Improvement of The Bisection Method Using Parallelism

Tariq Juman, Miguel A. Jardines, Daniel Raad, and Carlos M. Martinez

Florida International University

Abstract

The Bisection method is a root finding algorithm that continuously cuts an interval in half until it finds and returns the root. The downside with the bisection method is that it is generally considered to be slow, since the method must be iterated multiple times to find the root. So, the aim of the project was to use parallelism to reduce the amount of iterations required to find the root and reduce the time complexity of the algorithm. Miguel came up with the idea and general approach of the project, Tariq coded the method to CUDA, Carlos helped convert the original method to C, Daniel helped put together the overall presentation of the project.

Introduction

Globally, the innovation of technology continues to flourish exponentially. Computer Scientists continue to explore on the improvement of algorithms with the specific purpose of finding a faster time complexity. These findings impact our world by enhancing the efficiency of the economy. Companies like PayPal, which revolutionized the way we transfer money, and companies like Google, which allows anyone to search anything on the internet faster, are examples of the good impacts caused by improving the time complexity of algorithms. As Undergrads seeking a degree in Computer Science, we aim for the improvement in time complexity of algorithms and are very interested in reducing the complexity of one of the most popular root finding methods for approximating the solution to a continuous function within a given interval.

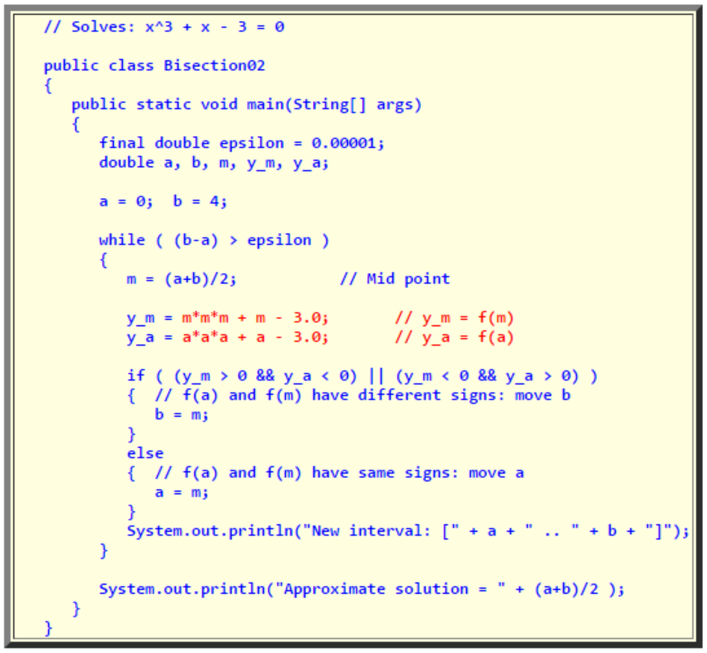
The main issue with the bisection method is how slow it is in comparison with other similar methods such as, secant, Newton's and fixed point iteration. This is because the nature of the bisection method requires it to go through multiple different iterations to locate the root of a given interval leading to a longer execution time. We propose to use multithreading to attempt and reduce the number of iteration the method must go through to improve the overall time complexity of the algorithm by using multiple threads to bisect the interval into many different smaller intervals and then apply the bisection method on this now smaller interval and check if the root is located in one of these smaller intervals until we find the root.

Related Works

There are methods similar to the bisection method that has been developed in order to attempt to be more optimized than the bisection method such as the Newton method. The Newton method is another root finding algorithm such as the bisection method. The Newton method uses tangent lines in order to attempt and find the root. It uses only one point which is a guess unlike bisection which requires an interval but a major downside is of the Newton method is that it struggles in aspects such as not being guaranteed to finding a root for larger intervals. For example, $f(x) = \tan^{-1}(x)$, due to the fact that the function has such a small slope the newton method will continuously guess that the root is further and further away from its actual location leading to the method running either forever or until failure, whereas the bisection method will always find a root leading to it being much more consistent and worth investing time on optimizing. Previous parallelism is been applied to specific cases such as Parallel Hybrid Algorithm of Bisection and Newton- Raphson methods to findnonlinear equation roots[2].

Process

First off we got the original bisection method in Java and transferred it to C language so we could then in turn use it for CUDA which is what we all ultimately chose to use due to the fact that we could use the multiple cores in order to be able to use multithreading in order to run the bisection method in parallel.



```
// Solves: x^3 + x - 3 = 0

public class Bisection02
{
   public static void main(String[] args)
   {
      final double epsilon = 0.00001;
      double a, b, m, y_m, y_a;

      a = 0;  b = 4;

      while ( (b-a) > epsilon )
      {
         m = (a+b)/2;           // Mid point

         y_m = m*m*m + m - 3.0;       // y_m = f(m)
         y_a = a*a*a + a - 3.0;       // y_a = f(a)

         if ( (y_m > 0 && y_a < 0) || (y_m < 0 && y_a > 0) )
         {  // f(a) and f(m) have different signs: move b
            b = m;
         }
         else
         {  // f(a) and f(m) have same signs: move a
            a = m;
         }
         System.out.println("New interval: [" + a + " .. " + b + "]");
      }

      System.out.println("Approximate solution = " + (a+b)/2 );
   }
}
```

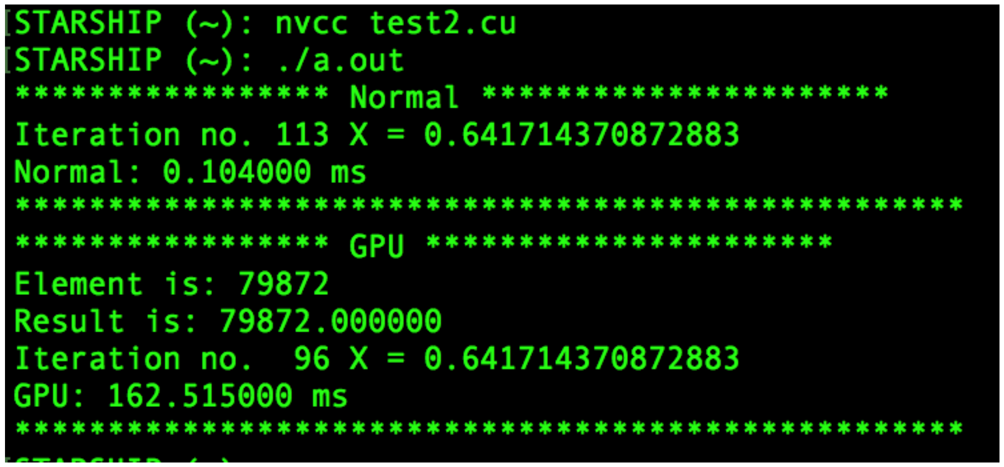
Figure 1. Original bisection method code [1].

After the conversion, we created a function that we would do the bisection on to find a single root, x – tan(x). Which we eventually found to be a bit of an inconsistent function due to the fact that it was not continuous which meant that we had to be careful with our intervals. Reason being, if we had picked an interval such as [4, 5] the root would never be consistent due to the method attempting to find the root towards infinity leading to overflow which then in turn produced an incorrect response. We then changed the function to x – cos^2(x) which resulted into more consistent results. We then started developing the CUDA function for the GPU.

The function used multithreading in order to run the bisection method in parallel by instead of bisecting the interval once, do it multiple times and check each interval in parallel to see which one has the root. Once we are given the interval that has the root we do the original bisection method one the now tiny interval to find the root of the function. The original results were underwhelming as the normal bisection method was performing quicker than the GPU method. This led us to the idea of increasing the normal bisection method complexity by applying it to several intervals with only one root in between. Once the time complexity of normal bisection method matches or exceeds the minimum GPU time complexity of 160ms then we can optimize the parallel bisection method by optimizing the number of threads used, the memory alignment and thus the number of cores. We're trying a different function for this process, 128sin(x/256). Which is a continuous function with multiple roots that take place at about every 800 units. Now, we chose a huge interval from 400-1,200,000,000 in order to having the original bisection method exceeding the 160 milliseconds complexity of the GPU method.

Results

Unfortunately, we found that during execution of our first attempt of the method(x – tan(x)) we found that the normal bisection method was beating the GPU method we had developed because the minimum GPU access time is roughly 160 ms. Also the intervals we used were too small due to asymptotic nature of the function so we decided to change the function to one that is continuous.

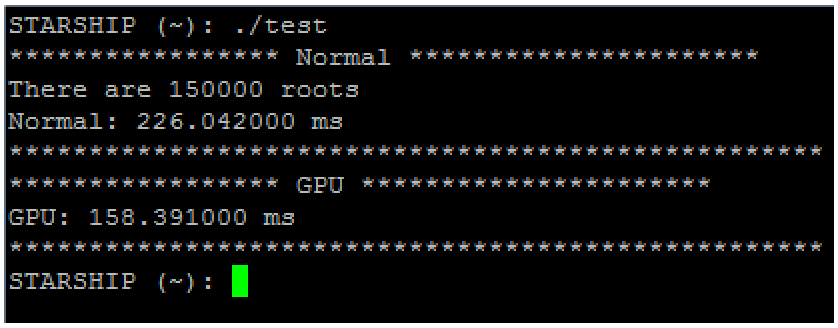


```
STARSHIP (~): nvcc test2.cu
STARSHIP (~): ./a.out
***************** Normal **********************
Iteration no. 113 X = 0.641714370872883
Normal: 0.104000 ms
***************************************************
***************** GPU **********************
Element is: 79872
Result is: 79872.000000
Iteration no.  96 X = 0.641714370872883
GPU: 162.515000 ms
***************************************************
```

Figure 2 An example of the output for our original method

For our second attempt, we used a continuous function x – cos^2(x) this solved the problem of having too small of an interval to make a difference but the GPU was still much slower than the normal bisection method. Everything conceivable was attempted, from using different numbers of threads, increasing and decreasing the number of cores used, increasing error size, and increasing the length of the interval used for testing and still the results stayed the same, the GPU was just much slower than the original bisection method.

But finally, on our third attempt, the GPU was finally much faster than the original. For this attempt, we used the function 128sin(x/256) which was also a continuous function but it also had very large intervals between roots and since we decided to find multiple roots we found that our new execution time for our GPU was 8x faster than the execution time for the original algorithm.



```
STARSHIP (~): ./test
***************** Normal **********************
There are 150000 roots
Normal: 226.042000 ms
***************************************************
***************** GPU **********************
GPU: 158.391000 ms
***************************************************
STARSHIP (~):
```

Figure 3 Output of successful attempt at with smaller numbers

Conclusions

Root finding algorithms have been around for a long time and a huge area of study for both Mathematicians and Computer Science and finding an optimization of one of the most commonly used root finding algorithms will have major implications in improving widely used applications such as GPS based software, and teaching applications such as Wolfram Alpha.

References

1. Cheung, Shun Yan. "The Bisection Method." *Department of Math/CS - Home*. N.p., n.d. Web. 04 Dec. 2016.
2. K. Ali Hussein, A. A. H. Altaee, and H. K. Hoomod, &quot;as Parallel Hybrid Algorithm of Bisection and Newton- Raphson methods to find nonlinear equation roots,&quot; IOSR Journal of Mathematics, vol. 11, no. 4, pp. 32–36, Aug. 2015.