ЗМОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторная работа №8 (3)
по курсу «Параллельная Обработка Данных»

Технология MPI и технология CUDA. MPI-IO

Выполнил: Иларионов Д.А.
Группа: М8О-408Б-17
Преподаватели: Крашенинников К.Г.,
Морозов А.Ю.

Москва, 2020

Условие

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 3. MPI_Type_hindexed

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

Долго мучался и с этой лабораторной. Дело в том, что в ней очень много чего нового, и это все нужно добавить в основную программу из ЛР 7. Нормальный вывод в файл через производные типы данных. С этим были проблемы. Очень часто файлы оказывались пустыми, а у меня сильно бомбило. Позже, понял причину этого. Оказывается, у меня другим процессам просто не передается имя файла. Я это исправил, преобразовав string в const char* , и потом передал его обычным isend/irecv

всем процессам. Файлы стали нормально создаваться и записываться. Затем, долго работал над форматированием, чтобы данные выводились красиво, были читабельны, а не все в одну строчку. Самая долгая и трудная задача – переписать бОльшую часть программы на CUDA. Создал кучу процедур, которые выполняют основные расчеты на графическом процессоре. Не спал целые сутки, выпил за это время 4 энергетика и чашку кофе, работал над данной лабораторной. Но была одна ошибка, с которой мне было тяжело справиться. Попросил преподавателя о помощи, и мы искали эту ошибку больше часа. Тут я понял. Чем труднее найти ошибку в программе – тем она глупее и нелепее. Да я же просто передавал данные, а в размере еще зачем-то на sizeof(double) умножал, а ведь это не надо было вовсе, тип указан справа. Походу, слишком помешался на этом производном типе hindexed, что везде теперь хочется умножать на размер типа. Ну, а когда программа прошла, я был рад! Не так, как после прохождения ЛР 7, но все же… Ну а о самой программе – мы обрабатываем данные на графическом процессоре (расчеты по самой формуле и инициаилизация массивов). Да и сами массивы хранятся в памяти видеокарты. Копируем только стенки. Только вот это постоянное копирование много времени с собой забирает. В конце мы через производный тип данных заполняем соответствующие ячейки (если так можно назвать) файла значениями.

Описание программы

Файл main.cpp

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda.h>
#include <thrust/extrema.h>
#include <thrust/device_vector.h>
#include <iostream>
#include <mpi.h>
#include <string>
#include <algorithm>

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)

int p1, p2, p3;
int g1, g2, g3;

// Index inside the block
#define _i(i, j, k) ((k + 1) * ((g2 + 2) * (g1 + 2)) + (j + 1) * (g1 + 2) + i + 1)
#define _ix(id) (((id) % (g1 + 2)) - 1)
#define _iy(id) ((((id) % ((g1 + 2) * (g2 + 2))) / (g1 + 2)) - 1)
#define _iz(id) (((id) / ((g1 + 2)*(g2 + 2))) - 1)

// Index by processes
#define _ib(i, j, k) ((k) * (p1 * p2) + (j) * p1 + (i))
#define _ibx(id) ((id) % p1)
#define _iby(id) (((id) % (p1 * p2)) / p1)
```

```
40. #define _ibz(id) ((id) / (p1 * p2))
41.
42.
43. #define printf(...) fprintf(File, __VA_ARGS__)
44.
45.
46.
47. __global__ void ArrInits(double* fastData, double* fastNext, int g1, int g2, int g3, double U0) {
48.     int tX = blockIdx.x * blockDim.x + threadIdx.x;
49.     int tY = blockIdx.y * blockDim.y + threadIdx.y;
50.     int tZ = blockIdx.z * blockDim.z + threadIdx.z;
51.     int offsetX = gridDim.x * blockDim.x;
52.     int offsetY = gridDim.y * blockDim.y;
53.     int offsetZ = gridDim.z * blockDim.z;
54.
55.     for (int a = tX - 1; a < g1 + 1; a += offsetX) {
56.         for (int b = tY - 1; b < g2 + 1; b += offsetY) {
57.             for (int c = tZ - 1; c < g3 + 1; c += offsetZ) {
58.                 fastData[_i(a, b, c)] = U0;
59.                 fastNext[_i(a, b, c)] = U0;
60.             }
61.         }
62.     }
63.
64. }
65.
66.
67. //1 - left , 2 - right, 3 - up, 4 - down, 5 - front, 6 - back
68. #define LEFT 1
69. #define RIGHT 2
70. #define UP 3
71. #define DOWN 4
72. #define FRONT 5
73. #define BACK 6
74.
75.
76. __global__ void cuda_get_side(double* fastData, double* fastSide, int side, int g1, int g2, int g3)
    {
77.     int tX = blockIdx.x * blockDim.x + threadIdx.x;
78.     int tY = blockIdx.y * blockDim.y + threadIdx.y;
79.     int offsetX = gridDim.x * blockDim.x;
80.     int offsetY = gridDim.y * blockDim.y;
81.     if (side == LEFT) {
82.         for (int k = tX; k < g3; k += offsetX) {
83.             for (int j = tY; j < g2; j += offsetY) {
84.                 fastSide[j + k * g2] = fastData[_i(0, j, k)];
85.             }
86.         }
87.     }
88.     else if (side == RIGHT) {
89.         for (int k = tX; k < g3; k += offsetX) {
90.             for (int j = tY; j < g2; j += offsetY) {
91.                 fastSide[j + k * g2] = fastData[_i(g1 - 1, j, k)];
92.             }
93.         }
94.     }
95.     else if (side == UP) {
96.         for (int k = tX; k < g3; k += offsetX) {
97.             for (int i = tY; i < g1; i += offsetY) {
98.                 fastSide[i + k * g1] = fastData[_i(i, 0, k)];
99.             }
100.                }
101.            }
102.            else if (side == DOWN) {
103.                for (int k = tX; k < g3; k += offsetX) {
104.                    for (int i = tY; i < g1; i += offsetY) {
105.                        fastSide[i + k * g1] = fastData[_i(i, g2 - 1, k)];
106.                    }
107.                }
108.            }
109.            else if (side == FRONT) {
110.                for (int j = tX; j < g2; j += offsetX) {
111.                    for (int i = tY; i < g1; i += offsetY) {
```

```
                        fastSide[i + j * g1] = fastData[_i(i, j, 0)];
                    }
                }
            }
            else {
                for (int j = tX; j < g2; j += offsetX) {
                    for (int i = tY; i < g1; i += offsetY) {
                        fastSide[i + j * g1] = fastData[_i(i, j, g3 - 1)];
                    }
                }
            }
        }


        __global__ void cuda_set_side(double* fastData, double* fastSide, int side, int g1, int g2, int g3) {
            int tX = blockIdx.x * blockDim.x + threadIdx.x;
            int tY = blockIdx.y * blockDim.y + threadIdx.y;
            int offsetX = gridDim.x * blockDim.x;
            int offsetY = gridDim.y * blockDim.y;
            if (side == LEFT) {
                for (int k = tX; k < g3; k += offsetX) {
                    for (int j = tY; j < g2; j += offsetY) {
                        fastData[_i(-1, j, k)] = fastSide[j + k * g2];
                    }
                }
            }
            else if (side == RIGHT) {
                for (int k = tX; k < g3; k += offsetX) {
                    for (int j = tY; j < g2; j += offsetY) {
                        fastData[_i(g1, j, k)] = fastSide[j + k * g2];
                    }
                }
            }
            else if (side == UP) {
                for (int k = tX; k < g3; k += offsetX) {
                    for (int i = tY; i < g1; i += offsetY) {
                        fastData[_i(i, -1, k)] = fastSide[i + k * g1];
                    }
                }
            }
            else if (side == DOWN) {
                for (int k = tX; k < g3; k += offsetX) {
                    for (int i = tY; i < g1; i += offsetY) {
                        fastData[_i(i, g2, k)] = fastSide[i + k * g1];
                    }
                }
            }
            else if (side == FRONT) {
                for (int j = tX; j < g2; j += offsetX) {
                    for (int i = tY; i < g1; i += offsetY) {
                        fastData[_i(i, j, -1)] = fastSide[i + j * g1];
                    }
                }
            }
            else {
                for (int j = tX; j < g2; j += offsetX) {
                    for (int i = tY; i < g1; i += offsetY) {
                        fastData[_i(i, j, g3)] = fastSide[i + j * g1];
                    }
                }
            }
        }


        __global__ void cuda_side_edge_values(double* fastData, int side, int g1, int g2, int g3, double value) {
            int tX = blockIdx.x * blockDim.x + threadIdx.x;
            int tY = blockIdx.y * blockDim.y + threadIdx.y;
            int offsetX = gridDim.x * blockDim.x;
            int offsetY = gridDim.y * blockDim.y;
            if (side == LEFT) {
                for (int k = tX; k < g3; k += offsetX) {
```

```
183.                     for (int j = tY; j < g2; j += offsetY) {
184.                         fastData[_i(-1, j, k)] = value;
185.                     }
186.                 }
187.             }
188.             else if (side == RIGHT) {
189.                 for (int k = tX; k < g3; k += offsetX) {
190.                     for (int j = tY; j < g2; j += offsetY) {
191.                         fastData[_i(g1, j, k)] = value;
192.                     }
193.                 }
194.             }
195.             else if (side == UP) {
196.                 for (int k = tX; k < g3; k += offsetX) {
197.                     for (int i = tY; i < g1; i += offsetY) {
198.                         fastData[_i(i, -1, k)] = value;
199.                     }
200.                 }
201.             }
202.             else if (side == DOWN) {
203.                 for (int k = tX; k < g3; k += offsetX) {
204.                     for (int i = tY; i < g1; i += offsetY) {
205.                         fastData[_i(i, g2, k)] = value;
206.                     }
207.                 }
208.             }
209.             else if (side == FRONT) {
210.                 for (int j = tX; j < g2; j += offsetX) {
211.                     for (int i = tY; i < g1; i += offsetY) {
212.                         fastData[_i(i, j, -1)] = value;
213.                     }
214.                 }
215.             }
216.             else {
217.                 for (int j = tX; j < g2; j += offsetX) {
218.                     for (int i = tY; i < g1; i += offsetY) {
219.                         fastData[_i(i, j, g3)] = value;
220.                     }
221.                 }
222.             }
223.         }
224.
225.
226.
227.         __global__ void cuda_main_function(double* fastData, double* fastNext, int g1,
228.          int g2, int g3, double hx, double hy, double hz) {
229.             int tX = blockIdx.x * blockDim.x + threadIdx.x;
230.             int tY = blockIdx.y * blockDim.y + threadIdx.y;
231.             int tZ = blockIdx.z * blockDim.z + threadIdx.z;
232.             int offsetX = gridDim.x * blockDim.x;
233.             int offsetY = gridDim.y * blockDim.y;
234.             int offsetZ = gridDim.z * blockDim.z;
235.
236.             for (int k = tZ; k < g3; k += offsetZ) {
237.                 for (int j = tY; j < g2; j += offsetY) {
238.                     for (int i = tX; i < g1; i += offsetX) {
239.                         fastNext[_i(i, j, k)] = 0.5 * ((fastData[_i(i + 1, j, k)] +
240.                             fastData[_i(i - 1, j, k)]) / (hx * hx) + (fastData[_i(i, j + 1, k)]
241.                             + fastData[_i(i, j - 1, k)]) / (hy * hy) + (fastData[_i(i, j, k + 1)]
242.                             + fastData[_i(i, j, k - 1)]) / (hz * hz)) /
243.                             (1.0 / (hx * hx) + 1.0 / (hy * hy) + 1.0 / (hz * hz));
244.                     }
245.                 }
246.             }
247.         }
248.
249.         __global__ void cuda_error_function(double* fastData, double* fastNext, int g1, int g2,
    int g3) {
250.             int tX = blockIdx.x * blockDim.x + threadIdx.x;
251.             int tY = blockIdx.y * blockDim.y + threadIdx.y;
252.             int tZ = blockIdx.z * blockDim.z + threadIdx.z;
253.             int offsetX = gridDim.x * blockDim.x;
254.             int offsetY = gridDim.y * blockDim.y;
```

```
255.             int offsetZ = gridDim.z * blockDim.z;
256.
257.             for (int k = tZ - 1; k < g3 + 1; k += offsetZ) {
258.                 for (int j = tY - 1; j < g2 + 1; j += offsetY) {
259.                     for (int i = tX - 1; i < g1 + 1; i += offsetX) {
260.                         bool lolkekval = (i != -1 && j != -1 && k != -1) *
261.                             (i != g1 && j != g2 && k != g3);
262.                         fastData[_i(i, j, k)] = fabs(fastNext[_i(i, j, k)] - fastData[_i(i, j,
    k)]) * lolkekval;
263.                     }
264.                 }
265.             }
266.         }
267.
268.
269.         int main(int argc, char** argv) {
270.             std::ios::sync_with_stdio(false);
271.             string outFile;
272.             int fileNameL = 16;
273.
274.             int deviceCount;
275.             cudaGetDeviceCount(&deviceCount);
276.
277.
278.             int id;
279.             int ib, jb, kb;
280.             int i, j, k, iter;
281.             int numproc, proc_name_len;
282.             int L = 16;
283.
284.             char proc_name[MPI_MAX_PROCESSOR_NAME];
285.
286.             double eps;
287.             double lx, ly, lz;
288.             double hx, hy, hz;
289.             double Udown, Uup, Uleft, Uright, Ufront, Uback;
290.             double U0;
291.             double * temp;
292.
293.             MPI_Status status;
294.
295.             MPI_Init(&argc, &argv);
296.             MPI_Comm_rank(MPI_COMM_WORLD, &id);
297.             MPI_Comm_size(MPI_COMM_WORLD, &numproc);
298.             MPI_Get_processor_name(proc_name, &proc_name_len);
299.
300.
301.             cudaSetDevice(id % deviceCount);
302.
303.             if (id == 0) {
304.                 cerr << "Found " << deviceCount << " devices\n";
305.             }
306.
307.
308.             MPI_Barrier(MPI_COMM_WORLD);
309.
310.             string nullString = "";
311.             const char* outFileC = nullString.c_str();
312.
313.
314.
315.             //input data for 0 process
316.             if (id == 0) {
317.
318.                 cin >> p1 >> p2 >> p3;
319.                 cin >> g1 >> g2 >> g3;
320.                 cin >> outFile;
321.                 cin >> eps;
322.                 cin >> lx >> ly >> lz;
323.                 cin >> Ufront >> Uback >> Uleft >> Uright >> Uup >> Udown;
324.                 cin >> U0;
325.
326.                 outFileC = outFile.c_str();
```

```
327.                 fileNameL = strlen(outFileC) + 1;
328.             }
329.
330.
331.             MPI_Barrier(MPI_COMM_WORLD);
332.
333.             //send data to all processes
334.             MPI_Bcast(&p1, 1, MPI_INT, 0, MPI_COMM_WORLD);
335.             MPI_Bcast(&p2, 1, MPI_INT, 0, MPI_COMM_WORLD);
336.             MPI_Bcast(&p3, 1, MPI_INT, 0, MPI_COMM_WORLD);
337.
338.             MPI_Bcast(&g1, 1, MPI_INT, 0, MPI_COMM_WORLD);
339.             MPI_Bcast(&g2, 1, MPI_INT, 0, MPI_COMM_WORLD);
340.             MPI_Bcast(&g3, 1, MPI_INT, 0, MPI_COMM_WORLD);
341.
342.             MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
343.
344.             MPI_Bcast(&lx, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
345.             MPI_Bcast(&ly, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
346.             MPI_Bcast(&lz, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
347.
348.             MPI_Bcast(&Udown, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
349.             MPI_Bcast(&Uup, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
350.             MPI_Bcast(&Uleft, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
351.             MPI_Bcast(&Uright, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
352.             MPI_Bcast(&Ufront, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
353.             MPI_Bcast(&Uback, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
354.             MPI_Bcast(&U0, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
355.             MPI_Bcast(&fileNameL, 1, MPI_INT, 0, MPI_COMM_WORLD);
356.
357.
358.
359.
360.
361.             if (p1 * p2 * p3 != numproc) {
362.                 MPI_Finalize();
363.                 if (id == 0) {
364.                     cout << "ERROR: proc.grid != processes\n";
365.                 }
366.
367.                 return -1;
368.             }
369.
370.
371.
372.             //block id by coordinates
373.             ib = _ibx(id);
374.             jb = _iby(id);
375.             kb = _ibz(id);
376.
377.             iter = 0;
378.
379.             //find hs
380.             hx = lx / ((double)p1 * (double)g1);
381.             hy = ly / ((double)p2 * (double)g2);
382.             hz = lz / ((double)p3 * (double)g3);
383.
384.             char* buff = (char*)malloc(sizeof(char) * (g1 * g2 * g3 * p1 * p2 * p3 * L));
385.
386.
387.             double* fastData, * fastNext;
388.
389.             double* fastLeftIn, * fastRightIn, * fastUPIn, * fastDownIn, * fastFrontIn, *
    fastBackIn;
390.             double* fastLeftOut, * fastRightOut, * fastUPOut, * fastDownOut, * fastFrontOut, *
    fastBackOut;
391.
392.             double* LeftIn, * LeftOut, * RightIn, * RightOut;
393.             double* UPIn, * UPOut, * DownIn, * DownOut;
394.             double* FrontIn, * FrontOut, * BackIn, * BackOut;
395.
396.             LeftIn = (double*)malloc(sizeof(double) * g2 * g3); LeftOut =
    (double*)malloc(sizeof(double) * g2 * g3);
```

```
397.              RightIn = (double*)malloc(sizeof(double) * g2 * g3); RightOut =
    (double*)malloc(sizeof(double) * g2 * g3);
398.
399.              UPIn = (double*)malloc(sizeof(double) * g1 * g3); UPOut =
    (double*)malloc(sizeof(double) * g1 * g3);
400.              DownIn = (double*)malloc(sizeof(double) * g1 * g3); DownOut =
    (double*)malloc(sizeof(double) * g1 * g3);
401.
402.              FrontIn = (double*)malloc(sizeof(double) * g1 * g2); FrontOut =
    (double*)malloc(sizeof(double) * g1 * g2);
403.              BackIn = (double*)malloc(sizeof(double) * g1 * g2); BackOut =
    (double*)malloc(sizeof(double) * g1 * g2);
404.
405.
406.              //Init GPU Memory
407.              CSC(cudaMalloc((void**)&fastData, sizeof(double)* (g1 + 2)* (g2 + 2)* (g3 + 2)));
408.              CSC(cudaMalloc((void**)&fastNext, sizeof(double)* (g1 + 2)* (g2 + 2)* (g3 + 2)));
409.
410.              CSC(cudaMalloc((void**)& fastLeftIn, sizeof(double)* g2 * g3));
411.              CSC(cudaMalloc((void**)& fastRightIn, sizeof(double)* g2* g3));
412.
413.              CSC(cudaMalloc((void**)& fastUPIn, sizeof(double)* g1* g3));
414.              CSC(cudaMalloc((void**)& fastDownIn, sizeof(double)* g1* g3));
415.
416.              CSC(cudaMalloc((void**)& fastFrontIn, sizeof(double)* g1* g2));
417.              CSC(cudaMalloc((void**)& fastBackIn, sizeof(double)* g1* g2));
418.
419.              CSC(cudaMalloc((void**)& fastLeftOut, sizeof(double)* g2* g3));
420.              CSC(cudaMalloc((void**)& fastRightOut, sizeof(double)* g2* g3));
421.
422.              CSC(cudaMalloc((void**)& fastUPOut, sizeof(double)* g1* g3));
423.              CSC(cudaMalloc((void**)& fastDownOut, sizeof(double)* g1* g3));
424.
425.              CSC(cudaMalloc((void**)& fastFrontOut, sizeof(double)* g1* g2));
426.              CSC(cudaMalloc((void**)& fastBackOut, sizeof(double)* g1* g2));
427.
428.              int SIZE = 4;
429.
430.              dim3 gridSz(SIZE, SIZE, SIZE);
431.              dim3 blockSz(SIZE, SIZE, SIZE);
432.
433.              ArrInits << < gridSz, blockSz >> > (fastData, fastNext, g1, g2, g3, U0);
434.
435.              //make buffer
436.
437.              int buffer_size;
438.
439.              MPI_Pack_size((g1 + 2) * (g2 + 2) * (g3 + 2), MPI_DOUBLE, MPI_COMM_WORLD,
    &buffer_size);
440.
441.              buffer_size = 2 * (buffer_size + MPI_BSEND_OVERHEAD); //6 edges
442.
443.
444.              double* buffer = (double*)malloc(buffer_size);
445.
446.              MPI_Buffer_attach(buffer, buffer_size);
447.
448.              //block init
449.
450.              //requests
451.
452.              MPI_Request send_request1_1, recv_request1_1;
453.              MPI_Request send_request2_1, recv_request2_1;
454.              MPI_Request send_request3_1, recv_request3_1;
455.
456.              MPI_Request send_request1_2, recv_request1_2;
457.              MPI_Request send_request2_2, recv_request2_2;
458.              MPI_Request send_request3_2, recv_request3_2;
459.
460.
461.              double* errors;
462.              errors = (double*)malloc(numproc * sizeof(double));
463.
```

```
464.             //string debug_name = "process_debug" + to_string(id) + ".txt";
465.
466.             int size2 = 8;
467.
468.             dim3 blocks(size2, size2);
469.             dim3 threads(size2, size2);
470.
471.             double maxErr = 0;
472.             do {
473.                 //send and get data
474.                 MPI_Barrier(MPI_COMM_WORLD);
475.
476.
477.                 if (ib > 0) { //only left side
478.                     cuda_get_side << < blocks, threads >> > (fastData, fastLeftIn, LEFT, g1, g2,
    g3);
479.                     CSC(cudaGetLastError());
480.                     CSC(cudaMemcpy(LeftIn, fastLeftIn, sizeof(double)* g2* g3,
    cudaMemcpyDeviceToHost));
481.                     MPI_Isend(LeftIn, g2* g3, MPI_DOUBLE, _ib(ib - 1, jb, kb), 0, MPI_COMM_WORLD,
    &send_request1_1);
482.                 }
483.
484.                 if (jb > 0) { //only up side
485.                     cuda_get_side << < blocks, threads >> > (fastData, fastUPIn, UP, g1, g2, g3);
486.                     CSC(cudaGetLastError());
487.                     CSC(cudaMemcpy(UPIn, fastUPIn, sizeof(double)* g1* g3,
    cudaMemcpyDeviceToHost));
488.                     MPI_Isend(UPIn, g1* g3, MPI_DOUBLE, _ib(ib, jb - 1, kb), 0, MPI_COMM_WORLD,
    &send_request2_2);
489.                 }
490.
491.                 if (kb > 0) { //only front side
492.                     cuda_get_side << < blocks, threads >> > (fastData, fastFrontIn, FRONT, g1, g2,
    g3);
493.                     CSC(cudaGetLastError());
494.                     CSC(cudaMemcpy(FrontIn, fastFrontIn, sizeof(double)* g1* g2,
    cudaMemcpyDeviceToHost));
495.
496.                     MPI_Isend(FrontIn, g1* g2, MPI_DOUBLE, _ib(ib, jb, kb - 1), 0, MPI_COMM_WORLD,
    &send_request3_2);
497.                 }
498.
499.
500.                 //wait for data
501.
502.                 if (ib > 0) { //only left side
503.                     MPI_Wait(&send_request1_1, &status);
504.                 }
505.
506.                 if (jb > 0) { //only up side
507.                     MPI_Wait(&send_request2_2, &status);
508.                 }
509.
510.                 if (kb > 0) { //only front side
511.                     MPI_Wait(&send_request3_2, &status);
512.                 }
513.
514.
515.
516.                 //set new data
517.
518.                 if (ib + 1 < p1) { //get right side
519.                     MPI_Irecv(RightOut, g2 * g3, MPI_DOUBLE, _ib(ib + 1, jb, kb), 0,
    MPI_COMM_WORLD, &recv_request1_2);
520.                     MPI_Wait(&recv_request1_2, &status);
521.
522.                     CSC(cudaMemcpy(fastRightOut, RightOut, sizeof(double)* g2* g3,
    cudaMemcpyHostToDevice));
523.                     cuda_set_side << < blocks, threads >> > (fastData, fastRightOut, RIGHT, g1,
    g2, g3);
524.                     CSC(cudaGetLastError());
525.                 }
```

```
526.                 else {
527.                     cuda_side_edge_values << < blocks, threads >> > (fastData, RIGHT, g1, g2, g3,
    Uright);
528.                     CSC(cudaGetLastError());
529.                 }
530.
531.                 if (jb + 1 < p2) { //get down side
532.                     MPI_Irecv(DownOut, g1* g3, MPI_DOUBLE, _ib(ib, jb + 1, kb), 0, MPI_COMM_WORLD,
    &recv_request2_1);
533.                     MPI_Wait(&recv_request2_1, &status);
534.
535.                     CSC(cudaMemcpy(fastDownOut, DownOut, sizeof(double)* g1* g3,
    cudaMemcpyHostToDevice));
536.                     cuda_set_side << < blocks, threads >> > (fastData, fastDownOut, DOWN, g1, g2,
    g3);
537.                     CSC(cudaGetLastError());
538.                 }
539.                 else {
540.                     cuda_side_edge_values << < blocks, threads >> > (fastData, DOWN, g1, g2, g3,
    Udown);
541.                     CSC(cudaGetLastError());
542.                 }
543.
544.
545.                 if (kb + 1 < p3) { //get back side
546.                     MPI_Irecv(BackOut, g1* g2, MPI_DOUBLE, _ib(ib, jb, kb + 1), 0, MPI_COMM_WORLD,
    &recv_request3_1);
547.                     MPI_Wait(&recv_request3_1, &status);
548.
549.                     CSC(cudaMemcpy(fastBackOut, BackOut, sizeof(double)* g1* g2,
    cudaMemcpyHostToDevice));
550.                     cuda_set_side << < blocks, threads >> > (fastData, fastBackOut, BACK, g1, g2,
    g3);
551.                     CSC(cudaGetLastError());
552.                 }
553.                 else {
554.                     cuda_side_edge_values << < blocks, threads >> > (fastData, BACK, g1, g2, g3,
    Uback);
555.                     CSC(cudaGetLastError());
556.                 }
557.
558.
559.
560.
561.
562.                 if (ib + 1 < p1) { //only right side
563.                     cuda_get_side << < blocks, threads >> > (fastData, fastRightIn, RIGHT, g1, g2,
    g3);
564.                     CSC(cudaGetLastError());
565.                     CSC(cudaMemcpy(RightIn, fastRightIn, sizeof(double) * g2 * g3,
    cudaMemcpyDeviceToHost));
566.                     MPI_Isend(RightIn, g2 * g3, MPI_DOUBLE, _ib(ib + 1, jb, kb), 0,
    MPI_COMM_WORLD, &send_request1_2);
567.                 }
568.
569.                 if (jb + 1 < p2) { //only down side
570.                     cuda_get_side << < blocks, threads >> > (fastData, fastDownIn, DOWN, g1, g2,
    g3);
571.                     CSC(cudaGetLastError());
572.                     CSC(cudaMemcpy(DownIn, fastDownIn, sizeof(double) * g1 * g3,
    cudaMemcpyDeviceToHost));
573.                     MPI_Isend(DownIn, g1 * g3, MPI_DOUBLE, _ib(ib, jb + 1, kb), 0, MPI_COMM_WORLD,
    &send_request2_1);
574.                 }
575.
576.                 if (kb + 1 < p3) { //only back side
577.                     cuda_get_side << < blocks, threads >> > (fastData, fastBackIn, BACK, g1, g2,
    g3);
578.                     CSC(cudaGetLastError());
579.                     CSC(cudaMemcpy(BackIn, fastBackIn, sizeof(double) * g1 * g2,
    cudaMemcpyDeviceToHost));
580.
```

```
581.                     MPI_Isend(BackIn, g1 * g2, MPI_DOUBLE, _ib(ib, jb, kb + 1), 0, MPI_COMM_WORLD,
    &send_request3_1);
582.                 }
583.
584.                 if (ib + 1 < p1) { //only right side
585.                     MPI_Wait(&send_request1_2, &status);
586.                 }
587.                 if (jb + 1 < p2) { //only down side
588.                     MPI_Wait(&send_request2_1, &status);
589.                 }
590.                 if (kb + 1 < p3) { //only back side
591.                     MPI_Wait(&send_request3_1, &status);
592.                 }
593.
594.
595.
596.                 if (ib > 0) { //get left side
597.                     MPI_Irecv(LeftOut, g2* g3, MPI_DOUBLE, _ib(ib - 1, jb, kb), 0, MPI_COMM_WORLD,
    &recv_request1_1);
598.                     MPI_Wait(&recv_request1_1, &status);
599.
600.                     CSC(cudaMemcpy(fastLeftOut, LeftOut, sizeof(double)* g2* g3,
    cudaMemcpyHostToDevice));
601.                     cuda_set_side << < blocks, threads >> > (fastData, fastLeftOut, LEFT, g1, g2,
    g3);
602.                     CSC(cudaGetLastError());
603.                 }
604.                 else {
605.                     cuda_side_edge_values << < blocks, threads >> > (fastData, LEFT, g1, g2, g3,
    Uleft);
606.                     CSC(cudaGetLastError());
607.                 }
608.
609.
610.                 if (jb > 0) { //get up side
611.                     MPI_Irecv(UPOut, g1* g3, MPI_DOUBLE, _ib(ib, jb - 1, kb), 0, MPI_COMM_WORLD,
    &recv_request2_2);
612.                     MPI_Wait(&recv_request2_2, &status);
613.
614.                     CSC(cudaMemcpy(fastUPOut, UPOut, sizeof(double)* g1* g3,
    cudaMemcpyHostToDevice));
615.                     cuda_set_side << < blocks, threads >> > (fastData, fastUPOut, UP, g1, g2, g3);
616.                     CSC(cudaGetLastError());
617.                 }
618.                 else {
619.                     cuda_side_edge_values << < blocks, threads >> > (fastData, UP, g1, g2, g3,
    Uup);
620.                     CSC(cudaGetLastError());
621.                 }
622.
623.
624.                 if (kb > 0) { //get front side
625.                     MPI_Irecv(FrontOut, g1* g2, MPI_DOUBLE, _ib(ib, jb, kb - 1), 0,
    MPI_COMM_WORLD, &recv_request3_2);
626.                     MPI_Wait(&recv_request3_2, &status);
627.
628.                     CSC(cudaMemcpy(fastFrontOut, FrontOut, sizeof(double)* g1* g2,
    cudaMemcpyHostToDevice));
629.                     cuda_set_side << < blocks, threads >> > (fastData, fastFrontOut, FRONT, g1,
    g2, g3);
630.                     CSC(cudaGetLastError());
631.                 }
632.                 else {
633.                     cuda_side_edge_values << < blocks, threads >> > (fastData, FRONT, g1, g2, g3,
    Ufront);
634.                     CSC(cudaGetLastError());
635.                 }
636.
637.                 MPI_Barrier(MPI_COMM_WORLD);
638.
639.                 cuda_main_function << < gridSz, blockSz >> > (fastData, fastNext, g1, g2, g3, hx,
    hy, hz);
640.
```

```
641.                 cuda_error_function << < gridSz, blockSz >> > (fastData, fastNext, g1, g2, g3);
642.
643.                 thrust::device_ptr<double> d_ptr = thrust::device_pointer_cast(fastData);
644.
645.                 double epsTemp[1];
646.
647.                 epsTemp[0] = *(thrust::max_element(d_ptr, d_ptr + (g1 + 2)*(g2 + 2)*(g3+2)));
648.
649.
650.                 MPI_Barrier(MPI_COMM_WORLD);
651.
652.
653.                 MPI_Allgather(epsTemp, 1, MPI_DOUBLE, errors, 1, MPI_DOUBLE, MPI_COMM_WORLD);
654.                 epsTemp[0] = 0;
655.                 for (i = 0; i < numproc; ++i) {
656.                     epsTemp[0] = max(epsTemp[0], errors[i]);
657.                 }
658.
659.                 temp = fastNext;
660.                 fastNext = fastData;
661.                 fastData = temp;
662.
663.                 maxErr = epsTemp[0];
664.
665.                 iter += 1;
666.
667.             } while (maxErr >= eps);
668.
669.
670.             //cout << iter << "\n";
671.
672.             CSC(cudaFree(fastNext));
673.
674.             double* data = (double*)malloc(sizeof(double) * (g1+2)*(g2+2)*(g3+2));
675.
676.             CSC(cudaMemcpy(data, fastData, sizeof(double)*(g1+2)*(g2+2)*(g3+2),
    cudaMemcpyDeviceToHost));
677.
678.
679.             CSC(cudaFree(fastData));
680.
681.             MPI_Barrier(MPI_COMM_WORLD);
682.
683.             memset(buff, ' ', L * g1 * g2 * g3 * sizeof(char));
684.
685.             for (k = 0; k < g3; ++k) {
686.                 for (j = 0; j < g2; j++) {
687.                     for (i = 0; i < g1; i++) {
688.                         if ((i == 0 && j == 0 && (k != 0 || _iby(id) != 0 || _ibz(id) != 0)) &&
    _ibx(id) == 0 && (_iby(id) > 0 || _ibz(id) > 0 || k > 0)) {
689.                             sprintf(buff + (k * (g1 * g2) + j * g1 + i) * L, "\n\n%.6e ",
    data[_i(i, j, k)]);
690.                         }
691.                         else if (i == 0 && _ibx(id) == 0 && (j > 0 || k > 0)) {
692.                             sprintf(buff + (k * (g1 * g2) + j * g1 + i) * L, "\n%.6e ", data[_i(i,
    j, k)]);
693.                         }
694.                         else {
695.                             sprintf(buff + (k * (g1 * g2) + j * g1 + i) * L, "%.6e ", data[_i(i,
    j, k)]);
696.                         }
697.                     }
698.                 }
699.             }
700.             for (i = 0; i < g1 * g2 * g3 * L; ++i) {
701.                 if (buff[i] == '\0')
702.                     buff[i] = ' ';
703.                 //cout << buff[i];
704.             }
705.
706.
707.
708.             MPI_Datatype fileCube;
```

```
709.
710.             MPI_Datatype fileType;
711.
712.             int count = g1 * g2 * g3;
713.
714.
715.             int* lens = new int[count];
716.
717.             for (int i = 0; i < count; ++i) lens[i] = L;
718.
719.             MPI_Aint* adrToFile = new MPI_Aint[count];
720.
721.             MPI_Aint* adrInFile = new MPI_Aint[count];
722.
723.             for (int k = 0; k < g3; ++k) {
724.                 for (int j = 0; j < g2; ++j) {
725.                     for (int i = 0; i < g1; ++i) {
726.                         int CZ = g1 * g2 * g3;
727.                         adrToFile[k * g2 * g1 + j * g1 + i] = (((_ibz(id) * p1 * p2 * CZ) + (k *
    p1 * g1 * p2 * g2)) +
728.                             ((_iby(id) * p1 * g1 * g2) + (j * p1 * g1)) + (_ibx(id) * g1 + i)) *
    sizeof(char) * L;
729.                     }
730.                 }
731.             }
732.
733.             for (int k = 0; k < g3; ++k) {
734.                 for (int j = 0; j < g2; ++j) {
735.                     for (int i = 0; i < g1; ++i) {
736.                         int t = k * g2 * g1 + j * g1 + i;
737.                         adrInFile[t] = t * sizeof(char) * L;
738.                     }
739.                 }
740.             }
741.
742.
743.             MPI_Type_create_hindexed(count, lens, adrToFile, MPI_CHAR, &fileCube);
744.             MPI_Type_create_hindexed(count, lens, adrInFile, MPI_CHAR, &fileType);
745.
746.             MPI_Type_commit(&fileCube);
747.             MPI_Type_commit(&fileType);
748.
749.
750.             if (id == 0) {
751.                 cerr << "Process GRID: " << p1 << "x" << p2 << "x" << p3 << "\n";
752.                 cerr << "Num GRID: " << g1 << "x" << g2 << "x" << g3 << "\n";
753.                 cerr << "File name " << outFile << "\n";
754.                 cerr << "Eps: " << eps << "\n";
755.                 cerr << "lx: " << lx << " ly: " << ly << " lz: " << lz << "\n";
756.                 cerr << "Us: " << Ufront << " , " << Uback << " , " << Uleft << " , " << Uright <<
    " , " << Uup << " , " << Udown << "\n";
757.                 cerr << "U0: " << U0 << "\n";
758.                 cerr << "Iterations: " << iter << "\n";
759.             }
760.
761.             MPI_File fp;
762.
763.             MPI_Request* send_request = new MPI_Request[numproc - 1];
764.             MPI_Request* recv_request = new MPI_Request[numproc - 1];
765.
766.
767.             char* buffName = new char[fileNameL];
768.
769.             if (id == 0) {
770.                 for (int t = 1; t < numproc; ++t) {
771.                     MPI_Isend(outFileC, fileNameL * sizeof(char), MPI_CHAR, t, 0, MPI_COMM_WORLD,
    &send_request[t - 1]);
772.                     MPI_Wait(&send_request[t - 1], &status);
773.                 }
774.                 for (int x = 0; x < fileNameL; ++x) {
775.                     buffName[x] = outFileC[x];;
776.                 }
777.             }
```

```
778.
779.             MPI_Barrier(MPI_COMM_WORLD);
780.
781.             if (id != 0) {
782.                 MPI_Irecv(buffName, fileNameL * sizeof(char), MPI_CHAR, 0, 0, MPI_COMM_WORLD,
    &recv_request[id - 1]);
783.                 MPI_Wait(&recv_request[id - 1], &status);
784.             }
785.
786.             MPI_Barrier(MPI_COMM_WORLD);
787.
788.
789.
790.             MPI_File_open(MPI_COMM_WORLD, buffName, MPI_MODE_CREATE | MPI_MODE_WRONLY,
    MPI_INFO_NULL, &fp);
791.
792.             MPI_File_set_view(fp, 0, MPI_CHAR, fileCube, "native", MPI_INFO_NULL);
793.
794.             MPI_File_write(fp, buff, 1, fileType, &status);
795.
796.             MPI_File_close(&fp);
797.
798.
799.             MPI_Barrier(MPI_COMM_WORLD);
800.
801.
802.             MPI_Finalize();
803.
804.
805.             free(data);
806.             free(buffer);
807.
808.             return 0;
809.         }
```

Результаты

Конфигурация ядра GPU - 64 (8x8 и 4x4x4)						
Размерность учитывается общая (eps 1e-3)						
Разм / Проц	1x1x1	CPU (1x1x1)	1x1x2	1x2x2	2x2x2	2x2x5
10x10x10	0.39613	0.01309	1.15962	1.56812	2.03052	15.2079
20x20x20	0.83935	0.20482	1.78878	3.12164	5.93209	39.7591
50x50x50	8.44923	9.00067	9.76722	13.5464	19.9702	162.493
100x100x100	74.0783	127.757	80.9000	86.5127	106.856	422.483

Надежды не оправдали ожиданий. Все еще даже хуже, чем в ЛР 9. При больших данных так и вовсе компьютер виснет, да причем вообще никак не реагирует, а мышка даже не двигается, даже медленно, как в ЛР 9 было. Похоже, комбинация CUDA + MPI не слишком подходит для данной задачи. У меня есть предположение, почему все так плохо? Мы очень много раз копируем память с GPU на CPU и обратно, храним кучу буферов, постоянно обращаемся к памяти GPU. На это уходит много времени, и видимо, программа будет работать лучше только на очень больших данных. Да и думаю, нужен целый кластер машин, чтобы она смогла превзойти программы ЛР9 и ЛР7. CUDA хорошо подходит, когда мы обращаемся один или пару раз к видеокарте, а не кучу раз копируем туда-сюда буфера и запускаем, к тому же запуск тоже требует какого-то времени (как было видно еще в ЛР1). Возможно, я установил еще высокую конфигурацию ядра, при меньшей, программа могла бы работать и получше.

Выводы

Очень рад, что все это позади. Эти три лабораторных мне показались настоящим испытанием, а я их делал так, будто это целая курсовая. Ошибки, над которыми я сидел целыми часами оказались наиглупейшими. Область видимости переменной или же лишнее умножение на размер типа. Такое даже не заметишь сразу, а программа не работает. Пытаешься искать часами, когда ошибка находится прям под носом. В данной “подборке” лабораторных я познакомился с MPI, OpenMP, MPI-io, производными типами данных, а также укрепил знания CUDA и узнал немного про библиотеку thrust, которая позволяет быстро выполнять некоторые операции прям на памяти видеокарты. Я думаю, данный опыт был полезен, возможно, он мне пригодится, хотя не факт, но всякое может быть. Было интересно, но и одновременно, потратил кучу нервов пытаясь отлаживать программы и находить ошибки. Программа из ЛР 8 очень хороша в производительности если много видеокарт, несколько машин выполняют одну программу. А с моей, хоть и игровой ноутбучной видеокартой, она работает хуже чем программы из ЛР 7 и ЛР 9. Вот как-то так.