Compiler for Remora

Liam Stevenson

December 15, 2023

Contents

1 Introduction

Remora is a typed, array-oriented programming language, whose type system and semantics were laid out in the thesis of Justin Slepak [1]. Inspired by the likes of Iverson's APL and J, it allows for implicit lifting of functions over arrays of data, allowing for shape-polymorphic computing. Unlike APL and J, it is a strongly typed language, which is beneficial beyond just catching errors at compile time. The

type system is dependent, allowing it to capture information about the shape of arrays. This shape information shows the compiler explicitly where parallelism occurs in the code. The hope is that the compiler can leverage this information to create programs that efficiently utilize a GPU. Remora is also a higher-order language, allowing operations like reduce and scan, which can also be parallelized. With these operators and the implicit lifting of functions over arrays, programmers can express highly parallel programs in a more intuitive and less error prone way. Remora's syntax is s-expression based, having a syntax that resembles Lisp, Scheme, and Racket. For more information on Remora, see [2] for a beginner's guide to the language.

The goal of this Master's Thesis is to create a compiler for the core of the language that produces GPU code as a stepping stone towards a more general and efficient compiler. Recursion was excluded from the core of the language, as users are expected to use higher-order functions like reduce and fold rather than relying on recursion. Additionally, higher-order usage of functions was restricted to cases where the compiler can determine the function being called statically. The compiler remains unfinished as of the time of writing this paper, with code generation not yet being complete.

The technical contributions of this thesis are as follows:

- A nearly complete compiler for core Remora that targets CUDA C
- An implementation of Remora's typing rules
- An algorithm for monomorphizing and fully inlining core Remora
- An algorithm for fusion of core Remora
- An algorithm for flattening and kernel generation of core Remora

2 Introduction to Remora

This section provides a quick overview of Remora. For a much better, more detailed introduction that is quite readable, see *An Introduction to Rank-polymorphic Programming in Remora* [2].

2.1 Rank-Polymorphism

Remora is rank-polymorphic, a programming model invented by Kenneth Iverson when he created APL [3]. In this model, all values are arrays. When you assign a scalar value like `7` to a variable `x`, `x` is thought of as a scalar array — that is, an array of rank 0. `x` could instead have a value of `[1 2]`, making it an array of rank 1, or a value of `[[1 2] [3 4]]`, making it an array of rank 2, *etc.*

In Remora, function parameters are specified to be of an expected rank, and it produces a value that also has an expected rank. These are the function's "cells", and the expected ranks are statically included as part of the function's definition. For example, `+` has arguments of rank 0 and a return value of rank 0. `dot-product`, on the other hand, has arguments of rank 1 and a return value of rank 0, since is expects two arrays and returns a scalar array.

Function arguments are allowed to have any rank greater than or equal to the rank of the parameter they correspond to. If the argument rank is greater than the parameter rank, the function is mapped over the argument's cells. For example, consider the function `succ`, which takes an integer array of rank 0 and produces an array of rank 0 by adding `1`. `(succ 1)` would evaluate to `(succ 2)`. `(succ [1 2])`, which is equivalent to `[(succ 1) (succ 2)]`, would evaluate to `[2 3]`.

Mapping functions over the cells of its arguments is also allowed on functions with multiple arguments. Consider `(+ [1 2] [3 4])`. The two arguments are mapped over simultaneously, so this is equivalent to `[(+ 1 3) (+ 2 4)]`, which evaluates to `[4 6]`. A case like `(+ [[1 2 3] [4 5 6]] [7 8])` is also legal. In this, the cells of the second argument are replicated so that the shapes match. Thus, `(+ [[1 2 3] [4 5 6]] [7 8])` is equivalent to `(+ [[1 2 3] [4 5 6]] [[7 7 7] [8 8 8]])`, which is the same as `[[(+ 1 7) (+ 2 7) (+ 3 7)] [(+ 4 8) (+ 5 8) (+ 6 8)]]`, which evaluates to `[[8 9 10] [12 13 14]]`. For a more thorough explanation of this, see the sections *Functions operate on "cells" of input* and *Functions distribute over a frame of cells* of *An Introduction to Rank-polymorphic Programming in Remora* [2].

2.2 Type System

Remora is a typed language, and its type system uses dependent typing to allow users to specify the shape of arrays. The type of an array thus includes both the type of the elements of the array (`int`, `bool`, *etc.*) and the shape of an array. For example, the array `[[1 2 3] [4 5 6]]` has type `[int 2 3]`, since it is a 2x3 array of integers. `[#t #f]` has type `[bool 2]`. `#t` has type `bool` (the `[]` can be dropped for scalar arrays). Function types are written as `(→ (arg-types...) return-type)`. Notice that since a function's type includes information about the expected shapes of arguments, the rank of the function's cells can be determined from its type.

Types can be abstracted over with "type-variables". Type variables are variables that can appear in types and stand in for other types. They are expressed using a forall expression, as in the type `(∀ (@t) (→ (@t) @t))`, which is the type of a polymorphic identity function. The `@` symbol in the variable name is important — it indicates to Remora that the expected type is the type of an array. Without the `@`, it would expect an atomic type, like `int` or `bool`, as in the type `(∀ (t) (→ ([t 3]) t))`, which might be the type of a function that extracts the first element of a three element array (note that the return type `t` is shorthand for `[t]`, a scalar array with entries of type `t`). To create a type-polymorphic value, you can write an expression with the syntax `(Tλ (`*`tvar ...`*`) e)`. The specified type variables will then be available in the expression *e*. A `Tλ` can be used with the syntax `(t-app `*`t-lambda t-args ...`*`)`.

Similarly, array shapes can be abstracted over with "index-variables". They are expressed using a pi expression, as in the type `(Π (n) (→ ([int n] [int n]) int))`, which might be the type of a function that computes the dot product of two arrays. Notice that since both arrays are statically guaranteed to be of the same length (`n`)! Similarly to type-variables, index-variable names can optionally begin with an `@`. In the case of index-variables, the absence of a `@` indicates that the variable represents a single dimension, while the presence of a `@` indicates that the variable represents a *shape* (a sequence of dimensions). To create an index-polymorphic value, you can write an expression with the syntax `(Iλ (`*`ivar ...`*`) e)`. The specified index variables will then be available in the expression *e*. A `Iλ` can be used with the syntax `(i-app `*`i-lambda i-args ...`*`)`.

Some operations are also allowed to be performed on index variables. Two dimension indices are allowed to be added together to produce a new dimension index, and two shape indices are allowed to be appended together. For example, the type of a function that appends two rank-1 arrays together may be `(Π (a b) (→ ([int a] [int b]) [int (+ a b)]))`. However, these are the limits of Remora's dependent type system. Notably, multiplying two dimensions together is forbidden.

With this under our belts, we can now look at some example function definitions. Below is `double`, which doubles an integer:

```
(define double (λ ([x int])
                 (+ x x)))
```

Below is `id`, which simply returns the given value:

```
(define id (Tλ (@t)
             (λ ([x @t])
               x)))
```

Below is `dot-product`, which computes the dot product of two vectors (assuming we have the function `sum` at our disposal):

```
(define dot-product (Iλ (n)
                      (λ ([x [int n]] [y [int n]])
                        (sum (+ x y)))))
```

There is one last major element of Remora's type system: boxes. Boxes exist in Remora for two reasons.

First, sometimes you may have an array of arrays where the length of the inner arrays is not uniform. As an example, consider the array `[[0] [0 1] [0 1 2]]`. It would be incorrect to give it the type `[int 3 1]`, `[int 3 2]`, or `[int 3 3]`. In fact, `[[0] [0 1] [0 1 2]]` is not a legal expression in Remora! To express such a value, boxes must be used.

Second, sometimes it is impossible to statically determine the size of an array. Maybe the program asks for user input that determines the length of an array. Or maybe Remora's dependent type system isn't powerful enough to express the shape of an array. Consider the function `filter-zero`, which takes an argument of type `[int 10]` (an array of 10 integers) and removes all zeroes occurring in it. What would be the shape return type? We cannot say because it could have a length anywhere from 0 to 10.

Boxes have an existential type that captures index variables. Let's consider the below example:

```
(boxes (len) [char len] [5]
  ((6) "Monday")
  ((7) "Tuesday")
  ((9) "Wednesday")
  ((8) "Thursday")
  ((6) "Friday"))
```

In Remora, strings are a syntactic sugar for an array of characters (`"Monday"` has type `[char 6]`). In this example, we want to create an array of arrays of characters, where each array of characters is the name of a day of the week. Since the name lengths are not uniform, we need to use boxes. This expression has type `[(Σ (len) [char len]) 5]`. That is, an array of 5 elements, where each element has type `(Σ (len) [char len])`. `Σ` is an existential type. `(Σ (len) [char len])` can be understood as, "There exists some index `len` such that the element has type `[char len]` (an array of characters with length `len`)". In the `boxes` expression, the meaning of the different elements are as follows:

- `(len)` — We are capturing the index variable `len`. Like with `Iλ`, we could use a `@` to indicate that the variable is a shape rather than a dimension.
- `[char len]` — The boxed elements will have type `[char len]`, for some `[char len]`.
- `[5]` — This is the shape of the array that contains the boxes.
- Last are individual expressions that represent the boxes. For each, we first specify what value `len` will have for that boxed value.

The example of `[[0] [0 1] [0 1 2]]` could be represented as follows:

```
(boxes (len) [int len] [3]
  ((1) [0])
  ((2) [0 1])
  ((3) [0 1 2]))
```

We can then look inside boxes using an unbox expression (using the polymorphic `length` function that is found in the standard library):

```
(unbox weekdays (day len)
  (= 6 ((t-app (i-app length len []) char) day)))
```

This would evaluate to `[#t #f #f #f #t]`.

2.3 Parallelism

Mapping of functions over the cells of arguments can be done completely in parallel. In the prior example of `(+ [[1 2 3] [4 5 6]] [7 8])`, separate threads or processors can independently compute `(+ 1 7)`, `(+ 2 7)`, `(+ 3 7)`, *etc.* Even more importantly, this parallelization is made explicit statically by the type system, enabling the compiler to parallelize the operation. Note that to enable this, Remora does not specify an execution order of the various `+` operations.

2.4 Higher-Order

Remora is a higher-order language. It allows functions to be treated as first-class objects, and the standard library provides higher-order functions like `reduce`, `fold`, and `filter`. (Note the absence of `map` — function calls in Remora provide `map` for free.)

Using these higher-order functions is strongly encouraged in Remora over using a recursive approach because they make the structure of the program explicit to the compiler. Consider a function that sums together the elements of an array. One way to write it is using a recursive approach. Below is such an implementation, written in pseudo-Remora:

```
(define sum-recursive (λ ([arr [int n]])
                        (if (> (length arr) 0)
                            (+ (head arr) (rest arr))
                            0)))
```

Below is such an implementation written using the `reduce` function:

```
(define sum-reduce (λ ([arr [int n]])
                     (reduce + arr)))
```

In Remora, an important distinction is drawn between `reduce` and `fold`. Unlike `fold`, `reduce` assumes the function given to it is associative, which enables it to be performed in parallel (see 3.2 for more information). Thus, by using `reduce` rather than a recursive approach, the compiler is able to easily make `sum-reduce` efficient on a parallel architecture. Doing so with `sum-recursive` would require complicated analysis of the code.

2.5 Example - Polynomial Evaluation

This section will show an implementation of a function that efficiently evaluates a polynomial at a specified value on a serial processor. To do so, a few standard library functions need to be introduced:

- `reduce/zero` - This performs a reduction, but an initial element is given. This is opposed to `reduce`, which doesn't take an initial value, but requires the array to be non-empty.
- `open-scan/zero` - A scan is similar to a reduction, except that it returns all intermediate values. For example, `(open-scan/zero * 1 [2 3 4])` would return `[1 2 6 24]`. The *open* refers to the result including the initial value as the first element of the array. (`(scan/zero * 1 [2 3 4])` returns `[2 6 24]`.)
- `with-shape` - This function creates an array with the shape of the first argument, replicating the second argument for each entry.

Below is the definition of `poly-eval`. For a more detailed explanation of the implementation, see [2]:

```
(define poly-eval (Iλ (l) (λ ([coeffs l] [x 0])
    ; x-powers is x^0, x^1, x^2, ...
    (define x-powers (open-scan/zero * 1
                                     (with-shape coeffs x)))
    ; Multiply the coefficients by their corresponding x-power,
    ; and then sum them up
    (reduce/zero + 0
                 (* coeffs x-powers)))))
```

3 Overview of GPU Programming

The target language of the compiler is CUDA C++, which is a C++ dialect developed by NVIDIA. CUDA C++ produces binaries that can be run on any NVIDIA CPU. This gives good portability due to the prevalence of them.

3.1 Basics

CUDA by Example [4] is a useful starting point for learning about GPU programming, as well as nVidia's documentation of CUDA [5]. GPUs allow a large number of threads to run simultaneously. Threads exist in a hierarchy: kernel, block, warp, and thread (moving from large- to small-scale granularity). A kernel can be thought of as a function that is run on a GPU (aka device in GPU programming) and is kicked off by code executing on the CPU (aka host). When a kernel is kicked off, the host specifies how many blocks and how many threads per block should be created. The device then queues blocks of threads to be run, and the kernel finishes once all blocks finish executing. Multiple blocks can be run at once, but the number of synchronous blocks is limited by device resources. When the GPU executes a block, it runs all of its threads concurrently. A block organized its threads into warps, which are discussed more later.

The host and device have separate memory spaces; the device cannot access memory stored on the host, and the host cannot access memory stored on device. There are three forms of memory available to the device: global memory, local memory, and registers. Global memory is memory that exists in device RAM. Any device thread can access global memory at any time, and pointers to locations may be passed around. Global memory can be copied to or from host memory. Also, a kernel cannot allocate global memory — it must be done by the host. Local memory exists on a per-block basis and is much more limited in space than global memory. When a block spawns, it may request a local memory allocation. This memory is shared across all threads in a block, but threads in different blocks do not have access to this memory. Lastly, GPUs may store data in registers, as with CPUs. Due to the large number of threads, the number of registers available on a per-thread basis is low, so the number of registers available can be a bottleneck for how many threads may run simultaneously.

When a block executes its threads, it arranges the threads into warps of size 32. All threads execute in a lockstep, SIMD way. This has two important consequences. First, MIMD parallelization is not suitable for GPUs. Consider the piece of code in Figure 1, in which the first thread of every warp executes a long running function `expensiveFunc`. Since all threads in a warp execute together, as the first thread executes `expensiveFunc`, the other 31 threads need to go along for the ride (although they won't perform any side-effects while doing so). This effectively means that the code slows down by a factor of 32. Second, when all 32 threads access 32 consecutive memory slots in the same step, these memory accesses are "coalesced", meaning that they cost the same as only 1 memory access. Since memory access is frequently a bottleneck for GPUs, such as in a dot-product, vector summation, or matrix transpose, ensuring memory coalesces can frequently be the difference between horribly inefficient code and highly optimized code. There is also a feature called "broadcasting", where all 32 threads can simultaneously access the same memory address with the cost of only 1 memory access.

The follow is a list of important takeaways to keep in mind:

- Kernels must be launched by the host, not the device. Although newer versions of CUDA do allow the device to spawn kernels, it is less efficient and best avoided. This means that nested loops are not conducive to parallel GPU execution and need to be "flattened" by the compiler.

Figure 1: Branching on a GPU

```
if (threadId % 32 == 0) {
    expensiveFunc();
}
```

- Warps prevent MIMD execution on GPUs.
- Efficient memory usage is important since it is frequently a bottleneck, so:
  - Coalescing is extremely important. The compiler should try its best to produces code that performs coalescing memory accesses.
  - Writing intermediate arrays to global memory is costly and should be avoided.
- Global memory cannot be allocated by the device. This can become an issue when the compiler does not know statically how much memory needs to be allocated, such as in an `iota` call.

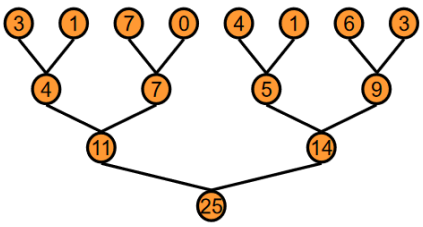
3.2 Experimentation

This section describes an investigation into implementing `reduce` in CUDA C++. While performing a `map` operation on GPU is straightforward, `reduce` is more complicated. Included are benchmarks of various reduce algorithm implementations on different sizes of data to determine which would be the best to use for Remora.

In parallel programming, we evaluate algorithm complexity in terms of two concepts: work and span. Work is the total amount of work that needs to be done and is essentially how long the algorithm would take with only one processor. Span is the maximum work that any one thread will perform, so it represents how long the algorithm would take to run with unlimited processors.

`reduce` does not immediately jump out as a parallel operation, and it's not in every case. However, if the combining operation is associative, the operations don't need to be performed in order, allowing for a tree-like reduction (see Figure 2). This allows `reduce` to have a span of $O(f_s(n) \cdot \log n)$ (and work of $O(f_w(n) \cdot n)$), assuming the combination operation has span $O(f_s(n))$ (and work $O(f_w(n))$).

Figure 2: Parallel Reduction



NVIDIA's library Thrust provides a parallel `reduce` operator, but it also requires the combining operation to be commutative, which is not necessary for the aforementioned asymptotic properties. The Remora standard library will have non-commutative versions of reduce, although commutative versions may be included. Thus, a goal was to implement an efficient `reduce` that does not use commutativity. The best performance reached was only 33% slower than the Thrust version. The reduction in speed is due to commutativity allowing for easy memory coalescing, whereas the non-commutative implementation needs to cache values in local memory in order to achieve coalesces.

Another case considered was performing a reduction along either rows or columns of a matrix, which has 4 cases:

1. Reduce along rows, where row length is large
2. Reduce along rows, where row length is small
3. Reduce along columns, where column length is large

4. Reduce along columns, where column length is small

This investigation assumed that matrices are stored in a row-major format. However, reducing along rows in a row-major format is the same problem as reducing along columns in a column-major format, and vice-versa. Case 1) is trivial — because rows are long and stored contiguously in memory, it is efficient to simply iterate over each row, performing a standard reduction on it. Case 2) is more complex. Initially, it is tempting to assign one thread to each row and iteratively sum elements, but this wouldn't allow for memory coalesces and is thus inefficient. However, a version was made that is about 50% slower than Case 1)'s by again utilizing local memory. It is also straightforward to make case 3) efficient, as individual threads iteratively sum the columns, which produces coalesces. Case 4) has not yet been explored, but it is believed that a similar method to Case 3) can be done.

A caveat with the experimentation that has been done is that it was only tested by summing floats. Array elements with a larger size may add additional challenges, mainly due to local memory constraints.

4 Compiler

The language of choice for implementing the compiler was OCaml. The project is available on GitHub at `https://github.com/liam923/remora`, which has a README describing how to run it. The project is currently in flux, but this section describes the state of it as of the time of writing (December 15, 2023).

The compiler is split into stages, which are described in the below sections. Each stage's code can be found in `./lib/Stages`, and intermediate data structures can be found in `./lib/IntermediateLanguages`. There are also supporting files in `./lib`.

4.1 Parse

The parsing stage is fairly standard and is split into two sub-stages.

First, the program is parsed into a list of s-expressions. The s-expressions used in Remora are an extension of the standard s-expression syntax found in languages like Lisp, and Figure 3 shows an Ocaml definition of the extended s-expressions (although it is simplified — the real definition also contains fields for tracking the source of tokens). A list has an additional field that says whether the brackets generating the list were parentheses or square brackets. This is important in Remora because unlike in Racket, there is a semantic difference between the two. There is also an additional form called `WithCurlies`. This captures expressions like `foo{a | b}`, which is used for some syntactic sugar that is discussed below. The parsing of s-expressions is performed using Menhir and OCamllex.

Second, the list of s-expressions is parsed into an AST by performing pattern matching on the s-expressions.

The syntax supported is dictated by two sources: *The Semantics of Rank Polymorphism* [6] and *Introduction to Rank-polymorphic Programming in Remora* [2]. An additional syntactic sugar was also added that makes explicitly-typed polymorphic Remora code more succinct. The syntax defined by these sources is overly verbose, making it hard to read, when defining or using polymorphic functions. Compare the verbose syntax in Figure 4 to the sugared syntax in Figure 5.

The rules governing this sugar are straightforward. There are two use cases: in definitions and in usages. For both, type arguments go before the `|` and index arguments go after. The bar must always be there even if there are just type arguments or just index arguments in order to distinguish the cases. In definitions, the presence of at least one type argument causes a type lambda to be inserted with those type arguments, and the same is true for indices. The index lambda goes on the outside if both are present. In usages, the system is similar. The presence of at least one type argument causes the expression to be wrapped in a type application, and the same is true for indices. If both are present, the index application goes on the inside, which corresponds to the index lambda being on the outside.

4.1.1 An Aside on Type Verbosity

Types in Remora can quickly become very large in size. For an example, see the type of `reduce` in Figure 6. To help with this issue, the intention is to eventually give Remora a type inference engine so that programmers do not need to write types explicitly. With type inference, the explicitly typed

Figure 3: Extended S-Expression Definition

```
type braceType =
  | Paren
  | Square

type t =
  | List of
      { braceType : braceType
      ; elements : t list
      }
  | WithCurlies of
      { base : 's t
      ; leftElements : t list
      ; rightElements : t list
      }
  | String of string
  | Integer of int
  | Symbol of string
```

Figure 4: Simple Polymorhpic Definition and Usage — No Sugar

```
; Note the verboseness of defining index and type variables
(define length (i-fn (d @cell-shape) (t-fn (t) (fn ([arr [t d @cell-shape]]) ...))))
; and of instantiating an instance of length
((t-app (i-app length 3 [2]) int) [[1 2] [3 4] [5 6]])
```

Figure 5: Simple Polymorhpic Definition and Usage — With Sugar

```
; Defining index and type variables is similar to many popular languages
(define (length{t | d @cell-shape} [arr [t d @cell-shape]]) ...)
; as is the usage of length
(length{t | d @cell-shape} [[1 2] [3 4] [5 6]])
```

version of Remora would act as an intermediate form to the compiler rather than be user facing. This would render the aforementioned sugar unnecessary, since types would rarely be written. However, since this type inference engine does not yet exist, this sugar was included so that current examples and tests are more readable, as the `t-app` and `i-app` forms quickly become confusing in moderately-complex code.

Figure 6: Type of `reduce`

```
(Π (d-1 @item-pad @cell-shape) (∀ (t)
    (→ ((→ ([t @cell-shape] [t @cell-shape]) [t @cell-shape])
        [t (+ d-1 1) @item-pad @cell-shape])
       [t @item-pad @cell-shape])))
```

4.2 Type Check

The type checker is also straightforward since the typing rules are explicitly laid out in *The Semantics of Rank Polymorphism* [6]. However, some conveniences that are described in *Introduction to Rank-polymorphic Programming in Remora (Draft)* [2] were implemented in the type-checking stage. Namely, implicit lifting of atoms to arrays. Whenever an atomic type `t` appears where an array type is expected, `t` is replaced with `(Arr t [])`. Similarly, whenever an atomic expression `e` appears where an array expression is expected, `e` is replaced with `(array [] e)`.

4.3 Explicitize

This stage of the compiler is so named because it makes explicit all of the maps that Remora implicitly includes at call sites. The result is a form where all function arguments are cells, and all of the implicit shape-lifting is reified in the map forms. The compiler treats the map form as a control syntax, not a primitive operation. It is then free to perform various optimizations on the maps down the line, such as fusion and flattening. An expression like `(+ [1 2] [[3 4 5] [6 7 8]])` is transformed to:

```
(map [2] (λ ([a int] [b [int 3]])
           (map [3] (λ (b2 int)
                      (+ a b2))
                    b))
         [1 2]
         [[3 4 5] [6 7 8]])
```

The map form is worth briefly elaborating on. First, the map is allowed to take multiple arguments. Second, the map takes an argument called the "frame shape" (`[2]` and `[3]` in the above example). The frame shape is the dimensions of the argument that are mapped over, allowing the map to map over multiple dimensions at once. As an example, `(+ [[1] [2]] [[3] [4]])` is transformed to:

```
(map [2 1]
     (λ ([a int] [b int])
       (+ a b))
     [[1] [2]]
     [[3] [4]])
```

Note that these transformed programs are actually a simplified form of what is produced. For implementation reasons, and to handle the case of the function being a non-scalar array, the full transformation of `(+ [1 2] [[3 4 5] [6 7 8]])` is:

```
(map [] (λ ([f1 (-> (int int) int)] [a1 [int 2]] [b1 [int 2 3]])
          (map [2] (λ ([a2 int] [b2 [int 3]])
                     (map [3] (λ ([b3 int])
                                (f1 a2 b3))
```

```
                        b2))
                  a1
                  b1)
 +
 [1 2]
 [[3 4 5] [6 7 8]])
```

Note that a map with frame shape `[]` is equivalent to a let. In fact, this compiler pass also transforms all lets into maps with an empty frame shape.

4.4 Inline

The inline stage presented the largest challenge to implement. It simultaneously monomorphizes the program while inlining all non-primitive function calls. It is so called because monomorphization can be thought of as inlining type and index applications. The produced code has no term, type, or index applications, besides primitive operations, as well as no term, type, or index lambdas. To reflect this, there are also no `->`, `Forall`, and `Pi` types.

Note that inlining all function calls and type applications *does not* remove all lambdas and type abstractions from the code, as they can remain in positions where they are not called or applied. For example, the program `(λ ([x int]) x)` remains the same after inlining. However, after inlining, it is guaranteed that no lambda or type abstraction will later be used. Therefore, it is safe to replace them with a dummy value. In this implementation, the compiler replaces them with a `unit` atom. Correspondingly, it replaces `->`, `Forall`, and `Pi` types with the type `Unit`.

At the end of this transformation, there are no remaining type variables. Some index variables, however, do remain. This is because index variables can arise from unbox expressions, and removing these in the general case would require runtime information unavailable at compile time. However, none of the remaining index variables arise from index lambdas, since there are no more index lambdas.

Fully inlining all function calls (besides primitives) is possible because no recursion is allowed, as well as one restriction that was added to the language. This restriction is that the compiler must be able to determine what function is in every function call position, and there cannot be multiple functions in the function call position (this can happen if you have an array of functions). Because of the aggressive inling and lack of recursion, the compiler can determine the function in most common use cases. Further, having an array of different functions is not supported for a second reason, which is that it represents MIMD parallelism. Since MIMD parallelism does not correspond well to GPUs, this project avoids it.

Also, because of full inlining, the issue of monomorphizing higher-rank polymorphism became a non-issue. There was, however, one restriction that needed to be added to the language. This was ML's value restriction, which says that a variable with polymorphic type must have a syntactic value as its value [7]. Remora expands this to also say that polymorphic function arguments must have a syntactic value. The value restriction ensures that monomorphization does not cause duplicated computations.

At the moment, the compiler also fails if a function that captures a variable is used outside the variable's scope. This restriction will hopefully be lifted with future work.

4.5 Nest

The Nest stage translates the IR to a form that is more conducive to the following stages. There are three main differences.

First, arrays can now have other arrays as their elements, and in exchange, the shape of an array can only include one "shape element", meaning either a single dimension or a variable referring to a shape. This is the reason for the name of the stage, as arrays are now nested rather than flattened. Similarly, maps are only allowed to map over exactly one shape element rather than an arbitrary shape. To handle the case of mapping over an empty shape, the let form is re-introduced. Mapping over a shape with multiple shape elements is converted to nested maps.

Second, tuples are now allowed. Tuples are a heterogeneous collection of elements and can be used to represent multi-valued returns of functions.

Third, the representation of "loops" (meaning library functions that would involve a loop when implemented serially, such as map, reduce, scan, and fold) is vastly different. The new representation is discussed in the below section on fusion.

4.6 Fuse and Simplify

This stage jointly performs simplification and fusion together in a loop until the program reaches a stable point. The reason for this loop is that simplification can enable fusion opportunities and vice-versa. The simplification and fusion sub-stages are described below:

4.6.1 Simplify

This stage performs a number of standard optimizations on the program. These are:

- Copy propagation
- If variable is used exactly once, and that usage is not in a loop, inline it
- If a variable's value is a value (a literal or another variable), inline it
- If a variable isn't used, delete it
- Constant folding
- Hoist expressions out of loops when possible (if the loop is guaranteed to be executed at least once)
- Remove unused elements of tuples

Some of these optimizations rely on Remora not allowing side-effects. However, this leads to some issues because Remora does allow exceptions to be thrown. For example, the removal of unused variables and unused tuple elements can cause exceptions that would otherwise be thrown to be silenced. This compiler implementation ignores these issues.

4.6.2 Fuse

Fusion is the process of merging together two loops where the output of one is the input of another. This is important to Remora because it removes intermediate arrays. The simplest example is `(map f (map g x))`. A simple evaluation of this expression produces and intermediate array where each element is `g` applied to an element of `x`. However, transforming the expression to the equivalent `(map (∘ f g) x)` immediately computes `(∘ f g)` on each element. Other transformations that are analogous on the CUDA C++ level but are not representable in Remora are also possible. For example, `(reduce f (map g x))` produces an intermediate array. However, a CUDA kernel can be generated that first performs `g` on each element and then performs a reduction using `f`.

To explain when fusion is and isn't possible, some definitions must be introduced. A **loop** is a function that takes an (n-dimensional) array as input. A **producer** is a loop where every element in an (n-dimensional) array that outputs is dependent on at most one element in the input array. A **consumer** is any loop that is not a producer. [8] While `map`, `iota`, and `transpose` are all producers, consumers include `reduce`, `fold`, and `filter`.

Consider a directed acyclic graph (DAG) where the nodes are the operations of a program and an edge from A to B represents the output of A being an input of B. A pair of nodes A and B can be fused if:

- A and B are both loops whose "frame shape" (number of iterations) is equal,
- A is a producer,
- and no node $C \neq A, B$ exists such that C is a child of A and a parent of B.

After fusing, the parents of the fused node F_{AB} is the union of the parents of A and B and likewise for the children.

In this implementation, for simplicity, the only function considered to be a producer is `map` (`iota` is implemented through the `map` construct so is also taken as a producer). The fusion algorithm is simply to search for a pair of fusible loops based on the definition above, fuse them, and repeat until there are no more fusible pairs.

To represent fused loops, a more powerful data structure called a "loop block" was created. A loop block contains two main parts: a `map` and an optional **consumer**. The map part includes a list of input arrays and a function which operates on the elements of the arrays. If there is a consumer, the result of the map is given as the input to it. The entire loop block then returns a tuple, where the first element is the result of the map and the second element is the result of the consumer (or `unit` if there is no consumer).

4.7 Kernelize

Kernelization is the process of flattening nested parallelism and selecting parallelization opportunities to translate into kernels. Flattening refers to squashing nested loops together into one loop. For example, `(map (\ (xs) (map f xs)) xss)` can be flattened to `(map f xss)`, where the new `map` maps over multiple dimensions. More complex patterns than this could also theoretically be matched, as Futhark does [9]. While for this implementation only nested maps are matched, this stage is implemented in a way such that adding additional patterns should be straightforward.

Selecting kernels is the process of choosing which loops will be translated into CUDA kernels and which will be translated to serial code. There are two complexities to this. First, code executing on device cannot spawn kernels.[1] This means that a loop nested inside a kernel cannot become a kernel, which is why flattening is important. Second, parallelizing a loop is sometimes not worthwhile. For example, if a loop adds 10 elements, it is faster to do this serially than pay the startup costs of spawning a kernel to do so.

Abstractly, one can think of a DAG where each node is a loop and A being a parent of B represents B being nested inside A. Note that this is different from the DAG in the section on fusion, in which arrows represented data dependencies. In this DAG, nodes may be labelled as kernels. However, a kernel node cannot be an ancestor of another kernel node. The question becomes: out of all possible ways of flattening nodes together and labelling nodes as kernels, which is the best?

For the sake of this implementation, "best" was taken to be the graph that provides the most parallelism while never over-parallelizing. The algorithm to do this, which relies on crude heuristics and is fairly straightforward, is a simple recursive walk of the the AST. At each node, if it is a loop, it considers all possible ways of turning the loop into a kernel. That is, in addition to simply kernelizing the loop, also kernelizing all possible flattenings of the loop. In addition to these, it considers not kernelizing the loop. It compares these choices and chooses the option that provides the highest degree of parallelism, or if there is not enough parallelism, it chooses the serial option. If this is unable to be determined due to a dimension not being known statically, multiple options are compiled and the decision is made at runtime.

4.8 Alloc

Memory allocation *can* be performed within a GPU kernel[2], but it is slow. Thus, it is best avoided. This compiler stage has three purposes. First, it annotates the AST with explicit memory allocations. Second, it pulls memory allocations out of loops where possible so that it is performed in larger chunks. Third, where possible, it hoists memory allocations out of GPU kernels.

4.9 Capture

This stage annotates kernels with the variables that they capture. This is because variables captured by kernels must be copied to device.

[1] Newer CUDA versions do allow this, but it is inefficient to spawn a large number of kernels and therefore not practical as a general strategy.

[2] Only on devices with CUDA compute capability 2.x or higher [5]

4.10 Codegen

The code generation stage involves translating the IR into CUDA C++. This stage is currently in progress as of the time of writing.

5 Future Work

To reach the goal of having a functional core Remora compiler that produces reasonably efficient CUDA C++ code, all that remains to be done is code generation. Beyond this, there is still a large amount of work to create a compiler that produces more optimized CUDA C++, and even more to support the full Remora language beyond just its core. Below are a suggestion of next steps after the completion of the core compiler:

1. Tuples and structs — The compiler handles tuples at a low level IR, but writing tuples in Remora is not currently supported. However, Justin Slepak has already created a proposal for semantic rules of records in Remora [10].
2. Loop tiling — Loop tiling can offer large performance gains on the GPU. It is an important technique in creating an optimized matrix multiplication, for example.
3. Transpose matrices — Futhark sometimes transposes matrices under the hood to get more memory coalesces.
4. Optimizations — Perform more complex standard optimizations on the program, like frame-map interchanging.
5. Hoist allocations — The compiler sometimes fails when it needs to allocate a dynamic amount of memory on device.
6. Box analysis — Unbox expressions don't *need* to handle arbitrary indices. In many cases, it's possible to narrow down the possible indices to a small set. An analysis could be performed that finds the possible values and compiles different versions of the unbox expression for those values. Then, a runtime check is performed to determine what path to go down. Since there is no recursion, this analysis should be straightforward to perform.
7. Optimize array operations — Operations like transpose, rotate, and subarray might not need to produced reified arrays.
8. Support MIMD — Add support for MIMD operations by allowing arrays of different functions in the function call position.
9. Support higher-order functions — Supporting truly higher-order functions would add much expressivity to the language and would be doable via defunctionalization once MIMD parallelization is supported.
10. Support recursion — Supporting recursion would make the language much more powerful, but also would make the compiler much trickier to implement.
11. Add Type Inference — Type inference would make the language much more usable

6 Related Work

6.1 Guy Blelloch

Guy Blelloch's book *Vector Models for Data-Parallel Computing* [11], which is a revision of his dissertation, shows a general algorithm for flattening any algorithm into one that "flattens" and utilizes all parallelization. However, while Blelloch's algorithm maintains asymptotic runtime characteristics, it is very inefficient in real world applications due to over-parallelizing in some cases, bad constant factors in other cases, and inefficient memory usage [12].

6.2 Accelerate

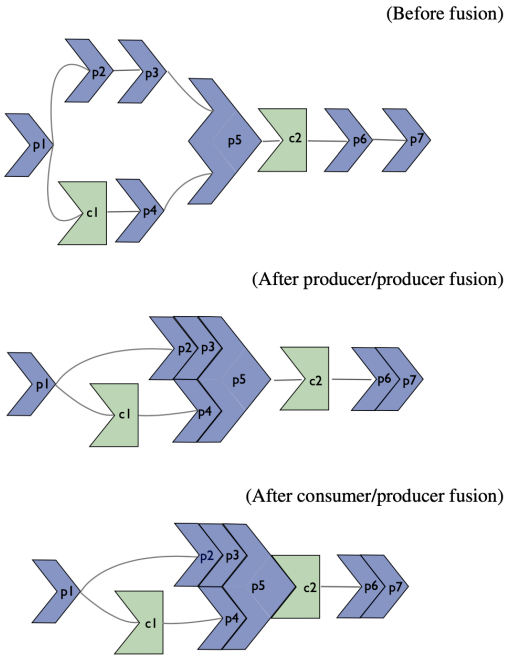
According to its website [13]:

> *Accelerate* is an embedded language of array-based computations for high-performance computing in Haskell. Programs in Accelerate are expressed in the form of parameterised collective operations on regular multidimensional arrays, such as maps, reductions, and permutations. Accelerate then takes these computations and optimises, compiles, and executes them for the chosen target architecture.

Accelerate has some limitations. For one, it does not have `filter` in it. It also is restricted to regular, flat parallelism. As previously mentioned, "flat" means that nested loops are not allowed. "Regular" means that all data being synchronously processed is of the same shape. An example of irregular parallelism would be performing a map over a 2-d array where the inner arrays have different size.

The paper *Optimising purely functional GPU programs* [14] explains how Accelerate performs fusion to eliminate intermediate arrays when performing computations. The paper divides all operations into producers and consumers. In producers, all elements of the output rely at most on one element of the input (map, iota, *etc.*), while consumers are others (reduce, fold, scan, *etc.*). Producer/producer fusion is done, where producers are combined together, and then consumer/producer fusion is done, where consumers acting on producers are combined (see below diagram). It is noted that producer/consumer would also be possible, but that isn't handled in this paper. This fusion is demonstrated in Figure 7.

Figure 7: Parallel Reduction



6.3 Futhark

According to its website [15]:

> Futhark is a small programming language designed to be compiled to efficient parallel code. It is a statically typed, data-parallel, and purely functional array language in the ML family, and comes with a heavily optimising ahead-of-time compiler that presently generates either GPU code via CUDA and OpenCL, or multi-threaded CPU code.

Futhark was created by Troels Henriksen for his PhD thesis [12] and is an ongoing research project. It is the state-of-the-art in compiling a functional language to efficient GPU code. In many cases, it produces GPU code that is competitive with hand-written, optimized GPU code. While it is a powerful language, it does have some limitations. Mainly, sometimes a type-safe program will fail to compile because the compiler wants to allocate memory from within a kernel. Also, like Remora's core, which is the focus of this project, it outlaws recursion.

Unlike Accelerate, Futhark is able to support irregular parallelism through support of segmented arrays. It also supports nested parallelism, which it is able to do through an algorithm called "Incremental Flattening". To flatten a program, Futhark first aggressively inlines function calls, which is possible due to a lack of recursion. It then looks for flattening opportunities, such as a map of a map, a reduce of a map, or a scan of a map. When it finds such cases, the compiler makes a choice. Based on the size of the array (which Futhark may know at compile time, due to having a dependent type system), the compiler chooses to either flatten the parallelism or only keep the inner or outer parallelism and do the other loop iteratively. If the size is not known at compile time, a runtime check is inserted to determine which way to go. This avoids the pitfalls of Guy Blelloch's method.

Futhark also does a number of other optimizations in order to achieve good performance, including fusion, loop tiling, and transposing matrices to ensure coalesced memory accesses.

6.4 PyTorch and TensorFlow

PyTorch and TensorFlow are extremely popular libraries for deep learning in Python. While the main purpose of the libraries is to enable users to create neural networks, both enable rank-polymorphic programming that is GPU-accelerated. They provide a large library of functions that are implemented in CUDA C and have Python wrappers for easy use.

Users can choose to run their programs in two different execution modes, which are usually called eager or graph modes. Eager mode is akin to how an interpreter would execute a program, where each function is executed without optimizations being performed. Graph mode is akin to compiling a program, where both PyTorch and TensorFlow construct a graph describing the program. Optimizations are then performed on these graphs before being run, which enables faster execution. [16] [17]

The optimizations performed by these libraries are fundamentally limited due to the nature of them. PyTorch and TensorFlow are both unable to generate *new* CUDA C kernels during optimization and instead are only able to piece together pre-defined ones. Thus, library authors can only hope to include kernels for common patterns and perform optimizations that efficiently assign sub-graphs of the graph to these kernels. This means that while PyTorch and TensorFlow can perform horizontal fusion, vertical fusion is limited.

7 References

[1] J. Slepak, "A typed programming language: The semantics of rank polymorphism.," Ph.D. dissertation, Northeastern University, 2020. [Online]. Available: https://ccs.neu.edu/~jrslepak/Dissertation.pdf.

[2] O. Shivers, J. Slepak, and P. Manolios, *Introduction to rank-polymorphic programming in remora (draft)*, 2020. arXiv: 1912.13451 [cs.PL].

[3] K. E. Iverson, *A Programming Language*. USA: John Wiley & Sons, Inc., 1962, ISBN: 0471430145.

[4] J. Sanders, E. Kandrot, and J. J. Dongarra, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley/Pearson Education, 2015.

[5] *Cuda toolkit documentation 12.2*, 2023. [Online]. Available: https://docs.nvidia.com/cuda/.

[6] J. Slepak, O. Shivers, and P. Manolios, *The semantics of rank polymorphism*, 2019. arXiv: 1907.00509 [cs.PL].

[7] *Value restriction*. [Online]. Available: http://mlton.org/ValueRestriction.

[8] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, "Optimising purely functional gpu programs," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '13, Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 49–60, ISBN: 9781450323260. DOI: 10.1145/2500365.2500595. [Online]. Available: https://doi.org/10.1145/2500365.2500595.

[9] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, Barcelona, Spain: Association for Computing Machinery, 2017, pp. 556–571, ISBN: 9781450349888. DOI: 10.1145/3062341.3062354. [Online]. Available: https://doi.org/10.1145/3062341.3062354.

[10] J. Slepak, O. Shivers, and P. Manolios, "Records with rank polymorphism," ser. ARRAY 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 80–92, ISBN: 9781450367172. DOI: 10.1145/3315454.3329961. [Online]. Available: https://doi.org/10.1145/3315454.3329961.

[11] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[12] T. Henriksen, "Design and implementation of the futhark programming language," Ph.D. dissertation, University of Copenhagen, Universitetsparken 5, 2100 KÃ¸benhavn, Nov. 2017.

[13] *Accelerate*. [Online]. Available: https://www.acceleratehs.org/documentation/users-guide/language.html.

[14] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, "Optimising purely functional gpu programs," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 49–60, 2013.

[15] *Why futhark?* [Online]. Available: https://futhark-lang.org/.

[16] J. Nie, C. Luk, X. Wang, and J. Xu. "Optimizing production pytorch models' performance with graph transformations." (2022), [Online]. Available: https://pytorch.org/blog/optimizing-production-pytorch-performance-with-graph-transformations/.

[17] TensorFlow. "Introduction to graphs and tf.function." (2023), [Online]. Available: https://www.tensorflow.org/guide/intro_to_graphs.

[18] J. Slepak, O. Shivers, and P. Manolios, "An array-oriented language with static rank polymorphism," in *Programming Languages and Systems*, Z. Shao, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 27–46.

[19] G. Blelloch, J. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee, "Implementation of a portable nested data-parallel language," *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 4–14, 1994, ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1994.1038. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731584710380.

[20] T. Henriksen and C. E. Oancea, "A t2 graph-reduction approach to fusion," in *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, ser. FHPC '13, Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 47–58, ISBN: 9781450323819. DOI: 10.1145/2502323.2502328. [Online]. Available: https://doi.org/10.1145/2502323.2502328.

[21] J. Reppy and J. Wingerter, λ_{cu} — *an intermediate representation for compiling nested data parallelism*, Presented at the *Compilers for Parallel Computing Workshop (CPC '16)*, Valladolid, Spain, Jul. 2016.

[22] T. Henriksen, K. F. Larsen, and C. E. Oancea, "Design and gpgpu performance of futhark's redomap construct," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY 2016, Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 17–24, ISBN: 9781450343848. DOI: 10.1145/2935323.2935326. [Online]. Available: https://doi.org/10.1145/2935323.2935326.

[23] T. Henriksen, M. Dybdal, H. Urms, *et al.*, "Apl on gpus: A tail from the past, scribbled in futhark," in *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, ser. FHPC 2016, Nara, Japan: Association for Computing Machinery, 2016, pp. 38–43, ISBN: 9781450344333. DOI: 10.1145/2975991.2975997. [Online]. Available: https://doi.org/10.1145/2975991.2975997.

[24] T. Henriksen, *Streaming combinators and extracting flat parallelism*, 2017. [Online]. Available: https://futhark-lang.org/blog/2017-06-25-futhark-at-pldi.html.

[25] R. W. Larsen and T. Henriksen, "Strategies for regular segmented reductions on gpu," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, ser. FHPC 2017, Oxford, UK: Association for Computing Machinery, 2017, pp. 42–52, ISBN: 9781450351812. DOI: 10.1145/3122948.3122952. [Online]. Available: https://doi.org/10.1145/3122948.3122952.

[26] T. Henriksen, *How futhark manages gpu memory*, 2018. [Online]. Available: https://futhark-lang.org/blog/2018-01-28-how-futhark-manages-gpu-memory.html.

[27] T. Henriksen, *Why futhark (sometimes) goes wrong*, 2018. [Online]. Available: https://futhark-lang.org/blog/2018-12-08-why-futhark-sometimes-goes-wrong.html.

[28] J. S. Bertelsen, "Implementing a cuda backend for futhark," M.S. thesis, University of Copenhagen, 2019.

[29] T. Henriksen, *Giving programmers what they want, or what they ask for*, 2019. [Online]. Available: https://futhark-lang.org/blog/2019-01-13-giving-programmers-what-they-want.html.

[30] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, "Incremental flattening for nested data parallelism," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19, Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 53–67, ISBN: 9781450362252. DOI: 10.1145/3293883.3295707. [Online]. Available: https://doi.org/10.1145/3293883.3295707.

[31] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea, "Memory optimizations in an array language," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22, Dallas, Texas: IEEE Press, 2022, ISBN: 9784665454445.

[32] T. Henriksen, *Incremental flattening for nested data parallelism on the gpu*, 2019. [Online]. Available: https://futhark-lang.org/blog/2019-02-18-futhark-at-ppopp.html.

[33] T. Henriksen, *What is the minimal basis for futhark?* 2019. [Online]. Available: https://futhark-lang.org/blog/2019-04-10-what-is-the-minimal-basis-for-futhark.html.

[34] T. Henriksen, *Higher-order parallel programming*, 2020. [Online]. Available: https://futhark-lang.org/blog/2020-05-03-higher-order-parallel-programming.html.

[35] T. Henriksen, *How futhark represents values at runtime*, 2021. [Online]. Available: https://futhark-lang.org/blog/2021-08-02-value-representation.html.

[36] P. Munksgaard, *Array short-circuiting*, 2022. [Online]. Available: https://futhark-lang.org/blog/2022-11-03-short-circuiting.html.

[37] T. Henriksen, *In-place mapping and the pleasure of beautiful code nobody will ever see*, 2022. [Online]. Available: https://futhark-lang.org/blog/2022-12-06-in-place-map.html.