Columbia University

Simple Image Processing Language (SIP)
Final Report

Under the Supervision of: Prof. Stephen A. Edwards

Emad Barsoum
eb2871@columbia.edu

COMS W4115 - Summer 2013
Programming Languages and Translators

CONTENTS

1 INTRODUCTION

1.1 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.

1.2 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 (i.e. Matlab, R, SQL, Prolog…etc), commercial and non-commercial, yet there are very few for image processing even with the increasing popularity of computer vision algorithms.

Most of the efforts in writing domain-specific language for image processing are geared toward performance and GPU parallelism, and not ease 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 SIP programing language.

1.3 SIP 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 (picture element) 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 most functional languages.

The secondary goal is to provide imperative access to the raw image data for advance algorithms that can't be achieved using functional style in SIP language.

1.4 RESERVED KEYWORDS

For SIP, we 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. For the first version, we will be supporting: red, green and blue channel only.
kernel	A function that takes a pixel fragment and return a pixel. This function will run on the GPU, so it can't have access to any global variables.
fun	A generic function that run on the CPU.

The above are not a comprehensive list, SIP 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 are for image specific type.

1.5 RUNTIME

For GPU backend we decided to use OpenCL language, the main reason for such choice is the following:

1. OpenCL works across all Graphic Card Vendors, unlike CUDA which is limited to NVIDIA. In addition, OpenCL performance and capability is very close to CUDA.
2. OpenCL is supported by all Operating Systems. Unlike DirectCompute or AMP which is supported only on Window.
3. OpenCL is an open standard, unlike other techs which are proprietary.
4. OpenCL is now mature, and more people are moving to it.
5. OpenCL has binding on pretty much most programming languages.

With the above list, it made sense to us not limit ourselves for specific architecture or specific vendor.

Like most GPU language, OpenCL need a host App such host is usually written in C\C++ (there are bindings to Java, Python and other languages). Also, like most GPU language there is a lot of limitation on what can be parallelized. In order to keep SIP language flexible, we decided to support a hybrid model, by that we mean that part of the language will be compiled into C++ code, in addition, to the hosting code and the other part will be compiled into OpenCL code (extension cl).

We believe with such decision, we strike a good balance between parallelism and functionality.

1.6 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.

1.7 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.

1.8 LIMITATION

Initially the plan was to support image format with 1 or more channels and the ability to specify custom channel name. This initial plan was before we decided to support GPU as backend in addition to CPU. Due to GPU limitation, underlying image library limitation and limited time to invent my own format. We decided to support only 3 channels: Red, Green and Blue for this release. And to only support bitmap format.

The reason for the restriction to only support bitmap format is to reduce system dependency. I looked at several image libraries that work cross platform, and most of them depend on some extra dynamic library for image decoding such as PNG and JPEG. In order, to make it easy to compile the output of SIP compiler and run it without the extra overhead of doing any system installation, I decided to use EasyBMP library[4]. EasyBMP[4] is a small C++ library that knows how to read and write bitmap file, and it doesn't depend on any other library. Which is perfect for SIP project.

Furthermore, to avoid tight dependency on EasyBMP library or OpenCL, I wrote a generic wrapper, in sip.h and sip.cpp, which abstract both the image library and the GPU backend. So it should be trivial in the future to add more image formats and more GPU backend.

2 LANGUAGE TUTORIAL

The main type in SIP is "image", which abstract two type's specific to image processing:

1. The actual image in memory: in this case image will hold 2D array of pixels and each pixel contains one or more channels. Current version support only 3 channels: Red, Green, and Blue.
2. An Image kernel presentation: image kernel is what used in the convolution operation. For example, if we convolve an image with an edge detector kernel, then the outcome will be an image with the edges are more pronounced. Image Kernel is simply a small 2D array, in SIP we support only 3x3 kernel (which is the most used size, for performance and practicality reason).

2.1 COMPILE

First to build the compiler, simply type make from your terminal, assuming that you already have OCAML compiler installed.

In the same folder where you build SIP, there is a folder called "out" in the same location of the compiler. This folder contains the needed dependency to build the output of the SIP compiler, so it must be in the same location.

To compiler a SIP file, you need to type the below in your terminal:

./sip –c <filename>.sip

Replace <filename> with the name of your SIP file, the compiler will output three files in the "out" folder, named the same as the input SIP file. Here the 3 generated files:

1. <filename>.cpp: C++ file that contains the CPU portion of the SIP code and the hosting code for OpenCL.
2. <filename>cl: OpenCL file that contains the GPU portion of the SIP code.
3. Makefile: A Makefile that support both MacOSX and Linux.

Once the above files got generated, all what you need is to build "out" folder by simply type make in the terminal.

2.2 PROGRAM LAYOUT

The main entry point to SIP program is a function called "main" that doesn't return any value. As shown below

```
/*
    Block comments are similar to C comments.
    You can declare here your global variables.
*/

//
// Your program entry point.
//
fun main()
{
    // You can declare here your local variables.
```

```
    // The rest of the program.
}
```

As shown above, SIP support both one line comments and block comments, similar to C\C++. SIP doesn't support nested comments (like OCAML).

2.3 Reading and writing image file

The first step to deal with images is to acquire them. In SIP, to read and write an image from a file, we borrowed C++ stream syntax "<<" for reading and ">>" for writing.

```
fun main()
{
  image readImage;

  //
  // Reading test.bmp.
  //
  readImage << "./test.bmp";

  //
  // Writing to result.bmp.
  //
  readImage >> "./result.bmp";
}
```

2.4 CPU Image Operation

Now that we know the program layout, and how to read and write from an image file. Let's look at some of basic image operations that can be done in SIP, which will demonstrate how easy you can manipulate images in SIP.

We will first focus on the operations that will be compiled into a CPU code, then in the next subsection will focus on the GPU operation. All "In" expression run on the CPU.

```
//
// Flip color channels.
//
fun main()
{
  image src;                    // Source image
  image dst;                    // Destination image

  src << "./blackbuck.bmp";    // Read source image

  //
  // Flip the color channels between the source and the destination.
```

```
    //
    dst = src in (red, green, blue) for { red: green, green: blue, blue: red};

    dst >> "./blackbuck-flipped.bmp";
}
```

Input image on the left and the result is on the right for the above code:

```
//
// Color to gray.
//
fun main()
{
    image src;                      // Source image
    image dst;                      // Destination image

    src << "./blackbuck.bmp";     // Read source image

    //
    // Set each channel to the gray value of the src.
    //
    dst = src in (red, green, blue) for { red: 0.2126*red + 0.7152*green + 0.0722*blue,
                                          green: 0.2126*red + 0.7152*green + 0.0722*blue,
                                          blue: 0.2126*red + 0.7152*green + 0.0722*blue };

    dst >> "./blackbuck-gray.bmp";
}
```

Input image on the left and the result is on the right for the above code:

```
//
// Color threshold.
//
fun main()
{
   image src;                        // Source image
   image dst;                        // Destination image

   src << "./blackbuck.bmp";     // Read source image

   //
   // Set each channel to 255 or 0 depend on certain threshold value.
   //
   dst = src in (red, green, blue) for { red: red > 128 ? 255 : 0,
                                         green: green > 128 ? 255 : 0,
                                         blue: blue > 128 ? 255 : 0 };

   dst >> "./blackbuck-threshold.bmp";
}
```

Input image on the left and the result is on the right for the above code:

2.5 GPU IMAGE OPERATION

All convolution operations will run on the GPU. We have two type of convolution operations:

2.5.1 Static convolution

This is where you convolute an image with a static 3x3 matrix. While this isn't flexible, yet, there are a lot of operations that can be achieved by convolving 3x3 matrix with an image.

As shown below:

```
//
// Blur image.
//
fun main()
{
   image  blur = [[0.11, 0.11, 0.11] [0.11, 0.11, 0.11] [0.11, 0.11, 0.11]]; // Set the 3x3 blur filter
   image src;                          // Source image
   image dst;                          // Destination image

   src << "./blackbuck.bmp";     // Read source image

   //
   // Convolve 3x3 matrix with the "src" image, the below operation will run on the GPU.
   //
   dst = src ^ blur;

   dst >> "./blackbuck-blur.bmp";
}
```

Input image on the left and the result is on the right for the above code:

```
//
// Edge image.
//
fun main()
{
   image  edge = [[0.0, -1.0, 0.0] [-1.0, 5.0, -1.0] [0.0, -1.0, 0.0]]; // Set the 3x3 edge filter
   image src;                          // Source image
   image dst;                          // Destination image

   src << "./blackbuck.bmp";     // Read source image

   //
   // Convolve 3x3 matrix with the “src” image, the below operation will run on the GPU.
   //
   dst = src ^ edge;

   dst >> "./blackbuck-edge.bmp";
}
```

Input image on the left and the result is on the right for the above code:

2.6 DYNAMIC CONVOLUTION

This is the most flexible image operation on the GPU. You convolute the input image with a kernel function, kernel function is almost identical to normal function with some minor limitation to make it suitable to run on the GPU.

Here is an example of blue operation on the GPU.

```
//
```

```
// Blue image in the GPU using dynamic convolution.
//
fun main()
{
  image src;
  image dst;

  src << "./blackbuck.bmp"; // Read source image

  dst = src ^ blur;          // Convolute the source image with a kernel function.
                             // blue is a full blown function defined below

  dst >> "./blackbuck-blur-dyn.bmp"; // Save the result to a file.
}

//
// This is how to write a kernel function that will run on the GPU.
// Same exactly as normal function, with the following differences:
//   1. Use “kernel” instead of “fun”, when you define your function.
//   2. Only two arguments of type image, the first is the input and the second is the output.
//   3. No return type.
//   4. No access to global variables.
//   5. Can’t call another function from within the kernel.
//   6. Can’t read or save file within the kernel.
//   7. Must define: red_out, green_out and blue_out for the resulted pixel.
//       We automatic bind them to the second argument.
//
kernel blur (image in_image, image out_image)
{
  int x;
  int y;
  float red_out;
  float green_out;
  float blue_out;

  red_out = 0.0;
  green_out = 0.0;
  blue_out = 0.0;

  for (y = -1; y <= 1; y = y + 1)
  {
    for (x = -1; x <= 1; x = x + 1)
    {
      red_out = red_out + in_image[y,x]->Red / 9;       //  Here how to access individual channel.
      green_out = red_out + in_image[y,x]->Green / 9;
      blue_out = red_out + in_image[y,x]->Blue / 9;
    }
  }
}
```

Input image on the left and the result is on the right for the above code:

As shown above, writing a kernel function is very similar to writing a normal function with some restriction stems from the fact that multiple instances of the kernel can run in parallel on the GPU.

A kernel function is a function that takes two image arguments and return nothing. The first argument is the input image and the second argument is the output image. Kernel function begin with the “kernel” keyword instead of the “fun” keyword, and has no access to global variables.

You must define the following 3 float values to hold the result: “float red_out”, “float green_out”, and “float blue_out”. We will automatically bind them to the out image. Kernel function will be called fpr each pixel in the image.

The coordinate of the input image “in_image[0,0]” mean this pixel, you can move to any of the 8 directions to read this pixel neighbors, as shown below (this why the coordinate can be negative):

[-1,-1]	[-1,0]	[-1,1]
[0,-1]	[0,0]	[0,1]
[1,-1]	[1,0]	[1,1]

The above is different than the coordinate of the CPU code, in CPU [0, 0] means top left pixel of the image, and the coordinate system can’t be negative.

Another restriction, you can’t call another function from the kernel or interacting with the file system.

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

3 LANGUAGE MANUAL

3.1 INTRODUCTION

SIP which stand for “Simple Image Processing Language” is a programing language designed to simplify writing image processing algorithm, by removing the plumbing work that is usually associated with dealing with images in most programming language.

To achieve that, SIP borrowed some concepts and ideas from functional language in order to let developers focus on the mathematic aspect of image processing instead of its presentation in computer language.

3.2 LEXICAL CONVENTIONS

We have the following tokens in SIP: identifiers, keywords, expression operators, separators, constant and string literals. Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens.

3.2.1 Comments

SIP support block and one line comments. We borrowed C\C++ comment syntax. So comment in SIP begin with /* and end with */, anything in between will be ignored. SIP won’t support nested comments. One line comment is also similar to C\C++ using // until the end of the line.

3.2.2 Identifiers

An identifier is a sequence of alphanumeric letters in which first letter cannot be a number. Upper and lower case are considered different. If their ASCII code is different than they are different letter.

3.2.3 Keywords

Here the list of reserved keywords:

image	kernel
histogram	float
fun	int
in	bool
for	uint
if	true
else	false
while	pixel
return	break

We are not planning to use “pixel” keyword in the first version of the compiler; however, we reserved them for future use.

3.2.4 Expression operators

SIP support the following operators:

^	=<

=	?
==	->
>	!
<	&&
>=	&
||	|
*	+
-	/
%	~
>>	<<
[	]
(	)
:	.

3.2.5 Separators

Token separator are blank spaces, tab, and newline. Expression separator are comma “,” and semicolon “;”.

3.2.6 Constant

SIP support constant for each of the basic type and initialization constant.

3.2.6.1 Basic type constant

- For uint (unsigned integer), constant is simply a sequence of numerical characters next to each other, each characters can be between ‘0’ and ‘9’ inclusively.
- int (integer) is similar to uint except it can be prepended with the unary ‘–‘ operator to indicate negative number.
- Float constant will support the same floating point format supported by the C language. Which is a sequence of numeric characters, then a ‘.’ Symbol to indicate fraction, then the fraction part with optional exponent “e” component.

3.2.6.2 String literals

String literals constant will be presented, similar to C, between double quotes. SIP doesn’t support escape sequence.

“string”

3.2.6.3 Image Matrix

Image object can be initialized with 3x3 matrix that can be used in the convolution operators. To initialize an “image” object with constant values, each row need to be between bracket and the separation between each row element is comma:

[[expr, expr, expr] [expr, expr, expr] [expr, expr, expr]]

Here an example:

image filter = [[0.0, -1.0, 0.0] [-1.0, 5.0, -1.0] [0.0, -1.0, 0.0]];

3.3 Meaning of Identifiers

3.3.1 Types

SIP support two category of types:

3.3.1.1 Object Types

Such as image and histogram. By object we mean that those types will have some extra properties and metadata associated with them.

3.3.1.2 Basic Types

Basic types are the common types available in most programming language such as: bool, float, int, and uint.

3.4 Objects

The two built in objects for image manipulation are “image” and “histogram”, “image” abstract the concept of 2D image with 1 or more channels, and “histogram” abstract the concept of 1D gray level image histogram.

3.4.1 image

In SIP language, image object represent a 2 dimensional image, with one or more named channels, in memory. Image object also can act as a kernel filter for the convolution operator, in such a case, the image needs to be 3x3, it will be applied to each channel in the source image.

Because image is an object it has some properties that can be used for inspection or in an imperative algorithm. SIP image object will expose Height, Width, and *<channel name>* properties.

<channel name> can be red, green or blue.

3.4.2 Histogram

Histogram object abstract image based histogram computation and results. Histogram will contains 1 dimensional histogram per each named channel, which mean that histogram will have one or more named channels to match the corresponding source image.

3.5 Expressions Operators

In this section, we will describe expression operators supported by SIP.

3.5.1 Logical not ! and binary not ~

Logical Not ‘!’: simply negate the Boolean evaluation of a condition expression, a Boolean expression is an expression in which all its terms return “true” or “false”, such as the result of <, >, <=, >=.

!logical-expression

Binary Not ‘~’: can be applied on int or uint type only, and the result is simply flipping each bit from 1 to 0, and vice versa.

~expression

3.5.2 Convolution operator ^

Convolution operator is a binary operator that takes two images, or one image and another kernel function. Kernel function is a function that takes two parameters: input image and output image.

The evaluation is from left to right.

image-object ^ image-kernel

3.5.3 Multiplication and Division operators * /

Multiplication and division operators are a binary operator that takes two basic types (int, uint, or float) and return the result of the operation. If one of the types is float, then the result will be float. Evaluation is from left to right.

*expression * expression*

expression / expression

3.5.4 Addition and subtraction operators + -

Addition and subtraction operators are a binary operator that takes two basic types (int, uint, or float) and return the result of the operation. If one of the types is float, then the result will be float. Evaluation is from left to right.

expression + expression

expression- expression

3.5.5 Unary operator –

The unary operator – can be applied to int or float to negate the given number.

-constant

-variable

3.5.6 Mod operator %

Mod operator are a binary operator that takes int or uint type and return the result of the mod operation. Evaluation is from left to right.

expression % expression

3.5.7 Assignment operator =

The result of the expression on the right of the assignment operator will be copied to the variable on the left of the assignment operator.

variable-name = expression

3.5.8 Equality operator ==

== is a binary operator which check if the result of the expression on the left side equal to the result of the expression on the right side. The result of the == operator is a Boolean which can't be casted to other basic type (different from C and C++).

expression == expression

3.5.9 Logical operators && and ||

&& and || are binary operators that takes two Boolean expressions, and return true or false based on the result of those two expressions.

logical-expression && logical-expression

logical-expression || logical-expression

3.5.10 Binary operators & and |

& and | are binary operators that takes two expressions that evaluate to a basic types only, and return a new basic type in which each bit is the result of the & or | operator corresponding to the same bit in the two expression results.

expression & expression

expression | expression

3.5.11 Condition operators <, >, <=, >=

Those are binary operators that takes two expression, and the return of the result of: smaller than <, greater than >, smaller than or equal<=, and greater than or equal operations.

expression< expression

expression > expression

expression <= expression

expression >= expression

3.5.12 Read and write operators << and >>

SIP use << operator to read from a file and >> operator to write to a file. The right side of << or >> is a string literal that contains the file path, and the left side is an image object.

image-object << string-literal;

image-object >> string-literal;

3.5.13 Accessor operator [ ] ,

[and] operators can be applied only to image object or histogram object, for histogram [ ] operator takes an integer from 0 to 255 (the range of pixel per channel will be from 0 to 255) that represent the index of the bin that need access.

histogram-object[expression]

For image, the operator is in the form of *[<row>, <col>]* where *<row>* is the index of the row that we need to access, and *<col>* is the index of the column that we need to access.

image-object[expression, expression]

For histogram [ ] operator will be read only.

3.6 OPERATOR PRECEDENCE

Operator	Associativity
[] () .	Left to right
! ~ - (unary)	Right to left
^	Left to right
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
& |	Left to right
&& ||	Left to right
expression ? true_expression : false_expression	Left to right
,	Left to right

3.7 DECLARATION

3.7.1 Variables

Variable declaration in SIP will be begin with the type, followed by the variable name, and ended by a semi colon. As follow:

type-specifier instance-name;

SIP support initialization during declaration, as follow:

type-specifier instance-name = expression;

3.7.2 Functions

Here how to declare a function in SIP language.

fun function-name (comma-separated-parameter-list) return-type-optional

{

Local-variables-declaration

statements

}

Parameter-list are a sequence of type name pairs separated by comma. Recursive and nested functions aren't supported.

3.7.3 Kernel Functions

Kernel functions are functions that run on the GPU, it can't be called and must be used in the convolution operator.

Here how to declare a kernel function:

kernel function-name (image instance-name1, image instance-name2)

{

Local-variables-declaration

statements

}

Kernel functions take only two parameters and doesn't return any value. The first parameter is the input image and the second parameter is the output image. Kernel functions has no access to global variables.

3.8 STATEMENTS

Statements are a sequence of statements separated by a semi colon ';' and executed from left to right.

3.8.1 Conditional statements

SIP support if\else statement and ternary operator, as shown below:

If (condition-expression) then statement

If (condition-expression) then statement else statement

(condition-expression) ? success- statement : failed-statement

3.8.2 While statements

SIP support loop on a condition:

while (condition-expression) statement

3.8.3 For statements

For statement is as follow:

for (expression; conditional-expression; expression) statement

3.8.4 In statements

In statement map the source image to the specified domain (list of named channels).

Image-object in (list-of-channel-names);

3.8.5 In for statements

In For statement map the source image to the specified domain (list of named channels) using the provided expression.

Image-object in (list-of-channel-names) for expression;

3.8.6 Return statement

Return exit the execution of the current function, there are two form of return:

return;

return expression;

3.8.7 Break statement

Break break from the inner loop.

```
break;
```

3.9 PROGRAM ENTRY

In SIP, the entry point to the application is a function called main similar to C. The main function takes no argument and return nothing.

```
Global-variables-declaration

fun main()

{

    Local-variables-declaration

    statements

}
```

4 PROJECT PLAN

The plan of the project went through different set of milestones, I used mainly agile methodology, instead of waterfall model, in order to adapt quickly on changes. Changes came from two directions:

1. As the course progress, I learn more about compiler and adapt my original spec based on the experience that I gained during the course.
2. Hitting a road block. Mix and match between CPU and GPU is a hard problem.

4.1 PLANNING

Planning when through the following phases

4.1.1 Research and Development (R&D)

I searched online to see if my ideas where implemented before or not, I found two projects that were close but not similar to SIP: Halide[1] and Neon[2]. I compared both languages with what I want to do, and as mentioned in the first section there are some similarity but the goal is different. SIP goal is easy to use for prototyping and Halide & Neon goal were performance.

4.1.2 Selecting GPU language

There are a lot of interest in parallel programing using the GPU, which cause the proliferation of GPU languages and technology. I compare the following GPU as a candidate for SIP backend:

GPU technology	Description
DirectCompute	DirectCompute is Microsoft answer to OpenCL. The biggest advantage of DirectCompute is that I am well familiar with. However, the main drawback, is that it runs only on Windows (and only on Windows 7 and above).
CUDA	From NVidia, is one of the well-known GPU language that is used for simulation and other heavy parallelized tasks. However, it run only on NVidia GPU.
WebGL	Expose OpenGL ES API and shader to Javascript, however, the exposed subset is too limited for SIP.
OpenCL	An open standard for heterogeneous parallel language, the spec is managed by Khronos[5] group. OpenCL is supported by pretty much everyone, so it fit nicely with my needs for SIP language.
AMP	A high level GPU language inspired by C++ STL syntax invented by Microsoft. AMP is actually pretty cool technology, however, it run only on Window.

4.2 SPECIFICATION

To come up with the specification of SIP language, I used TDD methodology (Test Driven Development). I am experienced in Computer Vision and image processing. So I just opened a text editor, and begin to write some image processing algorithm with pseudo like language. I iterated on it multiple times, until it felt good and easy.

My goal was getting rid of the plumbing and focus on the algorithm.

4.3 DEVELOPMENT AND TESTING

Initially development was focusing on two separate items in parallel:

1. Scanner and parser for the SIP language. Which including debugging both scanner and parser and verify that I get the expected output by dumping the generated AST into a text file.
2. Write an abstraction layer on the top of the chosen image library and OpenCL API. Also I wrote a couple of C++ and OpenCL program to test such layer and make sure that I am getting the right result.

After finishing the above two items, I spend the rest of the time focusing on the translator itself which translate from AST to C++ and CL. And make sure that the resulted C++ and CL code compile without any errors. Also, I fed the generated code sample bitmap image and verified that the resulted image matches the expected operation from the code.

4.4 PROGRAMING STYLE

I am new to OCAML so instead of inventing a new style, I followed the recommended style of the main maintainer of the language as much as possible:

http://caml.inria.fr/resources/doc/guides/guidelines.en.html

4.5 PROJECT TIMELINE

Regarding project timeline, it began initially slow, most of the initial time was spend on trying different syntax and trying different GPU technology, until I closed on both.

4.5.1 Here the summary of the timeline

Period	Summary
August 1 to 16	Complete Translate.ml for both C++ and OpenCL, add more test cases. Finish the remaining missing items: Kernel function, complete “In” Expression. Complete the final report.
July 16 to 31	Add Makefile, sip.ml (SIP compiler entry function). Write a C++ wrapper for OpenCL, Image Manipulation, and Histogram. Add a basic SIP to C++ translator, fixes a lot of bugs, and be able to compile a basic SIP program. Add first test case. Begin working on the final report.
July 1 to 15	Finish scanner.mll, ast.ml, parser.mly. Build everything manually.
June	Finish the proposal, LRM, and investigate multiple GPU technology.

4.6 Development environment

After a lot of testing and experiments, I decided to use TextMate 2.0 as my main editor with OCAML, GIT and Makefile bundle. With those 3 bundles, I could easily update my code, build and submit the change to my source control from the editor.

Textmate 2.0: http://blog.macromates.com/2012/textmate-2-at-github/

Instruction for settings Textmate 2.0 as OCAML enviroment: http://meandocaml.wordpress.com/ocaml-for-osx/textmate-my-choice/

For source control, I used GIT source control hosted by bitbucket: https://bitbucket.org/

4.7 Project log

Here my bitbucket commit history

Emad Barsoum pushed 1 commit to ebarsoum/er
16 hours ago
aa93095 - Fix test automation.

Emad Barsoum pushed 1 commit to ebarsoum/er
17 hours ago
e2edd1a - Add more test cases to cover most image operations..

Emad Barsoum pushed 1 commit to ebarsoum/er
22 hours ago
f281759 - Add a test script and update sip.ml to be more robust against bad argument.

Emad Barsoum pushed 1 commit to ebarsoum/er
yesterday
f8ae67c - Minor code cleanup...

Emad Barsoum pushed 1 commit to ebarsoum/er
2 days ago
14cd10f - Adding support to write kernel function on the GPU. Now both GPU target

Emad Barsoum pushed 1 commit to ebarsoum/er
yesterday
07a6adf - GPU now work on convolution with 3x3 filter.

Emad Barsoum pushed 1 commit to ebarsoum/er
2 days ago
4417b47 - Add good test cases folder, which contains the test case that expect to run without error.

Emad Barsoum pushed 1 commit to ebarsoum/er
2 days ago
ec5ff71 - Fix "In" expression, write a test case that flip color channels, and update makefile generators. This is the first end-to-end working sample...

Emad Barsoum pushed 1 commit to ebarsoum/er
2 days ago
bf0fbbf - Add make file generator for Mac OS X.

Emad Barsoum pushed 1 commit to ebarsoum/er
2 days ago
39c2653 - Add OpenCL support for 3x3 image filter.

Emad Barsoum pushed 1 commit to ebarsoum/er
2 days ago
74d49b2 - Add 3x3 matrix support.

Emad Barsoum pushed 1 commit to ebarsoum/er
2 days ago
3480918 - Fix In expression, now In expression expand to proper C++...

Emad Barsoum pushed 1 commit to ebarsoum/er
3 days ago
7aaeb05 - A lot of cleanup and code refactory. In addition, of adding variable initialization during declaration.

Emad Barsoum pushed 1 commit to ebarsoum/er
4 days ago
3fe6f88 - Update the OpenCL wrapper with the following:

Emad Barsoum pushed 1 commit to ebarsoum/er
5 days ago
4c554f2 - Update the shader wrapper, add wrapper around a generic shader kernel.

Emad Barsoum pushed 1 commit to ebarsoum/er
5 days ago
5583085 - Add CLProgram class to abstract OpenCL plumbings...

Emad Barsoum pushed 1 commit to ebarsoum/er
5 days ago
59cb0a8 - Add some helper C++ classes for image manipulation and OpenCL code.

Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-07
ef60d97 - Add the missing C++ headers...

Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-06
4642446 - Remove the extra brackets generated during compilation...

Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-06
f9834c2 - Fix translare.ml build break and add more conversion.

Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-06
95e59ff - Fix statement and expression translation in translate.ml module.

Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-05
33e9018 - Update translate unit with function support.

Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-05
ede26dd - Add translate.ml based on compile.ml from MicroC.

Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-04
8c2eb83 - Add support to inline "in" expression for image manipulation.

Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-04
30224f2 - Fix parser error exception, add string literal, and add read\write operator.

Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-02
4bb98d2 - Add more test cases, and add range operator.


Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-01
4c135c2 - Adding first test case, an empty main function.


Emad Barsoum pushed 1 commit to ebarsoum/er
2013-08-01
66a1761 - Minor update to main function.


Emad Barsoum pushed 1 commit to ebarsoum/er
2013-07-28
b117dac - Add a small OpenCL program and BMP library.


Emad Barsoum pushed 1 commit to ebarsoum/er
2013-07-25
41bbf05 - Add main entry to SIP compiler.


Emad Barsoum pushed 1 commit to ebarsoum/er
2013-07-25
2fbf4ac - Add sip.ml as the driving module for sip compiler.


Emad Barsoum pushed 1 commit to ebarsoum/er
2013-07-23
862e5a1 - Properly fix boolean type, and add support to the rest of the binary operators.


Emad Barsoum pushed 1 commit to ebarsoum/er
2013-07-21
5f0a1de - Add a makefile, and build everything through make. In addition, to fixing all warnings and errors.


Emad Barsoum pushed 1 commit to ebarsoum/er
2013-07-21
723f42d - Fix ast.ml build errors, and fix all build warnings.

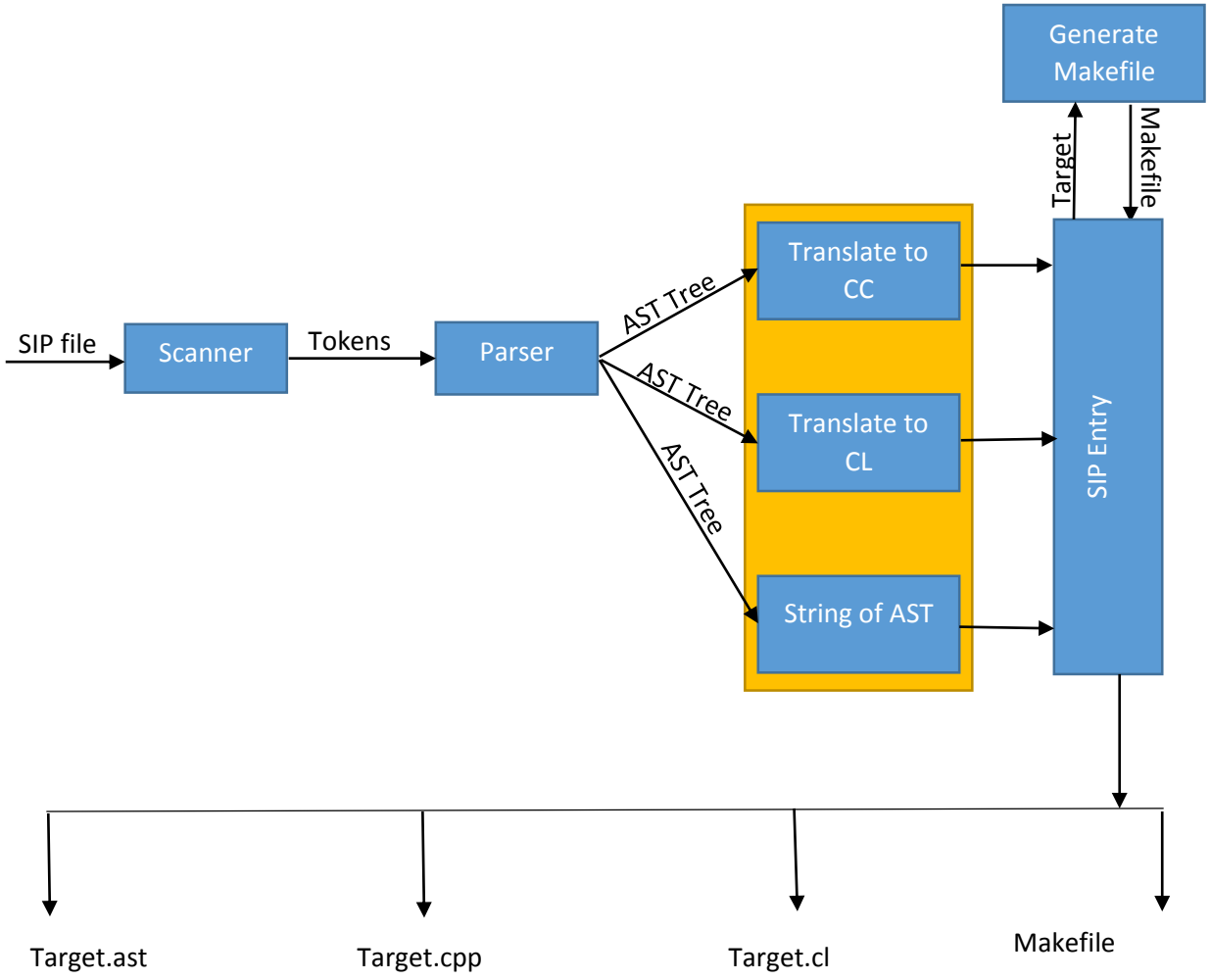
Older log:

Author	Commit	Message	Date
Emad	ea2957b	Implement basic support for the inline in operator in the parser.	2013-07-15

Barsoum			
? **Emad Barsoum**	d7cab3a	Add kernel function to the scanner, parser and AST. Kernel function will run on the GPU.	2013-07-14
? **Emad Barsoum**	ade702c	Add support to unary minus., variable declaration with types, precedence order.	2013-07-14
? **Emad Barsoum**	13f2695	Update the scanner, parser, and ast with more types.	2013-07-12
? **Emad Barsoum**	5c210d6	Add parser.mly.	2013-07-08
? **Emad Barsoum**	69591fe	Add ast.ml based on MicroC compiler. Add support to the following:	2013-07-06
? **Emad Barsoum**	537f3fd	Fix single line comment.	2013-07-04
? **Emad Barsoum**	7fb2076	Fix tab and space layout issue, and add additional supported types.	2013-07-04
? **Emad Barsoum**	272dfa5	Initial version of SIP scanner and makefile.	2013-07-04
? **Emad Barsoum**	580e020	Add readme file and set folder layout.	2013-06-08

5 Architectural Design

The high level architecture of the project is similar to most compiler projects, with the exception that SIP backend spew both: C++ code and GPU code (OpenCL language). We will talk about why we need both and the purpose of each one later.



5.1 Interface between components

5.1.1 Scanner

Takes SIP source code and produce a list of tokens.

5.1.2 Parser

Takes the list of tokens from the scanner and generate the Abstract Syntax Tree (AST) based on SIP language grammars.

5.1.3 Translate to CC

Takes the AST tree as input, do some validation and generate C++ code.

5.1.4 Translate to CL

Takes the AST tree as input, do some validation and generate CL code. CL is the GPU language supported by OpenCL which is subset of C with some GPU syntactic sugar.

5.1.5 String of AST

Used for debugging, takes the AST tree as input, then generate a string form of the AST tree without any validation. Also this module provides some helper function for the translate unit.

5.1.6 Generate Makefile

This module takes the target filename as an input and generates one Makefile that support both MacOS X and Linux.

5.1.7 SIP Entry

SIP Entry (sip.ml) is the entry point module for SIP compiler. It takes user input as inputs (source file path and a switch), and dispatch the input to the right module as needed.

5.2 GPU VERSUS CPU

For GPU backend we decided to use OpenCL, the main reason for such choice is that OpenCL works across all Graphic Card Vendors, and across all Operating Systems. Which made sense to not limit ourselves for specific architecture.

Like most GPU language, OpenCL need a host App such host is usually written in C\C++ (there is binding to Java, Python and other languages). Also, like most GPU language there is a lot of limitation on what can be parallelized. In order to keep SIP language flexible, we decided to support a hybrid model, by that we mean part of the language will be compiled into C++ code in addition to the hosting code and the other part will be compiled into OpenCL code.

We believe with such decision we strike a good balance between parallelism and functionality.

6 TEST PLAN

The test was done incrementally, first I have tested both the scanner and parser together, and verified that the resulted AST conformed to what I was expected. Once I was satisfied with both the scanner and parser, I moved to the translator. The translator test was a little more complicated because it spew both CPU and GPU code.

So to simplify the testing, I first focused on the CPU code and made sure that the part of SIP code that generate CPU code is working as expected, by compiling the output using GCC and verify that it performs as predicted using debugger and "printf". After that I moved to the GPU backend which was the last module that I implemented and tested.

6.1 HERE SOME SIP CODE AND THEIR CORRESPONDING C++ AND CL GENERATED CODE

Each SIP file generates three files: A C++ file with extension *.cpp, an OpenCL file with extension *.cl, and a Makefile beside those three files there are some dependency file needed for the project to build.

6.1.1 CPU Code: Converting Color image into a gray image

The below is a simple code that convert a color image into a gray image. "In" expression is compiled into a CPU code.

color-to-gray.sip

```
//
// Color to gray.
//
fun main()
{
   image src;                          // Source image
   image dst;                          // Destination image

   src << "./blackbuck.bmp";    // Read source image

   //
   // Set each channel to the gray value of the src.
   //
   dst = src in (red, green, blue) for { red: 0.2126*red + 0.7152*green + 0.0722*blue,
                                         green: 0.2126*red + 0.7152*green + 0.0722*blue,
                                         blue: 0.2126*red + 0.7152*green + 0.0722*blue };

   dst >> "./test-color-to-gray.bmp";
}
```

Here the generated C++ file:

```
#include "sip.h"
using namespace Sip;
```

```
ClProgram g_clProgram;
Image g__sip_temp__;


int main()
{
   g_clProgram.CompileClFile("./test-color-to-gray.cl");

Image dst;
Image src;

src.read("./blackbuck.bmp");
g__sip_temp__.clone(src);
for (int row = 0; row <src.height(); ++row)
{
   for (int col = 0; col <src.width(); ++col)
   {
      unsigned int red = src(row, col)->Red;
      unsigned int red_out = src(row, col)->Red;
      unsigned int green = src(row, col)->Green;
      unsigned int green_out = src(row, col)->Green;
      unsigned int blue = src(row, col)->Blue;
      unsigned int blue_out = src(row, col)->Blue;

red_out = 0.2126 * red + 0.7152 * green + 0.0722 * blue;
green_out = 0.2126 * red + 0.7152 * green + 0.0722 * blue;
blue_out = 0.2126 * red + 0.7152 * green + 0.0722 * blue;

      g__sip_temp__(row, col)->Red   = (char)red_out;
      g__sip_temp__(row, col)->Green = (char)green_out;
      g__sip_temp__(row, col)->Blue  = (char)blue_out;
      g__sip_temp__(row, col)->Alpha = src(row, col)->Alpha;
   }
}

dst = g__sip_temp__;
dst.write("./test-color-to-gray.bmp");


   return 0;
}
```

And here the generated OpenCL code:

```
__constant sampler_t sampler =  CLK_NORMALIZED_COORDS_FALSE |
```

```
CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

__kernel void apply_filter(__read_only image2d_t in_image, __write_only image2d_t out_image,
__constant float* filter)
{
  const int2 pos = {get_global_id(0), get_global_id(1)};

  float4 sum = (float4)(0.0f);
  for (int y = -1; y <= 1; y++)
  {
     for (int x = -1; x <= 1; x++)
     {
        sum.x += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).x;
        sum.y += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).y;
        sum.z += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).z;
     }
  }

  write_imagef (out_image, (int2)(pos.x, pos.y), sum);
}
```

6.1.2 GPU Code: Apply Edge filter using a static kernel

gpu-edge.sip

```
//
// Edge image.
//
fun main()
{
 image edge = [[0.0, -1.0, 0.0] [-1.0, 5.0, -1.0] [0.0, -1.0, 0.0]]; // Set the 3x3 edge filter
 image src;                          // Source image
 image dst;                          // Destination image

 src << "./blackbuck.bmp";     // Read source image

 //
 // Convolve 3x3 matrix with the “src” image, the below operation will run on the GPU.
 //
 dst = src ^ edge;

 dst >> "./test-gpu-edge.bmp";
}
```

Here the generated C++ code:

```
#include "sip.h"
```

```
using namespace Sip;

ClProgram g_clProgram;
Image g__sip_temp__;


int main()
{
   g_clProgram.CompileClFile("./test-gpu-edge.cl");

Image dst;
Image src;
float edge[3][3] = {{0., -1., 0.}, {-1., 5., -1.}, {0., -1., 0.}};

src.read("./blackbuck.bmp");
g_clProgram.ApplyFilter(src, g__sip_temp__, (float*)&edge);

dst = g__sip_temp__;
dst.write("./test-gpu-edge.bmp");


   return 0;
}
```

And here the generated OpenCL code:

```
__constant sampler_t sampler =  CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

__kernel void apply_filter(__read_only image2d_t in_image, __write_only image2d_t out_image,
__constant float* filter)
{
  const int2 pos = {get_global_id(0), get_global_id(1)};

  float4 sum = (float4)(0.0f);
  for (int y = -1; y <= 1; y++)
  {
     for (int x = -1; x <= 1; x++)
     {
        sum.x += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).x;
        sum.y += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).y;
        sum.z += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).z;
     }
  }

  write_imagef (out_image, (int2)(pos.x, pos.y), sum);
}
```

6.1.3 GPU Code: Apply a Blur filter using a kernel function

gpu-kfun-blur.sip

```
//
// Blue image in the GPU using dynamic convolution.
//
fun main()
{
  image src;
  image dst;

  src << "./blackbuck.bmp"; // Read source image

  dst = src ^ blur;        // Convolute the source image with a kernel function.
                           // blue is a full blown function defined below

  dst >> "./test-gpu-kfun-blur.bmp"; //  Save the result to a file.
}

//
// This is how to write a kernel function that will run on the GPU.
// Same exactly as normal function, with the following differences:
//    1. Use “kernel” instead of “fun”, when you define your function.
//    2. Only two arguments of type image, the first is the input and the second is the output.
//    3. No return type.
//    4. No access to global variables.
//    5. Can’t call another function from within the kernel.
//    6. Can’t read or save file within the kernel.
//    7. Must define: red_out, green_out and blue_out for the resulted pixel.
//         We automatic bind them to the second argument.
//
kernel blur (image in_image, image out_image)
{
    int x;
    int y;
    float red_out;
    float green_out;
    float blue_out;

    red_out = 0.0;
    green_out = 0.0;
    blue_out = 0.0;

    for (y = -1; y <= 1; y = y + 1)
```

```
   {
      for (x = -1; x <= 1; x = x + 1)
      {
         red_out = red_out + in_image[y,x]->Red / 9;
         green_out = green_out + in_image[y,x]->Green / 9;
         blue_out = blue_out + in_image[y,x]->Blue / 9;
      }
   }
}
```

Here the generated C++ code:

```
#include "sip.h"
using namespace Sip;

ClProgram g_clProgram;
Image g__sip_temp__;


int main()
{
   g_clProgram.CompileClFile("./test-gpu-kfun-blur.cl");

Image dst;
Image src;

src.read("./blackbuck.bmp");
g_clProgram.RunKernel(src, g__sip_temp__,"blur");

dst = g__sip_temp__;
dst.write("./test-gpu-kfun-blur.bmp");


   return 0;
}
```

And here the generated OpenCL code:

```
__constant sampler_t sampler =  CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

__kernel void apply_filter(__read_only image2d_t in_image, __write_only image2d_t out_image,
__constant float* filter)
{
  const int2 pos = {get_global_id(0), get_global_id(1)};

  float4 sum = (float4)(0.0f);
```

```
  for (int y = -1; y <= 1; y++)
  {
     for (int x = -1; x <= 1; x++)
     {
        sum.x += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).x;
        sum.y += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).y;
        sum.z += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).z;
     }
  }

  write_imagef (out_image, (int2)(pos.x, pos.y), sum);
}

__kernel void blur(__read_only image2d_t in_image , __write_only image2d_t out_image)
{
   const int2 pos = {get_global_id(0), get_global_id(1)};
float blue_out;
float green_out;
float red_out;
int y;
int x;

red_out = 0.;
green_out = 0.;
blue_out = 0.;
for (y = -1 ; y <= 1 ; y = y + 1) {
for (x = -1 ; x <= 1 ; x = x + 1) {
red_out = red_out + read_imagef(in_image, sampler, pos + (int2)(x,y)).x / 9;
green_out = green_out + read_imagef(in_image, sampler, pos + (int2)(x,y)).y / 9;
blue_out = blue_out + read_imagef(in_image, sampler, pos + (int2)(x,y)).z / 9;

}

}



   float4 _out_ = {red_out, green_out, blue_out, 0.0f};
   write_imagef (out_image, (int2)(pos.x, pos.y), _out_);
}
```

6.2 TEST SUITES

For the test suites, I focused to cover what makes SIP language unique. Which is the functional language inspired behavior that compile into C++ and GPU code. First, I compiled each test case manually and verified that the resulted C++\OpenCL project build and work as expected. I used a sample image[6] for each of the test cases and verified that the resulted image match the actual algorithm. Once I got my ground truth, I use the verified output as reference for regression test.

I modified the “testall.sh” script from MicroC language to run all test cases and return the result. I also added specific test switch to the SIP compiler to return C++ or OpenCL output on demand.

After having everything setup, for each new feature or fixes, I always run the test suite and verify that there is no regression.

6.2.1 Here the list of the most important test cases

Test Case	Description
test-flip-colors.bmp test-flip-colors.clout.cl test-flip-colors.cppout.cpp test-flip-colors.sip	Using the inline “in” expression, this test case flip the order of the channel.
test-color-threshold.bmp test-color-threshold.clout.cl test-color-threshold.cppout.cpp test-color-threshold.sip	This test case shows the power of the inline expression, by illustrating the ternary operator inside the expression to threshold the image.
test-color-to-gray.bmp test-color-to-gray.clout.cl test-color-to-gray.cppout.cpp test-color-to-gray.sip	This test case show a more complex expression inside the “In” expression to convert color inot a Luma (Gray).
test-gpu-blur.bmp test-gpu-blur.clout.cl test-gpu-blur.cppout.cpp test-gpu-blur.sip	This is a GPU test case that exercise a constant 3x3 filter on an image.
test-gpu-edge.bmp test-gpu-edge.clout.cl test-gpu-edge.cppout.cpp test-gpu-edge.sip	Another GPU test case that uses edge filter, which contains negative numbers.
test-gpu-kfun-blur.bmp test-gpu-kfun-blur.clout.cl test-gpu-kfun-blur.cppout.cpp test-gpu-kfun-blur.sip	More advance GPU test case, testing a custom kernel function that blue the image.
test-img-read-write.bmp test-img-read-write.clout.cl test-img-read-write.cppout.cpp test-img-read-write.sip	My first test case and my first hello world program. This test case read an image and write it with different name.

6.3 REASON FOR THOSE TEST SUITES

The goal of SIP language is to simplify writing image processing algorithm; therefore, image algorithms were the focus of the chosen test cases. Even that, SIP support imperative programing, this part was low priority so I ad-hoc tested it only.

6.4 AUTOMATION USED

I used the same shell script used my MicroC compiler project, I modified the script to match my need for testing both the generated C++ code and OpenCL code.

7 Lessons Learned

There are a lot of lessons learned during the course of this project, and sadly some of them were learned the hard way.

7.1 Know when to stop

Language design is an art, each programing language still continue to evolve, some with slower pace than other, nevertheless, there is no finished language. One of my biggest problem is when to stop, I kept adding and changing features for a little longer than I wanted before begin writing anything.

Recommendation: begin with small spec then once finished end to end, iterate on it if you have time. Small working language are way better than a complex non-working language.

7.2 Reduce the number of variables

In this project the risk was too high for me, everything was new. New to OCAML, new to OpenCL (used as GPU backend), new image library, new to the development environment, and new to compiler writing. That's is too much risk, especially with the time constrain that we have.

This course is about compiler and I should have focused on this learning aspect of the project and kept the rest as known as possible.

7.3 Begin early

Easily said than done, especially with homework's and exams. However, I should have planned the project better and shouldn't have underestimated the work required for such project.

8 APPENDIX

Code listing include all OCAML files and the C++ wrapper.

8.1 SIP.ML

sip.ml

```
(*
   Columbia University

   PLT 4115 Course - SIP Compiler Project

   Under the Supervision of: Prof. Stephen A. Edwards
   Name: Emad Barsoum
   UNI: eb2871

   sip.ml for SIP Compiler entry point
*)

open Printf

type action = Ast | Compile | TestCpp | TestOpenCL | Error

let out_name = ref "a"

(* Helper function that write "content" into a file named "name" *)
let fwrite name content =
   let out = open_out name in
    fprintf out "%s\n" content;
    close_out out

(* Return filename without extension or path in Unix like system. *)
let get_filename path =
   let i = (String.rindex path '/') in
   String.sub path (i + 1) ((String.length path) - i - 5)

let right_string =
   "\n          Columbia University\n\n" ^
   "Simple Image Processing Compiler Version 1.0 Preview\n\n" ^
          "Under the Supervision of: Prof. Stephen A. Edwards\n" ^
          "Name: Emad Barsoum, UNI: eb2871\n\n"

let usage_string =
   right_string ^
   "Usage:\n\n" ^
   "   Compiling to C++ and OpenCL  : sip -c <filename>\n" ^
```

```
   "    Compiling to AST Tree       : sip -a <filename>\n" ^
   "    Compiling to C++ to stdin   : sip -tcc <filename>\n" ^
   "    Compiling to OpenCL to stdin : sip -tcl <filename>\n"

let main () =
  let action =
    if Array.length Sys.argv > 2 then
            try
      List.assoc Sys.argv.(1) [ ("-a", Ast);
                          ("-c", Compile);
                          ("-tcc", TestCpp);
                          ("-tcl", TestOpenCL) ]
     with
              _ -> Error
    else Error in
      let lexbuf = ignore(out_name := get_filename Sys.argv.(2));
               Lexing.from_channel (open_in Sys.argv.(2)) in
      let program = Parser.program Scanner.token lexbuf in
      match action with
       Ast -> let listing = Ast.string_of_program program in
              fwrite ("./out/" ^ !out_name ^ ".ast") listing
      | Compile -> let listing = Translate.translate_to_cc program !out_name in
               let cllisting = Translate.translate_to_cl program !out_name in
              Makefile.gen_makefile !out_name;
              fwrite ("./out/" ^ !out_name ^ ".cpp") listing;
              fwrite ("./out/" ^ !out_name ^ ".cl") cllisting
      | TestCpp -> let listing = Translate.translate_to_cc program !out_name in
               print_endline listing
      | TestOpenCL -> let listing = Translate.translate_to_cl program !out_name in
               print_endline listing
      | Error -> print_string usage_string

let _ = main ()
```

8.2 SCANNER.MLL

Scanner.mll

```
(*
   Columbia University

   PLT 4115 Course - SIP Compiler Project

   Under the Supervision of: Prof. Stephen A. Edwards
   Name: Emad Barsoum
   UNI: eb2871
```

```
   Scanner.mll for SIP language
*)

{
  open Parser
  open Printf
}

let newline    = '\n' | "\r\n"
let whitespace = [' ' '\t']
let digit      = ['0'-'9']
let exp        = 'e' ['-' '+']? digit+
let int_t      = digit+
let float_t    = digit+ '.' digit* exp? | digit+ exp | '.' digit+ exp?
let string_r   = '\\'+ | [^ '"']
let string_t   = '"' string_r* '"'

rule token = parse
  newline                                   { Lexing.new_line lexbuf; token lexbuf }
  | whitespace                          { token lexbuf }

  (* Read and write image files *)
  | "<<"              { READ  }
  | ">>"              { WRITE }

  (* Arithmetic operations *)
  | '+'              { PLUS       }
  | '-'              { MINUS      }
  | '*'              { TIMES      }
  | '/'              { DIVIDES    }
  | '%'               { MOD        }
  | '^'              { CONV       }
  | '='              { ASSIGN     }

  (* Logic operations *)
  | "!="              { NEQ    }
  | '<'              { LT     }
  | "<="              { LEQ    }
  | '>'              { GT     }
  | ">="              { GEQ    }
  | "=="              { EQ     }
  | "&&"              { AND    }
  | "||"              { OR     }
  | '!'              { NOT    }
  | '?'              { QUES   }

  (* Bit operations *)
```

```
  | '&'            { BITAND  }
  | '|'            { BITOR   }
  | '~'            { BITNOT  }

  (* Scoping, accessors, and sequences *)
  | '('            { LPAREN    }
  | ')'            { RPAREN    }
  | '['            { LBRACKET  }
  | ']'            { RBRACKET  }
  | '{'            { LBRACE    }
  | '}'            { RBRACE    }
  | ';'            { SEMI      }
  | ':'            { COLON     }
  | ','            { COMMA     }
  | "->"             { ARROW     }
  | ".."             { RANGE     }

  (* Supported types *)
  | "bool"         { BOOL    }
  | "int"          { INT     }
  | "uint"         { UINT    }
  | "float"        { FLOAT   }
  | "histogram"      { HIST    }
  | "image"        { IMAGE   }

  (* Control flow and loop *)
  | "if"           { IF      }
  | "else"           { ELSE    }
  | "for"            { FOR     }
  | "in"           { IN      }
  | "while"          { WHILE   }
  | "return"         { RETURN  }
  | "break"          { BREAK   }

  (* function *)
  | "fun"            { FUN     }
  | "kernel"         { KERNEL  }

  (* Identifier, types, comments and EOF. *)
  | "true"           { BLITERAL(true) }
  | "false"          { BLITERAL(false)}
  | int_t  as lxm      { ILITERAL(int_of_string lxm)   }
  | float_t as flt     { FLITERAL(float_of_string flt) }
  | string_t as lxm    { SLITERAL(lxm) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
  | "/*"             { comment lexbuf       }
  | "//"             { line_comment lexbuf   }
  | eof              { EOF }
```

```
  | _ as char        { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
   "*/"   { token lexbuf   }
  | _     { comment lexbuf }

and line_comment = parse
   '\n' | "\r\n" { token lexbuf       }
  | _            { line_comment lexbuf }
```

8.3 AST.ML

Ast.ml

```
(*
   Columbia University

   PLT 4115 Course - SIP Compiler Project

   Under the Supervision of: Prof. Stephen A. Edwards
   Name: Emad Barsoum
   UNI: eb2871

   ast.ml for SIP language
*)

type binary_op = Add | Sub | Mult | Div | Mod |
            Neq | Lt | Leq | Gt | Geq | Eq | And | Or | Not |
            BitAnd | BitOr | BitNot

type unary_op = Neg

type image_op = Conv

type var_type = Void | Bool | Int | UInt | Float | Matrix3x3 | Histogram | Image
type var_decl = { vname : string; vtype : var_type }

type expr =
   BoolLiteral of bool
  | IntLiteral of int
  | FloatLiteral of float
  | StringLiteral of string
  | Id of string
  | Unop of unary_op * expr
  | Binop of expr * binary_op * expr
  | Assign of string * expr
```

```
  | Call of string * expr list
  | Ques of expr * expr * expr
  | Bracket of expr
  | Imaccessor of string * expr * expr * string
  | Accessor of string * string
  | Noexpr

type channel =
   Channel of string * string

type row3 =
   Row of expr * expr * expr

type img_expr =
   Imop of string * image_op * string
  | In of string * channel list * expr list
  | Imassign of string * img_expr
  | Imrange of string * int * int * int * int

type var_init =
   Iminit of var_decl * img_expr
  | Vinit of var_decl * expr
  | Immatrix3x3 of var_decl * row3 * row3 * row3

type var_def =
   VarDecl of var_decl
  | Varinit of var_init

type stmt =
   Block of stmt list
  | Expr of expr
  | Imexpr of img_expr
  | Imread of string * string
  | Imwrite of string * string
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Break

type func_decl = {
   fname   : string;
   fparams : var_decl list;
   flocals : var_def list;
   fbody   : stmt list;
          freturn : var_type;
          fgpu    : bool
  }
```

```
type program = var_def list * func_decl list

(*
  The below are some helper functions, some of them are used in the translate
  unit and other for testing
*)

let string_of_vartype = function
   Void -> "void"
  | Bool -> "bool"
  | Int -> "int"
  | UInt -> "unsigned int"
  | Float -> "float"
  | Matrix3x3 -> "float"
  | Histogram -> "Histogram"
  | Image -> "Image"

let rec string_of_expr = function
   BoolLiteral(l) -> string_of_bool l
  | IntLiteral(l) -> string_of_int l
  | FloatLiteral(l) -> string_of_float l
  | StringLiteral(l) -> l
  | Id(s) -> s
  | Unop(o, e) ->
     (match o with
           Neg -> "-") ^ string_of_expr e
  | Binop(e1, o, e2) ->
     string_of_expr e1 ^ " " ^
     (match o with
           Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/" | Mod -> "%"
     | Neq -> "!=" | Lt -> "<" | Leq -> "<=" | Gt -> ">" | Geq -> ">=" | Eq -> "=="
     | And -> "&&" | Or -> "||" | Not -> "!"
     | BitAnd -> "&" | BitOr -> "|" | BitNot -> "~") ^ " " ^
     string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
     f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Ques (e1, e2, e3) -> "(" ^ string_of_expr e1 ^ ") ? " ^
     string_of_expr e2 ^ ":" ^ string_of_expr e3
  | Bracket (e) -> "(" ^ string_of_expr e ^ ")"
  | Imaccessor (i, r, c, a) -> i ^ "(" ^ string_of_expr r ^ "," ^ string_of_expr c ^ ")->" ^ a
  | Accessor (i, a) -> i ^ "->" ^ a
  | Noexpr -> ""

let string_of_row3 = function
   Row(e1, e2, e3) -> "{" ^ string_of_expr e1 ^ ", "
                          ^ string_of_expr e2 ^ ", "
```

```
                                                    ^ string_of_expr e3 ^ "}"

let get_channel = function
   Channel(_, c) -> c

let string_of_channel = function
   Channel(i, c) -> i ^ "(row, col)->" ^ String.capitalize c

let string_of_channels c =
  if ((List.length c) != 0)
  then
   (string_of_channel (List.hd c) ^
    String.concat "" (List.map (fun f -> ", " ^ string_of_channel f) (List.tl c)))
  else ""

let rec string_of_img_expr = function
   Imop(s, o, k) -> "conv(" ^ s ^ "' " ^ k ^ ");\n";
  | In (v, a, el) -> "in (" ^ string_of_channels a ^ ")\n{\n" ^
     String.concat ";\n" (List.map string_of_expr el) ^ ";}\n"
  | Imassign(v, e) -> v ^ " = " ^ string_of_img_expr e
  | Imrange(v, x, y, w, h) -> v ^ "->range(" ^ string_of_int x ^ ", " ^
                              string_of_int y ^ ", " ^
                              string_of_int w ^ ", " ^
                              string_of_int h ^ ")"

let string_of_vdecl var = (string_of_vartype var.vtype) ^ " " ^ var.vname

let string_of_vinit = function
   Iminit(v, e) -> string_of_vdecl v ^ " = " ^ string_of_img_expr e
  | Vinit(v, e) -> string_of_vdecl v ^ " = " ^ string_of_expr e
  | Immatrix3x3(v, r1, r2, r3) -> string_of_vdecl v ^ "[3][3] = " ^
                                  "{" ^ string_of_row3 r1 ^ ", "
                               ^ string_of_row3 r2 ^ ", "
                                                                ^ string_of_row3 r3 ^ "}"

let string_of_vdef = function
   VarDecl(v) -> string_of_vdecl v ^ ";\n"
  | Varinit(vi) -> string_of_vinit vi ^ ";\n"

let rec string_of_stmt = function
   Block(stmts) ->
    "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Imexpr(imexpr) -> string_of_img_expr imexpr
  | Imread(i, p) -> i ^ " = imread(" ^ p ^ ");\n";
  | Imwrite(i, p) -> i ^ " = imwrite(" ^ p ^ ");\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
```

```
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
     string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
     "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
     string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Break -> "break;\n"

let string_of_fdecl fdecl =
  (string_of_vartype fdecl.freturn) ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_vdecl fdecl.fparams) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdef fdecl.flocals) ^
  String.concat "" (List.map string_of_stmt fdecl.fbody) ^
  "}\n"

let string_of_program (global_vars, funcs) =
  String.concat "" (List.map string_of_vdef global_vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

8.4 PARSER.MLY

Parser.mly

```
%{(*
   Columbia University

   PLT 4115 Course - SIP Compiler Project

   Under the Supervision of: Prof. Stephen A. Edwards
   Name: Emad Barsoum
   UNI: eb2871

   parser.mly for SIP language
*)
open Ast %}

%token READ WRITE
%token PLUS MINUS TIMES DIVIDES MOD CONV ASSIGN
%token NEQ LT LEQ GT GEQ EQ AND OR NOT QUES
%token BITAND BITOR BITNOT
%token LPAREN RPAREN LBRACKET RBRACKET LBRACE RBRACE SEMICOLON COLON COMMA SEMI
%token ARROW RANGE
%token BOOL INT UINT FLOAT HIST IMAGE
%token TRUE FALSE IF ELSE FOR IN WHILE RETURN BREAK FUN KERNEL
%token <bool> BLITERAL
%token <int> ILITERAL
```

```
%token <float> FLITERAL
%token <string> ID
%token <string> SLITERAL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN

%left EQ NEQ
%left LT LEQ GT GEQ
%left AND OR
%right NOT

%left PLUS MINUS
%left TIMES DIVIDES MOD
%right UMINUS

%left BITAND BITOR
%right BITNOT

%start program
%type <Ast.program> program

%%

program:
  /* nothing */ { [], [] }
 | program vdef  { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

vdef_list:
   /* nothing */    { [] }
  | vdef_list vdef { $2 :: $1 }

vdef:
   vdecl    { VarDecl($1) }
  | vinit    { Varinit($1) }

vdecl:
   basic_type ID SEMI            { { vname = $2; vtype = $1} }
  | img_type ID SEMI              { { vname = $2; vtype = $1} }

basic_type:
   BOOL  { Bool     }
  | INT   { Int      }
  | UINT  { UInt     }
```

```
  | FLOAT { Float    }

img_type:
   HIST  { Histogram }
  | IMAGE { Image    }

vinit:
   basic_type ID ASSIGN expr SEMI   { Vinit ({ vname = $2; vtype = $1}, $4) }
  | img_type ID ASSIGN img_expr SEMI { Iminit({ vname = $2; vtype = $1}, $4) }
  | img_type ID ASSIGN LBRACKET row3 row3 row3 RBRACKET SEMI { Immatrix3x3({ vname = $2; vtype
= Matrix3x3 }, $5, $6, $7) }

row3:
   LBRACKET expr COMMA expr COMMA expr RBRACKET  { Row($2, $4, $6) }

fdecl:
   FUN ID LPAREN formals_opt RPAREN ftype_opt LBRACE vdef_list stmt_list RBRACE
    { { fname    = $2;
                fparams  = $4;
                flocals  = List.rev $8;
                fbody    = List.rev $9;
                        freturn  = $6;
                        fgpu     = false } }
  | KERNEL ID LPAREN formals_opt RPAREN LBRACE vdef_list stmt_list RBRACE
    { { fname    = $2;
                fparams  = $4;
                flocals  = List.rev $7;
                fbody    = List.rev $8;
                        freturn  = Void;
                        fgpu     = true } }

formals_opt:
   /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
   fparam               { [$1] }
  | formal_list COMMA fparam { $3 :: $1 }

fparam:
   basic_type ID  { { vname = $2; vtype = $1} }
  | img_type ID    { { vname = $2; vtype = $1} }

ftype_opt:
  /* nothing */ { Void }
| ftype       { $1 }

ftype:
```

```
   BOOL  { Bool     }
  | INT   { Int      }
  | UINT  { UInt     }
  | FLOAT { Float    }

stmt_list:
   /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
   expr SEMI { Expr($1) }
  | img_expr { Imexpr($1) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | ID READ SLITERAL SEMI { Imread($1, $3) }
  | ID WRITE SLITERAL SEMI { Imwrite($1, $3) }
  | RETURN expr SEMI { Return($2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | BREAK { Break }

img_expr:
   ID IN LPAREN ID RPAREN SEMI { In($1, [Channel($1,$4)], []) }
  | ID IN LPAREN ID COMMA ID COMMA ID RPAREN SEMI { In($1, [Channel($1,$4); Channel($1,$6);
Channel($1,$8)], []) }
  | ID IN LPAREN ID RPAREN FOR LBRACE ID COLON expr RBRACE SEMI { In($1, [Channel($1,$4)],
[Assign($8 ^ "_out", $10)]) }
  | ID IN LPAREN ID COMMA ID COMMA ID RPAREN FOR LBRACE ID COLON expr RBRACE SEMI
     { In($1, [Channel($1,$4); Channel($1,$6); Channel($1,$8)], [Assign($12 ^ "_out", $14)]) }
  | ID IN LPAREN ID COMMA ID COMMA ID RPAREN FOR LBRACE ID COLON expr COMMA ID COLON
expr COMMA ID COLON expr RBRACE SEMI
     { In($1, [Channel($1,$4); Channel($1,$6); Channel($1,$8)], [Assign($12 ^ "_out", $14); Assign($16 ^
"_out", $18); Assign($20 ^ "_out", $22)]) }
  | ID CONV ID SEMI    { Imop($1, Conv, $3) }
  | ID ASSIGN img_expr { Imassign($1, $3) }
  | ID LBRACKET ILITERAL RANGE ILITERAL SEMI ILITERAL RANGE ILITERAL RBRACKET SEMI {
Imrange($1, $3, $7, $5, $9) }

expr_opt:
   /* nothing */ { Noexpr }
  | expr        { $1 }

expr:
   BLITERAL               { BoolLiteral($1) }
  | ILITERAL              { IntLiteral($1) }
  | FLITERAL               { FloatLiteral($1) }
```

```
  | SLITERAL                { StringLiteral($1) }
  | ID                    { Id($1) }
  | expr PLUS   expr          { Binop($1, Add,   $3) }
  | expr MINUS  expr          { Binop($1, Sub,   $3) }
  | expr TIMES  expr          { Binop($1, Mult,  $3) }
  | expr DIVIDES expr         { Binop($1, Div,   $3) }
  | expr MOD expr             { Binop($1, Mod,   $3) }
  | expr NEQ expr             { Binop($1, Neq,   $3) }
  | expr LT expr              { Binop($1, Lt,    $3) }
  | expr LEQ expr             { Binop($1, Leq,   $3) }
  | expr GT expr              { Binop($1, Gt,    $3) }
  | expr GEQ expr             { Binop($1, Geq,   $3) }
  | expr EQ expr              { Binop($1, Eq,    $3) }
  | expr AND expr             { Binop($1, And,   $3) }
  | expr OR expr              { Binop($1, Or,    $3) }
  | expr NOT expr             { Binop($1, Not,   $3) }
  | expr BITAND expr          { Binop($1, BitAnd,$3) }
  | expr BITOR expr           { Binop($1, BitOr, $3) }
  | expr BITNOT expr          { Binop($1, BitNot,$3) }
  | MINUS expr %prec UMINUS     { Unop(Neg, $2) }
  | ID ASSIGN expr            { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN          { Bracket($2) }
  | expr QUES expr COLON expr    { Ques($1, $3, $5) }
  | ID LBRACKET expr COMMA expr RBRACKET ARROW ID { Imaccessor($1, $3, $5, $8) }
  | ID ARROW ID               { Accessor($1, $3) }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

8.5 TRANSLATE.ML

Translate.ml

```
(*
    Columbia University

    PLT 4115 Course - SIP Compiler Project

    Under the Supervision of: Prof. Stephen A. Edwards
    Name: Emad Barsoum
```

```
   UNI: eb2871

   translate.ml for SIP language
*)

open Ast
open Printf

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
   function_decl : string StringMap.t;   (* Signature for each function *)
   global_var    : var_type StringMap.t; (* global variables and their types *)
   local_var     : var_type StringMap.t; (* locals + function params and their types *)
  }

(* Begining of the C++ file. *)
let cc_headers = "#include \"sip.h\"\n"        ^
            "using namespace Sip;\n\n"    ^
                                  "ClProgram g_clProgram;\n"    ^
                                        "Image g__sip_temp__;\n\n"

(* Begining of the OpenCL header, and a generic function for 3x3 filter *)
let cl_headers =
"__constant sampler_t sampler =  CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

__kernel void apply_filter(__read_only image2d_t in_image, __write_only image2d_t out_image,
__constant float* filter)
{
  const int2 pos = {get_global_id(0), get_global_id(1)};

  float4 sum = (float4)(0.0f);
  for (int y = -1; y <= 1; y++)
  {
     for (int x = -1; x <= 1; x++)
     {
        sum.x += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).x;
        sum.y += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).y;
        sum.z += filter[(y + 1) * 3 + (x + 1)] * read_imagef(in_image, sampler, pos + (int2)(x,y)).z;
     }
  }

  write_imagef (out_image, (int2)(pos.x, pos.y), sum);
}\n"

(* Return a string represntation of function signature *)
```

```
let fsig fdecl =
  fdecl.fname ^ "_" ^ String.concat "_" (List.map Ast.string_of_vdecl fdecl.fparams)

(* Extract the declared type from the fully qualified initializer *)
let vdecl_of_vinit = function
    Iminit(v, _) -> (v.vtype, v.vname)
  | Vinit(v, _) -> (v.vtype, v.vname)
  | Immatrix3x3(v, _, _, _) -> (v.vtype, v.vname)

(* Extract the declared type from variable definition. Variable definition is a declared variable
   or declared + initialized *)
let vdecl_of_vdef = function
    VarDecl(v) -> (v.vtype, v.vname)
  | Varinit(vi) -> vdecl_of_vinit vi

(* Some helper enum based on MicroC enum with some minor modifications *)
let rec enum_vdecl = function
    [] -> []
  | hd::tl -> (hd.vtype, hd.vname) :: enum_vdecl tl

let rec enum_vdef = function
    [] -> []
  | hd::tl -> (vdecl_of_vdef hd) :: enum_vdef tl

let rec enum_func = function
    [] -> []
 |  hd:: tl -> (fsig hd, hd.fname) :: enum_func tl

(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(* Translate the AST tree into a C++ program *)
let translate_to_cc (globals, functions) out_name =

  (* Allocate "addresses" for each global variable *)
  let global_variables = string_map_pairs StringMap.empty (enum_vdef globals) in
  let function_decls = string_map_pairs StringMap.empty (enum_func functions) in

  (* Translate a function in AST form into a list of bytecode statements *)
  let translate env fdecl =
    let local_var = enum_vdef fdecl.flocals
    and formal_var = enum_vdecl fdecl.fparams in
    let env = { env with local_var = string_map_pairs StringMap.empty (local_var @ formal_var) } in
    let dynamic_var = ref StringMap.empty in

    let rec expr e =
            (match e with
```

```
    BoolLiteral(l) -> string_of_bool l
    | IntLiteral(l) -> string_of_int l
    | FloatLiteral(l) -> string_of_float l
    | StringLiteral(l) -> l
    | Id(s) ->
                    if ((StringMap.mem s env.local_var) || (StringMap.mem s env.global_var) ||
(StringMap.mem s !dynamic_var))
        then s
                        else raise (Failure ("undeclared variable " ^ s))
    | Unop(o, e) ->
      (match o with
     Neg -> "-") ^ expr e
    | Binop (e1, op, e2) ->
                    expr e1 ^ " " ^
      (match op with
            Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/" | Mod -> "%"
      | Neq -> "!=" | Lt -> "<" | Leq -> "<=" | Gt -> ">" | Geq -> ">=" | Eq -> "=="
      | And -> "&&" | Or -> "||" | Not -> "!"
      | BitAnd -> "&" | BitOr -> "|" | BitNot -> "~") ^ " " ^
      expr e2
    | Assign (s, e) ->
                    if ((StringMap.mem s env.local_var) || (StringMap.mem s env.global_var) ||
(StringMap.mem s !dynamic_var))
                     then s ^ " = " ^ expr e
                        else raise (Failure ("undeclared variable " ^ s))
    | Call (fname, actuals) ->
                    if (StringMap.mem fname env.function_decl)
                     then fname ^ "(" ^ String.concat ", " (List.map expr (List.rev actuals)) ^ ")"
                        else (if ((String.compare fname "writeln") == 0) then
          (if ((List.length actuals) == 1) then
             "std::cout << " ^ expr (List.hd actuals) ^ " << std::endl"
          else raise (Failure ("writeln takes only one argument")))
        else
          raise (Failure ("undefined function " ^ fname)))
          | Ques (e1, e2, e3) -> "(" ^ expr e1 ^ ") ? " ^
            expr e2 ^ ":" ^ expr e3
    | Bracket (e) -> "(" ^ expr e ^ ")"
    | Imaccessor (i, r, c, a) -> i ^ "(" ^ expr r ^ "," ^ expr c ^ ")->" ^ a
    | Accessor (i, a) ->
                    if ((StringMap.mem i env.local_var) || (StringMap.mem i env.global_var) ||
(StringMap.mem i !dynamic_var))
        then i ^ "." ^ (match a with
                    "Width" -> "width()"
                  | "Height" -> "height()"
                  | _ -> raise (Failure ("Invalid attribute " ^ a)))
                        else raise (Failure ("undeclared variable " ^ i))
    | Noexpr -> "")
```

```
  in let add_channels_var c =
    if ((List.length c) != 0)
    then
     (String.concat "" (List.map (fun f ->
                    (* print_string ((Ast.get_channel f) ^ "\n"); *)
                    ignore(dynamic_var := StringMap.add (Ast.get_channel f) Ast.UInt !dynamic_var);
                    ignore(dynamic_var := StringMap.add ((Ast.get_channel f) ^ "_out") Ast.UInt
!dynamic_var);
                    "") c))
          else raise (Failure ("empty channel list in an \"In\" statement "))

  in let expand_channels c =
             if ((List.length c) != 0)
             then
              (String.concat "" (List.map (fun f ->
                           "       unsigned int " ^ Ast.get_channel f ^ " = " ^ Ast.string_of_channel f ^
";\n" ^
                        "       unsigned int " ^ Ast.get_channel f ^ "_out = " ^ Ast.string_of_channel f ^
";\n") c))
                   else raise (Failure ("empty channel list in an \"In\" statement "))

   in let rec img_expr = function
              Imop(s, o, k) ->
                           if ((StringMap.mem s env.local_var) || (StringMap.mem s env.global_var))
then begin
                               if ((StringMap.mem k env.local_var) || (StringMap.mem k
env.global_var)) then
                                 "g_clProgram.ApplyFilter(" ^ s ^ ", g__sip_temp__, (float*)&" ^ k ^
");\n"
                                          else "g_clProgram.RunKernel(" ^ s ^ ", g__sip_temp__,\"" ^ k
^ "\");\n"
                                 end
                                 else raise (Failure ("undeclared variable " ^ s))
            | In (v, a, el) -> ignore(add_channels_var a); (* To force the order, we need to add the
variable before evluating the expr. *)
         "g__sip_temp__.clone(" ^ v ^ ");\n" ^
         "for (int row = 0; row <" ^ v ^ ".height(); ++row)\n{\n"       ^
         "    for (int col = 0; col <" ^ v ^ ".width(); ++col)\n    {\n" ^
         expand_channels a ^ "\n" ^
                String.concat ";\n" (List.map expr el) ^ ";\n\n"  ^
                "       g__sip_temp__(row, col)->Red   = (char)red_out;\n"   ^
                "       g__sip_temp__(row, col)->Green = (char)green_out;\n" ^
                "       g__sip_temp__(row, col)->Blue  = (char)blue_out;\n"  ^
                           "       g__sip_temp__(row, col)->Alpha = " ^ v ^ "(row, col)->Alpha;\n" ^
                           "    }\n}\n"
      | Imassign(v, e) -> img_expr e ^ "\n" ^ v ^ " = g__sip_temp__;\n"
      | Imrange(v, x, y, w, h) -> v ^ ".copyRangeTo(" ^ string_of_int x ^ ", " ^
                                  string_of_int y ^ ", " ^
```

```
                                    string_of_int w ^ ", " ^
                                    string_of_int h ^ ", " ^
                                    "g__sip_temp__);"

 in let rec stmt = function
           Block(sl) ->
     String.concat "" (List.map stmt sl) ^ "\n"
         | Expr(e) -> expr e ^ ";\n";
         | Imexpr(imexpr) -> img_expr imexpr
         | Imread(i, p) -> i ^ ".read(" ^ p ^ ");\n";
         | Imwrite(i, p) -> i ^ ".write(" ^ p ^ ");\n";
         | Return(e) -> "return " ^ expr e ^ ";\n";
         | If(e, s, Block([])) -> "if (" ^ expr e ^ ")\n{\n" ^ stmt s ^ "}\n"
  | If(e, s1, s2) ->  "if (" ^ expr e ^ ")\n{\n" ^
           stmt s1 ^ "}\nelse\n{\n" ^ stmt s2 ^ "}\n"
         | For(e1, e2, e3, s) ->
           "for (" ^ expr e1  ^ " ; " ^ expr e2 ^ " ; " ^
           expr e3  ^ ") \n{\n" ^ stmt s ^ "}\n"
         | While(e, s) -> "while (" ^ expr e ^ ") \n{\n" ^ stmt s ^ "}\n"
  | Break -> "break;\n"

 in let vartype = function
    Void -> "void"
         | Bool -> "bool"
         | Int -> "int"
         | UInt -> "unsigned int"
         | Float -> "float"
  | Matrix3x3 -> "float"
         | Histogram -> "Histogram"
         | Image -> "Image"

 in let func_params_type = function
    Void -> "void"
  | Bool -> "bool"
  | Int -> "int"
  | UInt -> "unsigned int"
  | Float -> "float"
  | Matrix3x3 -> "float"
  | Histogram -> "Histogram&"
  | Image -> "Image&"

in  if (fdecl.fgpu) then ""
   else begin
      (if ((String.compare fdecl.fname "main") == 0)
      then "int main()\n{\n" ^ "    g_clProgram.CompileClFile(\"./" ^ out_name ^ ".cl\");\n\n"
      else (vartype fdecl.freturn) ^ " " ^ fdecl.fname
      ^ if ((List.length fdecl.fparams) != 0)
        then ("("
```

```
        ^ func_params_type (List.hd fdecl.fparams).vtype ^ " " ^ (List.hd fdecl.fparams).vname ^ " "
        ^ String.concat "" (List.map (fun formal -> ", " ^ func_params_type formal.vtype ^ " " ^
formal.vname) (List.tl fdecl.fparams)) ^ ")\n{\n")
          else "()\n{\n"
          )
        ^ String.concat "" (List.map Ast.string_of_vdef (List.rev fdecl.flocals)) ^ "\n"
        ^ stmt (Block fdecl.fbody) ^ "\n" ^
        if ((String.compare fdecl.fname "main") == 0)
            then "    return 0;\n}\n"
        else "\n}\n"
      end

  in let env = {
        function_decl = function_decls;
                        global_var = global_variables;
                        local_var = StringMap.empty } in

  (* Code executed to start the program *)
  let _ = try
    (StringMap.find "main" function_decls)
  with Not_found -> raise (Failure ("no \"main\" function"))

  (* Compile the functions *)
  in cc_headers ^
    String.concat "" (List.map Ast.string_of_vdef (List.rev globals)) ^ "\n" ^
           (String.concat "\n" (List.map (translate env) (List.rev functions))) ^ "\n"

(* Translate the AST tree into a OpenCL shader program *)
let translate_to_cl (globals, functions) out_name =

  (* Keep track of global variables *)
  let function_decls = string_map_pairs StringMap.empty (enum_func functions) in

  (* Translate a function in AST form into a C++ statements *)
  let translate env fdecl =
    let local_var = enum_vdef fdecl.flocals
    and formal_var = enum_vdecl fdecl.fparams in
    let env = { env with local_var = string_map_pairs StringMap.empty (local_var @ formal_var) } in

    let rec expr e =
            (match e with
      BoolLiteral(l) -> string_of_bool l
      | IntLiteral(l) -> string_of_int l
      | FloatLiteral(l) -> string_of_float l
      | StringLiteral(l) -> l
      | Id(s) ->
                      if (StringMap.mem s env.local_var)
          then s
```

```
                        else raise (Failure ("undeclared variable " ^ s))
    | Unop(o, e) ->
      (match o with
     Neg -> "-") ^ expr e
    | Binop (e1, op, e2) ->
                    expr e1 ^ " " ^
      (match op with
            Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/" | Mod -> "%"
      | Neq -> "!=" | Lt -> "<" | Leq -> "<=" | Gt -> ">" | Geq -> ">=" | Eq -> "=="
      | And -> "&&" | Or -> "||" | Not -> "!"
      | BitAnd -> "&" | BitOr -> "|" | BitNot -> "~") ^ " " ^
      expr e2
    | Assign (s, e) ->
                    if (StringMap.mem s env.local_var)
                     then s ^ " = " ^ expr e
                        else raise (Failure ("undeclared variable " ^ s))
    | Call (fname, actuals) -> raise (Failure ("Function call aren't supported in Kernel function " ^
fname))
          | Ques (e1, e2, e3) -> "(" ^ expr e1 ^ ") ? " ^
            expr e2 ^ ":" ^ expr e3
    | Bracket (e) -> "(" ^ expr e ^ ")"
    | Imaccessor (i, r, c, a) -> "read_imagef(" ^ i ^ ", sampler, pos + (int2)(" ^
                        expr c ^ "," ^ expr r ^ "))." ^
                        (match a with
                           "Red" -> "x"
                         | "Green" -> "y"
                         | "Blue" -> "z"
                         | _ -> raise (Failure ("Invalid channel " ^ a)))
    | Accessor(i, a) -> raise (Failure ("Accessor is not supported in a kernel function."))
    | Noexpr -> "")

   in let rec stmt = function
            Block(sl) ->
       String.concat "" (List.map stmt sl) ^ "\n"
          | Expr(e) -> expr e ^ ";\n";
          | Imexpr(imexpr) -> raise (Failure ("Image expression is not supported in a kernel function."))
          | Imread(i, p) -> raise (Failure ("Read operator is not supported in a kernel function."))
          | Imwrite(i, p) -> raise (Failure ("Write operator is not supported in a kernel function."))
          | Return(e) -> "return " ^ expr e ^ ";\n";
          | If(e, s, Block([])) -> "if (" ^ expr e ^ ")\n{\n" ^ stmt s ^ "}\n"
    | If(e, s1, s2) ->  "if (" ^ expr e ^ ")\n{\n" ^
            stmt s1 ^ "}\nelse\n{\n" ^ stmt s2 ^ "}\n"
          | For(e1, e2, e3, s) ->
            "for (" ^ expr e1  ^ " ; " ^ expr e2 ^ " ; " ^
            expr e3  ^ ") {\n" ^ stmt s ^ "}\n"
          | While(e, s) -> "while (" ^ expr e ^ ") " ^ "{\n" ^ stmt s ^ "}\n"
    | Break -> "break;\n"
```

```
   (* Return OpenCL specific type only *)
   in let func_params_type = function
      Image -> "image2d_t"
     | _ -> raise (Failure ("Only image type is supported in a kernel function."))

  (* Translate only kernel function. *)
  in  if (fdecl.fgpu) then begin
     (if (fdecl.freturn != Void) then raise (Failure ("Kernel function can't return any value."))
      else
        "__kernel void " ^ fdecl.fname
        ^ if ((List.length fdecl.fparams) != 2) then raise (Failure ("Kernel function must takes 2 image
types as argument."))
         else ("(__read_only "
        ^ func_params_type (List.hd fdecl.fparams).vtype ^ " " ^ (List.hd fdecl.fparams).vname ^ " "
        ^ String.concat "" (List.map (fun formal -> ", __write_only " ^ func_params_type formal.vtype ^ "
" ^ formal.vname) (List.tl fdecl.fparams))
        ^ ")\n{\n    const int2 pos = {get_global_id(0), get_global_id(1)};\n"
        ^ String.concat "" (List.map Ast.string_of_vdef (List.rev fdecl.flocals)) ^ "\n"
        ^ stmt (Block fdecl.fbody) ^ "\n" ^ "    float4 _out_ = {red_out, green_out, blue_out, 0.0f};\n"
        ^ "    write_imagef (" ^ (List.hd (List.tl fdecl.fparams)).vname
        ^ ", (int2)(pos.x, pos.y), _out_);" ^ "\n}\n"))
     end
  else ""

  in let env = {
       function_decl = function_decls;
                       global_var = StringMap.empty;
                       local_var = StringMap.empty }

  (* Compile the functions *)
  in cl_headers ^
           (String.concat "\n" (List.map (translate env) (List.rev functions))) ^ "\n"
```

8.6 MAKEFILE.ML

Makefile.ml

```
(*
   Columbia University

   PLT 4115 Course - SIP Compiler Project

   Under the Supervision of: Prof. Stephen A. Edwards
   Name: Emad Barsoum
   UNI: eb2871
```

```
   makefile.ml generate makefile for the output C++ and OpenCL code
*)

open Printf

let fwrite name content =
            let out = open_out name in
              fprintf out "%s\n" content;
              close_out out

let string_of_makefile t =
  "SHELLNAME := $(shell uname -s)\n\n" ^
  "TARGET = " ^ t ^ ".out\n" ^
  "OBJS = " ^ t ^ ".o sip.o EasyBMP.o\n" ^
  "CC = g++\n" ^
  "CFLAGS = -Wall -O3\n" ^
  "LFLAGS = -Wall\n\n" ^
  "ifeq ($(SHELLNAME), Darwin)\n" ^
  "\tLIBS = -framework OpenCL\n" ^
  "else\n" ^
  "\tLIBS = -lOpenCL\n" ^
  "endif\n\n" ^
  "$(TARGET) : $(OBJS)\n" ^
  "\t$(CC) $(LFLAGS) $(OBJS) $(LIBS) -o $@\n\n" ^
  t ^ ".o: " ^ t ^ ".cpp sip.h\n" ^
  "\tg++ $(CFLAGS) -c " ^ t ^ ".cpp\n\n" ^
  "sip.o: sip.cpp EasyBMP.h\n" ^
  "\tg++ $(CFLAGS) -c sip.cpp\n\n" ^
  "EasyBMP.o: EasyBMP.cpp EasyBMP*.h\n" ^
  "\tg++ $(CFLAGS) -c EasyBMP.cpp\n\n" ^
  "clean:\n" ^
  "\trm *.o\n\n" ^
  "cleanall:\n" ^
  "\trm *.o $(TARGET)\n"

let gen_makefile t =
  let mf = string_of_makefile t in
    fwrite "./out/Makefile" mf
```

8.7 MAKEFILE

Makefile

```
OBJS = scanner.cmo ast.cmo parser.cmo translate.cmo makefile.cmo sip.cmo

TARFILES = Makefile scanner.mll ast.ml parser.mly translate.ml makefile.ml sip.ml
```

```
sip : $(OBJS)
	ocamlc -o sip $(OBJS)

parser.ml parser.mli: parser.mly
	ocamlyacc -v parser.mly

scanner.ml : scanner.mll
	ocamllex scanner.mll

%.cmo : %.ml
	ocamlc -c $<

%.cmi : %.mli
	ocamlc -c $<

.PHONY : clean
clean :
	rm -f scanner.ml parser.ml parser.mli *.cmo *.cmi *.out *.diff

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
makefile.cmo:
makefile.cmx:
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
translate.cmo: ast.cmo
translate.cmx: ast.cmx
sip.cmo: scanner.cmo parser.cmi ast.cmo translate.cmo makefile.cmo
sip.cmx: scanner.cmx parser.cmx ast.cmx translate.cmx makefile.cmx
```

8.8 TESTALL.SH

Testall.sh

```
#!/bin/sh

SIPC="./sip"

# Set time limit for all operations
ulimit -t 30
```

```
globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
   echo "Usage: testall.sh [options] [.sip files]"
   echo "-k    Keep intermediate files"
   echo "-h    Print this help"
   exit 1
}

SignalError() {
   if [ $error -eq 0 ] ; then
          echo "FAILED"
          error=1
   fi
   echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
   generatedfiles="$generatedfiles $3"
   echo diff -b $1 $2 ">" $3 1>&2
   diff -b "$1" "$2" > "$3" 2>&1 || {
          SignalError "$1 differs"
          echo "FAILED $1 differs from $2" 1>&2
   }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
   echo $* 1>&2
   eval $* || {
          SignalError "$1 failed on $*"
          return 1
   }
}

Check() {
   error=0
   basename=`echo $1 | sed 's/.*\\///
                        s/.sip//'`
   reffile=`echo $1 | sed 's/.sip$//'`
```

```
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.cpp.out" &&
    Run "$SIPC" "-tcc" $1 ">" ${basename}.cpp.out &&
    Compare ${basename}.cpp.out ${reffile}.cppout.cpp ${basename}.cpp.diff

    generatedfiles="$generatedfiles ${basename}.cl.out" &&
    Run "$SIPC" "-tcl" $1 ">" ${basename}.cl.out &&
    Compare ${basename}.cl.out ${reffile}.clout.cl ${basename}.cl.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
            if [ $keep -eq 0 ] ; then
               rm -f $generatedfiles
            fi
            echo "OK"
            echo "###### SUCCESS" 1>&2
    else
            echo "###### FAILED" 1>&2
            globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
            k) # Keep intermediate files
               keep=1
               ;;
            h) # Help
               Usage
               ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
```

```
   files="tests/good/test-*.sip"
fi

for file in $files
do
   case $file in
          *test-*)
             Check $file 2>> $globallog
             ;;
          *fail-*)
             CheckFail $file 2>> $globallog
             ;;
          *)
             echo "unknown file type $file"
             globalerror=1
             ;;
   esac
done

exit $globalerror
```

8.9 SIP.H

Sip.h

```
/*
   Columbia University

   PLT 4115 Course - SIP Compiler Project

   Under the Supervision of: Prof. Stephen A. Edwards
   Name: Emad Barsoum
   UNI: eb2871

   sip.h
*/

#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif
```

```
#include "EasyBMP.h"

using namespace std;

#ifndef SIP_H
#define SIP_H

#define MEM_SIZE (128)
#define MAX_SOURCE_SIZE (0x100000)

namespace Sip
{
	class Image;

   class ClProgram
   {
   public:
      ClProgram();
      ~ClProgram();

      void CompileClFile(const char* filename);
      void RunKernel(Image& in_image, Image& out_image, const char* kernelName);
      void ApplyFilter(Image& in_image, Image& out_image, float* filter);

   private:
      void Init();
      void Uninit();

   private:
      cl_command_queue _commandQueue;
      cl_device_id     _deviceId;
      cl_context       _context;
      cl_program       _program;
      cl_platform_id   _platformId;
   };

   class Image
   {
   public:
                  Image();
                  Image(const Image& img);

      int width();
      int height();

      void clone(Image& img);
      void copyRangeTo(unsigned int offsetX,
```

```
                        unsigned int offsetY,
                        unsigned int width,
                        unsigned int height,
                        Image& img);

    RGBApixel* operator()(int i,int j);
    Image& operator=(Image &rhs);

    void read(const char* path);
    void write(const char* path);

  private:
    BMP _image;
  };

  class Histogram
  {
      public:
          Histogram();
          Histogram(Image& img);

          unsigned int operator()(int bin, int color);

      private:
          unsigned int _red[256];
          unsigned int _green[256];
          unsigned int _blue[256];
      };
}

#endif
```

8.10 SIP.CPP

Sip.cpp

```
/*
  Columbia University

  PLT 4115 Course - SIP Compiler Project

  Under the Supervision of: Prof. Stephen A. Edwards
  Name: Emad Barsoum
  UNI: eb2871

  sip.cpp
```

```
*/

#include "sip.h"

using namespace Sip;

ClProgram::ClProgram() :  _commandQueue(NULL),
                    _deviceId(NULL),
                    _context(NULL),
                    _program(NULL),
                    _platformId(NULL)
{
   Init();
}

ClProgram::~ClProgram()
{
   Uninit();
}

void ClProgram::Init()
{
   cl_int ret = 0;
   cl_uint platformCount = 0;
   cl_uint deviceCount = 0;

   ret = clGetPlatformIDs(1, &_platformId, &platformCount);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clGetPlatformIDs: " << ret << endl;
      return;
   }

   ret = clGetDeviceIDs(_platformId, CL_DEVICE_TYPE_GPU, 1, &_deviceId, &deviceCount);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clGetDeviceIDs: " << ret << endl;
      return;
   }

   _context = clCreateContext(NULL, 1, &_deviceId, NULL, NULL, &ret);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clCreateContext: " << ret << endl;
      return;
   }

   _commandQueue =  clCreateCommandQueue(_context, _deviceId, 0, &ret);
```

```
    if (ret != CL_SUCCESS)
    {
        cout << "Error: clCreateCommandQueue: " << ret << endl;

        clReleaseContext(_context);
        _context = NULL;
        return;
    }
}

void ClProgram::Uninit()
{
    if (_program != NULL)
    {
        clReleaseProgram(_program);
        _program = NULL;
    }

    if (_commandQueue != NULL)
    {
        clReleaseCommandQueue(_commandQueue);
        _commandQueue = NULL;
    }

    if (_context != NULL)
    {
        clReleaseContext(_context);
        _context = NULL;
    }
}

void ClProgram::CompileClFile(const char* filename)
{
    cl_int ret = 0;
    FILE *file   = NULL;
    char *source = NULL;
    size_t size  = 0;

    file = fopen(filename, "r");
    if (!file)
    {
        cout << "Couldn't open: " << filename << endl;
        return;
    }

    source = new char[MAX_SOURCE_SIZE];
    size = fread(source, 1, MAX_SOURCE_SIZE, file);
```

```
    if (_program != NULL)
    {
        clReleaseProgram(_program);
        _program = NULL;
    }

            _program = clCreateProgramWithSource(_context, 1, (const char **)&source, 0, &ret);
    if (ret != CL_SUCCESS)
    {
        cout << "Error: clCreateProgramWithSource :" << ret << endl;
        return;
    }

            ret = clBuildProgram(_program, 1, &_deviceId, NULL, NULL, NULL);
    if ((ret != CL_SUCCESS) || (ret == CL_BUILD_PROGRAM_FAILURE))
    {
        cout << "Error: clBuildProgram: " << ret << endl;
        return;
    }

    delete[] source;
    fclose(file);
}

void ClProgram::RunKernel(Image& in_image, Image& out_image, const char* kernelName)
{
    cl_int ret = 0;
            cl_image_format img_fmt;

            img_fmt.image_channel_order = CL_RGBA;
            img_fmt.image_channel_data_type = CL_UNORM_INT8;

            cl_mem imageSrc;
    cl_mem imageDst;

            size_t width = in_image.width();
    size_t height = in_image.height();

            char* input  = new char[width * height * 4];
            char* output = new char[width * height * 4];

    out_image.clone(in_image);

    for (size_t row = 0; row < height; ++row)
    {
        for (size_t col = 0; col < width; ++col)
        {
            input[row * 4 * width + 4 * col    ] = (char)in_image(row, col)->Red;
```

```
        input[row * 4 * width + 4 * col + 1] = (char)in_image(row, col)->Green;
        input[row * 4 * width + 4 * col + 2] = (char)in_image(row, col)->Blue;
        input[row * 4 * width + 4 * col + 3] = (char)in_image(row, col)->Alpha;
      }
   }

   imageSrc = clCreateImage2D(_context, CL_MEM_READ_ONLY, &img_fmt, width, height, 0, 0, &ret);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: imageSrc clCreateImage2D: " << ret << endl;
      return;
   }

         imageDst = clCreateImage2D(_context, CL_MEM_READ_WRITE, &img_fmt, width, height, 0, 0,
&ret);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: imageDst clCreateImage2D: " << ret << endl;
      return;
   }

         cl_event clevent[5];

         size_t origin[] = {0, 0, 0}; // Defines the offset in pixels in the image from where to write.
         size_t region[] = {width, height, 1}; // Size of object to be transferred
         ret = clEnqueueWriteImage(_commandQueue, imageSrc, CL_TRUE, origin, region, 0, 0, input,
0, NULL, &clevent[0]);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clEnqueueWriteImage: " << ret << endl;
      return;
   }

         cl_kernel kernel = clCreateKernel(_program, kernelName, &ret);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clCreateKernel: " << ret << endl;
      return;
   }

         ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&imageSrc);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clSetKernelArg: " << ret << endl;
      return;
   }

         ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&imageDst);
```

```
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clSetKernelArg: " << ret << endl;
      return;
   }

         size_t GWSize[] = {width, height, 1};
         ret = clEnqueueNDRangeKernel(_commandQueue, kernel, 2, NULL, GWSize, NULL, 1, clevent,
&clevent[1]);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clEnqueueNDRangeKernel: " << ret << endl;
      return;
   }

   ret = clEnqueueReadImage(_commandQueue, imageDst, CL_TRUE, origin, region, 0, 0, output, 2,
clevent, &clevent[2]);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clEnqueueReadImage: " << ret << endl;
      return;
   }

   for (size_t row = 0; row < height; ++row)
   {
      for (size_t col = 0; col < width; ++col)
      {
         out_image(row, col)->Red   = output[row * 4 * width + 4 * col    ];
         out_image(row, col)->Green = output[row * 4 * width + 4 * col + 1];
         out_image(row, col)->Blue  = output[row * 4 * width + 4 * col + 2];
         out_image(row, col)->Alpha = output[row * 4 * width + 4 * col + 3];
      }
   }

   delete[] input;
   delete[] output;

   clReleaseMemObject(imageDst);
   clReleaseMemObject(imageSrc);
   clReleaseKernel(kernel);
}

void ClProgram::ApplyFilter(Image& in_image, Image& out_image, float* filter)
{
   cl_int ret = 0;
         cl_image_format img_fmt;

         img_fmt.image_channel_order = CL_RGBA;
```

```
        img_fmt.image_channel_data_type = CL_UNORM_INT8;

        cl_mem imageSrc;
  cl_mem imageDst;
        cl_mem imageFilter;

        size_t width = in_image.width();
  size_t height = in_image.height();

        char* input  = new char[width * height * 4];
        char* output = new char[width * height * 4];
        float* filterWeights = new float[9];

        memcpy((void*)filterWeights, (const void*)filter, 9 * sizeof(float));

  out_image.clone(in_image);

  for (size_t row = 0; row < height; ++row)
  {
    for (size_t col = 0; col < width; ++col)
    {
      input[row * 4 * width + 4 * col    ] = (char)in_image(row, col)->Red;
      input[row * 4 * width + 4 * col + 1] = (char)in_image(row, col)->Green;
      input[row * 4 * width + 4 * col + 2] = (char)in_image(row, col)->Blue;
      input[row * 4 * width + 4 * col + 3] = (char)in_image(row, col)->Alpha;
    }
  }

  imageSrc = clCreateImage2D(_context, CL_MEM_READ_ONLY, &img_fmt, width, height, 0, 0, &ret);
  if (ret != CL_SUCCESS)
  {
    cout << "Error: imageSrc clCreateImage2D: " << ret << endl;
    return;
  }

        imageDst = clCreateImage2D(_context, CL_MEM_READ_WRITE, &img_fmt, width, height, 0, 0,
&ret);
  if (ret != CL_SUCCESS)
  {
    cout << "Error: imageDst clCreateImage2D: " << ret << endl;
    return;
  }

  imageFilter = clCreateBuffer(_context, CL_MEM_READ_ONLY, 9 * sizeof(float), 0, &ret);
  if (ret != CL_SUCCESS)
  {
    cout << "Error: imageFilter clCreateBuffer: " << ret << endl;
    return;
```

```
   }

        cl_event clevent[5];

        size_t origin[] = {0, 0, 0}; // Defines the offset in pixels in the image from where to write.
        size_t region[] = {width, height, 1}; // Size of object to be transferred
        ret = clEnqueueWriteImage(_commandQueue, imageSrc, CL_TRUE, origin, region, 0, 0, input,
0, NULL, &clevent[0]);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clEnqueueWriteImage: " << ret << endl;
      return;
   }

   ret = clEnqueueWriteBuffer(_commandQueue, imageFilter, CL_TRUE, 0, 9 * sizeof(float),
filterWeights, 0, NULL, &clevent[1]);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clEnqueueWriteBuffer: " << ret << endl;
      return;
   }

        cl_kernel kernel = clCreateKernel(_program, "apply_filter", &ret);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clCreateKernel: " << ret << endl;
      return;
   }

        ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&imageSrc);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clSetKernelArg imageSrc: " << ret << endl;
      return;
   }

        ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&imageDst);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clSetKernelArg imageDst: " << ret << endl;
      return;
   }

        ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&imageFilter);
   if (ret != CL_SUCCESS)
   {
      cout << "Error: clSetKernelArg filter: " << ret << endl;
      return;
```

```
    }

        size_t GWSize[] = {width, height, 1};
        ret = clEnqueueNDRangeKernel(_commandQueue, kernel, 2, NULL, GWSize, NULL, 1, clevent,
&clevent[2]);
    if (ret != CL_SUCCESS)
    {
        cout << "Error: clEnqueueNDRangeKernel: " << ret << endl;
        return;
    }

    ret = clEnqueueReadImage(_commandQueue, imageDst, CL_TRUE, origin, region, 0, 0, output, 2,
clevent, &clevent[3]);
    if (ret != CL_SUCCESS)
    {
        cout << "Error: clEnqueueReadImage: " << ret << endl;
        return;
    }

    for (size_t row = 0; row < height; ++row)
    {
        for (size_t col = 0; col < width; ++col)
        {
            out_image(row, col)->Red   = output[row * 4 * width + 4 * col    ];
            out_image(row, col)->Green = output[row * 4 * width + 4 * col + 1];
            out_image(row, col)->Blue  = output[row * 4 * width + 4 * col + 2];
            out_image(row, col)->Alpha = output[row * 4 * width + 4 * col + 3];
        }
    }

    delete[] filterWeights;
    delete[] input;
    delete[] output;

    clReleaseMemObject(imageDst);
    clReleaseMemObject(imageSrc);
    clReleaseKernel(kernel);
}

Image::Image()
{}

Image::Image(const Image& img) : _image(const_cast<BMP&>(img._image))
{}

void Image::read(const char* path)
{
    _image.ReadFromFile(path);
```

```
}

void Image::write(const char* path)
{
   _image.WriteToFile(path);
}

int Image::width()
{
   return _image.TellWidth();
}

int Image::height()
{
   return _image.TellHeight();
}

void Image::clone(Image& img)
{
          if (this == &img)
          {
             return;
          }

          _image.SetSize(img._image.TellWidth(), img._image.TellHeight());
          _image.SetBitDepth(img._image.TellBitDepth());
}

void Image::copyRangeTo(unsigned int offsetX,
                  unsigned int offsetY,
                  unsigned int width,
                  unsigned int height,
                  Image& img)
{
   if (((offsetX + width) > (unsigned int)_image.TellWidth()) ||
      ((offsetY + height) > (unsigned int)_image.TellHeight()))
   {
      cout << "Invalid image range..." << endl;
      return;
   }

          img._image.SetSize(width, height);
          img._image.SetBitDepth(_image.TellBitDepth());

   for (size_t row = 0; row < height; ++row)
   {
      for (size_t col = 0; col < width; ++col)
      {
```

```
        img(row, col)->Red   = _image(offsetY + row, offsetX + col)->Red;
        img(row, col)->Green = _image(offsetY + row, offsetX + col)->Green;
        img(row, col)->Blue  = _image(offsetY + row, offsetX + col)->Blue;
        img(row, col)->Alpha = _image(offsetY + row, offsetX + col)->Alpha;
      }
   }
}

RGBApixel* Image::operator()(int i,int j)
{
   return _image(i, j);
}

Image& Image::operator=(Image &rhs)
{
          if (this == &rhs)
          {
             return *this;
          }

          clone(rhs);

   for (int row = 0; row < height(); ++row)
   {
      for (int col = 0; col < width(); ++col)
      {
         _image(row, col)->Red   = rhs(row, col)->Red;
         _image(row, col)->Green = rhs(row, col)->Green;
         _image(row, col)->Blue  = rhs(row, col)->Blue;
         _image(row, col)->Alpha = rhs(row, col)->Alpha;
      }
   }

   return *this;
}

Histogram::Histogram()
{
   memset((void*)_red, 0, sizeof(_red));
   memset((void*)_green, 0, sizeof(_green));
   memset((void*)_blue, 0, sizeof(_blue));
}

Histogram::Histogram(Image& img)
{
   memset((void*)_red, 0, sizeof(_red));
   memset((void*)_green, 0, sizeof(_green));
   memset((void*)_blue, 0, sizeof(_blue));
```

```
   for (int row = 0; row < img.height(); ++row)
   {
      for (int col = 0; col < img.width(); ++col)
      {
         _red[img(row, col)->Red]++;
         _green[img(row, col)->Green]++;
         _blue[img(row, col)->Blue]++;
      }
   }
}

unsigned int Histogram::operator()(int bin, int color)
{
   if ((bin < 0) || (bin >= 256) || (color < 0) || (color > 2))
   {
      cout << "Invalid argument..." << endl;
      return 0;
   }

   switch (color)
   {
      case 0:
      return _red[bin];

      case 1:
      return _green[bin];

      case 2:
      return _blue[bin];
   }

   return 0;
}
```

9 REFERENCES

1. http://halide-lang.org/
2. http://research.microsoft.com/apps/pubs/default.aspx?id=144767
3. “The C Programming Language, Second Edition. Prentice-Hall, 1988”, B. W. Kernighan and D. Ritchie.
4. http://easybmp.sourceforge.net/
5. http://www.khronos.org/opencl/
6. Black duck image: http://www.yamillittle.com/au440/labs/lab2.html