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: Д. А. Ваньков
Группа: 8О-407Б-17
Преподаватели: А.Ю. Морозов,
К.Г. Крашенинников

Москва, 2020

Условие

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

Вариант 2. Расстояние Махаланобиса.

Программное и аппаратное обеспечение

Graphics card: GeForce 940M
Размер глобальной памяти: 4242604032
Размер константной памяти: 65536
Размер разделяемой памяти: 49152
Максимальное количество регистров на блок: 65536
Максимальное количество потоков на блок: 1024
Количество мультипроцессоров: 3
OS: Linux Ubuntu 18.04
Редактор: CLion, Atom

Метод решения

Перед вызовом ядра по формуле рассчитать все средние значения и матрицы ковариаций. Для каждого пикселя входного изображения следует выделить по отдельному потоку, каждый из которых будет обрабатывать окрестность этого потока. Результат обработки каждого пикселя будет записываться в выходной массив. Обработка производиться при помощи соответствующего ядра свертки. Класс пикселя выбирается как максимально соответствующий функции.

Описание программы

Для выполнения данной лабораторной работы я создал дополнительную функцию вычисления ковариаций и средних. Эта функция выполняется на CPU. Также завел массив в константной памяти для более быстрого доступа к элементам.

```
void calculate(vector<vector<point>> &v, uchar4* pixels, int nc, int w) {
  vector<vec3> avgs(32);
  vector<matr> covs(32);

  for (int i = 0; i < nc; ++i) {
    vec3 colors;
    avgs[i].x = 0;
    avgs[i].y = 0;
    avgs[i].z = 0;
```

```
for (int j = 0; j < v[i].size(); ++j) {
    point point_ = v[i][j];
    uchar4 pixel = pixels[point_.y * w + point_.x];

    getColors(colors, &pixel);

    avgs[i].x += colors.x;
    avgs[i].y += colors.y;
    avgs[i].z += colors.z;
}

double val = v[i].size();
avgs[i].x /= val;
avgs[i].y /= val;
avgs[i].z /= val;

for (int j = 0; j < v[i].size(); ++j) {
    point point_ = v[i][j];
    uchar4 pixel = pixels[point_.y * w + point_.x];

    getColors(colors, &pixel);

    vec3 diff;
    diff.x = colors.x - avgs[i].x;
    diff.y = colors.y - avgs[i].y;
    diff.z = colors.z - avgs[i].z;

    matr tmp;
    // diff * diff.T
    tmp.data[0][0] = diff.x * diff.x;
    tmp.data[0][1] = diff.x * diff.y;
    tmp.data[0][2] = diff.x * diff.z;
    tmp.data[1][0] = diff.y * diff.x;
    tmp.data[1][1] = diff.y * diff.y;
    tmp.data[1][2] = diff.y * diff.z;
    tmp.data[2][0] = diff.z * diff.x;
    tmp.data[2][1] = diff.z * diff.y;
    tmp.data[2][2] = diff.z * diff.z;

    for (int k = 0; k < 3; ++k) {
        for (int l = 0; l < 3; ++l) {
            covs[i].data[k][l] += tmp.data[k][l];
        }
    }
}

if (v[i].size() > 1) {
    val = (double)(v[i].size() - 1);
    for (auto & k : covs[i].data) {
```

```
            for (double & l : k) {
                l /= val;
            }
        }
    }
  }

  for (int i = 0; i < nc; ++i) {
    inverseMatr(covs[i]);
    copy_a[i] = avgs[i];
    copy_c[i] = covs[i];
  }
}
```

Перед вызовом kernel я записываю в константную память массив средних и ковариаций.

```
CUDA_ERROR(cudaMemcpyToSymbol(gpu_avgs, copy_a, 32 * sizeof(vec3)));
CUDA_ERROR(cudaMemcpyToSymbol(gpu_covs, copy_c, 32 * sizeof(matr)));
```

После расчета количества блоков по заданному количеству потоков на каждое из измерений я вызываю kernel.

```
mahalanobis_kernel<<<dim3(32, 32), dim3(32, 32)>>>(out_pixels, w, h, nc);
```

В самом kernel я вычисляю общий индекс исполняемой нити, который и будет индексом в массиве при условии idY < height idX < width. Далее производится расчет класса для соответствующего пикселя:

```
__global__ void mahalanobis_kernel(uchar4* pixels, int w, int h, int nc) {
  int idY = blockIdx.y * blockDim.y + threadIdx.y;
  int idX = blockIdx.x * blockDim.x + threadIdx.x;
  int offsetY = gridDim.y * blockDim.y;
  int offsetX = gridDim.x * blockDim.x;

  for (int row = idY; row < h; row += offsetY) {
    for (int col = idX; col < w; col += offsetX) {
      uchar4 pixel = pixels[row * w + col];
      double mx = findPixel(&pixel, 0);
      int mIdx = 0;
      for (int i = 1; i < nc; ++i) {
        double tmp = findPixel(&pixel, i);
        if (mx < tmp) {
          mx = tmp;
          mIdx = i;
        }
      }
      pixels[row * w + col].w = (unsigned char)mIdx;
    }
```

```
    }
}
```

В функции findPixel() производится поиск пикселя в данном классе.

```
__device__ double findPixel(uchar4* pixel, int idx) {
    vec3 colors;
    getColors(colors, pixel);

    double diff[3];
    diff[0] = colors.x - gpu_avgs[idx].x;
    diff[1] = colors.y - gpu_avgs[idx].y;
    diff[2] = colors.z - gpu_avgs[idx].z;

    double matrAns[3];
    matrAns[0] = 0;
    matrAns[1] = 0;
    matrAns[2] = 0;

    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            matrAns[i] += gpu_covs[idx].data[j][i] * diff[j];
        }
    }

    double ans = 0.0;
    for (int i = 0; i < 3; ++i) {
        ans += diff[i] * matrAns[i];
    }
    return -ans;
}
```

После вызова kernel я копирую данные в массив на хост и освобождаю выделенную память.

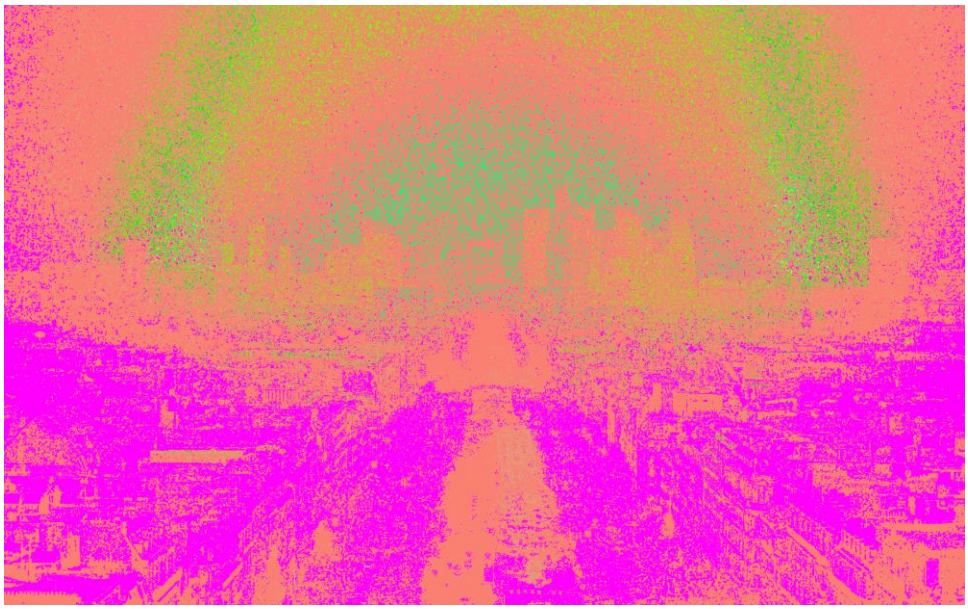
Результаты

Перед использование тестовые изображения нужно было с помощью конвертера конвертировать в нужный формат данных, и после его обратно в исходный формат.

Пример изображения до классификации:

Size = 3840 x 2400

С помощью конвертера, полученное изображение после классификации:



Я сравнил время работы на изображении размером 30x30 с разным количеством запущенных потоков:

Threads, size	ms
16 * 16	0.012
32 * 32	0.015
64 * 64	0.003
128 * 128	0.001
256 * 256	0.001

Также я сравнил время работы на CPU и GPU на картинках с разными размерами:

size	CPU, ms	GPU, ms
30 * 30	0.002	0.01
650 * 650	629.492	0.021
3840 * 2400	16097.1	0.023

Из этого можно сделать вывод, что время выполнение на GPU существенно выше, если данные средние или большие. На небольших тестах эффективнее использовать CPU, так как мы не тратим лишнее время на копирование на GPU. Также при небольших тестах потоки на GPU простаивают без данных и не производят вычислений.

Выводы

После выполнения данной лабораторной работы я, в очередной раз, убедился, что выполнение математических операций на GPU происходит гораздо быстрее, чем на CPU. Также понял, что данная особенность может использоваться в машинном обучении для классификации, где производится много вычислений.

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.