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 матрицам. Метод Гаусса.

Выполнил: Симонов С.Я.
Группа: 8О-406Б-18
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.

Вариант 5. Решение произвольной СЛАУ.

Программное и аппаратное обеспечение
GPU:

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

Сведения о системе:

Операционная система: Ubuntu 14.04 LTS 64-x битная
Рабочая среда: nano
Компилятор: nvcc

Метод решения

В данной лабораторной работе, было реализовано три ядра: swap_gen(), back(), kernel().

Изначально я считываю входные данные по принципу озвученному на лекции, причем вектор b записывается как последний столбец в нашу матрицу с значениями. Также я создаю массив “b” в котором на i-й позиции храню строку в которой необходимо будет выполнить обратный проход по методу Гаусса. “result” - хранит в себе конечный результат. Оба вектора создаются размером m.

Затем я прохожусь по всем столбцам и строкам заданной матрицы(если закончились строки, то заканчиваем проход, аналогично для столбцов), с целью приведения ее к верхнему треугольному виду. На каждой итерации данного цикла при помощи библиотеки thrust вычисляем максимальный элемент в текущем столбце и если он больше 1e-7, то выполняем прямой проход по методу Гаусса (kernel) при условии, что текущий элемент ij равен максимальному. Иначе меняем местами текущий и максимальный.

Наконец, выполняется обратный проход (back), начиная с последнего столбца. Если текущий элемент массива b больше нуля, то для данного столбца выполняется back.

Выводим полученные результаты.

Описание программы

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <iomanip>
#include <thrust/extrema.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.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)

struct comparator {
        __host__ __device__ bool operator()(double a, double b) {
                return fabs(a) < fabs(b);
        }
};

__global__ void swap_gen(double* data, int n, int m, int j, int max) {
        int idx = blockDim.x * blockIdx.x + threadIdx.x;
        int offset = blockDim.x * gridDim.x;
        for (int i = idx; i < m + 1; i += offset) {
                thrust::swap(data[i * n + j], data[i * n + max]);
        }
}

__global__ void back(double* data, int* b, int j, int n, int m, double* result) {
        int idx = blockDim.x * blockIdx.x + threadIdx.x;
        int offset = blockDim.x * gridDim.x;
        result[j] = data[n * m + b[j] - 1] / data[j * n + b[j] - 1];
```

```
	for (int i = j - idx - 1; i >= 0; i -= offset) {
		if (b[i] > 0) {
			int w = b[i];
			data[n * m + w - 1] -= result[j] * data[j * n + w - 1];
		}
	}
}

__global__ void kernel(double* data, int i, int jj, int n, int m) {
	int idx = blockDim.x * blockIdx.x + threadIdx.x;
	int idy = blockDim.y * blockIdx.y + threadIdx.y;
	int offsetx = blockDim.x * gridDim.x;
	int offsety = blockDim.y * gridDim.y;
	for (int j = idx + i + 1; j < n; j += offsetx) {
		for (int k = idy + jj + 1; k < m + 1; k += offsety) {
			data[k * n + j] -= data[k * n + i] * data[jj * n + j] / data[jj * n + i];
		}
	}
}

int main() {
	std::ios_base::sync_with_stdio(0);
	std::cin.tie(0);
	int n, m;
	std::cin >> n >> m;
	double* data, * dev_data, * dev_result;
	int t = m + 1, * dev_b;
	data = (double*)malloc(sizeof(double) * n * t);
	int* b = (int*)malloc(sizeof(int) * m);
	double* result = (double*)malloc(sizeof(double) * m);
	memset(result, 0, sizeof(double) * m);
	for (int i = 0; i < m; ++i) {
		b[i] = 0;
	}
	CSC(cudaMalloc(&dev_data, sizeof(double) * n * t));
	CSC(cudaMalloc(&dev_b, sizeof(int) * m));
	CSC(cudaMalloc(&dev_result, sizeof(double) * m));
	for (int i = 0; i < n; ++i) {
		for (int j = 0; j < m; ++j) {
			std::cin >> data[i + j * n];
		}
	}
	for (int i = 0; i < n; ++i) {
		std::cin >> data[i + n * m];
	}
	CSC(cudaMemcpy(dev_data, data, sizeof(double) * n * t,
cudaMemcpyHostToDevice));
	comparator comp;
```

```
	double tmp;
	int max;
	thrust::device_ptr<double> p_arr, res;
	for (int j = 0, i = 0; i < n && j < m; ++i, ++j) {
		p_arr = thrust::device_pointer_cast(dev_data + j * n);
		res = thrust::max_element(p_arr + i, p_arr + n, comp);
		max = res - p_arr;
		tmp = fabs(*res);
		if (tmp > 1e-7) {
			if (i != max) {
				swap_gen << <512, 512 >> > (dev_data, n, m, i, max);
				CSC(cudaGetLastError());
			}
			b[j] = i + 1;
			kernel << <dim3(32, 32), dim3(32, 32) >> > (dev_data, i, j, n, m);
			CSC(cudaGetLastError());
		}
		else {
			--i;
		}
	}
	CSC(cudaMemcpy(dev_b, b, sizeof(int) * m, cudaMemcpyHostToDevice));
	CSC(cudaMemcpy(dev_result, result, sizeof(double) * m,
cudaMemcpyHostToDevice));
	for (int i = m - 1; i >= 0; --i) {
		if (b[i] > 0) {
			back << <512, 512 >> > (dev_data, dev_b, i, n, m, dev_result);
			CSC(cudaGetLastError());
		}
	}
	CSC(cudaMemcpy(result, dev_result, sizeof(double) * m,
cudaMemcpyDeviceToHost));
	for (int i = 0; i < m; ++i) {
		printf("%.10e ", result[i]);
	}
	free(data);
	return 0;
}
```

Результаты

На вход подавались матрицы размерами 10*10, 100*100, 1000*1000, 5000*5000.

Конфигурация	Время работы (ms)
CPU	0.002376
<<<dim3(4, 4), dim3(4, 4)>>>	0.791577
<<<dim3(16, 16), dim3(16, 16)>>>	0.671324
<<<dim3(32, 32), dim3(32, 32)>>>	1.973892
CPU	126345.754361
<<<dim3(4, 4), dim3(4, 4)>>>	33571.791652
<<<dim3(16, 16), dim3(16, 16)>>>	8286.417382
<<<dim3(32, 32), dim3(32, 32)>>>	9338.065916
CPU	982575.021441
<<<dim3(4, 4), dim3(4, 4)>>>	251043.837509
<<<dim3(16, 16), dim3(16, 16)>>>	64091.281590
<<<dim3(32, 32), dim3(32, 32)>>>	72846.674246
CPU	3504318.398504
<<<dim3(4, 4), dim3(4, 4)>>>	411598.871272
<<<dim3(16, 16), dim3(16, 16)>>>	13732.835693
<<<dim3(32, 32), dim3(32, 32)>>>	16032.107354

Выводы

Выполнив данную лабораторную работу мной был реализован алгоритм решения произвольной СЛАУ методом Гаусса. Как видно из результатов работы распараллеленный алгоритм многократно превосходит реализацию на CPU. Большой проблемой при выполнении лабораторной работы стало обращение к глобальной памяти при помощи объединения запросов. По данному вопросу понимание пришло не сразу, но в конечном итоге я разобрался что же не так. Кроме того, я потрогал библиотеку thrust, с ней тоже были проблемы, но они исходили из чистой невнимательности.