МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №2
по курсу «Параллельная обработка данных»

Технология MPI и технология CUDA. MPI-IO

Выполнил: Симонов С.Я.
Группа: 8О-406Б-18
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Цель работы: Совместное использование технологии MPI и технологии CUDA. Применение библиотеки алгоритмов для параллельных расчетов Thrust. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода. Использование механизмов MPI-IO и производных типов данных.

Требуется решить задачу описанную в лабораторной работе No7, используя возможности графических ускорителей установленных на машинах вычислительного кластера. Учесть возможность наличия нескольких GPU в рамках одной машины. На GPU необходимо реализовать основной расчет. Требуется использовать объединение запросов к глобальной памяти. На каждой итерации допустимо копировать только граничные элементы с GPU на CPU для последующей отправки их другим процессам. Библиотеку Thrust использовать только для вычисления погрешности в рамках одного процесса.

Вариант 1. MPI_Type_create_subarray

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

Основная суть решения по отношению к предыдущей лабораторной работе не изменилась. Основными изменениями стало параллельный обмен граничными условиями, также параллельное вычисление основной формулы и использование MPI_Type_create_subarray.
Чтобы запись в файл всеми процессами была корректной необходимо определить правила записи в файл, для этого нам и нужен MPI_Type_create_subarray. Данный метод является пожалуй самым удобным в применении к подобным задачам, поскольку мы заранее говорим сколько элементов должно быть записано(задаем размер изначальной сетки), сколько элементов будет записано одним процессом(размер сетки обрабатываемой одним процессом) и начальное положение записи того или иного куска результатов.
Также были реализованы 12 ядер:

1) Вычисление основной формулы (formula)

2) Вычисление погрешности (epsil)
3) Остальные ядра реализуют обмен значений.

Описание программы

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <string>
#include <fstream>
#include <cmath>
#include "mpi.h"
#include <thrust/extrema.h>
#include <thrust/device_vector.h>
#define _i(i, j, k) (((k) + 1) * (y + 2) * (x + 2) + ((j) + 1) * (x + 2) + (i) + 1)  // от двумерной индексации к одномерной
#define _ibx(id) (((id) % (dim1 * dim2) ) % dim1) // как по номеру элемента получить индексы
#define _iby(id) (((id) % (dim1 * dim2) ) / dim1)
#define _ibz(id) ((id) / (dim1 * dim2) )
#define _ib(i, j, k) ((k) * (dim1 * dim2) + (j) * dim1 + (i)) // переход от двуменой к одноменорной индексации процессов

#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)

__global__ void formula(int x, int y, int z, double* next, double* data, double hx, double hy, double hz, double* check) {
        int i, j, k;
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int idy = blockIdx.y * blockDim.y + threadIdx.y;
        int idz = blockIdx.z * blockDim.z + threadIdx.z;
        int offsetx = blockDim.x * gridDim.x;
        int offsety = blockDim.y * gridDim.y;
        int offsetz = blockDim.z * gridDim.z;
        for (i = idx; i < x; i += offsetx) {
                for (j = idy; j < y; j += offsety) {
                        for (k = idz; k < z; k += offsetz) {
```

```
                                next[_i(i, j, k)] = 0.5 * ((data[_i(i + 1, j, k)] + data[_i(i - 1, j, k)]) /
(hx * hx) + (data[_i(i, j + 1, k)] + data[_i(i, j - 1, k)]) / (hy * hy) + (data[_i(i, j, k + 1)] + data[_i(i,
j, k - 1)]) / (hz * hz)) / (1.0 / (hx * hx) + 1.0 / (hy * hy) + 1.0 / (hz * hz));
                        }
                }
        }
}

__global__ void epsil(double* next, double* data, double* check, int x, int y, int z){
        int i, j, k;
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int idy = blockIdx.y * blockDim.y + threadIdx.y;
        int idz = blockIdx.z * blockDim.z + threadIdx.z;
        int offsetx = blockDim.x * gridDim.x;
        int offsety = blockDim.y * gridDim.y;
        int offsetz = blockDim.z * gridDim.z;
        for(i = idx - 1; i <= x; i += offsetx ) {
                for(j = idy - 1; j <= y; j += offsety ) {
                        for(k = idz - 1; k <= z; k += offsetz ) {
                                if( (i != -1) && (j != -1) && (k != -1) && (i != x) && (j != y) && (k
!= z) ){
                                        check[_i(i, j, k)] = abs(next[_i(i, j, k)] - data[_i(i, j, k)]);
                                } else{
                                        check[_i(i, j, k)] = 0.0;
                                }
                        }
                }
        }
}

__global__ void ib1_kernel(int x, int y, int z, double* buff, double* data, bool flag) {
        int k, j;
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int idy = blockIdx.y * blockDim.y + threadIdx.y;
        int offsetx = blockDim.x * gridDim.x;
        int offsety = blockDim.y * gridDim.y;
        if (flag) {
                for (k = idy; k < z; k += offsety) {
                        for (j = idx; j < y; j += offsetx) { //
                                buff[j + k * y] = data[_i(x - 1, j, k)]; // тогда копируем одну
границу нашей области в буффер
                        }
                }
        } else {
                for (k = idy; k < z; k += offsety) {
                        for (j = idx; j < y; j += offsetx) { //
                                buff[j + k * y] = data[_i(0, j, k)]; // тогда копируем одну границу
нашей области в буффер
```

```
                        }
                }
        }
}

__global__ void jb1_kernel(int x, int y, int z, double* buff, double* data, bool flag) {
        int i, k;
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int idy = blockIdx.y * blockDim.y + threadIdx.y;
        int offsetx = blockDim.x * gridDim.x;
        int offsety = blockDim.y * gridDim.y;
        if (flag) {
                for (k = idy; k < z; k += offsety) {
                        for (i = idx; i < x; i += offsetx) { //
                                buff[i + k * x] = data[_i(i, y - 1, k)]; //
                        }
                }
        } else {
                for (k = idy; k < z; k += offsety) {
                        for (i = idx; i < x; i += offsetx) { //
                                buff[i + k * x] = data[_i(i, 0, k)]; //
                        }
                }
        }
}

__global__ void kb1_kernel(int x, int y, int z, double* buff, double* data, bool flag) {
        int i, j;
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int idy = blockIdx.y * blockDim.y + threadIdx.y;
        int offsetx = blockDim.x * gridDim.x;
        int offsety = blockDim.y * gridDim.y;
        if (flag) {
                for (j = idy; j < y; j += offsety) {
                        for (i = idx; i < x; i += offsetx) { //
                                buff[i + j * x] = data[_i(i, j, z - 1)]; //
                        }
                }
        } else {
                for (j = idy; j < y; j += offsety) {
                        for (i = idx; i < x; i += offsetx) { //
                                buff[i + j * x] = data[_i(i, j, 0)]; //
                        }
                }
        }
}

__global__ void ib2_kernel(int x, int y, int z, double* data, double* buff, bool flag) {
```

```
	int k, j;
	int idx = blockIdx.x * blockDim.x + threadIdx.x;
	int idy = blockIdx.y * blockDim.y + threadIdx.y;
	int offsetx = blockDim.x * gridDim.x;
	int offsety = blockDim.y * gridDim.y;
	if (flag) {
		for (k = idy; k < z; k += offsety) {
			for (j = idx; j < y; j += offsetx) {
				data[_i(x, j, k)] = buff[j + k * y]; //
			}
		}
	} else {
		for (k = idy; k < z; k += offsety) {
			for (j = idx; j < y; j += offsetx) {
				data[_i(-1, j, k)] = buff[j + k * y]; //
			}
		}
	}
}

__global__ void jb2_kernel(int x, int y, int z, double* data, double* buff, bool flag) {
	int i, k;
	int idx = blockIdx.x * blockDim.x + threadIdx.x;
	int idy = blockIdx.y * blockDim.y + threadIdx.y;
	int offsetx = blockDim.x * gridDim.x;
	int offsety = blockDim.y * gridDim.y;
	if (flag) {
		for (k = idy; k < z; k += offsety) {
			for (i = idx; i < x; i += offsetx) {
				data[_i(i, y, k)] = buff[i + k * x]; // копируем данные
			}
		}
	} else {
		for (k = idy; k < z; k += offsety) {
			for (i = idx; i < x; i += offsetx) {
				data[_i(i, -1, k)] = buff[i + k * x]; // копируем данные
			}
		}
	}
}

__global__ void kb2_kernel(int x, int y, int z, double* data, double* buff, bool flag) {
	int i, j;
	int idx = blockIdx.x * blockDim.x + threadIdx.x;
	int idy = blockIdx.y * blockDim.y + threadIdx.y;
	int offsetx = blockDim.x * gridDim.x;
	int offsety = blockDim.y * gridDim.y;
	if (flag) {
```

```
                for (j = idy; j < y; j += offsety) {
                        for (i = idx; i < x; i += offsetx) {
                                data[_i(i, j, z)] = buff[i + j * x]; // копируем данные
                        }
                }
        } else {
                for (j = idy; j < y; j += offsety) {
                        for (i = idx; i < x; i += offsetx) {
                                data[_i(i, j, -1)] = buff[i + j * x]; // копируем данные
                        }
                }
        }
}

__global__ void ib3_kernel(int x, int y, int z, double* data, double u, bool flag) {
        int k, j;
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int idy = blockIdx.y * blockDim.y + threadIdx.y;
        int offsetx = blockDim.x * gridDim.x;
        int offsety = blockDim.y * gridDim.y;
        if (flag) {
                for (k = idy; k < z; k += offsety) {
                        for (j = idx; j < y; j += offsetx) {
                                data[_i(x, j, k)] = u;
                        }
                }
        } else {
                for (k = idy; k < z; k += offsety) {
                        for (j = idx; j < y; j += offsetx) {
                                data[_i(-1, j, k)] = u;
                        }
                }
        }
}

__global__ void jb3_kernel(int x, int y, int z, double* data, double u, bool flag) {
        int i, k;
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int idy = blockIdx.y * blockDim.y + threadIdx.y;
        int offsetx = blockDim.x * gridDim.x;
        int offsety = blockDim.y * gridDim.y;
        if (flag) {
                for (k = idy; k < z; k += offsety) {
                        for (i = idx; i < x; i += offsetx) {
                                data[_i(i, y, k)] = u;
                        }
                }
        } else {
```

```
                for (k = idy; k < z; k += offsety) {
                        for (i = idx; i < x; i += offsetx) {
                                data[_i(i, -1, k)] = u;
                        }
                }
        }
}

__global__ void kb3_kernel(int x, int y, int z, double* data, double u, bool flag) {
        int i, j;
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int idy = blockIdx.y * blockDim.y + threadIdx.y;
        int offsetx = blockDim.x * gridDim.x;
        int offsety = blockDim.y * gridDim.y;
        if (flag) {
                for (j = idy; j < y; j += offsety) {
                        for (i = idx; i < x; i += offsetx) {
                                data[_i(i, j, z)] = u;
                        }
                }
        } else {
                for (j = idy; j < y; j += offsety) {
                        for (i = idx; i < x; i += offsetx) {
                                data[_i(i, j, -1)] = u;
                        }
                }
        }
}

int main(int argc, char *argv[]) {
        int numproc, id, i, j, k, dim1, dim2, dim3_3, x, y, z, ib, jb, kb, max;
        // std::string fi;
        char fi[128];
        double eps, l_x, l_y, l_z, u_down, u_up, u_left, u_right, u_front, u_back, u_0, hx, hy,
hz, m = 0.0;
        double *data, *temp/*, *next*/, *buff, *dev_data, *dev_next, *dev_buff, *check;
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numproc);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Barrier(MPI_COMM_WORLD);
        if (id == 0) {
                std::cin >> dim1 >> dim2 >> dim3_3 >> x >> y >> z;
                std::cerr << dim1 << " " << dim2 << " " << dim3_3 << " " << x << " " << y << " "
<< z << " ";
                std::cin >> fi;
                std::cerr << fi << " ";
                std::cin >> eps;
```

```
		std::cerr << eps << " ";
		std::cin >> l_x >> l_y >> l_z;
		std::cerr << l_x << " " << l_y << " " << l_z << " ";
		std::cin >> u_down >> u_up >> u_left >> u_right >> u_front >> u_back >>
u_0;
		std::cerr << u_down << " " << u_up << " " << u_left << " " << u_right << " " <<
u_front << " " << u_back << " " << u_0 << " ";
	}
	int dev;
	CSC(cudaGetDeviceCount(&dev));
	CSC(cudaSetDevice(id % dev));
	MPI_Bcast(&dim1, 1, MPI_INT, 0, MPI_COMM_WORLD); // считанные переменные
пересылаем всем остальным процессам
	MPI_Bcast(&dim2, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&dim3_3, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&y, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&z, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&l_x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&l_y, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&l_z, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_down, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_up, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_left, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_right, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_front, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_back, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&u_0, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(fi, 128, MPI_CHAR, 0, MPI_COMM_WORLD);
	ib = _ibx(id);
	jb = _iby(id);
	kb = _ibz(id);
	hx = l_x / (double)(dim1 * x);
	hy = l_y / (double)(dim2 * y);
	hz = l_z / (double)(dim3_3 * z);
	data = (double*)malloc(sizeof(double) * (x + 2) * (y + 2) * (z + 2)); // n + 2 -- для того
чтобы организовать фиктивные ячейки
	// next = (double*)malloc(sizeof(double) * (x + 2) * (y + 2) * (z + 2));
	max = std::max(std::max(x ,y), z);
	buff = (double*)malloc(sizeof(double) * (max) * (max));
	double* check_mpi = (double*)malloc(sizeof(double) * dim1 * dim2 * dim3_3);
	int buffer_size;
	buffer_size = 6 * (sizeof(double) * (max) * (max) + MPI_BSEND_OVERHEAD);
	double *buffer = (double *)malloc(buffer_size);
	MPI_Buffer_attach(buffer, buffer_size);
	for (i = 0; i < x; ++i) { // инициализация блока
		for (j = 0; j < y; ++j) {
```

```
                        for (k = 0; k < z; ++k) {
                                data[_i(i, j, k)] = u_0;
                        }
                }
        }
        CSC(cudaMalloc(&dev_data, sizeof(double) * (x + 2) * (y + 2) * (z + 2)));
        CSC(cudaMalloc(&dev_next, sizeof(double) * (x + 2) * (y + 2) * (z + 2)));
        CSC(cudaMalloc(&dev_buff, sizeof(double) * (max) * (max)));
        CSC(cudaMemcpy(dev_data, data, sizeof(double) * (x + 2) * (y + 2) * (z + 2),
cudaMemcpyHostToDevice));
        for (;;) {
                CSC(cudaMalloc(&check, sizeof(double) * (x + 2) * (y + 2) * (z + 2)));
                // MPI_Barrier(MPI_COMM_WORLD); // синхронизация всех потоков
                if (ib + 1 < dim1) { // если наш поток не крайний
                        ib1_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_buff,
dev_data, true);
                        CSC(cudaGetLastError());
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Bsend(buff, y * z, MPI_DOUBLE, _ib(ib + 1, jb, kb), id,
MPI_COMM_WORLD); // и отправляем враво наши данные, процессу с номером ib + 1
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                }
                if (jb + 1 < dim2) { // отправна наверх наших данных
                        jb1_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_buff,
dev_data, true);
                        CSC(cudaGetLastError());
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Bsend(buff, x * z, MPI_DOUBLE, _ib(ib, jb + 1, kb), id,
MPI_COMM_WORLD); //
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                }
                if (kb + 1 < dim3_3) { // отправна наверх наших данных
                        kb1_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_buff,
dev_data, true);
                        CSC(cudaGetLastError());
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Bsend(buff, x * y, MPI_DOUBLE, _ib(ib, jb, kb + 1), id,
MPI_COMM_WORLD); //
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                }
                if (ib > 0) {
```

```
                        ib1_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_buff,
dev_data, false);
                        CSC(cudaGetLastError());
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Bsend(buff, y * z, MPI_DOUBLE, _ib(ib - 1, jb, kb), id,
MPI_COMM_WORLD);
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                }
                if (jb > 0) {
                        jb1_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_buff,
dev_data, false);
                        CSC(cudaGetLastError());
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Bsend(buff, x * z, MPI_DOUBLE, _ib(ib, jb - 1, kb), id,
MPI_COMM_WORLD); //
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                }
                if (kb > 0) {
                        kb1_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_buff,
dev_data, false);
                        CSC(cudaGetLastError());
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Bsend(buff, x * y, MPI_DOUBLE, _ib(ib, jb, kb - 1), id,
MPI_COMM_WORLD); //
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                }
                // CSC(cudaMemcpy(data, dev_data, sizeof(double) * (x + 2) * (y + 2) * (z +
2), cudaMemcpyDeviceToHost));
                // MPI_Barrier(MPI_COMM_WORLD);
                // поскольку Bsend не подвисающая операция мы сразу можем
принимать данные

                if (ib > 0) { // если мы можем принимать данные
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Recv(buff, y * z, MPI_DOUBLE, _ib(ib - 1, jb, kb), _ib(ib - 1, jb,
kb), MPI_COMM_WORLD, &status);// то мы принимаем данные
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                        ib2_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
dev_buff, false);
                        CSC(cudaGetLastError());
```

```
                } else {
                        ib3_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data, u_left,
false);
                        CSC(cudaGetLastError());
                }
                if (jb > 0) { // если мы можем принимать данные
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Recv(buff, x * z, MPI_DOUBLE, _ib(ib, jb - 1, kb), _ib(ib, jb - 1,
kb), MPI_COMM_WORLD, &status);// то мы принимаем данные
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                        jb2_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
dev_buff, false);
                        CSC(cudaGetLastError());
                } else {
                        jb3_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
u_front, false);
                        CSC(cudaGetLastError());
                }
                if (kb > 0) { // если мы можем принимать данные
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Recv(buff, x * y, MPI_DOUBLE, _ib(ib, jb, kb - 1), _ib(ib, jb, kb -
1), MPI_COMM_WORLD, &status);// то мы принимаем данные
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                        kb2_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
dev_buff, false);
                        CSC(cudaGetLastError());
                } else {
                        kb3_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
u_down, false);
                        CSC(cudaGetLastError());
                }
                if (ib + 1 < dim1) { // если мы можем принимать данные
                        CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                        MPI_Recv(buff, y * z, MPI_DOUBLE, _ib(ib + 1, jb, kb), _ib(ib + 1, jb,
kb), MPI_COMM_WORLD, &status);// то мы принимаем данные
                        CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                        ib2_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
dev_buff, true);
                        CSC(cudaGetLastError());
                } else {
                        ib3_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
u_right, true);
```

```
                CSC(cudaGetLastError());
            }
            if (jb + 1 < dim2) { // если мы можем принимать данные
                CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                MPI_Recv(buff, x * z, MPI_DOUBLE, _ib(ib, jb + 1, kb), _ib(ib, jb + 1,
kb), MPI_COMM_WORLD, &status);// то мы принимаем данные
                CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                jb2_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
dev_buff, true);
                CSC(cudaGetLastError());
            } else {
                jb3_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
u_back, true);
                CSC(cudaGetLastError());
            }
            if (kb + 1 < dim3_3) { // если мы можем принимать данные
                CSC(cudaMemcpy(buff, dev_buff, sizeof(double) * max * max,
cudaMemcpyDeviceToHost));
                MPI_Recv(buff, x * y, MPI_DOUBLE, _ib(ib, jb, kb + 1), _ib(ib, jb, kb +
1), MPI_COMM_WORLD, &status);// то мы принимаем данные
                CSC(cudaMemcpy(dev_buff, buff, sizeof(double) * max * max,
cudaMemcpyHostToDevice));
                kb2_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data,
dev_buff, true);
                CSC(cudaGetLastError());
            } else {
                kb3_kernel<<<dim3(32, 32), dim3(32, 32)>>>(x, y, z, dev_data, u_up,
true);
                CSC(cudaGetLastError());
            }
            //дпльше оргнизуем основной вычислительный цикл
            // MPI_Barrier(MPI_COMM_WORLD); // выполняем синзронизацию всех
процессов
            formula<<<dim3(8, 8, 8), dim3(32, 4, 4)>>>(x, y, z, dev_next, dev_data, hx,
hy, hz, check);
            CSC(cudaGetLastError());
            epsil<<<dim3(8, 8, 8), dim3(32, 4, 4)>>>(dev_next, dev_data, check, x, y, z);
            CSC(cudaGetLastError());
            thrust::device_ptr<double> p_arr = thrust::device_pointer_cast(check);
            thrust::device_ptr<double> res = thrust::max_element(p_arr, p_arr + (x + 2) *
(y + 2) * (z + 2));
            m = *res;
            temp = dev_next;
            dev_next = dev_data;
            dev_data = temp;
            // MPI_Barrier(MPI_COMM_WORLD);
```

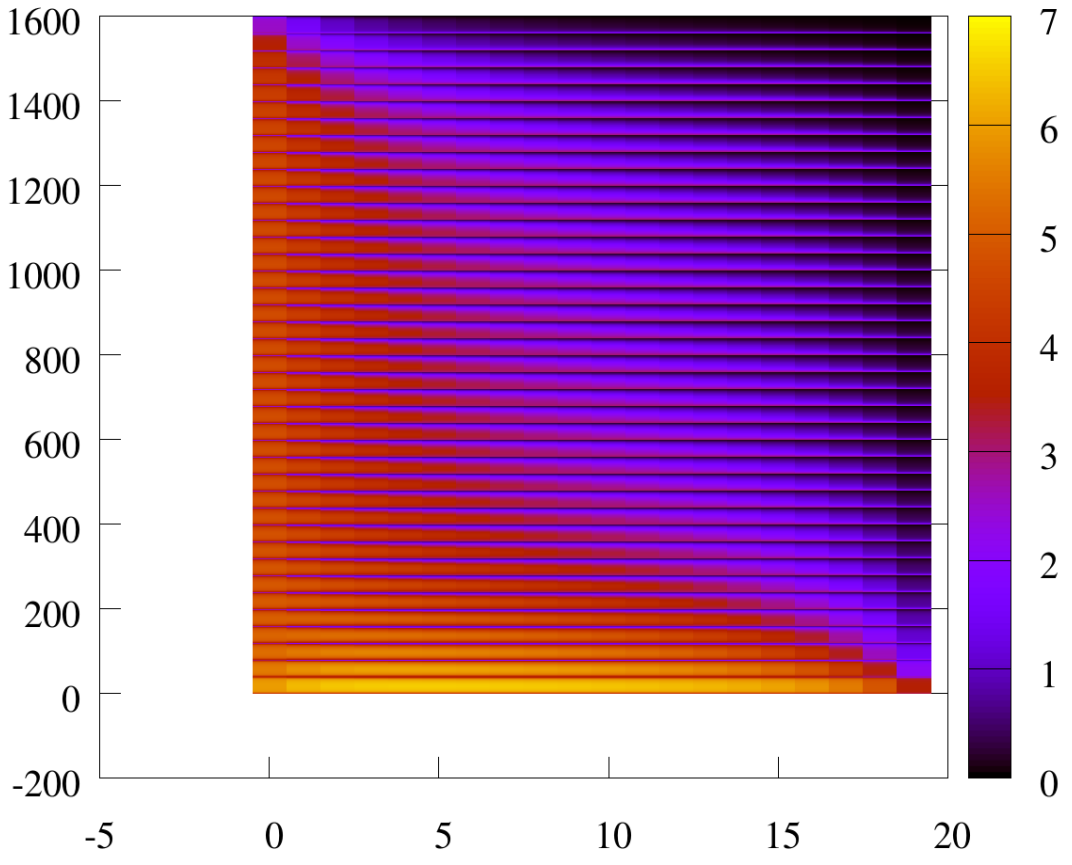
```
                MPI_Allgather(&m, 1, MPI_DOUBLE, check_mpi, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
                m = 0.0;
                for (i = 0; i < dim1 * dim2 * dim3_3; ++i) {
                        if (check_mpi[i] > m) {
                                m = check_mpi[i];
                        }
                }
                if (m < eps) {
                        break;
                }
                CSC(cudaFree(check));
        }
        CSC(cudaMemcpy(data, dev_data, sizeof(double) * (x + 2) * (y + 2) * (z + 2),
cudaMemcpyDeviceToHost));
        // MPI_Barrier(MPI_COMM_WORLD);
        int n_size = 30;
        char* bf = (char*)malloc(sizeof(char) * x * y * z * n_size);
        memset(bf, ' ', sizeof(char) * x * y * z * n_size);
        for (k = 0; k < z; k++) {
                for (j = 0; j < y; j++) {
                        for (i = 0; i < x; i++) {
                                sprintf(bf + ((k * x * y) + j * x + i) * n_size, "%.6e", data[_i(i, j,
k)]);
                        }
                        if (ib + 1 == dim1) {
                                bf[(k * x * y + j * x + i) * n_size - 1] = '\n';
                        }
                }
        }
        for (i = 0; i < x * y * z * n_size; i++)  {
                if (bf[i] == '\0') {
                        bf[i] = ' ';
                }
        }
        MPI_File fl;
        MPI_Datatype filetype;
        MPI_Type_contiguous(n_size, MPI_CHAR, &filetype);
        MPI_Type_commit(&filetype);
        MPI_Datatype sub, big;
        int sub_bigsizes[3] = {x, y, z};
        int sub_subsizes[3] = {x, y, z};
        int sub_starts[3] = {0, 0, 0};
        int big_bigsizes[3] = {x * dim1, y * dim2, z * dim3_3};
        int big_subsizes[3] = {x, y, z};
        int big_starts[3] = {ib * x, jb * y, kb * z};
        MPI_Type_create_subarray(3, sub_bigsizes, sub_subsizes, sub_starts,
MPI_ORDER_FORTRAN, filetype, &sub);
```

```
        MPI_Type_create_subarray(3, big_bigsizes, big_subsizes, big_starts,
MPI_ORDER_FORTRAN, filetype, &big);
        MPI_Type_commit(&sub);
        MPI_Type_commit(&big);
        MPI_File_delete(fi, MPI_INFO_NULL);
        MPI_File_open(MPI_COMM_WORLD, fi, MPI_MODE_CREATE |
MPI_MODE_WRONLY, MPI_INFO_NULL, &fl);
        MPI_File_set_view(fl, 0, MPI_CHAR, big, "native", MPI_INFO_NULL);
        MPI_File_write_all(fl, bf, 1, sub, MPI_STATUS_IGNORE);
        MPI_File_close(&fl);
        MPI_Finalize();
        return 0;
}
```

Результаты

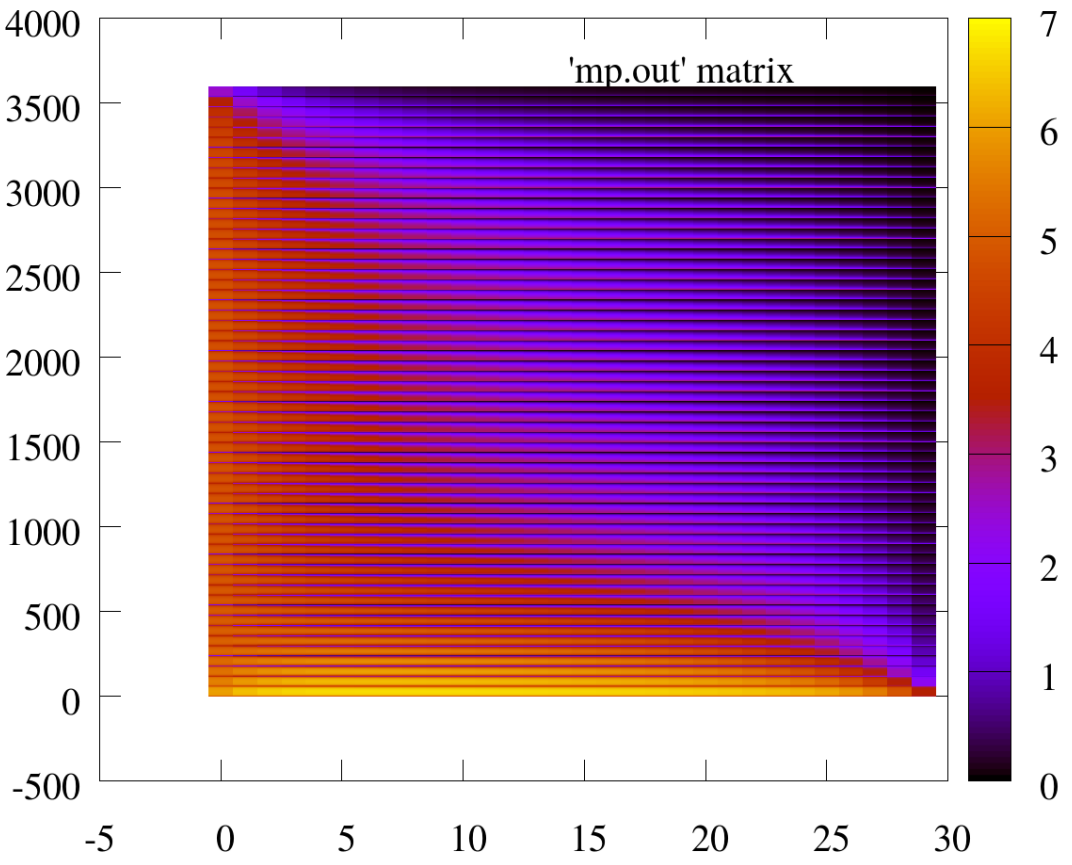
Размер: 20*20*20

Время работы	dim1	dim2	dim3
3.47894sec	1	1	1
145.194sec	2	2	2
341.562sec	2	3	2



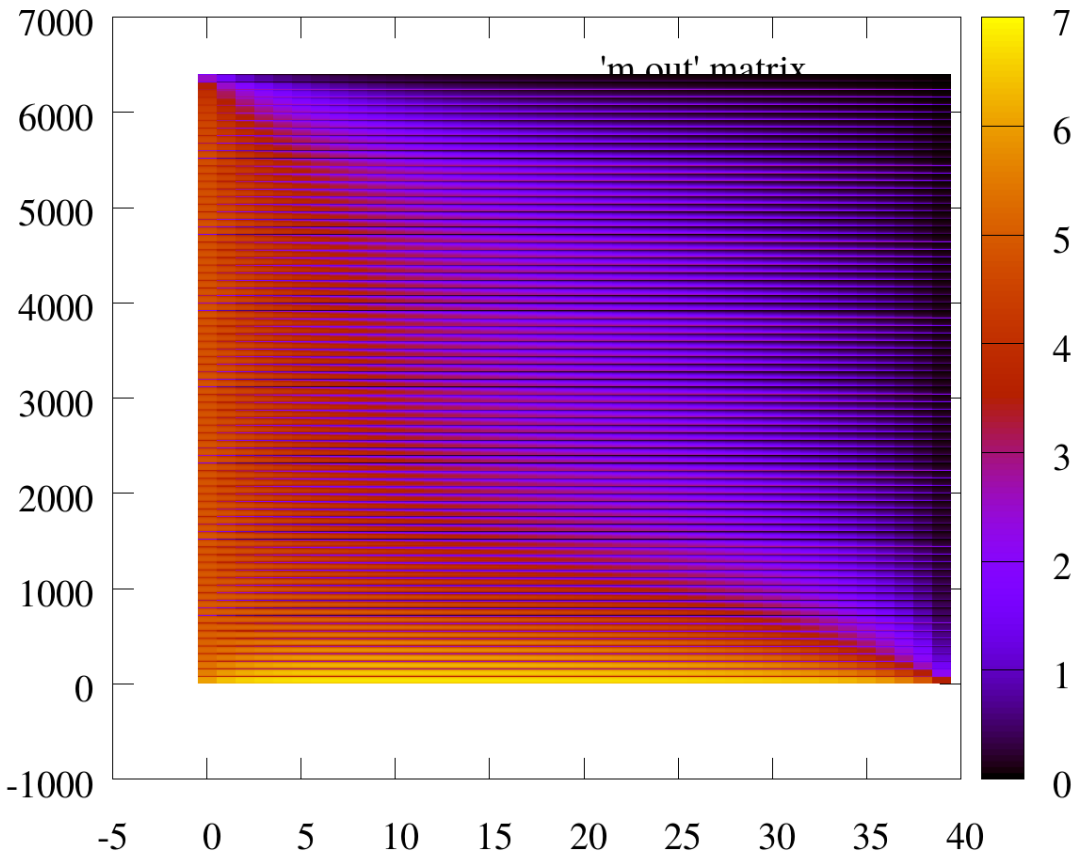
Размер: 30*30*30

Время работы	dim1	dim2	dim3
7.64189sec	1	1	1
320.726sec	2	2	2
715.938sec	2	3	2



Размер: 40*40*40

Время работы	dim1	dim2	dim3
16.2194sec	1	1	1
647.829sec	2	2	2
1917.09sec	2	3	2



Выводы

Выполнив данную лабораторную работу я применил технологию MPI в сочетании с распараллеленными вычислениями на CUDA. Также был реализован свой собственный тип данных позволяющий записывать в файл одновременно всеми процессами. Сама по себе лабораторная была не сложной, поскольку в ней по большей части применялись знания полученные в при выполнении предыдущих лабораторных. Код был написан очень быстро, но была одна ошибка с которой пришлось побороться весьма продолжительное время.