МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет «Информационные технологии и прикладная математика»
Кафедра «Вычислительная математика и программирование»

Лабораторная работа №5
по курсу «Программирование графических процессоров»

Сортировка чисел на GPU. Свертка, сканирование, гистограмма.

Студент: Лысенко Д.А.
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2019

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.
2. *Вариант 1.* Битоническая сортировка.

Требуется реализовать битоническую сортировку для чисел типа int.

Должна быть реализована адаптивная операция битонического слияния. Если данные помещаются в разделяемую память, то взаимодействие идет через неё, если нет, то через глобальную память(т.е. необходимо реализовать несколько вариантов ядра).

Ограничения: $n \leq 256 * 10^6$

Программное и аппаратное обеспечение

Computecapability: 5.0
Name: GeForce GTX 960M
Total Global Memory: -2147483648
Shared memory per block: 49152
Registers per block: 65536
Warp size: 32
Max threads per block: (1024, 1024, 64)
Max block: (2147483647, 65535, 65535)
Total constant memory: 65536
Multiprocessors count: 5
Процессор (CPU) –Intel Core i5-6300HQ 2.30GHz; 8 ГБ RAM;
OS Microsoft Windows 10 (x64)
CUDA V10.1

Метод решения

Для каждого шага слияния и сортировки бионической последовательности мы запускаем вычисления на ядре, которые производят сравнение двух элементов массива с заданным шагом и сортируют их по возрастанию или по убыванию, в зависимости от шага алгоритма. Если данные изначально умещаются в разделяемую память, то она используется при сортировке, иначе используется глобальная память.

Описание программы

Тело битонической сортировки, выполняющееся на хосте

```
__host__ void bitonicSort(int32_t * arr, uint32_t roundSize, uint32_t size) {

    int32_t * devArr;
    ErrorCheck(cudaMalloc(&devArr, roundSize * sizeof(int32_t)));
    ErrorCheck(cudaMemcpy(devArr, arr, roundSize * sizeof(int32_t), cudaMemcpyHostToDevice));
    bool flag = roundSize < 512 * 8;
    for (uint32_t mergeStep = 2; mergeStep <= roundSize; mergeStep <<= 1) {
        for (uint32_t step = mergeStep >> 1; step > 0; step >>=  1) {

            if (flag)
                bitonicSortShared<<<1, 512>>>(devArr, mergeStep, step, roundSize);
            else
                bitonicSort<<<32, 512 >> > (devArr, mergeStep, step, roundSize);
        }
        ErrorCheck(cudaGetLastError());
    }
    ErrorCheck(cudaMemcpy(arr, devArr, size * sizeof(int32_t), cudaMemcpyDeviceToHost));
    ErrorCheck(cudaFree(devArr));
    return;
}
```

Функции сравнения и обмена, выполняющиеся на девайсе (с использование разделяемой памяти и без)

```
__global__ void bitonicSortShared(int * devArr, const uint32_t mergeStep, const uint32_t step, const uint32_t
size) {
    uint32_t idx = threadIdx.x;
    __shared__  int32_t devArrShared[512 * 8];
    for (uint32_t i = idx; i < size; i +=512 ) {
        devArrShared[i] = devArr[i];
    }
    __syncthreads();
    for (uint32_t n = idx; n < size; n += 512) {
        uint32_t nPlusStep = n ^ step;

        if (nPlusStep > n) {
            if (((n&mergeStep) == 0) && (devArrShared[n] > devArrShared[nPlusStep])) {
                int32_t tmp = devArrShared[n];
                devArrShared[n] = devArrShared[nPlusStep];
                devArrShared[nPlusStep] = tmp;

            }
            else if (((n&mergeStep) != 0) && (devArrShared[n] < devArrShared[nPlusStep])) {
                int32_t tmp = devArrShared[n];
                devArrShared[n] = devArrShared[nPlusStep];
                devArrShared[nPlusStep] = tmp;
            }
        }
    }
    __syncthreads();
    for (uint32_t i = idx; i < size; i += 512) {
        devArr[i] = devArrShared[i];
    }
    return;
}

__global__ void bitonicSort(int * devArr, const uint32_t mergeStep, const uint32_t step, const uint32_t size) {
    uint32_t idx = threadIdx.x + blockDim.x * blockIdx.x,
        offsetx = blockDim.x * gridDim.x;
    for (uint32_t n = idx; n < size; n += offsetx) {
        uint32_t nPlusStep = n ^ step;
        if (nPlusStep > n) {
            if (((n&mergeStep) == 0)&&(devArr[n] > devArr[nPlusStep])) {
                int32_t tmp = devArr[n];
                devArr[n] = devArr[nPlusStep];
                devArr[nPlusStep] = tmp;
```

```
            }
            else if (((n&mergeStep) != 0) && (devArr[n] < devArr[nPlusStep])) {
                int32_t tmp = devArr[n];
                devArr[n] = devArr[nPlusStep];
                devArr[nPlusStep] = tmp;
            }
        }
    }
    return;
}
```

Оценка производительности

Test size	CUDA <1, 512> (shared) TIME	CUDA <32, 512> TIME	CUDA <32, 32> TIME	CUDA <512, 512> TIME	CPU TIME
500	0.36	0.53	0.40	4.90	0.2
10000	-	2.16	3.88	12.32	90
100000	-	13.33	38.73	25.79	915

Вывод:

Битоническая сортировка это один из видов параллельных сортировок, которых представлено не так много, если сравнивать с последовательными алгоритмами. Не рекурсивная и параллельная реализация данной сортировки достаточно сложна в реализации.

Как видно из тестов, использование shared памяти не дало сильного прироста в производительности. К тому же её не получится использовать для сортировки больших массивов.

Для ускорения работы сортировки можно было бы добавить запуск с использованием разделяемой памяти при шаге алгоритма, совпадающем с размерностью shared массива, также можно было бы перенести основное тело сортировки в функцию выполняющуюся на девайсе.