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: И.Д. Черненко
Группа: 8О-406Б
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 задания: Вариант 1, MPI_Type_create_subarray

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 и l_z, граничные условия: u_{down} , u_{up} , u_{left} , u_{right} , u_{front} ,u_{back} начальное значение u_0.

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

Программное и аппаратное обеспечение

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

OS:Linux
compiler:nvcc
IDE:VS Code
Editor:nano

Метод решения

За основу данной лабораторной была взята предыдущая, что несколько упростило задачу, но при этом необходимо было использовать CUDA, MPI-IO и производные типы данных. Кроме того, необходимо было реализовать параллельную запись в файл и создать свой производный тип данных с помощью MPI_Type_create_subarray. В данном случае с помощью полученного типа данных была написана условная “сетка” для вывода результатов. Неочевидным оказалось удобство использования аргумента MPI_ORDER_FORTRAN для организации порядка массивов по столбцам (MPI_ORDER_C не был использован из-за того что в документации шёл за первым аргументом, кроме того разница заключается только в порядке хранения по столбцам или по строке соответственно).

Описание программы

Вся программа была реализована в одном файле **main.cu**.

Само ядро

```
__global__ void kernel(double* next_data, double* data, int n_x, int n_y, int
n_z, double h_x, double h_y, double h_z) {
    int id_x = blockIdx.x * blockDim.x + threadIdx.x;
    int id_y = blockIdx.y * blockDim.y + threadIdx.y;
    int id_z = blockIdx.z * blockDim.z + threadIdx.z;
    int offset_y = blockDim.y * gridDim.y;
    int offset_x = blockDim.x * gridDim.x;
    int offset_z = blockDim.z * gridDim.z;

    for (int i = id_x; i < n_x; i += offset_x) {
        for (int j = id_y; j < n_y; j += offset_y) {
            for (int k = id_z; k < n_z; k += offset_z) {
                double h_x_squared = h_x * h_x;
                double h_y_squared = h_y * h_y;
                double h_z_squared = h_z * h_z;

                double add1 = (data[i + 2 + (j + 1) * (n_x + 2) + (k + 1) *
(n_x + 2) * (n_y + 2)] + data[i + (j + 1) * (n_x + 2) + (k + 1) * (n_x + 2) *
(n_y + 2)]) / h_x_squared;
                double add2 = (data[i + 1 + (j + 2) * (n_x + 2) + (k + 1) *
(n_x + 2) * (n_y + 2)] + data[i + 1 + j * (n_x + 2) + (k + 1) * (n_x + 2) *
(n_y + 2)]) / h_y_squared;
                double add3 = (data[i + 1 + (j + 1) * (n_x + 2) + (k + 2) *
(n_x + 2) * (n_y + 2)] + data[i + 1 + (j + 1) * (n_x + 2) + k * (n_x + 2) *
(n_y + 2)]) / h_z_squared;
                double devider = 2 * (1.0 / h_x_squared + 1.0 / h_y_squared +
1.0 / h_z_squared);

                int tmp = i + 1 + (j + 1) * (n_x + 2) + (k + 1) * (n_x + 2) *
(n_y + 2);
                next_data[tmp] = (add1 + add2 + add3) / devider;
            }
```

```
        }

    }
}
```

Вспомогательные функции

```
__global__ void get_error(double* next_data, double* data, int n_x, int n_y,
int n_z) {
    int id_x = blockIdx.x * blockDim.x + threadIdx.x;
    int id_y = blockIdx.y * blockDim.y + threadIdx.y;
    int id_z = blockIdx.z * blockDim.z + threadIdx.z;
    int offset_y = blockDim.y * gridDim.y;
    int offset_x = blockDim.x * gridDim.x;
    int offset_z = blockDim.z * gridDim.z;

    for (int i = id_x - 1; i <= n_x; i += offset_x) {
        for (int j = id_y - 1; j <= n_y; j += offset_y) {
            for (int k = id_z - 1; k <= n_z; k += offset_z) {
                 if (i == -1 || j == -1 || k == -1 || i == n_x || j == n_y ||
k == n_z) {
                    data[_i(i, j, k)] = 0;
                } else {
                    data[_i(i, j, k)] = fabs(next_data[_i(i, j, k)] -
data[_i(i, j, k)]);
                }
            }
        }
    }
}

__global__ void get_copy(double* edge, double* data, int n_x, int n_y, int
n_z, int index, int device_to_host, double u, int type) {
    int id_x = blockIdx.x * blockDim.x + threadIdx.x;
    int id_y = blockIdx.y * blockDim.y + threadIdx.y;
    int offset_y = blockDim.y * gridDim.y;
    int offset_x = blockDim.x * gridDim.x;

    if (type == 0) {
        if (device_to_host) {
            for (int k = id_y; k < n_z; k += offset_y) {
                for (int j = id_x; j < n_y; j += offset_x)
                    edge[j + n_y * k] = data[_i(index, j, k)];
            }
        } else  {
            if (edge) {
                for (int k = id_y; k < n_z; k += offset_y) {
                    for (int j = id_x; j < n_y; j += offset_x) {
                        data[_i(index, j, k)] = edge[j + n_y * k];
                    }
```

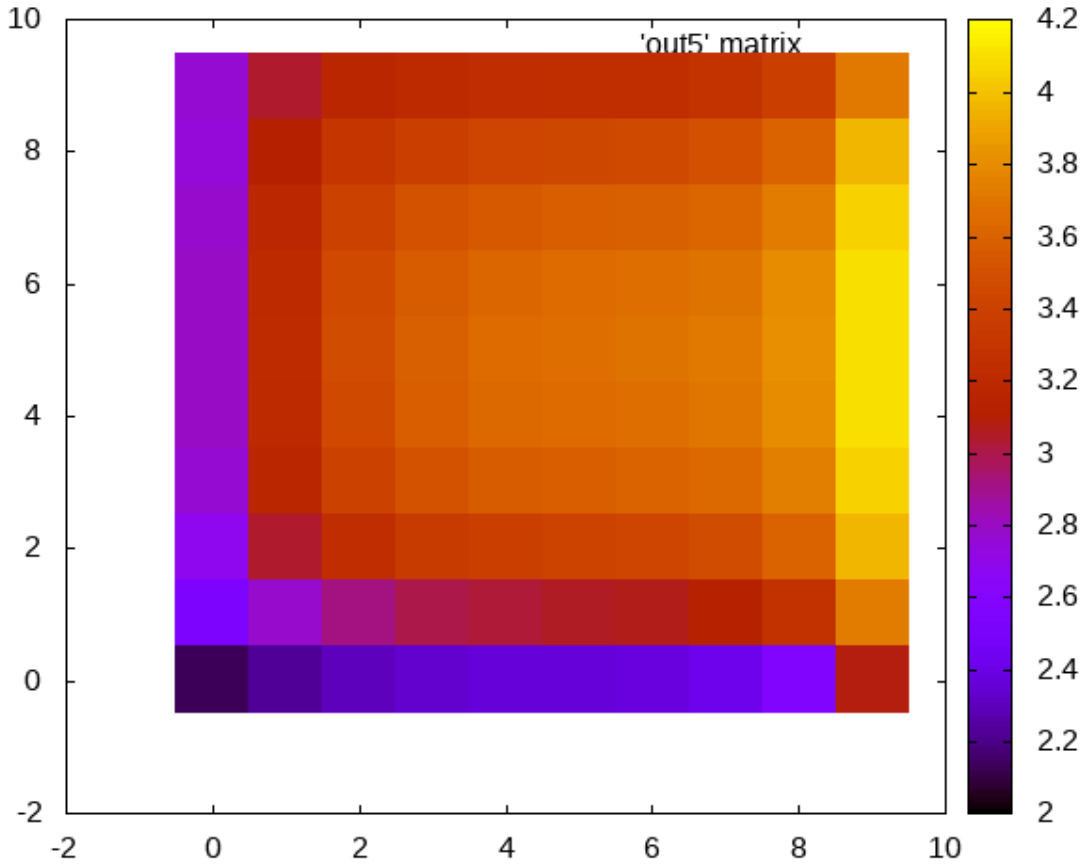
```
                }
            } else {
                for (int k = id_y; k < n_z; k += offset_y) {
                    for (int j = id_x; j < n_y; j += offset_x) {
                        data[_i(index, j, k)] = u;
                    }

                }
            }
        }
    } else if(type == 1) {
        if (device_to_host) {
            for (int k = id_y; k < n_z; k += offset_y) {
                for (int i = id_x; i < n_x; i += offset_x) {
                    edge[i + n_x * k] = data[_i(i, index, k)];
                }
            }
        } else {
            if (edge) {
                for (int k = id_y; k < n_z; k += offset_y) {
                    for (int i = id_x; i < n_x; i += offset_x) {
                        data[_i(i, index, k)] = edge[i + n_x * k];
                    }
                }
            } else {
                for (int k = id_y; k < n_z; k += offset_y) {
                    for (int i = id_x; i < n_x; i += offset_x) {
                        data[_i(i, index, k)] = u;
                    }
                }
            }
        }
    } else {
        if (device_to_host) {
            for (int j = id_y; j < n_y; j += offset_y) {
                for (int i = id_x; i < n_x; i += offset_x) {
                    edge[i + n_x * j] = data[_i(i, j, index)];
                }
            }
        } else {
            if (edge) {
                for (int j = id_y; j < n_y; j += offset_y) {
                    for (int i = id_x; i < n_x; i += offset_x) {
                        data[_i(i, j, index)] = edge[i + n_x * j];
                    }
                }
            } else {
                for (int j = id_y; j < n_y; j += offset_y) {
```

```
                    for (int i = id_x; i < n_x; i += offset_x) {
                        data[_i(i, j, index)] = u;
                    }
                }
            }
        }
    }
}
```

Результаты

net	block	mpi_cuda	mpi	cpu
1,1,1	10 10 10	166.618ms	43.646ms	16.698ms
1,1,1	20 20 20	779.714ms	498.166ms	248.949ms
2,2,2	10 10 10	627.18ms	164.939ms	254.609ms
2,2,2	20 20 20	2741ms	2361.61ms	7269.34ms
4,4,1	10 10 10	3224.13ms	904.25ms	1247.11ms
4,4,1	20 20 20	18634.7ms	14074ms	35821.9ms

Полученные изображения температуры в заданной области:
epsilon = 10e-1



Из полученных данных можно сделать вывод, что при правильных настройках вычисления с помощью **MPI_CUDA** превосходят **CPU,** но немного уступает отдельно MPI вероятно из-за высокой стоимости содержания CUDA.

Выводы

Данную лабораторную всё также можно использовать для решения задач математической физики. При реализации были получены навыки работы с тандемом CUDA и MPI, а также с созданием производных типов данных в MPI. Основные сложности как всегда вызвала работа с индексами и выводом данных в нужном формате, но все сложности были преодолены и требуемый алгоритм был реализован.