МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3-4**

**по курсу «Программирование графических процессоров»**

**Сортировка чисел на GPU.**

**Свертка, сканирование, гистограмма.**

Выполнил: Н.А. Зайцев

Группа: 8О-408Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2016

**Условие**

Описание задачи:

1. **Цель работы.** Ознакомление с фундаментальными алгоритмами GPU: свертка (reduce), сканирование (blelloch scan) и гистограмма (histogram). Реализация одной из сортировок на CUDA, с использованием различных видов памяти.
2. **Вариант 2.** Сортировка подсчетом. Диапазон от 0 до 255.

Требуется реализовать сортировку подсчетом для чисел типа uchar.

Должны быть реализованы:

● Алгоритм гистограммы, с использованием атомарных операций и

разделяемой памяти.

● Алгоритм сканирования, с использованием разделяемой памяти.

Ограничения: n ≤ 537 \*

**Программное и аппаратное обеспечение**

**GPU:**

Name : GeForce GTX 650

Compute capability : 3.0

Total Global Memory : 2147483648

Shared memory per block : 49152

Registers per block : 65536

Warp size : 32

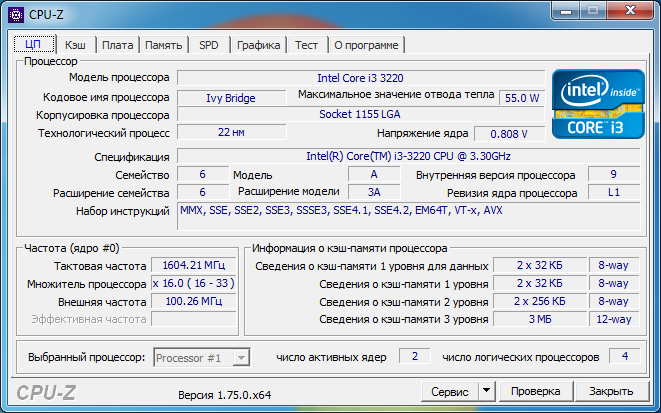
Max threads per block : (1024, 1024, 64)

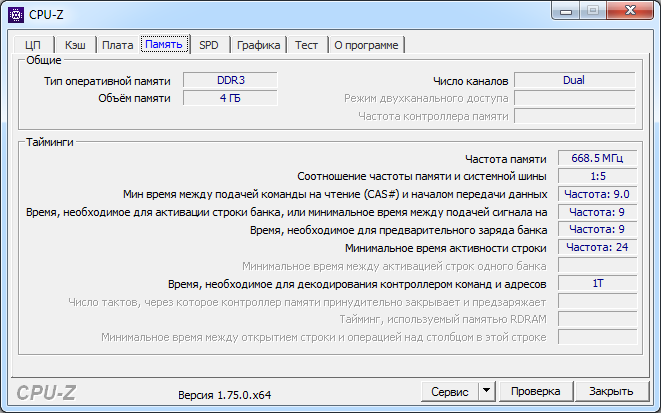
Max block : (2147483647, 65535, 65535)

Total constant memory : 65536

Multiprocessors count : 2

**CPU:**





**HDD:**

Производитель: Seagate

Модель: ST2000DM001-1CH164 ATA Device

Размер : 1,82 ТБ

**OS:**

Windows 7 Professional SP1 64-bit

**IDE:**

Microsoft Visual Studio 2013 + CUDA Plugins

**Compiler:**

nvcc

**Profilers:**

nvprof

nvvp

**Debugger**:

Nvidia CUDA Debugger

**Метод решения**

Программа была реализована при помощи двух ядер. Первое ядро с 32-мя блоками и 256-ю потоками в каждом реализовывает алгоритм гистограммы. Второе ядро из одного блока и 256-ти потоков реализует алгоритм blelloch scan. Остальная часть сортировки реализована на CPU.

**Описание программы**

Программа состоит из одного файла, но разбита на функцию main и два ядра.  
  
Ядро kernel\_histogram:

\_\_global\_\_ void kernel\_histogram(unsigned char \*src, int length, unsigned int \*histogram)

{

\_\_shared\_\_ unsigned int local\_histogram[256];

if (threadIdx.x == 0)

{

memset(&local\_histogram, 0, sizeof(unsigned int) \* 256);

}

\_\_syncthreads();

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

while (tid < length)

{

atomicAdd(&local\_histogram[src[tid]], 1);

tid += gridDim.x \* blockDim.x;

}

\_\_syncthreads();

atomicAdd(&histogram[threadIdx.x], local\_histogram[threadIdx.x]);

}

Ядро подсчитывает общую гистограмму для всех потоков одного блока, используя атомарные операции, а в конце выполнения ядра с помощью все тех же атомарных операций добавляет эту локальную гистограмму из разделяемой памяти к общей гистограмме из глобальной памяти.

Ядро kernel\_scan:

\_\_global\_\_ void kernel\_scan(unsigned int \*histogram)

{

\_\_shared\_\_ unsigned int local\_histogram[256];

if (threadIdx.x == 0)

{

memcpy(&local\_histogram, histogram, sizeof(unsigned int) \* 256);

}

\_\_syncthreads();

int pow = 2;

while (pow <= 256)

{

if (threadIdx.x \* pow + pow - 1 < 256)

{

local\_histogram[threadIdx.x \* pow + pow - 1] += local\_histogram[threadIdx.x \* pow + pow - pow / 2 - 1];

}

\_\_syncthreads();

pow \*= 2;

}

if (threadIdx.x == 0)

{

local\_histogram[255] = 0;

}

\_\_syncthreads();

pow = 256;

while (pow > 1)

{

if (threadIdx.x \* pow + pow - 1 < 256)

{

unsigned int sum = local\_histogram[threadIdx.x \* pow + pow - 1] + local\_histogram[threadIdx.x \* pow + pow - pow / 2 - 1];

local\_histogram[threadIdx.x \* pow + pow - pow / 2 - 1] = local\_histogram[threadIdx.x \* pow + pow - 1];

local\_histogram[threadIdx.x \* pow + pow - 1] = sum;

}

\_\_syncthreads();

pow /= 2;

}

histogram[threadIdx.x] += local\_histogram[threadIdx.x];

}

Реализует алгоритм blelloh scan. Cостоит из 2-х фаз — фаза свертки (reduce), и вторая фаза, называемая down-sweep (развёртка вниз). В результате получаем общее количество операций *O(N)*, а шагов *O(log(N))*.

**Результаты**

Давайте сравним производительность программы, реализованной на CPU и GPU с использованием CUDA на объёмном тесте. Вставил в код таймеры непосредственно перед и после выполнения алгоритма. Тест напрямую считывается программой с диска из файла.

Используем тест максимального размера – последовательность длиною в 537000000 элементов.

CPU:



GPU:



Распараллеленная программа на данном тесте работает на 25 % быстрее. Разница ощутима.

Если запускать таймеры в самом начале программы и в самом конце, таким образом включая в замеренное время помимо прочих операций операцию копирования последовательности на GPU, то результаты будут следующими:

CPU:



GPU:



Заметим, что хоть программа на GPU и работает всё равно быстрее чем на CPU, операция копирования существенно уменьшает наш выигрыш по времени. Прирост производительности в 10,6%.

**Выводы**

В результате выполнения данной работы я научился распараллеливать на видеокарте алгоритм свёртки, сканирования и гистограммы. Реализовал алгоритм линейной сортировки подсчётом с использованием разделяемой памяти и атомарных операций.

**Использованные источники**

1. https://habrahabr.ru/company/epam\_systems/blog/247805/