ASSIGNMENT NO 4

Title of the Assignment:

Design & implementation of Parallel (CUDA) algorithm to Add two large Vector

Problem Statement:

Design & implementation of Parallel (CUDA) algorithm to Add two large Vector, Multiply Vector and Matrix and Multiply two N × N arrays using n2.

Objective:

To offload parallel computations to the graphics card, when it is appropriate to do so, and to give some idea of how to think about code running in the massively parallel environment presented by today's graphics cards.

Outcome:

Students should understand the basic of GPU computing in the CUDA environment.

Prerequisites:

Strassen's Matrix Multiplication Algorithm and CUDA Tutorials.

Hardware Specification:

x86_64 bit, 2 – 2/4 GB DDR RAM, 80 - 500 GB SATA HD, 1GBNIDIA TITAN X Graphics Card.

Software Specification:

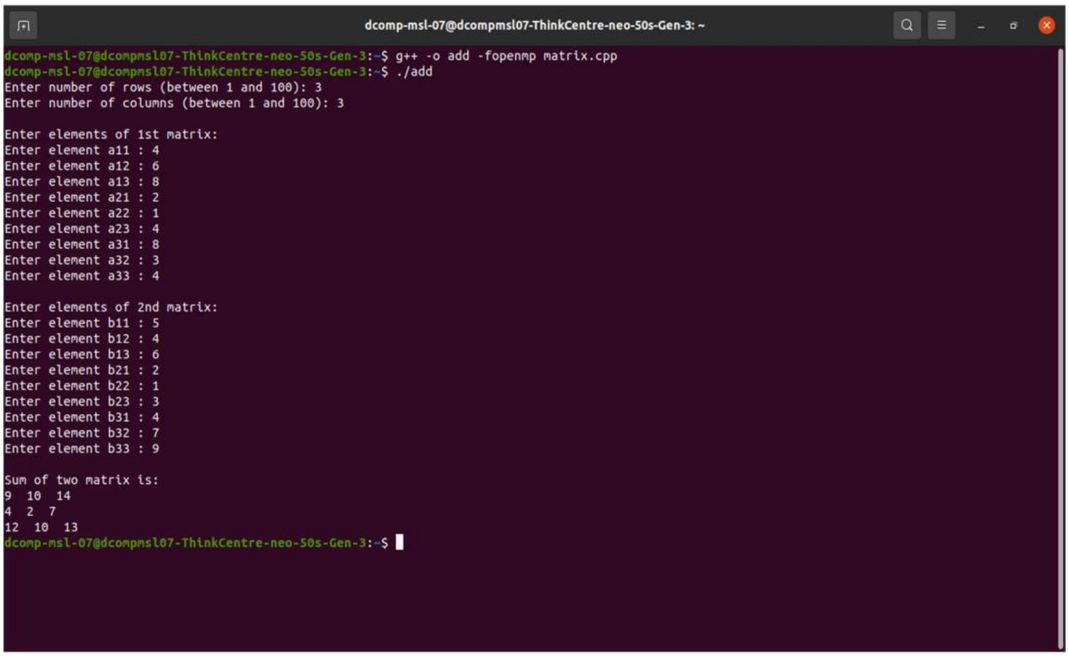
Ubuntu 14.04, GPU Driver 352.68, CUDA Toolkit 8.0, CUDNNLibrary v5.0

Source Code:

```
#include<iostream>
#include<bits/stdc++.h>
#include<cuda.h>
#define BLOCK_SIZE 16
using namespace std;
void initialize_matrix(int *array, int rows, int cols){
for(int i = 0 ; i < rows; i++){
for(int j = 0; j < cols; j++){
array[i*cols + j] = rand() % 10;
}
}
}
void print_matrix(int *array, int rows, int cols){
for(int i = 0 ; i < rows; i++){
for(int j = 0; j < cols; j++){
cout << array[i*cols + j] << " ";
}
cout << endl;
}
}
void matrix_multiplication_cpu(int *a, int *b, int *c, int common, int c_rows,int
c_cols){
for(int i = 0; i < c_rows; i++){
for(int j = 0; j < c_cols; j++){
int sum = 0;
for(int k = 0; k < common; k++){
sum += a[i*common + k] * b[k*c_cols + j];
}
c[i*c_cols + j] = sum;
}
}
}
__global__ void matrix_multiply(int *a, int *b, int *c, int c_rows, int common,
int c_cols)
{
int row = blockIdx.y*blockDim.y + threadIdx.y;
int col = blockIdx.x*blockDim.x + threadIdx.x;
int sum=0;
if(col < c_cols && row < c_rows) {
```

```
for(int j = 0 ;j < common;j++)
{
sum += a[row*common+j] * b[j*c_cols+col];
}
c[c_cols*row+col]=sum;
}
}
int main(){
int A_rows, A_cols, B_rows, B_cols, C_rows, C_cols;
cout << "Dimensions of matrix 1:\n";
cout << "Rows: ";
cin >> A_rows;
cout << "Columns: ";
cin >> A_cols;
cout << "Dimensions of matrix 2:\n";
cout << "Rows: " << A_cols << endl << "Columns: ";
cin >> B_cols;
B_rows = A_cols;
C_rows = A_rows;
C_cols = B_cols;
int A_size = A_rows * A_cols;
int B_size = B_rows * B_cols;
int C_size = C_rows * C_cols;
int *A, *B, *C;
int *m1,*m2,*result;
A = new int[A_size];
B = new int[B_size];
C = new int[C_size];
initialize_matrix(A,A_rows,A_cols);
cout << "Matrix 1\n";
print_matrix(A,A_rows,A_cols);
initialize_matrix(B,B_rows,B_cols);
cout << "Matrix 2\n";
print_matrix(B,B_rows,B_cols);
cudaMallocManaged(&m1, A_size * sizeof(int));
cudaMallocManaged(&m2, B_size * sizeof(int));
cudaMallocManaged(&result, C_size * sizeof(int));
cudaMemcpy(m1,A,A_size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(m2,B,B_size * sizeof(int), cudaMemcpyHostToDevice);
dim3 dimGrid(A_rows + BLOCK_SIZE - 1 / BLOCK_SIZE, B_cols +
BLOCK_SIZE - 1 / BLOCK_SIZE);
dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
float gpu_elapsed_time;
cudaEvent_t gpu_start,gpu_stop;
cudaEventCreate(&gpu_start);
cudaEventCreate(&gpu_stop);
cudaEventRecord(gpu_start);
matrix_multiply<<<dimGrid,dimBlock>>>(m1,m2,result,C_rows,A_cols,C_cols
);
cudaEventRecord(gpu_stop);
cudaEventSynchronize(gpu_stop);
cudaEventElapsedTime(&gpu_elapsed_time, gpu_start, gpu_stop);
cudaEventDestroy(gpu_start);
cudaEventDestroy(gpu_stop);
cudaMemcpy(C,result,C_size*sizeof(int),cudaMemcpyDeviceToHost);
cout << "GPU result:\n";
print_matrix(C,C_rows,C_cols);
cout<<"GPU Elapsed time is: "<<gpu_elapsed_time<<" milliseconds"<<endl;
cudaEventCreate(&gpu_start);
cudaEventCreate(&gpu_stop);
cudaEventRecord(gpu_start);
matrix_multiplication_cpu(A,B,C,A_cols,C_rows,C_cols);
cudaEventRecord(gpu_stop);
cudaEventSynchronize(gpu_stop);
cudaEventElapsedTime(&gpu_elapsed_time, gpu_start, gpu_stop);
cudaEventDestroy(gpu_start);
cudaEventDestroy(gpu_stop);
cout << "CPU result:\n";
print_matrix(C,C_rows,C_cols);
cout<<"CPU Elapsed time is: "<<gpu_elapsed_time<<" milliseconds"<<endl;
cudaFree(m1);
cudaFree(m2);
cudaFree(result);
return 0;
}
```

OUTPUT:



```
dcomp-msl-07@dcompmsl07-ThinkCentre-neo-50s-Gen-3:~$ g++ -o add -fopenmp matrix.cpp
dcomp-msl-07@dcompmsl07-ThinkCentre-neo-50s-Gen-3:~$ ./add
Enter number of rows (between 1 and 100): 3
Enter number of columns (between 1 and 100): 3

Enter elements of 1st matrix:
Enter element a11 : 4
Enter element a12 : 6
Enter element a13 : 8
Enter element a21 : 2
Enter element a22 : 1
Enter element a23 : 4
Enter element a31 : 8
Enter element a32 : 3
Enter element a33 : 4

Enter elements of 2nd matrix:
Enter element b11 : 5
Enter element b12 : 4
Enter element b13 : 6
Enter element b21 : 2
Enter element b22 : 1
Enter element b23 : 3
Enter element b31 : 4
Enter element b32 : 7
Enter element b33 : 9

Sum of two matrix is:
9  10  14
4  2  7
12  10  13
dcomp-msl-07@dcompmsl07-ThinkCentre-neo-50s-Gen-3:~$ 
```

Conclusion:

We have successfully implemented of Parallel (CUDA) algorithm to Add two large Vector