Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

# Sprawozdanie z Projektu:

# Gaz Siatkowy - Lattice Gas Automata (LGA)

Przedmiot: Modelowanie Dyskretne

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

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

Data: 5 grudnia 2024

Numer zadania: 1

Grupa laboratoryjna: 4

## Wstęp Teoretyczny

Lattice Gas Automata (LGA) jest metodą symulacji przepływu płynów opartą na dyskretnym modelu cząsteczek poruszających się i zderzających na regularnej siatce przestrzennej. Pozwala na badanie złożonych zachowań hydrodynamicznych przy użyciu prostych reguł, kolizji i propagacji.

## Cel Ćwiczenia

Celem ćwiczenia była implementacja automatu komórkowego LGA oraz jego wizualizacja w wybranym środowisku.

## 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 LGA z wykorzystaniem biblioteki SFML.
- **UI** – zajmuje się pozycjonowaniem graficznego interfejsu użytkownika (GUI) przy użyciu biblioteki TGUI.
- **Automaton** – implementuje sekwencyjny model automatu komórkowego LGA.
- **AutomatonCUDA** – implementuje model automatu LGA z wykorzystaniem GPU na platformie CUDA.

# Opis Modelu

## Reprezentacja siatki / komórki

Klasa **Automaton** definiuje siatkę komórek jako jednowymiarową tablicę 16-bitowych elementów. Struktura tych elementów jest następująca:

- 4 pierwsze bity reprezentują gaz wchodzący do danej komórki,
- kolejne 4 bity opisują gaz wychodzący,
- dwa ostatnie bity określają typ komórki, tj. powietrze, ścianę lub gaz.

```
// Bit format //
// 15 14 | 13 12 11 10 9 8 | 7 6 5 4 | 3 2 1 0
// State |      Unused     | Outputs | Inputs
uint16_t* cells;
```

Taka reprezentacja siatki została przyjęta w celu minimalizacji kosztów obliczeniowych, ponieważ pozwala na wykorzystanie szybkich operacji przesunięć bitowych do manipulacji danymi. Dzięki temu model jest bardziej wydajny pod względem czasowym i pamięciowym.

## 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.
- Na podstawie aktywnych kierunków wykonywane są odpowiednie operacje zmiany kierunków - przełączenie bitów.
- Wynik kolizji zapisywany jest jako nowy stan wyjścia dla każdej komórki, a wejście jest zerowane, przygotowując komórkę na kolejną fazę.

**Propagacja (Streaming)**

Na podstawie wyjść z każdej komórki cząsteczki są przesuwane do sąsiednich komórek zgodnie z ich kierunkami.

Dane sąsiednich komórek są aktualizowane odpowiednimi bitami, reprezentującymi przesunięcie cząsteczek.

Komórki będące ścianami są pomijane, a propagacja uwzględnia granice siatki.

### Aktualizacja stanów komórek

Po zakończeniu propagacji każda komórka jest klasyfikowana jako gaz, powietrze lub ściana w zależności od aktywności jej wejść i wyjść.

Metoda łączy te trzy etapy, aby zasymulować jeden krok czasowy automatu LGA na procesorze.

Poniżej znajduje się część metody **update_CPU()** zajmująca się kolizją:

```
void Automaton::update_cpu()
{
    // 1. Collisiion
    // Masks
    uint8_t up_down_mask = (1 << Automaton::UP) | (1 << Automaton::DOWN);
    uint8_t left_right_mask = (1 << Automaton::LEFT) | (1 << Automaton::RIGHT);

    // Create copy of the grid
    uint16_t* updated_cells = new uint16_t[width * height];

    // Handle all collision for every cell first
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < height; j++)
        {
            int cell_id = j * width + i;
            uint16_t cell = cells[cell_id];

            // Get the current input
            uint8_t input = get_input(cell);

            // Check if only UP and DOWN are active
            if ((input & up_down_mask) == up_down_mask && (input & ~up_down_mask) == 0)
            {
                // Flip UP and DOWN bits
                input = left_right_mask;
            }
            // Check if only LEFT and RIGHT are active
            else if ((input & left_right_mask) == left_right_mask && (input & ~left_right_mask) == 0)
            {
                // Flip LEFT and RIGHT bits
                input = up_down_mask;
            }

            // Convert input into output
            cell = set_output(cell, input);

            // Clear the input
            cell = set_input(cell, 0);

            // Add the cell to the copy array
            updated_cells[cell_id] = cell;
        }
    }
```

Część metody **update_CPU()** zajmująca się propagacją:

```
// 2. Streaming
uint16_t* streamed_inputs = new uint16_t[width * height];
for (int i = 0; i < width; i++)
{
       for (int j = 0; j < height; j++)
       {
            int cell_id = j * width + i;
            uint16_t cell = updated_cells[cell_id];

            // Skip walls
            if (get_state(cell) == WALL)
                continue;

            // Get the current output
            uint8_t output = get_output(cell);

            // Move particles to neighbouring cells
            if (output & (1 << Automaton::UP))
            {
                int neighbor_id = (j > 0) ? (j - 1) * width + i : -1; // Boundary check
                output_to(updated_cells, cell_id, neighbor_id, UP, DOWN);
            }

            if (output & (1 << Automaton::DOWN))
            {
                int neighbor_id = (j < height - 1) ? (j + 1) * width + i : -1; // Boundary check
                output_to(updated_cells, cell_id, neighbor_id, DOWN, UP);
            }

            if (output & (1 << Automaton::LEFT))
            {
                int neighbor_id = (i > 0) ? j * width + (i - 1) : -1; // Boundary check
                output_to(updated_cells, cell_id, neighbor_id, LEFT, RIGHT);
            }

            if (output & (1 << Automaton::RIGHT))
            {
                int neighbor_id = (i < width - 1) ? j * width + (i + 1) : -1; // Boundary check
                output_to(updated_cells, cell_id, neighbor_id, RIGHT, LEFT);
            }
        }
}
```

Metoda pomocnicza

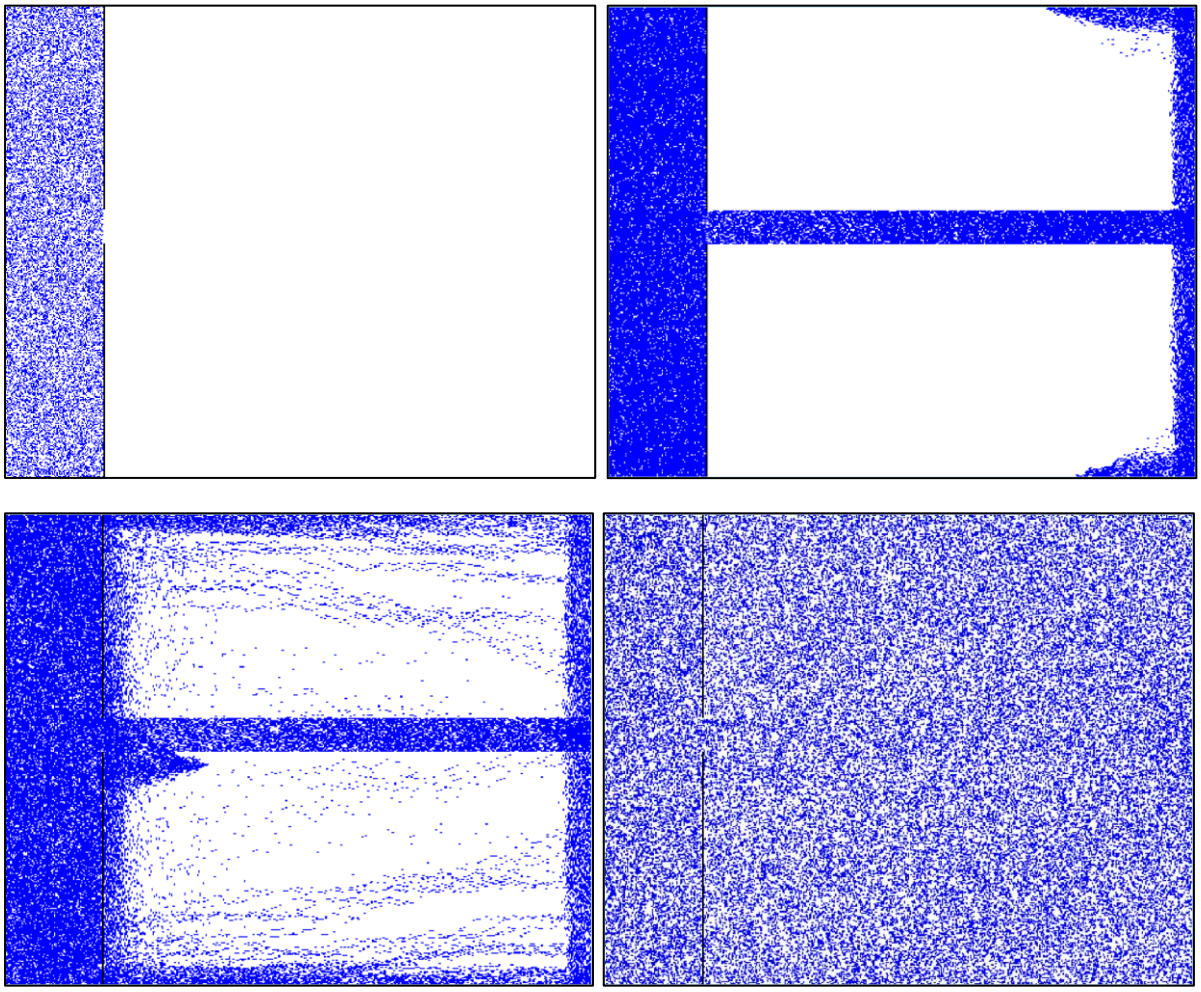
```
// Transfer input direction into output within a neighbour cell
void Automaton::output_to(uint16_t* output_arr, int sender_id, int receiver_id, Direction forward_direction,
Direction opposite_direction)
{
    if (receiver_id == -1 || get_state(output_arr[receiver_id]) == WALL)
    {
        // Reflect forward to opposite direction
        output_arr[sender_id] = set_input(output_arr[sender_id], get_input(output_arr[sender_id]) | (1 <<
opposite_direction));
    }
    else
    {
        uint16_t receiver_cell = output_arr[receiver_id];
        output_arr[receiver_id] = set_input(receiver_cell, get_input(receiver_cell) | (1 << forward_direction));
    }
}
```

Ustawienie stanów komórek oraz aktualizacja tablicy w metodzie **update_CPU():**

```
// Update the states
for (int i = 0; i < height * width; i++)
{
        uint16_t cell = updated_cells[i];

        // Check if the cell is a wall
        if (get_state(cell) == WALL)
        {
            continue; // Skip updating walls
        }

        // Check if there are any active inputs or outputs
        uint8_t input = get_input(cell);
        uint8_t output = get_output(cell);

        if (input != 0 || output != 0)
        {
            // Set the cell to GAS if any direction is active
            cell = set_state(cell, GAS);
        }
        else
        {
            // Otherwise, set the cell to EMPTY
            cell = set_state(cell, EMPTY);
        }

        // Update the cell in the grid
        updated_cells[i] = cell;
}

    // Update the cell array
    std::swap(cells, updated_cells);
    delete[] updated_cells;
```

## Wyniki Modelowania

Poniższe zrzuty ekranu przedstawiają proces symulacji w kolejnych punktach czasowych dla siatki 500 x 400.



Pod poniższym linkiem znajduje się nagranie pokazujące przebieg wybranych symulacji.