UNIVERSITY OF CAMBRIDGE

The Firtree Framework

A Dynamic Compilation System for Multi-Core Aware Image Processing

Dr Rich Wareham

Department of Engineering
University of Cambridge

Rutherford Appleton Laboratories, November 2009

Part I: Introduction

Outline

About Me

- Postdoc in the Signal Processing and Communications Laboratory[1], Department of Engineering, University of Cambridge
- Member of the Many Core Computing Group[2] in Cambridge
- Have a weakness for writing DSLs

[1] http://www-sigproc.eng.cam.ac.uk/
[2] http://www.many-core.group.cam.ac.uk/

The Firtree Project

- Developed by one man (me!)
- Personal project developed after I wrote a Sobel filter for the 30th time
- Hosted at `http://firtree.org/`
- Aim is to allow me to write once, use many times
- An exercise in an intrinsically multi-core capable system

Requirements

- Implemented image processing algorithms in C
- Want to make use of multi-core hardware
- Don't want to be fussed with mechanism
- Moderate programming skills
- Have some knowledge of nodal image processing pipelines

Motivation

- Want to spend time developing *algorithms* not *methods*
- Multi-core computing is tricksy
  - Race conditions
  - Deadlocks
  - ... (many more)
- Don't want to worry about Image I/O
  - PNGs/JPEGs/TIFFs
  - Movie files
  - Camera sources
- Specify *intent* of algorithm
  - Abstract image sources/sinks
  - Independent processing steps
  - DAG-like layout

Outline

Plain C

- A *Lowest Common Denominator* language
- Fast (single threaded)
- Verbose

```c
/* c-example.c */

#include <stdlib.h> /* malloc */
#include <...>      /* load() and save() */

int main(int argc, char* argv[])
{
  unsigned char *input, *output;
  int row, col;

  input = load();
  output = malloc(sizeof(unsigned char) * WIDTH * HEIGHT);
  for(row = 0; row < 480; ++row)
    for(col = 0; col < 640; ++col) {
      float in =
        (float)(input[col + row*WIDTH]) / 256.f;
      float out = sqrtf(in);
      output[col + row*WIDTH] = (unsigned char)
        (out / 256.f);
    }
  save(output);
  return 0;
}
```

Problems

- Complex and error prone
- Memory leaks easy (see example)
- Tends to lead to *ad hoc* solutions
  - Multiple channels?
  - Different pixel format?
- Requires external libraries
- Not easy to parallelise
  - Rely on auto vectorization per-CPU
  - Use extension like OpenMP for multi-CPU
- Abstractions lead to sub-optimal code

Outline

OpenGL Shader Language

- OpenGL 2.0 and above mandate programmable 'shader' support
- In a pixel shader, each pixel written to screen is generated via a small program from graphics-related inputs
  - Model co-ordinate
  - Texture co-ordinate
  - Surface normal
  - Light positions
  - Textures
  - etc...
- Geared to 3D realtime graphics

```
/* glsl-example.glsl */
uniform sampler2D inputTex;

vec4 shader_function()
{
  vec2 pixel_coord = gl_MultiTexCoord[0];
  vec4 input = texture2D(inputTex, pixel_coord);
  return sqrt(input);
}
```

```
/* glsl-example.c */
setup_and_load_shader();
use_shader();
glBegin(GL_QUADS);
  glTexCoord2f(0,0); glVertex2f(0,0);
  glTexCoord2f(0,1); glVertex2f(0,1);
  glTexCoord2f(1,1); glVertex2f(1,1);
  glTexCoord2f(1,0); glVertex2f(1,0);
glEnd();
```

Problems

- Focussed on rendering, not image processing
- Lots of boilerplate code to set up rendering
- Tricky to render results anywhere other than screen
  - Requires extensions to earlier OpenGL versions
  - not all rendering formats supported

Outline

CoreImage

- Image processing framework from Apple[3]
- Transparently accelerated
- DAG-like rendering pipeline
- 'Minimal work' solution
  - Rendering only happens *Just In Time*
  - Only needed pixels are rendered

[3] `http://developer.apple.com/macosx/coreimage.html`

```
/* coreimage-builtins-example.m */

NSURL *url;
CIImage *source;
CIImage *result;
CIFilter *hueAdjust;

url = [NSURL fileURLWithPath:path];
source = [CIImage imageWithContentsOfURL:url];

hueAdjust = [CIFilter filterWithName:@"CIHueAdjust"];
[hueAdjust setValue:source forKey:@"inputImage"];
[hueAdjust setValue:[NSNumber numberWithFloat:2.094]
            forKey:@"inputAngle"];

result = [hueAdjust valueForKey:@"outputImage"];
```

```
/* coreimage-example.knl */

/* Apply a vertically varying haze removal filter. */
kernel vec4 myHazeRemovalKernel(sampler src,
                                __color color,
                                float distance,
                                float slope)
{
    vec4 t;
    float d;

    d = destCoord().y * slope + distance;
    t = unpremultiply(sample(src, samplerCoord(src)));
    t = (t - d*color) / (1.0-d);

    return premultiply(t);
}
```

Problems

- Only available on Mac OS X 10.5 and later
- A little too transparent
  - Cannot select backend explicitly
  - Only GPU and CPU backend implemented
- Only supports 1-to-1 pixel transforms

Outline

MapReduce

- Massively parallel database framework from Google
- Not graphics focussed *per se*
- Split processing into two sequential individually parallelisable processes
  - **Map**: Transform source data 1-to-1
  - **Reduce**: Compute overall statistic

Definition

- Define record set.
  - For example, each element has count for {male, total} children

$$\mathcal{R} = \{\{1, 2\}, \{2, 4\}, \{0, 1\}, \cdots\}$$

- Wish to compute total number of female children
- Define *map* operation to calculate number of female children:

  funct map(r) $\equiv$
  $(r_2 - r_1)$;
- Define *reduce* operation to sum output from map

  funct reduce(m, s) $\equiv$ { s is pointer to a global state variable }
  atomicinc(s, m);

Definition (cont...)

- MapReduce conceptually runs the following program:

```
begin
    M := ∅
    for r ∈ R do                                   { loop 1 }
        appendtoset(M, map(r));
    od
    s := 0
    for m ∈ M do                                   { loop 2 }
        reduce(addressof(s), m);
    od
end
```

Observations

- Supports efficient insertion of elements into set $\mathcal{R}$
  - Need only run one extra iteration of loops 1 and 2
- If inverse of *reduce* operation is defined removal of an element r from $\mathcal{R}$ is also efficient
  - Need only call reduce$^{-1}(\, s, \text{map}(\, r\,)\,)$
- Inherently parallel
- Not convenient for image processing operations
- Supports computing ensemble statistics
  - Not easy using accelerated frameworks such as GLSL and CoreImage

Lessons Learned

Suppose we wish to create the 'best of all worlds' system:

- Efficient API with minimal boilerplate
- Supports transparent acceleration where possible
  - But retain 'knobs' for advanced usage
- DAG-based pipeline workflow
- Image source/sink agnostic
- Inherently parallel
- Support 1-to-1 and ensemble operations

Part II: The Firtree API

Outline

The Firtree Image Model

- Images are stored as pre-multiplied RGBA 4-tuples
  - Usually in the range (0,1) but not mandated to be so
  - Stored as single-precision floating point values
    - Optional double precision is an ultimate goal
  - Pre-multiplied so that common operations such as alpha over are efficient
- Images are conceptually infinite in extent
  - Do have a concept of a *non-transparent extent* however
- Pixel co-ordinates can be non-integer
- Integer co-ordinates refer to centre of the corresponding pixel
- Images consist of a *recipe* for how to calculate a pixel colour known as the *intent*

The Firtree API

- The Firtree API consists of a low-level GObject-based OO interface
  - The low-level API is easily bound to higher-level languages
  - Ships with Python API
  - Test-suite actually written in Python
- API is *intent* based
  - Record the intent of an image operation
    - DAG of input and processing nodes leading to a single output
    - No rendering is performed during specification of intent[4]
  - Intent compiled down to an intermediate representation
  - Pluggable backends render result on demand

[4] unless asked for!

Outline

GObject

- Pure C object model API
- As nice as possible, given it is C
- Have to manage own memory
- Very low-level
- Used by a number of projects as their *glue layer*
- Designed to be bound to high-level languages

The Firtree GObject API

```
1  /* firtree-gobject.c */
2
3  #include <firtree/firtree.h>
4
5  /* ... */
6
7  FirtreeKernel *k = firtree_kernel_new();
8  const char[] src =
9          "kernel vec4 redKernel() {"
10         "  return vec4(1,0,0,1);"
11         "}";
12 firtree_kernel_compile_from_source(k, &src, 1, NULL);
13
14 g_object_unref(k);
```

Outline

The Firtree Python API

- Automatically generated from GObject API
- Memory management taken care of by Python interpreter
- Language has object orientation syntactic sugar
- Greater SNR
- Interoperates with other Python libraries for image I/O, etc.
  - Don't re-invent the wheel
  - Let people use the tools they want

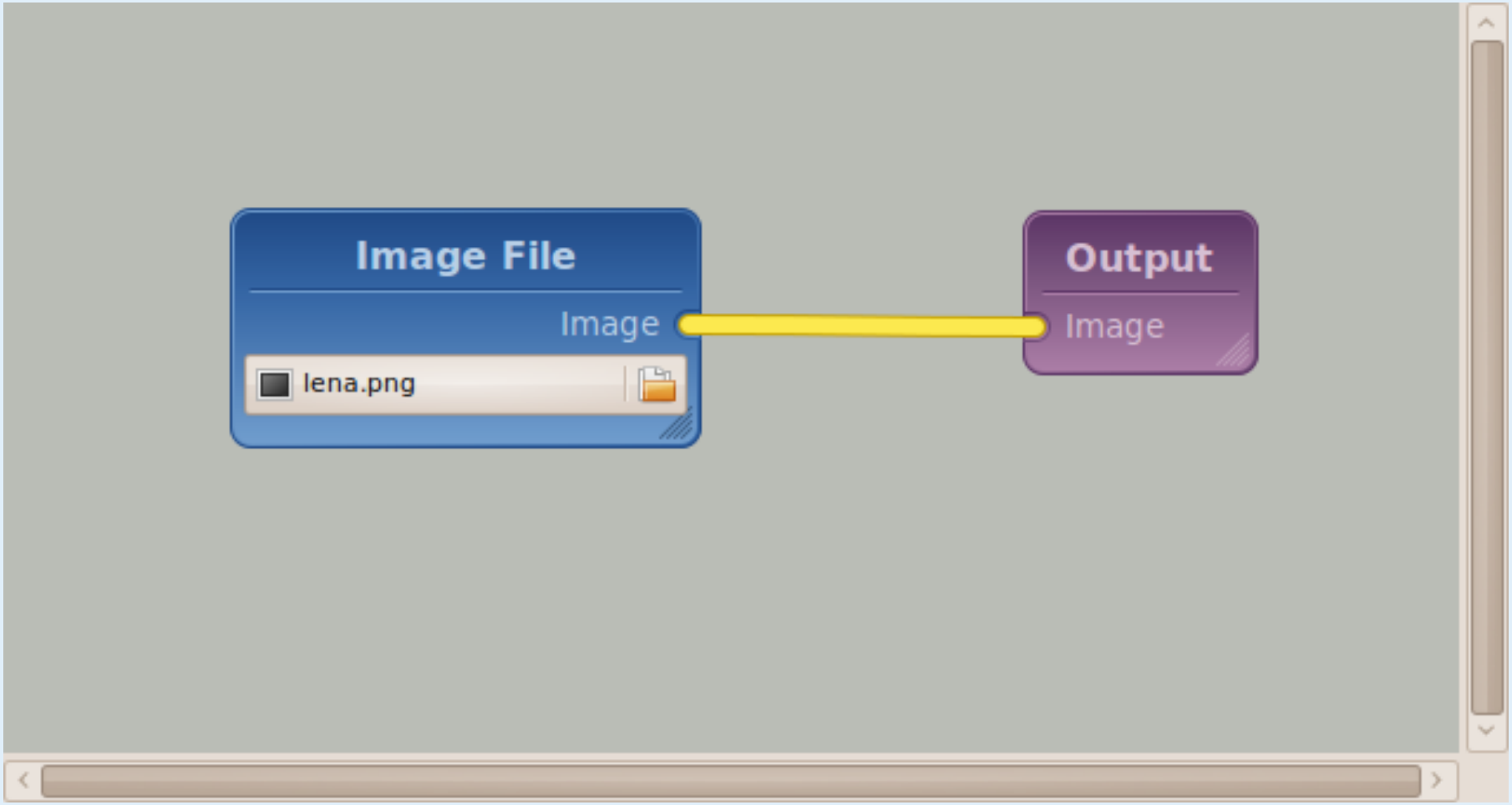


Image File
Image
lena.png
Output
Image

```
#!/usr/bin/env python

import cairo
import pyfirtree as ft

# Firstly, load the lena image
lena_surface = cairo.ImageSurface.create_from_png('lena.png')

# Create a sampler for the surface
lena_sampler = ft.CairoSurfaceSampler()
lena_sampler.set_cairo_surface(lena_surface)

# Create an output surface similar to the input
output_surface = cairo.ImageSurface(
        cairo.FORMAT_ARGB32,
        lena_surface.get_width(),
        lena_surface.get_height() )
```

```
# Create a CPU render engine.
engine = ft.CpuRenderer()

# Use the engine to write the input to the output.
engine.set_sampler(lena_sampler)
engine.render_into_cairo_surface(
        lena_sampler.get_extent(), # what area to render
        output_surface             # into what
        )

# Write the output
output_surface.write_to_png('output.png')
```

Input

Output

Samplers

- Know how to generate the red, green, blue and alpha components of an output pixel given its location
- Has an associated transform which maps from image-space co-ordinates to sampler-space co-ordinates
  - Allows transformation of images via a *pass through* sampler
- Black boxes but may depend on other samplers
- Supports three primitive operations
  - Prepare: Called once before the render
  - Query: Called one or more times during the render
  - Finish: Called once after the render

Sampler Types

- A sampler is an abstract node
- Concrete implementations exist for various different images sources
- Buffer samplers sample from an in-memory image
  - Cairo surfaces, GdkPixbufs, images loaded by your own libraries (e.g. The PIL), etc.
  - Also a GStreamer plugin which uses a buffer sampler to provide video stream support
- Kernel samplers use a small piece of code to generate the output pixel
  - The code may, itself, sample from other samplers
- Advanced users can define their own samplers by implementing the three sampler primitives

The Firtree Kernel Language

- The most interesting samplers are *kernel samplers*
- A function which is called (conceptually) once per output pixel
- Returns the colour of a pixel given a *destination co-ordinate*
- Language is based on a subset of C
- Extended with vectorised types
  - `a + b` → element-wise addition
  - `a * b` → element-wise multiplication (not matrix multiplication)
- Very similar to GLSL or CoreImage kernels
  - Inspired by CoreImage
  - Originally implemented with the GLSL reference compiler

Part III: Examples

Outline

Desaturate

Sobel Filter

Mandelbrot Set

Reduce Kernels

Real-time Video Processing

Desaturate

- Convert an RGB image into the Y-component of a YUV image
- $Y = 0.299R + 0.587G + 0.114B$
- Use a simple kernel which uses the builtin **dot** function.

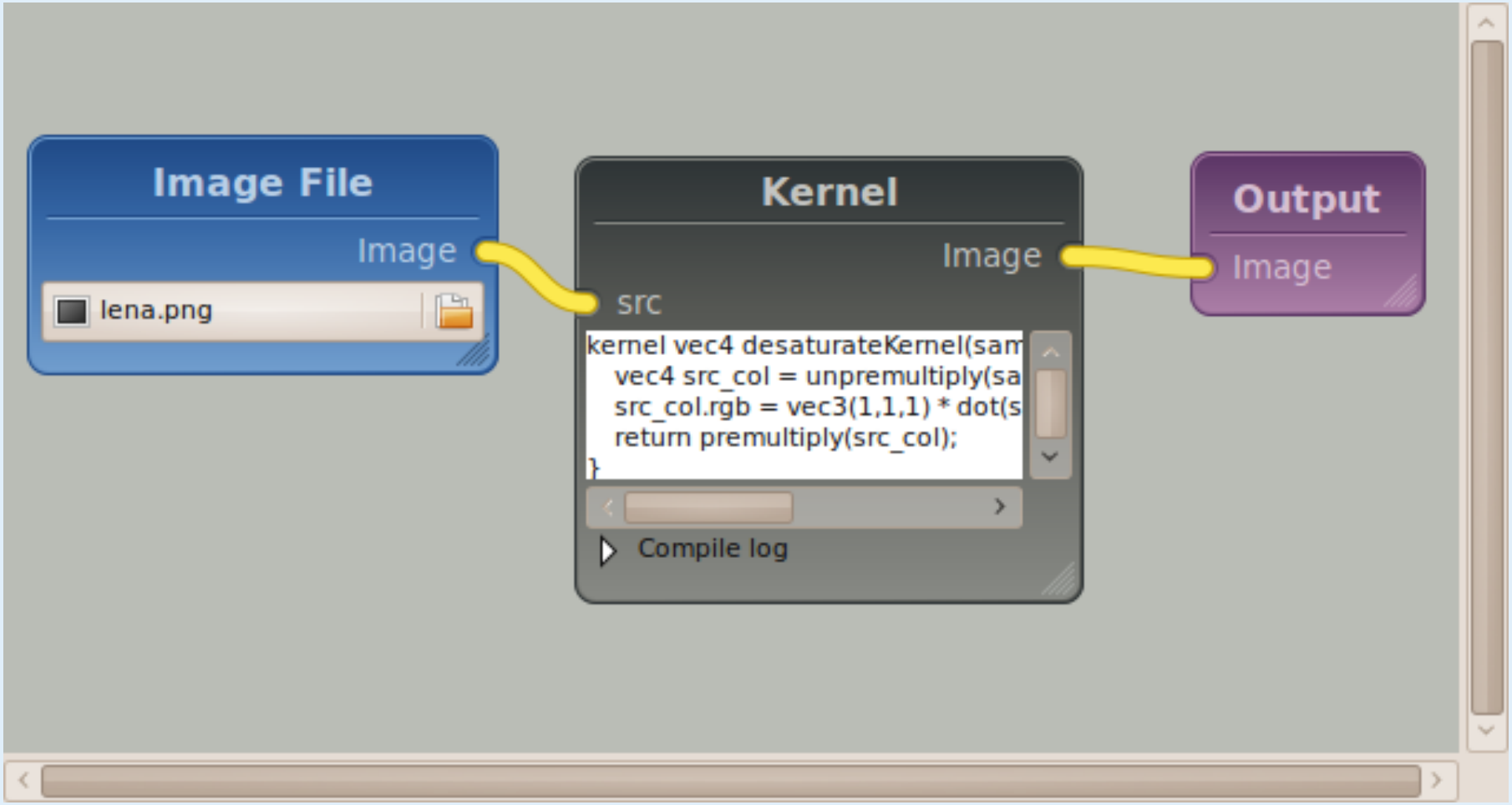


Image File
Image
lena.png
Kernel
Image
src
kernel vec4 desaturateKernel(sam
vec4 src_col = unpremultiply(sa
src_col.rgb = vec3(1,1,1) * dot(s
return premultiply(src_col);
}
Compile log
Output
Image

```
#!/usr/bin/env python

import cairo
import pyfirtree as ft

def compile_kernel(source):
        """
        A simple function which compiles a kernel, checks that
        the compilation succeeded and returns a pair containing
        the kernel and a sampler for it.
        """

        kernel = ft.Kernel()
        kernel.compile_from_source(source)
        if not kernel.get_compile_status():
                print("Error compiling kernel:")
                print("\n".join(kernel.get_compile_log()))
                return

        kernel_sampler = ft.KernelSampler()
        kernel_sampler.set_kernel(kernel)

        return (kernel, kernel_sampler)
```

```
# Firstly, load the lena image
lena_surface = cairo.ImageSurface.create_from_png('lena.png')

# Create a sampler for the surface
lena_sampler = ft.CairoSurfaceSampler()
lena_sampler.set_cairo_surface(lena_surface)

# Create a desaturate kernel.
(desat, desat_sampler) = compile_kernel("""
  kernel vec4 desaturate(sampler src)
  {
    vec4 src_colour =
      unpremultiply( sample(src, samplerCoord(src)) );
    float luminance =
      dot(src_colour, vec4(0.299,0.587,0.114,0));
    return premultiply(
      vec4(luminance,luminance,luminance,src_colour.a)
    );
  }
""")

# Wire the lena sampler into the desaturate kernel.
desat['src'] = lena_sampler
```

```
# Create an output surface similar to the input
output_surface = cairo.ImageSurface(
        cairo.FORMAT_ARGB32,
        lena_surface.get_width(),
        lena_surface.get_height() )

# Create a CPU render engine.
engine = ft.CpuRenderer()

# Use the engine to render the output.
engine.set_sampler(desat_sampler)
engine.render_into_cairo_surface(
        lena_sampler.get_extent(), # what area to render
        output_surface             # into what
        )

# Write the output
output_surface.write_to_png('output.png')
```

Input

Output

Sobel Filter

- The Sobel Filter is a simple edge detector
- Convolves image with two kernels

$$I_x = I * \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad I_y = I * \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- Computes edge response as $\sqrt{I_x^2 + I_y^2}$
- Need only append a kernel to the desaturate example above

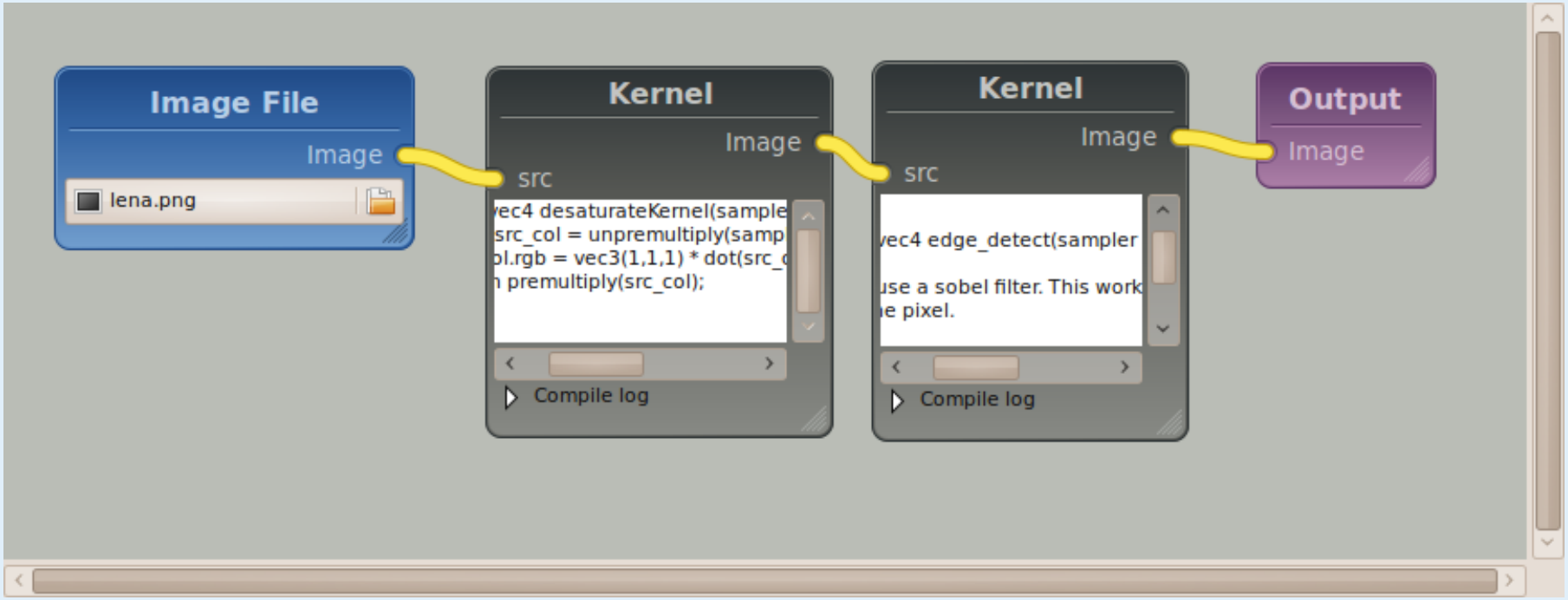


Image File
Image
lena.png
Kernel
Image
src
vec4 desaturateKernel(sample
src_col = unpremultiply(samp
ol.rgb = vec3(1,1,1) * dot(src_
n premultiply(src_col);
Compile log
Kernel
Image
src
vec4 edge_detect(sampler
use a sobel filter. This work
e pixel.
Compile log
Output
Image

```
# Create an edge detector kernel.
(edge, edge_sampler) = compile_kernel("""
        // Return the (unpremultiplied) value of the pixel
        // at a particular offset.

                                ...


""")

edge['src'] = desat_sampler

# Use the engine to render the output.
engine.set_sampler(edge_sampler)
engine.render_into_cairo_surface(
        lena_sampler.get_extent(), # what area to render
        output_surface             # into what
        )

# Write the output
output_surface.write_to_png('output.png')
```

```
// Return the (unpremultiplied) value of the pixel
// at a particular offset.
vec4 pixel_at(sampler src, vec2 offset)
{
  return unpremultiply( sample( src,
    samplerTransform( src, destCoord() + offset) ) );
}

kernel vec4 edge_detect(sampler src)
{
  // We use a sobel filter. This works on the 3x3 neighbourhood
  // of the pixel.

  // Get the neighbourhood.
  vec4 p00 = pixel_at(src, vec2( -1, -1 ));
  vec4 p10 = pixel_at(src, vec2(  0, -1 ));
  vec4 p20 = pixel_at(src, vec2(  1, -1 ));
  vec4 p01 = pixel_at(src, vec2( -1,  0 ));
  vec4 p21 = pixel_at(src, vec2(  1,  0 ));
  vec4 p02 = pixel_at(src, vec2( -1,  1 ));
  vec4 p12 = pixel_at(src, vec2(  0,  1 ));
  vec4 p22 = pixel_at(src, vec2(  1,  1 ));
```

```
    float src_alpha = sample(src, samplerCoord(src)).a;

    // Find the sobel response in X- and Y-directions.
    vec4 dx = p00 - p20 + 2 * (p01 - p21) + p02 - p22;
    vec4 dy = p00 - p02 + 2 * (p10 - p12) + p20 - p22;

    // Calculate the magnitude of the response, we
    // multiply by 0.25 so that the filter does not
    // give a response > 1.
    vec4 mag = 0.25 * sqrt( dx*dx + dy*dy );

    return premultiply( vec4( mag.rgb, src_alpha ) );
}
```

Input

Output

Extending Sobel Filter

- The Sobel kernel we defined earlier used 4-way vector arithmetic
- We can extend the program to give a colour output easily by simply 'unplugging' the desaturate stage
- This requires precisely one line change:

```
1 edge['src'] = lena_sampler
```

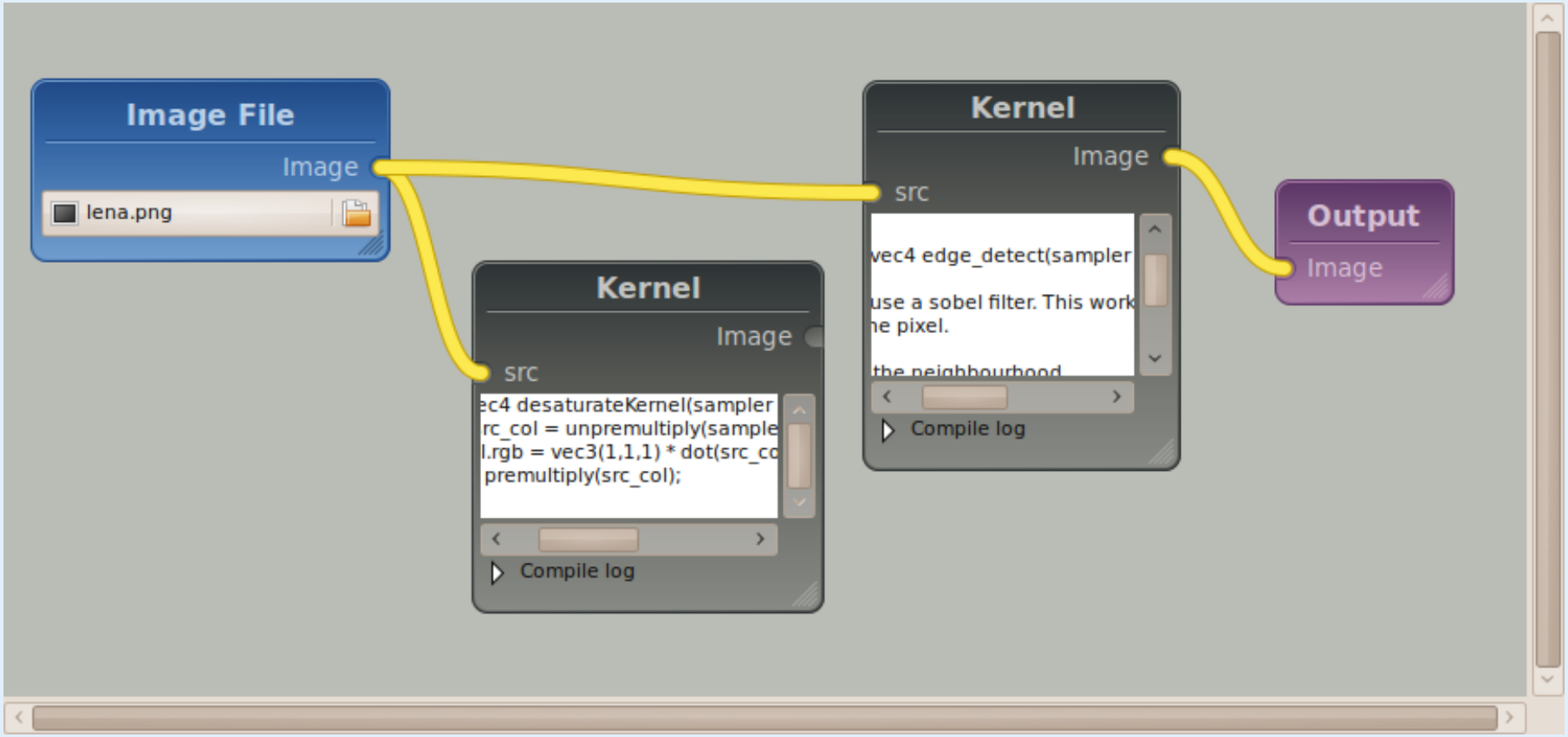


Image File
Image
lena.png
Kernel
Image
src
ec4 desaturateKernel(sampler
rc_col = unpremultiply(sample
l.rgb = vec3(1,1,1) * dot(src_co
premultiply(src_col);
Compile log
Kernel
Image
src
vec4 edge_detect(sampler
use a sobel filter. This work
he pixel.
the neighbourhood
Compile log
Output
Image

Input

Output

```
# Create a kernel which computes the mandelbrot set
(mandelbrot, mandelbrot_sampler) = compile_kernel("""

  vec2 complex_mul(vec2 a, vec2 b) {
    return vec2(a.x*b.x - a.y*b.y, a.x*b.y + a.y*b.x);
  }

  kernel vec4 mandelbrot()
  {
    int max_iterations = 40;
    vec2 c = destCoord(), z = c;

    int num_its;
    float mag_z_sq = dot(z,z);
    for(num_its = 0;
     (num_its < max_iterations) && (mag_z_sq < 4.0); ++num_its)
        z = complex_mul(z,z) + c;
        mag_z_sq = dot(z,z);
    }

    float output_val = num_its / max_iterations;
    return vec4(output_val, output_val, output_val, 1);
  }
""")
```

```
output_surface = cairo.ImageSurface(
        cairo.FORMAT_ARGB32,
        1200, 800 )
engine = ft.CpuRenderer()
engine.set_sampler(mandelbrot_sampler)
engine.render_into_cairo_surface(
        (-2, -1, 3, 2),  # what area to render
        output_surface   # into what
        )
output_surface.write_to_png('mandelbrot.png')
```

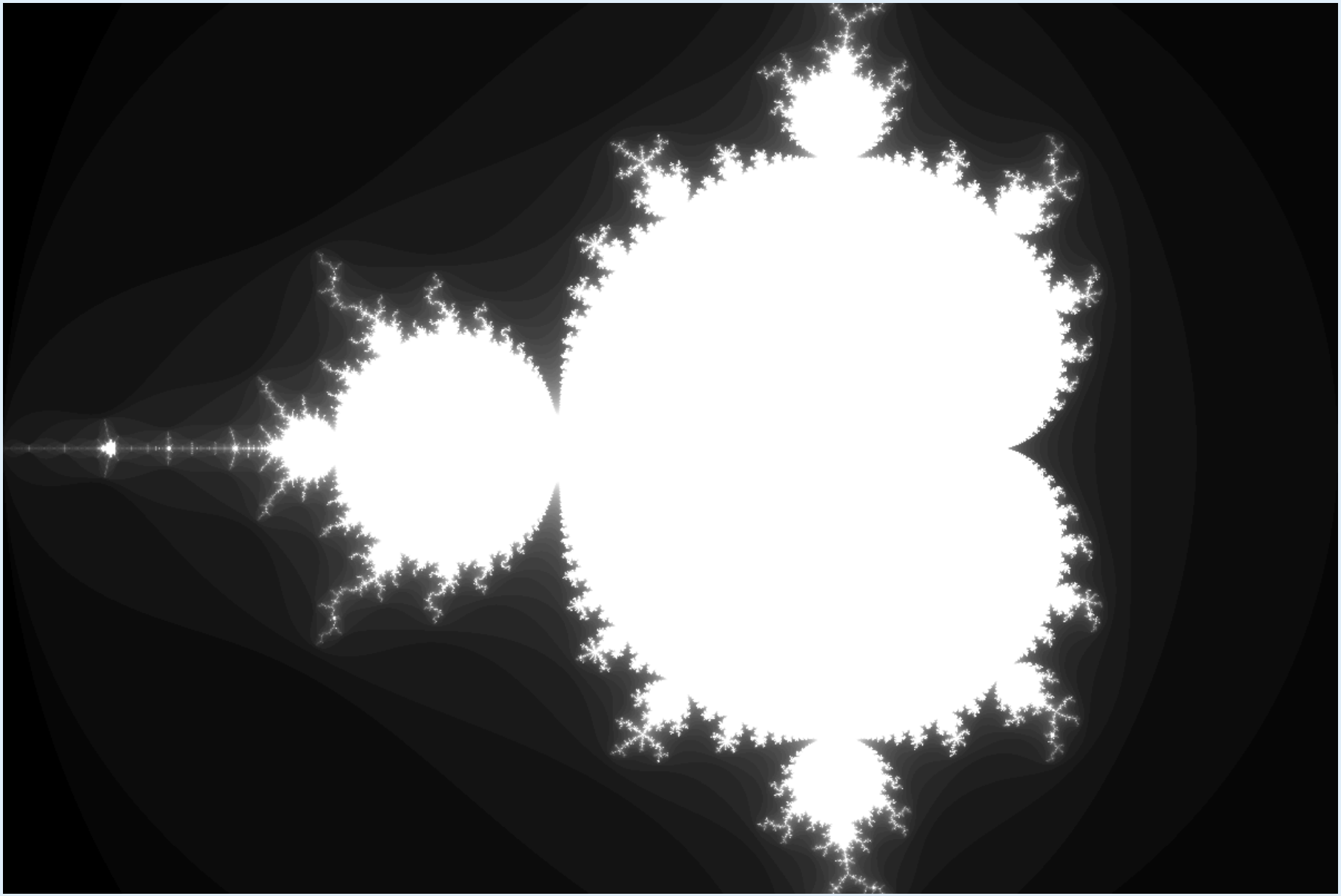


Figure: Output of `mandelbrot.py`

Reduce Kernels

- Like normal image kernels except that they have no output
  - Kernel function returns **void**
  - They cannot be associated with a sampler
- Are conceptually called once per-pixel but in undefined order
- May make use of one extra builtin: **emit()**
  - Appends a **vec4** value to a global list
- Somewhat misnamed; it is more of a selective filter + map

```
def compile_reduce_kernel(source):
        kernel = ft.Kernel()
        kernel.compile_from_source(source)
        if not kernel.get_compile_status():
                print("Error compiling kernel:")
                print("\n".join(kernel.get_compile_log()))
                return
        return kernel
```

```
# Create a reduce kernel which looks for areas of high red.
reduce = compile_reduce_kernel("""
  kernel __reduce void high_red(sampler src)
  {
    vec4 src_colour =
      unpremultiply( sample(src, samplerCoord(src)) );
    if( src_colour.r - max(src_colour.g, src_colour.b) > 0.1 )
    {
      emit( vec4( destCoord(), 0, 0 ) );
    }
  }
""")

reduce['src'] = edge_sampler

# Create a CPU reduce engine.
engine = ft.CpuReduceEngine()
engine.set_kernel(reduce)
```

```
# Find the matching points
edge_points = engine.run(
        lena_sampler.get_extent(),          # what area of the inp
        lena_surface.get_width(),          # how many horizontal s
        lena_surface.get_height()          # how many vertical sam
        )
print('%i edge points found' % len(edge_points))

# Draw them on the surface
cr = cairo.Context(lena_surface)
cr.set_source_rgba(0,1,0,1)
for point in edge_points:
        cr.move_to(point[0] - 2, point[1] - 2)
        cr.line_to(point[0] + 2, point[1] + 2)
        cr.move_to(point[0] + 2, point[1] - 2)
        cr.line_to(point[0] - 2, point[1] + 2)
        cr.stroke()

# Write the output
lena_surface.write_to_png('output.png')
```

Input

Output

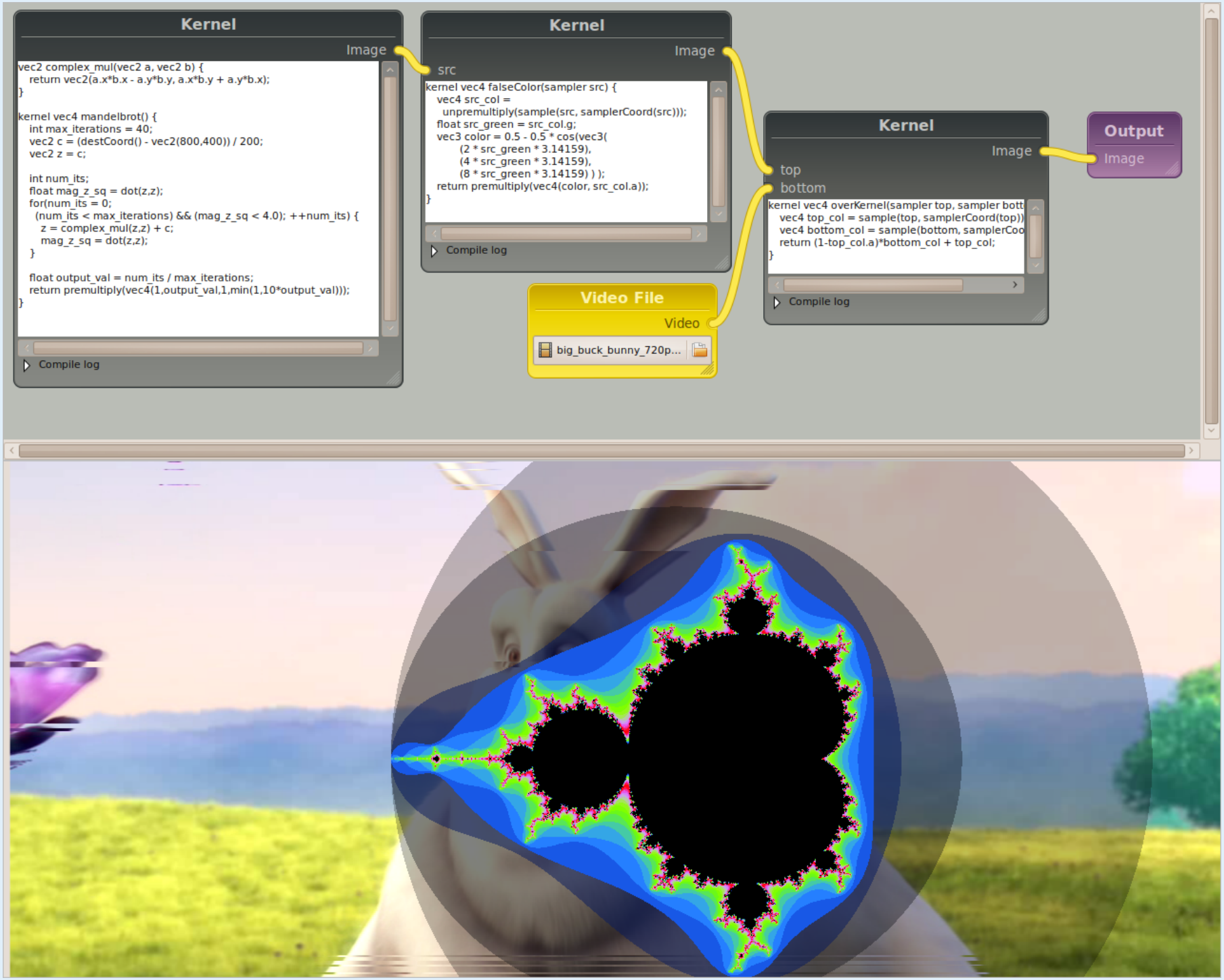
```
valid_extents = lena_sampler.get_extent()
valid_extents = (
        valid_extents[0] + 1, valid_extents[1] + 1,
        valid_extents[2] - 2, valid_extents[3] - 2 )

# Find the matching points
edge_points = engine.run(
        valid_extents,            # what area of the input to ren
        int(valid_extents[2]),    # how many horizontal samples
        int(valid_extents[3]),    # how many vertical samples
        )

# Draw them on the surface
cr = cairo.Context(lena_surface)
cr.set_source_rgba(0,1,0,1)
for point in edge_points:
        cr.move_to(point[0] - 2, point[1] - 2)
        cr.line_to(point[0] + 2, point[1] + 2)
        cr.move_to(point[0] + 2, point[1] - 2)
        cr.line_to(point[0] - 2, point[1] + 2)
        cr.stroke()

# Write the output
lena_surface.write_to_png('output.png')
```

Input

Output

Real-time Video Processing

- Sampler is sufficiently general that we can make a video source a sampler
- Have integrated Firtree with the GStreamer project
  - GStreamer allows one to create 'pluggable' modules that can decode video, camera streams, etc
- Create a GStreamer *sink* node which appears as a Firtree sampler


Kernel
Image
vec2 complex_mul(vec2 a, vec2 b) {
return vec2(a.x*b.x - a.y*b.y, a.x*b.y + a.y*b.x);
}
kernel vec4 mandelbrot() {
int max_iterations = 40;
vec2 c = (destCoord() - vec2(800,400)) / 200;
vec2 z = c;
int num_its;
float mag_z_sq = dot(z,z);
for(num_its = 0;
(num_its < max_iterations) && (mag_z_sq < 4.0); ++num_its) {
z = complex_mul(z,z) + c;
mag_z_sq = dot(z,z);
}
float output_val = num_its / max_iterations;
return premultiply(vec4(1,output_val,1,min(1,10*output_val)));
}
Compile log
Kernel
Image
src
kernel vec4 falseColor(sampler src) {
vec4 src_col =
unpremultiply(sample(src, samplerCoord(src)));
float src_green = src_col.g;
vec3 color = 0.5 - 0.5 * cos(vec3(
(2 * src_green * 3.14159),
(4 * src_green * 3.14159),
(8 * src_green * 3.14159)));
return premultiply(vec4(color, src_col.a));
}
Compile log
Video File
Video
big_buck_bunny_720p...
Kernel
Image
top
bottom
kernel vec4 overKernel(sampler top, sampler bott
vec4 top_col = sample(top, samplerCoord(top))
vec4 bottom_col = sample(bottom, samplerCoo
return (1-top_col.a)*bottom_col + top_col;
}
Compile log
Output
Image

Part IV: Implementation

Outline

Compilation Process

LLVM

Sampler Linking

Backends

- Multi-core CPU
- The future: GPU

Compilation Process

- Frontend compilation
  - Each sampler can represent its sampling action via a common Intermediate Representation (IR)
  - IR has a number of low-level intrinsic functions
    - Sampling from image buffers, arithmetic operations, transcendental/trigonometric functions, etc.
- Backend execution
  - On first render, backend JIT compiles IR into native code
  - Backend distributed work units to each compute unit
    - e.g. CPU backend uses one thread per CPU which render blocks of output independently

Compilation Process (cont...)

- Each sampler generates IR for its action
- The IR linker automatically combines each sampler with the IR for its dependent samplers to produce a single block of IR
- The linking stage and those that precede it are backend independent
- The same IR may be passed to different backends if necessary

LLVM

- Make use of the IR from the LLVM project[5]
- Provides an in-memory and human-readable variant
- Includes a suite of modern optimisation stages
- Includes native vector-aware backends for most modern CPUs
- nVidia have a backend which targets their compute-capable graphics cards

[5] http://llvm.org/

LLVM Example

- Use the simple identity pipeline example

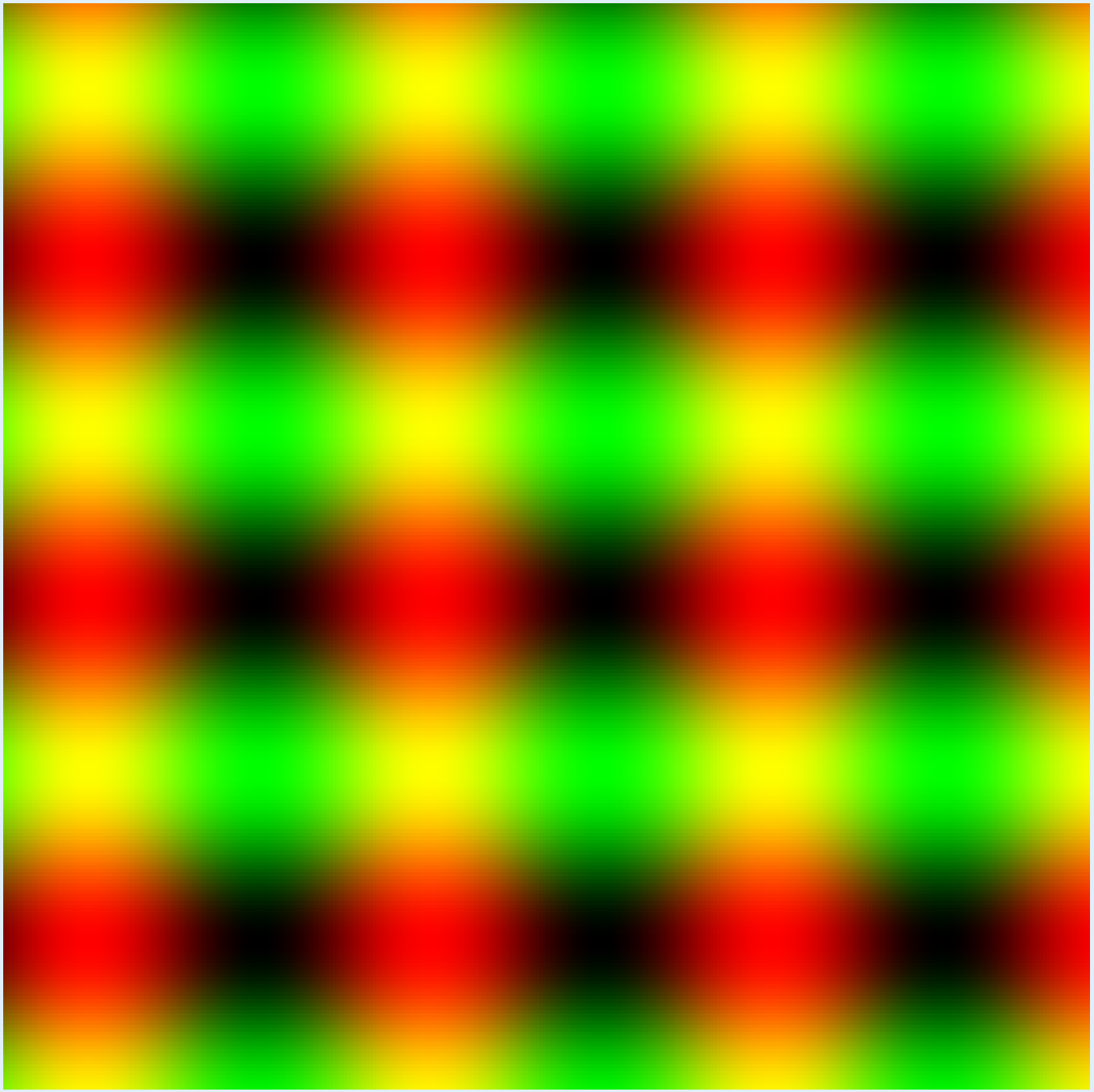
```
lena_sampler = ft.CairoSurfaceSampler()
lena_sampler.set_cairo_surface(lena_surface)
```

- Sampler generates following IR

```
; Builtin IR intrinsic
declare <4 x float> @sample_image_buffer_nn(
    i8*, i32, i32, i32, i32, <4 x float>)

; Sampler function itself
define <4 x float>
  @sampler_dc2e6e61_1920_08f4_898f_2456c40f1e6b(<4 x float>) {
entry:
  %rv = call <4 x float> @sample_image_buffer_nn(
    i8* inttoptr (i64 -1270542328 to i8*),
    i32 13, i32 512, i32 512,
    i32 2048, <4 x float> %0) ; <<4 x float>> [#uses=1]
  ret <4 x float> %rv
}
```

```
kernel vec4 sinKernelExample()
{
  return vec4( 0.5 + 0.5 * sin( destCoord() ), 0, 1 );
}
```

```
; IR builtin
declare <4 x float> @sin_v2(<4 x float>) nounwind readnone

; Sampler function proper
define <4 x float>
  @sampler_35014ecb_ea33_8994_f1ed_88ceb0e101b7(<4 x float>) {
entry:
  %tmp.i = call <4 x float> @sin_v2(<4 x float> %0) nounwind
  %tmp3.i = fmul <4 x float> %tmp.i, <float 5.000000e-01,
    float 5.000000e-01, float 0.000000e+00, float 0.000000e+00>
  %tmp6.i = fadd <4 x float> %tmp3.i, <float 5.000000e-01,
    float 5.000000e-01, float 0.000000e+00, float 0.000000e+00>
  %tmp13.i = insertelement <4 x float> %tmp6.i,
    float 0.000000e+00, i32 2
  %tmp14.i = insertelement <4 x float> %tmp13.i,
    float 1.000000e+00, i32 3
  ret <4 x float> %tmp14.i
}
```



Sampler Linking

- Samplers may call other samplers

```
1 kernel vec4 sinexample(sampler src)
2 {
3   vec2 dc = destCoord() + 5 * sin(destCoord() / 10);
4   return sample(src, samplerTransform(src, dc));
5 }
```

- Kernel is compiled into IR which makes use of sampler functions

```
kernel vec4 sinKernelExample()
{
  return vec4( 0.5 + 0.5 * sin( destCoord() ), 0, 1 );
}
```

```
; Intrinsics
declare <4 x float> @sin_v2(<4 x float>) nounwind readnone
declare <4 x float> @samplerTransform_sv2(i32, <4 x float>) nounwind readnone
declare <4 x float> @sample_sv2(i32, <4 x float>) nounwind readnone

; Kernel function
define <4 x float>
  @kernel_4bcd26e6_ec72_7454_f967_7b7d92125bba(
    <4 x float> %__this__destCoord, i32 %src) nounwind readnone {
entry:
  %tmp3 = fdiv <4 x float> %__this__destCoord, <float 1.000000e+01,
    float 1.000000e+01, float 0.000000e+00, float 0.000000e+00>
  %tmp4 = call <4 x float> @sin_v2(<4 x float> %tmp3)
  %tmp8 = fmul <4 x float> %tmp4, <float 5.000000e+00, float 5.000000e+00,
    float 0.000000e+00, float 0.000000e+00>
  %tmp9 = fadd <4 x float> %tmp8, %__this__destCoord
  %tmp13 = call <4 x float> @samplerTransform_sv2(i32 %src, <4 x float> %tmp9)
  %tmp14 = call <4 x float> @sample_sv2(i32 %src, <4 x float> %tmp13)
  ret <4 x float> %tmp14
}
```

Linker Output

Setting lena as `src` causes the linker to inline calls to the sampler:

```
; Builtins
declare <4 x float> @sin_v2(<4 x float>) nounwind readnone
declare <4 x float> @sample_image_buffer_nn(i8*, i32, i32, i32, i32, <4 x float>)

; Sampler function
define <4 x float>
  @sampler_8e662b89_bf6d_26f4_412b_61c73e556e92(<4 x float>) {
entry:
  %tmp3.i = fdiv <4 x float> %0, <float 1.000000e+01, float 1.000000e+01,
    float 0.000000e+00, float 0.000000e+00>
  %tmp4.i = call <4 x float> @sin_v2(<4 x float> %tmp3.i) nounwind
  %tmp8.i = fmul <4 x float> %tmp4.i, <float 5.000000e+00, float 5.000000e+00,
    float 0.000000e+00, float 0.000000e+00>
  %tmp9.i = fadd <4 x float> %tmp8.i, %0
  %rv.i.i.i = call <4 x float> @sample_image_buffer_nn(
    i8* inttoptr (i64 -1269264376 to i8*), i32 13, i32 512, i32 512, i32 2048,
    <4 x float> %tmp9.i) nounwind
  ret <4 x float> %rv.i.i.i
}
```

Backends

Backends are responsible for platform specific tasks:

- Compiling LLVM IR to native code
- Implementing the intrinsics
- Distributing rendering onto multiple cores

Outline

The CPU backend

- Wraps the sampler IR in LLVM IR which implements an inner rendering loop
  - Similar to the plain C example at the start of the talk
- Maintains one worker thread per-CPU
- Divides rendering task into small chunks
- Each worker thread gets a new chunk from a global queue
- Easier parts of the image do not cause threads to become idle

The generated CPU assembly (x86 in this case) can be quite daunting

```
    .file   ""


    .section    .rodata.cst8,"aM",@progbits,8
    .align  8
.LCPI1_0:
# constant double
    .quad   4841369599423283200
# double 4.503600e+15
    .section    .rodata.cst4,"aM",@progbits,4
    .align  4
.LCPI1_1:
# constant float
    .long   1056964608
# float 5.000000e-01
.LCPI1_2:
# constant float
    .long   998277249
# float 3.921569e-03
.LCPI1_3:
# constant float
    .long   1065353216
# float 1.000000e+00
    .section    .rodata.cst16,"aM",@progbits,16
    .align  16
.LCPI1_4:
# constant <4 x i32>
    .zero   16
    .text
    .align  16
    .globl  render_FIRTREE_FORMAT_RGBA_F32_PREMULTIPLIED
    .type   render_FIRTREE_FORMAT_RGBA_F32_PREMULTIPLIED,@function
render_FIRTREE_FORMAT_RGBA_F32_PREMULTIPLIED:
# @render_FIRTREE_FORMAT_RGBA_F32_PREMULTIPLIED
.LBB1_0:
# %entry
    pushl   %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    pushl   %ebx
    pushl   %esi
    subl    $40, %esp
    cmpl    $0, 16(%ebp)
    je  .LBB1_10
.LBB1_1:
# %entry
    movl    12(%ebp), %eax
    testl   %eax, %eax
    je  .LBB1_10
.LBB1_2:
# %bb.nph22.split
    movsd   .LCPI1_0, %xmm0
    movd    16(%ebp), %xmm1
    orps    %xmm0, %xmm1
    subsd   %xmm0, %xmm1
    cvtsd2ss    %xmm1, %xmm1
    movl    24(%ebp), %ecx
    movss   12(%ecx), %xmm2
    divss   %xmm1, %xmm2
    movss   %xmm2, (%esp)
    movaps  %xmm2, %xmm1
    movss   .LCPI1_1, %xmm2
    mulss   %xmm2, %xmm1
    movss   %xmm1, 8(%esp)
    movd    %eax, %xmm1
    orps    %xmm0, %xmm1
    subsd   %xmm0, %xmm1
    cvtsd2ss    %xmm1, %xmm0
    movss   8(%ecx), %xmm1
    divss   %xmm0, %xmm1
    mulss   %xmm1, %xmm2
    movss   %xmm2, 16(%esp)
    movss   4(%ecx), %xmm0
    movss   (%ecx), %xmm2
    movss   %xmm2, 20(%esp)
    movl    8(%ebp), %ecx
    movl    %ecx, 24(%esp)
    cmpl    $1, %eax
    movl    $1, %ecx
    cmova   %eax, %ecx
    movl    %ecx, 4(%esp)
    movl    $0, 12(%esp)
    movss   .LCPI1_2, %xmm2
    movss   .LCPI1_3, %xmm3
.LBB1_3:
# %bb
                                        # Loop Depth 1
                                        # Loop Header
```

```

Child Loop BB1_12 Depth 2

Child Loop BB1_6 Depth 2
    movss   8(%esp), %xmm4
    addss   %xmm0, %xmm4
    movss   %xmm4, 8(%esp)
    pxor    %xmm0, %xmm0
    ucomiss %xmm4, %xmm0
    movl    $0, %eax
    movl    $4294967295, %ecx
    cmova   %ecx, %eax
    cvttss2si   %xmm4, %ecx
    leal    (%eax,%ecx), %edx
    shll    $11, %edx
    movl    %edx, 28(%esp)
    addl    %ecx, %eax
    js  .LBB1_5
.LBB1_4:
# %bb


    cmpl    $511, %eax
    jbe .LBB1_11
.LBB1_5:
# %bb1.us.preheader


    movl    24(%esp), %eax
    movl    4(%esp), %ecx
    movss   16(%esp), %xmm0
    movss   20(%esp), %xmm4
    .align  16
.LBB1_6:
# %bb1.us


    movups  (%eax), %xmm5
    addps   .LCPI1_4, %xmm5
    movss   %xmm5, (%eax)
    pshufd  $1, %xmm5, %xmm6
    movss   %xmm6, 4(%eax)
    pshufd  $3, %xmm5, %xmm6
    movhlps %xmm5, %xmm5
    movss   %xmm5, 8(%eax)
    movss   %xmm6, 12(%eax)
    addss   %xmm4, %xmm0
    addl    $16, %eax
    decl    %ecx
    movaps  %xmm1, %xmm4
    jne .LBB1_6
    jmp .LBB1_9
.LBB1_7:
# %bb15.i69.i.i.i


    movl    28(%esp), %edx
    movl    -1269403640(%edx,%esi,4), %edx
    movl    %edx, %ebx
    movzbl  %bl, %esi
    cvtsi2ss    %esi, %xmm6
    mulss   %xmm2, %xmm6
    shrl    $16, %edx
    movzbl  %dl, %edx
    cvtsi2ss    %edx, %xmm5
    mulss   %xmm2, %xmm5
    unpcklps    %xmm6, %xmm5
    movzbl  %bh, %edx
    cvtsi2ss    %edx, %xmm6
    mulss   %xmm2, %xmm6
    unpcklps    %xmm3, %xmm6
    unpcklps    %xmm6, %xmm5
.LBB1_8:
# %sampler_render_function.exit


```

```
#

#


# Loop Depth 1
# Loop Header is BB1_3


# Loop Depth 1
# Loop Header is BB1_3


# Loop Depth 2
# Loop Header
# Inner Loop
# Inside Loop BB1_3 Depth 1


# Loop Depth 2
# Loop Header is BB1_12
# Inner Loop
# Inside Loop BB1_3 Depth 1


# Loop Depth 2
# Loop Header is BB1_12
# Inner Loop
```

```
                                        # Inside Loop BB1_3 Depth 1
    pshufd  $3, %xmm5, %xmm6
    movss   .LCPI1_3, %xmm7
    subss   %xmm6, %xmm7
    pshufd  $0, %xmm7, %xmm6
    mulps   %xmm4, %xmm6
    addps   %xmm5, %xmm6
    movss   %xmm6, (%eax)
    pshufd  $1, %xmm6, %xmm4
    movss   %xmm4, 4(%eax)
    pshufd  $3, %xmm6, %xmm4
    movhlps %xmm6, %xmm6
    movss   %xmm6, 8(%eax)
    movss   %xmm4, 12(%eax)
    addl    $16, %eax
    decl    %ecx
    movaps  %xmm1, %xmm4
    jne .LBB1_12
.LBB1_9:
# %bb5.split
                                        # Loop Depth 1
                                        # Loop Header is BB1_3
    movl    24(%esp), %eax
    addl    20(%ebp), %eax
    movl    %eax, 24(%esp)
    movl    12(%esp), %eax
    incl    %eax
    movl    %eax, 12(%esp)
    cmpl    16(%ebp), %eax
    movss   (%esp), %xmm0
    jb  .LBB1_3
.LBB1_10:
# %return
    addl    $40, %esp
    popl    %esi
    popl    %ebx
    movl    %ebp, %esp
    popl    %ebp
    ret
.LBB1_11:
# %bb1.preheader
                                        # Loop Depth 1
                                        # Loop Header is BB1_3
    movl    24(%esp), %eax
    movl    4(%esp), %ecx
    movss   16(%esp), %xmm0
    movss   20(%esp), %xmm4
.LBB1_12:
# %bb1
                                        # Loop Depth 2
                                        # Loop Header
                                        # Inner Loop
                                        # Inside Loop BB1_3 Depth 1
    addss   %xmm4, %xmm0
    pxor    %xmm4, %xmm4
    ucomiss %xmm0, %xmm4
    movl    $0, %edx
    movl    $4294967295, %esi
    cmova   %esi, %edx
    cvttss2si   %xmm0, %esi
    addl    %edx, %esi
    movups  (%eax), %xmm4
    js  .LBB1_14
.LBB1_13:
# %bb1
                                        # Loop Depth 2
                                        # Loop Header is BB1_12
                                        # Inner Loop
                                        # Inside Loop BB1_3 Depth 1
    cmpl    $512, %esi
    jb  .LBB1_7
.LBB1_14:
# %bb1.sampler_render_function.exit_crit_edge
                                        # Loop Depth 2
                                        # Loop Header is BB1_12
                                        # Inner Loop
                                        # Inside Loop BB1_3 Depth 1
    xorps   %xmm5, %xmm5
    jmp .LBB1_8
    .size   render_FIRTREE_FORMAT_RGBA_F32_PREMULTIPLIED, .-render_FIRTREE_FORMAT_RGBA_F32_PREMULTIPLIED
```

Outline

The future: GPU

- Post Firtree re-write GPU renderer has not been re-implemented
- A proof-of-concept LLVM IR to OpenCL/CUDA converter exists
- Not yet fully implemented
  - Very small number of intrinsics
  - Does not execute the code it compiles
- Waiting for an excuse to implement it
- Some interesting issues arise
  - Backend will also have to maintain buffer cache on compute card
  - Want to create some render *queue* so that entire rendering pipeline becomes asynchronous

The past: GPU

- An earlier version of Firtree had a GPU backend
- Was based on OpenGL + GLSL
- The 'unique' way OpenGL contexts are managed caused it to be brittle
- Was very fast indeed
- Even compute-intensive kernels required negligible CPU
- Some pretty videos were generated

Questions?