Compiling Second-Order Accelerate Programs to First-Order TensorFlow Graphs

Master Thesis Computing Science, 40 EC.

L.P.J. van Soest BSc

Student Number 5999995

Abstract

Hardware acceleration is the method of accelerating calculations with hardware specifically designed for the type of calculations. Accelerate and TensorFlow are libraries that make this accessible to many programmers, but these libraries differ in the level of abstraction and targeted hardware. This thesis investigates the possibility of compiling and executing Accelerate programs in TensorFlow. A compiler is introduced that converts second-order Accelerate programs to first-order TensorFlow graphs, covering 68% of the Accelerate language.

A thesis presented for the degree of

Master of Science

Supervisors:	I.G. de Wolff MSc
	dr. T.L. McDonell
	prof. dr. G.K. Keller
	dr. W.S. Swierstra

Utrecht University

July 14 2023

Utrecht University

Dedicated to my family

and Nikky, my beloved significant other

Contents

1 Introduction

Hardware acceleration is the method of accelerating calculations by using hardware specifically designed for the type of calculations. For instance, to simulate light in real-time, the recent NVIDIA 4090 graphics card (GPU) has 128 cores dedicated to raytracing computations [17]. Often, addressing the capability of dedicated hardware requires expert knowledge. To alleviate this need, Accelerate is a language designed to compile generic parallel array instructions into hardware-specific code [10]. For example, the language supports code generation for multi-core CPUs and GPUs. This thesis proposes the addition of TensorFlow support to Accelerate's compiler. TensorFlow is a parallel machine learning library by Google [1]. The library allows the construction of graphs with matrix-like operations. Among others, the library features GPU support to perform its calculations. To compile Accelerate programs into TensorFlow, a new implementation is added to Accelerate's compilation pipeline. This thesis investigates the possible extent of the instruction set to cover each Accelerate program and enumerates its limitations. The remainder of section 1 introduces Accelerate and TensorFlow. Section 2 discusses the compilation process for Accelerate. Section 3 depicts the implementation of the instruction set of this process. Finally, sections 4 and 5 elaborate on code coverage and performance, and how future research could improve these results.

1.1 Data-parallelism

Accelerate is a data-parallel language. To quote Bergstrom et al. [7], data-parallel computations apply a function to the elements of a collection in parallel. Data parallelism is an effective technique that takes advantage of parallel hardware and is especially suited for large-scale operations. With nested data parallelism, parallel instructions may contain nested parallel instructions of arbitrary complexity [22]. For illustration, example 1 depicts a simple quicksort implementation.

Example 1: A simple quicksort implementation, can be parallelised by spawning new processes to compute *ls* and *hs*. For readability, this example disregards the performance benefit of removing the array concatenations. To run it as a nested data-parallel program, the method *qs* spawns two separate processes to calculate *ls* and *hs*, which recursively spawn new processes by calling *qs*.

```
qs :: Ord a => [a] -> [a]
qs []           = []
qs (pivot : as) = let
  ls = qs [ a | a <- as, a <= pivot ]
  hs = qs [ a | a <- as, a >  pivot ]
  in ls ++ [pivot] ++ hs
```

Flat data parallelism Accelerate excludes nested data parallelism to ensure efficient execution on GPUs [23]. In contrast to the nested variant, flat data parallelism restricts the use of nested parallel instructions. In short, flat data parallelism is easier to implement, as it provides easier load balancing and cache-friendly, linear traversals of boxed arrays [19]. In contrast, nested data parallelism is more expressive to the programmer [7]. To bridge the gap between nested and flat data parallelism, Sabot and Blelloch [8, 9] introduced an approach named flattening. Flattening converts nested parallel instructions into flattened parallel instructions over multiple flat arrays. The method proved efficient on shared memory machines but performed poorly on distributed-memory machines [15]. With the intent to find an implementation technique that was efficient for such machines, Keller proposed to compile the nested data parallel instructions with a newly optimised set of flattening transformation instructions [20]. With flattening, each instruction of a program is mapped to a corresponding set of flat data parallel instructions. This process is also known as vectorisation. In a similar fashion, section 3 illustrates a vectorisation process where higher-order functions are flattened to a corresponding series of first-order operations. Finally, flattening increases the work complexity of an algorithm. Among others, Lippmeier et al. [22] and Keller et al. [19] introduced methods to improve the work complexity of flattening. For illustration, example 2 elaborates on a flattened version of the nested data-parallel QuickSort algorithm of example 1.

Example 2: To illustrate what a flat parallel program looks like, consider the following outline of a flat data-parallel quicksort program. The full implementation can be found on the Accelerate examples GitHub page [3]. In an overview, instead of recursively spawning new processes, the array is sorted by keeping track of element flags in a separate array. The flags indicate the start of segments in the array that should be sorted. Initially, the input array is the only segment. Then, while the array is not sorted, the algorithm runs in parallel on every segment and modifies the flags to indicate the new segments next to the pivot. In this example, there is no nested parallelism because it creates parallel processes only at the top level of each iteration. As a result, the definition is flat data-parallel.

1.2 Accelerate

Accelerate is a flat data parallel language for array computations embedded in Haskell [10, 23, 11, 24, 12]. Accelerate has backends to compile data-parallel array instructions into hardware-specific code. For instance, for multi-core CPUs, it uses LLVM [24], and for GPUs it uses CUDA [10]. Accelerate distinguishes between vanilla Haskell arrays and embedded arrays. The arrays are kept in the memory of the backend-specific device and are used to perform embedded calculations on the backend-specific hardware [10]. In Haskell, Accelerate's embedded array operations are represented by the type `Acc` [28]. Moreover, a term of type `Acc a` is an Accelerate program, which, once executed, will produce a value of type `a`. Specifically, each array instruction `Acc` consists of scalar operations that will be executed on the external hardware. Continuing from the Accelerate documentation [2], the type of allowed array elements is characterised by the `Elt` class, which roughly consists of signed and unsigned integers, floating point numbers, characters, booleans, shapes and tuples [2]. What's more, scalar operations are represented by the `Exp` type. The `Exp` type depicts collective array operations that are executed in parallel. Because Accelerate is a flat data-parallel language, scalar expressions can not spawn new operations. For illustration, example 3, illustrates an example of an embedded operation.

Example 3: Consider the `use` operation depicted below. The operation embeds a Haskell array on the external device. Correspondingly, the result is wrapped in the `Acc` type, and the type of elements is characterised by `Elt`.

```
use :: (Shape sh, Elt e) => Array sh e -> Acc (Array sh e)
```

The definition below provides an example use case of an embedded operation on an embedded array. Moreover, it applies the `map` instruction to execute function `f` to all elements of the array `vec`.

```
let vec = use $ fromList (Z:.5) [0..] :: Acc (Vector Int)
    f   = (\x -> (x + 1) * 2) :: Exp Int -> Exp Int
    in map f vec :: Acc (Vector Int)
```

Array representation In Accelerate, embedded arrays are kept in the memory of the backend-specific device [10]. In the Accelerate pipeline, an array is represented using a structure of arrays (SoA). SoA maps a data structure into one parallel array per field [18]. Instead of representing a set of objects in an array, SoA defines multiple arrays, where each array contains the values of a field for all objects in the set. When performing parallel instructions on a set of objects, object fields can be easily read and modified in parallel by addressing these field arrays. For Accelerate, an array is represented by a collection of pointers to the device memory. Each pointer points to a buffer in memory, which consists of an array of values. Buffers may isolate array dimensions, tuple fields, etc.

1.3 TensorFlow

TensorFlow is a machine learning library frequently used for deep learning. Deep learning is the process of training artificial neural networks. In TensorFlow, neural networks are represented as computational graphs that perform operations in parallel. A computational graph is a directed graph, with vertices to describe operations, and directed edges to represent the input and output data for operations. There are several potential advantages when running data-parallel programs with TensorFlow. For starters, the library allows its graphs to be run on GPUs. What's more, TensorFlow features common subgraph elimination, where repeated computations may be replaced by a single instance of computation [14]. With scheduling, node execution is scheduled as late as possible, such that the results of operations remain in memory for the least amount of time. Finally, TensorFlow programs can be accelerated with the TensorFlow Processing Unit (TPU) dedicated hardware. With such benefits, it could be interesting to run programs other than machine learning algorithms with TensorFlow. However, as TensorFlow programs are constructed as computational graphs, with the simplicity of available operations, writing a program may be time-consuming. This is where Accelerate could come in. Namely, Accelerate could compile parallel array instructions into TensorFlow graphs.

1.4 Research questions

As introduced above, this thesis adds a TensorFlow backend to Accelerate's new pipeline to embed its parallel array calculations in TensorFlow. The goal is to investigate the possible extent of running Accelerate programs in TensorFlow and to characterise limitations along the way. Among others, such limitations may be implied by Accelerate's compilation pipeline, or by unsupported functionality in TensorFlow. For instance, the Accelerate abstract syntax for backends is set up with the CPU and GPU libraries in mind [24]. And, perhaps already apparent from the second- and first-order differences in instructions, the new TensorFlow backend might fundamentally differ. As such, implementing a new compiler for TensorFlow may pose a challenge.

With the Accelerate language statically typed, the compilation pipeline preserves the types at every step. As a result, code generation in Accelerate backends is also statically typed. For non-statically typed languages, backends implement a custom statically typed compiler. For example, McDonell et al. [24] wrote a statically typed compiler for LLVM. As such, the goal is to investigate whether the new backend's logic fits inside the typed pipeline traversal. To limit the scope of this thesis, the emphasis is mostly on enabling, and not on performance. Yet, to give an impression, the results in section 4 also provide a benchmark that compares Accelerate and TensorFlow on a multi-core CPU. As further discussed in section 5, future research may look to improve on performance and find limitations. In summary, this research investigates the following research questions:

- Using the Accelerate pipeline, while preserving types at each step, do Accelerate GADTs allow sufficient manipulation to construct corresponding first-order computational graph operations?
- What is the difference in CPU performance between the new TensorFlow backend and the LLVM CPU backend?
- With Accelerate, is it possible to provide a friendly programming language to construct TensorFlow graphs?

2 Accelerate Pipeline

Accelerate supports embedded array calculations on a plethora of hardware. The compilation procedure for Accelerate programs is referred to as the pipeline. Each hardware-specific implementation for the pipeline is called a backend. Moreover, the pipeline to run Accelerate instructions with a hardware-specific backend consists of the following steps:

Desugaring First, the Accelerate instructions are desugared into hardware-specific operation types. Each Accelerate instruction is translated into a series of hardware-specific instructions that perform an embedded calculation with a final result equivalent to that of the Accelerate instruction. Section 2.2 provides a detailed overview of the requirements from the pipeline to implement this step.

Fusion Fusion aims to remove the overhead of programming by combining transformations on data structures to remove intermediate structures [12]. Some backends, such as LLVM [24], implement fusion for optimisation purposes. However, to limit the scope of this research, fusion will not be implemented for the TensorFlow backend.

Scheduling Subsequently, the operation types are scheduled for execution. Before running the schedule, the pipeline maps the operations to hardware bindings in Haskell.

Execution Finally, the bindings are executed in the order of the schedule. Listing 1 shows the method `run`, which executes Accelerate instructions of type `Acc` on the backend. The backend is characterised by the type-class constraint `Backend`, which requires an implementation for each of the pipeline steps. The sections below elaborate on each of the compilation steps. Section 3 depicts the steps in the pipeline, and elaborates how the steps above are implemented for the TensorFlow backend.

```
run :: forall backend t. (Arrays t, Backend backend) => Acc t -> t
```

Listing 1: The run operation, which compiles and executes Accelerate instructions on a hardware-specific backend.

2.1 Preliminaries

To provide a complete definition of the compilation process of the Accelerate pipeline, this section provides an overview of the foundations. For the remainder of this thesis, these notions are referenced when used in an explanation.

2.1.1 GADTs

Accelerate programs are typed with Haskell's Generalised Algebraic Data Structures (GADTs). GADTs in Haskell allow type parameters in syntax trees. For instance, in Accelerate, using GADTs, scalar expressions have type `Exp t`, which is parameterised over the return type `t`. As a result, the return type can be verified during compilation, which allows Accelerate components to restrict the type of expressions they accept. For example, the `filter` operation in Accelerate applies a predicate function to each element in an array. It drops elements that do not satisfy the predicate and returns the elements which pass the predicate, together with a description of how many elements were valid [2]. The predicate is given by a function parameter. The function takes in an element and returns a boolean to indicate whether the element satisfies the predicate or not. Using GADTs, Haskell's strict typing system restricts the user to define a function with an array element as input, and a boolean as output. Correspondingly, `filter` below restricts the predicate to be of type `(Exp e -> Exp Bool)`.

With Haskell's type system, McDonell et al. [24] argue that GADTs offer high confidence in compiler correctness. Their work addresses how to preserve static-type information throughout the pipeline, and how it decreases the cost of constructing and maintaining backends. GADTs allow for maintaining type information throughout the entire compilation pipeline. This enables type information to be transferred to type-safe code generation interfaces, such as the LLVM interface by McDonell et al.

2.1.2 Shapes

In Accelerate, arrays have an arbitrary amount of arbitrarily sized dimensions. An array's ordered collection of dimensions is also referred to as its shape. Listing 2 depicts the definition of the `ShapeR` GADT. Notably, the GADT does not contain shape values. As such, `ShapeR` represents the type of the shape of an array instead of the shape itself. The actual shape values are stored in the backend-specific device memory. The shape representation is often passed as a parameter to pattern match the shape representation. This can be convenient when writing recursive functions over an array, and accessing pointers on device memory to retrieve the actual shape values.

```
data ShapeR sh where
  ShapeRz     :: ShapeR ()
  ShapeRsnoc :: ShapeR sh -> ShapeR (sh, Int)
```

Listing 2: Using nested pairs, the `TupR` GADT represents a collection of elements of type `a` constructed by `s`.

Example 4: Listing 2 depicts a recursive definition to characterise an array shape. In general, similar recursive definitions can be used to represent any collection of elements. For instance, consider the `List` GADT below. Its minimal definition consists of the empty list (`Nil`). In Haskell, a list can be constructed using the `(:)` operator. Correspondingly, the `Cons` constructor adds one element to the left side of an existing list. Lists in Haskell are homogeneous. In other words, all elements of a list are of the same type. Correspondingly, the GADT below has a type parameter `a`, which defines the type of each element in the list. With the power of GADTs, heterogeneous lists may also be constructed. However, to illustrate the syntax of GADT, the definition below matches the definition of regular Haskell lists.

```
data List a where
  Nil  ::                List a
  Cons :: a -> List a -> List a
```

2.1.3 Tuples

Throughout the Accelerate pipeline, most GADT collections are represented in terms of nested pairs, also known as tuples. Listing 3 depicts the corresponding `TupR` GADT with elements of the higher order type `s a`. Alternatively, example 4 depicts a list definition to represent a collection. `TupR` differs from the regular list construction by separating the nullary and unary tuple constructors. Moreover, in a tuple, a single element is constructed without constructing `Nil` first. For illustration, `(a, ())` does not represent the unary tuple of `(a)`, but represents the binary tuple of `a` and `Nil`. Furthermore, in a list, the first field of `Cons` is a value, while the first field of a pair is a tuple. Tuples are used in most of the data structures in the rest of the pipeline. For instance, as elaborated by section 2.2, an `Acc` expression consists of pairs of expressions and void constructors.

```
data TupR s a where
  TupRunit    ::                              TupR s ()
  TupRsingle :: s a                        -> TupR s a
  TupRpair    :: TupR s a -> TupR s b -> TupR s (a, b)
```

Listing 3: Using nested pairs, the `TupR` GADT represents a collection of elements of type `a` constructed by `s`.

Tuples in Accelerate differ from Haskell's notion of tuples. For starters, Haskell allows elements in tuples to have different types. In contrast, all elements in `TupR s a` are of the same type `s a`. What's more, tuples in Haskell are of a fixed size. That is, elements can not be added with a constructor similar to `Cons` of example 4. In contrast, `TupR` can include more elements by nesting another pair with the `TupRpair` constructor.

2.1.4 De Bruijn indices

In the Accelerate pipeline, variables are referred to by their De Bruijn indices. For starters, De Bruijn indices are integers which represent terms of the lambda calculus [13]. As elaborated in example 5, the integer value of a De Bruijn index represents the number of binders between the use and declaration of a term. A benefit of using the De Bruijn index representation is the simplicity of preserving term uniqueness in an environment. When introducing a new term, its corresponding De Bruijn index is equal to the number of previously declared terms. De Bruijn indices are not limited to lambda expressions and can index any kind of recursively constructed data type.

Example 5: De Bruijn indices represent the number of binders between the use and declaration of a term in a lambda statement. Starting from the last term in a lambda instruction, or the leaf of the corresponding higher-order abstract syntax, the most inner term is set to 0, and each succeeding shadowing declaration is equal to its most recent inner term added by 1. For illustration, the expression below shows a lambda term converted to De Bruijn indices.

$$\lambda x.\lambda y.\lambda z.\ x\ \ z\ \ (y\ \ z) \rightsquigarrow \lambda\lambda\lambda\ \ 2\ \ 0\ \ (1\ \ 0)$$

2.1.5 Environments

In the Accelerate pipeline, variables are kept in an environment data structure. As depicted in listing 4, environments are represented by the `Env` GADT. The constructors use a similar recursive construction as the `List` GADT in example 4. The main difference is the result types of the constructors. Moreover, each constructor `List` has type `List a`. In contrast, the constructors of `Env` also denote the shape and element types of an environment. As a result, the type checker can distinguish one environment from another.

```
data Env f env where
  Empty :: Env f ()
  Push  :: Env f env -> f t -> Env f (env, t)
```

Listing 4: The constructors of the `Env` GADT, which recursively construct an environment.

Environment variables are represented by De Bruijn indices (section 2.1.4). The `Idx` GADT of listing 5 represents an index of an element in an environment that corresponds to the nested position of an element in `Env`. This GADT is used to reference a variable in an environment. As depicted by listing 4, variables are stored in an environment using a recursive definition. The number of recursive layers of a constructed `Idx` corresponds to the position of an element in an environment. As such, the `prj'` method from example 6 uses the `Idx` GADT to retrieve a value from an environment. As elaborated in the example, the main benefit of the `Env` GADT is that it allows other GADTs to require terms to be parameterised over the same environment. Correspondingly, `Idx` is parameterised over the type of environment (`env`) it points to.

```
data Idx env t where
  ZeroIdx ::                Idx (env, t) t
  SuccIdx :: Idx env t -> Idx (env, s) t
```

Listing 5: The constructors of the `Idx` GADT.

Example 6: The method `prj'` below returns the element of an environment that an index points to. Both the index (`Idx env t`) and environment (`Env f env`) parameters are parameterised over the same environment type (`env`). This ensures that the index is never out of bounds, and the element type is `f t`.

```
prj' :: Idx env t -> Env f env -> f t
prj' ZeroIdx       (Push _   v) = v
prj' (SuccIdx idx) (Push env _) = prj' idx env
```

Examples of variables As depicted in listing 6, environment variables are represented by the `Var` and `Vars` types. For example, consider the `ExpVar` in listing 2.1.5. The type denotes variables that point to scalar expressions. This construction allows referring to expressions inside of other expressions in the syntax tree.

```
data Var  s env t = Var { varType :: s t, varIdx :: Idx env t }
type Vars s env   = TupR (Var s env)
```

Listing 6: The `Var` GADT represents an environment variable. The variable consists of a De Bruijn index which points to a nested position of an element in the environment.

```
type ExpVar      = Var ScalarType
type ExpVars env = Vars ScalarType env
```

Listing 7: The `ExpVar` and `ExpVars` types represent variables that point to scalar expressions.

For another example, consider listing 8. The pipeline performs embedded executions by referring to memory on a backend-specific device. The `GroundVar` and `GroundVars` types refer to one or more pointers to the device memory. What's more, the variables contain a `GroundR` representation to indicate whether the allocated memory contains a scalar value or a buffer.

```
type GroundVar      = Var  GroundR
type GroundVars env = Vars GroundR env
data GroundR a where
  GroundRbuffer :: ScalarType e -> GroundR (Buffer e)
  GroundRscalar :: ScalarType e -> GroundR e
```

Listing 8: The `GroundR` GADT indicates the type of a value stored on device memory. The `GroundVar` and `GroundVars` types represent variables that point to values on device memory.

In the pipeline, variables are often introduced by function arguments. Consider the definition of `ELeftHandSide` in listing 9 below. The GADT represents function arguments and is parameterised over two environment types. Namely, the original environment (`env`), and the new environment after declaring the bindings (`env'`). Specifically, for single terms (`LeftHandSideSingle`), the new environment consists of the original environment and the new term variable. For wildcard variables (`LeftHandSideWildcard`) such as `()`, the environment remains the same. Finally, when introducing a tuple of terms (`LeftHandSidePair`), the new environment consists of the original environment and all of the variables inside of the tuple.

```
type ELeftHandSide = LeftHandSide ScalarType
data LeftHandSide s v env env' where
  LeftHandSideSingle
    :: s v
    -> LeftHandSide s v env (env, v)
  LeftHandSideWildcard
    :: TupR s v
    -> LeftHandSide s v env env
  LeftHandSidePair
    :: LeftHandSide s v1       env  env'
    -> LeftHandSide s v2       env' env''
    -> LeftHandSide s (v1, v2) env  env''
```

Listing 9: The `LeftHandSide` GADT represents function arguments. It features single values, wildcards and nested types for tuples.

2.1.6 Types

In Accelerate, array elements are scalars or tuples of scalars. Scalars are signed integers, unsigned integers, or floating-point numbers. Correspondingly, with the definition of `TupR` above, in the pipeline, array element types are represented with `TupR ScalarType`. The `ScalarType` GADT denotes the kind of scalars that are supported in Accelerate. Next to scalar types, the GADT features a constructor `VectorType` for fixed-size vectors. Moreover, in Accelerate, array elements may consist of vectors. For illustration, a vector may be used to store complex numbers such as coordinates. To limit the scope of this research, the vector types are not covered.

2.2 Desugaring

The pipeline desugars the Accelerate program into a series of corresponding backend-specific operations. A large part of the contribution of this thesis consists of converting Accelerate programs into TensorFlow. The conversion is implemented in the desugaring step of the pipeline. To provide a complete definition, this section elaborates on how the desugaring class of the pipeline is defined. Among others, the class depicts flow control, such as device operation execution, allocation, conditions, loops, and more. As depicted by listing 10 below, the set of operations is comprised by the `PreOpenAcc` GADT. In an overview, the set consists of the following control-flow instructions and execution instructions.

· Each kind of external backend features a unique set of hardware-specific computations. Correspondingly, the `PreOpenAcc` GADT needs an operation set type parameter `op`. This allows the type checker to limit execution statements to be constructed with the backend-specific operation set. Because `OperationAcc` denotes backend-specific instructions, `op` differs for each backend implementation. The `Exec` constructor represents the execution instruction to perform an operation on the backend-specific device. Correspondingly, the first parameter of `Exec` denotes such a parameterised operation. For the new TensorFlow backend, section 3.2.2 elaborates how the TensorFlow operations are defined for `op`.

· Operations are performed on the backend-specific device, such as a GPU. The results of device operations are stored in device memory. As depicted in section 2.1.5, the `GroundVars` type represents variables that point to values in this device memory. With `GroundVars` as its argument, the `Return` constructor represents the retrieval of a value from the device memory.

· The `Compute` constructor represents the evaluation of a scalar expression `Exp` with the default interpreter. Such computation may involve shape-size computation or conditions of if-statements. These computations do not have to be embedded.

· Similar to regular `let` construction in Haskell, the `Alet` constructor represents a local binding of a flow to a variable. The first argument `GLeftHandside` indicates the shape of the variable, the second argument represents the statement to assign, and the third argument represents the remainder of the flow that has the newly declared variable in its scope. Correspondingly, the remainder of the flow is parameterised over the new environment which includes the variable.

· Desugarring might require the allocation of new memory. The `Alloc` constructor represents the allocation of a new array and includes metadata such as the type of array elements, and variables that contain its shape information.

· The `Use` constructor fetches data from a buffer in device memory. For instance, it can be used to wrap an array from a previously run statement inside of a new `Acc` statement.

· To capture a single value in an array representation, the `Unit` constructor represents the creation of a buffer with a single element.

· Program execution often depends on the result of calculations. The `Acond` constructor represents an if-then-else statement based on a variable that points to a boolean.

· To enable loops, the `Awhile` constructor represents a while loop. Moreover, it performs a function if its condition is true, and re-evaluates until the condition becomes false.

The desugarring implementation provides a PreOpenAcc result for every Acc constructor. The process is implemented in the pipeline with the DesugarAcc class. The set of Accelerate instructions is extensive. This enhances the expressiveness of the language to the user. However, it introduces a considerable amount of required implementations in the DesugarAcc class. Fortunately, the DesugarAcc class provides default implementations for all Acc expressions in terms of `generate` and `permute`. The main benefit is that implementing desugaring with DesugarAcc only requires providing an implementation for `mkGenerate` and `mkPermute`. However, it may be more performant (or closer to the characteristics of the backend) to overwrite some of the default implementations. Section 3 further elaborates the definitions of `generate` and `permute`, and how they are desugared into TensorFlow graphs. After desugaring, the instructions are scheduled for execution. Section 3 elaborates on how each step is implemented for the new TensorFlow backend.

```haskell
data PreOpenAcc (op :: Type -> Type) env a where
-- Executes an executable operation.
  Exec    :: op args
          -> Args env args
          -> PreOpenAcc op env ()
  -- Returns the values of variables.
  Return  :: GroundVars env a
          -> PreOpenAcc op env a
  -- Return evaluated expression.
  Compute :: Exp env t
          -> PreOpenAcc op env t
  -- Local binding.
  Alet    :: GLeftHandSide bnd env env'
          -> Uniquenesses bnd
          -> PreOpenAcc op env  bnd
          -> PreOpenAcc op env' t
          -> PreOpenAcc op env  t
  -- Allocates a new buffer.
  Alloc   :: ShapeR sh
          -> ScalarType e
          -> ExpVars env sh
          -> PreOpenAcc op env (Buffer e)
  -- Buffer inlet
  Use     :: ScalarType e
          -> Int -- Number of elements
          -> Buffer e
          -> PreOpenAcc op env (Buffer e)
  -- Capture a scalar in a singleton buffer
  Unit    :: ExpVar env e
          -> PreOpenAcc op env (Buffer e)
  -- If-then-else
  Acond   :: ExpVar env PrimBool
          -> PreOpenAcc op env a
          -> PreOpenAcc op env a
          -> PreOpenAcc op env a
  -- Value-recursion
  Awhile  :: Uniquenesses a
          -> PreOpenAfun op env (a -> PrimBool)
          -> PreOpenAfun op env (a -> a)
          -> GroundVars     env a
          -> PreOpenAcc  op env a
```

Listing 10: The constructors for backend instructions. The `op` type parameter specifies the backend-specific instructions.

2.2.1 Arguments

The signature of a desugaring method in DesugarAcc contains the input arguments of the corresponding Acc instruction. Listing 11 depicts the constructors of two kinds of argument representations. As discussed in the approach, only these two kinds of arguments will be encountered in the desugaring process of the new TensorFlow pipeline. To provide further background, the paragraphs below elaborate on the constructor types. Finally, example 7 illustrates how the arguments are used to represent the input and output of `map` expression in Accelerate.

```
data Arg env t where
  ArgFun   :: Fun env e     -> Arg env (Fun' e)
  ArgArray :: Modifier m
           -> ArrayR (Array sh e)
           -> GroundVars env sh
           -> GroundVars env (Buffers e)
           -> Arg env (m sh e)
```

Listing 11: The function and array argument constructors of the Arg GADT. When desugaring, the data type represents the parameters of an Acc instruction.

Functions Accelerate supports second-order combinators with scalar expressions as arguments. As such, the ArgFun constructor depicted in listing 11 has a single argument of type Fun that corresponds to the function argument. Listing 12 depicts the function data structure. The Fun type shorthands the environment parameterisation of PreOpenFun with an expression environment and a buffer environment. The PreOpenFun GADT follows a recursive definition similar to a lambda expression. Moreover, it is constructed with a recursive series of input terms (Lam), followed by the function body, which consists of a scalar expression tree PreOpenExp. The scalar expressions correspond to the scalar expressions Exp in Accelerate [2].

```
type OpenFun env benv = PreOpenFun (ArrayInstr benv) env
type Fun benv = OpenFun () benv

data PreOpenFun arr env t where
  Body :: PreOpenExp arr env  t
       -> PreOpenFun arr env t
  Lam  :: ELeftHandSide a env env'
       -> PreOpenFun arr env' t
       -> PreOpenFun arr env (a -> t)
```

Listing 12: The PreOpenFun GADT comprises a recursive function definition.

The `arr` type parameter of PreOpenFun parameterises the type of array instructions. In the case of OpenFun, the type is represented with `ArrayInstr benv`. Consider the listing below, with the variable types GroundVar and ExpVar from section 2.1.5, the ArrayInstr GADT corresponds to two read operations in external memory. First, the Index operation consists of a variable pointing to a buffer. Its constructed type `ArrayInstr benv (Int -> e)` corresponds to the index operation. Given an integer buffer index, it returns an element from the buffer at the given position. Second, the Parameter operation retrieves a scalar value from the Accelerate program.

```
data ArrayInstr benv t where
  Index     :: GroundVar benv (Buffer e) -> ArrayInstr benv (Int -> e)
  Parameter :: ExpVar benv e -> ArrayInstr benv (() -> e)
```

Listing 13: The ArrayInstr GADT represents the two array indexing operations Index and Parameter.

As elaborated in section 2.1.5, Accelerate GADTs may be parameterised over an environment. For instance, as depicted by listing 12, the OpenFun type shorthands the parameterisation of PreOpenFun with a buffer environment (benv) and an expression environment (env). The expression environment env consists of previously declared expressions. And, the buffer environment includes the scope of previously declared buffers and scalars stored on the external hardware. A function argument can not refer to unbound scalar variables. Therefore, Fun initialises the expression environment env as empty (). Function input terms from the Lam constructor of PreOpenFun are represented by the ELeftHandSize GADT from section 2.1.5.

Scalar expressions As depicted by listing 12, the body of a function argument consists of a scalar expression tree of type PreOpenExp. The type represents embedded scalar expressions. The full set of scalar expression constructors is extensive. The full definition can be found in the Accelerate Haskell documentation [2]. In section 3.2.2, the constructors relevant to the approach of this thesis will be further investigated. Some constructors are recursive. To give an example, a let binding stores an expression in a variable with the Let constructor. The first argument declares the expression to assign to the variable. Then, the second expression of the Let constructor can reference that variable using the Evar constructor. As elaborated in section 1.1, Accelerate excludes flat data parallelism. Accordingly, although the GADT has a recursive structure, PreOpenExp does not spawn new parallel processes.

Arrays The ArgArray constructor depicted in listing 11 is used whenever an array is used as an input or output of an Acc expression. To correctly interact with the backend-specific device memory depicted by its structure of arrays (SoA), the ArgArray constructor features metadata and pointers to the backend-specific device memory. For starters, as depicted by listing 14, the first argument Modifier m specifies whether the memory is read-only, write-only, or neither.

```
data Modifier m where
  -- | Read-only
  In  :: Modifier In
  -- | Write-only
  Out :: Modifier Out
  -- | Read & Write
  Mut :: Modifier Mut
```

Listing 14: The Modifier GADT depicts the permitted actions with external memory.

As depicted in listing 15, the second argument ArrayR depicts the shape of the array with the ShapeR GADT (section 2.1.2) and the type of all array elements with the TypeR GADT (section 2.1.6). The final two arguments are of type GroundVars. As depicted in listing 8, GroundVars is a type synonym of Vars GroundR (section 2.1.5). Respectively for GroundRbuffer and GroundRscalar, the two constructors indicate whether a variable points to a scalar or buffer in the backend-specific device memory. Moreover, the third argument of ArgArray refers to shape variables corresponding to the size of each dimension of the array shape. Finally, the fourth argument points to the buffers that are part of the SoA.

```
data ArrayR a where
ArrayR :: { arrayRshape :: ShapeR sh
          , arrayRtype  :: TypeR e }
       -> ArrayR (Array sh e)
```

Listing 15: The ArrayR GADT provides metadata about the array. The arrayRshape record specifies the array shape, and the arrayRtype record the element type.

Example 7: The method `mkMap` of `DesugarAcc` implements desugaring `map` statements. With the constructors of the `Arg` GADT shown in figure 11, the first two arguments of `mkMap` consist of a function argument and an array argument of type `Fun' (s -> t)` and `In sh s`. Correspondingly, as shown in example 3, `map` takes a method and an array as its two arguments.

Accelerate performs calculations on a backend-specific device. The result is stored in the memory of the device. The final argument keeps track of where the resulting array of type `Out sh t` is stored after the desugared `map` operation is performed. Correspondingly, the output array of shape `sh` with elements of type `t` is the result of applying a method of type `s -> t` to all elements of type `s` of an array of shape `sh`.

```
class DesugarAcc (op :: Type -> Type) where
      mkMap :: Arg env (Fun' (s -> t))
            -> Arg env (In sh s)
            -> Arg env (Out sh t)
            -> OperationAcc op env ()
```

3 The TensorFlow Backend

To investigate the possible extent of adding TensorFlow support for Accelerate, this thesis introduces a new TensorFlow-specific backend for the compilation pipeline of Accelerate. In an overview, the backend desugars the Accelerate programs into a series of first-order TensorFlow operations and constructs corresponding TensorFlow graphs to execute the desugaring result. Section 3.1 elaborates on the differences between Accelerate and TensorFlow, and models how Accelerate programs can be converted into TensorFlow graphs and executed on a TensorFlow device. Section 3.2 depicts a detailed overview of how this model is implemented in the GADTs corresponding to the steps of the pipeline.

3.1 Model

The main idea of this thesis is to compile Accelerate programs into TensorFlow graphs and execute them in TensorFlow. In TensorFlow, the output of a graph node consists of a tensor of elements. Graph nodes have tensors as their input [1]. Tensors are arrays of an arbitrary dimension, where all elements are of the same type. Similarly, arrays in Accelerate are also of an arbitrary dimension where the elements are of a single type (section 2.1.6). As such, arrays in Accelerate can be converted to tensors of the computational graph.

3.1.1 Desugaring Accelerate programs into TensorFlow instructions

As elaborated in section 2.2, an Accelerate program is represented by a recursively constructed GADT of Accelerate components. The recursive definition connects Accelerate components by using other components as constructor arguments. As such, the GADT embodies the abstract syntax tree of the language. In TensorFlow, each graph consists of an operation node with the output of sub-graphs as input [1]. As such, similar to Accelerate, the abstract syntax tree for TensorFlow graphs consists of a recursive definition. As discussed below, the key difference is that Accelerate components can be parameterised over scalar expressions, whereas TensorFlow components are all specialised and only take in tensors as input. In other words, TensorFlow components are first-order, and Accelerate components are second-order. Therefore, when desugaring Accelerate programs into TensorFlow graphs, the higher-order methods should be compiled into a sequence of first-order computations. As elaborated in section 3.2, higher-order instructions can be flattened by reading through the provided function in the parameter and traversing its instructions from top to bottom to construct a corresponding series of computations. Similarly, as discussed in section 1.1, when flattening nested-data parallel programs, Keller eliminated higher-order operations in a similar fashion [20] by providing a transformation rule for each component of a program.

The set of Accelerate instructions is extensive. This enhances the expressiveness of the language to the user but introduces a considerable amount of desugaring in the pipeline. However, as elaborated in section 2.2, the pipeline already desugars all instructions in terms of the operations `permute` and generate by default. As a result, the new pipeline only has to provide a desugaring implementation for permute and generate. Both Accelerate components have at least one function as a parameter. Moreover, as further discussed in section 3, generate takes in a function that should be applied to every index of an element, and permute has its combination and permutation function parameters. Hence, they are higher-order components. The TensorFlow instruction set only contains first-order instructions. Therefore, to implement generate and permute, a total of three function parameters need to be flattened. Although the three function parameters have different signatures, their bodies comprise Accelerate's scalar expressions of type PreOpenExp. To solve the flattening problem, a generic function is defined that converts a recursively constructed scalar expression into a corresponding series of first-order TensorFlow operations. What's more, as elaborated in example 7, Accelerate's map function applies a function parameter to an input array and writes the result to an output array. To execute the map instruction with first-order TensorFlow instructions, each expression in the function parameter body must be desugared into TensorFlow instructions. As such, the method perfectly isolates the flattening problem. The result is a series of TensorFlow instructions with a final result equivalent to the passed function parameter. As elaborated in section 3.2, the desugaring implementations for generate and permute use map to flatten their function parameters.

3.1.2 Executing desugared programs in TensorFlow

After desugaring, the next step in the Accelerate pipeline is to schedule the set of desugared instructions for execution. Subsequently, the scheduled instructions are executed on the backend-specific device. In the pipeline, schedules may fork parallel work instructions, specifically optimised for the kind of backend. For instance, the LLVM backend implements a dynamically work-stealing scheduler [24]. Yet, the main goal of this thesis is to investigate the possibility of translating Accelerate to TensorFlow. Therefore, to limit the scope, the TensorFlow backend uses a sequential scheduler, which schedules the desired instructions into a sequential flow. The scheduled instructions are performed one by one, but can still depict the execution of data-parallel operations in TensorFlow. To reduce a bit of the schedule size, the scheduler does eliminate executions of the identity operation. After scheduling, the calculations are performed.

Haskell TensorFlow bindings Backends in Accelerate generate code which performs calculations on external hardware. Accordingly, to implement a new TensorFlow backend, the pipeline must use external bindings to perform TensorFlow calculations. Many programming languages have an Application Programming Interface (API) to construct and execute TensorFlow graphs [6]. For Haskell, the TensorFlow Haskell library [29] provides bindings to output TensorFlow graphs in Haskell. Using the API for a different programming language would enlarge the scope of this thesis from generating TensorFlow graphs to generating code in a different language in general. Therefore, the use of these bindings is the best option for this research. In this thesis, types and operations of the bindings are prefixed by TF. to distinguish them from other Haskell statements.

The Haskell TensorFlow bindings library is not maintained by the TensorFlow developers. Instead, it is one of the community-supported languages, which are open-source projects. Though each language follows the recommended approach by the TensorFlow maintainers [30], the library may not always be up to date. For instance, before January 2023, the bindings only supported TensorFlow version 2.3.0. A disadvantage of this version was the lack of Nvidia's ampere architecture support, which was added in version v2.4.0 [27]. Consequently, the programs could not be run on Nvidia's 3000 series. At the time of writing, the issue has only recently been resolved [26]. Still, the low frequency of maintenance might be something to remain in consideration when dealing with future issues. Finally, the binding documentation [31, 32] depicts the signatures available methods, with almost no descriptions. Fortunately, the method names resemble the names of the Python API documentation [25], which does contain a detailed description of almost every method that is also included in the Haskell bindings.

Graph construction In TensorFlow, calculations are performed by running TensorFlow graphs. With the TensorFlow bindings for Haskell, tensor operations take in graphs as their arguments, and return a graph of the tensor operation constructed with the graphs as input [29]. To illustrate, the addition operation takes in two graphs and returns a graph that introduces an addition operation node with both graphs as input. The pipeline schedule contains execution instructions for TensorFlow operations. Such an instruction does not return a value. Instead, it denotes a side effect that depicts which buffers are affected by running the operation. When encountering an execution instruction in a schedule, a simple approach would be to convert the inputs into tensors, construct a graph with the operation, run the graph, and store the value of the returned tensor in the output buffer. However, this introduces the overhead of converting values from and to tensors for every TensorFlow operation in a desugared Accelerate program. To reduce this overhead, this thesis introduces a different approach which combines many scheduled operations into a single graph. It connects as many operations as possible up until evaluation (running the graph) is absolutely necessary. That is, as further elaborated below, until the scheduled flow is dependent on an output of an operation. For instance, a while loop flow component may be dependent on a boolean value that is returned by an operation.

Memory allocation As elaborated above, in the Haskell bindings for TensorFlow, a tensor operation produces a graph. A tensor of type `TF.Build` refers to a tensor graph, of which the operation of the root node yields that tensor. Because a tensor operation produces a graph, the return type is also a tensor of type `TF.Build`. To obtain its value, the tensor graph can be executed with the `TF.run` operation, which returns all elements from the output tensor of the graph. TensorFlow instructions in the schedule denote a side effect. The side effect indicates which buffers are affected by the operation. Buffers are represented by a pointer to memory on the host device. Memory is represented differently for each backend. As such, each backend should implement memory handling by itself. For this reason, the schedule also contains instructions for memory allocation. To continue, as each backend should implement memory handling by itself, a buffer in the schedule only consists of a De Bruijn index. The approach is to combine tensor operations into one graph until the graph value is required to continue the flow execution. To combine operations from the TensorFlow bindings, the De Bruijn indices should point to graphs in memory. To access the graphs in memory with the De Bruijn indices, an index-to-memory environment is maintained. Each time a new buffer is created or modified, the schedule provides the corresponding De Bruijn index with the input and output buffers. Using the new environment, these indices can then be used to store and look up graphs in memory. To elaborate, for each schedule component, section 3.2 defines the implementation of schedule execution and memory management.

3.2 Implementation

Section 3.1 discusses a detailed model for the implementation approach of the new TensorFlow backend for the Accelerate pipeline. To provide a complete definition of the new backend, this section continues by describing the code implementation for each step of the pipeline. Moreover, section 3.2.2 discusses the implementation for desugaring. And, section 3.2.2 discusses the execution of TensorFlow operations.

3.2.1 Desugaring

In summary of sections 2.2 and 3.1, a backend must provide definitions for the `DesugarAcc` class to implement the desugaring step in the pipeline. The class requires an implementation for every `Acc` statement. It provides a default implementation for all statements in terms of `generate` and `permute`. As such, the implementation of the desugaring step only requires a definition for corresponding methods `mkGenerate` and `mkPermute` in DesugarAcc. Still, both are higher-order methods of which their function parameters must be flattened into a corresponding series of TensorFlow instructions. As elaborated above, by implementing the flattening method for `map`, it can be re-used as a generic flattening method for `mkGenerate` and `mkPermute`.

TensorFlow parameterisation The `op` type parameter of `PreOpenAcc` defines the set of operations that can be executed on the backend-specific device. Correspondingly, the execution constructor `Exec` in listing 10 requires an instance of `op` as its first argument. To parameterise the desugaring result over the set of TensorFlow instructions, a new GADT `TensorOp` is constructed. As elaborated in example 8, the constructors of the `TensorOp` GADT correspond to TensorFlow-specific operations.

Flattening example As elaborated above, higher-order functions must be flattened into a corresponding series of first-order operations. Listing 17 shows a manually constructed definition to flatten the statement `map (\ x -> x * 2 + 1)` into a corresponding series of `PreOpenAcc` with `TensorOp`. The signature of the flattening process is depicted by listing 16. The signature contains two integer arrays. The first array `(In sh Int64)` contains the input buffer to apply the method `(\ x -> x * 2 + 1)` to. The second array `(Out sh Int64)` is the output buffer to which the calculation result should be written.

Example 8: While desugaring, the TensorOp GADT depicts the set of TensorFlow operations that should be executed on the external hardware. For example, the TAdd constructor listed below represents the TF.add operation in TensorFlow. The type of each constructor is parameterised over its side effect type. That is, the affected buffers by executing the calculation on external hardware. For instance, TAdd has type TensorOp (In sh a -> In sh a -> Out sh a -> ()). In detail, given three buffers of the same shape and element type, the operation reads input from the first two buffers and outputs the result into the third. Finally, the TAdd constructor contains the OneOf TFNum a type constraint. This requires the type a to be compatible with the corresponding TensorFlow binding.

```
data TensorOp op where
  TAdd  :: OneOf TFNum  a => TensorOp (In sh a -> In sh a -> Out sh a -> ())
```

For illustration, the definition below depicts a construction of PreOpenAcc to perform the addition operation in TensorFlow. The three buffers are provided through the function arguments. The Exec constructor performs operations on the host device. To perform addition in TensorFlow, the definition constructs Exec with TAdd.

```
add :: forall env sh.
       Arg env (In sh Int64)
    -> Arg env (In sh Int64)
    -> Arg env (Out sh Int64)
    -> OperationAcc TensorOp env ()
add argIn1 argIn2 argOut = Exec TAdd (argIn1 :>: argIn2 :>: argOut :>: ArgsNil)
```

The function does not modify the shape of the input. Therefore, the shape sh of the input and output are equal.

```
mapXTimesTwoPlusOne :: forall env sh. Arg env (In sh Int64)
  -> Arg env (Out sh Int64) -> OperationAcc TensorOp env ()
```

Listing 16: The signature of the flattening of `map (\ x -> x * 2 + 1)`. The arguments consist of an input and output array.

Flattening disassembles the function parameter into a corresponding series of TensorFlow operations. The function mapped over arrays `(\ x -> x * 2 + 1)` can be desugared into the two TensorFlow operations of multiplication and addition. The operations can be interpreted as "mutiply with a tensor of twos" and "add to a tensor of ones". Therefore, for the resulting OperationAcc structure, an array of twos and an array of ones will be allocated. Example 8 elaborates how TAdd can be used to desugar tensor addition. For multiplication, TMul constructs the operation in a similar fashion.

As depicted in listing 17, to refer to the arrays in the constructed flow, two new variables are introduced. Allocating new arrays means declaring new terms, which should be included in the environment. Therefore, each time an existing environment is referenced, it is updated with the newly declared variables. This process is also known as weakening. For illustration, consider the execution of multiplication Exec TMul in listing 17. After assigning a new array of twos to a variable lhs, both environments of the input arrays must include the new variable. Accordingly, their environments are extended to include the variable by weakening them with `weakenVars w`. Finally, to include both the first and second variables in the final steps, the weakening operators are combined with `(w' .> w)`.

Automated flattening The example in listing 17 shows a manual construction of flattening for `map (\ x -> x * 2 + 1)`. The new backend should apply this process automatically for any function parameter. By traversing the body of a function, it should output a structure similar to the example. As

```
mapXTimesTwoPlusOne :: forall env sh. Arg env (In sh Int64)
  -> Arg env (Out sh Int64) -> OperationAcc TensorOp env ()
mapXTimesTwoPlusOne
  (ArgArray _ arrayR@(ArrayR _ t) gvIn gvbIn)
  argOut@(ArgArray _ _ _ gvbOut)
  | DeclareVars lhs  w  k  <- declareVars $ buffersR t -- variable to array of twos
  , DeclareVars lhs' w' k' <- declareVars $ buffersR t -- variable to array of ones
  = let sInt64 :: ScalarType Int64
        sInt64 = SingleScalarType (NumSingleType (IntegralNumType TypeInt64)) in
      aletUnique lhs (desugarAlloc arrayR (fromGrounds gvIn)) $ -- Allocate new array
      Alet (LeftHandSideWildcard TupRunit) TupRunit
        (Exec -- Fill new array with the number 2
          (TConstant sInt64 2) -- TensorOp constructor, creates a new Tensor of twos.
          (ArgArray Out arrayR (weakenVars w gvIn) (k weakenId) :>: ArgsNil)) $
        Alet (LeftHandSideWildcard TupRunit) TupRunit -- Concat flow instruction (;).
          (Exec -- (*2) Multiply input array with new array, store in output array.
            TMul (ArgArray In arrayR (weakenVars w gvIn) (weakenVars w gvbIn) :>:
                  ArgArray In arrayR (weakenVars w gvIn) (k weakenId) :>:
                  weaken w argOut :>:
                  ArgsNil)) $
          aletUnique lhs' (desugarAlloc arrayR (fromGrounds (weakenVars w gvIn))) $
            Alet (LeftHandSideWildcard TupRunit) TupRunit
              (Exec -- Fill new array with the number 1
                (TConstant sInt64 1)
                (ArgArray Out arrayR (weakenVars (w' .> w) gvIn) (k' weakenId) :>:
                 ArgsNil)) $
             Exec -- (+1) Add new array to the result array of (*2)
               TAdd
               (ArgArray In arrayR (weakenVars (w' .> w) gvIn)
                  (weakenVars (w' .> w) gvbOut) :>:
                ArgArray In arrayR (weakenVars (w' .> w) gvIn) (k' weakenId) :>:
                weaken (w' .> w) argOut :>:
                ArgsNil)
```

Listing 17: A desugaring implementation for `map (\ x -> x * 2 + 1)` for the TensorFlow backend.

elaborated in section 3.1.1, `mkMap` perfectly isolates the flattening problem. As such, it makes sense to implement the flattening process inside of `mkMap`. Listing 18 depicts the implementation of `mkMap`. For starters, the implementation of `mkMap` pattern matches the function argument `ArgFun`. Because `mkmap` applies a method to each element of an array, the function always has exactly one argument. Correspondingly, the pattern `ArgFun (Lam lhs (Body body))` is the only possible case. As elaborated in section 2.2.1 and depicted by listing 12, the body consists of a scalar expression tree of type `PreOpenExp`.

```
mkMap (ArgFun (Lam lhs (Body body))) (ArgArray _ (ArrayR _ t) _ gvb) aOut =
  mkExp (push' Empty (lhs, distributeBIdx t gvb)) body aOut
mkMap _ _ _ = error "impossible"
```

Listing 18: The implementation of `mkMap`.

To flatten the method body, `mkMap` uses a helper method `mkExp`, which flattens the scalar expression tree. As depicted by listing 3.2.1, the signature has the following features. The first argument of `mkExp` consists of a buffer environment. The buffer environment `BufferEnv` keeps track of the previously declared variables. The second argument depicts the scalar expression that is to be desugared. Finally, the signature features an output array argument to which the result of the embedded scalar expression result should be written.

```
newtype BufferIdx benv a = BIdx (Idx benv (Buffer a))
type BufferEnv benv env = Env (BufferIdx benv) env

mkExp :: forall env env' sh t.
            BufferEnv env env'
        -> PreOpenExp (ArrayInstr env) env' t
        -> Arg env (Out sh t)
        -> OperationAcc TensorOp env ()
```

Listing 19: The signature for the `mkExp` method. The method flattens a scalar expression tree into a corresponding desugared definition of type `OperationAcc`.

The buffer environment for `mkExp` is required when handling variables. Scalar expressions such as `Let` and `Evar` allow the declaration and referencing of variables that point to expression buffers. Buffers are represented by an index which points to memory on the backend-specific device. As discussed in section 2.1.5, variables of environments consist of De Bruijn indices. Correspondingly, to assign a buffer to a variable, the buffer environment `BufferEnv` maps scalar variables to buffer variables. It consists of pairs of variables parameterised over `env` and buffer indices `benv`. For illustration, consider listing 3.2.1 below. It depicts the desugaring implementation for variable references with the `Evar` constructor of `PreOpenExp`. In the definition, `idx` is the De Bruijn index of the variable. Using `prj'` from example 6, the index is used to retrieve the pointer that points to the buffer from the environment. The `TId` constructor of `TensorOp` represents copying the values from one tensor into another. In the definition, the operation is used to mimic the retrieval of a value from a variable.

```
mkExp env (Evar (Var st idx)) (ArgArray _ arrayR gv gvb@(TupRsingle (Var groundR _)))
  | Refl <- reprIsSingle @ScalarType @t @Buffer st
  , OneOfDict <- tfAllDict st
  = let (BIdx idx') = prj' idx env
        gvb'        = TupRsingle (Var groundR idx')
    in  Exec
          TId
          (
            ArgArray In arrayR gv gvb' :>:
            ArgArray Out arrayR gv gvb :>:
            ArgsNil
          )
```

Listing 20: The implementation of variable accessing from the `Evar` constructor of `PreOpenExp`.

The method `mkExp` should provide a flattening definition for each scalar expression of type `PreOpenExp`. The list below elaborates on the implementations. The set of constructors is quite extensive. Therefore, straightforward implementations are only summarised or left out. The GitHub repository [5] of this thesis provides the full implementation for each pattern of `PreOpenExp`.

- The `Pair` constructor features a pair of scalar expressions. Both expressions should be executed. Listing 21 includes the execution of two expressions in the desugaring result. To implement the inclusion of two expressions, an empty let binding is used. Moreover, the `ALet` constructor in `PreOpenAcc` takes in two expressions. After running the first expression, the result of the first expression is assigned to the variable. Then, the second expression is run. The second expression has access to the variable. Both expressions must be run, but the variable assignment is optional. Hence, the running of two expressions can be represented in `PreOpenAcc` by a let expression with a unit variable.

```
mkExp env (Pair exp1 exp2)
 (ArgArray _ (ArrayR sh (TupRpair t1 t2)) gv (TupRpair gvb1 gvb2))
 = Alet (LeftHandSideWildcard TupRunit) TupRunit -- empty let binding
   (mkExp env exp1 (ArgArray Out (ArrayR sh t1) gv gvb1))
   (mkExp env exp2 (ArgArray Out (ArrayR sh t2) gv gvb2))
```

Listing 21: The implementation of desugarring the Pair constructor of PreOpenExp.

· Occasionally used inside nested pairs of expressions, the Nil constructor represents an empty expression. The desugared result is also an empty flow, denoted Return TupRunit.

· The Const expression creates a new array with all elements initialised with the same value. Similarly, in TensorFlow, the constant operation creates a Tensor with all elements initialised with the same value. Correspondingly, in TensorOp, the TensorFlow operation is represented by `TConstant :: OneOf TFAll a => ScalarType a -> a -> TensorOp (Out sh a -> ())`. The desugaring result of Const solely consists of running the constant method in TensorFlow.

· First-order scalar operations such as TF.add, TF.mul are featured in the PrimApp constructor of PreOpenExp. Moreover, the constructor features a primitive scalar operation representation of type PrimFun. The implementation of PrimApp consists of an application of the corresponding primitive operations in TensorFlow. Table 1 depicts each mapping. Almost all operations are also supported by TensorFlow, but not all of them. The existence of these unsupported functions is one of the limitations when converting Accelerate to TensorFlow.

· The constructor VecPack converts tuples of values to fixed-size vectors, and the constructor VecUnpack converts vectors to tuples of values. As elaborated in section 2.1.6, array elements as vectors are not included in the scope of this thesis. Therefore, the implementations for both constructors are not implemented.

· Slice operations in Accelerate refer to sub-arrays of arrays. An array is represented by a structure of arrays. To implement array slicing, only the buffers that contain the slice dimensions should be returned.

· Shapes and indices can be converted to and from their linear representation. The linear representation of an array is a 1-dimensional array of all elements in column-row ordering. The size of a shape is equal to the length of the linear representation of a shape. For instance, given an array with dimensions $d_0, d_1, ...d_n$, the length of the linear representation is $\prod_{i=0}^{i=n} d_i$. The ShapeSize constructor represents the computation of the size of an array. As depicted by listing 22, to desugar the operation, the TensorFlow multiplication operator TMul is applied with all corresponding buffers that contain the size of a dimension.

```
mkExp env (ShapeSize sh exp) aOut
  | DeclareVars lhs _ k <- declareVars $ shapeType sh
  = mkExp env (Let lhs exp (shapeSize sh (k weakenId))) aOut

shapeSize :: ShapeR sh -> ExpVars env' sh
          -> PreOpenExp (ArrayInstr env) env' Int
shapeSize ShapeRz TupRunit = Const scalarTypeInt 1
shapeSize (ShapeRsnoc shr) (TupRpair sh (TupRsingle sz))
  = PrimApp (PrimMul (IntegralNumType TypeInt))
      (Pair
        (shapeSize shr sh)
        (Evar sz))
shapeSize _ _              = error "impossible"
```

Listing 22: The desugaring implementation for the calculation of the size of an array shape.

· To continue, the ToIndex constructor linearises an index. That is, given an index of an arbitrarily-sized array, it calculates the corresponding index in the linearised representation of that array. Specifically, listing 23 depicts a method to calculate the shape in regular Haskell. Correspondingly, listing 24 depicts the embedded variant to calculate the shape for the TensorFlow backend.

Operation	**Accelerate**	**TensorFlow**
addition, multiplication subtraction, negation, absolute, sign	PrimAdd, PrimMul PrimSub, PrimNeg PrimAbs, PrimSig	add, mul sub, neg abs, sign
integer devision, integer devision truncated to 0	PrimIDiv PrimQuot	realDiv truncateDiv
remainder	PrimRem	truncateMod
simultaneous `quot` and `rem`	PrimQuotRem	(truncateDiv, truncateMod)
integer devision	PrimIDiv	realDiv
simultaneous `div` and `mod`	PrimDivMod	(realDiv, truncateMod)
bitwise and, or, xor, not, shift right, shift left	PrimBAnd, PrimBOr PrimBXor, PrimBNot PrimBShiftL, PrimBShiftR	bitwiseAnd, bitwiseOr bitwiseXor, invert, shiftRight, shiftLeft
bitwise count ones, count leading zeros count trailing zeros	PrimPopCount PrimCountLeadingZeros PrimCountTrailingZeros	unsupported
reciprocal fraction	PrimRecip	reciprocal
sine, cosine, tangent	PrimSin, PrimCos, PrimTan	sin, cos, tan
arc sine, cosine, tangent, 2-argument tangent	PrimAsin, PrimAcos, PrimAtan PrimAtan2	asin, acos, atan atan2
hyperbolic sine cosine, tangent	PrimSinh, PrimCosh, PrimTanh	sinh cosh, tanh
arc, hyperbolic sine, cosine, tangent	PrimASinh, PrimACosh, PrimAtanh	asinh, acosh, atanh
exponent, log, sqrt, pow, logbase	PrimExpFloating, PrimLog, PrimFSqrt, PrimFPow, PrimLogBase	exp, log, sqrt, pow, (\ x y → log(y) / log(x))
truncate to 0	PrimTruncate	unsupported
round, floor, ceil	PrimRound, PrimFloor, PrimCeil	round, floor, ceil
isNaN, isInfinite	PrimIsNan, PrimIsInfinite	isNan, isInf
$<$, $>$, $\leq$, *geq*, $=$, *neq*,	PrimLt, PrimGt, PrimLtEq, PrimGtEq, PrimEq, PrimNEq	less, greater, lessEqual, greaterEqual equal, notEqual
maximum, minimum	PrimMax, PrimMin	maximum, minimum
logical and, or, not	PrimLand, PrimLor PrimNot	logicalAnd, logicalOr logicalNot
Haskell's fromIntegral, and toFloat	PrimFromIntegral, PrimToFloating	cast

Table 1: Direct mapping of all Accelerate primitive functions to TensorFlow operations.

```
toIndex :: ShapeR sh -> sh -> sh -> Int
toIndex ShapeRz () () = 0
toIndex (ShapeRsnoc shr) (sh, sz) (ix, i)
  = toIndex shr sh ix * sz + i
```

Listing 23: A regular Haskell definition to linearise an index, copied from the Accelerate source code [4].

```
mkExp env (ToIndex sh exp1 exp2) aOut
  | DeclareVars lhs w k <- declareVars $ shapeType sh
  , DeclareVars lhs' w' k' <- declareVars $ shapeType sh
  = mkExp env (Let lhs exp1 (Let lhs' (weakenE w exp2)
      (toIndex sh (k w') (k' weakenId)))) aOut

toIndex :: ShapeR sh -> ExpVars env' sh -> ExpVars env' sh
        -> PreOpenExp (ArrayInstr env) env' Int
toIndex ShapeRz TupRunit TupRunit = Const scalarTypeInt 0
toIndex (ShapeRsnoc shr) (TupRpair sh (TupRsingle sz)) (TupRpair ix (TupRsingle i))
  = PrimApp (PrimAdd (IntegralNumType TypeInt))
      (Pair
        (PrimApp (PrimMul (IntegralNumType TypeInt))
          (Pair
            (toIndex shr sh ix)
            (Evar sz)))
        (Evar i))
toIndex _ _ _                         = error "impossible"
```

Listing 24: The desugaring implementation for the linearisation of an index.

- The constructor FromIndex converts a linearised index back to its original form. In a similar fashion to the implementation of ToIndex, the Haskell variant of the computation of listing 25 is implemented by an embedded variant.

```
fromIndex :: ShapeR sh -> sh -> Int -> sh
fromIndex ShapeRz () _ = ()
fromIndex (ShapeRsnoc shr) (sh, sz) i
  = (fromIndex shr sh (i `quotInt` sz), r)
  -- If we assume that the index is in range, there is no point in computing
  -- the remainder for the highest dimension since i < sz must hold.
  --
  where
    r = case shr of -- Check if rank of shr is 0
      ShapeRz -> $indexCheck "fromIndex" i sz i
      _       -> i `remInt` sz
```

Listing 25: A regular Haskell definition to un-linearise an index, copied from the Accelerate source code [4].

- The ArrayInstr constructor represents array accessing instructions. For starters, as introduced in section 2.2.1, the Parameter instruction retrieves a scalar value from the Accelerate program. To implement variable accessing during execution, TensorOp is given a new constructor TVar. Although not directly mapped to an existing TensorFlow operation, section 3.2.2 will demonstrate how the operation is implemented in the execution phase. Finally, there is the Index operation which returns an element from the buffer at the given position. mkExp features an output array to store the result. In other words, the output of the Index operation is also an array. The TF.gather operation in TensorFlow retrieves values from an array given an array of indices. Thus, using an array with the index as its only element, the TGather constructor in TensorOp can be used to desugar the operation.

- The Undef constructor represents an operation that is somewhat similar to the undefined statement in Haskell. To quote the Accelerate documentation: "This is useful because a store of an undefined value can be assumed to not have any effect; we can assume that the value is overwritten with bits that happen to match what was already there. However, a store to an undefined location could clobber arbitrary memory, therefore, its use in such a context would introduce undefined behaviour." [2]. TensorFlow does not support undefined Tensors. However, assuming the input program is correct, the `Undef` value is supposed to be overwritten at some point. Therefore, the desugaring implementation can simply resort to initialising a Tensor with zeroes.

- The `Coerce` constructor interprets the elements of an array as a different type. Correspondingly, the `TF.cast` operation in TensorFlow casts elements of a Tensor into a different type. The desugaring implementation uses the `TCast` constructor of `TensorOp`.

- The `Cond` constructor represents a conditional expression. The operation has three input arrays of the same shape. The first array consists of booleans, if a boolean at position i is true, then the element at position i in the result is the i-th element in the second array. If the boolean is false, then it should be that of the third array. In the Accelerate pipeline, the elements of the second and third arrays are assumed to remain unevaluated before selecting them with the boolean input. In TensorFlow, the `TF.select` operator also has a tensor of booleans and two input tensors as input. However, both input tensors need to be fully computed before they can be used as input. As discussed in the results, this causes desugared programs with array-accessing while loops to fail. This characterises another limitation of converting Accelerate to TensorFlow.

- The `Case` constructor resembles a `switch` operation, which is a variant of `Cond`, but with any number of conditional cases. Because of its complexity and the aforementioned issues with `Cond`, the `Case` constructor is not implemented in the desugaring step.

- The `While` constructor depicts a loop instruction. Moreover, the arguments of the constructor consist of a condition function, an iteration function and a set of input values. First, it applies the condition function to each input. Subsequently, each element for which the condition function returns true is updated with the iteration function. Then, it repeats this process until there are no elements for which the condition function returns true. Similar to `Cond`, given a list of booleans, the computation sequence is different for each element, which causes desugared programs with array-accessing while loops to fail. As with `Case`, because of the complexity of `While`, and the incompleteness of its implementation, the `While` constructor is not implemented.

- Accelerate's Foreign Function Interface constructor `Foreign` calls a foreign scalar expression from the targeted backend. It also features a fallback method that yields the same result. To limit the scope of this thesis to just support Accelerate in TensorFlow, the desugaring implementation always runs the fallback method.

Desugaring generate With the higher-order operation generalised with the implementation of `mkMap`, the desugaring implementation is completed by implementing `mkGenerate` and `mkPermute` in terms of `mkMap`. For starters, the `generate` operation in Accelerate creates a new array, where each element contains the result of a given function applied to the index of that element. Listing 26 depicts the full implementation. The signature consists of a function argument and an output array. The method should apply the function argument to a new array of shape `sh`, and write the results in the output array. The process is defined as follows:

- To flatten the function argument, it needs to be desugared into a first-order series of expressions. As elaborated above, `mkMap` isolates the flattening problem. What's more, `mkMap` applies a method to all elements of an input array and writes the output to an output array. As such, in the implementation for `mkGenerate`, `mkMap` applies the function to an array of shape `sh`, where each element is the index of that element. TensorFlow does not feature a single method to obtain such an array of indices. However, the method `TF.where` returns an array of indices of all elements greater than 1. So, to obtain such a method, step 1 creates a tensor of ones, and step 2 applies `TWhere` to obtain the array of all indices.

- In Accelerate, an array of indices of shape `sh` is represented by an array of shape `sh` with elements of type `sh`. Therefore, the shape of the array remains `sh`. In contrast, in TensorFlow, a tensor of indices does not exist. Instead, a tensor of elements of type `sh` is just a tensor of rank `sh` + 1. That is, the tensor does not contain tuples of dimensions but is just a regular tensor of indices of one higher dimension. In Accelerate, tuple elements are stored in separate buffers. So, to run the step above, the indices returned by TensorFlow should somehow be mapped back to the allocated buffers. However, the TensorFlow bindings provide no type-safe guarantee over the kind of output of the `TF.where` operation. Regardless, for 1-dimensional arrays, TensorFlow just returns an array of integers. Therefore, the aforementioned mapping complexity can be avoided by linearising the input array. Correspondingly, step 1 creates a linearised array of ones with the same size as the original shape `sh`. Then, the `TWhere` operation returns a 1-d array of indices in step 2, and step 3 converts the linearised indices back to their original dimensions.

· With the input array of indices returned by TensorFlow by step 3, step 4 applies `mkMap` with the function parameter. The result is written to the output array from the second argument. To implement the steps above, new variables had to be declared. To parameterise the output over the newly declared arrays, the environments are weakened with the new declarations.

Desugaring permute In Accelerate, the permute function defines the generalised forward permutation operation (array scatter) [2]. As illustrated by example 9, `permute` initialises its result array with a given array of defaults. Then, it combines elements in the result with elements selected from a given list of permutation elements. The permutation function specifies a result index for each position in the given array of permutation elements. In other words, it provides a position of the result element. The permutation function may also return `Nothing`. If so, a permutation element is not combined with a result element. The combination function parameter gives a way of combining elements. With its two function parameters, `permute` is a higher-order method.

Example 9: An example use case for permute is the computation of the occurrence count (histogram) for an array of values [2]. As depicted below, the `histogram` method uses `permute` to count the number of occurrences using the `(+)` combination function. Each position `i` in the result array corresponds to the number of occurrences of `i` in the input.

```
histogram :: Acc (Vector Int) -> Acc (Vector Int)
histogram xs =
  let zeros = fill (constant (Z:.10)) 0
      ones  = fill (shape xs)         1
  in  permute (+) zeros (\ix -> Just_ (I1 (xs!ix))) ones

>>> let xs = fromList (Z :. 20) [0,0,1,2,1,1,2,4,8,3,4,9,8,3,2,5,5,3,1,2]
>>> run $ histogram (use xs)
Vector (Z :. 10) [2,4,4,3,2,2,0,0,2,1]
```

```
mkPermute :: Arg env (Fun' (e -> e -> e))
          -> Arg env (Mut sh' e)
          -> Arg env (Fun' (sh -> PrimMaybe sh'))
          -> Arg env (In sh e)
          -> OperationAcc TensorOp env ()
```

Listing 27: The signature of desugaring method for `permute`.

The signature of the corresponding `mkPermute` method is depicted in listing 27. The implementation of `mkPermute` has to tackle the following obstacles.

1. The combination and permutation function parameters must be flattened. In a similar fashion to `mkGenerate`, the application of both methods can be defined with `mkMap`.

2. TensorFlow does not contain a single method that exactly implements the same behaviour as permute. Instead, the desugaring result should be composed of other TensorFlow operations to mimic the behaviour of `permute`.

```
mkGenerate :: Arg env (Fun' (sh -> t)) -> Arg env (Out sh t)
         -> OperationAcc TensorOp env ()
mkGenerate f (ArgArray _ (ArrayR sh t) gv gvb)
| DeclareVars lhs w k          <- declareVars $ TupRsingle (GroundRscalar scalarTypeInt)
, DeclareVars lhs' w' k'       <- declareVars $ buffersR (TupRsingle scalarTypeInt)
, DeclareVars lhs'' w'' k''    <- declareVars $ buffersR (TupRsingle scalarTypeInt)
, DeclareVars lhs''' w''' k''' <- declareVars $ buffersR (shapeType sh)
= aletUnique lhs (Compute (ShapeSize sh (paramsIn' $ fromGrounds gv))) $
  -- 1) Create a Tensor of flattened shape sh with only ones
  aletUnique lhs' (desugarAlloc (ArrayR dim1 (TupRsingle scalarTypeInt))
    (fromGrounds (TupRpair TupRunit (k weakenId)))) $
  Alet (LeftHandSideWildcard TupRunit) TupRunit
  (Exec
    (TConstant scalarTypeInt 1)
    (ArgArray Out (ArrayR dim1 (TupRsingle scalarTypeInt))
      (TupRpair TupRunit (k w')) (k' weakenId) :>: ArgsNil)
  ) $
  -- 2) Obtain a tensor of indices (tf.where returns list of indices of values > 0)
  aletUnique lhs'' (desugarAlloc (ArrayR dim1 (TupRsingle scalarTypeInt))
    (fromGrounds (TupRpair TupRunit (k w')))) $
  Alet (LeftHandSideWildcard TupRunit) TupRunit
  (Exec
    TWhere
    (ArgArray In (ArrayR dim1 (TupRsingle scalarTypeInt))
      (TupRpair TupRunit (k (w'' .> w'))) (k' w'') :>:
     ArgArray Out (ArrayR dim1 (TupRsingle scalarTypeInt))
      (TupRpair TupRunit (k (w'' .> w'))) (k'' weakenId) :>:
     ArgsNil)
  ) $
  -- 3) Convert 1d indices to multidimensional indices
  aletUnique lhs''' (desugarAlloc
    (ArrayR dim1 (shapeType sh))
    (fromGrounds $ TupRpair TupRunit (k (w'' .> w')))) $
  Alet (LeftHandSideWildcard TupRunit) TupRunit
  (mkMap
    (ArgFun $ Lam (LeftHandSideSingle scalarTypeInt) $
      Body (FromIndex sh
        (paramsIn' $ fromGrounds $ weakenVars (w''' .> w'' .> w' .> w) gv)
        (Evar (Var scalarTypeInt ZeroIdx))))
    (ArgArray In (ArrayR dim1 (TupRsingle scalarTypeInt))
      (TupRpair TupRunit (k (w''' .> w'' .> w'))) (k'' w'''))
    (ArgArray Out (ArrayR dim1 (shapeType sh))
      (TupRpair TupRunit (k (w''' .> w'' .> w'))) (k''' weakenId))
  )
  -- 4) Apply f to the indices
  (mkMap
    (weaken (w''' .> w'' .> w' .> w) f)
    (ArgArray In (ArrayR dim1 (shapeType sh))
      (TupRpair TupRunit (k (w''' .> w'' .> w'))) (k''' weakenId))
    (ArgArray Out (ArrayR dim1 t)
      (TupRpair TupRunit (k (w''' .> w'' .> w')))
      (weakenVars (w''' .> w'' .> w' .> w) gvb))
  )
```

Listing 26: The desugaring implementation for `generate`.

In TensorFlow, array scatter functions are supported with scatter operations [6]. For example, the TF.scatter_nd operation initialises a tensor of zeroes and adds elements to their scattered position in the result array. In contrast to TensorFlow's TF.scatter_nd, Accelerate's permute uses a variable set of default values. And, it may use a combination different from TF.scatter_nd's addition method. More than one element can be combined with the same result position. Therefore, when using TF.scatter_nd, it is hard to replicate the actual combination afterwards. As a result, using TF.scatter_nd would yield a very limited implementation of permute. Fortunately, TensorFlow also features tensor scatter operations TF.tensorScatterAdd, TF.tensorScatterSub, TF.tensorScatterMax, TF.tensorScatterMin and TF.tensorScatterUpdate. For starters, these operations scatter elements into an array of existing values. Respectively, they use the Haskell equivalent combination functions (+), (-), max, min, const. Using these operators, at least five permutation functions can be implemented for permute. TensorFlow does not feature any other methods to scatter and combine values from one tensor to another. Therefore, the implementation will use the aforementioned tensor scatter operations, and throw an error if some other combination function is used for permute. This defines one of the limitations of the conversion to TensorFlow.

3. As with mkGenerate, because shapes and indices are represented differently in TensorFlow, TensorFlow operations that deal with shape should only use and return 1-dimensional arrays.

Listing 32 depicts the single pattern match for the implementation. It assumes that the type of elements matches the types used by the TensorFlow operations. For other types, it throws an error. As elaborated above, the implementation of mkPermute uses the tensor scatter operations to scatter and combine values from one tensor to another. Listing 28 below depicts the corresponding TTensorScatter constructor for TensorOp. The tensor scatter operations share the same signature. As such, the constructor represents all five tensor scatter operations. The constructor contains an argument of type ScatterFun which corresponds to which combination operation is used. Because all TensorFlow shape operations should only use 1-dimensional arrays, the shape of the arrays for TTensorScatter are restricted by DIM1.

```
data TensorOp op where
  TTensorScatter :: TensorType a
                 => ScatterFun
                 -> TensorOp (Mut DIM1 a -> In DIM1 Int -> In DIM1 a -> ())

data ScatterFun where
  ScatterFunAdd    :: ScatterFun
  ScatterFunMin    :: ScatterFun
  ScatterFunMax    :: ScatterFun
  ScatterFunSub    :: ScatterFun
  ScatterFunUpdate :: ScatterFun
```

Listing 28: The tensor scatter constructor in TensorOp.

To continue, listing 29 depicts the restriction over the kind of allowed combination methods for permute. The definition is used in the last step of listing 32.

```
scatterFun :: ScatterFun = case comb of
  Lam (LeftHandSideSingle _) (Lam (LeftHandSideSingle _)
    (Body (PrimApp fun (Pair (Evar (Var _ (SuccIdx ZeroIdx)))
      (Evar (Var _ ZeroIdx)))))) -> case fun of
    PrimAdd _ -> ScatterFunAdd -- (+)
    PrimSub _ -> ScatterFunSub -- (-)
    PrimMin _ -> ScatterFunMin -- min
    PrimMax _ -> ScatterFunMax -- max
    _         -> error "only add, sub, min, max, const allowed"
  Lam (LeftHandSideSingle _) (Lam (LeftHandSideSingle _)
    (Body (Evar (Var _ (SuccIdx ZeroIdx))))) -> ScatterFunUpdate -- const
  _ -> error "only add, sub, min, max, const allowed"
```

Listing 29: The tensor scatter function is pattern matched against the value of the comb parameter.

In TensorFlow, a tensor scatter operation requires a tensor `input` of default values, a list of scatter indices `indices`, and a list of permutable values `updates`. Each element `i` of updates is combined with element `indices!i` in the result array. The shapes of `input` and `updates` must be equal. This restriction does not exist in Accelerate. For instance, the histogram method in example 9 applies `permute` with different shapes. Therefore, yet another limitation is imposed on the possible permute definitions that can be supported by TensorFlow. As depicted by listing 32, the three tensors `input`, `indices` and `updates` are obtained as follows.

- The `input` array simply corresponds to the `defaults` array parameter of `mkPermute`. However, because TensorScatter may only deal with 1-dimensional arrays, the `defaults` array is flattened in the desugaring implementation.
- The `indices` array is obtained as follows. In the first step of the implementation, the permutation function is passed as a parameter to the `mkGenerate` method. The elements in the resulting array are of type `Maybe sh'`. Their value corresponds to the output of the permutation function based on their position in the array. The `indices` array may only have actual indices, as the tensor scatter operations do not support empty indices such as the `Nothing` value. So, to obtain the correctly formatted array, the resulting array of `mkGenerate` is filtered with a predicate `isJust`, and then mapped with `fromJust`. Listing 30 depicts the definitions for both predicates. Moreover, in the Accelerate backend, `PrimMaybe` is defined as `(Word8, ((), a))`. Therefore, applying the map operation for both predicates is as simple as extracting the correct buffer variable from the corresponding tuple of buffers.

```haskell
isJust :: TypeR a -> GroundVars env (Buffers (Word8, a))
       -> GroundVars env (Buffers Word8)
isJust _ (TupRpair word8 _) = word8
isJust _ _ = error "impossible"

fromJust :: TypeR a -> GroundVars env (Buffers (Word8, ((), a)))
         -> GroundVars env (Buffers a)
fromJust _ (TupRpair _ (TupRpair _ a)) = a
fromJust _ _ = error "impossible"
```

Listing 30: The implementation of mapping `isJust` and `fromJust`.

Filtering with a predicate result is a higher-order operation, which is not supported by TensorFlow. Instead, the desugared implementation of filtering implements the following approach. For starters, given a tensor with elements to filter, and an array of booleans of the same shape, the `TF.booleanMask` operation in TensorFlow removes each element from the input tensor if the element at the same position in the boolean tensor is `False`. To obtain the array of booleans, the predicate is mapped over the input array. Listing 31 below depicts the implementation of filtering with `TF.booleanMask`. The operation is missing from the TensorFlow Haskell bindings [29]. Therefore, it is replaced with an equivalent implementation with `TF.where` and `TF.gather`.

- The `updates` array corresponds to the values of the array of source values from the parameters. TensorFlow's tensor scatter methods require the inputs to be of equal size. From the definition of the tensor scatter operations, each element `i` of updates is combined with element `indices!i` in the result array. Hence, as some `Nothing` indices are left out, the corresponding permutable values should also be left out of `updates`. Fortunately, because the index returned `Nothing`, the permutable value will not be used. Therefore, this necessary step will not have an impact on the result of `permute`. With the mask obtained by mapping the `isJust` predicate over the array with elements of type `Maybe sh'`, the `booleanMask` operation is used to obtain the list of `updates` without the elements for which the permutation function returned `Nothing`.

```haskell
booleanMask :: TypeR a -> Arg env (In DIM1 Word8) -> GroundVars env (Buffers a)
            -> GroundVars env (Buffers a) -> PreOpenAcc TensorOp env ()
booleanMask TupRunit _ _ gvb = Return gvb
booleanMask t@(TupRsingle s) (ArgArray _ (ArrayR sh tIn1) gv gvbIn1) gvbIn2 gvbOut
  | OneOfDict <- tfAllDict s
  , DeclareVars lhs w k <- declareVars $ buffersR (TupRsingle scalarTypeInt)
  = aletUnique lhs
    (desugarAlloc (ArrayR sh (TupRsingle scalarTypeInt)) (fromGrounds gv)) $
    Alet (LeftHandSideWildcard TupRunit) TupRunit
    (Exec TWhere
      (ArgArray In (ArrayR dim1 tIn1) (weakenVars w gv) (weakenVars w gvbIn1) :>:
       ArgArray Out (ArrayR dim1 (TupRsingle scalarTypeInt)) (weakenVars w gv)
       (k weakenId) :>: ArgsNil)
    )
    (Exec TGather
      (ArgArray In (ArrayR dim1 t) (weakenVars w gv) (weakenVars w gvbIn2) :>:
       ArgArray In (ArrayR dim1 (TupRsingle scalarTypeInt)) (weakenVars w gv)
         (k weakenId) :>:
       ArgArray Out (ArrayR dim1 t) (weakenVars w gv) (weakenVars w gvbOut) :>:
       ArgsNil)
    )
```

Listing 31: The desugaring implementation for applying a boolean mask.

3.2.2 Execution

In summary of section 3.1.2, after desugaring, the next steps in the Accelerate pipeline are to schedule the desugared instructions and execute the schedule. The TensorFlow backend uses a sequential scheduler. A sequential schedule depicts a flow of backend operation execution instructions. Among others, the flow may depict instructions for memory allocation, conditions and loops. As each backend should implement memory handling by itself, an input or output buffer in the schedule consists of a De Bruijn index. The approach is to bundle as many execution instructions in a single TensorFlow graph as possible, so the values in memory should be able to store graphs. The paragraphs below discuss the implementation of each step in the code. For reference, the discussed implementation is provided in this thesis' GitHub repository [5].

Kernels The pipeline converts the desugared `OperationAcc` result into a schedule using the sequential scheduler. What's more, similar to how `OperationAcc` is parameterised over `TensorOp`, to make it backend-specific, a schedule is parameterised over a `Kernel` implementation. Moreover, a kernel contains backend-specific operations and metadata. For the new TensorFlow backend, the new `TensorKernel` type contains a constructor for every operation listed in `TensorOp`, and no metadata. In the code snippets below, every `TensorKernel` constructor corresponds to the similarly named constructor of `TensorOp`. For instance, the `TConstant` constructor in `TensorOp` corresponds to the `TensorConstant` constructor of `TensorKernel`.

Values in memory Using an environment, the De Bruijn index is mapped to variables in memory. Listing 33 depicts the type `TensorEnv` of the new environment. In general, state modification in Haskell is performed inside Haskell's `IO` monad. Furthermore, the data type `IORef` from Haskell's `Data.IORef` references variables stored in the `IO` monad. Specifically, the methods `newIORef` `writeIORef` and `readIORef` respectively create, store and read variables. When executing a schedule, the module `Data.IORef` is used to handle state manipulation. The result is wrapped inside TensorFlow bindings' `TF.Session` monad. The monad is an extension of the `IO` monad. As such, using Haskell's `liftIO` method in a `do` statement, the tensor variables can be updated inside the `TF.Session` monad. The buffer representations in the schedule point to references to variables in the `IO` monad that store input and output values for tensor operations. Accordingly, the `Tensor` constructor of the value data type `TensorElement` contains an `IORef` argument. The approach is to combine tensor operations into one graph until the graph value is required to continue the flow execution. Therefore, a buffer may point to a graph, or the result from running a graph. For the latter, the result of executing a graph with

```
mkPermute (ArgFun comb) (ArgArray _ (ArrayR sh' _) gv' gvb') perm
(ArgArray _ (ArrayR sh t) gv gvb)
| TensorTypeDict                     <- tfTensorTypeDict' t
, OneOfDict                          <- tfAllDict' t
, maybeSh'                           <- TupRpair (TupRsingle scalarTypeWord8)
                                                 (TupRpair TupRunit (shapeType sh'))
, DeclareVars lhs w k                <- declareVars $ buffersR maybeSh'
, DeclareVars lhs' w' k'             <- declareVars $ TupRsingle $ GroundRscalar scalarTypeInt
, DeclareVars lhs'' w'' k''          <- declareVars $ buffersR (shapeType sh')
, DeclareVars lhs''' w''' k'''       <- declareVars $ buffersR t
, DeclareVars lhs'''' w'''' k''''    <- declareVars $ buffersR (TupRsingle scalarTypeInt)
, DeclareVars lhs''''' w''''' k''''' <- declareVars $ TupRsingle $ GroundRscalar scalarTypeInt
, DeclareVars lhsSh' _ kSh'          <- declareVars $ shapeType sh'
, DeclareVars lhsSh'' wSh'' kSh''    <- declareVars $ shapeType sh'
= -- 1) Create an array of maybeSh' with perm
  aletUnique lhs (desugarAlloc (ArrayR sh maybeSh') (fromGrounds gv)) $
  Alet (LeftHandSideWildcard TupRunit) TupRunit
  (mkGenerate (weaken w perm)
    (ArgArray Out (ArrayR sh maybeSh') (weakenVars w gv) (k weakenId))
  ) $ -- 2) To apply filter with 1D arrays, we need to flatten. Calculate the dim1 size
  aletUnique lhs' (Compute (ShapeSize sh (paramsIn' $ weakenVars w $ fromGrounds gv))) $
  -- 3) Get 1D array of indices (sh'): filter with isJust and map with fromJust.
  aletUnique lhs'' (desugarAlloc (ArrayR sh (shapeType sh'))
    (fromGrounds (weakenVars (w' .> w) gv))) $
  Alet (LeftHandSideWildcard TupRunit) TupRunit (booleanMask (shapeType sh')
    (ArgArray In (ArrayR dim1 (TupRsingle scalarTypeWord8)) (TupRpair TupRunit (k' w''))
      (isJust (TupRpair TupRunit (shapeType sh')) (k (w'' .> w'))))
    (fromJust (shapeType sh') (k (w'' .> w'))) (k'' weakenId)
  ) $ -- 4) Get 1D array of updates by filtering with predicate isJust.
  aletUnique lhs''' (desugarAlloc (ArrayR sh t)
    (fromGrounds (weakenVars (w'' .> w' .> w) gv))) $
  Alet (LeftHandSideWildcard TupRunit) TupRunit
  (booleanMask t (ArgArray In (ArrayR dim1 (TupRsingle scalarTypeWord8))
      (TupRpair TupRunit (k' (w''' .> w'')))
      (isJust (TupRpair TupRunit (shapeType sh')) (k (w''' .> w'' .> w'))))
    (weakenVars (w''' .> w'' .> w' .> w) gvb) (k''' weakenId)
  ) $ -- 5) Map array of indices (sh') with toIndex to obtain 1D array of flattened indices.
  aletUnique lhs'''' (desugarAlloc (ArrayR sh (TupRsingle scalarTypeInt))
    (fromGrounds (weakenVars (w''' .> w'' .> w' .> w) gv))) $
  Alet (LeftHandSideWildcard TupRunit) TupRunit
  (mkMap  (ArgFun $ Lam lhsSh' $ Body (Let lhsSh''
    (paramsIn' $ fromGrounds (weakenVars (w'''' .> w''' .> w'' .> w' .> w) gv')) $
     toIndex sh' (kSh'' weakenId) (kSh' wSh'')))
    (ArgArray In (ArrayR dim1 (shapeType sh'))
      (TupRpair TupRunit (k' (w'''' .> w''' .> w''))) (k'' (w'''' .> w''')))
    (ArgArray Out (ArrayR dim1 (TupRsingle scalarTypeInt))
      (TupRpair TupRunit (k' (w'''' .> w''' .> w''))) (k'''' weakenId))
  ) $ -- 6) Compute linearised index of input shape
  aletUnique lhs''''' (Compute (ShapeSize sh'
    (paramsIn' $ weakenVars (w'''' .> w''' .> w'' .> w' .> w) $ fromGrounds gv'))) $
  -- 7) Apply tensor scatter to source values, flattened indices and updates.
  Exec (TTensorScatter scatterFun) (
    ArgArray Mut (ArrayR dim1 t) (TupRpair TupRunit (k''''' weakenId))
      (weakenVars (w''''' .> w'''' .> w''' .> w'' .> w' .> w) gvb') :>:
    ArgArray In (ArrayR dim1 (TupRsingle scalarTypeInt)) (TupRpair TupRunit
      (k' (w''''' .> w'''' .> w''' .> w''))) (k'''' w''''') :>:
    ArgArray In (ArrayR dim1 t) (TupRpair TupRunit
      (k' (w''''' .> w'''' .> w''' .> w''))) (k''' (w''''' .> w'''')) :>:  ArgsNil)
```

Listing 32: The desugaring implementation for permute.

TF.run operation consists of the elements of the result tensor, which are represented by Haskell's Vector type from Data.Vector.Storable. Accordingly, the TensorValue data type that is stored inside the IO monad may contain a graph or a graph execution result. Finally, the Scalar constructor also allows regular Haskell values to be stored. Moreover, in some cases, a computation may be performed by the default Accelerate interpreter instead, which yields a regular Haskell value. Such values may also be used as input for tensor operations. Because these values are not tensor values, they are referenced by a buffer. As such, the Scalar constructor parameterises over a, and the Tensor constructor over Buffer a.

```haskell
type TensorEnv = Env TensorElement

data TensorValue a where
  Build  :: TF.Shape -> TF.Tensor TF.Build a -> TensorValue a -- constructed graph
  Vector :: TF.Shape -> S.Vector a -> TensorValue a -- graph execution result
  Nil    :: TF.Shape -> TensorValue a -- empty value, used for allocation

data TensorElement a where
  Scalar :: TypeR a -> a -> TensorElement a
  Tensor :: VectorType (Type64 a) => ScalarType a
         -> IORef (TensorValue (Type64 a)) -> TensorElement (Buffer a)
```

Listing 33: The environment of type TensorEnv allows input and output values for tensor operations to be accessed with a De Bruijn index given by a buffer in a schedule.

Running a schedule Similar to the other steps in the pipeline, a sequential schedule is represented as a GADT with recursive constructors. To execute the schedule, all constructors of the recursively defined schedule are pattern matched. The type of the result is TF.Session (TupR TensorElement t). Each case yields a tuple of tensor elements, such that the implementation recursively returns the corresponding nested tuple of tensor elements. The approach only executes graphs when the scheduled flow is dependent on an output of an operation. Therefore, the result may still contain unexecuted graphs. As such, before yielding the final result, all returned graphs are first executed, such that all tensor elements are either Scalar, or a Tensor with a TensorValue IORef to a Vector constructor. Each constructor of the schedule GADT depicts a flow component. The paragraphs below elaborate on how each component is defined and implemented in the TensorFlow backend.

Memory allocation As elaborated above, buffer indices are used to retrieve values from the IO monad using the tensor environment with type TensorEnv. However, before these variables can be addressed, they must be created. The sequential schedule features three constructors to create new values, namely Alloc, Use and Unit.

For starters, the Alloc constructor represents memory allocation. The constructor only contains a shape reference that indicates the shape of a buffer that should be allocated. Using newIORef, the TensorFlow implementation creates a new variable in IO to be used later in the flow. The newIORef method requires an initial value for the new variable. However, because Alloc is only meant to allocate space, the value is initially Nil. For this reason, as depicted in listing 33, the Nil constructor represents a newly created, but yet unused variable. A downside of including empty variables in the execution phase is that it can cause unforeseen errors when a Nil tensor value is used as input for an operation. However, assuming the scheduler and the desugaring phase are implemented correctly, this behaviour should not occur. As depicted in listing 34, the corresponding implementation returns a single tuple element with the newly created tensor.

Regular Haskell values are represented by the Scalar constructor of TensorElement. To capture a scalar value inside of a new buffer, the Unit constructor provides a De Bruijn index pointing to a Scalar value in the environment of type TensorEnv. As depicted below, scalar values are the result of running the Compute constructor. To elaborate, the Unit constructor occurs whenever a scalar value is used as input for a buffer operation. In other words, it embeds a regular Haskell value such that it can be used

```
executeSeqSchedule env (Alloc shR st vars)
  | VectorTypeDict <- tfVectorTypeDict st
  = do
  sh <- liftIO $ getShape env shR vars
  ref <- liftIO $ newIORef (Nil (TF.Shape sh))
  return $ TupRsingle $ Tensor st ref
```

Listing 34: The implementation of the execution step for the `Alloc` schedule constructor.

for calculations on the backend-specific device. Similar `Alloc`, the implementation of `Unit` returns the created tensor with a single element.

Comparable to `Unit`, the `Use` constructor also embeds values such that they can be used on the backend-specific device. Instead of a single value, `Use` represents the embedding of a list of values. Correspondingly, the implementation returns a new tensor with a list of elements.

Variable handling The `Alet` constructor represents bindings of variables in the computation flow. It contains three arguments. The first argument of type `LeftHandSide GroundR` (listings 9, 8) denotes the kind of variable that will be declared. The second argument contains the sub-schedule that returns the variable value. The third argument contains the remainder of the schedule, which also has access to the new variable. As depicted in listing 35, the implementation returns the result of the remainder of the schedule. Often, programs contain multiple statements. In languages such as Java and C#, multiple calculations are indicated with `;`. In the pipeline, multiple calculations are indicated with `let () = a in b`. In that case, for `Alet`, the first argument is `TupRUnit`.

```
executeSeqSchedule env (Alet lhs _ sched sched') = do
  rhs <- executeSeqSchedule env sched
  let env' = push env (lhs, rhs)
  executeSeqSchedule env' sched'
```

Listing 35: The implementation of the execution step for the `Alet` schedule constructor.

Variables are used by other components in the flow. For instance, the `Return` constructor reads values from variables and returns their values. The constructor only contains a tuple of variables. The implementation retrieves each corresponding tensor element from the environment of type `TensorEnv` and returns its result.

Flow control The Accelerate language supports flow control with the `acond` and `awhile` operations. Listing 36 depicts the signatures of both operations. In the pipeline, with the sequential scheduler, occurrences of these operations are directly translated to the `Acond` and `Awhile` schedule constructors.

```
awhile :: forall a. Arrays a =>
  (Acc a -> Acc (Scalar Bool)) -> (Acc a -> Acc a) -> Acc a -> Acc a
acond :: forall a. Arrays a =>
  Exp Bool -> Acc a -> Acc a -> Acc a
```

Listing 36: The signatures of Accelerate's `awhile` and `acond` operations.

First, the higher-order `awhile` operation resembles a while loop operation. Specifically, it starts with an initial value of type `Acc a` that is run against a condition operation of type
`(Acc a -> Acc (Scalar Bool))`. If true, the body of type `(Acc a -> Acc a)` is executed once, and the process is repeated with the output from the body. As such, the process loops until the condition is false. To implement the equivalent `Awhile` schedule constructor for the TensorFlow backend, the execution is dependent on the output of the graph that outputs the condition. The `Awhile` constructor depicts a while loop, which has two sub-schedules as its arguments. The first schedule computes the boolean condition value of the loop. The second schedule depicts the computations of the loop body. The execution of the loop depends on the intermediate value returned by running the graph constructed from the condition

schedule. Namely, if the condition is true, the body schedule is executed, and the condition is checked again. To inspect the intermediate value, the tensor graph constructed from the condition schedule must be run. Listing 37 depicts the implementation of the `Awhile` constructor for the TensorFlow backend.

```
executeSeqSchedule env (Awhile _ cond exp vars) =
  executeAwhile env cond exp (mapTupR (\(Var _ idx) -> prj' idx env) vars)

executeAwhile :: TensorEnv env
              -> SeqScheduleFun TensorKernel env (t -> PrimBool)
              -> SeqScheduleFun TensorKernel env (t -> t)
              -> TensorElements t -> TF.Session (TensorElements t)
executeAwhile env
  cond@(Slam condLhs (Sbody condSched))        -- while condition schedule
  body@(Slam expLhs (Sbody expSched)) ts = do  -- while body schedule
    liftIO $ runTensorElements ts -- Execute all unexecuted graphs in scope.
    TupRsingle condElement <- executeSeqSchedule (push env (condLhs, ts)) condSched
    liftIO $ runTensorElement condElement -- Execute conditional graph.
    condValue <- liftIO $ returnTensorElement condElement
    if toBool condValue -- Check boolean result from conditional.
      then do -- If true, perform while body
              ts' <- executeSeqSchedule (push env (expLhs, ts)) expSched
              executeAwhile env cond body ts'
      else return ts -- If false, return tensor values.
executeAwhile _ _ _ _ = error "impossible"
```

Listing 37: The implementation of the execution step for the `Awhile` schedule constructor.

To continue, the `acond` operation in Accelerate returns one of two Accelerate expressions. Notably, in contrast to `awhile` with condition type `Acc (Scalar Bool)`, the type of the condition for `acond` is `Exp Bool`. In other words, the condition calculation is not embedded in the backend-specific device. As discussed below, the `Compute` constructor represents the execution of non-embedded expressions with Accelerate's interpreter. In a similar fashion to Accelerate's `acond` operation, the `Acond` schedule constructor picks either of two schedules based on a condition expression. Listing 38 depicts the implementation. Because the boolean condition is already computed by the `Compute` constructor, the `Acond` constructor contains a `Scalar` variable in which the boolean is stored. As such, the implementation does not require an intermediate graph execution.

```
executeSeqSchedule env (Acond (Var _ condIdx) exp1 exp2)
  | (Scalar _ cond) <- prj' condIdx env
  = if toBool cond -- convert PrimBool to Bool
      then executeSeqSchedule env exp1
      else executeSeqSchedule env exp2
```

Listing 38: The implementation of the execution step for the `Acond` schedule constructor.

Operation execution The schedule contains two constructors that represent performing calculations. The `Exec` constructor represents the execution of a TensorFlow operation. The `Compute` constructor represents the computation of a regular Accelerate expression. For starters, as mentioned above, for TensorFlow operations, the approach is to combine as many operation executions into a single TensorFlow graph. Therefore, the implementation of `Exec` in the TensorFlow backend should only construct a TensorFlow graph. The constructor implementations for `Awhile` and `Acond` depend on a calculation result. Only for those constructors must the graph be run. To construct a graph with the kernel from the constructor, the corresponding tensor operations from the TensorFlow bindings are used. The constructor also contains a list of input and output buffers that correspond to the operation side effect. Considering the definitions from listing 33, using the buffer indices, the input and output tensor variables in the `IO` monad are retrieved from the environment of type `TensorEnv`. The input variables are used as input for the bindings operation. The output variable is where the newly constructed graph will be stored. For each input tensor of type TensorValue, if its constructor is Build, it can be used directly as an argument of the binding function. Otherwise, if its constructor is Value, the values in the Vector type will be converted back into a graph using the TF.constant operation, which takes in a list of values. The desugared schedule may contain an execution instruction for the following operations:

- All operations from table 1 are represented by a constructor in TensorKernel. Each constructor is implemented using the binding operation listed in table 1. Each kernel can be directly mapped to the operation listed in the table. However, for boolean operators, Accelerate uses the type Word8 to represent booleans, which is technically equivalent. In TensorFlow, booleans are typed with Haskell's Bool type. Because the values are the same, the graph construction input can simply cast the Bool type using the TF.cast operation. Inversely, using TF.cast, the return type of Bool is cast to Word8. For instance, the operation that corresponds to TensorLogicalAnd is `(\x y -> TF.cast (TF.logicalAnd (TF.cast x) (TF.cast y)))`. To cast Bool to Word8 requires the Bool to be 1 byte. While testing, no errors appeared to occur. As such, the cast is assumed to be correct.

- The TensorConstant kernel corresponds to TConstant from TensorOp. It represents the creation of a new tensor of a given shape `sh` that is filled with a given constant `a`. To implement it, the bindings' operation `fill` is used to construct a graph with a single tensor of shape `sh` with all elements set to `a`. The constructor contains a single buffer that contains a De Bruijn index. Using the tensor environment TensorEnv, the constructed graph is stored in the corresponding variable in the IO monad using the `writeIORef` operation. To illustrate, listing 39 demonstrates the implementation. Moreover, schedule operations only contain a side effect, and do not return a value. As such, the implementation only modifies the state and the output type is `()`. Furthermore, the type of the buffer index is parameterised over `Buffer a`. Correspondingly, when retrieving the tensor element from the environment, the constructor is assumed to be Tensor, which, in contrast to Scalar is also parameterised over `Buffer a`.

```
executeKernel :: TensorEnv env -> TensorKernel env
              -> TF.Session (TensorElements ())
executeKernel env (TensorConstant (TensorArg shR shVars st (Var _ idx)) s)
  | Tensor _ ref <- prj' idx env
  = do
  sh <- liftIO $ getShape env shR shVars
  liftIO $ writeIORef ref $
    Build (TF.Shape sh) $
      TF.fill (TF.vector sh) (TF.scalar (toType64' st s))
  return TupRunit
executeKernel _ (TensorConstant _ _) = error "impossible"
```

Listing 39: The implementation of the execution step for the TensorConstant kernel.

- The TensorId kernel (TId from TensorOp) corresponds to Haskell's identity operation `id`. In the implementation, it simply sets the output variable to the value of the input variable.

- As discussed in section 3.2.2, the TVar constructor from TensorOp is introduced to retrieve values from the Parameter constructor of array instructions. As discussed below, these values are calculated using the Compute schedule constructor. To use this value, the corresponding TensorVar kernel implementation should retrieve it from the environment, and wrap the result in a new tensor. Correspondingly, listing 40 depicts the implementation.

- The TensorWhere kernel corresponds to the TWhere constructor from TensorOp. As stated in section 3.2.1, while desugaring, all inputs for operations dealing with shapes are converted to be one-dimensional. The side effect of TWhere is `In DIM1 a -> Out DIM1 Int -> ()`. However, the corresponding TF.where operation from TensorFlow returns a tensor of a higher dimension. Moreover, given a tensor of booleans or integers, it returns a tensor of one higher dimension of which its dimensions incorporate the indices of all the input elements that were true or greater than 0. As such, with a one-dimensional tensor of shape $[n]$ as input, the TF.where operation returns a tensor of shape $[n, 1]$ as output. However, the side effect assumes the output is of a one-dimensional shape (DIM1). As such, the result should be flattened. With index -1, the

`TF.reshape` operation from the bindings flattens an input. The operation for `TensorWhere` is `(\x -> TF.reshape (TF.where' x) (TF.vector [-1 :: Int64]))`.

- The `TensorGather` kernel (`TGather` from `TensorOp`) corresponds directly to the `gather` method from TensorFlow. Given a tensor of shape `sh`, and a tensor with indices of the first input, the `gather` operation returns a tensor with only those elements referenced by the index tensor. Because the operation uses indices, the side effect of `TGather` from `TensorOp` is parameterised over only one-dimensional input and output buffers. Interestingly, in contrast to the shape $[n, 1]$ of the out output of `TF.where`, the input of `TF.gather` must be of shape $[n]$. Therefore, without any shape conversion, the one-dimensional buffers can be used directly as input and output for the `TF.gather` operation.
- The `TensorCast` kernel (`TCast` from `TensorOp`) corresponds directly to the `TF.cast` method from TensorFlow. Which simply casts a tensor of type `a` as a tensor of type `b`. Similar to Haskell's `unsafeCoerce`, the `TF.cast` method does not specifically take in the types as input.

```
executeKernel env (TensorVar arg (Var _ idx))
  | Scalar _ s <- prj' idx env  = executeKernel env (TensorConstant arg s)
executeKernel _ (TensorVar _ _) = error "impossible"
```

Listing 40: The implementation of the execution step for the `TensorVar` kernel.

- The `TensorSelect` kernel (`TSelect` from `TensorOp`) represents the `select` operation from the TensorFlow bindings. Given a tensor of booleans and two input tensors, it returns a tensor where each element is selected from the two input tensors. Specifically, given a position, if the boolean at the position of the boolean tensor is true, the element at the position of the first input tensor is selected. If it is false, the element at the position of the second input tensor is selected. Similar to the implementation of boolean operation mappings in table 1, the `Word8` type of the boolean input can be cast to `Bool` using the `TF.cast` operation. The operation corresponding to `TensorSelect` is `(\x y z -> TF.select (TF.cast x) y z)`.
- Finally, the `TensorScatter` kernel (`TTensorScatter` from `TensorOp`) corresponds to the tensor scatter operations from the bindings. Based on the `ScatterFun` parameter, respectively for add, min, max, sub and update from listing 28, the kernel corresponds to the TensorFlow operations `TF.tensorScatterAdd`, `TF.tensorScatterMin`, `TF.tensorScatterMax`, `TF.tensorScatterSub` and `TF.tensorScatterUpdate`. As elaborated in section 3.2.1, given a tensor of indices, the tensor scatter operations update a tensor by combining the elements of another tensor. Given an input tensor of shape $[n]$, the input of indices must be of shape $[n, 1]$. This corresponds to the output shape requirement $[n, 1]$ of `TF.where` but differs from the input shape requirement $[n]$ of `TF.gather`.

The schedule constructor `Compute` is used for computations that should not be embedded on the backend-specific device. For instance, as elaborated above, the conditional expression of `Acond` is not embedded. Such calculations are not performed by TensorFlow but by the regular Accelerate interpreter. Because this yields a regular Haskell value and is not a TensorFlow execution, the result type is not a graph. Therefore, the values do not need to be stored in the `IO` monad and can be used directly as input for any subsequent calculation, which is what the `Scalar` constructor is for. Listing 41 illustrates the corresponding implementation. The `evalExpM` method uses Accelerate default interpreter to perform the expression on the host. This method is implemented by the default backend. To retrieve array values from the TensorFlow backend, the method `evalArrayInstr` depicted by listing 42 implements the array indexing instructions `Index` and `Parameter`. As discussed in section , `Index` retrieves a value from an array, and `Parameter` retrieves a scalar value from the Accelerate program. To implement `Index` on the host device for a `TensorValue` on the TensorFlow backend, the value is retrieved by accessing the tensor. If the tensor consists of a constructed graph, the graph is executed and the value is retrieved from the result tensor. For `Parameter`, the values are stored with the `Scalar` constructor. These values are already stored on the host device.

```
executeSeqSchedule :: TensorEnv env -> SeqSchedule TensorKernel env t
                   -> TF.Session (TupR TensorElement t)
executeSeqSchedule env (Compute expr) = do
  value <- evalExpM expr $ EvalArrayInstr $ evalArrayInstr env
  -- return as TupR TensorElement using Scalar constructors
  return $ toTensorElements (expType expr) value
```

Listing 41: The implementation of the execution step when for the `Compute` constructor of the schedule.

```
evalArrayInstr :: TensorEnv env -> ArrayInstr env (s -> t) -> s -> TF.SessionT IO t
evalArrayInstr env (Index (Var _ idx)) s     = case prj' idx env of
  Scalar _ _ -> error "impossible"
  Tensor st ref -> do
    tensorV <- liftIO $ readIORef ref
    case tensorV of
      Build sh build -> do vec <- TF.runSession $ TF.run build
                           liftIO $ writeIORef ref $ Vector sh vec
                           return $ fromType64' st $ vec S.! s
      Vector _ vec -> return $ fromType64' st $ vec S.! s
      Nil _ -> error "impossible"
evalArrayInstr env (Parameter (Var _ idx)) _ = case prj' idx env of
  Scalar _ e -> return e
  Tensor {} -> error "impossible"
```

Listing 42: The implementation for the `evalArrayInstr` method, which implements tensor indexing for the host device.

4 Results

This section aims to answer each of the research questions listed in section 1.4. Section 3 provides a detailed depiction of the model and implementation of the new TensorFlow backend and mentions its limitations. The answers are based on the given limitations and results from executing Accelerate programs with the new backend.

4.1 Embedding Accelerate instructions in TensorFlow graphs

With the existing compilation pipeline in Accelerate, the first research question stated in section ?? explores the possibility of running Accelerate programs in TensorFlow. As such, section 3 models and implements a new TensorFlow backend to desugar Accelerate programs into TensorFlow graphs, and execute them in TensorFlow. Section 3 also denotes some of the limitations. This section further elaborates on these limitations and depicts which parts of the Accelerate language remain unsupported.

To compile Accelerate into a hardware-specific language, a backend should provide compilation instructions for each component in the language. To demonstrate the compilation capability of a backend, it should ideally be tested for completeness and correctness. Accelerate does not feature a test suite to test these properties by default. Each component is listed in the documentation of Accelerate [2]. As such, this thesis defines a test suite with simple test cases for each component listed in the documentation. Some listings in the documentation have usage examples. For each listing, if there are examples, the examples are included as test cases in the test suite. If there are no examples, the suite includes a simple Accelerate program which addresses the listed method. By indicating the success rate of compiling and executing programs for each component in Accelerate, running the test suite provides a measure of the new backend's completeness.

To test for correctness, the suite should ideally include an elaborate testing strategy for each component. However, to limit the scope of this thesis, the suite does not define such strategies. Still, the suite verifies the results by comparing them to the results of using Accelerate's default interpreter backend. Assuming the interpreter results are correct, the suite can verify the results from the TensorFlow backend. Accordingly, each test case is constructed to pass if there are no errors for desugaring and execution and the result matches the result from the interpreter. If either is not the case, the test case will fail.

In total, the suite contains 260 tests to test 167 components. Due to type errors, missing packages, etc., not all components from the documentation could be implemented. Table 3 depicts a list of all 33 untested components. Table 2 depicts an overview of the results after running the test suite. In total, 179 test cases passed. Compilation and execution were found to be supported for 113 operations, which is 68% of the total number of components tested. The paragraphs below elaborate on why some of the cases failed.

Limitations As discussed in sections 2.2 and 3, the new backend implements each component and executes them by constructing and running graphs in TensorFlow. Briefly distinguished in section 3, with Accelerate's compilation pipeline and the new TensorFlow backend, there still are some limitations. As a result, as depicted in table 2, not all components were found to be supported. The failed tests can be explained by the following limitations:

Unsupported Scalar Expressions For starters, as shown in table 1, most primitive scalar expressions from Accelerate are supported in TensorFlow, but not all of them. As truncation to 0 is not supported in TensorFlow, the case for `truncate` failed.

Conditional Array Indexing As depicted in section 3, Accelerate's if-then-else constructor is implemented using TensorFlow's `TF.select` operation. Given a tensor of booleans, it selects elements from two other input tensors. Both input tensors are fully computed before being used as input in the graph of `TF.select`. This is different from Accelerate's semantics, which assumes that only selected array elements are evaluated. As a result, programs that use conditional array indexing may fail. For instance, when an array should be indexed in a loop while an index is in bounds, the resulting `TF.select` graph contains two input tensors for both cases. However, because both cases are computed in the graph, an out-of-bounds error may be thrown. Components in Accelerate such as folds, scans, etc. are implemented using such a construction. As a result, all cases for the categories enumeration, concatenation, filtering, folding, scans, segmented scans and stencils failed. Finally, because of this issue, as stated in section 3, the desugaring implementation does not implement `Case` and `While`, which caused some cases to fail for the categories segmented reduction, real and numeric. The example used to test `compute` also featured such a construction. As a result, the test case failed. However, other usages of the `compute` may instruction still work. Therefore, it is not marked as unsupported in table 2.

Unsupported Types Operations in TensorFlow may require types different from their corresponding Accelerate operations. In most cases, this issue can be resolved by casting the types to an equivalent representation, such as `Int64` for `Int`. However, in some cases, the desired TensorFlow operation may not support the use of types equivalent to the corresponding Accelerate operation. For instance, most TensorFlow operations do not support the type `Word32`, which can not be cast to any other type. For this reason, tests failed in the categories of real and conversion.

4.2 Difference in performance

The second research question stated in section 1.4 investigates the performance of the new backend. As mentioned often in this research, the backend was constructed mainly to investigate the possibility of conversion, without much focus on performance improvements. As such, optimisations such as fusion and improved scheduling are left out of scope. Yet, to provide a complete overview of the new backend, and to aid future work, some performance metrics are nice to have. As such, each successfully passing test case in section 4.1 is also benchmarked with the TensorFlow backend and Accelerate's default interpreter. The benchmarks are performed using the Criterion library for Haskell [16]. Each benchmark measures the time between when the `run` operation from Accelerate is called and the result is returned. As such, each step of the pipeline is included. The benchmarks were performed on a machine as indicated by table 5. Table 4 depicts the mean runtimes for each category in the Accelerate documentation. The mean runtime is calculated by computing the mean time of all benchmarks in the category. Finally, the mean runtimes for the TensorFlow and Interpreter backends are respectively 41.77 and 6.93 milliseconds.

Statistical significance To test the statistical significance of the performance difference for both backends on the same machine, an independent t-test was performed. Moreover, the 0-hypothesis $\mathcal{H}_0$

Category	**Operations**	**Passed tests**	**Support**
Introduction	use, unit	5/5	2/2
Initialisation	generate, fill	4/4	2/2
Enumeration	enumFromN, enumFromStepN	2/2	2/2
Concatenation	`(++)`, concatOn	0/3	0/2
Expansion	*expand*	0/1	0/1
Composition	`(?|)`, acond, awhile, if-then-else	6/6	4/4
Execution	`(>->)`, compute	1/2	2/2
Indexing	indexed, map, imap, zipwith, zipwith1-9, iZipWith, iZipWith1-9, zip, zip1-9, unzip, unzip1-9	34/34	11/11
Modifying arrays	reshape, flatten, replicate, rep0, rep1	7/7	5/5
Extracting subarrays	slice, sl0, sl1, init, tail, take, drop, slit, initOn, tailOn, takeOn, dropOn, slitOn	26/26	13/13
Permutations	permute, scatter, backpermute, gather, reverse, transpose, reverseOn, transposeOn	11/11	8/8
Filtering	*filter, compact*	0/2	0/2
Folding	*fold, fold1, foldAll, fold1All*	0/8	0/4
Segmented reductions	*foldSeg, fold1Seg, fold1Seg*	0/4	0/3
Specialised reductions	*all, any, and, or, sum, product, minimum, maximum*	0/18	0/8
Scans	*scanl, scanl1, scanl', scanr, scanr1, scanr', prescanl, postscanl, prescanr, postscanr*	0/20	0/10
Segmented scans	*scanlSeg, scanl1Seg, scanl'Seg, scanrSeg, scanr1Seg, scanr'Seg, prescanlSeg, postscanlSeg, prescanrSeg, postscanrSeg*	0/10	0/10
Stencils	*stencil*	0/1	0/1
Eq, Ord	`(==)`, `(/=)`, `(<)`, `(>)`, `(<=)`, `(>=)`, min, max, compare	18/18	9/9
Num	`(+)`, `(-)`, `(*)`, negate, abs, signum, quot, rem, div, quotRem,divMod	14/14	11/11
Rational	toRational	1/1	1/1
Fractional	divide, recip	2/2	2/2
Floating	sin, cos, tan, atan, sinh, cosh, tanh, asinh, aconh, exp, sqrt, log, `(**)`, logBase	14/14	14/14
RealFrac	*truncate*, round, ceiling, floor, div, mod	5/6	5/6
Real	floatRadix, floatDigits, floatRange, *decode*, encode, *exponent*, *significand*, *scale*, isNan, isInfinite, *isDenormalized*, *isNegativeZero*, isIEEE, atan2	8/14	8/14
Scalar	constant, fst, snd, uncurry, cond	5/5	5/5
Logical	and, or, not	10/10	3/3
Numeric	subtract, even, odd, *gcd*, *lcm*, *(^)*, *(^^)*	3/7	3/7
Shapes	index0	1/1	1/1
Conversion	*ord, chir*, boolTo	1/3	1/3
Plain arrays	fromList	1/1	1/1
Total		**179/260**	**113/167**

Table 2: Results from testing each listing in the Accelerate documentation. The subjects match the categories in the documentation. The third column denotes the number of passed tests compared to the total number of test cases. The fourth column denotes the number of supported operations compared to the total number of operations. In the second column, unsupported operations are highlighted in italics.

Untested methods
pi, properFraction, divMod, afst, asnd, curry, while, iterate, sfoldl, index0-3, unindex1-3, indexHead, indexTail, toIndex, fromIndex, intersect, bitcast, foreignAcc, foreignExp, arrayRank, arrayShape, arraySize, arrayReshape, indexArray, linearIndexArray, fromFunction, fromFunctionM, toList, countLeadingZeros, countTrailingZeros

Table 3: The table above lists all of the 33 components listed in the Accelerate documentation that were not tested, nor included in the results of table 2.

assumes that both sets of runtimes for the TensorFlow and Interpreter backend are equal. To reject this hypothesis, the p-value must be below $\alpha = 0.05$. After conducting the test, a p-value of $0.0001 < \alpha$ was obtained. As such, $\mathcal{H}_0$ is rejected. In other words, the performance difference is found to be statistically significant.

Category	TensorFlow Mean RunTime (ms)	Interpreter Mean RunTime (ms)
Introduction	5.34	4.33
Initialisation	8.97	4.78
Enumeration	13.54	5.42
Composition	8.55	5.06
Execution	11.96	9.28
Indexing	169.64	7.76
Modifying arrays	16.99	6.24
Extracting subarrays	34.03	7.65
Permutations	32.54	7.43
Eq, Ord	10.09	6.49
Num	8.09	6.52
Rational	7.43	8.29
Fractional	7.6	6.69
Floating	6.84	6.49
RealFrac	7.76	6.49
Real	7.07	6.5
Scalar	26.66	6.93
Logical	64.76	9.35
Numeric	9.65	6.24
Shapes	3.72	3.93
Plain arrays	6.15	6.5
Conversion	4.9	4.41
Mean	**41.77**	**6.93**

Table 4: Results from benchmarking every successful test case from table 2. For every test case, the criterion library outputs a mean runtime result. The runtime columns in the table denote the mean of the means of all tests in each category.

Operating System	Windows 11 Pro 64-bit
CPU	AMD Ryzen 9 5900X 12-Core Processor @ 3.70 GHz
RAM	32.0GB Dual-Channel DDR4 @ 1306MHz
Motherboard	ASUSTeK COMPUTER INC. ROG STRIX B550-F GAMING (WI-FI) (AM4)

Table 5: The hardware specifications of the device the benchmarks were run on.

4.3 A friendly programming language to construct TensorFlow graphs

As stated in section 1.4, the final research question concerns providing a friendly programming language to construct TensorFlow graphs with Accelerate. Currently, in Haskell, the way to construct and execute TensorFlow programs is by using the TensorFlow bindings. The library is limited to first-order operations. Difficulties may emerge from the lack of support and documentation. As such, using these bindings requires a thorough preliminary understanding of TensorFlow. With the new Accelerate backend from section 3, Accelerate programs can be executed with TensorFlow. As a result, the Accelerate language can be used to construct graphs instead. This alleviates the need to study TensorFlow before being able to use it. With the TensorFlow bindings as a first-order language, constructing graphs may be tedious. As shown in example 10, the second-order nature of Accelerate programs may pose a more programmer-friendly option to quickly express calculations in TensorFlow. Nevertheless, when opting for the new backend to construct graphs, there are some downsides the user should take into account. With the limitations mentioned in section 4.1, the user should remember that not every Accelerate program is accepted by the backend. And, as discussed in section 4.2, the graphs may not be optimised. To take into account the backend's limitations, the user still needs an understanding of TensorFlow. For Haskell programmers unfamiliar with both languages, it may be more intuitive to use the TensorFlow bindings, as using the backend would require an understanding of both Accelerate and the characteristics limiting its use. Still, considering that plenty of Accelerate programs are supported, the backend may pose as a friendly option to construct graphs for those already experienced with Accelerate.

Example 10: When constructing graphs, the TensorFlow bindings only offer first-order operations. As a result, constructing TensorFlow graphs may be tedious. For illustration, consider the graph to compute the quadratic formula below. Each operation of the formula is constructed as a node in the graph.

```haskell
quadratic :: (OneOf TFFloat a, Num a)
          => TF.Tensor v a
          -> TF.Tensor v a
          -> TF.Tensor v a
          -> (TF.Tensor TF.Build a, TF.Tensor TF.Build a)
quadratic a b c = (
  TF.div
    (TF.add
      (TF.neg b)
      (TF.sqrt (TF.sub (TF.mul b b) (TF.mul (TF.scalar 4) (TF.mul a c)))))
    (TF.mul (TF.scalar 2) a),
  TF.div
    (TF.sub
      (TF.neg b)
      (TF.sqrt (TF.sub (TF.mul b b) (TF.mul (TF.scalar 4) (TF.mul a c)))))
    (TF.mul (TF.scalar 2) a)
  )
```

In contrast to the bindings, Accelerate programs allow second-order operations. For instance, using the `zipWith3` operation below, the formula can be implemented with one or two lines of code. As an added benefit, compared to the above, the implementation below is more readable.

```haskell
quadratic :: (Num a, Data.Array.Accelerate.Floating a)
          => Acc (Vector a)
          -> Acc (Vector a)
          -> Acc (Vector a)
          -> (Acc (Vector a), Acc (Vector a))
quadratic x y z = (qAdd x y z, qSub x y z)
  where qAdd = zipWith3 (\a b c -> -b + sqrt (b * b - 4 * a * c) / 2 * a)
        qSub = zipWith3 (\a b c -> -b - sqrt (b * b - 4 * a * c) / 2 * a)
```

5 Conclusion

With the results in section 4.1, this research shows that second-order Accelerate programs can successfully be compiled into first-order computational graph operations. However, there are limitations. The limitations indicate the adequacy of the GADTS in the Accelerate pipeline to construct first-order graphs for TensorFlow, which constitutes the main contribution of this research. With a performance measure and an estimate of programmer-friendliness as motivation, future research may aim to optimise the new backend and solve problems imposed by the characterised limitations. Elaborating on the new backend requires an extensive illustration of the Accelerate compilation pipeline, which is provided in sections 2 and 3. In doing so, this research also contributes a detailed overview of the pipeline to future researchers looking to investigate backends for Accelerate.

With the new TensorFlow backend depicted by section 3, this research aimed to answer the research questions from section 1.4. In short, the aim was to investigate the possible extent of compiling and executing Accelerate programs as TensorFlow graphs, compare the difference in performance, and, in doing so, establish a friendly programming language to construct graphs in TensorFlow. The paragraphs below discuss the interpretation and implication of these results, and how further studies may look to improve on them.

Limited support With the results depicted in section 4.1, the new backend is found to be covering 68% of Accelerate's components. In other words, Accelerate programs comprised of these components can be run with TensorFlow. With this result, this research shows that such second-order programs can successfully be compiled into corresponding first-order computational graph operations. However, as not all components are supported, the result does not yet imply the feasibility of all Accelerate programs. As a result, users must take the limitations into account when using the TensorFlow backend.

The compilation result is assumed to be correct, but this property is not extensively tested. Though, with the computations directly mapped to TensorFlow operations, one might argue that the correctness may also be adopted by assuming the TensorFlow operations are correct. Still, this assumption does not incorporate technical differences such as rounding behaviour. Future work may propose a more extensive approach to test backend correctness.

The support rate is induced by limitations from the Accelerate compilation pipeline and characteristics of the TensorFlow backend. Most unsupported components are characterised by the limitation of conditional array indexing, which fails due to how if-then-else operations are implemented. Accelerate's semantics assume that only selected array elements are evaluated, while TensorFlow's `TF.select` operation fully computes both tensors before selecting elements from them. On one end, one might argue that these semantics from Accelerate are bold, especially when potentially supporting compilation to a plethora of backends. However, the evaluation style is in line with Haskell's method of lazy evaluation, where each value is only evaluated when absolutely necessary. And, the semantics efficiently implement conditional instructions for parallel executions on arrays. Still, when embedding computations on other hardware, such Haskell-like characteristics may be non-existent. Future work may reveal there still is a way to circumvent this issue. Speculatively, the conditional operator may be desugared into an if-then-else constructor in the schedule, which allows for intermediate graph execution. Then, after separating each tensor element in a separate graph, only the graphs that ought to be selected can be evaluated. Still, to prevent evaluation, each element probably has to be isolated in a separate graph from the start, which likely removes any data-parallel advantage TensorFlow may have. Alternatively, the default desugaring implementations of the unsupported methods may be overhauled to avoid using the conditional statement. This may require automatic program analysis to effectively eliminate occurrences of conditional array indexing. Finally, perhaps there still exists some desugared sequence of TensorFlow operations for the if-then-else component that remained uncovered in this research.

Performance Section 4.2 depicts the results after benchmarking all passing test cases for both the TensorFlow and Accelerate's default interpreter backend. Respectively, the mean runtimes for the TensorFlow and interpreter backends are 41.77 and 6.93 milliseconds. The performance difference was found to be statistically significant. The results show that, for the test cases, the TensorFlow backend is slower than the default interpreter on a CPU. As such, when looking for a performant execution of running

Accelerate programs on a CPU, the TensorFlow backend is not the best option.

The main goal of this thesis requires a working backend. Optimisations such as fusion, scheduling improvements, alternative definitions, etc. were left out of scope. The TensorFlow backend has plenty of overhead. Each program instruction is desugared, graphs are constructed, executed, etc. With the TensorFlow backend, some Accelerate programs desugar into extensive schedules. As the default interpreter is run in Haskell, it has much less overhead. Hence, without performance improvements, the worse performance of the TensorFlow backend is to be expected. Future work may aim to improve performance by providing optimisations for the TensorFlow backend, such as reducing schedule sizes by introducing a better scheduling algorithm.

The TensorFlow bindings for Haskell only support first-order operations. Perhaps future work may find that the workload of converting a second-order into a first-order language will always result in longer runtimes. In contrast to the bindings in Haskell, the Python programming language does support higher-order operations for TensorFlow. Still, both libraries construct and execute graphs. As such, the Python library presumably performs a similar conversion under the hood too. With the Python library officially supported by TensorFlow, perhaps it features a better-optimised conversion method compared to the one presented in this thesis. As such, future work may look into generating Python code to run the graphs. This may cause a loss in type safety but may provide better results. Alternatively, the Haskell bindings could be enhanced to adopt a similar conversion method to allow higher-order TensorFlow operations. In doing so, the number of execution instructions in the backend schedule may be reduced to address higher-order functions, which may alleviate some of the compilation overhead.

For the TensorFlow backend, the longest mean runtime was 169.64 milliseconds. As such, the benchmarks may indicate that most Accelerate programs can be run with TensorFlow without taking an indefinite amount of time. The benchmarked test cases focussed on the completeness of supporting each component and did not investigate every edge case. Speculatively, some Accelerate programs may still desugar into very large unoptimised schedules, perhaps too big to process. Future work may aim to characterise such cases and optimise the scheduling process.

The benchmarks were performed on a CPU. The TensorFlow bindings also feature running TensorFlow on a GPU. When running the test cases on a GPU, plenty of kernel errors were raised. The main goal of this thesis is to enable compilation and execution. To limit the scope, optimisations are often left out. As such, GPU tests were not included. However, GPUs often have a plethora of cores dedicated to executing large-scale parallel computations. Compared to a CPU, TensorFlow shows an increase in performance when training complex neural networks [21] on a GPU. The Accelerate interpreter backend can not run on a GPU, but the LLVM backend can. Future work may provide different performance results when the backends are compared on a GPU.

The benchmarks measure the time between the call of the `run` operation and the return of the result. This includes every step in the pipeline. Because the second-order components of Accelerate are compiled into first-order TensorFlow operations, compilation introduces plenty of overhead. The benchmark does not indicate whether the execution of a compiled graph is faster than its corresponding program from the Accelerate interpreter backend. However, considering that the TensorFlow backend uses intermediate graph execution, it becomes hard to identify where benchmarking should start. Also, with the way the Accelerate compilation pipeline is set up, the steps of the compilation are not cached and therefore re-performed for every call of the `run` operation. Knowledge about execution times does not matter when the pipeline will always perform desugaring. As such, the benchmarks may just be inclusive enough.

A programmer-friendly language As elaborated in section 4.3, with the new backend, Accelerate can be used as a programming language to construct and execute graphs in TensorFlow. Compared to using the first-order TensorFlow bindings in Haskell, the second-order components of Accelerate are often more readable and more comfortable to use. As a result, Accelerate may provide a more user-friendly programming language for TensorFlow in Haskell.

The result mainly focuses on comparing TensorFlow languages in Haskell. However, TensorFlow also has libraries in other languages, such as Python. The Python library supports higher-order operations, provides extensive documentation and is actively maintained. Still, compared to Python, the main benefit of using Accelerate for TensorFlow is that it can run Accelerate programs without requiring the user to have any knowledge about TensorFlow. In contrast, the Python library is specifically designed for TensorFlow. Data-parallel operations such as `generate` require more than one TensorFlow operation to be implemented. The Accelerate backend for TensorFlow provides such conversions automatically.

The limitations provided in section 4.1 mean that compilation may fail for programs that use unsupported components. As a result, the user is required to know the limitations when writing Accelerate programs to run with TensorFlow. This may diminish the programmer friendliness of the language for TensorFlow. What's more, pre-existing programs in Accelerate may have to be refactored to no longer use the unsupported components. To improve programmer friendliness, future work may need to aim to reduce users' need to take into account the backend's limitations. By increasing the limited support, the limitations may become less interfering. If the support rate can not be improved, perhaps the use of the unsupported components can be restricted by introducing type constraints. Doing so at least lets the user know about their use of unsupported components in the program before running ut. By directly pointing out the wrong component, the use of the backend with the limitations may become more intuitive.

References

[1] Martin Abadi et al. "TensorFlow: a system for large-scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[2] *Accelerate 1.3.0.0 Documentation*. URL: https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html (visited on 05/12/2023).

[3] *Accelerate QuickSort Example*. URL: https://github.com/AccelerateHS/accelerate-examples/tree/master/examples/quicksort (visited on 12/23/2022).

[4] *Accelerate Repository*. URL: https://github.com/AccelerateHS (visited on 12/23/2022).

[5] *Accelerate TensorFlow*. URL: https://github.com/larsvansoest/accelerate-tensorflow (visited on 12/23/2022).

[6] *API Documentation — TensorFlow v2.11.0*. URL: https://www.tensorflow.org/api_docs (visited on 12/12/2022).

[7] Lars Bergstrom et al. "Data-only flattening for nested data parallelism". In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 81–92.

[8] Guy E Blelloch. *Vector models for data-parallel computing*. Vol. 2. MIT press Cambridge, 1990.

[9] Guy E Blelloch and Gary W Sabot. "Compiling collection-oriented languages onto massively parallel computers". In: *Journal of parallel and distributed computing* 8.2 (1990), pp. 119–134.

[10] Manuel MT Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 2011, pp. 3–14.

[11] Robert Clifton-Everest et al. "Embedding foreign code". In: *International Symposium on Practical Aspects of Declarative Languages*. Springer. 2014, pp. 136–151.

[12] Robert Clifton-Everest et al. "Streaming irregular arrays". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. 2017, pp. 174–185.

[13] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[14] Peter Goldsborough. "A tour of tensorflow". In: *arXiv preprint arXiv:1610.01178* (2016).

[15] Jonathan C Hardwick. "Porting a vector library: a comparison of MPI, Paris, CMMD and PVM". In: *Proceedings Scalable Parallel Libraries Conference*. IEEE. 1994, pp. 68–77.

[16] *haskell/criterion: A powerful but simple library for measuring the performance of Haskell code*. URL: https://github.com/haskell/criterion (visited on 09/07/2023).

[17] Justus Henneberg and Felix Schuhknecht. "RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing". In: *arXiv preprint arXiv:2303.01139* (2023).

[18] *How to Manipulate Data Structure to Optimize Memory Use on 32-Bit Intel Architecture*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.html (visited on 10/07/2023).

[19] Gabriele Keller et al. "Vectorisation avoidance". In: *Proceedings of the 2012 Haskell Symposium*. 2012, pp. 37–48.

[20] Gabriele Cornelia Keller. "Transformation-based implementation of nested data parallelism for distributed memory machines". PhD thesis. Technical University of Berlin, Germany, 1999.

[21] Eric Lind and Ävelin Pantigoso Velasquez. *A performance comparison between CPU and GPU in TensorFlow*. 2019.

[22] Ben Lippmeier et al. "Work efficient higher-order vectorisation". In: *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*. 2012, pp. 259–270.

[23] Trevor L McDonell et al. "Optimising purely functional GPU programs". In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 49–60.

[24] Trevor L McDonell et al. "Type-safe runtime code generation: accelerate to LLVM". In: *ACM SIGPLAN Notices* 50.12 (2015), pp. 201–212.

[25] *Module : tf — TensorFlow v2.11.0 (Python documentation)*. URL: https://www.tensorflow.org/api_docs/python/tf (visited on 12/12/2022).

[26] *Nvidia (3000 series) Ampere Architecture Support - Update TensorFlow to v2.4.0 · Issue #280 · tensorflow/haskell*. 2022. URL: https://github.com/tensorflow/haskell/issues/280 (visited on 12/12/2022).

[27] *Release Tensorflow 2.4.0 · tensorflow/tensorflow*. 2022. URL: https://github.com/tensorflow/tensorflow/releases/tag/v2.4.0 (visited on 12/12/2022).

[28] Bo Joel Svensson et al. "Converting Data-Parallelism to Task-Parallelism by Rewrites". In: *FHPC '15: The 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*. ACM, Sept. 2015, pp. 12–22.

[29] *TensorFlow Haskell Bindings*. Version 0.3.0.0. 2019. URL: https://github.com/tensorflow/haskell.

[30] *TensorFlow in other languages*. URL: https://github.com/tensorflow/docs/blob/82a6055/site/en/r1/guide/extend/bindings.md (visited on 12/12/2022).

[31] *TensorFlow.GenOps.Core (Haskell bindings documentation)*. URL: https://tensorflow.github.io/haskell/haddock/tensorflow-core-ops-0.3.0.0/TensorFlow-GenOps-Core.html (visited on 12/12/2022).

[32] *TensorFlow.Ops (Haskell bindings documentation)*. URL: https://tensorflow.github.io/haskell/haddock/tensorflow-ops-0.3.0.0/TensorFlow-Ops.html (visited on 12/12/2022).