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:

```
user24@server-l72:~/PGP/lab5$ nvprof -e divergent_branch,global_store_transaction,l1_shared_bank_conflict,l1_local_load_hit -m sm_efficiency ./run < input.txt > output.t
==21636== NVPROF is profiling process 21636, command: ./run
==21636== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==21636== Replaying kernel "compute_histogram(unsigned int*, int const *, unsigned int)" (done)
==21636== Replaying kernel "scan_step(unsigned int*, unsigned int)" (done)
==21636== Replaying kernel "scan_step(unsigned int*, unsigned int)" (done)
==21636== Replaying kernel "scan_step(unsigned int*, unsigned int)" (done)
==21636== Replaying kernel "add_block_sums(unsigned int*, unsigned int const *, unsigned int)" (done)
==21636== Replaying kernel "add_block_sums(unsigned int*, unsigned int const *, unsigned int)" (done)
==21636== Replaying kernel "sort_by_counts(int*, unsigned int const *, unsigned int)" (done)
==21636== Warning: The following aggregate event values were extrapolated from limited profile data and may therefore be inaccurate. To see the non-aggregate event values, use "--aggregate-mode off".
l1_local_load_hit,l1_shared_bank_conflict,global_store_transaction
==21636== Profiling application: ./run
==21636== Profiling result:
==21636== Event result:
Invocations                                Event Name         Min         Max         Avg
Device "GeForce GT 545 (0)"
       Kernel: scan_step(unsigned int*, unsigned int)
         3                           divergent_branch           0           0           0
         3                  global_store_transaction          48      523776      174944
         3                   l1_shared_bank_conflict        1800    19641600     6561000
         3                         l1_local_load_hit           0           0           0
       Kernel: sort_by_counts(int*, unsigned int const *, unsigned int)
         1                           divergent_branch     4041798     4041798     4041798
         1                  global_store_transaction    31194795    31194795    31194795
         1                   l1_shared_bank_conflict           0           0           0
         1                         l1_local_load_hit           0           0           0
       Kernel: add_block_sums(unsigned int*, unsigned int const *, unsigned int)
         2                           divergent_branch           0           0           0
         2                  global_store_transaction        2016     1056768      529392
         2                   l1_shared_bank_conflict           0           0           0
         2                         l1_local_load_hit           0           0           0
       Kernel: compute_histogram(unsigned int*, int const *, unsigned int)
         1                           divergent_branch           0           0           0
         1                  global_store_transaction           0           0           0
         1                   l1_shared_bank_conflict           0           0           0
         1                         l1_local_load_hit           0           0           0
```

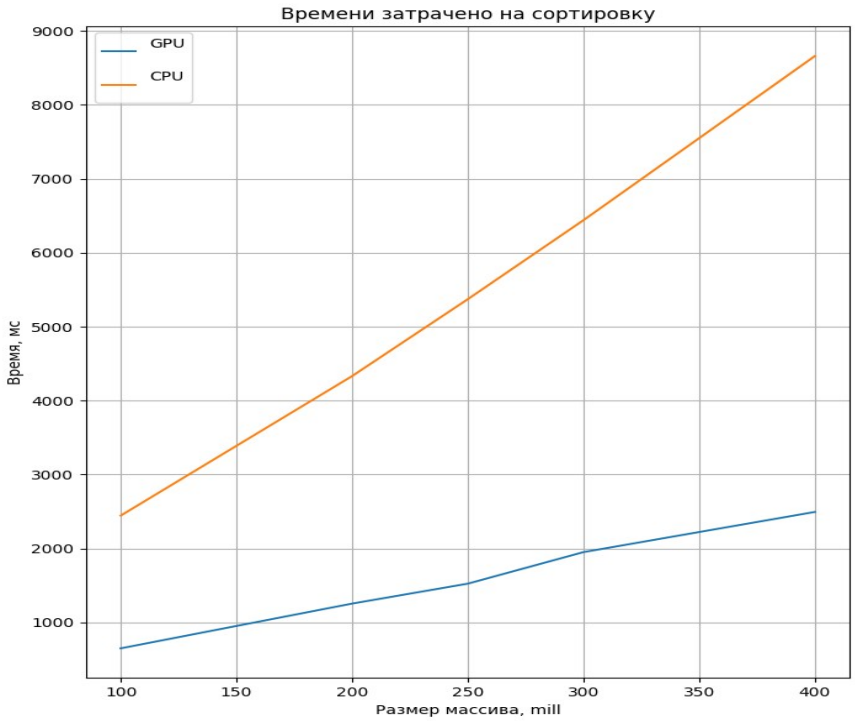
Когда как с оптимизацией мы получаем:

```
==21734== Profiling application: ./run
==21734== Profiling result:
==21734== Event result:
Invocations                                Event Name         Min         Max         Avg
Device "GeForce GT 545 (0)"
       Kernel: scan_step(unsigned int*, unsigned int)
         3                           divergent_branch           0           0           0
         3                  global_store_transaction          48      523776      174960
         3                   l1_shared_bank_conflict         432     4713984     1574496
         3                         l1_local_load_hit           0           0           0
       Kernel: sort_by_counts(int*, unsigned int const *, unsigned int)
         1                           divergent_branch     4041798     4041798     4041798
         1                  global_store_transaction    31197459    31197459    31197459
         1                   l1_shared_bank_conflict           0           0           0
         1                         l1_local_load_hit           0           0           0
       Kernel: add_block_sums(unsigned int*, unsigned int const *, unsigned int)
         2                           divergent_branch           0           0           0
         2                  global_store_transaction        2016     1050624      526320
         2                   l1_shared_bank_conflict           0           0           0
         2                         l1_local_load_hit           0           0           0
       Kernel: compute_histogram(unsigned int*, int const *, unsigned int)
         1                           divergent_branch           0           0           0
         1                  global_store_transaction           0           0           0
         1                   l1_shared_bank_conflict           0           0           0
         1                         l1_local_load_hit           0           0           0

==21734== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GT 545 (0)"
       Kernel: scan_step(unsigned int*, unsigned int)
         3                             sm_efficiency                   Multiprocessor Activity      22.69%      99.86%      70.24%
       Kernel: sort_by_counts(int*, unsigned int const *, unsigned int)
         1                             sm_efficiency                   Multiprocessor Activity      99.89%      99.89%      99.89%
       Kernel: add_block_sums(unsigned int*, unsigned int const *, unsigned int)
         2                             sm_efficiency                   Multiprocessor Activity      76.14%      99.90%      88.02%
       Kernel: compute_histogram(unsigned int*, int const *, unsigned int)
         1                             sm_efficiency                   Multiprocessor Activity      99.96%      99.96%      99.96%
```

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