3МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3**

**по курсу «Программирование графических процессоров»**

**Классификация и кластеризация изображений на GPU.**

Выполнил: Иларионов Д.А.

Группа: М8О-408Б-17

Преподаватели: Крашенинников К.Г.,

Морозов А.Ю.

Москва, 2020

**Условие**

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

**Вариант 2. Метод расстояния Махаланобиса.**

**Программное и аппаратное обеспечение**

**GPU:**

* Name: GeForce GTX 1060
* Compute capability: 6.1
* Частота видеопроцессора: 1404 – 1670 (Boost) МГц
* Частота памяти: 8000 МГц
* Графическая память: 6144 МБ
* Разделяемая память: отсутствует
* 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 i7-8750H 2.20GHz x 6
* Оперативная память: 16 ГБ
* SSD: 128 ГБ
* HDD: 1000 ГБ

**Программное обеспечение:**

* OS: Windows 10
* IDE: Visual Studio 2019
* Компилятор: nvcc

**Метод решения**

Текстурную память в данной лабораторной использовать нельзя. Представление изображений такое же, как и в прошлой лабе – одномерный массив. Для обращения к нижнему, надо прибавить ширину изображения, к правому пикселю – 1. Сначала мы находим средние вектора для каждого класса. Далее – просто по формуле, находим ковариационную матрицу, обращаем ее и далее уже вызываем потоки. Но перед этим закидываем наши матрицы и вектора в постоянную память. Далее, мы проходимся по тому же изображению, каждый поток обрабатывает свои пиксели. Ковариационные обращенные матрицы и средние вектора лежат в константной памяти. То есть они не изменяются во время работы потоков, а размер = 32 матрицы/вектора – постоянны, тк макс. количество возможных классов указано 32. Делаем циклов столько, сколько у нас классов. Находим максимальное значение и класс, при котором оно получилось. Просто перемножаем вектор на матрицу и снова на вектор, получаем один элемент, который и нужно сравнить. Класс выбираем тот, где значение этого элемента максимально. Для макс. эффективной памяти мы используем только 1 изображение и просто перезаписываем его в другой файл с уже установленными классами в альфа-канале. Для отчета и визуализации классификации – можем поменять цвета пикселей в зависимости от класса. Так я и сделал для примеров.

**Описание программы**

**Файл kernel.cu**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <iostream>

#include <fstream>

#include <cstdio>

#include <sstream>

#include <iomanip>

#include <math.h>

#include <algorithm>

#include <string>

#include <cuda.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 uchar4 pixels;

typedef double pixFloat[3];

typedef pixFloat matrix3[3];

\_\_constant\_\_ pixFloat dev\_cAvg[32];

\_\_constant\_\_ matrix3 dev\_cMatrInv[32];

typedef unsigned char bytes;

struct image {

int width;

int height;

pixels\* pixs;

};

struct pixel {

int x;

int y;

};

image newImage(int w, int h) {

image nIMG;

nIMG.width = w;

nIMG.height = h;

nIMG.pixs = new pixels[w \* h];

return nIMG;

}

image newImage(string filename) {

FILE\* file;

image thisImg;

if ((file = fopen(filename.c\_str(), "rb")) == NULL) {

std::cout << "Can't load image from file" << std::endl;

exit(1);

}

fread(&thisImg.width, sizeof(thisImg.width), 1, file);

fread(&thisImg.height, sizeof(thisImg.height), 1, file);

thisImg.pixs = new pixels[thisImg.width \* thisImg.height];

fread(thisImg.pixs, sizeof(pixels), thisImg.width \* thisImg.height, file);

fclose(file);

return thisImg;

}

void writeToFile(image img, string filename) {

FILE\* file = fopen(filename.c\_str(), "wb");

fwrite(&img.width, sizeof(img.width), 1, file);

fwrite(&img.height, sizeof(img.height), 1, file);

fwrite(img.pixs, sizeof(pixels), img.width \* img.height, file);

fclose(file);

}

string imgToString(image img) {

std::stringstream stream;

stream << img.width << " " << img.height << "\n";

for (int i = 0; i < img.height; i++) {

for (int j = 0; j < img.width; j++) {

int k = i \* img.width + j;

stream << hex << setfill('0') << setw(2) << (int)img.pixs[k].x << setfill('0') << setw(2) << (int)img.pixs[k].y << setfill('0') << setw(2) << (int)img.pixs[k].z << setfill('0') << setw(2) << (int)img.pixs[k].w << " ";

}

stream << "\n";

}

return stream.str();

}

\_\_global\_\_ void Mahalanobisse(pixels\* pixelsOut, int w, int h, int classes)

{

int tX = blockIdx.x \* blockDim.x + threadIdx.x;

int tY = blockIdx.y \* blockDim.y + threadIdx.y;

int offsetX = gridDim.x \* blockDim.x;

int offsetY = gridDim.y \* blockDim.y;

for (int i = tY; i < h; i += offsetY)

{

for (int j = tX; j < w; j += offsetX)

{

pixels thisPixel = pixelsOut[j + i \* w];

double thisRed = (double)thisPixel.x;

double thisGreen = (double)thisPixel.y;

double thisBlue = (double)thisPixel.z;

double maxAm = 0;

int argMax = -1;

for (int c = 0; c < classes; ++c) {

double ans = 0;

pixFloat vec1;

pixFloat vec2;

pixFloat vec3;

vec1[0] = -(thisRed - dev\_cAvg[c][0]);

vec2[0] = thisRed - dev\_cAvg[c][0];

vec1[1] = -(thisGreen - dev\_cAvg[c][1]);

vec2[1] = thisGreen - dev\_cAvg[c][1];

vec1[2] = -(thisBlue - dev\_cAvg[c][2]);

vec2[2] = thisBlue - dev\_cAvg[c][2];

vec3[0] = vec1[0] \* dev\_cMatrInv[c][0][0] + vec1[1] \* dev\_cMatrInv[c][1][0] + vec1[2] \* dev\_cMatrInv[c][2][0];

vec3[1] = vec1[0] \* dev\_cMatrInv[c][0][1] + vec1[1] \* dev\_cMatrInv[c][1][1] + vec1[2] \* dev\_cMatrInv[c][2][1];

vec3[2] = vec1[0] \* dev\_cMatrInv[c][0][2] + vec1[1] \* dev\_cMatrInv[c][1][2] + vec1[2] \* dev\_cMatrInv[c][2][2];

ans = vec3[0] \* vec2[0] + vec3[1] \* vec2[1] + vec3[2] \* vec2[2];

if (ans > maxAm || argMax == -1) {

maxAm = ans;

argMax = c;

}

}

pixelsOut[j + i \* w].w = argMax;

//coloring 3 for report

//if (argMax == 0) {

// pixelsOut[j + i \* w].x = 255;

// pixelsOut[j + i \* w].y = 0;

// pixelsOut[j + i \* w].z = 0;

//}

//else if (argMax == 1) {

// pixelsOut[j + i \* w].x = 0;

// pixelsOut[j + i \* w].y = 255;

// pixelsOut[j + i \* w].z = 0;

//}

//else if (argMax == 2) {

// pixelsOut[j + i \* w].x = 0;

// pixelsOut[j + i \* w].y = 0;

// pixelsOut[j + i \* w].z = 255;

//}

}

}

}

void begin(image\* image1, int classes) {

pixels\* oldPixels;

int size1 = sizeof(pixels) \* image1->width \* image1->height;

CSC(cudaMalloc((void\*\*)& oldPixels, size1));

dim3 gridSz(32, 32);

dim3 blockSz(32, 32);

CSC(cudaMemcpy(oldPixels, image1->pixs, size1, cudaMemcpyHostToDevice));

Mahalanobisse << < gridSz, blockSz >> > (oldPixels, image1->width, image1->height, classes);

CSC(cudaMemcpy(image1->pixs, oldPixels, size1, cudaMemcpyDeviceToHost));

CSC(cudaFree(oldPixels));

}

int main()

{

string input;

string output;

int w;

cin >> input >> output;

image myImage = newImage(input);

w = myImage.width;

int classes;

cin >> classes;

double curRed = 0;

double curGreen = 0;

double curBlue = 0;

pixFloat\* cAvg = new pixFloat[classes];

matrix3\* cMatr = new matrix3[classes];

matrix3\* cMatrInv = new matrix3[classes];

//генерация средних векторов и ков. матриц

for (int i = 0; i < classes; ++i) {

long long pixs\_am = 0;

curRed = 0;

curGreen = 0;

curBlue = 0;

cin >> pixs\_am;

pixel\* pixPairs = new pixel[pixs\_am];

for (long long j = 0; j < pixs\_am; ++j) {

int X, Y;

cin >> X >> Y;

curRed += (double)myImage.pixs[X + w \* Y].x;

curGreen += (double)myImage.pixs[X + w \* Y].y;

curBlue += (double)myImage.pixs[X + w \* Y].z;

pixPairs[j].x = X;

pixPairs[j].y = Y;

}

curRed /= pixs\_am;

curGreen /= pixs\_am;

curBlue /= pixs\_am;

cAvg[i][0] = curRed;

cAvg[i][1] = curGreen;

cAvg[i][2] = curBlue;

matrix3 totalMatrix;

for (int Ti = 0; Ti < 3; ++Ti) {

for (int Tj = 0; Tj < 3; ++Tj) {

totalMatrix[Ti][Tj] = 0;

}

}

for (int j = 0; j < pixs\_am; ++j) {

pixFloat vec;

vec[0] = (double)myImage.pixs[pixPairs[j].x + w \* pixPairs[j].y].x - cAvg[i][0];

vec[1] = (double)myImage.pixs[pixPairs[j].x + w \* pixPairs[j].y].y - cAvg[i][1];

vec[2] = (double)myImage.pixs[pixPairs[j].x + w \* pixPairs[j].y].z - cAvg[i][2];

for (int Ti = 0; Ti < 3; ++Ti) {

for (int Tj = 0; Tj < 3; ++Tj) {

totalMatrix[Ti][Tj] += vec[Ti] \* vec[Tj];

}

}

}

for (int Ti = 0; Ti < 3; ++Ti) {

for (int Tj = 0; Tj < 3; ++Tj) {

totalMatrix[Ti][Tj] /= max(0.000001, (double)pixs\_am - 1);

cMatr[i][Ti][Tj] = totalMatrix[Ti][Tj];

}

}

delete[] pixPairs;

}

for (int i = 0; i < classes; ++i) {

double det = 0;

det = cMatr[i][0][0] \* cMatr[i][1][1] \* cMatr[i][2][2] + cMatr[i][0][2] \* cMatr[i][1][0] \* cMatr[i][2][1] +

cMatr[i][0][1] \* cMatr[i][1][2] \* cMatr[i][2][0] - cMatr[i][2][0] \* cMatr[i][1][1] \* cMatr[i][0][2] -

cMatr[i][0][1] \* cMatr[i][1][0] \* cMatr[i][2][2] - cMatr[i][0][0] \* cMatr[i][1][2] \* cMatr[i][2][1];

if (det == 0) det = 0.0000001; //чтобы программа не вылетала

matrix3 transp;

for (int x = 0; x < 3; ++x) {

for (int y = 0; y < 3; ++y) {

transp[x][y] = cMatr[i][y][x];

}

}

double dop1 = transp[1][1] \* transp[2][2] - transp[1][2] \* transp[2][1];

double dop4 = transp[1][2] \* transp[2][0] - transp[1][0] \* transp[2][2];

double dop7 = transp[1][0] \* transp[2][1] - transp[1][1] \* transp[2][0];

double dop2 = transp[0][2] \* transp[2][1] - transp[0][1] \* transp[2][2];

double dop5 = transp[0][0] \* transp[2][2] - transp[0][2] \* transp[2][0];

double dop8 = transp[0][1] \* transp[2][0] - transp[0][0] \* transp[2][1];

double dop3 = transp[0][1] \* transp[1][2] - transp[0][2] \* transp[1][1];

double dop6 = transp[0][2] \* transp[1][0] - transp[0][0] \* transp[1][2];

double dop9 = transp[0][0] \* transp[1][1] - transp[0][1] \* transp[1][0];

cMatrInv[i][0][0] = dop1 / det;

cMatrInv[i][0][1] = dop2 / det;

cMatrInv[i][0][2] = dop3 / det;

cMatrInv[i][1][0] = dop4 / det;

cMatrInv[i][1][1] = dop5 / det;

cMatrInv[i][1][2] = dop6 / det;

cMatrInv[i][2][0] = dop7 / det;

cMatrInv[i][2][1] = dop8 / det;

cMatrInv[i][2][2] = dop9 / det;

}

cudaMemcpyToSymbol(dev\_cAvg, cAvg, sizeof(pixFloat) \* classes, 0, cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(dev\_cMatrInv, cMatrInv, sizeof(matrix3) \* classes, 0, cudaMemcpyHostToDevice);

delete[] cAvg;

delete[] cMatr;

delete[] cMatrInv;

begin(&myImage, classes);

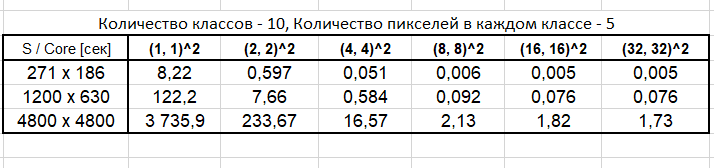
writeToFile(myImage, output);

return 0;

}

**Результаты**

Таблица работы программы с разными конфигурациями ядра GPU (в секундах). Количество классов – 10, точек в 1 классе – 5. Тест с Шабунькой на 1 потоке шел больше часа…

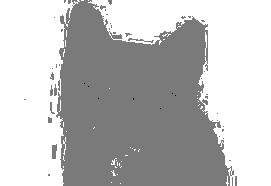


Результаты изображений при классификации.

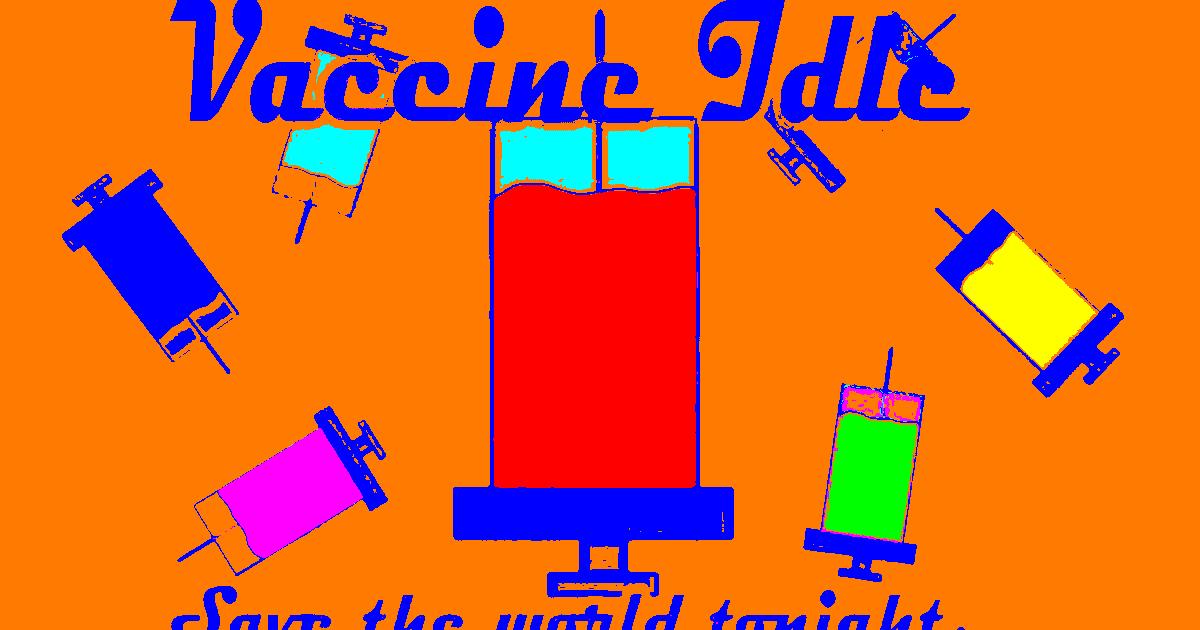
1 пример – 3 класса, 3 цвета (попытка выделить котяру от фона)

2 пример – 7 классов (каждый цвет ~~вакцины~~ лекарства)

3 пример – 3 класса (снова Любимая Шабунька с ее милым носиком :”)





**Выводы**

Данная лабораторная меня побольше познакомила с константной памятью. Удобно работать с ней в похожих задачах, выделяется определенный участок памяти с константами, к которому мы потом просто обращаемся. Также, я сделал задачу классификации цветов. Существует множество похожих алгоритмов, а есть даже нейросети, выполняющие такую задачу. Я познакомился с алгоритмом Малаханобиса, впервые слышу его имя) Мы находим средние вектора, ковариационные матрицы, обращаем их, а потом умножаем матрицы. Вспомнил линал и теорию вероятностей. Программа очень долго не проходила на 10 или 24 тестах. Очень долго мучался, пытался понять в чем проблема? Оказалось, совсем все тупо, я забыл инициализировать переменные внутри цикла, вот до чего может довести моя невнимательность. Пока идет первый тест, я уже заканчиваю писать вывод. Он у меня на 1 потоке с 10 классами и уже идет больше получаса. Это еще раз показывает, как тут важно работать с многими потоками, при 32х32 проходит примерно за 2 секунды на последней картинке, которая размером 4800х4800.