Zef : A Computer Vision Library for Haskell

Saumitro Dasgupta
saumitro@stanford.edu
Department of Computer Science
Stanford University

Abstract

We present a computer vision library for Haskell and discuss its design and architecture. The library is then used for developing two demo applications. Two case studies analyze these demos and show that Haskell is well-suited for implementing vision algorithms, with significant gains in productivity, readability, correctness, and conciseness. We demonstrate that by keeping a limited set of performance-critical operations in C/C++, the loss in performance is insignificant. To further support this claim, we use our library to implement a real-time video processing application on a mobile platform.

1 Introduction

Computer vision applications often involve performing computationally intensive tasks in real-time. As a result, most production-grade vision applications are written in C and/or C++. These languages allow the developer to perform low-level optimizations. It is not uncommon to find performance-critical sections optimized using inline assembly. Such optimizations would be infeasible in a higher level language such as Haskell.

However, in our experience, only a small set of operations require the degree of optimization afforded by these languages. Most of the remaining tasks can be implemented more elegantly using a higher level language. Haskell, in particular, is well-suited for this task. Below, we enumerate some of the reasons we believe this to be the case (in no particular order):

- Many computer vision algorithms can be succinctly expressed using Haskell's mathematical syntax.
- Functional abstractions are well-suited for expressing many common vision operations.
- The generated code is efficient enough for real-time vision applications.
- It allows for interoperability with C. This allows us to tap into the rich ecosystem of existing computer vision code.

In the following sections, we will describe our implementation of a computer vision library for Haskell, called Zef. Prior work in this area includes FVision by Peterson, et al. [5]. While FVision was primarily intended for visual tracking purposes, Zef intends to be a general purpose vision library.

2 Library Architecture

2.1 Internal Matrix Representation

Matrices are one of the core data structures of computer vision. They are commonly used for representing images, systems of linear equations, camera parameters, and other numerical data. Zef's internal matrix type is essentially a wrapper on top of

OpenCV's `CvMat` data type. There are two distinct advantages which lead to this choice:

- It allows for seamless interoperability with OpenCV's API.
  While OpenCV's modern API builds on top of the newer `Mat` C++ class, a significant portion of it is exposed via the C API which uses the `CvMat` data type. As a result, it is usually possible to directly call these C functions using Haskell's FFI. Internally in OpenCV, the conversion to an instance of the `Mat` class incurs a negligible cost as no data is coped - only a lightweight "header" is created.

- The matrix data is stored as a dense stride-based array. That is, for an n-dimensional matrix M, the address of the element $e = M[i_1, i_2, ..., i_n]$ is given by:

$$\texttt{addr}[e] = B + \sum_{j=1}^{n} S[j] \cdot i_j \qquad (1)$$

  where B is the base address of the data, and $S[j]$ is the stride for the j^{th} element. This is particularly useful because it opens up the possibility for interoperability with a large number of libraries which also use this format. For instance, Apple's Graphics and Media frameworks are compatible with this representation. As a result, it becomes trivial to work with images acquired from an iPhone's camera. Other examples include the popular library NumPy (and by extension, SciPy).

A pointer an OpenCV `CvMat` struct is represented in Zef by:

```
type PCvMat = Ptr CvMat
```

where `CvMat` is an empty data declaration (enabled by the `EmptyDataDecls` extension) representing the opaque OpenCV `CvMat` struct.

2.2 Image Types

For any image processing operation, the input and output images can vary in:

- The number of channels. (single channel, three channel)
- The numerical data type. (byte, single precision floating, etc)
- The color model. (RGB, HSV, etc)

For some operations, these distinctions are irrelevant. For instance, the addition operation is well defined for any two images of the same type. For others, only a particular configuration is permissible. As an example, consider the dilation morphological operation - it is only defined for binary and grayscale images. Furthermore, for some operations the return type is the same as the input, while for others this is not the case.

In OpenCV-based applications written in C++, these distinctions are not reflected in the type signatures. The multi-purpose `Mat` class is used for all types, with an integer mask representing the image type parameters. Using an invalid image type results in a runtime exception. For Zef, however, we make a clear distinction. We have an `Image` type-class, which is defined as follows:

```
newtype ImageData = ImageData { unImageData
    :: ForeignPtr CvMat } deriving (Show)

class Image a where
    toImageData     :: a -> ImageData
    fromImageData   :: ImageData -> a
```

Distinct image types, such as `RGBImage` and `GrayImage`, are then defined as follows:

```
-- |A matrix containing 3 channel RGB image.
newtype RGBImage = RGBImage { unRGBImage ::
    ImageData }
instance Image RGBImage where
    toImageData     = unRGBImage
    fromImageData   = RGBImage
```

We can now clearly specify the type signatures for image operations:

```
add :: Image a => a -> a -> a

blendImagePyr :: RGBImage -> GrayImage ->
    [RGBImage]
```

This makes code written using Zef less prone to mismatched type errors, and more readable.

2.3 Operator Syntax

Zef's mathematical syntax is inspired by that of `MATLAB`, which is commonly used for prototyping in the vision community. In particular, element-wise matrix operations are denoted using the "dot-prefixed" syntax. The following code snippet demonstrates this for two matrices/images `X` and `Y` of the same type:

```
P = X.+Y -- element-wise summation
Q = X.*Y -- element-wise product
```

2.4 Extensibility

The basic implementation of Zef includes a minimal set of low-level operations imported from OpenCV. It is not intended to be a full port of OpenCV's API to Haskell. However, it is likely that developers would require some of the more specialized functionality provided by OpenCV that is not included in Zef's core library. To this end, Zef includes a set of helper functions to make it easy to work with foreign code in idiomatic Haskell.

3 Matrix Fusion

3.1 Overview

Consider the following image processing function taken from the exposure fusion demo described in section 4.1:

```
contrast :: GrayImage -> GrayImage
contrast = Img.abs . Img.laplacian
```

In a naive implementation, the function composition above would result in two matrix allocations: one for the result of the Laplacian, and another for the result of the absolute value. However, since the intermediate results aren't required, we could reduce this to a single matrix allocation.

Inspired by the stream fusion feature implemented in the Text library[3], Zef avoids these unnecessary intermediate allocations using an optimizations it refers to as "matrix fusion" – composed functions only result in a single intermediate matrix allocation.

This feature is implemented using Haskell's rewrite rules. Consider the `abs` function, defined as follows:

```
performUnaryOp :: Image a => UnaryImageOp
    -> a -> a
performUnaryOp f src = uncascade $
    transformImage (cascade src) f
{-# INLINE performUnaryOp #-}

abs :: Image a => a -> a
abs = F.performUnaryOp c_zef_abs
{-# INLINE abs #-}
```

The `laplacian` function is similarly defined. Now, when these two inlined functions are composed, it results in the following chain:

```
uncascade $ transformImage (cascade
    (uncascade $ transformImage (cascade
    src) f1)) f2
```

We now define the rewrite rule:

```
{-# RULES "cascade/uncascade fusion" forall
    c. cascade (uncascade c) = c #-}
```

thereby reducing the chain to:

```
uncascade $ transformImage c f2
```

where `c` is the intermediate cascade produced by the first transformation. As we will show in the next section, only an intermediate cascade results in an allocation. Since these are re-used for composed chains, we effectively avoid unnecessary allocations.

3.2 Cascades

Fusion-enabled functions operate on instances of the data type `Cascade`, which is defined as follows:

```
data (Image a) => Cascade a =
    NascentCascade  a | BufferedCascade a
```

The `cascade` function we encountered earlier always returns a NascentCascade instance. The image field is just the input image. On the other hand, intermediate values are always of the type BufferedCascade, and the image field is an internal "buffer" matrix. A fusion-aware image transformation will check if a given Cascade is a BufferedCascade. If this is the case, it is re-used. Otherwise, one is constructed.

Note that this fusion process also works for binary operations, such as matrix addition. In this case, if either one of the operands is buffered, it's re-used.

When tested on the demo described in section 4.1, enabling matrix fusion resulted in 42 fewer matrix allocations.

4 Case Studies

In this section, we analyze two demos implementing actual computer vision applications using Zef. The source code for both of these demos can be found in the git repository referenced in Appendix A.

4.1 Exposure Fusion

4.1.1 Overview

The process of exposure fusion involves combining multiple images with varying exposures to generate a single perceptually enhanced image. For this demo, we implement the algorithm described by Mertens et al. in [4]. An example input sequence and the resulting output generated by our app is provided in figure 1.

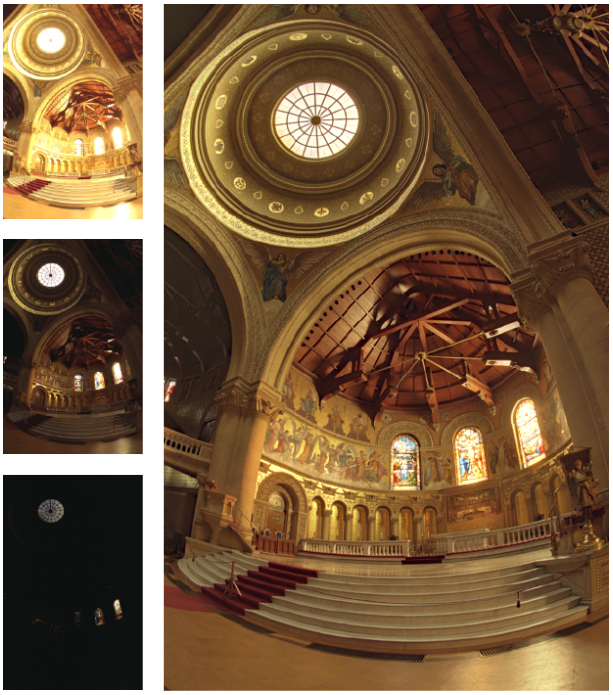

Figure 1: Exposure fusion. The three images on the left are the input sequence (only 3 of 7 are shown). The image on the right shows the output generated by our demo.

4.1.2 Observations

The implementation is extremely concise, without sacrificing readability. This can be attributed to the fact that function compositions, maps, and other common mathematical abstractions can be concisely expressed in Haskell. Indeed, parts of the implementation read very much like the original definitions in the paper [4] (an example of this would be the contrast function, shown in 3.1).

The explicit type signatures make the image processing operations easier to parse. For instance, it is immediately evident from the type signatures that the `saturation` function operates on the full color RGB image, whereas the `contrast` function only cares about the grayscale values.

It is also interesting to note that there are quite a few `map` calls that operate on a sequence of images. These are all parallelizable. A naive optimization where these map calls were replaced by `parMap` from the parallel package resulted in performance improvements (further details are provided in the performance section). Such parallelization would have been significantly more time consuming and error prone in C++.

4.1.3 Performance

We tested our implementation on a version of the Memorial Church dataset published by Paul Debevec in [2]. It consists of seven 472×728 RGB PNG images. The running time was measured using the standard Unix `time` utility. In addition to our Haskell implementation, we also evaluated the same algorithm implemented in C++. The running times are listed in table 2.

Also listed are the times for the parallelized version described in the previous section. These were

Implementation	Real	User	System
Haskell	0.36	0.25	0.12
Haskell (Parallel)	0.25	0.31	0.16
C++	0.25	0.23	0.02

Figure 2: Running times for different implementations of the exposure fusion demo.

obtained by using the `+RTS -N4` flags.

All tests were performed on a mid-2012 MacBook Pro with a 2.6 GHz Intel Core i7 processor.

4.2 Mean-Shift Based Tracking

4.2.1 Overview

Our exposure fusion demo demonstrated the viability of using Zef and Haskell for implementing vision based algorithms. However, it is unlikely that production grade vision systems could be implemented solely using Haskell. Consider, for instance, a mobile vision app running on an iPhone. It is likely that such a system would be written at least partially in Objective-C. We explore this scenario in this case study by implementing a hybrid vision system — an iPhone app that calls into Haskell from Objective-C.

Mean-shift is a non-parametric feature space based method that can be used for visual tracking purposes [1]. It can be elegantly expressed using Zef in Haskell. On the other hand, tasks such as image acquisition and rendering are better handled in Objective-C. Therefore, we export the following function from Haskell-land:

```
meanShift :: Ptr CvMat -> Ptr Rect -> IO ()
```

This function accepts a pointer to an image (representing the back projection of the object histogram) and a pointer to a `rect` struct that is initially set to match the object to be tracked. On return, the `rect` struct is updated to the estimated position of the tracked object. This function is called once per frame from Objective-C.

Figure 3 shows a sequence of frames captured on an iPhone running the tracker demo. The green tracking rectangle marking the soccer player is the one described by the `rect` struct mentioned above.

Figure 3: Mean-shift based tracker running on an iPhone. These screen captures show the same soccer player being tracked across the field. The green tracking rectangle marking the player is the one returned by our exported Haskell function.

4.2.2 Performance

We tested our demo on an iPhone 5 (1.3 GHz dual core ARMv7 based Apple A6) as well as an iPad mini (1.3 GHz dual core ARMv8-A based Apple A7).

The input video was an H.264/MPEG-4 AVC compressed video file of a soccer match, with a resolution of 854×480.

On iPhone 5, the demo ran at approximately 23 frames per second, nearly matching the frame rate of the input video. On the iPad, the demo ran at approximately 50 frames per second — 2x real-time.

5 Conclusion

We demonstrated that our library, Zef, can be used for rapidly and elegantly developing computer vision algorithms. By retaining the low-level performance critical sections in C/C++, the performance trade-off was found to be negligible. In contrast, the gains in productivity, readability, safety, and conciseness were significant.

We demonstrated that Zef was suitable for each of these use cases:

- Developing standalone vision applications.

- Developing a vision library for use within a hybrid-language application.
- Developing real-time vision applications for mobile platforms.

With continued refinement, we believe that Zef will prove to be a very compelling library for developing computer vision applications.

6 Acknowledgments

We would like to thank Stanford's CS240H instructors for their great course material — David Mazières, Bryan O'Sullivan, David Terei, and Edward Yang.

A Source Code

The complete source code for Zef, including all the case studies described in this paper, can be found at https://github.com/ethereon/Zef.

References

[1] Comaniciu, D., and Meer, P. Mean shift: A robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 24*, 5 (2002), 603–619.

[2] Debevec, P., and Malik, J. Recovering high dynamic range radiance maps from photographs. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques (SIGGRAPH '97)* (1998), ACM Press/Addison-Wesley Publishing Co., pp. 369–378.

[3] Harper, T. Stream fusion on haskell unicode strings. In *Implementation and Application of Functional Languages*. Springer, 2011, pp. 125–140.

[4] Mertens, T., Kautz, J., and Van Reeth, F. Exposure fusion. In *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on* (2007), IEEE, pp. 382–390.

[5] Peterson, J., Hudak, P., Reid, A., and Hager, G. Fvision: A declarative language for visual tracking. In *Practical Aspects of Declarative Languages*. Springer, 2001, pp. 304–321.