SIMD/MIMD Semantics of a Higher-Order Rank-Polymorphic Programming Language

Anfisa Bogdanenko

August 14, 2026

Khoury College of Computer Science
Northeastern University

Background

If You Want to Go Fast, Go Parallel!

- Computationally intensive applications are in high demand: scientific computing, graphics, finance, and most recently AI.
- These workloads require parallel computation over massive data, which can only achieve real speedups when executed on specialized parallel hardware.

If You Want to Go Fast, Go Parallel!

- Computationally intensive applications are in high demand: scientific computing, graphics, finance, and most recently AI.
- These workloads require parallel computation over massive data, which can only achieve real speedups when executed on specialized parallel hardware.

Programming parallel hardware is a notoriously complex and error-prone task.
The Remora project seeks to make it better.

Key Terminology: SIMD, MIMD, and Compute Elements

There are two fundamental models of parallel execution.

- Under **multiple-instruction multiple-data (MIMD)**, independent instruction streams run over independent pieces of data.
- Under **single-instruction multiple-data (SIMD)**, one instruction stream runs over many independent pieces of data.

MIMD



ex: multi-core CPU executes under MIMD*

SIMD



ex: parallel hardware executes under SIMD*

Key Terminology: SIMD, MIMD, and Compute Elements

There are two fundamental models of parallel execution.

- Under **multiple-instruction multiple-data (MIMD)**, independent instruction streams run over independent pieces of data.
- Under **single-instruction multiple-data (SIMD)**, one instruction stream runs over many independent pieces of data.

MIMD



ex: multi-core CPU executes under MIMD*

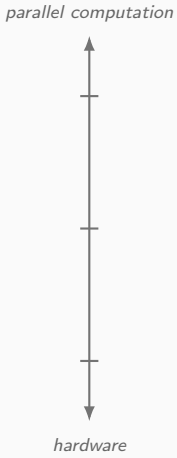
SIMD



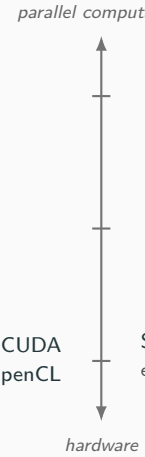
ex: parallel hardware executes under SIMD*

A **compute element (CE)** is one component of a SIMD computation that has its own data but shares the instruction stream with other CEs. Under SIMD, CEs run in lockstep.

How We Program Parallel Hardware



How We Program Parallel Hardware



SIMD kernel APIs

ex:
```
for (int i = 0; i < 10; i++)
    c[i] = a[i] + b[i];
```

+ precise control over the hardware
− verbose, error-prone, architecture-specific

How We Program Parallel Hardware



SAC, Futhark
DSLs like PyTorch

SIMD array languages

ex: `(+ [4 1 3] [6 1 2])`

\+ the compiler makes most parallelism decisions

– is not general-purpose; requires a host

CUDA
OpenCL

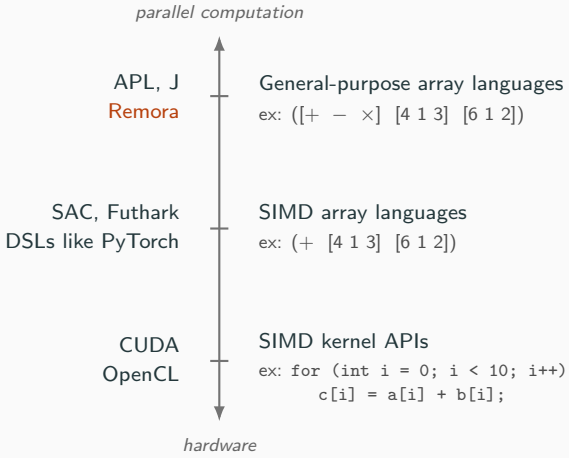
SIMD kernel APIs

```
ex: for (int i = 0; i < 10; i++)
      c[i] = a[i] + b[i];
```

\+ precise control over the hardware

– verbose, error-prone, architecture-specific

How We Program Parallel Hardware



\+ one high-level general-purpose language with a powerful optimizing compiler

− ???

\+ the compiler makes most parallelism decisions

− is not general-purpose; requires a host

\+ precise control over the hardware

− verbose, error-prone, architecture-specific

Remora: a Higher-Order Rank-Polymorphic Array Language

Higher-order: functions are first-class values and can be passed around freely. The function position of a call expects a procedure that will be applied to a sequence of arguments.

$$(f \;\; a_0 \;\; a_1 \;\; \ldots)$$

Remora: a Higher-Order Rank-Polymorphic Array Language

Higher-order: functions are first-class values and can be passed around freely. The function position of a call expects a procedure that will be applied to a sequence of arguments.

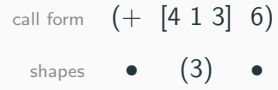
$$(f \;\; a_0 \;\; a_1 \;\; \ldots)$$

In an array language, *every* value is an array, including arrays of functions.
All four of these are valid Remora calls:

(+ 3 4)	scalar function, scalar arguments
(+ [4 1 3] [6 1 2])	scalar function, vector arguments — SIMD
([+ + +] [4 1 3] [6 1 2])	vector function, vector arguments — SIMD
([+ − ×] [4 1 3] [6 1 2])	vector function, vector arguments — MIMD

Remora: a Higher-Order Rank-Polymorphic Array Language

Rank-polymorphic: every function application lifts over the principal frame of its arguments.



Remora: a Higher-Order Rank-Polymorphic Array Language

Rank-polymorphic: every function application lifts over the principal frame of its arguments.

call form	$(+$	$[4\ 1\ 3]$	$6)$
shapes	$\bullet$	(3)	$\bullet$

- Primitive $+$ consumes two scalars, but we have one 3-vector and one scalar.
- A **cell** is the part of the argument consumed by the procedure.
- Extra dimensions become an argument's **frame**.

Remora: a Higher-Order Rank-Polymorphic Array Language

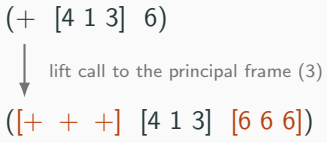
Rank-polymorphic: every function application lifts over the principal frame of its arguments.

call form	(+	[4 1 3]	6)
shapes	•	(3)	•

- Primitive + consumes two scalars, but we have one 3-vector and one scalar.
- A **cell** is the part of the argument consumed by the procedure.
- Extra dimensions become an argument's **frame**.
- The **principal frame** is the longest frame in the call. Here it is (3).

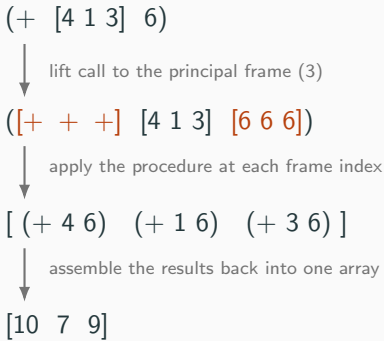
Remora: a Higher-Order Rank-Polymorphic Array Language

Rank-polymorphic: every function application lifts over the principal frame of its arguments.



Remora: a Higher-Order Rank-Polymorphic Array Language

Rank-polymorphic: every function application lifts over the principal frame of its arguments.



SIMD Kernel Extraction

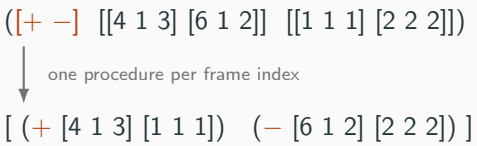
MIMD Splits

Parallel hardware can only execute SIMD, but Remora can express MIMD.

([+ −] [[4 1 3] [6 1 2]] [[1 1 1] [2 2 2]])

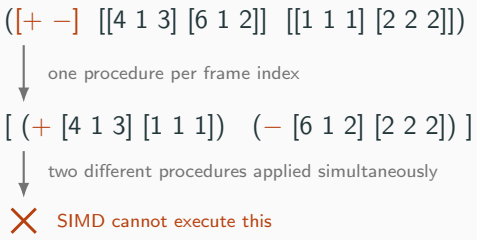
MIMD Splits

Parallel hardware can only execute SIMD, but Remora can express MIMD.



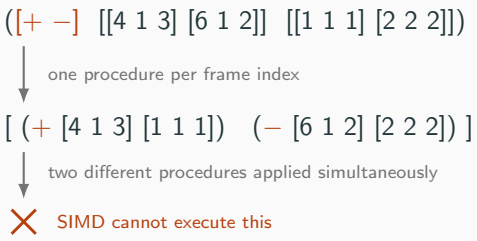
MIMD Splits

Parallel hardware can only execute SIMD, but Remora can express MIMD.



MIMD Splits

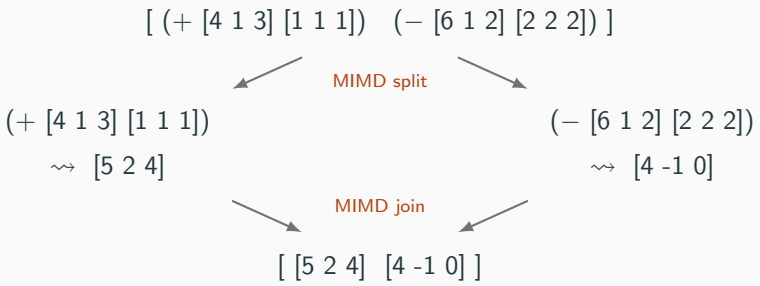
Parallel hardware can only execute SIMD, but Remora can express MIMD.



We call the point at which parallel execution diverges a **MIMD split**.

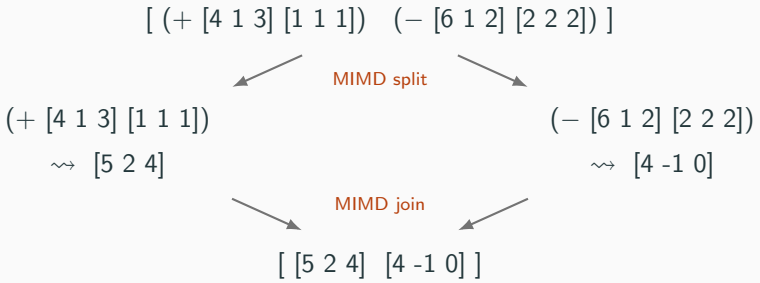
SIMD Subcomputations and MIMD Joins

Every MIMD computation breaks apart into independent **SIMD subcomputations**.



SIMD Subcomputations and MIMD Joins

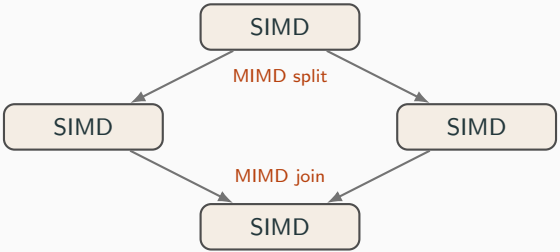
Every MIMD computation breaks apart into independent **SIMD subcomputations**.



The overall MIMD result can only be assembled once every subcomputation is done.
The MIMD join is a synchronization barrier.

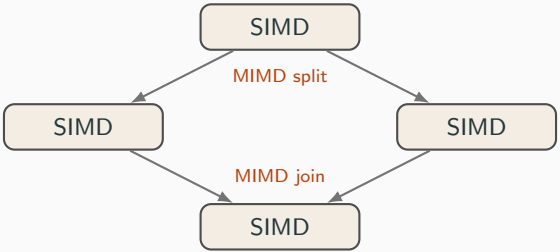
MIMD Splits are SIMD Boundaries

Every Remora computation consists of SIMD fragments bounded by MIMD splits.



MIMD Splits are SIMD Boundaries

Every Remora computation consists of SIMD fragments bounded by MIMD splits.



These fragments are the **SIMD kernels** that the compiler wants to run on parallel hardware.
Extracting them from the code statically turns out to be challenging.

MIMD Splits are Semantic

Remora is higher-order, so an array of procedures can be passed through a variable reference:

```
(f  [4 1 3] [6 1 2])
```

MIMD Splits are Semantic

Remora is higher-order, so an array of procedures can be passed through a variable reference:

$$(f \quad [4\ 1\ 3]\ [6\ 1\ 2])$$

This call site can execute SIMD or MIMD depending on *what value flows into it*.
For example:

- If f is bound to $[+\ -\ \times]$, the application results in a 3-way MIMD split;
- If f is bound to $\times$, the application executes SIMD.

MIMD Splits are Semantic

Remora is higher-order, so an array of procedures can be passed through a variable reference:

$$(f \;\; [4\ 1\ 3]\ [6\ 1\ 2])$$

This call site can execute SIMD or MIMD depending on *what value flows into it*.
For example:

- If f is bound to $[+\ -\ \times]$, the application results in a 3-way MIMD split;
- If f is bound to $\times$, the application executes SIMD.

Determining which procedures reach a call site, and whether they arrive as a part of a single array, is a control flow analysis question. We call it the **SIMD/MIMD analysis**.

My Thesis

A full SIMD/MIMD analysis is a large undertaking. This thesis lays the semantic groundwork needed to derive one: an operational semantics for Remora in which MIMD splits are explicit.

My Thesis

A full SIMD/MIMD analysis is a large undertaking. This thesis lays the semantic groundwork needed to derive one: an operational semantics for Remora in which MIMD splits are explicit.

It contributes two mathematical artifacts:

- CPS Remora, a factored continuation-passing-style intermediate representation, in which every transfer of control is an explicit call;
- The SIMD/MIMD Machine, an operational semantics for CPS Remora.

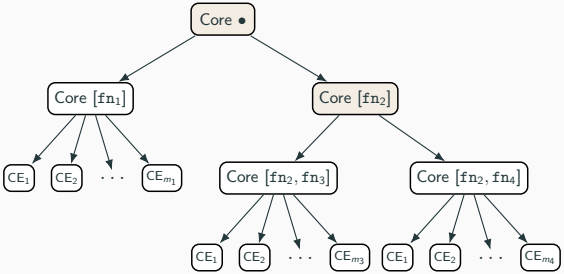
This Talk

1. The SIMD/MIMD Machine Overview
2. Collections: A Flexible Array Representation
3. Execution Trace, Step By Step
4. Remaining Work & Conclusion

The SIMD/MIMD Machine Overview

SIMD/MIMD Machine Structure

- **Core**: a SIMD evaluator with arbitrary parallelism. Each core addresses a set of CEs.
- **SIMD/MIMD Machine**: a tree of cores branching on MIMD splits.
- When a core encounters a MIMD split, it spawns child cores for every SIMD subcomputation and suspends until all children are done.

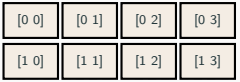


Core's Parallel State: D and Σ

Cores keep track of their current parallel state during execution:

- **Domain** D: set of the indices corresponding to the CEs this core owns.
- **Parallel context** Σ: the upper-bound iteration space.

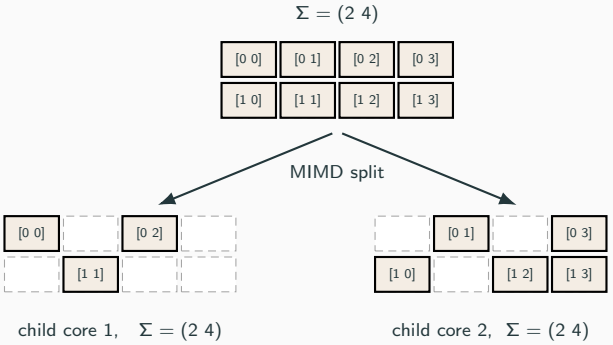
$$\Sigma = (2\ 4)$$



Core's Parallel State: D and Σ

Cores keep track of their current parallel state during execution:

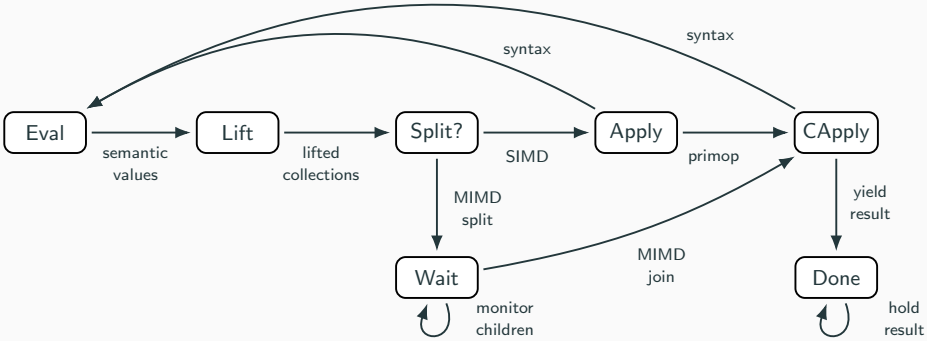
- **Domain** D: set of the indices corresponding to the CEs this core owns.
- **Parallel context** Σ: the upper-bound iteration space.



Core States and Transitions

A core is fully described by its state tuple: $\langle st,\ \Sigma,\ D \rangle^{cid}$.

Here, st is one of seven **execution states** and the core steps deterministically between them:



Collections

Collections

The fundamental value SIMD/MIMD Machine operates over is a **collection** $\lceil\!(\psi \mid \omega) \parallel m \rceil\!\rfloor$.

- **Index map** m holds the underlying data, as nested partial maps from indices to values;
- **Collection shape** $(\psi \mid \omega)$ encodes how that data is spread across CEs by splitting array shape into frame dimensions ψ and cell dimensions ω.

$$\lceil\!(2 \mid 3) \parallel \begin{pmatrix} 0 \mapsto \begin{pmatrix} 0 \mapsto 4 \\ 1 \mapsto 1 \\ 2 \mapsto 3 \end{pmatrix} \\ 1 \mapsto \begin{pmatrix} 0 \mapsto 6 \\ 1 \mapsto 1 \\ 2 \mapsto 2 \end{pmatrix} \end{pmatrix} \rceil\!\rfloor$$

One Array, Many Collections

The bar in $(\psi \mid \omega)$ partitions the array shape and it can be adjusted freely. Each position describes a different amount of parallelism over the same data.

$\lceil (2\ 3 \mid \bullet) \parallel m \rceil$

six CEs,
each sees a scalar

$\lceil (2 \mid 3) \parallel m \rceil$

two CEs,
each sees a 3-vector

$\lceil (\bullet \mid 2\ 3) \parallel m \rceil$

one CE,
sees the whole array

One Array, Many Collections

The bar in $(\psi \mid \omega)$ partitions the array shape and it can be adjusted freely. Each position describes a different amount of parallelism over the same data.

$\lceil (2\ 3 \mid \bullet) \parallel m \rceil$

six CEs,
each sees a scalar

$\lceil (2 \mid 3) \parallel m \rceil$

two CEs,
each sees a 3-vector

$\lceil (\bullet \mid 2\ 3) \parallel m \rceil$

one CE,
sees the whole array

This allows the SIMD/MIMD Machine to adjust the shape of parallelism without ever changing the underlying data.

Replicated Dimensions

Collections can encode data replicated along a dimension by underlining it:

$$\left\lceil (2\ 2 \mid \bullet) \right\| \begin{pmatrix} 0 \mapsto \begin{pmatrix} 0 \mapsto 3 \\ 1 \mapsto 3 \end{pmatrix} \\ 1 \mapsto \begin{pmatrix} 0 \mapsto 4 \\ 1 \mapsto 4 \end{pmatrix} \end{pmatrix} \left\rceil \right. \quad \text{behaves the same as} \quad \left\lceil (2\ \underline{2} \mid \bullet) \right\| \begin{pmatrix} 0 \mapsto 3 \\ 1 \mapsto 4 \end{pmatrix} \left\rceil \right.$$

- A replicated dimension is not backed by data in the index map because every position along the axis is the same.
- Rank-polymorphic lifting is free in the SIMD/MIMD Machine.

Indexing

The @ operator indexes the collection's frame and returns a cell sitting at that index. The index must have the same rank as the frame. Replicated dimensions are skipped.

$$\boxed{v} = \lceil (3\ \underline{4} \mid \bullet) \Vert \begin{pmatrix} 0 \mapsto 4 \\ 1 \mapsto 1 \\ 2 \mapsto 3 \end{pmatrix} \rfloor$$

$$\boxed{v} \text{ @ } [0\ 0] = 4 \qquad \boxed{v} \text{ @ } [1\ 0] = 1$$

$$\boxed{v} \text{ @ } [0\ 1] = 4 \qquad \boxed{v} \text{ @ } [2\ 2] = 3$$

Execution Trace, Step By Step

The Example Program

Direct-Style Remora

$$
\begin{array}{l}
(\times\ ((\lambda\ (f\ 0)\ .\ (f\ \ 1\ \ 10)) \\
\qquad [+\ -]) \\
\qquad 2)
\end{array}
$$

CPS Remora

$$
\begin{array}{l}
(\mathtt{fn}_{\mathtt{TOP}}\ []\ k_{\mathtt{TOP}}\ . \\
\quad ((\mathtt{fn}_{\mathtt{L1}}\ [(f\ 0)]\ k_1\ .\ (f\ \ 1\ \ 10\ \ k_1)) \\
\qquad [+\ -] \\
\qquad (\mathtt{cont}_{\mathtt{L2}}\ y\ .\ (\times\ y\ 2\ k_{\mathtt{TOP}}))))
\end{array}
$$

- f is bound to an array of *two different* procedures, so calling it causes a MIMD split.
- $\mathtt{fn}_{\mathtt{L1}}$ is the standard lambda, $\mathtt{cont}_{\mathtt{L2}}$ is the continuation lambda.
- k_1, $k_{\mathtt{TOP}}$ are variables binding continuation expressions.

Step 1: Eval Turns Syntax into Values

$$\langle \quad \text{execution state} \quad \Sigma,\ D \ \rangle^{cid}$$

1: $\langle \text{Eval}(\beta,\ [\![((\texttt{fn}_{\text{L1}}\ \ldots)\ \ [+\ -]\ \ (\texttt{cont}_{\text{L2}}\ \ldots))]\!]),\ \bullet,\ \{[\bullet]\}\rangle^{[\,]}$

2: $\langle \text{Lift}(\ulcorner\llcorner(\bullet \mid \bullet) \parallel \texttt{fn}_{\text{L1}}\urcorner,\ \ulcorner\llcorner(\bullet \mid 2) \parallel [0 \mapsto +,\ 1 \mapsto -]\urcorner,\ \kappa_{\text{L2}}),\ \bullet,\ \{[\bullet]\}\rangle^{[\,]}$

- Σ and D are empty because there is no parallelism yet.
- $\texttt{fn}_{\text{L1}}$ and $[+\ -]$ become collections with empty frames. Array shapes are cell shapes.
- $\texttt{cont}_{\text{L2}}$ becomes a continuation value. It captures $\texttt{cont}_{\text{L2}}$ and the core's current Σ and D. The parallel state will be reinstated when the continuation value is invoked.

$$\kappa_{\text{L2}} = \langle(\texttt{cont}_{\text{L2}}\ y\ \ldots),\ \bullet,\ \{[\bullet]\}\rangle$$

Step 2: Rank-Polymorphic Lifting

2: $\langle \mathsf{Lift}(\ulcorner\lfloor(\bullet \mid \bullet) \parallel \mathtt{fn}_{\mathrm{L1}}\urcorner,\ \ulcorner\lfloor(\bullet \mid 2) \parallel [0 \mapsto +,\ 1 \mapsto -]\urcorner\rfloor,\ \kappa_{\mathrm{L2}}),\ \bullet,\ \{[\bullet]\}\rangle^{[\,]}$

3: $\langle \mathsf{Split?}(\ulcorner\lfloor(\underline{2} \mid \bullet) \parallel \mathtt{fn}_{\mathrm{L1}}\urcorner,\ \ulcorner\lfloor(2 \mid \bullet) \parallel [0 \mapsto +,\ 1 \mapsto -]\urcorner\rfloor,\ \kappa_{\mathrm{L2}}),\ (2),\ \{[0], [1]\}\rangle^{[\,]}$

- The procedure in the function position ($\mathtt{fn}_{\mathrm{L1}}$ $[(f\ 0)]$ ...) expects one rank 0 argument.
- The argument collection has rank 1, so the principal frame is (2).
- The procedure collection is replicated to match the principal frame.
- Σ and D grow to match the principal frame. The core now addresses two CEs.

Step 3: Split? No, Stay SIMD

3: $\langle \text{Split?}(\ulcorner\underline{(2 \mid \bullet)} \parallel \texttt{fn}_{\text{L1}} \urcorner,\ [\ulcorner\underline{(2 \mid \bullet)} \parallel [0 \mapsto +,\ 1 \mapsto -] \urcorner],\ \kappa_{\text{L2}}),\ (2),\ \{[0], [1]\}\rangle^{[\,]}$

4: $\langle \text{Apply}(\texttt{fn}_{\text{L1}},\ \ulcorner\underline{(2 \mid \bullet)} \parallel [0 \mapsto +,\ 1 \mapsto -] \urcorner,\ \kappa_{\text{L2}}),\ (2),\ \{[0], [1]\}\rangle^{[\,]}$

- A MIMD split happens when the collection in the function position has more than one distinct procedure. Here it is just $\texttt{fn}_{\text{L1}}$, so the core can execute SIMD over two CEs.
- The core transitions to a SIMD *Apply* state that takes the procedure as its "single instruction".

Skipping Ahead

$$
\begin{array}{l}
(\mathtt{fn}_{\mathtt{TOP}}\ []\ k_{\mathtt{TOP}}\ . \\
\quad ((\mathtt{fn}_{\mathtt{L1}}\ [(f\ 0)]\ k_1\ .\ (f\ \ 1\ \ 10\ \ k_1)) \\
\quad\quad [+\ -] \\
\quad\quad (\mathtt{cont}_{\mathtt{L2}}\ y\ .\ (\times\ y\ 2\ k_{\mathtt{TOP}}))))
\end{array}
$$

Steps 4 to 7 enter the lambda and work through its body:

4 $\mathtt{fn}_{\mathtt{L1}}$ is applied. Its parameter f binds at rank 0, so each of the two CEs gets one procedure out of $[+\ -]$.

5 The body $(f\ \ 1\ \ 10\ \ k_1)$ is evaluated: f becomes the procedure collection bound to it, 1 and 10 become scalar collections, and k_1 becomes $\kappa_{\mathtt{L2}}$.

6 Lifting replicates scalars across the two CEs, so 1 becomes $\lceil (\underline{2} \mid \bullet) \parallel 1 \rceil$. This time Σ and D do not grow — the parallelism is already there.

Step 7: Split? Yes, MIMD Split

7: $\langle \mathsf{Split?}(\lceil\lfloor(2 \mid \bullet) \parallel [0 \mapsto +,\ 1 \mapsto -]\rfloor\rceil,\ [\lceil\lfloor(\underline{2} \mid \bullet) \parallel 1\rfloor\rceil,\ \lceil\lfloor(\underline{2} \mid \bullet) \parallel 10\rfloor\rceil],\ \kappa_{\mathrm{L2}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

8: $\langle \mathsf{Wait}(\{[+],[-]\},\ \kappa_{\mathrm{L2}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

$\langle \mathsf{Apply}(+,\ [\lceil\lfloor(\underline{2} \mid \bullet) \parallel 1\rfloor\rceil,\ \lceil\lfloor(\underline{2} \mid \bullet) \parallel 10\rfloor\rceil],\ \langle \texttt{yield}([\,]),\ (2),\ \{[0]\}\rangle),\ (2),\ \{[0]\}\rangle^{[+]}$

$\langle \mathsf{Apply}(-,\ [\lceil\lfloor(\underline{2} \mid \bullet) \parallel 1\rfloor\rceil,\ \lceil\lfloor(\underline{2} \mid \bullet) \parallel 10\rfloor\rceil],\ \langle \texttt{yield}([\,]),\ (2),\ \{[1]\}\rangle),\ (2),\ \{[1]\}\rangle^{[-]}$

- The collection in the function position has two distinct procedures: $\{+,\ -\}$.
  We partition D by procedure: $D_+ = \{[0]\}$, $D_- = \{[1]\}$
- Each child core gets a `yield` continuation with its parent core id.
- The parent suspends in *Wait* state, holding the original continuation.
  The state captures a set of spawned child core ids.

Steps 8 and 9: Computing Results Fragments

8: ... child cores execute SIMD Apply ...

9: $\langle \text{Wait}(\{[+],[-]\},\ \kappa_{L2}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

$\langle \text{CApply}(\langle \texttt{yield}([\,]),\ (2),\ \{[0]\}\rangle,\ \lceil\!\lfloor (2 \mid \bullet) \parallel [0 \mapsto 11] \rfloor\!\rceil),\ (2),\ \{[0]\}\rangle^{[+]}$

$\langle \text{CApply}(\langle \texttt{yield}([\,]),\ (2),\ \{[1]\}\rangle,\ \lceil\!\lfloor (2 \mid \bullet) \parallel [1 \mapsto \text{-}9] \rfloor\!\rceil),\ (2),\ \{[1]\}\rangle^{[-]}$

10: $\langle \text{Wait}(\{[+],[-]\},\ \kappa_{L2}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

$\langle \text{Done}(\lceil\!\lfloor (2 \mid \bullet) \parallel [0 \mapsto 11] \rfloor\!\rceil,\ [\,]),\ (2),\ \{[0]\}\rangle^{[+]}$

$\langle \text{Done}(\lceil\!\lfloor (2 \mid \bullet) \parallel [1 \mapsto \text{-}9] \rfloor\!\rceil,\ [\,]),\ (2),\ \{[1]\}\rangle^{[-]}$

- Each child ran SIMD *Apply* over its partial CE domain. We call these result fragments.
- The index maps in result collections are restricted to the core's domain.
- Child cores then yield and transition into a terminal *Done* state.

Step 10: MIMD Join

10: $\langle \text{Wait}(\{[+], [-]\},\ \kappa_{\text{L2}}),\ (2),\ \{[0], [1]\}\rangle^{[\,]}$

$\langle \text{Done}(\lceil\lfloor(2 \mid \bullet) \parallel [0 \mapsto 11]\rfloor\rceil,\ [\,]),\ (2),\ \{[0]\}\rangle^{[+]}$

$\langle \text{Done}(\lceil\lfloor(2 \mid \bullet) \parallel [1 \mapsto \text{-}9]\rfloor\rceil,\ [\,]),\ (2),\ \{[1]\}\rangle^{[-]}$

11: $\langle \text{CApply}(\kappa_{\text{L2}},\ \lceil\lfloor(2 \mid \bullet) \parallel [0 \mapsto 11,\ 1 \mapsto \text{-}9]\rfloor\rceil),\ (2),\ \{[0], [1]\}\rangle^{[\,]}$

(child cores die)

- The parent core stays in *Wait* state until all its child cores reach *Done* state.
- After result fragments are available, the parent assembles the MIMD result collection.
- The parent invokes the stored continuation value with the result.

Step 11: Parallelism Contracts

11: $\langle \mathsf{CApply}(\langle(\mathtt{cont}_{\mathtt{L2}}\ y\ \ldots), \bullet, \{[\bullet]\}\rangle, \lceil\!\lfloor (2 \mid \bullet) \parallel [0 \mapsto 11,\ 1 \mapsto \text{-}9] \rfloor\!\rceil), (2), \{[0], [1]\}\rangle^{[\,]}$

12: $\langle \mathsf{Eval}(\beta,\ [\![(\times\ y\ 2\ k_{\mathtt{TOP}})]\!]), \bullet, \{[\bullet]\}\rangle^{[\,]}$

- Invoking a continuation value reinstates the captured parallel state, that is Σ and D. The core drops back to a scalar CE.
- The result's parallelism is adjusted accordingly before binding it to y:

$$\lceil\!\lfloor (2 \mid \bullet) \parallel [0 \mapsto 11,\ 1 \mapsto \text{-}9] \rfloor\!\rceil \rightsquigarrow \lceil\!\lfloor (\bullet \mid 2) \parallel [0 \mapsto 11,\ 1 \mapsto \text{-}9] \rfloor\!\rceil$$

- Applications grow parallelism; returns shrink it.

Skipping To The End

$$
\begin{array}{l}
(\mathtt{fn}_{\mathtt{TOP}}\ []\ k_{\mathtt{TOP}}\ . \\
\quad ((\mathtt{fn}_{\mathtt{L1}}\ [(f\ 0)]\ k_1\ .\ (f\ \ 1\ \ 10\ \ k_1)) \\
\quad\quad [+\ -] \\
\quad\quad (\mathtt{cont}_{\mathtt{L2}}\ y\ .\ (\times\ y\ 2\ k_{\mathtt{TOP}}))))
\end{array}
$$

Steps 12 to 18 run the continuation's body and finish:

12 → 14 The call is evaluated and lifted. $\times$ expects scalars, so the principal frame is (2) again. The core's Σ and D grow back to (2) and $\{[0], [1]\}$.

14 → 16 One procedure in function position, so the call is SIMD. Each CE multiplies its scalar by 2.

16 → 18 $k_{\mathtt{TOP}}$ is bound to a top-level continuation, which halts the machine with the final result:

$$\ulcorner (\bullet \mid 2) \parallel [0 \mapsto 22,\ 1 \mapsto \text{-}18] \urcorner$$

Remaining Work & Conclusion

Remaining Work

- **Handle higher-order primitives** like `reduce`, a delayed source of splits.
- **Design the analysis output.** An IR that makes *maximal* kernels extractable: SIMD fragments as paths through call sites, not labels on them.
- **Abstract the machine.** In general, we cannot run the program at compile-time to find its splits because so the machine state space is infinite. It has to be safely collapsed into a finite one.
- **Proofs**. Given a CPS converter, prove soundness of the SIMD/MIMD Machine against direct-style Remora. Then, prove machine properties like schedule independence and progress. Finally, prove safety of abstraction.

Conclusion

This thesis delivers:

1. CPS Remora, a factored CPS IR for a higher-order rank-polymorphic language.
2. The SIMD/MIMD Machine, an operational semantics in which the shape of parallelism, the set of available compute elements, and MIMD splits are made explicit.
3. A Haskell implementation of the model that accumulates execution traces.

Conclusion

This thesis delivers:

1. CPS Remora, a factored CPS IR for a higher-order rank-polymorphic language.
2. The SIMD/MIMD Machine, an operational semantics in which the shape of parallelism, the set of available compute elements, and MIMD splits are made explicit.
3. A Haskell implementation of the model that accumulates execution traces.

In Remora, MIMD splits are a semantic notion.
To identify them statically, they need to be explicit in the semantics of the language.
The SIMD/MIMD Machine achieves that goal.

Questions?

Full Execution Trace

Full Execution Trace (1/3)

$$
\begin{array}{l}
(\mathtt{fn}_{\mathtt{TOP}}\ []\ k_{\mathtt{TOP}}\ . \\
\quad ((\mathtt{fn}_{\mathtt{L1}}\ [(f\ 0)]\ k_1\ .\ (f\ \ 1\ \ 10\ \ k_1)) \\
\qquad [+\ -] \\
\qquad (\mathtt{cont}_{\mathtt{L2}}\ y\ .\ (\times\ y\ 2\ k_{\mathtt{TOP}}))))
\end{array}
$$

0: $\langle \mathsf{Apply}(\varsigma_{\mathtt{TOP}},\ \bullet,\ \kappa_{\mathtt{HALT}}),\ \bullet,\ \{[\bullet]\}\rangle^{[\,]}$

1: $\langle \mathsf{Eval}(\beta,\ [\![((\mathtt{fn}_{\mathtt{L1}}\ \ldots)\ [+\ -]\ \mathtt{cont}_{\mathtt{L2}})]\!]),\ \bullet,\ \{[\bullet]\}\rangle^{[\,]}$

2: $\langle \mathsf{Lift}(\lceil\lfloor(\bullet \mid \bullet)\rfloor \parallel \mathtt{fn}_{\mathtt{L1}}\rceil,\ [\lceil\lfloor(\bullet \mid 2)\rfloor \parallel [0 \mapsto +,\ 1 \mapsto -]\rceil],\ \kappa_{\mathtt{L2}}),\ \bullet,\ \{[\bullet]\}\rangle^{[\,]}$

3: $\langle \mathsf{Split?}(\lceil\lfloor(\underline{2} \mid \bullet)\rfloor \parallel \mathtt{fn}_{\mathtt{L1}}\rceil,\ [\lceil\lfloor(2 \mid \bullet)\rfloor \parallel [0 \mapsto +,\ 1 \mapsto -]\rceil],\ \kappa_{\mathtt{L2}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

4: $\langle \mathsf{Apply}(\mathtt{fn}_{\mathtt{L1}},\ [\lceil\lfloor(2 \mid \bullet)\rfloor \parallel [0 \mapsto +,\ 1 \mapsto -]\rceil],\ \kappa_{\mathtt{L2}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

5: $\langle \mathsf{Eval}(\beta,\ [\![(f\ \ 1\ \ 10\ \ k_1)]\!]),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

6: $\langle \mathsf{Lift}(\lceil\lfloor(2 \mid \bullet)\rfloor \parallel [0 \mapsto +,\ 1 \mapsto -]\rceil,\ [\lceil\lfloor(\bullet \mid \bullet)\rfloor \parallel 1\rceil,\ \lceil\lfloor(\bullet \mid \bullet)\rfloor \parallel 10\rceil],\ \kappa_{\mathtt{L2}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

Full Execution Trace (2/3): the Split

7: $\langle \mathsf{Split?}(\lceil\lfloor (2 \mid \bullet) \parallel [0 \mapsto +,\ 1 \mapsto -] \rfloor\rceil,\ [\lceil\lfloor (2 \mid \bullet) \parallel 1 \rfloor\rceil,\ \lceil\lfloor (2 \mid \bullet) \parallel 10 \rfloor\rceil],\ \kappa_{\mathrm{L2}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

8: $\langle \mathsf{Wait}(\{[+],[-]\},\ \kappa_{\mathrm{L2}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$
$\langle \mathsf{Apply}(+,\ [\lceil\lfloor (2 \mid \bullet) \parallel 1 \rfloor\rceil,\ \lceil\lfloor (2 \mid \bullet) \parallel 10 \rfloor\rceil],\ \langle \texttt{yield}([\,]),\ (2),\ \{[0]\}\rangle),\ (2),\ \{[0]\}\rangle^{[+]}$
$\langle \mathsf{Apply}(-,\ [\lceil\lfloor (2 \mid \bullet) \parallel 1 \rfloor\rceil,\ \lceil\lfloor (2 \mid \bullet) \parallel 10 \rfloor\rceil],\ \langle \texttt{yield}([\,]),\ (2),\ \{[1]\}\rangle),\ (2),\ \{[1]\}\rangle^{[-]}$

9: $\langle \mathsf{Wait}(\{[+],[-]\},\ \kappa_{\mathrm{L2}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$
$\langle \mathsf{CApply}(\langle \texttt{yield}([\,]),\ (2),\ \{[0]\}\rangle,\ \lceil\lfloor (2 \mid \bullet) \parallel [0 \mapsto 11] \rfloor\rceil),\ (2),\ \{[0]\}\rangle^{[+]}$
$\langle \mathsf{CApply}(\langle \texttt{yield}([\,]),\ (2),\ \{[1]\}\rangle,\ \lceil\lfloor (2 \mid \bullet) \parallel [1 \mapsto \text{-}9] \rfloor\rceil),\ (2),\ \{[1]\}\rangle^{[-]}$

10: $\langle \mathsf{Wait}(\{[+],[-]\},\ \kappa_{\mathrm{L2}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$
$\langle \mathsf{Done}(\lceil\lfloor (2 \mid \bullet) \parallel [0 \mapsto 11] \rfloor\rceil,\ [\,]),\ (2),\ \{[0]\}\rangle^{[+]}$
$\langle \mathsf{Done}(\lceil\lfloor (2 \mid \bullet) \parallel [1 \mapsto \text{-}9] \rfloor\rceil,\ [\,]),\ (2),\ \{[1]\}\rangle^{[-]}$

11: $\langle \mathsf{CApply}(\langle(\texttt{cont}_{\mathrm{L2}}\ y\ \ldots),\ \bullet,\ \{[\bullet]\}\rangle,\ \lceil\lfloor (2 \mid \bullet) \parallel [0 \mapsto 11,\ 1 \mapsto \text{-}9] \rfloor\rceil),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

(children die)

Full Execution Trace (3/3)

12: $\langle \mathsf{Eval}(\beta,\ [\![(\times\ y\ 2\ k_{\mathtt{TOP}})]\!]),\ \bullet,\ \{[\bullet]\}\rangle^{[\,]}$

13: $\langle \mathsf{Lift}(\lceil\lfloor(\bullet \mid \bullet) \parallel \times \rceil\rfloor,\ [\lceil\lfloor(\bullet \mid 2) \parallel [0 \mapsto 11,\ 1 \mapsto \text{-}9] \rceil\rfloor,\ \lceil\lfloor(\bullet \mid \bullet) \parallel 2 \rceil\rfloor],\ \kappa_{\mathtt{HALT}}),\ \bullet,\ \{[\bullet]\}\rangle^{[\,]}$

14: $\langle \mathsf{Split?}(\lceil\lfloor(\underline{2} \mid \bullet) \parallel \times \rceil\rfloor,\ [\lceil\lfloor(2 \mid \bullet) \parallel [0 \mapsto 11,\ 1 \mapsto \text{-}9] \rceil\rfloor,\ \lceil\lfloor(\underline{2} \mid \bullet) \parallel 2 \rceil\rfloor],\ \kappa_{\mathtt{HALT}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

15: $\langle \mathsf{Apply}(\times,\ [\lceil\lfloor(2 \mid \bullet) \parallel [0 \mapsto 11,\ 1 \mapsto \text{-}9] \rceil\rfloor,\ \lceil\lfloor(\underline{2} \mid \bullet) \parallel 2 \rceil\rfloor],\ \kappa_{\mathtt{HALT}}),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

16: $\langle \mathsf{CApply}(\kappa_{\mathtt{HALT}},\ \lceil\lfloor(2 \mid \bullet) \parallel [0 \mapsto 22,\ 1 \mapsto \text{-}18] \rceil\rfloor),\ (2),\ \{[0],[1]\}\rangle^{[\,]}$

17: $\langle \mathsf{Done}(\lceil\lfloor(\bullet \mid 2) \parallel [0 \mapsto 22,\ 1 \mapsto \text{-}18] \rceil\rfloor,\ [\,]),\ \bullet,\ \{[\bullet]\}\rangle^{[\,]}$

18: final state — the machine halts