МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторная работа №*4*
по курсу «*Программирование графических процессоров*»
Освоение программного обеспечения для работы с технологией CUDA.

Работа с матрицам. Метод Гаусса.

Выполнил: Бачурин Павел Дмитриевич
Группа: М8О-403Б-22
Преподаватели: А.Ю. Морозов,
Е.Е. Заяц

Москва, *2025*

Условие

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 (обязательно отразить в отчете).

Вариант 1. Вычисление детерминанта матрицы.

Входные данные:
На первой строке задано число n -- размер матрицы. В следующих n строках, записано по n вещественных чисел - элементы матрицы. $n \leq 10^4$.

Выходные данные:
Необходимо вывести одно число -- детерминант матрицы.

Программное и аппаратное обеспечение

Графический процессор (GPU):
- Модель: NVIDIA GeForce MX330
- Архитектура CUDA: Pascal
- Compute Capability: 6.1
- Видеопамять (VRAM): 2 ГБ
- Пропускная способность памяти: ~64 ГБ/с
- Количество CUDA-ядер: 384
- Разделяемая память на блок: 48 КБ
- Константная память: 64 КБ
- Количество регистров на блок: 65 536 32-битных регистров
- Максимальное количество блоков (на одном мультипроцессоре): 1024
- Максимальное количество потоков (на один блок): 1024
- Количество мультипроцессоров: 3

Центральный процессор (CPU):
- Модель: Intel i5-1035G1
- Количество ядер: 4
- Количество потоков на ядро: 2
- Тактовая частота: 3.6 ГГц

Количество оперативной памяти: 12 ГБ
Объем жесткого диска: 200 ГБ
Операционная система:

Операционная система: Manjaro Linux Gnome x86_64
Ядро: Linux 6.x
Программная среда: Lenovo 81WD IdeaPad 3 14IIL05
CUDA Toolkit: 12.9
Драйвер NVIDIA: 580.82.09
Компилятор gcc: 11.4.0

Метод решения

1) Подготовка данных на CPU

1. Из стандартного ввода считывается размер матрицы n.
2. В функции read_matrix выделяется память и построчно считываются все n*n элементов матрицы в массив hA.
3. Проверяется корректность считывания данных; при ошибках программа завершается.
4. CPU выделяет память под вспомогательные массивы в GPU:
    - dA — исходная матрица,
    - d_abs_col — массив модулей текущего столбца,
    - d_factors — множители для исключения.
5. Исходная матрица копируется из hA в dA (глобальная память GPU).

2) Подготовка GPU и копирование данных

1. Выделяется глобальная память на GPU с помощью cudaMalloc.
2. Данные копируются на GPU через вызовы cudaMemcpy.
3. Создаются конфигурации сетки и блоков:
    - blockCopy, gridCopy — для вычисления модуля столбца,
    - blockSwap, gridSwap — для перестановки строк,
    - blockFactors, gridFactors — для вычисления множителей,
    - blockElim, gridElim — для исключения Гаусса,
    - blockZero, gridZero — для явного зануления столбца.
4. Все вызовы CUDA оборачиваются макросом CSC(...) для проверки ошибок.

3) Параллельные вычисления на GPU

Для каждой итерации Гаусса по столбцу k = 0 .. n-1 выполняются следующие шаги:

3.1 — Вычисление модуля элементов столбца |A[i][k]|

Запускается ядро:

copy_abs_col_kernel<<<gridCopy, blockCopy>>>(dA, d_abs_col, n, k);

Каждый поток получает один или несколько элементов столбца, вычисляет fabs(A[i][k]) и записывает в d_abs_col.

3.2 — Поиск строки с максимальным элементом (выбор ведущего элемента)

Библиотека Thrust выполняет:

max_element(d_abs_col, d_abs_col + (n - k))

Результат — индекс строки pivot_row с максимальным элементом в столбце.

3.3 — Проверка на нулевой столбец

Если значение pivot_val = |A[pivot_row][k]| ≤ ε:

- матрица вырождена,
- определитель равен нулю,
- выполнение завершается.

3.4 — Перестановка строк

Если строка pivot_row не совпадает с текущей k, то:

swap_rows_kernel<<<gridSwap, blockSwap>>>(...);

GPU параллельно меняет строки местами.
Меняется знак определителя (sign = -sign).

3.5 — Накопление значения определителя

На шаге k в определитель добавляется диагональный элемент:

det *= A[k][k]

3.6 — Вычисление множителей (коэффициентов Гаусса)

Запуск ядра:

compute_factors_kernel<<<...>>>(dA, d_factors, n, k, pivot);

Каждый поток вычисляет:

factor[i] = A[i][k] / pivot

Массив d_factors хранит коэффициенты для всех строк ниже текущей.

3.7 — Исключение Гаусса (обнуление правой части строки)

Главная часть алгоритма:

eliminate_kernel<<<...>>>(dA, d_factors, n, k);

Каждый поток:

- берёт строку i > k
- вычисляет новую строку:
  A[i][j] -= factor[i] * A[k][j]
- обновляет элементы подматрицы начиная с col = k.

3.8 — Явное зануление столбца

zero_column_kernel<<<...>>>(...)

Обнуляет элементы A[i][k], чтобы избежать накопления ошибок.

4) Синхронизация потоков

После каждого CUDA-ядра выполняется:

cudaGetLastError()
cudaDeviceSynchronize()

Это гарантирует корректное завершение вычислений перед переходом к следующей итерации.

5) Возврат результата и освобождение ресурсов

1. Вычисленный определитель выводится через printf.
2. Освобождается вся выделенная память GPU (cudaFree).
3. Освобождается память CPU (free).

Архитектура программы

1) CPU — управляющая часть

- Считывает входные данные.

- Выделяет память.
- Инициирует копирование данных на GPU.
- Запускает CUDA-ядра для выполнения Гаусса.
- Считывает результаты и печатает определитель.

CPU выполняет только управляющие функции, сама математика перенесена на GPU.

2) GPU — вычислительная часть

GPU выполняет:

- вычисление модулей столбца,
- перестановку строк,
- вычисление коэффициентов,
- весь шаг исключения Гаусса,
- обнуление столбца.

Каждое действие реализовано отдельным CUDA-ядром, работающим параллельно на многих потоках.

3) Взаимодействие CPU ↔ GPU

- Передача данных: cudaMemcpy.
- Выделение памяти: cudaMalloc, cudaFree.
- Запуск ядер: kernel<<<grid, block>>>.
- Проверка ошибок: CSC(call).

GPU работает как вычислительный сервер, CPU — как клиент, управляющий данными.

Основные функции и ядра CUDA

1. Макрос CSC(call)

Проверяет корректность каждого вызова CUDA API.
При ошибке — выводит сообщение и завершает программу.

2. read_matrix()

Считывает размер матрицы и все её элементы из входного потока.
Выделяет память и возвращает заполненную матрицу.

CUDA-ядра:

3. copy_abs_col_kernel()

Параллельно вычисляет абсолютные значения элементов текущего столбца ниже диагонали.
Используется для выбора pivot-строки.

4. swap_rows_kernel()

Меняет две строки матрицы местами.
Выполняется параллельно по столбцам.

5. compute_factors_kernel()

Вычисляет множители метода Гаусса для всех строк ниже текущей.

6. eliminate_kernel()

Параллельно вычитает из каждой строки `factor * pivot_row`,
реализуя основной шаг исключения Гаусса.

7. zero_column_kernel()

Явно зануляет элементы столбца под диагональю,
устраняя накопленные ошибки вычислений.

Результаты

1. Сравнение программы на CUDA 32*32 и программы на CPU с одним потоком:

Размерность матрицы	Время на CUDA (в мс)	Время на CPU (в мс)
500 x 500	62.359074	127.335
10000 x 10000	216.293732	1018.014
5000 x 5000	15916.067383	120935.113

Исследование производительности

nvprof ./gpu.out < matrix

==9351== NVPROF is profiling process 9351, command: ./gpu.out
time: 16149.474609 ms
==9351== Profiling application: ./gpu.out
==9351== Profiling result:

Category	%	Time	#	Avg	Min	Max	Name
GPU activities	98.11	15.3709s	4999	3.0748ms	11.519us	9.3889ms	eliminate_kernel(double*, double const *, int, int)
	0.54	84.537ms	5000	16.907us	12.413us	26.748us	copy_abs_col_kernel(double const , *double*, int, int)
	0.41	64.174ms	1	64.174ms	64.174ms	64.174ms	[CUDA memcpy HtoD]
	0.23	35.717ms	4999	7.1440us	2.0800us	26.267us	compute_factors_kernel(double const , *double*, int, int, double)
	0.22	35.105ms	4999	7.0220us	1.8880us	28.539us	zero_column_kernel(double*, int, int)
	0.14	21.309ms	15000	1.4200us	352ns	28.380us	[CUDA memcpy DtoH]
	0.12	18.772ms	4989	3.7620us	2.5920us	28.283us	swap_rows_kernel(double*, int, int, int)
API calls	96.70	15.6603s	24986	626.76us	909ns	9.3935ms	cudaDeviceSynchronize
	1.05	170.03ms	10001	17.001us	8.6110us	63.786ms	cudaMemcpy
	0.79	127.67ms	37426	3.4110us	2.0270us	712.08us	cudaLaunchKernel
	0.53	85.647ms	5003	17.119us	2.0060us	65.742ms	cudaMalloc
	0.39	63.562ms	5000	12.712us	9.0500us	605.83us	cudaMemcpyAsync
	0.25	40.371ms	10000	4.0370us	564ns	48.119us	cudaStreamSynchronize
	0.11	17.263ms	5003	3.4500us	1.6890us	3.2007ms	cudaFree
	0.07	11.211ms	199388	56ns	41ns	37.585us	cudaGetLastError
	0.05	7.4365ms	39881	186ns	128ns	32.987us	cudaGetDevice
	0.04	6.2533ms	29880	209ns	121ns	38.807us	cudaDeviceGetAttribute
	0.02	4.0446ms	7440	543ns	253ns	32.077us	CudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
	0.01	1.4252ms	24880	57ns	43ns	939ns	cudaPeekAtLastError
	0.00	225.80us	114	1.9800us	152ns	86.708us	cuDeviceGetAttribute
	0.00	26.990us	1	26.990us	26.990us	26.990us	cuDeviceGetName
	0.00	24.995us	2	12.497us	8.0260us	16.969us	cudaEventRecord
	0.00	12.015us	2	6.0070us	269ns	11.746us	cuDeviceGet
	0.00	8.9820us	2	4.4910us			

Выводы

В ходе работы была реализована программа вычисления детерминанта матрицы методом Гаусса с выбором главного элемента по столбцу на GPU. Использовалось объединение запросов к глобальной памяти и двухмерная сетка потоков для ускорения вычислений. Поиск максимального элемента выполнялся с помощью библиотеки Thrust. Производительность программы исследована утилитой nvprof, показано распределение времени между вычислениями и операциями с памятью. Результаты вычислений точны и соответствуют заданной точности.