3МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4**

**по курсу «Программирование графических процессоров»**

**Работа с матрицами. Метод Гаусса.**

Выполнил: Иларионов Д.А.

Группа: М8О-408Б-17

Преподаватели: Крашенинников К.Г.,

Морозов А.Ю.

Москва, 2020

**Условие**

Использование объединения запросов к глобальной памяти.

Реализация метода Гаусса с выбором главного элемента по столбцу. Ознакомление с

библиотекой алгоритмов для параллельных расчетов Thrust.

**Вариант 7. Решение матричного уравнения.**

Необходимо найти любое решение матричного уравнения AX = B, где A --

матрица n x m, X -- неизвестная матрица m x k, B -- матрица n x k.

**Программное и аппаратное обеспечение**

**GPU:**

* Name: GeForce GTX 1060
* Compute capability: 6.1
* Частота видеопроцессора: 1404 – 1670 (Boost) МГц
* Частота памяти: 8000 МГц
* Графическая память: 6144 МБ
* Разделяемая память: отсутствует
* 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 i7-8750H 2.20GHz x 6
* Оперативная память: 16 ГБ
* SSD: 128 ГБ
* HDD: 1000 ГБ

**Программное обеспечение:**

* OS: Windows 10
* IDE: Visual Studio 2019
* Компилятор: nvcc

**Метод решения**

Это такая лабораторная, которая, вроде бы на первый взгляд не сложная, однако в итоге требует быть очень внимательным, стараться оптимизировать все по максимуму, а также стоила мне многих нервов и было около 50 попыток сдать. С данной ЛР вышло проблем даже больше, чем с 7 или 8. Хотя, по сути, она и не такая трудная, но есть некоторые тонкости. В ядрах нужно проходить только по самым необходимым элементам, чтобы не было гонки потоков, то есть потоки не использовали один и тот же элемент. Для этого я разбивал функции. Сначала мы приводим к треугольному виду. Задача не слишком сложная, но и не особо приятная. Для того, чтобы не было гонки потоков, мы сначала вычитаем элементы справа от основного, а потом уже отдельным ядром основной. Также лучше сделать двойной цикл, во первых – так выглядит все более понятно и удобно. Во вторых, это решило мою проблему с непрохождением на 13 тесте. Я забывал некоторые элементы проверить. То есть мы двигаться можем вправо и вниз. Вместо сложного цикла с вычислением индексов строк, которые нам нужны, можно просто добавлять их, когда мы спускаемся вниз в самом цикле, а не еще одним ядром. Когда матрица у нас треугольная, мы отбираем лишь ступеньки, и потом по ним приводим матрицу к единичному виду. Матрица Х заполняется нулями, а строки, соответствующие конкретным столбцам заменяются на строки с матрицы B (видоизмененной после приведения матрицы А к треугольному, а потом и единичному виду). Мы просто сопрягаем ее с матрицей А и делаем те же самые вычитания в матрице B. Удобно даже хранить обе матрицы в одном массиве. А для самого быстрого поиска макс. элемента и лучшей оптимизации – мы используем хранение по столбцам. То есть следующий элемент – мы спускаемся вниз на строку, а не идем вправо. В конце концов, мы выводим получившуюся матрицу.

**Описание программы**

**Файл kernel.cu**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <cuda.h>

#include <thrust/functional.h>

#include <thrust/swap.h>

#include <thrust/extrema.h>

#include <thrust/execution\_policy.h>

#include <thrust/iterator/transform\_iterator.h>

#include <thrust/host\_vector.h>

#include <thrust/device\_vector.h>

#include <thrust/reduce.h>

#include <iostream>

#include <fstream>

#include <cstdio>

#include <sstream>

#include <iomanip>

#include <math.h>

#include <algorithm>

#include <string>

using namespace std;

using namespace thrust;

#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)

const int BLOCK = 1024;

const int THREAD = 1024;

\_\_constant\_\_ int N[1];

\_\_constant\_\_ int M[1];

\_\_constant\_\_ int K[1];

struct abs\_fun : public thrust::unary\_function<double, double> {

\_\_host\_\_ \_\_device\_\_

double operator()(double elem) const {

return elem < 0 ? -elem : elem;

}

};

struct abs\_comp {

abs\_fun fabs;

\_\_host\_\_ \_\_device\_\_ double operator()(double a, double b) {

return fabs(a) < fabs(b);

}

};

\_\_global\_\_ void fastSwap(int row, int col, int max\_id, double\* AB) {

unsigned tx = blockIdx.x \* blockDim.x + threadIdx.x;

unsigned ts = blockDim.x \* gridDim.x;

for (int el = tx; el < M[0] + K[0]; el += ts) {

double temp = AB[el \* N[0] + row];

AB[el \* N[0] + row] = AB[el \* N[0] + max\_id];

AB[el \* N[0] + max\_id] = temp;

}

}

\_\_global\_\_ void triagStep(double\* AB, int r, int c) {

unsigned tx = blockIdx.x \* blockDim.x + threadIdx.x;

unsigned txs = blockDim.x \* gridDim.x;

unsigned ty = blockIdx.y \* blockDim.y + threadIdx.y;

unsigned tys = blockDim.y \* gridDim.y;

for (int col\_ = ty + c + 1; col\_ < M[0] + K[0]; col\_ += tys) {

for (int row\_ = tx + r + 1; row\_ < N[0]; row\_ += txs) {

double rowDivisor = AB[row\_ + N[0] \* c] / AB[r + N[0] \* c];

AB[row\_ + N[0] \* col\_] -= AB[r + N[0] \* col\_] \* rowDivisor;

}

}

}

\_\_global\_\_ void backStep(double\* AB, int row, int col) {

unsigned tx = blockIdx.x \* blockDim.x + threadIdx.x;

unsigned txs = blockDim.x \* gridDim.x;

unsigned ty = blockIdx.y \* blockDim.y + threadIdx.y;

unsigned tys = blockDim.y \* gridDim.y;

for (int col\_ = ty + col + 1; col\_ < M[0] + K[0]; col\_ += tys) {

for (int row\_ = tx; row\_ < row; row\_ += txs) {

double rowDivisor = AB[row\_ + col \* N[0]] / AB[row + col \* N[0]];

AB[row\_ + col\_ \* N[0]] -= AB[row + col\_ \* N[0]] \* rowDivisor;

}

}

}

\_\_global\_\_ void backKill(double\* AB, int row, int col) {

unsigned tx = blockIdx.x \* blockDim.x + threadIdx.x;

unsigned txs = blockDim.x \* gridDim.x;

for (int row\_ = tx; row\_ < row; row\_ += txs) {

double rowDivisor = AB[row\_ + col \* N[0]] / AB[row + col \* N[0]];

AB[row\_ + col \* N[0]] -= AB[row + col \* N[0]] \* rowDivisor;

}

}

\_\_global\_\_ void division(double\* AB, double\* diag, int\* indices, int\* indices2, int freeX) {

unsigned tx = blockIdx.x \* blockDim.x + threadIdx.x;

unsigned txs = blockDim.x \* gridDim.x;

unsigned ty = blockIdx.y \* blockDim.y + threadIdx.y;

unsigned tys = blockDim.y \* gridDim.y;

for (int col\_ = ty; col\_ < K[0]; col\_ += tys) {

for (int i = tx; i < freeX; i += txs) {

int curRow = indices2[i];

double rowDivisor = diag[i];

AB[curRow + col\_ \* N[0] + N[0] \* M[0]] = AB[curRow + col\_ \* N[0] + N[0] \* M[0]] / rowDivisor;

}

}

}

\_\_global\_\_ void triagKill(double\* AB, int r, int c) {

unsigned tx = blockIdx.x \* blockDim.x + threadIdx.x;

unsigned txs = blockDim.x \* gridDim.x;

for (int row\_ = tx + r + 1; row\_ < N[0]; row\_ += txs) {

double rowDivisor = AB[row\_ + N[0] \* c] / AB[r + N[0] \* c];

AB[row\_ + N[0] \* c] -= AB[r + N[0] \* c] \* rowDivisor;

}

}

int main() {

int n, m, k;

scanf("%d%d%d", &n, &m, &k);

cudaMemcpyToSymbol(N, &n, sizeof(int));

cudaMemcpyToSymbol(M, &m, sizeof(int));

cudaMemcpyToSymbol(K, &k, sizeof(int));

int nm, nk;

nm = n \* m;

nk = n \* k;

host\_vector<double> matrAB\_host(nm + nk);

device\_vector<double> matrAB\_dev;

for (int i = 0; i < n; ++i) {

for (int j = 0; j < m; ++j) {

//cin >> matrAB\_host[i + j \* n];

scanf("%lf", &matrAB\_host[i + j \* n]);

}

}

for (int i = 0; i < n; ++i) {

for (int j = 0; j < k; ++j) {

//cin >> matrAB\_host[i + j \* n + nm];

scanf("%lf", &matrAB\_host[i + j \* n + nm]);

}

}

//cerr << "N: " << n << " M: " << m << " K: " << k << "\n";

//to GPU memory

matrAB\_dev = matrAB\_host;

// pointers to mem:

double\* AB\_ptr = thrust::raw\_pointer\_cast(matrAB\_dev.data());

dim3 gridSz(32, 32);

dim3 blockSz(32, 32);

int row = 0; //i - col

host\_vector<int> indices(0);

host\_vector<int> indices2(0);

for (int i = 0; i < m && row < n; ++i) {

auto iter = matrAB\_dev.begin() + i \* n;

auto max\_el = thrust::max\_element(

iter + row, iter + n, abs\_comp()

);

int max\_id = max\_el - iter;

if (fabs(\*max\_el) <= 1e-7) {

continue;

}

else {

if (max\_id != row) {

fastSwap << <BLOCK, THREAD >> > (row, i, max\_id, AB\_ptr);

CSC(cudaThreadSynchronize());

}

}

triagStep << <gridSz, blockSz >> > (AB\_ptr, row, i);

CSC(cudaThreadSynchronize());

triagKill << <BLOCK, THREAD >> > (AB\_ptr, row, i);

CSC(cudaThreadSynchronize());

indices.push\_back(i);

indices2.push\_back(row);

++row;

}

matrAB\_host = matrAB\_dev;

host\_vector<bool> i\_bool(m, false);

device\_vector<bool> i\_bool\_dev(m);

int freeX = indices.size();

for (int i = 0; i < freeX; ++i) {

i\_bool[indices[i]] = true;

}

i\_bool\_dev = i\_bool;

matrAB\_dev = matrAB\_host;

AB\_ptr = thrust::raw\_pointer\_cast(matrAB\_dev.data());

// reversing step

for (int i = 0; i < freeX - 1; ++i) {

int curRow = indices2[freeX - i - 1];

int curCol = indices[freeX - i - 1];

backStep << <gridSz, blockSz >> > (AB\_ptr, curRow, curCol);

//CSC(cudaThreadSynchronize());

backKill << <BLOCK, THREAD >> > (AB\_ptr, curRow, curCol);

CSC(cudaThreadSynchronize());

}

device\_vector<int> indices\_dev;

device\_vector<int> indices2\_dev;

host\_vector<double> diags(freeX);

device\_vector<double> diags\_dev;

matrAB\_host = matrAB\_dev;

for (int i = 0; i < freeX; ++i) {

diags[i] = matrAB\_host[indices2[i] + n \* indices[i]];

}

indices\_dev = indices;

indices2\_dev = indices2;

diags\_dev = diags;

int\* i\_ptr = thrust::raw\_pointer\_cast(indices\_dev.data());

int\* i2\_ptr = thrust::raw\_pointer\_cast(indices2\_dev.data());

double\* d\_ptr = thrust::raw\_pointer\_cast(diags\_dev.data());

//divide rights

division << <gridSz, blockSz >> > (AB\_ptr, d\_ptr, i\_ptr, i2\_ptr, freeX);

CSC(cudaThreadSynchronize());

matrAB\_host = matrAB\_dev;

//AtoE

for (int i = 0; i < freeX; ++i) {

int curRow = indices2[i];

int curCol = indices[i];

matrAB\_host[curRow + curCol \* n] = 1;

}

int t = 0;

//filling XMatrix

for (int i = 0; i < m; ++i) {

int curXRow;

if (indices.size() > t) {

curXRow = indices[t];

if (i == curXRow) {

for (int j = 0; j < k; ++j) {

printf("%.10e ", matrAB\_host[t + n \* j + n \* m]);

}

printf("\n");

t += 1;

}

else {

for (int j = 0; j < k; ++j) {

printf("%.10e ", 0.0);

}

printf("\n");

}

}

else {

for (int j = 0; j < k; ++j) {

printf("%.10e ", 0.0);

}

printf("\n");

}

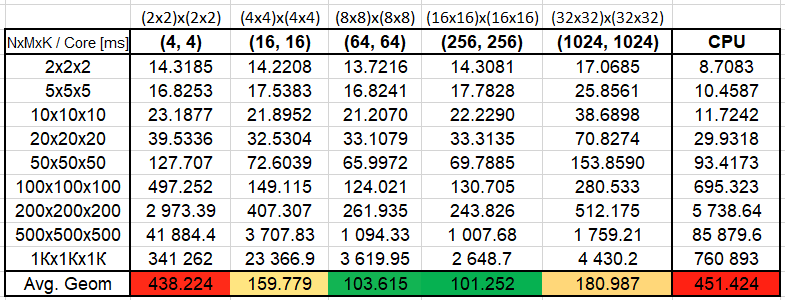
}

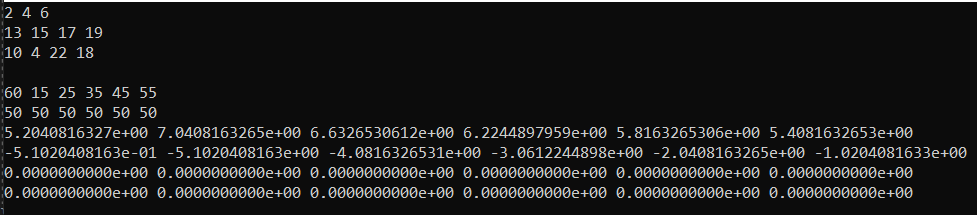
return 0;

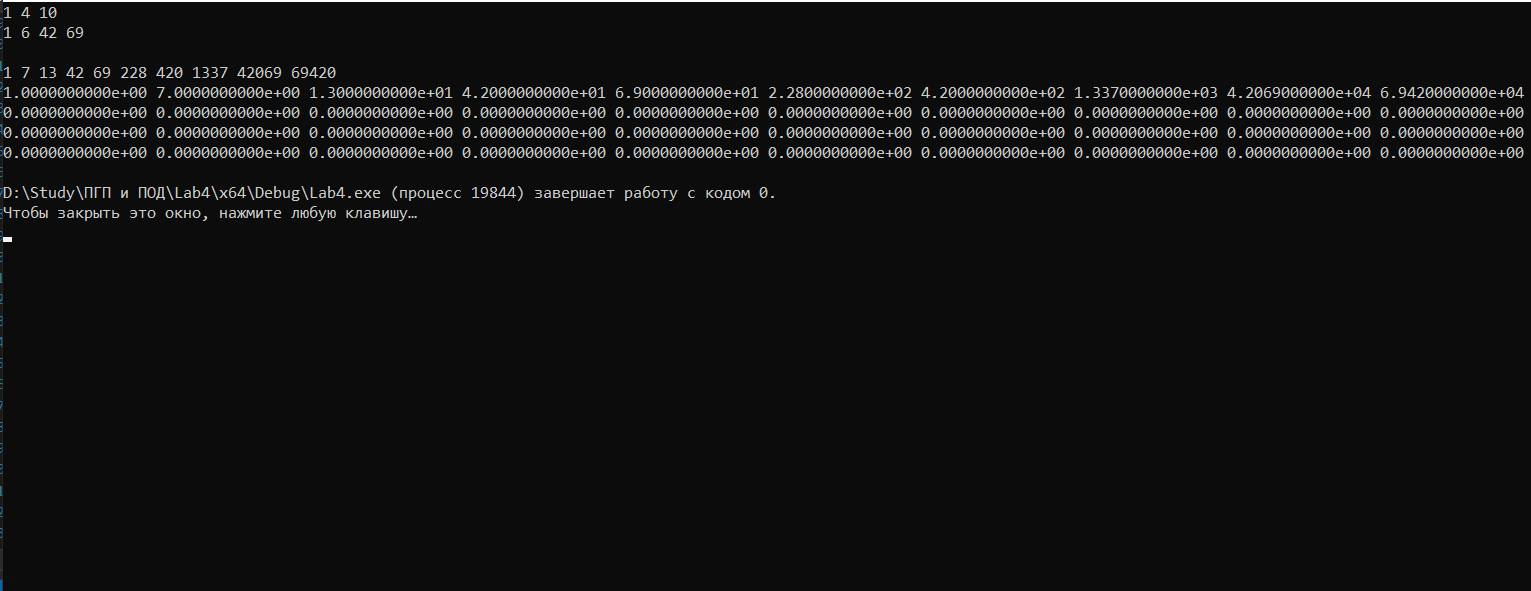
}

**Результаты**

Приведу таблицу и пару примеров.







**Выводы**

В данной лабораторной мне пришлось очень долго возиться с матрицами и линейной алгеброй. Задача не из самых приятных. Тем более, очень легко допустить ошибку, при которой происходит гонка потоков – один процесс меняет элемент, который использует другой процесс, предполагая, что он не изменился. Поэтому, приходится разделять ядра так, чтобы не было зависящих друг от друга элементов в каждом ядре. Также очень необходимо было оптимизировать все это. Хранение матриц в формате столбцов, как это реализовано в библиотеке CUBlas. Из-за того, что программа проходила по ненужным элементам (левее главной диагонали, например), у меня она не проходила по времени. Поэтому, данная работа была очень трудной для меня, и мне понадобилась неоднократная помощь преподавателя. С его помощью я смог завершить задание!