AutoHOOT: Automatic High-Order Optimization for Tensors

Linjian Ma[1], Jiayu Ye[2] and Edgar Solomonik[1]

L P N A @CS@Illinois

[1]Department of Computer Science
University of Illinois at Urbana-Champaign
[2]Google.Inc

PACT 2020

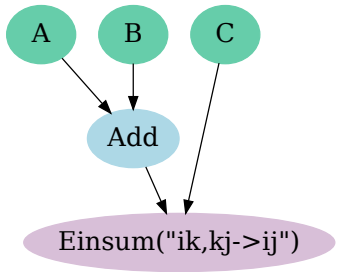
Outline

Introduction

- Tensors are important in both scientific computing and data science
  - A tensor is a multidimensional array of data
  - Convolution is an important kernel in signal processing and neural networks
  - We will focus on applications involving tensor decomposition/networks
- Tensor decompositions: approximate a tensor as a contraction of smaller ones (similar to low-rank matrix factorization)
  - Widely used in data analytics and quantum chemistry
- Tensor networks: seek to solve eigenvalue/optimization problems with a tensor that is already decomposed
  - Widely used in quantum chemistry and physics
- Derivatives are widely used in the optimization algorithms for tensor related problems

Introduction

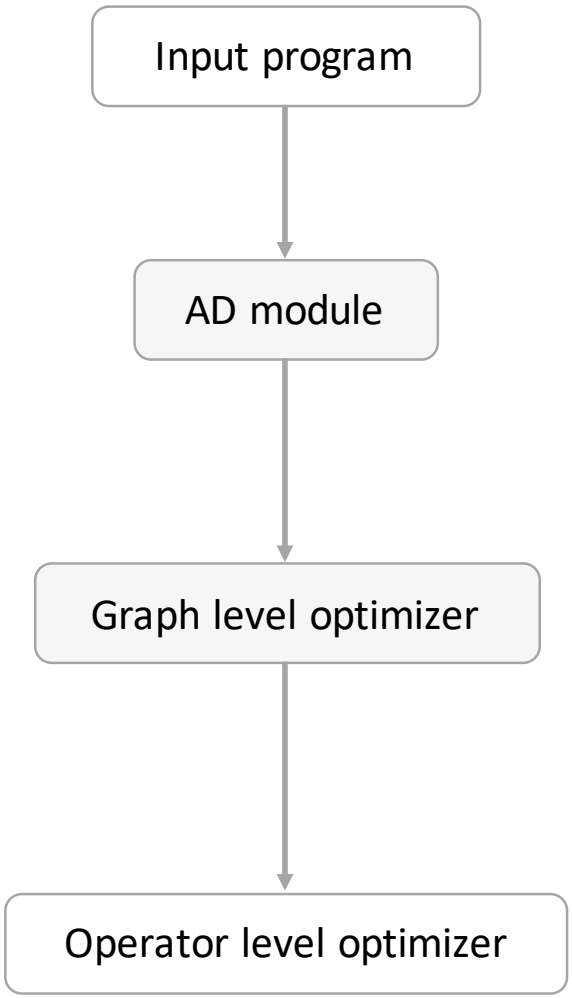
- Automatic differentiation (AD): automatically constructing a computational graph (program) for computing derivatives of an input graph
  - Apply symbolic differentiation at the elementary operation level (add, matmul, inverse)
  - Use chain rule to get overall differentiation
  - TensorFlow, PyTorch, JAX, Chainer...



- Nodes: variables / constants / operations
- Edges: data dependency

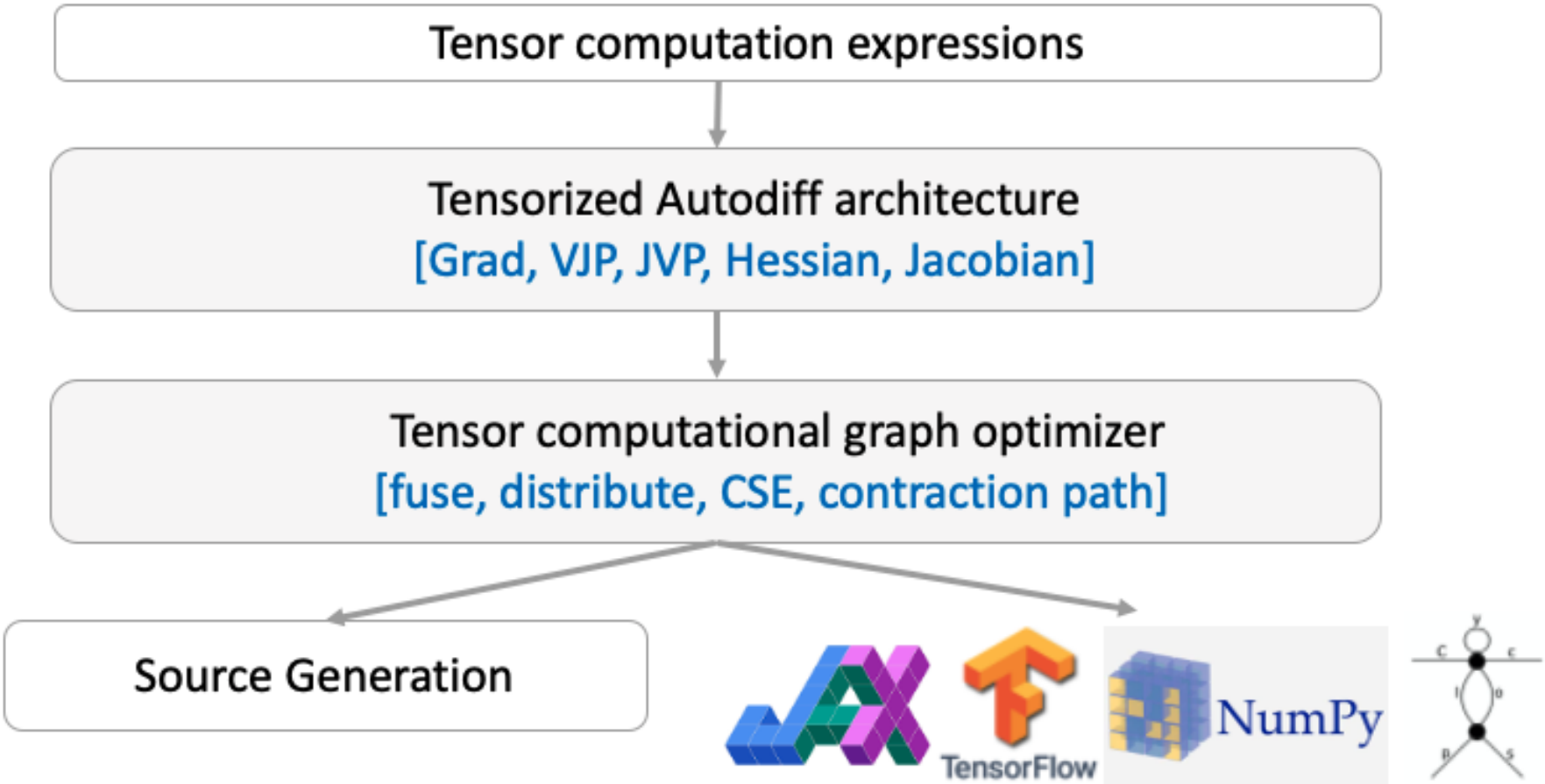
- Graphs of tensor computation applications contain a lot of
  - Einsum: include most of the tensor / matrix / vector operations
  - Distributive operations: add, sub
  - Tensor / matrix inverse

Motivation for a new AD framework



- Optimization of tensor computation applications involve
  - Structured high-order derivatives
  - A lot of multilinear operations and a small number of nonlinear operations
- Necessity for a new AD framework with
  - A graph optimizer with tensor-algebra specific transformation algorithms
  - An AD module to generate efficient representations for higher-order derivatives

AutoHOOT overview

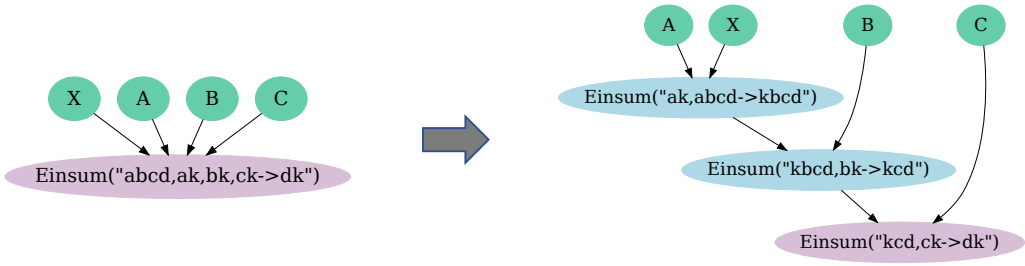


AutoHOOT novelty

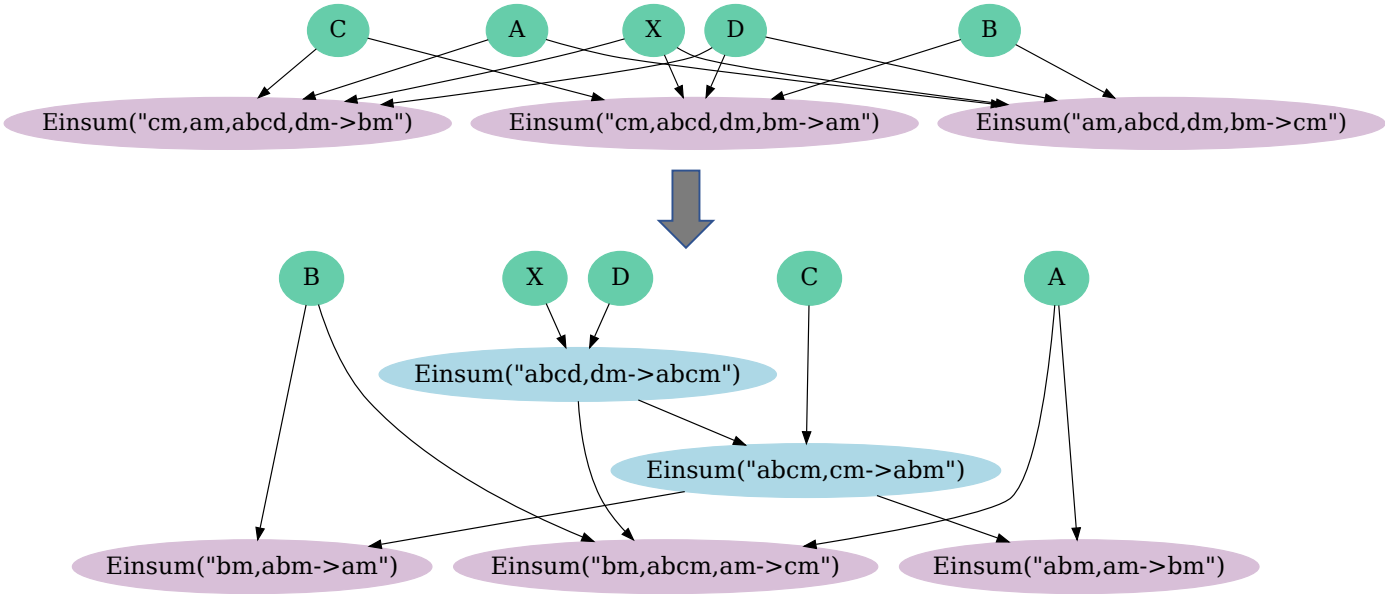
	AutoHOOT	TensorFlow, PyTorch, JAX
Graph optimizer	✓ Includes both traditional graph optimization techniques and tensor-algebra specific transformations	Includes traditional graph optimization techniques
AD module	✓ Output graph of high-order derivatives keeps the tensor structure granularity and easier to optimize	Output graph of high-order derivatives is relatively hard to optimize
Performance on tensor decomposition/network applications	✓ Comparable, sometimes better performance compared to existing tensor frameworks with manually implemented derivatives	1. Much slower than existing tensor frameworks 2. Sometimes have OOM issues

Kernels in the graph optimizer

- Optimized contraction path

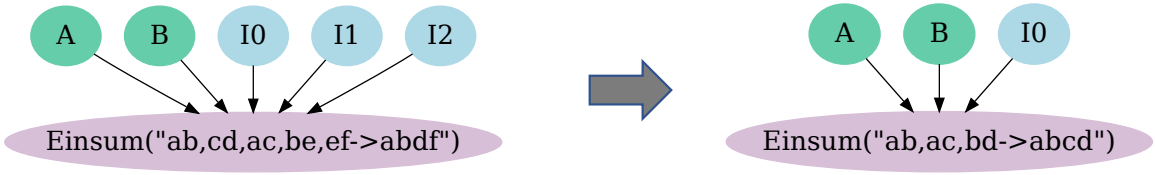


- Constrained contraction path
  (example: with the contraction order $D \prec C \prec B \prec A$)



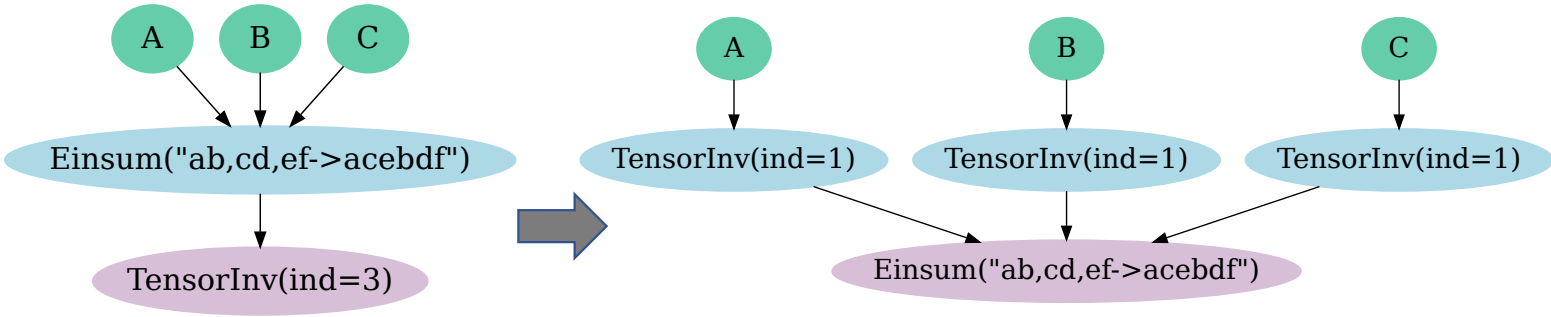
Kernels in the graph optimizer

- Redundant node pruning



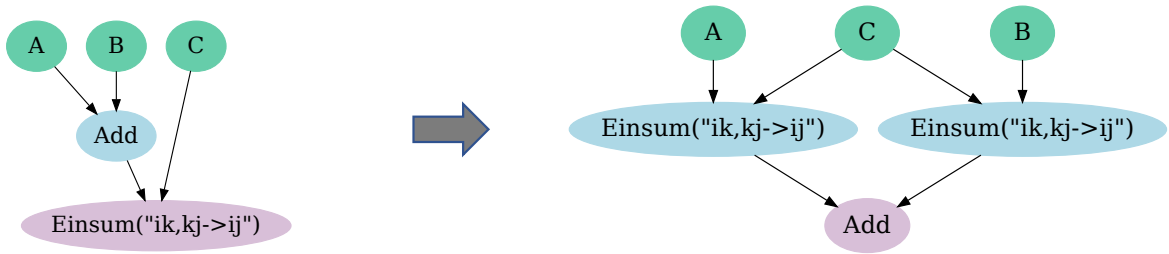
- Optimization of tensor inverse

$(A \otimes B)^{-1} \rightarrow A^{-1} \otimes B^{-1}$, where $A \otimes B := \begin{pmatrix} a_{1,1}B & \cdots & a_{1,n}B \\ \vdots & \ddots & \vdots \\ a_{m,1}B & \cdots & a_{m,n}B \end{pmatrix}$

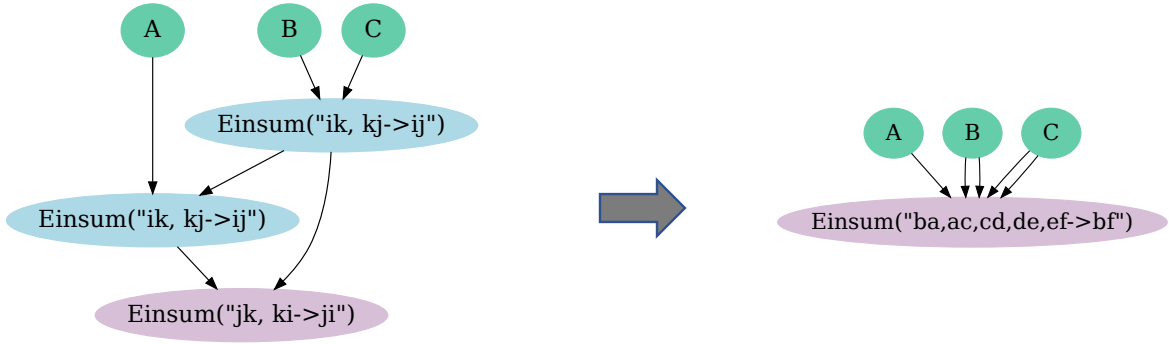


Kernels in the graph optimizer

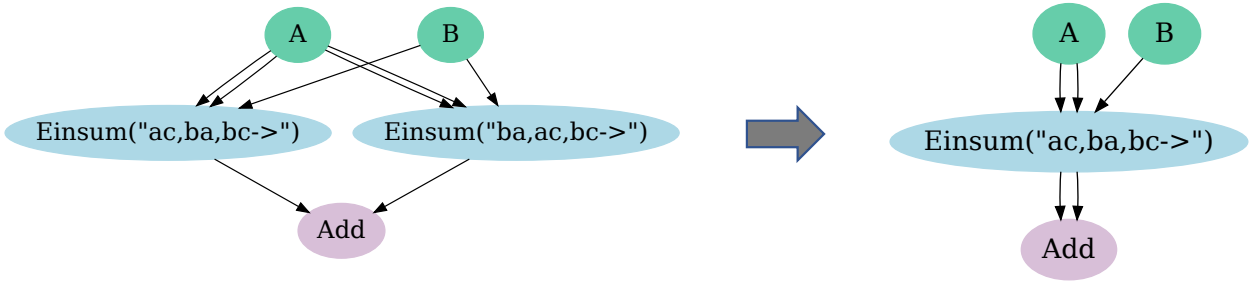
- Einsum distribution ($A(B+C) = AB + AC$)



- Einsum fusion



- Common subexpression elimination (CSE)



Automatic differentiation module

	AutoHOOT	TensorFlow, PyTorch, JAX
AD module	✓ Output graph of high-order derivatives keeps the tensor structure granularity and easier to optimize	Output graph of high-order derivatives is relatively hard to optimize

Consider a simple computational graph:

$$\boldsymbol{x}_{N+1} = f(\boldsymbol{x}_1) = f_N \cdots f_1(\boldsymbol{x}_1), \quad \boldsymbol{x}_{i+1} = f_i(\boldsymbol{x}_i), i \in [1, \ldots, N], \quad \text{where } \boldsymbol{x}_i \in \mathbb{R}^{s_i}$$

- Other frameworks' implementation: Jacobian matrix is based on stacks of vector-Jacobian-product (VJP) outputs

$$\text{Jacobian}(f, \boldsymbol{x}_i) = [\text{VJP}(\boldsymbol{e}_1, f, \boldsymbol{x}_i); \cdots; \text{VJP}(\boldsymbol{e}_n, f, \boldsymbol{x}_i)]$$

- A new Jacobian implementation based on reverse-mode AD

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

Implementation example

The API is similar to TensorFlow V1:

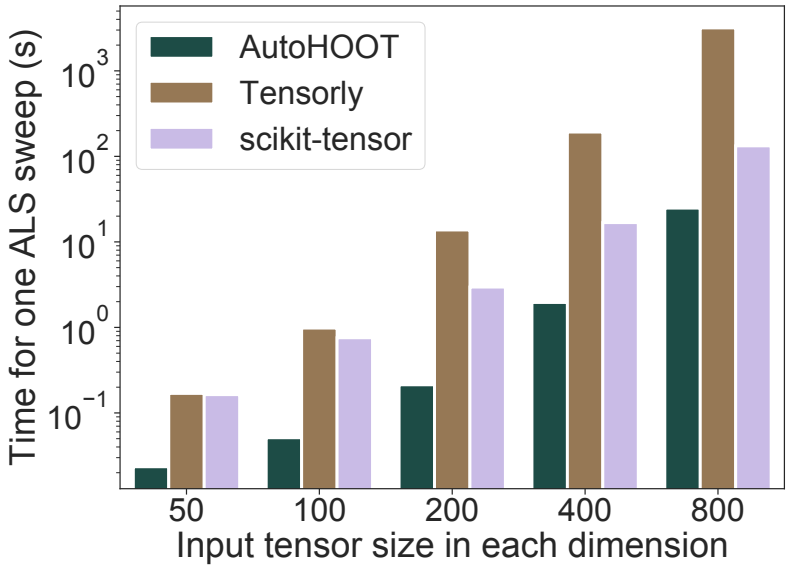
1. Define the computational graph
2. Optimize the graph
3. Execute the graph

```
# Define the computational graph
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 new_site

# Optimize the graph
new_A = optimize(update_site(A))
new_B = optimize(update_site(B))
new_C = optimize(update_site(C))

# Execute the graph
executor = ad.Executor([loss, new_A, new_B, new_C])
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])
```

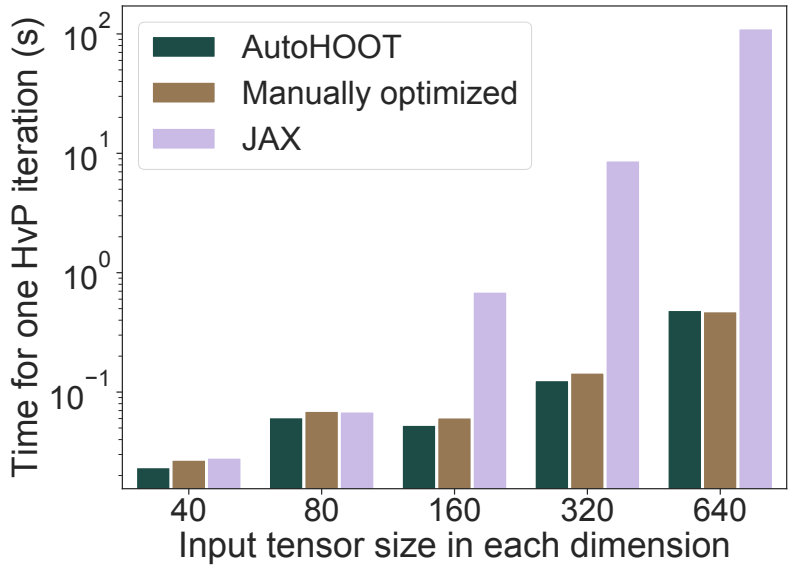
Experiments

- We test the performance of AutoHOOT on both tensor decompositions (CP and Tucker) and tensor network (DMRG) applications (all are most widely used methods)
- Performances are evaluated on NumPy/TensorFlow/Cyclops libraries
- We compare the performance to JAX and also other popular tensor computation frameworks (these frameworks optimize/implement derivative kernels manually)

Results for 3D CP Decomposition



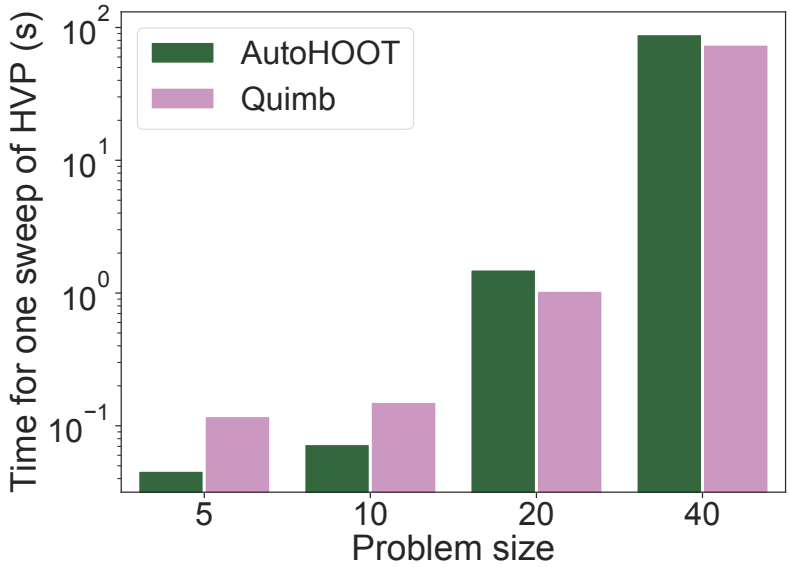
(a) ALS



(b) Gauss Newton

- At least 7X speedup compared to existing softwares when the tensor is reasonably large
- JAX results not shown on ALS, because it is too slow (needs to invert a big tensor)

Results for DMRG benchmark



(c) HVP kernel benchmark

- Performance comparable with Quimb
- JAX results not shown, because it will use up the memory quickly

Summary and Conclusion

	AutoHOOT	TensorFlow, PyTorch, JAX
Graph optimizer	✓ Includes both traditional graph optimization techniques and tensor-algebra specific transformations	Includes traditional graph optimization techniques
AD module	✓ Output graph keeps the tensor structure granularity and easier to optimize	Output graph is relatively hard to optimize
Performance on tensor decomposition/network applications	✓ Comparable, sometimes better performance compared to existing tensor frameworks with manually implemented derivatives	1. Much slower than existing tensor frameworks 2. Sometimes have OOM issues

- Framework publicly available at https://github.com/LinjianMa/AutoHOOT

Thank you!

- Framework publicly available at https://github.com/LinjianMa/AutoHOOT
- If you have questions, please contact Linjian Ma at 1ma16@illinois.edu.
- This presentation and recording belong to the authors. No distribution is allowed without the authors' permission.

Back-up Slides

Optimization algorithm

- The general optimization algorithm can be split into 4 steps
  - Distribute Einsum nodes
  - Fuse all Einsum subtrees into Einsum nodes and prune Identity nodes
  - Optimize and prune redundant inverse nodes
  - Optimize the expression using the symbolic mathematics library (Example: $0.5A + 0.5A + 0.5C - 0.5C \rightarrow A$)

- We will show how to perform the 3D CP decomposition with the framework

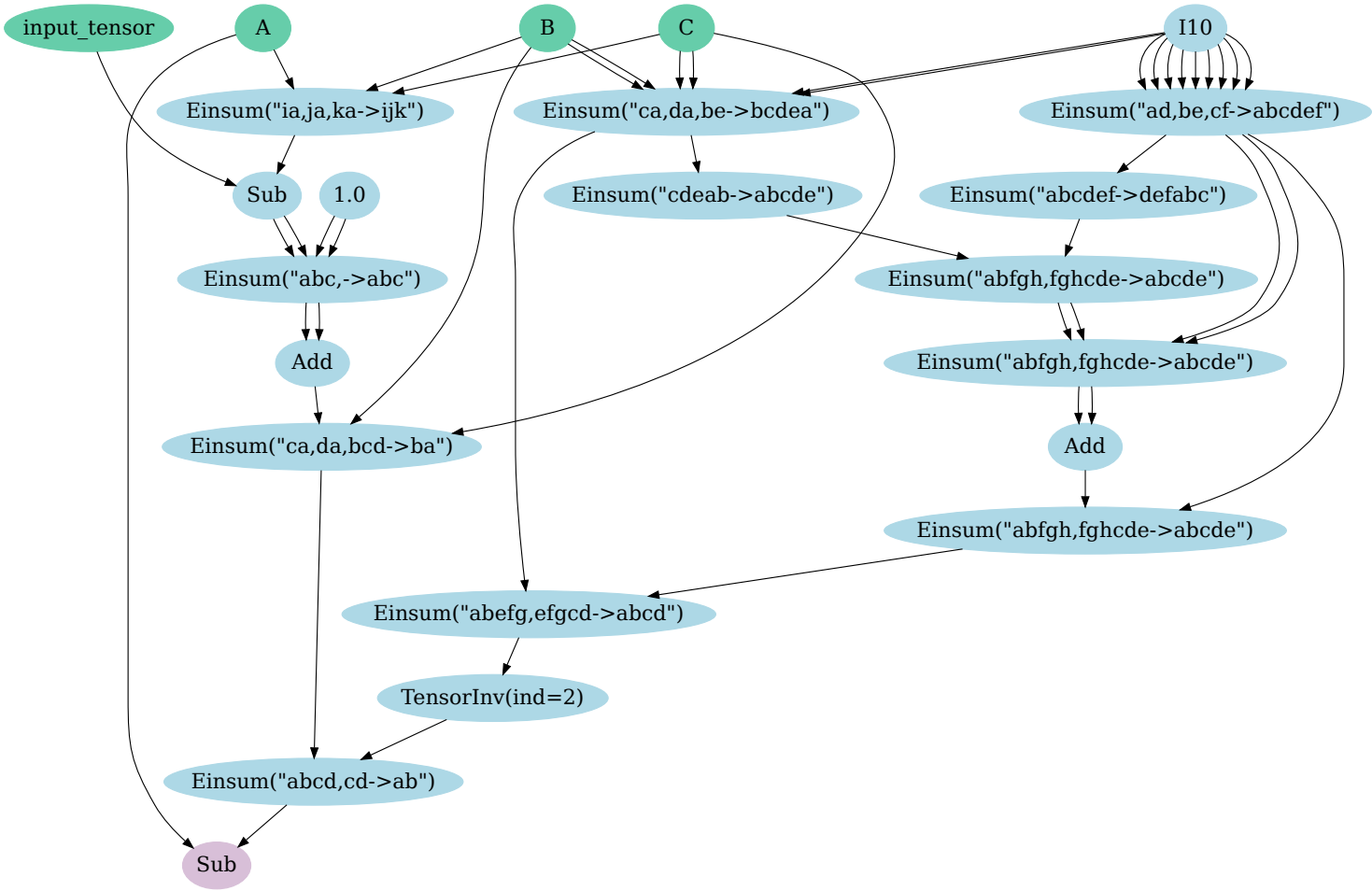
$$\min_{\boldsymbol{A},\boldsymbol{B},\boldsymbol{C}} \phi(\boldsymbol{A},\boldsymbol{B},\boldsymbol{C}) := \frac{1}{2}\|\boldsymbol{\mathcal{X}} - f_{\text{CP}}(\boldsymbol{A},\boldsymbol{B},\boldsymbol{C})\|^2,$$

- Alternating least squares (ALS) updates this in an alternating manner:

$$\boldsymbol{\mathcal{H}}_{\boldsymbol{A}}\boldsymbol{A}_{\text{new}} = \nabla\phi_{\boldsymbol{A}}, \quad \boldsymbol{\mathcal{H}}_{\boldsymbol{B}}\boldsymbol{B}_{\text{new}} = \nabla\phi_{\boldsymbol{B}}, \quad \boldsymbol{\mathcal{H}}_{\boldsymbol{C}}\boldsymbol{C}_{\text{new}} = \nabla\phi_{\boldsymbol{C}}$$

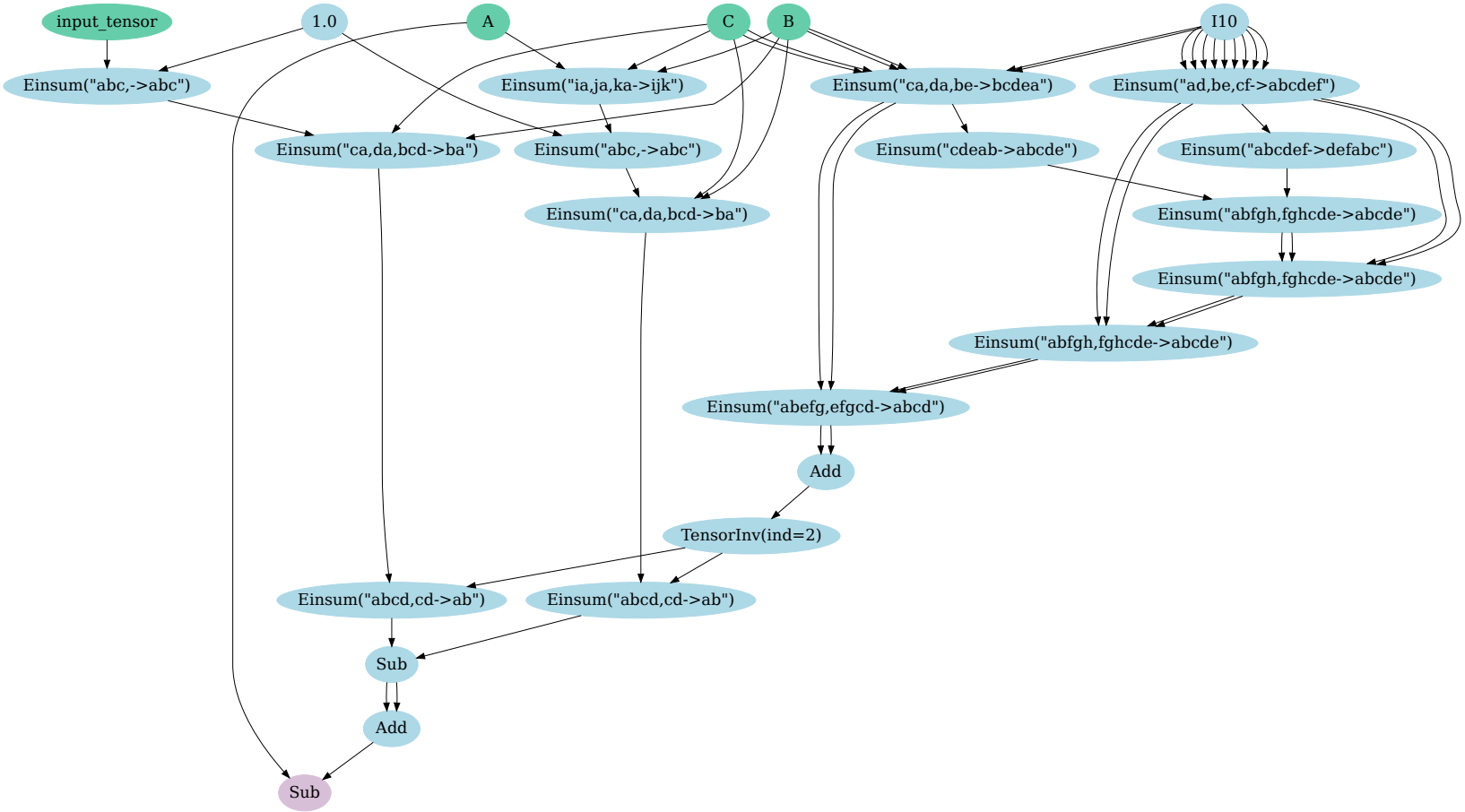
Optimization algorithm: example of CP-ALS

The graph of $\boldsymbol{A}_{\text{new}}$ generated from AD module:



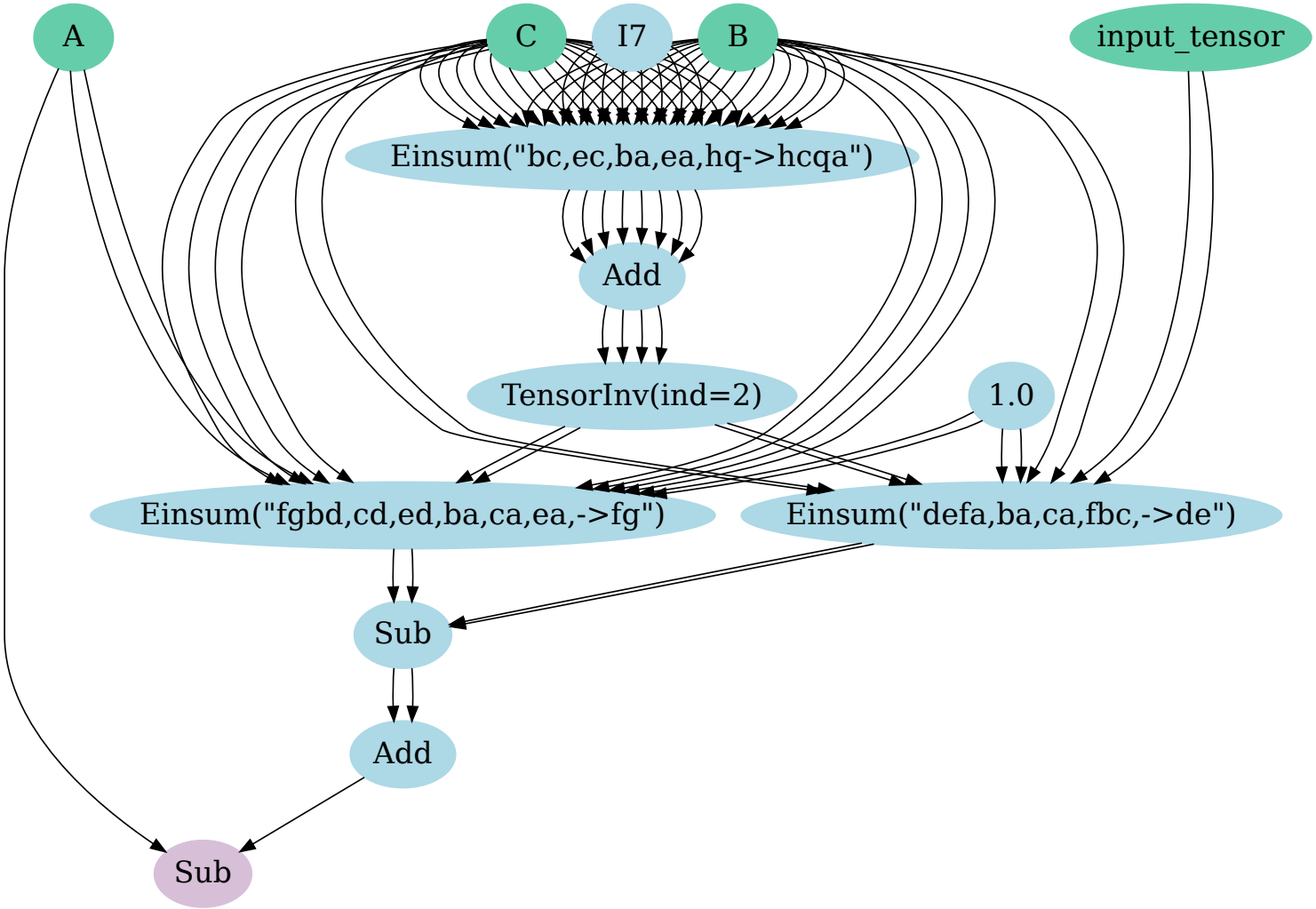
Optimization algorithm: example of CP-ALS

1. Distribute Einsum nodes



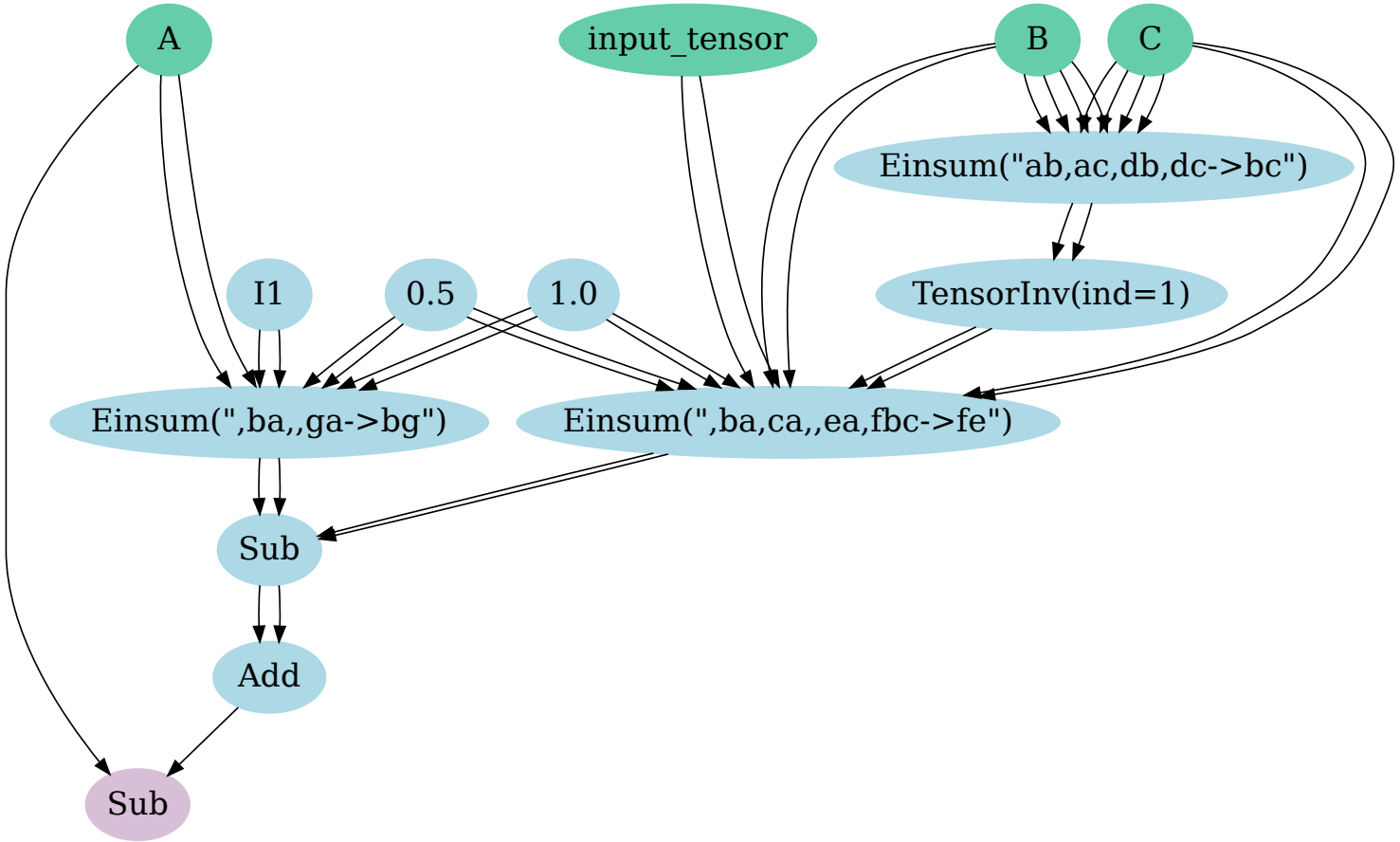
Optimization algorithm: example of CP-ALS

2. Fuse all Einsum subtrees into Einsum nodes and prune Identity nodes



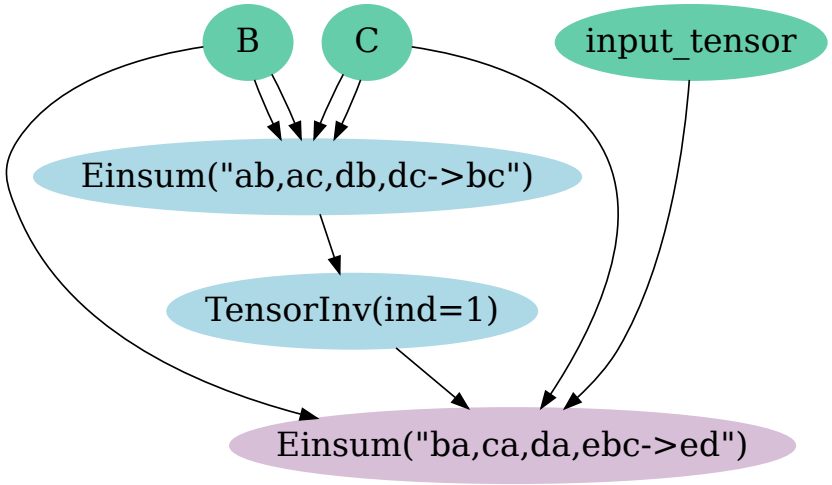
Optimization algorithm: example of CP-ALS

3. Optimize and prune redundant inverse nodes



Optimization algorithm: example of CP-ALS

4. Optimize the expression using the symbolic mathematics library



Automatic differentiation background

- VJP can be implemented efficiently based on reverse-mode AD

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

  - MatVec rather than MatMul needed in each step
  - Popular for gradient calculation of loss functions: when output size $s_{N+1} = 1$, gradient can be calculated calling VJP once: $\text{gradient}(f, \boldsymbol{x}_1) = \text{VJP}([1], f, \boldsymbol{x}_1)^T$
  - The Jacobian matrix can be calculated through calling VJP s_{N+1} times

$$\text{Jacobian}(f, \boldsymbol{x}_i) = [\text{VJP}(\boldsymbol{e}_1, f, \boldsymbol{x}_i); \cdots; \text{VJP}(\boldsymbol{e}_n, f, \boldsymbol{x}_i)]$$

  - Demo for the Jacobian calculation with VJP

Automatic differentiation background

Let's take a look at how TensorFlow implement the Jacobian...

```
48  flat_inputs = nest.flatten(inputs)
49  output_tensor_shape = output.shape
50  output_shape = array_ops.shape(output)
51  output = array_ops.reshape(output, [-1])
52
53  def loop_fn(i):
54    y = array_ops.gather(output, i)
55    return gradient_ops.gradients(y, flat_inputs)
56
57  try:
58    output_size = int(output.shape[0])
59  except TypeError:
60    output_size = array_ops.shape(output)[0]
61
62  if use_pfor:
63    pfor_outputs = control_flow_ops.pfor(
64        loop_fn, output_size, parallel_iterations=parallel_iterations)
65  else:
66    pfor_outputs = control_flow_ops.for_loop(
67        loop_fn,
68        [output.dtype] * len(flat_inputs),
69        output_size,
70        parallel_iterations=parallel_iterations)
71
72  for i, out in enumerate(pfor_outputs):
73    if isinstance(out, ops.Tensor):
74      new_shape = array_ops.concat(
75          [output_shape, array_ops.shape(out)[1:]], axis=0)
76      out = array_ops.reshape(out, new_shape)
77      out.set_shape(output_tensor_shape.concatenate(flat_inputs[i].shape))
78      pfor_outputs[i] = out
79
80  return nest.pack_sequence_as(inputs, pfor_outputs)
```

Flatten both inputs and the output

Wrap the gradients function

Calculate all the gradients

Stack gradients together and reshape

Automatic differentiation background

- JVP can be implemented efficiently based on forward-mode AD

$$\text{JVP}(\boldsymbol{v}, f, \boldsymbol{x}_i) = \boldsymbol{J}^{[f]}_{[\boldsymbol{x}_i]}\boldsymbol{v} = \boldsymbol{J}^{[f]}_{[\boldsymbol{x}_{i+1}]}(\boldsymbol{J}^{[f_i]}_{[\boldsymbol{x}_i]}\boldsymbol{v}) = \text{JVP}(\boldsymbol{J}^{[f_i]}_{[\boldsymbol{x}_i]}\boldsymbol{v}, f, \boldsymbol{x}_{i+1})$$

  - MatVec rather than MatMul needed in each step
  - When input size $s_1 = 1$, gradient can be calculated calling JVP once: $\text{gradient}(f, \boldsymbol{x}_1) = \text{JVP}([1], f, \boldsymbol{x}_1)$
  - The Jacobian matrix can be calculated through calling VJP s_1 times

- Useful trick: JVP can be implemented based on calling the VJP function twice.
  - Construct a function g:

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

  - Perform another VJP operation on the function g with related to its input $\boldsymbol{u}$:

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

Automatic differentiation background

- HVP is implemented based on calling VJP/grad twice
  - Popular for implicitly solving $\boldsymbol{H}\boldsymbol{x} = \boldsymbol{b}$ via conjugate gradient
  - The Hessian matrix can be calculated through calling HVP s_1 times

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

Motivation for a new AD library

	Tensor computation	Neural network
Numerical optimization	Second order	First order
Operators	$\boldsymbol{H}, \boldsymbol{H}^{-1}$, JVP, VJP	Gradient, VJP, HVP
Tensor size	Big (>1GB)	Small (<10MB)
Computational depth	Shallow	Deep
Functions	Multilinear	Linear + nonlinear
Backends	CPU, GPU, distributed parallel system	GPU, embedded system