Accepted for publication at ARRAY 2017

Modular Array-Based GPU Computing in a Dynamically-Typed Language

Matthias Springer Peter Wauligmann Hidehiko Masuhara
Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Japan
matthias.springer@acm.org peter.wauligmann@tum.de masuhara@acm.org

Abstract

Nowadays, GPU accelerators are widely used in areas with large data-parallel computations such as scientific computations or neural networks. Programmers can either write code in low-level CUDA/OpenCL code or use a GPU extension for a high-level programming language for better productivity. Most extensions focus on statically-typed languages, but many programmers prefer dynamically-typed languages due to their simplicity and flexibility.

This paper shows how programmers can write high-level modular code in *Ikra*, a Ruby extension for array-based GPU computing. Programmers can compose GPU programs of multiple reusable parallel sections, which are subsequently fused into a small number of GPU kernels. We propose a seamless syntax for separating code regions that extensively use dynamic language features from those that are compiled for efficient execution. Moreover, we propose symbolic execution and a program analysis for kernel fusion to achieve performance that is close to hand-written CUDA code.

CCS Concepts • **Software and its engineering** → **Source code generation**; • **Computing methodologies** → *Parallel programming languages*; • **Theory of computation** → Type theory

Keywords GPGPU, CUDA, Ruby, kernel fusion

1. Introduction

In recent years, one area of research in GPU computing focuses on high-level languages, making the performance gap between highly optimized low-level programs and high-level programs closer and closer. A variety of tools emerged that let programmers write parallel programs for execution on GPUs in a high-level language. Most tools are extensions or libraries for existing high-level languages [2, 9, 19]. Their goal is not to reach peak performance. With sufficient expert knowledge about CUDA/OpenCL and the underlying hardware platform, it is possible to write highly optimized low-level programs that perform better. However, previous work [15] has shown that writing code in a high-level language is easier and more productive also in the area of high-performance computing.

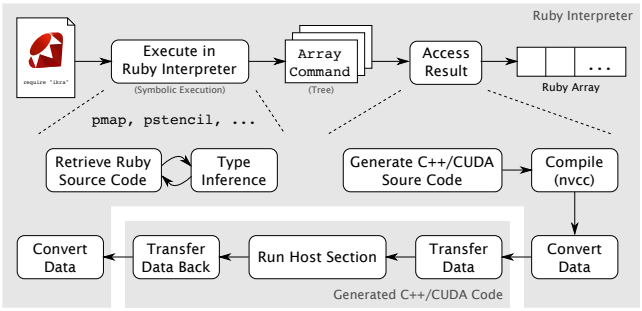


Figure 1: High-level Overview of Compilation Process

Ikra[1] is a language extension for data-parallel and scientific computations in Ruby on Nvidia GPUs. It uses arrays as an abstraction for expressing parallelism. Ikra provides parallel versions of map, reduce and a construct for stencil computations. When using Ikra, we encourage a *dynamic* programming style that is governed by the following two concepts.

Integration of Dynamic Language Features Code in parallel sections is limited to a restricted set of types and operations (dynamic typing and object-oriented programming is allowed). All Ruby features (incl. metaprogramming) may still be used in other parts. Therefore, programmers can still use external libraries (e.g., I/O or GUI libraries).

Modularity [11] While optimized low-level programs typically consist of a small number of kernels performing a variety of operations, Ikra allows programmers to compose a program from multiple reusable, smaller kernels.

Due to dynamic language features, whole-program static (ahead of time) analysis is difficult. Therefore, Ikra generates CUDA programs at runtime (just in time) when type information is known. Moreover, Ikra optimizes GPU programs using two techniques that are well-known in statically-typed languages but not in dynamically-typed languages such as Ruby. First, it fuses multiple kernels [16, 3, 12] into a single kernel. Such code can be faster because data can remain in registers and does not have to be transferred from/into slow global memory. Second, loops surrounding parallel code are compiled to C++ code and not executed in the Ruby interpreter. Microbenchmarks show that both techniques together achieve performance that is comparable to a single hand-written kernel.

[1] Project Website:
`https://prg-titech.github.io/ikra-ruby/`

Compilation Process Ikra is a RubyGem (Ruby library) that provides parallel versions of commonly used array operations. The notation and API for these operations is similar to Ruby's counterparts but method names are prefixed with `p` for *parallel*.

Figure 1 gives a high-level overview of the Ikra's compilation process. When one of Ikra's parallel operations is invoked in the Ruby interpreter, Ikra executes that operation symbolically. The result is an *array command* object. Such an object contains all information required for CUDA code generation and execution. An array command can be used like a normal Ruby array. However, only when its contents are accessed for the first time, Ikra generates CUDA/C++ source code, compiles it using the Nvidia compiler and runs the generated C++ program. The generated program copies data to the GPU, launches the parallel sections, copies the result back to the host memory and returns the result[2].

Instead of defining single parallel sections, programmers can also define *host sections*. A host section is a block of Ruby code that contains a more complex program with multiple parallel sections. In such a case, the entire block is translated to C++ code, avoiding switching from the Ruby interpreter to external C++ programs multiple times. The former case can be seen as a host section which directly returns the result of a single parallel section. Thus, we only mention the general case "Run Host Section" in Figure 1.

Symbolic Execution During symbolic execution, Ikra retrieves the source code of parallel sections (e.g., the body of a `pmap` operation), generates abstract syntax trees and infers types. The result of symbolic execution is an array command. Ikra currently supports various primitive types (`Fixnum`, `Float`, booleans, `NilClass`), user-defined classes and polymorphic types. A polymorphic type is represented by a pair of type/class ID and the actual value (*union type*).

Programmers can invoke other methods inside parallel sections, including method calls on objects. After Ikra has determined the receiver type(s) of a method call during type inference, the target method(s) are added to a work list of methods to be processed next.

Array Commands A command object in the Command Design Pattern [5] is an object that contains all information that is necessary to perform an action at a later point of time. An array command in Ikra is an object that contains all information required for code generation and launching a parallel section/host section. For example, a stencil array command contains the typed AST of the computation (block), a reference to the input array command, an array of neighborhood indices, and an out of bounds value.

An array command can be seen as a special *Ikra array*. It has all methods that an ordinary Ruby array has, but its content is computed once it is accessed. Figure 2 gives an overview of Ikra's integration in Ruby and the design of array commands. There are a variety of subclasses of `ArrayCommand` corresponding to parallel operations that are supported in Ikra (Section 2). The standard Ruby `Array` and Ikra's `ArrayCommand` include both mixins `Enumerable` and `ParallelOperations`. The first mixin provides standard collection API functionality. The second mixin provides parallel operations which are executed on the GPU.

2. Parallel Operations

This section gives an overview of operations that are provided by Ikra. All operations can handle multidimensional Ikra arrays, making code more readable if data is inherently multidimensional (e.g., images). If an operation performs a computation, then the size of the first argument determines the number of CUDA threads that are allocated.

[2] If object-oriented programming is used inside a kernel, data is first converted to a memory coalescing-friendly *structure of arrays layout* [14].

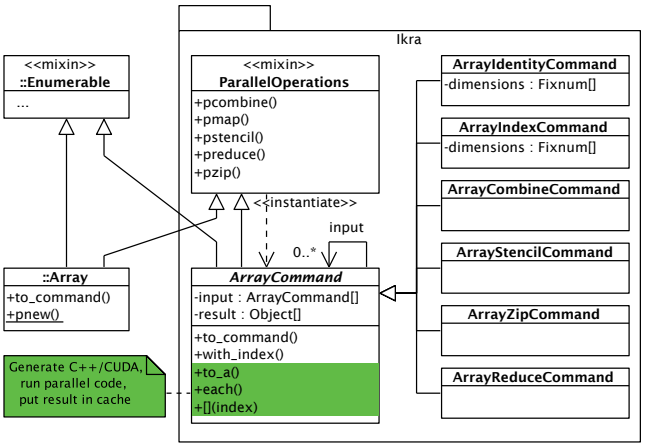


Figure 2: Integration of Ikra in Ruby

Array Identity This operation creates an Ikra array (array command) from an ordinary Ruby array. It can be used to load an external Ruby array A (not computed on the GPU) and make it available in Ikra. The star in A_* indicates that it is an Ikra array.

$$id(A) = \begin{cases} A_*, & \text{if } A \text{ is a Ruby array} \\ A, & \text{if } A \text{ is an Ikra array} \end{cases}$$

Array identity is applied implicitly where required. For example, when a map operation is applied to a Ruby array, Ikra applies this operation automatically. However, it is useful if programmers want to convert a one dimensional Ruby array to a multidimensional Ikra array. It is exposed to Ruby programmers as `to_command`, taking an optional parameter for dimensions.

```
A.to_command()
A.to_command(dimensions: [15, 20])
```

Ikra arrays can be converted back to Ruby arrays with `to_a`, which is recommended for performance reasons if large parts of the Ikra array are read randomly.

Combine This operation is used to map over one or more arrays A_i of same size m and dimensions. It takes as input n arrays and a block (anonymous function) f taking n scalar values. It applies f to every element of the input and retains the original shape of the input, regardless of dimensions.

$$\underset{\textit{(multidim. case omitted)}}{combine(A_1, \ldots, A_n, f)} = \begin{bmatrix} f(A_1[0], \ldots, A_n[0]) \\ \vdots \\ f(A_1[m-1], \ldots, A_n[m-1]) \end{bmatrix}$$

Ikra allocates m CUDA threads, i.e., every thread processes one tuple. This will likely change in future versions of Ikra. This operation is exposed to Ruby programmers as `pcombine`:

```
A1.pcombine(A2, ..., An, &f)
```

Map This operation is a special case of *combine* with one input array. It corresponds to an ordinary map operation but is executed in parallel.

$$map(A_1, f) = combine(A_1, f) = [f(A_1[0]), \ldots, f(A_1[m-1])]$$

This operation is exposed to Ruby programmers as `pmap`:

```
A1.pmap(&f)
```

Index This operation generates an array of size m of consecutive indices starting from 0 and ending with $m-1$.

$$index(m) = [0, 1, 2, \ldots, m-1]$$

In a multidimensional case, *index* takes d arguments m_i (d is the number of dimensions), where m_i is the size of the i-th dimension. Every value in the resulting array is then an array of size d containing the indices for every dimension.

$$index(m_1, m_2) = \begin{bmatrix} [0,0], \ldots, [0, m_1 - 1], \ldots, \\ [m_2 - 1, 0], \ldots, [m_2 - 1, m_1 - 1] \end{bmatrix}$$

This operation is not directly exposed to programmers. Similar to Ruby notation, programmers must invoke the method `with_index` after a parallel operation (the parameter m is provided implicitly) or use `Array.pnew` (see below).

```
A1.pmap.with_index(&f)
```

New This operation is a combination of *index* and *map*. It creates a new array of size m and initializes it using the block (anonymous function) f. It corresponds to `Array.new` in Ruby but is executed in parallel.

$$new(m, f) = map(index(m), f) = [f(0), \ldots, f(m-1)]$$

Similiar to *index*, this operation takes multiple arguments in a multidimensional case. This operation is exposed to Ruby programmers as `pnew`:

```
Array.pnew(m, &f)
```

Stencil (Convolution) This operation takes as arguments an input array A, an array of relative indices I (neighborhood) of size k, a block f, and an out-of-bounds value o. It creates an array of same size and dimensionality where every value i is initialized using f, passing the values in the neighborhood of $A[i]$ as arguments to f. Simple examples of stencil computations are image filtering kernels.

The following formula is used to calculate the value in the resulting array at position i. If all indices are within bounds (case 1), i.e., $0 \leq i + I[j] < m$ for all $0 \leq j < k$ (where m is the size of A), the value of the stencil computation is used. Otherwise (case 2), the fallback value o is used.

$$v(A, I, f, o, i) = \begin{cases} f(A[i + I[0]], \ldots, A[i + I[k-1]]), & \text{(case 1)} \\ o, & \text{(case 2)} \end{cases}$$

Using this helper function v, a stencil computation is defined as follows.

$$stencil(A, I, f, o) = [v(A, I, f, o, 0), \ldots, v(A, I, f, o, m-1)]$$

(multidimensional case omitted)

This operation is exposed to Ruby programmers as `pstencil`:

```
A.pstencil(I, o, &f)
```

Zip This operation does not perform a computation but groups values of two ore more arrays of same size and dimensionality.

$$zip(A_1, \ldots, A_n) \quad \text{(multidim. case omitted)} = \begin{bmatrix} [[A_1[0], \ldots, A_n[0]] \\ \vdots \\ [A_1[m-1], \ldots, A_n[m-1]]] \end{bmatrix}$$

The result of this operation is an array of arrays. This operation is exposed to Ruby programmers as `pzip`:

```
A1.pzip(A2, ..., An)
```

Reduce This operation takes as arguments an input array A and a block f, whose function must be associative. Every block application reduces two elements into a single one. The block is applied until only one element is left (regardless of the dimensions). This operation is similar to Ruby's reduce, but the return value is an array with one element instead of a scalar value.

$$reduce(A, f) = [f(\ldots f(f(A[0], A[1]), f(A[2], A[3]), \ldots)\ldots)]$$

(multidimensional case omitted)

There are no guarantees about the order in which element are combined, because reduction is done in parallel. This operation is exposed to Ruby programmers as `preduce`:

```
# Passing a block (function)
A.preduce(&f)
# Symbols are also possible as shortcuts
# E.g.: A.preduce do |a, b| a + b; end
A.preduce(:+)
```

3. Kernel Fusion

All array commands except for *index* and *array identity* have at least one input array command (`input` in Figure 2). E.g., the input for a *map* operation is the array that is being mapped over. During code generation, Ikra traverses the tree of such dependent (input) commands. Depending on the access pattern of dependent input, Ikra may merge multiple parallel operations into a single kernel.

Command	Input Access Pattern
combine	same location (for all inputs)
stencil	multiple (fixed pattern)
reduce	multiple
zip	same location (for all inputs)
(`with_index`)	same location

Figure 3: Input Access Patterns for Array Commands

Figure 3 lists access patterns for dependent computations for all array commands. "Same location" means that for computation of the element at position i only the element at the same position in dependent (input) command(s) is required. For example, to calculate element 12 in *map*, only element 12 from the input is required. "Multiple" means that an array command needs multiple elements from dependent command(s). For example, a stencil computation requires an entire neighborhood of values from the input.

Ikra can currently merge dependent computations if the access pattern is "same location". In that case, one thread can first compute the dependent operation and then directly proceed with the following computation without any synchronization. Figure 4 shows an example. The leftmost gray box corresponds to Lines 2–3. Those operations are fused because the input access pattern for *combine* is "same location"[3]. For the stencil computation, the first input cannot be fused because its access pattern is "multiple". The *index* input can be fused because input generated by `with_index` is always accessed as "same location". Future work will extend kernel fusion to access patterns used in stencil computations (Section 7).

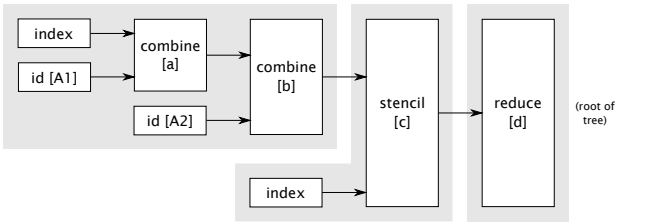
4. Examples

In this section, we present two examples to illustrate how Ikra is used in practice. The first example focuses on modular (reusable) parallel sections and the second example shows the usage of host sections.

[3] *id* is added implicitly when programmers use Ruby arrays.

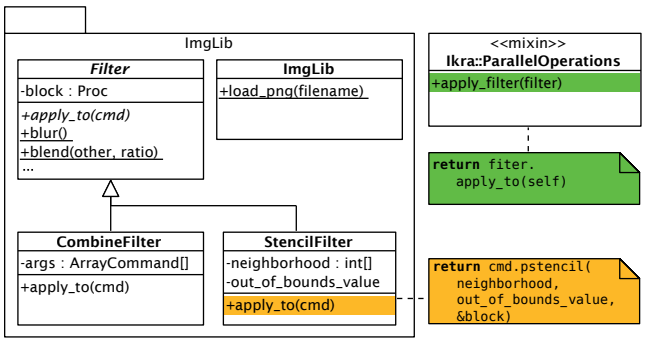
```ruby
A1 = [1, 2, 3]; A2 = [10, 20, 30]      # Ruby arrays
a = A1.pmap.with_index do |e, idx| ... end # combine
b = a.pcombine(A2) do |e1, e2| ... end     # combine
c = b.pstencil([-1, 0, 1], 0).             # stencil
   with_index do |values, idx| ... end
d = c.preduce do |r1, r2| ... end           # reduce
```

(a) Source Code



(b) Resulting Kernel Fusion. Gray boxes are kernels.

Figure 4: Example: Kernel Fusion



(a) Architecture of Image Manipulation Library

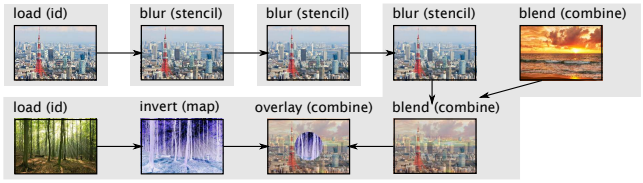
```ruby
module ImageLibrary::Filters
   def self.load_png(filename)
      image = read_png(filename)
      return image.pixels.to_command(
         dimensions: [image.height, image.width])
   end
   def self.blend(other, ratio)
      return CombineFilter.new(other) do |p1, p2|
         pixel_add(                  # Helper functions
            pixel_scale(p1, 1.0 - ratio),
            pixel_scale(p2, ratio))
      end
   end
   def self.blur
      return StencilFilter.new(
         neighborhood: STENCIL_2,
         out_of_bounds_value: 0) do |v|
            r = v[-1][-1][0] + ... + v[1][1][0]
            g = v[-1][-1][1] + ... + v[1][1][1]
            b = v[-1][-1][2] + ... + v[1][1][2]
            [r / 9, g / 9, b / 9]
      end
   end
end
```

(b) Example: Definition of Filters

Figure 5: Implementation of Image Manipulation Library

```ruby
require "image_library"

# Load Tokyo Tower image and blur it
tt = ImgLib.load_png("tokyo_tower.png")
for i in 0...3
   tt = tt.apply_filter(ImgLib::Filters.blur)
end

# Blend with sunset
sun = ImgLib.load_png("sunset.png")
combined = tt.apply_filter(
   ImgLib::Filters.blend(sun, 0.3))

# Load + overlay forest
forest = ImgLib.load_png("forest.png")
forest = forest.apply_filter(
   ImgLib::Filters.invert)
combined = combined.apply_filter(
   ImgLib::Filters.overlay(
      forest, ImgLib::Masks.circle(tt.height / 4)))

# Draw pixels on screen
ImgLib::Output.render(combined)
```

(a) Ruby Source Code



(b) Image Rendering. Gray boxes indicate generated GPU kernels.

Figure 6: Image Manipulation Library Usage Example

4.1 Image Manipulation Library

In this example, we design an image manipulation library which is based on Ikra. It provides methods for loading images from the file system and a number of filters and effects.

Figure 6 illustrates how to use the library and shows each step of the computation: First, a picture of the Tokyo Tower is loaded. Then, a blur filter is applied multiple times. Moreover, a picture of a sunset is loaded and both pictures are merged. Finally, a picture of a forest is loaded, inverted and overlaid with the previously merged picture. Notice how the code is modular with respect to composability, reusability, understandability: Image filters are provided by the library and can be arbitrarily combined. However, only when the final result is accessed (rendered) in the last line, Ikra generates a single GPU program with four kernels and executes it.

The filters are provided by the image manipulation library (Figure 5) and implemented using parallel map or stencil operations. The image manipulation library defines an extension method `apply_filter` for applying image filters using double dispatch.

4.2 Iterative Computation

Many scientific computations (e.g., numerical partial differential equations) exhibit an iterative structure where an array or matrix is updated for a constant number of times or until a convergence criteria is reached. The following source code snippet does not compute any meaningful result but illustrates how to write such computations in Ikra. The update loop is contained in a host section, a code section that is compiled to C++ and executed on the host, as

opposed to *parallel sections* which are executed on the device. The value of the last statement of a host section is the return value of the host section. Inside host sections, only *simple* Ruby code may be written: Everything that is allowed inside a parallel section plus parallel sections themselves. More advanced Ruby features (such as metaprogramming) are forbidden.

```
1  input = [10, 20, 30, 40, 50, 60]
2  result = Ikra.host_section do
3     arr = input.to_command(dimensions: [2, 3])
4
5     for i in 0...10
6        if arr.preduce(:+)[0] % 2 == 0
7           arr = arr.pmap do |i| i + 1; end   # map_A
8        else
9           arr = arr.pmap do |i| i + 2; end   # map_B
10       end
11
12       arr = arr.pmap do |i| i + 3; end      # map_C
13    end
14
15    arr
16 end
```

In the above example, one C++/CUDA program is generated. That program contains a C++ function for the host section and multiple CUDA kernels. Control flow statements inside host sections are executed symbolically as opposed to control flow statements outside of host sections, which are executed by the Ruby interpreter. Host sections are translated with a conservative kind of ahead of time compilation: In the above example, it is not clear if the parallel section in Line 12 will be executed together with the one in Line 7 (map_A + map_C) or the one in Line 9 (map_B + map_C), or both in different iterations. Therefore, Ikra generates both fused kernel variants and launches the appropriate one at runtime. The detailed code generation process for this example is described in Section 5.2.

5. Code Generation

5.1 Mapping Ruby Types to C++ Types

During type inference, Ikra analyzes which types an expression can have. If an expression is monomorphic (has a single type), a Ruby type is directly mapped to a corresponding type in the C++/CUDA source code. Figure 7 shows the mapping of Ruby data types to CUDA/C++ data types. Numeric values are currently represented by `int` and `float`; therefore, the range/precision of values is limited in Ikra. `nil` is represented by `int` value 0. Arrays are either represented by `array_t` (a pointer-size pair) or a generated struct type for arrays that appear in zip types (to allow efficient zipping of arrays of different type). Array commands will be discussed in the next section. All other objects are represented by object IDs generated by Ikra's object tracer [14].

Ruby Type	C++/CUDA Type
`Fixnum`	`int`
`Float`	`float`
`TrueClass`	`bool`
`FalseClass`	`bool`
`NilClass`	`int`
`Array`	`array_t`, generated struct type
`ArrayCommand`	`array_command_t *` *(only allowed in host sections)*
(other)	`object_id_t` (`int`)
(multiple types)	`union_t`

Figure 7: Mapping Ruby Types to C++ Types

```
union union_v_t
{
   int int_;
   float float_;
   bool bool_;
   void *pointer;
   array_t array;
} union_v_t;
```

```
struct union_t
{
   int class_id;
   union_v_t value;
}
```

Figure 8: Union Type Struct Definition

For polymorphic expressions (e.g., if `Fixnums` and `nil` are assigned to a variable), union type structs (Figure 8) are used [1]. Values are stored in `union_v_t` which can hold values (or pointers to values) for all C++ types in Figure 7. The class ID field contains a number which identifies the runtime type unambiguously. If a method is called on a polymorphic expression, Ikra generates a switch-case statement with all types that the expression can have at runtime. If a monomorphic value is assigned to a polymorphic lvalue, Ikra wraps the value in a union type struct. Arrays of union type structs are used to represent polymorphic arrays.

5.2 Symbolic Execution in Host Sections

Host sections are pieces of Ruby code that are entirely translated to C++ code. They may contain one or more parallel sections but no advanced language features like metaprogramming. The compilation process of host sections is identical to the one of parallel sections, but there are additional steps to handle parallel sections within them. Since no code generation can be done once a host section (C++ code) is executing, fused kernels must be generated up front. Ikra statically analyzes all code paths with parallel sections through the host section and generates a number of fused kernels, even some that might never be used at runtime. At runtime, Ikra keeps track of which parallel sections were executed symbolically and eventually launches a fused kernel, which may contain multiple parallel sections, when the result is accessed.

Kernel Fusion via Type Inference Within host sections, there are additional type inference rules to handle parallel sections. Ikra performs kernel fusion through type inference: The type of a parallel section (i.e., a method call AST node invoking a method defined in `ParallelOperations`) is the array command that it evaluates to when evaluating it symbolically. To that end, Ikra interprets such method call nodes in the Ruby interpreter. For example:

```
a = 10
# type(a) = int

b = Array.pnew(a) do ... end
# Eval Ruby: Array.pnew(CodeRef.new(:a)) do ... end
# type(b) := ArrayCombineCommand instance

c = b.pmap do ... end
# Eval Ruby: type(b).pmap do ... end
# type(c) = (different) ArrayCombineCommand instance
```

For performance reasons, the values of arguments of parallel operations (except for input or size arguments) must be known during symbolic execution (symbolic execution-time constants), so that they are constant in the GPU program [10]. For example, the neighborhood of a stencil computation inside a host section must be constant and cannot be generated inside the host section. However, all arguments of a *combine* operation may be variable.

Compilation Process The result of a host section in the Ruby interpreter is an array command (`HostSectionArrayCommand`). The following list gives an overview of the single steps in symbolic execution and code generation.

1. Retrieve Ruby source code of host section
2. Generate AST (abstract syntax tree) from source code
3. Convert AST to SSA (static single assignment) form
4. Insert `to_a` method call on last expression
5. Perform type inference
6. Generate C++ source code for host section and CUDA source code for all array commands on which `to_a` or `[]` is called

The first two steps are identical to symbolic execution of parallel sections. The SSA form simplifies type inference: If a variable is written a second time with a value of different type, a new C++ variable is allocated and a union type can sometimes be avoided. The return value (last expression) of a host section must be an array command or an array. In the former case, to ensure that an array command is executed, Ikra wraps the last expression in a method call node with method name `to_a`. That method does not have any effect for arrays but triggers array command execution. Type inference is currently implemented with multiple passes; i.e., Ikra extends the types of all expressions during every pass until a fixpoint is reached[4].

After type inference, Ikra generates C++ source code for the host section. If an expression has an array command type, Ikra uses a pointer to an `array_command_t` struct which has (among other things) a pointer to the cached result of the array command (if it was accessed before). Essentially, an `array_command_t` instance holds all information that an `ArrayCommand` instance in Ruby holds, except for information that is known statically (e.g., the out of bounds value of stencil computations is a numeric literal in the kernel source code). Whenever an array command access is detected (e.g., calling `to_a` on an array command), Ikra generates kernel source code for the array command (receiver type!) and a kernel invocation snippet which checks if a cached result is available and otherwise transfers data (if necessary) and launches the kernel. Since array commands may have dependent input commands, generated kernels may consist of multiple fused parallel sections.

Example As an example, consider the following Ikra source code which is already in SSA form and has a `to_a` method call at the end.

```
1  result = Ikra.host_section do
2     arr1 = input.to_command(dimensions: [2, 3])
3
4     for i in 0...10
5        arr2 = φ(arr1, arr6)
6
7        if arr2.preduce(:+)[0] % 2 == 0
8           arr3 = arr2.pmap do |i| i+1; end   # map_A
9        else
10          arr4 = arr2.pmap do |i| i+2; end   # map_B
11       end
12       arr5 = φ(arr3, arr4)
13
14       arr6 = arr5.pmap do |i| i+3; end      # map_C
15    end
16
17    arr7 = φ(arr1, arr6)
18    arr7.to_a
19 end
```

[4] There are better techniques for type inference using constraint solving, but this is an implementation detail.

In the following, we take a look at the inferred types of all $\texttt{arr}_i$ variables. $\texttt{arr}_1$ is an identity command for Ruby array `input` and $\texttt{arr}_2$ cannot be fully inferred yet because $\texttt{arr}_6$ is still unknown.

$$\texttt{arr}_1 = id[\texttt{input}]$$
$$\texttt{arr}_2 = \{\texttt{arr}_1, \texttt{arr}_6\} = \{id[\texttt{input}], \texttt{arr}_6\}$$

Next, we infer the types for the first two *map* operations by evaluating the `pmap` method calls in the Ruby interpreter with each type in the union type of $\texttt{arr}_2$ as the receiver. Different subscripts of *map* operations indicate that the operations are different array command objects after symbolic evaluation (because they have different parallel sections) and, therefore, represent different types.

$$\begin{aligned}\texttt{arr}_3 &= map_A(\texttt{arr}_2) = \{map_A(id[\texttt{input}]), map_A(\texttt{arr}_6)\}\\ \texttt{arr}_4 &= map_B(\texttt{arr}_2) = \{map_B(id[\texttt{input}]), map_B(\texttt{arr}_6)\}\\ \texttt{arr}_5 &= \{\texttt{arr}_3, \texttt{arr}_4\}\\ &= \{map_A(id[\texttt{input}]), map_A(\texttt{arr}_6),\\ &\quad map_B(id[\texttt{input}]), map_B(\texttt{arr}_6)\}\end{aligned}$$

Next, we infer the type of the last *map* operation.

$$\begin{aligned}\texttt{arr}_6 &= map_C(\texttt{arr}_5)\\ &= \{map_C(map_A(id[\texttt{input}])), map_C(map_A(\texttt{arr}_6)),\\ &\quad map_C(map_B(id[\texttt{input}])), map_C(map_B(\texttt{arr}_6))\}\end{aligned}$$

As can be seen from the definitions above, the type of $\texttt{arr}_6$ is circular. If we try to fully expand its definition, it will have an infinite number of elements. This is because our type inference mechanism effectively analyzes all dataflow code paths through the host section (but cannot "count"): The union type of $\texttt{arr}_7$ will have one type for every code path. Host sections with loops have an infinite number of paths, because the type inference engine is not aware of the number of iterations. Moreover, if `while` loops are used, the number of iterations cannot be determined statically in general.

Once a cycle like this one is detected, Ikra breaks it by inserting a `to_a` method call, which will launch the kernel and return its result as an array. Consequently, such a method call will stop the kernel fusion process. Where exactly the cycle is broken is an implementation detail. Ikra currently inserts the method call in Line 14, but it could also be inserted in Lines 5 or 12.

```
14 arr6 = arr5.to_a.pmap do |i| i + 3; end     # map_C
```

We can now complete the type inference process and fill in the $\texttt{arr}_2$ placeholders in the other definitions.

$$\begin{aligned}\texttt{arr}_6 &= map_C(id[\texttt{arr}_5])\\ \texttt{arr}_2 &= \{id[\texttt{input}], map_C(id[\texttt{arr}_5])\}\\ \texttt{arr}_5 &= \{\texttt{arr}_3, \texttt{arr}_4\}\\ &= \{map_A(id[\texttt{input}]), map_A(map_C(id[\texttt{arr}_5])),\\ &\quad map_B(id[\texttt{input}]), map_B(map_C(id[\texttt{arr}_5]))\}\\ \texttt{arr}_7 &= \{\texttt{arr}_1, \texttt{arr}_6\}\\ &= \{id[\texttt{input}], map_C(id[\texttt{arr}_5])\}\end{aligned}$$

For code generation, only $\texttt{arr}_2$, $\texttt{arr}_5$ and $\texttt{arr}_7$ are of interest, because their result is accessed. Ikra generates kernels and invocations for them: The type of the variable (or class ID field if polymorphic) determines the kernel to be launched. In total, Ikra generates the following kernels in this example (some might never be launched).

(5.1) $map_A(id[\texttt{input}])$
(5.2) $map_A(map_C(id[\texttt{arr}_5]))$
(5.3) $map_B(id[\texttt{input}])$
(5.4) $map_B(map_C(id[\texttt{arr}_5]))$
(7.1) $id[\texttt{input}]$
(7.2) $map_C(id[\texttt{arr}_5])$
(r.1) $reduce(id[\texttt{input}])$
(r.2) $reduce(map_C(id[\texttt{arr}_5]))$

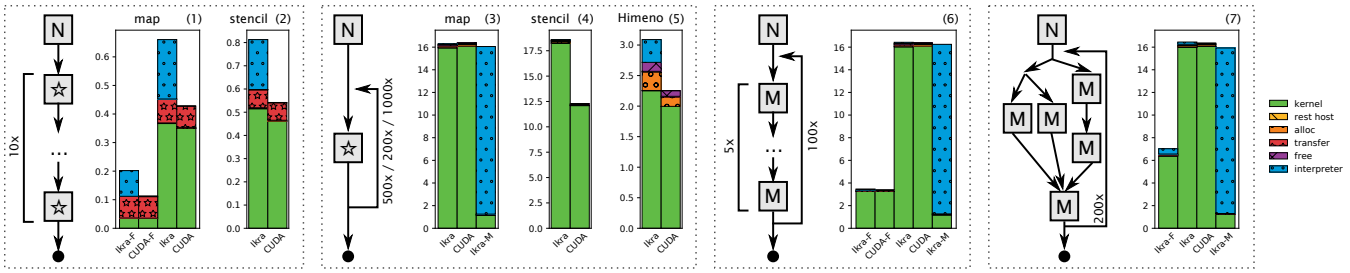


Figure 9: Microbenchmark Runtime in Seconds. *Ikra-F* is Ikra with all code in a single host section and kernel fusion. *Ikra* is without kernel fusion. *Ikra-M* is a lower bound where all code is in a single kernel (even among iterations). *CUDA-F* and *CUDA* are hand-written baseline implementations with/without (manual) kernel fusion. Compilation time (not shown here) is around 2 seconds for Ikra-generated code.

All polymorphic variables (all variables except for $\texttt{arr}_1$) will have type `union_t` in the generated C++ code. The class ID field is used to determine which kernel should be launched. For example, one of the last two kernels should be launched in Line 18 depending on whether there was at least one loop iteration or not. This information is implicitly encoded in the class ID field. In the generated code, Lines 2, 5, 8, 10, 12 and 17 do not launch a kernel but merely update their respective variable with a new `array_command_t` object, possibly wrapped inside a union type struct containing the class ID for the command[5].

6. Benchmarks

This section shows a number of microbenchmarks[6] (i.e., code inside parallel operations is simple) of the current Ikra implementation in various configurations. Benchmarks were run on a computer with an Intel Core i7-6820HQ CPU (2.70 GHz), 32 GB RAM, an Nvidia GeForce 940MX GPU, Ubuntu 16.04.1 (kernel version `4.4.0-43-generic`), Ruby 2.3.1 and the CUDA Toolkit V8.0.44. Program 5 is a 3D stencil computation on a matrix of size $129 \times 65 \times 65$. All other programs operate on matrices of size 60,000,000 (228 MB) with a one-dimensional CUDA block size of 1024.

The benchmarks show the performance speedup due to kernel fusion and how generated Ikra code performs in comparison to hand-written CUDA code. We compare 7 different programs with various compilation strategies. For every program, we show the kernel program structure. The square boxes indicate kernels and the arrows indicate data flow. The letter *M* indicates a *map* operation, the letter *S* a *stencil* operation and the letter *N* a *new* operation.

Program 1, Program 6 and Program 7 show the benefit of kernel fusion. In the first case, a single kernel is launched for all 11 parallel sections, giving a 10x speedup compared to the version without kernel fusion. In the second case, 101 kernels are launched for 500 parallel sections. All 5 kernels within the loop are fused together, giving a 5x speedup compared to the version without kernel fusion. Hypothetically, if all 501 *map*/*new* operations were fused together, another 2x speedup would be possible. However, in the general case, the number of loop iterations is unknown (e.g., for while loops). Moreover, the resulting kernel code becomes large; increasing the number of iterations even results in a `nvcc` compilation error. As can be seen from the *interpreter* time, Ikra also spends a very long time in the Ruby interpreter if a tree of 501 array commands is analyzed and fused together. This shows that there is still potential for optimization of the Ruby part of Ikra.

Program 7 shows the benefit of kernel fusion in a program with more complex data flow, giving a speedup of 2x. This program generates source code with union types to keep track of which kernel to launch at the end of an iteration. This leads to additional runtime overhead. However, this overhead (*rest host*) is much smaller than the kernel runtime and could not even be measured with confidence in our experiments. The reason that Program 7 does not benefit from kernel fusion in the same way as Program 6 is that the number of parallel sections inside the loop is smaller and the number of loop iterations is larger (200 vs. 100).

Programs 3–5 consist of a loop with a single *map* or *stencil* operation. In such cases, Ikra cannot perform kernel fusion inside the loop, which is why we only report the performance for *Ikra*. *Himeno* is a benchmark with a memory-bound stencil computation. It has a high *alloc* time because Ikra (and the CUDA baseline) do currently not reuse memory but allocate a new piece of memory every time an array is updated within the loop. When reusing the same memory location, the performance can be increased due to a lower allocation time and more efficient memory access (caching).

All benchmarks have a low *transfer* time, i.e., time spent for transferring data between the device and the host. This is because data is only transferred to the host when the result of a parallel section is accessed. The loops in all benchmarks are *for* loops with a fixed number of iterations. Only after the last iteration, the result is accessed and transferred back to the host. In a more realistic case, where the control flow (e.g., number of iterations) depends on the data, more data transfer will occur.

When comparing Ikra's performance with hand-written CUDA code, we can see that the kernel runtimes are almost identical for *map* operations. The generated code of *stencil* operations is not yet fully optimized (or fused). Ikra spends additional time in the Ruby interpreter for performing type inference and generating Ruby code, and very little time in host sections (allocating/comparing union types/array commands, loop overhead, etc.).

Number of Generated Kernels Due to the kernel fusion process described in Section 5.2, Ikra generates one kernel per data flow path (excluding loops). This can lead to a combinatorial explosion of the number of generated kernels. Based on an analysis of the kernel structure of a large number of parallel programs by J. Shen et al. [13], we believe that the number of kernels remains manageable in real applications. Their work showed that the kernel structure of all analyzed programs is similar to the ones in our benchmarks[7] and never more complex than the structure in Program 7.

7. Future Work

Kernel Fusion of Stencil Operations Ikra can currently only fuse operations whose input pattern is "same location". There are plans to extend kernel fusion to certain stencil computations that

[5] Recall that polymorphic method calls are translated to switch-case statements. Every case updates the respective variable with a different class ID.

[6] `https://github.com/prg-titech/ikra-ruby`, branch `array17`

[7] Ikra's programming style could increase the number of parallel sections.

exhibit a *simple* neighborhood. Kernels fusion can be done either with redundant computation or with synchronization. Consider, for example, the following two stencil operations.

$$A_1 = stencil(A_0, [-1, 0], f, 10)$$
$$A_2 = stencil(A_1, [-1, 0], g, 10)$$

The resulting arrays after each iteration are defined as follows.

$$A_1 = \begin{bmatrix} 10 \\ f(A_0[0], A_0[1]) \\ f(A_0[1], A_0[2]) \\ \cdots \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 10 \\ g(10, f(A_0[0], A_0[1])) \\ g(f(A_0[0], A_0[1]), f(A_0[1], A_0[2])) \\ \cdots \end{bmatrix}$$

The definition of A_2 represents the computation for the fused stencil operation. Most terms using the function f are computed twice (redundantly). Alternatively, Ikra could split A_0 in multiple arrays, assign every subarray to a CUDA block and store intermediate results in shared memory. Inter-block synchronization or redundant computation is then only necessary at the subarray borders (*ghost region* [8]). This technique works well only if the neighborhood is simple (i.e., the ghost region is small).

Reusing Memory Many programs that update a vector or matrix iteratively only need access to the data from the previous computation. For performance reasons (e.g., caching), CUDA programmers allocate one (for *combine* operations) or two (for *stencil* operations) arrays only and keep writing to these arrays. However, all operations in Ikra (and Ruby) create new arrays. Moreover, Ikra does not have a garbage collector, so the memory is released after the execution of the host section or if the programmer frees memory explicitly in the Ruby code. To decide whether it is safe to reuse a previously allocated array, Ikra must do an escape analysis to ensure that overwritten data is not read at a later point of time. Such advanced memory management issues are subject to future work.

8. Related Work

Kernel fusion is an optimization that is supported by many other GPGPU frameworks and languages, but the focus is on different aspects. For example, Harlan [7] supports nested kernels, Futhark [6] has support for nested parallelism and a powerful fusion engine for map/reduce combinations, and Kernel Weaver focuses on database queries [17]. Furthermore, all of these tools focus on statically-typed programming languages, making translation within a kernel easier compared to Ikra because no union types are required and making kernel fusion itself easier because it is known ahead of execution time which kernels are executed together.

Fumero et al. designed a GPU extension for the R programming language, a dynamically-typed language [4]. Their implementation is built on top of the Truffle AST interpreter framework [18] and the Graal JIT compiler. They use partial evaluation to generate optimized OpenCL code for *hot* code sections. In contrast to Ikra, the resulting OpenCL code can handle only monomorphic types, whereas Ikra generates a single CUDA program with union types that can handle all types which could theoretically show up during runtime. Consequently, their generated OpenCL code is more efficient, but requires recompilation if the runtime types are changing.

9. Summary

We presented the design and implementation of Ikra, a Ruby library for data-parallel computations. Ikra allows programmers to write modular code with respect to reusability and composability of parallel sections. Parallel sections that are used together are fused into a single kernel if they use only the values generated by the previous section at the same location. Host sections separate code regions that extensively use dynamic language features from computations and allow Ikra to compile entire loops to C++ code. If inside of a host section, parallel operations are fused during a static analysis using type inference. Future work will extend kernel fusion to stencil computations and focus on memory management features.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, April 1991.

[2] M. M.T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore GPUs. DAMP '11, pages 3–14. ACM, 2011.

[3] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing*, 71(10):3934–3957, 2015.

[4] J. Fumero, M. Steuwer, L. Stadler, and C. Dubach. Just-in-time GPU compilation for interpreted languages with partial evaluation. VEE '17, pages 60–73. ACM, 2017.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] T. Henriksen, K. F. Larsen, and C. E. Oancea. Design and GPGPU performance of Futhark's redomap construct. ARRAY 2016, pages 17–24. ACM, 2016.

[7] E. Holk, R. Newton, J. Siek, and A. Lumsdaine. Region-based memory management for GPU programming languages: Enabling rich data structures on a spartan host. OOPSLA '14, pages 141–155. ACM.

[8] F. B. Kjolstad and M. Snir. Ghost cell pattern. ParaPLoP '10. ACM.

[9] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, O. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.*, 38(3):157–174, March 2012.

[10] A. S. D. Lee and T. S. Abdelrahman. Launch-time optimization of OpenCL GPU kernels. GPGPU-10, pages 32–41. ACM, 2017.

[11] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988.

[12] S. Sato and H. Iwasaki. *A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming*, pages 79–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[13] J. Shen, A. L. Varbanescu, X. Martorell, and H. Sips. A study of application kernel structure for data parallel applications. Technical report, Delft University of Technology, 2015.

[14] M. Springer and H. Masuhara. Object support in an array-based GPGPU extension for Ruby. ARRAY 2016, pages 25–31. ACM, 2016.

[15] M. Viñas, Z. Bozkus, and B. B. Fraguela. Exploiting heterogeneous parallelism with the heterogeneous programming library. *J. Parallel Distrib. Comput.*, 73(12):1627–1638, December 2013.

[16] M. Wahib and N. Maruyama. Scalable kernel fusion for memory-bound GPU applications. SC '14, pages 191–202. IEEE Press, 2014.

[17] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. MICRO-45, pages 107–118. IEEE Computer Society, 2012.

[18] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. Onward! 2013, pages 187–204. ACM, 2013.

[19] Y. Yan, M. Grossman, and V. Sarkar. Jcuda: A programmer-friendly interface for accelerating Java programs with CUDA. Euro-Par '09, pages 887–899. Springer-Verlag, 2009.