МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №3
по курсу «Программирование графических процессоров»

Изучение технологии CUDA

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

Москва, 2019

Условие

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 5: Метод k-средних.
На первой строке задается путь к исходному изображению,
на второй, путь к конечному изображению. На следующей строке, число nc -- кол-во кластеров. Далее идут nc строчек описывающих начальные центры кластеров. Каждая i-ая строчка содержит пару чисел -- координаты пикселя который является центром. $nc \leq 32$.

Программное и аппаратное обеспечение

GPU:

Device 0: "GeForce GTX 1060 with Max-Q Design"

CUDA Driver Version / Runtime Version 10.1 / 10.1

CUDA Capability Major/Minor version number: 6.1

Total amount of global memory: 6078 MBytes (6373572608 bytes)

(10) Multiprocessors, (128) CUDA Cores/MP: 1280 CUDA Cores

GPU Max Clock rate: 1480 MHz (1.48 GHz)

Memory Clock rate: 4004 Mhz

Memory Bus Width: 192-bit

L2 Cache Size: 1572864 bytes

Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)

Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers

Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers

Total amount of constant memory: 65536 bytes

Total amount of shared memory per block: 49152 bytes

Total number of registers available per block: 65536

Warp size: 32

Maximum number of threads per multiprocessor: 2048

Maximum number of threads per block: 1024

Max dimension size of a thread block (x,y,z): (1024, 1024, 64)

Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

Maximum memory pitch: 2147483647 bytes

Texture alignment: 512 bytes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.1,

NumDevs = 1

CPU:

Архитектура: x86_64

CPU op-mode(s): 32-bit, 64-bit

CPU(s): 12

Потоков на ядро: 2

Ядер на сокет: 6

Имя модели: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz

CPU max MHz: 4100,0000

CPU min MHz: 800,0000

BogoMIPS: 4416.00

Виртуализация: VT-x

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 9216K

Ram:

32 GiB, 2666 MHz

HDD:

500 GB

Программное обеспечение:

Ubuntu 18.04.2 LTS

g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0

nvcc: release 10.1, V10.1.105

Метод решения

Метод k-средних – это метод кластерного анализа, цель которого является разделение m наблюдений на k кластеров, при этом каждое наблюдение относится к тому кластеру, к центру (центроиду) которого оно ближе всего.

В качестве меры близости используется Евклидово расстояние:

$$\rho(x,y) = \|x-y\| = \sqrt{\sum_{p=1}^{n}(x_p - y_p)^2}$$, где $x, y \in R^n$

В случае кластеризации картинки, в метрике участвуют векторы цветовых составляющих.

Рассмотрим первоначальный набор k средних (центроидов) $\mu_1, \cdots, \mu_k$ в кластерах $S_1, S_2, \ldots, S_k$. На первом этапе центроиды кластеров выбираются случайно или по определенному правилу (например, выбрать центроиды, максимизирующие начальные расстояния между кластерами).

Относим наблюдения к тем кластерам, чье среднее (центроид) к ним ближе всего. Каждое наблюдение принадлежит только к одному кластеру, даже если его можно отнести к двум и более кластерам.

Затем центроид каждого i-го кластера перевычисляется по следующему правилу:

$$\mu_j = \frac{1}{s_j} \sum_{x^{(j)} \in S_i} x^{(j)}$$

Таким образом, алгоритм k-средних заключается в перевычислении на каждом шаге центроида для каждого кластера, полученного на предыдущем шаге.

Алгоритм останавливается, когда значения μ_i не меняются: $\mu_i^{\text{шаг } t} = \mu_i^{\text{шаг } t+1}$

Описание программы

Программа состоит из функций: kernel и main.
В main считываются данные со стандартного ввода, заполняется массив данными об исходной картинке. Также считываются начальные положения центроидов на картинке и выделяется память под gpu массив и копируются в него данные о картинке. Далее вызывается основной цикл кластеризации: разбиваем множество на кластеры в соответствии с выбранными центрами, пересчитываем центроиды. После завершения итераций, из функции в main появляетться новый, кластеризированный массив данных.

Функция kernel реализует параллельной разбиение множества на кластеры: каждому пикселю картинки сопоставляется класс.

```
__device__ double SQRT(double A){
        return sqrt(A);
}
__device__ double POW(double a){
    return  a*a;
}
#define CSC(call) do {      \
    cudaError_t e = call;   \
    if (e != cudaSuccess) { \
        fprintf(stderr, "CUDA Error in %s:%d: %s\n"\
        , __FILE__, __LINE__, cudaGetErrorString(e)); \
        exit(0);            \
    }                       \
} while(0)

#define RGBToWB(r, g, b)  0.299*r + 0.587*g + 0.114*b
#define chek(a) a>255.1?255.1:a


//texture<uchar4, 2, cudaReadModeElementType> tex;
__constant__ double3 conTest[33];
__global__ void kernel(uchar4 *dst, int w, int h, int n) {
        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,k;
        uchar4 a;
        double3 b;
        int ans;
        double max,tmp;
        for(i = idx; i < w; i += offsetx){
                for(j = idy; j < h; j += offsety){
                        max=1000000;
                        a = dst[j * w + i];
                        for(k=0;k<n;k++){
                            b=conTest[k];
                            tmp=SQRT(POW(a.x-b.x)+
                            POW(a.y-b.y)+
                            POW(a.z-b.z));
                            //zzz1(b);
                            //printf(" %d %d %f %f\n",j * w + i,k,tmp,max);
                            if(max>tmp){
                                max=tmp;
                                ans=k;
                            }
                        }
                        dst[j * w + i] = make_uchar4(a.x,a.y,a.z, ans);
                }
        }
```

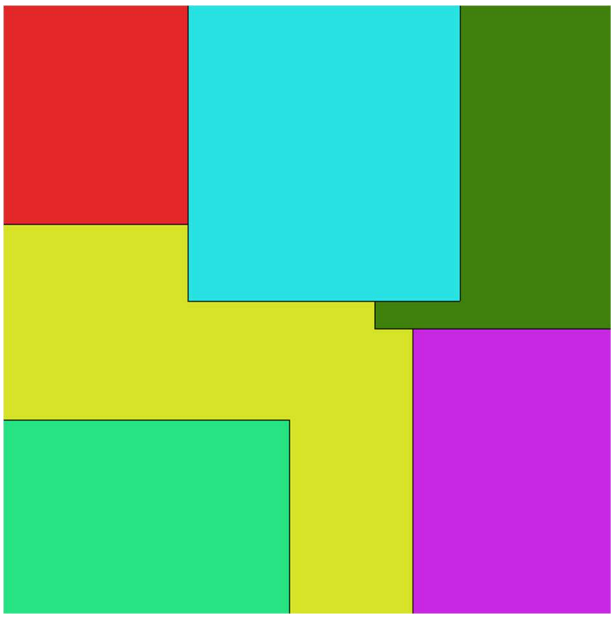
```
}

int main() {
        int h, w;
        char path_in[257];
        char path_out[257];
        scanf("%s", path_in);
        scanf("%s", path_out);
        FILE* in = fopen(path_in, "rb");
        fread(&w, sizeof(int), 1, in);
        fread(&h, sizeof(int), 1, in);
        uchar4 *img = (uchar4 *)malloc(sizeof(uchar4) * h * w);
        fread(img, sizeof(uchar4), h * w, in);
        fclose(in);
        int n,x,y;
        scanf("%d",&n);
        double3 *test = (double3 *)malloc(sizeof(double3) * n);
        double3 *testNext = (double3 *)malloc(sizeof(double3) * n);
        double4 *res = (double4 *)malloc(sizeof(double4) * n);
        for(int i=0;i<n;i++){
            scanf("%d%d",&x,&y);

            test[i].x=img[y*w+x].x;
            test[i].y=img[y*w+x].y;
            test[i].z=img[y*w+x].z;
            /*zzz1(test[i]);
            printf("\n");*/
        }
        /*cudaArray *dev_arr;
        cudaChannelFormatDesc ch = cudaCreateChannelDesc<uchar4>();
        CSC(cudaMallocArray(&dev_arr, &ch, w, h));
        CSC(cudaMemcpyToArray(dev_arr, 0,
         0, img, sizeof(uchar4) * w * h, cudaMemcpyHostToDevice));



        tex.addressMode[0] = cudaAddressModeClamp;
        tex.addressMode[1] = cudaAddressModeClamp;
        tex.channelDesc = ch;
        tex.filterMode = cudaFilterModePoint;
        tex.normalized = false;
        CSC(cudaBindTextureToArray(tex, dev_arr, ch));*/

        uchar4 *dev_img;
        CSC(cudaMalloc(&dev_img, sizeof(uchar4) * w * h));
            CSC(cudaMemcpy ( dev_img, img, sizeof(uchar4)
         * w * h, cudaMemcpyHostToDevice ));
        y=1;
         while (y) {
            /*for(int i=0;i<n;i++){
                zzz1(test[i]);
            }*/
            CSC( cudaMemcpyToSymbol(conTest, test, n*sizeof(double3)) );
```

```
    kernel<<< dim3(32, 32), dim3(32, 32) >>>(dev_img, w, h,n);
    CSC(cudaMemcpy(img, dev_img, sizeof(uchar4)
 * w * h, cudaMemcpyDeviceToHost));

    for(int i=0;i<n;i++){
        res[i].x = res[i].y = res[i].z = res[i].w = 0;
    }
    for(int i = 0; i < w*h; i++){
            //zzz3(img[i]);
            //printf("%d %d \n",i,img[i].w);
            res[img[i].w].x+=img[i].x;
            res[img[i].w].y+=img[i].y;
            res[img[i].w].z+=img[i].z;
            res[img[i].w].w+=1;
            //printf("%f %f     ",res[img[i].w].x,res[img[i].w].w);
    }
    //printf("\n");
    for(int i=0;i<n;i++){
        testNext[i].x=res[i].x/res[i].w;
        testNext[i].y=res[i].y/res[i].w;
        testNext[i].z=res[i].z/res[i].w;
        //zzz1(testNext[i]);
    }
    //printf("NEXT\n");
    y=0;
    for(int i=0;i<n;i++){
        if(test[i].x!=testNext[i].x){
            y=1;
        }
        if(test[i].y!=testNext[i].y){
            y=1;
        }
        if(test[i].z!=testNext[i].z){
            y=1;
        }
    }
    for(int i=0;i<n;i++){
        test[i].x=testNext[i].x;
        test[i].y=testNext[i].y;
        test[i].z=testNext[i].z;
    }

}


FILE* out = fopen(path_out, "wb");
fwrite(&w, sizeof(int), 1, out);
fwrite(&h, sizeof(int), 1, out);
fwrite(img, sizeof(uchar4), h * w, out);
fclose(out);
CSC(cudaFree(dev_img));
free(img);
free(test);
```

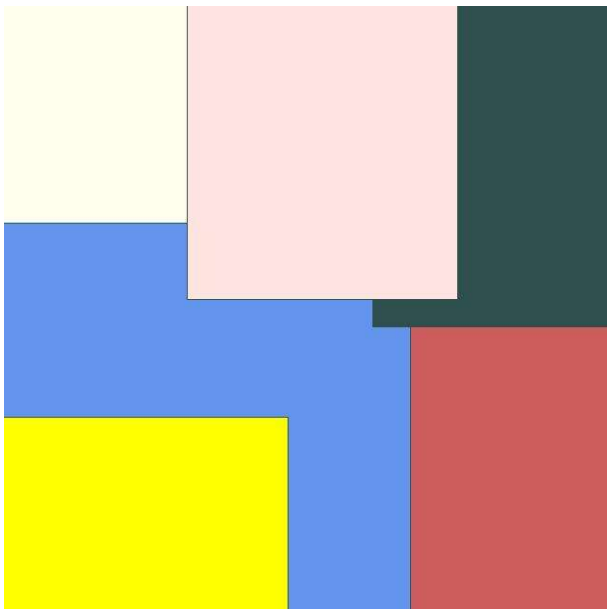
```
        free(testNext);
        free(res);
       return 0;
}
```

Результаты

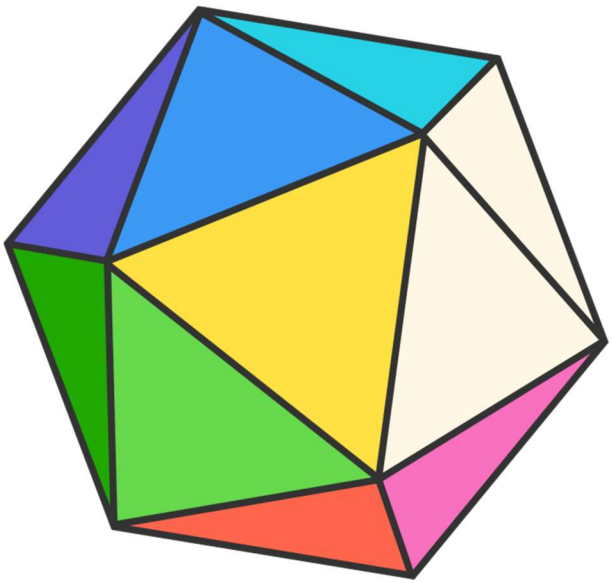
Цвета в конечном изображении подбирались в зависимости от класса пикселя



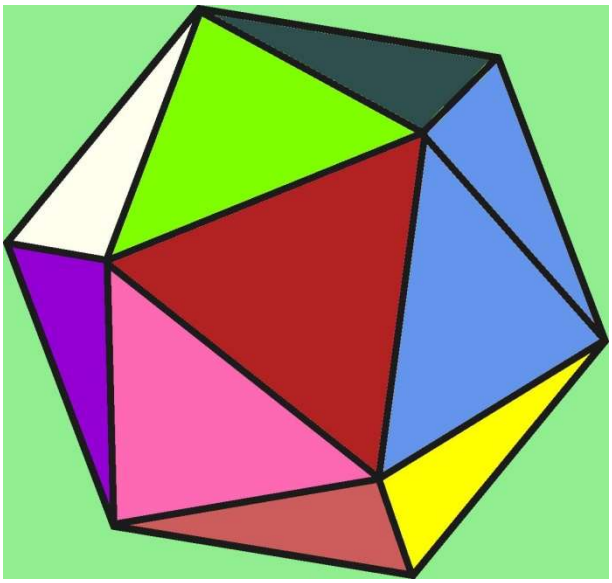
Исходное



Классифицированное



Исходное



Классифицированное

Временя выполнения для первой картинки

Время выполнения	Операция
4.97 ms	[CUDA memcpy DtoH]
1.33 ms	kernel
1.02 ms	[CUDA memcpy HtoD]

Время выполнения для второй картинки

Время выполнения	Операция
4.68 ms	[CUDA memcpy DtoH]
1.72 ms	kernel
1.13 ms	[CUDA memcpy HtoD]

Выводы

Алгоритм используется для кластеризации множества разбивкой на кластеры. Проведенные тесты показали, что алгоритм достойно справляется с задачей. Но не смотря на это у алгоритма имеются недостатки:

- результат зависит от выбора исходных центров кластеров, их оптимальный выбор неизвестен;
- число кластеров надо знать заранее.

Алгоритм активно применяется в области машинного обучения и нейронных сетях (сети векторного квантования сигналов).