Distributed and Parallel Computing Lab

CS461 Lab8

Name: Dipean Dasgupta **ID:202151188**

Task: Implementation of matrix multiplication using parallel programming.

Application: Microsoft Visual Studio

Architecture: CUDA (Compute Unified Device Architecture)

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <omp.h>
#define BLOCK_SIZE 16
// CUDA kernel for general matrix multiplication
__global__ void gpu_matrix_mult(int* a, int* b, int* c, int m, int n, int k) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if (col < k && row < m)
    {
        for (int i = 0; i < n; i++)
        {
            sum += a[row * n + i] * b[i * k + col];
        }
        c[row * k + col] = sum;
    }
}
// CUDA kernel for square matrix multiplication
__global__ void gpu_square_matrix_mult(int* d_a, int* d_b, int* d_result, int n) {
    __shared__ int tile_a[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int tile_b[BLOCK_SIZE][BLOCK_SIZE];
    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int tmp = 0;
    int idx;
    for (int sub = 0; sub < gridDim.x; ++sub)  {
        idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
        if (idx >= n * n) {
            tile_a[threadIdx.y][threadIdx.x] = 0;
```

```
        } else {
            tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
        }
        idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
        if (idx >= n * n)  {
            tile_b[threadIdx.y][threadIdx.x] = 0;
        } else {
            tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
        }
        __syncthreads();
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            tmp += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
        }
        __syncthreads();
    }
    if (row < n && col < n)  {
        d_result[row * n + col] = tmp;
    }
}
// OpenMP function for matrix multiplication (parallelized)
void openmp_matrix_mult(int* h_a, int* h_b, int* h_c, int m, int n, int k) {
#pragma omp parallel for collapse(2)
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < k; ++j)  {
            int tmp = 0;
            for (int h = 0; h < n; ++h) {
                tmp += h_a[i * n + h] * h_b[h * k + j];
            }
            h_c[i * k + j] = tmp;
        }
    }
}
// Normal (sequential) matrix multiplication function
void cpu_matrix_mult(int* h_a, int* h_b, int* h_result, int m, int n, int k) {
    for (int i = 0; i < m; ++i){
        for (int j = 0; j < k; ++j){
            int tmp = 0;
            for (int h = 0; h < n; ++h) {
                tmp += h_a[i * n + h] * h_b[h * k + j];
            }
            h_result[i * k + j] = tmp;
        }
    }
}
// Main function
```

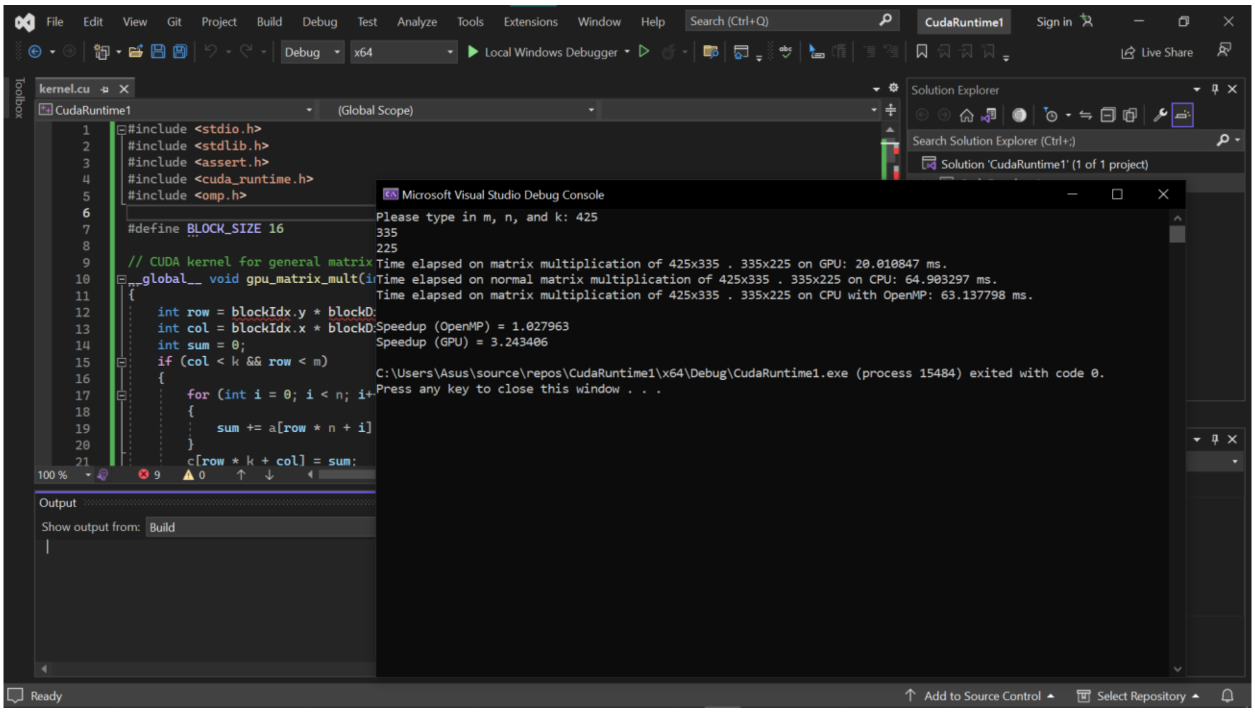
```
int main(int argc, char const* argv[]) {
    int m, n, k;
    srand(3333); // Fixed seed
    printf("Please type in m, n, and k: ");
    scanf("%d %d %d", &m, &n, &k);
    // Allocate memory in host RAM
    int* h_a, * h_b, * h_c, * h_cc;
    cudaMallocHost((void**)&h_a, sizeof(int) * m * n);
    cudaMallocHost((void**)&h_b, sizeof(int) * n * k);
    cudaMallocHost((void**)&h_c, sizeof(int) * m * k);
    cudaMallocHost((void**)&h_cc, sizeof(int) * m * k);
    // Random initialize matrix A
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            h_a[i * n + j] = rand() % 1024;
        }
    }
    // Random initialize matrix B
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < k; ++j) {
            h_b[i * k + j] = rand() % 1024;
        }
    }
   float gpu_elapsed_time_ms, cpu_elapsed_time_ms, normal_elapsed_time_ms;
    // Start measuring GPU execution time
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
    // Allocate memory space on the device
    int* d_a, * d_b, * d_c;
    cudaMalloc((void**)&d_a, sizeof(int) * m * n);
    cudaMalloc((void**)&d_b, sizeof(int) * n * k);
    cudaMalloc((void**)&d_c, sizeof(int) * m * k);
    // Copy matrix A and B from host to device memory
    cudaMemcpy(d_a, h_a, sizeof(int) * m * n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, sizeof(int) * n * k, cudaMemcpyHostToDevice);
    unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
    unsigned int grid_cols = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
    dim3 dimGrid(grid_cols, grid_rows);
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    // Launch the appropriate kernel
    if (m == n && n == k) {
        gpu_square_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, n);
    } else {
```

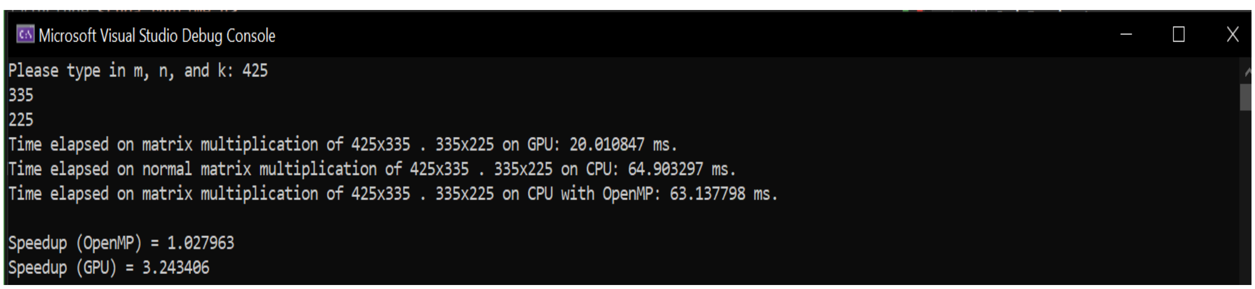
```
        gpu_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, m, n, k);
    }
    // Transfer results from device to host
    cudaMemcpy(h_c, d_c, sizeof(int) * m * k, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize(); // Wait for GPU to finish
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    // Compute time elapsed on GPU computing
    cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on GPU: %f ms.\n", m, n, n, k, gpu_elapsed_time_ms);
    // Start measuring normal (sequential) execution time
    double start_time = omp_get_wtime();
    cpu_matrix_mult(h_a, h_b, h_cc, m, n, k);
    double end_time = omp_get_wtime();
    normal_elapsed_time_ms = (end_time - start_time) * 1000.0; // Convert to milliseconds
    printf("Time elapsed on normal matrix multiplication of %dx%d . %dx%d on CPU: %f ms.\n", m, n, n, k,
normal_elapsed_time_ms);
    // Start measuring CPU execution time using OpenMP
    start_time = omp_get_wtime();
    openmp_matrix_mult(h_a, h_b, h_cc, m, n, k);
    end_time = omp_get_wtime();
    cpu_elapsed_time_ms = (end_time - start_time) * 1000.0; // Convert to milliseconds
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on CPU with OpenMP: %f ms.\n", m, n, n, k,
cpu_elapsed_time_ms);

    // Validate results computed by GPU
    int all_ok = 1;
    for (int i = 0; i < m; ++i){
        for (int j = 0; j < k; ++j){
            if (h_cc[i * k + j] != h_c[i * k + j]){
                all_ok = 0;
            }
        }
    }
    // Roughly compute speedup
    if (all_ok) {
        printf("\nSpeedup (OpenMP) = %f\n", normal_elapsed_time_ms / cpu_elapsed_time_ms);
        printf("Speedup (GPU) = %f\n", normal_elapsed_time_ms / gpu_elapsed_time_ms);
    } else {
        printf("Incorrect results\n");
    }    // Free memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFreeHost(h_a);
```

```
    cudaFreeHost(h_b);
    cudaFreeHost(h_c);
    cudaFreeHost(h_cc);
    return 0;
}
```

OUTPUT





So, here Mat A =425x335; MatB=335x225. Resultant MatC=425x225.

Time elapsed in calculation on GPU is far less than that of CPU or CPU with OpenMP.

---END of ASSIGNMENT--