AutoHOOT: Automatic High-Order Optimization for Tensors

Linjian Ma*
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL
lma16@illinois.edu

Jiayu Ye*
Google
Sunnyvale, CA
yejiayu@google.com

Edgar Solomonik
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL
solomon2@illinois.edu

ABSTRACT

High-order optimization methods, including Newton's method and its variants as well as alternating minimization methods, dominate the optimization algorithms for tensor decompositions and tensor networks. These tensor methods are used for data analysis and simulation of quantum systems. In this work, we introduce AutoHOOT, the first automatic differentiation (AD) framework targeting at high-order optimization for tensor computations. AutoHOOT takes input tensor computation expressions and generates optimized derivative expressions. In particular, AutoHOOT contains a new explicit Jacobian / Hessian expression generation kernel whose outputs maintain the input tensors' granularity and are easy to optimize. The expressions are then optimized by both the traditional compiler optimization techniques and specific tensor algebra transformations. Experimental results show that AutoHOOT achieves competitive CPU and GPU performance for both tensor decomposition and tensor network applications compared to existing AD software and other tensor computation libraries with manually written kernels. The tensor methods generated by AutoHOOT are also well-parallelizable, and we demonstrate good scalability on a distributed memory supercomputer.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; **Automatic differentiation**; *Nonconvex optimization.*

KEYWORDS

automatic differentiation; computational graph optimization; tensor computation; tensor decomposition; tensor network

ACM Reference Format:
Linjian Ma, Jiayu Ye, and Edgar Solomonik. 2020. AutoHOOT: Automatic High-Order Optimization for Tensors. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3410463.3414647

*Both authors contributed equally to this research.

This work is licensed under a Creative Commons Attribution International 4.0 License.

PACT '20, October 3–7, 2020, Virtual Event, GA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8075-1/20/10.
https://doi.org/10.1145/3410463.3414647

1 INTRODUCTION

Tensors, represented as multidimensional arrays in the computer program, are important in both scientific computing and machine learning. Tensor decomposition [25] is a powerful tool in compressing and approximating the high dimensional data, and is used widely in numerical PDEs [38], quantum chemistry [16, 17] and statistical modeling [4, 44]. Tensor networks are also widely used in physics to approximate quantum states [33, 36] and in neural networks to form tensorized neural architectures [35]. Convolution, which is a basic tensor operation, is widely used in computer vision applications [27]. Tensors are also widely used in methods for electronic structure calculations in computational chemistry [15].

Derivatives, mostly in the form of gradients, are ubiquitous in the optimization algorithms for tensor related problems. For neural networks, they are used to calculate the gradients of the loss function w.r.t. the model parameters. For tensor decomposition and tensor networks, first-order and higher-order derivatives are necessary to construct the operators used in the alternating optimization. Gradients of computational chemistry methods are used for optimization of the electronic geometry to identify stable states and state transitions [21]. Automatic differentiation (AD) frameworks, including popular Python tools such as PyTorch [37], JAX [7], and TensorFlow [1], can generate derivatives in all of these contexts. However, in tensor decomposition, tensor networks, and quantum chemistry, gradient calculations are most often done via manually written codes, as careful numerical and performance considerations are required in these more complex settings.

AD transforms a software or mathematical expression of a function into code for computation of its derivatives with respect to the desired parameters. Although mathematically correct, the output programs for the derivatives may be sub-optimal in computational cost, use of efficient kernels such as the BLAS, memory footprint, and numerical stability. Components of different frameworks address these problems jointly or independently. For example, transformations of the computational graph and operator fusion are used to improve computational efficiency and parallelizability [1, 20, 37]. Gradient checkpointing and garbage collection are used to address memory bottlenecks [1, 37]. For large scale tensor computations, computational and memory demands leave little leeway for error in these aspects.

Common commercial AD frameworks such as PyTorch [37], JAX [7], and TensorFlow [1] are focused on first-order numerical optimization methods on deep learning models. In the context of tensor decompositions, tensor network optimization, and differentiation of tensor methods, three major additional challenges arise.

(1) These domains predominantly employ alternating second-order optimization methods, as they provide monotonic convergence and rapid progress at almost the same per-iteration cost as first-order methods. These methods employ implicit representations of the Jacobian and Hessian to solve linear systems. Existing AD frameworks have limited logical constructs for second-order derivative information, and consequently generate code that can be sub-optimal in cost by orders of magnitude.
(2) Most tensor operations involved in the deep learning applications are related to small tensors, while in tensor network and tensor decomposition applications, there are many tensor contractions over high order (multidimensional) tensors with a large number of elements. Therefore, tensor network applications require better optimization algorithms to select optimized contraction order and eliminate redundant calculations.
(3) Deep learning computational graphs usually have large depth with many nonlinear operations, making the freedom to optimize tensor operations limited. On the other hand, in tensor decomposition and tensor network applications, the computational graphs are usually wide and have small depth, so there is more freedom to optimize the computation.

Although many frameworks, such as Tensorly [26], TensorNetwork [42] and Quimb [11], provide interfaces to optimize the tensor decomposition / networks algorithms with AD frameworks such as TensorFlow and PyTorch, the optimization algorithms are the general first-order methods and its variants. These frameworks explicitly implement popular second-order methods for these problems, such as Alternating Least Squares (ALS) for tensor decompositions and Density Matrix Renormalization Group (DMRG) for 1D tensor networks, rather than using AD. The ability to generate efficient expressions of these methods automatically via AD, would accelerate the development of new variants and their deployment on shared-memory, GPU, and distributed-memory architectures.

In this paper, we propose a new AD framework for tensor computations, Automatic High-Order Optimization for Tensors (AutoHOOT). AutoHOOT encapsulates the following novel ideas and capabilities:

- a new AD module that generates more efficient representations for higher-order derivative constructs such as Jacobians and Hessians, which are needed for tensor computation applications,
- a new computational graph optimization module that extends beyond the traditional optimization techniques for compilers with tensor-algebra specific transformations, such as distributivity of matrix inversion over the Kronecker product,
- portability via high-level support for different tensor contraction backends: NumPy for multi-core CPU, TensorFlow for GPUs, and Cyclops [49] for distributed memory systems,
- substantial improvements in sequential and parallel performance for tensor network and tensor decomposition optimizations over other AD libraries and competitive or improved performance w.r.t. manually-optimized implementations.

2 BACKGROUND

2.1 Notation and Definitions

For vectors, bold lowercase Roman letters are used, e.g., $\mathbf{x}$. For matrices, bold uppercase Roman letters are used, e.g., $\mathbf{X}$. For tensors, bold calligraphic uppercase Roman letters are used, e.g., $\boldsymbol{\mathcal{X}}$. An order N tensor corresponds to an N-dimensional array with dimensions $s_1 \times \cdots \times s_N$. Elements of vectors, matrices, and tensors are denoted in parentheses, e.g., $\mathbf{x}(i)$ denotes the ith entry of a vector $\mathbf{x}$, $\mathbf{X}(i,j)$ denotes the (i,j)th element of a matrix $\mathbf{X}$, and $\boldsymbol{\mathcal{X}}(i,j,k,l)$ denotes the (i,j,k,l)th element of an order 4 tensor $\boldsymbol{\mathcal{X}}$. Subscripts are used to label different vectors, matrices, tensors and functions (e.g. $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$, f_1 and f_2).

Matricization is the process of unfolding a tensor into a matrix. Given a tensor $\boldsymbol{\mathcal{X}}$ the mode-n matricized version is denoted by $\mathbf{X}_{(n)} \in \mathbb{R}^{s_n \times K}$ where $K = \prod_{m=1, m\neq n}^{N} s_m$. We generalize this matricization definition, so that $\mathbf{X}_{(i:j)}$ means that the dimensions from the ith index to the jth index are unfolded to the column dimension of the matrix, and all the other dimensions are unfolded to the row dimension of the matrix.

For a scalar output function $y = f(\mathbf{a}_1, \ldots, \mathbf{a}_N)$, We use the $\mathbf{g}^{[f]}_{[\mathbf{a}_i]}$ and $\mathbf{H}^{[f]}_{[\mathbf{a}_i]}$ to denote the gradient vector and Hessian matrix of f w.r.t the input vectors $\mathbf{a}_i$. When the inputs are tensors, the gradient and the Hessian will also be a tensor and denote $\boldsymbol{\mathcal{G}}^{[f]}_{[\boldsymbol{\mathcal{A}}_i]}$ and $\boldsymbol{\mathcal{H}}^{[f]}_{[\boldsymbol{\mathcal{A}}_i]}$. For a function with non-scalar output $\mathbf{y} = f(\mathbf{a}_1, \ldots, \mathbf{a}_N)$, we use $\mathbf{J}^{[f]}_{[\mathbf{a}_i]}$ to denote the Jacobian matrix of the function f w.r.t one of the input vectors $\mathbf{a}_i$. The shape of the Jacobian matrix will be $\mathbb{R}^{|\mathbf{y}| \times |\mathbf{a}_i|}$. If $\boldsymbol{\mathcal{Y}}$ is an output tensor with size $\mathbb{R}^{s_1 \times \cdots \times s_M}$, and $\boldsymbol{\mathcal{A}}_i$ is an input tensor with size $\mathbb{R}^{r_1 \times \cdots \times r_K}$, then the Jacobian will be a tensor denoted as $\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{A}}_i]}$ with dimensions $\mathbb{R}^{s_1 \times \ldots \times s_M \times r_1 \times \ldots \times r_K}$.

We also define generalized Vector Jacobian Product (VJP), Jacobian Vector Product (JVP) and Hessian Vector Product (HVP). When both Jacobian and Hessian are matrices, these are matrix-vector multiplication operations. When Jacobian and Hessian are both tensors defined above, these are tensor contractions, whose results are the same as unfolding the tensors into matrices and performing the matrix-vector product.

2.2 Numerical Optimization Algorithms for Tensor Computations

We consider two tensor numerical problems: the nonlinear least squares fitting and the eigenvalue problem. For both problems, we denote $\boldsymbol{\mathcal{X}}$ as the input tensor which can be an explicit tensor or implicit tensor network (e.g., Matrix Product Operator [57]), f as a tensor network function and $\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_N$ as the optimization variables. Then the objective for the nonlinear least squares problem is defined as

$$\min_{\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_N} \frac{1}{2} \|\boldsymbol{\mathcal{X}} - f(\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_N)\|^2, \tag{1}$$

which finds a generalized low rank approximation of the input tensor $\boldsymbol{\mathcal{X}}$. The objective for the eigenvalue problem is defined as

$$\min_{\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_N} \frac{\mathbf{v}^T_{(1:N)} \mathbf{X}_{(1:N)} \mathbf{v}_{(1:N)}}{\|\boldsymbol{\mathcal{V}}\|_F^2}, \tag{2}$$

where $\boldsymbol{\mathcal{V}} = f(\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_N)$ and the output of f serves as a generalized low rank approximation of the eigenvector of a Hermitian matrix that is a matricization of $\boldsymbol{\mathcal{X}}$.

Three categories of algorithms are generally used to optimize the problems: second-order methods, including Newton's method and

its variants, alternating minimization, which updates each input / site at one time, and first-order methods such as gradient descent and its variants.

Newton's method and its variants. Newton's method and its variants, such as Gauss-Newton (GN) method, are popular methods to solve non-linear least squares problems for a quadratic objective function defined in Equation 1. Let $\mathbf{a}$ denote the concatenation of all the vectorized sites $\text{vec}(\boldsymbol{\mathcal{A}}_i)$ and $\hat{f}(\mathbf{a}) = \text{vec}(f(\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_N))$, so that $r(\mathbf{a}) := \text{vec}(\boldsymbol{\mathcal{X}}) - \hat{f}(\mathbf{a})$ denotes the vectorized residual. Further, let $r_i(\mathbf{a})$ denote the ith element of the output of function r. The gradient and the Hessian matrix of

$$\phi(\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_N) := \frac{1}{2}\|\boldsymbol{\mathcal{X}} - f(\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_N)\|^2,$$

can be expressed as

$$\nabla\phi(\mathbf{a}) = \mathbf{J}_{[\mathbf{a}]}^{[r]T} r(\mathbf{a}), \quad \text{and} \quad \mathbf{H}_{[\mathbf{a}]}^{[\phi]} = \mathbf{J}_{[\mathbf{a}]}^{[r]T}\mathbf{J}_{[\mathbf{a}]}^{[r]} + \sum_i r_i(\mathbf{a})\mathbf{H}_{[\mathbf{a}]}^{[r_i]}.$$

The Newton iteration performs the update based on

$$\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} - (\mathbf{H}_{[\mathbf{a}^{(k)}]}^{[\phi]})^{-1}\mathbf{J}_{[\mathbf{a}^{(k)}]}^{[r]T} r(\mathbf{a}^{(k)}),$$

while the Gauss-Newton method leverages the fact that $\mathbf{H}_{[\mathbf{a}]}^{[r_i]}$ is negligible as its norm is small when the residual is small, therefore the update can be performed as

$$\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} - (\mathbf{J}_{[\mathbf{a}^{(k)}]}^{[r]T}\mathbf{J}_{[\mathbf{a}^{(k)}]}^{[r]})^{-1}\mathbf{J}_{[\mathbf{a}^{(k)}]}^{[r]T} r(\mathbf{a}^{(k)}),$$

where $\mathbf{a}^{(k)}$ represents the $\mathbf{a}$ at kth iteration. The Gauss-Newton updates can be regarded as normal equations for the linear least squares problem. Both Newton and Gauss-Newton methods can be solved via the conjugate gradient method with matrix-vector products performed with an implicit form of the Jacobian / Hessian to avoid costly matrix inversion [45, 53].

Alternating minimization. For tensor numerical optimization, in many cases both the input and output dimensions are large, and it's computationally expensive to form the explicit Hessian / Jacobian matrix w.r.t. all the variables and perform the second-order method directly. On the other hand, when optimizing a subset of variables, forming the Hessian or Jacobian with respect to those variables is affordable and effective. Most often, alternating minimization procedures update one tensor operand at a time. For Equation 1, such subproblem can be formulated as

$$\min_{\boldsymbol{\mathcal{A}}_i} \frac{1}{2}\|\boldsymbol{\mathcal{X}} - f(\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_N)\|^2. \tag{3}$$

Each $\boldsymbol{\mathcal{A}}_i$ for $i \in \{1, \ldots, N\}$ is updated once via its subproblem during an optimization sweep. For tensor decompositions and tensor networks, each subproblem is often quadratic, allowing for the minima to be found directly, often at a similar cost to updating $\boldsymbol{\mathcal{A}}_i$ with a first-order method. Alternating minimization also generally provides monotonic convergence.

In each sweep, many terms necessary to form the subproblems have many equivalent intermediates, and choosing the proper contraction paths to form and also amortize them can greatly save the cost. This scheme, called the *dimension tree* algorithm, is critical to the algorithm performance, and has been used in both tensor decompositions [29, 40, 56] and DMRG to save the cost.

First-order methods. The efficacy of the first-order methods on tensor computations is dependent on the applications. The first-order methods are shown to be advantageous on achieving high fitting accuracies on some tensor decomposition problems [2], while they also perform worse than alternating minimization in achieving high accuracy for large scale tensor completion problems [60]. The per-iteration cost of first-order methods is often comparable to that of both second-order methods and the alternating minimization method, due to the structure of tensor networks f in Equation 1,2.

Traditional AD frameworks can generate efficient kernels for first-order methods, while their performance on the kernels in higher-order methods is suboptimal. In this paper, we focus on the performance optimization over both second-order method and alternating minimization methods, to accelerate future development of efficient high-order methods for various applications. However, we believe our graph optimization techniques also have the potential to produce efficient formulations for first-order methods, where the objective involves contractions of high-order tensors, which arise in quantum chemistry methods [15].

2.3 Previous Work

Optimization for tensor computations requires three essential building blocks, automatic differentiation, optimization of the generated set of tensor operations, and a computational backend for individual tensor operations. Existing software for tensor computations, including Tensorly [26], TensorNetwork [42] and Quimb [11] permit the use of multiple backends for individual tensor operations, and provide some constructs to make use of AD. However, when using AD, these libraries employ general AD backends such as JAX or TensorFlow in a black-box fashion.

Automatic differentiation is generally provided via one of two ways, operator overloading [7, 31, 37, 54, 59] or source code transformation (SCT) [1, 19, 55]. Operator overloading requires the user to write functions in terms of the provided library constructs and constructs the derivatives at run-time, while SCT uses precompilation to generate code for derivative computation. Operator overloading provides a similar mental programming model as normal computer programs [54], yielding code that is easier to interpret and debug than SCT. On the flip side, SCT has more potential to optimize the computational graph with global graph information. Consequently, SCT is generally the method of choice for AD libraries that aim to achieve high performance (e.g., [1]).

Our work on graph optimization builds on substantial efforts for optimization of computational graphs of tensor operations. Tensor contraction can be optimized via parallelization [22, 23, 41, 49], efficient transposition [51], blocking [10, 18, 28, 43], exploiting symmetry [15, 48, 49], and sparsity [22, 24, 32, 39, 39, 47]. For complicated tensor graphs, specialized compilers like XLA [52] and TVM [8] rewrite the computational graph to optimize program execution and memory allocation on dedicated hardware. For machine independent optimization, Grappler in TensorFlow [1] and TASO [20] use rule based symbolic substitution to simplify the execution flow. Classical compiler optimization also includes relevant techniques such as common subexpression elimination [3] are widely used as well [1, 12]. Previous work, such as Opt_einsum [46]

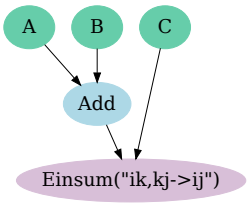


Figure 1: An example of a computational graph. We use green nodes to denote input variables, purple nodes to denote output nodes, and blue nodes to denote intermediate or constant nodes.

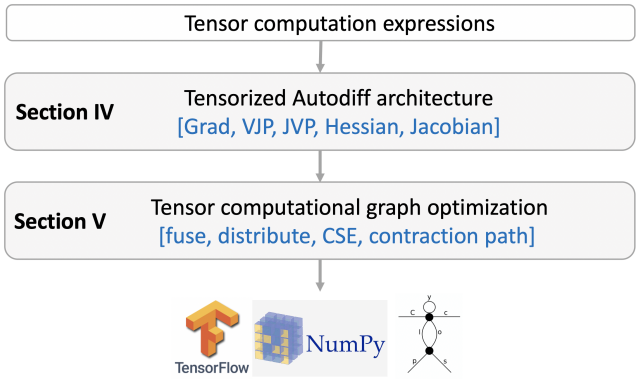


Figure 2: System overview of AutoHOOT. The arrows show the computation flow.

has yielded approaches for automatically determining efficient contraction orderings and selecting the best intermediates [5, 13–15]. The approaches generally rely on heuristic or exhaustive search to select a contraction path, as finding the optimal contraction order is NP-hard [9].

3 OVERALL ARCHITECTURE

The computations in AutoHOOT are described by *computational graphs*, which are directed graphs revealing the data dependency between different operations. Each *node* can be a source, intermediate or sink. Source / Sink nodes are inputs / outputs of the graph. Sink and intermediate nodes can be any mathematical computation, while input nodes are fed by the user or constants. An *edge* connecting two nodes represents the data dependency between them. An example of a computational graph is shown in Figure 1, where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are source nodes, the Einsum node is the sink, and the graph computes $(\mathbf{A} + \mathbf{B})\mathbf{C}$. We typically refer a node with its type, e.g., an Einsum node, which represents the tensor computations based on the Einstein summation convention. An *Einsum graph* is defined as a graph of nodes where all the nodes except the sources are Einsum nodes. An *Einsum tree* is defined as a tree of nodes where all the nodes except the sources are Einsum nodes.

AutoHOOT has two major components: an automatic differentiation architecture for tensor computations and a tensor computational graph optimizer. Figure 2 shows the system overview. For an input computation expression, the AD module will generate its tensorized differentiation expressions. Both the input expressions and the differentiation expressions will be optimized through the graph optimization module. With the optimized expressions, users have the choice to directly run the optimized expressions using the framework backends, including NumPy, TensorFlow and Cyclops, or to generate the Python source code through the source generation module.

Below we show an example to perform the CP decomposition based on alternating least squares using the framework. Rather than constructing each subproblem and building the dimension tree based algorithm manually, we only need to construct the updates of Newton's method for each subproblem, and the *optimize* function will reorganize the computational graph to minimize execution time automatically.

```
# construct input expressions
A, B, C, input_tensor, loss = cpd_graph(size, rank)

def update_site(site):
    hes = ad.hessian(loss, [site])
    grad, = ad.gradients(loss, [site])
    new_site = ad.tensordot(
        ad.tensorinv(hes[0][0]), grad)
    # return the optimized computational graph
    return optimize(new_site)
new_A = update_site(A)
new_B = update_site(B)
new_C = update_site(C)

# This executor is shared among all updates.
executor = ad.Executor([loss, new_A, new_B, new_C])
# ALS iterations
for i in range(num_iter):
    A_val = executor.run(feed_dict={
        input_tensor: input_tensor_val,
        A: A_val, B: B_val, C: C_val
    }, out=[new_A])
    B_val = executor.run(feed_dict={
        input_tensor: input_tensor_val,
        A: A_val, B: B_val, C: C_val
    }, out=[new_B])
    C_val = executor.run(feed_dict={
        input_tensor: input_tensor_val,
        A: A_val, B: B_val, C: C_val
    }, out=[new_C])
    loss_val = executor.run(feed_dict={
        input_tensor: input_tensor_val,
        A: A_val, B: B_val, C: C_val
    }, out=[loss])
```

In the AD module, we implement the reverse mode AD for first-order derivatives (Jacobian, VJP and JVP), as well as for higher-order derivatives, including Hessian and HVP. Both Jacobian and Hessian are formulated with a new algorithm, such that their calculations are not dependent on the JVP and HVP routines, which is more amenable to parallel execution as well as graph optimizations. We describe this approach in detail in Section 4.

The graph optimizer provides optimizations for tensor computational graphs. We adopt many machine independent optimization algorithms for common tensor computational graphs, such as selection of optimal contraction path and common sub-expression elimination. For second-order methods, the graph optimizer rewrites the structured inverse, such as the inverse of a Kronecker product, so that the inverses are operated on smaller tensors. For alternating methods, we developed a path selection algorithm with constraints

to construct the dimension trees. We describe this algorithm in detail in Section 5.

4 COMPUTATIONAL GRAPHS FOR HIGH-ORDER DERIVATIVES

We implement the reverse-mode AD based on the source code transformation (SCT) method, explicitly transforming the primal computation expression prior to execution to the adjoint expression. It allows us to flexibly perform the computational graph optimization after the adjoint expression production.

Our AD module supports the operations which calculate the Jacobian / Hessian expressions implicitly (VJP, JVP and HVP), and also explicit Jacobian and Hessian calculations. The implicit calculations are widely used in many other frameworks, because it is computationally cheaper. For example, for a Hessian matrix with size $n \times n$, explicitly forming the matrix costs $O(n^2)$, while the HVP calculation will only cost $O(n)$ leveraging the back-propagation gradient functions. For the explicit Jacobian and Hessian calculations, we introduce a new back-propagation algorithm that can produce a computational graph is more amenable to parallelization and downstream optimizations. The algorithm is detailed in Section 4.2.

4.1 VJP, JVP, and HVP

Our implementation of VJP is similar to many other frameworks [1, 7, 37], and is based on the reverse-mode AD. For functions involving matrix / vector operations whose inputs and outputs are both vectors,

$$\mathbf{x}_{i+1} = f_i(\mathbf{x}_i), i \in [1, \ldots, N],$$

consider a computational graph consisting of a chain of these functions,

$$\mathbf{y} = f(\mathbf{x}_1) = f_N \cdots f_1(\mathbf{x}_1),$$

the VJP adjoint of $\mathbf{x}_i$, $\mathbf{v}^T \mathbf{J}^{[f]}_{[\mathbf{x}_i]}$, is calculated based on the VJP adjoint of $\mathbf{x}_{i+1}$,

$$\text{VJP}(\mathbf{v}, f, \mathbf{x}_i) = \mathbf{v}^T \mathbf{J}^{[f]}_{[\mathbf{x}_i]} = (\mathbf{v}^T \mathbf{J}^{[f]}_{[\mathbf{x}_{i+1}]}) \mathbf{J}^{[f_i]}_{[\mathbf{x}_i]} = \text{VJP}(\mathbf{v}, f, \mathbf{x}_{i+1}) \mathbf{J}^{[f_i]}_{[\mathbf{x}_i]}.$$

Therefore, the VJP of all the inputs / intermediates $\mathbf{x}_i, i \in [1, \ldots, N]$ will be calculated with one backward propagation. It is also computationally efficient, because only matrix-vector product is necessary for each calculation.

Note that for the cases where sub function inputs and outputs contain matrices or tensors, VJP with reverse-mode AD is still valid and efficient, since we can think of each matrix or tensor as a reshaped vector. For the case where the output is a scalar, the gradient expression is implemented based on the VJP, if we fix the vector as a unit length vector with element being one.

Our JVP implementation is based on the VJP function[1]. Although it's more computationally efficient to implement JVP based on forward mode AD [6], we choose to implement it based on our reverse mode AD module, and optimize the computational graph afterwards to achieve computationally efficient expressions. The JVP implementation is based on calling the VJP function twice. First, we construct a function g, whose expression is as follows,

$$g(\mathbf{u}) = \text{VJP}(\mathbf{u}, f, \mathbf{x})^T = (\mathbf{u}^T \mathbf{J}^{[f]}_{[\mathbf{x}]})^T.$$

Afterwards, we perform another VJP operation on the function g with related to its input $\mathbf{u}$, and can get the JVP expression,

$$\text{VJP}(\mathbf{v}, g, \mathbf{u})^T = (\mathbf{v}^T \mathbf{J}^{[g]}_{[\mathbf{u}]})^T = (\mathbf{v}^T \mathbf{J}^{[f]T}_{[\mathbf{x}]})^T = \mathbf{J}^{[f]}_{[\mathbf{x}]} \mathbf{v} = \text{JVP}(\mathbf{v}, f, \mathbf{x}).$$

We also implement the HVP function based on the gradient function. We only consider the case when the function output is a scalar, because it is the general case where Hessian matrices are used. The HVP is formulated based on two gradient calculations, because HVP is equivalent to the gradient of the gradient-vector inner product. The expression is shown as follows,

$$\begin{aligned}\text{HVP}(\mathbf{v}, f, \mathbf{x}) = \mathbf{H}^{[f]}_{[\mathbf{x}]} \mathbf{v} &= \frac{\partial \mathbf{g}^{[f]}_{[\mathbf{x}]}}{\partial \mathbf{x}} \mathbf{v} = \frac{\partial \mathbf{g}^{[f]}_{[\mathbf{x}]}}{\partial \mathbf{x}} \mathbf{v} + \mathbf{g}^{[f]T}_{[\mathbf{x}]} \frac{\partial \mathbf{v}}{\partial \mathbf{x}} \\ &= \frac{\partial (\mathbf{g}^{[f]T}_{[\mathbf{x}]} \mathbf{v})}{\partial \mathbf{x}} = \text{grad}(\text{grad}(f, \mathbf{x})^T \mathbf{v}, \mathbf{x}).\end{aligned}$$

4.2 Explicit Jacobian and Hessian

To the best of our knowledge, all of the popular AD frameworks calculate explicit Jacobian and Hessian based on the VJP and HVP routines [1, 7, 37]. Taking the Jacobian calculation of

$$f(\mathbf{x}) = \mathbf{A}_1 \mathbf{A}_2 \mathbf{x}$$

as an example: when both $\mathbf{x}$ and $f(\mathbf{x})$ are of size n, current methods will compute the ith row of the Jacobian via VJP $\mathbf{e}_i^T \mathbf{J}^{[f]}_{[\mathbf{x}]}$ for $i \in \{1, \ldots, n\}$, where $\mathbf{e}_i$ is the ith elementary vector. There are two major disadvantages to this approach:

- It changes the BLAS-3 level matrix-matrix multiplications to multiple BLAS-2 level matrix-vector multiplications, and less flop intensity can be achieved. Although many frameworks provide the routine to compute all the matrix-vector multiplications in parallel, the parallelism is still sub-optimal and less efficient than the matrix multiplications, because the flop-to-byte ratio is $O(1)$ versus $O(n)$.
- The computational graph produced is difficult to optimize. Although having high dimensions, many Jacobians / Hessians in tensor computation operations are highly structured and the computational cost can be greatly reduced if being well optimized. However, calculating them based on matrix-vector products adds one more matrix-vector product operation, which usually break the structure and increase the cost. For example, if $\mathbf{A}_1 = \mathbf{B} \otimes \mathbf{C}$ and $\mathbf{A}_2 = \mathbf{D} \otimes \mathbf{E}$ and $\mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}$ have sizes $n \times n$, performing matrix-vector product for the Jacobian and each elementary vector costs $O(n^4)$ and the overall Jacobian calculation cost is $O(n^6)$. However, if we calculate the Jacobian directly, we can use the mixed-product property of the Kronecker product to optimize the expression,

  $$(\mathbf{B} \otimes \mathbf{C})(\mathbf{D} \otimes \mathbf{E}) = (\mathbf{BD}) \otimes (\mathbf{CE}),$$

  reducing the overall cost to $O(n^4)$.

To alleviate these disadvantages, we produce both Jacobian and Hessian expressions in a way that's independent of VJP and HVP routines.

[1]The JVP implementation is based on the technique introduced at https://j-towns.github.io/2017/06/12/A-new-trick.html.

For the Jacobian expression, our implementations are also based on the chain rule to perform back propagation, using

$$\text{Jacobian}(f, \mathbf{x}_i)=\mathbf{J}^{[f]}_{[\mathbf{x}_i]}=\mathbf{J}^{[f]}_{[\mathbf{x}_{i+1}]}\mathbf{J}^{[f_i]}_{[\mathbf{x}_i]}=\text{Jacobian}(f, \mathbf{x}_{i+1})\mathbf{J}^{[f_i]}_{[\mathbf{x}_i]}.$$

Therefore, the Jacobian of one target node is the matrix-matrix product between the Jacobian of its output node and the Jacobian of the local function. Note that when both $\mathbf{x}_i$ and the Jacobian have the tensor format, the above equation still holds, except that the matrix-matrix product is expressed in the form of tensor contractions (Einsums).

For linear operations, such as addition, subtraction, scalar-tensor multiplication and Einsum, we formulate the Jacobian expressions as an Einsum. To achieve that, we introduce the *Identity node*, which is a node that applies an identity matrix, to express the constraints in Jacobian tensors. For example, for the addition operations of two order N tensors,

$$f(\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}) = \boldsymbol{\mathcal{A}} + \boldsymbol{\mathcal{B}},$$

its Jacobian is a tensor of order $2N$, where $\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{A}}]}(x_1, \ldots, x_{2N}) = 1$ if and only if $x_i = x_{i+N}$ for $i \in \{1, \ldots, N\}$, and other elements are 0. This constraint can be easily specified with identity nodes. For the order 3 addition, the Jacobian of $\boldsymbol{\mathcal{A}}$ can be expressed as

$$\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{A}}]}(i, j, k, l, m, n) = \mathbf{I}(i, l)\mathbf{I}(j, m)\mathbf{I}(k, n).$$

Similarly, we can use the method to express the Jacobians for all the other linear operations. For example, for an Einsum expression below, its Jacobians are written as

$$f(\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}})(i, j, k) = \sum_l \boldsymbol{\mathcal{A}}(i, k, l)\boldsymbol{\mathcal{B}}(j, k, l),$$

$$\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{A}}]}(i, j, k, m, n, o) = \mathbf{I}(i, m)\mathbf{I}(k, n)\boldsymbol{\mathcal{B}}(j, n, o),$$

$$\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{B}}]}(i, j, k, m, n, o) = \mathbf{I}(j, m)\mathbf{I}(k, n)\boldsymbol{\mathcal{A}}(i, n, o).$$

Although we have introduced several identity nodes, they can be easily pruned so that only necessary identity nodes are left, which will be introduced in Section 5. The Hessian routines are based on the Jacobian routines: we perform Jacobian calculations twice to get the Hessian expressions. The advantage of this Jacobian / Hessian generation method is three-fold: first, we can leverage BLAS-3 level operations to perform most of the tensor contractions and can achieve higher performance. Second, the expressions are much easier to optimize, as will be introduced below. Third, the source code for Jacobian / Hessian expressions can be easily acquired, which is beneficial for both debugging and research purposes.

5 GRAPH OPTIMIZATIONS

We built a compiler to optimize tensor computational graphs. The compiler is specifically designed for tensor expressions with multilinear operations, including tensor contractions (Einsum) and linear algebra operations (addition, multiplication, summation, inversion and so on). Our goal is to reduce the computational cost by transforming the graph to an equivalent form. Given the fact that retrieving the optimal execution graph is NP-hard, we devise several application-driven heuristic strategies:

- *Generation of longer Einsum nodes:* To achieve this, we implement two kernels, Einsum distribution and Einsum fusion.
- *Symbolic rule execution:* We implement the structured inverse node decomposition and redundant node pruning kernels. In addition, we use SymPy [34] to simplify elementary algebraic operations.
- *Contraction order selection:* We select the contraction path on fully simplified expressions.
- *Constrained contraction path construction:* To accelerate alternating minimization, we provide a kernel to reuse intermediates between optimization subproblems.

Traditional compiler techniques, such as common sub-expressions elimination, are applied after the strategies above. The overall algorithm is described in Algorithm 1.

Algorithm 1: Graph optimization

```
input  : Input Graph: G
output : Optimized Graph: OG
G = FuseAllEinsum(Distribution(G))  ▷ Provide longer Einsums
G = SymbolicExecution(G)  ▷ Decompose Inverse / Prune identity / SymPy
G = OptContractPath(G)  ▷ Find efficient contraction order
OG = CSE(G)  ▷ Common Subexpression Elimination
return OG
```

5.1 Longer Einsum Nodes Generation

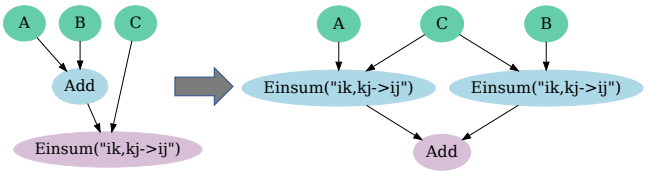
We aim to transform the computational graph into Einsum nodes with as many inputs as possible. This optimization will empower the contraction path selection with a global view and ease the discovery of optimizable patterns for downstream algorithms. To achieve this, we introduce two transformation kernels.

Einsum distribution. Einsum distribution recursively leverages distributivity of tensor contraction over tensor addition (or another distributive operation) to generate larger Einsum graphs. Larger Einsum graphs are the prerequisite for further graph depth reduction. This optimization moves the nodes performing the distributive operation (`dist_op`) closer to the graph sinks based on the programmatic rule below. Figure 3a illustrates the idea of an application of the algorithm. while the pseudo-code can be found in Appendix C of the accompanying technical report [30].

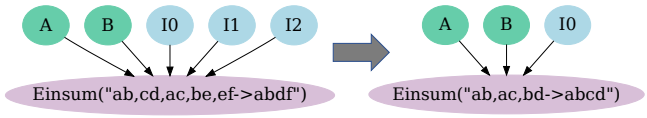
```
Einsum(dist_op(g1, g2), g3) =
dist_op(Einsum(g1, g3), Einsum(g2, g3))
```

Einsum fusion. Einsum fusion transforms an Einsum graph into several distinct Einsum nodes with the same set of source vertices (inputs) leveraging associativity of tensor contractions. It is a prerequisite for downstream graph optimization steps, such as contraction path selection and identity node pruning. An example can be seen in Figure 3c.

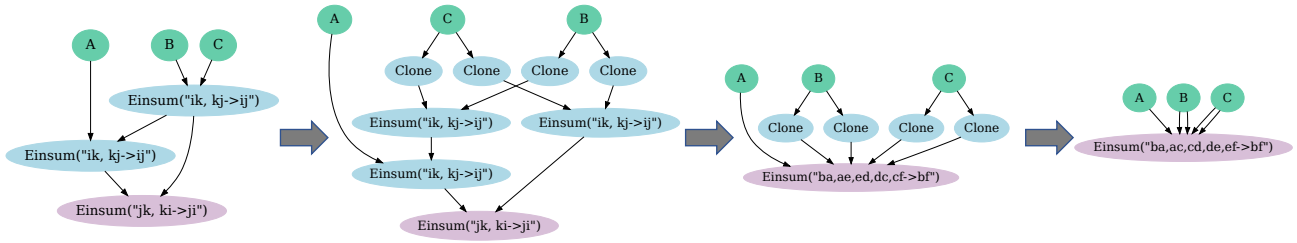
Einsum fusion has three steps: linearization of the graph, fusion of the generated Einsum Tree, and removal of the redundant clone nodes. The linearization step changes the input Einsum graph into an Einsum tree. When a source node is used in multiple Einsums, we create a clone of it for each Einsum. If an Einsum node has more than one output, we copy the subgraph defining its computation, including itself, and repeat until all nodes have a single output, yielding a forest (set of disconnected trees). The fusion step fuses



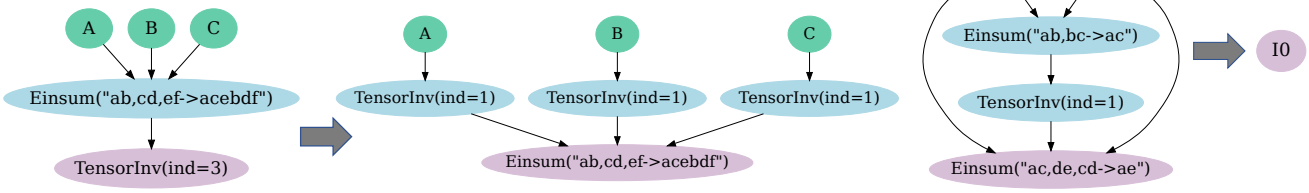
(a) Einsum distribution.



(b) Identity node pruning. The nodes whose name starts from "I" are identity nodes.

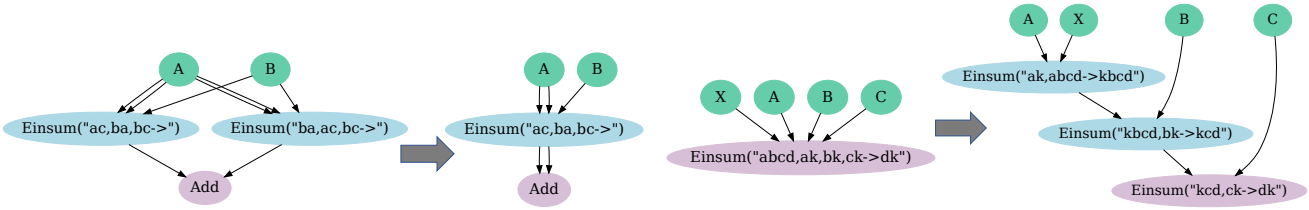


(c) Einsum fusion. It transforms an Einsum graph into one single Einsum node.



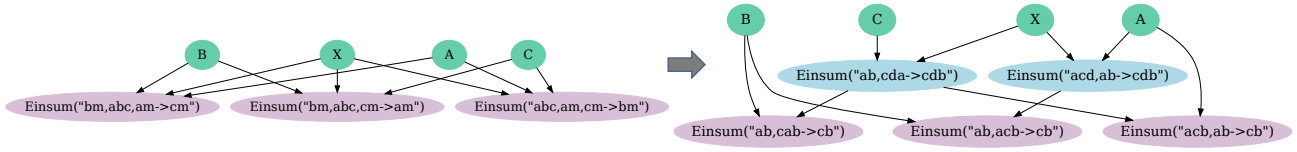
(d) Optimization of tensor inversion

(e) Inverse node pruning



(f) Common subexpression elimination

(g) Optimal contraction path



(h) Dimension tree generation

Figure 3: Visualization of different graph optimization kernels.

each generated Einsum tree. It leverages a union-find data structure, which puts two dimensions from two Einsum nodes into one set if they have the same subscript in one Einsum expression. After that, each disjoint set is assigned an unique character for the generation of the subscript of the new Einsum node. Finally, the clone node removal step removes the redundant clone nodes and returns an Einsum node. We illustrate both the pseudo-code sketch of the algorithm and the union-find data structure in Appendix C of the accompanying technical report [30].

5.2 Symbolic Execution

We employ several linear algebra constructs that can simplify the computational graph and reduce the computational cost.

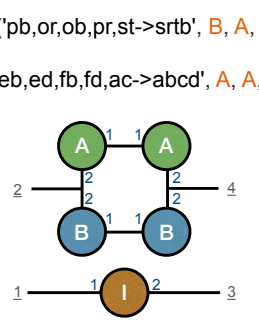


Figure 4: Tensor diagram of two Einsum expressions with the same tensor computations. The numbers around the input tensor denote the dimension numbers that are contracted by specific edges. The underlined numbers denote the dimension number of the output tensor. Two Einsum expressions with the same tensor diagram express the same tensor computations.

Structured Tensor inverse decomposition. An inverse of an Einsum graph may be the bottleneck of the computational graph because of the cubic order complexity. Fortunately, structured information may guide the optimization, e.g, the inverse of a Kronecker Product can be decomposed into the Kronecker product of inverses through $(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}$. We develop an algorithm to detect and break large tensor inverses into products of smaller tensor inverses so that the computation is cheaper. To keep it simple, the algorithm limits its applicability to specific forms of the tensors, and further details are described in Appendix B of the accompanying technical report [30]. An illustrative example is shown is Figure 3d.

Redundant node pruning. We prune the redundant nodes, including the Identity nodes and the inverse nodes, to simplify the expressions. Identity nodes are essential building blocks for the explicit Jacobian and the Hessian expressions, as is shown in Section 4.2. During the AD, redundant Identity nodes are introduced to aid the construction of the graph. Hence, we implement an algorithm to eliminate the unnecessary identity nodes afterwards for better efficiency. Identity nodes are removed unless they express necessary constraints in the output tensor structure, such as the tensor symmetry shown in the right graph of Figure 3b. In addition, we prune the unnecessary inverse nodes, as is shown in Figure 3e. When there exists an matrix multiplication between an Einsum Node and its corresponding inverse node, we directly return an identity node.

Elementary algebraic simplification. For elementary operations, such as addition, subtraction and multiplication, we use the SymPy library [34] to optimize them. SymPy can help us easily simplify the expressions. For the example shown below, it helps reducing the expression to one term.

```
sympy_simplify(
    (A-(((A*0.5)-(T*0.5))+((A*0.5)-(T*0.5))))
) = T
```

5.3 Optimized Contraction Path Selection

We identify the optimal contraction path for the Einsum expression after all the above transformations. For one Einsum node with multiple inputs, we provide an function to decompose it into an Einsum graph with the optimized contraction path, as is shown in Figure 3g. Our strategy is designed for the common tensor contractions with the following two assumptions:

- For simplicity, we only discuss the case where tensors are dense, and for a long Einsum expression with multiple inputs, it will first be split into multiple small Einsum expressions, each has only two inputs, and then dense tensor contractions will be executed.
- The chosen contraction path is hardware oblivious. We assume the contraction time for each operation is proportional to the flop counts. Other factors, such as the communication cost among different processes under the parallel execution settings, are not considered.

These assumptions allow us to implement the algorithm based on an interface provided by Opt_Einsum [46]. Note that whether we can find the optimal contraction path is based on the optimization algorithm, but we generally found that a greedy search algorithm is able to provide an optimal path for most of the Einsum expressions in tensor computation applications.

In addition, the assumptions above are not limitations of our overall approach. AutoHOOT is also capable of extracting the contraction path based on other libraries, such as Cyclops [60], where hardware and tensor sparsity are considered in the algorithm.

5.4 Constrained Contraction Path Construction

We provide a constrained contraction path selection routine, such that the contraction path is optimized under the constraint that partial inputs' contraction order is fixed. This routine is critical for the dimension tree construction used in the alternating minimization algorithms. Consider Equation 3, with the update sequence in each sweep starting from $\boldsymbol{\mathcal{A}}_1$ and ending at $\boldsymbol{\mathcal{A}}_N$, for the Einsum node used to update $\boldsymbol{\mathcal{A}}_i$, where $i \in \{1, \ldots, N\}$, we generate the contraction path such that it is optimized under the constraint that the contraction order for all the target sites is $\boldsymbol{\mathcal{A}}_N \prec \cdots \prec \boldsymbol{\mathcal{A}}_{i+1} \prec \boldsymbol{\mathcal{A}}_1 \prec \cdots \prec \boldsymbol{\mathcal{A}}_{i-1}$. This order ensures that the tensor that is updated just previously, $\boldsymbol{\mathcal{A}}_{i-1}$, affects only the last part of the contraction path, enabling the reuse of the calculations prior to it in the path as much as possible.

The constrained path selection algorithm is illustrated in Algorithm 2, and is implemented on top of the unconstrained one and uses the greedy search heuristic. We find that this heuristic works well for all the dimension tree selection in the tensor computation applications tested in Section 6. An example is shown in Figure 3h, which illustrate the dimension construction for the Matricized Tensor Times Khatri-Rao Product (MTTKRP) calculations of an order 3 CP decomposition. The pseudo-code is illustrated in Appendix C of the accompanying technical report [30].

5.5 Common Subexpression Elimination (CSE)

CSE is used to remove the duplicated Einsum expressions generated from the path selection above. We show one example in Figure 3f, where CSE helps saving one Einsum calculation. However, CSE is

CPD Kernel	Size (s)	Backend	Backend AD	AD	AD + OPT1	AD + OPT1,2	AD + OPT1,2,3	Overall speed-up
GN Jacobian	25	JAX	0.1449s	0.0632s	0.0126s	0.0126s	0.0126s	11X
		TensorFlow	1.5201s	0.1037s	0.0029s	0.0029s	0.0029s	524X
GN HVP	40	JAX	0.0107s	0.0011s	0.0012s	0.0011s	0.0011s	9X
		TensorFlow	0.0040s	0.0027s	0.0048s	0.0048s	0.0048s	0.8X
	640	JAX	0.3742s	0.776s	0.0056s	0.0054s	0.0051s	73X
		TensorFlow	0.9669s	0.9746s	0.4470s	0.3422s	0.2795s	3X
ALS Hessian	40	JAX	0.0713s	OOM	0.0017s	0.0017s	0.0017s	41X
		TensorFlow	0.3643s	OOM	0.0021s	0.0021s	0.0014s	260X
	160	JAX	OOM	OOM	1.0682s	1.0682s	0.8141s	/
		TensorFlow	OOM	OOM	3.0164s	3.0164s	1.5405s	/
ALS Hessian inv	40	JAX	0.1623s	OOM	0.0908s	0.0090s	0.0090s	18X
		TensorFlow	0.4237s	OOM	0.0278s	0.0028s	0.0028s	151X
	160	JAX	OOM	OOM	13.13s	1.5160s	1.5110s	/
		TensorFlow	OOM	OOM	OOM	0.5786s	0.5585s	/

Table 1: Detailed performance gain from each graph optimization technique on different CPD kernels. The rank is set the same as the input tensor dimension/size along each mode (s). Results are collected on an NVIDIA Titan X GPU. We denote each technique as: Einsum fusion + distribution: OPT1, Symbolic optimization: OPT2, CSE: OPT3.

Algorithm 2: Opt_contraction_path_w_constraint

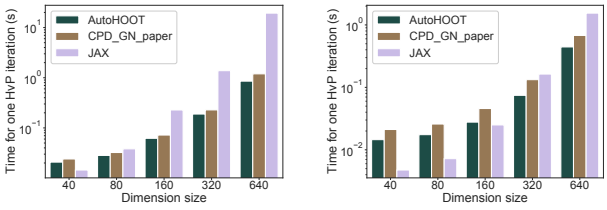
input : Einsum Node: N, Contraction order list: L
output: Einsum Tree: T
n = length(L)
T = N ▷ Initialize tree with single Einsum node
for $i \in \{1, \ldots, n\}$ **do**
  split_T = `SplitEinsum` (T, L[i+1:n]) ▷ Split T into an Einsum node that contracts all input nodes apart from L[i+1:n] and the subgraph induced by the remaining nodes, returning the former
  opt_contract_subtree = `OptContractPath` (split_T) ▷ Unconstrained optimized contraction path
  opt_contract_subtree = `Get_nearest_ancestor` (opt_contract_subtree, L[i]) ▷ Get the tree whose sink is the nearest ancestor of L[i]
  T = `Substitute_graph` (T, opt_contract_subtree) ▷ Return the equivalent graph of T whose inputs contain opt_contract_subtree
return T

nontrivial for Einsum nodes because different Einsum subscripts may represent the same computation. We show an example in Figure 4 where two Einsum nodes represent the same calculation despite different input ordering and subscripts. Hence, we transfer an Einsum expression into a tensor diagram graph, and compare the graph structures between two expressions.

Moreover, two nodes in an Einsum graph may be transpositions of each other. After detecting such conditions, we replace one of the nodes with its transpose node and update its outputs' expressions therein. This optimization greatly reduces the computation cost when transposes of large tensors appear in the graph.

6 BENCHMARKS

We evaluate the performance of AutoHOOT on both the Gauss-Newton method and the alternating minimization method discussed in Section 2.2. The performance of the critical Gauss-Newton kernel,



(a) KNL CPU (b) TESLA K80 GPU

Figure 5: Performance comparison among AutoHOOT, JAX and the existing implementation for the HVP kernel in the Gauss-Newton algorithm for the CP decomposition. The implementation of CPD_GN_paper comes from reference [45]. The tensor order is set as $N = 3$, and the CP rank is set equal to the dimension size. Each bar is the average result of 10 iterations.

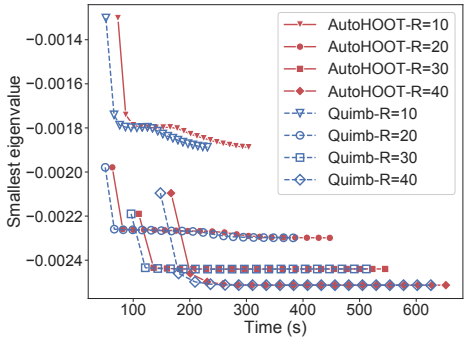
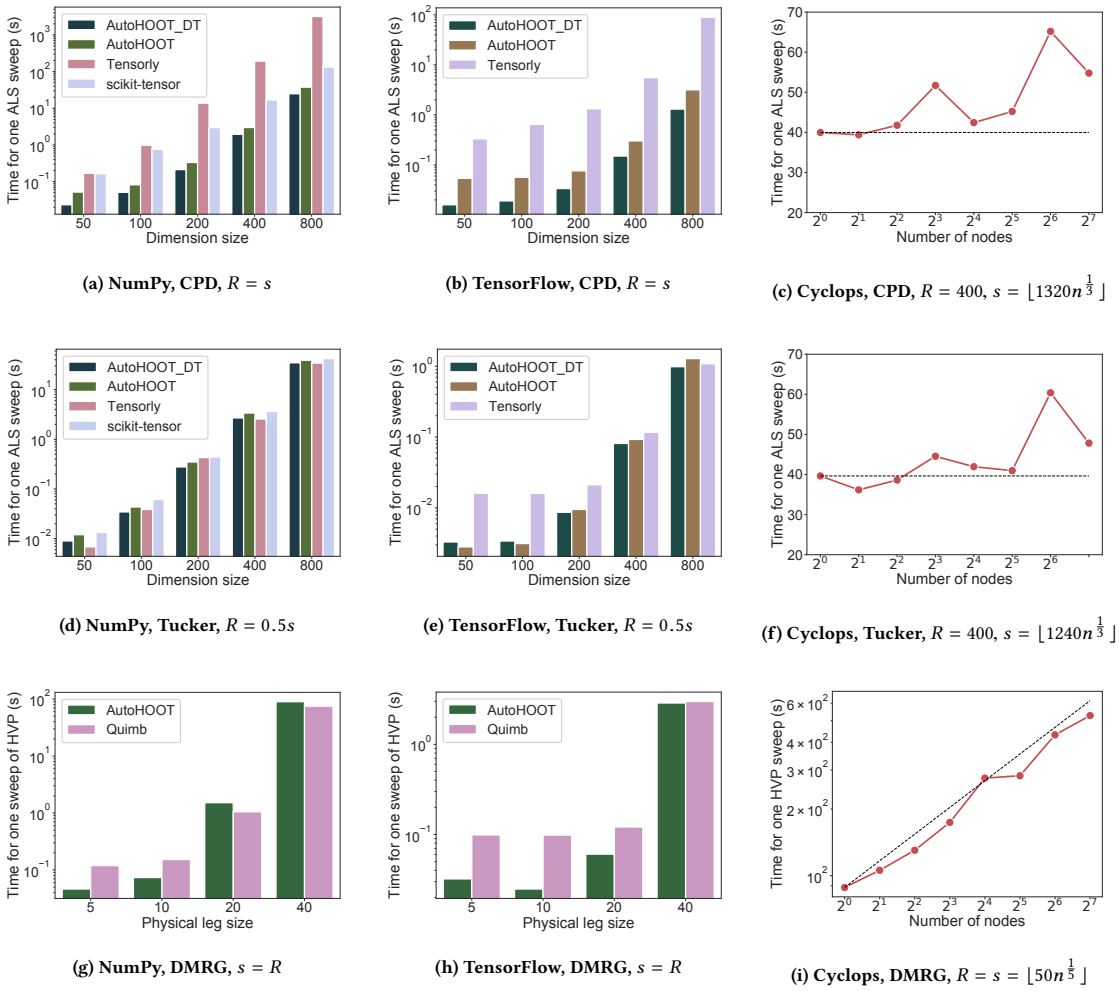


Figure 6: Comparison between AutoHOOT and Quimb on the full DMRG running curve. The input MPO is random and symmetric, has 6 sites, and its physical leg size equals 10 and MPO rank size equals 20. We compare the performance under different largest MPS rank constraints.



(a) **NumPy, CPD,** $R = s$ (b) **TensorFlow, CPD,** $R = s$ (c) **Cyclops, CPD,** $R = 400$, $s = \lfloor 1320n^{\frac{1}{3}} \rfloor$

(d) **NumPy, Tucker,** $R = 0.5s$ (e) **TensorFlow, Tucker,** $R = 0.5s$ (f) **Cyclops, Tucker,** $R = 400$, $s = \lfloor 1240n^{\frac{1}{3}} \rfloor$

(g) **NumPy, DMRG,** $s = R$ (h) **TensorFlow, DMRG,** $s = R$ (i) **Cyclops, DMRG,** $R = s = \lfloor 50n^{\frac{1}{5}} \rfloor$

Figure 7: AutoHOOT performance for kernels in the alternating minimization. TensorFlow results are collected on an NVIDIA TESLA K80 GPU. (a)-(c): Results for the CP decomposition. The tensor order is set as $N = 3$ for all the experiments. (d)-(f): Results for the Tucker decomposition. The tensor order is set as $N = 3$ for all the experiments. (g)-(i): Results for the DMRG experiment. The number of sites is set as $N = 10$ for the experiments with NumPy and TensorFlow, and set as $N = 6$ for the experiments with Cyclops. For the Cyclops benchmark, the dotted line denotes the perfect scaling curve. Each bar/dot is the average result of 10 iterations.

the Hessian-Vector Product, is evaluated on the CP decomposition application, where Gauss-Newton with conjugate gradient update is commonly used to achieve high accuracy [45, 50]. The performance of alternating minimization kernels generated by AutoHOOT is evaluated on both CP and Tucker decompositions, as well as the DMRG algorithm in tensor network applications used to calculate the smallest eigenvalue and eigenvector for a matrix product state.

The experiments are run on both CPUs and GPUs. On CPUs, we test the performance on both one process with the NumPy backend, and on the distributed parallel system with the Cyclops backend. The results are collected on the Stampede2 supercomputer located at the University of Texas at Austin. We leverage the Knight's Landing (KNL) nodes, each of which consists of 68 cores, 96 GB of DDR RAM, and 16 GB of MCDRAM. These nodes are connected via a 100 Gb/sec fat-tree Omni-Path interconnect. We use Intel compilers and the MKL library for threaded BLAS routines for both sequential and parallel experiments. We use 16 processes per node and 16 threads per process for the Cyclops benchmark experiments. We also collected results with both TensorFlow and JAX backends on both single NVIDIA TESLA K80 GPU and single NVIDIA Titan X GPU.

We first compare the detailed performance gain from each graph optimization technique proposed in Section 5. The experiments are performed on the Jacobians and HVPs kernels in the Gauss-Newton (GN) methods, as well as Hessians and Hessian inverses used in the ALS algorithm for CP decompositions and are shown in Table 1. As

can be seen in the table, Einsum fusion and distribution are critical for almost all the calculations, and Symbolic optimization is critical for tensor/matrix inverse. In addition, CSE provides incremental performance gain.

The performance of the HVP kernels in the Gauss-Newton algorithm for the CP decomposition is shown in Figure 5. As can be seen, AutoHOOT has at least 2X speed-up on the GPU and at least 7X speed-up on the CPU compared to JAX when the dimension size $s \geq 320$. Note that JAX performs better for small HVP kernels, because the experiments with AutoHOOT are performed on TensorFlow, where JAX has faster small contractions. It can be seen that the speed-up increases with the increase of the dimension size, indicating the advantage of AutoHOOT for large scale tensor computations. In addition, the AutoHOOT performance is comparable compared to the manually designed algorithms in the reference [45], indicating that the kernels generated by AutoHOOT reaches the state-of-art performance boundary.

The performance of the alternating minimization kernels for both tensor decompositions and the DMRG algorithm are shown in Figure 7. For the tensor decompositions, we compare the performance of AutoHOOT output expressions, both with and without dimension tree optimizations, to the popular tensor decomposition libraries Tensorly [26], both with NumPy and TensorFlow backend, and scikit-tensor[2] with NumPy backend. For the DMRG algorithm, we compare the performance to Quimb [11], which is an efficient library for tensor networks.

The benchmark results for the CP decomposition with both NumPy and TensorFlow can be seen in Figure 7a, 7b. We compare the performance with different CP ranks (R) and dimension size (s). As can be seen, the expressions generated with the dimension tree algorithm outperform all the other implementations. Note that Tensorly's performance is not as expected for the CP decomposition, because it slices the factor matrices over the rank mode and sums over all the MTTKRP results of the input tensor and the sliced factor matrices, which is not favorable. The weak scaling benchmark is also performed on the distributed parallel system with Cyclops, shown in Figure 7c, where we consider weak scaling with fixed input size and work per processor. The expressions generated from AutoHOOT scale well, obtaining 73% parallel scaling efficiency on 128 nodes (2048 cores).

The benchmark results for the Tucker decomposition with both NumPy and TensorFlow can be seen in Figure 7d, 7e. We compare the performance with different Tucker ranks (R) and dimension size (s). Note that we are only comparing the performance of the kernel generated through AutoHOOT to the Tensor Times Matrix-chain (TTMc) implementation in other libraries, which doesn't contain the low rank factorization step of splitting the factor matrix from the core tensor. The expressions generated with the dimension tree algorithm is comparable to all the other implementations. The weak scaling benchmark is shown in Figure 7f. Similar to the CP decomposition, the expressions generated from AutoHOOT scale with high efficiency.

The performance results for DMRG can be seen in Figure 7g, 7h, 7i. We benchmark over sweep of the HVP kernels with different MPO and MPS rank size (R) and physical dimension size (s), where the Hessian denotes the local Hessian of the DMRG loss function w.r.t. each local site. In DMRG, the HVP calculations are important kernels for the sparse eigensolver. Multiple HVP calculations are necessary for each site to get the local smallest eigenvalue, making it the computation bottleneck. The expressions generated with the dimension tree algorithm achieve comparable performance to the implementations in Quimb. In addition, the expressions generated from AutoHOOT scale nearly perfectly with Cyclops up to at least 128 nodes[3].

We also compare the performance between AutoHOOT and Quimb on the full DMRG experiments. Like Quimb, we use the sparse eigensolver in SciPy [58], and set the solver parameters the same as Quimb. The results are shown in Figure 6. We test the four cases where the maximum MPS rank ranges from 10 to 40, and the results show that both libraries have the similar performance, while AutoHOOT has a small fixed overhead.

Note that we did not report the ALS results of other AD libraries, because their performance is far worse than both AutoHOOT and other tensor computation libraries. For both CP and Tucker decompositions, existing AD libraries cannot efficiently decompose the structured inverse operations, leading to a big overhead from inverting large tensors. For the DMRG experiment, existing libraries fail to choose an optimized contraction path, and produce large intermediates which require too much memory.

7 CONCLUSION

AutoHOOT is the first automatic differentiation framework targeting high-order optimization for tensor computations. AutoHOOT contains a new explicit Jacobian / Hessian expression generation kernel whose outputs keep the input tensors' granularity and are easy to optimize. It also contains a new computational graph optimization module that combines both the traditional optimization techniques for compilers and techniques based on specific tensor algebra. The optimization module generates expressions as good as manually written codes in other frameworks for the numerical algorithms of tensor computations. AutoHOOT is compatible with other numerical computation libraries, and users can execute the generated expressions on CPU with NumPy, GPU with TensorFlow, and distributed parallel systems with Cyclops Tensor Framework. Experimental results show that AutoHOOT has competitive performance on both tensor decomposition and tensor network applications compared to both existing AD software and other tensor computation libraries with manually written kernels, both on CPU and GPU architectures.

8 ACKNOWLEDGEMENTS

Linjian Ma and Edgar Solomonik were supported by the US NSF OAC SSI program, award No. 1931258. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. We used XSEDE to employ Stampede2 at the Texas Advanced Computing Center (TACC) through allocation TG-CCR180006.

[2] https://github.com/mnick/scikit-tensor

[3] In these experiments, we constrain the physical leg size to be equal to the rank, e.g. $s = R$, so the computational cost is $O(R^7)$ and the memory footprint is $O(R^5)$.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] E. Acar, D. M. Dunlavy, and T. G. Kolda. A scalable optimization approach for fitting Canonical tensor decompositions. *Journal of Chemometrics*, 25(2):67–86, 2011.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9.

[4] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky. Tensor decompositions for learning latent variable models. *Journal of Machine Learning Research*, 15:2773–2832, 2014.

[5] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, et al. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[6] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1):5595–5637, 2017.

[7] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[8] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, Oct. 2018. USENIX Association.

[9] L. Chi-Chung, P. Sadayappan, and R. Wenger. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters*, 7(02):157–168, 1997.

[10] J. Choi, X. Liu, S. Smith, and T. Simon. Blocking optimization techniques for sparse tensor computation. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 568–577. IEEE, 2018.

[11] J. Gray. quimb: a Python library for quantum information and many-body calculations. *Journal of Open Source Software*, 3(29):819, 2018.

[12] A. Hartono, Q. Lu, X. Gao, S. Krishnamoorthy, M. Nooijen, G. Baumgartner, D. E. Bernholdt, V. Choppella, R. M. Pitzer, J. Ramanujam, et al. Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations. In *International Conference on Computational Science*, pages 267–275. Springer, 2006.

[13] A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, et al. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *The Journal of Physical Chemistry A*, 113(45):12715–12723, 2009.

[14] A. Hartono, A. Sibiryakov, M. Nooijen, G. Baumgartner, D. E. Bernholdt, S. Hirata, C.-C. Lam, R. M. Pitzer, J. Ramanujam, and P. Sadayappan. Automated operation minimization of tensor contraction expressions in electronic structure calculations. In *International Conference on Computational Science*, pages 155–164. Springer, 2005.

[15] S. Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[16] E. G. Hohenstein, R. M. Parrish, and T. J. Martínez. Tensor hypercontraction density fitting. I. quartic scaling second-and third-order Møller-Plesset perturbation theory. *The Journal of chemical physics*, 137(4):044103, 2012.

[17] F. Hummel, T. Tsatsoulis, and A. Grüneis. Low rank factorization of the Coulomb integrals for periodic coupled cluster theory. *The Journal of chemical physics*, 146(12):124105, 2017.

[18] K. Z. Ibrahim, S. W. Williams, E. Epifanovsky, and A. I. Krylov. Analysis and tuning of libtensor framework on multicore architectures. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[19] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[20] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[21] P. Jørgensen and J. Simons. *Geometrical derivatives of energy surfaces and molecular properties*, volume 166. Springer Science & Business Media, 2012.

[22] D. Kats and F. R. Manby. Sparse tensor framework for implementation of general local correlation methods. *The Journal of Chemical Physics*, 138(14):–, 2013.

[23] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, et al. High performance computational chemistry: An overview of NWChem a distributed parallel application. *Computer Physics Communications*, 128(1-2):260–283, 2000.

[24] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[25] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[26] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic. Tensorly: Tensor learning in Python. *The Journal of Machine Learning Research*, 20(1):925–930, 2019.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[28] J. Li, J. Sun, and R. Vuduc. HiCOO: hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 238–252. IEEE, 2018.

[29] L. Ma and E. Solomonik. Accelerating alternating least squares for tensor decomposition by pairwise perturbation. *arXiv preprint arXiv:1811.10573*, 2018.

[30] L. Ma, J. Ye, and E. Solomonik. AutoHOOT: Automatic High-Order Optimization for Tensors. *arXiv preprint arXiv:2005.04540*, 2020.

[31] D. Maclaurin, D. Duvenaud, and R. P. Adams. Autograd: Effortless gradients in NumPy.

[32] S. Manzer, E. Epifanovsky, A. I. Krylov, and M. Head-Gordon. A general sparse tensor framework for electronic structure theory. *Journal of chemical theory and computation*, 13(3):1108–1116, 2017.

[33] I. L. Markov and Y. Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008.

[34] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, et al. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, 2017.

[35] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov. Tensorizing neural networks. In *Advances in neural information processing systems*, pages 442–450, 2015.

[36] R. Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[38] W. Pazner and P.-O. Persson. Approximate tensor-product preconditioners for very high order discontinuous Galerkin methods. *Journal of Computational Physics*, 354:344–369, 2018.

[39] C. Peng, J. A. Calvin, F. Pavosevic, J. Zhang, and E. F. Valeev. Massively parallel implementation of explicitly correlated coupled-cluster singles and doubles using TiledArray framework. *The Journal of Physical Chemistry A*, 120(51):10231–10244, 2016.

[40] A.-H. Phan, P. Tichavskỳ, and A. Cichocki. Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations. *IEEE Transactions on Signal Processing*, 61(19):4834–4846, 2013.

[41] S. Rajbhandari, A. Nikam, P.-W. Lai, K. Stock, S. Krishnamoorthy, and P. Sadayappan. A communication-optimal framework for contracting distributed tensors. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 375–386. IEEE, 2014.

[42] C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, and S. Leichenauer. Tensornetwork: A library for physics and machine learning. *arXiv preprint arXiv:1905.01330*, 2019.

[43] R. Senanayake, F. Kjolstad, C. Hong, S. Kamil, and S. Amarasinghe. A unified iteration space transformation framework for sparse and dense tensor algebra, 2019.

[44] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing*, 65(13):3551–3582.

[45] N. Singh, L. Ma, H. Yang, and E. Solomonik. Comparison of accuracy and scalability of Gauss-Newton and alternating least squares for CP decomposition. *arXiv preprint arXiv:1910.12331*, 2019.

[46] D. Smith and J. Gray. opt_einsum - a Python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software*, 3(26):753, 2018.

[47] 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*, IA3 '15, New York, NY, USA, 2015. Association for Computing Machinery.

[48] E. Solomonik and J. Demmel. Fast bilinear algorithms for symmetric tensor contractions. *Computational Methods in Applied Mathematics*, 2020.

[49] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[50] L. Sorber, M. Van Barel, and L. De Lathauwer. Optimization-based algorithms for tensor decompositions: canonical polyadic decomposition, decomposition in

rank-(l_r,l_r,1) terms, and a new generalization. *SIAM Journal on Optimization*, 23(2):695–720, 2013.

[51] P. Springer, T. Su, and P. Bientinesi. HPTT: A High-Performance Tensor Transposition C++ Library. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2017, pages 56–62, New York, NY, USA, 2017. ACM.

[52] X. Team et al. XLA-TensorFlow compiled. post in the Google developers blog, 2017.

[53] P. Tichavskỳ, A. H. Phan, and A. Cichocki. A further improvement of a fast damped Gauss-Newton algorithm for CANDECOMP-PARAFAC tensor decomposition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5964–5968. IEEE, 2013.

[54] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2002–2011, 2019.

[55] B. van Merrienboer, D. Moldovan, and A. Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems*, pages 6256–6265, 2018.

[56] N. Vannieuwenhoven, K. Meerbergen, and R. Vandebril. Computing the gradient in optimization algorithms for the CP decomposition in constant memory through tensor blocking. *SIAM Journal on Scientific Computing*, 37(3):C415–C438, 2015.

[57] F. Verstraete, J. J. Garcia-Ripoll, and J. I. Cirac. Matrix product density operators: simulation of finite-temperature and dissipative systems. *Physical review letters*, 93(20):207204, 2004.

[58] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[59] S. F. Walter and L. Lehmann. Algorithmic differentiation in Python with AlgoPy. *Journal of Computational Science*, 4(5):334–344, 2013.

[60] Z. Zhang, X. Wu, N. Zhang, S. Zhang, and E. Solomonik. Enabling distributed-memory tensor completion in Python using new sparse tensor kernels. *arXiv preprint arXiv:1910.02371*, 2019.