МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №1**

**по курсу «Параллельная обработка данных»**

**Сортировка чисел на GPU. Свертка, сканирование, гистограмма**

Выполнил: Н.И. Лохматов

Группа: 8О-406Б

Преподаватель: А.Ю. Морозов

Москва, 2024

**Условие**

1. Цель работы: ознакомление с фундаментальными алгоритмами GPU: свертка (reduce), сканирование (blelloch scan) и гистограмма (histogram). Реализация одной из сортировок на CUDA. Использование разделяемой и других видов памяти. Исследование производительности программы с помощью утилиты nvprof.
2. Вариант 2: сортировка подсчетом. Диапазон от 0 до − 1

**Программное и аппаратное обеспечение**

1. Графический процессор: Nvidia GeForce RTX 3050 Mobile
   1. Количество потоковых процессоров: 2560
   2. Частота ядра: 1552 МГц
   3. Количество транзисторов: 8.7 млрд
   4. Тех. процесс: 8 нм
   5. Энергопотребление: 80 Вт
2. OC: Ubuntu 22.04
3. Текстовый редактор: VS Code
4. Компилятор: nvcc

**Метод решения**

Сначала строится гистограмма массива. После построения гистограммы выполняется её сканирование. Этот процесс преобразует массив таким образом, чтобы каждый элемент содержал индекс начала своего числа в отсортированном массиве. Сканирование реализовано поэтапно: сначала внутри каждого блока потоков вычисляются локальные суммы, затем эти суммы объединяются для всех блоков, и, наконец, корректируются значения для получения глобальной префиксной суммы. В конце запись отсортированных данных. Каждый поток обращается к массиву гистограммы, чтобы определить, куда записать числа в выходной массив. Для каждого числа из гистограммы заполняется диапазон в итоговом массиве, начиная с рассчитанной позиции и в количестве, равном частоте появления числа.

**Описание программы**

Программа реализована в основном файле, где находится логика сортировки массива чисел методом гистограммного подхода, а также макрос для обработки ошибок и CUDA-ядра для выполнения вычислений на GPU. Сначала входные данные считываются, после чего запускается процесс построения гистограммы. Для этого каждое число из массива подсчитывается с помощью атомарных операций, чтобы избежать конфликтов между потоками. На следующем этапе запускается ядро для выполнения операции сканирования над массивом гистограммы. Сначала в каждом блоке вычисляются локальные суммы, затем результаты блоков объединяются, и корректировки применяются к каждому блоку для получения глобальной префиксной суммы. В финальном этапе запускается ядро записи отсортированных данных, где каждый поток берет информацию из преобразованного массива гистограммы и заполняет итоговый массив числами в соответствующих позициях. Потоки работают независимо, что обеспечивает высокую степень параллелизма.

Программа корректно обрабатывает ошибки вызовов CUDA, освобождает память на GPU после завершения выполнения и выводит отсортированный массив в бинарный поток.

**Результаты**

1. Зависимость времени выполнения программы от BLOCK\_SIZE. Вычисления были проведены 100 раз и на их основе посчитано среднее время

|  |  |  |  |
| --- | --- | --- | --- |
| Количество потоков | Время, n = 10^3 (мс) | Время, n = 10^5 (мс) | Время, n = 10^6 (мс) |
| 1×32 | 929.38 | 1412.38 | 2801.55 |
| 32×32 | 326.46 | 299.91 | 449.54 |
| 128×128 | 319.68 | 287.54 | 379.12 |
| 512×512 | 311.87 | 283.12 | 326.44 |
| 1024×1024 | **303.76** | **279.21** | **314.63** |

2. Сравнение программы на CUDA с 1024×1024 потоками и программы на CPU с одним потоком

|  |  |  |
| --- | --- | --- |
| Размер n | Время CUDA (мс) | Время CPU (мс) |
| 10^3 | 303.76 | **134.72** |
| 10^5 | **279.21** | 23119.46 |
| 10^6 | 314.63 | 2062039.11 |

3. Результаты исследования производительности с помощью nvprof. Количество потоков 1024×1024, n = 10^4

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Ядро finalRes занимает 97.78% общего времени на GPU с временем выполнения 240.56 мс за один вызов. Это основной этап программы, требующий оптимизации, так как на него приходится подавляющая часть вычислений. Ядро prefixScan, отвечающее за выполнение префиксной суммы, занимает 1.83% времени (4.511 мс на три вызова). Ядро addScan, выполняющее корректировку промежуточных данных, использует 0.25% времени (622.96 мкс на два вызова). На инициализацию массива initRes затрачено 0.12% времени (295.13 мкс за один вызов), что не является узким местом программы. Ядро hist, выполняющее подсчёт гистограммы, также занимает лишь 0.01% времени (26.944 мкс).

Из API-вызовов значительное время (43.65%) затрачивается на выделение памяти с помощью cudaMalloc, которое занимает 191.35 мс на 8 вызовов. cudaMemcpy (копирование данных между устройством и хостом) занимает 0.02% времени (95.851 мкс на два вызова), что свидетельствует о низких затратах на передачу данных. Остальные вызовы API, такие как cudaLaunchKernel и синхронизация через cudaDeviceSynchronize, вносят небольшой вклад в общее время выполнения.

Используем ключ --print-gpu-trace, чтобы получить подробные данные по каждому ядру и узнать, какое ядро занимает больше всего времени, можно использовать (n = 10^4).

**Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание**

Передача данных между CPU и GPU демонстрирует высокий пропускной поток (до 7.7096 GB/s), при этом общий объём данных составляет 39.063 KB. Это указывает на эффективное использование PCIe-соединения для передачи данных. Операции CUDA выполняются с различными конфигурациями сетки и блоков. Большинство операций используют одномерную сетку с конфигурацией (1024 1 1) и размер блока (1024 1 1), что позволяет задействовать большое количество потоков для параллельных вычислений.

Для операции finalRes используется двухмерная структура сетки и блоков, что позволяет распределить вычисления по двум измерениям, оптимизируя операции записи данных. Количество регистров на поток варьируется от 16 до 26, что свидетельствует об умеренном использовании регистров, минимизирующем переполнения в локальную память. Использование разделяемой памяти фиксируется только в нескольких операциях, где она используется эффективно (до 4 KB на блок).

**Выводы**

Реализована сортировка чисел на GPU с использованием метода гистограммы и операции префиксной суммы. Для вычислений применены атомарные операции для подсчета частот чисел, а также эффективное сканирование с использованием разделяемой памяти для минимизации обращений к глобальной памяти. Для параллельной обработки массивов использована трехмерная структура сетки потоков, обеспечивающая равномерное распределение вычислений между потоками CUDA. Производительность программы исследована с помощью утилиты nvprof, что позволило выявить основные время затратные этапы алгоритма и подтвердить высокую эффективность реализации по сравнению с последовательными методами сортировки.