МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №2
по курсу «Программирование графических процессоров»

Обработка изображений на GPU. Фильтры.

Выполнил: Д. А. Ваньков
Группа: 8О-407Б-17
Преподаватели: А.Ю. Морозов,
К.Г. Крашенинников

Москва, 2020

Условие

Цель работы: научиться использовать GPU для обработки изображений.
Использование текстурной памяти.

Вариант 6. Метод Превитта.

Программное и аппаратное обеспечение

Graphics card: GeForce 940M
Размер глобальной памяти: 4242604032
Размер константной памяти: 65536
Размер разделяемой памяти: 49152
Максимальное количество регистров на блок: 65536
Максимальное количество потоков на блок: 1024
Количество мультипроцессоров: 3
OS: Linux Ubuntu 18.04
Редактор: CLion, Atom

Метод решения

Для каждого пикселя входного изображения следует выделить по отдельному потоку, каждый из которых будет обрабатывать окрестность этого потока. Поскольку при будет произведено много обращений к памяти, следует воспользоваться текстурной памятью, которая работает быстрее благодаря кэшированию. Результат обработки каждого пикселя будет записываться в выходной массив. Обработка производиться при помощи соответствующего ядра свертки.

Описание программы

Для выполнения данной лабораторной работы мной была реализована специальная структура _image со всеми функциями в отдельном заголовочном файле, для более удобного взаимодействия и передачи данных. Также в отдельном заголовочном файле был реализован фильтр filter и kernel.

Для выполнения операций я создал текстурную ссылку в качестве глобального объекта, применив серый фильтр.

```
__constant__ int g_filter[6];
__device__ double filterGrayScale(Pixel* pixel) {
   return pixel->x * 0.299 + pixel->y * 0.587 + pixel->z * 0.114;
}
```

__global__ void filterPrevittKernel(Pixel* pixel, int width, int height);

Для аппаратной обработки граничных условий я использую Clamp адресацию.

```
Texture2D g_tex;
g_tex.channelDesc = cudaCreateChannelDesc<Pixel>();
g_tex.addressMode[0] = cudaAddressModeClamp;
g_tex.addressMode[1] = cudaAddressModeClamp;
g_tex.filterMode = cudaFilterModePoint;
g_tex.normalized = false;
```

После расчета количества блоков по заданному количеству потоков на каждое из измерений я вызываю kernel:

```
dim3 gridSize(32, 32);
dim3 blockSize(32, 32);
filterPrevittKernel<<<gridSize, blockSize>>> (pixel, w, h);
```

В самом kernel я вычисляю общий индекс исполняемой нити который и будет индексом в массиве при условии idY < height idX < width. Далее производится расчет для соответствующего пикселя по методу Превитта:

```
for (int k = 0; k < 3; ++k) {
    int row = i + k - 1;
    int row_0 = i - 1;
    int row_1 = i + 1;
    int col = j + k - 1;
    int col_0 = j - 1;
    int col_1 = j + 1;
    onePixel = tex2D(g_tex, col_0, row);
    gx += g_filter[k] * filterGrayScale(&onePixel);
    onePixel = tex2D(g_tex, col_1, row);
    gx += g_filter[k + 3] * filterGrayScale(&onePixel);
    onePixel = tex2D(g_tex, col, row_0);
    gy += g_filter[k] * filterGrayScale(&onePixel);
    onePixel = tex2D(g_tex, col, row_1);
    gy += g_filter[k + 3] * filterGrayScale(&onePixel);
}
```

Где g_filter - копия массива filter {-1, -1, -1, 1, 1, 1}.

После вызова kernel я копирую данные в массив на хост и освобождаю выделенную память.

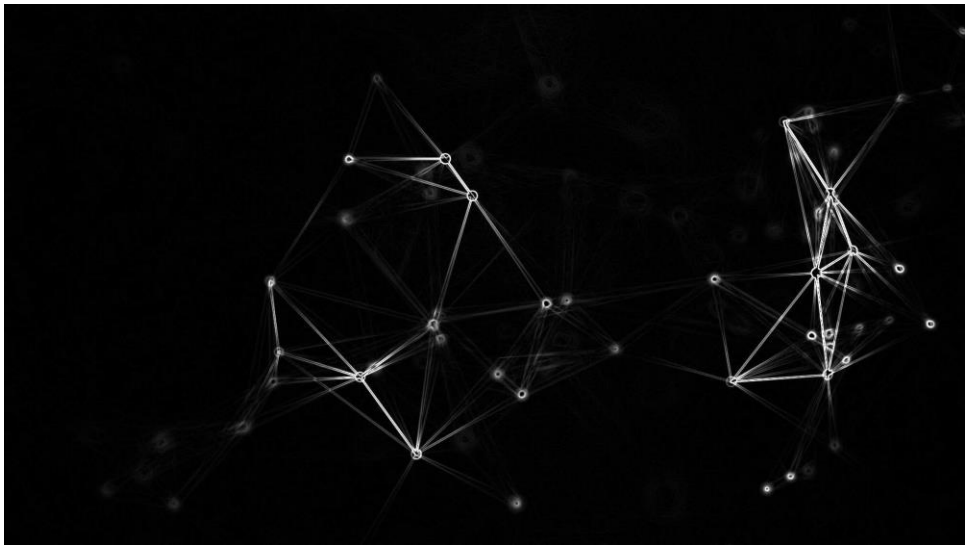
Результаты

Перед использование тестовые изображения нужно было с помощью конвертера конвертировать в нужный формат данных, и после его обратно в исходный формат.
Пример работы алгоритма на выбранных изображениях:

Тест 1.

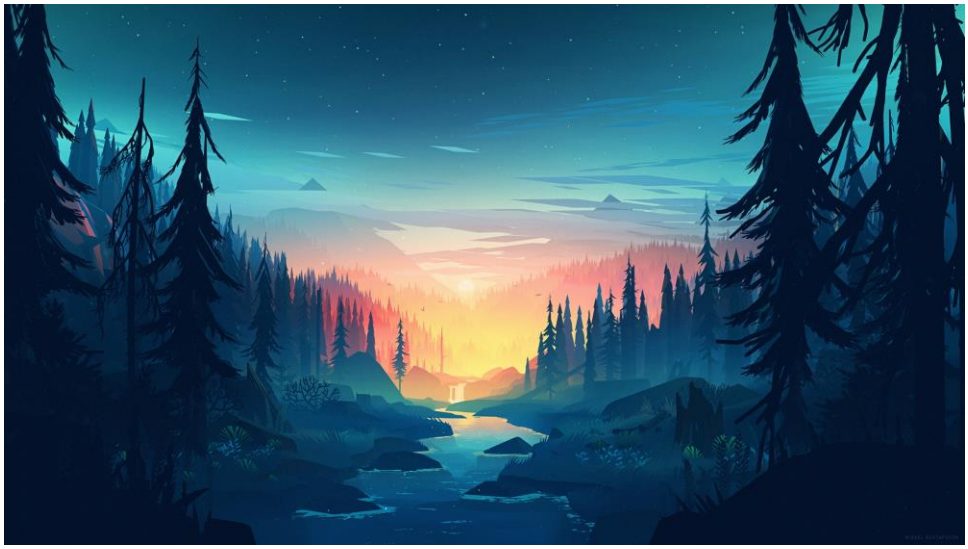
In:

Out:



Тест 2.

In:



Out:

Также я сравнил время работы на этих изображениях с разным количеством запущенных потоков:

Threads, size	Test 1, ms	Test 2, ms
32 * 32	0.023	0.032
64 * 64	0.002	0.003
256 * 256	0.006	0.002
4096 * 4096	0.002	0.003
8192 * 8192	0.003	0.002

Отсюда видно, что начиная с некоторого количества потоков, производительность работы на GPU не возрастает. При этом этот порог для разных размеров изображений разный, что довольно объяснимо, поскольку при большем размере данных требуется большее количество нитей для оптимального распараллеливания алгоритма.

Выводы

Как видно из тестов реализованный мной алгоритм позволяет четко выделить контуры, что бывает полезно в задачах машинного обучения. Однако, можно заметить, что алгоритм является достаточно шумным, то есть на изображении можно заметить светлые размывы. Алгоритм Собеля имеет практически схожие результаты (слегка отличаются коэффициентами), но в некоторых случаях изображения на выходе выглядят лучше.

Алгоритмы свертки хорошо распараллеливаются, что делает их эффективными для использования на графических процессорах.