МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №5
по курсу «Программирование графических процессоров»

Сортировка чисел на GPU. Свертка, сканирование, гистограмма.

Выполнил: А.Ю. Гришин
Группа: 8О-408Б
Преподаватель: А.Ю. Морозов

Москва, 2024

Условие

1. **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).
2. **Вариант 2:** сортировка подсчетом. Диапазон от 0 до $2^{24} - 1$

Программное и аппаратное обеспечение

- Графический процессор
  - Compute capability: 7.5
  - Объем графической памяти: 15,83 ГБ
  - Объем постоянной памяти: 65536 байт
  - Разделяемая память на блок: 49152 байт
  - Количество регистров на блок: 65536
  - Максимальное количество потоков на блок: 1024
  - Количество мультипроцессоров: 40
- Процессор
  - Количество физических ядер: 2
  - Количество логических ядер: 4
  - Частота: 2000 МГц
- Оперативная память
  - Тип оперативной памяти: DDR4
  - Объем: 32 ГБ
- Жесткий диск
  - Объем: 1 ТБ
- Программное обеспечение
  - ОС: Ubuntu 22.04.4 LTS
  - IDE: Visual Studio Code
  - Компилятор: nvcc

Метод решения

Решение данной задачи я начал с формализации алгоритма сортировки подсчетом. Как известно, основная идея сортировки подсчетом заключается в вычислении (подсчете) статистики о количестве элементов, которые встречаются в целевом массиве. Далее, на основе этой статистики формируется уже отсортированный массив. Итого, алгоритм сортировки подсчетом можно разделить на следующие этапы:

1. Формирование гистограммы H. На этом этапе на основе исходного массива A формируется гиастограмма H, которая хранит в себе информацию о том, сколько каждый элемент встречался;
2. Формирование префикс-суммы H'. Далее, для дальнейшего более оптимизированного формирования отсортированного массива необходимо посчитать на основе гистограммы H ее префикс-сумму H', в которой элемент H'_i

теперь хранит в себе информацию о том, сколько элементов не больше чем i находится в исходном массиве A;

3. Последним этапом алгоритма является построение уже отсортированного массива A' на основе исходного массива A и префикс-суммы гистограммы H'.

И так, задача сортировки массива была декомпозирована на 3 последовательных этапа, поэтому теперь задача сводится к нахождению оптимального решения для каждого из этапов в контексте выполнения на GPU.

Формирование гистограммы

Пусть дан исходный массив A, и известно, что элементы в нем не превышают некоторого значения N: $\forall i \ A_i \leq N$. Тогда гистограмму можно реализовать через массив из $N + 1$ элементов, элементы которого описывают, сколько раз встречалось то или иное значение в массиве A:

$$H_i = \left|\{A_j \in A \middle| A_j = i\}\right|$$

Самая наивная реализация, представляющая последовательный обход по всем элементами A_i и добавлению текущего элемента к гистограмме, имеет временную асимтотику $O(N)$, где N – размер исходного массива A, что слишком неоптимально при больших N.

Альтернативным решением может быть использование потоков на GPU. При таком алгоритме каждый из потоков добавляет свой текущий элемент к гистограмме и в случае необходимости переход к следующему. Иными словами, нагрузка распределяется между потоками.

Однако, стоит учитывать, что при добавлении элементов к гистограмме, работа потоков происходит с общей областью памяти, из-за чего возникает необходимость в использовании синхронизированных операций, таких как `atomicAdd`.

Такой алгоритм в общем случае имеет также линейную временную асимтотику, однако, теперь нагрузка распределена между потоками, количество которых может быть вплоть до 10^6, что значительно сокращает итоговое время исполнения.

Подсчет префикс-суммы

Подсчет префикс-суммы является одним из частных случаев алгоритма scan. Пусть $\oplus$ является оператором, определенным на алгебре, состоящим из элементов, которые имеет массив A, и единичного элемента I, который называют **единичным элементом**.

Тогда задача алгоритма scan состоит в том, чтобы на основе массива $A = (a_1, a_2, \dots, a_n)$ посчитать «префикс-сумму» $S = \left(I, I \oplus a_1, (I \oplus a_1) \oplus a_2, \dots, \left((\dots) \oplus a_{n-1}\right) \oplus a_n\right)$.

Наивная реализация данного алгоритма подразумевает последовательный обход всех элементов исходного массива A и подсчете промежуточных значений «свертки». Однако, временная асимтотика такого алгоритма также является линейной $O(N)$ и его работа не может быть распараллелена, что неоптимально при больших N.

Альтернативным решением может послужить алгоритм, автором которого является Гай Блеллок. Данный алгоритм состоит из 2-х последовательных этапов: прямой и обратный ход.

Во время прямого хода строится дерево частичны сумм, начиная с листьев и поднимаясь к корню. На текущем уровне каждый элемент вычисляется как сумма двух элементов предыдущего уровня. В результате, в конце алгоритма на последнем уровне остается один элемент, представляющий сумму всех элементов массива, а на всех предыдущих слоях находятся префикс суммы.

Однако, на данном этапе массив префикс-суммы сформирован не до конца. Для этого необходимо выполнить также и обратный ход. На данном этапе мы спускаемся по дереву вниз, и «свертка» происходит в обратную сторону. Теперь вместо суммы двух элементов, в ходе «свертки» вместо первого элемента устанавливается второй, а вместо второго – сумма двух элементов.

Данный алгоритм примечателен тем, что он позволяет распараллелить работу каждого из шагов во время прямого и обратного хода. Однако, с другой стороны, он подразумевает синхронизацию между шагами во время прямого и обратного хода, что делает невозможным исполнение данного алгоритма на нескольких блоках потоков. С другой стороны, етсь возможность реализовать данный алгоритм в виде нескольких ядер, описывающих один шаг прямого или обратного хода, однако синхронизация между ядрами значительно медленее, чем синхронизация между потоками.

Также, стоит учитывать, что длина исходного массива должна представлять из себя степень двойки, так как построение дерева частичных сумм основано на разбиении массива на пары элементов. Если длина массива не является степенью двойки, его необходимо дополнить нулями до ближайшей степени двойки. Это упрощает вычисления и структуру дерева, обеспечивая корректное выполнение всех шагов.

В результате своей работы алгоритм Blelloch Scan формирует массив префиксных сумм, где каждый элемент на позиции i содержит сумму всех элементов исходного массива от 0 до $i-1$. Также, данный алгоритм уже временную асимптотику $O(\log N)$, что значительно оптимальнее, чем наиваное решение с линейной временной асимтотикой.

Формирование отсортированного массива
Последним этапом является формирование отсортированного массива на основе полученного на прошлом этапе префикс-суммы H'. Решение основано на идее того, что элемент H'_i также показывает и позицию последнего элемента со значением i.

Итого, после установки очередного элемента i необходимо также выполнить уменьшение на единицу элемента H'_i для актуализации позиции для установки для дальнейших элементов с таким же значением.

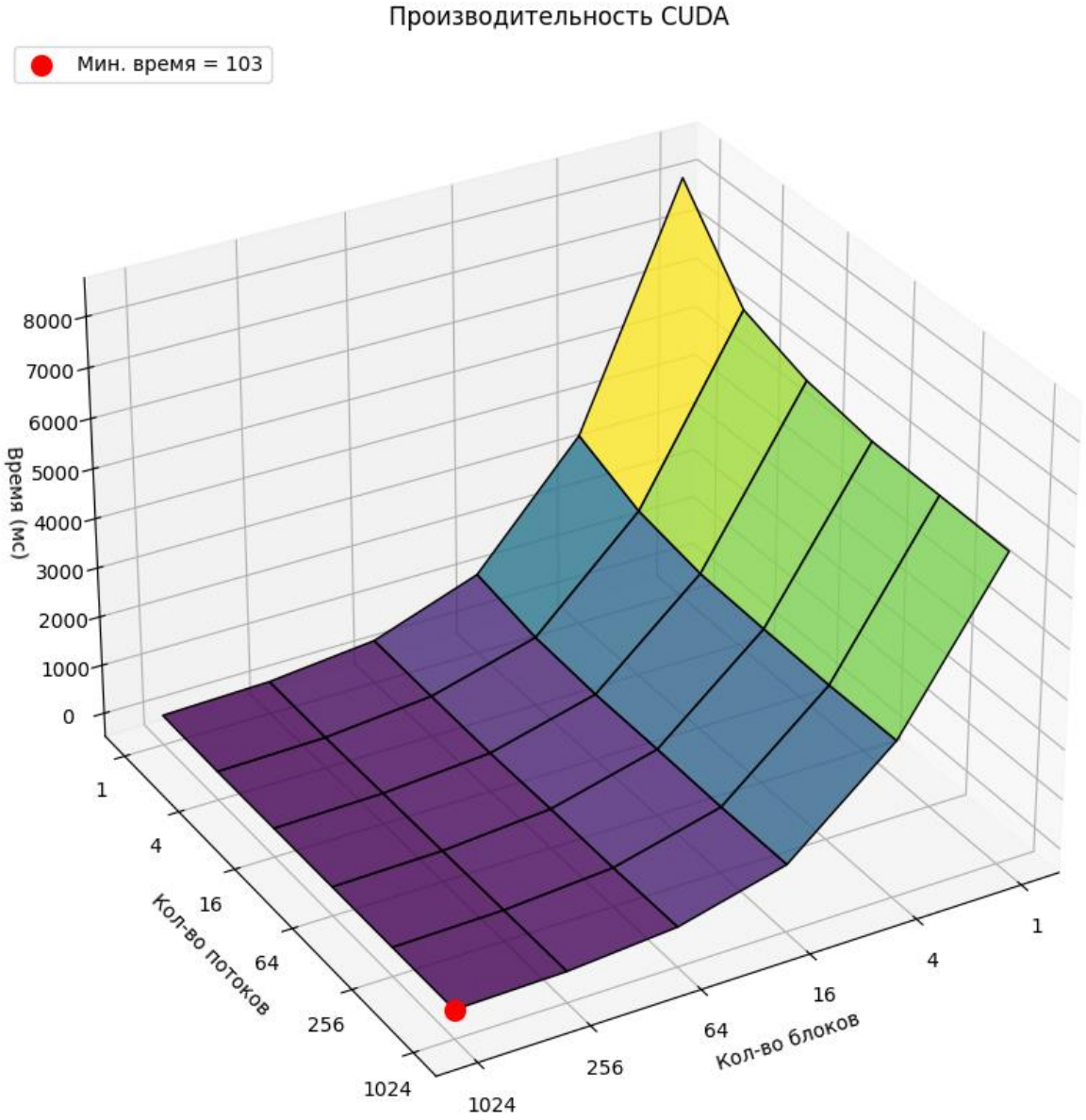
Аналогично с алгоритмом формирования гистограммы, работу данного алгоритма можно распараллелить, однако при изменении значения H'_i стоит использовать синхронизированные операции, такие как `atomicSub`.

Описание программы
Программа состоит из одного файла, содержащего код, написанный на языке CUDA. Код программы состоит из:

1. Типа `TArray`, который представляет комбинацию указателя на область в памяти (на GPU или на хосте), соответствующей массиву, и длины массива;
2. Функции `chekRunResult` для синхронизации ядер и проверки результата их работы на наличие ошибок;
3. Функции-ядра `hist`, содержащей алгоритм построения гистограммы;
4. Функции-ядра `upPhase`, которая описывает этап прямого хода в алгоритме blelloch scan;
5. Функции-ядра `downPhase`, котоаря описывает этап обратного хода в алгоритме blelloch scan;
6. Функции `blellochScan`, которая обьединяет компоненты алгоритма blelloch scan в единую точку вызова;
7. Функции-ядра `buildResult`, которая содержит алгоритм по построению отсортированного массива на основе префикс-суммы гистограммы;
8. Функции `countSort`, которая обьединяет компоненты алгоритма сортировки подсчетом в единую точку вызова;
9. Функции `readArray`, которая считывает входной массив из стандартного потока ввода;
10. Функции `writeArray`, котоаря записывает результирующий массив в стандартный опток вывода;

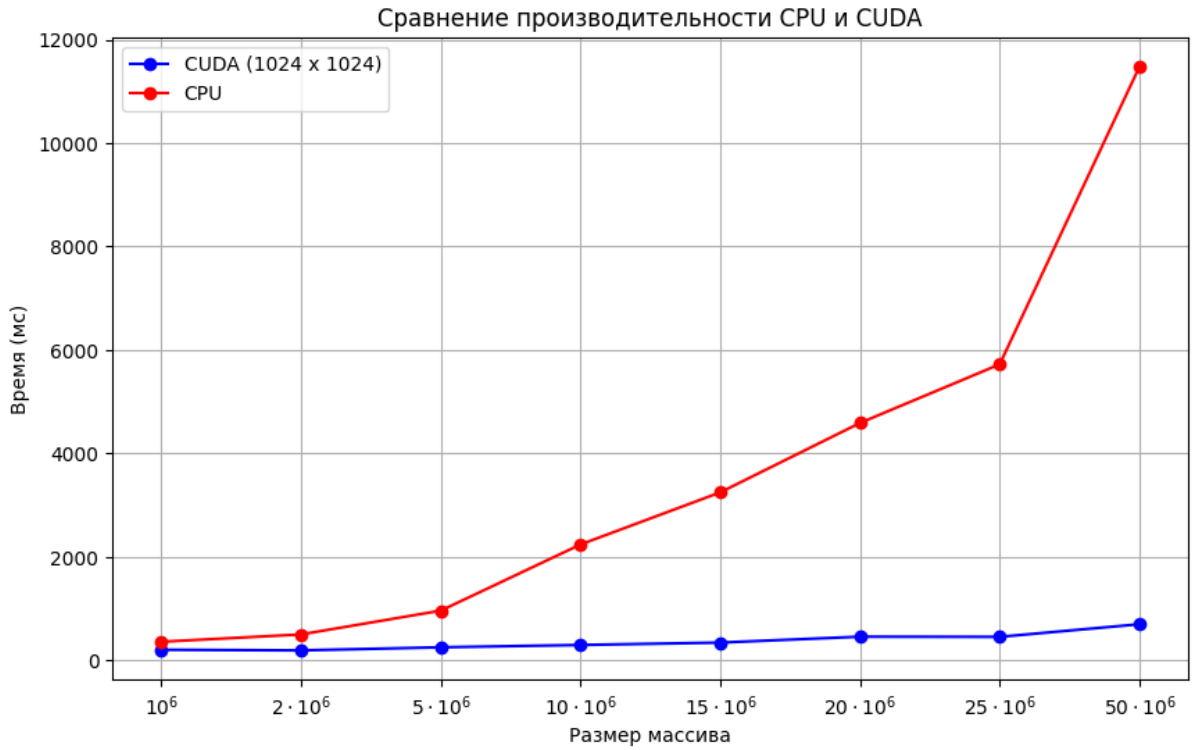
Результаты
1. Зависимость времени выполнения программы от количества используемых потоков. В качестве единиц измерения времени были выбраны миллисекунды (мс). Тестирование производилось на матрице с размерами $5 \cdot 10^6$.

Количество блоков / потоков	1	4	16	64	256	1024
1	8159	3468	1171	407	185	128
4	6455	2942	937	352	147	104
16	6018	2717	867	308	146	104
64	5820	2676	870	319	145	104
256	5786	2651	853	305	145	103
1024	5752	2676	877	322	145	103



2. Сравнение программы на CUDA с 512 блоками и 512 потоками и программы на CPU с одним потоком. В качестве единиц измерения времени были выбраны миллисекунды (мс).

Размер матрицы	Время на CUDA (мс)	Время на CPU (мс)
10^6	203	358
$2 \cdot 10^6$	191	500
$5 \cdot 10^6$	251	960
$10 \cdot 10^6$	296	2239
$15 \cdot 10^6$	342	3247
$20 \cdot 10^6$	456	4589
$25 \cdot 10^6$	453	5723
$50 \cdot 10^6$	697	11476



3. Результат исследования производительности с помощью nvprof (размер массива 1000, сетка потоков 32×32)

```
==291== NVPROF is profiling process 291, command: ./a.out
==291== Profiling application: ./a.out
==291== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   57.97%  280.65ms         1  280.65ms  280.65ms  280.65ms  upPhase(unsigned int*, unsigned int)
                   41.97%  203.19ms         1  203.19ms  203.19ms  203.19ms  downPhase(unsigned int*, unsigned int)
                    0.06%  284.57us         1  284.57us  284.57us  284.57us  [CUDA memset]
                    0.00%  4.4480us         1  4.4480us  4.4480us  4.4480us  hist(unsigned int*, unsigned int*, unsigned i
nt, unsigned int)
                    0.00%  2.8800us         1  2.8800us  2.8800us  2.8800us  buildResult(unsigned int*, unsigned int*, uns
igned int*, unsigned int, unsigned int)
                    0.00%  1.7280us         1  1.7280us  1.7280us  1.7280us  [CUDA memcpy DtoH]
                    0.00%  1.4080us         1  1.4080us  1.4080us  1.4080us  [CUDA memcpy HtoD]
      API calls:   80.71%  483.89ms         4  120.97ms  5.1720us  280.68ms  cudaDeviceSynchronize
                   19.03%  114.07ms         3  38.023ms  5.2260us  113.92ms  cudaMalloc
                    0.09%  532.89us       228  2.3370us     138ns  157.84us  cuDeviceGetAttribute
                    0.08%  454.65us         4  113.66us  6.9730us  347.49us  cudaLaunchKernel
                    0.06%  363.11us         3  121.04us  18.447us  185.54us  cudaFree
                    0.03%  180.20us         2  90.098us  34.294us  145.90us  cudaMemcpy
                    0.01%  34.064us         1  34.064us  34.064us  34.064us  cudaMemset
                    0.00%  15.254us         2  7.6270us  5.0610us  10.193us  cuDeviceGetName
                    0.00%  11.796us         2  5.8980us  3.5520us  8.2440us  cuDeviceGetPCIBusId
                    0.00%  1.6110us         4     402ns     114ns     612ns  cudaGetLastError
                    0.00%  1.1770us         3     392ns     124ns     844ns  cuDeviceGetCount
                    0.00%     995ns         4     248ns     130ns     536ns  cuDeviceGet
                    0.00%     596ns         2     298ns     235ns     361ns  cuDeviceTotalMem
                    0.00%     424ns         2     212ns     173ns     251ns  cuDeviceGetUuid
                    0.00%     345ns         1     345ns     345ns     345ns  cuModuleGetLoadingMode
```

Выводы

В данной лабораторной работе был реализован алгоритм сортировки массива подсчетом, адаптированный для исполнения на GPU. Реализация включала использование нескольких ключевых алгоритмов, каждый из которых оптимизирован для параллельных вычислений.

Алгоритм формирования гистограммы был реализован с использованием атомарных операций `atomicAdd`. Данный алгоритм позволяет подсчитать количество элементов, встречающихся в массиве и является первым этапом сортировки подсчетом.

Для вычисления префикс-суммы гистограммы был применен алгоритм Blelloch scan, основанный на параллельных вычислениях, что значительно ускоряет процесс накопления данных по сравнению с последовательным подходом.

Завершающим этапом стало формирование результирующего массива, где каждый элемент помещался в соответствующую позицию на основе вычисленной префикс-суммы.

Для анализа производительности итогового решения использовалась утилита `nvprof`, которая позволила оценить временные характеристики каждого этапа алгоритма и выявить узкие места. В ходе экспериментов было показано, что использование GPU обеспечивает значительное ускорение при работе с большими объемами данных.

Лабораторная работа позволила углубить понимание принципов реализации сортировки подсчетом и изучить особенности адаптации алгоритмов для параллельного исполнения на GPU. Полученные результаты могут быть использованы для решения более сложных задач, связанных с обработкой больших объемов данных.