Tensor Contraction with Extended BLAS Kernels on CPU and GPU

Yang Shi
University of California, Irvine, EECS

Joint work with U.N. Niranjan, Animashree Anandkumar and Cris Cecka

SIAM-ALA18
Tensor Contraction-Motivation

Scalar

Vector

Matrix

Tensor

Scalar
Vector
Matrix
Tensor

2 3 5 9

2 4 7 8 1

2 3 0 4 9 2 5

1 7 2 2 9 3

5 5 2 7 3

2 5 1 6

5 8 9 0
Tensor Contraction-Motivation

Why we need tensor?

Modern data is inherently multi-dimensional

Neural Networks

Input → Hidden 1 → Hidden 2 → Output

Method of Moment

\[ E(x_1 \otimes x_2) = \text{[Diagram]} + \ldots + \text{[Diagram]} \]

\[ E(x_1 \otimes x_2 \otimes x_3) = \text{[Diagram]} + \ldots + \text{[Diagram]} \]
Tensor Contraction-Motivation

What is tensor contraction?

\[ C_C = A_A B_B \]

Why do we need tensor contraction?

- Physics
- Chemistry
Tensor Contraction-Motivation

Why do we need tensor contraction?

• Deep Learning

• Learning latent variable model with tensor decomposition
  Example: Topic modeling
  
  h: Proportion of topics in a document
  
  \[ h = i \text{ with prob. } w_i \]

  A: Topic-word matrix
  
  \[ A(i, j) = \mathbb{P}(x_m = i | y_m = j) \]

  Third order moment:
  
  \[ M_3 = \mathbb{E}(x \otimes x \otimes x) = \sum_i w_i a_i \otimes a_i \otimes a_i \]
What do we have?

Tensor computation libraries:

- Arbitrary/restricted tensor operations of any order and dimension
- Such as: Matlab Tensor toolbox, BTAS, FTensor, Cyclops

Efficient computing frame:

- Static analysis solutions: loop reorganization, fusion
- Parallel and distributed computing system: BatchedGEMM functions in MKL 11.3, CuBLAS v4.1: compute many matrix-matrix multiplies at once.
Tensor Contraction-Motivation

What are the limitations?

• Explicit permutation takes long time in current tensor libraries:

Consider $C_{mnp} = A_{km}B_{pkn}$

$A_{km} \rightarrow A_{mk}$.  
$B_{pkn} \rightarrow B_{kpn}$.  
$C_{mnp} \rightarrow C_{mpn}$.  
$C_{mpn} = A_{mk}B_{kpn}$.  
$C_{mpn} \rightarrow C_{mnp}$.

Figure: The fraction of time spent in copies/transpositions when computing $C_{mnp} = A_{km}B_{pkn}$. Lines are shown with 1, 2, 3, and 6 total transpositions performed on either the input or output. (Left) CPU. (Right) GPU.
Overview

• Propose tensor operation kernel: StridedBatchedGEMM
  • Library-based approaches that avoid memory movement
  • Constant-strided BatchedGEMM that has more optimization opportunities

• Provide evaluation strategies for tensor contractions

• Apply to tensor decomposition

• Introduce TensorLy: Tensor learning in python
BLAS Operations

BLAS (Basic Linear Algebra Subprograms): Low-level routines for performing common linear algebra operations.

<table>
<thead>
<tr>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y \leftarrow \alpha x + y$</td>
<td>$y \leftarrow \alpha \text{op}(A)x + \beta y$</td>
<td>$C \leftarrow \alpha \text{op}(A)\text{op}(B) + \beta C$</td>
</tr>
</tbody>
</table>

Example:

GEMM (ORDER, TRANSA, TRANSB, M, N, K, $\alpha$, A, LDA, B, LDB, $\beta$, C, LDC)
Extended BLAS Operator

Focusing: one-index contraction

\[ C = \alpha \text{op}(A) \text{op}(B) + \beta C \]

\[ C_{mnp} = A_{**} \times B_{***} \]

If fixing indices of C, there are total \(3 \times 2 \times 3 \times 2 \times 1 = 36\) cases.
Extended BLAS Operator

<table>
<thead>
<tr>
<th>Case</th>
<th>Contraction</th>
<th>Kernel1</th>
<th>Kernel2</th>
<th>Kernel3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>$A_{mk}B_{kn}$</td>
<td>$C_{m(np)} = A_{mk}B_{k(np)}$</td>
<td>$C_{mn[p]} = A_{mk}B_{kn[p]}$</td>
<td>$C_{m[n]p} = A_{mk}B_{k[n]p}$</td>
</tr>
</tbody>
</table>

Table: Example: possible mappings to Level 3 BLAS routines

StridedBatchedGEMM(ORDER, TRANSA, TRANSB, M, N, K, $\alpha$, A, LDA, LOA, B, LDB, LOB, $\beta$, C, LDC, LOC, P)
Example

Table: List of 36 possible single mode contraction operations between a second-order tensor and a third-order tensor and possible mappings to Level-3 BLAS routines

<table>
<thead>
<tr>
<th>Case</th>
<th>Contraction</th>
<th>Kernel1</th>
<th>Kernel2</th>
<th>Kernel3</th>
<th>Case</th>
<th>Contraction</th>
<th>Kernel1</th>
<th>Kernel2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>$A_{km}B_{kp}\mathbf{n}$</td>
<td>$C_{m(pn)} = A_{km}B_{kn(\mathbf{n})}$</td>
<td>$C_{m[p]} = A_{mk}B_{k[p]n}$</td>
<td>$C_{m[n]}p = A_{mk}B_{k[n]p}$</td>
<td>4.1</td>
<td>$A_{kn}B_{km}\mathbf{p}$</td>
<td>$C_{m[p]} = B_{km[p]}A_{kn}$</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>$A_{km}B_{kpn}$</td>
<td>$C_{mn[p]} = A_{km}B_{k[p]n}$</td>
<td>$C_{m[p]} = A_{mk}B_{k[p]n}$</td>
<td></td>
<td>4.2</td>
<td>$A_{kn}B_{km}\mathbf{m}$</td>
<td>$C_{m[p]} = B_{km[p]}A_{kn}$</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>$A_{km}B_{knp}$</td>
<td>$C_{mn[p]} = A_{km}B_{k[p]n}$</td>
<td>$C_{m[p]} = A_{mk}B_{k[p]n}$</td>
<td></td>
<td>4.3</td>
<td>$A_{kn}B_{km}\mathbf{p}$</td>
<td>$C_{m[p]} = B_{mk[p]}A_{kn}$</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>$A_{km}B_{pkn}$</td>
<td>$C_{mn[p]} = A_{km}B_{k[p]n}$</td>
<td>$C_{m[p]} = A_{mk}B_{k[p]n}$</td>
<td></td>
<td>4.4</td>
<td>$A_{kn}B_{pkm}$</td>
<td>$\text{TRANS}(A_{kn}^T B_{pk[m]})$</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>$A_{km}B_{npk}$</td>
<td>$C_{mn[p]} = A_{km}B_{k[p]n}$</td>
<td>$C_{m[p]} = A_{mk}B_{k[p]n}$</td>
<td></td>
<td>4.5</td>
<td>$A_{kn}B_{mpk}$</td>
<td>$C_{m[p]} = B_{mp[k]}A_{kn}$</td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td>$A_{km}B_{pkn}$</td>
<td>$C_{mn[p]} = A_{km}B_{k[p]n}$</td>
<td>$C_{m[p]} = A_{mk}B_{k[p]n}$</td>
<td></td>
<td>4.6</td>
<td>$A_{kn}B_{pnmk}$</td>
<td>$\text{TRANS}(A_{kn}^T B_{pm[n][k]})$</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>$A_{km}B_{knp}$</td>
<td>$C_{mn[p]} = A_{km}^T B_{k(pn)}$</td>
<td>$C_{m[p]} = A_{km}^T B_{kn[p]}$</td>
<td>$C_{m[n]}p = A_{kn}^T B_{k[n]p}$</td>
<td>5.1</td>
<td>$A_{pk}B_{kmn}$</td>
<td>$C_{m[p]} = B_{km[n]}A_{pk}$</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>$A_{km}B_{kpn}$</td>
<td>$C_{mn[p]} = A_{km}^T B_{k[p]n}$</td>
<td>$C_{m[p]} = A_{km}^T B_{kp[n]}$</td>
<td></td>
<td>5.2</td>
<td>$A_{pk}B_{kmn}$</td>
<td>$C_{m[p]} = B_{mk[n]}A_{pk}$</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>$A_{km}B_{knp}$</td>
<td>$C_{mn[p]} = A_{km}^T B_{kn[p]}$</td>
<td>$C_{m[p]} = A_{km}^T B_{kn[p]}$</td>
<td></td>
<td>5.3</td>
<td>$A_{pk}B_{kmn}$</td>
<td>$C_{m[p]} = B_{mk[n]}A_{pk}$</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>$A_{km}B_{pkn}$</td>
<td>$C_{mn[p]} = A_{km}^T B_{pkn}$</td>
<td>$C_{m[p]} = A_{km}^T B_{pkn}$</td>
<td></td>
<td>5.4</td>
<td>$A_{pk}B_{kmn}$</td>
<td>$\text{TRANS}(B_{nk[m]} A_{pk})$</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>$A_{km}B_{npk}$</td>
<td>$C_{mn[p]} = A_{km}^T B_{npk}$</td>
<td>$C_{m[p]} = A_{km}^T B_{npk}$</td>
<td></td>
<td>5.5</td>
<td>$A_{pk}B_{mnk}$</td>
<td>$C_{m[n]}p = B_{m[n]} A_{pk}$</td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>$A_{km}B_{pkn}$</td>
<td>$C_{mn[p]} = A_{km}^T B_{pkn}$</td>
<td>$C_{m[p]} = A_{km}^T B_{pkn}$</td>
<td></td>
<td>5.6</td>
<td>$A_{pk}B_{mnk}$</td>
<td>$\text{TRANS}(B_{nk[m]} A_{pk})$</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>$A_{kn}B_{km\mathbf{p}}$</td>
<td>$C_{mn[p]} = B_{km[p]} A_{kn}$</td>
<td>$C_{m[n]}p = B_{km[p]} A_{kn}$</td>
<td></td>
<td>6.1</td>
<td>$A_{kp}B_{kmn}$</td>
<td>$C_{m[n]}p = B_{km[n]} A_{kp}$</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>$A_{kn}B_{kpm}$</td>
<td>$C_{mn[p]} = B_{k[p]m} A_{kn}$</td>
<td>$C_{m[n]}p = B_{km[p]} A_{kn}$</td>
<td></td>
<td>6.2</td>
<td>$A_{kp}B_{kmn}$</td>
<td>$C_{m[n]}p = B_{km[n]} A_{kp}$</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>$A_{kn}B_{mk\mathbf{p}}$</td>
<td>$C_{mn[p]} = B_{mk[p]} A_{kn}$</td>
<td>$C_{m[n]}p = B_{km[p]} A_{kn}$</td>
<td></td>
<td>6.3</td>
<td>$A_{kp}B_{kmn}$</td>
<td>$C_{m[n]}p = B_{km[n]} A_{kp}$</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>$A_{kn}B_{pk\mathbf{m}}$</td>
<td>$\text{TRANS}(A_{kn} B_{pk[m]})$</td>
<td>$C_{m[n]}p = B_{pk[m]} A_{nk}$</td>
<td></td>
<td>6.4</td>
<td>$A_{kp}B_{kmn}$</td>
<td>$\text{TRANS}(B_{nk[m]} A_{kp})$</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>$A_{kn}B_{mp\mathbf{k}}$</td>
<td>$C_{mn[p]} = B_{mp[k]} A_{nk}$</td>
<td>$C_{m[n]}p = B_{pk[m]} A_{nk}$</td>
<td></td>
<td>6.5</td>
<td>$A_{kp}B_{kmn}$</td>
<td>$C_{m[n]}p = B_{mk[n]} A_{kp}$</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>$A_{kn}B_{pkm}$</td>
<td>$\text{TRANS}(A_{kn} B_{pm[k]})$</td>
<td>$C_{m[n]}p = B_{pk[m]} A_{nk}$</td>
<td></td>
<td>6.6</td>
<td>$A_{kp}B_{kmn}$</td>
<td>$\text{TRANS}(B_{nk[m]} A_{kp})$</td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Case</th>
<th>Contraction</th>
<th>Kernel1</th>
<th>Kernel2</th>
<th>Kernel3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>$A_{mk}B_{knp}$</td>
<td>$C_{m(np)} = A_{mk}B_{k(np)}$</td>
<td>$C_{mn[p]} = A_{mk}B_{kn[p]}$</td>
<td>$C_{m[n]p} = A_{mk}B_{k[n]p}$</td>
</tr>
<tr>
<td>6.1</td>
<td>$A_{kp}B_{kmn}$</td>
<td>$C_{(mn)p} = B_{k(mn)}^{\top}A_{kp}$</td>
<td>$C_{m[n]p} = B_{km[n]}^{\top}A_{kp}$</td>
<td></td>
</tr>
</tbody>
</table>

Case 1.1, Kernel2: $C_{mn[p]} = A_{mk}B_{kn[p]}$

```c
    cublasDgemmStridedBatched(handle,
        CUBLAS_OP_N, CUBLAS_OP_N,
        M, N, K,
        &alpha,
        A, ldA1, 0,
        B, ldB1, ldB2,
        &beta,
        C, ldC1, ldC2,
        P)
```

Case 6.1, Kernel2: $C_{m[n]p} = B_{km[n]}^{\top}A_{kp}$

```c
    cublasDgemmStridedBatched(handle,
        CUBLAS_OP_T, CUBLAS_OP_N,
        M, P, K,
        &alpha,
        B, ldB1, ldB2,
        A, ldA1, 0,
        &beta,
        C, ldC2, ldC1,
        N)
```
Analysis

Figure: Performance of three strategies for computing N matrix-matrix multiplications of size NxN.

Overhead: (1) GPU memory allocation, (2) Pointer offset computations, (3) GPU memory transfers/writes, and (4) GPU memory deallocation
Analysis

Flatten v.s. SBGEMM

Prefer flatten than SBGEMM
Analysis

Batching in last mode v.s. middle mode

On CPU, it’s better to batch in last mode when tensor size is small/moderate
Analysis

Mixed mode batching

On CPU: mode of the output tensor is more important than the batching mode of the input tensor.
Analysis

• Flatten V.S. SBGEMM
  • A single large GEMM is more efficient
  • Flatten modes whenever possible

• Batching in last mode V.S. Batching in earlier mode
  • On CPU: prefer batching in the last mode when tensor size is small
  • On GPU: no discernible preference

• Mixed mode batching on input/output tensors
  • On CPU: mode of the output tensor is more important than the batching mode of the input tensor.
Application: Tucker Decomposition

\[ T_{mnp} = G_{ijk} A_{mi} B_{nj} C_{pk} \]

Main Steps:

- \[ Y_{mjk} = T_{mnp} B_{nj}^t C_{pk}^t \]
- \[ Y_{ink} = T_{mnp} A_{mi}^{t+1} C_{pk}^t \]
- \[ Y_{ijp} = T_{mnp} B_{nj}^{t+1} A_{mi}^{t+1} \]
Application: Tucker Decomposition

Figure: Performance on Tucker decomposition.
Conclusion

• StridedBatchedGEMM for generalized tensor contractions.

• Avoid explicit transpositions or permutations.

• 10x (GPU) and 2x (CPU) speedup on small and moderate sized tensors.

• Available in CuBLAS 8.0.
Introduction of TensorLy

by Jean Kossaifi, Imperial College London
Yannis Panagakis, Imperial College London
Anima Anandkumar, Caltech

- **Open source**
  
  Homepage: http://tensorly.org/dev/

  Github: https://github.com/tensorly/tensorly

  Suitable for academic / industrial applications

- **Reliability and easy to use**
  
  Depends only on NumPy, SciPy [Optionally Matplotlib, MXNet and PyTorch]

  Exhaustive documentation, Unit-testing for all functions

  Fast
User-friendly API

Tensor decomposition

Tensor regression

Tensors + Deep

Basic tensor operations

Unified backend

NumPy
SciPy
mxnet
PYTORCH
TensorFlow
TensorLy Operators

- Kronecker
- Khatri-rao
- Hadamard products
- Tensor unfolding/folding/vectorization
- N-mode product

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>khatra_rao</code></td>
<td>Khatri-Rao product of a list of matrices</td>
</tr>
<tr>
<td><code>kronecker</code></td>
<td>Kronecker product of a list of matrices</td>
</tr>
<tr>
<td><code>mode_dot</code></td>
<td>n-mode product of a tensor and a matrix or vector at the specified mode</td>
</tr>
<tr>
<td><code>multi_mode_dot</code></td>
<td>n-mode product of a tensor and several matrices or vectors over several modes</td>
</tr>
<tr>
<td><code>proximal.soft_thresholding</code></td>
<td>Soft-thresholding operator</td>
</tr>
<tr>
<td><code>proximal.svd_thresholding</code></td>
<td>Singular value thresholding operator</td>
</tr>
<tr>
<td><code>proximal.procrustes</code></td>
<td>Procrustes operator</td>
</tr>
<tr>
<td><code>inner</code></td>
<td>Generalised inner products between tensors</td>
</tr>
</tbody>
</table>

- CANONICAL-POLYADIC (CP)
- Non-negative CP Tucker (HO-SVD)
- Non-negative Tucker
- Robust Tensor PCA

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parafac</code></td>
<td>CANDECOMP/PARAFAC decomposition via alternating least squares (ALS)</td>
</tr>
<tr>
<td><code>non_negative_parafac</code></td>
<td>Non-negative CP decomposition</td>
</tr>
<tr>
<td><code>tucker</code></td>
<td>Tucker decomposition via Higher Order Orthogonal Iteration (HOI)</td>
</tr>
<tr>
<td><code>partial_tucker</code></td>
<td>Partial Tucker decomposition via Higher Order Orthogonal Iteration (HOI)</td>
</tr>
<tr>
<td><code>non_negative_tucker</code></td>
<td>Non-negative Tucker decomposition</td>
</tr>
<tr>
<td><code>robust_pca</code></td>
<td>Robust Tensor PCA via ALM with support for missing values</td>
</tr>
</tbody>
</table>
### TensorLy Backend

```python
import tensorly as tl

T = tl.tensor([[1, 2, 3], [4, 5, 6]])

# Kronecker product
import tensorly.tenalg.kronecker

T = tl.tenalg.kronecker([T, T])

# Clip tensor
import tensorly.clip

T = tl.clip(T, a_min=2, a_max=5)

# Change backend

# NumPy backend
import tensorly.set_backend

tl.set_backend('numpy')  # or 'mxnet' or 'pytorch'

T = tl.tensor([[1, 2, 3], [4, 5, 6]])

# MXNet backend

T = tl.tensor([[1, 2, 3], [4, 5, 6]])

tl.set_backend('mxnet')

# PyTorch backend

T = tl.tensor([[1, 2, 3], [4, 5, 6]])

tl.set_backend('pytorch')
```

- **NumPy ndarray**
- **MXNet NDArray**
- **PyTorch FloatTensor**
from tensorly.decomposition import parafac

factors = parafac(image, rank=50, init='random')
cp_reconstruction = tl.kruskal_to_tensor(factors)

from tensorly.decomposition import tucker

core, factors = tucker(image, ranks=(50, 50, 3), init='random')
tucker_reconstruction = tl.tucker_to_tensor(core, factors)
TensorLy Example

Back-propagate through tensor operations with PyTorch

```python
import tensorly as tl
from tensorly.random import tucker_tensor

tl.set_backend('pytorch')
core, factors = tucker_tensor((5, 5, 5), rank=(3, 3, 3))
core = Variable(core, requires_grad=True)
factors = [Variable(f, requires_grad=True) for f in factors]

optimiser = torch.optim.Adam([core]+factors, lr=lr)

for i in range(1, n_iter):
    optimiser.zero_grad()
    rec = tucker_to_tensor(core, factors)
    loss = (rec - tensor).pow(2).sum()
    for f in factors:
        loss = loss + 0.01*f.pow(2).sum()

    loss.backward()
    optimiser.step()
```

PyTorch FloatTensor
We can attach gradients
Penalty on the factors
Thank you!

Questions?