

**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: М.А. Бронников

Группа: 8О-407Б

Преподаватели: К.Г. Крашенинников,  
А.Ю. Морозов

Москва, 2020

## Условие

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

**Вариант 4.** Метод Собеля.

## Программное и аппаратное обеспечение

Device: GeForce GT 545

Размер глобальной памяти: 3150381056

Размер константной памяти : 65536

Размер разделяемой памяти: 49152

Регистров на блок: 32768

Максимум потоков на блок: 1024

Количество мультипроцессоров : 3

OS: Linux Mint 20 Cinnamon

Редактор: VSCode

## Метод решения

Для каждого пикселя входного изображения следует выделить по отдельному потоку, каждый из которых будет обрабатывать окрестность этого потока. Поскольку при этом 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 производится при помощи соответствующего ядра свертки.

## Описание программы

Для выполнения программы я реализовал собственный класс изображения в методе которого и вызывался kernel. Чтобы влезть в ограничения по размеру с учетом размера текстурной памяти я считываю матрицу в транспонированном виде, если высота превышает ширину.

Для выполнения операции я создал текстурную ссылку в качестве глобального объекта, после чего выделил память для массива на девайсе, скопировал в нее массив-изображение и сделал бинд ссылки с массивом:

```
throw_on_cuda_error(  
    cudaMalloc3DArray(&a_data, &cfDesc, {_widtht, _height, 0})  
);  
  
throw_on_cuda_error(  

```

```

        cudaMemcpy2DToArray(
            a_data, 0, 0, _data, _widtht * sizeof(uint32_t),
            _widtht * sizeof(uint32_t), _height, cudaMemcpyHostToDevice
        )
    );

    throw_on_cuda_error(
        cudaBindTextureToArray(g_text, a_data, cfDesc)
    );

```

Для аппаратной обработки граничных условий я использую Clamp адресацию.

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

```

dim3 threads = dim3(MAX_X, MAX_Y);
dim3 blocks = dim3(BLOCKS_X, BLOCKS_Y);

sobel<<<blocks, threads>>>(d_data, _height, _widtht);
throw_on_cuda_error(cudaGetLastError());

```

В самом kernel мы вычисляем общий индекс исполняемой нити который и будет индексом в массиве при условии  $idx < \text{размер массива}$ . Далее производим расчет оператора Собеля для соответствующего пикселя:

```

__global__ void sobel(uint32_t* d_data, uint32_t h, uint32_t w){
    int32_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    int32_t idy = blockIdx.y * blockDim.y + threadIdx.y;
    uint32_t step_x = blockDim.x * gridDim.x;
    uint32_t step_y = blockDim.y * gridDim.y;

    for(int32_t i = idx; i < w; i += step_x){
        for(int32_t j = idy; j < h; j += step_y){
            // ans pixel
            uint32_t ans = 0;

            // locate area in mem(32 bite)
            // compute grey scale for all pixels in area
            float w11 = GREY(tex2D(g_text, i - 1, j - 1));
            float w12 = GREY(tex2D(g_text, i, j - 1));
            float w13 = GREY(tex2D(g_text, i + 1, j - 1));
            float w21 = GREY(tex2D(g_text, i - 1, j));

            float w23 = GREY(tex2D(g_text, i + 1, j));
            float w31 = GREY(tex2D(g_text, i - 1, j + 1));
            float w32 = GREY(tex2D(g_text, i, j + 1));
            float w33 = GREY(tex2D(g_text, i + 1, j + 1));

```

```

// compute Gx Gy
float Gx = w13 + w23 + w23 + w33 - w11 - w21 - w21 - w31;
float Gy = w31 + w32 + w32 + w33 - w11 - w12 - w12 - w13;

// full gradient
int32_t gradf = (int32_t)sqrt(Gx*Gx + Gy*Gy);
// max(gradf, 255)
gradf = gradf > 255 ? 255 : gradf;
// store values in variable for minimize work with global mem
ans ^= (gradf << 16);
ans ^= (gradf << 8);
ans ^= (gradf);

// locate in global mem
d_data[j*w + i] = ans;
}
}
}

```

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

## Результаты

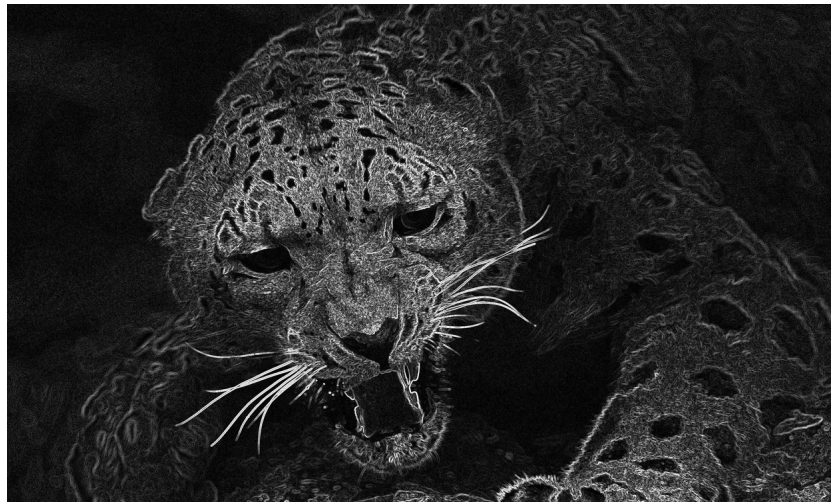
Для иллюстрации результатов работы алгоритма я выбрал 3 изображения:





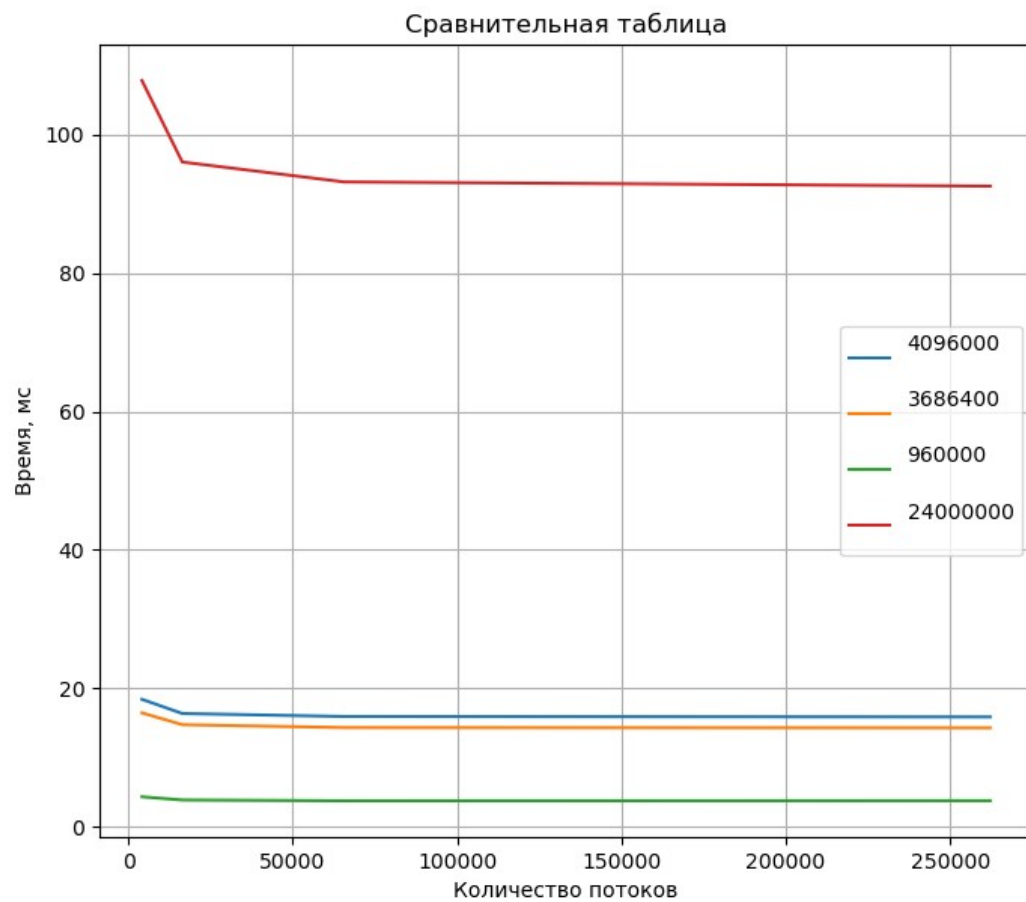


После чего я применил конвертер в заданный в задании формат, который я взял из материалов от преподавателя. После чего я применил к ним свою программу, и конвертировал обратно в jpg. Полученные результаты:





Я посмотрел как зависит время работы на этих изображениях от количества запущенных потоков:



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.

Однако куда интереснее посмотреть на разницу, полученную при запуске алгоритма на этих изображениях на GPU и на CPU:

Img №1:  
GPU threads 65536:  
size: 4096000  
time: 15.9292

CPU:  
size: 4096000  
time: 1643.48

Img №2:  
GPU threads 65536:  
size: 3686400  
time: 14.3269

CPU  
size: 3686400  
time: 1483.8

Img №3:  
GPU threads 65536:  
size: 960000  
time: 3.74608

CPU  
size: 960000  
time: 440.154

Img №4:  
GPU threads: 65536  
size: 24000000  
time: 93.1731

CPU  
size: 24000000  
time: 9604.2

## **Выводы**

Реализованный мной алгоритм широко применяется при обработке изображений, поскольку позволяет четко выделить контуры, что бывает полезно в задачах машинного обучения. При этом этот алгоритм является достаточно шумным(на изображении можно заметить светлые размывы), поэтому перед его применением рекомендуется применять сглаживающие фильтры.

Алгоритмы свертки хорошо распараллеливаются, что делает эффективным их использование на графических процессорах. Недаром, что современные свёрточные нейронные сети обучаются гораздо быстрее на GPU.

В ходе выполнения работы возникла трудность в том, как расположить данные в текстурной памяти так, чтобы влезть в ограничения, а также было совсем не очевидно, что следует решать задач в оттенках серого, а не в 3-ех каналах RGB изображения.