МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторная работа №*3*
по курсу «*Программирование графических процессоров*»
Освоение программного обеспечения для работы с технологией CUDA.

Классификация и кластеризация изображений на GPU.

Выполнил: Бачурин Павел Дмитриевич
Группа: М8О-403Б-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.

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 npj — количество пикселей в выборке, за ним следуют npj пар чисел — координаты пикселей выборки. nc, npj ≤ 32 ≤ 2 , w*h .

Выходные данные:
Необходимо записать в выходной файл в бинарном виде результат выполнения метода спектрального угла над входными данными.

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) Считываются названия входного и выходного файлов с помощью fgets.
   b) Считывается количество классов nc из стандартного ввода.
   c) Считывается исходное изображение с помощью функции readFile.
   d) Создаются массивы h_avg и h_avg_norm для хранения средних значений RGB каждого класса и их норм.
   e) Для каждого класса считываются координаты образцов пикселей (xi, yi). Если координаты выходят за пределы изображения, они корректируются. Вычисляется среднее значение RGB и норма для каждого класса.

2) Копирование данных на GPU:
   a) Копирование массивов h_avg и h_avg_norm в __constant__ память GPU с помощью cudaMemcpyToSymbol.
   b) Выделение памяти на GPU для входного изображения (d_img) и выходного (d_out) через cudaMalloc.
   c) Копирование исходного изображения img в память GPU (d_img) с помощью cudaMemcpy.
3) Параллельная обработка на GPU:
   a) Запускается kernel с блоками и потоками (blockDim=256, gridDim=1024). Каждый поток получает уникальный tid.
   b) Для каждого пикселя idx, вычисленного по tid и stride - извлекается цвет пикселя (r, g, b) и вычисляется его норма (pnorm). Инициализируются переменные best_class и best_cos для поиска ближайшего класса.
   c) Для каждого класса j: Берутся средние RGB значения (ar, ag, ab) и их норма (anorm) из __constant__ памяти. Вычисляется скалярное произведение и косинус угла между вектором пикселя и вектором класса. Ограничиваются значения cosv в диапазоне [-1, 1]. Если cosv больше текущего best_cos, обновляется best_cos и best_class. Формируется результирующий пиксель: RGB остаются как исходные, а в альфа-канал записывается best_class.
   d) Результат записывается в массив out (d_out).

4) Синхронизация и возврат данных:

a) После выполнения kernel вызывается cudaDeviceSynchronize для ожидания завершения всех потоков.
b) Результаты копируются обратно с GPU в память CPU (img) с помощью cudaMemcpy.

5) Сохранение результата и очистка ресурсов:
   a) Результирующее изображение записывается в выходной файл через writeFile.
   b) Освобождается память на GPU (cudaFree) и на CPU (free(img)).
   c) Все вызовы CUDA проверяются через макрос CSC(call) для обработки ошибок.

Архитектура программы:
Программа построена по клиент-серверной модели памяти CPU–GPU:
1) CPU: Чтение данных, выделение памяти, контроль ошибок, вывод результатов.
2) GPU: Параллельная обработка исходного изображения через kernel. Потоки организованы в блоки и сетку, каждый поток выполняет обрабатывает свою часть изображения.
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).

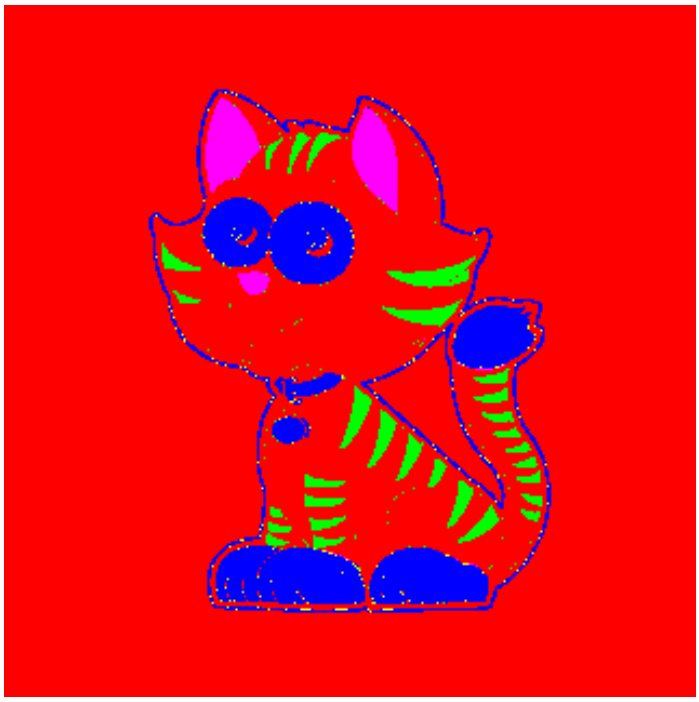
Описание программы

Основные типы данных:
1. Pixel — структура, содержащая четыре байта (r, g, b, a), соответствующие компонентам цвета RGBA. Используется для хранения пикселей изображения.
2. float — вещественный тип данных одинарной точности, применяется при вычислениях нормы цветового вектора и косинуса угла между векторами.
3. int — целочисленный тип данных, используется для хранения размеров изображения (ширина и высота), количества классов и индексов пикселей.
4. __constant__ float d_avg[MAX_CLASSES * 3], d_avg_norm[MAX_CLASSES] — специальные массивы постоянной памяти GPU для хранения средних значений RGB классов и их норм.

Основные функции и макросы:
1. Макрос CSC(call)
   Используется для проверки ошибок при вызовах CUDA API.
   Если вызов возвращает ошибку, выводится сообщение с указанием файла, строки и текста ошибки, после чего программа аварийно завершается.
2. Pixel* readFile(const char* filename, int* w, int* h)
   Считывает изображение из бинарного файла.
   Выделяет память под массив Pixel и возвращает указатель на него.

3. int writeFile(const char* filename, Pixel* data, int w, int h)
   Записывает изображение в бинарный файл, включая ширину и высоту.
4. __global__ void kernel(const Pixel* img, Pixel* out, int w, int h, int nc)
   CUDA-ядро для параллельной классификации пикселей изображения.
   Каждый поток:
   - Вычисляет норму цветового вектора пикселя.
   - Находит ближайший класс по косинусу угла между вектором пикселя и средним вектором класса.
   - Записывает результат в альфа-канал пикселя.

Результаты



1. Зависимость времени выполнения программы от количества используемых потоков (для тестов использовались изображения весом 3,7мб):

Потоки	Время (в мс)
8 × 8	11
64 × 64	3
128 × 128	1

2. Сравнение программы на CUDA 32*32 и программы на CPU с одним потоком:

Размер изображения	Время на CUDA (в мс)	Время на CPU (в мс)
3,7 мб	4	120

Выводы

В ходе выполнения лабораторной работы была реализована и протестирована программа для классификации изображений на графическом процессоре (GPU) с использованием технологии CUDA. В качестве метода классификации применён **метод спектрального угла**, основанный на вычислении косинусного сходства между вектором цвета пикселя и усреднённым вектором цвета класса. Каждый пиксель исходного изображения был отнесён к классу, для которого значение косинуса угла оказалось максимальным.

В соответствии с заданием, результат классификации записывался в альфа-канал выходного изображения в виде номера соответствующего класса. Для ускорения вычислений в программе использовалась **константная память** GPU, в которой хранились усреднённые векторы и их нормы для каждого класса. Распараллеливание обработки пикселей осуществлялось с помощью **одномерной сетки потоков**, где каждый поток обрабатывал один пиксель изображения.