МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №*2***

**по курсу «Программирование графических процессоров»**

***Обработка изображений на GPU. Фильтры.***

Выполнил: *А.Е. Аксенов*

Группа: *8О-408Б*

Преподаватели: К.Г. Крашенинников,

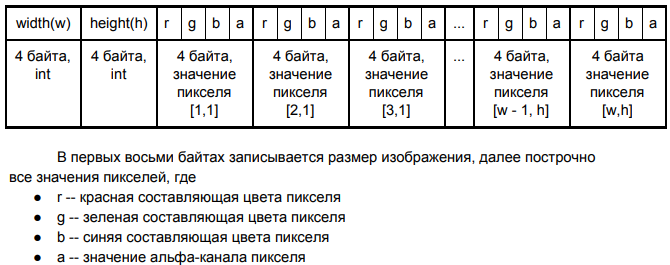
А.Ю. Морозов

Москва, 2023

**Условие**

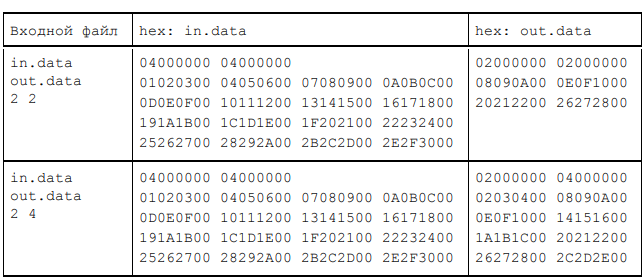
Цель работы. Научиться использовать GPU для обработки изображений. Использование текстурной памяти и двухмерной сетки потоков.

Формат изображений. Изображение является бинарным файлом, со следующей структурой:



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 ≤ 4 \* 10e8 .



**Программное и аппаратное обеспечение**

|  |  |
| --- | --- |
| Графический процессор | |
| GPU | NVIDIA RTX 2070 |
| Compute Capability | 7.5 |
| Объем видеопамяти | 8192 МБ |
| Тип видеопамяти | GDDR6 |
| Разделяемая память | 48 КБ |
| Константная память | 64 КБ |
| Число регистров в блоке | 65636 |
| Максимальное число блоков | 16 |
| Максимальное число нитей | 1024 |
| Количество мультипроцессоров | 34 |
| Центральный процессор | |
| СPU | Intel Core i5 9600K |
| Число физических ядер | 6 |
| Максимальное число потоков | 6 |
| Базовая частота | 3.6 GHz |
| Частота Turbo Boost | 4.5 GHz |
| Объем кэша L3 | 9 МБ |
| Оперативная память | |
| Тип оперативной памяти: | DDR4 |
| Частота памяти | 2666MHz |
| Объём | 32 ГБ |
| Постоянная память | |
| Тип постоянной памяти | SSD |
| Объём постоянной памяти | 1024ГБ |
| Программное обеспечение | |
| Операционная система | Microsoft Windows 11 |
| Версия операционной системы | 2004 |
| Среда разработки | Microsoft Visual Studio 2022 |
| Версия среды разработки | 16.7.3 |
| NVIDIA CUDA Toolkit | 12.2 |

**Метод решения**

1. Считываем размеры результирующего изображения.
2. Выделяем память в основной памяти компьютера.
3. Считываем входное изображение из файла.
4. Выделяем cuda массив на графическом ускорителе.
5. Копируем входное изображение в выделенный массив.
6. Привязываем текстурную ссылку к имеющемуся массиву.
7. Конфигурируем параметры текстурной ссылки.
8. Выделяем графическую память под выходное изображение.
9. Запускаем ядро с оптимальными параметрами, при этом каждая нить вычисляет усреднённый цвет в пределах своего блока изображения и записывает этот цвет в соответствующий пиксель выходного изображения. Размер блока фиксированный – 32 x 32. Размеры сетки подбираются в зависимости от изображения.
10. Копируем результаты обратно в основную память компьютера.
11. Выводим результаты в файл в соответствии с требованиями.

**Описание программы**

Программа реализована в виде одного файла с исходным кодом. В ней имеется реализация одного единственного ядра, которая сопровождается квалификатором \_\_global\_\_. Используются базовые типы, определяемые NVIDIA CUDA API. В частности, для того, чтобы описать архитектуру сетки и блоков, используется тип dim3, который представляет из себя кортеж из трех целочисленных чисел, каждое из которых описывает размеры сетки или блока в одном из трех измерений. Обрабатываемые изображения хранятся в массиве cuda с типом uchar4. В глобальной области видимости определена текстурная ссылка. Реализация ядра предполагает, что конфигурация сетки и блоков выполнена в двумерном формате. Далее, каждая нить определяет свои координаты относительно всей сетки и выполняет усреднение соответствующих элементов. Усреднение выполняется при условии, что координаты нити попадают в область входного изображения. Ниже представлена реализация ядра:

\_\_global\_\_ void ssaa\_kernel(ssaa\_context\_t context) {

    const int32\_t block\_index = gridDim.x \* blockIdx.y + blockIdx.x;        // линейный индекс блока относительно сетки

    const int32\_t thread\_index = blockDim.x \* threadIdx.y + threadIdx.x;    // линейный индекс потока относительно блока

    const int32\_t block\_size = blockDim.x \* blockDim.y;                     // размер блока

    const int32\_t ssaa\_volume = context.width\_ratio \* context.height\_ratio; // размер блока сглаживания

    const int32\_t grid\_size = gridDim.x \* gridDim.y \* block\_size;           // размер сетки в потоках

    const int32\_t image\_size = context.width \* context.height;              // размер изображения

    for(int32\_t k = block\_index \* block\_size + thread\_index;

                            k < image\_size; k += grid\_size)  {

        int32\_t x = (k % context.width) \* context.width\_ratio;               // координаты пикселя

        int32\_t y = (k / context.width) \* context.height\_ratio;              // координаты пикселя

        int32\_t sx = 0, sy = 0, sz = 0, sw = 0;                              // суммы по значениям каналов сглаживаемого блока

        for(int32\_t i = 0; i < context.width\_ratio; ++i){                    // суммируем каналы в блоке сглаживания

            for(int32\_t j = 0; j < context.height\_ratio; ++j){

                uchar4 p = tex2D(t\_ref, x + i, y + j);

                sx += static\_cast<unsigned int>(p.x);

                sy += static\_cast<unsigned int>(p.y);

                sz += static\_cast<unsigned int>(p.z);

            }

        }

        context.out[k].x = static\_cast<unsigned char>(sx / ssaa\_volume);    // усредняем эти значения и записываем

        context.out[k].y = static\_cast<unsigned char>(sy / ssaa\_volume);

        context.out[k].z = static\_cast<unsigned char>(sz / ssaa\_volume);

        context.out[k].w = static\_cast<unsigned char>(sw / ssaa\_volume);

    }

}

**Результаты**

ОРИГИНАЛ



SSAA (2x2)



SSAA(4x4)



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размеры изображения | Количество пикселей | Параметры SSAA | Ядро | Время GPU | Время CPU |
| 8715 x 5536 | 48246240 | 5 : 8 | (273 173) (32 32) | 0.237568 | 0.1 |
| 7512 x 10185 | 76509720 | 2 : 7 | (235 319) (32 32) | 0.536576 | 0.381 |
| 4158 x 33300 | 138461400 | 2 : 3 | (130 1041) (32 32) | 1.44867 | 1.078 |
| 23190 x 6975 | 16170250 | 3 : 5 | (725 218) (32 32) | 1.1063 | 1.185 |
| 28405 x 7672 | 217923160 | 5 : 7 | (888 240) (32 32) | 1.13846 | 1.207 |
| 29920 x 8982 | 268741440 | 2 : 6 | (935 281) (32 32) | 2.07037 | 3.813 |
| 27528 x 12425 | 342035400 | 3 : 5 | (861 389) (32 32) | 2.36083 | 3.746 |
| 22165 x 18036 | 399767940 | 5 : 3 | 693 564) (32 32) | 2.7815 | 3.868 |

**Выводы**

Рассмотренный алгоритм применяется в игровой индустрии для того, чтобы значительно улучшить качество изображения. При этом стоит отметить его ресурсоёмкость – для его применения требуется достаточно современное аппаратное обеспечение. В прошлом компания NVIDIA даже убрала его поддержку из своих продуктов в силу вышесказанных причин, но затем вернула обратно. Результаты тестирования показывают значительную временную разницу между параллельными и последовательными вычислениями. Так или иначе, для довольно маленьких объёмов обрабатываемых данных выгоднее произвести обработку посредством центрального процессора, т.к. копирование данных в графический процессор, планирование вычислений и их запуск требует некоторого времени. Важно сконфигурировать оптимальную конфигурацию ядра и учитывать специфику работы графической памяти, желательно повышать локальность обращений. Текстурная память позволяет упростить обработку изображения, взяв на себя некоторые часто используемые операции, а также помогает ускорить доступ к данным путём их занесения в кэши мультипроцессоров.