Functional Array Streams

Frederik M. Madsen
University of Copenhagen
Department of Computer Science (DIKU)
fmma@di.ku.dk

Robert Clifton-Everest
Manuel M. T. Chakravarty
Gabriele Keller
University of New South Wales,
School of Computer Science and Engineering
{robertce,chak,keller}@cse.unsw.edu.au

Abstract

Regular array languages for high performance computing based on aggregate operations provide a convenient parallel programming model, which enables the generation of efficient code for SIMD architectures, such as GPUs. However, the data sets that can be processed with current implementations are severely constrained by the limited amount of main memory available in these architectures.

In this paper, we propose an extension of the embedded array language Accelerate with a notion of sequences, resulting in a two level hierarchy which allows the programmer to specify a partitioning strategy which facilitates automatic resource allocation. Depending on the available memory, the runtime system processes the overall data set in streams of chunks appropriate to the hardware parameters.

In this paper, we present the language design for the sequence operations, as well as the compilation and runtime support, and demonstrate with a set of benchmarks the feasibility of this approach.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classification—Applicative (functional) languages; Concurrent, distributed, and parallel languages

Keywords Streams; Arrays; Data parallelism; GPGPU; Haskell; Embedded language

1. Introduction

Functional array languages facilitate high-performance computing on several levels. The programmer can express data-parallel algorithms declaratively, and the compiler can exploit valuable domain specific information to generate code for specialised parallel hardware. A standard array language separates itself from traditional languages by offering data-parallel collection oriented constructs as primitives such as map, fold, scan and permutation. Without these primitives, the same logic would have to be encoded as sequential for-loops or recursive definitions, obfuscating data-dependency and access patterns. Due to the artificial data-dependencies introduced by the loop counter or the recursion stack, these encodings prevent natural parallelisation, and the compiler must resort to program analysis to detect and exploit implicit parallelism. Such automatic parallelisation strategies are fragile, and small changes in the code may cause them to fail, significantly degrading the performance for reasons not obvious to the programmer.

Array languages present a complementary problem. The explicit data-parallelism exposed in an array program may vastly exceed the actual parallel capabilities of the target hardware. Data-parallel programs often require working memory in order of the degree of parallelism. Therefore, it is not always desirable or even possible to execute an array program in its full parallel form. To conserve space, we would like the compiler to make the program "less parallel" prior to execution. However, the absence of explicit sequential data-dependencies prevents natural *sequentialisation*.

If we would execute each parallel combinator in isolation, we could simply sequentialise the combinator by partitioning the index-space and scheduling the different parts in a tight loop. Evidently, this is how CUDA schedules a kernel in blocks on a large grid.

In practice, however, it is essential to fuse sequences of parallel combinators together to form complex computations, thereby reducing the number of array traversals and intermediate structures. As soon as such a sequence includes more than simple maps, the combinators may not traverse the index-space in a uniform way. Consequently, loop fusion can be very complex. Compiler-controlled sequentialisation affects the fusion-transformation, and complicates it further. Finding the optimal sequentialisation strategy in this context is not decidable, so we would have to resort to using heuristics, leaving the programmer at the mercy of the compiler again.

Therefore, we propose to give control over this step to the programmer, who has more knowledge about the nature of the application and size of the processed data set. We achieve this by including a set of *sequence* combinators for array languages, so sequential data-dependency over data-parallel computations can be specified and the amount of parallelism exposed be controlled.

This paper presents these new sequence combinators, using the language Accelerate as starting point and discusses the extensions to the runtime system with the required streaming and scheduling mechanism. In summary, the contributions of this paper are as follows:

- We present a new set of sequence combinators, which, together with the usual combinators like maps, folds and scans, can be used to express a two-level hierarchy sequentially combining a sequence of parallel operations over chunks of data.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

, .
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/

- We present a runtime system extension which implements the necessary scheduling and streaming mechanisms.
- We present an evaluation of the approach presented in the paper.

While we are currently only targeting single-GPU architectures, the programming model we propose in this paper also allows the programmer to expose pipeline-parallelism in a program, which we could exploit in an implementation for multi-GPU architectures. Although outside the scope of this paper, other data-parallel architectures, such as muti-processors and distributed systems, would also benefit from the model presented here.

2. Accelerate

Accelerate is a domain specific functional language for high-performance, multi-dimensional array computations, implemented as deep embedding in Haskell. In addition to the collection oriented operations similar to those on lists, like maps, scans, reductions, it also offers array-oriented operations, such as stencil convolutions, as well as forward- and backward permutations, conditionals and loops. Indeed, apart from the type annotation (to which we will get back shortly), many Accelerate programs – like this dot-product example – look almost like the corresponding list operation in Haskell:

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

In contrast to Haskell, though, Accelerate is strict and fully normalising.

The language has two non-obvious restrictions: (1) Arrays must be *regular*. By regular, we mean arrays cannot contain other arrays as elements. Instead, arrays are multidimensional. Scalars, vectors, matrices, tensors, and so on, are all regular arrays, but a vector of arbitrary-length vectors is not. (2) Accelerate does not permit nested data-parallelism. For example, even though one could imagine using a two dimensional array as a vector of vectors, mapping a map over each sub-vector is not allowed. These restriction enables a smooth compilation to SIMD hardware. Accelerate comes with a number of backends, among them a GPU implementation generating CUDA [23] code, that demonstrate excellent performance [7, 22].

The restriction to regular computations and arrays is enforced statically via the type system, which serves to separate Accelerate-expression into two distinct categories.

- `Elt a => Exp a`: Expressions which evaluate to values of type a, where a has to be a member of type class `Elt`, which includes basic types such as integers, floats, bools, as well a n-tuples of these. Accelerate generates valid CUDA C from `Exp` expressions.
- `(Shape sh, Elt a) => Acc (Array sh a)`: Expressions which evaluate to n-dimensional arrays of type with element type `a` and shape `sh`. Accelerate generates CUDA GPU kernels from `Acc` expressions.

Shapes are sequences of integer dimensions separated by `:.` (e.g. `Z :. 2 :. 3`), and `Scalar` is a type synonym for a zero-dimensional array, `Vector` for a one-dimensional array. The type annotation of the `dotp` example therefore states that the function accepts two floating point vectors as arguments, and returns a scalar floating point value as result. More precisely, since Accelerate is a deep embedding, `dotp` takes in accelerate expressions specifying computations which produce results of these types, and returns a new computation.

Functions like `map`, `fold` and so on are *rank polymorphic*. For example, the type of `zipWith` is

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

Several things are happening here: the type of the function passed to `zip` is, by its type, restricted to sequential computations over values of basic type. The type class constraint `Shape sh` essentially restricts `sh` to n-tuples of integers, where n determines the rank of the array. Both array arguments have to have the same rank, which is also the rank of the result. The actual sizes, however, are not tracked statically and may be different.

2.1 Fusion

It is well known that the collection oriented style of programming which Accelerate relies on has a serious potential drawback: if implemented naively, by executing each aggregate operation separately, it can result in an excessive number of array traversals, intermediate structures, and poor locality. For example, it would clearly be inefficient if the code for the `dotp` example would first produce an intermediate array of the pairwise products, and then, in a second traversal, add all these sums to the final result. Therefore, Accelerate aggressively employs *fusion*, merging the operations to more complex computations, trying to minimise the number of traversals.

Fusion cannot, in general, guarantee that its results are optimal. Consider, for example, a fusible computation whose result is consumed by two different operations. If we would fuse into both consumers, we would avoid creating the intermediate structure, but duplicate the work involved to compute the array element, which can, in theory, slow down the performance considerably. In practice, most computations are fairly cheap compared to creating and accessing an array, so fusion would result in a significant speed-up nevertheless. Without sophisticated cost-analysis, the compiler cannot decide which alternative results in the best performance, so we err on the side of caution and never fuse computations whose result is used more than once.

2.2 Handling large data sets

We have shown previously [7, 22] that the Accelerate approach of expressing parallel computations enables the generation of highly efficient code. The dot-product in Accelerate, for example, is only slightly slower than CUBLAS, a hand-written implementation of the Basic Linear Algebra Subprograms in CUDA by NVIDIA. However, on GPU architectures, we can only achieve peak performance if the data set we are processing in one parallel step is large enough to utilise all processing elements, yet small enough to still fit in the GPU memory, which is, at currently around 4GB, for the majority of hardware, much more restricted than CPU memory.

If programmers want to develop GPU programs which process larger set of data, they have to explicitly stage the computation into a sequence of parallel computations on smaller data chunks, and combine the subresults. While this is possible, it adds a significant layer of complexity to an already difficult task, and would lead to code whose relative performance is architecture dependent.

The other extreme option would be to try and let the compiler shoulder all the complexity of solving this problem. However, sophisticated optimisations like these have the downside that they usually cannot guarantee optimality, and behave in a way hard to predict by the programmer. Therefore, we choose an intermediate route: we allow the programmer to explicitly distinguish between

parallel, random access structures, and streamed ones, which allow only for a more limited set of operations. This gives the programmer the opportunity to design an algorithm tailored for this model, instead of hoping the compiler optimisations will work out.

In the following section, we describe the stream extension to Accelerate's program model, before we discuss its implementation and performance.

3. Programming Model

3.1 Examples

Let us go back to our `dotp` example. If we know that the input vectors most likely will not fit into memory, or we wish to ensure it minimises its space usage, we want to tell the compiler to split the input into chunks of appropriate size, calculate the product and sum for each chunk, and add the subresults as they are produced. Our stream extension makes this possible:

```
dotpSeq :: Acc (Vector Float)
        → Acc (Vector Float)
        → Acc (Scalar Float)
dotpSeq xs ys =
    collect
  $ foldSeqE (+) 0
  $ zipWithSeqE (*) (toSeqE xs) (toSeqE ys)
```

Here, `toSeqE` turns a normal `Vector` into a sequence of `Scalars`, `zipWithSeqE` performs element-wise multiplication of the two input sequences, `foldSeqE` calculates the sum, and `collect` takes the conclusion of the sequence computation and turns it into an `Acc` expression. The rest of this section will explain these primitives in more detail.

As Accelerate is rank-polymorphic, sequence operations can be parametrised by shape information. By convention, we denote specialised versions of these operations for sequences of scalars by the suffix `E`, as for example `toSeqE` above.

It is not just sequences of scalars that are supported, however. Our extension supports sequences of arbitrary rank. If for example we wanted to perform a matrix vector multiplication:

```
mvmSeq :: Acc (Matrix Float)
       → Acc (Vector Float)
       → Acc (Vector Float)
mvmSeq mat vec
  = let rows = toSeq (Z:.Split:.All) mat
    in collect
     $ fromSeqE
     $ mapSeq (dotp vec) rows
```

In this case, we first split the vector up into rows with `toSeq`, then apply `dotp vec` over every row, turn what is now a sequence of scalars into a `Vector` with `fromSeqE`, before finally `collect`ing the result.

In addition to not requiring the entire matrix be made manifest, this example also highlights how our extension enables an extra degree of nesting, in this case, defining matrix-vector multiplication in terms of the parallel dot-product, something not previously possible.

3.2 Streams

As we have seen in the previous examples, an Accelerate array is a collection where are all elements are simultaneously available, whereas a sequence value corresponds to a loop, where each iteration computes an element of the sequence. Sequences are ordered temporally, and are traversed from first to last element. Once an element has been computed, all previous elements are out of scope, and may not be accessed again. The arrays of a sequence are restricted to having the same rank, but not necessarily the same shape. If the shapes happen to be the same, we call the sequence *regular*. Using A to range over array values, and square brackets to denote sequences,

$$[A_1, A_2, ..., A_n]$$

denotes the sequence that computes A_1 first, computes A_2 second and so forth until the final array A_n is computed. Here, n is the length of the sequence (possibly zero).

Sequences model the missing high-level connection between the parallel notation of array languages and sequential notation of traditional for-loops. The basic sequence combinators are carefully selected such that the arrays of a sequence can be evaluated entirely sequentially, entirely parallel, or anything in between as long as the strategy respects the sequence order of arrays; even on SIMD hardware. The runtime system then selects a strategy that fits the parallel capabilities of the target hardware. The programmer may assume full parallel execution with respect to what the hardware can handle, while maintaining a limit on memory usage. A purely sequential CPU would evaluate one array at a time with a minimal amount of working memory. A GPU would evaluate perhaps the first 100 arrays in one go, and then evaluate the next 100 arrays, and so on. The working memory would be larger, but not as large as the cost of manifesting the entire sequence at once. Ideally, the runtime performance, in terms of execution time, should correspond to a fully parallel specification, and in terms of working memory, should be in the order of a constant factor related to the parallel capabilities of the hardware - Unless any one array of the sequence exceeds this amount.

Of course, not all array algorithms can be expressed as sequences. As sequences can only be accessed linearly, any algorithm which relies on permuting or reducing an array in a non-linear way, cannot be expressed as a sequence. It is the responsibility of the programmer, not the compiler, to expose inherent sequentialism.

3.3 From arrays to sequences and back

As we discussed previously, the type constructors `Exp` and `Acc` represent nodes in the AST from which Accelerate generates CUDA C code and CUDA GPU kernels, respectively. Sequence computations are represented by the type constructor `Seq`. Accelerate will generate CUDA kernels together with a schedule for executing the kernels over and over until completion.

While the type constructor `Seq` represents sequence AST nodes, we use the Haskell list syntax to represent the actual sequence type. That is, the type `[a]` represents sequences of `a`'s, and the type `Seq [a]` represents sequence computations that produce sequences of `a`'s when executed. The type `Seq a`, where `a` is *not* a sequence type, represents sequence computations that produce a single result of type `a`. We will see an example of such a type when we explain `foldSeqE`.

Sequences are introduced in Accelerate either by slicing an existing array, as we did in our examples, or by streaming an ordinary Haskell list into Accelerate, which we will discuss in detail in Section 3.4.

In our examples, we used the combinator `toSeqE` to convert one dimensional array into a sequence of values of the same element type:

```
toSeqE :: (Elt a)
    ⇒ Acc (Vector a)
    → Seq [Scalar a]
```

However, `toSeqE` is just a special case of the more general combinator `toSeq`, which operates on multi-dimensional arrays and is parametrised with a specific slicing strategy `div`:

```
toSeq :: (Division div, Elt a)
    ⇒ div
    → Acc (Array (FullShape  div) a)
    → Seq [Array (SliceShape div) a]
```

Values of types belonging to the `Division` type-class define how an array is divided into sub-arrays along one or more dimensions, where `Split` at a given position tells the compiler to divide the elements along the corresponding dimension into a sequence, `All` to leave it intact.

Divisions are generated by the following grammar.

$$\texttt{Div} \ni div ::= \texttt{Z} \mid div \texttt{ :. All} \mid div \texttt{ :. Split}$$

`FullShape` and `SliceShape` are type functions that, for a given division, yield the shape of the full array and the shape of every slice. Let us have a look at an example to see how divisions can be used to slice a two dimensional array in different ways. Let A be the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 10 & 11 & 12 \end{pmatrix}$$

then we can either leave the matrix intact and create a sequence containing one element (somewhat pointless), slice it column-wise, row-wise, or element-wise:

$$\begin{aligned}
\texttt{toSeq}\,(\texttt{Z :. All :. All})\; A &= [\left(\begin{smallmatrix} 1 & 2 & 3 \\ 10 & 11 & 12 \end{smallmatrix}\right)] \\
\texttt{toSeq}\,(\texttt{Z :. All :. Split})\; A &= [\left(\begin{smallmatrix} 1 \\ 10 \end{smallmatrix}\right), \left(\begin{smallmatrix} 2 \\ 11 \end{smallmatrix}\right), \left(\begin{smallmatrix} 3 \\ 12 \end{smallmatrix}\right)] \\
\texttt{toSeq}\,(\texttt{Z :. Split :. All})\; A &= [\left(\begin{smallmatrix} 1 & 2 & 3 \end{smallmatrix}\right), \left(\begin{smallmatrix} 10 & 11 & 12 \end{smallmatrix}\right)] \\
\texttt{toSeq}\,(\texttt{Z :. Split :. Split})\; A &= [1, 2, 3, 10, 11, 12]
\end{aligned}$$

Fusion, as described in Section 2.1, is applied across sequentialisation. This means that, if matrix A is the result of a computation, we leverage the existing fusion transformation of Accelerate to combine it with any operation on A_i. In this way, we avoid the full manifistation of A.

If A is the result of an operation that prevents subsequent fusion, such as a reduction or a scan, we have no choice but to materialise the entire input array prior to slicing. This undesirable effect can sometimes be avoided by a simple transformation that moves the fusion-preventing operation into the sequence computation. As an example, consider the following program that converts a matrix into a sequence of row sums:

```
rowSums :: Acc (Array DIM2 Int)
        → Seq [Scalar Int]
rowSums mat =
  toSeqE (fold (+) 0 mat)
```

Since `fold` prevents further fusion in Accelerate, the result of `(fold (+) 0 mat)` will be a fully materialised vector, that happens to hold the entire sequence at once. If `mat` has a huge number of rows, full manifestation is catastrophic. However, it is entirely unnecessary and can be avoided in this case by first constructing a sequence of rows, and then mapping a sum over that sequence:

```
rowSums' mat =
  mapSeq
    (fold (+) 0)
    (toSeq (Z:.Split:.All) mat)
```

Assuming the user of this function provides an array expression `mat` that does not prevent further fusion, `mat` will be fused row-wise into the first operation of the sequence. Therefore, no initial manifestation is required and this definition works for arbitrary many rows. As a subject for future work, the compiler could potentially perform array-to-sequence expression transformations like this one. For now, as a rule of thumb when working with sequences, it is advisable to slice early and put as much of the program logic in `Seq` rather then in `Acc`. Finally, as a specific optimization for the GPU backend, if A is a host-side array constant, it will be transferred to the device in parts.

Sequences of arrays can be converted into flat data vectors and a vector containing the shape of each array, or to the data vector only if we are not interested in the shapes:

```
fromSeq :: (Shape ix, Elt a)
     ⇒ Seq [Array ix a]
     → Seq (Vector ix, Vector a)
```

Ordinary Accelerate array function can be lifted from working on arrays to working on sequences using $mapSeq$ and $zipWithSeq$. These sequence combinators are parametrised by the to-be-lifted array function, and the denotation is simply to apply the function to each array of the input sequence(s).

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

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

The type class `Arrays` contains n-tuples of `Array` type, expressing the fact that the arguments of both operations can be multiple arrays.

In addition to mapping operations over sequences, we can fold a sequence of scalars with an associative binary array operator. Unlike with `map` and `zipWith`, `foldSeqE` is not implemented in terms of a more general `foldSeq`. The reason why is explained in Section 4.

```
foldSeqE :: Elt a
         ⇒ (Exp a → Exp a → Exp a)
         → Exp a
         → Seq [Scalar a]
         → Seq (Scalar a)
```

Note that `foldSeq` still returns a sequence computation, but the result of that computation is a scalar array, not a sequence. This allows multiple reductions to be expressed and contained in the same sequence computation, ensuring a single traversal. For example, here we have two reductions, one summing the elements of an array, the other calculating the maximum. We can combine this into a single traversal with `lift`.

```
maxSum :: Seq [Scalar Float]
       → Seq (Scalar Float, Scalar Float)
maxSum xs = lift ( foldSeqE (+) 0 xs
                 , foldSeqE max 0 xs)
```

If we want to convert this back into an `Acc` value, we need to use `collect`:

```
collect :: Arrays arrs
        ⇒ Seq arrs → Acc arrs
```

Note the `Arrays` constraint on `arrs` in `collect`. As sequences are not members of the `Arrays` class this ensures that we cannot embed a whole sequence into an array computation without first reducing it to an array.

3.4 Lazy lists to sequences

Our language extension allows interfacing with ordinary Haskell lists. We define two convenient operations for converting sequence expressions to Haskell list and vice versa.

```
streamIn  :: Arrays a ⇒ [a] → Seq [a]
streamOut :: Arrays a ⇒ Seq [a] → [a]
```

`streamIn` is a language construct that takes a constant sequence and embeds it in Accelerate. It is the sequence-equivalent of an array constant in ordinary Accelerate. `streamOut` on the other hand, is an interpretation that runs a sequence expression and produces a Haskell list as output. Therefore, it must be defined on a per-backend basis, just like the ordinary Accelerate interpretation function `run :: Arrays a ⇒Acc a →a`. Using `streamOut` is the only way to interpret a sequence expression that is not embedded in an array expression.

Accelerate is a strict language, and has not been equipped to deal with infinite sequences until now. Arrays are naturally finite, and the result sequence of `toSeq` is no longer than the size of the input array. However, there is nothing that prevents the programmer from passing an infinite list to `streamIn`. Being a strict language, Accelerate will go into an infinite loop if the programmer attempts to reduce an infinite sequence. It is however possible to productively stream out an infinite sequence to an infinite Haskell list. The elements will then be forced according to the evaluation strategy, which is hidden from the programmer. For example, if the programmer tries to print the third element of a streamed out sequence in Haskell, Accelerate may internally evaluate the first ten elements.

4. Execution Model

After discussing the language extensions for sequences, we are now looking into how we can generate efficient code from these sequence expressions. For a sequential CPU architecture, a sequential, element-by-element evaluation would be feasible, but would clearly lead to unacceptable performance on our main target architecture, GPUs. Instead, we want to process just enough data to saturate the GPU to achieve optimal performance. Therefore, before code generation, we group multiple elements of the sequence together in vectors to form *chunks*. Each chunk can then be streamed to the GPU and processed in parallel. The actual size of the chunk is chosen by the runtime, as the best choice depends on the concrete architecture the program is executed on.

We define a *chunk* to be a vector of arrays (or n-tuple of arrays) written with angular brackets $\langle A_1, ..., A_k\rangle$. Each array is required to have the same rank, but not necessarily the same shape. k is referred to as the length of the chunk, and the total size of the chunk elements $\sum_{i\in\{1..k\}} size(A_i)$ is referred to as the chunk size. If all the arrays have the same shape, we say that the chunk is regular. Note that a regular chunk is essentially just an array with rank $r+1$ where r is the rank of each element.

The execution model presented here implements the programming model by translating sequence expressions to stream-manipulating acyclic dataflow graphs, where the nodes consume and/or produce chunks that flow along the edges. There are two key challenges in this approach: Lifting and scheduling. Sequence operations must be lifted at compile time to operate on chunks instead of just arrays. At run time, appropriate chunk lengths must be selected as small as possible while still keeping the backend saturated in each step, and the sequence operations must be scheduled accordingly. We solve these challenges for regular chunks by means of *vectorisation* together with an analysis phase that yields a static schedule. We proceed to explain the vectorisation strategy of each primitive sequence operation.

- Array slicing is trivial to vectorise. `toSeq` is easily extended to produce chunks of slices, and the chunks will always be regular with known sizes.
- For `streamIn`, since Accelerate cannot track shapes, there is no guarantee that the list supplied by the programmer contains same-shape arrays, and consequently, we consider the resulting sequence to be irregular in all cases. One could imagine the addition of a `streamInReg` operation that takes the shape of elements as an additional argument. The programmer then promises that all arrays in the supplied list have this shape. Such an operation would be beneficial for applications streaming large amounts of regular data such as video processing.
- Sequence maps (and zipWith's) are vectorised by applying a lifting transformation on the argument array function as described in Section 4.2. Sequence maps are the main source of irregularity since we can map any array functions. As Accelerate cannot handle irregular arrays, we analyse the mapped array functions to detect irregularity and avoid chunking in that case. The analysis is described in Section 4.4.
- For sequence reduction with an array function as the combining operator, we need to turn an array-fold into a chunk-fold. Vectorizing the combining operator gives a function that combines chunks. We could fold the chunks of a sequence with this function and then use the unlifted function in the end to fold the final chunk. However, there are a number of problems with this approach:
  - The combining operator would have to be commutative since elements are combined, not with the next element in the sequence, but with the element in the next chunk at the same position.
  - It is not always desirable to keep a chunk of accumulated values. For example, `fromSeq` is a fold using array append as the combining operator, and the accumulated value is an array containing all the elements in the sequence seen so far. A chunk of accumulated values would be unreasonably large.

  A better solution is to fold each chunk with the unlifted function immediately and then combine the resulting folded value with the accumulator, again using the unlifted function. However, Accelerate does not support a general parallel array fold. Instead, as described in the following paragraphs, we opt to provide a less general sequence fold primitive more suitable for chunking.

The sequence fold primitive is named `foldSeqFlatten` and has the type signature:

```
foldSeqFlatten :: (Arrays a, Shape sh, Elt b)
    ⇒ (  Acc a → Acc (Vector sh)
       → Acc (Vector b) → Acc a)
    → Acc a
    → Seq [Array sh b]
    → Seq a
```

This operation works by applying the given function to each chunk in a sequence. In each step, the function is applied to an accumulated value, a vector of shapes and a vector of elements, and it produces a new accumulated value. The vector of shapes is the shapes of the arrays in the input chunk, and the vector of elements is all the elements of the chunk concatenated to a flat vector. Our representation of chunks enables extracting the shape vector and element vector of a chunk in constant time, also for irregular chunks. This means that we can execute `foldSeqFlatten` on chunks of any length without having to vectorise. The operation unfortunately ex-

poses the chunk size in the surface language as the size of the shape vector. This is not something the programmer should rely on since it is backend specific parameter. However, the programmer is obligated to obey the following rule for the folding function:

```
f (f a sh1 e1) sh2 e2 = f a (sh1 ++ sh2) (e1 ++ e2)
```

That is, applying the folding function twice on two shape and element vectors must be the same as applying it once on the appended vectors. This severely limits how the programmer can exploit knowing the chunk size.

Scalar fold `foldSeqE` is then defined as a prelude function using `foldSeqFlatten` and the standard `fold` operator of accelerate:

```
foldSeqE :: Elt a
         ⇒ (Exp a → Exp a → Exp a)
         → Exp a
         → Seq [Scalar a]
         → Seq (Scalar a)
foldSeqE f z =
  foldSeqFlatten
   (λ acc _ → fold f (the acc))
   (unit z)
```

Here `the` is convenience function for indexing a scalar array.

Likewise, `fromSeq` is also currently also a prelude function. Folding with append is not very efficient, so we plan to specialise this operation in the near future.

```
fromSeq :: (Shape ix, Elt a)
        ⇒ Seq [Array ix a]
        → Seq (Vector ix, Vector a)
fromSeq = foldSeqFlatten f (lift (empty, empty))
  where
    f x sh1 a1 =
      let (sh0, a0) = unlift x
      in lift (sh0 ++ sh1, a0 ++ a1)
```

Here `empty` produces the empty vector and (`++`) is vector append.

4.1 Translation

Sequence expression are first converted to A-normal form where producing sequences are let-bound, and the sequence arguments in a sequence operation must be variables. For example, the sequence dot-product example is converted into the form:

```
let s1 = toSeqE xs
    s2 = toSeqE ys
    s3 = zipWithSeqE (*) s1 s2
in foldSeqE (+) 0 s3
```

If a sequence expression contains more than one consumer, they are grouped together in a n-tuple. The A-normal expression is then traversed from top to bottom and translated into the following continuation-passing-style executable representation:

```
data StreamDAG senv res where
  Transduce
    :: (senv → s → (a, s))
    → s
    → (s → Bool)
    → StreamDAG (senv, a) res
    → StreamDAG senv res

  Consume
    :: (senv → s → s)
    → s
    → StreamDAG senv s

  Reify
    :: (senv → [a])
    → StreamDAG senv [a]
```

The type variable `senv` refers to the surrounding sequence context that holds the current live values in each step, and `res` is the final result of the entire stream execution.

`Transduce` is a stream transformer with a local state `s` that produces values of type `a` from the surrounding context `senv`. Sequence maps, `toSeq` and `streamIn` translates to transducers. In each step of the stream, the termination condition of type

```
(s -> Bool)
```

is checked. It is only `toSeq` and `streamIn` that can terminate a sequence, and they do so once there are no more chunks to produce. If the stream is not ready to terminate, the stepping function of type

```
(senv -> s -> (a, s))
```

will be applied to the current context and state to produce a value of type `a` that is made available to the subsequent nodes. The functional also produces a new state to be used in the next iteration. The subsequent nodes are then stepped by recursively stepping the argument of type

```
StreamDAG (senv, a) res.
```

`Consume` also carries a local state that is updated in each step. The final state in a consumer will be the result of the sequence. Once termination is reached, the result can be read from the value of the consumer. `foldSeqFlatten` is currently the only operation that translates to a consume. Multiple folds are combined in a single consume node.

`Reify` is a special kind of consumer that produces a list of values in each step. This constructor is only used when a sequence is streamed out. In this case, instead of a final result, the sequence produces a list of intermediate results. The function argument converts a chunk from the surrounding sequence context into a list of arrays. These lists are appended together to form the result of `streamOut`. When an element belonging to a chunk is forced in the host language, the whole chunk is forced along with all preceding chunks that have not been forced already.

The stepping functions in the stream DAG are mostly obtained by evaluating the array-typed arguments of the operations in the sequence using the existing backend. After vectorisation is applied, the translation becomes straight-forward. The only new backend-specific operation we had to define are related to slicing and streaming in and out.

4.2 Vectorization

Operations applied to the elements of a sequence in the source program have to be applied in parallel to all the elements of a chunk when we execute the code on a GPU. This process of lifting element-wise operations to a parallel operation on a collection, referred to as the *lifting transform* [4], was popularised by NESL [3].

Lifting, for a fully featured, higher-order functional language is a complicated process and can easily introduce inefficiencies [15, 18]. However, the constrained nature of the Accelerate array language works in our favour here, and we can get the job done with a much simpler version of the transformation. While the user-facing surface language may appear to be higher order, it is strictly first-order. That is to say, the only higher order functions are primitive operations (e.g. map, $zipWith$, $fold$) and functions cannot be let bound.

The transformation $\mathcal{L}[\![f]\!]$, which lifts a (potentially already parallel) function $f :: \alpha \to \beta$ into vector space has type:

$$\mathcal{L}[\![f]\!]_{xs} :: \texttt{Vector}\ \alpha \to \texttt{Vector}\ \beta$$

Conceptually, it is just a parallel map. However, we cannot implement it as such, since f may already be a parallel operation, and feeding it to a parallel map would result in nested parallelism, precisely what we want to avoid. In order to maintain flat parallelism, the lifting transform must recurse over the term applying the transformation to all subterms. Formally, this is defined as follows.

$$\begin{aligned}
\mathcal{L}[\![\mathbf{C}]\!]_{v:_} &= \mathit{replicate}\ (\mathit{length}\ v)\ \mathbf{C} \\
\mathcal{L}[\![x]\!]_{v:vs} &= \begin{cases} x & x \in (v:vs) \\ \mathit{replicate}\ (\mathit{length}\ v)\ x & x \notin (v:vs) \end{cases} \\
\mathcal{L}[\![\lambda v.\ e]\!]_{vs} &= \lambda v.\ \mathcal{L}[\![e]\!]_{v:vs} \\
\mathcal{L}[\![e_1\ e_2]\!]_{vs} &= \mathcal{L}[\![e_1]\!]_{vs}\ \mathcal{L}[\![e_2]\!]_{vs} \\
\mathcal{L}[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!]_{vs} &= \mathbf{let}\ v = \mathcal{L}[\![e_1]\!]_{vs}\ \mathbf{in}\ \mathcal{L}[\![e_2]\!]_{v:vs} \\
\mathcal{L}[\![\mathbf{P}]\!]_{_} &= \mathbf{P}^{\uparrow}
\end{aligned}$$

Here, the lifting transform, $\mathcal{L}[\![\cdot]\!]_{\cdot}$, takes a term along with a subset of its environment, expressed as a list of variables. The subset corresponds to the variables that have been lifted as part of the transform. For example, given `mapSeq` $f\ seq$, we want to vectorise f, but f may contain variables that were bound outside of the sequence computation, in which case we have to replicate them at their use sites. Similarly, constants, $\mathbf{C}$, are also replicated at their use sites.

In the case of primitive operations, $\mathbf{P}$, we use a lifted version of that operation, $\mathbf{P}^{\uparrow}$ defined in terms of existing primitives. Fortunately, the range of combinators Accelerate provides is rich enough that lifted versions of all of them are able to be implemented, with our chosen nested array representation.

4.2.1 Nested array representation

A consequence of this flattening transform is that it produces nested vectors. For example lifting

$$g :: \texttt{Vector Float} \to \texttt{Vector Int}$$

is going to yield

$$\mathcal{L}[\![g]\!] :: \texttt{Vector}(\texttt{Vector Float}) \to \texttt{Vector}(\texttt{Vector Int}).$$

Indeed, with the multidimensional arrays accelerate supports, arrays of type `Vector (Array (Z:.Int:.Int) Float)` could occur. One possible, and elsewhere very popular, flattened array representations is this: A vector of arrays is represented as a vector of segment descriptors (the shape of the sub-array) along with a vector containing all the values of the sub-arrays flattened and concatenated. For example:

$$\left[\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \end{pmatrix}\right]$$

$$([2 \mathrel{:.} 2, 3 \mathrel{:.} 2)], [1, 2, 3, 4, 10, 11, 12, 13, 14])$$

However, this representation is unnecessarily general for our model. As our execution model only allows chunked execution for *regular* chunks, with this representation, all the segment descriptors would be the same. Instead, we use a much simpler representation: A regular vector of arrays of rank sh can be represented as an array of rank $sh \mathrel{:.} Int$. Of course, this affects the lifting transform. If we only allow regular vectors then not all array functions can be lifted. For example, this function cannot be lifted.

```
enumFromTo :: Int → Int → Vector Int
```

The reason why is its lifted form does not produce a regular nested vector. The size of each sub-vector of the output is determined by the contents of the input vectors.

```
L⟦enumFromTo⟧. :: Vector Int
                → Vector Int
                → Vector (Vector Int)
```

We classify an array function as regular if the shape of the output, as well as the shape of any intermediate arrays, can be determined from the shape of the input and, assuming the function is open, its environment. We describe in Section 4.4 how we identify these functions and compute their parallel degree.

This regular nested array representation is also of benefit when it comes to defining the lifted version of the built in primitives. Typically, the lifting transform relies heavily on prefix-sums (scans) to perform these *segmented* operations [2]. Forgoing the segment descriptors means that, for example, our lifted `fold`, can be implemented purely in terms of regular fold. This is because it is already rank polymorphic.

```
fold :: (Exp e → Exp e)
     → Exp e
     → Array (sh:.Int) e
     → Array sh e
```

That is to say, it folds along the inner dimension, reducing the rank by one.

While most of Accelerate's primitives are rank polymorphic, there are some notable exceptions. Primarily, prefix-sums (scans) are not rank polymorphic. However, we can use segmented scans [8] and pass in a vector of segment descriptors that are all the same length.

4.2.2 Implementation

Accelerate is realised as a type preserving embedded language where closed terms are indexed by their type, and open terms are indexed by their type and their environment. Here is a simplified version of the core AST.

```
data OpenAcc aenv t where
  Avar :: Idx aenv t
       → OpenAcc aenv t
  Alet :: OpenAcc aenv bnd
       → OpenAcc (aenv,bnd) body
  Map  :: Exp aenv (a → b)
       → OpenAcc aenv (Array sh a)
       → OpenAcc aenv (Array sh b)
```

The `Idx aenv t` supplied to `Avar` represents a DeBruijn index of type `t` into the environment `aenv`.

```
data Idx aenv t where
  ZeroIdx :: Idx (aenv,t) t
  SuccIdx :: Idx aenv     t
          → Idx (aenv,s) t
```

A closed term can then be represented with the empty environment.

```
type Acc = OpenAcc ()
```

In order to represent the nested regular arrays that result from the lifting transform, we introduce a type family, `Regular`, that encodes the non- parametric representation. [1]

```
type family Regular t where
  Regular (Array sh e) = Array (sh:.Int)
  Regular (a,b)        = (Regular a, Regular b)
  Regular (a,b,c)
    = (Regular a, Regular b, Regular c)
  ...
```

Of course, our lifting transform will, by necessity change a terms environment. We can capture this change in environment by encoding the lifting context as follows.

```
data Context aenv aenv' where
  Base       :: Context aenv aenv
  Push       :: Context aenv aenv'
             → Context (aenv, t) (aenv', t)
  PushLifted :: Context aenv aenv'
             → Context (aenv,t) (aenv', Regular t)
```

Here, `Push` represents an unlifted variable and `PushLifted` one that was lifted into regular vector space. Our actual lifting transform then takes this form.

```
liftAcc :: Context aenv aenv'
        → OpenAcc aenv t
        → OpenAcc aenv' (Regular t)
```

This concludes the vectorisation. With this function, we can lift any Accelerate expression, including array functions, into a vectorised equivalent that works on regular chunks. We use this to achieve chunked stream transducers as described previously. It remains to show how streams are scheduled.

4.3 Scheduling

If a sequence is regular with element sizes n, a chunk of length k will have size $n * k$. Assuming this size corresponds to the parallel degree required to compute the chunk (we have to be a bit more careful here which we will return to shortly), if n is known, k can be fixed for all chunks in a sequence, consistently saturating the hardware. If on the other hand, a sequence is not regular, the size of elements, and thereby the optimal value of k, varies for each chunk, and the scheduler must re-calculate k in each step.

Right before executing a stream, we perform regularity analysis on the corresponding sequence expression that computes the element size n or reports that the sequence is potentially irregular. If n is found, we set $k = k_{\text{opt}}/n$ where k_{opt} is a constant that defines the optimal chunk size for the given hardware. For SIMD backends, k_{opt} is related to the number of processors and how many scalar elements each processor operates on in each step. We also multiply by a constant factor to reduce scheduling overhead. If n cannot be determined, we currently fall back to purely sequential (static) scheduling with $k = 1$, which is always possible. We plan to improve on this in the future.

As mentioned earlier, size does not always correspond to parallel degree. For example, summing a n-length vector in $\log(n)$ steps will have a parallel degree of at most n in each step and a result size of 1. If we sequence-map a vector-sum over a sequence of vectors s, regardless of s, the resulting sequence is trivially regular with element size 1 (the size of scalars). By fixing the chunk length to $k_{\text{opt}}/1 = k_{\text{opt}}$, one chunk depends on k_{opt} vectors in s. This means that the chunk length of s would have to be at least k_{opt}. However, the vectors of s may be arbitrarily large, and as a result, each step of the stream requires arbitrarily many computational resources, including working memory, which is what we are trying to avoid. The right approach here is to check that s is regular first, and if so, use that elements size to select the chunk length. Furthermore, since we allow mapping any array function over a sequence, the function may internally create large arrays that we have to keep track of as well.

In principal, each producer and consumer in a stream DAG may have its own optimal chunk length. Ideally, streams should be processed at different rates and communicate with each other using buffers that are either filling or draining at different points in time. Static scheduling in a multi-rate context has been covered in the signal processing community [17], but here, the rates are known a priori, and the schedule is then constructed. Our problem is slightly different. We have a trivial schedule (the purely sequential one), but we would like to pick the rates such that each step runs with optimal parallel degree. We take a simpler approach for now: By fixing the chunk length globally across all stream nodes to be the smallest of the optimal chunk lengths of each node, the schedule becomes a single simple loop. The downside is that some nodes may not saturate the hardware, however, the node with the smallest optimal chunk length (the one that will run optimally), is almost always the most costly node to execute.

4.4 Parallel degree and regularity analysis

We perform the regularity analysis by traversing the sequence expression at runtime, prior to executing the corresponding stream DAG. At this point, we have access to the arrays in the surrounding context, that we may use to refine the analysis. The traversal is essentially interpreting array functions using partial arrays as defined by the following type interpretation (encoded as a GADT in the actual implementation):

$$\mathcal{C}[\![\texttt{Array}\ sh\ e]\!] = (\texttt{Maybe}\ sh, \texttt{Maybe}\ (sh \rightarrow e))$$
$$\mathcal{C}[\![(\alpha_1, ..., \alpha_n)]\!] = (\mathcal{C}[\![\alpha_1]\!], ..., \mathcal{C}[\![\alpha_n]\!])$$

We allow an element lookup function in the type of a partial array, but we will only use it if it is a constant time operation, such as if the array is already manifest. The cost interpretation of a an array expression acc of type α in a surrounding array context $aenv$ is a function:

$$\mathcal{C}[\![aenv \vdash acc : \alpha]\!] : \mathcal{C}[\![aenv]\!] \rightarrow (\mathcal{C}[\![\alpha]\!], \texttt{Maybe Int})$$

Here, the result type $(\alpha, \texttt{Maybe Int})$ is a writer monad where the accumulator type `Maybe Int` represents the parallel degree or `Nothing` if it cannot be determined. Two parallel degree are combined by maximum. If even a single intermediate result has an unknown shape, we cannot predict the parallel degree of the expression. Array functions are distinguished syntactically from base array expressions, and the above interpretation does not apply to functions. Instead array functions are interpreted by extending the surrounding context:

$$\mathcal{C}[\![aenv \vdash \lambda acc : \alpha \rightarrow \beta]\!] = \mathcal{C}[\![(aenv, \alpha) \vdash acc : \beta]\!]$$

[1] In our implementation `Regular` is encoded with some slight differences, due to the use of *representation* types, but that is orthogonal to the transform we describe.

The definition of $\mathcal{C}[\![-]\!]$ is:

$$
\begin{aligned}
&\mathcal{C}[\![c]\!]v &&= (\texttt{Just}\ (shape\ c), \texttt{Just}\ (\lambda x.c!x))\\
&\mathcal{C}[\![\textbf{let}\ acc_1 \textbf{in}\ acc_2]\!]v &&= \mathcal{C}[\![acc_1]\!]v \gg= (\lambda x.\mathcal{C}[\![acc_2]\!](v,x))\\
&\mathcal{C}[\![x]\!]v &&= v(x)\\
&\mathcal{C}[\![\lambda acc]\!]v &&= \lambda x.\mathcal{C}[\![acc]\!](v,x)\\
&\mathcal{C}[\![acc_1\ acc_2]\!]v &&= \mathcal{C}[\![acc_2]\!]v \gg= \mathcal{C}[\![acc_1]\!]v\\
&\mathcal{C}[\![\texttt{Map}\ _\ acc]\!]v &&= \mathcal{C}[\![acc]\!]v \gg=\\
& && \quad (\lambda(sh,_).\\
& && \quad\ ((sh, \texttt{Nothing})), fmap\ size\ sh)\\
& && \quad \vdots
\end{aligned}
$$

Here v ranges over partial array contexts $\mathcal{C}[\![aenv]\!]$, *fmap* is the standard functor map over `Maybe`, and ($\gg=$) (bind) is standard bind operator for the writer monad. $size$ computes the size of a shape.

The cost analysis of a sequence expressions requires shape information for the sequence in the surrounding sequence context. For the purpose of detecting regularity, we do not need to track additional information besides the element type of a sequence type $[\alpha]$:

$$\mathcal{C}[\![[\alpha]]\!] = \mathcal{C}[\![\alpha]\!]$$

We assume that the sequence expression is in A-normal form. Note that we are analysing the unlifted array functions before vectorisation. The analysis for a sequence expression seq in surrounding sequence context $senv$ and array context $aenv$ has the following signature:

$$
\begin{aligned}
\mathcal{C}[\![aenv, senv \vdash seq : \alpha]\!] :\ & \mathcal{C}[\![aenv]\!]\\
\rightarrow\ & \mathcal{C}[\![senv]\!]\\
\rightarrow\ & (\mathcal{C}[\![\alpha]\!], \texttt{Maybe Int})
\end{aligned}
$$

It is defined as follows:

$$
\begin{aligned}
&\mathcal{C}[\![\texttt{ToSeq}\ div\ acc]\!]v\ _\\
&\quad = (sl, fmap\ size\ sl)\\
&\quad \text{where}\\
&\quad\quad ((sh,_),_) = \mathcal{C}[\![acc]\!]v\\
&\quad\quad sl = fmap\ (sliceShape\ div)\ sh\\
&\mathcal{C}[\![\texttt{StreamIn}\ xs]\!]_\ _ = \texttt{Nothing}\\
&\mathcal{C}[\![\texttt{MapSeq}\ f\ x]\!]v\ w = \mathcal{C}[\![f]\!](v, w(x))\\
&\mathcal{C}[\![\texttt{ZipWith}\ f\ x\ y]\!]v\ w = \mathcal{C}[\![f]\!](v, w(x), w(y))\\
&\mathcal{C}[\![\texttt{FoldSeqFlatten}\ f\ acc\ x]\!]v\ w\\
&\quad = \mathcal{C}[\![f]\!](v, (sh_a, \texttt{Nothing}), shs, elts)\\
&\quad \text{where}\\
&\quad\quad ((sh_a, \texttt{Nothing}),_) = \mathcal{C}[\![acc]\!]v\\
&\quad\quad (sh_x,_) = w(x)\\
&\quad\quad shs = (\texttt{Just}\ (\texttt{Z :.}\ 1), fmap\ unit\ sh_x)\\
&\quad\quad elts = (fmap\ ((\texttt{Z :.}) \circ size)\ sh_x, \texttt{Nothing})
\end{aligned}
$$

Here w ranges over partial sequence context $\mathcal{C}[\![senv]\!]$ tracking sequence element sizes. *sliceShape* computes the slice shape from a given source shape and division strategy. The array argument acc in both the `ToSeq` case and the `FoldSeqFlatten` case are not evaluated in each step of the sequence. We therefore discard the parallel degree reported by the size analysis on these - We are only interested in the shapes.

The current definition of the `FoldSeqFlatten` case is admittedly faulty. It performs the analysis using the shape of the initial accumulator. If the accumulator grows, such as when folding with vector append, the actual parallel degree may very well be much larger than the parallel degree we report here. A correct definition should check that the shape of the initial accumulator is equal to the shape of the new accumulator, and only report a parallel degree if that is the case. However, that would cause the analysis to fail for our current definition of `FromSeq`. We therefore omit this check until we have a better implementation of `FromSeq` or until we have better support for dynamic scheduling.

5. Evaluation

In order to evaluate the performance of our implementation, we give the results from two sets of benchmarks. Firstly, three smaller benchmarks where we compare the performance of accelerate sequences against arrays in order to show that no performance is lost by using sequences. Secondly, we demonstrate two larger scale applications, comparing them both with Accelerate array implementations and other CPU implementations. All benchmarks were run on a single Tesla K20c (compute capability 3.5, 13 multiprocessors = 2496 cores at 705.50 MHz, 5 GB RAM). The host machine has 4 8-core Xeon E5-2650 CPUs (64-bit, 2.00GHz, 64GB ram, with hyperthreading). Our benchmark times include time to transfer the data onto the GPU, the execution time, and the time to transfer data back to the host, but not compilation time. The fact that Accelerate is online compiled is orthogonal to what we present here, and compiled kernels are cached to prevent unnecessary recompilation.

5.1 Dot product

This is simply the example we show in Section 3 against the version that does not use sequences. While this example is very simple, it helps highlight that there is no significant loss of performance due to the overheads associated with scheduling sequence computations. At the ideal chunk-size, 2^{24} scalar elements, performance is essentially equal. It is worth noting that this is significantly less than the total size of the data which is 100 million floating point values in each input vector. It is not till 2^{27}, that the chunk size covers all input.

In the second graph, where the chunk size is fixed at 2^{24}, we see that normal Accelerate runs out of memory with input vectors of 700 million floating point numbers. However, with sequences we are able to process much larger inputs with no noticeable overhead. That is until it exceeds 3.5 billion in which case it runs out of physical memory on the host and pages start getting swapped to disk.

5.2 MaxSum

Here we demonstrate how, even when applying two separate reductions over a sequence, only one pass is ever made over the input. The program is simply:

```
maxSumSeq :: Vector Float
          → Acc (Scalar Float, Scalar Float)
maxSumSeq xs = collect
             $ lift ( foldSeqE (+) 0 xs'
                    , foldSeqE max 0 xs')
  where xs' = toSeqE xs
```

We compare it against a version written without sequences.

```
maxSumSeq :: Vector Float
          → Acc (Scalar Float, Scalar Float)
maxSumSeq xs = lift (fold (+) 0 xs, fold max 0 xs)
```

Here, `lift` converts `(Seq a, Seq b)` into `Seq (a,b)`, or `(Acc a, Acc b)` into `Acc (a,b)`, depending on the context.

Even though only one pass is made over the input as a whole, this example does not outperform normal Accelerate due to limitations in Accelerate's fusion system. Because `foldSeqE` applies `fold` over each chunk, two traversals of each chunk is made. The sort of *horizontal* fusion that would resolve this would be advantageous in this instance.

Once again, we apply this function over 100 million floating point values.

5.3 MVM

This is the matrix-vector multiplication example shown earlier. This is not just a simple reduction, but uses a sequence map, so requires vectorisation. Once again, we compare it to a version written without sequences.

```
mvm :: Acc (Matrix Float) → Acc (Vector Float)
    → Acc (Vector Float)
mvm mat vec
  = let h = fst (unindex2 (shape mat))
    in fold (+) 0
    $ zipWith (*) mat (replicate (lift (Z:.h:.All)) vec)
```

The net result in this case is that, after subsequent optimisation passes, both examples produce almost identical code. The only difference being index manipulation. From the graph it can be observed that, even with very small chunk sizes, the version with sequences runs in close to the same time as the version without. The input is a matrix of 10000×10000 floating point values.

5.4 MD5 hash

The MD5 message-digest algorithm [25] produces a 128-bit cryptographic hash. For the purposes of establishing the ideal chunk size we apply this algorithm to each word in a dictionary of 64 million words, attempting simple password recovery. Like MVM above, we generate very similar code to a version written without sequences. However, in this case, slight differences in index manipulation actually work in our favour, giving us minor improved performance over the contender. We also compare our results to that of Hashcat, a CPU-based password recovery tool.

To demonstrate the effectiveness of Accelerate sequences or out-of-core algorithms, we also run this benchmark against dictionaries of varying size. As can be seen, if sequences are not used, only a dictionary of up to 75 million words will fit in the GPUs memory. If they are used, however, we can use dictionaries up to an almost arbitrary size. Although, like the dot product example, if the dictionary is larger than can fit in physical host memory then there is slowdown due to paging. This occurs at around 400 million words.

5.5 PageRank

While our current restriction on sequences being regular is not ideally suited to graph processing, we can still implement graph algorithms, like PageRank [24] by changing the way we represent a graph. In this case, we represent it as a sequence of links (edges). This is a much heavier, in terms of space, than a representation using a sparse matrix in compressed sparse row (CSR) format, however it is still performant. Comparing our results to that of a CPU-based implementation written in Repa [14, 19], we see that while it can use a more space-efficient representation, we still outperform it in terms of overall speed.

For the benchmark, we use a dump of the Wikipedia link graph that contains approximately 130 million links.

6. Related Work

There are a number of languages which aim at facilitating GPU programming in the presence of data streams. For example, the Brook[5] stream programming language is an extension of C with support for data parallel computations. BrookGPU [6] is an implementation of a subset of Brook which targets GPUs.

Brook uses only streams to express parallelism.

Sponge[13] is a compilation framework for GPUs using the synchronous data flow streaming language StreamIt[27].

Both Sponge and BrookGPU are based on a very different programming model. Starting from a stream programming model, the compiler and runtime system tries to execute the code efficiently on a GPU architectures. We provide the programmer both with parallel arrays, which support a much wider range of parallel operations, and more restricted streams. This distinction enables the programmer to design algorithmic solutions for a problem better tailored towards the actual architecture, while still providing a high level of abstraction.

Though it does not target GPUs, the Proteus[10] language uses an idea similar to chunking to restrict the memory requirements of vectorisation based implementations of nested data-parallel programs.

While there are a number of domain specific languages to support GPU programming, some of them also embedded in Haskell, [1, 9, 11, 16, 21, 26] to name some, none of them currently, to the best of our knowledge, support streams.

A conceptual stream model for NESL has been presented in previous work [20]. The model is motivated by the lack of a good realizable space cost model, and it is conceptually similar to ours; Sequences are exposed at the source level and execute in streams of bounded chunks. The model is more general than ours in the sense that it allows nested sequences. However, it has yet to be implemented with a high-performance backend. Furthermore, the work presented here features more operations on multi-dimensional arrays.

7. Future work

The work presented in this paper is a first step towards full streaming support for collection-oriented parallel languages targeting GPUs. Our next steps will be to lift some of the current restrictions of the model, as well as further improving the performance of the generated code and runtime system. In particular, we are planning to address the following issues in the near future:

- Chunked execution of irregular sequences: This would allow us to express more algorithms over irregular data, like graph processing, but will require more sophisticated scheduling strategies than in the regular case.
- More combinators: We currently only support a minimal set of combinators in Accelerate, and there are a number of obvious additions, as for example scans on non-scalar sequences which we will add.
- Automatic sequentialisation: As mentioned in the introduction, the whole point of exposing sequences to the programmer is that optimal automatic sequentialisation is generally undecidable. That does not mean that the compiler should not attempt it however. Guided by a size analysis, the compiler could opportunistically transform array expressions into equivalent sequence traversals in the presence of very large intermediate arrays.

With respect to performance improvements:

- We do not yet attempt to transfer the data for the next chunk while we process the current one. This overlapping of commu-

nication and computation could potentially give significant improvements to overall runtime [12].

- As we have already mentioned, our model lends itself very well to multi-gpu support via pipeline-parallelism. This is our ultimate aim, but it does carry with it challenges in regards to fusion of sequence computations– i.e. that sequence operations should not be always fused, but rather split up amongst devices where possible.

Acknowledgments

The authors wish to acknowledge Trevor L. McDonell for his invaluable assistance and advice.

References

[1] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. In *ICFP: International Conference on Functional Programming*. ACM, 2012.

[2] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Nov. 1990.

[3] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, Carnegie Mellon University, 1995.

[4] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. In *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*, pages 575–585. IEEE, 1988.

[5] I. Buck. Brook language specification. *Outubro*, 2003. URL `http://merrimac.stanford.edu/brook`.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM. . URL `http://doi.acm.org/10.1145/1186562.1015800`.

[7] 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*. ACM, 2011.

[8] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proc. of Supercomputing*. IEEE Computer Society Press, 1990.

[9] K. Claessen, M. Sheeran, and J. Svensson. Obsidian: GPU programming in Haskell. In *IFL: Implementation and Application of Functional Languages*, 2008.

[10] 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.

[11] C. Elliott. Programming graphics processors functionally. In *Haskell Workshop*. ACM Press, 2004.

[12] J. GóMez-Luna, J. M. GonzáLez-Linares, J. I. Benavides, and N. Guil. Performance models for asynchronous data transfers on consumer graphics processing units. *J. Parallel Distrib. Comput.*

[13] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: Portable stream programming on graphics engines. *SIGARCH Comput. Archit. News*, 39(1):381–392, Mar. 2011. ISSN 0163-5964. . URL `http://doi.acm.org/10.1145/1961295.1950409`.

[14] 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*. ACM, 2010.

[15] 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.

[16] B. Larsen. Simple optimizations for an applicative array language for graphics processors. In *DAMP: Declarative Aspects of Multicore Programming*. ACM, 2011.

[17] E. A. Lee and D. G. Messerschmi'tt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 2(36), 1987.

[18] R. Leshchinskiy, M. M. Chakravarty, and G. Keller. Higher order flattening. In *Computational Science–ICCS 2006*, pages 920–928. Springer, 2006.

[19] B. Lippmeier, M. Chakravarty, G. Keller, and S. Peyton Jones. Guiding parallel array fusion with indexed types. In *Haskell Symposium*. ACM, 2012.

[20] F. M. Madsen and A. Filinski. Towards a streaming model for nested data parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2381-9. . URL `http://doi.acm.org/10.1145/2502323.2502330`.

[21] G. Mainland and G. Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *Haskell Symposium*. ACM, 2010.

[22] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP: International Conference on Functional Programming*, Sept. 2013.

[23] NVIDIA. CUDA C Programming Guide, 2012.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[25] R. Rivest. The md5 message-digest algorithm. 1992.

[26] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Odersky, and K. Olukotun. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL'13*. ACM, 2013.

[27] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer, 2002.

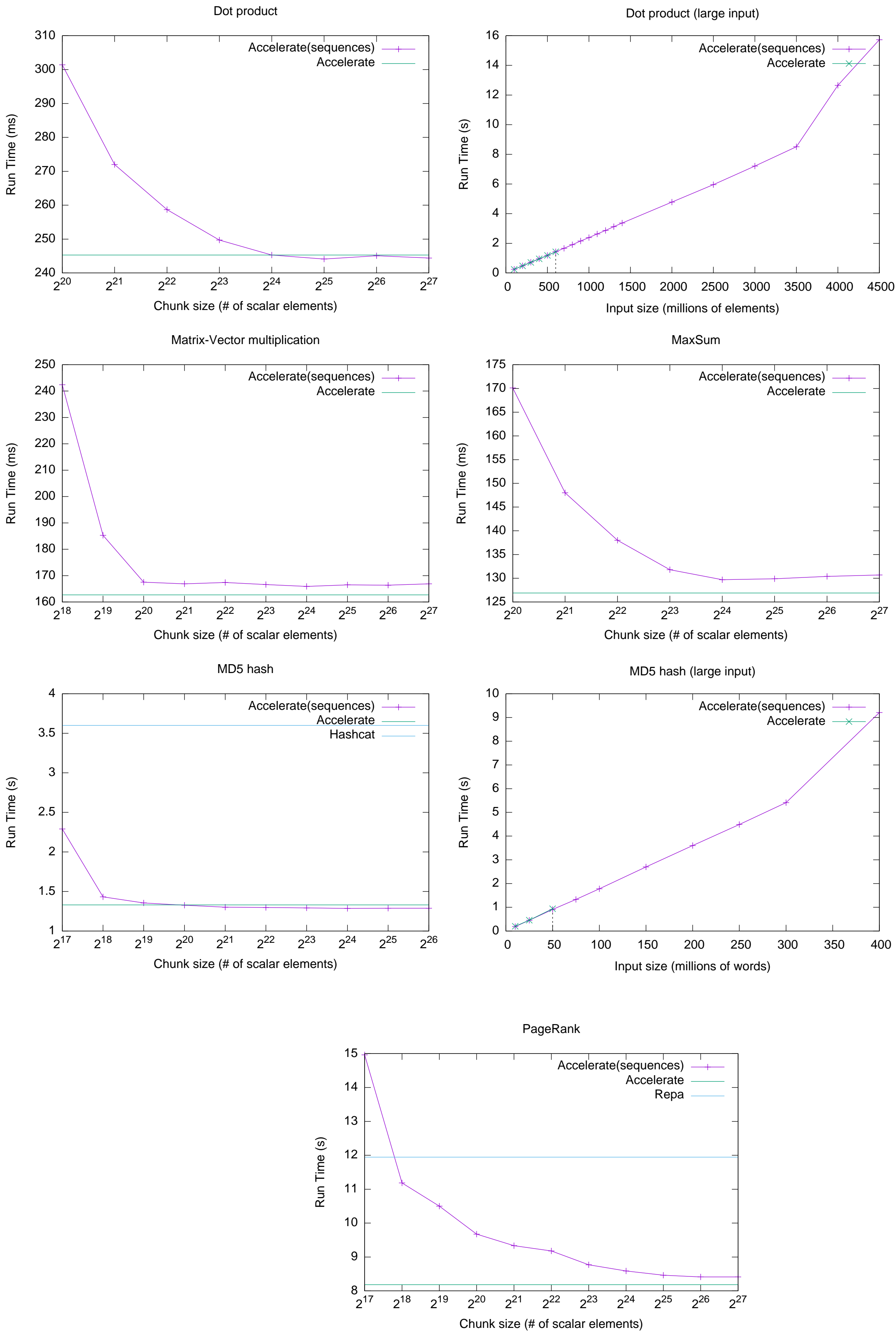

Figure 1. Benchmark results