3МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа №5**

**по курсу «Параллельная Обработка Данных»**

**Сортировка чисел на GPU. Свертка, сканирование, гистограмма.**

Выполнил: Иларионов Д.А.

Группа: М8О-408Б-17

Преподаватели: Крашенинников К.Г.,

Морозов А.Ю.

Москва, 2020

**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.

Исследование производительности программы с помощью утилиты nvprof

(обязательно отразить в отчете).

**Вариант 4. Сортировка чет-нечет.**

Требуется реализовать блочную сортировку чет-нечет для чисел типа int.

Должны быть реализованы:

● Алгоритм чет-нечет сортировки для предварительной сортировки

блоков.

● Алгоритм битонического слияния, с использованием разделяемой

памяти.

Ограничения: n ≤ 16 \* 10^6

**Программное и аппаратное обеспечение**

**GPU:**

* Name: GeForce GTX 1060
* Compute capability: 6.1
* Частота видеопроцессора: 1404 – 1670 (Boost) МГц
* Частота памяти: 8000 МГц
* Графическая память: 6144 МБ
* Разделяемая память: 2048 Б
* Количество регистров на блок: 65536
* Максимальное количество блоков: (2147483647, 65535, 65535)
* Максимальное количество нитей: (1024, 1024, 64)
* Количество мультипроцессоров: 10

**Сведения о системе:**

* Процессор: Intel Core i7-8750H 2.20GHz x 6
* Оперативная память: 16 ГБ
* SSD: 128 ГБ
* HDD: 1000 ГБ

**Программное обеспечение:**

* OS: Windows 10
* IDE: Visual Studio 2019
* Компилятор: nvcc

**Метод решения**

По сути данная лабораторная должна быть легче предыдущей. Однако, с ней я возился еще дольше, и она мне показалась сложнее, не смотря на несложный вариант. Много раз не проходила по времени. Сначала, я неправильно понял условие, и вместо битонического слияния каждый раз сортировал чет-нечет. Потом, я сделал рекурсивное битоническое слияние, преподаватель сказал, что это плохо, поэтому снова не проходила по времени на 14 тесте. Затем, у меня был неправильный ответ. Как оказалось – гонка потоков. А дело в том, что синхронизация работала некорректно, потому что была в конце 2 ветвей, и в итоге синхронизировались не все потоки. Да, я переделал рекурсию в цикл. А когда я перенес синхронизацию из ветвей в конец цикла, то все заработало. А сама программа работает так. Во первых у нас все данные в битовом формате (как в 2-3 ЛР), в итоге пришлось помучиться с тем, как их считывать и записывать, но нашел в интернете и разобрался. Далее, мы применяем сортировки и используем разделяемую память, доступ к которой примерно в 100 раз быстрее, чем к глобальной. Это позволяет выполнять итерации быстрее, поэтому и программа прошла в итоге по времени. Мы сначала сортируем чет-нечет весь массив. В итоге у нас блоки с возрастающими элементами. Радует то, что разделяемая память – общая для всех потоков одного блока, в итоге я мог запускать по 1024 блока потоков одновременно, что тоже очень оптимизировало программу. Далее мы используем битоническое слияние n+1 раз, в итоге у нас получается отсортированный массив, который мы так же записываем в выходной бинарный файл.

**Описание программы**

**Файл kernel.cu**

#include <string>

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

//for \_\_syncthreads()

#ifndef \_\_CUDACC\_RTC\_\_

#define \_\_CUDACC\_RTC\_\_

#endif // !(\_\_CUDACC\_RTC\_\_)

#include <device\_functions.h>

#define SHARED\_MEMORY 512

#define MAXIMAL\_INTEGER 2147483647 // 2^31 - 1

using namespace std;

#define CSC(call) \

do { \

cudaError\_t res = call; \

if (res != cudaSuccess) { \

fprintf(stderr, "ERROR in %s:%d. Message: %s\n", \

\_\_FILE\_\_, \_\_LINE\_\_, cudaGetErrorString(res)); \

exit(0); \

} \

} while (0)

using namespace std;

class ClassSort {

public:

int\* intArray;

int size = 0;

ClassSort()

{

freopen(NULL, "rb", stdin);

fread(&size, sizeof(int), 1, stdin);

cerr << "Len: " << size << "\n";

intArray = (int\*)malloc(sizeof(\*intArray) \* size);

fread(intArray, sizeof(int), size, stdin);

//for (int i = 0; i < size; i++) {

// cerr << (int)intArray[i] << " ";

//}

fclose(stdin);

};

ClassSort(string inpFile) {

FILE\* file;

if ((file = fopen(inpFile.c\_str(), "rb")) == NULL) {

cerr << "Can't load file" << "\n";

exit(1);

}

fread(&size, sizeof(size), 1, file);

cerr << "Len: " << size << "\n";

intArray = (int\*)malloc(sizeof(\*intArray) \* size);

fread(intArray, sizeof(int), size, file);

fclose(file);

}

void PrintRSS() {

for (int i = 0; i < size - 1; i++) {

if ((int)intArray[i + 1] < (int)intArray[i]) {

cerr << (int)intArray[i] << " : " << (int)intArray[i+1] << "| ";

}

}

cerr << "\n";

freopen(NULL, "wb", stdout);

fwrite(intArray, sizeof(\*intArray), size, stdout);

fclose(stdout);

}

void PrintRSS(string outFile) {

for (int i = 0; i < size; i++) {

cerr << (int)intArray[i] << " ";

}

cerr << "\n";

FILE\* file = fopen(outFile.c\_str(), "wb");

fwrite(intArray, sizeof(\*intArray), size, file);

fclose(file);

}

~ClassSort() {

fclose(stdin);

fclose(stdout);

free(intArray);

};

};

\_\_device\_\_ void oddEvenOne(const int tid, int\* tmp, int shift, int len) {

if ((tid + shift) % 2 == 0) {

int a = tid;

int b = tid + 1;

if (b < len && tmp[a] > tmp[b]) {

int temp = tmp[a];

tmp[a] = tmp[b];

tmp[b] = temp;

}

}

}

\_\_global\_\_ void OddEvenBlocks(int\* theArray, int size, int shift2) {

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

int tt = blockDim.x;

int tid = threadIdx.x;

int bid = blockDim.x \* blockIdx.x;

int tid\_full = bid + threadIdx.x;

\_\_shared\_\_ int tmp[SHARED\_MEMORY];

\_\_shared\_\_ int shift;

for (int begin = bid + shift2; begin < size; begin += tx) {

tid\_full = begin + threadIdx.x;

int end = (size > begin + tt ? begin + tt : size);

int len = end - begin;

if (tid < len) {

tmp[tid] = theArray[tid\_full];

}

else {

tmp[tid] = MAXIMAL\_INTEGER;

}

\_\_syncthreads();

for (int j = 0; j < len; ++j) {

if (j % 2 == 0) {

shift = 0;

}

else {

shift = 1;

}

oddEvenOne(tid, tmp, shift, len);

\_\_syncthreads();

}

if (tid < len) {

theArray[tid\_full] = tmp[tid];

}

\_\_syncthreads();

}

}

\_\_global\_\_ void BitonicMerge(int\* theArray, int size, int shift2) {

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

int tt = blockDim.x; // 2 \* SHARED\_MEMORY

int tid = threadIdx.x;

int bid = blockDim.x \* blockIdx.x;

int shiftt = shift2;

int tid\_full = bid + threadIdx.x;

\_\_shared\_\_ int tmp[2 \* SHARED\_MEMORY];

for (int begin = bid + shiftt; begin < size; begin += tx) {

tid\_full = begin + threadIdx.x;

int end = (size > begin + tt ? begin + tt : size);

int len = end - begin;

if ((tid < tt/2) && (tid < len)) {

tmp[tid] = theArray[tid\_full];

}

else if (tid < len) {

tmp[tt + tt/2 - tid - 1] = theArray[tid\_full];

}

else if ((tid < tt) && (tid >= tt / 2)) {

tmp[tt + tt / 2 - tid - 1] = MAXIMAL\_INTEGER;

}

else {

tmp[tid] = MAXIMAL\_INTEGER;

}

\_\_syncthreads();

int base = tt / 2;

int shift = 0;

int n\_tid = tid;

while (base >= 1) {

if (n\_tid >= base) {

int opTid = n\_tid - base;

if (tmp[opTid + shift] > tmp[n\_tid + shift]) {

int temp = tmp[opTid + shift];

tmp[opTid + shift] = tmp[n\_tid + shift];

tmp[n\_tid + shift] = temp;

}

if (base >= 1) {

base = base / 2;

n\_tid = n\_tid - base;

shift = shift + base;

}

}

else {

if (base >= 1) {

base = base / 2;

}

}

\_\_syncthreads();

}

\_\_syncthreads();

if (tid < len) {

theArray[tid\_full] = tmp[tid];

}

\_\_syncthreads();

}

}

void Sorting(ClassSort\* sort) {

int\* D\_theArray;

CSC(cudaMalloc((void\*\*)& D\_theArray, sizeof(\*D\_theArray) \* sort->size));

CSC(cudaMemcpy(D\_theArray, sort->intArray, sizeof(\*sort->intArray) \* sort->size, cudaMemcpyHostToDevice));

//pre-entry odd-even sort

OddEvenBlocks << <1024, SHARED\_MEMORY >> > (D\_theArray, sort->size, 0);

CSC(cudaGetLastError());

//cycle bitonic-merge / odd-even sort

int iters = (sort->size + 1) / SHARED\_MEMORY + 2;

for (int i = 0; i < iters; ++i) {

int evOdd = i % 2;

BitonicMerge << <1024, 2 \* SHARED\_MEMORY >> > (D\_theArray, sort->size, evOdd \* SHARED\_MEMORY);

CSC(cudaGetLastError());

//cerr << "Iter: " << i << "\n";

}

CSC(cudaMemcpy(sort->intArray, D\_theArray, sizeof(int) \* sort->size, cudaMemcpyDeviceToHost));

CSC(cudaFree(D\_theArray));

}

int main(void) {

//ClassSort theSort = ClassSort("test.data");

//Sorting(&theSort);

//theSort.PrintRSS("test\_out.data");

ClassSort theSort = ClassSort();

Sorting(&theSort);

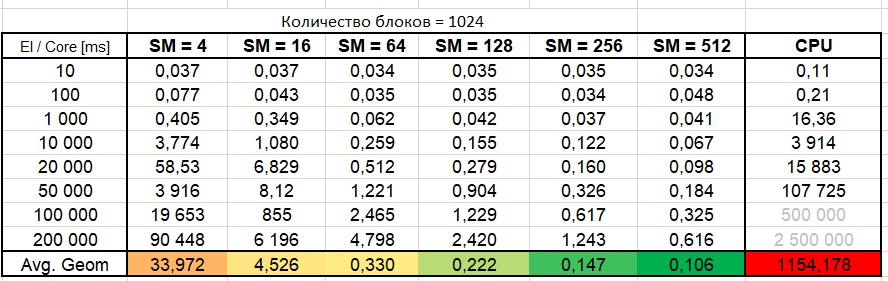
theSort.PrintRSS();

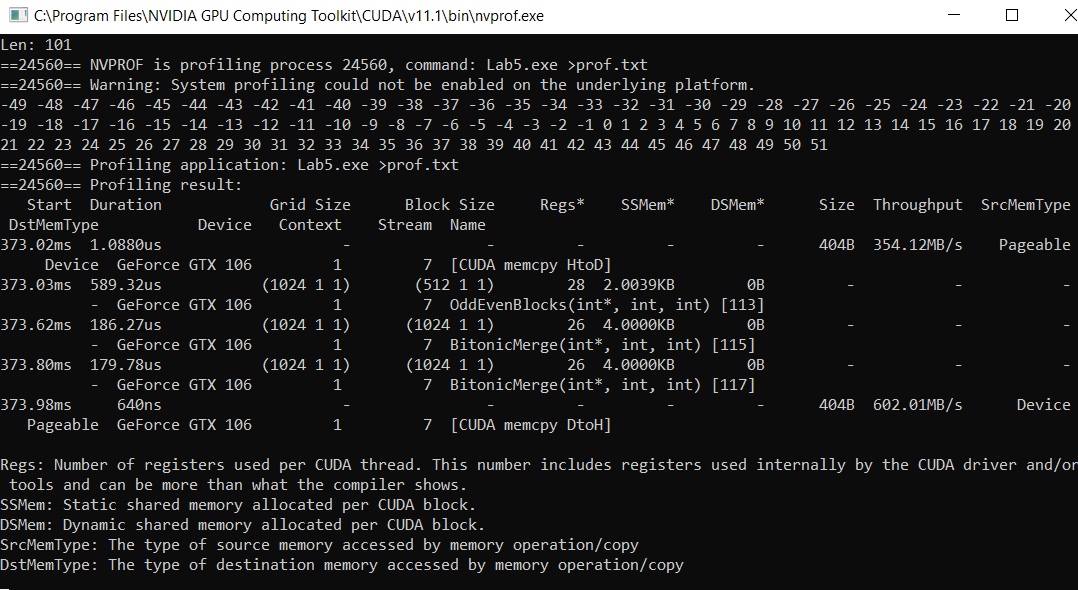
return 0;

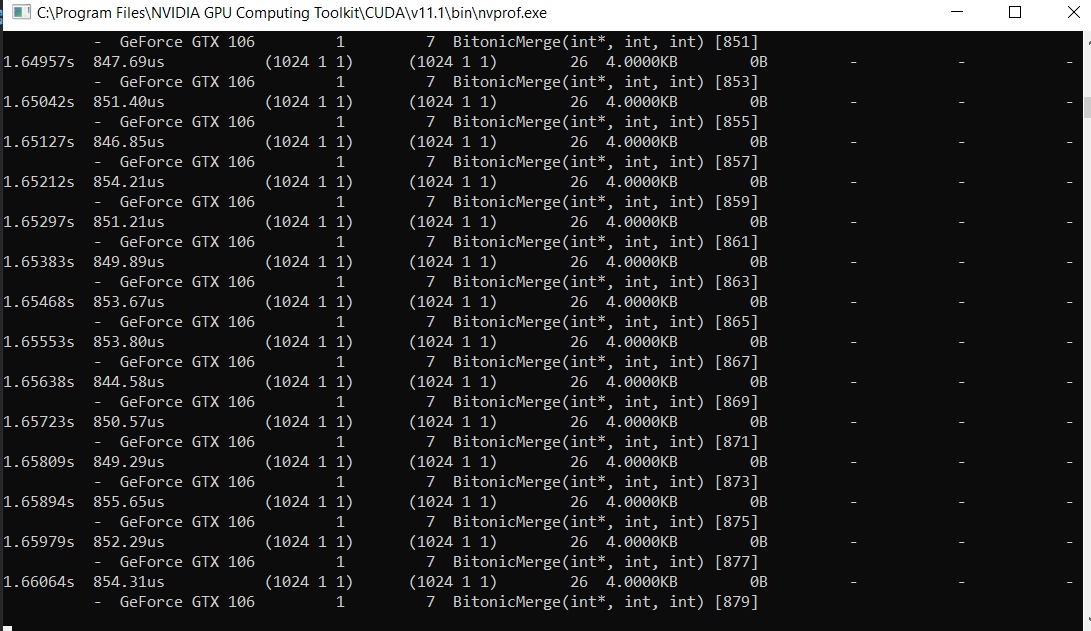
}

**Результаты**

Приведу таблицу и результаты nvprof. Программа в visual studio у меня почему-то сразу закрывалась и не давала полных логов. Поэтому, я успел сделать лишь пару скриншотов (для 10 и для 1 млн. элементов). В таблице отобразил время работы для одинакового количества блоков (1024), но с разной разделяемой памятью и числом нитей. А также для CPU я перестал тестировать примерно после 50 тыс. элементов, тк это уже занимало очень много времени, поэтому я примерно прикинул время, но оно не точное.







**Выводы**

В данной лабораторной мне пришлось работать с бинарным вводом и выводом. А также изучить некоторые сортировки. Из-за проблем со скоростью программы потратил много времени и нервов, а также потревожил преподавателя. Но в итоге, я доделал лабу! И наконец, свобода) Хотя, я думаю, я еще сделаю необязательную лабу по OpenGL, но это уже после праздников. А так я закрепил знания о сортировках, ну и понял, как работать с разделяемой памятью и как правильно синхронизировать потоки, чтобы не было их гонки. Также в выводе напишу еще кое-что. Поздравляю вас с наступающим 2021 годом, желаю, чтобы он принес вам много хорошего, чтобы этот год стал одним из лучших в вашей жизни! Счастья, удачи, любви, крепкого здоровья и стальных нервов, желаю успеха на работе, и исполнения всех желаний. Ведь в Новый Год случаются чудеса! Нужно лишь верить в это. Все будет хорошо:)