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.

Выполнила: Алексюнина Ю.В.

Группа:М8О-407Б

Преподаватели:

К.Г.Крашенинников, А.Ю. Морозов

Москва, 2021

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.

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 npj – количество пикселей в выборке, за ним следуют npj пар чисел -- координаты пикселей выборки.

Вариант: 4. Метод спектрального угла

Программное и аппаратное обеспечение

GPU:
- Name: GeForce GTX 750 Ti
- Compute capability: 5.0
- Графическая память: 4294967295
- 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: 5

Сведения о системе:
- Процессор: Intel Core i5-4460 3.20GHz
- ОЗУ: 16 ГБ
- HDD: 930 ГБ

Программное обеспечение:
- OS: Windows 8.1
- IDE: Visual Studio 2019
- Компилятор: nvcc

Метод решения:

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

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

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:

Прежде всего, необходимо рассчитать элементы формулы (в данном случае, средние значения, а также их нормирование). Эти методы выполняются на CPU, поскольку на GPU их запускать нецелесообразно.

Далее, создаем массив константной памяти нужного размера для промежуточных результатов для каждого из классов. Константная память используется тогда, когда в ядро необходимо передать много различных данных, которые будут одинаково использоваться всеми тредами ядра, а также является достаточно быстрой.

`__constant__ vector3 gpu_norm_avgs[32];` — объявление глобальной переменной для использования в качестве константной памяти.

`CSC(cudaMemcpyToSymbol(gpu_norm_avgs, copy_norm_avgs, 32 * sizeof(vector3)));` — копирование данных с центрального процессора в константную память.

Данные копируются непосредственно перед вызовом kernel.

В самом kernel вызываем метод `find(uchar4 pixel, int c)`, в котором ищем пиксель для данного класса. А в целом, в kernel как раз-таки происходит сама классификация для каждого пикселя по максимальному соответствию.

После того, как каждый пиксель «обрел свой класс», копируем данные обратно на хост, записываем в файл и чистим память.

Файл kernel.cu:

```
#include "cuda_runtime.h"
```

```
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <vector>
#include <math.h>
using namespace std;

#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)

typedef struct
{
     int x;
     int y;
} vector2;

typedef struct
{
     double x;
```

```
     double y;
     double z;
} vector3;

__constant__ vector3 gpu_avgs[32];
__constant__ vector3 gpu_norm_avgs[32];

vector3 copy_avgs[32];
vector3 copy_norm_avgs[32];

__device__ __host__ void RGBget(vector3& rgb, uchar4* pixel)
{
     rgb.x = pixel->x;
     rgb.y = pixel->y;
     rgb.z = pixel->z;
}

void _avgs_(vector<vector<vector2>>& vec, uchar4* data, int w,
int h, int nc)
{
     vector<vector3> avgs(32);
     for (int i = 0; i < nc; i++)
     {
          avgs[i].x = 0;
          avgs[i].y = 0;
          avgs[i].z = 0;

          for (int j = 0; j < vec[i].size(); j++)
          {
               vector2 point = vec[i][j];
               uchar4 pixel = data[point.y * w + point.x];
               vector3 rgb;
```

```
                RGBget(rgb, &pixel);

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

            avgs[i].x /= vec[i].size();
            avgs[i].y /= vec[i].size();
            avgs[i].z /= vec[i].size();
        }

        for (int i = 0; i < nc; i++)
            copy_avgs[i] = avgs[i];
        //return avgs;
}

void _norm_avgs_(int nc)
{
        for (int i = 0; i < nc; i++)
        {

            //cout << sqrt(pow(copy_avgs[i].x, 2) +
pow(copy_avgs[i].y, 2) + pow(copy_avgs[i].z, 2));
            copy_norm_avgs[i].x = (double)copy_avgs[i].x /
sqrt(pow(copy_avgs[i].x, 2) + pow(copy_avgs[i].y, 2) +
pow(copy_avgs[i].z, 2));
            copy_norm_avgs[i].y = (double)copy_avgs[i].y /
sqrt(pow(copy_avgs[i].x, 2) + pow(copy_avgs[i].y, 2) +
pow(copy_avgs[i].z, 2));
            copy_norm_avgs[i].z = (double)copy_avgs[i].z /
sqrt(pow(copy_avgs[i].x, 2) + pow(copy_avgs[i].y, 2) +
pow(copy_avgs[i].z, 2));

        }
```

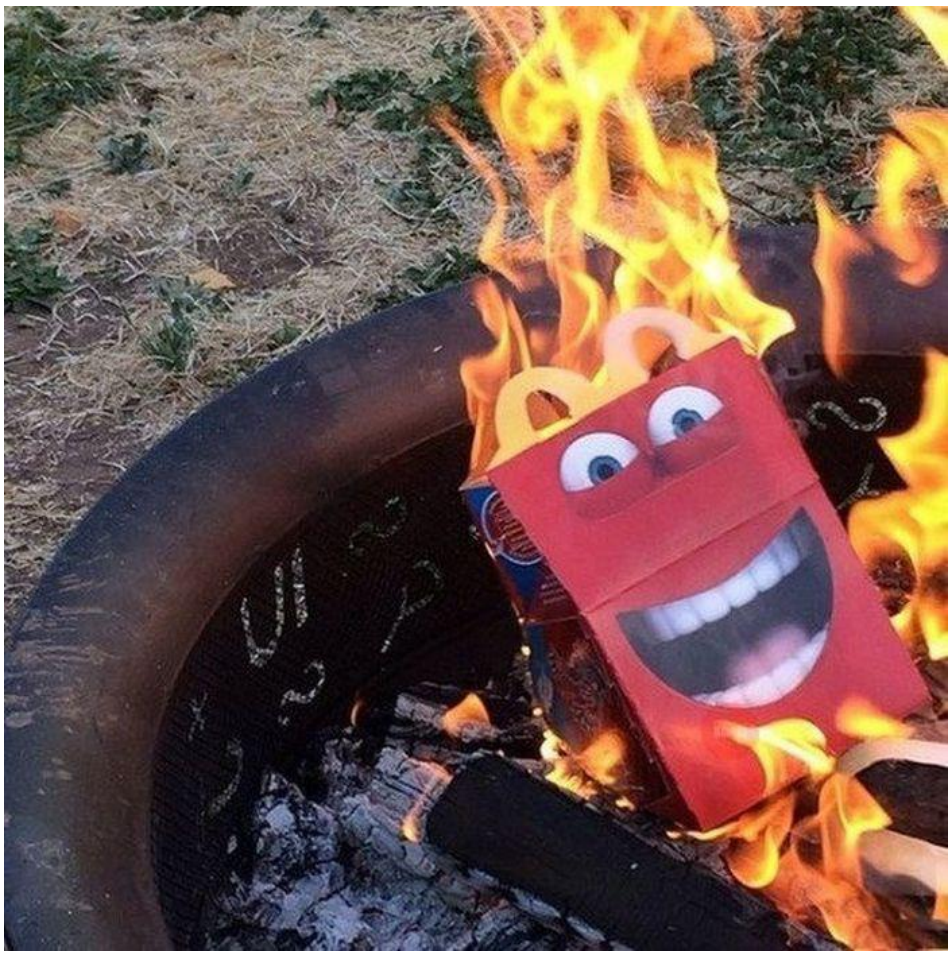
```
}

__device__ double find(uchar4 pixel, int c)
{
     vector3 rgb;
     RGBget(rgb, &pixel);
     //vector3 div_avgs;

     double res = 0;

     double trgb[3];
     trgb[0] = rgb.x;
     trgb[1] = rgb.y;
     trgb[2] = rgb.z;
     double tnorm[3];
     tnorm[0] = gpu_norm_avgs[c].x;
     tnorm[1] = gpu_norm_avgs[c].y;
     tnorm[2] = gpu_norm_avgs[c].z;
     for (int i = 0; i < 3; i++)
               res += trgb[i] * tnorm[i];
     return res;
}

__global__ void spectral_kernel(uchar4* data, int w, int h, int
nc)
{
     int idx = blockDim.x * blockIdx.x + threadIdx.x;
     int idy = blockDim.y * blockIdx.y + threadIdx.y;
     int offsetx = blockDim.x * gridDim.x;
     int offsety = blockDim.y * gridDim.y;
     int x, y;
```

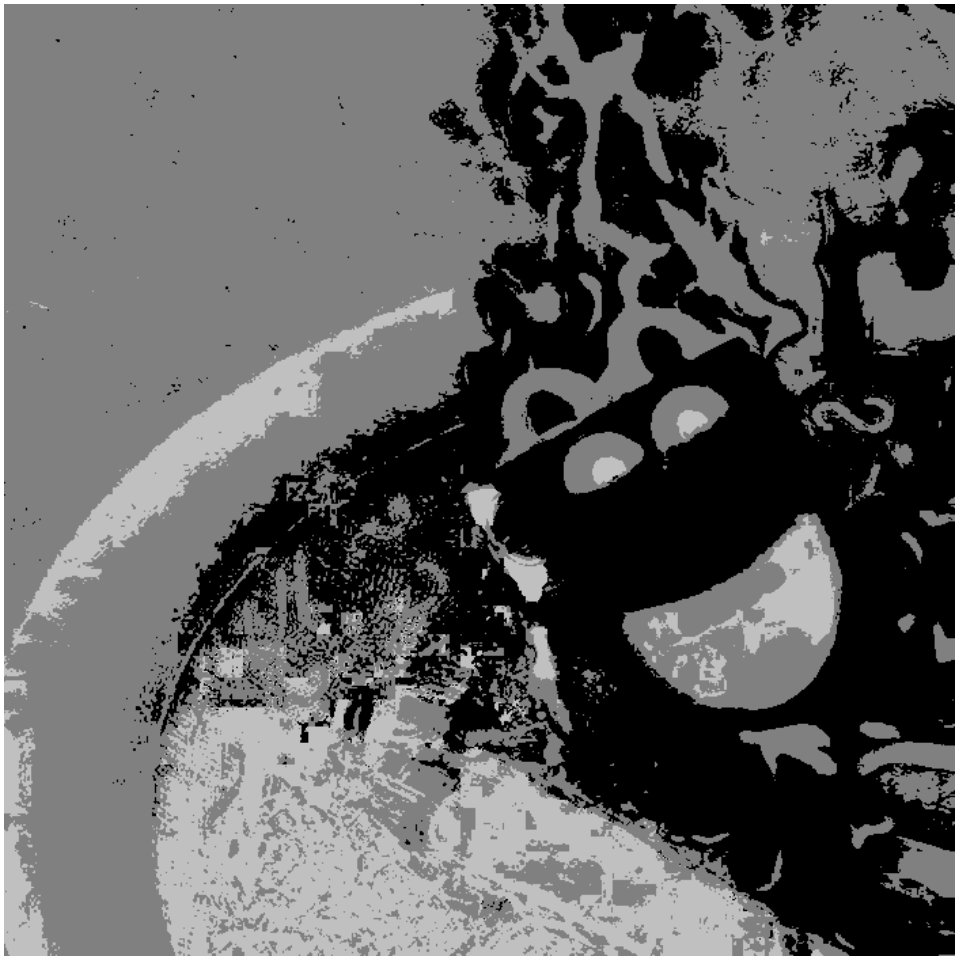
```
	for (y = idy; y < h; y += offsety)
		for (x = idx; x < w; x += offsetx)
		{
			uchar4 pixel = data[y * w + x];
			double sa = find(pixel, 0);
			int sac = 0;
			for (int i = 1; i < nc; i++)
			{
				double tsa = find(pixel, i);
				if (sa < tsa) //argmax
				{
					sa = tsa;
					sac = i;
				}
			}
			data[y * w + x].w = (unsigned char)sac;
		}
}
int main()
{
	int w, h;
	string inFile;
	string outFile;

	std::cin >> inFile >> outFile;

	FILE* fp = fopen(inFile.c_str(), "rb");
	//FILE* fp = fopen("in.data", "rb");
	fread(&w, sizeof(int), 1, fp);
	fread(&h, sizeof(int), 1, fp);
	uchar4* data = (uchar4*)malloc(sizeof(uchar4) * w * h);
	fread(data, sizeof(uchar4), w * h, fp);
```

```
    fclose(fp);
    int nc, np; //classes, pixels
    cin >> nc;
    vector<vector<vector2>> vec(nc); //coords
    for (int i = 0; i < nc; i++)
    {
        cin >> np;
        vec[i].resize(np);
        for (int j = 0; j < np; j++)
        {
            cin >> vec[i][j].x >> vec[i][j].y;
        }
    }

    _avgs_(vec, data, w, h, nc);
    _norm_avgs_(nc);

    CSC(cudaMemcpyToSymbol(gpu_avgs, copy_avgs, 32 *
sizeof(vector3)));

    CSC(cudaMemcpyToSymbol(gpu_norm_avgs, copy_norm_avgs, 32 *
sizeof(vector3)));

    uchar4* out;

    CSC(cudaMalloc(&out, sizeof(uchar4) * w * h));

    CSC(cudaMemcpy(out, data, sizeof(uchar4) * w * h,
cudaMemcpyHostToDevice));

    spectral_kernel << <dim3(32, 32), dim3(32, 32) >> > (out, w,
h, nc);
```

```
	CSC(cudaGetLastError());

	CSC(cudaMemcpy(data, out, sizeof(uchar4) * w * h,
cudaMemcpyDeviceToHost));

	CSC(cudaFree(out));

	fp = fopen(outFile.c_str(), "wb");
	//fp = fopen("out.data", "wb");
	fwrite(&w, sizeof(int), 1, fp);
	fwrite(&h, sizeof(int), 1, fp);
	fwrite(data, sizeof(uchar4), w * h, fp);
	fclose(fp);

	free(data);

	return 0;
}
```

Результаты:





Для изображения размером 640 x 1280

	GPU
dim3(32,32)	**0.473ms**
dim3(64,64)	**0.002ms**
dim3(128,128)	**0.001ms**
dim3(256,256)	**0.001ms**

	GPU	**CPU**
50x50	**0.004ms**	**0.02ms**
500x500	**518ms**	**0.02ms**
1500x1500	**6894ms**	**0.021ms**
2500x2500	**12766ms**	**0.023ms**

Выводы:

Константная память она является достаточно быстрой. На хосте в константную память можно что-то записать, вызвав функцию cudaMemcpyToSymbol. Из устройства константная память доступна только для чтения. Константная память очень удобна в использовании. Можно размещать в ней данные любого типа и читать их при помощи простого присваивания. 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.