МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет «Информационные технологии и прикладная математика»
Кафедра «Вычислительная математика и программирование»

Лабораторная работа №4
по курсу «Параллельная обработка данных»

Работа с матрицами. Метод Гаусса.

Выполнил: В.И. Лобов
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2019

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.

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.

Программное и аппаратное обеспечение

GPU:

Название: GeForce GTX 1060
Compute apability: 6.1
Графическая память: 2075328512
Разделяемая память: 49152
Константная память: 65536
Количество регистров на блок: 65536
Максимальное количество блоков: (2147483647, 65535, 65535)
Максимальное количество нитей: (1024, 1024, 64)
Количество мультипроцессоров: 10

Сведения о системе:

Процессор: Intel Core i7-8700k 4.5GHz
Оперативная память: 16Gb
HDD: 1Tb
Операционная система: Ubuntu 18.04
IDE: Nsight
Компилятор: nvcc

Метод решения

Метод Гаусса решения квадратной СЛАУ.

Пусть задана исходная система линейных алгебраических уравнений, которую можно записать в виде Ax = b, где A — матрица системы, x — вектор неизвестных, а b — вектор свободных значений. Пусть A|b — расширенная матрица системы.

Тогда расширенную матрицу A|b можно привести к ступенчатому виду с помощью элементарных преобразований над строками.

Будем менять строки таким образом, чтобы текущий диагональный элемент имел наибольшее значение среди элементов, расположенных в том же столбце под ним. Таким образом, если ранг матрицы A равен рангу расширенной матрицы A|b, то последняя ненулевая строка будет содержать единственную неизвестную.

Проходя строки в обратном порядке снизу вверх и вычитая нижнюю строку из соседней верхней, получим матрицу, где все элементы кроме диагональных равны 0. Поделим свободные значения на соответствующие им коэффициенты диагональных элементов, отличных от 0 и получим $\hat{b}$ - решение системы ($x=\hat{b}$).

Описание программы

С целью использования поиска максимального элемента столбца матрицы с помощью встроенной функции max_element из библиотеки thrust будем хранить матрицу по столбцам, а не по строкам.
Запустив цикл по столбцам матрицы, будем находить соответствующую строку, содержащую максимальный ведущий элемент, менять её местами с текущей строкой и обнулять значения под текущим ведущим элементом.
Дойдя до последней строки, пройдём матрицу «снизу-наверх», обнулив все недиагональные элементы матрицы, и преобразуем матрицу в единичную с помощью элементарных преобразований.

```
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <thrust/extrema.h>
#include <thrust/device_ptr.h>

#define CSC(call)                                   \
do {                                                \
        cudaError_t res = call;                     \
        if (res != cudaSuccess) {           \
                fprintf(stderr, "ERROR in %s:%d. Message: %s\n",                  \
                                __FILE__, __LINE__, cudaGetErrorString(res));          \
                exit(0);                            \
        }                                                   \
} while(0)

__global__ void subtract_row(double *matrix, int n, int column) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int idy = threadIdx.y + blockDim.y * blockIdx.y;
    int offsetx = blockDim.x * gridDim.x;
```

```
    int offsety = blockDim.y * gridDim.y;
    int i, j;

    double coeff;
    double divisor = matrix[column * n + column];
    for (i = 1 + column + idx; i < n; i += offsetx) {
        coeff = matrix[column * n + i] / divisor;
        for (j = 1 + column + idy; j < n + 1; j += offsety) {
            matrix[j * n + i] -= coeff * matrix[j * n + column];
        }
    }
}

__global__ void reverse_subtract_row(double *matrix, int n, int column) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int offsetx = blockDim.x * gridDim.x;
    int i;

    double coeff;
    double divisor = matrix[column * n + column];
    for (i = idx; i < column; i += offsetx) {
        coeff = matrix[column * n + i] / divisor;
        matrix[n * n + i] -= coeff * matrix[n * n + column];
    }
}

__global__ void normalize(double *matrix, int n) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int offsetx = blockDim.x * gridDim.x;
    int i;

    for (i = idx; i < n; i += offsetx) {
        matrix[n * n + i] /= matrix[i * n + i];
    }
}

__global__ void swap_rows(double *matrix, int n, int column, int max_row) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int offsetx = blockDim.x * gridDim.x;
    int i;
    double tmp;

    for (i = idx + column; i < n + 1; i+= offsetx) {
        tmp = matrix[i * n + column];
        matrix[i * n + column] = matrix[i * n + max_row];
        matrix[i * n + max_row] = tmp;
    }
}

struct compare {
        __host__ __device__ bool operator ()(double lhs, double rhs) {
        return fabs(lhs) < fabs(rhs);
        }
};
```

```
void solve(double *matrix, int n) {
    for (int column = 0; column < n; ++column) {
        thrust::device_ptr<double> thrust_matrix =
            thrust::device_pointer_cast(matrix) + n * column;
        int max_row = thrust::max_element(thrust_matrix + column,
                                          thrust_matrix + n, compare()) - thrust_matrix;
        if (max_row >= n) {
            continue;
        }

        swap_rows<<<32, 32>>>(matrix, n, column, max_row);
        subtract_row<<<dim3(32, 32), dim3(32, 32)>>>(matrix, n, column);
    }

    for (int column = n - 1; column >= 0; --column) {
        reverse_subtract_row<<<32, 32>>>(matrix, n, column);
    }

    normalize<<<32, 32>>>(matrix, n);
}

int main() {
    int n;
    scanf("%d", &n);
    double *matrix = (double *) malloc(sizeof(double *) * (n + 1) * n);

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            scanf("%lf", matrix + j * n + i);
        }
    }

    for (int i = 0; i < n; ++i) {
        scanf("%lf", matrix + n * n + i);
    }

        double *device_matrix;

        CSC(cudaMalloc(&device_matrix, sizeof(double) * (n + 1) * n));
            CSC(cudaMemcpy(device_matrix, matrix, sizeof(double) * (n + 1) * n,
cudaMemcpyHostToDevice));

        solve(device_matrix, n);

        CSC(cudaMemcpy(matrix + n * n, device_matrix + n * n, sizeof(double) * n,
cudaMemcpyDeviceToHost));

    for (int i = 0; i < n; ++i) {
        printf("%.10e ", matrix[n * n + i]);
    }
    printf("\n");
        CSC(cudaFree(device_matrix));
        free(matrix);
        return 0;
}
```

Результаты:

Количество строк/столбцов матрицы n = 500.

Конфигурация(вычитание строк)	Время(мс)
CPU	145
(32, 32), (32, 32)	39
(64, 64), (32, 32)	74
(128, 128), (32, 32)	174
(256, 256), (32, 32)	476

n = 1000.

Конфигурация(вычитание строк)	Время(мс)
CPU	1456
(32, 32), (32, 32)	171
(64, 64), (32, 32)	243
(128, 128), (32, 32)	517
(256, 256), (32, 32)	1233

n = 2000.

Конфигурация(вычитание строк)	Время(мс)
CPU	19620
(32, 32), (32, 32)	588
(64, 64), (32, 32)	829
(128, 128), (32, 32)	1443
(256, 256), (32, 32)	1344

Выводы

Как видно из результатов тестирования, вычисление решений СЛАУ большой размерности на видеокарте более предпочтительно, чем на CPU. Это связано с тем, что сложность метода Гаусса составляет $O(n^3)$, что не позволяет эффективно применять его при размерностях матриц больше, чем 1000.

Основным затратным действием на каждой стадии алгоритма является вычитание ведущей строки из всех остальных строк матрицы. Это действие хорошо распараллеливается, благодаря чему и достигается основной прирост в скорости при вычислении на видеокарте.

При увеличении количества потоков в блоке(больше, чем 32) происходит значительное ухудшение производительности. Это связано с тем, что количество запусков ядра пропорционально количеству строк — каждый раз инициализировать потоки слишком затратно.