taco: A Tool to Generate Tensor Algebra Kernels

Fredrik Kjolstad
MIT CSAIL
fred@csail.mit.edu

Stephen Chou
MIT CSAIL
s3chou@csail.mit.edu

David Lugato
CEA
david.lugato@cea.fr

Shoaib Kamil
Adobe
kamil@adobe.com

Saman Amarasinghe
MIT CSAIL
saman@csail.mit.edu

***Abstract*—Tensor algebra is an increasingly important computational abstraction that is 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 `taco`, a code generation tool that generates dense, sparse and mixed kernels from tensor algebra expressions. This paper describes the `taco` web and command-line tools, and discusses the benefits of a code generator over a traditional library approach.**

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 suffers from several limitations, and is 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 by McAuley and Leskovec 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 only store 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, TPUs [2] 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 only support 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 such a meta-programming approach because it is a small language.

However, performance is 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 `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 `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 1-order tensor and a matrix is a 2-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

```
1  for (int i = 0; i < m; i++) {
2
3
4
5   for (int j = 0; j < n; j++) {
6    int jB_pos = i*n + j;
7    int jA_pos = i*n + j;
8
9
10   for (int k = 0; k < p; k++) {
11    int kB_pos = jB_pos*p + k;
12
13
14
15
16
17
18    A[jA_pos] += B[kB_pos] * c[k];
19
20
21
22   }
23  }
24 }
```

Fig. 1: $A_{ij} = B_{ijk}c_k$

```
for (int iB_pos = B1.pos[0];
     iB_pos < B1.pos[1];
     iB_pos++) {
 int i = B1.idx[iB_pos];
 for (int jB_pos = B2.pos[iB_pos];
      jB_pos < B2.pos[iB_pos+1];
      jB_pos++) {
  int j = B2.idx[jB_pos];
  int jA_pos = i*n + j;
  for (int kB_pos = B3.pos[jB_pos];
       kB_pos < B3.pos[jB_pos+1];
       kB_pos++) {
   int k = B3.idx[kB_pos];



   A[jA_pos] += B[kB_pos] * c[k];


  }
 }
}
```

Fig. 2: $A_{ij} = B_{ijk}c_k$ with sparse B

```
for (int iB_pos = B1.pos[0];
     iB_pos < B1.pos[1];
     iB_pos++) {
 int i = B1.idx[iB_pos];
 for (int jB_pos = B2.pos[iB_pos];
      jB_pos < B2.pos[iB_pos+1];
      jB_pos++) {
  int j = B2.idx[jB_pos];
  int jA_pos = i*n + j;
  int kB_pos = B3.pos[jB_pos];
  int kc_pos = c1.pos[0];
  while (kB_pos < B3.pos[jB_pos+1] &&
         kc_pos < c1.pos[1]) {
   int kB = B3.idx[kB_pos];
   int kc = c1.idx[kc_pos];
   int k = min(kB, kc);
   if (kB == k && kc == k) {
    A[jA_pos] += B[kB_pos] * c[kc_pos];
   }
   if (kB == k) kB_pos++;
   if (kc == k) kc_pos++;
  }
 }
}
```

Fig. 3: $A_{ij} = B_{ijk}c_k$ with sparse B and c

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, appropriate for tensors that have few zeros. Dense formats have useful properties such as enabling 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 $\Theta(1)$ 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.

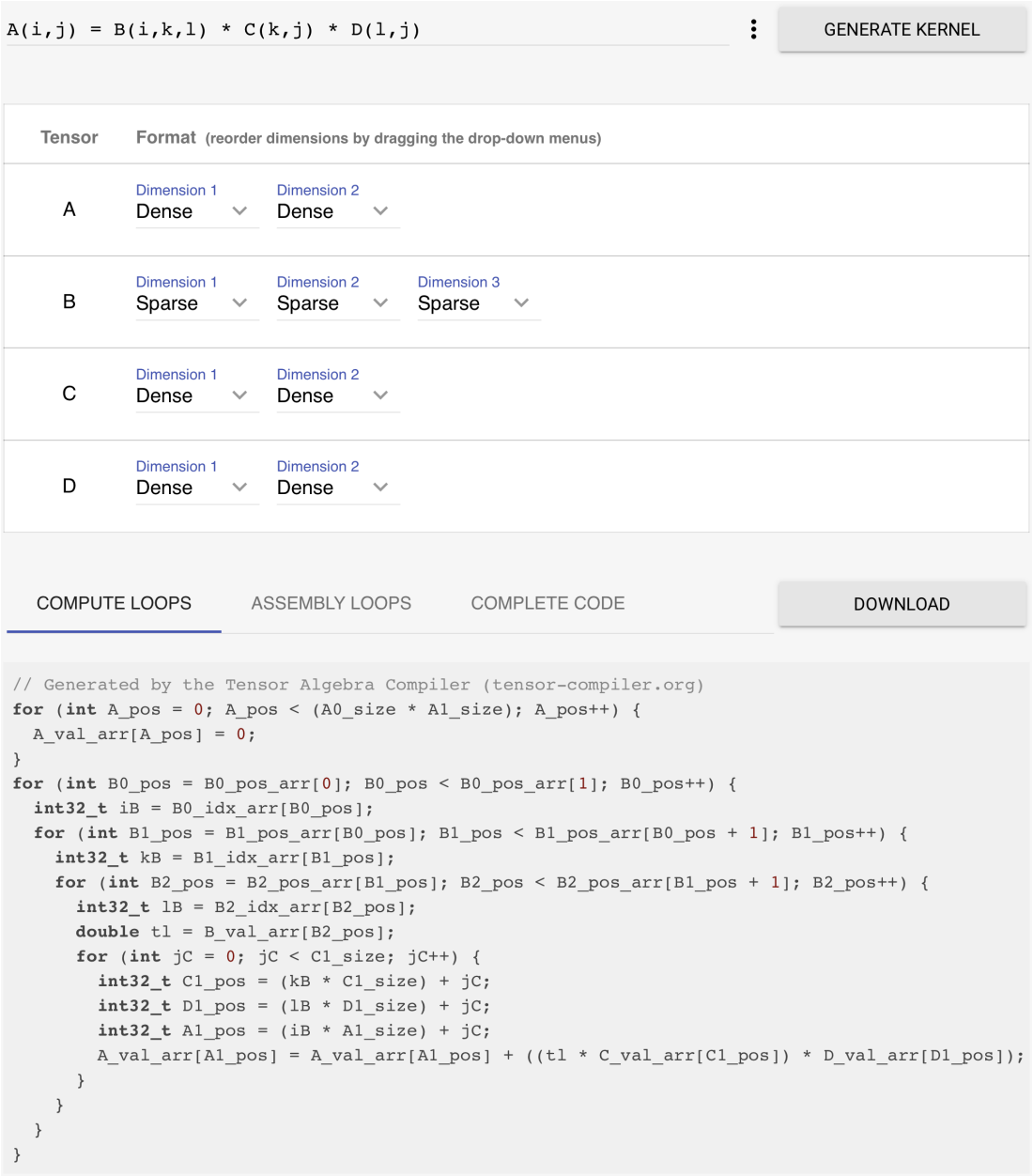

Fig. 4: The `taco` web tool with the MTTKRP tensor factorization kernel (tensor-compiler.org/codegen?demo=mttkrp). The generated code iterates through the sparse index of B; other operands are dense and support random access.

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. Fig. 4 shows a screenshot with the Matricized Tensor Times Khatri-Rao Product (MTTKRP) expression.[1] 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 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

[1]MTTKRP is a key kernel in algorithms that compute the Canonical Polyadic Decomposition tensor factorization, which is one generalization of SVD to higher-order tensors.

displayed at the bottom of the webpage. There are three tabs: one that only shows the loops to compute values, one that only shows the loops to assemble sparse result tensors, and one that shows the complete code. The complete code is a functioning C library and header file that the user can download or copy-paste into an application if it only needs that kernel. This is a light-weight alternative to downloading and linking against the full `taco` C++ library that supports every kernel and 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 Exhange 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]:

```
$taco -f=B:sss -i=B:facebook.tns
B size: (1504 x 42390 x 39986), 13768416 bytes
```

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:

```
$taco -f=B:dss -i=B:facebook.tns
B size: (1504 x 42390 x 39986), 13762396 bytes
```

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:

```
$taco "A(i,j) = B(i,k,l) * C(k,j) * D(l,j)" \
>     -f=B:sss -f=C:dd -f=D:dd              \
>     -i=B:facebook.tns -d=j:25             \
>     -g=C:d -g=D:d -time
B file read: 818.855 ms
B pack:      612.248 ms
B size: (1504 x 42390 x 39986), 13768416 bytes
C size: (42390 x 25), 8478008 bytes
D size: (39986 x 25), 7997208 bytes

Compile:  86.1086 ms
Assemble: 0.059579 ms
Compute:  27.9097 ms
```

The first four lines is the command. The first line contains 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 accross 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, 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 will write source code to a file when passed the `-write-source` option, enabling a developer to modify it and then verify and/or benchmark the modified code against the `taco`-generated kernel with the `-read-source` option. For example, suppose we want to try parallelizing the MTTKRP kernel using 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:

```
$taco "A(i,j) = B(i,k,l) * C(k,j) * D(l,j)" \
>     -f=B:sss -f=C:dd -f=D:dd              \
>     -i=B:facebook.tns -d=j:25             \
>     -g=C:d -g=D:d -time=10 -verify        \
>     -read-source=mttkrp_cilked: 826.351 ms
B pack:      629.852 ms
B size: (1504 x 42390 x 39986), 13768416 bytes
C size: (42390 x 25), 8478008 bytes
D size: (39986 x 25), 7997208 bytes

Compile:  173.053 ms
Assemble: 0.044249 ms
Compute time (ms)
mean:   18.7999
stdev:  1.3631
median: 17.7584

mttkrp_cilk.h:
Assemble: 0.040121 ms
Compute time (ms)
mean:   11.9734
stdev:  1.53868
median: 11.0135

Verifying... done
```

This workflow lets developers quickly try out, verify and benchmark new ideas for tensor algebra kernel optimization.

TABLE I: Benchmark data collected with the `taco` command-line tool `-time` option. The benchmarks show the time in milliseconds to compute a matrix-vector multiplication with four matrices stored in four different formats. The matrices exemplify common sparsity structures. The table diagonal shows the importance of choosing formats to match the matrices.

	Dense-Dense	Dense-Sparse	Sparse-Dense	Sparse-Sparse
Dense Matrix	**71.44**	111.24	71.49	112.20
Thermal Matrix	71.48	**0.10**	71.61	0.10
Slicing Matrix	71.20	0.97	**0.52**	0.96
Hypersparse Matrix	71.27	0.04	0.36	**0.03**

TABLE II: Time in milliseconds to compute a tensor-vector multiplication using synthetic tensors with increasing numbers of randomly located zeros. The data shows that an all sparse tensor outperforms an all dense tensor on this operation when 15% of the values are zero.

	0%	10%	15%	20%	50%	90%	99%
Dense	19.85	19.85	19.85	19.85	19.85	19.85	19.85
Sparse	30.18	22.01	18.48	15.37	3.77	0.16	0.01

C. Summary of Evaluation

We have evaluated the correctness of the `taco` tool suite using over 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 analyze to learn about tensor algebra usage.

We have also evaluated the performance of the code produced by `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, 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, `taco` is the first tool to generate code for such a large set of dense and sparse tensor index expressions. Library and application developers can use `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 `taco` tools have only been publicly available under the MIT license for two months and our webpage is seeing increasing traffic. There are efforts underway to integrate `taco` with Julia [21] and TensorFlow [22] and in the future we plan to expand the code generator to support more formats and distributed code generation.

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 implementations of the kernels. 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 [24]. Eigen is a modern and convenient C++ linear algebra header library that provides some meta-programming support for dense linear algebra operations through C++ templates [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 [18]. Finally, TensorFlow is a recent framework that provides many hand-written kernels for dense tensor algebra and some for sparse tensor algebra [22]. The TensorFlow developers have recently adopted meta-programming with their XLA (Accelerated Linear Algebra) compiler.

VI. Conclusion

The unbounded number of tensor algebra kernels and the need for performance makes a code generation approach necessary. This paper demonstrates `taco`, a tool to automatically generate tensor algebra kernels for many different formats. The user specifies a tensor algebra expression 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

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, and by the Toyota Research Institute.

References

[1] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 2013, pp. 165–172.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[3] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and interpretation of computer programs*. Justin Kelly, 1996.

[4] E. S. Raymond, *The art of Unix programming*. Addison-Wesley Professional, 2003.

[5] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The Tensor Algebra Compiler," in *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2017.

[6] G. Ricci-Curbastro and T. Levi-Civita, "Méthodes de calcul différentiel absolu et leurs applications," *Mathematische Annalen*, vol. 54, 1901.

[7] A. Einstein, "The foundation of the General Theory of Relativity," *Annalen der Physik*, vol. 354, pp. 769–822, 1916.

[8] D. R. Kincaid, T. C. Oppe, and D. M. Young, *ITPACKV 2D User's Guide*, 1989.

[9] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proceedings of the IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

[10] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 5.

[11] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin, "Efficient and scalable computations with sparse tensors," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–6.

[12] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[13] R. F. Boisvert, R. Pozo, and K. A. Remington, "The matrix market exchange formats: Initial design," *National Institute of Standards and Technology Internal Report, NISTIR*, vol. 5935, 1996.

[14] I. S. Duff, R. G. Grimes, and J. G. Lewis, "Users' guide for the harwell-boeing sparse matrix collection (release i)," 1992.

[15] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, August 2009.

[16] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[17] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[18] B. W. Bader and T. G. Kolda, "Efficient matlab computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, 2007.

[19] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[20] S. Smith, N. Ravindran, N. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015, pp. 61–70.

[21] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," 2012.

[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 265–283. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026899

[23] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[24] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, no. 1, pp. 521+, 2005.

[25] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Automatic code generation for many-body electronic structure methods: the Tensor Contraction Engine," *Molecular Physics*, vol. 104, no. 2, pp. 211–228, 2006.