Assignment No: 2

**Title:** Parallel Computing Using CUDA: Vector and Matrix Operation.

**Theory:**

Parallel computing refers to the process of breaking down larger problems into smaller, independent, often similar parts that can be executed simultaneously by multiple processors communicating via shared memory, the results of which are combined upon completion as part of an overall algorithm. The primary goal of parallel computing is to increase available computation power for faster application processing and problem solving.

Parallel computing infrastructure is typically housed within a single datacenter where several processors are installed in a server rack; computation requests are distributed in small chunks by the application server that are then executed simultaneously on each server.

The matrix-vector multiplication problem can be efficiently parallelized, since the naive fan-in algorithm attains the logarithmic lower bound to the computation time. On the other hand, the problem arises of distributing data among the local memories of the processors. In fact, note that matrix-vector multiplication consists of a set of scalar products, of the type (Ai, b ), where A; denotes the i-th column of the matrix, b is the vector, and (,) denotes the scalar product. It is easy to see that one of the primary concerns of the problem is to distribute the elements of vector b among different modules. The solution adopted in this paper consists of a column-oriented allocation of the components of the matrix among the leaves of the binary tree.

**Addition of two large vectors**

**Program:**

%%cu

#include<iostream>

#include<cstdio>

#include<cstdlib>

#include<cuda\_runtime.h>

#define n (2048\*2048)

#define THREADS\_PER\_BLOCK 512

using namespace std;

\_\_global\_\_ void add(int \*a, int \*b, int \*c) {

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

if (index < n){

c[index] = a[index] + b[index];

}

}

int main(){

//#Addition of two large vectors

cout<<"No of Elements: "<<n<<endl;

srand(time(0));

int \*h\_firstArr = new int[n];

int \*h\_secondArr = new int[n];

int \*h\_resultArr = new int[n];

int \*d\_firstArr;

int \*d\_secondArr;

int \*d\_resultArr;

for(int i=0;i<n;i++)

{

h\_firstArr[i]=rand()%20000;

h\_secondArr[i]=rand()%1000;

}

int size=n\*sizeof(int);

cudaMalloc((void \*\*)&d\_firstArr, size);

cudaMalloc((void \*\*)&d\_secondArr, size);

cudaMalloc((void \*\*)&d\_resultArr, size);

//#Copy inputs to device

cudaMemcpy(d\_firstArr, h\_firstArr, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_secondArr, h\_secondArr, size, cudaMemcpyHostToDevice);

add<<<n/THREADS\_PER\_BLOCK,THREADS\_PER\_BLOCK>>>(d\_firstArr, d\_secondArr, d\_resultArr);

//#Copy result back to host

cudaMemcpy(h\_resultArr, d\_resultArr, size, cudaMemcpyDeviceToHost);

cout<<"1. Addition two large vectors: "<<endl;

for(int i=0; i<100 ;i++){

cout<<"{"<<i<<"}"<<h\_firstArr[i]<<" + "<<h\_secondArr[i]<<" = "<<h\_resultArr[i]<<" ";

}

// Cleanup

free(h\_firstArr);

free(h\_secondArr);

free(h\_resultArr);

cudaFree(d\_firstArr);

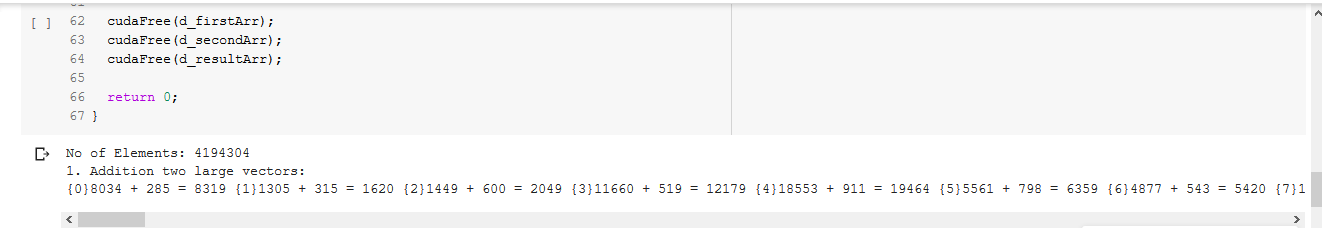
cudaFree(d\_secondArr);

cudaFree(d\_resultArr);

return 0;

}

**Output:**



**Multiply vector and matrix**

**Program:**

%%cu

#include<iostream>

#include<cstdio>

#include<cstdlib>

#include<cuda\_runtime.h>

#define col1row2 2048

#define col2 512

using namespace std;

\_\_global\_\_ void matproduct(int \*a,int \*b, int \*c){

int x = blockIdx.x;

int y = blockIdx.y;

int k;

c[col2\*y+x] = 0;

for(k = 0; k < col1row2 ; k++){

c[col2\*y+x]= c[col2\*y+x]+a[col1row2\*y+k]\*b[col2\*k+x];

}

}

int main(){

//#Multiply vector and matrix

int \*h\_vector;

int \*h\_resultVector;

int \*h\_matrix;

int \*d\_vector;

int \*d\_matrix;

int \*d\_resultVector;

h\_vector = (int \*)malloc(col1row2\*sizeof(int));

h\_resultVector = (int \*)malloc(col2\*sizeof(int));

h\_matrix = (int \*)malloc(col1row2\*col2\*sizeof(int));

cudaMalloc((void \*\*)&d\_vector, col1row2\*sizeof(int));

cudaMalloc((void \*\*)&d\_matrix, col1row2\*col2\*sizeof(int));

cudaMalloc((void \*\*)&d\_resultVector, col2\*sizeof(int));

srand(time(0));

for(int i=0; i < col1row2 ; i++){

for(int j = 0; j < col2 ; j++){

h\_matrix[i\*col2+j] = rand() % 4 + 1;

}

h\_vector[i]=rand()% 4 + 1;

}

//#Copy inputs to device

cudaMemcpy(d\_vector, h\_vector, col1row2\*sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_matrix, h\_matrix, col1row2\*col2\*sizeof(int), cudaMemcpyHostToDevice);

dim3 grid(col2,1);

matproduct<<<grid,1>>>(d\_vector,d\_matrix,d\_resultVector);

//#Copy result back to host

cudaMemcpy(h\_resultVector, d\_resultVector, col2\*sizeof(int), cudaMemcpyDeviceToHost);

cout << "2. Multiply Vector and Matrix:"<<endl;

cout << "Vector: "<<endl;

for(int i=0; i < col1row2 ; i++){

cout<<h\_vector[i]<<" ";

}

cout<<endl;

cout << "Matrix: "<<endl;

for(int i=0; i < col1row2 ; i++){

cout <<"{";

for(int j = 0; j < col2 ; j++){

cout << h\_matrix[i\*col2+j]<<" ";

}

cout <<"},";

}

cout << endl << "Result Vector: "<<endl;

for(int i=0; i < col2 ; i++){

cout<<h\_resultVector[i]<<" ";

}

//#cleanup

free(h\_vector);

free(h\_resultVector);

free(h\_matrix);

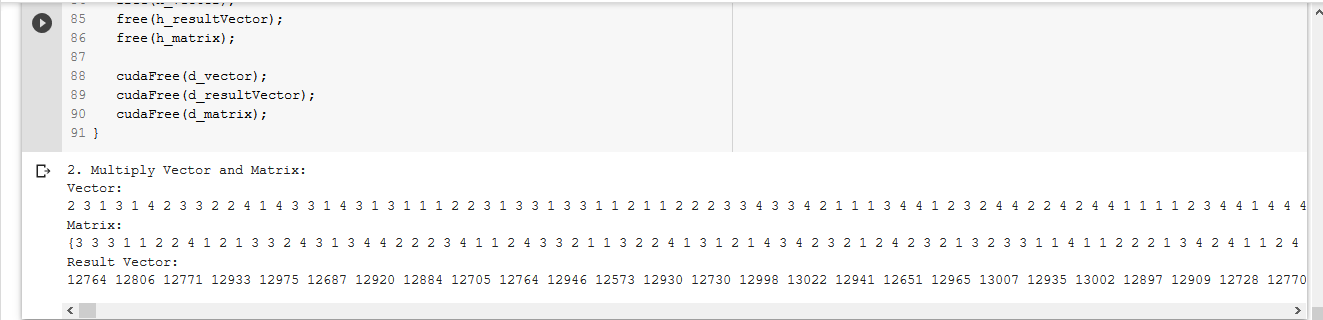
cudaFree(d\_vector);

cudaFree(d\_resultVector);

cudaFree(d\_matrix);

}

**Output:**



**Multiply NxN Arrays:**

**Program:**

%%cu

#include<iostream>

#include<cstdlib>

#include<cmath>

#define n 20

using namespace std;

\_\_global\_\_ void matrixMultiplication(int \*a, int \*b, int \*c)

{

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

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

int sum = 0;

//#n[col2\*y+x]=n[col2\*y+x]+l[col1\*y+k]\*m[col2\*k+x];

c[n\*row+col]=0;

if(row < n && col < n)

for(int j=0;j < n;j++)

{

c[n\*row+col] = c[n\*row+col] + a[n\*row+j]\*b[n\*j+col];

}

//# c[n\*row+col]=sum;

}

int main(){

//#Multiply NxN Arrays

int \*h\_matrixA;

int \*h\_matrixB;

int \*h\_resultMatrixC;

int \*d\_matrixA;

int \*d\_matrixB;

int \*d\_resultMatrixC;

h\_matrixA = (int \*)malloc(n\*n\*sizeof(int));

h\_matrixB = (int \*)malloc(n\*n\*sizeof(int));

h\_resultMatrixC = (int \*)malloc(n\*n\*sizeof(int));

cudaMalloc((void \*\*)&d\_matrixA, n\*n\*sizeof(int));

cudaMalloc((void \*\*)&d\_matrixB, n\*n\*sizeof(int));

cudaMalloc((void \*\*)&d\_resultMatrixC, n\*n\*sizeof(int));

srand(time(0));

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

for(int j = 0; j < n ; j++){

h\_matrixA[i\*n+j] = rand() % 4 + 1;

h\_matrixB[i\*n+j] = rand() % 4 + 1;

}

}

cudaMemcpy(d\_matrixA, h\_matrixA, n\*n\*sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_matrixB, h\_matrixB, n\*n\*sizeof(int), cudaMemcpyHostToDevice);

dim3 threadsPerBlock(n, n);

dim3 blocksPerGrid(1, 1);

if(n\*n>512){

threadsPerBlock.x=512;

threadsPerBlock.y=512;

blocksPerGrid.x=ceil((double)n/(double)threadsPerBlock.x);

blocksPerGrid.y=ceil((double)n/(double)threadsPerBlock.y);

}

matrixMultiplication <<<blocksPerGrid,threadsPerBlock>>>(d\_matrixA,d\_matrixB,d\_resultMatrixC);

//#Copy result back to host

cudaMemcpy(h\_resultMatrixC, d\_resultMatrixC, n\*n\*sizeof(int), cudaMemcpyDeviceToHost);

cout << "3. Multiplication of NxN Arrays using n^2 Processors "<<endl;

cout << "Matrix A: "<<endl;

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

cout <<"{";

for(int j = 0; j < n ; j++){

cout << h\_matrixA[i\*n+j]<<" ";

}

cout <<"},";

}

cout<<endl;

cout << "Matrix B: "<<endl;

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

cout <<"{";

for(int j = 0; j < n ; j++){

cout << h\_matrixB[i\*n+j]<<" ";

}

cout <<"},";

}

cout<<endl;

cout << "Result Matrix C: "<<endl;

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

cout <<"{";

for(int j = 0; j < n ; j++){

cout << h\_resultMatrixC[i\*n+j]<<" ";

}

cout <<"},";

}

//#cleanup

free(h\_matrixA);

free(h\_matrixB);

free(h\_resultMatrixC);

cudaFree(d\_matrixA);

cudaFree(d\_matrixB);

cudaFree(d\_resultMatrixC);

}

**Output:**

