Семинар 12

PyOpenCL - это библиотека Python для программирования на OpenCL, которая позволяет использовать GPU для параллельных вычислений. Вот несколько примеров фракталов, которые можно написать с помощью PyOpenCL:

* Множество Мандельброта
* Множество Жюлиа
* Множество Бернса-Шиппена
* Множество Ньютона
* Множество Лямберта
* Множество Серпинского
* Множество Кантора

Для создания фрактала Жюлиа, Ньютона и Серпинского можно использовать библиотеку PyOpenCL1. PyOpenCL предоставляет легкий доступ к API параллельных вычислений OpenCL на Python.

Фрактал Жюлиа - это графическое представление корней уравнения в комплексной плоскости. Фрактал Жюлиа строится следующим образом:

* выбирается уравнение с комплексными коэффициентами.
* выбирается область на комплексной плоскости.
* каждая точка в этой области рассматривается как начальное приближение для решения уравнения.
* для каждой точки вычисляется корень уравнения, к которому она сходится.
* каждому корню сопоставляется свой цвет.
* точки окрашиваются в цвет соответствующего корня.

Треугольник Серпинского - это один из самых простых и известных фракталов. Он состоит из треугольника, который делится на три равных треугольника, каждый из которых затем делится на три равных треугольника и так далее до бесконечности.

Фрактал Ньютона - это графическое представление корней уравнения в комплексной плоскости. Фрактал Ньютона строится следующим образом:

* выбирается уравнение с комплексными коэффициентами.
* выбирается область на комплексной плоскости.
* каждая точка в этой области рассматривается как начальное приближение для решения уравнения.
* для каждой точки вычисляется корень уравнения, к которому она сходится.
* каждому корню сопоставляется свой цвет.
* точки окрашиваются в цвет соответствующего корня.

1. Программа для фрактала Жюлиа:

# Импортируем необходимые библиотеки

import pyopencl as cl

import numpy as np

import matplotlib.pyplot as plt

# Задаем параметры фрактала

width = 800 # Ширина изображения в пикселях

height = 600 # Высота изображения в пикселях

max\_iter = 100 # Максимальное число итераций для каждой точки

x\_min = -2.0 # Минимальное значение x на комплексной плоскости

x\_max = 2.0 # Максимальное значение x на комплексной плоскости

y\_min = -1.5 # Минимальное значение y на комплексной плоскости

y\_max = 1.5 # Максимальное значение y на комплексной плоскости

c\_re = -0.8 # Действительная часть константы c

c\_im = 0.156 # Мнимая часть константы c

# Создаем массив для хранения результатов

output = np.empty((height, width), dtype=np.uint32)

# Создаем контекст и очередь OpenCL

ctx = cl.create\_some\_context()

queue = cl.CommandQueue(ctx)

# Создаем буферы для передачи данных между хостом и устройством

output\_buf = cl.Buffer(ctx, cl.mem\_flags.WRITE\_ONLY, output.nbytes)

prg = cl.Program(ctx, """

\_\_kernel void julia(\_\_global uint \*output,

const uint width,

const uint height,

const uint max\_iter,

const float x\_min,

const float x\_max,

const float y\_min,

const float y\_max,

const float c\_re,

const float c\_im)

{

// Получаем индексы пикселя в двумерном массиве

int i = get\_global\_id(0);

int j = get\_global\_id(1);

// Проверяем, что индексы не выходят за границы массива

if (i >= width || j >= height) return;

// Вычисляем координаты комплексного числа z, соответствующего пикселю

float z\_re = x\_min + i \* (x\_max - x\_min) / width;

float z\_im = y\_max - j \* (y\_max - y\_min) / height;

// Инициализируем счетчик итераций нулем

uint iter = 0;

// Повторяем итерации до тех пор, пока модуль z не превысит 2 или не достигнем максимального числа итераций

while (z\_re \* z\_re + z\_im \* z\_im < 4.0f && iter < max\_iter)

{

// Выполняем итерацию z = z^2 + c

float tmp\_re = z\_re \* z\_re - z\_im \* z\_im + c\_re;

float tmp\_im = 2.0f \* z\_re \* z\_im + c\_im;

z\_re = tmp\_re;

z\_im = tmp\_im;

// Увеличиваем счетчик итераций на единицу

iter++;

}

// Записываем результат в выходной массив

output[j \* width + i] = iter;

}

""").build()

# Запускаем ядро на устройстве с заданными параметрами

prg.julia(queue, output.shape, None, output\_buf,

np.uint32(width), np.uint32(height), np.uint32(max\_iter),

np.float32(x\_min), np.float32(x\_max), np.float32(y\_min), np.float32(y\_max),

np.float32(c\_re), np.float32(c\_im))

# Копируем результаты с устройства на хост

cl.enqueue\_copy(queue, output, output\_buf)

# Визуализируем результаты с помощью matplotlib

plt.imshow(output, cmap='inferno')

1. Программа для фрактала Серпинского

import pyopencl as cl

import numpy as np

import matplotlib.pyplot as plt

ctx = cl.create\_some\_context()

queue = cl.CommandQueue(ctx)

sierpinski\_fractal\_kernel = """

\_\_kernel void sierpinski\_fractal\_kernel(\_\_global int \*fractal,

const int width,

const int height,

const int maxiter,

const float xmin,

const float ymin,

const float dx,

const float dy)

{

int gid\_x = get\_global\_id(0);

int gid\_y = get\_global\_id(1);

float x = xmin + gid\_x \* dx;

float y = ymin + gid\_y \* dy;

float2 z = (float2)(x, y);

float2 c = (float2)(x, y);

int i;

for (i = 0; i < maxiter; i++)

{

z = (float2)(z.x \* z.x - z.y \* z.y, 2 \* z.x \* z.y) + c;

if (z.x \* z.x + z.y \* z.y > 4.f)

break;

}

fractal[gid\_y \* width + gid\_x] = i;

}

"""

prg = cl.Program(ctx, sierpinski\_fractal\_kernel).build()

def sierpinski\_fractal(xmin, xmax, ymin, ymax, width, height, maxiter):

x = np.linspace(xmin, xmax, width).astype(np.float32)

y = np.linspace(ymin, ymax, height).astype(np.float32)

c = x + y[:, None] \* 1j

fractal = np.zeros(c.shape, dtype=np.int32)

dx = (xmax - xmin) / width

dy = (ymax - ymin) / height

fractal\_buf = cl.Buffer(ctx,

cl.mem\_flags.WRITE\_ONLY,

fractal.nbytes)

prg.sierpinski\_fractal\_kernel(queue,

fractal.shape,

None,

fractal\_buf,

np.int32(width),

np.int32(height),

np.int32(maxiter),

np.float32(xmin),

np.float32(ymin),

np.float32(dx),

np.float32(dy))

cl.enqueue\_copy(queue, fractal, fractal\_buf)

return fractal

xmin, xmax, ymin, ymax = -2.0, 2.0, -2.0, 2.0

width, height = 1024, 1024

maxiter = 100

fractal = sierpinski\_fractal(xmin, xmax, ymin, ymax,

width=width,

height=height,

maxiter=maxiter)

plt.imshow(fractal.T[::-1], cmap='gray', extent=(xmin,xmax,ymin,ymax))

plt.show()

Варианты для самостоятельной разработки

Реализовать с использованием функций ядра в PyOpenCL фракталы:

1 вариант -

* Множество Бернса-Шиппена
* Множество Ньютона

2 вариант -

* Множество Лямберта
* Множество Кантора

Варианты соответствуют номеру в журнале – нечетному либо четному.