

Ikra: Object-oriented GPGPU Programming in Ruby with CUDA

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.

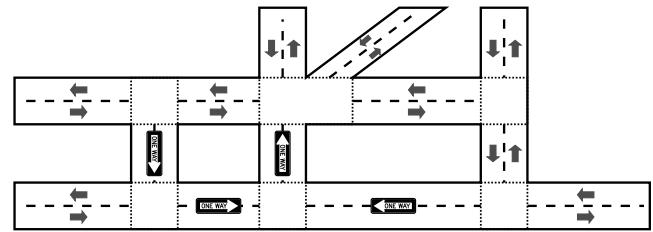
The goal of the Ikra project is to make GPU programming available to researchers who are not familiar with the details of GPUs their programming languages. Ikra is a library for Ruby that translates parallel sections to CUDA code and executes them in parallel on GPUs. We target the Ruby programming language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

2. Example: Traffic Simulation

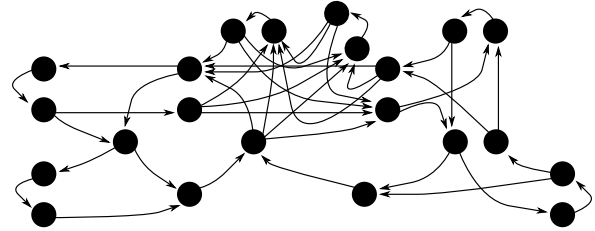
A simple actor-based traffic simulation will serve as a running example in this paper. The basic idea is to simulate the behavior of a number of actors (e.g., cars, buses, pedestrians, etc.), given a street network as a directed graph (Figure 1) in adjacency list representation. Every actor is located 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 if there is no other neighboring street).

A car moves at a constant speed of `@max_velocity`. A pedestrian moves at a random speed between $-0.5 \times @max_velocity$ and `@max_velocity`, i.e., a pedestrian can make negative progress. This is how we model strolling pedestrians. Furthermore, the progress of actors might be affected by weather conditions depending on their

type. For example, cars slow down if the weather conditions are bad, whereas pedestrians are not affected by weather conditions.



(a) Actual street network (map)



(b) Street network as directed graph

Figure 1: Street Network for Traffic Simulation

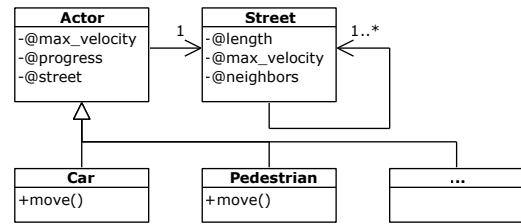


Figure 2: Class Diagram for Traffic Simulation

Data Structure The street network and the actors are designed in an object-oriented way. Figure 2 shows the class organization of the traffic simulation. Car and Pedestrian are subclasses of Actor and provide their own move methods which will be invoked for every tick of the simulation.

Main Simulation Loop The following code snippet contains the main simulation functionality. The method `pmap` designates a parallel section. Its parameter `ticks` determines how often the entire

peach statement should be executed and is equivalent to wrapping the peach statement in a loop that executes it ticks times¹.

```
actors = [...]
ticks = 1000
weather = Weather::Rainy

actors.peach(ticks) do |actor|
  actor.move(weather)
end
```

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.

3.1 Compilation Process

Figure 3 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 4.3). The type inferer traverses invoked methods in a may-point-to fashion². 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.

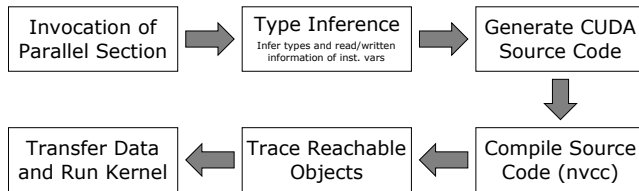


Figure 3: Overview of Ikra’s Architecture

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.

¹ Ikra does not support nested loops properly, which is why we suggest using this shortcut. See Section 8.1 for more details.

² Methods of all possible receiver types are taken into account.

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 4.6).

After kernel execution, changed local variables and instance variables are copied back to the Ruby side (see Section 4.5).

4. Implementation and Optimizations

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

4.1 Job Reordering

Before kernel invocation, Ikra analyzes all elements in the base array and reorders them according to their type. This is useful to avoid *thread divergence*, which can penalize performance when running programs on GPUs.

Thread Divergence In contrast to most CPU-based systems³, GPU-based systems are SIMD (single instruction, multiple data) systems. A GPU consists of a number of streaming multiprocessors. Such a processor has a single control unit that fetches and decodes instructions, but multiple arithmetic logic units (ALUs). Therefore, every instruction is executed in parallel on multiple chunks of data. Every ALU corresponds to one thread, but all threads that are executing on the same stream multiprocessor must follow the same control flow. In case two threads take a different branch, their execution is serialized until the control flow merges again. Consequently, jobs should be threads/jobs should be mapped to streaming multiprocessors in such a way that the control flow is unlikely to diverge among one such thread group (threads executing on one stream multiprocessor).

In CUDA, such a thread group is called *warp* and typically has a size of 32. There is no explicit interface to allocate threads to warps, but CUDA programmers try to write their programs in such a way that each consecutive group of 32 threads follows the same control flow.

Thread Allocation Ikra tries to avoid thread divergence by allocating jobs to warps automatically based on type information. Before kernel invocation, Ikra generates a *job reordering array*, such that the base array is sorted according to the elements’ types (Figure 4). Ikra does not actually change the order elements in the base array to ensure that other parts of the program outside of the parallel section are not affected.

During job reordering the number of threads can increase as shown in Figure 4. Jobs are reordered in such a way that no two elements of different type are allocated in the same warp. If the number of jobs of a particular type is not a multiple of the warp size, the last warp will not be filled up entirely, but some threads will not have a job, i.e., they are not *no operation* threads. This might

³ There are CPU extensions for SIMD computations, e.g., SSE.

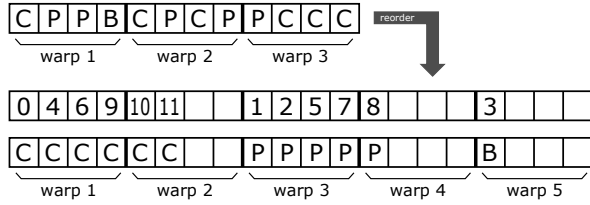


Figure 4: Example: Job Reordering. The first row is the original job order, the second row is the job reordering array, and the third row shows the resulting job order (warp size 4).

seem like a waste of computing power, but we expect the number of different types to be small (3 in this example).

The job reordering array can be computed in linear time by scanning all elements of the base array twice. The following pseudo code is similar to counting sort and bucket sort [3]. It generates one array of indices per type (class) and concatenates these arrays, making sure that every new array starts at a multiple of the warp size W .

Algorithm 1 Job Reordering

```

1: procedure REORDERINGARRAY(base, W)
2:    $types \leftarrow \text{Hash.new}$ 
3:   for all  $(el, idx) \in \text{base}$  do
4:      $types[el.class].add(idx)$ 
5:   end for
6:    $result \leftarrow \text{Array.new}(\sum_{arr \in types.values} \lceil |arr|/W \rceil * W)$ 
7:    $next \leftarrow 0$ 
8:   for all  $arr \in types.values$  do
9:     for all  $idx \in arr$  do
10:       $result[next] = idx$ 
11:       $next \leftarrow next + 1$ 
12:    end for
13:     $next \leftarrow \lceil next/W \rceil * W$ 
14:  end for
15:  return  $result$ 
16: end procedure

```

4.2 Columnar Object Layout

Objects are typically represented row-wise, i.e., every object is a contiguous chunk of data in the memory. Based on observations in previous work on GPU-powered database query execution [2] where a column-wise data organization proved to be superior compared to a traditional row-based data layout, Ikra stores objects in a columnar layout [4].

4.3 Polymorphic Expressions

In contrast to CUDA, Ruby is a dynamically-typed programming language. One of the goals of the Ikra project is to allow programmers to write Ruby code in a *natural* way, i.e., without having to think about GPU/CUDA characteristics. For this reason, Ikra should support Ruby expression whose types cannot be inferred unambiguously at translation time.

Ikra embeds dynamic types into CUDA’s static type system by generating explicit type dispatch statements at method call sites based on type tags [1] for receivers whose types cannot be inferred unambiguously. From a perspective of object-oriented design, this looks as if every object has a *type* instance variable.

4.4 Read/write Analysis for Instance Variables

4.5 Copying back Variables

4.6 Object Tracer

5. Optimizations

6. Benchmarks

7. Related Work

8. Future Work

8.1 Nested Loops

Ikra does not yet support nested loops properly. Putting a ticks loop inside the peach block works but contradicts intuition. In a sequential program, most programmers would formulate the simulation code as a series of simulation ticks, where every simulation tick iterates over all actors, as opposed to iterating over all actors, where every actor is moved for a series of simulation ticks.

```

actors.peach do |actor|
  for i in 1..ticks
    actor.move(weather)
    synchronize
  end
end

```

The following code snippet is more intuitive, but would allocate one thread per tick instead of one thread per actor. However, the mechanism described in this paper takes advantage of allocating threads based on the actors’ types.

```

(1..ticks).peach do
  actors.each do |actor|
    actor.move(weather)
  end
end

```

9. Conclusion

References

- [1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’89, pages 213–227, New York, NY, USA, 1989. ACM.
- [2] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 94–103, New York, NY, USA, 2010. ACM.
- [3] E. Corwin and A. Logar. Sorting in linear time - variations on the bucket sort. *J. Comput. Sci. Coll.*, 20(1):197–202, October 2004.
- [4] Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. Columnar objects: Improving the performance of analytical applications. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 197–210, New York, NY, USA, 2015. ACM.