Assignment No: 1

**Title:** Implement Parallel Reduction using Min, Max, Sum and Average operations. A. Parallel reduction and B. CUDA

**Theory:**

**OpenMP**

OpenMP is a set of C/C++ pragmas (or FORTRAN equivalents) which provide the programmer a high-level front-end interface which get translated as calls to threads (or other similar entities). The key phrase here is "higher-level"; the goal is to better enable the programmer to "think parallel," alleviating him/her of the burden and distraction of dealing with setting up and coordinating threads. For example, the OpenMP directive.

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run parallely, with the runtime environment allocating threads to different processors.

Core elements

The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.

Thread creation

The pragma omp parallel is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as master thread with thread ID 0.

**Program : -**

#include<iostream>

#include<time.h>

#include<omp.h>

#include<cstdlib>

using namespace std;

int main(){

clock\_t start, finish;

int array[100000];

cout << "\n\*\*\*\*\*\*\*\*\*Parallel Execution\*\*\*\*\*\*\*\*\*\*\n";

int n = 100000;

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

array[i] = rand()%10;

}

int min = 9999;

int max = -9999;

int min\_index;

int max\_index;

float sum = 0;

double start\_parallel = omp\_get\_wtime( );

#pragma omp parallel for

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

if(array[i] < min){

min = array[i];

min\_index = i;

}

if(array[i] > max){

max = array[i];

max\_index = i;

}

sum = sum + array[i];

}

double finish\_parallel = omp\_get\_wtime( );

cout << "\nmin is :" << min;

cout << "\nmax is :" << max;

cout << "\nsum is :" << sum;

cout << "\navg is :" << sum/n;

cout << "\ntotal time taken " << finish\_parallel - start\_parallel << " sec\n";

cout << "\n\*\*\*\*\*\*\*\*\*Sequential Execution\*\*\*\*\*\*\*\*\*\*\n";

min = 9999;

max = -9999;

min\_index;

max\_index;

sum = 0;

start = clock();

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

if(array[i] < min){

min = array[i];

min\_index = i;

}

if(array[i] > max){

max = array[i];

max\_index = i;

}

sum = sum + array[i];

}

finish = clock();

cout << "\nmin is :" << min;

cout << "\nmax is :" << max;

cout << "\nsum is :" << sum;

cout << "\navg is :" << sum/n;

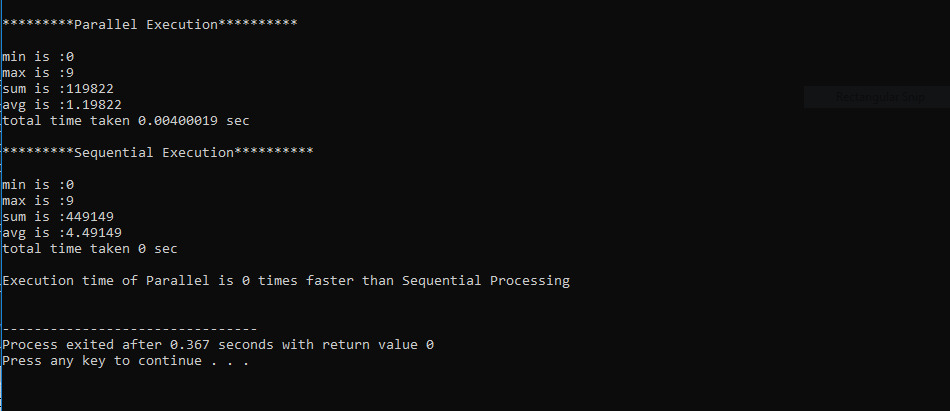
cout << "\ntotal time taken " << finish - start << " sec\n\n";

cout << "Execution time of Parallel is " << (finish - start) / (finish\_parallel - start\_parallel) << " times faster than Sequential Processing \n\n";

return 0;

}

**Output:-**

****

**CUDA:**

1. **The maximum elements in the vector**

**Program:**

#include <cuda.h>

#include <iostream>

using namespace std;

#define SIZE 100

\_\_global\_\_ void max(int \*a , int \*c)

{

int i = threadIdx.x;

\*c = a[0];

if(a[i] > \*c)

{

\*c = a[i];

}

}

int main()

{

int i;

int a[SIZE];

int c;

int \*dev\_a, \*dev\_c;

cudaMalloc((void \*\*) &dev\_a, SIZE\*sizeof(int));

cudaMalloc((void \*\*) &dev\_c, SIZE\*sizeof(int));

for( i = 0 ; i < SIZE ; i++)

{

a[i]=i;

}

for( i = 0 ; i < SIZE ; i++)

{

cout<<a[i]<<" ";

}

cudaMemcpy(dev\_a , a, SIZE\*sizeof(int),cudaMemcpyHostToDevice);

max<<<1,SIZE>>>(dev\_a,dev\_c);

cudaMemcpy(&c, dev\_c, SIZE\*sizeof(int),cudaMemcpyDeviceToHost);

cout<<"\n max value = ";

cout<<c;

cudaFree(dev\_a);

cudaFree(dev\_c);

return 0;

}

**Output:-**



1. **The minimum element in the vector**

**Program: -**

#include <cuda.h>

#include <stdio.h>

#include <time.h>

#define SIZE 1000

\_\_global\_\_ void max(int \*a , int \*c)

{

int i = threadIdx.x;

\*c = a[0];

if(a[i] < \*c)

{

\*c = a[i];

}

}

int main()

{

int i;

srand(time(NULL));

int a[SIZE];

int c;

int \*dev\_a, \*dev\_c;

cudaMalloc((void \*\*) &dev\_a, SIZE\*sizeof(int));

cudaMalloc((void \*\*) &dev\_c, SIZE\*sizeof(int));

for( i = 0 ; i < SIZE ; i++)

{

a[i] = i;

}

cudaMemcpy(dev\_a , a, SIZE\*sizeof(int),cudaMemcpyHostToDevice);

max<<<1,SIZE>>>(dev\_a,dev\_c);

cudaMemcpy(&c, dev\_c, SIZE\*sizeof(int),cudaMemcpyDeviceToHost);

printf("\nmin = %d ",c);

cudaFree(dev\_a);

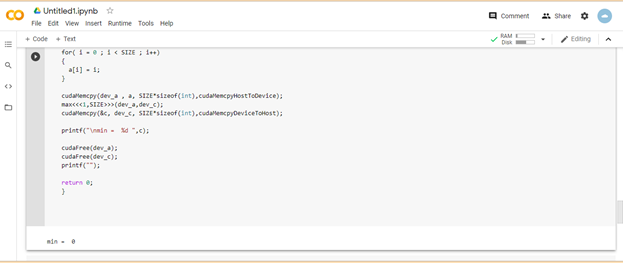
cudaFree(dev\_c);

printf("");

return 0;

}

**Output: -**



1. **The arithmetic mean of the vector**

**Program:-**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// CUDA kernel. Each thread takes care of one element of c

\_\_global\_\_ void vecAdd(double \*a, double \*b, double \*c, int n)

{

// Get our global thread ID

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

// Make sure we do not go out of bounds

if (id < n)

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

}

int main( int argc, char\* argv[] )

{

// Size of vectors

//int n = 100000;

int n=5;

// Host input vectors

double \*h\_a;

double \*h\_b;

//Host output vector

double \*h\_c;

// Device input vectors

double \*d\_a;

double \*d\_b;

//Device output vector

double \*d\_c;

// Size, in bytes, of each vector

size\_t bytes = n\*sizeof(double);

// Allocate memory for each vector on host

h\_a = (double\*)malloc(bytes);

h\_b = (double\*)malloc(bytes);

h\_c = (double\*)malloc(bytes);

// Allocate memory for each vector on GPU

cudaMalloc(&d\_a, bytes);

cudaMalloc(&d\_b, bytes);

cudaMalloc(&d\_c, bytes);

int i;

// Initialize vectors on host

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

// h\_a[i] = sin(i)\*sin(i);

h\_a[i]=i;

h\_b[i]=i;

//h\_b[i] = cos(i)\*cos(i);

}

// Copy host vectors to device

cudaMemcpy( d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

cudaMemcpy( d\_b, h\_b, bytes, cudaMemcpyHostToDevice);

int blockSize, gridSize;

// Number of threads in each thread block

blockSize = 1024;

// Number of thread blocks in grid

gridSize = (int)ceil((float)n/blockSize);

// Execute the kernel

vecAdd<<<gridSize, blockSize>>>(d\_a, d\_b, d\_c, n);

// Copy array back to host

cudaMemcpy( h\_c, d\_c, bytes, cudaMemcpyDeviceToHost );

// Sum up vector c and print result divided by n, this should equal 1 within error

double sum = 0;

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

sum += h\_c[i];

printf("Average mean of 2 vectors: %f\n", sum/n);

// Release device memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

// Release host memory

free(h\_a);

free(h\_b);

free(h\_c);

return 0;

}

**Output: -**



1. **The standard deviation of the values in the vector**

**Program: -**

#include<stdio.h>

#include<time.h>

#include<iostream>

#define w 256

#define h 256

#define N w\*h

using namespace std;

\_\_global\_\_ void reduce(int\*,int\*,int\*);

int main(void)

{

int\* hostA = (int\*)malloc(N\*sizeof(int));

int\* hostB = (int\*)malloc(N\*sizeof(int));

int\* hostMean = (int\*)malloc(sizeof(int));

\*hostMean = 32767;

int\* deviceA; int \*deviceB;int\*deviceMean;

cudaMalloc(&deviceA,sizeof(int)\*N);

cudaMalloc(&deviceB,sizeof(int)\*N);

cudaMalloc(&deviceMean,sizeof(int));

//randomly generate array hostA

srand(time(0));

int i;

//initialize host vector by random elements

for(i=0;i<N;i++)

{

hostA[i] = i;

}

hostB[0]=0.0;

cudaMemcpy(deviceA,hostA,N\*sizeof(int),cudaMemcpyHostToDevice);

cudaMemcpy(deviceB,hostB,N\*sizeof(int),cudaMemcpyHostToDevice);

cudaMemcpy(deviceMean,hostMean,sizeof(int),cudaMemcpyHostToDevice);

dim3 blocksize(256);

dim3 gridsize(N/blocksize.x);

float gpu\_elapsed\_time;

cudaEvent\_t gpu\_start,gpu\_stop;

cudaEventCreate(&gpu\_start);

cudaEventCreate(&gpu\_stop);

cudaEventRecord(gpu\_start,0);

reduce<<<gridsize,blocksize>>>(deviceA,deviceB,deviceMean);

cudaDeviceSynchronize();

cudaMemcpy(hostB,deviceB,sizeof(int),cudaMemcpyDeviceToHost);

cudaEventRecord(gpu\_stop, 0);

cudaEventSynchronize(gpu\_stop);

cudaEventElapsedTime(&gpu\_elapsed\_time, gpu\_start, gpu\_stop);

cudaEventDestroy(gpu\_start);

cudaEventDestroy(gpu\_stop);

double std\_dev = pow(hostB[0]/(N),0.5);

cout<<"Reduced array standard deviation is = "<<std\_dev<<endl;

std::cout<<"The gpu took: "<<gpu\_elapsed\_time<<" milli-seconds"<<std::endl;

clock\_t cpu\_start = clock();

int sum=0;

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

sum = sum + int(pow((hostA[i] - (\*hostMean)),2.0));

}

//cout<<"sum == "<<sum<<endl;

double std\_dev\_actual = pow(sum/(N),0.5);

printf("Actual value of standard deviation should be: %f \n", std\_dev\_actual);

clock\_t cpu\_stop = clock();

clock\_t cpu\_elapsed\_time = 1000\*(cpu\_stop - cpu\_start)/CLOCKS\_PER\_SEC;

std::cout<<"The cpu took: "<<cpu\_elapsed\_time<<" milli-seconds"<<std::endl;

cudaFree(deviceA);

cudaFree(deviceB);

delete[] hostB;

delete[] hostA;

}

\_\_global\_\_ void reduce(int\* input,int\* output,int\* mean)

{

\_\_shared\_\_ int shared\_data[256];

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

shared\_data[threadIdx.x] = int( pow(double(input[i]- \*mean),2.0));

\_\_syncthreads();

for(int s=1;s<blockDim.x;s\*=2)

{

int index = 2 \* s \* threadIdx.x;;

if (index < blockDim.x)

{

shared\_data[index] += shared\_data[index + s];

}

\_\_syncthreads();

}

if (threadIdx.x == 0)

atomicAdd(output,shared\_data[0]);

}

**Output: -**

****