HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

PROJECT REPORT

Diffusion Limited Aggregation using CUDA

TRUONG GIA BACH **TRAN DUONG CHINH**
20210087 20210122

NGUYEN VIET MINH **PHAN DUC HUNG**
20214917 20214903

Course: Distributed and Parallel Computing

Supervisor: PhD. Vu Van Thieu _____

Signature

Department: Computer Science

School: Information and Communication Technology

HANOI, 12/2023

TABLE OF CONTENTS

1 Introduction

Diffusion Limited Aggregation (DLA) is a physical process that describes the growth of clusters through the random motion of particles. The process was introduced by Witten and Sander in 1981 and has applications in various fields such as physics, biology, and material science. DLA produces fractal structures and is a key model for studying pattern formation in nature.

In DLA, particles are released from a source and move randomly until they come into contact with a growing cluster. Upon contact, the particles stick to the cluster, causing it to grow. This simple rule can lead to the formation of highly complex and beautiful fractal patterns.

The objective of this project is to simulate the DLA process using CUDA to take advantage of the parallel processing capabilities of modern GPUs. The project involves:

- Random initialization of particles.
- Simulation of random movement of particles.
- Detection of when particles touch the cluster and their crystallization.

2 Mathematical Formulation

The mathematical formulation of DLA involves several key components:

2.1 Random Initialization of Particles

Particles are initialized at random positions outside a predefined radius from the cluster. Let $\mathbf{r}_i$ denote the position of the i-th particle:

$$\mathbf{r}_i(0) = (x_i(0), y_i(0)), \quad \text{with} \quad x_i(0)^2 + y_i(0)^2 > R_{\text{init}}^2$$

where R_{init} is the initial radius beyond which particles are released.

2.2 Random Movement of Particles

Particles undergo random walk, which can be described by the following equations:

$$\mathbf{r}_i(t+1) = \mathbf{r}_i(t) + \Delta\mathbf{r}$$

where $\Delta\mathbf{r}$ is a random displacement vector. In a 2D simulation, this can be expressed as:

$$\Delta\mathbf{r} = (\Delta x, \Delta y)$$

where Δx and Δy are random values typically chosen from a uniform distribution, such as $\{-1, 0, 1\}$.

2.3 Crystallization Condition

A particle sticks to the cluster if it moves to a position adjacent to any particle in the cluster. Mathematically, if $\mathbf{r}_c$ denotes the position of a cluster particle and $\mathbf{r}_i(t+1)$ is the new position of the moving particle, the condition for sticking is:

$$\|\mathbf{r}_i(t+1) - \mathbf{r}_c\| \leq d$$

where d is a small distance (typically 1 in lattice-based simulations).

2.4 Cluster Growth

As particles stick to the cluster, it grows in a branched, fractal-like pattern. The growth process continues until a stopping criterion is met, such as reaching a maximum cluster size or time limit.

2.5 Simulation Parameters

DLA simulations often involve parameters such as the number of particles released, the size of the simulation grid, and the criteria for cluster growth termination. These parameters affect the final structure and computational efficiency of the simulation.

3 Implementation

3.1 C Implementation

The following C code simulates Diffusion Limited Aggregation (DLA). It includes functions for initializing particles, moving them randomly, checking if they have touched the cluster, and updating their state accordingly.

3.1.1 Functionalities

All attributes of functions are listed as below:

- **cpuSecond()**: Returns the current time in seconds.
- **random_number(seed)**: Generates a pseudo-random number based on the seed.
- **initialize(points, number)**: Initializes the particle positions and states.
- **move(p)**: Moves a particle randomly in one of four directions.
- **check_occupied(x, y)**: Checks if a position is occupied by a particle in the cluster.

- **occupy(p)**: Marks a particle as part of the cluster.
- **fill(points, number)**: Updates the state of particles based on their proximity to the cluster.
- **move_particles(points, number)**: Moves all particles that are not yet part of the cluster.
- **save_map(file_name, map, width, height)**: Saves the map to a file.

3.1.2 Pseudo Code

Here is the pseudo code for the main functions:

initialize(points, number):

1. For each particle:
2. Set state to 0.
3. Randomly initialize x and y coordinates.
4. Set seed to the index.
5. Store the particle in the points array.

move(p):

1. Update seed with a new random value.
2. Determine movement direction based on seed.
3. Update x and y coordinates accordingly.
4. Ensure coordinates wrap around if they go out of bounds.

check_occupied(x, y):

1. For each neighboring cell:
2. Check if it is occupied.
3. Return 1 if occupied, else 0.

occupy(p):

1. Set particle state to 1.
2. Mark the corresponding map cell as occupied.

fill(points, number):

1. Repeat until no changes:
2. For each particle:
3. If not already occupied and a neighbor is occupied:

4. Mark the particle as occupied.

5. Update the map.

move_particles(points, number):

1. For each particle:

2. If not already occupied, move the particle.

3.1.3 Random Function

Because the random functions from C code and CUDA code will generate differently even though we have set the same seed, so we will implement a pseudo random function. Random function used in the code generates a pseudo-random number based on a given seed, defined as follows:

```
float random_number(int seed){
  int x = seed;
  int m = 65537;
  int a = 75;
  int k = 10;
  for(int i=0; i<k; i++){
    x = (a*x)%m;
  }
  float ans = (float)x/(float)m;
  return ans;
}
```

Listing 1: Random Number Generator

This function uses a linear congruential generator (LCG) method, which is a simple pseudo-random number generator. The parameters used are:

- $m = 65537$: The modulus.
- $a = 75$: The multiplier.
- $k = 10$: The number of iterations.

The function iterates k times to update the seed and then scales the result to a float between 0 and 1. This random number is used to determine the initial positions of the particles and their movements during the simulation.

3.2 CUDA Implementation

The CUDA implementation of the Diffusion Limited Aggregation (DLA) algorithm includes the following functions to parallelize the process of initializing particles, moving them, checking their positions, and updating the map.

3.2.1 Functionalities

The modification in functions are listed as below:

- **init_particles(points, number)**: Initializes the particle positions and states in parallel using CUDA threads.
- **move(p)**: Moves a particle randomly in one of four directions, ensuring it stays within the boundaries. This function is executed on the device.
- **check_occupied(x, y)**: Checks if a position is occupied by a particle in the cluster, executed on the device.
- **occupy(p)**: Marks a particle as part of the cluster, executed on the device.
- **fill(points, number, changed)**: Updates the state of particles based on their proximity to the cluster. If any particle changes state, it updates the 'changed' flag.
- **move_particles(points, number)**: Moves all particles that are not yet part of the cluster, executed in parallel using CUDA threads.
- **save_map(file_name, map, width, height)**: Saves the map to a file on the host.

Key aspects of the implementation include:

- Copying the dimensions of the map and the map itself to the device using 'cudaMemcpyToSymbol'.
- Using CUDA kernels to initialize particle positions ('init_particles'), move particles ('move_particles'), and check and update their states ('fill').
- Synchronizing the device to ensure all threads complete their tasks before moving on to the next step.

The main steps of the CUDA implementation are as follows:

1. Allocate memory for the map and particles on the device.
2. Initialize particles in parallel.
3. Perform the simulation by repeatedly checking and updating particle states and moving particles in parallel.
4. Copy the final map back to the host and save it to a file.

3.2.2 Detail

To illustrate more about the parallelism, we use CUDA which includes grids and blocks. CUDA kernels are launched in a grid of thread blocks, where each block executes independently on a GPU multiprocessor. Key points include:

- Use `<<<...>>>` notation to launch kernels (`init_particles`, `fill`, `move_particles`) with

specified `init_blocks` and `init_threads`.

- `init_blocks` covers all particles (`number`), ensuring each thread processes one particle.
- Adjust `init_threads` for cases where `number` is less than a predefined base (`base`).

For memory management, CUDA provides specialized memory spaces (`__device__`, `__shared__`, etc.) managed explicitly by the programmer. Use `cudaMalloc` and `cudaMemcpy` for CPU-GPU data transfer (`d_map`, `points`, `d_changed`).

For synchronization, we ensure CUDA operations complete with `cudaDeviceSynchronize()` before dependent CPU operations.

For data parallelism, we leverage GPU parallelism for tasks like particle initialization (`init_particles`), concurrent processing (`fill`, `move_particles`), speeding computations compared to CPUs.

CUDA's grid and block structure, along with optimized memory and synchronization, efficiently parallelize particle simulations. GPU parallelism enhances performance for large-scale computations, surpassing CPU capabilities.

3.3 Other Functions

In addition to the core CUDA functions, the project includes utility functions for plotting the crystallization map and comparing output files of C code and CUDA code. These functions are implemented in Python using the NumPy and Matplotlib libraries.

3.3.1 Plotting Function

The `plot_map` function is used to visualize the crystallization map generated by the simulation.

Functionality:

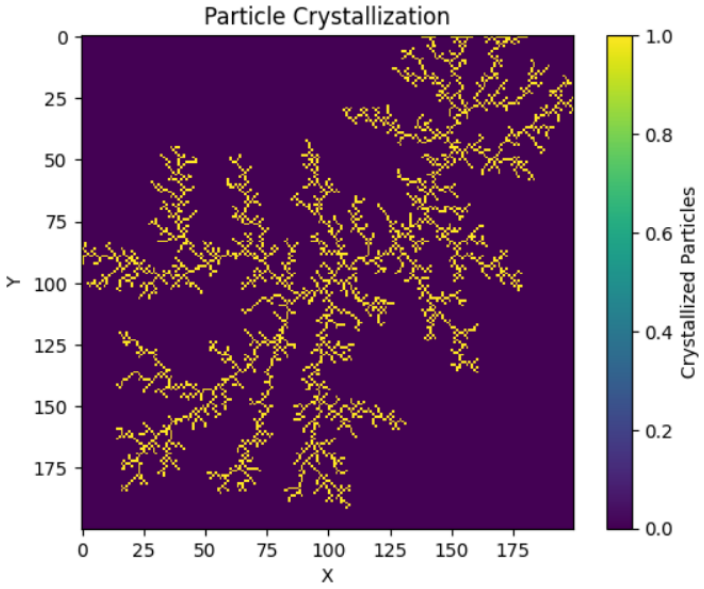
- Load the crystallization data from the specified file.
- Plot the data using Matplotlib, displaying crystallized particles in a color-coded map.
- Add labels and a color bar to the plot for better visualization.

3.3.2 Comparison Function

The `compare_files` function is used to compare two output files line by line to check for any differences.

Functionality:

- Open and read the content of two files.
- Check if the number of lines in both files are the same.



h

Figure 3.1: Map plot of an instance

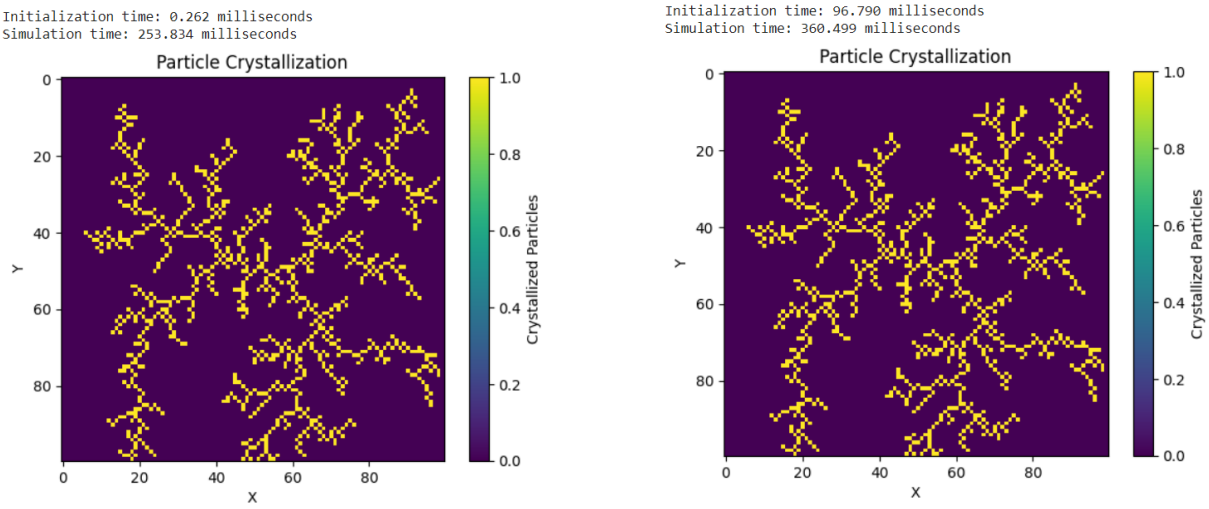
- Compare the content of the files line by line.
- Print the line number and content if a difference is found.
- Return `True` if the files are identical, otherwise `False`.

4 Results

The most important objective of the project is implmenting the code using CUDA, which return the same result as C code in a shorter time. To test the correctness of the CUDA function, we will run both the implementations on a test case, here we will set:

Variable	**Value**
Width	100
Height	100
Number	1000
Steps	10000
sx	50
sy	50

The result of two implementations are below, where both of the implementations return the same:



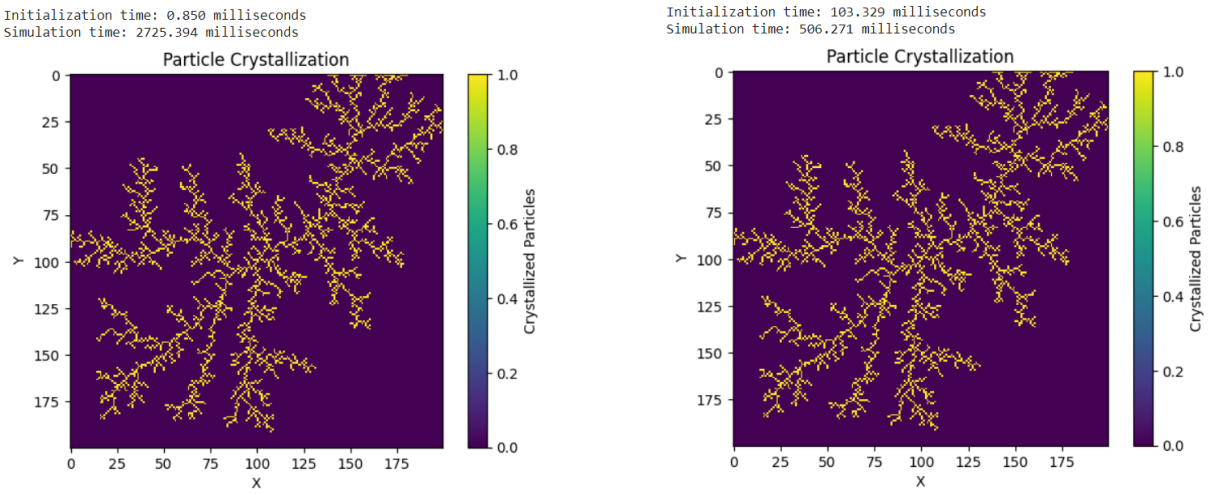
(a) C code output **(b)** CUDA code output

Figure 4.1: Compare the two implementations: Test 1

We also conduct another test for this to make sure our implementation do not luckily return correct output. We set the value of variables much higher than the previous test, the detail as below:

Variable	**Value**
Width	200
Height	200
Number	3000
Steps	10000
sx	100
sy	100

The result of two implementations still return the same, this time the simulation time of CUDA implementation is faster than this of C:



(a) C code output **(b)** CUDA code output

Figure 4.2: Compare the two implementations: Test 2

5 Conclusion

In this project, we have outlined the steps to simulate Diffusion Limited Aggregation using CUDA. DLA is a process that creates complex fractal structures through the random movement and aggregation of particles. By leveraging the parallel computing capabilities of CUDA, we aim to achieve efficient simulation of DLA, allowing for the exploration of larger systems and more intricate patterns.

References

[1] Moore, C and Machta, J *2D Internal Diffusion Limited Aggregation: Parallel Algorithm and Complexity*, Journal of Statistical Physics, 2000. https://scholarworks.umass.edu/items/3817c395-92ec-4e3e-9b94-05e265d3fd89

[2] T.A. Witten Jr. and L.M. Sander, *Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon*, Physical Review Letters, 1981.

[3] L.M. Sander, *Diffusion-limited aggregation: A kinetic critical phenomenon*, Contemporary Physics, 2000.