Embedding Foreign Code

Robert Clifton-Everest, Trevor L. McDonell,
Manuel M.T. Chakravarty, and Gabriele Keller

University of New South Wales,
School of Computer Science and Engineering
{robertce,tmcdonell,chak,keller}@cse.unsw.edu.au

Abstract. Special purpose embedded languages facilitate generating high-performance code from purely functional high-level code; for example, we want to program highly parallel GPUs without the usual high barrier to entry and the time-consuming development process. We previously demonstrated the feasibility of a skeleton-based, generative approach to compiling such embedded languages.

In this paper, we (a) describe our solution to some of the practical problems with skeleton-based code generation and (b) introduce our approach to enabling interoperability with native code. In particular, we show, in the context of a functional embedded language for GPU programming, how template meta programming simplifies code generation and optimisation. Furthermore, we present our design for a foreign function interface for an embedded language.

1 Introduction

Accelerate is an *embedded language* for general-purpose GPU programming. It is implemented in Haskell, which also serves as its host language, and generates optimised CUDA code [14] from regular, multi-dimensional array programs [2,13]. Accelerate is an example of a class of embedded languages aiming at simplifing the programming of specialised high-performance architectures by offering a restricted high-level language with a specialised code generator. Other recent examples are Nikola [12], Obsidian [4], Delight/LMS [19], as well as embedded hardware description languages [1,10]. These embedded languages reuse part of the language infrastructure of their host language, while supplying a dedicated and specialised code generator. This reuse is in contrast to standalone languages with similar aims, such as StreamIT [22], Halide [18], and NOVA [7].

Among those languages, Accelerate's implementation is unique by being based on a generative, template-based code generator, in the spirit of Cole's algorithmic skeletons [6]. The main advantage of this approach to code generation is the simplicity with which code idioms of the target architecture can be adopted — this is crucial for GPU programs as GPUs only deliver high performance if both control structures and data access patterns are suitably constrained [20]. The approach's main challenges are two: (1) we need a mechanism to express, instantiate, and compose code skeletons and (2) we need a fusion framework that

M. Flatt and H.-F. Guo (Eds.): PADL 2014, LNCS 8324, pp. 136–151, 2014.

eliminates intermediate structures at skeleton boundaries. In previous work [13], we addressed the second challenge by a novel fusion framework for SIMD languages. In the present paper, we address the first challenge and also explain the interplay between our fusion framework and skeleton instantiation.

Moreover, the use of any special-purpose language in practice needs to address interoperability with native code. In particular, we need to be able to use existing, third-party library code from embedded code as well as enable the use of embedded code from native applications. To this end, we present the design of a *foreign function interface* for embedded array code.

In summary, this paper discusses the generation of high-performance foreign code by way of code skeletons as well as a foreign function interface for embedded programs to leverage native libraries and applications. It makes the following main contributions:

- We discuss how to implement skeleton-based code generation with template meta programming (Section 2).
- We explain how to implement consumer-producer fusion as skeleton instantiation (Section 3).
- We introduce the, to our knowledge, first foreign function interface for an embedded language (Section 4).
- We explain how to integrate embedded Haskell GPU code in a CUDA C program (Section 5).

We discuss benchmarks in Section 6 and related work in Section 7. All code is available from `https://github.com/AccelerateHS/accelerate`.

2 Embedding GPU Programs as Skeletons

Accelerate offers a range of aggregate operations on multi-dimensional arrays. They include operations modelled after Haskell's list library, such as `map` and `fold`, but also array-oriented operations, such as `permute` and stencil convolutions.

As a simple example, consider the dot product of two vectors:

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

The crucial difference to vanilla Haskell is the `Acc` type constructor representing *embedded array-valued computations*. The types `Vector e` and `Scalar e` represent one-dimensional and zero-dimensional (singleton) arrays, respectively.

The expression `zipwith (*) xs ys` implements pointwise multiplication of the two argument vectors, and `fold (+) 0` sums the resulting products up to yield the final, scalar result, wrapped into a singleton array. The type of `fold` is

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

It uses a binary folding function operating on *embedded scalar computations* of type `Exp a` to implement a parallel reduction along the innermost dimension of an n-dimensional, embedded array of type `Array (sh:.Int) a`. The *shape* `sh:.Int` consist of a polymorphic shape `sh` with one added (innermost) dimension, which is missing from the shape of the result array.

2.1 Array Operations as Skeletons

Accelerate's CUDA[1] backend is based around the idea of *algorithmic skeletons* [6]. In other words, the backend implements each of the aggregate array operations, such as `map`, by way of a CUDA C *code template* that is parameterised with array types and worker functions, such as the mapped function.

This generative approach is attractive for specialised hardware, such as GPUs, as the CUDA C code templates are hand-tuned to avoid expensive control flow, ensure efficient global-memory access, and use fast on-chip shared memory for local communication, all of which is required for high-performance GPU code [14]. It is much more difficult —and subject to open research questions— to generate the corresponding code idioms with a synthetic code generator.

In the first version of Accelerate, we implemented CUDA C code templates and template instantiation with a mixture of C++ templates and C preprocessor macros — see [2] for details. While workable, this approach turned out to have a number of problems. Firstly, the use of CPP is fragile and hard to maintain. Template instantiation by inlining of CPP macros required the use of fixed variables with no static checking to ensure the consistent use of names or that used names where defined before their use. Moreover, it was easy to generate code that wasn't even syntactically valid. All this seriously complicated maintenance and further extension of the code generator. Secondly, the approach led to the generation of dead code whenever specific template instances didn't use some of their parameters or fields of structured data. (The CUDA compiler was not able to remove most of this dead code.) Finally, and most importantly, the use of CPP did not scale to support the implementation of producer-consumer skeleton fusion, which is a crucial optimisation, even for code as simple as dot product.

Next, we discuss a new approach to template definition avoiding these problems. Then, we will discuss the implementation of producer-consumer skeleton fusion and general template instantiation in the following section.

2.2 Skeletons as Template Meta Programs

Due to the shortcomings of C++ templates and CPP, we explored the use of template meta programming to implement CUDA skeletons. More specifically, we use Mainland's *quasiquotation* extensions [11] to Template Haskell to define skeletons as quoted CUDA C templates with splices for the template parameters.

[1] CUDA is NVIDIA's C/C++-based framework for general-purpose GPU programming: `http://www.nvidia.com/object/cuda_home_new.html`

```
[cunit|
  __global__ void map( $params:argIn, $params:argOut )                    — (3)
  {
      const int shapeSize     = size(shOut);
      const int gridSize      = $exp:(gridSize dev);

      for (int ix = $exp:(threadIdx dev); ix < shapeSize; ix += gridSize)
      {
          $items:(dce x       .=. get ix)                                 — (2)
          $items:(setOut "ix" .=. f x)                                    — (1)
      }
  }
|]
```

Listing 1. Accelerate CUDA skeleton for the `map` operation

Listing 1 displays the skeleton template for the map family of functions (which also includes `zipWith`). The `[cunit|···|]` brackets enclose CUDA C definitions. CUDA uses the `__global__` keyword to indicate that `map` is a *GPU kernel*: a single data-parallel computation launched on the GPU by the CPU. *Antiquotations* `$params:e`, `$exp:e`, `$items:e`, and `$stms:e` denote template parameters using a Haskell expression `e` to splice CUDA C parameters, expressions, items, and statements, respectively, into the skeleton.

The map skeleton is parameterised by a function `f` that gets applied to the individual array elements in the line marked *(1)*. The arguments to `f` are extracted from the input arrays in the line marked *(2)*, and we will explain the meaning of the auxiliary combinators `get`, `setOut`, `dce`, and `(.=.)` in the next section. Finally, the arguments to a specific instantiation of the `map` template are computed and spliced in the function head marked *(3)*.

As the quasiquoter `[cunit|···|]` executes at Haskell compile time, syntactic errors in the quotations and antiquotations as well as in their composition are flagged at compile time; i.e., we can be sure that the generated code is syntactically correct if we can compile our backend. See [11] for more details on quasiquoters.

3 Instantiating Skeletons

In the first, pre-template meta programming, version of Accelerate we generated one or more CUDA GPU kernels for each aggregate array operation. This scheme led to superfluous intermediate arrays and array traversals. Recall the body of the definition of the dot product: `fold (+) 0 (zipWith (*) xs ys)`. The function `zipWith` compiles to an instance of the `map` template that we discussed in the previous section. Similarly, `fold` compiles to an instance of the `fold` template. As a result, the execution of `zipWith` produces an array that the `fold` kernels consume.

This is not what a CUDA programmer would manually implement; it is more efficient to inline the `zipWith` computation into the kernel of the `fold`. This strategy eliminates one GPU kernel and an intermediate array that is of the same size as the two input arrays. To achieve the same performance as handwritten CUDA code, we developed the array fusion system described in [13].

Our fusion system distinguishes producer-producer and consumer-producer fusion. The former combines two skeletons that produce complex arrays, whereas the latter combines an array producer (such as `map`) with a skeleton reducing an array (such as `fold`). Central to our approach is a representation of arrays as functions, which we call *delayed arrays* (in contrast to *manifest arrays*) and represent as follows:

```
data DelayedAcc a where
  Delayed :: (Shape sh, Elt e)
          => Exp sh                       — array extent
          -> Fun (sh  -> e)               — generate element at index
          -> Fun (Int -> e)               — ...at linear index
          -> DelayedAcc (Array sh e)
```

Instead of generating a `map` skeleton instance for `zipWith` straight away, we represent the computation implemented by `zipWith` as a function —actually, a pair of functions— together with the extent (domain) of the array as a value of type `DelayedAcc`. For more details on this representation, see [13].

As far as skeleton template instantiation goes, the crucial step in Accelerate's CUDA backend is the function `codegenAcc`, which turns an Accelerate array operation (of type `Acc a`) into the AST of instantiated skeleton CUDA code `CUSkeleton a`:

```
codegenAcc :: DeviceProperties -> Acc a -> CUSkeleton a
codegenAcc dev (Fold f z a)
  = mkFold dev (codegenFun dev f) (codegenExp dev z) (codegenDelayed dev a)
codegenAcc dev (Map f a)
  = mkMap ...
```

Here we see that `mkFold`, which generates an instance of the `fold` template, gets the code generated from a delayed array as its last argument from the call to `codegenDelayed`. In the case of the dot product code, that delayed array will be a delayed representation of `zipWith` whose code —as an AST— will be passed to `mkFold`. In the following, we will discuss template instantiation by our skeleton constructors such as `mkFold` and `mkMap`.

3.1 Consumer Producer Fusion by Template Instantiation

The use of template meta programming to implement CUDA skeletons is crucial to enable consumer-producer fusion by way of template instantiation. In the dot product example, the delayed producer is equivalent to the scalar function `λix -> (xs!ix) * (ys!ix)`. The call to `mkFold` in `codegenAcc` passes a CUDA version of this function, which is bound to the argument `get` in the `mkFold` definition given in Listing 2. This delayed producer function is used in the line marked *(1)*, where it expands to the following C code:

```
mkFold :: DeviceProperties -> CUFun (e -> e -> e) -> CUExp e
  -> CUDelayedAcc (Array (sh :. Int) e) -> CUSkeleton (Array sh e)
mkFold dev combine seed (CUDelayed shape _ get)
  = CUSkeleton [cunit|
      __global__ void foldAll( $params:argIn, $params:argOut )
      {   // omitted variable declarations
          if ( ix < shapeSize ) {
              $items:(y .=. get ix)

              for ( ix += gridSize; ix < shapeSize; ix += gridSize ) {
                  $items:(x .=. get ix)                              — (1)
                  $items:(y .=. combine x y)
              }
          }
          $items:(sdata "threadIdx.x" .=. y)
          __syncthreads();                                           — (2)
          $stms:(treeReduce dev combine sdata)
          // first thread writes the result to memory
      }
    |]
```

Listing 2. Accelerate CUDA skeleton for the `foldAll` operation

```
const Int64 v2 = ix;
const int v3 = toIndex(shIn0, shape(v2));
const int v4 = toIndex(shIn1, shape(v2));
y0 = arrIn0_a0[v3] * arrIn1_a0[v4];
```

The functions `shape` and `toIndex` map multi-dimensional indices to linear array representations. In this example these functions do not contribute anything as dot product consumes two vectors, and the CUDA compiler is able to remove the superfluous assignments in this case.

In contrast to the `map` skeleton, the code generated by `mkFold` proceeds in two phases of parallel activities. The first phase is the sequential `for` loop including the use of `get`. The second phase starts after the CUDA `__syncthreads()` statement at the line marked *(2)* and implements a parallel tree reduction [3].

3.2 Instantiating Skeletons with Scalar Code

Most aggregate array operations in Accelerate are parameterised by scalar functions, such as the mapping function for `map` and the binary operator for `fold`. Hence, a crucial part of template instantiation is the inlining of CUDA code implementing scalar Accelerate functions into template code. Inlining of scalar functions is always possible as the scalar sublanguage of Accelerate is first-order and does not support recursion. These restrictions are necessary to generate GPU code as GPU hardware neither supports large stacks (for recursion) nor closures (for higher-order functions).

To splice scalar code fragments into the skeleton code of array operations, we define a typeclass of l-values and r-values to define a generic assignment operator `(.=.)`, which is, for example, used in the lines marked *(1)* and *(2)* in Listing 1. This representation abstracts over whether our skeleton uses l-values in single static assignment-style to `const` declarations or as a statement updating a mutable variable. The class declarations are the following:

```
class Lvalue a where
  lvalue :: a -> C.Exp -> C.BlockItem

class Rvalue a where
  rvalue :: a -> C.Exp

class Assign l r where
  (.=.) :: l -> r -> [C.BlockItem]

instance (Lvalue l, Rvalue r) => Assign l r
  -- method definition omitted
```

Furthermore, we can also bring any additional terms into scope before evaluating an r-value. As an example, see the `get` code fragment in Section 3.1 in the calculations of `toIndex`. We enable this by way of the following class instance:

```
instance Assign l r => Assign l ([C.BlockItem], r)
  -- method definition omitted
```

3.3 Eliminating Dead Code

As mentioned before, one problem of the original code generator based on CPP and C++ templates was its inability to remove some forms of dead code. As an example, consider the following Accelerate function that projects the first component of each element of a vector of quadruples:

```
fst4 :: Acc (Vector (a,b,c,d)) -> Acc (Vector a)
fst4 = map (\v -> let (x,_,_,_) = unlift v in x)
```

The function `unlift` turns an embedded scalar expression that yields a quadruple into a quadruple comprising four embedded scalar expressions — hence, we can pattern match on the quadruple in the let-binding. The use of `fst4` can lead to serious inefficiencies as Accelerate uses a *non-parametric* array representation: arrays of tuples are represented as tuples of arrays. This helps us to maintain the strict memory access rules that CUDA requires for best performance. Clearly, an efficient implementation of this operation should simply select the first tuple component of the representation, only taking constant time.

If a value of type `Vector (a,b,c,d)` is represented as a tuple of arrays, an application of `fst4` should execute in constant time (independent of the size of the array). As explained in [2], to keep the number of skeletons reasonable, our CPP/C++-template code generator represented scalar tuples as C-structs and resorted, during skeleton instantiation, to a family of getter and setter functions

consuming these structs to read and write the elements from the non-parametric array representation.

As a consequence, in `fst4`, array elements are copied into a struct, only for the first element to be extracted again and the struct to be discarded. One might hope that the CUDA compiler spots (1) the redundant copying of array elements and (2) that the elements of three of the four arrays are never used. Alas, it does not and as a result `fst4` does not run in constant time, and it generates considerable memory traffic.

With template meta programming and the `Assign` type class introduced previously, we fare much better. Template instantiation inlines the scalar computations, including all array accesses, directly into the AST representing the skeleton. Instead of packaging the tuple into a `struct`, we represent it by a set of individuals values, one per component. During code generation, we keep track of the values constituting a tuple by maintaining a list of expressions, one for each component of the tuple. Moreover, a generalised version of the `(.=.)` operator allows us to assign all values making up a tuple with one assignment in our meta programming system — i.e., we use lists of l- and r-values:

```
instance Assign l r => Assign [l] [r]
  -- method definition omitted
```

Unfortunately, the CUDA compiler doesn't always eliminate memory reads, as it does not always detect if the values are not used. Hence, rather than rely on the CUDA compiler, we explicitly keep track of which values are used at all in generated scalar code, and when splicing assignments into a skeleton template, we elide dead statement; i.e., those whose results are not used. The following instance of the `Assign`-class uses a flag that is `False` whenever the assigned value of an assignment is not used:

```
instance Assign l r => Assign (Bool,l) r where
  (.=.) (used,lhs) rhs
    | used      = lhs .=. rhs
    | otherwise = []
```

The `map` skeleton of Listing 1 exploits this: when generating code for the mapped function `f`, the function `dce :: [a] -> [(Bool,a)]` —on the line marked *(2)*— determines for each term whether it is being used. Thus, when the code generated by `get` reads data from the input array, it doesn't read unused values. Consequently, `fst4` only touches the array representing the first component of the quadruple of arrays. In combination with fusion, we completely avoid any unnecessary memory traffic.

In summary, the use of template meta programming for skeleton definition and instantiation enables us to combine the advantages of conventional synthetic code generators (such as def-use analysis for dead code elimination) with those of generative skeleton-based code generators (such as handwritten idiomatic code for special-purpose architectures).

4 Using Foreign Libraries

Accelerate is a high-level language framework capturing idioms suitable for massively parallel GPU architectures, without requiring the expert knowledge needed to achieve good performance at the level of CUDA. However, there are existing highly optimised CUDA libraries, for example, for high performance linear algebra and fast Fourier transforms. For Accelerate to be practically useful, we need to provide a means to use those libraries. Moreover, access to native CUDA code also provides a developer the opportunity to drop down to raw CUDA C in those parts of an application where the code generated by Accelerate is not sufficiently efficient. We achieve access to CUDA libraries and native CUDA components with the *Accelerate Foreign Function Interface* (or FFI).

The Accelerate FFI is a two-way street: (1) it enables calling native CUDA C code from embedded Accelerate computations and (2) it facilitates calling Accelerate computations from non-Haskell code. Overall, a developer can implement an application in a mixture of Accelerate and other languages in a manner that the source code is portable across multiple Accelerate backends.

Given that Accelerate is embedded in Haskell, it might seem that Haskell's standard FFI should be sufficient to enable interoperability with foreign code. Unfortunately, this is not the case. With Haskell's standard FFI, we can call C functions that in turn invoke GPU computations from Haskell host code. However, we want to call GPU computations from within embedded Accelerate code and pass data structures located in GPU memory directly to native CUDA code and vice versa. The latter is crucial, as transferring data from CPU memory to GPU memory and back is very expensive.

4.1 Importing Foreign Functions

Calling foreign code in an embedded Accelerate computation requires two steps: (1) the foreign function must be made accessible to the host Haskell program and (2) the foreign function must be lifted into an Accelerate computation to be available to embedded code. For the first step, we use the standard Haskell FFI. The second step requires an extension to Accelerate.

As a concrete example, let us use the vector dot product of the highly optimised *CUDA Basic Linear Algebra Subprograms (CUBLAS)* library [15]. This CUBLAS function is called `cublasSDot()`; it computes the vector dot product of two arrays of 32-bit floating point values. To access it from Haskell, we use this Haskell FFI import declaration:

```
foreign import ccall "cublas_v2.h␣cublasSdot_v2" cublasSdot
  :: Handle
  -> Int                                  — Number of array elements
  -> DevicePtr Float -> Int               — The two input arrays, and...
  -> DevicePtr Float -> Int               — ...element stride
  -> DevicePtr Float                      — Result array
  -> IO ()
```

The `Handle` argument is required by the foreign library and created on initialisation. The `DevicePtr` arguments are pointers into GPU memory. As mentioned before, the primary aim of the Accelerate FFI is to ensure that we do not unnecessarily transfer data between GPU and CPU memory.

To manage device pointers, the Accelerate FFI provides a GPU memory allocation function `allocateArray` and a function `devicePtrsOfArray` to extract the device pointers of an Accelerate array. We can use these functions to invoke `cublasSdot` with GPU-side data:

```
dotp_cublas :: Handle
            -> (Vector Float, Vector Float)
            -> CIO (Scalar Float)
dotp_cublas handle (xs, ys) = do
  let n  = arraySize (arrayShape xs)    — number of input elements
  result    <- allocateArray Z          — allocate a new Scalar array
  ((),xptr) <- devicePtrsOfArray xs     — get device memory pointers
  ((),yptr) <- devicePtrsOfArray ys
  ((),rptr) <- devicePtrsOfArray result
  liftIO $ cublasSdot handle n xptr 1 yptr 1 rptr
  return result
```

The result of `devicePtrsOfArray` is a nested tuple of pointers, as we represent arrays of tuples as tuples of arrays; hence, we can have multiple CUDA arrays for one Accelerate array. In the above example, there is only one, though. The `CIO` monad is simply the `IO` monad enriched with some information used by the CUDA backend to manage devices, memory, and caches.

4.2 Executing Foreign Functions with Accelerate

The function `dotp_cublas` invokes native CUDA code in such a manner that it directly uses arrays in GPU memory. This leaves us with two challenges: (1) we need to enable calling functions, such as `dotp_cublas`, in embedded code and (2) we need to account for Accelerate supporting multiple backends, while Accelerate programs should be portable across backends.

To discuss these issues, we need to briefly recap some of the Accelerate internals described in [2]. Accelerate reifies embedded programs into an abstract syntax tree (AST) encoded as a generalised abstract data type (GADT) to track types of the embedded language in the host language — i.e., the AST can only represent well-typed embedded programs. Accelerate compiles fused collections of array operations into GPU kernels and orchestrates the execution of those kernels CPU-side by a tree traversal of the AST.

Returning to the two remaining challenges, we address the challenge of enabling calling functions, such as `dotp_cublas`, by extending the AST with a new node type `Aforeign` representing foreign calls. One instance of an `Aforeign` node encodes the code for one backend, but it also contains a fallback implementation in case a different backend is being used. The AST data constructor is defined as follows:

```
Aforeign :: (Arrays as, Arrays bs, Foreign f)
         => f as bs                        — foreign function
         -> (Acc as -> Acc bs)             — fallback implementation
         -> Acc as                         — input array
         -> Acc bs
```

When the tree walk during code execution encounters an `Aforeign` AST node, it dynamically checks whether it can execute the foreign function. If it can't, it instead executes the fallback implementation. A fallback implementation might be another `Aforeign` node with native code for a different backend (e.g., for OpenCL instead of CUDA), or it can simply be a vanilla Accelerate implementation of the same functionality that is provided by the foreign code. With a cascade of `Aforeign` nodes, we can provide an optimised native implementation of a function for a range of backends and still maintain a vanilla Accelerate version of the same functionality for execution in the Accelerate interpreter.

The dynamic check for the suitability of a foreign function is facilitated by the class constraint `Foreign f` in the context of `Aforeign`. The class `Foreign` is a subclass of `Typeable` with instances for data types that represent foreign functions for specific backends. For the CUDA backend, we have the following:

```
class Typeable2 f => Foreign f where ...
instance Foreign CUDAForeignAcc where ...
data CUDAForeignAcc as bs where
  CUDAForeignAcc :: as -> CIO bs
```

`CUDAForeignAcc` wraps calls to foreign CUDA code executed in the `CIO` monad. When the CUDA backend encounters an AST node `Aforeign foreignFun alt arg`, it attempts to `cast`[2] the value of `foreignFun` to type `CUDAForeignAcc as bs`. If that `cast` succeeds, it can unwrap the `CUDAForeignAcc` and invoke the function it contains. Otherwise, it needs to execute the alternative implementation `alt`.

Finally, we can define an embedded vector dot product that uses CUBLAS when possible and, otherwise, falls back to the version defined in Section 3.1:

```
dotp' :: Acc (Vector Float) -> Acc (Vector Float)
      -> Acc (Scalar Float)
dotp' xs ys = Aforeign (CUDAForeignAcc (dotp_cublas handle))
                       (uncurry dotp)
                       (lift (xs, ys))
```

Foreign calls are not curried; hence, they only have got one argument, which is an instance of the class `Arrays` of tuples of Accelerate arrays.

4.3 Embedding Foreign Scalar Functions

So far, we discussed the use of foreign array computations from Accelerate. However, we also wish to be able to use foreign scalar operations in embedded array computations. For example, CUDA provides fused floating-point multiply-add intrinsics with a variety of rounding modes.

[2] See Haskell's `Data.Typeable` library for details on `cast`.

We import foreign scalar functions similarly to foreign array computations. In particular, the AST type `Exp` for scalar embedded computations includes a data constructor `Foreign` that serves the same purpose as `Aforeign` for `Acc`:

```
Foreign :: (Elt x, Elt y, Foreign f)
        => f x y -> (Exp x -> Exp y) -> Exp x -> Exp y
```

Where we used `CUDAForeignAcc` to wrap CUDA array computations for use with `Aforeign`, we use `CUDAForeignExp` to wrap scalar CUDA functions for use with `Foreign`. However, instead of wrapping a Haskell FFI call, the scalar case simply encodes the textual representation of the CUDA function in CUDA code. As discussed in Section 2, scalar code is used to instantiate skeleton templates. The skeleton code is a template for CUDA code; so, a Haskell function invocation wouldn't be appropriate. As in the array case, functions are uncurried, but in the scalar case, they can only return a single scalar argument:

```
data CUDAForeignExp x y where
  CUDAForeignExp :: IsScalar y
                 => [String] -> String -> CUDAForeignExp x y
```

The first argument is a list of header files that need to be included when compiling an instantiated skeleton template including this specific foreign function.

Overall, we define a foreign function based on CUDA's explicitly fused floating-point multiply-add intrinsics as follows (using IEEE rounding towards zero):

```
fmaf :: Exp Float -> Exp Float -> Exp Float -> Exp Float
fmaf x y z = Foreign (CUDAForeignExp [] "__fmaf_rz")
                     (\v -> let (x,y,z) = unlift v in x * y + z)
                     (lift (x, y, z))
```

5 Embedding Embedded Programs

Accelerate simplifies writing GPU code as it obviates the need to understand most low-level details of GPU programming. Hence, we would like to use Accelerate from other languages. As with importing foreign code into Accelerate, the foreign export functionality of the standard Haskell FFI is not sufficient for efficiently using Accelerate from languages, such as C. In the following, we describe how the Accelerate FFI supports exporting Accelerate code as standard C calls.

5.1 Exporting Accelerate Programs

To export Accelerate functions as C functions, we make use of Template Haskell [21]. For example, we might export our Accelerate dot product:

```
dotp :: Acc (Vector Float, Vector Float) -> Acc (Scalar Float)
dotp = uncurry $ \xs ys -> fold (+) 0 (zipWith (*) xs ys)

exportAfun 'dotp "dotp_compile"
```

The function `exportAfun` is defined in Template Haskell and takes the name of an Accelerate function, here `dotp`, as an argument. It generates the necessary export declarations by inspecting the properties of the name it has been passed, such as its type.

Compiling a module that exports Accelerate computations in this way (say, `M.hs`) generates the additional file `M_stub.h` containing the C prototype for the foreign exported function. For the dot product example, this header contains:

```
#include "HsFFI.h"
extern AccProgram dotp_compile(AccContext a1);
```

A C program needs to include this header to call the Accelerate dot product.

5.2 Running Embedded Accelerate Programs

One of the functions to execute an Accelerate computation in Haskell is:

```
run1In :: (Arrays as, Arrays bs)
       => Context -> (Acc as -> Acc bs) -> as -> bs
```

This function comprises two phases: (1) program optimisation and instantiation of skeleton templates of its second argument and (2) execution of the compiled code in a given CUDA context (first argument). The implementation of `run1In` is structured such that, partially applying it to only its first and second argument, yields a new function of type `as -> bs`, where Phase (1) has been executed already — in other words, it precompiles the Accelerate code. Repeated application of this function of type `as -> bs` executes the CUDA code without any of the overheads associated with just-in-time compilation.

The Accelerate export API retains the ability to precompile Accelerate code. The C function provided by `exportAfun` compiles the Accelerate code, returning a reference to the compiled code. Then, in a second step, `runProgram` marshals input arrays, executes the compiled program, and marshals output arrays:

```
OutputArray   out;
InputArray    in[2]   = { ... };
AccProgram    dotp    = dotp_compile( context );

runProgram( dotp, in, &out );
```

The function `dotp_compile` was generated by `exportAfun 'dotp "dotp_compile"`.

5.3 Marshalling Input and Output Arrays

Accelerate uses a non-parametric representation of multi-dimensional arrays: an array of tuples is represented as a tuple of arrays. The type `InputArray` follows this convention. It is a C struct comprising an array of integers indicating the extent of the array in each dimension together with an array of pointers to each underlying GPU array of primitive data.

```
typedef struct { int* shape; void** adata; } InputArray;
```

Table 1. General performance of Accelerate (in ms) — c.f., [13]

Benchmark	Input Size	Contender	Accelerate
Black Scholes	20M	6.70 (CUDA)	6.19 (0.92×)
Dot Product	20M	1.88 (CUBLAS)	2.35 (1.25×)
N-Body	32k	54.42 (CUDA)	102.47 (1.88×)
SMVM (protein)	4M	0.641 (CUSP)	0.637 (0.99×)

Table 2. Fast Fourier Transform based benchmarks (in ms)

Benchmark	Input Size	Contender	Accelerate full	Accelerate no fusion	Accelerate no FFI
FFT	512×512	43 (FFTW)	4.36 (0.1×)	5.9 (0.14×)	3658 (8.5×)
High pass	512×512	65 (FFTW)	14.97 (0.23×)	27.82 (0.43×)	21936 (34×)
SmoothLife	128×128	16.21 (MATLAB)	4.01 (0.25×)	6.38 (0.39×)	6829 (42×)

`OutputArray` includes an extra field, a stable pointer, that maintains a reference to the associated Haskell-side `Array`. This keeps the array from being garbage collected until the `OutputArray` is explicitly released with `freeOutput`.

```
typedef struct { int* shape; void** adata;
                 HsStablePtr stable_ptr; } OutputArray;
```

6 Applications and Benchmarks

We conducted benchmarks 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 runtimes are the average of 100 runs.

Table 1 establishes baseline Accelerate performance, showing a comparison of kernel runtimes for a selection of Accelerate programs compared to native CUDA implementations. Accelerate is clearly competitive.

6.1 Fast Fourier Transform (FFT) — Foreign Import

The column "Accelerate, no FFI" in Table 2 measures a pure Accelerate implementation of an out-of-place Cooley-Tukey FFT algorithm [8], whereas "Accelerate, full" uses the FFI to access NVIDIA's highly optimised CUFFT library [16]. "Accelerate, no fusion" also uses the FFI, but without fusion.

The row labelled "FFT" measures a single forward Fourier transform of a greyscale image. The row labelled "High pass" is a high-pass filter of an RGB image, which for each component performs a forward transform, zeros out the centre (high) frequencies, then performs the inverse transform. Finally, the row "SmoothLife" measures a generalisation of Conway's *Game of Life* to a continuous domain [17], which is based on Fourier transforms.

We compare the single FFT and the high-pass filter to the highly regarded FFTW library [9] (multithreaded, estimate mode). We compare the Accelerate implementation of SmoothLife to SmoothLife's reference implementation in MATLAB (version R2012B). FFTW and MATLAB execute on multicore CPUs.

In all cases, our out-of-place Cooley-Tukey implementation of FFT in pure Accelerate is much slower than the highly optimised FFTW and MATLAB multicore implementations. However, once we use the Accelerate FFI to utilise CUFFT, the Accelerate code clearly outperforms the FFTW and MATLAB implementations. This is although we incur significant overhead due to a mismatch of complex number representations. CUFFT represents complex numbers in a packed AoS format, requiring marshalling to and from Accelerate's SoA representation. Array fusion allows this additional overhead to be integrated into surrounding operations, amortizing the cost of this impedance mismatch when calling foreign libraries. This is particularly noticeable in the high-pass filter benchmark. We leave native support of packed vector types to future work.

6.2 N-Body — Foreign Export

To demonstrate the use of Accelerate code from C, we use an n-body example that simulates Newtonian gravitational forces on a set of massive bodies in 3D space, using the naive $O(n^2)$ algorithm. We export the Accelerate n-body implementation into an OpenGL program that visualises the positions of the particles at each step of the simulation. The visualisation program — part of the n-body example from the NVIDIA CUDA distribution — uses a packed AoS representation for which we had to introduce additional marshalling. We did not note any performance difference between executing the Accelerate program from Haskell compared to execution via the C-based visualisation program. This is because the $O(n)$ additional marshalling is dominated by the $O(n^2)$ n-body calculations.

7 Related Work

Our work is based on the quasiquotation extension to Template Haskell described in [11] to instantiate the skeletons by splicing in parameters and customised code. The flexibility of this approach is essential for many of our optimisations.

Nikola [12] and Obsidian [5] also embed GPU computations in Haskell, but are not based on skeletons. Obsidian offers no FFI. Nikola does not have an FFI as such, but it allows to embed CUDA code blocks in Nikola programs. Since it only supports single kernel programs, it only deals with limited interactions between the imported code and the rest of the EDSL program.

Delite/LMS [19] is a framework for parallel DSLs in Scala using library-based multi-pass staging. It is not based on skeletons and doesn't seem to have an FFI.

NOVA [7] is a *standalone* functional language for GPU programming, which unlike Accelerate supports nested parallel computations. It also allows importing foreign functions, but not for exporting NOVA computations.

Acknowledgements. We thank Serge Le Huitouze for helpful comments.

References

1. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in Haskell. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming. ACM (1998)
2. Chakravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: DAMP: Declarative Aspects of Multicore Programming. ACM (2011)
3. Chatterjee, S., Prins, J.: COMP663: Parallel Computing Algorithms. Department of Computer Science, University of North Carolina at Chapel Hill (2009)
4. Claessen, K., Sheeran, M., Svensson, B.J.: Expressive array constructs in an embedded GPU kernel programming language. In: DAMP: Declarative Aspects and Applications of Multicore Programming. ACM (2012)
5. Claessen, K., Sheeran, M., Svensson, J.: Obsidian: GPU programming in Haskell. In: IFL: Implementation and Application of Functional Languages (2008)
6. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. The MIT Press (1989)
7. Collins, A., Grewe, D., Grover, V., Lee, S., Susnea, A.: Nova: A functional language for data parallelism. Tech. rep., NVIDIA (2013)
8. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation (90) (1965)
9. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005); Special issue on "Program Generation, Optimization, and Platform Adaptation"
10. Gill, A., Bull, T., Kimmell, G., Perrins, E., Komp, E., Werling, B.: Introducing Kansas Lava. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 18–35. Springer, Heidelberg (2010)
11. Mainland, G.: Why it's nice to be quoted. In: Haskell Symposium, p. 73. ACM Press, New York (2007)
12. Mainland, G., Morrisett, G.: Nikola: Embedding compiled GPU functions in Haskell. In: Haskell Symposium. ACM (2010)
13. McDonell, T.L., Chakravarty, M.M.T., Keller, G., Lippmeier, B.: Optimising Purely Functional GPU Programs. In: ICFP: International Conference on Functional Programming (September 2013)
14. NVIDIA: CUDA C Programming Guide (2012)
15. NVIDIA: CUBLAS Library (2013)
16. NVIDIA: CUFFT Library (2013)
17. Rafler, S.: Generalization of Conway's "Game of Life" to a continuous domain–SmoothLife (2011)
18. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: PLDI 2013. ACM (2013)
19. Rompf, T., Sujeeth, A.K., Amin, N., Brown, K.J., Jovanovic, V., Lee, H., Odersky, M., Olukotun, K.: Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In: POPL 2013. ACM (2013)
20. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Symposium on Graphics Hardware. Eurographics Association (2007)
21. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, pp. 1–16. ACM (2002)
22. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)