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 X заполняется нулями, а строки, соответствующие конкретным столбцам заменяются на строки с матрицы B (видоизмененной после приведения матрицы A к треугольному, а потом и единичному виду). Мы просто сопрягаем ее с матрицей A 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.

	(2x2)x(2x2)	(4x4)x(4x4)	(8x8)x(8x8)	(16x16)x(16x16)	(32x32)x(32x32)	
NxMxK / Core [ms]	**(4, 4)**	**(16, 16)**	**(64, 64)**	**(256, 256)**	**(1024, 1024)**	**CPU**
2x2x2	14.3185	14.2208	13.7216	14.3081	17.0685	8.7083
5x5x5	16.8253	17.5383	16.8241	17.7828	25.8561	10.4587
10x10x10	23.1877	21.8952	21.2070	22.2290	38.6898	11.7242
20x20x20	39.5336	32.5304	33.1079	33.3135	70.8274	29.9318
50x50x50	127.707	72.6039	65.9972	69.7885	153.8590	93.4173
100x100x100	497.252	149.115	124.021	130.705	280.533	695.323
200x200x200	2 973.39	407.307	261.935	243.826	512.175	5 738.64
500x500x500	41 884.4	3 707.83	1 094.33	1 007.68	1 759.21	85 879.6
1Kx1Kx1K	341 262	23 366.9	3 619.95	2 648.7	4 430.2	760 893
Avg. Geom	438.224	159.779	103.615	101.252	180.987	451.424

```
2 4 6
13 15 17 19
10 4 22 18

60 15 25 35 45 55
50 50 50 50 50 50
5.2040816327e+00 7.0408163265e+00 6.6326530612e+00 6.2244897959e+00 5.8163265306e+00 5.4081632653e+00
-5.1020408163e-01 -5.1020408163e+00 -4.0816326531e+00 -3.0612244898e+00 -2.0408163265e+00 -1.0204081633e+00
0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00
0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00
```

```
1 4 10
1 6 42 69

1 7 13 42 69 228 420 1337 42069 69420
1.0000000000e+00 7.0000000000e+00 1.3000000000e+01 4.2000000000e+01 6.9000000000e+01 2.2800000000e+02 4.2000000000e+02 1.3370000000e+03 4.2069000000e+04 6.9420000000e+04
0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00
0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00
0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00

D:\Study\ПГП и ПОД\Lab4\x64\Debug\Lab4.exe (процесс 19844) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу…
```

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!