Vector Language Manual

Howard Mao (zm2169@columbia.edu), Zachary Newman (zjn2101@columbia.edu), Sidharth Shanker (sps2133@columbia.edu), Jonathan Yu (jy2432@columbia.edu), Harry Lee (hhl2114@columbia.edu)

Introduction

This is the reference manual for Vector, a programming language for the GPU.

Lexical Conventions

Comments

The characters `/*` introduce a comment, and the first `*/` ends the comment. Single-line comments are also supported and denoted with `//` at the beginning of a line.

Identifiers

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore `_` counts as alphabetic. Identifiers are case-sensitive.

Keywords

The following identifiers are reserved for language keywords

`bool char int8 byte uint8 int16 uint16 int int32 uint uint32 int64 uint64 double float float32 double float64 complex complex64 complex128 string if else while for pfor return sync`

Constants

Vector has the following constants:

Integer Constants An integer constant is a sequence of digits.

Character Constants A character constant is a single character enclosed in single quotes `' '`. Single quotes must be preceded by a backslash `\`. The `\` character, along with some non-graphic characters, can be escaped according to the following rules:

- Backspace `\b`
- Newline `\n`
- Carriage Return `\r`
- Tab `\t`
- `\` `\\`

Character constants behave like integers. Characters are stored in two bytes, with the integer code for the character stored in the lower-order byte and 0 in the higher-order byte. For characters of length 2, for example if an escaped character is used, the integer code for the first character is stored in the lower-order byte and the integer code for the second character is stored in the higher-order byte.

String Constants String constants consist of a series of characters delimited by quotation marks `""`.

Floating Constants Floating constants consist of an integer part, a decimal point, a fraction part, an `e` and a signed exponent. If decimal point is not included, then the `e` and signed exponent must be included, otherwise, they are optional.

Syntax Notation

In this manual, a `typewriter` typeface indicates literal words and characters. An *italic* typeface indicates a category with special meaning. Lists are presented either inline or using bullets. If two items are presented on same line of a bulleted list separated by commas, they are equivalent. <epsilon> is used to indicate the empty string. Backus-Naur Form is used to express the grammar of Vector.

Types

type-specifier ::=

primitive-type | *array-type* | *function-type* | `void`

Each identifier is associated with a type that determines how it is interpreted. A void type has no value.

Primitive Types

primitive-type ::= *integer-type* | *floating-point-type* | *complex-type*

Vector supports three categories of primitive types: integers, floating point numbers, and complex numbers.

Integer Types Integer types are given by the following literals:

- `bool`, `char`, `int8`
- `byte`, `uint8`
- `int16`
- `uint16`
- `int`, `int32`
- `uint`, `uint32`
- `int64`
- `uint64`

The types starting with `u` are unsigned types. The number at the end of a type name indicates the size of that type and equivalent types in bits.

Floating Point Types Floating-point types are given by the following literals:

- `float`, `float32`
- `double`, `float64`

These two types correspond to IEEE single-precision and double-precision floating point numbers, respectively, as defined in IEEE 754.

Complex Number Types Complex number types are given by the following literals:

- `complex`, `complex64`
- `complex128`

These two complex types are constructed from two `float32` or two `float64` types, respectively. The real and imaginary parts of the numbers can be accessed or assigned by appending `.re` or `.im` to the identifier.

```
a := #(3.1, 2.1)
b := a.re // b is 3.1
a.im = 1.2 // a is now #(3.1, 1.2)
```

String Types String types are given by the following literal:

- `string`

Array Types

> *array-type* ::= *primitive-type* `[]`

Arrays are composed of multiple instances of primitive types laid out side-by-side in memory.

Arrays are a very important part of the Vector language, as they are the only type that can be modified on both the CPU and GPU. Allocation of arrays on the GPU and transfer of data between CPU and GPU are handled automatically.

Array elements are accessed using square-bracket notation. For instance `a[4]` returns the element at index 4 of array `a` (arrays are zero-indexed). The built-in `len` function returns an `int` representing the length of an array.

Array elements can be assigned to using the same syntax, so `a[4] = 3` will set the value at index 4 of the array to 3.

Arrays can also be multi-dimensional. Indexing into a multi-dimensional array is achieved by separating the dimensional index numbers by commas. So `a[1, 2]` will access row 1, column 2 of the two-dimensional array `a`.

Arrays can be initialized using comma-seperated list delimited by curly braces `{}`.

Function Types

Functions take in zero or more variables of primitive or array types and optionally return a variable of primitive or array type.

Objects and LValues

An object is a named region in memory whose value can be read and modified. An LValue is an expression referring to an object. It has a value, and a corresponding region in memory where the value is stored.

An expression is an RValue if it only has a value, but not a corresponding region in memory, i.e., it cannot have a new value assigned to it. In the following code:

```
a := 4
b := 5
a = b
```

a and b are LValues, because they have a value that can be assigned. 4 and 5 are RValues, because they cannot have new values assigned to them.

LValues are named because they can appear on the left side of an assignment (or also on the right), whereas RValues can appear only on the right side. All expressions in Vector are either RValues or LValues.

Conversions

Scalar types Any scalar type can be converted to another scalar type. For any conversion from a narrower to a wider type, for example from a `int16` to `int32`, or from `float` to `double`, the conversion can be done with no loss of precision. If a `double` is converted to a `float`, then the `double` will be rounded and then truncated to be the length of the `float` type that it is converted to. For conversions between signed types and unsigned types, if the signed type had a negative number, then `UINT_MAX + 1`, where `UINT_MAX` is the maximum unsigned integer of the target type, will be added to it so that the value is a valid unsigned type.

For conversions between any integer and floating-point type, everything up to 16 bits can be cast to `float` with no loss of precision, and everything up to 32 bits can be cast to `double` with no loss of precision. For any floating-point to integer type conversions, the fraction portion of the floating-point type is discarded.

When a `char` object is converted to an int, its sign is propagated through the upper 8 bits of the resulting int object.

explicit-cast ::= *primitive-type-specifier* `(` *identifier* `)`

Expressions

expression ::=

primary-expression

postfix-expression

unary-expression

cast-expression

multiplicative-expression

additive-expression

shift-expression

relational-expression

equality-expression

logical-bitwise-expression

logical-expression

assignment

compound-assignment

function-call

higher-order-function-call

Primary Expressions

Primary expressions consist of constants, identifiers, or expressions in parentheses.

primary-expression ::=

identifier

constant

`(`*expression*`)`

Postfix Expressions

The operators in postfix expressions group left to right.

postfix-expression ::=

expression`++`

expression`--`

Operators

Unary Operators

> *unary-expression* ::=
>
> > *unary-operator expression*

unary operators include `-`, `!`, and `~`. The `-` unary operator returns the negative of its operand. If necessary, the operand is promoted to a wider type. For unsigned quantities, the negative is computed by subtracting the promoted value from the largest value of the promoted type and adding 1. For the case that the value is 0, the value returned is also 0. The `!` operator returns 1 if its operand is 0 and returns 0 otherwise. The `~` operator returns the one's complement of its operand.

Multiplicative Operators Multiplicative operators include `*`, `/`, and `%`, and group to the right. There are two operands and both must have arithmetic types.

> *multiplicative-expression* ::=
>
> > *expression* * *expression*
> >
> > *expression* `/` *expression*
> >
> > *expression* `%` *expression*

The `*` operator denotes multiplication, the `/` operation gives the quotient, and the `%` operator gives the remainder after a division of the two operands.

Additive Operators The additive operators include `+` and `-` group, and they group left-to-right. If the operands have arithmetic types, then the appropriate arithmetic operation is performed.

> *additive-expression* ::=
>
> > *expression* `+` *multiplicative-expression*
> >
> > *expression* `-` *multiplicative-expression*

The `+` operator gives the sum of the two operands, and the `-` operator gives the difference.

Shift Operators The shift operators include $<<$ and $>>$. These operators group left to right, and each operator must be of an integral type.

shift-expression ::=

expression `<<` *expression*

expression `>>` *expression*

The value of shift expression *E1* $<<$ *E2* is interpreted as *E1* left-shifted by *E2* bits, and *E1* $>>$ *E2* is interpreted as *E1* right-shifted *E2* bits.

Relational Operators Relational operators group left-to-right.

relational-expression ::=

expression `>` *expression*

expression `<` *expression*

expression `<=` *expression*

expression `>=` *expression*

The operator `>` denotes the greater-than operation, `<` denotes less-than, `>=` denotes greater-than-or-equal, and `<=` denotes less-then-or-equal.

Each of these operators returns 0 if false and 1 if true, and this result is always of type `int`.

Equality Operators

equality-expression :==

expression `==` *expression*

expression `!=` *expression*

The equality operator `=` denotes equal-to, and the operator `!=` denotes not-equal-to. These both return 1 if true and 0 if false, and this value is of type `int`. These operators have lower precedence than relational operators.

Bitwise Logical Expressions

logical-bitwise-expression ::=

expression `&` *expression*

expression `^` *expression*

expression `|` *expression*

The `&` operator denotes the bitwise-and operation. The result is the bitwise-and function applied to the operands.

The `^` operator denotes the bitwise-exclusive-or operation. The result is the bitwise-exclusive-or operation applied to the two operands.

The `|` operator denotes the bitwise-inclusive-or operation. The result is the bitwise-inclusive-or operation applied to the two operands.

These operations require both operands to have integral types.

Short-Circuit Logical Expressions

logical-expression ::=

expression `&&` *expression*

expression `||` *expression*

The `&&` operator returns 1 if both operands are not equal to 0, and 0 if at least one operand is equal to 0. The `||` operator returns 1 if at least one operand is not equal to 0, and 0 otherwise.

These operators group expressions left-to-right. Operands must be of arithmetic types, and the return value is `int`.

These operators are short circuiting. For `&&`, if the left operand evaluates to 0, the entire expression evaluates to 0 and the right operand is not evaluated. For `||`, if the left operand evaluates to something other than 0, the expression returns 1 and the right operand is not evaluated.

Operator Precedence and Associativity

For binary operators, it is necessary to specify operator precedence and associativity in order to avoid ambiguity. The precedence of operators in vector from highest to lowest precedence is as follows.

- `*`, `/`, `%`
- `+`, `-`
- `<<`, `>>`
- `<`, `<=`, `>`, `>=`
- `==`, `!=`
- `&`
- `^`
- `|`
- `&&`
- `||`
- `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `|=`, `&=`, `^=`

All of these operators are associated left-to-right, with the exception of the last row (the assignment operators) which are associated right-to-left.

Function Calls

function-call ::= *identifier* `(` *argument-list* `)` | *identifier* `()`

argument-list ::= *argument-list* `,` *expression* | *expression*

The type of the identifier must be a function. When a function call is encountered, each of the expressions in its argument list (if it has one) is evaluated (with side effects); the order of evaluation is unspecified. Then, control of execution is given to the function specified by the identifier, with the a copy of the result of each of the expressions available in the scope of the function block as *parameters*.

The types of the each of the expressions in the argument list must match exactly the types of the parameters of the function.

All argument passing is done by-value; that is, a copy of each argument is made before the function has access to it as a parameter. A function may change the value of its parameters without affecting the value of the arguments in the calling function.

However, if an array type is passed to a function, the array is not copied, but merely the reference to the array. Therefore, any modifications the function makes to the array affect the value of the array in the calling context.

A function may call itself.

The result of evaluating the function call is the value returned by the function called.

Higher-Order Functions

higher-order-function-call ::=

`@` *identifier* `(` *identifier* `,` *argument-list* `)`

Vector supports a set of builtin higher order functions. Since these functions require compile-time support, they have a slightly modified syntax to distinguish them from normal functions. Higher-order functions start with an at sign and take a function identifier as the first argument. Currently-supported higher-order functions are `@map` and `@reduce`. Map takes a function of a single argument and an array and returns an array resulting from applying the function to each element of the array. This mapping operation is performed on the GPU. For instance,

```
int evenorodd(int num)
{
    return num % 2;
}

parities := @map(evenorodd, {1, 2, 3, 4, 5});
```

The `parities` variable will get the value `{1, 0, 1, 0, 1}`, where each element is 1 if the corresponding input element was odd of 0 if the input element was even.

The reduce HOF takes a function of two variables and an array and returns the resulting scalar value from applying the function to pairs of the inputs. For instance,

```
int add(int a, int b)
{
    return a + b;
}

sum := @reduce(add, {1, 2, 3, 4, 5});
```

This code takes the sum of the array. There is no guarantee on the order in which items in the array are reduced. Therefore, the reducing function must be associative and commutative or the result with be non-deterministic.

Assignment

assignment ::= *identifier* `=` *expression*

An assignment is simply an identifier and an expression separated by an equals sign. An assignment is itself an expression returning the value that was assigned, so they can be chained. For instance

```
a = b = 3;
```

Assigns the variables a and b to the value 3.

Compound Assignment

compound-assignment ::=

identifier `+=` *expression*

identifier `-=` *expression*

identifier `*=` *expression*

identifier `/=` *expression*

identifier `%=` *expression*

identifier `<<=` *expression*

identifier `>>=` *expression*

identifier `|=` *expression*

identifier `&=` *expression*

identifier `^=` *expression*

Compound assignments perform an operation with the identifier and expression given as operands and then assign the variable specified by the identifier to the result. For example `a += 5` is equivalent to `a = a + 5`.

Declarations

declaration ::=

primitive-declaration

array-declaration

function-declaration

A declaration specifies the type of an identifier; it may or may not allocate memory for the identifier.

Primitive Type Declarations

primitive-declaration ::=

primitive-type-specifier identifier `;`

identifier `:=` *expression* `;`

The first primitive declaration declares a primitive type variable unintialized. In this case, the value of the identifier before first assignment is unspecified.

The second declaration declares a primitive variable with the given identifier with its initial value set to the result of the expression. The type of the identifier will be inferenced from the expression. If you wish to specify the exact type of the identifier, use an explicit cast.

Array Declarations

array-declaration ::= *primitive-type-specifier identifier* `[];`

primitive-type-specifier identifier `[` *index-list* `];`

identifier `:=` *expression* `;`

index-list ::= *index-list* `,` *expression* | *expression*

The first syntax does not initialize the array or allocate any storage for it.

The second syntax declares an array and allocates storage but does not initialize its members. The expression is evaluated (with side effects) and the result is the number of members the array will have (and the size of the array is the size of the primitive type multiplied by the number of members). The type of the expression must be an unsigned integer.

The third syntax inferences the type of the array from the expression and is identical to the initializing declaration for primitives.

Function Declarations

function-declaration ::=

type-specifier identifier (*parameter-list*) *compound-statement*

param-list ::=

non-empty-param-list | <epsilon>

non-empty-param-list ::=

param , *non-empty-param-list* | *param*

A function declaration declares a function that accepts the parameters given by the parameter list and, when called, evaluates the given block (also known as a *function body*). A function may not be modified after declaration.

The parameter list is a series of primitive or array declarations separated by commas. Only the non-initializing primitive declarations and non-sizing array declarations are allowed. The identifiers specified by the parameter list are available in the function body.

Forward Declarations

forward-declaration ::= *type-specifier identifier* (*parameter-list*) ;

A forward declaration is the same as a function declaration except instead of having a compound statement implementing the function, it is terminated by a semicolon. Forward declarations are mainly used for declaring functions that are implemented in C.

Statements

statement ::= *expression-statement*

declaration

compound-statement

selection-statement

iteration-statement

jump-statement

sync-statement

Statements in Vector are executed in sequence except as described as part of compound statements, selection statements, iteration statements, and jump statements.

Expression Statements

expression-statement ::= *expression* `;`

An expression statement is an expression with its value discarded followed by a semicolon. The side effects of the expression still occur.

Declarations

Declarations are also considered statements. The only caveat is that nested function declarations are not allowed.

Compound Statements

compound-statement ::= `{` *statement-list* `}`

statement-list ::= *statement-list* *statement*

<epsilon>

A compound statement is also called a *block*. When a block is executed, each of the statements in its statement list are executed in order. Blocks allow the grouping of multiple statements, especially where only one is expected. In addition, scoping is based on blocks; see the "Scope" section for more details.

Selection Statements

selection-statement ::=

`if (` *expression* `)` *statement* `else` *statement* `if` `(` *expression* `)` *statement*

When a selection statement is executed, first the expression associated with the `if` is evaluated (with all side effects). If the resulting value is nonzero, the first substatement is executed. and control flow resumes after the selection statement. If the resulting value is zero and there is an `else` clause, the substatement of the `else` clause gets executed, and control resumes after the selection statement. If there is no `else` clause, control flow resumes after the selection statement without executing either of the substatements.

If statements can be nested (as in `else if`). If there is ambiguity in which `if` an `else` corresponds to, the ambiguity is always resolved to the closest non-matched `if`.

Iteration Statements

iteration-statement ::= `while` *expression statement*

`for` `(` *iterator-list* `)` *statement*

`pfor` `(` *iterator-list* `)` *statement*

iterator-list ::= *iterator-list* `,` *iterator* | *iterator*

iterator ::= *identifier* `in` (*array-expression* | *range*)

range ::=

expression : *expression* : *expression*

expression : *expression*

: *expression* : *expression*

: *expression*

When a while statement is reached, the expression is evaluated (with all side effects). If its value is nonzero, its block is executed, and after the execution of the block, the while statement is executed again. If the value of the expression is zero, the execution of the while statement is finished.

A for statement allows you to sweep one or more variables across an array or range, evaluating the inner statement with the identifiers assigned to a new set of values each time.

The expression following the `in` in an iterator expression can be an array or a range. If it is an array, the identifier is assigned to the value of each element of the array sequentially. If it is a range, the identifier is assigned to each integer in the range.

A range consists of three integers separated by colons. The integers represent the start of the range, the exclusive end of the range, and the step size of the range. So, for instance, the range `0:5:2` will generate the sequence `0, 2, 4`. The first integer can be ommitted, in which case it will default to zero. The third integer can also be ommitted, in which case it will default to one. If the third integer is ommitted, the second colon is ommitted as well.

The step size can also be negative, in which case the iteration will proceed in reverse order. Therefore, `5:0:-1` will generate the sequence `5, 4, 3, 2, 1`.

If multiple iterators are given in the for statement, the rightmost iterator will go to completion before the iterator to the left is advanced. For instance, the statement

```
for (i in 0:2, j in 0:2) { printf("%d, %d\n", i, j); }
```

Will result in the output

```
0, 0
0, 1
1, 0
1, 1
```

A pfor statement is identical to the for statement, except the iterations of the for statement all happen in parallel on the GPU.

Jump Statements

jump-statement ::= `return` *expression* `;` | `return ;`

A return statement returns control of execution to the caller of the current function. If the statement has an expression, the expression is evaluated (with side effects) and the result is returned to the caller.

Synchronization Statements

sync-statement ::= `sync ;`

The `sync` statement is used inside `pfor` statements for synchronization. When a thread in a `pfor` reaches a `sync`, it will wait at that point until all other threads reach the `sync` statement before continuing.

External Declarations

Scope

Vector uses block-level scoping. A block is another name for a compound statement (see "Compound Statements" section). Most frequently, a block is a section of code contained by a function, conditional, or looping construct. Each nested block creates a new scope, and variables declared in the new scope supersede variables declared in higher scopes.

Include Statements

> *include-statement* ::= `include` *string-literal* `;`

The `include` statement allows a Vector program to include code from other files. It take a file name as a string literal and includes code from that file. The file could either be another Vector program or a C/C++ header file, and the file extension is optional.

Grammar

> *top-level* ::=
>
> > *top-level-statement top-hevel*
> >
> > *top-level-statement*
>
> *top-level-statement* ::=
>
> > *datatype identifier* `(` *param-list* `)` `{` *statment-seq* `}`
> >
> > *datatype identifier* `(` *param-list* `);`
> >
> > *declaration*
> >
> > *include-statement*
>
> *statement-seq* ::=
>
> > *statement statement-seq*

<epsilon>

include-statement ::= `include` *string-literal* `;`

datatype ::=

`int` | `char` | `float` | `bool` | `char` | `int8` | `byte` | `uint8`

`int16` | `uint16` | `int` | `int32` | `uint` | `uint32` | `int64`

`uint64` | `double` | `float` | `float32` | `double` | `float64`

`complex` | `complex64` | `complex128` | `string`

expression ::=

expression `+` *expression*

expression `-` *expression*

expression `*` *expression*

expression `/` *expression*

expression `%` *expression*

expression `<<` *expression*

expression `>>` *expression*

expression `<` *expression*

expression `<=` *expression*

expression `>` *expression*

expression `>=` *expression*

expression `==` *expression*

expression `!=` *expression*

expression `&` *expression*

expression `^` *expression*

expression `|` *expression*

expression `||` *expression*

expression `&&` *expression*

lvalue `+=` *expression*

lvalue `-=` *expression*

lvalue `*=` *expression*

lvalue `/=` *expression*

lvalue `%=` *expression*

lvalue `<<=` *expression*

lvalue `>>=` *expression*

lvalue `|=` *expression*

lvalue `&=` *expression*

lvalue `^=` *expression*

`-`*expression*

`!`*expression*

`~`*expression*

expression`++`

expression`--`

`(`*expression*`)`

lvalue `=` *expression*

lvalue

expression `[`*expression-list*`]`

expression `[`*expression-list*`]` `=` *expression*

constant

datatype `(`*expression*`)`

`{`*expression-list*`}`

identifier `()`

identifier `(`*expression-list*`)`

`@`*identifier* `(`*identifier*`,`*expression-list*`)`

lvalue ::=

identifier

expression `[`*expression-list*`]`

expression-list ::=

expression `,` *expression-list*

expression

declaration ::=

identifier `:=` *expression*`;`

datatype *identifier* `;`

datatype *identifier* `[]`;

datatype *identifier* `[`*expression-list*`]``;`

statement ::=

`if (`*expression*`)` *statement* `else` *statement*

`if (`*expression*`)`*statement*

`while (` *expression* `)` *statement*

`for (` *iterator-list* `)` *statement*

`pfor (` *iterator-list* `)` *statement*

`{` *statement-seq* `}`

expression `;`

`;`

declaration

`return` *expression* `;`

`return;`

iterator-list ::=

iterator `,` *iterator-list*

iterator

iterator ::=

identifier `in` *range*

identifier `in` *expression*

range ::=

expression `:` *expression* `:` *expression*

expression `:` *expression*

: *expression* : *expression*

: *expression*

param ::=

datatype identifier

datatype identifier `[]`

non-empty-param-list ::=

param , *non-empty-param-list*

param

param-list ::=

non-empty-param-list

<epsilon>

constant ::=

int

int64

float

complex

string

char