МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторная работа №*1*
по курсу «*Программирование графических процессоров*»
Освоение программного обеспечения для работы с технологией CUDA.

Примитивные операции над векторами.

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

Москва, *2025*

Условие

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

Вариант №4: Поэлементное нахождение минимума 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 поэлементного нахождения минимума исходных векторов.

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

Алгоритм решения:

1) Подготовка данных на CPU:
    a) Со стандартного ввода считывается количество элементов в массивах
    b) Выделяется память на хосте (CPU) и проверяется успешность выделения.
    c) Со стандартного ввода считываются два массива чисел.
2) Копирование данных на GPU:
    a) Выделяется память на GPU (cudaMalloc).
    b) Для параллельной обработки массивы копируются в глобальную память устройства с помощью cudaMemcpy.
3) Параллельная обработка на GPU:
    a) Используется схема с шагом offset, которая позволяет одному потоку обрабатывать несколько элементов, если количество элементов больше числа потоков.
    b) Функция ядра (kernel) выполняет операцию fminf для каждого назначенного элемента.
4) Синхронизация и возврат данных:
    a) После завершения kernel все потоки синхронизируются (cudaDeviceSynchronize).
    b) Результаты копируются обратно на CPU и выводятся на экран.
5) Очистка ресурсов:
    a) Освобождается память на GPU и CPU.
    b) Обрабатываются ошибки CUDA через макрос CSC(call).

Архитектура программы:

Программа построена по клиент-серверной модели памяти CPU–GPU:

1) CPU: Чтение данных, выделение памяти, контроль ошибок, вывод результатов.
2) GPU: Параллельное вычисление минимума элементов массива через kernel. Потоки организованы в блоки и сетку, каждый поток выполняет вычисления на своей части массива.
3) Взаимодействие CPU и GPU: Данные передаются на GPU и обратно через глобальную память с использованием cudaMemcpy. Время выполнения kernel измеряется с помощью событий CUDA (cudaEvent_t).

Для выполнения данной лабораторной работы были использованы следующие источники информации:

1) https://vk.com/video5017467_456239107
2) https://vk.com/video5017467_456239106

Описание программы

Основные типы данных:

1) doube - вещественный тип данных, используется для хранение введенных вещественных чисел
2) int — знаковый целочисленный тип данных, используется для хранения значения n, которое задает размер массива для входных данных.
3) cudaEvent_t - Специальный тип для событий CUDA, используется для измерения времени выполнения на GPU.

Основные функции и макросы:

1) Макрос CSC(call) предназначен для проверки ошибок CUDA. Любой вызов CUDA, обёрнутый в CSC(...), автоматически проверяет результат. Если есть ошибка, выводится сообщение и программа завершает выполнение.
2) CUDA-ядро kernel принимает указатель на массивы и их размер. Вычисляет глобальный индекс текущего потока и шаг для обработки потоков элементов. Берет минимум чисел из двух массивов по каждому индексу и сохраняет результат в новый массив по тому же индексу.

Результаты

1) Зависимость времени выполнения программы от количества используемых потоков (для тестов использовались два вектора по 10 миллионов чисел):

Потоки	Время (в мс)
1×32	109
32×32	5
1024×1024	4

2) Сравнение программы на CUDA с 32×32 потоками и программы на CPU с одним потоком:

Размер векторов	Время на CUDA (в мс)	Время на CPU (в мс)
10 млн	5	40

25 млн	14	92
50 млн	27	187

Выводы

Реализованный алгоритм вычисляет поэлементный минимум двух массивов чисел с плавающей точкой. Такие операции широко применяются в научных и инженерных вычислениях, где требуется нахождение нормы векторов или подготовка данных для дальнейших вычислений.
Одним из типовых задач, где может пригодиться программа это подсчёт абсолютных значений в численных методах (например, при вычислении нормы вектора).
На CPU алгоритм реализуется просто, используя стандартные функции fminf и обычный цикл. Программирование несложное, требуются базовые знания работы с массивами и функциями стандартной библиотеки.
На GPU (CUDA) добавляется сложность: нужно выделять память на устройстве, копировать данные, запускать ядро и синхронизировать потоки. Требуются знания параллельного программирования и архитектуры CUDA.