HIGH PERFORMANCE COMPUTING CUDA

Implementation of Durand Kerner Method to solve Polynomial Equations

Durand Kerner

$$r_{n+1} = r_n - \frac{p(r_n)}{(r_n - s_n)(r_n - t_n)}$$

$$s_{n+1} = s_n - \frac{p(s_n)}{(s_n - r_n)(s_n - t_n)}$$

$$t_{n+1} = t_n - \frac{p(t_n)}{(t_n - r_n)(t_n - s_n)}$$

-Report By:
Sharan SK
CED18I049

Serial Code:

```
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include<omp.h>

#define M_PI 3.14159265358979323846
#define coff_size 500

double R=0;
double complex z[coff_size];
double complex deltaZ[coff_size];
double deltaZMax;
double epsilon = 1e-6;
double complex QsubJ,fz;
int max_iter = 1000;


//----------------------Function Prototypes-------------------------------
void durand_kerner(); //Prototypes
void calc_theta();
double max_cof();
void printz();
void update_z();
void update_fz();


int main() {

   double complex cList[coff_size];  //List of coefficients
   double complex z;
   double x,y; //x for real and y for imaginary parts of the coefficient
   int n=0; //n is number degree of polynomial


//------Read Coefficients------------------------------------------
```

```
   printf("Enter coefficients and enter any char other than number when 
done:\n");
   while(scanf("%lf %lf",&x,&y) == 2)  { //Read coefficients from stdin
       cList[n] = (x + y*I);
       n++;
   }
   x = 1;  //Cn = 1, because the equation has to be normalized
   y = 0;
   z = (x + y*I);
   cList[n] = z; //Store in cList[]


   durand_kerner(cList,n);



}
//-----------------------------------Function 
Definition-------------------------------


void durand_kerner(double complex cList[],int n) {
   float st;
   st=omp_get_wtime();
   R = 1 + max_cof(cList,n);  //End Equation 5

   calc_theta(n);
   int k;
   for(k=1;k <= max_iter;k++) {

       //printz(cList,n,k);

       deltaZMax = 0;

       update_fz(cList,n);
       update_z(n);

       if(deltaZMax <= epsilon) {
```

```
                break;
            }

        }
        st=omp_get_wtime()-st;
        printf("%d\n",k);
        printz(cList,n);
        printf("Time Taken=%f\n",st);

}

void calc_theta(int n) {
        for(int j=0;j < n;j++) {
            z[j] = ( cos( j*((2*M_PI)/n) ) + (I*sin( j*((2*M_PI)/n) )) )*R;
        }

}

double max_cof(double complex cList[],int n)
{
        double r;
        for(int j=0;j < n;j++) {
            if(cabs(cList[j]) > R) {
                r = cabs(cList[j]);
            }
        }

        return r;
}

void printz(double complex cList[],int n)
{
            printf("Final Output:(Note: if the roots repeat then there exist
less than n-1 roots for the equation)\n");
            for(int i=0;i < n;i++) {
                            printf("z[%d] = %0.10f +
%0.10f*I\n",i,creal(z[i]),cimag(z[i]));
```

```
                fflush(stdout);
            }
}

void update_z(int n)
{
   for(int j=0;j < n;j++) {
           z[j] = z[j] + deltaZ[j];
       }
}

void update_fz(double complex cList[],int n)
{
   for(int j=0;j < n;j++) {

           QsubJ = 1;
           for(int i=0;i < n;i++) {
               if(i != j) {
                   QsubJ = (z[j]-z[i])*QsubJ;
               }
           }
           fz = 1;
           for(int k = n-1;k >= 0;k--) {
               fz = fz*z[j] + cList[k];

           }

           deltaZ[j] = (-fz/QsubJ);

           if(cabs(deltaZ[j]) > deltaZMax) {
               deltaZMax = cabs(deltaZ[j]);
           }
       }
}
```

Pseudo Code of the Algorithm:

```
Compute initial values $\{z_0, \ldots, z_{n-1}\}$ using Equation 6.
for $k = 1 \ldots k_{\max}$
  Let $\Delta z_{\max} = 0$.
  for $j = 0 \ldots n-1$
    Compute the product $Q_j = \prod_{\substack{i=0 \\ i \neq j}}^{n-1} (z_j - z_i)$ (Equation 3).
    Set $\Delta z_j = -f(z_j)/Q_j$.
    if $|\Delta z_j| > \Delta z_{\max}$
      Set $\Delta z_{\max} = |\Delta z_j|$.
  for $j = 0 \ldots n-1$
    Update $z_j = z_j + \Delta z_j$.
  if $\Delta z_{\max} \leq \epsilon$ quit.
```

Serial Code Explanation/Observations:

- ➢ GIven the Max No of iterations,the program will run for either that many no of iterations or until the maximum absolute value of deltaZ becomes less e^{-5}.
- ➢ From the profiling report ,it was observed that the update_z function has the lines that run most of the times.Now these lines are inside a for loop which runs upto ‘n’ which is the degree of the polynomial equation/input provided.
- ➢ Every root is stored inside complex array called ‘z’ which is updated by deltaZ value which is calculated by -fz/QsubJ,where QsubJ is the difference between every root other than the j^{th} root where j is the iterating variable/root and fz represents the result that is obtained by substituting the previous iteration’s calculated j^{th} root value.

- ➢ The initial values of the root are calculated using the formula below:

$$z_j^{(0)} = (\cos\theta_j + i\sin\theta_j)\cdot R, \quad j = 0,\ldots,n-1,$$

$$\theta_j = j\frac{2\pi}{n}.$$

Where ‘j’ represents the jth root of the equation and (0) represents that this value is initial value ,i.e of 0th iteration.

How can the program be parallelized in CUDA?:

- ➢ From the Above pseudo code ,we can see that the every iteration’s root value calculation depends on the previous root values meaning $Z_j^{(k)}$,which is jth root of the equation calculated in Kth iteration depends on $Z_j^{(k-1)}$ i.e on the previous iteration values .Hence the outer loop cannot be parallelized.
- ➢ It is also observed that the inner for loop i.e update_fz function depends on the previous Z values but calculates the root independently in each of the iteration ,which means if the threads wait until deltaZ is calculated and Z values of Kth iteration are updated then for every Kth iteration ,the roots can be parallely calculated at the same time .
- ➢ The calculation of roots can be done in the GPU function by using threads to independently calculate the deltaZ values.

CUDA Code Changes:

```
//----------------------------------------------------DK
Function--------------------------------

void durand_kerner(complex<double> cList[],int n) {

 R = 1 + max_cof(cList,n);  //End Equation 5
```

```
float time = 0,total=0;
calc_theta(n);
int k=0;
  cudaEvent_t start, stop;
  float elapsedTime;
  cuDoubleComplex *d_a, *d_b,*d_c;
  int size = n*sizeof(cList[0]);

  for(int j=0;j<n;j++)
      {
          z[j]=z[j]+deltaZ[j];
      }


  cudaMalloc((void **)&d_a, size);
   cudaMemcpy(d_a, &cList2, size, cudaMemcpyHostToDevice);

  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);

  cudaEventCreate(&start);
  cudaEventRecord(start,0);

  for(int i=0;i<max_iter;i++)
  {
      k+=1;
      deltaZMax=0;
      cudaMemcpy(d_b, &z, size, cudaMemcpyHostToDevice);
      calc_delta<<<n/threads + 1 ,threads>>>(d_a,d_b,d_c);
      cudaDeviceSynchronize();
      cudaMemcpy(&deltaZ, d_c, size, cudaMemcpyDeviceToHost);

      for(int j=0;j<n;j++)
      {
          z[j]=z[j]+deltaZ[j];
          if(abs(deltaZ[j]) > deltaZMax)
          {
             deltaZMax = abs(deltaZ[j]);
           }
       }
     if(deltaZMax <= epsilon)
     {
         break;
     }

   }
   cudaEventCreate(&stop);
   cudaEventRecord(stop,0);
   cudaEventSynchronize(stop);
   cudaEventElapsedTime(&elapsedTime, start,stop);
   printf("No of Threads=%d\nNo of iterations=%d\nElapsed time (in
seconds): %f\n" ,threads,k,elapsedTime/1000);

  //printz(cList,n);
  printfile(cList,n,k,elapsedTime);

}

//----------------------------------------------------DK
Function----------------------------------------

__global__ void calc_delta(cuDoubleComplex *a,cuDoubleComplex
*b,cuDoubleComplex *c)
{

   int j=threadIdx.x+blockIdx.x*blockDim.x;

   cuDoubleComplex QsubJ = make_cuDoubleComplex(1,0);
   cuDoubleComplex mo=make_cuDoubleComplex(-1,0);
    for(int i=0;i < nsize;i++) {

     if(i != j)
     {
         cuDoubleComplex b1=cuCsub(b[j],b[i]);
         QsubJ =cuCmul(QsubJ,b1);
      }
   }

   cuDoubleComplex fz =make_cuDoubleComplex(1,0);
   for(int k = nsize-1;k >= 0;k--)
   {
     //printf("a[%d] = %0.10f +
%0.10f*I\n",k,cuCreal(a[k]),cuCimag(a[k]));
     cuDoubleComplex a1=cuCmul(fz,b[j]);
     fz = cuCadd(a1, a[k]);
   }
   c[j]=cuCdiv(cuCmul(mo,fz),QsubJ);
}
```

How is the Parallelization achieved in CUDA?:

- In Cuda ,The coefficients of the cList and the roots are copied onto cuda Memory and are send into gpu function named “calc_delta” along with a empty cuda Memory of size ‘n’ variable to store all the updated deltaZ values where n is the degree of the polynomial.
- Each of the thread accesses its corresponding index and calculates the deltaZ value by finding QsubJ and fz using the same formula and for loop procedure.
- Cuda Emphasizes on using cuDoubleComplex data type as an alternative to double complex in c and cuda allows only cuda inbuilt functions to be called from the GPU function for which cuCadd,cuCmul,cuCadd and cuCdiv had been used to perform arithmetic operations on complex numbers.
- Once all the thread’s deltaZ are synchronized using cudaDeviceSynchronize() the z values are updated using the deltaZ value calculated and the updated z value is send back to the GPU function for the next iteration.

- ➢ This process continues until the change in roots becomes constant or max number of iterations are reached.

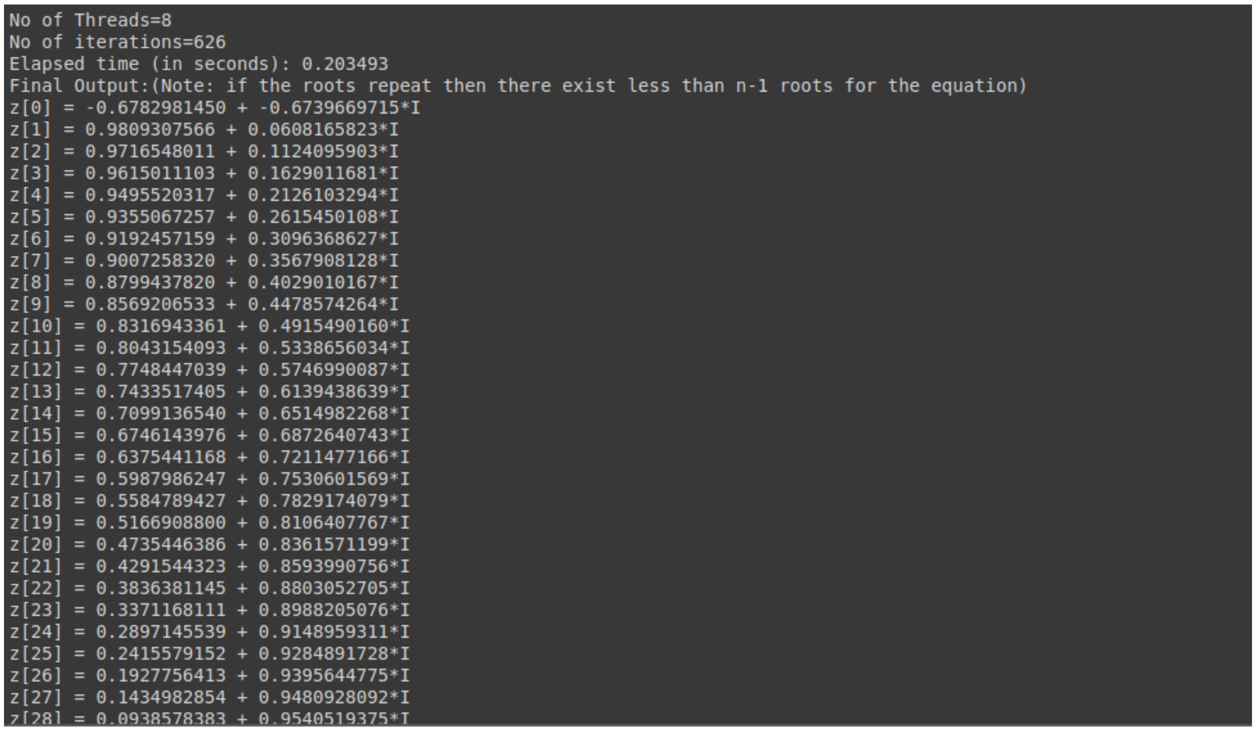
Observations on Execution time of the program:

Note :

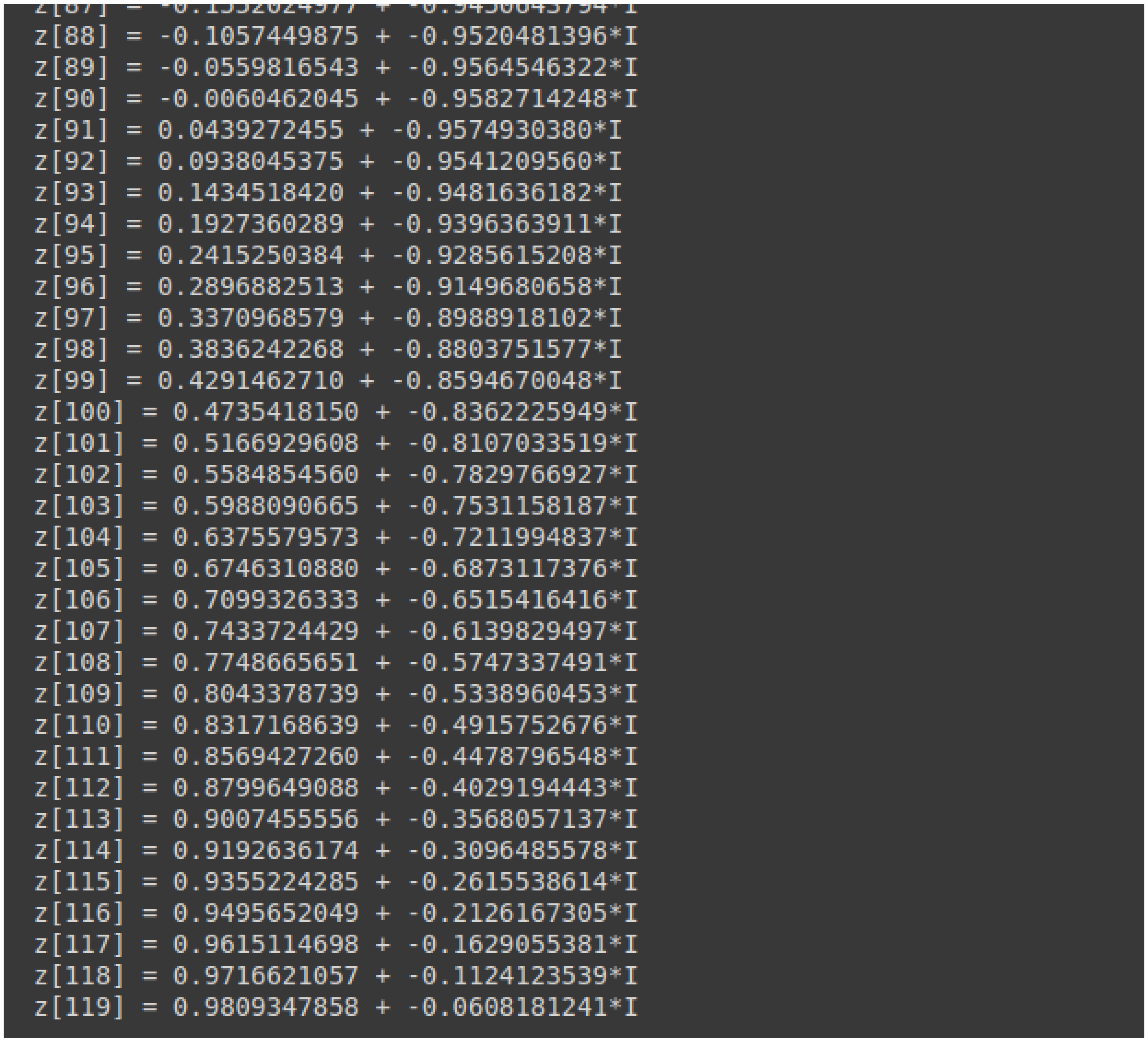
- ➢ CUDA doesn’t perform for polynomials of degree greater than 140.Hence for this project it is advised to run for degree polynomials less than 120.
- ➢ Cuda is also stable like MPI ,since it is being run on colab ,the full functionalities of the program hadn’t been explored.
- ➢ The Execution Time and the no of iterations that the program runs are more or less remains same with increase in threads

Output::

Input is generated using a for loop for degree=120 and the values of each coefficient are (index+1) + (index+1)*I.



```
No of Threads=8
No of iterations=626
Elapsed time (in seconds): 0.203493
Final Output:(Note: if the roots repeat then there exist less than n-1 roots for the equation)
z[0] = -0.6782981450 + -0.6739669715*I
z[1] = 0.9809307566 + 0.0608165823*I
z[2] = 0.9716548011 + 0.1124095903*I
z[3] = 0.9615011103 + 0.1629011681*I
z[4] = 0.9495520317 + 0.2126103294*I
z[5] = 0.9355067257 + 0.2615450108*I
z[6] = 0.9192457159 + 0.3096368627*I
z[7] = 0.9007258320 + 0.3567908128*I
z[8] = 0.8799437820 + 0.4029010167*I
z[9] = 0.8569206533 + 0.4478574264*I
z[10] = 0.8316943361 + 0.4915490160*I
z[11] = 0.8043154093 + 0.5338656034*I
z[12] = 0.7748447039 + 0.5746990087*I
z[13] = 0.7433517405 + 0.6139438639*I
z[14] = 0.7099136540 + 0.6514982268*I
z[15] = 0.6746143976 + 0.6872640743*I
z[16] = 0.6375441168 + 0.7211477166*I
z[17] = 0.5987986247 + 0.7530601569*I
z[18] = 0.5584789427 + 0.7829174079*I
z[19] = 0.5166908800 + 0.8106407767*I
z[20] = 0.4735446386 + 0.8361571199*I
z[21] = 0.4291544323 + 0.8593990756*I
z[22] = 0.3836381145 + 0.8803052705*I
z[23] = 0.3371168111 + 0.8988205076*I
z[24] = 0.2897145539 + 0.9148959311*I
z[25] = 0.2415579152 + 0.9284891728*I
z[26] = 0.1927756413 + 0.9395644775*I
z[27] = 0.1434982854 + 0.9480928092*I
z[28] = 0.0938578383 + 0.9540519375*I
```

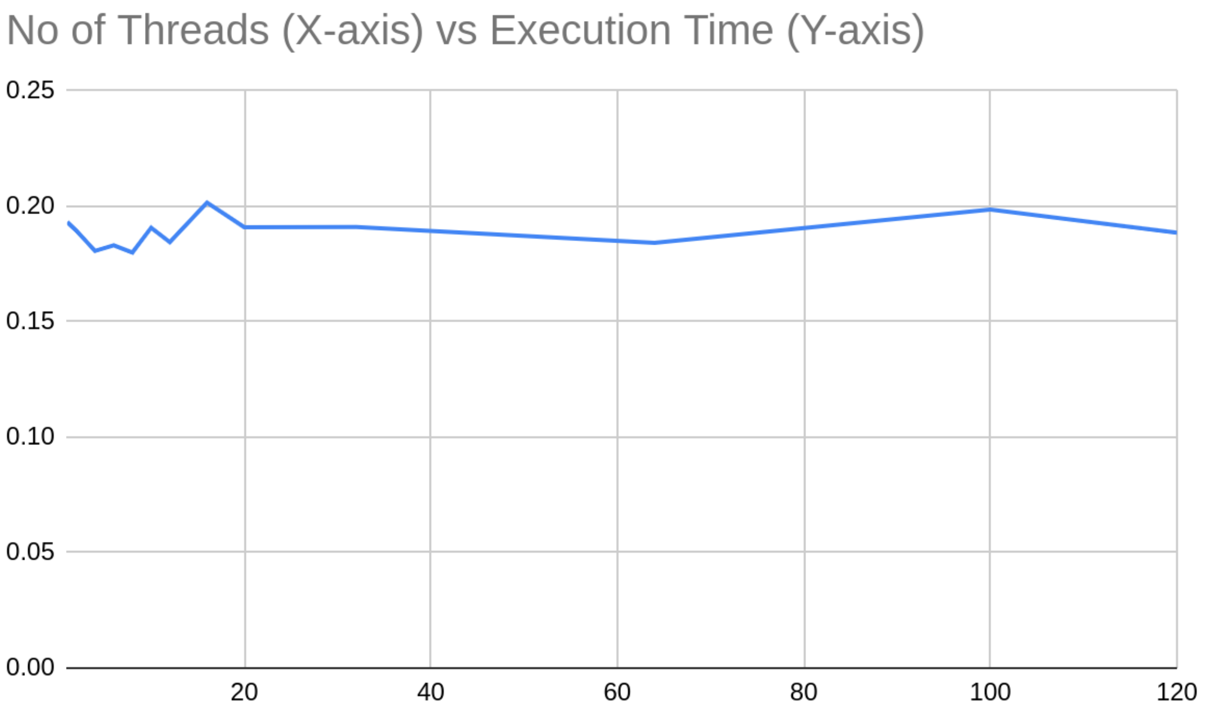


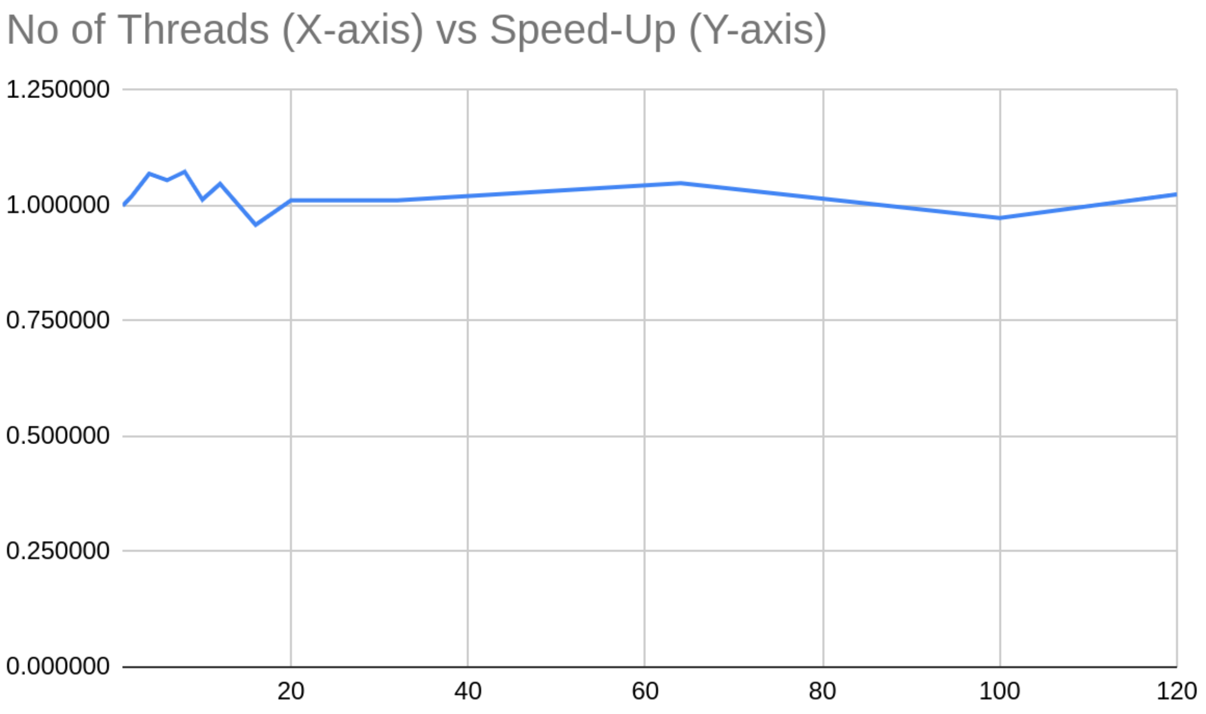
```
 z[87] = -0.1552024977 + -0.9450043794*I
 z[88] = -0.1057449875 + -0.9520481396*I
 z[89] = -0.0559816543 + -0.9564546322*I
 z[90] = -0.0060462045 + -0.9582714248*I
 z[91] = 0.0439272455 + -0.9574930380*I
 z[92] = 0.0938045375 + -0.9541209560*I
 z[93] = 0.1434518420 + -0.9481636182*I
 z[94] = 0.1927360289 + -0.9396363911*I
 z[95] = 0.2415250384 + -0.9285615208*I
 z[96] = 0.2896882513 + -0.9149680658*I
 z[97] = 0.3370968579 + -0.8988918102*I
 z[98] = 0.3836242268 + -0.8803751577*I
 z[99] = 0.4291462710 + -0.8594670048*I
 z[100] = 0.4735418150 + -0.8362225949*I
 z[101] = 0.5166929608 + -0.8107033519*I
 z[102] = 0.5584854560 + -0.7829766927*I
 z[103] = 0.5988090665 + -0.7531158187*I
 z[104] = 0.6375579573 + -0.7211994837*I
 z[105] = 0.6746310880 + -0.6873117376*I
 z[106] = 0.7099326333 + -0.6515416416*I
 z[107] = 0.7433724429 + -0.6139829497*I
 z[108] = 0.7748665651 + -0.5747337491*I
 z[109] = 0.8043378739 + -0.5338960453*I
 z[110] = 0.8317168639 + -0.4915752676*I
 z[111] = 0.8569427260 + -0.4478796548*I
 z[112] = 0.8799649088 + -0.4029194443*I
 z[113] = 0.9007455556 + -0.3568057137*I
 z[114] = 0.9192636174 + -0.3096485578*I
 z[115] = 0.9355224285 + -0.2615538614*I
 z[116] = 0.9495652049 + -0.2126167305*I
 z[117] = 0.9615114698 + -0.1629055381*I
 z[118] = 0.9716621057 + -0.1124123539*I
 z[119] = 0.9809347858 + -0.0608181241*I
```

Observation:

Execution Time vs Number of Threads:

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.193184	1.000000	
2	0.189429	1.019823	0.038875
4	0.18068	1.069205	0.086301
6	0.183124	1.054935	0.062490
8	0.17999	1.073304	0.078054
10	0.190682	1.013121	0.014390
12	0.184484	1.047159	0.049129
16	0.20156	0.958444	-0.046248
20	0.190962	1.011636	0.012107
32	0.191023	1.011313	0.011547
64	0.184211	1.048710	0.047185
100	0.198537	0.973038	-0.027989
120	0.188577	1.024430	0.024048


No of Threads (X-axis) vs Execution Time (Y-axis)
0.25
0.20
0.15
0.10
0.05
0.00
20
40
60
80
100
120


No of Threads (X-axis) vs Speed-Up (Y-axis)
1.250000
1.000000
0.750000
0.500000
0.250000
0.000000
20
40
60
80
100
120

Inference:

➢ The Speedup of the program does not increase or decrease much for 120 degree polynomials.As the threads increase speedup increases by a marginal amount up to 8 threads and the max speedup achieved is 1.073304 with a parallelization fraction as 0.078054.

➢ The Execution time follows the similar pattern as in speedup with marginal increase or decrease as we increase threads.

➢ Even Though the code contains communication overhead for copying roots from GPU memory to CPU memory during every iteration ,the program still runs on par with the serial code of the same project.

Complete CUDA Parallel Code:

```
%%cu
#include <complex>
#include <stdio.h>
#include <math.h>
#include<complex.h>
#include <cuComplex.h>
#define M_PI 3.14159265358979323846
#define coff_size 500
#define threads 8
using namespace std;


//----------------------------------------------------Complex
Variables---------------------------------
__managed__ int nsize;


double R=0;
complex<double> z[coff_size];
complex<double> deltaZ[coff_size];
```

```
__managed__  double deltaZMax;
complex<double> cList2[coff_size];

double epsilon = 1e-6;
//complex<double> QsubJ,fz;
int max_iter = 800;



//----------------------------------------------------Complex
Variables---------------------------------


//--------------------------------------------------Function
Prototypes--------------------------------
void durand_kerner(complex<double> cList[],int n); //Prototypes
void calc_theta(int n);
double max_cof(complex<double> cList[],int n);
void printz(complex<double> cList[],int n);
void update_fz(complex<double> cList[],int n,int o);
void printfile(complex<double> cList[],int n,int k,float st);


//--------------------------------------------------Function
Prototypes--------------------------------

//--------------------------------------------------GPU
Function---------------------------------------

__global__ void calc_delta(cuDoubleComplex *a,cuDoubleComplex
*b,cuDoubleComplex *c)
{

   int j=threadIdx.x+blockIdx.x*blockDim.x;

   cuDoubleComplex QsubJ = make_cuDoubleComplex(1,0);
   cuDoubleComplex mo=make_cuDoubleComplex(-1,0);
    for(int i=0;i < nsize;i++) {

     if(i != j)
     {
         cuDoubleComplex b1=cuCsub(b[j],b[i]);
         QsubJ =cuCmul(QsubJ,b1);
     }
   }

   cuDoubleComplex fz =make_cuDoubleComplex(1,0);
   for(int k = nsize-1;k >= 0;k--)
   {
     //printf("a[%d] = %0.10f +
%0.10f*I\n",k,cuCreal(a[k]),cuCimag(a[k]));
     cuDoubleComplex a1=cuCmul(fz,b[j]);
     fz = cuCadd(a1, a[k]);
   }
   c[j]=cuCdiv(cuCmul(mo,fz),QsubJ);
}

//---------------------------------------------------GPU
Function---------------------------------------

//-----------------------------------------------------Main----------------
-----------------

int main() {
  complex<double> cList[coff_size];
 complex<double> z;
 double x,y; //x for real and y for imaginary parts of the coefficient
 int n=0; //n is number degree of polynomial



//------Read Coefficients------------------------------------------------
```

```
 n=120;
 for(int i=0;i<n;i++ )
 {
     cList[i]=complex<double>(i+1,i+1);
 }

  nsize=n;

 cList[n] = complex<double>(1,0); //Store in cList[]

 if(n>=threads)
 durand_kerner(cList,n);
 else
 printf("No of Threads> No of Blocks,hence  program terminated");

}

//-----------------------------------------------------Main----------------
------------------------------

//-----------------------------------------------------DK
Function---------------------------------

void durand_kerner(complex<double> cList[],int n) {

 R = 1 + max_cof(cList,n);  //End Equation 5
 float time = 0,total=0;
 calc_theta(n);
 int k=0;
   cudaEvent_t start, stop;
   float elapsedTime;
   cuDoubleComplex *d_a, *d_b,*d_c;
   int size = n*sizeof(cList[0]);

   for(int j=0;j<n;j++)
       {
           z[j]=z[j]+deltaZ[j];
```

```
    }



cudaMalloc((void **)&d_a, size);
 cudaMemcpy(d_a, &cList2, size, cudaMemcpyHostToDevice);

cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

cudaEventCreate(&start);
cudaEventRecord(start,0);

for(int i=0;i<max_iter;i++)
{
    k+=1;
    deltaZMax=0;
    cudaMemcpy(d_b, &z, size, cudaMemcpyHostToDevice);
    calc_delta<<<n/threads + 1 ,threads>>>(d_a,d_b,d_c);
    cudaDeviceSynchronize();
    cudaMemcpy(&deltaZ, d_c, size, cudaMemcpyDeviceToHost);

    for(int j=0;j<n;j++)
    {
        z[j]=z[j]+deltaZ[j];
        if(abs(deltaZ[j]) > deltaZMax)
        {
          deltaZMax = abs(deltaZ[j]);
        }
    }
  if(deltaZMax <= epsilon)
  {
      break;
  }

}
cudaEventCreate(&stop);
cudaEventRecord(stop,0);
```

```
   cudaEventSynchronize(stop);
   cudaEventElapsedTime(&elapsedTime, start,stop);
   printf("No of Threads=%d\nNo of iterations=%d\nElapsed time (in
seconds): %f\n" ,threads,k,elapsedTime/1000);

  printz(cList,n);
  printfile(cList,n,k,elapsedTime);

}

//-----------------------------------------------------DK
Function----------------------------------------

//-----------------------------------------------------Auxiliary
Function---------------------------------

void calc_theta(int n) {
 for(int j=0;j < n;j++) {
       z[j]=complex<double> (cos(  j*((2*M_PI)/n) )*R,sin(  j*((2*M_PI)/n)
)*R);
 }

}

double max_cof(complex<double> cList[],int n)
{
 double r;
 for(int j=0;j < n;j++) {
     cList2[j]=cList[j];
   if(abs(cList[j]) > R) {
     r = abs(cList[j]);
   }
 }
 return r;
}

void printz(complex<double> cList[],int n)
```

```
{
   printf("Final Output:(Note: if the roots repeat then there exist less
than n-1 roots for the equation)\n");
   for(int i=0;i < n;i++) {
                 printf("z[%d] = %0.10f +
%0.10f*I\n",i,real(z[i]),imag(z[i]));
               fflush(stdout);
         }
}

void printfile(complex<double> cList[],int n,int k,float st)
{
   FILE *fp;
     fp = fopen("project_roots.txt", "w");
   fprintf(fp,"Durand Kerner Serial Algorithm:\n");
   fprintf(fp,"Max Iteration=%d\n",k);
   fprintf(fp,"Time Taken=%f\n",st);
   fprintf(fp,"Final Output:(Note: if the roots repeat then there exist
less than n-1 roots for the equation)\n");
   for(int i=0;i < n;i++) {
                 fprintf(fp,"z[%d] = %0.10f +
%0.10f*I\n",i,real(z[i]),imag(z[i]));
               fflush(stdout);
         }
   fclose(fp);
}
//-----------------------------------------------------Auxiliary
Function---------------------------------
```