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 матрицам. Метод Гаусса.

Студент: А. А. Садаков
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 (*обязательно отразить в отчете*).

Вариант 2: Вычисление обратной матрицы.

Программное и аппаратное обеспечение

ГРАФИЧЕСКИЙ ПРОЦЕССОР
Имя устройства: NVIDIA GeForce GTX 1650
Compute capability: 7.5
Размер графической памяти: 4242604032
Размер разделяемой памяти: 49152
Размер константной памяти: 65536
Максимальное количество регистров на блок: 65536
Максимальное количество потоков на блок: 1024
Количество мультипроцессоров: 14

ПРОЦЕССОР
Имя устройства: Intel Core I5-10300H
Архитектура: Comet Lake-H
Количество ядер (потоков): 4(8)
Базовая (максимальная) частота: 2,5(4,5)ГГц
Кеш 1-го уровня: 64Кб (на ядро)
Кеш 2-го уровня: 256Кб (на ядро)
Кеш 3-го уровня: 6Мб (всего)

ОПЕРАТИВНАЯ ПАМЯТЬ
Объём: 8Гб
Тип: DDR4
Частота:2933МГц

ЖЁСТКИЙ ДИСК
Имя устройства: Intel SSDPEKNW512G8L
Тип диска: SSD

Объём памяти: 512Гб
Скорость чтения/записи: 1500/1000 Мб/с

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
OS: Windows 10,Подсистема Ubuntu 20.04
IDE: Visual Studio Code
Компилятор: nvcc

Метод решения

Для нахождения обратной матрицы методом Гаусса создадим единичную матрицу и присоединим её справа к исходной (под «присоединением» имеется ввиду, что все преобразования будут проходить одинаково для обеих матриц, но в памяти они будут храниться отдельно). Далее при помощи преобразований строк приведём исходную матрицу к единичному виду.

Описание программы

Вспомогательные функции:

- $swapRows$,
- $normalisation$,
- $iteration$,
- $backIteration$

Преобразования происходят в цикле. На каждом шаге сначала находится главный элемент и индекс строки, к которой он принадлежит, при помощи библиотеки Thrust. Далее текущая строка и найденная меняются местами функцией $swapRows$, все числа в найденной строке делятся на главный элемент функцией $normalisation$ и, наконец, полученная строка, домноженная на коеффициент, вычитается из нижних так, чтобы все элементы под главным были равны нулю при помощи функции $iteration$.

После оканчания цикла исходная матрица принимает верхний треугольный вид. Для того, чтобы довести матрицу до единичной, используется обратный ход, реализованный в функции $backIteration$.

Вызов функций $iteration$ и $backIteration$ происходит с использованием двухмерной сетки потоков. Все остальные функции вызываются с параметрами ядра $<<< 256, 256 >>>$.

Код функций:

```
__global__ void swapRows (double *data, double *new_data, uint64_t n,
    uint64_t i, uint64_t j, uint64_t border) {
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      int offsetX = gridDim.x * blockDim.x;

      double tmp;
      for (uint64_t k = idx + i; k < n; k += offsetX) {
            tmp = data[i * n + k];
            data[i * n + k] = data[j * n + k];
            data[j * n + k] = tmp;
      }
      for (uint64_t k = idx; k < border + 1; k += offsetX) {
            tmp = new_data[i * n + k];
            new_data[i * n + k] = new_data[j * n + k];
            new_data[j * n + k] = tmp;
      }
}

__global__ void normalisation (double *data, double *new_data, uint64_t n,
    uint64_t id, uint64_t border) {
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      int offsetX = gridDim.x * blockDim.x;

      double coeff;
      coeff = data[id * n + id];
      for (uint64_t k = idx + id + 1; k < n; k += offsetX) {
            data[id * n + k] /= coeff;
      }
      for (uint64_t k = idx; k < border + 1; k += offsetX) {
            new_data[id * n + k] /= coeff;
      }
}

__global__ void iteration (double *data, double *new_data, uint64_t n,
    uint64_t id, uint64_t border) {
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      int idy = blockIdx.y * blockDim.y + threadIdx.y;
```

```
35      int offsetX = gridDim.x * blockDim.x;
36      int offsetY = gridDim.y * blockDim.y;
37
38      double coeff;
39
40      for (uint64_t i = idy + id + 1; i < n; i += offsetY) {
41          coeff = data[i * n + id];
42          for (uint64_t j = idx + id + 1; j < n; j += offsetX) {
43              data[i * n + j] -= coeff * data[id * n + j];
44          }
45          for (uint64_t j = idx; j < border + 1; j += offsetX) {
46              new_data[i * n + j] -= coeff * new_data[id * n + j];
47          }
48      }
49  }
50
51  __global__ void backIteration (double *data, double *new_data, uint64_t n,
        uint64_t id) {
52      int idx = blockIdx.x * blockDim.x + threadIdx.x;
53      int idy = blockIdx.y * blockDim.y + threadIdx.y;
54      int offsetX = gridDim.x * blockDim.x;
55      int offsetY = gridDim.y * blockDim.y;
56
57      double coeff;
58      for (uint64_t i = idy; i <= id - 1; i += offsetY) {
59          coeff = data[i * n + id];
60          for (uint64_t j = idx; j < n; j += offsetX) {
61              new_data[i * n + j] -= coeff * new_data[id * n + j];
62          }
63      }
64  }
```

Результаты

1. Замеры времени работы ядер с различными конфигурациями (время указано в микросекундах).

Конфигурации ядра \ Размер матрицы	10x10	100x100	500x500	1500x1500	3000x3000
1, 32	489	6 256	68 161	691 531	5 124 627
32, 32	492	5 761	53 164	603 206	4 622 832
32, 256	491	5 869	54 139	602 148	4 619 617
256, 256	496	5 876	54 690	599 696	4 619 521
1024, 1024	809	9 554	70 408	624 864	4 646 478

2. Сравнение с CPU

Размер входных данных	10x10	100x100	500x500	1500x1500	3000x3000
GPU(256, 256)	496	5 876	54 690	599 696	4 619 521
CPU	7	5 539	604 241	16 961 677	135 534 879

3. nv-nsight-cu-cli

   Для сравнения скорости обращения к глобальной памяти с объединением запросов и без были реализована версия функции $iteration$ с обращением к элементам по столбцам и по строкам.

   Количество обращений к глобальной памяти при обращении к элементам:

	10x10	100x100	500x500
по столбцам	5 568	2 522 660	634 312 088
по строкам	1 402	687 050	157 250 144

   Как видно по результатам, при обращении к элементам по строкам количество обращений к глобальной памяти меньше в 4 раза.

Выводы

Объединение запросов в глобальную память и использование двумерной сетки потоков может сильно ускорить работу программы. В библиотеке Thrust есть много параллельных алгоритмов обработки данных на GPU.