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.

Advances in Programming Languages and Neurosymbolic Systems (AIPLANS), NeurIPS 2021.

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 $X_{ij}Y_{jk}$ 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.

$$\begin{array}{lcllcl}
\langle e \rangle & ::= & \mathrm{T}_n \, \langle sh \rangle & & & \\
& | & \text{contract} \, \langle ix \rangle \, \langle ix \rangle \, \langle e \rangle \, \langle e \rangle & & & \\
& | & \text{binary} \, \langle op2 \rangle \, \langle e \rangle \, \langle e \rangle & \langle op2 \rangle & ::= & + \mid \circ \\
& | & \text{unary} \, \langle op1 \rangle \, \langle e \rangle & \langle op1 \rangle & ::= & \text{scale} \, \mathbb{R} \mid \exp \mid \log
\end{array}$$

Figure 1: Abstract syntax of tensor expressions e. Each tensor constant T is bound to a distinct name n, op_1 is the type of unary componentwise operations (e.g. scaling by a real coefficient, etc.), whereas op_2 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, ontraction 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 {                data SDim :: [Nat] → * where
  tsShape :: SDim sh ,             SNil :: SDim '[]
  tsData :: Vector a  }            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 2×3 matrix on a 3-vector. The user declares their expression in a separate module, using the provided combinators, and `compile`s it elsewhere (the GHC "stage restriction" prevents code from different stages to run in the same module).

Figure 4 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.

```haskell
-- module Expressions.hs
testMatVec = do
  aa <- matrix 2 3
  v <- vector 3
  let y = contract 1 0 aa v
  pure y
```

```haskell
-- 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.

```haskell
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

[1] O. Kiselyov, “Reconciling abstraction with high performance: A MetaOCaml approach,” *Foundations and Trends in Programming Languages*, vol. 5, no. 1, pp. 1–101, 2018.

[2] W. Taha and T. Sheard, “Multi-stage programming with explicit annotations,” in *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ser. PEPM ’97, 1997, 203–217.

[3] O. Kiselyov, K. N. Swadi, and W. Taha, “A methodology for generating verified combinatorial circuits,” in *Proceedings of the 4th ACM International Conference on Embedded Software*, ser. EMSOFT ’04, 2004, 249–258.

[4] G. Belter *et al.*, “Automating the generation of composed linear algebra kernels,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC ’09, 2009.

[5] B. Aktemur *et al.*, “Shonan challenge for generative programming: Short position paper,” in *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM ’13, 2013, 147–154.

[6] Y. Kameyama, O. Kiselyov, and C.-C. Shan, “Combinators for impure yet hygienic code generation,” *Science of Computer Programming*, vol. 112, pp. 120–144, 2015, Selected and extended papers from Partial Evaluation and Program Manipulation 2014.

[7] O. Kiselyov, “Implementing explicit and finding implicit sharing in embedded DSLs,” *Electronic Proceedings in Theoretical Computer Science*, vol. 66, 210–225, 2011.

[14] R. C. Whaley and J. J. Dongarra, “Automatically tuned linear algebra software,” in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC ’98, 1998, 1–27.

[15] M. Frigo and S. Johnson, “FFTW: An adaptive software architecture for the FFT,” in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP ’98*, vol. 3, 1998, 1381–1384 vol.3.

[16] J. Ragan-Kelley *et al.*, “Decoupling algorithms from schedules for easy optimization of image processing pipelines,” *ACM Trans. Graph.*, vol. 31, no. 4, Jul. 2012.

[17] M. Steuwer *et al.*, “Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code,” in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015, 2015, 205–217.

[18] T. Chen *et al.*, “TVM : An automated end-to-end optimizing compiler for deep learning,” in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18, 2018, 579–594.

[19] R. Baghdadi *et al.*, *Tiramisu: A polyhedral compiler for expressing fast and portable code*, 2018. arXiv: `1804.10694 [cs.PL]`.

[20] N. Vasilache *et al.*, *Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions*, 2018. arXiv: `1802.04730 [cs.PL]`.

[8] F. Kjolstad *et al.*, “The tensor algebra compiler,” *Proc. ACM Program. Lang.*, vol. 1, 77:1–77:29, Oct. 2017.

[21] T. Grosser, A. Groesslinger, and C. Lengauer, “Polly — performing polyhedral optimizations on a low-level intermediate representation,” *Parallel Processing Letters*, vol. 22, no. 4, 2012.

[23] G. Baumgartner *et al.*, “Compile-time optimizations for tensor contraction expressions,” in *Proc. of the Int. Workshop on Compilers for Parallel Computers*, 2003.

[24] R. N. C. Pfeifer, J. Haegeman, and F. Verstraete, “Faster identification of optimal contraction sequences for tensor networks,” *Physical Review E*, vol. 90, no. 3, 2014.

[9] G. Mainland, “Explicitly heterogeneous metaprogramming with MetaHaskell,” in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’12, 2012, 311–322.

[10] T. Sheard and S. P. Jones, “Template meta-programming for Haskell,” in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, ser. Haskell ’02, 2002, 1–16.

[11] S. Najd *et al.*, “Everything old is new again: Quoted domain-specific languages,” in *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM ’16, 2016, 25–36.

[12] J. Willis, N. Wu, and M. Pickering, “Staged selective parser combinators,” *Proc. ACM Program. Lang.*, vol. 4, 2020.

[13] M. Pickering, A. Löh, and N. Wu, “Staged sums of products,” in *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2020, 2020, 122–135.

[22] M. Pickering, N. Wu, and C. Kiss, “Multi-stage programs in context,” in *Proceedings of the 12th International Symposium on Haskell*, ACM, 2019, pp. 71–84.

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 `Code a` denote program fragments producing values of type `a`, whereas quoting and splicing are here denoted with ⟦·⟧ and $\sigma(.)$ respectively.

```
-- Function abstraction
lam :: (Code x → Code y)
     → Code (x → y)
lam f = ⟦ \y → σ( f ⟦ y ⟧ ) ⟧

-- 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 ⟦ exp σ(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 `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 *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