The Mandelbrot Set

CECS 625-75-4158 Parallel Programming Fall 2015

Department of Computer Science & Engineering

University of Louisville

By

Viralkumar Intwala (5045025) & Rohini Basavaraj Nataraj (5013281)

Table of Contents

Introduction

Problem

General Programming on Graphics Processing Units (GPGPU) provides massive parallelism through the use of graphics processing units (colloquially, video cards) for non-graphics related computation. This has traditionally been done by treating graphics primitives (such as vertices) as data, and then applying shaders on the graphics primitives to simulate general programming. Recently, however, frameworks have emerged to aid in GPGPU. One of the first to emerge is Nvidia CUDA, which is a GPGPU framework developed by Nvidia for exclusive use with Nvidia GPUs.

The Mandelbrot Set computation of the Mandelbrot Set involves repetitive application of the map

$$z = z^2 + c$$

on the points of the complex plane.

Objectives

- Successful implementation of algorithms for the computation of the Mandelbrot Set in OpenGL, C++.
- Successful implementation of algorithms for the zoom in/out computation on a specific viewport of the Mandelbrot Set in OpenGL, C++.
- Comparison of performance between OpenGL running on a GPU, OpenGL running on a CPU.

- Additionally, Animation using colors through each iteration of computing the Mandelbrot Set.

Approach

Comparing the implementations of the same algorithm for the computation of Mandelbrot Set in OpenGL on CPU and GPU. Further adding the functionality of Zoom in/out on a specific viewport of the set and finally adding animation using each iteration of the set.

Background

Key Concepts

GPGPU and GPU Architecture

GPGPU, or General Programming on Graphics Processing Units, is the usage of graphics processing units (video cards) for non-graphics-related computation. This is advantageous because graphics processing units allow for massive parallelism due to their architecture: since graphics processing units have traditionally been used for graphics-related computations, such as applying a Gaussian mask.

Graphics-related computations usually involve applying the same computation over a large set of graphical primitives, such as the convolution of the Gaussian kernel on an image. Thus, the architecture of graphics processing units are oriented towards performing tasks that involve data parallelism. Figure 1 depicts a sample GPU architecture.

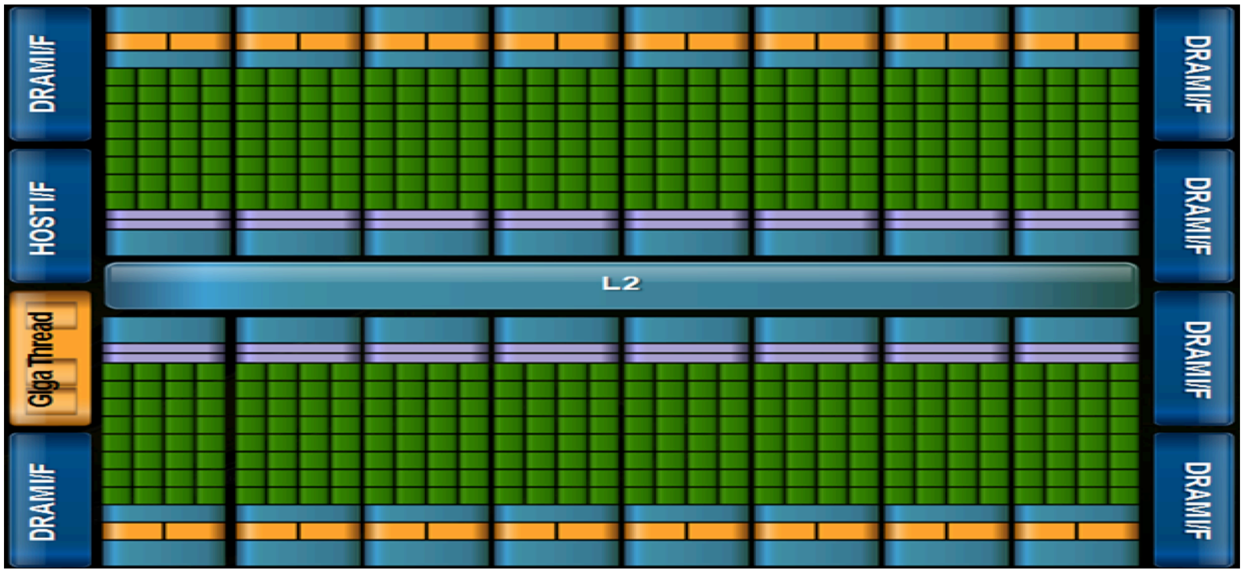

Figure 1: Nvidia Fermi GPU architecture [17]

Whereas an average CPU contains 2 to 8 cores, a GPU, such as the chip below from Nvidia [17], which is especially oriented towards GPGPU computation, contains 512 cores, each capable of SIMD (single instruction, multiple data) – in other words, a GPU has up to hundreds of cores (as compared to CPUs which have 8-16 at the most), and each of those cores is capable of executing dozens of instruction streams as the same time. This, of course, comes at a cost: GPUs have few control structures, and no current GPUs contain advanced control structures such as branch prediction; even branching is a computationally expensive task.

The general flow of computation on a GPU starts with the loading of data onto the GPU. This is often one of the most expensive operations, since the data has to travel through the bus. In normal GPU processing, this step has the transfer of graphics primitives, such as an image to be rendered; in GPGPU, this is the transfer of the data to be operated on.

After the data is loaded, the data is operated on using a set of operations – a shader, in traditional GPU terms. Whereas before, shaders were constrained to a fixed set, nowadays shaders are programmable – which is what enables GPGPU. In GPGPU, a shader is called a kernel instead, to reflect the more general scope.

Afterwards, data is then transferred back to main memory, where the CPU can operate on it once again.

Mandelbrot Set

The Mandelbrot Set is the set of points in the complex plane that remain bounded after infinitely iterations of the map

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

where $z_0 = c$ is the original point. Note that once a point leaves the circle of radius 2 centered on the origin of the complex plane (i.e., $|z_n| > 2$), the point will eventually escape to infinity. Thus, the Mandelbrot Set is often visualized by coloring the complex plane according to the number of iterations it takes for a point to escape the circle of radius 2. This visualization gives insight to the structure of the Mandelbrot Set; namely, its infinite self-similarity. The border of the Mandelbrot Set contains infinitely many replicas of the original Mandelbrot Set, and each replicas itself contains infinitely replicas.

The detail most essential to this project, however, is that the computation of the Mandelbrot Set is highly amenable to parallelization: the same map is performed on each point in the complex plane: a perfect example of data parallelism, and an excellent candidate for GPGPU.

Approach

High Level Design

The Mandelbrot Set will be implemented with the Zoom functionality in both CPU and GPU using OPENGL. The implementations will be straightforward adaptations of the algorithms, and are detailed in the following section.

Note that this implementation will require freeGLUT (1.8.1), OpenGL, CUDA SDK (7.5) to properly function.

Implementation

The implementation of the Mandelbrot Set will be as follows: the 2-dimensional region of the Mandelbrot Set to be visualized will be divided into a 500 x 500 grid (can be changed), each corresponding to a pixel in the visualization. Each pixel will then be iterated upon for up to "maxIter" (variable - can be changed) iterations, or until it escapes the circle of radius 2. In the CPU implementation, this will be done using a simple for-loop, while this will be performed in a kernel for the GPU implementation. The region will then be colored according to the number of iterations performed using Open GL.

As the Mandelbrot Set is a set represented in a complex plane, a Mapping system is required to actually draw the complex numbers on the coordinate system. Implementing a structure representing the complex numbers is a good choice with the basic math operations such as "add" , "multiply" and "magnitude". To properly display the set on the window setting up a GLUT window is required which will produce a window with the desired grid dimensions and key listeners. On the GLUT window the change in the size of the window needs to be addressed as the user might change the size of the window displaying the set, using OpenGL clipping functions such as “glortho” that can be achieved.

If you take any part of the border of the set, the length of this part will also be infinite. This means that the border of the set has "infinite details", that is, you'll never find a place where the border is smooth, with a finite length (when calculating the set with a program you can get images where it seems that the border is smooth, but that's always caused because a too low iteration amount and/or image resolution is used).

Thanks to this property you can zoom into any part of the border of the set and always get something to see (as long as you have enough resolution, you make enough iterations and your numbers don't overflow/underflow).

MAPPING:

The mapping of the complex plane on the coordinate system depends on four factors RealMax, RealMin, ImagMax, ImagMin which would be the ones affecting the viewport of the Complex set on the window. Initially these values needs to be such that the Mandelbrot set appears in the middle of the window within radius of 2.

```
realMax=0.75f,realMin=-2.25f,imagMax=1.25f,imagMin=-1.25f;
```

VIEWPORT:

To enable the implementation to zoom in and out first a viewport is required as to where does the zoom is performed. Simple Method is just to change the four mapping factors to change the viewport. On a single stroke depending on the Axis of the coordinate system, two mapping factors needs to be changed. A difference factor (diff) decides how much the mapping factors should change on a single stroke.

To move up (Y-Axis) :
```
imagMax += diff;
imagMin += diff;
```

To move down (Y-Axis) :
```
imagMax -= diff;
imagMin -= diff;
```

To move right (X-Axis) :
```
realMax += diff;
realMin += diff;
```

To move left (X-Axis) :
```
realMax -= diff;
realMin -= diff;
```

ZOOM:

After the viewport is set Zooming needs to be addressed. Same thing can be applied for zooming but on a single stroke all the four mapping factors needs to be changed. Two factors a difference factor as in changing viewport and a multiplier decides how much the mapping factors should change in a single stroke.

To Zoom In:
```
realMax -= diff;
realMin += diff;
imagMax -= diff*mul;
imagMin += diff*mul;
```

To Zoom Out:
```
realMax += diff;
realMin -= diff;
imagMax += diff*mul;
imagMin -= diff*mul;
```

COLOR SCHEME:

In this implementation there are six different color schemes which can be cycled through.

Depending on the no of iterations a pixel value has, a math function is applied on it to produce Red, Green and Blue component of the respective pixel. If the pixel value is equal to the maxIter value i.e it is outside the set then it has black color.

Color:

```
if(cnt>=0&&cnt<=31)   {b=cnt*4; g=cnt*8; r=0;  }
else if(cnt>=32&&cnt<=63)  {b=200; g=500-cnt*8; r=0;  }
else if(cnt>=64&&cnt<=95)  {b=200; g=0; r=(cnt-64)*4;}
else if(cnt>=96&&cnt<=127) {r=200; g=0; b=1000-cnt*8;}
else if(cnt>=128&&cnt<=159){r=200; g=(cnt-128)*8; b=0;}
else if(cnt>=160&&cnt<=191){g=200; r=1500-cnt*8; b=0;}
else if(cnt>=192&&cnt<=223){g=200; r=0; b=(cnt-192)*8;}
else if(cnt>=224&&cnt<=255){g=230; r=(cnt-224)*8; b=256;}
```

Scheme:

```
//to change color by prssing key 'c'

tr=r;tb=b;tg=g;
switch(c)
{
       case 0: break;
       case 1: r=tb;b=tr;break;
       case 2: r=tg;g=tr;break;
       case 3: b=tg;g=tb;break;
       case 4: r=tg; g=tb; b=tr; break;
       case 5: r=tb; g=tr; b=tg; break;

}
```

ANIMATION:

Animation in glut is achieved by using the Timer function of the glut library which handles the redrawing of pixels every specified time interval. To animate through each iteration, for every redrawing that the glut does, a variable "iter" changes and a new set of pixels are calculated.

TIMER:

This time.h header file contains definitions of functions to get and manipulate date and time information. This is used to record the processor time consumed by the program. This is used to compare the speed up from CPU to GPU.

KEYS FOR EXECUTION:

- C or c = Run on CPU
- G or g = Run on GPU
- A or a = Shows animation
- W or w = zoom in
- E or e = zoom out
- Right arrow = moves image to right
- Left arrow = moves image to left
- Up arrow = moves image to up
- Down arrow = moves image to down

Results and Analysis

Results

Grid Size	CPU	GPU	Speed Up
500x500	0.432122	0.179151	2.412054635
500x500	0.433238	0.189928	2.281064403
500x500	0.430039	0.156506	2.74774769
700*500	0.596712	0.145374	4.10466796
700*500	0.605249	0.147849	4.093696948

700*500	0.592293	0.153292	3.863821987
1024*768	1.360394	0.196402	6.926579159
1024*768	1.347179	0.267897	5.028719993
1024*768	1.359992	0.210954	6.446865193

Figure 2: Timing record for CPU v/s GPU.

Clearly the GPU implementation of the Mandelbrot algorithm has much better performance than a CPU implementation in the tested resolution.

Screenshots

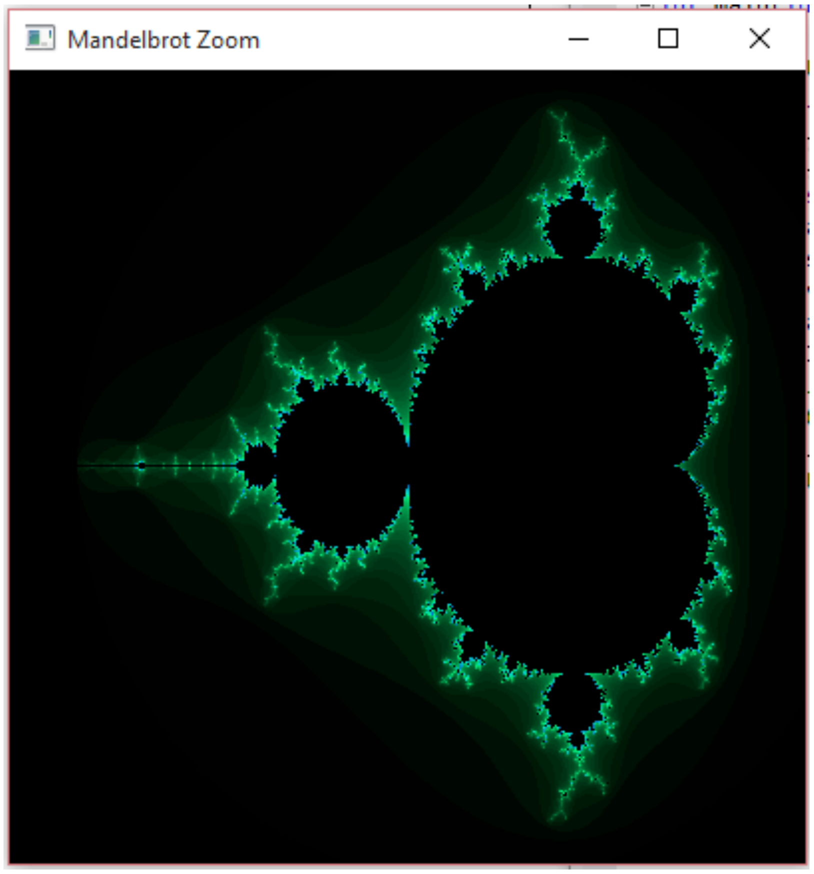

Figure 3: Mandelbrot CPU.

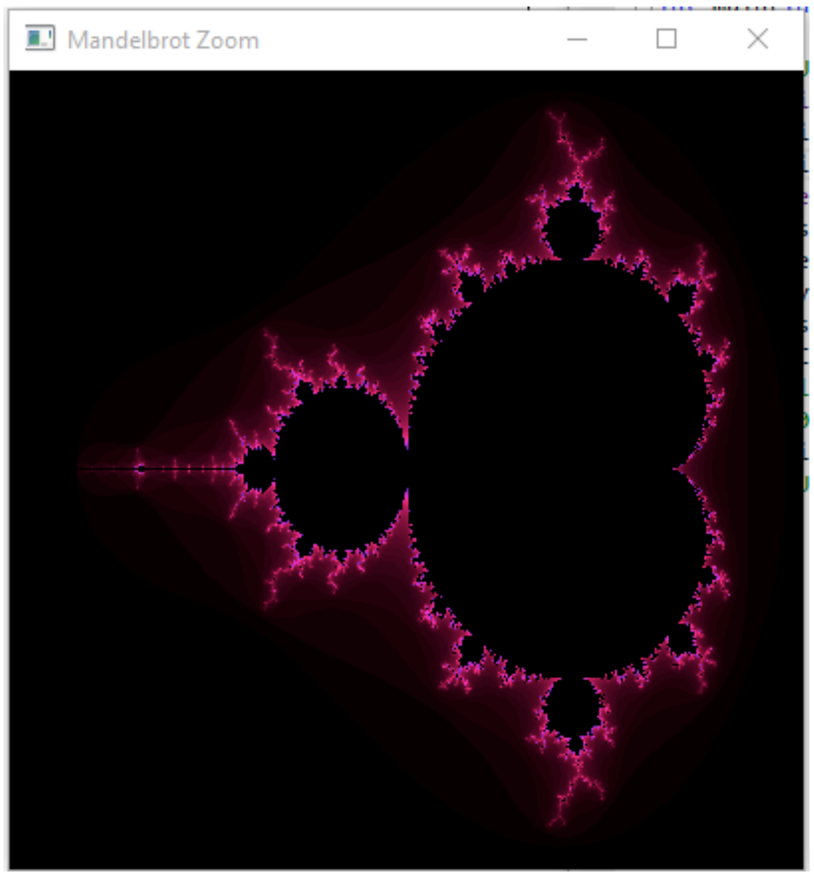

Figure 4: Mandelbrot GPU.

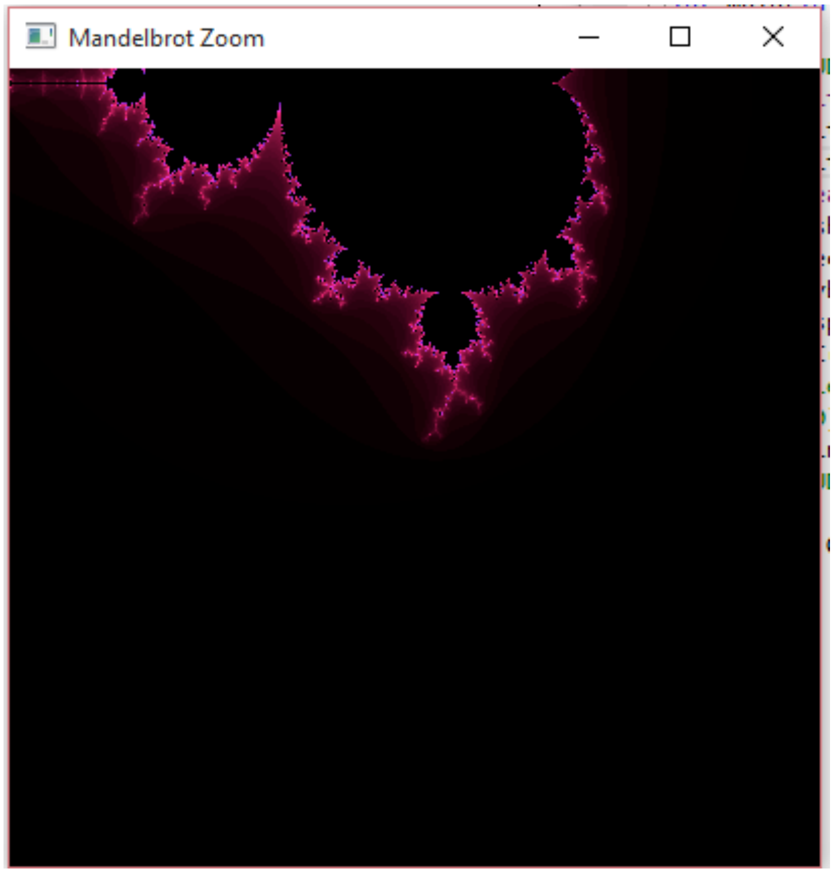

Figure 5: Mandelbrot GPU Viewport.

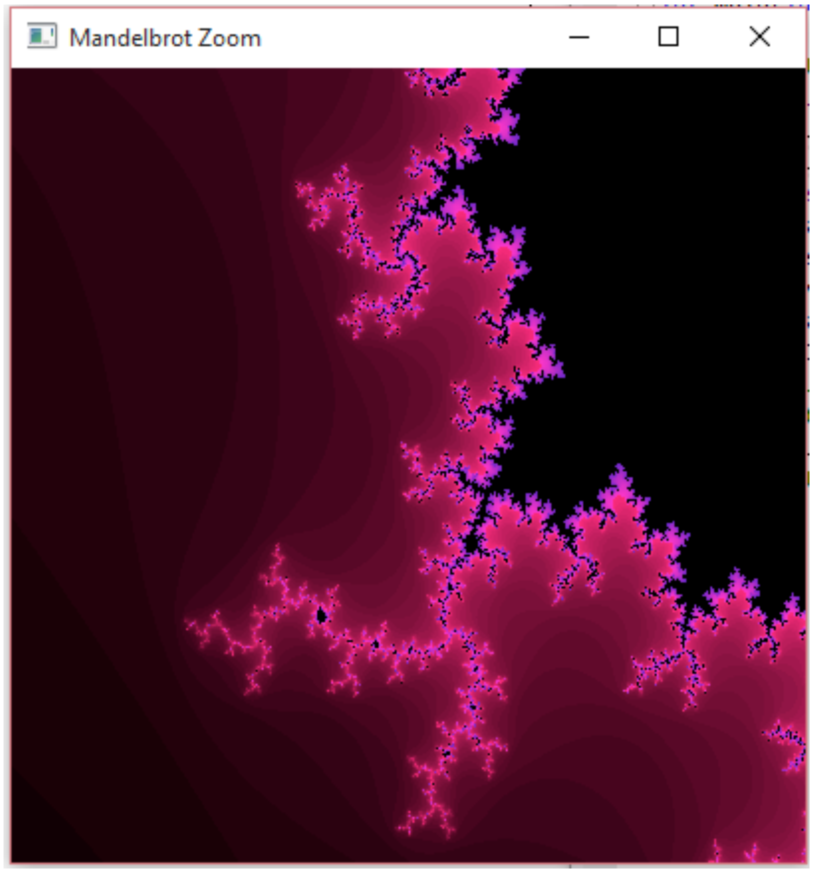

Figure 6: Mandelbrot GPU Zoom.

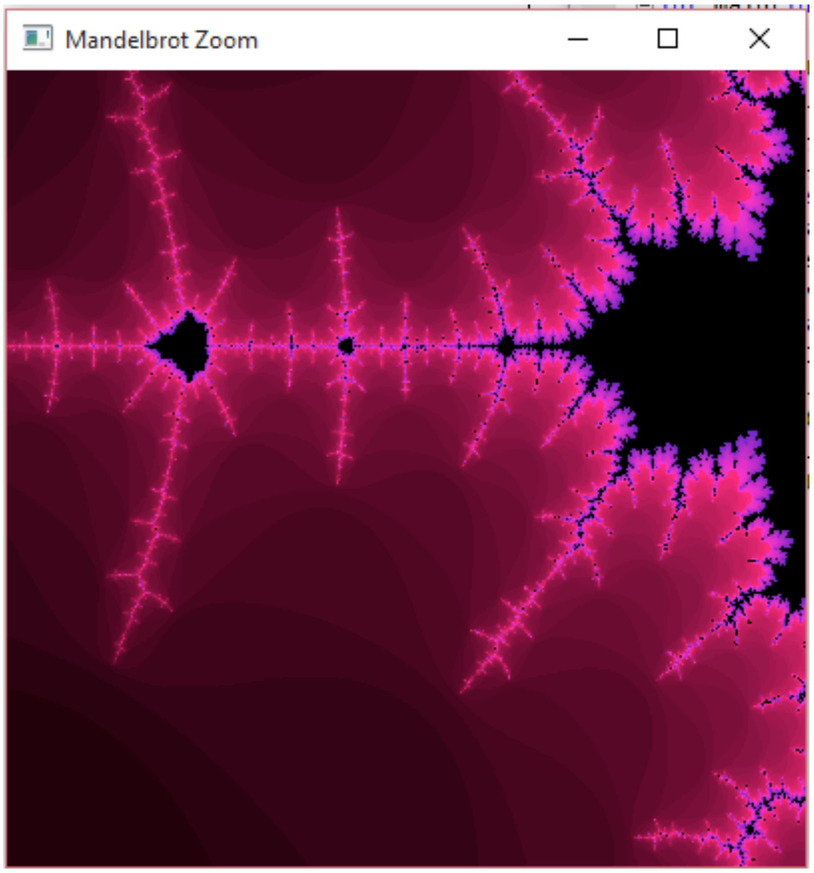

Figure 7: Mandelbrot GPU Zoom.

Analysis

For the algorithm tested, the GPU implementation outperformed the plain (single-core) C++ implementation by an order of magnitude depending on the grid size. These shows the potential for large performance improvements with little effort by simply porting code from CPU to GPU.

Conclusions

GPGPU (General Programming on Graphics Processing Units) vastly improves over traditional methods of computation by utilizing the massive parallelism available through graphics programming units. This project aimed to utilize GPGPU through the use of OpenGL in particular, the Mandelbrot Set. The performance provided by the GPU through OpenGL is compared to CPU performance through OpenGL.

In this project, we saw how to use CUDA programming model, and we presented Mandelbrot example that shows the power of CUDA for computational intensive tasks (computing Mandelbrot fractal in real time).

The experiment results of comparing the GPU and CPU implementation of the Mandelbrot algorithm are very straightforward, showing great performance increase when utilizing the GPU for the primary calculations. This result gives us validation of the benefits of GPU for solving other computing intensive tasks. This project also addresses Zoom in/out functionality on a specific viewport of the set and animation of Mandelbrot set. Which again results in faster performance with GPU.

Appendix I: Code

Mandelbrot, OpenGL

main.cpp

```
// press arrow keys to move the viewport
//f1,f2 to change gradient of zoom in and out and to move, printed(printf) as diff
//press 'w' to zoom in and press 'e' to zoom out
//press 'c' to change color
//To zoom out u should increase gradient(diff) by pressing f2

#include <stdio.h>
#include <GL/freeglut.h>
#include <math.h>
#include "timers.h"

extern "C" int cuComputeMandelbrotSet (int *ptr, int width, int height, GLdouble Rmin, GLdouble
Rmax, GLdouble Imin, GLdouble Imax, int nIterations);
extern "C" bool resetCUDADevice();

int nx, ny,c=0;
GLdouble diff = 0.1;
bool isCPU = false, animate = false;
```

```
//GLdouble realMax=-1.49436,realMin=-
1.4944f,imagMax=0.347133696469,imagMin=0.347130303531,realInc,imagInc;
GLdouble realMax=0.75f,realMin=-2.25f,imagMax=1.25f,imagMin=-1.25f;
//GLdouble realMax = -0.774f, realMin = -0.782f, imagMax = 0.139f, imagMin = 0.133f, realInc,
imagInc;
GLdouble r, g, b,tr,tg,tb;

typedef struct {
    GLdouble x, y;
} complex;

struct cuComplex
{
    GLdouble   r;
    GLdouble   i;

    cuComplex( GLdouble a, GLdouble b ) : r(a), i(b)  {}

    GLdouble magnitude2( void ) { return r * r + i * i; }

    cuComplex operator*(const cuComplex& a)
        {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }

    cuComplex operator+(const cuComplex& a)
        {
        return cuComplex(r+a.r, i+a.i);
    }
};


int MandelbrotColor( int col, int row, int width, int height, GLdouble Rmin, GLdouble Rmax,
GLdouble Imin, GLdouble Imax, int nIterations)
{
        // Normalize (col, row) to {(R,I) | Rmin < R < Rmax, Imin < I < Imax }
        GLdouble R = ((Rmax - Rmin)/(float)width) * (float)col + Rmin;
        GLdouble I = ((Imax - Imin)/(float)height) * (float)row + Imin;
    //float I = Imax - ((Imax - Imin)/(float)height) * (float)y;          // Bottom up

    cuComplex c(R, I);
    cuComplex a(R, I) ;
    int i = 0;
    for (i = 0; i < nIterations; i++)
    {
        a = a * a + c;
        if (a.magnitude2() > 4.0)
            return i;
    }

    return i;
}

void ComputeMandelbrotSet(int width, int height, GLdouble Rmin, GLdouble Rmax, GLdouble Imax,
GLdouble Imin, int maxIter)
{
        int cnt = 0;
        for (int row = 0; row < height; row++)
        {
        for (int col = 0; col < width; col++)
                {
              cnt = MandelbrotColor (col, row, width, height, Rmin, Rmax, Imin, Imax, maxIter);

            if (cnt == maxIter)
                {}
            else {
                       if(cnt>=0&&cnt<=31)   {b=cnt*4; g=cnt*8; r=0;  }
                  else if(cnt>=32&&cnt<=63)  {b=200; g=500-cnt*8; r=0;  }
                  else if(cnt>=64&&cnt<=95)  {b=200; g=0; r=(cnt-64)*4;}
```

```
                    else if(cnt>=96&&cnt<=127) {r=200; g=0; b=1000-cnt*8;}
                    else if(cnt>=128&&cnt<=159){r=200; g=(cnt-128)*8; b=0;}
                    else if(cnt>=160&&cnt<=191){g=200; r=1500-cnt*8; b=0;}
                    else if(cnt>=192&&cnt<=223){g=200; r=0; b=(cnt-192)*8;}
            else if(cnt>=224&&cnt<=255){g=230; r=(cnt-224)*8; b=256;}


                   //to change color by prssing key 'c'

                  tr=r;tb=b;tg=g;
                  switch(c)
                                  {
                                  case 0: break;
                                  case 1: r=tb;b=tr;break;
                                  case 2: r=tg;g=tr;break;
                                  case 3: b=tg;g=tb;break;
                                  case 4: r=tg; g=tb; b=tr; break;
                                  case 5: r=tb; g=tr; b=tg; break;

                                  }



                     glColor3f(r/256,g/256,b/256);

                  glVertex3d(col - nx / 2, row - ny / 2, 0.0f);
              }
          }
      }
  }

// Called to draw scene
int iter = 0;
void RenderScene(void) {
printf("Rendreing Scene == cpu? %d == %d == %d \n",isCPU,nx,ny);

    int maxIter;
    r=g=b=256.0;

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    maxIter = 50;
        if ( !animate ) iter = maxIter;

    // Save matrix state and do the rotation
    glPushMatrix();

 //   realInc = (realMax - realMin) / (GLdouble)nx;
 //   imagInc = (imagMax - imagMin) / (GLdouble)ny;

    // Call only once for all remaining points
    glBegin(GL_POINTS);

        int *h_src = NULL;

        if (h_src != NULL) free (h_src);
                h_src = (int *) calloc (nx * ny, sizeof(int));

        if (h_src == NULL)
                h_src = (int *) calloc (nx * ny, sizeof(int));


        if(isCPU){

                printf("CPU timing .....\n");
                StartCounter();
                ComputeMandelbrotSet(nx, ny, realMin, realMax, imagMin, imagMax, iter);
                printf ("\tCPU time = %lf seconds\n\n", GetCounter());
        }else{

                printf("GPU timing .....\n");
                StartCounter();
                cuComputeMandelbrotSet(h_src, nx, ny, realMin, realMax, imagMin, imagMax, iter);

        int cnt = 0;
        for (int row = 0; row < ny; row++)
        {
        for (int col = 0; col < nx; col++)
                {

                        int index = (col + (ny-row-1) * nx);
                        cnt = h_src[index];
                        if (cnt == maxIter)
                {}
            else {
                       if(cnt>=0&&cnt<=31)   {b=cnt*4; g=cnt*8; r=0;  }
                  else if(cnt>=32&&cnt<=63)  {b=200; g=500-cnt*8; r=0;  }
                  else if(cnt>=64&&cnt<=95)  {b=200; g=0; r=(cnt-64)*4;}
                  else if(cnt>=96&&cnt<=127) {r=200; g=0; b=1000-cnt*8;}
                  else if(cnt>=128&&cnt<=159){r=200; g=(cnt-128)*8; b=0;}
                  else if(cnt>=160&&cnt<=191){g=200; r=1500-cnt*8; b=0;}
                  else if(cnt>=192&&cnt<=223){g=200; r=0; b=(cnt-192)*8;}
          else if(cnt>=224&&cnt<=255){g=230; r=(cnt-224)*8; b=256;}


                 //to change color by prssing key 'c'

                                tr=r;tb=b;tg=g;
                                switch(c)
                                 {
                                 case 0: break;
                                 case 1: r=tb;b=tr;break;
                                 case 2: r=tg;g=tr;break;
                                 case 3: b=tg;g=tb;break;
                                 case 4: r=tg; g=tb; b=tr; break;
                                 case 5: r=tb; g=tr; b=tg; break;

                                 }



                  glColor3f(r/256,g/256,b/256);

                glVertex3d(col - nx / 2, row - ny / 2, 0.0f);
                        }
                }
                }
         printf ("\tGPU time = %lf seconds\n\n", GetCounter());
         free (h_src);
        }


    printf("realMax=%.12lf\nrealMin=%.12lf\nimagMax=%.12lf\nimagMin=%.12lf\ndiff=%.12lf\n",
realMax, realMin, imagMax, imagMin, diff);

        if( iter < maxIter ) iter++;
        else iter = 0;

    // Done drawing points
    glEnd();

    // Restore transformations
    glPopMatrix();

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

```
}

void Timer(int iunused) {
        glutPostRedisplay();
        glutTimerFunc(30, Timer, 0);
}

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

void SetupRC() {
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear (GL_COLOR_BUFFER_BIT);

    // Set drawing color to green
    glColor3f(0.0f, 1.0f, 0.0f);
}

void SpecialKeys(int key, int x, int y) {
    if (key == GLUT_KEY_UP){
        imagMax += diff;
        imagMin += diff;
    }
    if (key == GLUT_KEY_DOWN){
        imagMax -= diff;
        imagMin -= diff;
    }


    if (key == GLUT_KEY_RIGHT){
        realMax += diff;
        realMin += diff;
    }
    if (key == GLUT_KEY_LEFT){
        realMax -= diff;
        realMin -= diff;
    }
    if (key == GLUT_KEY_F1)
        diff /= 10.0f;

    if (key == GLUT_KEY_F2)
        diff *= 10.0f;

    // Refresh the Window
    glutPostRedisplay();
}

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

        GLdouble mul=0.833333333333333333333333;//this multiplier helps to zoom in x and y
properly

        printf("at pronormal key");

         if (key == 'a'||key == 'A'){
                animate = !animate;
         }

         if (key == 'g'||key == 'G'){
                 isCPU = !isCPU;
        }
         if(key == 27 ) {
                exit(0);
         }

    if (key == 'w'||key == 'W'){
        if(realMax-realMin>3*diff){
            realMax -= diff;
```

```
                realMin += diff;
                imagMax -= diff*mul;
                imagMin += diff*mul;
            }
            else
                diff=diff/10;
    }


    if (key == 'e' || key == 'E'){
        realMax += diff;
        realMin -= diff;
        imagMax += diff*mul;
        imagMin -= diff*mul;
    }

    if (key == 'c' || key == 'C'){
                c++;
                if(c==6) c=0;

    }
    // Refresh the Window
    glutPostRedisplay();
}

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

    // Prevent a divide by zero
    if (h == 0)
        h = 1;

    // 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)
    if (w <= h){
        nRange=nx/2;
        glOrtho(-nRange, nRange, -nRange * h / w, nRange * h / w, -nRange, nRange);
    }
    else{
        nRange=ny/2;
        glOrtho(-nRange * w / h, nRange * w / h, -nRange, nRange, -nRange, nRange);
    }

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

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

//      resetCUDADevice();
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
        glutInitWindowSize(500, 500);
    glutCreateWindow("Mandelbrot Zoom");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutKeyboardFunc(proNormalKeys);
    glutDisplayFunc(RenderScene);
    SetupRC();
//      glutIdleFunc(RenderScene);
```

```
        Timer(0);
    glutMainLoop();
//      resetCUDADevice();

    return 0;
}
```

Kernel.cu

```
//
//      kernel.cu - Kernels for fractal (Mandelbrot) set generation
//

#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <GL/freeglut.h>
#include <stdio.h>
#include <stdlib.h> // including: #define EXIT_SUCCESS    0  #define EXIT_FAILURE    1

struct cuComplex
{
    GLdouble   r;
    GLdouble   i;
    __device__ cuComplex( GLdouble a, GLdouble b ) : r(a) , i( b)  { }
    __device__ GLdouble magnitude2( void )
    {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a)
    {
        return cuComplex( r*a.r - i*a.i, i*a.r + r*a.i) ;
    }
    __device__ cuComplex operator+(const cuComplex& a)
    {
        return cuComplex( r+a.r, i+a.i) ;
    }
};

__device__ float toColor (int i)
{
        float intensity [10] = {0, 0.1f, 0.2f, 0.3f, 0.4f, 0.5f, 0.6f, 0.7f, 0.8f, 0.9f};

        return intensity[i%10];
}

//================================================================================================
=
//
//      Mandelbrot set GPU device functions
//
//================================================================================================
=

__device__ int mandelbrot(int row, int col, int width, int height, GLdouble Rmin, GLdouble Rmax,
GLdouble Imin, GLdouble Imax, int nIterations)
{
        // Normalize (row, col) to {(R,I) | Rmin < R < Rmax, Imin < I < Imax }
        GLdouble R = ((Rmax - Rmin)/(float)width) * (float)col + Rmin;
    //float I = ((Imax - Imin)/(float)height) * (float)row + Imin;
        GLdouble I = Imax - ((Imax - Imin)/(float)height) * (float)row;
        //if (I < 0.00001f && I > -0.00001f) I = 0.0f;

    cuComplex c(R, I);
    cuComplex a(R, I) ;
    int i = 0;
    for (i = 0; i < nIterations; i++)
    {
        a = a * a + c;
        if (a. magnitude2() > 4.0)
            return i;
    }

    return i;
}

////////////////////////////////////////////////////////////////////////////////
//
//      Kernel : kernel1
//
//              Computes colors of width by height pixels representing Julia set in
//              {(u,v) | -scale < u < scale, -scale < v < scale }
//
//      Thread grid requirments:
//
//              1) 2D grid of 2D thread blocks covering width by height pixels
//              2) one pixel per thread computing
//
////////////////////////////////////////////////////////////////////////////////

__global__ void Mandelbrot_kernel(int *ptr,int width, int height, GLdouble Rmin, GLdouble Rmax,
GLdouble Imin, GLdouble Imax, int nIterations)
{
    // map from threadIdx/BlockIdx to pixel position
    int col = threadIdx.x + blockIdx.x * blockDim.x;     // column index to the width X height
pixels
    int row = threadIdx.y + blockIdx.y * blockDim.y;     // row index to the width X height
pixels

    // Assuming the origin of the width X height pixels is at upper-left corner
    if (row < height && col < width)
    {
                // Calculate Mandelbrot value at (x,hy) position
                int index = (col + (height-row-1) * width);
                ptr[index] = mandelbrot(row, col, width, height, Rmin, Rmax, Imin, Imax,
nIterations);
        }
}


 #define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"GPU assert: %s %s %d\n", cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}

#define block_size (16)

//==================================================================================================
=======
//
//      Compute Mandelbrot set using CUDA
//
//==================================================================================================
=======

extern "C" int cuComputeMandelbrotSet (int *ptr,int width, int height, GLdouble Rmin, GLdouble
Rmax, GLdouble Imin, GLdouble Imax, int nIterations)
{

        printf("@cuComputeMandelbrotSet %d == %d\n",width, height);
        int *d_ptr = 0;

        cudaError_t cudaStatus;
        cudaDeviceReset();

        // Make sure CUDA device 0 is available
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess)
        {
        fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU installed?");
        return 1;
    }

    // Allocate device memory
    if (cudaMalloc((void **)&d_ptr, width * height * sizeof(int)) != cudaSuccess)
    {
                printf("cuda mem failed");

        fprintf(stderr, "!!!! device memory allocation error (allocate A)\n");
                return EXIT_FAILURE;
    }

        cudaDeviceSynchronize();
        // Copy host memory to device
        cudaStatus = cudaMemcpy (d_ptr, ptr, width*height*sizeof(int), cudaMemcpyHostToDevice);
         if (cudaStatus != cudaSuccess)
    {
        printf("cudaMemcpy (d_ptr, ptr) returned error code\n", cudaStatus);
        exit(EXIT_FAILURE);
    }

        // Setup execution parameters and call kernel
    dim3 block(block_size, block_size);
    dim3 grid ((width+block_size-1)/block_size, (height+ block_size-1)/block_size);
        Mandelbrot_kernel<<< grid, block >>>(d_ptr, width, height, Rmin, Rmax, Imin, Imax,
nIterations);
        //gpuErrchk( cudaPeekAtLastError() );
        //gpuErrchk( cudaDeviceSynchronize() );

        // Copy result from device to host
    cudaStatus = cudaMemcpy(ptr, d_ptr, width*height*sizeof(int), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess)
    {
        printf("cudaMemcpy (ptr, d_ptr) returned error code %d\n", cudaStatus);
        exit(EXIT_FAILURE);
    }


    // Device memory clean up
    if (cudaFree(d_ptr) != cudaSuccess)
    {
        fprintf(stderr, "!!!! memory free error (d_ptr)\n");
                return EXIT_FAILURE;
    }
        return 0;
}

extern "C" bool resetCUDADevice()
{
        cudaError_t cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess)
        {
        fprintf(stderr, "cudaDeviceReset failed!");
        return false;
    }

        return true;
}
```

Timers.h

```
//      timers.h
//
//              Wall timer
// and
//              CPU timer
//

#pragma once

#include <iostream>
#include <Windows.h>     // for LARGE_INTEGER and QueryPerformanceFrequency function

//===============================================================================
//
// Wall-time timer Variables and functions
//
// Usage:       StartCounter();
//                      ................ // time the elapsed wall-time in seconds of this region
//                      double elapsedTime = GetCounter();
//
//===============================================================================
double PCFreq = 0.0;
__int64 CounterStart = 0;

//Start Counter
void StartCounter()
{
     LARGE_INTEGER li;
     if(!QueryPerformanceFrequency(&li))
         std::cout << "QueryPerformanceFrequency failed!\n";

        //This gives us a value in milli seconds
    //PCFreq = double(li.QuadPart)/1000.0;

         //This gives us a value in seconds
     PCFreq = double(li.QuadPart);

     QueryPerformanceCounter(&li);
     CounterStart = li.QuadPart;
}

//Get Counter
double GetCounter()
{
    LARGE_INTEGER li;
     QueryPerformanceCounter(&li);
    return double(li.QuadPart-CounterStart)/PCFreq;
}

double get_wall_time()
{
    LARGE_INTEGER time,freq;
    if (!QueryPerformanceFrequency(&freq)){
        //  Handle error
        return 0;
    }
    if (!QueryPerformanceCounter(&time)){
        //  Handle error
        return 0;
    }
    return (double)time.QuadPart / freq.QuadPart;
}

//////////////////////////////////////////////////////////////////////////////////////
//
//      CPU timer
```

```
//
//  Usage:      double start = cputimer();
//                      ................ // time the elapsed cpu-time in seconds of this region
//                      double elapsedTime = cputimer() - start;
//
////////////////////////////////////////////////////////////////////////////////////////

double cputimer()
{
    FILETIME createTime;
    FILETIME exitTime;
    FILETIME kernelTime;
    FILETIME userTime;

    if ( GetProcessTimes( GetCurrentProcess( ),
        &createTime, &exitTime, &kernelTime, &userTime ) != -1 )
    {
        SYSTEMTIME userSystemTime;
        if ( FileTimeToSystemTime( &userTime, &userSystemTime ) != -1 )
            return (double)userSystemTime.wHour * 3600.0 +
            (double)userSystemTime.wMinute * 60.0 +
            (double)userSystemTime.wSecond +
            (double)userSystemTime.wMilliseconds / 1000.0;
    }

        return -1.0;   // error
}

// Return user cpu time in seconds
double get_cpu_time()
{
    FILETIME a,b,c,d;
    if (GetProcessTimes(GetCurrentProcess(),&a,&b,&c,&d) != 0){
        //  Returns total user time.
        //  Can be tweaked to include kernel times as well.
        return
            (double)(d.dwLowDateTime | ((unsigned long long)d.dwHighDateTime << 32)) * 0.0000001;
    }else{
        //  Handle error
        return 0;
    }
}
```