МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №3
по курсу «Параллельная обработка данных»

Технология MPI и технология CUDA. MPI-IO

Выполнил: В.А. Петросян
Группа: 8О-408Б-17
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

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.
Вариант 1: MPI_Type_create_subarray. (Обмен граничными слоями через bsend, контроль сходимости allgather)

Программное и аппаратное обеспечение

Сведения о системе:

1. Процессор: Intel Core i7-Q720 1.60GHz
2. Количество ядер 4.
3. Количество потоков 8.
4. Оперативная память: 8 ГБ
5. HDD: 465 ГБ

Программное обеспечение:

1. OS: Windows 7
2. IDE: Visual Studio 2019
3. Компиляторы: nvcc and mpic++

Метод решения

Математическая постановка:

$$\frac{d^2u(x,y,z)}{dx^2} + \frac{d^2u(x,y,z)}{dy^2} + \frac{d^2u(x,y,z)}{dz^2} = 0\,,$$

$$u(x \le 0, y, z) = u_{left}\,,$$
$$u(x \ge l_x, y, z) = u_{right}\,,$$
$$u(x,\ y \le 0, z) = u_{front}\,,$$
$$u(x,\ y \ge l_y, z) = u_{back}\,,$$
$$u(x,\ y,\ z \le 0) = u_{down}\,,$$
$$u(x,\ y,\ z \ge l_z) = u_{up}\,.$$

Над пространством строится регулярная сетка. С каждой ячейкой сопоставляется значение функции u в точке соответствующей центру ячейки. Граничные условия реализуются через виртуальные ячейки, которые окружают рассматриваемую область.

Поиск решения сводится к итерационному процессу:

$$u_{i,j,k}^{(k+1)} = \frac{\left(u_{i+1,j,k}^{(k)} + u_{i-1,j,k}^{(k)}\right)h_x^{-2} + \left(u_{i,j+1,k}^{(k)} + u_{i,j-1,k}^{(k)}\right)h_y^{-2} + \left(u_{i,j,k+1}^{(k)} + u_{i,j,k-1}^{(k)}\right)h_z^{-2}}{2\left(h_x^{-2} + h_y^{-2} + h_z^{-2}\right)}\,,$$

Опишу немного логику работы с данными. Допустим размер блока, который обсчитывает один процесс это x * y * z. Мы выделим на каждое измерение два дополнительных элемента для хранения граничных условий. Теперь блок имеет размер (x + 2) * (y + 2) * (z + 2) . Храним данные блока в виде одномерного массива, но обращаемся к нему как к трёхмерному. Чтобы было проще взаимодействовать с массивом, напишем пару макросов для правильного доступа по индексу к данным.

```
// Индексация внутри блока
#define _i(i, j, k) (((k) + 1)*(blockY + 2)*(blockX + 2) + ((j) + 1)
*(blockX + 2) + (i) + 1)
#define _ix(id) (((id) % (blockX + 2)) - 1)
#define _iy(id) ((((id) % ((blockY + 2) * (blockX + 2))) / (blockX + 2)) - 1)
#define _iz(id) ( ( (id) / ((blockY + 2) * (blockX + 2)) ) - 1)
```

Пока не достигнем нужной точности $\mathcal{E}$ будем делиться нашими данными с соседними по сетке процессами.

Описание программы

```
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numproc); - общее количество процессов.
MPI_Comm_rank(MPI_COMM_WORLD, &id); - номер нашего процесса 0 <= id < numproc.
```

```
MPI_Bcast(&blockX, 1, MPI_INT, 0, MPI_COMM_WORLD); - “Широковещательное
сообщение” передает всем процессам значение переменной blockX.
```

```
ib = _ibx(id); - индексация процесса в сетке блоков по x
jb = _iby(id); - индексация процесса в сетке блоков по y
kb = _ibz(id); - индексация процесса в сетке блоков по z
```

```
int buffer_size = 12 * sizeOfBuff * sizeof(double) + 12 * MPI_BSEND_OVERHEAD;
double *buffer = (double *)malloc(buffer_size);
MPI_Buffer_attach(buffer, buffer_size); - через этот буфер процесс будет
общаться со всеми остальными. Я выделим место с двойным запасом.
```

```
for(i = -1; i <= blockX; i++){
   for(j = -1; j <= blockY; j++){
       for(k = -1; k <= blockZ; k++){
           data[_i(i, j, k)] = startU; - инициализация начальным условием
       }
   }
}
```

```
__global__ void kernel_copy_xy(double *plane, double *data, int blockX, int bl
ockY, int blockZ, int k, bool direction, double defVal){
    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;
    int i, j;

    if( direction == true ){
        for(j = idy; j < blockY; j += offsety){
            for(i = idx; i < blockX; i += offsetx){
                plane[j * blockX + i] = data[_i(i, j, k)];
            }
        }
    }
    else{
        if( plane != NULL ){
            for(j = idy; j < blockY; j += offsety){
                for(i = idx; i < blockX; i += offsetx){
                    data[_i(i, j, k)] = plane[j * blockX + i];
                }
            }
        }
        else{
            for(j = idy; j < blockY; j += offsety){
                for(i = idx; i < blockX; i += offsetx){
                    data[_i(i, j, k)] = defVal;
                }
            }
        }
    }
}
```

Выше приведён код ядра, которое копирует значения между CPU и GPU в двух направлениях. Поведение определяется с помощью булевского флага **direction**. Если его значение **True**, то нужно с GPU копировать значения на CPU, иначе наоборот. При копировании на GPU есть свои нюансы. Чтобы не писать много ядер, всё было максимально компактно помещено в одно ядро, работающее с плоскостью XY в данном случае. Так как некоторые процессы ничего не получают от соседей по сетке процессов, то для них массив plane ничего не будет содержать. Можно было конечно в цикле на CPU инициализировать plane граничными значениями, но я выбрал другой путь. Добавил вложенный if, который проверяет если plane == NULL, то GPU сам инициализирует данные граничными значениями, которые передаются через параметр **defVal**.

В ядре есть два условных оператора. Считаю, что это не вызовет дивергенции потоков потому что если процесс крайний по какому-то из трёх измерений, то у него все значения будут инициализированы одинаковым образом. Возможно, я допускаю

ошибку, написав всё компактно в одном ядре, и если разобью код на отдельные ядра, то программа заработает быстрее.

Код ядра, который подсчитывает значения для конкретного процесса, не представляет особого интереса. Используется трехмерная сетка для сохранения логики доступа к массиву как в 7 лабораторной работе. Цикл почти никак не изменился.

После подсчета значений приходится запустить ядро заполнения ошибок, которое в коде называется **kernel2.** Оно во все внутренние значения записывает модуль разности текущего и предыдущего шага, а во все граничные ячейки записывает ноль. Далее вызывается поиск максимального значения из библиотеки thrust, который возвращает нам нашу локальную ошибку. У thrust очень приятный интерфейс работы, напоминающий STL из C++. В вычислительной части больше ничего не менял. Обмен ошибками происходит точно так же как в предыдущей лабораторной работе.

Стоит уделить особое внимание записи данных в файл. Если в прошлой лабораторной работе мы использовали схему пересылки всеми процессами значений нулевому процессу, у которого был открыт файл на запись, то теперь всё иначе. Теперь все процессы пишут в один файл параллельно.

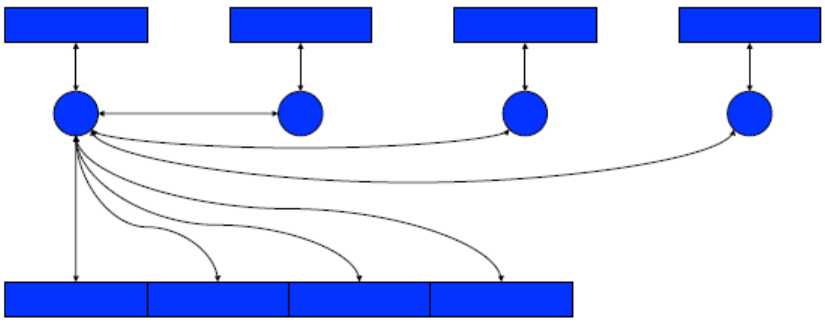
По варианту использую MPI_Type_create_subarray.

```
MPI_Datatype filetype;
int array_of_sizes[3] = { gridZ * blockZ, gridY * blockY, gridX * blockX *
 n_size};
int array_of_subsizes[3] = { blockZ, blockY, blockX * n_size};
int array_of_starts[3] = {_ibz(id) * blockZ,_iby(id) * blockY, _ibx(id) *
blockX * n_size};
MPI_Type_create_subarray(3, array_of_sizes, array_of_subsizes, array_of_st
arts, MPI_ORDER_C , MPI_CHAR, &filetype);
MPI_Type_commit(&filetype);
```

Запись в файл происходит при помощи созданного типа filetype. Он определяет для каждого процесса, каким образом нужно делать смещения при записи данных в файл.

```
MPI_File fp;
MPI_File_delete(outputFile.c_str(), MPI_INFO_NULL);
MPI_File_open(MPI_COMM_WORLD, outputFile.c_str(), MPI_MODE_CREATE |
MPI_MODE_WRONLY, MPI_INFO_NULL, &fp);
MPI_File_set_view(fp, 0, MPI_CHAR, filetype, "native", MPI_INFO_NULL);
MPI_File_write_all(fp, buff, (blockX) * (blockY) * (blockZ) * n_size,
MPI_CHAR, MPI_STATUS_IGNORE);
MPI_File_close(&fp);
```

Non-Parallel I/O



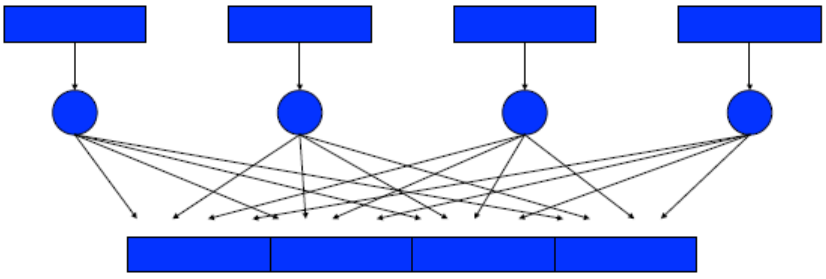
Плюсы :

- Нет необходимости подключать специализированную библиотеку по вводу/выводу данных.
- Проще логика записи данных в файл.

Минусы:

- Из-за того что запись в файл идёт через один процесс у нас появляется «узкое горлышко», которое превращает код из параллельного в последовательный.

Cooperative Parallel I/O



Плюсы :

- Больше нет «узкого горлышка»

Минусы:

- Зависимость от специализированных библиотек по вводу/выводу данных.
- Приходится разбираться как именно будут происходить смещения при записи данных в файл для каждого процесса.

Результаты

Общий размер задачи 30 x 30 x 30

Размер Сетки <<< dim3(16, 16, 16), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
4.8481	1	1	1
9.52896	1	1	2
19.7348	1	2	2
6.23902	3	2	1
41.8693	2	2	2

Размер Сетки <<< dim3(8, 8, 8), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
2.33294	1	1	1
4.36011	1	1	2
9.46814	1	2	2
3.01707	3	2	1
21.4179	2	2	2

Размер Сетки <<< dim3(4, 4, 4), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
1.94265	1	1	1
3.60383	1	1	2
7.57626	1	2	2
2.55494	3	2	1
18.2521	2	2	2

Размер Сетки <<< dim3(2, 2, 2), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
2.00564	1	1	1
3.60328	1	1	2
7.48675	1	2	2
2.57147	3	2	1
18.0287	2	2	2

Размер Сетки <<< dim3(1, 1, 1), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
2.66088	1	1	1
4.26865	1	1	2
8.26073	1	2	2
2.91212	3	2	1
18.8379	2	2	2

Размер Сетки <<< dim3(1, 1, 1), dim3(8, 8, 8) >>>

Time	gridX	gridY	gridZ
2.76112	1	1	1
4.37526	1	1	2
8.20594	1	2	2
2.57614	3	2	1
18.2226	2	2	2

Размер Сетки <<< dim3(1, 1, 1), dim3(4, 4, 4) >>>

Time	gridX	gridY	gridZ
9.81771	1	1	1
11.5817	1	1	2
15.7241	1	2	2
4.11858	3	2	1
25.6839	2	2	2

Общий размер задачи 40 x 40 x 40

Размер Сетки <<< dim3(16, 16, 16), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
12.2892	1	1	1
22.4799	1	1	2
43.3202	1	2	2
77.3324	2	2	2

Размер Сетки <<< dim3(8, 8, 8), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
6.44637	1	1	1
10.7165	1	1	2
20.5984	1	2	2
40.4748	2	2	2

Размер Сетки <<< dim3(4, 4, 4), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
5.47565	1	1	1
8.63293	1	1	2
16.6117	1	2	2
33.9354	2	2	2

Размер Сетки <<< dim3(2, 2, 2), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
5.56286	1	1	1
8.36452	1	1	2
17.2477	1	2	2
33.7588	2	2	2

Размер Сетки <<< dim3(1, 1, 1), dim3(32, 4, 4) >>>

Time	gridX	gridY	gridZ
10.4391	1	1	1
13.3105	1	1	2
20.6579	1	2	2
38.4926	2	2	2

Размер Сетки <<< dim3(1, 1, 1), dim3(8, 8, 8) >>>

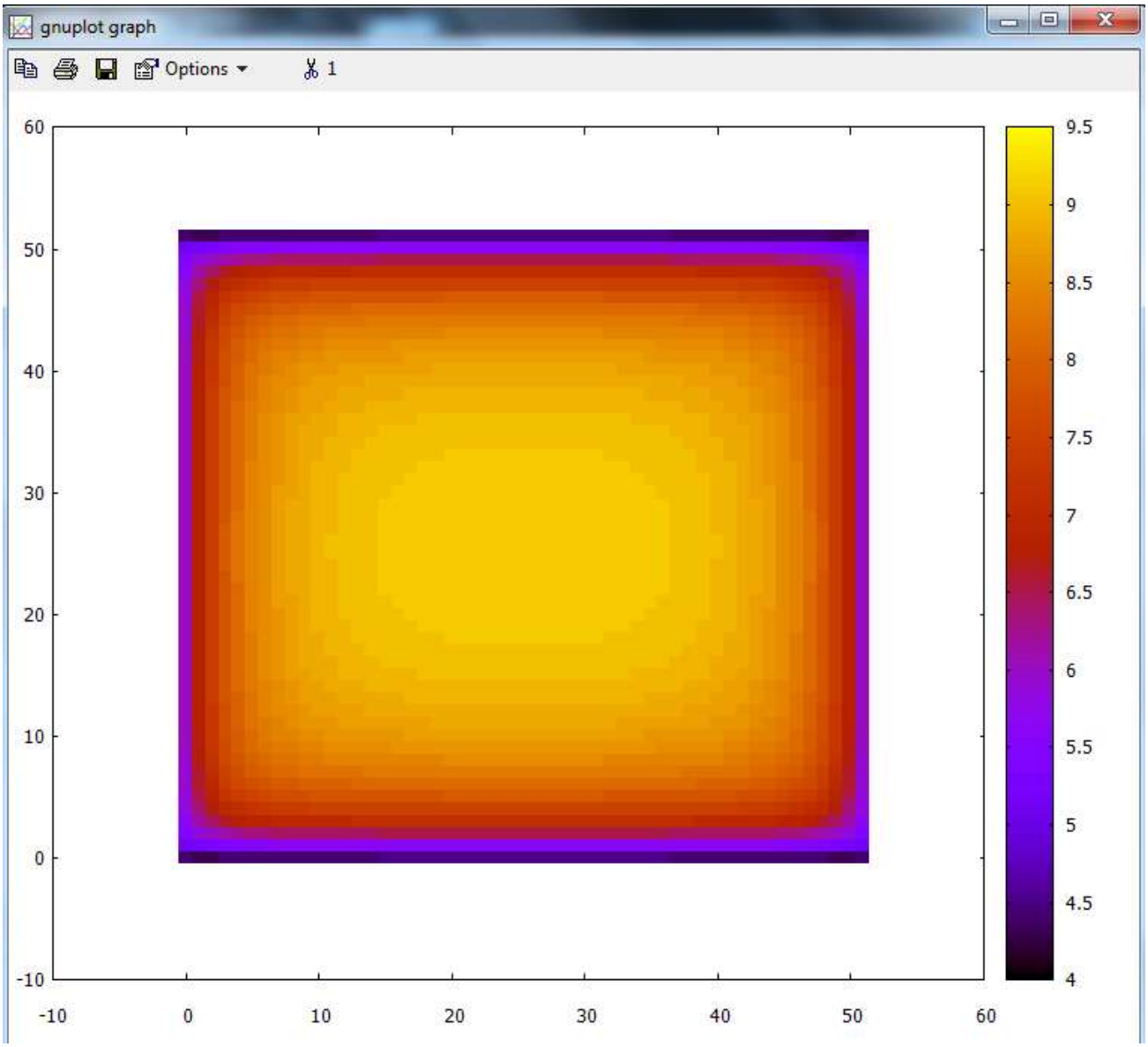
Time	gridX	gridY	gridZ
8.77947	1	1	1
11.8805	1	1	2
19.8014	1	2	2
38.8192	2	2	2

Размер Сетки <<< dim3(1, 1, 1), dim3(4, 4, 4) >>>

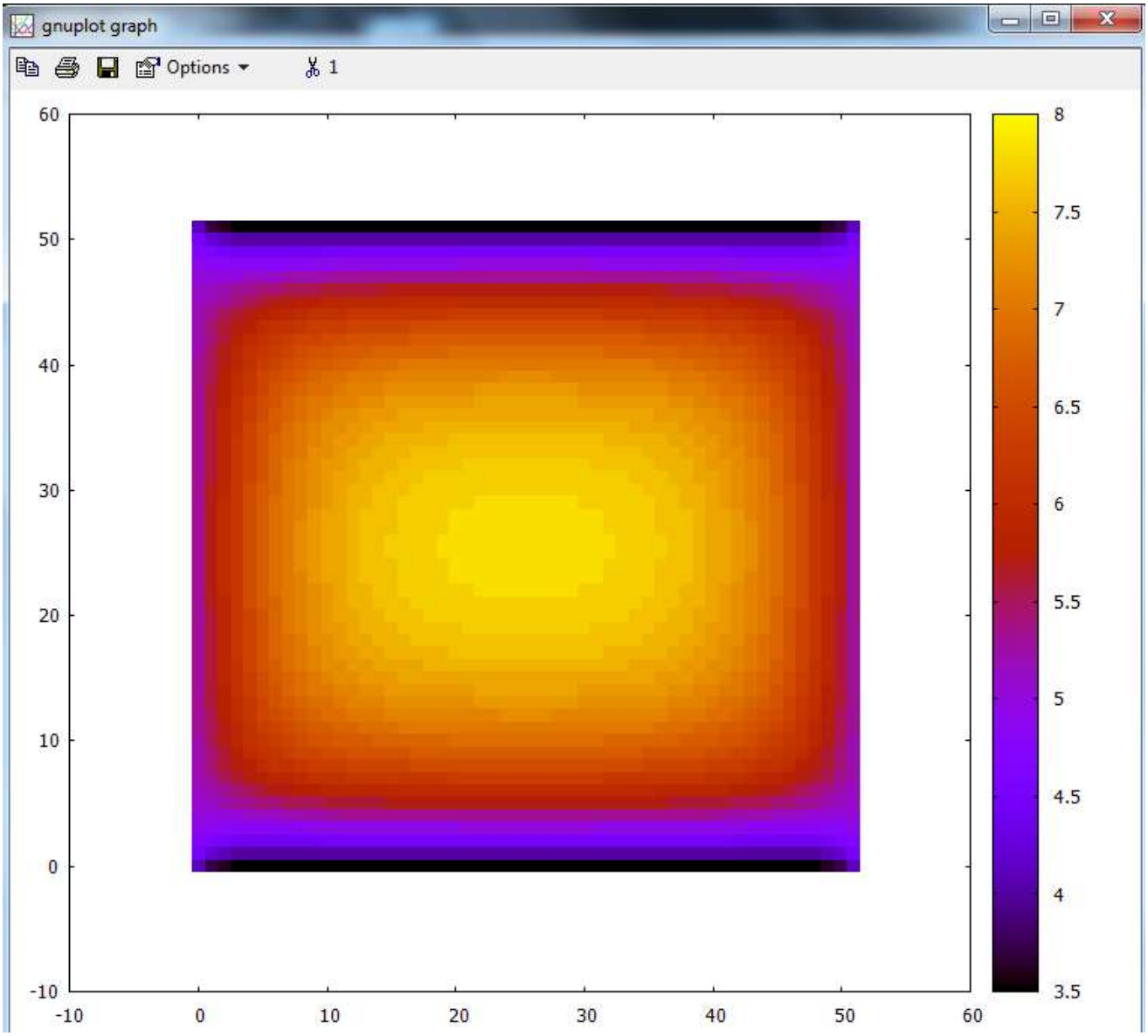
Time	gridX	gridY	gridZ
38.5474	1	1	1
40.7146	1	1	2
47.7539	1	2	2
63.8355	2	2	2

Температурный срез для задачи 52 x 52 x 52

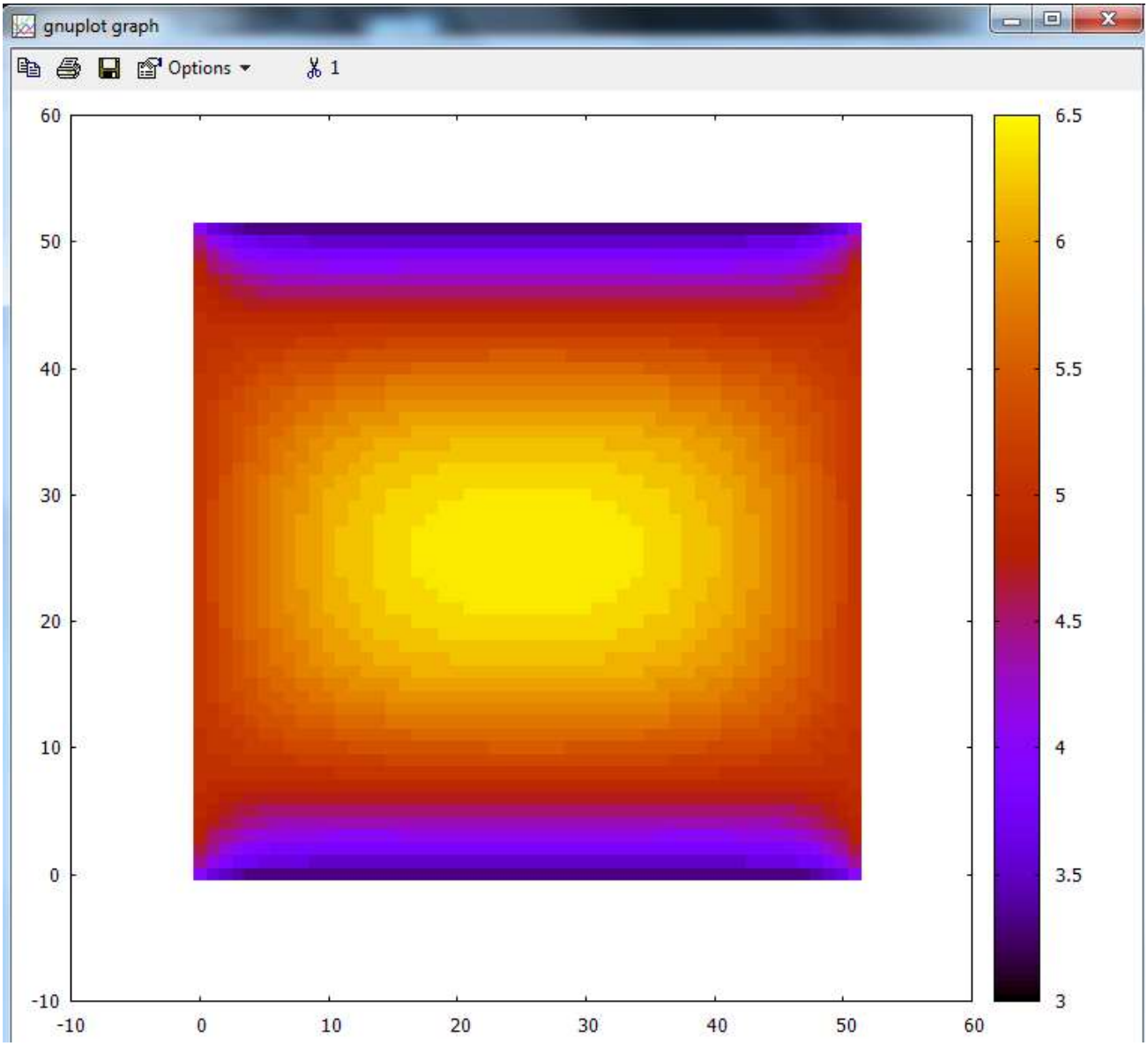
- Для z = 2



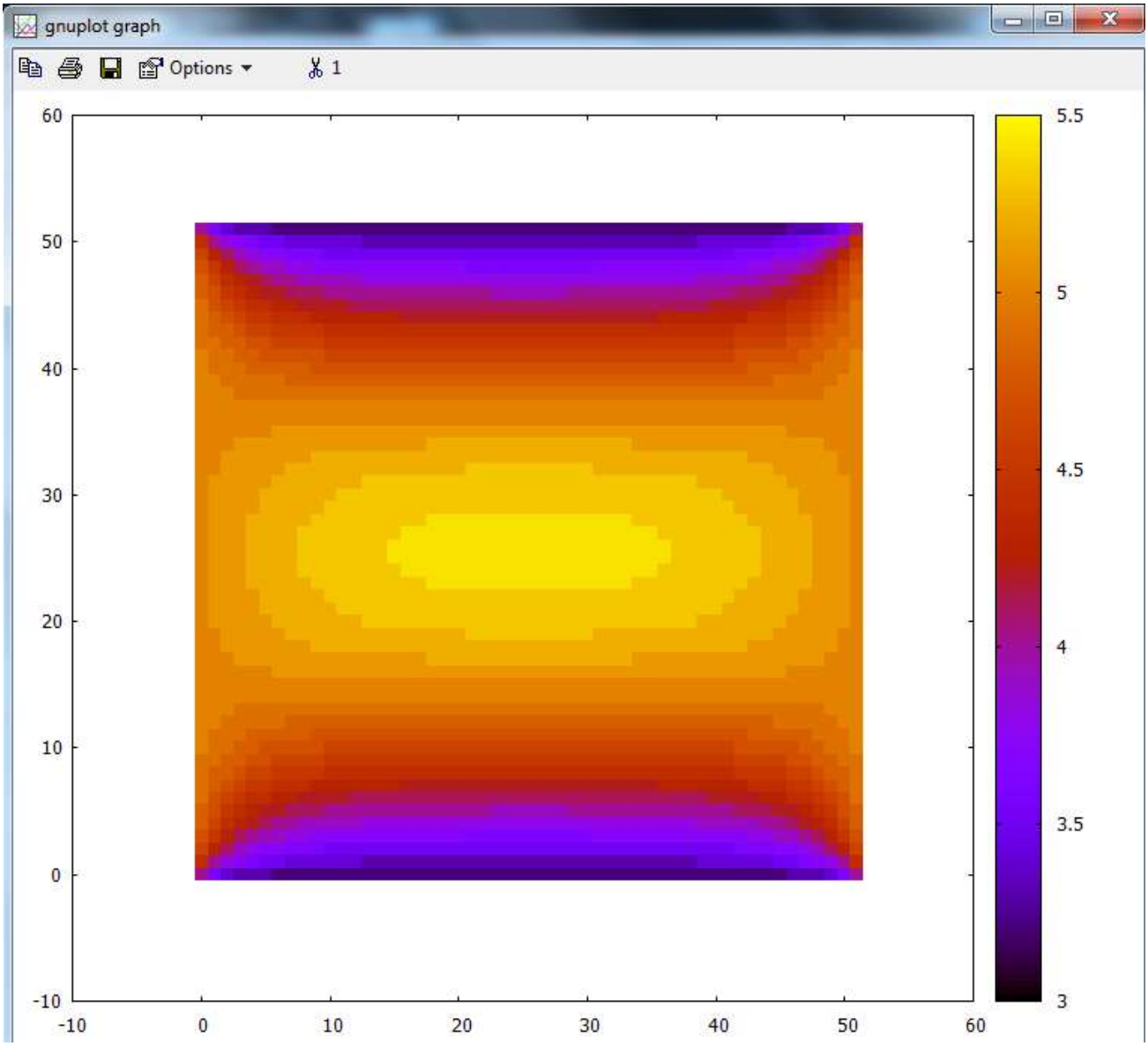
- Для z = 7


gnuplot graph
Options
1
60
50
40
30
20
10
0
-10
-10
0
10
20
30
40
50
60
8
7.5
7
6.5
6
5.5
5
4.5
4
3.5

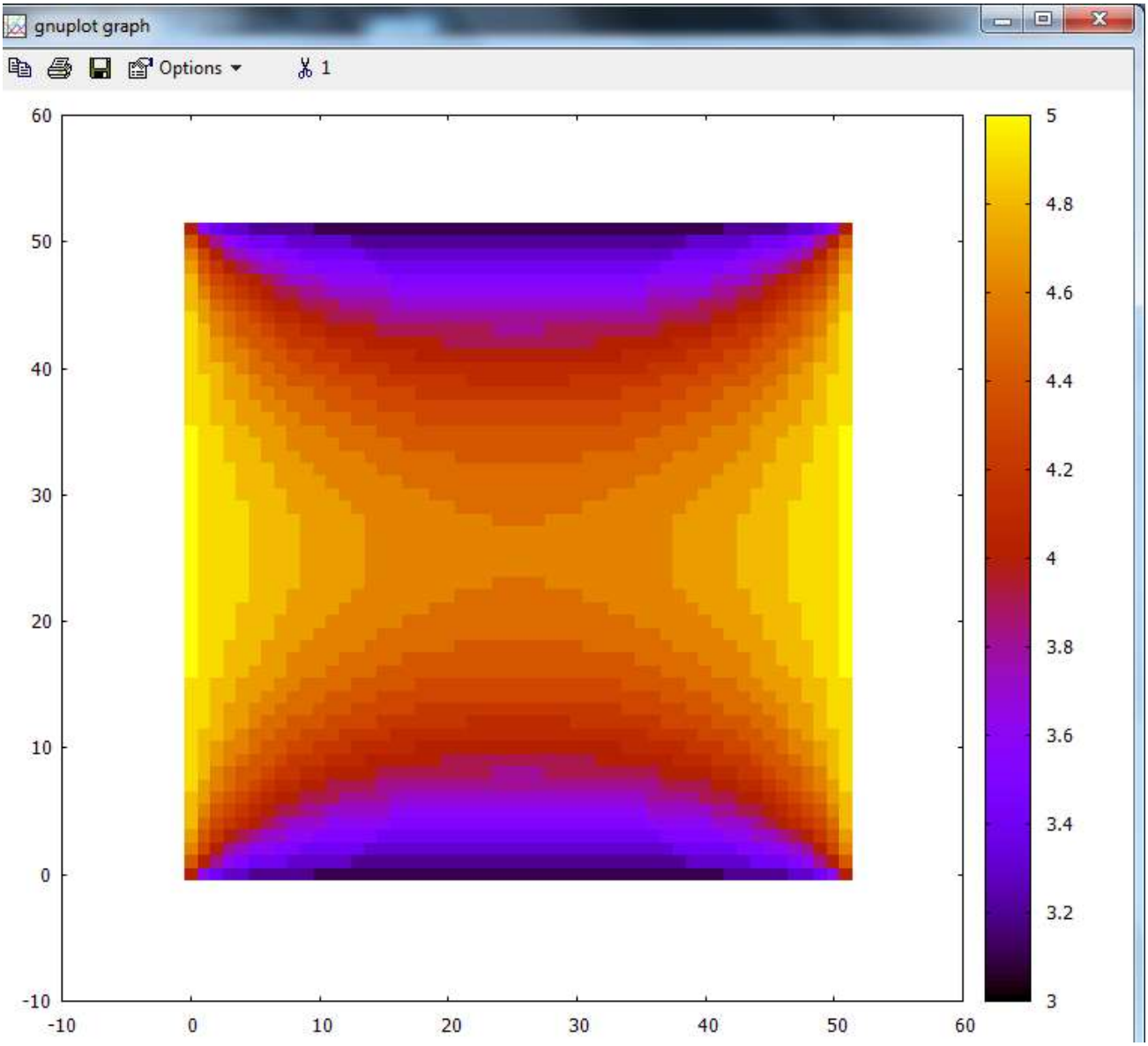
- Для z = 13


gnuplot graph
Options
1
60
50
40
30
20
10
0
-10
-10
0
10
20
30
40
50
60
6.5
6
5.5
5
4.5
4
3.5
3

- Для z = 19


gnuplot graph
Options
1
60
50
40
30
20
10
0
-10
-10
0
10
20
30
40
50
60
5.5
5
4.5
4
3.5
3

- Для z = 25


gnuplot graph
Options
1
60
50
40
30
20
10
0
-10
-10
0
10
20
30
40
50
60
5
4.8
4.6
4.4
4.2
4
3.8
3.6
3.4
3.2
3

Выводы

В данной лабораторной работе все основные расчеты выполняются на GPU. Несмотря на это, сильного выигрыша по времени не наблюдается. Сделав бенчмарки кода данной лабораторной работы с кодами прошлых двух выяснил, что прирост составил всего 12%. Данный код можно разогнать как минимум в четыре раза, если поменять логику хранения данных. Отказаться от идеи хранения большого количества лишних элементов. Возможно, ещё имеет смысл использовать текстурную память так как в формуле используются смежные элементы по всем трём координатам. Всё зависит от размера решаемой задачи.