

**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: В. В. Бирюков  
Группа: 8О-407Б  
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.

**Вариант:** 6. Карманная сортировка с битонической сортировкой в каждом кармане.

## Программное и аппаратное обеспечение

Характеристики графического процессора:

- Наименование: GeForce GT 545
- Compute capability: 2.1
- Графическая память: 3150381056 Б
- Разделяемая память на блок: 49152 Б
- Количество регистров на блок: 32768
- Максимальное количество потоков на блок: (1024, 1024, 64)
- Максимальное количество блоков: (65535, 65535, 65535)
- Константная память: 65536 Б
- Количество мультипроцессоров: 3

Характеристики системы:

- Процессор: Intel(R) Core(TM) i7-3770 CPU 3.40GHz
- Память: 15 ГБ
- HDD: 500 ГБ

Программное обеспечение:

- ОС: Ubuntu 16.04.6 LTS
- IDE: Visual Studio Code 1.72
- Компилятор: nvcc 7.5.17

## Метод решения

Карманная сортировка сортирует массив путем распределения его по карманам, и сортировки каждого кармана по отдельности. Разделение по карманам осуществляется в два этапа: сначала массив распределяется по маленьким карманам (распределение происходит из предположения, что данные распределены равномерно), затем маленькие карманы объединяются в большие, к которым непосредственно применяется другой алгоритм сортировки. Размер большого кармана выбирается таким образом, чтобы его можно было отсортировать в разделяемой памяти. Если карман получился больше нужного размера, к нему рекурсивно применяется алгоритм с самого начала.

Для распределения элементов по карманам необходимо знать минимум и максимум в массиве, которые можно найти алгоритмом редукции. Перегруппировка массива в соответствии с карманами есть ни что иное, как модифицированная сортировка подсчетом, поэтому ее так же можно выполнить при помощи алгоритма гистограммы и алгоритма scan для суммы. Сортировка больших карманов осуществляется битонической сортировкой.

Сложность работы карманной сортировки в среднем линейная. Наилучшие результаты достигаются если числа в массиве равномерно распределены.

## Описание программы

Программа разбита на заголовочные файлы, соответствующие основным алгоритмам — редукции, scan, алгоритму гистограммы, битонической сортировки и карманной сорти-

ровки. Файл `utils.hpp` содержит дополнительные математические и вспомогательные функции.

Реализация редукции и `scan` позволяет использовать их вместе с любой бинарной функцией, однако в C++11 это не очень удобно делать, из-за отсутствия шаблонов переменных. Битоническая сортировка реализована полностью, для работы с массивом любой длины, хотя используется только для сортировки в разделяемой памяти.

Редукция использует все основные оптимизации, кроме развертки цикла; `scan` реализован без оптимизаций, кроме использования фиктивных элементов; алгоритм гистограммы всегда строит ее в глобальной памяти; битоническая сортировка оптимизирована для небольших массивов и не дополняет их до степени двойки в глобальной памяти — это значительно уменьшает время работы карманной сортировки.

## Результаты

### Анализ программы профилировщиком

Размер тестового файла:  $10^8$ , конфигурация ядер:  $128 \times 1024$ .

Profiling result:

| Time(%) | Time     | Calls | Avg      | Min      | Max      | Name                              |
|---------|----------|-------|----------|----------|----------|-----------------------------------|
| 78.63%  | 6.02814s | 99315 | 60.697us | 59.362us | 61.570us | void                              |
|         |          |       |          |          |          | bitonic_sort_shared_memory<float> |
| 9.05%   | 693.59ms | 1     | 693.59ms | 693.59ms | 693.59ms | void group<float>                 |
| 6.01%   | 460.67ms | 1     | 460.67ms | 460.67ms | 460.67ms | void histogram<uin32_t, uin32_t>  |
| 3.39%   | 260.19ms | 6     | 43.365ms | 2.6950us | 130.05ms | void reduce<float>                |
| 1.41%   | 108.41ms | 3     | 36.136ms | 22.140ms | 64.096ms | [CUDA memcpy HtoD]                |
| 0.88%   | 67.743ms | 7     | 9.6776ms | 2.8480us | 65.610ms | [CUDA memcpy DtoH]                |
| 0.44%   | 33.688ms | 1     | 33.688ms | 33.688ms | 33.688ms | void split<float>                 |
| 0.16%   | 12.610ms | 2     | 6.3050ms | 15.802us | 12.594ms | void scan<uin32_t>                |
| 0.01%   | 906.58us | 2     | 453.29us | 2.4790us | 904.10us | void per_block_func<uin32_t>      |
| 0.01%   | 516.76us | 2     | 258.38us | 2.3680us | 514.39us | [CUDA memset]                     |
| 0.00%   | 14.210us | 5     | 2.8420us | 2.6250us | 3.1680us | [CUDA memcpy DtoD]                |
| 0.00%   | 14.191us | 1     | 14.191us | 14.191us | 14.191us | void scan<uin32_t>                |

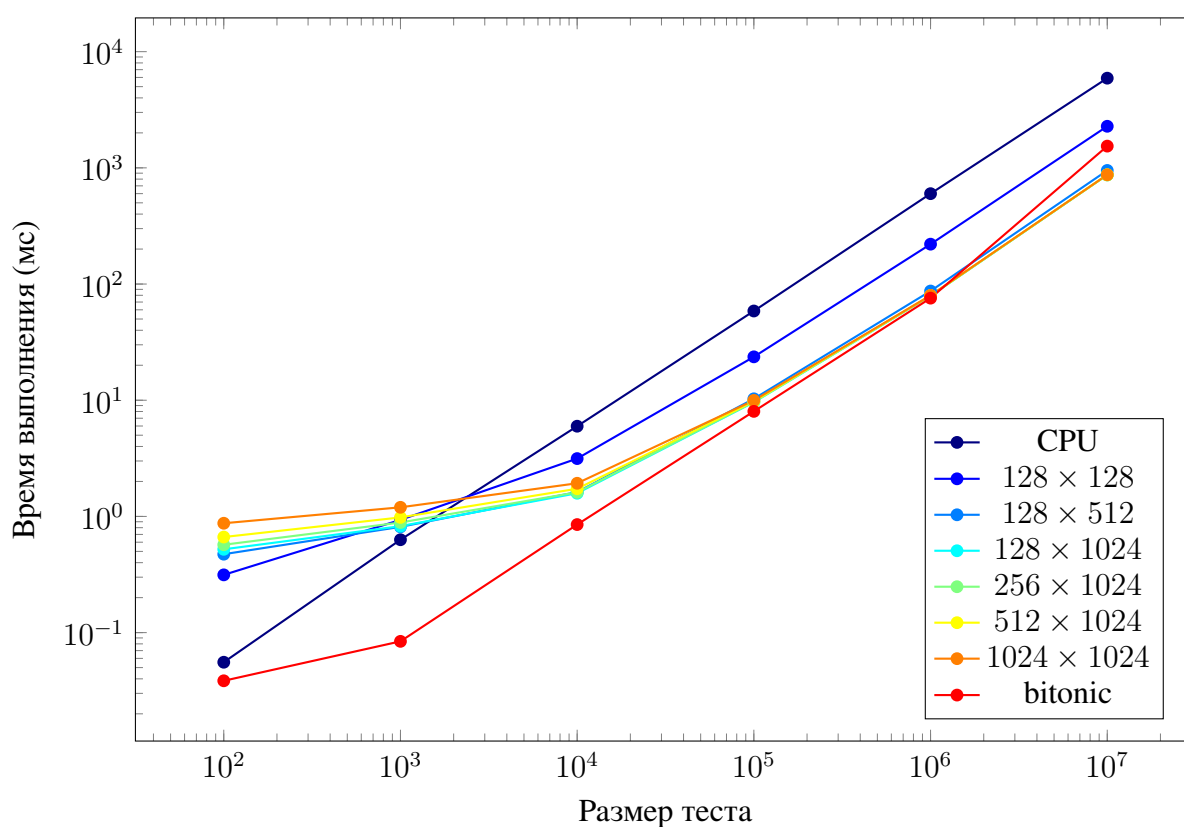
Больше всего ожидаемо выполняется битоническая сортировка, вызванная примерно  $\frac{10^8}{1024} = 97656$  раз.

| Invocations   | Event Name              | Min  | Max  | Avg  |
|---|-------------------------|------|------|------|
| Device "GeForce GT 545 (0)"   |                         |      |      |      |
| Kernel: void scan<uin32_t>(uin32_t*, uin32_t (*) (uin32_t, uin32_t))                          |                         |      |      |      |
| 1   | l1_shared_bank_conflict | 0    | 0    | 0    |
| 1   | divergent_branch        | 0    | 0    | 0    |
| Kernel: void per_block_func<uin32_t>(uin32_t*, uin32_t*, int, uin32_t (*) (uin32_t, uin32_t)) |                         |      |      |      |
| 2   | l1_shared_bank_conflict | 0    | 0    | 0    |
| 2   | divergent_branch        | 0    | 0    | 0    |
| Kernel: void histogram<uin32_t, uin32_t>(uin32_t*, int, uin32_t*)                             |                         |      |      |      |
| 1   | l1_shared_bank_conflict | 0    | 0    | 0    |
| 1   | divergent_branch        | 0    | 0    | 0    |
| Kernel: void group<float>(float*, float*, int, uin32_t*, uin32_t*)                            |                         |      |      |      |
| 1   | l1_shared_bank_conflict | 0    | 0    | 0    |
| 1   | divergent_branch        | 0    | 0    | 0    |
| Kernel: void reduce<float>(float*, int, float*, float (*) (float, float))                     |                         |      |      |      |
| 6   | l1_shared_bank_conflict | 0    | 0    | 0    |
| 6   | divergent_branch        | 0    | 0    | 0    |
| Kernel: void scan<uin32_t>(uin32_t*, int, uin32_t*, uin32_t (*) (uin32_t, uin32_t))           |                         |      |      |      |
| 2   | l1_shared_bank_conflict | 0    | 0    | 0    |
| 2   | divergent_branch        | 0    | 0    | 0    |
| Kernel: void bitonic_sort_shared_memory<float>(float*, int, int, float)                       |                         |      |      |      |
| 99315   | l1_shared_bank_conflict | 0    | 0    | 0    |
| 99315   | divergent_branch        | 1143 | 1499 | 1425 |
| Kernel: void split<float>(float*, uin32_t*, int, int, float, float)                           |                         |      |      |      |
| 1   | l1_shared_bank_conflict | 0    | 0    | 0    |
| 1   | divergent_branch        | 0    | 0    | 0    |

От конфликтов банков разделяемой памяти удалось избавиться во всех алгоритмах, где она используется. Дивергенция потоков наблюдается только в битонической сортировке, вероятно это связано с тем, что на каждой итерации активна только часть потоков.

## Сравнение времени работы

| Размер теста            | 100                  | 1000   | $10^4$ | $10^5$ | $10^6$  | $10^7$  | $10^8$   |
|-------------------------|----------------------|--------|--------|--------|---------|---------|----------|
| Конфигурация            | Время выполнения, мс |        |        |        |         |         |          |
| CPU                     | 0.0556               | 0.6301 | 5.9759 | 58.623 | 599.690 | 5913.16 | 65898.06 |
| $128 \times 128$        | 0.3141               | 0.9337 | 3.1465 | 23.631 | 220.344 | 2279.35 | 22634.16 |
| $128 \times 512$        | 0.4731               | 0.8130 | 1.6202 | 10.287 | 87.063  | 949.79  | 9346.23  |
| $128 \times 1024$       | 0.5211               | 0.8248 | 1.5834 | 9.617  | 79.065  | 869.78  | 8519.17  |
| $256 \times 1024$       | 0.5704               | 0.8861 | 1.6135 | 9.657  | 79.278  | 870.03  | 8534.99  |
| $512 \times 1024$       | 0.6647               | 0.9807 | 1.7309 | 9.779  | 79.598  | 871.51  | 8523.13  |
| $1024 \times 1024$      | 0.8734               | 1.1977 | 1.9291 | 9.970  | 79.859  | 872.14  | 8519.02  |
| Битоническая сортировка |                      |        |        |        |         |         |          |
| $128 \times 1024$       | 0.0385               | 0.0841 | 0.8510 | 8.019  | 75.615  | 1538.61 | 15307.46 |



Из замеров видно, что уменьшение числа потоков, от которого зависит количество элементов, обрабатываемых в разделяемой памяти, значительно увеличивает время работы. Слишком большое число потоков — тоже, но не так существенно, оптимальная конфигурация:  $128 \times 1024$ . Сортировка на CPU быстрее на маленьких тестах, но после 1000 элементов начинает сильно проигрывать. Так как битоническая сортировка была реализована полностью, я решил сравнить еще и с ней. На маленьких тестах она работает быстрее всего, но, начиная с  $10^7$  — в два раза медленнее карманной.

## **Выводы**

В ходе выполнения лабораторной работы я познакомился с параллельной реализацией некоторых линейных алгоритмов и сортировок. Распараллеливание таких алгоритмов дает очень хорошие результаты, однако реализовывать их гораздо сложнее.

Во время тестирования алгоритмов было очень хорошо заметно влияние различных оптимизаций, и то, как конфликты в разделяемой памяти или лишнее выделение памяти могут негативно сказаться на времени работы.