МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет «Информационные технологии и прикладная математика»
Кафедра «Вычислительная математика и программирование»

Лабораторная работа №3
по курсу «Параллельная обработка данных»

Классификация и кластеризация изображений на GPU.

Выполнил: В.И. Лобов
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2019

Условие

Цель работы: 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 1. Метод максимального правдоподобия.

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 np_jпар чисел -- координаты пикселей выборки.
$nc \le 32, np_j \le 2^{19}, w * h \le 4 * 10^8$.

Программное и аппаратное обеспечение

GPU:

Название: GeForce GTX 1060
Compute apability: 6.1
Графическая память: 2075328512
Разделяемая память: 49152
Константная память: 65536
Количество регистров на блок: 65536
Максимальное количество блоков: (2147483647, 65535, 65535)
Максимальное количество нитей: (1024, 1024, 64)
Количество мультипроцессоров: 10

Сведения о системе:

Процессор: Intel Core i7-8700k 4.5GHz
Оперативная память: 16Gb
HDD: 1Tb
Операционная система: Ubuntu 18.04
IDE: Nsight
Компилятор: nvcc

Метод решения

Метод максимального правдоподобия.

Для некоторого пикселя p номер класса j_cопределяется следующим образом:

$$j_c = arg\,max_j\left(-\left(p-avg_j\right)^T * cov_j^{-1} * \left(p-avg_j\right) - \log\left(det\left(cov_j\right)\right)\right)$$

Оценку вектора средних и ковариационной матрицы можно выполнить следующим образом:

$$avg_j = 1/np_j * \sum\left(ps_i^j\right)$$

$$cov_j = 1/\left(np_j - 1\right) * \sum\left(ps_i^j - av_j\right) * \left(ps_i^j - av_j\right)^T$$

где $ps_i^j = \left(r_i^j, g_i^j, b_i^j\right)$- i-й пискель из j-ой выборки.

Описание программы

Выполним расчёт векторов средних, матриц ковариации и их определителей на центральном процессоре, а затем данные скопируем в константную память на видеокарте. Найдём для каждого пикселя наибольшее значение функции максимального правдоподобия и выберем номер класса в качестве ответа.

```
typedef struct {
   int x, y;
} Point;

typedef struct {
   float4 avg;
   double inverse_cov[3][3];
   double log_det;
} Class;

__constant__ Class dev_class[32];

float4 Average(uchar4 *data, int w, int h, Point *class_points, int point_n) {
   float4 result = make_float4(0, 0, 0, 0);

   for (int i = 0; i < point_n; ++i) {
      Point p = class_points[i];
      uchar4 pixel = data[p.y * w + p.x];

      result.x += pixel.x;
      result.y += pixel.y;
      result.z += pixel.z;
   }

   result.x /= point_n;
   result.y /= point_n;
   result.z /= point_n;

   return result;
}

void CalculateCovariance(double cov[3][3], uchar4 *data, int w, int h,
```

```
                    Point *class_points, int point_n, float4 avg) {
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            cov[i][j] = 0;
        }
    }

    for (int i = 0; i < point_n; ++i) {
        Point p = class_points[i];
        uchar4 pixel = data[p.y * w + p.x];
        double delta[3] = {pixel.x - avg.x, pixel.y - avg.y, pixel.z - avg.z};

        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                cov[i][j] += delta[i] * delta[j];
            }
        }
    }

    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            cov[i][j] /= point_n - 1;
        }
    }
}

double Determinant(double cov[3][3]) {
    double det = 0;

    for (int i = 0; i < 3; ++i) {
        det += cov[0][i] *
               cov[1][(i + 1) % 3] *
               cov[2][(i + 2) % 3];
        det -= cov[0][(i + 2) % 3] *
               cov[1][(i + 1) % 3] *
               cov[2][i];
    }

    return det;
}

void Inverse(double in[3][3], double out[3][3]) {
    double det = Determinant(in);
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            out[i][j] = in[(j + 1) % 3][(i + 1) % 3] * in[(j + 2) % 3][(i + 2) % 3] -
                        in[(j + 1) % 3][(i + 2) % 3] * in[(j + 2) % 3][(i + 1) % 3];
            out[i][j] /= det;
        }
    }
}

__device__ double MaxLikehoodEstimation(uchar4 p, int class_idx) {
    Class c = dev_class[class_idx];
    double delta[3] = {p.x - c.avg.x, p.y - c.avg.y, p.z - c.avg.z};
```

```
    double temp[3] = {0,};
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            temp[i] += delta[j] * c.inverse_cov[j][i];
        }
    }

    double result = -c.log_det;

    for (int i = 0; i < 3; ++i) {
        result -= temp[i] * delta[i];
    }

    return result;
}

__global__ void kernel(uchar4 *image, int w, int h, int class_count) {
        int idx = threadIdx.x + blockDim.x * blockIdx.x;
        int idy = threadIdx.y + blockDim.y * blockIdx.y;
        int offsetx = blockDim.x * gridDim.x;
        int offsety = blockDim.y * gridDim.y;
        int i, j;

        int class_idx;
        int max_idx = 0;
        double value;
        double max_value;

        for (i = idx; i < w; i += offsetx) {
            for (j = idy; j < h; j += offsety) {
                uchar4 pixel = image[j * w + i];

                max_value = INT_MIN;

                for (class_idx = 0; class_idx < class_count; ++class_idx) {
                    value = MaxLikehoodEstimation(pixel, class_idx);
                    if (value > max_value) {
                        max_idx = class_idx;
                        max_value = value;
                    }
                }

                image[j * w + i] = make_uchar4(pixel.x, pixel.y, pixel.z, max_idx);
            }
        }
}
```

Результаты:

Размер картинки: 640x400

Конфигурация	Время(мс)
CPU	273.689972
(32, 32), (32, 32)	7.859168
(64, 64), (32, 32)	5.529536
(128, 128), (32, 32)	5.512832
(256, 256), (32, 32)	5.614688

Размер картинки: 1920x1440

Конфигурация	Время(мс)
CPU	2899.072754
(32, 32), (32, 32)	30.774561
(64, 64), (32, 32)	29.398848
(128, 128), (32, 32)	29.830591
(256, 256), (32, 32)	31.664576

Размер картинки: 3840x2160

Конфигурация	Время(мс)
CPU	8759.058594
(32, 32), (32, 32)	85.312927
(64, 64), (32, 32)	85.517952
(128, 128), (32, 32)	73.000191
(256, 256), (32, 32)	71.735809

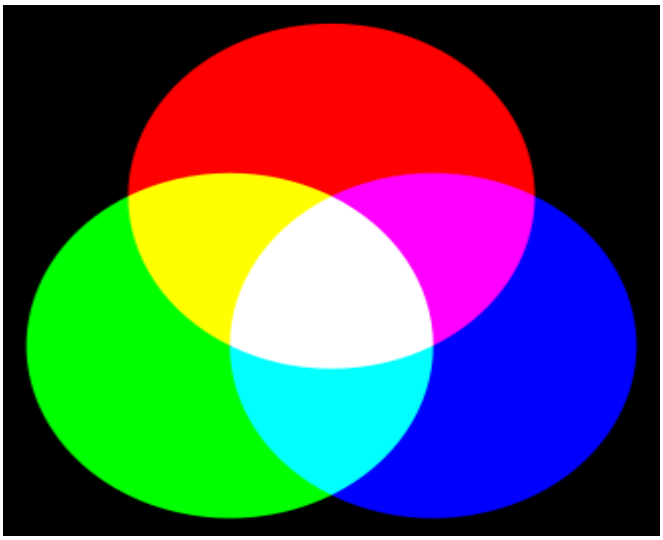

Иллюстрация 2: Исходное изображение

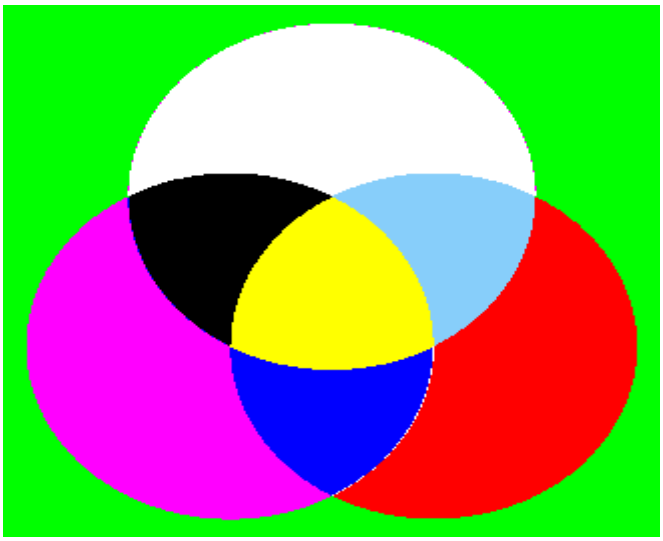

Иллюстрация 3: Классифицированные точки

В данном примере цвета классов не соответствуют цветам на исходной картинке, а лишь отличают классы друг от друга.

Пример с 32 классами(палитра в grayscale):

Иллюстрация 4: Исходное изображение

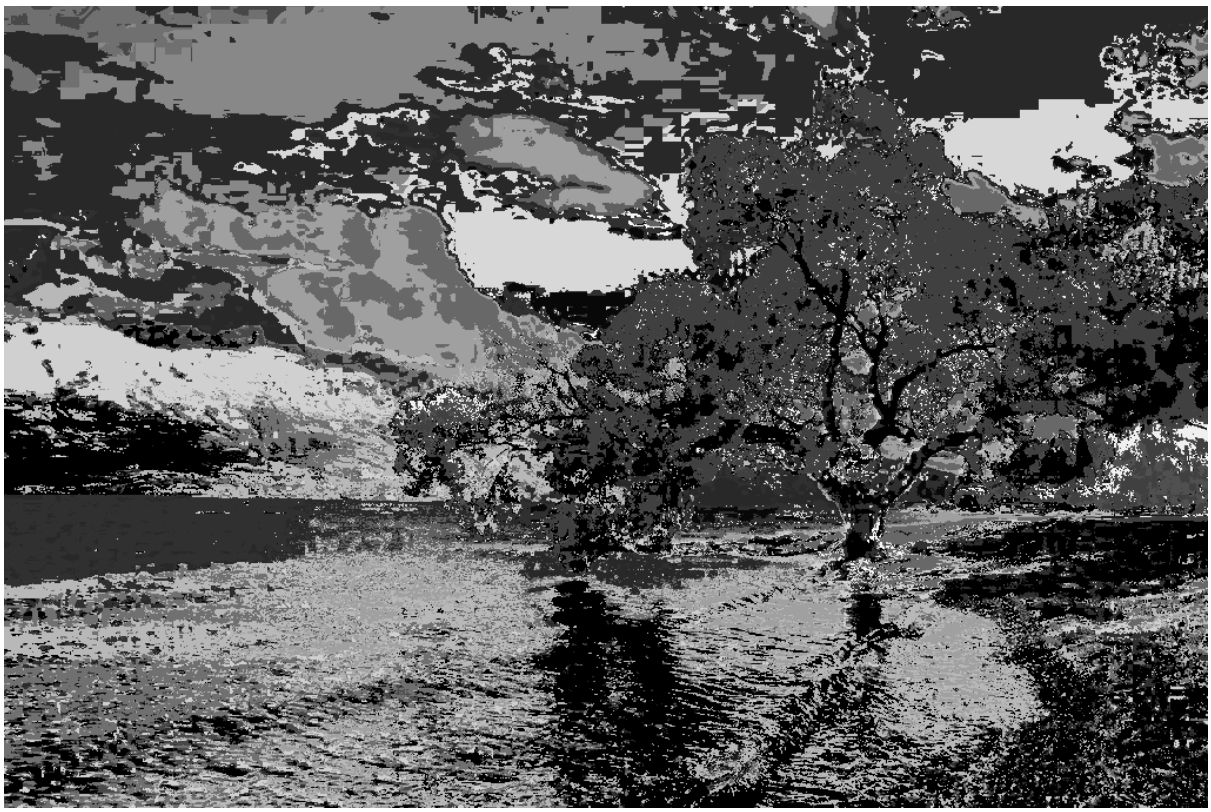

Иллюстрация 5: Классифицированные точки

Выводы

Выполнив данную лабораторную работу, я научился работать с константной памятью на видеокарте. Данной памяти не так много на видеокарте, в моём случае — 64кб, поэтому хранить изображения целиком в ней не удастся. Этот вид памяти можно эффективно использовать для хранения данных, к которым часто выполняется обращение в функции ядра. При вычислении матриц ковариации и векторов средних выполняется суммирование большого количества слагаемых. Этот процесс можно ускорить, выполнив вычисление суммы на видеокарте с помощью редукции. Это становится необходимо при большем количестве данных, чем во входных лабораторной работы.