Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №5 по курсу «Программирование графических процессоров»
(Лабораторная работа №4 по курсу «Параллельная обработка данных»)

Сортировка чисел на GPU. Свертка, сканирование, гистограмма.

Студент: Л. Я. Вельтман
Преподаватель: К. Г. Крашенинников
А. Ю. Морозов
Группа: М8О-407Б
Дата:
Оценка:
Подпись:

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 (обязательно отразить в отчете).

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.

Программное и аппаратное обеспечение:
Device Number: 0
Device name: GeForce GT 545
TotalGlobalMem: 3150381056
Const Mem : 65536
Max shared mem for blocks 49152
Max regs per block 32768
Max thread per block 1024

multiProcessorCount : 3
maxThreadsDim 1024 1024 64
maxGridSize 65535 65535 65535
OS: macOS Catalina version 10.15.5
Text Editor: Sublime Text 3

1 Описание

Метод решения и описание программы
Алгоритм сортировки подсчетом:

- Считаем сколько раз в поседовательности встречается значение из заданного диапазона.
- Вычисление префиксных сумм.
- С помощью полученной префиксной суммы получаем отсортированную последовательность.

Алгоритм гистограммы нужно применить с испоьзованием атомарных операций и разделяемой памяти. Для этого выделяем память, отмечая, что она разделяемая $\_\_shared\_\_$, размером 256 элементов, так как диапазон возможных значений состоит из 256 символов. Мы перебираем все элементы буфера данных, пока наш абсолютный идентификатор не коснется значения size - количество элементов последовательности. С помощью атомарной операции atomicAdd извлекаем значение, находящееся в буфере, и увеличиваем счетчик в массиве разделяемой памяти tmp. После окончания подсчетов, наконец, обновляем окончательный результат и сохраняем его в массив histo, где и будут храниться значения после работы алгоритма гистограммы.

Алгоритм сканирования нужно применить с бесконфликтным использованием разделяемой памяти. Будем использовать алгоритм, который часто возникает при параллельных вычислениях: сбалансированные деревья. Идея состоит в том, чтобы построить сбалансированное двоичное дерево на основе входных данных и развернуть его к корню и от него, чтобы вычислить сумму префиксов. Бинарное дерево с n листьями имеет $lvl = log_2(n)$ уровней, и каждый уровень lvl имеет 2^d узлов. Если мы выполним одно добавление для каждого узла, то мы выполним O(n) добавлений при одном обходе дерева.

Конфликтов банков можно избежать в большинстве вычислений CUDA, если соблюдать осторожность при доступе к массивам памяти $\_\_shared\_\_$. Можно избежать большинства конфликтов банков при сканировании, добавляя переменную величину к каждому вычисляемому индексу массива разделяемой памяти. Для этого используется макрос во избежание конфликтов банка разделяемой памяти.

```
1 #define AVOID_BANK_CONFLICTS(idx) ((idx) >> BANKS + (idx) >> (LOG_2_BANKS << 1u))
```

Остается заполнить результирующие данные в отсортированном порядке. В качестве эталона используется этот последовательный цикл, но для параллельной версии будем использовать атомарную операцию atomicSub. Таким образом, гарантируется, что операция выполнится без вмешательства других потоков.

```
// pseudocode
// etalone algo
    for(i = 0; i < size; ++i) {
        res[prefixSum[data[i]] - 1] = data[i];
        --prefixSum[data[i]];
    }
```

2 Исходный код

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <cuda.h>
#include <sys/time.h>
#include <chrono>


#define BLOCK_COUNT 256u
#define HALF_BLOCK_COUNT 128u
#define BANKS 16
#define LOG_2_BANKS 4
// macro used for computing
// Bank-Conflict-Free Shared Memory Array Indices
#define AVOID_BANK_CONFLICTS(idx) ((idx) >> BANKS + (idx) >> (LOG_2_BANKS << 1))

#define CSC(call) do { \
    cudaError_t res = call; \
    if (res != cudaSuccess) { \
        fprintf(stderr, "CUDA Error in %s:%d: %s\n", __FILE__, __LINE__,
            cudaGetErrorString(res)); \
        exit(0); \
    } \
} while (0)


__global__ void Histogram(unsigned char* data, int size, int* histo)
{
    __shared__ int tmp[BLOCK_COUNT];

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

    tmp[threadIdx.x] = 0;

    __syncthreads();

    while (idx < size)
    {
        atomicAdd(&tmp[data[idx]], 1);
        idx += offset;
    }
    __syncthreads();

    int i = threadIdx.x;
    while (i < BLOCK_COUNT)
    {
```

```
47          atomicAdd(&histo[i], tmp[i]);
48          i += blockDim.x;
49      }
50  }
51
52
53  __global__ void Scan(int* histo, int* prefixSum)
54  {
55      __shared__ int tmp[BLOCK_COUNT];
56
57      int threadId = threadIdx.x;
58      int offset = 1;
59
60      int aIdx = threadIdx.x;
61      int bIdx = threadIdx.x + HALF_BLOCK_COUNT;
62
63      int bankOffsetA = AVOID_BANK_CONFLICTS(aIdx);
64      int bankOffsetB = AVOID_BANK_CONFLICTS(bIdx);
65
66      tmp[aIdx + bankOffsetA] = histo[aIdx];
67      tmp[bIdx + bankOffsetB] = histo[bIdx];
68
69      {
70          int lvl = BLOCK_COUNT >> 1;
71
72          while (lvl > 0)
73          {
74              __syncthreads();
75
76              if (threadId < lvl)
77              {
78                  int aIndex = (offset * (threadId * 2 + 1) - 1);
79                  int bIndex = (offset * (threadId * 2 + 2) - 1);
80                  aIndex += AVOID_BANK_CONFLICTS(aIndex);
81                  bIndex += AVOID_BANK_CONFLICTS(bIndex);
82                  tmp[bIndex] += tmp[aIndex];
83              }
84              offset <<= 1;
85              lvl >>= 1;
86          }
87      }
88
89      if (threadId == 0)
90      {
91          tmp[BLOCK_COUNT - 1 + AVOID_BANK_CONFLICTS(BLOCK_COUNT - 1)] = 0;
92      }
93
94      {
95          int lvl = 1;
```

```
96          while (lvl < BLOCK_COUNT)
97          {
98              offset >>= 1;
99              __syncthreads();
100             if (threadId < lvl)
101             {
102                 int aIndex = (offset * (threadId * 2 + 1) - 1);
103                 int bIndex = (offset * (threadId * 2 + 2) - 1);
104                 aIndex += AVOID_BANK_CONFLICTS(aIndex);
105                 bIndex += AVOID_BANK_CONFLICTS(bIndex);
106                 int temp = tmp[aIndex];
107                 tmp[aIndex] = tmp[bIndex];
108                 tmp[bIndex] += temp;
109             }
110             lvl <<= 1;
111         }
112     }
113
114     __syncthreads();
115
116     prefixSum[aIdx] = histo[aIdx] + tmp[aIdx + bankOffsetA];
117     prefixSum[bIdx] = histo[bIdx] + tmp[bIdx + bankOffsetB];
118 }
119
120
121 __global__ void CountSort(unsigned char* data, int* prefixSum, unsigned char* result,
        int size)
122 {
123     int idx = blockDim.x * blockIdx.x + threadIdx.x;
124     int offset = gridDim.x * blockDim.x;
125
126     int i = idx, j;
127     while (i < size)
128     {
129         j = atomicSub(&prefixSum[data[i]], 1) - 1;
130         result[j] = data[i];
131         i += offset;
132     }
133 }
134
135
136
137 int main()
138 {
139     int size;
140
141     freopen(NULL, "rb", stdin);
142     fread(&size, sizeof(int), 1, stdin);
143
```

```
144     unsigned char* data = new unsigned char[size];
145
146     fread(data, sizeof(unsigned char), size, stdin);
147     fclose(stdin);
148
149     unsigned char* deviceData;
150     unsigned char* deviceResult;
151     int* deviceHisto;
152     int* devicePrefix;
153
154     CSC(cudaMalloc((void**)&deviceData, sizeof(unsigned char) * size));
155     CSC(cudaMemcpy(deviceData, data, sizeof(unsigned char) * size,
            cudaMemcpyHostToDevice));
156
157     CSC(cudaMalloc((void**)&deviceHisto, sizeof(int) * BLOCK_COUNT));
158     CSC(cudaMalloc((void**)&devicePrefix, sizeof(int) * BLOCK_COUNT));
159     CSC(cudaMemset(deviceHisto, 0, sizeof(int) * BLOCK_COUNT));
160
161     CSC(cudaMalloc((void**)&deviceResult, sizeof(unsigned char) * size));
162
163     Histogram<<<BLOCK_COUNT, BLOCK_COUNT>>>(deviceData, size, deviceHisto);
164     cudaThreadSynchronize(); // wait end
165     CSC(cudaGetLastError());
166
167     Scan<<<1, HALF_BLOCK_COUNT>>>(deviceHisto, devicePrefix);
168     cudaThreadSynchronize(); // wait end
169     CSC(cudaGetLastError());
170
171     CountSort<<<1, BLOCK_COUNT>>>(deviceData, devicePrefix, deviceResult, size);
172     cudaThreadSynchronize(); // wait end
173     CSC(cudaGetLastError());
174
175     CSC(cudaMemcpy(data, deviceResult, sizeof(unsigned char) * size,
            cudaMemcpyDeviceToHost));
176
177     freopen(NULL, "wb", stdout);
178     fwrite(data, sizeof(unsigned char), size, stdout);
179     fclose(stdout);
180
181     CSC(cudaFree(deviceData));
182     CSC(cudaFree(deviceHisto));
183     CSC(cudaFree(devicePrefix));
184     CSC(cudaFree(deviceResult));
185
186     delete[] data;
187     return 0;
188 }
```

3 Результаты

```
user25@server-i72:~/PGP/lab5$ nvprof -e divergent_branch,global_store_transaction,l1_shared_bank_conflict,l1_l
ocal_load_hit -m sm_efficiency ./run < test > output
==21129== NVPROF is profiling process 21129, command: ./run
==21129== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==21129== Replaying kernel "Histogram(unsigned char*, int, int*)" (done)
==21129== Replaying kernel "Scan(int*, int*)" (done)
==21129== Replaying kernel "CountSort(unsigned char*, int*, unsigned char*, int)" (done)
==21129== Warning: The following aggregate event values were extrapolated from limited profile data and may th
erefore be inaccurate. To see the non-aggregate event values, use "--aggregate-mode off".
l1_local_load_hit,l1_shared_bank_conflict,global_store_transaction
==21129== Profiling application: ./run
==21129== Profiling result:
==21129== Event result:
Invocations                                Event Name         Min         Max         Avg
Device "GeForce GT 545 (0)"
        Kernel: Scan(int*, int*)
          1                            divergent_branch           0           0           0
          1                   global_store_transaction          24          24          24
          1                    l1_shared_bank_conflict         840         840         840
          1                          l1_local_load_hit           0           0           0
        Kernel: CountSort(unsigned char*, int*, unsigned char*, int)
          1                            divergent_branch           0           0           0
          1                   global_store_transaction  1518005466  1518005466  1518005466
          1                    l1_shared_bank_conflict           0           0           0
          1                          l1_local_load_hit           0           0           0
        Kernel: Histogram(unsigned char*, int, int*)
          1                            divergent_branch    32569410    32569410    32569410
          1                   global_store_transaction           0           0           0
          1                    l1_shared_bank_conflict    83440824    83440824    83440824
          1                          l1_local_load_hit           0           0           0
```

```
==21129== Metric result:
Invocations                               Metric Name                        Metric Description         Min
      Max         Avg
Device "GeForce GT 545 (0)"
        Kernel: Scan(int*, int*)
          1                               sm_efficiency                  Multiprocessor Activity      25.45%
   25.45%      25.45%
        Kernel: CountSort(unsigned char*, int*, unsigned char*, int)
          1                               sm_efficiency                  Multiprocessor Activity      33.33%
   33.33%      33.33%
        Kernel: Histogram(unsigned char*, int, int*)
          1                               sm_efficiency                  Multiprocessor Activity      99.10%
   99.10%      99.10%
```

time CPU	time GPU	size	winner
0.04 ms	0.082 ms	390	CPU
0.051 ms	0.088 ms	780	CPU
0.073 ms	0.095 ms	1562	CPU
0.112 ms	0.111 ms	3125	GPU
0.186 ms	0.14 ms	6250	GPU
0.349 ms	0.202 ms	12500	GPU
0.666 ms	0.327 ms	25000	GPU
1.295 ms	0.575 ms	50000	GPU
2.551 ms	1.073 ms	100000	GPU
5.15 ms	2.065 ms	200000	GPI
10.204 ms	4.185 ms	400000	GPU
19.775 ms	8.567 ms	800000	GPU

На маленьких данных проигрывает параллельная сортировка, так как выделение нитей трудоемкий процесс и его время суммируется с временем выполнения самой сортировки, а маленькое количество данных в таком случае быстрее отсортируется на CPU. Далее по таблице видно, что с увеличением количества данных последовательная сортировка подсчетом проигрывает параллельной почти более чем в 2 раза.

4 Выводы

Сортировка является одной из проблем обработки данных, это задача размещения элементов неупорядоченного набора значений в порядке монотонного возрастания или убывания. Вычислительная трудоемкость процедуры упорядочивания достаточно высока. Так, для ряда известных простых методов (пузырьковая сортировка и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных. Ускорение сортировки может быть обеспечено при использовании нескольких вычислительных элементов (процессоров или ядер). Исходный упорядочиваемый набор в этом случае разделяется на блоки, которые могут обрабатываться вычислительными элементами параллельно. После выполнения данной лабораторной работы я научилась реализовывать параллельную сортировку подсчетом, используя алгоритмы параллельной обработки данных: histogram, scan. Особое внимание нужно было уделить второму алгоритму, так как нужно было организовать использование разделяемой памяти без конфликтов.