МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №4
по курсу «Программирование графических процессоров»

Работа с матрицами. Метод Гаусса

Выполнил: Полей-Добронравова Амелия
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников, А.Ю. Морозов

Москва, 2021

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.

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 10^{-7}. Все результаты выводить с относительной точностью 10^{-10}.

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. $n \leq 10^4$.

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

Пример:

Входной файл	Выходной файл
2 1 2 3 4 5 6	-4.0000000000e+00 4.5000000000e+00

Программное и аппаратное обеспечение

Компилятор nvcc версии 7.0(g++ версии 4.8.4) на 64-х битной Ubuntu 14.04 LTS.
Параметры графического процессора:
Compute capability : 6.1
Name : GeForce GTX 1050
Total Global Memory : 2096103424
Shared memory per block : 49152
Registers per block : 65536
Max threads per block : (1024, 1024, 64)
Max block : (2147483647, 65535, 65535)
Total constant memory : 65536
Multiprocessors count : 5

Метод решения

В методе Гаусса с поиском главного элемента по столбцам на каждой итерации ищется максимальный элемент в столбце (начиная с главной диагонали) и строка с ним перемещается наверх. Поиск максимального элемента у меня реализован с помощью библиотеки алгоритмов для параллельных расчетов Thrust.

```
i_ptr = thrust::device_pointer_cast(data + m * n);
i_max_ptr = thrust::max_element(i_ptr + m, i_ptr + n, comparator);
i_max = i_max_ptr - i_ptr;
```

Далее если найден макс элемент на другой строке, не той, с которой начат поиск - запускается функция для gpu - **swap**, меняющая местами текущую строку и строку с максимальным элементом. После этого функция для gpu **change** делит все элементы столбца на максимальный. После приведения матрицы к треугольному виду, на cpu ищу последовательно все неизвестные, начиная с последней, скопировав данные из **data** gpu в **a** cpu.

Описание программы

Макрос **CSC** - макрос для отслеживания ошибок со стороны GPU, вызывается около функций для cuda и выводит текст ошибки при cudaError_t не равным cudaSuccess.
swap - меняет местами две строки матрицы, на GPU.
change - делит все элементы столбца на максимальный, на GPU.
main - ввод данных, поиск неизвестных, вывод результата.
dist - евклидово расстояние, функция только для GPU. Важно, что она работает только с входными переменными с типами данных double, потому что sqrt с входными переменными с типами данных int только для CPU.
struct action - перегрузка оператора () для использования thrust в поиске максимального, используется fabs - модуль вещественного числа.
data - матрица для обработки на GPU.
a - матрица для обработки на CPU.
x - массив неизвестных.

Результаты

Тест:	Результат на GPU:	на CPU:
`A = np.random.random((40,40))` `k= 0` `for i in A:` `  i[k] = k` `  k += 1` `  for j in i:` `  print(j)` `x = np.random.random(40)` `print(x)`	kernel = «<1, 64»>, time = 15.726976 kernel = «<1, 128»>, time = 20.046144 kernel = «<1, 256»>, time = 40.285793 kernel = «<1, 512»>, time = 124.905922 kernel = «<1, 1024»>, time = 531.128418 kernel = «<2, 32»>, time = 9.057440 kernel = «<2, 64»>, time = 10.442848 kernel = «<2, 128»>, time = 16.373184 kernel = «<2, 256»>, time = 40.524097 kernel = «<2, 512»>, time = 162.764633 kernel = «<2, 1024»>, time = 8.193376 kernel = «<4, 32»>, time = 9.047936	230

	kernel = «<4, 64»>, time = 11.009056 kernel = «<4, 128»>, time = 19.546240 kernel = «<4, 256»>, time = 60.251938 kernel = «<4, 512»>, time = 8.200800 kernel = «<4, 1024»>, time = 8.188256 kernel = «<8, 32»>, time = 9.361536 kernel = «<8, 64»>, time = 12.623456 kernel = «<8, 128»>, time = 27.842367 kernel = «<8, 256»>, time = 8.184352 kernel = «<8, 512»>, time = 8.260480 kernel = «<8, 1024»>, time = 8.309632 kernel = «<16, 32»>, time = 10.573504 kernel = «<16, 64»>, time = 18.509344 kernel = «<16, 128»>, time = 8.465856 kernel = «<16, 256»>, time = 8.247680 kernel = «<16, 512»>, time = 8.217952 kernel = «<16, 1024»>, time = 8.387680 kernel = «<32, 32»>, time = 15.832096 kernel = «<32, 64»>, time = 8.154720 kernel = «<32, 128»>, time = 8.264608 kernel = «<32, 256»>, time = 8.175168 kernel = «<32, 512»>, time = 8.257408 kernel = «<32, 1024»>, time = 8.166880 kernel = «<64, 32»>, time = 8.154528 kernel = «<64, 64»>, time = 8.201344 kernel = «<64, 128»>, time = 8.270112 kernel = «<64, 256»>, time = 8.178048 kernel = «<64, 512»>, time = 8.218080 kernel = «<64, 1024»>, time = 8.158528 kernel = «<128, 32»>, time = 8.362976 kernel = «<128, 64»>, time = 8.321408 kernel = «<128, 128»>, time = 8.212864 kernel = «<128, 256»>, time = 8.295616 kernel = «<128, 512»>, time = 8.302432 kernel = «<128, 1024»>, time = 8.262784 kernel = «<256, 32»>, time = 8.231008 kernel = «<256, 64»>, time = 8.280416 kernel = «<256, 128»>, time = 8.295680 kernel = «<256, 256»>, time = 8.176512 kernel = «<256, 512»>, time = 8.175552 kernel = «<256, 1024»>, time = 8.293568 kernel = «<512, 32»>, time = 8.189888 kernel = «<512, 64»>, time = 8.233504 kernel = «<512, 128»>, time = 8.318240 kernel = «<512, 256»>, time = 8.292192 kernel = «<512, 512»>, time = 8.214752 kernel = «<512, 1024»>, time = 8.288960 kernel = «<1024, 32»>, time = 8.314880 kernel = «<1024, 64»>, time = 8.187840 kernel = «<1024, 128»>, time = 8.279168 kernel = «<1024, 256»>, time = 8.223776 kernel = «<1024, 512»>, time = 8.271456 kernel = «<1024, 1024»>, time = 8.170304	

`` A = np.random.random((10,10)) k= 0 for i in A: i[k] = k k += 1 for j in i: print(j) ```	• kernel = «<1, 32»>, time = 1.819424 kernel = «<1, 64»>, time = 1.309632 kernel = «<1, 128»>, time = 1.605856 kernel = «<1, 256»>, time = 2.988928 kernel = «<1, 512»>, time = 8.466496 kernel = «<1, 1024»>, time = 36.320095 kernel = «<2, 32»>, time = 1.028480 kernel = «<2, 64»>, time = 1.133376 kernel = «<2, 128»>, time = 1.634016 kernel = «<2, 256»>, time = 3.706464 kernel = «<2, 512»>, time = 13.568960 kernel = «<2, 1024»>, time = 0.928832 kernel = «<4, 32»>, time = 1.015968 kernel = «<4, 64»>, time = 1.171040 kernel = «<4, 128»>, time = 2.018688 kernel = «<4, 256»>, time = 5.853472 kernel = «<4, 512»>, time = 0.895008 kernel = «<4, 1024»>, time = 0.890816 kernel = «<8, 32»>, time = 1.038560 kernel = «<8, 64»>, time = 1.436160 kernel = «<8, 128»>, time = 3.463392 kernel = «<8, 256»>, time = 0.867872 kernel = «<8, 512»>, time = 0.864576 kernel = «<8, 1024»>, time = 0.870944 kernel = «<16, 32»>, time = 1.275744 kernel = «<16, 64»>, time = 2.835168 kernel = «<16, 128»>, time = 0.849824 kernel = «<16, 256»>, time = 0.843584	10

	kernel = «<16, 512»>, time = 0.848448 kernel = «<16, 1024»>, time = 0.844512 kernel = «<32, 32»>, time = 2.564832 kernel = «<32, 64»>, time = 0.849568 kernel = «<32, 128»>, time = 0.852352 kernel = «<32, 256»>, time = 0.843680 kernel = «<32, 512»>, time = 0.842528 kernel = «<32, 1024»>, time = 0.843872 kernel = «<64, 32»>, time = 0.843648 kernel = «<64, 64»>, time = 0.849472 kernel = «<64, 128»>, time = 0.826944 kernel = «<64, 256»>, time = 0.826624 kernel = «<64, 512»>, time = 0.828864 kernel = «<64, 1024»>, time = 0.825376 kernel = «<128, 32»>, time = 0.827584 kernel = «<128, 64»>, time = 0.826432 kernel = «<128, 128»>, time = 0.826592 kernel = «<128, 256»>, time = 0.825568 kernel = «<128, 512»>, time = 0.833120 kernel = «<128, 1024»>, time = 0.831616 kernel = «<256, 32»>, time = 0.828384 kernel = «<256, 64»>, time = 0.826976 kernel = «<256, 128»>, time = 0.829792 kernel = «<256, 256»>, time = 0.838880 kernel = «<256, 512»>, time = 0.803136 kernel = «<256, 1024»>, time = 0.804544 kernel = «<512, 32»>, time = 0.802240 kernel = «<512, 64»>, time = 0.801152 kernel = «<512, 128»>, time =	

	0.811968 kernel = «<512, 256»>, time = 0.806368 kernel = «<512, 512»>, time = 0.803360 kernel = «<512, 1024»>, time = 0.806592 kernel = «<1024, 32»>, time = 0.806048 kernel = «<1024, 64»>, time = 0.812064 kernel = «<1024, 128»>, time = 0.805536 kernel = «<1024, 256»>, time = 0.804512 kernel = «<1024, 512»>, time = 0.807520 kernel = «<1024, 1024»>, time = 0.807520	

Код программы для CPU:

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <vector>
#include <string>
#include <time.h>

void swap(double* a, int k, int y, int n) {
    double w;
    for(int i = 0; i < n+1; i++) {
        w = a[i * n + k];
        a[i * n + k] = a[i * n + y];
        a[i * n + y] = w;
    }
}

int main() {
    int n;
    std::cin >> n;
    double* a = (double*)malloc(sizeof(double) * n * (n+1));
    double* x = (double*)malloc(sizeof(double) * n);
    double p;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            std::cin >> p;
            a[j*n + i] = p;
        }
    }

    for (int i = 0; i < n; i++) {
            std::cin >> p;
            int s = n*n + i;
            a[s] = p;
    }
```

```
    double max = 0;
    int max_i = -1;
    clock_t begin = clock();
    for(int m = 0; m < n-1; m++) {
        for(int s = 0; s < n; s++) {
            if (abs(a[m*n + s]) > max) {
                max = abs(a[m*n + s]);
                max_i = s;
            }
        }
        if(m != max_i) {
            swap(a, m, max_i, n);
        }
        for (int j = m+1; j < n; j++) {
            double d = a[m*n+j] / a[m*n+m];
            for (int i = m; i < n + 1; i++) { //для приписанной матрицы
                a[i*n+j] = a[i*n+j] - a[i*n+m] * d;
            }
        }
    }
    clock_t end = clock();
    double time_spent = (double)(end - begin) * 1000 / CLOCKS_PER_SEC;
    printf("%lf\n", time_spent);
    printf("\n");

    x[n - 1] = a[(n+1)*n - 1] / a[(n+1)*n - n - 1];
    for(int k = n-1; k >= 0; k--) {
        double d = 0;
        for (int j = k + 1; j < n; j++) {
            d = d + a[j*n + k] * x[j];
        }
        x[k] = (a[n*n + k] - d) / a[k*n + k];
    }

    for (int i = 0; i < n; i++) {
         std::cout << x[i] << " ";

    }
    return 0;
}
```

Выводы

1. Thrust - удобная библиотека алгоритмов для GPU.
2. Можно выделить два способа оптимизации в работе с глобальной памятью: выравнивание размеров используемых типов и использование объединенных запросов. В данной лабораторной было использовано объединение запросов к глобальной памяти. Обращения идут через 32/64/128 - битовые слова.
3. Условия необходимые для объединения при обращении в память: Нити должны обращаться либо к 32-битовым словам, давая при этом в результате один

64-байтовый блок (транзакцию), либо к 64-битовым словам, давая при этом один 128-байтовый блок (транзакцию).

Если используется обращение к 128-битовым словам, то в результате будет выполнено две транзакции, каждая из которых вернет по 128 байт информации.

Нити должны обращаться к элементам памяти последовательно, каждой следующей нити должно соответствовать следующее слово в памяти (некоторые нити могут вообще не обращаться к соответствующим словам).

Все 16 слов должны быть в пределах блока памяти, к которому выполняется доступ.

4. При объединение запросов происходит увеличение скорости работы с памятью на порядок; лучше использовать не массив структур, а набор массивов отдельных компонент.
5. Анализируя время работы на представленных мной тестах, можно сказать, что время не сильно меняется при большом количестве блоков и потоков после определенного нижнего значения. В первом тесте это время 8, во втором - 0.85. Худшее время на GPU в два раза лучше, чем время на CPU.