LP1 Assignment HPC H1

Parallel Computing using CUDA

Date - 16th August, 2020.

Assignment Number - HPC H1

Title

Parallel Computing using CUDA

Problem Definition

1. Implement Parallel Reduction using Min, Max, Sum and Average operations.
2. Write a CUDA program that, given an N-element vector, find-
    a. The maximum element in the vector
    b. The minimum element in the vector
    c. The arithmetic mean of the vector
    d. The standard deviation of the values in the vector

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

Learning Objectives

- Learn parallel decomposition of problems.
- Learn parallel computing using CUDA

Learning Outcomes

I will be able to decompose problems into subproblems, to learn how to use GPUs, to learn to solve sub problems using threads on GPU cores.

Software Packages and Hardware Apparatus Used

- Operating System : 64-bit Ubuntu 18.04
- Browser : Google Chrome
- Programming Language : C++, Python 3
- Jupyter Notebook Environment : Google Colaboratory

Related Mathematics

Mathematical Model

Let S be the system set:

S = {s; e; X; Y; Fme;Ff;DD;NDD; Fc; Sc}

s=start state

e=end state

X=set of inputs
X = {X1}
where X1 = Vector

Y= Output Set
Y = {Y1,Y2,Y3,Y4,Y5,Y6} where
Y1 = Minimum element
Y2 = Maximum element
Y3 = Average of all elements
Y4 = Sum of all elements
Y5 = Variance
Y6 = Standard Deviation

Fme is the set of main functions
Fme = {f0} where
F0 = output display function

Ff is the set of friend functions
Ff = {f1,f2,f3,f4,f5,f6,f7,f8,f9} where
f1 = function to find Minimum
f2 = function to find Maximum
f3 = function to find Sum
f4 = function to find Mean
f5 = function to find Variance
f6 = function to find Standard Deviation
f7 = function to write CUDA code from Google Colab Cell
into a file and compile the code
f8 = function to initialize vector with random elements
f9 = function to display vector

DD = Deterministic Data
Input Vector X1

NDD=Non-deterministic data
No non deterministic data

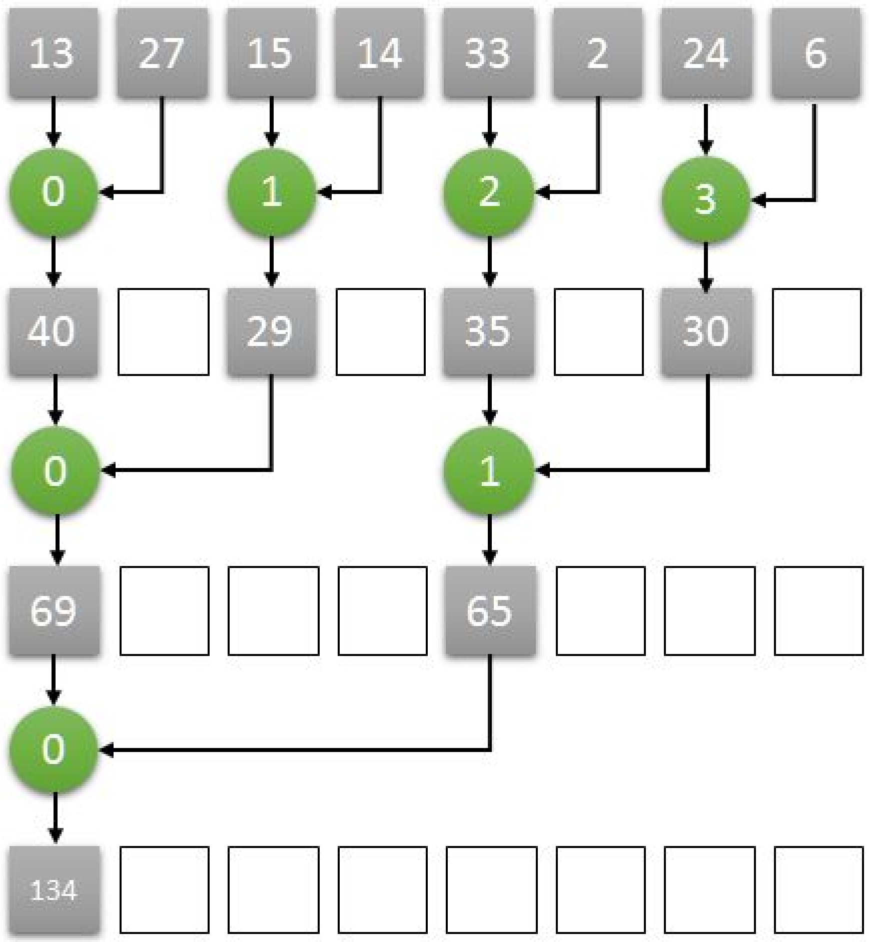
Fc =failure case:
No failure case identified for this application

Concepts related Theory

Parallel computing is a type of computing architecture in which several processors or processes execute or process an application or computation simultaneously. The primary objective of parallel computing is to increase the available computation power for faster application processing or task resolution

Diagram for Parallel Reduction

Sum Function Demonstrated



Algorithms

Algorithm to get sum of all the elements

1. Let N be the number of elements
2. threadCount is N/2 which compares two elements at a time
3. threadCount := N/2
4. stepSize := 1
5. Each thread stores the sum of the two elements in place of first element
6. For each iteration
    a. Select two consecutive elements
    b. Compute their sum
    c. Store the sum in place of first array
    d. threadCount := threadCount/2
    e. stepSize := stepSize*2

7. return first element of the array

Algorithm to get maximum element

1. Let N be the number of elements
2. threadCount is N/2 which compares two elements at a time
3. threadCount := N/2
4. stepSize := 1
5. Each thread stores the sum of the two elements in place of first element
6. For each iteration
    a. Select two consecutive elements
    b. Compute the maximum between the two
    c. Store the sum in place of first array
    d. threadCount := threadCount/2
    e. stepSize := stepSize*2
7. return first element of the array

Algorithm to get minimum element

1. Let N be the number of elements
2. threadCount is N/2 which compares two elements at a time
3. threadCount := N/2
4. stepSize := 1
5. Each thread stores the sum of the two elements in place of first element
6. For each iteration
    a. Select two consecutive elements
    b. Compute the minimum between the two
    c. Store the sum in place of first array
    d. threadCount := threadCount/2
    e. stepSize := stepSize*2
7. return first element of the array

Algorithm to get variance

1. Let N be the number of elements
2. threadCount is N/2 which compares two elements at a time
3. threadCount := N/2
4. stepSize := 1
5. Each thread stores the sum of the two elements in place of first element
6. Mean Square all elements in place
7. Get sum of all elements, store it in var

8. Var := var/N
9. return

Greek Symbols in Probability and Statistics

1) μ represents the population mean
2) σ represents the population standard deviation

Source Code

```
#include<iostream>
#include<math.h>

#define n 8

using namespace std;

//Kernel Functions

__global__ void minimum(int *input) {

   int threadId = threadIdx.x;
   int stepSize = 1;
   int threadCount = blockDim.x;

   while(threadCount>0) {
       if(threadId < threadCount) {
           int first = threadId*stepSize*2;
           int second = first + stepSize;
           //Modify Array In Place
           if(input[second] < input[first])
             input[first] = input[second];
       }
       stepSize *=2;
       threadCount /= 2;
   }
}
```

```
__global__ void maximum(int *input) {
   int threadId = threadIdx.x;
   int stepSize = 1;
   int threadCount = blockDim.x;

   //Array is updated Inplace
   while(threadCount>0) {
       if(threadId < threadCount) {
           int first = threadId*stepSize*2;
           int second = first + stepSize;
           //Modify Array In Place
           if(input[second] > input[first])
             input[first] = input[second];
       }
       stepSize <<= 1;
       threadCount >>= 1;
   }
}

__global__ void sum(int *input) {
   const int threadId = threadIdx.x;
   int stepSize = 1;
   int threadCount = blockDim.x;

   while(threadCount > 0) {
       if(threadId < threadCount) {
           int first = threadId * stepSize * 2;
           int second = first + stepSize;
           //Modify Array In Place
           input[first] += input[second];
       }
       stepSize <<= 1;
       threadCount >>= 1;

   }
}
```

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

__global__ void sum(float *input) {
   int threadId = threadIdx.x;
   int stepSize = 1;
   int threadCount = blockDim.x;

   while(threadCount > 0) {
       if(threadId < threadCount) {
           int first = threadId * stepSize * 2;
           int second = first + stepSize;
           //Modify array in place
           input[first] += input[second];
       }
       stepSize <<= 1;
       threadCount >>= 1;

   }
}

//Host Functions

void initialize_vector(int *input, int size) {
   for(int i=0; i<size; i++)  {
       input[i] = rand()%500;
   }
}

void display_vector(int *input, int size) {
   if(size==0){
     cout<<"[]";
     return;
   }
   cout<<"["<<input[0];
   for(int i=1; i<size; i++)  {
       cout<<", "<<input[i];
   }
   cout<<"]";
}

int getMinimum(int* arr_d, int* arr, int size){
   int result;
   cudaMemcpy(arr_d, arr, size, cudaMemcpyHostToDevice);
   minimum<<<1,n/2>>>(arr_d);
   //Copying  Just the first Element of the array
   cudaMemcpy(&result, arr_d, sizeof(int), cudaMemcpyDeviceToHost);
   return result;
}

int getMaximum(int* arr_d, int* arr, int size){
   int result;
   cudaMemcpy(arr_d, arr, size, cudaMemcpyHostToDevice);
   maximum<<<1,n/2>>>(arr_d);
   //Copying  Just the first Element of the array
   cudaMemcpy(&result, arr_d, sizeof(int), cudaMemcpyDeviceToHost);
   return result;
}

int getSum(int* arr_d, int* arr, int size){
   int result;
   cudaMemcpy(arr_d, arr, size, cudaMemcpyHostToDevice);
   sum<<<1,n/2>>>(arr_d);
   //Copying  Just the first Element of the array
   cudaMemcpy(&result, arr_d, sizeof(int), cudaMemcpyDeviceToHost);
   return result;
}

float getAverage(int* arr_d, int* arr, int size){
   float result = float(getSum(arr_d,arr,size))/n;
   return result;
```

```
}

float getVariance(int* arr_d, int* arr, int size){

   //int n = size/sizeof(int);

   float mean = getAverage(arr_d, arr, size);

   float *arr_float;
   float *arr_std, result;

   arr_float = (float *)malloc(n*sizeof(float));

   cudaMalloc((void **)&arr_std, n*sizeof(float));

   for(int i=0; i<n; i++)
       arr_float[i] = float(arr[i]);

   //Mean Squared
   cudaMemcpy(arr_std, arr_float, n*sizeof(float),
cudaMemcpyHostToDevice);
   meanSquared <<<1,n>>>(arr_std, mean);
   cudaMemcpy(arr_std, arr_float, n*sizeof(float),
cudaMemcpyDeviceToHost);

   //Add Mean Squared of all the elements
   sum<<<1,n/2>>>(arr_std);
   cudaMemcpy(&result, arr_std, sizeof(float), cudaMemcpyDeviceToHost);

   result /= n;

   cudaFree(arr_std);

   return result;

}
```

```
float getStdDeviation(int* arr_d, int* arr, int size){
   float result = sqrt(getVariance(arr_d,arr,size));
   return result;
}

int main() {

   //Host Variable
   int *arr;

   //Device Variable
   int *arr_d;

   int size = n*sizeof(int);

   //Allocate Memory to Host Variable
   arr = (int *)malloc(size);

   initialize_vector(arr, n);
   cout<<endl<<"Vector - ";
   display_vector(arr, n);

   //Allocate Memory to Device Variable
   cudaMalloc((void **)&arr_d, size);

   //Output Variables
   int min, max, sum;
   float avg, var, stddev;

   //Host Function Calls - They Launch the Kernel
   min = getMinimum(arr_d, arr, size);
   max = getMaximum(arr_d, arr, size);
   sum = getSum(arr_d, arr, size);
   avg = getAverage(arr_d, arr, size);
   var = getVariance(arr_d, arr, size);
   stddev = getStdDeviation(arr_d, arr, size);
```

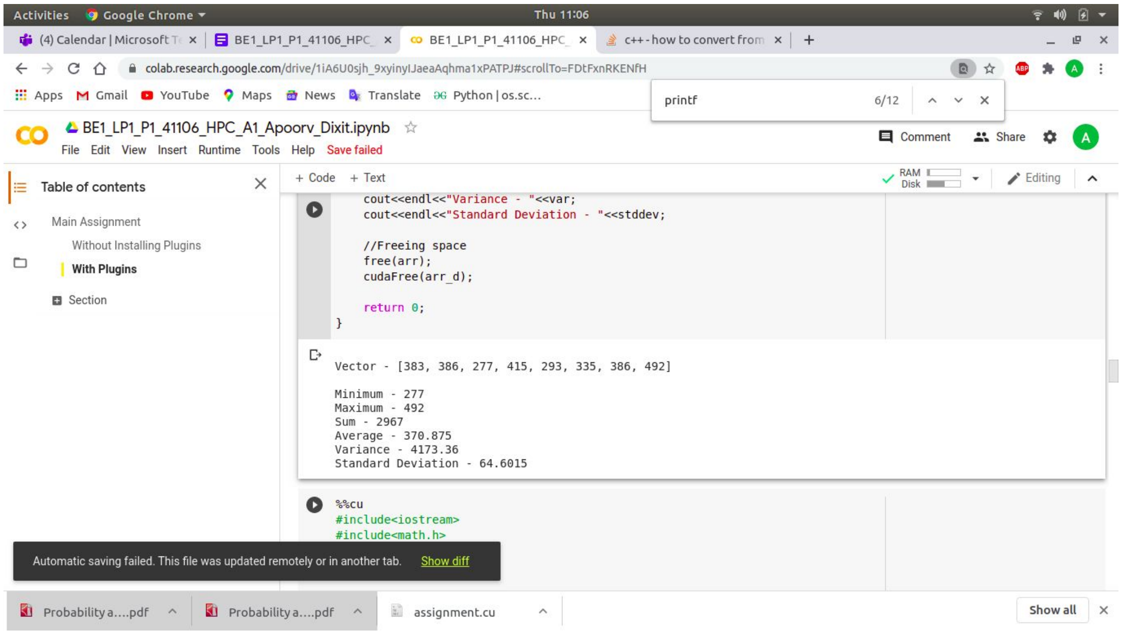
```
    //Output to the console
    cout<<endl<<"\nMinimum - "<<min;
    cout<<endl<<"Maximum - "<<max;
    cout<<endl<<"Sum - "<<sum;
    cout<<endl<<"Average - "<<avg;
    cout<<endl<<"Variance - "<<var;
    cout<<endl<<"Standard Deviation - "<<stddev;

    //Freeing space
    free(arr);
    cudaFree(arr_d);

    return 0;
}
```

Google Colab Notebook Link

https://colab.research.google.com/drive/1iA6U0sjh_9xyinyIJaeaAqhma1xPATPJ?usp=sharing

Output Screenshots

Min, Max, Sum, Average, Variance, Standard Deviation



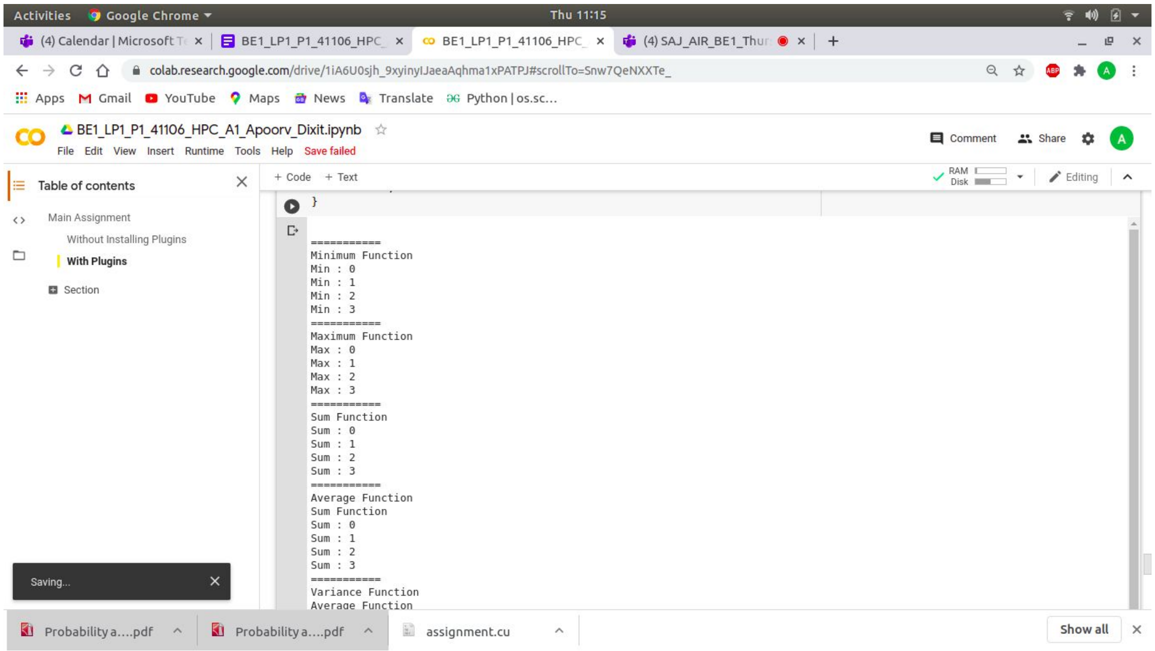
```
    cout<<endl<<"Variance - "<<var;
    cout<<endl<<"Standard Deviation - "<<stddev;

    //Freeing space
    free(arr);
    cudaFree(arr_d);

    return 0;
}
```

```
Vector - [383, 386, 277, 415, 293, 335, 386, 492]

Minimum - 277
Maximum - 492
Sum - 2967
Average - 370.875
Variance - 4173.36
Standard Deviation - 64.6015
```

```
%%cu
#include<iostream>
#include<math.h>
```

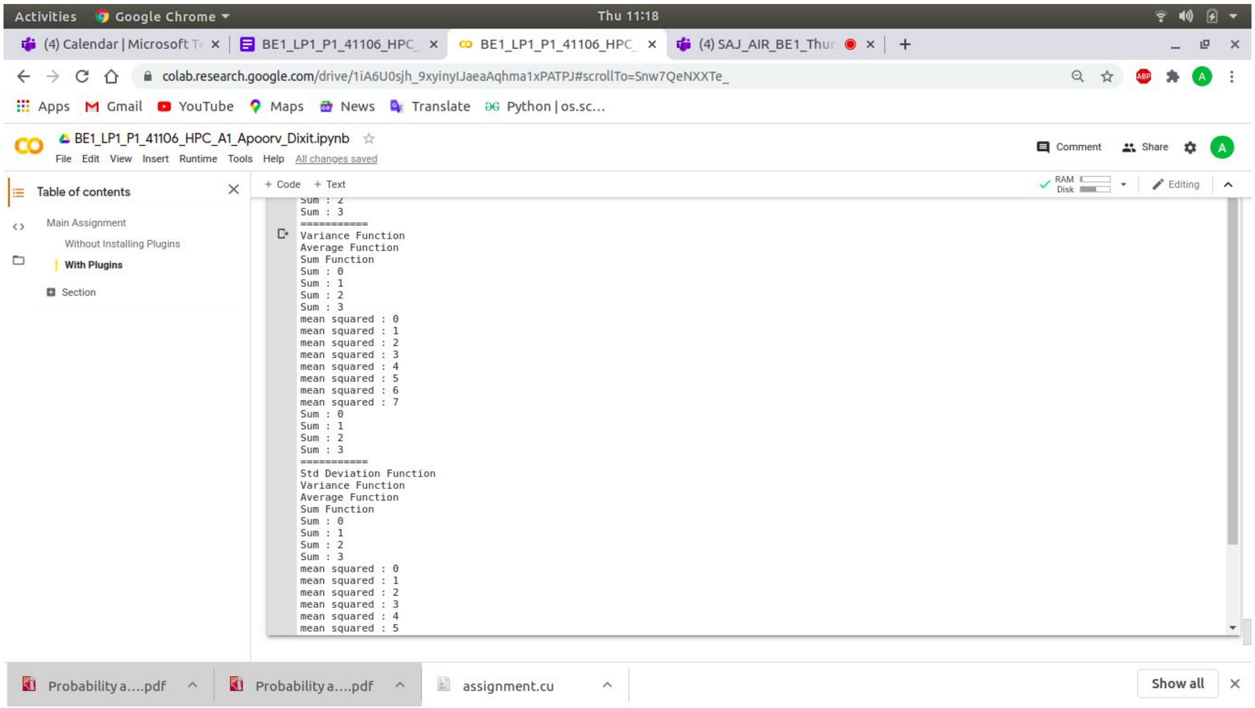
Automatic saving failed. This file was updated remotely or in another tab.

ThreadID and Function Output (Modification)



```
}
```

```
===========
Minimum Function
Min : 0
Min : 1
Min : 2
Min : 3
===========
Maximum Function
Max : 0
Max : 1
Max : 2
Max : 3
===========
Sum Function
Sum : 0
Sum : 1
Sum : 2
Sum : 3
===========
Average Function
Sum Function
Sum : 0
Sum : 1
Sum : 2
Sum : 3
===========
Variance Function
Average Function
```



```
Sum : 2
Sum : 3
===========
Variance Function
Average Function
Sum Function
Sum : 0
Sum : 1
Sum : 2
Sum : 3
mean squared : 0
mean squared : 1
mean squared : 2
mean squared : 3
mean squared : 4
mean squared : 5
mean squared : 6
mean squared : 7
Sum : 0
Sum : 1
Sum : 2
Sum : 3
===========
Std Deviation Function
Variance Function
Average Function
Sum Function
Sum : 0
Sum : 1
Sum : 2
Sum : 3
mean squared : 0
mean squared : 1
mean squared : 2
mean squared : 3
mean squared : 4
mean squared : 5
```

Conclusion

I have successfully divided problems into subproblems, learnt how to use GPUs and learnt how to solve sub problems using threads on GPU cores.