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.

Вариант 6. Карманная сортировка с битонической сортировкой в каждом кармане.

Выполнил: А.В. Синявский
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 (обязательно отразить в отчете).

Вариант 6. Карманная сортировка с битонической сортировкой в каждом кармане.

Входные данные. В первых четырех байтах записывается целое число n -- длина массива чисел, далее следуют n чисел типа заданного вариантом (float).

Выходные данные. В бинарном виде записывают n отсортированных по возрастанию чисел.

Программное и аппаратное обеспечение

Nvidia GeForce GTX 660

Compute capability: 3.0
Графическая память: 2048MB
Регистров на блок: 65536
Нитей на блок: 1024
Мультипроцессоров: 5
Всего ядер: 960

Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz

Тактовая частота: 3.4 GHz
Кэш-память: 6 MB

Оперативная память

Объём: 8 GB

Жёсткий диск

Объём: 2 TB

Программное обеспечение

OS: Windows 10
IDE: Visual Studio Code
CUDA: v10.2 nvcc

Метод решения

Для реализации карманной сортировки нам нужно разбить данные на карманы такого размера, чтобы каждый карман помещался в разделяемую память блока. Данные в карманах должны быть отсортированы в масштабе карманов. Затем для каждого кармана можно применить любую сортировку (в данном случае битоническую).
Для разделения данных на карманы определим размер сплита – «маленького кармана». Присвоим каждому значению номер сплита, в который это значение попадает, затем построим по значениям номеров сплитов гистограмму, а по ней префиксную сумму (скан). При помощи префиксной суммы можно перегруппировать значения, используя сортировку подсчётом. Получим отсортированные в масштабе сплитов данные.
Затем попытаемся объединить сплиты в карманы так, чтобы значений в каждом кармане было как можно больше, но чтобы для каждого кармана можно было реализовать битоническую сортировку на разделяемой памяти. Если же сплит получился изначально слишком большим, рекурсивно выполним разбиение для данных, попавших в этот сплит. После успешного разбиения выполним битоническую сортировку каждого кармана. Так как все значения в i-ом кармане благодаря сортировке подсчётом гарантированно будут меньше чем любое значение из i+1-ого кармана, рекурсивно вызывать битоническую сортировку не понадобится.

Описание программы

1. Макрос CSC
   Проверяет, с каким статусом завершаются CUDA-операции, и в, случае ошибки, выводит на стандартный поток ошибок debug-информацию.

2. _reduce
   Ядро, выполняющее алгоритм редукции на отдельных блоках

3. My_reduce_rec
   Функция, определяющая необходимое для редукции число блоков в зависимости от длины данных, и запускающая ядро редукции. Если Данных оказалось много, и необходимо продолжить алгоритм редукции на результатах блоков, функция рекурсивно запускает ядро, выбирая число блоков в зависимости от того, сколько блоков было использовано на предыдущем витке рекурсии. Функция возвращает число – результат редукции. В зависимости от флага это будет максимум или минимум.
4. My_reduce
   Изменяет размер данных, чтобы он был кратен размеру блока, и в зависимости от того, ищем мы максимум или минимум, дополняет данными минимальным

или максимальным значением типа float. Вызывает рекурсивную функцию, описанную выше, на обработанных данных и возвращает результат.

5. Hist
   Обычная гистограмма на глобальной памяти (т.к. число сплитов может не влезть в разделяемую память) с использованием атомарных операций
6. Count_sort
   Сортировка подсчётом на глобальной памяти, использующая данные префиксной суммы (exclusive scan из библиотеки thrust)

7. bucketCalc
   Рекурсивная функция, строящая и возвращающая вектор индексов карманов.
   Вычисляет минимальный и максимальный элемент при помощи функции my_reduce, вычисляет индексы сплитов (hist + thrust::exclusive_scan), и пытается собрать сплиты в карманы. Если сплит слишком большой – функция рекурсивно вызывается для этого сплита
8. shared_BitonicSort
   Битоническая сортировка на разделяемой памяти, сортирующая значения внутри каждого кармана.
9. bucketSort
   Функция – обёртка для предыдущих двух функций, по сути последовательно вызывает функцию для разбиения на карманы и функцию для сортировки внутри этих карманов
10. main
    Основная функция, работающая на процессоре, отвечает за ввод и вывод данных, часть работы с памятью и запуск сортировки

Результаты

Размер теста	GPU
10K	1.73 мс
100K	2.85мс
1M	18.99 мс
10M	190 мс
100M	1950 мс

Из результатов очевидно, что при больших данных сортировка начинает работать за O(n), что хоть и приемлемо для прохождения чекера, имеет смысл разобраться в природе этой проблемы. Для анализа используем утилиту nvprof.

```
==11250== Profiling application: ./sort
==11250== Profiling result:
==11250== Event result:
Invocations                                Event Name         Min         Max         Avg
Device "GeForce GT 545 (0)"
        Kernel: count_sort(float*, float*, int*, float, float, int, int)
          1                            divergent_branch           1           1           1
          1                  global_store_transaction        8787        8787        8787
          1                   l1_shared_bank_conflict           0           0           0
          1                         l1_local_load_hit           0           0           0
        Kernel: shared_BitonicSort(float*, int*, int)
          1                            divergent_branch       11655       11655       11655
          1                  global_store_transaction         654         654         654
          1                   l1_shared_bank_conflict       39753       39753       39753
          1                         l1_local_load_hit           0           0           0
        Kernel: void thrust::system::cuda::detail::bulk_::detail::launch_by_value<unsigned i
a::detail::bulk_::concurrent_group<thrust::system::cuda::detail::bulk_::agent<unsigned long=
il::exclusive_scan_n, thrust::tuple<thrust::system::cuda::detail::bulk_::detail::cursor<unsi
ust::null_type, thrust::null_type>>>>(unsigned long=3)
          1                            divergent_branch           2           2           2
          1                  global_store_transaction          30          30          30
          1                   l1_shared_bank_conflict           0           0           0
          1                         l1_local_load_hit          60          60          60
        Kernel: hist(float*, int*, float, float, int, int)
          1                            divergent_branch           1           1           1
          1                  global_store_transaction           0           0           0
          1                   l1_shared_bank_conflict           0           0           0
          1                         l1_local_load_hit           0           0           0
        Kernel: _reduce(float*, int, bool)
          4                            divergent_branch           1           5           3
          4                  global_store_transaction           3           6           4
          4                   l1_shared_bank_conflict           0           0           0
          4                         l1_local_load_hit           0           0           0
user67@server-i72:~/pod_lab4$
```

```
user67@server-i72:~/pod_lab4$ nvprof ./sort < test.data
==12624== NVPROF is profiling process 12624, command: ./sort
==12624== Profiling application: ./sort
==12624== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 34.38%  459.47us         1  459.47us  459.47us  459.47us  count_sort(float*, float*, int*
, float, float, int, int)
 31.95%  426.97us         1  426.97us  426.97us  426.97us  hist(float*, int*, float, float
, int, int)
 25.56%  341.51us         1  341.51us  341.51us  341.51us  shared_BitonicSort(float*, int*
, int)
  2.73%  36.522us         4  9.1300us  5.9610us  12.470us  _reduce(float*, int, bool)
  2.41%  32.258us         6  5.3760us     800ns  7.8410us  [CUDA memcpy HtoD]
  2.06%  27.489us         5  5.4970us  2.9760us  9.1850us  [CUDA memcpy DtoH]
  0.91%  12.126us         1  12.126us  12.126us  12.126us  void thrust::system::cuda::detail
::bulk_::parallel_group<thrust::system::cuda::detail::bulk_::concurrent_group<thrust::system
ure<thrust::system::cuda::detail::scan_detail::exclusive_scan_n, thrust::tuple<thrust::syste
, thrust::null_type, thrust::null_type, thrust::null_type, thrust::null_type>>>>(unsigned lo
```

Как можно видеть, в ядре битонической сортировки происходит довольно много конфликтов банков памяти (тест состоит из 10К элементов). Также высока дивергенция потоков, так что данное ядро – самое вероятное «бутылочное горлышко» в программе. Действительно, профилирование на различных тестовых данных показывает следующее:

10K:

Time(%)	Time	Calls	Avg	Min	Max	Name
34.36%	459.99us	1	459.99us	459.99us	459.99us	count_sort(float*, float*, in
32.02%	428.61us	1	428.61us	428.61us	428.61us	hist(float*, int*, float, flo
25.51%	341.52us	1	**341.52us**	341.52us	341.52us	**shared_BitonicSort**(float*, in

100K:

Time(%)	Time	Calls	Avg	Min	Max	Name
52.67%	2.9104ms	1	**2.9104ms**	2.9104ms	2.9104ms	**shared_BitonicSort**(float*, in
19.01%	1.0506ms	1	1.0506ms	1.0506ms	1.0506ms	count_sort(float*, float*, in
16.33%	902.41us	1	902.41us	902.41us	902.41us	hist(float*, int*, float, flo

1M:

Time(%)	Time	Calls	Avg	Min	Max	Name
68.50%	28.688ms	1	**28.688ms**	28.688ms	28.688ms	**shared_BitonicSort**(float*, in
9.85%	4.1238ms	1	4.1238ms	4.1238ms	4.1238ms	count_sort(float*, float*, in
6.51%	2.7279ms	1	2.7279ms	2.7279ms	2.7279ms	hist(float*, int*, float, flo

10M:

Time(%)	Time	Calls	Avg	Min	Max	Name
63.52%	286.90ms	1	**286.90ms**	286.90ms	286.90ms	**shared_BitonicSort**(float*, int*, int)
13.90%	62.780ms	1	62.780ms	62.780ms	62.780ms	count_sort(float*, float*, int*, float, float, int, int)
8.91%	40.246ms	1	40.246ms	40.246ms	40.246ms	hist(float*, int*, float, float, int, int)

В идеальной ситуации (без конфликтов банков) данная сортировка работала бы за O(log^2(n)), что конечно лучше, чем O(n). Большое число конфликтов связано с тем, что в ходе алгоритма мы сравниваем числа на расстоянии, кратном степени двойки. Исправить эту ситуацию можно было бы каким-то хитрым добавлением фиктивного элемента в данные, но я посчитал текущую производительность полного алгоритма сортировки приемлемой, с учётом того, что его часть всё равно выполняется на процессоре.

Выводы

Проделав работу, я изучил основные алгоритмы обработки данных на GPU: редукцию, гистограмму, скан (префиксная сумма), а также реализовал по сути 2,5 сортировки: битоническую, сортировку подсчётом, и карманную, использующую две предыдущие. Также я впервые после первого курса воспользовался подключением к серверу по shh, познакомился с утилитой nvprof, проанализировал с её помощью свой код и выявил его недостатки.