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: А. В. Куликов
Группа: 8О-408Б
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. Исследование производительности программы с помощью утилиты nvprof.

Вариант 4. Сортировка чет-нечет.
Требуется реализовать блочную сортировку чет-нечет для чисел типа int.
Должны быть реализованы:

- Алгоритм чет-нечет сортировки для предварительной сортировки блоков.
- Алгоритм битонического слияния, с использованием разделяемой памяти.

Ограничения: n ≤ 16 * 10^6

Программное и аппаратное обеспечение

Видеокарта	GeForce GTX 1650
Compute capability	7.5
Графическая память	3911 Мб
Разделяемая память	48 Кб
Константная память	64 Кб
Количество регистров на блок	65536
Максимальное кол-во блоков	2147483647*65535*65535
Максимальное кол-во нитей в блоке	1024
Кол-во мультипроцессоров	16
Ядер CUDA	896

Процессор	AMD Ryzen 5 3550H
ОЗУ	8 Гб
ЖД	

Операционная система	Ubuntu 20.04.6 LTS
IDE	VS Code
Компилятор	nvcc V10.1, mpi V3.3.2

Метод решения

Программа считывает данные для сортировки из стандартного потока ввода.

Затем производится предварительная сортировка: обычной сортировкой чет-не-чет сортируются числа в каждом блоке. Далее производятся итерации блочной чет-не-чет сортировки в количестве равном числу блоков. Каждая такая итерация представляет из себя попарное битоническое слияние двух соседних блоков начиная с первого, затем попарное битоническое слияние двух соседних блоков начиная со второго. Битоническое слияние представляет из себя последовательность условных swap-ов в определенном порядке.

Перед началом сортировки все данные считываются в разделяемую память.
Все операции по сравнению и обмену значениями производятся в разделяемой памяти.
По звершению операций данные из разделяемой памяти записываются в глобальную память.

После сортировки данных программа записывает отсортированные данные в стандартный поток вывода.

Описание программы

Вся программа реализована в одном файле main.cu. Ввод\вывод реализован прямо в функции main. Саму сортировку производит функция block_odd_even_sort. В программе реализованы два основных ядра и одно вспомогательное: sort_blocks производит сортировку в рамках каждого блока, merge производит попарное битоническое слияние. Вспомогательное ядро int_memset позволяет инициализировать участок памяти определенным значением.

Исследование производительности программы с помощью утилиты nvprof

```
==7216== Profiling application: ./lab5
==7216== Profiling result:
==7216== Event result:
Invocations                                Event Name         Min
Max         Avg
Device "GeForce GT 545 (0)"
        Kernel: sort_blocks(int*)
          1                    l1_shared_bank_conflict    19159053
19159053    19159053
          1                             divergent_branch           0
0           0
        Kernel: int_memset(int*, int, int)
          1                    l1_shared_bank_conflict           0
0           0
          1                             divergent_branch           0
0           0
```

```
        Kernel: merge(int*, int, int)
       1954                   l1_shared_bank_conflict      171834
193086      185062
       1954                            divergent_branch           0
0           0

==7216== Metric result:
Invocations                                   Metric Name
Metric Description         Min         Max         Avg
Device "GeForce GT 545 (0)"
        Kernel: sort_blocks(int*)
          1      shared_load_transactions_per_request  Shared Memory Load
Transactions Per Requ    1.251157    1.251157    1.251157
          1     shared_store_transactions_per_request  Shared Memory Store
Transactions Per Req    1.128955    1.128955    1.128955
          1              gld_transactions_per_request      Global Load
Transactions Per Request    1.001024    1.001024    1.001024
        Kernel: int_memset(int*, int, int)
          1      shared_load_transactions_per_request  Shared Memory Load
Transactions Per Requ    0.000000    0.000000    0.000000
          1     shared_store_transactions_per_request  Shared Memory Store
Transactions Per Req    0.000000    0.000000    0.000000
          1              gld_transactions_per_request      Global Load
Transactions Per Request    0.000000    0.000000    0.000000
        Kernel: merge(int*, int, int)
       1954      shared_load_transactions_per_request  Shared Memory Load
Transactions Per Requ    1.358048    1.369219    1.363705
       1954     shared_store_transactions_per_request  Shared Memory Store
Transactions Per Req    1.259472    1.314415    1.287064
       1954              gld_transactions_per_request      Global Load
Transactions Per Request    0.995902    1.002049    0.999995
```

К сожалению, совсем избавится от конфликтов банков памяти не удалось. Но удалось улучшить ситуацию, расположив элементы в разделемой памяти особым образом. После каждого 32-го (по количеству потоков в варпе) элемента добавляется один фиктивный. Таким образом при предварительной сортировке все обмены без сдвига происходят без конфликтов банков памяти. При обмене со сдвигом присутствует конфликт 2-го порядка. Ровно в один банк попадают 2 потока варпа и один остается не тронутым. Похожая ситуация наблюдается и с процедурой битонического слияния. Остается надеятся на cuda latency hiding. И судя по всему поэтому конфликты банков памяти и не приводят к катострофической просадке производительности.

Результаты

Тестирование ядер с различными конфигурациями

Маленький файл 1000 чисел

Размер блока	Время
1024	0m 0,224s
512	0m 0,260s
256	0m 0,253s
128	0m 0,263s
64	0m 0,262s
32	0m 0,272s

Лучший результат: 0m 0,224s при размере блока 1024 (т.е при 512 запущенных процессах) процессах.

Средний файл 100000 чисел

Размер блока	Время
1024	0m 0,262s
512	0m 0,296s
256	0m 0,272s
128	0m 0,274s
64	0m 0,307s
32	0m 0,360s

Лучший результат: 0m 0,262s при размере блока 1024 (т.е при 512 запущенных процессах) процессах.

Большой файл 10000000 чисел

Размер блока	Время
1024	0m 15,565s
512	0m 28,937s
256	0m 57,130s
128	1m 54,305s
64	4m 13,761s
32	13m 30,468s

Лучший результат: 0m 15,565s при размере блока 1024 (т.е при 512 запущенных процессах) процессах.

Сравнение с CPU
Маленький файл 1000 чисел
Результат: 0m 0,005s

Средний файл 100000 чисел
Результат: 0m 0,250s

Большой файл 10000000 чисел
Результат: 21m 11,786s

Выводы

Данный алгоритм может быть использован для внешней сортировки данных либо же для распределенной сортировки.

Основные сложности в реализации программы возникли при попытки устранения конфликтов банков разделяемой памяти.

В решении данной задачи программа с использованием CUDA значительно превзошла стандартную реализацию на C.