Bags in Accelerate

Tristan van der Hoogte - 6874916

April 23, 2025

Daily supervisor: Ivo Gabe de Wolff
First supervisor: Gabriele Keller
Second supervisor: Wouter Swierstra
Master Computing Science, Utrecht University

Abstract

Many data-parallel languages provide their interface using arrays, which allow for efficient execution. However, some algorithms, in particular graph algorithms, are better expressed in terms of sets or bags, where the order of elements does not matter. Besides expressiveness, using sets or bags may be more efficient than using arrays as it is not necessary to maintain a specific order. Recognizing this, this thesis extends Accelerate, a data-parallel array language embedded in Haskell, to handle unordered arrays, referred to as bags. This thesis lays the foundation for bags in Accelerate. We implemented bag combinators in the front-end of the language, and desugar them to array functions halfway in the compiler pipeline. Later work may also extend the back-end to support bag versions natively, to apply optimizations that are specifically possible on unordered data.

To test the effectiveness of these additions, breadth-first search (BFS) and Kahn's algorithm for topological sorting were implemented in both their traditional form and using the new bag-based approach. The results showed that the use of bags simplified the implementation of these algorithms. As a proof of concept for the kind of performance improvements that this research may allow, a unordered version of filter was implemented and compared to the ordered version.

The introduction of bags in Accelerate was found to be an improvement, providing a foundation for further optimizations. The possibility of potential performance enhancements in parallel computing remains uncertain.

Contents

1 Introduction **5**
- 1.1 Accelerate . . . 5
- 1.2 Sets, Bags, and Arrays . . . 7
  - 1.2.1 Sets . . . 7
  - 1.2.2 Bags . . . 7
  - 1.2.3 Arrays . . . 7
- 1.3 Unorderedness . . . 7
- 1.4 Research questions . . . 8
- 1.5 Example . . . 8

2 Related work **9**
- 2.1 Array languages . . . 9
- 2.2 Set languages . . . 9

3 Design **11**
- 3.1 Representation . . . 11
- 3.2 Introduction of bags . . . 11

4 Implementation **12**
- 4.1 Place in pipeline . . . 12
- 4.2 Functions . . . 13
  - 4.2.1 Translation functions . . . 13
  - 4.2.2 Size related . . . 14
  - 4.2.3 Special folds . . . 14
  - 4.2.4 Mapping bags . . . 14
  - 4.2.5 Folding bags . . . 15
  - 4.2.6 Cartesian product . . . 15
  - 4.2.7 Filter bags . . . 15
  - 4.2.8 Intersect . . . 16
  - 4.2.9 Union . . . 16
  - 4.2.10 Subtract . . . 16
- 4.3 Limitations . . . 17

5 Experimental setup **18**
- 5.1 Front-end . . . 18
  - 5.1.1 Breath first search . . . 18
  - 5.1.2 Kahn . . . 19
- 5.2 Back-end . . . 20
  - 5.2.1 Functions . . . 20
  - 5.2.2 Input . . . 21

6 Results **22**
- 6.1 Front-end . . . 22
  - 6.1.1 Combine . . . 22
  - 6.1.2 BFS . . . 22
  - 6.1.3 Kahn . . . 23
- 6.2 Back-end . . . 24

7 Conclusion **25**

A Algorithms **27**
- A.1 Vector BFS 1 . . . 27
- A.2 Bag BFS 1 . . . 28
- A.3 Vector BFS 2 . . . 29
- A.4 Bag BFS 2 . . . 30
- A.5 Vector BFS 3 . . . 31

A.6 Bag BFS 3 32
A.7 Vector BFS 4 33
A.8 Bag BFS 4 34
A.9 Vector Kahn 1 35
A.10 Bag Kahn 1 36
A.11 Vector Kahn 2 37
A.12 Bag Kahn 2 38

1 Introduction

Parallel computing aims to improve performance, and one tool in this domain is Accelerate [CKL+11], a domain-specific language embedded in Haskell [Hut16]. The functional nature of Haskell, its immutability, and the absence of side effects make it particularly amenable to parallelization. Accelerate is an array-based language designed to allow for the creation of parallelized programs in a functional setting. GPUs serve as prime examples of parallel computing, finding applications in diverse fields such as image processing and neural networks. Purely functional embedded array programs, like those developed in Accelerate, align well with parallel hardware, especially GPUs.

Accelerate, as an array language, inherently maintains order for its elements. However, there are cases where the programmer does not require this order. In such situations, if the program still expends effort to track element order by default, it introduces unnecessary computational overhead. Given that parallel hardware is typically selected to achieve optimal performance, removing inefficiencies is prioritized. Therefore, recognizing and addressing unnecessary work related to element order becomes crucial.

This thesis extends Accelerate to handle unordered arrays, treating them as bags. This extension aims to simplify the implementation process for algorithms that do not require ordered data structures. By introducing bags, programmers can more intuitively express algorithms that are naturally set-based or bag-based, without the need to manually manage order. This not only makes the code more readable and maintainable, but also reduces the cognitive load on developers. Additionally, the introduction of bags lays the groundwork for future performance improvements by enabling more efficient parallel computations.

The focus of this thesis is to make additions to the Accelerate front-end, using Haskell's expressive power. Specifically, it will add bags and a user-facing selection of bag based functions, and extend them through the pipeline of the front-end of Accelerate. In addition, a proof of concept of the potential performance improvements is presented.

The thesis will first introduce the basic concepts that the research is based on. Then, existing languages similar to Accelerate or the features that this thesis introduces are listed and analyzed for how they implemented their equivalent features. Decisions on how to implement the desired features are described, followed by implementation details. Finally, the ease of use of the new features is tested through the implementation of specific algorithms, the potential for performance improvements is tested using the proof-of-concept, and conclusions are drawn.

1.1 Accelerate

Accelerate is a functional language embedded in Haskell. It is a combinator language, meaning that the language is only constructed of functions, without (mutable) variables. In Accelerate, the basic units of computation are arrays and expressions. Arrays are the structures over which parallel operations are performed. Expressions are single values. Arrays can be of any dimensionality, with one-dimensional arrays being called vectors, two-dimensional arrays being matrices, and zero-dimensional arrays being scalars.

The following is an example of a line of code that you might find in an Accelerate program.

```
dotProduct a b = sum (zipWith (*) a b)
```

It is an implementation of the dot product, which takes two vectors, multiplies each pair of elements together, and sums them. In this example, `a` and `b` are the aforementioned input vectors. They don't need to be the same length, because `zipWith` will only take the elements up to the length of the shortest vector. More specifically, `zipWith` takes two arrays of the same dimensionality and produces an array whose size is the intersection of the two input arrays. The elements of the output are the result of a user-given function applied to the elements of the input arrays at the corresponding locations. The `sum` function is applied to the output. It sums the values along a single direction, thereby reducing the dimensionality of the array by one. In this case, it turns the one-dimensional vector into a zero-dimensional scalar array.

Accelerate does not perform array computations directly. Instead, it constructs an abstract syntax tree (see Figure 1), making it a deep embedding language by definition. The AST is then passed through various processing steps for efficiency and mapping to the target architecture. This happens

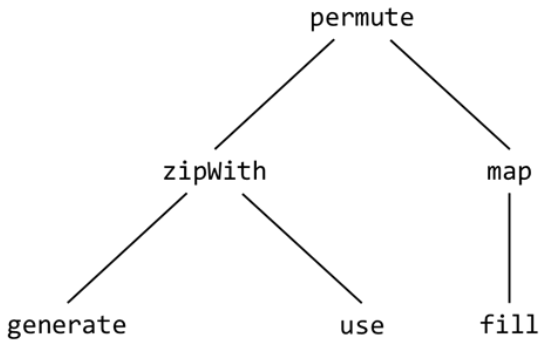

Figure 1: An AST

in the front-end of Accelerate. It is then turned into LLVM [1] code in the back-end. Figure 2 shows the structure of Accelerate. At the time of writing, a new version of Accelerate is in development that changes the structure of Accelerate somewhat. These changes are not reflected in the figure.

The AST is represented by the `Acc` type. Specifically, any Accelerate program will have the type `Acc a`, where `a` is a member of the `Arrays` type class. The `Arrays` consists of the singular `Array` type, and tuples of `Arrays` of up to 15 elements long. The `Array` type has two arguments, usually written as `Array sh a`. Here, `sh` is the shape of the array, that is, its size and dimensionality. The shape can only be a value of the `Shape` type class. In the example, `a` is the type of the content of the array. This can only be a type that belongs to the `Elt` (element) type class. Most basic Haskell types (`Int`s, `Char`s, `Bool`s, etc.) belong to `Elt`. `Elt` exists so that only simple values can be elements of arrays, e.g. lists and user-defined types cannot be used. Any values of the `Shape` type class are also part of `Elt`.

The shape of an array is a list-like structure. However, it is constructed on type level. The "end of list" operator is `Z` and on its own represents a zero-dimensional array, also known as a scalar. The "cons" operator is `a :. b` where a is another element of `Shape`, and `b` is an `Int`. The "cons" represents an array of one dimension higher than the `Shape` it contains. The elements of this "list" are ordinary `Int`s, so the exact size of the array is value level information.

You can construct the AST using Accelerate's interface of functions, such as

```
indexed :: (Shape sh, Elt a) => Acc (Array sh a) -> Acc (Array sh (sh, a))
```

These functions take one or more ASTs (`Acc`s) and produce a new AST. Using these, the user can, of course, define their own functions to construct further ASTs. Accelerate's functions do not only have `Acc`s as arguments. Many functions require single values or functions on single values as arguments. Placing such values or functions on the AST requires that they are contained in a special type, the `Exp a` (expression) type. Here, `a` is again an element of the `Elt` type class. Many basic Haskell functions have `Exp` equivalents, including those of the `Eq` and `Ord` classes and many basic mathematic operations beyond that. Also present are `Exp` versions of booleans and functions on them.

As an example of `Exp`s, this is a function that incorporates them.

```
map :: (Shape sh, Elt a, Elt b) =>
       (Exp a -> Exp b) -> Acc (Array sh a) -> Acc (Array sh b)
```

The function `map` takes a function on `Exp`s and an array of `a`s and produces an array of `b`s. The input and output arrays are of the same shape. The function is applied to each element individually.

[1] LLVM optimizes the code between Accelerate and the target hardware [LA04]

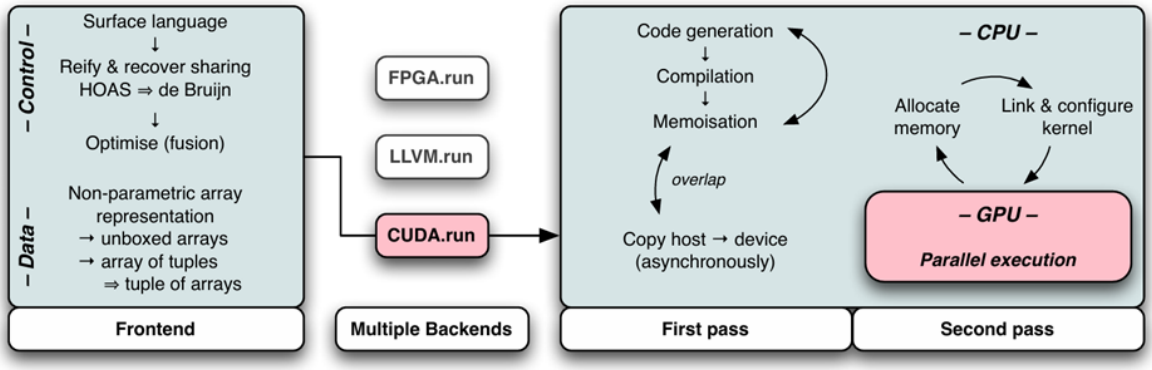

Figure 2: The (old) structure of Accelerate. Figure from [CKL+11]

1.2 Sets, Bags, and Arrays

In this thesis, the terms set, bag, and array ([CLRS22]) are used frequently, since we discuss replacing one by another in certain algorithms. The following explanation is intended to make clear the difference between the three.

1.2.1 Sets

A **set** is a collection of distinct elements, meaning no element can appear more than once. Sets are fundamental in mathematics and computer science for representing collections of unique items. They are inherently unordered. Operations on sets include union, intersection, and difference. For example, the union of two sets contains all elements from both sets, while the intersection contains only the elements present in both sets.

1.2.2 Bags

A **bag**, also known as a multiset, is similar to a set, but allows multiple instances of the same element. The number of times an element appears in a bag is called its multiplicity. Bags are useful in scenarios where the frequency of elements matters, such as counting occurrences of words in a text. Unlike sets, bags do not enforce uniqueness of elements, although they are also unordered.

1.2.3 Arrays

An **array** is an ordered collection of elements, typically of the same type, stored in contiguous memory locations. Arrays allow for efficient access to elements using indices. They are widely used in programming for tasks that require fast data retrieval and manipulation. Arrays can be one-dimensional (like a list) or multi-dimensional (like a matrix). For this thesis, they can also be zero-dimensional, like a single number.

1.3 Unorderedness

This thesis discusses the implementation of the order of an array being irrelevant (to the user). This property is referred to as unorderedness. This is opposed to orderedness, which is currently the only state of Arrays in Accelerate.

Unorderedness might also be referred to as order-irrelevance. The property of being unordered causes the array to be treated as a set, or at least like a bag, where the elements simply do not have an internal order. An array being ordered should not be confused with it being sorted. Because an unordered array does not have an internal order, it will never be sorted. This is unlike an ordered array, which can be sorted or unsorted. An example is seen in Figure 3.

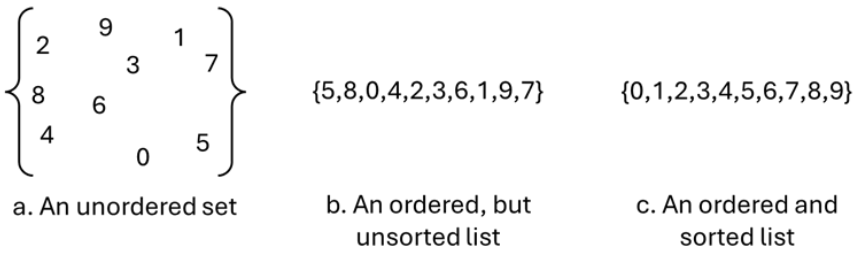

Figure 3: Ordered vs. sorted

1.4 Research questions

The main goal of this thesis is to allow algorithms that do not require its data structures to be ordered to take advantage of this fact. Primarily, this thesis focuses on taking advantage of unorderedness in implementation, trying to give developers the means to simplify the implementation, and thereby the implementation process. Secondarily, this thesis tries to create the opportunity to improve performance using the knowledge of unorderedness. This leads to the following research questions:

- **Representation:** How should unordered arrays be represented to the user?
- **Integration:** How can unordered arrays be integrated.
- **Performance improvement:** What is the degree of performance improvement achieved through these optimizations?

To investigate these questions, bags have been implemented into Accelerate. The implementation can be found via: https://github.com/t-hoogte/accelerate/tree/new-pipeline-bags.

1.5 Example

Take for example the algorithm for breath first search (BFS) [CLRS22] on graphs. This will later be explained in more depth, and will be used to evaluate the results of this thesis. This algorithm creates a frontier, the set of vertices currently being processed, from the previous frontier. This frontier is a list of vertices in the graph that are currently being expanded and therefore have the same distance from the starting vertex. The creation of a frontier goes as follows.

- For each vertex in the previous frontier, find its neighbors.
- For each neighbor, see if it was already found.
- For all neighbors that were not, put them in the next frontier.

The order of the frontier is not relevant in this algorithm. The algorithm contains steps, such as those that require finding something in an array for each element in another array, which are not intuitive to implement using the current features of Accelerate. These kinds of issues in implementation may be able to be resolved by the addition of some form of unorderedness to Accelerate.

In addition, the process of finding a neighbor, seeing if it has already been found, and adding it to the next frontier, can take different amounts of time for different vertices. In most architectures, currently including Accelerate, this means that effort needs to be expended to keep the vertices in the correct order, even if a later vertex finishes earlier. If it is known ahead of time that the order of the output does not matter, as it clearly does in this algorithm, then the ordering effort can be bypassed. The above-mentioned process can be run in parallel, and whichever vertex gets processed first will be put in the next frontier first. The vertices will be put in the next frontier in the order they finish processing. This is an example of the kind of improvement that can be made possible by this research.

2 Related work

Accelerate is an array language, and like many other such languages it uses its structure to allow for efficient parallel computations. With this research Accelerate has been improved with additional set-like features and thereby has joined the much shorter list of set languages. Improvements that are applicable to Accelerate might also be applicable to other array languages. In the following sections, set and array languages are listed.

2.1 Array languages

Array languages, that is, languages that feature arrays and operations that act on entire arrays, are much more common than set languages. In principle, many algorithms that are more naturally expressed as sets or bags are made using arrays. This holds for array languages besides Accelerate as well. Therefore, it could potentially be beneficial for them to implement sets as well.

The following list is not exhaustive.

- Fortran [Bac78].
- APL [FI78].
- GNU Octave [EBH+97].
- R [IG96].
- Julia [BEKS17].
- Futhark [HSE+17].
- IDL [Bow06]

While Matlab supports some bag functions, it is more well known for its array capabilities. The two features do not necessarily interact with each other.

2.2 Set languages

The following is an exhaustive or nearly exhaustive list of languages that operate on sets. The field of set-based languages is underdeveloped.

- SETL [SDDS12] has sets as the basic type of the language. The functions on sets that are available are: union, intersect, difference, size, min/, max/, query.
- CLAIRE [CL98] contains sets as first-class citizens, as well as lists. Both of which can contain multiple different types, but can be forced to contain only one type. The authors [CL98] say "The concept of bag in CLAIRE is the unifier between lists and sets: a collection of objects with possible duplicates and without order." CLAIRE has concatenation on bags (not sets specifically) that produces a list, as well as indexing on bags. On sets it has regular intersection and difference, addition and deletion of an element, length, max, min. It also has the functions exists, some, and forall.
- JavaScript [Cro08] has sets as a build-in object type. This object type has many methods. The ones that are set-specific are: difference, intersection, union, symmetricDifference, isDisjoint-From, isSubsetOf, and isSupersetOf.
- Python [VRD03] has set as a build-in data type. It has a similar set of set-specific methods as JavaScript. They are: difference, intersection, union, symmetric_difference, isdisjoint, issubset, and issuperset.
- SQL [Dat89] operates on database tables as sets. SQL has several join functions to combine tables: join, left join, right join, outer join, and cross join, which is a Cartesian product. The rest of the joins match on a single column, effectively treating the tables as bags due to a different notion of equality. Sets of equal elements are handled by taking their Cartesian product. There is also union, which uses complete equality, and group by, where you have to supply an aggregate function.

- LINQ [MBB06] is an extension for .NET languages that adds SQL-like queries that allow you to interact with a variety of data-structures as though they are sets. LINQ has join, which behaves like the join in SQL, and GroupJoin, which can be used to create joins and left joins. It also has groupBy, which produces a group object.

- Matlab [HH16] has several functions that produce bags. No standard bag/set operations are available. The bag is more an abstraction than an actual bag datatype.

In the above list the languages have been ordered by how relevant they are to this thesis.

These languages can serve as an inspiration for the features implemented in this thesis. Set languages are most useful for making it more intuitive to write certain algorithms, and thereby improving programmer speed and productivity. This improvement is relevant when the algorithm calls for the usage of set based functions like intersection. From the presence or absence of certain functions we can conclude what functions are the most important to set programming.

All of these languages, except Matlab, have an equivalent of intersection. In LINQ and SQL the equivalent is join. Out of these, only LINQ does not have a version of union, although it probably relies on the concatenation operation or equivalent of the specific data structure that is being queried. SETL and CLAIRE have equivalent lists of set operations.

Not mentioned in the above overviews is the fact that all of these languages fall into one of two categories.

- They are not meant to express computations on the elements of the sets/bags, like SQL and LINQ.

- They contain imperative elements.

This allows for functionality that, while not explicitly set or bag based, is still important for working with them. This is especially apparent with Python and JavaScript, where the set objects have a list of methods that have nothing to do with it being a set specifically. An example of this is iterating over a set, creating a new set based on its elements. Accelerate is purely functional and has intentionally restrictive syntax. While something like iteration over an array is possible, there are restriction over what can be done in such an iteration. To be as expressive as other array languages, Accelerate uses functions from the functional paradigm like `map`, `fold`, and `filter` to process arrays. Accelerate faces the same problem with sets and bags. For a set or bag construct to be as expressive as those in the set languages, Accelerate will have to use a similar approach as with the array languages. Functions on sets/bags will have to be created that are not present in other set languages, but instead are sets/bags equivalents of functional programming operations. This is of course in addition to the standard set operations present in most set languages covered.

3 Design

During the proposal of this thesis, several possibilities for how to implement the desired effect were identified. In the following sections the decisions that were made are described.

3.1 Representation

The manner in which the information is stored needed to be determined. There were three options:

1. Extend arrays with additional value(s), such as (a) boolean(s), which represent whether it is ordered.
2. Extend arrays with a new type argument, namely one of a new type class containing the orderedness information.
3. Make a new type for unordered arrays.

Option 2 - adding a new type argument - would be very complicated relative to the other two options and was therefore not chosen. To elaborate, adding a new type argument would require changing the type annotations every time `Array` appears in Accelerate, which is the majority of all functions in Accelerate. Not only would this take a lot of work, this would endanger the backwards compatibility of Accelerate. There are options to maintain it, but those are unnecessarily complicated and would not be worth the effort.

If the goal is to simplify the process of writing algorithms that are best expressed using sets or bags, then explicitly defining them as bags is beneficial. This approach helps to improve the clarity and efficiency of algorithm development. Having distinct bag types appears to be the most intuitive solution to this problem. Since the objective is to enhance intuitiveness, this suggests that separate bag types are indeed the optimal solution.

We could also have chosen to make the new type be a set instead of a bag. On the one hand, making them sets would make it easy to ensure only one of each element is present. On the other hand, using sets would make it impossible to store multiple of any element. Because the latter is a significantly stronger restriction for the user, bags were chosen. And bags were chosen despite the fact that using bags will also make it more complicated to make implementation decisions about certain functions. More about that in Section 4.

3.2 Introduction of bags

In addition to the existence of bags, there also needs to be a way to construct bags to use them. There were two ways this could be accomplished:

1. Take inspiration from the arrays and have a collection of introduction functions for bags.
2. Add a function that turns an array into a bag.

The former would make usage easier because if you want to create a bag from nothing you would only have to call one function instead of two. However, some introduction functions are not well defined for bags. Specifically `generate`, which uses a function that takes an index and produces an element for that index, indexes not being defined for bags.

In addition, many algorithms will need functions to translate between arrays and bags to be implemented. Not implementing such functions would defeat the point of adding bags, so there is no possibility of not doing that work. Therefore, the main way to introduce bags will be through translation functions. Instead, the question becomes: Is it necessary to add introduction functions in addition to the translation functions? It was decided to add introduction functions whenever it was concluded that they needed less effort to add than it would take to work around their absence.

4 Implementation

The user is given the new type `Bag` and a set of functions that operate on `Bag`s.

When writing a program in Accelerate, an AST (abstract syntax tree) is constructed internally. This is what the `Acc` type represents. Because the AST is the program in its entirety, all the information relating to orderedness is stored in the AST. Internally, the `Bag` is actually a vector. The knowledge of unorderedness is maintained because all the new functions implicitly contain the knowledge that their operands are unordered.

4.1 Place in pipeline

The Accelerate pipeline consists of several stages, divided between the front- and back-end. The purpose of this division is to allow Accelerate to target multiple architectures. As seen in Figure 2, Accelerate has multiple back-ends. What back-end is being used, and therefore what architecture is being targeted, is selected by the user when running an Accelerate program. The different back-ends translate the program into appropriate forms for the different architectures and optimize the program differently. Architectures may vary in what operations are available in them; therefore, the back-end needs to be given a program containing only operations that are available. Ensuring this is the purpose of the desugaring step.

The desugaring step translates from the front-end to the chosen back-end. It has more complicated effects than the term usually implies. The effects of this step differ depending on what back-end is being targeted. Desugaring takes the operations of the AST, and translates them into operations that are available on the architecture targeted by the back-end in question. In this way, the front-end does not have to account for the possibility that some operation is not available on some back-end.

Functions in Accelerate are divided into two categories, core functions and auxiliary functions. Core functions have a corresponding AST constructor and a corresponding node if they are used. Auxiliary functions do not have corresponding AST constructors. They are implemented in terms of the core functions.

Among the functions newly introduced in this thesis, there are both core and auxiliary functions. Like most core functions, the core functions introduced in this thesis exist at least up until the desugaring step in the pipeline, where the back-end is given the opportunity to implement its own version of that function. This is to allow for the bags to be used to implement more efficient versions of the functions. Implementing these more efficient versions is beyond the scope of this thesis. Therefore, the desugaring step is the deepest part of Accelerate processing where changes have been made.

This means that core functions need to be present through the entire front-end. The front-end part of the pipeline consists of the steps sharing-recovery and array-split-lets. The former plays an important role for performance, and the second removes some syntactic sugar.

To add a core function to Accelerate in this way, there are several type classes, data types, and functions that need new instances, constructors, and cases, respectively. The user-facing `Acc` type is implemented in `Smart.hs`, and is internally called `SmartAcc`. It represents the AST for the user, but it gets immediately transformed during the sharing-recovery step into a data type defined in `AST.hs` called `PreOpenAcc`. The array-split-lets is performed on that type, but it is transformed again in desugaring, into `OperationAcc`, which is defined in the back-end. This means that the new type of the AST is variable, making translation more complicated than usual. In addition, the AST types at this point use an explicit environment, meaning that variables are no longer build-in Haskell terms, but new types that index into the environment. Due to this, transforming the AST means rebuilding the environment from scratch. To do both these things, the desugaring function is broken into two parts: the main desugaring function, and the default implementations of the new AST constructors, also called the "make" or "mk" functions. The main desugaring function breaks down the old tree and environment, and build up the new environment. The correct mk function is then called using indexing into the new environment. The main desugaring function is independent of the back-end. The back-end does not need to implement a constructor for every constructor in the front-end's data type. Instead, it is only required to implement a few constructors, the others have default implementations in terms of the existing ones. Of course, the back-end can implement any of the other constructors too, which allows the back-end to substitute a more performant implementation.

4.2 Functions

In Section 2 the most common functions in other languages that use sets or bags are listed. In addition it was explained that additional functions would be needed because Accelerate is a functional language. Based on that, the following core functions were implemented:

- `bagMap`
- `bagFold`
- `cartesian`
- `cartesianWith`
- `bagFilter`
- `bagIntersect`
- `bagUnion`
- `bagSubtract`

In addition, the following auxiliary functions were also implemented:

- `arrayToBag`
- `bagToArray`
- `bagFill`
- `bagSize`
- `bagNull`
- `emptyBag`
- `bagAll`
- `bagAny`
- `bagAnd`
- `bagOr`

In the following, all these functions will be discussed in detail, starting with the auxiliary functions.

4.2.1 Translation functions

The purposes `arrayToBag` and `bagToArray` are self-explanatory: they translate between arrays and bags. Their types are:

```
arrayToBag :: Acc (Array DIM1 a) -> Acc (Bag a)

bagToArray :: Acc (Bag a) -> Acc (Array DIM1 a)
```

When turning a bag into an array, ordering needs to be reintroduced. There are two ways this can be done:

1. All array elements (all members of the class `Elt`) have an ordering defined on them. Using this, any bag could be translated into array with its elements sorted.
2. The result could be allowed to be non-deterministic, meaning that the order depends on implementation details of the parallel functions, which we do not give any guarantees about.

The second option has been chosen.

Due to the implementation of bags, they do not do anything under the hood.

4.2.2 Size related

The types of `bagFill`, `bagSize`, and `bagNull` are:

```
bagFill :: Elt e => Exp Int -> Exp e -> Acc (Bag e)

bagNull :: Elt e => Acc (Bag e) -> Exp Bool

bagSize :: Elt e => Acc (Bag e) -> Exp Int

emptyBag :: Elt e => Acc (Bag e)
```

`bagFill` creates a bag of specified size and fills it with a given element. `bagSize` Gets the size of a bag. `bagNull` checks if a bag is empty. All are bag versions of preexisting Accelerate functions. `bagFill` and `bagSize` are made using their preexisting versions. `bagNull` is made using `bagSize`.

The only new function is `emptyBag`. It creates an empty bag.

4.2.3 Special folds

The types of `bagAll`, `bagAny`, `bagAnd`, and `bagOr` are:

```
bagAll :: (Elt e)
       => (Exp e -> Exp Bool)
       -> Acc (Bag e)
       -> Acc (Array Z Bool)

bagAny :: (Elt e)
       => (Exp e -> Exp Bool)
       -> Acc (Bag e)
       -> Acc (Array Z Bool)

bagAnd :: Acc (Bag Bool)
       -> Acc (Scalar Bool)

bagOr :: Acc (Bag Bool)
      -> Acc (Scalar Bool)
```

These are bag versions of their preexisting Accelerate functions `all`, `any`, `and`, and `or`. They are all implemented using `bagFold`.

4.2.4 Mapping bags

The type of `bagMap` is:

```
bagMap :: (Elt a, Elt b)
       => (Exp a -> Exp b)
       -> Acc (Bag a)
       -> Acc (Bag b)
```

Internally, `bagMap` is Accelerate's default `map`. This is because being a bag does not give any opportunity for faster implementations of map.

4.2.5 Folding bags

The type of `bagFold` is:

```
bagFold :: (Elt a, Elt b)
        => (Exp a -> Exp b)
        -> Acc (Bag a)
        -> Acc (Bag b)
```

Like `fold` on arrays, we assume that the function is commutative. In addition, `bagFold` assumes that the function is associative. The default implementation (a.k.a. the mk function) of `bagFold` is to refer to the default implementation of `fold`, which makes no use of the fact that the order of the elements is irrelevant.

The main desugaring code for `fold` in arrays was studied, and the desugaring code for `bagFold` is heavily based on it. Later, the code was improved by relying on the fact that all input arrays will be vectors due to all inputs being bags.

4.2.6 Cartesian product

The Cartesian product is an operation on sets that produces a set of pairs consisting of every pair of an element of the first set as the first element and an element of the second set as the second element. This definition is naturally extended to bags.

The Cartesian product on sets or bags is the closest equivalent to the `zip` function on arrays. Similarly to how `zip` has `zipWith`, which applies an arbitrary function to the elements instead of pairing them, the `cartesian` function has `cartesianWith`, which applies a function to each pair of elements.

The types of `cartesian` and `cartesianWith` are:

```
cartesian :: (Elt a, Elt b)
          => Acc (Bag a)
          -> Acc (Bag b)
          -> Acc (Bag (a, b))

cartesianWith :: (Elt a, Elt b, Elt c)
              => (Exp a -> Exp b -> Exp c)
              -> Acc (Bag a)
              -> Acc (Bag b)
              -> Acc (Bag c)
```

Also like `zip` is defined in terms of `zipWith`, `cartesian` is defined in terms of `cartesianWith`.

4.2.7 Filter bags

The type of `bagFilter` is:

```
bagFilter :: (Elt e)
          => (Exp e -> Exp Bool)
          -> Acc (Bag e)
          -> Acc (Bag e)
```

The standard `filter` function is implemented in terms of `compact`, which takes an array of booleans instead of a predicate as its first input. Likewise, the standard implementation of `bagFilter` is defined using the new helper function `mkCompact`, which is also used in `bagSubtract`.

`bagFilter` is the first function that needed to be implemented - and, indeed, the first function implemented at that level in the code - where the size of the output depends on the contents of the input. The AST of Accelerate by default expects that the sizes of the output of its functions are specified upfront, using only the sizes of the input. To make this possible, we changed the type of the internal mk function that constructs the filter from `OperationAcc op env ()` to `OperationAcc op env (DesugaredArrays (Array DIM1 t))`. In effect this means that the resulting array will be given as a direct result instead of putting it in the environment. This is because making space for it on the environment was what required us knowing the size beforehand.

4.2.8 Intersect

The type of `bagIntersect` is:

```
bagIntersect :: (Elt a)
             => Acc (Bag a)
             -> Acc (Bag a)
             -> Acc (Bag a)
```

All versions of intersect on bags need a formula to determine the amount of any element produced based on the amounts of that element in the input bags. In mathematics, the standard formula is the minimum of the two values. Our default implementation of `bagIntersect` instead has multiplication. This means that if bag A contains x of some element, and bag B contains y of that element, then the intersection has $x * y$ copies of that element. This is for ease of implementation. Specifically, `bagIntersect` is implemented using `cartestianWith` and `bagFilter`. Due to the former the default implementation of `bagIntersect` is quadratic in complexity.

Because it is implemented using `bagFilter`, the size of the result of `bagIntersect` also depends on the contents of its input and therefore it also has the type of its internal mk function changed.

Any back-end has the ability to overwrite this implementation.

4.2.9 Union

The type of `bagUnion` is:

```
bagUnion :: (Elt a)
         => Acc (Bag a)
         -> Acc (Bag a)
         -> Acc (Bag a)
```

Like intersect, union needs a function to determine the amount of an element to produce. Also like intersect, the default implementation does not use the standard function (which is maximum). Instead, addition is used, again for implementation reasons. Specifically, `bagUnion` is implemented as concatenation of arrays.

Like `bagIntersect`, this implementation can be overwritten by the back end.

4.2.10 Subtract

The type of `bagSubtract` is:

```
bagSubtract :: (Elt a)
            => Acc (Bag a)
            -> Acc (Bag a)
            -> Acc (Bag a)
```

Like intersect and union, subtracting bags from each other requires a function to determine the amount of an element to produce. The standard is, as the name suggest, subtraction. In our default implementation, the amount of elements in the output is not determined by a basic mathematical function, instead it is determined by a conditional. When calculating the result of $A - B$, if A contains some amount of some element a then the output will contain that many of a if and only if B does not contain a. Otherwise the output will not contain a single a. Our default implementation compares every element in A with every element in B, and is therefore quadratic in complexity. This function uses `mkCompact`, which exists because otherwise 90% of `bagFilter` would be copied into `bagSubtract`.

Like the previous functions, this implementation can be overwritten.

4.3 Limitations

When comparing the functions for arrays and those for bags, the most glaring omission is an equivalent for indexing. This is because there are no indices in bags. For simplicity, and to make it more obvious that a non-deterministic result is produced, we do not add a bag-equivalent for indexing.

Using non-standard formulas for the output of `bagIntersect`, `bagUnion`, and `bagSubtract` causes the standard set algebraic properties to no longer hold. This was done for ease of implementation. For example, `bagUnion` does not distribute over `bagIntersect`. Normally, with

$$A = \{1, 2, 2\}$$
$$B = \{1, 2, 3\}$$
$$C = \{1, 3\}$$

it would be

$$C \cup (A \cap B) = C \cup \{1, 2\} = \{1, 2, 3\}$$
$$(C \cup A) \cap (C \cup B) = \{1, 2, 2, 3\} \cap \{1, 2, 3\} = \{1, 2, 3\}$$

However, in our formulation, it is

$$C \cup (A \cap B) = C \cup \{1, 2, 2\} = \{1, 1, 2, 2, 3\}$$
$$(C \cup A) \cap (C \cup B) = \{1, 1, 2, 2, 3\} \cap \{1, 1, 2, 3, 3\} = \{1, 1, 1, 1, 2, 2, 3, 3\}$$

5 Experimental setup

This research is divided into two areas. The research into the front-end of Accelerate, where the usability of Accelerate is intended to be improved by adding bags; and the research into the back-end of Accelerate, where performance improvements are aimed at being achieved by not caring about the order of the elements of an array. The experiments were ran on a machine with an AMD 2950X processor which has 16 cores and 32 threads, with 32GB of RAM on Ubuntu 22.04.

To test the results of this research, two different experiments have been performed.

5.1 Front-end

To test the increase in usability, several algorithms have been implemented. Each algorithm has been implemented twice, once without the features added in this thesis, and once with those features. The experiences of implementing those algorithms are compared for each pair of implementations. These algorithms are examples that are naturally expressed in terms of sets or bags, because it is for those kinds of algorithms that these features are naturally going to be used.

Two different algorithms have been implemented, each in several variations. The algorithms are:

1. Breath-first search on directed graphs. [CLRS22]
2. Kahn's algorithm for topological sorting of directed graphs [Kah62].

5.1.1 Breath first search

Breath first search (BFS) is an algorithm that, for the purposes of this thesis, takes a directed graph and a starting vertex in that graph and traverses the graph from closest to furthest from the starting vertex. What the output of the algorithm is varies between uses. In this thesis, two possible outputs are used.

The following is pseudocode for breath first search on directed graphs. This version returns the distance from every vertex to the start vertex,

```
function BFS (graph, start):
    for vertex in graph.vertices:
        distance[vertex] = -1;
    distance[start] = 0;
    level = 1;
    frontier = {start};
    nextFrontier = {};
    while frontier is not empty:
        for vertex in frontier:
            for neighbour of vertex:
                if distance[neighbour] == -1:
                    nextFrontier = union nextFrontier neighbour;
                    distance[neighbour] = level;
        frontier = nextFrontier;
        level += 1;
        nextFrontier = {};
    return distance;
```

Variations For the experiment four versions of BFS are used, not counting the fact that every version will have a bag based counterpart. The four versions are differentiated in two aspects: the input format of their graph, and the output format: whether we want to know the vertices' distance from the start, or only whether they are reachable.

The two formats of the input graphs are the following standard Haskell types.

```
type Graph1 = (Int, [(Int, Int)])

type Graph2 = ([Int], [Int])
```

The former is an edge list. The latter is similar to an adjacency list, but here its first element is a list of indices for each vertex into the second list, which represents the edges departing from that vertex. These formats have been made from the perspective that an outside system wishes to use Accelerate to process its graphs efficiently. Therefore, these representations are standard Haskell list and pairs, instead of Accelerate vectors and pairs.

There are two variations of the algorithm's output. The first is an array with one integer per vertex, the integer represents the distance of that particular vertex to the start vertex, or is `-1` if the vertex is unreachable. The second is a boolean per vertex, representing whether that vertex is reachable or not. This boolean can be given directly, in an array, or indirectly, by returning a bag containing all reachable vertices.

5.1.2 Kahn

A topological sorting of a directed graph is an ordering of its vertices such that for any vertex, its vertex of origin will come before its vertex of destination. Some directed graphs may have more than one possible topological sorting. Others may not have any.

Kahn's algorithm [Kah62] is an algorithm for finding a topological sorting of a directed graph. This means it will not produce all possible topological sorting, or even their number, only (at most) one. The implementations of Kahn's algorithm in this thesis will return an empty array if there is no topological sorting.

```
function kahn graph:
    list = [];
    set = all vertices with no incoming edge;
    newSet = {};
    while set is not empty:
        list.append(set);
        for vertex1 in set:
            for vertex2 where there exists edge (vertex1, vertex2):
                graph.remove(edge);
                if graph has no edge to vertex2:
                    newSet = union nextFrontier vertex2;
        frontier = nextFrontier;
        nextFrontier = {};
    if graph has edge:
        return {};
    else
        return list;
```

Variations Similarly to BFS, multiple versions of Kahn's algorithm have been implemented. In Kahn's case, there are two versions. One for each of the two graph formats.

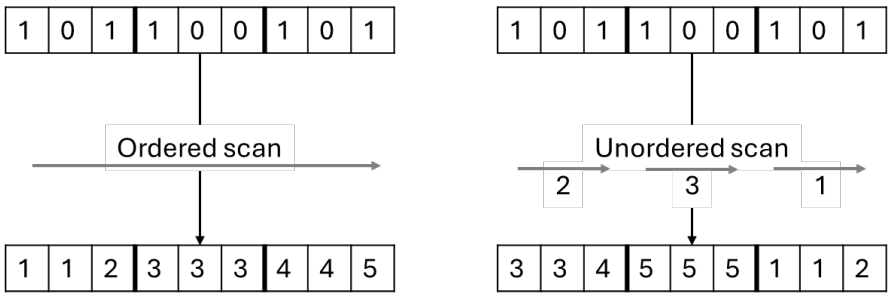

Figure 4: Difference between ordered and unordered scans

5.2 Back-end

To test the improvement of performance, the function `filter` has been implemented in both an ordered and an unordered manner. It was used as an example in Section 1.5 to show the potential performance improvements of unordered arrays. These functions have been benchmarked using the Haskell benchmarking library Criterion. Specifically, each function is given 28 different inputs, four sizes with seven success rates each.

5.2.1 Functions

The filter function takes an array and a predicate and filters out all elements that do not fulfill the predicate. The effective difference between the two filters is whether they guarantee that the elements that make it through are in the same orders as they went in. The default filter, which is the ordered filter, provides this guarantee. The unordered filter does not.

Both filters have the same basic structure.

1. Map the predicate over the input array to get an array of booleans.
2. Scan over the booleans to determine the location of all elements that made it through, and the size of the output array.
3. Permute the elements into the output array.

The difference in the filters lies in the scan. Both an ordered and an unordered version of scan have been implemented. They are what is really being tested; the filter functions are used as examples because scan is realistically only used in the context of functions like filter. The unordered scan does not change the order of the output elements as they belong to the input elements. However, unlike a normal (additive scan), the output is not purely incrementing. Figure 4 shows the difference between the two scans. Both scans are parallel and therefore split up the input array, if it is large enough, into blocks of predetermined size. These blocks are then distributed among threads as they become available.

Ordered scan Accelerate has recently been updated with a new scan algorithm [dWvBKM24]. This scan is more performant than the previous implementation due to the reduced overhead of not having to iterate over the whole array as many times. In this algorithm, threads claim blocks from left to right, and each block performs the following steps.

1. It folds over itself to determine its aggregate.
2. Any block, but the first one, waits for the block before it to determine its prefix (a.k.a. finish steps 1, 2, and 3).
3. It sums the prefix of the previous block with its own aggregate to determine its prefix.
4. It takes the previous block's prefix and uses it as the first element of a serial scan over the block.

Unlike the current implementation, the previous implementation was executed globally in three phases, which are as follows.

1. Each block folds over itself to determine its aggregate.
2. One thread performs a serial scan over the aggregates in order.
3. Each block takes its element from step 2 and uses it as the first element of a serial scan over the block.

This implementation requires three array accesses (per element), compared to the new implementation's two. This, in addition to the increased degree of parallelism, means that the new implementation is significantly faster.

Unordered scan The unordered scan in Accalerate is based on the atomic operation-based approach from a paper by Darius Bakunas-Milanowski et al. [BMRSC17]. In this algorithm, threads claim blocks and each block performs the following steps.

1. Each block folds over itself to determine its aggregate.
2. Each block reads a global counter and adds its aggregate as an atomic operation.
3. Each block takes what it read in Step 2 and uses it as the first element of a serial scan over the block.

There are various similarities between the unordered and current ordered implementations. Both implementations have each block take the steps independently. Both implementations start by folding over the block and end by scanning over the block. Both versions perform two array accesses per element. The difference is that instead of waiting for the previous block to have its prefix ready, it gets the prefix of the most recent block that finished its prefix (only potentially waiting for it to finish its atomic operation). Both implementations make their own prefixes available as soon as possible, during the same atomic operation in the unordered case. The end result is that both are the same except that the unordered implementation treats the blocks as though they were in some other order, without actually changing their order. The unordered variant treats the blocks as though they were in the order that they finished step one in.

5.2.2 Input

The four basic inputs are four arrays of integers with a size of one thousand, one million, ten million, and one hundred million, respectively. The integers are random 64 bit integers. Each of these arrays will be filtered using seven predicates, each differing in what fraction of the input they let through. The seven fractions are as follows:

- 1 in 1, that is, all of it.
- 255 in 256.
- 7 in 8.
- 1 in 2.
- 1 in 8.
- 1 in 256.
- 0 in 1, that is, none of it.

Each of these 28 different benchmarks will be performed on both the ordered and unordered filter functions. Per standard Criterion procedure, the OLS [2] estimate will be given. The results of the two functions will be compared to determine if the removal of ordering has increased performance.

[2] ordinary least-squares regression

6 Results

6.1 Front-end

To investigate the usability of the feature added in this thesis, we have implemented six pairs of algorithms, for a total of twelve implementations. Their code can be found in the Appendix A. In that code, many terms of the `Acc a` type have a type annotation. The code in the appendix sometimes contains functions starting with "`P.`". These are functions where both Accelerate and base Haskell contain functions with the same name. For those functions, the Haskell versions will have "`P.`", while the Accelerate versions will not be preceded by anything.

All implementations have a similar structure. Firstly, there is a main function that creates some initial values and contains a while loop. The body of the loop is a separate function. In addition to the main and body functions, several of the vector implementations have identical functions called `combine`.

6.1.1 Combine

The need for a `combine` function is an excellent example of the need for sets or bags in these algorithms. At various points in the algorithms, operations such as union, intersect, subtract, or the cartesian product are called for. In the vector implementations, these operations need to be seperately implemented, and to do so, every element of one array needs to be compared to every element of another array. Depending on the operation, the end result needs to be filtered, folded, or flattened. In any case, the initial step of pairing every element of an array with every element of another is shared. This is what is implemented in `combine`. It takes two vectors as input and produces a matrix with side lengths equal to the vectors' sizes. This matrix is then used in the individual implementation to complete the desired operations.

The `combine` function is not a built-in function of Accelerate. Due to this, its code has been added to every implementation that uses it, because you would need to implement it if you wanted to make only one of these implementations. The following sections will not take `combine` into consideration. In other words, the code will be analyzed as though `combine` were a built-in Accelerate function. Therefore, be aware that BFS implementations 1 and 3 (A.1, A.5), and Kahn implementations 1 and 2 (A.9, A.11) have greater improvements than described below. Specifically, seven lines of code of the function have been saved in those bag implementations.

6.1.2 BFS

There are four pairs of BFS implementations, each pair consisting of a vector implementation and a bag implementation. Each pair implements a different version of the BFS algorithm, BFS 1,2 ,3, and 4. The first version takes the edge-list graph representation as input, and produces the integer output. The second version takes the adjacency-list representation as input, and produces the integer output. The third version takes the edge-list graph representation as input, and produces the boolean output, and the fourth version takes the adjacency-list representation as input, and produces the boolean output.

BFS 1 Vector implementation 1 (A.1) and bag implementation 1 of BFS (A.2) have the same amount of code. It was only practical to rewrite half of the algorithm in terms of bags. The half that was not rewritten consists of two parts. Firstly, the vector that stores whether any vertex has already been visited by the algorithm uses its length to index the correct vertex. To make that functionality using bags would require implementing dictionary/map-like functionality, which would greatly overcomplicate the implementation. Secondly, filtering out duplicate vertices from the next frontier is based on saving only one of the indices at which the vertex is stored. Doing the same using bags would again require dictionary/map-like functionality at least.

BFS 2 Like the first pair of implementations, both implementations of the second pair (A.3, A.4) are of the same size. Unlike with the first pair, practically none of this pair's "bag" implementation is based on bags. The half of the algorithm that was still implemented using vectors in the first bag implementation is still present. The difference between the first and second implementations is the input format of the graph. The first format contained a straightforward list of edges, which allowed the use of a cartesian product and a filter to determine the next frontier. However, the indexing of the second graph format is built for functions like `expand`. This makes the vector implementation very intuitive. However, there is no bag equivalent of expand, either introduced in this thesis or in other languages with set or bag constructs covered in the related work section.

There is an alternative way that bag BFS 2 could have been implemented. Instead of using the adjacency-list format as it is given, it could be transformed into the edge -list format in the main function and have the body function as an effective copy of body function of the first bag implementation. The code to transform the adjacency-list format into the edge-list format already exists; it is part of the second implementations of Kahn's algorithm. This would make the bag implementation 2 lines larger, however.

BFS 3 Compared to the first bag implementation, a larger part of the third bag implementation (A.6) is implemented in terms of bags. This is because this version of the algorithm uses a bag of visited vertices instead of a vector. This, of course, means that calculating the new set of visited vertices can be done using bags.

Bag implementation 3 is about one line of code longer than vector implementation 3 (A.5). This is the case because previously duplicated and already visited vertices were being filtered out of the new frontier at the same time. Now that what vertices have already been visited is stored in a bag, that information can no longer be accessed using indexing. Instead, it is filtered out using the subtract operation. This means that it must be done separately, hence the increased complexity. Doing this filtering separately also means that a greater part of the algorithm is implemented in terms of bags.

BFS 4 Bag implementation 4 (A.8) is similar to bag implementation 2 because it is still predominantly written without using bags, still for the same reasons. There are exceptions to this, namely the additional line of bag calculations gained in bag implementation 3. The set of visited vertices is now a bag, so it is calculated using bags. This also means that filtering already visited vertices out of the next frontier is done using subtract.

6.1.3 Kahn

Kahn 1 The first bag implementation of Kahn's algorithm (A.10) has less code than the first vector implementation (A.9). Unlike any of the BFS implementations, the bag implementations of Kahn's algorithm are drastically simpler in both the main and body functions, whereas the BFS implementations had practically all their improvements in the body function. Kahn's algorithm requires finding the set of vertices without incoming edges at the start. This is easier to do using bags, which is the reason that the main bag function is about two lines shorter. In the body function, the source of the improvements is the greater ease with which elements can be filtered out of a bag based on the contents of another bag than can be done with vectors. With bags this can either be done using built-in functions such as subtract, or this can be done with the functions `cartesian`, `bagFilter`, and `bagMap snd`. To do the same with vectors, the before mentioned `combine` function is needed, followed by mapping a predicate, folding over the resulting matrix, and compacting the result with the original vector. This results in one or two less lines of code needed per comparison for the bag implementation, depending on whether a subtract could be used.

Fraction of elements kept	1,000	1,000,000	10,000,000	100,000,000
All	114.1 μs	1.261 ms	15.69 ms	357.6 ms
255 in 256	119.3 μs	1.247 ms	15.10 ms	319.9 ms
7 in 8	111.5 μs	1.132 ms	13.96 ms	296.9 ms
1 in 2	113.5 μs	747.3 μs	8.933 ms	157.8 ms
1 in 8	115.3 μs	455.2 μs	4.651 ms	71.99 ms
1 in 256	114.9 μs	332.4 μs	2.714 ms	30.14 ms
None	115.8 μs	344.0 μs	1.977 ms	18.05 ms

Table 1: Criterion OLS estimates of ordered filter

Fraction of elements kept	1,000	1,000,000	10,000,000	100,000,000
All	124.0 μs	1.205 ms	15.77 ms	356.7 ms
255 in 256	117.6 μs	1.163 ms	14.92 ms	359.5 ms
7 in 8	124.1 μs	1.035 ms	13.13 ms	296.0 ms
1 in 2	131.4 μs	685.2 μs	8.911 ms	154.7 ms
1 in 8	122.3 μs	375.1 μs	4.406 ms	71.48 ms
1 in 256	121.6 μs	242.9 μs	2.559 ms	25.97 ms
None	111.7 μs	234.9 μs	2.099 ms	19.64 ms

Table 2: Criterion OLS estimates of unordered filter

Kahn 2 The differences between the second bag and the second vector implementations (A.12, A.11) are the same as those between the first bag and vector implementations. The differences from the first to the second implementations are the same for both vectors and bags. They stem from the change in graph format. Unlike BFS, Kahn's algorithm has specific requirements for the format of the graph. Specifically, Kahn's algorithm needs to modify the list of edges, and attempting to do that with the adjacency-list representation is impractical. Because of this, at the beginning of each implementation, the adjacency-list representation needs to be translated into an edge list. This is done in both the vector and bag versions, which causes the additional code. The code that does this uses vectors because the original format is dependent on order, so the code is identical in both versions.

6.2 Back-end

To investigate the performance improvements made possible by removing the requirement of orderedness, ordered and unordered filter functions have been benchmarked. The results are contained in Tables 1 and 2. From these tables, it is clear that time taken increases with the size of the input (as would be expected) and with the fraction of elements kept. Table 3 contains the difference between the two tables. There is a pattern of lower times in all but the smallest array size, with increasing improvements for lower fractions of elements kept. However, there are exceptions to that pattern, where the time taken actually increases. These increases seem to also scale with the fraction of elements filtered out. All of these differences, positive or negative, are at most a tenth and sometimes a hundredth of the original times. Overall, we conclude that there is no significant difference between the two functions in terms of performance.

Fraction of elements kept	1,000	1,000,000	10,000,000	100,000,000
All	+9.9 μs	-56 μs	+8 μs	-9 μs
255 in 256	-1,7 μs	-84 μs	-18 μs	+39.6 μs
7 in 8	+12.6 μs	-97μs	-83 μs	-9 μs
1 in 2	+17.9 μs	-97μs	-22 μs	-3.1 μs
1 in 8	+7.0 μs	-80.1 μs	-245 μs	-51 μs
1 in 256	+6.7 μs	-89.5 μs	-245 μs	-417 μs
None	-4.1 μs	-109.1 μs	+122 μs	+159 μs

Table 3: Differences between ordered and unordered filter

Version	Difference in number of lines	Judgement
BFS: edge-list input and interger output	7 shorter	Easier to write
BFS: adjacency-list input and interger output	Same length	Same difficulty
BFS: edge-list input and boolean output	6 shorter	Easier to write
BFS: adjacency-list input and boolean output	1 longer	Same difficulty
Kahn: edge-list input	12 shorter	Easier to write
Kahn: adjacency-list input	12 shorter	Easier to write

Table 4: Summery of results

7 Conclusion

In this thesis, the `Bag` type was added to Accelerate. Along with it, a collection of functions was added to work with `Bag`s. These new functions were added through the processing pipeline from the user-facing code, through sharing-recovery and array-split-lets, into desugaring. Then several versions of two algorithms were implemented, both with and without the new features. The two versions were then compared. In addition, two versions of `filter` were compared.

In transitioning from vector implementations to bag implementations, several improvements to conciseness were observed. BFS 1 was enhanced by seven lines of code due to the removal of the `combine` function. However BFS 2, did not show any improvement. BFS 3 saw a six-line improvement, attributed to the removal of `combine` and the addition of separate filtering for already visited vertices. In contrast, BFS 4 experienced a slight regression, increasing by one line of code.

Kahn 1 benefited significantly, with a 12-line improvement resulting from the removal of `combine`, easier identification of starting vertices, and more straightforward dependent filtering. Similarly, Kahn 2 improved by 12 lines for the same reasons as Kahn 1.

Overall, out of the six implementations, four demonstrated noticeable improvements. The contributions of this thesis have successfully shortened the work of coding these algorithms, which were chosen for their natural expression in terms of bags. This improvement not only simplifies the implementation process but also potentially increases the efficiency and readability of the code. By reducing the complexity associated with vector operations and taking advantage of the inherent advantages of bag structures, these changes pave the way for more intuitive and maintainable algorithm designs.

The two implementations that did not improve were characterized by the fact that almost no new features were used. This may indicate a shortcoming in the set of bag functions that were added. The added functions were inspired by two sources. Firstly, a variety of languages that already feature bag functions. These functions are bag- or set-specific, such as union, intersect, and subtract. However, it was concluded that these would not be sufficient for Accelerate. So, secondly, inspiration was taken from pre-existing Accelerate functions where applicable. Such functions are map, fold, and filter. However, none of these functions allowed for the functionality that was required for BFS bag implementations 2 and 4. This could have been prevented by having some bag equivalent for `expand` and some way to filter out duplicates.

It is worth noting that to code the vector versions, the bag functions effectively had to be implemented like `subtract`, mimicking the work that was done to actually implement those functions. This redundancy highlights the inherent value of using bag structures directly, as it avoids the need for such workaround implementations. Without these improvements, every user would have to replicate this effort to achieve the same results, leading to inefficiencies and inconsistencies.

This also supports the decision to implement the unorderedness as bags, as the entirely new set of functions was what allowed for this improvement. If the choice was made to extend the existing arrays, then the old set of functions would still have to be used, and the difficulty would have persisted.

On a related note, the choice to implement bags specifically instead of sets has created two issues. Firstly, several bag functions have been implemented in an untypical manner. The functions union, intersect, and subtract all produce different amounts of copies of an element than is standard. These implementations were chosen because they were the easiest to create. Specifically, the functions were implemented that way due to the time constraints of this thesis. If more work is done, the functions can be implemented to behave as is standard. Secondly, it is difficult to remove duplicates from the frontiers in these implementations. It should be expected that if other algorithms were chosen for the

experiments, the ability to contain duplicates might have been very important, and its absence might have complicated the implementations greatly. Therefore, the benefits of bags still outweigh those of sets.

There was no significant difference between the ordered and unordered versions of filter. Thus, the most obvious source of performance improvement is no longer available. This leaves an open question whether performance improvement is practically possible in general using the features introduced in this thesis. It could be that the premise of not maintaining the order of an array is intrinsically impossible. However, the lack of improvement could also be due to limitations of the experiment. Specifically, the experiment was only performed on a CPU and not on GPU. The used CPU might not have been parallel enough to make use of the extra opportunity for parallelism afforded by unorderedness. Alternatively, it could also be that performance improvements through unorderedness is viable and that the lack of improvements is caused by the especially performant version of scan used in Accelerate.

Even if this research does not lead to any performance improvements anywhere, the lack of improvements for filter is also a credit to the implementation already in Accelerate. Any implementation with a noticeable degree of inefficiency would likely have been improved upon by an unordered variant. In this way, it was a pleasant surprise that the experiment showed no difference.

Was the effort worthwhile? For a specific class of graph algorithms, the answer is affirmative. The improvements in code simplicity, maintainability, and readability justify the transition to bag implementations. Moreover, the reduction in redundant code and the alignment with the natural expression of these algorithms in terms of bags underscore the practical benefits. This suggests that other array languages might also benefit from incorporating bags, potentially leading to broader advancements in algorithm design and implementation.

In the future Accelerate could be improved by adding the expansion and duplicate filtering functions that were found to be lacking. Accelerate could be improved by changing the unconventional implementation of intersect, union, and bag. Furthermore, the question whether performance improvements can be achieved through this research remains unanswered. To test this possibility, future research could add new features to the back-ends of Accelerate that take advantage of the absence of orderedness requirement.

A Algorithms

A.1 Vector BFS 1

```
bfs1 :: Graph1 -> Exp Int -> Acc (Vector Int)
bfs1 gr s = let
    n = constant (P.fst gr)
    vs = fill ( I0 ::. n ) (constant (-1))
    fr = fill (lift $ Z :. (1 :: Int)) s
    T3 _ out _ = awhile
        (\(T3 f _ _) -> unit (not (null f)))
        (body1 gr)
        (T3 fr vs (unit (constant 0)))
    in out

body1 :: Graph1
      -> Acc (Vector Int, Vector Int, Scalar Int)
      -> Acc (Vector Int, Vector Int, Scalar Int)
body1 gr (T3 fr vs lvl) = T3
    nextFrontier
    nvs
    (map (+1) lvl)
    where
        nvs = scatter fr vs (fill (shape fr) (the lvl))
        es :: Acc (Vector (Int, Int))
        es = use (fromList (Z  :. P.length (P.snd gr)) (P.snd gr))
        rs :: Acc (Matrix (Int, (Int, Int)))
        rs = combine fr es
        nextfr :: Acc (Vector Int)
        nextfr = map (snd . snd)
                     (afst (filter (\t -> fst t == fst (snd t)) rs))
        nonDuplIndices :: Acc (Vector Int)
        nonDuplIndices = scatter nextfr
            (fill (constant (Z :. P.fst gr)) (-1)) (enumFromN (shape nextfr) 0)
        nextFrontier :: Acc (Vector Int)
        nextFrontier = map snd $ afst $ filter (\(T2 i x) ->
            (nvs !! x) < constant 0 &&
            (nonDuplIndices !! x) == shapeSize i)
            (indexed nextfr)

combine :: (Elt a, Elt b) => Acc (Vector a) -> Acc (Vector b) -> Acc (Matrix (a, b))
combine a b = let
    na = length a
    nb = length b
    ra = replicate (Z_ ::. constant All ::. nb) a
    rb = replicate (Z_ ::. na ::. constant All) b
    rs = zip ra rb
    in rs
```

A.2 Bag BFS 1

```
bagBfs1 :: Graph1 -> Exp Int -> Acc (Vector Int)
bagBfs1 gr s = let
    n  = constant (P.fst gr)
    vs = fill (I0 ::. n) (constant (-1))
    fr = bagFill (lift (1 :: Int)) s
    T3 _ out _ = awhile
        (\(T3 f _ _) -> unit (not (bagNull f)))
        (bagBody1 gr)
        (T3 fr vs (unit (constant 0)))
    in out

bagBody1 :: Graph1
         -> Acc (Bag Int, Vector Int, Scalar Int)
         -> Acc (Bag Int, Vector Int, Scalar Int)
bagBody1 gr (T3 fr vs lvl) = T3
    nextFrontier
    nvs
    (map (+1) lvl)
    where
        nvs = scatter (bagToArray fr) vs (fill (I1 $ bagSize fr) (the lvl))
        es :: Acc (Bag (Int, Int))
        es = arrayToBag $ use (fromList (Z  :. P.length (P.snd gr)) (P.snd gr))
        ps :: Acc (Bag (Int, (Int, Int)))
        ps = cartesian fr es
        nextf :: Acc (Vector Int)
        nextf = bagToArray $ bagMap (snd . snd)
            (bagFilter (\t -> fst t == fst (snd t)) ps)
        nonDuplIndices :: Acc (Vector Int)
        nonDuplIndices = scatter nextf
            (fill (constant (Z  :. P.fst gr)) (-1)) (enumFromN (shape nextf) 0)
        nextFrontier :: Acc (Bag Int)
        nextFrontier = arrayToBag $ map snd $ afst $ filter (\(T2 i x) ->
            (nvs !! x) < constant 0 &&
            (nonDuplIndices !! x) == shapeSize i)
            (indexed nextf)
```

A.3 Vector BFS 2

```
bfs2 :: Graph2 -> Exp Int -> Acc (Vector Int)
bfs2 gr s = let
    n = constant (P.length (P.fst gr))
    vs = fill ( I0 ::. n ) (constant (-1))
    fr = fill (lift $ Z :. (1 :: Int)) s
    T3 _ out _ = awhile
        (\(T3 f _ _) -> unit (not (null f)))
        (body2 gr)
        (T3 fr vs (unit (constant 0)))
    in out

body2 :: Graph2
      -> Acc (Vector Int, Vector Int, Scalar Int)
      -> Acc (Vector Int, Vector Int, Scalar Int)
body2 gr (T3 fr vs lvl) = T3
    nextFrontier
    nvs
    (map (+1) lvl)
    where
        nvs = scatter fr vs (fill (shape fr) (the lvl))
        avs :: Acc (Vector Int)
        avs = use (fromList (Z :. P.length (P.fst gr)) (P.fst gr))
        es :: Acc (Vector Int)
        es = use (fromList (Z :. P.length (P.snd gr)) (P.snd gr))
        nextf :: Acc (Vector Int)
        nextf = expand (\x -> (avs !! min (x+1) (length avs - 1))
        nonDuplIndices :: Acc (Vector Int)
        nonDuplIndices = scatter nextf
            (fill (constant (Z  :. P.length (P.fst gr))) (-1)) (enumFromN (shape nextf) 0)
        nextFrontier :: Acc (Vector Int)
        nextFrontier = map snd (afst (filter (\(T2 i x) ->
            (nvs !! x) < constant 0 &&
            (nonDuplIndices !! x) == shapeSize i)
            (indexed nextf)))
```

A.4 Bag BFS 2

```
bagBfs2 :: Graph2 -> Exp Int -> Acc (Vector Int)
bagBfs2 gr s = let
    n  = constant (P.length (P.fst gr))
    vs = fill (I0 ::. n) (constant (-1))
    fr = bagFill (lift (1 :: Int)) s
    T3 _ out _ = awhile
        (\(T3 f _ _) -> unit (not (bagNull f)))
        (bagBody2 gr)
        (T3 fr vs (unit (constant 0)))
    in out

bagBody2 :: Graph2
         -> Acc (Bag Int, Vector Int, Scalar Int)
         -> Acc (Bag Int, Vector Int, Scalar Int)
bagBody2 gr (T3 fr vs lvl) = T3
    nextFrontier
    nvs
    (map (+1) lvl)
    where
        nvs = scatter (bagToArray fr) vs (fill (I1 $ bagSize fr) (the lvl))
        avs :: Acc (Vector Int)
        avs = use (fromList (Z :. P.length (P.fst gr)) (P.fst gr))
        es :: Acc (Vector Int)
        es = use (fromList (Z :. P.length (P.snd gr)) (P.snd gr))
        nextf :: Acc (Vector Int)
        nextf = expand (\x -> (avs !! min (x+1) (length avs - 1)) - (avs !! x))
            (\x y -> es !! ((avs !! x) + y)) (bagToArray fr)
        nonDuplIndices = scatter nextf
            (fill (constant (Z  :. P.length (P.fst gr))) (-1)) (enumFromN (shape nextf) 0)
        nextFrontier :: Acc (Bag Int)
        nextFrontier = arrayToBag $ map snd $ afst $ filter (\(T2 i x) ->
            (nvs !! x) < constant 0 &&
            (nonDuplIndices !! x) == shapeSize i)
            (indexed nextf)
```

A.5 Vector BFS 3

```
bfs3:: Graph1 -> Exp Int -> Acc (Vector Bool)
bfs3 gr s = let
    n = constant (P.fst gr)
    vs = fill (I0 ::. n) (constant False)
    fr = fill (lift $ Z :. (1 :: Int)) s
    T2 _ out = awhile
        (\(T2 f _) -> unit (not (null f)))
        (body3 gr)
        (T2 fr vs)
    in out

body3 :: Graph1 -> Acc (Vector Int, Vector Bool) -> Acc (Vector Int, Vector Bool)
body3 gr (T2 fr vs) = T2
    nextFrontier
    nvs
    where
        nvs = scatter fr vs (fill (shape fr) (constant True))
        es :: Acc (Vector (Int, Int))
        es = use (fromList (Z  :. P.length (P.snd gr)) (P.snd gr))
        rs :: Acc (Matrix (Int, (Int, Int)))
        rs = combine fr es
        nextfr :: Acc (Vector Int)
        nextfr = map (snd . snd)
                       (afst (filter (\t -> fst t == fst (snd t)) rs))
        nonDuplIndices :: Acc (Vector Int)
        nonDuplIndices = scatter nextfr (fill (constant (Z  :. P.fst gr)) (-1))
            (enumFromN (shape nextfr) 0)
        nextFrontier :: Acc (Vector Int)
        nextFrontier = map snd $ afst $ filter (\(T2 i x) ->
            not (nvs !! x) &&
            (nonDuplIndices !! x) == shapeSize i)
            (indexed nextfr)

combine :: (Elt a, Elt b) => Acc (Vector a) -> Acc (Vector b) -> Acc (Matrix (a, b))
combine a b = let
    na = length a
    nb = length b
    ra = replicate (Z_ ::. constant All ::. nb) a
    rb = replicate (Z_ ::. na ::. constant All) b
    rs = zip ra rb
    in rs
```

A.6 Bag BFS 3

```
bagBfr3 :: Graph1 -> Exp Int -> Acc (Bag Int)
bagBfr3 gr s = let
    vs = emptyBag
    fr = bagFill (lift (1 :: Int)) s
    T2 _ out = awhile
        (\(T2 f _) -> unit $ not (bagNull f))
        (bagBody3 gr)
        (T2 fr vs)
    in out

bagBody3 :: Graph1 -> Acc (Bag Int, Bag Int) -> Acc (Bag Int, Bag Int)
bagBody3 gr (T2 fr vs) = T2
    nextFrontier
    nvs
    where
        nvs = bagUnion fr vs
        es :: Acc (Bag (Int, Int))
        es = arrayToBag $ use (fromList (Z  :. P.length (P.snd gr)) (P.snd gr))
        ps :: Acc (Bag (Int, (Int, Int)))
        ps = cartesian fr es
        nextf :: Acc (Vector Int)
        nextf = bagToArray $ bagMap (snd . snd)
            (bagFilter (\t -> fst t == fst (snd t)) ps)
        nonDuplIndices :: Acc (Vector Int)
        nonDuplIndices = scatter nextf (fill (constant (Z  :. P.fst gr)) (-1))
            (enumFromN (shape nextf) 0)
        filteredFrontier :: Acc (Bag Int)
        filteredFrontier = arrayToBag $ map snd $ afst $ filter (\(T2 i x) ->
            (nonDuplIndices !! x) == shapeSize i)
            (indexed nextf)
        nextFrontier :: Acc (Bag Int)
        nextFrontier = bagSubtract filteredFrontier nvs
```

A.7 Vector BFS 4

```
bfs4:: Graph2 -> Exp Int -> Acc (Vector Bool)
bfs4 gr s = let
    n = constant (P.length (P.fst gr))
    vs = fill (       I0 ::. n       ) (constant False)
    fr = fill (lift $ Z   :. (1 :: Int)) s
    T2 _ out = awhile
        (\(T2 f _) -> unit (not (null f)))
        (body4 gr)
        (T2 fr vs)
    in out

body4 :: Graph2 -> Acc (Vector Int, Vector Bool) -> Acc (Vector Int, Vector Bool)
body4 gr (T2 fr vs) = T2
    nextFrontier
    nvs
    where
        nvs = scatter fr vs (fill (shape fr) (constant True))
        avs :: Acc (Vector Int)
        avs = use (fromList (Z  :. P.length (P.fst gr)) (P.fst gr))
        es :: Acc (Vector Int)
        es = use (fromList (Z  :. P.length (P.snd gr)) (P.snd gr))
        nextf :: Acc (Vector Int)
        nextf = expand (\x -> (avs !! min (x+1) (length vs - 1)) - (avs !! x))
            (\x y -> es !! ((avs !! x) + y)) fr
        nonDuplIndices :: Acc (Vector Int)
        nonDuplIndices = scatter nextf (fill (constant (Z  :. P.length (P.fst gr))) (-1))
            (enumFromN (shape nextf) 0)
        nextFrontier :: Acc (Vector Int)
        nextFrontier = map snd (afst (filter (\(T2 i x) ->
            not (nvs !! x) &&
            (nonDuplIndices !! x) == shapeSize i)
            (indexed nextf)))
```

A.8 Bag BFS 4

```
bagBfr4 :: Graph2 -> Exp Int -> Acc (Bag Int)
bagBfr4 gr s = let
    vs = emptyBag
    fr = bagFill (lift (1 :: Int)) s
    T2 _ out = awhile
        (\(T2 f _) -> unit $ not (bagNull f))
        (bagBody4 gr)
        (T2 fr vs)
    in f'

bagBody4 :: Graph2 -> Acc (Bag Int, Bag Int) -> Acc (Bag Int, Bag Int)
bagBody4 gr (T2 fr vs) = T2
    nextFrontier
    nvs
    where
        nvs = bagUnion fr vs
        avs :: Acc (Vector Int)
        avs = use (fromList (Z :. P.length (P.fst gr)) (P.fst gr))
        es :: Acc (Vector Int)
        es = use (fromList (Z :. P.length (P.snd gr)) (P.snd gr))
        nextf :: Acc (Vector Int)
        nextf = expand (\x -> (avs !! min (x+1) (length avs - 1)) - (avs !! x))
            (\x y -> es !! ((avs !! x) + y)) (bagToArray fr)
        nonDuplIndices :: Acc (Vector Int)
        nonDuplIndices = scatter nextf (fill (constant (Z  :. P.length (P.fst gr))) (-1))
            (enumFromN (shape nextf) 0)
        filteredFrontier :: Acc (Bag Int)
        filteredFrontier = arrayToBag $ map snd $ afst $ filter (\(T2 i x) ->
            (nonDuplIndices !! x) == shapeSize i)
            (indexed nextf)
        nextFrontier :: Acc (Bag Int)
        nextFrontier = bagSubtract filteredFrontier nvs
```

A.9 Vector Kahn 1

```
kahn :: Graph1 -> Acc (Vector Int)
kahn gr = let
    vs = enumFromN (I0 ::. constant (P.fst gr)) 0
    es = use $ fromList (Z  :.  P.length (P.snd gr)) (P.snd gr)
    list = use (fromList (Z :. 0) [] :: Vector Int)
    yesIn = map snd es
    rs :: Acc (Matrix Bool)
    rs = map (\(T2 x y) -> x == y) $ combine vs yesIn
    noIn = map not $ or rs
    start = afst $ compact noIn vs
    T3 out _ es' = awhile
        (\(T3 _ n _) -> unit $ not (null n))
        kahnBody
        (T3 list start es)
    in acond (null es') out (use (fromList (Z :. 0) []))

kahnBody :: Acc (Vector Int, Vector Int, Vector (Int, Int))
         -> Acc (Vector Int, Vector Int, Vector (Int, Int))
kahnBody (T3 ls ni es) = T3 nls nni eNFrNi
    where
        nls    = ls ++ ni
        comp   = map (\(T2 (T2 x _) y) -> x == y) $ combine es ni
        fromNi = and comp
        eFrNi  = afst $ compact fromNi es
        eNFrNi = afst $ compact (map not fromNi) es
        pni    = map snd eFrNi
        m2     = map (\(T2 y (T2 _ x)) -> x == y) $ combine pni eNFrNi
        nNoIn  = map not $ or m2
        nni    = afst $ compact nNoIn pni

combine :: (Elt a, Elt b) => Acc (Vector a) -> Acc (Vector b) -> Acc (Matrix (a, b))
combine a b = let
    na = length a
    nb = length b
    ra = replicate (Z_ ::. constant All ::. nb) a
    rb = replicate (Z_ ::. na ::. constant All) b
    rs = zip ra rb
    in rs
```

A.10 Bag Kahn 1

```
bagKahn :: Graph1 -> Acc (Vector Int)
bagKahn gr = let
    vs = arrayToBag $ enumFromN (I0 ::. constant (P.fst gr)) 0
    es = arrayToBag $ use $ fromList (Z  :.  P.length (P.snd gr)) (P.snd gr)
    list = use (fromList (Z :. 0) [] :: Vector Int)
    yesIn = bagMap snd es
    start = bagSubtract vs yesIn
    T3 out _ es' = awhile
        (\(T3 _ n _) -> unit $ not (bagNull n))
        bagKahnBody
        (T3 list start es)
    in acond (bagNull es') out (use (fromList (Z :. 0) []))

bagKahnBody :: Acc (Vector Int, Bag Int, Bag (Int, Int))
            -> Acc (Vector Int, Bag Int, Bag (Int, Int))
bagKahnBody (T3 ls ni es) = T3 nls nIVFNI nfromNI
    where
        nls     = ls ++ bagToArray ni
        fromNI  = bagMap snd $ bagFilter (\(T2 x (T2 y _)) -> x == y) $ cartesian ni es
        vFromNI = bagMap snd fromNI
        nYesIn  = bagMap snd nfromNI
        nIVFNI  = bagSubtract vFromNI nYesIn
        nfromNI = bagSubtract es fromNI
```

A.11 Vector Kahn 2

```
kahn2 :: Graph2 -> Acc (Vector Int)
kahn2 gr = let
    vs = enumFromN (I0 ::. constant (P.length (P.fst gr))) 0
    avs = use (fromList (Z  :. P.length (P.fst gr)) (P.fst gr))
    es :: Acc (Vector (Int, Int))
    es = zip
            (expand (\x -> (avs !! min (x+1) (length avs - 1)) - (avs !! x)) const vs)
            yesIn
    list = use (fromList (Z :. 0) [] :: Vector Int)
    yesIn = use $ fromList (Z  :.  P.length (P.snd gr)) (P.snd gr) -- has duplicates
    rs :: Acc (Matrix Bool)
    rs = map (\(T2 x y) -> x == y) $ combine vs yesIn
    noIn = map not $ or rs
    start = afst $ compact noIn vs
    T3 out _ es' = awhile
        (\(T3 _ n _) -> unit $ not (null n))
        kahn2Body
        (T3 list start es)
    in acond (null es') out (use (fromList (Z :. 0) []))

kahn2Body :: Acc (Vector Int, Vector Int, Vector (Int, Int))
          -> Acc (Vector Int, Vector Int, Vector (Int, Int))
kahn2Body (T3 ls ni es) = T3 nls nni eNFrNi
    where
        nls    = ls ++ ni
        ba     = map (\(T2 (T2 x _) y) -> x == y) $ combine es ni
        fromNi = and ba
        eFrNi  = afst $ compact fromNi es
        eNFrNi = afst $ compact (map not fromNi) es
        pni    = map snd eFrNi
        m2     = map (\(T2 y (T2 _ x)) -> x == y) $ combine pni eNFrNi
        nNoIn  = map not $ or m2
        nni    = afst $ compact nNoIn pni

combine :: (Elt a, Elt b) => Acc (Vector a) -> Acc (Vector b) -> Acc (Matrix (a, b))
combine a b = let
    na = length a
    nb = length b
    ra = replicate (Z_ ::. constant All ::. nb) a
    rb = replicate (Z_ ::. na ::. constant All) b
    rs = zip ra rb
    in rs
```

A.12 Bag Kahn 2

```
bagKahn2 :: Graph2 -> Acc (Vector Int)
bagKahn2 gr = let
    vs = enumFromN (I0 ::. constant (P.length (P.fst gr))) 0
    avs = use (fromList (Z  :. P.length (P.fst gr)) (P.fst gr))
    es = arrayToBag $ zip
            (expand (\x -> (avs !! min (x+1) (length avs - 1)) - (avs !! x)) const vs)
            yesIn
    yesIn = use $ fromList (Z  :.  P.length (P.snd gr)) (P.snd gr)
    list = use (fromList (Z :. 0) [] :: Vector Int)
    start = bagSubtract (arrayToBag vs) (arrayToBag yesIn)
    T3 out _ es' = awhile
        (\(T3 _ n _) -> unit $ not (bagNull n))
        bagKahnBody2
        (T3 list start es)
    in acond (bagNull es') out (use (fromList (Z :. 0) []))

bagKahnBody2 :: Acc (Vector Int, Bag Int, Bag (Int, Int))
             -> Acc (Vector Int, Bag Int, Bag (Int, Int))
bagKahnBody2 (T3 ls ni es) = T3 nls nIVFNI nfromNI
    where
        nls     = ls ++ bagToArray ni
        fromNI  = bagMap snd $ bagFilter (\(T2 x (T2 y _)) -> x == y) $ cartesian ni es
        vFromNI = bagMap snd fromNI
        nYesIn  = bagMap snd nfromNI
        nIVFNI  = bagSubtract vFromNI nYesIn
        nfromNI = bagSubtract es fromNI
```

References

[Bac78] John Backus. The history of fortran i, ii, and iii. *ACM Sigplan Notices*, 13(8):165–180, 1978.

[BEKS17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[BMRSC17] Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Yu Chansu. Efficient algorithms for stream compaction on gpus. *International Journal of Networking and Computing*, 7(2):208–226, 2017.

[Bow06] Kenneth P Bowman. *An Introduction to Programming with IDL: Interactive data language*. Elsevier, 2006.

[CKL+11] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14, 2011.

[CL98] Yves Caseau and François Laburthe. *Introduction to the CLAIRE programming language*. Ecole Normale Supérieure (Paris). Laboratoire d'Informatique [LIENS], 1998.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[Cro08] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts.* " O'Reilly Media, Inc.", 2008.

[Dat89] Chris J Date. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[dWvBKM24] Ivo Gabe de Wolff, David P. van Balen, Gabriele K. Keller, and Trevor L. McDonell. Zero-overhead parallel scans for multi-core cpus. In *Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '24, page 52–61, New York, NY, USA, 2024. Association for Computing Machinery.

[EBH+97] John Wesley Eaton, David Bateman, Søren Hauberg, et al. *Gnu octave*. Network thoery London, 1997.

[FI78] Adin D Falkoff and Kenneth E Iverson. The evolution of apl. *ACM SIGAPL APL Quote Quad*, 9(1):30–44, 1978.

[HH16] Desmond J Higham and Nicholas J Higham. *MATLAB guide*. SIAM, 2016.

[HSE+17] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 556–571, 2017.

[Hut16] Graham Hutton. Programming in haskell, 2016.

[IG96] Ross Ihaka and Robert Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[Kah62] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, 2006.

[SDDS12] Jacob T Schwartz, Robert BK Dewar, Edward Dubinsky, and Edith Schonberg. *Programming with sets: An introduction to SETL*. Springer Science & Business Media, 2012.

[VRD03] Guido Van Rossum and Fred L Drake. *An introduction to Python*. Network Theory Ltd. Bristol, 2003.