Abstract—Tensor algebra is an important computational abstraction that is increasingly used in data analytics, machine learning, engineering, and the physical sciences. However, the number of tensor expressions is unbounded, which makes it hard to develop and optimize libraries. Furthermore, the tensors are often sparse (most components are zero), which means the code has to traverse compressed formats. To support programmers we have developed \texttt{taco}, a code generation tool that generates dense, sparse, and mixed kernels from tensor algebra expressions. This paper describes the \texttt{taco} web and command-line tools and discusses the benefits of a code generator over a traditional library. See also the demo video at tensor-compiler.org/ase2017.

Index Terms—Tensor algebra, linear algebra, sparse, compiler

I. INTRODUCTION

Tensors generalize matrices to any number of dimensions and are used in diverse domains from quantum physics to machine learning and data analytics. Tensor algebra generalizes linear algebra to tensors and is a powerful abstraction that can be used to express many sophisticated computations.

The traditional approach to linear algebra is to create software libraries with optimized functions or methods (kernels) for all the binary expressions (e.g. matrix addition and matrix-vector multiplication). To compute a compound (non-binary) expression the programmer calls a sequence of binary kernels with vector and matrix temporaries. This approach to software development worked well in the past, but is now unsuitable due to new features that cause an explosion in the number of variants that must be developed.

First, the vectors, matrices and tensors of interest are often sparse, which means that most of the components are zero. For example, a tensor of Amazon reviews used to predict how a user will respond to a new product contains 13 gigabytes of non-zeros, but 107 exabytes of zeros [1]. To take advantage of sparsity several compressed formats have been devised that store only non-zeros, but 107 exabytes of zeros [1]. To take advantage of sparsity several compressed formats have been devised that store only non-zeros, however, this requires library developers to develop a variant of each kernel for each combination of supported formats. Second, the number of target platforms, such as multi-core CPUs, GPUs, TPs and distributed systems, is increasing. To take advantage of these architectures the library developers must rewrite each kernel for each platform. Third, it is expensive to compute compound expressions through a sequence of kernels, because the vector, matrix and tensor temporaries that are passed between them can be large, resulting in poor temporal locality. To address this issue, library developers write kernels that compute compound expressions. However, the number of compound expressions is unbounded, so developers can support only a subset of them. Finally, when we generalize linear algebra to tensor algebra, the number of binary expressions also becomes unbounded, since tensors can have any number of dimensions.

Compositional complexity is often managed with composable software components. At some point the interactions between these components become so complex that the interfaces start to look like a language and a meta-programming approach becomes necessary [3, Chapter 4], [4, Chapter 8]. The mathematical tensor algebra notation that we are concerned with in this paper calls for a meta-programming approach, because it is a small language.

Performance is, however, essential for tensor and linear algebra. For example, Google has expressed concern that the cost of deep neural networks will become prohibitive unless the performance of tensor computations is improved [2]. To resolve the tension between the need for generality and performance we turn to code generation.

In this demo, we discuss \texttt{taco}, the first tool that can generate efficient parallel code for compound tensor expressions, where the tensors are stored in dense and sparse formats. The tool implements the tensor algebra compiler theory described in previous work [5] and is available as a web tool, a command-line tool, and as a library. These tools can be used to generate and benchmark kernels, to search for optimal formats, and to interactively optimize code.

The \texttt{taco} command-line tool and library are available under the permissive MIT license at tensor-compiler.org and the web tool at tensor-compiler.org/codegen. A video demo is available at tensor-compiler.org/ase2017.

II. TENSOR ALGEBRA, STORAGE AND KERNELS

A tensor generalizes a matrix (with two dimensions) to any number of dimensions, called the tensor’s order. A vector is thus a 1st-order tensor and a matrix is a 2nd-order tensor. Tensor algebra, also called multilinear algebra, is a generalization of linear algebra to work on tensors of any order (linear algebra is a subset of tensor algebra). Tensor algebra expressions are best expressed using tensor index notation, developed by Ricci-Curbastro, Levi-Civita [6] and Einstein [7]. The following example uses index notation to compute a tensor-vector multiplication/contraction resulting in a matrix:

\[ A_{ij} = \sum_k B_{ijk} c_k. \]

With tensor index notation, tensor algebra expressions are written as scalar expressions with subscripted index variables.
that connect each component of the result to components of the operands. The tensor-vector contraction has two free variables $i$ and $j$ and one summation variable $k$. Free variables always index the result tensor, while summation variables never index the result tensor.

The simplest storage format for a tensor is a multi-dimensional array, which we call a dense format. Dense formats are appropriate for tensors that have few zeros and have useful properties such as fast random access and simple iteration spaces. For example, dense tensors with the same dimensions have the same iteration space, which can be used to generate efficient code.

Many tensors are sparse, which means that most components are zero. For these tensors storing only the non-zero values saves memory and may increase performance. Many sparse storage formats have been devised for matrices [8], [9] and for higher-order tensors [10], [11]. The key idea is to store the non-zero values together with an index data structure that identifies the tensor coordinates of each non-zero.

Compute kernels for tensor algebra expressions must iterate over operands to produce the non-zero values of the result. For dense expressions, the loop nest simply iterates over the polyhedral space defined by the range of each tensor index variable. Fig. 1 shows a C kernel for tensor-vector multiplication. Note the simple loop bounds and the statements on lines 6, 7 and 11 that compute locations in the tensor multi-dimensional arrays.

Compute kernels for tensor expressions with sparse operands require more care. The loops iterate over a sparse subset of the dense polyhedral iteration space—a polyhedron with holes. Fig. 2 shows tensor-vector multiplication code when tensor $B$ is stored as a sparse tensor (in every dimension) with corresponding index structures. Each loop iterates over the entries in a single dimension; the last loop iterates over non-zeros. Unlike dense storage, indirect loads are needed to traverse the index structure, which consist of two arrays for each dimension: a pos and an idx array. The idx array stores the coordinates of non-zero entries in that dimension, and the pos array stores the ranges of idx values belonging to each tensor slice in the preceding dimension.

The code becomes more complicated when more than one tensor operand is sparse. When only one operand is sparse, the code can iterate over its index structure and access the other operand’s components by computing their location. However, index structures do not permit such fast $O(1)$ random access. If more than one operand indexed by an index variable is sparse we must iterate over their merged iteration spaces (similar to a database merge join or the merge in mergesort). Fig. 3 shows the tensor-vector multiplication kernel when both $B$ and $c$ are sparse. Since the operator is a multiplication, the loops must iterate over the intersection between each row of $B$ and the vector $c$. The intersection merge code is shown on lines 10–22. We iterate over the intersection because if a component of either $B$ or $c$ at a location is zero then the result is zero and we do not need to compute it. In contrast, for addition we must iterate over the union of iteration spaces. With sparse iteration spaces and merges the kernels become more difficult to write by hand, motivating the automated code generation approach taken by the taco tools.

### III. The Taco Tools

The taco tool suite consists of a web tool, a command-line tool, and a C++ library. The command-line tool is built on top of the library and the web tool is built on top of the command-line tool. All three can be used to generate kernels. In addition, the command-line tool can be used to benchmark kernels and to interactively optimize code.
Fig. 4: The taco web tool with the MTTKRP tensor factorization kernel (tensor-compiler.org/codegen?demo=mttkrp). The generated code iterates through the sparse index of B; the other operands are dense and support random access.

A. Web Tool

The taco web tool is a hosted code generation tool available at tensor-compiler.org/codegen. It consists of a JavaScript client and a remote code generation server written in Python.

The web client implements a GUI where users can enter tensor index notation expressions in textual form (summations are implied when a variable does not index the result). Fig. 4 shows a screenshot with the Matricized Tensor Times Khatri-Rao Product (MTTKRP) expression. As a user enters an expression in the text box, the client parses it and dynamically populates a table with one format description row per tensor. The format descriptions specify the format of the tensor in each dimension. taco currently supports dense and sparse dimensions, and we plan to support more format types in the future. The dropdown menus can also be re-ordered through drag-and-drop to specify formats that store tensors in different directions (e.g. row-major versus column-major).

The user can instruct the web tool to generate code for the expression and tensor formats by pressing the button labeled “Generate Kernel”. The client then sends a request to a code generation server that calls the taco command-line tool to...
generate code. The code is then sent back to the client and displayed at the bottom of the webpage. There are three tabs: one that shows only the loops to compute values, one that shows only the loops to assemble sparse result tensors, and one that shows the complete code. The complete code is a C header file that the user can download or copy-paste into an application if it only needs that kernel. This is a lightweight alternative to downloading and linking against the full taco C++ library that supports every kernel and that provides convenient functionality such as file loaders.

B. Command-line Tool

The taco command-line tool is written in C++ and is built on top of the taco C++ library [5]. Both are publicly available under the permissive MIT license at code.tensor-compiler.org. The command-line tool provides all the code generation functionality of the web tool, but also supports measuring the size of tensors in different formats as well as benchmarking and code optimization workflows.

1) Tensor Size Measurements: It is useful to be able to measure the data size of a tensor in different formats. If the tensor is stored on disk the user can measure its size in a given format by combining the -i option that loads a tensor from a file with the -f option that sets tensor formats. The -i option supports several file formats including the FROSTT Sparse Tensor format (.tns) [12] and the Tensor Market Exchange format (.ttx) for general tensors, and the Matrix Market Exchange format (.mtx) [13] and the Harwell-Boing format (.rb) [14] for matrices. The following command creates a tensor B whose format is sparse in all dimensions and fills it with data from a file containing the Facebook Activities data set [15]:

$\text{taco } -f=B:sss -i=B:facebook.tns$

Choosing the first dimension to be dense slightly decreases the memory consumption, which means most matrix slices have at least one value. Dense dimensions also often lead to faster kernels, so this format is likely better for this tensor:

$\text{taco } -f=B:dss -i=B:facebook.tns}$

2) Benchmarking: Since performance is essential for tensor algebra, taco supports benchmarking kernel performance with the -time option. To aid benchmarking the tool also provides the -g option to generate synthetic data. With these options we can use the following command to benchmark the MTTKRP kernel from Fig. 4 on the Facebook tensor:

$\text{taco } "A(i,j) = B(i,k,l) \times C(k,j) \times D(l,j)"$ \
> -f=B:sss -f=C:dd -f=D:dd \
> -i=B:facebook.tns -d=j:25 \
> -g=C:d -d=D:dd -time=10 -verify \
> -read-source=mttkrp_cilk.h$

Finally, the MTTKRP tensor index notation expression. The second line specifies formats for the operands: B is all sparse while C and D are all dense. The third line loads B from a file. The i, k and l index variables are used to index into B so their ranges are inferred from the input file. Since the j index variable is not used to index B we set its size manually with the -d option. Finally, on the fourth line we use the -g option to generate dense data for the C and D matrices and include -time to tell taco to run benchmarks. Note that this option takes an optional number (e.g., -time=10), which denotes the number of times the compute kernel should be run. If this option is given then taco emits the mean, standard deviation, and median across the runs. The output of this command is given on the following lines. First, it prints the time spent reading the file and packing it into the sparse format of B. Next, it prints the size of each tensor, and finally it prints the time spent compiling the MTTKRP kernel and assembling and computing the values of A. Assembly is cheap since A is a dense matrix without indices.

3) Interactive Optimization Workflow: Finally, taco supports an interactive optimization workflow where programmers can use code generated by taco as a starting point for further manual optimization. The motivation for this workflow is to give developers an easy way to instrument kernels and to provide them with an escape hatch when taco does not (yet) support an optimization they need. The taco command-line tool writes source code to a file when passed the -write-source option. This lets a developer modify the code and then verify and/or benchmark it against the taco-generated kernel with the -read-source option. For example, suppose we want to try parallelizing the MTTKRP kernel with the Cilk parallel programming model [16]. taco emits code that uses OpenMP [17], but we can write out the kernel, modify it to use Cilk, and then use the following command line option to load, verify and benchmark it:

$\text{taco } "$A(i,j) = B(i,k,l) \times C(k,j) \times D(l,j)"$ \
> -f=B:sss -f=C:dd -f=D:dd \
> -i=B:facebook.tns -d=j:25 \
> -g=C:d -d=D:dd -time=10 -verify \
> -read-source=mttkrp_cilk.h$

Finally, the MTTKRP tensor index notation expression. The second line specifies formats for the operands: B is all sparse while C and D are all dense. The third line loads B from a file. The i, k and l index variables are used to index into B so their ranges are inferred from the input file. Since the j index variable is not used to index B we set its size manually with the -d option. Finally, on the fourth line we use the -g option to generate dense data for the C and D matrices and include -time to tell taco to run benchmarks. Note that this option takes an optional number (e.g., -time=10), which denotes the number of times the compute kernel should be run. If this option is given then taco emits the mean, standard deviation, and median across the runs. The output of this command is given on the following lines. First, it prints the time spent reading the file and packing it into the sparse format of B. Next, it prints the size of each tensor, and finally it prints the time spent compiling the MTTKRP kernel and assembling and computing the values of A. Assembly is cheap since A is a dense matrix without indices.

This workflow lets developers quickly try out, verify, and benchmark new ideas for tensor algebra kernel optimization.
SPLATT [20]. Furthermore, it is typically 14–1600 hand-optimized kernels from libraries such as Eigen and of sparsity and that sparse formats make sense at about 15% (in all dimensions). This result demonstrates the importance of matrices of increasing sparsity for sparse and dense formats shows tensor-vector multiplication performance with synthetic can use \texttt{taco} to generate linear and tensor algebra kernels, freeing them from error prone kernel development.

We believe the existence of this tool opens up many exciting opportunities. With the ability to automatically generate code, programmers can quickly explore the space of tensor kernels and different formats. Furthermore, this process can now be automated using heuristics or an autotuner. Such automatic systems become much more powerful when they are no longer constrained to a limited set of kernels and formats, but can instead optimize across the set of all kernels and many formats. The \texttt{taco} tools is publicly available under the MIT license and we are seeing increasing traffic. There are efforts underway to integrate \texttt{taco} with Julia [21] and TensorFlow [22] and in the future we plan to expand the code generator to support more formats. We also plan to support distributed code generation and accelerators, so that programmers are also relieved from porting kernels. We believe this requires support for new formats that lay out data to fit each accelerator architecture. Finally, another exciting direction for future work is to provide a scheduling language that lets users interactively control code optimization.


taco

We have evaluated the correctness of the \texttt{taco} tool suite with more than 300 unit tests. We have also run it on large data sets and compared the results to those produced by other popular libraries such as the Matlab Tensor Toolbox [18] and Eigen [19]. To further test the code generator we plan to develop a fuzz tester that generates tensor algebra expressions and runs them with operands in every combination of formats, comparing the results to each other. We also collect anonymized information from the web tool that we will use to learn about tensor algebra usage.

We have also evaluated the performance of the code produced by \texttt{taco} and found that it is competitive with hand-optimized kernels from libraries such as Eigen and SPLATT [20]. Furthermore, it is typically 14–1600× faster than the Matlab Tensor Toolbox, which is the most general prior system we found. For additional performance results, including a comparison of formats and plots showing when sparse formats are better than dense formats, see our paper on the Tensor Algebra Compiler theory and library [5].

Tables I and II shows performance results collected using the command-line tool. Table I shows matrix-vector multiplication performance for four matrices in four formats. The results demonstrate the importance of an approach that can generate code for many different formats; each matrix performs best when using a different storage format. Table II shows tensor-vector multiplication performance with synthetic matrices of increasing sparsity for sparse and dense formats (in all dimensions). This result demonstrates the importance of sparsity and that sparse formats make sense at about 15% sparsity for this operation.

<table>
<thead>
<tr>
<th>Dense-Dense</th>
<th>Dense-Sparse</th>
<th>Sparse-Dense</th>
<th>Sparse-Sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>71.44</td>
<td>111.24</td>
<td>71.49</td>
<td>112.20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dense Matrix</th>
<th>Thermal Matrix</th>
<th>Slicing Matrix</th>
<th>Hypersparse Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>71.48</td>
<td>71.20</td>
<td>71.27</td>
<td>90% 99%</td>
</tr>
</tbody>
</table>

C. Summary of Evaluation

We have evaluated the correctness of the \texttt{taco} tool suite with more than 300 unit tests. We have also run it on large data sets and compared the results to those produced by other popular libraries such as the Matlab Tensor Toolbox [18] and Eigen [19]. To further test the code generator we plan to develop a fuzz tester that generates tensor algebra expressions and runs them with operands in every combination of formats, comparing the results to each other. We also collect anonymized information from the web tool that we will use to learn about tensor algebra usage.

We have also evaluated the performance of the code produced by \texttt{taco} and found that it is competitive with hand-optimized kernels from libraries such as Eigen and SPLATT [20]. Furthermore, it is typically 14–1600× faster than the Matlab Tensor Toolbox, which is the most general prior system we found. For additional performance results, including a comparison of formats and plots showing when sparse formats are better than dense formats, see our paper on the Tensor Algebra Compiler theory and library [5].

Tables I and II shows performance results collected using the command-line tool. Table I shows matrix-vector multiplication performance for four matrices in four formats. The results demonstrate the importance of an approach that can generate code for many different formats; each matrix performs best when using a different storage format. Table II shows tensor-vector multiplication performance with synthetic matrices of increasing sparsity for sparse and dense formats (in all dimensions). This result demonstrates the importance of sparsity and that sparse formats make sense at about 15% sparsity for this operation.

IV. Discussion

To the best of our knowledge, \texttt{taco} is the first tool to generate code for such a large set of dense and sparse tensor index notation expressions. Library and application developers can use \texttt{taco} to generate linear and tensor algebra kernels, freeing them from error prone kernel development.

We believe the existence of this tool opens up many exciting opportunities. With the ability to automatically generate code, programmers can quickly explore the space of tensor kernels and different formats. Furthermore, this process can now be automated using heuristics or an autotuner. Such automatic systems become much more powerful when they are no longer constrained to a limited set of kernels and formats, but can instead optimize across the set of all kernels and many formats.

The \texttt{taco} tools is publicly available under the MIT license and we are seeing increasing traffic. There are efforts underway to integrate \texttt{taco} with Julia [21] and TensorFlow [22] and in the future we plan to expand the code generator to support more formats. We also plan to support distributed code generation and accelerators, so that programmers are also relieved from porting kernels. We believe this requires support for new formats that lay out data to fit each accelerator architecture. Finally, another exciting direction for future work is to provide a scheduling language that lets users interactively control code optimization.

V. Related Work

There are many available libraries and frameworks for linear and tensor algebra. Most are written in the traditional way with hand-optimized kernel implementations. On one extreme is the ubiquitous BLAS dense linear algebra library, which consists of 142 functions that are mostly variants of a smaller set of kernels [23]. OSKI is a library that supports the Blocked Compressed Sparse Row format (BCSR)—a sparse format with dense inner blocks—and supports autotuning the block sizes for a few kernels [18]. Eigen is a modern and convenient C++ linear algebra header library that uses meta-programming through C++ templates to optimize dense linear algebra operations [19].

Many dense and sparse tensor libraries have been developed in the last decade. Because tensor algebra results in an unbounded number of kernels for binary expressions and because compound kernels are often essential for performance, these libraries tend to use meta-programming techniques. The Tensor Contraction Engine is an early library for compound dense tensor expressions that relied on compiler techniques to produce a fused loops [24]. OSKI is a modern and convenient C++ linear algebra header library that uses metaprogramming through C++ templates to optimize dense linear algebra operations [19].

Many dense and sparse tensor libraries have been developed in the last decade. Because tensor algebra results in an unbounded number of kernels for binary expressions and because compound kernels are often essential for performance, these libraries tend to use meta-programming techniques. The Tensor Contraction Engine is an early library for compound dense tensor expressions that relied on compiler techniques to produce a fused loops [25]. The Matlab Tensor Toolbox is an early library for sparse tensor algebra that supports many sparse expressions and is built on top of the sparse linear algebra in Matlab [19]. Finally, TensorFlow is a recent framework that provides many hand-written kernels for dense tensor

<table>
<thead>
<tr>
<th>0% 10% 20% 50% 64% 90% 99%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
</tr>
<tr>
<td>Sparse</td>
</tr>
</tbody>
</table>
algebra and some for sparse tensor algebra [22]. The TensorFlow developers have recently adopted meta-programming with their XLA (Accelerated Linear Algebra) compiler for dense linear algebra.

VI. CONCLUSION

The unbounded number of tensor algebra kernels and the need for performance makes a code generation approach necessary. The tool we have demonstrated, taco, automatically generates tensor algebra kernels for many different formats. The user specifies a tensor algebra expression and formats the necessary. The tool we have demonstrated, taco, automatically generates tensor algebra kernels and formats and taco generates a C function to compute the expression. This relieves programmers from kernel development and lets them quickly explore different kernels and formats.

ACKNOWLEDGMENT

We thank Rami Manna and Apurva Shrivastava for their help with the tensor-compiler.org webpage and documentation. The development of taco was supported by the National Science Foundation under Grant No. CCF-1533753, by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC008923 and DE-SC014204, by the Direction Générale de l’Armement (Projet ERE 2016929), and by the Toyota Research Institute.

REFERENCES