3) THE MANDELBROT SET AND ITS USE AS BENCHMARK

3.1 The Mandelbrot set

The Mandelbrot set is a mathematical set of points whose boundary is a distinctive and easily recognizable two-dimensional fractalshape. The set is closely related to Julia sets (which include similarly complex shapes), and is named after the mathematician Benoit Mandelbrot, who studied and popularized it.

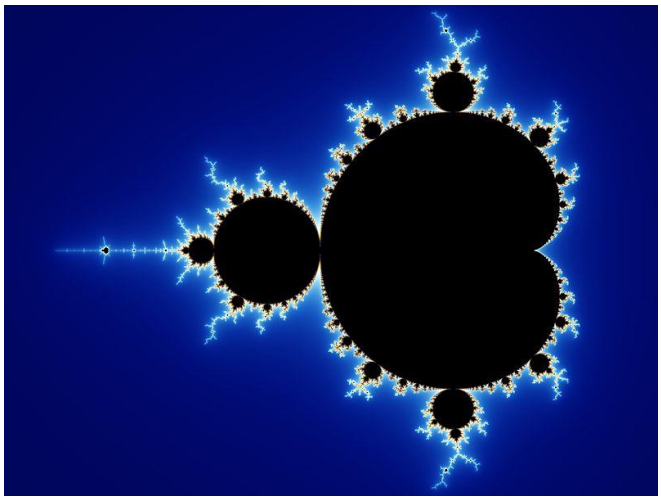
Mandelbrot set images are made by sampling complex numbers and determining for each whether the result tends towards infinity when a particular mathematical operation is iterated on it. Treating the real and imaginary parts of each number as image coordinates, pixels are colored according to how rapidly the sequence diverges, if at all.

More precisely, the Mandelbrot set is the set of values of c in the complex plane for which the orbit of 0 under iteration of the complex quadratic polynomial

$$z_{n+1} = z_n^2 + c$$

remains bounded. That is, a complex number c is part of the Mandelbrot set if, when starting with $z_0 = 0$ and applying the iteration repeatedly, the absolute value of z_n remains bounded however large n gets.

For example, letting c = 1 gives the sequence 0, 1, 2, 5, 26,…, which tends to infinity. As this sequence is unbounded, 1 is not an element of the Mandelbrot set. On the other hand, c = −1 gives the sequence 0, −1, 0, −1, 0,…, which is bounded, and so −1 belongs to the Mandelbrot set.
Images of the Mandelbrot set display an elaborate boundary that reveals progressively ever-finer recursive detail at increasing magnifications. The "style" of this repeating detail depends on the region of the set being examined. The set's boundary also incorporates smaller versions of the main shape, so the fractal property of self-similarity applies to the entire set, and not just to its parts.
The Mandelbrot set has become popular outside mathematics both for its aesthetic appeal and as an example of a complex structure arising from the application of simple rules, and is one of the best-known examples of mathematical visualization.



3.2 The use of Mandelbrot set as a potential benchmark

The simplest algorithm for generating a representation of the Mandelbrot set is known as the "escape time" algorithm. A repeating calculation is performed for each x, y point in the plot area and based on the behavior of that calculation, a color is chosen for that pixel.

The x and y locations of each point are used as starting values in a repeating, or iterating calculation (described in detail below). The result of each iteration is used as the starting values for the next. The values are checked during each iteration to see if they have reached a critical 'escape' condition or 'bailout'. If that condition is reached, the calculation is stopped, the pixel is drawn, and the next x, y point is examined. For some starting values, escape occurs quickly, after only a small number of iterations. For starting values very close to but not in the set, it may take hundreds or thousands of iterations to escape. For values within the Mandelbrot set, escape will never occur. The programmer or user must choose how much iteration, or 'depth,' they wish to examine. The higher the maximum number of iterations, the more detail and subtlety emerge in the final image, but the longer time it will take to calculate the fractal image.

To render such an image, the region of the complex plane we are considering is subdivided into a certain number of pixels. To color any such pixel, let c be the midpoint of that pixel. We now iterate the critical point 0 under P_c, checking at each step whether the orbit point has modulus larger than 2. When this is the case, we know that c does not belong to the Mandelbrot set, and we color our pixel according to the number of iterations used to find out. Otherwise, we keep iterating up to a fixed number of steps, after which we decide that our parameter is "probably" in the Mandelbrot set, or at least very close to it, and color the pixel black.
In pseudocode, this algorithm would look as follows. The algorithm does not use complex numbers, and manually simulates complex number operations using two real numbers, for those who do not have a complex data type. The program may be simplified if the programming language includes complex data type operations.
For each pixel (Px, Py) on the screen, do:

```
{
x0 = scaled x coordinate of pixel (should be scaled to lie somewhere in the Mandelbrot X scale (-2.5, 1))
y0 = scaled y coordinate of pixel (should be scaled to lie somewhere in the Mandelbrot Y scale (-1, 1))
  x = 0.0
  y = 0.0
  iteration = 0
  max_iteration = 1000
  while ( x*x + y*y < 2*2  AND  iteration < max_iteration )
  {
    xtemp = x*x - y*y + x0
    y = 2*x*y + y0
    x = xtemp
    iteration = iteration + 1
  }
  color = palette[iteration]
  plot(Px, Py, color)
}
```

where, relating the pseudocode to c, z and P_c:

- $z = x + iy$

- $z^2 = x^2 + i2xy - y^2$
- $c = x_0 + iy_0$

and so, as can be seen in the pseudocode in the computation of x and y:

- $x = \mathrm{Re}(z^2 + c) = x^2 - y^2 + x_0$ and $y = \mathrm{Im}(z^2 + c) = 2xy + y_0.$

To get colorful images of the set, the assignment of a color to each value of the number of executed iterations can be made using one of a variety of functions (linear, exponential, etc.).

As is mentioned above the computations involved in calculating the colour of each pixel is directly dependent on the number of iterations which are executed before the escape condition is reached. Hence the faster the processor, the faster is the calculation of color values of the pixels and better the result. Hence by measuring the time it takes for executing the same number of calculations (we fix the escape limit) for each processor we use the results obtained from the Mandelbrot set as a benchmark to compare CUDA based and non CUDA based systems.

3.3 The CODE

a) CUDA BASED

```
#include<iostream>

#include <stdio.h>
#include <gl/glut.h>
#include <math.h>
#include<vector>
#include <cuda_runtime_api.h>
#include<windows.h>
#include<time.h>

class Complex1
{
public:
   float   r;
   float   i;
   __device__ Complex1( float a, float b ) : r(a), i(b)  {}
   __device__ Complex1(const Complex1& x) : r(x.r), i(x.i) {}
   __device__ float magnitude2( void ) {
      return r * r + i * i;
   }
   __device__ Complex1 operator*(const Complex1& a) {
      return Complex1(r*a.r - i*a.i, i*a.r + r*a.i);
   }
   __device__ Complex1 operator+(const Complex1& a) {
      return Complex1(r+a.r, i+a.i);
   }
};
using namespace std;
int StartX = -1;
int StartY = -1;
int EndX = -1;
int EndY = -1;
clock_t t,t1;
```

```
int nx, ny,c=0,r,g=256,b=256,noofani=0,updateRate=200;
double *arrx,*arry,*arrop;
//GLdouble realMax=0.75f,realMin=-2.25f,imagMax=1.25f,imagMin=-1.25f,realInc,imagInc;
GLdouble realMax=1.0f,realMin=-2.0f,imagMax=1.8f,imagMin=-1.2f,realInc,imagInc;
void timer(int)
{noofani++;
if(noofani<5)
{
 // Adjust rotation angles as needed here
 // Then tell glut to redisplay
 glutPostRedisplay();
 // And reset tht timer
 glutTimerFunc(1000.0 / updateRate, timer, 0);
}
else if(noofani==5)
{//cout << "done benchmarking " << (tp.tv_sec+tp.tv_usec/1000000.0) - startSec << " seconds" << endl;
DWORD points=GetTickCount();
t1=clock();
cout << "done benchmarking and points afer "<<noofani <<" iterations are "<< t1-t<< endl;
}

else
        return;
}
class memory
{public:
double minx,miny,maxx,maxy;
memory(double a, double b, double c, double d):minx(a),miny(b),maxx(c),maxy(d)
{}

};
vector<memory> m1;

__global__ void kernal(double *dev_arrx,double *dev_arry, double *dev_arrop)
{

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

                int i = tid / 512;
                int j = tid % 512;

                Complex1 c(dev_arrx[j],dev_arry[i]),op(0,0),temp(0,0);
        int cnt=0;

                while(((op.r)*(op.r))+((op.i)*(op.i))<=4 && cnt<=2000)
                {
                        op=(temp*temp)+c;
                        temp=op;
                        cnt++;
                }
                dev_arrop[tid]=cnt;

}


// Called to draw scene
```

```
void RenderScene(void) {


   // Clear the window with current clearing color
   glClear(GL_COLOR_BUFFER_BIT);

         int arropcnt=0,cnt=0;


   realInc = (realMax - realMin) / (GLdouble)nx;
   imagInc = (imagMax - imagMin) / (GLdouble)ny;
   // Call only once for all remaining points

   for(int i=0;i<ny;i++)
         {
         arry[i]=imagMin+(i*imagInc);
         }
         for(int j=0;j<nx;j++)
         {
                        arrx[j]=realMin + (j*realInc);

         }


         double *dev_a, *dev_b, *dev_c;
         cudaMalloc( (void**)&dev_a, 512*sizeof(double) );
         cudaMalloc( (void**)&dev_b, 512*sizeof(double) );
         cudaMalloc( (void**)&dev_c, 512*512*sizeof(double) );

         cudaMemcpy( dev_a, arrx, 512 * sizeof(double), cudaMemcpyHostToDevice);
         cudaMemcpy( dev_b, arry, 512 * sizeof(double), cudaMemcpyHostToDevice);

         kernal<<<8192,32>>>(dev_a, dev_b, dev_c);

         cudaMemcpy (arrop, dev_c, 512 * 512 * sizeof(double), cudaMemcpyDeviceToHost );



         arropcnt=0;
         glBegin(GL_POINTS);
         for(int i=0;i<ny;i++)
         {
               for( int j=0;j<nx;j++,arropcnt++)
               {

                        if(arrop[arropcnt]<2000 && arrop[arropcnt]>0)
                        {r=arrop[arropcnt];
                        r=r%16;
                        switch (r)
                              {case 1: glColor3f(25, 0, 26);
                                          break;
                              case 2: glColor3f(9, 0, 47);
                                          break;

                              case 3: glColor3f(4, 0, 73);
                                          break;
```

```
                                        case 4: glColor3f(0, 7, 100);
                                                break;
                                        case 5: glColor3f(0, 44, 138);
                                                break;
                                        case 6: glColor3f(0, 82, 177);
                                                break;

                                        case 7: glColor3f(0, 125, 209);
                                                break;
                                        case 8: glColor3f(0, 181, 229);
                                                break;
                                        case 9: glColor3f(0, 236, 248);
                                                break;
                                        case 10: glColor3f(241, 233, 0);
                                                break;

                                        case 11: glColor3f(248, 201, 0);
                                                break;
                                        case 12: glColor3f(255, 170, 0);
                                                break;
                                        case 13: glColor3f(204, 128, 0);
                                                break;
                                        case 14: glColor3f(153, 87, 0);
                                                break;

                                        case 15: glColor3f(106, 52, 0);
                                                break;
                                        case 0: glColor3f(66, 30, 0);
                                                break;
                                        default: glColor3f(66, 30, 0);break;
                                }




                                }
                                else
                                        glColor3f(0,0,0);

                                 glVertex2d( j, i );
                        }
                }


 // Done drawing points
 glEnd();

 glMatrixMode( GL_PROJECTION );
 glLoadIdentity();
 glOrtho( 0, nx, ny, 0, -nx, nx );

 glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
        //----selection
 if( StartX > 0 && StartY > 0 && EndX > 0 && EndY > 0 )
 {
   glLogicOp(GL_XOR);
```

```
      glEnable(GL_COLOR_LOGIC_OP);
      glColor3f(1.0, 1.0, 1.0);
      glLineWidth(1.0);
      glBegin(GL_LINE_LOOP);

      glVertex2i(StartX, StartY);
      glVertex2i(EndX, StartY);
      glVertex2i(EndX, EndY);
      glVertex2i(StartX, EndY);

      glEnd();
      glDisable(GL_COLOR_LOGIC_OP);
   }

   // Flush drawing commands
   glutSwapBuffers();
            realMax=realMax/1.8+0.25;
            imagMax=imagMax/1.8+0.25;
            realMin=realMin/1.8-0.25;
            imagMin=imagMin/1.8-0.25;
}



void mouse( int button, int state, int x, int y )
{
   if( button == GLUT_LEFT && state == GLUT_DOWN )
   {
      StartX = x;
      StartY = y;


   }
   if( button == GLUT_LEFT && state == GLUT_UP )
   {
            if(StartX<EndX)
            {realMin=arrx[StartX];
            realMax=arrx[EndX];
            }
            else
            {realMin=arrx[EndX];
            realMax=arrx[StartX];
            }
            if(StartY<EndY)
            {
            imagMin=arry[StartY];
            imagMax=arry[EndY];
            }
            else
            {imagMin=arry[EndY];
            imagMax=arry[StartY];
            }

            m1.push_back(memory(realMin,imagMin,realMax,imagMax));
      StartX = -1;
      StartY = -1;
      EndX = -1;
```

```
      EndY = -1;
                  glutPostRedisplay();
   }
}

void motion( int x, int y )
{
   EndX = x;
         cout<<StartX<<" "<<StartY<<" ";
         if(StartY-y>0 && StartX-EndX>0)
   EndY = StartY-(StartX-EndX);
         else if(StartY-y>0 && StartX-EndX<0)
   EndY = StartY+(StartX-EndX);
         else if(StartY-y<0 && StartX-EndX>0)
         EndY = StartY+(StartX-EndX);
         else
         EndY = StartY-(StartX-EndX);


         glutPostRedisplay();
         cout<<EndX<<" "<<EndY<<endl;
}



void Key(unsigned char key, int x, int y) {


         if(key=='b'|| key== 'B')
         {
         if(m1.size()>1)
                  {imagMin=m1[m1.size()-2].miny;
                  realMin=m1[m1.size()-2].minx;
                  imagMax=m1[m1.size()-2].maxy;
                  realMax=m1[m1.size()-2].maxx;
         }
                  if(m1.size()!=1)
                  m1.erase(m1.begin()+(m1.size()-1));

         }
         if(key=='q'|| key== 'Q')
                  exit(0);


          glutPostRedisplay();

}


void ChangeSize(int w, int h) {
   nx = 512;
   ny = 512;

         if(arrx!=0)
         {delete[] arrx;}
         if(arry!=0)
                  delete[] arry;
```

```
        if(arrop!=0)
                delete[] arrop;

   arrx=new double [512];
        arry=new double [512];
        arrop=new double [512*512];
        if(m1.size()>0)
                m1.clear();

        m1.push_back(memory(realMin,imagMin,realMax,imagMax));

   // Set Viewport to window dimensions
   glViewport(0, 0, w, h);

   // Reset projection matrix stack
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   // Establish clipping volume (left, right, bottom, top, near, far)

      glOrtho(0,w,0,h,0,w);



   // Reset Model view matrix stack
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

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

        t=clock();
        glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(512, 512);
        glutCreateWindow("CUDA LAB");
   glutReshapeFunc(ChangeSize);
        glutKeyboardFunc(Key);
        glutMouseFunc( mouse );
   glutMotionFunc( motion );
   glutDisplayFunc(RenderScene);
        glutTimerFunc(1000.0 / updateRate, timer, 0);
   glutMainLoop();

   return 0;
}
```

b) NON- CUDA BASED

```
#include<iostream>
#include"Complex.h"
#include <stdio.h>
#include <GL/glut.h>
#include <math.h>
#include<vector>
//#include <time.h>
//#include<Winbase.h>
```

```
#include<windows.h>
#include<time.h>
using namespace std;
int StartX = -1;
int StartY = -1;
int EndX = -1;
int EndY = -1;
int nx, ny,c=0,updateRate=200,noofani=0;
double *arrx,*arry,*arrop;
clock_t t,t1;
//struct timeval tp;
typedef unsigned long DWORD;
//GLdouble realMax=-1.49436,realMin=-
1.4944f,imagMax=0.347133696469,imagMin=0.347130303531,realInc,imagInc;
GLdouble realMax=1.0f,realMin=-2.0f,imagMax=1.8f,imagMin=-1.2f,realInc,imagInc;
void timer(int)
{noofani++;
if(noofani<5)
{
  // Adjust rotation angles as needed here
  // Then tell glut to redisplay
  glutPostRedisplay();
  // And reset tht timer
  glutTimerFunc(1000.0 / updateRate, timer, 0);
}
else if(noofani==5)
        {//cout << "done benchmarking " << (tp.tv_sec+tp.tv_usec/1000000.0) - startSec << " seconds" <<
endl;
DWORD points=GetTickCount();
t1=clock();
cout << "done benchmarking and points afer "<<noofani <<" iterations are "<< t1-t<< endl;
}
else
        return;
}
class memory
{public:
double minx,miny,maxx,maxy;
memory(double a, double b, double c, double d):minx(a),miny(b),maxx(c),maxy(d)
{}

};
vector<memory> m1;



// Called to draw scene

void RenderScene(void) {



   // Clear the window with current clearing color
   glClear(GL_COLOR_BUFFER_BIT);

        int arropcnt=0,cnt=0;
```

```
realInc = (realMax - realMin) / (GLdouble)nx;
imagInc = (imagMax - imagMin) / (GLdouble)ny;
// Call only once for all remaining points

for(int i=0;i<ny;i++)
	{
	arry[i]=imagMin+(i*imagInc);

		for(int j=0;j<nx;j++)
		{
			arrx[j]=realMin + (j*realInc);

		}
	}
	// cuda----------


	for(int i=0;i<ny;i++)
	{
		for( int j=0;j<nx;j++,arropcnt++)
		{
			Complex c(arrx[j],arry[i]),op(0,0),temp(0,0);

			cnt=0;

			while(((op.real)*(op.real))+((op.imag)*(op.imag))<=4 && cnt<2000)
			{
				op=(temp*temp)+c;
				temp=op;
				cnt++;
			}
			arrop[arropcnt]=cnt;


		}
	}

	//cuda ends----------
	arropcnt=0;
	glBegin(GL_POINTS);
	int r=0,g=0,b=0;
	for(int i=0;i<ny;i++)
	{
		for( int j=0;j<nx;j++,arropcnt++)
		{

			if(arrop[arropcnt]<2000 && arrop[arropcnt]>0)
			{r=arrop[arropcnt];
			r=r%16;
			switch (r)
				{case 1: glColor3f(25, 0, 26);
					break;
				case 2: glColor3f(9, 0, 47);
					break;

				case 3: glColor3f(4, 0, 73);
```

```
                                        break;
                                case 4: glColor3f(0, 7, 100);
                                        break;
                                case 5: glColor3f(0, 44, 138);
                                        break;
                                case 6: glColor3f(0, 82, 177);
                                        break;

                                case 7: glColor3f(0, 125, 209);
                                        break;
                                case 8: glColor3f(0, 181, 229);
                                        break;
                                case 9: glColor3f(0, 236, 248);
                                        break;
                                case 10: glColor3f(241, 233, 0);
                                        break;

                                case 11: glColor3f(248, 201, 0);
                                        break;
                                case 12: glColor3f(255, 170, 0);
                                        break;
                                case 13: glColor3f(204, 128, 0);
                                        break;
                                case 14: glColor3f(153, 87, 0);
                                        break;

                                case 15: glColor3f(106, 52, 0);
                                        break;
                                case 0: glColor3f(66, 30, 0);
                                        break;
                                default: glColor3f(66, 30, 0);break;
                        }



                                //glColor3f(r%256,r%128,r%32);
                        }
                        else
                                glColor3f(0,0,0);

                         glVertex2d( j, i );
                }
        }
/* for (x = 0, z.x = realMin; x < nx; x++, z.x += realInc) {
   for (y = 0, z.y = imagMin; y < ny; y++, z.y += imagInc) {


      //cnt=iterate(z,maxIter);


      t2=z;
      cnt = 0;

      while ((t2.x * t2.x + t2.y * t2.y <=4) && (cnt < maxIter)) {
         t.x=t2.x*t2.x-t2.y*t2.y;
         t.y=2*t2.x*t2.y;
         t2 = t;
```

```
          t2.x +=z.x;
          t2.y +=z.y;
          cnt++;
        }*/


          /* if(cnt<=255)
                                        glColor3f(256,256,256);
                                                        else
                                        glColor3f(0,0,0);
                    */









         //   glVertex2d(x , y );
        // }




//}

// Done drawing points
glEnd();

glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho( 0, nx, ny, 0, -nx, nx );

glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
          //----selection
if( StartX > 0 && StartY > 0 && EndX > 0 && EndY > 0 )
{
   glLogicOp(GL_XOR);
   glEnable(GL_COLOR_LOGIC_OP);
   glColor3f(1.0, 1.0, 1.0);
   glLineWidth(1.0);
   glBegin(GL_LINE_LOOP);

   glVertex2i(StartX, StartY);
   glVertex2i(EndX, StartY);
   glVertex2i(EndX, EndY);
   glVertex2i(StartX, EndY);

   glEnd();
   glDisable(GL_COLOR_LOGIC_OP);
}

// Flush drawing commands
glutSwapBuffers();
```

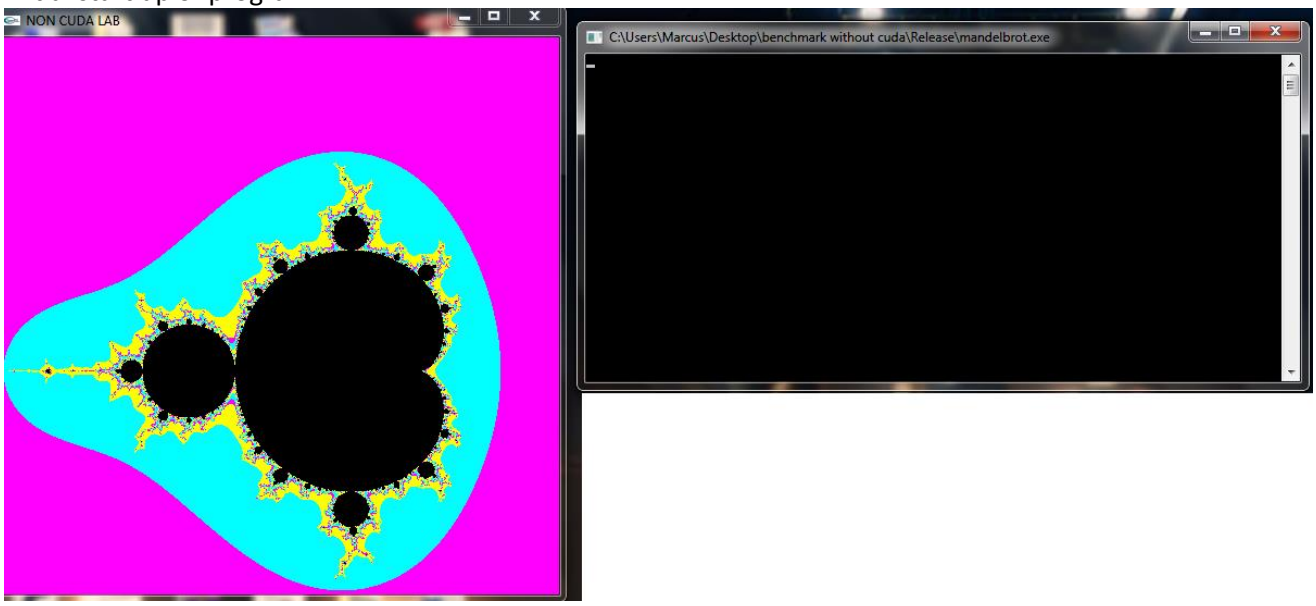
```
            realMax=realMax/1.8+0.25;
            imagMax=imagMax/1.8+0.25;
            realMin=realMin/1.8-0.25;
            imagMin=imagMin/1.8-0.25;
            //cout<<"i"<<endl;
}

// This function does any needed initialization on the rendering
// context.

void mouse( int button, int state, int x, int y )
{
   if( button == GLUT_LEFT && state == GLUT_DOWN )
   {
      StartX = x;
      StartY = y;


   }
   if( button == GLUT_LEFT && state == GLUT_UP )
   {
            if(StartX<EndX)
            {realMin=arrx[StartX];
            realMax=arrx[EndX];
            }
            else
            {realMin=arrx[EndX];
            realMax=arrx[StartX];
            }
            if(StartY<EndY)
            {
            imagMin=arry[StartY];
            imagMax=arry[EndY];
            }
            else
            {imagMin=arry[EndY];
            imagMax=arry[StartY];
            }

            m1.push_back(memory(realMin,imagMin,realMax,imagMax));
      StartX = -1;
      StartY = -1;
      EndX = -1;
      EndY = -1;
                        glutPostRedisplay();
   }
}

void motion( int x, int y )
{
   EndX = x;
            cout<<StartX<<" "<<StartY<<" ";
            if(StartY-y>0 && StartX-EndX>0)
   EndY = StartY-(StartX-EndX);
            else if(StartY-y>0 && StartX-EndX<0)
   EndY = StartY+(StartX-EndX);
            else if(StartY-y<0 && StartX-EndX>0)
```

```
        EndY = StartY+(StartX-EndX);
        else
        EndY = StartY-(StartX-EndX);


        glutPostRedisplay();
        cout<<EndX<<" "<<EndY<<endl;
}



void Key(unsigned char key, int x, int y) {


        if(key=='b'|| key== 'B')
        {cout<<"pakda"<<endl;
        if(m1.size()>1)
                {imagMin=m1[m1.size()-2].miny;
                realMin=m1[m1.size()-2].minx;
                imagMax=m1[m1.size()-2].maxy;
                realMax=m1[m1.size()-2].maxx;
        }
                if(m1.size()!=1)
                m1.erase(m1.begin()+(m1.size()-1));

        }
        if(key=='q'|| key== 'Q')
                exit(0);


         glutPostRedisplay();

}


void ChangeSize(int w, int h) {
   nx = w;
   ny = h;
   nx = w;
   ny = h;
        if(arrx!=0)
        {delete[] arrx;}
        if(arry!=0)
                delete[] arry;
        if(arrop!=0)
                delete[] arrop;

   arrx=new double [w];
        arry=new double [h];
        arrop=new double [w*h];
        if(m1.size()>0)
                m1.clear();

        m1.push_back(memory(realMin,imagMin,realMax,imagMax));

   // Set Viewport to window dimensions
   glViewport(0, 0, w, h);
```

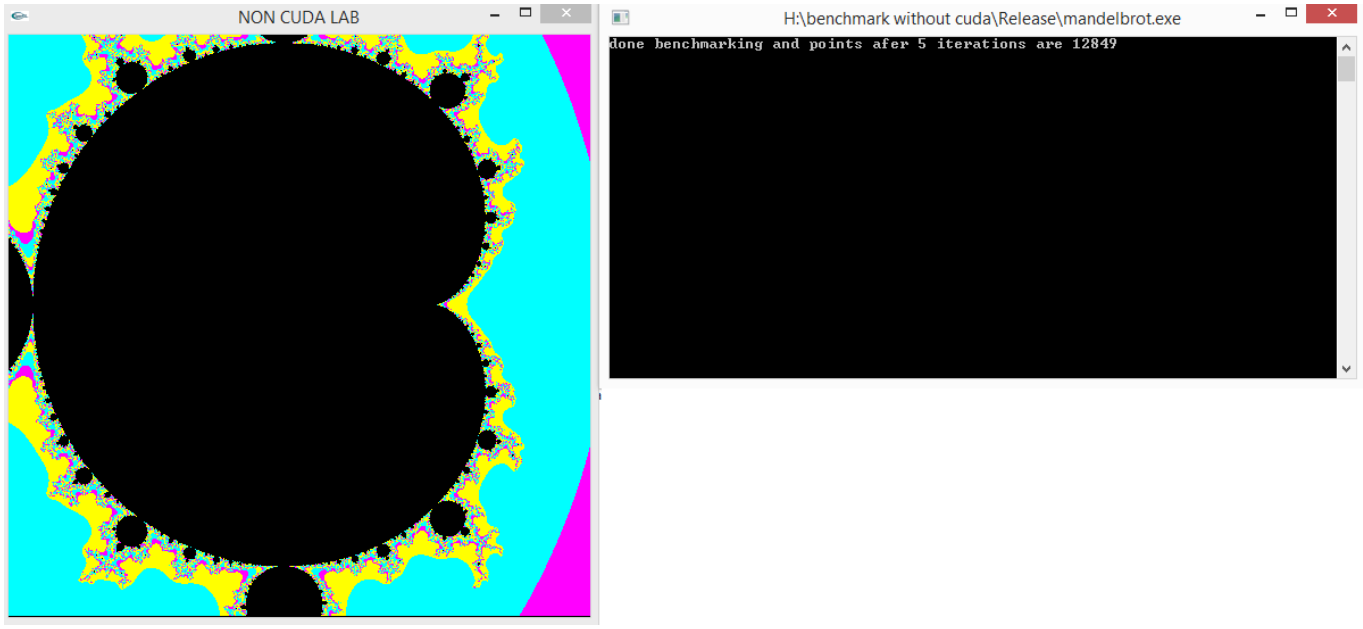
```
   // Reset projection matrix stack
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   // Establish clipping volume (left, right, bottom, top, near, far)

      glOrtho(0,w,0,h,0,w);



   // Reset Model view matrix stack
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char* argv[]) {
            //GetSystemTime(&tp, 0);
            //double startSec = tp.tv_sec + tp.tv_usec/1000000.0;
            t=clock();
            glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(512, 512);
            glutCreateWindow("NON CUDA LAB");
   glutReshapeFunc(ChangeSize);
            glutKeyboardFunc(Key);
            glutMouseFunc( mouse );
   glutMotionFunc( motion );
   glutDisplayFunc(RenderScene);
            glutTimerFunc(1000.0 / updateRate, timer, 0);
   glutMainLoop();



   return 0;
}
```

b) RESULTS

Initial start up of program

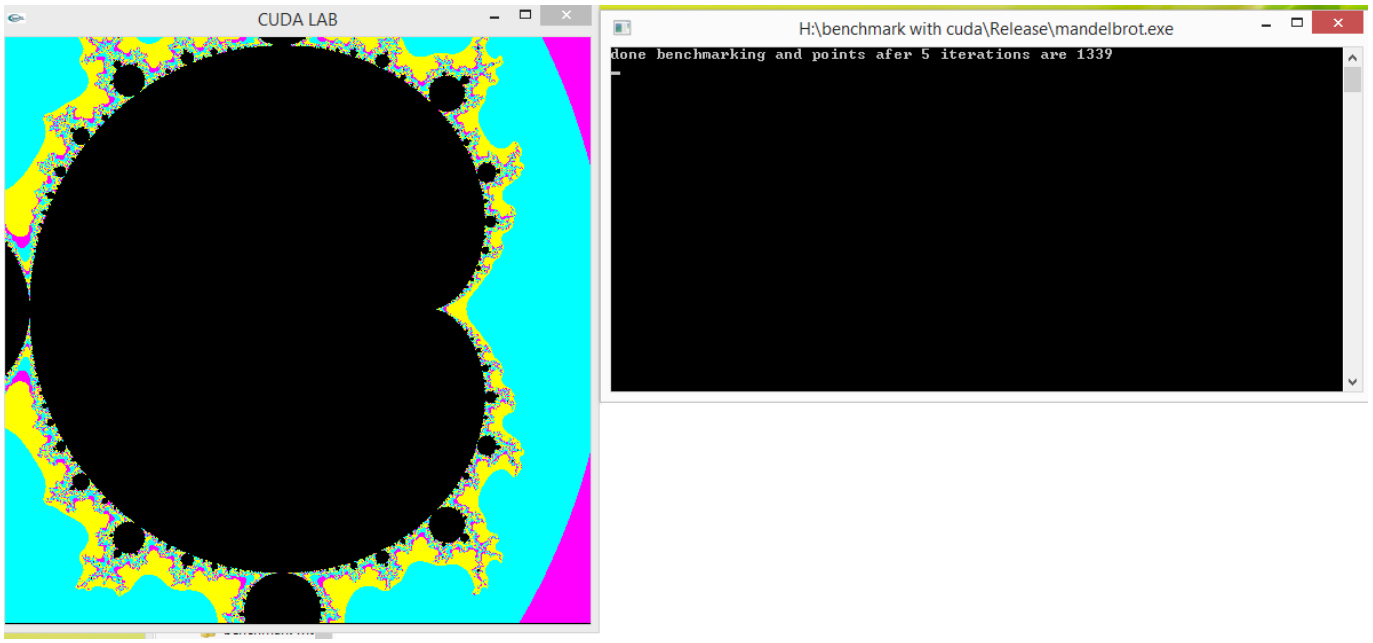


Program when run only on the CPU



As can be seen the code executes in 12849 ms. There are a total of 5 iterations. Each iteration results in one level of zoom and each zoom increases the calculations greatly.

Program when run on GPGPU (GT 750M)



It is seen that the program executes 5 levels of zoom in 1339 ms which is in accordance with the results that the program executes faster on the GPGPU. Hence it is proved that the Mandelbrot set can be used as an effective benchmark.