МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №3**

**по курсу «Параллельная обработка данных»**

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

Выполнил: А.С. Федоров

Группа: 8О-407Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2022

**Условие**

**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.

**Вариант задания.** 8. Поразрядная сортировка.

**Программное и аппаратное обеспечение**

**Графический процессор**

Название: NVIDIA GeForce GTX 1050

Compute capability: 6.1

Объем графической памяти: 2147352576 байтов

Объем разделяемой памяти на блок: 49152 байтов

Объем регистров на блок: 65536

Размер варпа: 32

Максимальное количество потоков на блок: (1024, 1024, 64)

Максимальное число блоков: (2147483647, 65535, 65535)

Объем постоянной памяти: 65536 байтов

Число мультипроцессоров: 5

**Процессор**

Название: i5-8250U

Базовая тактовая частота процессора: 1,60 ГГц

Количество ядер: 4

Количество потоков: 8

Кеш L1: 256 Кб

Кеш L2: 1 Мб

Кеш L3: 6 Мб

**Оперативная память**

Тип: DDR4

Объем: 11.9 Гб

Частота: 2400 МГц

**Програмное обеспечение**

ОС: WSL2 (Windows 11)

IDE: Microsoft Visual Studio 2022 (аддон NVIDIA Nsight)

Компилятор: nvcc

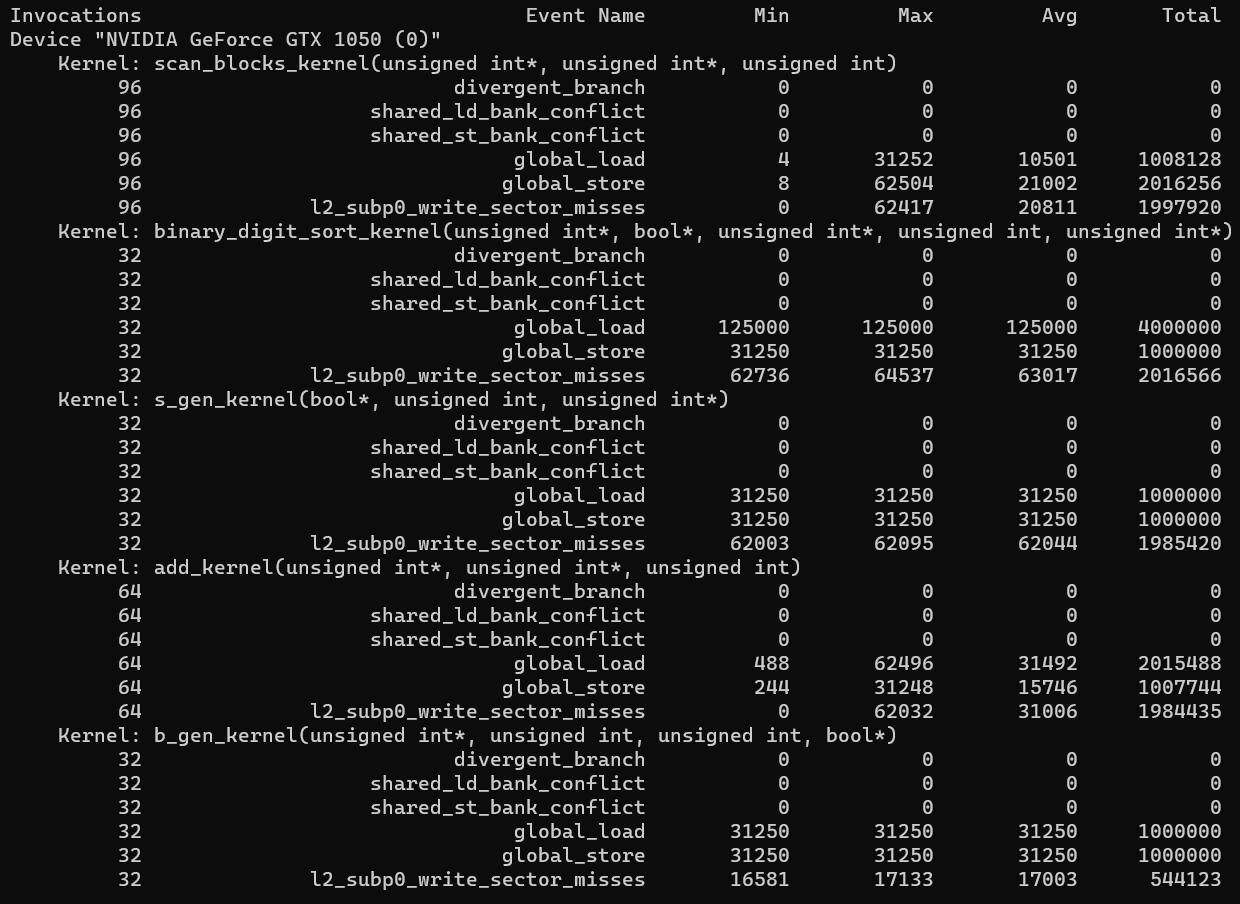
**Метод решения**

Поразрядная сортировка по битам осуществляется итерационно. Начиная с первого и заканчивая 32-м битом числа. На каждой итерации посчитываются исключающие префиксные суммы для текущего бита и на их основе выполняется устойчивая сортировка тоже по текущему биту. Так как значение бита может быть ноль или один, сортировку можно свести к простому правилу: если текущий бит числа равен нулю, то его новая позиция – это , где *i* – изначальная позиция, а *s[i]* – *i*-ое значение префиксной суммы, иначе новая позиция – это , где *n* -число элементов в массиве, а *s[n]* – сумма всех текущих битов чисел.

**Описание программы**

Алгоритм сортировки реализован в виде функции, которая вызывает в цикле следующие ядра: ядро вычисления значений битов для элементов массива за сохранение их в массив типа bool, ядро копирования этого массива, функция scan, которая внутри себя вызывает еще ядра, сортировка элементов массива по правилу, описанному выше. Во время сортировки поддерживается два массива: расположение элементов на прошлой итерации и новое положение.

Результат профилировки программы на тесте размером в :



Конфликтов банков памяти нет.

Код алгоритма blelloch scan:

**const** **uint** BLOCK\_SIZE = 128;

**const** **uint** LOG2\_BLOCK\_SIZE = 7;

**const** **uint** GRID\_SIZE = 64;

**\_\_global\_\_** **void** scan\_blocks\_kernel(**uint\*** data, **uint\*** sums, **uint** sums\_size) {

**int** blockId = **blockIdx**.x;

    while (blockId < sums\_size) {

**extern** **\_\_shared\_\_** **uint** temp[];

        temp[\_index(threadIdx.x)] = data[blockDim.x \* blockId + **threadIdx**.x];

        \_\_syncthreads();

**uint** stride = 1;

        for (**uint** d = 0; d < LOG2\_BLOCK\_SIZE; ++d) {

            if ((threadIdx.x + 1) % (stride<<1) == 0) {

                temp[\_index(threadIdx.x)] += temp[\_index(threadIdx.x-stride)];

            }

            stride <<= 1;

            \_\_syncthreads();

        }

        if (threadIdx.x == 0) {

            sums[blockId] = temp[\_index(BLOCK\_SIZE - 1)];

            temp[\_index(BLOCK\_SIZE - 1)] = 0;

        }

        \_\_syncthreads();

        stride = 1 << LOG2\_BLOCK\_SIZE-1;

        while (stride != 0) {

            if ((threadIdx.x + 1) % (stride<<1) == 0) {

**uint** tmp = temp[\_index(threadIdx.x)];

                temp[\_index(threadIdx.x)] += temp[\_index(threadIdx.x - stride)];

                temp[\_index(threadIdx.x - stride)] = tmp;

            }

            stride >>= 1;

            \_\_syncthreads();

        }

        data[blockDim.x \* blockId + **threadIdx**.x] = temp[\_index(threadIdx.x)];

        blockId += **gridDim**.x;

    }

}

**\_\_global\_\_** **void** add\_kernel(**uint\*** data, **uint\*** sums, **uint** sums\_size) {

**uint** blockId = **blockIdx**.x+1;

    while (blockId < sums\_size) {

        if (blockId != 0) {

            data[blockDim.x \* blockId + **threadIdx**.x] += sums[blockId];

        }

        blockId += **gridDim**.x;

    }

}

**void** scan(**uint\*** dev\_data, **uint** size) {

    if (size % BLOCK\_SIZE != 0)

        size += BLOCK\_SIZE - (size % BLOCK\_SIZE);

**uint** sums\_size = size/BLOCK\_SIZE;

**uint**\* dev\_sums;

    CSC(cudaMalloc(&dev\_sums, (sums\_size \* sizeof(**uint**))));

    scan\_blocks\_kernel << < GRID\_SIZE, BLOCK\_SIZE, \_index(BLOCK\_SIZE) \* sizeof(**uint**) >> > (dev\_data, dev\_sums, sums\_size);

    CSC(cudaGetLastError());

    if (size <= BLOCK\_SIZE)

        return;

    scan(dev\_sums, sums\_size);

    add\_kernel << < GRID\_SIZE, BLOCK\_SIZE >> > (dev\_data, dev\_sums, sums\_size);

    CSC(cudaGetLastError());

    cudaFree(dev\_sums);

}

**Результаты (в миллисекундах)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Размер теста (по вертикали и горизонтали)** | | | | | |
|  |  |  |  |  |  |
| **GPU(CUDA)**  **<1, 32>** | 10.850 | 5.356 | 32.155 | 279.678 | 2657.003 | 25989.757 |
| **GPU(CUDA)**  **<32, 32>** | 3.450 | 2.729 | 13.493 | 15.485 | 114.099 | 938.435 |
| **GPU(CUDA)**  **<64, 32>** | 2.598 | 2.684 | 6.326 | 10.552 | 64.897 | 540.955 |
| **GPU(CUDA)**  **<64, 64>** | 4.294 | 2.869 | 4.296 | 7.806 | 45.551 | 353.679 |
| **GPU(CUDA)**  **<128, 64>** | 3.148 | 3.303 | 4.785 | 11.233 | **33.200** | **250.957** |
| **GPU(CUDA)**  **<128, 128>** | 1.557 | 11.398 | 10.950 | **6.462** | 36.288 | 275.310 |
| **GPU(CUDA)**  **<256, 128>** | 6.685 | 4.495 | **3.068** | 7.048 | 35.299 | 283.017 |
| **CPU** | **0.051** | **0.509** | 5.182 | 51.356 | 531.957 | 5219.426 |

**Выводы**

Алгоритм сортировки имеет применение практически везде, где есть программирование в принципе. Распараллеливание данной задачи на GPU может быть особенно полезно, если сортировку необходимо выполнить на очень большом объеме данных. В ходе реализации основной сложностью была реализация алгоритма blelloch scan. Также для прохождения ограничений по времени и по памяти были сделаны некоторые доработки. Было реализовано быстро целочисленное деление и умножение на два и хранение битов как bool.

Судя по результатам тестирования распараллеливание на большее число потоков и блоков дает хороший прирост по скорости практически на всех размерах тестируемых данных. Как и ожидалось, CPU работает быстрее на маленьких данных, но начинает сильно проигрывать на больших. Для тестов размером и ускорения после варианта в 128 блоков и 64 потока нет. Видимо потоков становится достаточно, чтобы на каждый элемент пришлось по одному.