Accelerating Naperian functors

EPSRC Summer Internship Report

Nick Hu
University of Oxford
nick.hu@cs.ox.ac.uk

ABSTRACT

Naperian functors (Gibbons 2017) provide a dimensionally-polymorphic abstract representation of structured multidimensional data, e.g. matrices in n-dimensions. Such structures are frequently utilised in data-intensive computing, which is common in all experimental sciences. Such structures themselves are sufficiently abstract such that they do not preclude parallel, high-performance instances. This work explores integrating the genericity of programming with Naperian functors with the existing Haskell Accelerate (Chakravarty et al. 2011) library, an embedded array language capable of generating heavily optimised parallel machine code and CUDA code via LLVM (McDonell et al. 2015).

1 INTRODUCTION

A Naperian functor is one which holds shape

```
class Functor f => Naperian f where
    type Log f
  lookup :: f a -> (Log f -> a)
  tabulate :: (Log f -> a) -> f a
  positions :: f (Log f)
  tabulate h = fmap h positions
  positions = tabulate id
```

such that `tabulate . lookup = id = lookup . tabulate`, taking liberties for the polymorphic type of `id`.

From this interface, we can define transposition in a very general way (Figure 1), and indeed this typechecks. Informally, it produces a function to index the outer and then the inner dimension, flips the first two arguments, and then tabulates from inside out. This produces a structure with its dimensions swapped, and completely captures transposition.

Note that due to `fmap` laws and the above identity, `tabulate . fmap tabulate . fmap lookup . lookup = id`.

This work concerns the application of this abstraction to data structures from the GPU domain-specific language Accelerate (Chakravarty et al. 2011).

2 HYPERCUBOIDS AND ACCELERATE ARRAYS

Gibbons (2017) defines a polymorphic nested datatype providing the structure necessary to represent arbitrary rank hypercuboids (tensors):

```
data Hyper :: [Type -> Type] -> Type -> Type where
  Scalar   :: a -> Hyper '[] a
  Prism    :: Hyper fs (f a) -> Hyper (f ': fs) a
```

A `Hyper` at the type level contains a list of type constructors, which give the dimensions of a hypercuboid, from innermost dimension to outermost, and also a base type for the contents. For example, the `Hyper` representation of the two-dimensional `Int` matrix

```
m :: Vector 2 (Vector 3 Int)
m = [ [ 1, 2, 3],
      [ 4, 5, 6] ]
```

is

```
h :: Hyper '[Vector 3, Vector 2] Int
h = Prism . Prism $ Scalar m
```

Functions are provided to generically lift and align operations for hypercuboids of different but reconcilable shapes in an APL-like manner.

`Hyper` is not hugely dissimilar to the Accelerate `Array`, which has a representation for the same matrix as[1]:

```
a :: Array (Z :. Int :. Int) Int
a = A.fromList (Z :. 2 :. 3) [1, 2, 3, 4, 5, 6]
```

An Accelerate `Array` is essentially a flat list of elements, and a description of its shape (aptly contained within typeclass called `Shape`). However, the fundamental difference is that at the type level, `Array` only represents rank and not extent — the type signature of `a` allows us to infer that it is some two-dimensional matrix, but we require the type signature of `h` to discern that it is a 3×2 matrix.

We would like to create some kind of correspondence between `Hyper` and `Array`, and to this aim we introduce the `Flat` constructor which essentially wraps an `Array`, storing the type information required for its equivalent `Hyper` form.

A `Hyper` can be converted to an `Array` via `A.fromList`, given a `Shape` and its contents, which can be extracted as follows:

```
elements :: Hyper fs a -> [a]
elements (Scalar x) = [x]
elements (Prism x) = concatMap F.toList $ elements x
```

For the construction of the `Shape`, observe that the types in an `Array`'s `Shape` describe the index along dimension, for which we assert that only `Int` is effective as a target for GPUs. Thus, every `Hyper` dimension list can be safely mapped to a snoc-list of `Z` and `Int`.

At the type level, by use of a closed type family[2]:

```
type family ToShape (f :: [Type -> Type]) where
  ToShape '[] = Z
  ToShape (x ': xs) = ToShape xs :. Int
```

[1] We will refer to the Accelerate version of `fromList` as `A.fromList`, and the `OverloadedLists` version as `fromList` throughout. Similarly, `toList` refers to the version from `OverloadedLists`, while `F.toList` refers to the version from `Data.Foldable`.

[2] Conveniently, we also assert that every dimension of the hypercuboid has strictly positive extent.

```
transpose :: (Naperian f, Naperian g) => f (g a) -> g (f a)
transpose = tabulate . fmap tabulate . flip . fmap lookup . lookup
-- with following specialisations
fmap lookup :: (Log f -> g a) -> Log f -> Log g -> a -- (->) e Functor
fmap tabulate :: (Log g -> Log f -> a) -> Log g -> f a -- (->) e Functor
lookup :: f (g a) -> (Log f -> g a)
fmap lookup . lookup :: f (g a) -> (Log f -> Log g -> a)
flip . fmap lookup . lookup :: f (g a) -> (Log g -> Log f -> a)
fmap tabulate . flip . fmap lookup . lookup :: f (g a) -> (Log g -> f a)
tabulate . fmap tabulate . flip . fmap lookup . lookup :: f (g a) -> g (f a)
```

Figure 1: Naperian transposition

and at the term level:

```
toShape :: Hyper fs a -> ToShape fs
toShape (Scalar _) = Z
toShape h@(Prism x) = toShape x :. topDimension h

topDimension :: Hyper fs a -> Int
topDimension (Prism x) = F.length . head $ elements x
```

`Flat` can now be defined as follows:

```
data Flat fs a where
  Flat :: (Shape (ToShape fs))
       => Array (ToShape fs) a -> Flat fs a
```

The closed type family is a Haskell idiom for a total function on types: given a type-level list of kind `Type -> Type` (i.e. single-argument type constructors), `ToShape` computes the corresponding `Shape`.

Combining the above, any `Hyper fs a` can be transformed into an `Array (ToShape fs) a` (provided that `a` is an Accelerate element type `Elt`):

```
toArray :: (Shape (ToShape fs), Elt a)
        => Hyper fs a -> Array (ToShape fs) a
toArray h = A.fromList (toShape h) (elements h)
```

The composition gives:

```
flatten :: (Shape (ToShape fs), Elt a)
        => Hyper fs a -> Flat fs a
flatten = Flat . toArray
```

`ToShape` is non-injective, but this composite allows all of the information of `fs` to be preserved.

3 GOING BACK WITH OVERLOADED LISTS

GHC's `OverloadedLists` mechanism, which given an instance `IsList l`, provides the following:

$$\texttt{[Item a]} \underset{\texttt{toList}}{\overset{\texttt{fromList}}{\rightleftarrows}} \texttt{l}$$

Given a functor `f` with the type bounds `IsList (f a)` and `Item (f a) ~ a`:

$$\texttt{[a]} \underset{\texttt{toList}}{\overset{\texttt{fromList}}{\rightleftarrows}} \texttt{f a}$$

Every `Dimension` (an instance of `Naperian`, `Applicative`, and `Traversable`) gives rise to an instance of `IsList`.

Hence `OverloadedLists` can be leveraged to provide a way to turn an `Array` back into a specific `Hyper`, by first building the `Array`'s element list into a nested list with structure matching that of the desired `Hyper` and then mapping `fromList` along the innermost nesting outwards:

```
class Hyperise fs a where
  hyperise :: ToShape fs -> [a] -> Hyper fs a

instance Hyperise '[] a where
  hyperise Z [x] = Scalar x

instance ( Dimension f
         , Hyperise fs (f a)
         , Item (f a) ~ a
         , IsList (f a)
         ) =>
        Hyperise (f ': fs) a where
  hyperise (ss :. s) xs =
    Prism . hyperise ss $ map fromList (chunksOf s xs)
```

[3]

Now, we just need a method of extracting the `Shape` and elements of an `Array`, which is conveniently provided by `arrayShape` and `A.toList` respectively. This allows for the complete definition of `unflatten`:

```
getArray :: Flat fs a -> Array (ToShape fs) a
getArray (Flat xs) = xs

unflatten :: (Hyperise fs a, Shape (ToShape fs))
          => Flat fs a -> Hyper fs a
unflatten f = hyperise (arrayShape arr) (A.toList arr)
  where
    arr = getArray f
```

4 HYPER-FLAT EMBEDDING

We define the `Flat`s created via `flatten` as the *well-formed* `Flat`s, and go on to prove that there is a correspondence between `Hyper` and the well-formed `Flat`s. See Figure 2 for a commutative diagram describing the relationships between the different types.

[3] `chunksOf :: Int -> [a] -> [[a]]` is provided by `Data.List.Split`, and given an integer `n` and a list `xs`, produces a list of `n`-length segments of `xs` in order. Later on, we go to show that it serves as a kind of inverse for `concat` in this context.

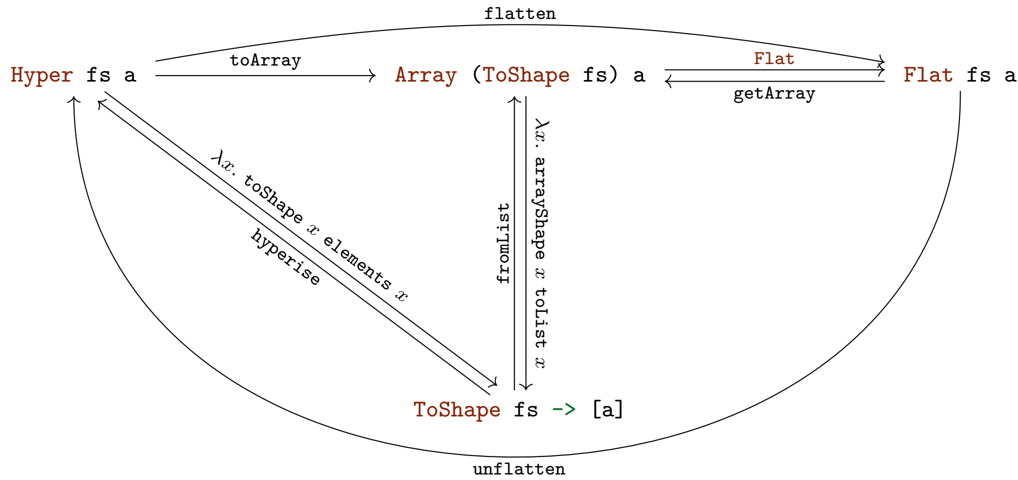

Figure 2: Embedding of Hyper into Flat

LEMMA 4.1. *Given nonempty* `xs :: Foldable t => [t a]` *such that each* `x` *in* `xs` *has identical length:*

```
xs = map fromList (chunksOf (F.length (head xs))
                             (concatMap F.toList xs)
```

PROOF. By structural induction over `xs`.
Case `[x]`:

```
concatMap F.toList [x]
= -- concatMap
concat (map F.toList [x])
= -- map.2
concat [(F.toList x)]
= -- concat.2
F.toList x

map fromList (chunksOf (F.length (head [x]))
                       (concatMap F.toList [x])
= -- head, equation above
map fromList (chunksOf (F.length x) (F.toList x))
= -- chunksOf (input list size is equal to chunking size)
map fromList [F.toList x]
= -- map
[fromList (F.toList x)]
= -- lemma: fromList . F.toList = id
[x]
```

Case `(x:xs)` with `xs` nonempty:

```
concatMap F.toList (x:xs)
= -- concatMap
concat (map F.toList (x:xs))
= -- map.2
concat (F.toList x : map F.toList xs)
= -- concat.2
F.toList x ++ concatMap F.toList xs

map fromList (chunksOf (F.length (head (x:xs)))
                       (concatMap F.toList (x:xs))
= -- head, equation above
map fromList
  (chunksOf (F.length x)
            (F.toList x ++ concatMap F.toList xs))
= -- chunksOf (first chunk is x)
map fromList
  (F.toList x : (chunksOf (F.length x)
                          (concatMap F.toList xs)))
= -- map.2
fromList (F.toList x)
  : map fromList (chunksOf (F.length x)
                           (concatMap F.toList xs))
= -- property 1 ⟹ F.length x = F.length (head xs)
fromList (F.toList x)
  : map fromList (chunksOf (F.length (head xs))
                           (concatMap F.toList xs))
= -- inductive hypothesis
fromList (F.toList x) : xs
= -- lemma: fromList . F.toList = id
x : xs
```

□

THEOREM 4.2. *There is an isomorphism between* `Hyper fs a` *and the well-formed* `Flat fs a`.

PROOF. For the direction `Hyper fs a -> Flat fs a`, the proof is given by structural induction over `Hyper`.
Base case `Scalar x`:

```
(unflatten . flatten) (Scalar x)
= -- composition
unflatten (flatten (Scalar x))
= -- flatten
unflatten (Flat (toArray (Scalar x))
= -- toArray
unflatten (Flat (A.fromList (toShape (Scalar x))
                (elements (Scalar x))))
= -- toShape.1, elements
unflatten (Flat (A.fromList Z [x]))
= -- unflatten
hyperise (arrayShape (getArray (Flat (A.fromList Z [x]))))
         (A.toList (getArray (Flat (A.fromList Z [x]))))
= -- lemma: getArray . Flat = id
hyperise (arrayShape (A.fromList Z [x]))
         (A.toList (A.fromList Z [x]))
= -- lemmas: arrayShape (A.fromList sh _) = sh,
```

```
  --         A.toList (A.fromList _ xs) = xs
hyperise Z [x]
= -- hyperise
Scalar x
= -- id
id (Scalar x)
```

For the inductive case, it is useful to transform the inductive hypothesis into a more useful form; suppose it holds for `x`, then:

```
x
= -- inductive hypothesis
(unflatten . flatten) x
= -- composition
unflatten (flatten x)
= -- flatten
unflatten (Flat (toArray x))
= -- toArray
unflatten (Flat (A.fromList (toShape x) (elements x)))
= -- unflatten
hyperise (arrayShape (getArray (Flat
           (A.fromList (toShape x) (elements x)))))
         (A.toList (getArray (Flat
           (A.fromList (toShape x) (elements x)))))
= -- lemma: getArray . Flat = id
hyperise (arrayShape (A.fromList (toShape x) (elements x)))
         (A.toList (A.fromList (toShape x) (elements x)))
= -- lemmas: arrayShape (A.fromList sh _) = sh,
  --         A.toList (A.fromList _ xs) = xs
hyperise (toShape x) (elements x)
```

Inductive case `Prism x`:

```
(unflatten . flatten) (Prism x)
= -- composition
unflatten (flatten (Prism x))
= -- flatten
unflatten (Flat (toArray (Prism x)))
= -- toArray
unflatten (Flat (A.fromList
                 (toShape (Prism x)) (elements (Prism x))))
= -- toShape.2
unflatten (Flat (A.fromList
                 (toShape x :. topDimension (Prism x))
                 (elements (Prism x))))
= -- unflatten
hyperise (arrayShape (getArray (Flat
           (A.fromList (toShape x :. topDimension (Prism x))
                       (elements (Prism x))))))
         (A.toList (getArray (Flat
           (A.fromList (toShape x :. topDimension (Prism x))
                       (elements (Prism x))))))
= -- lemma: getArray . Flat = id
hyperise (arrayShape
           (A.fromList (toShape x :. topDimension (Prism x))
                       (elements (Prism x))))
         (A.toList
           (A.fromList (toShape x :. topDimension (Prism x))
                       (elements (Prism x))))
= -- lemmas: arrayShape (A.fromList sh _) = sh,
  --         A.toList (A.fromList _ xs) = xs
hyperise (toShape x :. topDimension (Prism x))
         (elements (Prism x))
= -- hyperise
Prism (hyperise (toShape x)
                (map fromList (chunksOf
                  (topDimension (Prism x))
                  (elements (Prism x)))))
= -- lemma chunksOf and concatMap
Prism (hyperise (toShape x) (elements x))
= -- inductive hypothesis
Prism x
= -- id
id (Prism x)

map fromList (chunksOf (topDimension (Prism x))
                       (elements (Prism x)))
= -- topDimension, elements
map fromList (chunksOf (F.length (head (elements x)))
                       (concatMap F.toList (elements x)))
= -- lemma below
elements x
```

As the only way to create well-formed inhabitants of `Flat fs a` is via the `flatten` function, the reverse direction is trivial:

```
(flatten . unflatten) (flatten h)
= -- composition
(flatten . unflatten . flatten) h
= -- forward direction
(flatten . id) h
= -- id
id (flatten h)
```

□

This suffices to show that there is an adequate embedding of all `Hyper` terms into `Flat`.

For a `Flat` which is not well-formed, consider the construction

```
λ: let a = A.fromList (Z :. 6 :. 4) [1..]
              :: Array (Z :. Int :. Int) Int
λ: a
Matrix (Z :. 6 :. 4)
  [ 1, 2, 3, 4,
    5, 6, 7, 8,
    9,10,11,12,
   13,14,15,16,
   17,18,19,20,
   21,22,23,24]
λ: let f = Flat a :: Flat '[Vector 3, Vector 8] Int
λ: unflatten f
*** Exception: list cast to vector of wrong length
```

The problem arises due to Accelerate's `Array` type only carrying dimensionality but not extent at the type level. There is an inconsistency between the shape of the data in the `Flat` type and the shape of the `Array` term `a`, making the `Flat` term `f` not well-formed. The `flatten` function carries over the shape of a `Hyper`, automatically fulfilling this consistency criteria.

4.1 Embed

As Accelerate is more like a domain specific language which compiles to some backend, rather than a library running in

the Haskell runtime, Accelerate itself is not able to operate on `Flat`s themselves in a useful way: while they carry the information of an Accelerate `Array`, Accelerate must first 'lift' its inputs into the domain of embedded expressions before it can begin processing: into either the `Acc` or `Exp` constructor, corresponding to lifted array and scalar types respectively. Accelerate provides the functions[4]

```
use :: (Shape s, Elt t) => Array s t -> Acc (Array s t)
constant :: Elt t => t -> Exp t
```

to do this.

In general, lifted data cannot be unlifted without evaluation; conceptually, lifted data corresponds to an abstract representation on which Accelerate is able to perform optimisations on like stream fusion. In that sense, the values which correspond to notional unlifting are in general not available until execution time, as Accelerate itself utilises a backend (say, LLVM) to generate code, which cannot directly interface with Haskell due to this abstraction barrier. Instead, for each backend, the

```
run :: (Shape s, Elt t) => Acc (Array s t) -> Array s t
```

function is provided[5], which when called generates code for the target architecture, compiles it and then executes it.

Hence we introduce a constructor `Embed` to carry `Flats` which have lifted data:

```
data Embed fs a where
  Embed :: (Shape (ToShape fs))
        => Acc (Array (ToShape fs) a) -> Embed fs a

embed :: Elt a => Flat fs a -> Embed fs a
embed (Flat xs) = Embed (use xs)
```

This, along with some wrappers around primitive Accelerate functions, allows us to lift `Hyper`s into the domain of Accelerate and manipulate it just as efficiently.

To avoid introducing `Embed`, one might think to instead wrap computations concerning `Flats` with calls to `use` and `run`, which conceptually loads data into the hardware-accelerated backend (say, the GPU), runs the computation, and then retrieves the result, locally for every calculation. However, this is inherently flawed for two reasons:

1. this would prevent Accelerate from being able to perform many of its optimisations, like stream fusion;
2. communication overhead is extremely high — the rate of transfer across the system bus on modern hardware is far slower than actual calculation;

doing this in practice would result in a program which is too slow for all but the most trivial programs.

5 MONONAPERIAN

Another area in which performance can be gained is the memory architecture of GHC. Due to parametric polymorphism, all `Functor`s contents are represented by a individual pointers to the heap — each element is a *boxed* value; for a vector of integers, this is less than desirable, and creates noticable overhead.[6]

If we are to specialise for vectors of primitave types, we would like to build our `Vector` type atop something like `Data.Vector.Unbox`, which stores unboxed values like integers of type `Int#` (`long int`s). However, while `Data.Vector.Unbox` exposes a similar interface to `Data.Vector`, its implementation is based on type families to pick a specialised representation for every element type (known as *monomorphic specialisation*). As such, its elements need to be instances of the `Unbox` type class (for which instances for standard primitave types are provided by the `vector` package), and also `Data.Vector.Unbox` is not a `Functor` (which is a polymorphic container). This means that our existing machinery will not work as-is.

In order to overcome this, the `mono-traversable` package provides the typeclass `MonoFunctor` for working with monomorphic containers:

```
class MonoFunctor mono where
    omap :: (Element mono -> Element mono)
         -> mono -> mono
```

Categorically, where an ordinary `Functor` can be thought of as an endofunctor from **Hask** to itself, a `MonoFunctor` can be thought of as a functor from a one-object subcategory of **Hask** to another one-object subcategory of **Hask**, through the inverse image of the `Element` type function. `Element` is a type function which takes a monomorphic container (such as `Text`) to the type of its elements (`Char`).

A monomorphic container does not expose a type variable for its `Element` type, so its implementation can be fixed. Most importantly, it knows exactly what size its elements are and so can optimise accordingly.

We describe the `MonoNaperian` typeclass as follows:

```
class MonoFunctor f => MonoNaperian f where
  type MonoLog f
  olookup :: f -> MonoLog f -> Element f
  otabulate :: (MonoLog f -> Element f) -> f
```

This interface is not as nice to work with, as there is no way to implement the equivalent of `positions` because the `MonoFunctor f` is only able to contain elements of type `Element f` and not `MonoLog f`.

Fortunately, this is not crucial to the essence of a Naperian functor, and we can still implement the three key array operations:

```
-- replication
oreplicate :: MonoPointed f => Element f -> f
oreplicate = opoint

-- transposition
otranspose :: ( MonoNaperian f
              , MonoNaperian (Element f)
              , MonoNaperian g
              , MonoNaperian (Element g)
```

[4]Slightly simplified compared to the real thing.

[5]Simplified again. `Exp` scalars embed into `Acc` arrays too.

[6]For a detailed discussion on boxing, along with a GHC proposal to combine unboxing with polymorphism, see Eisenberg and Peyton Jones (2017).

```
              , Element (Element f)
                ~ Element (Element g)
              , MonoLog (Element f) ~ MonoLog g
              , MonoLog f ~ MonoLog (Element g) )
           => f -> g
otranspose = otabulate . fmap otabulate
           . flip . fmap olookup . olookup

-- zipping
ozipWith :: MonoNaperian f
         => (Element f -> Element f -> Element f)
         -> f -> f -> f
ozipWith f xs ys = otabulate (\i -> f (olookup xs i)
                                      (olookup ys i))
```

The type signature of otranspose is much longer, but is essentially the same as before — take careful note that the fmaps are for the (->) e instance and can be replaced by function composition. Zipping this time is implemented leveraging the MonoNaperian interface, as we lack an analogue to the Applicative <*>. In fact, there is no MonoApplicative, only MonoPointed:

```
class MonoPointed mono where
    opoint :: Element mono -> mono
```

which provides the equivalent of Applicative's pure without <*>, and this achieves replication.

5.1 MonoHyper

The notion of a dimension of along a monomorphic type is as before[7]:

```
class (MonoPointed f, MonoNaperian f, MonoTraversable f)
      => MonoDimension f where
  osize :: f -> Int
  osize = length . otoList
```

As before, we define a polymorphic nested datatype to represent monomorphic arbitrary rank hypercuboids:

```
data MonoHyper :: [Type] -> Type -> Type where
  OScalar :: a -> MonoHyper '[] a
  OPrism :: (MonoDimension f)
         => MonoHyper fs f
         -> MonoHyper (f ': fs) (Element f)

type instance Element (MonoHyper fs a) = a
```

To pursue this avenue further, one would need to overcome the problem of nesting unboxed vectors inside one-another, as a Data.Vector.Unbox is not an instance of Unbox. Arguably, this is crucial to the usefulness of hypercuboids.

6 CASE STUDY: k-MEANS CLUSTERING

The k-means clustering problem, in machine learning and data mining, is the problem of partitioning a dataset into k clusters, with each point associated to the cluster with the nearest mean. Visualising each point as a point in n-dimensional space, a cluster of points is merely a set, and we call its mean (or 'centroid') the virtual point given by the point-wise mean along each dimension. Intuitively, this is the problem of splitting a dataset into k groups based on similarity.

Distance is typically given by the squared Euclidean distance, as this avoids a computationally expensive square root operation and preserves the 'closer' relation[8]. The problem of finding the optimal clusters, such that the sum over distances from each centroid to their associated points is minimised, is NP-hard; however, there is a standard algorithm (Lloyd's) which iteratively reaches a local optimum and is sufficient for most applications. In the sequel, we outline the algorithm:

1. fix k initial centroids;
2. for each point, associate it to its nearest centroid;
3. each centroid is now associated to a cluster of points - the new centroids in the next iteration are given by point-wise mean of each cluster.

The algorithm terminates when the centroids have stabilised, and for alternative distance functions termination is not guaranteed.

We chose to case study this algorithm for three reasons:

1. it is simple and short to implement;
2. it is extremely paralisable;
3. it naturally extends up to n-dimensions.

6.1 kmeans benchmarking program

This benchmarking program consists of a Haskell program which implements a kmeans clustering algorithm using Naperian functors, and Flat embeddings of Naperian functors into Accelerate, which can be executed using LLVM CPU-native instructions or LLVM PTX (similar to Nvidia CUDA) Nvidia GPU instructions. The components consist of:

- kmeans - the main program;
- GenSamples - a program to generate sample data to be fed into the algorithm based on parameters specified from the command line;
- benchmark.sh - a shell script to run everything together, reading seeds for the random generation of samples from seeds.txt.

6.1.1 Methodology. The kmeans program can be run in one of three modes,

- --naperian - i.e. in pure Haskell mode without any hardware acceleration;
- --accelerate-llvm-native - leveraging Accelerate and LLVM to generate optimised CPU instructions;
- --accelerate-llvm-ptx - leveraging Accelerate and LLVM PTX to generated instructions for Nvidia GPUs (formerly, this was Accelerate's CUDA backend).

As a parameter, it accepts the number of points to run with the algorithm, loaded from the start of points.bin with truncation: i.e. if points.bin contains 3000 points, yet

[7] with an additional convenience function to extract the size of the dimension

[8] i.e. if a is Euclidean-closer to b than c, then it is also squared-Euclidean-closer.

`kmeans` is ran with `-p 1000` it will load only the first 1000 points in `points.bin` and ignore the rest. We consider efficiency of the algorithm by considering the number of points input against running time until convergence (with a timeout specified by `-t`), fixing the number of target clusters at 5, and the dimensionality at 2^9. The `-m` flag specifies a *point multiplier*, which is applied repeatedly applied to the number of points until either the algorithm runs so long it times out or we reach the maximum number of points in the data.

For each dataset, we run the program three times (one for each implementation), recording the number of points, the time taken, and the result of the algorithm (assuming it didn't timeout), in each iteration of the algorithm. We then regenerate the dataset and repeat this process, until we have 30 sets of results. It is expected that by taking the average running time of n points using each of the different implementations that we can measure how 'fast' each implementation is.

6.1.2 Reproducibility. For reproducibility, everything is neatly packaged into a Docker image (see Dockerfile in the directory above), so data can be collected from any machine with Docker installed[10] simply by

```
docker run --runtime=nvidia \
  -v /directory/to/dump/data:/mnt/results \
  -it nickhu/kmeans
```

Alternatively, `stack` should be able to install and run this program, but note especially that installing the required Nvidia runtimes for LLVM PTX is quite fiddly.

Also included is a preset list of 30 seeds in `seeds.txt` (which were randomly generated) which are used to collect data in the main write up, but principally this can be replaced with any file containing lines of seed integers.

6.2 Experimental results

The benchmark was performed on three different machines:

1. a desktop computer with a high-end consumer Nvidia graphics card;
2. an ultrabook-class laptop (no discrete GPU, hence the absence of the green line in Figure 4);
3. a `p3.2xlarge` Amazon Web Services instance, designed for high-performance GPU computation.

Lloyd's algorithm is practically linear in running time, so we performed a linear regression analysis on samples which remained after discarding all samples for point sizes where there was a sample of that size or smaller such that a timeout was observed. The rationale behind this cut off is as follows: a timeout of 30 seconds means that we do not get any measurements for runs which would have terminated after slightly more than 30 seconds, so if we were to include any samples for points of that size that we did have measurements for, then the regression analysis would be skewed towards a shorter time. The gradient of the linear regression approximates the constant of the practically linear running time, while the intercept approximates the overhead of the rest of the program, along with other factors such as hardware and memory latency.

In practice, the native Haskell implementation is several orders of magnitude slower. One major observation is that even in the absence of a GPU, Accelerate in combination with LLVM produces code which still runs much faster. In part, this is expected as attempt was made to write optimised Haskell code; rather, the focus was comparing the relative performance of the 'obvious' implementation with and without using Accelerate.

Another observation, consistent with both Figure 3 and Figure 5, is that until we have roughly 100,000 points of data, the LLVM CPU implementation outperforms the GPU version. It is expected this is due to latency: time-wise, the transfer of data to the GPU is significantly more expensive than to the CPU, so it is only when we have sufficiently large data to process that we begin to see tangible gains from GPU computing.

In summary, Accelerate in either mode does provide very substantial performance gains.

7 CONCLUSION

In this work, we explored some ways to improve the performance of Naperian functors, particularly focusing on the Accelerate library. We have shown that Haskell, while not explicitly designed to be efficient with respect to data parallel programming, can achieve fast performance in conjunction with Accelerate. Using the techniques from before, we have shown that Naperian data types and Accelerate data types have some compatibility, which can be utilised to enable us to write programs in an abstract style with Naperian functors while leveraging the raw performance of Accelerate.

REFERENCES

Chakravarty, Manuel M T, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. "Accelerating Haskell array codes with multicore GPUs." In *DAMP '11: The 6th Workshop on Declarative Aspects of Multicore Programming*. ACM.

Eisenberg, Richard A., and Peyton JonesSimon. 2017. "Levity Polymorphism." In *Proceedings of the 38th Acm Sigplan Conference on Programming Language Design and Implementation*, 525–39. PLDI 2017. New York, NY, USA: ACM. https://doi.org/10.1145/3062341.3062357.

Gibbons, Jeremy. 2017. "APLicative Programming with Naperian Functors." In *European Symposium on Programming*, edited by Hongseok Yang, 10201:568–83. LNCS. https://doi.org/10.1007/978-3-662-54434-1_21.

McDonell, Trevor L, Manuel M T Chakravarty, Vinod Grover, and Ryan R Newton. 2015. "Type-safe Runtime

[9]In principle, the code should require only slight modifications to change the dimensionality, especially the `Hyper` (native Haskell) version; however, Accelerate integration required further specialisation so modifying the `Flat` (Accelerate) version to support dimensional polymorphism is expected to be more difficult.

[10]The `--runtime=nvidia` flag is optional, if the LLVM PTX Accelerate backend is desired - this requires installation of `nvidia-docker`.

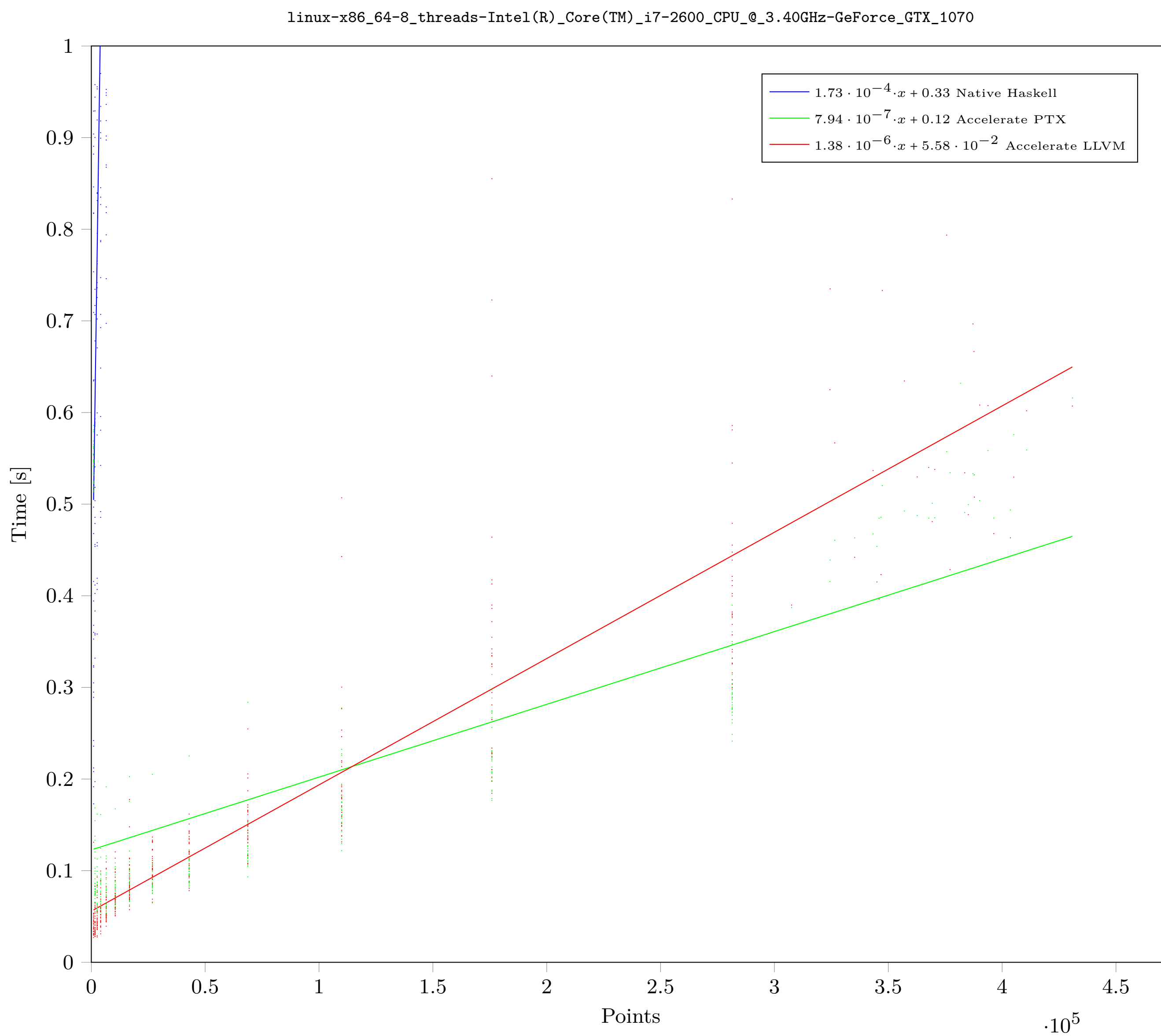

Figure 3: Desktop benchmark

Code Generation: Accelerate to LLVM." In *Haskell '15: The 8th Acm Sigplan Symposium on Haskell*, 201–12. ACM.

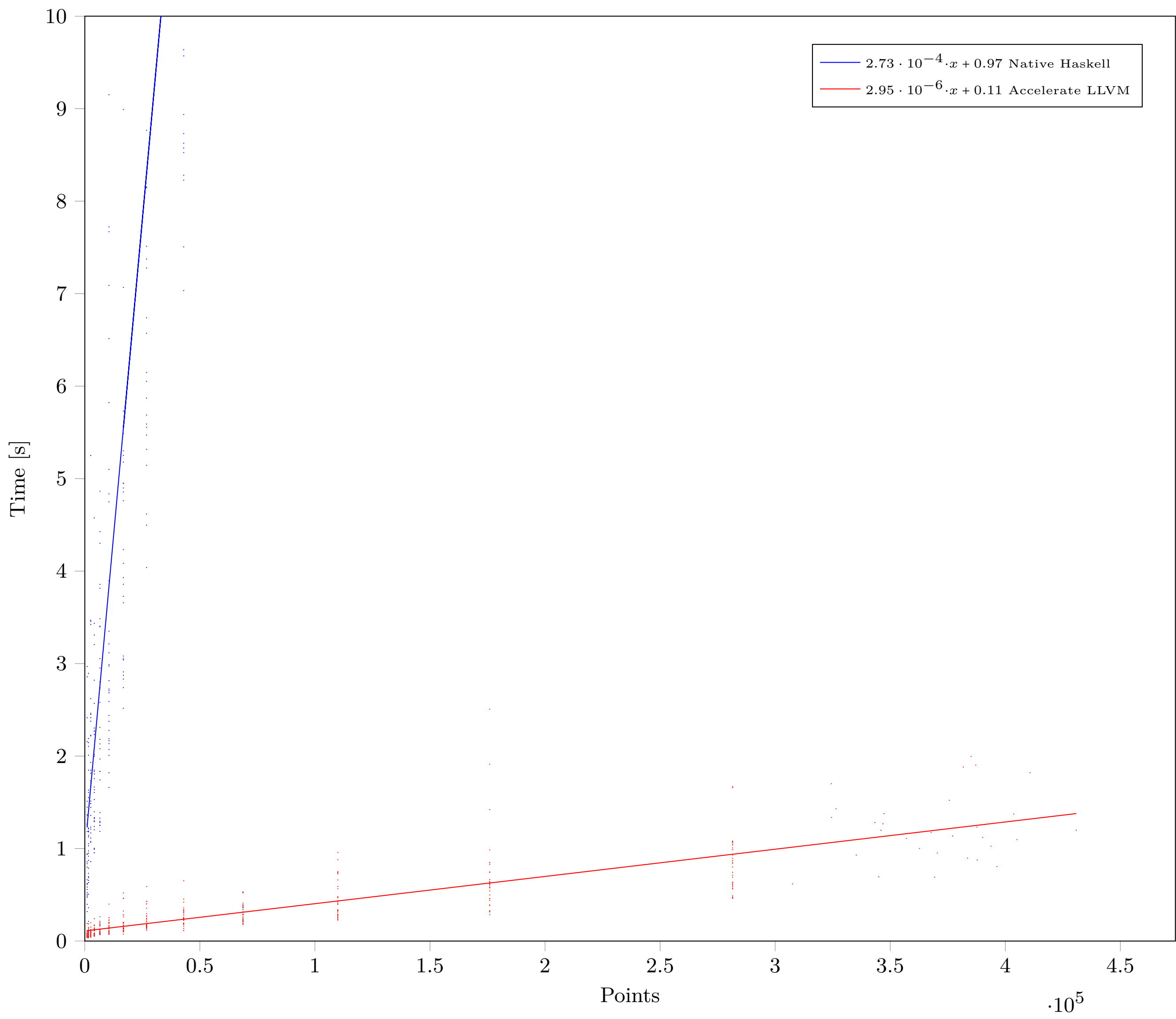

Figure 4: Laptop benchmark

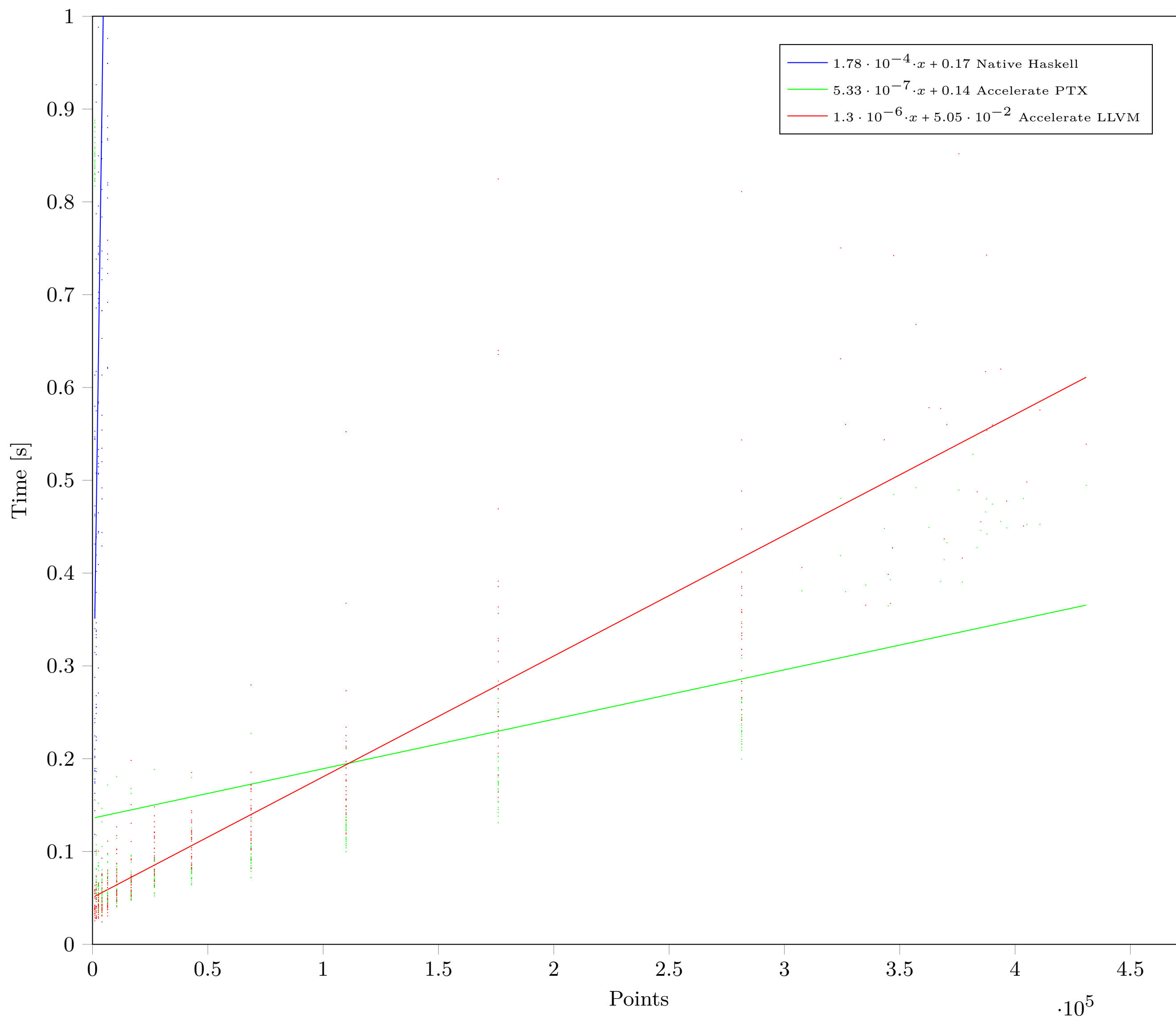

Figure 5: Amazon Web Services `p3.2xlarge`