Fractal Generation on GPU

Sean Esterkin Advisor: Dr. Don Spickler
Department of Math and Computer Science
Salisbury University, Salisbury, MD 21801

Abstract

Most fractal generation software uses shortcuts and optimizations to run efficiently on a CPU. However with the rising power of graphics processors, the calculation and display of fractals would be much more easily done on a graphics card. My work is to start applying the shortcuts and functionality of free fractal software to code that runs on the GPU using the CUDA programming language.

Fractal Creation

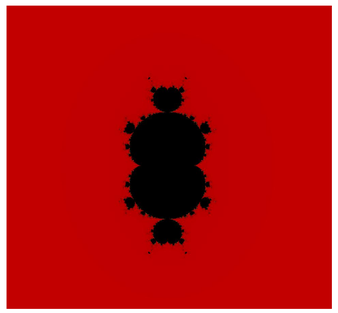
A fractal is generally an image that exhibits self similarity. The fractals that we are concerned with are generated using recurrence relation. Some complex number z(n) is calculated using z(n-1) as well as whatever point on the complex plane we are currently looking at. If z stays within a certain bound after a set number of iterations, it is considered to be part of the set. If it breaks out of the bound, then it is not part of the set. These are called escape time fractals.

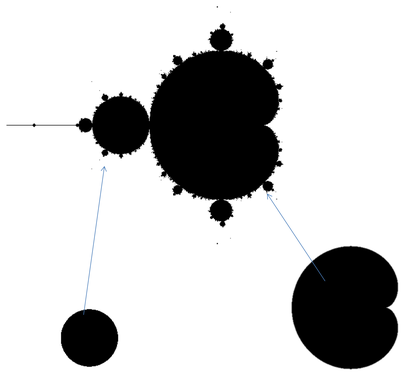
Each point can be tested independently of all of the other points on the complex plane. Thus we can hand off each point to a different processor. On the CPU each point would be tested sequentially. A GPU should be much faster because it can do many points at once.

Acknowledgements

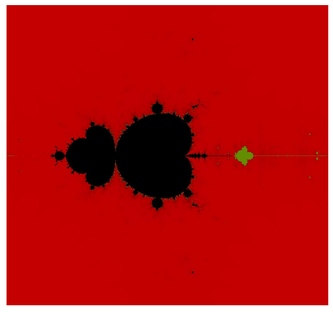
- NSF grant 1156509
- Dr. Enyue Lu
- NVIDIA and CUDA

Z=Z^3+C
Bounds (-2.5,-2.5) to(2.5,2.5)

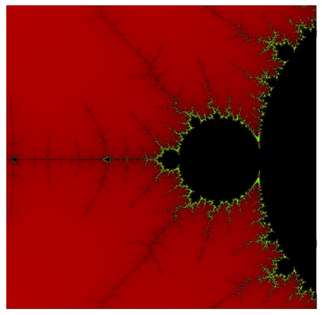




Z=Sin(Z^2)+C
Bounds (-2.5,-2.5) to(2.5,2.5)



Z=Z^2+C
Bounds (-1.45,-.05) to(-1.35, .05)



Kernel

```
dim3    threads(numb,numb);
dim3    grids((DIM+threads.x-1)/threads.x,(DIM+threads.y
                                -1)/threads.y);
kernel<<<grids,threads>>>(d->dev_bitmap,x,y,r);
```

Period 2 Check

```
temp = mx+1.0;
temp = temp*temp+yy;
if(temp<.0625) {
//Part of the Mandelbrot set
return;
}
```

Complex Sine function

```
__device__ cuComplex
complexSin(void) {
float real = sin(r)*cosh(i);
float imag = cos(r)*sinh(i);
      return cuComplex(real,
imag);
```

Cardioid Check

```
float yy = my*my;
float temp = mx-.25;
float q = temp*temp+yy;
float a = q*(q+temp);
float b = .25*yy;
if(a<b) {
//Part of the Mandelbrot set
return;
}
```

Timings

Test Machine: XPS15 running Windows 7 with a GT540m graphics card (96 CUDA cores) with an i7 2720qm processor (2.2GHz with boost up to 3.3GHz) and 8GB of RAM.

Optimizations with dimension 1024x1024 , 200 iterations and blocks of 16x16 threads:

- 30 fps without optimizations
- 33 fps with cardioid and period 2 bulb checks
- 22 fps with periodicity checking

Optimizations with dimension 1024x1024, 200 iterations and cardioid and period 2 bulb checks:

- 27 fps with blocks of 8x8 threads
- 33 fps with blocks of 16x16 threads
- 31 fps with blocks of 32x32 threads

Runtime with CPU version of software with dimension 1024x1024, 200 iterations

- 3.8 fps without optimizations
- 12.8 fps with cardioid and period 2 bulb check
- 4.5 fps with periodicity checking

Process

The base code used was provided by CUDA by Example but needed to be heavily modified before it was useful. The first test was to determine which block size runs the fastest. Blocks of 16x16 threads were the fastest and most reliable.

The next test was to find if other display methods would create the image faster. These methods proved to be more complex and no more effective and were not used.

The Mandelbrot set generation can be optimized by checking if points lie within the cardioid or period 2 bulb. Other sets cannot be calculated that way and so periodicity checking could help speed up calculations.

With more time solid guessing could be applied to the program. Solid guessing requires that certain points be calculated before others which is more difficult to do with parallel processing.

Moving Forward

- Optimize threads per block
- Implement better Periodicity checking
- Implement other shortcuts
  - Solid guessing
  - Dynamic resolution

References

- http://locklessinc.com/articles/mandelbrot/
- CUDA By Example