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: Симонов С.Я.
Группа: 8О-406Б-18
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

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

Вариант 4. Метод спектрального угла.

Программное и аппаратное обеспечение

GPU:

Compute capability : 6.1
Name : GeForce GTX 1050
Total Global Memory : 2096103424
Shared memory per block : 49152
Registers per block : 65536
Max threads per block : (1024, 1024, 64)
Max block : (2147483647, 65535, 65535)
Total constant memory : 65536
Multiprocessors count : 5

Сведения о системе:

Операционная система: Ubuntu 14.04 LTS 64-x битная
Рабочая среда: nano
Компилятор: nvcc

Метод решения

Изначально на CPU считаем средние значения и нормы средних значений. Потом переносим их в константную память на GPU. В моем случае достаточно было перенести только норму, а само среднее значение не трогать. Константная память использовалась по причине того, что было необходимо неоднократно передвать большое количество неизменяемых переменных. Далее вычисляем в kernel по ниже приведенной формуле номер класса текущего пикселя.

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

Присваиваем альфа-каналу значение номера класса к которому был отнесен соответствующий пиксель.

Также были реализованы две структуры, по факту можно было обойтись и без них.

Описание программы

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

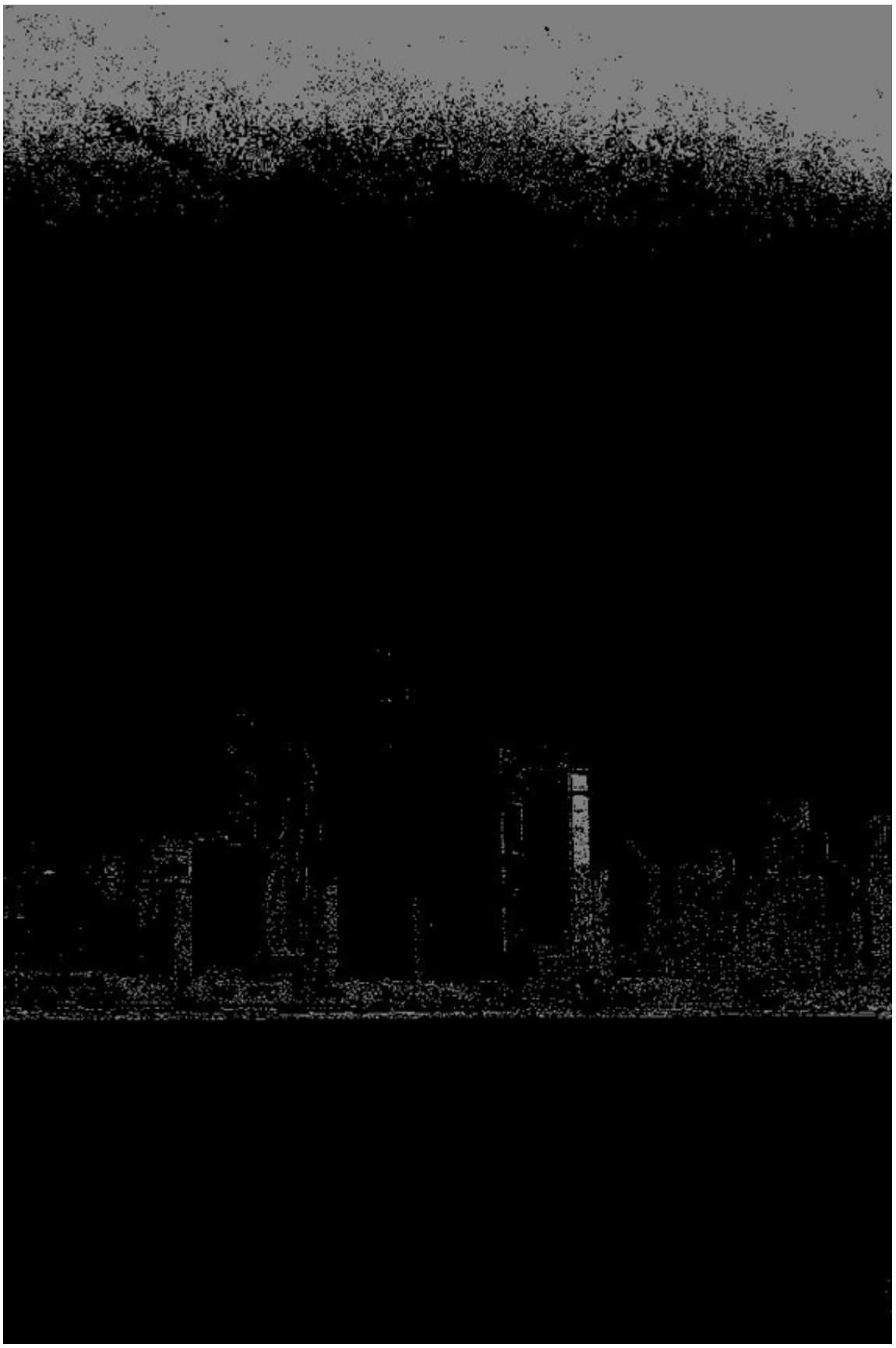
```
#include <iostream>
#include <vector>

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

typedef struct {
	float x = 0;
	float y = 0;
	float z = 0;
} cord_float;

__constant__ float avg[32][3];
__constant__ float norm_avg[32][3];

__global__ void kernel(uchar4* pixels, int w, int h, int count_clases) {
	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, i, max_p;
	cord_float rgb;
	float max_val;
	float result[32];
	for(y = idy; y < h; y += offsety) {
		for(x = idx; x < w; x += offsetx) {
			uchar4 p;
			p.x = pixels[y * w + x].x;
			p.y = pixels[y * w + x].y;
			p.z = pixels[y * w + x].z;
			for (i = 0; i < count_clases; ++i) {
				rgb.x = p.x * norm_avg[i][0];
				rgb.y = p.y * norm_avg[i][1];
				rgb.z = p.z * norm_avg[i][2];
				result[i] = rgb.x + rgb.y + rgb.z;
			}
			max_val = result[0];
			max_p = 0;
			for (i = 1; i < count_clases; ++i) {
				if (max_val < result[i]) {
					max_val = result[i];
					max_p = i;
				}
			}
			pixels[y * w + x].w = max_p;
		}
```

```
	}
}

int main(int argc, const char* argv[])
{
	std::string in, out;
	int w, h, count_clases, size;
	std::cin >> in >> out >> count_clases;
	std::vector<std::vector<point>> vv(count_clases);
	for (int i = 0; i < count_clases; ++i) {
		std::cin >> size;
		vv[i].resize(size);
		for (int j = 0; j < size; ++j) {
			point t;
			std::cin >> t.x >> t.y;
			vv[i][j] = t;
		}
	}
	FILE *fp = fopen(in.c_str(), "rb");
	fread(&w, sizeof(int), 1, fp);
	fread(&h, sizeof(int), 1, fp);
	uchar4 *data;
	data = (uchar4*)malloc(sizeof(uchar4) * w * h);
	fread(data, sizeof(uchar4), w * h, fp);
	fclose(fp);uchar4 *dev_out;
	cudaMalloc(&dev_out, sizeof(uchar4) * w * h);
	cudaMemcpy(dev_out, data, sizeof(uchar4) * w * h, cudaMemcpyHostToDevice);


	float cpu_avg[32][3];
	for (int i = 0; i < count_clases; ++i) {
		point c;
		cpu_avg[i][0] = 0;
		cpu_avg[i][1] = 0;
		cpu_avg[i][2] = 0;
		for (int j = 0; j < vv[i].size(); ++j) {
			c.x = vv[i][j].x;
			c.y = vv[i][j].y;
			cpu_avg[i][0] += data[c.y * w + c.x].x;
			cpu_avg[i][1] += data[c.y * w + c.x].y;
			cpu_avg[i][2] += data[c.y * w + c.x].z;
		}
		cpu_avg[i][0] /= vv[i].size();
		cpu_avg[i][1] /= vv[i].size();
		cpu_avg[i][2] /= vv[i].size();
	}

	float norm_cpu_avg[32][3];
```

```
	for (int i = 0; i < count_clases; ++i) {
		norm_cpu_avg[i][0] = cpu_avg[i][0] / std::sqrt(cpu_avg[i][0] * cpu_avg[i][0] +
cpu_avg[i][1] * cpu_avg[i][1] + cpu_avg[i][2] * cpu_avg[i][2]);
		norm_cpu_avg[i][1] = cpu_avg[i][1] / std::sqrt(cpu_avg[i][0] * cpu_avg[i][0] +
cpu_avg[i][1] * cpu_avg[i][1] + cpu_avg[i][2] * cpu_avg[i][2]);
		norm_cpu_avg[i][2] = cpu_avg[i][2] / std::sqrt(cpu_avg[i][0] * cpu_avg[i][0] +
cpu_avg[i][1] * cpu_avg[i][1] + cpu_avg[i][2] * cpu_avg[i][2]);
	}

	cudaMemcpyToSymbol(avg, cpu_avg, sizeof(float) * 32 * 3);
	cudaMemcpyToSymbol(norm_avg, norm_cpu_avg, sizeof(float) * 32 * 3);
   kernel<<<dim3(32, 32), dim3(32, 32)>>>(dev_out, w, h, count_clases);
	cudaMemcpy(data, dev_out, sizeof(uchar4) * w * h, cudaMemcpyDeviceToHost);
	cudaFree(dev_out);
	FILE *fp1;
	fp1 = fopen(out.c_str(), "wb");
	fwrite(&w, sizeof(int), 1, fp1);
	fwrite(&h, sizeof(int), 1, fp1);
	fwrite(data, sizeof(uchar4), w * h, fp1);
	fclose(fp1);
	free(data);
	return 0;
}
```

Результаты


IDEAS



На вход программе подается одна и также картинка размером 2304000.

Конфигурация	Время работы
CPU	23.086
<<<dim3(4, 4), dim3(4, 4)>>>	2.814592
<<<dim3(16, 16), dim3(16, 16)>>>	0.382336
<<<dim3(16, 32), dim3(16, 32)>>>	0.429440
<<<dim3(32, 32), dim3(32, 32)>>>	0.742688

Выводы

Выполнив данную лабораторную работу,я реализовал алгоритм “метод спектрального угла”. Данный алгоритм является алгоритмом классификации с обучением, по данной причине необходимо изначально задавать классы состоящие из выборки пикселей. Кроме того, следует отметить, что данный метод не чувствителен к яркости пикселей, всевозможные яркости обрабатываются одинаково, так происходит поскольку используется только направление векторов.

Также следует упомянуть про константную память, это крайне удобный инструмент в использовании, динамическое выделение не поддерживается, необходимо указывать размер массива заранее.