МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №4
по курсу «Параллельная обработка данных»

Сортировка чисел на GPU. Свертка, сканирование, гистограмма.

Выполнил: И.Д. Черненко
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

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: Вариант 4, Сортировка чет-нечет.

Входные данные: Входные данные. В первых четырех байтах записывается целое число n -- длина массива чисел, далее следуют n чисел типа заданного вариантом.

Выходные данные: В бинарном виде записывают n отсортированных по возрастанию чисел.

Требуется реализовать блочную сортировку чет-нечет для чисел типа int.
Должны быть реализованы:

- Алгоритм чет-нечет сортировки для предварительной сортировки блоков.
- Алгоритм битонического слияния, с использованием разделяемой памяти.

Ограничения: $n \leq 16 * 10^6$

Программное и аппаратное обеспечение

Compute capability : 6.1
Name : GeForce GTX 1050
Total Global Memory : 2096103424
Shared memory per block : 49152
Registers per block : 65536
Max threads per block : (1024, 1024, 64)
Max block : (2147483647, 65535, 65535)
Total constant memory : 65536
Multiprocessors count : 5

OS:Linux
compiler:nvcc
IDE:VS Code
Editor:nano

Метод решения

Сортировкой чет-нечет производится предварительная сортировка блоков. Потом производятся итерации блочной чет-нечет сортировки. Внутри каждой из таких частей производится попарное битоническое слияние двух соседних блоков.
Само битоническое слияние можно описать как последовательность обменов в определенном порядке.
Все операции проводятся с использованием разделяемой памяти, потом полученные результаты записываются в глобальную память и выводятся в стандартный поток вывода.

Описание программы

Вся программа была реализована в одном файле **main.cu**.

Реализованные ядра:

```
__global__ void full_device_array(int n, int value, int* device_array) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int offset = gridDim.x * blockDim.x;
    for(int i = idx; i < n; i += offset) {
        device_array[i] = value;
    }

}

__global__ void get_sorted(int* device_array) {
    __shared__ int shared_mem_array[blocks + (blocks / threads) + 1];
    int tmp1 = ((threads + 1) * (threadIdx.x / threads) + (threadIdx.x %
threads));
    shared_mem_array[((threads + 1) * (threadIdx.x / threads) + (threadIdx.x %
threads))] = device_array[threadIdx.x + blockIdx.x * blocks];
    int tmp2 =  ((threads + 1) * ((threadIdx.x + blocks / 2) / threads) +
((threadIdx.x + blocks / 2) % threads));
    shared_mem_array[tmp2] = device_array[threadIdx.x + blockIdx.x * blocks +
blocks / 2];

    if(threadIdx.x == 0)
        shared_mem_array[((threads + 1) * (blocks / threads) + (blocks %
threads))] = INT_MAX;

    __syncthreads();

    for(int i = 0; i < blocks; i++) {
        get_swap(((threads + 1) * (2 * threadIdx.x / threads) + (2 *
threadIdx.x % threads)), ((threads + 1) * ((2 * threadIdx.x + 1) / threads) +
((2 * threadIdx.x + 1) % threads)), shared_mem_array);
        __syncthreads();
        get_swap(((threads + 1) * ((2 * threadIdx.x + 1) / threads) + ((2 *
threadIdx.x + 1) % threads)), ((threads + 1) * ((2 * threadIdx.x + 2) /
threads) + ((2 * threadIdx.x + 2) % threads)), shared_mem_array);
        __syncthreads();
    }
    __syncthreads();

    device_array[threadIdx.x + blockIdx.x * blocks] = shared_mem_array[tmp1];

    device_array[threadIdx.x + blockIdx.x * blocks + blocks / 2] =
shared_mem_array[tmp2];
}
```

```
__global__ void get_merged(int* device_array) {
    __shared__ int shared_mem_array[blocks + (blocks / threads)];
    int tmp1 = ((threads + 1) * (threadIdx.x / threads) + (threadIdx.x %
threads));
    shared_mem_array[tmp1] = device_array[threadIdx.x + blockIdx.x * blocks];
    int tmp2 = ((threads + 1) * ((blocks - threadIdx.x - 1) / threads) +
((blocks - threadIdx.x - 1) % threads));
    shared_mem_array[tmp2] = device_array[threadIdx.x + blockIdx.x * blocks +
blocks / 2];
    __syncthreads();

    int cut = blocks / 2;
    do {
        int i = threadIdx.x / cut;
        int j = threadIdx.x % cut;
        __syncthreads();
        get_swap(((threads + 1) * ((2 * cut * i + j) / threads) + ((2 * cut *
i + j) % threads)), ((threads + 1) * ((2 * cut * i + j + cut) / threads) + ((2
* cut * i + j + cut) % threads)), shared_mem_array);
        cut /= 2;
    } while (cut > 0);
    __syncthreads();

    device_array[threadIdx.x + blockIdx.x * blocks] = shared_mem_array[tmp1];
    int tmp3 = ((threads + 1) * ((threadIdx.x + blocks / 2) / threads) +
((threadIdx.x + blocks / 2) % threads));
    device_array[threadIdx.x + blockIdx.x * blocks + blocks / 2] =
shared_mem_array[tmp3];
}
```

Вспомогательные функции:

```
__device__ void get_swap(int indx1, int indx2, int* array) {
    int value1 = array[indx1];
    int value2 = array[indx2];
    if (value1 > value2) {
        array[indx2] = value1;
        array[indx1] = value2;
    }
}
```

Исследование производительности программы с помощью утилиты Nvprof

```
==24817== Profiling application: ./a.out
==24817== Profiling result:
==24817== Event result:
Invocations                                Event Name         Min         Max         Avg
Device "GeForce GT 545 (0)"
        Kernel: get_merged(int*)
          20                            divergent_branch           0           0           0
          20                   global_store_transaction         288         384         312
          20                    ll_shared_bank_conflict        1728        2574        2091
          20                         ll_local_load_hit           0           0           0
        Kernel: get_sorted(int*)
           1                            divergent_branch           0           0           0
           1                   global_store_transaction         384         384         384
           1                    ll_shared_bank_conflict      239166      239166      239166
           1                         ll_local_load_hit           0           0           0
        Kernel: full_device_array(int, int, int*)
           1                            divergent_branch           1           1           1
           1                   global_store_transaction          45          45          45
           1                    ll_shared_bank_conflict           0           0           0
           1                         ll_local_load_hit           0           0           0
```

Как можно заметить не удалось избавиться от всех конфликтов памяти, но ситуация не критическая, вероятнее всего конфликты возникают при свапе когда в один блок попадают несколько потоков варпа.

Результаты

Всё время в ms

block	threads	1000	10000	100000	1000000
512	32	0.51248	1.29859	39.2738	3341.84
512	64	0.569248	1.51104	41.356	3413.25
512	128	0.555616	1.51725	41.3339	3410.39
1024	32	0.653728	1.71546	29.2746	2021.62
1024	64	0.854176	2.13133	32.7777	2097.18
1024	128	0.902368	2.17331	32.6285	2077.65
CPU		0.4	125	6453.125	610546.875

Из полученных данных можно сделать вывод, что при правильных настройках вычисления на **GPU** превосходят **CPU** примерно в **300 раз**, что особенно заметно на больших данных**.**

Выводы

Данный алгоритм можно было бы использовать в задачах для сортировки больших данных. Сложнее всего было достичь некритического количества конфликтов банков памяти. Но по итогу полученная реализация намного превзошла вычисления на CPU, что очевидно так как сама задача довольно неплохо распараллеливается на GPU.