МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторная работа № 2
по курсу «Технологии параллельного программирования»

Работа с матрицами. Метод Гаусса

Выполнил: И.В. Сектименко
Группа: М8О-410Б-22
Преподаватели: А.Ю. Морозов,
Е.Е. Заяц

Москва, 2025

Условие

Кратко описывается задача:

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 двумерной сетки потоков. Исследование производительности программы с помощью утилиты *nvprof*.
2. Вариант 3. Решение квадратной СЛАУ.
   Необходимо решить систему уравнений $Ax = b$, где A – квадратная матрица размера n x n, b, – вектор-столбец свободных коэффициентов длинной n, x – вектор неизвестных.
   Входные данные. На первой строке задано число n ($n \leq 10^4$) – размер матрицы. В следующих n строках записано по n вещественных чисел – элементы матрицы. Далее записывается n элементов вектора свободных коэффициентов.
   Выходные данные. Необходимо вывести n значений, являющиеся элементами вектора неизвестных x.

Программное и аппаратное обеспечение

Характеристики:

- графического процессора:

compute capability:	7.5
графическая память:	15828320256
разделяемая память на блок:	49152
константная память:	65536
количество регистров на блок:	65536
максимальное количество блоков:	(2147483647, 65535, 65535)
максимальное количество нитей на блок:	(1024, 1024, 64)
количество мультипроцессоров:	40

- процессора: Intel(R) Xeon(R) CPU @ 2.00GHz;
- оперативной памяти: 12Гб общей памяти;
- жесткого диска: 256 Гб общей памяти.

Во время выполнения работы использовалась IDE Google Colab. В нее встроена ОС Ubuntu 22.04.4 LTS с видеокартой NVIDIA-SMI 550.54.15.

Метод решения

Для решения задания необходимо привести расширенную матрицу (матрицу системы с приписанным к ней справа вектор-столбцом свободных коэффициентов) к ступенчатому виду методом Гаусса. При этом чтобы избежать ошибок из-за деления на очень маленькие числа (это может привести к переполнению и непредсказуемому поведению программы), нужно менять строки местами там, чтобы в итоге вычиталась строка с наибольшим ведущим элементом.

Описание программы

Программа состоит из 1 файла.

Сначала считывается вся необходимая информация из стандартного потока ввода. При этом важно отметить, что матрица системы и вектор-столбец свободных коэффициентов хранятся в транспонированном виде, то есть по столбцам.
Расширенная матрица системы перекопируется в глобальную память GPU.
Запускается цикл, который проходится по столбцам. На каждой итерации цикла с помощью функции *max_element* из библиотеки *Thrust* определяется позиция максимального элемента столбца на данной итерации (все элементы столбца хранятся рядом). Если это не диагональный элемент, то строки меняются местами так, чтобы этот элемент стал диагональным. Для этого применяется функция *replace* на GPU с использованием одномерной сетки потоков. Далее функция *division* производит деление элементов «ведущей» строки на диагональный элемент. Таким образом, диагональный элемент обращается в единицу. Это позволяет уменьшим количество обращений к элементам матрицы, один раз реализовать предподсчет вместо использования диагонального элемента в вычислении каждого элемента каждой поддиагональной строки. И наконец-то после всех подготовок применяется основная функция *kernel*, в которой используется двумерная сетка потоков, ведь происходит обработка двумерной матрицы, а не какой-то ее отдельной строки или столбца. В этой функции как раз и реализуется сам метод Гаусса: вычисляется новое значения для каждого незануленного элемента поддиагональных строк. При этом циклы построены так, чтобы элементы обрабатывались по возможности последовательно, как хранятся в памяти, то есть по столбцам. Это и является объединением запросов к глобальной памяти, что дает ускорение при работе ядра.
Все вычисления происходят с изменением исходной расширенной матрицы системы. Результирующая матрица перекопируется в память на CPU и уже там последовательно вычисляется вектор неизвестных, проходясь снизу вверх по верхнетреугольной матрице. Полученный результат выводится в стандартный поток вывода.

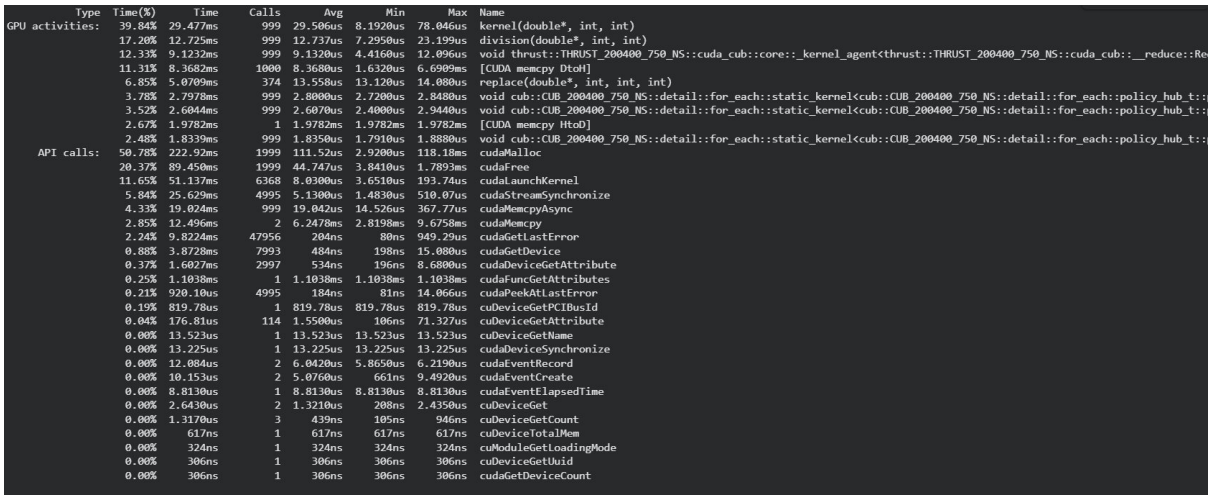
Результаты

Ниже приведена сравнительная таблица. В строках – программа выполняется ядрами функции *kernel* с разными конфигурациями (двумерная сетка потоков) или на центральном процессоре, в столбцах – разные объемы данных, то есть размер матрицы.

	10	100	1 000
dim3(1,1), dim3(4, 4)	2,01 мс	23,32 мс	6289,41 мс
dim3(4, 4), dim3(8, 8)	1,71 мс	8,71 мс	418,62 мс
dim3(8, 8), dim3(16, 16)	1,57 мс	8,49 мс	216,76 мс
dim3(16, 16), dim3(32, 32)	1,62 мс	9,09 мс	209,65 мс
dim3(32, 32), dim3(32, 32)	1,74 мс	9,87 мс	264,33 мс
CPU	0,005 мс	1,56 мс	3593,82 мс

На картинке 1 приведены результаты исследования программы с конфигурацией <<<dim3(16, 16), dim3(32, 32)>>> и размером данных при n = 1 000 с помощью утилиты

nvprof. Так можно посмотреть сколько времени и на что тратится при выполнении программы.



Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	39.84%	29.477ms	999	29.506us	8.1920us	78.046us	kernel(double*, int, int)
	17.20%	12.725ms	999	12.737us	7.2950us	23.199us	division(double*, int, int)
	12.33%	9.1232ms	999	9.1320us	4.4160us	12.096us	void thrust::THRUST_200400_750_NS::cuda_cub::core::_kernel_agent<thrust::THRUST_200400_750_NS::cuda_cub::__reduce::Re
	11.31%	8.3682ms	1000	8.3680us	1.6320us	6.6909ms	[CUDA memcpy DtoH]
	6.85%	5.0709ms	374	13.558us	13.120us	14.080us	replace(double*, int, int, int)
	3.78%	2.7978ms	999	2.8000us	2.7200us	2.8480us	void cub::CUB_200400_750_NS::detail::for_each::static_kernel<cub::CUB_200400_750_NS::detail::for_each::policy_hub_t::
	3.52%	2.6044ms	999	2.6070us	2.4000us	2.9440us	void cub::CUB_200400_750_NS::detail::for_each::static_kernel<cub::CUB_200400_750_NS::detail::for_each::policy_hub_t::
	2.67%	1.9782ms	1	1.9782ms	1.9782ms	1.9782ms	[CUDA memcpy HtoD]
	2.48%	1.8339ms	999	1.8350us	1.7910us	1.8880us	void cub::CUB_200400_750_NS::detail::for_each::static_kernel<cub::CUB_200400_750_NS::detail::for_each::policy_hub_t::
API calls:	50.78%	222.92ms	1999	111.52us	2.9200us	118.18ms	cudaMalloc
	20.37%	89.450ms	1999	44.747us	3.8410us	1.7893ms	cudaFree
	11.65%	51.137ms	6368	8.0300us	3.6510us	193.74us	cudaLaunchKernel
	5.84%	25.629ms	4995	5.1300us	1.4830us	510.07us	cudaStreamSynchronize
	4.33%	19.024ms	999	19.042us	14.526us	367.77us	cudaMemcpyAsync
	2.85%	12.496ms	2	6.2478ms	2.8198ms	9.6758ms	cudaMemcpy
	2.24%	9.8224ms	47956	204ns	80ns	949.29us	cudaGetLastError
	0.88%	3.8728ms	7993	484ns	198ns	15.080us	cudaGetDevice
	0.37%	1.6027ms	2997	534ns	196ns	8.6800us	cudaDeviceGetAttribute
	0.25%	1.1038ms	1	1.1038ms	1.1038ms	1.1038ms	cudaFuncGetAttributes
	0.21%	920.10us	4995	184ns	81ns	14.066us	cudaPeekAtLastError
	0.19%	819.78us	1	819.78us	819.78us	819.78us	cuDeviceGetPCIBusId
	0.04%	176.81us	114	1.5500us	106ns	71.327us	cuDeviceGetAttribute
	0.00%	13.523us	1	13.523us	13.523us	13.523us	cuDeviceGetName
	0.00%	13.225us	1	13.225us	13.225us	13.225us	cudaDeviceSynchronize
	0.00%	12.084us	2	6.0420us	5.8650us	6.2190us	cudaEventRecord
	0.00%	10.153us	2	5.0760us	661ns	9.4920us	cudaEventCreate
	0.00%	8.8130us	1	8.8130us	8.8130us	8.8130us	cudaEventElapsedTime
	0.00%	2.6430us	2	1.3210us	208ns	2.4350us	cuDeviceGet
	0.00%	1.3170us	3	439ns	105ns	946ns	cuDeviceGetCount
	0.00%	617ns	1	617ns	617ns	617ns	cuDeviceTotalMem
	0.00%	324ns	1	324ns	324ns	324ns	cuModuleGetLoadingMode
	0.00%	306ns	1	306ns	306ns	306ns	cuDeviceGetUuid
	0.00%	306ns	1	306ns	306ns	306ns	cudaGetDeviceCount

Рисунок 1 – Результаты исследования программы с помощью утилиты *nvprof*

Выводы

Метод Гаусса часто используется при работе с матрицами. В свою очередь матрицы – удобный и распространенный способ представления данных. Поэтому наиболее оптимальная по времени реализация алгоритма очень важна. В моем варианте метод Гаусса применялся для решения системы уравнений, что уже является прикладной задачей, ведь зачастую математические модели разных задач сводятся к решению системы линейных уравнений.

Запрограммировать данный алгоритм было достаточно легко. Однако я не сразу поняла, что вынесение деления всех элементов ведущей строки на ведущий элемент приведет к значительному ускорению работы программы.

Распараллеливание алгоритма на больших данных привело к хорошим результатам и значительно ускорило программу. Однако важно помнить, что на выделение потоков и ресурсов требуется время, поэтому конфигурации с малым количеством потоков не дают значительного ускорения. И по этой же причине на малых данных оптимальнее использовать CPU.