Optimising Purely Functional GPU Programs

Trevor L. McDonell Manuel M. T. Chakravarty Gabriele Keller Ben Lippmeier
University of New South Wales, Australia
{tmcdonell,chak,keller,benl}@cse.unsw.edu.au

Abstract

Purely functional, embedded array programs are a good match for SIMD hardware, such as GPUs. However, the naive compilation of such programs quickly leads to both code explosion and an excessive use of intermediate data structures. The resulting slowdown is not acceptable on target hardware that is usually chosen to achieve high performance.

In this paper, we discuss two optimisation techniques, *sharing recovery* and *array fusion*, that tackle code explosion and eliminate superfluous intermediate structures. Both techniques are well known from other contexts, but they present unique challenges for an embedded language compiled for execution on a GPU. We present novel methods for implementing sharing recovery and array fusion, and demonstrate their effectiveness on a set of benchmarks.

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

Keywords Arrays; Data parallelism; Embedded language; Dynamic compilation; GPGPU; Haskell; Sharing recovery; Array fusion

1. Introduction

Recent work on stream fusion [12], the `vector` package [23], and the parallel array library Repa [17, 19, 20] has demonstrated that (1) the performance of purely functional array code in Haskell can be competitive with that of imperative programs and that (2) purely functional array code lends itself to an efficient parallel implementation on control-parallel multicore CPUs.

So far, the use of purely functional languages for programming data parallel SIMD hardware such as GPUs (Graphical Processing Units) has been less successful. Vertigo [13] was an early Haskell EDSL producing DirectX 9 shader code, though no runtime performance figures were reported. Nikola [22] produces code competitive with CUDA, but without supporting generative functions like `replicate` where the result size is not statically fixed. Obsidian [10] is additionally restricted to only processing arrays of a fixed, implementation dependent size. Additionally, both Nikola and Obsidian can only generate single GPU kernels at a time, so that in programs consisting of multiple kernels the intermediate data structures must be shuffled back and forth across the CPU-GPU bus.

We recently presented *Accelerate*, an EDSL and skeleton-based code generator targeting the CUDA GPU development environment [8]. In the present paper, we present novel methods for optimising the code using *sharing recovery* and *array fusion*.

Sharing recovery for embedded languages recovers the sharing of let-bound expressions that would otherwise be lost due to the embedding. Without sharing recovery, the value of a let-bound expression is recomputed for every use of the bound variable. In contrast to prior work [14] that decomposes expression trees into graphs and fails to be type preserving, our novel algorithm preserves both the tree structure and typing of a deeply embedded language. This enables our runtime compiler to be similarly type preserving and simplifies the backend by operating on a tree-based intermediate language.

Array fusion eliminates the intermediate values and additional GPU kernels that would otherwise be needed when successive bulk operators are applied to an array. Existing methods such as `foldr/build` fusion [15] and stream fusion [12] are not applicable to our setting as they produce tail-recursive loops, rather than the GPU kernels we need for Accelerate. The NDP2GPU system of [4] *does* produce fused GPU kernels, but is limited to simple map/map fusion. We present a fusion method partly inspired by Repa's *delayed arrays* [17] that fuses more general producers and consumers, while retaining the combinator based program representation that is essential for GPU code generation using skeletons.

With these techniques, we provide a high-level programming model that supports shape-polymorphic maps, generators, reductions, permutation and stencil-based operations, while maintaining performance that often approaches hand-written CUDA code.

In summary, we make the following contributions:

- We introduce a novel sharing recovery algorithm for type-safe ASTs, preserving the tree structure (Section 3).
- We introduce a novel approach to array fusion for embedded array languages (Section 4).
- We present benchmarks for several applications including Black-Scholes options pricing, Canny edge detection, and a fluid flow simulation, including a comparison with hand-optimised CPU and GPU code (Section 5).

This paper builds on our previous work on a skeleton-based CUDA code generator for Accelerate [8]. Although we motivate and evaluate our novel approaches to sharing recovery and array fusion in this context, our contribution is not limited to Accelerate. Specifically, our sharing recovery applies to any embedded language based on the typed lambda calculus and our array fusion applies to any dynamic compiler targeting bulk-parallel SIMD hardware.

We discuss related work in detail in Section 6. The source code for Accelerate including our benchmark code is available from `https://github.com/AccelerateHS/accelerate`.

2. Optimising embedded array code

Accelerate is a *domain specific* language, consisting of a carefully selected set of array operators that we can produce efficient GPU code for. Accelerate is *embedded* in Haskell, meaning that we write Accelerate programs using Haskell syntax, but do not compile arbitrary Haskell programs to GPU machine code. Accelerate is stratified into *array computations*, wrapped in a type constructor `Acc`, and *scalar expressions*, represented by terms of type `Exp t`, where `t` is the type of value produced by the expression. This stratification is necessary due to the hardware architecture of GPUs and their reliance on SIMD parallelism, as we discussed in our previous work [8].

2.1 Too many kernels

For example, to compute a dot product, we use:

```
dotp :: Vector Float -> Vector Float
     -> Acc (Scalar Float)
dotp xs ys = let xs' = use xs
                 ys' = use ys
             in  fold (+) 0 (zipWith (*) xs' ys')
```

The function `dotp` consumes two one-dimensional arrays (`Vector`) of floating-point values and produces a single (`Scalar`) floating-point result. From the return type `Acc (Scalar Float)`, we see that `dotp` is an embedded Accelerate computation, rather than vanilla Haskell code.

The functions `zipWith` and `fold` are defined in our library `Data.Array.Accelerate`, and have *massively parallel* GPU implementations with the following type signatures:

```
zipWith :: (Shape sh, Elt a, Elt b, Elt c)
        => (Exp a -> Exp b -> Exp c)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
        -> Acc (Array sh c)

fold    :: (Shape ix, Elt a)
        => (Exp a -> Exp a -> Exp a)
        -> Exp a
        -> Acc (Array (ix:.Int) a)
        -> Acc (Array ix a)
```

The type classes `Elt` and `Shape` indicate that a type is admissible as an array element and array shape, respectively. Array shapes are denoted by type-level lists formed from `Z` and `(:.)` — for example, `Z:.Int:.Int` is the shape of a two-dimensional array (see [8, 17] for details). The type signatures of `zipWith` and `fold` clearly show the stratification into scalar computations using the `Exp` type constructor, and array computations wrapped in `Acc`.

The arguments to `dotp` are of plain Haskell type `Vector Float`. To make these arguments available to the Accelerate computation they must be embedded with the `use` function:

```
use :: Arrays arrays => arrays -> Acc arrays
```

This function is overloaded so that it can accept entire tuples of `arrays`. Operationally, `use` copies array data from main memory to GPU memory, in preparation for processing by the GPU.

The above Haskell version of the GPU-accelerated dot product is certainly more compact than the corresponding CUDA C code. However, when compiled with the skeleton-based approach we described in previous work [8], it is also significantly slower. The CUDA C version executes in about half the time (see Table 2).

The slow-down is due to Accelerate generating one *GPU kernel function* for `zipWith` and another one for `fold`. In contrast, the CUDA C version only uses a single kernel. The use of two separate kernels requires an intermediate array to be constructed, and in a memory bound benchmark, such as `dotp`, this doubles the runtime. To eliminate this intermediate array we need to *fuse* the adjacent aggregate array computations. Although there is an existing body of work on array fusion, no existing method adequately deals with massively parallel GPU kernels. We present a suitable fusion framework as the first major contribution of this paper.

2.2 Too little sharing

As a second example, consider the pricing of European-style options using the Black-Scholes formula. The Accelerate program is in Figure 1. Given a vector of triples of underlying stock price, strike price, and time to maturity (in years), the Black-Scholes formula computes the price of a call and a put option. The function `callput` evaluates the Black-Scholes formula for a single triple, and `blackscholes` maps it over a vector of triples, such that all individual applications of the formula are executed in parallel.

If we compare the performance of the GPU code generated by Accelerate with that of an equivalent implementation in CUDA C, the Accelerate version is almost twenty times slower. As `blackscholes` includes only one aggregate array computation, the problem can't be a lack of fusion. Instead, as we noted previously [8], it is due to the embedding of Accelerate in Haskell.

The function `callput` includes a significant amount of sharing: the helper functions `cnd'`, and hence also `horner`, are used twice —for `d1` and `d2`— and its argument `d` is used multiple times in the body. Our embedded implementation of Accelerate reifies the abstract syntax of the (deeply) embedded language in Haskell. Consequently, each occurrence of a let-bound variable in the source program creates a separate unfolding of the bound expression in the compiled code.

This is a well known problem that has been solved elegantly by the *sharing recovery* algorithm of Gill [14], which makes use of stable names. Unfortunately, Gill's original approach (1) reifies the abstract syntax in *graph* form and (2) it assumes an *untyped* syntax representation. In contrast, Accelerate is based on a typed tree representation using GADTs and type families in conjunction with type-preserving compilation in most phases. In other words, we use Haskell's type checker to statically ensure many core properties of our Accelerate compiler. The fact that the compiler for the *embedded language* is type preserving, means that many bugs in the Accelerate compiler itself are caught by the Haskell compiler during development. This in turn reduces the number of Accelerate compiler bugs that the end-user might be exposed to.

As we require typed trees, where sharing is represented by let bindings rather than untyped graphs, we cannot directly use Gill's approach to sharing recovery. Instead, we have developed a novel sharing recovery algorithm, which like Gill's, uses stable names, but unlike Gill's, operates on typed abstract syntax. Our algorithm produces a typed abstract syntax *tree*, and we are able to recover exactly those let bindings used in the source program. This is the second major contribution of this paper.

2.3 Summary

In summary, a straightforward skeleton-based implementation of an embedded GPU language suffers from two major inefficiencies: *lack of sharing* and *lack of fusion.* Both sharing recovery in embedded languages, and array fusion in functional languages have received significant prior attention. However, we have found that none of the existing techniques are adequate for a *type-preserving* embedded language compiler targeting *massively parallel SIMD hardware,* such as GPUs.

3. Sharing recovery

Gill [14] proposed to use *stable names* [27] to recover the sharing of source terms in a deeply embedded language. The stable names

```
blackscholes :: Vector (Float, Float, Float)
             -> Acc (Vector (Float, Float))
blackscholes = map callput . use
  where
  callput x =
    let (price, strike, years) = unlift x
        r       = constant riskfree
        v       = constant volatility
        v_sqrtT = v * sqrt years
        d1      = (log (price / strike) +
                   (r + 0.5 * v * v) * years) / v_sqrtT
        d2      = d1 - v_sqrtT
        cnd d   = let c = cnd' d in d >* 0 ? (1.0 - c, c)
        cndD1   = cnd d1
        cndD2   = cnd d2
        x_expRT = strike * exp (-r * years)
    in
    lift ( price * cndD1 - x_expRT * cndD2
         , x_expRT * (1.0 - cndD2) - price * (1.0 - cndD1))

riskfree, volatility :: Float
riskfree   = 0.02
volatility = 0.30

horner :: Num a => [a] -> a -> a
horner coeff x = x * foldr1 madd coeff
  where
    madd a b = a + x*b

cnd' :: Floating a => a -> a
cnd' d =
  let poly     = horner coeff
      coeff    = [0.31938153, -0.356563782,
                  1.781477937, -1.821255978,
                  1.330274429]
      rsqrt2pi = 0.39894228040143267793994605993438
      k        = 1.0 / (1.0 + 0.2316419 * abs d)
  in
  rsqrt2pi * exp (-0.5*d*d) * poly k
```

Figure 1. Black-Scholes option pricing in Accelerate

of two Haskell terms are equal only when the terms are represented by the same heap structure in memory. Likewise, when the abstract syntax trees (ASTs) of two terms of an embedded language have the same stable name, we know that they represent the same value. In this case we should combine them into one shared AST node. As the stable name of an expression is an intensional property, it can only be determined in Haskell's `IO` monad.

Accelerate's runtime compiler preserves source types throughout most of the compilation process [8]. In particular, it converts the source representation based on higher-order abstract syntax (HOAS) to a type-safe internal representation based on de Bruijn indices. Although developed independently, this conversion is like the *unembedding* of Atkey et al. [1]. Unembedding and sharing recovery necessarily need to go hand in hand for the following reasons. Sharing recovery must be performed on the source representation; otherwise, sharing will already have been lost. Nevertheless, we cannot perform sharing recovery on higher-order syntax as we need to traverse below the lambda abstractions used in the higher-order syntax representation. Hence, both need to go hand in hand.

As shown by Atkey et al. [1], the conversion of HOAS in Haskell can be implemented in a type-preserving manner with the exception of one untyped, but dynamically checked environment look up, using Haskell's `Typeable`. To preserve maximum type safety, we do not want any further operations that are not type preserving when adding sharing recovery. Hence, we cannot use Gill's algorithm. The variant of Gill's algorithm used in Syntactic [2] does not apply either: it (1) also generates a graph and (2) discards static type information in the process. In contrast, our novel algorithm performs simultaneous sharing recovery and conversion from HOAS to a typed de Bruijn representation, where the result of the conversion is a tree (not a graph) with sharing represented by let bindings. Moreover, it is a type-preserving conversion whose only dynamically checked operation is the same environment lookup deemed unavoidable by Atkey et al. [1].

In the following, we describe our algorithm using plain typed lambda terms. This avoids much of the clutter that we would incur by basing the discussion on the full Accelerate language. In addition to its implementation in Accelerate, a working Haskell implementation of our algorithm for the plain lambda calculus can be found at `https://github.com/mchakravarty/hoas-conv`.

3.1 Our approach to sharing recovery

Before we formalise our sharing recovery algorithm in the following subsections, we shall illustrate the main idea. Consider the following source term:

```
let    inc  = (+) 1
in let nine = let three = inc 2
              in
              (*) three three
in
(-) (inc nine) nine
```

This term's abstract syntax DAG is the leftmost diagram in Figure 2. It uses `@` nodes to represent applications; as in this grammar:

$$
\begin{array}{ll}
T \to C & T^\tau \textbf{ where} \\
\quad \mid x & \quad C^\tau \quad :: \; T^\tau \\
\quad \mid \lambda x.\, T & \quad x^\tau \quad :: \; T^\tau \\
\quad \mid T_1 \texttt{ @ } T_2 & \quad \lambda x^{\tau_1}.T^{\tau_2} \quad :: \; T^{\tau_1 \to \tau_2} \\
C \to \langle \text{constants} \rangle & \quad T_1^{\tau_1 \to \tau_2} \texttt{ @ } T_2^{\tau_1} \quad :: \; T^{\tau_2}
\end{array}
$$

The left definition does not track types, whereas the right one does. We implement typed ASTs in Haskell with GADTs and work with typed representations henceforth. Typed HOAS conversion with sharing recover proceeds in three stages:

1. *Prune shared subterms:* A depth first traversal over the AST annotates each node with its unique stable name, where we build an occurrence map of how many times we've already visited each node. If we encounter a previously visited node, it represents a *shared subterm*, and we replace it by a placeholder containing its stable name. The second diagram in Figure 2 shows the outcome of this stage. Each node is labeled by a number that represents its stable name, and the dotted edges indicate where we encountered a previously visited, shared node. The placeholders are indicated by underlined stable names.
2. *Float shared terms:* All shared subterms float upwards in the tree to just above the lowest node that dominates all edges to the original position of that shared subterm — see the third diagram in Figure 2. Floated subterms are referenced by circled stable names located *above* the node that they floated to. If a node collects more than one shared subterm, the subterm whose origin is deeper in the original term goes on top — here, 9 on top of 5. Nested sharing leads to subterms floating up inside other floated subterms — here, 8 stays inside the subterm rooted in 5.

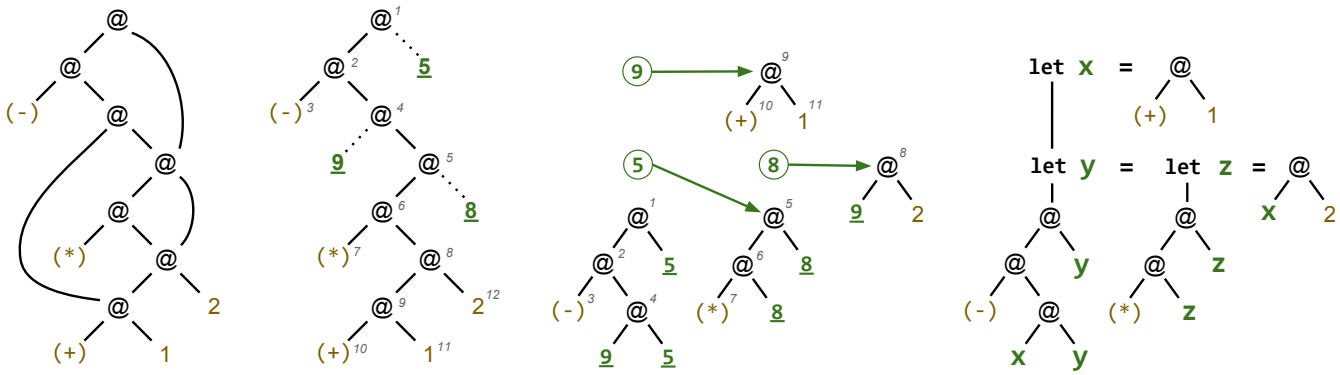


Figure 2. Recovering sharing in an example term

3. *Binder introduction:* Each floated subterm gets let-bound right above the node it floated to (rightmost diagram in Figure 2). While we use explicit, bound names in the figure, we introduce de Bruijn indices at the same time as introducing the lets.

3.2 Prune shared subterms

First, we identify and prune shared subtrees, producing a pruned tree of the following form (second diagram in Figure 2):

$$
\begin{array}{lll}
{}^{\circ}T^{\tau}\ \textbf{where} & & \\
\quad \ell & :: {}^{\circ}T^{\tau} & \text{-- binder conversion level} \\
\quad \underline{\nu}^{\tau} & :: {}^{\circ}T^{\tau} & \text{-- pruned subtree (name)} \\
\quad C^{\tau} & :: {}^{\circ}T^{\tau} & \\
\quad \lambda\ell.{}^{\circ}T^{\tau_2} & :: {}^{\circ}T^{\tau_1 \to \tau_2} & \\
\quad {}^{\circ}T_1^{\tau_1 \to \tau_2} \mathbin{@} {}^{\circ}T_2^{\tau_1} & :: {}^{\circ}T^{\tau_2} &
\end{array}
$$

A stable name (here, of type $Name$) associates a unique name with each unique term node, so that two terms with the same stable name are identical, and are represented by the same data structure in memory. Here, we denote the stable name of a term as a superscript during pattern matching — e.g., 1^{ν} is a constant with stable name ν, just as in the second and third diagram in Figure 2.

An *occurrence map*, $\Omega :: Name \mapsto Int$, is a finite map that determines the number of occurrences of a *Name* that we encountered during a traversal. The expression $\Omega\nu$ yields the number of occurrences of the name ν, and we have $\nu \in \Omega \equiv (\Omega\nu > 0)$. To add an occurrence to Ω, we write $\nu \triangleright \Omega$. We will see in the next subsection that we cannot simplify Ω to be merely a set of occurring names. We need the actual occurrence count to determine where shared subterms should be let-bound.

The identification and pruning of shared subtrees is formalised by the following function operating on *closed* terms from T^{τ}:

$$
\begin{array}{l}
prune :: Level \to (Name \mapsto Int) \to T^{\tau} \to ((Name \mapsto Int), {}^{\circ}T^{\tau}) \\
prune\ \ell\ \Omega\ e^{\nu} \mid \nu \in \Omega \quad = (\nu \triangleright \Omega,\ \underline{\nu}) \\
prune\ \ell\ \Omega\ e^{\nu} \mid otherwise = enter\ (\nu \triangleright \Omega)\ e \\
\quad \textbf{where} \\
\qquad enter\ \Omega\ c \quad\quad = (\Omega,\ c) \\
\qquad enter\ \Omega\ (\lambda x.e) \quad = \textbf{let} \\
\qquad\qquad\qquad\qquad (\Omega',\ e') = prune\ (\ell+1)\ \Omega\ ([\ell/x]e) \\
\qquad\qquad\qquad\qquad \textbf{in} \\
\qquad\qquad\qquad\qquad (\Omega',\ \lambda\ell.e') \\
\qquad enter\ \Omega\ (e_1 \mathbin{@} e_2) = \textbf{let} \\
\qquad\qquad\qquad\qquad (\Omega_1,\ e_1') = prune\ \ell\ \Omega\ e_1 \\
\qquad\qquad\qquad\qquad (\Omega_2,\ e_2') = prune\ \ell\ \Omega_1\ e_2 \\
\qquad\qquad\qquad\qquad \textbf{in} \\
\qquad\qquad\qquad\qquad (\Omega_2,\ e_1' \mathbin{@} e_2')
\end{array}
$$

The first equation of *prune* covers the case of a term's repeated occurrence. In that case, we prune sharing by replacing the term e^{ν} by a tag $\underline{\nu}$ containing its stable name — these are the dotted lines in the second diagram in Figure 2.

To interleave sharing recovery with the conversion from HOAS to typed de Bruijn indices, *prune* tracks the nesting *Level* of lambdas. Moreover, the lambda case of *enter* replaces the HOAS binder x by the level ℓ at the binding and usage sites.

Why don't we separate computing occurrences from tree pruning? When computing occurrences, we must not traverse shared subtrees multiple times, so we can as well prune at the same time. Moreover, in the first line of *prune*, we cannot simply return e instead of $\underline{\nu}$ — e is of the wrong form as it has type T and not ${}^{\circ}T$!

As far as type-preservation is concerned, we do lose information due to replacing variables by levels ℓ. This is the inevitable loss described by Atkey et al. [1], which we make up for by a dynamic check in an environment lookup, as already discussed.

3.3 Float shared subterms

Second, we float all shared subtrees out to where they should be let-bound, represented by (see third diagram in Figure 2)

$$
\begin{array}{lll}
{}^{\uparrow}T^{\tau} \to \overline{\nu : {}^{\uparrow}T^{\tau'}} \succ {}^{\downarrow}T^{\tau} & & \\
{}^{\downarrow}T^{\tau}\ \textbf{where} & & \\
\quad \underline{\nu}^{\tau} & :: {}^{\downarrow}T^{\tau} & \\
\quad C^{\tau} & :: {}^{\downarrow}T^{\tau} & \\
\quad \lambda\nu.{}^{\uparrow}T^{\tau_2} & :: {}^{\downarrow}T^{\tau_1 \to \tau_2} & \\
\quad {}^{\uparrow}T_1^{\tau_1 \to \tau_2} \mathbin{@} {}^{\uparrow}T_2^{\tau_1} & :: {}^{\downarrow}T^{\tau_2} &
\end{array}
$$

A term in ${}^{\uparrow}T$ comprises a sequence of floated-out subterms labelled by their stable name as well as a body term from ${}^{\downarrow}T$ from which the floated subterms where extracted. Moreover, the levels ℓ that replaced lambda binders in ${}^{\circ}T$ get replaced by the stable name of their term node. This simplifies a uniform introduction of de Bruijn indices for let and lambda bound variables.

We write $\nu : {}^{\uparrow}T$ for a possibly empty sequence of items: $\nu_1 : {}^{\uparrow}T_1, \ldots, \nu_n : {}^{\uparrow}T_n$, where $\bullet$ denotes an empty sequence.

The floating function *float* maintains an auxiliary structure of floating terms and levels, defined as follows:

$$\Gamma \to \nu \overset{i}{:} {}^{\uparrow}T^{\tau} \mid \nu \overset{i}{:} \cdot \mid \nu \overset{i}{:} \ell$$

These are floated subtrees named ν of which we have collected i occurrences. The occurrence count indicates where a shared subterm gets let bound: namely at the node where it matches $\Omega\nu$. This is why *prune* needed to collect the number of occurrences in Ω. When the occurrence count matches $\Omega\nu$, we call the floated

term *saturated*. The following function determines saturated floated terms, which ought to be let bound:

$$
\begin{array}{l}
\mathit{bind} :: (\mathit{Name} \mapsto \mathit{Int}) \to \overline{\Gamma} \to \overline{\exists \tau.\nu : {}^{\uparrow}T^{\tau}} \\
\mathit{bind}\ \Omega\ \bullet \qquad\qquad\qquad\qquad\quad = \bullet \\
\mathit{bind}\ \Omega\ (\nu \overset{i}{:} e,\ \overline{\Gamma}) \mid \Omega\nu == i = \nu : e,\ \mathit{bind}\ \Omega\ \overline{\Gamma} \\
\mathit{bind}\ \Omega\ (\nu \overset{i}{:} \_,\ \overline{\Gamma}) \qquad\qquad\quad = \mathit{bind}\ \Omega\ \overline{\Gamma}
\end{array}
$$

Note that Γ does not keep track of the type τ of a floated term ${}^{\uparrow}T^{\tau}$; hence, floated terms from *bind* come in an existential package. This does *not* introduce additional loss of type safety as we already lost the type of lambda bound variables in $\nu \overset{i}{:} \ell$. It merely means that let bound, just like lambda bound, variables require the dynamically checked environment look up we already discussed.

When floating the *first occurrence* of a shared tree (not pruned by *prune*), we use $\nu \overset{i}{:} {}^{\uparrow}T^{\tau}$. When floating *subsequent occurrences* (which were pruned), we use $\nu \overset{i}{:} \cdot$. Finally, when floating a level, to replace it by a stable name, we use $\nu \overset{i}{:} \ell$.

We define a partial ordering on floated terms: $\nu_1 \overset{i}{:} x \ < \ \nu_2 \overset{j}{:} y$ iff the direct path from ν_1 to the root of the AST is shorter than that of ν_2. We keep sequences of floated terms in *descending order* — so that the deepest subterm comes first. We write $\overline{\Gamma_1} \uplus \overline{\Gamma_2}$ to merge two sequences of floated terms. Merging respects the partial order, and it combines floated trees with the same stable name by adding their occurrence counts. To combine the first occurrence and a subsequent occurrence of a shared tree, we preserve the term of the first occurrence. We write $\overline{\Gamma} \setminus \overline{\nu}$ to delete elements of $\overline{\Gamma}$ that are tagged with a name that appears in the sequence $\overline{\nu}$.

We can now formalise the floating process as follows:

$$
\begin{array}{l}
\mathit{float} :: (\mathit{Name} \mapsto \mathit{Int}) \to {}^{\circ}T^{\tau} \to (\overline{\Gamma}, {}^{\uparrow}T^{\tau}) \\
\mathit{float}\ \Omega\ \ell^{\nu} \ = \ (\nu \overset{1}{:} \ell,\ \underline{\nu}) \\
\mathit{float}\ \Omega\ \underline{\nu} \ = \ (\nu \overset{1}{:} \cdot,\ \underline{\nu}) \\
\mathit{float}\ \Omega\ e^{\nu} = \mathbf{let} \\
\qquad\qquad (\overline{\Gamma},\ e') = \mathit{descend}\ e \\
\qquad\qquad \overline{\nu_b\ :\ e_b} = \mathit{bind}\ \Omega\ \overline{\Gamma} \\
\qquad\qquad d \qquad\quad = \overline{\nu_b\ :\ e_b} \succ e' \\
\qquad\quad \mathbf{in} \\
\qquad\quad \mathbf{if}\ \Omega\nu == 1\ \mathbf{then} \\
\qquad\qquad (\overline{\Gamma} \setminus \overline{\nu_b},\ d) \\
\qquad\quad \mathbf{else} \\
\qquad\qquad (\overline{\Gamma} \setminus \overline{\nu_b} \uplus \{\nu : d\},\ \underline{\nu}) \\
\mathbf{where} \\
\qquad \mathit{descend} :: {}^{\circ}T^{\tau} \to (\overline{\Gamma}, {}^{\downarrow}T^{\tau}) \\
\qquad \mathit{descend}\ c \qquad = (\bullet,\ c) \\
\qquad \mathit{descend}\ (\lambda\ell.e) \ = \mathbf{let} \\
\qquad\qquad\qquad\qquad (\overline{\Gamma},\ e') = \mathit{float}\ \Omega\ e \\
\qquad\qquad\qquad\quad \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{if}\ \exists\ \nu'\ i.\ (\nu' \overset{i}{:} \ell) \in \overline{\Gamma}\ \mathbf{then} \\
\qquad\qquad\qquad\qquad (\overline{\Gamma} \setminus \{\nu'\},\ \lambda\nu'.e') \\
\qquad\qquad\qquad\quad \mathbf{else} \\
\qquad\qquad\qquad\qquad (\overline{\Gamma},\ \lambda\_.e') \\
\qquad \mathit{descend}\ (e_1\ @\ e_2) = \mathbf{let} \\
\qquad\qquad\qquad\qquad (\overline{\Gamma}_1,\ e_1') = \mathit{float}\ \Omega\ e_1 \\
\qquad\qquad\qquad\qquad (\overline{\Gamma}_2,\ e_2') = \mathit{float}\ \Omega\ e_2 \\
\qquad\qquad\qquad\quad \mathbf{in} \\
\qquad\qquad\qquad\quad (\overline{\Gamma}_1 \uplus \overline{\Gamma}_2,\ e_1'\ @\ e_2')
\end{array}
$$

The first two cases of *float* ensure that the levels of lambda bound variables and the names of pruned shared subterms are floated regardless of how often they occur. In contrast, the third equation floats a term with name ν only if it is shared; i.e., $\Omega\nu$ is not 1. If it is shared, it is also pruned; i.e., replaced by its name $\underline{\nu}$ — just as in the third diagram of Figure 2.

Regardless of whether a term gets floated, all saturated floated terms, $\overline{\nu_b\ :\ e_b}$, must prefix the result, e', and be removed from $\overline{\Gamma}$.

When *descend*ing into a term, the only interesting case is for lambdas. For a lambda at level ℓ, we look for a floated level of the form $\nu' : \ell$. If that is available, ν' replaces ℓ as a binder and we remove $\nu' : \ell$ from $\overline{\Gamma}$. However, if $\nu' : \ell$ is not in $\overline{\Gamma}$, the binder introduced by the lambda doesn't get used in e. In this case, we pick an arbitrary new name; here symbolised by an underscore "_".

3.4 Binder introduction

Thirdly, we introduce typed de Bruijn indices to represent lambda and let binding structure (rightmost diagram in Figure 2):

$$
\begin{array}{l}
{}^{env}T^{\tau}\ \mathbf{where} \\
\quad C^{\tau} \qquad\qquad\qquad\qquad :: {}^{env}T^{\tau} \\
\quad {}^{env_\iota}\iota^{\tau} \qquad\qquad\qquad :: {}^{env}T^{\tau} \\
\quad \lambda^{(\tau_1,\ env)}T^{\tau_2} \qquad\quad :: {}^{env}T^{\tau_1 \to \tau_2} \\
\quad {}^{env}T_1^{\tau_1 \to \tau_2}\ @\ {}^{env}T_2^{\tau_1} \ :: {}^{env}T^{\tau_2} \\
\quad \mathtt{let}\ {}^{env}T_1^{\tau_1}\ \mathtt{in}\ {}^{(\tau_1,\ env)}T_2^{\tau_2} :: {}^{env}T^{\tau_2}
\end{array}
$$

With this type of terms, $e :: {}^{env}T^{\tau}$ means that e is a term representing a computation producing a value of type τ under the type environment env. Type environments are nested pair types, possibly terminated by a unit type $()$. For example, $(((), \tau_1), \tau_0)$ is a type environment, where de Bruijn index 0 represents a variable of type τ_0 and de Bruijn index 1 represents a variable of type τ_1.

We abbreviate $\mathtt{let}\ e_1\ \mathtt{in}\ \cdots\ \mathtt{let}\ e_n\ \mathtt{in}\ e_b$ as $\mathtt{let}\ \overline{e}\ \mathtt{in}\ e_b$. Both λ and $\mathtt{let}$ use de Bruijn indices ι instead of introducing explicit binders.

To replace the names of pruned subtrees and of lambda bound variables by de Bruijn indices, we need to construct a suitable type environment as well as an association of environment entries, their de Bruijn indices, and the stable names that they replace. We maintain the type environment with associated de Bruijn indices in the following *environment layout* structure:

$$
\begin{array}{l}
{}^{env}\Delta^{env'}\ \mathbf{where} \\
\quad \circ \qquad\qquad\qquad\qquad :: {}^{env}\Delta^{()} \\
\quad {}^{env}\Delta^{env'}; {}^{env_\iota}\iota^{\tau} :: {}^{env}\Delta^{(env',\ t)}
\end{array}
$$

Together with a layout, we use a sequence of names $\overline{\nu}$ of the same size as the layout, where corresponding entries represent the same variable. As this association between typed layout and untyped sequence of names is not validated by types, the lookup function $\mathit{lyt} \downarrow i$ getting the ith index of layout lyt makes use of a dynamic type check. It's signature is $(\downarrow) :: \mathbb{N} \to {}^{env}\Delta^{env'} \to {}^{env_\iota}\iota^{\tau}$.

Now we can introduces de Bruijn indices to body expressions:

$$
\begin{array}{l}
\mathit{body} :: {}^{env}\Delta^{env} \to \overline{\nu} \to {}^{\downarrow}T^{\tau} \to {}^{env}T^{\tau} \\
\mathit{body}\ \mathit{lyt}\ (\nu_{\rho,0}, \ldots, \nu_{\rho,n})\ \underline{\nu} \\
\quad \mid \nu == \nu_{\rho,i} \qquad = \mathit{lyt} \downarrow i \\
\mathit{body}\ \mathit{lyt}\ \overline{\nu_\rho}\ c \qquad = c \\
\mathit{body}\ \mathit{lyt}\ \overline{\nu_\rho}\ (\lambda\nu.e) \ = \lambda(\mathit{binders}\ \mathit{lyt}^{+}\ (\nu, \overline{\nu_\rho})\ e) \\
\mathit{body}\ \mathit{lyt}\ \overline{\nu_\rho}\ (e_1\ @\ e_2) = (\mathit{binders}\ \mathit{lyt}\ \overline{\nu_\rho}\ e_1)\ @\ (\mathit{binders}\ \mathit{lyt}\ \overline{\nu_\rho}\ e_2)
\end{array}
$$

The first equation performs a lookup in the environment layout at the same index where the stable name ν occurs in the name environment $\overline{\nu}$. The lookup is the same for lambda and let bound variables. It is the only place where we need a dynamic type check and that is already needed for lambda bound variables alone.

In the case of a lambda, we add a new binder by extending the layout, denoted lyt^{+}, with a new zeroth de Bruijn index and shifting all others one up. Keeping the name environment in sync, we add the stable name ν, which ${}^{\downarrow}T$ used as a binder.

In the same vein, we bind n floated terms $\overline{\nu : e}$ with $\mathtt{let}$ bindings in body expression e_b, by extending the type environment n times (map applies a function to each element of a sequence):

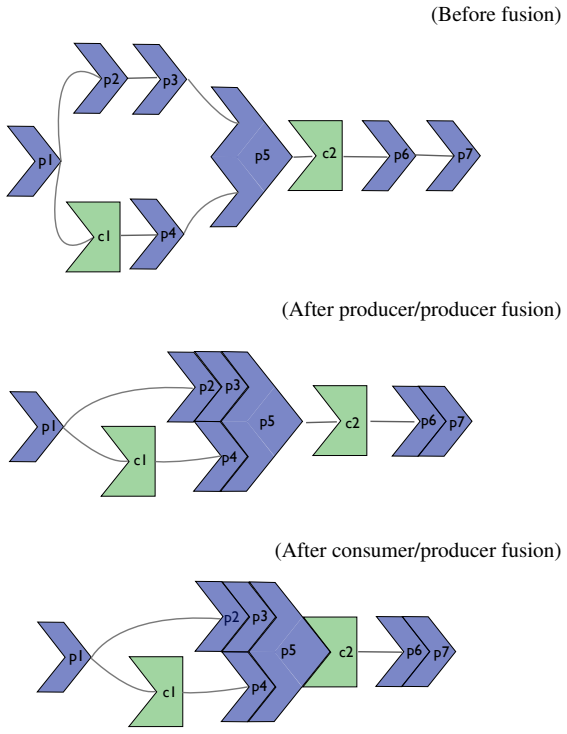


Figure 3. Produce/producer and consumer/producer fusion

$$
\begin{array}{l}
\mathit{binders} :: {}^{env}\Delta^{env} \to \overline{\nu} \to {}^{\uparrow}T^{\tau} \to {}^{env}T^{\tau} \\
\mathit{binders}\ \mathit{lyt}\ \overline{\nu_\rho}\ (\overline{\nu : e} \succ e_b) = \\
\quad \texttt{let}\ \mathit{map}\ (\mathit{binders}\ \mathit{lyt}\ \overline{\nu_\rho})\ \overline{e}\ \texttt{in}\ \mathit{body}\ \mathit{lyt}^{+n}\ (\overline{\nu}, \overline{\nu_\rho})\ e_b \\
\textbf{where}\ n = \mathit{length}\ (\overline{\nu : e})
\end{array}
$$

We tie the three stages together to convert from HOAS with sharing recovery producing let bindings and typed de Bruijn indices:

$$
\begin{array}{l}
\mathit{hoasSharing} :: T^{\tau} \to {}^{()}T^{\tau} \\
\mathit{hoasSharing}\ e = \textbf{let} \\
\qquad\qquad (\Omega,\ e') = \mathit{prune}\ 0 \bullet e \\
\qquad\qquad (\bullet,\ e'') = \mathit{float}\ \Omega\ e' \\
\qquad\quad \textbf{in} \\
\qquad\quad \mathit{binders} \circ \bullet\ e''
\end{array}
$$

4. Array fusion

Fusion in a massively data-parallel, embedded language for GPUs, such as Accelerate, requires a few uncommon considerations.

Parallelism. While fusing parallel collective operations, we must be careful not to lose information essential to parallel execution. For example, `foldr/build` fusion [15] is not applicable, because it produces sequential tail-recursive loops rather than massively parallel GPU kernels. Similarly, the `split/join` approach used in Data Parallel Haskell (DPH) [16] is not helpful, although fused operations are split into sequential and parallel subcomputations, as they assume an explicit parallel scheduler, which in DPH is written directly in Haskell. Accelerate compiles massively parallel array combinators to CUDA code via template skeleton instantiation, so any fusion system must preserve the combinator representation of the intermediate code.

Sharing. Existing fusion transforms rely on inlining to move producer and consumer expressions next to each other, which allows producer/consumer pairs to be detected. However, when let-bound variables are used multiple times in the body of an expression, unrestrained inlining can lead to duplication of work. Compilers such as GHC, handle this situation by only inlining the definitions of let-bound variables that have a single use site, or by relying on some heuristic about the size of the resulting code to decide what to inline [26]. However, in typical Accelerate programs, each array is used at least twice: once to access the shape information and once to access the array data; so, we must handle at least this case separately.

Filtering. General array fusion transforms must deal with filter-like operations, for which the size of the result structure depends on the *value* of the input structure, as well as its size. Accelerate does not encode filtering as a primitive operation, so we do not need to consider it further.[1]

Fusion at run-time. As the Accelerate language is embedded in Haskell, compilation of the Accelerate program happens at Haskell *runtime* rather than when compiling the Haskell program. For this reason, optimisations applied to an Accelerate program contribute to its overall runtime, so we must be mindful of the cost of analysis and code transformation. On the flip-side, runtime optimisations can make use of information that is only available at runtime.

Fusion on typed de Brujin terms. We fuse Accelerate programs by rewriting typed de Bruijn terms in a type preserving manner. However, maintaining type information adds complexity to the definitions and rules, which amounts to a partial proof of correctness checked by the type checker, but is not particularly exciting for the present exposition. Hence, in this section, we elide the steps necessary to maintain type information during fusion.

4.1 The Main Idea

All collective operations in Accelerate are array-to-array transformations. Reductions, such as `fold`, which reduce an array to a single element, yield a singleton array rather than a scalar expression. Hence, we can partition array operations into two categories:

1. Operations where each element of the result array depends on at most one element of each input array. Multiple elements of the output array may depend on a single input array element, but all output elements can be computed independently. We refer to these operations as *producers*.
2. Operations where each element of the result array depends on multiple elements of the input array. We call these functions *consumers*, in spite of the fact that they also produce an array.

Table 1 summarises the collective array operations that we support. In a parallel context, producers are more pleasant to deal with because independent element-wise operations have an obvious mapping to the GPU. Consumers are a different story, as we need to know exactly how the computations depend on each other to implement them efficiently. For example, a parallel fold (with an associative operator) can be implemented efficiently as a tree reduction, but a parallel scan requires two separate phases [9, 31]. Unfortunately, this sort of information is obfuscated by most fusion techniques. To support the different properties of producers and consumers, our fusion transform is split into two distinct phases:

- *Producer/producer:* fuse sequences of producers into a single producer. This is implemented as a source-to-source transformation on the AST.
- *Consumer/producer:* fuse producers followed by a consumer into the consumer. This happens during code generation, where we specialise the consumer skeleton with the producer code.

[1] `filter` is easily implemented as a combination of the core primitives, and is provided as part of the library.

Producers

`map`	`:: (Exp a -> Exp b) -> Acc (Array sh a) -> Acc (Array sh b)`	map a function over an array
`zipWith`	`:: (Exp a -> Exp b -> Exp c) -> Acc (Array sh a) -> Acc (Array sh b)`	apply funciton to...
	`-> Acc (Array sh c)`	... a pair of arrays
`backpermute`	`:: Exp sh' -> (Exp sh' -> Exp sh) -> Acc (Array sh a)`	backwards permutation
	`-> Acc (Array sh' e)`	
`replicate`	`:: Slice slix => Exp slix`	extend array across...
	`-> Acc (Array (SliceShape slix) e)`	... new dimensions
	`-> Acc (Array (FullShape  slix) e)`	
`slice`	`:: Slice slix`	remove existing dimensions
	`=> Acc (Array (FullShape  slix) e) -> Exp slix`	
	`-> Acc (Array (SliceShape slix) e)`	
`generate`	`:: Exp sh -> (Exp sh -> Exp a) -> Acc (Array sh a)`	array from index mapping

Consumers

`fold`	`:: (Exp a -> Exp a -> Exp a) -> Exp a -> Acc (Array (sh:.Int) a)`	tree reduction along...
	`-> Acc (Array sh a)`	... innermost dimension
`scan{l,r}`	`:: (Exp a -> Exp a -> Exp a) -> Exp a -> Acc (Vector a)`	left-to-right or right-to-left
	`-> Acc (Vector a)`	... vector pre-scan
`permute`	`:: (Exp a -> Exp a -> Exp a) -> Acc (Array sh' a)`	forward permutation
	`-> (Exp sh -> Exp sh') -> Acc (Array sh a) -> Acc (Array sh' a)`	
`stencil`	`:: Stencil sh a stencil => (stencil -> Exp b) -> Boundary a`	map a function with local...
	`-> Acc (Array sh a) -> Acc (Array sh b)`	... neighbourhood context

Table 1. Summary of Accelerate's core collective array operations, omitting `Shape` and `Elt` class constraints for brevity. In addition, there are other flavours of folds and scans as well as segmented versions of these.

Separating fusion into these two phases reduces the complexity of the task, though there is also a drawback: as all collective operations in Accelerate output arrays, we might wish to use the output of a consumer as an input to a producer as in `map g . fold f z`. Here, the `map` operation could be fused into the `fold` by applying the function `g` to each element produced by the reduction before storing the final result in memory. This is useful, as Accelerate works on multidimensional arrays, so the result of a `fold` can be a large array rather than just a singleton array. Our approach currently does not fuse producer/consumer pairs, only consumer/producer and producer/producer combinations.

Figure 3 illustrates how fusion affects the AST: blue boxes p_1 to p_7 represent producers, where p_5 is a producer like `zipWith` with two input arrays. The consumers are c_1 and c_2. Firstly, we fuse all producers, with the exception of p_1 whose result is used by both c_1 and p_2. Next, we plug the fused producers into consumers where possible. Again, p_1 is left as is. It would be straightforward to change our implementation such that it would fuse p_1 into both p_2 and c_1. This would duplicate the work of p_1 into both p_2 and c_1, which, despite reducing memory traffic, is not always advantageous. Our current implementation is conservative and never duplicates work; we plan to change this in future work as the restricted nature of Accelerate means that we can compute accurate cost estimates and make an informed decision. In contrast, producer/consumer fusion of c_1 into p_4 would require fundamental changes.

4.2 Producer/producer fusion for parallel arrays

The basic idea behind the representation of producer arrays in Accelerate is well known: simply represent an array by its shape and a function mapping indices to their corresponding values. We previously used it successfully to optimise purely functional array programs in Repa [17], but it was also used by others [11].

However, there are at least two reasons why it is not always beneficial to represent all array terms uniformly as functions. One is *sharing*: we must be able to represent some terms as manifest arrays so that a delayed-by-default representation can not lead to arbitrary loss of sharing. This is a well known problem in Repa. The other consideration is *efficiency*: since we are targeting an architecture designed for performance, we prefer more specific operations. An opaque indexing function is too general, conveying no information about the pattern in which the underlying array is accessed, and hence no opportunities for optimisation. We shall return to this point in Section 5, but already include a form of structured traversal over an array (`Step`) in the following definition:

```
data DelayedAcc a where
  Done  :: Acc a
        -> DelayedAcc a

  Yield :: (Shape sh, Elt e)
        => Exp sh
        -> Fun (sh -> e)
        -> DelayedAcc (Array sh e)

  Step  :: (Shape sh, Shape sh', Elt e, Elt e')
        => Exp sh'
        -> Fun (sh' -> sh)
        -> Fun (e -> e')
        -> Idx (Array sh e)
        -> DelayedAcc (Array sh' e')
```

We have three constructors: `Done` injects a manifest array into the type. `Yield` defines a delayed array in terms of its shape and a function which maps indices to elements. The third constructor, `Step`, encodes a special case of the more general `Yield` that represents the application of an index and/or value space transformation to the argument array. The type `Fun (sh -> e)` is that of a term representing a scalar function from shape to element type. The type `Idx (Array sh e)` is that of a de Bruijn index representing an array valued variable. Representing the argument array in this way means that both `Step` and `Yield` are non-recursive in `Acc` terms, and so they can always be expressed as scalar functions and embedded into consumers in the second phase of fusion.

We represent all array functions as constructors of the type `DelayedAcc`. Producer/producer fusion is achieved by tree con-

traction on the AST, merging sequences of producers into a single one. All array producing functions, such as `map` and `backpermute`, are expressed in terms of smart constructors for the `DelayedAcc` type. The smart constructors manage the integration with successive producers, as shown in the following definition of `mapD`, the delayed version of the `map` function:

```
mapD :: (Shape sh, Elt a, Elt b)
     => Fun (a -> b)
     -> DelayedAcc (Array sh a)
     -> DelayedAcc (Array sh b)
mapD f (Step  sh p g v) = Step  sh p (f . g) v
mapD f (Yield sh g)     = Yield sh   (f . g)
```

The function composition operator `(.)` is overloaded here to work on scalar function terms. With this definition we now have the well known fusion rule that reduces `mapD f . mapD g` sequences to `mapD (f . g)`. Similarly, the definition of delayed backpermute means that `backpermuteD sh p (backpermuteD _ q arr)` reduces to `backpermute sh (q . p) arr`:

```
backpermuteD
    :: (Shape sh, Shape sh', Elt e)
    => Exp        sh'
    -> Fun        (sh' -> sh)
    -> DelayedAcc (Array sh  e)
    -> DelayedAcc (Array sh' e)
backpermuteD sh' p acc = case acc of
  Step  _ ix f a  -> Step  sh' (ix . p) f a
  Yield _ f       -> Yield env sh' (f . p)
  Done  env a     -> Step  sh' p identity (toIdx a)
```

Of course, combinations of maps with backpermutes also reduce to a single producer.

As desired, this approach also works on producers which take their input from multiple arrays. This is in contrast to `foldr/build` [15], which can fuse one of the input arguments, but not both. The definition of `zipWithD` considers all possible combinations of constructors (only some of which we list here) and can therefore fuse producers of both arguments:

```
zipWithD :: (Shape sh, Elt a, Elt b, Elt c)
         => Fun (a -> b -> c)
         -> DelayedAcc (Array sh a)
         -> DelayedAcc (Array sh b)
         -> DelayedAcc (Array sh c)
zipWithD f (Yield sh1 g1) (Yield sh2 g2)
  = Yield (sh1 `intersect` sh2)
          (\sh -> f (g1 sh) (g2 sh))
zipWithD f (Yield sh1 g1) (Step  sh2 ix2 g2 a2)
  = Yield ...
```

In this manner, sequences of producers fuse into a single producer term; then, we turn them back into a manifest array using the function `compute`. It inspects the argument terms of the delayed array to identify special cases, such as maps or backpermutes, as shown in the following snippet of pseudo-code:

```
compute :: DelayedAcc a -> Acc a
compute (Done a)          = a
compute (Yield sh f)      = Generate sh f
compute (Step sh p f v)
  | sh == shape a, isId p, isId f
                          = a
  | sh == shape a, isId p = Map f a
  | isId f                = Backpermute sh p a
  | otherwise             = Transform sh p f a
  where a = Avar v
```

Since we operate directly on the AST of the program, we can inspect function arguments and specialise the code accordingly. For example, `isId :: Fun (a->b) -> Bool` checks whether a function term corresponds to the term $\lambda x.x$.

4.3 Consumer/Producer Fusion

Now that we have the story for producer/producer fusion, we discuss how to deal with consumers. We pass producers encoded in the `DelayedAcc` representation as arguments to consumers, so that the consumers can compute the elements they need on-the-fly. Consumers themselves have no `DelayedAcc` representation, however.

Consumers such as `stencil`, access elements of their argument array multiple times. These consumers are implemented carefully not to duplicate work. Indeed, even when the argument of such a consumer is a manifest array, the consumer should ensure that it caches already fetched elements, as GPUs impose a high performance penalty for repeated memory loads. Some consumers can be implemented more efficiently when given a producer expressed in terms of a function from a multi-dimensional array index to an element value. Other consumers prefer functions that map the flat linear index of the underling array to its value. Our consumer-friendly representation of delayed arrays therefore contains both versions:

```
data Embedded sh e
  =  (Shape sh, Elt e)
  => DelayedArray { extent      :: Exp sh
                  , index       :: Fun (sh  -> e)
                  , linearIndex :: Fun (Int -> e) }

embedAcc :: (Shape sh, Elt e)
         => Acc (Array sh e) -> Embedded sh e
embedAcc (Generate sh f)
  = DelayedArray sh f (f . (fromIndex sh))
embedAcc (Map f (AVar v))
  = DelayedArray (shape v) (indexArray v)
                           (linearIndexArray v)
embedAcc ...
```

The function `embedAcc` intelligently injects each producer into the `Embedded` type by inspection of the constructor, as shown in the code snippet above. In theory, `compute` and `embedAcc` could be combined to go directly from the delayed representation which is convenient for producer/producer fusion to the one for consumer/producer fusion. However, these two steps happen at different phases in the compilation, so we want to limit the visibility of each delayed representation to the current phase.

Producers are finally embedded into consumers during code generation. During code generation, the code for the embedded producers is plugged into the consumer template. The `codegenAcc` function inspects the consumer and generates code for the arguments of the consumer. It then passes these CUDA code snippets to a function specialised on generating code for this particular consumer (`mkFold` in the example below), which combines these snippets with the actual CUDA consumer code:

```
codegenAcc :: DeviceProperties -> OpenAcc arrs
           -> Gamma aenv -> CUTranslSkel arrs
codegenAcc dev (OpenAcc (Fold f z a)) aenv
  = mkFold dev  aenv (codeGenFun f) (codeGenExp z)
           (codegenDelayedAcc $ embedAcc a)
codegenAcc dev (OpenAcc (Scanl f z a)) aenv
  = ...
```

As a result, the code producing each element is integrated directly into the consumer, and no intermediate array needs to be created.

4.4 Exploiting all opportunities for fusion

As mentioned previously, we need to be careful about fusing shared array computations, to avoid duplicating work. However, *scalar* Accelerate computations that manipulate *array shapes*, as opposed to the bulk array data, can lead to terms that employ sharing, but can never duplicate work. Such terms are common in Accelerate code, and it is important to that they do not inhibit fusion.

Consider the following example that first reverses a vector with `backpermute` and then maps a function `f` over the result. Being a sequence of two producer operations, we would hope these are fused into a single operation:

```
reverseMap f a
  = map f
  $ backpermute (shape a) (\i->length a-i-1) a
```

Unfortunately, sharing recovery, using the algorithm from Section 3, causes a problem. The variable `a` is used three times in the arguments to `backpermute`; hence, sharing recovery will introduce a let binding at the lowest common meet point for all uses of the array. This places it between the `map` and `backpermute` functions:

```
reverseMap f a
  = map f
  $ let v = a
    in backpermute (shape v) (\i->length v-i-1) v
```

This binding, although trivial, prevents fusion of the two producers, and it does so unnecessarily. The argument array is used three times: twice to access shape information, but only once to access the array data — in the final argument to `backpermute`.

Fortunately, there is a simple work around. Recall that our delayed array constructors `Step` and `Yield` carry the shape of the arrays they represent. Hence, we can eliminate all uses of the array that *only access shape information*, leaving us with a single reference to the array's payload. That single reference enables us to remove the let binding and to re-enable fusion.

Similarly, we may float let bindings of manifest data out (across producer chains). This helps to expose further opportunities for producer/producer fusion. For example, we allow the binding of `xs` to float above the `map` so the two producers can be fused:

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

While floating let bindings opens up the potential for further optimisations, we are careful to not increase the lifetime of bound variables, as this would increase live memory usage.

5. Benchmarks

Benchmarks were conducted on a single Tesla T10 processor (compute capability 1.3, 30 multiprocessors = 240 cores at 1.3GHz, 4GB RAM) backed by two quad-core Xenon E5405 CPUs (64-bit, 2GHz, 8GB RAM) running GNU/Linux (Ubuntu 12.04 LTS). The reported GPU runtimes are averages of 100 runs.

5.1 Execution Overheads

Runtime program optimisation, code generation, kernel loading, data transfer, and so on can contribute significant overheads to short lived GPU computations. Accelerate mitigates these overheads via caching and memoisation. For example, the first time a particular expression is executed it is compiled to CUDA code, which is reused for subsequent invocations. In the remainder of this section we factor out the cost of runtime code generation, and the associated caching, by reporting just the runtimes of the GPU kernels — for our system as well as the others we compare against.

5.2 Dot product

Dot product uses the code from Section 2.1. Without fusion the intermediate array produced by `zipWith` is created in GPU memory before being read back by `fold`. This is a simple memory-bound benchmark; hence, fusion roughly doubles the performance.

The `Data.Vector` baseline is sequential code produced via stream fusion [12], running on the host CPU. The `Repa` version runs in parallel on all eight cores of the host CPU, using the fusion method reported in [17]. The `NDP2GPU` version [4] compiles NESL code [6] to CUDA. The performance of this version suffers because the `NDP2GPU` compiler uses the legacy NESL compiler for the front-end, which introduces redundant administrative operations that are not strictly needed when evaluating a dot product.

The Accelerate version is still slightly slower than CUBLAS. The fused code embeds a function of type `(sh -> a)` into the reduction. The extra arithmetic for converting the multidimentional `sh` index to a linear index is significant for this simple benchmark.

5.3 Black-Scholes options pricing

Black-Scholes uses the code from Figure 1. The source contains several let-bound variables that are used multiple times, and without sharing recovery the corresponding values are recomputed for each occurrence. The Accelerate version is faster than the reference CUDA version because the latter contains a common subexpression that is not eliminated by the CUDA compiler. The CUDA version is part of the official NVIDIA CUDA distribution. The common subexpression performs a single multiplication.

5.4 N-Body gravitational simulation

The n-body example simulates Newtonian gravitational forces on a set of massive bodies in 3D space, using the naive O(n^2) algorithm. In a data-parallel setting, the natural implementation first computes the forces between every pair of bodies, before reducing the components applied to each body using a segmented sum. Without fusion this approach requires O(n^2) space for the intermediate array of forces, which exhausts the memory of our device for more than about 5k bodies. With fusion, the reduction consumes each force value on-the-fly, so that the program only needs O(n) space to store the final force values.

Even with fusion the hand-written CUDA version is over $10\times$ faster, as it uses on-chip *shared memory* to reduce the memory bandwidth requirements of the program. The shared memory is essentially a software managed cache, and making automatic use of it remains an open research problem [21].

5.5 Mandelbrot fractal

The Mandelbrot set is generated by sampling values c in the complex plane, and determining whether under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ that $|z_n|$ remains bounded however large n gets. As recursion in Accelerate always proceeds through the host language, we define each step of the iteration as a collective operation and unfold the loop a fixed number of times.

Table 2 shows that without fusion performance is poor because storing each step of the iteration saturates the memory bus. The CUDA version is about 70% faster because it includes a custom software thread block scheduler to manage the unbalanced workload inherent to this benchmark.

5.6 Fluid Flow

Fluid flow implements Jos Stam's stable fluid algorithm [32], a fast approximate algorithm intended for animation and games, rather than accurate engineering simulation. The core of the algorithm is a finite time step simulation on a grid, implemented as a matrix relaxation involving the discrete Laplace operator (∇^2). Relaxation

Benchmark	Input Size	Contender (ms)		Accelerate full (ms)		Accelerate no fusion (ms)		Accelerate no sharing (ms)	
Black Scholes	20M	6.70	(CUDA)	6.19	(92%)	(not needed)		116	(1731%)
Canny	16M	50.6	(OpenCV)	78.4	(155%)	(not needed)		82.7	(164%)
Dot Product	20M	1.88	(CUBLAS)	2.35	(125%)	3.90	(207%)	(not needed)	
Fluid Flow	2M	5461	(Repa -N7)	107	(1.96%)	(not needed)		119	(2.18%)
Mandelbrot (limit)	2M	14.0	(CUDA)	24.0	(171%)	245	(1750%)	245	(1750%)
N-Body	32k	54.4	(CUDA)	607	(1116%)	(out of memory)		(out of memory)	
Radix sort	4M	780	(Nikola)	442	(56%)	657	(84%)	657	(84%)
SMVM (protein)	4M	0.641	(CUSP)	0.637	(99%)	32.8	(5115%)	(not needed)	

Table 2. Benchmark Summary

is performed by stencil convolution using the standard four-point Laplace kernel. The program is very memory intensive, performing approximately 160 convolutions of the grid per time step. The Repa version running on the host CPUs is described in [20]; it suffers from a lack of memory bandwidth compared with the GPU version.

5.7 Canny edge detection

Edge detection applies the Canny algorithm [7] to square images of various sizes. The algorithm consists of seven phases, the first six are naturally data parallel and performed on the GPU. The last phase uses a recursive algorithm to "connect" pixels that make up the output lines. In our implementation this phase is performed sequentially on a host CPU, which accounts for the non-linear slowdown visible with smaller images. We also show the runtime for just the first six data parallel phases.

The data parallel phases are slightly slower than the baseline OpenCV version. This is as our implementation of stencil convolution in Accelerate checks whether each access to the source array is out of bounds, and must have boundary conditions applied. To address this shortcoming we intend to separate computation of the border region which requires boundary checks, from the main internal region which does not [19], but we leave this to future work.

5.8 Radix sort

Radix sort implements the algorithm described in Blelloch [5] to sort an array of signed 32-bit integers. We compare our implementation against a Nikola [22] version.[2]

The Accelerate version is faster then Nikola because Nikola is limited to single kernel programs and must transfer intermediate results back to the host. Hand written CUDA implementations such as in the Thrust [29] library make use of on-chip shared memory and are approximately $10\times$ faster. As mentioned earlier, automatically making use of GPU shared memory remains an open research problem [21].

5.9 Sparse-matrix vector multiplication (SMVM)

SMVM multiplies a sparse matrix in compressed row format (CSR) [9] with a dense vector. Table 3 compares Accelerate to the CUSP library [3], which is a special purpose library for sparse matrix operations. For test data we use a 14 matrix corpus derived from a variety of application domains [33].

Compared to our previous work [8] the fusion transformation converts the program to a single segmented reduction. The corresponding reduction in memory bandwidth puts Accelerate on par with CUSP for several test matrices. In a balanced machine SMVM should be limited by memory throughput, and a dense matrix in sparse format should provide an upper bound on performance.

[2] We repeat figures of [22] as Nikola no longer compiles with recent GHC versions. The figures from [22] were obtained using the same GPU as ours.

Name	Non-zeros (nnz/row)	CUSP	Accelerate	Accelerate no fusion
Dense	4M (2K)	14.48	14.62	3.41
Protein	4.3M (119)	13.55	13.65	0.26
FEM/Spheres	6M (72)	12.63	9.03	4.70
FEM/Cantilever	4M (65)	11.98	7.96	4.41
Wind Tunnel	11.6M (53)	11.98	7.33	4.62
FEM/Harbour	2.37M (50)	9.42	6.14	0.13
QCD	1.9M (39)	7.79	4.66	0.13
FEM/Ship	3.98 (28)	12.28	6.60	4.47
Economics	1.27M (6)	4.59	0.90	1.06
Epidemiology	1.27M (4)	6.42	0.59	0.91
FEM/Accelerator	2.62M (22)	5.41	3.08	2.92
Circuit	959k (6)	3.56	0.82	1.08
Webbase	3.1M (3)	2.11	0.47	0.74
LP	11.3M (2825)	5.22	5.04	2.41

Table 3. Overview of sparse matrices tested and results of the benchmark. Measurements are in GFLOPS/s (higher is better).

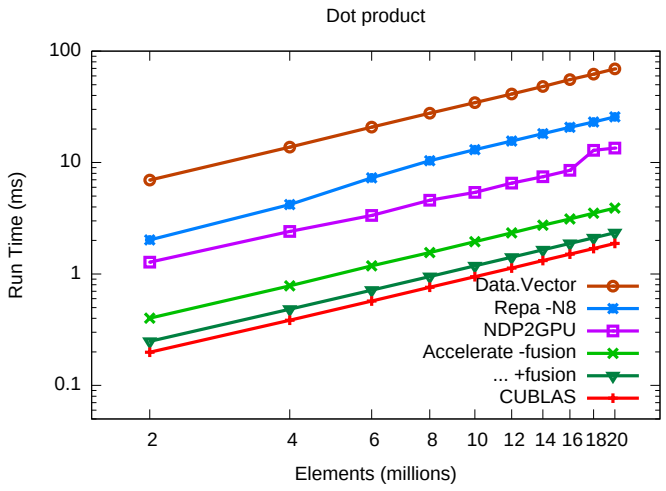
However, with matrices such as FEM/Spheres which contain only a few non-zeros per row ($\lesssim 2 \times \text{warp size} = 64$), Accelerate is slightly slower than CUSP. We conjecture that this is due to the way our skeleton code vector read of each matrix row is coalesced and aligned to the warp boundary to maximise global memory throughput, but is then not able to amortise this extra startup cost over the row length.

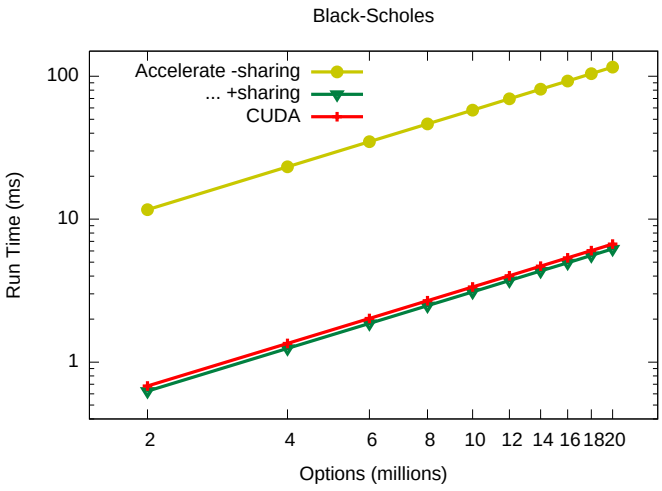
Matrices with large vectors and few non-zeros per row (e.g., Epidemiology), exhibit low flop/byte ratio and poorly suit the CSR format, with all implementations performing well below peak.

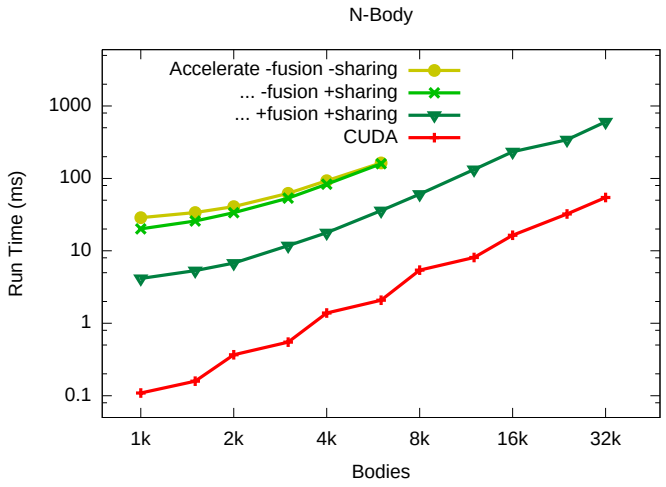
6. Related work

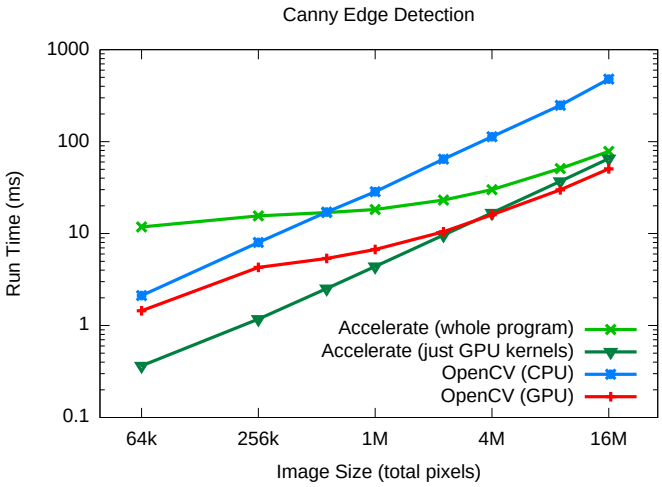
Repa [17] is a Haskell library for parallel array programming on shared-memory SMP machines. Repa uses the delayed/manifest representation split on which our `DelayedAcc` type is based, though the idea of representing arrays as functions is folklore. With Repa the conversion between array representations is done manually and can cause shared expressions to be recomputed rather than stored. Such recomputation can improve runtime performance depending on the algorithm. In Accelerate the conversion is automatic and conservative, so that shared expressions are never recomputed.

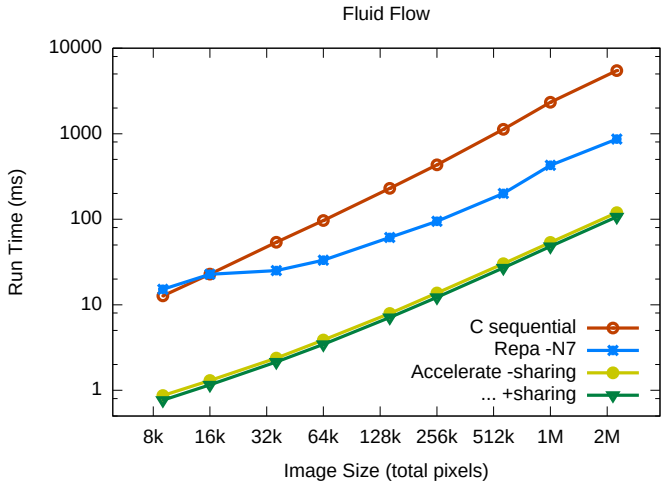
Vertigo [13], Nikola [22] and Obsidian [10] are EDSLs in Haskell and were mentioned in Section 1. Vertigo is a first-order language for writing shaders, and does not provide higher-order combinators such as `map` and `fold`. Nikola uses an instance of Gill's approach [14] to sharing recovery, is limited to single GPU kernel programs, and performs no fusion.

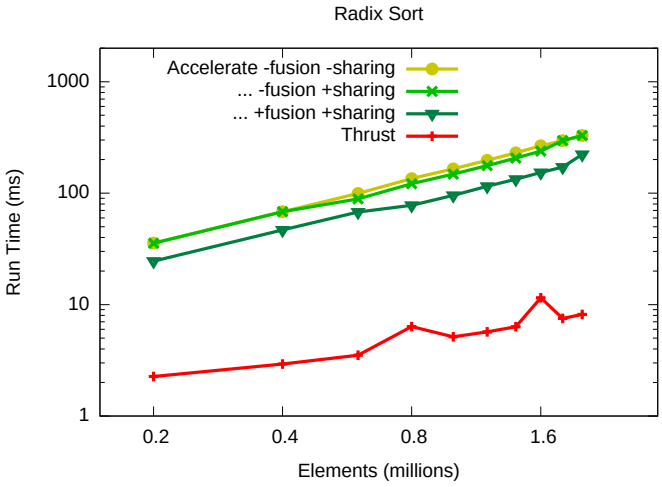

Dot product
Run Time (ms)
100
10
1
0.1
2
4
6
8
10
12
14
16
18
20
Elements (millions)
Data.Vector
Repa -N8
NDP2GPU
Accelerate -fusion
... +fusion
CUBLAS


Black-Scholes
Run Time (ms)
100
10
1
2
4
6
8
10
12
14
16
18
20
Options (millions)
Accelerate -sharing
... +sharing
CUDA


N-Body
Run Time (ms)
1000
100
10
1
0.1
1k
2k
4k
8k
16k
32k
Bodies
Accelerate -fusion -sharing
... -fusion +sharing
... +fusion +sharing
CUDA


Canny Edge Detection
Run Time (ms)
1000
100
10
1
0.1
64k
256k
1M
4M
16M
Image Size (total pixels)
Accelerate (whole program)
Accelerate (just GPU kernels)
OpenCV (CPU)
OpenCV (GPU)


Fluid Flow
Run Time (ms)
10000
1000
100
10
1
8k
16k
32k
64k
128k
256k
512k
1M
2M
Image Size (total pixels)
C sequential
Repa -N7
Accelerate -sharing
... +sharing


Radix Sort
Run Time (ms)
1000
100
10
1
0.2
0.4
0.8
1.6
Elements (millions)
Accelerate -fusion -sharing
... -fusion +sharing
... +fusion +sharing
Thrust

Obsidian [10] is a lower level language where more details of the GPU hardware are exposed to the programmer. Recent versions of Obsidian [11] implement Repa-style delayed *pull arrays* as well as *push arrays*. Whereas a pull array represents a general producer, a push array represents a general consumer. Push arrays allow the intermediate program to be written in continuation passing style (CPS), and helps to compile (and fuse) append-like operations.

Baracuda [18] is another Haskell EDSL that produces CUDA GPU kernels, though is intended to be used offline, with the kernels being called directly from C++. The paper [18] mentions a fusion system that appears to be based on pull arrays, though the mechanism is not discussed in detail. Barracuda steps around the sharing problem by requiring let-bindings to be written using the AST node constructor, rather than using Haskell's native let-expressions.

Delite/LMS [28] is a parallelisation framework for DSLs in Scala that uses library-based multi-pass staging to specify complex optimisations in a modular manner. Delite supports loop fusion for DSLs targeting GPUs using rewrite rules on a graph-based IR.

NDP2GPU [4] compiles NESL code down to CUDA. As the source language is not embedded there is no need for sharing recovery. NDP2GPU performs `map`/`map` fusion but cannot fuse `map`s into reduction combinators.

Sato and Iwasaki [30] describe a C++ library for GPGPU programming that includes a fusion mechanism based on list homomorphisms [25]. The fusion transformation itself is implemented as a source to source translation. SkeTo [24] is a C++ library that provides parallel skeletons for CPUs. SkeTo's use of C++ templates provides a fusion system similar to delayed arrays, which could be equivalently implemented using CUDA templates. The authors of SkeTo note that the lack of type inference in C++ leads them to write their array code as nested expressions — to avoid intermediate variable bindings and their required type annotations.

Acknowledgements. This work was supported in part by the Australian Research Council under grant number LP0989507.

References

[1] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Haskell Symposium*, 2009.

[2] E. Axelsson. A generic abstract syntax model for embedded languages. In *ICFP: International Conference on Functional Programming*. ACM, 2012.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. of High Performance Computing Networking, Storage and Analysis*. ACM, 2009.

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

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

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

[7] J. F. Canny. A Computational Approach to Edge Detection. *Pattern Analysis and Machine Intelligence*, (6), 1986.

[8] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP: Declarative Aspects of Multicore Programming*. ACM, 2011.

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

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

[11] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP: Declarative Aspects and Applications of Multicore Programming*. ACM, 2012.

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

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

[14] A. Gill. Type-Safe Observable Sharing in Haskell. In *Haskell Symposium*, 2009.

[15] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA: Functional Programming Languages and Computer Architecture*. ACM, 1993.

[16] G. Keller and M. M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In *Workshop on High-Level Parallel Programming Models and Supportive Environments*. Springer-Verlag, 1999.

[17] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. L. Peyton Jones, and B. Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *ICFP: International Conference on Functional Programming*. ACM, 2010.

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

[19] B. Lippmeier and G. Keller. Efficient Parallel Stencil Convolution in Haskell. In *Haskell Symposium*. ACM, 2011.

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

[21] W. Ma and G. Agrawal. An integer programming framework for optimizing shared memory use on GPUs. In *PACT: Parallel Architectures and Compilation Techniques*. ACM, 2010.

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

[23] G. Mainland, R. Leshchinskiy, and S. Peyton Jones. Exploiting vector instructions with generalized stream fusion. In *ICFP: International Conference on Functional Programming*. ACM, 2013.

[24] K. Matsuzaki and K. Emoto. Implementing fusion-equipped parallel skeletons by expression templates. In *IFL: Implementation and Application of Functional Languages*. Springer-Verlag, 2010.

[25] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA: Functional Programming and Computer Architecture*, 1991.

[26] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Program.*, 12(5), July 2002.

[27] S. Peyton Jones, S. Marlow, and C. Elliott. Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell. In *IFL: Implementation of Functional Languages*. Springer Heidelberg, 2000.

[28] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Odersky, and K. Olukotun. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL: Symposium on Principles of Programming Languages*. ACM, 2013.

[29] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS: Parallel and Distributed Processing*. IEEE Computer Society, 2009.

[30] S. Sato and H. Iwasaki. A skeletal parallel framework with fusion optimizer for GPGPU programming. In *APLAS: Asian Symposium on Programming Languages and Systems*. Springer-Verlag, 2009.

[31] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Eurographics Association, 2007.

[32] J. Stam. Stable fluids. In *SIGGRAPH: Computer graphics and Interactive Techniques*. ACM Press, 1999.

[33] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3), 2009.