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: Гамов Павел Антонович
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Цель работы:

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

Вариант 5:

Метод k-средних.

Программное и аппаратное обеспечение

nvcc 7.0

Ubuntu 14.04 LTS

Compute capability	6.1
Name	GeForce GTX 1050
Total Global Memory	2096103424
Shared Mem per block	49152
Registers per block	65534
Max thread per block	(1024,1024,64)
Max block	(2147483647, 65535, 65535)
Total constant memory	65536
Multiprocessor’s count	5

Метод решения

Использование двухэтапного алгоритма классификации изображений метода k-средних. Даются начальные центры кластеров, коих число фиксировано. Далее первый этап – присвоение ближайшим к кластерам пикселям номера кластеров, к которым их можно отнести. Далее используя расчерченную картинку по кластерам, линейно считаем новые RGB компоненты центров новых векторов как среднее по всем элементам каждых кластеров.

Использованная литература:

https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
http://harmanani.github.io/classes/csc447/Notes/Lecture15.pdf

Описание программы

Считываем картинку в вектор uchar4, создаем побочные структуры для новых центров и их количества.

```
typedef struct { double x, y, z; } D3;
__constant__ D3 CLASSES[32];
__constant__ info inf[1];
```

Используем константную память чтобы хранить RGB для центра каждого кластера. А в структуре inf будем хранить ширину, высоту и количество кластеров, так как эти параметры не меняются в течении программы.

```
__global__ void Kmean(uchar4 * pic) {
   for (int y=blockDim.y*blockIdx.y+threadIdx.y; y<inf[0].h; y+=blockDim.y*gridDim.y){
      for (int x=blockDim.x*blockIdx.x+threadIdx.x;x<inf[0].w;x+=blockDim.x*gridDim.x){
         double maxDist = sqrt((double)3*(255*255))+(double)1;
         uchar4 piv = pic[x + inf[0].w * y];
         for (int i = 0; i < inf[0].n; i++) {
            double pivDist = sqrt(
                  (((double)piv.x-CLASSES[i].x) * ((double)piv.x-CLASSES[i].x)) + \
                  (((double)piv.y-CLASSES[i].y) * ((double)piv.y-CLASSES[i].y)) + \
                  (((double)piv.z-CLASSES[i].z) * ((double)piv.z-CLASSES[i].z)));
            if (pivDist < maxDist) {
                  pic[x + y * inf[0].w].w = (unsigned char)i;
                  maxDist = pivDist;
            }
         }
      }
   }
}
```

Ядро пробегает по всем пикселям для каждого класса считает расстояние до него и если оно лучше и ближе, присваивает пикселю данный класс.

Далее скачиваем обратно картинку, в цикле начинаем считать количество пикселей каждого класса и считать сумму их составляющих.

Как только новые центры становятся равны старым – сразу выходим из вечного цикла.

Результаты

	1500x1500	3000x3000	5000x5000
(32,32), (32,32)	1.8 sec	16.3 sec	16.5 sec

	100x100	500x500	1000x1000
C++	15 sec	1 m 12 sec	2 min 48 sec

Выводы

Использование технологии Cuda позволяет существенно сократить время обработки изображений. В качестве улучшений можно было записать картинку в текстуру, так как изображение не меняется, позволив обрабатывать задачу быстрее, так же можно подумать как можно распараллелить подсчет средних в кластерах, я долго думал, каждое мое решение требовало существенной перестройки кода и выделения дополнительных тяжелых структур, так как количество элементов кластеров меняется, можно было попробовать реализовать асинхронный массив, куда используя mutex

можно было добавлять позиции элементов каждого кластера, но для меня это показалось избыточно и слишком тяжело. Распараллеливание такое как оно есть сейчас уже дает существенный прирост в скорости над линейным.