VLC : Language Reference Manual

Table Of Contents

Introduction

VLC is a Python-like high level language for GPU(Graphical Processing Unit) programming on Nvidia GPUs.

VLC is primarily intended for numerical computation, which can be performed orders of magnitude faster on parallelizable GPU architecture than on traditional x86 architecture. VLC is intended to provide convenient and safe access to the GPU's computational power by abstracting common lower level operations - for example, data transfer between the CPU and the GPU - from the user.

Types and Declarations

The VLC language has two data types: primitives and non-primitives.

Primitives

`<primitive_type> myVar = <value>` declares a primitive type ***primitive_type*** called ***myVar*** with value ***value***

A primitive cannot be ***null***.

	Primitive	Description
	`byte`	An 8-bit signed two's complement integer between -128 and 127
	`int`	32-bit signed two's complement integer between -2147483647 and +2147483647
	`float`	Single precision 32-bit IEEE 754 floating point number with absolute value between 1.4E-45 and 3.4028235E38
	`bool`	A Boolean `true` or `false`
	`char`	16-bit alphanumeric character, valid escape `"\"` character, or punctuation in the ASCII character set

Non-primitives

Declared but uninitialized non-primitives assume a `null` value. Non-primitives cannot be declared `null`, but can only take on the `null` value if they have not been initialized.

Strings

	Non-Primitive	Description
	`string`	A sequence that can be made of characters, valid escape `"\"` characters, or punctuation, immutable.

`string myString = "This is a string"` declares a string with name `myString` and value ***"This is a string"***

Arrays are objects hold a fixed number of primitives or non-primitives. All elements must be values of a single type, unless otherwise specified for special cases.

	Non-Primitive	Description
	`<type> [] myArray`	1-Dimensional array of type ***type***
	`<type> [][] my2DArray`	2-Dimensional array of type ***type***
	`<type> [][][]...[] myArray`	n-Dimensional array of type ***type***

For any array A, A[i][j]...[z] retrieves the element at the ith index of the first dimension, jth index of the second, etc.

	Array Declarations	Description
	`<type>[][] my2DArray = block(myArray,n)`	2-Dimensional array created from `myArray` by blocking every n-elements of `myArray`
	`<type> [n] myArray = {<type1>, <type2>,<type3>...}`	Initializes `myArray` with n user-specified ***type***
	`<primitive_type> [10] myArray = {0}`	Initializes `myArray` with 10 zeros
	`<primitive_type> [10] myArray = {*}`	Initializes `myArray` with 10 random ***primitive_type***

Lexical Conventions

Whitespace

Whitespace refers to the space, horizontal tab, form feed and new line characters. White space is used to separate tokens as well as determine scope. Other than in these uses, it is ignored.

WHITESPACE = `[' ' '\n' '\r' '\t']`

Like Python, whitespace found after a newline in VLC denotes the scope of a statement. A statement located within a scope of another statement should contain at least one recognized delimiting white space character at its start.

delimiting white-space = `[ ' ' , '\t']`

See below for an example.

(Note `i = i + 1` is not aligned with `if` control statement, but begins several white spaces to the right of the line. This defines `i = i + 1` to be in the scope of the `if` block) VLC allows tabs and interprets them as four spaces.

Multi-line statements can be achieved through the use of the line-join character `\`, as shown in the following example:

```
if \
(i=0):
    i = i + 1
```

Comments

VLC comments follow standard comment conventions of C, C++, and Java.

`//` denotes single line comments.

`/*` and `*/` denote start and termination of multi-line comments.

Per C, C++, and Java comment conventions, comments cannot be nested within each other. For example the sequence `/* /* */ */` is not fully recognized as a comment. Only the substring `/* /* */` is recognized as a comment.

COMMENT = `'/' '*'+ [^'*']* '*'+ '/' | '/' '/' [^'\n']*`

Identifiers

An identifier is a case-sensitive sequence of characters consisting of letters, numbers, or underscore, and the first character in an identifier cannot be a number. Identifiers may not take the form of reserved keywords.

ID = `['a'-'z' 'A'-'Z' '_' ] ['a'-'z' 'A'-'Z' '_' '1'-'9']*`

Keywords

`int` `float` `char` `bool` `if` `elif` `else` `for` `while` `continue` `break` `return` `auto` `map` `reduce` `name` `def` `defg` `string` `null` `import` `map` `reduce` `const`

Literals

Integer Literals

An integer constant is an optionally signed sequence of digits. An integer constant can take the form of a `byte` or `int`primitive. A `byte` primitive ranges from -128 to 127 and an `int`

INT = `[‘+’ ‘-’]?[‘0’-’9’]+`

Floating Point Literals

A floating point constant is denoted by an optionally signed integer, a decimal point, a fraction part, an "e" or "E" and an optionally signed exponent. A floating point constant can take the form `float`. A `float` primitive's absolute value ranges from approximately 1.4E−45 to 3.4E38.

Either the fraction part or the integer part must be present, and either the decimal point or the "e" and signed exponent must be present.

FLOAT =

```
  ['+' '-']?['0'-'9']+'.'['0'-'9']*(['e' 'E']['+' '-']?['0'-'9']+)?
| ['+' '-']?['0'-'9']*'.'['0'-'9']+(['e' 'E']['+' '-']?['0'-'9']+)?
| ['+' '-']?['0'-'9']['e' 'E']['+' '-']?['0'-'9']+
```

Boolean Literals

A boolean has two possible values, true or false. These are denoted by the identifiers "true" and "false".

BOOL = `'true'|'false'`

Character Literals

A character literal is denoted by enclosing single quotes ' ', and can be constructed from alphanumeric characters, traditional punctuation characters, and the specified valid escape characters.

	Valid Escape Sequence	Description
	`\'`	Single quote
	`\"`	Double quote
	`\\`	Backslash
	`\n`	New Line
	`\r`	Carriage Return
	`\t`	Horizontal Tab

CHAR = `''' ([' '-'!' '#'-'&' '('-'[' ']'-'~'] | '\\' ['\\' '"' 'n' 'r' 't' ''']) '''`

String Literals

characters.

	Valid Escape Sequence	Description
	`\’`	Single quote
	`\”`	Double quote
	`\\`	Backslash
	`\n`	New Line
	`\r`	Carriage Return
	`\t`	Horizontal Tab

STRING = `'”' ([' '-'!' '#'-'&' '('-'[' ']'-'~'] | '\\' [ '\\' '”' 'n' 'r' 't' ''']) * '”'`

Separators

A separator is a character that separates tokens. White space is also used as a separator, unless it is defining scope.

	Character	Separator
	`'('`	{LPAREN}
	`')'`	{RPAREN}
	`':'`	{COLON}
	`'['`	{LBRACKET}
	`']'`	{RBRACKET}
	`'.'`	{DOT}
	`','`	{COMMA}

Functions

Regular Functions

Functions are declared using the `def` keyword, and must specify their arguments, return type, and a colon `:`. The scope of a function is defined by whitespace - that is, all statements that are part of the function cannot be aligned with the function declaration, but must be "indented", All function arguments that are primitive types are passed by value, meaning all arguments are copied to the function, meaning changes to the argument within the function will not change the argument's value outside of the function.

All function arguments that are non-primitive types are passed by reference, meaning changes to the argument will change the argument's value outside of the function.

Function declaration: `<return type> def <function name>(<type1> arg1, <type2> arg2...):`

GPU Functions

The GPU function `defg` creates a user-defined function that is meant to be run on the GPU kernel. A `defg` function is declared outside of the main function. These functions will be called by the higher-order functions `map` and `reduce` within the `main` function.

There may be only one or two parameters within a `defg` declaration. These restrictions are for `map` and `reduce` respectively. Each parameter is an identifier for a single element in the array(s) that are being handled by `map` and `reduce`.

Constant non-primitives are specified with an input array of constants under the field `const`. These constants should also be specified with the same name in `map` or `reduce`.

GPU function declaration: `<return type> defg <function name> (<type1> arg_1, <type2> arg_2):`

```
<return type> defg <function name> (<type1> arg, const = const[array1, ...]):
```

For convenience, within a `defg` function the index of the element within the index of the input array can be accessed with `ID.x` and `ID.y`. This operation is only available for 1- and 2-D arrays, and the order of a 2-D array will be assumed to be row-major.

Higher Order Functions

VLC contains built-in higher order which take a `defg` as an argument. These built-in higher order functions provide needed abstraction for users who do not wish to be boggled by the specifics of GPU computing but still want to take advantage of parallelism.

The first parameter in a `map` or `reduce` function must be a `defg`. For the remaining parameters, `reduce` takes in only one 1-D array as the second input, but `map` may take a variable number of N-dimensional arrays. All input arrays may not be `NULL`. If the input arrays are multi-dimensional, each dimension must have fixed-length rows. The output of `map` is an N-dimensional array of the same size as the inputs, where `defg` has been applied to the element in the corresponding index as the output. The output of `reduce` is an element of the same type as an element of the input array. The result is obtained by performing pair-wise reduction on adjacent members of the input array. In order to receive correct results, the`defg` function applied to the elements of the input should be commutative.

`map` and `reduce` may capture outside variables through the field `const`. `const` accepts an array of variables to be used in the `defg`. These variables will be copied onto the global memory of the device to be used by the threads executing the `defg` on the elements in the input arrays. `map`

, ,

	Higher Order Function	Description	
	`map(<defg>, <array1>, <array2>...)`	Function that takes as input a function `func` with X open paremeters, and X N-dimensional arrays, performs `func` on the X arrays and returns one resulting array. `map` also accepts 2-Dimensional arrays.	
	`reduce(<func>, <array>)`	Function that takes as input a function `func` with two open paremeters and an array of types `array`, performs pairwise reduction on every pair in `array`, and returns final reduced result. `reduce` also accepts 2-Dimensional arrays.	
	`[map	reduce](, , const = const[array1, array2...]`	The field `const` is optional. The `const` array may contain a variable number of inputs of different types.

Functions `defg` passed to `map` and `reduce`

1) Must have the corresponding number of arguments specified by map (X) and reduce (two)

2) Must have arguments that are the same type as the `array` passed into map and reduce. In the case of map, the order of the argument types to `func` should be match the type of each array

3) Must use the same names in the `const` field.

Map

```
int defg triple(int x):
    return x * 3

main():
    int [4] numbers = {0,1,2,3}
    int [4] triple_numbers = map(triple, numbers)
    print triple_numbers //0 3 6 9
```

Reduce

```
int defg sum(int x, int y):
    return x + y

main():
    print sum // 6
```

Casting

Primitive Types

`byte`,`int`, `float` are primitive types that can be cast to each other. When casting from lower-bit type to a higher-bit type, for example from a `byte` to a `int`, there is no loss of precision. Likewise, casting a higher-bit-type to a lower-bit-type with a value that fits into the lower-bit-type will also generate no loss of precision.

For `int` to `byte` conversions, the latter 8 bits of the `int` are set as the value of the `byte` and the former 24 bits of the `int` are dropped. Performing an unsafe conversion between `int` and `byte` can cause the program to execute falsely.

Casting from a floating-point-type,`float`, to an integer,`byte` or `int`, type drops the fractional part of the floating point type.

Non-primitive Types

VLC is a strongly typed language, and does not allow casting between non-primitive types.

Syntax

Control Flow

If, Elif, and Else Statements

VLC uses standard `if` `else` `elif` control statements. These control statements take a boolean expression as input, and execute branching according to the value of the boolean expression.

An `if` may be followed by optional multiple `elif` statements and an optional`else` statement, and `if` and `elif` statements need not be concluded with an `else`.

Furthermore, every `if`,`else`, and `elif` block defines a new scope. `if`,`elif`, and `else` can also be nested in other `if`,`elif`, and `else` loops.

The below example demonstrates proper use of`if`,`elif`, and `else` loops.

Example:

```
int num = 5
if(num < 5):
    print "Number is less than five!"
elif(num >=5 and num <10):
    print "Number is between five and ten!"
```

Ternary Operator

VLC also provides a shortcut `if else` ternary operator. The below example shows a case setting integer `x` to `<valueA>` if `<condition>` is true, and set to `<valueB>` if `<condition>` is false.

Example:

```
int x = <valueA> if (<condition>=true) else <valueB>
```

While Loops

VLC supports traditional `while` loops, where the substatements within the scope of a while loop are repeated so long as the expression is evaluated to true.

Scope within a `while` loop is defined by prefacing white space characters. See White Space section for further clarification.

Users can break out of a `while` loop using the `break` keyword, or skip to the next iteration of a `while` loop using the `continue` keyword.

A `while` loop in VLC has the following syntax:

Example

```
while ( <expression> ):
    <substatement>
```

For Loops

`for` loops in VLC take as input an `iterator assignment`, a `condition`, and an `iterating statement`.

Scope within a `for` loop is defined by prefacing white space characters. See White Space section for further clarification.

The substatements within the `for` loop repeatedly executes until `condition` is `false`, increasing the iterator defined in the `iterator statement` by the `iterating statement`.

Users can break out of `for` loop iteration using the `break` keyword, or skip to the next iteration of a `for` loop using the `continue` keyword.

In essence, VLC supports traditional `for` loops that follow the below structure.

Example:

```
    for(<statement>,<condition>,<statement>):
        // do something until condition is false
```

Scoping in VLC is static, and follows the conventions of block-level scoping. Variables defined at the top level of a program are available in the global scope of the program.

Expressions

Arithmetic Operators

Traditional Arithmetic Operators

Traditional arithmetic operators can be used between two primitives of type `byte` `int` `long` `float` or `double` . Operators must be used between two elements of the same primitive type.

Traditional Arithmetic Operators	Description
`+`	Addition operator
`-`	Subtraction operator
`/`	Division operator
`*`	Multiplication operator
`%`	Modulo operator
`^`	Exponent/Power operator
`log`	Logarithmic operator
`<<`	Bitshift left
`>>`	Bitshift right

Array Arithmetic Operators

Array arithmetic operators can be used between two arrys consisting of primitive types `byte` `int` `long` `float` or `double`. Operators must be used between two arrays that are of equal length and that contain the same primitive type.

Array Arithmetic Operators	Description

Arithmetic Operators	Description
`arr1+arr2`	Pairwise element addition on two arrays of equal length, returns array of equal length
`arr1-arr2`	Pairwise element subtraction on two arrays of equal length, returns array of equal length
`arr1/arr2`	Pairwise element division on two arrays of equal length, returns array of equal length
`arr1*arr2`	Pairwise element multiplication on two arrays of equal length,returns array of equal length
`arr1.arr2`	Dot product on two arrays of equal length
`arr1**arr2`	Matrix multiplication on two arrays of appropriate dimensions for matrix multiplication, only works for 2-Dimensional arrays

Scalar Array Arithmetic Operators

Scalar array arithmetic operators can be used between an array that contains primitive types of `byte` `int` `long` `float` or `double` and a scalar factor of primitive type `byte` `int` `long` `float` or `double`. The array must contain the same primitive type as the scalar factor.

Scalar Array Arithmetic Operators	Description
`array + n`	Adds scalar factor ***n*** to every element in array, returns array of equal length
`array - n`	Subtracts scalar factor ***n*** from every element in array, returns array of equal length
`array / n`	Divides every element in array by scalar factor ***n***, returns array of equal length
`array * n`	Multiples every element in array by scalar factor ***n***,returns array of equal length
`array ^ n`	Raises every element in array to power of scalar factor ***n***, returns array of equal length

`log(array,n)`
`log(array,n,floor)`
	Takes log scalar factor of every element in array , returns array of equal length

VLC supports the following logic operators, which are most often used in control statements `if` `elif` `else` `while` and `for`.

Logic Operators
`and` `or` `not` `xor` `!=` `==` `>=` `<=` `>` `<`

`and` and `or` logic operators are evaluated using short circuiting principles.

Operator Precedence and Associativity

Operators are listed below from highest to lowest precedence, and operators listed on the same level share the same level of precedenc=

Operator Hierarchy	Operators
1	Logarithmic `log`, Power `^`, Dot Product for Arrays `.`, Matrix Multiplication for 2D Arrays `**`
2	`*` (Multiplication), `/`(Division)
3	Addition `+`, Subtraction `-`
4	Bitshift Operators <<, >>
5	Relational Logic Operators `and` ,`or`, `not` ,`xor`, `==`,`>=`, `<=`, `<`, `>`
6	Assignment `=`

The `=` assignment operator is right associative. All other operators are left-associative.

External Declarations

`main` function and Code Execution

VLC code execution begins at global statements, and then proceeds to execute at a predefined `main` function in the file.

`import` Statements

The `#import` keyword allows VLC to import code from other VLC files. When importing other

For example, if we have file ***a.vlc*** that imports ***b.vlc*** , any `main` function in ***b.vlc*** will be ignored.