Loom Compiler Proposal

Johnathan Jenkins

June 8, 2011

Abstract

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

1 Introduction and Background

Loom is a language for programming massively parallel co-processors used on many modern computers. The specific devices supported are nvidia GPUs 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 GPU device). This simplifies loom, which is solely for compiling GPU 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.download.nvidia.com*.

2 Overview of Loom

2.1 Design Goals and Language Features

LOOM attempts to abstract away many of the repetitive and error-prone details involved in writing GPU 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.

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

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])
    current: Range <- block(B)
    shared t: Float32[] <- in[current]  -- size is BLOCKSIZE
    out[B] <- sum /. (sqr /~ t)
```