МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №*1*
по курсу «Параллельные и распределенные вычисления»

Освоение программного обеспечения для работы с технологией CUDA.
Примитивные операции над векторами.

Выполнил: *Д.А. Коряков*
Группа: *М8О-211М-23*
Преподаватель: А.Ю. Морозов

Москва, 2024

Условие

Цель работы. Ознакомление и установка программного обеспечения для работы с программно-аппаратной архитектурой параллельных вычислений (CUDA). Реализация одной из примитивных операций над векторами.

В качестве вещественного типа данных необходимо использовать тип данных double. Все результаты выводить с относительной точностью 10^{-10}. Ограничение: $n < 2^{25}$.

Вариант 2. Вычитание векторов.

Входные данные. На первой строке задано число n -- размер векторов. В следующих 2-х строках, записано по n вещественных чисел -- элементы векторов.

Выходные данные. Необходимо вывести n чисел -- результат вычитания исходных векторов.

Пример:

Входной файл	Выходной файл
3 1 2 3 4 5 6	-3.0000000000e+00 -3.0000000000e+00 -3.0000000000e+00

Программное и аппаратное обеспечение

IDE: CLion 2024.2.3 Build #CL-242.23726.125, built on October 27, 2024

gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Wed_Nov_22_10:17:15_PST_2023
Cuda compilation tools, release 12.3, V12.3.107
Build cuda_12.3.r12.3/compiler.33567101_0

GPU:

Compute capability	: 8.6
Name	: NVIDIA RTX A6000
Total Global Memory	: 51032686592
Shared memory per block	: 49152
Max threads per block	: (1024, 1024, 64)
Max block	: (2147483647, 65535, 65535)
Total constant memory	: 65536
Multiprocessors count	: 84

CPU:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Address sizes:	46 bits physical, 48 bits virtual
Byte Order:	Little Endian
CPU(s):	32
On-line CPU(s) list:	0-31
Vendor ID:	GenuineIntel
Model name:	Intel(R) Xeon(R) Silver 4215R CPU @ 3.20GHz
CPU family:	6
Model:	85
Thread(s) per core:	2
Core(s) per socket:	8
Socket(s):	2
Stepping:	7
CPU max MHz:	4000.0000
CPU min MHz:	1000.0000
BogoMIPS:	6400.00
Flags:	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl smx est tm2 ssse3 s dbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3 invpcid_single intel_ppin ssbd mba ibrs ibpb stibp ibrs_enhanced fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm mpx rdt _a avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm _total cqm_mbm_local dtherm ida arat pln pts pku ospke avx512_vnni md_clear flush_l1d arch_capabilities
Caches (sum of all):	
L1d:	512 KiB (16 instances)
L1i:	512 KiB (16 instances)
L2:	16 MiB (16 instances)

```
        L3:                     22 MiB (2 instances)
NUMA:
        NUMA node(s):           2
        NUMA node0 CPU(s):      0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
        NUMA node1 CPU(s):      1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
Vulnerabilities:
        Gather data sampling:   Mitigation; Microcode
        Itlb multihit:          KVM: Mitigation: VMX unsupported
        L1tf:                   Not affected
        Mds:                    Not affected
        Meltdown:               Not affected
        Mmio stale data:        Mitigation; Clear CPU buffers; SMT vulnerable
        Retbleed:               Mitigation; Enhanced IBRS
        Spec rstack overflow:   Not affected
        Spec store bypass:      Mitigation; Speculative Store Bypass disabled via prctl
and seccomp
        Spectre v1:             Mitigation; usercopy/swapgs barriers and __user pointer
sanitization
        Spectre v2:             Mitigation; Enhanced IBRS, IBPB conditional, RSB
filling, PBRSB-eIBRS SW sequence
        Srbds:                  Not affected
        Tsx async abort:        Mitigation; TSX disabled
```

RAM:

	total	used	free	shared	buff/cache	available
Mem:	62Gi	9.6Gi	12Gi	10Mi	40Gi	52Gi
Swap:	8.0Gi	4.0Mi	8.0Gi			

Storage:

Filesystem	Size	Used	Avail	Use%	Mounted on
tmpfs	6.3G	3.5M	6.3G	1%	/run
/dev/mapper/ubuntu--vg-ubuntu--lv	1.9T	1.6T	182G	90%	/
tmpfs	32G	416K	32G	1%	/dev/shm
tmpfs	5.0M	4.0K	5.0M	1%	/run/lock
/dev/nvme0n1p2	2.0G	340M	1.5G	19%	/boot
/dev/nvme0n1p1	1.1G	6.1M	1.1G	1%	/boot/efi
tmpfs	6.3G	120K	6.3G	1%	/run/user/1000
tmpfs	6.3G	68K	6.3G	1%	/run/user/1001

OS:

Linux sedanllm 5.15.0-97-generic #107-Ubuntu SMP Wed Feb 7 13:26:48 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux

Метод решения

Был написан относительно простой код с вычитанием двух векторов. Были просмотрены лекции. Данный алгоритм предназначен для вычисления поэлементной разности двух векторов на GPU с использованием параллельных вычислений CUDA. Программа выделяет память для массивов на CPU (хост) и GPU (устройство), загружает данные с CPU на GPU, выполняет вычисления в GPU и передает результаты обратно на CPU.

Основная цель алгоритма — сократить время выполнения за счет использования вычислительных ресурсов GPU, что особенно эффективно для крупных векторов.

Архитектура решения:

1. Выделение памяти для данных на CPU и GPU.
2. Передача данных с CPU на GPU.
3. Запуск CUDA-ядра, которое вычисляет поэлементную разность двух векторов.
4. Передача результатов обратно на CPU.
5. Освобождение выделенной памяти.

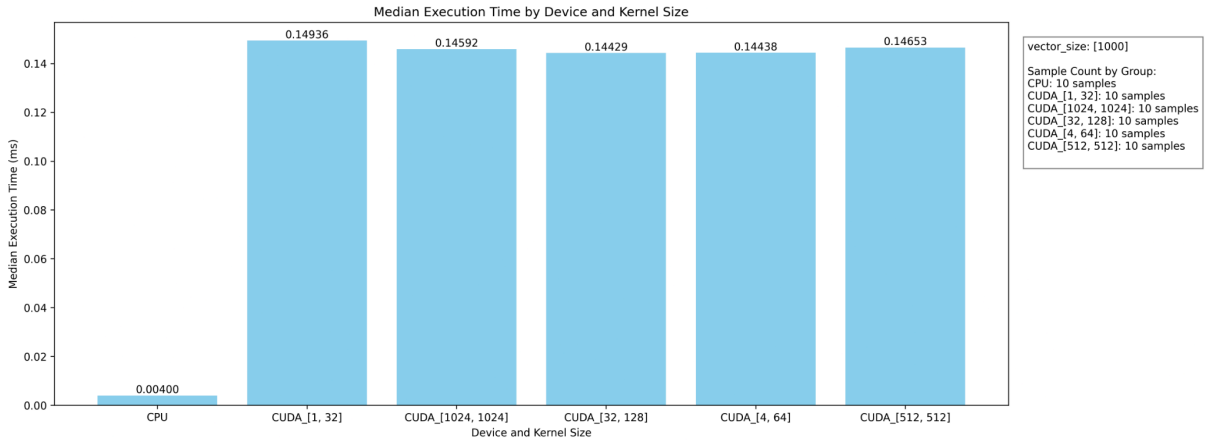
Описание программы

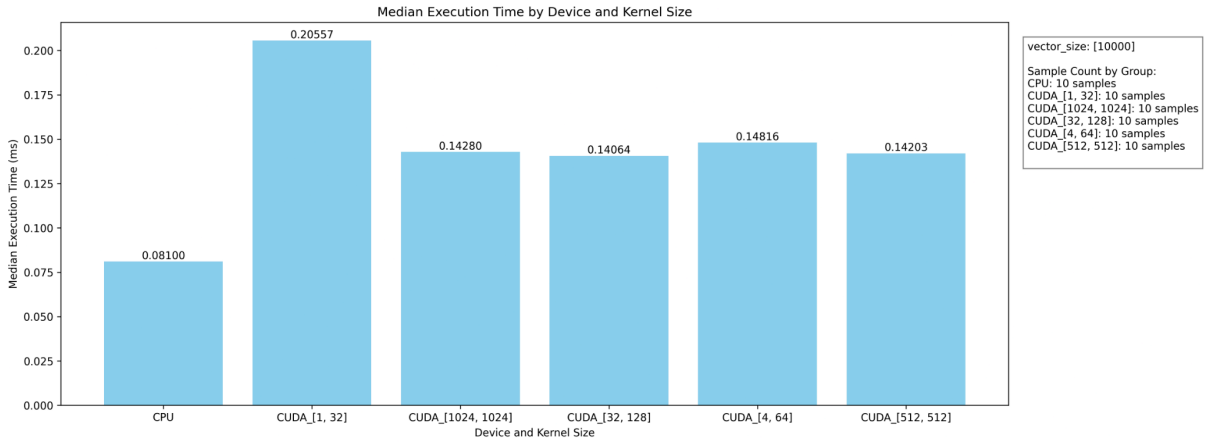
Программа состоит из:

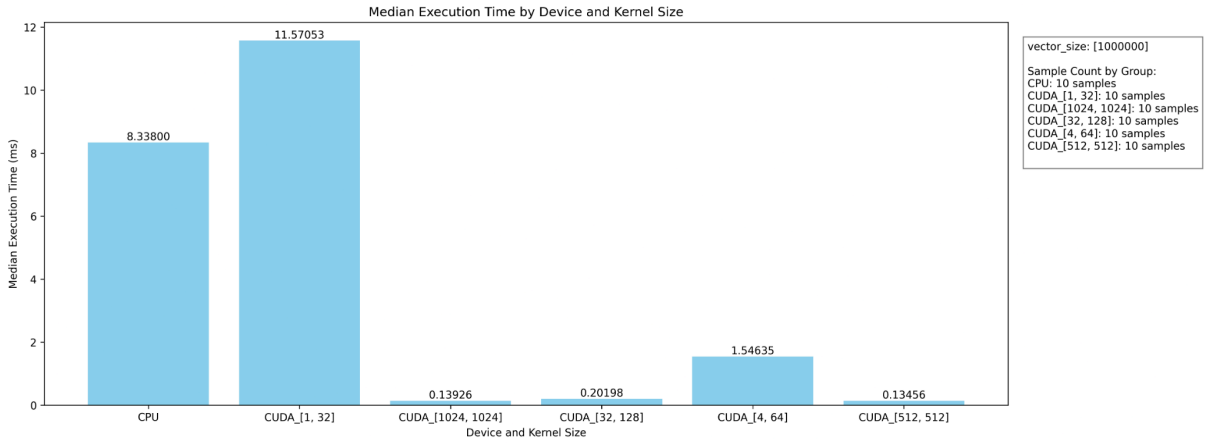
1. **Основных функций и макросов**:
    - `CSC`: Макрос для проверки ошибок CUDA-вызовов.
    - `ALLOCATE_VECTOR_CPU`: Макрос для выделения памяти для векторов на CPU и проверки успешности выделения.
2. **CUDA ядро (`kernel`)**:
    - Основное CUDA-ядро `kernel` выполняет поэлементную разность двух массивов `arr_1` и `arr_2`, сохраняя результат в `arr_3`.
    - Алгоритм использует индексы блоков и потоков для распределения вычислений на несколько потоков. Каждый поток обрабатывает элемент в массиве, продвигаясь с шагом, равным общему количеству потоков (`blockDim.x * gridDim.x`).
3. **Главная функция (`main`)**:
    - Пользователь вводит размер вектора `n`.
    - Используются макросы для выделения памяти для векторов на CPU.
    - Память для массивов также выделяется на GPU.
    - Массивы заполняются данными, затем передаются на устройство.
    - CUDA-ядро `kernel` запускается с конфигурацией `512 блоков x 512 потоков` для выполнения вычислений.

- Время выполнения замеряется с использованием CUDA событий (`cudaEventRecord`).
- По завершении вычислений результаты копируются обратно на CPU, и память освобождается.

Результаты







Для проведения экспериментов, были написаны python скрипты, в дальнейшем они могут быть использованы для последующих ЛР. https://github.com/KoryakovDmitry/cuda-mpi-openmp

Выводы

Данный алгоритм предназначен для вычисления разности двух векторов. Это может быть полезно при выполнении численных расчетов, обработке данных, графических вычислениях, анализе сигналов и в других задачах, требующих элементарных операций над векторами, таких как линейная алгебра и обработка массивов данных.

Типовые задачи

1. Вычисление разности между набором сигналов или временных рядов.
2. Поддержка операций над массивами данных в численных и графических вычислениях.
3. Элементарные операции в системах машинного обучения или при обработке изображений.

Для выполнения этого задания я реализовал собственный вариант алгоритма на CPU без использования технологий CUDA. В результате CPU оказался значительно быстрее для небольших векторов. Также можем заметить, что при увеличении размера вектора – CUDA выигрывает, что логично. GPU показывает высокую производительность на больших данных за счёт параллелизма, но для небольших данных накладные расходы делают CPU более предпочтительным.