Ikra: Supporting Objects in a Ruby-to-CUDA Compiler

Matthias Springer Hidehiko Masuhara
Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Japan
matthias.springer@acm.org masuhara@acm.org

Abstract

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Class extension, context-oriented programming, mixins

1. Introduction

With the availability and affordability of powerful GPUs, general purpose computing on graphics processing units (GPGPU) is becoming more and more popular in high-performance computing. Nowadays, many supercomputers rely on GPUs as main processing units, because they allow for massively parallel execution of algorithms or simulations with thousands of threads per GPU. However, GPU programming differs from traditional CPU programming, mostly because of architectural differences.

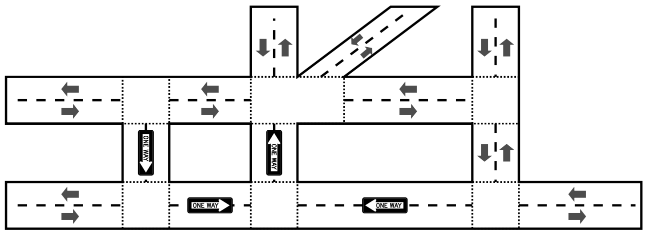
The goal of the Ikra project is to make GPU programming available to researchers who are not familiar with the details of GPUs their programming languages. Ikra is a library for Ruby that translates parallel sections to CUDA code and executes them in parallel on GPUs. We target the Ruby programming language because it provides powerful mechanisms for embedding DSLs in the language, which will be useful for later experiments.

2. Running Example

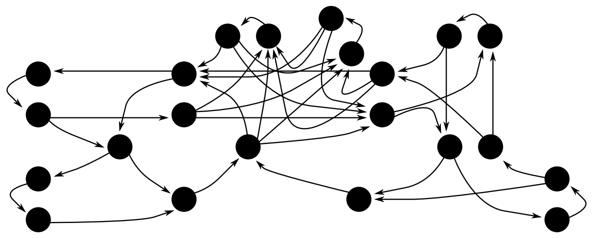
A simple actor-based traffic simulation will serve as a running example in this paper. The basic idea is to simulate the behavior of a number of actors (e.g., cars, buses, pedestrians, etc.), given a street network as a directed graph in adjacency list representation. Every actor is located on one street. Every street has a *length* attribute and every actor has a *progress* attribute representing the distance from the beginning of the street. Once these two attributes have the same value, the actor reached an intersection and should be moved to a different street (or make a U-turn).

3. Architecture

Ikra is a library for Ruby. It adds functionality to arrays to execute `map`, `select` and `each` operations in parallel. Programmers can *require* Ikra in Ruby files, upon which new parallel versions of array operations are available (e.g., `pmap`). These parallel array operations take a block as an argument and designate the only parts of a Ruby programs that are parallelized using Ikra. The default behavior is to spawn one thread per array element, which is why all these computations must be independent of each other. Every *tick* of the simulated progresses the current time by a certain constant value and actors are required to update their progress and street attributes accordingly.



(a) Actual street network (map)



(b) Street network as directed graph

Figure 1: Running Example

3.1 Compilation Process

Figure 2 gives a high-level overview of Ikra's compilation process. Upon invocation of a parallel section, Ikra acquires the source code of the parallel block, generates an abstract syntax tree (AST), and infers the type of all expressions. As a result, the type of every local and instance variable is known. In the best case, the type of an expression is monomorphic and primitive, but Ikra also supports arbitrary Ruby classes as types, as well as polymorphic types (see Section 4.1). The type inferer traverses invoked methods in a may-point-to fashion[1]. Based on the type-annotated AST, Ikra generates CUDA kernel code and boilerplate code for kernel invocation, and compiles the CUDA code using the nVidia CUDA toolchain. The result is a shared library which is loaded via Ruby's foreign function interface. Before kernel invocation, the base array along with all reachable objects (via instance variables) is transferred to the GPU's global memory. After kernel invocation, all changed objects and the result of the parallel section (if applicable) are written back to Ruby.

[1] Methods of all possible receiver types are taken into account.

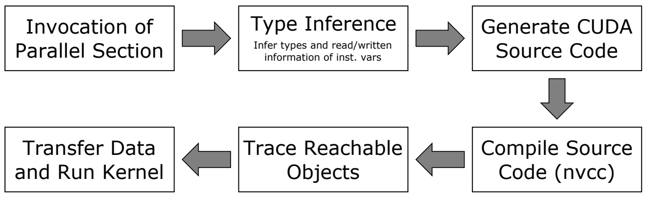

Invocation of Parallel Section
Type Inference
Infer types and read/written information of inst. vars
Generate CUDA Source Code
Compile Source Code (nvcc)
Trace Reachable Objects
Transfer Data and Run Kernel

Figure 2: Overview of Ikra's Architecture

3.2 Integration in Ruby

In contrast to some other projects, Ikra transforms Ruby code to CUDA code while the Ruby program is running (just-in-time compilation). Therefore, Ikra can determine the types of variables that are passed into parallel sections at runtime instead of doing a dataflow analysis of the entire program. This is not only faster but also more accurate in the light of reflection and metaprogramming, which is allowed outside of parallel sections but not inside them.

Two different kinds of variables can be used inside a parallel section: iterator variables and lexical variables. In the following code snippet, `el` is an iterator variable and `increment` is a lexical variable. The types of these variables are used as the foundation for type inference of the remaining parallel section.

```
increment = 10
[1, 2, 3].pmap do |el|
   el + increment
end
```

Programmers can use not only primitive objects (`Fixnum`, `Float`, etc.) but also objects which are instances of Ruby classes inside parallel sections, allowing for object-oriented modelling of the problem (e.g., a traffic simulation). Consequently, a graph of *reachable* (connected) objects must be transferred to the GPU. The *object tracer* is responsible for determining which objects should be copied to the GPU's global memory (see Section 4.4).

After kernel execution, changed local variables and instance variables are copied back to the Ruby side (see Section 4.3).

4. Implementation

In this section, we give an overview of some interesting aspects of Ikra's implementation.

4.1 Polymorphic Expressions

4.2 Read/write Analysis for Instance Variables

4.3 Copying back Variables

4.4 Object Tracer

5. Optimizations

6. Benchmarks

7. Related Work

8. Future Work

9. Conclusion