Repa and Accelerate

Data-parallel and GPGPU programming in Haskell

Andres Löh

(thanks to Simon Marlow and the Repa and Accelerate teams)

13 July 2012

Well-Typed
The Haskell Consultants

Preparations

We use `ghc-7.4.1`.

We need the following Cabal packages:

```
repa-3.2.1.1
accelerate-0.12.1.0
```

Ideally also the following:

```
repa-examples-3.2.1.1
accelerate-cuda-0.12.1.0
accelerate-examples-0.12.1.0
```

Version differences are at your own risk `:-)`

Even without CUDA, you can still do:

```
cabal install -f-cuda accelerate-examples
```

Well-Typed

Overview

Parallelism

Parallelism

Running (parts of) programs in parallel on multiple cores (or nodes), in order to speed up the program.

Concurrency

Language constructs that support structuring a program as if it has many independent threads of control.

Concurrency vs. Parallelism

Concurrency:

- is a goal in its own (program structure),
- usually rather low-level (shared memory, message passing, communication problems, deadlocks, race conditions),
- does not require parallel hardware at all (can be simulated by multitasking on a single core),
- while supported in Haskell, is not the primary choice for parallelism.

Concurrency vs. Parallelism

Concurrency:

- is a goal in its own (program structure),
- usually rather low-level (shared memory, message passing, communication problems, deadlocks, race conditions),
- does not require parallel hardware at all (can be simulated by multitasking on a single core),
- while supported in Haskell, is not the primary choice for parallelism.

Parallelism:

- the goal is speed,
- using several cores is the main point,
- there's conceptually no need for low-level effects or IO,
- we would like deterministic results.

Many approaches

Concurrency:

- forkIO and MVars,
- `async`,
- Software Transactional Memory (`stm`),
- Cloud Haskell (`remote`).

Parallelism:

- par, pseq,
- strategies (`parallel`),
- `monad-par`,
- data-parallelism (`repa`, `accelerate`, DPH),
- tasks in Cloud Haskell.

This list is not complete. Parallelism and concurrency are hot topics.

Concurrency for parallelism

We can use concurrency for achieving parallelism in Haskell, but:

- we have to communicate results between threads,
- we have to manage threads, i.e., wait for them to finish etc.,
- everything is forced into `IO`.

All this is tedious, error-prone, and distracts from the main goal.

Pure, deterministic parallelism

- Most parallel code does not need IO.
- We just describe an algorithm, to be run in parallel.
- Ideally, its result should not depend on whether it is run in parallel or not.

Pure, deterministic parallelism

- Most parallel code does not need IO.
- We just describe an algorithm, to be run in parallel.
- Ideally, its result should not depend on whether it is run in parallel or not.

Several Haskell approaches to parallelism try to ensure deterministic results – independent of number of cores or scheduling decisions.

Task and data-parallelism

Task parallelism

Dividing the overall work into many components that are partially independent.

Data parallelism

Performing the exact same operations for many items of data.

Task and data-parallelism

Task parallelism

Dividing the overall work into many components that are partially independent.

Data parallelism

Performing the exact same operations for many items of data.

Data parallelism is more limited than task parallelism, but can often be supported rather efficiently, even by hardware.

Today

A look at two Haskell libraries specifically designed with data-parallelism in mind:

Today

A look at two Haskell libraries specifically designed with data-parallelism in mind:

Repa

A library for regular, shape-polymorphic arrays with different representation and "automatic" flat data-parallelism.

Today

A look at two Haskell libraries specifically designed with data-parallelism in mind:

Repa

A library for regular, shape-polymorphic arrays with different representation and "automatic" flat data-parallelism.

Accelerate

A library for regular, shape-polymorphic arrays and operations that are "deeply embedded" into Haskell, but can be compiled and run elsewhere, for example on the GPU.

Today

A look at two Haskell libraries specifically designed with data-parallelism in mind:

Repa

A library for regular, shape-polymorphic arrays with different representation and “automatic” flat data-parallelism.

Accelerate

A library for regular, shape-polymorphic arrays and operations that are “deeply embedded” into Haskell, but can be compiled and run elsewhere, for example on the GPU.

Both libraries share a lot of similarities ...

Isn't Accelerate just better?

No.

- The desire to completely reflect all computations imposes some restrictions on Accelerate programs that Repa does not share.
- Repa explicitly distinguishes between different array representations.
- But Repa could be used as a backend for Accelerate.

The relation to DPH

Data-parallel Haskell aims at offering nested data-parallelism:

- parts of parallel computations can themselves be parallel,
- nested arrays can be of irregular shape.

The relation to DPH

Data-parallel Haskell aims at offering nested data-parallelism:

- parts of parallel computations can themselves be parallel,
- nested arrays can be of irregular shape.

This is required for truly modular and compositional data-parallel functional programming, but neither Accelerate nor Repa currently offer this.

Repa

Introducing Repa

A library for data-parallelism in Haskell:

- implemented as an EDSL,
- based on adaptive unboxed arrays,
- offers “delayed” arrays,
- arrays can be re-shaped,
- makes use of advanced type system features,
- offers high-level parallelism.

Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

- the type is a data family – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);

Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e   -- abstract
```

There are a number of things worth noting:

- the type is a data family – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are three type arguments;

Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e   -- abstract
```

There are a number of things worth noting:

- the type is a data family – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are three type arguments;
- the final is the element type;

Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

- the type is a data family – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are three type arguments;
- the final is the element type;
- the first denotes the representation of the array;

Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e   -- abstract
```

There are a number of things worth noting:

- the type is a data family – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are three type arguments;
- the final is the element type;
- the first denotes the representation of the array;
- the second the shape.

Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

- the type is a data family – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are three type arguments;
- the final is the element type;
- the first denotes the representation of the array;
- the second the shape.

But what are representation and shape?

Array shapes

Repa can represent multi-dimensional arrays:

- as a first approximation, the shape of an array describes its dimension;
- the shape also describes the type of an array index.

Array shapes

Repa can represent multi-dimensional arrays:

- as a first approximation, the shape of an array describes its dimension;
- the shape also describes the type of an array index.

```
data Z = Z               -- similar to the () type, Z for "zero"
data t :. h = !t :. !h   -- similar to (,) , but strict

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

Array shapes

Repa can represent multi-dimensional arrays:

- as a first approximation, the shape of an array describes its dimension;
- the shape also describes the type of an array index.

```
data Z = Z              -- similar to the () type, Z for "zero"
data t :. h = !t :. !h  -- similar to (,), but strict
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
...
```

So `DIM2` is the type of strict pairs of integers.

Array representations

Repa distinguishes two fundamentally different states an array can be in:

Well-Typed

Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a manifest array is an array that is represented as a block in memory, as we'd expect;

Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a manifest array is an array that is represented as a block in memory, as we'd expect;
- a delayed array is not a real array at all, but merely a computation that describes how to compute each of the elements.

Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a manifest array is an array that is represented as a block in memory, as we'd expect;
- a delayed array is not a real array at all, but merely a computation that describes how to compute each of the elements.

Let's look at the "why" and the delayed representation in a moment.

The standard manifest representation is denoted by a type argument `U` (for unboxed).

Creating manifest arrays

$$\text{fromListUnboxed} \\ \quad :: (\text{Shape sh}, \text{Unbox a}) \Rightarrow \text{sh} \rightarrow [\text{a}] \rightarrow \text{Array U sh a}$$

Creating manifest arrays

```
fromListUnboxed
  :: (Shape sh, Unbox a) ⇒ sh → [a] → Array U sh a
```

Example:

```
〉 fromListUnboxed (Z :. 10 :: DIM1) [1 .. 10 :: Int]
AUnboxed (Z :. 10) (fromList [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
〉 fromListUnboxed (Z :. 2 :. 5 :: DIM2) [1 .. 10 :: Int]
AUnboxed ((Z :. 2) :. 5) (fromList [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

The shape argument provides the dimensions and size of the array; the list must match the size of the shape:

```
〉 size (Z :. 2 :. 5 :: DIM2)
10
```

Well-Typed

The Unbox class

The fromListUnboxed function creates an adaptive unboxed array.

The Unbox class is defined in the `vector` package:

```
class Unbox a
instance Unbox Int
instance Unbox Float
instance Unbox Double
instance Unbox Char
instance Unbox Bool
instance (Unbox a, Unbox b) ⇒ Unbox (a, b)
```

- Choose an efficient representation depending on element type.
- Represent arrays of tuples as tuples of arrays.

What if our type is not in `Unbox`?

Two options:

- define an `Unbox` instance (tedious, but generally possible);
- use a less efficient manifest array representation (`V`).

For the purposes of this tutorial, base types and `U` are sufficient.

Array access

$$
\begin{aligned}
&\mathsf{extent} :: (\mathsf{Shape}\ \mathsf{sh}, \mathsf{Source}\ \mathsf{r}\ \mathsf{e}) \Rightarrow \mathsf{Array}\ \mathsf{r}\ \mathsf{sh}\ \mathsf{e} \to \mathsf{sh} \\
&(!) \quad\quad :: (\mathsf{Shape}\ \mathsf{sh}, \mathsf{Source}\ \mathsf{r}\ \mathsf{e}) \Rightarrow \mathsf{Array}\ \mathsf{r}\ \mathsf{sh}\ \mathsf{e} \to \mathsf{sh} \to \mathsf{e}
\end{aligned}
$$

Array access

$$\begin{array}{ll} \text{extent} & :: (\text{Shape sh}, \text{Source r e}) \Rightarrow \text{Array r sh e} \rightarrow \text{sh} \\ (!) & :: (\text{Shape sh}, \text{Source r e}) \Rightarrow \text{Array r sh e} \rightarrow \text{sh} \rightarrow \text{e} \end{array}$$

Array with two rows, five columns:

$$\begin{array}{l} \text{example} :: \text{Array U DIM2 Int} \\ \text{example} = \text{fromListUnboxed} \ (\text{Z} :. \ 2 :. \ 5 :: \text{DIM2}) \ [1 \,..\, 10 :: \text{Int}] \end{array}$$

Array access

```
extent :: (Shape sh, Source r e) ⇒ Array r sh e → sh
(!)    :: (Shape sh, Source r e) ⇒ Array r sh e → sh → e
```

Array with two rows, five columns:

```
example :: Array U DIM2 Int
example = fromListUnboxed (Z :. 2 :. 5 :: DIM2) [1 .. 10 :: Int]
```

```
〉 extent example
(Z :. 2) :. 5
〉 example ! (Z :. 1 :. 3)
9
```

Well-Typed

The Source class

The class Source keeps track which element types are allowed for which representation:

```
class Source r e
instance Unbox a ⇒ Source U a
instance           Source V a
```

The unboxed representation is only valid for elements in the Unbox class.

Operations on arrays

$$
\begin{aligned}
\mathsf{map} \;&:: (\mathsf{Shape\ sh}, \mathsf{Source\ r\ a}) \Rightarrow \\
&\quad (\mathsf{a} \to \mathsf{b}) \to \mathsf{Array\ r\ sh\ a} \to \mathsf{Array\ D\ sh\ b} \\
\mathsf{extract} \;&:: (\mathsf{Shape\ sh}, \mathsf{Source\ r\ e}) \Rightarrow \\
&\quad \mathsf{sh} \to \mathsf{sh} \to \mathsf{Array\ r\ sh\ e} \to \mathsf{Array\ D\ sh\ e} \\
(+\!\!+) \;&:: (\mathsf{Shape\ sh}, \mathsf{Source\ r1\ e}, \mathsf{Source\ r2\ e}) \Rightarrow \\
&\quad \mathsf{Array\ r1\ (sh :. Int)\ e} \to \mathsf{Array\ r2\ (sh :. Int)} \to \\
&\quad \mathsf{Array\ D\ (sh :. Int)\ e} \\
(*\hat{}) \;&:: (\mathsf{Num\ c}, \mathsf{Shape\ sh}, \mathsf{Source\ r1\ c}, \mathsf{Source\ r2\ c}) \Rightarrow \\
&\quad \mathsf{Array\ r1\ sh\ c} \to \mathsf{Array\ r2\ sh\ c} \to \mathsf{Array\ D\ sh\ c}
\end{aligned}
$$

Note:

- What does the shape requirement on $(+\!\!+)$ tell us?
- All these functions return delayed arrays (D).

Why delayed arrays?

Consider “map fusion”:

$$(\mathrm{map}\ f \circ \mathrm{map}\ g)\ xs == \mathrm{map}\ (f \circ g)\ xs$$

- For lists, rather than traversing a list several times, we can traverse it once and do several operations at once.

Why delayed arrays?

Consider "map fusion":

```
(map f ∘ map g) xs == map (f ∘ g) xs
```

- For lists, rather than traversing a list several times, we can traverse it once and do several operations at once.
- However, lists can be traversed one by one. Even if we don't fuse the computations, we only allocate the intermediate cons-cells for the cons-cells we evaluate in the end.

Why delayed arrays?

Consider "map fusion":

$(\mathrm{map}\ f \circ \mathrm{map}\ g)\ xs == \mathrm{map}\ (f \circ g)\ xs$

- For lists, rather than traversing a list several times, we can traverse it once and do several operations at once.
- However, lists can be traversed one by one. Even if we don't fuse the computations, we only allocate the intermediate cons-cells for the cons-cells we evaluate in the end.
- For arrays, we have to make a full intermediate copy for every traversal, so performing fusion becomes essential – so important that we'd like to make it explicit in the type system.

Delayed arrays

How should we represent a delayed array?

Delayed arrays

How should we represent a delayed array?

By describing how to compute each element if needed:

data instance Array D sh e $=$ ADelayed !sh (sh $\rightarrow$ e)

- Delayed arrays aren't really arrays at all.
- Operating on an array does not create a new array.
- Performing another operation on a delayed array just performs function composition.
- If we want to have a manifest array again, we have to explicitly force the array.

Well-Typed

Creating delayed arrays

From a function:

$$\text{fromFunction} :: \text{sh} \to (\text{sh} \to \text{a}) \to \text{Array D sh a}$$

Directly maps to ADelayed.

From an arbitrary Repa array:

$$\text{delay} :: (\text{Shape sh}, \text{Source r e}) \Rightarrow \text{Array r sh e} \to \text{Array D sh e}$$

The implementation of map

```
map :: (Shape sh, Source r a)
    ⇒ (a → b) → Array r sh a → Array D sh b
map f arr = case delay arr of
              ADelayed sh g → ADelayed sh (f ∘ g)
```

Well-Typed

The implementation of map

$$
\begin{aligned}
&\mathsf{map} :: (\mathsf{Shape}\ sh, \mathsf{Source}\ r\ a) \\
&\qquad \Rightarrow (a \to b) \to \mathsf{Array}\ r\ sh\ a \to \mathsf{Array}\ \mathsf{D}\ sh\ b \\
&\mathsf{map}\ f\ arr = \mathbf{case}\ \mathsf{delay}\ arr\ \mathbf{of} \\
&\qquad\qquad \mathsf{ADelayed}\ sh\ g \to \mathsf{ADelayed}\ sh\ (f \circ g)
\end{aligned}
$$

Many other functions are only slightly more complicated:

- think about pointwise multiplication `(*^)`,
- or the more general `zipWith`.

Forcing delayed arrays

Sequentially:

$$\mathsf{computeS} :: (\mathsf{Target}\ \mathsf{r2}\ \mathsf{e}, \mathsf{Load}\ \mathsf{r1}\ \mathsf{sh}\ \mathsf{e}) \Rightarrow \mathsf{Array}\ \mathsf{r1}\ \mathsf{sh}\ \mathsf{e} \rightarrow \mathsf{Array}\ \mathsf{r2}\ \mathsf{sh}\ \mathsf{e}$$

Forcing delayed arrays

Sequentially:

$$\begin{aligned}\mathsf{computeS} :: &(\mathsf{Target}\ \mathsf{r2}\ \mathsf{e}, \mathsf{Load}\ \mathsf{r1}\ \mathsf{sh}\ \mathsf{e}) \Rightarrow \\ &\mathsf{Array}\ \mathsf{r1}\ \mathsf{sh}\ \mathsf{e} \rightarrow \mathsf{Array}\ \mathsf{r2}\ \mathsf{sh}\ \mathsf{e}\end{aligned}$$

In parallel:

$$\begin{aligned}\mathsf{computeP} :: &(\mathsf{Monad}\ \mathsf{m}, \mathsf{Source}\ \mathsf{r2}\ \mathsf{e}, \mathsf{Target}\ \mathsf{r2}\ \mathsf{e}, \mathsf{Load}\ \mathsf{r1}\ \mathsf{sh}\ \mathsf{e}) \Rightarrow \\ &\mathsf{Array}\ \mathsf{r1}\ \mathsf{sh}\ \mathsf{e} \rightarrow \mathsf{m}\ (\mathsf{Array}\ \mathsf{r2}\ \mathsf{sh}\ \mathsf{e})\end{aligned}$$

Forcing delayed arrays

Sequentially:

$$\text{computeS} :: (\text{Target r2 e}, \text{Load r1 sh e}) \Rightarrow$$
$$\text{Array r1 sh e} \rightarrow \text{Array r2 sh e}$$

In parallel:

$$\text{computeP} :: (\text{Monad m}, \text{Source r2 e}, \text{Target r2 e}, \text{Load r1 sh e}) \Rightarrow$$
$$\text{Array r1 sh e} \rightarrow \text{m (Array r2 sh e)}$$

Forcing works by:

- temporarily allocating a mutable vector,
- computing all the elements of the source (Load),
- writing them to the vector (Target),
- and then freezing that vector.

“Automatic” parallelism

Behind the scenes:

- Repa starts a gang of threads.
- Depending on the number of available cores, Repa assigns chunks of the array to be computed by different threads.
- The chunking and scheduling and synchronization don’t have to concern the user.

“Automatic” parallelism

Behind the scenes:

- Repa starts a gang of threads.
- Depending on the number of available cores, Repa assigns chunks of the array to be computed by different threads.
- The chunking and scheduling and synchronization don’t have to concern the user.
- But: Repa only supports flat data-parallelism! If the delayed computations forced by `computeP` are themselves parallel, Repa will fall back to sequential computation.
- This is why `computeP` is fake-monadic.

Reducing arrays

Reductions or folds are also available in both sequential and parallel variants:

```
sumS    :: (Num a, Shape sh, Source r a, Unbox a, Elt a) ⇒
           Array r (sh :. Int) a → Array U sh a
sumP    :: (Monad m, Num a, Shape sh, Source r a, Unbox a, Elt a) ⇒
           Array r (sh :. Int) a → m (Array U sh a)

sumAllS :: (Num a, Shape sh, Source r a, Unbox a, Elt a) ⇒
           Array r sh a → a
sumAllP :: (Monad m, Num a, Shape sh, Source r a, Unbox a, Elt a) ⇒
           Array r sh a → m a

foldS   :: (Shape sh, Source r a, Unbox a, Elt a) ⇒
           (a → a → a) → a → Array r (sh :. Int) a → Array U sh a
foldP   :: (Monad m, Shape sh, Source r a, Unbox a, Elt a) ⇒
           (a → a → a) → a → Array r (sh :. Int) a → m (Array U sh a)
```

The constraint `Elt` is comparable to `Unbox`.

Examples

```
example :: Array U DIM2 Int
example = fromListUnboxed (Z :. 2 :. 5) [1 .. 10]
```

```
〉computeS (map (+ 1) example) :: Array U DIM2 Int
AUnboxed ((Z :. 2) :. 5) (fromList [2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
〉computeUnboxedS (extract (Z :. 0 :. 1) (Z :. 2 :. 3) example
AUnboxed ((Z :. 2) :. 3) (fromList [2, 3, 4, 7, 8, 9])
〉sumS it
AUnboxed (Z :. 2) (fromList [9, 24])
〉sumS it
AUnboxed Z (fromList [33])
〉sumAllS example
55
```

Larger example: Matrix multiplication

Goal

- Implement naive matrix multiplication.
- Benefit from parallelism.
- Learn about a few more Repa functions.

This is taken from the `repa-examples` package which contains more than just this example.

Goal

- Implement naive matrix multiplication.
- Benefit from parallelism.
- Learn about a few more Repa functions.

This is taken from the `repa-examples` package which contains more than just this example.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} \begin{pmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 12 & 9 & 6 \\ 30 & 23 & 16 \\ 48 & 37 & 26 \\ 66 & 51 & 36 \end{pmatrix}$$

Goal

- Implement naive matrix multiplication.
- Benefit from parallelism.
- Learn about a few more Repa functions.

This is taken from the `repa-examples` package which contains more than just this example.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} \begin{pmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 12 & 9 & 6 \\ 30 & 23 & 16 \\ 48 & 37 & 26 \\ 66 & 51 & 36 \end{pmatrix}$$

Start with the types

We want something like this:

```
mmultP :: Monad m =>
          Array U DIM2 Double -> Array U DIM2 Double ->
          m (Array U DIM2 Double)
```

- We inherit the `Monad` constraint from the use of a parallel compute function.
- We work with two-dimensional arrays, it's an additional prerequisite that the dimensions match.

Strategy

- Two matrices of shapes `Z :. h1 :. w1` and `Z :. h2 :. w2`.
- We expect `w1` and `h2` to be equal.
- The resulting matrix will have shape `Z :. h1 :. w2`.
- We have to traverse the rows of the first and the columns of the second matrix, yielding one-dimensional arrays.
- For each of these pairs, we have to take the sum of the products.
- And these results determine the values of the result matrix.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} \begin{pmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 12 & 9 & 6 \\ 30 & 23 & 16 \\ 48 & 37 & 26 \\ 66 & 51 & 36 \end{pmatrix}$$

Well-Typed

Strategy – contd.

Some observations:

- the result is given by a function,
- we need a way to slice rows or columns out of a matrix,

Starting top-down

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
mmultP m1 m2 =
  do
    let (Z :. h1 :. w1) = extent m1
    let (Z :. h2 :. w2) = extent m2
    computeP (fromFunction   (Z :. h1 :. w2)
                             (λ(Z :. r   :. c   ) → ...)
```

Well-Typed

Slicing

A quite useful function offered by Repa is `backpermute`:

```
backpermute :: (Shape sh1, Shape sh2, Source r e) ⇒
               sh2 →                     -- new shape
               (sh2 → sh1) →             -- map new index to old index
               Array r sh1 e → Array D sh2 e
```

- We compute a delayed array simply by saying how each index can be computed in terms of an old index.
- This is trivial to implement in terms of `fromFunction`.

Slicing – contd.

We can use backpermute to slice rows and columns.

```
sliceCol, sliceRow
  :: Source r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceCol  c a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. h ) (λ(Z :. r ) → (Z :. r :. c)) a
sliceRow r  a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. w) (λ(Z :. c) → (Z :. r :. c)) a
```

Slicing – contd.

We can use backpermute to slice rows and columns.

```
sliceCol, sliceRow
  :: Source r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceCol  c a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. h ) (λ(Z :. r ) → (Z :. r :. c)) a
sliceRow r  a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. w) (λ(Z :. c) → (Z :. r :. c)) a
```

```
〉 computeUnboxedS (sliceCol 3 example)
AUnboxed (Z :. 2) (fromList [4, 9])
```

Note that sliceCol and sliceRow do not actually create a new array unless we force it!

Slicing – contd.

Repa itself offers are more general slicing function (but it's based on the same idea):

$$\text{slice} :: (\text{Slice sl}, \text{Shape (SliceShape sl)}, \text{Shape (FullShape sl)}, \text{Source r e}) \Rightarrow \text{Array r (FullShape sl) e} \rightarrow \text{sl} \rightarrow \text{Array D (SliceShape sl) e}$$

A member of class Slice :

- looks similar to a member of class Shape ,
- but describes two shapes at once, the orginal and the sliced.

Slicing – contd.

Repa itself offers are more general slicing function (but it's based on the same idea):

$$\begin{aligned}&\mathsf{slice} :: (\mathsf{Slice}\ sl, \mathsf{Shape}\ (\mathsf{SliceShape}\ sl), \mathsf{Shape}\ (\mathsf{FullShape}\ sl),\\ &\qquad \mathsf{Source}\ r\ e) \Rightarrow\\ &\qquad \mathsf{Array}\ r\ (\mathsf{FullShape}\ sl)\ e \to sl \to \mathsf{Array}\ D\ (\mathsf{SliceShape}\ sl)\ e\end{aligned}$$

A member of class Slice:

- looks similar to a member of class Shape,
- but describes two shapes at once, the orginal and the sliced.

$$\begin{aligned}&\mathsf{sliceCol}, \mathsf{sliceRow} :: \mathsf{Source}\ r\ e \Rightarrow\\ &\qquad\qquad \mathsf{Int} \to \mathsf{Array}\ r\ \mathsf{DIM2}\ e \to \mathsf{Array}\ D\ \mathsf{DIM1}\ e\\ &\mathsf{sliceCol}\ \ c\ a = \mathsf{slice}\ a\ (Z :. \mathsf{All} :. c\ \ )\\ &\mathsf{sliceRow}\ r\ \ a = \mathsf{slice}\ a\ (Z :. r\ \ \ :. \mathsf{All})\end{aligned}$$

Putting everything together

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
mmultP m1 m2 =
  do
    let (Z :. h1 :. w1) = extent m1
    let (Z :. h2 :. w2) = extent m2
    computeP (fromFunction (Z :. h1 :. w2)
                 (λ(Z :. r :. c) →
                    sumAllS (sliceRow r m1 *^ sliceCol c m2)
                 ))
```

That's all. Note that we compute no intermediate arrays.

Summary

- The true magic of Repa is in the `computeP` -like functions, where parallelism is automatically handled.
- Haskell's type system is used in various ways:
  - Adapt the representation of unboxed arrays to element types.
  - Keep track of the shape of an array, to make fusion explicit.
  - Keep track of the state of an array.
- We have seen yet another embedded domain-specific language:
  - for efficient array computations,
  - allowing high-level deterministic parallelism,
  - where the types direct us towards correct use.
- A large part of Repa's implementation is actually quite understandable.

Accelerate

Introducing Accelerate

A library for GPGPU programming in Haskell:

- GPGPU = general-purpose graphics processing unit.
- Use the GPU to perform all sorts of computations.
- GPUs have lots of cores, but all cores have to run the same operations.
- GPUs have a different instruction set than CPUs and have to be programmed differently.

In Haskell?

- Yes, Accelerate is a deeply embedded domain-specific language.
- When you write an Accelerate program in Haskell, you in truth define the abstract syntax tree of a GPU program.
- But not all of Haskell can be transported to the GPU – even if we wanted, it would be horribly inefficient.

In Haskell?

- Yes, Accelerate is a deeply embedded domain-specific language.
- When you write an Accelerate program in Haskell, you in truth define the abstract syntax tree of a GPU program.
- But not all of Haskell can be transported to the GPU – even if we wanted, it would be horribly inefficient.

Therefore, we separate:

- Accelerate distinguishes between code that runs on the CPU and code that runs on the GPU.
- GPUs have their own memory, so we explicitly move data back and forth.

Multiple backends

Accelerate is structured such that its programs can be run on different targets:

- accelerated backend using CUDA,
- interpreted (but slow) backend for the CPU,
- several other backends (OpenCL, Repa, ...) in development.

Multiple backends

Accelerate is structured such that its programs can be run on different targets:

- accelerated backend using CUDA,
- interpreted (but slow) backend for the CPU,
- several other backends (OpenCL, Repa, ...) in development.

The interpreter backend has the huge advantage that you do not have to have an NVIDIA card in order to play with Accelerate!

Multiple backends

Accelerate is structured such that its programs can be run on different targets:

- accelerated backend using CUDA,
- interpreted (but slow) backend for the CPU,
- several other backends (OpenCL, Repa, ...) in development.

The interpreter backend has the huge advantage that you do not have to have an NVIDIA card in order to play with Accelerate!

The CUDA backend invokes the CUDA compiler in the background (but code fragments that are used repeatedly are cached).

Using Accelerate

Generic Accelerate library

```
import Data.Array.Accelerate
```

Using Accelerate

Generic Accelerate library

```
import Data.Array.Accelerate
```

Pick a backend:

Interpreter backend

```
import Data.Array.Accelerate.Interpreter   -- provides run
```

or

CUDA backend

```
import Data.Array.Accelerate.CUDA   -- provides run
```

Accelerate arrays

```
data Array sh e   -- abstract
```

- shapes (nearly exactly as in Repa) and element type,
- but no explicit representation.

Accelerate arrays

```
data Array sh e   -- abstract
```

- shapes (nearly exactly as in Repa) and element type,
- but no explicit representation.

Type synonyms:

```
type Scalar e = Array DIM0 e
type Vector e = Array DIM1 e
```

Array creation

Almost as in Repa:

$$\mathsf{fromList} \\ \quad :: (\mathsf{Shape}\ sh, \mathsf{Elt}\ e) \Rightarrow sh \rightarrow [e] \rightarrow \mathsf{Array}\ sh\ e$$

Array creation

Almost as in Repa:

```
fromList
  :: (Shape sh, Elt e) ⇒ sh → [e] → Array sh e
```

Example:

```
〉 fromList (Z :. 10) [1 .. 10] :: Vector Int
Array (Z :. 10) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
〉 fromList (Z :. 2 :. 5) [1 .. 10] :: Array DIM2 Int
Array (Z :. 2 :. 5) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Actually, Accelerate is more lenient than Repa and allows us to pass in a longer list:

```
〉 fromList (Z :. 2 :. 5) [1 ..] :: Array DIM2 Int
Array (Z :. 2 :. 5) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Well-Typed

Array access

$$\begin{array}{ll} \text{arrayShape} & :: (\text{Shape sh}) \Rightarrow \text{Array sh e} \rightarrow \text{sh} \\ \text{arraySize} & :: (\text{Shape sh}) \Rightarrow \text{sh} \rightarrow \text{Int} \\ \text{indexArray} & :: \text{Array sh e} \rightarrow \text{sh} \rightarrow \text{e} \end{array}$$

$$\begin{array}{l} \text{example} :: \text{Array DIM2 Int} \\ \text{example} = \text{fromList}\ (\text{Z} :. 2 :. 5)\ [1 \mathinner{..} 10] \end{array}$$

$$\begin{array}{l} \rangle\ \text{arrayShape example} \\ (\text{Z} :. 2) :. 5 \\ \rangle\ \text{example `indexArray`}\ (\text{Z} :. 1 :. 3) \\ 9 \end{array}$$

Well-Typed

Moving between CPU and GPU

Arrays created this way still live in the main memory.

$$\mathsf{use} :: \mathsf{Arrays}\ \mathsf{arrays} \Rightarrow \mathsf{arrays} \rightarrow \mathsf{Acc}\ \mathsf{arrays}$$
$$\mathsf{run}\ \ :: \mathsf{Arrays}\ \mathsf{arrays} \Rightarrow \mathsf{Acc}\ \mathsf{arrays} \rightarrow \mathsf{arrays}$$

Data on the GPU is marked by Acc.

$$\textbf{class}\ \mathsf{Arrays}\ \mathsf{a}$$
$$\textbf{instance}\ (\mathsf{Shape}\ \mathsf{sh}, \mathsf{Elt}\ \mathsf{e}) \Rightarrow \mathsf{Arrays}\ (\mathsf{Array}\ \mathsf{sh}\ \mathsf{e})$$
$$\textbf{instance}\ (\mathsf{Arrays}\ \mathsf{a}, \mathsf{Arrays}\ \mathsf{b}) \Rightarrow \mathsf{Arrays}\ (\mathsf{a}, \mathsf{b})$$
$$\ldots$$

Performing a computation

```
import Data.Array.Accelerate as A
import Data.Array.Accelerate.Interpreter
```

```
example :: Array DIM2 Int
example = fromList (Z :. 2 :. 5) [1 .. 10]
```

```
〉 run (A.map (+ 1) (use example))
Array (Z :. 2 :. 5) [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Performing a computation

```
import Data.Array.Accelerate as A
import Data.Array.Accelerate.Interpreter
```

```
example :: Array DIM2 Int
example = fromList (Z :. 2 :. 5) [1 .. 10]
```

```
〉 run (A.map (+ 1) (use example))
Array (Z :. 2 :. 5) [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Question: Where does `(+ 1)` run?

Performing a computation

```
import Data.Array.Accelerate as A
import Data.Array.Accelerate.Interpreter
```

```
example :: Array DIM2 Int
example = fromList (Z :. 2 :. 5) [1 .. 10]
```

```
〉 run (A.map (+ 1) (use example))
Array (Z :. 2 :. 5) [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Question: Where does `(+ 1)` run?

On the GPU. So what's the type of `A.map`?

GPU computations

$$\begin{aligned}\mathsf{map} :: &(\mathsf{Shape\ ix}, \mathsf{Elt\ a}, \mathsf{Elt\ b}) \Rightarrow \\ &(\mathsf{Exp\ a} \to \mathsf{Exp\ b}) \to \mathsf{Acc\ (Array\ ix\ a)} \to \mathsf{Acc\ (Array\ ix\ b)}\end{aligned}$$

- Exp is like Acc, but for scalars,
- functions between Exp and Acc values can be "quoted".

GPU computations

```
map :: (Shape ix, Elt a, Elt b) ⇒
       (Exp a → Exp b) → Acc (Array ix a) → Acc (Array ix b)
```

- Exp is like Acc, but for scalars,
- functions between Exp and Acc values can be "quoted".

Let's try to omit the run:

```
⟩ A.map (+ 1) (use example)
map
  (λx0 → x0 + 1)
  (use ((Array (Z :. 2 :. 5) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])))
```

There's really an AST of the expression being built, and we are allowed to see it!

Embedding expressions

How can $(+\ 1)$ be of type $\text{Exp Int} \to \text{Exp Int}$?

$$\textbf{instance}\ (\text{Elt}\ t, \text{IsNum}\ t) \Rightarrow \text{Num}\ (\text{Exp}\ t)$$

Similarly for other classes (although not all methods are implemented).

Examples:

$\rangle\ 1 + 1 :: \text{Exp Int}$
$1 + 1$
$\rangle\ \text{foldr}\ (+)\ 0\ [1, 2, 3] :: \text{Exp Int}$
$1 + (2 + (3 + 0))$
$\rangle\ \text{foldl}\ (+)\ 0\ [1, 2, 3] :: \text{Exp Int}$
$((0 + 1) + 2) + 3$

Folding arrays

Knowing Repa and extrapolating from Accelerate's map, the fold should not come as a surprise:

$$
\begin{aligned}
\mathsf{fold} \;&::\; (\mathsf{Shape}\ \mathsf{ix}, \mathsf{Elt}\ \mathsf{a}) \Rightarrow \\
&\; (\mathsf{Exp}\ \mathsf{a} \to \mathsf{Exp}\ \mathsf{a} \to \mathsf{Exp}\ \mathsf{a}) \to \mathsf{Exp}\ \mathsf{a} \to \\
&\; \mathsf{Acc}\ (\mathsf{Array}\ (\mathsf{ix} :. \mathsf{Int})\ \mathsf{a}) \to \mathsf{Acc}\ (\mathsf{Array}\ \mathsf{ix}\ \mathsf{a}) \\
\mathsf{foldAll} \;&::\; (\mathsf{Shape}\ \mathsf{sh}, \mathsf{Elt}\ \mathsf{a}) \Rightarrow \\
&\; (\mathsf{Exp}\ \mathsf{a} \to \mathsf{Exp}\ \mathsf{a} \to \mathsf{Exp}\ \mathsf{a}) \to \mathsf{Exp}\ \mathsf{a} \to \\
&\; \mathsf{Acc}\ (\mathsf{Array}\ \mathsf{sh}\ \mathsf{a}) \to \mathsf{Acc}\ (\mathsf{Scalar}\ \mathsf{a})
\end{aligned}
$$

No explicit forcing – and we get parallel folds automatically on the GPU.

Folding arrays

Knowing Repa and extrapolating from Accelerate's map, the fold should not come as a surprise:

$$
\begin{aligned}
\mathrm{fold} \; &:: (\mathrm{Shape}\ ix, \mathrm{Elt}\ a) \Rightarrow \\
&\quad (\mathrm{Exp}\ a \to \mathrm{Exp}\ a \to \mathrm{Exp}\ a) \to \mathrm{Exp}\ a \to \\
&\quad \mathrm{Acc}\ (\mathrm{Array}\ (ix :. \mathrm{Int})\ a) \to \mathrm{Acc}\ (\mathrm{Array}\ ix\ a) \\
\mathrm{foldAll} \; &:: (\mathrm{Shape}\ sh, \mathrm{Elt}\ a) \Rightarrow \\
&\quad (\mathrm{Exp}\ a \to \mathrm{Exp}\ a \to \mathrm{Exp}\ a) \to \mathrm{Exp}\ a \to \\
&\quad \mathrm{Acc}\ (\mathrm{Array}\ sh\ a) \to \mathrm{Acc}\ (\mathrm{Scalar}\ a)
\end{aligned}
$$

No explicit forcing – and we get parallel folds automatically on the GPU.

Lots of other operations: zip, backpermute, slice, ...

More conversions:

Between `Exp` and `Acc`:

```
unit     :: Elt e ⇒ Exp e → Acc (Scalar e)
the      :: Elt e ⇒ Acc (Scalar e) → Exp e
constant :: Elt e ⇒ e → Exp e
```

From shapes to `Exp`:

```
index0 ::                            Exp DIM0
index1 :: Exp Int                  → Exp DIM1
index2 :: Exp Int → Exp Int → Exp DIM2
```

For example needed in:

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

Creating arrays directly on the GPU

This also needs "shape expressions":

$$\begin{aligned}\mathsf{generate} :: {} & (\mathsf{Shape\ sh}, \mathsf{Elt\ e}) \Rightarrow \\ & \mathsf{Exp\ sh} \rightarrow (\mathsf{Exp\ sh} \rightarrow \mathsf{Exp\ e}) \rightarrow \mathsf{Acc\ (Array\ sh\ e)}\end{aligned}$$

Similar to fromFunction in Repa.

$$\begin{aligned}\mathsf{fill} :: {} & (\mathsf{Shape\ sh}, \mathsf{Elt\ e}) \Rightarrow \\ & \mathsf{Expr\ sh} \rightarrow \mathsf{Exp\ e} \rightarrow \mathsf{Acc\ (Array\ sh\ e)}\end{aligned}$$

Uniformly fills an array.

Booleans and conditionals

Accelerate deviates from standard Haskell notation here, as Booleans are not overloaded:

$$
\begin{aligned}
&(==*) :: (\mathsf{IsScalar}\ e, \mathsf{Elt}\ e) \Rightarrow \mathsf{Exp}\ e \to \mathsf{Exp}\ e \to \mathsf{Exp\ Bool} \\
&(\&\&*)\ :: \mathsf{Exp\ Bool} \to \mathsf{Exp\ Bool} \to \mathsf{Exp\ Bool} \\
&\ldots \\
&(?)\quad :: \mathsf{Elt}\ e \Rightarrow \mathsf{Exp\ Bool} \to (\mathsf{Exp}\ e, \mathsf{Exp}\ e) \to \mathsf{Exp}\ e
\end{aligned}
$$

Note that using $(?)$ leads to SIMD divergence – only one branch can be executed at a time. In particular nested conditionals quickly remove all GPU parallelism.

Well-Typed