

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторная работа №3
по курсу «Программирование графических процессоров»

Классификация и кластеризация изображений на GPU.

Выполнил: *И.Д. Фомин*

Группа: *M80-409Б-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.

Формат изображений. Изображение является бинарным файлом, со следующей структурой:

width(w)	height(h)	r	g	b	a	r	g	b	a	r	g	b	a	...	r	g	b	a	r	g	b	a
4 байта, int	4 байта, int	4 байта, значение пикселя [1, 1]	4 байта, значение пикселя [2, 2]	4 байта, значение пикселя [3, 1]	...	4 байта, значение пикселя [w-1, h]	4 байта, значение пикселя [w, h]															

В первых восьми байтах записывается размер изображения, далее построчно все значения пикселей, где

- r - красная составляющая цвета пикселя
- g - зеленая составляющая цвета пикселя
- b - синяя составляющая цвета пикселя
- a - значение альфа-канала пикселя

Пример картинки размером 2 на 2, синего цвета, в шестнадцатеричной записи:

02000000 02000000 0000FF00 0000FF00 0000FF00 0000FF00

Студентам предлагается самостоятельно написать конвертер на любом языке программирования для работы с вышеописанным форматом. В данной лабораторной работе используются только цветовые составляющие изображения (r g b), альфа-канал не учитывается. При расчетах значений допускается ошибка в ± 1 .

Ограничение: $w < 2^{16}$, $h < 2^{16}$.

Во всех вариантах, кроме 2-го и 4-го, в пограничном случае, необходимо “расширять” изображение за его границы, при этом значения соответствующих пикселей дублируют граничные. То есть, для любых индексов i и j, координаты пикселя [ip, jp] будут определяться следующим образом:

```
ip := max(min(i, h), 1)
jp := max(min(j, w), 1)
```

В вариантах 1-4, формат входных данных одинаковый. 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 pr_j - количество пикселей в выборке, за ним следуют pr_j пар чисел - координаты пикселей выборки.

$nc \leq 32$

$$np_j \leq 2^{19}$$

$$w * h \leq 4 * 10^8$$

Оценка вектора средних и ковариационной матрицы:

$$\begin{aligned} avg_j &= \frac{1}{np_j} \sum_{i=1}^{np_j} p_i^j \\ cov_j &= \frac{1}{np_j - 1} \sum_{i=1}^{np_j} (ps_i^j - avg_j) * (ps_i^j - avg_j)^T \end{aligned}$$

где $ps_i^j = (r_i^j, g_i^j, b_i^j)^T$ - i -ый пиксель из j -ой выборки.

Вариант 4. Метод спектрального угла.

Для некоторого пикселя p , номер класса jc определяется следующим образом:

$$jc = \arg \max_j \left[p^T * \frac{avg_j}{|avg_j|} \right]$$

Пример:

Входной файл	hex: in.data	hex: out.data
in.data out.data 2 4 1 2 1 0 2 2 2 1 4 0 0 0 1 1 1 2 0	03000000 03000000 A2DF4C00 F7C9FE00 9ED84500 B4E85300 99D14D00 92DD5600 A9E04C00 F7D1FA00 D4D0E900	03000000 03000000 A2DF4C01 F7C9FE00 9ED84501 B4E85301 99D14D01 92DD5600 A9E04C01 F7D1FA00 D4D0E900
in.data out.data 5 4 5 0 0 2 6 1 1 1 6 2 0 7 1 1 0 1 2 6 0 4 0 4 3 0 3 1 0 1 0 0 4 0 3 6 2 5 2 7 2 9 6 4 5 1 7 0 2 1 2 3 4 1 1 5 3 3 2 6	08000000 08000000 D2E27502 CFF65201 D3ED5701 D6E76902 C8F35B01 8E168200 CFF45001 AE977604 D3DC7102 7D1E7B00 AB9A8004 D9E58602 AB967E04 AE9D8004 87058200 D0F95B01 74148000 D0F55901 86136C00 85077400 D6E27702 D3609F03 D1609F03 CC5EA103 CC739D03 7C127F00 AA988804 AFA07D04 D0E37702 7D117A00 D6EB5901 D6E37C02 C9F85701 D655A103 D7EA7402 93127D00 D35BA403 D4DD7902 B0A18404 D6DE7502 D765A900 AD928404 D0D87C02 D7E97F02 CD509E00 CAF85201 CFF75601 CEF45E01 D0E86902 D1D17F02 AD928104 AFA18304 D4DB5C02 88077D00 C6F75701 7D127D00 A99A8E04 C8609E03 D15DA503 AB957E04 AE9A8004 79218100 D065A103 A99E9A04	08000000 08000000 D2E27502 CFF65201 D3ED5701 D6E76902 C8F35B01 8E168200 CFF45001 AE977604 D3DC7102 7D1E7B00 AB9A8004 D9E58602 AB967E04 AE9D8004 87058200 D0F95B01 74148000 D0F55901 86136C00 85077400 D6E27702 D3609F03 D1609F03 CC5EA103 CC739D03 7C127F00 AA988804 AFA07D04 D0E37702 7D117A00 D6EB5901 D6E37C02 C9F85701 D655A103 D7EA7402 93127D00 D35BA403 D4DD7902 B0A18404 D6DE7502 D765A900 AD928404 D0D87C02 D7E97F02 CD509E00 CAF85201 CFF75601 CEF45E01 D0E86902 D1D17F02 AD928104 AFA18304 D4DB5C02 88077D00 C6F75701 7D127D00 A99A8E04 C8609E03 D15DA503 AB957E04 AE9A8004 79218100 D065A103 A99E9A04

Программное и аппаратное обеспечение

OS	Ubuntu 22.04
IDE	VSCode 1.102.3
RAM	Общий объем: 32 GB Конфигурация: 16 GB + 16 GB Тип памяти: DDR5 Тактовая частота: 4800 МГц Форм-фактор: SODIMM Производители: Samsung
SSD	Тип: NVMe SSD Производитель: The Western Digital Модель: WD PC SN735 SDBPNHH-1T00 Объем: 1024 GB Интерфейс: NVMe Форм-фактор: M.2 2280
Compiler	ivan@DESKTOP-JOOUDAE:~/mai/course4 /pgp/lab1\$ nvcc --version nvcc: NVIDIA (R) Cuda compiler driver Copyright (c) 2005-2023 NVIDIA Corporation Built on Fri_Jan_6_16:45:21_PST_2023 Cuda compilation tools, release 12.0, V12.0.140

	Build cuda_12.0.r12.0/compiler.32267302_0 ivan@DESKTOP-JOOUDAE:~/mai/course4/pgp/lab1\$ gcc --version gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0 ivan@DESKTOP-JOOUDAE:~/mai/course4/pgp/lab1\$ g++ --version g++ (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
--	---

GPU:

Compute capability	8.6
Name	NVIDIA GeForce RTX 3080 Mobile
Total Global Memory	17179869184
Shared memory per block	102400
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	38

CPU:

brand	AMD Ryzen 9 6900HS with Radeon Graphics
arch	AMD64
bits	64
hz_advertised	3.2940 GHz
hz_actual	2.4980 GHz
cores_physical	8
cores_logical	16

Метод решения

Считываем файлы и классы => подсчитываем мат ожидания и матрицы ковариаций для всех классов. После этого уже запускается само ядро, которое выполняет классификацию. Каждый поток обрабатывает свой пиксель и смотрит на то, к какому классу следует отнести этот пиксель путем сравнения величин, характеризующих вероятность. Сами мат ожидания и матрицы расположены в константной памяти, что позволяет получить очень быстрый доступ к этим данным, которые используются в каждом потоке. После отработки ядра полученный класс записывается в альфа-канал для каждого исходного пикселя.

Описание программы

Первым делом рассмотрим вспомогательные макросы, структуры и константную память, используемую в ядре.

```
#define MAX_CLASSES 32
```

ограничивает максимальное число классов 32 - этого будет достаточно, я бы даже сказал, что даже 10 классов было бы норм.

```
#define CSC(call) \
do { \
    cudaError_t res = call; \
    if (res != cudaSuccess) { \
        fprintf(stderr, "ERROR in %s:%d. Message: %s\n", \
                __FILE__, __LINE__, cudaGetErrorString(res)); \
        exit(0); \
    } \
} while(0)
```

Макрос Cuda Safe Call (csc) обеспечивает безопасный вызов всех CUDA-функций: при ошибке выводится сообщение с тем, где ошибка и в чем и потом программа завершается.

Структуры:

```
struct Image { \
    int width = 0; \
    int height = 0; \
    std::vector<uchar4> pixels; \
    \
    Image(int w = 0, int h = 0) : width(w), height(h), pixels(w * h) {} \
}; \
 \
struct Task { \
    std::string in_path; \
    std::string out_path;
```

```
    int nc = 0;
    std::vector<std::vector<std::pair<int, int>>> samples;
};
```

Image хранит изображение в оперативной памяти: width, height - размеры, pixels - вектор цветов пикселей

Task же хранит путь входного и выходного файлов, в также количество классов и входные координаты пикселей

Константная память видеокарты

```
__constant__ float c_norm_avg[MAX_CLASSES][3];
__constant__ int c_num_classes;
```

Для ускорения вычислений в ядре используем две переменные в константной памяти GPU: c_norm_avg - нормированный средний цвет класса, c_num_classes - текущее число классов.

Константная память кэшируется и даёт высокую пропускную способность при чтении одинаковых данных всеми потоками

Функция main

считываем параметры задачи с помощью read_task();

```
void read_task(Task& task) {
    std::string line;

    std::getline(std::cin, task.in_path);
    std::getline(std::cin, task.out_path);
    std::getline(std::cin, line);

    task.nc = std::stoi(line);
    task.samples.resize(task.nc);

    for (int j = 0; j < task.nc; ++j) {
        std::getline(std::cin, line);

        std::stringstream ss(line);
        int np_j;
        ss >> np_j;

        for (int i = 0; i < np_j; ++i) {
            int x, y;
            ss >> x >> y;
            task.samples[j].push_back({y, x});
        }
    }
}
```

```
        }
    }
}
```

считываем картинку

```
void read_image(const std::string& path, Image& img) {
    std::ifstream file(path, std::ios::binary);

    file.read(reinterpret_cast<char*>(&img.width), sizeof(int));
    file.read(reinterpret_cast<char*>(&img.height), sizeof(int));

    img.pixels.resize(img.width * img.height);
    file.read(reinterpret_cast<char*>(img.pixels.data()), img.pixels.size() *
sizeof(uchar4));
}
```

После этого создаётся выходное изображение той же размерности и вычисляются нормированные средние цвета для каждого класса:

```
Image h_out_image(h_in_image.width, h_in_image.height);
int num_pixels = h_in_image.width * h_in_image.height;

float h_norm_avg[MAX_CLASSES][3] = {0};

for (int j = 0; j < task.nc; ++j) {
    int np_j = task.samples[j].size();
    if (np_j == 0) continue;

    double sum_r = 0, sum_g = 0, sum_b = 0;

    for (const auto& coord : task.samples[j]) {
        int y = coord.first;
        int x = coord.second;
        if (y >= h_in_image.height || x >= h_in_image.width) {
            return 1;
        }
        uchar4 p = h_in_image.pixels[y * h_in_image.width + x];
        sum_r += p.x;
        sum_g += p.y;
        sum_b += p.z;
    }

    float avg_j[3];
```

```

        avg_j[0] = (float)(sum_r / np_j);
        avg_j[1] = (float)(sum_g / np_j);
        avg_j[2] = (float)(sum_b / np_j);

        float mag = sqrt(avg_j[0] * avg_j[0] + avg_j[1] * avg_j[1] + avg_j[2]
* avg_j[2]);

        if (mag > 1e-6) {
            h_norm_avg[j][0] = avg_j[0] / mag;
            h_norm_avg[j][1] = avg_j[1] / mag;
            h_norm_avg[j][2] = avg_j[2] / mag;
        } else {
            h_norm_avg[j][0] = 0;
            h_norm_avg[j][1] = 0;
            h_norm_avg[j][2] = 0;
        }
    }
}

```

Для каждого класса суммируются r,g,b всех образцов, вычисляется среднее, вектор среднего цвета нормируется, но если вектор нулевой (близок к нему), получаем черный вектор, и чтобы не попасть в дальнейшем на арифметические ошибки из-за действий со слишком высокой точностью, ставим сразу нули.

Далее выделяем и копируем данные на GPU и запускаем ядро

```

uchar4* d_in_pixels;
uchar4* d_out_pixels;
size_t img_size_bytes = num_pixels * sizeof(uchar4);

CSC(cudaMalloc(&d_in_pixels, img_size_bytes));
CSC(cudaMalloc(&d_out_pixels, img_size_bytes));

CSC(cudaMemcpy(d_in_pixels, h_in_image.pixels.data(), img_size_bytes,
cudaMemcpyHostToDevice));

CSC(cudaMemcpyToSymbol(c_num_classes, &task.nc, sizeof(int)));
CSC(cudaMemcpyToSymbol(c_norm_avg, h_norm_avg, task.nc * 3 *
sizeof(float)));

classifySpectralAngleKernel<<<1024, 1024>>>(d_in_pixels, d_out_pixels,
num_pixels);

```

В ядре происходит следующее (сама классификация по спектральному углу):

```
__global__ void classifySpectralAngleKernel(const uchar4* in_pixels, uchar4* out_pixels, int num_pixels) {
    int global_idx = blockDim.x * blockDim.x + threadIdx.x;
    int total_threads = gridDim.x * blockDim.x;

    for (int i = global_idx; i < num_pixels; i += total_threads) {
        uchar4 p_in = in_pixels[i];
        float p[3];
        p[0] = (float)p_in.x; // R
        p[1] = (float)p_in.y; // G
        p[2] = (float)p_in.z; // B

        float mag_p = sqrtf(p[0] * p[0] + p[1] * p[1] + p[2] * p[2]);
        float p_norm0 = 0.0f, p_norm1 = 0.0f, p_norm2 = 0.0f;
        if (mag_p > 1e-6f) {
            p_norm0 = p[0] / mag_p;
            p_norm1 = p[1] / mag_p;
            p_norm2 = p[2] / mag_p;
        }

        int best_class = 0;
        float max_dot_product = -FLT_MAX;

        for (int j = 0; j < c_num_classes; j++) {
            float norm_avg_j[3];
            norm_avg_j[0] = c_norm_avg[j][0];
            norm_avg_j[1] = c_norm_avg[j][1];
            norm_avg_j[2] = c_norm_avg[j][2];

            float dot_product = p_norm0 * norm_avg_j[0] + p_norm1 *
norm_avg_j[1] + p_norm2 * norm_avg_j[2];

            if (dot_product > max_dot_product) {
                max_dot_product = dot_product;
                best_class = j;
            }
        }

        uchar4 p_out = p_in;
        p_out.w = (unsigned char)best_class;
```

```
        out_pixels[i] = p_out;
    }
}
```

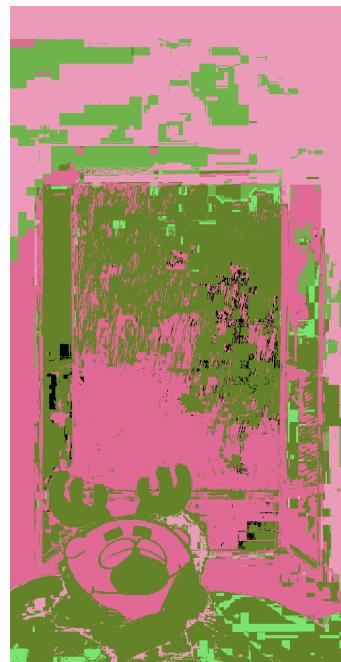
Для каждого пикселя:

1. RGB преобразуется в float и нормируются (если не нулевой вектор)
2. Для каждого класса вычисляется скалярное произведение между нормированным цветом пикселя и нормированным средним цветом класса
3. Поскольку оба вектора единичные, скалярное произведение равно cos угла между ними
4. Выбирается класс с максимальным скалярным произведением - с минимальным углом
5. Исходный пиксель копируется в выходной буфер, а в W записывается номер класса

Далее обратно копируем данные с девайса на CPU и записываем получившееся изображение в бинарник, который в дальнейшем переконвертируем в png с помощью питона

Результаты

651x1280



```
<<<1, 1>>> 2579.676мс
<<<1, 2>>> 1253.270мс
<<<1, 4>>> 632.739мс
<<<1, 8>>> 322.438мс
<<<1, 16>>> 161.407мс
```

```
<<<1, 32>>> 80.875mc
<<<2, 1>>> 1254.494mc
<<<2, 2>>> 633.247mc
<<<2, 4>>> 322.553mc
<<<2, 8>>> 161.445mc
<<<2, 16>>> 80.865mc
<<<2, 32>>> 40.531mc
<<<4, 1>>> 635.989mc
<<<4, 2>>> 328.456mc
<<<4, 4>>> 166.810mc
<<<4, 8>>> 80.998mc
<<<4, 16>>> 40.545mc
<<<4, 32>>> 20.319mc
<<<8, 1>>> 323.398mc
<<<8, 2>>> 183.728mc
<<<8, 4>>> 84.635mc
<<<8, 8>>> 40.509mc
<<<8, 16>>> 25.765mc
<<<8, 32>>> 10.404mc
<<<16, 1>>> 162.963mc
<<<16, 2>>> 83.029mc
<<<16, 4>>> 46.385mc
<<<16, 8>>> 20.618mc
<<<16, 16>>> 10.193mc
<<<16, 32>>> 5.234mc
<<<32, 1>>> 80.864mc
<<<32, 2>>> 40.497mc
<<<32, 4>>> 20.284mc
<<<32, 8>>> 10.182mc
<<<32, 16>>> 5.089mc
```

CPU time 95.635 ms

Image: 2560x1920



```
<<<1, 1>>> 15444.890mc
<<<1, 2>>> 7820.754mc
<<<1, 4>>> 3881.949mc
<<<1, 8>>> 2027.712mc
<<<1, 16>>> 1016.605mc
<<<1, 32>>> 508.043mc
<<<2, 1>>> 7561.720mc
<<<2, 2>>> 3880.524mc
<<<2, 4>>> 2028.378mc
<<<2, 8>>> 1014.297mc
<<<2, 16>>> 507.728mc
<<<2, 32>>> 254.326mc
<<<4, 1>>> 3880.399mc
<<<4, 2>>> 2040.755mc
<<<4, 4>>> 1014.924mc
<<<4, 8>>> 510.819mc
<<<4, 16>>> 255.112mc
<<<4, 32>>> 128.564mc
<<<8, 1>>> 2025.404mc
<<<8, 2>>> 1014.017mc
<<<8, 4>>> 507.069mc
<<<8, 8>>> 255.632mc
<<<8, 16>>> 128.103mc
<<<8, 32>>> 64.098mc
<<<16, 1>>> 1019.164mc
<<<16, 2>>> 509.295mc
<<<16, 4>>> 255.361mc
<<<16, 8>>> 127.923mc
<<<16, 16>>> 64.024mc
<<<16, 32>>> 32.082mc
<<<32, 1>>> 508.742mc
```

```
<<<32, 2>>> 254.629MC  
<<<32, 4>>> 127.799MC  
<<<32, 8>>> 64.031MC  
<<<32, 16>>> 32.033MC
```

CPU time 6565.253 ms

Image: 1280x960



```
<<<1, 1>>> 3797.098MC  
<<<1, 2>>> 1897.842MC  
<<<1, 4>>> 973.082MC  
<<<1, 8>>> 509.156MC  
<<<1, 16>>> 254.994MC  
<<<1, 32>>> 127.742MC  
<<<2, 1>>> 1897.370MC  
<<<2, 2>>> 972.504MC  
<<<2, 4>>> 509.045MC  
<<<2, 8>>> 254.668MC  
<<<2, 16>>> 127.564MC  
<<<2, 32>>> 63.971MC  
<<<4, 1>>> 971.809MC  
<<<4, 2>>> 507.996MC  
<<<4, 4>>> 254.321MC  
<<<4, 8>>> 132.889MC  
<<<4, 16>>> 63.795MC  
<<<4, 32>>> 32.011MC  
<<<8, 1>>> 508.159MC
```

```
<<<8, 2>>> 254.094мс
<<<8, 4>>> 127.202мс
<<<8, 8>>> 63.719мс
<<<8, 16>>> 31.986мс
<<<8, 32>>> 16.015мс
<<<16, 1>>> 253.797мс
<<<16, 2>>> 126.844мс
<<<16, 4>>> 63.609мс
<<<16, 8>>> 31.898мс
<<<16, 16>>> 15.990мс
<<<16, 32>>> 8.019мс
<<<32, 1>>> 126.694мс
<<<32, 2>>> 63.435мс
<<<32, 4>>> 31.847мс
<<<32, 8>>> 15.968мс
<<<32, 16>>> 8.008мс
```

CPU time 71.254 ms

Выводы:

Метод спектрального угла нужен для Дистанционном зондировании Земли, геологии, сельского хозяйства и т.д. - в общем везде, где классификация по цветам может дать что-то полезное

В CUDA хорошо развита работа с памятью - так, например, мы можем оптимизировать работу с ней при помощи константной памяти (=кеша).

По замерам производительности видно, что GPU работает сильно быстрее CPU. Чем больше потоков на GPU мы задействуем, тем больше эта разница становится. Наименее эффективны вычисления с единственным потоком в блоке и единственным блоком в гриде.