LP1
High Performance Computing
Assignment 2.

Date of Completion:- 26.8.2020
Title:- Vector and Matrix operation using CUDA.

Problem Statement:- Design parallel algorithm to.
1) Add two large vectors
2) Multiply Vector and Matrix
3) Multiply two N x N arrays using n^2 processors.

Objective :- Learn CUDA architecture & programming concepts.

Outcomes :- Use CUDA programming concepts to perform operation on vector and matrix.

Requirements :- Ubuntu, NVCC compiler, google Colab (if NVIDIA GPU is not available) NVIDIA GPU.

Theory:-
CUDA architecture:-
CUDA → Compute Unified Device Architecture

Host and Device
CPU → Host
GPU → Device

Kernel →

function to be executed on GPU.
prefixed with __global__

eg
```
__global__ void add(int *a, int *b)
{

}.
```

Thread

Single instance of execution.

Block

A group of threads.

Grid

A group of blocks.

Architecture Diagram

Applications — Applications — Applications using CUDA driver API — Applications

DirectX Compute — OpenGL Driver — C Runtime for CUDA

CUDA Driver — PTX (ISA)

CUDA support for OS Kernel

CUDA parallel Compute engine inside NVIDIA GPUs

Matrix Matrix Multiplication

Matrix 1

4	7	8	6
4	6	7	3
10	2	3	8
1	10	4	7
1	7	3	7

Matrix 2

2	9	8
10	3	1
3	4	8
6	10	3

Result

138	149	121
107	112	103
97	188	130
156	125	71
123	112	60

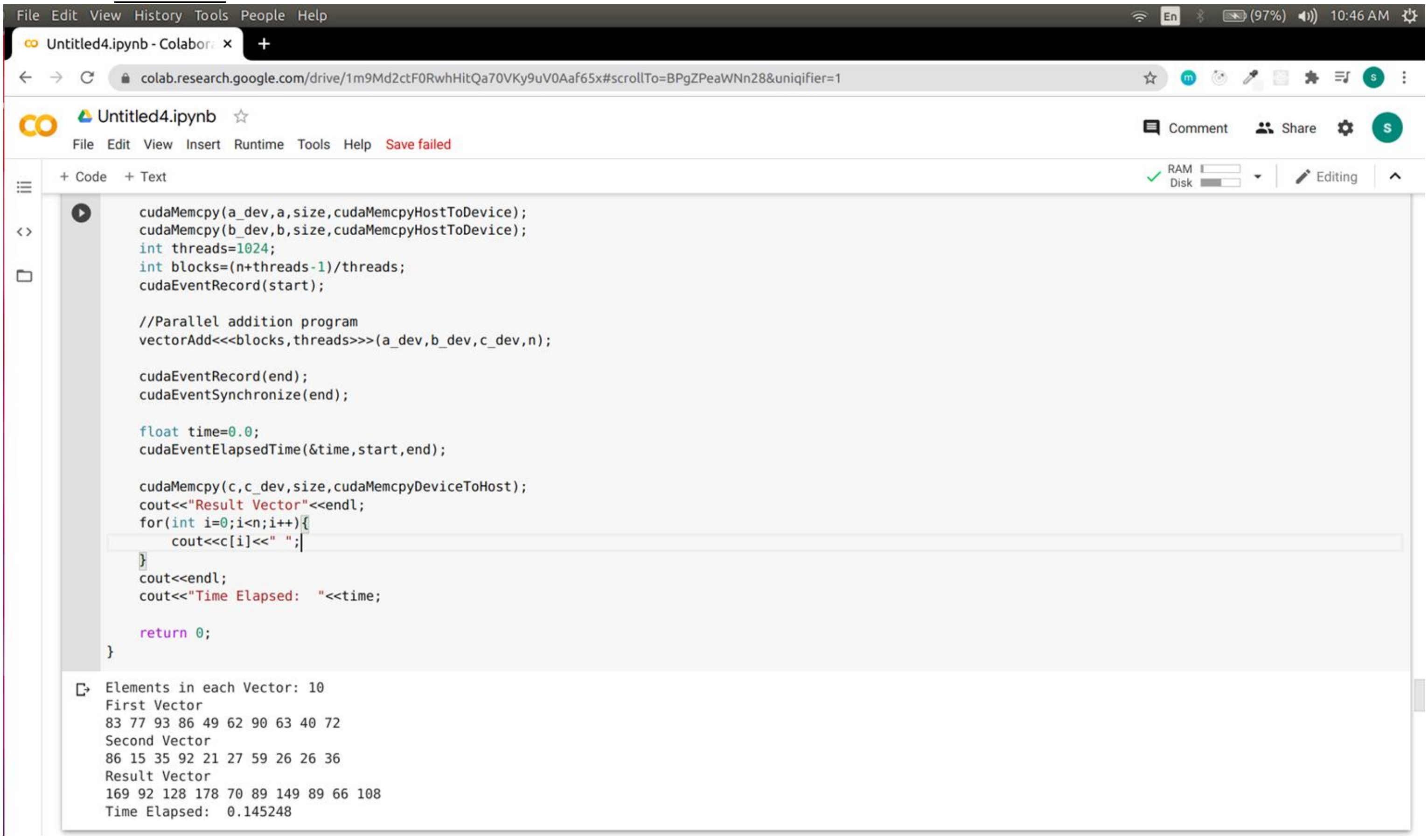
Conclusion :- Thus I understood parallel implementation of matrix, vector operations in CUDA & implement the algorithms successfully.

Assignment 2

1) Vector Addition

```
%%cu
#include<iostream>
#include<cstdlib>
using namespace std;
//VectorAdd parallel function
__global__ void vectorAdd(int *a, int *b, int *result, int n)
{
int tid=threadIdx.x+blockIdx.x*blockDim.x;
if(tid<n)
{
result[tid]=a[tid]+b[tid];
}
}
int main()
{
int *a,*b,*c;
int *a_dev,*b_dev,*c_dev;
int n=10;
a=new int[n];
b=new int[n];
c=new int[n];
int *d=new int[n];
int size=n*sizeof(int);
cudaMalloc(&a_dev,size);
cudaMalloc(&b_dev,size);
cudaMalloc(&c_dev,size);
for(int i=0;i<n;i++)
{
a[i]=rand()%100;
b[i]=rand()%100;
d[i]=a[i]+b[i]; //calculating serial addition
}
cout<<"Elements in each Vector: "<<n<<endl;
cout<<"First Vector"<<endl;
for(int i=0;i<n;i++){
cout<<a[i]<<" ";
}
cout<<endl;
cout<<"Second Vector"<<endl;
for(int i=0;i<n;i++){
cout<<b[i]<<" ";
}
cout<<endl;
cudaEvent_t start,end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaMemcpy(a_dev,a,size,cudaMemcpyHostToDevice);
cudaMemcpy(b_dev,b,size,cudaMemcpyHostToDevice);
int threads=1024;
int blocks=(n+threads-1)/threads;
cudaEventRecord(start);
//Parallel addition program
vectorAdd<<<blocks,threads>>>(a_dev,b_dev,c_dev,n);
cudaEventRecord(end);
cudaEventSynchronize(end);
float time=0.0;
cudaEventElapsedTime(&time,start,end);
cudaMemcpy(c,c_dev,size,cudaMemcpyDeviceToHost);
cout<<"Result Vector"<<endl;
for(int i=0;i<n;i++){
cout<<c[i]<<" ";
}
cout<<endl;
cout<<"Time Elapsed: "<<time;
return 0;
}
```

OUTPUT :

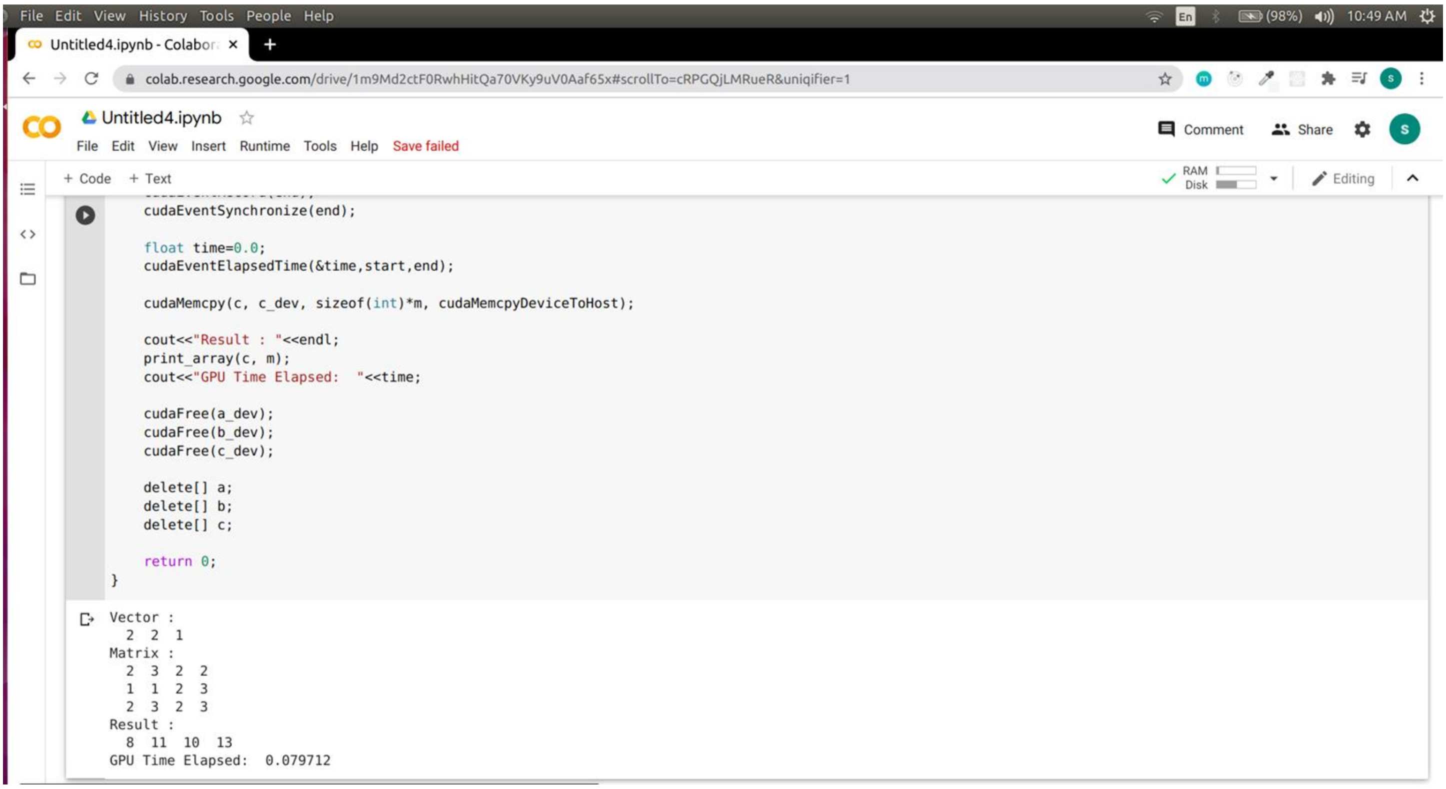


2) Vector Matrix Multiplication

```
%%cu
#include<iostream>
using namespace std;
__global__
void matrixVector(int *vec, int *mat, int *result, int n, int m)
{
int tid = blockIdx.x*blockDim.x + threadIdx.x;
int sum=0;
if(tid <= n) {
for(int i=0; i<n; i++) {
sum += vec[i]*mat[(i*m) + tid];
}
result[tid] = sum;
}
}
void init_array(int *a, int n) {
for(int i=0; i<n; i++)
a[i] = rand()%n + 1;
}
void init_matrix(int *a, int n, int m) {
for(int i=0; i<n; i++) {
for(int j=0; j<m; j++) {
a[i*m + j] = rand()%n + 1;
}
}
}
void print_array(int *a, int n) {
for(int i=0; i<n; i++) {
cout<<" "<<a[i];
}
cout<<endl;
}
void print_matrix(int *a, int n, int m) {
for(int i=0; i<n; i++) {
for(int j=0; j<m; j++)
cout<<" "<<a[i*m + j];
```

```
cout<<endl;
}
}
int main() {
int *a, *b, *c;
int *a_dev, *b_dev, *c_dev;
int n = 3;
int m = 4;
a = new int[n];
b = new int[n*m];
c = new int[m];
init_array(a, n);
init_matrix(b, n, m);
cout<<"Vector : "<<endl;
print_array(a, n);
cout<<"Matrix : "<<endl;
print_matrix(b, n, m);
cudaMalloc(&a_dev, sizeof(int)*n);
cudaMalloc(&b_dev, sizeof(int)*n*m);
cudaMalloc(&c_dev, sizeof(int)*m);
cudaEvent_t start,end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaMemcpy(a_dev, a, sizeof(int)*n, cudaMemcpyHostToDevice);
cudaMemcpy(b_dev, b, sizeof(int)*n*m, cudaMemcpyHostToDevice);
cudaEventRecord(start);
matrixVector<<<m/256+1, 256>>>(a_dev, b_dev, c_dev, n, m);
cudaEventRecord(end);
cudaEventSynchronize(end);
float time=0.0;
cudaEventElapsedTime(&time,start,end);
cudaMemcpy(c, c_dev, sizeof(int)*m, cudaMemcpyDeviceToHost);
cout<<"Result : "<<endl;
print_array(c, m);
cout<<"GPU Time Elapsed: "<<time;
cudaFree(a_dev);
cudaFree(b_dev);
cudaFree(c_dev);
delete[] a;
delete[] b;
delete[] c;
return 0;
}
```

OUTPUT:



---

3) Matrix Matrix Multiplication

```
%%cu
#include<iostream>
using namespace std;
__global__
void matrixMultiplication(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 j=0;j<n;j++)
{
sum += a[row*n+j] * b[j*k+col];
}
c[k*row+col]=sum;
}
}
void init_result(int *a, int m, int k) {
for(int i=0; i<m; i++) {
for(int j=0; j<k; j++) {
a[i*k + j] = 0;
}
}
}
void init_matrix(int *a, int n, int m) {
for(int i=0; i<n; i++) {
for(int j=0; j<m; j++) {
a[i*m + j] = rand()%10 + 1;
}
}
}

void print_matrix(int *a, int n, int m) {
for(int i=0; i<n; i++) {
for(int j=0; j<m; j++) {
cout<<" "<<a[i*m + j];
}
cout<<endl;
```

```
}
cout<<endl;
}
int main()
{
int *a,*b,*c;
int *a_dev,*b_dev,*c_dev;
int m=5, n=4, k=3;
a = new int[m*n];
b = new int[n*k];
c = new int[m*k];
init_matrix(a, m, n);
init_matrix(b, n ,k);
init_result(c, m, k);
cout<<"Matrix 1: "<<endl;
print_matrix(a, m, n);
cout<<"Matrix 2: "<<endl;
print_matrix(b, n, k);
cudaMalloc(&a_dev, sizeof(int)*m*n);
cudaMalloc(&b_dev, sizeof(int)*n*k);
cudaMalloc(&c_dev, sizeof(int)*m*k);
cudaMemcpy(a_dev, a, sizeof(int)*m*n, cudaMemcpyHostToDevice);
cudaMemcpy(b_dev, b, sizeof(int)*n*k, cudaMemcpyHostToDevice);
dim3 dimGrid(1,1);
dim3 dimBlock(16,16);
matrixMultiplication<<<dimGrid, dimBlock>>>(a_dev,b_dev,c_dev, m, n, k);
cudaMemcpy(c, c_dev, sizeof(int)*m*k, cudaMemcpyDeviceToHost);
cout<<"Result Matrix: "<<endl;
print_matrix(c, m, k);
cudaFree(a_dev);
cudaFree(b_dev);
cudaFree(c_dev);
delete[] a;
delete[] b;
delete[] c;
return 0;
}
```

<u>OUTPUT:</u>