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, $\text{contract} \ (1) \ (0) \ x \ y$ 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...
\[ (e) \quad ::= \quad T_n \langle sh \rangle \\
| \quad \text{contract} \langle ix \rangle \langle ix \rangle \langle e \rangle \\
| \quad \text{binary} \langle op2 \rangle \langle e \rangle \langle e \rangle \\
| \quad \text{unary} \langle op1 \rangle \langle e \rangle \\
\langle op2 \rangle \quad ::= \quad + \circ \\
\langle op1 \rangle \quad ::= \quad \text{scale} \ | \ \text{exp} \ | \ \text{log} \]

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.

\[
\begin{align*}
A_1 & \quad + \quad B_1 \\
A_2 & \quad + \quad B_2 \\
A_{ik} & = \quad B_{ijk} \quad c_j \\
A & \quad + \quad B \\
\end{align*}
\]

\( \forall (i, k) \in \text{range}(A_1) \times \text{range}(A_3) \quad D[i, k] = \sum_j A[i, k] + B[i, j, k]c[j] \)

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 \[⟦·⟧\] 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 \( \text{VIN} \rightarrow \text{VOut} \) as written by a human programmer with \( \text{AST} \rightarrow \text{Code} (\text{VIN} \rightarrow \text{VOut}) \).

Our current focus is on loop lowering combinators that construct indexing functions from the iteration graph and user-provided AST.
Figure 3: Left: Typed TH combinators for abstraction and $\beta$-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


eprint: https://doi.org/10.1142/S0129626412500107

