

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»  
Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №3  
по курсу «Программирование графических процессоров»**

**Классификация и кластеризация изображений на GPU.**

Выполнил: М.А. Жерлыгин  
Группа: 8О-408Б  
Преподаватели: К.Г. Крашенинников,  
А.Ю. Морозов

Москва, 2021

## **Условие**

**Цель работы:** научиться использовать GPU для классификации и кластеризации изображений. Использование константной памяти.

**Вариант №2:** Расстояние Махalanобиса.

## **Программное и аппаратное обеспечение**

GPU:

1. Compute capability: 7.5;
2. Графическая память: 4294967296;
3. Разделяемая память: 49152;
4. Константная память: 65536;
5. Количество регистров на блок: 65536;
6. Максимальное количество блоков: (2147483647, 65535, 65535);
7. Максимальное количество нитей: (1024, 1024, 64);
8. Количество мультипроцессоров: 6.

Сведения о системе:

1. Процессор: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
2. Память: 16,0 ГБ;
3. HDD: 237 ГБ.

Программное обеспечение:

1. OS: Windows 10;
2. IDE: Visual Studio 2019;
3. Компилятор: nvcc.

## **Метод решения**

Перед вызовом ядра мы по формуле рассчитываем все средние значения и матрицы ковариаций. Затем для каждого пикселя входного изображения обрабатываем окрестность. Результат обработки каждого пикселя записываем в выходной массив. Обработка производится при помощи соответствующего ядра свертки. Класс пикселя выбирается как максимально соответствующий функции.

## **Описание программы**

Заводим массивы в константной памяти для более быстрого доступа к элементам:

```
__constant__ v3 gpu_avgs[32];
__constant__ matrix gpu_covs[32];
```

Функция вычисления матриц ковариации и средних (на CPU).

```

void pre_calculate(std::vector<std::vector<p>> &image, uchar4* pixels, int nc, int width) {
    std::vector<v3> a;
    std::vector<matrix> c;
    a.resize(32);
    c.resize(32);

    for (int i = 0; i < nc; i++) {
        int size = image[i].size();
        a[i].x = 0;
        a[i].y = 0;
        a[i].z = 0;
        for (int j = 0; j < size; j++) {
            p point = image[i][j];
            uchar4 pixel = pixels[point.x + point.y * width];
            a[i].x += pixel.x;
            a[i].y += pixel.y;
            a[i].z += pixel.z;
        }
        a[i].x /= size;
        a[i].y /= size;
        a[i].z /= size;
        for (int j = 0; j < size; j++) {
            p point = image[i][j];
            uchar4 pixel = pixels[point.y * width + point.x];
            c[i].values[0][0] += (pixel.x - a[i].x) * (pixel.x - a[i].x);
            c[i].values[0][1] += (pixel.x - a[i].x) * (pixel.y - a[i].y);
            c[i].values[0][2] += (pixel.x - a[i].x) * (pixel.z - a[i].z);
            c[i].values[1][0] += (pixel.y - a[i].y) * (pixel.x - a[i].x);
            c[i].values[1][1] += (pixel.y - a[i].y) * (pixel.y - a[i].y);
            c[i].values[1][2] += (pixel.y - a[i].y) * (pixel.z - a[i].z);
            c[i].values[2][0] += (pixel.z - a[i].z) * (pixel.x - a[i].x);
            c[i].values[2][1] += (pixel.z - a[i].z) * (pixel.y - a[i].y);
            c[i].values[2][2] += (pixel.z - a[i].z) * (pixel.z - a[i].z);
        }
        if (size > 1) {
            size = (double)(size - 1);
            for (auto& row : c[i].values) {
                for (auto& item : row) {
                    item /= size;
                }
            }
        }
    }

    matrixInverse(c[i]);
}

```

```

    glob_avgs[i] = a[i];
    glob_covs[i] = c[i];
}
}

```

В самом kernel я вычисляю общий индекс исполняемой нити, который и будет индексом в массиве при условии  $\text{idY} < \text{height}$   $\text{idX} < \text{width}$  и произвожу расчет класса для соответствующего пикселя:

```

global __ void mahalanobis_kernel(uchar4* image, int nc, int width, int height) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int offset_x = gridDim.x * blockDim.x;
    int offset_y = gridDim.y * blockDim.y;

    for (int j = idy; j < height; j += offset_y) {
        for (int i = idx; i < width; i += offset_x) {
            int temp = 0;
            uchar4 p = image[i + j * width];
            double max_jc = count_class_number(&p, 0);

            for (int k = 1; k < nc; k++) {

                double next_jc = count_class_number(&p, k);

                if (next_jc > max_jc) {
                    temp = k;
                    max_jc = next_jc;
                }
            }

            image[i + j * width].w = (unsigned char)temp;
        }
    }
}

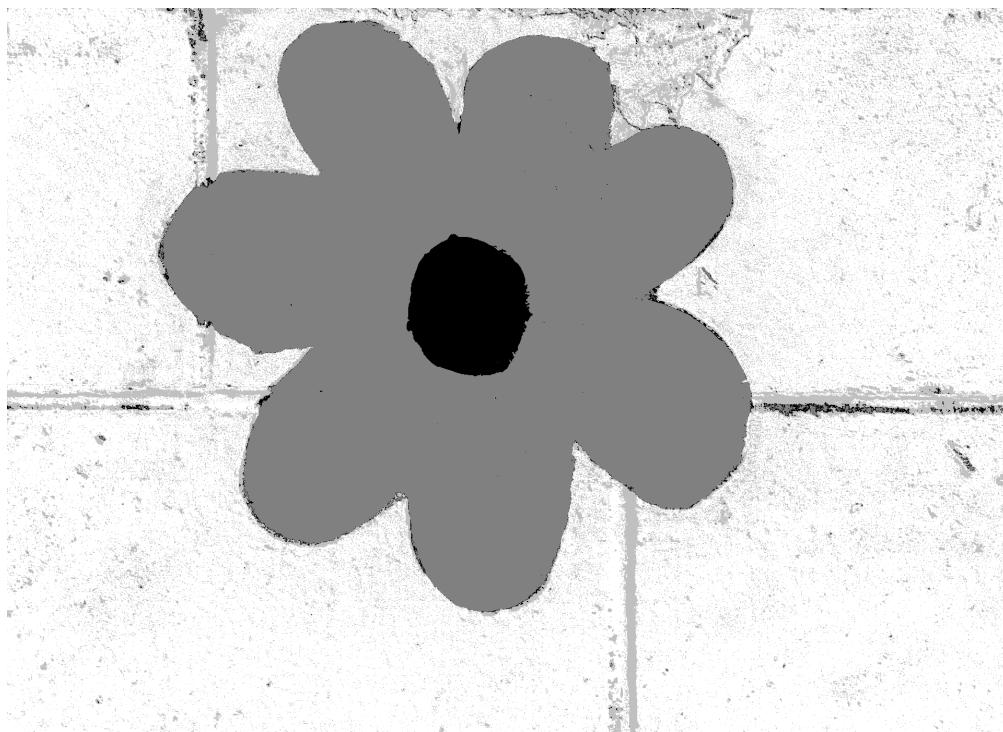
```

## Результаты

Пример входного изображения:



Изображение после классификации:



Threads	16 x 16	32 x 32	64 x 64	128 x 128	256 x 256
time	0.021524	0.022412	0.018424	0.017636	0.012533

## Выводы

Во время выполнения я пользовался константной памятью и убедился, что она является достаточно быстрой из доступных GPU. Отличительной особенностью константной памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти. Если необходимо использовать массив в константной памяти, то его размер необходимо указать заранее, так как динамическое выделение в отличие от глобальной памяти в константной не поддерживается.