МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 810 «Информационные технологии в моделировании и управлении»

Лабораторная работа №3
по курсу «Основы Python, Java и Scala, платформы CUDA для анализа данных»

Классификация и кластеризация изображений на GPU.

Выполнил: А.С.Бобряков
Группа: М8О-103М-19
Преподаватель: А.Ю. Морозов

Москва, 2020

Условие

Научиться использовать GPU для классификации и кластеризации изображений. Использование константной памяти.

Формат изображений соответствует формату описанному в лабораторной работе 2. Во всех вариантах, в результирующем изображении, на месте альфа-канала должен быть записан номер класса(кластера) к которому был отнесен соответствующий пиксель. Если пиксель можно отнести к нескольким классам, то выбирается класс с наименьшим номером.

Вариант 2. Метод расстояний Махаланобиса.

Программное и аппаратное обеспечение

Видеокарта: NVIDIA GeForce GTX 1060 3Gb

Компоненты	Подробности	
GeForce GTX 1060 3GB	Версия драйвера:	441.22
	Тип драйвера:	Standard
	Версия API Direct3D:	12
	Уровень возможносте...	12_1
	Ядра CUDA:	1152
	Тактовая частота гра...	1594 МГц
	Скорость передачи д...	8.01 Гбит/с
	Интерфейс памяти:	192 бит
	Пропускная способнос...	192.19 ГБ/с
	Доступная графическ...	11237 МБ
	Выделенная видеопам...	3072 МБ GDDR5
	Системная видеопамя...	0 МБ
	Разделяемая системна...	8165 МБ
	Версия BIOS видео:	86.06.3C.00.7D
	IRQ:	Not used

Процессор: Intel® Core™ i7-8700K CPU @ 3.70GHz

Другое: ОС Windows, IDE – Clion EAP,

Метод решения

Решение выполнено путем реализации представленных в условии формул:

$$jc = arg\ max_j \left[-(p - avg_j)^T * cov_j^{-1} * (p - avg_j) \right]$$

где:

Оценка вектора средних и ковариационной матрицы:

$$avg_j = \frac{1}{np_j} \sum_{i=1}^{np_j} ps_i^j$$

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

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

Описание программы

В программе использовано ядро для реализации основной логики приложения. Код ядра описан на листинге 1.

```
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int idy = threadIdx.y + blockDim.y * blockIdx.y;
    int offsetx = blockDim.x * blockDim.x;
    int offsety = blockDim.y * blockDim.y;

    for (int i = idy; i < h; i += offsety) {
        for (int j = idx; j < w; j += offsetx) {
            int cls = 0x100;
            float max_val = 0;
            bool isFirst = true;
            uchar4& p = v[i * w + j];
            for (int k = 0; k < nc; ++k) {
                float3 f3_1 = make_float3(
                        x static_cast<float>(p.x) - g_avgs[k].x,
                        y static_cast<float>(p.y) - g_avgs[k].y,
                        z static_cast<float>(p.z) - g_avgs[k].z
                );
                float3 f3_2 = make_float3(
                        x f3_1.x * g_covs_inv[k][0][0] + f3_1.y * g_covs_inv[k][1][0] + f3_1.z * g_covs_inv[k][2][0],
                        y f3_1.x * g_covs_inv[k][0][1] + f3_1.y * g_covs_inv[k][1][1] + f3_1.z * g_covs_inv[k][2][1],
                        z f3_1.x * g_covs_inv[k][0][2] + f3_1.y * g_covs_inv[k][1][2] + f3_1.z * g_covs_inv[k][2][2]
                );
                float f_3   = f3_1.x * f3_2.x
                              + f3_1.y * f3_2.y
                              + f3_1.z * f3_2.z;
                float val = - f_3 ;
                if (isFirst){
                    max_val = val;
                    isFirst = false;
                }
                if (abs( x max_val - val) < 1e-6) {
                    cls = min(cls, k);
                }
                else {
                    if (max_val < val) {
                        max_val = val;
                        cls = k;
                    }
                }
            }
            p.w = cls;
        }
    }
}
```

Листинг 1 – Код ядра программы.

Также использованы функции для получения обратной матрицы:

```
bool inversion(float33& f_s, float33& f_d){
    double det = determinant( & f_s);
    if (det == 0){
        return false;
    }
    float33 f_buff;
    memcpy(&f_buff, &f_s, sizeof(float33));
    float fSwap;
    fSwap = f_buff.data[0 * 3 + 1];
    f_buff.data[0 * 3 + 1] = f_buff.data[1 * 3 + 0];
    f_buff.data[1 * 3 + 0] = fSwap;
    fSwap = f_buff.data[0 * 3 + 2];
    f_buff.data[0 * 3 + 2] = f_buff.data[2 * 3 + 0];
    f_buff.data[2 * 3 + 0] = fSwap;
    fSwap = f_buff.data[1 * 3 + 2];
    f_buff.data[1 * 3 + 2] = f_buff.data[2 * 3 + 1];
    f_buff.data[2 * 3 + 1] = fSwap;
    double d_d[9];
    d_d[0 * 3 + 0] = 0 + (f_buff.data[1 * 3 + 1] * f_buff.data[2 * 3 + 2] - f_buff.data[2 * 3 + 1] * f_buff.data[1 * 3 + 2]);
    d_d[0 * 3 + 1] = 0 - (f_buff.data[1 * 3 + 0] * f_buff.data[2 * 3 + 2] - f_buff.data[2 * 3 + 0] * f_buff.data[1 * 3 + 2]);
    d_d[0 * 3 + 2] = 0 + (f_buff.data[1 * 3 + 0] * f_buff.data[2 * 3 + 1] - f_buff.data[2 * 3 + 0] * f_buff.data[1 * 3 + 1]);
    d_d[1 * 3 + 0] = 0 - (f_buff.data[0 * 3 + 1] * f_buff.data[2 * 3 + 2] - f_buff.data[2 * 3 + 1] * f_buff.data[0 * 3 + 2]);
    d_d[1 * 3 + 1] = 0 + (f_buff.data[0 * 3 + 0] * f_buff.data[2 * 3 + 2] - f_buff.data[2 * 3 + 0] * f_buff.data[0 * 3 + 2]);
    d_d[1 * 3 + 2] = 0 - (f_buff.data[0 * 3 + 0] * f_buff.data[2 * 3 + 1] - f_buff.data[2 * 3 + 0] * f_buff.data[0 * 3 + 1]);
    d_d[2 * 3 + 0] = 0 + (f_buff.data[0 * 3 + 1] * f_buff.data[1 * 3 + 2] - f_buff.data[1 * 3 + 1] * f_buff.data[0 * 3 + 2]);
    d_d[2 * 3 + 1] = 0 - (f_buff.data[0 * 3 + 0] * f_buff.data[1 * 3 + 2] - f_buff.data[1 * 3 + 0] * f_buff.data[0 * 3 + 2]);
    d_d[2 * 3 + 2] = 0 + (f_buff.data[0 * 3 + 0] * f_buff.data[1 * 3 + 1] - f_buff.data[1 * 3 + 0] * f_buff.data[0 * 3 + 1]);
    for(int i = 0; i < 3; ++i){
        for(int j = 0; j < 3; ++j){
            f_d.data[i * 3 + j] = static_cast<float>(d_d[i * 3 + j] / det);
        }
    }
    return true;
}
```

Результаты

Пример исходной картинки изображен на рисунке 1.

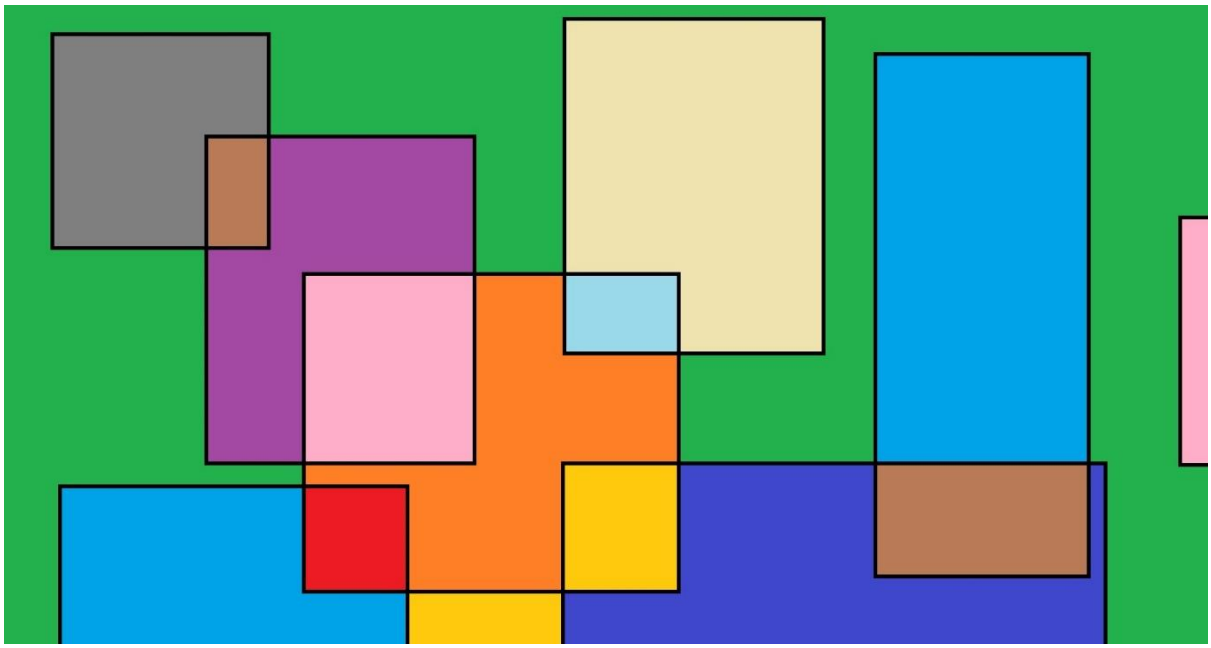

Рисунок 1. Исходное изображение.

Результат разбиения на два класса.

Рисунок 2 – Результат сглаживания SSAA.

Время работы ядра в зависимости от конфигурации представлены в Таблице 1.

Таблица 1. Время выполнения ядра программы в зависимости от конфигурации.

Число блоков \ Число потоков	32	128	512
32	0.065128	0.070523	0.125220
128	0.068230	0.132602	0.395266
512	0.135982	0.356440	1.536802

Выводы

В ходе выполнения лабораторной работы был исследован метод расстояния Махаланобиса для кластеризации изображений путем реализации соответствующих алгоритмов на CUDA.