Nikola: Embedding Compiled GPU Functions in Haskell

Geoffrey Mainland and Greg Morrisett
Harvard School of Engineering and Applied Sciences
{mainland,greg}@eecs.harvard.edu

Abstract

We describe Nikola, a first-order language of array computations embedded in Haskell that compiles to GPUs via CUDA using a new set of type-directed techniques to support re-usable computations. Nikola automatically handles a range of low-level details for Haskell programmers, such as marshaling data to/from the GPU, size inference for buffers, memory management, and automatic loop parallelization. Additionally, Nikola supports both compile-time and run-time code generation, making it possible for programmers to choose when and where to specialize embedded programs.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages

General Terms Languages, Design

Keywords Meta programming, CUDA, GPU

1. Introduction

A domain-specific language (DSL) captures knowledge unique to a specialized problem domain, allowing programmers to write concise, understandable programs tailored to a specific class of problems. By embedding a domain-specific language in a rich host language, yielding an embedded domain-specific language (EDSL), system designers can leverage the existing type system, syntax and libraries available in the host language, freeing them to focus on the issues unique to the problem at hand instead of the details of language implementation. Haskell has been a particularly popular vehicle for EDSLs with domains as varied as parsing (Hutton 1992), pretty-printing (Hughes 1995), efficient image manipulation (Elliott 2003), robotics (Pembeci et al. 2002) and hardware circuit design (Bjesse et al. 1998).

Broadly speaking, there are two styles of EDSLs. Shallow embeddings make little or no effort to represent the syntax of the embedded language, using host language functions to represent function in the embedded language and host language values to represent embedded language values. Deep embeddings manifest the abstract syntax of the embedded language as data that can be manipulated. The former style of EDSL is suited to applications like pretty-printing and parsing, where the DSL serves as a "short-hand" for a program that could be written directly in the host language. DSLs that denote programs in a language other than the host language require a deep embedding because they interface to a compiler or interpreter that expects to be handed a program represented as data, e.g., a string or an abstract syntax tree. For example, Nikola re-uses the CUDA compiler, which takes care of the lowest-level details of mapping C-like programs onto the GPU instruction set.

Deep embeddings that generate code in a target language that is callable from Haskell allow functional programming to be used in new domains without the overhead of writing a complete parser, type checker and compiler. This style of embedding not only provides the syntactic convenience and aesthetic satisfaction of combinator libraries like those for parsing and pretty-printing, but it allows programmers to express computations that cannot be expressed practically in Haskell. These computations may be impractical to express in Haskell because they take place off-CPU on devices such as GPUs or FPGAs, or expressing them using an embedding may admit a much more efficient compilation strategy than a pure Haskell implementation. Haskell's FFI provides one way to integrate external code, but using it means losing the convenience of writing only Haskell. Ideally, code-generating EDSLs should integrate with Haskell as smoothly as pure-Haskell EDSLs.

We advocate code-generating EDSLs that are *first-class* in the sense that EDSL functions are compiled to Haskell-callable functions and function compilation and function invocation can occur in the same run of a program, i.e., stages can be freely mixed. This allows functions to be either compiled once and for all or specialized to their arguments. For example, a routine that calculates the product of several matrices could optimize the order of matrix-multiply operations based on the dimensions of the matrices. Embedding DSLs in this way retains many of the benefits of staged languages like MetaML (Taha and Sheard 1997) with the added advantage that the object language can differ from the meta-language.

Contributions

We demonstrate the power of deep embeddings using Nikola, an EDSL for efficient array manipulation using GPUs with an interface in the style of the Haskell `vector` package (Leshchinskiy 2010). Our contributions are:

- We demonstrate how a deep embedding's abstract syntax representation can preserve sharing of lambda expressions even when they occur in an application. EDSL functions are translated to target language functions, and EDSL function applications are translated to target language function calls. Existing work on observable sharing shows how Haskell-level sharing of expressions can be preserved in a deep embedding's abstract syntax (Claessen and Sands 1999; Gill 2009), but to our knowledge we are the first to demonstrate how to observe sharing of functions within function applications, where it was previously assumed that full inlining was inevitable (see Elliott et al. 2003, Section 11). The programmer chooses the lambda expressions for which sharing is preserved.
- We show how to compile functions of arbitrary arity in an embedded, first-order array manipulation language to a target lan-

guage that runs directly on GPUs. Furthermore, because we can compile functions and not just computations, functions in our embedded language can be compiled once and applied to many inputs. Values move fluidly and automatically between the host language and embedded language. Our compilation target has the added constraint that all memory needed by a function must be pre-allocated, which we also handle automatically. In general, the size of the outputs will depend upon the values of the inputs, which requires support for size inference in the compiler.

- Our compilation strategy permits compilation at either run-time or at *Haskell* compile time. That is, the embedded language can be compiled during the host language's compilation phase. This requires no changes to GHC.
- One key advantage of our embedding is that although programmers can use the higher-level abstractions provided by Nikola, they can also directly embed CUDA functions. Thus, if the Nikola compiler does not provide needed functionality, one can always drop down and write CUDA directly. Calling a directly-embedded CUDA function requires only an appropriate type signature and a small amount of glue code to pre-allocate any memory the function needs.

The rest of this paper is organized as follows. We begin in Section 2 by providing background and discussing related work. Section 3 describes how we embed the Nikola language in Haskell and describes how to rewrite a pure-Haskell array manipulation function in Nikola so that it can be executed efficiently on a GPU. Our strategy for translating Nikola to CUDA is discussed in Section 4, and in Section 5 we describe how a Nikola function is compiled and called. In Section 6 we evaluate the performance of Nikola. We describe future work and conclude in Section 7. Nikola is available at `http://www.eecs.harvard.edu/~mainland/projects/nikola`.

2. Background and related work

Embedding code-generating domain-specific languages in Haskell was originally advocated by Leijen and Meijer (1999). They developed a DSL for describing database queries that could be translated to SQL and handled marshalling data between Haskell and the SQL execution engine. Pan (Elliott 2003), a DSL for describing image manipulation, generates C code. The techniques we use to convert higher-order abstract syntax (HOAS) to a first-order representation were pioneered by Pan and described in detail by Elliott et al. (2003).

Our past work, Flask (Mainland et al. 2008), was a domain-specific language for sensor networks. Flask allowed programmers to mix code in a high-level first-order functional language with code written in NesC (Gay et al. 2003), a low-level C-like language designed explicitly for sensor networks. Sensor network programs were translated to NesC which was then compiled to binaries that could be installed on individual sensor nodes. Flask required explicit staging; programmers could not mix the execution of Haskell and sensor network programs.

In the the GPU domain, Vertigo (Elliott 2004) is an EDSL for programming 3D graphics that compiles to GPU code. Obisidian (Svensson et al. 2008, 2010) is a vector-manipulation DSL embedded in Haskell similar in style to the Lava (Bjesse et al. 1998) circuit description DSL; computations are written using combinators provided by a library. It provides relatively low-level primitives for describing GPU computations. All Obsidian computations are functions from a single input array to a single output array, although the types of the values contained in the input and output arrays may vary. The size of the input array is statically known at compile time and is limited to the maximum number of threads that a CUDA thread block can legally contain (512 on current CUDA-capable hardware).

The embedded GPU language presented by Lee et al. (2009) is higher-level than Obsidian; one could imagine implementing some of the primitives provided by this language in Obsidian. A full compiler from the embedded language to CUDA is incomplete, so it is unclear how it maps the high-level language onto CUDA. The representation used for the DSL abstract syntax (Chakravarty et al. 2009) does not admit functions; it is only able to represent computations. This necessitates specializing a function to its inputs before compilation. Although specialization can enable additional argument-specific optimizations, our focus is on compiling general functions.

Libraries that share some of our goals exist for other languages. Perhaps the most widely used is PyCUDA (Klöckner et al. 2009), a Python library for accessing CUDA-enabled hardware. It provides run-time code generation facilities, allowing CUDA functions to be compiled on-the-fly and called from Python. CUDA code is represented either as a string or by using a (partially) data type-like representation in which some components of the abstract syntax are represented using Python classes and others using strings. Unlike Nikola, the programmer must explicitly specify how to marshal data to the CPU, and no size inference is performed, so memory management is manual. PyCUDA is also dynamically typed.

Nikola is a high-level language for array computations, similar in that way to the DSL described by Lee et al. (2009). Its contributions relative to the discussed related work include:

- **Minimal syntactic overhead**. As shown in Section 3, Nikola requires minimal changes to Haskell code in order to compile it for execution on a GPU as long as the functionality of the Haskell code falls within Nikola's domain. Programmers do not need to write in a monadic style or use new combinators, and Haskell's binding constructs are sufficient for expressing binding in Nikola. We also show how to use Haskell's function application to represent function application in an embedded language which to our knowledge has not been done before.
- **General function compilation**. We do not require that functions have type $\mathsf{Array}\ \alpha \to \mathsf{Array}\ \beta$, artificially limit the size of the arrays a function may use, or specialize a function to its arguments before compilation. Nikola functions may be of any arity, and after functions are compiled, they may be called an arbitrary number of times with differing arguments, e.g., a Nikola function that increments every element in a vector of floats is compiled once and the compiled function can be called many times, each time with a vector of floats having any length.
- **Choice between compile-time and run-time compilation**. Nikola functions can be compiled at run-time, permitting specialization to a particular piece of hardware, or at Haskell compile-time. The trade-off is between the overhead of compiling a function every time a program is invoked vs. the flexibility to specialize a function to a device.
- **Ability to directly embed CUDA code**. The programmer always has the option to drop down to pure CUDA code. As long as the function obeys the Nikola calling convention, the programmer only needs to add a Haskell type signature and a small amount of glue code to enable calling the CUDA function directly from Haskell.

3. Embedding Nikola

We begin by showing how to attain a deep embedding in Haskell, allowing programmers to write in a subset Haskell that is eventually compiled and loaded not by the Haskell compiler, but by

```
blackscholes :: Vector Float   -- Stock prices
             → Vector Float    -- Option strikes
             → Vector Float    -- Option years
             → Vector Float
blackscholes ss xs ts =
    zipWith3 (λs x t → blackscholes1 s x t r v)
      ss xs ts
  where
    r = ...
    v = ...

blackscholes1 :: Float   -- Stock price
              → Float    -- Option strike
              → Float    -- Option years
              → Float    -- Riskless rate
              → Float    -- Volatility rate
              → Float
blackscholes1 s x t r v =
    s ∗ normcdf d1 − x ∗ exp (−r ∗ t) ∗ normcdf d2
  where
    d1 = (log (s / x) + (r + v ∗ v / 2) ∗ t) / (v ∗ sqrt t)
    d2 = d1 − v ∗ sqrt t

normcdf :: Float → Float
normcdf x = if x < 0 then 1 − w else w
  where
    w     = 1.0 − 1.0 / sqrt (2.0 ∗ π) ∗
              exp (−l ∗ l / 2.0) ∗ poly k
    k     = 1.0 / (1.0 + 0.2316419 ∗ l)
    l     = abs x
    poly  = horner coeff
    coeff = [0.0, 0.31938153,
              − 0.356563782, 1.781477937,
              − 1.821255978, 1.330274429]

horner coeff x = foldr1 madd coeff
  where
    madd a b = b ∗ x + a
```

Listing 1: Black-Scholes call option valuation in Haskell

the DSL library. Our running example is Black-Scholes call option valuation. A Haskell implementation, utilizing the `vector` library, is shown in Listing 1. This implementation is similar to the CUDA implementation included in NVIDIA's CUDA SDK and uses Horner's algorithm in computing a polynomial approximation to the cumulative distribution function of the standard normal distribution.

Our high-level goal is to maintain the syntactic convenience of Haskell, allowing the programmer to write a Nikola version of blackscholes much as the Haskell implementation is written while still allowing the function to be converted to a first-order representation suitable for compilation. To do this we utilize higher-order abstract syntax (HOAS) (Pfenning and Elliot 1988), which represents binders in our embedded language using Haskell's binders. Ideally, we want also to be able to represent let bindings in our embedded language using Haskell's let bindings, and function application in our embedded language using Haskell's function application. We describe how to accomplish these two tasks in Section 3.1 and Section 3.2.

Instead of directly computing values, we will trick our Haskell functions into computing program fragments that, when run, compute the appropriate value. This use of multi-stage programming (Taha and Sheard 1997) in Haskell was pioneered by Conal Elliott in his work on Pan (Elliott 2003). Fortunately, Haskell's pervasive use of type classes to overload standard mathematical operators lets us accomplish this rather easily without having to change program syntax, which we demonstrate using a small functional language with the following first-order representation for its abstract syntax:

```
type Var = String
data DExp = VarE Var
          | LetE Var DExp DExp
          | LamE Var DExp
          | AppE DExp DExp
          | FloatE Float
          | IfThenElseE DExp DExp DExp
          | BinopE Binop DExp DExp
data Binop = LessThan | GreaterThan | ...
          | Add | Mul | Sub | ...
newtype Exp a = E {unE :: DExp}
```

Here DExp is the type of dynamic (untyped) expressions. In practice we wish to maintain the ability to assign meaningful types to the abstract syntax trees we build in a DSL. To simplify the presentation, we use phantom types here and discuss the use of GADTs in Section 3.3. Exp wraps a DExp while adding a phantom type parameter, a, that represents the (embedded) type of the wrapped abstract syntax.

Because addition and multiplication are overloaded and integer literals are desugared into calls to the overloaded function fromInteger, we can define an appropriate instance of the Num type class so that + and ∗ operate over abstract syntax. Note that the seemingly recursive call to fromInteger on the right-hand side of our definition of fromInteger is actually a call to the instance that converts an Integer to a Float; the instance we define converts an Integer to an Exp. Instances for the other numeric type classes are defined similarly.

```
instance Num (Exp Float) where
  e1 + e2 = E $ AddE (unE e1) (unE e2)
  e1 ∗ e2 = E $ MulE (unE e1) (unE e2)
  e1 − e2 = E $ SubE (unE e1) (unE e2)
  fromInteger n = E $ FloatE (fromInteger n)
```

While overloading numeric operators enables us to use the same syntax for an embedded language as we would for Haskell as long as we are writing numeric expressions, expressions involving control flow require new syntax. Consider the normcdf function in Listing 1, which tests whether or not x is less than zero and branches in the result of the test. If we rewrite this as a Nikola function, then x is not a value, but an *expression*. Performing the comparison test on x would require writing a decision procedure that can in general determine whether or not the expression x is less than zero. Clearly no such decision procedure exists. Instead, we can construct a term in the embedded language that compares the sub-expression x to zero and executes the proper branch; we delay the comparison-and-branch so that it is executed not when the embedded term representing normcdf x is built, but when this term is later evaluated. In general, any expression that scrutinizes a value must be re-written to incorporate the scrutinization into the expression. Since we cannot overload the if-then-else construct in Haskell, we have to introduce new operators specific to our embedding. This allows us to write a Nikola version of normcdf as follows:

```
(?) :: Exp Bool → (Exp a, Exp a) → Exp a
test ? (e1, e2) =
    E $ IfThenElseE (unE test) (unE e1) (unE e2)
(.<.) :: Exp Float → Exp Float → Exp Bool
e1 .<. e2 =
    E $ BinopE LessThan (unE e1) (unE e2)
normcdf :: Exp Float → Exp Float
normcdf x = (x .<. 0) ? (1 − w, w)
  where
    ...
```

Note that the only necessary syntactic changes to make normcdf a Nikola function are to rewrite the if-then-else construct and to change the type signature of normcdf. With these changes, normcdf now operates not on Float values, but on program fragments of type Exp Float.

3.1 let-sharing

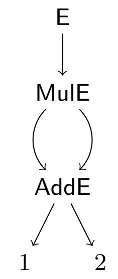
Consider the simple function square, defined as:

```
square :: Exp Float → Exp Float
square x = x ∗ x
```

The expression square $(1 + 2)$ evaluates to the following abstract syntax for our embedded language:

```
E (MulE (AddE (FloatE 1.0) (FloatE 2.0))
        (AddE (FloatE 1.0) (FloatE 2.0)))
```

When this term is eventually evaluated, the value of the subexpression $1 + 2$ will be computed twice, even though we expect by looking at the definition of square that it would only be computed once. Of course GHC knows to only calculate the *term* representation of $1 + 2$ once, so the in-memory representation of our embedded language expressions is:



The problem is that we cannot observe the sharing introduced by Haskell bindings. When we try to do a code generation pass, we will therefore end up processing the expression twice, losing the sharing, and the code generated for the expression $1 + 2$ will do twice the work it needs to. In this simple example the duplicated work is minimal, but for normcdf this kind of loss of sharing causes k to be re-evaluated five times in the expression generated by poly k, leading to a substantial increase in the cost of calling normcdf.

Ideally we would like to find a way to make the sharing implicit in our term representation explicit. This would allow us to use Haskell's let bindings to represent let bindings in our embedded language and yield the following alternate representation for square $(1 + 2)$:

```
E (LetE "x" (AddE (FloatE 1) (FloatE 2))
       (MulE (VarE "x") (VarE "x")))
```

We call this type of sharing **let**-sharing because by properly detecting it we can construct an embedded language term where shared sub-terms are replaced with a let-bound expression. Our goal is to define a family of functions, reify, that rewrites terms in our embedded language to express sharing using the LetE constructor just as we have here for the term representing square $(1 + 2)$. This family of functions will have a member at each type (Exp a_1 → ... → Exp a_n) → IO DExp.

There have been a number of approaches in the literature to making sharing observable. Pan attempted to recover some sharing post hoc by performing common sub-expression elimination on the embedded language's abstract syntax (Elliott et al. 2003). Another solution, proposed by O'Donnell (1993), is to require that the programmer label each expression in the embedded language with an explicit tag. This burdens the programmer with ensuring that different terms have different tags. Lava (Bjesse et al. 1998) lifts this burden by requiring that embedded terms be written in monadic style so that fresh names can be gensym'ed. However, forcing the programmer to write in a monadic style is undesirable; our goal is to require as few syntactic changes as possible relative to Haskell when writing in Nikola.

Claessen and Sands (1999) add a reference type Ref a to Haskell, along with the following operations:

```
type Ref a = ...
ref    :: a → Ref a
deref  :: Ref a → a
(<=>)  :: Ref a → Ref a → Bool
```

Although this avoids forcing the programmer to write in a monadic style, it introduces a non-conservative extension to the language and requires explicitly marking any value for which one might want to recover sharing.

The first truly satisfying solution to the let-sharing problem is given by Gill (2009), which allows the sharing implicit in Haskell's let-bindings to be observed from within the IO monad. Although the observation of sharing occurs in a monad, the term under scrutiny *does not* need to have been written in a monadic style. Observation of sharing occurs once as part of the process of converting HOAS to a first-order representation.

Gill's technique relies on *stable names*, a feature provided by GHC that enables what is essentially pointer equality. The primitives we need are provided by the GHC-specific module System.Mem.StableName and have the following interface:

```
data StableName a
makeStableName :: a → IO (StableName a)
hashStableName :: StableName a → Int
instance Eq (StableName a)
```

Critically, given x and y, if the StableName values returned by applying makeStableName to x and y are equal, then x and y are equal. The implication is one-way; equality of x and y does not imply equality of their stable names. Stable names therefore only allow us to conservatively approximate the true sharing in a Haskell expression, although in practice the approximation is not very conservative and we observe exactly the sharing we expect to see. Note that makeStableName is not strict in its argument; we can force values to weak-head normal form using strict application (the $! operator) to maximize sharing.

We can use these tools to write reify. We begin with a function reifyR :: DExp → R DExp that uses the *reification monad* R to maintain a map from the stable name of a DExp to the rewritten version of the same DExp in which implicit sharing has been made explicit. As reifyR recurses over a term, sub-expressions that have not been seen before are bound to a gensym'ed variable and an entry mapping the sub-expression's stable name to the gensym'ed variable is added to the map; a sub-expression whose stable name already exists in the map is replaced by the variable with which it is associated in the map.

Functions are reified by gensym'ing a fresh variable for each parameter and passing them as the arguments to the function. We use type classes to define a family of functions, reifyFunR, at each type Exp $a_1 \rightarrow \ldots \rightarrow$ Exp a_n.

```
class (Typeable a, Typeable b) ⇒ ReifiableFun a b where
    reifyFunR :: (a → b) → R DExp
instance (Typeable a, Typeable b)
  ⇒ ReifiableFun (Exp a) (Exp b) where
    reifyFunR = ...
instance (Typeable a, ReifiableFun b c)
  ⇒ ReifiableFun (Exp a) (b → c) where
    reifyFunR = ...
```

With these definitions and the function evalR :: R a → IO a, writing the reify function that was our original goal is trivial.

```
class Reifiable a where
  reify :: a → IO DExp
instance Reifiable (Exp a) where
  reify = evalR ∘ reifyR ∘ unE
instance ReifiableFun a b ⇒ Reifiable (a → b) where
  reify = evalR ∘ reifyFunR
```

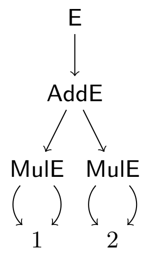
Using this technique to recover let-sharing does not require adding a reference type or writing in a monadic style; sharing is recovered using a single rewriting step that takes place in the IO monad. Our reify is a special case of Gill's more general technique that rewrites a tree as a graph that reflects shared structure; we rewrite a tree (abstract syntax) as a new tree that reflects shared structure using the abstract syntax's let binding construct.

3.2 λ-sharing

There is another kind of sharing that, to our knowledge, no previous techniques allow us to observe. Consider the expression square $1.0 +$ square 2.0 which evaluates to the following abstract syntax:

```
E (AddE (MulE (FloatE 1.0) (FloatE 1.0))
        (MulE (FloatE 2.0) (FloatE 2.0)))
```

Again, we show the in-memory representation of this value, which reflects the physical sharing that results from square's binding of its argument x.



Although we can use any of the techniques described in Section 3.1 to recover the physical sharing in this expressions, there is no way to use them to observe that the two MulE sub-terms both resulted from a function application. The problem is that Haskell's β-reduction effectively inlines the definition of square. The term we *want* to construct binds square to its definition and then applies it in each sub-term of the addition:

```
E (LetE "square"
        (LamE "x" (MulE (VarE "x") (VarE ("x")))
        (AddE (AppE (VarE "square") (FloatE 1.0))
              (AppE (VarE "square") (FloatE 2.0))))
```

We call this kind of sharing λ-sharing because by properly detecting it we can construct an embedded language term where all sub-terms that result from the application of the same lambda share that lambda. This allows us to use Haskell's function application to represent function application in Nikola.

Lack of λ-sharing does not cause any additional work to be done when the embedded language expression is evaluated, but it does lead to an increase in code size. Looking at the definition of blackscholes1 in Listing 1, we see that normcdf would be inlined twice in the first-order representation of the Nikola version of the function if we couldn't detect λ-sharing, substantially increasing the size of the Nikola term for blackscholes1. Our own past experience with Flask (Mainland et al. 2008) indicates that code explosion resulting from loss of λ-sharing can be a serious issue.

The only way we can hope to recover λ-sharing is to gain control over β-reduction. If we could overload application, that would give us the tool we need to get started. Since that is not possible in Haskell, we use a different approach: we will write a family of variadic application functions, vapply, that allow us to control β-reduction.

Consider the expression vapply square. We would like this expression to return a function like square′, where "square" is somehow bound to the proper abstract syntax for the original definition of the embedded square function.

```
square′ :: Exp Float → Exp Float
square′ (E e) = E $ AppE (VarE "square") e
```

To control β-reduction, we want vapply to perform as follows. The "first" time vapply is passed square, it should reify the function square and add a top-level binding for its reified definition to the abstract syntax. It then returns a new function, which we call square′ in this example, that takes an Exp Float and returns abstract syntax for the *application* of the new top-level binding it just created for square to the Exp Float argument. Subsequent applications of vapply to square re-use the previously generated binding.

Note that the type we have given for vapply is not monadic; how can vapply then tell when it is "first" applied to square and keep track of top-level bindings? The answer is that we delay the monadic portion of vapply's responsibilities until reification. This requires altering the DExp data type to add a new constructor (recall that R is the monad we use to maintain the state needed for reification):

```
data DExp = DelayedE (R DExp)
          | ...
```

Using this new definition, vapply constructs a DelayedE value whose argument is the monadic action that either creates or finds the binding for square and then generates an application of this value to the Exp Float that is the argument to square.

We use the same type class technique from the previous section to generate our family of functions, vapply, at each of the types:

(Exp $a_1 \rightarrow \ldots \rightarrow$ Exp a_n) $\rightarrow$ Exp $a_1 \rightarrow \ldots \rightarrow$ Exp a_n

```
class (ReifiableFun a b) ⇒ VApply a b c d | a → c,
                                             b → d,
                                             c → a,
                                             d → b where
    vapply :: (a → b) → c → d
instance (Typeable a, Typeable b)
  ⇒ VApply (Exp a) (Exp b) (Exp a) (Exp b) where
    vapply = ...
instance (Typeable a, VApply b c d e)
  ⇒ VApply (Exp a) (b → c) (Exp a) (d → e) where
    vapply = ...
```

Note that we do not have to use vapply every time we write an application and want to avoid inlining. Instead, the vapply can occur in the definition of square, in which case square will never be inlined:

```
square :: Exp Float → Exp Float
square = vapply $ λx → x ∗ x
```

3.3 Alternative embedding strategies

Encoding our first-order representation of Nikola using GADTs would allow us to leverage Haskell's type system to help guarantee that our manipulation of abstract syntax maintains type safety of embedded language terms. Atkey et al. (2009) describe how to convert HOAS to a GADT-based first-order representation. However, using GADTs would substantially complicate the implementation because it requires the use of de Bruijn indices, coercion rules for weakening and contraction, etc. The only consumer of the first-order representation is our compiler, and the difficult part of the compiler is the portion that translates our first-order representation to CUDA which would not be helped by the use of GADTs in any case. We therefore leave to future work a GADT-based implementation of Nikola. We also note that our solution to the λ-sharing problem is applicable to any method for converting HOAS to a first-order representation, including those based on GADTs. Furthermore, the techniques for compiling and calling Nikola functions that we describe in Section 4 and Section 5 also apply in the GADT setting.

3.4 Summary

We can now complete our translation of the pure Haskell function in Listing 1 to a corresponding Nikola implementation, shown in Listing 2 (we have elided some unchanged code). Beyond altering a few type annotations, rewriting a conditional and using vapply to control inlining, the Nikola code is identical to the Haskell code. The up side to making these modifications is that the blackscholes function can now be run on a GPU.

The machinery in this section, including our new technique for detecting λ-sharing, allows us all the advantages of embedding Nikola, e.g., not having to write a custom parser or implement a type system, while retaining the ability to produce the same first-order representations of Nikola programs that a custom parser would produce. Specifically, we can use Haskell let bindings and application to represent Nikola let bindings and application and thereby avoid the duplicated work and code explosion that result when let-sharing and λ-sharing remain unobserved. However, this only solves the embedding problem; the remaining sections show how to take the first-order representation our embedding provides, compile it to GPU binary code, and make it callable from Haskell.

4. Translating Nikola to CUDA

Thus far, we've seen how to embed a first-order language such as Nikola into Haskell in a convenient way while still maintaining both let-sharing and λ-sharing. In this section, we discuss the rest of the implementation of Nikola including size inference and loop translation. We use GHC's quasiquoting feature (Mainland 2007) in translating Nikola to CUDA. This feature provides the syntactic convenience of writing CUDA code directly in a Haskell program while maintaining the benefits of using a data type representation—in contrast to a string representation—for CUDA; the GHC front-end takes care of converting CUDA syntax to nested data constructor applications. Quasiquoting also makes it easy to splice together CUDA program fragments. The translation is simplified by the fact that Nikola is first-order except for the small set of higher-order functions that are made explicit in the language.

```
blackscholes :: Exp (Vector Float)    -- Stock prices
             → Exp (Vector Float)     -- Option strikes
             → Exp (Vector Float)     -- Option years
             → Exp (Vector Float)
blackscholes ss xs ts =
    zipWith3 (λs x t → blackscholes1 s x t r v)
      ss xs ts
  where
    ...

blackscholes1 :: Exp Float   -- Stock price
              → Exp Float    -- Option strike
              → Exp Float    -- Option years
              → Exp Float    -- Riskless rate
              → Exp Float    -- Volatility rate
              → Exp Float
blackscholes1 s x t r v =
    s ∗ normcdf d1 − x ∗ exp (−r ∗ t) ∗ normcdf d2
  where
    ...

normcdf :: Exp Float → Exp Float
normcdf = vapply $ λx → (x .<. 0) ? (1 − w, w)
  where
    ...

horner coeff x = foldr1 madd coeff
  where
    madd a b = b ∗ x + a
```

Listing 2: Black-Scholes call option valuation in Nikola

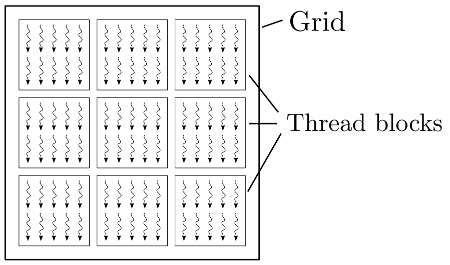


Figure 1: **CUDA thread hierarchy**

CUDA itself has a single instruction, multiple data (SIMD) style execution model in which 1, 2 or 3-dimensional arrays of threads forming *thread blocks* are grouped into a 1 or 2-dimensional *grid*, as shown in Figure 1. An $m \times n$ grid of $k \times l$ thread blocks will lead to $m \cdot n$ invocations of a thread block, each of which runs $k \cdot l$ threads; the $m \cdot n$ thread blocks are executed in an unspecified order, and the threads in each thread block are run in groups whose size depends on the number of threads the CUDA-enabled device on which they are running can support. Each thread has access to (constant) variables that specify its index within its thread block and the index of its thread block in the grid. This provides a natural way to express (nested) parallel loops. All threads execute the same *kernel*.

$$
\begin{array}{lcl}
\tau & ::= & \mathsf{float} \\
\nu & ::= & |\mathsf{i}| \mid \mathsf{min}(\nu_1, \ldots, \nu_k) \\
\rho & ::= & \tau \\
 & | & \mathsf{Vec}\ \nu\ \tau \\
 & | & \{\rho_1, \ldots, \rho_n\} \to \rho \\
e & ::= & i \\
 & | & x \\
 & | & \mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2 \\
 & | & \lambda(x_1 :: \rho_1) \ldots (x_n :: \rho_n)\ .\ e \\
 & | & e\ x_1 \ldots x_n \\
 & | & e_1 + e_2 \\
 & | & e_1 * e_2 \\
 & | & \mathsf{map} \\
 & | & \mathsf{zipWith}
\end{array}
$$

Figure 2: **Nikola language**

$$
\frac{}{\Gamma \vdash i : \mathsf{float}}\ \textsc{Const}
\qquad
\frac{}{\Gamma, x : \rho \vdash x : \rho}\ \textsc{Var}
$$

$$
\frac{\Gamma \vdash e_1 : \rho_1 \qquad \Gamma, x : \rho_1 \vdash e_2 : \rho_2}{\Gamma \vdash \mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2 : \rho_2}\ \textsc{Let}
$$

$$
\frac{x_1 : \rho_1, \ldots, x_n : \rho_n \vdash e : \rho \qquad \text{if } \rho_i = \mathsf{Vec}\ \nu\ \tau, \text{ then } \nu = |\mathsf{i}|}{\Gamma \vdash \lambda(x_1 :: \rho_1) \ldots (x_n :: \rho_n)\ .\ e : \{\rho_1, \ldots, \rho_n\} \to \rho}\ \textsc{Abs}
$$

$$
\frac{\Gamma \vdash e_1 : \bar{\varphi}(\rho_1)\ \ldots\ \Gamma \vdash e_n : \bar{\varphi}(\rho_n) \qquad \Gamma \vdash f : \{\rho_1, \ldots, \rho_n\} \to \rho}{\Gamma \vdash f\ e_1 \ldots e_n : \bar{\varphi}(\rho)}\ \textsc{App}
$$

$$
\frac{\Gamma \vdash e_1 : \mathsf{float} \qquad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 + e_2 : \mathsf{float}}\ \textsc{Plus}
\qquad
\frac{\Gamma \vdash e_1 : \mathsf{float} \qquad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 * e_2 : \mathsf{float}}\ \textsc{Mul}
$$

$$
\frac{}{\Gamma \vdash \mathsf{map} : \{\tau_1 \to \tau_2, \mathsf{Vec}\ |\mathsf{2}|\ \tau_1\} \to \mathsf{Vec}\ |\mathsf{2}|\ \tau_2}\ \textsc{Map}
$$

$$
\frac{}{\Gamma \vdash \mathsf{zipWith} : \{\{\tau_1, \tau_2\} \to \tau, \mathsf{Vec}\ |\mathsf{2}|\ \tau_1, \mathsf{Vec}\ |\mathsf{3}|\ \tau_2\} \to \mathsf{Vec}\ \mathsf{min}(|\mathsf{2}|, |\mathsf{3}|)\ \tau}\ \textsc{Zip}
$$

Figure 3: **Nikola type system** *See Section 4.2 for a description of the $\bar{\varphi}$ function.*

The execution model underlying CUDA imposes a number of additional severe constraints, making it a challenging target for compilation. Recursion is disallowed, as are function pointers; any memory used by threads must be allocated before kernel invocation; and the GPU cannot access arbitrary host CPU memory, so any data used or returned by a GPU computation must be explicitly transferred between the host CPU and GPU.

4.1 The Nikola Language

A simplified version of the Nikola language is shown in Figure 2, and its type system in Figure 3. The language is fairly standard, but has a few features that allow it to be compiled to CUDA code. The full language includes additional scalar and vector operators, but the subset presented here is sufficient to demonstrate the interesting aspects of translation to CUDA.

The ABS rule requires that the body of a function be typeable in a type environment where only the arguments to the function are bound, i.e., every time a lambda is encountered, the type environment is "wiped clean." This disallows closures and partial applications. A Nikola function can still return another function, but the returned function may not close over variables bound by the outer function's lambda. For example, the Nikola function $\lambda(x :: float)\ .\ \lambda(y :: float)\ .\ x+y$ is not typeable because $x+y$ cannot be typed in the type environment $y : float$.

The only higher-order functions available in the language are "baked in," so the memory used by an expression can always bounded above by a function of the memory used by its subexpressions. In the simplified version of the language presented here, this bound is strict. These features allow us to pre-allocate all memory a computation needs before it is invoked, as described in Section 4.2. Baking-in higher-order functions that operate on vectors permits us to compile these operations to efficient parallel code; we detail this process in Section 4.3.

4.2 Size inference

We guarantee that we can bound the memory requirements of a Nikola function by avoiding general looping constructs and restricting all vector operations to those for which a static bound can be calculated. Vector variables bound by a lambda are assigned a vector type that includes a tag indicating the index of the bound variable in the (ordered) set of variables bound by the lambda (see the ABS rule). The typing rule for each vector operation calculates a bound for the result of the operation that is a function of the sizes of its inputs (see MAP and ZIP). In practice this is only an upper bound, e.g., consider a filter operation on vectors. Determining the size of the outputs is in general undecidable until run time, but bounding the size of the outputs can be done statically. This upper bound is sufficient for pre-calculating the GPU memory needed by a Nikola function. For the subset of the language shown here, the bound is exact.

The size of the memory needed by a Nikola function is a mathematical function of the sizes of its inputs. For example, map needs as much memory for its output as is occupied by its second argument, so the size index to the right of the arrow in the type of map is $|\mathsf{2}|$. However, every time map is applied, we must substitute the size index of the actual second argument for $|\mathsf{2}|$. This is the purpose of the function φ. From an algorithmic point of view, if the i^{th} argument (e_i in rule APP) in an application has type $\mathsf{Vec}\ \nu_i\ \tau$, then we take $\varphi(|\mathsf{i}|) = \nu_i$. The function $\bar{\varphi}$ in Figure 3 is the lifting of φ to types; it rewrites every size index using φ. Note that the type inference rules guarantee that $|\mathsf{i}|$ can appear in a type-level size only if the lexically enclosing lambda has at least i arguments.

Our size inference is deliberately restricted so that a Nikola function can be compiled to a single GPU kernel. This rules out many functions one may wish to write. For example, consider a vector operator replicate such that replicate n x returns a vector of size n where each element is x. This might seem like as easy operation to add, but what if the programmer now writes replicate (fold (+) xs) 1, where xs is a vector; how can we perform size inference on this expression?

Because we must allocate all memory needed by a CUDA function before we call it, compiling this expression would require that we either perform the initial fold on the CPU before executing replicate on the GPU, or that we compile the expression to a sequence of multiple calls to the GPU. In general, this kind of dependency requires either moving computation off the GPU and back to the CPU, or intelligently splitting computation between the CPU and GPU to properly interleave memory operations and GPU operations. We believe the former is a non-starter because it nullifies much of the benefit of using Nikola in the first place, and the latter is more complexity that we wished to take on here, so our size inference simply disallows this kind of dependency. We leave as future work enhancing the compiler so that it can automate the CPU/GPU interleaving required for this sort of operation.

4.3 Loop translation

As described earlier, the CUDA execution model allows implicit parallelism to be nested to a depth of (in practice) at most two. In this section we describe how to map Nikola's parallel looping constructs onto the model provided by CUDA.

The simplest case is a top-level Nikola function whose entire body consist of a single loop over a vector of length n. We decompose this kind of computation by picking a static width w for the thread block and then using an execution grid of size $\lceil n/w \rceil$ (with appropriate guards on the loop body to avoid overrun) to invoking the kernel. The translated code does not contain an outer loop construct because the loop is implicit in the grid and thread block dimensions that are specified as part of kernel invocation.

When a kernel contains sequentially executed parallel constructs, we cannot use this simple strategy because in general it would require that all loops somehow be fused into a single, massive loop. Instead, we use strip-mining to parallelize each loop. Because some CUDA-enabled devices can run more than one thread block at a time, this can be much less efficient than decomposing a parallel loop to run on a grid, but it allows us to handle the more general case. Assuming that the width of the thread block is given as `threadBlockWidth` and the index of a thread in this block by `threadIndex`, strip-mining transforms a loop of the form

```
for (int i = 0; i < n; ++i) {
    ...
}
```

to a loop of the form

```
for (int is = 0; is < n; is += threadBlockWidth) {
    const int i = is + threadIndex;
    if (i < n) {
        ...
    }
}
```

A third possibility is that looping constructs are nested. Because the same thread index cannot be used to strip-mine multiple nested loops, any nested loop is translated to a standard `for` loop. All of these cases are abstracted away from the bulk of the translator through the use of a single combinator that wraps its body in the proper CUDA nesting construct depending on whether the looping construct occurs as the body of a top-level function and on whether there is an available loop index. This combinator also computes the mapping from a function's arguments' sizes to the proper thread block and grid dimensions for invoking the kernel.

5. Compiling and calling a Nikola function

Once a Nikola function is translated to CUDA, it is compiled to binary object code by invoking `nvcc`, the NVIDIA CUDA compiler. Calling a CUDA function requires manually copying its arguments to GPU memory and explicitly building a call stack. All state related to calling a compiled CUDA function, including a handle to the compiled function, the call stack and a list of temporary allocations needed by the kernel, is kept in a data structure of type ExState. The entire process of converting a Nikola function written using HOAS to an ExState is handled by the function reifyAndCompile. We maintain type information about the compiled version of the function by wrapping an ExState in a phantom type using the type constructor F:

```
newtype F a = F {unF :: ExState}
```

The type of reifyAndCompile is therefore

```
reifyAndCompile :: ReifiableFun a b
                ⇒ (a → b)
                → IO (F (a → b))
```

For a type to be capable of marshalling to the GPU, it must be an instance of the Embeddable type class. The instance for a type provides a small amount of glue code to allow passing a value of the type to and from a CUDA kernel. Calling a CUDA function requires manually copying its arguments to GPU memory and explicitly building a call. Any additional memory used by a kernel is allocated prior to its invocation and passed as additional arguments, as is space needed for its results. This is all done using the driver API. After building the call stack, the kernel is invoked. When it returns, the results are copied back to CPU memory and all memory previously allocated for the kernel is released. Building the call stack and invoking the kernel is done in the Ex monad using the ExState described previously; running this computation yields a result in the IO monad and is done using evalEx.

```
evalEx :: ExState → Ex a → IO a
```

5.1 Generating GPU binary code

Given a Nikola function of type $\text{Exp } a_1 \to \ldots \to \text{Exp } a_{n-1} \to \text{Exp } a_n$, the compilation process guarantees that it compiles to a Haskell function of type $a_1 \to \ldots \to a_{n-1} \to \text{IO } a_n$. This invariant isn't witnessed by a Haskell type—indeed it cannot be—so we must reason outside the type system about the compilation process itself to conclude that the invariant holds. Assuming that it does hold, we can ensure type safety when calling GPU object code.

Our goal is to define a family of functions, call, that converts a Nikola function (written using HOAS) into a callable Haskell function. Calling a Nikola function requires reification (to convert from HOAS to a first-order representation), compilation (to convert the first-order representation to CUDA and then compile the CUDA to object code), call stack construction, memory allocation, function invocation, and cleanup. We use type classes to define the family of functions call at each type:

$$(\text{Exp } a_1 \to \ldots \to \text{Exp } a_{n-1} \to \text{Exp } a_n) \to a_1 \to \ldots \to a_{n-1} \to \text{IO } a_n$$

The straightforward way to handle building up the call stack while retaining the ability to build our instances inductively is to use continuation-passing style. The auxiliary callk takes a continuation that pushes the previously seen arguments on the stack, invokes the function and performs cleanup. There will be one call to callk for each argument; at each step in the call chain, the monadic action that will push arguments onto the CUDA call stack is built up. The final call to callk will perform this monadic action, creating the call stack, and invoke the kernel.

```
class Callable a b c | a b → c where
  call :: (a → b) → c
  call f = callk (reifyAndCompile f) id
  callk :: IO (F (a → b)) → (∀a.Ex a → Ex a) → c
```

As we can see, the base case performs the actual compilation, invokes the monad action that pushes all arguments other than the first onto the stack, pushes the first argument onto the stack, and then executes the kernel and cleans up.

```
instance (Embeddable a, Embeddable b) ⇒
  Callable (Exp a) (Exp b) (a → IO b) where
    callk compileF pushArgs = λx → do
      F f ← compileF
      evalEx f $
        pushArgs $
        pushArg x $
        launchKernel $
        returnResult
```

The inductive case requires casting the compilation action so that it has the proper type; this type drives the type class resolution algorithm to choose the correct instance of Callable for the nested call to callk.

```
instance (Embeddable a,
          Reifiable (Exp a) (b → c),
          Callable b c d) ⇒
  Callable (Exp a) (b → c) (a → d) where
    callk compileF pushArgs =
        λx → callk compileF′ (pushArgs ∘ pushArg x)
      where
        compileF′ :: IO (F (b → c))
        compileF′ = castF <$> compileF

        castF :: F a → F b
        castF = F ∘ unF
```

5.2 Compile-time vs. run-time compilation

The astute reader will notice one glaring problem with the code in the previous section; every time a Nikola function is called using call, it will be re-compiled! From the perspective of the caller, Nikola functions are pure, so it makes sense for us to treat them as such. This requires the use of unsafePerformIO, but referential transparency is maintained; note that in the rewritten version the first argument to callk is no longer a monadic action.

```
class Compilable a b c | a b → c where
    compile :: (a → b) → c
    compile f =
      compilek (unsafePerformIO (reifyAndCompile f))
               id

    compilek :: F (a → b) → (∀a.Ex a → Ex a) → c
instance (Embeddable a, Embeddable b) ⇒
  Compilable (Exp a) (Exp b) (a → b) where
    compilek f pushArgs = λx →
        unsafePerformIO $
        evalEx (unF f) $
        pushArgs $
        withArg x $
        launchKernel $
        returnResult
instance (Embeddable a,
          Reifiable (Exp a) (b → c),
          Compilable b c d) ⇒
  Compilable (Exp a) (b → c) (a → d) where
    compilek f pushArgs =
        λx → compilek f (pushArgs ∘ withArg x)
      where
        f′ :: F (b → c)
        f′ = castF f
```

This alternative implementation allows us to use Nikola functions in pure code. It has the added benefit that given a Nikola function f, compilation is only performed once for f if we create a top-level binding g = compile f and then use g in Haskell code. However, even this approach requires that f is compiled *every time the program is exececuted*. What we would really like it to compile f once, at the same time the rest of the Haskell code is compiled.

Template Haskell (Sheard and Peyton Jones 2002) provides exactly the features we need. The CUDA object code produced as the result of compilation is reified in Template Haskell as a string constant. The first time the compiled function is invoked, this object code is loaded into the GPU and the rest of the function's execution state is constructed. The glue function that builds the call stack is generated using the same technique we saw before. We can create a top-level binding for a pre-compiled Nikola function f as follows:

```
g = $ (compileTH f)
```

As we show in Section 6, compilation of each function takes about 500ms. By using Template Haskell, every time we run a Haskell program that embeds Nikola functions, we save approximately 500ms for every Nikola function that is used at least once by the program.

6. Evaluation

In this section we use two benchmarks, Black-Scholes call option evaluation and radix sort, to help characterize the performance trade-offs among various implementation platforms for the same algorithm. All benchmarks were done on a machine with two quad-core (64-bit) E5462 Xeon processors running at 2.80GHz, 16GB of CPU RAM, and 4 NVIDIA Tesla T10 GPUs. All Haskell code, including the `vector` library, was compiled with optimization [1].

For comparison, we wrote two implementations of the Black-Scholes method. The first uses efficient unboxed vectors provided by the Haskell `vector` library; we feel at the time of writing that this library provides best-in-class performance for Haskell array manipulation code. The second version was implemented in Nikola. The code for both implementations is identical to what appears in Section 3. We also used Nikola's ability to embed CUDA code directly in a Haskell program as a callable function to appropriate the hand-written implementation from the SDK. By embedding the SDK version directly, we can compare the quality of the CUDA code Nikola generates in isolation from factors such as function call overhead.

Figure 4 shows four runs of the Black-Scholes function where the run-time is a function of the size of the input. The Haskell version using the `vector` library and the CUDA SDK version were each run once. The Nikola version was run twice; in one run, we compiled the Nikola version at Haskell compile-time, and in the other we compiled the function *every time it was called*. The latter gives an indication of the overhead imposed by an embedded DSL compiler that always specializes a function to its argument. In our runs, this overhead amounted to about a 500ms per function call. A lot of computation can occur in 500ms, and it takes a large number of inputs to amortize this cost. There is one source of overhead than cannot be avoided when running on a GPU: the time needed to shuffle data back-and-forth from the main processor. This cost is quickly offset by the performance increase obtained by moving computation onto the GPU. The performance of the pre-compiled Nikola version is identical to the performance of the hand-written CUDA version.

Implementing a radix sort on a GPU requires substantially more effort than Black-Scholes option evaluation because it utilizes more than a single parallel loop. Our Nikola version is a fairly direct translation of the algorithm given by Blelloch (1990), which requires executing multiple passes over the data. Because of limita-

[1] `-O2 -funbox-strict-fields -fvia-C -optc-O3 -optc-march=core2`

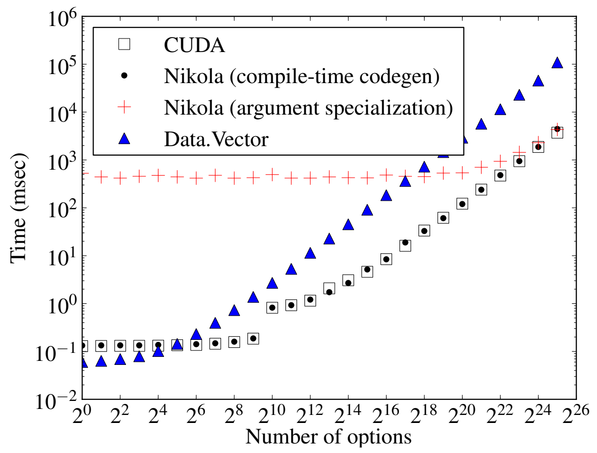


Figure 4: **Performance of Black-Scholes implementations.** *Note that the CUDA and pre-compiled Nikola implementations overlap point-for-point.*

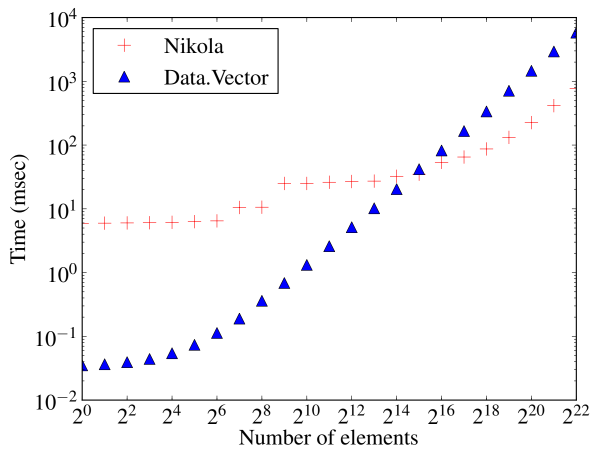


Figure 5: **Performance of radix sort implementations.**

tions of Nikola, this necessitates invoking multiple kernels instead of a single fused kernel. Nonetheless, the Nikola version, generating unoptimized CUDA code, still substantially outperforms a native Haskell version for data sets larger than about 32kB, as shown in Figure 5.

The performance data demonstrates that Nikola can generate code that is as efficient as hand-written CUDA. Admittedly, we have engineered Nikola so that it can be compiled to efficient GPU code, but we make no apology for this—DSLs are, after all, *domain specific*.

7. Summary and Future Work

Nikola provides a lightweight, low-effort way for Haskell programmers to offload computation onto a GPU without leaving Haskell. In some cases, Nikola can outperform a pure Haskell implementation by several orders of magnitude. Programmers retain the benefits of a high-level language while still obtaining many of the performance benefits of low-level CUDA code. Writing a Nikola function is almost as easy as writing an equivalent Haskell function, and Nikola functions can even be compiled at GHC compile time. We obtain all these features without any modification of GHC.

One of the main motivations for doing a deep embedding of a DSL is to avoid having to write a parser and type checker by instead re-using Haskell's syntax and type inference engine. Now that both let- and λ-sharing can be observed, there is even less of a reason to reinvent the wheel for each new domain-specific language since a deep embedding can yield the same first-order abstract syntax—complete with let-bound lambdas—that a custom parser would yield. Our technique for observing λ-sharing is applicable to any DSL that uses HOAS to represent DSL functions.

There are several language enhancements that would make Haskell an even better host for domain-specific languages. If we could overload *all* of Haskell's syntax, including pattern matching and if-then-else expressions, then rewriting a Haskell function as a Nikola function would require at most a change in its type signature. We could increase our confidence in the back-end by assigning better types to the CUDA fragments it constructs; however, even GADTs are not sufficient for capturing the details of C's type system. Extending Haskell's type system so that it could integrate smoothly with the type system of an arbitrary embedded object language—including C-like languages—is an interesting research challenge.

We plan to add additional array operations to Nikola; obvious candidates include matrix and signal processing functions. It would be interesting to see how much of NESL's (Blelloch et al. 1994) functionality we can successfully provide using only an embedding. Another possible avenue of research is integrating Nikola's functionality directly with Data Parallel Haskell (Peyton Jones et al. 2008). Finally, because it presents such a high-level interface, we should be able to easily re-target Nikola's back-end to languages such as OpenCL.

Acknowledgments

This research is supported by the Office of Naval Research under grant number N00014-09-1-0770.

References

Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*, pages 37–48, Edinburgh, Scotland, 2009. ACM.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 174–184, Baltimore, Maryland, United States, 1998. ACM.

Guy E Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

Manuel Chakravarty, Gabriele Keller, and Sean Lee. accelerate, October 2009. URL `http://www.cse.unsw.edu.au/~chak/project/accelerate/`.

Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, pages 62–73. Springer-Verlag, 1999.

Conal Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*, pages 45–56, Snowbird, Utah, USA, 2004. ACM.

Conal Elliott, Sigbjörn Finne, and Oege De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLDI '03)*, page 1–11. ACM, 2003.

Andy Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*, pages 117–128, Edinburgh, Scotland, 2009. ACM.

John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.

Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323—-343, July 1992.

Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA: GPU Run-Time code generation for High-Performance computing. *0911.3456*, November 2009.

Sean Lee, Manuel Chakravarty, Vinod Grover, and Gabriele Keller. GPU kernels as Data-Parallel array computations in haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPAHM 2009)*, 2009.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122, Austin, Texas, United States, 1999. ACM.

Roman Leshchinskiy. vector: Efficient arrays, February 2010. URL `http://hackage.haskell.org/package/vector`.

Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell '07)*, page 73–82, New York, NY, USA, 2007. ACM.

Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, page 335–346, New York, NY, USA, 2008. ACM.

John T. O'Donnell. Generating netlists from executable circuit specifications. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 178–194. Springer-Verlag, 1993.

Izzet Pembeci, Henrik Nilsson, and Gregory Hager. Functional reactive robotics: an exercise in principled integration of domain-specific languages. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 168–179, Pittsburgh, PA, USA, 2002. ACM.

Simon L. Peyton Jones, Roman Leshchinskiy, and Manuel Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In *Programming Languages and Systems*, page 138. 2008.

F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation (PLDI '88)*, pages 199–208, Atlanta, Georgia, United States, 1988. ACM.

Tim Sheard and Simon L. Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*, pages 1–16, Pittsburgh, Pennsylvania, 2002. ACM.

Joel Svensson, Koen Claessen, and Mary Sheeran. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Proceedings of 20th International Symposium on the Implementation and Application of Functional Languages (IFL '08)*, Hatfield, UK, 2008.

Joel Svensson, Koen Claessen, and Mary Sheeran. GPGPU kernel implementation using an embedded language: a status report. Technical Report 2010:01, Chalmers University of Technology, January 2010.

Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pages 203–217, Amsterdam, The Netherlands, 1997. ACM.