T13: CUDA Implementation of Diffusion-Limited Aggregation (DLA) for Procedural Generation of Resources

Ajmal
cs22b2046@iiitdm.ac.in

March 19, 2025

1 Introduction

CUDA implementation of a variation of Diffusion-Limited Aggregation (DLA) simulation for procedural resource generation. The simulation models the behavior of particles (walkers) moving randomly on a grid and aggregating to form clusters, where each cluster indicates a particular type of resource (red and blue in our environment, including purple depending on requirements). The CUDA implementation is compared against a serial implementation. The simulation was executed on an NVIDIA T4 GPU, which has 2560 CUDA cores and 16 GB of memory in Google Colab.

2 Performance Comparison

2.1 Serial Implementation

- **Execution Time**: 218.856 seconds

2.2 CUDA Implementation

- **Execution Time**: 6.179 seconds
- **GPU Used**: NVIDIA T4 (2560 CUDA cores, 16 GB memory)

2.3 Speedup

The speedup is calculated as the ratio of the serial execution time (in colab CPU) to the parallel execution time:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{218.856}{6.179} \approx 35.419$$

2.4 Parallelization Factor

The parallelization factor indicates how effectively the parallel implementation utilizes the available hardware resources. It is calculated as:

For the NVIDIA T4 GPU with 2560 CUDA cores, the parallelization factor is:

$$\text{Parallelization Fraction} \approx 0.97214$$

3 Code Explanation

3.1 Grid Initialization

The grid is initialized with a fixed width and height, and a seed is placed at the center to start the aggregation process. Modifying the seed placement will give rise to different structures.

```
Grid grid = GridInit(width, height);
grid.g[(height / 2) * width + (width / 2)] = YELLOW;
grid.g[(height / 2) * width + (width / 2) + 1] = YELLOW;
```

3.2 Walker Initialization

Walkers are initialized with random positions and colors (in device, using curand for thread safety). The `WalkerInit` function sets up the walker's initial state.

```
Walker *h_walkers = (Walker*)malloc(maxWalkers * sizeof(Walker));
for (int i = 0; i < maxWalkers; i++) {
    h_walkers[i] = WalkerInit(0, 0, 0);
    h_walkers[i].active = 1;
}
```

3.3 CUDA Kernel for DLA Simulation

The `dlaKernel` function is the core of the CUDA implementation. It handles the random movement of walkers and their interaction with the grid.

```
__global__ void dlaKernel(int *grid, int width, int height, Walker
    *walkers, int numWalkers, int maxSteps, curandState *states) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= numWalkers) return;

    curandState localState = states[idx];
    Walker walker = walkers[idx];

    if (!walker.active) return;

    int color = (idx % 2 == 0) ? RED : BLUE;
    int edge = curand(&localState) % 4;

    // Spawn walker at edge
```

```
    switch (edge) {
        case 0: walker.x = curand(&localState) % width; walker.y = 0; break;
        case 1: walker.x = curand(&localState) % width; walker.y = height - 1; break;
        case 2: walker.x = 0; walker.y = curand(&localState) % height; break;
        case 3: walker.x = width - 1; walker.y = curand(&localState) % height; break;
    }

    walker.color = color;

    for (int step = 0; step < maxSteps; step++) {
        walkerMove(&walker, grid, width, height, &localState);
        int adjacent = isAdjacentToCluster(grid, width, height, &walker);

        if (adjacent) {
            if (isWalkerClrMatch(&walker, YELLOW) || isWalkerClrMatch(&walker, PURPLE) || isWalkerClrMatch(&walker, walker.color)) {
                walkerPlaceOnGrid(&walker, grid, width);
                spawnGreen(walker, grid, width, height, 1, 0.01, &localState);
            } else {
                detonate(grid, walker.x, walker.y, width, height, 1, 1.0 , 0.4, &localState);
            }
            walker.active = 0;
            break;
        }
    }

    walkers[idx] = walker;
    states[idx] = localState;
}
```

3.4 Batch Processing

The simulation processes walkers in batches to manage memory and computational load efficiently.

```
int batchSize = 1000;
int numBatches = (maxWalkers + batchSize - 1) / batchSize;

for (int batch = 0; batch < numBatches; batch++) {
    int startIdx = batch * batchSize;
    int endIdx = min(startIdx + batchSize, maxWalkers);
    int currentBatchSize = endIdx - startIdx;

    dlaKernel<<<numBlocks, blockSize>>>(d_grid, width, height, d_walkers + startIdx, currentBatchSize, maxSteps, d_states + startIdx);
    cudaDeviceSynchronize();
}
```

4 Results

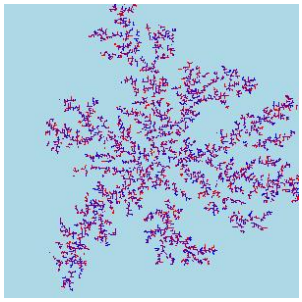

Figure 1: CUDA 1

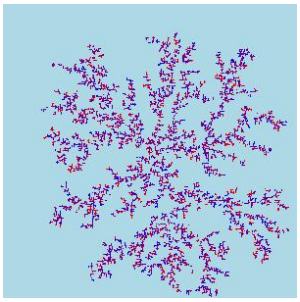

Figure 2: CUDA 2

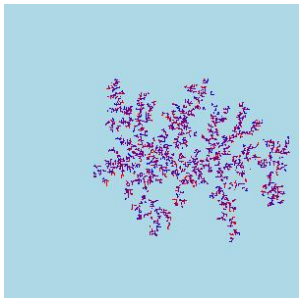

Figure 3: CUDA 3

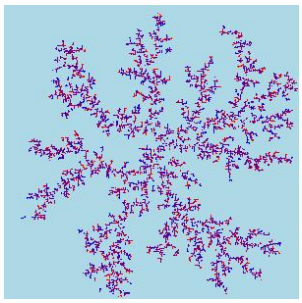

Figure 4: CUDA 4

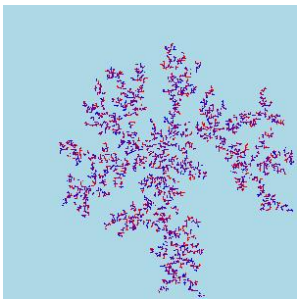

Figure 5: CUDA 5

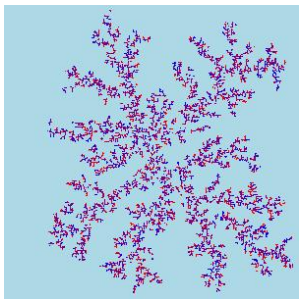

Figure 6: CUDA 6

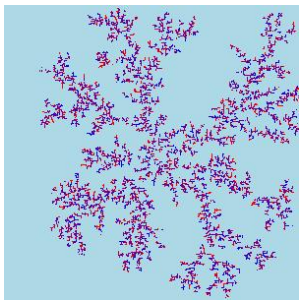

Figure 7: CUDA 7

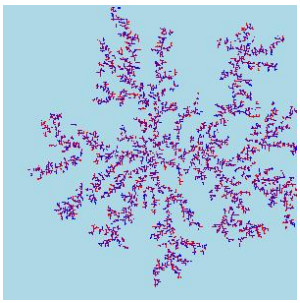

Figure 8: CUDA 8

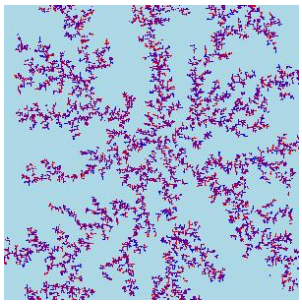

Figure 9: CUDA 9