Diffusion Limited Aggregation with CUDA

Members:

Nguyen Viet Minh	20214917
Phan Duc Hung	20214903
Truong Gia Bach	20210087
Tran Duong Chinh	20210122

HUST

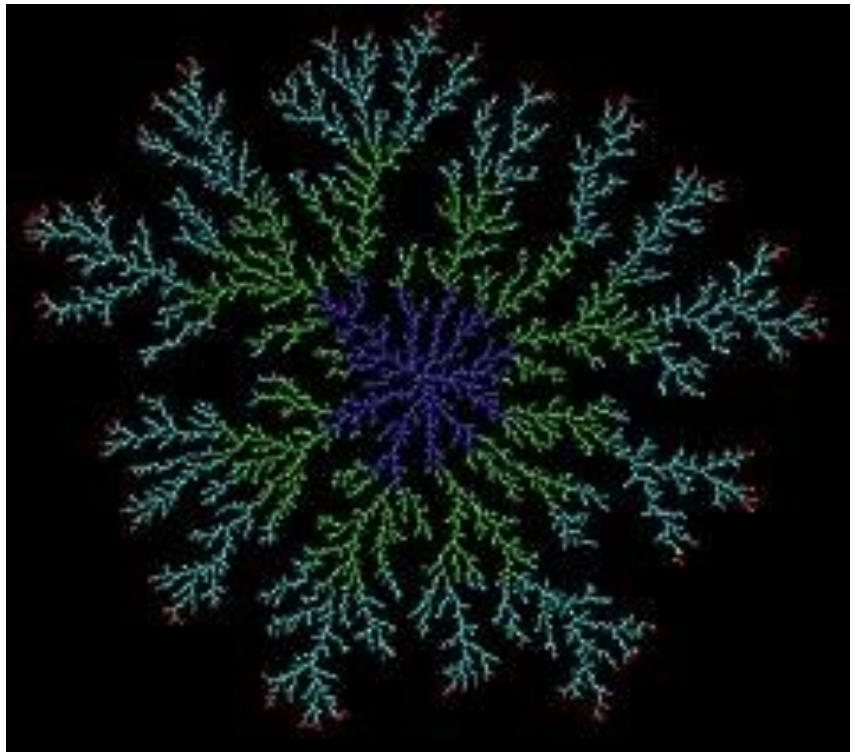
Table Of Content

HUST

1. Introduction

Diffusion Limited Aggregation (DLA) is a nature process, describes the growth of clusters due to the diffusion of particles.

The particles follow a random walk until meet the cluster and stick to it.



Example of Diffusion-limited aggregation

1. Introduction

Applications of DLA:

- Formation of mineral deposits.
- Growth patterns of bacterial colonies.

Aim of our project:

- Implement the DLA process in C code.
- Optimize the running time of implementation using CUDA.

2. Mathematical Formulation

Random Initialization of Particles:

Particles are initialized at random points outside a prefined radius of the cluster:

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

2. Mathematical Formulation

Crystallization Condition:

A particle sticks to the cluster if it moves to a position adjacent to any particle in the cluster:

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

rc denotes the position of a cluster particle

ri(t + 1) is the new position of the moving particle

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

3. Implementation

Main structures and variables:

- Point: Represents a particle with properties x, y, and state.
- width, height: Dimensions of the grid.
- map: The 2D grid where particles aggregate.

3. Implementation

Main structures and variables:

- number: number of particles initialized.
- steps: number of iteration steps for the simulation.
- sx, sy: vertical and horizontal position of the first particle clustered.

3. Implementation

Princical functions:

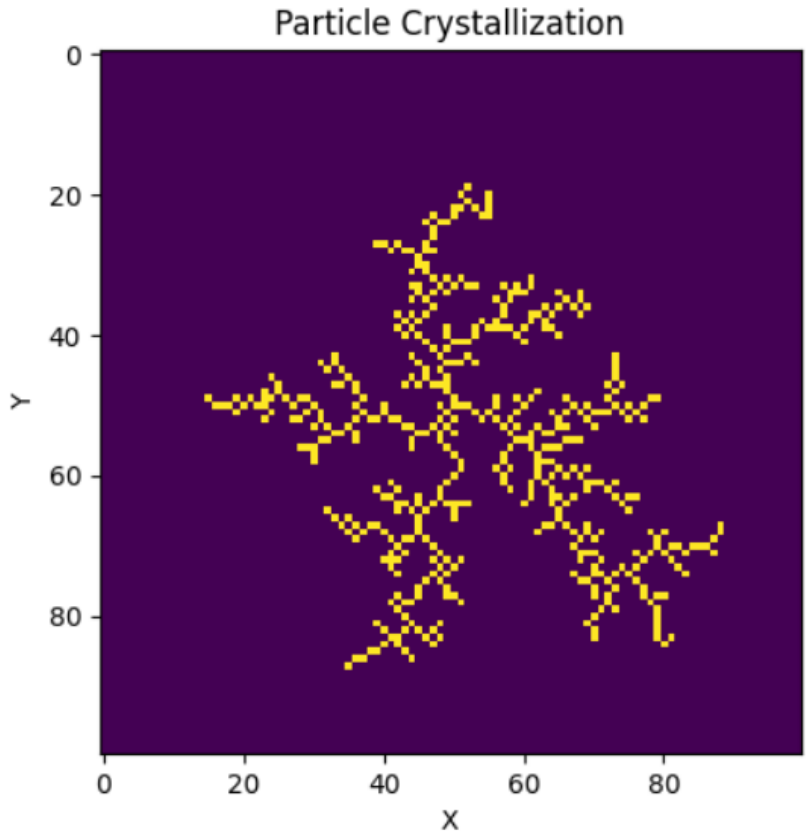
- init_particles(Point* points, int number): Initializes particles at random positions.
- move(Point* p): Moves a particle in a random direction.
- check_occupied(int x, int y): Checks if a neighboring cell is occupied.
- fill(Point* points, int number): Moves particles and updates their state based on neighboring cells.

3. Implementation

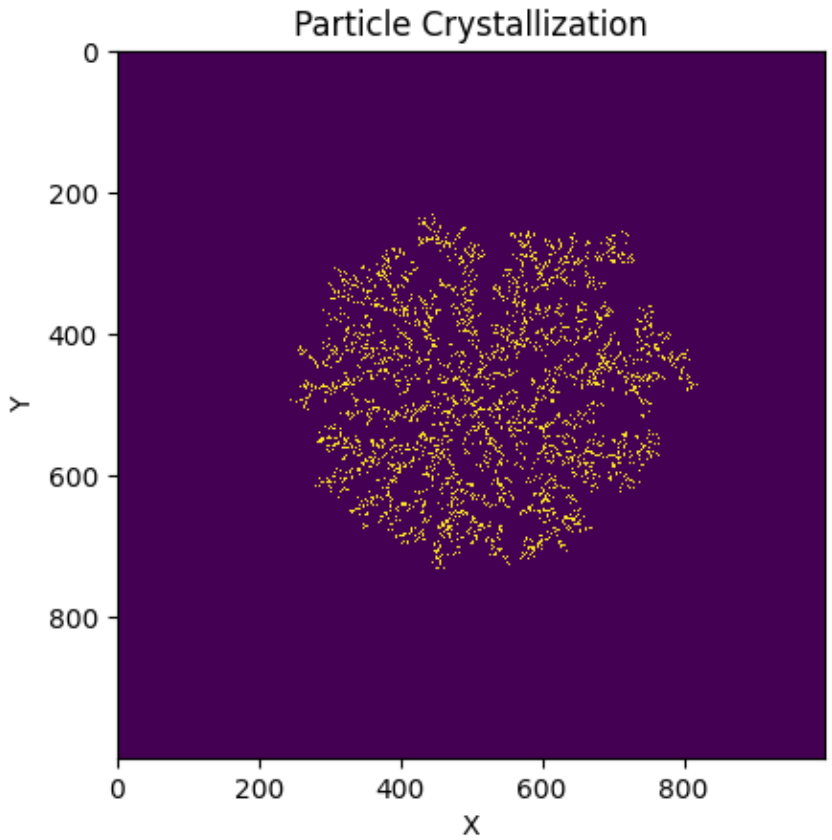
Additional functions:

- cpuSecond(): Returns current time in seconds, use to evaluate the running time.
- random_int(): Use the random function to generate a random integer.
- save_map(const char* file_name, int* map, int width, int height): Saves the grid state to a file .txt.
- plot_map(file_name): python function use matplotlib.pyplot library to visualize.

3. Implementation



Map size (100, 100), 1000 particles



Map size (1000, 1000), 100000 particles

3. CUDA Implementation

Initialize CUDA variables:

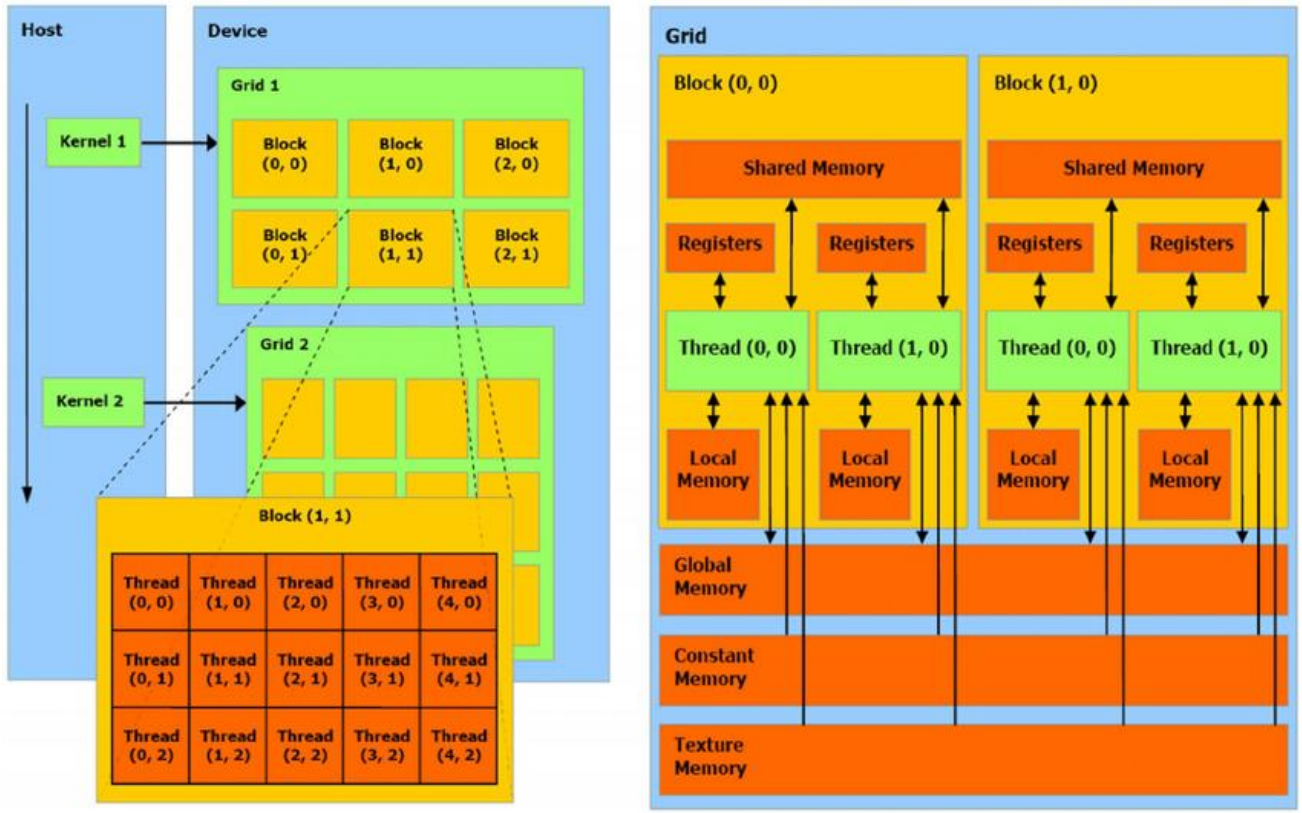
- cudaMemcpyToSymbol(width, &cwidth, sizeof(int));
- cudaMemcpyToSymbol(height, &cheight, sizeof(int));
- cudaMalloc(&d_map, sizeof(int) * cwidth * cheight);
- cudaMemcpyToSymbol(map, &d_map, sizeof(int*));
- cudaMemcpy(d_map, cmap, sizeof(int) * cwidth * cheight, cudaMemcpyHostToDevice);
- cudaMalloc(&points, sizeof(Point) * number);

3. CUDA Implementation

CUDA kernels are launched in a grid of thread blocks, each block executes independently on a GPU multiprocessor:

- Launch kernels (init_particles, fill, move_particles) with specified init_blocks and init_threads.
- init_blocks covers all particle, ensuring each thread processes one particle.
- Adjust init_threads for cases where number is less than a predefined base.

3. CUDA Implementation



CUDA Architecture

3. CUDA Implementation

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).

Synchronization:

- Ensure CUDA operations complete with cudaDeviceSynchronize() before dependent CPU operations.

3. CUDA Implementation

Data parallelism:

- Leverage GPU parallelism for tasks like particle initialization, concurrent processing, speeding computations compared to CPUs.

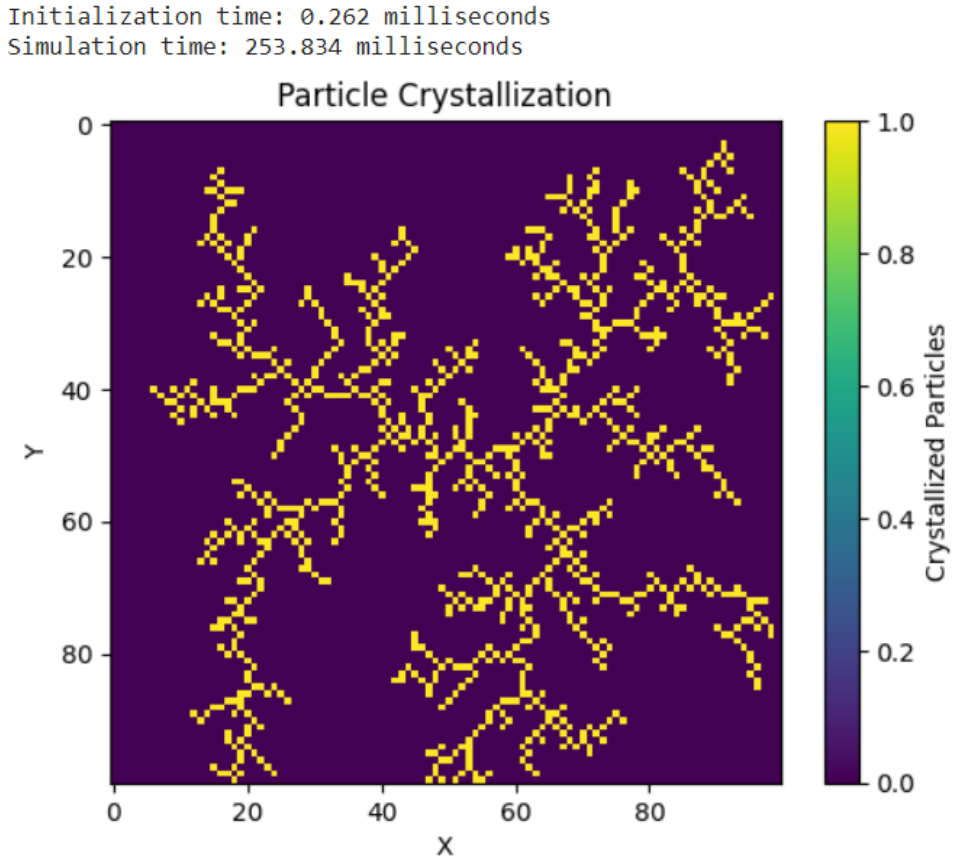
→ CUDA's grid and block structure, along with optimized memory and synchronization, efficiently parallelize particle simulations.

4. Experimental Results

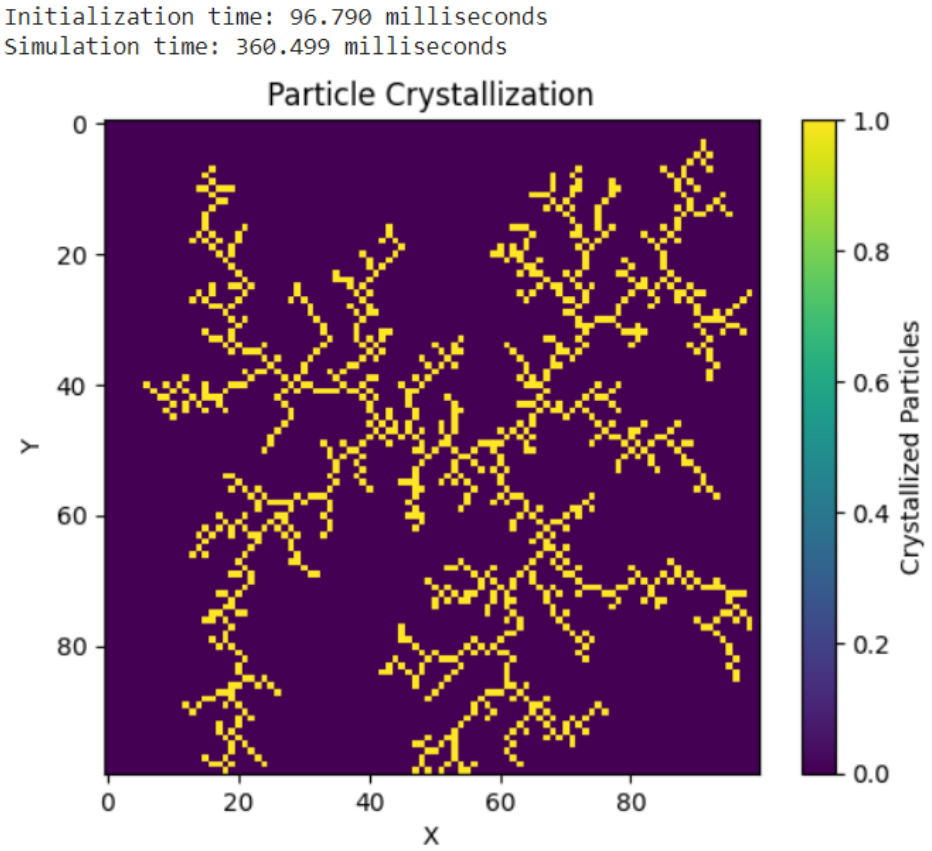
Experimentation:

- Map size: (100, 100)
- Number of particles: 1000
- Number of steps: 10000
- Cluster position: [50, 50]

4. Experimental Results



C code output



CUDA code output

HUST

5.Conclusion

In this project, we have outlined the steps to simulate Diffusion Limited Aggregation using CUDA.

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.

THANK YOU
FOR LISTENING