MatPix Language Reference Manual

David Burkat

Oliver Cossairt

Robert Hawkins

Ben London

1. Introduction

Modern GPUs are designed to perform extremely fast, highly parallelized matrix processing; but their power is often under-utilized by most common computing tasks. By offloading certain specialized processes to the GPU, one can expect massive performance gains. MatPix is a programming language for matrix arithmetic -- similar to the popular software package, Matlab, but with the performance gains that are realizable by utilizing the GPU. Essentially, the MatPix compiler will translate MatPix source code into C code with embedded OpenGL calls for optimized matrix operations.

1.1 How to read this document

All character values are contained within double quotes, i.e. "A", "/*", etc. It is implied that, when using the language, the double-quotes should be omitted. Code examples will be indented and in `Courier New` font. Grammatical definitions follow ANTLR syntax, and will be indented and italicized.

2 Lexical conventions

A token can be a non-empty string of non-whitespace characters. There are five categories of tokens: identifiers, keywords, constants, operators and separators. During parsing, whitespace (space, tab, newlines) and comments are ignored, except as they serve to separate tokens; at least one of these non-tokens is required to separate certain tokens.

2.2 Comments

The language supports both single line and multi-line (i.e. traditional "C-style") comments. Single line comments begin with two forward-slash characters "//" and terminate at the end of the line in which they appear. Multi-line comments begin with the characters "/*" and terminate after the first instance of the characters "*/" is encountered.

2.3 Identifiers

An identifier is a non-empty string over upper and lower case letters, digits, and the underscore "_" character. The first character must be either a letter. Identifiers are case-sensitive.

The following are examples of *valid* identifiers:

```
myid   my_id   myId123   a
```

The following are examples of *invalid* identifiers:

```
123myId   $myId   my#Id   _123MyId
```

The grammar for identifiers is as follows:

```
identifier
   : ( 'a'..'z' | 'A'..'Z' ) ( ( 'a'..'z' | 'A'..'Z' ) | ("0".."9") | "_" )* ;
```

2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
print
if
else
for
while
break
continue
return
function
```

3 Expressions

3.1 Constants

3.1.1 Scalar constants

All scalar constants are single precision floating point values, following closely the the well-known C syntax. In its full form, a scalar constant consists of an integer part, followed by a decimal point, followed by a fractional part, followed by an optionally signed integer exponent. Only the fractional part may be omitted. If the fractional part is omitted, the period must also be omitted. The exponent is in all cases optional.

Both the integer and fractional parts are non-empty strings over digits '0' ... '9'. The exponent must be preceded by an 'e' (case-insensitive), optionally followed by a sign, followed by a non-empty string over digits '0' ... '9'.

The following are examples of *valid* scalar constants:

```
123   123.45   0.45   123.45e-6   123E4
```

The grammar for scalar constants is as follows:

```
DIGIT
   : ("0".."9");
DECIMAL
   : DIGIT+ ( "." DIGIT+ )? ;
SCALAR
   : DECIMAL ( 'e' (MINUS)? DIGIT+ )? ;
```

3.1.2 Matrix constants

Matrix constants begin with a left bracket '[', followed by a series of rows, separated by the row separator '|' and optional whitespace, and closed with a right bracket ']'. Each row consists of a series of scalar constants, separated by commas and optional whitespace. All rows must contain the same number of scalars, except for the last row, which may optionally be empty.

The following are examples of *valid* matrix constants:

```
[ 1, 2, 3 ]
[ 1, 2 | 3, 4 ]
[ 1, 2 | ]
```

The following are examples of *invalid* matrix constants:

```
[ 1 2 3 ]
[ 1, 2, 3 | 4, 5 ]
```

The grammar for matrix constants is as follows:

```
cols
   : SCALAR ("," SCALAR)* ;
row
   : cols ("|")? ;
matrix
   : "[" (row)+ "]" ;
```

3.1.3 String constants

String constants consist of a sequence of characters enclosed in double quotes. A double-quote character may be included in the string by writing two double quote characters side by side. An example of a string containing two double-quote characters is:

```
"A bee flew ""over"" the house"
```

Strings are only to be used inside `print` statements. Everything within the quotes will be printed to the output stream. This is literal transcription, with the only exception being the escape double quotes.

3.2 Expressions

Expressions in MatPix are combinations of unary and binary operations, matrix slicing, function calls, variables (including slices of matrix variables), and constants. Binary operations are either mathematical or logical operations. Assignment is handled as a binary operator. Function calls and variables can either be used as expressions on their own or in nested expressions. All operators except assignment can accept expressions as their argument(s). The arguments of binary operators are required to have the same dimension and size. MatPix grammar does not enforce this rule, so an error is reported during the semantic analysis stage. All operators are performed element-wise on their arguments. Thus an increment performed on a matrix variable increments each of its elements. Similarly a multiply performed with two matrix variables as input returns a matrix variable where each element is the product of the two elements of the same index from the input variables.

3.2.1 Unary Operations

Unary operations can be performed on any expression. The Unary operators are "−", "!", "++", "−−". Respectively, these operators perform sign negation, logical negation, increment by one and decrement by one. Logical negation simply sets the variable value(s) to one if it is zero, and sets it to zero otherwise. Examples of expressions including unary operators are:

```
c = b++;
a[1:10]++;
x = -z;
b = !a[1:10];
```

The grammar for unary operations is as follows:

```
signedExpression
   : "-" incrementExpression;
incrementExpr
```

: variable ("++" | "--")?;

3.2.2 Mathematical Binary Operations

Binary math operations take two arguments that can be any expression. The operators are "%", "^", "*", "/", "+", and "-". Respectively, these operators perform modulo, exponentiation, multiplication, addition, and subtraction. Examples are:

```
x = a % b;
3 * 4 + 5 * 6;
a = 3 * (4 - 5) / 6;
z = 2 + 3^4;
```

The grammar for mathematical binary operations is as follows:

addExpression
: multiplyExpression (("+" |"-") addExpression)?;
multiplyExpression
: signedExpression (("" | "/" | "%") multiplyExpression)? ;*

3.2.3 Logical Operations

Logical operations take two arguments that can be any expression and return floating point values of zero or one. The supported logical operators are "<", ">", "<=", ">=", "==", "!=", "&&", "||". Like the other binary operators, the logical operators perform element-wise comparisons. The "<", ">", "<=", ">=", "==", "!=" respectively perform greater than, less than, greater than or equal, less than or equal, equal, and not equal comparisons to their arguments. The "&&" and "||" operators first convert the left and right input arguments to boolean representation by setting them to zero if they have a value of one, and one otherwise. After converting to boolean, the logical operators then perform the familiar operations on the input values (logical "and" and logical "or"). Examples of logical operations are:

```
a > b;
x = a < b;
x = a == b;
x = a != b;

if (a > b)
{
  a = b;
}
```

The grammar for logical binary operations is as follows:

logicalExpression
: predExpr (("&&"|"||") logicalExpression)?;
predicateExpression
: (("!")? addExpression) (("<"|">"|"<="|">=") predicateExpression)?;

3.2.4 Function Calls

Function calls accept arguments that can be any expression and return values that can be used as temporary variables in nested expressions. MatPix grammar allows a function to either explicitly return a value or not. If a value is not returned explicitly, semantic analysis detects this and implicitly returns a scalar with value zero. Thus, it is valid in MatPix to make an assignment to a function that does not declare a return value (unless the assignee has a different size, which is not allowed for any binary operation). The grammar allows an arbitrary amount of function parameters. The size of the variable returned by a function call will depend on the function definition. If no return statement is issued in the function definition, then the function call will return zero.

Examples of function call usage are:

```
A[:] = foo(a);
foo(a, b);
a = foo(a, b);
```

The grammar for function calls is as follows:

functionCall
: identifier argumentList;
argumentList
: "(" (expression ("," expression))? ")"* ;

3.2.5 Variables

Variables in MatPix can either be temporary (as in a constant or function call return value) or peristent (as in an identifier which refers to a variable in stored in memory, or the result of a matrix index which refers to a region of memory). Only persistent variables can be assigned to, but all variables can be used in nested expressions. Since there is no typing in MatPix, all variables are implemented as matrices. Single value floating point variables are implemented as 1x1 matrices. For convenience, a syntax that is familiar for a single value variable is provided. The syntax is enforced by automatically converting scalar values to 1x1 matrix representation, so that assignment is valid. Thus, the syntax allows a 1x1 matrix to be assigned to a scalar without indexing. The following examples demonstrate the use of variables.

```
a;
a = 1.2;
a = [1.2];
a[1:2] = foo();
z = [1,2,3];
a = (foo() + b) * c[1:2] + [1, 2];
```

The grammar for variables is as follows:

variable
: (identifier | *matrixIndex* | *constant* | *functionCall)*;

3.2.6 Matrix Indexing

MatPix allows flexible matrix indexing using customizable ranges, similar to Matlab. Matrix indexing allows us to slice arrays using integer increments.

A range is a scalar followed by an optional semicolon and scalar followed by another optional semicolon and scalar. All scalars that are used in a range expression are converted to integer representation during semantic analysis by flooring. Thus, non-integer ranges are allowed, but automatically converted to integer representation. Semantic analysis ensures that the calculated indices are within the range of valid indices of the matrix being sliced.

The grammar for matrix indexing is as follows:

matrixIndex
: *identifier arraySlice*;
arraySlice:
"[" *rangelist* "]";
rangeList:
range ("," *range*)?;
range:
(*SCALAR* (":" *SCALAR* (":" *SCALAR*)?)?) | ":";

Examples of matrix indexing are:

```
A[1:5] = [1, 2, 3, 4, 5];
A[1:2:10] = [1, 2, 3, 4, 5];
A[1:2, 1:2] = [1, 2| 3, 4];
A[1:2:4, 1:2:4] = [1, 2| 3, 4];
A[1:2:4, :] = [1, 2];
B[:, :] = [1, 2| 3, 4];
```

Note: A variable without a subscript is shorthand for a slice consisting of every element in the matrix. Thus for a 4x4 matrix C:

```
C;
```

and

```
C[:, :]
```

both return a 4x4 matrix of values. Thus the following examples are also valid syntax (assuming the matrices on both sides of the assignment have the same size and dimension)

```
C = [1,2 | 3,4];
C[1:2, 1:2] = D;
A = C;
```

3.2.7 Assignment

Assignment is the only binary operator does not accept all expressions as arguments. The left side of an assignment is restricted to a persistent variable (which includes a slice of a matrix), while the right side of the matrix is allowed to be any expression. The grammar does not enforce this rule and allows for any expression to be used as either argument. Semantic analysis enforces this rule by reporting an error when a constant, function call, or any nested expression is detected on the left side of an assignment. Examples of assignments are :

```
a[:,:] = [ 1, 2 | 3, 4  ];
a = foo(x,y);
a[1:2, 1:2 ] = b[1:3, 2:5 ];
foo = 3 + 4;
a = b[2:4, 5:7 ];
```

The grammar for assignment is as follows:

expression
: assignment ;
assignment
: logicalExpression ("=" logicalExpression)? ;

3.3 Operator Precedence

The primary-expression operator:
()
has highest priority and groups left-to-right. Binary and unary operators all group left-to-right, and have priority decreasing as indicated:
++ −−
- (unary)
^
* / %
+ − (binary)
! < > <= >=
== !=
&& ||
Assignment operators all have the same (lowest) priority, and all group right-to-left.

4 Statements

A statement is an expression, function definition, "return", "break", or "continue" followed by a semicolon, a control statement, or just a semicolon.

The grammar for statements is as follows:

statement
: *expression ";"*
| *functionDefinition ";"*
| *controlStatement ";"*
| "return" *expr ";"*
| "break" *";"*
| "continue" *";"*
| *";"* ;

4.1 Functions Definitions

A function definition consists of the string literal "function" followed by an identifier and an argument list. The function body consists of a left curly bracket followed by zero or more statements followed by a right curly bracket.

The grammar for function definitions is as follows:

functionDefinition
: "function" *identifier definitionArgumentList* "{" (*statement*)* "}" ;

definitionArgumentList:
: "(" *identifier* ("," *identifier)?* ")" ;

4.2 Control Flow Statements

A Control Flow Statement consists of either an if statement, a while statement, or a for statement.

An if statement consists of the word "if" followed by a logical expression enclosed in parenthesis, followed by one or more statements enclosed in curly brackets, followed by an optional else clause. The else clause consists of the word "else" followed by one or more statements enclosed in curly brackets.

A while statement consists of the word "while" followed by a logical expression enclosed in parenthesis, followed by one or more statements enclosed in curly brackets.

A for statement consists of the word "for" followed by an identifier, followed by an equals sign, followed by a range, followed by one or more statements enclosed in curly brackets.

The grammar for control flow statements is as follows:

controlflow
: *ifStatement* | *whileStatement* | *forStatment;*
ifStatement
: "if" "(" *logicalExpression* ")" "{" (*statement*)+ *"*}" (else "{" (*statement*)+ "}")?;
whileStatement
: "while" "(" *logicalExpression* ")" "{" (*statement*)+ *"*}";
forStatement
: "for" *identifier* "=" *range* "{" (*statement*)+ "}";

5 Grammar Summary

```
// scalars and identifiers
DIGIT
    : ("0".."9");
DECIMAL
    : DIGIT+ ( "." DIGIT+ )? ;
SCALAR
    : DECIMAL ( 'e' (MINUS)? DIGIT+ )? ;

identifier
    : ( 'a'..'z' | 'A'..'Z' ) (LETTER | DIG | "_")* ;

// matrix constants
cols
    : SCALAR ("," SCALAR)* ;
row
    : cols ("|")? ;
matrix
    : "[" (row)+ "]" ;

// expressions
expression
    : assignment ;
assignment
    : logicalExpression ("=" logicalExpression)? ;
logicalExpression
    : predExpr (("&&"|"||") logicalExpression)?;
predicateExpression
    : (("!")? addExpression) (("<"|">"|"<="|">=") predicateExpression)?;
addExpression
    : multiplyExpression ( ("+" |"-") addExpression )?;
multiplyExpression
    : signedExpression ( ("*" | "/" | "%") multiplyExpression )? ;
signedExpression
    : "-" incrementExpression;
incrementExpr
    : variable ("++" | "--")?;
variable
    : (identifier | matrixIndex | constant | functionCall );

//function calls
functionCall
    : identifier argumentList;
argumentList
    : "(" (expression ("," expression)* )? ")" ;

// matrix indexing

matrixIndex

    : identifier arraySlice;

arraySlice:

    "[" rangelist "]";

rangeList:
```

```
    range ("," range )?;

range:

    (SCALAR (":" SCALAR (":" SCALAR)?)?) | ":";


        // statements
        statement
            : expression ";"
            | functionDefinition ";"
            | controlStatement ";"
            | "return" expr ";"
            | "break"  ";"
            | "continue"  ";"
            | ";" ;

        // function definitions
        functionDefinition
            : "function" identifier definitionArgumentList "{" (statement)* "}" ;

definitionArgumentList:
    : "(" identifier ("," identifier)? ")" ;
// control flow
controlflow
    : ifStatement | whileStatement | forStatment;

ifStatement
    : "if" "(" logicalExpression ")" "{" (statement)+ "}" ( else "{" (statement)+ "}" )?;

whileStatement
    : "while" "(" logicalExpression ")" "{" (statement)+ "}";

forStatement
    : "for" identifier "=" range "{" (statement)+ "}";
```