Staged compilation of tensor expressions

Marco Zocca
UnfoldML
marco.zocca@unfoldml.com

Abstract

We present our current progress towards a metaprogramming framework for tensor expressions embedded in Haskell; the system offers a high-level syntax for linear algebra, and generates specialized source code with type-level dimension annotations.

1 Introduction

The design of domain-specific languages (DSL) for numerical computing is characterized by a tension between performance and expressiveness [1]. Traditional numerical libraries expose a number of highly optimized “kernels” (for instance, the linear combination $Ax + b$); a user is expected to program against this interface in order to achieve high performance. This approach is only limited by the availability of such kernels for the operations and data formats of interest.

Recently, generative programming techniques have been applied to synthesize specialized code from high-level specifications by leveraging statically known data (e.g. vector dimensions and types of the input expression). Multi-stage compilation (or staging) is a form of metaprogramming in which one or more code synthesis phases take place, each evaluating at least in part the output of the previous one and emitting source code and metadata that will be consumed downstream [2–6].

Contribution In this work we describe a two-stage compiler for a high-level tensor DSL embedded in Haskell, which aims to strike a balance between user-friendliness and high performance. In particular, the multi-stage approach makes it possible to synthesize code and corresponding specialized type signatures from a concise mathematical specification, circumventing certain limitations of the host language.
2 Compiler pipeline

**AST analysis**  Users define the expression to be compiled using the grammar shown in Figure 1, for instance, `contract (1) (0) x y` denotes a contraction of variables `x` and `y` over the second and first index respectively, and it may correspond to the matrix product `X_i Y_j` if both operands `x` and `y` have order 2. The concrete syntax looks very similar to the abstract one, since it is encoded as a Haskell sum type.

```
⟨e⟩ ::= T_n ⟨sh⟩
  | contract ⟨ix⟩ ⟨ix⟩ ⟨e⟩ ⟨e⟩
  | binary ⟨op2⟩ ⟨e⟩ ⟨e⟩
  | unary ⟨op1⟩ ⟨e⟩
⟨op2⟩ ::= + | ◦
⟨op1⟩ ::= scale | exp | log
```

Figure 1: Abstract syntax of tensor expressions `e`. Each tensor constant `T` is bound to a distinct name `n`, `op1` is the type of unary componentwise operations (e.g. scaling by a real coefficient, etc.), whereas `op2` denotes binary componentwise operations (e.g. sum and product). Tensor shapes `sh` and contraction multi-indices `ix` are natural number tuples.

After checking its dimensional consistency, a number of structural operations are performed on the expression tree. First, subexpressions are recovered from the abstract syntax tree (AST) by hash-consing, which reconstructs the dataflow graph\[7\]. Afterwards, contraction nodes are decorated with stride data structures which map tensor indices to the underlying memory layout of the variable (e.g. specifying that the fibers `3 × 4 × 5` tensor to be reduced over its second index are stored every 3 elements).

Given the expression DAG, we order the internal nodes into a min-heap according to their distance to the leaves, which then induces the expression reduction order (i.e. from the bound variables to the result).

**Memory layout**  All operands are stored in memory as 1D unboxed arrays, as implemented by the `vector` library. This library provides implicit fusion of intermediate mapping and reduction operations over contiguous ranges, which can decrease memory pressure and improve performance.

The tensor storage type `T` is decorated with dimension annotations, implemented with a type-level list of naturals, as shown in Figure 2.

```
-- | Dense tensor
data T sh a = T { tsShape :: SDim sh , tsData :: Vector a }
data SDim :: [Nat] -> * where
  SNil :: SDim '
SCons :: KnownNat n => Proxy n -> SDim ns -> SDim (n ': ns)
```

Figure 2: In-memory operands with type-checked dimension annotations

**Code and type signature generation**  The Q monad (for quotation, implemented in the `template-haskell` package) provides an effect system for creating unique names (analogous to Lisp’s “gensym”), as well as the syntactic marker separating our compiler stages. In the code generation stage we employ a simple monad transformer type over `Q` which adds safe mutation of code fragments and read-only access to the expression DAG.

The expression node heap from the AST analysis phase is popped until empty, and code corresponding to a `let` declaration is created and inserted into a hashmap; the last node represents the expression value, and will be returned by reference in the body of a `let` (whereas the internal nodes are bound as clauses).

Template Haskell also permits the generation of types, including those having existentially quantified type variables and type-level literals such as strings and natural numbers. In our case, this lets us specialize the signature of the generated bindings by adding mentions of the `T` tensor type as well as dimension annotations computed from the AST itself. This can be seen as a convenient way of circumventing part of the typesystem that don’t support fully dependent-typing techniques (e.g. computing specific functions of types).
3 Example session

Here we show a typical interaction with the system. The expression in this example is the action of a $2 \times 3$ matrix on a 3-vector. The user declares their expression in a separate module, using the provided combinators, and compiles it elsewhere (the GHC “stage restriction” prevents code from different stages to run in the same module).

Figure 3 shows the source code resulting from the compile step: its main features are the inlined inner products of the rows of the first operand with the second operand, as well as the packing and unpacking with the T type that provides type-level dimension annotations.

```
-- module Expressions.hs

testMatVec = do
  aa <- matrix 2 3
  v <- vector 3
  let y = contract 1 0 aa v
  pure y

-- module Program.hs

{-# language TemplateHaskell #-}

$compile testMatVec "fMatVec"
```

Figure 3: Left: user expression. Right: once module Program is loaded, a new binding fMatVec appears in scope, with type shown in Figure 4.

```
fMatVec ::
  ∀ a. (RealFloat a, Unbox a) ⇒ T '[2, 3] a → T '[3] a → T '[2] a
fMatVec t_0 t_1
  = let
      x_1 = tsData t_1
      x_0 = tsData t_0
      v_result
          = generate 2 $ \ i →
            (x_0 (mod i 2 + 0)) * (x_1 i)
            + (x_0 (mod i 2 + 2)) * (x_1 i)
            + (x_0 (mod i 2 + 4)) * (x_1 i)
            + 0
      in
          (T ((SCons (Proxy :: Proxy 2)) SNil)) v_result
```

Figure 4: Code produced by the meta-code shown in Figure 3 desugared and with infix operators. Juxtaposition (e.g. x_1 i) here denotes indexing into a vector. This generated source code is only visible to the user by passing the -ddump-splices GHC flag.
4 Related work

Array compilers and metaprogramming Generative programming has a history of successes in self-tuning numerical libraries such as ATLAS [14] and FFTW [15]. Later research produced whole-program compilers that optimize parallelism and arithmetic intensity [16], adapt deep learning workloads to various hardware backends or generic compiler IR [17–20], and offer a mathematically-intuitive API for tensor expressions while retaining the performance of hand-tuned kernels [8]. Similarly to other polyhedral compilers [21], our DSL is limited to modeling the static control part (“SCoP”) of a numerical program, and it overlaps in scope with TACO [8], with the major difference of being embedded in a declarative language.

5 Discussion

In this paper we have briefly described a metaprogramming environment that transforms symbolic specifications of linear algebra programs into sequences of imperative loops and reductions with statically-typed dimensions. The main rationale for this approach is to extend a preexisting, widespread language (Haskell in this case) with tensor computing capabilities, rather than producing a radically new domain-specific language. As such, further work will aim at proving the correctness of this system and improving the ergonomics of the surface language.

Some authors (e.g. [20, 23]) have shown that transforming memory layout can improve overall arithmetic efficiency of tensor reduction sequences, even though picking the optimal such schedule is NP-hard [24], which motivates the use of approximations and heuristics at the expense of compiler complexity. Others (e.g. [19]) have demonstrated user-facing schedule combinators in their compiler API, but this is beyond the scope of the present work.

Hardware support is another important aspect; since the one we present is a Haskell metaprogramming environment, it only targets the hardware backends that are natively supported by the GHC compiler (i.e. no GPUs or FPGAs at present).

6 Acknowledgements

The author is grateful to Jamie H. Willis and Oleg Kiselyov for helpful discussions, and to the anonymous reviewers whose feedback greatly improved this manuscript.
References


A  Typed Template Haskell

An early version of this work used typed template Haskell for code generation, [9], a type-checked extension to template Haskell (TH) [10] which provides syntactic support for quoting and splicing (“anti-quoting”). In practice this means that we can compose target programs using higher-order combinators (e.g. those shown in Figure 5), resting assured that the resulting code will be well-formed. This approach to compiling embedded DSLs has been demonstrated e.g. in [11] and recently applied to parser combinators [12] and extensible algebraic datatypes [13].

In typed TH, values of type $\text{Code } a$ denote program fragments producing values of type $a$, whereas quoting and splicing are here denoted with $⟦·⟧$ and $σ(.)$ respectively.

```
-- Function abstraction
lam :: (Code x → Code y)
     → Code (x → y)
lam f = $\left[ y → σ(f[ y ]) \right]$    

-- Function application
(>*<) :: Code (a → b)
      → Code a → Code b
f >*< a = $σ(f) σ(a)$

-- | Numeric typeclasses
instance (Num a, Lift a) ⇒ Num (Code a) where
  Code x + Code y = Code $σ(x) + σ(y)$

instance (Floating a, Lift a) ⇒ Floating (Code a) where
  exp (Code x) = Code $σ(exp x)$
```

Figure 5: Left: Typed TH combinators for abstraction and $β$-reduction. Right: Sample implementations of numeric typeclasses for Code.

The initial idea behind this paper was to provide a type-safe user-facing DSL based on $\text{Code}$ combinators, which would then be compiled into efficient array programs. Unfortunately this approach fell short because GHC doesn’t currently support impredicative polymorphism [22], i.e. type variables cannot be instantiated with polymorphic types. More specifically, precise, dependently-typed signatures such as $\text{Code (∀ n. KnownNat n ⇒ Vec n a → Vec n a → a)}$ indicating binary operations on vectors of statically-known identical length will be rejected.

B  Development environment

The library described in this paper was developed using GHC 8.10.4 together with these additional libraries:

- vector 0.12.3
- template-haskell 2.16