МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2**

**по курсу «Программирование графических процессоров»**

**Обработка изображений на GPU. Фильтры.**

Выполнил: Иларионов Д.А.

Группа: М8О-408Б-17

Преподаватели: Крашенинников К.Г.,

Морозов А.Ю.

Москва, 2020

**Условие**

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

**Вариант: 4. SSAA**

**Программное и аппаратное обеспечение**

**GPU:**

* Name: GeForce GTX 1060
* Compute capability: 6.1
* Частота видеопроцессора: 1404 – 1670 (Boost) МГц
* Частота памяти: 8000 МГц
* Графическая память: 6144 МБ
* Разделяемая память: отсутствует
* Количество регистров на блок: 65536
* Максимальное количество блоков: (2147483647, 65535, 65535)
* Максимальное количество нитей: (1024, 1024, 64)
* Количество мультипроцессоров: 10

**Сведения о системе:**

* Процессор: Intel Core i7-8750H 2.20GHz x 6
* Оперативная память: 16 ГБ
* SSD: 128 ГБ
* HDD: 1000 ГБ

**Программное обеспечение:**

* OS: Windows 10
* IDE: Visual Studio 2019
* Компилятор: nvcc

**Метод решения**

Все блоки и нити в этот раз имеют 2 измерения и каждый поток берет “прямоугольную область” изображения и обрабатывает ее. В данной ЛР не нужно выходить за границы изображения, поэтому, у меня проблем особо и не возникло. Сам вариант представляет собой программу, которая “сглаживает” изображение, а точнее – уменьшает его в несколько раз. Сам фильтр SSAA предназначен для того, чтобы сгладить изображение, убрать некоторые неровности. Например, при уменьшении в 4 раза, в одном из 4 пикселей может быть цвет, который сильно различается от других, поэтому, при сглаживании берется средний цвет. По сути – обычное уменьшение размеров изображения с небольшой потерей качества (ну в зависимости от того, во сколько раз мы уменьшаем изображение). Мы получаем на вход изображение (в формате DATA), выход, и новое разрешение изображения. В условии сказано, что расширение исходного изображение всегда кратно расширению нового, что очень сильно упрощает задачу. Мы находим коэффициенты ps\_X и ps\_Y – во сколько раз изображение уменьшается по X и по Y. затем уже каждый поток обрабатывает участок картинки ps\_X\*ps\_Y (если мало потоков и пикселей больше, то несколько участков с разницей offset по x и по y изображения соответственно). Находим средний цвет и передаем его в соответствующий пиксель новой картинки. Таким образом мы уменьшаем саму картинку. Первую картинку (в формате data) мы храним в текстурной памяти. Она позволяет выходить за границы изображения и с ней удобнее работать с изображениями, хотя, в моем варианты за границы выходить и не надо. К пикселю легко обратиться с помощью функции tex2D. С выходным изображением решил не заморачиваться и представил его в виде массива, не знаю, но мне так удобнее. Задачу я свою выполнил, продемонстрировал первое изображение в текстурной памяти.

**Описание программы**

**Файл kernel.cu**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <iostream>

#include <fstream>

#include <cstdio>

#include <sstream>

#include <iomanip>

#include <math.h>

#include <algorithm>

#include <string>

#include <cuda.h>

using namespace std;

#define CSC(call) \

do { \

cudaError\_t res = call; \

if (res != cudaSuccess) { \

fprintf(stderr, "ERROR in %s:%d. Message: %s\n", \

\_\_FILE\_\_, \_\_LINE\_\_, cudaGetErrorString(res)); \

exit(0); \

} \

} while (0) \

\

typedef uchar4 pixels;

typedef unsigned char bytes;

struct image {

int width;

int height;

pixels\* pixs;

};

image newImage(int w, int h) {

image nIMG;

nIMG.width = w;

nIMG.height = h;

nIMG.pixs = new pixels[w \* h];

return nIMG;

}

image newImage(string filename) {

FILE\* file;

image thisImg;

if ((file = fopen(filename.c\_str(), "rb")) == NULL) {

std::cout << "Can't load image from file" << std::endl;

exit(1);

}

fread(&thisImg.width, sizeof(thisImg.width), 1, file);

fread(&thisImg.height, sizeof(thisImg.height), 1, file);

thisImg.pixs = new pixels[thisImg.width \* thisImg.height];

fread(thisImg.pixs, sizeof(pixels), thisImg.width \* thisImg.height, file);

fclose(file);

return thisImg;

}

void writeToFile(image img, string filename) {

FILE\* file = fopen(filename.c\_str(), "wb");

fwrite(&img.width, sizeof(img.width), 1, file);

fwrite(&img.height, sizeof(img.height), 1, file);

fwrite(img.pixs, sizeof(pixels), img.width \* img.height, file);

fclose(file);

}

string imgToString(image img) {

std::stringstream stream;

stream << img.width << " " << img.height << "\n";

for (int i = 0; i < img.height; i++) {

for (int j = 0; j < img.width; j++) {

int k = i \* img.width + j;

stream << hex << setfill('0') << setw(2) << (int)img.pixs[k].x << setfill('0') << setw(2) << (int)img.pixs[k].y << setfill('0') << setw(2) << (int)img.pixs[k].z << setfill('0') << setw(2) << (int)img.pixs[k].w << " ";

}

stream << "\n";

}

return stream.str();

}

texture<pixels, 2, cudaReadModeElementType> tex;

cudaArray\* c\_arr;

void makeTexture(image\* img) {

int w = img->width;

int h = img->height;

cudaChannelFormatDesc ch = cudaCreateChannelDesc<pixels>();

CSC(cudaMallocArray(&c\_arr, &ch, w, h));

CSC(cudaMemcpyToArray(c\_arr, 0, 0, img->pixs, sizeof(pixels) \* w \* h, cudaMemcpyHostToDevice));

tex.addressMode[0] = cudaAddressModeClamp;

tex.addressMode[1] = cudaAddressModeClamp;

tex.channelDesc = ch;

tex.filterMode = cudaFilterModePoint;

tex.normalized = false;

CSC(cudaBindTextureToArray(tex, c\_arr, tex.channelDesc));

}

\_\_global\_\_ void filterSSAA(pixels\* pixelsOut, int w, int h, int psx, int psy)

{

int tX = blockIdx.x \* blockDim.x + threadIdx.x;

int tY = blockIdx.y \* blockDim.y + threadIdx.y;

int offsetX = gridDim.x \* blockDim.x;

int offsetY = gridDim.y \* blockDim.y;

int imW = w \* psx;

for (int i = tY; i < h; i += offsetY)

{

for (int j = tX; j < w; j += offsetX)

{

pixels thisPixel;

double thisRed = 0.0;

double thisGreen = 0.0;

double thisBlue = 0.0;

for (int Y = psy \* i; Y < psy \* i + psy; ++Y) {

for (int X = psx \* j; X < psx \* j + psx; ++X) {

thisPixel = tex2D(tex, X, Y);

thisRed += thisPixel.x;

thisGreen += thisPixel.y;

thisBlue += thisPixel.z;

}

}

thisRed /= (psx \* psy);

thisGreen /= (psx \* psy);

thisBlue /= (psx \* psy);

bytes nRed = (bytes)min((int)thisRed, (int)0xFF);

bytes nGreen = (bytes)min((int)thisGreen, (int)0xFF);

bytes nBlue = (bytes)min((int)thisBlue, (int)0xFF);

pixelsOut[j + i \* w].x = nRed;

pixelsOut[j + i \* w].y = nGreen;

pixelsOut[j + i \* w].z = nBlue;

pixelsOut[j + i \* w].w = 0;

}

}

}

void begin(image\* image1, image\* image2, int psX, int psY) {

pixels\* newPixels;

makeTexture(image1);

int size2 = sizeof(pixels) \* image2->width \* image2->height;

CSC(cudaMalloc((void\*\*)& newPixels, size2));

dim3 gridSz(32, 32);

dim3 blockSz(32, 32);

filterSSAA << < gridSz, blockSz >> > (newPixels, image2->width, image2->height, psX, psY);

CSC(cudaUnbindTexture(tex));

CSC(cudaFreeArray(c\_arr));

CSC(cudaMemcpy(image2->pixs, newPixels, size2, cudaMemcpyDeviceToHost));

CSC(cudaFree(newPixels));

}

int main()

{

string input;

string output;

int wNew, hNew;

std::cin >> input >> output;

std::cin >> wNew >> hNew;

image myImage = newImage(input);

int PS\_x = 0;

int PS\_y = 0;

if (myImage.width % wNew != 0 || myImage.height % hNew) {

cout << "ERROR: Not prorortional!\n";

return 0;

}

else {

PS\_x = myImage.width / wNew;

PS\_y = myImage.height / hNew;

}

image newIM = newImage(myImage.width / PS\_x, myImage.height / PS\_y);

begin(&myImage, &newIM, PS\_x, PS\_y);

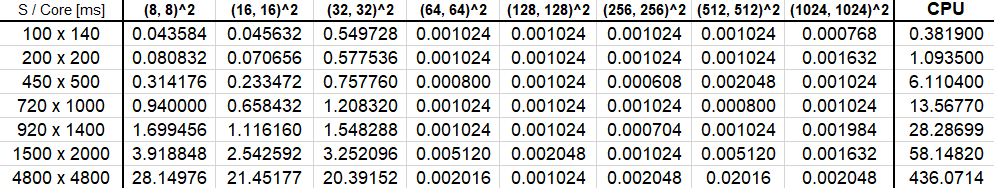
writeToFile(newIM, output);

return 0;

}

**Результаты**

Таблица работы программы с разными конфигурациями ядра GPU и с CPU (в миллисекундах) – уменьшение картинок в 2 раза.



А теперь самое прикольное :D – сами результаты картинок. В качестве образцов я взял фотографии самого милого, маленького кошачье-женского носика Ани Шабуниной, ну и фоточку самой девочки :") Обработанные результаты представляют собой уменьшенные изображения в 4 (2х2) раза. (до, после)

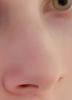
 

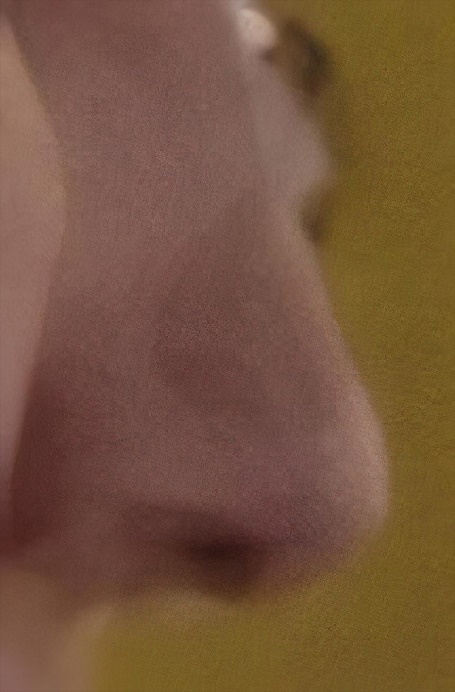
 

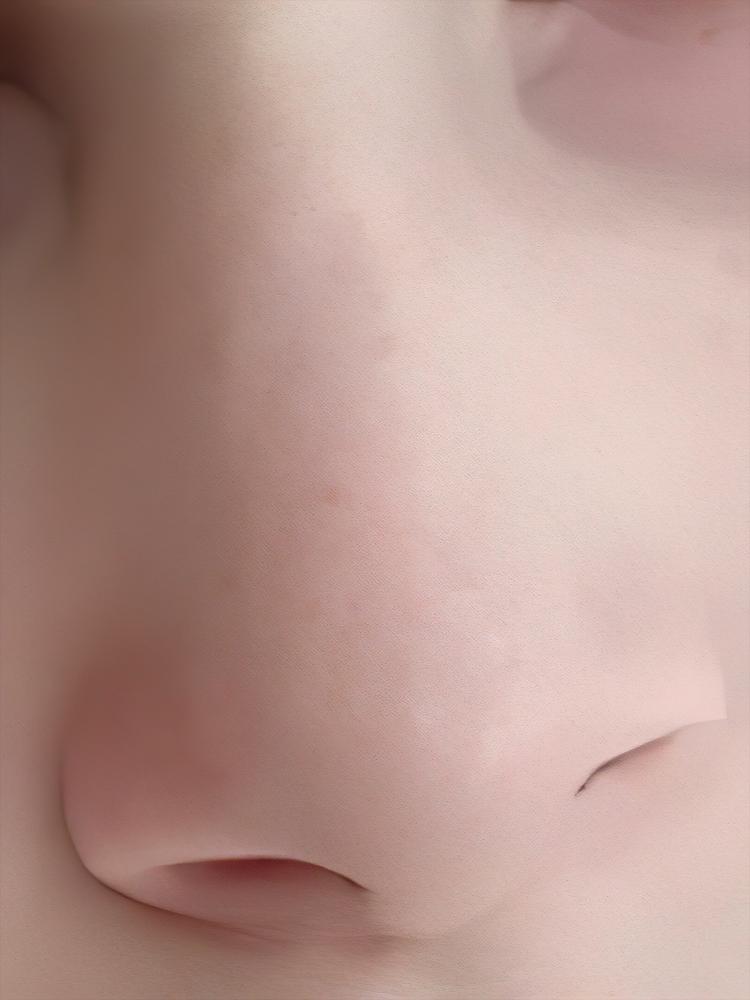
 

А вот уменьшение этого же носика :") в 100 раз (10 x 10)

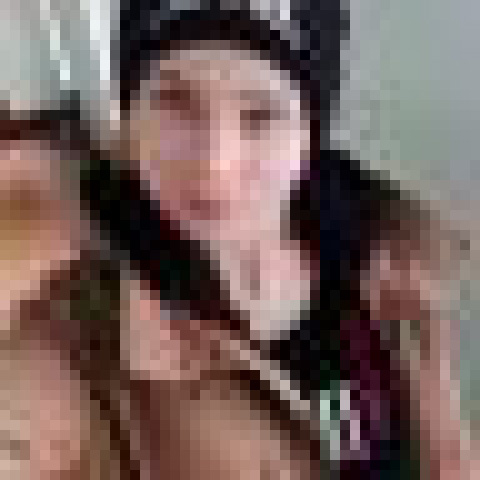
Ну и сама Шабунька)

Как мы видим, на последних фотографиях разница почти незаметна. Дело в том, что разрешение и так очень высокое, и отличить 4800x4800 от 2400x2400 не приближая изображение очень трудно.

А теперь давайте уменьшим Шабуньку в 100 (10х10) и 10,000 (100х100) раз

 - в 100 раз

 - в 10000 раз (48x48)

Также, если уменьшить изображение до размера 1x1, то будет 1 пикселя цвета среднего цвета по всему изображению, рассмотрим несколько примеров.

Средний цвет носика – нежно-розовый немного с темными тонами. Дело в том, что кожа на носике Шабуньки розового цвета :3 Но тон немного темный, потому что на изображении есть тени.

А вот средний цвет Ани совсем другой. Так как на фотографии присутствует темная одежда, стены, пол, поэтому и оттенок темно-сине-коричневый.

Ну а “синька” (посИневший от холода носик Ани Шабуниной в феврале :3) оказалась вовсе и не синей в среднем, а коричневой. Дело в том, что на картинке еще присутствует желтый фон, ну а синевато-розовый и темно желтый цвет дают как раз вместе примерно коричневый.

**Выводы**

В данной лабораторной я уменьшал носики Ани Шабуниной. Он и так у нее очень маленький, так я его сделал и еще меньше, поэтому мне это доставило много милоты и удовольствия) Ну а если серьезно, то я укрепил свои знания работы с GPU, обработки процессов, познакомился с текстурной памятью и хранил пиксели первого изображения в ней. Во втором изображении мы на нижний пиксель спускаемся, просто прибавляя ширину изображения. Я научился производить обработку изображений с использованием CUDA, узнал основной принцип конвертации изображений в файл data для работы с ними. Мы представляем каждый пиксель в виде 16-ричного числа длинной 8 цифр, где каждые 2 цифры (макс число – 255) представляют собой интенсивность каждого цвета в пикселе (RGB), а последние 2 – альфа канал. Однако он нам тут не нужен, поэтому я его переводил просто в нули. Также узнал основной принцип сглаживания/уменьшения изображений, он очень простой, так что у меня с ним особо проблем не возникло. Но в первый раз у меня программа не прошла на 2 тесте. Позже, понял, что это из-за моей невнимательности, я не совсем правильно прочитал условие. Оказывается, программа должна уменьшать изображение не только пропорционально обеим сторонам, ну и может уменьшить в одну сторону больше, чем в другую. Поэтому, я быстренько пофиксил программу, отправил ее и она прошла!