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 матрицам. Метод Гаусса.

Выполнил: А.Т. Бахарев
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2020

Условие

1. Вариант: Вычисление детерминанта матрицы

Программное и аппаратное обеспечение

GPU:
Название: GeForce GTX 750TI
Графическая память: 2094071808
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: AMD FX-8320 3.5Ghz
Оперативная память: 16Gb
HDD: 1Tb
Операционная система: Ubuntu 18.04
IDE: VSC
Компилятор: nvcc

Метод решения

Для численного нахождения определителя квадратной матрицы, необходимо использовать LU-разложение матрицы.

LU – разложение матрицы A представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т.е.

$$A = LU ,$$

где L - нижняя треугольная матрица (матрица, у которой все элементы, находящиеся выше главной диагонали равны нулю, l_^ij = 0 при i < j), U - верхняя треугольная матрица
(матрица, у которой все элементы, находящиеся ниже главной диагонали равны нулю, u_^ij = 0 при i > j).

L — матрица коэффицентов, при использовании которых мы можнм привести исходную матрицу A к верхнетреугольному виду — к матрице U.

Определитель(детерминант) матрицы находится простым перемножением диагональных элементов матриц L и U.

Описание программы

Исходную матрицу транспонируем, чтобы обьединить запросы к глобальной памяти. Основное время работы алгоритма LU — это добавление одной строки ко всем другим, умноженной на коэффицент. После транспонирования варп будет обрабатывать подряд идущие блоки памяти, за счет чего время работы будет меньше.

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

#define THREADS_PER_BLOCK 256
#define BLOCKS_PER_GRID 256

#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)

__host__ void gpu_print_matrix(double* matrix, int size)
{
        for (int i = 0; i < size; ++i)
        {
                for (int j = 0; j < size; ++j)
                {
                        printf("%.1f ", matrix[i * size + j]);
                }
                printf("\n");
        }
}

__global__ void gpu_transpose(double* matrix, int size)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int offsetx = gridDim.x * blockDim.x;
   double temp;
   int curr_row;
   int curr_col;
   while(idx < size * size)
   {
      curr_row = idx / size;
      curr_col = idx % size;
```

```
        if(curr_col > curr_row)
        {
           temp = matrix[curr_row * size + curr_col];
           matrix[curr_row * size + curr_col] = matrix[curr_col * size + curr_row];
           matrix[curr_col * size + curr_row] = temp;
        }
              idx += offsetx;

           }
}

__global__ void gpu_swap(double* matrix, int size, int row_from, int row_to)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int offsetx = gridDim.x * blockDim.x;
        double tmp;
        for (int i = idx; i < size; i += offsetx)
        {
                tmp = matrix[(i * size) + row_from];
                matrix[(i * size) + row_from] = matrix[(i * size) + row_to];
                matrix[(i * size) + row_to] = tmp;10000*10000
        }
}

double* multiplication(double* lhs, double* rhs, int n)
{
   double* res = (double*) malloc(sizeof(double) * n * n);

   res = (double*) calloc(n * n, sizeof(double));


   for(int i = 0; i < n; ++i)
   {
      for(int j = 0; j < n; ++j)
      {
         for(int t = 0; t < n; ++t)
         {
            res[j * n + i] += lhs[j * n + t] * rhs[t * n + i];
         }
      }
   }
   return res;
}

__global__ void gpu_compute_L(double* matrix, double* L, int size, int curr_row)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int offsetx = gridDim.x * blockDim.x;

        for(; idx < size; idx += offsetx)
```

```
	{
		if(idx < curr_row)
			continue;

		if(idx == curr_row )
		{
			L[curr_row * size + curr_row] = 1.0;
		}

		else if(fabs(matrix[curr_row * size + curr_row]) > 10e-7)
		{
			L[curr_row * size + idx] = matrix[curr_row * size + idx] /
matrix[curr_row * size + curr_row];
		}
	}
}

__global__ void gpu_modify_matrix(double* matrix, double* L, int size, int max_col)
{
	int idx = blockIdx.x * blockDim.x + threadIdx.x;
	int offsetx = gridDim.x * blockDim.x;
	int curr_row;
	int curr_col;
	for(; idx < size * size; idx += offsetx)
	{
		curr_row = idx / size;
		curr_col = idx % size;
		if(curr_col == max_col)
			continue;
		else
		{
			matrix[curr_row * size + curr_col] -= L[max_col * size + curr_col] *
matrix[curr_row * size + max_col];
		}
	}
}

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

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

```
	for (int i = 0; i < n * n; ++i)
	{
		scanf("%lf", &matrix[i]);
	}
	double* matrix_dev;
	CSC(cudaMalloc(&matrix_dev, sizeof(double) * n * n));
	CSC(cudaMemcpy(matrix_dev, matrix, sizeof(double) * n * n,
cudaMemcpyHostToDevice));


	double* L = (double*) calloc(n * n, sizeof(double));
	double* L_dev;
	CSC(cudaMalloc(&L_dev, sizeof(double) * n * n));
	CSC(cudaMemcpy(L_dev, L, sizeof(double) * n * n, cudaMemcpyHostToDevice));

	int pos_of_max;
	int sign = 1;
	comparator comp;
	thrust::device_ptr<double> p_matrix = thrust::device_pointer_cast(matrix_dev);
	thrust::device_ptr<double> max_elem;

	cudaEvent_t start, end;
	CSC(cudaEventCreate(&start));
	CSC(cudaEventCreate(&end));
	CSC(cudaEventRecord(start));

	gpu_transpose << <BLOCKS_PER_GRID, THREADS_PER_BLOCK >> >
(matrix_dev, n);

	for (int row = 0; row < n; ++row)
	{
		max_elem = thrust::max_element(p_matrix + (row * n) + row, p_matrix +
((row + 1) * n), comp);
		pos_of_max = (int)(max_elem - p_matrix) % n;

		if(row != pos_of_max)
		{
			sign *= -1;
			gpu_swap<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>
(matrix_dev, n, row, pos_of_max);
		}

		gpu_compute_L << <BLOCKS_PER_GRID, THREADS_PER_BLOCK >> >
(matrix_dev, L_dev, n, row);
		gpu_modify_matrix <<< BLOCKS_PER_GRID, THREADS_PER_BLOCK
>>> (matrix_dev, L_dev, n, row);
	}
	//
	CSC(cudaEventRecord(end));
	CSC(cudaEventSynchronize(end));
```

```
	float t;
	CSC(cudaEventElapsedTime(&t, start, end));
	CSC(cudaEventDestroy(start));
	CSC(cudaEventDestroy(end));
	printf("GPU time = %.2fms\n", t);

	CSC(cudaMemcpy(L, L_dev, sizeof(double) * n * n, cudaMemcpyDeviceToHost));
	CSC(cudaMemcpy(matrix, matrix_dev, sizeof(double) * n * n,
cudaMemcpyDeviceToHost));

	long double det = 1;
	for(int i = 0; i < n; ++i)
	{
		det *= matrix[i * n + i] * L[i * n + i];
	}
	if(fabs(det) <= 10e-7)
		printf("%.10Lf\n", det);

	else
		printf("%.10Lf\n", det * sign);

	free(matrix);
	free(L);
	cudaFree(matrix_dev);
	cudaFree(L_dev);
}
```

Результаты

N	CPU(ms)	<<<64, 64>>>	<<<128, 128>>>	<<<256, 256>>>
200*200	19.87	12.68	12.97	12.86

N	CPU(ms)	<<<64, 64>>>	<<<128, 128>>>	<<<256, 256>>>
2000*2000	19661.87	1942.51	1923.83	1881.98

Выводы

Выполнив 4ую лабораторную работу, я научился работать с матрицами на GPU. Я закрепил навыки нахождения независимых участков алгоритма, которые можно обрабатывать одновременно и независимо друг от друга. Так же, я познакомился с таким понятием, как обьединение запросов к глобальной памяти. Получается довольно серьезный прирост производительности за счет иного представления данных в глобальной памяти видеокарты. Наконец, я повторил алгоритм LU и реализовал его еще раз. Эти знания будут очень полезны для написания диплома.