1. **Code Listing (both tiling and basic CUDA kernel)**

#include <iostream>

#include <cuda\_runtime.h>

#include <cstdlib>

#include <ctime>

#define TILE\_WIDTH 32

using namespace std;

\_\_global\_\_ void matrixBasicCUDA(int \*A, int \*B, int \*C, int m, int p, int n) {

//get row, col index to operate on C matrix

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

if (row < m && col < n) {

int sum = 0.0;

for (int k = 0; k < p; k++) {

sum += A[row \* p + k] \* B[k \* n + col];

}

C[row \* n + col] = sum;

}

}

\_\_global\_\_ void matrixTiledMultiply(int \*A, int \*B, int \*C, int m, int p, int n) {

\_\_shared\_\_ int aTile[TILE\_WIDTH][TILE\_WIDTH];

\_\_shared\_\_ int bTile[TILE\_WIDTH][TILE\_WIDTH];

int tx = threadIdx.x, ty = threadIdx.y;

int row = blockIdx.y \* blockDim.y + ty;

int col = blockIdx.x \* blockDim.x + tx;

int sum = 0.0;

for (int i = 0; i < (p - 1) / TILE\_WIDTH + 1; i++) {

if (row < m && i \* TILE\_WIDTH + tx < p) {

aTile[ty][tx] = A[row \* p + i \* TILE\_WIDTH + tx];

} else {

aTile[ty][tx] = 0.0;

}

if (col < n && i \* TILE\_WIDTH + ty < p) {

bTile[ty][tx] = B[(i \* TILE\_WIDTH + ty) \* n + col];

} else {

bTile[ty][tx] = 0.0;

}

\_\_syncthreads();

for (int k = 0; k < TILE\_WIDTH; k++) {

sum += aTile[ty][k] \* bTile[k][tx];

}

\_\_syncthreads();

}

if (row < m && col < n) {

C[row \* n + col] = sum;

}

}

void initializeMatrix(int \*matrix, int size) {

for (int i = 0; i < size; i++) {

matrix[i] = static\_cast<int>(rand()) / 20;

}

}

int main(int argc, char \*\*argv) {

int m = stoi(argv[1]);

int p = stoi(argv[2]);

int n = m;

int \*hA = new int[m \* p];

int \*hB = new int[p \* n];

int \*hC\_basic = new int[m \* n];

int \*hC\_tiled = new int[m \* n];

srand(time(0));

initializeMatrix(hA, m \* p);

initializeMatrix(hB, p \* n);

int \*dA, \*dB, \*dC;

cudaMalloc(&dA, sizeof(int) \* m \* p);

cudaMalloc(&dB, sizeof(int) \* p \* n);

cudaMalloc(&dC, sizeof(int) \* m \* n);

cudaMemcpy(dA, hA, sizeof(int) \* m \* p, cudaMemcpyHostToDevice);

cudaMemcpy(dB, hB, sizeof(int) \* p \* n, cudaMemcpyHostToDevice);

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH, 1);

dim3 dimGrid((n - 1) / TILE\_WIDTH + 1, (m - 1) / TILE\_WIDTH + 1, 1);

cudaEvent\_t start, stop;

float time\_basic, time\_tiled;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start);

matrixBasicCUDA<<<dimGrid, dimBlock>>>(dA, dB, dC, m, p, n);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time\_basic, start, stop);

cudaMemcpy(hC\_basic, dC, sizeof(int) \* m \* n, cudaMemcpyDeviceToHost);

cudaEventRecord(start);

matrixTiledMultiply<<<dimGrid, dimBlock>>>(dA, dB, dC, m, p, n);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time\_tiled, start, stop);

cudaMemcpy(hC\_tiled, dC, sizeof(int) \* m \* n, cudaMemcpyDeviceToHost);

cout << "Matrix Size: " <<" m: " << m << " p: " << p << " n: " << n << endl;

cout << "Basic Kernel Time: " << time\_basic << " ms" << endl;

cout << "Tiled Kernel Time: " << time\_tiled << " ms" << endl;

cudaFree(dA);

cudaFree(dB);

cudaFree(dC);

delete[] hA;

delete[] hB;

delete[] hC\_basic;

delete[] hC\_tiled;

return 0;

}

**Console Log –**

Matrix Size: m: 32 p: 1024 n: 32

Basic Kernel Time: 4.21578 ms

Tiled Kernel Time: 0.08704 ms

Matrix Size: m: 64 p: 1024 n: 64

Basic Kernel Time: 3.65731 ms

Tiled Kernel Time: 0.086816 ms

Matrix Size: m: 128 p: 1024 n: 128

Basic Kernel Time: 3.53472 ms

Tiled Kernel Time: 0.074272 ms

Matrix Size: m: 256 p: 1024 n: 256

Basic Kernel Time: 3.97094 ms

Tiled Kernel Time: 0.077248 ms

Matrix Size: m: 512 p: 1024 n: 512

Basic Kernel Time: 3.61437 ms

Tiled Kernel Time: 0.155616 ms

Matrix Size: m: 1024 p: 1024 n: 1024

Basic Kernel Time: 3.48429 ms

Tiled Kernel Time: 0.441504 ms

Matrix Size: m: 2048 p: 1024 n: 2048

Basic Kernel Time: 7.05469 ms

Tiled Kernel Time: 1.62266 ms

Matrix Size: m: 4096 p: 1024 n: 4096

Basic Kernel Time: 12.8776 ms

Tiled Kernel Time: 6.36195 ms

Matrix Size: m: 8192 p: 1024 n: 8192

Basic Kernel Time: 53.8089 ms

Tiled Kernel Time: 29.9575 ms

Matrix Size: m: 16384 p: 1024 n: 16384

Basic Kernel Time: 186.941 ms

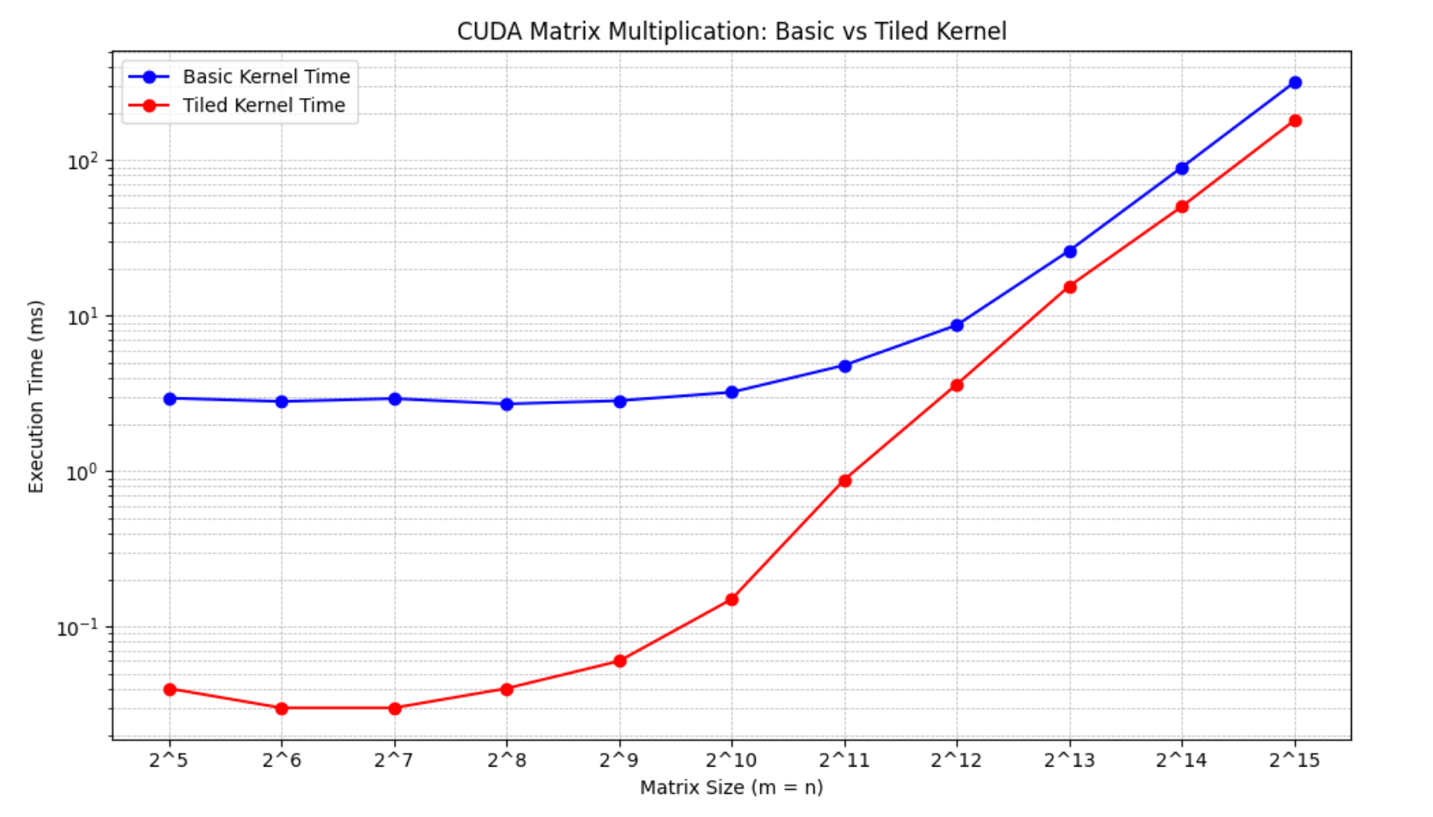
Tiled Kernel Time: 104.428 ms

Matrix Size: m: 32768 p: 1024 n: 32768

Basic Kernel Time: 728.779 ms

Tiled Kernel Time: 419.721 ms

1. Timing Plot



1. Discussion

As expected, the tiling mechanism is **better for all dimensions** of m,n (2^5 – 2^15) with p fixed at 1024. I was a little surprised to observe that the **speedup (Tiling / basic CUDA kernel) decreased as matrix size increased**. I expected the tiling mechanism to perform significantly better as we increased matrix size but I have 2 reasons why this may be happening –

1. **For small matrices – (2^5 to 2^9) the basic CUDA kernel is dominated by overhead**. Basically, this is too small of a matrix to see any observable gains through a GPU. The tiling mechanism on the other hand uses a shared memory, which significantly reduces memory access time. This makes the speedup comparison look enormous as the CUDA kernel is simple inefficient for the given matrix sizes.
2. **As matrix size increases exponentially – 2^11 to 2^15 – the overhead becomes negligible**. These are the perfect conditions in which GPU thrives on, therefore even though there is a considerable speedup by using the tiling mechanism, the difference is shrinking. Additionally, there is a slight disadvantage for the tiling mechanism in the aspect of copying data and synchronizing all threads. This may add to overall runtime, further shrinking the speedup ratio.

Overall, this was a great assignment which showed how a tiling mechanism is better than a basic CUDA kernel for matrix operations.