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 1. Гауссово размытие.

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 я создал дополнительную функцию подсчета коэффициента или веса при заданном радиусе. Также завел массив в константной памяти для более быстрого доступа к элементам, тем самым оптимизировав подсчет коэффициентов, так как я посчитал их до запуска ядра.

```
__constant__ double weights[1024 * 2 + 5];

__host__ double calc(int r) {
    double divisor = 0.0;
    double tmp_array[1024 * 2 + 5];
    for (int i = -r; i <= r; ++i) {
        double tmp = exp((-(double)(i * i)) / (double)(2 * r * r));
        divisor += tmp;
```

```
        tmp_array[i + r] = tmp;
    }
 CUDA_ERROR(cudaMemcpyToSymbol(weights, tmp_array, (1024*2 + 5) * sizeof(double)));
    return divisor;
}
```

Для выполнения операций я создал текстурную ссылку в качестве глобального объекта.

```
texture<uchar4, 2, cudaReadModeElementType> tex;
```

Подготовка текстурной ссылки, настройка интерфейса работы с данными, используя политику обработки выхода за границы по каждому измерению Clamp.

```
tex.channelDesc = channel;
tex.addressMode[0] = cudaAddressModeClamp;
tex.addressMode[1] = cudaAddressModeClamp;
tex.filterMode = cudaFilterModePoint;
tex.normalized = false;
```

После расчета количества блоков по заданному количеству потоков на каждое из измерений я вызываю kernel сначала для горизонтального прохода, затем копирую данные и вызываю для вертикального прохода:

```
gaussian_filter<<<dim3(32, 32), dim3(32, 32)>>>(out_pixels, w, h, r, true, divisor);

CUDA_ERROR(cudaMemcpy(pixels, out_pixels, sizeof(uchar4) * w * h, cudaMemcpyDeviceToHost));
CUDA_ERROR(cudaMemcpyToArray(array, 0, 0, pixels, sizeof(uchar4) * w * h, cudaMemcpyHostToDevice));

 gaussian_filter<<<dim3(32, 32), dim3(32, 32)>>>(out_pixels, w, h, r, false, divisor);
```

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 в зависимости от прохода (горизонтального, вертикального):

```
double cur_func_res = 0.0;
double out_r = 0.0, out_g = 0.0, out_b = 0.0;
for (int k = - r; k <= r; ++k) {
        //cur_func_res = exp((-(double)(k * k)) / (double)(2 * r * r));
        cur_func_res = weights[k + r];
```

```
        // Border case
        double new_y = compare(row, h);
        double new_x = compare(col + k, w);

        // Get pixel from texture
        pixel = tex2D(tex, new_x, new_y);
        // Calculate red, green, blue
        out_r += pixel.x * cur_func_res;
        out_g += pixel.y * cur_func_res;
        out_b += pixel.z * cur_func_res;
}
int idx = col + row * w;
out_pixels[idx].x = out_r / divisor;
out_pixels[idx].y = out_g / divisor;
out_pixels[idx].z = out_b / divisor;
out_pixels[idx].w = 0.0;
```

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.

Raduis: 10

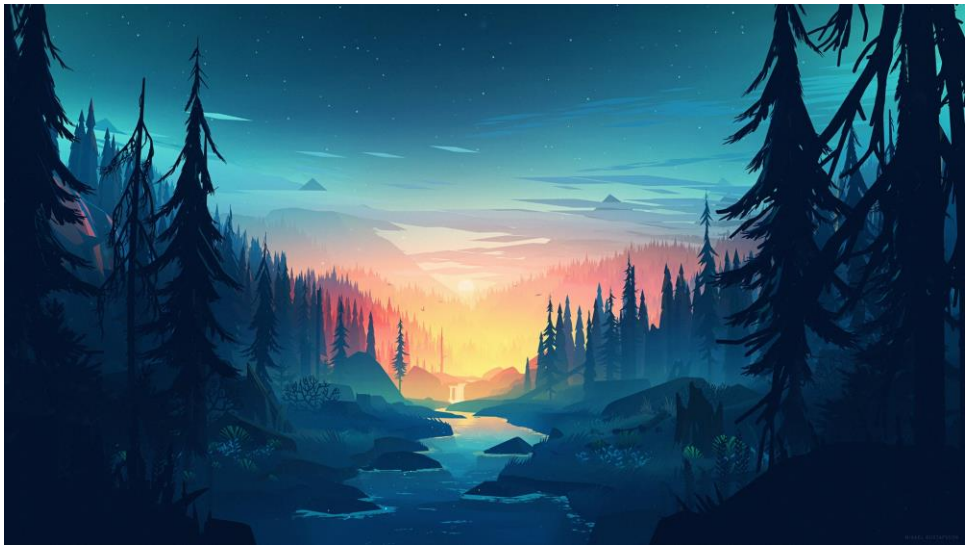
In:

Out:

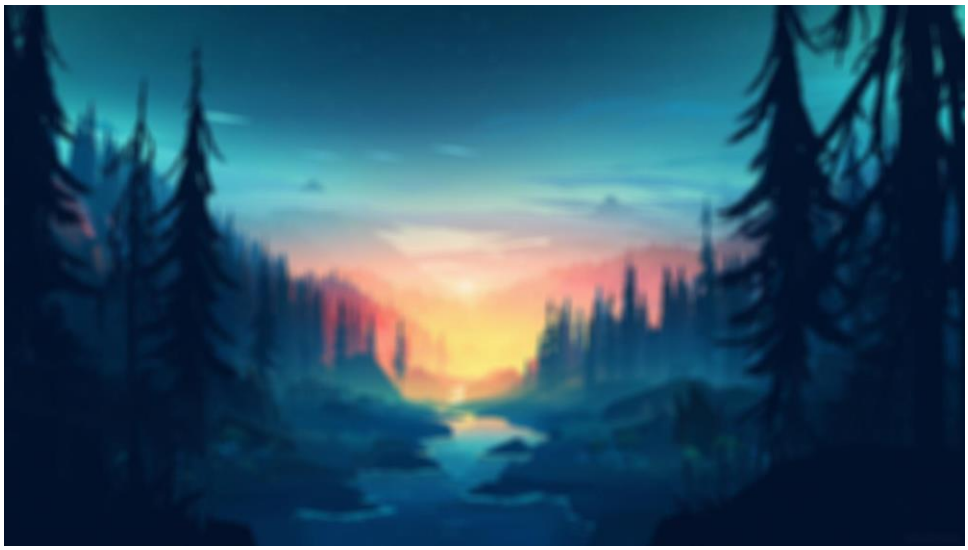
Тест 2.

Raduis: 10

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 данный алгоритм позволяет применять фильтр размытия для изображений. После выполнения этой лабораторной работы я могу примерно представить, как работают фильтры в программах по редактированию изображений со стороны математики.

Во время выполнения возникла проблема оптимизации. Чтобы решить эту проблему, я решил вместо того, чтобы вычислять коэффициент в цикле на каждой итерации, я могу завести массив в константной памяти и записать туда все коэффициенты перед вызовом kernel. Эта оптимизация не даёт асимптотического улучшение, однако существенно сокращает время исполнения программы.