МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №1
по курсу «Параллельная обработка данных»

Сортировка чисел на GPU. Свертка, сканирование, гистограмма

Выполнил: Н.И. Лохматов
Группа: 8О-406Б
Преподаватель: А.Ю. Морозов

Москва, 2024

Условие

1. Цель работы: ознакомление с фундаментальными алгоритмами GPU: свертка (reduce), сканирование (blelloch scan) и гистограмма (histogram). Реализация одной из сортировок на CUDA. Использование разделяемой и других видов памяти. Исследование производительности программы с помощью утилиты nvprof.
2. Вариант 2: сортировка подсчетом. Диапазон от 0 до $2^{24} - 1$

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

Сначала строится гистограмма массива. После построения гистограммы выполняется её сканирование. Этот процесс преобразует массив таким образом, чтобы каждый элемент содержал индекс начала своего числа в отсортированном массиве. Сканирование реализовано поэтапно: сначала внутри каждого блока потоков вычисляются локальные суммы, затем эти суммы объединяются для всех блоков, и, наконец, корректируются значения для получения глобальной префиксной суммы. В конце запись отсортированных данных. Каждый поток обращается к массиву гистограммы, чтобы определить, куда записать числа в выходной массив. Для каждого числа из гистограммы заполняется диапазон в итоговом массиве, начиная с рассчитанной позиции и в количестве, равном частоте появления числа.

Описание программы

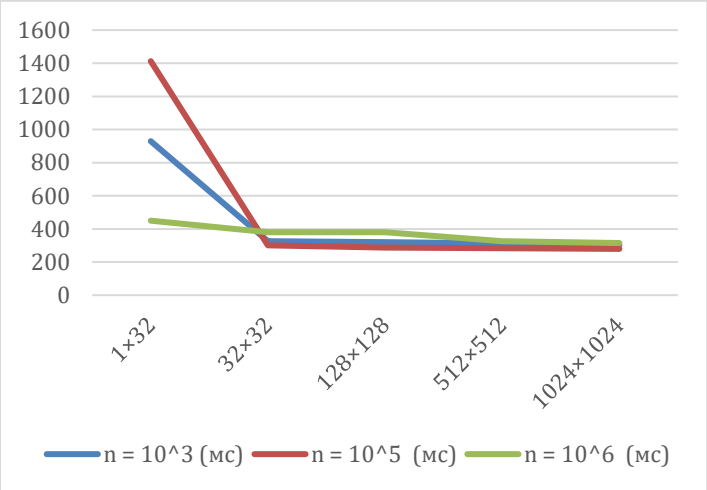
Программа реализована в основном файле, где находится логика сортировки массива чисел методом гистограммного подхода, а также макрос для обработки ошибок и CUDA-ядра для выполнения вычислений на GPU. Сначала входные данные считываются, после чего запускается процесс построения гистограммы. Для этого каждое число из массива подсчитывается с помощью атомарных операций, чтобы избежать конфликтов между потоками. На следующем этапе запускается ядро для выполнения операции сканирования над массивом гистограммы. Сначала в каждом блоке вычисляются локальные суммы, затем результаты блоков объединяются, и корректировки применяются к каждому блоку для получения глобальной префиксной суммы. В финальном этапе запускается ядро записи отсортированных данных, где каждый поток берет информацию из преобразованного массива гистограммы и

заполняет итоговый массив числами в соответствующих позициях. Потоки работают независимо, что обеспечивает высокую степень параллелизма.
Программа корректно обрабатывает ошибки вызовов CUDA, освобождает память на GPU после завершения выполнения и выводит отсортированный массив в бинарный поток.

Результаты

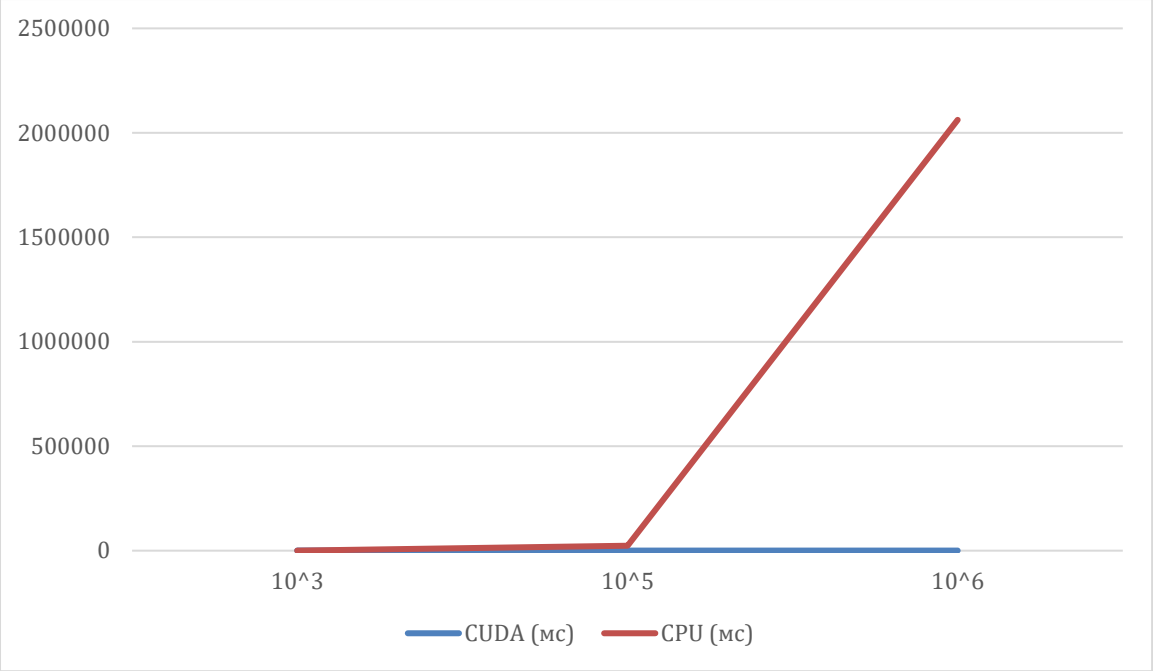
1. Зависимость времени выполнения программы от BLOCK_SIZE. Вычисления были проведены 100 раз и на их основе посчитано среднее время

Количество потоков	Время, n = 10^3 (мс)	Время, n = 10^5 (мс)	Время, n = 10^6 (мс)
1×32	929.38	1412.38	2801.55
32×32	326.46	299.91	449.54
128×128	319.68	287.54	379.12
512×512	311.87	283.12	326.44
1024×1024	**303.76**	**279.21**	**314.63**



2. Сравнение программы на CUDA с 1024×1024 потоками и программы на CPU с одним потоком

Размер n	Время CUDA (мс)	Время CPU (мс)
10^3	303.76	**134.72**
10^5	**279.21**	23119.46
10^6	314.63	2062039.11



3. Результаты исследования производительности с помощью nvprof. Количество потоков 1024×1024, n = 10^4

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	97.78%	240.56ms	1	240.56ms	240.56ms	240.56ms	finalRes(int*, int*, int)
	1.83%	4.5114ms	3	1.5038ms	25.727us	4.4556ms	prefixScan(int*, int*, int)
	0.25%	622.96us	2	311.48us	26.271us	596.69us	addScan(int*, int*, int)
	0.12%	295.13us	1	295.13us	295.13us	295.13us	initRes(int*)
	0.01%	26.944us	1	26.944us	26.944us	26.944us	hist(int*, int*, int)
	0.00%	6.1120us	1	6.1120us	6.1120us	6.1120us	[CUDA memcpy HtoD]
	0.00%	4.7360us	1	4.7360us	4.7360us	4.7360us	[CUDA memcpy DtoH]
API calls:	56.13%	246.07ms	8	30.759ms	29.598us	240.58ms	cudaDeviceSynchronize
	43.65%	191.35ms	5	38.270ms	4.1330us	191.18ms	cudaMalloc
	0.10%	431.97us	8	53.996us	5.1280us	379.28us	cudaLaunchKernel
	0.05%	236.13us	4	59.032us	4.9410us	204.45us	cudaFree
	0.04%	172.85us	114	1.5160us	140ns	63.849us	cuDeviceGetAttribute
	0.02%	95.851us	2	47.925us	32.901us	62.950us	cudaMemcpy
	0.00%	13.111us	1	13.111us	13.111us	13.111us	cuDeviceGetName
	0.00%	5.3620us	1	5.3620us	5.3620us	5.3620us	cuDeviceGetPCIBusId
	0.00%	4.2010us	1	4.2010us	4.2010us	4.2010us	cuDeviceTotalMem
	0.00%	1.4050us	3	468ns	193ns	946ns	cuDeviceGetCount
	0.00%	837ns	2	418ns	163ns	674ns	cuDeviceGet
	0.00%	539ns	1	539ns	539ns	539ns	cuModuleGetLoadingMode
	0.00%	257ns	1	257ns	257ns	257ns	cuDeviceGetUuid

Ядро finalRes занимает 97.78% общего времени на GPU с временем выполнения 240.56 мс за один вызов. Это основной этап программы, требующий оптимизации, так как на него приходится подавляющая часть вычислений. Ядро prefixScan, отвечающее за выполнение префиксной суммы, занимает 1.83% времени (4.511 мс на три вызова). Ядро addScan, выполняющее корректировку промежуточных данных, использует 0.25% времени (622.96 мкс на два вызова). На инициализацию массива initRes затрачено 0.12% времени (295.13 мкс за один вызов), что не является узким местом программы. Ядро hist, выполняющее подсчёт гистограммы, также занимает лишь 0.01% времени (26.944 мкс).

Из API-вызовов значительное время (43.65%) затрачивается на выделение памяти с помощью cudaMalloc, которое занимает 191.35 мс на 8 вызовов. cudaMemcpy (копирование данных между устройством и хостом) занимает 0.02% времени (95.851 мкс на два вызова), что свидетельствует о низких затратах на передачу данных. Остальные вызовы API, такие как cudaLaunchKernel и синхронизация через cudaDeviceSynchronize, вносят небольшой вклад в общее время выполнения.

Используем ключ --print-gpu-trace, чтобы получить подробные данные по каждому ядру и узнать, какое ядро занимает больше всего времени, можно использовать (n = 10^4).

Start	Duration	Grid Size	Block Size	Regs*	SSMem*	DSMem*	Size	Throughput	SrcMemType	DstMemType	Device	Context	Stream
359.61ms	6.4640us	-	-	-	-	-	39.063KB	5.7631GB/s	Pageable	Device	Tesla T4 (0)	1	7
360.13ms	296.38us	(1024 1 1)	(1024 1 1)	16	0B	0B	-	-	-	-	Tesla T4 (0)	1	7
360.45ms	26.815us	(1024 1 1)	(1024 1 1)	16	0B	0B	-	-	-	-	Tesla T4 (0)	1	7
360.50ms	4.4574ms	(1024 1 1)	(1024 1 1)	26	4.0000KB	4.1250KB	-	-	-	-	Tesla T4 (0)	1	7
365.01ms	30.079us	(1024 1 1)	(1024 1 1)	26	4.0000KB	4.1250KB	-	-	-	-	Tesla T4 (0)	1	7
365.07ms	25.695us	(1024 1 1)	(1024 1 1)	26	4.0000KB	4.1250KB	-	-	-	-	Tesla T4 (0)	1	7
365.11ms	26.207us	(1024 1 1)	(1024 1 1)	24	0B	0B	-	-	-	-	Tesla T4 (0)	1	7
365.16ms	593.88us	(1024 1 1)	(1024 1 1)	24	0B	0B	-	-	-	-	Tesla T4 (0)	1	7
365.81ms	240.01ms	(32 32 1)	(32 32 1)	18	0B	0B	-	-	-	-	Tesla T4 (0)	1	7
605.88ms	4.8320us	-	-	-	-	-	39.063KB	7.7096GB/s	Device	Pageable	Tesla T4 (0)	1	7

Передача данных между CPU и GPU демонстрирует высокий пропускной поток (до 7.7096 GB/s), при этом общий объём данных составляет 39.063 KB. Это указывает на эффективное использование PCIe-соединения для передачи данных. Операции CUDA выполняются с различными конфигурациями сетки и блоков. Большинство операций используют одномерную сетку с конфигурацией (1024 1 1) и размер блока (1024 1 1), что позволяет задействовать большое количество потоков для параллельных вычислений.
Для операции finalRes используется двухмерная структура сетки и блоков, что позволяет распределить вычисления по двум измерениям, оптимизируя операции записи данных. Количество регистров на поток варьируется от 16 до 26, что свидетельствует об умеренном использовании регистров, минимизирующем переполнения в локальную память. Использование разделяемой памяти фиксируется только в нескольких операциях, где она используется эффективно (до 4 KB на блок).

Выводы

Реализована сортировка чисел на GPU с использованием метода гистограммы и операции префиксной суммы. Для вычислений применены атомарные операции для подсчета частот чисел, а также эффективное сканирование с использованием разделяемой памяти для минимизации обращений к глобальной памяти. Для параллельной обработки массивов использована трехмерная структура сетки потоков, обеспечивающая равномерное распределение вычислений между потоками CUDA. Производительность программы исследована с помощью утилиты nvprof, что позволило выявить основные время затратные этапы алгоритма и подтвердить высокую эффективность реализации по сравнению с последовательными методами сортировки.