МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №2**

**по курсу «Параллельная обработка данных»**

**Работа с матрицами. Метод Гаусса.**

Выполнил: Е.С. Пищик

Группа: 8О-406Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2022

**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.

**Вариант 2.** Вычисление обратной матрицы.

**Программное и аппаратное обеспечение**

GPU via SSH:

Compute capability: 2.1

Name: GeForce GT 545

Total Global Memory: 3150381056

Shared memory per block: 49152

Registers per block: 32768

Warp size: 32

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

Max block: (65535, 65535, 65535)

Total constant memory: 65536

Multiprocessors count: 3

CPU via SSH: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz

RAM via SSH: 16 Gb.

HDD via SSH: 471 Gb.

OS: Windows 10

OS via SSH: Ubuntu 16.04.6 LTS

IDE: Visual Studio Code

Compiler via SSH: nvcc

**Метод решения**

Приписываем к исходной матрицы единичную, проходим по столбцам, выбираем ведущий элемент (свапаем строки если ведущий элемент лежит не на первой строке), обнуляем столбец, далее когда обработали все столбцы исходной матрицы и получили верхнетреугольную матрицу начинаем обратный проход зануляем все элементы выше главной диагонали исходной матрицы, аналогичным способом, проходя по столбцам. Итоговая матрица - матрица полученная из приписанной единичной. Зануление элементов на самом деле не происходит, чтобы не тратить на это лишние ресурсы.

**Описание программы**

Программа состоит из одного основного файла gpu.cu. Файл состоит из функций swap\_kernel, nullification\_down\_kernel, nullification\_up\_kernel, divide\_by\_diagonal\_kernel, main. В функции main происходит выделение памяти, копирование, запуск ядер, вывод информации. Ядра swap\_kernel, nullification\_down\_kernel, nullification\_up\_kernel происходят в цикле по столбцам.

В функции ядра swap\_kernel происходит обмен строк матрицы. В ядре проходим по столбцам и меняем соответствующие элементы в строках i и j.

\_\_global\_\_ void swap\_kernel(double \*out, int i, int j, int n)

{

int idx = blockDim.x \* blockIdx.x + threadIdx.x;

int offsetx = blockDim.x \* gridDim.x;

double tmp;

for (int k = idx; k < 2 \* n; k += offsetx)

{

tmp = out[i + k \* n];

out[i + k \* n] = out[j + k \* n];

out[j + k \* n] = tmp;

}

}

В функции ядра nullification\_down\_kernel происходит прямой проход - зануление элементов i-ого столбца под главной диагональю и обновление матрицы.

\_\_global\_\_ void nullification\_down\_kernel(double \*out, int i, int n)

{

int idx = blockDim.x \* blockIdx.x + threadIdx.x;

int idy = blockDim.y \* blockIdx.y + threadIdx.y;

int offsetx = blockDim.x \* gridDim.x;

int offsety = blockDim.y \* gridDim.y;

int startx = idx + i + 1;

int endx = n;

int starty = idy + i + 1;

int endy = 2 \* n;

for (int x = startx; x < endx; x += offsetx)

for (int y = starty; y < endy; y += offsety)

out[x + y \* n] = -out[x + i \* n] / out[i \* (n + 1)] \* out[i + y \* n] + out[x + y \* n];

}

В функции ядра nullification\_down\_kernel происходит обратный проход - зануление элементов i-ого столбца над главной диагональю и обновление матрицы.

\_\_global\_\_ void nullification\_up\_kernel(double \*out, int i, int n)

{

int idx = blockDim.x \* blockIdx.x + threadIdx.x;

int idy = blockDim.y \* blockIdx.y + threadIdx.y;

int offsetx = blockDim.x \* gridDim.x;

int offsety = blockDim.y \* gridDim.y;

int startx = i - idx - 1;

int endx = 0;

int starty = n + idy;

int endy = 2 \* n;

for (int x = startx; x >= endx; x -= offsetx)

for (int y = starty; y < endy; y += offsety)

out[x + y \* n] = -out[x + i \* n] / out[i \* (n + 1)] \* out[i + y \* n] + out[x + y \* n];

}

В функции ядра divide\_by\_diagonal\_kernel происходит деление столбца присоединенной матрицы на соответствующий элемент с главной диагонали основной матрицы, таким образом основная матрица приводится к единичной (в теоретическом подходе, реализация не требует обновления этих элементов, чтобы не тратить лишние ресурсы видеокарты).

\_\_global\_\_ void divide\_by\_diagonal\_kernel(double \*out, int n)

{

int idx = blockDim.x \* blockIdx.x + threadIdx.x;

int idy = blockDim.y \* blockIdx.y + threadIdx.y;

int offsetx = blockDim.x \* gridDim.x;

int offsety = blockDim.y \* gridDim.y;

int startx = idx;

int endx = n;

int starty = n + idy;

int endy = 2 \* n;

for (int x = startx; x < endx; x += offsetx)

for (int y = starty; y < endy; y += offsety)

out[x + y \* n] /= out[x \* (n + 1)];

}

**Результаты**

Замеры времени работы алгоритма.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n \* n, el. | 10^4 | 25 \* 10^4 | 56.25 \* 10^4 | 10^6 | 4 \* 10^6 |
| CPU, ms. | 7.586 | 1201.870 | 5368.802 | 15415.122 | 167462.609 |
| GPU<<<dim3(8, 8), dim3(8,8)>>>, ms. | 34.961 | 332.199 | 863.946 | 1838.414 | 14335.934 |
| GPU<<<dim3(16, 16), dim3(16,16)>>>, ms. | 38.292 | 285.524 | 648.563 | 1320.698 | 9066.626 |
| GPU<<<dim3(32, 16), dim3(32,16)>>>, ms. | 40.009 | 320.356 | 736.289 | 1466.789 | 9138.830 |
| GPU<<<dim3(32, 32), dim3(32,32)>>>, ms. | 80.699 | 623.807 | 1356.669 | 2499.095 | 12405.008 |

Профилировка программы при помощи nvprof с ядрами GPU<<<dim3(16, 16), dim3(16,16)>>> для матрицы 1000 \* 1000 элементов.

==16654== NVPROF is profiling process 16654, command: build/gpu

==16654== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.

==16654== Profiling application: build/gpu

==16654== Profiling result:

==16654== Event result:

Invocations Event Name Min Max Avg

Device "GeForce GT 545 (0)"

Kernel: nullification\_down\_kernel(double\*, int, int)

999 divergent\_branch 0 5191 141

999 global\_store\_transaction 864 251676 101551

Kernel: swap\_kernel(double\*, int, int, int)

989 divergent\_branch 1 1 1

989 global\_store\_transaction 4608 4608 4608

Kernel: divide\_by\_diagonal\_kernel(double\*, int)

1 divergent\_branch 128 128 128

1 global\_store\_transaction 94020 94020 94020

Kernel: nullification\_up\_kernel(double\*, int, int)

999 divergent\_branch 0 628 124

999 global\_store\_transaction 864 126408 60972

==16654== Metric result:

Invocations Metric Name Metric Description Min Max Avg

Device "GeForce GT 545 (0)"

Kernel: nullification\_down\_kernel(double\*, int, int)

999 sm\_efficiency Multiprocessor Activity 78.90% 99.75% 98.21%

999 branch\_efficiency Branch Efficiency 97.09% 100.00% 99.72%

Kernel: swap\_kernel(double\*, int, int, int)

989 sm\_efficiency Multiprocessor Activity 71.00% 78.18% 75.06%

989 branch\_efficiency Branch Efficiency 99.95% 99.95% 99.95%

Kernel: divide\_by\_diagonal\_kernel(double\*, int)

1 sm\_efficiency Multiprocessor Activity 99.02% 99.02% 99.02%

1 branch\_efficiency Branch Efficiency 99.91% 99.91% 99.91%

Kernel: nullification\_up\_kernel(double\*, int, int)

999 sm\_efficiency Multiprocessor Activity 78.40% 99.57% 97.87%

999 branch\_efficiency Branch Efficiency 97.11% 100.00% 99.68%

Количество запросов в глобальную память (global\_store\_transaction) в каждом ииз ядер меньше, чем если бы мы хранили матрицу 1000 \* 1000 по строкам, как минимум если мы для каждого элемента матрицы обращаемся в глобальную память будет не менее 10^6 обращений, nvprof выдал нам что количество обращений в глобальную память не более 0.25 \* 10^6, т.е. как минимум мы уменьшили количество обращений в глобальную память в 4 раза, а в среднем в 10 раз. Также видно что число разделений веток (divergent\_branch) тоже небольшое.

Если посмотреть на метрики то видно, что во всех ядрах эффективность веток почти 100%, а эффективность мультипроцессора в среднем в 3 из 4 ядер почти 100%, а в 1 из 4 ядер 75%. При этом минимальная эффективность мультипроцессора во всех ядрах не менее 71%, что тоже неплохой результат.

**Выводы**

В данной лабораторной работе я изучил параллельную реализацию алгоритма Гаусса для нахождения обратной матрицы, научился ускорять написанный код при помощи уменьшений запросов в глобальную память за счет хранения матрицы по столбцам. Познакомился с утилитой для профилировки nvprof, исследовал различные эвенты и метрики из nvprof на примере программы с ядрами GPU<<<dim3(16, 16), dim3(16,16)>>> для матрицы 1000 \* 1000 элементов.