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.

Студент: Л. Я. Вельтман
Преподаватель: К. Г. Крашенинников
А. Ю. Морозов
Группа: М8О-407Б
Дата:
Оценка:
Подпись:

Москва, 2020

Условие

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

Вариант 4: SSAA.

Необходимо реализовать избыточную выборку сглаживания. Исходное изображение представляет собой экранный буфер, на выходе должно быть сглаженное изображение, полученное уменьшением исходного.

Входные данные: На первой строке задается путь к исходному изображению, на второй, путь к конечному изображению. На следующей строке, два числа wn и hn – размеры нового изображения, гарантируется, что размеры исходного изображения соответственно кратны им. $w * h \leqslant 4 * 10^8$.

Программное и аппаратное обеспечение:

Device Number: 0
Device name: GeForce GT 545
TotalGlobalMem: 3150381056
Const Mem : 65536
Max shared mem for blocks 49152
Max regs per block 32768
Max thread per block 1024
multiProcessorCount : 3
maxThreadsDim 1024 1024 64
maxGridSize 65535 65535 65535
OS: macOS Catalina version 10.15.5
Text Editor: Sublime Text 3

1 Описание

Метод решения

Дискретные артефакты — результат выборки, или, если точнее, недостаток низкого суммарного количества обработанных отсчетов. Избыточная выборка, как видно из названия, решает эту проблему, используя некоторое количество отсчетов на точку, что больше, чем в обычном случае (в обычном случае 1), усредняя их для получения окончательного результата. В этом случае у нас есть возможность более точно смоделировать визуальное представление бесконечно детализированного естественного мира. Итак, первый важный элемент — избыточная выборка — использование дополнительных отсчетов для увеличения плотности данных в изображении. Таким образом, вместо одного центрального отсчета на пиксель, метод избыточной выборки использует несколько.

Описание программы и метод решения

Сконвертируем изображение в бинарное представление при помощи скрипта. Считаем изображение из файла в RAM.

Для работы с текстурной памяти нужно было создать глобальную текстурную ссылку. Далее нужно выделить память для нашего изображения (массива) на девайсе, скопировать в нее массив. Настроить текстурные параметры. Для аппаратной обработки граничных условий я использовала адресный режим clamp. То есть, если координаты текстуры выходят за пределы текстуры: c[k] продолжается за пределами $k = 0, ..., M-1, c[k] = c[0]k < 0c[k] = c[M-1]k \geqslant M$. И сделать бинд массива к текстуре.

В ядре реализован сам алгоритм SSAA. Количество сэмплов на пиксель нового изображения равно числу, которое отвечает во сколько раз было уменьшено исходное изображение по ширине и высоте, назовем его коэффициентом соотношения. Также это можно рассматривать как фильтр, который прикладывается к исходному изображению, размер этого фильтра как раз и есть коэффициент соотношения по высоте x коэффициент соотношения по ширине. Далее при помощи tex2D мы читаем из текстурного сэмплера (из текстуры) цвет её текселя с заданными текстурными координатами. Это и будет цвет выводимого пикселя. По алгоритму SSAA нужно просуммировать цвета сэмплов и поделить на их количество - это и будет конечный цвет одного пикселя.

Вычисляем линейный индекс элемента строки исходной матрицы и записываем туда значение(цвет) найденного пикселя.

Сдвиг на blockDim.x * gridDim.x фактически обеспечит уникальный индекс каждого потока в сетке. Это потому, что blockDim.x будет размером каждого блока, а gridDim.x будет общим количеством блоков.

2 Исходный код

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>


texture<uchar4, 2, cudaReadModeElementType> Texture2D;


void checkCudaError(const char* msg)
{
    cudaError_t err = cudaGetLastError();
    if (cudaSuccess != err)
    {
        fprintf(stderr, "ERROR: %s: %s!\n", msg, cudaGetErrorString(err));
        exit(0);
    }
}



__global__ void SSAA(uchar4 *colorPixels, int width, int height, int proportionWidth,
    int proportionHeight)
{
    int xId = blockDim.x * blockIdx.x + threadIdx.x;
    int yId = blockDim.y * blockIdx.y + threadIdx.y;
    int xOffset = blockDim.x * gridDim.x;
    int yOffset = blockDim.y * gridDim.y;
    int numSample = proportionWidth * proportionHeight;

    /*
    colorPixels = colorSample_0 + colorSample_1 + ... + colorSample_n-1 SUM(
        colorSample_i)
                  ------------------------------------------------------ =
                      ------------------
                               numSample numSample

     colorPixels - the final color of the pixel,
     numSample - the number of samples per pixel,
     colorSample_i - color of the i-th sample.
    */

    for (int col = xId; col < width; col += xOffset)
    {
        for (int row = yId; row < height; row += yOffset)
        {
            int3 colorSample;
            colorSample.x = 0;
```

```
45              colorSample.y = 0;
46              colorSample.z = 0;
47              for (int i = 0; i < proportionWidth; ++i)
48              {
49                  for (int j = 0; j < proportionHeight; ++j)
50                  {
51                      uchar4 pix = tex2D(Texture2D, col * proportionWidth + i, row *
                            proportionHeight + j);
52                      colorSample.x += pix.x;
53                      colorSample.y += pix.y;
54                      colorSample.z += pix.z;
55                  }
56              }
57              colorSample.x /= numSample;
58              colorSample.y /= numSample;
59              colorSample.z /= numSample;
60              // Write to global memory
61              colorPixels[col + row * width] = make_uchar4(colorSample.x, colorSample.y,
                    colorSample.z, 0);
62          }
63      }
64  }
65
66
67
68  int main(int argc, const char* argv[])
69  {
70      std::string input, output;
71      int widthNew, heightNew, width, height;
72      uchar4 *pixels;
73      std::cin >> input >> output >> widthNew >> heightNew;
74
75      FILE* file;
76      if ((file = fopen(input.c_str(), "rb")) == NULL)
77      {
78          std::cerr << "ERROR: something wrong with opening the file!\n";
79          exit(0);
80      }
81      else
82      {
83          fread(&width, sizeof(int), 1, file);
84          fread(&height, sizeof(int), 1, file);
85          if (width >= 65536 || width < 0 || height < 0 || height >= 65536)
86          {
87              std::cerr << "ERROR: incorrect input.\n";
88              exit(0);
89          }
90          //fread(&height, 1, sizeof(int), file);
91          pixels = new uchar4[width * height];
```

```
92          fread(pixels, sizeof(uchar4), width * height, file);
93
94          fclose(file);
95      }
96
97      int proportionWidth = width / widthNew;
98      int proportionHeight = height / heightNew;
99
100     // Allocate CUDA array in device memory
101     cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<uchar4>();
102     cudaArray *array;
103
104     cudaMallocArray(&array, &channelDesc, width, height);
105     checkCudaError("Malloc array");
106
107     cudaMemcpyToArray(array, 0, 0, pixels, sizeof(uchar4) * width * height,
            cudaMemcpyHostToDevice);
108     checkCudaError("Memcpy array");
109
110     // set texture parameters
111     Texture2D.addressMode[0] = cudaAddressModeClamp;
112     Texture2D.addressMode[1] = cudaAddressModeClamp;
113     Texture2D.filterMode = cudaFilterModeLinear;
114     Texture2D.normalized = false; // access with normalized texture coordinates
115
116     // Bind the array to the texture
117     cudaBindTextureToArray(Texture2D, array, channelDesc);
118     checkCudaError("Bind");
119
120     uchar4 *deviceRes;
121     cudaMalloc(&deviceRes, sizeof(uchar4) * widthNew * heightNew);
122     checkCudaError("Malloc");
123
124     // Max quantity of threads is 1024 in one block => sqrt(1024) = 32 is dim
125     const int maxThreads = 1024;
126     dim3 threadsCount = dim3(sqrt(maxThreads), sqrt(maxThreads));
127
128     int xBlockCount = width / maxThreads;
129     int yBlockCount = height / maxThreads;
130
131     if (xBlockCount * maxThreads != width)
132         ++xBlockCount;
133     if (yBlockCount * maxThreads != height)
134         ++yBlockCount;
135
136     dim3 blockCount = dim3(xBlockCount, yBlockCount);
137
138     SSAA<<<blockCount, threadsCount>>>(deviceRes, widthNew, heightNew, proportionWidth,
            proportionHeight);
```

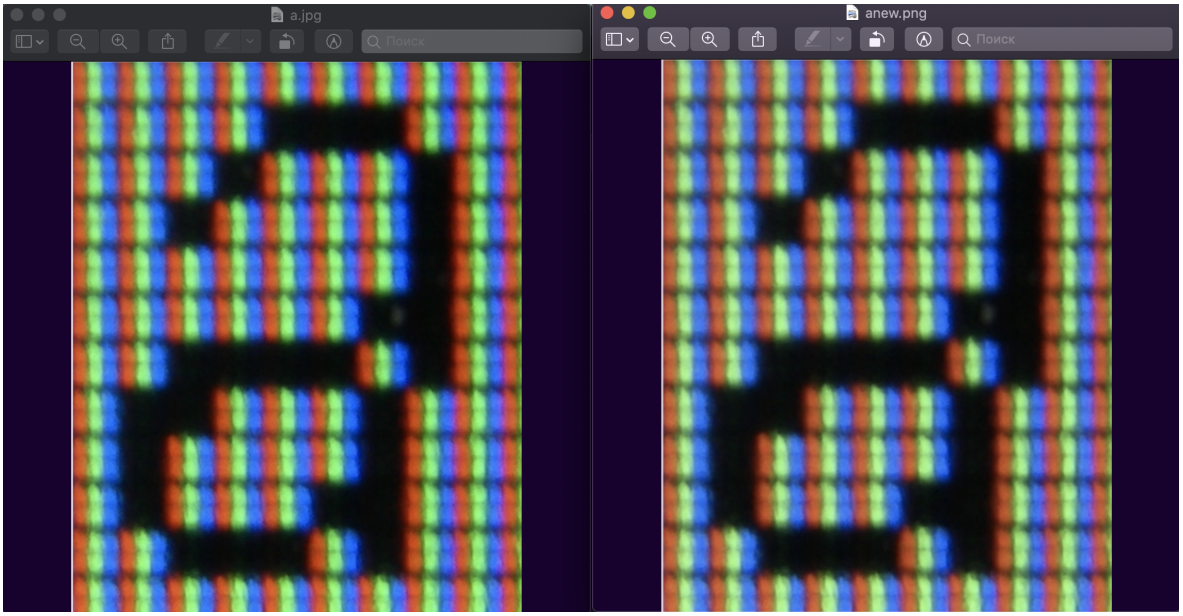
```
139     checkCudaError("Kernel invocation");
140 
141     cudaMemcpy(pixels, deviceRes, sizeof(uchar4) * widthNew * heightNew,
            cudaMemcpyDeviceToHost);
142     checkCudaError("Memcpy");
143 
144     if ((file = fopen(output.c_str(), "wb")) == NULL)
145     {
146         std::cerr << "ERROR: something wrong with opening the file.";
147         exit(0);
148     }
149     else
150     {
151         fwrite(&width, sizeof(int), 1, file);
152         fwrite(&height, sizeof(int), 1, file);
153         fwrite(pixels, sizeof(uchar4), width * height, file);
154 
155         fclose(file);
156     }
157 
158 
159     cudaUnbindTexture(Texture2D);
160     checkCudaError("Unbind");
161 
162     cudaFreeArray(array);
163     checkCudaError("Free");
164 
165     cudaFree(deviceRes);
166     checkCudaError("Free");
167 
168     delete[] pixels;
169     return 0;
170 }
```

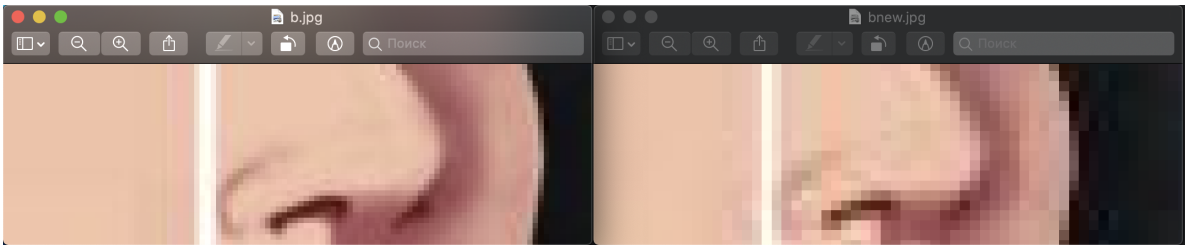
3 Результаты

```
Исходный размер: 844x1034

a.data
aa.data
422
517
CPU
time = 7.001748
GPU
time = 2.142944
blocks = 16
threads = 16


a.data
aa.data
422
517
CPU
time = 2.592399
GPU
time = 0.362656
blocks = 256
threads = 256

Исходный размер: 5760x3840

3086.data
3086new.data
1440
960
CPU
time = 1301.946454
GPU
time = 12.621824
blocks = 256
threads = 256
```

```
3086.data
3086new.data
360
240
CPU
time = 3154.563842
GPU
time = 30.603329
blocks = 256
threads = 256
```

До/после сглаживания





4 Выводы

Для выполнения данной лабораторной работы нужно было реализовать алгоритм избыточной выборки сглаживания (Supersample anti-aliasing, SSAA). Он используется для исправления алиасинга (или «зубцов») на полноэкранных изображениях. SSAA было доступно на ранних видеокартах, вплоть до DirectX 7. Начиная с DirectX 8 из-за огромной вычислительной сложности было заменено всеми производителями графических процессоров на множественную выборку сглаживания, который также был заменён другими методами, такими как CSAA + TrAA/AAA. Поскольку SSAA даёт более высокое качество изображения, он не был полностью исключён и до сих пор реализуется аппаратно в продуктах AMD и NVIDIA. Качество сглаживания ограничено пропускной способностью видеопамяти, поэтому GPU с быстрой памятью сможет просчитать полноэкранное сглаживание с меньшим ущербом для производительности, чем GPU более низкого класса.