МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №5
по курсу «Параллельная обработка данных»

Сортировка чисел на GPU. Свертка, сканирование, гистограмма.

Выполнил: Лобанов О. А.
Группа: 8О-408Б-20
Преподаватель: А.Ю. Морозов

Москва, 2023

Условие
Цель работы: 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.

Вариант 2. Сортировка подсчетом.

Программное и аппаратное обеспечение

Device: Tesla T4
Compute capability: 7.5
Total constant memory: 65536
Registers per block: 65536
Max threads per block: 1024
Multiprocessors count: 40
OS: Ubuntu 22.04.2 LTS
Redactor: colab google

Метод решения

Главная идея алгоритма — посчитать, сколько раз встречается каждый элемент в массиве, а потом заполнить исходный массив результатами этого подсчёта. Для этого нам нужен вспомогательный массив, где мы будем хранить результаты подсчёта. Даже если нам надо отсортировать миллион чисел, мы всё равно знаем диапазон этих чисел заранее, например, от 1 до 100. Это значит, что во вспомогательном массиве будет не миллион элементов, а сто.
Сложность данного алгоритма $O(n + k)$.

Описание программы

Алгоритм сортировки

```
__global__ void CountSort(int* input, int* histogram, int* outPut, int size) {
    int xdx =  threadIdx.x + blockDim.x * blockIdx.x;
    int offsetx = gridDim.x * blockDim.x;
    for (int i = xdx; i < size; i +=offsetx)
    {
        outPut[atomicAdd(&input[histogram[i]], -1) - 1] = histogram[i];
    }
```

Сканирование и гистограмма

```
__global__ void Histohram(int* input, int* histogram, int size) {
    int xdx = blockDim.x * blockIdx.x + threadIdx.x;
```

```
    int offsetx = blockDim.x * gridDim.x;
    for (int i = xdx; i < size; i+= offsetx) {
        atomicAdd(&input[histogram[i]], 1);
    }
}

__global__ void ScanKernel(int* histogram, int* arr) {
    int kek = blockDim.x;
    __shared__ int SharedHistogram[1024];
    int check = 1;
    int xdx = blockDim.x * blockIdx.x + threadIdx.x;
    SharedHistogram[threadIdx.x] = histogram[xdx];
    __syncthreads();

    while (kek > check) {
        if (2 * check + threadIdx.x * 2 *check - 1 < kek) {
            SharedHistogram[2 * check + threadIdx.x * 2 * check - 1] +=
SharedHistogram[check+ threadIdx.x *  check * 2 - 1];
        }
        check= check* 2;
        __syncthreads();
    }
    int temp = 0;

    if (threadIdx.x == kek - 1) {
        temp = SharedHistogram[threadIdx.x];
        SharedHistogram[threadIdx.x] = 0;
    }
    check = check / 2;
    __syncthreads();

    while (check >= 1) {
        if (check* 2 * threadIdx.x + 2 * check - 1 < kek) {
            auto swap = SharedHistogram[2 *check * threadIdx.x + check - 1];
            SharedHistogram[check * 2 * threadIdx.x + check - 1] =
SharedHistogram[check * 2 * threadIdx.x + 2 * check - 1];
            SharedHistogram[check * 2 * threadIdx.x + 2 * check - 1] += swap;
        }
        check = check /2;
        __syncthreads();
    }

    if (threadIdx.x == kek - 1) {
        histogram[xdx] = temp;
        arr[blockIdx.x] = temp;
    } else {
        histogram[xdx] = SharedHistogram[threadIdx.x + 1];
    }
}
```

```
__global__ void Shift(int* histogram, int* newArr) {
    int xdx = blockDim.x * blockIdx.x + threadIdx.x;
    if (0 < blockIdx.x)
        histogram[xdx] += newArr[blockIdx.x - 1];
}

void Scan(int* input, int size) {
    int maximum = fmax(1.0, ((double)size / 1024.0));
    int minimum = fmin((double)size, 1024.0);
    int* newInput;
    CSC(cudaMalloc((void**)&newInput, sizeof(int) * maximum));
    ScanKernel<<< maximum, minimum >>>(input, newInput);
    cudaDeviceSynchronize();
    if (size > 1024)
    {
        Scan(newInput, (size/ 1024));
        Shift<<<(size / 1024), 1024 >>>(input, newInput);
        cudaDeviceSynchronize();
    }
    cudaFree(newInput);
}
```

Результаты

	Размер	Время
CPU	10^6	0.503
GPU <<<256, 256 >>>	10^6	0.214
GPU <<<64, 64 >>>	10^6	0.228
CPU	10^8	112.307
GPU <<<256, 256>>>	10^8	25.251
GPU <<<64, 64>>>	10^8	24.947

Выводы

В данной лабораторной работе был реализован алгоритм сортировки подсчетом с использованием скан, который подходит для работы с массивами любого размера. Также познакомился с работой с разделяемой памятью.