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: Полей-Добронравова Амелия
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников, А.Ю. Морозов

Москва, 2021

Условие

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

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

width(w)	height(h)	r	g	b	a	r	g	b	a	r	g	b	a	...	r	g	b	a	r	g	b	a
4 байта, int	4 байта, int	4 байта, значение пикселя [1,1]				4 байта, значение пикселя [2,1]				4 байта, значение пикселя [3,1]				...	4 байта, значение пикселя [w - 1, h]				4 байта значение пикселя [w,h]			

В первых восьми байтах записывается размер изображения, далее построчно все значения пикселей, где

- r -- красная составляющая цвета пикселя
- g -- зеленая составляющая цвета пикселя
- b -- синяя составляющая цвета пикселя
- a -- значение альфа-канала пикселя

Пример картинки размером 2 на 2, синего цвета, в шестнадцатеричной записи:

```
02000000 02000000 0000FF00 0000FF00 0000FF00 0000FF00
```

Студентам предлагается самостоятельно написать конвертер на *любом* языке программирования для работы с вышеописанным форматом.

В данной лабораторной работе используются только цветовые составляющие изображения (r g b), альфа-канал не учитывается. При расчетах значений допускается ошибка в ±1. Ограничение: $w < 2^{16}$ и $h < 2^{16}$. Во всех вариантах, кроме 2-го и 4-го, в пограничном случае, необходимо “расширять” изображение за его границы, при этом значения соответствующих пикселей дублируют граничные. То есть, для любых индексов i и j, координаты пикселя [ip, jp] будут определятся следующим образом:

```
ip := max(min(i, h), 1)
jp := max(min(j, w), 1)
```

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. $w*h \leq 4 * 10^8$.

Пример:

Входной файл	hex: in.data	hex: out.data

in.data out.data 2 2	04000000 04000000 01020300 04050600 07080900 0A0B0C00 0D0E0F00 10111200 13141500 16171800 191A1B00 1C1D1E00 1F202100 22232400 25262700 28292A00 2B2C2D00 2E2F3000	02000000 02000000 08090A00 0E0F1000 20212200 26272800
in.data out.data 2 4	04000000 04000000 01020300 04050600 07080900 0A0B0C00 0D0E0F00 10111200 13141500 16171800 191A1B00 1C1D1E00 1F202100 22232400 25262700 28292A00 2B2C2D00 2E2F3000	02000000 04000000 02030400 08090A00 0E0F1000 14151600 1A1B1C00 20212200 26272800 2C2D2E00

Программное и аппаратное обеспечение

Компилятор nvcc версии 7.0(g++ версии 4.8.4) на 64-х битной Ubuntu 14.04 LTS.
Параметры графического процессора:
Compute capability : 6.1
Name : GeForce GTX 1050
Total Global Memory : 2096103424
Shared memory per block : 49152
Registers per block : 65536
Max threads per block : (1024, 1024, 64)
Max block : (2147483647, 65535, 65535)
Total constant memory : 65536
Multiprocessors count : 5

Метод решения

Каждый поток начинает работать с точки картинки, координаты которой равны номеру потока сетки по x и y. От нёё выделяется прямоугольник размерами diff_w, diff_h. Это переменные, равные отношению размера исходной картинки к размеру итоговой картинки. По этому квадрату считается сумма по каждому каналу цвета, и с помощью деления этой суммы на diff_w*diff_h мы получаем среднее значение по всему квадрату. Его мы записываем как одну точку в итоговый массив out, её номер в массиве равен номеру квадрата: номер потока в сетке по y умножить на ширину итоговой картинки плюс номер потока в сетке по x. Но из-за того, что потоков может не хватить на все такие квадраты, у меня два цикла для x и y, которые перемещают на начало следующего квадрата через offset_x и offset_y квадратов, где offset - общее число потоков.

Описание программы

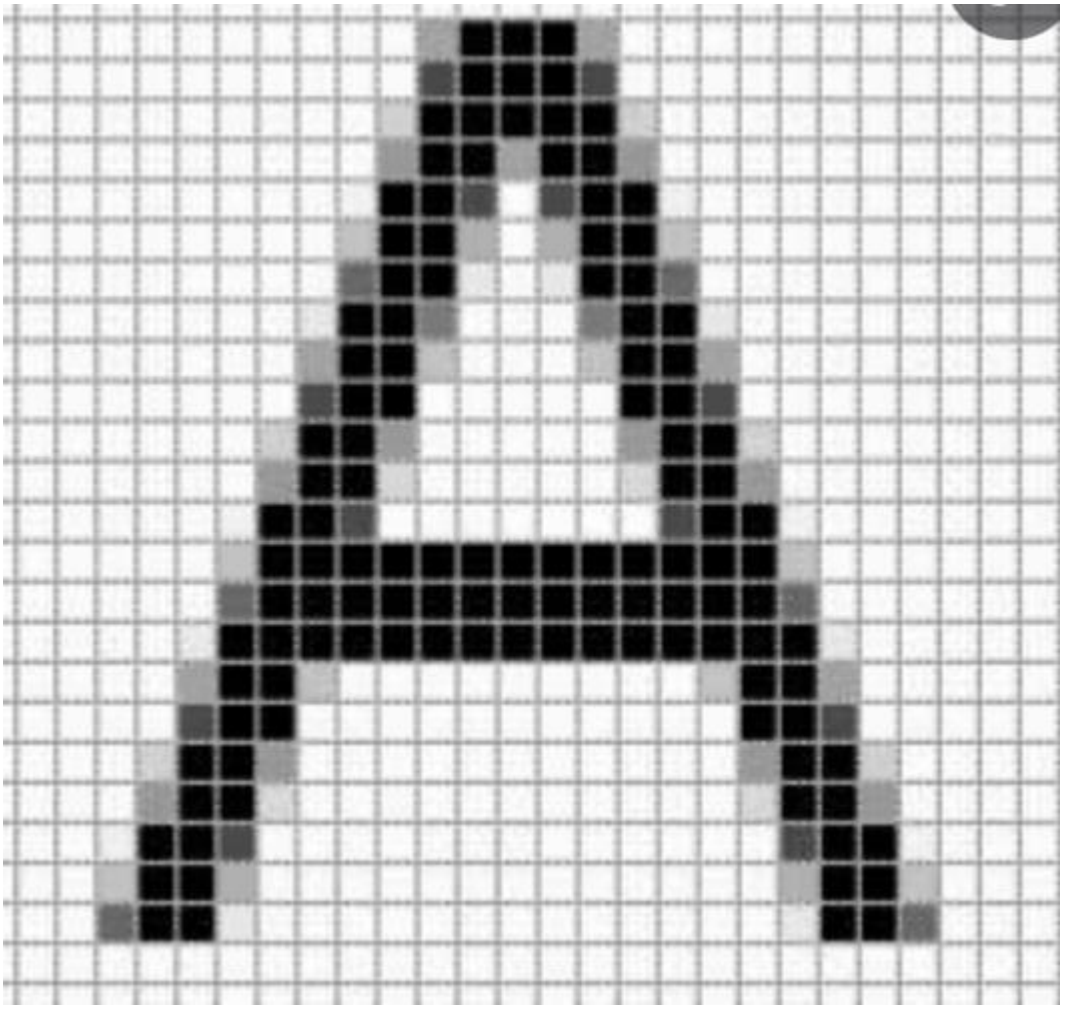
Макрос **CSC** - макрос для отслеживания ошибок со стороны GPU, вызывается около функций для cuda и выводит текст ошибки при cudaError_t не равным cudaSuccess.
kernel - обработка массивов по потокам gpu.

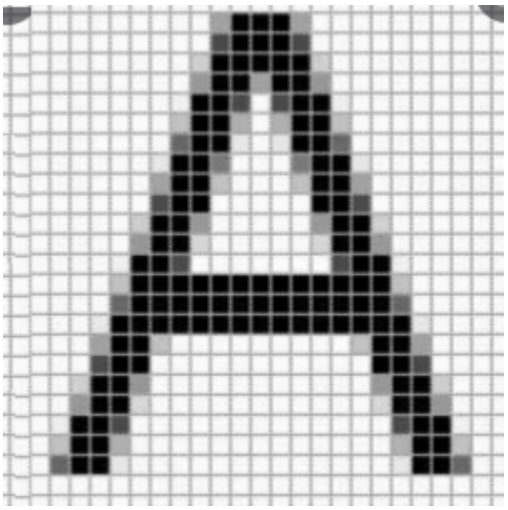
main - ввод данных, подготовка для передачи данных kernel для обработки, вывод результата.

texture<uchar4, 2, cudaReadModeElementType> tex - текстурная ссылка <тип элементов, размерность, режим нормализации>

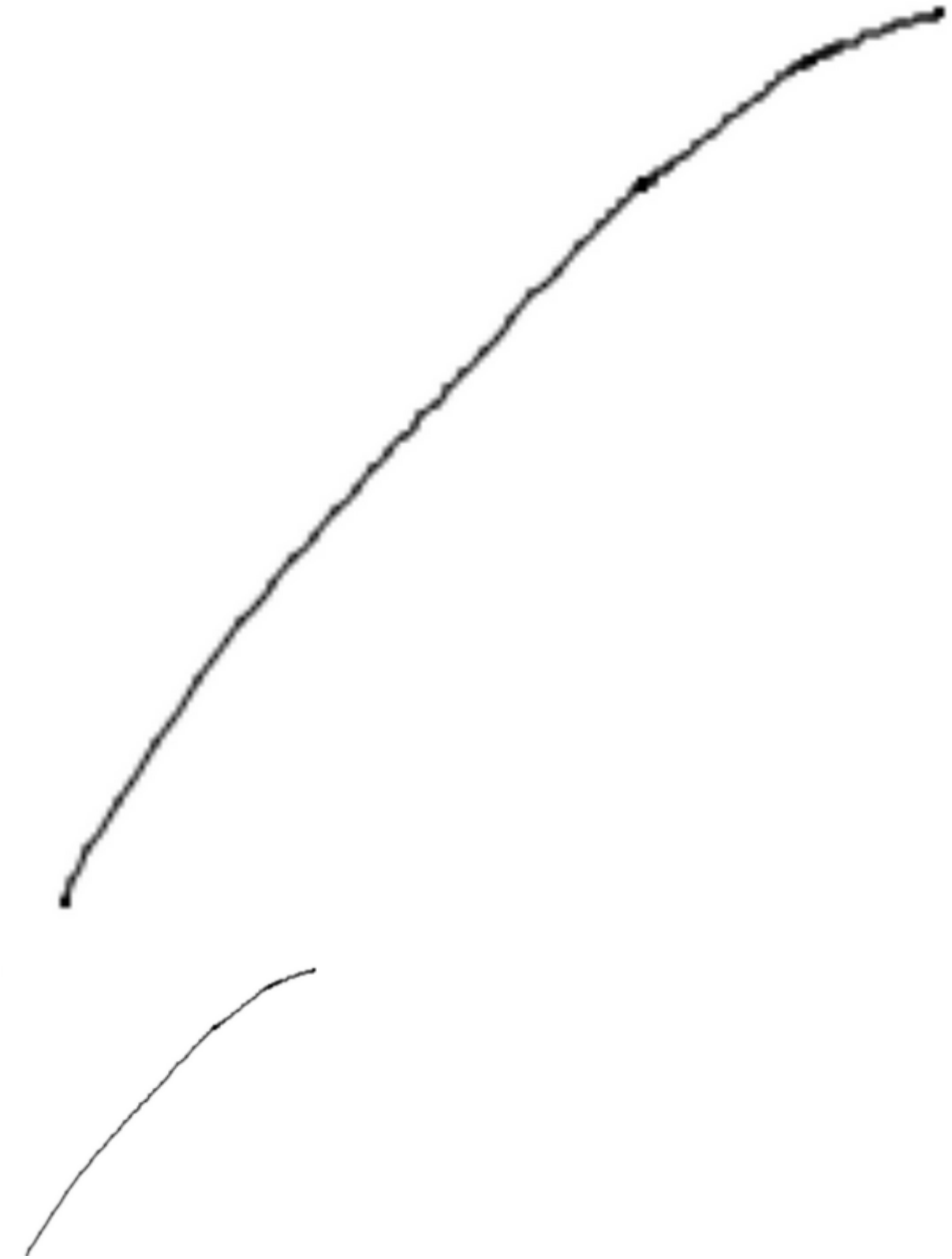
Примеры работы уменьшения в два раза

1)

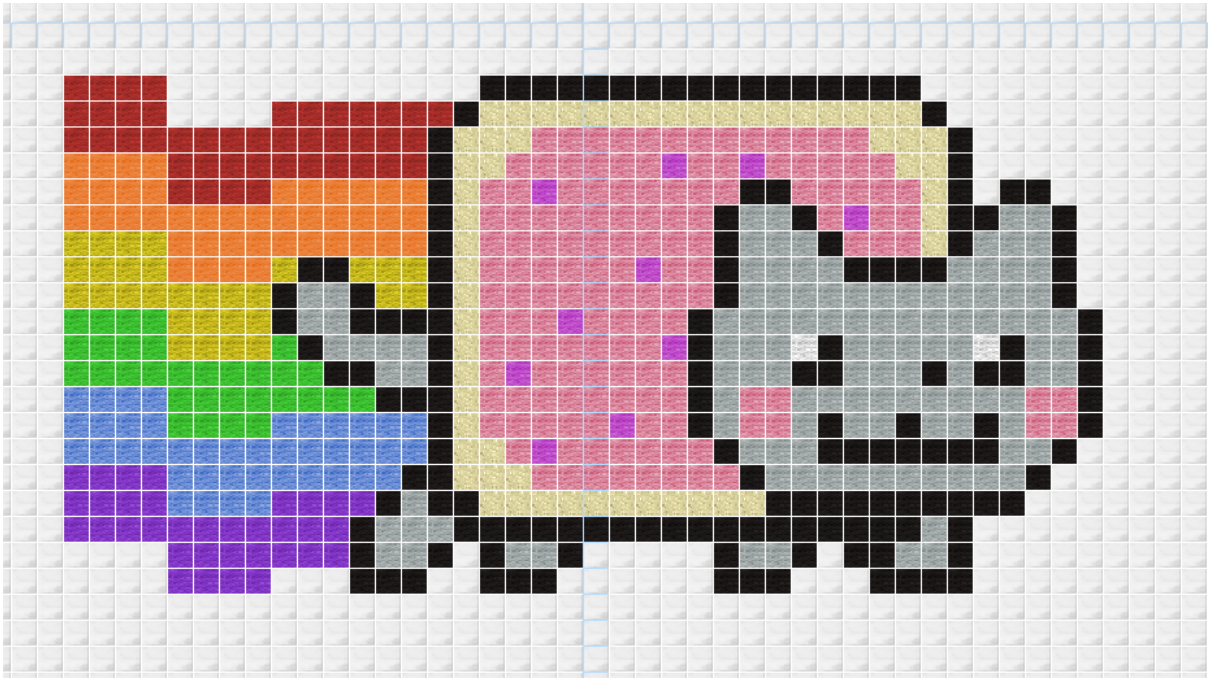


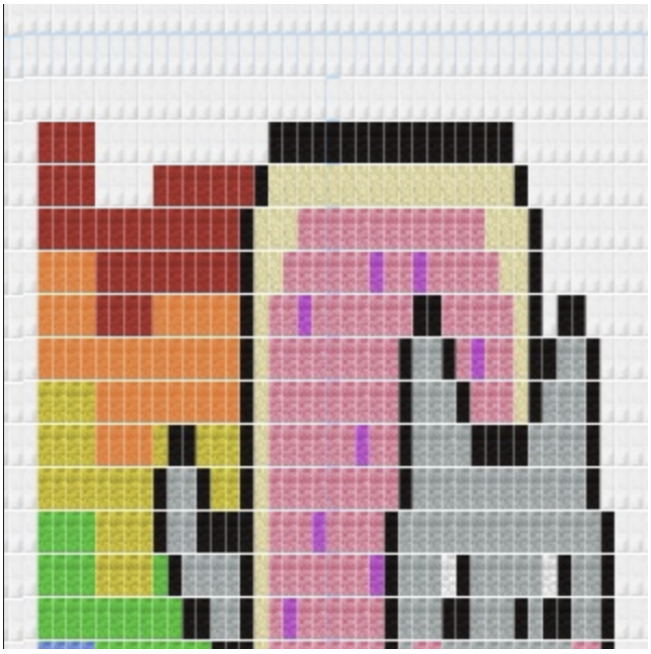


2)



3)





этот тест немного обрезан по высоте из-за неточного задания нового размера.

Результаты

Работа с GPU на разных конфигурациях ядер:

Тест:	Результат:

исходная 8*8 выход 2*2	kernel = <<<1, 32>>>, time = 0.041728 kernel = <<<1, 64>>>, time = 0.022656 kernel = <<<1, 128>>>, time = 0.030592 kernel = <<<1, 256>>>, time = 0.049536 kernel = <<<1, 512>>>, time = 0.088864 kernel = <<<1, 1024>>>, time = 0.168224 kernel = <<<2, 32>>>, time = 0.019520 kernel = <<<2, 64>>>, time = 0.029472 kernel = <<<2, 128>>>, time = 0.049632 kernel = <<<2, 256>>>, time = 0.088832 kernel = <<<2, 512>>>, time = 0.168320 kernel = <<<2, 1024>>>, time = 0.326624 kernel = <<<4, 32>>>, time = 0.029408 kernel = <<<4, 64>>>, time = 0.049184 kernel = <<<4, 128>>>, time = 0.089408 kernel = <<<4, 256>>>, time = 0.168288 kernel = <<<4, 512>>>, time = 0.327040 kernel = <<<4, 1024>>>, time = 0.643872 kernel = <<<8, 32>>>, time = 0.049280 kernel = <<<8, 64>>>, time = 0.089248 kernel = <<<8, 128>>>, time = 0.168640 kernel = <<<8, 256>>>, time = 0.326784 kernel = <<<8, 512>>>, time = 0.644512 kernel = <<<8, 1024>>>, time = 1.278944 kernel = <<<16, 32>>>, time = 0.089056 kernel = <<<16, 64>>>, time = 0.168608 kernel = <<<16, 128>>>, time = 0.326944 kernel = <<<16, 256>>>, time = 0.642368 kernel = <<<16, 512>>>, time = 1.277664 kernel = <<<16, 1024>>>, time = 2.540192 kernel = <<<32, 32>>>, time = 0.168192 kernel = <<<32, 64>>>, time = 0.326368 kernel = <<<32, 128>>>, time = 0.642432 kernel = <<<32, 256>>>, time = 1.278016 kernel = <<<32, 512>>>, time = 2.545984 kernel = <<<32, 1024>>>, time = 5.081184 kernel = <<<64, 32>>>, time = 0.328224 kernel = <<<64, 64>>>, time = 0.644960 kernel = <<<64, 128>>>, time = 1.275776 kernel = <<<64, 256>>>, time = 2.547392 kernel = <<<64, 512>>>, time = 5.077024 kernel = <<<64, 1024>>>, time = 10.152576 kernel = <<<128, 32>>>, time = 0.647008 kernel = <<<128, 64>>>, time = 1.276736 kernel = <<<128, 128>>>, time = 2.542880 kernel = <<<128, 256>>>, time = 5.089632 kernel = <<<128, 512>>>, time = 10.145728 kernel = <<<128, 1024>>>, time = 20.262079 kernel = <<<256, 32>>>, time = 1.278336 kernel = <<<256, 64>>>, time = 2.542656 kernel = <<<256, 128>>>, time = 5.086080 kernel = <<<256, 256>>>, time = 10.159488 kernel = <<<256, 512>>>, time = 20.297119 kernel = <<<256, 1024>>>, time = 40.597023 kernel = <<<512, 32>>>, time = 2.554496 kernel = <<<512, 64>>>, time = 5.088224 kernel = <<<512, 128>>>, time = 10.157600 kernel = <<<512, 256>>>, time = 20.312639 kernel = <<<512, 512>>>, time = 40.653313

	kernel = <<<512, 1024>>>, time = 81.080223 kernel = <<<1024, 32>>>, time = 5.093408 kernel = <<<1024, 64>>>, time = 10.140704 kernel = <<<1024, 128>>>, time = 20.341312 kernel = <<<1024, 256>>>, time = 40.563744 kernel = <<<1024, 512>>>, time = 81.280128 kernel = <<<1024, 1024>>>, time = 161.803772
исходная 1000*1000 выход 500*500	kernel = <<<1, 32>>>, time = 0.381216 kernel = <<<1, 64>>>, time = 0.357120 kernel = <<<1, 128>>>, time = 0.356000 kernel = <<<1, 256>>>, time = 0.360128 kernel = <<<1, 512>>>, time = 0.399808 kernel = <<<1, 1024>>>, time = 0.480448 kernel = <<<2, 32>>>, time = 0.358144 kernel = <<<2, 64>>>, time = 0.358240 kernel = <<<2, 128>>>, time = 0.366336 kernel = <<<2, 256>>>, time = 0.408416 kernel = <<<2, 512>>>, time = 0.469952 kernel = <<<2, 1024>>>, time = 0.644864 kernel = <<<4, 32>>>, time = 0.340224 kernel = <<<4, 64>>>, time = 0.369088 kernel = <<<4, 128>>>, time = 0.405344 kernel = <<<4, 256>>>, time = 0.490432 kernel = <<<4, 512>>>, time = 0.645440 kernel = <<<4, 1024>>>, time = 0.967072 kernel = <<<8, 32>>>, time = 0.382880 kernel = <<<8, 64>>>, time = 0.402592 kernel = <<<8, 128>>>, time = 0.501504 kernel = <<<8, 256>>>, time = 0.647680 kernel = <<<8, 512>>>, time = 0.976128 kernel = <<<8, 1024>>>, time = 1.614176 kernel = <<<16, 32>>>, time = 0.439584 kernel = <<<16, 64>>>, time = 0.516608 kernel = <<<16, 128>>>, time = 0.679872 kernel = <<<16, 256>>>, time = 0.983520 kernel = <<<16, 512>>>, time = 1.603072 kernel = <<<16, 1024>>>, time = 2.901728 kernel = <<<32, 32>>>, time = 0.558368 kernel = <<<32, 64>>>, time = 0.717984 kernel = <<<32, 128>>>, time = 1.028640 kernel = <<<32, 256>>>, time = 1.662112 kernel = <<<32, 512>>>, time = 2.934080 kernel = <<<32, 1024>>>, time = 5.475168 kernel = <<<64, 32>>>, time = 0.740448 kernel = <<<64, 64>>>, time = 1.061056 kernel = <<<64, 128>>>, time = 1.692672 kernel = <<<64, 256>>>, time = 2.962752 kernel = <<<64, 512>>>, time = 5.485792 kernel = <<<64, 1024>>>, time = 10.577248 kernel = <<<128, 32>>>, time = 1.134336 kernel = <<<128, 64>>>, time = 1.771456 kernel = <<<128, 128>>>, time = 3.033344 kernel = <<<128, 256>>>, time = 5.561920 kernel = <<<128, 512>>>, time = 10.663360 kernel = <<<128, 1024>>>, time = 20.746113 kernel = <<<256, 32>>>, time = 1.926144 kernel = <<<256, 64>>>, time = 3.186368 kernel = <<<256, 128>>>, time = 5.715040

	kernel = <<<256, 256>>>, time = 10.817056 kernel = <<<256, 512>>>, time = 20.885599 kernel = <<<256, 1024>>>, time = 41.289856 kernel = <<<512, 32>>>, time = 3.500224 kernel = <<<512, 64>>>, time = 6.019232 kernel = <<<512, 128>>>, time = 11.092448 kernel = <<<512, 256>>>, time = 21.208704 kernel = <<<512, 512>>>, time = 41.429153 kernel = <<<512, 1024>>>, time = 81.879555 kernel = <<<1024, 32>>>, time = 6.639104 kernel = <<<1024, 64>>>, time = 11.734016 kernel = <<<1024, 128>>>, time = 21.815968 kernel = <<<1024, 256>>>, time = 42.069279 kernel = <<<1024, 512>>>, time = 82.549248 kernel = <<<1024, 1024>>>, time = 163.412735
исходная 10000*10000 выход 500*500	kernel = <<<1, 32>>>, time = 134.404922 kernel = <<<1, 64>>>, time = 134.306946 kernel = <<<1, 128>>>, time = 133.988358 kernel = <<<1, 256>>>, time = 134.607712 kernel = <<<1, 512>>>, time = 134.218750 kernel = <<<1, 1024>>>, time = 135.063583 kernel = <<<2, 32>>>, time = 139.454178 kernel = <<<2, 64>>>, time = 139.909241 kernel = <<<2, 128>>>, time = 139.539993 kernel = <<<2, 256>>>, time = 139.562973 kernel = <<<2, 512>>>, time = 139.443298 kernel = <<<2, 1024>>>, time = 139.982651 kernel = <<<4, 32>>>, time = 140.803909 kernel = <<<4, 64>>>, time = 141.032196 kernel = <<<4, 128>>>, time = 141.095779 kernel = <<<4, 256>>>, time = 141.008194 kernel = <<<4, 512>>>, time = 140.961090 kernel = <<<4, 1024>>>, time = 141.076035 kernel = <<<8, 32>>>, time = 139.737091 kernel = <<<8, 64>>>, time = 140.031815 kernel = <<<8, 128>>>, time = 138.106659 kernel = <<<8, 256>>>, time = 139.782532 kernel = <<<8, 512>>>, time = 140.119461 kernel = <<<8, 1024>>>, time = 140.914978 kernel = <<<16, 32>>>, time = 139.960510 kernel = <<<16, 64>>>, time = 140.587646 kernel = <<<16, 128>>>, time = 139.420441 kernel = <<<16, 256>>>, time = 140.350525 kernel = <<<16, 512>>>, time = 140.901306 kernel = <<<16, 1024>>>, time = 141.955490 kernel = <<<32, 32>>>, time = 138.946564 kernel = <<<32, 64>>>, time = 138.450531 kernel = <<<32, 128>>>, time = 139.468796 kernel = <<<32, 256>>>, time = 139.913315 kernel = <<<32, 512>>>, time = 140.482407 kernel = <<<32, 1024>>>, time = 143.483200 kernel = <<<64, 32>>>, time = 139.100739

	kernel = <<<64, 64>>>, time = 138.744965 kernel = <<<64, 128>>>, time = 139.994202 kernel = <<<64, 256>>>, time = 139.865723 kernel = <<<64, 512>>>, time = 143.453094 kernel = <<<64, 1024>>>, time = 148.820007 kernel = <<<128, 32>>>, time = 138.174240 kernel = <<<128, 64>>>, time = 139.658524 kernel = <<<128, 128>>>, time = 140.049255 kernel = <<<128, 256>>>, time = 142.326752 kernel = <<<128, 512>>>, time = 148.526367 kernel = <<<128, 1024>>>, time = 158.450363 kernel = <<<256, 32>>>, time = 139.899963 kernel = <<<256, 64>>>, time = 140.192474 kernel = <<<256, 128>>>, time = 142.500092 kernel = <<<256, 256>>>, time = 148.530075 kernel = <<<256, 512>>>, time = 157.889023 kernel = <<<256, 1024>>>, time = 178.764542 kernel = <<<512, 32>>>, time = 140.686111 kernel = <<<512, 64>>>, time = 143.712891 kernel = <<<512, 128>>>, time = 148.135193 kernel = <<<512, 256>>>, time = 158.705765 kernel = <<<512, 512>>>, time = 178.577927 kernel = <<<512, 1024>>>, time = 219.327011 kernel = <<<1024, 32>>>, time = 144.262466 kernel = <<<1024, 64>>>, time = 149.408096 kernel = <<<1024, 128>>>, time = 158.910492 kernel = <<<1024, 256>>>, time = 179.278534 kernel = <<<1024, 512>>>, time = 220.016006 kernel = <<<1024, 1024>>>, time = 301.258057

Работа на CPU:

Тест:	Результат:
исходная 8*8 выход 2*2	0.003
исходная 1000*1000 выход 500*500	17.38
исходная 10000*10000 выход 500*500	461.38

Код программы для CPU:

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

```
int main() {
    std::string inputFile;
    std::string outputFile;
    int wn, hn, w, h;

    std::cin >> inputFile >> outputFile;
    scanf("%d %d", &wn, &hn);

    FILE* fp = fopen(inputFile.c_str(), "rb");

    fread(&w, sizeof(int), 1, fp);
    fread(&h, sizeof(int), 1, fp);
    char** data = (char**)malloc(sizeof(char*) * w * h); //по 2 измерению размерность 4
    for (int i = 0; i < w * h; i++) {
        data[i] = (char*)malloc(sizeof(char) * 4);
    }
    for (int i = 0; i < w * h; i++) {
        fread(data[i], sizeof(char), 4, fp);
    }
    fclose(fp);


    int diff_w = w / wn;
    int diff_h = h / hn;
    int k = diff_w * diff_h;
    int a[3];
    a[0] = 0;
    a[1] = 0;
    a[2] = 0;
    char **data_out = (char **)malloc(sizeof(char*) * wn * hn);
    for (int i = 0; i < wn * hn; i++) {
        data_out[i] = (char*)malloc(sizeof(char) * 4);
    }

    clock_t begin = clock();
    for(int y = 0; y < h; y = y + diff_h) {
        for(int x = 0; x < w; x = x + diff_w) {
            a[0] = 0;
            a[1] = 0;
            a[2] = 0;
            for(int i = x; i < x + diff_w; i++) {
                for(int j = y; j < y + diff_h; j++) {
                    a[0] = data[y*w + x][0];
                    a[1] = data[y*w + x][1];
                    a[2] = data[y*w + x][2];
                }
            }
            a[0] /= k;
            a[1] /= k;
            a[2] /= k;
            data_out[y * wn / diff_h + x / diff_w][0] = a[0];
            data_out[y * wn / diff_h + x / diff_w][1] = a[1];
            data_out[y * wn / diff_h + x / diff_w][2] = a[2];
        }
    }

    clock_t end = clock();
    double time_spent = (double)(end - begin) * 1000 / CLOCKS_PER_SEC;
```

```
    printf("%lf\n", time_spent);

    fp = fopen(outputFile.c_str(), "wb");
    fwrite(&wn, sizeof(int), 1, fp);
    fwrite(&hn, sizeof(int), 1, fp);
    for (int i = 0; i < wn * hn; i++) {
        fwrite(data_out, sizeof(char), 4, fp);
    }
    fclose(fp);

    free(data);
    free(data_out);
    return 0;
}
```

Выводы

Анализируя результат работы на разных конфигурациях ядра GPU и алгоритма для CPU, можно сказать, что на маленьких тестах(8*8) время работы на большом количестве потоков нецелесообразно и очень медленно. Наилучшее время на GPU для теста 8*8 совпадает по порядку с временем на CPU. Если взять уже 1000 на 1000 и ужимать в два раза до 500 на 500, то лучший результат на GPU превосходит в 100 раз CPU. На самом большом тесте 10.000 на 10.000, время на GPU лучше только в 4 раза. Наверно это связано с тем, что для оптимизации на уровне предыдущего теста необходимо иметь сетку с большим числом потоков, потому что тест 10.000 на 10.000 очень тяжелый. Вцелом, вычисления с использованием графических адаптеров показывают максимальную эффективность в задачах, не требующих интенсивного обращения к памяти, поэтому на больших тестах эффективность будет не очень высока. Считается, если задача требует большого количества памяти (несколько гигабайт), то, скорее всего, на данном этапе развития технологии CUDA её вообще не целесообразно решать при помощи GPU.

Если говорить о тематике лабораторной, то *текстурная память* — память, располагающаяся в микросхемах DRAM, кэшируется. Используется для хранения больших массивов данных. Выделяется целиком на грид.

Текстурная память является особым образом выделенной областью глобальной памяти. Обращение к текстурной памяти производится с использованием кэша. Текстурная память также позволяет использовать адресацию с плавающей точкой (при этом применяется линейная или билинейная интерполяция). Соответственно, существуют дополнительные стадии конвейера (преобразование адресов, фильтрация, преобразование данных), которые снижают скорость первого обращения. Для использования текстурной памяти необходимо задать объявление текстуры как глобальной переменной, а потом связать её с требуемой областью глобальной памяти.

Кроме самого объявления текстуры, требуется задать несколько параметров.

• **Нормализация адресов.** При нормализации адресов происходит перевод отрезка [*K*,*N*] в отрезок [0, 1].

• **Преобразование адресов.** Если координата не попадает в заданный диапазон (отрезок [*K*,*N*] или [0,1]), то видеокарта на аппаратном уровне производит преобразование. Существует два типа преобразования:

– Clamp — возвращается значение на ближайшей границе диапазона;
– Wrap — возвращается значение внутри диапазона, по сути, происходит взятие остатка от деления адреса на длину диапазона.

• **Фильтрация.** Когда обращение происходит по адресу типа float, а данные были заданы для целочисленных адресов, то необходимо определить, какое значение будет возвращено из текстуры. Существует два способа:
– Point — берется ближайшее значение из массива;
– Linear — расчет значения проводится на основе линейной (билинейной) интерполяции.

• **Преобразование данных.** Графический процессор имеет возможность преобразовывать считываемые данные, например, массив char4 может быть преобразован в float4. В CUDA существует два типа текстур — линейная и cudaArray.

После объявления текстуры и задания всех её параметров необходимо «привязать» данные, загруженные в глобальную память или cudaArray, к объявлению текстуры с помощью функций cudaBindTexture и cudaBindTextureToArray соответственно.