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»

Технология MPI и технология CUDA. MPI-IO

Выполнил: Полей-Добронравова Амелия
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников, А.Ю. Морозов

Москва, 2022

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.

Все **входные-выходные данные** и **варианты заданий по межпроцессорному взаимодействию** совпадают с входными-выходными данными и вариантами заданий из лабораторной работы №7.

Запись результатов в файл должна осуществляться параллельно всеми процессами. Необходимо создать производный тип данных, определяющий шаблон записи данных в файл. **Варианты** конструкторов типов:

1. MPI_Type_create_subarray
2. MPI_Type_hvector
3. MPI_Type_hindexed

Допускается двойное использование графической памяти относительно размера блока обрабатываемого одним процессом.

Вариант 1.

Программное и аппаратное обеспечение

Компилятор nvcc версии 7.0(g++ версии 4.8.4) на 64-х битной Ubuntu 14.04 LTS.
Параметры графического процессора:
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

Метод решения

Часть решения базируется на 7 ЛР. Отличающиеся моменты:

Копирование граничных значений блоков сетки в буферы для обмена осуществляется с помощью ядер GPU. Максимальную погрешность в одном процессе я ищу автоматически с помощью библиотеки Thrust, далее для проверки по всем процессам собираю их с помощью MPI_Allgather.

Каждый процесс записывает свой итоговый результат в общий файл с результатом работы программы. Перед записью в выходной файл формируется буферный массив с выходными данными, далее с помощью возможностей MPI описывается способ парсинга из данного буфера в выходной файл.

Описание программы

Макросы с именами **next_(индексы)** - макросы для итерации по координатам буферов обмена во время параллельной обработки на GPU.
idx - макрос определения индекса записи в буфер для последующей записи в файл.
import_(координата) - функции GPU параллельного копирования граничных значений блоков сетки в буферы для обмена.
export_(координата) - функции GPU параллельного копирования граничных значений блоков сетки из буферов обмена.
new_grid(...) - функция GPU параллельного пересчета значений после получения новых границ.

Результаты

Имеет смысл сравнивать результат с вариантом в 7ЛР, чтобы доказать эффективность совместного использования MPI и CUDA.

Размер общей сетки тестов ниже не меняется.

Сетка / Расчет	Результат на MPI+CUDA, мс	Результат на MPI, мс	на CPU, мс
1 1 1 / 40 40 40	5680	19861	9843
2 2 2 / 20 20 20	38510	4943	9952
2 2 4 / 20 20 10	97931	6224	9890

Время работы стало хуже даже чем на CPU.

Код программы для CPU:

```
#include <iostream>
#include <string>
#include <fstream>
#include <algorithm>
#include <stdlib.h>
#include <iomanip>
#include <string.h>
#include <chrono>

using namespace std;
using namespace std::chrono;
```

```
const int ndims = 3;
const int ndims_x_2 = 6;

double u_next(double ux0, double ux1,
double uy0, double uy1, double uz0,
double uz1, double h2x, double h2y,
double h2z){
    double ans = (ux0 + ux1) * h2x;
    ans += (uy0 + uy1) * h2y;
    ans += (uz0 + uz1) * h2z;
    return ans;
}

double    max_determine(double    val1,
double val2, double curr_max){
    double diff = val1 - val2;
    diff = diff < 0.0 ? -diff : diff;

     return diff > curr_max ? diff :
curr_max;
}

void  print_line(ostream&  os,  double*
line, int size){
    for(int i = 0; i < size; ++i){
        os << line[i] << " ";
    }
}


int main(int argc, char **argv){

std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int dimens[ndims], blocks[ndims];
    double l[ndims];
    double u[ndims_x_2];
    double u0, eps;
    string path;

    enum orientation{
        left = 0, right,
        front, back,
        down, up,
    };
    enum direction{
        dir_x = 0,
        dir_y,
        dir_z
    };

          cin   >>   dimens[dir_x]   >>
dimens[dir_y] >> dimens[dir_z];
          cin   >>   blocks[dir_x]   >>
blocks[dir_y] >> blocks[dir_z];
    cin >> path;
    cin >> eps;
      cin  >> l[dir_x] >> l[dir_y] >>
l[dir_z];
    cin >> u[down] >> u[up];
    cin >> u[left] >> u[right];
    cin >> u[front] >> u[back];
    cin >> u0;

    auto start = steady_clock::now();

    double max_diff = 0.0;

                        int     nx     =
dimens[dir_x]*blocks[dir_x];
                        int     ny     =
dimens[dir_y]*blocks[dir_y];
                        int     nz     =
dimens[dir_z]*blocks[dir_z];

    int sizex = nx + 2;
    int sizey = ny + 2;
    int sizez = nz + 2;

    double h2x, h2y, h2z;
    h2x = l[dir_x] / ((double)nx);
    h2y = l[dir_y] / ((double)ny);
    h2z = l[dir_z] / ((double)nz);

    h2x *= h2x;
    h2y *= h2y;
    h2z *= h2z;

    {
                double  denuminator  =
2.0*(1.0/h2x + 1.0/h2y + 1.0/h2z);
          h2x = 1.0 / (denuminator *
h2x);
          h2y = 1.0 / (denuminator *
h2y);
          h2z = 1.0 / (denuminator *
h2z);
    }

     double* buffer0 = new double[sizex
* sizey * sizez];
    double* buffer1 = new double[sizex
* sizey * sizez];

      fill_n(buffer0, sizex * sizey *
sizez, u0);

    int orr = 0;
     for(int i = 0; i < sizex; i += nx
+ 1, ++orr){
          for(int j = 1; j < ny + 1;
++j){
            for(int k = 1; k < nz + 1;
++k){
                        buffer0[i + (j +
k*sizey)*sizex] = u[orr];
                        buffer1[i + (j +
k*sizey)*sizex] = u[orr];
```

```
            }
        }
    }

    for(int j = 0; j < sizey; j += ny
+ 1, ++orr){
          for(int k = 1; k < nz + 1;
++k){
            for(int i = 1; i < nx + 1;
++i){
                      buffer0[i + (j +
k*sizey)*sizex] = u[orr];
                      buffer1[i + (j +
k*sizey)*sizex] = u[orr];
            }
        }
    }

    for(int k = 0; k < sizez; k += nz
+ 1, ++orr){
          for(int j = 1; j < ny + 1;
++j){
            for(int i = 1; i < nx + 1;
++i){
                      buffer0[i + (j +
k*sizey)*sizex] = u[orr];
                      buffer1[i + (j +
k*sizey)*sizex] = u[orr];
            }
        }
    }

    do{
        max_diff = 0.0;
        for(int k = 1; k <= nz; ++k){
              for(int j = 1; j <= ny;
++j){
                  for(int i = 1; i <=
nx; ++i){
                      buffer1[i + (j +
k*sizey)*sizex] = u_next(buffer0[i - 1
+ (j + k*sizey)*sizex], buffer0[i + 1
+ (j + k*sizey)*sizex], buffer0[i + (j
- 1 + k*sizey)*sizex], buffer0[i + (j
+ 1 + k*sizey)*sizex], buffer0[i + (j
+ k*sizey - sizey)*sizex], buffer0[i +
(j  +  k*sizey  +  sizey)*sizex],  h2x,
h2y, h2z);
                                max_diff =
max_determine(buffer0[i    +    (j    +
k*sizey)*sizex],    buffer1[i  +  (j  +
k*sizey)*sizex], max_diff);
                }
            }
        }

        double* temp = buffer0;
        buffer0 = buffer1;
        buffer1 = temp;
    }while(max_diff >= eps);

    ofstream fout(path, ios::out);
         fout  <<  std::scientific  <<
std::setprecision(7);

    for(int k = 1; k <= nz; ++k){
        for(int j = 1; j <= ny; ++j){
              for(int i = 1; i <= nx;
++i){
                 fout << buffer0[i + (j
+ k*sizey)*sizex] << " ";
            }
        }
    }
    fout << endl;
    delete[] buffer0;
    delete[] buffer1;
    fout.close();
    auto end = steady_clock::now();

    cout << "Inference time: ";
                            cout      <<
((double)duration_cast<microseconds>(e
nd - start).count()) / 1000.0 << "ms"
<< endl;
    return 0;
}
```

Выводы

1. Возможность записывать результат работы из нескольких процессов в один файл, избегая пересылки в один (главный) процесс для записи, сокращает время работы программы, но усложняет ее написание.
2. Замедление работы программы можно объяснить тем, что для использования CUDA приходится постоянно перемещать память с хоста на видеокарту, что является долгим процессом. Лишнее подтверждение того, что не стоит использовать вычисления на видеокарте, при которых требуется частый обмен данными.