Staged compilation of tensor expressions

Anonymous Author(s)
Affiliation
Address
email

Abstract

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

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), a form of partial evaluation known as *staging* [2–5].

Contribution This paper aims to demonstrate staged metaprogramming, as currently supported by the GHC Haskell compiler, to the machine learning community. To do so, we show how array programs can be synthesized from a user-friendly specification using the machinery of typed template Haskell.

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* may correspond to the matrix product $A_{ij}B_{jk}$ if both operands have order 2. The concrete syntax looks very similar to the abstract one, since it is encoded as a Haskell sum type.

After checking dimensional consistency, common subexpressions can be recovered from the abstract syntax tree (AST) by hash-consing, which reconstructs the dataflow graph [6].

Iteration graphs and operator fusion The expression AST is reduced recursively from the leaves in order to build an iteration graph [7], which determines the traversal order of the tensor components and stores one dimension per node. In our system iteration graph nodes also store "free/bound" annotations to track pairs of contracted indices. The graph of a tensor constant is initialized as having all free indices. At each recursion level, the iteration graphs of the branches are merged while

Submitted to 35th Conference on Neural Information Processing Systems (aiplans 2021). Do not distribute.

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

Figure 1: Abstract syntax of tensor expressions e. Each tensor constant T is bound to a distinct name n , op_1 and op_2 denote componentwise operations, tensor shapes sh and contraction multi-indices ix are natural number tuples.

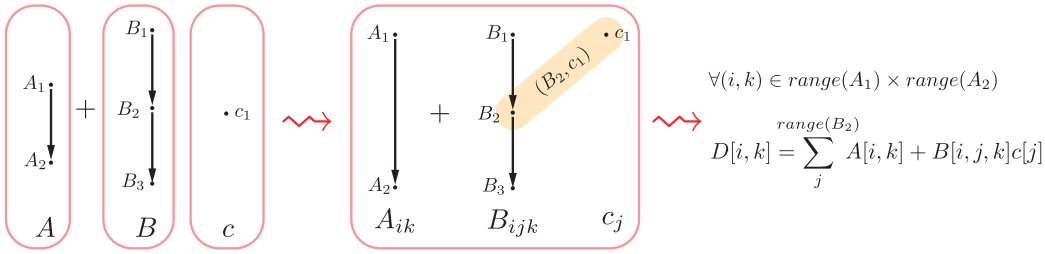

Figure 2: Reduction of the iteration graph of expression $D_{ik} = A_{ik} + B_{ijk}c_j$. If the operand shapes are conformal, the addition of A will be fused with the construction of the result thereby avoiding the allocation of an intermediate array.

preserving the relative ordering of the indices; here, contraction of paired indices results in a "bound" index tuple in the resulting graph, see Figure 2.

Once the complete iteration graph is built, the final set of free indices determines the shape of the output tensor, and the bound index pairs determine the ranges of the reduction loops.

Code generation The code generation stage is enabled by typed template Haskell, [8], a type-checked extension to template Haskell (TH) [9] 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 3), resting assured that the resulting code will be well-formed. This approach to compiling embedded DSLs has been demonstrated e.g. in [10] and recently applied to parser combinators [11] and extensible algebraic datatypes [12].

We say that a compiler is "staged" when each phase evaluates at least in part the output of the previous one and emits source code and metadata that will be consumed by the subsequent phase. In typed TH, values of type `Code a` denote program fragments producing values of type `a`, whereas quoting and splicing are denoted with $[\![\cdot]\!]$ and `$$(.)` respectively.

We use staged compilation to synthesize iteration and reduction loops over the data that are specific to the algebraic expression provided as compiler input and hopefully optimal in some sense (e.g. by coalescing iterations over the same index). Informally, we aim to generate a function `VIn -> VOut` as written by a human programmer with `AST -> Code (VIn -> VOut)` .

Our current focus is on loop lowering combinators that construct indexing functions from the iteration graph and user-provided AST.

```haskell
-- | function abstraction
lam :: (Code x -> Code y)
      -> Code (x -> y)
lam f =
  [|| \y -> $$( f [|| y ||] ) ||]

-- | function application
(>*<) :: Code (a -> b)
        -> Code a -> Code b
f >*< a =
  [|| $$(f) $$(a) ||]
```

```haskell
loop2 ::
  Code (t1 -> t2 -> t3)
  -> Code (Int -> Int)
  -> Code (Int -> Int)
  -> Code (Int
             -> Vector t1
             -> Vector t2
             -> t3)
loop2 f fi1 fi2 =
  [|| \i v1 v2 ->
        let
           i1 = $$(fi1) i
           i2 = $$(fi2) i
         in
           $$(f) (v1 ! i1) (v2 ! i2)
   ||]
```

Figure 3: Left : Typed TH combinators for abstraction and β-reduction. Right : staged loop combinator for iterating over two arrays. Note : `loop2` produces a function that has the output index and the data vectors as inputs. Writing to the output array happens only at runtime.

3 Related work

Array compilers and metaprogramming Generative programming has a history of successes in self-tuning numerical libraries such as ATLAS [13] and FFTW [14]. Later research produced whole-program compilers that optimize parallelism and arithmetic intensity [15], adapt deep learning workloads to various hardware backends or generic compiler IR [16–19] and offer a mathematically-intuitive API for tensor expressions while retaining the performance of hand-tuned kernels [7]. Similarly to other polyhedral compilers [20], our DSL is limited to modeling static control part ("SCoP") of a numerical program, and it overlaps in scope with TACO [7], with the major difference of being embedded in a declarative language.

4 Discussion

In its current version, our system reassociates algebraic operations greedily, in order to fuse iteration loops. Some authors (e.g. [19, 21, 22]) have shown that storing intermediate arrays and transforming memory layout can improve overall arithmetic efficiency, even though picking the optimal such schedule is NP-hard [23], which motivates the use of approximations and heuristics at the expense of compiler complexity. Others (e.g. [18]) 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).

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1 250 010, 2012. eprint: `https://doi.org/10.1142/S0129626412500107`.

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

[22] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," Citeseer, Tech. Rep., 2008.

[23] 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.