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

Москва, 2021

Условие

1. **Цель работы**: Научиться использовать GPU для классификации и кластеризации изображений. Использование константной памяти.
2. **Вариант задания: Вариант 5, Метод k-средних.**
   Входные данные: 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

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

OS:Linux
compiler:nvcc
IDE:VS Code
Editor:nano

Метод решения

Основная идея метода k-средних состоит в сопоставлении каждого пикселя с ближайшим по цвету кластером, причем изначально цвета кластеров задаются явно или выбираются из случайных пикселей изображения. Для завершения алгоритма необходимо условие выхода, есть два варианта:
1.Все пиксели на текущей итерации принадлежат тем же классам, как и на предыдущей итерации.
2.Центры классов отличаются по норме не более чем на eps от прошлых значений.

Описание программы

Вся программа была реализована в одном файле **main.cu**.

Реализованное ядро:

```
__global__ void kernel(uchar4* data, size_t n, ulonglong4* cache, uint32_t
clasters_cnt) {

    int id = threadIdx.x + blockIdx.x * blockDim.x;
    int offset = gridDim.x * blockDim.x;

    for (size_t i = id; i < n; i += offset) {
        uchar4& element = data[i];
        element.w = find_best_distance(element, clasters_cnt);

        ulonglong4* tmp_element = &cache[element.w];

        atomicAdd(&tmp_element->x, element.x);
        atomicAdd(&tmp_element->y, element.y);
        atomicAdd(&tmp_element->z, element.z);
        atomicAdd(&tmp_element->w, 1);

    }
}
```

Вспомогательные функции:

```
__device__ float find_dist(const uchar4& a, const float4& b) {
    float x = b.x - float(a.x);
    float y = b.y - float(a.y);
    float z = b.z - float(a.z);
    return x * x + y * y + z * z;
}



__device__ int find_best_distance(const uchar4& point, const int clasters_cnt)
{
    int match = 0;
    float closest = 1e14;

    for (int i = 0; i < clasters_cnt; i++) {
        float curr_dist = find_dist(point, device_mids[i]);
        if (curr_dist < closest) {
            match = i;
            closest = curr_dist;
        }
    }

    return match;
}
```

Результаты

block	threads	time
16	16	105.69ms
16	32	114.51ms
32	16	105.21ms
32	32	109.91ms
64	32	107.45ms
1024	32	104.18ms
CPU		218,75ms

Из полученных данных можно сделать вывод, что при правильных настройках вычисления на **GPU** превосходят **CPU** примерно в **2 раза.**

Выводы

Данный алгоритм можно было бы использовать в задачах кластеризации. Написание кода заняло достаточно много времени, так как пришлось разбираться с устройством константной памяти. Как можно заметить из результатов, выполнение таких алгоритмов на GPU намного эффективнее чем на CPU, благодаря использованию константной памяти и прочих оптимизаций.