

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторная работа №1
по курсу «Основы CUDA»

Программирование графических процессоров

Выполнил: *Ши Хуэй*

Группа: 638

Преподаватели: А.Ю. Морозов,
Е.Е. Заяц

Москва, 2025

Условие

Цель работы — освоить основы программирования на CUDA путём реализации примитивной операции над векторами.

Задача: реализовать программу, выполняющую **вычитание двух вещественных векторов** одинаковой длины.

Вариант 2 - вычитание векторов

Программное и аппаратное обеспечение

- **Графический процессор:** NVIDIA Tesla T4
 - Compute capability: 7.5
 - Общая глобальная память: 15 ГБ (15360 MiB)
 - Shared memory per block: 49152 байт
 - Registers per block: 65536
 - Max threads per block: (1024, 1024, 64)
 - Max blocks: (2147483647, 65535, 65535)
 - Constant memory: 65536 байт
 - Количество мультипроцессоров: 40
- **Процессор:** Виртуальная машина Google Colab (2 vCPU Intel Xeon, архитектура x86_64)
- **Оперативная память:** 12–16 ГБ (в зависимости от сессии Colab)
- **Жёсткий диск (виртуальный):** ~100 ГБ
- **Программное обеспечение:**
 - Среда выполнения: Google Colab (облачная среда на базе Linux)
 - ОС: Ubuntu 20.04/22.04 (облачный сервер Colab)
 - Компилятор CUDA: nvcc 12.5
 - Драйвер NVIDIA: 550.54.15
 - Версия CUDA Runtime: 12.4
 - Дополнительные инструменты: Python 3.10, Jupyter/Colab notebooks

Метод решения

Для решения задачи используется параллельная модель CUDA.

Каждый элемент двух входных векторов обрабатывается независимо, поэтому операция вычитания легко распараллеливается.

В программе реализовано ядро:

```
__global__ void vec_sub(double* a, double* b, double* c, size_t n) {  
    size_t offset = gridDim.x * blockDim.x;  
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;  
    for (; i < n; i += offset) c[i] = a[i] - b[i];  
}
```

Здесь применяется цикл с шагом offset (grid-stride loop), что позволяет фиксировать число блоков и потоков независимо от размера входных данных.

Память для векторов выделяется на GPU с помощью cudaMalloc, данные копируются с хоста на устройство, затем выполняется ядро и результат возвращается обратно.

Описание программы

На хосте (CPU) организован ввод данных: сначала число n , затем два вектора длины n . Выделяется память на устройстве (cudaMalloc), исходные данные копируются на GPU (cudaMemcpy).

Запускается ядро `vec_sub`, которое поэлементно вычисляет разность векторов.

Используется схема `grid-stride loop`, что позволяет фиксировать число блоков и потоков независимо от размера входных данных.

Результат копируется обратно на хост и выводится с требуемой точностью (`std::setprecision(10)`).

После завершения работы память на устройстве освобождается (`cudaFree`).

Результаты

Для сравнения с GPU была реализована последовательная версия программы на C++, выполняющая поэлементное вычитание векторов с использованием стандартной операции вычитания (`c[i] = a[i] - b[i];`).

```
for (size_t i = 0; i < n; ++i) {  
    c[i] = a[i] - b[i];  
}
```

В таблице ниже приведены экспериментальные данные.

Конфигурация <<<Blocks, Threads>>>

Время *GPU(kernel)* показывает реальное время вычислений на устройстве (менее 1 мс даже при больших n).

Время *GPU(total)* включает расходы на выделение/освобождение памяти и инициализацию контекста в среде Google Colab, поэтому оно значительно выше (сотни миллисекунд).

Ускорение = $\text{Время CPU} / \text{Время GPU(kernel)}$

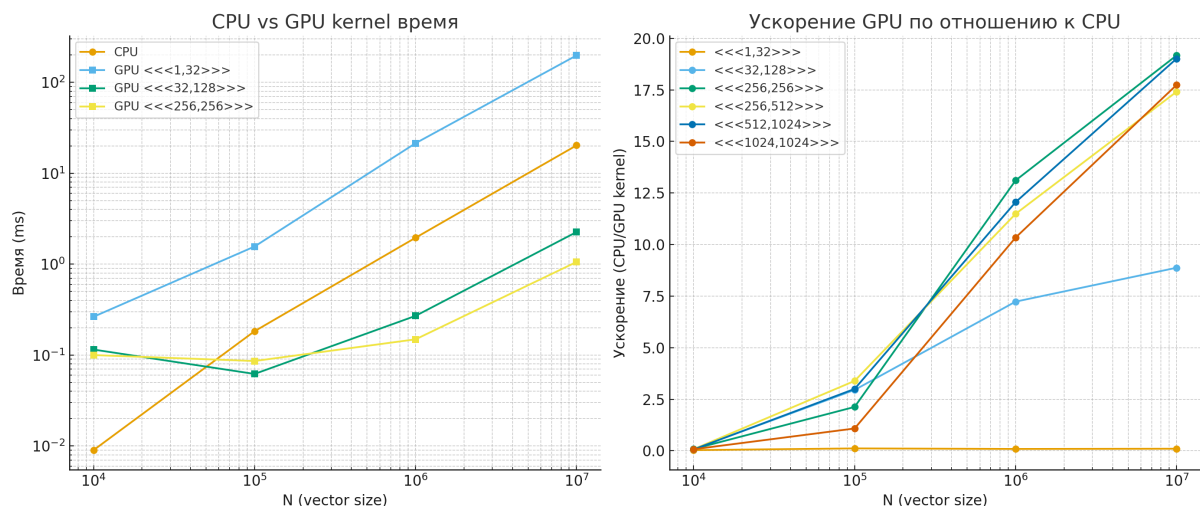
N (размер вектора)	Конфигурация	Время GPU(kernel), мс	Время GPU(total), мс	Время CPU, мс	Ускорение
10^4	<<<1, 32>>>	0.264	202.289	0.009	0.034
	<<<32, 128>>>	0.115	207.313		0.078
	<<<256, 256>>>	0.100	168.300		0.090
	<<<256, 512>>>	0.116	181.534		0.078
	<<<512, 1024>>>	0.133	214.681		0.068
	<<<1024, 1024>>>	0.117	171.929		0.077
10^5	<<<1, 32>>>	1.564	193.738	0.183	0.117
	<<<32, 128>>>	0.062	171.799		2.952
	<<<256, 256>>>	0.086	181.175		2.128
	<<<256, 512>>>	0.054	174.004		3.389
	<<<512, 1024>>>	0.061	180.485		3.000
	<<<1024, 1024>>>	0.169	200.206		1.083
10^6	<<<1, 32>>>	21.413	224.453	1.953	0.091
	<<<32, 128>>>	0.270	204.337		7.233
	<<<256, 256>>>	0.149	189.351		13.110
	<<<256, 512>>>	0.170	176.355		11.488

10 ⁷	<<<512,1024>>>	0.162	193.268	20.279	12.056
	<<<1024,1024>>>	0.189	214.667		10.339
	<<<1, 32>>>	198.039	529.548		0.102
	<<<32, 128>>>	2.258	323.518		8.876
	<<<256, 256>>>	1.058	287.352		19.170
	<<<256,512>>>	1.164	328.910		17.410
	<<<512,1024>>>	1.066	241.145		19.014
	<<<1024,1024>>>	1.143	330.659		17.740

Сравнение времени и ускорения

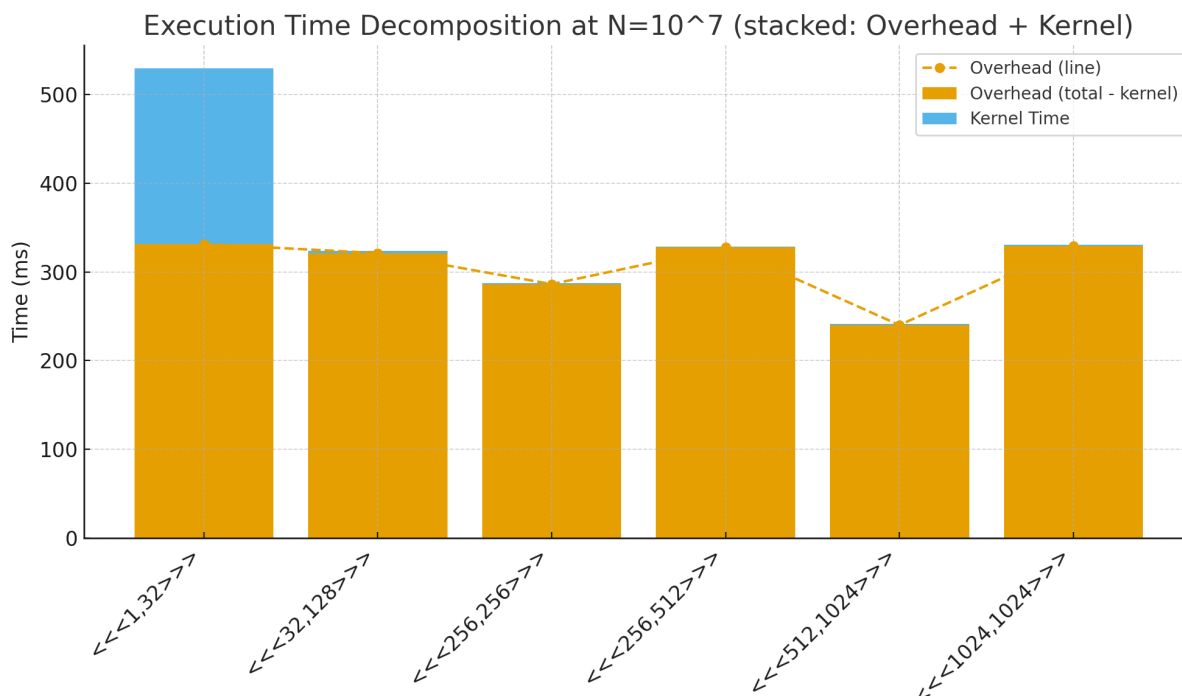
На левой диаграмме показано время работы ядра GPU в сравнении с CPU при различных размерах вектора. Используется логарифмическая шкала, чтобы отразить широкий диапазон значений.

На правой диаграмме приведены значения ускорения (отношение времени CPU к времени GPU kernel) для разных конфигураций сетки и блока.



Анализ временных затрат при $N=10^7$.

На рисунке приведено разложение общего времени выполнения на две составляющие: собственно вычисления на GPU (Kernel time, голубым) и накладные расходы (Overhead, оранжевым), включающие выделение/освобождение памяти и копирование данных между CPU и GPU.



Выводы

1. Реализованный алгоритм поэлементного вычитания векторов показал корректную работу как на CPU, так и на GPU.
2. Сравнение показало, что при малых размерах задач (например, $N=10^4$) затраты на запуск ядра и передачу данных делают использование GPU неэффективным.
3. Начиная с $N=10^5$, GPU начинает демонстрировать ускорение, которое растёт с увеличением размера задачи. При $N=10^7$ достигнуто ускорение порядка 18–19 раз относительно CPU (по времени ядра).
4. Время GPU(total) значительно выше времени GPU(kernel), так как включает накладные расходы на выделение и освобождение памяти, а также копирование данных между CPU и GPU. Это подтверждает, что при практическом применении необходимо минимизировать количество операций копирования и эффективно управлять памятью.
5. Наиболее эффективные конфигурации потоков для данной задачи оказались средние и крупные (например, <<<256,256>>> или <<<512,1024>>>), обеспечивающие стабильное ускорение при больших входных данных.
6. Алгоритм может быть использован как базовый строительный блок для более сложных операций линейной алгебры и обработки больших массивов данных.