Практическая работа №5

Тема: Реализация параллельных структур данных на GPU

Цель работы:

1. Освоить программирование параллельных структур данных с использованием CUDA.
2. Реализовать параллельные структуры данных (например, параллельный стек и очередь).
3. Провести исследование производительности реализованных структур данных.

Теоретическая часть

1. Параллельные структуры данных

Параллельные структуры данных — это структуры, которые поддерживают безопасный доступ к данным несколькими потоками одновременно. Их разработка требует эффективной синхронизации и минимизации конфликтов при доступе к памяти.

Основные структуры данных:

- **Стек (LIFO):** Добавление и удаление элементов происходит только с одного конца.
- **Очередь (FIFO):** Добавление происходит в конец, а удаление — из начала.

Применение параллельных структур данных:

- Буферизация данных.
- Реализация задач планирования (Task Scheduling).
- Организация потоков данных в алгоритмах обработки.

2. Программирование с использованием CUDA

CUDA — это параллельная вычислительная платформа и программная модель, разработанная NVIDIA, которая позволяет разработчикам использовать графические процессоры (GPU) для выполнения общих вычислений.

Ключевые концепции CUDA:

- **Блоки и потоки:** Потоки объединяются в блоки, а блоки — в сетку для масштабируемости вычислений.

- **Память:** Глобальная, разделяемая и локальная память для оптимизации производительности.
- **Синхронизация:** Механизмы для управления доступом к данным.

Практическая часть

Часть 1. Реализация параллельного стека на CUDA

Задание: Реализовать структуру данных стек с использованием атомарных операций для безопасного доступа к данным.

Пример кода для реализации стека:

```
struct Stack {
    int *data;
    int top;
    int capacity;

    __device__ void init(int *buffer, int size) {
        data = buffer;
        top = -1;
        capacity = size;
    }

    __device__ bool push(int value) {
        int pos = atomicAdd(&top, 1);
        if (pos < capacity) {
            data[pos] = value;
            return true;
        }
        return false;
    }

    __device__ bool pop(int *value) {
        int pos = atomicSub(&top, 1);
        if (pos >= 0) {
            *value = data[pos];
            return true;
        }
        return false;
    }
};
```

Задачи:

1. Инициализировать стек с фиксированной емкостью.
2. Написать ядро CUDA, использующее push и pop параллельно из нескольких потоков.
3. Проверить корректность выполнения операций.

Часть 2. Реализация параллельной очереди на CUDA

Задание: Реализовать очередь с использованием атомарных операций для безопасного добавления и удаления элементов.

Пример кода для реализации очереди:

```
struct Queue {
    int *data;
    int head;
    int tail;
    int capacity;

    __device__ void init(int *buffer, int size) {
        data = buffer;
        head = 0;
        tail = 0;
        capacity = size;
    }
    __device__ bool enqueue(int value) {
        int pos = atomicAdd(&tail, 1);
        if (pos < capacity) {
            data[pos] = value;
            return true;
        }
        return false;
    }
    __device__ bool dequeue(int *value) {
        int pos = atomicAdd(&head, 1);
        if (pos < tail) {
            *value = data[pos];
            return true;
        }
        return false;
    }
};
```

Задачи:

1. Инициализировать очередь с заданной емкостью.
2. Написать ядро CUDA, использующее enqueue и dequeue параллельно.
3. Сравнить производительность очереди и стека.

Контрольные вопросы

1. В чём отличие стека и очереди?
2. Какие проблемы возникают при параллельном доступе к данным?
3. Как атомарные операции помогают избежать конфликтов в параллельных структурах данных?
4. Какие типы памяти CUDA используются для хранения данных?
5. Как синхронизация потоков влияет на производительность?
6. Почему разделяемая память важна для оптимизации работы параллельных структур данных?

Дополнительные задания

1. Реализовать очередь с поддержкой нескольких производителей и потребителей (MPMC).
2. Оптимизировать использование памяти, включая работу с разделяемой памятью.
3. Сравнить производительность реализованных структур данных с последовательными версиями.

Заключение

Эта практическая работа направлена на изучение реализации и оптимизации параллельных структур данных с использованием CUDA. Работа с динамическими структурами данных в параллельных вычислениях требует особого внимания к синхронизации и управлению памятью для достижения высокой производительности.

Рекомендуемая литература:

- Документация NVIDIA CUDA: https://docs.nvidia.com/cuda/