Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Projektu:

Modelowanie Dyfuzji Metodą LBM (Lattice Boltzmann Method)

Przedmiot: Modelowanie Dyskretne

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący ćwiczenia: prof. dr hab. Inż. Dmytro Svyetlichnyy

Data: 21 grudnia 2024

Numer zadania: 2

Grupa laboratoryjna: 4

Wstęp Teoretyczny

Metoda kratowego równania Boltzmanna (Lattice Boltzmann Method - LBM) to technika numeryczna, która rozwija się z metody LGA (Lattice Gas Automata). LBM opisuje procesy transportowe, takie jak dyfuzja, poprzez zastosowanie funkcji rozkładu na dyskretnej siatce, zachowując podstawowe prawa fizyki, takie jak zasada zachowania masy i pędu

Cel Ćwiczenia

Celem ćwiczenia było opracowanie modelu dyfuzji LBM w przestrzeni 2D i przeprowadzenie przykładowej symulacji, podobnej do symulacji LGA wykonanej w poprzednim zadaniu.

Środowisko Pracy

Implementacja została przeprowadzona w języku **C++** z wykorzystaniem bibliotek:

- **SFML** (Simple and Fast Multimedia Library) – do obsługi grafiki 2D, oraz interfejsu użytkownika.
- **TGUI** (Texus' Graphical User Interface) – do tworzenia elementów graficznego interfejsu użytkownika (GUI).
- **CUDA** (Compute Unified Device Architecture) – do zastosowania obliczeń równoległych przy użyciu GPU.

IDE użytym w pracy było **Visual Studio 2022**.

Realizacja Zadania

W celu przejrzystej implementacji, funkcjonalność została podzielona na klasy, z których każda odpowiada za określony aspekt działania:

- **Controller** – klasa zarządzająca działaniem całego programu, koordynująca interakcje pomiędzy innymi komponentami.
- **Visualization** – odpowiada za wizualizację procesu automatu LBM z wykorzystaniem biblioteki SFML.
- **UI** – zajmuje się pozycjonowaniem graficznego interfejsu użytkownika (GUI) przy użyciu biblioteki TGUI.
- **Grid** – opisuje siatkę automatu komórkowego oraz właściwości komórek. Deklaracja przyjaźni z klasą **Automaton** w celach uniknięcia nakładu obliczeniowego akcesorów i maksymalizacji prędkości obliczeń.

- **Automaton** – implementuje sekwencyjne obliczenia automatu komórkowego LBM.
- **AutomatonCUDA** – implementuje równoległe obliczenia automatu z wykorzystaniem GPU na platformie CUDA.

Opis Modelu

Reprezentacja siatki / komórki

Klasa **Grid** definiuje komórki w siatce w postaci zbioru tablic.

```
// Arrays for cell data
double* concentration;    // 0.0 - (double)direction_num
bool* is_wall;                  // Treated as impassable by gas

// Indexing: [direction][cell_num]
double* f_in[direction_num];    // Input functions
```

Tablice reprezentują:

- **concentration** – koncentracje gazu w danej komórce.
- **is_wall** – flaga definiująca komórkę jako ścianę.
- **f_in** – funkcje wejścia dla 4 kierunków.

Funkcje rozkładu równowagowego oraz funkcje wyjścia nie zostały uwzględnione w definicji komórki ponieważ w obecnym modelu nie znajdują one zastosowania poza funkcją aktualizującą stan automatu. Funkcje te są zdefiniowanie jako zmienne lokalne.

Aplikacja reguł

Opisana zostanie funkcja **update_CPU()** klasy **Automaton**, której zadaniem jest sekwencyjna aktualizacja siatki poprzez zastosowanie procesów kolizji oraz propagacji.

Kolizja (Collision)

- Dla każdej komórki odczytywany jest bieżący stan wejścia w danym kierunki.
- Na podstawie tego wejścia obliczana jest funkcja równowagowa i następnie funkcja wyjściowa z wykorzystaniem współczynnika relaksacji τ.

Propagacja (Streaming)

- Odnajdowana i weryfikowana jest pozycja sąsiada dla danego kierunku.
- Jeżeli sąsiad jest ścianą lub znajduje się poza siatką, kierunek jest odbijany – funkcja wyjścia jest zapisana w funkcji wejścia przeciwnego kierunku. W przypadku przeciwnym – funkcja wyjścia zostaje zapisana jako funkcja wejścia.

Aktualizacja stężenia komórki

- Po zakończeniu propagacji aktualizowane jest stężenie każdej z komórek na podstawie sumy ich funkcji wejściowych

Poniżej został przedstawiony fragment kodu źródłowego metody **update_CPU**, zajmującej się aktualizacją automatu na procesorze. Dla poprawy czytelności, fragment kodu zajmujący się kolizją i propagacją został przesunięty poniżej.

Fragment 1: Ogólny zarys metody

```
void Automaton::update_cpu()
{
	// X-axis loop
	for (int x = 0; x < width; x++)
	{
		// Y-axis loop
		for (int y = 0; y < height; y++)
		{
			// Get this cell's id
			int cell_id = grid.get_id(x, y);

			// Skip if cell is a wall
			if (grid.is_wall[cell_id])
				continue;

			// Loop over all directions
			for (int direction = 0; direction < grid.direction_num; direction++)
			{
				// Streaming & Collision
			}
		}
	}

	// Update Concentration for each cell
	for (int i = 0; i < width * height; i++)
	{
		if (grid.is_wall[i])
		{
			grid.concentration[i] = 0.f;
			continue;
		}

		// Get the sum of all input directions
		double input_sum = 0.f;
		for (int dir = 0; dir < Grid::direction_num; dir++)
			input_sum += grid.f_in[dir][i];

		// Set the concentration of this cell
		grid.concentration[i] = input_sum;
	}
}
```

Proces kolizji oraz propagacji wewnątrz poniższej pętli (obecna w *Fragment 1*)

```
for (int direction = 0; direction < grid.direction_num; direction++)
```

Fragment 2: Kolizja i propagacja wewnątrz pętli kierunku

```
// 1. Collision
// Equilibrium function
double f_eq = grid.weights[direction] * grid.concentration[cell_id];

// Output function
double f_out = grid.f_in[direction][cell_id] + 1.0 /
       grid.tau * (f_eq - grid.f_in[direction][cell_id]);

// 2. Streaming
// Find the neighbour position
int offset_x = grid.directions_x[direction];
int offset_y = grid.directions_y[direction];

int neighbour_x = x + offset_x;
int neighbour_y = y + offset_y;

// Check if the neighbour is either out of bounds or a wall
int neighbour_id = grid.get_id(neighbour_x, neighbour_y);
if (neighbour_x < 0 || neighbour_x >= grid.width ||
     neighbour_y < 0 || neighbour_y >= grid.height ||
     grid.is_wall[neighbour_id])

{      // Bounce Back
     int opposite_dir = grid.opposite_directions[direction];
     grid.f_in[opposite_dir][cell_id] += f_out;
     grid.f_in[opposite_dir][cell_id] /= 2.f;
}
else // Within bounds
{
     grid.f_in[direction][neighbour_id] = f_out;
}
```

Stałe użyte w obliczeniach – fragment klasy **Grid**.

```
public:
      /* Statics & Constants */
      static constexpr int direction_num = 4;      // Left, Top, Right, Down
      static constexpr int directions_x[direction_num] = { -1, 0, 1, 0 };
      static constexpr int directions_y[direction_num] = { 0, 1, 0, -1 };
      static constexpr int opposite_directions[direction_num] = { 2, 3, 0, 1 };
      static constexpr double weights[direction_num] = { 0.25f, 0.25f, 0.25f, 0.25f };

private:
      /* Attributes */
      // Dimensions
      int width;
      int height;

      // Modifiers
      double tau = 1.5f;
```

Wyniki Modelu

Poniżej znajdują się nagrania oraz zrzuty ekranu przedstawiające działanie modelu dla siatki 120x108 z τ równym 1.5.

Nagranie 1

Nagranie 2

Nagranie 3

Przykładowa symulacja