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: М.А. Бронников

Группа: М8О-407Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2020

**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 №2:** Сортировка подсчетом.

**Программное и аппаратное обеспечение компьютера:**

Device: GeForce GT 545

Размер глобальной памяти: 3150381056

Размер константной памяти : 65536

Размер разделяемой памяти: 49152

Регистров на блок: 32768

Максимум потоков на блок: 1024

Количество мультипроцессоров : 3

OS: Linux Mint 20 Cinnamon

Редактор: VSCode

Машины в кластере:

1. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 16 Gb, GeForce GTX 1050, 2 Gb

2. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 16 Gb, GeForce GT 545, 3 Gb

3. Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 16 Gb, GeForce GTX 650, 2 Gb

4. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 12 Gb, GeForce GT 530, 2 Gb

5. Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 8 Gb, GeForce GT 530, 2 Gb

Все машины соединены гигабатным ethernet и находятся в подсети 10.10.1.1/24. В

качестве операционной системы установлена Ubuntu 16.04.6 LTS. Версии софта:

mpirun 1.10.2, g++ 4.8.4, nvcc 7.0

**Метод решения**

Сортировку подсчетом можно разбить на 3 части:

1. Подсчет количества повторений для каждого из значений в диапазоне. (Гистограмма)
2. Подсчет префиксных сумм от гистограммы чтобы получить. (Inclusive Scan)
3. Получение отсортированной последовательности на основе полученной префиксной суммы.

Поскольку первый и третий этапы алгоритма примитивны, стоит рассказать об особенностях алгоритма сканнирования.

Алгоритм сканнирования предложенный Blelloch позволяет эффективно расчитывать префиксные суммы в этапа с использованием разделяемой памяти, однако только для одного блока. Для того, чтобы получить префиксную сумму для неограниченных данных следует использовать технику scan and propagate, которая позволяет искать рекурсивно преффиксные суммы для последовательности любой длины следующим образом:

1. Расчитывается алгоритм сканирования для каждого блока.
2. Для получения сканирования для всей последовательности следует добавить сумму всех элементов массива до текущего блока ко всем элементам текущего блока. Для того, чтобы расчитать эти суммы следует воспользоваться scan-ом для массива последних элементов каждого из блоков.
3. Расчет этиого сканирования запускается рекурсивно, пока количество запускаемых блоков не станет равным 1.

**Описание программы**

Алгоритм сканирования был описан выше и имеет следующую реализацию:

\_\_device\_\_

void block\_scan(const uint32\_t tid, uint32\_t\* data, uint32\_t\* shared\_temp, const uint32\_t size){

uint32\_t ai = tid, bi = tid + (size >> 1);

// Block A:

uint32\_t bank\_offset\_a = AVOID\_OFFSET(ai);

uint32\_t bank\_offset\_b = AVOID\_OFFSET(bi);

uint32\_t offset = 1;

shared\_temp[ai + bank\_offset\_a] = data[ai];

shared\_temp[bi + bank\_offset\_b] = data[bi];

// up-sweep pass

for(uint32\_t d = size >> 1; d > 0; d >>= 1, offset <<= 1){

\_\_syncthreads(); // before next change of shared mem we need to sync last access

if(tid < d){

// Block B:

uint32\_t a\_idx = offset\*((tid << 1) + 1) - 1;

uint32\_t b\_idx = offset\*((tid << 1) + 2) - 1;

a\_idx += AVOID\_OFFSET(a\_idx);

b\_idx += AVOID\_OFFSET(b\_idx);

shared\_temp[b\_idx] += shared\_temp[a\_idx];

}

}

if(!tid){

shared\_temp[size - 1 + AVOID\_OFFSET(size - 1)] = 0;

}

offset >>= 1;

for(uint32\_t d = 1; d < size; d <<= 1, offset >>= 1){

\_\_syncthreads(); // before next change of shared mem we need to sync last access

if(tid < d){

// Block D:

uint32\_t a\_idx = offset\*((tid << 1) + 1) - 1;

uint32\_t b\_idx = offset\*((tid << 1) + 2) - 1;

a\_idx += AVOID\_OFFSET(a\_idx);

b\_idx += AVOID\_OFFSET(b\_idx);

uint32\_t t = shared\_temp[a\_idx];

shared\_temp[a\_idx] = shared\_temp[b\_idx];

shared\_temp[b\_idx] += t;

}

}

\_\_syncthreads();

data[ai] += shared\_temp[ai + bank\_offset\_a];

data[bi] += shared\_temp[bi + bank\_offset\_b];

}

\_\_global\_\_

void scan\_step(uint32\_t\* data\_in, const uint32\_t size){

\_\_shared\_\_ uint32\_t temp[THREADS\_X2 \* sizeof(uint32\_t)];

const uint32\_t thread\_id = threadIdx.x;

const uint32\_t start = blockIdx.x \* THREADS\_X2;

const uint32\_t step = THREADS\_X2 \* gridDim.x;

for(uint32\_t offset = start; offset < size; offset += step){

// launch scan algo for block

block\_scan(thread\_id, &data\_in[offset], temp, THREADS\_X2);

}

}

Алгоритм расчета сканирования был реализован на основе [2] таким образом, чтобы избежать конфликты при обращении с памятью. Для этого служит макрос, который зависит от количества банков памяти:

#define AVOID\_OFFSET(n) ((n) >> NUM\_BANKS + (n) >> (LOG\_NUM\_BANKS << 1))

С расчетом скана для генеральной последовательности помогает справиться рекурсивный алгоритм на хосте, недостатком которого является выделение нового блока памяти на каждом шаге алгоритма.

Для расчета гистограммы использование разделяемой памяти проблематично, поскольку большой диапазон значений не позволяет эффективно работать со всем спектром значений в разделяемой памяти. Однако этот же диапазон в среднем позволяет избежать конфликтов в глобальной памяти, поскольку потоки будут реже обращаться в одинаковые ячейки. Код ядра гистограммы:

\_\_global\_\_

void compute\_histogram(uint32\_t\* histogram, const int32\_t\* data, const uint32\_t data\_size){

uint32\_t step = blockDim.x \* gridDim.x;

uint32\_t start = threadIdx.x + blockIdx.x\*blockDim.x;

// compute statistic for each value with step

for(uint32\_t i = start; i < data\_size; i += step){

atomicAdd(&histogram[data[i]], 1);

}

}

Код сортировки в таком случае будет выглядеть следующим образом:

void cuda\_count\_sort(int32\_t\* h\_data, const uint32\_t size){

uint32\_t\* d\_counts;

int32\_t\* d\_data;

throw\_on\_cuda\_error(cudaMalloc((void\*\*)&d\_data, size\*sizeof(int32\_t)));

throw\_on\_cuda\_error(cudaMalloc(

(void\*\*)&d\_counts,

compute\_offset(INT\_LIMIT, THREADS\_X2)\*sizeof(uint32\_t))

);

throw\_on\_cuda\_error(cudaMemcpy(d\_data, h\_data, sizeof(int32\_t)\*size, cudaMemcpyHostToDevice));

throw\_on\_cuda\_error(cudaMemset(d\_counts, 0, INT\_LIMIT\*sizeof(uint32\_t)));

compute\_histogram<<<MAX\_BLOCKS, THREADS>>>(d\_counts, d\_data, size);

throw\_on\_cuda\_error(cudaGetLastError()); // catch errors from kernel

cudaThreadSynchronize();

scan(d\_counts, INT\_LIMIT);

// step 3: change input to true order

sort\_by\_counts<<<MAX\_BLOCKS, THREADS>>>(d\_data, d\_counts, INT\_LIMIT);

throw\_on\_cuda\_error(cudaGetLastError()); // catch errors from kernel

// copy data back:

throw\_on\_cuda\_error(cudaMemcpy(h\_data, d\_data, sizeof(int32\_t)\*size, cudaMemcpyDeviceToHost));

// Free data:

throw\_on\_cuda\_error(cudaFree(d\_data));

throw\_on\_cuda\_error(cudaFree(d\_counts));

}void cuda\_count\_sort(int32\_t\* h\_data, const uint32\_t size){

// device data:

uint32\_t\* d\_counts;

int32\_t\* d\_data;

// alloc data with overheap for scan algo

throw\_on\_cuda\_error(cudaMalloc((void\*\*)&d\_data, size\*sizeof(int32\_t)));

throw\_on\_cuda\_error(cudaMalloc(

(void\*\*)&d\_counts,

compute\_offset(INT\_LIMIT, THREADS\_X2)\*sizeof(uint32\_t))

);

// copy data into buffers

throw\_on\_cuda\_error(cudaMemcpy(d\_data, h\_data, sizeof(int32\_t)\*size, cudaMemcpyHostToDevice));

throw\_on\_cuda\_error(cudaMemset(d\_counts, 0, INT\_LIMIT\*sizeof(uint32\_t))); // init histogram with zero

// step 1: compute histogram

compute\_histogram<<<MAX\_BLOCKS, THREADS>>>(d\_counts, d\_data, size);

throw\_on\_cuda\_error(cudaGetLastError()); // catch errors from kernel

cudaThreadSynchronize(); // wait end

// step 2: prefix sums(inclusive scan) of histogram

scan(d\_counts, INT\_LIMIT);

// step 3: change input to true order

sort\_by\_counts<<<MAX\_BLOCKS, THREADS>>>(d\_data, d\_counts, INT\_LIMIT);

throw\_on\_cuda\_error(cudaGetLastError()); // catch errors from kernel

// copy data back:

throw\_on\_cuda\_error(cudaMemcpy(h\_data, d\_data, sizeof(int32\_t)\*size, cudaMemcpyDeviceToHost));

// Free data:

throw\_on\_cuda\_error(cudaFree(d\_data));

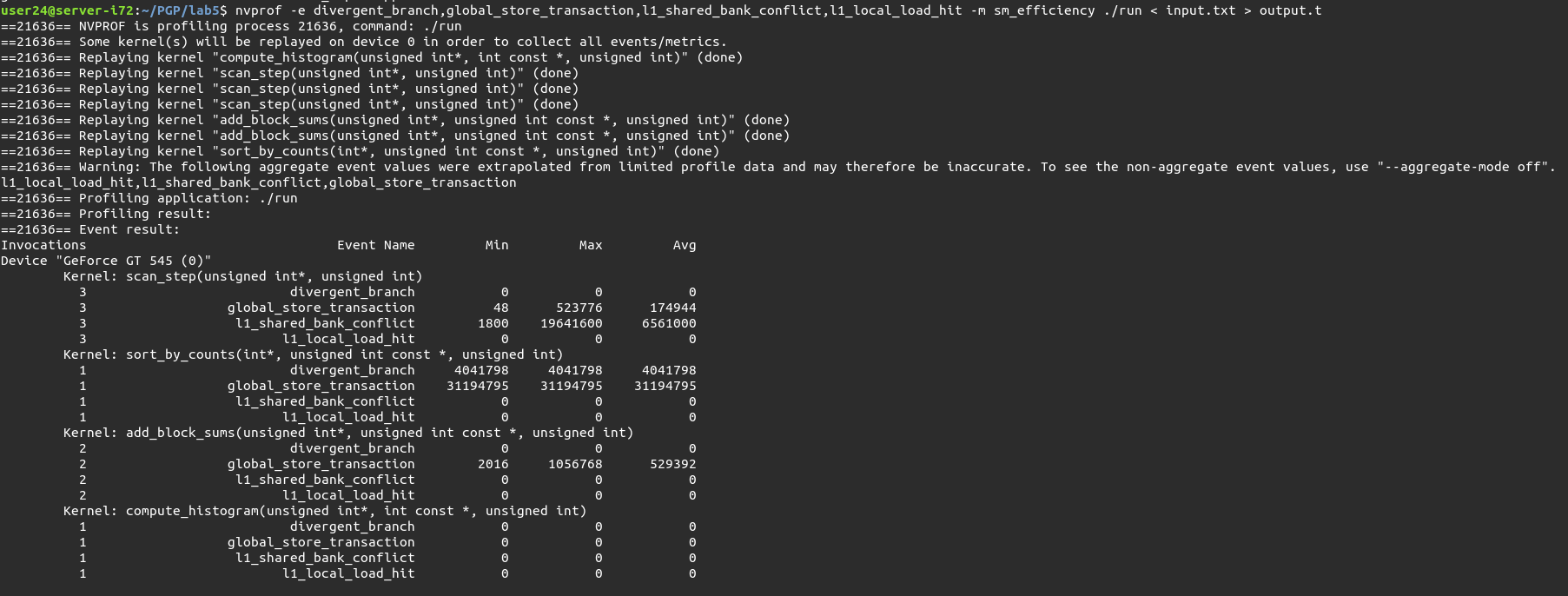
throw\_on\_cuda\_error(cudaFree(d\_counts));

}

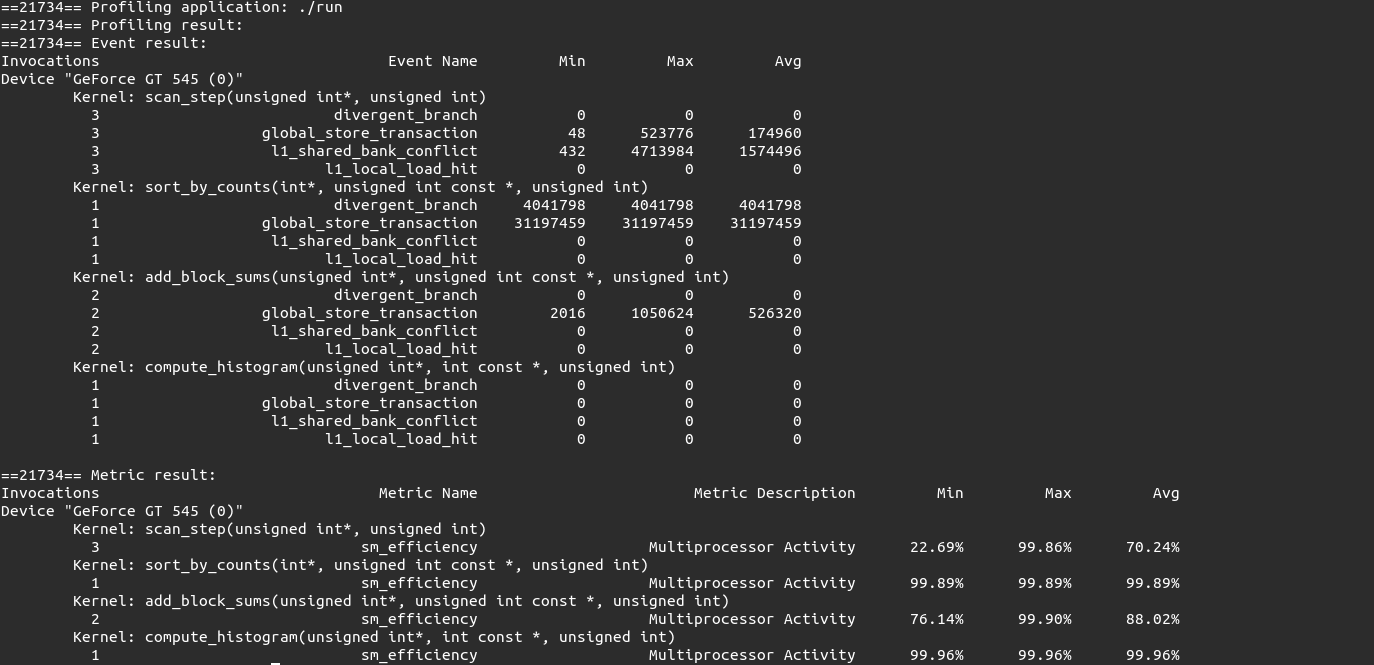
Заполнение значений в отсортрованном порядке не представляет особой информационной ценности, однако при его реализации я вдохновлялся источником [1].

**Результаты:**

Для того, чтобы проанализировать работу алгоритма на больших данных я воспользовался профилировщиком nvprof. Версия программы без оптимизации для уменьшения количества конфликтов банков памяти:

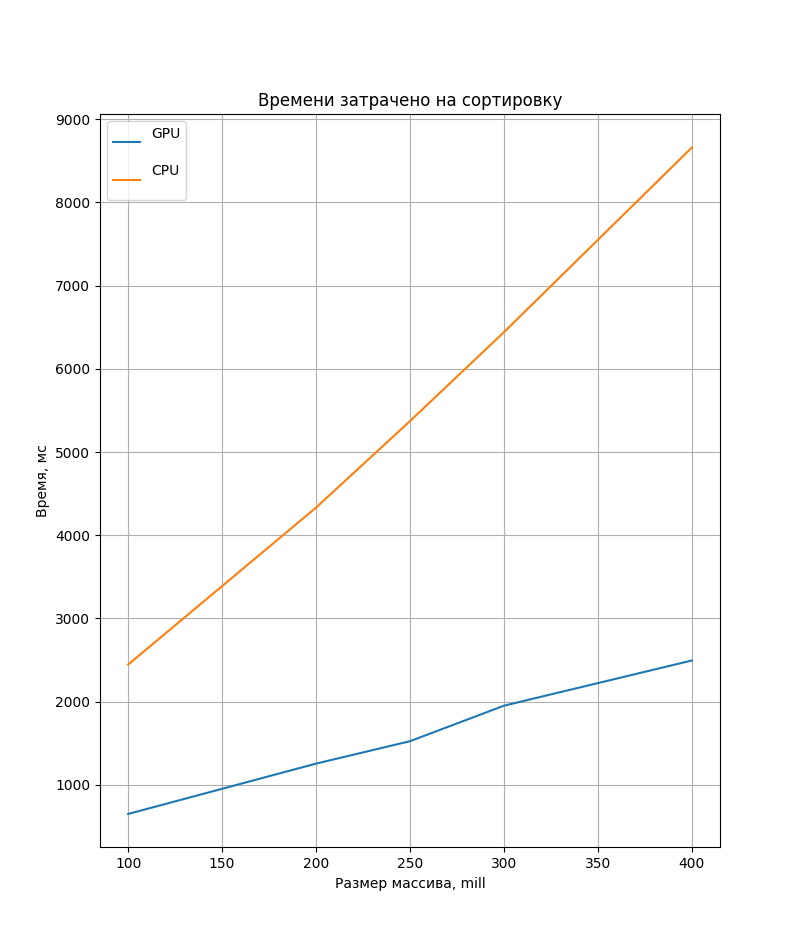


Когда как с оптимизацией мы получаем:



Как мы можем заметить, уменьшилось количество конфликтов в банках разделяемой памяти. Отмечу также в качестве недостатка алгоритма заполнения отсортированной последовательности — высокую дивергенцию нитей.

Выигрыш от использования GPU позволяет оценить следующий график:

****

**Выводы**

Сортировки — одно из самых важных и фундаментальных семейств алгоритмов классической информатики и трудно переоценить как их значимость, так и важность скорости их работы. Поэтому глупо пренебрегать любыми методами, которые позволяют получить необходмый результат в сжатые сроки. В этом нам и приходит на помощь GPU, который позволяет распараллелить алгоритмы сортировки.

В этой работе я научился реализовывать сортировку подсчетом с помощи классических алгоритмов в области параллельной обработки данных — scan и histogram, которые реализовал самостоятельно. Наиболее трудным в реализации мне показался алгоритм scan, поскольку он требует использование разделяемой памяти и его нетривиально применить на последовательности любой длины, что заставляет пользоваться рекурсией.

**Источники**

1. <https://www.researchgate.net/publication/245542734> — заполнение сортированных значений.

2. <https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda> - scan