МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №4**

**по курсу «Программирование графических процессоров»**

***Работа с матрицами. Метод Гаусса.***

Выполнил: *А.А. Почечура*

Группа: *8О-406Б*

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2023

**Условие**

**Цель работы.** Использование объединения запросов к глобальной памяти. Реализация метода Гаусса с выбором главного элемента по столбцу. Ознакомление с библиотекой алгоритмов для параллельных расчетов Thrust. Использование двухмерной сетки потоков. Исследование производительности программы с помощью утилиты nvprof (обязательно отразить в отчете).

В качестве вещественного типа данных необходимо использовать тип данных double. Библиотеку Thrust использовать только для поиска максимального элемента на каждой итерации алгоритма. В вариантах(1,5,6,7), где необходимо сравнение по модулю с нулем, в качестве нулевого значения использовать . Все результаты выводить с относительной точностью .

**Вариант 3. Решение квадратной СЛАУ.**

Необходимо решить систему уравнений Ax = b, где A – квадратная матрица n x n, b – вектор-столбец свободных коэффициентов длинной n, x – вектор неизвестных.

**Входные данные.** На первой строке задано число n – размер матрицы. В следующих n строках, записано по n вещественных чисел – элементы матрицы. Далее записываются n элементов вектора свободных коэффициентов. n ≤ .

**Выходные данные**. Необходимо вывести n значений, являющиеся элементами вектора неизвестных x.

**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**

Для того, чтобы осуществить решение квадратного СЛАУ, необходимо для начала привести матрицу к верхнему диагональному виду с помощью метода Гаусса с выбором главного элемента по столбцу. Матрицу будем хранить по столбцам, чтобы оптимизировать количество запросов к глобальной памяти. Метод Гаусса будет осуществляться в 4 этапа:

1) Найти индекс максимального по модулю элемента в текущем столбце. Это будет индекс строки, в которой находится максимальный элемент для текущего столбца.

2) Поменять местами строку, номер которой соответствует номеру текущей итерации, со строкой, индекс которой соответствует индексу, найденному в пункте 1).

3) Произвести преобразование всех элементов, находящихся ниже текущей строки, в соответствии с коэффициентами, полученными для каждой строки отдельно (данные операции будут выполняться на ГПУ параллельно, поэтому не нужно занулять элементы, находящиеся в столбце, номер которого соответствует номеру итерации алгоритма, т.к. при вычислении коэффициента для каждой строки требуется незанулённое значение этого элемента).

4) Производить этапы 1) – 3), спускаясь на одну строку вниз до тех пор, пока не будет достигнута последняя строка.

После того, как верхняя треугольная матрица получена, нужно найти значения неизвестных, которые являются решениями СЛАУ. Для этого будем идти с последней строки, получать значения неизвестных, запоминать их и использовать для вычисления значений уже других неизвестных, индекс которых меньше.

**Описание программы**

Для начала считаем матрицу со стандартного ввода, записывая её по столбцам последовательно. Считаем вектор b, чтобы он был последним столбцом нашей матрицы. Затем копируем на ГПУ нашу матрицу и выделяем место под массив *dev\_arr,* с помощью которого мы будем находить максимальный элемент в столбце*.* Далее проделываем *n –* 1итерацию. В каждой из них делаем следующее: с помощью ядра *take\_col* записываем в *dev\_arr* значения элементов в текущем столбце *i* (номер текущей итерации), кроме тех, что лежат выше элемента с координатами *(i, i)* (их значения заполним как нули). После этого, с помощью функции *thrust* найдём индекс строки, в которой находится максимальный по модулю элемент текущего столбца и передадим это значение в ядро *swap* (запускаем это ядро только в том случае, если найденный индекс строки и текущий номер итерации не совпадают). В нём мы меняем местами строку, номер которой равен номеру текущей итерации, и строку, в которой находится максимальный по модулю элемент в текущем столбце. Затем запускаем ядро *change\_rows*, в котором меняем значения элементов, лежащих ниже строки с номером *i* и одновременно не лежащих в столбце с номером *i* (значения, находящиеся в столбце *i* используются для вычисления коэффициентов, с помощью которых происходит преобразования строк. Так как все элементы преобразуются параллельно, важно оставлять значение ненулевыми, иначе при вычислении коэффициента будет происходить деление на 0). После прохождения всех итераций мы получим верхнюю треугольную матрицу (нулевых элементов по факту в ней не будет, но для вычисления значений неизвестных мы всё равно не будем использовать те элементы), поэтому теперь можно начать вычислять значения неизвестных. Завёдем массив *ans*, в котором будут храниться найденные значения. Будем проходится по каждой строке, начиная с последней, считать сумму для уже найденных элементов, перемноженных на соответствующий коэффициент в матрице, вычитать это значение из текущего элемента в столбце b и делить всё это на коэффициент, лежащий в матрице с координатами *(i, i).* Так мы найдём значение неизвестного . Проделаем эту операцию *n* раз, после чего получим все значения вектора *x*. Теперь осталось вывести ответ и освободить используемую память.

**Результаты**

Продемонстрируем скорость работы программы в зависимости от входных данных. n в таблице – размерность матрицы (всего в матрице n\*n элементов).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество блоков и потоков в блоке | Размер входных данных | n = 10 | n = 100 | n = 500 | n = 1000 |
| << < 1, 32 >> >  (<< < dim3(1, 1), dim3(4, 4) >> >) | | 0.671776 мс | 5722.990234 мс | 18474.493145 мс | 56342.325495 мс |
| << < 4, 64 >> >  (<< < dim3(2, 2), dim3(8, 8) >> >) | | 0.639968 мс | 623.539001 мс | 13639.567896 мс | 46951.937913 мс |
| << < 32, 128 >> >  (<< < dim3(4, 4), dim3(16, 16) >> >) | | 0.605184 мс | 154.348511 мс | 11285.239131 мс | 42591.987787 мс |
| << < 128, 128 >> >  (<< < dim3(8, 8), dim3(16, 16) >> >) | | 0.635232 мс | 126.649216 мс | 8993.729993 мс | 37217.745357 мс |
| << < 256, 256 >> >  (<< < dim3(16, 16), dim3(16, 16) >> >) | | 0.786592 мс | 118.091042 мс | 6784,143120 мс | 27652.764669 мс |
| << < 512, 512 >> >  (<< < dim3(16, 16), dim3(32, 32) >> >) | | 0.678400 мс | 152.021347 мс | 4341.360177 мс | 15487.698242 мс |
| << < 1024, 1024 >> >  (<< < dim3(32, 32), dim3(32, 32) >> >) | | 1.178624 мс | 188.135239 мс | 3409.757942 мс | 11031.250851 мс |

Сравним работу алгоритма на ГПУ и ЦПУ:

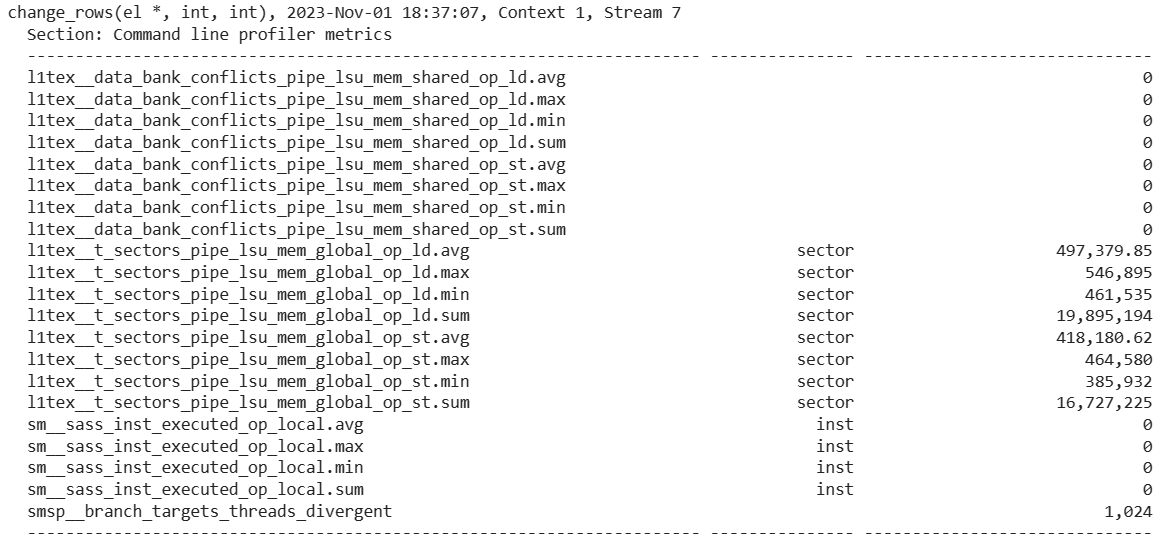
|  |  |  |  |
| --- | --- | --- | --- |
| Размер входных данных | Вид процессора | ГПУ  << < 512, 512 >> >  (<< < dim3(16, 16), dim3(32, 32) >> >) | ЦПУ |
| n = 10 | | 0.678400 мс | 1.402018 мс |
| n = 100 | | 152.021347 мс | 1463.289076 мс |
| n = 500 | | 4341.360177 мс | 535014.838992 мс |
| n = 1000 | | 15487.698242 мс | 5040724.358080 мс |

**Исследование производительности с помощью утилиты nvprof**

Для начала исследуем производительность правильно работающей программы, а затем производительность двух других версий: в одной из них будет использоваться представление матрицы по строкам (как она и задаётся), а в другой будет нарушена работа с двумерной сеткой потоков. Были получены следующие результаты:

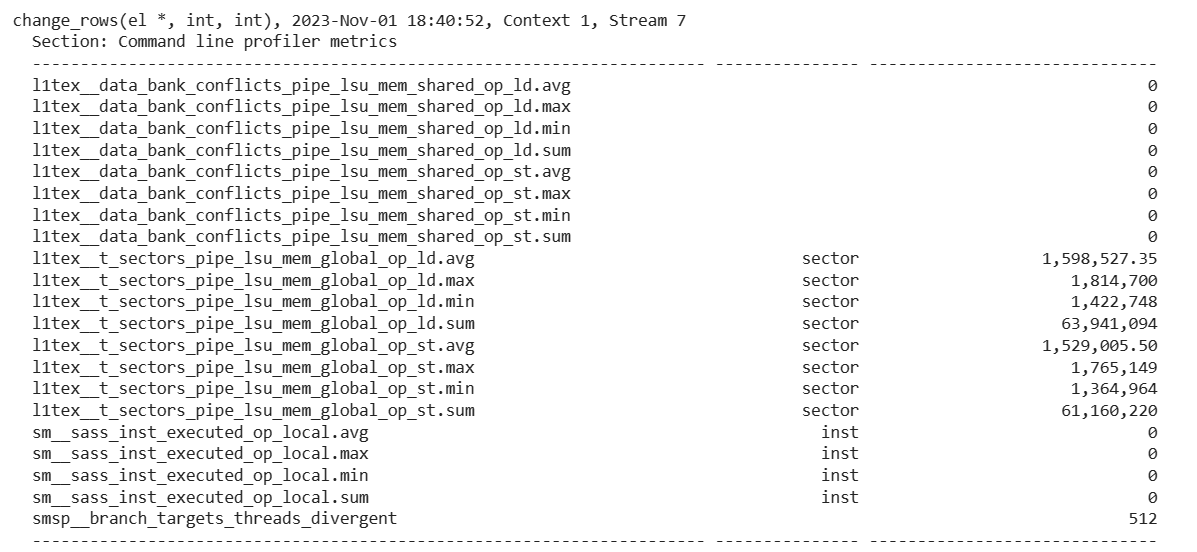
*prog\_true:*

**

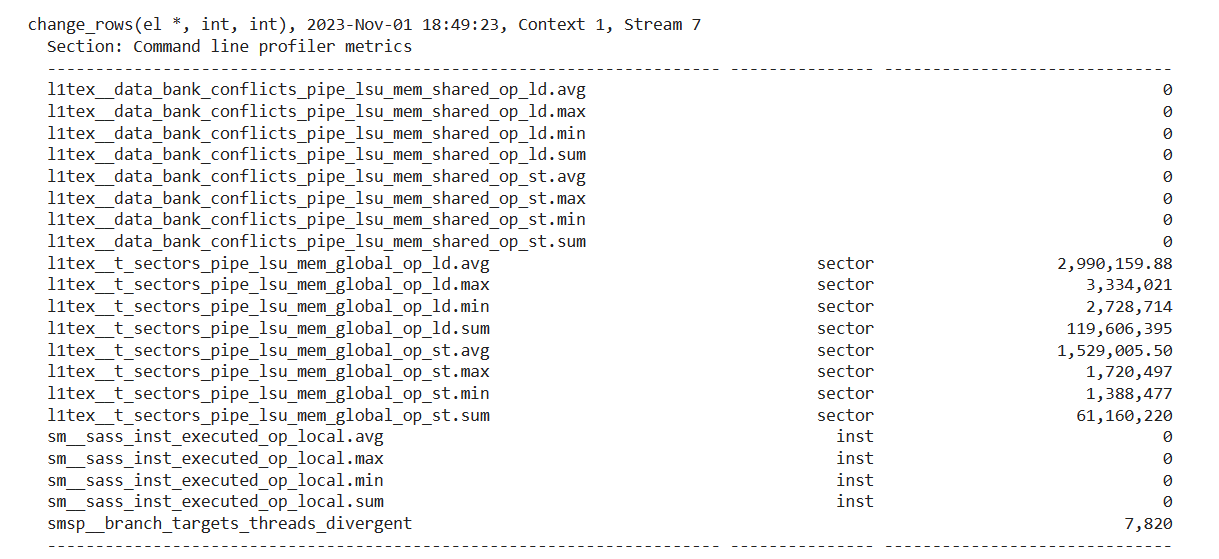


*prog\_wrong\_matr:*

**



*prog\_wrong\_grid:*



Для анализа были выделены поля, которые сильно отличались в данных версиях (даже данные по конфликтам банков памяти были очень схожи). По результатам тестирования можно сделать вывод, что наибольшая нагрузка по производительности приходится на функцию, в которой с помощью двумерной сетки потоков происходит преобразование строк в матрице. Несложно заметить, что, среднее обращение к памяти у трёх версий отличается не менее чем в два раза. Обращение на чтение и запись двух последних версий одинаково, но у правильной версии эти значения меньше в несколько раз. Зато значения дивергенций во всех данных практически не отличаются.

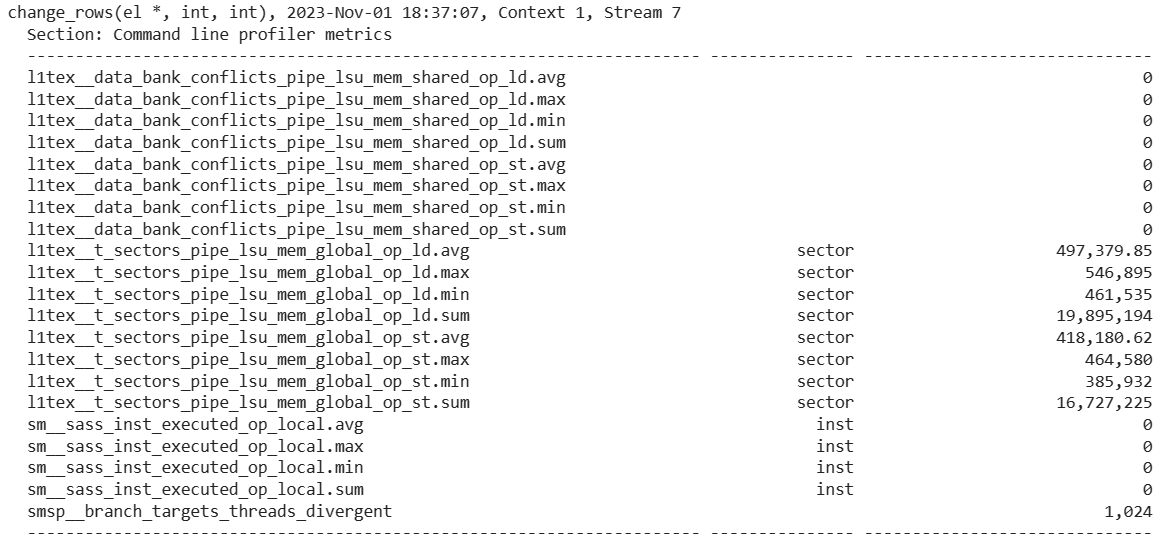
**Исследование производительности на разных данных и на разном количестве потоков:**

Тест 1:

Количество потоков: << < 512, 512 >> > (<< < dim3(16, 16), dim3(32, 32) >> >)

Входные данные: *n =* 1000

**

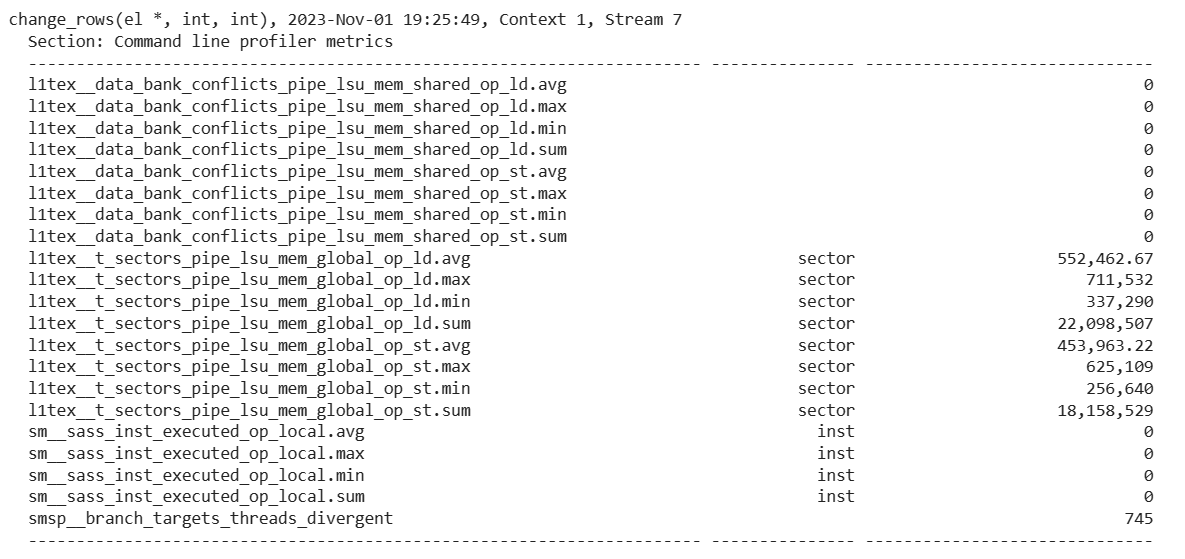


Тест 2:

Количество потоков: << < 256, 256 >> > (<< < dim3(16, 16), dim3(16, 16) >> >)

Входные данные: *n =* 1000

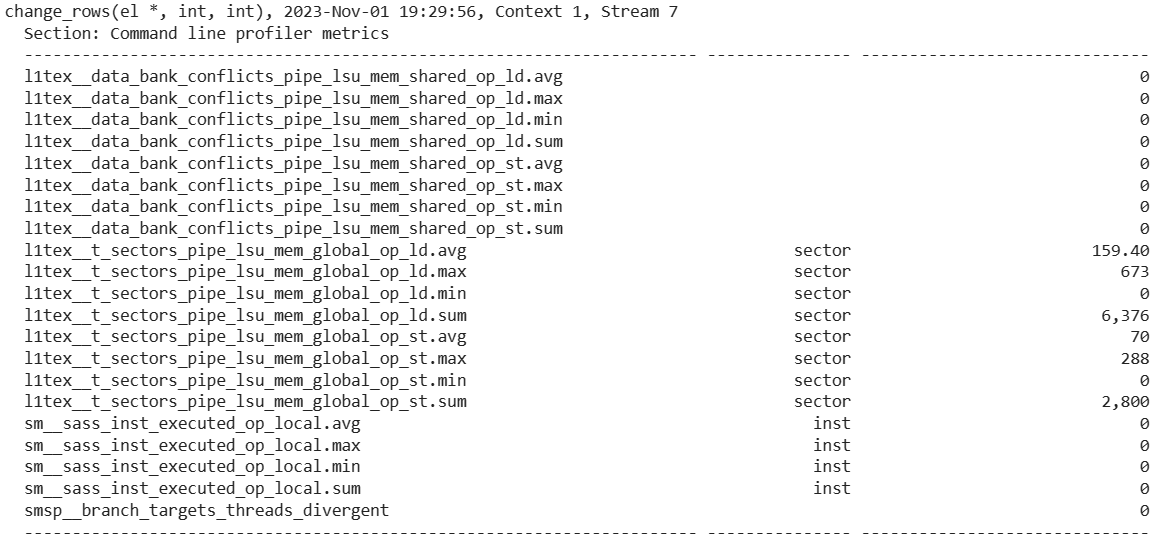




Тест 3:

Количество потоков: << < 512, 512 >> > (<< < dim3(16, 16), dim3(32, 32) >> >)

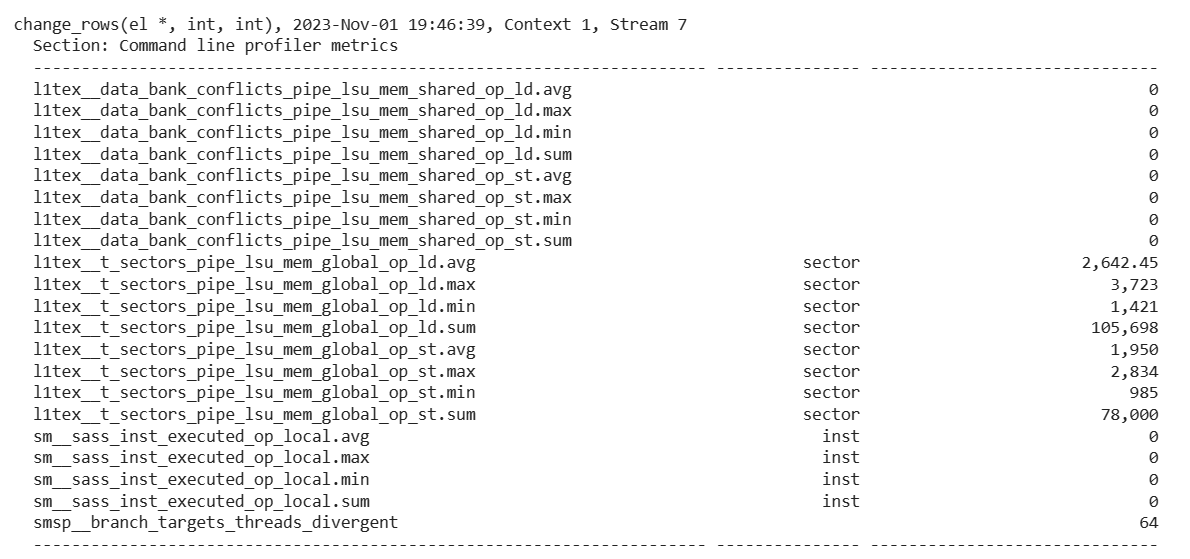
Входные данные: *n =* 100



Тест 4:

Количество потоков: << < 128, 128 >> > (<< < dim3(8, 8), dim3(16, 16) >> >)

Входные данные: *n =* 100



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 решения квадратной СЛАУ, используя ресурсы ГПУ. Впервые мною было написано и использовано в одной программе сразу несколько ядер, имеющих разные размерности сетки потоков. Исследование производительности с помощью *nvprof* в данной работе также производилось мною первый раз. Интересно было изучать изменение различных характеристик производительности в зависимости от изменения алгоритма программы, входных данных и количества использованных потоков.