Streaming Irregular Arrays

Robert Clifton-Everest
University of New South Wales
Australia
robertce@cse.unsw.edu.au

Trevor L. McDonell
University of New South Wales
Australia
tmcdonell@cse.unsw.edu.au

Manuel M. T. Chakravarty
University of New South Wales
Australia
chak@cse.unsw.edu.au

Gabriele Keller
University of New South Wales
Australia
keller@cse.unsw.edu.au

Abstract

Previous work has demonstrated that it is possible to generate efficient and highly parallel code for multicore CPUs and GPUs from combinator-based array languages for a range of applications. That work, however, has been limited to operating on flat, rectangular structures without any facilities for irregularity or nesting.

In this paper, we show that even a limited form of nesting provides substantial benefits both in terms of the expressiveness of the language (increasing modularity and providing support for simple irregular structures) and the portability of the code (increasing portability across resource-constrained devices, such as GPUs). Specifically, we generalise Blelloch's flattening transformation along two lines: (1) we explicitly distinguish between definitely regular and potentially irregular computations; and (2) we handle multidimensional arrays. We demonstrate the utility of this generalisation by an extension of the embedded array language Accelerate to include irregular streams of multidimensional arrays. We discuss code generation, optimisation, and irregular stream scheduling as well as a range of benchmarks on both multicore CPUs and GPUs.

CCS Concepts • **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → **Parallel programming languages**; **Functional languages**; **Domain specific languages**; • **Theory of computation** → *Vector / streaming algorithms*;

Keywords Data parallelism; Functional programming; Streaming

ACM Reference Format:
Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. 2017. Streaming Irregular Arrays. In *Proceedings of 10th ACM SIGPLAN International Haskell Symposium, Oxford, UK, September 7-8, 2017 (Haskell'17),* 12 pages.
https://doi.org/10.1145/3122955.3122971

1 Introduction

Combinator-based array languages strike a good balance between a high-level declarative notation and good parallel performance [1, 28, 29, 34, 36]. For example, we can write the dot-product of two vectors of floating point numbers xs and ys using the embedded *Accelerate*[1] array language as follows:

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

Aside from the types, this is a natural way to express a dot-product operation in Haskell. It abstracts away details, such as how to effectively use multiple cores, how to generate vector SIMD instructions, or how to make use of a compute accelerator such as a GPU. Indeed, since this high-level description does not mention any of these details, it remains *portable* to all of these targets [28].

This works nicely for code operating on *regular*, *n*-dimensional arrays. Despite repeated attempts [3, 17, 24, 32], it proves difficult to extend it to computations on *irregular* structures (such as sparse matrices or streams of arrays of varying sizes) without compromising the performance of the regular parts of a computation. This is a severe limitation as having nested irregular structures addresses two important concerns: (1) it supports directly representing and computing with sparse data; and (2) it facilitates modularity (as discussed in previous work).

In addition, first-class support for irregularity helps to declaratively decompose computations such that a compiler can generate code for resource-constrained devices. For example, if we have many vectors v_i of which we need to compute the dot-product with w, we could put all the v_i into a matrix V and perform a matrix vector multiplication of V with w. On a resource-constrained device, such as a GPU with limited main memory, this may fail once V becomes too large. The alternative is to use a *stream* of vectors v_i and map the dot-product over that stream.

```
dotp_stream :: Acc (Vector Float)
            → Seq [Vector Float]
            → Seq [Scalar Float]
dotp_stream w vs = mapSeq (dotp w) vs
```

Given this code, it is up to the compiler and runtime system to subdivide the stream into *chunks* of vectors v_i to v_{i+j}, such that simultaneously performing the dot-product for j vectors allows for optimal resource utilisation on the specific hardware used for execution. In contrast to matrix operations, stream operations are chosen such that partitioning them into chunks is always semantically sound (without requiring static analysis or similar).

Despite these advantages, irregular parallelism comes with a serious downside: code generators can usually produce significantly

[1]See http://acceleratehs.org

more efficient code for regular computations, as the additional constraint of regularity allows for far-reaching optimisations, which is especially important for SIMD architectures such as GPUs. Consequently, the additional expressiveness of irregularity is a double-edged sword unless all regular computations can be identified and optimised as usual, even when they occur as part of a larger irregular computation.

The present work tackles this problem in the following manner: (1) we extend Blelloch's flattening transformation [5] to multi-dimensional regular arrays; (2) we apply the extended transformation to a generalisation of Accelerate with streams that allow for one level of extra nesting; and (3) we describe a scheduler that dynamically determines suitable chunk sizes for the execution of stream computations on varying hardware.

Specifically, we make the following contributions:

- We extend previous work on combinator-based streaming programs in Accelerate [25] by generalising to *irregular* sequences, where each piece of the sequence may be a different size, while maintaining execution efficiency for (sub)programs containing only *regular* subdivisions (Section 3).
- We introduce a novel extension to Blelloch's flatting transformation which allows us to distinguish between regular and irregular sequence computations, enabling more efficient execution of regular sequences, while still supporting fully irregular sequences (Section 4).
- We generalise the flattening transformation to multidimensional arrays, and describe its implementation which preserves static type information (Section 4).
- We describe a series of static type-preserving optimisations which improve the performance of both array and sequence computations (Section 6).
- We evaluate this work through a series of example programs and benchmarks (Section 7).

This paper expands on existing work on the embedded language Accelerate [8, 28, 29] and its extension to sequence computations [25]. The source code is available from https://github.com/AccelerateHS/accelerate/tree/feature/sequences.

2 Background: Accelerate

Accelerate is a parallel array language consisting of a carefully selected set of operations on multidimensional arrays, which can be compiled efficiently to bulk-parallel SIMD hardware. Accelerate 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. The collective operations in Accelerate are based on the scan-vector model [9, 37], 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, recall the dot-product program from Section 1. The function `dotp` consumes two one-dimensional arrays (`Vector` or `Array DIM1`) of values, and produces a single result as output (`Scalar` or `Array DIM0`). The type `Acc` indicates that the inputs and outputs to this function are embedded Accelerate computations—they are evaluated in the *object* language of dynamically generated parallel code, rather than the *meta* language, which is vanilla Haskell. 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-vectorized[2] low-level dot-product, while still compiling to efficient code [8, 28, 29].

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)
```

The type classes `Shape` and `Elt` indicate that a type is admissible as an array shape and array element, respectively. We use snoc-lists formed from `Z` and `(:.)` at both the type and value level, to define the dimensionality and extent of an array, or the index of a particular element [8, 21]. Array elements consist of signed and unsigned integers, floating point numbers, `Char`, `Bool`, indexes formed from `Z` and `(:.)`, as well as nested tuples of these.

The type signature for `fold` also shows how Accelerate is stratified into collective *array computations*, represented by terms of the type `Acc t`, and *scalar expressions* `Exp t`. 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 *can not* initiate collective operations. This stratification helps to statically exclude unbound nested, irregular parallelism, as discussed in our previous work [8, 28, 29].

2.1 Limitation #1: Regular, Flat Data-Parallelism

A known limitation of flat data-parallel programming models is that they can inhibit modularity. For example, we might wish to lift our dot-product program to the following (incorrect) matrix-vector product, by applying `dotp` 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       -- get extent of input matrix
  in generate (Z:.m) (λrow → the $ dotp vec (slice mat (row :. All)))
```

Here, we use `generate` to create a one-dimensional vector of `m` elements—the number of rows in the input matrix `mat`—by applying the supplied function at each index in data-parallel. At each index, we extract the appropriate row of the matrix using `slice`,[3] and pass this to our existing `dotp` function together with the input vector. Unfortunately, since both `generate` and `dotp` are data-parallel operations, this definition requires nested data-parallelism and is not permitted in flat data-parallelism.[4, 5] Consequently, we cannot reuse `dotp` to implement matrix-vector multiplication; instead, we have to write it from scratch.

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

[3] The `slice` operation is a generalised array indexing function which is used to cut out *entire dimensions* of an array. In this example, we extract `All` columns at one specific row of the given two-dimensional matrix, resulting in a one-dimensional vector.

[4] Accelerate rejects such programs at metaprogram compilation time.

[5] 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 that in order to create a program which will be accepted by the type checker, we must use the function (`the :: Acc (Scalar a) → Exp a`) to retrieve the result of the `dotp` operation, effectively concealing that `dotp` is a collective array computation in order to match the type expected by `generate`, which is that of scalar expressions; in this instance (`Exp DIM1 → Exp Float`).

Similarly, computations which are more naturally expressed in nested form, such as operations over sparse or irregular data structures, require an unwieldy encoding when restricted to only flat data-parallelism.

2.2 Limitation #2: Memory Usage

Collection-oriented array-based programming languages like Accelerate provide a convenient programming model for SIMD architectures, but are hampered by resource constraints, such as limited main memory. This is particularly problematic for compute accelerators such as GPUs, which have their own —much smaller— high-performance memory areas separate from the host's main memory.[6] As many collective array operations require random access to statically unspecified index ranges, 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. As illustrated by the function `dotp`$_{\texttt{stream}}$ in the introduction, 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 Irregular Sequences

We propose the use of *irregular sequences* (or *streams*) of arrays as a significant step towards overcoming the limitations outlined in the previous section. An irregular sequence of arrays is an ordered collection of arrays, where the extent—the size of an array in each dimension—of each array within a sequence may vary. Sequences of arrays, whose type we denote as `[Array sh e]` in the embedded language, immediately provide an irregular nested data structure, addressing the first limitation. Furthermore, we will see that sequences of arrays in an embedded language naturally give rise to a notion of *sequences of array computations*, providing us with greater freedom of scheduling. This freedom is required to generate code which minimises memory use in resource-constrained target architectures, thereby addressing the second limitation.

In the practical implementation that we use for the benchmarks, we restrict the level of nesting to a depth of one; that is, we support sequences of arrays, but not sequences of sequences of arrays. Previous work [24] 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, we would like to stress that the program transformation at the core of this work, presented in Section 4, is *free of this limitation* and may be used on programs with deeper nesting levels.[7] Nevertheless, we leave further exploration of this generalisation to future work.

Previous work [25] explored the use of *regular* sequences—where the extent of all arrays in the sequence must be the same—in order to address only the second of the discussed two limitations. We substantially improve on this previous work by showing that with irregular sequences, we can (1) simultaneously make progress towards overcoming both limitations; and (2) address the second limitation for a wider variety of applications. In addition to being less expressive, regular sequences come with another significant downside: since Accelerate tracks the *dimensionality* of arrays at the type level, but not their *extent* in each dimension, regularity is a dynamic property of the program. Thus, supporting only regular sequence computations compromises static program safety.

3.1 Sequences

In Accelerate, we distinguish embedded scalar computations and embedded array computations by the type tags `Exp` and `Acc`, where an array computation of type `Acc (Array sh e)` encompasses many data-parallel scalar computations of type `Exp e` to produce an array. Similarly, we use a new tag `Seq` to mark sequence computations; `Seq [arr]` represents a sequence computation comprising many stream-parallel array computations of type `Acc arr`. Specifically, we consider values of type `[arr]` to be sequences (or streams) of arrays, and values of type `Seq seq` to be sequences (or streams) of array *computations*. This is an important distinction. Evaluating a value of type `Seq seq` 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:

```
consume :: 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]`. 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 :: Seq [Array sh e] → Seq (Vector e)
tabulate :: Seq [Array sh e] → Seq (Array (sh:.Int) e)
```

Conversely, we can produce a stream of array computations by a function that is not unlike a one-dimensional sequence variant of `generate` (we encountered the latter in the problematic `mvm`$_{\texttt{ndp}}$):

```
produce :: Exp Int → (Acc (Scalar 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 :: (Acc a → Acc b) → Seq [a] → Seq [b]
```

to be able to elaborate the function `dotp`$_{\texttt{stream}}$ 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))
  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 `elements`, and finally `consume` that vector into a single array computation that yields the result.

Although we are re-using Haskell's list type constructor `[]` for sequences in the embedded language, Accelerate does not actually

[6]The current top-of-the-line NVIDIA Quadro GP100 has 16GB of on-board 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.
[7]Although the implementation of the core transformation supports deeper nesting levels, our surface language can not currently express such programs.

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 :: [Array sh e] → Seq [Array sh e]
```

3.2 Irregularity

The arrays in the sequence used in the implementation of $\texttt{mvm}_{\texttt{seq}}$ are all of the same size; after all, they are the individual rows of a dense matrix. 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 popular *compressed sparse 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):

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

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 (Int, a)
type SparseMatrix a = [SparseVector a]

smvm_seq :: Seq (SparseMatrix Float) → Acc (Vector Float)
         → Acc (Vector Float)
smvm_seq smat vec = consume (elements (mapSeq sdotp smat))
  where sdotp srow = let (ix,vs) = unzip srow
                     in  dotp vs (gather ix vec)
```

When the irregularity is pronounced, we need to be careful with scheduling; otherwise, performance will suffer. We will come back to this issue in later sections.

3.3 Streaming

The `streamIn` function (Section 3) 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 one-at-a-time or in chunks of consecutive elements, then the input list will be demanded lazily as the stream is processed. Similarly, we have the function:

```
streamOut :: Seq [a] → [a]
```

which consumes the results of a sequence computation to produce an incrementally constructed Haskell list of the results of the array computations contained in the sequence. 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 (a) exposes sufficient work to fully utilises the considerable parallelism offered by GPUs; while (b) ensuring that the limits imposed by the relatively small amounts of working memory 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 preprocessing, 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. The pre- and post-processing steps are comparatively cheap and consist of standard matrix operations, so we delegate those to an existing matrix library.[8]

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 this 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.

Although even the core of the algorithm is too long to discuss in detail, we outline the essential structure of the computation and how it relates to Accelerate sequences below:

```
type AudioData = ⟨tuple of arrays⟩

zc_stream :: AudioData → [(Matrix Double, Matrix Double)]
zc_stream audioData =
  streamOut (mapSeq (processWindow audioData) windowIndexes)
  where windowIndexes :: Seq [Scalar Int]
        windowIndexes = produce (sizeOf audioData) id
        processWindow :: AudioData → Scalar Int
                      → Acc (Matrix Double, Matrix Double)
        processWindow = ...
```

The algorithm consists of the steps of pre-processing the audio data, the Accelerate stream computation `zc_stream`, and finally post-processing. The 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`. 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 `produce` and `slice` in $\texttt{mvm}_{\texttt{seq}}$ in Section 3.1.[9]

We use pipeline parallelism to overlap the stream processing performed by `mapSeq (processWindow audioData)` with the post-processing consuming the results of `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 (windows over the audio data) in parallel; and (3) `streamOut` provides pipeline parallelism between `zc_stream` 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 Section 7 that this is helpful in providing performance portability between multicore CPUs and GPUs.

[8] https://hackage.haskell.org/package/hmatrix
[9] As the data in successive steps of the sliding windows strongly overlaps, this setup is more efficient than explicitly creating a stream of windowed data.

4 Implementing Irregular Sequence Computations

To efficiently implement irregular sequence computations, we need to address two requirements: (1) we need to get the irregular data-parallelism into a form where it can be efficiently executed on SIMD hardware; and (2) we need to schedule sequence computations, such that they utilise parallelism as much as possible without exceeding the resource constraints (such as memory limits) of the concrete hardware. The remainder of this section is addressing the first item by building on Blelloch and Sabot [5]'s *flattening transformation*—also known as *vectorisation*—to statically turn irregular, nested data-parallel code into flat data-parallel code operating on regular structures. On the basis of this transformed code, Section 5 outlines the dynamic scheduler addressing the second item.

Blelloch and Sabot [5]'s original version of the flattening transformation was for a first-order language with built-in second-order combinators (called NESL), which makes it a good fit for our embedded array language.[10] Nevertheless, the original transformation has two severe shortcomings, which make the generated code non-competitive: (1) it vectorises code that ought to remain as is for best performance; and (2) it doesn't treat the important special case of operations on regular multi-dimensional arrays specially. We address point (1) by adapting the work on *vectorisation avoidance* [20], and tackle point (2) with a novel generalisation of vectorisation that explicitly distinguishes between regular and irregular computations, to generate efficient code for the former. Moreover, our transformation is, in contrast to previous work, type-preserving.

4.1 Type-safe Flattening with Regularity Preservation and Vectorisation Avoidance

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

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

In addition to the *inner-function* parallelism in `f`, we want to exploit as much of the *intra-function* parallelism of the outer `mapSeq` as possible, to achieve optimal performance on a given machine. That is, we may execute some—but not necessarily *all*—elements of the `mapSeq` in parallel. This helps us to support out-of-core datasets on GPUs, by only loading into device memory those elements of the stream `xs` which are being processed. 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 $f^{\uparrow}$, such that semantically $f^{\uparrow}$ `= mapSeq f`; in other words, $f^{\uparrow}$ can process multiple elements of the stream `xs` at once, in parallel.

Preservation of regularity. Computations on regular, multidimensional arrays are usually much more efficient than performing the same computation 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. For example, consider the following definition of a regular parallel sum across the innermost dimension of a two-dimensional array:

```
sum :: Acc (Array DIM2 Double) → Acc (Array DIM1 Double)
sum = fold (+) 0
```

The definition makes use of the built-in Accelerate combinator:

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

Regarding the dimensionality of the input array, the shape argument of the `Array` type constructor gets stripped of one dimension during the reduction; that is, the type goes from `(sh :. Int)` to `sh`. We refer to `fold` as a shape-polymorphic operation [21].

Now, let us assume the stream `xs` is regular; that is, all arrays in `xs` have the same extent. In that case, we can store multiple elements of the stream in a single array whose dimensionality is one greater: a chunk of arrays from the stream `xs`, where each array has dimensionality n, can be represented by an array of dimensionality $n + 1$. Thus, we can vectorise `sum` by *simply changing its type*:

```
sum↑reg :: Acc (Array DIM3 Double) → Acc (Array DIM2 Double)
sum↑reg = fold (+) 0
```

This is possible since `fold` is shape polymorphic, operating on an array of any dimensionality greater than zero [21].

Matters get more involved in the case of an irregular stream, where the extent of every array in `xs` is not known to be the same. In that case, we can no longer represent chunks of the stream of arrays of dimension n as an array of dimension $n + 1$. Instead, we need to move to a more sophisticated representation that decomposes the chunk of arrays into a *flat data vector* (containing all the array elements) together with a *segment descriptor* which describes how the individual array elements are distributed over the various arrays in the chunk. The type for segment descriptors in Accelerate is `Segments sh` which describes the structure of a chunk of arrays of shape `sh`. (Its concrete representation is orthogonal to the work presented here, so we will keep it abstract.)

On that basis, the definition of `sum` vectorised for processing of an irregular stream is:

```
sum↑irreg :: Acc (Segments DIM2, Vector Double)
          → Acc (Segments DIM1, Vector Double)
sum↑irreg (segs, vals) =
  foldseg (+) 0 vals segs
```

where we use an irregular variant of `fold` based on segment descriptors which has the type:

```
foldseg :: (Elt a, Shape sh)
        ⇒ (Exp a → Exp a → Exp a)
        → Exp a
        → Acc Vector a
        → Acc (Segments (sh :. Int))
        → Acc (Segments sh, Vector a)
```

The implementation of $\texttt{fold}_{\texttt{seg}}$ is significantly more expensive than that of `fold`. Moreover, maintaining and passing around of the segment descriptors is an additional overhead. Clearly, we want to incur this only when necessary; when the processed stream is actually irregular.

[10] While the host language is higher-order, the *embedded* language, Accelerate, is not.

Vectorisation avoidance. Even in the case of an entirely regular computation, we need to be careful to avoid inefficiencies due to vectorisation. As a simple example, consider this scalar function:

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

Vectorisation of this code takes each subcomputation from a scalar operation to a vector-valued function, and replicates constants according to the size of the argument vectors:

```
average^↑ :: Acc (Vector Float) → Acc (Vector Float)
          → Acc (Vector Float)
average^↑ xs ys =
  zipWith (/) (zipWith (+) xs ys) (replicate (Z :. length xs) 2)
```

This code is not efficient due to the excessive array traversals and superfluous intermediate structures. In this simple example, fusion optimisations can improve the code, but more generally we will arrive at better code when vectorisation directly *avoids* vectorising purely scalar subexpressions, generating the following instead:

```
average^↑_avoid :: Acc (Vector Float) → Acc (Vector Float)
                → Acc (Vector Float)
average^↑_avoid xs ys = zipWith (λx y → (x + y) / 2) xs ys
```

The concrete transformation presented in the rest of this section avoids vectorisation of purely scalar subexpressions, and preserves regularity when vectorisation is over a regular domain.

4.2 Lifted Type Relation

Vectorisation is a type-directed transformation. Hence, we first introduce type transformations $\mathcal{N}$ and $\mathcal{V}$ before turning to the term transformation $\mathcal{L}$. We denote both type transformations in relational form as regular versus irregular vectorisation contexts give us a choice between different representations. As the concrete notation, we use Haskell's notation for Generalised Algebraic Data Types (GADTs) [33], as this coincides with our concrete implementation in Accelerate.

Normalisation. The first type transformation, $\mathcal{N}$ `t` $\mathtt{t}_{norm}$, computes an array normal form $\mathtt{t}_{norm}$ of an Accelerate type `t`. Under this normalisation, a scalar value of type `e` is wrapped in a singleton array of type `Array Z e`, and a regular n-dimensional array of regular m-dimensional arrays becomes an $n+m$-dimensional array.

```
data 𝒩 t t' where
  Scalar :: IsScalar e
         ⇒ 𝒩 e (Array Z e)
  Nest   :: 𝒩 e (Array sh e')
         → 𝒩 (Array Z e) (Array sh e')
  Higher :: 𝒩 (Array sh e) (Array sh' e')
         → 𝒩 (Array (sh:.Int) e) (Array (sh':.Int) e')
```

For example, a vector of vectors of `Floats` has type:

```
Array DIM1 (Array DIM1 Float)
```

so would be normalised to a two-dimensional array of `Floats`, and is witnessed by:

```
Higher (Nest (Higher (Nest Scalar))) ::
  𝒩 (Array (Z :. Int) (Array (Z :. Int) Float))  — original type
     (Array (Z :. Int :. Int)  Float)             — normalised type
```

Vectorisation. The second type relation, $\mathcal{V}$ `t` $\mathtt{t}_{vect}$, takes an Accelerate type to its vectorised form $\mathtt{t}_{vect}$ by first normalising `t` by way of $\mathcal{N}$ `t` $\mathtt{t}_{norm}$, then transforming the normalised $\mathtt{t}_{norm}$. This last step is ambiguous, and our reason for expressing the transformation relationally. If the enclosing data-parallel context is regular, we simply increase the dimensionality of $\mathtt{t}_{norm}$ by one. However, if the context is irregular, we need to introduce a segment descriptor as discussed in the previous subsection. These two cases are covered by the alternatives `Regular` and `Irregular`:

```
data 𝒱 t t' where
  Avoid     :: 𝒱 t t                                  — avoid vectorisation
  Regular   :: 𝒩 t (Array sh e)                       — regular context
            → 𝒱 t (Array (sh:.Int) e)
  Irregular :: 𝒩 t (Array sh e)                       — irregular context
            → 𝒱 t (Segments sh, Vector e)
  Tuple     :: (𝒱 t_1 t_1', 𝒱 t_2 t_2', ..., 𝒱 t_n t_n')
            → 𝒱 (t_1, t_2, ..., t_n) (t_1', t_2', ..., t_n')
```

The final case `Tuple` allows us to vectorise each component of a tuple of arrays independently. This is crucial, and allows us to mix regular, irregular, and vectorisation avoiding computations.

Vectorisation avoidance is covered by the `Avoid` alternative, where we keep the type the same. Note that we do not need to normalise the type `t` as it is always a scalar type; otherwise, we wouldn't need vectorisation avoidance at all. To determine whether all components of a compound type use vectorisation avoidance, we can produce a type witness for the equality between the original and transformed types:

```
isAvoid :: 𝒱 t t' → Maybe (t :~: t')
isAvoid Avoid = Just Refl
isAvoid (Tuple (r_1,...,r_n))
  | Just Refl ← isAvoid r_1
  ...
  , Just Refl ← isAvoid r_n
  = Just Refl
isAvoid _ = Nothing
```

4.3 The Lifting Transformation

To formalise the lifting transformation, we use a core language with fewer parallel operations than Accelerate, but which is also more general in that it supports arbitrarily nested parallel arrays. The latter not only simplifies explanation but shows that this transformation truly is an extension of similar vectorisation approaches.

The grammar of our language is shown in Figure 1. The primitive operations of the language are a selection of the usual arithmetic operations on expressions and `indexInit :: sh:.Int → sh`, which strips the *outermost* nesting level from a shape descriptor. Expressions can be variables, constants, shape descriptors, tuples,[11] array computations indexed by a shape descriptor, applications of

variables	v	::=	$v_0 \mid v_1 \mid \ldots$
literals	l	::=	$0 \mid 1 \mid 2 \mid \ldots$
constants	c	::=	$l \mid [c, c, \ldots]$
shapes	sh	::=	$\mathrm{Z} \mid sh\ :.\ e$
tuples	t	::=	$(e_0, \ldots, e_n)$
primitive	p	::=	$(+) \mid (*) \mid (-) \mid \mathrm{indexInit} \ldots$
expressions	e	::=	$v \mid c \mid sh \mid t \mid e\ !\ sh \mid p\ t$
		$\mid$	$(\lambda v_0 .\, e_1)\ e_2$
		$\mid$	$\mathrm{prj}\ l\ e$
		$\mid$	$\mathrm{extent}\ e$
		$\mid$	$\mathrm{generate}\ sh\ (\lambda v_0 .\, e)$
		$\mid$	$\mathrm{fold}\ (\lambda v_1\, v_0 .\, e_1)\ e_2\ e_3$

Figure 1. The nested data-parallel core language

[11]We use ... as an informal way of generalising over n-ary tuples. In practice our encoding uses representation types for this purpose, but we elide those details as they are orthogonal to what we present here.

primitives or lambda abstractions, and projections on tuples. The function `extent :: Array sh e → sh` returns the shape of an array, while `generate` creates an array of given extent with all elements initialised by a function mapping array indices to values. More specialised functions, such as `map`, can be expressed in terms of `generate`. Finally, we have `fold`, which is parametrised by a scalar function and a scalar initial value.

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

```
ℒ⟦.⟧ :: Expr Γ t → Env Γ Γ' → (∃ t'. (𝒱 t t', Expr Γ' t'))
```

Here, the first argument, of type `Expr Γ t`, is the typed abstract syntax (AST) of the core language term that is to be vectorised, where `Γ` is a type-level list of the free variables in the term and `t` is the term's type.[12] The second argument, of type `Env Γ Γ'`, is an environment providing a symbolic valuation for the free variables captured in `Γ`. It also relates the types of the free variables in `Γ` to their vectorised form `Γ'`. Finally, the result combines the witness for the vectorised result type `t'` with the lifted term `Expr Γ' t'`, whose type parameters have been vectorised to match, establishing type-preservation of the transformation.

The structure of the environment `Env Γ Γ'` is somewhat more involved than usual, due to special requirements during vectorisation. Recall the fully vectorised version of the function `average`, which we called `average`$^{\uparrow}$ (Section 4.1). It contains the subexpression `replicate (Z :. length xs) 2` to produce a vector of the constant scalar 2 whose length matches that of the argument vector `xs`; this constitutes the context for 2's vectorisation. We need to similarly replicate the value of any term that is constant within a vectorisation context, regardless of whether it is a scalar or an array, unless vectorisation avoidance indicates that this is not necessary. In addition to the usual purpose of tracking the types of free variables, our environment tracks both the original and vectorised types of free variables (the `(:)` alternative). Furthermore, we track both regular (`(:`$_R$`)`) and irregular (`(:`$_{Ir}$`)`) contexts. The last two maintain the symbolic form of a term that represents the vectorisation context, which we can use for `replicate`.

```
data Env Γ Γ' where
  []     :: Env [] []                    — empty environment
  (:)    :: 𝒱 t t'                        — standard free variable
         → Env Γ Γ'
         → Env (t : Γ) (t' : Γ')
  (:_R)  :: Expr Γ' sh                   — regular vectorisation context
         → Env Γ Γ'
         → Env Γ Γ'
  (:_Ir) :: Expr Γ' (Segments sh)        — irregular vectorisation context
         → Env Γ Γ'
         → Env Γ Γ'
```

The use of this environment structure can be seen in the definition of the auxiliary function `var`, defined in Figure 2, 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; that is, 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.

[12]Just like the full implementation in Accelerate, we use a typed representation of the core language to define a type-preserving transformation; see McDonell et al. [28] for details on Accelerate's design in this respect.

```
— Replicate a regular variable
var :: Env Γ Γ' → Var Γ t → (∃ t. (𝒱 t t', Expr Γ' t))
var (r : _)    v_0      = (r, v_0)                     — lookup variable...
var (_ : env) v_{n+1}   | (r, e) ← var env v_n          — ... in environment
                        = (r, weaken e)
var (sh :_R env) v      | (r, e) ← var env v           — regular nesting
                        = (r, replicate_R r sh e)
var (segs :_Ir env) v   | (r, e) ← var env v           — irregular nesting
                        = (r, replicate_Ir r segs e)

— Add new fresh variable
weaken :: Expr Γ t → Expr (a : Γ) t

— If a lifted expression, replicate each subarray by the size of the given shape
replicate_R   :: 𝒱 t t' → Expr Γ sh → Expr Γ t' → Expr Γ t'

— If a lifted expression, replicate each subarray by the size of the corresponding segment
replicate_Ir  :: 𝒱 t t' → Expr Γ sh → Expr Γ t' → Expr Γ t'

— Get the shapes from segment descriptors
shapes :: Segments sh → Vector sh

($_R)   :: Expr Γ (Vector a → Array sh b)
        → Expr Γ (Array (sh:.Int) a)
        → Expr Γ (Array (sh:.Int) b)
($_Ir)  :: Expr Γ (Vector a → (Segments sh, Vector b))
        → Expr Γ (Segments (sh:.Int), Vector a)
        → Expr Γ (Segments (sh:.Int), Vector b)

enum_R  :: Expr Γ sh → Expr Γ (Array (sh:.Int) sh)
enum_Ir :: Expr Γ (Segments sh) → Expr Γ (Segments sh, Vector sh)

— A generalised zip for lifted scalars e_0 ... e_n
zip_S   :: (𝒱 e_0 a_1,...,𝒱 e_n a_n)
        → (e_0,..,e_n)
        → Vector (e_0,...,e_n)
```

Figure 2. Auxiliary functions for the lifting transformation

The main rules of the vectorisation transformation are given in Figure 3. When it encounters a constant value, vectorisation returns it as it is, paired with the `Avoid` constructor to signal that it has not been vectorised yet. When vectorising a variable, the transformation refers to the auxiliary function `var` discussed previously. The application rule defines how the context of a variable is determined by the argument it is bound to: the transformation first vectorises the argument, inserting the resulting $\mathcal{V}$-term in the environment of the variable.

The rules for `extent` returns the original expression, if vectorisation of its argument expression can be avoided. If the argument expression vectorises in a regular context, `extend` gets vectorised to the shape of the vectorised argument, stripped of the outermost dimension. Otherwise the context is irregular, and the application of `extend` is vectorised to be the vector of shapes stored in the segment descriptor of the vectorised argument.

The rules for array indexing `(!)` are defined in terms of $\mathcal{L}_F$, which enforces vectorisation by ignoring `Avoid` in the vectorisation of the term e, the first argument of the indexing operation:

```
ℒ_F ⟦e⟧ env
  | (Avoid, e') ← ℒ⟦e⟧ env = (Regular, replicate envSize e')
  | (r, e')     ← ℒ⟦e⟧ env = (r, e')

envSize :: Env Γ Γ' → Expr Γ' Int
envSize (sh :_R  _)   = indexLast sh
envSize (segs :_Ir _) = length segs
envSize (_ : env)     = envSize env
```

```
ℒ⟦c⟧ _   = (Avoid, c)
ℒ⟦v⟧ env = var env v
ℒ⟦extent e⟧
  | (Avoid, e')       ← ℒ⟦e⟧ env
  = (Avoid, extent e')
  | (Regular _, e')   ← ℒ⟦e⟧ env
  = (Avoid, indexInit (extent e'))
  | (Irregular r, e') ← ℒ⟦e⟧ env
  = (Regular r, shapes (prj 0 e'))
ℒ⟦e ! sh⟧
  | (Avoid, sh') ← ℒ⟦sh⟧ env
  , (Avoid, e')  ← ℒ⟦e⟧ env
  = (Avoid, e' ! sh')
  | (Regular Scalar, sh') ← ℒ_F⟦sh⟧ env
  , (Regular r, e')       ← ℒ_F⟦e⟧ env
  = (Regular r, e' !_R sh')
  | (Regular Scalar, sh') ← ℒ_F⟦sh⟧ env
  , (Irregular r, e')     ← ℒ_F⟦e⟧ env
  = (Irregular r, e' !_Ir sh')
ℒ⟦prj l e⟧ env
  | (Tuple (..,r_l,..), e') ← ℒ⟦e⟧ env
  = (r_l, prj l e')
```

```
ℒ⟦(e_0,...,e_n)⟧ env
  | (r_0, e_0') ← ℒ⟦e_1⟧ env
  ...
  , (r_n, e_n') ← ℒ⟦e_n⟧ env
  = (Tuple (r_1,...,r_n), (e_0',...,e_n'))
ℒ⟦p t⟧
  | (Tuple ts, t') ← ℒ⟦t⟧ env
  = ( Regular Scalar
    , map (λv_0. p v_0) (zip_S ts e'))
ℒ⟦generate sh (λv_0. e)⟧ env
  | (Avoid, sh') ← ℒ⟦sh⟧ env
  , (r, e')      ← ℒ⟦e⟧ (Avoid : env)
  , Just Refl    ← isAvoid r
  = (r, generate sh' (λv_0. e'))
  | (Avoid, sh')    ← ℒ⟦sh⟧ env
  , (Regular r, e') ← ℒ⟦e⟧
        (Regular Scalar : sh' :_R env)
  = (Regular (Nest r), (λv_0. e') $_R enum_R sh')
  | (Regular Scalar, sh') ← ℒ⟦sh⟧ env
  , (Irregular r, e')     ← ℒ_F⟦e⟧
        (Regular Scalar : makeSegs sh' :_Ir env)
  = (Irregular (Nest r), (λv_0. e') $_Ir enum_Ir sh')
```

```
ℒ⟦(λv_0. e_1) e_2⟧ env
  | (r_1, e_2') ← ℒ⟦e_2⟧ env
  , (r_2, e_1') ← ℒ⟦e_1⟧ (r_1 : env)
  = (r_2, (λv_0. e_1') e_2')
ℒ⟦fold (λv_1 v_0. e_1) e_2 e_3⟧ env
  | (Avoid, e_2') ← ℒ⟦e_2⟧ env
  , (Avoid, e_1') ← ℒ⟦e_1⟧ (Avoid : env)
  , (Avoid, e_3') ← ℒ⟦e_3⟧ env
  = (Avoid, fold (λv_1 v_0. e_1') e_2' e_3')
  | (Avoid, e_2') ← ℒ⟦e_2⟧ env
  , (Avoid, e_1') ← ℒ⟦e_1⟧ (Avoid : env)
  , (Regular (Higher r), e_3') ← ℒ⟦e_3⟧ env
  = (Regular r, fold (λv_1 v_0. e_1') e_2' e_3')
  | (Avoid, e_2') ← ℒ⟦e_2⟧ env
  , (Avoid, e_1') ← ℒ⟦e_1⟧ (Avoid : env)
  , (Irregular (Higher r), e_3') ← ℒ⟦e_3⟧ env
  = ( Irregular r
    , fold_seg (λv_1 v_0. e_1') e_2' e_3')
```

Figure 3. The lifting transformation

The generate rule is one of the more interesting ones. In order to lift the body expression of the argument function, the transformation essentially implements a backtracking search. First, it checks if vectorisation can be avoid for this expression, which is the optimal case. If vectorisation is required, we first check if it can be lifted into a regular computation, otherwise the expression is lifted to an irregular computation. This backtracking in theory can lead to exponential work complexity, but in practice there is no deep nesting of generate-expressions. Keller et al. [20], 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; we could follow a similar approach. With respect to the definition of Env Γ Γ', it is worth noting how the recursive use of lifting in generate extends the environment with regular and irregular vectorisation contexts, which subsequently lead to the generation of the appropriate uses of $\texttt{replicate}_R$ and $\texttt{replicate}_{Ir}$ by var, as discussed above.

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, so the transform can rely on the fact that vectorising these two arguments is not necessary, and only has to check the result of vectorising the third argument. As we saw in the sum example (Section 4.1), in the first two cases the vectorised fold is a plain fold with the dimensionality increased by one, while in the irregular case it is transformed into the segmented reduction operator $\texttt{fold}_{\text{seg}}$.

5 Scheduling

We are still left with the problem of deciding how many elements of a sequence we should compute at any one time. Madsen et al. [25] 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, where the size of individual elements may vary greatly, there is in general no good fixed static size. Hence, we use a dynamic scheduling approach, constantly 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) apply the lifted version of g to the chunk; (3) apply the lifted version of f to that result; and (4) store the result. Now, suppose that $g^{\uparrow}$ achieves best performance when processing N elements at a time, but $f^{\uparrow}$ prefers a size of $2N$ for best performance. We could conceivably compute two N-sized chunks with $g^{\uparrow}$, then combine these into a $2N$-sized chunk for $f^{\uparrow}$. 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.
- 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.

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 remains saturated.

6 Optimisation

6.1 Sequence Fusion

Composing functions to build programs has advantages for clarity and modularity. However, the naïve compilation of such programs quickly leads to both code explosion and use of intermediate data structures, hurting performance. Fusion, or deforestation, attempts to remove the overhead of programming in this style by combining adjacent transformations on data structures to remove intermediate structures, and has been studied extensively [12, 23, 29, 39].

Similarly, it is important that we remove intermediate structures from a sequence computation. Recall that a sequence computation of type `Seq [Array sh e]` consists of many stream parallel array computations of type `Acc (Array sh e)`. In the same way, sequence fusion consists of two components. First, we use a variant of stream fusion [12] in order to combine the sequence computations.[13] This exposes the array computations of the sequences to each other, allowing us to then apply an improved version of the Accelerate array fusion system [29] to further combine the operations, which we describe next.

6.2 Improved Array Fusion

The core idea underlying the existing Accelerate array fusion system [29] is well known: simply represent an array by its size and a function mapping array indices to their corresponding values. Fusion then becomes an automatic property of the data representation. This method has been used successfully to optimise purely functional array programs in Repa [21, 23], although the idea of representing arrays as functions is well known [10, 15, 18].

However, a straightforward implementation of this approach results in a loss of *sharing*, which was a problem in early versions of Repa [23]. For example, consider the following program:[14]

```
let xs  = use (Array ...)
    ys₁ = map f xs
in zipWith g ys₁ ys₁
```

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. In the standard Accelerate fusion system, the solution to this problem is to not fuse terms, such as `ys`, whose results are used more than once.

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 ...)
    ys₂ = (map f₁ xs, map f₂ xs)
in zipWith g₂ (fst ys₂) (snd ys₂)
```

Although the term $\texttt{ys}_2$ 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, since we represent irregular sequences as a pair consisting of the segment descriptor together with a vector of the array values. Thus, in the standard Accelerate system, irregular sequences would never fuse, severely impacting performance. We extend the Accelerate fusion system in order to fuse the individual components of a tuple independently, which improves the performance of both our sequence computations as well as regular Accelerate programs.

The solution is to be more precise with what constitutes the use of a variable in a term, for variables binding tuples. Rather than counting only how many times the variable occurs in a term, we must keep track of how often each component of the tuple that variable references is accessed. In the above example, we would then see that the first and second components of the binding $\texttt{ys}_2$ are each only accessed once, thus fulfilling our criteria that the terms should be fused.

At a more interesting example, consider the following where only some of the components of the bound tuple expression should be fused into the body expression:

```
let xs  = use (Array ...)
    ys₃ = ( map f₁ xs, ( generate sh f₃, scanl (+) 0 xs ) )
in ( zipWith g₂ (fst ys₃) (fst (snd ys₃) )
   , zipWith g₃ (fst ys₃) (snd (snd ys₃) ) )
```

Here, the result of `map` is used twice, so is not considered for fusion, while the `generate` and `scanl` results are each used only once. As `scanl` is not an element-wise operation it can not be fused into its use site [29], leaving us with only `generate`. In our new system, for each sub-component of a tuple which should be fused, we split it out of the original expression into its own let-binding, which then allows the existing fusion system to handle it.

7 Evaluation

The objective of this 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. However, we would also like that this extra expressiveness imposes only a reasonable cost on performance. We evaluate the work through a series of benchmarks, which are summarised in Table 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 E5-2670v3 CPUs (64-bit, 2.3GHz, 32GB RAM, hyperthreading is enabled) running GNU/Linux (Ubuntu 14.04 LTS). We used GHC-8.0.2, LLVM-4.0.0, 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.[15] We use `numactl` to pin threads to the physical CPU cores. Times are measured using criterion[16] via linear regression.

7.1 Sparse-Matrix Vector Multiplication (SMVM)

SMVM multiplies a sparse general matrix in CSR format [9] with a dense vector. Table 2 compares Accelerate to the Intel Math Kernel Library[17] (MKL v11.3.2, `mkl_cspblas_dcsrgemv`) on the CPU, and NVIDIA cuSPARSE[18] (v8.0, `cusparseDcsrmv`) on the GPU. Test matrices are derived from a variety of application domains [14] and are available online.[19] Matrices use double precision values and 32-bit integer indices. GPU implementations do not include host/device data transfer time.

[13] One of the complexities of general fusion transformations, including stream fusion, is needing handle filtering operations, where the size of the result structure depends on the *values* of the input structure, as well as its size. Our sequences do not support dropping (or skipping) elements of the stream, so we do not need to consider it here.
[14] The operator use lifts a regular Haskell array into an embedded language array.

[15] `+RTS -qa -A30M -N`*n*
[16] http://hackage.haskell.org/package/criterion
[17] https://software.intel.com/en-us/intel-mkl
[18] https://developer.nvidia.com/cusparse
[19] http://www.cise.ufl.edu/research/sparse/matrices/list_by_id.html

Name	Input Size	Competitor		Accelerate		Accelerate +Sequences	
SMVM (Queen_4147)	330M	62.0	(MKL)	74.3	(120%)	80.1	(129%)
Audio processor	(continuous)	2.10	(C)	4.29	(204%)	3.69	(176%)
MD5 Hash	14M	49.9	(Hashcat)	92.0	(184%)	435.0	(872%)
PageRank	130M	5840	(Repa)	1767	(30%)	3059	(52%)

Table 1. Benchmark summary. Execution times in milliseconds.

	Non-zeros	Competitor				Accelerate				Accelerate+Sequences			
Name	(nnz/row)	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	1.84	13.31	15.72	7.96	1.66	7.89	7.31	0.96
consph	6.0M (72)	2.11	20.72	26.38	15.44	1.93	11.84	7.37	7.79	1.60	6.75	4.34	1.18
cant	4.0M (64)	2.18	30.77	52.21	13.99	1.98	14.43	16.29	7.12	1.61	7.05	6.46	0.80
pwtk	11.6M (53)	2.07	5.41	17.21	13.20	1.91	9.33	4.68	8.36	1.61	6.51	3.77	2.06
rma10	2.4M (50)	2.70	20.48	38.60	13.31	2.06	8.84	10.10	5.68	1.63	4.75	4.49	0.54
shipsec1	7.8M (55)	2.06	13.72	17.66	12.23	1.93	10.42	5.09	7.91	1.21	6.78	3.76	1.52
rail4284	11.3M (10)	1.06	3.68	5.10	7.08	1.01	1.90	3.30	4.58	0.72	1.58	2.83	1.63
TSOPF_FS_b300_c2	8.8M (154)	2.04	5.47	6.57	8.47	1.70	4.28	2.88	5.27	1.64	3.78	2.55	1.46
FullChip	26.6M (9)	0.97	1.96	3.36	0.10	1.18	2.58	1.90	0.32	0.65	1.70	1.43	0.26
dielFilterV2real	48.5M (42)	1.40	4.31	8.06	14.50	1.75	8.37	4.50	7.89	1.45	6.22	3.97	4.26
Flan_1565	117.4M (75)	1.49	4.86	8.40	22.43	1.97	11.79	5.00	9.36	1.69	9.27	4.62	5.69
Queen_4147	329.5M (80)	1.78	9.07	9.71	23.00	1.45	8.20	5.85	9.02	1.31	8.56	5.79	6.35
nlpkkt240	774.5M (28)	1.51	6.90	6.27	16.88	1.40	6.28	7.00	7.02	1.15	5.82	7.19	5.17
HV15R	283.0M (140)	1.50	10.66	10.58	22.45	1.38	6.85	10.93	9.75	1.46	11.12	13.30	6.85

Table 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.

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, particularly on small matrices.

Figure 4 shows the strong scaling when computing the sparse-matrix multiply of the Queen_4147 dataset on the CPU. The work in this paper achieves 77% the performance of the highly-tuned reference implementation (or a 15% slowdown compared to the base Accelerate implementation).

7.2 Audio Processor

The audio processing benchmark tests the algorithm described in Section 3.3, 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.

Figure 5 shows the strong scaling performance compared to the implementation in regular Accelerate on the CPU.[20] As the number of cores increases, the performance of the vanilla Accelerate implementation begins to drop off as there is no longer enough work in a single window to saturate all processors. On the other hand, the sequences implementation is able to process multiple windowed elements at a time in order to keep all the cores busy, resulting in a speedup over regular Accelerate of 16% on the CPU (at 24 threads), and 13% on the GPU.

[20]Unfortunately we do not have a parallel reference implementation of the algorithm to compare to.

7.3 MD5 Hash

The MD5 message-digest algorithm [35] is a cryptographic hash function producing a 128-bit hash value. The MD5 benchmark attempts to find the plain text of an unknown hash via a standard dictionary attack using a database of 14 million known plain texts. We compare to Hashcat,[21] the self-proclaimed world's fastest CPU-based password recovery tool. Results from Hashcat are as reported by its inbuilt benchmark mode.

Hashcat achieves a maximum throughput of 287 million hashes per second (MH/sec), compared to vanilla Accelerate at 155 MH/s and our sequences implementation at 46 MH/sec. One key difference between our sequences version is that we stream in and process small chunks of the input dictionary at a time, 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. Figure 6 shows the strong scaling performance.

7.4 PageRank

PageRank [30] is a link analysis algorithm which estimates the relative importance of each element of a linked set of documents.

[21]https://hashcat.net/hashcat/

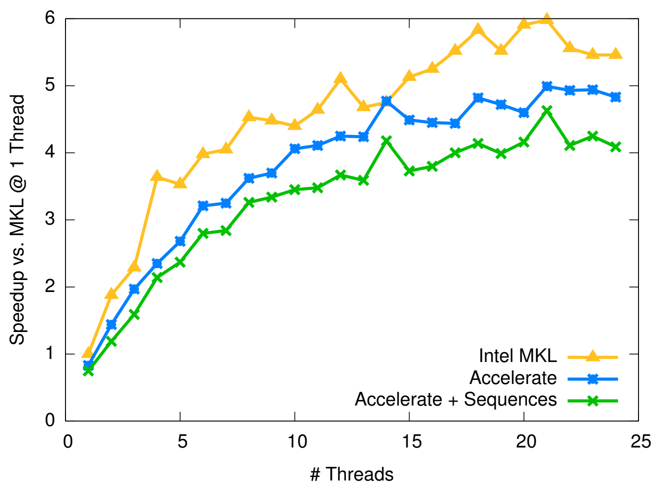

Figure 4. SMVM of Queen_4147 dataset

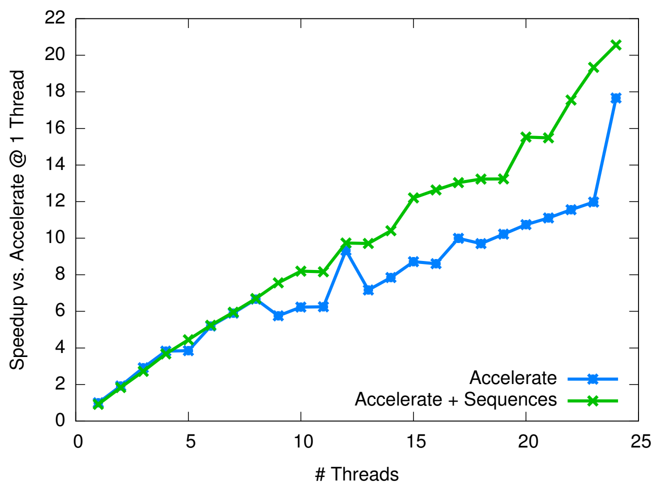

Figure 5. Audio processor

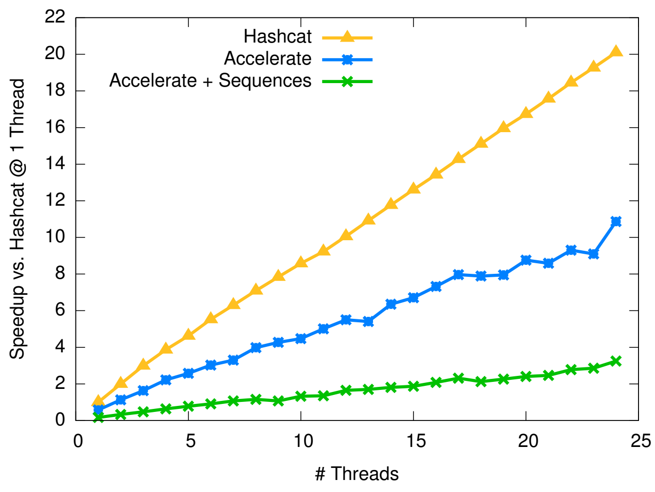

Figure 6. MD5 hash recovery

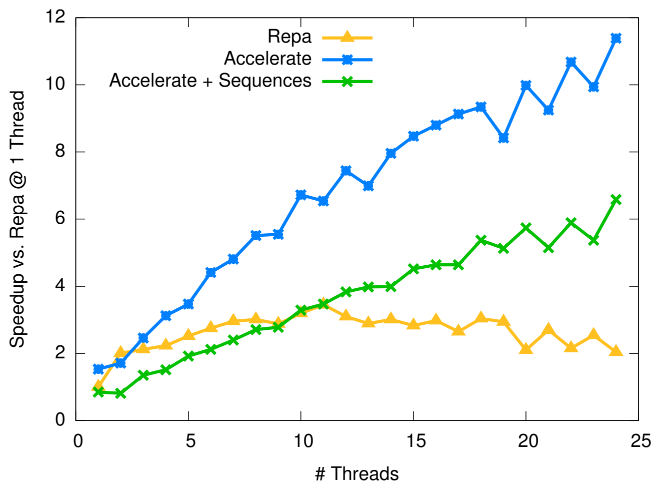

Figure 7. PageRank analysis of Wikipedia (English)

As input we use a link graph of Wikipedia (English) consisting of 5.7 million pages and 130 million links. Figure 7 compares the performance against an implementation written in Repa [21, 23], 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.

8 Related Work

While there are a number of domain specific languages to support GPU programming [3, 11, 16, 22, 27, 36], none of them currently support both parallel arrays as well as streams.

A number of languages aim at facilitating GPU programming in the presence of data streams. Brook [6] is a stream programming extension to C with support for data parallel computations on the CPU. BrookGPU [7] implements a subset of this language on the GPU. Sponge [19] is a compilation framework for GPUs using the synchronous data flow streaming language StreamIt [38].

Both Sponge and BrookGPU are based on a very different programming model to our work. In Sponge and BrookGPU, the stream is the only parallel data-structure available, and the compiler is responsible for transforming this into code suitable for the GPU. In contrast, we provide both parallel arrays as well as streams. As arrays support a wider range of data-parallel operations than streams, this distinction allows the programmer to design for the strengths of each structure while still providing a high level of abstraction.

Proteus [13] uses an idea related to chunking to restrict the memory requirements of vectorisation-based [5] nested data-parallelism. Palmer et al. [31] tackle this problem by partially serialising the flattened program into smaller pieces to be executed sequentially. This is similar in spirit to our work, but aimed as an automatic compiler transformation, rather than being explicit in the language, as we have chosen to do. Our work also distinguishes between regular and irregular computations, which neither of these consider.

As mentioned in Section 4, our transformation implements a variant of vectorisation avoidance [20], which first appeared in Data Parallel Haskell (DPH) [32]. In contrast to DPH, our language is first order and we do not support arbitrary nesting of irregular computations. On the other hand, we extend vectorisation avoidance to distinguish between not only scalar and parallel subcomputations, but to also consider computations which are guaranteed to be regular and those which may be irregular.

Manticore [17] a dialect of ML for NESL-style parallelism on CPUs. Manticore flattens nested data structures [2], but does not need to flatten nested *computations*, as their runtime uses a fork-join execution model, rather than the SIMD flat data-parallel execution model we use.

Nesl/GPU [3] compiles NESL [4] code to CUDA, a system which was independently developed as CuNesl [40]. Performance suffers because the implementations rely on the legacy NESL compiler, which produces a significant number of intermediate computations. Like our work, NESL is a first-order language, but unlike our work supports arbitrary nested parallelism, which we have chosen to instead restrict to a single level of nesting. Their work also does not tackle the problem of space usage in flattened programs, which we address with sequences.

A model for streaming NESL was introduced by Madsen and Filinski [26], motivated by the lack of a space cost model, and is conceptually similar to sequences in Accelerate. Their work is more general than ours in the sense that it allows nested sequences, however it has yet to be implemented into a high-performance compiler. Furthermore, we consider a larger range of operations on, and generalise to, multi-dimensional arrays.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/

[2] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. 2013. Data-Only Flattening for Nested Data Parallelism. In *PPoPP'13: Principles and Practice of Parallel Programming*. ACM, 81–92.

[3] Lars Bergstrom and John Reppy. 2012. Nested data-parallelism on the GPU. In *ICFP: International Conference on Functional Programming*. ACM.

[4] Guy E. Blelloch. 1995. *NESL: A Nested Data-Parallel Language.* Technical Report CMU-CS-95-170. Carnegie Mellon University.

[5] Guy E Blelloch and Gary W Sabot. 1988. Compiling collection-oriented languages onto massively parallel computers. In *Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 575–585.

[6] I Buck. 2003. Brook Language Specification. *Outubro* (2003). http://merrimac.stanford.edu/brook

[7] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH Papers*. ACM, 777–786.

[8] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP: Declarative Aspects of Multicore Programming*. ACM, 3–14.

[9] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. 1990. Scan primitives for vector computers. In *Supercomputing*. IEEE, 666–675.

[10] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP: Declarative Aspects and Applications of Multicore Programming*. ACM.

[11] Koen Claessen, Mary Sheeran, and Joel Svensson. 2008. Obsidian: GPU programming in Haskell. In *IFL: Implementation and Application of Functional Languages.*

[12] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion from lists to streams to nothing at all. In *ICFP: International Conference on Functional Programming*. ACM.

[13] Jan Prins Daniel Palmer and Stephen Westfold. 1995. Work-Efficient Nested Data-Parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers 95).* IEEE.

[14] Tim A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (2011), 1–25. http://www.cise.ufl.edu/research/sparse/matrices

[15] Conal Elliott. 2003. Functional Images. In *The Fun of Programming.* Palgrave.

[16] Conal Elliott. 2004. Programming Graphics Processors Functionally. In *Haskell Workshop.* ACM.

[17] Matthew Fluet, Nic Ford, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Status Report: The Manticore Project. In *ML'07: Workshop on ML.* ACM, 15–24.

[18] Leo J Guibas and Douglas K Wyatt. 1978. Compilation and Delayed Evaluation in APL. In *POPL '78: Principles of Programming Languages.* 1–8.

[19] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. 2011. Sponge: Portable Stream Programming on Graphics Engines. *SIGARCH: Computer Architecture News* 39, 1 (March 2011), 381–392.

[20] Gabriele Keller, Manuel MT Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. 2012. Vectorisation avoidance. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 37–48.

[21] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, and Ben Lippmeier. 2010. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *ICFP: International Conference on Functional Programming.* ACM, 261–272.

[22] Bradford Larsen. 2011. Simple optimizations for an applicative array language for graphics processors. In *DAMP: Declarative Aspects of Multicore Programming.*

[23] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2012. Guiding parallel array fusion with indexed types. In *Haskell Symposium.* ACM, 25–36.

[24] Ben Lippmeier, Manuel M T Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Simon Peyton Jones. 2012. Work Efficient Higher-Order Vectorisation. In *ICFP'12: International Conference on Functional Programming.* ACM, 259–270.

[25] Frederik M. Madsen, Robert Clifton-Everest, Manuel M. T. Chakravarty, and Gabriele Keller. 2015. Functional Array Streams. In *FHPC'15: Workshop on Functional High-Performance Computing.* ACM, 23–34.

[26] Frederik M. Madsen and Andrzej Filinski. 2013. Towards a Streaming Model for Nested Data Parallelism. In *FHPC'13: Workshop on Functional High-performance Computing.* ACM, 13–24.

[27] Geoffrey Mainland and Greg Morrisett. 2010. Nikola: Embedding Compiled GPU Functions in Haskell. In *Haskell Symposium.* ACM.

[28] Trevor L. McDonell, Manuel M T Chakravarty, Vinod Grover, and Ryan R Newton. 2015. Type-safe Runtime Code Generation: Accelerate to LLVM. In *Haskell '15: Symposium on Haskell.* ACM, 201–212.

[29] Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *ICFP: International Conference on Functional Programming.* 49–60.

[30] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: Bringing order to the web. (1999).

[31] Daniel W. Palmer, Jan F. Prins, Siddhartha Chatterjee, and Rickard E. Faith. 1996. Piecewise execution of nested data-parallel programs. In *Languages and Compilers for Parallel Computing.* Springer Heidelberg, 346–361.

[32] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M T Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Foundations of Software Technology and Theoretical Computer Science.*

[33] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *ICFP'06: International Conference on Functional Programming.* 50–61.

[34] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI'13: Programming Language Design and Implementation.* ACM.

[35] Ronald Rivest. 1992. The MD5 message-digest algorithm. (1992).

[36] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Martin Odersky, and Kunle Olukotun. 2013. Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers based on Staging. In *POPL'13: Principles of Programming Languages.* ACM.

[37] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. 2007. Scan primitives for GPU computing. In *Symposium on Graphics Hardware.* Eurographics Association, 97–106.

[38] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Compiler Construction.* Springer.

[39] Philip Wadler. 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (June 1990), 231–248.

[40] Yongpeng Zhang and F Mueller. 2012. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. In *ICPP '12: International Conference on Parallel Processing.* 340–349.