東京工業大学
Tokyo Institute of Technology

Modular Array-based GPU Computing in a Dynamically-typed Language

ARRAY 2017

Matthias Springer, Peter Wauligmann, Hidehiko Masuhara

Tokyo Institute of Technology

Overview

東京工業大学
Tokyo Institute of Technology

Introduction

- *Ikra:* Ruby Ext. for Array-based GPU Computing
- **CUDA**/C++ Code Generator
- Supports Object-oriented Programming
- Encourages a **Modular Programming** Style
- Employs various **Performance Optimizations**

Example

東京工業大学
Tokyo Institute of Technology

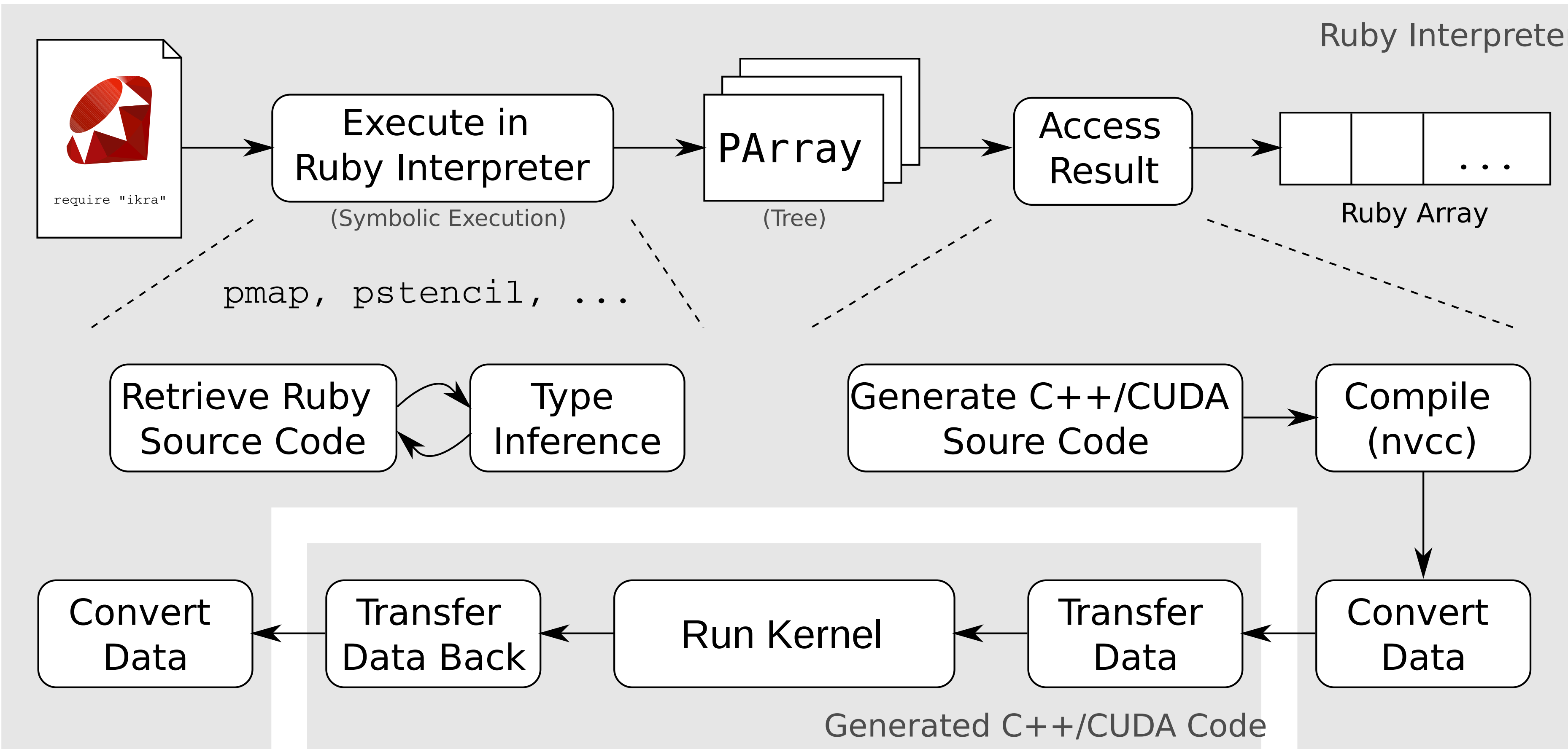
```
require “ikra”

SIZE = 100

a = PArray.new(SIZE) do rand() end

b = a.map do |i| i + 1 end

puts b[0]
```

Generate parallel array

Operation on parallel array

Lazy execution

Overview: Compilation Process



Programming Style

- Integration of **Dynamic Language Features**: GPU programming in dynamic Ruby programs
  - Restricted set of types/operations in parallel sections (incl. dynamic typing)
  - All Ruby features (incl. ext. libraries, metaprogramming) allowed in other code
    → Ahead-of-time translation not feasible
- **Modularity**: Compose parallel program of small, reusable parallel sections/kernels

Overview

東京工業大学
Tokyo Institute of Technology

Parallel Operations

東京工業大学
Tokyo Institute of Technology

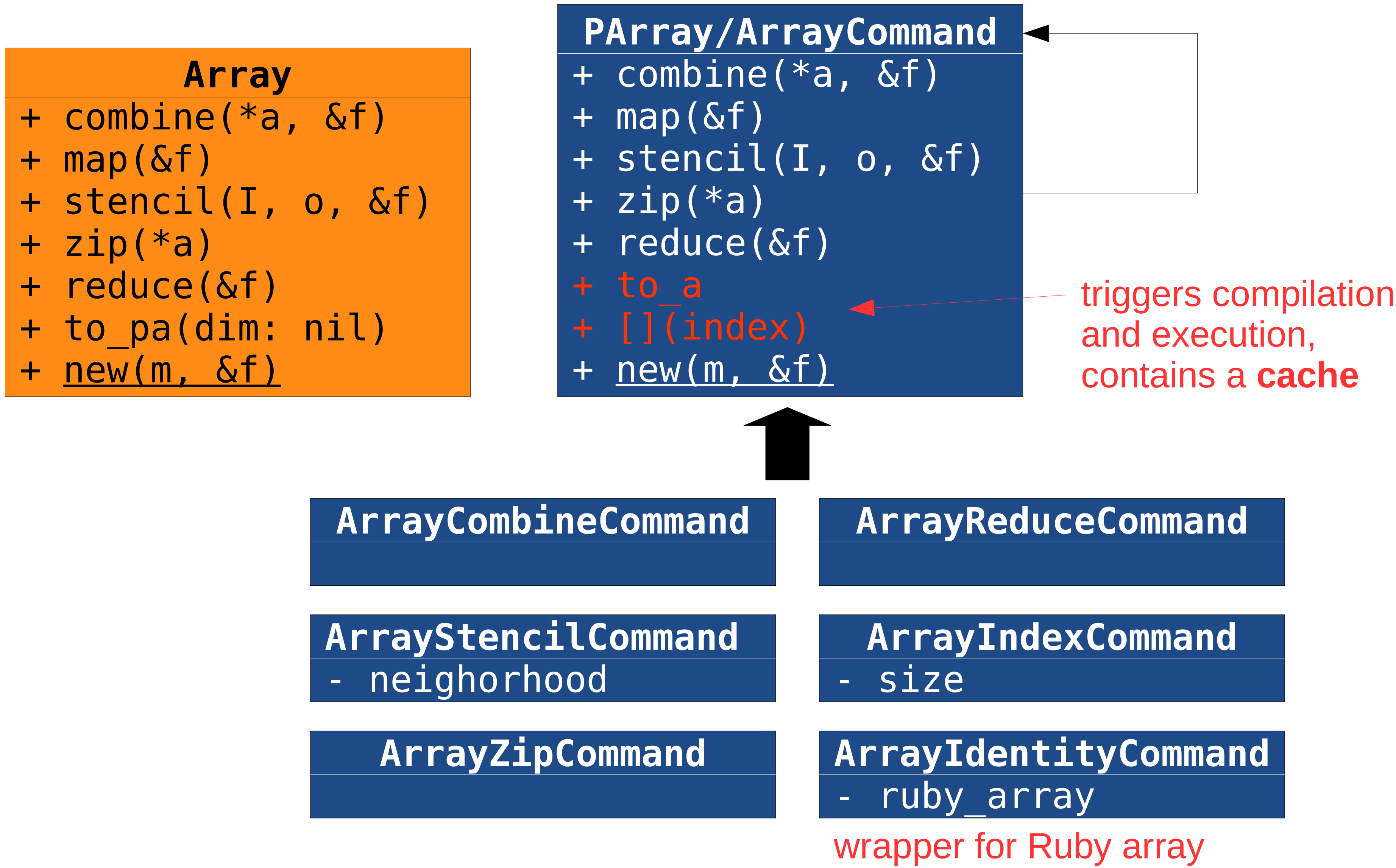
- `A`$_1$`.combine(A`$_2$`, ..., A`$_n$`, &f)`
  where `f` is `A`$_1$ `x ... x A`$_n$ → `B`
- `A.map(&f) = A.combine(&f)`
- `index(m) = [0, ..., m – 1]`
- `PArray.new(m, &f) = index(m).map(&f)`
- `A.stencil(I, o, &f)`
- `A`$_1$`.zip(A`$_2$`, ..., A`$_n$`) = [[A`$_1$`[0], ..., A`$_n$`[0]], ...]`
- `A.reduce(&f)`
- `A.select, A.prefix_sum, A.sort(&f), A.flatten, A.uniq`

Integration in Ruby

- Two kinds of arrays:
  Ruby array and Parallel (Ikra) Array
- Can be converted into each other:
  `Array.to_pa(dimensions: nil)`
  `PArray.to_a`
- Easy to switch between parallel/seq. versions

Only used in combination with `.with_index`

Integration in Ruby

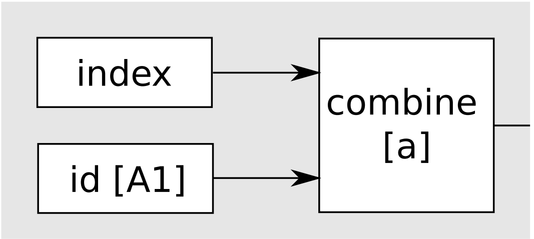


Example

```
A1 = [1, 2, 3]; A2 = [10, 20, 30]
```

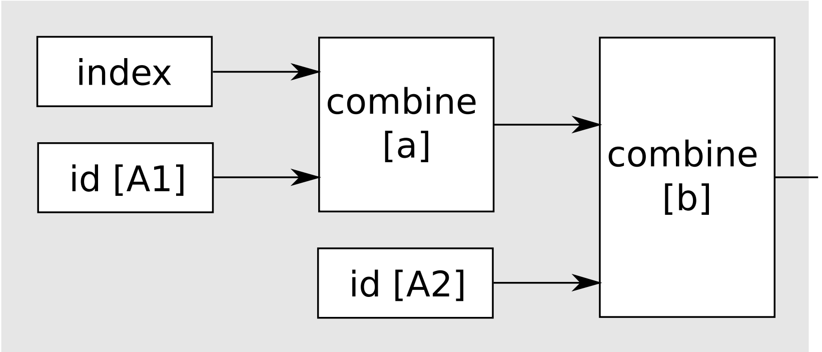
Example

```
A1 = [1, 2, 3]; A2 = [10, 20, 30]
a = A1.to_pa.map.with_index do |e, idx| ... end
```



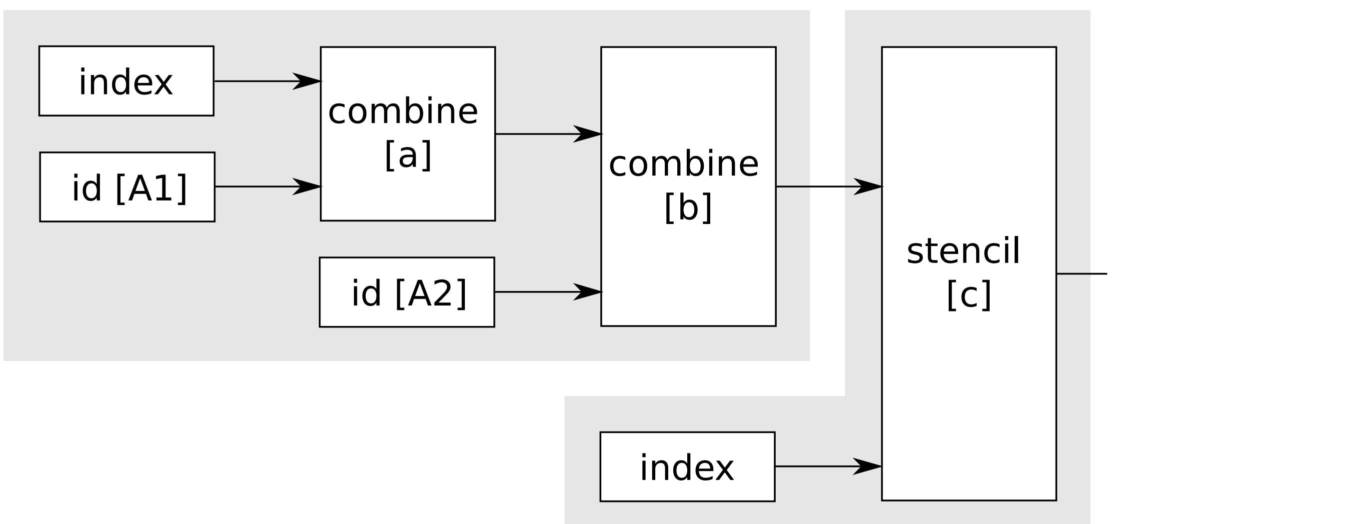
Example

```
A1 = [1, 2, 3]; A2 = [10, 20, 30]
a = A1.to_pa.map.with_index do |e, idx| ... end
b = a.combine(A2) do |e1, e2| ... end
```



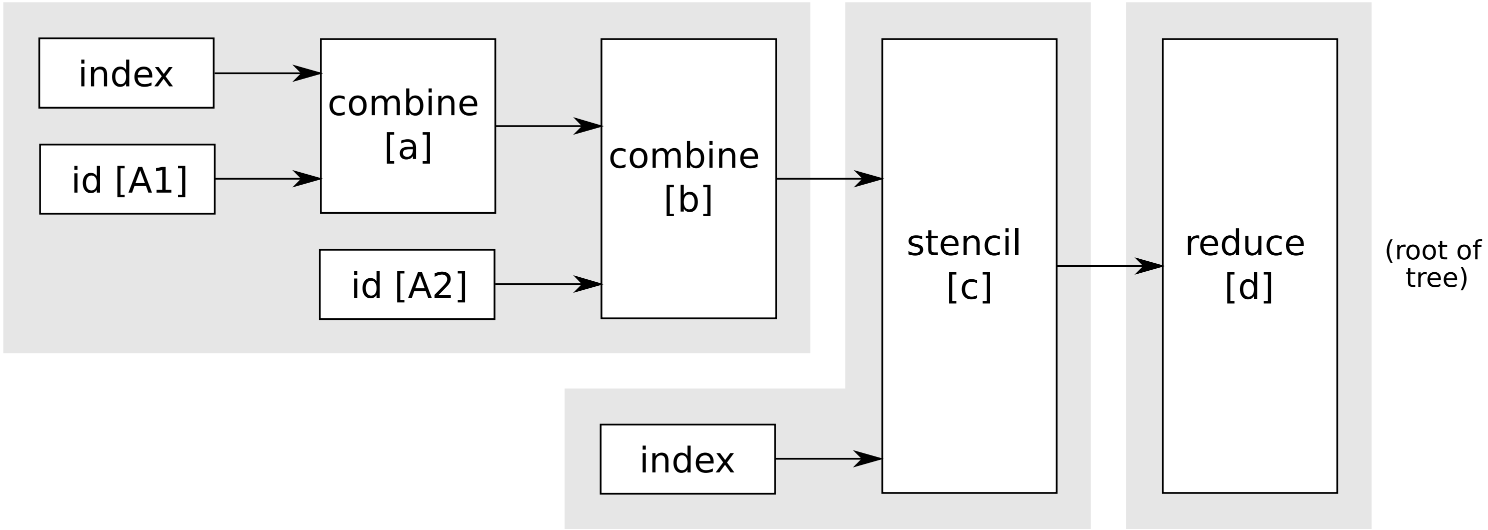
Example

```
A1 = [1, 2, 3]; A2 = [10, 20, 30]
a = A1.to_pa.map.with_index do |e, idx| ... end
b = a.combine(A2) do |e1, e2| ... end
c = b.stencil([-1, 0, 1], 0).
        with_index do |values, idx| ... end
```



Example

```
A1 = [1, 2, 3]; A2 = [10, 20, 30]

a = A1.to_pa.map.with_index do |e, idx| ... end

b = a.combine(A2) do |e1, e2| ... end

c = b.stencil([-1, 0, 1], 0).
        with_index do |values, idx| ... end

d = c.reduce do |r1, r2| ... end
```

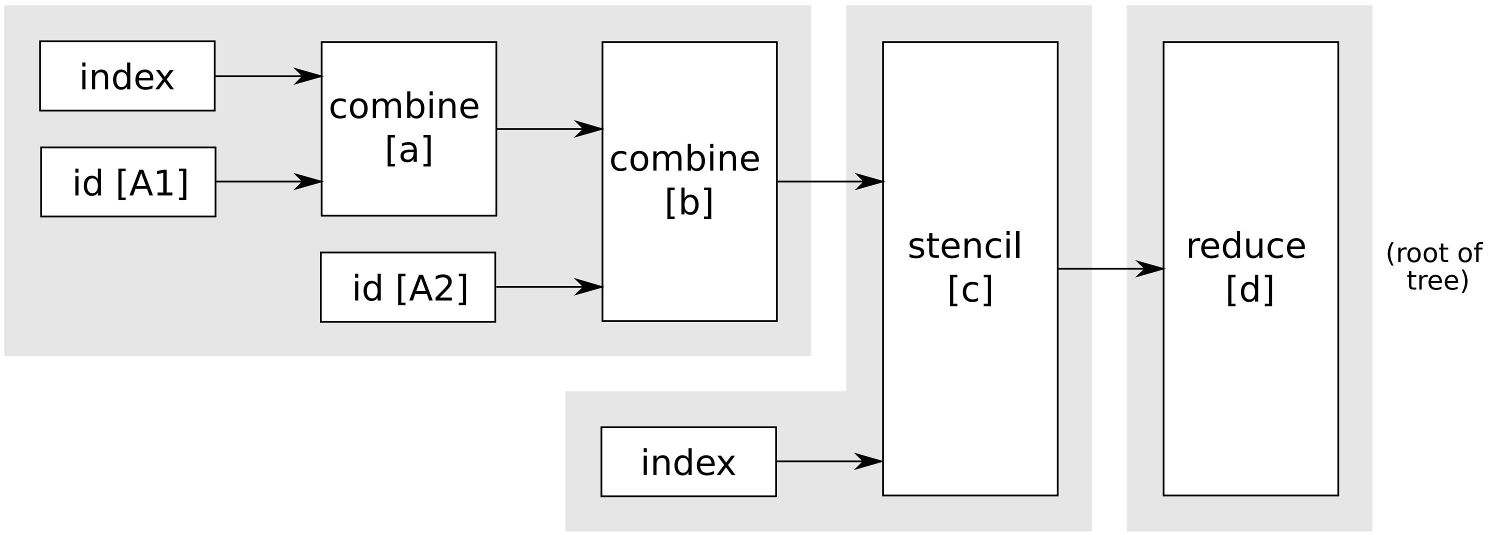


Kernel Fusion

Command	Input Access Pattern
combine	same location
stencil	multiple (fixed pattern)
reduce	multiple
zip	same location
with_index	(no input)
identity	(no input)

Optimization: Input with "same location" is combined (fused) into same kernel

Fusion possible (*temporal blocking* or *redundant computation*), but currently not implemented



Overview

東京工業大学
Tokyo Institute of Technology

Modular Programming

東京工業大学
Tokyo Institute of Technology

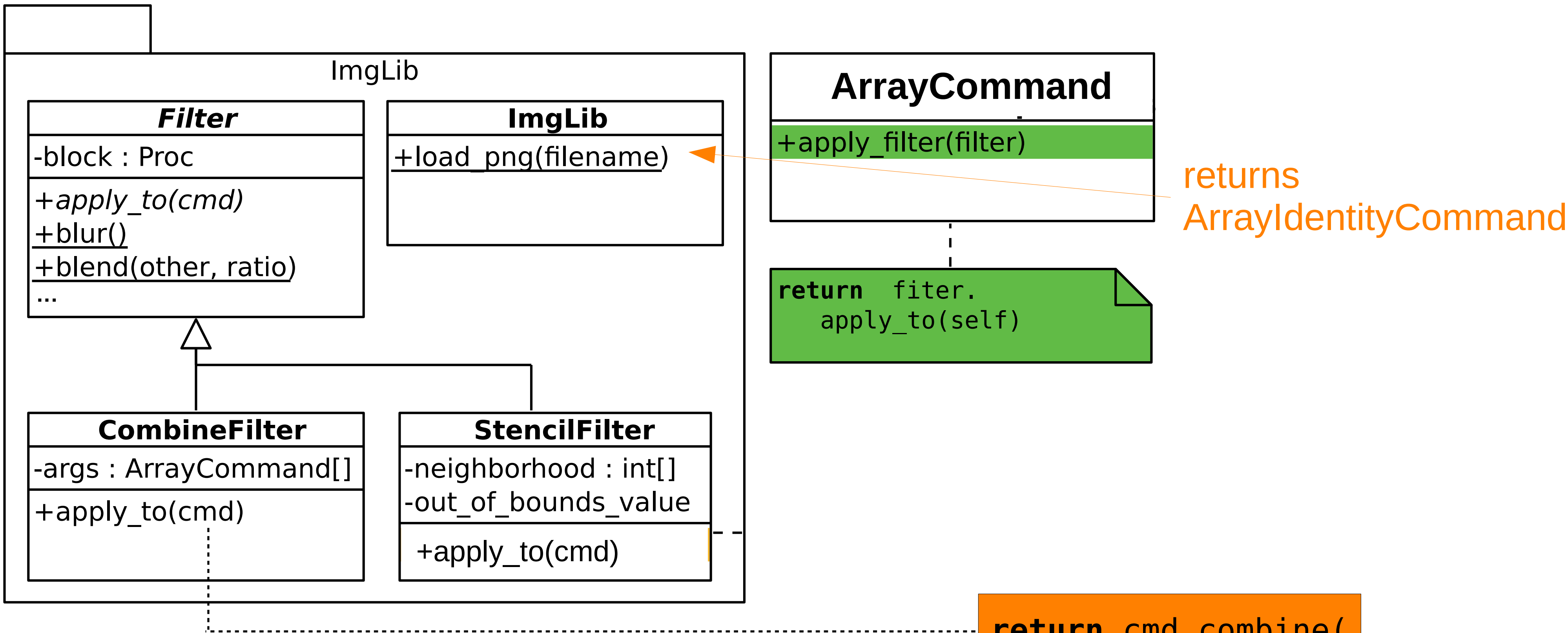
- **Modularity:** Understandability, reusability, composability
- Write multiple small parallel sections instead of a single big one, e.g.:
  - Matrix Multiplication
  - BFS Graph Traversal
  - Image Manipulation Library

```
img = ImgLib.load_png(“file.png”)
img2 = ImgLib.load_png(“file2.png”)

result = img
    .blur
    .blur
    .blur
    .blend(img2, 0.75)
```

Example: Image Manipulation Library

- Ruby library
- Load, render (show) images (2D int array)
- Filters
  - I_1`.blend(`I_2`, ratio)`
  - `I.invert`
  - I_1`.overlay(`I_2`, mask)`
  - `I.blur`
  - `I.sharpen`

Example: Image Manipulation Library

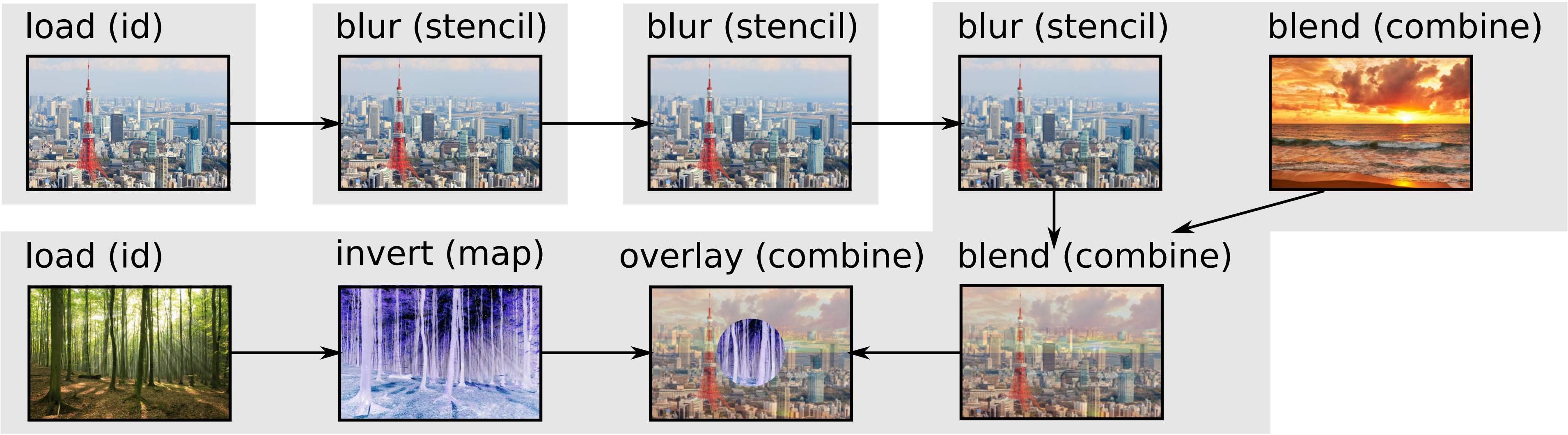



```
return cmd.combine(
   *args,
   &block)
```

```
def self.blend(other, ratio)
    return CombineFilter.new(other) do |p1, p2|
        pixel_add(
            pixel_scale(p1, 1.0 - ratio),
            pixel_scale(p2, ratio))
    end
end
```

Example: Image Manipulation Library

```
require "image_library"

tt = ImgLib.load_png("tokyo_tower.png")
for i in 0...3
    tt = tt.apply_filter(ImgLib::Filters.blur)
end

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

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)))

ImgLib::Output.render(combined)
```



Overview

1. Introduction
2. Parallel Operations
3. Modular Programming
4. **Iterative Computations**
5. Benchmarks
6. Summary

Iterative Computations

```
arr = [ ... ].to_pa(...)
while (arr.reduce(:+)[0] < 100)
    arr = arr.stencil([[-1, 0, 1], 0) do |v|
        3 * v[0] – v[-1] – v[1]
    end
end
```

executed on the GPU/CUDA

while loop executed in Ruby interpreter

- Overhead:
  - FFI Call Overhead (Switching between Ruby and C++)
  - Data format conversion for objects (SoA ↔ AoS)
- *Our solution:* Translate while loop to C++

Host Sections

```
arr = [ ... ].to_pa(...)
Ikra.host_section do
    while (arr.reduce(:+)[0] < 100)
        arr = arr.stencil([[-1, 0, 1], 0) do |v|
            3 * v[0] – v[-1] – v[1]
        end
    end
    arr.to_a
end
```

host section, executed in C++

- *Host section:* Translated to C++, invoked from Ruby
- *Parallel section:* Translated to CUDA, invoked from host section
- *Challenge:* Kernel fusion inside host sections

Host Section: Example

東京工業大学
Tokyo Institute of Technology

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr = input.to_pa

    for i in 0...10
        if arr.reduce(:+)[0] % 2 == 0
            arr = arr.map do |i| i + 1; end
        else
            arr = arr.map do |i| i + 2; end
        end

        arr = arr.map do |i| i + 3; end
    end

    arr.to_a
end
```

Challenge: Kernel fusion depends on runtime branches
1. Generate all fused kernels up front
2. Execute host section in C++, record all branches taken
3. Run specialized kernel corresponding to control flow path

Host Sections

東京工業大学
Tokyo Institute of Technology

- Generate all possible combination of fused kernels up front (before execution).
  - **Fusion by Type Inference**: The type of a parallel section (e.g., `map` method call) is the array command it evaluates to in the Ruby interpreter.
- Instead of executing kernels directly, remember (**trace**) which kernels an array command-typed expr. consists of.
- Execute array commands on access (**lazily**).

Host Section: Example

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr1 = input.to_pa

    for i in 0...10
        arr2 = φ(arr1, arr6)

        if arr2.reduce(:+)[0] % 2 == 0
            arr3 = arr2.map do |i| i + 1; end      α
        else
            arr4 = arr2.map do |i| i + 2; end      β
        end

        arr5 = φ(arr3, arr4)
        arr6 = arr5.map do |i| i + 3; end          γ
    end

    arr7 = φ(arr1, arr6)
    arr7.to_a
end
```

Host Section: Example

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr1 = input.to_pa

    for i in 0...10
        arr2 = φ(arr1, arr6)

        if arr2.reduce(:+)[0] % 2 == 0
            arr3 = arr2.map do |i| i + 1; end      α
        else
            arr4 = arr2.map do |i| i + 2; end      β
        end

        arr5 = φ(arr3, arr4)
        arr6 = arr5.map do |i| i + 3; end          γ
    end

    arr7 = φ(arr1, arr6)
    arr7.to_a
end
```

```
arr1 = I[input]
```

Host Section: Example

東京工業大学
Tokyo Institute of Technology

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr1 = input.to_pa

    for i in 0...10
        arr2 = φ(arr1, arr6)

        if arr2.reduce(:+)[0] % 2 == 0
            arr3 = arr2.map do |i| i + 1; end      α
        else
            arr4 = arr2.map do |i| i + 2; end      β
        end

        arr5 = φ(arr3, arr4)
        arr6 = arr5.map do |i| i + 3; end          γ
    end

    arr7 = φ(arr1, arr6)
    arr7.to_a
end
```

```
arr1 = I[input]

arr2 = {I[input], arr6}
```

Host Section: Example

東京工業大学
Tokyo Institute of Technology

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr_1 = input.to_pa

    for i in 0...10
        arr_2 = φ(arr1, arr6)

        if arr_2.reduce(:+)[0] % 2 == 0
            arr_3 = arr_2.map do |i| i + 1; end      α
        else
            arr_4 = arr_2.map do |i| i + 2; end      β
        end

        arr_5 = φ(arr_3, arr_4)
        arr_6 = arr_5.map do |i| i + 3; end          γ
    end

    arr_7 = φ(arr_1, arr_6)
    arr_7.to_a
end
```

$arr_1 = I[input]$

$arr_2 = \{I[input], arr_6\}$

$arr_3 = \{ C_\alpha[I[input]], C_\alpha[arr_6] \}$

$arr_4 = \{ C_\beta[I[input]], C_\beta[arr_6] \}$

$arr_5 = \{ C_\alpha[I[input]], C_\alpha[arr_6], C_\beta[I[input]], C_\beta[arr_6] \}$

Host Section: Example

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr₁ = input.to_pa

    for i in 0...10
        arr₂ = φ(arr1, arr6)

        if arr₂.reduce(:+)[0] % 2 == 0
            arr₃ = arr₂.map do |i| i + 1; end      α
        else
            arr₄ = arr₂.map do |i| i + 2; end      β
        end

        arr₅ = φ(arr₃, arr₄)
        arr₆ = arr₅.map do |i| i + 3; end          γ
    end

    arr₇ = φ(arr₁, arr₆)
    arr₇.to_a
end
```

```
arr₁ = I[input]

arr₂ = {I[input], arr₆}

arr₃ = { Cα[I[input]],
         C[arr₆] }

arr₄ = { Cβ[I[input]],
         Cβ[arr₆] }

arr₅ = { Cα[I[input]],
         Cα[arr₆],
         Cβ[I[input]],
         Cβ[arr₆] }

arr₆ = { Cγ[Cα[I[input]]],
         Cγ[Cα[arr₆]],
         Cγ[Cβ[I[input]]],
         Cγ[Cβ[arr₆]] }
```

Circular definition

Host Section: Example

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr₁ = input.to_pa

    for i in 0...10
        arr₂ = φ(arr1, arr6)

        if arr₂.reduce(:+)[0] % 2 == 0
            arr₃ = arr₂.map do |i| i + 1; end
        else
            arr₄ = arr₂.map do |i| i + 2; end
        end

        arr₅ = φ(arr₃, arr₄)
        arr₆ = arr₅.to_a.to_pa.map do |i| i + 3; end
    end

    arr₇ = φ(arr₁, arr₆)
    arr₇.to_a
end
```

arr_1 = I[input]

arr_2 = { I[input], C_γ[I[arr_5]] }

arr_3 = { C_α[I[input]], C_α[C_γ[I[arr_5]]] }

arr_4 = { C_β[I[input]], C_β[C_γ[I[arr_5]]] }

arr_5 = { C_α[I[input]], C_α[C_γ[I[arr_5]]], C_β[I[input]], C_β[C_γ[I[arr_5]]] }

arr_6 = { C_γ[I[arr_5]] }

Host Section: Example

東京工業大学
Tokyo Institute of Technology

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr_1 = input.to_pa

    for i in 0...10
        arr_2 = φ(arr1, arr6)

        if arr_2.reduce(:+)[0] % 2 == 0
            arr_3 = arr_2.map do |i| i + 1; end
        else
            arr_4 = arr_2.map do |i| i + 2; end
        end

        arr_5 = φ(arr_3, arr_4)
        arr_6 = arr_5.to_a.to_pa.map do |i| i + 3; end
    end

    arr_7 = φ(arr_1, arr_6)
    arr_7.to_a
end
```

$arr_1 = I[input]$

$arr_2 = \{ I[input], C_\gamma[I[arr_5]] \}$

$arr_3 = \{ C_\alpha[I[input]], C_\alpha[C_\gamma[I[arr_5]]] \}$

$arr_4 = \{ C_\beta[I[input]], C_\beta[C_\gamma[I[arr_5]]] \}$

$arr_5 = \{ C_\alpha[I[input]], C_\alpha[C_\gamma[I[arr_5]]], C_\beta[I[input]], C_\beta[C_\gamma[I[arr_5]]] \}$

$arr_6 = \{ C_\gamma[I[arr_5]] \}$

$arr_7 = \{ I[input], C_\gamma[I[arr_5]] \}$

Host Section: Example

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr1 = input.to_pa

    for i in 0...10
        arr2 = φ(arr1, arr6)

        if arr2.reduce(:+)[0] % 2 == 0
            arr3 = arr2.map do |i| i + 1; end
        else
            arr4 = arr2.map do |i| i + 2; end
        end

        arr5 = φ(arr3, arr4)
        arr6 = arr5.to_a.to_pa.map do |i| i + 3; end
    end

    arr7 = φ(arr1, arr6)
    arr7.to_a
end
```

```
arr_1 = I[input]

arr_2 = { I[input],
          C_γ[I[arr_5]] }

arr_3 = { C_α[I[input]],
          C_α[C_γ[I[arr_5]]] }

arr_4 = { C_β[I[input]],
          C_β[C_γ[I[arr_5]]] }

arr_5 = { C_α[I[input]],
          C_α[C_γ[I[arr_5]]],
          C_β[I[input]],
          C_β[C_γ[I[arr_5]]] }

arr_6 = { C_γ[I[arr_5]] }

arr_7 = { I[input],
          C_γ[I[arr_5]] }
```

8 kernels generated up front (may consist of mult. CUDA kernels)

Polymorphic Expressions

```
a = 37
a = true

a & 9
```

```
union value_v_t {
    int int_;
    bool bool_;
    ...
}

struct union_t {
    int class_id;
    union_v_t value;
}

union_t a;
a = union_t::make_int(1, 37);
a = union_t::make_bool(2, true);

switch (a.class_id) {
    case 1:  /* integer & */ break;
    case 2:  /* bool & */ break;
}
```

class ID determines type of expression

Host Section: Translation

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr1 = input.to_pa

    for i in 0...10
        arr2 = φ(arr1, arr6)

        if arr2.reduce(:+)[0] % 2 == 0
            arr3 = arr2.map do |i| i + 1; end
        else
            arr4 = arr2.map do |i| i + 2; end
        end

        arr5 = φ(arr3,
        arr6 = arr5.to_
    end

    arr7 = φ(arr1, arr6)
    arr7.to_a
end
```

```
arr4 = [&] {
  union_t result;
  switch (arr2.class_id) {
    case ID(I[Input]):
      result = union_t::make_cmd(ID(Cβ[I[input]]), arr2);
      break;
    case ID(Cγ[I[arr5]]):
       result = union_t::make_cmd(ID(Cβ[Cγ[I[arr5]]]), arr2);
       break;
  } result; } ();
```

maintain pointer to depending array command (containing kernel input)

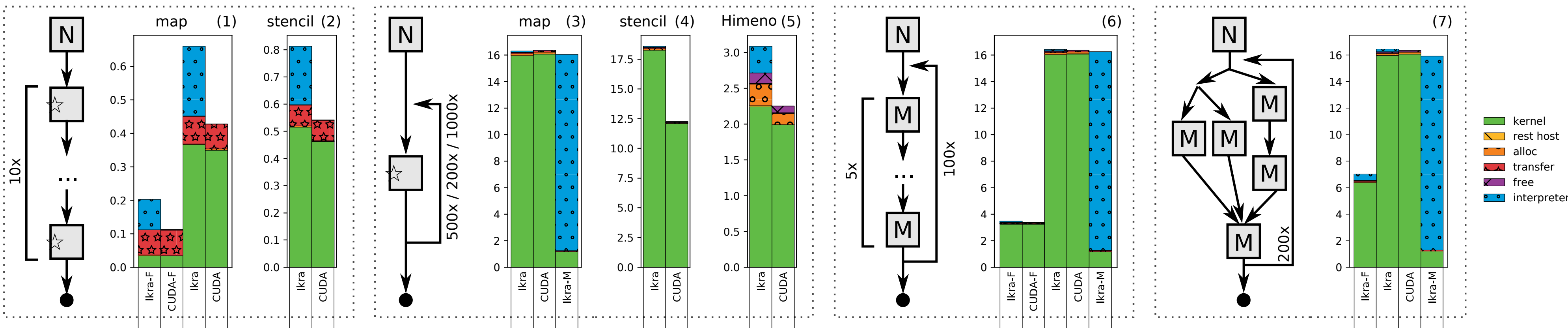
class_id corresponds to specific kernel combination

Host Section: Translation

```
input = [10, 20, 30, 40, 50, 60]

result = Ikra.host_section do
    arr₁ = input.to_pa

    for i in 0...10
        arr₂ = φ(arr1, arr6)

        if arr₂.reduce(:+)[0]
            arr₃ = arr₂.map do
        else
            arr₄ = arr₂.map
        end

        arr₅ = φ(arr₃, arr₄)
        arr₆ = arr₅.to_a.to_pa.map do |i| i + 3; end
    end

    arr₇ = φ(arr₁, arr₆)
    arr₇.to_a
end
```

```
[&] {
  location_aware_array_t result;
  switch (arr₅.class_id) {
    case ID(Cα[I[input]]):
      int *d_result;
      cudaMalloc(&d_result, 6 * sizeof(int));
      kernel_Mα_I_input<<<...>>>(arr₅.value);
      result = make_array(DEVICE, d_result);
      break;
    case ...
  } result; } ()
```

Benchmarks



	(1) Ikra-F	(1) CUDA-F	(1) Ikra	(1) CUDA	(2) Ikra	(2) CUDA	(3) Ikra	(3) CUDA	(3) Ikra-M	(4) Ikra	(4) CUDA	(5) Ikra	(5) CUDA	(6) Ikra-F	(6) CUDA-F	(6) Ikra	(6) CUDA	(6) Ikra-M	(7) Ikra-F	(7) Ikra	(7) CUDA	(7) Ikra-M
style	no loop						simple loop							simple loop					complex loop			
kernel operation	map				stencil		map			stencil				map					map			
#loop iterations	n/a				n/a		500			200		1000		100					200			
#kernels in loop	n/a				n/a		1			1		1		5					5			
with fusion	✓	✓							✓					✓	✓			✓	✓			✓
with host section	✓		✓		✓		✓			✓		✓		✓		✓			✓			
#kernels after fusion	1	1							1					4	2			1	8			1
#runtime kernel invocations	1	1	11	11	11	11	501	501	1	201	201	1001	1001	101	101	501	501	1	201	var.	var.	1

- Ikra-F/CUDA-F: With Kernel Fusion
- Ikra/CUDA: Without Kernel Fusion
- Ikra-M: Without host section, single kernel

Overview

1. Introduction
2. Parallel Operations
3. Modular Programming
4. Iterative Computations
5. **Benchmarks**
6. Summary

Future Work

- More parallel operations (select, prefix_sum, ...)
- Memory Management and Garbage Collection
  - Free device memory automatically
- Fusion of Stencil Operations: Temporal Blocking

Summary

東京工業大学
Tokyo Institute of Technology

- *Ikra:* Ruby extension for GPU Computing
- *Modularity:* Compose program of small parallel operations
- Integration with *Dynamic Language Features*
  - Restricted set of types/operations in parallel sect.
  - All Ruby features (metaprogramming, external libraries, ...) in other code
- Optimization for Iterative Computations:
  (Host) section of code that is entirely translated to C++

Appendix

Kernel Fusion

```
f = proc { |i| i + 1 }
g = proc { |i| i + 2 }
arr.map(&f).map(&g)
```

Every map operation creates a new array (i.e., must write to global memory)

```
fg = proc { |i| (i + 1) + 2 }
arr.map(&g)
```

```
h = proc { |i, j, k| i + j + k }
arr.map(&g).stencil([-1, 0, 1], 0, &h)
```

For every i, g(i) is computed three times

```
gh = proc { |i, j, k| g(i), g(j), g(k) }
arr.stencil([-1, 0, 1], 0, &gh)
```

Modular Programming

東京工業大学
Tokyo Institute of Technology

- **Modularity:** Understandability, reusability, composability
- Write multiple small parallel sections instead of a single big one, e.g.:
  - Matrix Multiplication
  - Graph Traversal Frontier
  - Image Manipulation Library

```
left.map { |row|
  right.transpose.map { |col|
    row
      .zip(col)
      .map { |x, y| x * y }
      .reduce(0, :+)
  }
}
```

Modular Programming

- **Modularity:** Understandability, reusability, composability
- Write multiple small parallel sections instead of a single big one, e.g.:
  - Matrix Multiplication
  - Graph Traversal Frontier
  - Image Manipulation Library

```
queue = [start_vertex].to_pa
step = proc { |v|
  ...
  next_vertices }

while !queue.empty?
  queue = queue
    .map(&step)
    .flatten
    .uniq
end
```

Modular Programming

- **Modularity:** Understandability, reusability, composability
- Write multiple small parallel sections instead of a single big one, e.g.:
  - Matrix Multiplication
  - Graph Traversal Frontier
  - Image Manipulation Library

```
queue = [start_vertex].to_pa

while !queue.empty?
  frontier = PArray.new(|v|, false)
  queue.each { |v|
    ...; frontier[?] = true }
  queue = frontier
   .map.with_index { |f, i| [f, i] }
   .select { |z| z[0] }
   .map { |z| z[1] }
end
```

bool frontier array + stream compactation

F	T	T	F	T
[F, 0]	[T, 1]	[T, 2]	[F, 3]	[T, 4]
[T, 1]	[T, 2]	[T, 4]		
1	2	4		