МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №*1*
по курсу «Параллельная обработка данных»

Сортировка чисел на GPU. Свертка, сканирование, гистограмма.

Выполнил: *Чистяков К.С.*
Группа: *8О-406Б-21*
Преподаватель: А.Ю. Морозов

Москва, 2025

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. Исследование производительности программы с помощью утилиты nvprof.
2. **Вариант задания.** Вариант 3. Сортировка подсчетом. Диапазон от 0 до 255. Требуется реализовать сортировку подсчетом для чисел типа uchar.
   Должны быть реализованы:
   - Алгоритм гистограммы, с использованием атомарных операций и разделяемой памяти.
   - Алгоритм сканирования, с бесконфликтным использованием разделяемой памяти.

   Ограничения: $n \leq 537 * 10^6$

Все входные-выходные данные являются бинарными и считываются из stdin и выводятся в stdout.
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.

Пример:

Входной файл (stdin), hex	Выходной файл (stdout), hex
0A 00 00 00 01 02 03 01 02 03 01 02 03 04	01 01 01 02 02 02 03 03 03 04

Программное и аппаратное обеспечение

Графический процессор (Google Colab)

Compute capability	7.5
Name	Tesla T4
Total Global Memory	15828320256
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	6553
Multiprocessors count	40

Процессор AMD Ryzen 5 4500U with Radeon Graphics (2.38 GHz)

Technology	7 nm
Cores	6
Threads	6
Core Speed	1390 MHz
Cache (L1 Data)	6 x 32 KBytes (8-way)
Cache (L1 Inst.)	6 x 32 KBytes (8-way)
Cache (Level 2)	6 x 512 KBytes (8-way)
Cache (Level 3)	2 x 4 MBytes (16-way)

Оперативная память

Number of modules	2
Module Manuf.	Samsung
Module Size	8 GBytes (total - 16 GBytes)
Generation	DDR4

Жесткий диск - 512 ГБ SSD
OS - Windows 10 Домашняя + Linux
IDE - VS Code + Google Colab
Compiler - nvcc, gcc

Метод решения

В данной лабораторной работе реализуется алгоритм основанный на сортировке подсчетом. В начале мы проходимся по массиву и вычисляем его гистограмму, то есть частоту появления каждого из значений в пределах от 0 до 255. Далее мы используем суммирование, чтобы для каждого элемента можно было определить его окончательную позицию в отсортированном массиве. Потом используя эти данные мы

перемещаем каждый элемент из исходного массива в его конечную позицию в отсортированном массиве.

Описание программы

Вся программа описана в одном файле.

- CSC(call) - функция для отлова ошибок CUDA
- __global__ void compute_histogram(unsigned char* d_input, int* d_histogram, int n) - функция для вычисления гистограммы
- __global__ void compute_sum(int* d_histogram, int* d_sum) - функция для суммирования
- __global__ void kernel_sort(unsigned char* d_input, unsigned char* d_output, int* d_sum, int n) - функция сортировки, которая расставляет элементы на их конечные позиции в отсортированном массиве
- int main() - главная функция, в которой происходит считывание входных данных, выделение памяти, копирование данных на GPU, вызов основных функций, копирование данных обратно на хост, освобождение памяти и вывод результата.

Ядро для вычисления гистограммы:

```
__global__ void compute_histogram(unsigned char* d_input, int* d_histogram, int n) {
  __shared__ int hist[256];
  if (threadIdx.x < 256) {
    hist[threadIdx.x] = 0;
  }
  __syncthreads();

  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int offsetx = gridDim.x * blockDim.x;

  for (int i = idx; i < n; i += offsetx) {
    atomicAdd(&hist[d_input[i]], 1);
  }
  __syncthreads();

  if (threadIdx.x < 256) {
    atomicAdd(&d_histogram[threadIdx.x], hist[threadIdx.x]);
  }
}
```

Ядро для суммирования:

```
__global__ void compute_sum(int* d_histogram, int* d_sum) {
  __shared__ int shared_memory[256];
  shared_memory[threadIdx.x] = d_histogram[threadIdx.x];
```

```
  __syncthreads();

  for (int j = 1; j < 256; j *= 2) {
    int i = (threadIdx.x + 1) * j * 2 - 1;
    if (i < 256) {
      shared_memory[i] += shared_memory[i - j];
    }
    __syncthreads();
  }

  if (threadIdx.x == 0) shared_memory[256 - 1] = 0;
  __syncthreads();

  for (int j = 256 / 2; j > 0; j /= 2) {
    int i = (threadIdx.x + 1) * j * 2 - 1;
    if (i < 256) {
      int temp = shared_memory[i - j];
      shared_memory[i - j] = shared_memory[i];
      shared_memory[i] += temp;
    }
    __syncthreads();
  }

  d_sum[threadIdx.x] = shared_memory[threadIdx.x];
  __syncthreads();
}
```

Ядро для сортировки:

```
__global__ void kernel_sort(unsigned char* d_input, unsigned char* d_output, int* d_sum,
int n) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int offsetx = gridDim.x * blockDim.x;
  for (int i = idx; i < n; i += offsetx) {
    d_output[atomicAdd(&d_sum[d_input[i]], 1)] = d_input[i];
  }
}
```

Результаты

Замеры времени работы ядер с различными конфигурациями для compute_histogram и kernel_sort, а compute_sum трогать не будем, так как увеличение числа блоков или потоков прироста производительности не принесёт (гистограмма содержит 256 элементов, и каждый поток в блоке выполняет операцию над одним элементом). Тесты будут следующих размеров: n = 100, n = 1000000, n = 10000000, n = 50000000. А также будет приведён замер работы аналогичной программы без использования технологии CUDA - на CPU.

1. **Конфигурация <<<1, 32>>>**

n	Время (в мс)
100	0.183328
1000000	43.566593
10000000	369.425507
50000000	1313.187500

2. **Конфигурация <<<256, 256>>>**

n	Время (в мс)
100	0.238048
1000000	0.809280
10000000	6.939520
50000000	34.227776

3. **Конфигурация <<<512, 512>>>**

n	Время (в мс)
100	0.214656
1000000	0.845312
10000000	6.847360
50000000	33.469250

4. **Конфигурация <<<1024, 1024>>>**

n	Время (в мс)
100	0.221184
1000000	0.839520
10000000	6.863680
50000000	33.433407

5. **CPU**

n	Время (в мс)
100	0,001
1000000	8,845
10000000	59,175
50000000	276,365

Использование nvprof

Для теста с n = 10000000 и конфигурации <<< 512, 512>>> :

```
==16543== NVPROF is profiling process 16543, command: ./my_program
==16543== Profiling application: ./my_program
==16543== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   44.11%  6.4846ms         1  6.4846ms  6.4846ms  6.4846ms  kernel_sort(unsigned char*, unsigned char*, int*, int)
                   40.84%  6.0043ms         1  6.0043ms  6.0043ms  6.0043ms  [CUDA memcpy DtoH]
                   13.70%  2.0136ms         1  2.0136ms  2.0136ms  2.0136ms  [CUDA memcpy HtoD]
                    1.31%  192.38us         1  192.38us  192.38us  192.38us  compute_histogram(unsigned char*, int*, int)
                    0.04%  5.7920us         1  5.7920us  5.7920us  5.7920us  compute_sum(int*, int*)
                    0.01%  1.1520us         1  1.1520us  1.1520us  1.1520us  [CUDA memset]
      API calls:   84.33%  94.167ms         4  23.542ms  3.0240us  93.973ms  cudaMalloc
                    8.35%  9.3275ms         2  4.6637ms  2.2070ms  7.1205ms  cudaMemcpy
                    5.99%  6.6912ms         3  2.2304ms  7.4410us  6.4880ms  cudaDeviceSynchronize
                    0.92%  1.0297ms         4  257.42us  4.3110us  453.37us  cudaFree
                    0.19%  215.37us         3  71.790us  9.3290us  193.02us  cudaLaunchKernel
                    0.17%  190.61us       114  1.6720us     104ns  85.433us  cuDeviceGetAttribute
                    0.01%  15.649us         1  15.649us  15.649us  15.649us  cudaMemset
                    0.01%  13.580us         1  13.580us  13.580us  13.580us  cuDeviceGetName
                    0.01%  5.8230us         1  5.8230us  5.8230us  5.8230us  cuDeviceGetPCIBusId
                    0.00%  1.7290us         3     576ns     146ns  1.2790us  cuDeviceGetCount
                    0.00%  1.0080us         3     336ns     190ns     604ns  cudaGetLastError
                    0.00%     946ns         1     946ns     946ns     946ns  cuDeviceTotalMem
                    0.00%     866ns         2     433ns     172ns     694ns  cuDeviceGet
                    0.00%     518ns         1     518ns     518ns     518ns  cuModuleGetLoadingMode
                    0.00%     320ns         1     320ns     320ns     320ns  cuDeviceGetUuid
```

Выводы

В ходе выполнения лабораторной работы я ознакомился с фундаментальными алгоритмами GPU: свертка (reduce), сканирование (blelloch scan) и гистограмма (histogram). Реализовал сортировку подсчётом на CUDA. Использовал при этом

разделяемую память. Исследовал производительность программы с помощью утилиты nvprof.

Этот алгоритм может применяться в области анализа данных и статистики (сортировка больших массивов данных с ограниченным диапазоном), в области кибербезопасности и криптографии (анализ распределения символов в шифротексте, поиск аномалий в сетевом трафике), в области обработки сигналов (анализ звуковых сигналов) и т.п.

Написание этой программы не вызвало особых сложностей, так как на видеозаписи занятия посвященного сортировкам было достаточно понятно изложена последовательность необходимых действий для реализации лабораторной работы.

Использование nvprof помогло проанализировать программу и корректировать некоторые моменты для лучшей оптимизации.

Сравнивая время работы ядер с различными конфигурациями и размерами массива, а также аналогичной программы без использования технологии CUDA - на CPU, можно заметить, что программа на CPU выигрывает у всех конфигураций на GPU на относительно маленьких тестах. Однако на больших тестах уже существенно отстаёт от всех конфигураций, кроме самой маленькой (она проигрывает CPU во всех тестах). Конфигурации с числом блоков/потоков от 256 и до 1024 практически не отличаются по показателям времени (может быть на более крупных тестах разница была бы виднее, но создание массива бинарных чисел размером более 50 млн. занимает у меня очень много времени). Интересно, что самая маленькая конфигурация выигрывает по времени у больших на относительно маленьком тесте. В целом можно сказать, что конфигурации на GPU с достаточным количеством потоков всегда выигрывают по скорости у CPU, причём значительно.