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: В.А. Петросян
Группа: 8О-408Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2020

Условие

Цель работы: Научиться использовать GPU для обработки изображений.
Использование текстурной памяти.
Вариант № 3 "Билинейная интерполяция"

Программное и аппаратное обеспечение

GPU:

1. Compute capability: 3.0
2. Графическая память: 2147155968
3. Разделяемая память: 49152
4. Константная память: 65536
5. Количество регистров на блок: 65536
6. Максимальное количество блоков: (65535, 65535, 65535)
7. Максимальное количество нитей: (1024, 1024, 64)
8. Количество мультипроцессоров: 6

Сведения о системе:

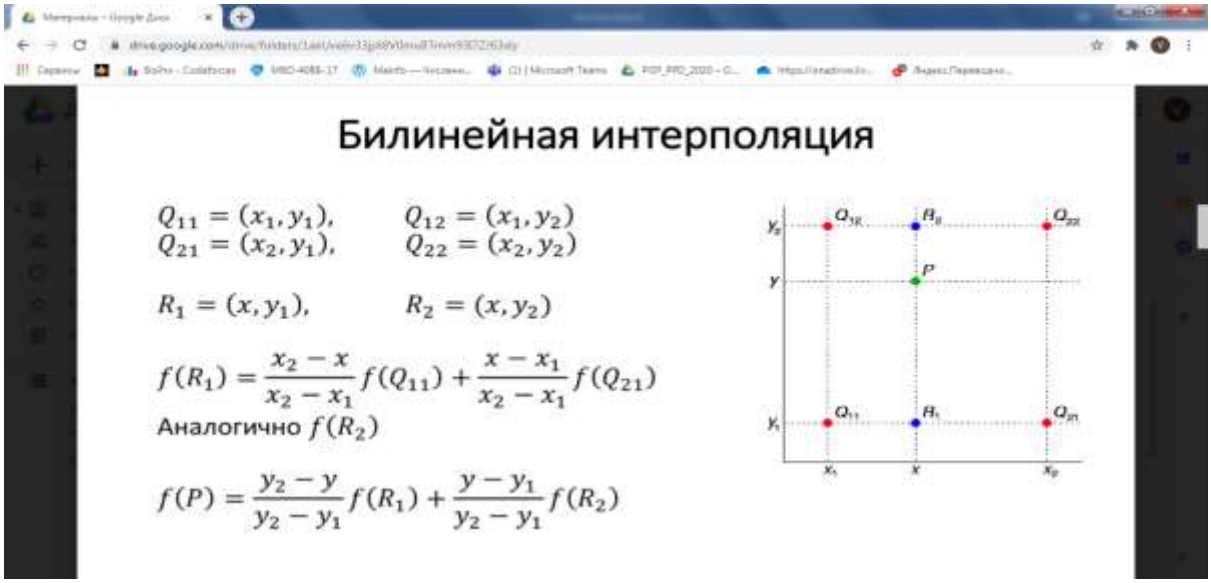
1. Процессор: Intel Core i7-Q720 1.60GHz
2. Память: 8 ГБ
3. HDD: 465 ГБ

Программное обеспечение:

1. OS: Windows 7
2. IDE: Visual Studio 2019
3. Компилятор: nvcc

Метод решения

Сначала интерполируем по одной оси, получаем два значения. После интерполируем использую предыдущие два значения и получаем искомый результат. Замечу, что значения пикселей привязывал к их центрам, а не к углам. Формулу чуть улучшил. В ней больше нет делений, а только умножения.



Описание программы

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <stdio.h>
#include <stdlib.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)

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

```
__global__ void kernel(uchar4* dst, int w, int h, int wNew, int hNew)
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   int idy = blockDim.y * blockIdx.y + threadIdx.y;
   int offsetx = blockDim.x * gridDim.x;
   int offsety = blockDim.y * gridDim.y;

   for (int y = idy; y < hNew; y += offsety) {
      for (int x = idx; x < wNew; x += offsetx) {
         int i = (float)(x + 0.5) * w / wNew - 0.5;
         int j = (float)(y + 0.5) * h / hNew - 0.5;
         float xx = (float)(x + 0.5) * w / wNew - 0.5 - i;
         float yy = (float)(y + 0.5) * h / hNew - 0.5 - j;

         if (xx < (float)0.0) {
            i -= 1;
            xx += (float)1.0;
         }

         if (yy < (float)0.0) {
            j -= 1;
            yy += (float)1.0;
         }
         uchar4 pIJ = tex2D(tex, i, j);
         uchar4 pI1J = tex2D(tex, i + 1, j);
         uchar4 pIJ1 = tex2D(tex, i, j + 1);
         uchar4 pI1J1 = tex2D(tex, i + 1, j + 1);
         //uchar4 res;
         float r = pIJ.x * (1.0f - xx) * (1.0f - yy) + pI1J.x * xx * (1.0f - yy) +
pIJ1.x * (1.0f - xx) * yy + pI1J1.x * xx * yy;
         float g = pIJ.y * (1.0f - xx) * (1.0f - yy) + pI1J.y * xx * (1.0f - yy) +
pIJ1.y * (1.0f - xx) * yy + pI1J1.y * xx * yy;
         float b = pIJ.z * (1.0f - xx) * (1.0f - yy) + pI1J.z * xx * (1.0f - yy) +
pIJ1.z * (1.0f - xx) * yy + pI1J1.z * xx * yy;
         float w = pIJ.w;
         dst[y * wNew + x] = make_uchar4(r, g, b, w);
      }
```

```
    }
    return;
}


int main()
{
    string input;
    string output;
    int wNew, hNew;
    cin >> input >> output >> wNew >> hNew;

    uchar4* data = nullptr;
    int w, h;

    ifstream inputFile(input, std::ios::in | std::ios::binary);
    if (inputFile.is_open()) {
        if (!inputFile.read((char*)&w, sizeof(w))) {
            cerr << "ERROR: can't read from file " << __LINE__ << endl;
            abort();
        }
        if (!inputFile.read((char*)&h, sizeof(h))) {
            cerr << "ERROR: can't read from file " << __LINE__ << endl;
            abort();
        }

        int size = w * h;
        if (size < wNew * hNew) {
            size = wNew * hNew;
        }
        data = new uchar4[size];

        if (!inputFile.read((char*)data, w * h * sizeof(uchar4))) {
            cerr << "ERROR: can't read from file " << __LINE__ << endl;
            abort();
        }
        inputFile.close();
```

```
    }
    else {
        cerr << "ERROR: can't open file " << __LINE__ << endl;
        abort();
    }

    cudaArray* arr;
    cudaChannelFormatDesc ch = cudaCreateChannelDesc<uchar4>();
    CSC(cudaMallocArray(&arr, &ch, w, h));
    CSC(cudaMemcpyToArray(arr, 0, 0, data, sizeof(uchar4) * h * w,
cudaMemcpyHostToDevice));

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

    CSC(cudaBindTextureToArray(tex, arr, ch));

    uchar4* new_image;
    CSC(cudaMalloc(&new_image, sizeof(uchar4) * hNew * wNew));


    kernel <<<dim3(32, 32), dim3(32, 32) >>> (new_image, w, h, wNew,
hNew);
    CSC(cudaGetLastError());

    CSC(cudaMemcpy(data, new_image, sizeof(uchar4) * hNew * wNew,
cudaMemcpyDeviceToHost));

    std::ofstream outputFile(output, std::ios::out | std::ios::binary);
    if (outputFile.is_open()) {
        if (!outputFile.write((char*)&wNew, sizeof(wNew))) {
            cerr << "ERROR: can't open write " << __LINE__ << endl;
            abort();
        }
```

```
        if (!outputFile.write((char*)&hNew, sizeof(hNew))) {
            cerr << "ERROR: can't open write " << __LINE__ << endl;
            abort();
        }
        if (!outputFile.write((char*)data, wNew * hNew * sizeof(uchar4))) {
            cerr << "ERROR: can't open write " << __LINE__ << endl;
            abort();
        }
        outputFile.close();
    }
    else {
        cerr << "ERROR: can't open file " << __LINE__ << endl;
        abort();
    }

    CSC(cudaUnbindTexture(tex));
    CSC(cudaFreeArray(arr));
    CSC(cudaFree(new_image));

    delete[] data;
    return 0;
}
```

Результаты

CPU

w * h	**10^8**	**10^8**	**10^8**	**10^7**	**10^6**
wNew * hNew	**10^6**	**10^7**	**10^8**	**10^8**	**10^8**
Time	**0.09**	**0.44**	**4.05**	**3.68**	**3.65**

GPU <<< (16, 16), (16, 16) >>>

w * h	**10^8**	**10^8**	**10^8**	**10^7**	**10^6**
wNew * hNew	**10^6**	**10^7**	**10^8**	**10^8**	**10^8**
Time	**0.0016**	**0.0101**	**0.1009**	**0.1006**	**0.1005**

Оригинал

1000x500

1000x1000

500x1000

Выводы

Во время выполнения данной лабораторной работы я научился производить обработку изображений с использованием CUDA. Алгоритм билинейной интерполяции применяется в компьютерных играх. Обнаружил его недавно в настройках графики COD1 и COD2. Сложность программирования средняя. Проблемы возникли в реализации кода ядра. Не правильно обрабатывал граничные условия.
Убедился, что текстурная память работает быстрее глобальной.