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.

Выполнила: Алексюнина Ю.В.

Группа:М8О-407Б

Преподаватели:

К.Г.Крашенинников, А.Ю. Морозов

Москва, 2021

Условие:

Цель работы. Ознакомление с фундаментальными алгоритмами GPU: свертка (reduce), сканирование (blelloch scan) и гистограмма (histogram). Реализация одной из сорти-ровок на CUDA. Использование разделяемой и других видов памяти. Исследование производительности программы с помощью утилиты nvprof.

Вариант 8. Поразрядная сортировка.

Требуется реализовать поразрядную сортировку для чисел типа uint.

Должны быть реализованы:

1. Алгоритм сортировки через префиксные суммы для одного битового разряда.
2. Алгоритм сканирования для любого размера, с рекурсией и бесконфликтным использованием разделяемой памяти.
   Ограничения: $n < 128*10^6$:

Все входные-выходные данные являются бинарными и считываются из stdin и выводятся в stdout.

Программное и аппаратное обеспечение

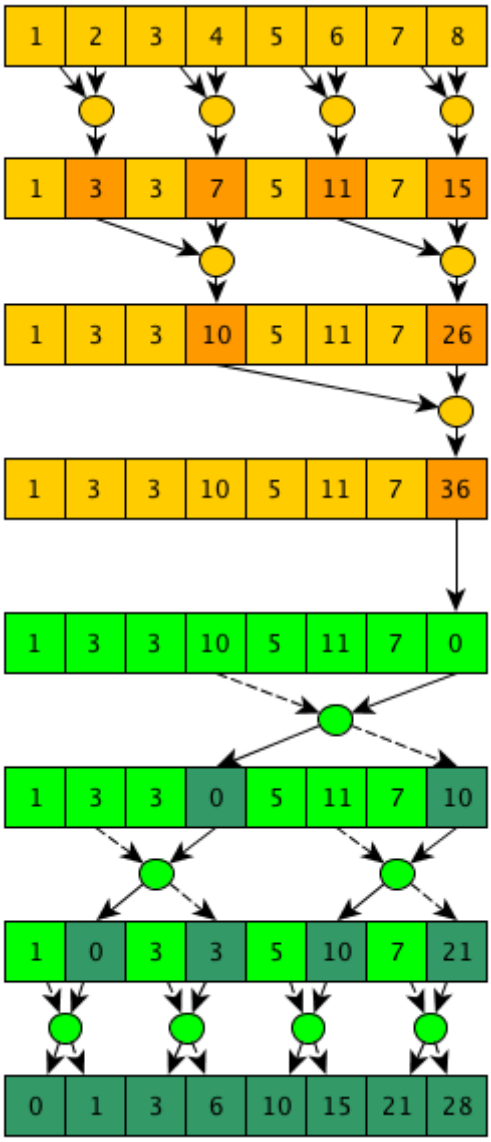
GPU:
• Name: GeForce GTX 750 Ti
• Compute capability: 5.0
• Графическая память: 4294967295
• Разделяемая память: 49152
• Константная память: 65536
• Количество регистров на блок: 65536
• Максимальное количество блоков: (2147483647, 65535, 65535)
• Максимальное количество нитей: (1024, 1024, 64)
• Количество мультипроцессоров: 5

Сведения о системе:
• Процессор: Intel Core i5-4460 3.20GHz
• ОЗУ: 16 ГБ
• HDD: 930 ГБ

Программное обеспечение:
• OS: Windows 8.1

• IDE: Visual Studio 2019

Метод решения:



Поразрядная сортировка реализована на префиксных суммах с помощью алгоритма сканирования scan. Идея метода состоит в побитовой сортировке чисел, так, алгоритм проходит 32 итерации (число бит). На каждой итерации вычисляется префиксная сумма и выполняется перемещение элементов. На последней итерации все элементы считаются отсортированными.

Будем применять алгоритм сканирования по блокам, т.к. размер разделяемой памяти ограничен. Далее, будем составлять из последних элементов блока новую последовательность и повторять алгоритм рекурсивно, двигаясь вниз до тех пор, пока последние элементы не поместятся в один блок.

Далее, будем подниматься наверх, прибавляя все элементы подряд, кроме последнего, ко всем элементам предшествующего блока, начиная со второго блока. Повторяем процесс, пока не вернёмся на первый уровень.

Алгоритм scan можно разделить на два этапа – на первом этапе строится дерево сумм, а на втором на основе этих сумм строится результирующий массив префиксных сумм.

Файл kernel.cu:

```
#include "cuda_runtime.h"

#include "device_launch_parameters.h"

#include <iostream>


#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)


typedef unsigned int uint;


__global__ void kernel_radixsort(uint* gpu_arr_prev, int arr_size, int
k, uint* gpu_arr, uint* bits)
{
	//чисто формулка
	int idx = blockIdx.x * blockDim.x + threadIdx.x;
	int offsetx = blockDim.x * gridDim.x;


	for (int i = idx; i < arr_size; i += offsetx)
	{
		if (((gpu_arr_prev[i] >> k) & 1) == 0)
		{
			gpu_arr[i - (int)bits[i]] = gpu_arr_prev[i];
		}
		else
		{
			gpu_arr[(int)bits[i] + (arr_size -
(int)bits[arr_size])] = gpu_arr_prev[i];
		}
	}
}
```

```
__global__ void kernel_bitsshift(uint* gpu_arr, int arr_size, int k,
uint* bits)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      int offsetx = blockDim.x * gridDim.x;

      for (int i = idx; i < arr_size; i += offsetx)
            bits[i] = (gpu_arr[i] >> k) & 1;
}

__global__ void kernel_bitsshift_block(uint* bits, int arr_size, uint*
lel, int threads)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      int offsetx = blockDim.x * gridDim.x;

      for (int i = idx + threads; i < arr_size; i += offsetx)
      {
            bits[i] += lel[i / threads - 1];
      }
}

__global__ void kernel_blellochscan_block(uint* bits, int arr_size,
uint* lel, int threads)
{
      int idx = threadIdx.x;
      int offsetx = blockDim.x * gridDim.x;

      __shared__ uint data[1024]; //threads

      uint lel_el, temp;
      int it;

      for (int ib = 0; ib < arr_size; ib += offsetx)
      {
```

```
            it = 1;
            if (idx + ib + blockIdx.x * blockDim.x < arr_size)
            {
                  data[idx] = bits[idx + ib + blockIdx.x * blockDim.x];

                  __syncthreads();

                  for (int i = 1; i < blockDim.x; i *= 2, it++) //i <<=
1
                  {
                        if (((idx - i + 1) & ((1 << it) - 1)) == 0)
                              data[idx + i] += data[idx];
                        __syncthreads();
                  }

                  if (idx == 0)
                  {
                        lel_el = data[blockDim.x - 1];
                        data[blockDim.x - 1] = 0;
                  }
                  __syncthreads();

                  it = it - 1;
                  for (int i = blockDim.x / 2; i > 0; i /= 2, it--) //i
>>= 1
                  {
                        if (((idx - i + 1) & ((1 << it) - 1)) == 0)
                        {
                              temp = data[idx + i];
                              data[idx + i] = data[idx] + data[idx + i];
                              data[idx] = temp;
                        }

                        __syncthreads();
                  }
```

```
                if (idx == 0)
                {
                      lel[ib / blockDim.x + blockIdx.x] = lel_el;
                      bits[ib + (blockIdx.x + 1) * blockDim.x - 1] =
lel_el;
                }
                else
                {
                      bits[ib + blockIdx.x * blockDim.x + idx - 1] =
data[idx];
                }
           }
     }
}



void blellochscan(uint* bits, int arr_size, int blocks, int threads)
{
     int lel_size = arr_size / threads;
     int mult_block_lel = (lel_size + (threads - 1)) & (-threads);
     //size_bLastEls = (sizeLastEls % BLOCK_SIZE == 0) ? sizeLastEls :
BLOCK_SIZE * ((int)sizeLastEls / BLOCK_SIZE + 1);


     uint* lel;
     cudaMalloc(&lel, mult_block_lel * sizeof(uint));
     cudaMemset(lel + lel_size, 0, (mult_block_lel - lel_size) *
sizeof(uint));


     kernel_blellochscan_block << <blocks, threads >> > (bits,
arr_size, lel, threads);


     if (lel_size > 1)
     {
          blellochscan(lel, mult_block_lel, blocks, threads);
```

```
            kernel_bitsshift_block << <blocks, threads >> > (bits,
arr_size, lel, threads);
      }
      cudaFree(lel);
}

int main()
{
      int blocks = 1024;
      int threads = 1024;

      int arr_size;
      fread(&arr_size, sizeof(int), 1, stdin);
      //std::cin >> arr_size;

      uint* arr = new uint[arr_size];
      fread(arr, sizeof(uint), arr_size, stdin);
      //for (int i = 0; i < arr_size; i++)
      //{
            //std::cin >> arr[i];
      //}

      //bit mask
      int mult_block = (arr_size + (threads - 1)) & (-threads);
//наибольшее кратное блоку

      uint* gpu_arr;
      CSC(cudaMalloc(&gpu_arr, arr_size * sizeof(uint)));
      CSC(cudaMemcpy(gpu_arr, arr, arr_size * sizeof(uint),
cudaMemcpyHostToDevice));

      uint* gpu_arr_prev;
      CSC(cudaMalloc(&gpu_arr_prev, arr_size * sizeof(uint)));
      CSC(cudaMemcpy(gpu_arr_prev, arr, arr_size * sizeof(uint),
cudaMemcpyHostToDevice));
```

```
	uint* bits; //bit array
	CSC(cudaMalloc(&bits, (mult_block + 1) * sizeof(uint)));

	for (int k = 0; k < 32; k++) //uint - 32 бита
	{
		kernel_bitsshift << <blocks, threads >> > (gpu_arr, arr_size, k, bits + 1); //k-тый бит
		CSC(cudaGetLastError());

		cudaMemset(bits, 0, sizeof(uint)); //1 эл-т в 0

		if (mult_block > 1) //если он равен единице, тогда не нужно ничего дополнять
		{
			//дополняем нулями до кратности, (mult_block - arr_size) - yekb
			cudaMemset(bits + arr_size + 1, 0, (mult_block - arr_size) * sizeof(uint));

			blellochscan(bits + 1, mult_block, blocks, threads);
		}

		uint* temp = gpu_arr;
		gpu_arr = gpu_arr_prev;
		gpu_arr_prev = temp;

		kernel_radixsort << <blocks, threads >> > (gpu_arr_prev, arr_size, k, gpu_arr, bits);
		CSC(cudaGetLastError());

	}

	CSC(cudaMemcpy(arr, gpu_arr, arr_size * sizeof(uint), cudaMemcpyDeviceToHost));
```

```
	fwrite(arr, sizeof(uint), arr_size, stdout);
	//for (int i = 0; i < arr_size; i++)
	//{
		//std::cout << std::endl << arr[i] << ' ';
	//}

	cudaFree(bits);
	cudaFree(gpu_arr);
	cudaFree(gpu_arr_prev);

	free(arr);

	return 0;
}
```

Результаты:

Выводы:

В данной лабораторной работе был реализован распараллеленный алгоритм поразрядной сортировки на основе алгоритма scan, который реализован для работы с массивами любого размера. Также, я познакомилась с работой с разделяемой памятью. Лабораторная работа показалась мне трудной, в ней немало частей и для понимания, и для реализации, особенно, если несколько дней подряд не спать…. Но, к счастью, у меня была возможности послушать видео объяснения и осознать эту лабораторную.