DigiPen Institute of Technology Singapore

CS 398

Mini Project

Alvin Tan, Tham Cheng Jiang, Kenneth Toh, Ong Yong Kiat
12-7-2018

SUMMARY

The team has attempted to accelerate the performance of certain fractal implementation codes using CUDA as compared to CPU.

Results are obtained from Tesla Computer with the following specifications:

Processor: Intel(R) Xeon(R) CPU E5-1620 v4 @3.50GHz

RAM: SK hynix PC4-2400 - 2 x 8GB DDR4 @2400MHz

GPU: NVIDIA GeForce GTX 1060 Graphics Cards (6GB)

HÉNON FRACTAL

BACKGROUND

The Hénon map is a discrete-time dynamical system. It is studied heavily as it is a dynamical systems that exhibit chaotic behavior.

The Hénon map takes a point and maps it to a new point such that:

$$\begin{cases} x_{n+1} = 1 - ax_n^2 + y_n \\ y_{n+1} = bx_n. \end{cases}$$

The map is depended on the parameters a and b such that the classical Hénon map have values of a = 1.4 and b = 0.3. The classical Hénon map is chaotic. For other values of a and b, the map may be chaotic, intermittent, or convergent (to a periodic orbit).

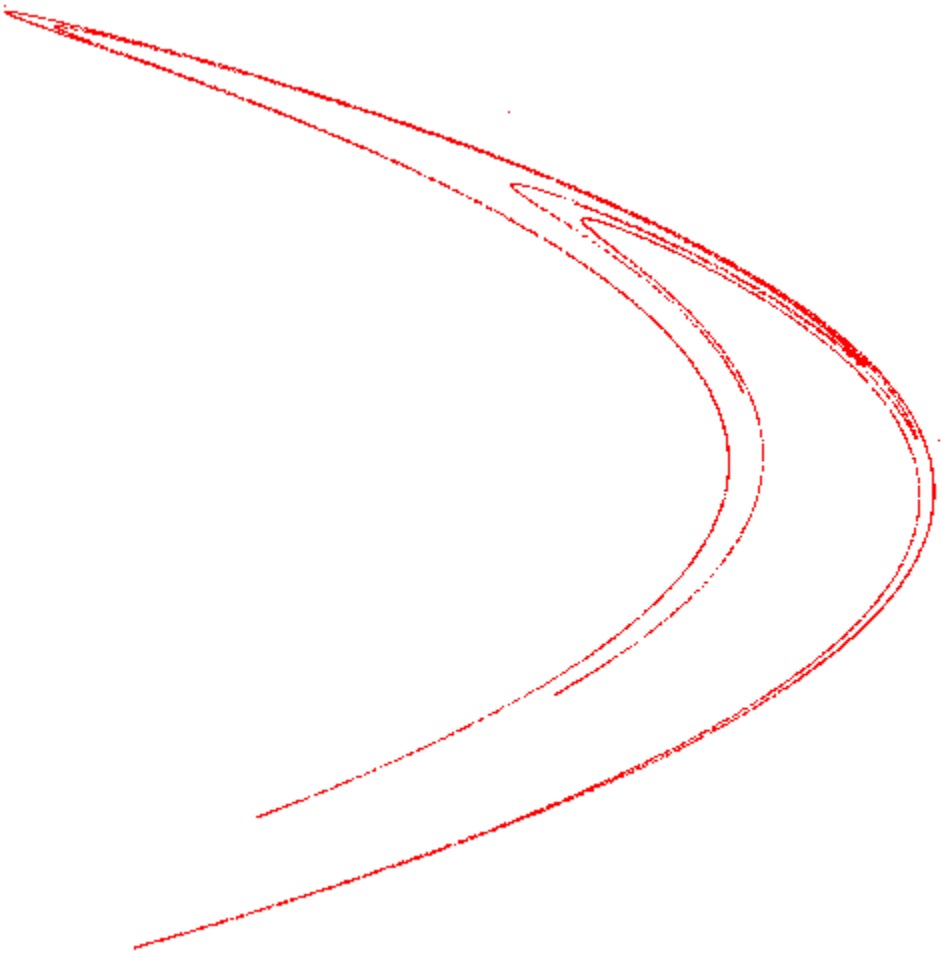
For the classical map, an initial point of the plane will either approach a set of points known as the Hénon strange attractor or diverge to infinity. It is also a fractal, smooth in one direction and a Cantor set in another. Numerical estimates it yields a correlation dimension of 1.25 ± 0.02 and a Hausdorff dimension of 1.261 ± 0.003 for the attractor of the classical map.

APPROACH

Due to the nature of this iterative map that is dependent on it previous value, it is is dropped for CUDA optimization due to the nature of it being extremely tough to optimize using this current mathematical model.

RESULTS

Expected output using the classical map values:



NEWTON FRACTAL

BACKGROUND

The Newton fractal is a boundary set in the complex plane which is defined by Newton's method that is being applied to a fixed polynomial or transcendental function. It is the Julia set of the meromorphic function given from Newton's method.

Using one of the many Newton Fractals, we will use the Julia set for the rational function associated to Newton's method:

Starting function: $f(z) = z^3 - 1$

Derivative function: $f'(z) = 3z^2$

Roots of function: $z = 1, -0.5 \pm \sqrt{3}/2i$

Pseudocode of implementation:

```
For each pixel (x, y) on the target, do:
{
    zx = scaled x coordinate of pixel (scaled to lie in the Mandelbrot X scale (-2.5, 1))
    zy = scaled y coordinate of pixel (scaled to lie in the Mandelbrot Y scale (-1, 1))

    float2 z = float2(zx, zy); //Z is originally set to the pixel coordinates

    float2 roots[3] = //Roots (solutions) of the polynomial
    {
        float2(1, 0),
        float2(-.5, sqrt(3)/2),
        float2(-.5, -sqrt(3)/2)
    };

    color colors[3] =  //Assign a color for each root
    {
        red,
        green,
        blue
    }

    for (int iteration = 0;
         iteration < maxIteration;
         iteration++;)
    {
        z -= cdiv(Function(z), Derivative(z)); //cdiv is a function for dividing complex
numbers

        float tolerance = 0.000001;

        for (int i = 0; i < roots.Length; i++)
        {
            float difference = z - roots[i];

            //If the current iteration is close enough to a root, color the pixel.
            if (abs(difference.x) < tolerance && abs(difference.y) < tolerance)
            {
                return colors[i]; //Return the color corresponding to the root
            }
        }

    }

    return black; //If no solution is found
}
```

APPROACH

As seen in the pseudocode, the algorithm is capable of running in parallel such that for every pixel, we will run on 1 thread. The aim for this fractal would to be to look into how fast parallel programming on the GPU will help as well as to look into the complex calculation Cuda provides.

To start optimizing the equation, I will launch the number of threads to the number of pixels.

By simply running 1 pixel per thread things should speed up significantly as compared to doing it sequentially.

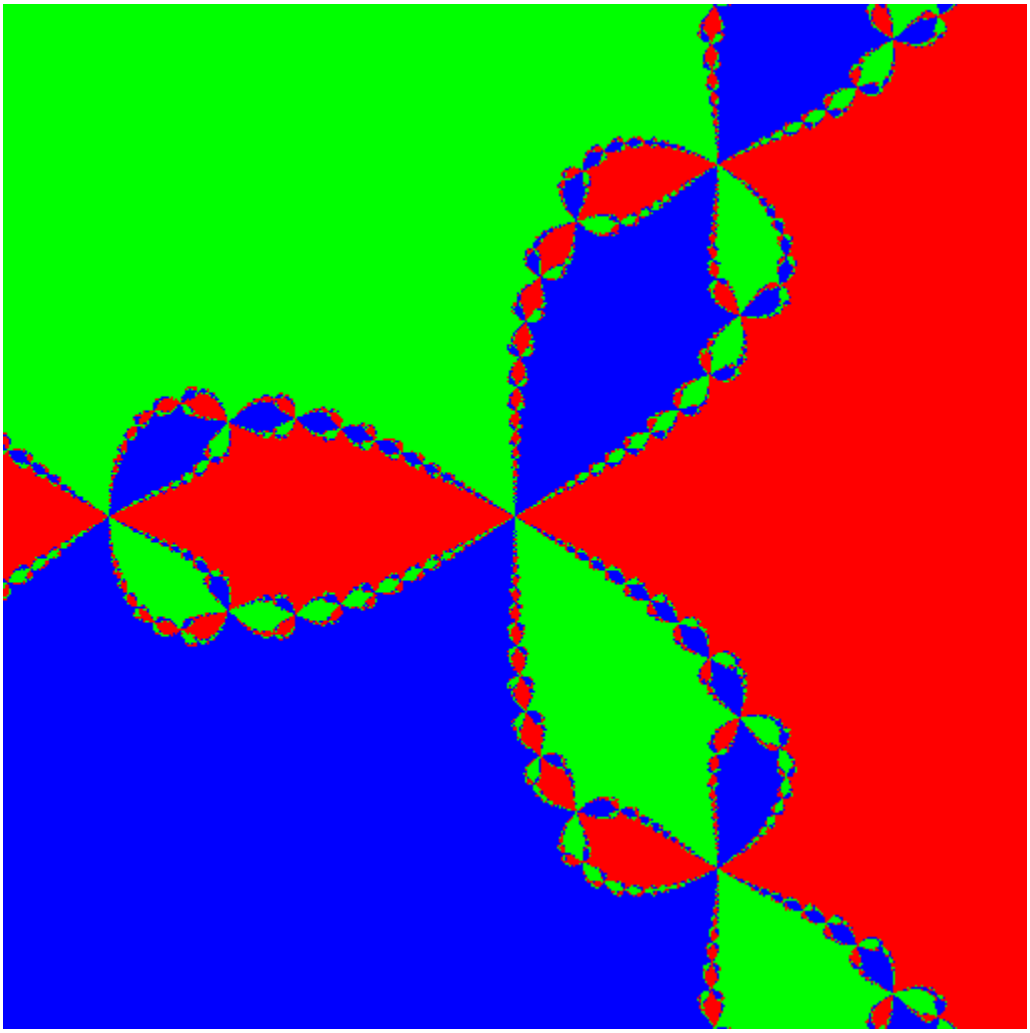
We will test out the different block size and see which will be faster too.

The number of blocks will be given from the dimensions of the image as well as the block size such that: PIXELDIM / BLOCK_SIZE

There is no need for synchronizing the thread as each thread will only write to 1 specific memory location. There is also no need for using pinned and unified memory as there is only 1 copy back and fro to be done

RESULTS

Expected output:



Newton	CPU (s)	GPU (s)	Speed up factor
Thrust (Block - 16)	0.04721	0.00162	29.1x
Thrust (Block - 32)	0.04721	0.00176	26.8x
cuComplex (Block - 16)	0.04721	0.00159	29.7x
cuComplex (Block - 32)	0.04721	0.00174	27.1x

In general, by simply using CUDA programming, we can achieve a speedup factor of more than 20x.

The block size affects the speed slightly such that block size of 32 will lose out 2x speedup as compared to block size of 16. (Within 8-10% difference)

There is a minimal speed up of only 0-2% difference from using the native cuda complex lib (cuComplex) as compared to Thrust:: complex library.

The next possible speed would would be to use lesser precision such as half-precision in the cuda library.

IKEDA MAP

BACKGROUND

Ikeda map is a discrete-time dynamical system given by the complex map

$$z_{n+1} = A + Bz_n e^{i(|z_n|^2 + C)}$$

This was proposed by Ikeda as a model of light going across a nonlinear optical resonator (ring cavity containing a nonlinear dielectric medium) in a more general form.

A indicates the laser light applied from the outside

C indicates the linear phase across the resonator

B <= 1 is called dissipation parameter describing the loss of resonator and if B = 1, the Ikeda map becomes a conservative map.

It is then modified to:

$$z_{n+1} = A + Bz_n e^{iK/(|z_n|^2 + 1) + C}$$

To take the saturation effect of nonlinear dielectric medium into account.

2D approach to this:

$$x_{n+1} = 1 + u(x_n \cos t_n - y_n \sin t_n),$$

$$y_{n+1} = u(x_n \sin t_n + y_n \cos t_n),$$

where u is a parameter and

$$t_n = 0.4 - \frac{6}{1 + x_n^2 + y_n^2}.$$

For u >= 0.6, this system has a chaotic attractor!

Trajectory:

```
%u is the ikeda parameter
%x,y is the starting point
%N is the number of iterations
function [X] = compute_ikeda_trajectory(u, x, y, N)
    X = zeros(N,2);
    X(1,:) = [x y];

    for n = 2:N

        t = 0.4 - 6/(1 + x^2 + y^2);
        x1 = 1 + u*(x*cos(t) - y*sin(t)) ;
        y1 = u*(x*sin(t) + y*cos(t)) ;
        x = x1;
        y = y1;

        X(n,:) = [x y];

    end
end
```

Mapping:

```
% u = ikeda parameter
% option = what to plot
%  'trajectory' - plot trajectory of random starting points
%  'limit' - plot the last few iterations of random starting points
function ikeda(u, option)
    P = 200;%how many starting points
    N = 1000;%how many iterations
    Nlimit = 20; %plot these many last points for 'limit' option

    x = randn(1,P)*10;%the random starting points
    y = randn(1,P)*10;

    for n=1:P,
        X = compute_ikeda_trajectory(u, x(n), y(n), N);

        switch option
        case 'trajectory' %plot the trajectories of a bunch of points
            plot_ikeda_trajectory(X);hold on;

        case 'limit'
            plot_limit(X, Nlimit); hold on;

        otherwise
            disp('Not implemented');
        end
    end

    axis tight; axis equal
    text(-25,-15,['u = ' num2str(u)]);
    text(-25,-18,['N = ' num2str(N) ' iterations']);
end
```

APPROACH

As seen in the pseudocode, the algorithm is capable of running in parallel such that for for every thread that is running, we will paint a pixel for it. The aim for this fractal would to be to look into how fast parallel programming on the GPU and the difference of single precision computation and double precision computation.

To start optimizing the equation, I will launch the number of threads to the number of pixels.

Each thread will denotes the “random” starting point in the Ikeda and continuous iterating over it.

We will test out the different block size and see which will be faster too.

The number of blocks will be given from the dimensions of the image as well as the block size such that: PIXELDIM / BLOCK_SIZE

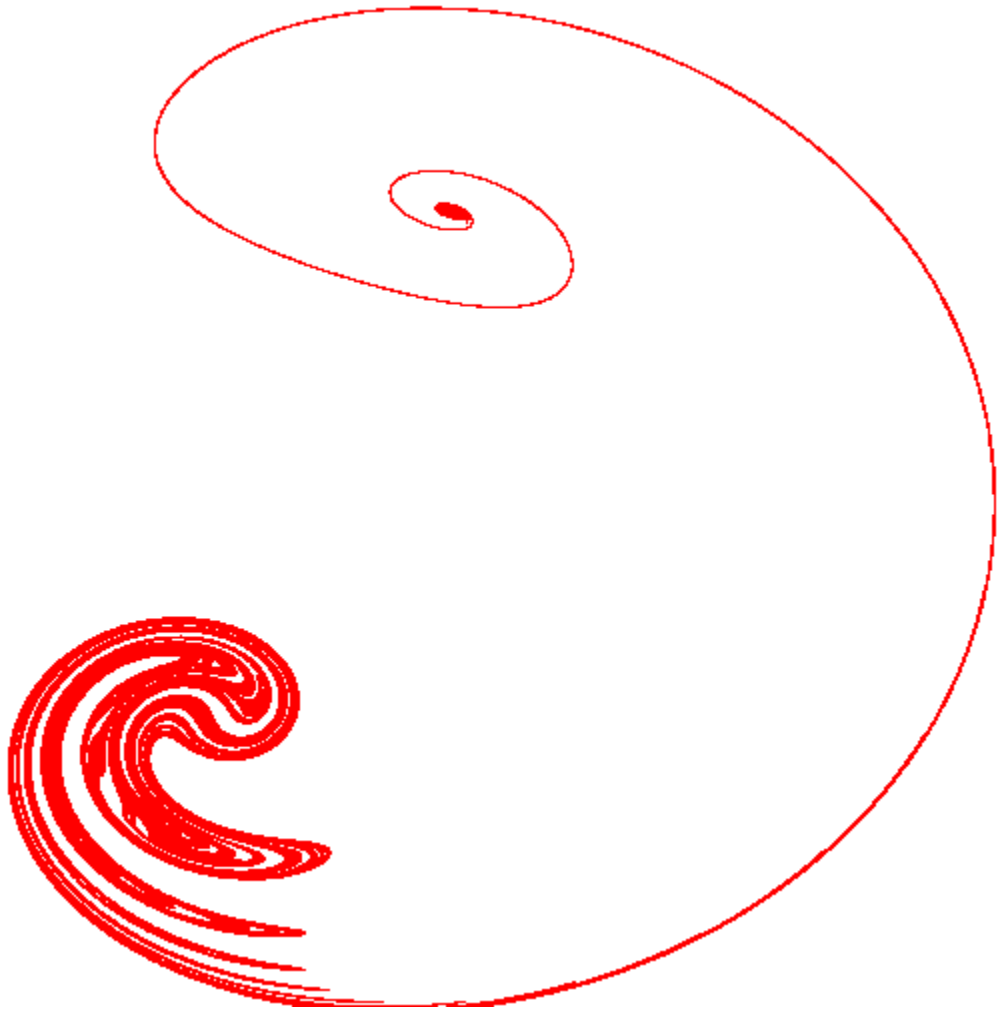
There is no need for synchronizing the thread as each thread will write and not read. If two or more threads is writing to the same position, it will not matter. There is also no need for using pinned and unified memory as there is only 1 copy back and fro to be done

RESULTS

Expected output from using u = 0.918 with mapping:

x [-0.5, 6.5] and y [-2.5, 6.5]

Plotting only iterations (100, 1000]



Ikeda	CPU (s)	GPU (s)	Speed up factor
Single Precision (Block - 16)	6.51056	0.02263	287.7x
Single Precision (Block - 32)	6.51056	0.02323	280.2x
Double Precision (Block - 16)	7.25123	0.39943	18.2x
Double Precision (Block - 32)	7.25123	0.40658	17.8x

The difference between the result of CUDA's single and double precision computing is much larger than expected such that single precision computing is around 15 to 16 times faster than double precision in this Ikeda map test.

As compared to the CPU's result, single precision computing is only 10% faster than the double precision version. This is probably due to the raw processing power of the CPU as well as the much shorter latency involved in the CPU compared to GPU.

Block size affects minimally (0-1% difference) between both block sizes by comparing the raw speed.

BROWNIAN TREE

BACKGROUND

Brownian Tree, is a form of computer art. The idea is firstly to “plant a seed” somewhere on the screen. Then a particle is placed in a random position of the screen, and moved randomly until it bumps against the seed. The particle is left there, and another particle placed in a random position and the process repeats for the number of iterations defined by the user.

The resulting tree can have many different shapes, depending on principally three factors:

1) The Seed Position
2) The Initial Particle Position (anywhere on the screen, from a circle surrounding the seed, from the top of the screen, etc.)
3) The Moving Algorithm (usually random, but for example a particle can be deleted if it goes too far from the seed, etc.)

Because of the lax rules governing the random nature of the particle's placement and motion, no two resulting trees are really expected to be the same, or even necessarily have the same general shape.

For the purpose of our experiment, we would use the Moving Algorithm.

APPROACH

```
14      srand((unsigned)time(nullptr));
15      int px, py; // particle values
16      int dx, dy; // offsets
17
18      // set the seed
19      input[(rand() % PIXELDIM) * PIXELDIM + rand() % PIXELDIM] = 1;
20
21      for (int i = 0; i != BROWNIAN_ITERATIONS; ++i) {
22          // set particle's initial position
23          px = rand() % PIXELDIM;
24          py = rand() % PIXELDIM;
25
26          while (1) {
27              // randomly choose a direction
28              dx = rand() % 3 - 1;
29              dy = rand() % 3 - 1;
30
31              if (dx + px < 0 || dx + px >= PIXELDIM || dy + py < 0 || dy + py >= PIXELDIM) {
32                  // plop the particle into some other random location
33                  px = rand() % PIXELDIM;
34                  py = rand() % PIXELDIM;
35              }
36              else if (input[(py + dy) * PIXELDIM + px + dx] != 0) {
37                  // bumped into something
38                  input[py * PIXELDIM + px] = 1;
39                  break;
40              }
41              else {
42                  py += dy;
43                  px += dx;
44              }
45          }
46      }
```

The approach is to first plant a seed anywhere on the screen, then randomly plant a position and move randomly in a direction (hopefully facing towards the seed). If the particle goes off screen, a new position will be randomized. As noted from the code above, the algorithm requires dependency on the previous iteration.

To optimize the code through GPGPU, we will attempt to parallize the random position to the seed and the first thread to find the seed will break out and inform the rest of the threads that there is a location available. A sample code will be shown here:

```
__global__ void heatDistrCalculation(uchar *data, curandState *state, int py, int px)
{
    int dx, dy;

    // Every thread to perform the same operation and determine who add it first
    while (!mutex)
    {
        randomDirection(state, &dx);
        randomDirection(state, &dy);

        int dpx = dx + px;
        int dpy = dy + py;

        if (dpx < 0 || dpx >= PIXELDIM || dpy < 0 || dpy >= PIXELDIM)
            break;

        else if (data[dpy * PIXELDIM + dpx] != 0)
        {
            // Bumped into something
            atomicOr(&mutex, 1);
            data[py * PIXELDIM + px] = 1;
        }
        else
        {
            py += dy;
            px += dx;
        }
    }

}
```

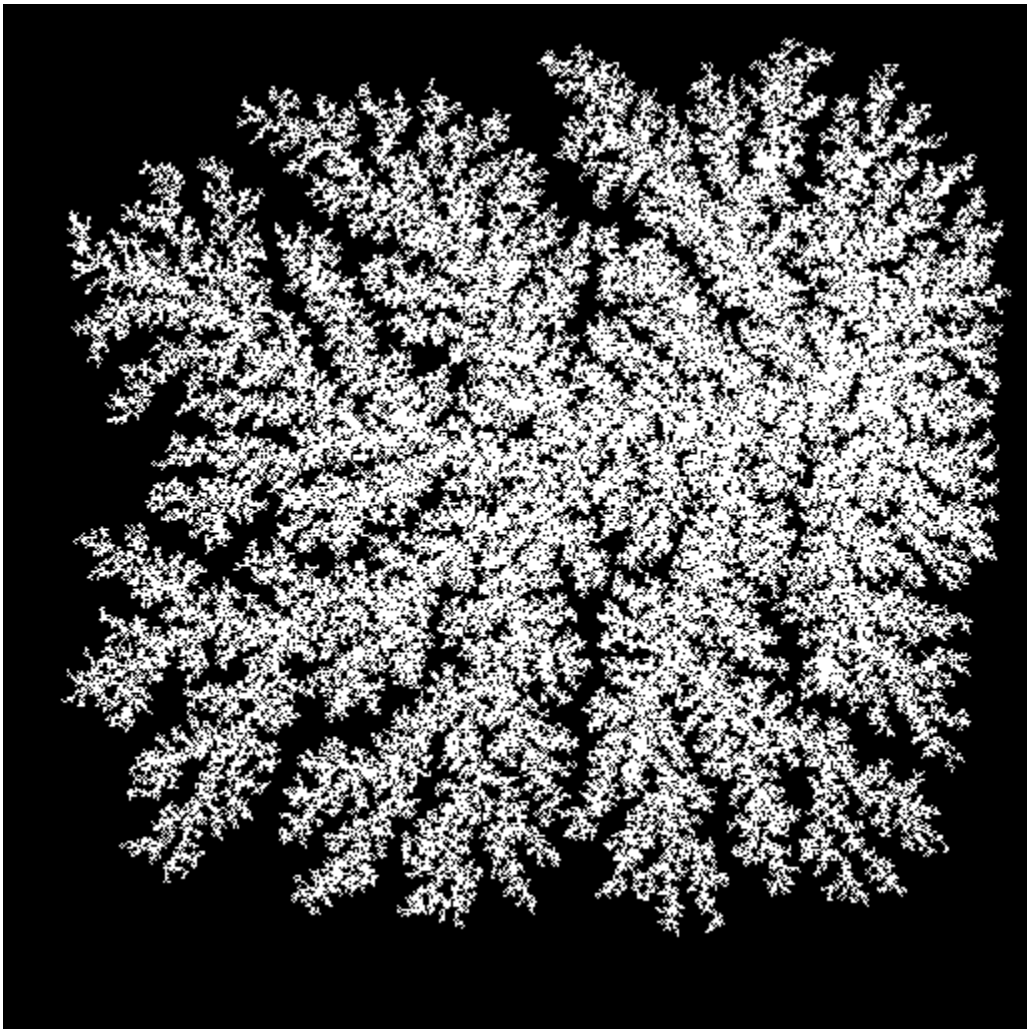
As noticed, a mutex will be required as all threads run in parallel. An alternate way to use the mutex is using “lock” in CUDA with the following code:

```
struct Lock
{
    int *mutex;

    Lock()
        : mutex{ nullptr }
    {
        int state = 0;
        cudaMalloc((void**)&mutex, sizeof(int));
        cudaMemcpy(mutex, &state, sizeof(int), cudaMemcpyHostToDevice);
    }

    ~Lock()
    {
        cudaFree(mutex);
    }

    __device__ void lock()
    {
        while (atomicCAS(mutex, 0, 1) != 0);
    }

    __device__ void unlock()
    {
        atomicExch(mutex, 0);
    }
};
```

Where only one thread can modify the value at any time. And when that value has been modified (from 0 to 1), all threads should break out and join back into the main scene. And repeat until all of the iterations are completed.

RESULTS

The expected output is (100000 iterations):



However, through experimenting with CUDA, it seemed that it does not benefit on the GPU side and even though the usage of mutex, there is a chance where neither thread can find the position to the seed due to randomization.

It will be possible to certain iterations for GPU but the speed will be slow compared to CPU due to the overhead and bottleneck of synchronizing all the threads before the next iteration.

Through this experiment, it is understandable that despite GPGPU's capability to simultaneously run computations in parallel, not every algorithm or problem could be solved through GPGPU, and this experiment proves to be one of them.

Though it the result is achievable, but it did not optimize as intended. In addition of the randomization of the particle's position, it would be possible to determine through how many steps for the individual particle to reach the destination by adding in algorithmic checks - that would ultimately change and defeat the purpose of the Brownian Tree implementation.

SIERPINSKI CARPET

BACKGROUND

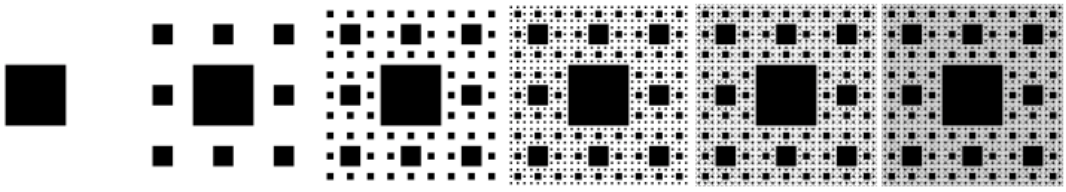
Sierpinski Carpet is a plane fractal and its a technique of subdividing a shape into smaller copies of itself, removing one or more copies, and continuing recursively can be extended to other shapes. In this experiment, we would construct the shape through a square.

APPROACH

The construction of the Sierpinski carpet begins with a square. The square is cut into 9 congruent subsquares in a 3-by-3 grid, and the central subsquare is removed. The same procedure is then applied recursively to the remaining 8 subsquares. It can be realized as the set of points in the unit square whose coordinates written in base three do not both have a digit '1' in the same position.

```
/*
* Decides if a point at a specific location is filled or not.  This works by iteration first checking if
* the pixel is unfilled in successively larger squares or cannot be in the center of any larger square.
* x is the x coordinate of the point being checked with zero being the first pixel
* y is the y coordinate of the point being checked with zero being the first pixel
* 1 if it is to be filled or 0 if it is open
*/
int isSierpinskiCarpetPixelFilled(int x, int y)
{
    while (x > 0 || y > 0) // when either of these reaches zero the pixel is determined to be on the edge
                           // at that square level and must be filled
    {
        if (x % 3 == 1 && y % 3 == 1) //checks if the pixel is in the center for the current square level
            return 0;
        x /= 3; //x and y are decremented to check the next larger square level
        y /= 3;
    }
    return 1; // if all possible square levels are checked and the pixel is not determined
                  // to be open it must be filled
}
```

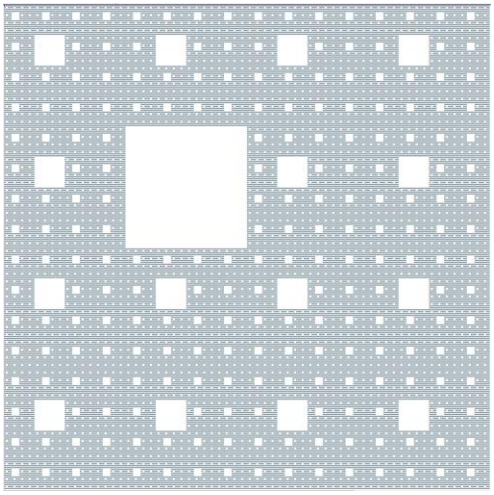
Process [edit]



With some modification to the code, the final CPU algorithm will be:

```
    // Initialization
    int dim = 1, d = 0;
    int depth = SIERPINSKI_DEPTH;
    for (uint i = 0; i != depth; ++i)
        dim *= SIERPINSKI_DEPTH;

    uint width = dim + 2;           // accounting for 2 extra spaces
    uint height = dim + 2;          // just to make it a square
    uint size = width * height;

    // Create the data for printing
    uchar *data = new uchar[size]{ };

    // Performing calculation
    for (int i = 0; i != dim; ++i) {
        for (int j = 0; j != dim + 1; ++j) {
            for (d = dim / SIERPINSKI_DEPTH; d; d /= SIERPINSKI_DEPTH)
                if ((i % (d * SIERPINSKI_DEPTH)) / d == 1 && (j % (d * SIERPINSKI_DEPTH)) / d == 1)
                    break;
            // fprintf(file, d ? " " : "#");
            data[i * width + j] = d ? WHITESPACE_PRINT : HEX_PRINT;
        }
        // fprintf(file, "\n");
        data[i * width + dim] = NEWLINE_PRINT;
    }
```

RESULTS



And the relevant table will be shown below for the performance difference

Dimension	Memory Type	CPU	GPU
29	Pinned	0.00096s	0.00424s
	Unified	0.00095s	0.02435s
	Managed	0.00090s	0.00174s
258	Pinned	0.00947s	0.00484s
	Unified	0.00996s	0.03243s
	Managed	0.00915s	0.01163s
3127	Pinned	1.27223s	0.97100s
	Unified	1.29957s	1.08174s
	Managed	1.26726s	0.99038s
46656	Pinned	299.12525s	217.25147s
	Unified	300.42934s	233.11236s
	Managed	295.73384s	226.78161s

In GPGPU, we have parallelized the computation of each “pixel” (to determine if we should display the square). In addition, we have also maximized our thread usage by using all of the available threads in the GPU to help further parallelize the computation of each “pixel”.

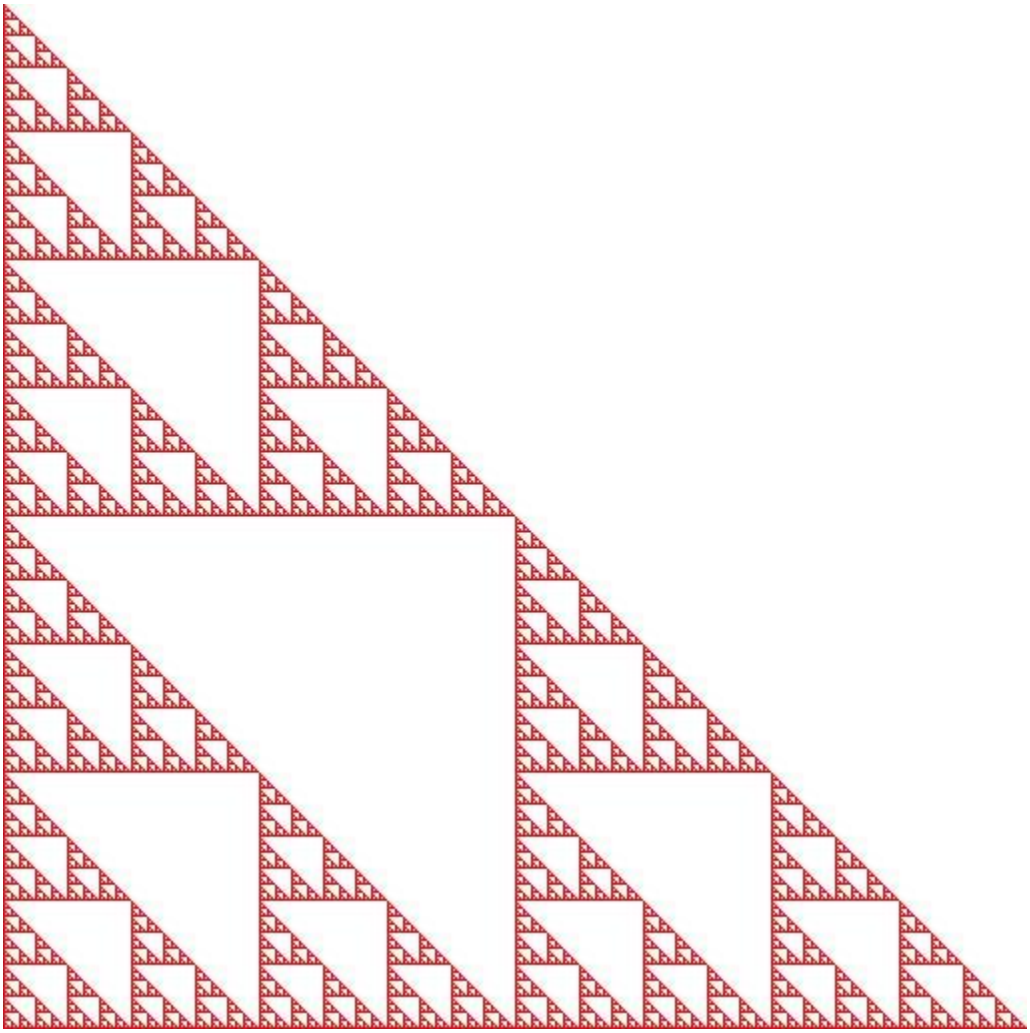
The ideal implementation and performance was to have it increase the performance speed by at least 50%. However, it is not achievable due to limitations such as writing the data into a file. In this aspect, CUDA does not have the ability to write data into files simultaneously and as such, one of the bottleneck comes from the data transfer into a file.

Another bottleneck to this experiment is the overhead of transferring the data from device to host and vice versa. It is not noticeable in smaller dimensions but as the values get bigger, the performance shows a difference (through nvprof command line).

SIERPINSKI TRIANGLE

BACKGROUND

The Sierpinski Triangle also known as the Sierpinski gasket or the Sierpinski Sieve is an Iterative Fractal Set. The Sierpinski Triangle takes the shape of a big equilateral triangle which produce a end result of more subdivided triangles using recursion.



APPROACH

With respect to the aboved pseudocode, the main issue was to get the calculations for the position to where i put colors on it.

```
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y* blockDim.y;
```

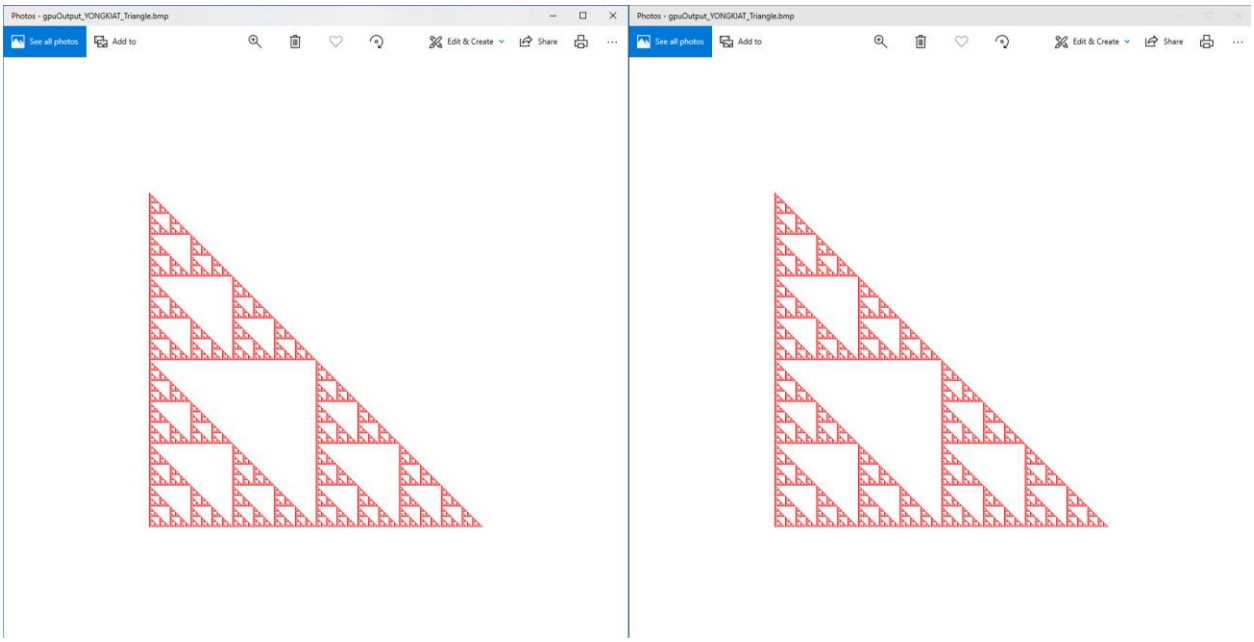
```
for (; x + y < PIXELDIM;++x)
{

    // printing '*' at the appropriate position
    // is done by the and value of x and y
    // wherever value is 0 we have printed '*'
    if ((x&y))
    {

        d_DataOut[x + PIXELDIM * y] = 0XFF; // b
        d_DataOut[x + PIXELDIM * y + PIXELDIM2] = 0XFF; // g
        d_DataOut[x + PIXELDIM * y + PIXELDIM2 + PIXELDIM2] = 0XFF; // r
        //outFile << " " << " ";
        //SetData(x, y, 255, data);
        //SetData(x, y, 255, data);
    }
    else
    {
        d_DataOut[x + PIXELDIM * y] = 0x00; // b
        d_DataOut[x + PIXELDIM * y + PIXELDIM2] = 0x00; // g
        d_DataOut[x + PIXELDIM * y + PIXELDIM2 + PIXELDIM2] = 0xff; // r
        //SetData(x, y, 0, data);
        //outFile << "* ";

    }

}
```

Tried a text version of the Sierpinski Triangle print just ‘*’ with “ “ .

RESULTS

```
CPU time (average) : 0.00033 sec, 2350.3847 MB/sec
CPU Fractal, Throughput = 2350.3847 MB/s, Time = 0.00033 s, Size = 786432 Bytes, NumDevsUsed = 1

GPU time (average) : 0.00305 sec, 258.1871 MB/sec
GPU Fractal, Throughput = 258.1871 MB/s, Time = 0.00305 s, Size = 786432 Bytes, NumDevsUsed = 1
Press any key to continue . . .
```

This is the result of computing Sierpinski Triangle using CPU and GPU. We are not able achieve the speed up factor for the GPU using parallelism or other memory methods.



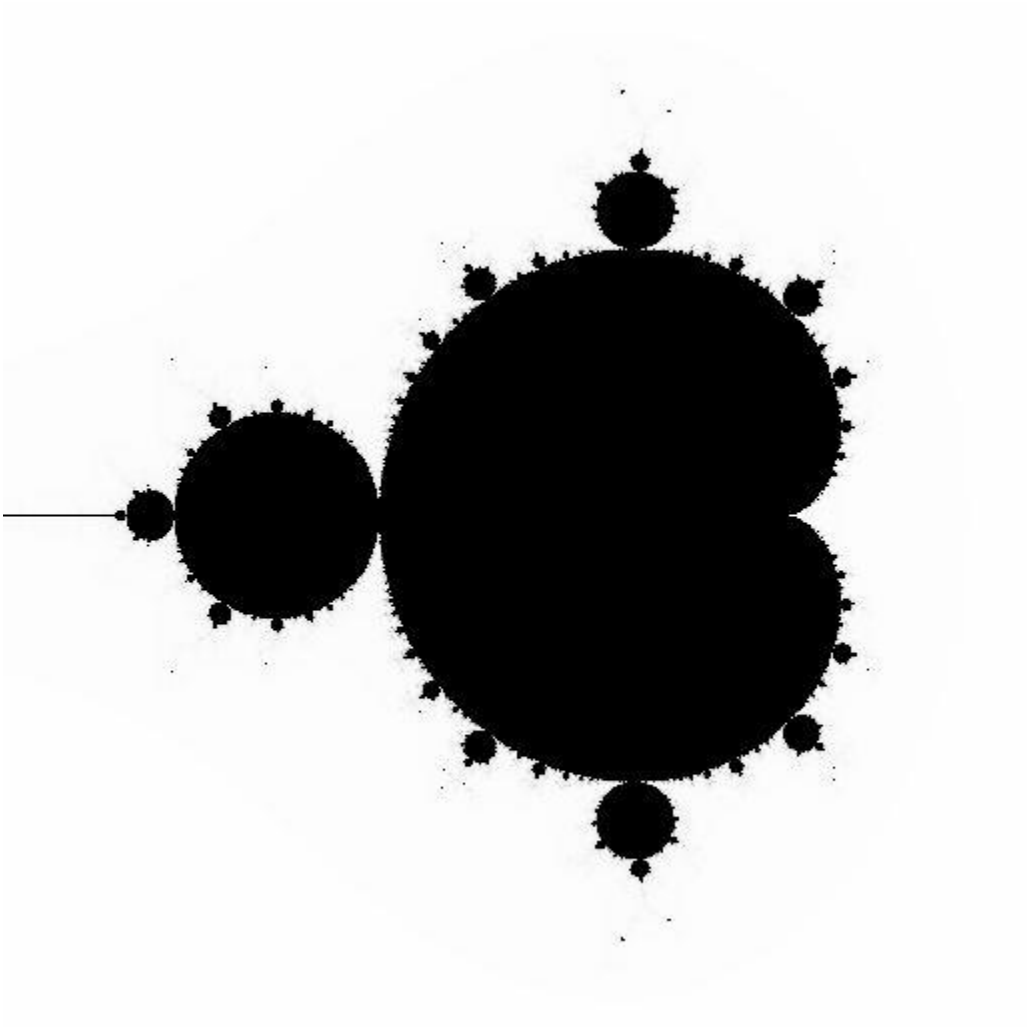
Even though we are not able to speed up the process, we still managed to get the two outputs correctly.

Through this experiment, it is understandable that despite GPGPU's capability to simultaneously run computations in parallel, not every algorithm or problem could be solved through GPGPU, and this experiment proves to be one of them.

MANDELBROT SET

BACKGROUND

The Mandelbrot Set is a set of connected points in the complex plane. The x and y coordinates of each point are the starting values of a iterative calculations. With the result of each iteration being used as the starting values of the next iteration.



Pseudocode provided by https://en.wikipedia.org/wiki/Mandelbrot_set

```
For each pixel (Px, Py) on the screen, do:
{
  x0 = scaled x coordinate of pixel (scaled to lie in the Mandelbrot X
scale (-2.5, 1))
```

```
  y0 = scaled y coordinate of pixel (scaled to lie 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)
}
```

APPROACH

With respect to the above pseudocode, the main issue was to get the calculations of the next values using previously computed values.

To proceed on, setup a two-dimensional grid and block, and we will ignore the case if the mapping is greater than our original image size.

```
int tx = threadIdx.x + blockIdx.x * blockDim.x;
int ty = threadIdx.y + blockIdx.y * blockDim.y;

if (tx >= PIXELDIM || ty >= PIXELDIM)
    return;
```

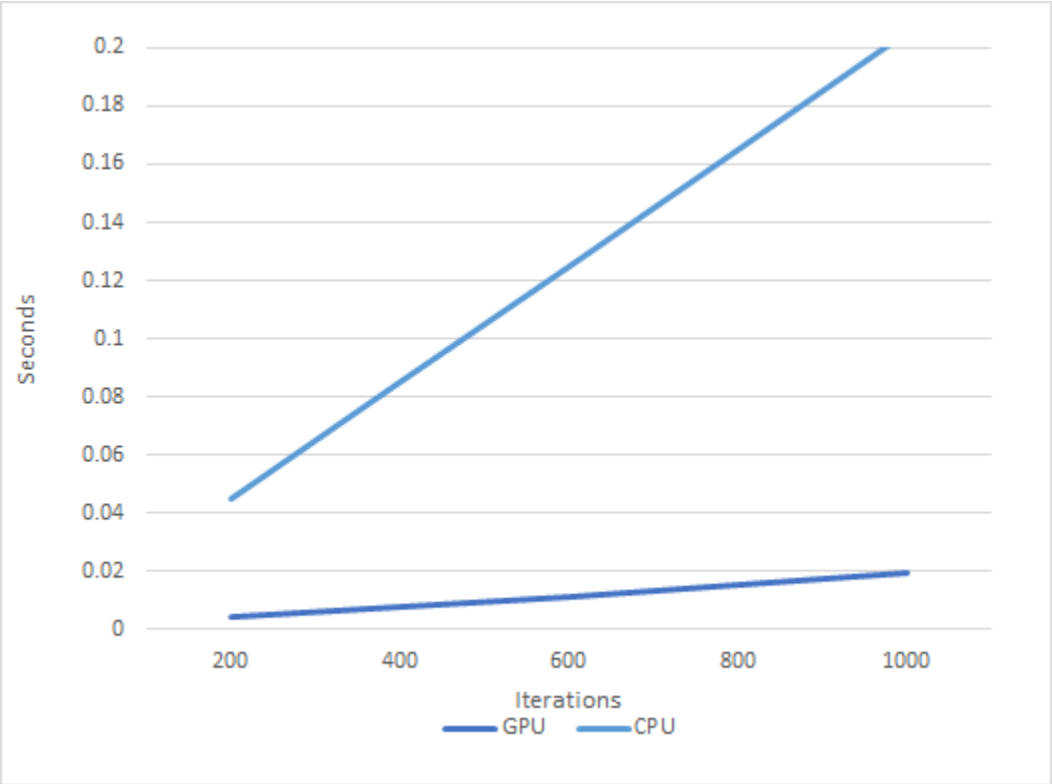
Next, we set up the necessary variables in the function, when every iterations these variables will be off different values.

```
double a = 0.0;
double b = 0.0;
double norm2 = 0.0;
int n;
double x = (double)((tx + shiftBS2) *magBS) / PIXELDIM;
double y = (double)((PIXELDIM - 1 - ty + shiftBS) *magBS) / PIXELDIM;
double iter = 1;
```

From the above variables tx and ty, it will update the respective x and y values. With the new x and y values, we are ready to do the actual calculations of the Mandelbrot.

```
for (n = 0; norm2 < 4.0 && n < iterationBS; ++n)
{
    double c = a*a - b*b + x;
    b = 2.0 * a * b + y;
    a = c;
    norm2 = a*a + b*b;

}
```

RESULTS



From the above graph, we can see that we succeeded in improving the speed for the Mandelbrot calculations.

BURNING SHIP FRACTAL

BACKGROUND

The burning ship fractals is an equation in the complex plane that takes in position of row and column and generate an image that is capable of infinite zoom. Which at a certain point, an impression of a burning ship can be seen. The equation goes through an input number of iterations which smoothen the differences.

Pseudo code given by Wikipedia : https://en.wikipedia.org/wiki/Burning_Ship_fractal

```
zx = x; // zx represents the real part of z
 8     zy = y; // zy represents the imaginary part of z
 9
10
11     iteration = 0
12     max_iteration = 1000
13
14     while (zx*zx + zy*zy < 4  AND  iteration < max_iteration)
15     {
16         xtemp = zx*zx - zy*zy + x
17         zy = abs(2*zx*zy) + y //abs returns the absolute value
```

```
18          zx = abs(xtemp)
19
20          iteration = iteration + 1
21      }
```

Since for the burning ship fractals, each pixel or point does not affect the surrounding points, hence there is definite capability of parallelism that could be used(zero dependencies). Hence my focus would be testing of page-able, unified and pinned memory, in addition to intrinsic commands, to observe the speed up.

APPROACH

When considering the optimization, I realize that the main focus would be optimizing the equations since the program for each point would be spending most time in the loop base on iterations in-order generate the color of that pixel.

Steps:

Use of two dimension grid and block to get the x and y, then apply the shift and zoom needed to look at the burning ship

Next, apply the equations with the input iterations.

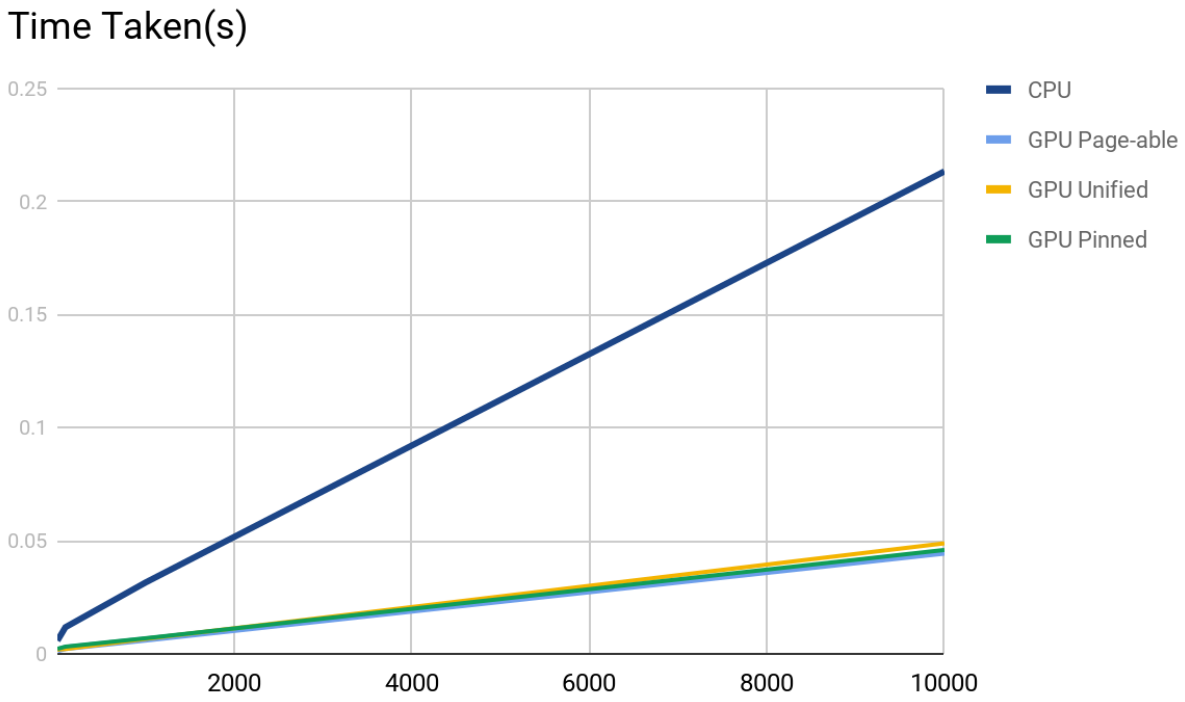
Set-up the page-able, unified and pinned memory to test with the GPU functions

Lastly, convert all double calculation and casting into intrinsic.

Update:

Convert all double calculation into half precision from <cuda_fp16.h>

RESULTS



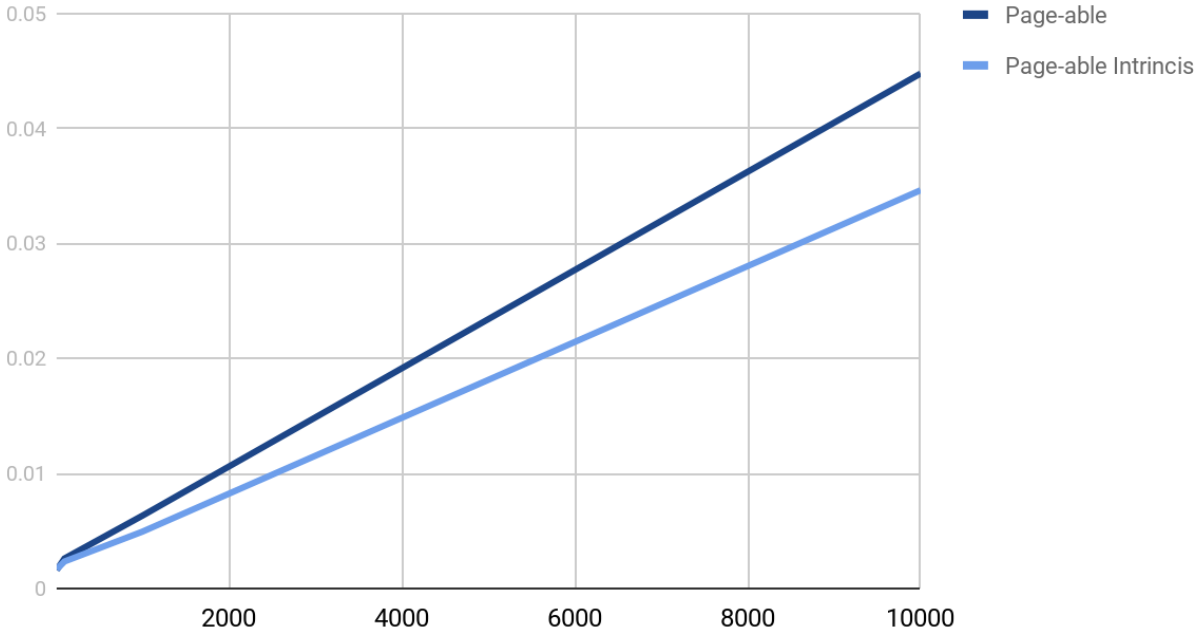
As predicted, the GPU of the equations is faster than the CPU due to the parallelism (with roughly 5 times as compare to the CPU at 10000 iterations)

Next, I observed that the unified memory after a 1000 iterations have a slower time as compare to the Page-able memory. Which indicate that after an 1000 iterations, the write of the page-able memory is faster that the single memcpy back to host as compare to the write of the unified memory (main difference between the two memory).

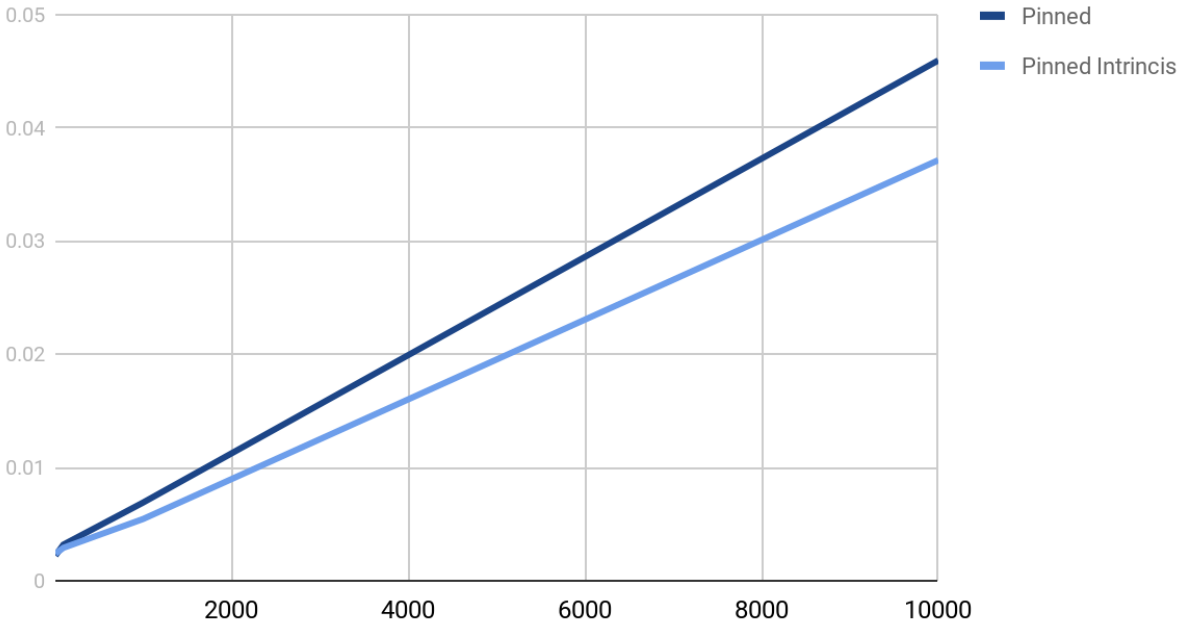
Lastly, I observed that the pinned memory did not perform efficiently since only a single memcpy is done and that the memory allocation for pinned memory is greater than the page-able memory.

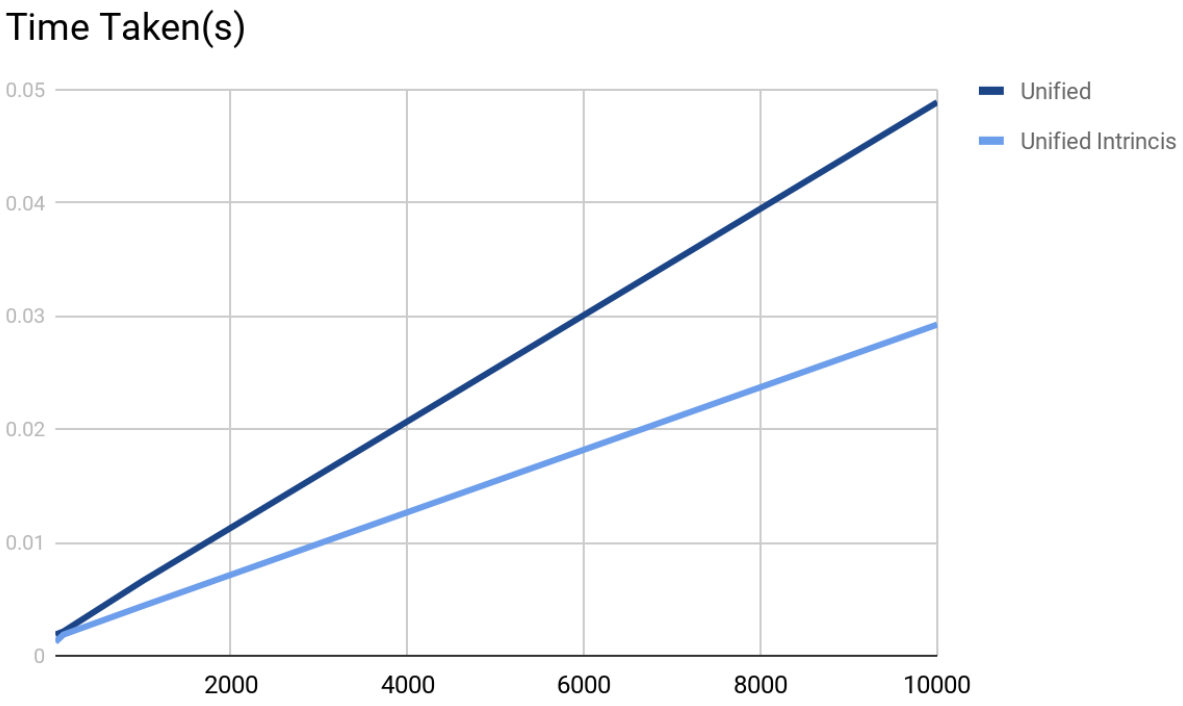
Next, INTRINICS

Time Taken(s)



Time Taken(s)





In conclusion, the application of Intrinsic into the equation truly benefits this situation as shown comparing all three memory type. However Unified memory with Intrinsic benefits the most out the most with an improvement of roughly 1.6 speed up to the Unified memory without Intrinsic. Do take note that when using intrinsic, there might be precision differences.

Update:

After trying out half precision mode for the following fractal I have these results:

Mode (100 Iterations)	Time(s)	Speed Up
CPU	0.01595	1x
half (Block Size - 16)	0.00363	4.4x
half (Block Size - 32)	0.00316	5.0x
Instrinsic (Block Size - 16)	0.00193	8.3x
Instrinsic (Block Size - 32)	0.00208	7.7x

**results were produced by Intel i7-7700HQ for CPU and Nvidia GTX1070 for GPU, CPU results is just to act as a base line for comparison*

Our initial guess was that half precision was supposed to be faster, but the results showed otherwise. The approach was to change the iteration loop to in the burning ship fractal. With these results from 100 iterations, I decided to go ahead and try 10000 iterations.

Mode (10 000 Iterations) [Block size 16]	Time(s)	Speed Up
CPU	0.30752	1x
half	0.07007	4.4x
Instrinsic	0.01916	16.0x

**results were produced by Intel i7-7700HQ for CPU and Nvidia GTX1070 for GPU, CPU results is just to act as a base line for comparison*

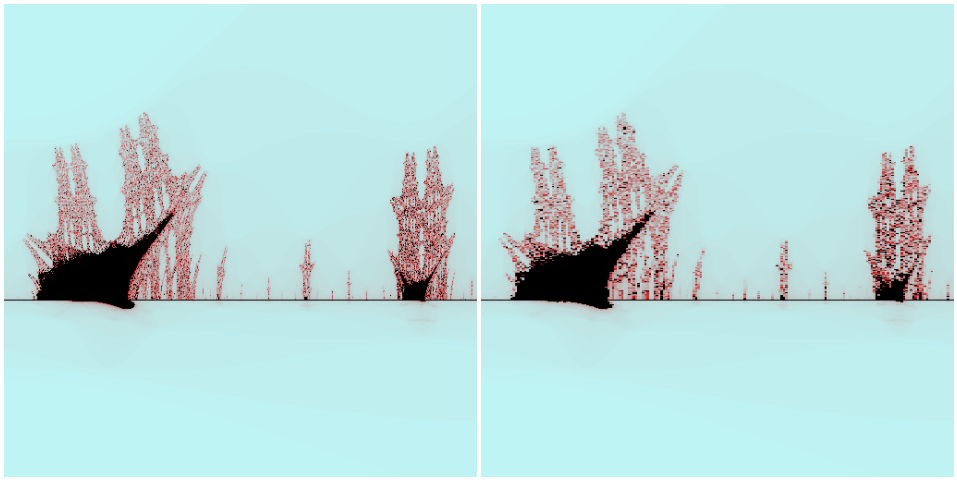
Even with higher iterations, the half precision mode was slower, but it seems to be steadily increasing with the CPU.

Iterations	CPU(s)	GPU(s)	Speed Up
100	0.01595	0.00363	4.4
1000	0.04316	0.01013	4.3
10000	0.30752	0.07007	4.4
100000	2.89102	0.66084	4.4

**results were produced by Intel i7-7700HQ for CPU and Nvidia GTX1070 for GPU, CPU results is just to act as a base line for comparison*

With these results, I have concluded that half precision is not much faster than the CPU and it direct proportionately faster by the CPU with block size of 16.

The next sad news is that half precision provided a way off image of the Burning Ship fractal which is hence inaccurate.



CPU (on the left) and GPU (on the right)

In conclusion, half precision may not be an improvement in our project due to both speed and accuracy loss in our application into fractals. In theory, half precision should yield a higher speed improvement but not in our application itself.

FRACTAL TREE

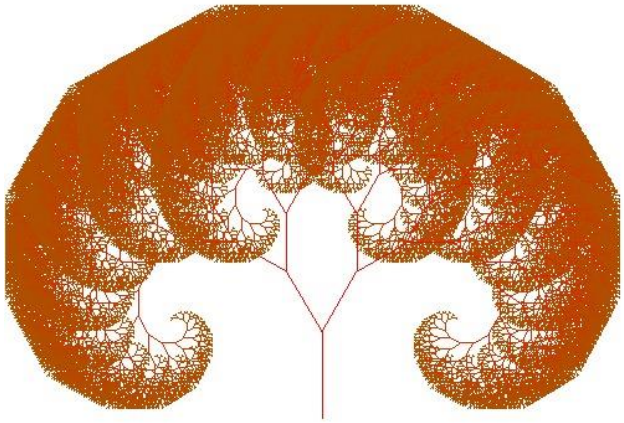
BACKGROUND

The Fractal Tree is a recursive binary algorithm that first draw a line and at the end of the line, draw two more line of +- degree and a percentage length of the previous line.

The input for this tree would be:

- The start and end point,
- The percentage of length lost for each recursion
- The terminating length

Each recursive line depends on the previous line vector and end position which makes it hard for parallelism.



APPROACH

1. Wrote a recursive GPU function using a single thread
    a. Problem Faced:
        i. Default Stack Size may not be enough to hold all recursion in a single thread. Hence, there is a need to increase the limit of stack. However, unable to calculate the number of stack needed at run-time.
        ii. A single thread means zero parallelism which indicates that is almost the exact same as CPU but say slower.
2. Write an iteration version of the recursive GPU function.

I build a struct of data to be use for the iteration

```
/// data structure for the iterative
struct FracTreeGPU
{
    float locX;
    float locY;
    float vecX;
    float vecY;


};
```

Next, I calculate the total number of depth the recursion/iteration would take.

Allocate enough memory to use for each iteration

```
uint count = 0;
float dist = eY - sY;
while (dist >= lim)
{
    count++;
    dist *= per;

}

uint maxNode = 0x01 << count;
count++;
checkCudaErrors(cudaMalloc((void **)&ptr1, PIXELDIM3 * sizeof(uchar)));
cudaMemset(ptr1, 0xFF, PIXELDIM3 * sizeof(uchar));
checkCudaErrors(cudaMalloc((void **)&fptr1, maxNode * sizeof(FracTreeGPU)));
checkCudaErrors(cudaMalloc((void **)&fptr2, maxNode * sizeof(FracTreeGPU)));

setUpFirstVariable << <1, 1 >> > (fptr1);

for (uint itr = 0; itr < count; itr++)
{
    uint i = 0x01 << itr;

    FractalTreeGPUiterative << <ceil(((float)i) / fTsingleBlock), fTsingleBlock >> >(ptr1, i, fptr1, fptr2, count - 1, itr);

    float * tmp = fptr1;
    fptr1 = fptr2;
    fptr2 = tmp;
}
```

Ptr1 to store the data of the board

fptr1 and fptr2 is used to store the data of the struct

Problem Faced: Iteration Version 1

Uses too much memory, however fast due to swapping of pointers and not memcpy

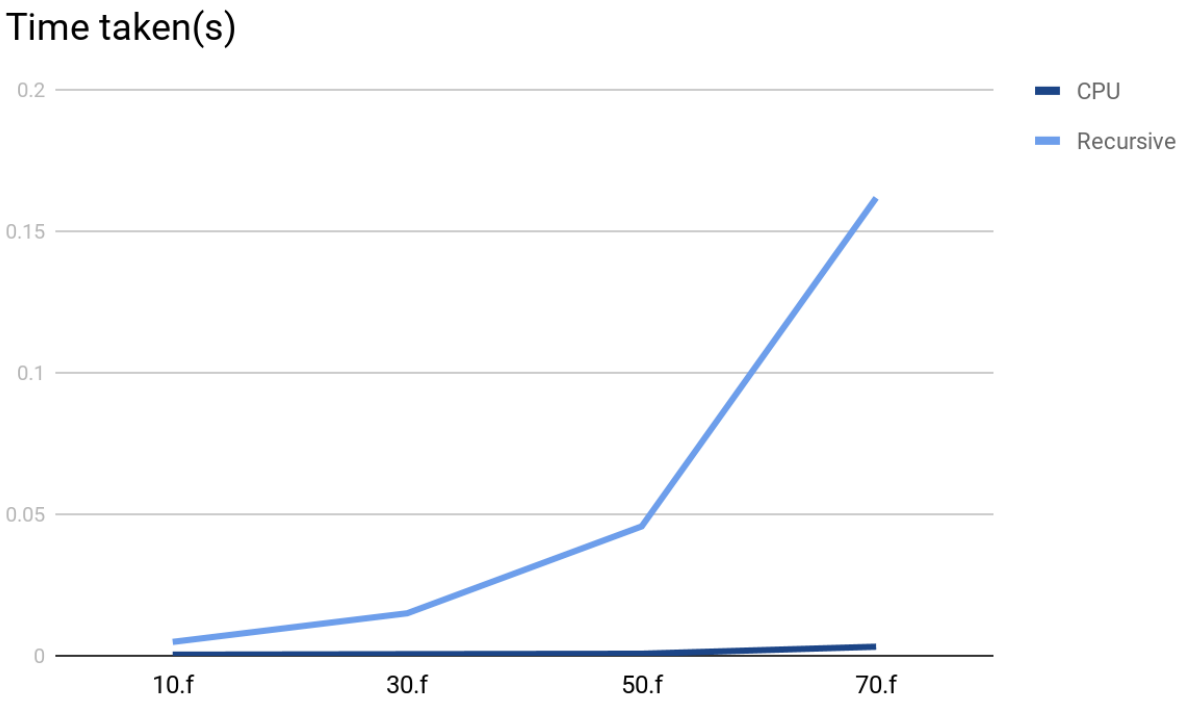
In-order to use save memory,

I use shared memory to act like fptr2, which succeed in removing fptr2

Lastly, I made changes based on Iteration Version 1 with the knowledge gained while writing the shared memory version which efficiently remove fptr2 and shared memory while increasing the speed.

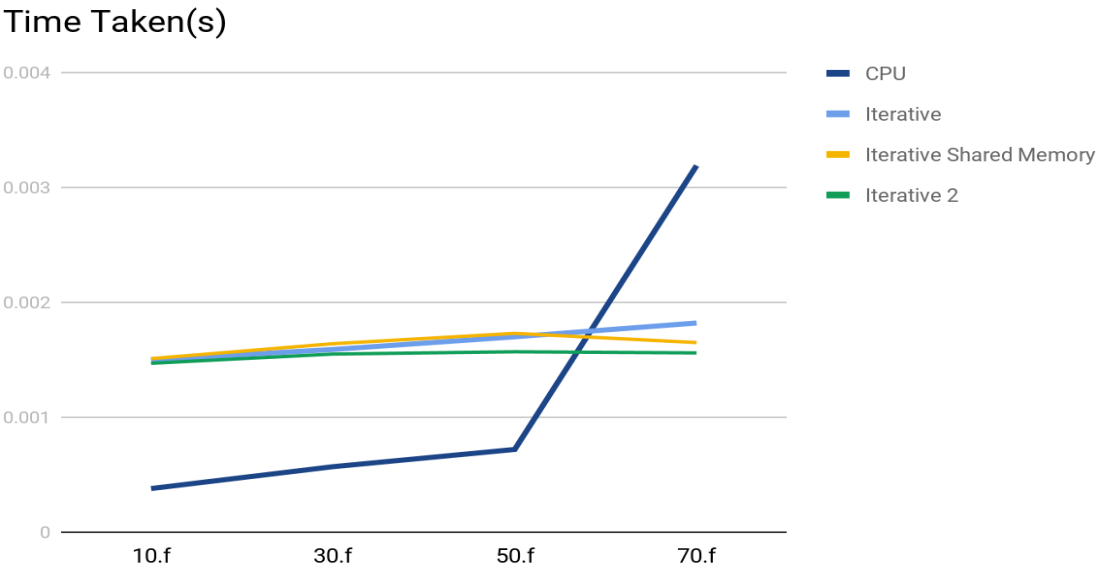
RESULTS

Graph based on initial length of first line and the time taken



The CPU is roughly 50 times faster than the Recursive function of the GPU.

Next is the comparison of the CPU with the iteration GPU version

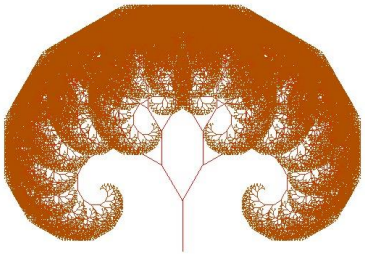


At the low level of iteration, the speed needed for the GPU iterative is roughly consistent unlike the CPU which increase greatly after initial length of 50.f

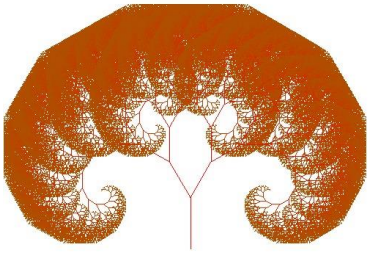
Problem:

Recursive is depth first

Iteration is breadth first



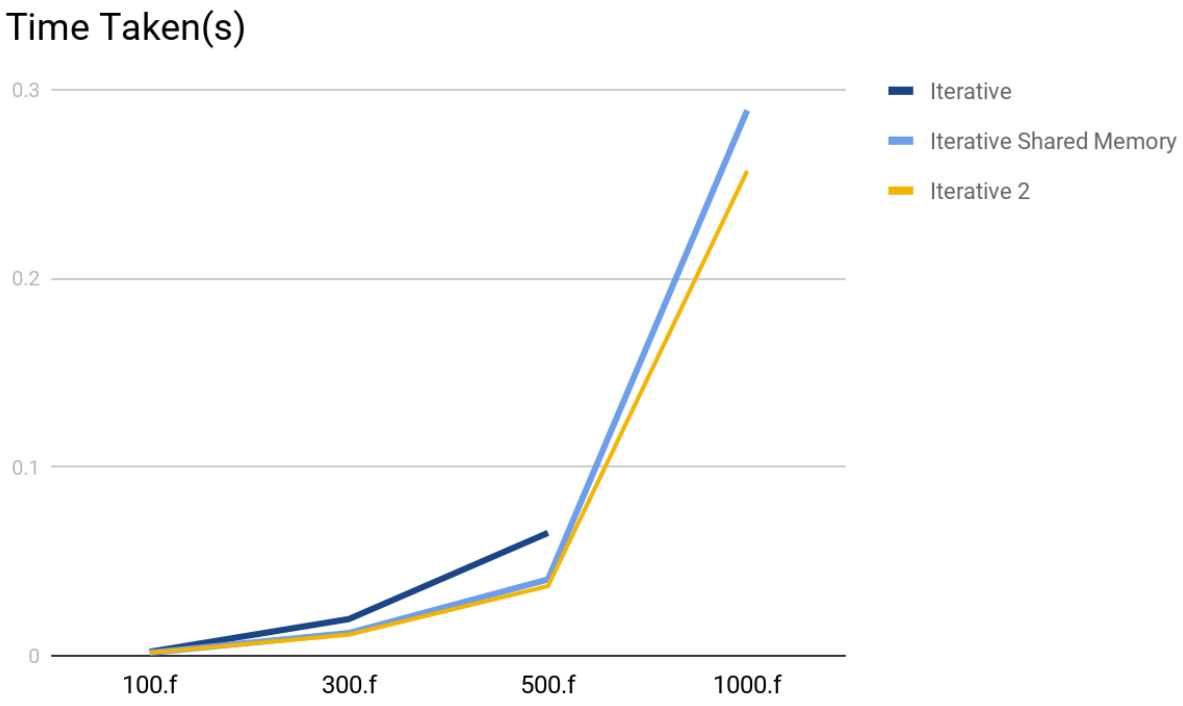
GPU



CPU

If you notice, the CPU version have red lines on top of the branches due to the fact that recursion is depth first so some newer branches are render earlier than the older branches. However for the GPU version, there are no red lines due to the fact that all branches of the same level are rendered before proceeding to the next level of branches.

Note: at this point, the system that we provide to test the function is no longer capable of drawing the tree due to out of range of the image size.



Note that after Initial length of 500.f, there isn't enough memory allocation enough for iterative version 1.

However, Iterative 1 as compare with Iterative 2 and share memory version, we notice that the slower speed differences due to the need to access another pageable memory.

Next, Iterative 2 have lesser read as compare with the shared memory version which is showcase in the graph as well by the speed.

Hence, the optimization for Fractal Tree is possible as an iteration and not as a recursive function (need to increase to an unknown stack limit & no parallelism).

REFERENCES

Newton Fractal: https://en.wikipedia.org/wiki/Newton_fractal

Ikeda Map: https://en.wikipedia.org/wiki/Ikeda_map

Brownian Tree: https://rosettacode.org/wiki/Brownian_tree

Sierpinski Carpet: https://rosettacode.org/wiki/Sierpinski_carpet

Sierpinski Triangle: https://rosettacode.org/wiki/Sierpinski_triangle

Mandelbrot Set: https://rosettacode.org/wiki/Mandelbrot_set

Burning Ship: https://en.wikipedia.org/wiki/Burning_Ship_fractal

Fractal Tree: https://en.wikipedia.org/wiki/L-system

LIST OF WORK BY EACH STUDENT

• Henon Map [Dropped] • Newton Fractals • Ikeda Map • Burning Ship Fractals (Half Precision)	Alvin Tan
• Brownian Tree • Sierpinski Carpet	Kenneth Toh
• Sierpinski Triangle • Mandelbrot Set	Ong Yong Kiat
• Burning Ship Fractals • Fractal Tree	Tham Cheng Jiang