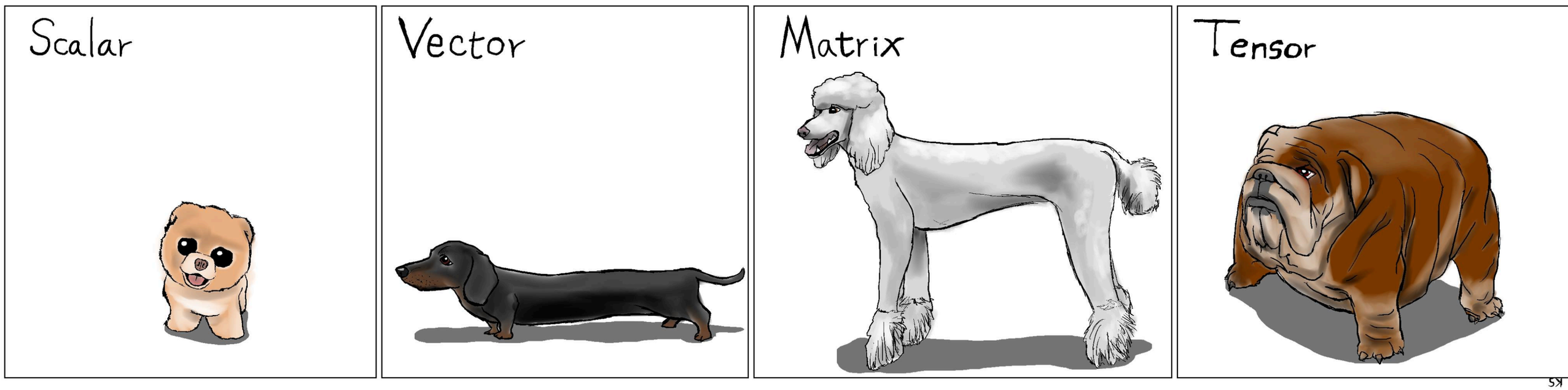
Tensor Contraction with Extended BLAS Kernels on CPU and GPU

Yang Shi
University of California, Irvine, EECS

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

SIAM-ALA18

Tensor Contraction-Motivation



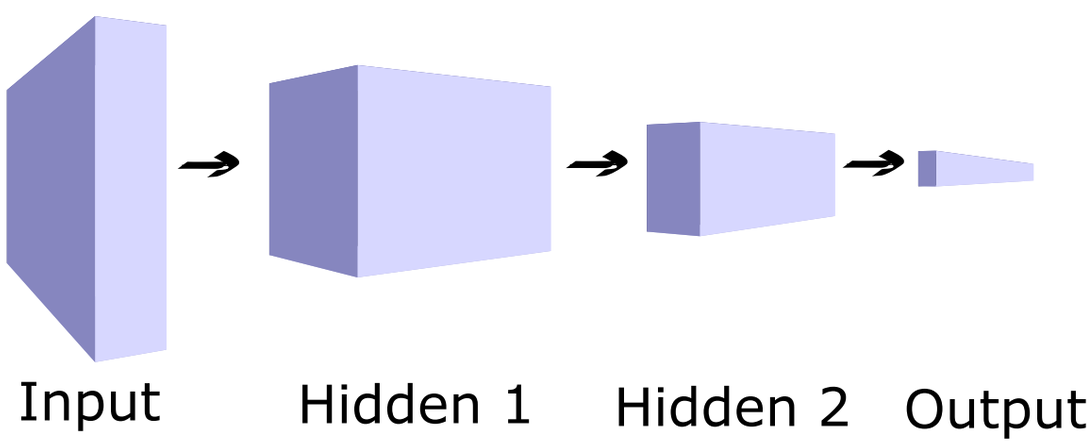
Scalar	Vector	Matrix	Tensor
2	$\begin{matrix} 2 \\ 3 \\ 5 \\ 9 \end{matrix}$	$\begin{matrix} 4 & 7 & 8 & 1 \\ 2 & 5 & 1 & 6 \\ 3 & 3 & 9 & 8 \end{matrix}$	$\begin{matrix} 1 & 4 & 2 & 5 \\ 2 & 3 & 4 & 2 \\ 3 & 0 & 9 & 0 \\ 1 & 0 & 8 & 9 \\ 7 & 2 & 2 & 3 \\ 5 & 5 & 2 & 7 \end{matrix}$

Tensor Contraction-Motivation

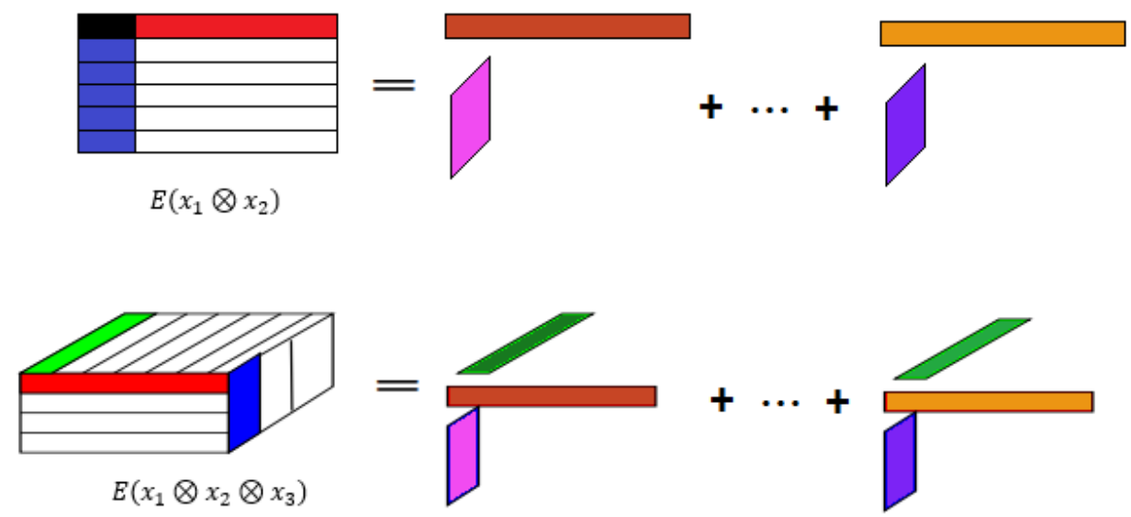
Why we need tensor?

Modern data is inherently multi-dimensional

Neural Networks



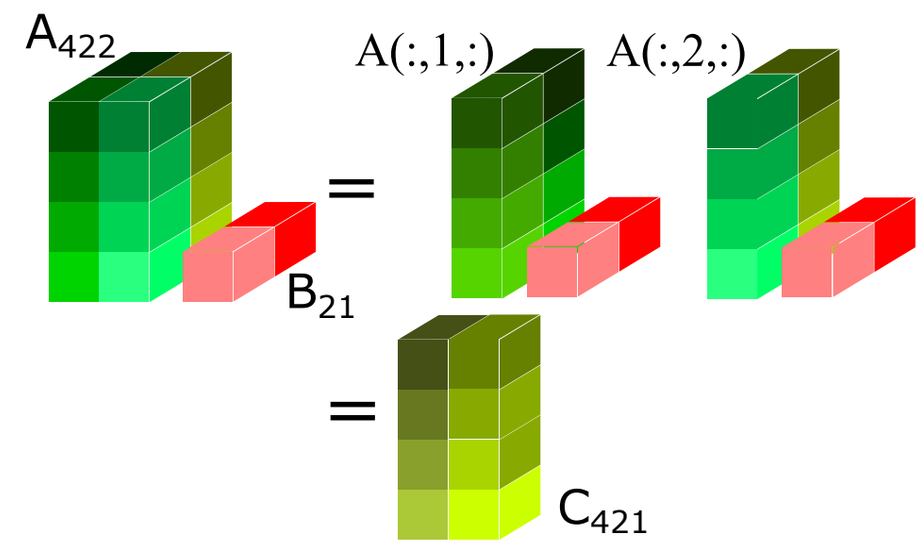
Method of Moment



Tensor Contraction-Motivation

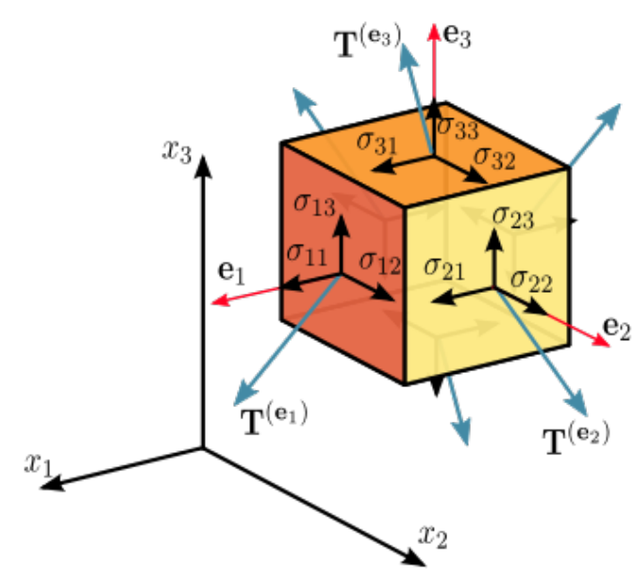
What is tensor contraction?

$$C_{\mathcal{C}} = A_{\mathcal{A}} B_{\mathcal{B}}$$



Why do we need tensor contraction?

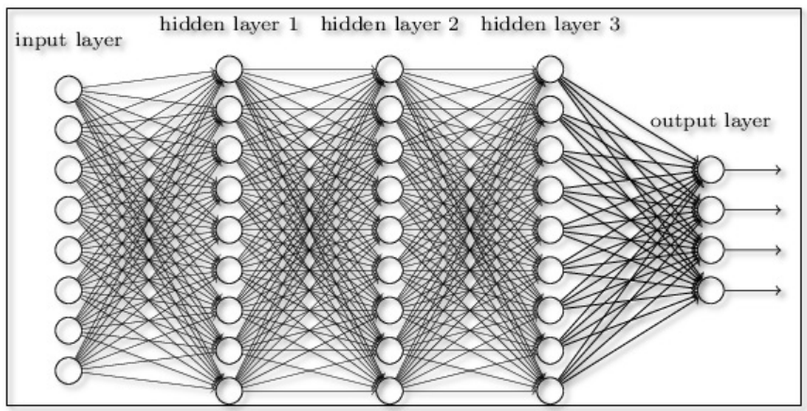
- Physics
- Chemistry



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

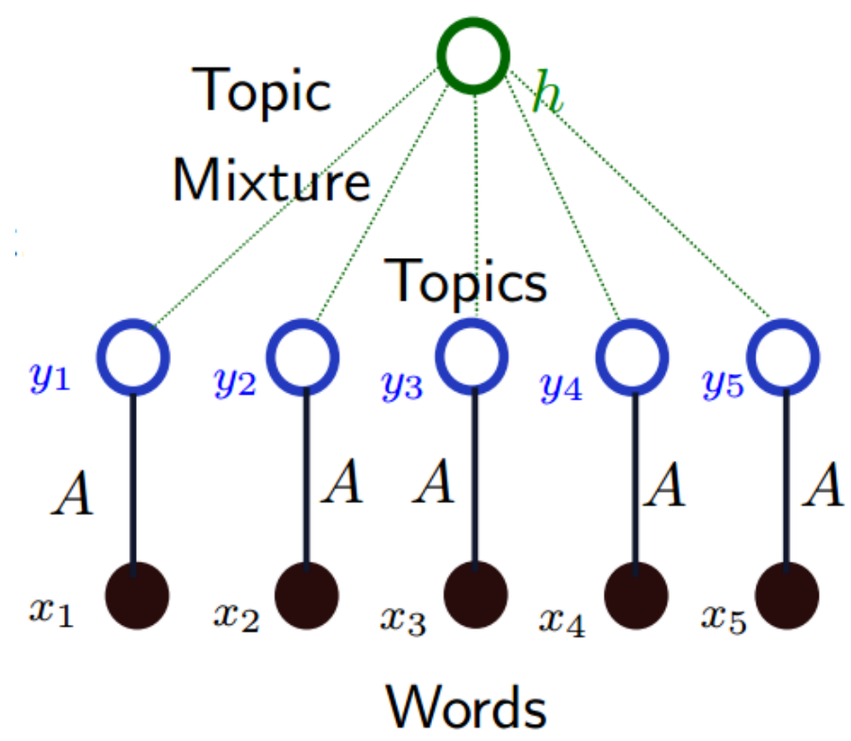
  $$h = i \text{ with prob. } w_i$$

  A: Topic-word matrix

  $$A(i,j) = \mathcal{P}(x_m = i | y_m = j)$$

  Third order moment:

  $$M_3 = \mathbb{E}(x \otimes x \otimes x) = \sum_i w_i a_i \otimes a_i \otimes a_i$$



Tensor Contraction-Motivation

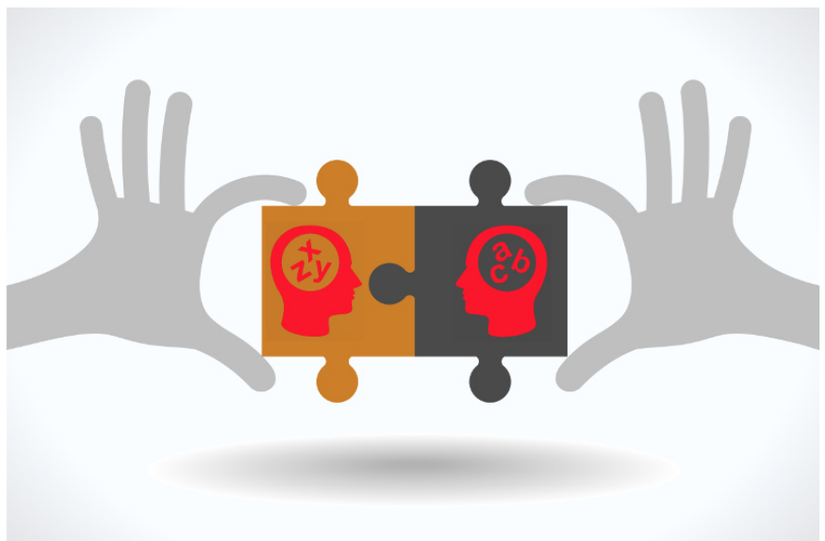
What do we have?

Tensor computation libraries:

- Arbitrary/restricted tensor operations of any order and dimension
- Such as: Matlab Tensortoolbox, 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 $C_{mnp} = A_{km} B_{pkn}$

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

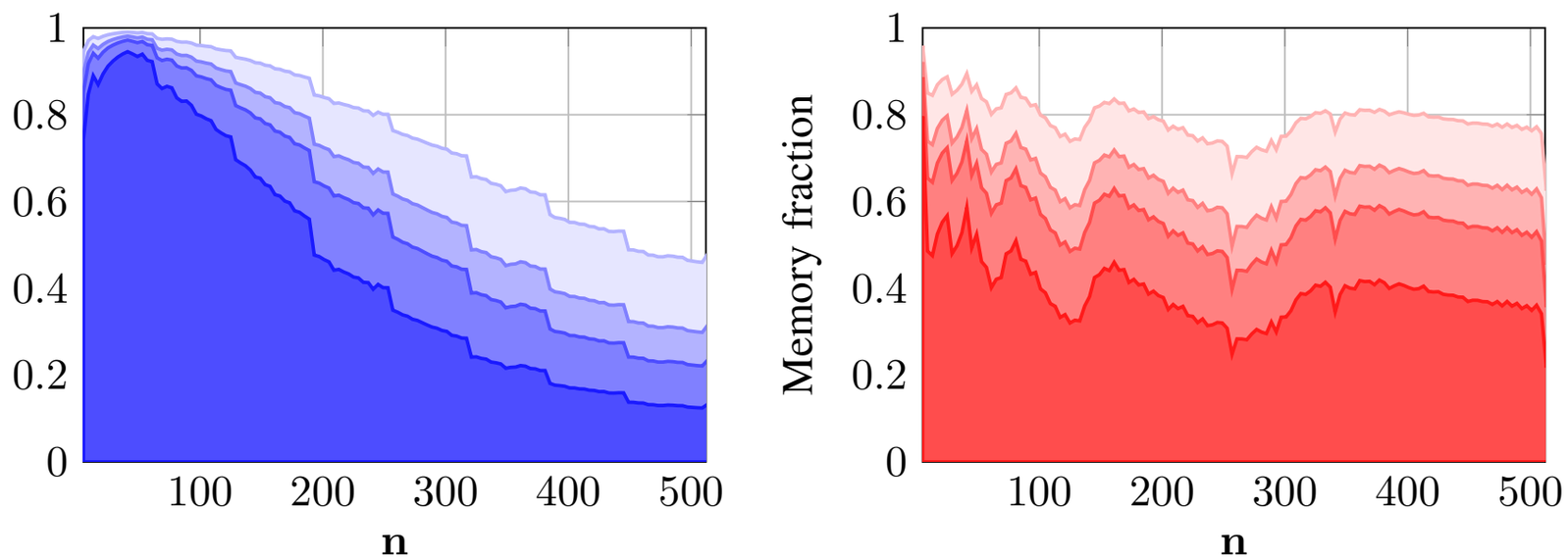


Figure: The fraction of time spent in copies/transpositions when computing Cmnp = AmkBpkn . Lines are shown with 1, 2, 3, and 6 total transpositions performed on either the input or output. (Left) CPU. (Right) GPU.

Overview

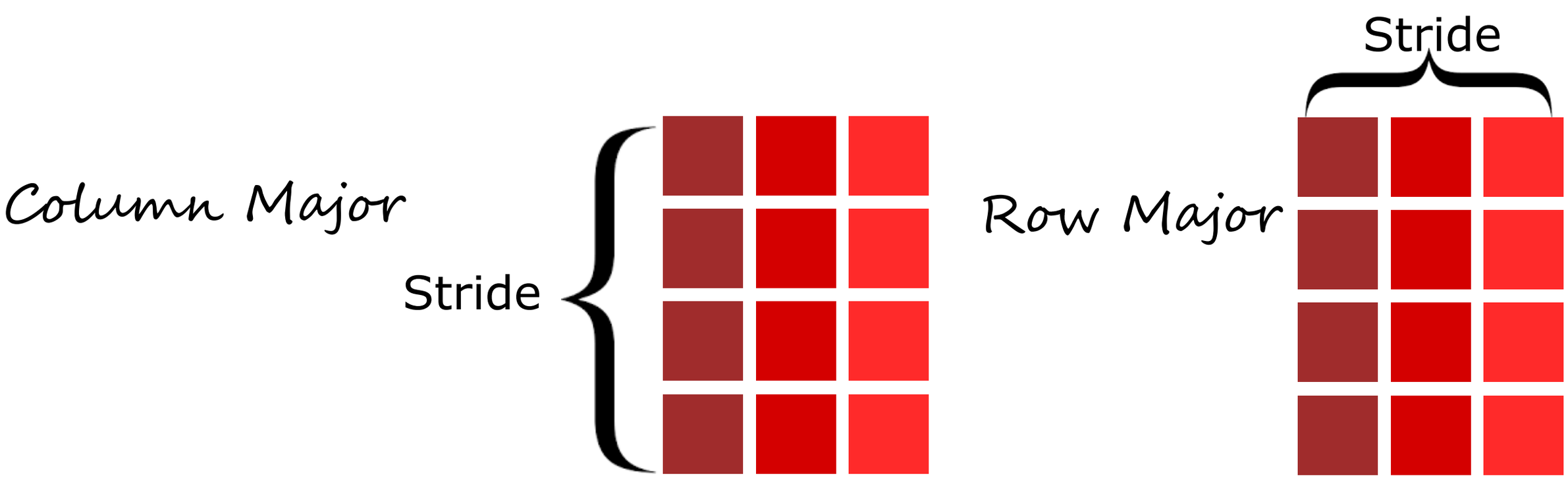
- Propose tensor operation kernel: StridedBatchedGEMM
  - Library-based approaches that avoid memory movement
  - Constant-strided BatchedGEMM that has more optimization opportunities
- Provide evaluation strategies for tensor contractions
- Apply to tensor decomposition
- Introduce TensorLy: Tensor learning in python

BLAS Operations

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

Level 1	**Level 2**	**Level 3**
$y \leftarrow \alpha x + y$	$y \leftarrow \alpha op(A)x + \beta y$	$C \leftarrow \alpha op(A)op(B) + \beta C$

Example:
GEMM(ORDER, TRANSA, TRANSB, M, N, K, α, A, LDA, B, LDB, β, C, LDC)



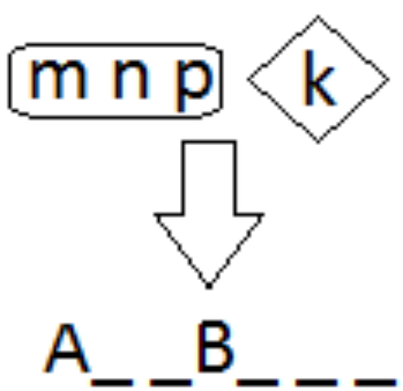
Extended BLAS Operator

Focusing: one-index contraction

Extended BLAS Kernel for tensor one-index contraction

$$C = \alpha op(A) op(B) + \beta C$$

$$C_{mnp} = A_{**} \times B_{***}$$



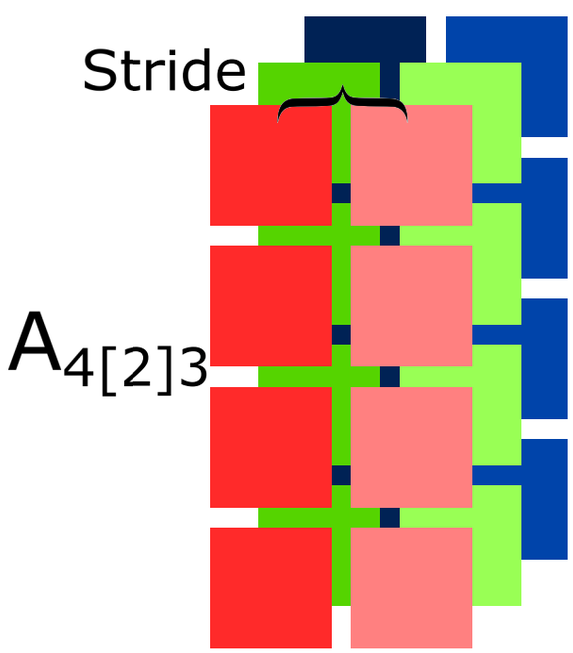
If fixing indices of C, there are total 3 x 2 x 3 x 2 x 1 = 36 cases.

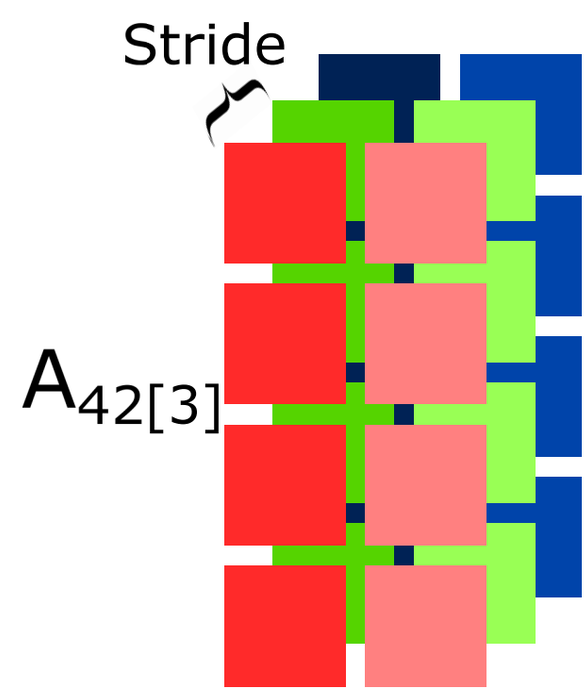
Extended BLAS Operator

Case	Contraction	Kernel1	Kernel2	Kernel3
1.1	$A_{mk}B_{knp}$	$C_{m(np)} = A_{mk}B_{k(np)}$	$C_{mn[p]} = A_{mk}B_{kn[p]}$	$C_{m[n]p} = A_{mk}B_{k[n]p}$

Table: Example: possible mappings to Level 3 BLAS routines

StridedBatchedGEMM(ORDER, TRANSA, TRANSB, M, N, K, α, A, LDA, LOA, B, LDB, LOB, β, C, LDC, LOC, P)





Example

Case	Contraction	Kernel1	Kernel2	Kernel3	Case	Contraction	Kernel1	Kernel2
1.1	$A_{mk}B_{knp}$	$C_{m(np)} = A_{mk}B_{k(np)}$	$C_{mn[p]} = A_{mk}B_{kn[p]}$	$C_{m[n]p} = A_{mk}B_{k[n]p}$	4.1	$A_{kn}B_{kmp}$	$C_{mn[p]} = B^{\top}_{km[p]}A_{kn}$	
1.2	$A_{mk}B_{kpn}$	$C_{mn[p]} = A_{mk}B_{k[p]n}$	$C_{m[n]p} = A_{mk}B_{kp[n]}$		4.2	$A_{kn}B_{kpm}$	$C_{mn[p]} = B^{\top}_{k[p]m}A_{kn}$	
1.3	$A_{mk}B_{nkp}$	$C_{mn[p]} = A_{mk}B^{\top}_{nk[p]}$			4.3	$A_{kn}B_{mkp}$	$C_{mn[p]} = B_{mk[p]}A_{kn}$	
1.4	$A_{mk}B_{pkn}$	$C_{m[n]p} = A_{mk}B^{\top}_{pk[n]}$			4.4	$A_{kn}B_{pkm}$	$TRANS(A^{\top}_{kn}B^{\top}_{pk[m]})$	$C_{[m][n]p} = B_{pk[m]}A_{k[n]}$
1.5	$A_{mk}B_{npk}$	$C_{m(np)} = A_{mk}B^{\top}_{(np)k}$	$C_{mn[p]} = A_{mk}B^{\top}_{n[p]k}$		4.5	$A_{kn}B_{mpk}$	$C_{mn[p]} = B_{m[p]k}A_{kn}$	
1.6	$A_{mk}B_{pnk}$	$C_{m[n]p} = A_{mk}B^{\top}_{p[n]k}$			4.6	$A_{kn}B_{pmk}$	$TRANS(A^{\top}_{kn}B^{\top}_{p[m]k})$	$C_{[m][n]p} = B_{p[m]k}A_{k[n]}$
2.1	$A_{km}B_{knp}$	$C_{m(np)} = A^{\top}_{km}B_{k(np)}$	$C_{mn[p]} = A^{\top}_{km}B_{kn[p]}$	$C_{m[n]p} = A^{\top}_{km}B_{k[n]p}$	5.1	$A_{pk}B_{kmn}$	$C_{(mn)p} = B^{\top}_{k(mn)}A^{\top}_{pk}$	$C_{m[n]p} = B^{\top}_{km[n]}A^{\top}_{pk}$
2.2	$A_{km}B_{kpn}$	$C_{mn[p]} = A^{\top}_{km}B_{k[p]n}$	$C_{m[n]p} = A^{\top}_{km}B_{kp[n]}$		5.2	$A_{pk}B_{knm}$	$C_{m[n]p} = B^{\top}_{k[n]m}A^{\top}_{pk}$	
2.3	$A_{km}B_{nkp}$	$C_{mn[p]} = A^{\top}_{km}B^{\top}_{nk[p]}$			5.3	$A_{pk}B_{mkn}$	$C_{m[n]p} = B_{mk[n]}A^{\top}_{pk}$	
2.4	$A_{km}B_{pkn}$	$C_{m[n]p} = A^{\top}_{km}B^{\top}_{pk[n]}$			5.4	$A_{pk}B_{nkm}$	$TRANS(B_{nk[m]}A^{\top}_{pk})$	$C_{[m]n[p]} = B_{nk[m]}A_{[p]k}$
2.5	$A_{km}B_{npk}$	$C_{m(np)} = A^{\top}_{km}B^{\top}_{(np)k}$	$C_{mn[p]} = A^{\top}_{km}B^{\top}_{n[p]k}$		5.5	$A_{pk}B_{mnk}$	$C_{(mn)p} = B_{(mn)k}A^{\top}_{pk}$	$C_{m[n]p} = B_{m[n]k}A^{\top}_{pk}$
2.6	$A_{km}B_{pnk}$	$C_{m[n]p} = A^{\top}_{km}B^{\top}_{p[n]k}$			5.6	$A_{pk}B_{nmk}$	$TRANS(B_{n[m]k}A^{\top}_{pk})$	$C_{[m]n[p]} = B_{n[m]k}A_{[p]k}$
3.1	$A_{nk}B_{kmp}$	$C_{mn[p]} = B^{\top}_{km[p]}A^{\top}_{nk}$			6.1	$A_{kp}B_{kmn}$	$C_{(mn)p} = B^{\top}_{k(mn)}A_{kp}$	$C_{m[n]p} = B^{\top}_{km[n]}A_{kp}$
3.2	$A_{nk}B_{kpm}$	$C_{mn[p]} = B^{\top}_{k[p]m}A^{\top}_{nk}$			6.2	$A_{kp}B_{knm}$	$C_{m[n]p} = B^{\top}_{k[n]m}A_{kp}$	
3.3	$A_{nk}B_{mkp}$	$C_{mn[p]} = B_{mk[p]}A^{\top}_{nk}$			6.3	$A_{kp}B_{mkn}$	$C_{m[n]p} = B_{mk[n]}A_{kp}$	
3.4	$A_{nk}B_{pkm}$	$TRANS(A_{nk}B^{\top}_{pk[m]})$	$C_{[m][n]p} = B_{pk[m]}A_{[n]k}$		6.4	$A_{kp}B_{nkm}$	$TRANS(B_{nk[m]}A_{kp})$	$C_{[m]n[p]} = B_{nk[m]}A_{k[p]}$
3.5	$A_{nk}B_{mpk}$	$C_{mn[p]} = B_{m[p]k}A^{\top}_{nk}$			6.5	$A_{kp}B_{mnk}$	$C_{(mn)p} = B_{(mn)k}A_{kp}$	$C_{m[n]p} = B_{m[n]k}A_{kp}$
3.6	$A_{nk}B_{pmk}$	$TRANS(A_{nk}B^{\top}_{p[m]k})$	$C_{[m][n]p} = B_{p[m]k}A_{[n]k}$		6.6	$A_{kp}B_{nmk}$	$TRANS(B_{n[m]k}A_{kp})$	$C_{[m]n[p]} = B_{n[m]k}A_{k[p]}$

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

Example

Case	Contraction	Kernel1	Kernel2	Kernel3
1.1	$A_{mk}B_{knp}$	$C_{m(np)} = A_{mk}B_{k(np)}$	$C_{mn[p]} = A_{mk}B_{kn[p]}$	$C_{m[n]p} = A_{mk}B_{k[n]p}$
6.1	$A_{kp}B_{kmn}$	$C_{(mn)p} = B^{\top}_{k(mn)}A_{kp}$	$C_{m[n]p} = B^{\top}_{km[n]}A_{kp}$	

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

```
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: $C_{m[n]p} = B^{\top}_{km[n]}A_{kp}$

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

Analysis

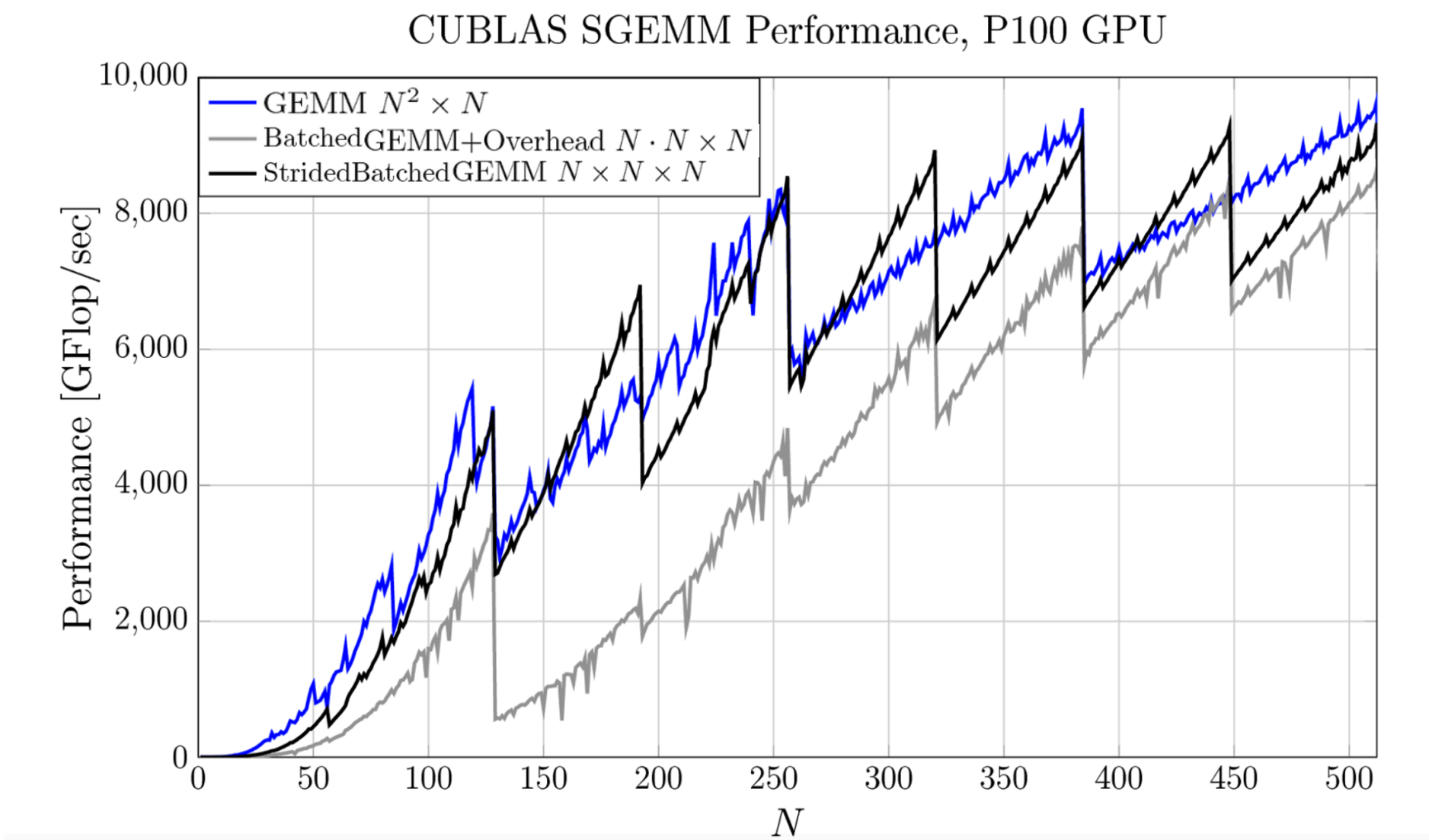
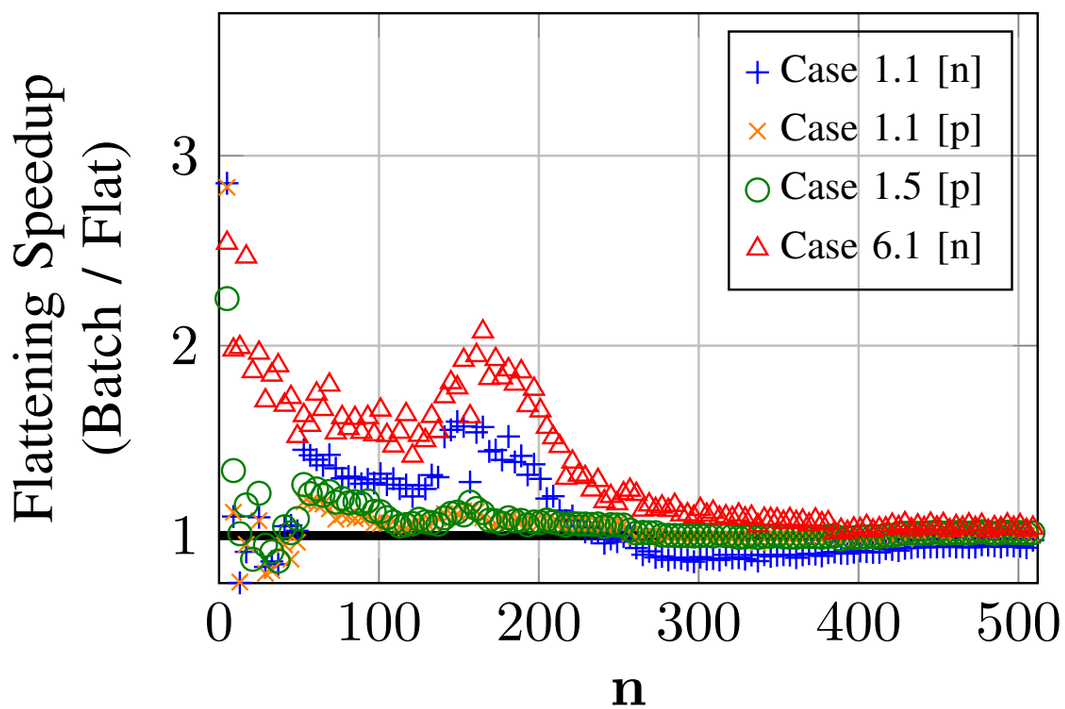


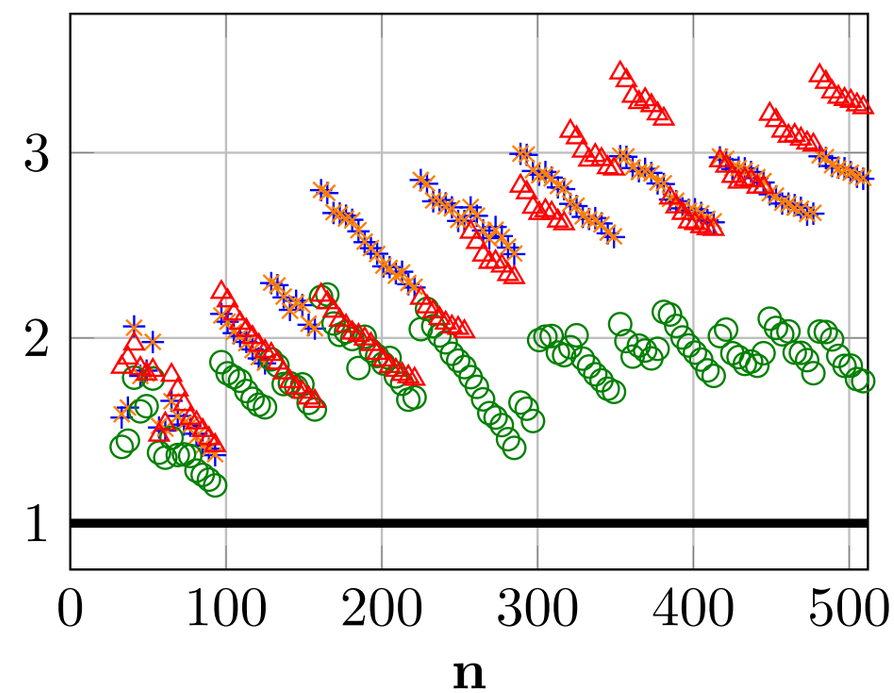
Figure: Performance of three strategies for computing N matrix-matrix multiplications of size NxN.

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

Analysis

Flatten v.s. SBGEMM

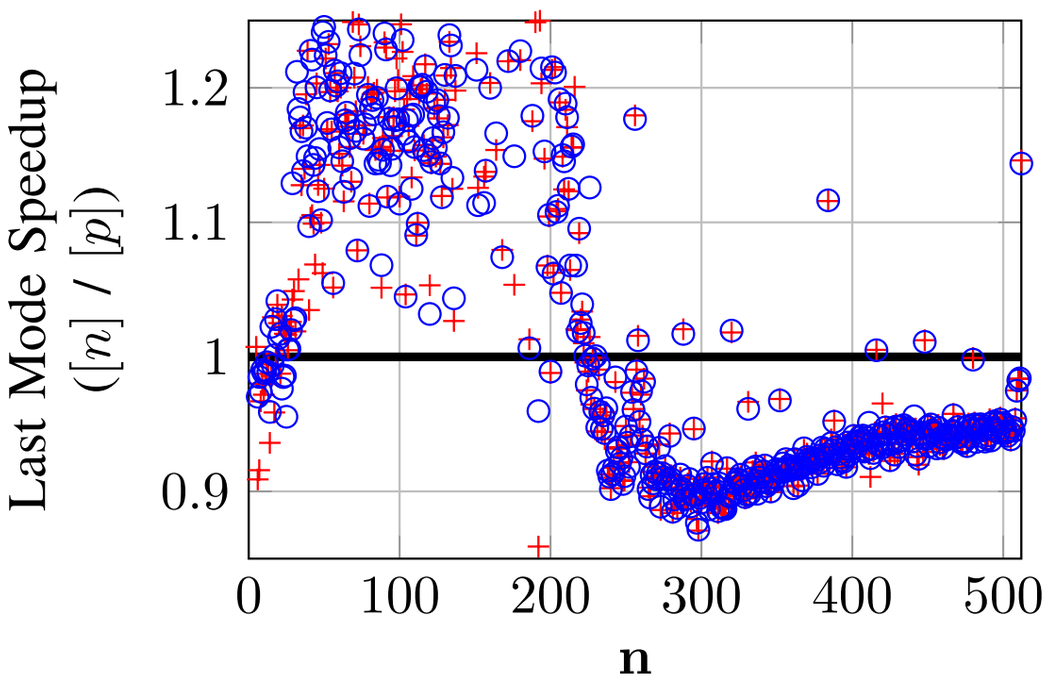


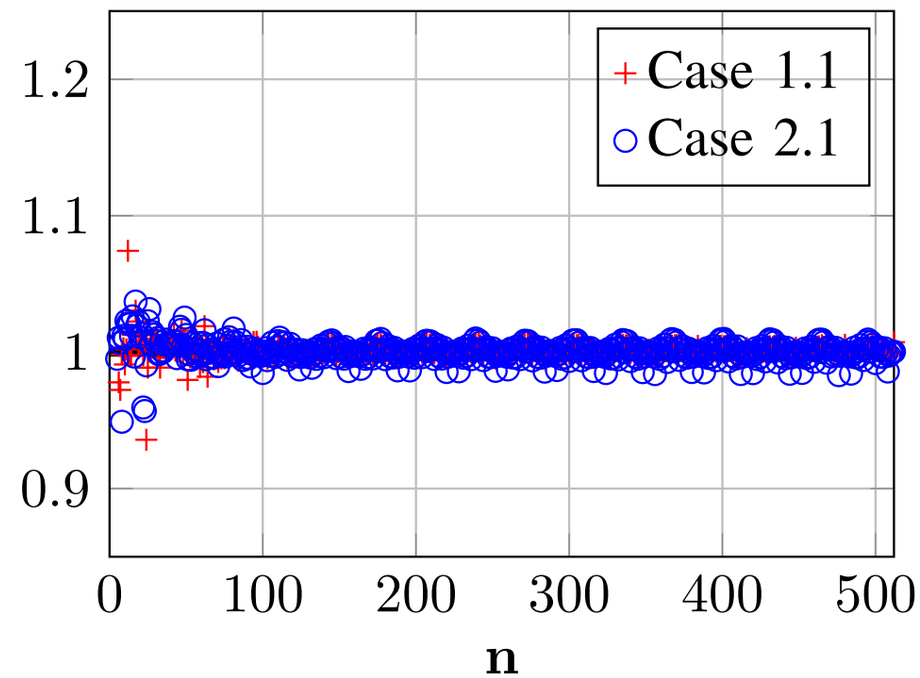


Prefer flatten than SBGEMM

Analysis

Batching in last mode v.s. middle mode

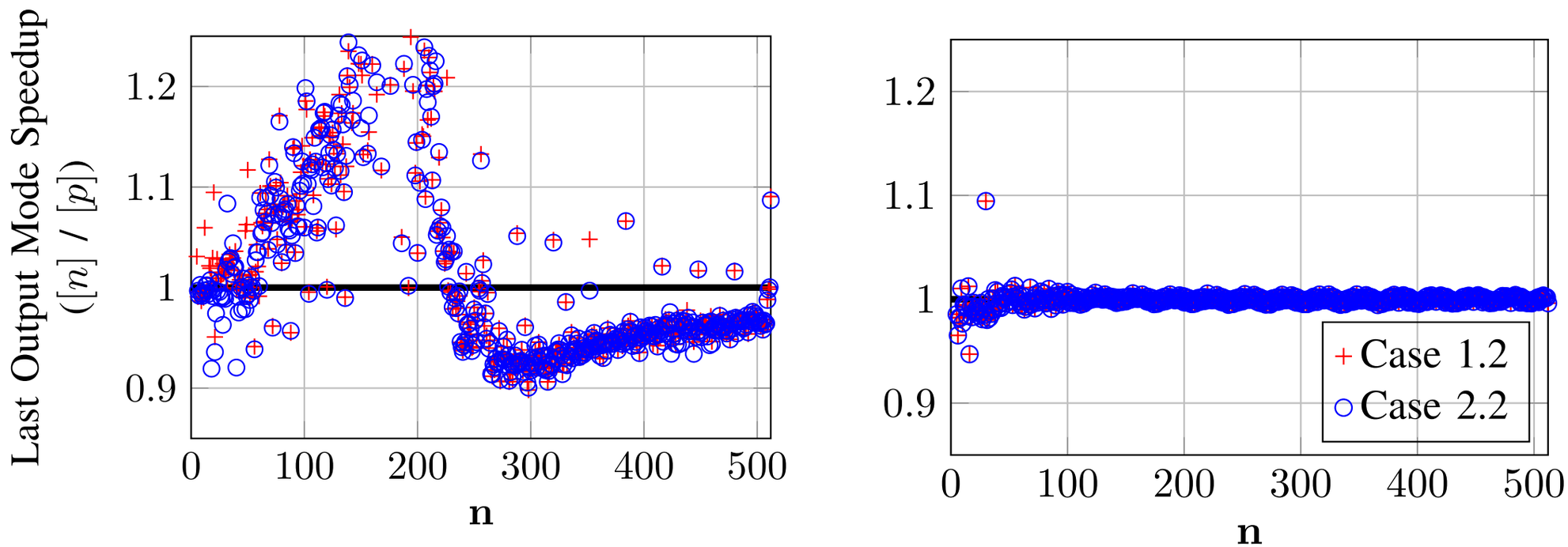




On CPU, it's better to batch in last mode when tensor size is small/moderate

Analysis

Mixed mode batching



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

Analysis

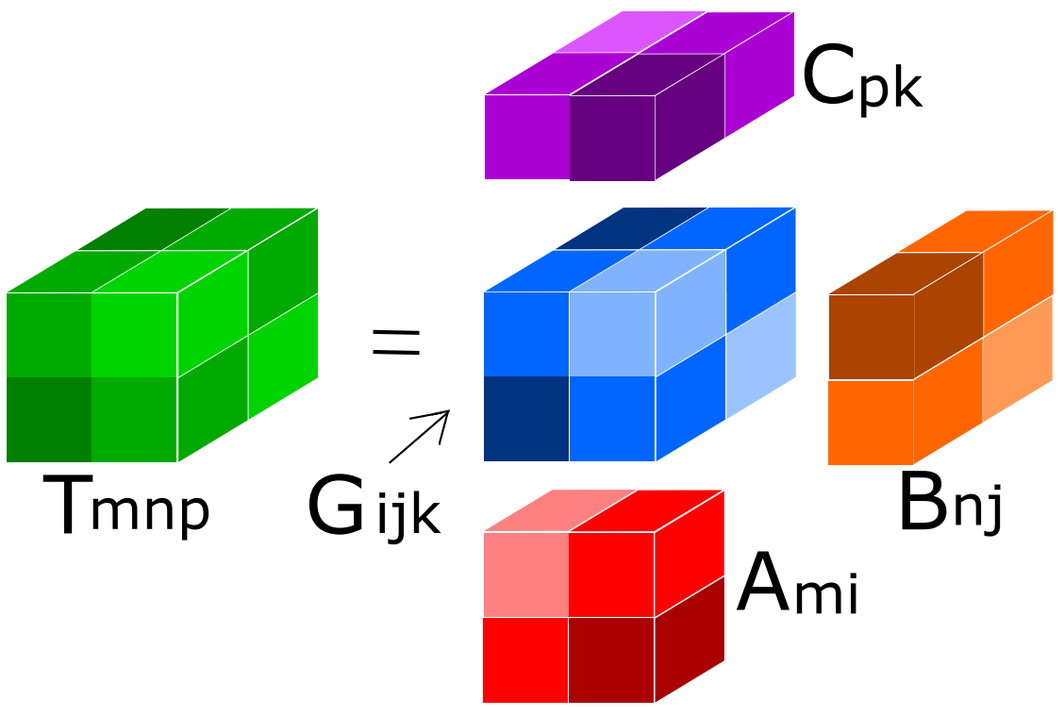
- Flatten V.S. SBGEMM
  - A single large GEMM is more efficient
  - Flatten modes whenever possible

- 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:

- $Y_{mjk} = T_{mnp} B^t_{nj} C^t_{pk}$
- $Y_{ink} = T_{mnp} A^{t+1}_{mi} C^t_{pk}$
- $Y_{ijp} = T_{mnp} B^{t+1}_{nj} A^{t+1}_{mi}$

Application: Tucker Decomposition

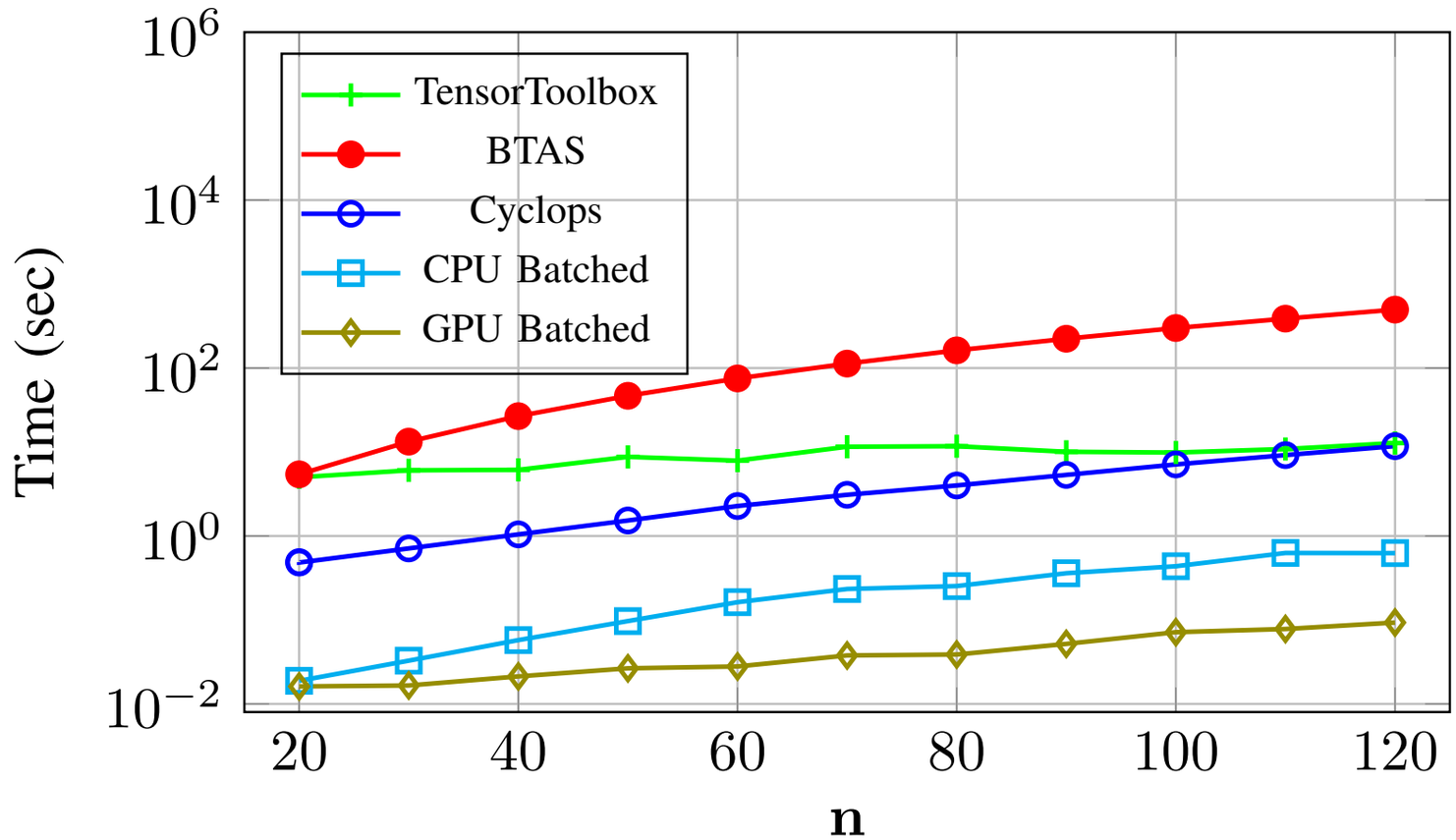


Figure: Performance on Tucker decomposition.

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 Lond
Anima Anandkumar, Caltech

- **Open source**

  Homepage: http://tensorly.org/dev/

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

  Suitable for academic / industrial applications

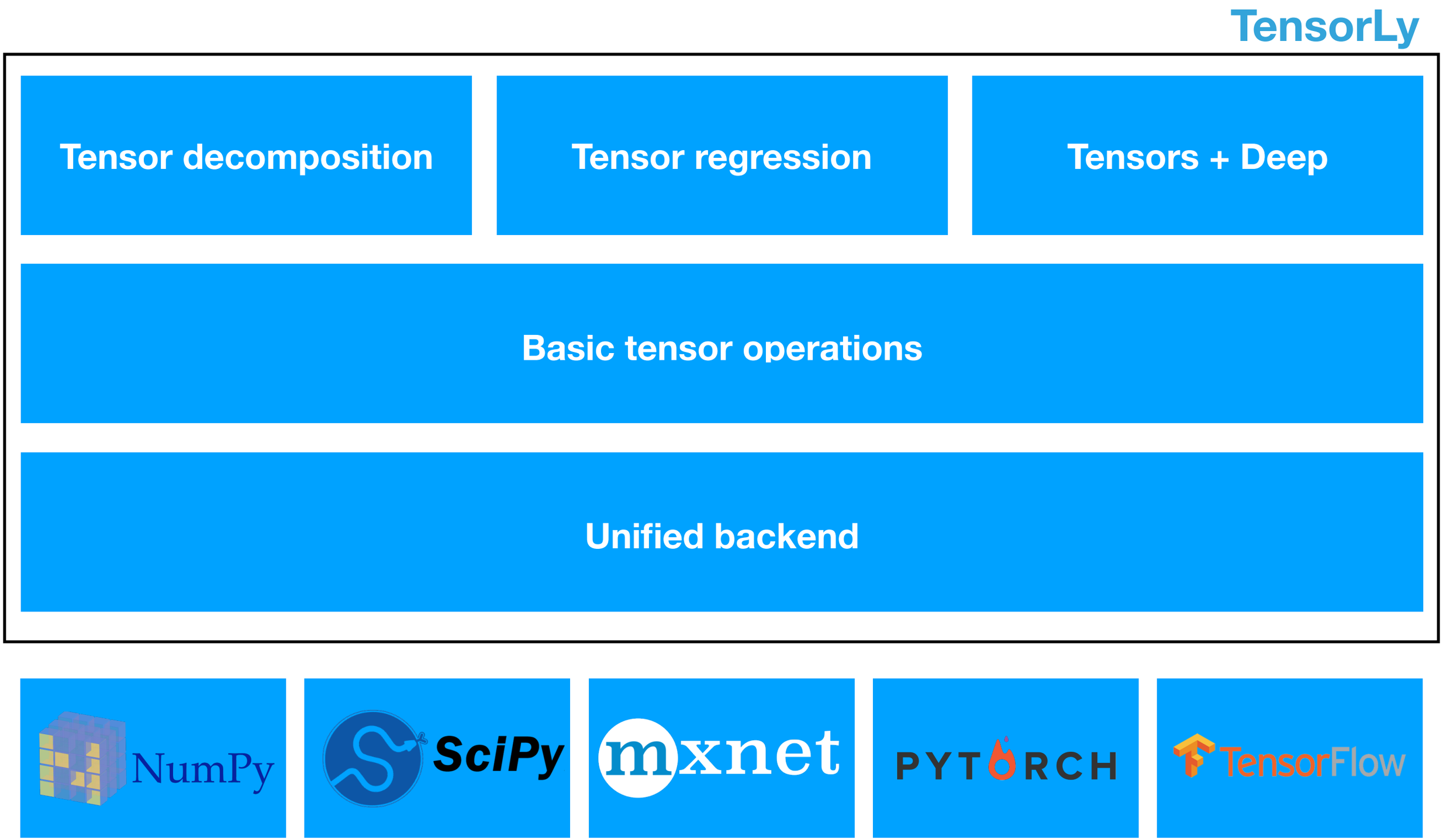
- **Reliability and easy to use**

  Depends only on NumPy, SciPy [Optionally Matplotlib, MXNet and PyTorch]

  Exhaustive documentation, Unit-testing for all functions

  Fast

User-friendly API



TensorLy Operators

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

`khatri_rao` (matrices[, skip_matrix, reverse])	Khatri-Rao product of a list of matrices
`kronecker` (matrices[, skip_matrix, reverse])	Kronecker product of a list of matrices
`mode_dot` (tensor, matrix_or_vector, mode)	n-mode product of a tensor and a matrix or vector at the specified mode
`multi_mode_dot` (tensor, matrix_or_vec_list[, ...])	n-mode product of a tensor and several matrices or vectors over several modes
`proximal.soft_thresholding` (tensor, threshold)	Soft-thresholding operator
`proximal.svd_thresholding` (matrix, threshold)	Singular value thresholding operator
`proximal.procrustes` (matrix)	Procrustes operator
`inner` (tensor1, tensor2[, n_modes])	Generalised inner products between tensors

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

`parafac` (tensor, rank[, n_iter_max, init, ...])	CANDECOMP/PARAFAC decomposition via alternating least squares (ALS)
`non_negative_parafac` (tensor, rank[, ...])	Non-negative CP decomposition
`tucker` (tensor[, rank, ranks, n_iter_max, ...])	Tucker decomposition via Higher Order Orthogonal Iteration (HOI)
`partial_tucker` (tensor, modes[, rank, ...])	Partial tucker decomposition via Higher Order Orthogonal Iteration (HOI)
`non_negative_tucker` (tensor, rank[, ...])	Non-negative Tucker decomposition
`robust_pca` (X[, mask, tol, reg_E, reg_J, ...])	Robust Tensor PCA via ALM with support for missing values

TensorLy Backend

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

```
import tensorly as tl

T = tl.tensor([[1, 2, 3], [4, 5, 6]])
tl.tenalg.kronecker([T, T])
tl.clip(T, a_min=2, a_max=5)

tl.set_backend('mxnet')
T = tl.tensor([[1, 2, 3], [4, 5, 6]])

tl.set_backend('pytorch')
T = tl.tensor([[1, 2, 3], [4, 5, 6]])
```

NumPy ndarray

MXNet NDArray

PyTorch FloatTensor

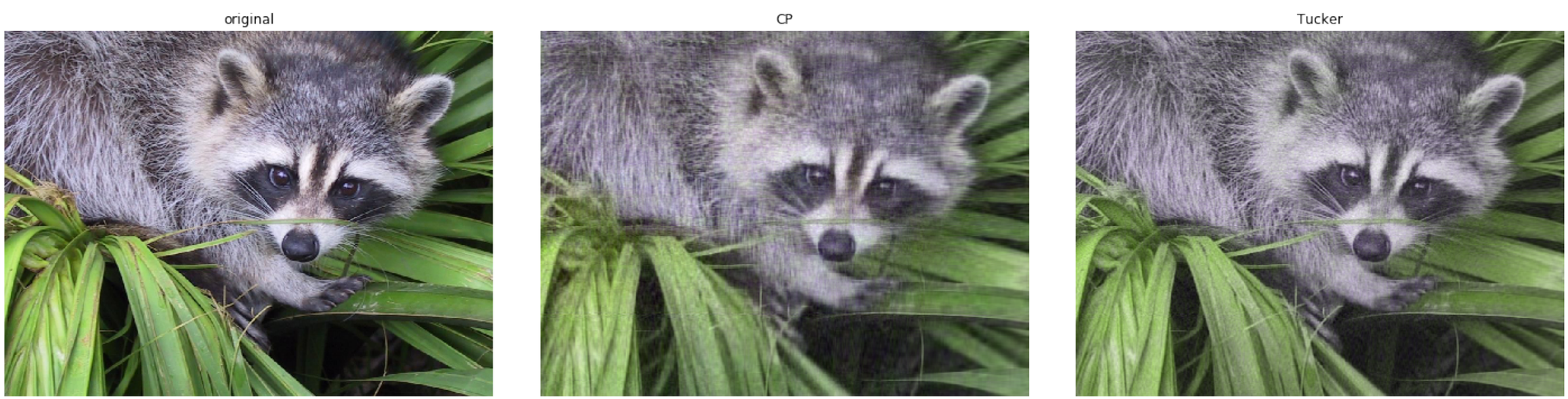
TensorLy Example

```
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

```
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?