Utrecht University

July 11, 2024

Accelerate

Ivo Gabe de Wolff
AFP Summer School

Accelerate

Domain specific language for functional, parallel, array programming.

Embedded in Haskell: Accelerate is a library providing a ‘language’ within Haskell

We look at Accelerate from two perspectives:

- Perspective from a user,
- Perspective from an implementor

Parallel programming

Parallelism is needed for maximal performance,
and widely available on multi-core CPUs and massively-parallel GPUs.

Parallel programming

Parallelism is needed for maximal performance,
and widely available on multi-core CPUs and massively-parallel GPUs.

Parallelism is hard:
It is difficult to make a fast and correct algorithm.

Common parallel patterns

Parallel algorithms can often be built with **common patterns**:

- Map
- Reduction (fold)
- Prefix sum (scan)
- Stencil (map with neighbourhood)
- Scatter (permute)

Common parallel patterns

Parallel algorithms can often be built with **common patterns**:

- Map
- Reduction (fold)
- Prefix sum (scan)
- Stencil (map with neighbourhood)
- Scatter (permute)

Domain-Specific Languages may provide these **patterns as building blocks** (combinators/functions).

Familiar combinators

Some of these combinators are familiar from Data.List.

Familiar combinators

Some of these combinators are familiar from Data.List.

Consider this function to compute a dot product:

$$\begin{aligned} &\mathit{dotp} :: [\mathrm{Float}] \to [\mathrm{Float}] \to \mathrm{Float} \\ &\mathit{dotp}\ \mathit{xs}\ \mathit{ys} = \mathbf{foldl}\ (+)\ 0\ (\mathbf{zipWith}\ (*)\ \mathit{xs}\ \mathit{ys}) \end{aligned}$$

Familiar combinators

Some of these combinators are familiar from Data.List.

Consider this function to compute a dot product:

$$\mathit{dotp} :: [\mathrm{Float}] \to [\mathrm{Float}] \to \mathrm{Float}$$
$$\mathit{dotp}\ \mathit{xs}\ \mathit{ys} = \mathbf{foldl}\ (+)\ 0\ (\mathbf{zipWith}\ (*)\ \mathit{xs}\ \mathit{ys})$$

Accelerate provides similar combinators, as a library in Haskell.
The types are different:

```haskell
import Prelude()
import Data.Array.Accelerate
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Deep embedding

Combinators in Accelerate build the **representation of a computation**.
They don't compute anything yet.

Deep embedding

Combinators in Accelerate build the **representation of a computation**.
They don't compute anything yet.

Acc *a* is the representation or AST of an array computation.

This is similar to the Expr *data type that you saw earlier:*

```
data Expr :: * -> * where
  LitI   :: Int -> Expr Int
  LitB   :: Bool -> Expr Bool
  IsZero :: Expr Int -> Expr Bool
  Plus   :: Expr Int -> Expr Int -> Expr Int
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Deep embedding

Combinators in Accelerate build the **representation of a computation**.
They don't compute anything yet.

$\mathrm{Acc}\ a$ is the representation or AST of an array computation.

This is similar to the Expr *data type that you saw earlier:*

```
data Expr :: * -> * where
  LitI   :: Int -> Expr Int
  LitB   :: Bool -> Expr Bool
  IsZero :: Expr Int -> Expr Bool
  Plus   :: Expr Int -> Expr Int -> Expr Int
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

run $:: \mathrm{Acc}\ a \to a$ executes such a computation, *similar to eval* $:: \mathrm{Expr}\ e \to e$.

Language design

Language only contains parallelisable constructs/combinators.

These combinators are data-parallel: parallelism is structured by the data.

Nested data (like matrices) are supported.
The inner sizes must be equal.
This allows for efficient parallel execution.

The type of the elements of arrays is restricted.

Design an embedding in Haskell that ensures these properties

Types

Accelerate has two types to represent computations:

- Acc for array computations
- Exp for scalar computations

Types

Accelerate has two types to represent computations:

- Acc for array computations
- Exp for scalar computations

Data is stored in (possibly multi-dimensional) arrays, with type Array *sh* *t*.

- *sh* denotes the shape or dimension of the array.
- *t* is the type of the elements of the array.

Types

Accelerate has two types to represent computations:

- Acc for array computations
- Exp for scalar computations

Data is stored in (possibly multi-dimensional) arrays, with type Array *sh* *t*.

- *sh* denotes the shape or dimension of the array.
- *t* is the type of the elements of the array.

Type classes Elt *t* and Shape *sh* range over the valid element types and shapes/dimensions.

Types

Accelerate has two types to represent computations:

- Acc for array computations
- Exp for scalar computations

Data is stored in (possibly multi-dimensional) arrays, with type $\mathrm{Array}\ sh\ t$.

- sh denotes the shape or dimension of the array.
- t is the type of the elements of the array.

Type classes $\mathrm{Elt}\ t$ and $\mathrm{Shape}\ sh$ range over the valid element types and shapes/dimensions.

The type of map is now: $(\mathrm{Shape}\ sh, \mathrm{Elt}\ t_1, \mathrm{Elt}\ t_2) \Rightarrow$

$$(\mathrm{Exp}\ t_1 \rightarrow \mathrm{Exp}\ t_2) \rightarrow \mathrm{Acc}\ (\mathrm{Array}\ sh\ t_1) \rightarrow \mathrm{Acc}\ (\mathrm{Array}\ sh\ t_2)$$

Shapes

A **shape** defines the **dimensionality** of an array.

Z is dimension zero.
sh :. Int is one dimension higher than sh.

Shapes are used for the **size** of arrays and **indices** of elements of arrays.

Type class $\mathrm{Shape}\ sh$ ranges over these shapes.

Building blocks: map

Apply the given function element-wise to an array.

$$\begin{aligned}\textbf{map} :: &\ (\text{Shape } sh, \text{Elt } t_1, \text{Elt } t_2)\\ &\Rightarrow (\text{Exp } t_1 \rightarrow \text{Exp } t_2)\\ &\rightarrow \text{Acc } (\text{Array } sh\ t_1)\\ &\rightarrow \text{Acc } (\text{Array } sh\ t_2)\end{aligned}$$

Building blocks: map

Apply the given function element-wise to an array.

$$\begin{aligned}\textbf{map} :: (&\text{Shape } sh, \text{Elt } t_1, \text{Elt } t_2)\\ &\Rightarrow (\text{Exp } t_1 \to \text{Exp } t_2)\\ &\to \text{Acc } (\text{Array } sh\ t_1)\\ &\to \text{Acc } (\text{Array } sh\ t_2)\end{aligned}$$

Variants:

- **stencil**: map with access to neighboring elements.

Building blocks: map

Apply the given function element-wise to an array.

$$\begin{aligned} \textbf{map} :: (&\text{Shape}\ sh, \text{Elt}\ t_1, \text{Elt}\ t_2) \\ &\Rightarrow (\text{Exp}\ t_1 \rightarrow \text{Exp}\ t_2) \\ &\rightarrow \text{Acc}\ (\text{Array}\ sh\ t_1) \\ &\rightarrow \text{Acc}\ (\text{Array}\ sh\ t_2) \end{aligned}$$

Variants:

- **stencil**: map with access to neighboring elements.
- **imap**: map with access to the index (besides the value).

Building blocks: map

Apply the given function element-wise to an array.

$$
\begin{aligned}
\textbf{map} :: &\ (\text{Shape}\ sh, \text{Elt}\ t_1, \text{Elt}\ t_2) \\
&\Rightarrow (\text{Exp}\ t_1 \rightarrow \text{Exp}\ t_2) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh\ t_1) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh\ t_2)
\end{aligned}
$$

Variants:

- **stencil**: map with access to neighboring elements.
- **imap**: map with access to the index (besides the value).
- **zipWith**: map with two input arrays.

Building blocks: map

Apply the given function element-wise to an array.

$$\textbf{map} :: (\text{Shape } sh, \text{Elt } t_1, \text{Elt } t_2) \Rightarrow (\text{Exp } t_1 \to \text{Exp } t_2) \to \text{Acc } (\text{Array } sh\ t_1) \to \text{Acc } (\text{Array } sh\ t_2)$$

Variants:

- **stencil**: map with access to neighboring elements.
- **imap**: map with access to the index (besides the value).
- **zipWith**: map with two input arrays.
- **zipWithN**: map with *N* input arrays.

Building blocks: generate

Construct a new array of the given size, by applying the function for each index.

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

Building blocks: fold

Reduces the innermost dimension of an array.

$$
\begin{aligned}
\textbf{fold} :: &\ (\mathrm{Shape}\ sh, \mathrm{Elt}\ t) \\
&\Rightarrow (\mathrm{Exp}\ t \rightarrow \mathrm{Exp}\ t \rightarrow \mathrm{Exp}\ t) \\
&\rightarrow \mathrm{Acc}\ (\mathrm{Array}\ (sh :. \mathrm{Int})\ t) \\
&\rightarrow \mathrm{Acc}\ (\mathrm{Array}\ sh\ t)
\end{aligned}
$$

A 1-dimensional vector becomes a 0-dimensional scalar (single value).
A 2-dimensional matrix becomes a 1-dimensional vector.

The function argument must be associative, for parallel execution.

Building blocks: scan

For each element, computes the reduced value of all previous elements.
Also known as prefix sum.

$$\textbf{scanl} :: (\text{Shape } sh, \text{Elt } t) \Rightarrow (\text{Exp } t \to \text{Exp } t \to \text{Exp } t) \to \text{Acc } (\text{Array } (sh :. \text{Int})\ t) \to \text{Acc } (\text{Array } (sh :. \text{Int})\ t)$$

Scan operates on the inner dimension.
In a matrix, the scan works per row.

The function argument must be associative, for parallel execution.

Building blocks: scan

For each element, computes the reduced value of all previous elements.
Also known as prefix sum.

$$\begin{aligned}\textbf{scanl} :: &\ (\text{Shape } sh, \text{Elt } t)\\ &\Rightarrow (\text{Exp } t \rightarrow \text{Exp } t \rightarrow \text{Exp } t)\\ &\rightarrow \text{Acc } (\text{Array } (sh :. \text{Int})\ t)\\ &\rightarrow \text{Acc } (\text{Array } (sh :. \text{Int})\ t)\end{aligned}$$

Scan operates on the inner dimension.
In a matrix, the scan works per row.

The function argument must be associative, for parallel execution.

Variants:

- Left-to-right or right-to-left scans.
- Inclusive or exclusive.
- Storing the total reduced value in a separate array.

Building blocks: permute

Performs random writes: each element of the input is written to some index.
Also known as prefix sum.

$$\begin{aligned}
\textbf{permute} &:: (\text{Shape}\ sh, \text{Shape}\ sh', \text{Elt}\ t) \\
&\Rightarrow (\text{Exp}\ sh \rightarrow \text{Exp}\ sh') \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh\ t) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh'\ t) \\
\textbf{permute}\ & indexTransform\ \ input = \ldots
\end{aligned}$$

Each index from the input is mapped to an index in the output.

Building blocks: permute

Performs random writes: each element of the input is written to some index.
Also known as prefix sum.

$$
\begin{aligned}
\textbf{permute} &:: (\text{Shape}\ sh, \text{Shape}\ sh', \text{Elt}\ t) \\
&\Rightarrow (\text{Exp}\ sh \rightarrow \text{Maybe}\ (\text{Exp}\ sh')) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh\ t) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh'\ t) \\
\textbf{permute}\ & \ indexTransform\ \ input = \ldots
\end{aligned}
$$

The index transformation may be partial: some elements of the input are skipped.

Building blocks: permute

Performs random writes: each element of the input is written to some index.
Also known as prefix sum.

$$
\begin{aligned}
\textbf{permute} &:: (\text{Shape}\ sh, \text{Shape}\ sh', \text{Elt}\ t) \\
&\Rightarrow \text{Acc}\ (\text{Array}\ sh'\ t) \\
&\rightarrow (\text{Exp}\ sh \rightarrow \text{Maybe}\ (\text{Exp}\ sh')) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh\ t) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh'\ t) \\
\textbf{permute}&\ \ \mathit{defaults}\ \ \mathit{indexTransform}\ \ \mathit{input} = \ldots
\end{aligned}
$$

The index transformation might not be surjective: some elements of the output may not be covered.

Building blocks: permute

Performs random writes: each element of the input is written to some index.
Also known as prefix sum.

$$
\begin{aligned}
\textbf{permute} :: &\ (\text{Shape}\ sh, \text{Shape}\ sh', \text{Elt}\ t) \\
&\Rightarrow (\text{Exp}\ t \rightarrow \text{Exp}\ t \rightarrow \text{Exp}\ t) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh'\ t) \\
&\rightarrow (\text{Exp}\ sh \rightarrow \text{Maybe}\ (\text{Exp}\ sh')) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh\ t) \\
&\rightarrow \text{Acc}\ (\text{Array}\ sh'\ t) \\
\textbf{permute}\ &\ combine\ \ defaults\ \ indexTransform\ \ input = \ldots
\end{aligned}
$$

It takes a combination function to combine the value from the input with the existing value.

Building blocks: permute

Performs random writes: each element of the input is written to some index.
Also known as prefix sum.

$$\begin{aligned}
\textbf{permute} &:: (\text{Shape } sh, \text{Shape } sh', \text{Elt } t) \\
&\Rightarrow (\text{Exp } t \to \text{Exp } t \to \text{Exp } t) \\
&\to \text{Acc } (\text{Array } sh' \; t) \\
&\to (\text{Exp } sh \to \text{Maybe } (\text{Exp } sh')) \\
&\to \text{Acc } (\text{Array } sh \; t) \\
&\to \text{Acc } (\text{Array } sh' \; t) \\
\textbf{permute} & \;\; combine \;\; defaults \;\; indexTransform \;\; input = \ldots
\end{aligned}$$

This is also important if the index transformation is not injective: multiple values can then be mapped to the same index.

Building blocks: permute

Performs random writes: each element of the input is written to some index.
Also known as prefix sum.

$$
\begin{aligned}
\textbf{permute} :: \ & (\mathrm{Shape}\ sh, \mathrm{Shape}\ sh', \mathrm{Elt}\ t) \\
& \Rightarrow (\mathrm{Exp}\ t \to \mathrm{Exp}\ t \to \mathrm{Exp}\ t) \\
& \to \mathrm{Acc}\ (\mathrm{Array}\ sh'\ t) \\
& \to (\mathrm{Exp}\ sh \to \mathrm{Maybe}\ (\mathrm{Exp}\ sh')) \\
& \to \mathrm{Acc}\ (\mathrm{Array}\ sh\ t) \\
& \to \mathrm{Acc}\ (\mathrm{Array}\ sh'\ t) \\
\textbf{permute}\ & \mathit{combine}\ \ \mathit{defaults}\ \ \mathit{indexTransform}\ \ \mathit{input} = \ldots
\end{aligned}
$$

The combination function is often *const*, which will simply overwrite the old value with the new value, or $+$, which adds the new value to the old value.

What can you express in this language?

More than you think!

Examples:

- Filter & partition
- Quicksort

Filter

(On the blackboard)
Use **map** to create an array where at each element, 1 denotes that the element is preserved and 0 that it is dropped.
Use **scanl** $(+)$ 0 to compute for each element, the index in the output it should be written to.
Use **permute** to write the elements to the correct indices.
See `https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/src/Data.Array.Accelerate.Prelude.html#filter`.

Partition

Partition is an extension of filter, with one case for the True-elements and one for the False-elements

Quicksort

$$
\begin{aligned}
&\mathit{quicksort}\ [\,] = [\,] \\
&\mathit{quicksort}\ (p : \mathit{xs}) = \\
&\quad \mathit{quicksort}\ \mathit{smaller} +\!\!+ [\,p\,] +\!\!+ \mathit{quicksort}\ \mathit{greater} \\
&\quad \mathbf{where} \\
&\qquad \mathit{smaller} = \mathit{filter}\ (< p)\ \mathit{xs} \\
&\qquad \mathit{larger} = \mathit{filter}\ (>= p)\ \mathit{xs}
\end{aligned}
$$

Quicksort

$$
\begin{aligned}
&\mathit{quicksort}\ [\,] = [\,] \\
&\mathit{quicksort}\ (p : \mathit{xs}) = \\
&\quad \mathit{quicksort}\ \mathit{smaller} \mathbin{+\!\!+} [\,p\,] \mathbin{+\!\!+} \mathit{quicksort}\ \mathit{greater} \\
&\quad \textbf{where} \\
&\qquad \mathit{smaller} = \mathit{filter}\ (< p)\ \mathit{xs} \\
&\qquad \mathit{larger} = \mathit{filter}\ (>= p)\ \mathit{xs}
\end{aligned}
$$

How can we make this run in parallel?

Quicksort

$$
\begin{aligned}
&\mathit{quicksort}\ [\,] = [\,]\\
&\mathit{quicksort}\ (p : \mathit{xs}) =\\
&\quad \mathit{quicksort}\ \mathit{smaller} +\!\!+ [\,p\,] +\!\!+ \mathit{quicksort}\ \mathit{greater}\\
&\quad \textbf{where}\\
&\qquad \mathit{smaller} = \mathit{filter}\ (< p)\ \mathit{xs}\\
&\qquad \mathit{larger} = \mathit{filter}\ (>= p)\ \mathit{xs}
\end{aligned}
$$

How can we make this run in parallel?

We now know how to perform partition in parallel.

GPUs like to work on the entire array,
instead of the shorter segments (*smaller* and *larger*).

Segments

Instead of performing two recursive calls,
we keep working on the **entire array**.

Segments

Instead of performing two recursive calls,
we keep working on the **entire array**.

We mark **segments** in that array.
Each segment in the parallel algorithm corresponds to a call in the sequential algorithm.

Segments

Instead of performing two recursive calls,
we keep working on the **entire array**.

We mark **segments** in that array.
Each segment in the parallel algorithm corresponds to a call in the sequential algorithm.

Partitioning happens via scans. We can use **segmented scans** instead, which ‘reset’ at segment boundaries.

Segmented scans

A segmented scan can be defined in terms of a normal scan:

$$segmentedScanl\ f\ segmentBoundaries\ values = \ldots\ \mathbf{scanl1}\ (segmented\ f)\ \ldots$$

We can define our own combinators in terms of the basic combinators of Accelerate.

Note that we represent segment boundaries as a vector of booleans. Segments are sometimes also represented by a vector of segment lengths, but that has more computational overhead in this case.

An implementation is available at `https://github.com/tmcdonell/containers-accelerate/blob/master/src/Data/Array/Accelerate/Data/Sort/Quick.hs`.

How does it work?

We’ve now seen how you can use Accelerate.

But how does it work?

The Accelerate compiler

Accelerate consists of a compiler, targetting multi-core CPUs and GPUs.

But it is only a library: it doesn't require special trickery from the Haskell compiler.

This works via a deep embedding.

Deep embedding

Combinators in Accelerate build the **representation of a computation**.
They don't compute anything yet.

Deep embedding

Combinators in Accelerate build the **representation of a computation**.
They don't compute anything yet.

Acc *a* is the representation or AST of an array computation.
Exp *t* is the representation or AST of a scalar computation.

Deep embedding

Combinators in Accelerate build the **representation of a computation**.
They don't compute anything yet.

$\mathrm{Acc}\ a$ is the representation or AST of an array computation.
$\mathrm{Exp}\ t$ is the representation or AST of a scalar computation.

run $:: \mathrm{Acc}\ a \rightarrow a$ passes that representation to our compiler and executes the compiled program.

Deep embedding (simplified)

Instead of directly executing the computation:

$$add :: \text{Exp Int} \to \text{Exp Int} \to \text{Exp Int}$$
$$add\ x\ y = x + y$$

Deep embedding (simplified)

Instead of directly executing the computation, combinators construct a representation or AST of the computation:

$$
\begin{aligned}
&\mathit{add} :: \mathrm{Exp\ Int} \to \mathrm{Exp\ Int} \to \mathrm{Exp\ Int} \\
&\mathit{add}\ x\ y = \mathrm{Add}\ x\ y
\end{aligned}
$$

Deep embedding (simplified)

Instead of directly executing the computation, combinators construct a representation or AST of the computation:

$$add :: \text{Exp Int} \to \text{Exp Int} \to \text{Exp Int}$$
$$add\ x\ y = \text{Add}\ x\ y$$

Data type Exp is the representation or AST of a program:

$$\textbf{data}\ \text{Exp}\ t\ \textbf{where}$$
$$\text{Add} :: \text{Exp Int} \to \text{Exp Int} \to \text{Exp Int}$$
$$\text{ConstInt} :: \text{Int} \to \text{Exp Int}$$
$$\text{ConstBool} :: \text{Bool} \to \text{Exp Bool}$$
$$\ldots$$

Using a Generalized Algebraic Data Type (GADT), we can preserve the type-safety during the compilation.

Variables in the embedding

During the construction of the program, we can use **variables**, **functions** and **let-bindings** just like we normally do in Haskell.

$$\textbf{map}\ (\lambda x \rightarrow 2\ *\ x + 1)\ xs$$

Variables in the embedding

During the construction of the program, we can use **variables**, **functions** and **let-bindings** just like we normally do in Haskell.

$$\textbf{map}\ (\lambda x \rightarrow 2 * x + 1)\ xs$$

It would have been annoying if the embedding had its own syntax for functions and variables, like:

$$\textbf{map}\ (\textbf{fn}\ \text{“x”}\ (2 * \textbf{var}\ \text{“x”} + 1))\ xs$$

Variables in the embedding

During the construction of the program, we can use **variables**, **functions** and **let-bindings** just like we normally do in Haskell.

$$\textbf{map}\ (\lambda x \rightarrow 2\ *\ x + 1)\ \mathit{xs}$$

It would have been annoying if the embedding had its own syntax for functions and variables, like:

$$\textbf{map}\ (\textbf{fn}\ \text{“x”}\ (2\ *\ \textbf{var}\ \text{“x”} + 1))\ \mathit{xs}$$

The first is a **higher-order embedding** and is used in the public API.
It is more convenient for the user and provides more type-safety.

Variables in the embedding

During the construction of the program, we can use **variables**, **functions** and **let-bindings** just like we normally do in Haskell.

$$\textbf{map}\ (\lambda x \rightarrow 2\ *\ x + 1)\ xs$$

It would have been annoying if the embedding had its own syntax for functions and variables, like:

$$\textbf{map}\ (\textbf{fn}\ \text{“x”}\ (2\ *\ \textbf{var}\ \text{“x”} + 1))\ xs$$

The first is a **higher-order embedding** and is used in the public API. It is more convenient for the user and provides more type-safety.

The second is a **first-order embedding** and is used internally. It is easier to write program analyses and optimisations in such a representation.

Compile-time?

In deep embedded languages like Accelerate, there are two ‘compile times’:

- **Host compile-time**: the host language is compiled before running the program.
- **DSL compile-time**: the compiler for the DSL is ran during the run-time of the program, or with meta-programming (like Template Haskell) at host compile-time.

Compile-time?

In deep embedded languages like Accelerate, there are two ‘compile times’:

- **Host compile-time**: the host language is compiled before running the program.
- **DSL compile-time**: the compiler for the DSL is ran during the run-time of the program, or with meta-programming (like Template Haskell) at host compile-time.

Preferably, any problems with a program are reported during host compile-time.
Verifying analyses thus preferably use the type system of the host language.

Compiler pipeline

Input
↓

We have now seen how the input to the compiler is constructed.

The compiler performs several passes over the input:

Compiler pipeline

Input

↓

Sharing recovery

We have now seen how the input to the compiler is constructed.

The compiler performs several passes over the input:

Compiler pipeline

We have now seen how the input to the compiler is constructed.

The compiler performs several passes over the input:

Input

↓

Sharing recovery

↓

Fusion & other optimisations

Compiler pipeline

We have now seen how the input to the compiler is constructed.

The compiler performs several passes over the input:

Input
↓
Sharing recovery
↓
Fusion & other optimisations
↓
Code generation to LLVM IR
↓
LLVM compilation
↓
Machine code

Sharing recovery

Accelerate programs may be constructed using let-bindings in Haskell:

$$\textbf{let}\ xs = \textbf{map}\ (\lambda x \rightarrow \ldots)\ \ldots$$
$$\textbf{in}\ \ldots\ xs\ \ldots\ xs$$

Sharing recovery

Accelerate programs may be constructed using let-bindings in Haskell:

$$\textbf{let}\ \mathit{xs} = \textbf{map}\ (\lambda x \rightarrow \ldots)\ \ldots$$
$$\textbf{in}\ \ldots\ \mathit{xs}\ \ldots\ \mathit{xs}$$

This creates a tree (or graph) where **map** is present twice.

Sharing recovery

Accelerate programs may be constructed using let-bindings in Haskell:

$$\textbf{let}\ xs = \textbf{map}\ (\lambda x \to \ldots)\ \ldots$$
$$\textbf{in}\ \ldots\ xs\ \ldots\ xs$$

This creates a tree (or graph) where **map** is present twice.

Sharing recovery converts trees with shared nodes to a tree with let-bindings.

Sharing recovery

Accelerate programs may be constructed using let-bindings in Haskell:

$$\textbf{let}\ xs = \textbf{map}\ (\lambda x \to \ldots)\ \ldots$$
$$\textbf{in}\ \ldots\ xs\ \ldots\ xs$$

This creates a tree (or graph) where **map** is present twice.

Sharing recovery converts trees with shared nodes to a tree with let-bindings.

Add let-bindings to the data type:

$$\textbf{data}\ \mathrm{Exp}\ t\ \textbf{where}$$
$$\mathrm{Let}\ :: \mathrm{Id} \to \mathrm{Exp}\ t_1 \to \mathrm{Exp}\ t_2 \to \mathrm{Exp}\ t_2$$

Variables

And add variables:

$$
\begin{aligned}
&\textbf{data}\ \mathrm{Exp}\ t\ \textbf{where}\\
&\quad \mathrm{Let}\ :: \mathrm{Id} \to \mathrm{Exp}\ t_1 \to \mathrm{Exp}\ t_2 \to \mathrm{Exp}\ t_2\\
&\quad \mathrm{Var}\ :: \mathrm{Id} \to \mathrm{Exp}\ t
\end{aligned}
$$

What is the type of a variable?

Variables

And add variables:

$$\textbf{data}\ \mathrm{Exp}\ t\ \textbf{where}$$
$$\quad \mathrm{Let}\ :: \mathrm{Id} \rightarrow \mathrm{Exp}\ t_1 \rightarrow \mathrm{Exp}\ t_2 \rightarrow \mathrm{Exp}\ t_2$$
$$\quad \mathrm{Var}\ :: \mathrm{Id} \rightarrow \mathrm{Exp}\ t$$

What is the type of a variable?

That depends on the environment.

We already added type variable *t* for the result of an expression, let's also add a type variable *env* for the environment.

Typed environments

The environment becomes a type-level list.
De Bruijn indices (for variable names) index into that list.

Typed environments

The environment becomes a type-level list.
De Bruijn indices (for variable names) index into that list.

Consider this environment:

$$((((), \text{Bool}), \text{Float}), \text{Int})$$

Typed environments

The environment becomes a type-level list.
De Bruijn indices (for variable names) index into that list.

Consider this environment:

$$((((), \mathrm{Bool}), \mathrm{Float}), \mathrm{Int})$$

The most-recently introduced variable has type Int.
That variable has De Bruijn index 0 (assuming zero-based indices).

Typed environments

The environment becomes a type-level list.
De Bruijn indices (for variable names) index into that list.

Consider this environment:

$$((((), \mathrm{Bool}), \mathrm{Float}), \mathrm{Int})$$

The first introduced variable has type Bool.
That variable has De Bruijn index 2 (assuming zero-based indices).

GADTs for typed environments

When introducing a variable, we extend the list:

$$\text{Let} :: \text{Exp}\ \mathit{env}\ t_1 \to \text{Exp}\ (\mathit{env}, t_1)\ t_2 \to \text{Exp}\ \mathit{env}\ t_2$$

GADTs for typed environments

When introducing a variable, we extend the list:

$$\mathrm{Let} :: \mathrm{Exp}\ env\ t_1 \to \mathrm{Exp}\ (env, t_1)\ t_2 \to \mathrm{Exp}\ env\ t_2$$

Define a data type $\mathrm{Idx}\ env\ t$.
It guarantees that an index corresponds to a variable of type t in environment env:

$$\begin{aligned}&\textbf{data}\ \mathrm{Idx}\ env\ t\ \textbf{where}\\ &\quad \mathrm{ZeroIdx} :: \mathrm{Idx}\ (env, t)\ t\\ &\quad \mathrm{SuccIdx} :: \mathrm{Idx}\ env\ t \to \mathrm{Idx}\ (env, s)\ t\end{aligned}$$

GADTs for typed environments

When introducing a variable, we extend the list:

$$\text{Let} :: \text{Exp}\ env\ t_1 \to \text{Exp}\ (env, t_1)\ t_2 \to \text{Exp}\ env\ t_2$$

Define a data type $\text{Idx}\ env\ t$.
It guarantees that an index corresponds to a variable of type t in environment env:

$$\begin{aligned}&\textbf{data}\ \text{Idx}\ env\ t\ \textbf{where}\\ &\quad \text{ZeroIdx} :: \text{Idx}\ (env, t)\ t\\ &\quad \text{SuccIdx} :: \text{Idx}\ env\ t \to \text{Idx}\ (env, s)\ t\end{aligned}$$

Then we can add a constructor for variables in Exp:

$$\text{Var} :: \text{Idx}\ env\ t \to \text{Exp}\ env\ t$$

Fusion

- The DSL advocates splitting the program into many small steps
- Naive: one (parallel) loop per combinator
- Fusion: combine multiple combinators into one loop

Fusion

- The DSL advocates splitting the program into many small steps
- Naive: one (parallel) loop per combinator
- Fusion: combine multiple combinators into one loop

Fusion minimizes:

- Number of (parallel) loops
- Number of (intermediate) arrays
- Number of memory operations

Fusion examples

$\textbf{map}\ f\ (\textbf{map}\ g\ xs)$

is coverted to:

$\textbf{map}\ (f \circ g)\ xs$

Fusion examples

This can also be fused:

$\textbf{fold}\ (+)\ 0\ (\textbf{map}\ f\ xs)$

This cannot be expressed in the same language,
we have a different IR for after this optimisation.

Conclusion

Accelerate makes **data-parallelism accessible** via an **embedding** in Haskell.

- Reuses the syntax and type system of Haskell:
  No need to implement that ourselves
- Restricted to the syntax and type system of Haskell

Using **GADTs**, we can **preserve type-safety** in the compiler. The types guarantee:

- The type of expressions
- The types of variables in the environment

Note that Haskell makes this easy; many other compilers only have type-safety for the type of expressions, or neither of these.