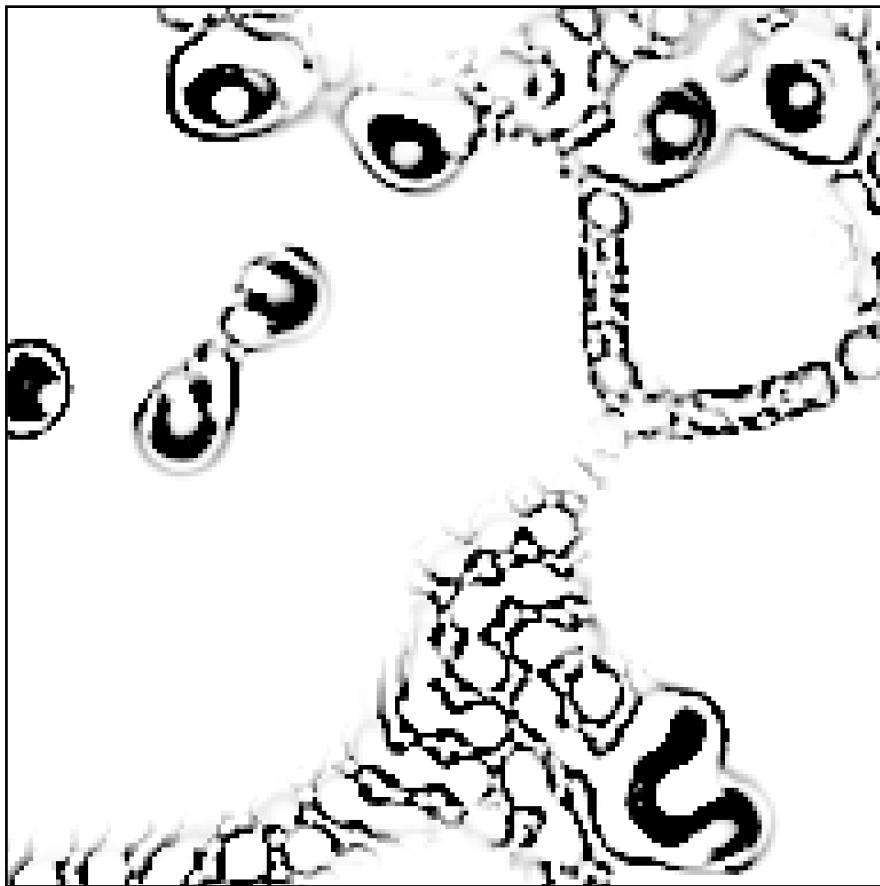


Streaming Irregular Arrays

ROBERT CLIFTON-EVEREST

A THESIS IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
UNIVERSITY OF NEW SOUTH WALES

MAY 2018

Thesis/Dissertation Sheet

Surname/Family Name : **Clifton-Everest**

Given Name/s : **Robert Leonard**

Abbreviation for degree as give in the University calendar : **PhD**

Faculty : **Faculty Of Engineering**

School : **School of Computer Science and Engineering**

Thesis Title : **Streaming irregular arrays**

Abstract 350 words maximum: (PLEASE TYPE)

It is well established that declarative array languages are a high-level and expressive means of writing programs for parallel architectures. However, they are not without their disadvantages. Flat data parallel array languages are limited in their modularity as they do not support nested arrays, and parallel functions cannot be called from within parallel contexts. Nested data parallel languages solve this problem, but they also assume irregularity of all nested structures. This places a cost on nesting, a cost that is needlessly paid for a certain class of programs, those where nesting is strictly regular.

A second problem is that arrays, by definition, allow for random access. This means that they must be loaded into memory in their entirety. If memory is limited, as is the case with GPUs, array languages offer no high-level means of expressing programs containing structures too large for that memory but do not require random access.

The research in this dissertation describes an extension to the Accelerate language: irregular array sequences. They allow for both a limited, but still useful, form of nesting and a streaming execution model that can work under limited memory.

Furthermore, in realising irregular array sequences, we describe a generalised program flattening (vectorisation) transform that does not introduce the unnecessary cost on nesting that is incurred for strictly regular (sub)programs. This transform applies to a much broader domain than just array sequences.

As a further complement to sequences, this dissertation also describes two other extensions to Accelerate, a foreign function interface (FFI) and GPU-aware garbage collection. The former is the first instance of an embedded domain specific language having an FFI, and both are of considerable practical value to programmers using Accelerate framework.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).



Signature



Witness Signature

20/05/2018

Date


The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

**FOR OFFICE USE
ONLY**

Date of completion of requirements for
Award:

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed 

Date 20/05/2018

Contents

Contents	v
Figures	ix
Tables	xi
Abstract	xiii
Acknowledgements	xv
Publications	xvii
1 Introduction	1
2 Background	7
2.1 The Nested Data Parallel (NDP) programming model	7
2.1.1 Array representation	8
2.1.2 Flattening	10
2.1.3 Avoidance	13
2.1.4 Fusion	13
2.2 Accelerate	14
2.2.1 Overview	14
2.2.2 Internals	18
2.3 CUDA	22
3 Sequences	25
3.1 Limitations of array-based flat data parallelism	25
3.1.1 Modularity	25
3.1.2 Memory Usage	27
3.2 Irregular array sequences	27
3.2.1 Irregularity	29
3.2.2 Streaming	30

4	Flattening	33
4.1	Acc_{NDP}	34
4.1.1	Literals	36
4.1.2	Variables and let bindings	37
4.1.3	Tuples	37
4.1.4	Scalar Operations	38
4.1.5	Shapes and extents	38
4.1.6	Array indexing	40
4.1.7	Array extents	40
4.1.8	Array generation	41
4.1.9	Array reduction	41
4.2	Segmented arrays	41
4.3	Regularity-identifying flattening	44
4.3.1	Lifted type relation	46
4.3.2	The lifting transformation	50
4.3.3	Array reduction	60
5	Implementation and Optimisation	61
5.1	Architecture	61
5.1.1	What is a sequence?	62
5.2	Flattening	63
5.2.1	Types	63
5.2.2	Segment representation	65
5.2.3	Transformation	66
5.2.4	Lifted operations	67
5.3	Fusion	70
5.3.1	Array fusion	70
5.3.2	Fusion through tuples	72
5.4	Scheduling	74
6	Enhancements	77
6.1	Foreign Function Interface	77
6.1.1	Importing foreign functions	78
6.1.2	Executing foreign functions with Accelerate	79
6.1.3	Exporting functions	80
6.2	GPU aware garbage collection	82
6.2.1	When to transfer data?	82
6.2.2	The previous solution	83
6.2.3	Array eviction	83

6.2.4	Discussion	84
7	Evaluation	85
7.1	Sparse-Matrix Vector Multiplication (SMVM)	86
7.2	Audio processor	89
7.3	MD5 Hash	89
7.4	PageRank	90
8	Conclusion	93
8.1	Contribution	94
8.1.1	Regularity identifying flattening	94
8.1.2	Irregular array sequences	94
8.1.3	GPU-aware garbage collection	94
8.1.4	A Foreign Function Interface for an embedded language	95
8.2	Future work	95
8.2.1	More sequence operations	95
8.2.2	More nested structures	95
8.2.3	Multi-GPU parallelism	95
	Bibliography	97

Figures

4.1	The grammar of ACC_{NDP}	34
4.2	Type level predicates	34
4.3	The typing rules for ACC_{NDP}	35
4.4	The typing rules for segmented operations in ACC_{NDP}	36
4.5	Lifting variables into higher dimensions.	52
4.6	Auxiliary operations implemented as meta-functions	53
4.7	The lifting transformation	54
4.8	Forced lifting	58
7.1	SMVM of Queen_4147 dataset	88
7.2	Audio processor	90
7.3	MD5 hash recovery	91
7.4	PageRank analysis of Wikipedia (English)	91

Tables

7.1	Benchmark summary. Execution times in milliseconds.	86
7.2	Overview of sparse matrices tested and results of the benchmark. Measurements in GFLOPS/s (higher is better). Columns labelled $N=n$ are CPU implementations using n threads. Competitor implementations are Intel MKL on the CPU and NVIDIA cuSPARSE on the GPU.	87

Abstract

Flat data parallel array languages suffer from poor modularity. Despite being established as a high-level and expressive means of programming parallel architectures, the fact they do not support nested arrays, and that parallel functions cannot be called from within parallel contexts limits their usefulness to only a select few domains. Nested data parallel languages solve this problem, but they also assume irregularity of all nested structures. This places a cost on nesting, a cost that is needlessly paid for a certain class of programs, those where nesting is strictly regular.

A second limitation of such languages is that arrays, by definition, allow for random access. This means that they must be loaded into memory in their entirety. If memory is limited, as is the case with GPUs, array languages offer no high-level means of expressing programs containing structures too large for that memory but do not require random access.

This dissertation describes an extension to the Accelerate language: irregular array sequences. They allow for a limited, but still useful, form of nesting as well a streaming execution model that can work under limited memory.

Furthermore, in realising irregular array sequences, we describe a generalised program flattening (vectorisation) transform that does not introduce the unnecessary cost on nesting that is incurred for strictly regular (sub)programs. This transform applies to a much broader domain than just array sequences.

As a further complement to this work, this dissertation also describes two other extensions to Accelerate, a foreign function interface (FFI) and GPU-aware garbage collection. The former is the first instance of an embedded domain specific language having an FFI, and both are of considerable practical value to programmers using Accelerate framework.

Acknowledgements

My supervisors, Gabi and Manuel, for their expertise and support throughout a long 5 years.

My family, not for any expertise, but certainly for their support, for an even longer period.

More broadly, the Programming Languages and Systems group at UNSW: Gabi, Manuel, Trevor, Amos, Ben, Liam, Zilin, and the many others who came and went in my time there. Their support and advice were invaluable. Being a member has been a formative experience for me. Sadly the group is coming to an end as its members move on to better opportunities. It will not, however, be forgotten. It has too great a legacy for that.

Publications

During the course of my research, I coauthored the following publications. I briefly describe my contribution to these papers.

- R. Clifton-Everest, T. L. McDonell, M. M. T. Chakravarty, and G. Keller. Embedding Foreign Code. In *PADL '14: The 16th International Symposium on Practical Aspects of Declarative Languages*, Jan. 2014

Contains a description of my work in designing and implementing a foreign function interface for Accelerate (Section 2.2.1). It explains and justifies the design and evaluates it through a series of benchmarks. This work can be found in this dissertation in Section 6.1.

- F. M. Madsen, R. Clifton-Everest, M. M. T. Chakravarty, and G. Keller. Functional array streams. In *FHPC'15: Workshop on Functional High-Performance Computing*, pages 23–34. ACM, 2015

Describes joint-work with myself and Frederick Meisner Madsen on *regular* sequences (Section 3) and their introduction to Accelerate. The design philosophy is explored along with motivating examples and benchmarks. My primary contribution was in how programs exploiting sequences could be flattened (Section 2.1).

- R. Clifton-Everest, T. L. McDonell, M. M. Chakravarty, and G. Keller. Streaming irregular arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, pages 174–185. ACM, 2017

Builds upon my work in the above by introducing *irregular* sequences and how they relate to flattening. This dissertation goes into detail on this work in Chapters 4, 5 and 7.

Chapter 1

Introduction

A programmer wanting to take advantage of the immense processing power provided by modern parallel hardware has a common but difficult choice to make. They must decide what language to use. If they use a low-level language, it raises many questions: How time-consuming will that be? How much expert knowledge of the hardware is required? What safety guarantees are there? Will programs be portable to the hardware that comes out next year? The answers to these questions are invariably: very, a lot, not many, and no.

Alternatively, they could use a high-level language. These are typically less time consuming to use, require less hardware specific knowledge, give greater safety guarantees, and can offer portability, provided there is a compiler for the target platform. They are, however, not often capable of generating native code with the desired performance properties.

Array languages have proven a robust compromise to this problem. They lift the level of abstraction beyond that of a low-level language, but still, most importantly, offer competitive performance. By expressing computations in terms of parallel arrays, there is a near limitless amount of parallelism available of which a sufficiently smart compiler can take advantage. Programs can be written by domain experts, as opposed to high-performance computing experts.

The canonical example for such languages is the vector dot product, which may be written as:

```
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Here, two different array operations are used to compute the final result. `zipWith (*)` combines the two input arrays, via element-wise multiplication, into one. `fold (+) 0` reduces the array down, adding up all the elements in its input, to a scalar value.

A `zipWith` operation has trivial parallel semantics, as each scalar operation it performs is independent. With `fold` it's not as simple; the final result depends on all inputs. In the general case, a `foldl` or `foldr` precludes parallel execution. However, if the scalar operation is known to be associative then a tree reduction can be used to implement a direction-less

`fold`.

Similarly, other operations like `map`, prefix sums (`scans`), permutations, and stencil convolutions all either have parallel semantics naturally or can be given them by restricting them in a way that still makes them useful.

This programming model is attractive to existing functional programmers as it uses combinators they are already familiar with, and compose nicely to form elegant programs. Where such programmer may be surprised, however, what happens when they *nest* them. We say an array operation is nested if it happens during the execution of another array operation. For example, in `map (map (+1)) xs`, the inner `map` is a nested array operation. Similarly, we say an array is nested if its elements contain other arrays. Both `xs` and the result of the expression are nested arrays.

Specifically, we divide arrays into two categories, flat and nested. Flat arrays contain only elements of the same statically known size. An array of 32-bit integers is flat, as is an array of pairs of 32-bit integers, but an array of arrays of integers is not.

Some information it only makes sense to treat as a flat structure. For example, a bitmap image is most naturally thought of as a 2D array of pixels. It is difficult to imagine a situation where an image would be viewed as an array of rows of pixels. Similarly, nested structures make the most sense for other types of data. For example, a text document would more likely be viewed as an array of lines of characters, and only as a 2D grid of characters when displayed in an editor.

However, there is not always one obvious way to structure data. A matrix, for example, could be treated as a 2D array in one context, an array of rows in another, or an array of columns in yet another. This could be true even within the same program, so there is no one true matrix representation.

Note that here we are not talking about the actual representation of these structures in memory, but rather how the programmer writes programs to traverse and manipulate them. This is a subtle distinction that it is important to highlight. A commonly accepted definition of a type is it is the set of values of which a value that type can hold. However, in most programming languages the type also defines how the value is represented in memory. In practice, if a programmer chooses to define a matrix as an array of rows, the representation in memory will be different than if they had defined it as a 2D array. The compiler is not aware that all the rows are the same length so uses the most general data structure. Thus a performance penalty is paid for nesting, even though in this case it is used only because it happens to be the most natural way to express the program.

In this work, we contend that whether the programmer chooses to use nested structures should entirely be a decision based on usability. The programmer should be able to nest arrays freely, only paying a penalty when actually necessary.

Some languages do partially mitigate this problem by using more efficient nested array

representations. Nested Data Parallel (NDP) languages (like NESL or Data Parallel Haskell) *flatten* nested arrays. This involves separating the structure of a nested array from its data. For example this array

```
{ {A,B,C}, {D,E}, {F,G,H,I}, {}, {J} }
```

could be represented as

```
({3,2,4,0,1}, {A,B,C,D,E,F,G,H,I,J})
```

The first component of the above pair stores the length of each sub-array and the second the actual values¹. These languages flatten arrays like this as it offers considerably more efficiency when compared to classic pointer-based representations.

Flattening removes some of the cost of working with nested data, but not all of it. Maintaining the list of subarray lengths involves extra computation, plus if parallelism is to be preserved across both the inner and outer arrays there are typically scheduling overheads in dealing with inner arrays of different lengths.

Furthermore, this approach typically requires the flattening of array programs as well as the arrays themselves. Programs are transformed to directly use the flattened array representation. By doing this, nested array operations are also removed. To give a short simple example:

```
map (map (+1)) xs
```

would be flattened to

```
(fst xs, map (+1) (snd xs))
```

Here, `fst` and `snd` are used to access the subarray lengths and the values respectively. There is now only one `map`, removing the nested array operation.

This transform is performed to make parallel scheduling of array operations easier. By not having to launch parallel operations from within other parallel operations the workload remains balanced. Moreover, on certain architectures (e.g. GPUs), there is no, or little, hardware support for launching parallel computations from within a parallel context.

By flattening programs in this way, languages are able to support both nested arrays and the nested array operations they imply. However, such flattening transformations treat all nested arrays as if the best representation for them is the one above. Even though nested arrays *can* store elements varying in size, not all of them necessarily *do*. A programmer treating a matrix as an array of rows would pay a penalty for this representation over a flat one.

Because of these issues, many languages opt not to support nested arrays and instead only support flat array programs. If they do support nested array operations, those operations

¹In practice, most modern languages supporting NDP use a more complicated representation than this. This is discussed in Chapter 2

are treated as sequential. However, many useful programs can be expressed using only flat arrays, particularly programs in the image processing domain. Furthermore, programs that would normally rely on nested arrays can be flattened by hand, albeit often obfuscating the original program structure.

A leading language in this category is Accelerate. A highly expressive language rich with features and libraries. Embedded in Haskell, it offers combinators familiar to functional programmers, enabling them to write fast programs that closely resemble conventional Haskell programs. In fact, the code examples above are valid Accelerate programs.

The the dot-product example alongside its type signature demonstrates how Accelerate is embedded in Haskell and the basic constructs of the language.

```
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

The type `Acc a` can be thought of as an Accelerate computation that when evaluated will produce an `a`. So, in this case, each of the argument computations produces a one-dimensional array (`Vector`) of `Float` when evaluated. This function then returns a new computation which, when evaluated, yields a single (`Scalar`) result as output.

The signatures for `fold` and `zipWith` show how the language prevents certain forms of nested parallelism.

```
zipWith :: (Shape sh, Elt a, Elt b)
         ⇒ (Exp a → Exp b → Exp c)
         → Acc (Array sh a)
         → Acc (Array sh b)
         → Acc (Array sh c)
fold :: (Shape sh, Elt e)
      ⇒ (Exp e → Exp e → Exp e)
      → Acc (Array (sh:.Int) e)
      → Acc (Array sh e)
```

`Elt` is a type class that restricts array elements to those of a fixed size. This make nested arrays unrepresentable. Similarly, `Exp` represents scalar computations, those not involving arrays. In this context, products of scalar values are themselves considered scalar. Under this system, it would seem that all nested parallelism is statically excluded. Unfortunately, however, there are certain cases where `Acc` computations can be embedded in `Exp` computations, in particular to support indexing.

These signatures also highlight how Accelerate treats multidimensional arrays. Operations like `zipWith` and `fold` are rank-polymorphic. `zipWith` will element-wise combine any 2 arrays of the same rank. `fold` reduces an array along its innermost dimension. The type variable `sh` corresponds to the *shape* of the array. This will be explored in more depth in Section 2.2, but for now it sufficient to consider a shape as representing the rank of an array. The rank is tracked at the type level, but not the actual dimensions themselves. The `:.Int` operator adds an extra dimension, so if a shape `sh` has represents a rank `n` then `sh:.Int` is `n+1`.

With these combinators and others described in later chapters, it is possible to write many useful programs. Accelerate programs exist for physics simulations, image processing, graph processing, and password cracking, to name a few.

What does all this tell us? Yes, nested parallelism in array languages is possible, but it carries with it an unnecessary extra cost in the common case when the nesting is regular. Not having nesting is possible, and such languages are still useful, but it makes it hard to express certain programs. What we want is the best of both worlds. We want nesting, but only pay the price for it when we have to.

In this dissertation, we build upon Accelerate to widen the domain of useful programs that can be written in it. Not only that but, in doing so, we produce more general results that can be applied to any array language. Primarily, a program transformation for flattening nested parallelism that identifies and tracks the shape of nested structures.

This dissertation builds on existing work on Accelerate, but also array languages generally. During my enrolment as a PhD student, numerous other researchers also contributed to Accelerate and at times I collaborated with them. The research to which I have been a major contributor is listed below. I explicitly state which work is my own and which I did in close collaboration with others:

- We have developed a new parallel structure in addition to the parallel array. Irregular arrays sequences offer a single level of nesting with greater modularity and controlled memory usage (Chapter 3).
- I developed a novel extension to Blelloch’s flattening transform [5, 6] that identifies regular (sub)computations and transforms them according to different rules than irregular computations. By doing this I was able to take advantage of the greater efficiency afforded by regular computations, but fall back to less efficient methods when computations are irregular (Chapter 4).
- I implemented a system for treating the memory on Graphics Processing Units (GPUs) as a cache for GPU computations. A programmer using Accelerate with this feature does not have to be as concerned about their programs working memory fitting into the limited memory of the GPU (Section 6.2).
- I developed an foreign function interface (FFI) for Accelerate. This was the first, to my knowledge, FFI for an embedded language. The technique for doing this applies to all deeply embedded languages with multiple execution backends. In Accelerate, it allows for highly optimised low-level libraries to be called from within a high-level Accelerate program and for Accelerate programs to be called from CUDA C programs (Section 6.1).

Before describing any of this work, however, we first introduce the background and context of our research in Chapter 2.

Chapter 2

Background

EXAMINER: *The background chapter covers the major theoretical concepts, however, I found it at time difficult to understand (more on this below).*

While Accelerate itself is a flat data-parallel language, its design was strongly inspired by the Nested Data Parallel (NDP) model of programming, in particular, the approach taken by Blelloch and Sabot [6] involving program flattening or vectorisation. Furthermore, the sequence extension of Accelerate, which is one of the main topics of this thesis, brings Accelerate closer to that of a Nested Data Parallel language. Therefore, we will start with introducing the NDP programming model and the concept of program flattening (Section 2.1), before describing the Accelerate language and its implementation in Haskell (Section 2.2).

While Accelerate abstracts over concrete architectures and has multiple backends, one that is of particular importance is the backend that targets Nvidia GPUs via the CUDA framework. The sequence extension provides a benefit to that backend, due to the memory limitations of GPUs, and we also introduce some other extensions specifically for it. Hence, we will also give a brief overview of CUDA (Section 2.3).

2.1 The Nested Data Parallel (NDP) programming model

The Nested Data Parallel (NDP) programming model is the most general form of data parallelism. It allows for the arbitrary nesting of parallel operations within other parallel operations, as well as parallel arrays within other parallel arrays. It is an expressive, high level and modular method of parallel programming [4]. A variety of different useful algorithms have been implemented in languages supporting this model [2, 3, 25].

Nested data parallelism (NDP) was first popularised by the NESL language [5], but it currently exists in many different languages, albeit in a restricted form in some. These include Proteus [19], Futhark [27], Nova [17], Manticore [23], Lift [47], Copperhead [7], Nikola [35], and Data Parallel Haskell (DPH) [43]. These languages implement nested parallelism by a variety of means. Manticore has an execution model based on fork-join parallelism. Lift,

Copperhead, and Nikola directly map nested parallel combinators to the parallel hierarchy of GPUs. The rest rely on the program transformation we introduced in Chapter 1. First described by Blelloch and Sabot [6], the flattening transform flattens nested parallel programs while preserving behaviour. This technique is the one we use in our work, so will describe it here.

Blelloch’s Transform has two distinct components. The first being a *flattened array representation*, the second, program flattening. We will address each of these in turn.

2.1.1 Array representation

In practice, computing with nested arrays requires a means for representing nested structure in the flat memory provided by the von Neumann model we are used to.

This is an example of a simple nested array:

```
{ {A,B,C}, {D,E}, {F,G,H,I}, {}, {J} }
```

Let us assume that A to J are scalar elements. That gives this array one level of nesting. The immediately obvious way of representing this is with an array of pointers. Indeed, most contemporary general-purpose languages represent nested arrays in this way. Note that this is distinct from multidimensional arrays. While one can treat, say, a 2D array as an array of arrays, the fact that each *subarray* has to be the same length does not make it a true nested array structure.

So if an array of pointers allows us to represent arbitrary nested structures, what is the problem? It lies in the fact that modern computer hardware is optimised for locality of reference. Having different (often small) parts of the structure spread over memory with pointers between them means that any operations performed over them have to continually chase pointers and access new areas of memory. This results in many cache misses and is unworkable in the context of high performance. In addition, allocating pointer-based structures involves numerous small allocations. This means construction and destruction is expensive, and, in the context of a garbage collector, longer GC pauses.

The alternative solution to this problem, first proposed by Blelloch and Sabot [6], is *flattening* the array structure, splitting it into an array of *segment descriptors* and an array of values. The example above is represented as:

```
({3,2,4,0,1}, {A,B,C,D,E,F,G,H,I,J})
```

The segment descriptors in the *segments* array hold the size of corresponding subarray. So, we know that the first subarray contains the first 3 elements of the values vector, the next subarray 2, the next 4, etc.

Further nesting levels can then be represented by adding additional segment descriptors. For example, in this array with a nesting depth of 2:

```
{ { {A,B}, {C} },
  { {D,E}, {}, {F} } }
```

becomes,

```
({2,3}, ({2,1,2,0,1}, {A,B,C,D,E,F}))
```

The segment descriptors in both of these examples are what we will refer to as *size segment descriptors*, as they hold the size of subarrays. However, they are somewhat limited. Many operations with arrays in this form have worse complexity than the pointer-based equivalent. In particular, indexing, one of the most basic array operations, is no longer constant time. For example, finding the element at index 3 of the subarray at index 2 of

```
({3,2,4,0,1}, {A,B,C,D,E,F,G,H,I,J})
```

involves taking the size of all subarrays before the one at index 2 and adding them together. In this case that's $3 + 2 = 5$. Then we have to add on 3 to that, $5 + 3 = 8$. This gives us the index of I, which is what we want. In general, size segment descriptors require at $O(n)$ (where n is the number of subarrays) work to perform indexing.

There is another form of segment descriptor that solves this problem. Instead of tracking the size of each subarray, it holds the offset. So, for our first example,

```
({0,3,5,9,9}, {A,B,C,D,E,F,G,H,I,J})
```

We will call these *offset segment descriptors*. In general, it takes $O(1)$ work to index arrays in this form as we already know where the corresponding subarray is located in the values vector.

Another useful operation, calculating the size of a given subarray, is similarly constant time. Merely take the offset of the subarray and subtract from the offset of the next subarray. However, this does not work for the final subarray and requires accessing two different elements of the segments array, which can be a problem for fusion (more on that in Chapter 5). Thus, for these practical reasons, the preferred choice is *size-offset segment descriptors* where both the size and the offset are stored.

```
({(3,0),(2,3),(4,5),(0,9),(1,9)}, {A,B,C,D,E,F,G,H,I,J})
```

This has the advantages of both representations, but does require twice as much memory to store the segment descriptors.

It is worth noting that there are other, more complex, nested array representations in existence. In particular Lippmeier et al. [32] describe one that introduces the concept of *virtual segments*. They allow for arrays to be replicated without explicitly storing them more than once. This problem arises when trying to support higher-order nested parallelism: when arrays can store functions as elements. In the context of this work, we do not support higher-order nested parallelism, so we will not explore it in detail.

2.1.2 Flattening

Flattening, sometimes referred to as vectorisation, is a syntax-driven program transformation. When given a program as input it outputs a flat data-parallel program which is semantically equivalent but operates on a flattened array representation. Most, but not all (Proteus [19] being the exception), languages that rely on a flattened array representation also perform a transform like this. In Chapter 1 we first introduced this idea, but before describing it in more detail, we must first discuss *why* it is a necessary in so many cases.

As an example, consider this simple program.

```
foo xs = map (map (\x → 2*x + 1)) xs
```

How might this program be scheduled on parallel hardware? To start with, let's simply assume that a `map` operation can either be executed in parallel or sequentially. In the sequential case, the entire input array is processed on one thread, one element at a time, equivalent to implementing the `map` as a simple `for` loop. In the parallel case, we assume that at least some of the elements of the array are processed in parallel. There are many possible ways that this could be done, but for the purposes of this explanation we only consider the fact that, for a simple `map`, the parallelism available can be exploited, at least to some degree. The best means by which an array can be divided up amongst multiple processors can depend on the hardware, but it satisfies the explanation in this section to assume that an array of length n is divided up amongst k processors by giving each processor a chunk of n/k elements.

Coming back to the example, there are therefore 4 possible ways it could be executed:

1. Both maps *sequentially*
2. The inner map sequentially, the outer map in parallel
3. The outer map sequentially, the inner map in parallel
4. Both maps in *parallel*

Of these strategies, (1) doesn't take advantage of any of the available parallelism, so is not tenable in this work; (2) only works well if `xs` is sufficiently large; and (3) only works if the *elements* of `xs` are sufficiently large. (4) seems the most obvious choice, as it takes advantage of all the available parallelism. However, it assumes no cost in launching a new parallel operation. In practice, this is rarely the case.

The best possible solution would be if we could treat these two nested maps as one single map. This is what the flattening transform offers us. It converts all nested array operations into operations on flat arrays. This means that as long as we have a means to execute a `map` in parallel, we also have a means to execute a program with nested `maps` and can take advantage of all available parallelism.

Before looking at the transform itself, we'll start by rewriting `foo` by hand such that it no longer has the nested `maps` and uses the nested array representation described above.

```
foo' xs = let (segs, vals) = xs
          in (segs, map (\x → 2*x + 1) vals)
```

In this case, we are simply mapping $2x + 1$ for all x in the nested arrays. It is trivial to see that `foo` and `foo'` are equivalent, just with different structures being employed. As can be seen, `foo'` gives us exactly what we want as it preserves all available parallelism, but without requiring nested array operations.

Blelloch's flattening transformation is a method for producing these equivalent programs. It works by *lifting* functions into vector space. That is, given some function

```
f :: a → b
```

it lifts it into the function

```
f↑ :: Vector a → Vector b
```

operating over vectors. Of course, if `a` or `b` are already vectors¹ then this function will be working over the nested representation of the now nested vectors.

Going back to our example, if we will start by lifting the lambda term

```
ℒ[λx → 2*x + 1]
```

For a function, we need to remember the length of the vector it should return. For all lifted functions, the length of the result should be the same as length of the argument. So we keep track of the argument as part of the *lifting context*.

```
λx → ℒx[2*x + 1]
```

We can then use lifted version of the binary operation `+`. We will call this `+↑`.

```
λx → ℒx[2*x] +↑ ℒx[1]
```

For constants, we have to **replicate** that constant out over a new vector. So for 1 and 2, we replicate it out by the length of the lifting context.

```
λx → replicate (length x) 2 *↑ ℒx[x] +↑ replicate (length x) 1
```

All that leaves us with is lifting the variable `x`. In this case, no extra work needs to be done. We just leave it as `x` as it refers to the vector bound as argument to in the lambda term².

```
λx → replicate (length x) 2 *↑ x +↑ replicate (length x) 1
```

This is now the lifted version of our lambda term. Going back to the complete function.

¹In these examples, we are treating vectors and arrays as being the same thing. For now, that is acceptable, but we will refine our definitions of both of these things in the following section.

²In general this is not always true. See Chapter 4 for an in-depth explanation of how variables are treated during the lifting process.

```
foo xs = map (map (\x → 2*x + 1)) xs
```

We want to use lifting to remove the nested `map`. We can do that by first lifting the inner `map` into application of the lifted function.

```
ℒ[map (\x → 2*x + 1)]
```

Because we're now lifting a term that is already a vector, we are lifting into the nested array representation.

```
λ(segs, vals) → (segs, ℒ[λx → 2*x + 1] vals)
```

Essentially, we are able to replace a `map` with the application of the lifted version of its first argument to the values of its second. More generally, the *lifting rule* for `map` is

```
ℒ[map f] = λ(segs, vals) → (segs, ℒ[f] vals)
```

Continuing on with our example, we can substitute the result of $\mathcal{L}[\lambda x \rightarrow 2*x + 1]$ into the lifted version of the inner `map`.

```
λ(segs, vals) → (segs, (λx → replicate (length x) 2 *↑ x +↑ replicate (length x) 1) vals)
```

For the outer `map`, we could perform much the same thing, if we wanted a lifted version of `foo`, but if we instead just want `foo` to be *flattened* then we can replace it with the function above. More generally, this equivalence will always hold

```
map f = ℒ[f]
```

Using it we can rewrite programs with nested `maps` to remove any nesting.

A similar approach can be taken for other array combinators. All that is required is to have a version of that combinator that can operate over the segmented representation. However, this is problematic for `folds`. If we want a parallel `fold` that assumes its first argument is associative, we have to have this lifting rule:

```
ℒ[fold f z arr] = foldseg f z ℒ[arr]
```

For this combinator, we do not lift the `f` or `z` arguments, meaning they cannot contain any nested parallelism. This is a common restriction on NDP languages.

The combinator `foldseg` is a necessary one to support `fold` in NDP languages. It performs a *segmented* fold. It can be thought of as performing a fold for every subvector of a vector of vectors.

```
foldseg :: (a → a → a)
         → a
         → (Vector (Int,Int), Vector a)
         → Vector a
```

In Chapter 4 we will describe a more general form of flattening and formally specify these operations.

2.1.3 Avoidance

One extension to flattening which is important to our work is *vectorisation avoidance*, avoiding vectorisation where necessary. For example, in this simple function from above,

```
foo xs = map (map (\x → 2*x + 1)) xs
```

flattening resulted in

```
λ(segs, vals) → (segs, (λx → replicate (length x) 2 *† x +† replicate (length x) 1) vals)
```

Doing so results in many intermediate arrays. If vectorisation can be avoided for the term

```
λx → 2*x + 1
```

then it is not necessary to construct all the intermediate arrays. Keller et al. [30] describe a way of avoiding vectorisation by having a pre-vectorisation tagging phase tagging sub-computations that do not need to be vectorised. In Chapter 4 we will describe a different solution to vectorisation avoidance, but one that could be improved by taking ideas from Keller et al. [30].

2.1.4 Fusion

Even with vectorisation avoidance, flattened array programs typically still contain many intermediate arrays. The naïve compilation of them quickly drags down performance as much of the program’s runtime is spent reading and writing values to and from memory. This problem is not unique to flattened array programs, but is prevalent in the domain of functional programming and collection-oriented languages generally. These languages, by their very nature, produce more intermediate structures than necessary. They facilitate and encourage programs to be written in that style because of the compositionality it yields.

Fusion (or deforestation) attempts to alleviate this problem by combining adjacent transformations on structures to remove intermediate results. This has been studied extensively, both in the general area of functional programming [18, 24, 38, 48] and specifically for arrays [8, 31, 33, 36].

For simple cases, fusion can be achieved by basic first-order term rewrites. For example, the rule

```
map f (map g) = map (f . g)
```

removes the intermediate structure from applying `map` to the result of another `map`. However, this does not work in the presence of `let`-bindings; for example, something as simple as

```
let ys = map g xs in map f ys
```

may not be fused without additional inlining. Additionally, there is no simple way to remove the intermediate structure in expressions of the form:

```
fold f z (map g xs)
```

At first glance, this may appear untrue, as

```
fold f z (map g xs) = fold (\acc x. f acc (g x)) z xs
```

However, this rule cannot be used for a parallel fold. In order to be executed in parallel, the first argument to a fold must be associative. This is not necessarily the case in the right hand side of this rule, not without knowing more about `g`. In fact, the two arguments to the function may not even be of the same type.

Because of these drawbacks, a more general approach is required. One that does more than just simple rewrites of the syntax tree.

On such approach is simple and well known [14, 22, 26]: represent an array by its size and a function mapping indices to their corresponding values. Fusion then becomes an automatic property of the data representation. This method is used successfully to optimise purely functional array programs in Repa [29, 31] and Accelerate.

However, a straightforward implementation of this approach results in a loss of *sharing*, which was a problem in early versions of Repa [31]. For example, consider the following program:

```
let xs = { ... }
    ys = map f xs
in zipWith g ys ys
```

Every access to an element `ys` will apply the (arbitrarily expensive) function `f` to the corresponding element in `xs`. It follows that these computations will be done *at least* twice, once for each argument in `g`, quite contrary to the programmer's intent. Accelerate's solution to this problem is not to fuse such terms. However, this can prevent fusion in cases where it may actually be the most efficient choice. We will explore this problem in greater depth in Section 5.3.

2.2 Accelerate

As part of our work, we extend the array language *Accelerate*. What follows is an overview of Accelerate from the perspective of a user of the language: how to write programs, how they are executed, what its limitations are, etc. In the subsequent section (2.2.2) we will explore some of the internal design of the compiler.

2.2.1 Overview

Accelerate is a parallel array language consisting of a carefully selected set of operations (combinators) on multidimensional arrays, which can be compiled efficiently to bulk-parallel SIMD hardware. It is *deeply embedded* in Haskell, meaning that we write Accelerate programs

with (slightly stylised) Haskell syntax.

Accelerate code embedded in Haskell is not compiled to SIMD code by the Haskell compiler; instead, the Accelerate library includes a *runtime compiler* that generates parallel SIMD code at application runtime. Currently, there are two complete code-generation backends, one for multi-core CPUs and one for Nvidia GPUs. Both share a significant code base owing to the fact they both work via LLVM.

The collective operations in Accelerate are based on the scan-vector model [13, 45], and consist of multi-dimensional variants of familiar Haskell list operations such as `map` and `fold`, as well as array-specific operations such as index permutations.

For example, this is a simple vector dot-product function written in Accelerate:

```
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```

The function `dotp` consumes two Accelerate computations. The type `Acc a` can be interpreted to mean an Accelerate computation producing an `a`. So, in this case, each of the argument computations produces a one-dimensional array (`Vector`) of `Float` when evaluated. This function then returns a new computation which, when evaluated, yields a single (`Scalar`) result as output.

The Accelerate code for `dotp` is almost identical to what we would write in standard Haskell over lists and is certainly more concise than an explicitly GPU-accelerated or SIMD-vectorized³ low-level dot-product, while still compiling to efficient code [12, 36, 37].

The functions `zipWith` and `fold` are defined by the Accelerate library and have *highly parallel* semantics, supporting up to as many parallel threads as data elements. The type of `fold` is:

```
fold :: (Shape sh, Elt e)
      ⇒ (Exp e → Exp e → Exp e)
      → Exp e
      → Acc (Array (sh :: Int) e)
      → Acc (Array sh e)
```

This definition exposes some other important components to the language. The type class `Shape` indicates that a type is admissible as an array *shape*. In the simple terms, this means that it must be one of 1D, 2D, 3D, etc. The valid members of the `Shape` class are:

```
instance Shape Z
instance Shape sh ⇒ sh :: Int
```

where `::` is the *snoc* constructor defined as

```
data sh :: i = sh :: i
```

It allows us to build lists where the number (and type) of elements is tracked in the type. For

³That is, a program utilising the SSE/AVX instruction set extensions for x86 processors.

example

```
Z :: 1 :: 2 :: 3 :: Z :: Int :: Int :: Int
```

It is called the `snoc` constructor as it is used for lists where the rightmost element is outermost in the structure, as opposed to the traditional `cons` lists in Haskell where the leftmost element is outermost. The `Z` type is just an alternative version of the unit type.

```
data Z = Z
```

An array of shape `Z` is called a scalar array and only has 1 element. This was the result type of `dotp`, via this synonym:

```
type Scalar = Array Z
```

Higher dimension arrays can then be built up with `:..`. Accelerate defines some synonyms for common dimensions:

```
type DIM0 = Z
type DIM1 = DIM0 :: Int
type DIM2 = DIM1 :: Int
type DIM3 = DIM2 :: Int
type DIM4 = DIM3 :: Int
...
```

From this it should be clear that `Vector` is just a 1D array.

```
type Vector = Array DIM1
```

When used to describe the dimensionality of an array, we say that the *shape* is the type of the `snoc`-list and the *extent* a value of that type. So for a 3×4 2D array, its shape would be:

```
Z :: Int :: Int
```

and its extent would be

```
Z :: 3 :: 4
```

Going back to `fold`, we also have the `Elt` type class. This indicates that a type is admissible as an element of an array. It encompasses many primitive types including:

```
instance Elt Int
instance Elt Float
instance Elt Double
instance Elt Char
instance Elt Bool
instance Elt Word8
instance Elt Word16
instance Elt Word32
...
```

In addition, tuples of arity up to 16 are admissible.

```

instance (Elt a, Elt b)           ⇒ Elt (a,b)
instance (Elt a, Elt b, Elt c)    ⇒ Elt (a,b,c)
instance (Elt a, Elt b, Elt c, Elt d) ⇒ Elt (a,b,c,d)
...

```

In fact, `Elt` is user extensible via representation types. We won't cover how that works in detail here, but the salient point is that `Elt` is a truly open type class.

The type signature for `fold` also shows how Accelerate is stratified into two different levels. We have already seen collective *array computations*, represented by terms of the type `Acc t`, but there is also *scalar expressions*, represented by terms of type `Exp t`. More concretely, values of type `Acc t` and `Exp t` do not execute computations directly, rather they represent abstract syntax trees (ASTs) constituting a computation that, once executed, will yield a value of type `t`. Collective operations comprise many scalar operations which are executed in data-parallel, but scalar operations *cannot* initiate collective operations. This stratification helps to statically exclude nested parallelism. We will explore in more detail what this restriction means for modularity in Chapter 3.

Operations

Accelerate offers many different array operations. We give a brief overview of the key ones here.

`map` and `zipWith` behave in much the same way as those same operations over lists. For multidimensional arrays, `zipWith` yields an array of the minimum extent of all dimensions in the array. For example, if array `a` has extent

```
z :: 5 :: 4
```

and array `b` has extent

```
z :: 3 :: 6
```

then the result of

```
zipWith f a b
```

Will have extent

```
z :: 3 :: 4
```

The `fold` we used in `dotp` works by folding along the inner dimension. One can view this as performing a series of many folds. Every vector along the inner dimension is reduced.

`generate` is the most general array operation.

```
generate :: (Shape sh, Elt e) ⇒ Exp sh → (Exp sh → Exp e) → Array sh e
```

It constructs an array of the given extent using the supplied function to fill its elements. For example:

```
evens :: Exp Int → Acc (Vector Int)
evens sz = generate (index1 sz) (\ix → unindex 1ix * 2)
```

This example also shows the scalar operations `index1` and `unindex1`.

```
index1 :: Exp Int → Exp DIM1
unindex1 :: Exp DIM1 → Exp Int
```

They convert between `Ints` and `DIM1s`.

Similarly, we have the operations `lift` and `unlift`. They are generic functions for converting tuples in the meta-language to tuples in the object language, and vice versa. Expressing these functions in the Haskell type system is tricky, but one can think of them in this form.

```
lift    :: (Exp e1, Exp e2, ... , Exp en) → Exp (e1, e2, ... , en)
unlift  :: Exp (e1, e2, ... , en)      → (Exp e1, Exp e2, ... , Exp en)
```

Various forms of prefix sum (`scan`) are supported. One of the main ones is, `scanl`, for performing a left scan.

```
scanl :: (Shape sh, Elt e)
      ⇒ (Exp e → Exp e → Exp e)
      → Exp e
      → Acc (Array (sh:.Int) e)
      → Acc (Array (sh:.Int) e)
```

It too is similar in semantics to `scanl` in Haskell. Like `fold`, it assumes the binary operation is associative and works over the inner dimension.

Other array and scalar operations we will introduce as the need arises.

2.2.2 Internals

The Accelerate compilation pipeline has 4 major stages and uses two different syntax representations. The stages are:

- Sharing recovery – Converts the higher-order abstract syntax (HOAS) into first-order syntax with De Bruijn indices and also recovers sharing.
- Fusion and optimisation – Tries to minimise unnecessary array traversals and optimises for optimal code generation. This produces an AST with explicit delayed arrays.
- Code generation – Generates LLVM code from the AST and passes it off to the LLVM compiler. This stage is different for each backend. The CPU backend generates x86 binaries (with vector instructions) and the PTX backend generates PTX code for CUDA capable GPUs.
- Execution – Schedules and executes the computations on the target hardware.

The first one of these stages we won't explore in detail, as it is not a concern of this

work. It is sufficient to understand it as the process that produces the first-order syntax in Listing 2.1 from the programmer-friendly higher-order syntax. See McDonell et al. [36] for details on how this stage of the pipeline works.

The fusion and optimisation stage will be discussed in Chapter 5 where we will build upon the previous work described in McDonell et al. [36].

Code generation is not discussed in detail as it is not a focus of this work. See McDonell et al. [37] for how LLVM is generated from the AST in Listing 2.1.

Execution will be covered in Chapter 5 where we build upon previous work [12, 36, 37].

Abstract Syntax Tree

The abstract syntax tree for the core Accelerate language is split into two parts, much like the higher-order syntax exposed to the programmer. Listing 2.1 contains the data type for `Acc` stratum, the syntax nodes representing array operations, and Listing 2.2 contains the data type for `Exp`, the nodes for simple scalar operations. Given the number of features that Accelerate offers, there are many different AST nodes. We won't describe all of them here, just describe the basic structure and enough information to form a basis for the remaining chapters.

A key to understanding the definition of the ASTs is know that they are designed to be *type-preserving*, and extensible. Looking at the definition for `Acc` we have:

```
data PreOpenAcc acc aenv a where
```

From right to left, the type parameters are: the type of term, the type of all free variables in the term (assuming it is an open term), and a recursive continuation. We will describe how these are used to construct an AST, but firstly some background.

The AST uses DeBruijn indices [21] to represent variables. This is a common technique when representing ASTs to avoid issues of alpha-equivalence and fresh name generation. Briefly, they work by using the natural numbers in place of variables. The number indicates how many levels of binding outside the super-term the variable is bound. For example

```
let A in
  let B in
    (v_0 + v_1) / 2
```

Here, we're calculating the average of `A` and `B`, where `A` and `B` are normal terms. The let-bindings do not introduce a named variable, but the bound values can be referred to with numbers. `v_0` refers to `B` as it is the closest binding going up the tree, and `v_1` refers to `A` as it is the next binding up.

The typed DeBruijn indices Accelerate uses are defined as

```
data Idx env a where
  ZeroIdx :: Idx (env,a) a
```

```
SuccIdx :: Idx env a
        → Idx (env,b) a
```

The `env` parameter is a left-nested tuple (equivalent to a snoc-list; see 2.2). The AST node for a variable is then:

```
Avar :: Arrays a
      ⇒ Idx aenv a
      → PreOpenAcc acc aenv a
```

New variables are introduced to environment via let-binding:

```
Alet :: Arrays a
      ⇒ acc aenv a
      → acc (aenv,a) b
      → PreOpenAcc acc aenv b
```

These AST nodes also show how the first parameter, the recursive continuation, is used. Rather than simply making the GADT recursive, having an explicit type argument, used whenever recursion is expected, allows additional information and structure to be embedded in the AST. The original recursion can be recovered with `OpenAcc`:

```
newtype OpenAcc aenv a = OpenAcc (PreOpenAcc OpenAcc aenv a)
```

But also, for example, an AST can be annotated with labels for user-debugging,

```
data LabelledAcc aenv a = LabelledAcc String (PreOpenAcc OpenAcc aenv a)
```

We can see the relation between the `Exp` and `Acc` strata by looking at the definition of the `Generate` constructor and expanding the type synonyms within it

```
Generate :: (Elt e, Shape sh)
          ⇒ PreOpenExp acc aenv () sh
          → PreOpenFun acc aenv () (sh → e)
          → PreOpenAcc acc aenv (Array sh e)
```

we observe how both arguments are parameterised by the recursive `acc` step and array environment but have closed scalar environments. We say that these term are open with respect to the array environment but closed with respect to the scalar environment.

In the case of `Index`, we see how the recursive step can be embedded in `Exp` terms. While the type system does not enforce this, for most stages of the compilation we have as an invariant that any `Acc` term embedded in `Exp` must be an `Avar`. This way, scalar computations can access arrays previously computed but cannot compute further array operations.

```
type Acc = OpenAcc ()
newtype OpenAcc aenv a = OpenAcc (PreOpenAcc OpenAcc aenv a)

data PreOpenAcc acc aenv a where
  Map      :: (Elt a, Elt b, Shape sh)
            ⇒ PreFun      acc aenv (a → b)
```

```

→ acc          aenv (Array sh a)
→ PreOpenAcc acc aenv (Array sh b)

ZipWith :: (Elt a, Elt b, Elt c, Shape sh)
⇒ PreFun    acc aenv (a → b → c)
→ acc       aenv (Array sh a)
→ acc       aenv (Array sh b)
→ PreOpenAcc acc aenv (Array sh c)

Generate :: (Elt e, Shape sh)
⇒ PreExp    acc aenv sh
→ PreFun    acc aenv (sh → e)
→ PreOpenAcc acc aenv (Array sh e)

Fold      :: (Elt a, Shape sh)
⇒ PreFun    acc aenv (a → a → a)
→ acc       aenv (Array (sh::Int) a)
→ acc       aenv (Array sh a)
→ PreOpenAcc acc aenv (Array sh a)

Avar      :: Arrays a
⇒ Idx aenv a
→ PreOpenAcc acc aenv a

Alet      :: (Arrays a, Arrays b)
⇒ acc       aenv      a
→ acc       (aenv,a) b
→ PreOpenAcc acc aenv  b

...

```

Listing 2.1: The first-order abstract syntax of the `Acc` level of Accelerate.

```

type Fun = OpenFun ()
type PreFun acc = PreOpenFun acc ()
type OpenFun = PreOpenFun OpenAcc

data PreOpenFun acc env aenv f where
  Body :: Elt b
    ⇒ PreOpenExp acc env      aenv b

Lam :: (Elt a, Elt b)
    ⇒ PreOpenFun acc (env,a) aenv b
    → PreOpenFun acc env      aenv (a → b)

type Exp = OpenExp ()
type PreExp acc = PreOpenExp acc ()
type OpenExp = PreOpenExp OpenAcc

data PreOpenExp acc env aenv e where
  Index :: (Shape sh, Elt e)
    ⇒ acc          aenv (Array sh e)
    → PreOpenExp acc env aenv sh
    → PreOpenExp acc env aenv e

```

```

Let      :: (Elt a, Elt b)
          ⇒ PreOpenExp acc env      aenv a
          → PreOpenExp acc (env,a) aenv b
          → PreOpenExp acc env      aenv b

Var      :: Elt e
          ⇒ Idx env e
          → PreOpenExp env aenv e

Tuple    :: (Elt t, IsTuple t)
          ⇒ Tuple      acc env aenv (TupleRepr t)
          → PreOpenExp acc env aenv t
...

data Tuple acc env aenv t where
NilTup  :: Tuple acc env aenv ()

SnocTup :: Elt a
          ⇒ Tuple      acc env aenv t
          → PreOpenExp acc env aenv a
          → Tuple      acc env aenv (t,a)

```

Listing 2.2: The first-order abstract syntax of the `Exp` level of Accelerate.

2.3 CUDA

The CUDA (Compute Unified Device Architecture)[40] programming environment provides libraries and extensions to LLVM, C and C++, as well as Fortran. They enable programs to be written for and executed on NVIDIA GPUs.

The Accelerate compiler I described above has a backend that generates LLVM IR which is subsequently compiled by the CUDA framework. For explanation, however, examples are given in CUDA C as that is how the majority of programmers use CUDA.

Concretely, a C programmer wanting to take advantage of a CUDA capable GPU would annotate functions in their program with special directives indicating they are intended for GPU execution. When run through the *nvcc* compiler any function annotated with these directives will be compiled into code in the Parallel Thread Execution (PTX) instruction set. These annotated functions, known as kernels, can be called with a special notation that specifies how they should be scheduled, in particular, how many multiprocessors to use. The code in 2.3 shows a simple example where two vectors of size N are added together by N threads.

```

// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

```



```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Listing 2.3: Adding two N length vectors in CUDA C[40]

This example corresponds to this simple Haskell function

```
vecAdd :: [Float] → [Float] → [Float]
vecAdd = zipWith (+)
```

In general, it is reasonably clear how trivially parallel programs, like those that only rely on `maps` and `zipWiths`, can be implemented in CUDA C. It is less clear, however, how to map other high-level combinators (for example `fold` and `scan`) to a corresponding GPU kernel. To do such things, the programmer has to concern themselves with the hierarchy of the GPU. Kernels are executed by threads in SIMD groups called warps arranged in thread blocks. Different threads in the same block can coordinate by fast shared memory, whereas threads in different blocks can only coordinate via the slower global memory. To further complicate the programmer’s task, there are also issues of SIMD divergence and memory access patterns. All of these must be taken into account to achieve optimal performance.

The difficulties of directly programming through the low-level CUDA framework is what inspired many languages, compilers, and frameworks that try to lift the level of abstraction. This was and is the motivation for Accelerate.

Chapter 3

Sequences

EXAMINER: *This chapter makes the case that Accelerate suffers from modularity and memory consumption issues. While I agree that these are serious limitations, and that being able to support irregular arrays is a great improvement, this chapter does not really explain clearly enough why supporting sequences improve modularity, which is one of the key claims that this thesis makes. This chapter needs to make a better case for this and should for instance present the applications presented in the evaluation as individual use-cases, explaining how they would be written in Vanilla Accelerate versus Accelerate+Sequences.*

In this chapter, we introduce *irregular array sequences* as an extension to Accelerate. This extension represents a novel way of overcoming two major limitations of flat data parallelism. We first explore these limitations in depth (Section 3.1) before describing the extension itself (Section 3.2).

3.1 Limitations of array-based flat data parallelism

While flat data parallel array languages, like Accelerate, offer a powerful, high-level means of parallel programming, they are not without drawbacks. In this work, we focus on two of the major ones. Firstly, the reduction of modularity that results from strictly flat data-parallelism (Section 3.1.1), and, secondly, the memory usage that arises from the inherent random access nature of arrays (3.1.2).

3.1.1 Modularity

Recall the simple dot product example from Section 2.2.1:

```
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```

TODO FINAL: *This part is where a reader not familiar with data parallel languages might get lost. Try to introduce it more gently.*

We would like to re-use this function to implement a matrix-vector product, by applying it to each row of the input matrix:

```
mvm_ndp :: Acc (Matrix Float) → Acc (Vector Float) → Acc (Vector Float)
mvm_ndp mat vec =
  let Z :: m :: _ = shape mat
  in generate (Z:.m) (λrow → the $ dotp vec (slice mat (row :: All)))
```

Here, `generate` creates a one-dimensional vector containing `m` elements by applying the supplied function at each index in data-parallel. At each index, we extract the appropriate row of the matrix using `slice`, and pass this to our existing `dotp` function together with the input vector.

The `slice` operation is a generalised array indexing function which is used to cut out *entire dimensions* of an array. In `slice mat (row :: All)`, we extract `All` columns at one specific `row` of the given two-dimensional matrix, resulting in a one-dimensional vector. `the` is a specialised form of indexing for extracting the value from a scalar array.

```
the :: Acc (Scalar a) → Exp a
the = (! Z)
```

Unfortunately, since both `generate` and `dotp` are data-parallel operations, this definition requires nested data-parallelism and is thus not permitted.¹ More specifically, the problem lies with the data-parallel operation `slice` which *depends on* the scalar argument `row`. The clue that this definition includes nested data-parallelism is in the use of `the`. This effectively conceals that `dotp` is a collective array computation to match the type expected by `generate`, which is that of scalar expressions. Consequently, we cannot reuse `dotp` to implement matrix-vector multiplication; instead, we have to write it from scratch using `replicate`

```
mvm_rep :: Acc (Matrix Float) → Acc (Vector Float) → Acc (Vector Float)
mvm_rep mat vec =
  let Z :: m :: _ = shape mat
      vec'      = replicate (Z :: m :: All) vec
  in
    fold (+) 0 ( zipWith (*) mat vec' )
```

The dual of `slice`, `replicate` replicates a given array across an arbitrary set of dimensions. With `replicate (Z :: m :: All) vec`, we are treating `vec` as a row vector and replicating it across the `Y` dimension `m` times.

Similar to this example, computations on irregular structures, such as sparse matrices, which are most naturally expressed in nested form, require an unwieldy encoding when we are restricted to only flat data-parallelism (see Section 3.2.1).

¹By not permitted, we mean it will not compile. It will type check, but the Accelerate compiler will reject it. This happens at Accelerate compile time which is actually Haskell runtime.

3.1.2 Memory Usage

In addition to the loss of modularity, array-based programming languages like Accelerate suffer from another limitation. In particular, they are hampered by resource constraints, such as limited working memory. This is particularly problematic for compute accelerators such as GPUs, which have their own, usually much smaller, high-performance memory areas separate from the host's main memory². An array, by definition, supports constant-time random access. While certain optimisations can be made to avoid this in some cases (e.g. fusion, see Section 5.3), arrays must generally be loaded into device memory in their entirety.

Where algorithmically feasible, such devices require us to split the input into *chunks*, which we stream onto, process, and stream off of the device one by one. This requires a form of nesting if we want to maintain code portability across architectures with varying resource limits, while still writing high-level, declarative code.

3.2 Irregular array sequences

The type of sequences we present are *irregular sequences* of arrays. An irregular sequence of arrays is an ordered collection of arrays, where the extent (the size of the array in each dimension³) of the arrays within one sequence may vary. This is in contrast to *regular sequences* which are sequences that contain arrays that are all the same extent. Clearly, irregular sequences are the more general form, so opting for them in order to widen the domain of programs that we can write is the obvious choice. There are, however, advantages in having sequences that are known to be regular in terms of scheduling and code generation. In chapter 4 we explore this in more detail, but here we will put aside the issue of performance and simply explore what can be expressed with our sequences. Later we will show how opting for the more general form of them does not degrade performance for the subset of programs that do not require irregularity.

We denote sequences as `[Array sh e]` in Accelerate. That can be read as a sequence of arrays of shape `sh` containing elements of type `e`. We also support sequences of array tuples, so `[(Array sh e, Array sh' e')]` is possible as well. This syntax, borrowed from Haskell lists, we use because, in the context of a deep embedding, cannot clash with actual lists.

In Accelerate, we distinguish embedded scalar computations and embedded array computations by the type constructors `Exp` and `Acc`, respectively. Similarly, we use a new constructor, `Seq`, to mark sequence computations. Moreover, just like an array computation of type `Acc (Array sh e)` encompasses many data-parallel scalar computations of type `Exp e`

²The current top-of-the-line NVIDIA Tesla V100 has 16GB of onboard memory; much less than the amount of host memory one can expect in a workstation-class system this product is aimed at. Most other GPUs include significantly less memory.

³While Accelerate uses the term *shape* to refer to both the type level dimensionality and the value level size of each dimension, to avoid ambiguity we will reserve *shape* for the former and use *extent* for the latter.

to produce an array, a sequence computation of type `Seq [Array sh e]` comprises many stream-parallel array computations of type `Acc (Array sh e)`. Specifically, we consider values of type `[Array sh e]` to be sequences (or streams) of arrays, and values of type `Seq [Array sh e]` to be *computations* that, when run, produce sequences of arrays. This is an important distinction. Evaluating a value of type `Seq t` does not trigger any sequence computation, it only yields the *representation* of a sequence computation. To actually trigger a sequence computation, we must consume the sequence into an array computation to produce an array by way of the function:

```
consume :: Arrays arr ⇒ Seq arr → Acc arr
```

To consume a sequence computation, we need to combine the stream of arrays into a single array first — note how the argument of `consume` takes a `Seq arr` and not a `Seq [arr]` and the `Arrays` constraint on `arr`.⁴ Depending on the desired functionality, this can be achieved in a variety of ways. The most common combination functions are `elements`, which combines all elements of all the arrays in the sequence into one flat vector; and `tabulate`, which concatenates all arrays along the outermost dimension (trimming according to the smallest extent along each dimension, much like multi-dimensional uniform `zip`):

```
elements :: (Shape sh, Elt e) ⇒ Seq [Array sh e] → Seq (Vector e)
tabulate :: (Shape sh, Elt e) ⇒ Seq [Array sh e] → Seq (Array (sh:.Int) e)
```

Conversely, we produce a stream of array computations by a function that is not unlike a one-dimensional sequence variant of `generate`:

```
produce :: Arrays a ⇒ Exp Int → (Exp Int → Acc a) → Seq [a]
```

Its first argument determines the length of the sequence and the second is a stream producer function that is invoked once for each sequence element.

In addition to these operations for creating and collapsing sequences, we only need to be able to map over sequences with:

```
mapSeq :: (Arrays a, Arrays b) ⇒ (Acc a → Acc b) → Seq [a] → Seq [b]
```

to be able to elaborate the function `dotp` (the first example of an Accelerate program) into sequence-based matrix-vector multiplication:

```
mvm_seq :: Acc (Matrix Float) → Acc (Vector Float) → Acc (Vector Float)
mvm_seq mat vec =
  let Z :: m :: _ = shape mat
      rows = produce m (λrow → slice mat (Z :: row :: All)) :: Seq [Vector Float]
  in consume (elements (mapSeq (dotp vec) rows))
```

We stream the matrix into a sequence of its rows using `produce`, apply the previously defined dot-product function `dotp` to each of these rows, `combine` the scalar results into a vector with

⁴Recall from Chapter 2 that the type class `Arrays` constrains a type to be either an array or a tuple of arrays

elements, and finally consume that vector into a single array computation that yields the result.

We are also able to combine two sequences with `zipWithSeq`:

```
zipWithSeq :: (Arrays a, Arrays b, Arrays c) => (Acc a -> Acc b -> Acc c) -> Seq [a] -> Seq [b]
```

Naturally, this yields sequence that is equal in length to the shortest of the two inputs.

Although we are re-using Haskell's list type constructor `[]` for sequences in the embedded language, the Accelerate code generator does not actually represent them in the same way. Nevertheless, the notation is justified by our ability to incrementally stream a lazy list of arrays into a sequence of arrays for pipeline processing with:

```
streamIn :: (Shape sh, Elt e) => [Array sh e] -> Seq [Array sh e]
```

3.2.1 Irregularity

The arrays in the sequence used in the implementation of `mvmseq` are all of the same size; after all, they are the individual rows of a dense matrix. Hence, the sequence is *regular*. In contrast, if we use sequences to compactly represent *sparse* matrices, the various sequence elements will be of varying size, representing an irregular sequence or stream computation.

We illustrate this with the example of the multiplication of a sparse matrix with a dense vector. We represent sparse matrices in the popular *compressed row (CSR)* format, where each row stores only the non-zero elements together with the corresponding column index of each element. For example, the following matrix is represented as follows (where indexing starts at zero):

$$\begin{pmatrix} 7 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 2 & 3 \end{pmatrix} \Rightarrow [[(0,7.0)], [], [(1,2.0),(2,3.0)]]$$

Representing our sparse matrix as a sequence of the matrix rows in CSR format, we can define sparse-matrix vector product as:

```
type SparseVector a = Vector (DIM1, a)
type SparseMatrix a = [SparseVector a]

smvmseq :: Seq (SparseMatrix Float) -> Acc (Vector Float) -> Acc (Vector Float)
smvmseq smat vec
  = collect . elements
  $ mapSeq (\srow -> let (ix,vs) = unzip srow in dotp vs (gather ix vec)) smat
```

Here, `gather` is a form of backwards permutation.

```
gather :: Acc (Array sh sh') -> Acc (Array sh' e) -> Acc (Array sh e)
```

As discussed above, when the irregularity is pronounced, we need to be careful with scheduling; otherwise, performance will suffer. We will come back to this issue in Chapter 5.

Again though, we emphasise that whether a sequence is regular or irregular is a detail tracked by the compiler. The programmer can treat all sequences as irregular and be assured that any available regularity will be exploited on their behalf.

3.2.2 Streaming

The `streamIn` function (whose signature was given above) turns a Haskell list of arrays into a sequence or stream of those same arrays. If that stream is not consumed all at once, but rather array by array or perhaps chunkwise (processing several consecutive arrays at once), then the input list will be demanded lazily as the stream gets processed. Similarly, we have the function:

```
streamOut :: Arrays a => Seq [a] -> [a]
```

which consumes the results of a sequence computation to produce an incrementally constructed Haskell list with all the results of all the array computations contained in the sequence. Overall, this allows for stream computations exploiting pipeline parallelism. In particular, the production of the stream, the processing of the stream, and the consumption of the stream can all happen concurrently and possibly on separate processing elements.

Moreover, our support for irregularity facilitates balancing of resources. For example, if the sequence computation is executed on a GPU, the underlying scheduler can dynamically pick chunks of consecutive arrays from the sequence such that it balances two competing constraints. Firstly, modern hardware, in particular, GPUs, offer considerable parallelism. The number of arrays in the chunk needs to expose sufficient parallelism to fully utilise the parallelism available. Secondly, the limits imposed by the relatively small amounts of working memory (on GPUs) are not exceeded. This flexibility, together with the option to map pipeline parallelism across multiple CPU cores, provides the high-level declarative notation that we sought in the introduction.

To illustrate, we have implemented the core of an audio compression algorithm. The computation proceeds by, after some pre-processing, moving a sliding window across the audio data, performing the same computation at each window position. Finally, some post-processing is performed on the results of those windowed computations. As the computations at the various window positions are independent, they may be parallelised. However, each of the windowed computations on its own is also compute intensive and offers ample data-parallelism. Here, we will not go into detail about the pre and post-processing steps nor much about the algorithm itself. We are primarily concerned with the expressiveness of our sequences extension here.

This style of decomposition is common and well suited to a stream processing model. We perform the preprocessing in vanilla Haskell, stream the sequence of windowed computations through Accelerate code, and consume the results for post-processing. In our application,

the stream is irregular, as the information density of the audio waveform varies at different times in the audio stream, which in turn leads to different array sizes for each windowed computation. If we choose to offload the Accelerate stream computations to a GPU, we realise a stream-processing pipeline between CPU and GPU computations.

This is the essential structure of the computation:

```
type AudioData = (tuple of arrays)

zc_core = post_processing . zc_stream . pre_processing

zc_stream :: AudioData → [(Array DIM2 Double, Array DIM2 Double)]
zc_stream audioData = streamOut $ mapSeq (processWindow audioData) windowIndexes
  where
    windowIndexes :: Seq [Scalar Int]
    windowIndexes = produce (sizeof audioData) id

    processWindow :: AudioData → Scalar Int → Acc (Array DIM2 Double, Array DIM2 Double)
    processWindow = ...
```

The core of the algorithm, `zc_core`, consists of the steps of pre-processing the audio data, the Accelerate stream computation, and finally post-processing. The Accelerate stream computation generates a stream of window indexes (`windowIndexes`) using `produce`, maps the windowed data processing function `processWindow` over that sequence using `mapSeq`, and incrementally produces a list of outputs, one per window, with `streamOut`. In the current setup, `pre_processing` must be complete before stream processing can start; we choose to do this because, since the data in successive steps of the sliding window strongly overlap, the approach here is more efficient than explicitly creating a stream of windowed data. Instead, the input stream `windowIndexes` is a sequence of integer values that indicate which window (subset of the input data) the associated sequence computation ought to extract from the input `audioData`. This setup is similar to the use of `generate` and `slice` in `mvmndp` in Section 3.1.1.

In contrast to `pre_processing` and stream generation, we do use pipeline parallelism to overlap the stream processing performed by `mapSeq (processWindow audioData)` with the `post_processing` by using `streamOut`. Hence, we have got three sources of parallelism: (1) `processWindow` contains considerable data-parallelism; (2) the stream scheduler can run multiple array computations corresponding to distinct stream elements in parallel; and (3) `streamOut` provides pipeline parallelism between stream processing and `post_processing`. The second source of parallelism allows the Accelerate runtime considerable freedom in adapting to the resources of the target system, and we will see in Chapter 7 that this is helpful in providing performance portability between multicore CPUs and GPUs.

Chapter 4

Flattening

To efficiently implement the sequence computations described in the previous chapter many issues need to be addressed. This chapter focuses on the most significant one: transforming nested, possibly irregular, data-parallelism into a form that can be efficiently executed on SIMD hardware. We do this via a novel extension to program flattening, one that not only allows for sequence computations but also solves the more general problem of efficient execution of nested arrays in a SIMD context. What distinguishes our flattening transform from others, is that we consider both regular and irregular computation, brought about by our desire to have efficient regular sequences as they have different needs with regard to their representation in memory.

TODO FINAL: *This paragraph mentions things not explained or defined till later. Restructure!*

Blelloch and Sabot [6]’s original version of the flattening transformation was for a first-order language with built-in second-order combinators (NESL) which makes it a good fit for Accelerate which has similar properties. Nevertheless, the original transformation has two severe shortcomings, which makes the generated code non-competitive: (1) it lifts code into vector space that ought to remain as is for best performance; and (2) it does not treat the important special case of operations on regular multi-dimensional arrays specially. We address point (1) by adapting the work on *vectorisation avoidance* [30], and tackle point (2) with a generalisation of flattening that identifies regular (sub)computations and optimises for them. Moreover, this transformation is, in contrast to previous work, type-preserving. This is necessary for integration with Accelerate, which is built around a typed AST and is type preserving in all its compilation stages [37]. It has the added benefit of statically guaranteeing that the transform we describe here produces type-correct programs.

As a basis on which to describe the transform, we first introduce a simple language which we call ACC_{NDP} (Section 2.1). For the sake of simplicity, the language has fewer parallel operations than Accelerate. However, it is also more general in that it supports arbitrarily nested parallel arrays, as the transformation truly is an extension of similar flattening

literals	l	$::=$	$\mathbb{Z} \mid Z$
variables	v	$::=$	$v_0 \mid v_1 \mid \dots$
tuples	t	$::=$	(e_0, \dots, e_n)
unary-ops	p_1	$::=$	$\text{indexHead} \mid \text{indexTail} \mid \text{indexLeft} \mid \text{indexRight}$
binops	p_2	$::=$	$(+) \mid (*) \mid (-) \mid \text{intersect} \mid (:) \mid (\#)$
expressions	e	$::=$	$l \mid v \mid t \mid e_1 ! e_2 \mid p_1 e \mid p_2 e_1 e_2$ $\mid \text{let } v_0 = e_1 \text{ in } e_2$ $\mid \text{prj } \mathbb{N}^+ e$ $\mid \text{extent } e$ $\mid \text{generate } e_1 (\lambda v_0. e_2)$ $\mid \text{fold } (\lambda v_1 v_0. e_1) e_2 e_3$ $\mid s$
segments	s	$::=$	$\text{generate}_{seg} e_1 (\lambda v_0. e_2)$ $\mid \text{fold}_{seg} (\lambda v_1 v_0. e_1) e_2 e_3 e_4$ $\mid \text{segmented } e$ $\mid \text{cross } e_1 e_2$ $\mid e_1 \# e_2$ $\mid \text{intersects } e_1 e_2$ $\mid \text{lefts } e$ $\mid \text{rights } e$

Figure 4.1: The grammar of ACC_{NDP}

$\frac{}{\text{IsShape } Z}$	$\frac{\text{IsShape } sh}{\text{IsShape } (sh :. \text{Int})}$				
$\frac{}{\text{IsScalar } \text{Int}}$	$\frac{\text{IsShape } sh}{\text{IsScalar } sh}$	$\frac{\text{IsScalar } \alpha_1 \quad \text{IsScalar } \alpha_2 \quad \dots \quad \text{IsScalar } \alpha_n}{\text{IsScalar } (\alpha_1, \alpha_2, \dots, \alpha_n)}$			

Figure 4.2: Type level predicates

approaches. The reader will recognise its similarity to Accelerate, but unlike Accelerate we will formally specify its syntax and type system. Following that, we describe our new flattening transformation in terms of ACC_{NDP} , using it as both the source and target language (Section 4.3).

4.1 Acc_{NDP}

Before introducing the precise syntax, let us have a look at the high-level properties to give a general overview of the language:

- **Parallel arrays:** All parallelism is specified with multi-dimensional arrays and data parallel combinators that operate over them. The arrays are immutable, thus avoiding

$\frac{}{\Gamma \vdash Z :: Z}$	$\frac{l \in \mathbb{Z}}{\Gamma \vdash l :: \text{Int}}$	$\frac{\Gamma \vdash ix :: sh \quad \Gamma \vdash i :: \text{Int}}{\Gamma \vdash ix.i :: sh.\text{Int}}$
$\frac{\Gamma \vdash e_1 :: \alpha_1 \quad \Gamma \vdash e_2 :: \alpha_2 \quad \dots \quad \Gamma \vdash e_n :: \alpha_n}{\Gamma \vdash (e_1, e_2, \dots, e_n) :: (\alpha_1, \alpha_2, \dots, \alpha_n)}$		
$\frac{\Gamma \vdash e :: sh.\text{Int} \quad \text{IsShape } sh}{\Gamma \vdash \text{indexHead } e :: \text{Int}}$	$\frac{\Gamma \vdash e :: sh.\text{Int} \quad \text{IsShape } sh}{\Gamma \vdash \text{indexTail } e :: sh}$	$\frac{\Gamma \vdash e_0 :: sh \quad \Gamma \vdash e_1 :: sh \quad \text{IsShape } sh}{\Gamma \vdash \text{intersect } e_0 e_1 :: sh}$
$\frac{\Gamma \vdash e :: sh \# sh' \quad \text{IsShape } sh \quad \text{IsShape } sh'}{\Gamma \vdash \text{indexLeft } e :: sh}$	$\frac{\Gamma \vdash e :: sh \# sh' \quad \text{IsShape } sh \quad \text{IsShape } sh'}{\Gamma \vdash \text{indexRight } e :: sh'}$	$\frac{\Gamma \vdash e_0 :: sh \quad \Gamma \vdash e_1 :: sh' \quad \text{IsShape } sh \quad \text{IsShape } sh'}{\Gamma \vdash e_0 \# e_1 :: sh \# sh'}$
$\frac{\Gamma \vdash e :: sh \# Z}{\Gamma \vdash e :: sh}$	$\frac{\Gamma \vdash e :: sh \# (sh'.\text{Int})}{\Gamma \vdash e :: (sh \# sh').\text{Int}}$	$\frac{\Gamma \vdash e :: (sh \# sh') \# sh''}{\Gamma \vdash e :: sh \# (sh' \# sh'')}$
$\frac{p \in \{(+), (*), (-)\} \quad \Gamma \vdash e_0 :: \text{Int} \quad \Gamma \vdash e_1 :: \text{Int}}{\Gamma \vdash p e_0 e_1 :: \text{Int}}$	$\frac{}{\alpha, \Gamma \vdash v_0 :: \alpha}$	$\frac{\Gamma \vdash v_n :: \alpha}{\beta, \Gamma \vdash v_{n+1} :: \alpha}$
$\frac{\Gamma \vdash e :: \text{Array } sh \alpha \quad \Gamma \vdash ix :: sh}{\Gamma \vdash e ! ix :: \alpha}$	$\frac{\Gamma \vdash e_1 :: \alpha \quad \alpha, \Gamma \vdash e_2 :: \beta}{\Gamma \vdash \text{let } v_0 = e_1 \text{ in } e_2 :: \beta}$	
$\frac{\Gamma \vdash e :: (\alpha_1, \alpha_2, \dots, \alpha_n) \quad 1 \leq i \leq n}{\Gamma \vdash \text{prj } i e :: \alpha_i}$	$\frac{\Gamma \vdash e :: \text{Array } sh \alpha}{\Gamma \vdash \text{extent } e :: sh}$	$\frac{\Gamma \vdash e_1 :: sh \quad sh, \Gamma \vdash e_2 :: \alpha \quad \text{IsShape } sh}{\Gamma \vdash \text{generate } e_1 (\lambda v_0. e_2) :: \text{Array } sh \alpha}$
$\frac{\alpha, \alpha, \Gamma \vdash e_1 :: \alpha \quad \Gamma \vdash e_2 :: \alpha \quad \Gamma \vdash e_3 :: \text{Array } (sh.\text{Int}) \alpha}{\Gamma \vdash \text{fold } (\lambda v_1 v_0. e_1) e_2 e_3 :: \text{Array } sh \alpha}$		

Figure 4.3: The typing rules for ACC_{NDP}

$$\begin{array}{c}
\frac{\Gamma \vdash e_1 :: \text{Segments } sh \quad sh, \Gamma \vdash e_2 :: \alpha \quad \text{IsShape } sh}{\Gamma \vdash \text{generate}_{seg} e_1 (\lambda v_0. e_2) :: \text{Vector } \alpha} \\
\\
\frac{\alpha, \alpha, \Gamma \vdash e_1 :: \alpha \quad \Gamma \vdash e_2 :: \alpha \quad \Gamma \vdash e_3 :: \text{Segments } (sh.\text{Int}) \quad \Gamma \vdash e_4 :: \text{Vector } \alpha}{\Gamma \vdash \text{fold}_{seg} (\lambda v_1 v_0. e_1) e_2 e_3 e_4 :: (\text{Segments } sh, \text{Vector } \alpha)} \\
\\
\frac{\text{IsShape } sh' \quad \Gamma \vdash e :: \text{Array } sh \ sh'}{\Gamma \vdash \text{segmented } e :: \text{Segments } (sh \# sh')} \quad \frac{\Gamma \vdash e_1 :: \text{Segments } sh \quad \Gamma \vdash e_2 :: \text{Segments } sh'}{\Gamma \vdash \text{cross } e_1 \ e_2 :: \text{Segments } (sh \# sh')} \\
\\
\frac{\Gamma \vdash e_1 :: \text{Segments } sh \quad \Gamma \vdash e_2 :: sh}{\Gamma \vdash e_1 \# e_2 :: \text{Int}} \quad \frac{\Gamma \vdash e :: \text{Segments } (sh \# sh')}{\Gamma \vdash \text{expand } e :: (\text{Segments } sh, \text{Vector } sh')} \\
\\
\frac{\Gamma \vdash e_1 :: \text{Segments } sh \quad \Gamma \vdash e_2 :: \text{Segments } sh}{\Gamma \vdash \text{intersects } e_1 \ e_2 :: \text{Segments } sh} \\
\\
\frac{\Gamma \vdash e :: \text{Segments } (sh \# sh')}{\Gamma \vdash \text{lefts } e :: \text{Segments } sh} \quad \frac{\Gamma \vdash e :: \text{Segments } (sh \# sh')}{\Gamma \vdash \text{rights } e :: \text{Segments } sh'}
\end{array}$$

Figure 4.4: The typing rules for segmented operations in ACC_{NDP}

the possibility of race conditions and impurity.

- **Non-array values:** Not every value in ACC_{NDP} is an array. This is in contrast to the APL family of languages[28] but corresponds with Accelerate.
- **Strictness:** As in Accelerate, let bindings and array elements are strict. Despite being embedded in the non-strict meta-language of Haskell, Accelerate is itself wholly strict. We are not concerned with lazy arrays.

The grammar of the language is listed in Figure 4.1 and the typing rules in 4.3. What follows is a description of each of the constructs of the language in detail. We also define some of the conventions we will use throughout this chapter.

4.1.1 Literals

The supported literals are integer literals and the Z shape descriptor. The latter is explained in more detail below. It would be trivial to support additional forms of numeric literal (e.g. floating point literals), but only integer literals are necessary to define the flattening transformation and to give illustrative examples. While not practical for implementation, we also treat all integers as unbounded. Once again, this simplifies explanation without sacrificing correspondence to the implementation.

4.1.2 Variables and let bindings

TODO FINAL: *This explanation of debruijn indices is not good. It needs to be refined. Also, syntax? It has been moved to background, so may need to be removed entirely?*

We use DeBruijn indices [21] to represent variables. This is a common technique when representing ASTs to avoid issues of alpha-equivalence and fresh name generation. Briefly, they work by using the natural numbers in place of variables. The number indicates how many levels of binding outside the super-term the variable is bound. For example

```
let v_0 = A in
  let v_0 = B in
    (v_0 + v_1) / 2
```

Here, we're calculating the average of A and B. Note that while v_0 is bound at the outermost let binding, inside the body of the inner binding it is referred to via v_1 .

Note that there is no general lambda term in the definition of ACC_{NDP} and hence no ability to abstract over functions. While there exists the second order operations **generate** and **fold** that appear to take functions as arguments, this is just syntax. The only places lambdas occur is in the arguments to these operations. This first order nature may appear a burdensome restriction for the language, but as is indicated by NESL[5], even first-order array languages can express useful examples. Also by embedding a first-order language in a higher order meta-language, we can recover a lot of the expressiveness. Accelerate itself is a good example of this.

4.1.3 Tuples

Tuples of arbitrary arity are supported in ACC_{NDP} . They are constructed with the familiar notation of comma separated values wrapped in parens. To be precise about our terminology, we will use the Accelerate convention of referring to tuples as scalar if they contain other scalar values. For example,

```
(Int,Int)
```

is still scalar despite it being a product of primitive `Ints`. Similarly,

```
Array (Z:.Int) (Int,Int)
```

is an array of scalar elements. We consider a type scalar if it is a tuple of scalars or is a primitive (in this case just `Int` or `Z`). Formally, this is the type level predicate `IsScalar` as defined in Figure 4.2.

Tuple projection works by supplying a numeric literal representing which element of the tuple to project out. For example:

```
prj 2 (3,4,5)
```

yields 4, the second element. Thus, we have

```
fst :: (a,b) → a
fst = prj 1
```

and

```
snd :: (a,b) → b
snd = prj 2
```

4.1.4 Scalar Operations

The scalar operations of the language are divided into the binary and unary operations. We treat all operations as prefix when traversing the AST, but when writing programs, we will often use infix notation for clarity. The semantics of the arithmetic operations are much as you would expect. We do not specify behaviour in edge cases, like division by zero, leaving that to be a property of the implementation. By assuming all integers are unbounded we can also ignore issues of overflow. All non-arithmetic operations are over array extents. We describe those below.

4.1.5 Shapes and extents

We treat array extents (the value level representations of shapes) as heterogeneous snoc-lists. The primary way in which they are constructed is via `Z` and `(:.)`. Like Accelerate, we use `Z` to represent the zero dimension. An array of extent `Z` has a single element. We refer to such arrays as unit arrays. The snoc-operator `(:.)` extends the dimensionality of an extent by 1.

We capture shapes with the type level predicate `IsShape`. In Accelerate we achieve the same via a type class, and we can view the shape predicate in the same way, just lacking a dictionary and extensibility. As such, we will often write functions in this form.

```
foo :: IsShape sh ⇒ ... sh ...
```

Essentially, we steal the syntax for typeclasses from Haskell for ease of explanation.

The innermost dimension of an array is at the head of its shape descriptor, and therefore on the right end, of the snoc-list. For example the extent

```
Z :: 3 :: 4 :: 2
```

has an inner dimension of size 2 and an outer dimension of size 3. While for the most part, the ordering of the array is orthogonal to what is described in the chapter, we will be consistent with Accelerate and opt for a row-major order. Therefore, an expression of the form

```
generate (Z :: 3 :: 2) (λv_0. v_0)
```

¹While `ACCNDP` does not explicitly support array literals we use them in this chapter to demonstrate what the actual values within arrays are. As seen here, we use curly braces to denote them.

results in an array with this layout¹

```
{ { Z:.0:.0, Z:.0:.1 },
  { Z:.1:.0, Z:.1:.1 },
  { Z:.2:.0, Z:.2:.1 } }
```

We support the following shape operations to construct and extract dimensions from shapes.

- `indexHead :: IsShape sh \Rightarrow (sh:.Int) \rightarrow Int` gives the innermost dimension of the shape.
- `indexTail :: IsShape sh \Rightarrow (sh:.Int) \rightarrow sh` strips the innermost dimension from a shape descriptor.
- `(:.) :: IsShape sh \Rightarrow sh \rightarrow Int \rightarrow sh:.Int` extends the shape by adding a new inner dimension.
- `(#) :: (IsShape sh, IsShape sh') \Rightarrow sh \rightarrow sh' \rightarrow sh#sh'` concatenates two shape descriptors.
- `indexLeft :: (IsShape sh, IsShape sh') \Rightarrow sh#sh' \rightarrow sh` gives the left shape of a concatenation of shapes. This depends on explicit type information to decide how to split its argument. It is not necessary to write programs but is produced by flattening.
- `indexRight :: (IsShape sh, IsShape sh') \Rightarrow sh#sh' \rightarrow sh'` gives the right shape of a concatenation of shapes. This depends on explicit type information to decide how to split its argument. It is not necessary to write programs but is produced by flattening.

These operations on array extents are necessary to implement our flattening transform fully. They allow us to combine extents arbitrarily and extract their components. The first three of these, are simple polymorphic functions over shapes. The remaining three, however, introduce some extra complexity in the form of the `(#)` type level shape concatenation operator. Unlike `(:.)`, we do not treat `(#)` as a type constructor, but rather as a type function that allows us to combine shape descriptors. For example,

```
Z:.Int:.Int # Z:.Int ~ Z:.Int:.Int:.Int:.Int
```

Rather than trying to add support for functions at the type level (and all the complexity that involves) to our language, we simply add typing rules specifically for this one function. Naturally, in the case of `indexLeft` and `indexRight`, we encounter the problem that they are ambiguous without explicit type information, making type inference undecidable in their presence. Fortunately, in our case, these two operations do not occur in source programs, only the output of our flattening transformation described below. As such, the ambiguity is resolved by our AST being typed.

It is also important for the purposes of flattening that we recognise $(\#)$ is associative. We go so far as to encode this fact into the type system by having rules that allow us to treat terms of type

```
(sh#sh')#sh''
```

also having type

```
sh#(sh'#sh'')
```

In more advanced type systems (such as Haskell's) it is possible to capture such properties with value-level witnesses and so it is not necessary for such things to be encoded in the type system. For our purposes, however, it is simpler to include the extra rules.

$(\#)$ is also commutative, but this is not a fact that is relied upon at all, so we do not encode it.

In addition to the above operations we also support

```
intersect :: Shape sh => sh -> sh -> sh
```

This is a useful operation for combining shapes and implementing operations like `zipWith`. We get the largest shape that *fits* inside both given shapes with `intersect`. For example:

```
intersect (Z:.1:.2) (Z:.2:.1) = Z:.1:.1
```

4.1.6 Array indexing

Arrays are indexed by the bang $(!)$ operator. As per the typing rule, the shape of the array and the type of the index must match. For example, if `a` is a matrix,

```
a ! Z:.2:.1
```

will yield the value in the 2nd row and 1st column.

We do not specify what happens in the event an array is accessed outside its extent. In this regard, indexing is a partial operation. Moreover, we make no attempt at preserving this undefined behaviour as part of our flattening transformation – i.e. a program that accesses an array out of bounds may, after flattening, no longer do so. We assume all programs given to the transform are safe in this regard.

4.1.7 Array extents

The function `extent :: Array sh e -> sh` returns the extent of an array. This is equivalent to `shape` in Accelerate, renamed to avoid ambiguity between type level and value level information.

4.1.8 Array generation

The built-in operation `generate` can be used to create arrays from a generation function.

```
generate :: IsShape sh => sh -> (sh -> e) -> Array sh e
```

An expression of the form `generate ix (\v0. e)` constructs an array of extent `ix` with every element computed from `(\v0. e)`. For example,

```
evens :: Vector Int
evens = generate (Z:.10) (\ix -> indexHead ix * 2)
```

Here we are assuming we can write terms using named variables instead of DeBruijn indices and that we have a notion of higher-order functions. For the purposes of explanation, we will assume we can write terms in a higher-order Haskell-like meta-language that we can use to construct ACC_{NDP} terms. By doing this, we can write additional higher order array combinators that will be useful not only to write more interesting programs but also, as we will show, implementing many auxiliary functions for flattening. This is one of a small number of liberties we take to make the description of the flattening transform simpler.

Going back to `generate`, combining it with indexing gives us the more traditional array combinator:

```
map :: IsShape sh => (a -> b) -> Array sh a -> Array sh b
map f arr = generate (extent arr) (\ix -> f (arr ! ix))
```

Similarly, we can also combine arrays by defining

```
zipWith :: IsShape sh => (a -> b -> c) -> Array sh a -> Array sh b -> Array sh c
zipWith f arr1 arr2 = generate (intersect (extent arr1) (extent arr2))
  (\ix -> f (arr1 ! ix) (arr2 ! ix))
```

4.1.9 Array reduction

To perform reductions, we have `fold`, which is parameterised by a scalar function and a scalar starting value. Like the `fold` in Accelerate, it is shape polymorphic [29] and assumes associativity of the scalar function.

```
fold :: (a -> a -> a) -> a -> Array (sh:.Int) a -> Array sh a
```

Also, like in Accelerate, this is a multi-dimensional fold. It works over arrays of arbitrary rank by reducing only along the inner dimension.

4.2 Segmented arrays

In addition to the core constructs of ACC_{NDP} , we add constructs for operating with *segment descriptors*. The rules for these are given in 4.4. We treat **Segments** as an abstract data type,

defining it only by the operations on it. To understand what it represents, we need to be more specific about what exactly the shape of an array is. Up until this point, we have said that the shape of an array is its dimensionality. However, another interpretation is that it is a mapping from a higher dimensional index into a position in a flat vector.

When we have, an array `a` such that

```
a :: Array sh e
```

then `a` can be indexed with values of type `sh`. We know that in memory, the elements of `a` are stored in a contiguous block, but we want to abstract away from that. From an index of type `sh` into the array, we can determine the corresponding (linear) index into the contiguous block by using the extent of the array. For example, if `a` has the shape

```
Z:.5:.4:.3
```

and an index of

```
Z:.2:.1:.0
```

we know that the value of the array at this index is at position

TODO FINAL: *One reviewer didn't understand what was going on here. This needs to be made clearer. Maybe a diagram?*

```
0 + 3(1 + 4(2 + 5(0))) = 27
```

in the contiguous block. We're calculating the linear index via the formula.

```
linearIndex Z Z = 0
linearIndex (sh:.n) (ix:.i) = i + n*(linearIndex sh ix)
```

where the first argument is the shape of the array and the second is the index into it. Hence, we can view a shape descriptor as specifying a set of possible indices and a mapping from them to linear indices.

Given some shape, `sh`, we can enumerate all its indices with

```
indices :: IsShape sh => sh -> Array sh sh
indices sh = generate sh (\v_0. v_0)
```

For example, suppose

```
sh = Z:.3:.4
```

then its indices are

```
indices (Z:.3:.4) = { { Z:.0:.0, Z:.0:.1, Z:.0:.2, Z:.0:.3 }
                      , { Z:.1:.0, Z:.1:.1, Z:.1:.2, Z:.1:.3 }
                      , { Z:.2:.0, Z:.2:.1, Z:.2:.2, Z:.2:.3 } }
```

If we are viewing shapes in this way, then we can form an intuition about segment descriptors by viewing them in a similar fashion. They also represent a set of indices and a mapping

to linear indices but are less restricted. A shape descriptor specifies only a single size for each dimension, so can only represent regular sets of indices. However, with segment descriptors, we represent sets of indices that are more general. For example, suppose we want to represent an irregular 2D array with 4 rows and that the first row has 4 elements, the second has 3, the third has 0, and the fourth has 1. We would need a value of type

```
unevenRows :: Segments (Z:.Int:.Int)
```

Before approaching how to construct an expression of the right type, let's first address how it is used.

We can enumerate its indices with

```
indices_seg :: Segments sh → Vector sh
indices_seg segs = generate_seg segs (λv₀. v₀)
```

so therefore its indices are

```
indices_seg unevenRows = { Z:.0:.0, Z:.0:.1, Z:.0:.2, Z:.0:.3
                          , Z:.1:.0, Z:.1:.1, Z:.1:.2
                          , Z:.3:.0 }
```

Here, we're using `generate_seg`. It is the segmented equivalent of `generate`.

```
generate_seg :: IsShape sh ⇒ Segments sh → (sh → e) → Vector e
```

It constructs a flat vector given some segments descriptors and a generation function. The `Vector` type is, like in Accelerate, just a synonym for a 1-dimensional array.

```
type Vector = Array (Z:.Int)
```

That still leaves us with the problem of how to construct segment descriptors. Unlike with shape descriptors, we cannot just represent them with a snoc-list of integer dimensions. Instead, we use `segmented`. It takes an array of some shape `sh` containing values of `sh'` and gives back `Segments (sh#sh')`. For our example

```
unevenRows = segmented { Z:.4, Z:.3, Z:.0, Z:.1 }
```

We say that this set of indices is irregular.

We can also construct segment descriptors with `cross`.

```
cross :: Segments sh → Segments sh' → Segments (sh#sh')
```

It is equivalent to `(#)`, but operating over segments. That is to say, if

```
ix ∈ indices_seg segs
```

and

```
ix' ∈ indices_seg segs'
```

then

```
ix # ix' ∈ indicesseg (cross segs segs')
```

There are a number of other operations for working with segment descriptors. We will introduce them where required in the next section.

4.3 Regularity-identifying flattening

EXAMINER: *Section 4.3 mentions that it is crucial to preserve regularity since handling irregular structures might cause a significant runtime overhead. However, the experimental results presented in the thesis seems to imply that there is still a large overhead (see comments for Chapter 7). Does the flattening transformation offers any guarantees that is always possible to get rid of the irregular data representation when possible? If this is the case, would it be possible to prove this (either formally, or experimentally?). This needs to be discussed in this chapter. As a suggestion, maybe this chapter could take one of the use-cases (from the evaluation) and show in details how the flatten representation is derived and how it compares with the same program written in Vanilla Accelerate. This would perhaps also help understand where the performance gap observed in the current evaluation comes from and how these could be addressed. It would also be nice to have an example starting from the high-level all the way down to CUDA for instance (in this chapter or somewhere else in the thesis).*

With ACC_{NDP} defined, we can now construct a flattening transform. This transform will use ACC_{NDP} as its source language (minus the segmented operations) and also as the target language (including the segmented operations).

We will describe flattening first by example before specifying it formally. Consider the simple function of summing up elements of a vector.

```
sum :: Vector Int → Int
sum xs = the (fold (+) 0 xs)
```

Here, we use the auxiliary function `the`, borrowed from Accelerate, that indexes a zero dimension array, extracting the single element it contains.

```
the :: Array Z e → e
the arr = arr ! Z
```

Using nesting, we can use this to write a function that sums up each vector contained in a larger vector.

```
sums :: Vector (Vector Int) → Vector Int
sums xss = map sum xss
```

If the ultimate goal is to remove any nesting from a program, what we require is a version of this function that does not depend on nesting. How can we do that? We will look at two possible situations. Suppose `xss` is regular, that is, all sub-vectors are of the same length, then we can treat our vector of vectors as a 2D array. This gives an alternative version of

`sums`.

```
sumsR :: Array DIM2 Int → Vector Int
sumsR xss = fold (+) 0 xss
```

What we have is essentially the same as `sum`, but with the fold now occurring over the inner dimension of the 2D array and no longer needing to call `the` to remove the nesting.

If, however, we do not know that `xss` is regular, we would need to write a more general version of `sums`.

```
sumsIr :: (Segments DIM2, Vector Int) → Vector Int
sumsIr (segs, vs) = snd (foldseg (+) 0 segs vs)
```

Here, we’re representing this collection of vectors in a *segmented* form. Like the segmented arrays described in Section 2.1, it is split into a *flat data vector* (containing all the elements of all vectors) together with a set of segment descriptors describing how to index the flat vector.

Unlike our regular representation, we cannot just use the same version of the fold operation to work over a segmented representation. Instead, we use the segmented variant of `fold`. This has the type:

```
foldseg :: (a → a → a)
          → a
          → Segments (sh :: Int)
          → Vector a
          → (Segments sh, Vector a)
```

We will cover in more detail the implementation of this operation in chapter 5. Suffice to say, however, any implementation of it is going to be significantly more expensive than that of `fold`. Moreover, whatever the concrete representation of `Segments` is, maintaining and passing them around is an additional overhead. Clearly, we want to incur these costs only when necessary. The use of the segmented fold should be avoided wherever possible. Putting it another way, we want to make sure that when it is possible to do so, the *regular* fold is used instead.

This is not just true for this particular example or just for `fold`. In general, computations on regular, multi-dimensional arrays are much more efficient than performing the same computations on nested, potentially irregular structures. In other words, the mere ability to handle irregular structures presents a significant runtime overhead, even if it is never used. Hence, it is crucial for our variant of flattening to preserve regularity. Computations which are regular, even when written using nesting, should remain regular when flattened.

So, to recap, what we require is a method by which to take ACC_{NDP} terms and convert them to terms that perform the same function over the most appropriate representations of its arrays. We do this by *lifting* all open terms into terms that operate in a lifted context. We will describe this in the next section. However, before that, it is important to note that the regular and irregular representations are not the only ones we care about. The *avoided*

representation is when we do not change the representation of arrays or scalar values at all.

As a simple example, consider this scalar function:

```
average :: Int → Int → Int
average x y = (x + y) / 2
```

Flattening of this code takes each subcomputation from a scalar function to one operating over higher dimensional arrays, the actual shape depending on the context.

```
averageR :: Array sh Int → Array sh Int → Array sh Int
averageR xs ys = zipWith (/) (zipWith (+) xs ys) (replicate (extent xs) 2)
```

This code features excessive array traversals and many superfluous intermediate structures. In this simple example, fusion optimisations can improve the code, but more generally we will arrive at better code when flattening directly *avoids* flattening purely scalar or non-nested subexpression, and generates the following code instead:

```
averageavoid :: Array sh Int → Array sh Int → Array sh Int
averageavoid xs ys = zipWith (\x y → (x + y) / 2) xs ys
```

The concrete flattening transformation presented in the rest of this section avoids the flattening of subexpressions not used in a nested context or containing nesting and preserves regularity, where flattening is over a regular domain.

4.3.1 Lifted type relation

While type information is not necessary to perform a flattening transformation, if we wish to describe it in such a way that it is type *preserving* we have to be mindful of what is happening at the type level. Here, we discuss the type transformations \mathcal{N} and \mathcal{F} . In the next section, we will show how the term transformation \mathcal{L} matches up to \mathcal{N} and \mathcal{F} .

Previous work on flattening [11] describes the type level component as a simple function. However, in this work, some types have more than one possible flattened representation. Having both regular and irregular flattening contexts give us a choice between different representations, the exact type of a term after flattening will not be known until after that term has been traversed. For this reason, we will represent our type level transformation relationally.

To aid in explanation, we will use something similar to Haskell’s generalised algebraic data types (GADTs) [42] both to describe the type level relations and also to track information in the term level transformation. This advantage of this it allows us to bring type equivalencies into the environment by simply matching on constructors. It has the added benefit of more closely matching the concrete implementation in Accelerate.

Normalisation

The first type transformation, $\mathcal{N} \vdash \mathbf{t} \vdash_{norm}$, computes an array normal form \mathbf{t}_{norm} of an Accelerate type \mathbf{t} . The input is any (possibly nested) type, and the output is a tuple (can be a

nested tuple) of arrays with no array nesting.

```
data  $\mathcal{N}$  t t' where
  Scalar :: IsScalar e
         =>  $\mathcal{N}$  e (Array Z e)
  Nest   ::  $\mathcal{N}$  e (Array sh1 e1, Array sh2 e2, ... , Array shn en)
         →  $\mathcal{N}$  (Array sh e) (Array (sh # sh1) e1, Array (sh # sh2) e2, ... , Array (sh # shn) en)
  Tuple $\mathcal{N}$  :: ( $\mathcal{N}$  t1 t1',  $\mathcal{N}$  t2 t2', ... ,  $\mathcal{N}$  tn tn')
         →  $\mathcal{N}$  (t1, t2, ... , tn) (t1', t2', ... , tn')
```

Before we begin describing this fundamentally simple relation, it is first necessary to discuss an unusual feature of the meta language we use for describing the transform. For the most part, one can view the meta-language as being Haskell. This matches the implementation in Accelerate and leads to an understandable definition, for a reader already familiar with Haskell. However, to further aid explanation, we allow for the meta-language to generalise over tuples. We use the (...) notation to capture the fact that something can take an arbitrary n-tuple. Suppose we have a function like

```
foo :: (Array sh1 e1, Array sh2 e2, ... Array shn en) → r
```

it is able to take any tuple value as argument; e.g. both (Array sh₁ e₁, Array sh₂ e₂) and (Array sh₁ e₁, Array sh₂ e₂, Array sh₃ e₃) would be considered valid argument types. In addition, we treat one-tuples as being equivalent to a single value. So Array sh e would be a valid argument as well. While it is possible to write tuple-arity polymorphic functions in Haskell, with representation types or generic programming libraries, doing so require significant extra type-level programming that tends to permeate throughout the code, even in the parts where it is not relevant. We opt not to do this in this chapter as what we describe already has a complex type level component and it would only obscure the core ideas of our work. By assuming our meta language lets us generalise over tuples and single values we sidestep the issue, leaving it to be addressed in Chapter 5.

Going back to the relation, we will first look at the **Scalar** constructor. It specifies that scalars of type **e** are wrapped in a zero dimension array of type **Array Z e**, keeping in mind that our definition of scalar values includes tuples of scalars.

The **Nest** constructor captures how a regular *n*-dimensional array of regular *m*-dimensional arrays becomes an *n + m*-dimensional array. Alternatively, to put it in terms of shapes, an array of shape **sh** containing arrays of shape **sh'** becomes an array of shape **sh#sh'**. For example, vectors of vectors of **Ints**, after expanding synonyms, has the type:

```
Array (Z :: Int) (Array (Z :: Int) Int)
```

We normalise that type to two-dimensional arrays of type **Array DIM2 Int**, as witnessed by:

```
Nest (Nest Scalar) ::
   $\mathcal{N}$  (Array (Z :: Int) (Array (Z :: Int) Int)) (Array (Z :: Int :: Int) Int)
```

The complexity comes when dealing with tuples. For example:

```
Array (Z :: Int) (Array (Z :: Int) Int, Array (Z :: Int) Int)
```

This is a vector of pairs of vectors witnessed by:

```
Nest (TupleN (Nest Scalar), (Nest Scalar)) ::
  N (Array (Z :: Int) (Array (Z :: Int) Int, Array (Z :: Int) Int))
  (Array (Z :: Int :: Int) Int, Array (Z :: Int :: Int) Int)
```

Because there is an array of non-scalar pairs, this has to be expanded out into a pair of arrays.

This normalisation process gives us a way to remove the nesting from data structures without specifying precisely what representation is necessary. Alone it is not sufficient in the case of irregular nesting. It is also actually a function even though we choose to handle it representationally. The need for that becomes apparent with the subsequent type level transformation.

Flattening

The second type relation, $\mathcal{F} \text{ sh}_{\mathcal{L}} \mathbf{t} \mathbf{t}^{flat}$, takes an Accelerate type to its flattened form \mathbf{t}^{flat} by first normalising \mathbf{t} by way of $\mathcal{N} \mathbf{t} \mathbf{t}^{norm}$, and then extending the dimensionality of the normalised \mathbf{t}^{norm} in either a regular or irregular form.

The first argument to \mathcal{F} is what we refer to as the lifting shape. It represents to what dimension the type has been lifted. To put it another way, a term of type

```
 $\mathcal{F} \text{ sh}_{\mathcal{L}} \mathbf{t} \mathbf{t}^{flat}$ 
```

witnesses that \mathbf{t}^{flat} can also be represented as an array of shape $\text{sh}_{\mathcal{L}}$ containing values of type \mathbf{t} . The difference between the two representations being that \mathbf{t}^{flat} does not contain any non-scalar nesting. It is important to note that the two types are not actually isomorphic. While it is possible to convert from values of \mathbf{t}^{flat} to $\text{Array sh}_{\mathcal{L}} \mathbf{t}$, the reverse direction is not always possible. That is because \mathcal{F} also captures information about regularity and avoidance.

This desire to capture the regularity makes this transform ambiguous. This is why it is necessary to treat the whole transformation as a relation. If the enclosing data-parallel context is regular, we simply increase the dimensionality of all arrays in \mathbf{t}^{norm} by $\text{sh}_{\mathcal{L}}$. However, if the context is irregular, we need to introduce segment descriptors, as discussed in the previous section. These two cases are covered by the alternatives **Regular** and **Irregular** below:

```
data F shL t tflat where
  AvoidS    :: IsScalar t
             => F shL t t
             -- avoid flattening
  AvoidA    :: IsScalar e
             => F shL (Array sh e) (Array sh e)
  Regular   :: N t (Array sh1 e1, Array sh2 e2, ... , Array shn en)
             -- regular context
             -> F shL t ( Array (shL+sh1) e1
                           , Array (shL+sh2) e2
                           , ...
                           , Array (shL+shn) en)
```

```

Irregular ::  $\mathcal{N}$  t (Array sh1 e1, Array sh2 e2, ... , Array shn en) -- irregular context
  →  $\mathcal{F}$  sh $\mathcal{L}$  t ( (Segments (sh $\mathcal{L}$ #sh1), Vector e1)
    , (Segments (sh $\mathcal{L}$ #sh2), Vector e2)
    , ...
    , (Segments (sh $\mathcal{L}$ #shn), Vector en))
Tuple $\mathcal{F}$  :: ( $\mathcal{F}$  sh $\mathcal{L}$  t1 t1flat,  $\mathcal{F}$  sh $\mathcal{L}$  t2 t2flat, ... ,  $\mathcal{F}$  sh $\mathcal{L}$  tn tnflat)
  →  $\mathcal{F}$  sh $\mathcal{L}$  (t1, t2, ... , tn) (t1flat, t2flat, ... , tnflat)

```

The `Tuple \mathcal{F}` constructor may at first glance seem unnecessary, but it allows for different components of tuples to be flattened independently. This is crucial as one component might be used in a regular context, while another in an irregular context; similarly one might use avoidance, while the others don't, et cetera. One example of this is:

```

foo :: Vector Int → (Vector (Vector Int), Vector (Vector Int))
foo xs = ( generate (Z:.10) (λix → map (+ indexHead ix) xs)
    , map (λx → generate (Z:.x) (_ → x)) xs)

```

Here we are constructing a pair of nested vectors from a single non-nested vector of `Int`s. The first component of the pair is a vector of length 10 with each element the same length as `xs`. Because of this, it is regular and thus can use a regular representation after flattening. The second component is of the same length as `xs`, but with each sub-vector of length equal to an element of `xs`. Just looking at this function in isolation, we have no way of knowing the contents of `xs` ahead of time, so we have to treat the second component as irregular.

To further underpin this, consider how tuples are often used in functional programs. A programmer would expect

```

bar = let a = A
      b = B
      in C

```

to be the same as

```

bar' = let (a,b) = (A, B)
        in C

```

Assuming that `B` does not depend on `A` (or vice versa), it would be undesirable for the simple introduction of a tuple grouping related bindings to introduce substantial performance changes. Similarly, *currying* or *uncurrying* functions should not alter their performance characteristics.

Flattening avoidance is covered by the `Avoid \mathcal{S}` and `Avoid \mathcal{A}` alternatives, where we keep the type the same. We do not need to normalise as the `e` in `Avoid \mathcal{S} :: \mathcal{F} sh \mathcal{L} e e` and `Avoid \mathcal{A} :: \mathcal{F} sh \mathcal{L} (Array sh e) (Array sh e)` is guaranteed to be scalar by the `IsScalar` constraint.

The \mathcal{F} type is not a singleton. That is to say, there are multiple witness for some type relations. For example,

```

 $\mathcal{F}$  sh $\mathcal{L}$  (Int, Int) (Int, Int)

```

has both

`AvoidS`

and

`TupleF (AvoidS, AvoidS)`

as inhabitants. This can be a problem when trying to resolve whether a particular type is *completely* avoided: when flattening was avoidable for all components. We resolve this by introducing a *smart* constructor:

$$\begin{aligned} \text{tuple}_F &:: (F \text{ sh}_{\mathcal{L}} \mathbf{t}_1 \mathbf{t}_1^{flat}, F \text{ sh}_{\mathcal{L}} \mathbf{t}_2 \mathbf{t}_2^{flat}, \dots, F \text{ sh}_{\mathcal{L}} \mathbf{t}_n \mathbf{t}_n^{flat}) \\ &\rightarrow F \text{ sh}_{\mathcal{L}} (\mathbf{t}_1, \mathbf{t}_2, \dots, \mathbf{t}_n) (\mathbf{t}_1^{flat}, \mathbf{t}_2^{flat}, \dots, \mathbf{t}_n^{flat}) \end{aligned}$$

This function will check its arguments to see if they are all `AvoidS` and return `AvoidS` if they are. This excludes the second inhabitant above and enables a simple check for avoidance of scalars.

4.3.2 The lifting transformation

The lifting transform \mathcal{L} takes a term in our language and yields a term in the same language of a \mathcal{F} -related type that contains no nested parallelism nor nested arrays. Its complete type is:

$$\mathcal{L}[\cdot] :: \text{Expr } \Gamma \mathbf{t} \rightarrow \text{Env sh}_{\mathcal{L}} \Gamma \Gamma^{flat} \rightarrow (\exists \mathbf{t}^{flat} . (F \text{ sh}_{\mathcal{L}} \mathbf{t} \mathbf{t}^{flat}, \text{Expr } \Gamma^{flat} \mathbf{t}^{flat}))$$

This transformation has many components, so we will look at each one in turn. The first argument of type `Expr Γ \mathbf{t}` is the typed abstract syntax (AST) of the core language term that is to be flattened, where Γ is a type-level list of the free variables in the term and \mathbf{t} is its type.

The second argument, of type `Env shℒ Γ Γ^{flat}` , is an environment that relates the types of the free variables Γ to their flattened form Γ^{flat} . Like the \mathcal{F} relation, it is also parameterised by the lifting shape.

Finally, the result combines the witness for the flattened result type \mathbf{t}^{flat} with the lifted term `Expr Γ^{flat} \mathbf{t}^{flat}` , whose type parameters have been flattened to match, establishing type-preservation of the transformation.

The structure of the environment `Env shℒ Γ Γ^{flat}` is somewhat more involved than usual, due to special requirements during flattening. Specifically, we need to handle access to variables brought into scope with a `generate` outside the most immediate one. To give an example of this:

```
mults :: Expr [] (Vector (Vector Int))
mults = generate (Z:10) (\i → generate i (\j → indexHead i * indexHead j))
```

Here we are constructing the lower triangle of a multiplication table. If we want to flatten

this, we have to lift the inner **generate** from 1D to 2D, but not lift the outer **generate** at all. However, in the body of the inner **generate**, we are referring to a variable bound by the outer **generate**. This presents a problem. The variable was bound at a point which is not being lifted to the same dimension. We partially resolve this by tracking lifting shape in the environment, but this does not take into account the actual *extent* of the outer array. Because of this, we track both regular (the $(:R)$ alternative) and irregular contexts (the $(:Ir)$ alternative). When the lifting transform encounters a **generate** it must extend the lifting shape by the shape of the generated array. The generation function can then be lifted under this larger context.

```

data Env shL  $\Gamma$   $\Gamma^{flat}$  where
  []    :: Env Z [] []
  (:)   ::  $\mathcal{F}$  shL t tflat          -- standard free variable
        → Env shL  $\Gamma$   $\Gamma^{flat}$ 
        → Env shL (t :  $\Gamma$ ) (tflat :  $\Gamma^{flat}$ )
  (:R)  :: Shape sh                  -- regular flattening context
        ⇒ Expr  $\Gamma^{flat}$  sh
        → Env shL  $\Gamma$   $\Gamma^{flat}$ 
        → Env (shL#sh)  $\Gamma$   $\Gamma^{flat}$ 
  (:Ir) :: Shape sh                  -- irregular flattening context
        → Expr  $\Gamma^{flat}$  (Segments sh)
        → Env shL  $\Gamma$   $\Gamma^{flat}$ 
        → Env (shL#sh)  $\Gamma$   $\Gamma^{flat}$ 

```

A key insight of the environment structure is that a closed term can only be lifted into Z . Intuitively, this makes sense. We only need to lift things into a higher dimension when it is used in a nested context. A closed term, by definition, is not used in any context, nested or otherwise.

The use of this environment structure can be seen in the definition of the auxiliary function **var** defined in Figure 4.5, which looks up free variables in the given environment. In addition to that conventional purpose, it also replicates the variable according to each enclosing context; i.e., while traversing the environment to look up the variable, it inserts a **replicate_R** and **replicate_{Ir}** for every $(:R)$ and $(:Ir)$ that it comes across, respectively.

The transformation rules of the flattening transformation are given in Figure 4.7. What follows is an explanation of how each language construct is flattened.

Literals

This is the simplest construct in terms of flattening. When literal is encountered, flattening returns it as it is, paired with the **Avoid_S** constructor to signal that it does not need to be lifted.

```

-- Lifting variables
var :: Env shL Γ tflat → Var Γ t → (∃ tflat . (F shL t tflat , Expr Γ tflat tflat ))
var (r : _ ) v0 = (r, v0)
var ( _ : env ) vn+1 | (r, e) ← var env vn = (r, weaken e)
var (sh :R env) v | (r, e) ← var env v = replicateR r sh e
var (segs :Ir env) v | (r, e) ← var env v = replicateIr r segs e

-- Replication under a regular context
replicateR :: F shL t tflat
→ Expr Γ sh
→ Expr Γ tflat
→ (∃ tnewflat . (F (shL#sh) t tnewflat , Expr Γ tnewflat ))
replicateR AvoidS _ t = (AvoidS, t)
replicateR AvoidA _ t = (AvoidA, t)
replicateR (Regular r) sh t = (Regular r, replicateR→R r sh t)
replicateR (Irregular r) sh t = (Irregular r, replicateIr→R r sh t)
replicateR (Tuple (r1,...)) sh t | (r'1, t1) ← replicateR r0 sh (prj 1 t)
, ...
= (Tuple (r'1, ... ), (t1, ... ))

-- Replicate a regular array under a regular context
replicateR→R :: Expr Γ sh
→ Expr Γ (Array (shL#sh1) e1, ... )
→ Expr Γ (Array (shL#sh#sh1) e1, ... )
replicateR→R sh arr = ( generate (inside sh (extent (prj 1 arr))) (λix → prj 1 arr ! outside ix)
, ... )

-- Replicate an irregular array under a regular context
replicateIr→R :: Expr Γ sh
→ Expr Γ ((Segments (shL#sh1), Vector e1), ... )
→ Expr Γ ((Segments (shL#sh#sh1), Vector e1), ... )
replicateIr→R sh arr = replicateIr→Ir (segmented (generate Z sh)) arr

-- Replication under an irregular context
replicateIr :: F shL t tflat
→ Expr Γ (Segments sh)
→ Expr Γ tflat
→ (∃ tnewflat . (F (shL#sh) t tnewflat , Expr Γ tnewflat ))
replicateIr AvoidS _ t = (AvoidS, t)
replicateIr AvoidA _ t = (AvoidA, t)
replicateIr (Regular r) sh t = (Regular r, replicateR→Ir r sh t)
replicateIr (Irregular r) sh t = (Irregular r, replicateIr→Ir r sh t)
replicateIr (Tuple (r1,...)) sh t | (r'1, t1) ← replicateIr r0 sh (prj 1 t)
, ...
= (Tuple (r'1, ... ), (t1, ... ))

-- Replicate a regular array under an irregular context
replicateR→Ir :: Expr Γ (Segments sh)
→ Expr Γ (Array (shL#sh1) e1, ... )
→ Expr Γ ((Segments (shL#sh#sh1), Vector e1), ... )
replicateR→Ir segs arr = replicateIr→Ir segs (irregular arr)

-- Replicate an irregular arrays under an irregular context
replicateIr→Ir :: Expr Γ (Segments sh)
→ Expr Γ ((Segments (shL#sh1), Vector e1), ... )
→ Expr Γ ((Segments (shL#sh#sh1), Vector e1), ... )
replicateIr→Ir segs arr = ( let segsold = prj 1 (prj 1 arr)
segsnew = insides segs segsold
valsold = prj 2 (prj 1 arr)
in (segsnew, generateseg segsnew (λix → valsold ! segsold # outside ix)
, ... )

```

Figure 4.5: Lifting variables into higher dimensions.

```

map :: (∀ Γflat . Expr Γflat a → Expr Γflat b)
      → Expr Γ (Array sh a)
      → Expr Γ (Array sh b)
map f as = generate (extent as) (λv0. f (weaken as ! v0))

mapseg :: (∀ Γflat . Expr Γflat a → Expr Γflat b)
         → Expr Γ (Segments sh, Vector b)
         → Expr Γ (Segments sh, Vector a)
mapseg f as = let segs = prj 1 as
              in ( segs, generateseg segs (λv0. f (weaken as ! Z :. weaken segs # v0)))

zipWith :: (∀ Γflat . Expr Γflat a → Expr Γflat b → Expr Γflat c)
         → Expr Γ (Array sh a)
         → Expr Γ (Array sh b)
         → Expr Γ (Array sh c)
zipWith f as bs =
  generate (intersect (extent as) (extent bs)) (λv0. f (weaken as ! v0) (weaken bs ! v0))

zipWithseg :: (∀ Γflat . Expr Γflat a → Expr Γflat b → Expr Γflat c)
            → Expr Γ (Segments sh, Vector a)
            → Expr Γ (Segments sh, Vector b)
            → Expr Γ (Segments sh, Vector c)
zipWithseg f as bs =
  let segs = intersects (prj 1 as) (prj 1 bs)
  in ( segs
    , generateseg segs (λv0. f (weaken as ! Z :. weaken segs # v0) (weaken bs ! Z :. weaken segs # v0)))

unit :: IsScalar a ⇒ Expr Γ a → Expr Γ (Array Z a)
unit a = generate Z (λ_. (weaken a))

inside :: Expr Γ sh' → Expr Γ (sh#sh'') → Expr Γ (sh#sh'#sh'')
inside sh' sh = indexLeft sh # sh' # indexRight sh

outside :: Expr Γ (sh#sh'#sh'') → Expr Γ (sh#sh'')
outside sh = indexLeft sh # indexRight sh

insides :: Expr Γ (Segments sh') → Expr Γ (sh#sh'') → Expr Γ (sh#sh'#sh'')
insides segsinner segsouter = lefts segsouter `cross` segsinner `cross` rights segsouter

-- Add a fresh variable to the environment (complete definition elided)
weaken :: Expr Γ t → Expr (a : Γ) t

```

Figure 4.6: Auxiliary operations implemented as meta-functions

```

 $\mathcal{L}[\![c]\!] \_ = (\text{Avoid}_S, c)$ 
 $\mathcal{L}[\![v]\!] \text{ env} = \text{var env } v$ 
 $\mathcal{L}[\![p_1 \ e]\!] \text{ env}$ 
  |  $(\text{Avoid}_S, e^{flat}) \leftarrow \mathcal{L}[\![e]\!] \text{ env}$ 
  =  $(\text{Avoid}_S, p_1 \ e^{flat})$ 
  |  $(\text{Regular Scalar}, e^{flat}) \leftarrow \mathcal{L}[\![e]\!] \text{ env}$ 
  =  $(\text{Regular Scalar}, \text{map } p_1 \ e^{flat})$ 
  |  $(\text{Irregular Scalar}, e^{flat}) \leftarrow \mathcal{L}[\![e]\!] \text{ env}$ 
  =  $(\text{Irregular Scalar}$ 
    ,  $(\text{prj } 1 \ e^{flat}, \text{map } p_1 \ (\text{prj } 2 \ e^{flat})))$ 
 $\mathcal{L}[\![p_2 \ e_1 \ e_2]\!] \text{ env}$ 
  |  $(r_1, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \text{ env}$ 
  ,  $(r_2, e_2^{flat}) \leftarrow \mathcal{L}[\![e_2]\!] \text{ env}$ 
  =  $\text{liftBinOp } p_2 \ r_1 \ r_2 \ e_1^{flat} \ e_2^{flat}$ 
 $\mathcal{L}[\![\text{extent } e]\!] \text{ env}$ 
  |  $(\text{Avoid}_A, e^{flat}) \leftarrow \mathcal{L}[\![e]\!] \text{ env}$ 
  =  $(\text{Avoid}_A, \text{extent } e^{flat})$ 
  |  $(\text{Regular (Nest } \_), e^{flat}) \leftarrow \mathcal{L}[\![e]\!] \text{ env}$ 
  =  $(\text{Avoid}, \text{extent}_R \ e^{flat})$ 
  |  $(\text{Irregular (Nest } \_), e^{flat}) \leftarrow \mathcal{L}[\![e]\!] \text{ env}$ 
  =  $(\text{Irregular Scalar}, \text{extent}_{I_r} \ e^{flat})$ 
 $\mathcal{L}[\![e_1 \ ! \ e_2]\!] \text{ env}$ 
  |  $(\text{Avoid}_A, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \text{ env}$ 
  ,  $(\text{Avoid}_S, e_2^{flat}) \leftarrow \mathcal{L}[\![e_2]\!] \text{ env}$ 
  =  $(\text{Avoid}_S, e_1^{flat} \ ! \ e_2^{flat})$ 
  |  $(\text{Regular (Nest } r), e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \text{ env}$ 
  ,  $(\text{Regular Scalar}, e_2^{flat}) \leftarrow \mathcal{L}_F[\![e_2]\!] \text{ env}$ 
  =  $(\text{Regular } r, e_1^{flat} \ !_R \ e_2^{flat})$ 
  |  $(\text{Irregular (Nest } r), e_1^{flat}) \leftarrow \mathcal{L}_F[\![e_1]\!] \text{ env}$ 
  ,  $(\text{Irregular Scalar}, e_2^{flat}) \leftarrow \mathcal{L}_F[\![e_2]\!] \text{ env}$ 
  =  $(\text{Irregular } r, e_1^{flat} \ !_{I_r} \ e_2^{flat} \ \text{sh}')$ 
 $\mathcal{L}[\![\text{let } v_0 = e_1 \ \text{in } e_2]\!] \text{ env}$ 
  |  $(r_1, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \text{ env}$ 
  ,  $(r_2, e_2^{flat}) \leftarrow \mathcal{L}[\![e_2]\!] \ (r_1 : \text{env})$ 
  =  $(r_2, \text{let } v_0 = e_1^{flat} \ \text{in } e_2^{flat})$ 
 $\mathcal{L}[\![e_1, \dots, e_n]\!] \text{ env}$ 
  |  $(r_0, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \text{ env}$ 
  ...
  ,  $(r_n, e_n^{flat}) \leftarrow \mathcal{L}[\![e_n]\!] \text{ env}$ 
  =  $(\text{tuple}_F \ (r_1, \dots, r_n), (e_1^{flat}, \dots, e_n^{flat}))$ 

 $\mathcal{L}[\![\text{prj } l \ e]\!] \text{ env}$ 
  |  $(\text{Avoid}_S, e^{flat}) \leftarrow \mathcal{L}[\![e]\!] \text{ env}$ 
  =  $(\text{Avoid}_S, \text{prj } l \ e^{flat})$ 
  |  $(\text{Tuple } (\dots, r_l, \dots), e^{flat}) \leftarrow \mathcal{L}[\![e]\!] \text{ env}$ 
  =  $(r_l, \text{prj } l \ e^{flat})$ 
 $\mathcal{L}[\![\text{generate } e_1 \ (\lambda v_0. e_2)]\!] \text{ env}$ 
  -- Flattening is avoidable
  |  $(\text{Avoid}_S, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \text{ env}$ 
  ,  $(\text{Avoid}_S, e_2^{flat}) \leftarrow \mathcal{L}[\![e_2]\!] \ (\text{Avoid} : \text{env})$ 
  =  $(\text{Avoid}_A, \text{generate } e_1^{flat} \ (\lambda v_0. e_2^{flat}))$ 
  -- Regular array in a regular context
  |  $\text{Left ctx} \leftarrow \text{context env}$ 
  ,  $(\text{Avoid}_S, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \text{ env}$ 
  ,  $(r, e_2^{flat}) \leftarrow$ 
     $\mathcal{L}_F[\![e_2]\!] \ (\text{Regular Scalar} : e_1^{flat} :_R \text{env})$ 
  =  $(\text{nest } r, \text{let } v_0 = \text{enum}_R \ \text{ctx } e_1^{flat} \ \text{in } e_2^{flat})$ 
  -- Array is irregular or in irregular context
  |  $(r_1, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \text{ env}$ 
  ,  $\text{segs} \leftarrow \text{segmentsOf } r_1 \ e_1^{flat}$ 
  ,  $(r_2, e_2^{flat}) \leftarrow$ 
     $\mathcal{L}_F[\![e_2]\!] \ (\text{Irregular Scalar} : \text{segs} :_{I_r} \text{env})$ 
  =  $(\text{nest } r_2, \text{let } v_0 = \text{enum}_{I_r} \ (\text{context env}) \ \text{segs}$ 
     $\text{in } e_2^{flat})$ 
 $\mathcal{L}[\![\text{fold } (\lambda v_1 \ v_0. e_1) \ e_2 \ e_3]\!] \text{ env}$ 
  |  $(\text{Avoid}_S, e_2^{flat}) \leftarrow \mathcal{L}[\![e_2]\!] \text{ env}$ 
  ,  $(\text{Avoid}_S, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \ (\text{Avoid} : \text{env})$ 
  ,  $(\text{Avoid}_A, e_3^{flat}) \leftarrow \mathcal{L}[\![e_3]\!] \text{ env}$ 
  =  $(\text{Avoid}_A, \text{fold } (\lambda v_1 \ v_0. e_1^{flat}) \ e_2^{flat} \ e_3^{flat})$ 
  |  $(\text{Avoid}_S, e_2^{flat}) \leftarrow \mathcal{L}[\![e_2]\!] \text{ env}$ 
  ,  $(\text{Avoid}_S, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \ (\text{Avoid} : \text{env})$ 
  ,  $(\text{Regular (Nest } r), e_3^{flat}) \leftarrow \mathcal{L}[\![e_3]\!] \text{ env}$ 
  =  $(\text{Regular } r, \text{fold } (\lambda v_1 \ v_0. e_1^{flat}) \ e_2^{flat} \ e_3^{flat})$ 
  |  $(\text{Avoid}_S, e_2^{flat}) \leftarrow \mathcal{L}[\![e_2]\!] \text{ env}$ 
  ,  $(\text{Avoid}_S, e_1^{flat}) \leftarrow \mathcal{L}[\![e_1]\!] \ (\text{Avoid} : \text{env})$ 
  ,  $(\text{Irregular (Nest } r), e_3^{flat}) \leftarrow \mathcal{L}[\![e_3]\!] \text{ env}$ 
  =  $(\text{Irregular } r, \text{fold}_{\text{seg}} \ (\lambda v_1 \ v_0. e_1^{flat}) \ e_2^{flat} \ e_3^{flat})$ 

```

Figure 4.7: The lifting transformation

Variables

When flattening a variable, the transformation refers to the auxiliary function `var` discussed previously. If one views the variable as just a simple Debruijn index, then the variable itself is left unchanged but may have a series of replicates applied to it to make it *fit* the current context. Because our indices are typed, we need to produce a witness of the relation between the type of the variable in the original term and the type in the flattened term.

Let bindings

The let-binding rule shows how bindings are introduced and how they are tracked via `Env`. The binding is first flattened, giving a \mathcal{F} -witness. This witness is placed into the environment for flattening the body of the let-binding. This ensures that any occurrences of that variable are assigned the right type.

To avoid any confusion, the let-binding we are introducing here exists in the target language, not the meta-language. We assume that the let-syntax of our meta-language is overloaded to construct let bindings in either the meta language or the object language as necessary.

Tuples and tuple projection

To lift a tuple constructor, we lift each component and construct a new tuple containing them. Similarly to let bindings, we assume that our meta language overloads tuple construction to support construction in our object language.

In the case of tuple projection, it is just a matter of projecting the same component from the lifted tuple.

Primitive operations

For unary operations, lifting is straightforward. We simply `map` the operation over the lifted operand.

It gets more complex with binary operations as we have the possibility that each operand ends up having a different representation after lifting. Each could be either avoided, regular or irregular. In total, this gives us 9 different possible combinations, but we can cheat a bit by handling symmetric cases by swapping the arguments of the operation.

```
liftBinOp :: (IsScalar e1, IsScalar e2)
  => (Expr Γ e1 → Expr Γ e2 → Expr Γ e)
  →  $\mathcal{F}$  sh $\mathcal{L}$  e1 e1flat
  →  $\mathcal{F}$  sh $\mathcal{L}$  e2 e2flat
  → Expr Γ e1flat
  → Expr Γ e2flat
  → (∃ eflat. ( $\mathcal{F}$  sh $\mathcal{L}$  e eflat, Expr Γ eflat))

liftBinOp p r1 r2 e1 e2
```

```

= case (r1, r2) of
  (AvoidS,      AvoidS)      → p e1 e2
  (AvoidS,      Regular Scalar) → map (λv0. p e1 v0) e2
  (AvoidS,      Irregular Scalar) → (prj 1 e2, map (p e1) (prj 2 e2))
  (Regular Scalar, Regular Scalar) → zipWith p e1 e2
  (Regular Scalar, Irregular Scalar) → zipWithseg p (irregular e1) e2
  (Irregular Scalar, Irregular Scalar) → zipWithseg p e1 e2
  -                                   → liftBinOp (flip p) r2 r1 e2 e1

```

Array shapes

With `extent`, if the expression it is applied to does not need to be flattened, then it just returns the original expression.

The more interesting case occurs when the argument expression flattens with a regular representation. In this case, we simply take the shape of the resulting array and strip off the outer dimensions corresponding to the lifting shape and the inner dimensions corresponding to the original array. This is done with `extentR`.

```

extentR :: Expr Γ (Array (shL+(sh#sh1)) e1, ... )
          → Expr Γ sh
extentR (arr, ... ) = indexLeft (indexRight (extent arr))

```

This is the one case where flattening can still be avoided even if an expression contains a subexpression that had to be flattened.

We do something similar in the irregular case with `extentIr`, but here we cannot avoid flattening.

```

extentIr :: Expr Γflat ((Segments (shL+(sh#sh1)), Vector e1), ... )
          → Expr Γflat (Segments shL, Vector sh)
extentIr arr = expand (lefts (reassoc (prj 1 (prj 1 arr))))

reassoc :: Expr Γ (Segments (sh#(sh'#sh'')))
          → Expr Γ (Segments ((sh#sh')#sh''))
reassoc = id  -- encoded in type system

```

Here, we take the segment descriptors and using `lefts` extract new descriptors that ignore the inner dimensionality. We use `reassoc` to *reassociate* the ordering of each level of nesting. This is not necessary, as the type system handles this for us, but it helps us to see what is going on. We start with

```
Segments (shL+(sh#sh1))
```

We then reassociate to

```
Segments ((shL#sh)#sh1)
```

And remove the innermost level of nesting

```
Segments (shL#sh)
```

The **expand** operation

```
expand :: Expr  $\Gamma$  (Segments (sh#sh'))  $\rightarrow$  Expr  $\Gamma$  (Segments sh, Vector sh')
```

then lets us split this up into a set of segment descriptors that describe a mapping into a vector of extents.

Indexing

The rules for indexing (!) are defined in terms of \mathcal{L}_F , which enforces flattening by ignoring **Avoid_S** and **Avoid_A** in the flattening of the term e that constitutes the first argument of the indexing operation. The exact definition is in Figure 4.8

The lifted indexing itself is relatively straightforward. In the regular case, we take a series of slices of the input arrays.

```
(!R) :: Expr  $\Gamma$  (Array (shL#sh#sh1) e, ... )
       $\rightarrow$  Expr  $\Gamma$  (Array shL sh)
       $\rightarrow$  Expr  $\Gamma$  (Array (shL#sh1) e, ... )
arr !R ix = ( generate (outside (extent arr)) ( $\lambda v_0$  . prj 0 arr ! inside (ix ! indexLeft v0) v0)
              , ... )
```

We use **outside** as defined in 4.6 in order to construct an array of shape **sh_L#sh₁**. Then by doing some index manipulation, we can pull out the corresponding elements from the input array.

For irregular indexing, it works in much the same way, except for needing to work with the segmented representation.

```
(!Ir) :: Shape sh
       $\Rightarrow$  Expr  $\Gamma$  ((Segments (shL#sh#sh1), Vector e1), ... )
       $\rightarrow$  Expr  $\Gamma$  (Segments shL, Vector sh)
       $\rightarrow$  Expr  $\Gamma$  ((Segments (shL#sh1), Vector e1), ... )
arr !Ir ix =
  ( let segs = (outsides (prj 0 (prj 1 arr)))
    in segs, generateseg segs ( $\lambda v_0$  . prj 2 (prj 1 arr) ! inside (ix ! prj 1 ix # indexLeft v0) v0)
    , ... )
```

Array construction

The **generate** rule is one of the more interesting ones. In our language, it is the only way in which nesting can be introduced. All other operations either remove nesting or preserve it. Because of this, we have to handle a greater number of possible cases. These cases are handled by a backtrack search. First, we have to check if flattening can be avoided entirely, which would be the cheapest solution. This is only true if flattening can be avoided for both arguments and the second argument is scalar.

If it is not possible to avoid flattening for both arguments, but it is for the first argument (the extent of the output), then we can be smarter. We know that this occurrence of **generate**

```

 $\mathcal{L}_F[\![\cdot]\!] :: \text{Expr } \Gamma \ t \rightarrow \text{Env } \text{sh}_\mathcal{L} \ \Gamma \ \Gamma^{flat} \rightarrow (\exists \ t^{flat} . (F \ \text{sh}_\mathcal{L} \ t \ t^{flat} , \text{Expr } \Gamma \ t^{flat}))$ 
 $\mathcal{L}_F[\![e]\!] \text{ env} \mid (r, e^{flat}) \leftarrow \mathcal{L}[\![e]\!] \text{ env}$ 
 $\quad = \text{replicate } r \ (\text{context env}) \ e^{flat}$ 

context :: Env shℒ Γ Γflat → Either (Expr Γflat shℒ) (Expr Γflat (Segments shℒ))
context [] = Left Z
context (_ : env) =
  case context env of
    Left sh → Left (weaken sh)
    Right segs → Right (weaken segs)
context (sh' :R env) =
  case context env of
    Left sh → Left (sh # sh')
    Right segs → Right (segs `cross` segmented (unit sh))
context (segs' :Ir env) =
  case context env of
    Left sh → Right (segments (unit sh) `cross` segs')
    Right segs → Right (segs `cross` segs')

replicate :: F t tflat
  → Either (Expr Γ shℒ) (Expr Γ (Segments shℒ))
  → Expr Γ tflat
  → (∃ tflat '. (F shℒ t tflat ', Expr Γ tflat '))
replicate AvoidS      ctx t = replicateS ctx t
replicate AvoidA      ctx t = replicateA ctx t
replicate (TupleF (r1, ..., rn)) ctx t | (r1flat, t1) ← replicate r1 ctx (prj 1 t)
  , ...
  . (rnflat, tn) ← replicate rn ctx (prj n t)
  = (TupleF (r1flat, ..., rnflat) (t1, ..., tn))

replicateS :: IsScalar t
  ⇒ Either (Expr Γ shℒ) (Expr Γ (Segments shℒ))
  → Expr Γ t
  → (∃ tflat . (F shℒ t tflat , Expr Γ tflat))
replicateS (Left ctx) t = (Regular Scalar, generate ctx (λ_ . weaken t))
replicateS (Right ctx) t = (Irregular Scalar, generateseg ctx (λ_ . weaken t))

replicateA :: IsScalar e
  ⇒ Either (Expr Γ shℒ) (Expr Γ (Segments shℒ))
  → Expr Γ (Array sh e)
  → (∃ tflat . (F shℒ (Array sh e) tflat , Expr Γ tflat))
replicateA (Left ctx) t = ( Regular (Nest Scalar)
  , backpermute (ctx # extent t) indexRight)
replicateA (Right ctx) t = ( Irregular (Nest Scalar)
  , let segs = ctx # segments (unit (extent t))
    in (segs, generateseg segs (ix → t ! indexRight ix)))

```

Figure 4.8: Forced lifting

is not introducing any new irregular nesting. However, it may still be getting used in an irregular context, or its second argument (the generation function) may itself be introducing irregularity. For this reason, we need to use `context`, given in Figure 4.8, to determine whether we are in a regular or irregular context. If we are in regular context, then we do a lifted enumeration with `enumR`.

```
enumR :: Shape sh ⇒ Expr Γ shL → Expr Γ sh → Expr Γ (Array (shL#sh) sh)
enumR shL sh = generate (shL#sh) (λv0 . indexRight v0)
```

The intuition behind this function is it produces a set of enumerations we wish to generate values for. As an example,

```
enumR 2 (Z:.2) (Z:.3) = { {Z:.0, Z:.1, Z:.2}
                          , {Z:.0, Z:.1, Z:.2} }
```

Here, we are producing a 2D array containing 2 enumerations of `Z:.3`. In 4.7, we use an enumeration like this to satisfy our generation function. This gives us the actual result we need, but to convince the type checker of the meta-language of this fact, we have to do some manipulation of the \mathcal{F} relation witness.

```
nest :: F (shL#sh) e eflat
      → F shL (Array sh e) eflat
nest (Regular r)      = Regular (Nest r)
nest (Irregular r)    = Irregular (Nest r)
nest (Tuple (r1, ... , rn)) = Tuple (nest r1, ... , nest rn)
```

This function, captures a key insight of the flattening transform. Namely, that a term of type `e` flattened under the context `shL#sh` would have the same possible flattened representations as a term of type `Array sh e` flattened under the context `shL`.

We are left with the final case. This occurs when flattening cannot be avoided for the first argument. In such a case, irregular nesting is being introduced. We must extract segment descriptors from the flattened first argument with `segmentsOf`

```
segmentsOf :: Shape sh
           ⇒ F shL sh shflat
           → Expr Γ shflat
           → Expr Γ (Segments sh)
segmentsOf AvoidS sh      = segmented (unit sh)
segmentsOf (Regular Scalar) sh = segmented sh
segmentsOf (Irregular Scalar) sh = prj 1 sh `cross` segmented (prj 2 sh)
```

We then use these segments, and the current lifting shape to construct segmented enumerations.

```
enumIr :: Either (Expr Γ shL) (Expr Γ (Segments shL))
        → Expr Γ (Segments sh)
        → Expr Γ (Segments (shL#sh), Vector sh)
enumIr (Left ctx) segs = let segs' = segments (unit ctx) `cross` segs
                        in (segs', generateseg segs' (λv0. indexRight v0))
```

```
enumIr (Right ctx) segs = let segs' = ctx `cross` segs
                          in (segs', generateseg segs' (λv0. indexRight v0))
```

The backtracking we rely on for lifting **generate** could, in theory, lead to exponential work complexity. In the implementation described in chapter 5, this is not a problem, as there is no deep nesting of **generate**-expressions. Keller et al. [30], who have to distinguish between only two cases, circumvent backtracking by splitting the transformation into an analysis phase which first labels the expression, followed by a separate lifting phase. Such an approach would be compatible with what is described here and likely be necessary for a language where deep nesting was more prevalent.

4.3.3 Array reduction

EXAMINER: *Section 4.3.3 states that programs that rely on parallelism in reduction functions will fail. What happens in such cases? The compiler crashes? The generated program crashes? It produces the incorrect result?*

For **fold**, the type system ensures that the function it is applied to is a sequential computation over scalars, and the initial value is a scalar as well. This does not, however, preclude the possibility that one of them may contain parallelism. This presents a problem for us as a fold with a reduction function containing parallelism is not typically supported by data-parallel models of computing. For this reason, we leave the transform as partial in this regard. Flattening will fail for programs that rely on parallelism in reduction functions. This is true of other nested data parallel languages (e.g. Data Parallel Haskell [9] and NESL [5]),

Flattening **folds** is relatively straightforward compared to **generates**. It all depends on the 3rd argument. If the flattening of it can be avoided, then the entire fold can be avoided. If it has a regular flattened representation, then it can be reduced just by using a normal **fold**. If the representation is irregular, then **fold_{seg}** is used.

Chapter 5

Implementation and Optimisation

To demonstrate the utility of sequences, we must explore what is required to implement them in the feature-rich array language, Accelerate. We will provide a high-level overview of the implementation. For more detailed information we refer to the publicly available implementation.¹

We first address how sequences fit into Accelerate’s existing compiler infrastructure (Section 5.1), before describing the implementation of the flattening transform (Section 5.2). However, this alone is not sufficient to achieve competitive absolute performance for sequence programs, so we also describe extensions to Accelerate’s optimisation passes (Section 5.3). Lastly, to execute sequence computation with the desired property of minimised memory usage, we require a dynamic scheduler (Section 5.4).

5.1 Architecture

As discussed in Chapter 2, Accelerate has a fairly standard compiler architecture: a multi-stage pipeline. Prior to our extension, this pipeline consisted of these stages.

- **Sharing recovery** – Converts the higher-order abstract syntax (HOAS) into first-order syntax with Debruijn indices and also recovers sharing.
- **Fusion and optimisation** – Tries to minimise unnecessary array traversals through fusion, folds constant expressions, performs Constant Subexpression Elimination (CSE), and shrinks terms.
- **Code generation** – Generates LLVM IR from AST and passes it off to the LLVM compiler. Binaries produced are either x86 (with vector instructions) or PTX, for CUDA capable GPUs.
- **Execution** – Schedules and executes the binaries on the target hardware.

¹This is available at <https://github.com/AccelerateHS/accelerate/tree/feature/sequences>

To support sequence computations, we have to both extend the AST with a set of sequence combinators and change the pipeline to support them. The changes needed, at the high-level, are:

- An update to sharing recovery to support the additional sequence AST constructors. This will not be explored in detail as it is not novel or different to how Accelerate handles other AST forms. See McDonnell et al. [36] for details on how sharing recovery works.
- An additional flattening stage after sharing recovery to flatten the array computations embedded in sequence computations (Section 5.2).
- An extension to the fusion system to exploit additional fusion opportunities presented by sequence computations (Section 5.3).
- An additional runtime system component for scheduling sequence computations such that memory usage is minimised while processor utilisation is maximised (Section 5.4).

5.1.1 What is a sequence?

As introduced in Section 3.2, a sequence computation `Seq [Array sh b]` adds a second level of nested, irregular data-parallelism on top of the flat regular parallelism of an array computation `Acc (Array sh b)`. Hence, we can generally regard such a sequence computation as a mapping of a flat, regular function `f` over an irregular sequence `xs`:

```
Seq [Array sh b] ~ mapSeq f xs
where
  f :: Acc (Array sh' a) → Acc (Array sh b)
  xs :: Seq [Array sh' a]
```

In addition to the *inner-function* parallelism in `f`, we want to exploit as much of the parallelism of the outer `mapSeq` (the *intra-function* parallelism) as possible on a given architecture to achieve optimal performance. That is, we execute multiple—but not necessarily *all*—elements of the `mapSeq` in parallel. Avoiding the need to utilise all outer parallelism helps us to support out-of-core datasets on GPUs, by only loading into device memory those elements of the sequence `xs` that are processed together. By dynamically adjusting the number of elements operated on at once, we aim to minimise memory usage while keeping all processing elements fully utilised. The flattening transformation takes the definition of the function `f` and rewrites it to a definition for `f†`, such that semantically `f† = mapSeq f`; in other words, `f†` can process multiple elements of `xs` at once, in parallel.

In the implementation we describe here, we restrict the level of sequence nesting to a depth of one; that is, we support sequences of arrays, but not sequences of sequences of arrays. Previous work [32] showed that the efficient implementation of more deeply nested

irregular structures requires sophisticated runtime support whose SIMD implementation (e.g., for GPUs) raises an entire set of questions in its own right. However, even a single level of nesting is sufficient to encode many examples without resorting to flattening by hand, as we highlight in Chapter 7.

5.2 Flattening

The design of the flattening stage closely follows the what is described in 4.3. However, it differs in a few key areas.

- There are considerably more combinators in Accelerate than ACC_{NDP} . This means there needs to be lifted versions of nearly all combinators. In many cases, there is both a regular and irregular lifted equivalent.
- There are no nested arrays and sequences are only one level of nesting. This means that the implementation of the transform can be simplified in a few places, but fundamentally remains the same.
- Accelerate uses representation types to handle tuples of multiple arities. In ACC_{NDP} we just assumed that we could easily generalise over tuples of arbitrary arity.

We address these points as we give an overview of the flattening implementation. Like we did with ACC_{NDP} , we will begin by first describing the type level component before moving on to the actual program transform.

5.2.1 Types

Recall that, to represent values in our language at a higher dimension; we had to first normalise then flatten types. This was captured by these relations:

```
data  $\mathcal{N}$  t t' where
  Scalar :: IsScalar e
           $\Rightarrow$   $\mathcal{N}$  e (Array Z e)
  Nest   ::  $\mathcal{N}$  e (Array sh1 e1, Array sh2 e2, ... , Array shn en)
           $\rightarrow$   $\mathcal{N}$  (Array sh e) (Array (sh # sh1) e1, Array (sh # sh2) e2, ... , Array (sh # shn) en)
  Tuple $\mathcal{N}$  :: ( $\mathcal{N}$  t1 t1',  $\mathcal{N}$  t2 t2', ... ,  $\mathcal{N}$  tn tn')
            $\rightarrow$   $\mathcal{N}$  (t1, t2, ... , tn) (t1', t2', ... , tn')

data  $\mathcal{F}$  sh $\mathcal{L}$  t tflat where
  AvoidS  :: IsScalar t
            $\Rightarrow$   $\mathcal{F}$  sh $\mathcal{L}$  t t
           -- avoid vectorisation
  AvoidA  :: IsScalar e
            $\Rightarrow$   $\mathcal{F}$  sh $\mathcal{L}$  (Array sh e) (Array sh e)
  Regular ::  $\mathcal{N}$  t (Array sh1 e1, Array sh2 e2, ... , Array shn en) -- regular context
            $\rightarrow$   $\mathcal{F}$  sh $\mathcal{L}$  t ( Array (sh $\mathcal{L}$ #sh1) e1
                        , Array (sh $\mathcal{L}$ #sh2) e2
                        , ...
```

```

    , Array (shL#shn) en)
Irregular ::  $\mathcal{N}$  t (Array sh1 e1, Array sh2 e2, ... , Array shn en) -- irregular context
    → F shL t ( (Segments (shL#sh1), Vector e1)
    , (Segments (shL#sh2), Vector e2)
    , ...
    , (Segments (shL#shn), Vector en))
TupleF   :: (F shL t1 t1flat, F shL t2 t2flat, ... , F shL tn tnflat)
    → F shL (t1, t2, .., tn) (t1flat, t2flat, ... , tnflat)

```

Because of the differences between Accelerate and ACC_{NDP} , in the implementation we combine both of these relations into a single GADT.

```

data LiftedType t t' where
  UnitT      :: LiftedType () ()
  LiftedUnitT :: LiftedType () (Scalar Int)
  AvoidedT   :: (Shape sh, Elt e) ⇒ LiftedType (Array sh e) (Array sh e)
  RegularT   :: (Shape sh, Elt e) ⇒ LiftedType (Array sh e) (Array (sh:.Int) e)
  IrregularT :: (Shape sh, Elt e) ⇒ LiftedType (Array sh e) (Segments sh, Vector e)
  TupleT     :: (IsProduct Arrays t, IsProduct Arrays t')
    ⇒ LiftedTupleType (TupleRepr t) (TupleRepr t')
    → LiftedType t t'

```

It is important to note that this structure is not parameterised by the lifting shape. This is because of the lack of deep nesting in Accelerate. As our only nested structure is the sequence, and sequences themselves cannot be nested, the only possible nested context is being an element of a sequence. Hence, we can represent all possible lifting shapes with an `Int`.

The first two constructors deal with the unit type. This is something we did not encounter in Chapter 4. In short, the unit type `()` in a lifted context is represented by the lifting shape. There is only one possible unit type, so a collection of elements of that type is equivalent to the shape of that collection. For a sequence, that context is an `Int` indicating the length of the sequence (or the size of the current chunk of the sequence; more on that below).

The `AvoidedT` constructor is equivalent to Avoid_A . In the array language strata of Accelerate (Acc), scalar values do not occur so Avoid_S is not needed.

`RegularT` and `IrregularT` are equivalent to `Regular` and `Irregular` respectively, but without being generalised over tuples. For the regular case, we simply add the lifting shape to the dimensionality of the array. This is equivalent to $\text{sh}_L \# \text{sh}$ but sh_L in this case is just $\mathbb{Z}:\text{Int}$. For the irregular case, we have a slightly different notion of segment descriptors than before. The lifting shape is implicit in the constructor. What this means is that `Segments (sh:.Int)` in ACC_{NDP} is equivalent to `Segments sh` in Accelerate. The outer dimension is assumed to always be present.

To correctly handle tuples, we have the `TupleT` constructor. It captures the `IsProduct` constraint on both the original type `t` and the transformed type `t'`. In essence, it means they are both products of which all components are members of the `Arrays` type class. To specify the lifted type of each component in this product we use `LiftedTupleType`.

Recall from Section 2.2 that Accelerate uses representation types such that arbitrary tuples are flattened into left nested tuples.

EXAMINER: Section 5.2.2 mentions that Sequences can only contain regular arrays. What would it take to extend the current system to support nesting of Sequences? Either Chapter 3,4 or 5 should at least discuss this and explain why the current limitation is fine or what it would take to make it more general and support arbitrary nesting of sequences.

$$\text{Segments}_{NDP} \text{ (Z::Int::Int::Int)} = (\text{Vector Int}, \text{Vector Int})$$

```
(#) :: SegmentsNDP (Z::Int::Int::Int) → Z::Int::Int::Int → Int
(k,j) @ (Z::z::y::x) = let kz = k !! z
                        jy = j !! (kz + y)
                        ix = jy + x
in ix
```

```
Segments_NDP (Z::Int::Int::Int) = Vector (Z::Int::Int, Int)
```

```
(#) :: SegmentsNDP (Z:.Int:.Int:.Int) → Z:.Int:.Int:.Int → Int
segs # (Z:.z:.y:.x) = let (sh,off) = segs !! z
                        in off + toIndex sh (Z:.y:.x)
```

This is the multi-dimensional equivalent of the size-offset representation in Chapter 2. This representation is of size l . Therefore, its space requirement is significantly less than the more general representation. Furthermore, implementing (#) only requires a single indexing operation.

Returning to our concept of sequences, we see that we only need this second way of representing the segment descriptors. Sequences can only contain regular arrays, so we only have irregularity in the outer dimension. Therefore our definition for **Segments** is

```
SegmentsNDP (sh:.Int) = Vector (sh, Int)
```

For Accelerate, we make some small changes. Firstly, as noted above, we only ever have segment descriptors of dimension at least one, so we can ignore that part of the parameter. Secondly, as shown in Section 2.1, there is a benefit to storing the offsets alongside the extents. Lastly, we often need access to the total size of all segments. This can be calculated by taking the last offset and the size of the last shape and adding them together but doing this calculation each time we need this value interferes with other optimisations. Hence, we store this value separately.

```
Segments sh = (Vector (sh, Int), Scalar Int)
```

Our final definition is a vector containing the extent and offset of each segment, and an **Int** representing the total size of all segments.

5.2.3 Transformation

With a concrete representation for segment descriptors, we can fully implement the lifting transform in Accelerate.

Recall ACC_{NDP} 's lifting environment:

```
data Env shL  $\Gamma$   $\Gamma^{flat}$  where
  []      :: Env Z [] []
  (:)     ::  $\mathcal{F}$  t tflat -- standard free variable
           → Env shL  $\Gamma$   $\Gamma^{flat}$ 
           → Env shL (t :  $\Gamma$ ) (tflat :  $\Gamma^{flat}$ )
  (:R)    :: Shape sh -- regular flattening context
           ⇒ Expr  $\Gamma^{flat}$  sh
           → Env shL  $\Gamma$   $\Gamma^{flat}$ 
           → Env (shL#sh)  $\Gamma$   $\Gamma^{flat}$ 
  (:Lr)   :: Shape sh -- irregular flattening context
           → Expr  $\Gamma^{flat}$  (Segments sh)
           → Env shL  $\Gamma$   $\Gamma^{flat}$ 
           → Env (shL#sh)  $\Gamma$   $\Gamma^{flat}$ 
```

In Accelerate, we have the same structure. The main difference is that we don't make the lifting shape explicit in the type.

```
data Environment acc aenv aenv' where
```

```

BaseC      :: Environment acc aenv aenv

PushC      :: Arrays t'
            => Environment acc aenv aenv'
            -> LiftedType t t'
            -> Nesting acc (aenv', t')
            -> Environment acc (aenv, t) (aenv', t')

data Nesting acc aenv = NoNest
                      | RegularNest (acc aenv (Scalar Int))
                      | ∀ sh. Shape sh => IrregularNest (acc aenv (Segments sh))

```

We also combine the different cons-like constructors into one with an additional *nesting indicator* to track nesting information. The `acc` type parameter exists as a consequence of the *tying the recursive knot* trick described in Section 2.2.

The transform itself proceeds in much the same way as for ACC_{NDP} . The result of the lifting transform, which in ACC_{NDP} was

$$\exists t^{flat}. (F \text{ sh}_t \ t \ t^{flat}, \text{Expr } \Gamma^{flat} \ t^{flat})$$

can be directly converted into this GADT.

```

data LiftedAcc acc aenv t where
  LiftedAcc :: Arrays t' => LiftedType t t' -> acc aenv t' -> LiftedAcc acc aenv t

```

5.2.4 Lifted operations

EXAMINER: Section 5.2.4 explains that stencil are not supported. Why is this case? Is there a major limitation in the implementation that makes these hard to support? Or could these be easily implemented given more time? This should be discussed in details otherwise one might questioned the generality of the proposed approach. Another related question I have is with respect to the extensibility/scaling of the proposed approach. It would be good if this section could discuss in general what is needed to support other operations that might be added later into the language (or are currently in Accelerate such as stencils).

In ACC_{NDP} we had to have `foldseg` and `generateseg` as the segmented versions of existing operations. In Accelerate we require both of these, as well as segmented versions of many other combinators. Some of these are available as language primitives, meaning each backend has its own implementation. Others, however, have to be implemented with the existing language constructs.

As described in Section 2.2, the internal syntax of Accelerate is first-order and uses typed De Bruijn indices. Consequentially, the lifted version of operations, which are specialised and spliced into the output program, must also be written in this way. In this section, for simplicity, take we take the liberty of writing them in the familiar Accelerate frontend language instead.

We divide Accelerate’s array combinators into these categories:

- Producers – This includes `generate`, `map`, `zipWith` and `backpermute`. All of these can be implemented in terms of `generate` with indexing.
- Folds – This includes `fold` and `fold1`.
- Scans – This includes `scanl`, `scanr`, `scanl'`, `scanr'`.
- Stencil – Accelerate has built-in support for stencil operations. We do not support these with sequence computations, leaving it for future work.

What follows is an explanation for the implementations of the *lifted irregular* versions of each category. For the regular versions, the combinators are already rank-polymorphic, so little additional work is necessary. For lifted irregular, or segmented, operations we typically have to do more involved computations involving the segment descriptors.

Producers

As all of these combinators can be implemented in terms of `generate`, their segmented versions can be implemented in terms of a segmented `generate`.

```
generateSeg :: (Shape sh, Elt e)
  => Acc (Segments sh)
  -> (Exp Int -> Exp sh -> Exp e)
  -> Acc (Vector e)
generateSeg (unlift -> (shapes, offsets, totalSize)) f
  = map (\(unlift -> (n,i)) -> f n (toIndex (shapes ! n) i)) indices
where
  ones      = fill (index1 totalSize) (-1,1)
  zeroes    = generate (shape offsets) (\ix -> lift (unindex1 ix, 0))
  heads     = scatter offsets ones zeroes
  indices   = scanl1 (⊕) heads
  (unlift -> (a,b)) ⊕ (unlift -> (a',b')) = a == a' ? (lift (a, b+b'), lift (a',0))
```

We first create a vector of pairs of the same length as the total size of the segmented array (`ones`). The first and second component of each pair are `-1` and `1` respectively. We use `-1` here because this value needs to be outside the range of valid indices. We then create an indexed vector that is the same length as the offsets vector (`zeroes`). It contains the index of the element as the first component of each pair and `0` as the second. Using these two vectors, we then compute a vector of head flags (`heads`). This vector has a pair of `(-1,1)` at all positions except for those at the start of each segment where there is the number of the segment and a `0`. We do this via `scatter`.

```
scatter :: Elt e
  => Acc (Vector Int)      -- destination indices to scatter into
  -> Acc (Vector e)       -- default values
  -> Acc (Vector e)       -- source values
  -> Acc (Vector e)
```

By performing a scan over this flags vector, we can construct the `indices` for the segmented generate. It is a vector of pairs where the first component is the segment that element occurs in and the second is the index of that element within the segment. Using these indices we can then compute the result of the segmented generate.

Folds

Accelerate has two non-segmented versions of fold, `fold` and `fold1`. Both of these need flattened equivalents. Fortunately, this is simpler than flattening generate as Accelerate provides native `foldSeg` and `foldSeg1`. We still have to do some work to get our input in the right form for these operations, but we do not have to fully implement it or make scheduling decisions in the process of flattening.

We will describe the segmented version of `fold`. Doing the same for `fold1` is similar enough that we elide it. Here is the type signature for `foldSeg` in Accelerate.

```
foldSeg :: (Shape sh, Elt a, Elt i, IsIntegral i)
  => (Exp a → Exp a → Exp a)      -- The reduction function
  → Exp a                          -- The initial value
  → Acc (Array (sh:.Int) a)         -- The array to reduce
  → Acc (Vector i)                  -- Segment descriptors of the inner dimension
  → Acc (Array (sh:.Int) a)
```

First observe that it expects a vector of integral segment descriptors, but a potentially multidimensional array of values. This version of a segmented fold allows for multiple segmented reductions to be performed over the inner dimension of an array. This is more general than we require, so we specialise it to

```
foldSeg1D :: Elt a
  => (Exp a → Exp a → Exp a)
  → Exp a
  → Acc (Vector a)
  → Acc (Vector Int)
  → Acc (Vector a)
foldSeg1D = foldSeg
```

This gives us something close to the segmented folds described in Section 2.1. Using this, we can build the segmented fold we had in Section 4.3. First, we need to convert our higher dimensional segment descriptors into the integral segment descriptors.

```
segSizes :: Shape sh => Acc (Segments sh) → Acc (Vector Int)
segSizes (unlift → (shapes, _, _)) = map shapeSize shapes
```

Then, we apply `foldSeg1D` to the integral segments and the values vector. This gives us the values vector of the result, but we still need to compute the segment descriptors.

We do that with the following function.

```
foldSegments :: Shape sh => Acc (Segments (sh:.Int)) → Acc (Segments sh)
foldSegments (unlift → (shapes, _, _)) = segsFromShapes (map indexTail shapes)
```

Here, `segsFromShapes` computes segment descriptors from a vector of shapes.

```
segsFromShapes :: Shape sh => Acc (Vector sh) -> Acc (Segments sh)
segsFromShapes shapes = let (offs, totalSize) = scanl' shapes
                           in (shapes, offs, totalSize)
```

The `scanl'` combinator computes the left-scan of its input, but rather than returning a vector of length $n + 1$, where n is the size of the input, it returns a vector of length n and a scalar value containing the final element.

Scans

For the segmented version of the various scans, we define them in terms of a segmented generate and a non-segmented scan. The approach taken is similar for all the different variants. We described the simplest variant here `scanl1` and refer to the representation for the implementations of segmented `scanl`, `scanr`, `scanr1`, `scanl'`, `scanr'`, etc.

```
scanl1Seg :: (Exp e -> Exp e -> Exp e)
           -> Acc (Segments (sh::Int))
           -> Acc (Vector e)
           -> Acc (Vector e)
scanl1Seg f z segs vals
  = map fst (scanl1 f' (zip vals flags))
  where
    flags = generate_seg segs (\n (unlift -> sh::i) -> i == 0)
    f' (e,_) (e',flag) = (flag ? (e', f e e'), false)
```

We first compute a flags vector of the same length as the values vector. It contains `true` at the start of each segment and `false` everywhere else. Then we create a version of the binary function `f` that only applies it when this flag is `false`. This technique of tagging the values with a flag indicating whether they are at the start of a segment is similar to previous approaches[4].

5.3 Fusion

While we have shown that it is possible to take our sequence programs with nesting and transform them so that the nesting is removed, doing so has introduced a performance problem we first highlighted in Section 2.1.4. We introduce *numerous* extra intermediate arrays in our version of the flattening transform.

5.3.1 Array fusion

Recall that in Accelerate, fusion is achieved by representing arrays by their extent and function mapping indices to their values. As a consequence of that, terms of the form

```
let xs = use (Array ... )
```



```
ys = map f xs
in zipWith g ys ys
```

`ys` is not fused. That is because `ys` is used twice, and if it were fused, the potentially expensive operation `f` would be executed for each element of `xs` twice.

However, this approach does not take into account what the term `ys` actually *is*; it simply sees that `ys` occurs twice in `zipWith` and so refuses to fuse it further. Consider the following example:

```
let xs = use (Array ... )
    ys = (map f1 xs, map f2 xs)
in
  zipWith g (fst ys) (snd ys)
```

Although the term `ys` still occurs twice in the `zipWith`, we can see that the individual components of the tuple each occur only once, and thus should still be subject to fusion.

This lack of fusion in the regular Accelerate optimisation system is particularly problematic for us, as we represent irregular sequences as a pair consisting of the segment descriptors together with a vector of the values. Thus, in the prior Accelerate fusion system, irregular sequences would never fuse, severely impacting performance. We extend the Accelerate fusion system to fuse the individual components of a tuple independently, which improves the performance of both our sequence computations as well as regular Accelerate programs.

To explain this, we will first describe more concretely how Accelerate's prior fusion system works, then cover the *fusion-through-tuples* implemented as part of Accelerate sequences. To start with, consider this program here

```
λxs → map (+1) (map (*2) xs)
```

we fuse it by a bottom up traversal. We must first *delay* `xs` into

```
(shape xs, (xs !))
```

The mapping function is just indexing `xs`. Proceeding up the AST, the `map (*2)` can now be delayed by taking the delayed form of `xs`, composing it with `*2` and doing some further simplification:

```
(shape xs, (λix → (xs ! ix)*2))
```

In general, if arrays are represented in delayed form:

```
map f (sh, g) = (sh, f . g)
```

We can then do the same with the `map (+1)`, giving us a final form of

```
λxs → (shape xs, (λix → (xs ! ix)*2 + 1))
```

This can be brought back into array form by turning the delayed array into a generate.

```
λxs → generate (shape xs) (λix → (xs ! ix)*2 + 1)
```

This process works for all producer combinators. However, the consumer combinators (`fold`, `scanl`, etc.) cannot be delayed, so those must be left as *manifest*: still introducing intermediate arrays.

The other time intermediate arrays must be introduced, as already described, is when an array is accessed more than once. In Accelerate, this can only occur with `let` bindings. Looking at this example here,

```
let a = generate sh f
in map g a
```

we see that it would be simple, and of no obvious disadvantage if `g` does not reference `a`, to embed (inline) the binding `a` into the body to expose fusion opportunities there. In general, though, that is not always desired, as highlighted above. So, when should we embed bindings like this? Many possible measures could be used, but we opt for a simple one. Accelerate inlines whenever a variable is only “used” once. By “used” here, we mean how often the variable occurs in the body of the `let` binding.

There is an additional problem as well. Consider this term.

```
let a = let b = scanl f z
      in map g b
in map h a
```

In this case, it is possible to fuse the term into

```
let b = scanl f z
in map (h . g) b
```

However, by doing this, we have had to float the binding for `b` outside of its previous scope. This extends the lifetime of `b` potentially affecting the programs memory usage. In general, Accelerate will perform this `let` floating, but only provided it gives additional fusion opportunities.

5.3.2 Fusion through tuples

Going back to our original example:

```
let xs = use (Array ... )
    ys = (map f1 xs, map f2 xs)
in
zipWith g (fst ys) (snd ys)
```

Under the previous system, this would not fuse, despite the tuple only being “used” once, each component of the tuple is only accessed once.

By extending the concept of what it means for a binding to be used, we can enable terms like this to be fused. We simply keep a separate occurrence count for each component of a tuple and embed bindings if no component is used more than once. However, if we were to

embed via conventional substitution, just replacing `ys` with its definition at all use sites, we would end up with

```
zipWith h (fst (generate sh f, generate sh' g)) (snd (generate sh f, generate sh' g))
```

which, while still fusing after further simplification, is much larger than the original term. In general, this can cause an explosion in the size of the term, slowing down subsequent optimisation. Hence, we instead use a specialised form of embedding that immediately simplifies when it encounters tuple projection. This results in

```
zipWith h (generate sh f) (generate sh' g)
```

which can be fused using the rules described above.

Let's now look at a more complicated example:

```
let t = (generate sh f, scanl g z arr, map h arr)
in ( zipWith h (prj 1 t) (prj 3 t)
    , zipWith k (prj 3 t) (prj 2 t) )
```

This brings up the question of what we should do when some components of a tuple are *delayable* (the `generate`) and are only used once, but others are either not delayable (the `scanl`) or occur multiple times (the `map`)? In this case, we inline the first component of `t` but leave the second and 3rd components bound.

```
let t' = (scanl g 0 arr, map h arr)
in ( zipWith h (generate sh f) (prj 2 t')
    , zipWith k (prj 2 t') (prj 1 t) )
```

In general, when we encounter a `let` binding of a tuple like this, we split the tuple into two tuples, one containing all the components that are delayable and only occur once, the other containing all the components that should remain let-bound. The delayable tuple gets inlined using the specialised `inline`; the bound one remains let-bound.

Of course, like above, we also have to consider nested bindings:

```
let a = let b = scanr j 1 arr
        in (map f b, scanl g 0 b)
in zipWith h (fst a) (snd a)
```

This gets transformed into

```
let b = scanr j 1 arr in
let a = scanl g 0 b
in zipWith h (map f b) a
```

Here we are floating `b` out, possibly extending its lifetime. However, by doing this, we can fuse the first component of `a`. As above, we consider that the benefit of fusion outweighs the cost of let floating.

While initially, these fusion rules relating to tuples may seem to only work in specific situations, it is worth noting that they, in fact, stop tuples being a barrier to fusion in all

cases: simply due to the restricted nature of the Accelerate language. The two key properties that make this happen are that:

1. Tuple constructors can only occur in 3 places. In the argument to tuple projections, in let bindings and other tuple constructors. None of the combinators takes tuples as arguments.
2. Unnecessary indirection of the form `let v0 = vn in c[v0]` is eliminated prior to embedding and use counting.

We have already shown how tuple constructors in let bindings can be fused. The case where a tuple constructor is passed to tuple projection is trivial. The only remaining case is tuple constructors in other tuple constructors. For example,

```
let x = (a, (b, c, d))
in ...
```

Supposing that `b` is delayable and is only “used” once by the body, but that `a`, `c` and `d` need to remain bound, we have to be careful. We capture the exact usage pattern of such tuples by maintaining usage information in the following representation:

```
data Uses a where
  UsesArray :: Int                -- How many times the contents of the array is accessed
             → Int                -- How many times the shape of the array is used
             → Uses (Array sh e)
  UsesTuple :: (Uses t_0, ... , Uses t_n)
             → Uses (t_0,...,t_n)
```

As a result, we capture how every component or sub-component of a tuple is accessed.

When we inline with nested tuples, we similarly split each tuple into two smaller tuples. In the above example, we get

```
let x = (a, (c, d))
in ...
```

with `b` now fused into the body.

5.4 Scheduling

While we have described how sequences are implemented in the language and compiler, we are still left with the problem of deciding how many elements of a sequence we should compute at any one time. This problem is not the focus of this work but is still necessary to enable efficient execution. One approach, as described in Madsen et al. [34] is to analyse whole sequence computations and chose a static number based on an analysis of their parallel degree. While this works for regular computations, in the presence of irregularity, statically determining the parallel degree is not always feasible. In addition, where the size of individual elements may

vary considerably, there is, in general, no good fixed static size. Hence, we use a dynamic scheduling approach, continually adjusting the number of elements of the sequence to execute at once. For example, consider the following sequence computation:

```
consume (elements (mapSeq f (mapSeq g xs)))
```

Executing a step of the sequence computation consists of (1) computing some chunk of the input `xs`; (2) applying the lifted version of `g` to the chunk; (3) applying the lifted version of `f` to that result; and (4) store the result. Now, suppose that g^\dagger achieves best performance when processing N elements at a time, but f^\dagger prefers a size of $2N$ for best performance. We could conceivably compute two N -sized chunks with g^\dagger , then combine these into a $2N$ -sized chunk for f^\dagger . However, even though the size of the chunk may be known, the actual size of the data in each chunk (of an irregular computation) is not. For this reason, any sort of multi-rate scheduling requires either considerable copying of intermediates or unbounded buffers. We avoid this issue by choosing to only adjust the chunk size after each complete step of the sequence computation.

We have two competing considerations for determining how many elements we should process in a step of a sequence computation: (1) we want to maximise processor utilisation, to ensure that all processing elements are busy; and (2) minimise the amount of time taken for each element of the sequence, which also acts as an approximate measure to minimising overall system resource requirements, such as memory usage. Sequence computations initially execute a single element of the sequence, then subsequently select a chunk size for the next step based on the following strategy:

- If the overall processor utilisation (time spent executing sequence computations compared to the elapsed wall time) is below a target threshold (80%), increase the chunk size. This is particularly important when the computation is bootstrapping from small chunk sizes and the elements of the sequence are small.
- Once the processor is sufficiently utilised, we continue to increase the chunk size only if it decreases the average time per element. If the time per element instead increases, decrease the chunk size. Otherwise, maintain the chunk size.

In particular, the ability to decrease the chunk size is necessary to handle sequences containing elements that grow in size. Additionally, this acts as a proxy to minimise usage of other system resources, such as memory.

As practical considerations, our implementation uses weighted moving averages of sampled variables, and increases the chunk size at twice the rate that we decrease it, which reduces warm-up time and biases towards ensuring the processors remain saturated.

Chapter 6

Enhancements

In this chapter, we introduce two extensions to the Accelerate language and runtime system. They complement the sequences extension already described, but also improve the overall practical value of the Accelerate framework.

To take advantage of already highly-optimised CUDA libraries from within Accelerate programs, we describe a foreign function interface (FFI) (Section 6.1). It supports both importing foreign functions into Accelerate and exporting Accelerate programs for access via conventional CUDA C/C++ programs. This is the first, to our knowledge, instance of an Embedded Domain Specific Language (EDSL) having an FFI.

To further widen the domain of programs that can practically be executed by Accelerate, we also describe a GPU aware garbage collector that automatically transfers arrays out of GPU memory when not being used (Section 6.2). For certain classes of programs, this reduces the memory they require such that they can be executed with the limited memory GPUs provide.

6.1 Foreign Function Interface

The main advantage of Accelerate has over languages is that programs can be written without requiring the expert knowledge needed to achieve good performance. This is particularly true for programming GPU architectures, as that requires specific knowledge of the architecture, as highlighted in Section 2.3. However, there are existing highly optimised libraries, for example, for high-performance linear algebra and fast Fourier transforms. For Accelerate to be practically useful, we need to provide a means to use those libraries. Moreover, access to native code also provides a developer the opportunity to drop down to low-level C or CUDA C in those parts of an application where the code generated by Accelerate is not sufficiently efficient. We achieve access to CUDA libraries and native CUDA components with the *Accelerate Foreign Function Interface* (or FFI).

The Accelerate FFI works in both directions: (1) it enables calling native CUDA C code

from embedded Accelerate computations and (2) it facilitates calling Accelerate computations from non-Haskell code. Overall, a developer can implement an application in a mixture of Accelerate and other languages in a manner that the source code is portable across multiple Accelerate backends.

Given that Accelerate is embedded in Haskell, it might seem that Haskell’s standard FFI should be sufficient to enable interoperability with foreign code. Unfortunately, this is not the case. With Haskell’s standard FFI, we can call C functions from Haskell host code. However, we want to call functions from within embedded Accelerate code and, in the case of CUDA, pass data structures located in GPU memory directly to native CUDA code and vice versa. The latter is crucial, as transferring data from CPU memory to GPU memory and back is very expensive.

6.1.1 Importing foreign functions

TODO FINAL: *Need to talk about allocation and deallocation more here. Can refer to the next section, but should introduce the basics.*

Calling foreign code in an embedded Accelerate computation requires two steps: (1) the foreign function must be made accessible to the host Haskell program and (2) the foreign function must be lifted into an Accelerate computation to be available to embedded code. For the first step, we use the standard Haskell FFI. The second step requires an extension to Accelerate. We will focus on interfacing with CUDA, as it also requires us to consider data transfer. Interfacing with CPU-based libraries is much the same, but somewhat more straightforward as they work in the same memory space.

As a concrete example, we use the vector dot product of the highly optimised *CUDA Basic Linear Algebra Subprograms (CUBLAS)* library [39]. This CUBLAS function is called `cublasSdot()`; it computes the vector dot product of two arrays of 32-bit floating point values. To access it from Haskell, we use this Haskell FFI import declaration:

```
foreign import ccall "cublas_v2.h_cublasSdot_v2" cublasSdot
  :: Handle
  → Int                                -- Number of array elements
  → DevicePtr Float → Int             -- The two input arrays, and...
  → DevicePtr Float → Int             -- ...element stride
  → DevicePtr Float                   -- Result array
  → IO ()
```

The `Handle` argument is required by the foreign library and created on initialisation. The `DevicePtr` arguments are pointers into GPU memory. As mentioned before, one of the primary aims of the Accelerate FFI is to ensure that we do not unnecessarily transfer data between GPU and CPU memory.

To manage device pointers, the Accelerate FFI provides a GPU memory allocation function `allocateArray` and a function `withDevicePtr` to perform a computation that depends

on a device pointer of an Accelerate array. We can use these functions to invoke `cublasSdot` with GPU-side data:

```
dotp_cublas :: Handle
    → (Vector Float, Vector Float)
    → LLVM PTX (Scalar Float)
dotp_cublas handle (xs, ys) = do
  let n = arraySize (arrayShape xs)  -- number of input elements
  result ← allocateArray Z           -- allocate a new Scalar array
  withDevicePtr xs $ \xptra →       -- get device memory pointers
    withDevicePtr ys $ \yptra →
      withDevicePtr result $ \rptra →
        liftIO $ cublasSdot handle n xptra 1 yptra 1 rptra
  return result
```

The LLVM PTX monad is simply the IO monad enriched with some information used by the PTX backend to manage devices, memory, and caches.

6.1.2 Executing foreign functions with Accelerate

The function `dotp_cublas` invokes native CUDA code in such a manner that it directly uses arrays in GPU memory. This leaves us with two challenges: (1) we need to enable calling functions, such as `dotp_cublas`, in embedded code and (2) we need to account for Accelerate supporting multiple backends, while Accelerate programs should be portable across backends.

We address this challenge by extending the AST with a new node type **Aforeign** representing foreign calls. One instance of an **Aforeign** node encodes the code for one backend, but it also contains a fallback implementation in case a different backend is being used. The AST data constructor is defined as follows:

```
Aforeign :: (Arrays as, Arrays bs, Foreign asm)
    ⇒ asm                (as → bs)  -- The foreign function for a given backend
    → PreAfun      acc    (as → bs)  -- Fallback implementation(s)
    → acc          aenv as           -- Arguments to the function
    → PreOpenAcc   acc aenv bs
```

When the tree walk during code execution encounters an **Aforeign** AST node, it dynamically checks whether it can execute the foreign function. If it cannot, it instead executes the fallback implementation. A fallback implementation might be another **Aforeign** node with native code for a different backend (e.g., for CPU instead of PTX), or it can simply be a vanilla Accelerate implementation of the same functionality that is provided by the foreign code. With a cascade of **Aforeign** nodes, we can provide an optimised native implementation of a function for a range of backends and still maintain a vanilla Accelerate version of the same functionality for execution in the Accelerate interpreter.

The dynamic check for the suitability of a foreign function is facilitated by the class constraint **Foreign f** in the context of **Aforeign**. The class **Foreign** is a subclass of **Typeable** with instances for data types that represent foreign functions for specific backends. For the

CUDA backend, we have the following:

```
class Typeable2 f => Foreign f where ...
instance Foreign ForeignAcc where ...
data ForeignAcc f where
  ForeignAcc :: String
               -> (Stream -> a -> LLVM PTX b)
               -> ForeignAcc (a -> b)
```

`ForeignAcc` wraps calls to foreign CUDA code executed in the LLVM PTX monad. When the CUDA backend encounters an AST node `Aforeign foreignFun alt arg`, it attempts to `cast`¹ the value of `foreignFun` to type `ForeignAcc f`. If that `cast` succeeds, it can unwrap the `CUDAForeignAcc` and invoke the function it contains. Otherwise, it needs to execute the alternative implementation `alt`.

Finally, we can define an embedded vector dot product that uses CUBLAS when possible and, otherwise, falls back to the version defined in Section 2.2:

```
dotp' :: Acc (Vector Float) -> Acc (Vector Float)
       -> Acc (Scalar Float)
dotp' xs ys = Aforeign (CUDAForeignAcc (dotp_cublas handle))
                  (uncurry dotp)
                  (lift (xs, ys))
```

Foreign calls are not curried; hence, they only have got one argument, which is an instance of the class `Arrays` of tuples of Accelerate arrays.

6.1.3 Exporting functions

Accelerate simplifies writing high-performance code as it lessens the need to understand most low-level details of higher level programming. Hence, we would like to use Accelerate from other languages. As with importing foreign code into Accelerate, the foreign export functionality of the standard Haskell FFI is not sufficient for efficiently using Accelerate from languages, such as C. In the following, we describe how the Accelerate FFI supports exporting Accelerate code as standard C calls.

Exporting Accelerate programs

To export Accelerate functions as C functions, we make use of Template Haskell [46]. For example, we might export our Accelerate dot product:

```
dotp :: Acc (Vector Float, Vector Float) -> Acc (Scalar Float)
dotp = uncurry $ \xs ys -> fold (+) 0 (zipWith (*) xs ys)

exportAfun 'dotp "dotp_compile"
```

The function `exportAfun` is defined in Template Haskell and takes the name of an Accelerate

¹See Haskell's `Data.Typeable` library for details on `cast`.

ate function, here `dotp`, as an argument. It generates the necessary export declarations by inspecting the properties of the name it has been passed, such as its type.

Compiling a module that exports Accelerate computations in this way (say, `M.hs`) generates the additional file `M_stub.h` containing the C prototype for the foreign exported function. For the dot product example, this header contains:

```
#include "HsFFI.h"
extern AccProgram dotp_compile(AccContext a1);
```

A C program needs to include this header to call the Accelerate dot product.

Running embedded Accelerate programs

One of the functions to execute an Accelerate computation in Haskell is:

```
run1In :: (Arrays as, Arrays bs)
       => Context -> (Acc as -> Acc bs) -> as -> bs
```

This function comprises two phases: (1) program optimisation and instantiation of skeleton templates of its second argument and (2) execution of the compiled code in a given CUDA context (first argument). The implementation of `run1In` is structured such that, partially applying it to only its first and second argument, yields a new function of type `as -> bs`, where Phase (1) has been executed already — in other words, it precompiles the Accelerate code. Repeated application of this function of type `as -> bs` executes the CUDA code without any of the overheads associated with just-in-time compilation.

The Accelerate export API retains the ability to precompile Accelerate code. The C function provided by `exportAfun` compiles the Accelerate code, returning a reference to the compiled code. Then, in a second step, `runProgram` marshals input arrays, executes the compiled program, and marshals output arrays:

```
OutputArray out;
InputArray  in[2] = { ... };
AccProgram  dotp  = dotp_compile( context );

runProgram( dotp, in, &out );
```

The function `dotp_compile` was generated by `exportAfun 'dotp "dotp_compile"`.

Marshalling input and output arrays

Accelerate uses a non-parametric representation of multi-dimensional arrays: an array of tuples is represented as a tuple of arrays. The type `InputArray` follows this convention. It is a C struct comprising an array of integers indicating the extent of the array in each dimension together with an array of pointers to each underlying GPU array of primitive data.

```
typedef struct { int* shape; void** adata; } InputArray;
```

`OutputArray` includes an extra field, a stable pointer, that maintains a reference to the associated Haskell-side `Array`. This keeps the array from being garbage collected until the `OutputArray` is explicitly released with `freeOutput`.

```
typedef struct { int* shape; void** adata;
                HsStablePtr stable_ptr; } OutputArray;
```

6.2 GPU aware garbage collection

One major design consideration of high-level languages is how to manage memory. Forcing the programmer to manually allocate and free memory hardly satisfies the property of being high-level, but any alternative involves some compromise. The most common solution to this problem is automatic garbage collection where the runtime system associated with a language will periodically scan the heap and determine what allocated memory is reachable and free any allocations that are not. There are many different methods for doing this, but what we propose here is agnostic to which method is used.

The Glasgow Haskell Compiler (GHC) supports garbage collection already. As Accelerate is embedded in Haskell, we can introduce garbage collection to it by hooking into GHC's garbage collector. To do this, however, we have to address the question of when data should be transferred both to and from the GPU.

6.2.1 When to transfer data?

As we have already highlighted, the limited working memory provided by GPUs makes it difficult to write programs that have inputs of sizes exceeding this memory. We have also shown how array sequences can remove this limitation, for certain classes of programs. However, even with sequences, there are still cases where this is a problem. To see why suppose we have program pipeline with 2 stages. The first stage uses sequences to compute values intended for the second stage. The second stage, however, is not able to process its input as a stream but instead requires all of the results before it can do anything. Such a program might have this structure:

```
secondStage . streamOut . firstStage
```

For programs like this, the Accelerate runtime system needs to decide when to copy arrays from the *host* memory to the *device* memory on the GPU, when to copy back and when to free the device memory allocated for them. For our example, the first stage could do as follows for each chunk of the sequence:

1. Compute the chunk from `firstStage`.
2. Copy it back to main memory.

3. Free it from GPU memory.

This has the advantage of minimising space usage on device memory, but it is not always the most optimal thing to do. If `secondStage` is purely CPU based processing, then we have not lost anything by transferring the chunk as soon as we have computed it. If, however, `secondStage` does its own GPU processing, with some or all of its input, then we might not want to transfer immediately. If it needs that data to be in device memory again, then it makes little sense to transfer to host memory and back again.

6.2.2 The previous solution

Prior to our work, Accelerate already had a solution to this problem. The transfer of arrays from device memory were and still are, delayed till an attempt is made to access the values in the array from the CPU. For freeing the array, Accelerate hooks into the Garbage Collector (GC) of the Glasgow Haskell Compiler (GHC). It attaches a finaliser that frees the array in GPU memory when the host-side version of the array is collected.

This solves the problem of premature data transfer. Arrays will not be transferred back to host memory till they are needed there and won't be left in device memory once the host-side array is no longer needed. However, it introduces another problem as device memory usage will grow with host memory usage. Results from `firstStage` aren't freed from device memory till `secondStage` has finished processing them. Naturally, this leads to memory exhaustion and takes away one of the advantages of streaming (at least for examples like this one).

6.2.3 Array eviction

As part of this work, we implemented a more advanced form of GPU aware garbage collection. It works in much the same way as the previous solution but allows for arrays to be copied (*evicted*) back from device memory when memory is exhausted. Specifically, allocation of arrays works as follows:

1. Attempt to allocate space on the device. If that is successful, use it for the device-side representation of the array.
2. If it fails due to an out-of-memory-error, trigger a GHC garbage collection, try to yield to the GC thread, and then try to allocate again.
3. If that fails, find the *least recently used* array in device memory and, if necessary, copy it back to host memory before freeing it. After that, try to allocate again.
4. If that fails, repeat the above step.

For (1), we use CUDA's version of `malloc()` which will fail with an error if it is unable to allocate on device memory. With (2) we rely on `performGC` from `System.Mem`, which causes a GC

to occur in a separate thread. We cannot yield to this thread exclusively, but, in practice, calling `yield` from `Control.Concurrent` achieves that.

To do (3) requires us to keep track of when arrays were used. To do this, we attach a timestamp to each record of a device-side array. In Section 6.1.1 we introduced `withDevicePtr`; internally Accelerate uses that function whenever it needs to access an array on the device. We extended that function to update the timestamp associated with the array.

6.2.4 Discussion

This new method of memory management is not only advantageous to programming with sequences but also more generally. Programs that do not use sequences, but do require inputs larger than available memory also benefit from automatic eviction. That said, an LRU based eviction is not always the most optimal. One example is if a program works by doing multiple passes over the same input, then evicting the least recently used array may not always be the best choice, as it may represent future input. It may be better, in that case, to evict the result, assuming it is not needed for the next pass of the algorithm. We have found that in most cases the LRU strategy is the best, but there is room for further exploration in this area.

Chapter 7

Evaluation

EXAMINER: *The evaluation chapter requires much more work. The first major problem with the evaluation is that it does not present the benchmarks implementation. This thesis makes the case for a more modular approach, however, this is not evaluated. It would be good to show how each benchmark is implemented in Vanilla Accelerate and Accelerate + Sequence. Related to this, I was under the impression that programs with irregular data structure were not currently supported in Vanilla Accelerate. So showing how these are actually Implemented without using Sequences would shed some light on this. This could be completely separated in it's own chapter or perhaps integrated in Chapter 3.*

EXAMINER: *The second major issue I have is that the results show that Accelerate+Sequence is most of the time slower than Vanilla Accelerate. The evaluation does not discuss this in great details and I found this surprising. Chapter 4 mentions that flattening is performed in a way to keep overheads as low as possible, yet it seems like the overhead are quite signijicant. Therefore, I would like to see a more in-depth analysis of the reasons for this issue and if possible, have the implementation fixed to bring the performance on par with Vanilla Accelerate, or if it is not possible, give much more details as to why this is the case and the performance improvement expected when the problem would be fixed.*

EXAMINER: *Finally, the last major issue is that the enhancements described in chapter 6 are not evaluated at all. For instance, there is no evaluation of the foreign function interface (I could easily imagine that marshalling has some costs for instance). Similarly, the GPU aware garbage collection is not evaluated. At the very least, it would be good to report how many times the garbage collector was called during execution or how much total memory was consumed by the benchmarks. Ultimately, the evaluation should demonstrate the need for garbage collection.*

EXAMINER: *On a slightly minor note, the evaluation chapter lacks some details about the experimental methodology. How many times each application was run? Are the number reported median execution time?*

The objective of our work is to improve the expressive power of a flat data-parallel array language with the introduction of irregular structures and a limited form of nested parallelism.

Name	Size	Competitor	Accelerate	Accelerate +Sequences
SMVM (Queen_4147)	330M	62.0 (MKL)	78.1 (126%)	110.7 (179%)
Audio processor	∞	2.1 (C)	0.36 (18%)	0.11 (5.2%)
MD5 Hash	14M	49.9 (Hashcat)	92.0 (184%)	285.5 (572%)
PageRank	130M	5840 (Repa)	2220 (38%)	4240 (73%)

Table 7.1: Benchmark summary. Execution times in milliseconds.

However, to be of practical relevance, this extra expressiveness may only impose a reasonable cost on performance. We demonstrate the performance of this work through a series of benchmarks, which are summarised in Table 7.1.

Benchmarks were conducted using a single Tesla K40c GPU (compute capability 3.5, 15 multiprocessors = 2880 cores at 750MHz, 11GB RAM) backed by two 12-core Xeon E5-2670 CPUs (64-bit, 2.3GHz, 32GB RAM, hyperthreading is enabled) running GNU/Linux (Ubuntu 14.04 LTS). We used GHC-8.0.2, LLVM-3.9.1, and NVCC-8.0.44. Haskell programs are run with RTS options to set thread affinity and match the allocation size to the processor cache size.¹ We use `numactl` to pin threads to the physical CPU cores. Execution times are measured using `criterion`² via linear regression.

7.1 Sparse-Matrix Vector Multiplication (SMVM)

¹+RTS -qa -A30M -Nn

²<http://hackage.haskell.org/package/criterion>

Name	Non-zeros (nnz/row)	Competitor				Accelerate				Accelerate+Sequences			
		N=1	N=12	N=24	GPU	N=1	N=12	N=24	GPU	N=1	N=12	N=24	GPU
pdb1HYS	4.3M (119)	2.09	28.36	47.90	21.61	1.81	13.71	16.36	7.96	0.86	1.50	1.15	0.07
consph	6.0M (72)	2.11	20.72	26.38	15.44	1.82	11.60	9.12	7.79	1.18	1.61	1.26	0.09
cant	4.0M (64)	2.18	30.77	52.21	13.99	2.02	14.52	15.52	7.12	0.82	1.32	0.97	0.06
pwtk	11.6M (53)	2.07	5.41	17.21	13.20	1.97	7.91	4.90	8.36	1.01	2.19	1.74	0.16
rma10	2.4M (50)	2.70	20.48	38.60	13.31	2.09	8.94	10.33	5.68	0.77	0.88	0.65	0.04
shipsec1	7.8M (55)	2.06	13.72	17.66	12.23	2.02	9.61	9.00	7.91	0.90	1.60	1.43	0.11
rail4284	11.3M (10)	1.06	3.68	5.10	7.08	0.83	2.14	3.31	4.58	0.59	1.09	1.18	0.21
TSOPF_FS_b300_c2	8.8M (154)	2.04	5.47	6.57	8.47	1.81	4.32	4.33	5.27	1.21	1.86	1.66	0.14
FullChip	26.6M (9)	0.97	1.96	3.36	0.10	1.19	2.73	2.89	0.32	0.53	1.29	1.28	0.17
dielFilterV2real	48.5M (42)	1.40	4.31	8.06	14.50	1.76	4.21	7.59	7.89	1.34	3.10	3.22	0.54
Flan_1565	117.4M (75)	1.49	4.86	8.40	22.43	1.44	12.77	11.70	9.36	1.35	5.06	5.43	1.20
Queen_4147	329.5M (80)	1.78	9.07	9.71	23.00	1.30	8.09	6.83	9.02	1.24	6.68	5.28	2.36
nlpkkt240	774.5M (28)	1.51	6.90	6.27	16.88	1.39	6.49	8.15	7.02	0.85	4.10	4.37	3.19
HV15R	283.0M (140)	1.50	10.66	5.16	22.45	1.40	13.77	25.08	9.75	1.35	6.24	6.43	2.30

Table 7.2: Overview of sparse matrices tested and results of the benchmark. Measurements in GFLOPS/s (higher is better). Columns labelled N= n are CPU implementations using n threads. Competitor implementations are Intel MKL on the CPU and NVIDIA cuSPARSE on the GPU.

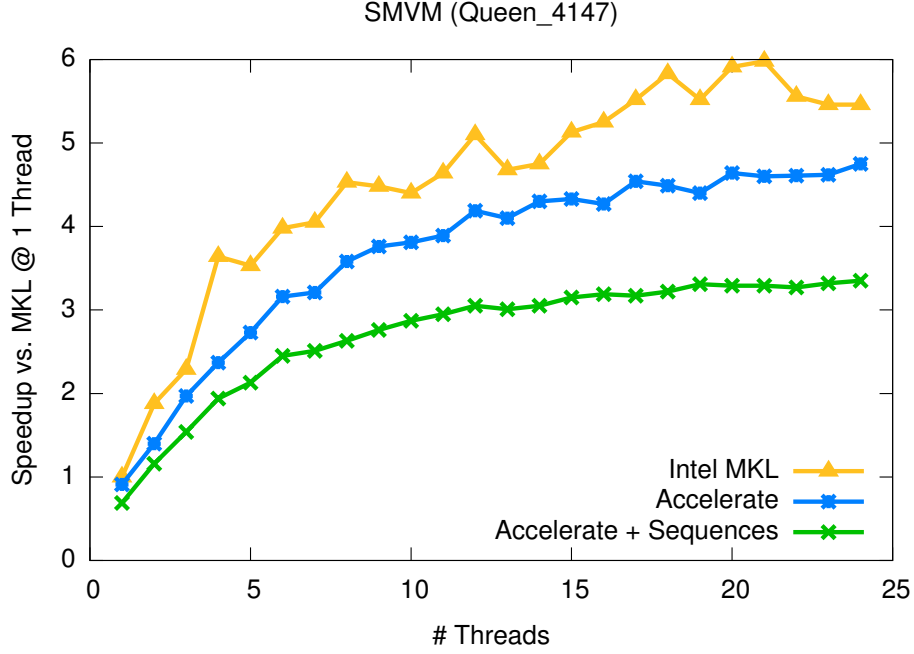


Figure 7.1: SMVM of Queen_4147 dataset

SMVM multiplies a sparse general matrix in compressed row format (CSR) [13] with a dense vector (see Section 3.2.1). Table 7.2 compares Accelerate to the Intel Math Kernel Library³ (MKL v11.3.2, `mk1_cspblas_dcsrcgemv`) on the CPU, and NVIDIA cuSPARSE⁴ (v8.0, `cusparseDcsrmmv`) on the GPU. Test matrices are derived from a variety of application domains [20] and are available online.⁵ Matrices use double precision values and 32-bit integer indices. GPU implementations do not include host/device data transfer time.

In a balanced machine, SMVM should be limited by memory bandwidth. Accelerate is at a disadvantage in this regard, since MKL and cuSPARSE require some pre-processing to construct the segment descriptor, which is not included in their timing results. Our Sequences implementation has some additional bookkeeping work over standard Accelerate in order to track sequence chunks, further reducing overall throughput.

Figure 7.1 shows the strong scaling when computing the sparse-matrix multiply of the Queen_4147 dataset on the CPU. The work in this paper achieves 56% the performance of the highly-tuned reference implementation (or a 30% slowdown compared to the base Accelerate implementation).

³<https://software.intel.com/en-us/intel-mkl>

⁴<https://developer.nvidia.com/cusparse>

⁵http://www.cise.ufl.edu/research/sparse/matrices/list_by_id.html

7.2 Audio processor

EXAMINER: *Section 7.2 mentions that Accelerate+Sequence is able to exploit extra parallelism which is great. However, it is once again not clear why it cannot outperform Accelerate and only a vague explanation is given. In general, the thesis should go more in depth at explaining where these overheads come from and try to fix these.*

The audio processing benchmark tests the algorithm described in Section 3.2.2, computing the `zc_stream` part of the algorithm which processes the input audio data along a sliding window. Since standard Accelerate is limited to flat data parallelism only, it can only take advantage of a single source of parallelism, and applies the `processWindow` operation to a single element of the stream at a time. However, with sequences, we can also take advantage of another source of parallelism and process multiple windowed elements at a time. This is particularly important for the GPU, since we must have enough work to fully saturate all the cores of the device, as well as to amortize the overhead of streaming data between the CPU and GPU. On our test machine the runtime chooses to process 512 elements at a time on the GPU, resulting in a speedup of 3.3× over regular Accelerate.

Figure 7.2 shows the strong scaling performance compared to the implementation in regular Accelerate⁶ on the CPU. As seen in the other benchmark programs, our implementation currently has some overhead compared to regular Accelerate (which has had more time to mature). However, since we can also take advantage of the extra parallelism of processing multiple windowed elements in parallel, we are able to make up the difference in this benchmark.

7.3 MD5 Hash

EXAMINER: *Section 7.3 mentions that pre-loading might help close the gap. How would this be done? Why wasn't it done? If this could be done, what would be the expected benefits? These questions need to be answered in this section.*

The MD5 message-digest algorithm [44] is a cryptographic hash function producing a 128-bit hash value that can be used for cryptographic and data integrity applications. The MD5 benchmark attempts to find the plain text of an unknown hash with a database of known plaintexts. We compare to Hashcat,⁷ the self-proclaimed world's fastest CPU-based password recovery tool. Results from Hashcat are as reported by its inbuilt benchmark mode.

Our sequences based implementation achieves a maximum throughput of 50 million words per second, compared to Hashcat at 287Mword/sec and standard Accelerate at 155Mword/sec. One key difference between our version using sequences and the implementation in standard Accelerate is that we stream in and process small chunks of the input dictionary at a time,

⁶Unfortunately we do not have a parallel reference implementation of the algorithm to compare to.

⁷<https://hashcat.net/hashcat/>

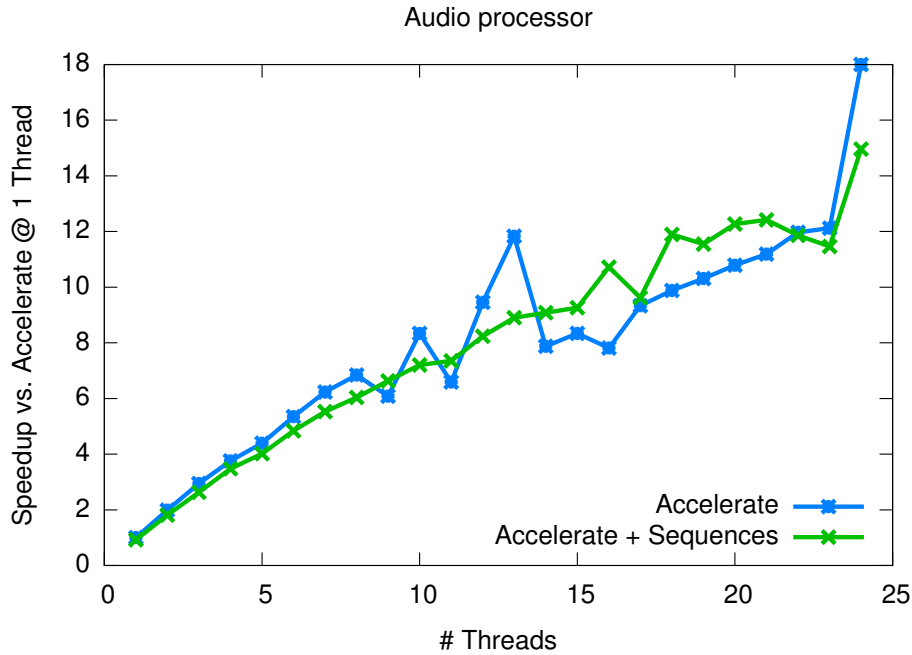


Figure 7.2: Audio processor

rather than loading the entire database into memory at once. However, our runtime system currently does not overlap computation with pre-loading the next chunk of input into memory, which would help close this performance gap. The strong scaling graph is shown in Figure 7.3.

7.4 PageRank

PageRank [41] is a link analysis algorithm which estimates the relative importance of each element of a linked set of documents. As input we use a link graph of Wikipedia (English) consisting of 5.7 million pages and 130 million links. Figure 7.4 compares the performance against an implementation written in Repa [29, 31], a data-parallel array programming language similar to Accelerate. Both Accelerate and Repa represent the link graph as index pairs (Int, Int) . While this representation is not as space efficient as an adjacency list, it is the one typically chosen for this algorithm as it is more suitable for parallelism.

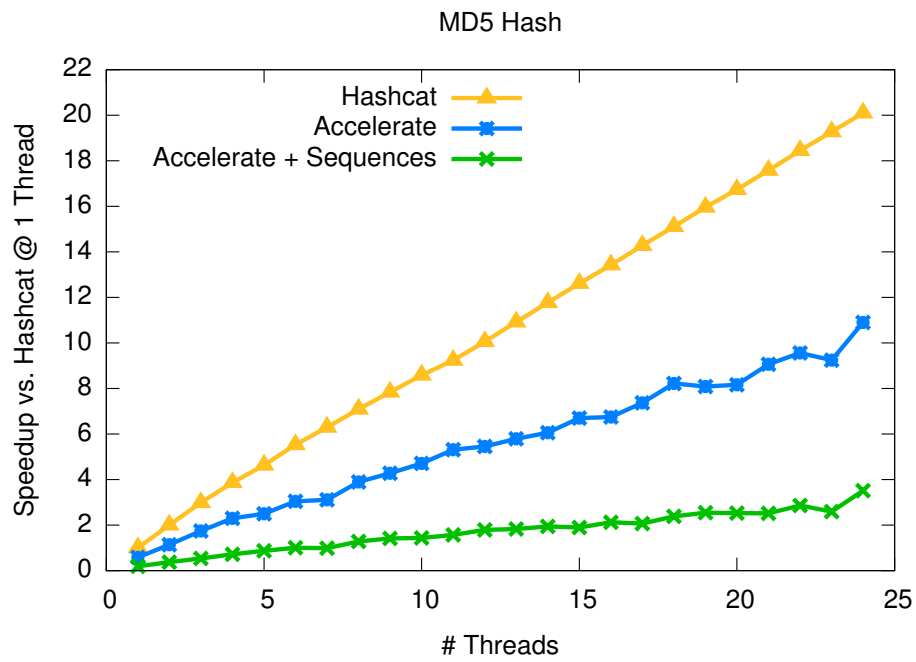


Figure 7.3: MD5 hash recovery

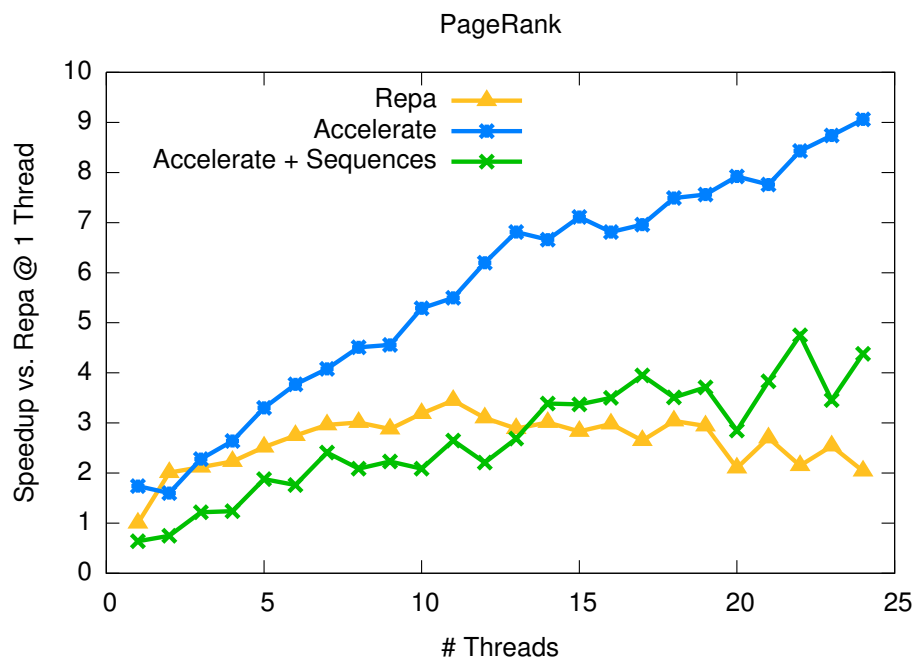


Figure 7.4: PageRank analysis of Wikipedia (English)

Chapter 8

Conclusion

EXAMINER: *The conclusion first stresses that one level of nesting for irregular array offers modularity and reduce memory constrains. However, the thesis currently does not do a good job at demonstrating this modularity aspect and reduction of memory constraints. The experiments also didn't demonstrate that GPU memory can be exhausted and didn't show the effect of the GPU-aware garbage collector and overhead/performance of the foreign function interface.*

In this work, we argued that adding even one level of nesting to array programs offers considerable benefits to modularity. Moreover, if that form of nesting is in sequences, we get the added benefit of reduced memory constraints.

In Chapter 3 we presented the irregular array sequence as an abstract structure and how it adds an additional layer of expressiveness to the array language Accelerate. Even with a relatively small set of operations on sequences, we can express programs that were awkward and non-modular to write before, as well as programs that would otherwise require specific knowledge of the hardware to write without exhausting available GPU memory.

In Chapter 4 we described a novel extension to Blelloch's flattening transform, that we have used to enable irregular arrays sequences, but which can be applied more generally to all nested array structures. This extension can identify computations which are regular and generate code that is optimal for that regularity while generating conventional flattened array code for irregular computations.

In Chapter 6 we described two further novel components of our work. We introduced the Accelerate Foreign Function Interface (FFI), a new way of allowing for an embedded language to support foreign functions. We also introduced a method of garbage collection that allows for GPU memory to be treated like a cache for arrays that are used in GPU computations.

Finally, Chapter 7 evaluated our work with a series of benchmarks.

8.1 Contribution

The work contains 2 major contributions and 2 minor contributions. The two major ones are regularity identifying flattening and irregular array sequences. They are closely related as the former is necessary to implement the latter with expected performance gains. However, the former is also more widely applicable. Regularity identifying flattening applies to any context where efficient nested arrays are needed.

The minor contributions further extend what we can practically express in an array language, by dynamically reducing memory usage through GPU-aware garbage collection and by giving access to highly-optimised 3rd party libraries through a foreign function interface.

8.1.1 Regularity identifying flattening

While program flattening via Blelloch’s flattening transform is well understood, languages that implement it have either assumed everything is irregular [1, 5, 9, 19, 23], or, more recently, that everything is regular [27]. While the former is most general, it misses opportunities to take advantage of the efficiency of regular scheduling. The latter, however, is unable to express irregular computations.

By having a flattening transform that can identify regular (sub)programs and generate flat array code for them, while still generating code suitable for irregular structures otherwise, we get the best of both worlds. This enables us to support not only irregular nested programs and regular nested programs but also programs that combine both.

8.1.2 Irregular array sequences

Given two, almost entirely independent, limitations on flat array programs, in this work we show that both can be overcome with the addition of one feature. Irregular sequences of arrays allow us to alleviate the loss of modularity in flat array programs, without having to solve the challenges of more deeply nested structures. Moreover, they give us portability between GPUs and CPUs, in particular, their different memory limitations. The same program can be executed on both architectures and, even if the GPU does not have sufficient memory to perform the computation in-core, it will have reasonable performance on both.

8.1.3 GPU-aware garbage collection

A feature that is useful for both sequence programs and array programs alike is for the garbage collector to be aware of the GPU and its, typically, much smaller memory than the main memory of the system containing it. While the CPU can swap pages out to disk, prior to our work, there was no system for swapping memory out of the GPU’s memory to host memory. By both hooking into GHC’s garbage collector, and keeping track of what arrays are in GPU memory, we can dynamically manage the memory of the GPU such that

the programmer no longer has to concern themselves with exhausting it. A high-level language should abstract away such details. With our work, a programmer only has to concern themselves with whether individual arrays will fit in GPU memory, not whatever the working memory of their program is, which is typically much harder.

8.1.4 A Foreign Function Interface for an embedded language

Foreign function interfaces have proven to be invaluable to the adoption of high-level languages[10]. Prior to our work, this was an area unexplored in the domain of embedded languages. By introducing an FFI to Accelerate, we made it possible to call out to already highly-optimised low-level libraries from within high-level programs. In addition, different backends can be supported by having different foreign functions for each backend.

8.2 Future work

What we describe in this dissertation could be extended in many different ways. Here, we present a few key research possibilities.

8.2.1 More sequence operations

Even though we can express realistic sequence programs with the operations we have, like with different array combinators there is room for more elaborate sequence combinators. In our work, we wanted to avoid the problem of unbounded buffers, but if such a restriction were relaxed (say by supporting data structures that allow for fast concatenation) a much wider range of operations is possible.

8.2.2 More nested structures

Given the regularity identifying flattening transform in Chapter 4 an obvious piece of future work is to support arbitrarily nested arrays in Accelerate. While we have demonstrated that it is possible to do so in a way that takes advantage of regularity, there are still questions that arise. In particular, scheduling of deeply nested programs on GPUs has proven to be challenging[1]. Additionally, to support the nested structures many algorithms require, sum and recursive data types are needed. This can be achieved, in a limited fashion, with such types in the meta-language, but if that is sufficient or whether Accelerate itself needs them is still an open question.

8.2.3 Multi-GPU parallelism

With sequences, we introduce a new level of parallelism. We've already shown how pipeline parallelism can be used to take advantage of both the CPU and GPU by executing two

different stages at the same time, one on the GPU and one on the CPU. However, doing this on multiple GPUs is yet to be explored. It's already possible, with Accelerate's existing support for multiple GPUs, to manually schedule different computations on different GPUs, but to do so automatically is unexplored. Being able to define a sequence pipeline and have the different stages allocated across the available GPUs in such a way as to maximise efficiency (both in terms of work and in terms of data transfer) would be a powerful language feature.

Bibliography

- [1] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. In *ICFP: International Conference on Functional Programming*. ACM, 2012.
- [2] G. Blelloch and G. Narlikar. A practical comparison of n -body algorithms. In *Parallel Algorithms*, Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.
- [3] G. Blelloch, G. L. Miller, and D. Talmor. Developing a practical projection-based parallel delaunay algorithm. In *Proceedings ACM Symposium on Computational Geometry*, May 1996.
- [4] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Nov. 1990.
- [5] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, Carnegie Mellon University, 1995.
- [6] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. In *The 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 575–585. IEEE, 1988.
- [7] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPOPP ’11, pages 47–56, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941562. URL <http://doi.acm.org/10.1145/1941553.1941562>.
- [8] M. M. Chakravarty and G. Keller. Functional array fusion. In *ACM SIGPLAN Notices*, volume 36, pages 205–216. ACM, 2001.
- [9] M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

- [10] M. M. T. Chakravarty. The Haskell Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report, 2003.
- [11] M. M. T. Chakravarty and G. Keller. More Types for Nested Data Parallel Programming. In *In Proceedings ICFP 2000: International Conference on Functional Programming*, pages 94–105. ACM Press, 2000.
- [12] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP: Declarative Aspects of Multicore Programming*, pages 3–14. ACM, 2011.
- [13] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proc. of Supercomputing*, pages 666–675. IEEE Computer Society Press, 1990.
- [14] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP: Declarative Aspects and Applications of Multicore Programming*. ACM, 2012.
- [15] R. Clifton-Everest, T. L. McDonell, M. M. T. Chakravarty, and G. Keller. Embedding Foreign Code. In *PADL '14: The 16th International Symposium on Practical Aspects of Declarative Languages*, Jan. 2014.
- [16] R. Clifton-Everest, T. L. McDonell, M. M. Chakravarty, and G. Keller. Streaming irregular arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, pages 174–185. ACM, 2017.
- [17] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. Nova: A functional language for data parallelism. Technical report, NVIDIA, 2013.
- [18] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion from lists to streams to nothing at all. In *ICFP: International Conference on Functional Programming*. ACM, 2007.
- [19] J. P. Daniel Palmer and S. Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers 95)*. IEEE, 1995.
- [20] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011. URL <http://www.cise.ufl.edu/research/sparse/matrices>.
- [21] N. G. 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)*, volume 75, pages 381–392. Elsevier, 1972.

- [22] C. Elliott. Functional Images. In *The Fun of Programming*. Palgrave, 2003.
- [23] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status report: The manticore project. In *ML'07: Workshop on ML*, pages 15–24. ACM, 2007. doi: 10.1145/1292535.1292539.
- [24] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA: Functional Programming Languages and Computer Architecture*. ACM, 1993.
- [25] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 16–25, Cape May, NJ, June 1994.
- [26] L. J. Guibas and D. K. Wyatt. Compilation and Delayed Evaluation in APL. In *POPL '78: Principles of Programming Languages*, pages 1–8, 1978.
- [27] T. Henriksen, N. G. Serup, M. Elsmann, F. Henglein, and C. E. Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 556–571. ACM, 2017.
- [28] K. E. Iverson. A programming language. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), pages 345–351, New York, NY, USA, 1962. ACM. doi: 10.1145/1460833.1460872. URL <http://doi.acm.org/10.1145/1460833.1460872>.
- [29] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. L. Peyton Jones, and B. Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *ICFP: International Conference on Functional Programming*, pages 261–272. ACM, 2010. doi: 10.1145/1863543.1863582.
- [30] G. Keller, M. M. Chakravarty, R. Leshchinskiy, B. Lippmeier, and S. Peyton Jones. Vectorisation avoidance. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.
- [31] B. Lippmeier, M. Chakravarty, G. Keller, and S. Peyton Jones. Guiding parallel array fusion with indexed types. In *Haskell Symposium*. ACM, 2012.
- [32] B. Lippmeier, M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and S. Peyton Jones. Work efficient higher-order vectorisation. In *ICFP'12: International Conference on Functional Programming*, pages 259–270. ACM, 2012. doi: 10.1145/2364527.2364564.
- [33] B. Lippmeier, M. M. Chakravarty, G. Keller, and A. Robinson. Data flow fusion with series expressions in haskell. In *ACM SIGPLAN Notices*, volume 48, pages 93–104. ACM, 2013.

- [34] F. M. Madsen, R. Clifton-Everest, M. M. T. Chakravarty, and G. Keller. Functional array streams. In *FHPC'15: Workshop on Functional High-Performance Computing*, pages 23–34. ACM, 2015.
- [35] G. Mainland and G. Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *Haskell Symposium*. ACM, 2010.
- [36] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP: International Conference on Functional Programming*, pages 49–60, Sept. 2013.
- [37] T. L. McDonell, M. M. T. Chakravarty, V. Grover, and R. R. Newton. Type-safe Runtime Code Generation: Accelerate to LLVM. In *Haskell '15: The 8th ACM SIGPLAN Symposium on Haskell*, pages 201–212, New York, New York, USA, 2015. ACM Press.
- [38] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA: Functional Programming and Computer Architecture*, 1991.
- [39] NVIDIA. CUBLAS Library, 2018.
- [40] NVIDIA. CUDA C Programming Guide, 2018.
- [41] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.
- [42] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP'06: International Conference on Functional Programming*, pages 50–61, 2006.
- [43] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Foundations of Software Technology and Theoretical Computer Science*, Oct. 2008.
- [44] R. Rivest. The md5 message-digest algorithm. 1992.
- [45] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Symposium on Graphics Hardware*, pages 97–106. Eurographics Association, 2007.
- [46] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.
- [47] M. Steuwer, T. Rummelg, and C. Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*, pages 74–85. IEEE, 2017.

- [48] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.

Cover art generated by `accelerate-smoothlife` from the `accelerate-examples` package.

Copyright 2013-2018 © Robert Clifton-Everest