МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки 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.А. Почечура*

Группа: *8О-406Б*

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2023

**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).

Все входные-выходные данные являются бинарными и считываются из **stdin** и выводятся в **stdout**.

**Вариант 8. Поразрядная сортировка.**

Требуется реализовать поразрядную сортировку для чисел типа uint.

Должны быть реализованы:

● Алгоритм сортировки через префиксные суммы для одного битового

разряда.

● Алгоритм сканирования для любого размера, с рекурсией и

бесконфликтным использованием разделяемой памяти.

Ограничения: n ≤ 128 \*

**Входные данные.** В первых четырех байтах записывается целое число n – длина массива чисел, далее следуют n чисел типа заданного вариантом.

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

**Программное и аппаратное обеспечение**

**Графический процессор:**

Compute capability: 7.5

Name: Tesla T4

Total Global Memory: 15835398144

Shared memory per block: 49152

Registers per block: 65536

Warp size: 32

Max threads per block: (1024, 1024, 64)

Max block: (2147483647, 65535, 65535)

Total constant memory: 65536

Multiprocessors count: 40

**Процессор:**

vendor\_id : GenuineIntel

cpu family: 6

model: 85

model name: Intel(R) Xeon(R) CPU @ 2.00GHz

stepping: 3

microcode : 0xffffffff

cpu MHz: 2000.184

cache size : 39424 KB

physical id: 0

siblings: 2

core id: 0

cpu cores : 1

apicid: 0

initial apicid: 0

fpu: yes

fpu\_exception: yes

cpuid level: 13

wp : yes

bogomips : 4000.36

clflush size: 64

cache\_alignment : 64

address sizes: 46 bits physical, 48 bits virtual

**Оперативная память и жёсткий диск:**

Memory:

description: System memory

physical id: 0

size: 13GiB

Mem**:** 12982.6 MiB

Storage**:** 55 GiB

**Программное обеспечение:**

*Google Colab*

Операционная система: Ubuntu 22.04.2 LTS

Оболочка: python3

inxi: 3.3.13

**Метод решения**

Поразрядная сортировка на GPU реализуется следующим образом: каждое число будет представлено в двоичной системе счисления и по каждому разряду мы будем считать префиксные суммы с помощью алгоритма *scan* для случая массива произвольной длины. Для этого сначала нужно каждый блок массива прогнать через операцию *blelloch scan*, представить каждый блок в виде одного числа (сумма элементов в массиве) и на получившемся массиве снова выполнить операцию *blelloch scan* для каждого блока уже нового массива. Данный алгоритм нужно выполнять до тех пор, пока размер выходного массива не станет меньше или равен размеру блока потоков. В таком случае нужно будет посчитать префиксные суммы массива (с помощью того же *blelloch scan*; алгоритм во всех случаях *исключающий*) и прибавить соответствующие элементы массива к элементам массива с прошлой итерации (индекс блока соответствует индексу элемента, значение которого нужно прибавить ко всем элементам блока массива с прошлой итерации). Данные действия также проделываем до тех пор, пока не вернёмся к изначальному массиву.

После того, как мы получили массив префиксных сумм для текущего разряда, с его помощью мы можем определить место для каждого числа в соответствии с их значениями на текущем разряде и значением префиксной суммы.

Все шаги нужно повторить 32 раза (именно столько разрядов в типе uint, если числа представляются в двоичном виде), после чего мы получим отсортированный массив.

**Описание программы**

Считываем данные с ввода и копируем массив на ГПУ. Заведём массив *pref,* в котором будем хранить префиксные суммы для текущего разряда (дополняем его до размера, кратного количеству потоков в блоке, если это требуется). Ядро *get\_vals* будем использовать для выделения значений на текущем разряде у каждого числа. В функции *main\_scan* происходит рекурсивный алгоритм, который был подробно описан в прошлом пункте: исходный массив прогоняется через алгоритм *blelloch scan* и преобразуется в массив, каждый элемент которого соответствует сумме элементов в блоке из первичного массива. Данное сжатие продолжается до тех пор, пока размер массива не станет меньше или равен размеру блока. Затем происходит возвращения по рекурсивным вызовам: с помощью функции *put\_sums* происходит прибавление значений в соответствии с тем, к какому блоку какое значение принадлежит (прибавление происходит к каждому элементу). Сжатие массива и подсчёт префиксных сумм производилось с помощью функции *bleloch\_scan.* В ней с использованием разделяемой памяти для каждого блока оптимизированно считались префиксные суммы в соответствии с алгоритмом (в два прохода: вниз и вверх). После получения массива префиксных сумм для текущего разряда вызываем ядро *rank\_swap,* где в соответствии с формулами происходит расстановка элементов на нужные позиции. Алгоритм проходится по всем разрядам типа uint, проделывает каждый раз все вышеописанные операции и в конце мы получаем отсортированный массив, который и подаём на выходные данные.

**Результаты**

Продемонстрируем скорость работы программы в зависимости от входных данных.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество блоков и потоков в блоке | Размер входных данных | n = 15 | n = 1000 | n = 100000 | n = 10000000 |
| << < 1, 32 >> > | | 0.3731899 мс | 12.3873191 мс | 103.12312901 мс | 5436.9120311 мс |
| << < 4, 64 >> > | | 0.2787819 мс | 10.4101914 мс | 87.13289121 мс | 4596.1230129 мс |
| << < 32, 128 >> > | | 0.3012391 мс | 13.201302 мс | 80.12312931 мс | 4259.7182731 мс |
| << < 128, 128 >> > | | 0.4781278 мс | 9.0120315 мс | 54.12391092 мс | 3127.2132314 мс |
| << < 256, 256 >> > | | 0.2217231 мс | 10.12381921 мс | 40.12931021 мс | 2569.991725 мс |
| << < 512, 512 >> > | | 0.3712790 мс | 8.19203912 мс | 38.13212391 мс | 1854.138418 мс |
| << < 1024, 1024 >> > | | 0.4912031 мс | 11.1293019 мс | 38.12301923 мс | 1301.318921 мс |

Сравним работу алгоритма на ГПУ и ЦПУ:

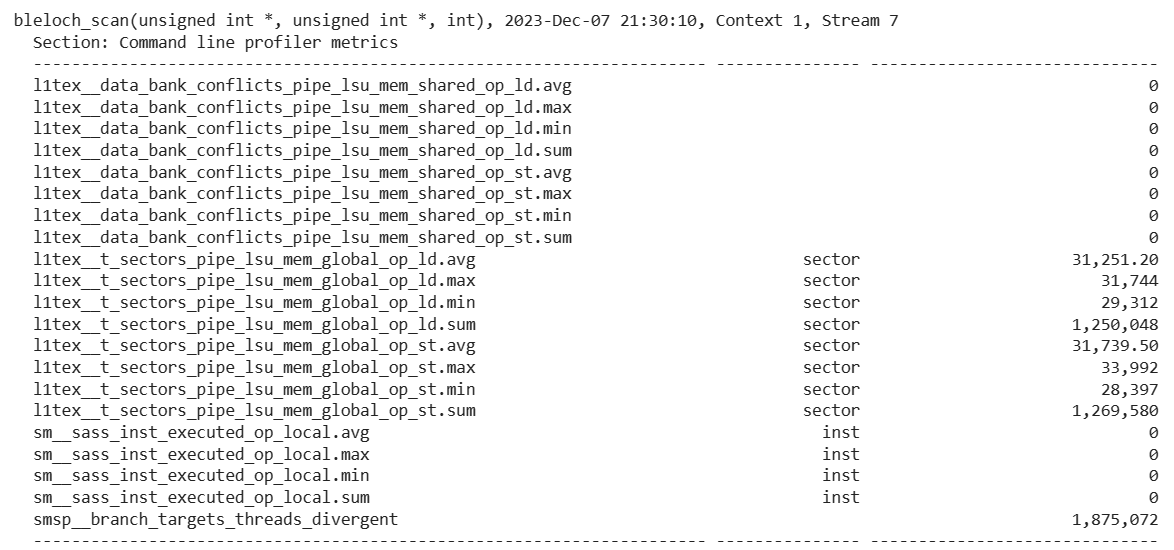
|  |  |  |  |
| --- | --- | --- | --- |
| Размер входных данных | Вид процессора | ГПУ  << < 512, 512 >> > | ЦПУ |
| n = 15 | | 0.3712790 мс | 1.0920131 мс |
| n = 1000 | | 8.19203912 мс | 123.8213828 мс |
| n = 100000 | | 38.13212391 мс | 2912.1293912 мс |
| n = 10000000 | | 1854.138418 мс | 52704.958641 мс |

**Исследование производительности с помощью утилиты nvprof**

Для начала исследуем производительность правильно работающей программы, а затем производительность другой версии: в ней не будет использоваться разделяемая память. Были получены следующие результаты:

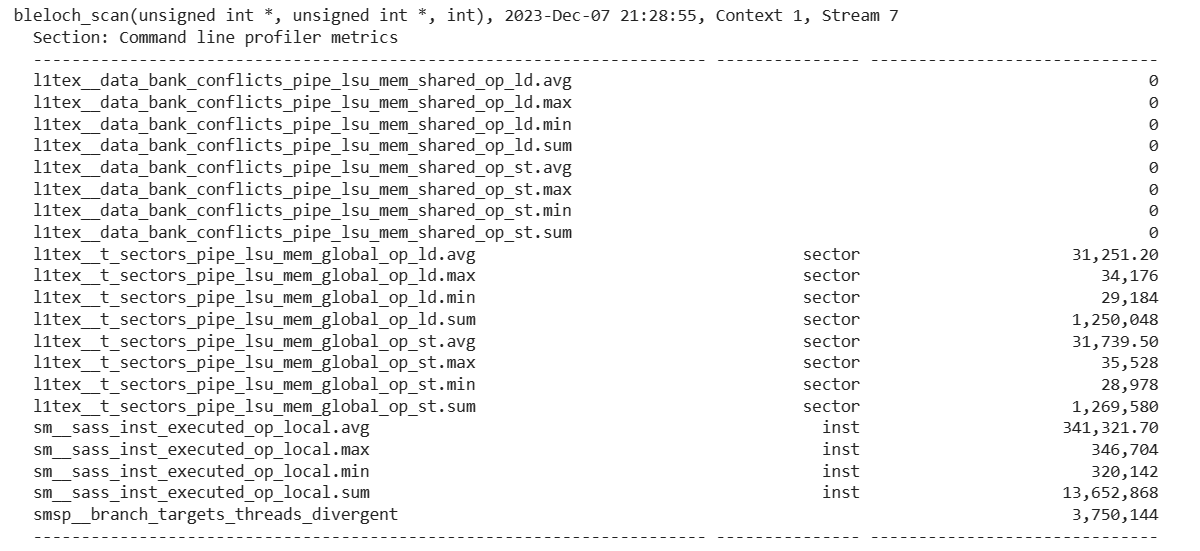
*prog\_true:*

*time =* 1854.138418 мс



*prog\_wrong:*

*time =* 3281.1947198 мс

**

Для анализа были выделены поля, которые сильно отличались в данных версиях. По результатам тестирования можно сделать вывод, что использование разделяемой памяти очень сильно влияет на количество обращений программы к глобальной памяти. Это количество значительно уменьшается, что способствует ускорению работы программы.

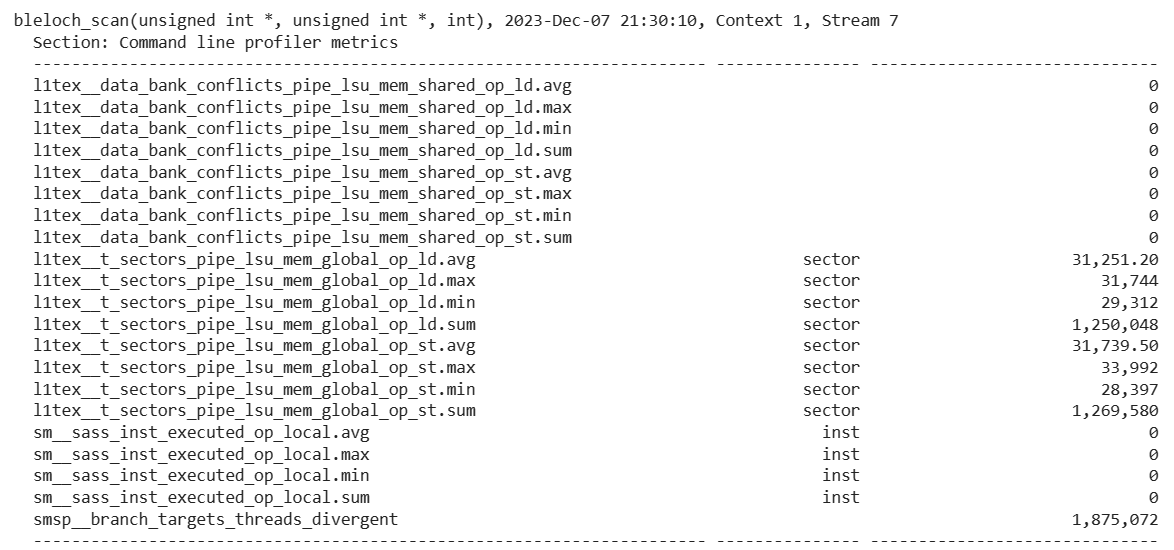
**Исследование производительности на разных данных и на разном количестве потоков:**

Тест 1:

Количество потоков: << < 512, 512 >> >

Входные данные: *n =* 10000000

*time =* 1854.138418 мс

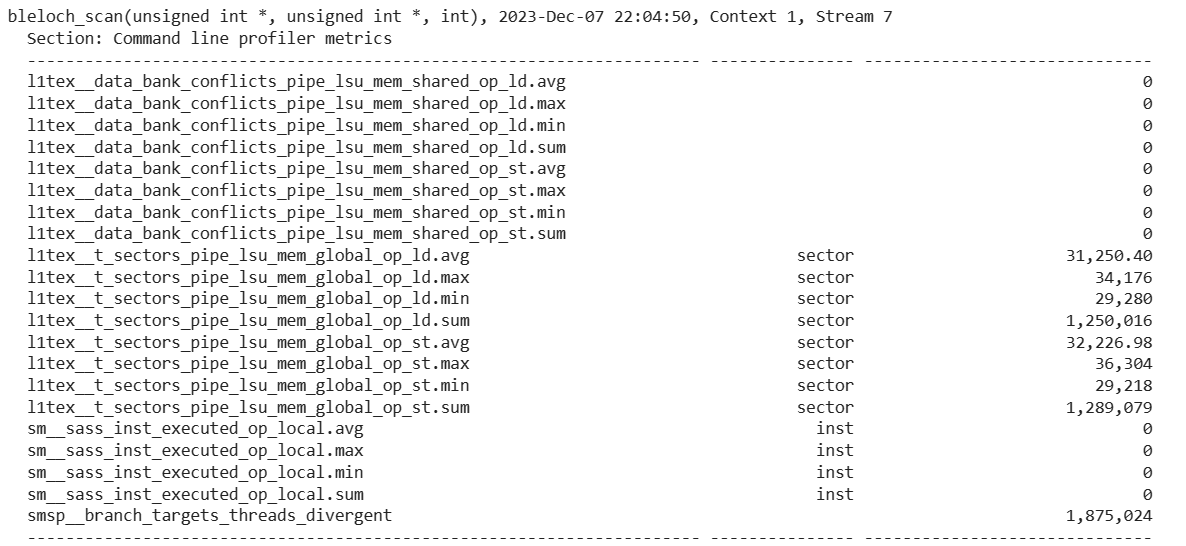


Тест 2:

Количество потоков: << < 256, 256 >> >

Входные данные: *n =* 10000000

*time =* 2569.991725 мс

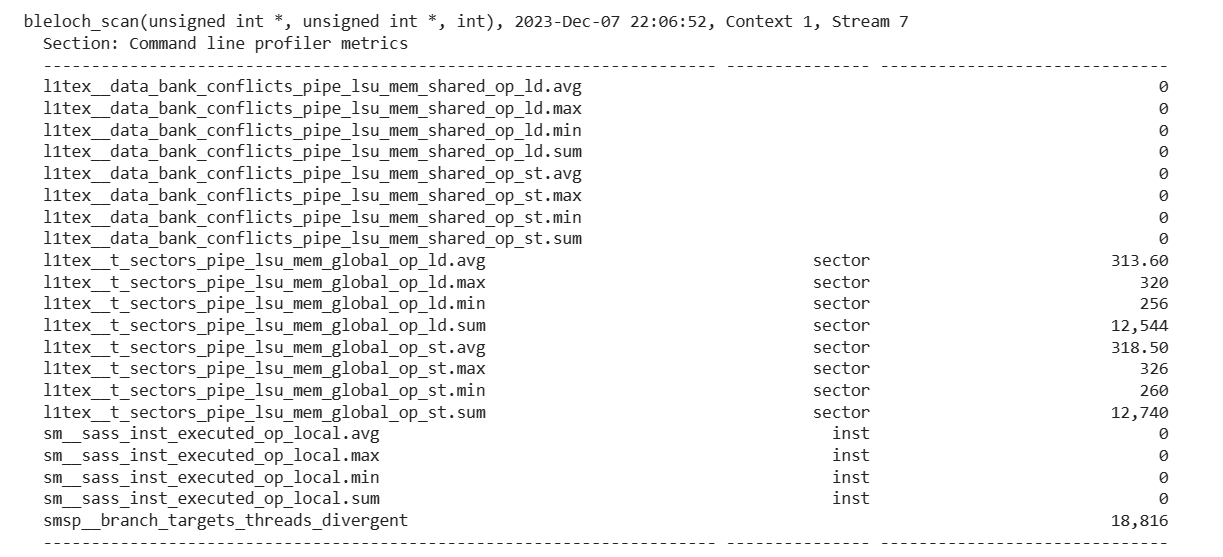


Тест 3:

Количество потоков: << < 512, 512 >> >

Входные данные: *n =* 100000

*time =* 38.13212391 мс

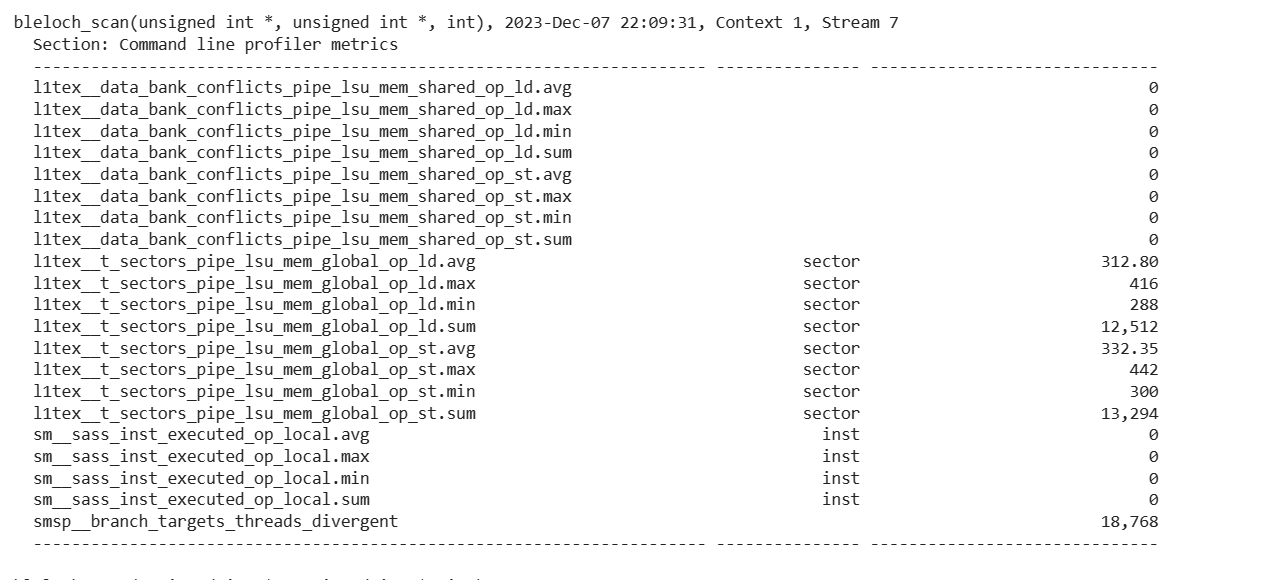


Тест 4:

Количество потоков: << < 128, 128 >> >

Входные данные: *n =* 100000

*time =* 54.12391092 мс



По полученным данным можно сделать вывод, что увеличение числа потоков способствует уменьшению количества обращений к памяти. Уменьшение размеров данных также значительно уменьшает количество обращений.

**Выводы**

В данной лабораторной работе мне удалось написать оптимальных алгоритм поразрядной сортировки массива на ГПУ. Пришлось довольно долго разбираться в алгоритме scan, чтобы сортировка работала так, как нужно. Работа с разделяемой памятью в данной работе позволила мне понять, как можно ускорять некоторые процессы на графическом процессоре. Также в данной работе я вновь потренировался в работе с утилитой *nvprof*, с помощью которой довольно интересно изучать изменение характеристик работы программы в зависимости от изменения алгоритма или входных данных.