МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №4**

**по курсу «Программирование графических процессоров»**

**Работа с матрицами. Метод Гаусса**

Выполнил: Н.И. Лохматов

Группа: 8О-406Б

Преподаватель: А.Ю. Морозов

Москва, 2024

**Условие**

1. Цель работы: использование объединения запросов к глобальной памяти. Реализация метода Гаусса с выбором главного элемента по столбцу. Ознакомление с библиотекой алгоритмов для параллельных расчетов Thrust. Использование двухмерной сетки потоков. Исследование производительности программы с помощью утилиты nvprof
2. Вариант 2: вычисление обратной матрицы

**Программное и аппаратное обеспечение**

1. Графический процессор: Nvidia GeForce RTX 3050 Mobile
   1. Количество потоковых процессоров: 2560
   2. Частота ядра: 1552 МГц
   3. Количество транзисторов: 8.7 млрд
   4. Тех. процесс: 8 нм
   5. Энергопотребление: 80 Вт
2. OC: Ubuntu 22.04
3. Текстовый редактор: VS Code
4. Компилятор: nvcc

**Метод решения**

Метод Гаусса представляет собой последовательное применение элементарных преобразований для приведения матрицы к ступенчатой форме, а затем к диагональной форме. Мы применяем те же преобразования к единичной матрице, чтобы получить обратную матрицу. Читаем матрицу A из входных данных и создаем единичную матрицу I размером n×n, которая будет модифицироваться в процессе преобразований и станет обратной матрицей для A. Для текущей строки i ищем максимальный элемент в столбце, чтобы минимизировать ошибки из-за деления на малые значения. Если максимальный элемент находится не на главной диагонали, меняем текущую строку с той, в которой найден максимальный элемент. Переходим к обратному ходу: обнуляем элементы выше главной диагонали, снова вычитая кратные строки, пока не получим единичную матрицу в A. Для каждой строки, начиная с i = 0, выполняем нормировку строки — делим её на значение на главной диагонали. Затем вычитаем кратные текущей строки из всех нижележащих строк, чтобы сделать нули ниже диагонального элемента. Единичная матрица в результате преобразований превращается в обратную матрицу, выводим её.

**Описание программы**

Программа реализована в основном файле, где находится логика нахождения обратной матрицы методом Гаусса, макрос для обработки ошибок и CUDA-ядра для выполнения вычислений на GPU. Сначала матрица считывается из ввода и копируется в память устройства. На этапе приведения к верхней треугольной форме для каждой строки осуществляется поиск ведущего элемента и при необходимости выполняется перестановка строк, что повышает точность расчетов. Затем запускается обратный ход, в котором зануляются элементы выше главной диагонали, начиная с последней строки и постепенно поднимаясь вверх. В заключительном этапе выполняется нормировка диагональных элементов до единицы, что приводит исходную единичную матрицу к виду обратной. Программа также учитывает обработку ошибок при вызове функций CUDA и корректно освобождает память по завершении выполнения.

**Результаты**

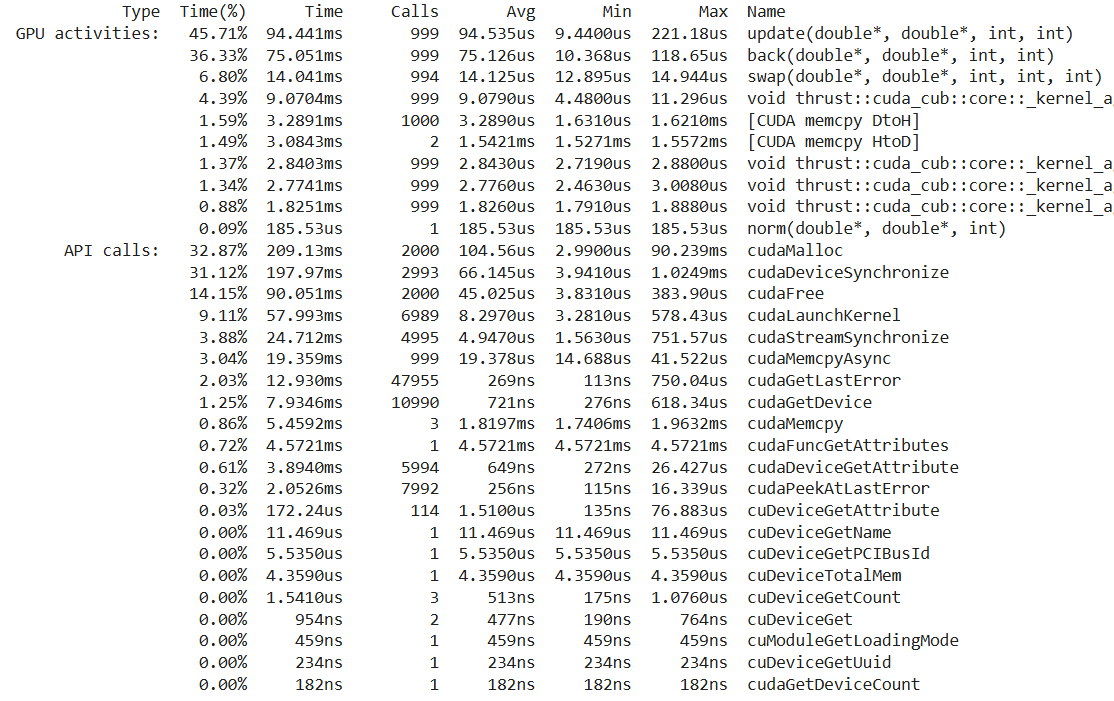
1. Зависимость времени выполнения программы от количества используемых потоков. Вычисления были проведены 100 раз и на их основе посчитано среднее время

|  |  |  |  |
| --- | --- | --- | --- |
| Количество потоков | Время, n = 10^2 (мс) | Время, n = 5 \* 10^2 (мс) | Время, n = 10^3 (мс) |
| 2×2×2×2 | 75.38 | 3236.38 | 32904.55 |
| 8×8×8×8 | 52.46 | 287.87 | 1869.75 |
| 16×16×16×16 | **48.44** | **267.34** | 1726.63 |
| 16×16×32×32 | 70.34 | 299.61 | **1637.62** |
| 32×32×32×32 | 60.19 | 316.79 | 1764.94 |

2. Сравнение программы на CUDA с 16×16×16×16 потоками и программы на CPU с одним потоком

|  |  |  |
| --- | --- | --- |
| Размер n | Время CUDA (мс) | Время CPU (мс) |
| 10^2 | **67.75** | **4.89** |
| 5 \* 10^2 | 303.46 | 855.08 |
| 10^3 | 1480.49 | 12252.88 |

3. Результаты исследования производительности с помощью nvprof. Количество потоков 16×16×16×16, n = 10^3



Ядро update занимает 45.71% общего времени на GPU с временем выполнения 94.441 мс на 999 вызовов. Это основная операция, требующая оптимизации, так как оно выполняет большинство вычислений. Ядро back занимает 36.33% времени (выполнение в течение 75.051 мс на 999 вызовов). Ядро swap, выполняющее перестановки строк для улучшения точности, занимает 6.80% времени (около 14.041 мс на 994 вызова), что также является значительным расходом времени. Ядро norm занимает 0.09% (всего 185.53 мкс за один вызов), что указывает на его небольшое влияние на общую производительность. В профилировании видно, что значительное время (около 4.39% и другие мелкие значения) затрачено на вызовы thrust (например, ReduceAgent), которые используются для поиска максимальных значений в столбцах матрицы при выборе ведущего элемента.

Используем ключ --print-gpu-trace, чтобы получить подробные данные по каждому ядру и узнать, какое ядро занимает больше всего времени, можно использовать (n = 10^2).

**Изображение выглядит как текст, меню, число, снимок экрана

Автоматически созданное описание**

Видно, что операции передачи данных между CPU и GPU имеют высокий пропускной поток (до 8.3761 GB/s) и занимают 78.125 KB для обоих направлений (Pageable to Device и Device to Pageable). Операции CUDA выполняются в различных конфигурациях сетки и блоков. Основные параметры конфигурации сетки (1 1 1) с размером блока (256 1 1) и (16 16 1). Конфигурация сетки (256 1 1) позволяет использовать большое количество потоков в блоке для параллельных вычислений. Конфигурация (16 16 1) может быть использована для задач, где требуется двумерная структура, что может подходить для операций над матрицами. Количество регистров на поток варьируется от 16 до 54, что показывает оптимальное использование регистров CUDA.

**Выводы**

Реализован метод Гаусса с выбором главного элемента по столбцу на GPU с использованием объединения запросов к глобальной памяти. Для выполнения расчетов применена библиотека Thrust, позволяющая эффективно находить максимальные элементы в столбцах. Для параллельной обработки использована двухмерная сетка потоков, обеспечивающая распределение вычислений между потоками CUDA. Производительность программы исследована с помощью утилиты nvprof, что позволило оценить эффективность реализации, выявить возможные узкие места и подтвердить прирост производительности относительно последовательного метода.