# A Linear-algebraic Approach to Distributed Deep Learning

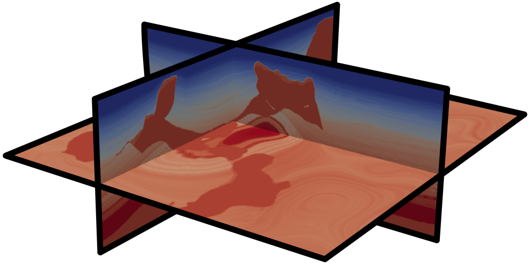
Russell J. Hewett
Mathematics & CMDA, Virginia Tech

SLIM Group Seminar

February 19, 2021

- FWI is a PDE-constrained optimization method for subsurface recovery

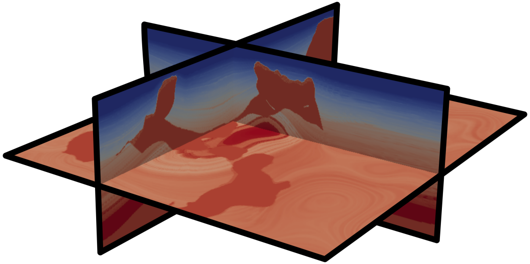
$$\arg\min_{m} \; d(u_{\text{obs}}, u) \text{ s.t. } \mathcal{L}(m, f; u) = 0$$



- Recover, e.g., subsurface velocity $m$ from seismic data $u_{\text{obs}}$ through physics $\mathcal{L}$
- Applications in hydrocarbon exploration, carbon sequestration, aquifer monitoring, etc.

- FWI is a PDE-constrained optimization method for subsurface recovery

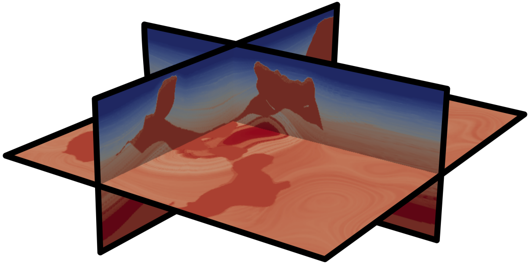
$$\arg\min_{m} \; d(u_{\text{obs}}, u) \text{ s.t. } \mathcal{L}(m, f; u) = 0$$



- Data $u_{\text{obs}}$ is petascale, $\sim 10^5$ source functions $f$
- Computation made feasible by data parallelism

- FWI is a PDE-constrained optimization method for subsurface recovery

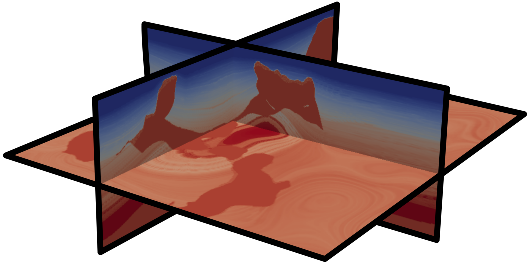
$$\arg\min_{m} \; d(u_{\text{obs}}, u) \text{ s.t. } \mathcal{L}(m, f; u) = 0$$



- 3D elastic physics $\mathcal{L}$ over $\sim 35\text{km} \times 40\text{km} \times 15\text{km}$ domain
- Simulation is feasible due to pre-exascale high-performance computing systems

- FWI is a PDE-constrained optimization method for subsurface recovery

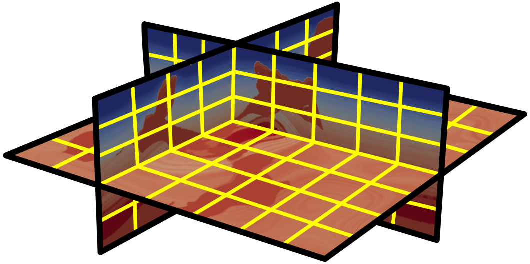
$$\arg\min_{m} \; d(u_{\mathsf{obs}}, u) \text{ s.t. } \mathcal{L}(m, f; u) = 0$$



- Model $m$ is gigascale for $20\text{m}$ grid scale

- FWI is a PDE-constrained optimization method for subsurface recovery

$$\arg\min_{m} \; d(u_{\text{obs}}, u) \text{ s.t. } \mathcal{L}(m, f; u) = 0$$



- Model $m$ is gigascale for $20\mathrm{m}$ grid scale
- Computation made feasible by model parallelism via domain decomposition

Mathematical and computational challenges for modern FWI:

- Solution time is weeks to months
- Requires significant hands-on, expert intervention
- Uncertainty Quantification is generally infeasible
  - Solution to the inverse problem is but one of many possible estimates
  - Very important to characterize distribution of solutions

# FWI: Full-waveform Inversion

Mathematical and computational challenges for modern FWI:

- Solution time is weeks to months
- Requires significant hands-on, expert intervention
- Uncertainty Quantification is generally infeasible
  - Solution to the inverse problem is but one of many possible estimates
  - Very important to characterize distribution of solutions

Modern solution to modern challenges: Throw machine learning at it!

Scientific Machine Learning (SciML)

- Intersection of Computational Science & Engineering and Machine Learning
- Take our already large problems and make them larger!

# Scientific Machine Learning (SciML)

- Intersection of Computational Science & Engineering and Machine Learning
- Take our already large problems and make them larger!
- Scientific problems:
  - Quasi-regular computation
  - Massive compute
  - Massive models
  - Massive data
  - Driven by physics
- Deep learning:
  - Irregular computation
  - Massive compute
  - (Increasingly) massive models
  - Massive data
  - Driven by data

# PDE-Constrained Optimization and Deep Learning Models

- PDE Constrained Optimization

$$\arg\min_{m} \; d(u_{\text{obs}}, u) \text{ s.t. } \mathcal{L}(m, f; u) = 0$$

- Deep Prior

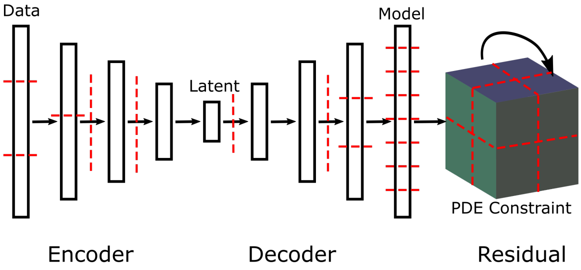
$$\arg\min_{\theta} \; d(u_{\text{obs}}, u) \text{ s.t. } \mathcal{L}(\mathcal{N}(\theta; z), f; u) = 0$$

- Physics-informed NN

$$\arg\min_{\theta} \; d(u_{\text{obs}}, \mathcal{N}(\theta; z)) \text{ s.t. } \mathcal{L}(m, f; \mathcal{N}(\theta; z)) = 0$$

Consider a DNN surrogate for subsurface models (deep prior) or wavefield (PINN).

- We *functionally* cannot accept smaller models.
- If classical problem requires domain decomposition, then so must the DNN!
- These networks also have extremely large number of DoFs, too!



In the face of high computational cost, seek parallelism

- Parallelism in PDE-constrained optimization
  - Data parallelism over multiple source functions
  - Accelerated compute kernels on modern hardware
  - Model parallelism over physical domain

In the face of high computational cost, seek parallelism

- Parallelism in PDE-constrained optimization
  - Data parallelism over multiple source functions
  - Accelerated compute kernels on modern hardware
  - Model parallelism over physical domain
- Parallelism in deep learning
  - Data parallelism over multiple inputs
  - Accelerated compute kernels on modern hardware

In the face of high computational cost, seek parallelism

- Parallelism in PDE-constrained optimization
  - Data parallelism over multiple source functions
  - Accelerated compute kernels on modern hardware
  - Model parallelism over physical domain
- Parallelism in deep learning
  - Data parallelism over multiple inputs
  - Accelerated compute kernels on modern hardware
- What are we missing?
  - Model parallelism!
  - More challenging than for PDEs due to causality and lack of physical domain

DistDL
Model Parallelism
for PyTorch

Joint work w/
- Thomas Grady (Math+CMDA '20)
- Daniel Hagialigol (CMDA '22)
- Jacob Merizian (Math+CS '20)
- CMDA Capstone team (2020)

- A DNN is the composition of many non-linear functions

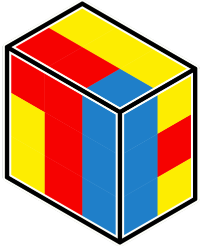
$$\mathcal{N}(\theta; z) = f_{n-1}(\theta_{n-1}, f_{n-2}(\theta_{n-2}, \ldots f_1(\theta_1, f_0(\theta_0, z))))$$

- Each function $f_i$ is a *layer*
- All layer inputs, outputs, and internal parameters (weights) are high-order tensors

- A DNN is the composition of many non-linear functions

$$\mathcal{N}(\theta; z) = f_{n-1}(\theta_{n-1}, f_{n-2}(\theta_{n-2}, \ldots f_1(\theta_1, f_0(\theta_0, z))))$$

- Each function $f_i$ is a *layer*
- All layer inputs, outputs, and internal parameters (weights) are high-order tensors



- Typical input or output tensor:

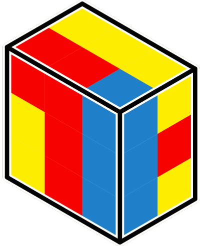
$$x \in \mathbb{R}^{\wedge} \{b \times c \times n_{i-1} \times \cdots n_1 \times n_0\}$$

  - $b$ is the *batch* dimension
  - $c$ is the *channel* dimension
  - $n_i$ is the i^th *feature* dimension

- A DNN is the composition of many non-linear functions

$$\mathcal{N}(\theta; z) = f_{n-1}(\theta_{n-1}, f_{n-2}(\theta_{n-2}, \ldots f_1(\theta_1, f_0(\theta_0, z))))$$

- Each function $f_i$ is a *layer*
- All layer inputs, outputs, and internal parameters (weights) are high-order tensors



- Typical convolution weight tensor:

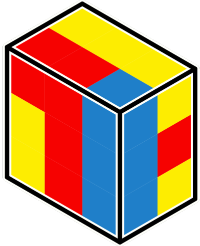
$$w \in \mathbb{R}^{\wedge}\{c_{\text{out}} \times c_{\text{in}} \times k_{i-1} \times \cdots k_1 \times k_0\}$$

  - $c_{\text{out}}$ is the output *channel* dimension
  - $c_{\text{in}}$ is the input *channel* dimension
  - $k_i$ is the i$^{\text{th}}$ dimension's kernel size

- A DNN is the composition of many non-linear functions

$$\mathcal{N}(\theta; z) = f_{n-1}(\theta_{n-1}, f_{n-2}(\theta_{n-2}, \ldots f_1(\theta_1, f_0(\theta_0, z))))$$

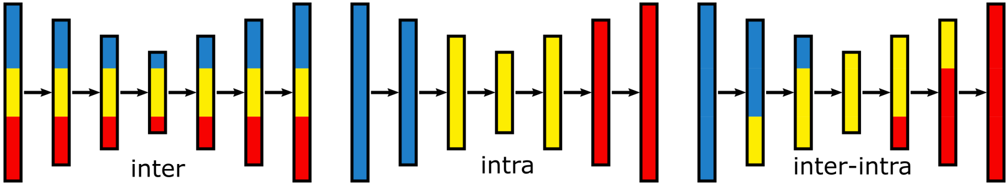
- Each function $f_i$ is a *layer*
- All layer inputs, outputs, and internal parameters (weights) are high-order tensors



- Typical linear or affine weight tensor:

$$W \in \mathbb{R}^\wedge \{c_{\text{out}} \times c_{\text{in}}\}$$

  - $c_{\text{out}}$ is the output *channel* dimension
  - $c_{\text{in}}$ is the input *channel* dimension




Data parallelism:

- Parallelize over the batch dimension

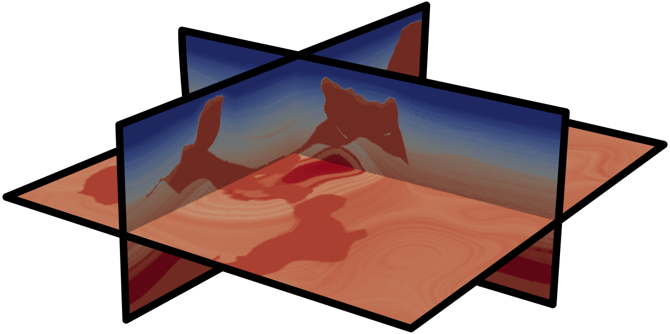
Pipelining:

- Parallelize across layers (*intra*-layer)

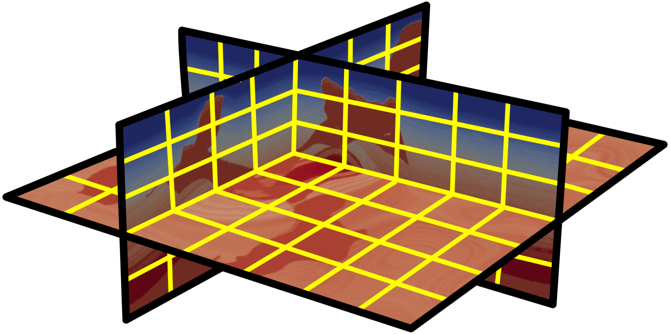
Model parallelism:

- Parallelize within layers (*inter*-layer)
- Parallelize within and across layers (*inter-intra*-layer)

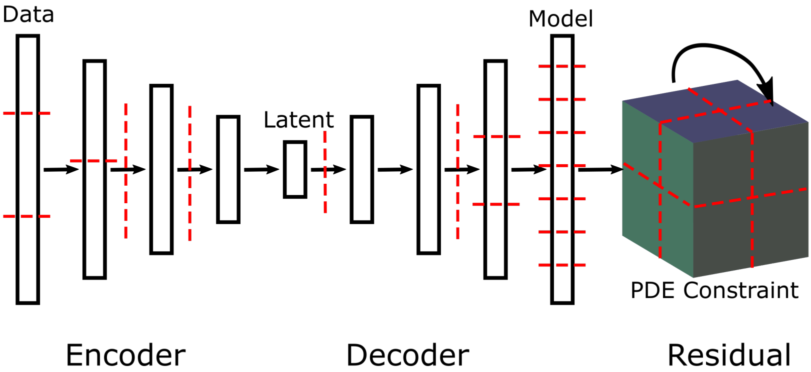
In classical PDE-constrained optimization:



In classical PDE-constrained optimization:

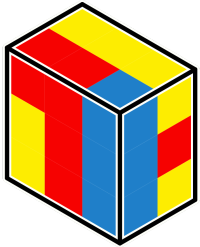


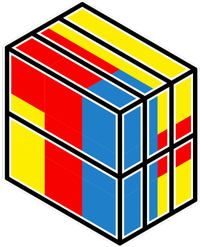
In deep prior or PINN:

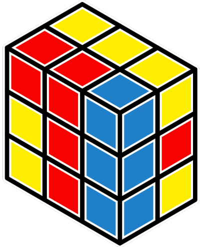


In general:

- No natural domain to decompose
  - Each tensor has generally unique structure: dimensions vary wildly!
- Partition the input/output tensors
  - Net result of decomposing classical domains
  - Partially induced by individual layer structure
- Make intelligent choices for decomposing layer parameter tensors
  - HPC architecture-induced
  - Partially induced by input structure







To motivate our approach, consider how deep neural networks are constructed and trained.

- A DNN is the composition of many non-linear functions

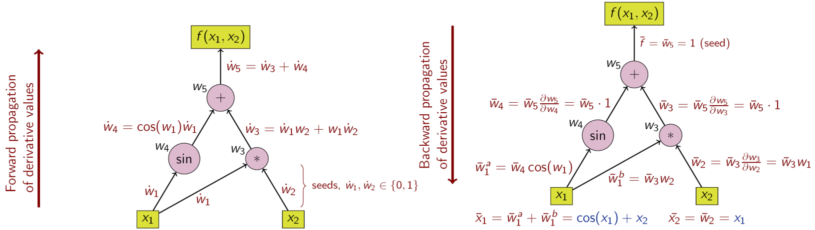
$$\mathcal{N}(\theta; z) = f_{n-1}(\theta_{n-1}, f_{n-2}(\theta_{n-2}, \dots f_1(\theta_1, f_0(\theta_0, z))))$$

- Computing the gradient is the composition of many linear functions

$$N^* \delta\theta = \mathsf{f}_0^* \mathsf{f}_1^* \dots \mathsf{f}_{n-2}^* \mathsf{f}_{n-1}^* \delta\theta_{n-1}$$

- For each layer (function)
  - An implementation of the action of forward function, $f$
  - An implementation of the action of the adjoint of the Jacobian of $f$, $\mathsf{f}^*$

- Given a function, or an implementation of a function, $f$
- Two modes:
  - Forward mode: application of the Jacobian of $f$, f
  - Adjoint mode: application of the adjoint of the Jacobian of $f$, f*




Forward Mode AD (PD) and Reverse Mode AD (PD)

- Implementation strategies:
  - Pre-compiled source transformation (syntactic)
  - Just-in-time construction of computation graph (semantic)

## Defining Network Layers

- In deep learning, automatic differentiation is used to get $f^*$
- But distributed computing libraries, like MPI, do not easily integrate with AD tools!

- Issues with syntactic methods / source transformation:
  - Blocking communication
    - What are the adjoints of `MPI_Send`, `MPI_Recv`, `MPI_Bcast`, etc.?
  - Nonblocking communication
    - What are the adjoints of `MPI_Isend`, `MPI_Irecv`, `MPI_Ibcast`, `MPI_Wait`, etc.?
  - No one AD's the communication library (or the network!) (or the switch!)

# Defining Network Layers

- In deep learning, automatic differentiation is used to get $f^*$
- But distributed computing libraries, like MPI, do not easily integrate with AD tools!

- Issues with syntactic methods / source transformation:
  - Blocking communication
    - What are the adjoints of `MPI_Send`, `MPI_Recv`, `MPI_Bcast`, etc.?
  - Nonblocking communication
    - What are the adjoints of `MPI_Isend`, `MPI_Irecv`, `MPI_Ibcast`, `MPI_Wait`, etc.?
  - No one AD's the communication library (or the network!) (or the switch!)

- How can we compute the adjoints if we can't differentiate operations?
  - Exploit semantics – mathematical meaning
  - Linear functions are their own Jacobians
  - We have freedom to choose inner products
  - PyTorch supports this approach

- A computer's memory can represent $\mathbb{F}^n$, a floating-point subset of $\mathbb{R}^n$.
- Consider $\langle \mathbf{a}, \mathbf{b} \rangle_{\mathbb{F}^n} = \sum_{i=0}^{n-1} a_i b_i$ to be the inner product
- Then the adjoint (of the Jacobian) of a linear operator arises by requiring satisfaction of

$$\langle A\mathbf{x}, \mathbf{y} \rangle_{\mathbb{F}^n} = \langle \mathbf{x}, A^*\mathbf{y} \rangle_{\mathbb{F}^m}$$

- When we have such a trivial inner product, $A^* = A^T$

We can exploit this to define three key operations on memory and data stored there:

*Allocate* (and *Deallocate*)

- Assume $\mathbf{x}_a$ has been allocated. We need more space, $\mathbf{x}_b = \mathbf{0}_b$.
- Allocation is an operator $A_b : \mathbb{F}^m \to \mathbb{F}^n$, and

$$A_b\mathbf{x} = \begin{bmatrix} I_a \\ O_b \end{bmatrix} \begin{bmatrix} \mathbf{x}_a \end{bmatrix} = \begin{bmatrix} \mathbf{x}_a \\ \mathbf{0}_b \end{bmatrix}$$

- The adjoint of allocation is found through the inner product

$$A_b^*\mathbf{y} = A_b^T\mathbf{y} = \begin{bmatrix} I_a & O_b \end{bmatrix} \begin{bmatrix} \mathbf{y}_a \\ \mathbf{y}_b \end{bmatrix} = \begin{bmatrix} \mathbf{y}_a \end{bmatrix}$$

- The adjoint of allocation is deallocation (and vice versa): $A_b^* = D_b$

We can exploit this to define three key operations on memory and data stored there:

*Clear*

- Sets a of a subset of allocated memory $\mathbf{x}$, $\mathbf{x}_b$ to $\mathbf{0}$
- Clear is an operator $K_b : \mathbb{F}^m \rightarrow \mathbb{F}^m$, and

$$K_b \mathbf{x} = \begin{bmatrix} I_a & \\ & O_b \end{bmatrix} \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{bmatrix} = \begin{bmatrix} \mathbf{x}_a \\ \mathbf{0}_b \end{bmatrix}$$

- Clear is self-adjoint, $K_b^* = K_b$

We can exploit this to define three key operations on memory and data stored there:

*Add*

- In-place summation $\mathbf{x}_a$ into $\mathbf{x}_b$
- Add is the operator $S_{a\to b} : \mathbb{F}^m \to \mathbb{F}^m$, and

$$S_{a\to b}\mathbf{x} = \begin{bmatrix} I_a & \\ I_a & I_b \end{bmatrix} \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{bmatrix} = \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_a + \mathbf{x}_b \end{bmatrix}$$

- The adjoint of add is found through the inner product

$$S^*_{a\to b}\mathbf{y} = \begin{bmatrix} I_a & I_b \\ & I_b \end{bmatrix} \begin{bmatrix} \mathbf{y}_a \\ \mathbf{y}_b \end{bmatrix} = \begin{bmatrix} \mathbf{y}_a + \mathbf{y}_b \\ \mathbf{y}_b \end{bmatrix} = S_{b\to a}\mathbf{y}$$

- The adjoint of add is also add, in reverse direction

Allocate, clear, and add give us two more important primitives:

*Copy*

- Can define in-place and out-of-place copy
- In-place copy is $C_{a\to b} : \mathbb{F}^m \to \mathbb{F}^m$

$$C_{a\to b} = \begin{bmatrix} I_a & O_b \\ I_a & O_b \end{bmatrix} = \begin{bmatrix} I_a & O_b \\ I_a & I_b \end{bmatrix} \begin{bmatrix} I_a & O_b \\ O_a & O_b \end{bmatrix} = S_{a\to b} K_b.$$

- Both copies and their adjoints are composition of previous primitives:

| In-place Copy | Out-of-place Copy |
|---|---|
| $C_{a\to b} = S_{a\to b} K_b$ | $C_{a\to b} = S_{a\to b} A_b$ |
| $C^*_{a\to b} = K_b S_{b\to a}$ | $C^*_{a\to b} = D_b S_{b\to a}$ |

- **Critical observation:** the adjoint of a copy involves a sum!

Allocate, clear, and add give us two more important primitives:

*Move*

- Can define in-place and out-of-place move
- In-place move is $M_{a\to b} : \mathbb{F}^m \to \mathbb{F}^m$

$$M_{a\to b} = \begin{bmatrix} O_a & O_b \\ I_a & O_b \end{bmatrix} = \begin{bmatrix} O_a & O_b \\ O_a & I_b \end{bmatrix} \begin{bmatrix} I_a & O_b \\ I_a & I_b \end{bmatrix} \begin{bmatrix} I_a & O_b \\ O_a & O_b \end{bmatrix} = K_a S_{a\to b} K_b.$$

- Both moves and their adjoints are composition of previous primitives:

In-place Move

$$M_{a\to b} = K_a S_{a\to b} K_b$$
$$M^*_{a\to b} = K_b S_{b\to a} K_a = M_{b\to a}$$

Out-of-place Move

$$M_{a\to b} = D_a S_{a\to b} A_b$$
$$M^*_{a\to b} = D_b S_{b\to a} A_a = M_{b\to a}$$

- This previous model applies for any memory on any computer!
- We were all probably thinking about a single node
  - Local memory
- But our definition of memory is very inclusive
  - Device memory
  - Whole system memory / remote nodes (HPC)
  - Local/remote disk

- We can compose these memory primitives to build a linear algebraic formulation of many parallel data movement operations
  - Send/receive (*)
  - Scatter/gather
  - Broadcast (*)
  - Sum-reduce
  - All-to-all / Transpose / Shuffle
  - All-(sum)-reduce

*Send/receive*

- A send-receive pair is merely a copy or move from one node/worker/task to another
- Choice of ‘copy’ or ‘move’ interpretation is semantic
  - Impacts structure of adjoint implementation
  - If data is used locally after a send, it is a copy
  - If data is not used locally after a send, it is a move
- If we interpret as a copy, the adjoint is still a sum then a clear

*Broadcast*

- Broadcast $\mathbf{x}_a$ to $k$ realizations $\mathbf{x}_0, \ldots \mathbf{x}_{k-1}$ or $k$ copy operations
- We construct the broadcast operator, $B_{a\to\{k\}}$,

$$B_{a\to\{k\}}\mathbf{x}_a = \begin{bmatrix} C_{a\to 0} \\ C_{a\to 1} \\ \vdots \\ C_{a\to k-1} \end{bmatrix} \mathbf{x}_a = \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_a \\ \vdots \\ \mathbf{x}_a \end{bmatrix} = \mathbf{x}_{\{k\}}.$$

- And its adjoint,

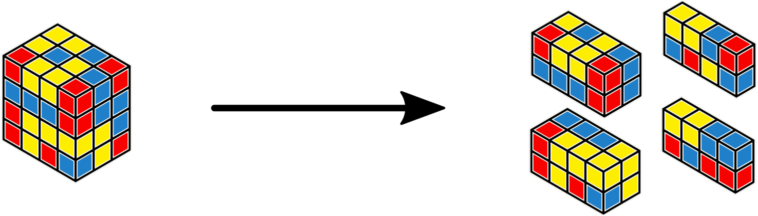
$$B^*_{a\to\{k\}}\mathbf{y}_{\{k\}} = \begin{bmatrix} C^*_{a\to 0} & C^*_{a\to 1} & \cdots & C^*_{a\to k-1} \end{bmatrix} \mathbf{y}_{\{k\}} = \sum_{i=0}^{k-1} K_i S_{i\to a}\mathbf{y}_i = \mathbf{y}_a.$$

- The adjoint of a broadcast is a sum-reduction

*Other Primitives*

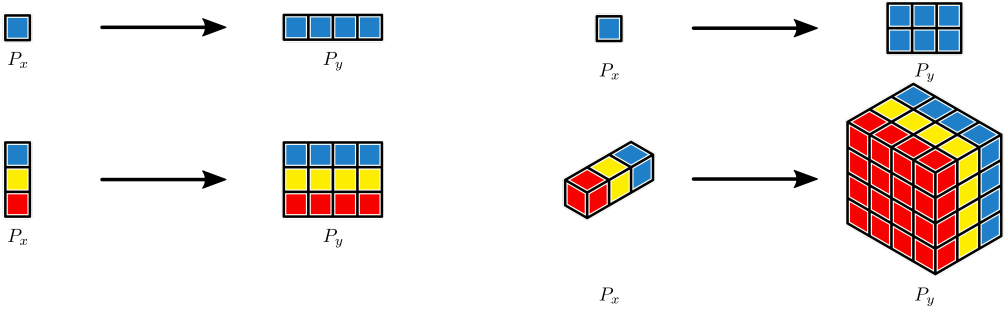
- Sum-reduce
  - adjoint is broadcast
- All-to-all/Shuffle/Transpose
  - Block matrix of copy/moves
  - adjoint is also transpose (not self-adjoint)
- Scatter (special case of Transpose)
  - adjoint is gather
- Gather (special case of Transpose)
  - adjoint is scatter
- All-sum-reduce
  - self-adjoint

- We want to generalize these ideas to deep learning (and PDEs too)
- We propose data movement primitives specific to high-order tensors
  - Broadcast
  - Sum-reduce
  - All-to-all / Transpose / Shuffle
  - Halo Exchange
  - All-(sum)-reduce



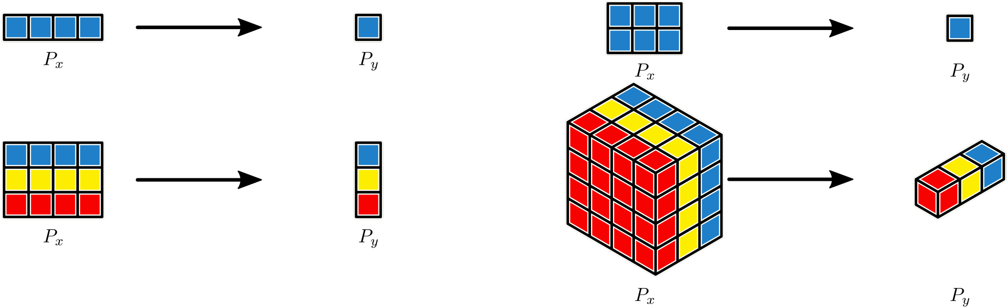
*Broadcast*

- The broadcast primitive holds for more than just standard MPI-style broadcast
- We can express NumPy-style broadcast semantics across tensor dimensions



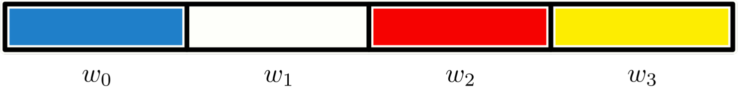
*Sum-reduce*

- The sum-reduce primitive holds for more than just standard MPI-style reductions
- We can use the reverse of NumPy-style broadcast semantics across tensor dimensions



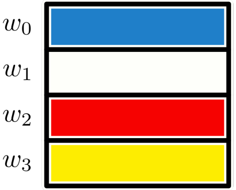
*Transpose*

- All-to-all works slightly differently
- We had to adapt an interpretation of all-to-all to high-order tensors



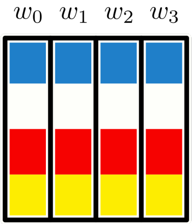
*Transpose*

- All-to-all works slightly differently
- We had to adapt an interpretation of all-to-all to high-order tensors



*Transpose*

- All-to-all works slightly differently
- We had to adapt an interpretation of all-to-all to high-order tensors

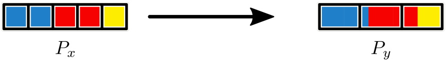


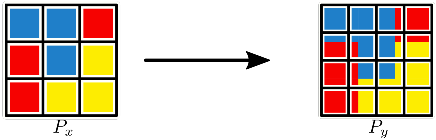
*Transpose*

- All-to-all works slightly differently
- We had to adapt an interpretation of all-to-all to high-order tensors

*Transpose*

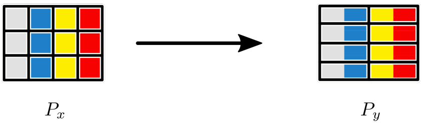
- All-to-all works slightly differently
- We had to adapt an interpretation of all-to-all to high-order tensors

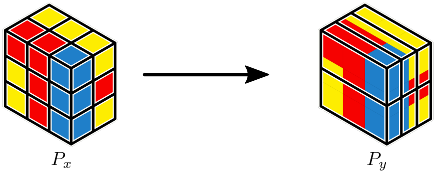


*Transpose*

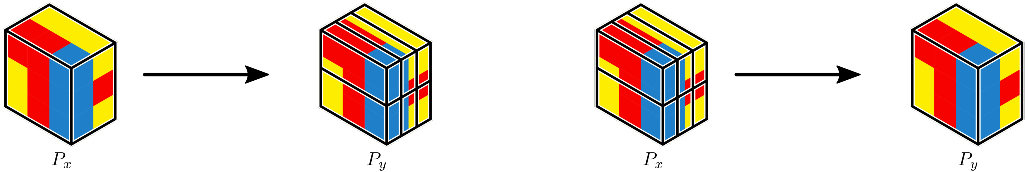
- All-to-all works slightly differently
- We had to adapt an interpretation of all-to-all to high-order tensors



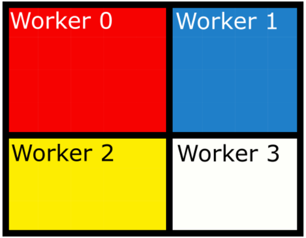



*Transpose*

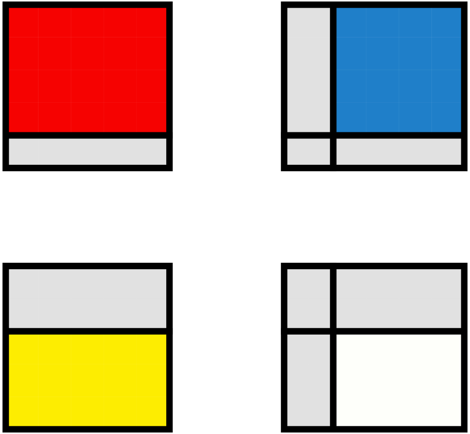
- All-to-all works slightly differently
- We had to adapt an interpretation of all-to-all to high-order tensors
- This is how we get tensor scatters and gathers, too!



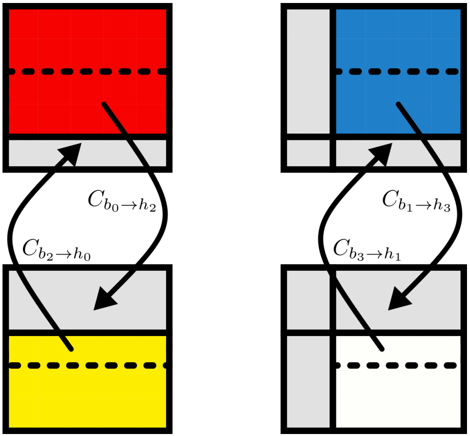
*Halo Exchange*



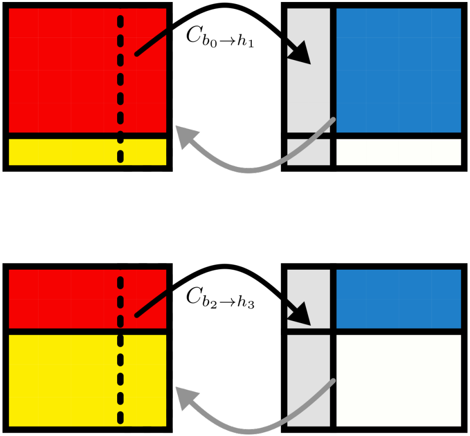
*Halo Exchange*



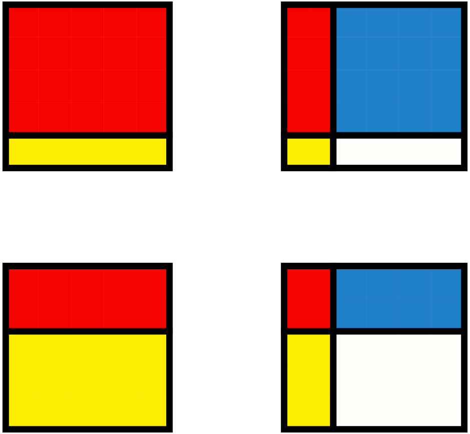
*Halo Exchange*



*Halo Exchange*



*Halo Exchange*

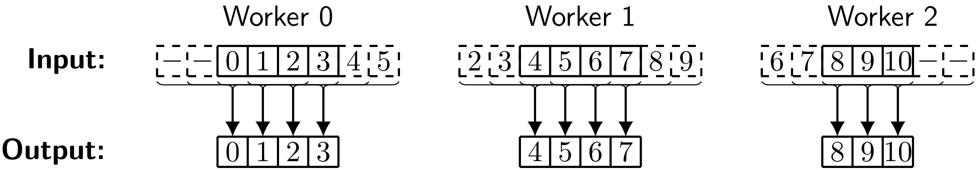


*Halo Exchange*

(w/ Thomas Grady)

- Halo exchange is not a standard parallel primitive
- We generally impose that the output tensor is computationally load balanced
  - In general, even if input is load balanced, output is not guaranteed to be load balanced
- Required for sliding-window kernels on distributed tensors
  - These kernels do not have regular size, as they would in, e.g., standard finite-differences
- Also required for, e.g., interpolation / upsampling

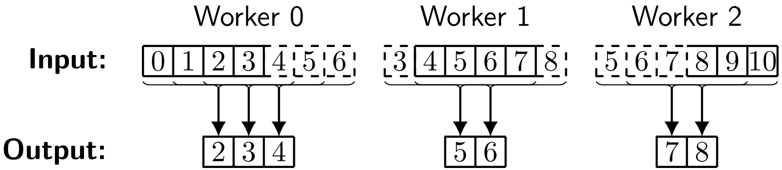
*Halo Exchange*




This situation yields the "normal", uniform halo sizes.

- Centered convolution kernel, size $k = 5$
- 1D input tensor, size $n = 11$
- 1D partition, size $P = 3$
- Zero-padding of width 2, implicitly on input boundaries

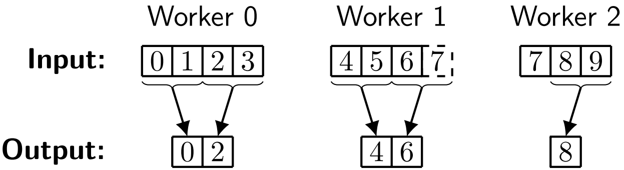
*Halo Exchange*




This situation yields unbalanced halo sizes.

- Centered convolution kernel, size $k = 5$
- 1D input tensor, size $n = 11$
- 1D partition, size $P = 3$
- No implicit zero-padding on input boundaries

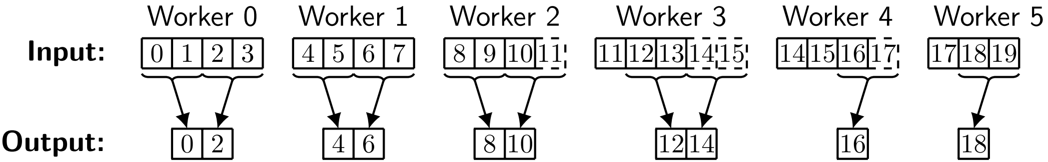
*Halo Exchange*




This situation yields “simple” unbalanced halo sizes.

- Right-looking pooling kernel, size $k = 2$, stride $s = 2$
- 1D input tensor, size $n = 10$
- 1D partition, size $P = 3$
- No implicit zero-padding or dilation

*Halo Exchange*




This situation yields “complicated” unbalanced halo sizes.

- Right-looking pooling kernel, size $k = 2$, stride $s = 2$
- 1D input tensor, size $n = 20$
- 1D partition, size $P = 6$
- No implicit zero-padding or dilation

*Halo Exchange*

- Halo exchange is not a standard parallel primitive
- We construct a halo exchange from a series of clear and copy operations,

$$H = K_{\mathsf{T}} C_{\mathsf{U}} C_{\mathsf{E}} C_{\mathsf{P}} K_{\mathsf{S}},$$

  - $K_{\mathsf{S}}$ the setup operator to clear exchange buffers
  - $C_{\mathsf{P}}$ the pack operator to copy bulk region to send buffer
  - $C_{\mathsf{E}}$ the exchange operator to copy from workers' send buffers to neighboring workers' receive buffers
  - $C_{\mathsf{U}}$ the unpack operator to copy from receive buffer to halo region
  - $K_{\mathsf{T}}$ the teardown operator to clear exchange buffers

*Halo Exchange*

- $D$-dimensional partitioned tensors require one halo exchange for each dimension

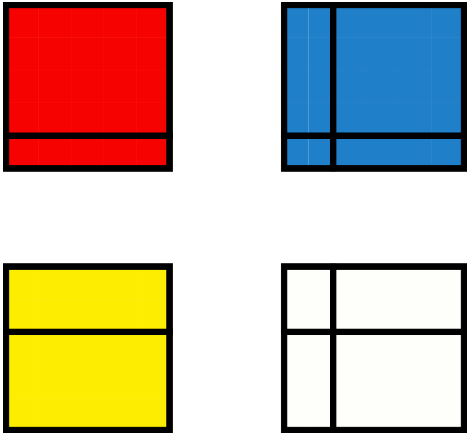
$$H = K_{\mathbf{T}_{d-1}} C_{\mathbf{U}_{d-1}} C_{\mathbf{E}_{d-1}} C_{\mathbf{P}_{d-1}} K_{\mathbf{S}_{d-1}} \dots K_{\mathbf{T}_1} C_{\mathbf{U}_1} C_{\mathbf{E}_1} C_{\mathbf{P}_1} K_{\mathbf{S}_1} K_{\mathbf{T}_0} C_{\mathbf{U}_0} C_{\mathbf{E}_0} C_{\mathbf{P}_0} K_{\mathbf{S}_0}$$

- This handles all corner cases
- The adjoint thus requires *summation* into the bulk region from the halo regions

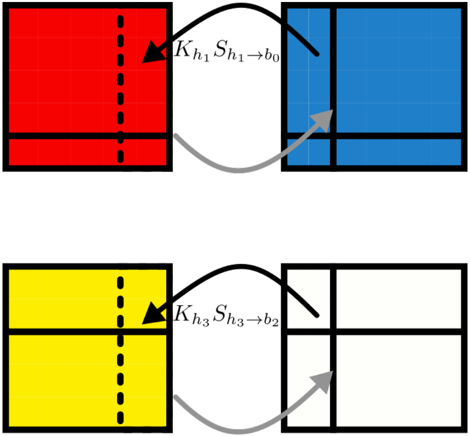
$$H^* = K^*_{\mathbf{S}_0} C^*_{\mathbf{P}_0} C^*_{\mathbf{E}_0} C^*_{\mathbf{U}_0} K^*_{\mathbf{T}_0} K^*_{\mathbf{S}_1} C^*_{\mathbf{P}_1} C^*_{\mathbf{E}_1} C^*_{\mathbf{U}_1} K^*_{\mathbf{T}_1} \dots K^*_{\mathbf{S}_{d-1}} C^*_{\mathbf{P}_{d-1}} C^*_{\mathbf{E}_{d-1}} C^*_{\mathbf{U}_{d-1}} K^*_{\mathbf{T}_{d-1}}.$$

- We see this most easily from the linear algebraic definitions: each of the adjoint copies is an add-clear

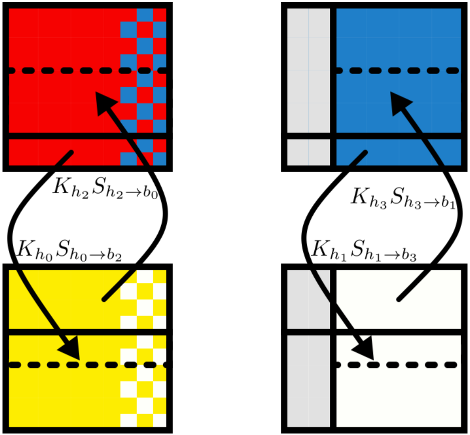
*Adjoint Halo Exchange*



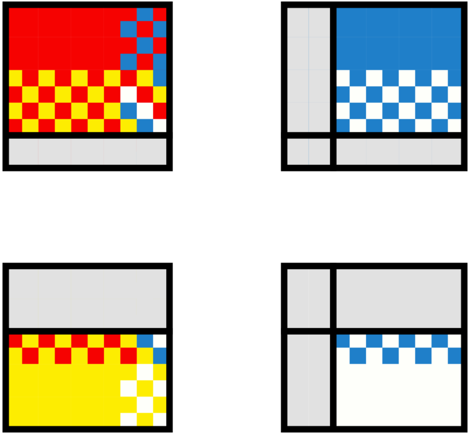
*Adjoint Halo Exchange*



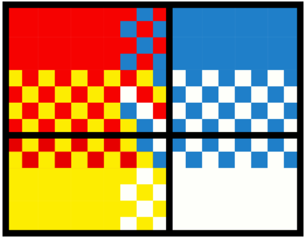
*Adjoint Halo Exchange*



*Adjoint Halo Exchange*



*Adjoint Halo Exchange*



- We can compose distributed DNN Layers using linear operator forms of parallel primitives
- Right now we support the basic building blocks:
  - Distributed Convolutional layers
  - Distributed Pooling layers
  - Distributed Linear/Affine layers
  - Distributed Batchnorm layers
  - Distributed Upsampling layers
- Support for other functions will be added as needed
- Element-wise layers (e.g., ReLU) do not require data movement

A simple distributed convolutional layer:

$$y_i = \texttt{SequentialConv}(\texttt{HaloExchange}(x_i); \texttt{Broadcast}(w), \texttt{Broadcast}(b))$$

A simple distributed convolutional layer:

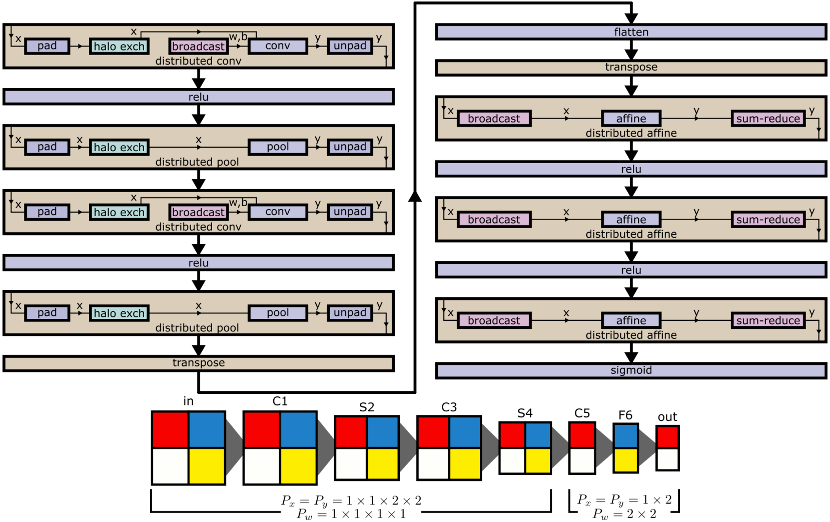
$$y_i = \texttt{SequentialConv}(\texttt{HaloExchange}(x_i); \texttt{Broadcast}(w), \texttt{Broadcast}(b))$$

**Forward Convolution Algorithm**

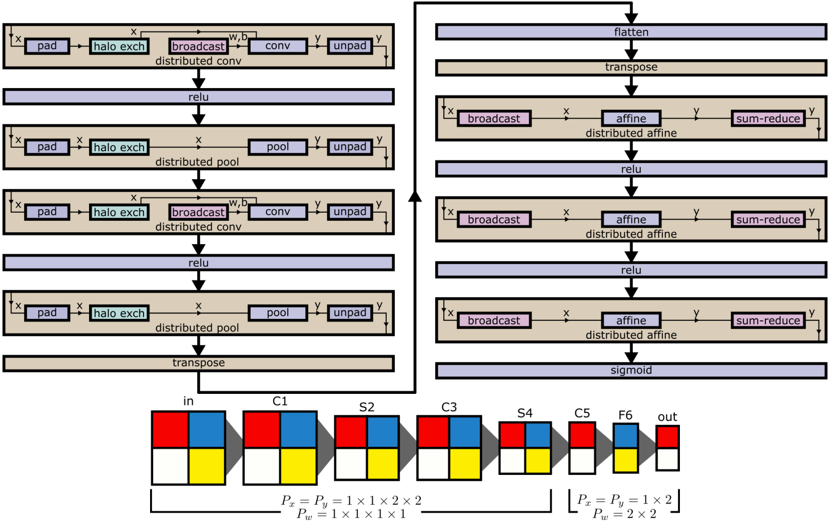
1: Input: $\mathbf{x}$
2: $\hat{\mathbf{x}} \leftarrow H\mathbf{x}$
3: $\hat{\mathbf{w}} \leftarrow B_{\{P_r\}\to\{P_x\}}\mathbf{w}$
4: $\hat{\mathbf{b}} \leftarrow B_{\{P_r\}\to\{P_x\}}\mathbf{b}$
5: $\mathbf{y} \leftarrow \texttt{Conv}(\hat{\mathbf{w}}, \hat{\mathbf{b}}; \hat{\mathbf{x}})$
6: Output: $\mathbf{y}$

**Adjoint Convolution Algorithm**

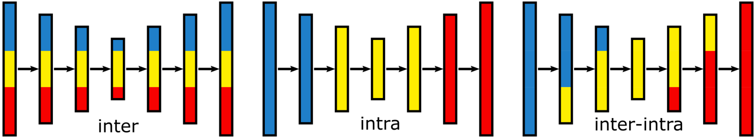
1: Input: $\delta\mathbf{y}$
2: $\delta\hat{\mathbf{w}}, \delta\hat{\mathbf{b}}, \delta\hat{\mathbf{x}} \leftarrow [\delta\texttt{Conv}]^*(\delta\mathbf{y})$
3: $\delta\mathbf{b} \leftarrow R_{\{P_x\}\to\{P_r\}}\delta\hat{\mathbf{b}}$
4: $\delta\mathbf{w} \leftarrow R_{\{P_x\}\to\{P_r\}}\delta\hat{\mathbf{w}}$
5: $\delta\mathbf{x} \leftarrow H^*\delta\hat{\mathbf{x}}$
6: Output: $\delta\mathbf{x}$



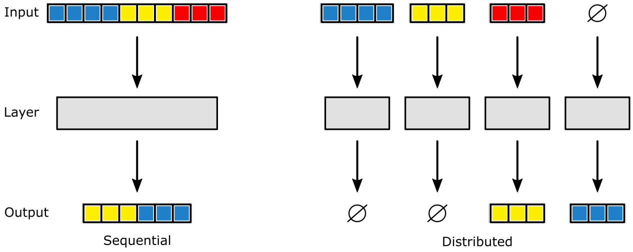
x
pad
halo exch
broadcast
w,b
conv
y
unpad
y
distributed conv
relu
x
pad
x
halo exch
x
pool
y
unpad
y
distributed pool
x
pad
halo exch
broadcast
w,b
conv
y
unpad
y
distributed conv
relu
x
pad
x
halo exch
x
pool
y
unpad
y
distributed pool
transpose
flatten
transpose
x
broadcast
x
affine
y
sum-reduce
y
distributed affine
relu
x
broadcast
x
affine
y
sum-reduce
y
distributed affine
relu
x
broadcast
x
affine
y
sum-reduce
y
distributed affine
sigmoid
in
C1
S2
C3
S4
C5
F6
out
$P_x = P_y = 1 \times 1 \times 2 \times 2$
$P_w = 1 \times 1 \times 1 \times 1$
$P_x = P_y = 1 \times 2$
$P_w = 2 \times 2$


x
pad
halo exch
broadcast
w,b
conv
y
unpad
y
distributed conv
relu
distributed pool
pool
transpose
flatten
transpose
broadcast
affine
sum-reduce
distributed affine
relu
sigmoid
in
C1
S2
C3
S4
C5
F6
out
$P_x = P_y = 1 \times 1 \times 2 \times 2$
$P_w = 1 \times 1 \times 1 \times 1$
$P_x = P_y = 1 \times 2$
$P_w = 2 \times 2$

- DistDL: Distributed Deep Learning
  - PyTorch + MPI based tool
  - `https://github.com/distdl/distdl`
  - `https://distdl.readthedocs.io/en/latest/`
  - Absolutely a work in progress. . .
- Paper: https://arxiv.org/abs/2006.03108
  - RJH, Thomas Grady



- DistDL: Distributed Deep Learning
  - PyTorch + MPI based tool
  - `https://github.com/distdl/distdl`
  - `https://distdl.readthedocs.io/en/latest/`
  - Absolutely a work in progress...
- Paper: https://arxiv.org/abs/2006.03108
  - RJH, Thomas Grady



# DistDL: Distributed Deep Learning

- DistDL: Distributed Deep Learning
  - PyTorch + MPI based tool
  - `https://github.com/distdl/distdl`
  - `https://distdl.readthedocs.io/en/latest/`
  - Absolutely a work in progress...
- Paper: https://arxiv.org/abs/2006.03108
  - RJH, Thomas Grady

Others involved:

- Daniel Hagialigol (current CMDA student)
- Thomas Grady (recent CMDA & Math graduate)
- Jacob Merizian (recent Math & CS graduate)
- Ananiya Admasu, Mason Beahr, & Sarah Kauffman (CMDA Capstone Team)

# DistDL: Distributed Deep Learning

- DistDL: Distributed Deep Learning
  - PyTorch + MPI based tool
  - `https://github.com/distdl/distdl`
  - `https://distdl.readthedocs.io/en/latest/`
  - Absolutely a work in progress...
- Paper: https://arxiv.org/abs/2006.03108
  - RJH, Thomas Grady

Thank you! Question time!