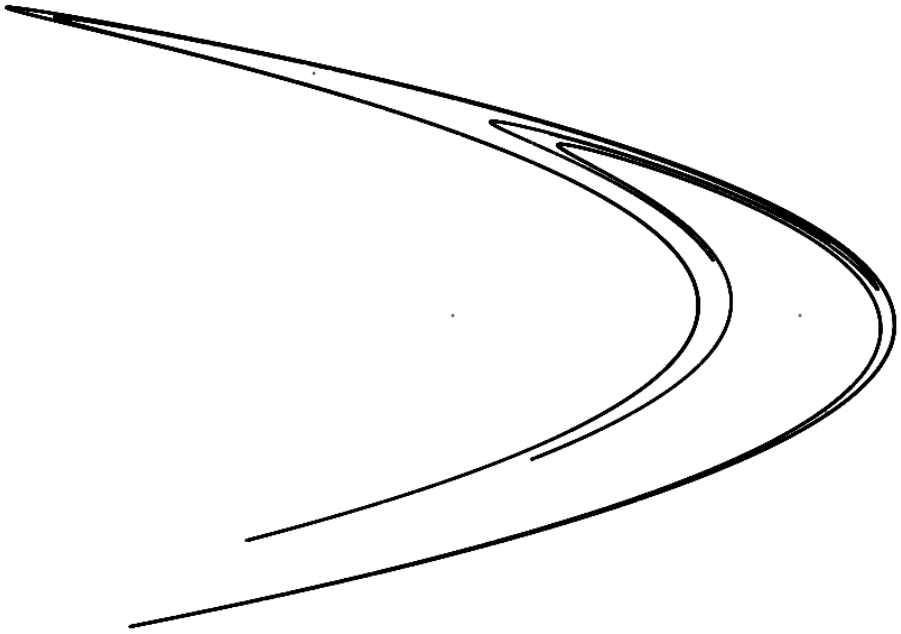


Henon Map

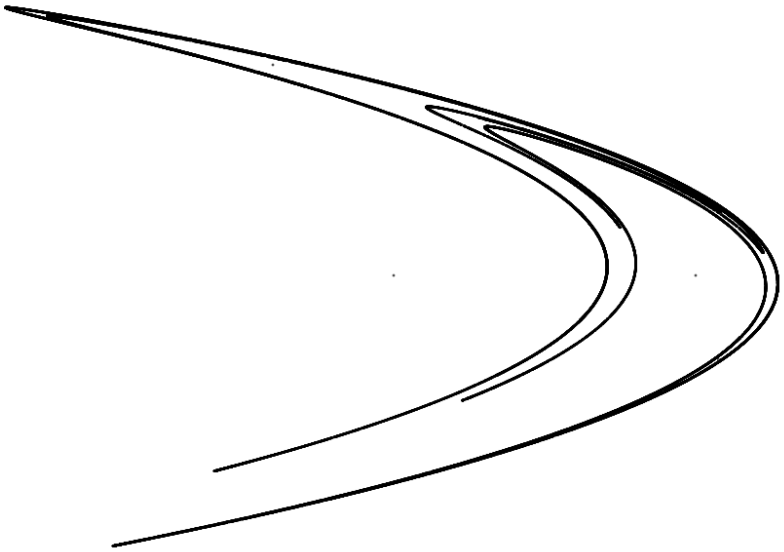
A THEORETICAL ANALYSIS

Samarth Sudarshan Inamdar (CED18I045) | High Performance Computing
August 2021

Department of Computer Science and Engineering,
Indian Institute of Information Technology Design and Manufacturing Kancheepuram, Chennai

Table of Contents

Introduction



The Henon map is an iterated discrete-time dynamical system that exhibits chaotic behavior in two-dimension. It is sometimes called Hénon-Pomeau attractor/map is a discrete-time dynamical system. It is one of the most studied examples of dynamical systems that exhibit chaotic behavior.

What is Henon Map?

The Henon map presents a simple two-dimensional invertible iterated map with quadratic nonlinearity and chaotic solutions called strange attractor. Strange attractors are a link between the chaos and the fractals.

The Henon map takes a point (x_n, y_n) in the plane and maps it to a new point. The below equation helps us to find the next point based on the previous point.

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

Code (Henon Map)

SERIAL CODE

```
// Compilation
// g++ main.cpp -o main -lGL -lGLU -lglut -lm -lGLEW
#include <stdio.h>
#include <GL/glut.h>
#include <GL/glu.h>
#include <GL/gl.h>
#include <GL/freeglut.h>

typedef struct point
{
    double x;
    double y;
} Point;

int start = 0, end = 100000;
double a = 1.4;
double b = 0.3;
static GLfloat theta[] = {0.0, 0.0, 0.0};
GLint axis = 1;
Point initial = {1.0, 1.0};

Point eq(Point prev, Point pprev)
{
    Point p;
    p.y = b * prev.x;
    p.x = 1 - a * prev.x * prev.x + b * pprev.x;
    return p;
}

void henon_gen()
{
    int n = end - start;
    Point *coord = (Point *)malloc(n * sizeof(Point));
    coord[0] = initial;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    glPointSize(1.2);
    coord[1].x = 1 - a * coord[0].x * coord[0].x + coord[0].y;
```

```
    coord[1].y = b * coord[0].x;
    for (int i = 1; i < n; i++)
    {
        coord[i] = eq(coord[i - 1], coord[i - 2]);
        glBegin(GL_POINTS);
        glColor3f(1, 1, 1);
        glVertex2f(coord[i].x, coord[i].y);
        glEnd();
        glFlush();
        glutSwapBuffers();
    }
}

void spinCube()
{
    if (theta[axis] > 360.0)
        theta[axis] -= 360.0;
    else if (theta[axis] < 0)
        theta[axis] += 360.0;
    glutPostRedisplay();
}

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1.5, 1.5, -1.5, 1.5, -1.5, 1.5);
    glMatrixMode(GL_MODELVIEW);
}

int main()
{
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Henon Map");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(henon_gen);
    glutIdleFunc(spinCube);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
    glutLeaveMainLoop();
}
```

PARALLEL CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <GL/glut.h>
#include <GL/glu.h>
#include <GL/gl.h>
#include <GL/freeglut.h>
#include <stddef.h>

#define N 4096
#define NUM_BLOCKS 4
#define NUM_THREADS 1024
#define a 1.4
#define b 0.3

typedef struct point
{
    double x;
    double y;
} Point;

static GLfloat theta[] = {0.0, 0.0, 0.0};
GLint axis = 1;
Point initial = {1.0, 1.0};

__global__ void eq(Point *prev, Point *pprev)
{
    __shared__ float temp[2];
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    switch (index)
    {
    case 0:
        temp[0] = a * prev->x * prev->x;
    case 1:
        temp[1] = b * pprev->x;
    }
```

```
    __syncthreads();
    if (index == 0)
    {
        memcpy(pprev, prev, sizeof(Point));
        prev->y = b * pprev->x;
        prev->x = 1 - temp[0] + temp[1];
    }
}

void plot(Point coord)
{
    glBegin(GL_POINTS);
    glColor3f(1, 1, 1);
    glVertex2f(coord.x, coord.y);
    glEnd();
    glFlush();
    glutSwapBuffers();
}

void henon_gen()
{
    Point *prev, *pprev, *d_prev, *d_pprev;
    prev = (Point *)malloc(sizeof(Point));
    pprev = (Point *)malloc(sizeof(Point));
    cudaMalloc((void **)&d_prev, sizeof(Point));
    cudaMalloc((void **)&d_pprev, sizeof(Point));
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    *pprev = initial;
    prev->x = 1 - a * pprev->x * pprev->x + pprev->y;
    prev->y = b * pprev->x;

    for (int i = 1; i < N; i++)
    {
        cudaMemcpy(d_prev, prev, sizeof(Point), cudaMemcpyHostToDevice);
        cudaMemcpy(d_pprev, pprev, sizeof(Point), cudaMemcpyHostToDevice);
        eq<<<NUM_BLOCKS, NUM_THREADS>>>(d_prev, d_pprev);
        cudaError_t err = cudaGetLastError();
        if (err != cudaSuccess)
```

```
        {
            printf("Error: %s\n", cudaGetErrorString(err));
            break;
        }
        cudaMemcpy(prev, d_prev, sizeof(Point), cudaMemcpyDeviceToHost);
        cudaMemcpy(pprev, d_pprev, sizeof(Point), cudaMemcpyDeviceToHost);
        plot(*prev);
    }
    glutLeaveMainLoop();
}

void spinCube()
{
    if (theta[axis] > 360.0)
        theta[axis] -= 360.0;
    else if (theta[axis] < 0)
        theta[axis] += 360.0;
    glutPostRedisplay();
}

void myReshape(int w, int h)
{
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    glPointSize(1.2);
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1.5, 1.5, -1.5, 1.5, -1.5, 1.5);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Henon Map");
```

```
    glutReshapeFunc(myReshape);
    glutDisplayFunc(henon_gen);
    glutIdleFunc(spinCube);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```

Critical Section

```
__global__ void eq(Point *prev, Point *pprev)
{
    __shared__ float temp[2];
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    switch (index)
    {
    case 0:
        temp[0] = a * prev->x * prev->x;
    case 1:
        temp[1] = b * pprev->x;
    }

    __syncthreads();
    if (index == 0)
    {
        memcpy(pprev, prev, sizeof(Point));
        prev->y = b * pprev->x;
        prev->x = 1 - temp[0] + temp[1];
    }
}
```

Above code snippet is the critical section of the code i.e., major computation is being done here as this part runs most frequently.

This part calculates the next point using the previous point using the henon equation stated above. Since there is dependency of the current point with

the previous point, this gives us a very small room for parallelization and performance improvement.

We have two cases in the switch case which corresponds to the index of the thread as per the expression and here, number of cases are 2 and hence we need atleast 2 threads when executing the program.

After this part is done (i.e., the intermediate calculations are done), we use __syncthreads() before the master starts working as we need all the other threads to be at the point where all the intermediate calculations are done. Now the master will have all the values and can hence compute the final points (x, y) that are to be plotted on the screen.

Observations

COMPILATION AND EXECUTION

Enabling CUDA environment use while compiling using nvcc.

```
nvcc -o cuda cuda.cu -lGL -lGLU -lglut -lGLEW -lm
```

For execution, use

```
./cuda
```

For execution and recording the time to execute,

We use the CLI time command,

```
time ./cuda
```

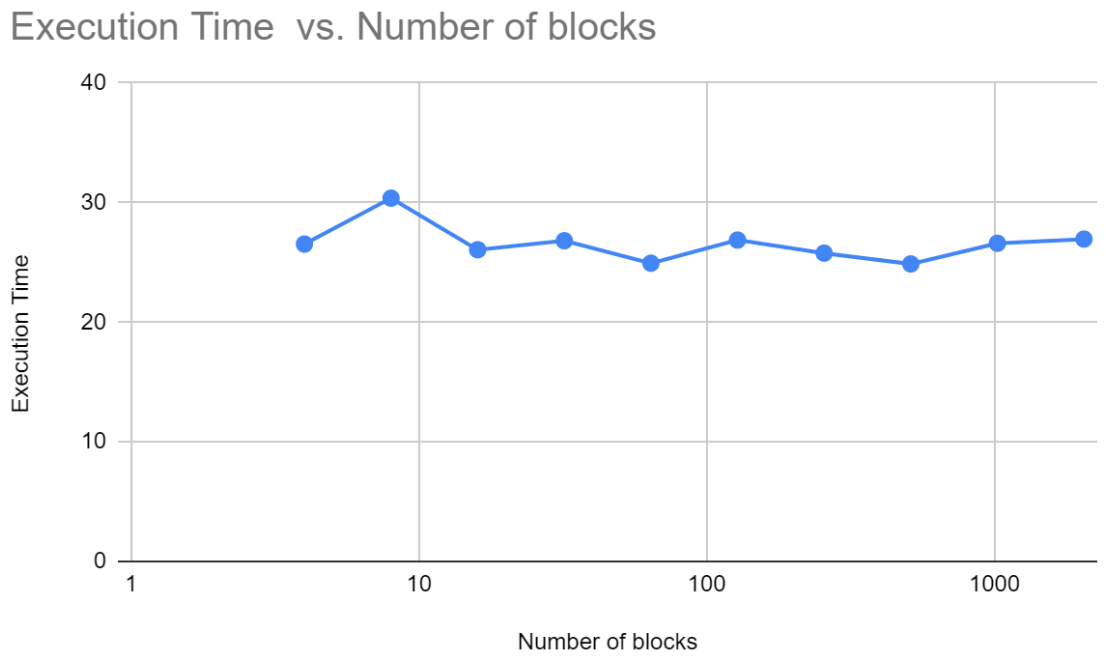
This gives the time taken to compute all the points .

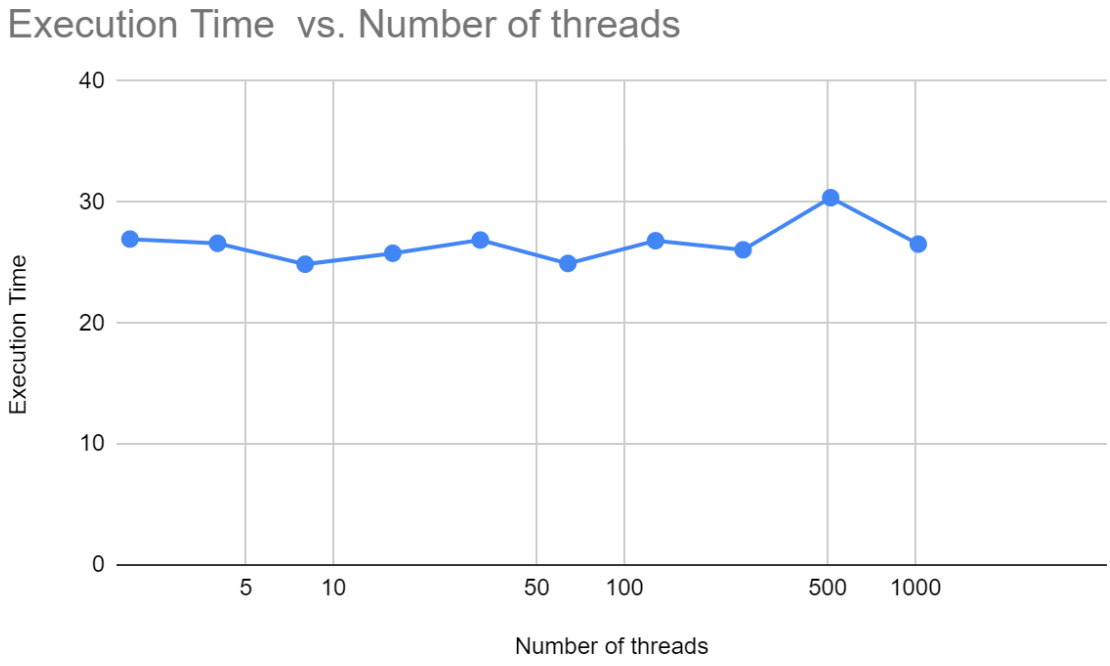
OBSERVATION TABLE

Number of blocks	Number of threads	Execution Time
2048	2	26.947
1024	4	26.598
512	8	24.869
256	16	25.773
128	32	26.866
64	64	24.933
32	128	26.811
16	256	26.059
8	512	30.37
4	1024	26.531
2	2048	NA
1	4096	NA

NA -> The number of threads is limited to 1024 according to the Nvidia CUDA Toolkit Documentation and hence we get an error if we try to give threads above 1024 (i.e., even giving 1025 throws error) [link].

GRAPH


Execution Time vs. Number of blocks
Execution Time
40
30
20
10
0
1
10
100
1000
Number of blocks


Execution Time vs. Number of threads
Execution Time
40
30
20
10
0
5
10
50
100
500
1000
Number of threads

Conclusion

The performance of the programs is almost the same for different values of threads and blocks, there is a bit of increase or decrease observed in some cases which maybe due to the resources available for the program.

At least 2 threads are required by the program as there are 2 switch cases and hence the kernel requires at least 2 threads to work on.

The programs also throw an error "Error: invalid configuration argument" if we increase the number of threads above 1024 as it is mentioned in the critical section of the report above and the documentation of Nvidia CUDA Toolkit.

References

Wikipedia | Henon Map

Wikipedia |Dynamical System

Wikipedia | Attractor

Dynamical Properties of the Hénon Mapping

CUDA | CUDA Toolkit Documentation