LP-1

High Performance Computing

Assignment 1.

Date of Completion – 19 · 8 · 2020

Title :- Parallel Reduction using CUDA

Problem Statement :-

a) Implement parallel reduction using MIN, MAX, SUM & AVG operation

b) Write a CUDA program that, given an N-element vector find
- The maximum element in the vector
- The minimum element in the vector
- The arithmetic mean of the vector
- The standard deviation of the values in the vect

Test for input N and generate a randomized vector V of length N (N should be large) The program should generate sup output as the two compared maximum values as well as the time taken to find each value.

Objectives :- a) To learn parallel programming concepts.
b) To learn parallel computing using CUDA.

Outcomes :- a) Know parallel computing concepts
b) Use CUDA for parallel programming.

Requirements :- Ubuntu OS, Nvidia GPU, CUDA API (c/c++) Google Colab

Theory :- CUDA :-

1) It is a parallel Computing platform and API model created by NVIDIA.
2) It enables programmers to use CUDA enable GPU for general purpose processing
3) It is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of Computer kernels.

CUDA Programming

1) NVCC compiler separates the host and device code (GPU) in compilation phase.
2) Source code has .cu extension.

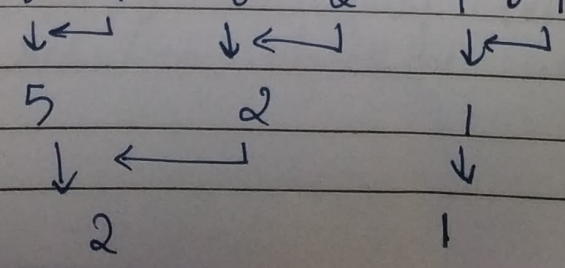
Program Structure:-

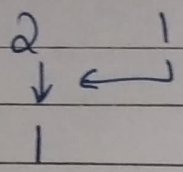
1) include headers
2) allocate GPU memories
3) Copy data from CPU to GPU.
4) Invoke the Kernel code
5) Copy data back from GPU to CPU
6) Destroy GPU memories.

Parallel Reduction:-

1) Here Every thread calculate result from its own element and some other element.
2) This result is forwarded to next round of threads.
3) Number of threads are halfed in each round untill single thread remains.

Eg for calculating minimum from 6 element array





Test case :-

for n = 1000

minimum = ~~121~~ 3

parallel execution time : 8ms

normal execution time : 24ms

4 maximum = 199

parallel execution : 2ms

normal execution : 38ms

Sum : 1210

Parallel execution : 2ms

normal execution : 15ms

Standard deviation : 0.11726

parallel execution : 2ms

Efficiency = $\frac{WSA}{WCPA}$

Worst case execution time for sequential algorithm => WCSA

Worst case ~~parallel~~ execution time for parallel algorithm => WCPA

Conclusion :- ~~we~~ I successfully executed the parallel reduction algorithms using CUDA.

```
%%cu

#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<stdio.h>
#include<stdlib.h>
#include<cuda_runtime.h>
#include<time.h>
using namespace std;



__global__ void minimum(int *input)
{
int tid=threadIdx.x;
auto step_size=1;
int number_of_threads=blockDim.x;
while(number_of_threads>0)
{
if(tid<number_of_threads)
{
int first=tid*step_size*2;
int second=first+step_size;
if(input[second]<input[first])
input[first]=input[second];
}
step_size=step_size*2;
number_of_threads/=2;
}

}

__global__ void max(int *input)
{
int tid=threadIdx.x;
auto step_size=1;
int number_of_threads=blockDim.x;
while(number_of_threads>0)
{
if(tid<number_of_threads)
{
int first=tid*step_size*2;
int second=first+step_size;
if(input[second]>input[first])
input[first]=input[second];
}
step_size*=2;
number_of_threads/=2;
}
}

__global__ void sum(int *input)
{
const int tid=threadIdx.x;
auto step_size=1;
int number_of_threads=blockDim.x;
while(number_of_threads>0)
{
if(tid<number_of_threads)
{
const int first=tid*step_size*2;
const int second=first+step_size;
```

```
input[first]=input[first]+input[second];
}
step_size = step_size*2;;
number_of_threads =number_of_threads/2;
}
}

__global__ void average(int *input) //You can use above sum() to calculate sum and divide it by num_of_elememts
{
const int tid=threadIdx.x;
auto step_size=1;
int number_of_threads=blockDim.x;
int totalElements=number_of_threads*2;
while(number_of_threads>0)
{
if(tid<number_of_threads)
{
const int first=tid*step_size*2;
const int second=first+step_size;
input[first]=input[first]+input[second];
}
step_size = step_size*2;;
number_of_threads =number_of_threads/2;
}
input[0]=input[0]/totalElements;
}

__global__ void mean_diff_sq(float *input, float mean) {
input[threadIdx.x] -= mean;
input[threadIdx.x] *= input[threadIdx.x];
}

__global__ void sum_floats(float *input) {
int tid = threadIdx.x;
int step_size = 1;
int number_of_threads = blockDim.x;
while(number_of_threads > 0) {
if(tid < number_of_threads) {
int first = tid * step_size * 2;
int second = first + step_size;
input[first] += input[second];
}
step_size <<= 1;
number_of_threads >>= 1;
}
}

void copy_int_to_float(float *dest, int *src, int size){
for(int i=0; i<size; i++)
dest[i] = float(src[i]);
}

int min_normal(int *input,int size){
int minnum=1000000;
for(int i=0;i<size;i++){
if(input[i]<minnum){
minnum=input[i];
}
}
return minnum;
}
int max_normal(int *input,int size){
```

```
int maxnum=0;
for(int i=0;i<size;i++){
if(input[i]>maxnum){
maxnum=input[i];
}
}
return maxnum;
}

int main()
{
int maxCPU=0;
int n;
n=8800;
srand(n);
int *arr=new int[n];
int min=20000;
//# Generate Input array using rand()
for(int i=0;i<n;i++)
{
arr[i]=rand()%200;
//if(arr[i]<min)
//min=arr[i];
if(arr[i]>maxCPU)
maxCPU=arr[i];
cout<<arr[i]<<" ";
}
cout<<endl;
int size=n*sizeof(int); //calculate no. of bytes for array
int *arr_d,result1;
//#Minimum Operation
cout<<"Minimum Operation Comparison"<<endl;
cudaMalloc(&arr_d,size);
cudaMemcpy(arr_d,arr,size,cudaMemcpyHostToDevice);
clock_t b1=clock();
minimum<<<1,n/2>>>(arr_d);
clock_t e1=clock();
cudaMemcpy(&result1,arr_d,sizeof(int),cudaMemcpyDeviceToHost);
cout<<"(Parallel) The minimum element is "<<result1<<endl;
cout<<"Parallel execution takes "<<e1-b1<<"ms"<<endl;
clock_t b2=clock();
min=min_normal(arr,n);
clock_t e2=clock();
cout<<"Normal execution takes "<<(e2-b2)<<"ms"<<endl;
//cout<<"(Normal) The minimum element is "<<min<<endl;
cout<<endl;
//#Maximum OPERATION
cout<<"Maximum Operation Comparison"<<endl;
int *arr_max,maxValue;
cudaMalloc(&arr_max,size);
cudaMemcpy(arr_max,arr,size,cudaMemcpyHostToDevice);
clock_t tb=clock();
max<<<1,n/2>>>(arr_max);
clock_t e=clock();
cudaMemcpy(&maxValue,arr_max,sizeof(int),cudaMemcpyDeviceToHost);
int maxC;
clock_t tb1=clock();
maxC=max_normal(arr,n);
clock_t te=clock();
//cout<<"(Normal) The maximum element is "<<macC<<endl;
cout<<"(Parallel) The maximum element is "<<maxCPU<<endl;
cout<<"Parallel execution takes "<<e-tb<<"ms"<<endl;
cout<<"Normal execution takes "<<te-tb1<<"ms"<<endl;
```

```
//#SUM OPERATION
cout<<endl;
cout<<"Sum Operation Comparison"<<endl;
int *arr_sum,sumValue;
cudaMalloc(&arr_sum,size);
cudaMemcpy(arr_sum,arr,size,cudaMemcpyHostToDevice);
clock_t t;
t= clock();
sum<<<1,n/2>>>(arr_sum);
cudaMemcpy(&sumValue,arr_sum,sizeof(int),cudaMemcpyDeviceToHost);
clock_t end;
end= clock();
double time=((double)(end-t));
cout<<"Parallel execution takes "<<time<<"ms"<<endl;
cout<<"The sum of elements is "<<sumValue<<endl;
//# OR----------
//#AVG OPERATION
cout<<endl;
cout<<"Averge Operation Comparison"<<endl;
int *arr_avg,avgValue;
cudaMalloc(&arr_avg,size);
cudaMemcpy(arr_avg,arr,size,cudaMemcpyHostToDevice);
clock_t tb2;
tb2= clock();
average<<<1,n/2>>>(arr_avg);
clock_t te2;
te2= clock();
cudaMemcpy(&avgValue,arr_avg,sizeof(int),cudaMemcpyDeviceToHost);

cout<<"The average of elements is "<<avgValue<<endl;
cout<<"Parallel execution takes "<<te2-tb2<<"ms"<<endl;
//#STANDARD DEVIATION
cout<<endl;
cout<<"Standard Deviation Operation Comparison"<<endl;
float *arr_float;
float *arr_std, stdValue;
arr_float = (float *)malloc(n*sizeof(float));
cudaMalloc((void **)&arr_std, n*sizeof(float));
copy_int_to_float(arr_float, arr, n);
cudaMemcpy(arr_std, arr_float, n*sizeof(float), cudaMemcpyHostToDevice);
clock_t tb3;
tb3= clock();
mean_diff_sq <<<1,n>>>(arr_std, avgValue);
sum_floats<<<1,n/2>>>(arr_std);
clock_t te3;
te3= clock();
cudaMemcpy(&stdValue, arr_std, sizeof(float), cudaMemcpyDeviceToHost);
stdValue = stdValue / n;
cout<<"The variance is "<<stdValue<<endl;
stdValue = sqrt(stdValue);
cout<<"The standard deviation is "<<stdValue<<endl;
cout<<"Parallel execution takes "<<te3-tb3<<"ms"<<endl;
//# Free all allcated device memeory
cudaFree(arr_d);
cudaFree(arr_sum);
cudaFree(arr_max);
cudaFree(arr_avg);

return 0;

}
```

OUTPUT :
Parallel execution takes less time when n is large.