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: А.Ю. Гришин
Группа: 8О-408Б
Преподаватель: А.Ю. Морозов

Москва, 2024

Условие

1. **Цель работы:** использование объединения запросов к глобальнйо 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

- Графический процессор
  - Compute capability: 7.5
  - Объем графической памяти: 15,83 ГБ
  - Объем постоянной памяти: 65536 байт
  - Разделяемая память на блок: 49152 байт
  - Количество регистров на блок: 65536
  - Максимальное количество потоков на блок: 1024
  - Количество мультипроцессоров: 40
- Процессор
  - Количество физических ядер: 2
  - Количество логических ядер: 4
  - Частота: 2000 МГц
- Оперативная память
  - Тип оперативной памяти: DDR4
  - Объем: 32 ГБ
- Жесткий диск
  - Объем: 1 ТБ
- Программное обеспечение
  - ОС: Ubuntu 22.04.4 LTS
  - IDE: Visual Studio Code
  - Компилятор: nvcc

Метод решения

Решение данной задачи я начал с определения алгоритма вычисления обратной матрицы с использованием метода Гаусса. Пусть A – исходная матрица. Перед использованием метода Гаусса необходимо построить блочную матрицу $(A \mid E)$.

На следующем этапе, который называется прямым ходом, согласно методу Гаусса, используя элементарные преобразования, необходимо привести левую часть блочной матрицы к верхнетреугольной матрице.

$$(A \mid E) \to (A_2 \mid E_2) = \left(\begin{array}{ccc|ccc} a_{11}^{(2)} & \cdots & a_{1n}^{(2)} & e_{11} & \cdots & e_{1n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & a_{nn}^{(2)} & e_{n1} & \cdots & e_{nn} \end{array}\right)$$

После данного этапа происходит обратный ход. На этом этапе необходимо повторно выполнить метод Гаусса, однако, в этот раз в обратном направлении. В результате получится блочная матрица, левый компонент котороый представляет собой диагональную матрицу.

$$(A_2 \mid E_2) \to (A_3 \mid E_3) = \left(\begin{array}{ccc|ccc} a_{11}^{(3)} & \cdots & 0 & e_{11}^{(3)} & \cdots & e_{1n}^{(3)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & a_{nn}^{(3)} & e_{n1}^{(3)} & \cdots & e_{nn}^{(3)} \end{array}\right)$$

Заключительным этапом является нормализация значений. Для этого необходимо выполнить элементарное преобразование для каждой строки, разделив элементы правой компоненты на соответствующие значения диагональных элементов.

$$(A_3 \mid E_3) \to (A_4 \mid E_4) = \left(\begin{array}{ccc|ccc} 1 & \cdots & 0 & e_{11}^{(4)} & \cdots & e_{1n}^{(4)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & e_{n1}^{(4)} & \cdots & e_{nn}^{(4)} \end{array}\right)$$

Метод Гаусса состоит из двух основных этапов на каждом шаге:

1. Вычисление коэффициентов;
2. Обновление значений элементов ниже текущей строки.

Вычисление коэффициентов

На произвольном шаге метода, когда текущая строка имеет номер i, для всех строк с номерами $j = \overline{i + 1, n}$ вычисляются коэффициенты по следующей формуле:

$$k_j = -\frac{a_{ii}}{a_{ji}}$$

Эти коэффициенты используются для последующего обновления значений элементов в строках ниже текущей. Поскольку вычисление каждого коэффициента k_j зависит только от значений элементов текущей строки и не зависит от других коэффициентов, данный этап можно распараллелить.

Для этого этапа достаточно использовать одномерную сетку потоков, так как вычисление каждого коэффициента требует работы с данными, относящимися к одной строке, и не требует доступа к другим строкам.

Обновление значений элементов

После вычисления коэффициентов происходит обновление значений элементов строк ниже текущей по формуле:

$$a'_{jt} = a_{jt} + k_j \cdot a_{it}, \quad t = \overline{1,n}$$

Каждое новое значение элемента a'_{jt} зависит только от вычисленных ранее коэффициентов, что также позволяет распараллелить вычисления. Для этого этапа целесообразно использовать двумерную сетку потоков, так как каждому потоку необходимо работать с элементами конкретной строки и столбца, то есть с элементами матрицы в двумерной форме.

Оптимизация точности вычислений

При вычислении коэффициентов методом Гаусса может возникнуть ошибка, связанная с делением на малые значения знаменателя, что приводит к значительной потере точности. Чтобы уменьшить эту погрешность, можно использовать оптимизацию путем выбора строки с наибольшим по модулю элементом в качестве ведущей строки на каждом шаге.

Таким образом, на каждом шаге метода выбирается строка с максимальным элементом по модулю в текущем столбце, что позволяет обеспечить максимально возможное значение знаменателя и, следовательно, повысить точность вычислений.

Описание программы

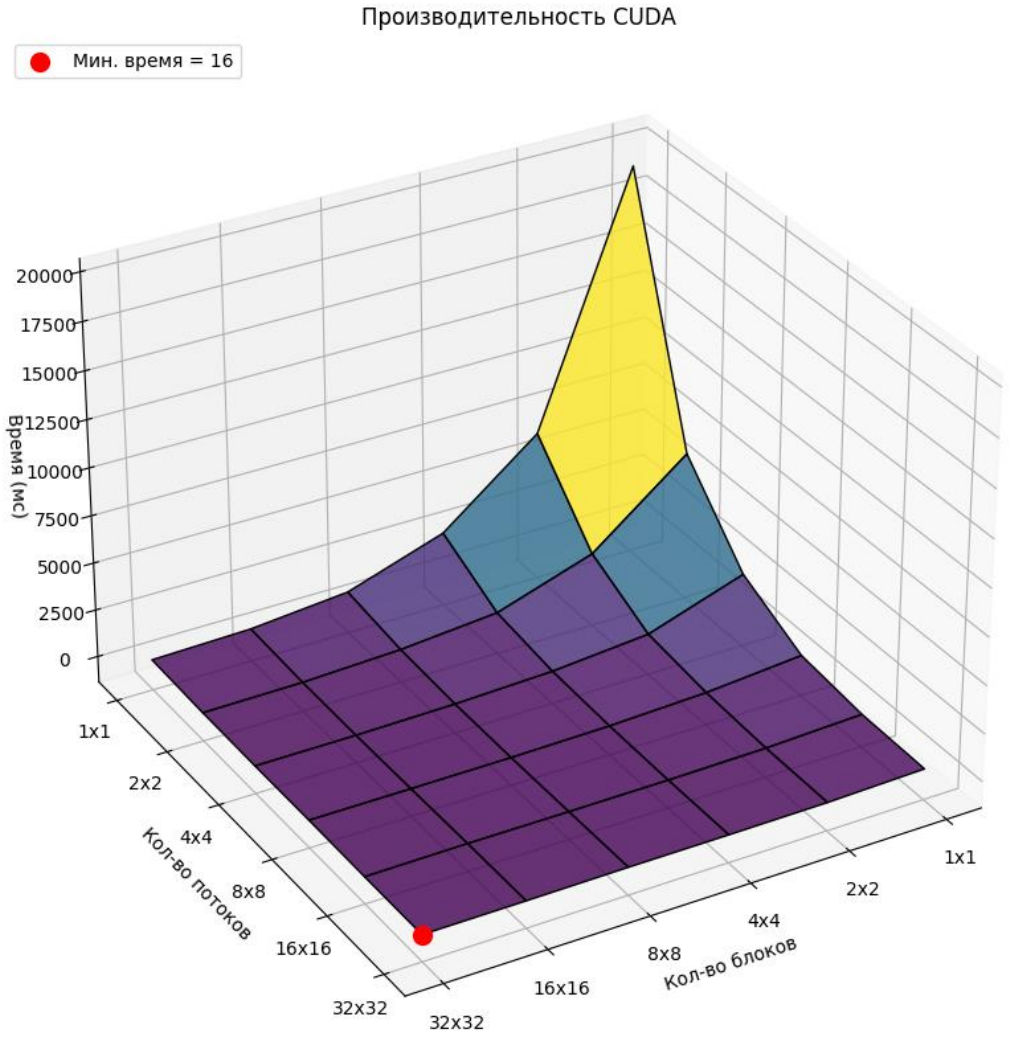
Программа состоит из одного файла, содержащего код, написанный на языке CUDA. Код программы состоит из:

1. Макроса `EXIT_WITH_ERROR`, который отвечает за вывод сообщения об ошибке в `stderr` и экстренный выход из программы с кодом возврата 0;
2. Макроса `SAVE_CUDA`, который проверяет, что CUDA функция сработала успешно. В противном случае происходит экстренный выход через `EXIT_WITH_ERROR`;
3. Функции-ядра `swapRows`, которая отвечает за смену двух строк матрицы местами;
4. Функции-ядра `initCoefs`, которая отвечает за вычисление коэффициентов на каждом шаге прямого хода метода Гаусса;
5. Функции-ядра `updateRows`, которая отвечает за вычисление новых значений элементов на каждом шаге прямого хода метода Гаусса;
6. Функции-ядра initCoefsReversed, которая отвечает за вычисление коэффициентов на каждом шаге обратного хода метода Гаусса;
7. Функции-ядра updateRowsReversed, которая отвечает за вычисление новых значений элементов на каждом шаге прямого хода метода Гаусса;
8. Функции-ядра mult, которая выполняет умножение элементов строки матрицы на переданный коэффициент;
9. Функции-ядра initEyeMatrix для инициализации единичной матрицы размера n;
10. Типа Comparator, содержащего логику по сравнению элементов, что необходимо при поиске максимального по модулю элемента в матрице.

Результаты

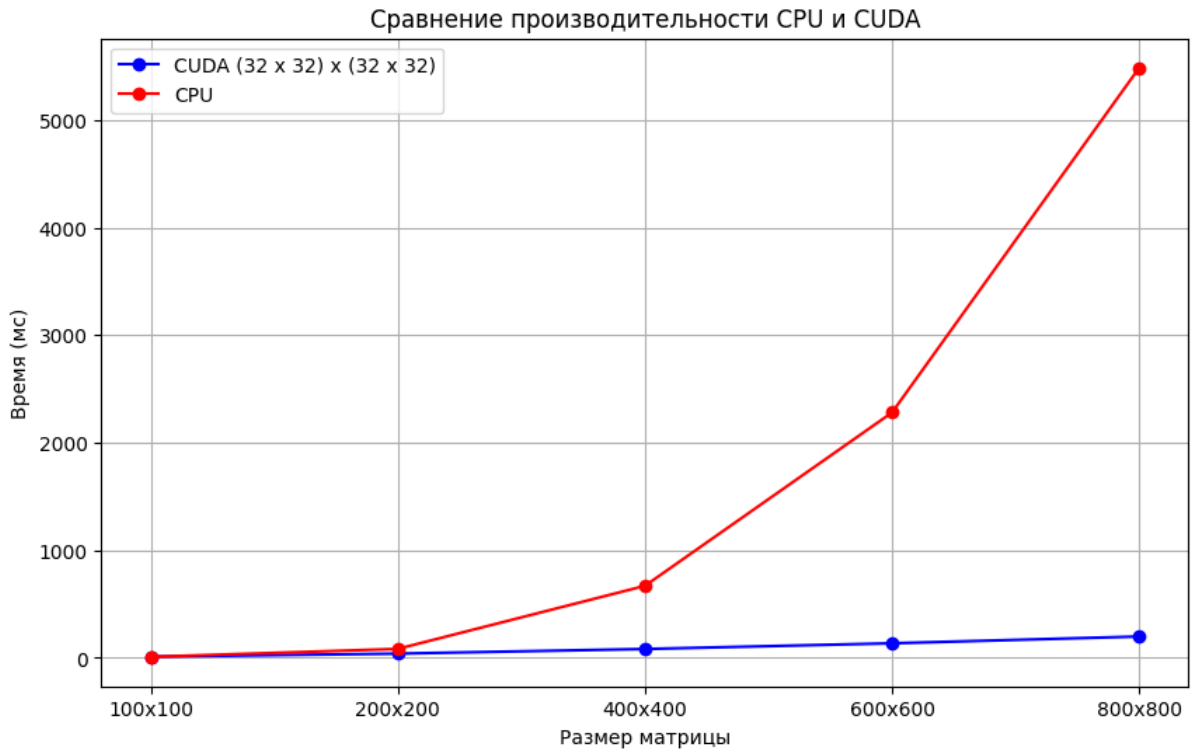
1. Зависимость времени выполнения программы от количества используемых потоков. В качестве единиц измерения времени были выбраны миллисекунды (мс). Тестирование производилось на матрице с размерами 500×500.

Количество блоков / потоков	1×1	2×2	4×4	8×8	16×16	32×32
1×1	19255	6483	2536	830	353	232
2×2	6433	2501	817	368	164	119
4×4	2492	783	342	161	89	79
8×8	771	346	157	87	71	71
16×16	347	156	87	72	72	34
32×32	244	113	80	72	35	16



2. Сравнение программы на CUDA с блоками (32×32) и потоками (32×32) и программы на CPU с одним потоком. В качестве единиц измерения времени были выбраны миллисекунды (мс).

Размер матрицы	Время на CUDA (мс)	Время на CPU (мс)
100×100	12	10
200×200	40	84
400×400	83	670
600×600	136	2281
800×800	199	5484



3. Результат исследования производительности с помощью nvprof (матрица 800×800, сетка потоков $32 \times 32 \times 32 \times 32$)

```
==296== NVPROF is profiling process 296, command: ./a.out
==296== Profiling application: ./a.out
==296== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   32.43%  55.111ms       800  68.888us  31.904us  104.96us  updateRows(double*, double*, double*, int, int)
                   23.17%  39.379ms      1600  24.611us  18.976us  26.272us  swapRows(double*, int, int, int)
                   15.19%  25.823ms       800  32.279us  21.919us  50.144us  updateRowsReversed(double*, double*, double*, int, int)
                   11.42%  19.406ms       800  24.257us  18.719us  25.920us  initCoefs(double*, double*, int, int)
                    8.94%  15.191ms       800  18.989us  18.879us  19.392us  initCoefsReversed(double*, double*, int, int)
                    1.22%  2.0819ms       801  2.5990us  1.5350us  827.38us  [CUDA memcpy DtoH]
                    0.55%  929.78us         1  929.78us  929.78us  929.78us  [CUDA memcpy HtoD]
                    0.08%  135.17us         1  135.17us  135.17us  135.17us  mult(double*, double*, int)
                    0.07%  124.29us         1  124.29us  124.29us  124.29us  initEyeMatrix(double*, int)
      API calls:   42.60%  150.69ms      1602  94.060us  7.2280us  857.76us  cudaDeviceSynchronize
                   31.13%  110.11ms      1603  68.691us  2.7410us  104.49ms  cudaMalloc
                   11.67%  41.293ms      8002  5.1600us  3.4580us  2.5454ms  cudaLaunchKernel
                    5.11%  18.078ms      4000  4.5190us  1.6290us  18.202us  cudaStreamSynchronize
                    2.74%  9.6755ms       800  12.094us  11.102us  49.112us  cudaMemcpyAsync
                    2.25%  7.9588ms     40005     198ns     146ns  697.77us  cudaGetLastError
                    1.66%  5.8589ms      1603  3.6540us  2.2150us  296.61us  cudaFree
                    0.98%  3.4669ms      8801     393ns     332ns  12.774us  cudaGetDevice
                    0.78%  2.7414ms      4800     571ns     330ns  713.75us  cudaDeviceGetAttribute
                    0.65%  2.3057ms         2  1.1529ms  1.1070ms  1.1987ms  cudaMemcpy
                    0.31%  1.0877ms      6400     169ns     145ns  13.473us  cudaPeekAtLastError
                    0.12%  409.04us       228  1.7940us     155ns  94.941us  cuDeviceGetAttribute
                    0.01%  22.351us         1  22.351us  22.351us  22.351us  cudaFuncGetAttributes
                    0.00%  14.463us         2  7.2310us  3.9000us  10.563us  cuDeviceGetName
                    0.00%  13.664us         2  6.8320us  2.3590us  11.305us  cuDeviceGetPCIBusId
                    0.00%  1.3880us         4     347ns     185ns     582ns  cuDeviceGet
                    0.00%  1.2680us         3     422ns     246ns     728ns  cuDeviceGetCount
```

Выводы

В данной лабораторной работе был реализован алгоритм нахождения обратной матрицы методом Гаусса с выбором главного элемента по столбцу на языке CUDA.

В реализации алгоритма были применены оптимизации для эффективного использования глобальной памяти GPU, такие как объединение запросов к глобальной памяти. Данный подход позволил сократить общее количество обращений к памяти и повысить итоговую производительность программы. Основные вычислительные процессы распределялись с использованием двухмерной сетки потоков, что позволило более удобно отобразить вычисляемые элемены на потоки.

Для поиска максимального элемента по модулю в текущем столбце применялась библиотека Thrust, что упростило реализацию этой операции и обеспечило эффективный параллельный поиск.

Алгоритм был структурирован с помощью нескольких функций, каждая из которых выполняет отдельный этап, включая выбор главного элемента, вычисление коэффициентов для преобразования строк и нормализации элементов матрицы. Такая декомпозиция позволила выполнять некоторые

действия алгоритма параллельно, при этом сохраняя общее последовательное выполнение.

Для оценки производительности программы была использована утилита nvprof. С ее помощью было измерено время выполнения отдельных ядер и анализированы узкие места, связанные с доступом к памяти.

Таким образом, в ходе лабораторной работы были изучены возможности эффективного использования памяти, а также были использовани инструменты для анализа производительности программы, таких как nvprof, для достижения более высокой эффективности вычислений на графических процессорах.