Name: Emad Barsoum
UNI: eb2871

COMS W4115
Project Proposal

SIP: Simple Image Processing Language

Vision

Providing a high-level domain-specific language that abstract the complexity of implementing image-processing algorithms, facilitating writing image based techniques quickly without much plumbing and making image processing a first class citizen to the language. SIP has some inspiration from functional programming concept and GPU shading language, however, it is tailored toward image operations.

Motivation

Most of my work involve image processing and computer vision algorithms; and although, there are a lot of tools, libraries and programming languages that assist in writing image algorithms, most of them require writing a lot of code in order to implement a simple image task. There are a lot of domain-specific languages (Matlab, R, SQL, Prolog…etc), commercial and non-commercial, yet there are very few for image processing even with the increasing need for such language.

Most of the efforts in writing domain-specific language for image processing are geared toward performance and GPU parallelism, and not easy of use. I am aware of two efforts for such language, one from MIT and another one from Microsoft, which are Halide[1] and Neon[2] respectively. While both Halide[1] and Neon[2] are easier than using C++ or even Matlab for image processing, they still require more code than the proposed SIP language.

Proposed Language

Most image processing algorithm boil down to a convolution between a source image and a certain kernel, such kernel can be a constant matrix or a complex function. Image is a 2D array with height and width, each element in the image is called pixel that differ in size and meaning depend on the format of the image.

The primarily goal of SIP, is to be able to perform most image operation without the need to iterate on each pixel. In order to achieve such goal, the design of SIP was influenced by the concepts and techniques used in functional languages.

Reserved Keywords

For SIP, we will have the below reserved keywords that cover basic image related types, most of those types can be swizzle, and accessed element by element. Basically we will support some of the pixel operation found in pixel or fragment GPU shader.

histogram	1 dimensional histogram, the number of pixels with the same color.
image	Represent an image type, image can have from

	one to any number of named channels. Those named channels will be accessed using SIP syntax shown in next section.
fun	A function that takes a pixel fragment and return a pixel.

The above is not a comprehensive list, we will support most basic types found in other languages such as bool, float, int…etc; in addition to some imperative construct such as loop. The above list is for image specific type.

Image Operation

Let's look at some basic image operations that show how easy such operations can be done in SIP.

```
/* Comments in SIP are similar to C\C++*/

image hsvImage;                    // An HSV image format

//
// Convert from HSV to RGB, using the 'in' operator. Red, green, blue, hue, saturation, value are
//  named channel that SIP library will treat them as RGB and HSV color space, however, there is no
// restriction on them. Developers can freely add or remove named channels or put whatever value
// in the already predefined names.
//
image rgbImage = hsvImage in (red, green, blue);

//
// Manually convert each pixel from rgb to a single channel named 'l', and store the result in
// custImage. "rgbImage in rgb" means map the input image to RGB color space, which will be nop
// if rgbImage is already in RGB color space,  "rgb for (l:2*r + 4*g + b)" means that each mapped
// pixel in RGB space is provided as argument to the 'l' expression and the result for such expression
// will be stored in "custImage".
//
image custImage = rgbImage in (red, green, blue) for {l:2*red + 4*green + blue};

//
// 2 channels image "lx", comma in the expression mean a new expression for a new channel.
// In the below, we have one expression for channel l and another for channel x.
//
image  twoChannels = rgbImage in (red, green, blue) for {l:2*red + 4*green + blue,
                                                         x:2.5*red + 3*green + 0.5*blue};

//
// Threshold, SIP will support ternary operators.
//
image  threshold = rgbImage in (red, green, blue) for {l: ((red + green + blue) / 3) > 128 ? 255 : 0};

//
```

```
// Crop, just specify the range for Height and Width in pixels.
//
image croppedImage = rgbImage[3..57, 34..77];
```

Let’s move on to a more advance image processing code, histogram is one of the most computed features for most image algorithms. It is the number of pixels with the same color. In SIP, histogram is a primitive, as shown below:

```
histogram hist = hsvImage in (hue); // hist now has the histogram of
                                    // channel h.
```

Now let’s talk about convolution; convolution, in image processing, is simply for each pixel in the image we look at its value and its neighbor values, and from those values we perform some operations and return the final result.

In SIP, convolution will be presented by the operator ‘^’, as shown below:

```
image blurredImage = rgbImage ^ blurredKernel;
```

Operator ‘^’ takes image in one side and an image with a compile time size on the other side. You can’t convolve two images read from disk together.

We will have a list of predefined common image kernels; however, we will allow custom kernel implementation. Custom kernel fall into two categories:

1. Most flexible, the kernel is simply a function that takes 2D arrays of pixels and returns a color.
2. Constant, the kernel is simply 1D array or a 2D matrix –The reason of the 1D array is for kernel with separable components (i.e. Gaussian…etc).

```
//
// kernel function is the most flexible option, it takes a pixel as an input and return a pixel as a result.
// The channels in ‘p’ depend on the image in the convolution.
// What the below kernel does, it takes a pixel ‘p’ with the assumption that it has red, green and blue
// named channel (if any channel is missing or all of them, we will put zero in the missing one), then
// return an image with the same red channel, the same green channel, and a zero blue channel.
// “->(red, green, blue)” means that this kernel return the name channels provided in the parenthesis
// in the below case red, green and blue.
//
fun blur(p) -> (red, green, blue)
{
   red = p[0,0].red;
   green = p[0,0].green;
   blue = 0;
}

//
// image kernel that can be applied to all channels.
```

```
//
image blur = [[1 1 1][1 1 1][1 1 1]];

//
// Image kernel that can be applied for each of the named channel.
//
image blur = r:[[1 1 1][1 1 1][1 1 1]], g:[[1 2 1][2 2 2][1 2 1]], b:[[1 1 1][1 1 1][1 1 1]];
```

For more advance functionality, developers still have the option to go imperative.

Reading and writing image file

To read and write an image from a file, we will borrow C++ stream syntax "<<" for reading and ">>" for writing.

```
//
// Reading test.jpg.
//
image readImage << "c:\Photos\test.jpg"

//
// Writing to result.jpg.
//
resultImage >> "c:\Photos\result.jpg"
```

Error handling

As in most languages, there are two types of errors. Compile time error, in which we favor and we will try to detect most compile time error that we can. Runtime error that can happens for conversion failure, non-existing channel, and for a lot of other reasons, we will throw an exception in this case.

C inspiration

We will borrow most imperative syntax from C language, due to its familiarity. We will support if\else statement, while statement, for loop and function.

Runtime

We will compile SIP to a GPU language, some of the options that we have are: CUDA, OpenCL, OpenGL fragment, Direct3D Compute, or AMP. As the project progress, we will decide which one of the above is more suitable.

References

1. http://halide-lang.org/
2. http://research.microsoft.com/apps/pubs/default.aspx?id=144767