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: А. О. Дубинин
Группа: 8О-407Б
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.

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

Вариант 1. Конструктор типа MPI_Type_create_subarray

Программное и аппаратное обеспечение

GeForce 940MX

Compute capability:	5.0
Dedicated video memory:	4096 MB
shared memory per block:	49152 bytes
constant memory:	65536 bytes
Total number of registers available per block:	65536
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
(3) Multiprocessors, (128) CUDA Cores/MP:	384 CUDA Cores

Intel(R) Core (TM) i5-7200U CPU @ 2.50GHz

Architecture:	x86_64
Byte Order:	Little Endian
CPU(s):	4
Thread(s) per core:	2
Core(s) per socket:	2
CPU MHz: CPU max MHz: CPU min MHz:	713.848 3100,0000 400,0000
L1d cache: L1i cache: L2 cache: L3 cache:	64 KiB 64 KiB 512 KiB 3 MiB

RAM	8GiB SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz (0,4 ns)

SSD(SPCC_M.2_SSD)	223,6G
HDD(ST1000LM035-1RK172)	931,5G

OS: Ubuntu 20.04 focal
IDE: jetbrains clion
compiler: nvcc

Метод решения

Основная логика решения данной ЛР, была взята с лабораторной работы №7. Главные сложности были перенести сложные по скорости вычисления for'ов на gpu и организовать запись в файл с помощью mpi io.

Для копирования значений и подсчета были написаны несколько ядер. Было написано 3 ядра для копирования и инициализации данных. Эти ядра отличались лишь поверхностью, через которую происходит копирование. Так же были написаны два ядра для вычисления основного цикла и вычисления ошибки. С помощью thrust находилась максимальная погрешность(ошибка), которая была в одной mpi node. Сложность была не запутаться в индексах, так 3мерная индексация требует определенных усилий для восприятия, особенно в контексте дополнительных индексов от cuda.

Параллельная запись в файл была рассмотрена подробно на лекции, оставалось лишь разобраться с вариантом (MPI_Type_create_subarray). После осознания того, что с помощью MPI_Type_create_subarray можно создавать сетку для вывода данных, я смог написать многопроцессорную запись в файл.

Описание программы

Ядро для копирования и инициализации было подсмотрено на лекции. Для одной стороны использовалось одно ядро, для принятия данных (т.е. перекопированния из буфера в массив), для отправки (копирования из массива в буфер) и для граничной инициализации.

```
__global__ void kernel_copy_yz(double *plane_yz, double *data, int nx, int ny, int
nz, int i, int dir, int bc) {
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int offsety = blockDim.y * gridDim.y;
    int offsetx = blockDim.x * gridDim.x;
    int j, k;
    if (dir) {
        for (k = idy; k < nz; k += offsety)
```

```
            for (j = idx; j < ny; j += offsetx)
                plane_yz[_iyz(j, k)] = data[_i(i, j, k)];
    } else {
        if (plane_yz) {
            for (k = idy; k < nz; k += offsety)
                for (j = idx; j < ny; j += offsetx)
                    data[_i(i, j, k)] = plane_yz[_iyz(j, k)];
        } else {
            for (k = idy; k < nz; k += offsety)
                for (j = idx; j < ny; j += offsetx)
                    data[_i(i, j, k)] = bc;
        }
    }
}
```

Вычисления значений сетки было интересно только из-за 3х мерной сетки потоков, когда сами вычисления тривиальны.

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

Так же было написать, как использовать функцию из библиотеки thrust. Как мы видим Интересно было, что для вычисления максимального значения потребовалось закастить указатель gpu памяти в указатель thrust'a.

```
double error = 0.0;
thrust::device_ptr<double> p_arr = thrust::device_pointer_cast(dev_data);
thrust::device_ptr<double> res = thrust::max_element(p_arr, p_arr + _size_b);
error = *res;
```

Для многопроцессорной записи в файл сначала был создан тип данных с помощью которого происходила запись одной ячейки, т.е. одного значения.

```
MPI_Datatype cell;
MPI_Type_contiguous(n_size, MPI_CHAR, &cell);
MPI_Type_commit(&cell);
```

Далее был создан подмассив, который расчерчивал данные из одной mpi node.

```
MPI_Datatype subarray;
int subarray_starts[3] = {0, 0, 0};
int subarray_subsizes[3] = {nx, ny, nz};
int subarray_bigsizes[3] = {nx, ny, nz};
MPI_Type_create_subarray(3, subarray_bigsizes, subarray_subsizes,
subarray_starts, MPI_ORDER_FORTRAN, cell, &subarray); // memtype
MPI_Type_commit(&subarray);
```

Потом мы расчерчивали глобальную сетку, чтобы мы смогли вписать наш подмассив из ноды в нужное место.

```
MPI_Datatype bigarray;
int bigarray_starts[3] = {ib * nx, jb * ny, kb * nz};
int bigarray_subsizes[3] = {nx, ny, nz};
int bigarray_bigsizes[3] = {nx * nbx, ny * nby, nz * nbz};
MPI_Type_create_subarray(3, bigarray_bigsizes, bigarray_subsizes,
bigarray_starts, MPI_ORDER_FORTRAN, cell, &bigarray); // memtype
MPI_Type_commit(&bigarray);
```

В конце открывали файл на запись и с помощью хитрых махинаций записывали данные из mpi node.

```
MPI_File fp;
MPI_File_delete(file_name, MPI_INFO_NULL);
MPI_File_open(MPI_COMM_WORLD, file_name, MPI_MODE_CREATE |
MPI_MODE_WRONLY, MPI_INFO_NULL, &fp);


MPI_File_set_view(fp, 0, MPI_CHAR, bigarray, "native", MPI_INFO_NULL);
MPI_File_write_all(fp, out_buff, 1, subarray, MPI_STATUS_IGNORE);
MPI_File_close(&fp);
```

Результаты

В данной лабораторной работе мы можем сравнить с результатом программы из 7 Лр и 9 Л.

1.

	MPI	MPI + OpenMP	Mpi + CUDA
2 2 2, 20 20 20	5006.32ms	4134.12ms	4566.45ms
2 2 4, 20 20 10	5894.24ms	4323.8ms	4898.81ms

Как мы видим данные результаты показывают, что MPI с CUDA работает не так быстро в сравнении с openmp, быть может, потому что код на openmp оптимизирован намного лучше кода, который был реализован мной.

Вывод

Данную лабораторную было приятно реализовывать особенно из-за параллельной записи в файл, ведь раннее я никогда не задумывался об этой функции. Особенно интересно было, когда я узнал, как это красиво можно реализовать через MPI_Type_create_subarray. Работа с cuda была уже привычна, только лишь очень было легко ошибиться со всей сложностью индексов, что пришлось отлаживать ЛР, постоянно перекопирую данные обратно на CPU.