Loom Report (Draft)

Johnathan Jenkins

July 4, 2011

Abstract

Loom is a domain-specific language for general-purpose graphics processing units ('gpgpus') designed to support millions of parallel execution threads. It is designed to interface with code written in a 'host' language (typically C or C++) running on the cpu. Loom is sufficiently low-level to implement efficient parallel algorithms, but includes facilities such as *parallel map*, *parallel reduce* and *parallel scan* to abstract common patterns.

Contents

Chapter 1

Introduction

1.1 Background

Loom is a language for programming massively parallel co-processors used on many modern computers. The specific devices supported are nvidia gpgpus using the cuda architecture.

Cuda devices support tens of thousands, or even millions of simultaneous execution threads, with hundreds of threads running in parallel at any time. The threads are organized into a hierarchy: groups of 32 threads make up *warps* that execute in series on single core, and do not have to be synchronized; groups of up to 512 threads make up *blocks*, which can use high-speed shared memory for inter-thread communication and user-managed caching; the blocks collectively form a *grid*, and all threads in a single grid execute the same *kernel* code. Although the kernel is the same for each thread in a grid, individual threads need not follow the same execution path.

In addition to shared memory, devices have cached *constant memory*, registers, and *global memory*. Transfer of data from the host computer's main memory to the device global memory space occurs in code on the host side (that is, not within kernels running on the gpgpu device). This simplifies loom, which is solely for compiling gpgpu kernels. In addition to memory transfer to and from the host computer, all input and output occurs on the host side.

Most cuda device programming is done in a C-like language supplied by nvidia. The target language for loom, however, is a lower-level language called ptx. Ptx looks very much like a traditional assembly language, although it runs on a device-independent virtual machine and is jit-compiled to a *cubin* binary for execution.

Extensive documentation on cuda is available from *http://developer.nvidia.com*.

1.2 Overview of Loom

1.2.1 Design Goals and Language Features

Loom attempts to abstract away many of the repetitive and error-prone details involved in writing gpgpu kernels, such as explicit array index calculations and thread barrier synchronization, while remaining at a sufficiently low level to allow interesting parallel algorithms to be implemented (rather than merely used in a black-box library, such as the CUDA linear algebra libraries supplied by NVIDIA).

The language is statically typed. Types are indicated following a colon after a variable name: `x: Int32`. In addition to *basic types* such as Int32 and Float32, which correspond directly to PTX types, there are *records* such as `pair: {first:Int32, second:Int32}` (a pair of integers), *vectors* such as `v: Int32[10]`, and two dimensional arrays such as `a: Int32[5,5]`. Several special constants make it easy to work on large vectors and arrays in parallel.

LOOM has several standard control constructs for conditionals and looping, which are demonstrated in the sample programs below. The language also has operators designed to simplify parallel programming. There is a *parallel map* operator which applies a function of one variable to each element in a vector or array: `f // v`. The *parallel reduce* operator, `/.`, applies a function of two variables (which should be associative in those variables) repeatedly to reduce a vector or array to a single value per CUDA block (subsequent kernel calls, or code running on the host CPU, can then be used to complete the reduction). LOOM also defines left and right *parallel scan* operators, `/:` and `:/`. As with the reduction operator, scans work across blocks rather than across entire vectors.

Shared memory, which functions both as a user-managed cache and a mechanism for sharing data between threads in a single block, is allocated by declaring a variable with the `shared` keyword. CUDA programs typically use explicit barrier synchronization instructions to synchronize threads within a block; in LOOM, statements that move data between memory state spaces and alter state are synchronized by default.

Due to limitations of at least certain CUDA architectures, functions cannot be recursive. A newer architecture used on high-end graphics cards, called 'Fermi', permits recursive functions as well as a number of other powerful features. It would be an interesting exercise to extend LOOM to take advantage of some of those features.

1.2.2 Representative Programs

Finally, we show listings of a few short programs illustrating some of LOOM's features.

▷ *Find the maximum values in a two-dimensional array (by CUDA block):*

```
kernel maximum(in: Int32[X_THREADS, Y_THREADS],
    out: Int32[X_BLOCKS, Y_BLOCKS])
out <- max /. in
```

▷ *Shift the values in a vector to the left by exactly one block:*

```
kernel shiftLeft(in: Float32[THREADS], out: Float32[THREADS])
    current: Range <- block(B)    -- 'B' is the current block index
    previous: Range <- block(B-1)    -- Range type {Int32, Int32}
    out[[current]] <- in[[previous]]
```

Note that arrays indexed with double brackets are bounds-checked and padded to zero outside the defined range. Single-brackets perform unsafe array indexing. This following is a more explicit, but equivalent, implementation:

```
kernel shiftLeft1(in: Float32[THREADS], out: Float32[THREADS])
    if B > 0 then
        for i: Int32 <- B*BLOCKSIZE .. (B+1)*BLOCKSIZE
```

```
            out[i] <- in[i-BLOCKSIZE]
    else
        for i: Int32 <- B*BLOCKSIZE .. (B+1)*BLOCKSIZE
            out[i] <- 0
```

The `..` symbol in the `for` statement shows that `i` takes on successive values from `B*BLOCKSIZE` (inclusive) to `(B+1)*BLOCKSIZE` (exclusive). We could have expressed the same range of values by writing `B*BLOCKSIZE ... (B+1)*BLOCKSIZE-1`, where the `...` symbol shows that `i` goes up to `(B+1)*BLOCKSIZE-1` (inclusive).

▷ *Compute the sum of squares (by block):*

```
func sum(x: Float32, y: Float32): Float32
    return x + y

func sqr(x: Float32): Float32
    return x*x

kernel sumOfSquares(in: Int32[THREADS], out: Int32[BLOCKS])
    local current: Range <- block(B)
    shared t: Float32[] <- in[current]  -- size is BLOCKSIZE
    out[B] <- sum /. (sqr // t)
```

Chapter 2

Tutorial

Chapter 3

Loom Reference Manual

3.1 Introduction

This reference manual gives a brief description of the LOOM language, following the model of Appendix A to Kernighan and Ritchie, *The C Programming Language* (2nd ed.). In some cases, sections headings have been taken directly from K&R.

Because LOOM is designed to closely match the target CUDA architecture, in a number of cases LOOM language features should be understood with reference to the relevant NVIDIA documentation. For example, internal floating-point formats, limitations on the number of threads in a warp, and similar information can be found in the CUDA API Reference Manual Version 4.0, the CUDA C Programming Guide Version 4.0, and the CUDA PTX: Parallel Thread Execution ISA Version 1.4, all of which are available at *http://developer.nvidia.com/nvidia-gpu-computing-documentation*.

text separated by newline characters. A program consists of exactly one `kernel` definition, along with `func` definitions for supporting device functions, and type declarations.

Source files containing libraries of `func`s and type declarations can be included with an `import` directive:

```
import <path>
```

`<path>` is a relative or absolute path to the additional source file to be imported. `import` is not a true statement in the language; it simply causes additional files to be spliced in, like C's `#include`.

LOOM programs are typically written using line breaks and indentation to indicate block structure, rather than with explicit curly braces and semicolons as for C-syntax languages. To facilitate parsing, however, the first stage in compiling a LOOM program is running the input source code through a preprocessor that inserts braces and semicolons to mark blocks and statements.

The preprocessor goes through the input line by line, keeping a stack of indentation amounts, as well as the indentation of the preceding line. For simplicity, indentation is indicated solely by spaces at the beginning of the line – behavior is undefined if there are tab characters in the input. The preprocessor applies the following rules for each line:

1. If the line contains a keyword that introduce a block (`if-then`, `else`, `for`, `while`, `func`, and `kernel`), insert an open brace at the end of the line.

2. If the line does not contain a block-introducing keyword, and is indented further than the preceding line, push the current indentation onto the stack and add a semicolon at the end of the line.

3. If the line does not contain a block-introducing keyword, and is indented the same as the preceding line, add a semicolon at the end.

4. If the line is indented less than the preceding line, pop the stack, and check that the indentation is the same as the new top of the stack. If it is the same, insert a closing brace and continue with steps 1–3 as appropriate. Otherwise, indicate an indentation error.

▷ *Example: convert the following code to use explicit braces and semicolons:*

```
func foo(a: Int32): Int32
    local sum: Int32 <- 0
    for i: Int32 <- 1 .. 5
        iSqr: Int32 <- i*i
        sum += i2
    return sum   -- sum = 1 + 2*2 + 3*3 + 4*4 = 30
```

The preprocessor converts this into

```
func foo(a: Int32): Int32 {
    local sum: Int32 <- 0;
    for i: Int32 <- 1 .. 5 {
        iSqr: Int32 <- i*i;
        sum += i2;
    }
    return sum;
}
```

3.2 Lexical Elements

Tokens may be comments, keywords, operators, identifiers, constants, or thread parameters. There are also a few delimiter tokens that do not fall into these categories, such as parentheses (which are used to indicate grouping within expressions and to set off function arguments) and the two- and three-dot range symbols used within `for` statements.

After the preprocessing stage, whitespace is generally ignored except where necessary to separate adjacent tokens that would otherwise be lexically ambiguous.

3.2.1 Comments

Comments are indicated by two adjacent hyphen characters (‘`--`’). They may begin anywhere in a line, and continue until the next newline character, which marks the end of the line. There is no special syntax for multiline comments. Comments are treated as whitespace.

3.2.2 Reserved Words

The following keywords, typenames, and predefined value symbols are reserved:

```
break       global      Bool        B
const       kernel      Byte        BLOCKS
continue    local       Float32     BLOCKSIZE
else        return      Float64     T
if          shared      Int32       THREADS
import      then        Int64
for         type        Uint32      FALSE
func        while       Uint64      TRUE
```

In addition, to facilitate working with two-dimensional arrays, 2-D variations of the block and thread symbols in the right-hand column are also used (X_B, Y_B, X_BLOCKS and Y_BLOCKS, etc.).

3.2.3 Operators and Delimiters

Loom recognizes the following operators and delimiters, ranked in order of precedence from highest (top row) to lowest (bottom row):

```
{   }   ;
(   )   [   ]   [[  ]]
:
!
*   /   %   <<  >>  &
+   -   |   ^
=   !=  <   <=  >   >=
&&
||
//  /.  /:  :/  ..  ...
<-
```

Individual operators and delimiters are described in the following table:

`{ } ;`	statement blocks, statement termination
`[ ]`	unsafe array indexing
`[[ ]]`	safe array indexing with zero-padding
`:`	type tag
`!`	boolean negation
`* / %`	multiply, divide, modulo
`<< >> &`	shift-left, shift-right, bitwise and
`+ -`	add, subtract / unary negate
`|^`	bitwise or, xor / unary one's complement
`= != < <= > >=`	relational operators
`&& ||`	logical and, or
`// /. /: :/`	parallel map, reduce, scanl and scanr
`.. ...`	range delimiter in `for` statements
`<-`	assignment

Most binary operators group left-to-right, although assignments and expressions with unary operators group right-to-left.

3.2.4 Identifiers

User-defined identifiers may refer to functions, variables, types, arrays, or records. Identifiers must be accepted by the following regular expression:

```
identifier:: [a-zA-Z][a-zA-Z0-9_']*
```

3.2.5 Constants

LOOM supports literal constant expressions for booleans and 64-bit integers and floating-point numbers.

Floating-point literals are written with an optional decimal point, and an optional signed exponent following `[eE]`.

Constants may be followed by a type tag to force their type to be different from the default (e.g., `123: Float32` will be read in as a `Float32`).

Constants may be combined into *constant expressions* using the evaluation rules for expressions set forth below.

3.2.6 Boolean Constants

TRUE and FALSE are predefined symbolic constants of type `Bool`.

3.2.7 Thread Parameters

A number of symbolic parameters are defined at the time the kernel is invoked by host code running on the CPU. These parameters allow the GPGPU device thread to know about its execution context.

`B`	Index of the current thread block
`BLOCKS`	Number of thread blocks in the grid
`BLOCKSIZE`	Number of threads per block
`T`	Index of the current thread
`THREADS`	Number of threads in the grid

The parameters in the table correspond to grids laid out in one dimension (i.e., laid out to easily accommodate one-dimensional data structures for inputs and outputs to and from the GPGPU). LOOM also supports two-dimensional layouts, in which case analogues to the parameters above are defined at the time of kernel invocation, prefaced by `X_` and `Y_`.

3.3 Data Types

Each variable in LOOM has an associated identifier (its name) and a data type. The basic LOOM data types generally correspond their CUDA/PTX equivalents, except for `Bool`s, which are primarily used to control program execution. In addition to the basic types, LOOM supports (one- and two-dimensional) array and record container types, as well as new types defined in a `type` statement.

3.3.1 Type Declarations

Variables must be declared before they are used (although in the special case of `for` statements, the declaration can occur within the statement itself). Declaration statements take an optional initialization assignment. If no explicit initialization is present, the variable is set to zero (in the case of numerical types) or `FALSE` (in the case of `Bool`s).

```
local x: Int32  -- Initialized to 0
local e: Float32 <- 2.718281828
for i: Int32 <- 0 .. 10
    x <- x + i
```

All variable objects exist in one of three memory spaces: global memory (the main memory on the GPGPU device), shared memory (a much smaller amount of high-speed memory simultaneously accessible to the threads in a single CUDA block) or local memory (memory local to a single thread, often mapped to registers).

```
variable-decl:  memory-space identifier ':' type initializer
memory-space:   'local' | 'shared' | 'global'
type:           identifier | basic-type | array-type | record-type
basic-type:     'Bool' | 'Byte' | 'Float32' | ...
array-type:     type '[' constant-expression ']'
record-type:    '{' identifier ':' type maybe-more '}'
maybe-more:     '' | ',' identifier ':' type maybe-more
initializer:    '' | '<-' constant-expression
```

3.3.2 Defining New Types

Users may define their own types to supplement the built-ins.

```
type-definition:  'type' identifier type
```

3.4 Expressions

Expressions are combinations of variables, constants, function calls and operators that have a value, and therefore a type. Instead of specifying an unambiguous grammar for expressions (as in K&R Appendix A), ambiguities in the following grammar are resolved with reference to the operator precedence and grouping rules provided above.

```
expression:       identifier
                |  '(' expression ')'
                | binop-expr | unary-expr
                | reference
                | assignment-expr
                | parallel-expr
                | function-call
binop-expr:       expression binop expression
binop:            + | - | * | / | ...
unary-expr:       unary-op expression
unary-op:         ! | - | ^
reference:        array-unsafe-ref | array-safe-ref | record-ref
array-unsafe-ref: identifier '[' expression second-index ']'
array-safe-ref:   identifier '[[' expression second-index ']]'
```

```
second-index:     '' | ',' expression
record-ref:       identifier '.' identifier
assignment-expr:  lhs '<-' expression
lhs:              identifier | reference
parallel-expr:    map-expr | reduce-expr | scanl-expr | scanr-expr
map-expr:         function-name // array-expr
reduce-expr:      function-name /. array-expr
scanl-expr:       function-name /: array-expr
scanr-expr:       function-name :/ array-expr
function-name:    identifier
array-expr:       identifier | map-expr
function-call:    identifier '(' expression maybe-others ')'
maybe-others:     '' | ',' expression maybe-others
```

Note that an `lhs` may be the name of an (unindexed) array, in which case the corresponding assignment expression is compiled into a loop over the array, where each iteration performs a sub-assignment to one element of the array.

3.5 Control Structures

Most LOOM control structures are similar to those in traditional imperative languages. The exceptions are the *parallel operators*. Strictly speaking, LOOM programs run inside a single thread on a CUDA device, so in a sense these operators are not parallel by themselves. Rather, they are designed to encapsulate common patters for coordinating many threads to do work and share data in parallel.

3.5.1 Parallel Operators

Certain type restrictions apply to `parallel-expr` expressions. For each of the four operators, the function on the left must accept a value having the same type as the type contained within the array on the right, and the array's dimension should match the number of CUDA threads in the grid (in either one or two dimensions).

The map operator, `//`, produces an array with the same dimensions as the starting array (`THREADS` or `X_THREADS` × `Y_THREADS`). The reduce operator, `/.`, produces a scalar.

The two scan operators, `/:` and `:/`, produce arrays with dimensions `BLOCKS` or `X_BLOCKS` × `Y_THREADS` (in the case of two-dimensional arrays, the reduce and scan operators work over the first array index).

3.5.2 Statements and Blocks

```
statement:      expression ';'
              | block
              | if-stmt | for-stmt | while-stmt
              | 'break' ';'
              | 'continue' ';'
              | 'return' return-val ';'
block:          '{' statement other-stmts '}'
other-stmts:    '' | other-stmts
return-val:     '' | expression
```

Blocks are significant not only for grouping statements within control structures, but also because local variables have block scope.

The loop control statements `break` and `continue` act in the usual way on the innermost surrounding loop.

3.5.3 Conditionals

```
if-stmt:     'if' expression 'then' block maybe-else
maybe-else:  '' | 'else' block
```

LOOM follows the C convention for resolving `else` ambiguity: the `else` connects with the last-encountered `else`-less `if`.

3.5.4 Loops

```
while-stmt:  'while' expression block
for-stmt:    'for' loop-var '<-' expression dots expression block
loop-var:    identifier maybe-type
maybe-type:  '' | ':' type
```

If a loop variable is declared within the `for` statement, its scope is the surrounding block rather than the block in the body of the statement.

3.6 Functions

LOOM has two kinds of functions: kernel functions (of which there is exactly one per program, and which is called from code running on the host CPU) and device functions (which can be called by the kernel function or by other device functions). Due to limitations of the CUDA architecture, recursion is not permitted.

3.6.1 Kernels

```
kernel-func:  'kernel' identifier '(' parameters ')' block
parameters:   parm maybe-parms
parm:         identifier ':' type
maybe-parms:  '' | ',' maybe-parms
```

The kernel function is the only function that does not have a return type – it can affect the world only by altering the global variables passed in as arguments.

The `block` in the body of the kernel function must contain at least one `return` statement.

3.6.2 Device Functions

```
device-func:  'func' identifier '(' parameters ')' ':' type block
```

Device function definitions are similar to kernel functions, except that they have a return type and are introduced by a different keyword.

Arguments are generally passed by value except for arrays, which are passed by reference. In a function-call statement such as `f(1+1,2+2,a)`, expressions in the argument list are evaluated from left to right before the function call.

Chapter 4

Project Plan

Chapter 5

Architecture

Chapter 6

Testing Procedures

Chapter 7

Lessons Learned

Appendix: Source Listings