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.В. Ефимов
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

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: MPI_Type_hindexed

Программное и аппаратное обеспечение

```
##### CUDA Info

/opt/cuda/extras/demo_suite/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce MX150"
  CUDA Driver Version / Runtime Version          11.4 / 11.4
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 2003 MBytes (2099904512
bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1532 MHz (1.53 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072,
65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048
layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >
```

```
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA Runtime
Version = 11.4, NumDevs = 1, Device0 = NVIDIA GeForce MX150
Result = PASS

##### CPU Info

Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Address sizes:                   39 bits physical, 48 bits virtual
Byte Order:                      Little Endian
CPU(s):                          8
On-line CPU(s) list:             0-7
Vendor ID:                       GenuineIntel
Model name:                      Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU family:                      6
Model:                           142
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
Stepping:                        10
CPU max MHz:                     3400.0000
CPU min MHz:                     400.0000
BogoMIPS:                        3601.00
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64
monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2
x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
abm 3dnowprefetch cpuid_fault epb invpcid_single pti ibrs ibpb stibp
tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2
smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt
xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window
hwp_epp
Virtualization:                  VT-x
L1d cache:                       128 KiB (4 instances)
L1i cache:                       128 KiB (4 instances)
L2 cache:                        1 MiB (4 instances)
L3 cache:                        6 MiB (1 instance)
NUMA node(s):                    1
NUMA node0 CPU(s):               0-7
Vulnerability Itlb multihit:     KVM: Mitigation: VMX disabled
Vulnerability L1tf:             Mitigation; PTE Inversion; VMX conditional
cache flushes, SMT vulnerable
Vulnerability Mds:               Vulnerable: Clear CPU buffers attempted,
no microcode; SMT vulnerable
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and
__user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, IBPB
conditional, IBRS_FW, STIBP conditional, RSB filling
Vulnerability Srbds:             Vulnerable: No microcode
Vulnerability Tsx async abort:   Not affected

##### RAM Info
               total          used          free        shared  buff/cache
available
Mem:            7.6Gi         1.7Gi         4.6Gi         633Mi         1.4Gi
5.1Gi
```

```
Swap:              0B          0B          0B
```

Метод решения

Лабораторная является надстройкой над предыдущей в том, что:

1. Весь алгоритм подсчета выполняется на GPU;
2. Вывод в файл выполняется через MPI-IO.

Первое решается двумя этапами. Для начала нужно переписать все циклы копирования буфферных данных, одно свое ядро для каждой стороны, а в случае с граничными условиями – отдельные ядра для инициализации. Второй этап – это подсчет новой сетки и погрешности. Это достигается в двух ядрах – один для подсчета новой сетки, по сути достаточно знаний предыдущих лабораторных. Другое ядро подсчитывает погрешности следующим методом: если это граничное значение, то в нем ставится ноль, иначе высчитывается модуль разницы. Максимальный модуль разницы ищется через Thrust.

Вторая задача сначала копированием всех данных в отдельный буффер символов (в символьный формат), затем заданием на файле такого вида (view), что при выводе всего буффера все данные окажутся на корректных местах.

Для этого нужно задать два типа данных – один элементарный тип для единственной клетки, в которой содержится одно выводимое число. Второй тип создается через $Indexed$ и берет за основу прошлый тип. Рассматривая каждую строку по X за единый блок, длина всех блоков одинакова и равна длине сетки по X. Далее идет подсчет сдвига каждого блока начиная с нулевого y и z. Тогда формула сдвига i-ого по Y и j-ого по Z блока: $i * line + j * face$, где $line$ – длина блока, а $face$ – длина блока, умноженная на количество блоков по Y (фактически, размер стороны XY).

Так как глобальный сдвиг одинаковый для каждого локального блока, его можно посчитать один раз по той же формуле (добавив сдвиг по X) и добавить в обзор на файл.

Описание программы

Подсчет погрешности (результат записывается в старую сетку чтобы не использовать тройной буффер).

```
__global__
void calc_error(double *new_grid, double *old_grid,
                int    block_x, int    block_y, int    block_z) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int idz = blockIdx.z * blockDim.z + threadIdx.z;
    int stridex = blockDim.x * gridDim.x;
    int stridey = blockDim.y * gridDim.y;
    int stridez = blockDim.z * gridDim.z;
```

```
    for (int z = idz - 1; z < block_z + 1; z += stridez) {
        for (int y = idy - 1; y < block_y + 1; y += stridey) {
            for (int x = idx - 1; x < block_x + 1; x += stridex) {
                bool is_edge_elem = (x == -1      || y == -1      || z ==
-1 ||
                                     x == block_x || y == block_y || z ==
block_z);
                size_t i = _i(x, y, z);
                old_grid[i] = (!is_edge_elem) * fabs(new_grid[i] -
old_grid[i]);
            }
        }
    }
}
```

Поиск максимального элементы во всех блоках:

```
thrust::device_ptr<double> p_old_grid =
thrust::device_pointer_cast(d_old_grid);
thrust::device_ptr<double> max_diff_ptr =
thrust::max_element(p_old_grid, p_old_grid + elem_count);
double max_diff = *max_diff_ptr;

MPI_Allgather(&max_diff, 1, MPI_DOUBLE, block_maxes, 1,
MPI_DOUBLE, MPI_COMM_WORLD);
for (int i = 0; i < proc_count; ++i) {
      max_diff = std::max(max_diff, block_maxes[i]);
}

if (max_diff < eps) {
      break;
}
```

Создание типа, используемый при задании вида на файл:

```
size_t x_line_count = block_size[y_dir] * block_size[z_dir];
int *lengths   = new int[x_line_count];
int *disp = new int[x_line_count];
for (size_t i = 0; i < x_line_count; ++i) {
    lengths[i] = block_size[x_dir];
}

const int line_size = block_size[x_dir] * proc_size[x_dir];
const int face_size = line_size * block_size[y_dir] *
proc_size[y_dir];

const int x_first_i = block_size[x_dir] * proc_x;
const int y_first_i = block_size[y_dir] * proc_y;
const int z_first_i = block_size[z_dir] * proc_z;

MPI_Aint global_offset = face_size * z_first_i + line_size *
y_first_i + x_first_i;
for (int z = 0; z < block_size[z_dir]; ++z) {
    for (int y = 0; y < block_size[y_dir]; ++y) {
        int i = z * block_size[y_dir] + y;
```

```
            disp[i] = y * line_size + z * face_size;
        }
    }

    MPI_Type_indexed(x_line_count, lengths, disp, cell,
    &something_complex_idk);
    MPI_Type_commit(&something_complex_idk);
```

Само задание вида (сдвиг задается в байтах, поэтому умножается на размер ячейки):

```
MPI_File_set_view(fp, global_offset * n_size, cell,
something_complex_idk, "native", MPI_INFO_NULL);
```

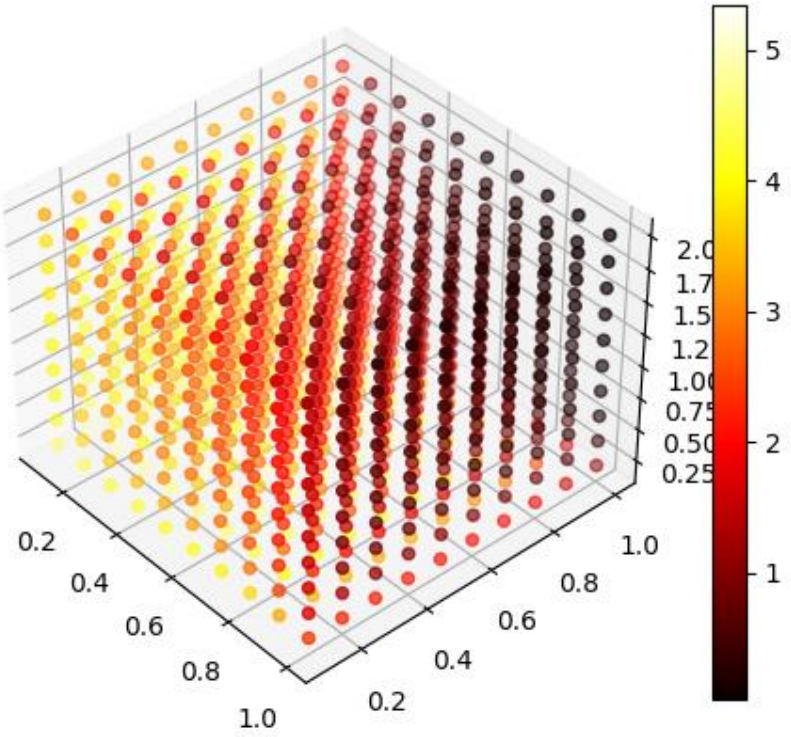
Результаты

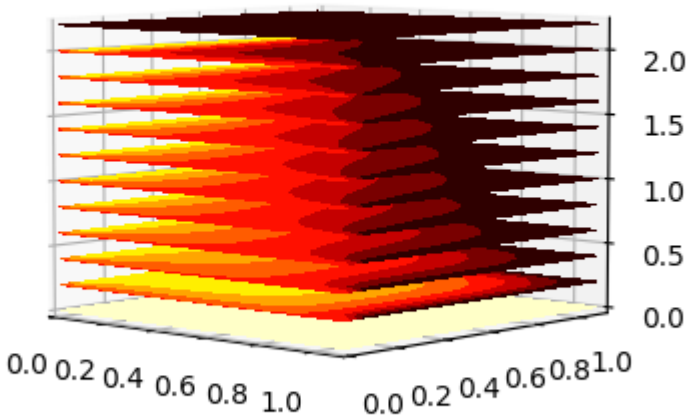
Так как основной бенчмаркинг проводился в предыдущем отчете, здесь будет приведено сравнение CPU и GPU (1024 потока, меньше получалось слишком долго) версии кода на кубе длиной в 48 ячеек. Длина по Z делится на количество процессов по Z:

Количество процессов по Z	CPU	GPU
1	37.3877	39.7421
2	19.0648	58.6405
3	29.2369	75.095
4	26.138	90.5429

Наблюдается экспоненциальное замедление по GPU с увеличением количество процессов. CPU показывает, что такой подход к бенчмарку плохой, но ничего с этим не поделать.

Так как выходные данные не поменялись (а если поменялись, то незначительно для графика), используются те же графики:


5
4
3
2
1
0.2
0.4
0.6
0.8
1.0
0.2
0.4
0.6
0.8
1.0


2.0
1.5
1.0
0.5
0.0
0.0 0.2 0.4 0.6 0.8 1.0
0.0 0.2 0.4 0.6 0.8 1.0

Выводы

- Все сделанные алгоритмы необходимо тестировать на производительность. Если мы усложнили алгоритм, а скорость упала, возникает вопрос: “А в том ли мы направлении усложняли вообще?”;
- Возможной причиной такого роста времени выполнения является чрезмерное копирование на границах буффера, что само выполняется чаще вычисления сетки и для каждой из 6 сторон по 2 раза.