LéPiX - Ceci n'est pas un Photoshop

Fatima Koly (fak2116)
Manager
fak2116@barnard.edu

Gabrielle Taylor (gat2118)
Tester
gat2118@columbia.edu

Jackie Lin (jl4162)
Tester
jl4162@columbia.edu

Akshaan Kakar (ak3808)
Codegen & Language Guru
ak3808@columbia.edu

ThePhD (jm3689)
Codegen & Language Guru
jm3689@columbia.edu

https://github.com/ThePhD/lepix

September 28, 2016

Contents

1 An Overview **2**
- 1.1 Introduction . . . 2
- 1.2 Enter LéPiX . . . 2

2 The LéPiX Language **3**
- 2.1 Language . . . 3
  - 2.1.1 User Defined Types . . . 3
  - 2.1.2 Syntax . . . 4
  - 2.1.3 Built-ins . . . 5
  - 2.1.4 Primitive Types . . . 6

3 Examples **8**

4 Codegen **10**
- 4.1 LLVM IR . . . 10
- 4.2 SPIR-V . . . 10

5 Stretch Goals **12**
- 5.1 User Defined Types . . . 12
- 5.2 Namespacing . . . 12
- 5.3 Named Parameters . . . 12
- 5.4 Lambda Functions . . . 13
- 5.5 Standard Library Implementation . . . 13
- 5.6 Movies: Encoding . . . 13
- 5.7 Windowing: Realtime Visuals . . . 13
- 5.8 Error Noises . . . 13

Chapter 1

An Overview

1.1 Introduction

Heterogeneous computing and graphics processing is an area of intense research. Many existing solutions – such as C++ AMP [5] and OpenCL [3] – leverage the power of an existing language and add preprocessors and software libraries to connect a user to allow code to be run on the GPU. Due to its massively parallelizable nature, code executed on the GPU can be orders of magnitude faster, but comes at the cost of having to master a specific programming library and often learn new framework-specific or platform-specific language subsets in order to compute on the GPU.

1.2 Enter LéPiX

We envision LéPiXto be a graphics processing language based loosely on a subset of the C language. Using an imperative style with strong static typing, we plan to support primitives that enable quick and concise programs for image creation and manipulation. The most novel feature we have planned for the language is the ability to compile to both the CPU as well as the GPU. The reason is to enable high performance and ease the pain most notably found with trying to write applications which leverage the power of the GPU at the cost of a steep learning curve for explicit non-CPU device computation APIs, e.g. OpenGL Compute Shaders, DirectCompute, OpenCL, CUDA, and others. The final goal is to enable the writing of computer vision and computer graphics algorithms in LéPiX with relative ease compared to other languages.

Chapter 2

The LéPiX Language

2.1 Language

The language itself is meant to follow loosely from imperative C, but it subject to change as we refine our desired set of primitives and base operations. What follows is a loose definition of the primitive types, operations we would like to implement to get a baseline for the language, and how would would like to put those together syntactically and grammatically. All of these definitions will be mostly informal.

The goal of LéPiX is to provide a strongly and statically typed language by which to perform image manipulation easily. The hope is that powerful algorithms can be expressed in the language by providing a useful set of basic types, including the concept of a pixel and a natively slice-able matrix type that will serve as the basis for an image.

- Primitive Data Type: the core types defined by the language itself and given a set of supportable operations
- Built-ins: some of the built-in functions to help users
- Function Definitions: how to define a function in the language and use it
- Operators: which operators put into the language and operate on the primitive types

2.1.1 User Defined Types

Support for user-defined types is planned, but will be a stretch goal (5.1). We want to support having user-defined types that can be used everywhere,

and for it to be able to overload the conceivable set of all operations.

2.1.2 Syntax

This is a basic guide to the syntax of the language. We are striving to develop a C-Like imperative language. It will follow many of the C conventions, but with some differences that we think will better fit the domain we are striving to work within. As we do not have a formal grammer just yet, we will present potential programs that we wish to allow to generate appropriate code. You can find these example programs in 3. Below are some quick points about the LéPiX language.

Namespacing As a stretch goal, LéPiX will attempt to support namespacing, to avoid record collision problems as present in OCaml (without the use of modules) and to formalize the good practice of of prepending the short name of the module / library to all functions in C code. Other languages have explicit support. LéPiX will attempt to encourage code sharing and reuse by including the use of namespacing.

Keywords The following words will be reserved for use with the language: namespace, struct, class, typename, typedef, for, while, break, if, elseif, else, void, unit, int, float, uint, pixel, image, vec, vector, mat, matrix. In addition, all identifiers containing __ (two underscores) are reserved for the use of the compiler and the standard library.

The standard library reserves the usage of the namespace lib. The language will place intrinsics and built-ins within the lpx namespace.

Function Definitions Typical function definitions will follow a usual C-style syntax. An example in pseudo-lexer code:

```
function <return type> name ( <parameter list> ) {
    <statement / expression>
    ...
}
```

code/func–def.lex

Ideally, we would like the order of function definition not to matter, so long as it appears *somewhere* in the whole source code listing. Early versions of the LéPiX compiler might require definition before use, for sanity purposes.

As a stretch goal, there are plans for lambda functions (anonymous function values) to be generated by a much more terse syntax.

Control Flow Control flow will follow a C-style syntax as well. An example in pseudo-lexer code:

```
if expr {
    <statement / expression>
    ...
}
elseif expr {
    <statement / expression>
    ...
}
else {
    <statement / expression>
    ...
}
```

code/flow–control.lex

Operations The LéPiX language will support most of the basic mathematical operators. Other operators will be provided VIA functions. These include:

- Mathematical - `plus (+), minus (-), multiply (*), divide (/), modulus (%), power of (**)`
- Logical - `and (&&), or (||), less than (<), greater than (>), equal to (==), not equal to (!=) negate (!)`

Some of these will use both the symbol and the name, such as "not" for negation. Support for bitwise operations, such as left shift and right shift as well as bitwise and / bitwise or, will come as a stretch goal, depending on whether we can handle these basics. Ternary conditionals may also prove useful, but will not be immediately supported.

Comments Single-line comments will begin with //. Multi-line and *nestable* comments will begin with /**/ will be supported as well.

2.1.3 Built-ins

Some useful built-in functions that will be provided with the language:

- Trigonometric functions: lib.sin, lib.cos, lib.tan, lib.asin, lib.acos, lib.atan, lib.atan2

- Power Functions: lib.sqrt, lib.cbrt, lib.pow
- Exponential Functions: lib.expe, lib.exp2, lib.loge, lib.log2, lib.log10

All degree arguments will come in radians. Other functions that operate on built-in types will be provided as library functions, to allow for replacement if necessary.

2.1.4 Primitive Types

Primitives are integral types and multi-dimensional array types. Vectors, Matrices, and Pixels are all subsets of N-dimensional array types. String will be presented as a built-in type, but may be implemented either as a built-in or just an always-included library type. The purpose of string will be specifically to handle reading in and writing out from the file system, as that is the only way to handle such a case. See 2.1 for details.

Table 2.1: Primitive Types in LéPiX

Type	variants	Purpose
void	nothing	built in single-value / empty type
int#	# can be 8/16/32/64 (defaults to 32 without name)	signed integral type
uint#	# can be 8/16/32/64 (defaults to 32 without name)	unsigned integral type
float#	# can be 16/32/64 (defaults to 32 without name)	floating type
<type name>[<optional #>]	array type; can be multidimensional by adding [<optional # >]	basic primitive that will help us represent an image in its decoded form; can be sliced to remove 1 dimension
vec#<type name>	aliases for 1-dimensional fixed-size arrays	base type for pixels
mat#<type name>	aliases to 2-dimensional fixed-size arrays	can be sliced into vectors
string	utf8-encoded string; potentially more later	primarily, to address the file system; not really interested in complex text handling; basically treated as an array of specially-typed uint8
pixel	rgba (red-green-blue-alpha); hsv (hue saturation value)	in the future, this will be something that will need to be customizable to support varying image types of different bit depths

Chapter 3

Examples

The following are some examples of programs that we would like to be written in LéPiX. This does not reflect final syntax and is mostly based on equivalent or near-equivalent C-style code:

```
function image red(int width, int height) {
     // Create an image of a specific width / height
     image ret = image(width, height);
     for(int i = 0; i < width;  i++) {
          for(int j = 0; j < height; j++) {
               // r, g, b creation
               ret[i][j] = pixel(1,0,0);
          }
     }

     return ret;
}

function int main () {
     image img = red(img);
     lib.save(img);
     return 0;
}
```

Listing 3.1: red.lpx

The red example shows up some simple conditionals in a for loop to write all-red to an image. It is not the most exciting code, but it has a lot of moving parts and will help us test several parts of the library, from how to call functions to basic iteration techniques.

```
function void flip(image img) {
     for (int ri = 0; ri < img.height / 2; ri++) {
          // matrix slicing: get back array from index
```

```
4             pixel[] toprow = img[ri];
5             // 0-based indexing
6             pixel[] bottomrow = img[img.height - ri - 1];
7             for (int ci = 0; ci < img.width; ++ci) {
8                  // generic swap call in library
9                  lib.swap(toprow[ci], bottomrow[ci]);
10            }
11       }
12  }
13
14  function int main () {
15       image img = lib.read("meow.png");
16       flip(img);
17       lib.save(img);
18
19       // implicit return 0:
20       // we want this to be able to work with
21       // command-line environments as well
22  }
```

Listing 3.2: flip.lpx

This above code is a bit more complicated. It shows that we can save a slice of a matrix's (image's) row, operate on it, and even call the library function lib.swap on its pixel elements. It also demonstrates a string literal, and passing it to the lib.read function to pull out a regular image from a PNG, and then saving that same image. It also shows off an implicit return 0 (we expect our programs to be run in the context of a shell environment, and to play nice with the existing C tools in that manner).

Chapter 4

Codegen

4.1 LLVM IR

The current goal for generating the code for this language is to use LLVM and serialize to LLVM IR. This will allow our language to work on a multiple of platforms that LLVM supports, provided we can successfully connect our AST / DAG with the our code generator.

4.2 SPIR-V

For the GPU, we still want to compile to LLVM IR. But, with the caveat that we make it work to push out to SPIR-V code using the Khronos LLVM ¡-¿ SPIR-V Bidirectional Translator[1][1]. The good news is that everything from our source code processing steps to our DAG / AST generation can be done in OCaml. However, SPIR-V is new and OCaml bindings for this relatively new project are not something that is quite established: it is conceivable that our code generator will be written in C++ as opposed to any other language, simply because of the library power behind what is already present is written in C and C++, and interfacing that with OCaml might be exceptionally difficult. Granted, we could also write an OCaml Code Generator for SPIR-V from scratch, but this does not seem like a prudent use of our time.

It is also very important to note that the LLVM IR ¡-¿ SPIR-V Translator produces SPIR-V, which by itself cannot be run on anything. We would need a C or C++ compiled Vulkan Driver to take that bit of our program and run it in SPIR-V land. Furthermore, many operations – such as data reading –

[1] Available here: https://github.com/KhronosGroup/SPIRV-LLVM

are not instructions we can exactly slot into SPIR-V code, and would require a bootstrapper of some sort nontheless.

Chapter 5

Stretch Goals

5.1 User Defined Types

User-defined structures are a stretch goal of this project. The core idea is that if we can manage to create pixel, vec#∗, mat#∗, etc. types using the language, we would be able to simply make this kind of functionality available to users. Currently, we plan to hard code these types in at the moment, however.

5.2 Namespacing

Similar to user defined types, namespacing allows an element of organization to be brought to written code. Currently, we are going to hard code built ins to the lib and lpx namespaces.

5.3 Named Parameters

This is an entirely fluff goal to make it easier to call certain functions. The idea is that arguments not yet initialized by the ordered list of arguments to a function call can be specified out-of-order – as long as others inbetween are defaulted – by passing a name=(expression) pairing, separated by commas like regular function arguments.

5.4 Lambda Functions

As mentioned in 2.1.2, we would like to support lambdas as a way of definition functions. Currently, we do not know what the most succinct and terse syntax for our language would be.

5.5 Standard Library Implementation

It would be nice to fill out a standard library implementation, to vet the LéPiX compiler. Candidates would include some basic functions in the lib namespace for manipulation of the image type.

5.6 Movies: Encoding

If we can have built in types for image and the like, then LéPiX could theoretically handle movies by presenting to the user frames of data in sequential order. Doing this is orders of magnitude difficult.

5.7 Windowing: Realtime Visuals

Part of the magic of the graphics card is its ability to perform specific kinds of computation very quickly. It would be very beneficial to have some sort of way to display those visuals without having to serialize them to disk (e.g., a display function or a window of some sort which can be backed by a write-only image).

5.8 Error Noises

The compiler should make a snobby "Ouhh Hooo!" noise in french when the user puts in ill-formed code.

Bibliography

[1] Khronos Group, *Khronos Registry: SPIR-V 1.1*, April 18, 2016. https://www.khronos.org/registry/spir-v/

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Pat Hanrahan, Mike Houston, Kayvon Fatahalian, *Brook for GPUs*, Stanford University, 2016. https://graphics.stanford.edu/projects/brookgpu/arch.html

[3] Khronos Group, *OpenCL 2.2*, April 12, 2016. https://www.khronos.org/opencl/

[4] Chuck Walbourn, Microsoft, *Compute Shaders*, July 14, 2010. https://blogs.msdn.microsoft.com/chuckw/2010/07/14/directcompute/

[5] Dillion Sharlet, Aaron Kunze, Stephen Junkins, Deepti Joshi, *Shevlin Park: Implementing C++ AMP with Clang/LLVM and OpenCL*, November 2012. http://llvm.org/devmtg/2012-11/Sharlet-ShevlinPark.pdf