HIPHOP (Handy Image Processing for Highly Over-caffeinated Programmers): A DSL for Image Processing

Kristen Kwong
12557154
kristenkwong@gmail.com

Kalli Leung
29615151
kalli417leung@gmail.com

Cindy (Yu-Hsin) Tu
19677153
cindytu1535@gmail.com

Chrysen Park
23749154
chrysen777@gmail.com

ABSTRACT

Image processing has widespread applications, ranging from signal analysis in medical imaging, 3D rendering and post-proc in video games, to pre-processing steps for various computer vision and machine learning methods in academia [8]. Currently, existing domain specific languages (DSLs) provide programmers the ability to optimize performance for image processing pipelines for different architectures and languages [2], and existing image libraries [3,5,6] provide users with functions to perform image transformations and build image processing pipelines. However, for the average user whose goal is to perform image transformations with file operations (seek, read, file pointers, etc.) abstracted away in a general purpose macro language, they may find existing DSLs and libraries to be wanting.

In the following paper, we illustrate the steps to construct a proof of concept for our image DSL, as well as the minimum set of goals attained in completing the implementation and future goals for subsequent iterations of hiphop-lang.

KEYWORDS

Domain specific languages, image processing

1 Overview of Image Processing

Image processing is the procedure of converting an image into digital form and applying transformation operations on it [15]. Its applications in the field of computer vision range from medical image processing and video game rendering in industry to real time computer visioning in academia.

Currently two main approaches exist for image processing tools: image processing DSLs, and image processing libraries imported as modules into various programming languages. In our project, we will focus on building a domain specific language with a straightforward and simple syntax, which will allow programmers to invoke common algorithms to manipulate images. These algorithms will be implemented through the utilization of Python's image processing libraries.

1.1 Applications

Before discussing approaches for programmatic image processing, it is important to first delineate the usage and value of image processing, so that we may better understand tasks that users may wish to perform. We will also discuss high level steps that our project will take to provide these functionalities in hiphop-lang.

There are three main steps in image processing. First, the image is imported using optical scanners, or created using graphic manipulation and saved [16]. Second, the image can be analyzed and manipulated; this can include improvements and enhancements on the image, or gathering data that is difficult for the human eye to see [16]. Lastly, the results are output, which could include the image itself and/or any data gathered from its analysis [16].

We will now take a look at some more specific application areas of image processing techniques.

Biomedical Image Enhancement and Analysis The goal of image processing within this area corresponds to image enhancements of medical imagery such as MRI (Magnetic Resonance Imaging) or CT (Computed Tomography) scans [16]. These images are composed of individual pixels corresponding to color or luminosity values and can be enhanced with techniques such as increasing contrast or clustering similar-valued pixels to be more useful to medical professionals for diagnosis and treatment [16].

Signature Recognition Handwriting and signatures can be inputted as images, then sharpened and smoothed in order to be compared with other copies to verify a person's identity [16].

Transformations, such as rotations and scales, can also be used before comparison to reduce noise in the analysis.

Image Restoration For those working with photographs, there may be cases where the live conditions require post-processing after the photograph is taken in order to enhance features. For example, when pictures are taken underwater, colour manipulation may be necessary to offset lighting conditions [16]. Edges can also be preserved, and noise reduced.

Computer Vision A growing field in computer science is the ability for artificial systems acquiring knowledge from images. With complicated algorithms and lots of data, programs are able to detect objects within images by looking at per pixel data, as well as relationships between pixels, and comparing to other images. Thus, image processing is necessary to manipulate and extract subsections of images for various algorithms [16].

Although many more areas exist where image processing tasks are essential to work and research, these examples demonstrate the importance of the ability to perform such tasks easily and efficiently. Next, we will discuss some existing solutions that have been researched to do so.

1.2 Existing Image Processing DSLs

In the following section, we briefly examine four existing domain specific languages for image processing, to summarize general areas of existing work and discuss some areas that may be lacking for its users.

Halide is a DSL for image processing based on Clang/LLVM, using a functional programming paradigm. Code can be generated with Halide for various architectures, but this process is not entirely automatic - the user must still specify a schedule for the mapping to target architecture [4]. Thus, a developer must have a knowledge of both computer architecture and the graphics domain to make efficient use of Halide.

IPOL (Image Processing Operator Language) is a DSL for image processing applications that presents a holistic domain specific system description language using a hardware and software design [10]. However, programming in IPOL requires the user to interact with various components across all the layers, including sensors, displays, execution units, and software [10] – for those who are not well-versed in tuning such algorithms at such a low level, using IPOL would be a very difficult task.

HIPA^{CC} (Heterogeneous Image Processing Acceleration) is a DSL and source-to-source compiler for image processing. With it, users can design image processing kernels and algorithms, and algorithms can be mapped to a large spectrum of GPU target architectures and exploit the different types of parallelism available [14]. However, programming in HIPA^{CC} requires knowledge of matrix operations to manipulate images. For example, implementing a smoothing Gaussian filter requires the implementation of the summation and calculation of a Gaussian mask for individual pixels [14], which would be challenging for programmers who are looking simply to apply a “blur” to their image.

Chipotle, which is heavily inspired by HIPA^{CC}, focusses on high-performance image processing, generating code from an algorithm specification, and can also be extended to provide a schedule employing GPUs and CPUs together. Features of Chipotle gives Chipotle the ability to use Lisp macros and feature-oriented programming, as well as an improvement in performance over HIPA^{CC}. However, much like the aforementioned languages, even the task of describing a filter algorithm requires the usage of edges in a graph structure to specify the manipulation to the pixel level [12].

Much of the existing work that has been done is focussed heavily on improving the performance of graphics processing through algorithms or architectures, or alternatively is based on efficient compilation for niche hardware architectures. Many of these languages also require the implementation of complex mathematical linear algebra operations to manipulate pixels within the image, or knowledge of computer architecture in order for programs to be compiled efficiently for various hardware – a step that those seeking to perform simple image manipulation may not have the necessary background to do.

1.3 Motivation

With the proliferation of image processing applications in fields outside of computer science (i.e. biomedical imaging, digital media and design), the user demographic of image processing methods has likewise expanded beyond the average programmer. Existing image processing DSLs provide fine-tuned hardware and compiler level performance optimizations, albeit with low learnability and accessibility for programmers without an extensive background in graphics processing architectures. Consequently, novice programmers or non-programmers may find existing image DSL alternatives to be unsuitable for their needs.

Our DSL, HIPHOP, aims to provide a subset of functionality to enhance and manipulate images without well defined output formats, from within the three main stages of image processing (generating input from analog or digital sources into digital images, analyze and transform images, and output data from

previous stages) [16]. By using simple syntax for file operations (users specify a filename and an identifier to save an image for further processing), hiphop-lang provides users ways to perform file operations with high efficiency and learnability. We also aim to provide built-in functions (i.e. blur, grayscale, rotate, reflect) that allow for straightforward manipulation of images. Lastly, the user should be able to save any images manipulated by specifying its identifier and a filename to write to. Thus, our DSL abstracts away file I/O and transformation geometry context specific knowledge.

Overall, our DSL provides an interface for users of varying levels of expertise to build image processing pipelines by allowing amateurs to work without concerning themselves with file operations or complex knowledge of graphics algorithms, while still providing the more well-versed aficionados with the flexibility to customize image pipelines for functionality and performance using optional syntax extensions to the basics of the HIPHOP language.

2 Final Project

2.1 Core Goals

The minimum "low-risk" set of goals that we intended to complete in our DSL include:

- consuming an image input
- applying image transformation operations using the DSL on image input data
- setting up custom tasks by writing functions for sequences of transformation operations

Users should be able to use hiphop-lang to do all the above without having to write code involving file operations, as this is abstracted away.

To consume images as input data for manipulation, we relied on using native Python file open/close and read/write operations.

In order to apply image transformations, we took some input in the DSL, applied lexical analysis (see section below, Proof of Concept), and translated this into corresponding calls in Python imaging libraries such as OpenCV [3]. These transforms are chained to create image processing pipelines, which in turn corresponds to chained function calls to an image processing library. To circumscribe this goal within a "simplest suitable" approach, we limited the amount of requisite function transforms to 2-3 (chosen from rudimentary transforms such as colour/contrast manipulations, blur, and simple filters) in our core goals while still being able to demonstrate the functionality to chain different image operations together. This was successfully accomplished with implementing scaling, cropping, blurring functions and additional transforms that were intended as part of the reach goals.

Custom tasks for sequences of operations and macros provide a natural mapping for instructions to apply transforms on images. The former is implemented in hiphop-lang through the use of named functions (that can be applied) within the DSL grammar, which is then translated into a corresponding series of image processing library function calls.

Generally, we strived to support a minimal set of core functionalities that demonstrated the capabilities of hiphop-lang and its value as an image processing DSL while keeping the implementation of each tractable, all of which were fully completed in our DSL.

2.2 Implementation Details

Overview The final project implemented contains modules and functions for the following:

- parsing a multi-line HIPHOP function according to the HIPHOP EBNF specification
- creating objects to represent each expression, similar to a define-type for the intermediate value used in interpretation, whose state can be modified as the program continues
- evaluating each expression object
- core functions for essential file operations, such as opening images from files, as well as saving modified images back to files
- image processing functions to process an image can be loaded from a previously saved identifier, and modified "in-memory"

Installation HIPHOP is built with Python3, using OpenCV modules. To install these, perform the following steps:

1. Install Python3 via https://www.python.org/downloads/
2. Install OpenCV using the command `` `pip3 install opencv-python` `` after Python3 has been installed
3. Install colorama using the command `` `pip3 install colorama` `` and termcolor using `` `pip3 install termcolor` ``.
4. Install numpy using the command `` `pip3 install numpy` ``.
5. Ensure that git is installed, following instructions on https://www.atlassian.com/git/tutorials/install-git.
6. Clone the repository for hiphop-lang by using the command:
   `` `git clone ``

`https://github.com/kristenkwong/hiphop-lang.git`

Running the program The program can be run in both command line mode, or by parsing in a hiphop program saved into a file. For parsing in a file, run using the Python command: `python3 main.py <hiphop-filename>`. For the HIPHOP command line, run using the Python command: `python3 main.py`.

EBNF

```
<HHE> ::= open <filename> as <id>
        | save <id> as <filename>
        | apply <func> to <id>
        | apply-all [<funcs>] to <id>
        | save-macro [<funcs>] as <id>

<filename> ::= "<literal>"

<funcs> ::= <func>
        | <func>, <funcs>

<func> ::= <id> <nums>

<literal> ::= STRING

<nums> ::=
         | <num> <nums>
```

A <HHE> (pronounced "hee-hee"*) refers to a HIPHOP expression, which corresponds to an action within the program, which either performs file operations, changes the state of objects within the program, or performs a core function to an object. For a test program of expression, see next section.

* in a Michael Jackson voice, preferably

<img> refers to functions that return the representation of an image within the program. Although not currently implemented within the POC, there should be a way for the user to show an image with simply using its identifier, or after an application of a function.

<filename> is a string enclosed in quotations to refer to a path to open containing an image, and further on, image folders as well.

<func> refers to a function name, with a variable number of integer arguments <nums>, which will be input into various image processing algorithms.

Example HIPHOP Program Follows is a simple example of a HIPHOP program that opens the specified file, performs the grayscale filter, and saves the file with the given filename.

```
open "test-image.jpg" as img
apply grayscale to img
save img as "test-output.jpg"
```

This program can be successfully parsed and run using our program, and performs the given task of applying the filter and saving a new output. As one can see, this is a very simple syntax that is easy to understand and memorize for the average user, and we hope to continue this for the functionality in our final project.

Here is a slightly more nuanced example:

```
open "test-image.jpg" as img
save-macro [blur 10, grayscale] as blurscale
apply blurscale to img
save img as "test-output_macro.jpg"
open "testimage2.jpg" as img2
apply blurscale to img2
save img2 as "test-output_macro2.jpg"
```

This example makes use of the save-macro function to save a chain of blur and grayscale functions to the name blurscale. It then applies this to two different images.

Modules This section will describe the modules used to build the basic functionality of the HIPHOP DSL.

Main.py The main function checks to make sure the Python program has been called with the correct arguments. It then calls parse with this filename.

HiphopParser.py The file is opened in read mode, and each line of the HIPHOP program is parsed into instances of the HIPHOP expression types, which are implemented as classes. The parse function splits each line into tokens, and based on checking the specific syntax in the tokens, decides which type to instantiate. It supports apply (`apply grayscale to img`), open (`open "test-image.jpg" as img`), save (`save img as "test-output.jpg"`), save-macro (`save-macro [blur 10, grayscale] as blurscale`), and apply-all (`apply-all [blur 10, grayscale] to img`) statements. As an improvement on the proof-of-concept, `apply` was modified to allow the application of saved macros. For each statement "case", a function is called to check whether the syntax is correct - if so, the correct type object is returned; if not, a `hiphop_error` instance is returned and then printed, followed

by a system exit. If a correct type object is returned, the object is then asked to evaluate the given expression.

HiphopTypes.py This file contains the classes used to represent expressions in our language. Our type-syntax checker (such as `is_open_expr`) is implemented using regular expressions, which attempt to parse each expression and extract the corresponding variables needed for the expression types. As stated earlier, these functions return an instance of either an error or a type object, such as `open_expr`.

When evaluate is called on the instance of an object, the expression is evaluated using the state currently stored in its fields. Evaluation for application functions corresponds to calls to image processing functions implemented in *core.py*, while file operations are also calls to file operations in the same file.

Core.py This module contains all of the functions used to carry out tasks for our DSL. This includes file operations, such as opening and saving files corresponding to identifiers within the program execution, as well as image processing operations, such as grayscale, which is currently implemented. This was also extended from the proof of concept to include the saving and retrieval of macros - the environment is updated to include any macros defined using the `save-macro` calls, and any `apply` functions will look at both built-in functions and saved macros before throwing errors for functions not defined.

RunEnv.py This file will contain global variables that will be used and modified through program execution from different modules. This is where the map of identifiers to opencv images and also the map of identifiers to macros are stored.

OpenCV In order to write image processing functions for evaluating HIPHOP expressions, we used the OpenCV module imported into Python. OpenCV is an open-source library that contains several computer vision algorithms, which allows us to more easily implement filters, transformations, and more. For example, OpenCV is used to implement a grayscale function with the following code snippet:

```
img = saved_vars.get_var(id)
gray_image = cv2.cvtColor(img,
cv2.COLOR_BGR2GRAY)
```

This code can be found in the *core.py* module of the hiphop-lang project. We have also implemented more functions in the final project.

2.3 Reach Goals

Besides the core goals proposed earlier, we previously outlined a set of higher-risk "reach" goals and an implementation plan for each including: providing additional transformation operations (and extending existing methods with optional parameters for fine-tuning), adding supplementary file operations to work with file paths and directories, including modules that interoperate with scikit-learn and other machine learning methods.

2.3.1 Reach Goals Implemented

Of the reach goals proposed above (and beyond), the following were successfully implemented:

- Adding additional image transformation operations (morphological transforms, colour masks & filters).
- Allowing saving and writing to file paths and directories in the supported file operations.
- Adding an interactive hiphop-lang terminal that allows users to enter commands line by line.

Additional geometric and morphological transforms from with methods from OpenCV or scikit-image [3, 5] were exposed so they can be applied when building image processing pipelines, thereby increasing the utility of our DSL. To implement these additional transforms, additional steps for lexical analysis (new keywords) were added, which will then call functions to the OpenCV or scikit-image library to apply the transform. Morphological transformations such as erosion, dilation, morphological gradients were added, with the potential to clean up noise in images as a pre-processing step in many image processing pipelines.

Supplementary file operations have been added to hiphop-lang; the language syntax has been extended to allow saving processed images to file paths or custom named directories. This application may be useful for running multiple pipelines on the same output, while allowing users the convenience of distinguishing the result of each pipeline by simply inspecting directory names or placing each in different locations. File paths in operations were supported by making use of Python's native file functions and path module. If the user attempts to save images to a new directories or file paths, hiphop-lang handles this by creating the directories for the user and saving the processed images there.

A command line interface (CLI) was implemented, allowing users to use the HIPHOP terminal to write commands in line by line. By modifying the main module of hiphop-lang to support a terminal mode and adding extra lexical analysis and sanitization of terminal inputs, the CLI was successfully created.

2.3.1 Plan for Remaining Reach Goals

Currently, optional parameters as extensions to the basic image transformation operations in the set of minimum core goals, are not provided. To achieve this and allow users more functionality by means of customization - users can further fine-tune parameters such as filters or convolutions, the parsing of expressions would need to change to account for the additional number of parameters, and this would be provided to the user by way of named arguments (similar to Scala) or flags followed by additional arguments. Much like some customizable functions in OpenCV, when custom parameters are not provided, the image transformation function would be called with a default parameter. For example, if left unspecified, scaling operations will default to using bilinear interpolation [3]. Furthermore, a set of expressions that correspond to the above described customizations (e.g. consume a filter mode as an additional parameter, or consume user-inputted matrices as custom convolutions & filters) would have to be updated in the EBNF. In general, by updating the EBNF and parse method, and respective evaluate methods to account for additional parameters (e.g. linear/bilinear filters, Sobel filters, custom convolutions for edge detection algorithms), customizability and efficiency of hiphop-lang would have been greatly improved.

At the moment, hiphop-lang does not consume directory names for batch processing as previously proposed: while a worthy use case, the implementation of this feature would require iteration over a directory, checking the file type of each file in the directory, and then apply the specified image transformations over each image file in the directory. As well, symlink and hardlink entries on certain file systems (e.g. *nix, MacOS) may need special handling (or we can also make the design decision to support only image files for the sake of simplicity).

Although unimplemented in the present version of hiphop-lang, exploiting the semantic interoperability between sci-kit image library functions and sci-kit learn's machine learning methods [5, 13] was previously proposed and would allow users the ability to call machine learning methods powered by highly performant algorithms under the hood (i.e. Canny edge detection algorithm or watershed algorithm to segment objects in images) [13]. While sci-kit learn and sci-kit image are natively compatible [13] and don't require extensive changes in terms of formatting image input data, the addition of sci-kit learn would require the installation and upkeep of this module, and some boilerplate code to be added in order for calls to sci-kit learn methods to be accessible. However, as this goal addresses a need that only expert users may have and requires a lot of busywork with setting up sci-kit learn boilerplate code, it was ranked with lower priority than other goals.

3 Conclusion

Although image processing DSLs currently exist, most are focussed on the hardware optimization aspects [2] or are based on esoteric computer architectures familiar only to scholars with extensive computer graphics or architecture knowledge. Thus, to fulfill the need for non-experts, such as medical professionals and other programmers, to be able to build image processing pipelines for tasks ranging from bioimaging [7] to gaming [8]. In the end, we were able to build a functional DSL that supports simple transforms (ex. scaling, cropping, blurring, grayscale), macros, applying multiple image functions at once to an image, along with some more advanced imaging processing techniques (ex. colour filters, erosion, dilation, and other morphological transforms to denoise and preprocess images) into hiphop-lang.

REFERENCES

[1] Robert Stewart, Deepayan Bhowmik, Greg Michaelson, and Andrew Wallace. 2015. RIPL: An Efficient Image Processing DSL for FGPAs. DOI: http://dx.doi.org/10.1017/S0956707006296.

[2] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. DOI: http://dx.doi.org/10.1145/2967938.2967969.

[3] Alexander Mordvintsev. 2013. OpenCV-Python Tutorials. Retrieved October 27, 2019 from https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html.

[4] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN* 48, 6 (Jun. 2013), 519-530. DOI: http://dx.doi.org/10.1145/2499370.2462176.

[5] Emmanuelle Gouillart. 2019. Scikit-image: Image Processing. Retrieved October 15, 2019 from https://scipy-lectures.org/packages/scikit-image/index.html.

[6] Fredrik Lundh and Alex Clark. 2019. Pillow. Retrieved October 27, 2019 from https://pillow.readthedocs.io/en/stable/?badge=latest. DOI: http://dx.doi.org/10.5281/zenodo.44297.

[7] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Generating Device-specific GPU code for Local Operators in Medical Imaging. Retrieved October 27, 2019 from

https://www12.cs.fau.de/publications/membarth/membarth2012gdg.pdf

[8] André W.B. Furtado and André L.M. Santos. Using Domain-Specific Modeling towards Computer Games Development Industrialization. Retrieved October 27, 2019 from https://pdfs.semanticscholar.org/3b86/480e2a2e4de0af1729ae2965895e2463b69e.pdf

[9] Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. Prototyping Real-Time Vision Systems: An Experiment in DSL Design. Retrieved October 27, 2019 from https://dl.acm.org/citation.cfm?id=302681

[10] Christian Hartmann, Marc Reichenbach, and Dietmar Fey. 2015. IPOL - A Domain Specific Language for Image Processing . (2015). Retrieved October 29, 2019 from http://www.thinkmind.org/download.php?articleid=icons_2015_3_20_40047.

[11] Richard Membarth, Oliver Reiche, and Mehmet Akif Özkan. 2016. Overview. (2016). Retrieved October 29, 2019 from https://hipacc-lang.org/

[12] Kai Selgrad, Alexander Lier, Jan Dörntlein, Oliver Reiche, and Marc Stamminger. 2016. A High-Performance Image Processing DSL for Heterogeneous Architectures. (2016). Retrieved October 29, 2019 from https://dl.acm.org/citation.cfm?id=3005734

[13] Fabian Pedregosa, Gaël Varoquaux, Vincent Michiel, Bertrand Thirion, David Cournapeau, and Matthieu Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *JMLR 12*, 2825-2830.

[14] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2015. HIPAcc: A Domain-Specific Language and Compiler for Image Processing. (2015). Retrieved October 29, 2019 from https://graphics.cg.uni-saarland.de/papers/membarth-2016-tpds.pdf

[15] Basavaprasad B. and Ravi M. A STUDY ON THE IMPORTANCE OF IMAGE PROCESSING AND ITS APPLICATIONS. Retrieved November 7, 2019 from https://pdfs.semanticscholar.org/7656/d3db8962a5a75d162842065319155db73af8.pdf

[16] S. Padmappriya and K. Sumalatha. 2018. Digital Image Processing Real Time Applications. Retrieved November 13, 2019 from http://www.ijesi.org/papers/NCIOT-2018/Volume-1/9.%2046-51.pdf.