МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №5
по курсу «Программирование графических процессоров»

Сортировка чисел на GPU. Свертка, сканирование, гистограмма.

Выполнил: *И. П. Попов*
Группа: *8О-406Б*
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2023

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 (обязательно отразить в отчете).

Вариант 5. Сортировка чет-нечет с предварительной битонической сортировкой.
Требуется реализовать блочную сортировку чет-нечет для чисел типа int.
Должны быть реализованы:
● Алгоритм битонической сортировки для предварительной сортировки блоков.
● Алгоритм битонического слияния, с использованием разделяемой памяти.
Ограничения: $n \leq 16 * 10^6$

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.

Программное и аппаратное обеспечение

NVIDIA GeForce GTX 1660 Super:
Compute capability: 7.5
Dedicated video memory: Typically 6 GB (may vary by manufacturer)
Shared memory per block: 49152 bytes
Register per block: 65536 bytes
Total constant memory: 65536 bytes
Max threads per multiprocessor: 2048
Max threads per block: 1024

CPU AMD Ryzen 3600X
Physical cores: 6
Threads: 12
Base clock speed: 3.8 GHz
Boost clock speed: 4.4 GHz
L1 cache: 384KB (per core)
L2 cache: 512KB (per core)
L3 cache: 32 MB (shared)
Chip lithography: 7 nm

16 Гб RAM
1 ТБ HDD
OS – Windows 11 ProWSL, IDE – VS Code, compiler - nvcc

Метод решения

Программа начинается с выделения памяти на графическом процессоре, приводя ее размер к ближайшей степени двойки. После этого следует первый этап - битоническая сортировка слиянием. Каждый блок массива размером до 1024 элементов сортируется в цикле.

Затем следует второй этап, включающий два прохода. Первый проход - четный, где элементы переставляются в порядке возрастания. Каждый элемент помещается в свой поток с использованием разделяемой памяти для оптимизации второго прохода. Второй проход представляет собой слияние блоков с последующим разбиением.

Завершающий этап включает в себя возвращение данных с графического процессора и вывод их в бинарном виде в стандартный поток вывода.

Описание программы

1. Битоническая сортировка (bitonicSortStep):

   Используется для предварительной сортировки блоков массива.
   Каждый блок размером BLOCK_SIZE сортируется в соответствии с битонической сортировкой.

```
__global__ void bitonicSortStep(int* nums, int j, int k, int
size) {
    __shared__ int shArray[BLOCK_SIZE];

    int* tmp = nums;
    unsigned int i = threadIdx.x;
    int idBlock = blockIdx.x;
    int offset = gridDim.x;

     for (int shift = idBlock * BLOCK_SIZE; shift < size; shift +=
offset * BLOCK_SIZE) {
        tmp = nums + shift;
        shArray[i] = tmp[i];

        __syncthreads();
```

```
        for (j = k / 2; j > 0; j /= 2) {
            unsigned int XOR = i ^ j;
               if ((XOR > i) && (((i & k) != 0) ? (shArray[i] <
shArray[XOR]) : (shArray[i] > shArray[XOR])))
                thrust::swap(shArray[i], shArray[XOR]);

            __syncthreads();
        }

        tmp[i] = shArray[i];
    }
}
```

2. Чет-нечет сортировка (performBitonicSort):

   Использует битоническую сортировку для каждого блока массива.
   Итерируется по разрядам и блокам для предварительной сортировки.

```
void performBitonicSort(int* devData, int updSize) {
    // Итерация по разрядам в битонической сортировке
    for (int k = 2; k <= updSize; k *= 2) {
        if (k > BLOCK_SIZE)
            break;
        // битоническая сортировка для каждого разряда
        for (int j = k / 2; j > 0; j /= 2) {
             bitonicSortStep << <BLOCKS, BLOCK_SIZE >> > (devData,
j, k, updSize);
            CSC(cudaGetLastError());
        }
    }
}
```

3. Чет-нечет сортировка (kernelBitonicMergeStep):

   Вызывается для битонического слияния на каждом этапе.
   Использует разделяемую память для слияния блоков с учетом четности и нечетности этапа.

```
__global__ void kernelBitonicMergeStep(int* nums, int size, bool
isOdd, bool flag) {
    int* tmp = nums;
    unsigned int i = threadIdx.x;
    int idBlock = blockIdx.x;
```

```
    int offset = gridDim.x;

     isOdd ? bitonicMergeStep(nums, tmp, size, (BLOCK_SIZE / 2) +
idBlock * BLOCK_SIZE, size - BLOCK_SIZE, offset * BLOCK_SIZE, i)
:
          bitonicMergeStep(nums, tmp, size, idBlock * BLOCK_SIZE,
size, offset * BLOCK_SIZE, i);
}
```

Результаты

Все измерения представлены в ms

	1000	1000000
<<<32, 32>>>	0.988288	32577.581482
<<<64, 64 >>>	0.373623	22167.226216
<<<128, 128>>>	0.299921	9255.864075
<<<512, 512 >>>	0.123048	2462.232357
<<<1024,1024 >>>	0.285925	1673.473263
CPU	0.305211	404515.837495

Результат nvprof

```
==31856== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   47.72%  25.729us        10  2.5720us  2.4320us  2.7530us  bitonicSortStep(int*, int, int, int)
                   46.94%  25.313us        10  2.5310us  2.4960us  2.5920us  kernelBitonicMergeStep(int*, int, bool, bool)
                    2.73%  1.4720us         1  1.4720us  1.4720us  1.4720us  [CUDA memcpy HtoD]
                    2.61%  1.4080us         1  1.4080us  1.4080us  1.4080us  [CUDA memcpy DtoH]
      API calls:   81.65%  130.45ms         1  130.45ms  130.45ms  130.45ms  cudaMalloc
                   17.27%  27.600ms         1  27.600ms  27.600ms  27.600ms  cuDevicePrimaryCtxRelease
                    0.65%  1.0412ms        20  52.060us  6.9000us  808.00us  cudaLaunchKernel
                    0.10%  160.00us        20  8.0000us  5.3000us  15.000us  cudaEventRecord
                    0.09%  140.30us         1  140.30us  140.30us  140.30us  cuModuleUnload
                    0.08%  135.80us         1  135.80us  135.80us  135.80us  cudaFree
                    0.06%  88.600us        10  8.8600us  7.9000us  13.000us  cudaEventSynchronize
                    0.04%  63.000us         2  31.500us  31.300us  31.700us  cudaMemcpy
                    0.02%  26.600us        20  1.3300us     600ns  8.2000us  cudaEventCreate
                    0.02%  25.000us        20  1.2500us     600ns  3.1000us  cudaEventDestroy
                    0.01%  16.300us       101     161ns     100ns  1.1000us  cuDeviceGetAttribute
                    0.01%  10.800us        10  1.0800us     700ns  2.7000us  cudaEventElapsedTime
                    0.00%  4.7000us         3  1.5660us     200ns  4.2000us  cuDeviceGetCount
                    0.00%  3.6000us        20     180ns     100ns     300ns  cudaGetLastError
                    0.00%  2.2000us         2  1.1000us     100ns  2.1000us  cuDeviceGet
                    0.00%  2.0000us         1  2.0000us  2.0000us  2.0000us  cuModuleGetLoadingMode
                    0.00%     700ns         1     700ns     700ns     700ns  cuDeviceGetName
                    0.00%     300ns         1     300ns     300ns     300ns  cuDeviceTotalMem
                    0.00%     300ns         1     300ns     300ns     300ns  cuDeviceGetLuid
                    0.00%     200ns         1     200ns     200ns     200ns  cuDeviceGetUuid
```

Некоторые выводы, которые можно сделать исходя из результатов профилирования

- Основная часть времени тратится на выполнение ядер bitonicSortStep и kernelBitonicMergeStep, что соответствует ожидаемому поведению сортировочного алгоритма.
- Значительное время тратится на выделение памяти на GPU (cudaMalloc), что может свидетельствовать о необходимости оптимизации использования памяти.
- Передача данных между хостом и устройством также занимает значительное время, что может подсказывать о возможности оптимизации работы с памятью.

Выводы

В ходе выполнения этой лабораторной работы я освоил и научился реализовывать параллельную сортировку чет-нечет при помощи классических алгоритмов, применяемых в области обработки данных на графическом процессоре (GPU). Работа с CUDA позволила ускорить процесс сортировки благодаря параллельному выполнению сортировки блоков, что избегает взаимных мешающих воздействий.