CUDA EXECUTION & ANALYSIS

LORENZ ATTRACTOR

High Performance Computing Project Report

By,

SAI KAUSHIK SUDHAKARAN

CED18I044

Department of Computer Science and Engineering,

Indian Institute of Information Technology Design and Manufacturing

Kancheepuram, Chennai

November 2021

ACKNOWLEDGEMENT

I would like to express my gratitude toward staff of Department of Computer Science and Engineering of Indian Institute of Information Technology Design and Manufacturing Kancheepuram for providing me a great opportunity to complete a project on Parallelizing Lorenz Attractor.

My sincere thanks go to **Dr. Noor Mahammad SK, Assistant Professor, Department of Computer Science and Engineering.** Your valuable guidance and suggestions helped me in various phases of the completion of this project. I will always be thankful to you in this regard.

I am ensuring that this project was done by me and not copied from anywhere.

ABSTRACT

This project is designed to implement parallel programs for the simulation of objects following **Lorenz attractor** satisfying the Lorenz system.

Lorenz System is a system of differential equations, noted to have chaotic solutions for given parameters and the initial conditions. Lorenz attractor is the set of chaotic solution to the above system.

Using multi-dimensional arrays and multiple threads to increase the speed-up of the program. Generating high performance programs for the same of multi-core processor (OpenMP), cluster computers (MPI), and general-purpose graphic processing unit (Cuda C/C++).

Drawback of serial programs is the inability to run multiple objects to visually see the chaos with slight variations in the initial conditions.

TABLE OF CONTENT

INTRODUCTION

The Lorenz System is a system of differential equations, noted to have chaotic solutions for given parameters and the initial conditions, first studied by Edward Lorenz in 1962 [1].

Figure 1: *Lorenz Attractor Animation*

Lorenz attractor is the set of chaotic solutions of the Lorenz System. It is a part of an area of study, Chaos Theory.

The shape of the Lorenz attractor, when plotted graphically, represents that of a butterfly. The butterfly effect is a real-world implication of the Lorenz attractor.

This project is designed to implement parallel programs for the simulation of objects following Lorenz attractor satisfying the Lorenz system.

LITERATURE SURVEY

CHAOS THEORY

Chaos theory is a branch of mathematics focusing on the study of chaos—dynamical systems whose apparently random states of disorder and irregularities are actually governed by underlying patterns and deterministic laws that are highly sensitive to initial conditions [2].

A commonly used definition, to classify a dynamical system as chaotic, it must satisfy three properties:

- It must be sensitive to initial conditions.
- It must be topologically transitive.
- It must have dense periodic orbits.

BUTTERFLY EFFECT

In chaos theory, the butterfly effect is the sensitive dependence on initial conditions in which a small change in one state of a deterministic nonlinear system can result in large differences in a later state. [3]

It is metaphorically derived from the example of a tornado being influenced by minor perturbations such as a distant butterfly flapping its wings several weeks earlier.

ATMOSPHERIC CONVECTION

Atmospheric convection is the result of a temperature difference layer in the atmosphere. Different rate of temperature change with altitude within the air masses lead to instability. This instability with moist air masses causes thunderstorms, which are responsible for severe weather in the world. [4]

HISTORY

Lorenz had been studying the simplified models describing the motion of the atmosphere, in terms of ordinary differential equations depending on few variables. In his 1962 article, he analysed differential equations, for atmospheric convection, in a space of 12 dimensions, in which he numerically detects a sensitive dependence to initial conditions. [5]

In 1963, with the help of Ellen Fetter, Lorenz simplified the above mathematical model to a system of differential equations, now known as the Lorenz equations.

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

Here, x is proportional to the rate on convection

y is proportional to the horizontal temperature variation

z is proportional to the vertical temperature variation

σ is proportional to the Prandtl number, ratio of momentum diffusivity to thermal diffusivity

ρ is proportional to the Rayleigh number, describes the behaviour of fluids when the mass density is non-uniform

β is proportional to certain physical dimensions

CODE (LORENZ ATTRACTOR)

SERIAL CODE

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

typedef struct point
{
    double x;
    double y;
    double z;
    float t;
} Point;

double sigma = 10.0;
double rho = 28.0;
double beta = 8.0 / 3.0;

Point initial = {1.0, 1.0, 1.0, 0.0};
double dt = 0.01;

unsigned long int start = 0;
unsigned long int end = 1000000000000;

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

unsigned long int op = 0;

Point differential(Point curr)
{
    Point diff;
    // 4 artih ops, 5 reads, 1 write
    diff.x = (sigma * (curr.y - curr.x)) * dt + curr.x;
    op += 4;
    // 5 arith ops, 6 reads, 1 write
    diff.y = (curr.x * (rho - curr.z) - curr.y) * dt + curr.y;
    op += 5;
    // 5 artih ops, 6 reads, 1 write
    diff.z = (curr.x * curr.y - beta * curr.z) * dt + curr.z;
    op += 5;
    // 1 arith op, 2 reads, 1 write
    diff.t = curr.t + dt;
    op += 1;
    return diff;
}

void lorenzGenerator()
{
    Point coord = initial;
```

```c
    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(1.0);
    FILE *fp = fopen("new.txt", "w");
    for (unsigned long int i = start; i < end; i++)
    {
        coord = differential(coord);
        fprintf(fp, "%f, %f, %f %f\n", coord.t, coord.x, coord.y, coord.z);
        glBegin(GL_POINTS);
        glColor3f(1, 1, 1);
        glVertex3f(coord.x, coord.y, coord.z);
        glEnd();
        glFlush();
        glutSwapBuffers();
    }
    fclose(fp);
    printf("Ops = %ld\n", op);
    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)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-50.0, 50.0, -50.0, 50.0, -50.0, 50.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(1280, 720);

    glutCreateWindow("Lorenz");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(lorenzGenerator);
    glutIdleFunc(spinCube);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```

PARALLEL CODE

```
/*
nvcc -o lorenz_cuda lorenz_cuda.cu -lGL -lGLU -lglut -lGLEW -lm
time ./lorenz_cuda
*/

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

#define N 65536
#define NUM_BLOCKS 32
#define NUM_THREADS 2048

typedef struct point
{
    float x;
    float y;
    float z;
    float t;
} Point;

Point initial = {1.0, 1.0, 1.0, 0.0};

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

__global__ void diff(Point *a)
{
    __shared__ float temp[4];
    __shared__ float sigma;
    __shared__ float rho;
    __shared__ float beta;
    __shared__ float dt;
    sigma = 10.0;
    rho = 28.0;
    beta = 8.0 / 3.0;
    dt = 0.01;
    int index = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    switch (index)
    {
    case 0:
        temp[0] = sigma * (a->y - a->x);
        break;
    case 1:
        temp[1] = a->x * (rho - a->z) - a->y;
        break;
    case 2:
        temp[2] = a->x * a->y;
        break;
    case 3:
        temp[3] = beta * a->z;
        break;
    }

    __syncthreads();
    if (index == 0)
    {
        atomicAdd(&a->x, temp[0] * dt);
        atomicAdd(&a->y, temp[1] * dt);
        atomicAdd(&a->z, (temp[2] - temp[3]) * dt);
        atomicAdd(&a->t, dt);
    }
}

void draw(Point coord)
{
    glBegin(GL_POINTS);
    glColor3f(1, 1, 1);
    glVertex3f(coord.x, coord.y, coord.z);
    glEnd();
    glFlush();
    glutSwapBuffers();
}

void lorenzGenerator()
{
    Point *coord, *d_coord;
    coord = (Point *)malloc(sizeof(Point));
    cudaMalloc((void **)&d_coord, sizeof(Point));
    *coord = initial;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glPointSize(1.0);
    for (unsigned long int i = 0; i < N; i++)
    {
        cudaMemcpy(d_coord, coord, sizeof(Point), cudaMemcpyHostToDevice);
```

```
        diff<<<NUM_BLOCKS, NUM_THREADS>>>(d_coord);

        cudaError_t err = cudaGetLastError();
        if (err != cudaSuccess)
            printf("Error: %s\n", cudaGetErrorString(err));

        cudaMemcpy(coord, d_coord, sizeof(Point), cudaMemcpyDeviceToHost);
        draw(*coord);
    }
    glutLeaveMainLoop();
    free(coord);
    cudaFree(d_coord);
}

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);
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-50.0, 50.0, -50.0, 50.0, -50.0, 50.0);
    glMatrixMode(GL_MODELVIEW);
}

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

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(1280, 720);
    glutCreateWindow("Lorenz");

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

CRITICAL SECTION OF THE CODE

In the “diff” function,

```
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    switch (index)
    {
    case 0:
        temp[0] = sigma * (a->y - a->x);
        break;
    case 1:
        temp[1] = a->x * (rho - a->z) - a->y;
        break;
    case 2:
        temp[2] = a->x * a->y;
        break;
    case 3:
        temp[3] = beta * a->z;
        break;
    }

    __syncthreads();

    if (index == 0)
    {
        atomicAdd(&a->x, temp[0] * dt);
        atomicAdd(&a->y, temp[1] * dt);
        atomicAdd(&a->z, (temp[2] - temp[3]) * dt);
        atomicAdd(&a->t, dt);
    }
```

The above snippet is the critical section of the program, i.e., the most frequently run section in the program. This section calculates the new point with the help of the Lorenz equations mentioned above [link].

Due to the dependency of the current point with the previous point, gives us a small room for parallelization.

Here, each processor has a copy of the current point which will be broadcasted in the start of the function execution.

We use a switch case with the index of the thread as the expression. As we have 4 critical expressions to be solved, we use the first four threads.

We add a __syncthreads function before the master works because we need all the threads to reach the point. This way the master has all the values to compute.

Here, we use 4 workers to complete the execution. Theoretically, the performance should remain the same for thread count of 4 or above.

OBSERVATION

COMPILATION AND EXECUTION

Compiling the code with mpi compiler with all essential flags for OpenGL.

```
mpicc lorenz_mpi.c -o lorenz_mpi -lm -lGL -lGLU -lglut -lGLEW
```

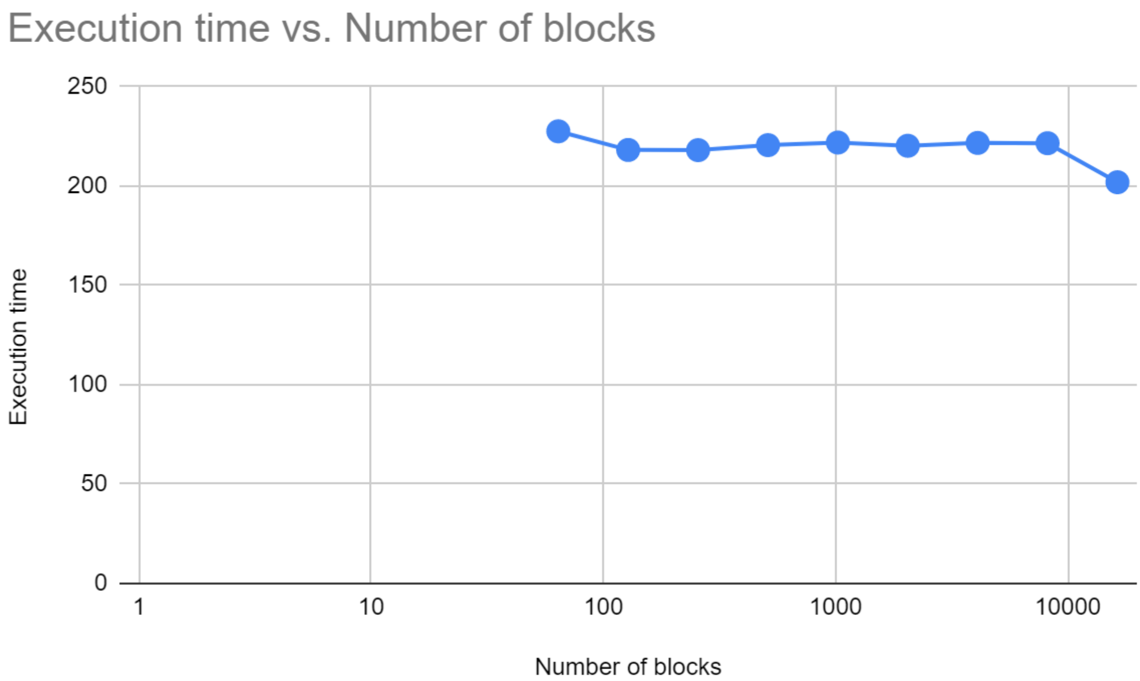
For execution, use

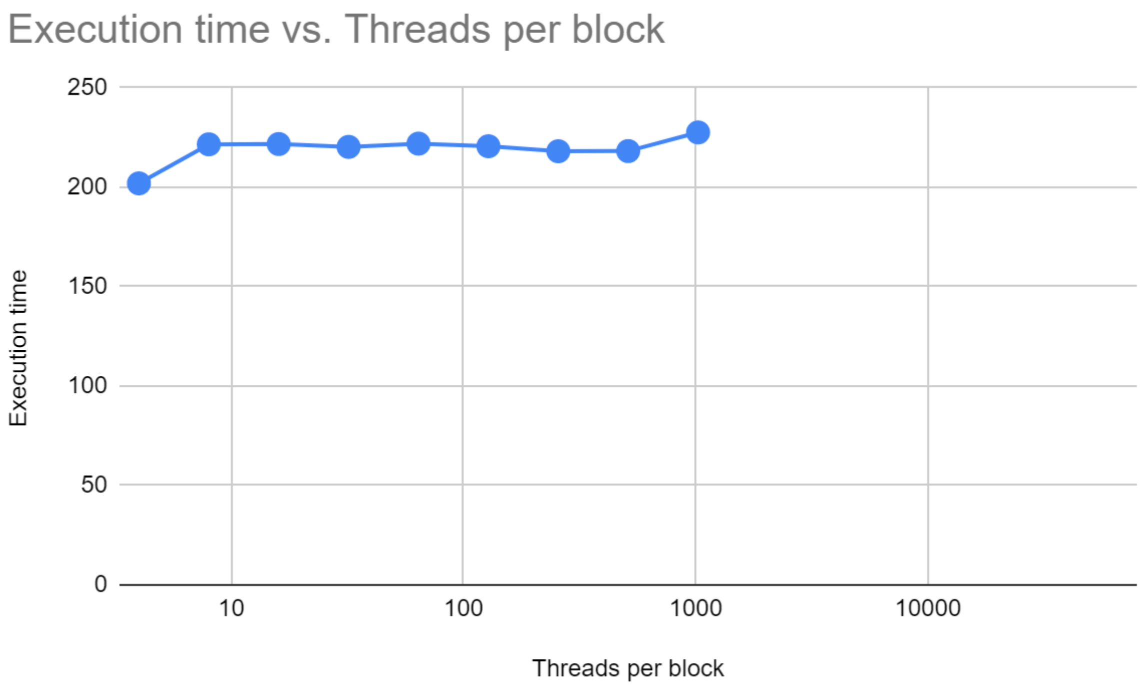
```
mpirun -n NUMBER_OF_PROCESSORS -f machinefile ./lorenz_mpi
```

TABLE

Lorenz Attractor (n = 65536)		
Number of blocks	Threads per block	Execution time
16384	4	201.925
8192	8	221.563
4096	16	221.713
2048	32	220.275
1024	64	221.914
512	128	220.6
256	256	218.081
128	512	218.166
64	1024	227.548
32	2048	*
16	4096	*
8	8192	*
4	16384	*
2	32768	*
1	65536	*
* The number of threads cannot cross 1024, as given by Nvidia CUDA Toolkit Documentation. [6]		

GRAPHS


Execution time vs. Number of blocks
Execution time
250
200
150
100
50
0
1
10
100
1000
10000
Number of blocks


Execution time vs. Threads per block
Execution time
250
200
150
100
50
0
10
100
1000
10000
Threads per block

CONCLUSION

The performance of the program remains similar with varying values for threads and blocks.

The program at least needs 4 threads to work, due to hard coded kernel, each computing an essential part of the point generation.

The program gets an "Error: invalid configuration argument" when the number of threads crosses 1024. It was verified in the documentation [6] that the maximum x-dimension of a block can be 1024.

REFERENCES

[1] Wikipedia, "Lorenz system - Wikipedia," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Lorenz_system.

[2] Wikipedia, "Chaos theory - Wikipedia," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Chaos_theory.

[3] Wikipedia, "Butterfly effect - Wikipedia," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Butterfly_effect.

[4] Wikipedia, "Atmospheric convection - Wikipedia," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Atmospheric_convection.

[5] E. Lorenz, "The statistical prediction of solutions of dynamic equations.," Meteorological Society of Japan, Tokyo, 1962.

[6] Nvidia, "Nvidia Developer Zone CUDA Toolkit Documentation," [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications.

[7] J. Burkardt, "LORENZ_ODE - The Lorenz System," Florida State University, [Online]. Available: https://people.sc.fsu.edu/~jburkardt/c_src/lorenz_ode/lorenz_ode.html.