Assignment 3 (Matrix Multiplication using CUDA)

Anas Albaqeri

202004427

Git: https://github.com/Anas-Albaqeri/Paralllelizing-Matrix-Multiplications-using-CUDA-/tree/main

Introduction

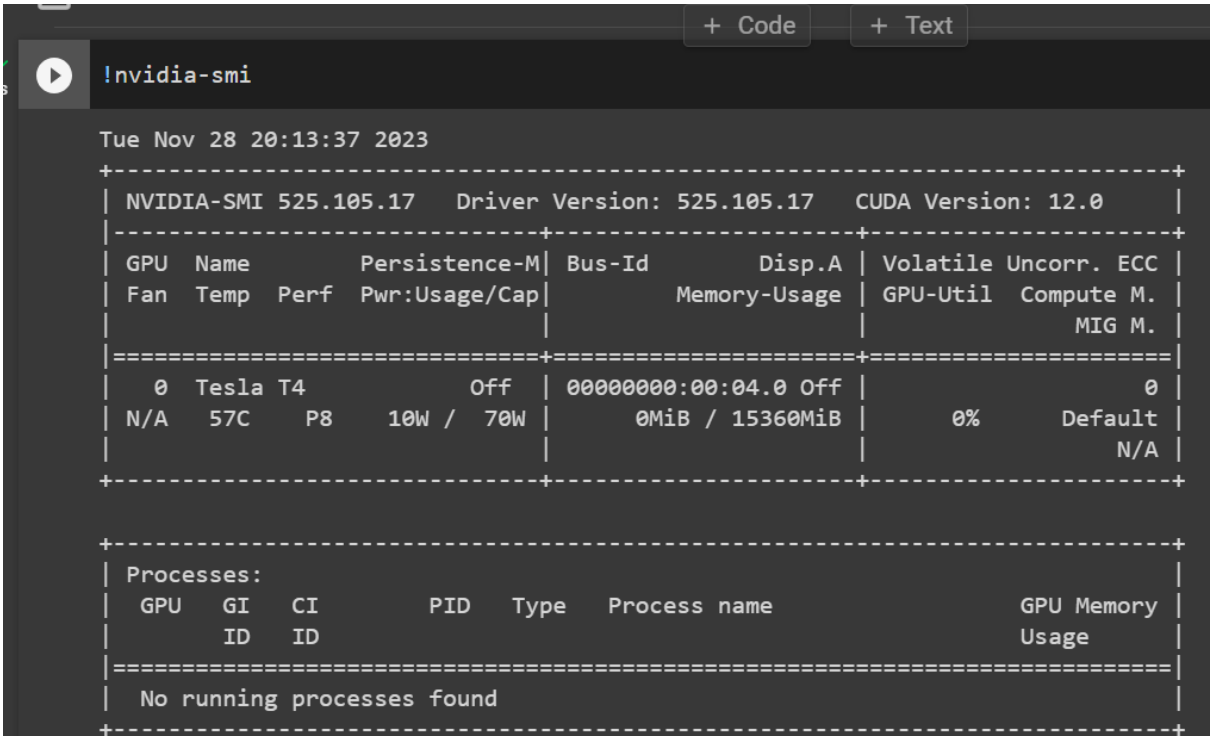
Tackling the issue of efficiently multiplying large matrices has always been a topic of research in scientific computing. The breakthrough in this issue is by using parallel methods of computing and especially utilizing modern GPUs using CUDA or OpenCL. For the purpose of this assignment, CUDA kernel will be used as the primary method through Google Collaboratory, however; since I am running an AMD GPU native in my local machine, I will include some personal tests I ran using open cl.

The CUDA kernel is a group of instructions basically designed to organize the specifications of parallelism in the GPU. This includes the level and number of threads working simultaneously in the GPU and the memory pool shared amongst them including the structuring of that memory.

By choosing correct dimensions for the working teams i.e. the blocks and grids, the program ensures the GPU's parallel work is put to maximum capabilities.

Tools Used:

The following CUDA programs were all run using Google Colab- free access, Google Colab provides an Nvidia Tesla T4 GPU with 16 gb of VRAM and total of 1024 threads which is very capable and therefore all tests will be run using GPU even sequentially to get a real reflection on speedup.



Matrix Multiplication without using tiling

To preform the matrix multiplication we need two main programs running; the kernel function and the main function.

A. Kernel function

the CUDA kernel function is the main function responsible for performing the multiplication and it is executed in parallel by multiple threads in the GPU.

Each thread is then responsible for computing a single element of the output matrix.

below is the detailed breakdown of the function

```
__global__ void multiplyMatrices(const float *inputMatrixA, const float *inputMatrixB, float *outputMatrixC, int rowsA, int colsA, int
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Ensure that the thread is within the valid matrix dimensions
    if (row < rowsA && col < colsB)
    {
        float result = 0;

        // Perform dot product for the given element of the output matrix
        for (int i = 0; i < colsA; i++)
        {
            result += inputMatrixA[row * colsA + i] * inputMatrixB[i * colsB + col];
        }

        // Save the result to the output matrix
        outputMatrixC[row * colsB + col] = result;
    }
}
```

1. `__global__ void multiplyMatrices(const float *inputMatrixA, const float *inputMatrixB, float *outputMatrixC, int rowsA, int colsA, int colsB)`
    - `__global__` is a CUDA keyword indicating that this function is a GPU kernel, which will be called from the host and executed on the device.

    In CUDA, a block is used as a group of threads that can be scheduled together and share a common region of fast on-chip memory called shared memory. The block index refer to the index assigned to a specific block within the grid of blocks that execute the kernel on the GPU.

2. `int row = blockIdx.y * blockDim.y + threadIdx.y;` and `int col = blockIdx.x * blockDim.x + threadIdx.x;`
    - Calculate the global index of the current thread within the 2D grid of threads. `blockIdx` gives the block index, `blockDim` is the size of a block, and `threadIdx` gives the thread index within a block.
3. `if (row < rowsA && col < colsB)`
    - Check if the calculated global indices are within the valid dimensions of the output matrix (`outputMatrixC`).
4. `float result = 0;`
    - Initialize a variable `result` that would hold the results of the dot product.
5. `for (int i = 0; i < colsA; i++)`
    - Loop over the common dimension of the matrices (`colsA`) to perform the dot product.
6. `result += inputMatrixA[row * colsA + i] * inputMatrixB[i * colsB + col];`
    - Perform the dot product by multiplying corresponding elements from `inputMatrixA` and `inputMatrixB` and accumulating the result.
7. `outputMatrixC[row * colsB + col] = result;`
    - Save the final result to the appropriate position in the output matrix (`outputMatrixC`).

Multiplication is done column by row like shown in the following figure:

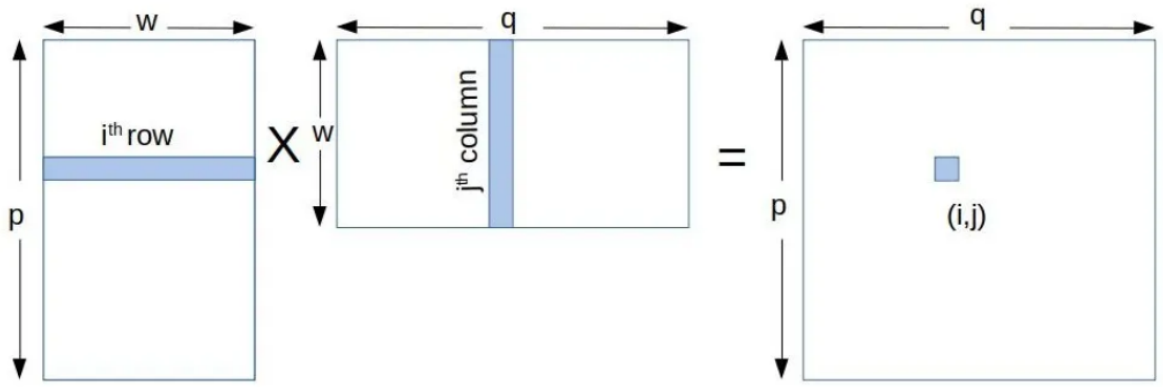


Fig. 1: What happens in matrix multiplication?

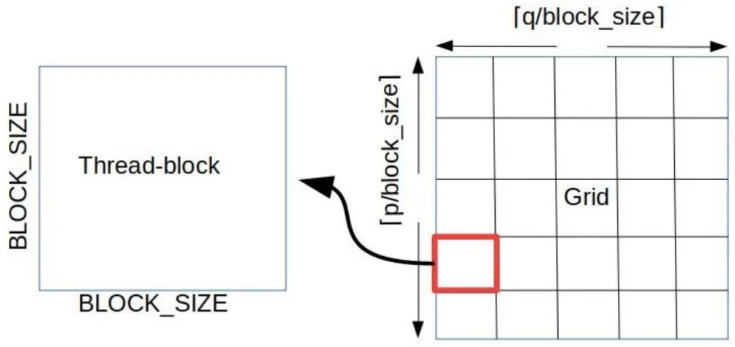


Fig.2 : Thread-block and grid organization for simple matrix multiplication

In a guide on matrix multiplication using CUDA by Analytics Vidhya (2020), the recommendation is made to arrange thread-blocks and the grid in a 2-D fashion when multiplying 2-D matrices. The source notes that in most modern NVIDIA GPUs, one thread-block can accommodate a maximum of 1024 threads (Analytics Vidhya, 2020).

This means that we could use all of these different block sizes to measure the performance of our program: 4x4, 8x8, 16x16 , 32x32.

B. Main function

```
int main()
{
    // Set matrix dimensions
    int rowsA = 10000;
    int colsA = 30000;
    int colsB = 20000;

    // Allocate memory on the host for input and output matrices
    float *matrixA, *matrixB, *matrixC;
    matrixA = (float *)malloc(rowsA * colsA * sizeof(float));
    matrixB = (float *)malloc(colsA * colsB * sizeof(float));
    matrixC = (float *)malloc(rowsA * colsB * sizeof(float));

    // Initialize input matrices with random values
    for (int i = 0; i < rowsA * colsA; i++)
        matrixA[i] = rand() / (float)RAND_MAX;
    for (int i = 0; i < colsA * colsB; i++)
        matrixB[i] = rand() / (float)RAND_MAX;

    // Allocate memory on the device for input and output matrices
    float *deviceMatrixA, *deviceMatrixB, *deviceMatrixC;
    cudaMalloc((void **)&deviceMatrixA, rowsA * colsA * sizeof(float));
    cudaMalloc((void **)&deviceMatrixB, colsA * colsB * sizeof(float));
```

```
    cudaMalloc((void **)&deviceMatrixC, rowsA * colsB * sizeof(float));

    // Copy input matrices from host to device memory
    cudaMemcpy(deviceMatrixA, matrixA, rowsA * colsA * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceMatrixB, matrixB, colsA * colsB * sizeof(float), cudaMemcpyHostToDevice);

    // Set block and grid dimensions for CUDA kernel execution
    dim3 blockDim(16, 16);
    dim3 gridDim((colsB + blockDim.x - 1) / blockDim.x, (rowsA + blockDim.y - 1) / blockDim.y);

    // Create and record events
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    // Launch CUDA kernel for matrix multiplication
    multiplyMatrices<<<gridDim, blockDim>>>(deviceMatrixA, deviceMatrixB, deviceMatrixC, rowsA, colsA, colsB);

    // Stop and destroy timer
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    // calculate elapsed time and print it
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, start, stop);
    printf("Execution Time: %f ms\n", elapsedTime);

    // Copy output matrix from device to host memory
    cudaMemcpy(matrixC, deviceMatrixC, rowsA * colsB * sizeof(float), cudaMemcpyDeviceToHost);

    // Display the result matrix
    // printf("Result Matrix:\n");
    // displayMatrix(matrixC, rowsA, colsB);

    // Free device memory
    cudaFree(deviceMatrixA);
    cudaFree(deviceMatrixB);
    cudaFree(deviceMatrixC);

    // Free host memory
    free(matrixA);
    free(matrixB);
    free(matrixC);

    return 0;
}
```

1. **Matrix Dimensions**: First we start by Define the dimensions of matrices `matrixA`, `matrixB`, and `matrixC`. In this example, `matrixA` is a 10000x30000 matrix, `matrixB` is a 30000x20000 matrix, and `matrixC` will be the resulting 10000x20000 matrix.
2. **Memory Allocation**: Allocate memory on the host for input and output matrices using `malloc`.
3. **Matrix Initialization**: Populate `matrixA` and `matrixB` with random float values between 0 and 1.
4. **Device Memory Allocation**: Allocate memory on the device (GPU) for input and output matrices using `cudaMalloc`.
5. **Memory Copy**: Copy the input matrices (`matrixA` and `matrixB`) from the host to the device.
6. **Block and Grid Dimensions**: Set the dimensions for CUDA kernel execution. In this case, a 2D grid with blocks of sizes 16x16 by default. However, I did ran a script with different block sizes and matrix sizes that was specified as follows:

```
%%bash
#!/bin/bash

# Compile the CUDA code
nvcc -o optimized_matrix_mult optimized_matrix_mult.cu

# Specify the output file
output_file="matrix_mult_results.txt"

# Loop through matrix sizes and write results to the output file
for block_size in 4 8 16 32 64;
do
  for ((rowsA = 1000, colsA=3000, colsB=2000; rowsA <= 10000; rowsA += 1000, colsA = 3*rowsA, colsB = 2*rowsA))
  do
            echo "Running matrix multiplication for rowsA=$rowsA, colsA=$colsA, colsB=$colsB, block_size=($block_size, $block_size
```

```
            # Run the compiled CUDA code with current matrix size and append results to the output file
            ./optimized_matrix_mult $rowsA $colsA $colsB $block_size>> $output_file

            echo "-------------------------------------------------------" >> $output_file
  done
done
```

7. **Event Recording**: Create CUDA events for measuring execution time.
8. **Kernel Launch**: Launch the CUDA kernel (`multiplyMatrices`) to perform matrix multiplication on the GPU.
9. **Event Synchronization and Timing**: Record and calculate the elapsed time for kernel execution.
10. **Copy Output Matrix**: Copy the result (`matrixC`) from the device to the host.
11. **Display Result Matrix**: Optionally, you can uncomment the `displayMatrix` function to print the result matrix.
12. **Memory Deallocation**: Free the allocated memory on both the host and the device.
13. **Return**: Exit the program with a return code of 0

Alternative to CUDA

For the purpose of testing my local machine, I translated the above code to OpenCL implementation and ran it on my machine utilizing my AMD RX 640 with 10GB total RAM (2gb clocked for VRAM and 8gb shared memory)

the following implementation was used but the results for the tests were far off comparison with the Nvidia Tf4 used in google-Colab so I am not going to include them in the results discussion section:

```
#include <CL/cl.h>
#include <stdio.h>
#include <stdlib.h>

// Display a 2D matrix
void displayMatrix(float *matrix, int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            printf("%f ", matrix[i * cols + j]);
        }
        printf("\n");
    }
}

int main()
{
    // Set matrix dimensions
    int rowsA = 10000;
    int colsA = 30000;
    int colsB = 20000;

    // Allocate memory on the host for input and output matrices
    float *matrixA, *matrixB, *matrixC;
    matrixA = (float *)malloc(rowsA * colsA * sizeof(float));
    matrixB = (float *)malloc(colsA * colsB * sizeof(float));
    matrixC = (float *)malloc(rowsA * colsB * sizeof(float));

    // Initialize input matrices with random values
    for (int i = 0; i < rowsA * colsA; i++)
        matrixA[i] = rand() / (float)RAND_MAX;
    for (int i = 0; i < colsA * colsB; i++)
        matrixB[i] = rand() / (float)RAND_MAX;

    // Set up OpenCL

    // Get the platform and device
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, NULL);

    cl_device_id device;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);

    // Create OpenCL context and command queue
    cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
    cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);
```

```
    // Create OpenCL buffers
    cl_mem bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * rowsA * colsA, matrixA, NULL);
    cl_mem bufferB = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * colsA * colsB, matrixB, NULL);
    cl_mem bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * rowsA * colsB, NULL, NULL);

    // Compile the OpenCL kernel
    const char *source = "#include <cl.cl>\n"; // Add the content of optimized_matrix_mult.cl here
    cl_program program = clCreateProgramWithSource(context, 1, &source, NULL, NULL);
    clBuildProgram(program, 1, &device, NULL, NULL, NULL);
    cl_kernel kernel = clCreateKernel(program, "multiplyMatrices", NULL);

    // Set kernel arguments
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufferA);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufferB);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufferC);
    clSetKernelArg(kernel, 3, sizeof(int), &rowsA);
    clSetKernelArg(kernel, 4, sizeof(int), &colsA);
    clSetKernelArg(kernel, 5, sizeof(int), &colsB);

    // Set global and local work sizes
    size_t globalSize[2] = {rowsA, colsB};
    size_t localSize[2] = {16, 16};

    // Execute the kernel
    clEnqueueNDRangeKernel(queue, kernel, 2, NULL, globalSize, localSize, 0, NULL, NULL);
    clFinish(queue);

    // Read the result back to the host
    clEnqueueReadBuffer(queue, bufferC, CL_TRUE, 0, sizeof(float) * rowsA * colsB, matrixC, 0, NULL, NULL);

    // Display the result matrix
    // displayMatrix(matrixC, rowsA, colsB);

    // Release OpenCL resources
    clReleaseMemObject(bufferA);
    clReleaseMemObject(bufferB);
    clReleaseMemObject(bufferC);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);

    // Free host memory
    free(matrixA);
    free(matrixB);
    free(matrixC);

    return 0;
}
_kernel void multiplyMatrices(__global const float *inputMatrixA, __global const float *inputMatrixB, __global float *outputMatrixC, in
{
    int row = get_global_id(0);
    int col = get_global_id(1);

    // Ensure that the thread is within the valid matrix dimensions
    if (row < rowsA && col < colsB)
    {
        float result = 0;

        // Perform dot product for the given element of the output matrix
        for (int i = 0; i < colsA; i++)
        {
            result += inputMatrixA[row * colsA + i] * inputMatrixB[i * colsB + col];
        }

        // Save the result to the output matrix
        outputMatrixC[row * colsB + col] = result;
    }
}
```

Results

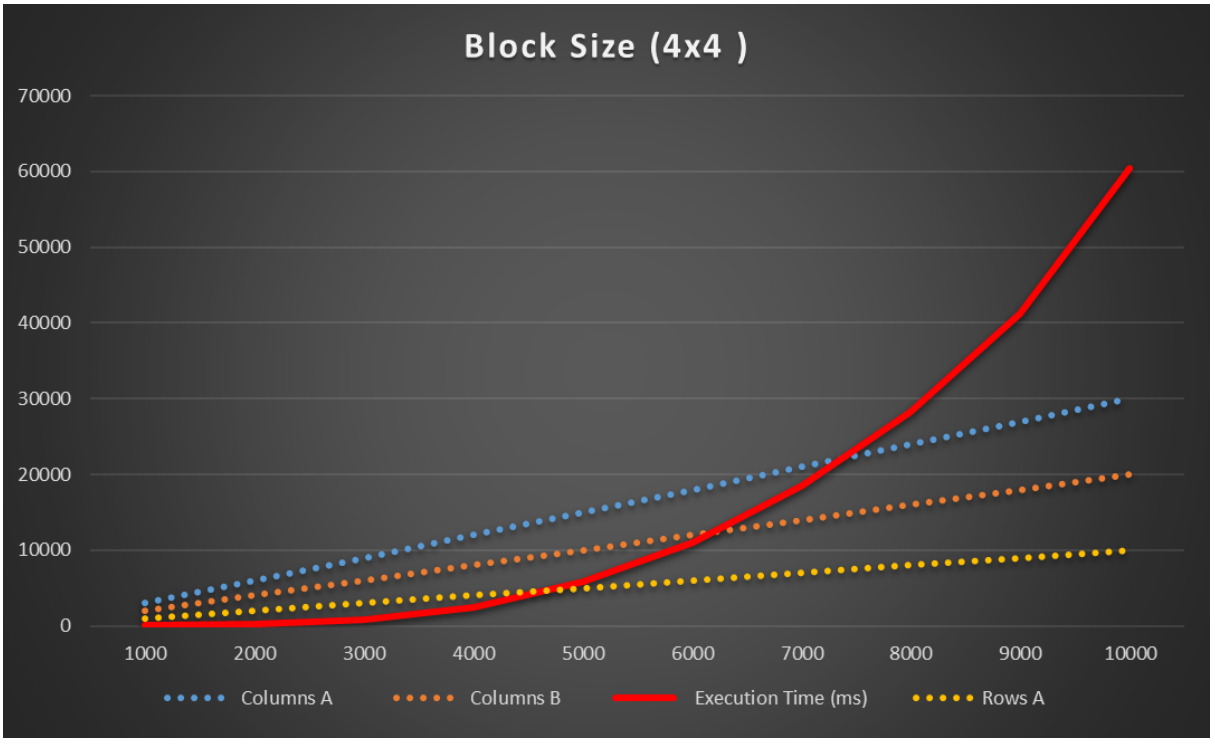
Below are the different execution time for different sizes of blocks and matrices:

Results Output file available on GitHub:

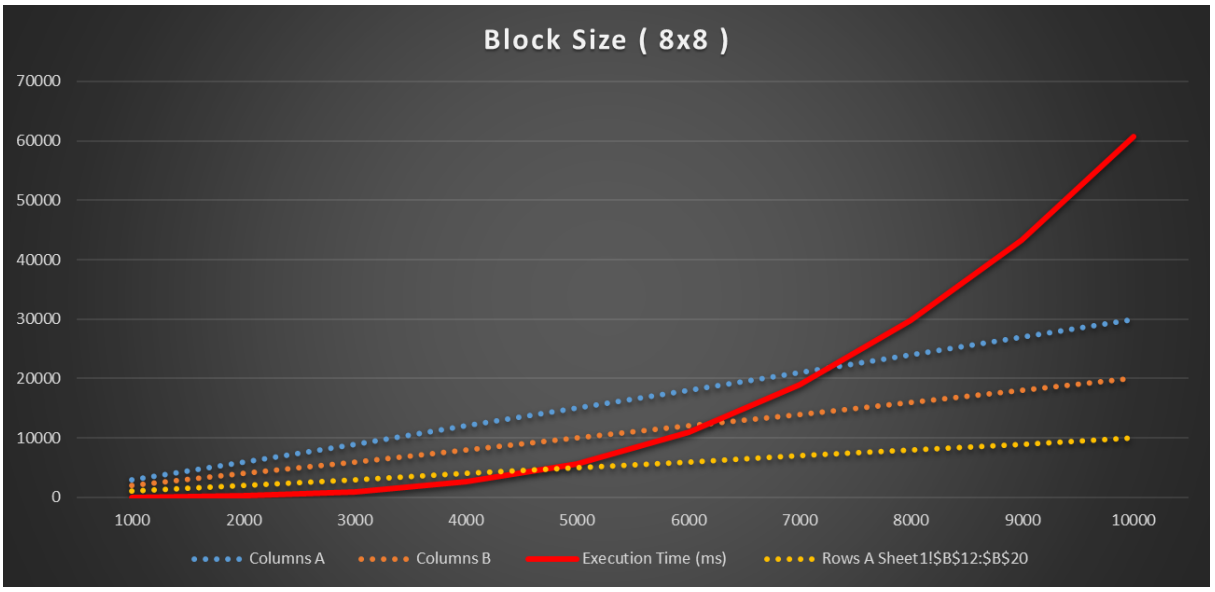
https://github.com/Anas-Albaqeri/Paralllelizing-Matrix-Multiplications-using-CUDA-/blob/main/Assignment%203%20-%20Parallelizing%20Matric%20Multiplication%20using%20Cuda/Standrad%20-%20no%20tiling/Results.txt

```
Running matrix multiplication for rowsA=6000, colsA=18000, colsB=12000, block_size=(4, 4)
Execution Time: 11001.480469 ms
-------------------------------------------------------
Running matrix multiplication for rowsA=7000, colsA=21000, colsB=14000, block_size=(4, 4)
Execution Time: 18592.324219 ms
-------------------------------------------------------
Running matrix multiplication for rowsA=8000, colsA=24000, colsB=16000, block_size=(4, 4)
Execution Time: 28231.667969 ms
-------------------------------------------------------
Running matrix multiplication for rowsA=9000, colsA=27000, colsB=18000, block_size=(4, 4)
Execution Time: 41293.171875 ms
-------------------------------------------------------
Running matrix multiplication for rowsA=10000, colsA=30000, colsB=20000, block_size=(4, 4)
Execution Time: 60423.898438 ms
-------------------------------------------------------
Running matrix multiplication for rowsA=1000, colsA=3000, colsB=2000, block_size=(8, 8)
Execution Time: 16.798656 ms
-------------------------------------------------------
Running matrix multiplication for rowsA=2000, colsA=6000, colsB=4000, block_size=(8, 8)
Execution Time: 215.801056 ms
-------------------------------------------------------
Running matrix multiplication for rowsA=3000, colsA=9000, colsB=6000, block_size=(8, 8)
Execution Time: 920.392517 ms
-------------------------------------------------------
Running matrix multiplication for rowsA=4000, colsA=12000, colsB=8000, block_size=(8, 8)
Execution Time: 2582.696289 ms
-------------------------------------------------------
Running matrix multiplication for rowsA=5000, colsA=15000, colsB=10000, block_size=(8, 8)
Execution Time: 5703.814453 ms
```

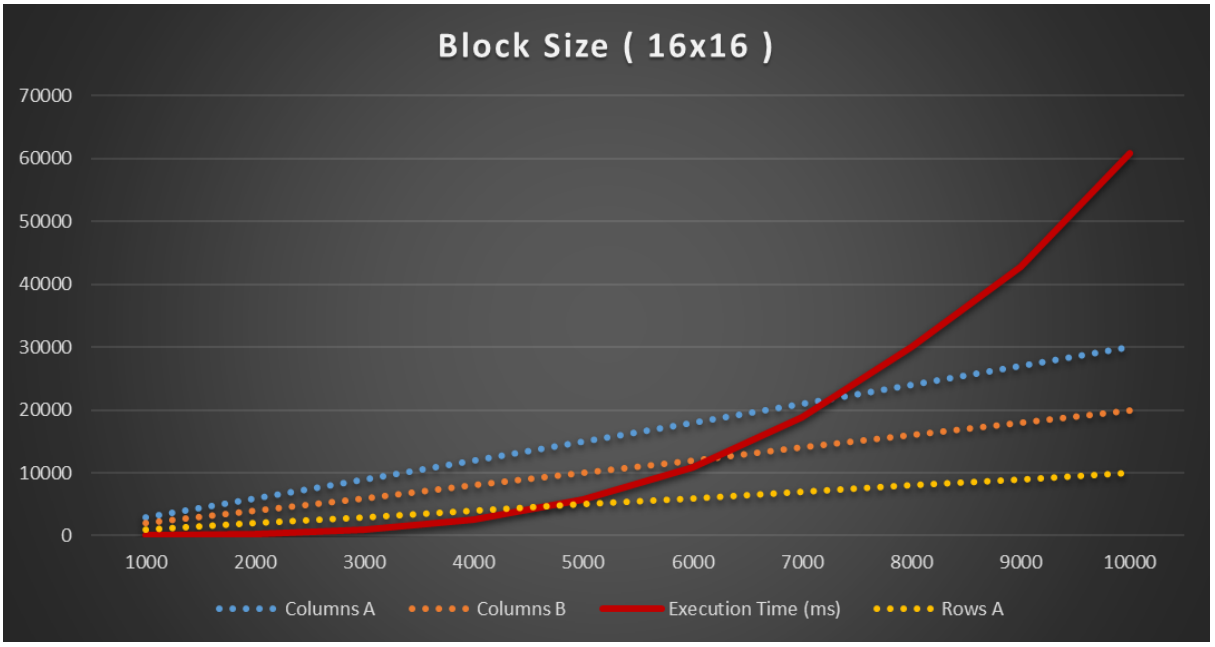
A. block size = 4 x 4, matrices sizes = (1000x2000 - 2000x3000) ———> (10000x20000 - 20000x30000)



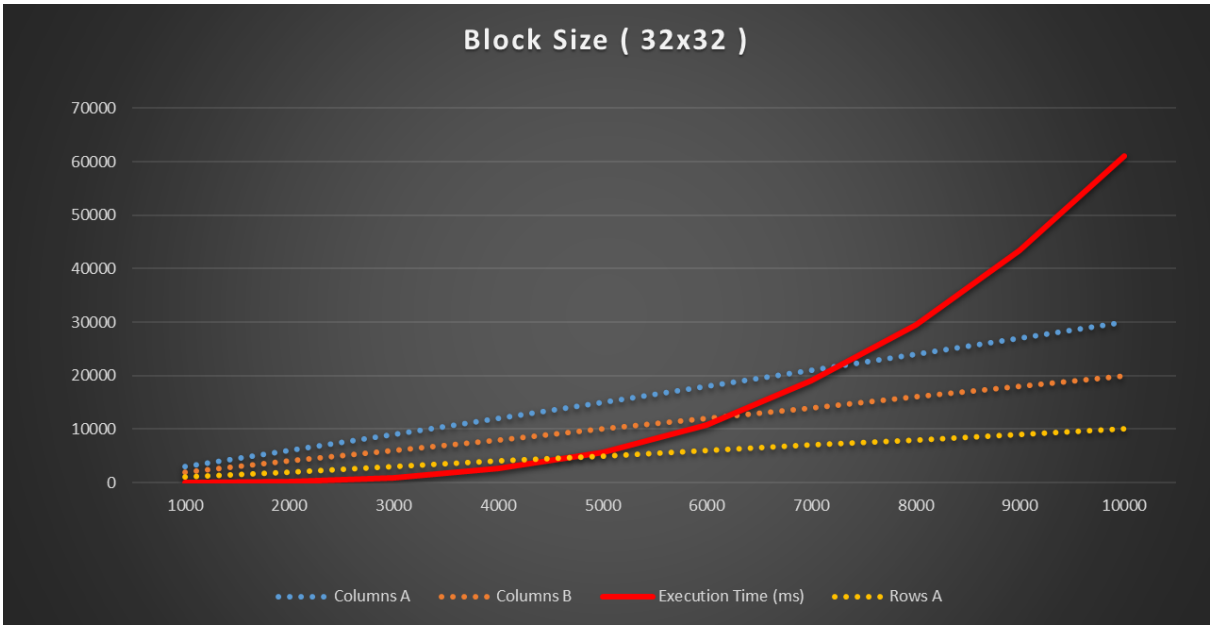
B. block size = 8 x 8, matrices sizes = (1000x2000 - 2000x3000) ———> (10000x20000 - 20000x30000)



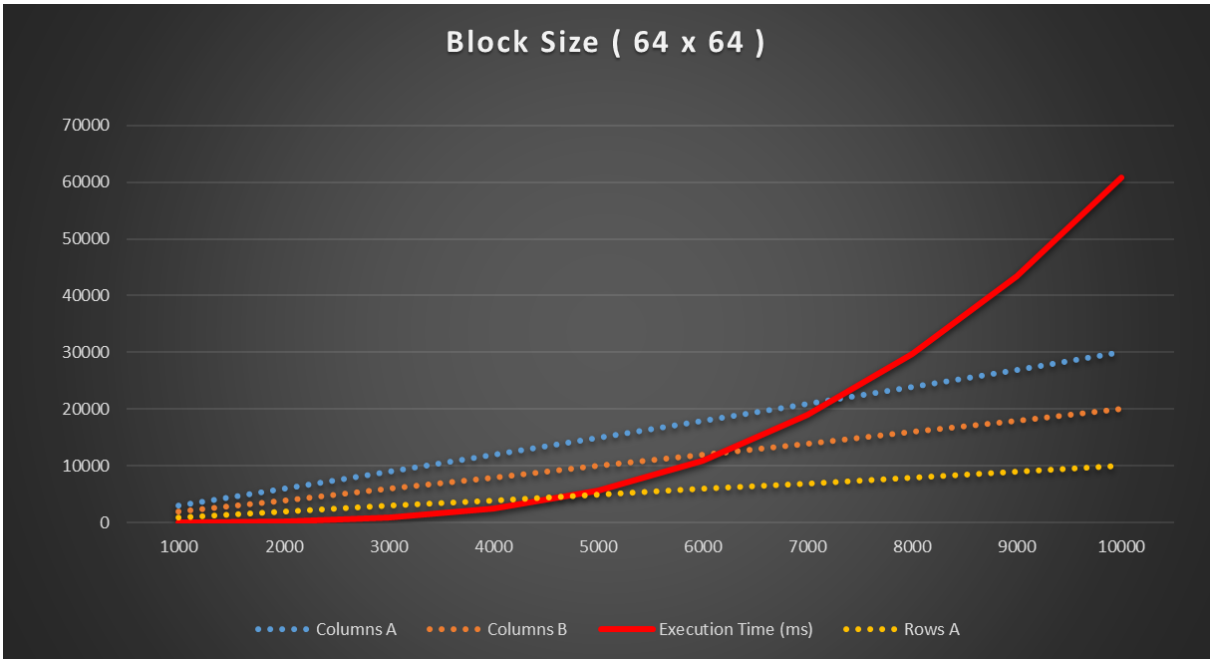
C. block size = 16 x 16, matrices sizes = (1000x2000 - 2000x3000) ———> (10000x20000 - 20000x30000)



D. block size = 32 x 32, matrices sizes = (1000x2000 - 2000x3000) ———> (10000x20000 - 20000x30000)



E. block size = 64 x 64, matrices sizes = (1000x2000 - 2000x3000) ———> (10000x20000 - 20000x30000)



We notice that all the previous results look almost identical indicating a rather low correlation between number of threads used in the test and performance acquired.

For the purpose of comparison and calculating best speed up possible we will choose block size 16x16 as it gave an average performance in comparison to 8x8 which had a slightly better performance and 4x4 which had a slightly worse performance.

It is also notable that as we increase the block size larger than the given window available to us: 1024, the performance is not notably affected at all and is clocked in at maximum window.

Performance Measures:

The ideal method here is to compare the best sequential time and parallel time using same machine . Therefore, I used my previous sequential code to run the test.

Please note that I also ran my sequential code locally on my machine and it took12 minutes for one specified matrix size (10000x20000, 20000x30000). Again my humble laptop's CPU is not of any comparison to Google's dedicated Tesla T4

running the test on these sizes (10000x20000, 20000x30000) we acquire this sequential time:

```
Running matrix multiplication for rowsA=8000, colsA=24000, colsB=16000, block_size=(1, 1)
Execution Time: 752,930.09756 ms
-------------------------------------------------------
```

Speedup and Efficiency:

CUDA - no tiling - block size 16x16

Rows A	Columns A	Columns B	Execution time (ms)
1000	3000	2000	22.374657
2000	6000	4000	215.156281
3000	9000	6000	910.1745
4000	12000	8000	2571.66333
5000	15000	10000	5695.79834
6000	18000	12000	10839.30078
7000	21000	14000	18818.23047
8000	24000	16000	30002.06445
9000	27000	18000	42715.10156
10000	30000	20000	60884.70313

Speedup Factor

for **matrices sizes = (1000x2000 - 2000x3000) ———> (10000x20000 - 20000x30000)**

$$S(P) = \frac{Best\ Sequential\ Time}{Best\ Parallel\ Time}$$

$$= \frac{752,930.09\dot{7}56}{60884.70313} = 12.366490$$

Efficiency

$$Efficiency = \frac{S(p)}{Number\ of\ threads} = 0.7729$$

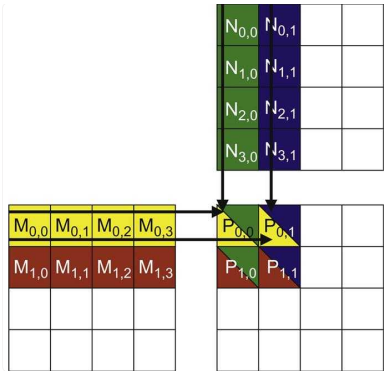
Matrix Multiplication with Tiling

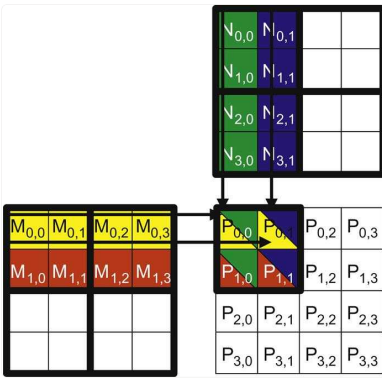
Introduction

First let us explain what tiling means in the context of matrix multiplication. Tiling, in the context of parallelizing matrix multiplications using CUDA, is a strategic approach to optimizing memory access and computational efficiency. The inherent tradeoff between the large but slow global memory and the fast but small shared memory in CUDA necessitates thoughtful memory management. Tiling addresses this challenge by partitioning the input matrices into subsets, or tiles, that fit into the faster shared memory. This becomes particularly crucial when the size of the data exceeds the capacity of the global memory. Each tile is chosen such that the kernel computation on these subsets can be independently performed, allowing for parallelization. The term "tile" is aptly used, likening the large wall of global memory data to be covered by smaller, manageable subsets.

In the context of matrix multiplication, tiling operates by breaking down the input matrices into smaller tiles, and each thread within a block collaboratively loads a subset of the matrices into the shared memory. This collaborative loading ensures that certain elements are only loaded once from the global memory, effectively reducing the overall number of accesses to global memory. The reduction in global memory traffic is proportional to the dimensions of the blocks used in tiling, providing a

scalable solution. Tiling is analogous to a carpooling arrangement, where threads act as commuters accessing data, and DRAM access requests act as vehicles. By having multiple threads accessing data from the same DRAM location, they form a "carpool," combining their accesses into one DRAM request and mitigating traffic congestion. Moreover, tiling introduces the concept of locality, focusing on smaller subsets of input matrix elements in each phase, allowing threads to collaboratively load and use shared memory efficiently. Overall, tiling proves to be a powerful technique for minimizing memory traffic, leveraging shared memory effectively, and optimizing the parallel execution of matrix multiplications in CUDA.





Code

A. Kernel Function

We employ a very similar kernel function with some slight adjustments to specify tiling. The following is a breakdown of the kernel function highlighting the differences between it and the previously kernel function used.

```
#define TILE_SIZE 16

__global__ void matrixMultiplication(const float *inputMatrixA, const float *inputMatrixB, float *outputMatrixC, int rowsA, int colsA,
    // ... (thread indexing)

    __shared__ float A_tile[TILE_SIZE][TILE_SIZE];
    __shared__ float B_tile[TILE_SIZE][TILE_SIZE];

    float result = 0;

    for (int tileIdx = 0; tileIdx < (colsA + TILE_SIZE - 1) / TILE_SIZE; tileIdx++) {
        // ... (load A_tile and B_tile from global memory to shared memory)

        __syncthreads();

        for (int i = 0; i < TILE_SIZE; i++) {
            result += A_tile[threadIdx.y][i] * B_tile[i][threadIdx.x];
        }

        __syncthreads();
    }

    if (row < rowsA && col < colsB) {
        outputMatrixC[row * colsB + col] = result;
    }
}
```

Thread Indexing:

- Both kernels use similar thread indexing to determine the current row and column.

Tiling:

- `matrixMultiplication`:
  - We Employ tiling using shared memory (`A_tile` and `B_tile`) where each thread loads a tile of the input matrices into shared memory. Moreover, we are now including synchronization barriers: (`__syncthreads()`) to ensure each block is accessing the correct designated data and consequentially performing the dot product using the tiles in shared memory.

The main function is almost the same with minor changes to accommodate tiles size.

Results

As usual, I used a script to run the test on different tile sizes and matrix sizes as follows:

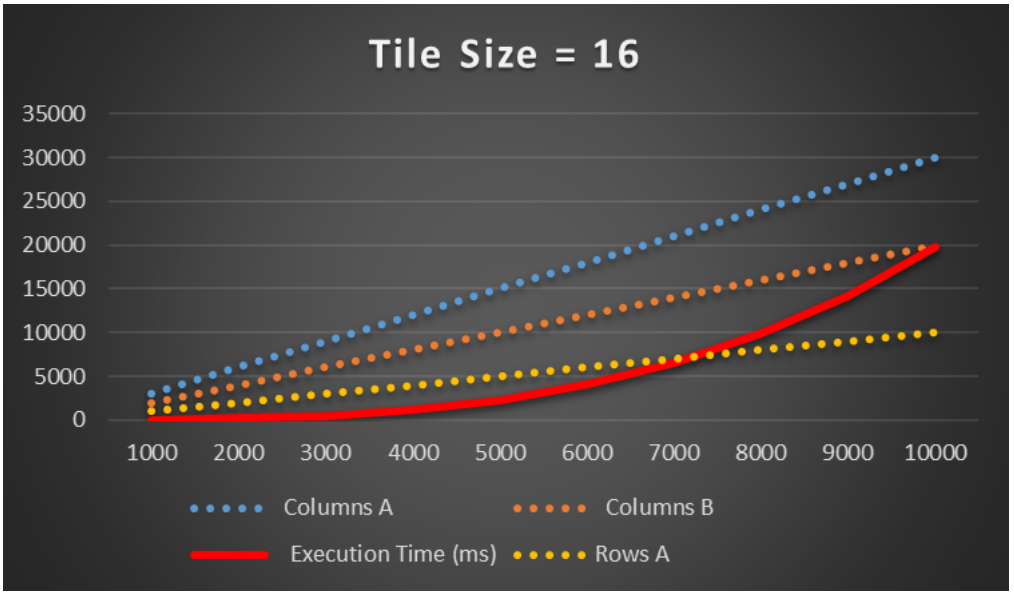
```
%%bash
#!/bin/bash

# Compile the CUDA code
!nvcc -o matrix_multi_tiling matrix_multi_tiling.cu

# Specify the output file
output_file="matrix_mult_tiling_results.txt"

# Loop through matrix sizes and write results to the output file
for tile_size in 4 8 16 32 64;
do
  for ((rowsA = 1000, colsA=3000, colsB=2000; rowsA <= 10000; rowsA += 1000, colsA = 3*rowsA, colsB = 2*rowsA))
  do
            echo "Running matrix multiplication for rowsA=$rowsA, colsA=$colsA, colsB=$colsB, tile=($tile_size, $tile_size)" >> $output

            # Run the compiled CUDA code with current matrix size and append results to the output file
            ./matrix_multi_tiling $rowsA $colsA $colsB $tile_size>> $output_file

            echo "-----------------------------------------------------" >> $output_file
  done
      echo"#############################################################">> $output_file
done
```

The results file is available on GitHub, here is a screenshot of some of the results acquired for tile size of 16

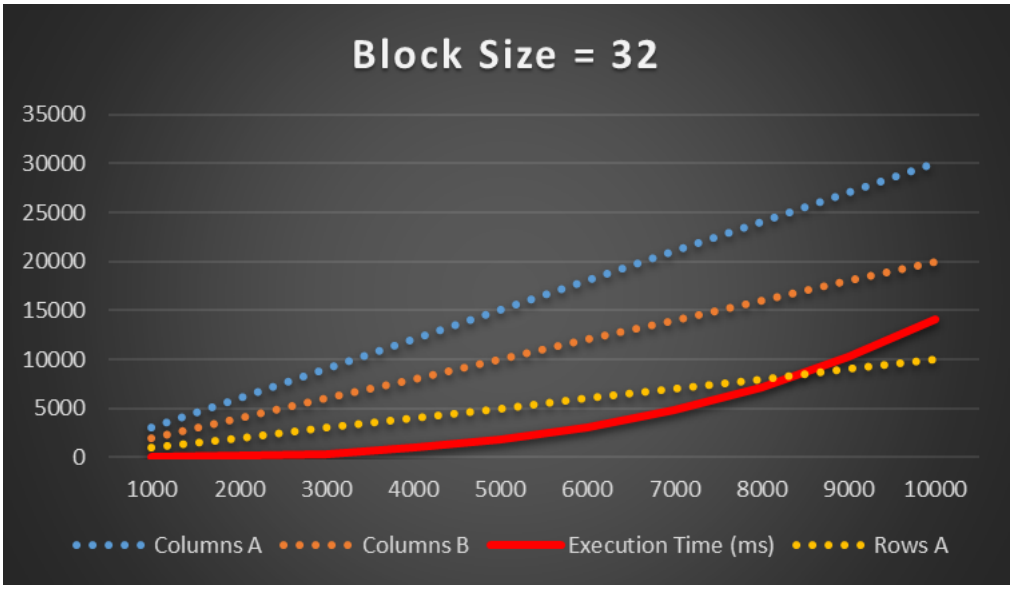
https://github.com/Anas-Albaqeri/Paralllelizing-Matrix-Multiplications-using-CUDA-/blob/main/Assignment%203%20-%20Parallelizing%20Matric%20Multiplication%20using%20Cuda/with%20tiling/results%20tile%20size%2016.txt

```
Running matrix multiplication for rowsA=2000, colsA=6000, colsB=4000
Execution Time: 203.555908 ms
---------------------------------------------------------
Running matrix multiplication for rowsA=3000, colsA=9000, colsB=6000
Execution Time: 463.899780 ms
---------------------------------------------------------
Running matrix multiplication for rowsA=4000, colsA=12000, colsB=8000
Execution Time: 1209.835693 ms
---------------------------------------------------------
Running matrix multiplication for rowsA=5000, colsA=15000, colsB=10000
Execution Time: 2329.417480 ms
---------------------------------------------------------
Running matrix multiplication for rowsA=6000, colsA=18000, colsB=12000
Execution Time: 4029.329590 ms
---------------------------------------------------------
Running matrix multiplication for rowsA=7000, colsA=21000, colsB=14000
Execution Time: 6469.350586 ms
---------------------------------------------------------
Running matrix multiplication for rowsA=8000, colsA=24000, colsB=16000
Execution Time: 9830.050781 ms
---------------------------------------------------------
Running matrix multiplication for rowsA=9000, colsA=27000, colsB=18000
Execution Time: 14203.209961 ms
---------------------------------------------------------
Running matrix multiplication for rowsA=10000, colsA=30000, colsB=20000
Execution Time: 19729.265625 ms
---------------------------------------------------------
```

A. tile size = 16 matrices sizes = (1000x2000 - 2000x3000) ———> (10000x20000 - 20000x30000)

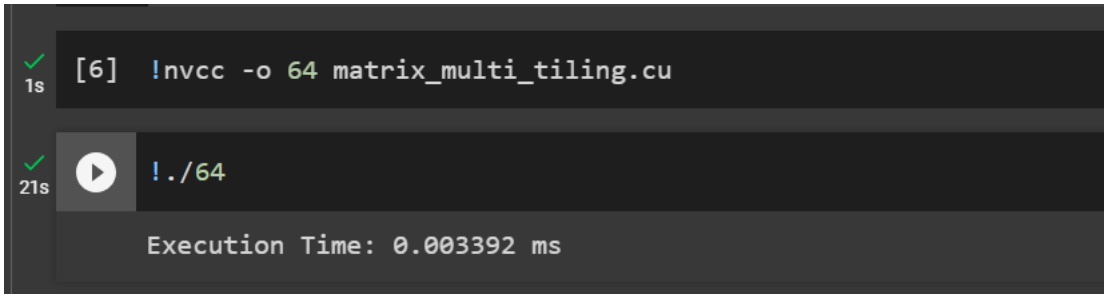


B. tile size = 32 matrices sizes = (1000x2000 - 2000x3000) ———> (10000x20000 - 20000x30000)



Note that tile size 64 was having calculation problems as for **matrices sizes =** **(1000x2000 - 2000x3000) ———> (10000x20000 - 20000x30000)**

it performed the calculations in sub 20 seconds which makes sense. However the recorder was not giving precise data at all: note on the left is the actual time it took to run the program in comparison to the execution time returned.



for that reason we will calculate the speed up and performance measures using tile size set to 16.

Performance Measures

Specs	rows A	Columns A	Columns B	Execution Time (ms)
Tile Size = 16	1000	3000	2000	32.078175
	2000	6000	4000	203.555908
	3000	9000	6000	463.89978
	4000	12000	8000	1209.835693
	5000	15000	10000	2329.41748
	6000	18000	12000	4029.32959
	7000	21000	14000	6469.350586
	8000	24000	16000	9830.050781
	9000	27000	18000	14203.20996
	10000	30000	20000	19729.26563
Block Size = 32 x 32	1000	3000	2000	30.751713
	2000	6000	4000	197.392838
	3000	9000	6000	378.874084

	4000	12000	8000	949.954468
	5000	15000	10000	1826.546875
	6000	18000	12000	3042.122559
	7000	21000	14000	4860.96875
	8000	24000	16000	7101.797852
	9000	27000	18000	10327.92481
	10000	30000	20000	14138.41406

Speedup Factor

for **matrices sizes = (1000x2000 - 2000x3000) —> (10000x20000 - 20000x30000) , tile size = 16**

$$S(P) = \frac{Best\ Sequential\ Time}{Best\ Parallel\ Time}$$

$$= \frac{752,930.09756}{19729.26563} = 38.16310812983869$$

Speedup over regular CUDA

$$S(P) = \frac{60884.70313}{19729.26563} = 3.086009$$

Efficiency

$$Efficiency = \frac{S(p)}{Number\ of\ threads} = 2.38519425$$

Discussion

Scalability

Scalability is an important metric to evaluate the performance of parallel computations. It is a good measure of how efficiently we are computing over larger chunks of data.

1. without tiling

- As the matrix size increases, we notice that the execution time also increases significantly.
- The speedup also notably diminishes for larger matrices.
- Efficiency was also observed to decrease as the number of threads increased, indicating challenges in scaling.

all these observations indicate that without tiling, the scalability of the CUDA matrix multiplication implementation is limited, especially for larger matrices as the overhead with managing a larger number of threads becomes more apparent, impacting scalability severely.

2. without tiling

- Tiling leads to a notable reduction in execution time across all matrix sizes used. The speedup is consistent in comparison to the previous implementation showcasing better scalability. Moreover, efficiency improves with increased number of threads also indicating further improved scalability.

Conclusion

The experiments compared CUDA matrix multiplication with and without tiling. Without tiling, execution time rises notably with matrix size, showing challenges with larger datasets. Tiling significantly reduces execution time, especially effective for larger matrices. Without tiling, speedup diminishes for larger matrices, while tiling consistently provides higher speedup. Efficiency without tiling decreases with more threads, indicating diminishing returns, whereas tiling significantly improves efficiency. Tiling proves effective for large matrices, emphasizing its benefits. Diminishing returns without tiling highlight the need for optimized

strategies. Recommendations include considering tiling for large matrices, exploring dynamic thread management, experimenting with additional optimizations, and acknowledging the structured thread management advantage provided by tiling.