МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №3
по курсу «Параллельная обработка данных»

Сортировка чисел на GPU. Свертка, сканирование, гистограмма.

Выполнил: М. А. Инютин
Группа: М8О-407Б-19
Преподаватель: А. Ю. Морозов

Москва, 2022

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.

Требуется реализовать карманную сортировку для чисел типа float. Должны быть реализованы:

- Алгоритм гистограммы, с использованием атомарных операций;
- Алгоритм свертки для любого размера, с использованием разделяемой памяти;
- Алгоритм сканирования для любого размера, с использованием разделяемой памяти;
- Алгоритм чет-нечет сортировки для карманов.

Ограничения: $n \leq 10^8$

Программное и аппаратное обеспечение

Характеристики графического процессора:

- Compute capability: 6.1
- Наименование: NVIDIA GeForce GTX 1050 Ti
- Графическая память: 4238802944
- Разделяемая память на блок: 49152
- Количество регистров на блок: 65536
- Максимальное количество потоков на блок: (1024, 1024, 64)
- Максимальное количество блоков: (2147483647, 65535, 65535)
- Константная память: 65536
- Количество мультипроцессоров: 6

Характеристики системы:

- Процессор: «Intel i7-9750H (12) @ 4.500GHz»
- Память: два модуля «Kingston KHX2666C15S4/16G 16384 MB @ 2667MHz»
- SSD: «KINGSTON RBUSNS8154P3256GJ (E8FK11.C) 256 GB»

Программное обеспечение:

- ОС: «Ubuntu 20.04.5 LTS x86_64»
- Текстовый редактор: «Atom 1.58.0»
- Компилятор: «nvcc: Cuda compilation tools, release 10.1, V10.1.243»

Метод решения

Алгоритм карманная сортировки:

1. Разбить все данные на n_split карманов. Здесь $n\_split = \frac{n-1}{split\_size} + 1$, $split\_size$ — размер кармана.
2. Если рядом оказываются два кармана, суммарный размер которых не первышает размер кармана $split\_size$, то они объединяются.
3. Для карманов, размеров которых первышает $split\_size$ рекурсивно вызывается карманная сортировка для дальнейшего разбиения, для остальных карманов вызывается чет-нечет сортировка внутри каждого кармана.

Первый шаг алгоритма требует вычисления максимума и минимума, построения гистограммы и вычисления массива префиксных сумм.

Максимум и минимум могут быть вычислены параллельно с помощью редукции, основная идея которой заключается в другом порядке операций — вместо того, что последовательно сравнивать максимум и элемент, мы будет сравнивать все пары чисел, затем все пары результатов сравнения и так далее, алгоритм похож на построение дерева отрезков за $O(n)$.

Для параллельного построения гистограммы требуется выполнять атомарное сложение, чтобы избежать гонки потоков — `atomicAdd`.

Префиксные суммы вычисляютя параллельно с помощью алгоритма `blelloch scan`. Он подробно описан в [1]. Идея в том, чтобы вычислить префиксные суммы для одного блока, который помещается в разделяюмую память, затем добавить к каждому элементу следующего блока суммы всех предыдущих блоков. Чтобы сделать это эффективно, необходимо вызвать рекурсивно `scan` для массива сумм блоков.

Второй шаг алгоритма выполняется на центральном процессоре обычным проходом в цикле.

Для чет-нечет сортировки для кармана необходимо сначала загрузить его в разделяемую память, затем каждый поток сравнивает пары элементов (начиная с чётного или нечётного индекса в зависимости от фазы алгоритма) и меняет их в случае неупорядоченности по неубыванию.

Описание программы

Вычисление максимума и минимума с помощью редукции реализовано в функциях `max_val`, `min_val` и ядрах `max_val_reduction`, `min_val_reduction` соответственно. Пока размер массива первышает размер одного блока, вызывается ядро свёртки, результат выполнения каждого блока записывается по индексу самого блока в новый массив.

Построение гистограммы реализовано в функции `build_hist`, она вычисляет максимальный элемент массива и вызывает ядро `cnt_kernel_gl` — построение гистограммы в глобальной памяти. При желании можно реализовать параллельное построение гистограммы для массивов из небольних элементов на разделяемой памяти.

Вычисление массива префиксных сумм реализовано в шаблонной функции `scan`, которая вызывает ядро `scan_kernel`, оно же проходит по всем блокам и в каждом вызывает ядро `scan_block`. Если размер массива превышает максимальный размер блока, то `scan` вызывается рекурсивно для массива сумм каждого блока, после чего вызывается ядро `update_kernel`, прибавляющее эти значения к элементам блоков.

Чет-нечет сортировка реализована в ядре `odd_even_kernel`. Каждый поток хранит флаг `flag`, отвечающий за фазу алгоритма — чётную или нечётную. Так как каждый раз проверять отсортированность массива долго, алгоритм проходит по всем n фазам.

Для карманной сортировки реализованы ядра `key_kernel` и `distribution_kernel`. Первое нужно для вычисления номера кармана каждого элемента, а второе для распределения элементов по карманам. Функция `process_pockets` реализует второй шаг алгоритма, описанного ранее. Сами карманы хранятся в структуре `pocket` из двух полей pos и len — индекса начала и размера кармана.

Результаты

Замеры времени работы CPU и ядер с различными конфигурациями.

Конфигурация	$n = 10$, мс	$n = 100$, мс	$n = 1000$, мс	$n = 10^4$, мс	$n = 10^5$, мс	$n = 10^6$, мс	$n = 10^7$, мс	$n = 10^8$, мс
CPU	0.005	0.010	0.388	5.545	50.242	496.332	5237.005	4845.027
(1, 32)	0.171	0.245	0.475	3.582	33.010	436.983	7139.787	25199.008
(32, 32)	0.177	0.184	0.235	0.322	1.304	160.804	5116.830	6499.184
(1024, 32)	0.187	0.216	0.216	0.311	0.578	158.028	5247.908	6554.873
(1, 256)	0.364	0.170	0.318	2.244	17.346	176.823	2686.780	8845.993
(64, 256)	0.291	0.191	0.274	0.332	2.043	16.032	1655.158	6175.717
(256, 256)	0.171	0.190	0.263	0.328	1.301	14.278	1630.894	6315.127
(1, 1024)	0.205	0.337	0.788	2.919	30.126	284.368	2308.319	7051.791
(16, 1024)	0.174	0.224	0.391	0.744	5.591	52.069	472.682	6372.650
(64, 1024)	0.175	0.185	0.387	1.370	4.346	61.167	409.574	6044.309

Профилировка nvprof

Конфигурация ядра — (64, 256), размер выходных данных — $n = 10^6$.

Конфликты банков памяти

```
engineerxl-GF63-Thin-9RCX:~/Study/7 term/pgp/lab5\$ sudo nvprof -e
    shared_ld_bank_conflict,shared_st_bank_conflict ./solution < tests/6.in
==11890== NVPROF is profiling process 11890, command: ./solution
51.783
==11890== Profiling application: ./solution
==11890== Profiling result:
==11890== Event result:
Invocations Event Name Min Max Avg Total
Device "NVIDIA GeForce GTX 1050 Ti (0)"
    Kernel: void odd_even_kernel<float>(float*, float*, pocket*, int)
          1 shared_ld_bank_conflict 15290717 15290717 15290717 15290717
          1 shared_st_bank_conflict 31341 31341 31341 31341
    Kernel: void max_val_reduction<float>(float const *, int, float*)
          3 shared_ld_bank_conflict 0 0 0 0
          3 shared_st_bank_conflict 0 0 0 0
    Kernel: void key_kernel<float>(float const *, int, int*, float, float, int)
          1 shared_ld_bank_conflict 0 0 0 0
          1 shared_st_bank_conflict 0 0 0 0
    Kernel: void cnt_kernel_gl<int>(int const *, int, int*)
          1 shared_ld_bank_conflict 0 0 0 0
          1 shared_st_bank_conflict 0 0 0 0
    Kernel: void scan_kernel<int>(int*, int, int*)
          2 shared_ld_bank_conflict 315 48000 24157 48315
          2 shared_st_bank_conflict 189 28800 14494 28989
    Kernel: void min_val_reduction<float>(float const *, int, float*)
          3 shared_ld_bank_conflict 0 0 0 0
          3 shared_st_bank_conflict 0 0 0 0
    Kernel: void distribution_kernel<float>(float const *, float*, int, int*, float,
        float, int)
          1 shared_ld_bank_conflict 0 0 0 0
          1 shared_st_bank_conflict 0 0 0 0
    Kernel: void max_val_reduction<int>(int const *, int, int*)
          3 shared_ld_bank_conflict 0 0 0 0
          3 shared_st_bank_conflict 0 0 0 0
    Kernel: void update_kernel<int>(int*, int, int const *)
          1 shared_ld_bank_conflict 0 0 0 0
          1 shared_st_bank_conflict 0 0 0 0
```

Время выполнения

```
engineerxl-GF63-Thin-9RCX:~/Study/7 term/pgp/lab5\$ sudo nvprof ./solution < tests/6.
    in
==13276== NVPROF is profiling process 13276, command: ./solution
15.040
==13276== Profiling application: ./solution
==13276== Profiling result:
            Type Time(%) Time Calls Avg Min Max Name
 GPU activities: 73.86% 10.466ms 1 10.466ms 10.466ms 10.466ms void odd_even_kernel<
     float>(float*, float*, pocket*, int)
                 12.35% 1.7503ms 1 1.7503ms 1.7503ms 1.7503ms void distribution_kernel
                     <float>(float const *, float*, int, int*, float, float, int)
                  2.62% 371.11us 5 74.222us 736ns 348.23us [CUDA memcpy DtoH]
                  2.57% 363.56us 11 33.050us 1.0880us 85.602us [CUDA memcpy DtoD]
                  2.49% 352.26us 2 176.13us 2.4000us 349.86us [CUDA memcpy HtoD]
                  1.36% 192.16us 3 64.054us 2.0800us 186.37us void max_val_reduction<
                     float>(float const *, int, float*)
                  1.35% 191.72us 3 63.905us 2.0800us 185.83us void min_val_reduction<
                     float>(float const *, int, float*)
                  1.35% 190.76us 3 63.585us 2.1120us 184.93us void max_val_reduction<
                     int>(int const *, int, int*)
                  1.09% 154.02us 1 154.02us 154.02us 154.02us void cnt_kernel_gl<int>(
                     int const *, int, int*)
                  0.66% 94.018us 1 94.018us 94.018us 94.018us void key_kernel<float>(
                     float const *, int, int*, float, float, int)
                  0.25% 35.489us 2 17.744us 5.2160us 30.273us void scan_kernel<int>(
                     int*, int, int*)
                  0.04% 6.3370us 1 6.3370us 6.3370us 6.3370us void update_kernel<int>(
                     int*, int, int const *)
                  0.01% 1.5360us 2 768ns 512ns 1.0240us [CUDA memset]
      API calls: 87.17% 105.55ms 17 6.2087ms 1.2650us 104.87ms cudaMalloc
                  9.75% 11.803ms 17 694.30us 2.3340us 10.507ms cudaFree
                  2.81% 3.3980ms 18 188.78us 4.3530us 1.8607ms cudaMemcpy
                  0.13% 153.86us 97 1.5860us 88ns 78.939us cuDeviceGetAttribute
                  0.08% 95.124us 16 5.9450us 2.8720us 12.377us cudaLaunchKernel
                  0.02% 24.433us 1 24.433us 24.433us 24.433us cuDeviceGetName
                  0.01% 16.157us 3 5.3850us 246ns 11.702us cudaMemset
                  0.01% 7.9670us 2 3.9830us 2.4570us 5.5100us cudaEventRecord
                  0.01% 7.7010us 1 7.7010us 7.7010us 7.7010us cuDeviceGetPCIBusId
                  0.01% 7.3660us 2 3.6830us 505ns 6.8610us cudaEventCreate
                  0.00% 4.5090us 33 136ns 79ns 403ns cudaGetLastError
                  0.00% 3.3050us 1 3.3050us 3.3050us 3.3050us cudaEventSynchronize
                  0.00% 3.2690us 1 3.2690us 3.2690us 3.2690us cudaDeviceSynchronize
                  0.00% 2.0950us 1 2.0950us 2.0950us 2.0950us cudaEventElapsedTime
                  0.00% 2.0880us 2 1.0440us 436ns 1.6520us cudaEventDestroy
                  0.00% 1.4160us 3 472ns 104ns 1.1280us cuDeviceGetCount
                  0.00% 537ns 2 268ns 94ns 443ns cuDeviceGet
                  0.00% 224ns 1 224ns 224ns 224ns cuDeviceTotalMem
                  0.00% 152ns 1 152ns 152ns 152ns cuDeviceGetUuid
```

Я реализовывал простую версию алгоритма `scan`, поэтому в ней есть конфликты банков памяти. Так как в чет-нечет сортировке сравниваются соседние элементы, то часто один поток обращается к паре элементов, которые находятся в разных банках памяти, поэтому и возникают конфликты. Она же занимает больше всего времени, так как выполняется фиксированное число фаз даже если массив сортируется за две-три фазы.

Выводы

Сортировка используется во многих алгоритмах, например она необходима для применения бинарного поиска и для построения выпуклой оболочки алгоритмом Грэхема.

Средняя сложность алгоритма $O(n)$, так как карманная сортировка основывается на идее, что все карманы будут примерно равны по размеру.

Результаты показали, что при полученной сложности центральный выигрывает у графического при больших n. При параллельном вычислении возрастает константа у функций поиска максимума и минимума, построения массива префиксных сумм: $O(n \cdot \log k)$, где k — количество потоков на блок. Так же эти функции требуют дополнительную память, на выделение которой тоже тратится немало времени.

Список литературы

[1] *Chapter 39. Parallel Prefix Sum (Scan) with CUDA.*
URL: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda
(дата обращения: 01.11.2022)