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: И. П. Моисеенков
Группа: М8О-408Б-19
Преподаватель: 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 1. Битоническая сортировка**:

**Входные данные**. В первых четырех байтах записывается целое число n - длина массива чисел int, далее следуют n чисел.

**Выходные данные**. В бинарном виде записываются n отсортированных по возрастанию чисел.

## Программное и аппаратное обеспечение

В качестве графического процессора использую видеокарту Nvidia GeForce GT 545, установленную на сервере преподавателя.

```
Compute capability      : 2.1
Name                    : GeForce GT 545
Total Global Memory     : 3150381056
Shared memory per block : 49152
Registers per block     : 32768
Warp size               : 32
Max threads per block   : (1024, 1024, 64)
Max block   : (65535, 65535, 65535)
Total constant memory   : 65536
Multiprocessors count   : 3
```

В качестве редактора кода использовался Visual Studio Code.

## Метод решения

Битоническая последовательность - это последовательность чисел, которая сначала возрастает, а потом убывает.

Для битонической сортировки нужно выполнить последовательно несколько битонических слияний M(x). С помощью M(x) мы получим битонические последовательности длины x. Цель - получить битоническую последовательность длины, равной длине исходной последовательности, которая будет возрастать. Для этого нужно выполнить M(2), M(4), M(8), …, M(длина последовательности).

Каждое битоническое слияние M(x) - это последовательное применение полуочистителей B(x), B(x/2), B(x/4), …, B(2). Каждый полуочислитель B(a) выполняет упорядочивание двух элементов, находящихся на расстоянии a/2 друг от друга. Порядок упорядочивания (по возрастанию или по убыванию) определяется параметром слияния.

Битоническая сортировка работает только для последовательностей, длина которых равна степени двойки. Иначе нужно дополнить последовательность бексконечными по величине элементами. В отсортированном массиве эти элементы будут в самом конце, их выводить не будем.

Распараллелить можно выполнение полуочистителя. Сравнивать элементы можно независимо друг от друга.

Сложность сортировки с учетом распараллеливания - O(log(n) * log(n)).

## Описание программы

Для сортировки будем использовать два ядра. Первое будет выполнять сортировку в глобальной памяти, второе - в разделяемой.

Если нужно выполнить получиститель B(x), где x > 1024, то один такой полуочиститель будет выполняться за один вызов ядра в глобальной памяти. Если x <= 1024, то все полуочистители B(1024), B(512), …., B(2) выполнятся за один вызов в разделяемой памяти.

## Результаты

Рассмотрим время работы программы на различных тестах при различных размерах сетки (и без использования графического процессора вообще). В тестах будем выполнять сортировку массивов различной длины. Время считывания данных и печати результата не учитываем. Результаты приведены в таблице ниже.

| Размер сетки ядра \ Длина массива | 100к элементов, мс | 1кк элементов, мс | 10кк элементов,мс |
|---|---|---|---|
| CPU | 112,81 | 706,20 | 13237,70 |
| <<<(64, 512)>>> | 19,30 | 203,34 | 4390,13 |
| <<<(128, 512)>>> | 19,18 | 198,22 | 4264,75 |
| <<< (256, 512)>>> | 19,02 | 199,13 | 4258,96 |
| <<< (512, 512)>>> | 19,14 | 200,18 | 4234,81 |

Для корректной работы алгоритма на разделяемой памяти я не менял второй параметр сетки. Поэтому результаты получились примерно одинаковыми на различных параметрах ядра. Но в любом случае эти результаты в разы лучше, чем при сортировке на CPU.

Применим утилиту nvprof для исследования производительности программы. Применять будем на тесте с 100к элементами.

```
==14056== Profiling application: ./a.out
==14056== Profiling result:
==14056== Event result:
Invocations                Event Name         Min       Max       Avg
Device "GeForce GT 545 (0)"
        Kernel: kernel_shared(int*, int, int, int)
        8          divergent_branch       10532     18469     12316
        8  global_store_transaction        4032      4128      4080
        8   l1_shared_bank_conflict      105423    123837    116782
        8         l1_local_load_hit           0         0         0

        Kernel: kernel_global(int*, int, int, int)
        73          divergent_branch           1      7221      1591
        73  global_store_transaction        1080     12672      4851
        73   l1_shared_bank_conflict           0         0         0
        73         l1_local_load_hit           0         0         0
```

Параметр, отвечающий за обращение за конфликты при работе с разделяемой памятью - l1_shared_bank_conflict. В моей реализации присутствуют конфликты, т.к. различные потоки одного варпа обращаются в один банк памяти. Чтобы избежать конфликтов, нужно класть каждый элемент в свой банк памяти.

## Выводы

В данной лабораторной работе я реализовал параллельную битоническую сортировку на CUDA. Я реализовал два ее варианта - на глобальной и на разделяемой памяти. Реализация на глобальной памяти была тривиально. Но с разделяемой пришлось повозиться. Основная проблема была в ее ограниченном размере, приходилось придумывать нетривиальный алгоритмы. Но зато использование разделяемой памяти дает выигрыш по времени. Если получится избегать конфликтов банков памяти, то время работы программы еще снизится.