IT-477 Introduction to GPU Programming

Project Report

STATISTICAL ANALYSIS OF 2D DATA USING CORRELATION AND REGRESSION

Submitted by -

K.Rajiv Reddy (201301128)

Mallipeddi Akshay (201301216)

Section 1

Problem statement -

Given a set of 2D data in the form of 2-tuple (x,y), determine how closely related the two sets of variables are, using correlation coefficient. Also estimate the unknown set of y-values for given x-values using regression.

Mathematical details and equations -

Correlation coefficient - The correlation coefficient between two sample variables *x* and y is a scale-free measure of linear association between the two variables, and is given by the formula

$$r = cov(x,y)/s_x s_y$$

We can calculate correlation coefficient as :

$$\frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sqrt{\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right)\left(\sum_{i=1}^{n} y_i^2 - n\bar{y}^2\right)}}$$

- If *r* is close to 1 then *x* and y are positively correlated. A positive linear correlation means that high values of *x* are associated with high values of y and low values of x are associated with low values of y.

-If *r* is close to -1 then *x* and y are negatively correlated. A negative linear correlation means that high values of *x* are associated with low values of y, and low values of *x* are associated with high values of y.

-When *r* is close to 0 there is little linear relationship between *x* and y.

Regression line - The goal of regression analysis is to describe the relationship between two variables based on observed data and to predict the value of the dependent variable based on the value of the independent variable. Even though we can make such predictions, this doesn't imply that we can claim any causal relationship between the independent and dependent variables.

The regression line can be calculated as follows -

The formula for the slope, *m,* of the best-fitting line is

$$m = r\left(\frac{s_y}{s_x}\right)$$

where, s_x = standard deviation of x-values.,

s_y = standard deviation of y-values.

r= correlation coefficient

The formula for the *y*-intercept, *b*, of the best-fitting line is

$$b = \bar{y} - m\bar{x}, \text{ where } \bar{x} \text{ and } \bar{y}$$

are the means of the *x*-values and the *y*-values, respectively, and *m* is the slope.

And hence, the line equation is given by y=mx+b.

Example -

Input -

Sample size: 10

x	y
934.049255	235.657532
571.251099,	269.530273
474.735901,	496.427185
824.942505,	902.275635
898.403870,	784.847229
142.320221,	798.319702
627.300110,	156.531525
815.758423,	800.895508
382.410736,	677.940857
432.310303,	533.090210

Output -

Correlation coefficient = -0.060404692113968

Mean x ($\bar{x}$): 610.3482423

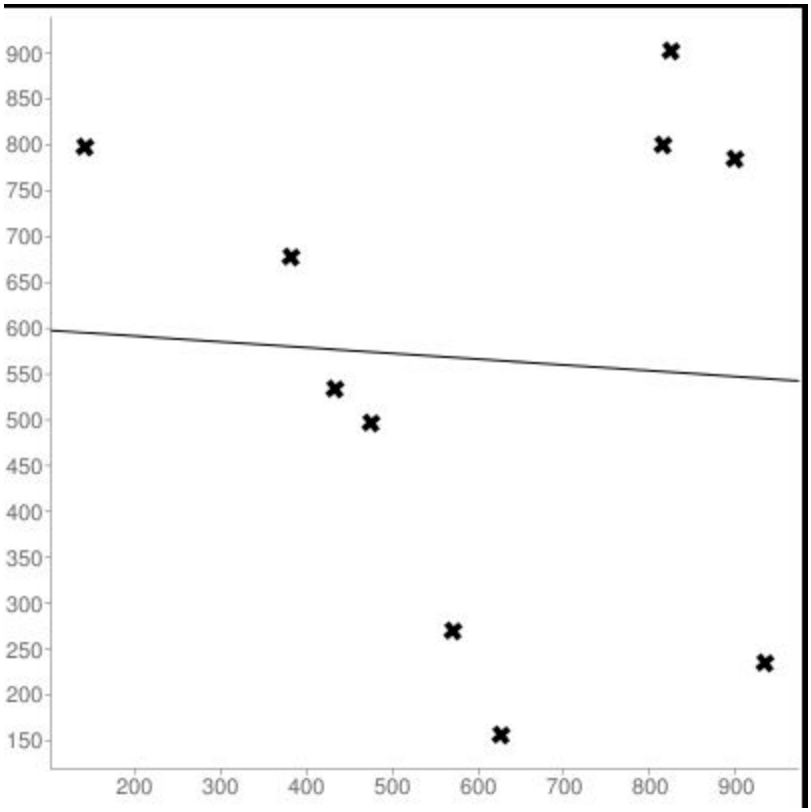
Mean y ($\bar{y}$): 565.5515656

Intercept of regression line : 604.05593783215

Slope of regression line: -0.063085906640853

Regression line equation: y=604.05593783215-0.063085906640853x

Scatterplot with regression line -



Estimated y-values for a few x-values based on calculated regression line -

x	y
100.00	597.7473471680647
200.00	591.4387565039794
400.00	578.8215751758088

Note: These are just **estimated values** based on regression analysis done on the input data set.

Applications:-

-Evaluating Trends and Sales Estimates

Linear regressions can be used in business to evaluate trends and make estimates or forecasts. For example, if a company's sales have increased steadily every month for the past few years, conducting a linear analysis on the sales data with monthly sales on the y-axis and time on the x-axis would produce a line that that depicts the upward trend in sales. After creating the trend line, the company could use the slope of the line to forecast sales in future months.

-Analyzing the Impact of Price Changes

Linear regression can also be used to analyze the effect of pricing on consumer behavior. For instance, if a company changes the price on a certain product several times, it can record the quantity it sells for each price level and then perform a linear regression with quantity sold as the dependent variable and price as the explanatory variable. The result would be a line that depicts the extent to which consumers reduce their consumption of the product as prices increase, which could help guide future pricing decisions.

-As the temperature decreases, the speed at which molecules move decreases.

-As the speed of a wind turbine increases, the amount of electricity that is generated increases.

-References for further reading

https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient

Section 2

Data generation:- The data set for the values of x and y values(two arrays for each variable) are generated **uniformly** in the range 1 to 1000.The data set thus generated is written to a file.The whole project uses this file as a reference for the data.

Serial Code:-

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

*sumx+=x[i];
*sumy+=y[i];
*xiyi+=x[i]*y[i];
*xi2+=x[i]*x[i];
*yi2+=y[i]*y[i];

}

*xavg=*sumx/n;
*yavg=*sumy/n;

*crc = ((*xiyi)-(n*(*xavg)*(*yavg)))/(sqrt((*xi2)-n*(*xavg)*(*xavg))*sqrt((*yi2)-n*(*yavg)*(*yavg)));

printf("serial crc%f\n",*crc);

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

*sx+=(x[i]-*xavg)*(x[i]-*xavg);
*sy+=(y[i]-*yavg)*(y[i]-*yavg);

}
*sx=sqrt(*sx/n);
*sy=sqrt(*sy/n);
*m=(*crc)*(*sy/(*sx));
*b=(*yavg)-(*m)*(*xavg);
printf("serial regressiony=%fx+%f\n",*m,*b);
```

The calculation of **CRC** (Correlation coefficient) requires the reduced sum of all x-values(taken as **sumx** in code),reduced sum of all y-values(taken as **sumy** in code),reduced sum of pair wise multiplication of corresponding x-values(taken as **xi^2** in code),reduced sum of pair wise multiplication of corresponding y-values (taken as **yi^2** in code) and reduced sum of pair wise multiplication of corresponding x and y values (taken as **xiyi** in code).The averages are also calculated as **xavg** and **yavg** corresponding to sumx and sumy .These averages are used to calculate standard deviations required to calculate **Regression line** equation .

Complexity of the algorithm used:-

O(N) where N is the number of values generated in the file.Because a single for loop iterates through whole data only once.

Pseudo code:-

-First we generate data into a file .
-Then we read the corresponding data into two array namely x and y.
-Then these arrays are used to calculate sumx,sumy,xi^2,yi^2 and xiyi.
-After calculating these values CRC is calculated using the equation mentioned above.
-Similarly Regression line equation is calculated using the equation mentioned above.

Scope of parallelism:-

- As we can see there are lot of computations involved we can use parallel programming where several threads are used to calculate various parts in the serial code.
- As mentioned above we need various reduced sum values.We can use **reduction algorithm** as discussed in the class.

Sections of the code that can be parallelized:-

-The formula to compute correlation coefficient is

$$\frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sqrt{\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right)\left(\sum_{i=1}^{n} y_i^2 - n\bar{y}^2\right)}}$$

The computations which are independent of each other and can be computed parallely are

- $\sum xy$ (*sum of product of x and corresponding y − value*)

- $\sum x$

- $\sum y$

$\sum x2$ (*sum of squares of all x values*)

$\sum y$ (*sum of squares of all y values*)

-Calculating these sums can be parallelized using reduction algorithm.

Section 3

Naive parallel implementation

Strategy of parallelization

Use of global memory to access array elements and perform computations

The formula to compute correlation coefficient is

$$\frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sqrt{\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right)\left(\sum_{i=1}^{n} y_i^2 - n\bar{y}^2\right)}}$$

The computations which are independent of each other and can be computed parallely are

- $\sum xy$ (*sum of product of x and corresponding y − value*)

- $\sum x$

- $\sum y$

$\sum x2$ (*sum of squares of all x values*)

$\sum y$ (*sum of squares of all y values*)

The strategy used here is to launch threads, equal in number to the input array size, which will compute individual xy, xi2, yi2 and other necessary values, by loading x and y values from the global memory. Threads are then synced. This is followed by reduction of all of the computed values using the strategy discussed below.

CUDA C-code (pseudo-code) -

After data generation and reading from file, the following is executed.

1) Allocate memory of size 2 times input array size on the device using cudaMalloc.

2) Copy the 2 input arrays into the device using cudaMemcpy. Call them dev_x and dev_y.

3) Launch kernel with number of blocks equal to (n/1024)+1, each having 1024 threads, where n=input arrays size.

4) Each thread accesses the global memory and gets the x-value and y-value from the 2 input arrays stored based on its threadIdx (say x[tid] and y[tid]).

5) Each thread computes the following -

- xi2[tid]=x[tid]*x[tid]

- yi2[tid]=y[tid]*y[tid]

- xiyi[tid]=x[tid]*y[tid]

6) All threads are synced.

7) Starting with stride=(n-1), the following is executed -

x[tid]=x[tid]+x[stride-tid]

y[tid]=y[tid]+y[stride-tid];

xiyi[tid]=xiyi[tid]+xiyi[stride-tid];

xi2[tid]=xi2[tid]+xi2[stride-tid];

yi2[tid]=yi2[tid]+yi2[stride-tid];

Stride is halved at every step. This loop repeats until stride>=1.

8) This gives reduced sums of

- input array x

- input array y

- product of x and corresponding y value

- squares of x value

- squares of y value

10) All threads are synced.

11) Average x and y values are computed from the computed reduced sum, using which each thread calculates it's own contribution to the standard deviation of x and y.

12) All these individual contributions are summed up using the above strategy to get standard deviation of x and y.

13) All the necessary computed values are copied back into the host memory and substituted in the formula to calculate the correlation coefficient.

Kernel -

```
__global__ void naive(float* x, float * y, int n, float* sux, float* sumy, float* xiyi, float* xi2, float* yi2, float* sx, float* sy){
int index = blockIdx.x*blockDim.x + threadIdx.x;

float tempx=x[index];
float tempy=y[index];

if(index<n){
xi2[index]=tempx*tempx;
yi2[index]=tempy*tempy;
xiyi[index]=tempx*tempy;
__syncthreads();

for(int stride=n-1;stride>=1;stride=stride/2){
if(index<=stride/2 && index!=stride-index)
{
x[index]=x[index]+x[stride-index];
y[index]=y[index]+y[stride-index];
xiyi[index]=xiyi[index]+xiyi[stride-index];
xi2[index]=xi2[index]+xi2[stride-index];
yi2[index]=yi2[index]+yi2[stride-index];
}
__syncthreads();
}
float xavg = x[0]/n;
float yavg = y[0]/n;

sx[index]=(tempx-xavg)*(tempx-xavg);
sy[index]=(tempy-yavg)*(tempy-yavg);
__syncthreads();

for(int stride=n-1;stride>=1;stride=stride/2){
if(index<=stride/2 && index!=stride-index)
{
sx[index]=sx[index]+sx[stride-index];
sy[index]=sy[index]+sy[stride-index];
}
__syncthreads();
}
}
```

Optimised implementation -

Use of shared memory to access array elements and perform computations

- Consecutive blocks of 1024 elements from both the input arrays are loaded into the shared memory of each block.

- Once loaded, each thread in a block accesses data from the shared memory based on its thread index and performs computations in a manner similar to the naive implementation, followed by reduction to compute the required values.

- Each block will have it's own partial sum i.e. the kth block will compute sums of x-values, y-values, xiyi values, xi2 values and yi2 values with x and y values ranging from 1024*k th index to 1024*(k+1) th index.

- Number of blocks launched is equal to (n/1024)+1. For the first n/1024 blocks, stride=1024 for reduction. For the last block, stride value is initialised to n-1024*k, where k=number of blocks(since there are only as many elements loaded into the shared memory of the last block).

- All the partial sums calculated by each block are copied into the global memory. Total sum is obtained by summing each of the partial sums on the host.

-The sums hence obtained are used to calculate the correlation coefficient and the regression line.

Kernel -

```
__global__ void shared(float *a,float *b,int n,int k,float *sumx,float *sumy,float *xi2,float *yi2,float *xiyi,float *sumxi2,float *sumyi2, floa
t* sumxiyi){


    __shared__    float sharedx[1024];
    __shared__    float sharedy[1024];
                int tid= threadIdx.x;
               int i=blockIdx.x*blockDim.x+threadIdx.x;

if(i<n){
sharedx[tid]    = a[i];
sharedy[tid]    = b[i];
__syncthreads();
float tempx=a[i];
float tempy=b[i];
xi2[i]=tempx*tempx;
yi2[i]=tempy*tempy;
xiyi[i]=tempx*tempy;
 __syncthreads();
if(blockIdx.x==k){
for(int stride=n-1024*k-1;stride>=1;stride=stride/2){
if(tid<=stride/2 && tid!=stride-tid)
{
sharedx[tid]=sharedx[tid]+sharedx[stride-tid];
sharedy[tid]=sharedy[tid]+sharedy[stride-tid];
}


__syncthreads();
}
}
else
{     for(int stride=1023;stride>=1;stride=stride/2){

if(tid<=stride/2 && tid!=stride-tid)
{
sharedx[tid]=sharedx[tid]+sharedx[stride-tid];
sharedy[tid]=sharedy[tid]+sharedy[stride-tid];
}
__syncthreads();
```

```
}
}
if(tid==0){
sumx[blockIdx.x]=sharedx[0];
sumy[blockIdx.x]=sharedy[0];
                  }
       __syncthreads();

                              sharedx[tid]=xi2[i];

                              sharedy[tid]=yi2[i];


                                    __syncthreads();

                                 if(blockIdx.x==k){
                               for(int stride=n-1024*k-1;stride>=1;stride=stride/2){

                                    if(tid<=stride/2 && tid!=stride-tid)
                                      {
                                               sharedx[tid]=sharedx[tid]+sharedx[stride-tid];
                                               sharedy[tid]=sharedy[tid]+sharedy[stride-tid];
                                      }


                                              __syncthreads();
                                       }

                           }
                       else
                    {



                           for(int stride=1023;stride>=1;stride=stride/2){

                                    if(tid<=stride/2 && tid!=stride-tid)
                                      {
                                               sharedx[tid]=sharedx[tid]+sharedx[stride-tid];
                                               sharedy[tid]=sharedy[tid]+sharedy[stride-tid];
```

```
                                              __syncthreads();
                                }


                               }

         if(tid==0)
           {

                 sumxi2[blockIdx.x]=sharedx[0];
                 sumyi2[blockIdx.x]=sharedy[0];


           }
s();

         sharedx[tid]=xiyi[i];

        __syncthreads();

                        if(blockIdx.x==k){
             for(int stride=n-1024*k-1;stride>=1;stride=stride/2){

                          if(tid<=stride/2 && tid!=stride-tid)
                            {
                                     sharedx[tid]=sharedx[tid]+sharedx[stride-tid];

                            }


                                    __syncthreads();
                             }
                    }
                else
                   {
```

```
                for(int stride=1023;stride>=1;stride=stride/2){

                        if(tid<=stride/2 && tid!=stride-tid)
                          {
                                  sharedx[tid]=sharedx[tid]+sharedx[stride-tid];
                                  sharedy[tid]=sharedy[tid]+sharedy[stride-tid];
                          }


                                  __syncthreads();
                           }

                          }

        if(tid==0)
          {

                sumxiyi[blockIdx.x]=sharedx[0];

          }


}
}
```

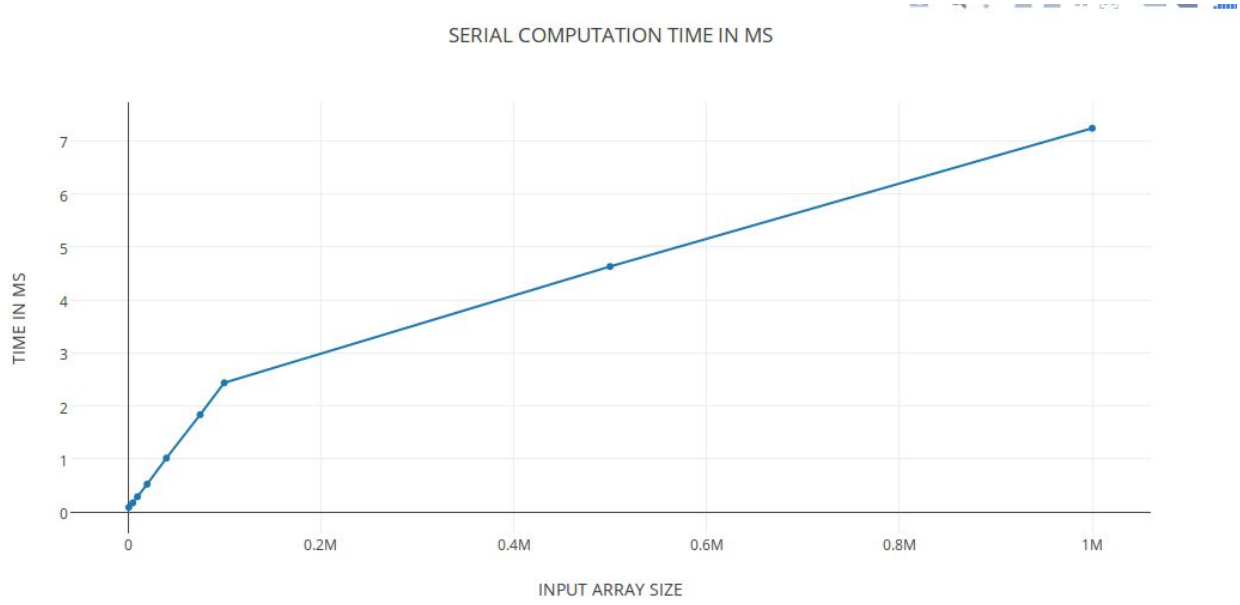
Section 4

Analysis for serial code:-

Input arrays size	Time taken for computation(ms)
1000	0.087680
5000	0.170464
10000	0.286944
20000	0.52428

40000	1.015712
75000	1.835584
100000	2.437696
500000	4.63052
1000000	7.2398

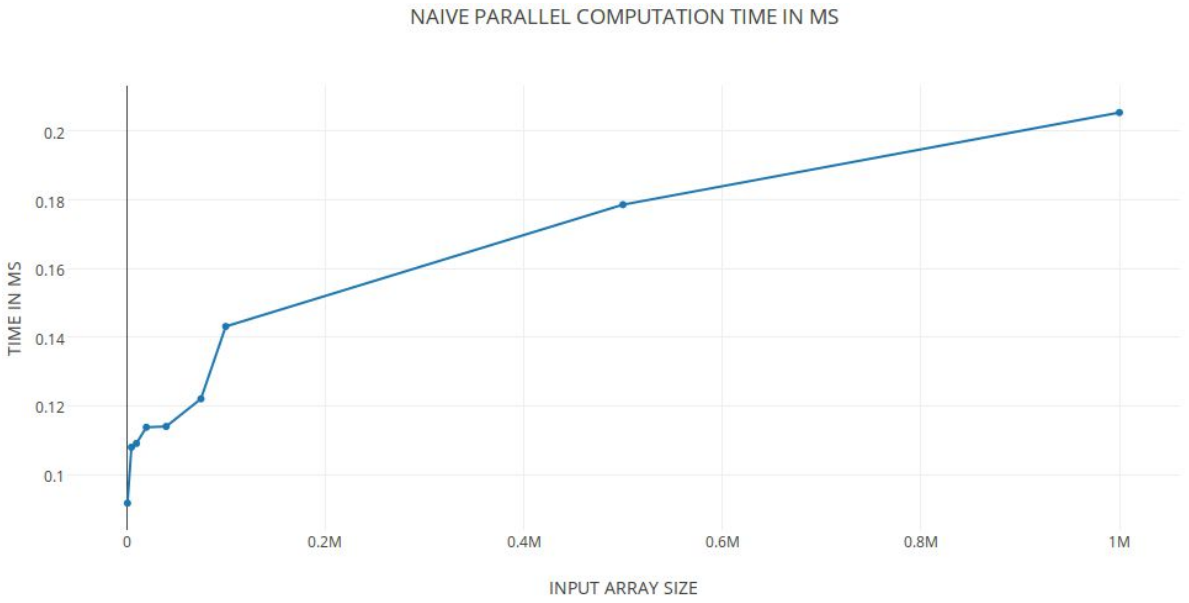
Graph for size v/s computation time:-



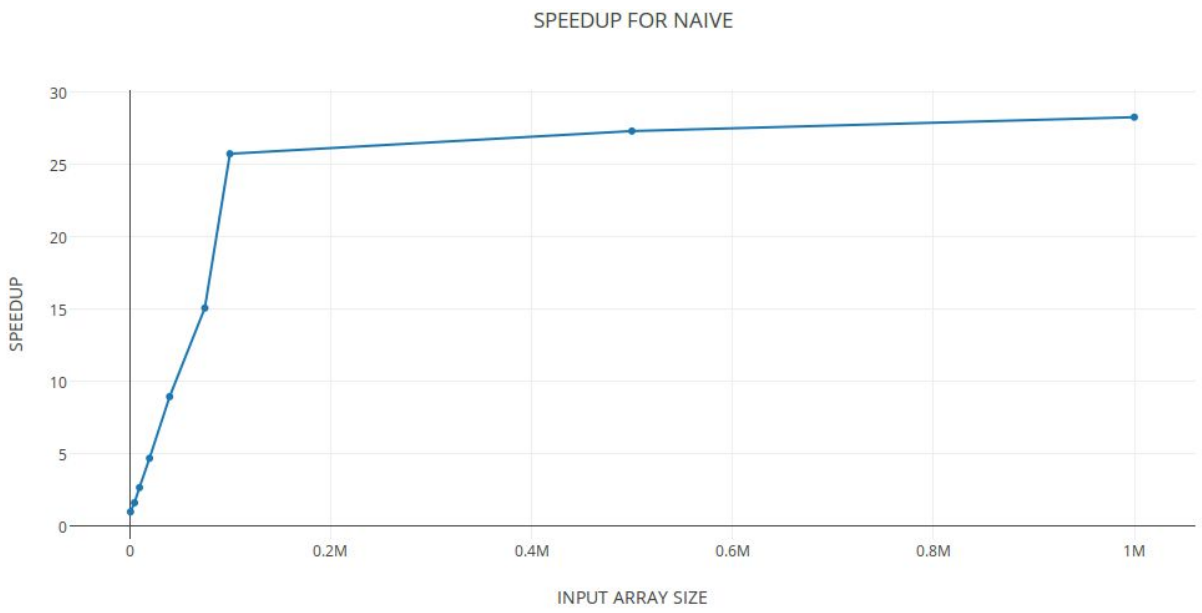
Analysis for naive parallel:-

Input arrays size	Time(ms)	Speed-up
1000	0.091648	0.95
5000	0.107936	1.579
10000	0.109056	2.631
20000	0.11376	4.652
40000	0.11400	8.9097
75000	0.122048	15.03
100000	0.143136	25.687
500000	0.17862	27.256
10000000	0.20547	28.214

Graph for size v/s computation time:-



Graph for size v/s speedup:-

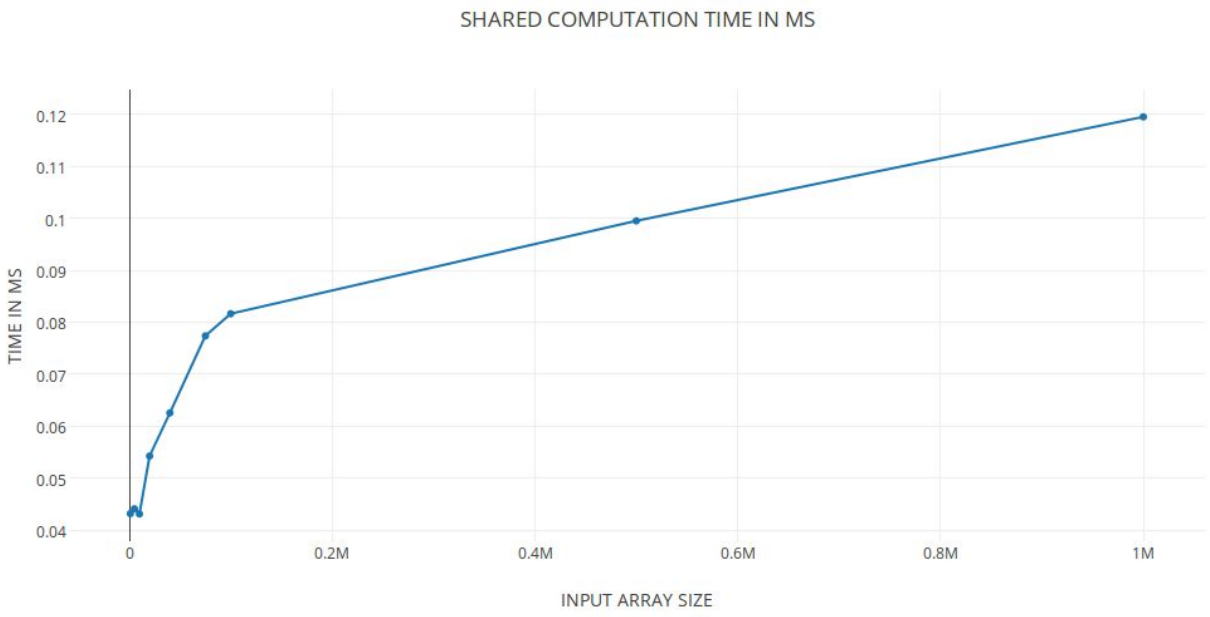


Analysis for shared implementation:-

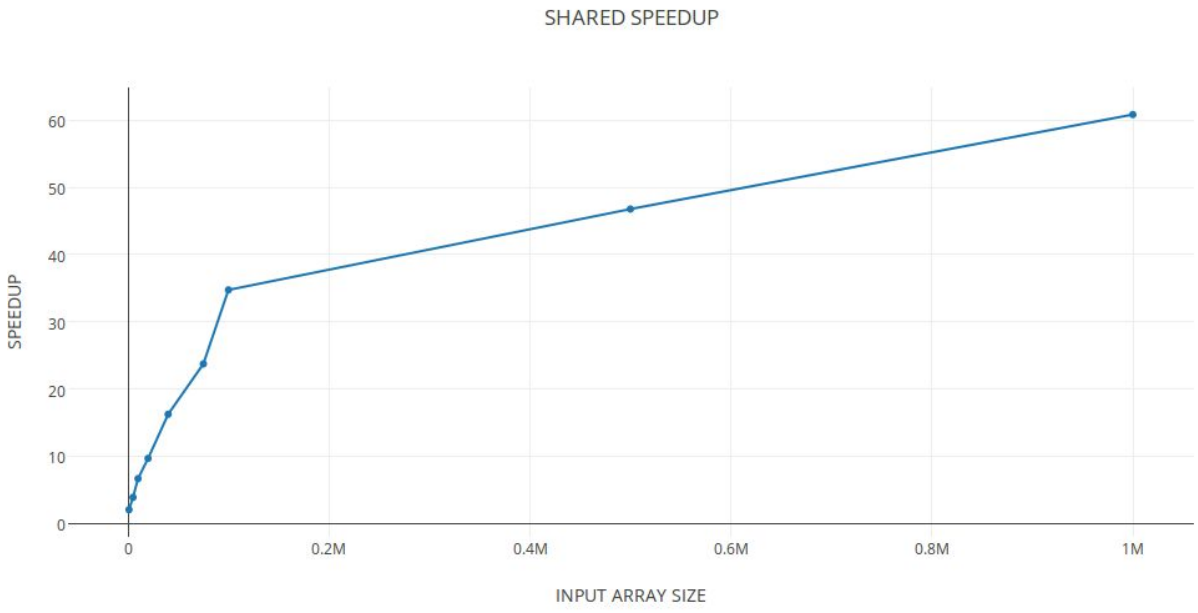
Input arrays size	Time taken(ms)	Speed-up
1000	0.043200	2.0296
5000	0.044128	3.8629
10000	0.043136	6.652
20000	0.054272	9.66
40000	0.062560	16.2358

75000	0.077376	23.7229
100000	0.08164	34.7412
500000	0.0995	46.772
1000000	0.1195	60.838

Graph for size v/s computation time:-

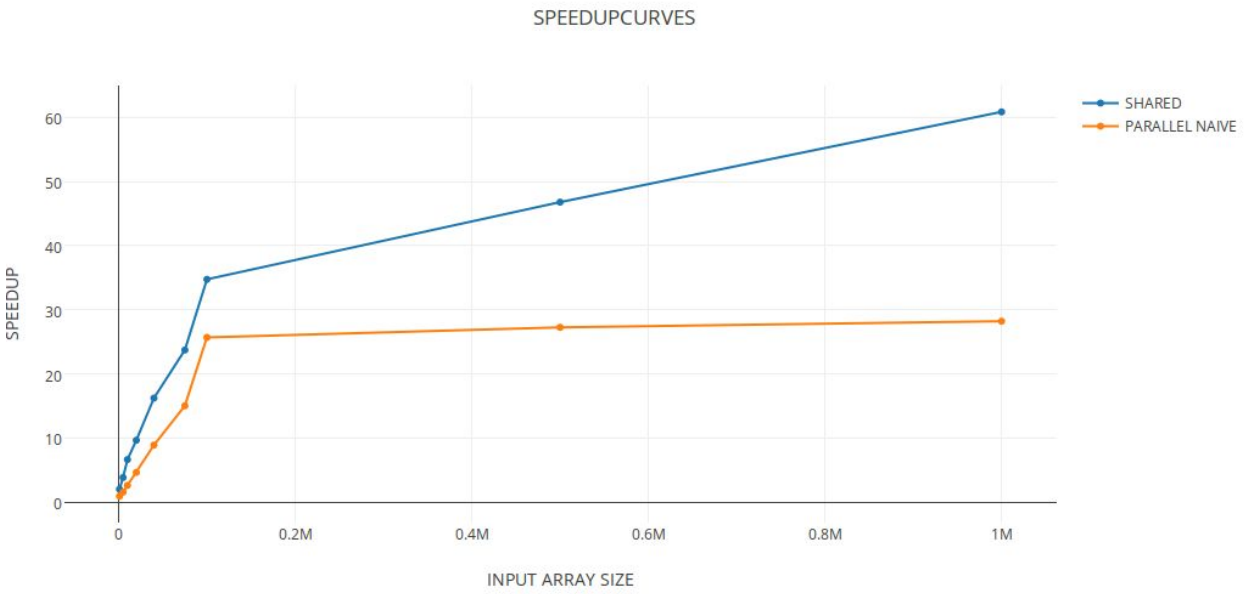


Graph for input size v/s speedup :-



Graph of times in different implementations:-

Graph of speedups for naive and shared:-



MAJOR OBSERVATIONS-

1. Effect of increasing problem size on serial code to show that it is a computationally expensive problem:-

-*sumx=*sumx+x[i]; (Takes one arithmetic addition)

*sumy=*sumy+y[i]; (Takes one arithmetic addition)

*xiyi=*xiyi+x[i]*y[i]; (Takes one arithmetic addition and one multiplication)

*xi2=*xi2+x[i]*x[i]; (Takes one arithmetic addition and one multiplication)

*yi2=*yi2+y[i]*y[i]; (Takes one arithmetic addition and one multiplication)

So total operations for each value of x and y are **8**.So the operations to input data ratio would be **8:1.** When we taken some millions of values then we need to do some extra millions of operations.By this we can say that it is a computationally expensive problem.

Hence it is an **embarrassingly parallel problem.** And parallelising the code will give us a better execution time than serial implementation.

2. On further implementation of shared-memory for memory accessing, i.e., in code rather than accessing the global array every time, the part of the global array which is going to be accessed is placed in the shared memory and then for calculating the crc for elements. When shared memory was used rather than global memory, Effect on speed-up was considerably good.