МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет «Информационные технологии и прикладная математика»

Кафедра «Вычислительная математика и программирование»

**Лабораторная работа №2**

**по курсу «Программирование графических процессоров»**

**Изучение технологии CUDA**

Выполнил: И.И. Тишин

Группа: 8О-406Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2019

**Условие**

Научиться использовать GPU для обработки изображений. Использование текстурной памяти.

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 < 216 и h < 216.

Вариант 5: Метод Робертса.  
Необходимо реализовать выделение контуров. Исходное

изображение представляет собой “экранный буфер”, на выходе должен быть

контур исходного изображения.

Входные данные. На первой строке задается путь к исходному изображению,

на второй, путь к конечному изображению. w\*h ≤ 10 .

**Программное и аппаратное обеспечение**

**Программное и аппаратное обеспечение**

GPU:

Device 0: "GeForce GTX 1060 with Max-Q Design"

CUDA Driver Version / Runtime Version 10.1 / 10.1

CUDA Capability Major/Minor version number: 6.1

Total amount of global memory: 6078 MBytes (6373572608 bytes)

(10) Multiprocessors, (128) CUDA Cores/MP: 1280 CUDA Cores

GPU Max Clock rate: 1480 MHz (1.48 GHz)

Memory Clock rate: 4004 Mhz

Memory Bus Width: 192-bit

L2 Cache Size: 1572864 bytes

Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)

Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers

Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers

Total amount of constant memory: 65536 bytes

Total amount of shared memory per block: 49152 bytes

Total number of registers available per block: 65536

Warp size: 32

Maximum number of threads per multiprocessor: 2048

Maximum number of threads per block: 1024

Max dimension size of a thread block (x,y,z): (1024, 1024, 64)

Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

Maximum memory pitch: 2147483647 bytes

Texture alignment: 512 bytes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.1, NumDevs = 1

CPU:

Архитектура: x86\_64

CPU op-mode(s): 32-bit, 64-bit

CPU(s): 12

Потоков на ядро: 2

Ядер на сокет: 6

Имя модели: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz

CPU max MHz: 4100,0000

CPU min MHz: 800,0000

BogoMIPS: 4416.00

Виртуализация: VT-x

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 9216K

Ram:

32 GiB, 2666 MHz

HDD:

500 GB

Программное обеспечение:

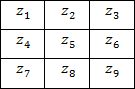
Ubuntu 18.04.2 LTS

g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0

nvcc: release 10.1, V10.1.105

**Метод решения**

Пусть область 3х3, показанная на рисунке ниже, представляет собой значения яркости в окрестности некоторого элемента изображения.



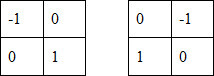
Один из простейших способов нахождения первых частных производных в точке image состоит в применении следующего перекрестного градиентного оператора Робертса**:**

image

и

image

Эти производные могут быть реализованы путем обработки всего изображения с помощью оператора, описываемого масками на рисунке 3, используя процедуру фильтрации, описанную ранее.



**Описание программы**

Программа состоит из 2-х функций: main и kernel.  
В main считываются данные со стандартного ввода, заполняется массив данными об исходной картинке. Далее выделяется память под текстурный массив и копируются в него данные о картинке, настраивается конфигурация отображения данных. Далее вызывается функция kernel c количеством блоков 32x32 и количеством тредов в блоке 32х32.

В kernel производится вычисление пикселей новой картинки. Каждый тред вычисляет производную в точке.

\_\_global\_\_ void kernel(uchar4 \*dst, int w, int h) {

int idx = threadIdx.x + blockDim.x \* blockIdx.x;

int idy = threadIdx.y + blockDim.y \* blockIdx.y;

int offsetx = blockDim.x \* gridDim.x;

int offsety = blockDim.y \* gridDim.y;

int i, j;

uchar4 p,p1;

double g12,g21,g22,g11;

for(i = idx; i < w; i += offsetx)

for(j = idy; j < h; j += offsety){

p1 = tex2D(tex, i, j);

g11 = RGBToWB(~p1.x,~p1.y,~p1.z);

if (i + 1 == w) {

g12 = g11;

}

else {

p = tex2D(tex, i+1, j);

g12= RGBToWB(~p.x,~p.y,~p.z);

}

if (j + 1 == h) {

g21 = g11;

}

else {

p = tex2D(tex, i, j+1);

g21 = RGBToWB(~p.x,~p.y,~p.z);

}

if (j + 1 == h || i + 1 == w) {

if(j + 1 == h)

g22 = g12;

else

g22 = g21;

}

else {

p = tex2D(tex, i+1, j+1);

g22 = RGBToWB(~p.x,~p.y,~p.z);

}

g11 -= g22;

g21 -= g12;

g11 = f(g11\*g11 + g21 \* g21);

g11=chek(g11);

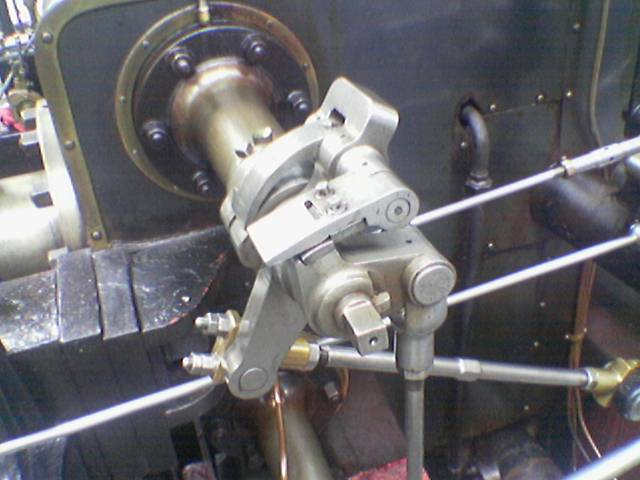
dst[j \* w + i] = make\_uchar4(g11,g11,g11, p1.w);

}

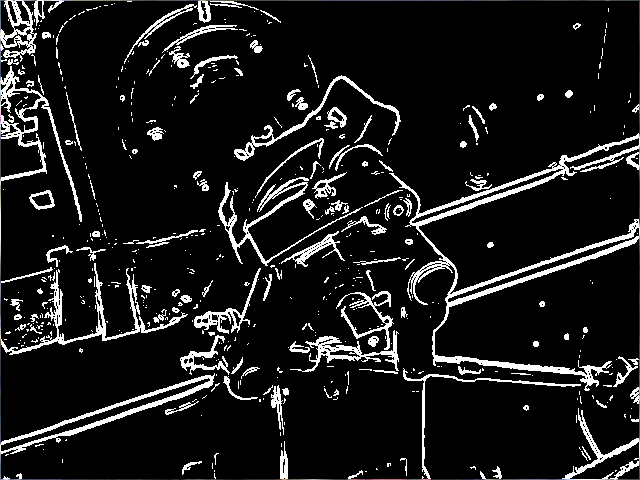
}

}

**Результаты**









|  |  |  |
| --- | --- | --- |
| Исходный размер | CPU | Время выполнения GPU |
| 100х100 | 2.549ms | 47.264us |
| 1000x1000 | 43.937ms | 853.47us |
| 10000x10000 | 4.283185s | 79.188675ms |

**Выводы**

Реализация масок размерами 2х2 не очень удобна, т.к. у них нет четко выраженного центрального элемента, что существенно отражается на результате выполнения фильтрации. Но этот «минус» порождает очень полезное свойство данного алгоритма – высокую скорость обработки изображения