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
    a. Количество потоковых процессоров: 2560
    b. Частота ядра: 1552 МГц
    c. Количество транзисторов: 8.7 млрд
    d. Тех. процесс: 8 нм
    e. Энергопотребление: 80 Вт
2. ОС: 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

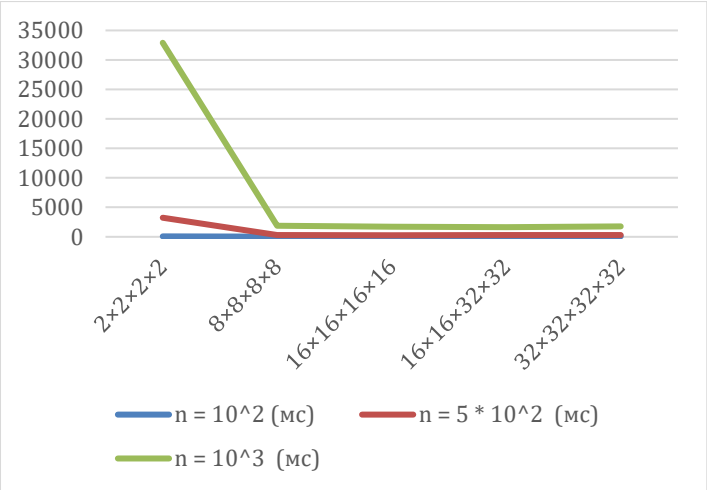
Программа реализована в основном файле, где находится логика нахождения обратной матрицы методом Гаусса, макрос для обработки ошибок и CUDA-ядра для выполнения вычислений на GPU. Сначала матрица считывается из ввода и копируется в память устройства. На этапе приведения к верхней треугольной форме для 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

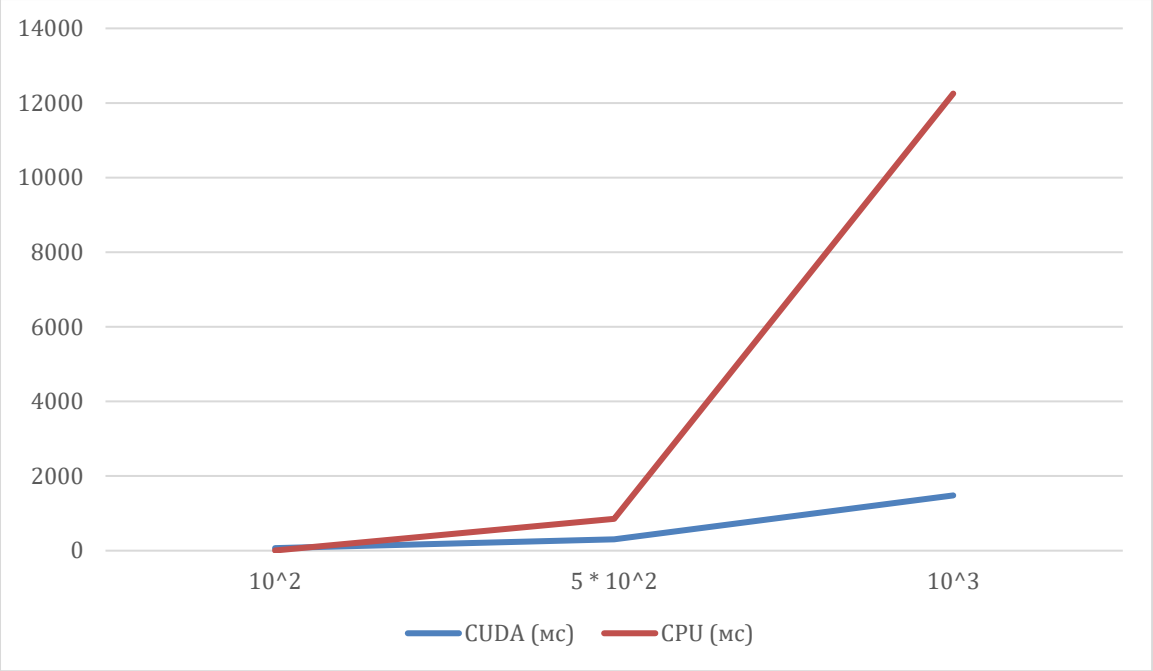
1. Зависимость времени выполнения программы от количества используемых потоков. Вычисления были проведены 100 раз и на их основе посчитано среднее время

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

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	45.71%	94.441ms	999	94.535us	9.4400us	221.18us	update(double*, double*, int, int)
	36.33%	75.051ms	999	75.126us	10.368us	118.65us	back(double*, double*, int, int)
	6.80%	14.041ms	994	14.125us	12.895us	14.944us	swap(double*, double*, int, int, int)
	4.39%	9.0704ms	999	9.0790us	4.4800us	11.296us	void thrust::cuda_cub::core::_kernel_a
	1.59%	3.2891ms	1000	3.2890us	1.6310us	1.6210ms	[CUDA memcpy DtoH]
	1.49%	3.0843ms	2	1.5421ms	1.5271ms	1.5572ms	[CUDA memcpy HtoD]
	1.37%	2.8403ms	999	2.8430us	2.7190us	2.8800us	void thrust::cuda_cub::core::_kernel_a
	1.34%	2.7741ms	999	2.7760us	2.4630us	3.0080us	void thrust::cuda_cub::core::_kernel_a
	0.88%	1.8251ms	999	1.8260us	1.7910us	1.8880us	void thrust::cuda_cub::core::_kernel_a
	0.09%	185.53us	1	185.53us	185.53us	185.53us	norm(double*, double*, int)
API calls:	32.87%	209.13ms	2000	104.56us	2.9900us	90.239ms	cudaMalloc
	31.12%	197.97ms	2993	66.145us	3.9410us	1.0249ms	cudaDeviceSynchronize
	14.15%	90.051ms	2000	45.025us	3.8310us	383.90us	cudaFree
	9.11%	57.993ms	6989	8.2970us	3.2810us	578.43us	cudaLaunchKernel
	3.88%	24.712ms	4995	4.9470us	1.5630us	751.57us	cudaStreamSynchronize
	3.04%	19.359ms	999	19.378us	14.688us	41.522us	cudaMemcpyAsync
	2.03%	12.930ms	47955	269ns	113ns	750.04us	cudaGetLastError
	1.25%	7.9346ms	10990	721ns	276ns	618.34us	cudaGetDevice
	0.86%	5.4592ms	3	1.8197ms	1.7406ms	1.9632ms	cudaMemcpy
	0.72%	4.5721ms	1	4.5721ms	4.5721ms	4.5721ms	cudaFuncGetAttributes
	0.61%	3.8940ms	5994	649ns	272ns	26.427us	cudaDeviceGetAttribute
	0.32%	2.0526ms	7992	256ns	115ns	16.339us	cudaPeekAtLastError
	0.03%	172.24us	114	1.5100us	135ns	76.883us	cuDeviceGetAttribute
	0.00%	11.469us	1	11.469us	11.469us	11.469us	cuDeviceGetName
	0.00%	5.5350us	1	5.5350us	5.5350us	5.5350us	cuDeviceGetPCIBusId
	0.00%	4.3590us	1	4.3590us	4.3590us	4.3590us	cuDeviceTotalMem
	0.00%	1.5410us	3	513ns	175ns	1.0760us	cuDeviceGetCount
	0.00%	954ns	2	477ns	190ns	764ns	cuDeviceGet
	0.00%	459ns	1	459ns	459ns	459ns	cuModuleGetLoadingMode
	0.00%	234ns	1	234ns	234ns	234ns	cuDeviceGetUuid
	0.00%	182ns	1	182ns	182ns	182ns	cudaGetDeviceCount

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).

Start	Duration	Grid Size	Block Size	Regs*	SSMem*	DSMem*	Size	Throughput	SrcMemType	DstMemType	
164.04ms	9.3120us	-	-	-	-	-	78.125KB	8.0011GB/s	Pageable	Device	Tesla
164.07ms	8.8950us	-	-	-	-	-	78.125KB	8.3761GB/s	Pageable	Device	Tesla
171.25ms	6.7520us	(1 1 1)	(256 1 1)	46	0B	160B	-	-	-	-	Tesla
171.29ms	2.8800us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.34ms	2.8480us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.39ms	2.1440us	-	-	-	-	-	16B	7.1170MB/s	Device	Pageable	Tesla
171.43ms	1.9200us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.47ms	4.9280us	(256 1 1)	(256 1 1)	36	0B	0B	-	-	-	-	Tesla
171.49ms	13.632us	(16 16 1)	(16 16 1)	54	0B	0B	-	-	-	-	Tesla
171.54ms	6.0160us	(1 1 1)	(256 1 1)	46	0B	160B	-	-	-	-	Tesla
171.57ms	2.4640us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.59ms	2.8480us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.61ms	1.9200us	-	-	-	-	-	16B	7.9473MB/s	Device	Pageable	Tesla
171.64ms	1.7920us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.67ms	4.8000us	(256 1 1)	(256 1 1)	36	0B	0B	-	-	-	-	Tesla
171.69ms	13.568us	(16 16 1)	(16 16 1)	54	0B	0B	-	-	-	-	Tesla
171.73ms	6.1750us	(1 1 1)	(256 1 1)	46	0B	160B	-	-	-	-	Tesla
171.76ms	2.5600us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.79ms	2.8160us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.81ms	1.6640us	-	-	-	-	-	16B	9.1699MB/s	Device	Pageable	Tesla
171.83ms	1.8240us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.87ms	4.8320us	(256 1 1)	(256 1 1)	36	0B	0B	-	-	-	-	Tesla
171.88ms	13.536us	(16 16 1)	(16 16 1)	54	0B	0B	-	-	-	-	Tesla
171.93ms	6.1440us	(1 1 1)	(256 1 1)	46	0B	160B	-	-	-	-	Tesla
171.97ms	2.5280us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
171.99ms	2.8480us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
172.01ms	1.6640us	-	-	-	-	-	16B	9.1699MB/s	Device	Pageable	Tesla
172.03ms	1.8240us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
172.06ms	4.8000us	(256 1 1)	(256 1 1)	36	0B	0B	-	-	-	-	Tesla
172.08ms	13.504us	(16 16 1)	(16 16 1)	54	0B	0B	-	-	-	-	Tesla
172.13ms	5.8240us	(1 1 1)	(256 1 1)	46	0B	160B	-	-	-	-	Tesla
172.16ms	2.5600us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla
172.18ms	2.8480us	(1 1 1)	(256 1 1)	16	0B	0B	-	-	-	-	Tesla

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.