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. Жерлыгин
Группа: 8О-408Б
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 сортировок на CUDA. Использование разделяемой и других видов памяти. Исследование производительности программы с помощью утилиты nvprof

Вариант 7. Карманная сортировка с чет-нечет сортировкой в каждом кармане.

Программное и аппаратное обеспечение

GPU:

1. Compute capability: 7.5;
2. Графическая память: 4294967296;
3. Разделяемая память: 49152;
4. Константная память: 65536;
5. Количество регистров на блок: 65536;
6. Максимальное количество блоков: (2147483647, 65535, 65535);
7. Максимальное количество нитей: (1024, 1024, 64);
8. Количество мультипроцессоров: 6.

Сведения о системе:

1. Процессор: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
2. Память: 16,0 ГБ;
3. HDD: 237 ГБ.

Программное обеспечение:

1. OS: Windows 10;
2. IDE: Visual Studio 2019;
3. Компилятор: nvcc.

Метод решения

Сортировка останавливается при равенстве max и min, которые мы вычисляем при помощи reduce.

1. Вычисляем количество карманов и присваиваем им ключи.
2. Считаем histogramm и scan. Получаем отсортированные элементы по ключам.
3. После scan мы получаем индексы начала и конца каждого кармана.
4. Группируем карманы.
5. В каждом кармане сортируем сортировкой чет-нечет.
6. Если карман не помещается в выделяемую память, запускаем для этого кармана рекурсию.

Описание программы

Основные части программы:

- Карманная сортировка

```
__host__ void pocket_sort(float* gpu_data, int size) {
   int count = size / 550 + 1;
   thrust::device_ptr<const float> ptr = thrust::device_pointer_cast(gpu_data);
   auto minmax_values = thrust::minmax_element(thrust::device, ptr, ptr + size);

   float minimum = *minmax_values.first;
   float maximum = *minmax_values.second;

   if (fabs(minimum - maximum) < 1e-9) {
       return;
   }

   int* s_size = NULL;
   int* spl_values = NULL;
   unsigned int* s_sizes = NULL;
   float* gpu_val = NULL;

   cudaMalloc((void **) &s_size, count * sizeof(int));
   cudaMemset(s_size, 0, count * sizeof(int));

     kernel_calculate_histogramm <<<512, 512>>>(gpu_data, size, s_size, minimum, maximum,
count);

   cudaMalloc((void **) &spl_values, count * sizeof(int));

   r_scan(s_size, count, spl_values);

   cudaMalloc((void **) &s_sizes, count * sizeof(unsigned int));
   cudaMemset(s_sizes, 0, count * sizeof(unsigned int));

   cudaMalloc((void **) &gpu_val, size * sizeof(float));
    kernel_split_histogramm <<<512, 512 >>>(gpu_data, size, gpu_val, spl_values, s_sizes,
minimum, maximum, count);

   int pockets_count = count;
   int* pockets_size = (int* ) malloc(pockets_count * sizeof(int));
   memset(pockets_size, 0, pockets_count * sizeof(int));

   int* pocket = (int* ) malloc(pockets_count * sizeof(int));
   int pocket_id = 0;

   for (int i = 0; i < count; i++) {
       int spl_1 = 0;
       cudaMemcpy(&spl_1, &(spl_values[i]), sizeof(int), cudaMemcpyDeviceToHost);

       int spl_size = 0;
       cudaMemcpy(&spl_size, &(s_size[i]), sizeof(int), cudaMemcpyDeviceToHost);

       if (spl_size > 2048) {
           pocket_id++;

           float* temp_sp = &(gpu_val[spl_1]);

           pocket_sort(temp_sp, spl_size);

           pocket[pocket_id] = spl_1;
           pockets_size[pocket_id] = -1;
```

```
            pocket_id++;
        } else {
            int current_value = 2048 - pockets_size[pocket_id];

            if (spl_size <= current_value) {
                if (current_value == 2048) pocket[pocket_id] = spl_1;

                pockets_size[pocket_id] += spl_size;
            } else {
                pocket_id++;

                pocket[pocket_id] = spl_1;
                pockets_size[pocket_id] = spl_size;
            }
        }
    }

    if (pockets_size[pocket_id] == 0) {
        pockets_count = pocket_id;
    } else {
        pockets_count = pocket_id + 1;
    }

    int* gpu_v = NULL;
    cudaMalloc((void **) &gpu_v, pockets_count * sizeof(int));
    cudaMemcpy(gpu_v, pocket, pockets_count * sizeof(int), cudaMemcpyHostToDevice);

    int* v_size = NULL;
    cudaMalloc((void **) &v_size, pockets_count * sizeof(int));
    cudaMemcpy(v_size, pockets_size, pockets_count * sizeof(int), cudaMemcpyHostToDevice);

     kernel_sort23 <<<dim3(pockets_count, 1, 1), dim3(2048 / 2, 1, 1)>>>(gpu_val, size, gpu_v,
v_size);

    cudaMemcpy(gpu_data, gpu_val, size * sizeof(float), cudaMemcpyDeviceToDevice);
}
```

- Сортировка чет-нечет:

```
__global__ void kernel_sort23(float* pockets, int size, int* pocket_pos, int* pocket_sizes) {
    __shared__ float pocket_shared_buff[2048];

    pocket_shared_buff[threadIdx.x * 2] = FLT_MAX;
    pocket_shared_buff[threadIdx.x * 2 + 1] = FLT_MAX;

    int current_size = pocket_sizes[blockIdx.x];

    if (current_size == -1) {
        return;
    }

    __syncthreads();

    if (threadIdx.x * 2 < current_size) {
                    pocket_shared_buff[threadIdx.x   *   2]   =   pockets[threadIdx.x   *   2   +
pocket_pos[blockIdx.x]];
    }

    if (threadIdx.x * 2 + 1 < current_size) {
                pocket_shared_buff[threadIdx.x  *  2  +  1]  =  pockets[threadIdx.x  *  2  +  1  +
pocket_pos[blockIdx.x]];
    }

    __syncthreads();

    for (int idx = 0; idx < blockDim.x; idx++) {
        if (threadIdx.x * 2 + 1 < 2047) {
```

```
            if (pocket_shared_buff[threadIdx.x * 2 + 1] > pocket_shared_buff[threadIdx.x * 2 +
2]) {
                float temp_value = pocket_shared_buff[threadIdx.x * 2 + 1];
                 pocket_shared_buff[threadIdx.x * 2 + 1] = pocket_shared_buff[threadIdx.x * 2 +
2];
                pocket_shared_buff[threadIdx.x * 2 + 2] = temp_value;
            }
        }

        __syncthreads();

        if (threadIdx.x < 2048) {
            if (pocket_shared_buff[threadIdx.x * 2] > pocket_shared_buff[threadIdx.x * 2 + 1])
{
                float temp_value = pocket_shared_buff[threadIdx.x * 2];
                pocket_shared_buff[threadIdx.x * 2] = pocket_shared_buff[threadIdx.x * 2 + 1];
                pocket_shared_buff[threadIdx.x * 2 + 1] = temp_value;
            }
        }
        __syncthreads();
    }

    if (threadIdx.x * 2 < current_size) {
          pockets[threadIdx.x * 2 + pocket_pos[blockIdx.x]] = pocket_shared_buff[threadIdx.x *
2];
    }

    if (threadIdx.x * 2 + 1 < current_size) {
         pockets[threadIdx.x * 2 + 1 + pocket_pos[blockIdx.x]] = pocket_shared_buff[threadIdx.x
* 2 + 1];
    }
}
```

Результаты

Для того, чтобы проанализировать работу алгоритма на больших данных я воспользовался профилировщиком nvprof. В сводном режиме по умолчанию nvprof представляет обзор ядер графического процессора и копий памяти в исполняемом файле. Сводка группирует все вызовы одного и того же ядра вместе, показывая общее время и процент от общего времени приложения для каждого ядра. В дополнение к сводному режиму nvprof поддерживает режимы GPU-Trace и API-Trace, которые позволяют видеть полный список всех запусков ядра и копий памяти, а в случае режима API-Trace - все вызовы API CUDA. Ниже я привел пример профилирования примера приложения с использованием nvprof --print-gpu-trace.

Start	Duration	Grid Size	Block Size	Regs*	SSMem*	DSMem*	Size	Throughput
337.03ms	1.2800us	-	-	-	-	-	4.0000KB	2.9802GB/s
338.45ms	5.6580us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.47ms	5.5790us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.48ms	5.3710us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.49ms	5.8590us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.50ms	5.8770us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.51ms	5.8360us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.52ms	6.6940us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.53ms	6.2190us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.55ms	6.2640us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.56ms	6.2580us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.57ms	7.0730us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.58ms	6.4190us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.59ms	6.4220us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.60ms	6.4560us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.61ms	6.4250us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.63ms	7.3080us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.64ms	7.1360us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.65ms	7.1770us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.66ms	7.1530us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.67ms	7.2040us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.68ms	7.1560us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.69ms	7.9690us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.70ms	7.8120us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.72ms	7.7880us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.73ms	7.7500us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.74ms	7.7510us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.75ms	7.7300us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.76ms	7.7840us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.77ms	8.7900us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.78ms	8.3940us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.79ms	8.3420us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.81ms	8.2920us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.82ms	8.3580us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.83ms	8.1350us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.84ms	8.1570us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.86ms	8.1300us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.87ms	9.5950us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.88ms	8.8630us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.89ms	8.8480us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.90ms	8.8440us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.91ms	8.8640us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.92ms	8.8520us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-
338.93ms	8.8690us	(32 1 1)	(1024 1 1)	16	4.0000KB	0B	-	-

Выводы

В этой работе я познакомился и научился реализовывать сортировку чет-нечет и карманную сортировку с помощи классических алгоритмов в области параллельной обработки данных. Было интересно узнать про еще один вариант сортировки чисел и реализовать его с помощью CUDA.

Также я познакомился с профилировщиком nvprof. Это довольно полезный и удобный инструмент для профилирования кода для GPU.

Во время выполнения возникали проблемы с понимаем того, в какой момент и что нужно реализовать.