МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторная работа №*1*
по курсу «Програ*ммирование графических процессоров*»

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

Выполнил: Е. С. Кострюков
Группа: М8О-407Б-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.
В качестве вещественного типа данных необходимо использовать тип данных double. Все результаты выводить с относительной точностью 10^{-10}.
Ограничение: $n < 2^{25}$.

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.

Пример:

Входной файл	Выходной файл
3 1 5 3 4 2 6	1.0000000000e+00 2.0000000000e+00 3.0000000000e+00

Программное и аппаратное обеспечение

Compute capability	7.5
Name	Tesla T4
Total Global Memory	15828320256
Shared memory per block	49152
Registers per block	65536
Warp size	32
Max threads per block	(1024, 1024, 64)
Max block	(2147483647, 65535, 65535)
Total constant memory	65536
Multiprocessors count	40

Метод решения

В разработанной программе сначала в динамической памяти создаются два массива длиной n, которые заполняются входными значениями. После этого в памяти графического ускорителя выделяется пространство под соответствующие массивы и результирующий массив для хранения ответа. Считанные данные копируются из оперативной памяти на видеокарту.

Далее запускается CUDA-ядро, в котором каждый поток обрабатывает свою часть данных: для каждой позиции вектора вычисляется минимум между элементами двух входных массивов. Результаты этой операции записываются в выходной массив, расположенный в памяти GPU.
После завершения вычислений готовый массив с результатами копируется обратно в память центрального процессора, где происходит его вывод в требуемом формате с заданной точностью. В конце работы программы освобождаются все выделенные участки памяти как на стороне CPU, так и на стороне GPU.

Описание программы

Программа реализована на языке C с использованием технологии CUDA. Она состоит из основной функции main и вычислительного ядра.
В main происходит ввод данных: сначала считывается размер векторов n, затем в динамической памяти на CPU выделяются два массива типа double, которые заполняются элементами исходных векторов. После этого в памяти GPU выделяются три массива: два для копирования входных данных и один для сохранения результата.
CUDA-ядро выполняет поэлементное нахождение минимума двух векторов. Для этого каждому потоку вычисляется свой глобальный индекс, на основе которого он обрабатывает элемент векторов. При этом используется шаг, позволяющий одному потоку при необходимости обработать несколько элементов массива, если их общее количество превышает число потоков. Таким образом достигается равномерное распределение нагрузки между потоками.
После выполнения ядра результат работы копируется обратно в память CPU и выводится. Завершающим этапом является освобождение динамически выделенной памяти как на CPU, так и на GPU.

Результаты

```
1000
GPU timings (in ms):
|grids: 1|blocks: 32|time: 0.164640 ms|
|grids: 1|blocks: 64|time: 0.017280 ms|
|grids: 1|blocks: 128|time: 0.013664 ms|
|grids: 1|blocks: 256|time: 0.011328 ms|
|grids: 1|blocks: 512|time: 0.009568 ms|
|grids: 1|blocks: 1024|time: 0.010176 ms|
|grids: 8|blocks: 32|time: 0.012288 ms|
|grids: 8|blocks: 64|time: 0.008160 ms|
|grids: 8|blocks: 128|time: 0.008192 ms|
|grids: 8|blocks: 256|time: 0.007584 ms|
|grids: 8|blocks: 512|time: 0.008000 ms|
|grids: 8|blocks: 1024|time: 0.008192 ms|
|grids: 64|blocks: 32|time: 0.006784 ms|
```

grids: 64	blocks: 64	time: 0.006624 ms
grids: 64	blocks: 128	time: 0.006880 ms
grids: 64	blocks: 256	time: 0.008192 ms
grids: 64	blocks: 512	time: 0.008160 ms
grids: 64	blocks: 1024	time: 0.008384 ms
grids: 512	blocks: 32	time: 0.006816 ms
grids: 512	blocks: 64	time: 0.007904 ms
grids: 512	blocks: 128	time: 0.007616 ms
grids: 512	blocks: 256	time: 0.008256 ms
grids: 512	blocks: 512	time: 0.010240 ms
grids: 512	blocks: 1024	time: 0.013600 ms
grids: 1024	blocks: 32	time: 0.008160 ms
grids: 1024	blocks: 64	time: 0.008480 ms
grids: 1024	blocks: 128	time: 0.007808 ms
grids: 1024	blocks: 256	time: 0.008192 ms
grids: 1024	blocks: 512	time: 0.011968 ms
grids: 1024	blocks: 1024	time: 0.020096 ms
CPU: 0.013000 ms

100000
GPU timings (in ms):
grids: 1	blocks: 32	time: 1.676960 ms
grids: 1	blocks: 64	time: 0.808992 ms
grids: 1	blocks: 128	time: 0.416832 ms
grids: 1	blocks: 256	time: 0.217408 ms
grids: 1	blocks: 512	time: 0.120128 ms
grids: 1	blocks: 1024	time: 0.114720 ms
grids: 8	blocks: 32	time: 0.207520 ms
grids: 8	blocks: 64	time: 0.110624 ms
grids: 8	blocks: 128	time: 0.059488 ms
grids: 8	blocks: 256	time: 0.034816 ms
grids: 8	blocks: 512	time: 0.023520 ms
grids: 8	blocks: 1024	time: 0.022400 ms
grids: 64	blocks: 32	time: 0.032896 ms
grids: 64	blocks: 64	time: 0.020480 ms
grids: 64	blocks: 128	time: 0.014880 ms
grids: 64	blocks: 256	time: 0.012480 ms
grids: 64	blocks: 512	time: 0.012256 ms
grids: 64	blocks: 1024	time: 0.014336 ms
grids: 512	blocks: 32	time: 0.011680 ms
grids: 512	blocks: 64	time: 0.011776 ms
grids: 512	blocks: 128	time: 0.010976 ms
grids: 512	blocks: 256	time: 0.012160 ms
grids: 512	blocks: 512	time: 0.013312 ms

grids: 512	blocks: 1024	time: 0.018464 ms
grids: 1024	blocks: 32	time: 0.013856 ms
grids: 1024	blocks: 64	time: 0.012352 ms
grids: 1024	blocks: 128	time: 0.012288 ms
grids: 1024	blocks: 256	time: 0.012608 ms
grids: 1024	blocks: 512	time: 0.016544 ms
grids: 1024	blocks: 1024	time: 0.025312 ms
CPU: 0.690000 ms

30000000
GPU timings (in ms):
grids: 1	blocks: 32	time: 480.577728 ms
grids: 1	blocks: 64	time: 181.399750 ms
grids: 1	blocks: 128	time: 91.928574 ms
grids: 1	blocks: 256	time: 46.682335 ms
grids: 1	blocks: 512	time: 24.839584 ms
grids: 1	blocks: 1024	time: 14.258176 ms
grids: 8	blocks: 32	time: 45.492226 ms
grids: 8	blocks: 64	time: 23.449280 ms
grids: 8	blocks: 128	time: 12.932032 ms
grids: 8	blocks: 256	time: 7.290304 ms
grids: 8	blocks: 512	time: 4.264960 ms
grids: 8	blocks: 1024	time: 3.095456 ms
grids: 64	blocks: 32	time: 6.863072 ms
grids: 64	blocks: 64	time: 4.033600 ms
grids: 64	blocks: 128	time: 3.070368 ms
grids: 64	blocks: 256	time: 2.962112 ms
grids: 64	blocks: 512	time: 2.988128 ms
grids: 64	blocks: 1024	time: 2.960512 ms
grids: 512	blocks: 32	time: 3.151872 ms
grids: 512	blocks: 64	time: 3.068928 ms
grids: 512	blocks: 128	time: 2.957536 ms
grids: 512	blocks: 256	time: 2.977824 ms
grids: 512	blocks: 512	time: 2.957632 ms
grids: 512	blocks: 1024	time: 2.938880 ms
grids: 1024	blocks: 32	time: 3.089856 ms
grids: 1024	blocks: 64	time: 2.988640 ms
grids: 1024	blocks: 128	time: 2.969792 ms
grids: 1024	blocks: 256	time: 2.953184 ms
grids: 1024	blocks: 512	time: 2.936928 ms
grids: 1024	blocks: 1024	time: 2.925792 ms
CPU: 205.493000 ms

При проведении экспериментов было выполнено сравнение времени работы программы при различных конфигурациях сетки и блоков. Замеры показали, что для небольших входных данных увеличение числа потоков не даёт выигрыша, так как накладные расходы на запуск GPU сопоставимы с временем вычислений. Однако при росте размеров векторов параллелизм начинает проявляться, и время работы на GPU заметно сокращается. При этом последовательная реализация на CPU для больших массивов оказывается существенно медленнее, тогда как GPU обеспечивает ускорение в десятки раз за счёт одновременной обработки элементов тысячами потоков.

Выводы

В результате данной работы было установлено и освоено программное обеспечение для работы с архитектурой параллельных вычислений CUDA. Реализован алгоритм поэлементного нахождения минимума двух векторов с использованием GPU.

В процессе разработки основной сложностью стало правильное управление памятью между CPU и GPU, а также выбор параметров сетки и блоков для обеспечения эффективного параллелизма. Проведённые эксперименты показали, что при малых размерах массивов использование GPU не даёт значительного выигрыша из-за накладных расходов, но при увеличении входных данных параллельная реализация обеспечивает ускорение в десятки раз по сравнению с последовательным вариантом.