CLAM: The Concise Linear Algebra Manipulation Language

Jeremy Andrus and Robert Martin and Kevin Sun and Yongxu Zhang
{jca2119, rdm2128, kfs2110, yz2419}@columbia.edu

October 5, 2011

Language Proposal

CLAM is a linear algebra manipulation language specifically targeted for image processing. It provides an efficient way to express complex image manipulation algorithms through compact matrix operations. Traditional image processing is performed using a language such as C, or C++. Algorithms in these languages are quite complex and error-prone due to the large number of lines of code required to implement something as conceptually simple as, "make this image blurry." The complexity arises from the need to perform elaborate calculations on every pixel in an image. For example, to blur an image you first need to calculate the luminance of the pixel (from the red, green, and blue channels), then you need to mathematically combine this with the luminance of adjacent pixels, and finally re-calculate red, green, and blue values for an output image.

CLAM will simplify image processing, and more generally linear algebra, through domain-specific data types and operators. The basic data type in CLAM is a `Matrix`. Matrices can be manipulated by operators that perform functions such as matrix multiplication, or rotation. An `Image` is another CLAM data type which is expressed as a collection of matrices, or channels. For example, when reading an image into memory, CLAM creates a *Red*, *Green*, and *Blue* channel automatically. Additional Image channels can either be assigned, or calculated using an expression syntax which defines a calculation involving the values of other, previously defined, channels. The basic image processing operator in CLAM is the convolution operator. This operator takes a channel and a `Kernel`, another basic data type, and outputs an `Image`. This operator convolves each `Matrix` within the `Kernel` with the input channel, and collects the resulting output channels into an `Image`.

Two primary use cases of CLAM are basic image information extraction, and filtering. The compact syntax and powerful basic data types of CLAM will make information extraction, such as finding all the edges in an image, simple, compact, and easy to read.

Features

CLAM uses implicit loops, i.e. there is no explicit looping construct in the language. Loops are implicitly defined by per-pixel matrix or convolution operations. Additionally, CLAM automatically determines or calculates image and matrix dimensions. There is no need to explicitly size these data types. This further reduces complexity, and eliminates frequent mistakes such as going beyond array bounds in a calculation.

Example Syntax

The goal of the CLAM syntax will be to make conceptually simple image manipulations into simple language constructs. For instance, convolutions make frequent use of constant matrices, so our language will provide a simple way to specify them, such as:

```
Matrix sobelGy := { +1 +2 +1 |  0  0  0 | -1 -2 -1 };
```

Another common image processing technique is performing the same calculation on every pixel in the image. An example of this is calculating the luminance of a pixel from the red, green, and blue channels. CLAM makes this calculation simple and compact by defining an additional image channel. Assuming there exists an instance of a `Image` variable named, *myimg*, a channel can be added to the image with:

```
Int32 myimg:Luminance := #[ (3*Red + 6*Green + 1*Blue) / 10 ];
```

Where the expression within `#[ ...]` is evaluated once for every pixel in the `Image`. The *Red*, *Green*, and *Blue* variables correspond to previously defined channels in *myimg*, and their values during expression evaluation will be the value of the corresponding channel at the current pixel location.

Image processing also frequently involves describing a series of operations that should be carried out for each pixel, and then repeating it for every pixel in an image. CLAM makes it simple to describe this process through the `Kernel` data type and the convolution operator. Here is an example of how one might perform a Sobel edge detector in the CLAM language:

```
1  // read an image into the 'srcimg' variable
2  Image srcimg = imgread("someimage.jpg");
3
4  // define a luninance channel for this image
5  // (Red, Green, and Blue channels are implicit from imgread)
6  Uint8 srcimg:Lum := #[(3*Red + 6*Green + 1*Blue)/10];
7
8  // Kernel definitions are ordered i.e. the channels
9  // are calculated in the order they are defined
10 Kernel sobel = {
11         Uint8 @Gx    := [1 | 1]{ -1  0 +1 | -2  0 +2 | -1  0 +1 };
12         Uint8 @Gy    := [1 | 1]{+1 +2 +1 |  0  0  0 | -1 -2 -1 };
13         Uint8 G      := #[sqrt(Gx*Gx + Gy*Gy)];
14         Angle Theta  := #[arctan(Gy/Gx)];
15 };
16
17 // Convolution - resulting image will have the same number
18 // of channels as the filtering kernel.
19 Image edges = $(srcimg:Lum) ** $(sobel);
20
21 // compose an output image which is a grayscale of
22 // edge gradient magnitude
23 Image output = {
24         Uint8 Red   = $(edges:G);
25         Uint8 Green = $(edges:G);
26         Uint8 Blue  = $(edges:G);
27 };
28 imgwrite( output, "edges_of_someimage.jpg");
```