МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа № 3
по курсу «Программирование графических процессоров»

Технология MPI и технология CUDA

Выполнил: А.В. Скворцов
Группа: 8О-408Б
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.

Требуется решить задачу описанную в лабораторной работе № 2, 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 5: обмен граничными слоями через send/receive, контроль сходимости allreduce

Программное и аппаратное обеспечение

- GPU: Geforce 940MX
  - Compute capability : 5.0
  - Total Global Memory : 2147483648
  - 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 : 3
- CPU: Intel Core i5-6200U 2.30GHz
- RAM: 4GB
- Software: Windows 10, Visual Studio Code, nvcc

Метод решения

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

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

процесс останавливается, когда:

$$max_{i,j,k}|u_{i,j,k}^{(k+1)} - u_{i,j,k}^{(k)}| < \epsilon$$

Когда размер сетки становится настолько большим, что его невозможно поместить в оперативной памяти одного компьютера, либо уже не хватает вычислительной мощности одного процессора имеет смысл распределить нагрузку между несколькими компьютерами. Для этого разобьем сетку на несколько смежных подсеток и будем на каждой итерации обмениваться их граничными элементами, осуществить это можно, например, с помощью технологии MPI. Тогда алгоритм работы распределенной программы будет выглядеть следующим образом:

1. Обмен граничными слоями между процессами
2. Обновление значений во всех ячейках
3. Вычисление локальной погрешности в рамках каждого процесса, а затем обмен погрешностями по всей области для поиска максимальной

Описание программы

Вся машинерия алгоритма происходит внутри одного класса HeatMap вместе с вспомогательными cuda-ядрами:

1. конструктор HeatMap – подготавливает память для сетки на гпу, инициализирует ее, вычисляет ранги соседних сеток.
2. void HeatMap::set_gpu(int my_rank) — равномерно распределяет гпу между процессами на одном компьютере.
3. int HeatMap::get_rank(int x, int y, int z) — возвращает ранг процесса с заданными координатами
4. void HeatMap::set_limits — задает индексы границ областей, в рамках которых происходит копирование/установка границ
5. void HeatMap::init_boundary(double val, int dir) — метод-оболочка над init_boundary_kernel
6. __global__ void init_boundary_kernel — проставляет заданное значение на указанную границу.
7. __device__ size_t (*get_index_func(int axis))(int, int, int, int, int, int) — возвращает функцию для индексации буфера в зависимости от выбранной границы
8. __device__ inline size_t index_yz(int nx, int ny, int nz, int i, int j, int k) — индексация буфера вдоль плоскости yz
9. __device__ inline size_t index_xz(int nx, int ny, int nz, int i, int j, int k) — индексация буфера вдоль плоскости xz
10. __device__ inline size_t index_xy(int nx, int ny, int nz, int i, int j, int k) — индексация буфера вдоль плоскости xy
11. void HeatMap::boundary_to_buff(double *buff, int axis) — метод-оболочка над boundary_to_buff_kernel
12. __global__ void boundary_to_buff_kernel — копирование заданной границы в буфер с помощью ядра.
13. void HeatMap::set_boundary(double *buff, int axis) — метод-оболочка над set_boundary_kernel
14. __global__ void set_boundary_kernel — установка значений буфера на заданную границу
15. void HeatMap::sendrecv_along_axis — неблокирующий обмен граничными условиями вдоль заданной оси
16. void HeatMap::exchange_boundaries — обмен всеми границами вдоль всех осей
17. double HeatMap::approximate — один шаг аппроксимации сетки, включает в себя обмен границами, вычисление новых локальных значений (через ядро approximate_kernel), обмен локальной погрешностью для поиска максимальной (с помоью thrust)
18. __global__ void approximate_kernel – ядро, вычисляющее новые значения сетки.
19. struct MaxDiffComparator — компаратор над значениями ткущей и предыдущей сетки для поиска максимального отклонения.
20. void HeatMap::write(char *filepath) — запись всей текущей сетки в файл с указанным именем

Результаты

Скорость работы программы (в миллисекундах) в зависимости от сетки процессов (по горизонтали) от самой сетки (по вертикали) на 4х ядерном процессоре:

	30x30x30	40x40x40
2 блока 32x2x1		
1x1x1	2778	9596
2x1x1	12338	32769
2x2x1	18806	47377
4 блока 32x2x1		
1x1x1	2236	5961
2x1x1	11149	29041
2x2x1	17306	43559
8 блоков 32x2x1		
1x1x1	1869	4218
2x1x1	10353	27174
2x2x1	16678	41848

Если сравнить результаты, с результатами предыдущей лр, можно заметить, что программа в рамках одного процесса стала выполняться быстрее, что логично, однако результаты при использовании нескольких процессов (даже в пределах количеств ядер процессора) заметно ухудшились.

Выводы

Message Passing Interface предоставляет API, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу, применяется в первую очередь при разработке кластеров и суперкомпьютеров. С его помощью можно относительно легко и гибко масштабировать ресурсоемкие программы, которым уже недостаточно вычислительных мощностей одного компьютера. Для дальнейшего повышения производительности таких программ нужно повысить производительность каждого процесса в отдельности. Осуществить это можно, например, с помощью гпу и технологии cuda.