints of our hardware without employing tiling techniques. Problem size : 1024 x 1024. For these tests, we utilized the implementations discussed in the following sections:

- **Global Memory Implementation:** Details can be found in Section 10.3.1.
- **Pitched Memory Implementation:** Details can be found in Section 10.3.2.

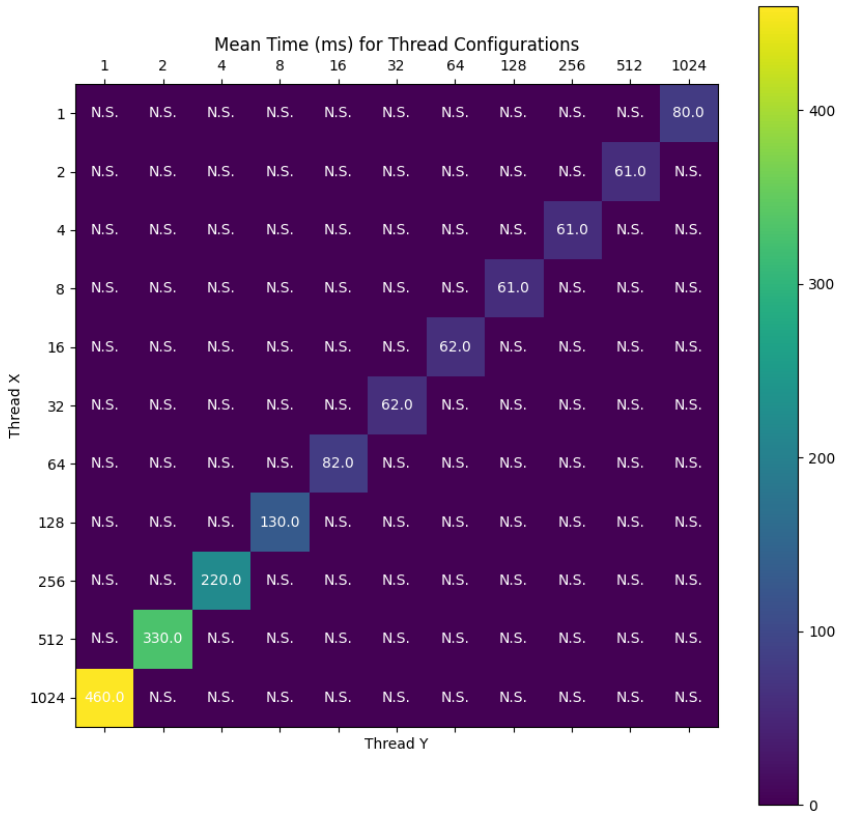


Figure 37: Performance with 1024x1024 Matrices using Global Memory

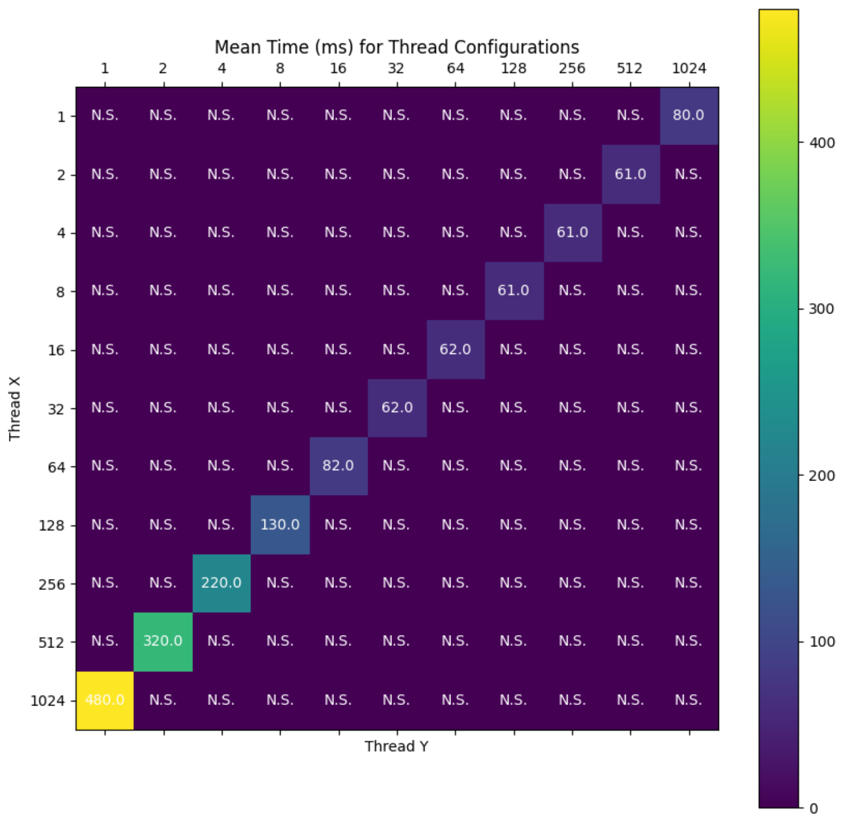


Figure 38: Performance with 1024x1024 Matrices using Pitched Memory

And we also try unified memory implementation (*more details in the next section*):

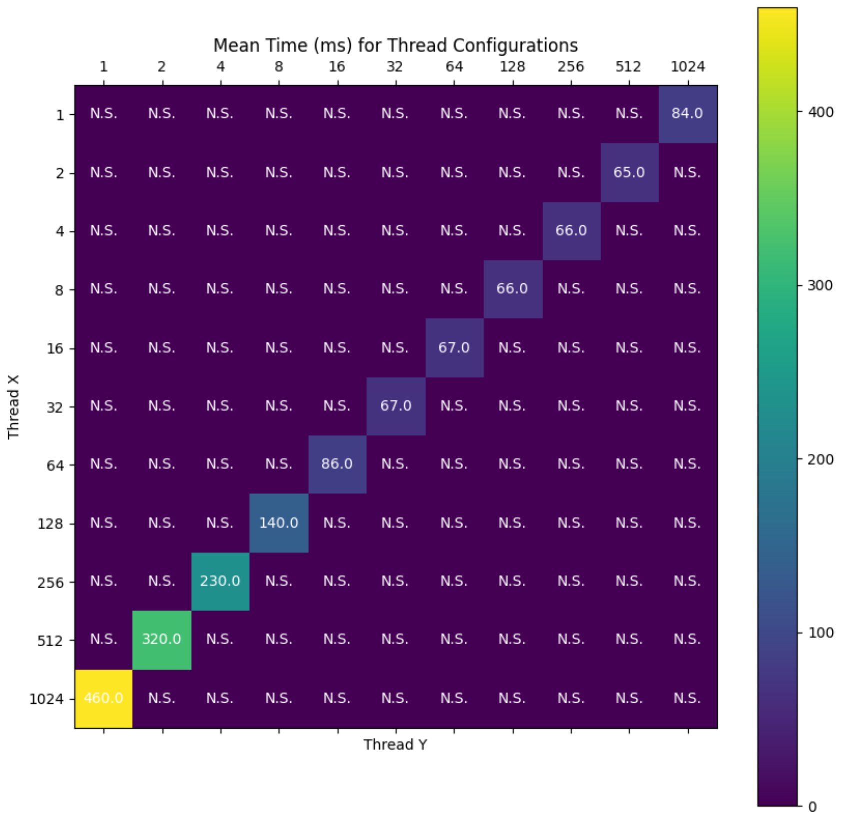
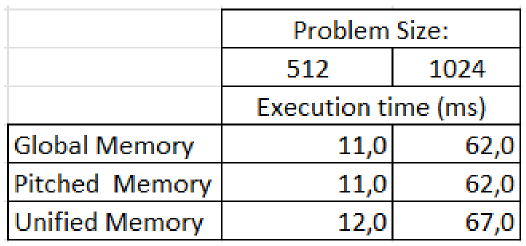


Figure 39: Performance with 1024x1024 Matrices using Unified Memory



	Problem Size:	
	512	1024
	Execution time (ms)	
Global Memory	11,0	62,0
Pitched Memory	11,0	62,0
Unified Memory	12,0	67,0

Figure 40: Execution time with configuration (32, 32)

10.5.2 Choosing tile size

We determined that the optimal tile size is 32 by experimenting with sizes 2, 4, 8, 16, and 32. Performance improved significantly up to a tile size of 32, which provided the best balance between shared memory usage and computational efficiency. The maximum tile size is 32 because 32 × 32 = 1024 and we can have a maximum of 1024 block per grid. To identify the best tile size, we made tests using the matrices dimension equal to 2000.

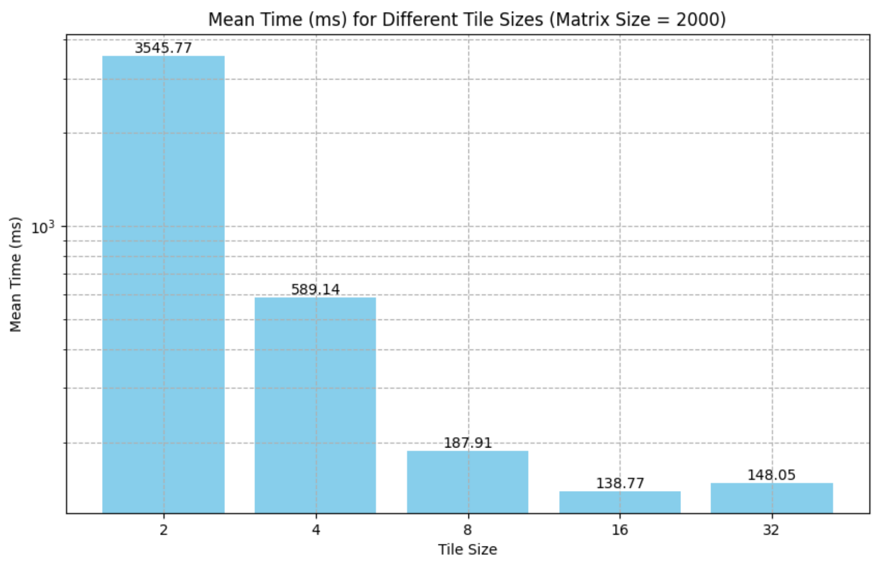


Figure 41: Mean Time (ms) for Different Tile Sizes

Given the minimal difference between 16 and 32, we also tested with the other sizes.

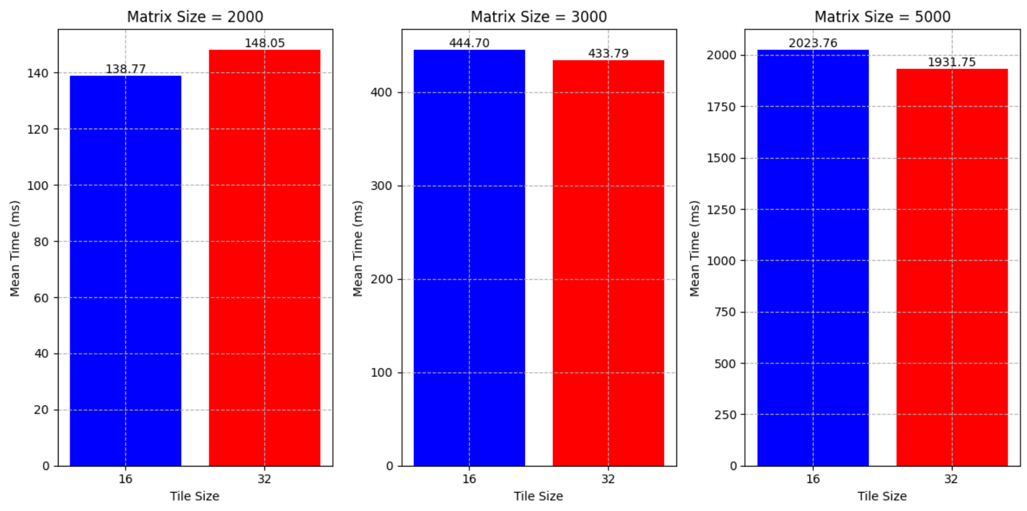
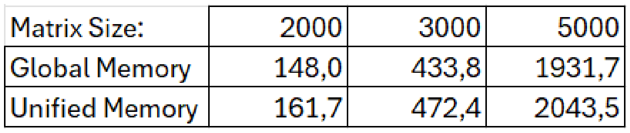


Figure 42: Choosing between 16 or 32

After determining that a tile size of 32 provides optimal performance, we conducted further experiments with the shared memory approach:

Global Memory Approach: Matrices are initially stored in global memory.

Unified Memory Approach: Matrices are managed using unified memory, allowing both CPU and GPU to share the same memory space.



Matrix Size:	2000	3000	5000
Global Memory	148,0	433,8	1931,7
Unified Memory	161,7	472,4	2043,5

Figure 43: Execution time (ms) using shared memory, tile_size = 32

For a graphic confrontation:

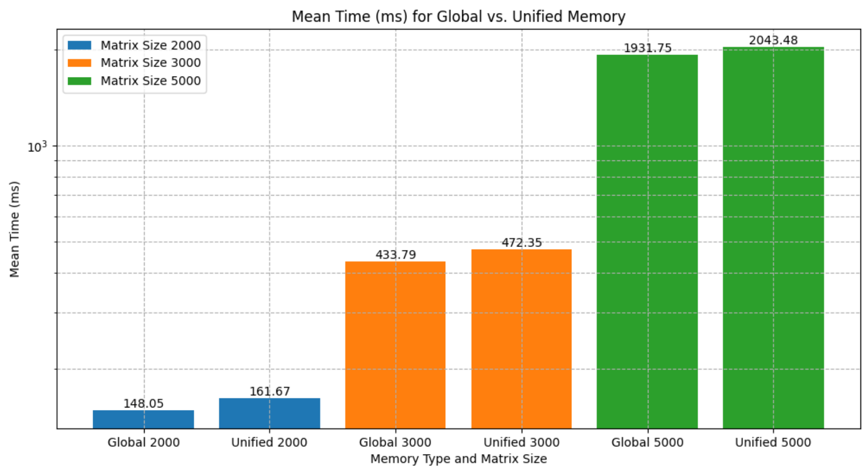
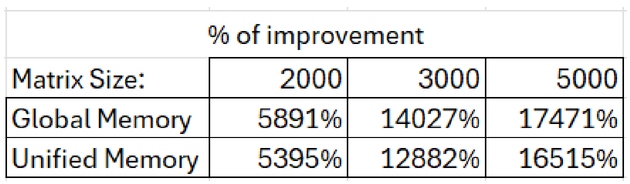


Figure 44: Mean Time (ms) for Global vs. Unified Memory

We also computed the percentages of improvements with respect to the execution time required for running the basic version of the program.

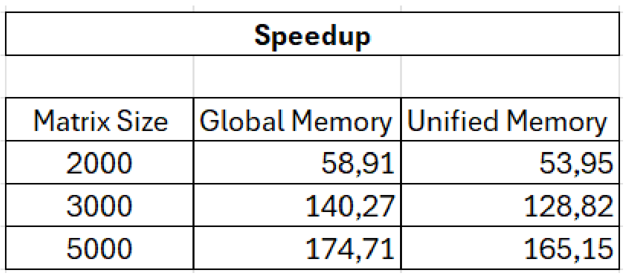


% of improvement			
Matrix Size:	2000	3000	5000
Global Memory	5891%	14027%	17471%
Unified Memory	5395%	12882%	16515%

Figure 45: Improvement respect basic execution

10.6 The Speedup

We calculated the speed up increase (*as in the section* 8)



Speedup

Matrix Size	Global Memory	Unified Memory
2000	58,91	53,95
3000	140,27	128,82
5000	174,71	165,15

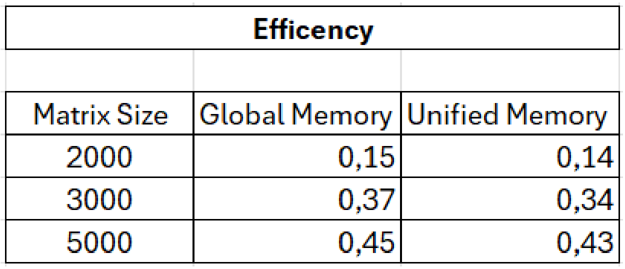
Figure 46: Speedup comparison

10.7 The Efficiency respect Cuda cores

In this section, we calculated the efficiency (as discussed in Section 9) to understand if the resources are well exploited.

In this case, we have to make a slight modification to the formula. Knowing that "*Job efficiency describes how well a job makes use of the available resources*" (source: HPC Wiki), we computed it based on the number of cores (in our case 384).

$$E = \frac{\text{Speedup}}{\text{Number of cores}}$$



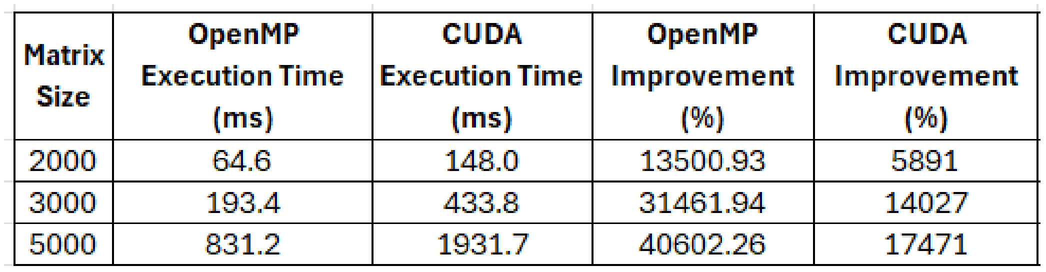
Efficency

Matrix Size	Global Memory	Unified Memory
2000	0,15	0,14
3000	0,37	0,34
5000	0,45	0,43

Figure 47: Efficiency comparison

11 Comparison and Results

In conclusion, we compare the best implementations of OpenMP and CUDA for the matrix multiplication operation, using metrics such as execution time, percentage improvement over the base version and speedup. The matrix sizes used in the tests were 2000x2000, 3000x3000 and 5000x5000.



Matrix Size	OpenMP Execution Time (ms)	CUDA Execution Time (ms)	OpenMP Improvement (%)	CUDA Improvement (%)
2000	64.6	148.0	13500.93	5891
3000	193.4	433.8	31461.94	14027
5000	831.2	1931.7	40602.26	17471

Figure 48: Comparison Cuda and OpenMP

Here are the graphs:

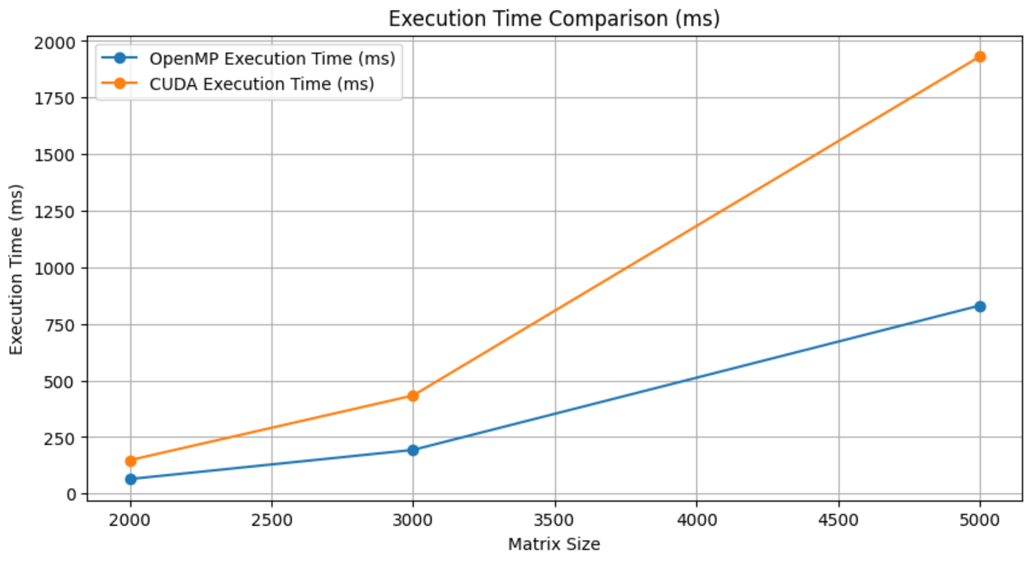


Figure 49: Comparison Execution Time

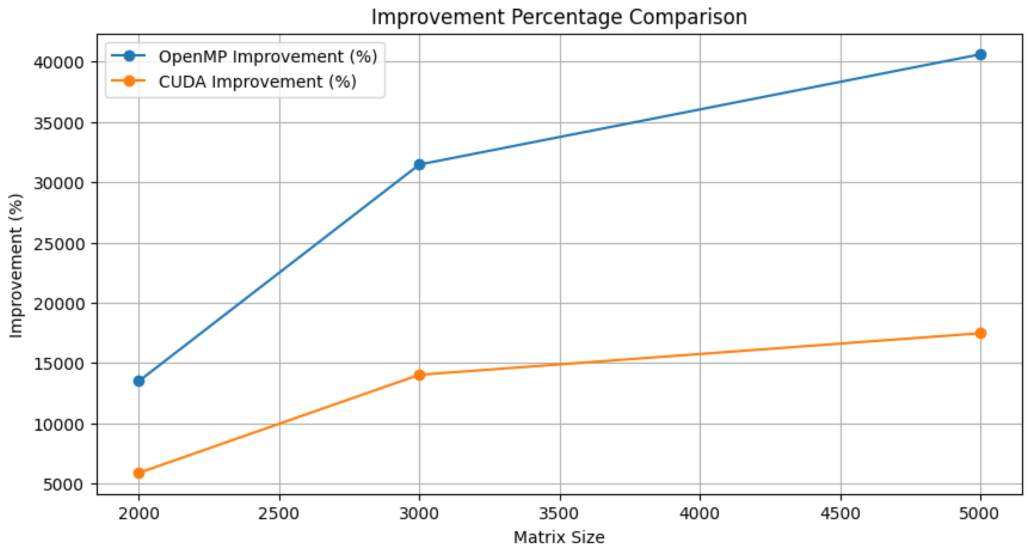


Figure 50: Comparison % improvement Time

The data showed that the **OpenMP** implementation showed a significant improvement over the basic implementation. In particular it achieved significant improvements (40,000% for matrix sizes 5000x5000) .

On the other hand, the **CUDA** implementation has proven to be very effective, but with some limitations due to handling matrix sizes and data transfer between CPU and GPU. Some highlights include:

- CUDA showed considerable improvements, but less than OpenMP.
- Memory management and data transfer affected the overall performance.
- Splitting the arrays into sub-blocks to fit the size limits of the GPU introduced an overhead that limited the overall performance.

Additional Considerations:

- **Hardware Limitations**: The CUDA implementation was limited by the available hardware resources. More powerful GPUs with more CUDA cores could have greatly improved performance.