

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторная работа №2
по курсу «Программирование графических процессоров»

Классификация и кластеризация изображений на GPU.

Выполнил: Е. С. Кострюков
Группа: М8О-407Б-22
Преподаватели: А.Ю. Морозов,
Е.Е. Заяц

Москва, 2025

Условие

Цель работы: научиться использовать GPU для классификации и кластеризации изображений. Использование константной памяти и одномерной сетки потоков.

Ограничение: $nc \leq 32$, $n_{pj} \leq 2^{19}$, $w * h \leq 4 * 10^8$.

Вариант 4. Метод спектрального угла.

Входные данные. На первой строке задается путь к исходному изображению, на второй, путь к конечному изображению. На следующей строке, число nc -- количество классов. Далее идут nc строчек описывающих каждый класс. В начале j -ой строки задается число n_{pj} – количество пикселей в выборке, за ним следуют n_{pj} пар чисел -- координаты пикселей выборки.

Пример:

Входной файл	hex: in.data	hex: out.data
in.data	03000000 03000000	03000000 03000000
out.data	A2DF4C00 F7C9FE00 9ED84500	A2DF4C01 F7C9FE00 9ED84501
2	B4E85300 99D14D00 92DD5600	B4E85301 99D14D01 92DD5601
4 1 2 1 0 2 2 2 1	A9E04C00 F7D1FA00 D4D0E900	A9E04C01 F7D1FA00 D4D0E900
4 0 0 0 1 1 1 2 0		

Входной файл	hex: out.data
in.data	08000000 08000000
out.data	D2E27502 CFF65201 D3ED5701 D6E76902
5	C8F35B01 8E168200 CFF45001 AE977604
4 5 0 0 2 6 1 1 1	D3DC7102 7D1E7B00 AB9A8004 D9E58602
6 2 0 7 1 1 0 1 2 6 0 4 0	AB967E04 AE9D8004 87058200 D0F95B01
4 3 0 3 1 0 1 0 0	74148000 D0F55901 86136C00 85077400
4 0 3 6 2 5 2 7 2	D6E27702 D3609F03 D1609F03 CC5EA103
9 6 4 5 1 7 0 2 1 2 3 4 1 1 5	CC739D03 7C127F00 AA988804 AFA07D04
3 3 2 6	D0E37702 7D117A00 D6EB5901 D6E37C02 C9F85701 D655A103 D7EA7402 93127D00 D35BA403 D4DD7902 B0A18404 D6DE7502 D765A903 AD928404 D0D87C02 D7E97F02 CD509E03 CAF85201 CFF75601 CEF45E01 D0E86902 D1D17F02 AD928104 AFA18304 D4DB5C02 88077D00 C6F75701 7D127D00 A99A8E04 C8609E03 D15DA503 AB957E04 AE9A8004 79218100 D065A103 A99E9A04

Программное и аппаратное обеспечение

Аппаратная часть:

- **Процессор:** Intel(R) Xeon(R) CPU @ 2.00GHz (1 физическое, 2 логических ядра).

- **Оперативная память:** 12.67 GB.
- **Накопитель (жёсткий диск):** 73.59 GB (тип файловой системы - ext4).
- **Графический процессор (GPU):** NVIDIA Tesla T4
 - Compute Capability: 7.5.
 - Total Global Memory: 15 828 320 256 байт (~15.8 GB).
 - Shared memory per block: 49 152 байт.
 - Registers per block: 65 536.
 - Warp size: 32.
 - Max threads per block: (1024, 1024, 64).
 - Max grid size: (2147483647, 65535, 65535).
 - Total constant memory: 65 536 байт.
 - Multiprocessors count: 40.

Программная часть:

- **Операционная система:** Linux 6.6.105+ (x86_64, glibc 2.35).
- **Компилятор CUDA:** nvcc 12.5 (Cuda compilation tools, V12.5.82).
- **Интерпретатор Python:** 3.12.12.
- **Используемая IDE:** Google Colab.

Метод решения

В данной лабораторной работе реализуется алгоритм классификации пикселей изображения на основе метода спектрального угла. Основная идея метода заключается в сравнении спектрального вектора каждого пикселя с усреднёнными векторами классов и определении того класса, с которым пиксель образует минимальный угол. На первом этапе программа считывает входное изображение из бинарного файла заданного формата. В начале файла содержатся размеры изображения (ширина и высота), после чего следуют значения всех пикселей, представленных структурами типа uchar4_custom, содержащими компоненты цвета (R, G, B) и служебное поле A. Далее считывается количество классов и обучающая выборка: для каждого класса задано количество эталонных пикселей и их координаты.

После чтения данных на центральном процессоре выполняется вычисление усреднённого цветового вектора для каждого класса. Для этого значения R, G и B всех пикселей выборки суммируются и нормируются на количество элементов. Затем каждый усреднённый вектор дополнительно нормируется на свою длину, чтобы получить единичный спектральный вектор класса. Эти нормированные векторы копируются в константную память GPU, что обеспечивает быстрый и одновременный доступ всех потоков к неизменяемым данным.

Изображение полностью копируется в глобальную память на видеокарте. Далее запускается CUDA-ядро, работающее в режиме одномерной сетки потоков, где каждому потоку соответствует один пиксель изображения. Внутри ядра поток читает компоненты R, G и B своего пикселя, нормирует этот вектор и последовательно вычисляет скалярное произведение с каждым эталонным вектором из константной

памяти. На основе максимального значения выбирается номер класса, который записывается в поле w соответствующего пикселя результата.

После завершения вычислений все данные копируются обратно на CPU и сохраняются в выходной бинарный файл в исходном формате. В конце работы освобождается выделенная память как на центральном процессоре, так и на графическом ускорителе.

Описание программы

Программа реализована на языке C с использованием технологии CUDA и предназначена для классификации пикселей изображения методом спектрального угла. В состав программы входят функции для загрузки и сохранения бинарных изображений, вычисление усреднённых спектральных векторов классов, а также одно CUDA-ядро, выполняющее классификацию пикселей на GPU.

Основные функции программы:

1. **int loadImage(const char *fname, int *pw, int *ph, uchar4_custom **pdata)**

Функция загружает изображение из бинарного файла заданного формата.

Она считывает ширину и высоту изображения, выделяет память под массив пикселей и загружает каждую запись формата uchar4_custom, содержащую компоненты R, G, B и служебный байт A.

Возвращает 1 при успешном чтении и 0 при ошибке.

2. **int saveImage(const char *fname, int w, int h, const uchar4_custom *data)**

Функция сохраняет изображение в бинарный файл в том же формате, что и входные данные.

Записываются размеры изображения и полный массив пикселей.

3. **__global__ void kernel(uchar4_custom *input, uchar4_custom *output, int size)**

Это основное вычислительное ядро CUDA, выполняющее классификацию пикселей.

Каждый поток обрабатывает один пиксель изображения:

- берёт компоненты RGB;
- нормирует спектральный вектор пикселя;
- последовательно вычисляет скалярные произведения с нормированными усреднёнными векторами всех классов, находящимися в константной памяти;
- выбирает класс с максимальным значением нормированного dot-product;
- записывает номер класса в поле w выходного пикселя.

Использование константной памяти позволяет всем потокам эффективно и быстро получать параметры классов, а одномерная сетка потоков обеспечивает независимую обработку каждого пикселя.

4. **int main()**

Основная функция программы.

Выполняет полный цикл классификации изображения:

- считывает пути входного и выходного файлов;
- загружает изображение функцией loadImage;

- считывает количество классов и координаты выборок;
- вычисляет на CPU усреднённые цветовые векторы классов и их нормировку;
- копирует полученные векторы в константную память GPU;
- выделяет память на GPU и копирует изображение;
- запускает CUDA-ядро kernel с фиксированными параметрами сетки и блоков;
- получает результат обратно в память CPU;
- сохраняет классифицированное изображение через saveImage;
- освобождает всю выделенную память как на CPU, так и на GPU.

Таким образом, программа реализует параллельный алгоритм классификации изображения методом спектрального угла. Использование константной памяти, одномерной сетки потоков и вычислений на GPU позволяет эффективно обрабатывать изображения большого размера и значительно ускоряет выполнение по сравнению с последовательной CPU-версией.

Результаты

```
TEST 1 (300x300)
90000
GPU timings (in ms):
|grids: 1|blocks: 32|time: 0.091424 ms|
|grids: 1|blocks: 64|time: 0.024512 ms|
|grids: 1|blocks: 128|time: 0.021952 ms|
|grids: 1|blocks: 256|time: 0.021408 ms|
|grids: 1|blocks: 512|time: 0.021536 ms|
|grids: 8|blocks: 32|time: 0.022400 ms|
|grids: 8|blocks: 64|time: 0.023648 ms|
|grids: 8|blocks: 128|time: 0.020928 ms|
|grids: 8|blocks: 256|time: 0.023968 ms|
|grids: 8|blocks: 512|time: 0.025504 ms|
|grids: 64|blocks: 32|time: 0.045280 ms|
|grids: 64|blocks: 64|time: 0.040416 ms|
|grids: 64|blocks: 128|time: 0.040512 ms|
|grids: 64|blocks: 256|time: 0.045600 ms|
|grids: 64|blocks: 512|time: 0.065952 ms|
|grids: 512|blocks: 32|time: 0.735040 ms|
|grids: 512|blocks: 64|time: 0.682336 ms|
|grids: 512|blocks: 128|time: 0.521184 ms|
|grids: 512|blocks: 256|time: 0.946432 ms|
|grids: 512|blocks: 512|time: 2.201600 ms|
CPU: 12.931899 ms
```

TEST 2 (640x640)

409600

GPU timings (in ms):

|grids: 1|blocks: 32|time: 0.020736 ms|
grids: 1	blocks: 64	time: 0.026688 ms
grids: 1	blocks: 128	time: 0.022816 ms
grids: 1	blocks: 256	time: 0.023008 ms
grids: 1	blocks: 512	time: 0.023488 ms
grids: 8	blocks: 32	time: 0.023040 ms
grids: 8	blocks: 64	time: 0.023968 ms
grids: 8	blocks: 128	time: 0.021824 ms
grids: 8	blocks: 256	time: 0.021984 ms
grids: 8	blocks: 512	time: 0.026912 ms
grids: 64	blocks: 32	time: 0.055904 ms
grids: 64	blocks: 64	time: 0.064256 ms
grids: 64	blocks: 128	time: 0.090144 ms
grids: 64	blocks: 256	time: 0.098816 ms
grids: 64	blocks: 512	time: 0.120256 ms
grids: 512	blocks: 32	time: 0.784448 ms
grids: 512	blocks: 64	time: 0.717248 ms
grids: 512	blocks: 128	time: 0.572768 ms
grids: 512	blocks: 256	time: 0.998848 ms
grids: 512	blocks: 512	time: 2.263104 ms

CPU: 52.914823 ms

TEST 3 (2048x2048)

4194304

GPU timings (in ms):

|grids: 1|blocks: 32|time: 0.022752 ms|
grids: 1	blocks: 64	time: 0.031232 ms
grids: 1	blocks: 128	time: 0.028800 ms
grids: 1	blocks: 256	time: 0.026944 ms
grids: 1	blocks: 512	time: 0.027520 ms
grids: 8	blocks: 32	time: 0.027072 ms
grids: 8	blocks: 64	time: 0.026528 ms
grids: 8	blocks: 128	time: 0.025216 ms
grids: 8	blocks: 256	time: 0.028128 ms
grids: 8	blocks: 512	time: 0.033408 ms
grids: 64	blocks: 32	time: 0.070656 ms
grids: 64	blocks: 64	time: 0.088096 ms
grids: 64	blocks: 128	time: 0.147232 ms
grids: 64	blocks: 256	time: 0.278400 ms
grids: 64	blocks: 512	time: 0.539488 ms
grids: 512	blocks: 32	time: 1.693184 ms

```
|grids: 512|blocks: 64|time: 1.490240 ms|
|grids: 512|blocks: 128|time: 1.433280 ms|
|grids: 512|blocks: 256|time: 1.892064 ms|
|grids: 512|blocks: 512|time: 3.154816 ms|
CPU: 761.502579
```

В рамках работы было проведено сравнение времени выполнения алгоритма классификации пикселей методом спектрального угла на CPU и GPU при различных размерах тестовых изображений и при разных конфигурациях сетки и блоков CUDA. В отличие от предыдущей лабораторной работы, где использовался оператор Собеля, и вычислительная нагрузка на каждый пиксель была высокой, метод спектрального угла оперирует всего несколькими умножениями и сложениями. Поэтому объём вычислений на поток минимален, и общее время работы GPU в большей степени определяется накладными расходами на запуск ядра и планирование блоков.

Эксперименты показали, что:

- Для малых изображений запуск GPU занимает сопоставимое время с одним лишь выполнением ядра. Поэтому изменение числа потоков и блоков почти не влияет на итоговую производительность.
- Для средних изображений время работы GPU остаётся практически неизменным, тогда как время работы CPU растёт пропорционально количеству пикселей.
- Для больших изображений процессорная реализация становится значительно медленнее, тогда как GPU продолжает выполнять работу за доли миллисекунды, демонстрируя значительный выигрыш за счёт параллелизма.
- При больших значениях gridDim (например, 64 и 512) наблюдается рост времени на GPU - это связано не с вычислениями, а с увеличением количества блоков и, соответственно, накладными расходами на их планирование.

Таким образом, измерения подтверждают эффективность применения GPU для массовой классификации пикселей, особенно на больших изображениях. При этом для алгоритмов с малой вычислительной нагрузкой ключевую роль играет не столько число потоков, сколько архитектурные особенности CUDA и характер распределения задач между блоками.

Пример обработки изображений

Рис. 1. Оригинальное изображение.



Рис. 2. Классификация изображения методом спектрального угла при 5 классах (50 пикселей в выборке каждого класса).

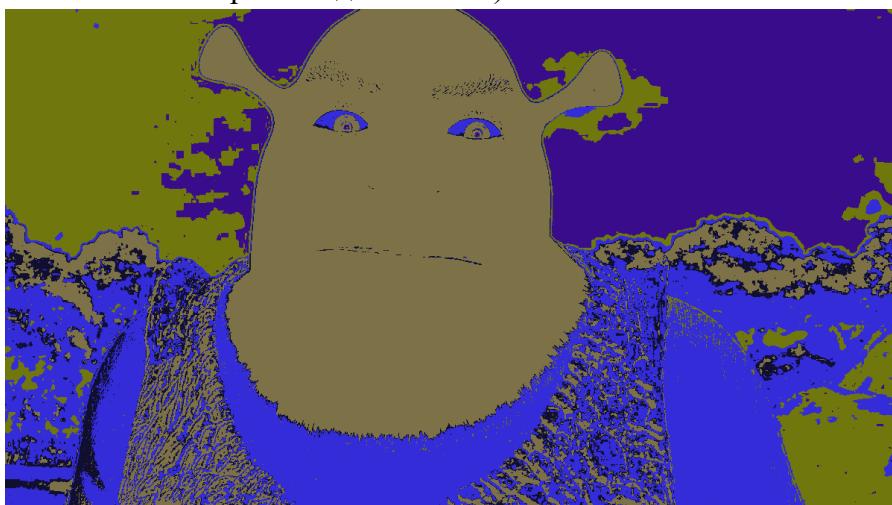
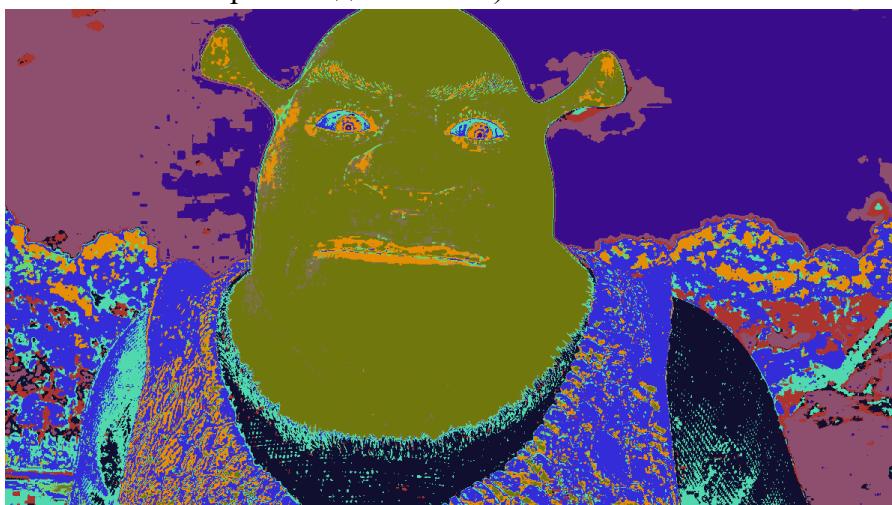


Рис. 3. Классификация изображения методом спектрального угла при 10 классах (50 пикселей в выборке каждого класса).



В ходе экспериментов были использованы конфигурации с 5 и 10 классами. Увеличение числа классов до 20 привело к появлению визуальных артефактов из-за чрезмерной фрагментации изображения, поэтому для отчёта были выбраны оптимальные значения $nc = 5$ и $nc = 10$. Объём обучающей выборки - 50 пикселей на класс.

Выводы

В этой лабораторной работе был реализован метод спектрального угла для классификации пикселей на GPU с использованием CUDA. Для обработки данных использовалась одномерная сетка потоков и константная память, в которую сохранялись усреднённые спектральные векторы классов.

Основные сложности возникли при корректной подготовке выборок для классов и при настройке конфигурации сетки. Поскольку сам алгоритм очень лёгкий по вычислениям, стало заметно, что время работы на GPU в значительной степени зависит от накладных расходов на запуск ядра, а не от самих операций. Это особенно проявилось при разных значениях числа блоков.

Эксперименты показали, что CPU начинает заметно проигрывать, как только изображение становится достаточно большим. GPU же выполняет классификацию практически мгновенно. При этом увеличение числа блоков сверх разумного приводит лишь к лишним накладным расходам, но не ускоряет работу, что тоже хорошо видно по результатам.

В целом лабораторная работа позволила на практике увидеть, как ведут себя лёгкие алгоритмы на GPU и почему важно учитывать не только количество потоков, но и стоимость их запуска. Также стало понятно, как использовать константную память и однородную одномерную сетку потоков для массовой обработки пикселей.