Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №8 по курсу «Программирование графических процессоров»
(Лабораторная работа №3 по курсу «Параллельная обработка данных»)

Технология MPI и технология CUDA. MPI-IO

Студент: Л. Я. Вельтман
Преподаватель: К. Г. Крашенинников
А. Ю. Морозов
Группа: М8О-407Б
Дата:
Оценка:
Подпись:

Москва, 2020

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 7, 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.

Запись результатов в файл должна осуществляться параллельно всеми процессами. Необходимо создать производный тип данных, определяющий шаблон записи данных в файл.

Вариант 2:
MPI_Type_hvector.
Входные данные: На первой строке заданы три числа: размер сетки процессов. Гарантируется, что при запуске программы количество процессов будет равно произведению этих трех чисел. На второй строке задается размер блока, который будет обрабатываться одним процессом: три числа. Далее задается путь к выходному файлу, в который необходимо записать конечный результат работы программы и точность ϵ. На последующих строках описывается задача: задаются размеры области, l_x, l_y, l_z, граничные условия: $u_{down}, u_{up}, u_{left}, u_{right}, u_{front}, u_{back}$ и начальное значение u_0.

Выходные данные: В файл, определенный во входных данных, необходимо напечатать построчно значения $(u_{1,1,1}, u_{2,1,1}, \ldots, u_{1,2,1}, u_{2,2,1}, .., , u_{n_x-1,n_y,n_z}, u_{n_x,n_y,n_z})$ в ячейках сетки в формате с плавающей запятой с семью знаками мантиссы.

Программное и аппаратное обеспечение:
Device Number: 0
Device name: GeForce GT 545
TotalGlobalMem: 3150381056
Const Mem : 65536
Max shared mem for blocks 49152
Max regs per block 32768
Max thread per block 1024
multiProcessorCount : 3

maxThreadsDim 1024 1024 64
maxGridSize 65535 65535 65535
OS: macOS Catalina version 10.15.5
Text Editor: Sublime Text 3

1 Описание

Метод решения

В данной лабораторной работе нужно решить задачу, описанную в лабораторной работе 7. Одна итерация решения исходной задачи состоит из трех этапов.
На первом этапе происходит обмен граничными слоями между процессами.
На втором этапе выполняется обновление значений во всех ячейках.
И третий этап заключается в вычислении погрешности: сначала локально в рамках каждого процесса, а потом через обмены и во всей области.

Описание программы

Обмен граничными слоями между процессами происходит с помощью вспомогательных буферов, на каждой итерации копируются данные между device и host памятью, чтобы обновить принимаемые или отсылаемые данные. При помощи функций MPI_Isend и MPI_Irecv производится асинхронный обмен, затем ожидаем завершения этих асинхронных процедур благодаря MPI_Wait и обновление значений во всех ячейках.

Для расчета значения, отвечающего за сходимость, используется библиотека thrust, с помощью которой находится максимум. Нужно создать массив, в котором количество ячеек равно максимальному значению числа нитей. В каждой ячейке этого массива будет хранится максимальное значение, полученное каждым из потоков. Позже из этого массива находится максимум и дальше применяется функция MPI_Allreduce. Она имеет варианты каждой из операции редукции, где результат возвращается всем процессам группы (в нашем случае максимум).

Запись результатов в файл осуществляется параллельно всеми процессами. Для этого было необходимо создать производный тип данных - hvector, определяющий шаблон записи данных в файл.

2 Исходный код

```
#include <iostream>
#include <iomanip>
#include <string>
#include <algorithm>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <thrust/extrema.h>
#include <thrust/device_vector.h>



#define CSC(call) \
do { \
    cudaError_t res = call; \
    if (res != cudaSuccess) { \
        fprintf(stderr, "ERROR: Cuda in %s:%d. Message: %s\n", \
                __FILE__, __LINE__, cudaGetErrorString(res)); \
        exit(0); \
    } \
} while(0)


#define Update_ij(i, j, step, side) \
{ \
    i += step; \
    while(i >= side) \
    { \
        i -= side; \
        ++j; \
    } \
} \


const int axes = 3;
const int directions = 6;
const int BlockCount = 32;
const int ThreadsCount = 32;


void WaitAll(int* coords, int* gridProc, MPI_Request *arrOfRequests)
{
    MPI_Status tmp;

    for (int i = 0, j = 1; i < axes; ++i, j += 2)
    {
        if (coords[i] > 0)
```

```
48          {
49              MPI_Wait(&arrOfRequests[i * 2], &tmp);
50          }
51          if (coords[i] < gridProc[i] - 1)
52          {
53              MPI_Wait(&arrOfRequests[j], &tmp);
54          }
55      }
56  }
57
58
59  __host__ __device__
60  double FindMax(double a, double b, double max)
61  {
62      double fbs = a - b;
63      fbs = fbs > 0.0 ? fbs : -fbs;
64      if (fbs > max)
65      {
66          max = fbs;
67      }
68      return max;
69  }
70
71
72  __host__ __device__
73  int GetPos(int i, int j, int k, int nY, int nX)
74  {
75      return i + (j + k * nY) * nX;
76  }
77
78
79  void GetCoords(int* coords, int rank, int* gridProc)
80  {
81      coords[2] = rank / gridProc[0] / gridProc[1];
82      coords[1] = (rank - coords[2] * gridProc[0] * gridProc[1]) / gridProc[0];
83      coords[0] = rank - (coords[2] * gridProc[1] + coords[1]) * gridProc[0];
84  }
85
86
87  int GetRank(int* coords, int* gridProc)
88  {
89      return coords[0] + gridProc[0] * (coords[2] * gridProc[1] + coords[1]);
90  }
91
92
93  void Printer(FILE* out, double* arr, int size)
94  {
95      for (int i = 0; i < size; ++i)
96      {
```

```
97          fprintf(out, "%.6e ", arr[i]);
98      }
99      fprintf(out, "\n");
100 }
101
102 void WriteOutWithMPI(int* gridProc, int* block,
103                      std::string& output, double* grid, int* coords)
104 {
105     //convert double data to char values
106     int size = block[0] * block[1] * block[2];
107     const int doubleSize = 14;
108     char* charValues = new char[size * doubleSize];
109
110     //memset(charValues, ' ', size * doubleSize);
111     for (int k = 1; k <= block[2]; ++k)
112     {
113         for (int j = 1; j <= block[1]; ++j)
114         {
115             int i, len;
116             for (i = 1; i < block[0]; ++i)
117             {
118                 len = sprintf(&charValues[GetPos(i - 1, j - 1, k - 1, block[1], block
                        [0]) * doubleSize], "%.6e ", grid[GetPos(i, j, k, block[1] + 2,
                        block[0] + 2)]);
119                 if (len < doubleSize)
120                 {
121                     charValues[GetPos(i - 1, j - 1, k - 1, block[1], block[0]) *
                            doubleSize + len] = ' ';
122                 }
123             }
124
125             len = sprintf(&charValues[GetPos(i - 1, j - 1, k - 1, block[1], block[0]) *
                     doubleSize], "%.6e\n", grid[GetPos(i, j, k, block[1] + 2, block[0] + 2)
                    ]);
126             if (len < doubleSize)
127             {
128                 charValues[GetPos(i - 1, j - 1, k - 1, block[1], block[0]) * doubleSize
                        + len] = '\n';
129             }
130         }
131     }
132
133     MPI_Aint stride = block[0] * doubleSize * gridProc[0];
134     MPI_Aint gstride = stride * block[1] * gridProc[1];
135     MPI_Aint position = coords[0] * doubleSize * block[0];
136     position += stride * block[1] * coords[1];
137     position += gstride * block[2] * coords[2];
138
139     int blocklength = block[0] * doubleSize;
```

```
140
141     MPI_File fp;
142     MPI_Datatype square, rectangle;
143
144     MPI_Type_create_hvector(block[1], blocklength, stride, MPI_CHAR, &square);
145     MPI_Type_commit(&square);
146
147     MPI_Type_create_hvector(block[2], 1, gstride, square, &rectangle);
148     MPI_Type_commit(&rectangle);
149
150     MPI_File_delete(output.c_str(), MPI_INFO_NULL);
151     MPI_File_open(MPI_COMM_WORLD, output.c_str(), MPI_MODE_CREATE | MPI_MODE_RDWR,
            MPI_INFO_NULL, &fp);
152
153     MPI_File_set_view(fp, position, MPI_CHAR, rectangle, "native", MPI_INFO_NULL);
154     MPI_File_write_all(fp, charValues, doubleSize*size, MPI_CHAR, MPI_STATUS_IGNORE);
155
156     MPI_File_close(&fp);
157
158     delete[] charValues;
159 }
160
161
162 void FillBuffer(double* buf, int size, double val)
163 {
164     for (int i = 0; i < size; ++i)
165     {
166         buf[i] = val;
167     }
168 }
169
170
171 void InitBufsEdge(double** sendBuf, double** getBuf, double** deviceSendBuf, double**
        deviceGetBuf, int* sizeEdges, int* gridProc, int* coords, double* u, double u0)
172 {
173     for (int i = 0, j = 1; i < axes; ++i, j += 2)
174     {
175         sendBuf[i * 2] = new double[sizeEdges[i]];
176         getBuf[i * 2] = new double[sizeEdges[i]];
177         sendBuf[j] = new double[sizeEdges[i]];
178         getBuf[j] = new double[sizeEdges[i]];
179         FillBuffer(sendBuf[i * 2], sizeEdges[i], u0);
180         FillBuffer(sendBuf[j], sizeEdges[i], u0);
181
182         CSC(cudaMalloc((void**)&deviceSendBuf[i * 2], sizeof(double) * sizeEdges[i]));
183         CSC(cudaMalloc((void**)&deviceGetBuf[i * 2], sizeof(double) * sizeEdges[i]));
184         CSC(cudaMalloc((void**)&deviceSendBuf[j], sizeof(double) * sizeEdges[i]));
185         CSC(cudaMalloc((void**)&deviceGetBuf[j], sizeof(double) * sizeEdges[i]));
186
```

```
187         if (!coords[i])
188         {
189             FillBuffer(getBuf[i * 2], sizeEdges[i], u[i * 2]);
190         }
191         if (coords[i] == gridProc[i] - 1)
192         {
193             FillBuffer(getBuf[j], sizeEdges[i], u[j]);
194         }
195     }
196 }
197
198
199 void Clear(double* grid, double* newGrid, double** sendBuf, double** getBuf,
200           double* deviceGrid, double* deviceNewGrid, double* deviceMaxValues,
201           double** deviceSendBuf, double** deviceGetBuf)
202 {
203     delete[] grid;
204     delete[] newGrid;
205
206     CSC(cudaFree(deviceGrid));
207     CSC(cudaFree(deviceNewGrid));
208     CSC(cudaFree(deviceMaxValues));
209
210     for (int i = 0, j = 1; i < axes; ++i, j += 2)
211     {
212         delete[] sendBuf[i * 2];
213         delete[] getBuf[i * 2];
214         delete[] sendBuf[j];
215         delete[] getBuf[j];
216
217         CSC(cudaFree(deviceSendBuf[i * 2]));
218         CSC(cudaFree(deviceGetBuf[i * 2]));
219         CSC(cudaFree(deviceSendBuf[j]));
220         CSC(cudaFree(deviceGetBuf[j]));
221     }
222 }
223
224
225 void GetNeighbours(int* neighb, int* gridProc, int* coords)
226 {
227     int tmp[axes] = {coords[0], coords[1], coords[2]};
228
229     for (int i = 0, j = 1; i < axes; ++i, j += 2)
230     {
231         --tmp[i];
232         neighb[2 * i] = GetRank(tmp, gridProc);
233         tmp[i] += 2;
234         neighb[j] = GetRank(tmp, gridProc);
235         --tmp[i];
```

```
236     }
237 }
238
239
240 void Isend_Irecv(MPI_Request* in, MPI_Request* out, double** sendBuf, double** getBuf,
        int* sizeEdges, int* gridProc, int* coords, int* neighb)
241 {
242     for (int i = 0, j = 1; i < axes; ++i, j += 2)
243     {
244         if (coords[i] > 0)
245         {
246             MPI_Isend(sendBuf[i * 2], sizeEdges[i], MPI_DOUBLE,
247                       neighb[i * 2], 0, MPI_COMM_WORLD, &out[i * 2]);
248             MPI_Irecv(getBuf[i * 2], sizeEdges[i], MPI_DOUBLE,
249                       neighb[i * 2], 0, MPI_COMM_WORLD, &in[i * 2]);
250         }
251         if (coords[i] < gridProc[i] - 1)
252         {
253             MPI_Isend(sendBuf[j], sizeEdges[i], MPI_DOUBLE,
254                       neighb[j], 0, MPI_COMM_WORLD, &out[j]);
255             MPI_Irecv(getBuf[j], sizeEdges[i], MPI_DOUBLE,
256                       neighb[j], 0, MPI_COMM_WORLD, &in[j]);
257         }
258     }
259 }
260
261
262 __global__
263 void FillInEdgesKernel(double* grid, double* getBuf, int ax,
264                        int lim, int nX, int nY, int nZ, int tmpX, int tmpY, int tmpZ)
265 {
266     int offset = gridDim.x * blockDim.x;
267     int threadId = blockDim.x * blockIdx.x + threadIdx.x;
268
269     // formula to get coords x + (y + z * ny) * nx
270     #define idx_in(i, j) ( \
271         tmpX * (lim * (nX - 1) + (i + j * nY) * nX) + \
272         tmpY * (i + (lim * (nY - 1) + j * nY) * nX) + \
273         tmpZ * (i + (j + (lim * (nZ - 1)) * nY) * nX) \
274     ) \
275
276     int first_n = (ax == 2) ? nY : nZ;
277     int second_n = (ax != 0) ? nX : nY;
278     int i = 0;
279     int j = 0;
280     Update_ij(i, j, threadId, second_n);
281
282     while (j < first_n)
283     {
```

```
        grid[idx_in(i, j)] = getBuf[i + j * second_n];
        Update_ij(i, j, offset, second_n);
    }
}


__global__
void FillOutEdgesKernel(double* newGrid, double* sendBuf, int ax,
                        int lim, int nX, int nY, int nZ, int tmpX, int tmpY, int tmpZ)
{
    int offset = gridDim.x * blockDim.x;
    int threadId = blockDim.x * blockIdx.x + threadIdx.x;

    #define idx_out(i, j) ( \
        tmpX * (lim * (nX - 3) + 1 + (i + j * nY) * nX) + \
        tmpY * (i + (lim * (nY - 3) + 1 + j * nY) * nX) + \
        tmpZ * (i + (j + (lim * (nZ - 3) + 1) * nY) * nX) \
    ) \

    int first_n = (ax == 2) ? nY : nZ;
    int second_n = (ax != 0) ? nX : nY;

    int i = 0;
    int j = 0;
    Update_ij(i, j, threadId, second_n);

    while (j < first_n)
    {
        sendBuf[i + j * second_n] = newGrid[idx_out(i, j)];
        Update_ij(i, j, offset, second_n);
    }
}


__device__
void UpdateCoords(int& i, int& j, int& k, int shift, int nX, int nY)
{
    i += shift;
    while (i > nX)
    {
        i -= nX;
        ++j;
    }
    while (j > nY)
    {
        j -= nY;
        ++k;
    }
}
```

```
333
334
335 __global__
336 void CalculateNewValuesKernel(double* grid, double* newGrid, double* maxValues,
337              int nX, int nY, int nZ, double hX, double hY, double hZ, double b, int
                 proc)
338 {
339     int offset = gridDim.x * blockDim.x;
340     int threadId = blockDim.x * blockIdx.x + threadIdx.x;
341
342     int i = 1;
343     int j = 1;
344     int k = 1;
345
346     UpdateCoords(i, j, k, threadId, nX - 2, nY - 2);
347     maxValues[threadId] = 0.0;
348
349     while(k <= nZ - 2)
350     {
351         double a = (grid[GetPos(i - 1, j, k, nY, nX)] + grid[GetPos(i + 1, j, k, nY, nX
                )]) / (hX * hX);
352         a += (grid[GetPos(i, j - 1, k, nY, nX)] + grid[GetPos(i, j + 1, k, nY, nX)]) /
                (hY * hY);
353         a += (grid[GetPos(i, j, k - 1, nY, nX)] + grid[GetPos(i, j, k + 1, nY, nX)]) /
                (hZ * hZ);
354
355         newGrid[GetPos(i, j, k, nY, nX)] = a / b;
356         maxValues[threadId] = FindMax(grid[GetPos(i, j, k, nY, nX)], newGrid[GetPos(i,
                j, k, nY, nX)], maxValues[threadId]);
357         UpdateCoords(i, j, k, offset, nX - 2, nY - 2);
358     }
359 }
360
361
362
363 void Start(int* gridProc, int* block, std::string& output,
364           double eps, double* l, double* u,
365           double u0, int numProcs, int rank)
366 {
367     MPI_Bcast(gridProc, axes, MPI_INT, 0, MPI_COMM_WORLD);
368     MPI_Bcast(block, axes, MPI_INT, 0, MPI_COMM_WORLD);
369     MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
370     MPI_Bcast(l, axes, MPI_DOUBLE, 0, MPI_COMM_WORLD);
371     MPI_Bcast(u, directions, MPI_DOUBLE, 0, MPI_COMM_WORLD);
372     MPI_Bcast(&u0, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
373     {
374         int cntChars = output.size();
375         MPI_Bcast(&cntChars, 1, MPI_INT, 0, MPI_COMM_WORLD);
376
```

```
377         output.resize(cntChars);
378 
379         MPI_Bcast((char*)output.c_str(), cntChars, MPI_CHAR, 0, MPI_COMM_WORLD);
380         output.push_back('\0');
381     }
382 
383 
384     MPI_Request in[directions], out[directions];
385 
386     int coords[axes];
387     GetCoords(coords, rank, gridProc);
388 
389     int neighb[directions];
390     GetNeighbours(neighb, gridProc, coords);
391 
392     int size = (block[0] + 2) * (block[1] + 2) * (block[2] + 2);
393     double* grid = new double[size];
394     double* newGrid = new double[size];
395     FillBuffer(grid, size, u0);
396     FillBuffer(newGrid, size, u0);
397 
398     int deviceCount;
399     cudaGetDeviceCount(&deviceCount);
400     cudaSetDevice(rank % deviceCount);
401 
402     // for device
403     double* deviceSendBuf[directions];
404     double* deviceGetBuf[directions];
405     double* deviceMaxValues;
406     double* deviceGrid;
407     double* deviceNewGrid;
408 
409     CSC(cudaMalloc((void**)&deviceMaxValues, sizeof(double) * BlockCount * ThreadsCount
        ));
410 
411     CSC(cudaMalloc((void**)&deviceGrid, sizeof(double) * size));
412 
413     CSC(cudaMalloc((void**)&deviceNewGrid, sizeof(double) * size));
414 
415     CSC(cudaMemcpy(deviceGrid, grid, sizeof(double) * size, cudaMemcpyHostToDevice));
416 
417     double* sendBuf[directions];
418     double* getBuf[directions];
419 
420     int sizeEdges[axes];
421     sizeEdges[0] = (block[1] + 2) * (block[2] + 2); // y z
422     sizeEdges[1] = (block[0] + 2) * (block[2] + 2); // x z
423     sizeEdges[2] = (block[0] + 2) * (block[1] + 2); // x y
424 
```

```
425     InitBufsEdge(sendBuf, getBuf, deviceSendBuf, deviceGetBuf, sizeEdges, gridProc,
            coords, u, u0);
426
427     double nX = gridProc[0] * block[0];
428     double nY = gridProc[1] * block[1];
429     double nZ = gridProc[2] * block[2];
430
431     double hX = (double)(l[0] / nX);
432     double hY = (double)(l[1] / nY);
433     double hZ = (double)(l[2] / nZ);
434
435     double b = 2 * (1/(hX * hX) + 1/(hY * hY) + 1/(hZ * hZ));
436
437     int tmpkernel[axes] = {0};
438     int n_X = block[0] + 2, n_Y = block[1] + 2, n_Z = block[2] + 2;
439
440     double maxConvergence;
441     double globalMax = 0.0;
442
443     thrust::device_ptr<double> ptr = thrust::device_pointer_cast(deviceMaxValues);
444     do {
445         maxConvergence = 0.0;
446
447         Isend_Irecv(in, out, sendBuf, getBuf, sizeEdges, gridProc, coords, neighb);
448
449         WaitAll(coords, gridProc, in);
450         for (int i = 0, j = 1; i < axes; ++i, j += 2)
451         {
452             CSC(cudaMemcpy(deviceGetBuf[i * 2], getBuf[i * 2], sizeof(double) *
                    sizeEdges[i], cudaMemcpyHostToDevice));
453             CSC(cudaMemcpy(deviceGetBuf[j], getBuf[j], sizeof(double) * sizeEdges[i],
                    cudaMemcpyHostToDevice));
454         }
455
456         for (int d = 0; d < directions; ++d)
457         {
458             int ax = d >> 1;
459             tmpkernel[ax] = 1;
460             int lim = (d & 1);
461
462             FillInEdgesKernel<<<BlockCount, ThreadsCount>>>(deviceGrid, deviceGetBuf[d
                    ],
463                                                             ax, lim, n_X, n_Y, n_Z,
464                                                             tmpkernel[0], tmpkernel
                                                                [1], tmpkernel[2]);
465             tmpkernel[ax] = 0;
466         }
467         cudaThreadSynchronize();
468         CSC(cudaGetLastError());
```

```
469
470         CalculateNewValuesKernel<<<BlockCount, ThreadsCount>>>(deviceGrid,
                deviceNewGrid, deviceMaxValues,
471                                                                n_X, n_Y, n_Z, hX, hY, hZ, b,
                                                                    rank);
472         cudaThreadSynchronize();
473         CSC(cudaGetLastError());
474
475         WaitAll(coords, gridProc, out);
476
477         for (int d = 0; d < directions; ++d)
478         {
479             int ax = d >> 1;
480             tmpkernel[ax] = 1;
481             int lim = (d & 1);
482
483             FillOutEdgesKernel<<<BlockCount, ThreadsCount>>>(deviceNewGrid,
                    deviceSendBuf[d],
484                                                              ax, lim, n_X, n_Y, n_Z,
485                                                              tmpkernel[0], tmpkernel
                                                                  [1], tmpkernel[2]);
486             CSC(cudaGetLastError());
487
488             tmpkernel[ax] = 0;
489         }
490         cudaThreadSynchronize();
491
492
493         for (int i = 0, j = 1; i < axes; ++i, j += 2)
494         {
495             CSC(cudaMemcpy(sendBuf[i * 2], deviceSendBuf[i * 2], sizeof(double) *
                    sizeEdges[i], cudaMemcpyDeviceToHost));
496             CSC(cudaMemcpy(sendBuf[j], deviceSendBuf[j], sizeof(double) * sizeEdges[i],
                     cudaMemcpyDeviceToHost));
497         }
498
499
500         maxConvergence = *thrust::max_element(ptr, ptr + BlockCount * ThreadsCount);
501
502         MPI_Allreduce(&maxConvergence, &globalMax, 1, MPI_DOUBLE, MPI_MAX,
                MPI_COMM_WORLD);
503
504
505         double* tmp = deviceGrid;
506         deviceGrid = deviceNewGrid;
507         deviceNewGrid = tmp;
508     } while (globalMax >= eps);
509
510     cudaMemcpy(grid, deviceGrid, sizeof(double) * size, cudaMemcpyDeviceToHost);
```

```
511
512     WriteOutWithMPI(gridProc, block, output, grid, coords);
513     Clear(grid, newGrid, sendBuf, getBuf, deviceGrid, deviceNewGrid, deviceMaxValues,
            deviceSendBuf, deviceGetBuf);
514 }
515
516
517
518 int main(int argc, char *argv[])
519 {
520     std::ios_base::sync_with_stdio(false);
521     std::cin.tie(nullptr);
522     int gridProc[axes], block[axes];
523     double l[axes];
524     double eps, u0;
525     double u[directions];
526     std::string output;
527
528     int numProcs, rank;
529
530     MPI_Init(&argc, &argv);
531     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
532     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
533
534     if (!rank) // main process
535     {
536         std::cin >> gridProc[0] >> gridProc[1] >> gridProc[2];
537         std::cin >> block[0] >> block[1] >> block[2];
538         std::cin >> output;
539         std::cin >> eps;
540         std::cin >> l[0] >> l[1] >> l[2];
541         //front back down up left right
542         std::cin >> u[4] >> u[5] >> u[0] >> u[1] >> u[2] >> u[3];
543         std::cin >> u0;
544     }
545
546     Start(gridProc, block, output, eps, l, u, u0, numProcs, rank);
547
548     MPI_Finalize();
549
550     return 0;
551 }
```

3 Результаты

proc	block	time CPU	time MPI	time MPI + CUDA	winner
(1, 1, 1)	(10, 10, 10)	104.456 ms	134.657 ms	302.983 ms	CPU
(2, 2, 2)	(10, 10, 10)	2477.87 ms	4168.76 ms	11067.1 ms	CPU
(1, 1, 1)	(20, 20, 20)	1774.47 ms	1020.22 ms	1014.8 ms	MPI + CUDA
(2, 2, 2)	(20, 20, 20)	58620.1 ms	5777.1 ms	42431.8 ms	MPI
(1, 1, 1)	(40, 40, 40)	61367.1 ms	23948.9 ms	7827.49 ms	MPI + CUDA
(2, 2, 2)	(40, 40, 40)	1.65423e+06 ms	156571 ms	275238 ms	MPI

4 Выводы

Для выполнения данной лабораторной работы нужно было решить задачу Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода. Для этого понадобилось совместить работу CUDA и MPI. MPI позволяет программе работать в мультипроцессорном режиме, причем каждый процесс будет распараллелен на несколько потоков. При всей своей благозвучности данная схема в реальности работает не так хорошо, как ожидалось. Связка CUDA + MPI выиграла на 2 тестах из 6, где использовалась сетка процессов 1x1x1, когда использовалась 2x2x2, то такая программа уступала программе, работающей только с MPI. Думаю, что полученные результаты сильно зависят от того, что время тратится на создание и инициализацию потоков, также кластеру могло не хватить ресурсов, к тому же допускаю, что сказывается моя неопытность в использовании этих технологий, уверена, что есть еще много разных фич, которые могут ускорить программу.