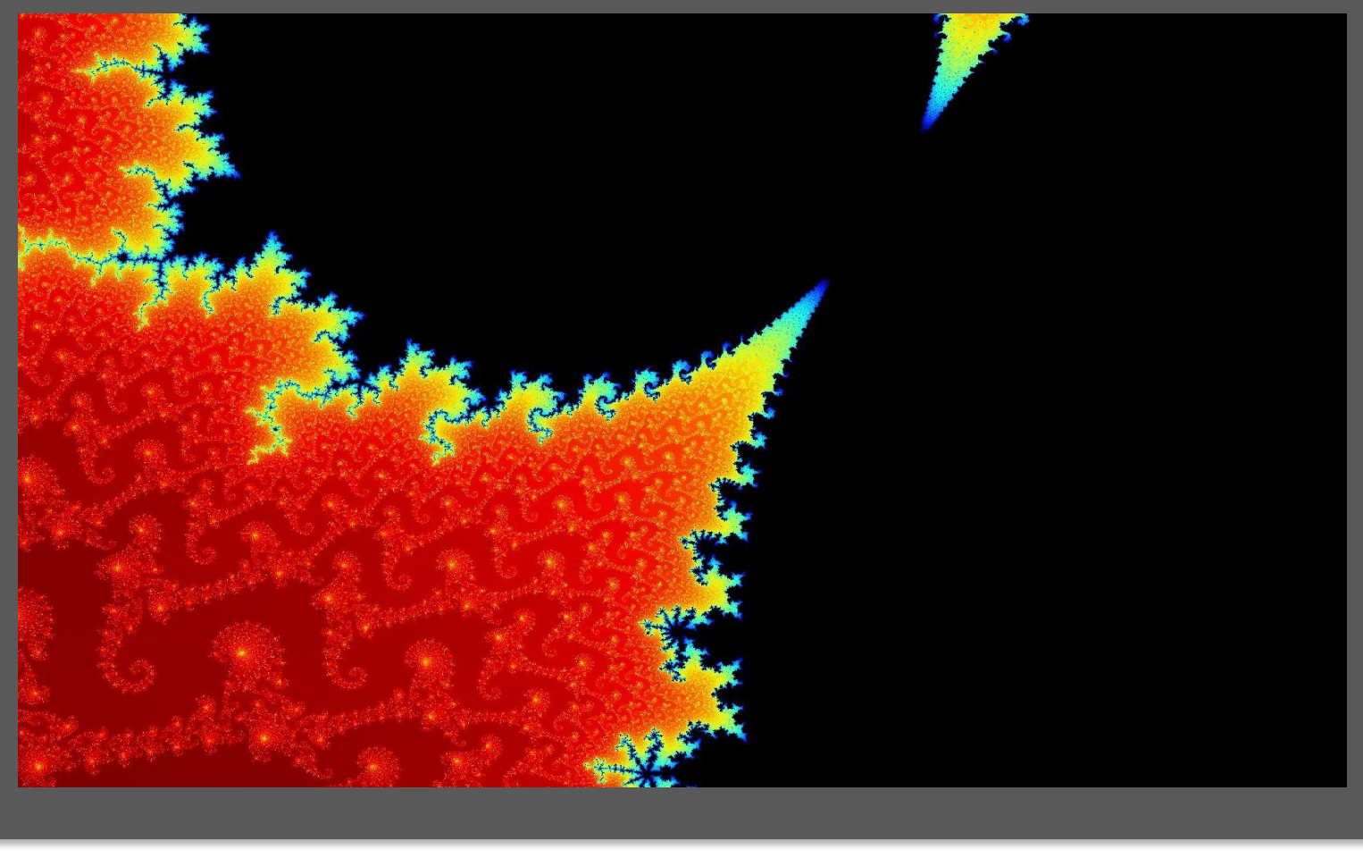
Zooming into the Mandelbrot Set using CUDA Programming

David Andersen[1], Juana Moreno[2], Mark Jarrell[2], Ingmar Schoegl[3]
[1]Instructor – LSMSA, [2]Physics Professor – LSU, [3]Associate Professor - LSU

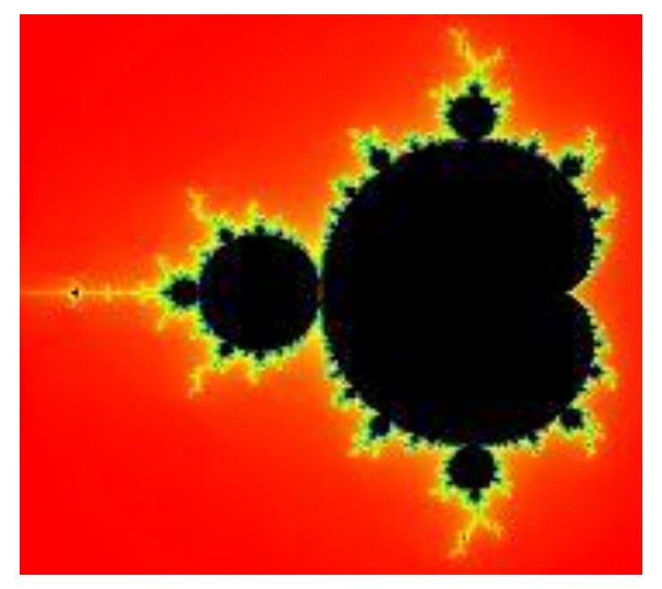


The Goal

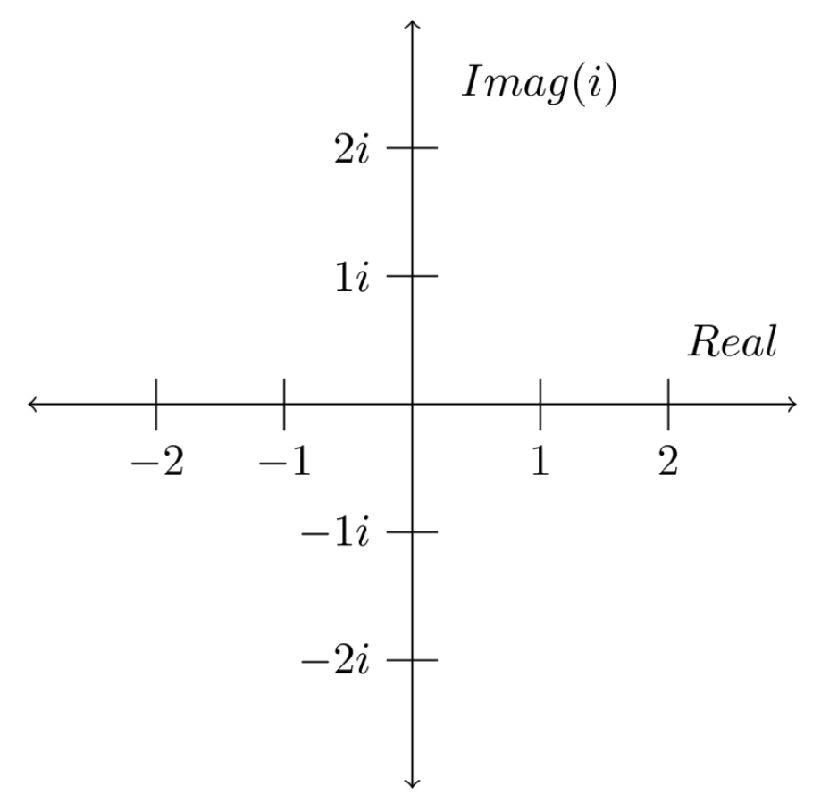
Learn parallel programming using NVIDIA's Compute Unified Device Architecture (CUDA) language and offer a CUDA class at my home school, LSMSA.

The Vehicle

Write a program that would use parallel programming to draw the Mandelbrot Set (Pictured to the right) and zoom into any desired position.



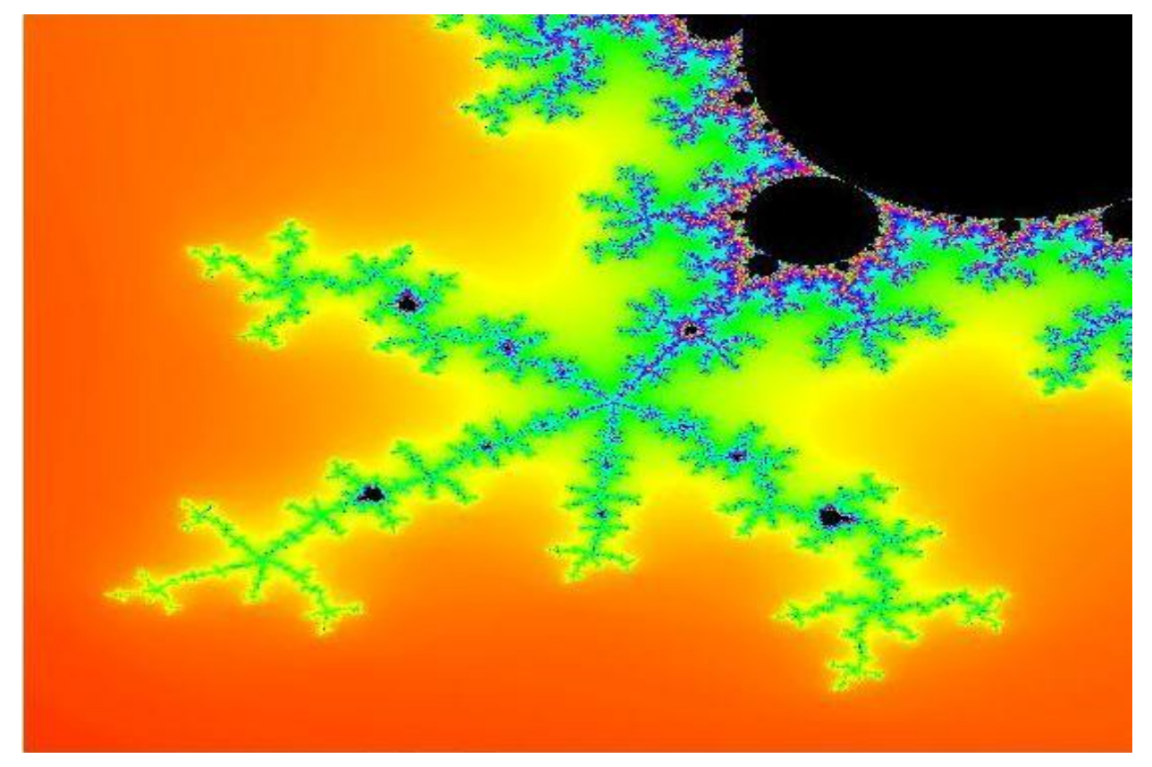
The Construction

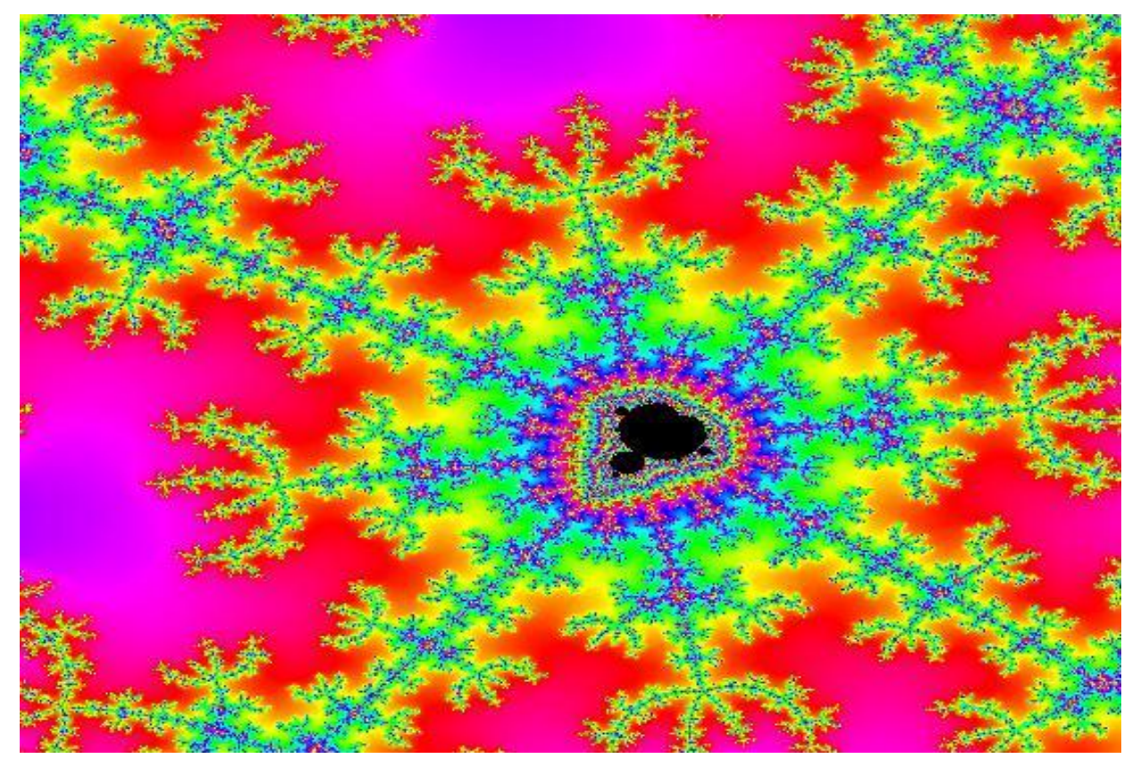


Consider the monitor screen as a complex plane (shown to left). Every pixel becomes a complex number, a + bi. For each pixel use the "complex quadratic"…

$$Q(z) = z^2 + c$$

where c = a + bi and z starts at 0 + 0i. Iterate Q to get a list of complex numbers. If this list diverges, goes to infinity, color the pixel -- hot colors if it diverges quickly and cold colors if it diverges slowly. The black points, those that don't escape, ARE the Mandelbrot Set.





Notice the black "blips" just outside the set. These "baby" Mandelbrot Sets are at every depth we zoom into. As we zoom we must increase the number of iterations to increase the accuracy which increases the calculation time. We need CUDA!

C++ Method

In C++ we envision calculating the entire screen one pixel at a time using one worker. The outer "for" loop is for the real part of the complex number, a + bi, and the inner "for" loop is for the imaginary part of the complex number. The "while" loop counts up the iterations for the complex function Q. All loops use our one worker.

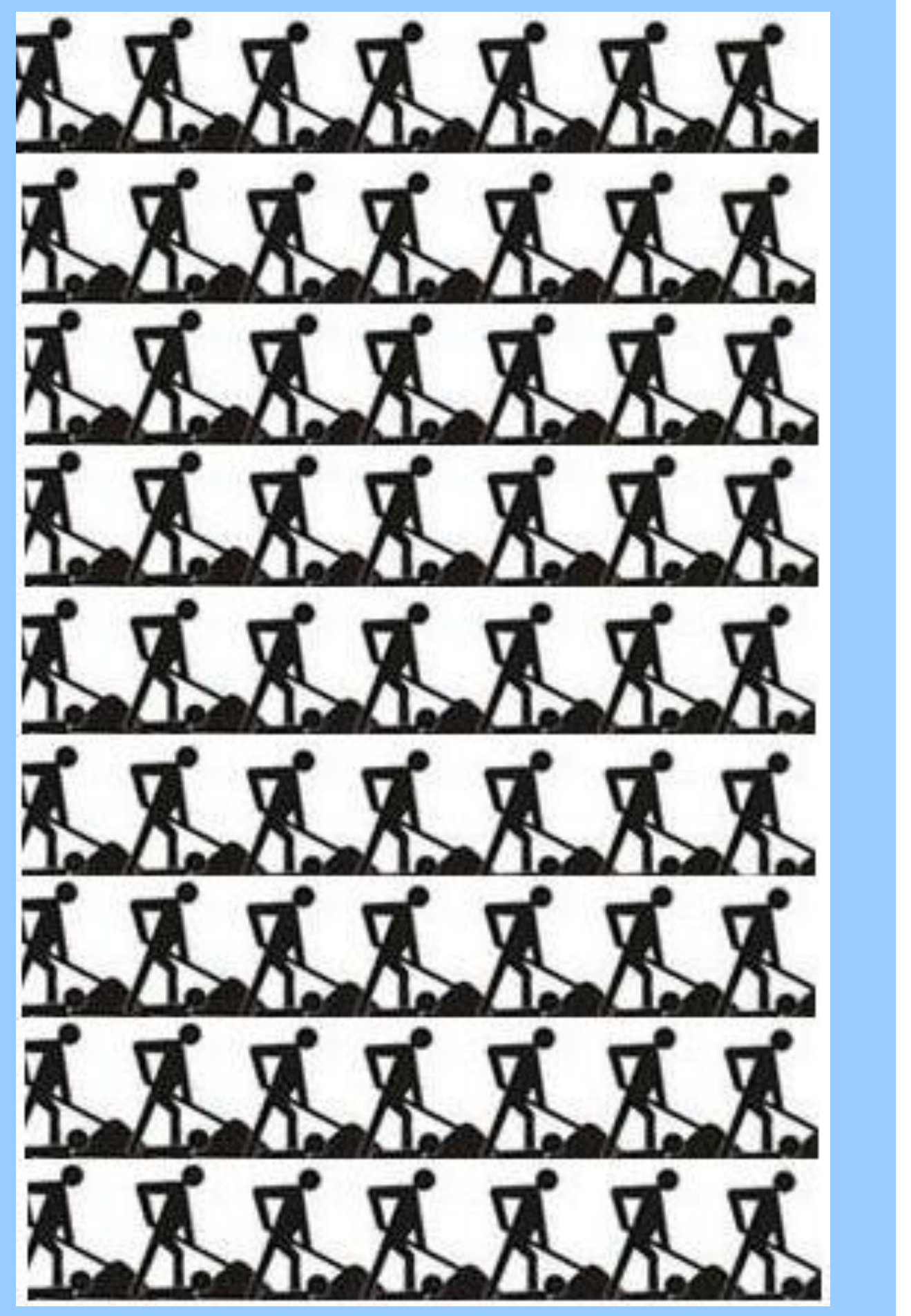
```
for (a = realMin ; a < realMax ; a = a + dx) {
  for (b = imagMin ; b < imagMax ; b = b + dy) {
    xReal = 0;
    yImag = 0;
    iter = 0;
    while (iter < maxIter && xReal*xReal + yImag*yImag < 4) {
      temp = xReal;
      xReal = xReal*xReal - yImag*yImag + a;
      yImag = 2 * temp * yImag + b ;
      iter = iter + 1;
    } // end while
  } // end inner for
} // end outer for
```

This could be 1920 x 1080 x 100 = 207,360,000 calculations!

CUDA Method

In parallel programming we can envision each pixel having its own "worker". For the grid below, there are 256 workers iterating and keeping track of the results. When all the workers are done, they report the number of iterations back to the CPU and the screen is finished. We only need the "while" loop (100 calculations) for the iterations, but we use 1920 x 1080 = 2,073,600 workers to color the screen.

```
1  1  1  1  2  2  2  2  2  2  2  2  2  2  2  2
1  1  1  2  2  2  3  3  2  2  2  2  2  2  2  2
1  1  2  2  3  3  3  3  5  4  3  2  2  2  2  2
1  1  3  3  3  3  4  4  7  8  4  3  2  2  2  2
1  2  3  3  3  4  4  62225  6  4  3  2  2  2
1  3  3  3  4  5  72525252511  3  3  2  2
1  3  4  6  7  7212525252516  4  3  2  2
1  4  5  725252525252512  4  3  2  2
125252525252525252525  6  4  3  2  2
1  4  5  725252525252512  4  3  2  2
1  3  4  6  7  7212525252516  4  3  2  2
1  3  3  3  4  5  72525252511  3  3  2  2
1  2  3  3  3  4  4  62225  6  4  3  2  2  2
1  1  3  3  3  3  4  4  7  8  4  3  2  2  2  2
1  1  2  2  3  3  3  3  5  4  3  2  2  2  2  2
1  1  1  2  2  2  3  3  2  2  2  2  2  2  2  2
```



16 x 16 grid example

CUDA cont'd

The only difference between the CUDA code snippet below and the C++ "while" loop to the left is the [idx]. The "index", allows all threads to be referenced and calculated at the same time.

```
// *** Iteration loop ***
while (d_iter[idx] <  maxiter  && d_radius[idx] <  escapeCriteria) {
    tmp = d_rz[idx];
    d_rz[idx] = d_rz[idx] * d_rz[idx] – d_iz[idx] * d_iz[idx] + d_rc[idx];
    d_iz[idx] = 2 * tmp * d_iz[idx] + d_ic[idx];
    d_radius[idx] = d_rz[idx] * d_rz[idx] + d_iz[idx] * d_iz[idx];
    d_iter[idx] = d_iter[idx] + 1;
} // end while
```

Results and Future Work:

1. FAST! Created 1000 images 1920 x 1080 at 200 iterations in 13 minutes or a picture in less than a second. Created 2000 images using from 200 to 4200 iterations in 56 minutes or 1.68 seconds/image.
2. Python should be capable of managing a "front end" and call the CUDA program for the image data in "real time". Dr. Schoegl is working on it!
3. Limit of 1000 images per run on local workstation. Can get at least 4000 on the GPU queue at HPC.
4. The "double" numeric type allows for only 15 places of accuracy. Use CUMP CUDA Multiple Precision?
5. Difficult to choose good points for zooming.

Acknowledgements

This material is based upon work supported by the National Science Foundation under the NSF EPSCoR Cooperative Agreement No. EPS-1003897 with additional support from the Louisiana Board of Regents.
Many "Thanks" to …
Brad Burkman LSMSA, Zhifeng Yun CCT LSU,
Manjunatha Shenoy CCT LSU, Evan Cordell Tulane University & NewAperio,