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: В.А. Петросян
Группа: 8О-408Б-17
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2020

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.
Вариант 1: Битоническая сортировка.

Программное и аппаратное обеспечение

Сведения о системе:

1. Процессор: Intel Core i7-Q720 1.60GHz
2. Количество ядер 4.
3. Количество потоков 8.
4. Оперативная память: 8 ГБ
5. HDD: 465 ГБ

Программное обеспечение:

1. OS: Windows 7
2. IDE: Visual Studio 2019
3. Компиляторы: nvcc

Метод решения

Битоническая сортировка это применение в строго определенном порядке некоторого количества сортирующих фильтров. Применив один фильтр мы обратимся ко всем элементам ровно один раз, поэтому сложность работы одного фильтра будет $\theta(N)$.
Общее количество применений фильтров это $O((\log N)^2)$ так как всего у алгоритма $\log N$ итераций на каждой из которых совершается не более $\log N$ применений фильтров.
Итоговая асимптотика алгоритма это $O(N * (\log N)^2)$.

В данной лабораторной работе алгоритм использует разделяемую память при возможности загрузки данных в неё. Если такой возможности нет, то используется глобальная память.

Описание программы

```
#define _idxShared(id) ((id) + ((id) / 31))
#define _sizeShared(id) ((id) + ((id) / 31) + 100)
```

Пришлось написать два макроса для удобного обращения к элементам в shared memory. Каждый 31-й элемент сделал фейковым. Это обеспечивает решение проблем с банками памяти. Если обращаться к элементам без макроса **_idxShared()**, то возникают конфликты 32-ого порядка.

Компаратор, который упорядочивает в возрастающем порядке.

```
__device__ void compASC(int* left, int* right){
    if( *left > *right ){
        int tmp = *left;
        *left = *right;
        *right = tmp;
    }
}
```

Компаратор, который упорядочивает в убывающем порядке.

```
__device__ void compDESC(int* left, int* right){
    if( *left < *right ){
        int tmp = *left;
        *left = *right;
        *right = tmp;
    }
}
```

Ядро, которое работает через глобальную память.

```
__global__ void BitonicSort(int* nums, int n, int iterShift, int localStep, int localShift){

    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int offset = blockDim.x * gridDim.x;

    int prev;
    int comp = ASCENDING;
    int i = 2 * idx - (idx & (localStep - 1));
    int kernelShift = (localStep > offset) ? localStep : offset;
    if( (i >> iterShift) % 2 == 1){
        comp = DESCENDING;
    }
    while(i < n){
        if( comp == ASCENDING ){
            compASC(nums + i, nums + i + localStep);
        }
        else{
            compDESC(nums + i, nums + i + localStep);
        }

        prev = i;
        i += offset;

        if( (i >> localShift) != (prev >> localShift) ){
            i += kernelShift;
            if( (i >> iterShift) != (prev >> iterShift) ){
                comp ^= 1;
            }
        }
    }
}
```

Параметр **iterShift** нужен только для того, чтобы сортирующий фильтр понимал в каком порядке сортировать числа. В зависимости от итерации элементы, находящиеся на одной и той же позиции, сортируются по-разному. На итерации $Iter$ параметр $\mathbf{iterShift} = \log_2(\mathrm{Iter})$.

Параметр **localShift** нужен для того, чтобы понять находится ли следующий индекс в той же самой области, в которой находился предыдущий или нет. Если индексы находятся в разных областях, то нужно смещаясь дальше не попасть в область действия других потоков GPU или в область действия элементов, которые участвовали в сравнении неявным образом (как правый элемент в функции компаратора). Для этого существует параметр **kernelShift**, который позволяет избежать коллизий.

Параметр **localStep** задаёт смещение между сравниваемыми элементами.

Ядро, которое работает через разделяемую память.

```
__global__ void BitonicSortShared(int* nums, int n, int iterShift, int iterToStart){

    __shared__ int shared[_sizeShared(SharedSize)];

    int idx = blockIdx.x * SharedSize;
    int offset = SharedSize * gridDim.x;
    int prev = idx - (idx & (1 << iterShift));
    int j;
    int iter, localStep;
    int comp = ASCENDING;

    for(int i = idx; i < n; i += offset){
        if ((i >> iterShift) != (prev >> iterShift)){
            comp ^= 1;
        }
        prev = i;

        shared[_idxShared(threadIdx.x)] = nums[i + threadIdx.x];
        shared[_idxShared(ThreadPerBlock + threadIdx.x)] = nums[i + ThreadPerBlock + threadIdx.x];
        __syncthreads();

        for (iter = 2 * iterToStart; iter > 1; iter /= 2){
            for (localStep = iter / 2; localStep > 0; localStep /= 2){

                j = (2 * threadIdx.x) - (threadIdx.x & (localStep - 1));
                if( comp == ASCENDING ){
                    compASC(shared + _idxShared(j), shared + _idxShared(localStep + j));
                }
                else{
                    compDESC(shared + _idxShared(j), shared + _idxShared(localStep + j));
                }
                __syncthreads();

            }
        }

        nums[i + threadIdx.x] = shared[_idxShared(threadIdx.x)];
        nums[i + ThreadPerBlock + threadIdx.x] = shared[_idxShared(ThreadPerBlock + threadIdx.x)];
    }
}
```

Сначала объясню идейно. На конкретной фиксированной итерации алгоритма, после того как мы получили битоническую последовательность будет $\log N$ раз применяться сортирующий фильтр с разным **localStep**. Начнет с самого большого шага и закончит шагом равным 1. Можно заметить, что при уменьшении шага все элементы внутри одного “блока” не обращаются к элементам другого блока. Это свойство алгоритма битонической сортировки можно использовать следующим образом. Начиная с какого-то большого шага, как только элементы одного “блока” поместятся в разделяемую память, загрузим их туда и отсортируем независимо от остальных элементов массива. По сути, мы откусываем куски от массива на конкретной итерации и через shared memory их сортируем.
Теперь объясню код ядра. Всё тот же параметр **iterShift**. Без него не обойтись. Остальные два параметра помещаются внутри самого ядра. Сначала загружаем “блок” данных в память и определяем, каким образом нам нужно его отсортировать через параметр **iterShift**. После этого просто применяем наши фильтры, пока не достигнем шага 1. Чтобы не было гонки потоков и для

безопасности алгоритма используем синхронизацию. Она синхронизирует только один warp. После того, как “блок” данных отсортирован, выгружаем его из shared memory обратно в наш массив. В коде видно, что при обращении к shared memory как раз используются оба вышеупомянутых макроса.

Так же из особенностей алгоритма стоит отметить, что если длина массива не является степенью двойки, то алгоритм не отработает правильно. Чтобы решить эту проблему в main высчитывается ближайшая степень двойки, больше либо равная чем размер массива. Так как нужно отсортировать массив по возрастанию, то недостающие элементы заполняются значением INT_MAX. После сортировки рассматриваются только первые n чисел. Остальные числа это как раз таки наши INT_MAX значения.

Результаты

<<<64, 256>>>

N	10^6	10^7	10^8
Time	**0.116478**	**2.55003**	**24.7329**

<<<64, 512>>>

N	10^6	10^7	10^8
Time	**0.116622**	**2.54912**	**24.7362**

<<<32, 512>>>

N	10^6	10^7	10^8
Time	**0.118713**	**2.56892**	**24.7477**

<<<32, 256>>>

N	10^6	10^7	10^8
Time	**0.116521**	**2.5518**	**24.7438**

<<<1, 32>>>

N	10^6	10^7	10^8
Time	**1.30651**	**30.8003**	**-**

<<<2, 32>>>

N	10^6	10^7	10^8
Time	**0.705477**	**16.5289**	**-**

<<<1, 64>>>

N	10^6	10^7	10^8
Time	**0.704965**	**16.512**	**-**

<<<2, 64>>>

N	10^6	10^7	10^8
Time	**0.403658**	**9.3678**	**96.3225**

<<<4, 64>>>

N	10^6	10^7	10^8
Time	**0.254237**	**5.82141**	**59.0718**

<<<8, 64>>>

N	10^6	10^7	10^8
Time	**0.178382**	**4.02424**	**40.2166**

<<<4, 128>>>

N	10^6	10^7	10^8
Time	**0.178339**	**4.02348**	**40.2139**

CPU with –O3 optimization

N	10^6	10^7	10^8
Time	**0.20175**	**2.86284**	**27.0183**

CPU

N	10^6	10^7	10^8
Time	**0.607808**	**12.0459**	**120.67**

Выводы

Оцениваю эту лабораторную работу как сложную. Изначально она казалась более дружелюбной. CPU версию алгоритма написал через 2-3 часа после лекции про Bitonic Sort. На написание Global версии ядра ушло около двух дней. Была не очевидна “игра” со сдвигами. На написание Shared версии ядра и его дебаг ушло ещё пару дней.

Битоническая сортировка очень хорошо ложится на архитектуру GPU. Каждая задача решается независимо от остальных. Потоки нигде не пересекаются при обращении к данным. Коэффициент распараллеливания близок к 1. Единственный минус это трудность реализации алгоритма на GPU, так как приходится возиться с огромным количеством сдвигов. Ошибиться довольно легко.