

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторная работа №2
по курсу «Программирование графических процессоров»

Обработка изображений на GPU. Фильтры

Выполнил: *И.Д. Фомин*

Группа: *M80-409Б-22*

Преподаватели: *А.Ю. Морозов,
Е.Е. Заяц*

Москва, 2025

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:

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

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 полограничном 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 определяться следующим образом:

ip := max(min(i, h), 1)

jp := max(min(j, w), 1)

Вариант 7. Выделение контуров. Метод Собеля.

Входные данные. На первой строке задается путь к исходному изображению, на второй, путь к конечному изображению. $w \cdot h \leq 10^8$.

Пример:

Входной файл	hex: in.data	hex: out.data
in.data out.data	03000000 03000000 01020300 04050600 07080900 09080700 06050400 03020100 00000000 14141400 00000000	03000000 03000000 14141400 0C0C0C00 11111100 13131300 15151500 18181800 39393900 14141400 3F3F3F00
in.data out.data	03000000 03000000 00000000 00000000 00000000 00000000 80808000 00000000 00000000 00000000 00000000	03000000 03000000 B5B5B500 FFFFFF00 B5B5B500 FFFFFFFFFF00 00000000 FFFFFF00 B5B5B500 FFFFFF00 B5B5B500

Программное и аппаратное обеспечение

OS	Ubuntu 22.04
IDE	VSCode 1.102.3
RAM	Общий объем: 32 GB Конфигурация: 16 GB + 16 GB Тип памяти: DDR5 Тактовая частота: 4800 МГц Форм-фактор: SODIMM Производители: Samsung
SSD	Тип: NVMe SSD Производитель: The Western Digital Модель: WD PC SN735 SDBPNHH-1T00 Объем: 1024 GB Интерфейс: NVMe Форм-фактор: M.2 2280
Compiler	<pre>ivan@DESKTOP-JOOUDAE:~/mai/course4 /pgp/lab1\$ nvcc --version nvcc: NVIDIA (R) Cuda compiler driver Copyright (c) 2005-2023 NVIDIA Corporation Built on Fri_Jan_6_16:45:21_PST_2023 Cuda compilation tools, release 12.0, V12.0.140 Build cuda_12.0.r12.0/compiler.32267302_0</pre> <pre>ivan@DESKTOP-JOOUDAE:~/mai/course4 /pgp/lab1\$ gcc --version gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0</pre> <pre>ivan@DESKTOP-JOOUDAE:~/mai/course4 /pgp/lab1\$ g++ --version g++ (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0</pre>

GPU:

Compute capability	8.6
Name	NVIDIA GeForce RTX 3080 Mobile

Total Global Memory	17179869184
Shared memory per block	102400
Registers per block	65536
Warp size	32
Max threads per block	(1024, 1024, 64)
Max block	(2147483647, 65535, 65535)
Total constant memory	65536
Multiprocessors count	38

CPU:

brand	AMD Ryzen 9 6900HS with Radeon Graphics
arch	AMD64
bits	64
hz_advertised	3.2940 GHz
hz_actual	2.4980 GHz
cores_physical	8
cores_logical	16

Метод решения

Считываем файл, который мы получили на вход

Выделяем память для работы с изображением

Копируем данные с хоста на девайс

Описываем текстуру и ресурс и на их основе создаем объект текстуры

Затем запускаем наше ядро и для каждого пикселя и его соседей применяем формулу светимости и получаем цвет этого пикселя в greyscale

```
uchar4 neighborPixel = tex2D<uchar4>(texObj, (float)(x + j) + 0.5f, (float)(y + i) + 0.5f);
```

- получаем конкретный пиксель (RGBA) из текстуры, +0.5f нужен для того, чтобы мы переместились в центр пикселя и взяли цвет оттуда

В результате применения двух фильтров получаются два значения – два градиента. Далее подсчитывается величина – корень из суммы квадратов градиентов. Полученное значение и есть итоговое значение в greyscale текущего пикселя.

$$|\nabla f| = \sqrt{G_x^2 + G_y^2}$$

Результат вычислений копируется на хост и выводится в нужном формате в выходной файл.

Описание программы

Считываем данные по путям из стандартного ввода, выделяем соответствующую память под все данные изображения - ширина * высота * 4 (RGBA каждого пикселя)

При cudaMemcpy2DToArray указываем что мы хотим копировать массив w*h*4, с шириной w*4 и высотой h, сдвиг каждой строки - w (т.к. данные у нас стоят плотно и GPU не делает выравнивание) на девайс

создаем текстурный объект:

```
cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.addressMode[0] = cudaAddressModeClamp;
texDesc.addressMode[1] = cudaAddressModeClamp;
texDesc.filterMode = cudaFilterModePoint;
texDesc.readMode = cudaReadModeElementType;
texDesc.normalizedCoords = 0;
```

addressModeClamp - выход за границы будет зажат к краю.

filterModePoint - выборка без интерполяции - нам нужна точная выборка по пикселям.

readMode = ElementType - возвращаемый тип совпадает с типом хранимых элементов, здесь uchar4.

normalizedCoords = 0 - координаты в пикселях (не 0..1).

То, что происходит в ядре, описано пунктом ранее, но можно ещё добавить пару технических моментов

```
__global__ void sobelKernel(cudaTextureObject_t texObj, uchar4* d_outData, int width, int height) {
    const int thread_x_idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int thread_y_idx = blockIdx.y * blockDim.y + threadIdx.y;
    const int total_threads_x = gridDim.x * blockDim.x;
    const int total_threads_y = gridDim.y * blockDim.y;
```

Каждый поток индексируется двумя индексами, на основе которых он будет обрабатывать определенные пиксели изображения - это нам надо чтобы каждый поток обладал каким-то уникальным идентификатором и обрабатывал свой пиксель(и).

Далее описываем фильтры - это два градиента (горизонтальный и вертикальный) - эти градиенты помогут посчитать корень суммы их квадратов, который будет численно равен яркости этого пикселя

```
const float Gx[3][3] = {
    {-1.0f, 0.0f, 1.0f},
    {-2.0f, 0.0f, 2.0f},
    {-1.0f, 0.0f, 1.0f}
};

const float Gy[3][3] = {
    {-1.0f, -2.0f, -1.0f},
    { 0.0f, 0.0f, 0.0f},
    { 1.0f, 2.0f, 1.0f}
};
```

Получаем яркость с помощью двух градиентов (sumX и sumY), и если наш пиксель слишком яркий - ограничиваем его на 255 т.к. это самое яркое что мы можем показать

```
float magnitude = sqrtf(sumX * sumX + sumY * sumY);
```

```

        int val = (int)(magnitude + 0.5f);
        if (val > 255) val = 255;
        unsigned char finalVal = (unsigned char)val;
        int idx = y * width + x;
        d_outData[idx] = make_uchar4(finalVal, finalVal, finalVal, 0);
    }
}

```

Произвели вычисления, провели синхронизацию, проверили, есть ли ошибки - теперь можно считать то, что мы посчитали

```

sobelKernel<<<numBlocks, threadsPerBlock>>>(texObj, d_outData, w, h);
CSC(cudaGetLastError());
CSC(cudaDeviceSynchronize());

unsigned char* h_outData = (unsigned char*)malloc(dataSize);
CSC(cudaMemcpy(h_outData, d_outData, dataSize, cudaMemcpyDeviceToHost));

```

Результаты

Image: 1080x1080



```

<<<dim3(1,1), dim3(1,1)>>> 509.883мс
<<<dim3(1,2), dim3(1,2)>>> 79.297мс
<<<dim3(1,4), dim3(1,4)>>> 20.361мс
<<<dim3(1,8), dim3(1,8)>>> 5.122мс
<<<dim3(1,16), dim3(1,16)>>> 1.642мс
<<<dim3(1,32), dim3(1,32)>>> 0.701мс
<<<dim3(2,1), dim3(2,1)>>> 91.948мс

```

```
<<<dim3(2,2), dim3(2,2)>>> 35.194MC
<<<dim3(2,4), dim3(2,4)>>> 9.056MC
<<<dim3(2,8), dim3(2,8)>>> 2.294MC
<<<dim3(2,16), dim3(2,16)>>> 0.774MC
<<<dim3(2,32), dim3(2,32)>>> 0.310MC
<<<dim3(4,1), dim3(4,1)>>> 23.148MC
<<<dim3(4,2), dim3(4,2)>>> 9.099MC
<<<dim3(4,4), dim3(4,4)>>> 2.401MC
<<<dim3(4,8), dim3(4,8)>>> 0.663MC
<<<dim3(4,16), dim3(4,16)>>> 0.238MC
<<<dim3(4,32), dim3(4,32)>>> 0.118MC
<<<dim3(8,1), dim3(8,1)>>> 5.994MC
<<<dim3(8,2), dim3(8,2)>>> 2.330MC
<<<dim3(8,4), dim3(8,4)>>> 0.675MC
<<<dim3(8,8), dim3(8,8)>>> 0.172MC
<<<dim3(8,16), dim3(8,16)>>> 0.090MC
<<<dim3(8,32), dim3(8,32)>>> 0.052MC
<<<dim3(16,1), dim3(16,1)>>> 1.968MC
<<<dim3(16,2), dim3(16,2)>>> 0.762MC
<<<dim3(16,4), dim3(16,4)>>> 0.204MC
<<<dim3(16,8), dim3(16,8)>>> 0.105MC
<<<dim3(16,16), dim3(16,16)>>> 0.091MC
<<<dim3(16,32), dim3(16,32)>>> 0.076MC
<<<dim3(32,1), dim3(32,1)>>> 0.930MC
<<<dim3(32,2), dim3(32,2)>>> 0.357MC
<<<dim3(32,4), dim3(32,4)>>> 0.127MC
<<<dim3(32,8), dim3(32,8)>>> 0.095MC
<<<dim3(32,16), dim3(32,16)>>> 0.072MC
<<<dim3(32,32), dim3(32,32)>>> 0.068MC
```

CPU time 65.235 ms

Image: 2560x1920



```
<<<dim3(1,1), dim3(1,1)>>> 1170.369MC
<<<dim3(1,2), dim3(1,2)>>> 324.107MC
<<<dim3(1,4), dim3(1,4)>>> 82.340MC
<<<dim3(1,8), dim3(1,8)>>> 20.589MC
<<<dim3(1,16), dim3(1,16)>>> 5.600MC
<<<dim3(1,32), dim3(1,32)>>> 1.606MC
<<<dim3(2,1), dim3(2,1)>>> 408.575MC
<<<dim3(2,2), dim3(2,2)>>> 145.914MC
<<<dim3(2,4), dim3(2,4)>>> 37.235MC
<<<dim3(2,8), dim3(2,8)>>> 9.497MC
<<<dim3(2,16), dim3(2,16)>>> 2.649MC
<<<dim3(2,32), dim3(2,32)>>> 0.717MC
<<<dim3(4,1), dim3(4,1)>>> 107.477MC
<<<dim3(4,2), dim3(4,2)>>> 37.588MC
<<<dim3(4,4), dim3(4,4)>>> 9.874MC
<<<dim3(4,8), dim3(4,8)>>> 2.602MC
<<<dim3(4,16), dim3(4,16)>>> 0.706MC
<<<dim3(4,32), dim3(4,32)>>> 0.301MC
<<<dim3(8,1), dim3(8,1)>>> 26.818MC
<<<dim3(8,2), dim3(8,2)>>> 9.570MC
<<<dim3(8,4), dim3(8,4)>>> 2.579MC
<<<dim3(8,8), dim3(8,8)>>> 0.755MC
<<<dim3(8,16), dim3(8,16)>>> 0.263MC
<<<dim3(8,32), dim3(8,32)>>> 0.215MC
<<<dim3(16,1), dim3(16,1)>>> 6.857MC
<<<dim3(16,2), dim3(16,2)>>> 2.487MC
<<<dim3(16,4), dim3(16,4)>>> 0.685MC
<<<dim3(16,8), dim3(16,8)>>> 0.267MC
<<<dim3(16,16), dim3(16,16)>>> 0.176MC
<<<dim3(16,32), dim3(16,32)>>> 0.223MC
<<<dim3(32,1), dim3(32,1)>>> 2.191MC
```

```
<<<dim3(32,2), dim3(32,2)>>> 0.809MC  
<<<dim3(32,4), dim3(32,4)>>> 0.237MC  
<<<dim3(32,8), dim3(32,8)>>> 0.259MC  
<<<dim3(32,16), dim3(32,16)>>> 0.203MC  
<<<dim3(32,32), dim3(32,32)>>> 0.166MC
```

CPU time 87.2667 ms

Image: 1280x960



```
<<<dim3(1,1), dim3(1,1)>>> 339.168MC  
<<<dim3(1,2), dim3(1,2)>>> 81.234MC  
<<<dim3(1,4), dim3(1,4)>>> 20.744MC  
<<<dim3(1,8), dim3(1,8)>>> 5.211MC  
<<<dim3(1,16), dim3(1,16)>>> 1.423MC  
<<<dim3(1,32), dim3(1,32)>>> 0.399MC  
<<<dim3(2,1), dim3(2,1)>>> 95.306MC  
<<<dim3(2,2), dim3(2,2)>>> 36.578MC  
<<<dim3(2,4), dim3(2,4)>>> 9.289MC  
<<<dim3(2,8), dim3(2,8)>>> 2.457MC  
<<<dim3(2,16), dim3(2,16)>>> 0.696MC  
<<<dim3(2,32), dim3(2,32)>>> 0.193MC  
<<<dim3(4,1), dim3(4,1)>>> 24.214MC  
<<<dim3(4,2), dim3(4,2)>>> 9.404MC  
<<<dim3(4,4), dim3(4,4)>>> 2.463MC  
<<<dim3(4,8), dim3(4,8)>>> 0.641MC  
<<<dim3(4,16), dim3(4,16)>>> 0.180MC  
<<<dim3(4,32), dim3(4,32)>>> 0.113MC
```

```
<<<dim3(8,1), dim3(8,1)>>> 6.163мс
<<<dim3(8,2), dim3(8,2)>>> 2.391мс
<<<dim3(8,4), dim3(8,4)>>> 0.650мс
<<<dim3(8,8), dim3(8,8)>>> 0.220мс
<<<dim3(8,16), dim3(8,16)>>> 0.102мс
<<<dim3(8,32), dim3(8,32)>>> 0.073мс
<<<dim3(16,1), dim3(16,1)>>> 1.717мс
<<<dim3(16,2), dim3(16,2)>>> 0.637мс
<<<dim3(16,4), dim3(16,4)>>> 0.199мс
<<<dim3(16,8), dim3(16,8)>>> 0.085мс
<<<dim3(16,16), dim3(16,16)>>> 0.127мс
<<<dim3(16,32), dim3(16,32)>>> 0.112мс
<<<dim3(32,1), dim3(32,1)>>> 0.778мс
<<<dim3(32,2), dim3(32,2)>>> 0.278мс
<<<dim3(32,4), dim3(32,4)>>> 0.185мс
<<<dim3(32,8), dim3(32,8)>>> 0.069мс
<<<dim3(32,16), dim3(32,16)>>> 0.067мс
<<<dim3(32,32), dim3(32,32)>>> 0.061мс
```

CPU time 71.254 ms

Выводы:

Метод Собеля является нужен для выделения границ на изображениях.

В результате применения фильтра Собеля пиксели, где наблюдаются резкие перепады яркости, получают высокие значения градиента - это соответствует границам объектов на изображении, а области с плавным изменением яркости дают малые значения - они становятся фоном.

Метод Собеля может быть применён в системах компьютерного зрения для распознавания объектов и выделения контуров, в робототехнике для навигации и обнаружения препятствий, в медицинской визуализации для определения границ органов и патологий, в промышленности для контроля формы и поиска дефектов изделий, в системах видеонаблюдения для обнаружения движения, а также в обработке спутниковых и аэрофотоснимков для выделения дорог, зданий и природных объектов.

По замерам производительности видно, что GPU работает сильно быстрее CPU. Чем больше потоков на GPU мы задействуем, тем больше эта разница становится. Наименее эффективны вычисления с единственным потоком в блоке и единственным блоком в гриде.