Object Support in an Array-Based GPGPU Extension for Ruby

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

Abstract
This paper presents implementation and optimization techniques to support objects in Ikra, an array-based parallel extension to Ruby with dynamic compilation. The high-level goal of Ikra is to allow developers to exploit GPU-based high-performance computing without paying much attention to intricate details of the underlying GPU infrastructure and CUDA.

Ikra supports dynamically-typed object-oriented programming in Ruby and performs a number of optimizations. It supports parallel operations (e.g., map, each) on arrays of polymorphic objects, allowing polymorphic method calls inside a kernel by compiling them to conditional branches. To reduce branch divergence, Ikra shuffles thread assignments to base array elements based on runtime types of elements. To facilitate memory coalescing, Ikra stores objects in a structure-of-arrays (SoA) representation (columnar object layout). To eliminate intermediate data in global memory, Ikra merges cascaded parallel sections into one kernel using symbolic execution.

Categories and Subject Descriptors D.1.3 [Concurrent Programming]: Parallel Programming; D.3.4 [Processors]: Code generation, Compilers

Keywords GPGPU, CUDA, Ruby, object-oriented programming

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 developers who are not familiar with the details of GPUs and their programming languages. Ikra is a library for Ruby that translates parallel sections to CUDA code and executes them in parallel on GPUs. It extends our previous work [14] with a dynamic compilation approach to allow for a larger number of optimizations and tighter integration with Ruby. To that end, Ikra also supports polymorphic expressions and variables, allowing programmers to write Ruby code in a natural way, i.e., programmers should be able to write the same source code that they would write in a standard Ruby environment. We target the Ruby programming language because it provides powerful mechanisms for embedding DSLs in the language, which will be useful for future work.

2. Example: Agent-based Traffic Simulation
A simple object-oriented, agent-based traffic simulation will serve as a running example in this paper. The basic idea is to simulate the behavior of a number of agents (e.g., cars, buses, pedestrians, etc.), given a street network as a directed graph (Figure 1) in adjacency list representation. Every agent is located on one street. Every street has a length attribute and every agent has a progress attribute representing the distance from the beginning of the street. Once these two attributes have the same value, the agent reached an intersection and should be moved to a different street (or make a U-turn if there is no other neighboring street).

\[
\text{(a) Actual street network (map)}
\]

\[
\text{(b) Street network as directed graph}
\]

Figure 1: Example: Street Network for Traffic Simulation

A car moves at a constant speed of \( \min(M_c, M_s) \), where \( M_c \) is the maximum velocity of the car and \( M_s \) is the maximum speed allowed on the current street. A pedestrian moves at a random speed between \(-2\) mph and \(4\) mph, i.e., a pedestrian can make negative progress. This is how we model strolling pedestrians. Furthermore, depending on their type, the progress of agents might be affected by weather conditions. For example, cars slow down if the weather conditions are bad, whereas pedestrians are not affected by weather.

Data Structure The street network and the agents are designed in an object-oriented way. Figure 2 shows the class organization of the traffic simulation. Car and Pedestrian are subclasses of...
Main Simulation Loop: 

The following code snippet contains the main simulation functionality. The method `peach` 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` times.

```ruby
agents = # load scenario from file system
ticks = 1000
weather = Weather::Rainy

agents.peach(ticks) do |agent|
  agent.move(weather)
end
```

Every `tick` of the simulation progresses the current time by a certain constant value and agents are required to update their progress and street attributes accordingly.

3. Architecture

Ikra is a library for Ruby. It adds functionality to arrays to execute map, reduce, 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., `map`). These parallel array operations take a block as an argument and designate the only parts of a Ruby programs that are parallelized using Ikra.

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 4.2). The type inference engine traverses invoked method bodies for all possible receiver types (union type). Based on the type-annotated AST, Ikra generates CUDA kernel code and initialization 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 and lexical variables along with all reachable objects (via instance variables) are 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.

4. Implementation and Optimizations

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

4.1 Symbolic Execution

Parallel array operations (except for `peach`) are executed symbolically (lazily) in Ikra. They can be cascaded and are executed only if the result is actually accessed. The default behavior is to spawn one thread per array element, which is why all these computations must be independent of each other.

Ikra performs kernel fusion \[24; 23\], i.e., cascaded parallel operations are merged into a single CUDA kernel to avoid reading/writing intermediate results from/to the global memory. Instead, they can be kept in registers. The process of merging two parallel blocks is not relevant in the scope of this paper and omitted.

4.2 Polymorphic Expressions

In Ikra, non-primitive object references are represented by an ID (pointer). The types of polymorphic expressions (also types that are not in a subtype relationship) are embedded into CUDA’s static type system using class tags \[3\]. The type inference engine uses union types for polymorphic expressions and ignores inheritance relationships. A value of a union type is represented by a tuple (C++

---

1. There are currently certain limitations for loop nesting. See Section 7.3 for more details.

2. Instance variables can be used when calling an instance method.
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 streaming 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, threads/jobs should be mapped to streaming multiprocessors in such a way that the control flow is unlikely to divergence among one such thread group (threads executing on one streaming multiprocessor).

In CUDA, such a thread group is called *warp* and has a size of 32. CUDA programmers try to write their programs such that each consecutive group of 32 threads follows the same control flow.

**Thread Allocation** Ikra tries to avoid branch divergence due to polymorphic method calls on array elements by allocating jobs to warps automatically based on runtime type information. Before kernel invocation, Ikra generates a *job reordering array* (see *jobs* parameter in Figure 4), such that the base array is sorted according the elements’ types (Figure 5). Ikra does not actually change the order of 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 5. Jobs are reordered in such a way that no two elements of different types 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, so some threads will not have a job, i.e., they are *no operation* threads. This might 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 algorithm is similar to counting sort and bucket sort [6]. 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.

**4.4 Structure-of-Arrays Representation**

In traditional programming languages and virtual machines, an array of objects is typically represented as an array of structures, i.e., every object is a contiguous chunk of data in the memory. However, it is common practice in GPU programming to work with multiple arrays of structure fields (*structure-of-arrays*) instead of one array of structures (*array-of-structures*) for coalescing field accesses [16; 4].

**Memory Coalescing** Global memory is one of the main bottlenecks of GPUs. One approach is to aim for memory access patterns where memory that is accessed in parallel by a number of threads is spatially local. Such memory accesses can be coalesced, i.e., the GPU can process such accesses in a single request, alleviating the global memory bottleneck.

Since a GPU is a SIMD system, all threads within a warp have to execute the same instruction at a time. Consequently, if one thread accesses an instance variable, then all other threads within the same warp access the same instance variable (or block because of branch divergence), probably in a different object. In this situation, a structure-of-arrays layout is superior to an array-of-structures layout, because parallel accesses to the same instance variable are more likely to be spatially local (Figure 6).

**Generating Structure of Arrays** In the following, we present a first approach for representing objects as a structure of arrays. After running the object tracer, we know which objects should be transferred to the GPU. Objects are grouped by class and assigned class-specific IDs, which are used as indices into the newly-created structure of arrays (similar to the *system tracer* in Smalltalk [13]).

1. Group objects by their class $c$, resulting in arrays $O_c$. 

---

3 There are CPU extensions for SIMD computations, e.g., SSE.
2. Assign an ID to every object for all $O_c$, starting from 0 in every $O_c$. IDs being consecutive. This results in a hash map $H$, mapping objects to class-specific IDs.

3. For every instance variable $v$ of every class $c$, create an array $A_{c,v}$ of size $m + 1$, where $m$ is the maximum ID in $O_c$. The base type of the array is the type of the instance variable if it is primitive, or the union type struct if polymorphic, or int otherwise (referencing other non-primitive objects via their IDs).

4. For every object $o$ with class $c$ and ID $H_c[o]$, store every instance variable $v$ in the corresponding array slot $A_{c,v}[H_c[o]]$. If the instance variable is non-primitive, look up its ID and store it.

Note that after this transformation, the base type of the base array that is passed to the kernel, contains object IDs (or union type structs) if it consists of non-primitive objects (see agent parameter in Figure 4). In our implementation, the object tracer is combined with the SoA generator.

**Source Code Transformation** Since objects are now represented as a structure of arrays, Ikra must generate different source code for reading from or writing to instance variables. We do not consider generating new objects at this time (see Section 7).

In the following, we consider reading/writing instance variables of an object and calling methods on an object, where the object is identified by its type $c$ and its ID $i$ (or a union type struct). Whenever objects are passed around, Ikra generates source code that passes their IDs (or union type structs for polymorphic values) around.

Reading/writing an instance variable $v$ of object $o$ with type $c$ and ID $i$ translates to reading/writing the array $A_{c,v}[i]$.

Every instance method is translated to a device function, where the type $c$ is mangled into its name and the first parameter is the ID (or union type struct) of the self object. Whenever Ikra encounters a method call during code translation, it generates a call to the appropriate device function (or a switch-case statement for polymorphic expressions).

**Representation of Arrays** The previously-described SoA object layout works well with equally-sized objects, but not for arrays, which are variable-sized objects. For example, the instance variable @neighbors of class Street is an array of streets.

Ikra effectively represents such $n : m$ relationships as join tables that are collapsed. Such a table is sorted by object IDs for $n$. Furthermore, the $n$ array (column) is not stored as a full array but as RLE tuples consisting of an implicit ID for $n$, a start offset into the $m$ array, and a length value, distributed among multiple columns. RLE tuples are a well-known optimization in column databases.

From an implementation point of view, an array is an ordinary object with an offset and a size attribute. The offset attribute points into a single large array containing the contents of all arrays. This layout might change in future versions of Ikra (see Section 7).

28
one used), running Linux 3.0.76-0.11-default x86_64, CUDA 7.0.27, and Ruby 1.9.3p448.

We simulated 1,000,000 iterations of a random street network with 500 streets and random vertex out-degrees between 1 and 10, 4,096 cars, and 16,384 pedestrians. All benchmark running times are average values of 5 runs using the same random scenario.

**Kernel Execution** Figure 8 shows the kernel running time of the traffic simulation in various configurations. Figure 8a shows the running time with a structure-of-arrays (SoA) layout, which is around 30% faster than an array-of-structures (AoS) layout in Figure 8b. There are around 10 accesses to instance variables in each move method. The source code is omitted for brevity reasons in Figure 4.

**Kernel Running Time**

\[
\text{Figure 8}: \text{Kernel Running Time for Traffic Simulation (CUDA)}
\]

The objects involved in this example are quite small. Agents are represented by 32-byte structs (three instance variables and a class tag) in an AoS layout, or three 4-byte arrays (class tag is passed as an argument) in a SoA layout, respectively. The GPU’s L1 cache is 48 KB, with a cache line size of 128 bytes. To analyze the effect of prefetching, we ran the AoS benchmark with artificially enlarged object sizes (10/5 additional instance variables that are never read or written for agents/streets, resulting in an object size of 72 bytes/32 bytes, respectively; Figure 8c). This configuration is interesting because the example code accesses all instance variables of an agent subsequently, diminishing the advantage of a SoA layout, because the entire object (and three subsequent objects) can be held in cache (prefetching). A SoA layout is around 60% faster compared to this configuration.

The running time for transferring data to the GPU and generating the CUDA code is not included in this benchmark. These two steps are currently clearly dominated by the execution time of the nVidia compiler (nvcc), which is around 2-4 seconds. We hope to be able to reduce the running time of this step by caching and by generating LLVM intermediate code from Ruby bytecode, which would skip some steps in the CUDA compilation process (see Section 6).

**Job Reordering** Figure 9 shows the running time for generating the job reordering array (warp size 32). This is currently done in the Ruby interpreter but could be moved to the GPU side in future versions. The running time increases linearly with the number of elements in the base array. Changing the number of types (classes) has only a small effect on the running time. We assume that this number is much smaller than the number of elements. For the traffic simulation example, the running time for generating the job reordering array is neglectable.

**Tracing Objects and Generating Structure of Arrays** Figure 10 shows the running time for tracing objects and generating the SoA layout for the traffic simulation example. For the number of agents/streets that were used in the kernel benchmarks, the running time is neglectable. Moreover, in future versions of Ikra we want to perform this step only once and reuse data that was already processed and moved to the GPU earlier (see Section 7). The running time for type inference is not included in this benchmark.

**Related Work**

Columnar (SoA) data layouts are known to be superior compared to row-based data layouts for certain kinds of database queries (e.g., OLAP queries) [19], and especially for GPU-powered databases [5]. In fact, one of the benefits of column stores for CPU-based database systems is prefetching, which is similar to coalescing on GPUs, but without the parallel aspect. Columnar data layouts have also been evaluated for object-oriented programming languages. Mattis et al. have implemented a columnar object layout in Pypy to increase the performance of analytical queries [15]. Ikra essentially uses the same columnar object layout, but extended to polymorphic types.

A number of different techniques exist for avoiding branch divergence. Most of them are proposed at application-level, while Ikra aims at enabling a similar technique at the language level. For example, one technique is to detect and delay divergent branches at runtime in order to execute them at a later time [8], or factoring out instructions that are common to two (divergent) branches [10]. A different approach is to reorder jobs, either with a reordering array (which is what Ikra does) or by physically changing the order of the jobs in the base array. Both techniques can be combined to increase memory coalescing [26] (physically reordering data, then using a reordering array to restore the original semantics), but detailed knowledge about memory access patterns is required. Previous work has also investigated how the overhead of these transformations can
be hidden using a CPU-GPU pipelining scheme [25], if done at runtime in order to react to changes.

Ishizaki et al. presented a framework for executing lambda expression used with the parallel streams API in Java 8 programs on nVidia GPUs [12]. Their approach is to generate LLVM intermediate code (IR) from Java bytecode, which is in our opinion superior to Ikra’s approach of performing a Ruby-to-CUDA source-code-to-source-code transformation from an engineering point of view. Future versions of Ikra might generate LLVM IR code from YARV bytecode [21]. Further optimizations of their Java 8 compiler include a check for array aliasing (which is what Ikra’s object tracer does implicitly) and utilizing a read-only cache. Their implementation supports virtual method calls using direct devirtualization if the receiver type can be uniquely determined at compile time, or guarded devirtualization, executing an iteration on the GPU if the guard fails; in either case, the GPU code must only be able to handle monomorphic method calls. Ikra’s code generator applies direct devirtualization, i.e., it does not generate type dispatch statements if the receiver type is monomorphic and can be inferred unambiguously. Otherwise, it generates CUDA code that dispatches to the correct method based on class tags.

Firepile is a Scala-to-CUDA compiler [17]. It supports Scala classes and generates a struct definition per class. The first field has as type the struct type for the superclass and is needed for inherited instance variables. The struct for object contains a class tag used for method dispatch. Ikra passes and stores class tags together with object IDs. If class tags were stored in one column shared by all objects, then all objects (and types) would have to follow the same numbering scheme, which would lead to sparse columns and a waste of global memory. The same problem occurs already in a less severe form in the light of subclassing (see Section 4.4 null values).

7. Future Work

This section gives a brief overview of ideas for future work on Ikra.

7.1 Minimizing Data Transfers

Our current Ikra implementation transfers objects to the GPU’s global memory every time a kernel is invoked. However, memory access is one of the main bottlenecks of GPUs and should be avoided. Future versions of Ikra will try to minimize data transfers by only transferring changed objects during consecutive kernel invocations, even if two different kernels were invoked. Similarly, objects should only be transferred back to the Ruby side once they are actually accessed. One approach replaces instance variable accessors with code that retrieves the actual value from the GPU and caches it.

It is our vision that the parallel CUDA code is in full control of instance creation. The only reason for transferring data to the GPU should be cases where an object graph is loaded from an external source or must be swapped from/to the main memory. For example, a researcher might want to load a street network of a real city from the file system. It is then not necessary to allocate this data structure both on the GPU and on the Ruby side. The Ruby program might, however, access certain objects and some of their instance variables for UI purposes or to display the result of a computation.

7.2 Data Modification

Ikra’s capabilities to modify data inside a parallel section are still limited, nevertheless sufficient for use cases like agent-based traffic simulations or OLAP applications, where data is mostly static.

For example, new objects can be created only on the Ruby side, but not inside parallel sections. This is because instance variable arrays must be increased in size when adding new objects. However, increasing their size might require moving them to a different place in the global memory, which is expensive.

As another example, it is currently not possible to add or remove elements from an array[i]. Future versions of Ikra might store arrays separately instead of using a single big array (Figure 7). Instead of storing an offset, this would require storing a pointer.

Ikra does currently not expose CUDA synchronization constructs [7] or atomic operations. However, these constructs are necessary if computations between two threads are not independent.

7.3 Nested Loops

Ikra does not yet support nested loops properly. Consider the main loop of the traffic simulation as an example. Putting a ticks loop inside the peach block works (only because there is no synchronization necessary in this example) 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 agents, as opposed to iterating of over all agents, where every agent is moved for a series of simulation ticks:

\[
\begin{align*}
\text{agents}.\text{peach do} & \text{ |agent|} \\
& \text{for i in 1..ticks} \\
& \quad \text{agent.move(weather)} \\
& \text{end} \\
\end{align*}
\]

The following code snippet is more intuitive, but would allocate one thread per tick instead of one thread per agent (and works only if there are no data dependencies between ticks). However, the mechanism described in this paper takes advantage of allocating threads based on the agents’ types.

\[
\begin{align*}
\text{(1..ticks)}.\text{peach do} & \text{ |agent|} \\
& \text{agents.each do} \text{ |agent|} \\
& \quad \text{agent.move(weather)} \\
& \text{end} \\
\end{align*}
\]

Nesting two parallel peach statements within each other is not supported at the moment. Parallelization is allowed only on one level. In future versions of Ikra, data will be held in the global memory as long as possible and only be transferred from/to the Ruby side if necessary (see Section 7.1). In such a situation, code that is similar to the one shown in the previous listing could be written with less kernel invocation overhead.

7.4 Performance Optimizations

Further ideas for performance optimizations include taking advantage of shared memory, which is much faster than global memory. However, it is not obvious what kind of data to store in shared memory because of its limited size.

Ikra’s SoA object layout is similar to the data layout of column databases. Future work might investigate to what degree optimizations in the area of column databases [1] are applicable to a column-based object graph. Data compression mechanisms for minimizing data transfer time look particularly promising, given the performance gap between global memory and shared memory, and have been subject to previous work in GPU computing [18][20].
We presented Ikra, a dynamic Ruby-to-CUDA compiler for array-based parallel operations. Ikra allows programmers to write source code in an object-oriented way and applies optimizations to reduce branch divergence and to increase memory coalescing. Runtime type information is used to reorder objects in the base array, making sure that only objects of the same type are executed in a warp. A structure-of-arrays object layout is beneficial for memory coalescing, because threads inside a warp are executed in a SIMD manner. Future work will focus on additional performance optimizations and take into account a broader set of examples and benchmarks.

8. Summary

We presented Ikra, a dynamic Ruby-to-CUDA compiler for array-based parallel operations. Ikra allows programmers to write source code in an object-oriented way and applies optimizations to reduce branch divergence and to increase memory coalescing. Runtime type information is used to reorder objects in the base array, making sure that only objects of the same type are executed in a warp. A structure-of-arrays object layout is beneficial for memory coalescing, because threads inside a warp are executed in a SIMD manner. Future work will focus on additional performance optimizations and take into account a broader set of examples and benchmarks.

References


