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: К.О. Вахрамян

Группа: 8О-406Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2022

**Условие**

1. Цель работы, общая постановка задачи (один абзац).

Ознакомление с фундаментальными алгоритмами GPU: свертка

(reduce), сканирование (blelloch scan) и гистограмма (histogram). Реализация одной изсортировок на CUDA.

1. Вариант на “два”. Сортировка подсчетом.

Вариант №2, с использованием алгоритма сканирования из библиотеки Thrust.

**Программное и аппаратное обеспечение**

**GPU:**

--- General Information for device ---

Name: NVIDIA GeForce GTX 1650

Compute capability: 7.5

Clock rate: 1560000

Device copy overlap: Enabled

Kernel execution timeout : Enabled

--- Memory Information for device ---

Total global mem: 4100521984

Total constant Mem: 65536

Max mem pitch: 2147483647

Texture Alignment: 512

--- MP Information for device ---

Multiprocessor count: 16

Shared mem per mp: 49152

Registers per mp: 65536

Threads in warp: 32

Max threads per block: 1024

Max thread dimensions: (1024, 1024, 64)

Max grid dimensions: (2147483647, 65535, 65535)

**CPU:**

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

Address sizes: 39 bits physical, 48 bits virtual

CPU(s): 8

On-line CPU(s) list: 0-7

Thread(s) per core: 2

Core(s) per socket: 4

Socket(s): 1

NUMA node(s): 1

Vendor ID: GenuineIntel

CPU family: 6

Model: 158

Model name: Intel(R) Core(TM) i5-9300HF CPU @ 2.40GHz

Stepping: 13

CPU MHz: 1274.759

CPU max MHz: 2400.0000

CPU min MHz: 800.0000

BogoMIPS: 4800.00

Virtualization: VT-x

L1d cache: 128 KiB

L1i cache: 128 KiB

L2 cache: 1 MiB

L3 cache: 8 MiB

**OS:**

Linux Mint 20

**Compiler:**

nvcc

**Code Editor:**

VS Code

**Метод решения**

Сортировка подсчетом состоит из 3-х частей.

1. Гистограмма. Создается массив C, состоящий из нулей, размера N, где N — максимальный элемент. Далее производится проход по исходному массиву A и каждый C[A[i]] увеличивается на 1. Таким образом получаем частотный массив.
2. Алгоритм сканирования. В виду того, что данный алгоритм линейный, распараллеливание является нетривиальной задачей. Первый этап алгоритма — редукция. Складываем элементы, удаленные на 2^d друг от друга. В результате на n-1 позиции стоит сумма всех элементов. Второй этап — нисходящая развертка. Берутся по 2 элемента, также удаленные на 2^d друг от друга. Первый элемент суммируется со вторым и встает на его позицию, второй перемещается на позицию первого. В результате получаем массив, i-й элемент которого равен сумме предыдущих и данного.
3. Восстановление ответа. В цикле по результирующему массиву O на C[A[i]] позицию ставится A[i] элемент.

**Описание программы**

Гистограмма реализована при помощи атомарных операций. За счет них происходит конкурентный корректный доступ к одним и тем же участкам памяти, хотя и в угоду распараллеливания.

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

int offsetx = gridDim.x \* blockDim.x;

for (int i = idx; i < size; i += offsetx)

atomicAdd(hist + data[i], 1);

Реализация сканирования была взята из библиотеки thrust.

thrust::inclusive\_scan(thrust::device, hist, hist + N, hist);

Алгоритм работает с массивом произвольной длины.

Самостоятельно мною был реализован данный алгоритм.

Скан зависит от числа нитей в блоке и поэтому применяется над массивом определенной длины. Каждый тред обрабатывает 2 элемента.

\_\_shared\_\_ int temp[THREADS\_X2];  
int tID = threadIdx.x;

int ai = tID;

int bi = ai + (n >> 1);

Первая часть — редукция.

int offset = 1;

for(int d = n >> 1; d > 0; d >>= 1) {

\_\_syncthreads();

if (tID < d) {

int aidx = offset \* ((tID << 1) + 1) - 1;

int bidx = offset \* ((tID << 1) + 2) - 1;

aidx += CONFLICT\_FREE\_OFFSET(aidx);

bidx += CONFLICT\_FREE\_OFFSET(bidx);

temp[bidx] += temp[aidx];

}

offset <<= 1;

}

макрос CONFLICT\_FREE\_OFFSET необходим для предотвращения конфликтов банков памяти.

Вторая часть — развертка

for (int d = 1; d < n; d <<= 1) {

offset >>= 1;

\_\_syncthreads();

if (tID < d) {

int aidx = offset \* ((tID << 1) + 1) - 1;

int bidx = offset \* ((tID << 1) + 2) - 1;

aidx += CONFLICT\_FREE\_OFFSET(aidx);

bidx += CONFLICT\_FREE\_OFFSET(bidx);

int t = temp[aidx];

temp[aidx] = temp[bidx];

temp[bidx] += t;

}

}

Здесь также удалось избежать конфликтов банков памяти. Кроме того, использование битовых сдвигов вместо умножения/деления на 2 повышает производительность.

Т.к. данный алгоритм для ограниченного массива, а нам нужно обрабатывать массивы произвольной длины, используем разбиение на подмассивы.

\_\_global\_\_ void prescanArbitary(int\* data, int size) {

for (int i = THREADS\_X2 \* blockIdx.x; i < size; i += THREADS\_X2 \* gridDim.x)

blockScan(data + i, THREADS\_X2);

}

Затем в каждом подмассиве берем последний элемент и выполняем сканирование рекурсивно. Скан сумм складываем со сдвигом с исходным массивом

int tID = threadIdx.x;

int idx = blockIdx.x + 1;

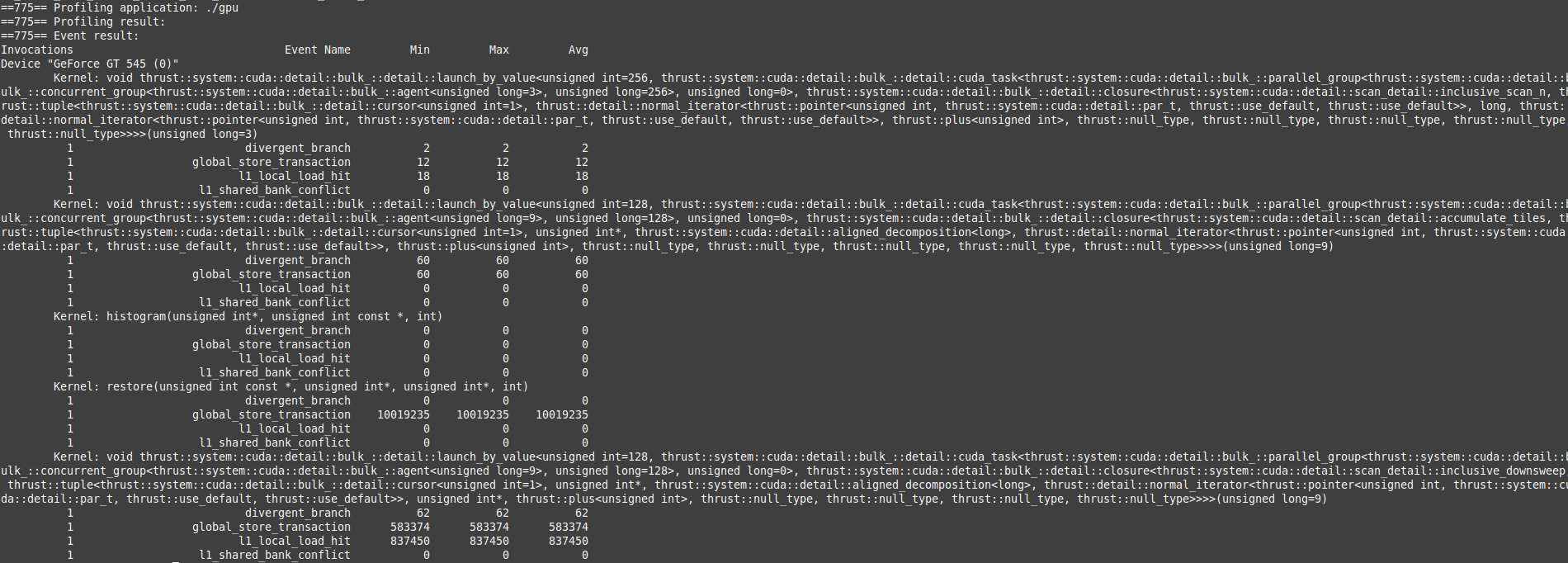
int offsetx = gridDim.x;

for (int i = idx; i < size; i += offsetx)

atomicAdd(data + THREADS\_X2 \* i + tID, sums[i — 1]);

**Результаты**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **1e3** | **1e5** | **1e7** |
| <<<1,32>>> | 1.4 | 5.26 | 361.97 |
| <<<32,32>>> | 1.37 | 1.88 | 59.83 |
| <<<1,128>>> | 1.36 | 2.58 | 129.62 |
| <<<1,1024>>> | 1.36 | 1.86 | 60.87 |
| <<<1024, 1024>>> | 1.39 | 1.86 | 59.38 |
| **CPU** | **94.77** | **106.05** | **1507.13** |



Спилинга регистров нет.

**Выводы**

Алгоритм сканирования может применяться с любым бинарным оператором, что делает его очень полезным. Линейные сортировки тоже используются повсеместно (там, где нужно отсортировать натуральные числа и т. д.). Что касается реализации параллельного алгоритма на CUDA, я понял идею и написал его, кроме того протестировал локально, но прохода тестов на чекере мне добиться не удалось. В связи с этим пришлось использовать библиотечную реализацию алгоритма.