МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет «Информационные технологии и прикладная математика»
Кафедра «Вычислительная математика и программирование»

Лабораторная работа №5
по курсу «Программирование графических процессоров»

Изучение технологии CUDA

Выполнил: И.И. Тишин
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2019

Условие

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. Все входные-выходные данные являются бинарными.

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.

Вариант 2. Сортировка подсчетом. Диапазон от 0 до 2^24-1.
Требуется реализовать сортировку подсчетом для чисел типа int.
Должны быть реализованы:

- Алгоритм гистограммы, с использованием атомарных операций.
- Алгоритм сканирования для любого размера, с рекурсией и бесконфликтным использованием разделяемой памяти.

Ограничения: $n \leq 135 * 10^6$

Программное и аппаратное обеспечение

GPU:

Device 0: "GeForce GTX 1060 with Max-Q Design"

CUDA Driver Version / Runtime Version 10.1 / 10.1

CUDA Capability Major/Minor version number: 6.1

Total amount of global memory: 6078 MBytes (6373572608 bytes)

(10) Multiprocessors, (128) CUDA Cores/MP: 1280 CUDA Cores

GPU Max Clock rate: 1480 MHz (1.48 GHz)

Memory Clock rate: 4004 Mhz

Memory Bus Width: 192-bit

L2 Cache Size: 1572864 bytes

Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)

Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers

Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers

Total amount of constant memory: 65536 bytes

Total amount of shared memory per block: 49152 bytes

Total number of registers available per block: 65536

Warp size: 32

Maximum number of threads per multiprocessor: 2048

Maximum number of threads per block: 1024

Max dimension size of a thread block (x,y,z): (1024, 1024, 64)

Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

Maximum memory pitch: 2147483647 bytes

Texture alignment: 512 bytes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.1,

NumDevs = 1

CPU:

Архитектура: x86_64

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

CPU(s): 12

Потоков на ядро: 2

Ядер на сокет: 6

Имя модели: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz

CPU max MHz: 4100,0000

CPU min MHz: 800,0000

BogoMIPS: 4416.00

Виртуализация: VT-x

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 9216K

Ram:

32 GiB, 2666 MHz

HDD:

500 GB

Программное обеспечение:

Ubuntu 18.04.2 LTS

g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0

nvcc: release 10.1, V10.1.105

Метод решения

Считать входные данные и скопировать их на GPU память. Перевести данные в гистограмму. Гистограмму по рекурсивно суммируем каждый блок, потом рекурсивно складываем блоки. После того, как мы просуммировали Гистограмму, мы можем отсортировать всю последовательность, получая с помощью гистограммы нужные нам значения.

Сложность n/shared_size+2^24/ shared_size, где n – размер массива, а shared_size – размер массива в разделяемой памяти.

Описание программы

Программа состоит из функций: main, kernel1, kernel2, DTH и HTD.

В main считываются данные со стандартного ввода. После чего, вызывается функция DTH, которая переводит массив в гистограмму размера 2^24-1. Далее вызывается функция сложения kernel1.

В этой функции происходит рекурсивное, поблочное сложение, затем функция kernel2 выполняет рекурсивное сложение блоков, полученных с помощью kernel1. Полученную гистограмму мы преобразуем в отсортированный массив. Отсортированный массив возвращается функции main, где он подается на стандартный вывод.

```
#define f(n) (n)+((n)>>5)
#define f1(n,step) (((n)/step-1)*step+step-1)
__global__ void kernel1(int* height, long long sz,long long step) {
        long long idx = threadIdx.x + blockIdx.x * blockDim.x;
        long long offsetx = gridDim.x * blockDim.x;
        long long j,i,k;
        __shared__ int data[f(tread_size)];
        for(i=idx*step+step-1;i<sz;i+=offsetx*step){
            __syncthreads();
            data[f(threadIdx.x)]=height[i];
            //редукция
            for(k=1;k<tread_size;k*=2){
                __syncthreads();
                for( j=threadIdx.x*k*2+k-
1;j+k<tread_size;j+=gridDim.x*k*2){
                    data[f(j+k)]+=data[f(j)];
                }
                __syncthreads();
            }
```

```
                __syncthreads();
                //обратная редукция
                for( k=tread_size/2;k>1;k/=2){
                    __syncthreads();
                    for(j=k-1+threadIdx.x*k;j+k/2<tread_size-1;j+=gridDim.x*k){
                        data[f(j+k/2)]+=data[f(j)];
                    }
                    __syncthreads();
                }
                __syncthreads();
                height[i]=data[f(threadIdx.x)];
            }

            /*for (int j = i - 1; j + i < sz; j += i * 2) {
                height[j + i] += height[j];
                }*/
    }
    __global__ void kernel2(int* height, long long sz,long long step) {
        long long idx = threadIdx.x + blockIdx.x * blockDim.x;
        long long offsetx = gridDim.x * blockDim.x;
        long long i;
        for(i=idx*step/tread_size+step+step/tread_size-
1;i<sz;i+=offsetx*step/tread_size){
            if(i%step!=step-1){
                height[i]+=height[f1(i,step)];
            }
        }
    }
    __global__ void DTH(int* data, int n,int* height) {
            int idx = threadIdx.x + blockIdx.x * blockDim.x;
            int offsetx = gridDim.x * blockDim.x;
            int i;
            for(i = idx; i < n; i += offsetx){
                //__threadfence_system();
                atomicAdd(height+data[i],1);
                //printf("%d %d\n",height[data[i]],data[i]);
            }
    }
    __global__ void HTD(int* data,int* in, int n,int* height) {
            int idx = threadIdx.x + blockIdx.x * blockDim.x;
            int offsetx = gridDim.x * blockDim.x;
            int i;
            for(i = idx; i < n; i += offsetx){
                //__threadfence_system();

                //__threadfence();

                //printf("%d %d\n",height[in[i]],in[i]);
                data[atomicSub(height + in[i],1)-1]=in[i];
                //__threadfence();
            }
    }
    int main() {
            long long sz=16777216 ;
            int n;
            //cin >> n;
```

```
            fread(&n, sizeof(int), 1, stdin);
            //int* in=(int*) malloc( sizeof(int) * n);
            int* data=(int*) malloc( sizeof(int) * n);
            //int* height=(int* ) malloc( sizeof(int) * n);
            fread(data, sizeof(int), n, stdin);
            /*for(int i=0;i<n;i++){
                cin >> data[i];
            }*/
            int *gpu_in,*gpu_data,*gpu_height;
            fprintf(stderr, "n=%d\n",n);
            for(int i=0;i<min(n,100);i++){
                fprintf(stderr, "%d ",data[i]);
            }
            fprintf(stderr, "\n");
            CSC( cudaMalloc( &gpu_in, n*sizeof(int) ) );
            CSC( cudaMalloc( &gpu_data, n*sizeof(int) ) );
            CSC( cudaMemcpy( gpu_in, data, n*sizeof(int),
cudaMemcpyHostToDevice ) );
            CSC( cudaMalloc( &gpu_height, sz*sizeof(int)));
            CSC( cudaMemset(gpu_height,0,sz*sizeof(int)));
            dim3 threads = tread_size;
            dim3 blocks = tread_size;
            /*for(i = 0; i < n; i++){
                    height[data[i]]++;
            }*/

            DTH<<<blocks,threads>>>(gpu_in,n,gpu_height);
            long long i=1;
            for(;i<sz;i*=tread_size)
                kernel1<<<blocks,threads>>>(gpu_height,sz,i);
            /*for (int j = i - 1; j + i < sz; j += i * 2) {
            height[j + i] += height[j];
            }*/
            //__threadfence_system();

            for(;i>1;i/=tread_size)
            kernel2<<<blocks,threads>>>(gpu_height,sz,i);
            /*for(j=i-1;j+i/2<sz-1;j+=i){
                height[j+i/2]+=height[j];
            }*/
            //__threadfence_system();
            /*for(i = idx; i < n; i += offsetx){
                    in[--height[in[i]]]=data[i];
            }*/
            HTD<<<blocks,threads>>>(gpu_data,gpu_in,n,gpu_height);
            CSC( cudaMemcpy( data,gpu_data,  n*sizeof(int),
cudaMemcpyDeviceToHost ) );
            /*for(int i=0;i<n;i++){
                cout << data[i]<<" ";
            }
            cout << endl;*/
            fwrite(data, sizeof(int), n, stdout);
            /*for(int i=0;i<n;i++){
                fprintf(stderr, "%d ",data[i]);
            }
            fprintf(stderr, "\n");*/
```

```
        CSC(cudaFree ( gpu_height ));
        CSC(cudaFree ( gpu_in ));
        CSC(cudaFree ( gpu_data ));
         free(data);
        return 0;
}
```

Результаты

N \ tread_size	Radix	Quicksort
1500 \ 128	13.31 ms	503 us
1500 \ 1024	16.94 ms	
5000 \ 128	13.36 ms	1 ms
5000 \ 1024	16.95 ms	
12000 \ 128	13.38 ms	2.5 ms
12000 \ 1024	16.99 ms	
1000000 \ 128	19.64 ms	245 ms
1000000 \ 1024	23.06 ms	

Выводы

Реализованный алгоритм правильно сортирует любую последовательность чисел. Реализация сортировки на GPU дает выигрыш во времени, при большом количество исходных данных, так как у алгоритма есть 2 асимптотики одна из которых не зависит от входных данных. Ещё я получил ускорение, при уменьшении используемых ядер и потоков. Это связано увеличением оптимальности их работы.