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.

Выполнила: Алексюнина Ю.В.

Группа:М8О-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: 4. LU-разложение матрицы.

Программное и аппаратное обеспечение

GPU:
- Name: GeForce GTX 750 Ti
- Compute capability: 5.0
- Графическая память: 4294967295
- 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: 5

Сведения о системе:
- Процессор: Intel Core i5-4460 3.20GHz
- ОЗУ: 16 ГБ
- HDD: 930 ГБ

Программное обеспечение:
- OS: Windows 8.1
- IDE: Visual Studio 2019
- Компилятор: nvcc

Метод решения:

В задании требуется найти элементы матриц L и U, объединенные в одну матрицу.

В цикле от 0 до N осуществим следующие действия для получения объединенной матрицы:

1. Найдем максимальный по модулю элемент в i-том столбце ниже главной диагонали и номер строки j, в которой находится этот элемент.
2. Поменяем i-тую и j-тую строки местами.
3. Поделим все элементы i-того столбца на i-тый элемент на главной диагонали
4. Применим к оставшимся элементам матрицы формулу

$$a[k][j] = a[k][j] – a[k][i] * a[i][j]$$

Описание программы:

Изначально я вспомнила, как реализовывала подобную задачу в курсе численных методов, и переписала ее для CPU, с языка C# на C++.

В реализации на GPU было использовано 2 ядра: kernel_swap – для пункта 2 алгоритма и kernel_LUP – пункты 3 и 4.

Чтобы быстрее поменять элементы местами, используется метод swap из thrust:

```
thrust::swap(A[i + N * k], A[max_idx + N * k]);
```

Кроме этого, из библиотеки thrust также используется метод для нахождения максимального элемента:

```
data_ptr = thrust::device_pointer_cast(dev_A + i * N);// + i * N);

max_ptr = thrust::max_element(data_ptr + i, data_ptr + N, comp); //+i, +N

max_idx = max_ptr - data_ptr;
```

Метод max_element использует компаратор:

```
struct abs_comp
{
    __host__ __device__ bool operator()(double x, double y)
    {
        return fabs(x) < fabs(y);
    }
};
```

Файл kernel.cu:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <algorithm>
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <vector>
#include <stdio.h>

#include <thrust/swap.h>
#include <thrust/device_vector.h>
#include <thrust/extrema.h> //swap

using namespace std;

#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 abs_comp
{
    __host__ __device__ bool operator()(double x, double y)
```

```
    {
        return fabs(x) < fabs(y);
    }
};


__global__ void kernel_swap(double* A, int N, int i, int max_idx)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int offsetX = gridDim.x * blockDim.x;

    for (int k = idx; k < N; k += offsetX)
    {
        //thrust::swap(A[i][k], A[max_idx][k]);
        thrust::swap(A[i + N * k], A[max_idx + N * k]);
    }
}
__global__ void kernel_LUP(double* A, int N, int i)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int offsetx = gridDim.x * blockDim.x;
    int offsety = gridDim.y * blockDim.y;

    //compute LU
    for (int k = i + 1 + idx; k < N; k += offsetx)
    {
       // A[k][i] /= A[i][i];
        A[k + N * i] /= A[i + N * i];
        for (int j = i + 1 + idy; j < N; j += offsety) // iterate
across rows
           // A[k][j] = A[k][j] - A[i][j] * A[k][i];
            A[k + N * j] = A[k + N * j] - A[i + N * j] * A[k + N * i];
    }
```

```
}

int main()
{
    ios_base::sync_with_stdio(false);
    //std::cin.tie(nullptr);
    //init
    int N;
    scanf("%d",&N);
    //cin >> N;
    double* A, * dev_A;
    CSC(cudaMalloc((void**)&dev_A, sizeof(double) * N * N));
    int* P = (int*)malloc(sizeof(int) * N);
    A = (double*)malloc(sizeof(double) * N * N);

    for (int i = 0; i < N; i++) {
        P[i] = i;
     double t;
        for (int j = 0; j < N; j++) {
            //cin >> A[i + N * j];
         scanf("%lf", &t);
         A[i + N * j] = t;
        }
    }

    CSC(cudaMemcpy(dev_A, A, sizeof(double) * N * N,
cudaMemcpyHostToDevice));

    int max_idx;
    abs_comp comp;
    thrust::device_ptr<double> data_ptr;
    thrust::device_ptr<double> max_ptr;
```

```
    for (int i = 0; i < N - 1; ++i) {
        max_idx = i;

        data_ptr = thrust::device_pointer_cast(dev_A + i * N);// + i *
N);
        max_ptr = thrust::max_element(data_ptr + i, data_ptr + N,
comp); //+i, +N
        max_idx = max_ptr - data_ptr;



        P[i] = max_idx;
        if (max_idx != i) {
            kernel_swap << <256, 256 >> > (dev_A, N, i, max_idx);
            CSC(cudaGetLastError());
        }

        kernel_LUP << <256, 256 >> > (dev_A, N, i);
        CSC(cudaGetLastError());
        CSC(cudaThreadSynchronize());

    }

    CSC(cudaMemcpy(A, dev_A, sizeof(double) * N * N,
cudaMemcpyDeviceToHost));
    CSC(cudaFree(dev_A));

    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            printf("%.10e ", A[i + N * j]);
        }
        printf("\n");
    }

    for (int j = 0; j < N; ++j) {
        printf("%d ", P[j]);
```

```
    }

    free(A);
    free(P);

    return 0;
}
```

Результаты:

Размер матрицы	GPU	CPU
500x500	1012ms	9349ms
1000x1000	2625ms	74262ms
1500x1500	5104ms	250487ms
2000x2000	8194ms	595392ms
2500x2500	11975ms	1160041ms
3000x3000	16929ms	2355710ms

Выводы:

Из результатов видно, что использование GPU с применением thrust существенно ускоряет работу с матрицами, по сравнению с CPU.

По итогу данной лабораторной работы я получила знания по использованию библиотеки thrust и осознала, что вычисления и работа с матрицами на GPU могут быть существенно упрощены по времени.