Project Fiddle
Fast & Efficient Infrastructure for Distributed Deep Learning

Amar Phanishayee

with

Nikhil Devanur, ***Aaron Harlap, Animesh Jain,***
Liang Luo, Deepak Narayanan, Jacob Nelson,
Gennady Pekhimenko, Vivek Seshadri, Jorgen Thelin,
Shivaram Venkataraman, ***Guanhua Wang***

Big Data Systems – Gen AI

- Motivating Scenarios:



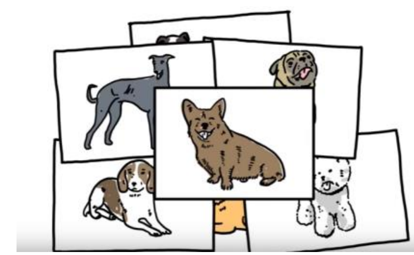
Image Recognition, Classification

Translation & Speech – Text - Speech

- Solutions:

DistBelief

PROJECT ADAM



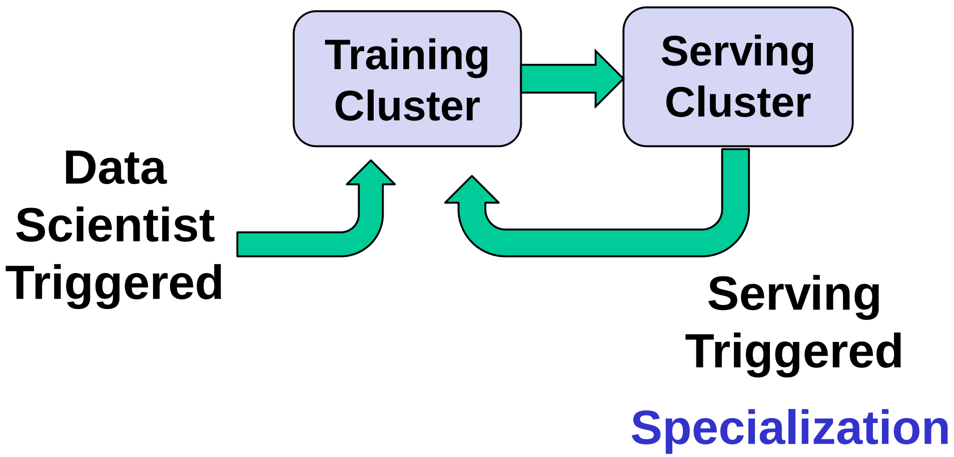
TensorFlow™

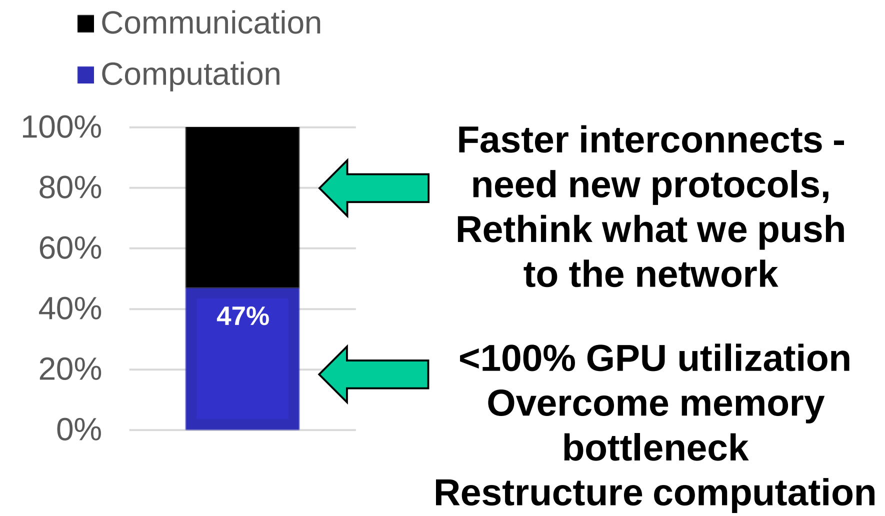
CNTK

Deep Learning for X

- Image recognition, classification
- Segmentation
- Captioning
- Stereo
- Depth estimation
- Video action recognition
- Sentiment Analysis
- Speech-Text, Text-Speech, OCR
- Translation
- …

Training is time and resource intensive





- Long running **training** jobs: Days to multiple weeks
- Train, Validate, Repeat

  Performance and accuracy specs validated only by running system
- Naïvely parallelizing work can be detrimental
- No luxury of limiting solution to one part of the stack

Need for speed

From EE Times – September 27, 2016

"Today the job of training machine learning models is limited by compute, if we had faster processors we'd run bigger models…in practice we train on a reasonable subset of data that can finish in a matter of months. We could use improvements of several orders of magnitude – 100x or greater."

– Greg Diamos, Senior Researcher, SVAIL, Baidu

Slide Credit: Joel Emer (MIT, NVIDIA)

Problem

Systematically speed-up distributed learning tasks while eking out the most from the resources used

Goal: 100x more efficient training

Day long training of model in 15-min coffee break

Approach

Cut through layers of the system stack and optimize all relevant parts

	Single Machine Training	Multi-Machine Training
Memory, Computation	**Gist** Train larger networks on a *single* GPU by reducing memory footprint.	**PipeDream** Efficiently scale training. New way to parallelize DNN computation.
Interconnects	**Blink** Blazingly fast transfers over PCIe + NVLink + IB	**Hub** Fast parameter servers

	Single Machine Training	Multi-Machine Training
Memory, Computation	**Gist** Train larger networks on a *single* GPU by reducing memory footprint. **Up to 2x compression ratio**	**PipeDream** Efficiently scale training. New way to parallelize DNN computation. **3x faster training**
Interconnects	**Blink** Blazingly fast transfers over PCIe + NVLink + IB **Collective transfer primitives 8x faster than NCCL**	**Hub** Fast parameter servers **Up to 3x speedup**

	Single Machine Training	Multi-Machine Training
Memory, Computation	**Gist** Train larger networks on a *single* GPU by reducing memory footprint. **Up to 2x compression ratio**	
Interconnects		

Background

Training Examples
<x1, y1>
<x2, y2>
…

Learning Algorithm

Hypothesis

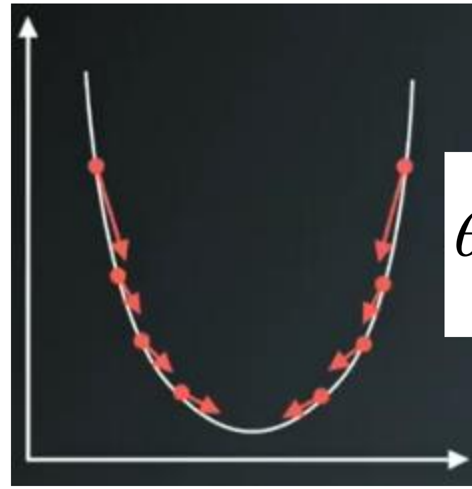
$$h(\theta, x) \rightarrow y$$

$g(\theta^T x)$ or $g(t)$

Learn θ

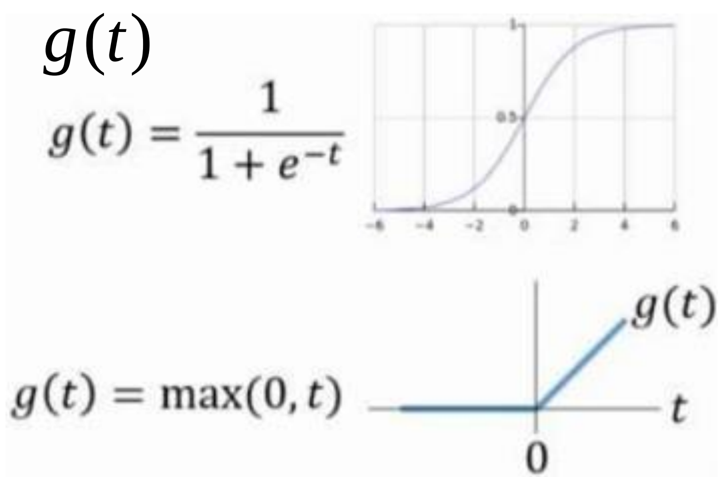
$$J(\theta) = (1/m) \times \sum_{i=1}^{m} c(h(\theta)_i, y_i)$$

$J(\theta)$



θ_i

$$\theta_i = \theta_i - \mu\left(\frac{\partial J(\theta)}{\partial \theta_i}\right)$$



Background

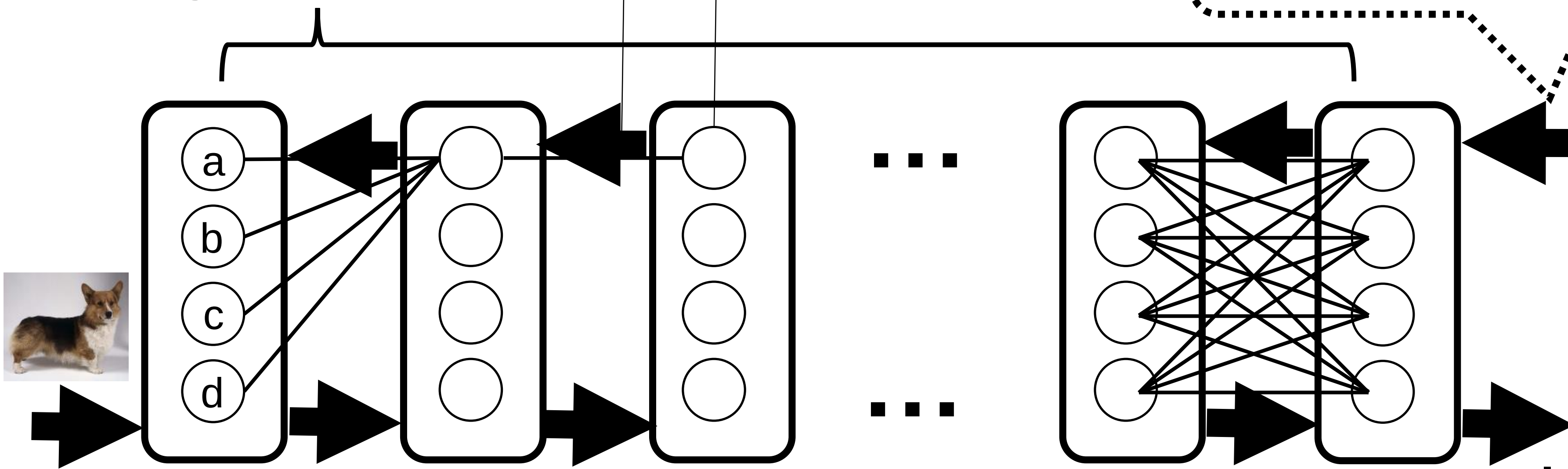
Structure of DNN computation

$$\boldsymbol{g}(\,\boldsymbol{a} * \propto + \boldsymbol{b} * \beta + \boldsymbol{c} * \gamma + \boldsymbol{d} * \delta\,) \textbf{ or } g(\theta^T x)$$

Dog

Layers

Neuron

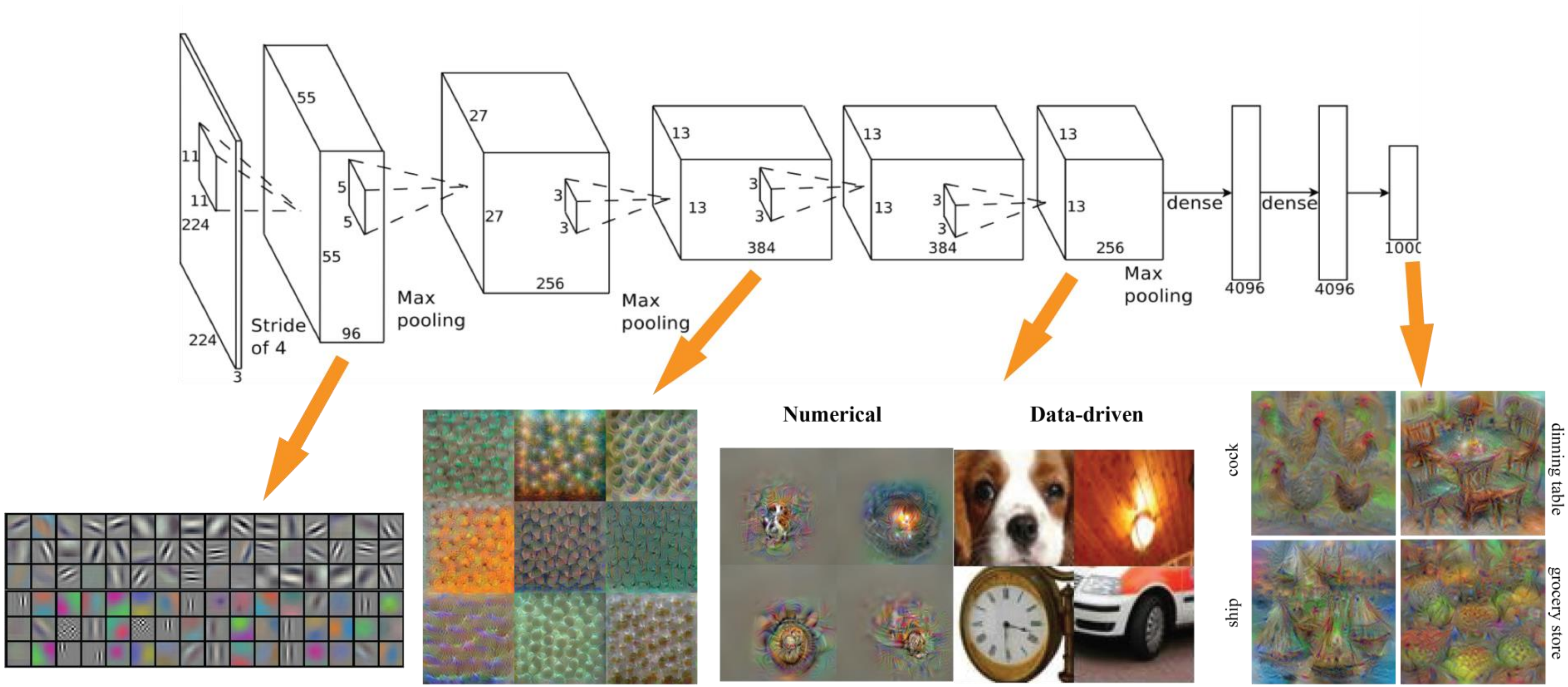


Activations (feature maps)

Cat?

Batch. Run for all input items.
Repeat over many "epochs"

Example – AlexNet
(Image Classification)



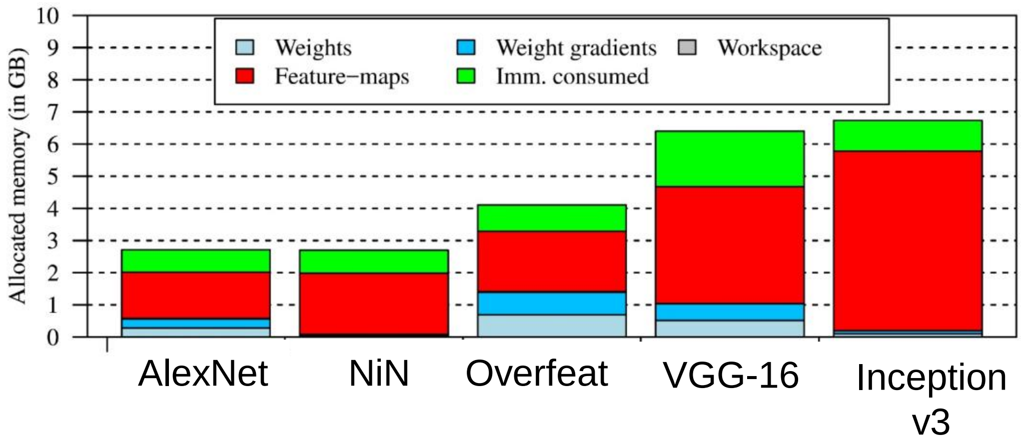
Conv 1: Edge+Blob | Conv 3: Texture | Conv 5: Object Parts | Fc8: Object Classes

Slide Credit: http://vision03.csail.mit.edu/cnn_art/

Tasks, datasets, challenges

- 1989: MNIST
- 2001: Caltech 101
- 2007 – 2012: PASCAL Visual Object Classes
  - 20 classes
- 2014 - … : Imagenet 1K
  - Deng et al.
  - Classification and Detection

Profile of memory consumption



Feature Maps are a major consumer of GPU memory

CNTK Profiling

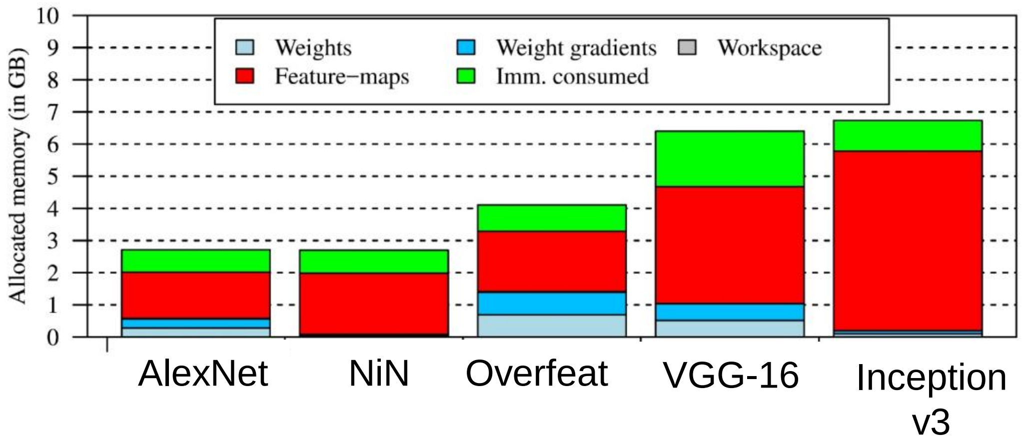
Larger minibatch size ➔ potential crash

Input Feature Map (X) → **DNN Layer** → Output Feature Map (Y)

Input Gradient (dx) ← **DNN Layer** ← Output Gradient (dy)

dx = f(X, Y, dy)

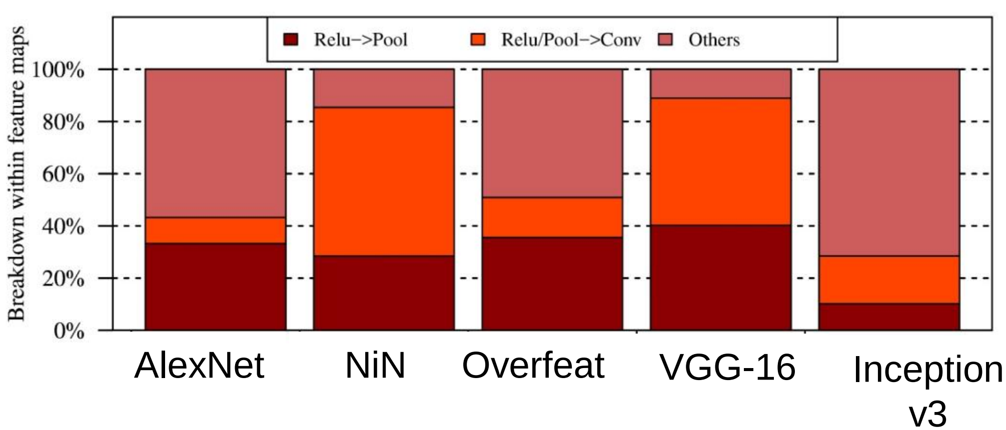
Profile of memory consumption



Feature Maps are a major consumer of GPU memory

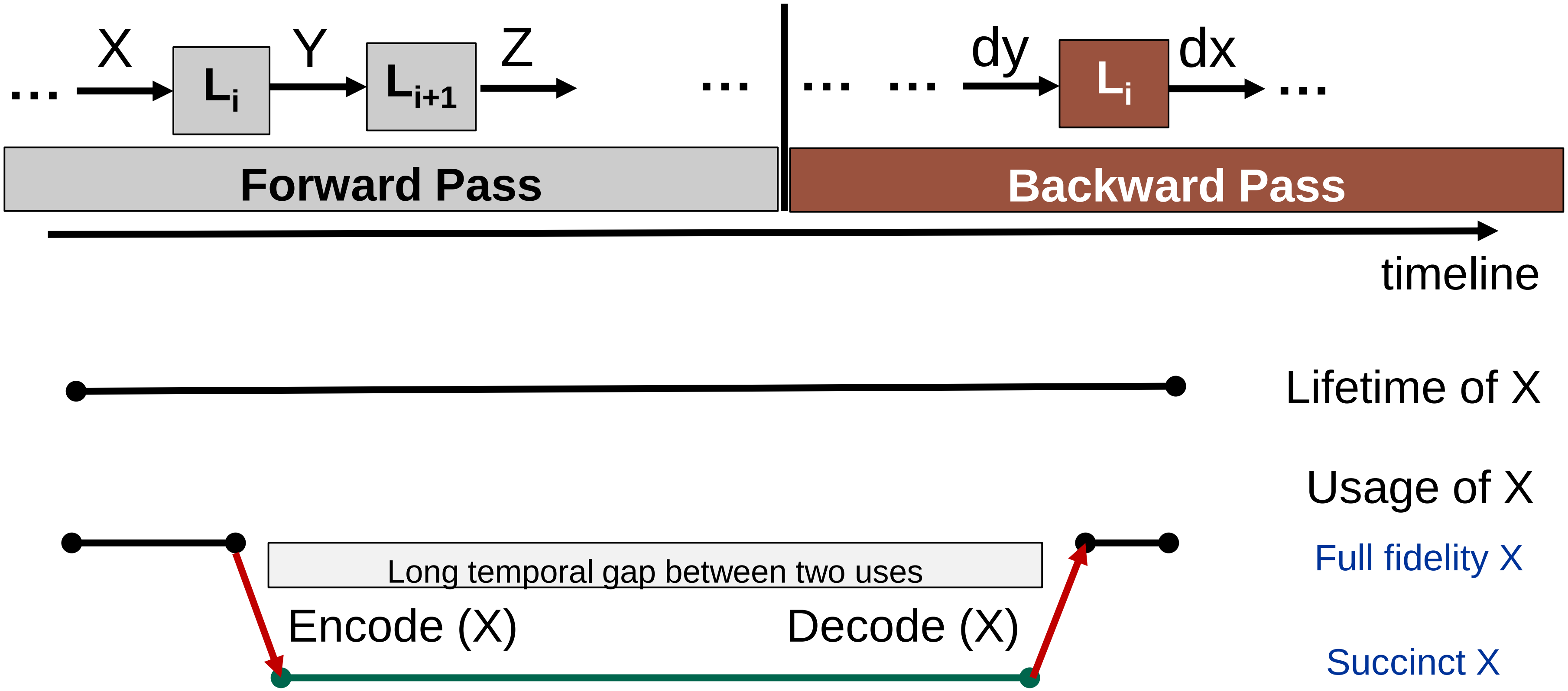
CNTK Profiling

Larger minibatch size ➔ potential crash



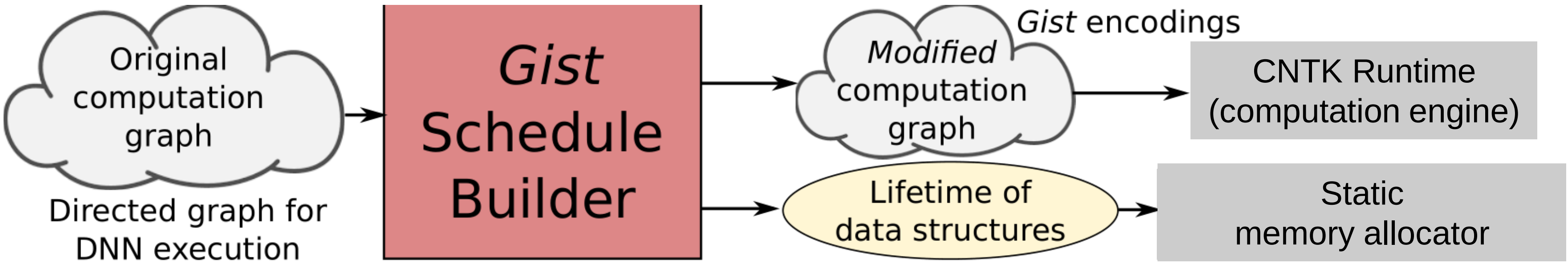
Heavy hitters: Relu->Pool, Relu/Pool-> Conv

Idea: Encode data structures when unused, Decode on use



Statically construct usage schedule for all data structures

Gist Architecture

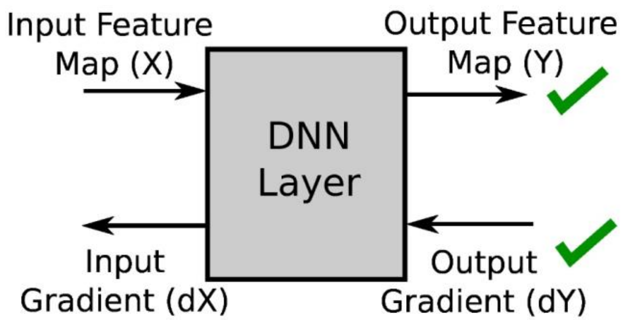


- ***Statically*** construct usage schedule for all data structures

- Layer-specific encodings
  - Lossless (Binarize, Sparse Storage/Dense Compute)
  - Lossy (Delayed Precision Reduction)

Gist Lossless Encodings

For Relu->Pool



dX = f(Y, dY)
dx = y > 0 ? dy : 0;

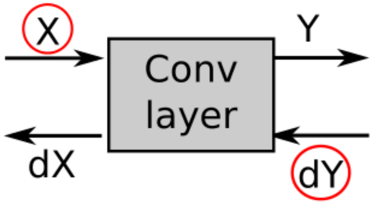
Relu Backward Propagation

<u>Binarize</u>

1 bit representation of 32bit values in Y

32x reduction

For Relu/Pool->Conv

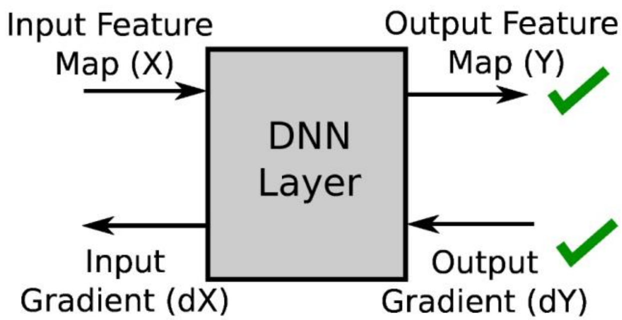


dX = f(**X**, **dY**)

Naively applying Binarize for Relu/Pool followed by Conv can increase memory consumption!

Gist Lossless Encodings

For Relu->Pool



$$dX = f(Y, dY)$$

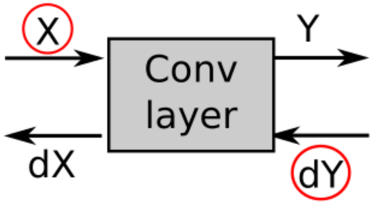
$$dx = y > 0 \; ? \; dy : 0;$$

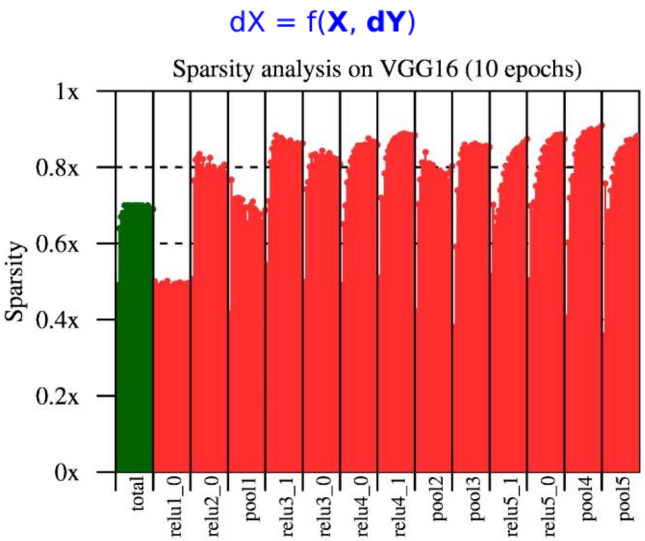
Relu Backward Propagation

Binarize

1 bit representation of 32bit values in Y

For Relu/Pool->Conv



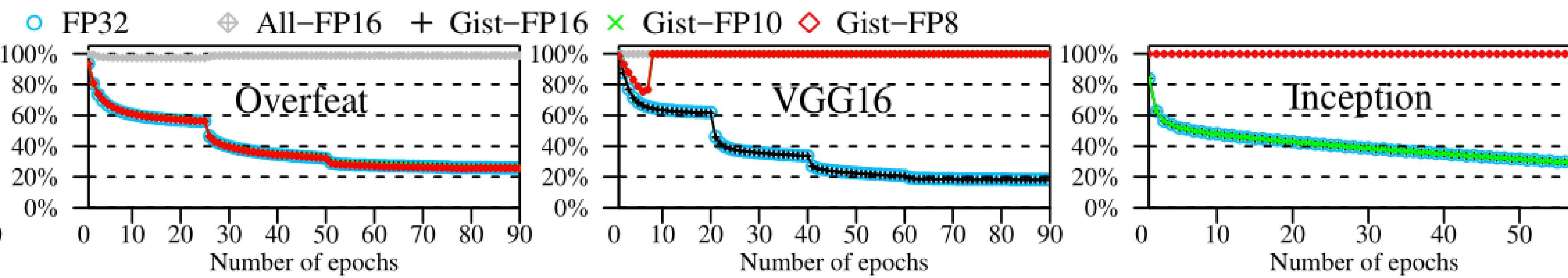


Sparse Storage , Dense Compute

(sparse compute is computationally expensive)

Gist Lossy Encodings

- Used for all other feature maps
- Training with reduced precision (8/10/16 bits) – done only when encoding/stashing values
- Forward pass uses full fidelity values

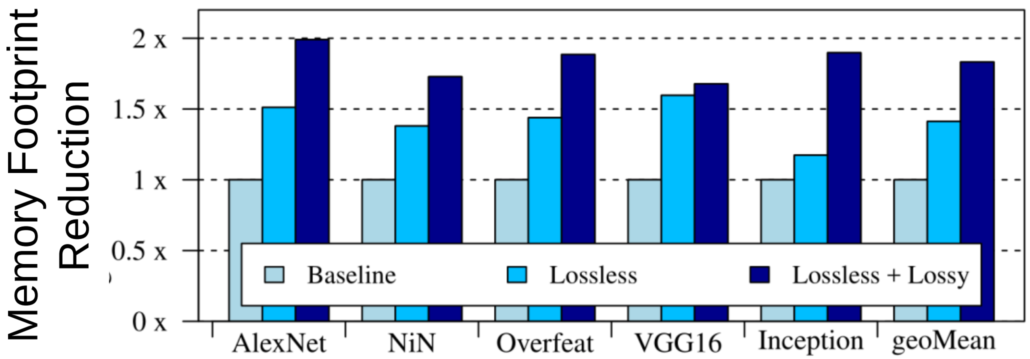


8 bits enough!

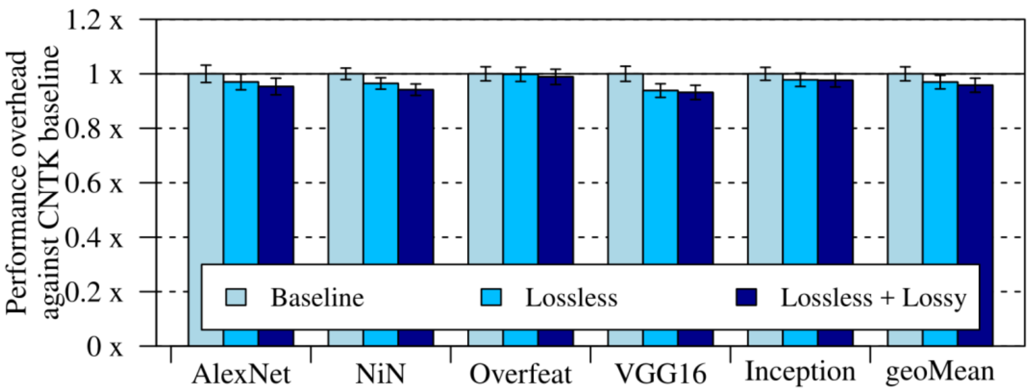
8/10 bits not enough.
16 bits works out.

10 bits enough!

Gist – Putting it all together



Up to 2x compression ratio



Minimal runtime overhead (1 - 7%) for same batch size

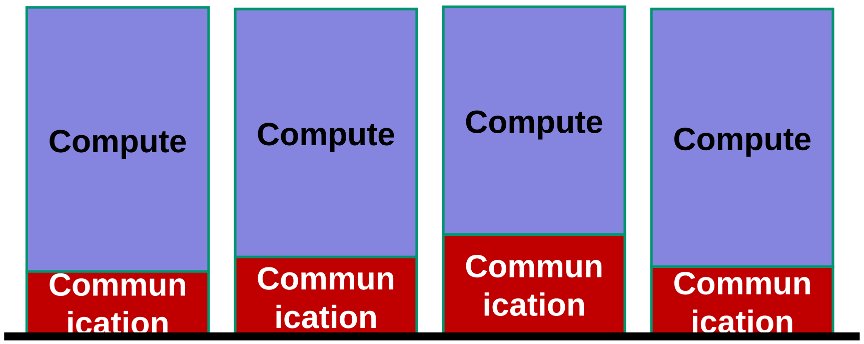
With just lossless, we go faster at times (memory bandwidth bound)

	Single Machine Training	Multi-Machine Training
Memory, Computation	**Fiddle Gist** Train larger networks on a *single* GPU by reducing memory footprint. **Up to 2x compression ratio**	
Interconnects		

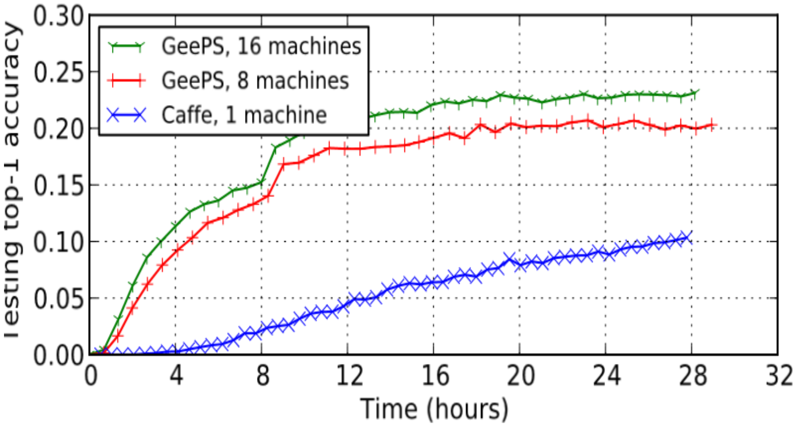
Distributed Deep Learning

- Data Parallelism
  - Replicas run in parallel on different machines
  - Occasionally exchange what they have learnt
  - Naïve setup can hurt convergence, accuracy
  - **Total time = (Time per Epoch) * (#Epochs for given accuracy)**

Hardware Efficiency



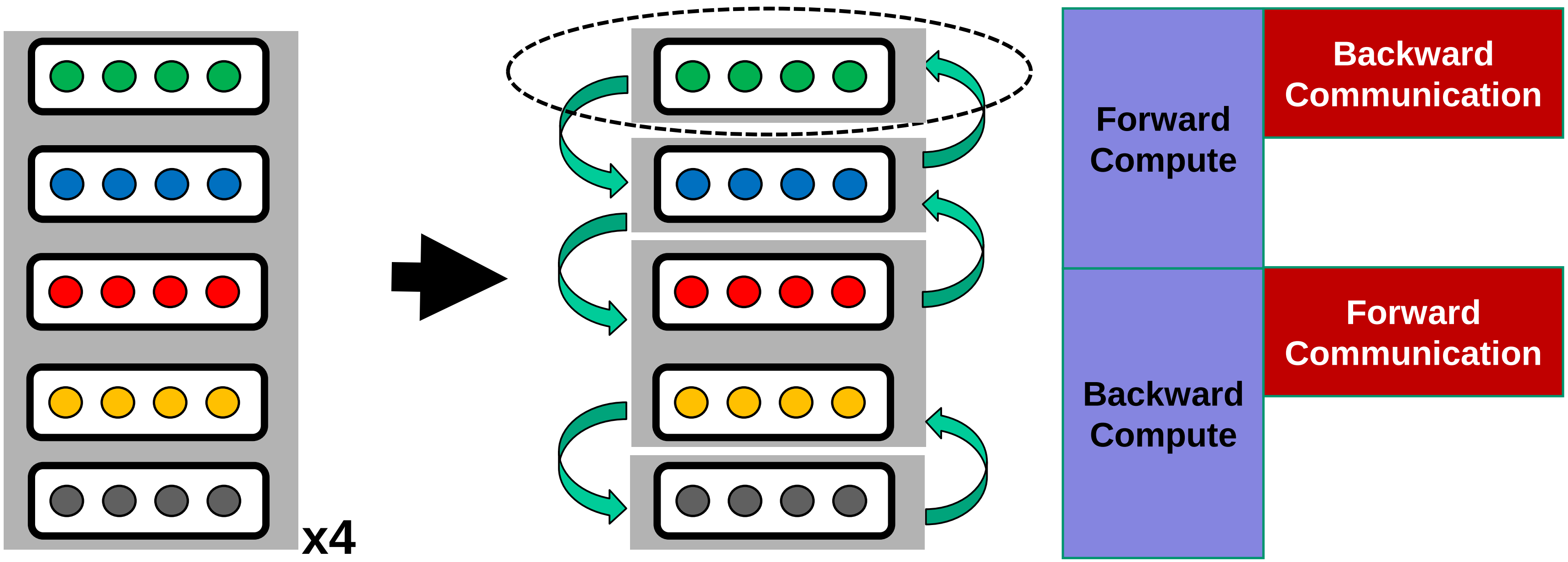
Statistical Efficiency



Manually tuned for performance of individual jobs

Pipeline Parallelism in Fiddle

- Idea: Pipeline computation across machines
  - For training tasks, optimize for throughput
  - Each machine runs a subset of layers
    - Compute one thing, data flows to compute task
    - Better FLOPs due to cache locality
  - Fits CPU, GPU, hybrid, heterogeneous clusters



Challenges

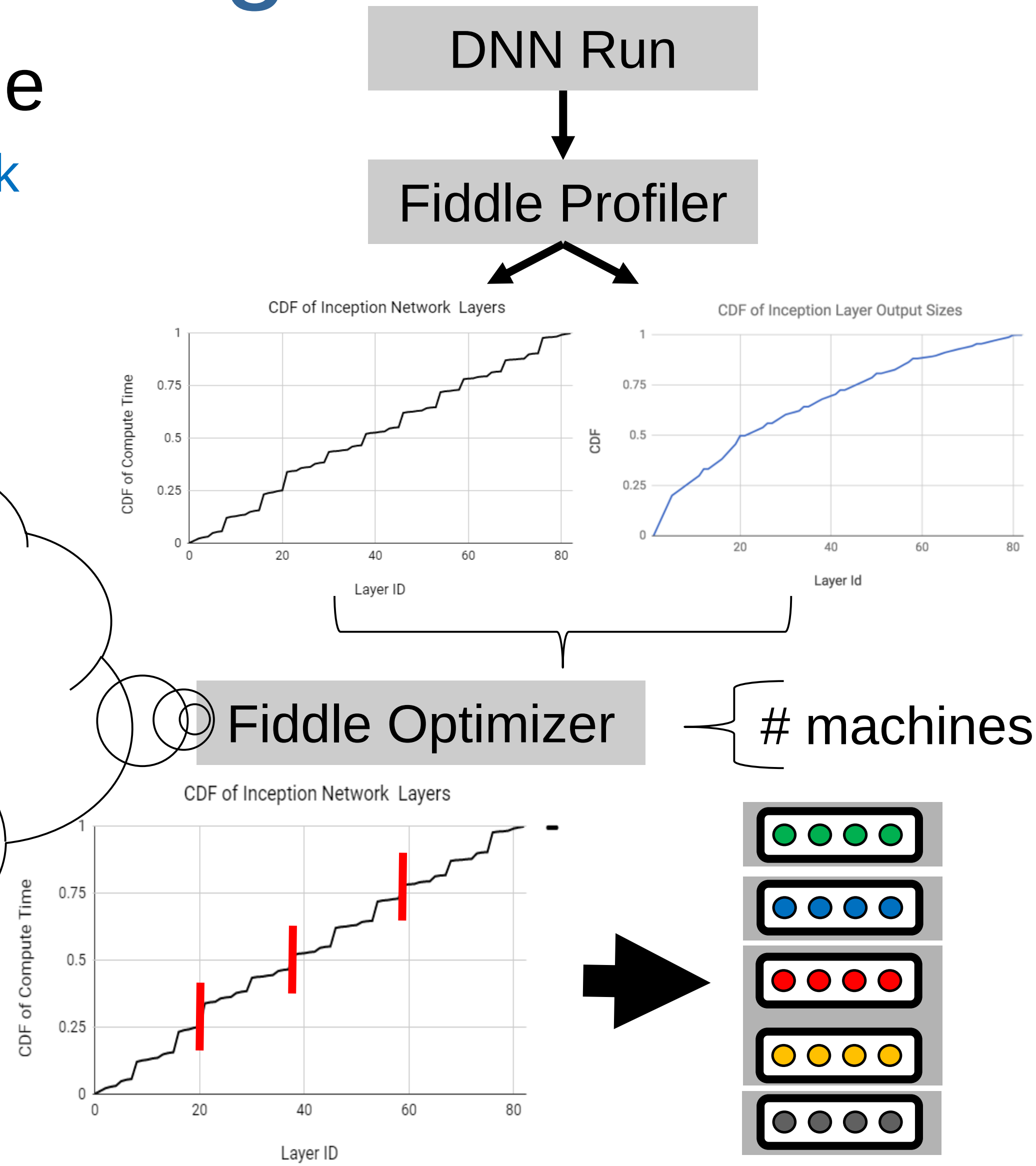
- No bubbles in pipeline
  - How should we divvy work across machines?

$$r_i = runtime(f_i + b_i)$$

$$t_i = \text{transfer-time(i, i+1)}$$

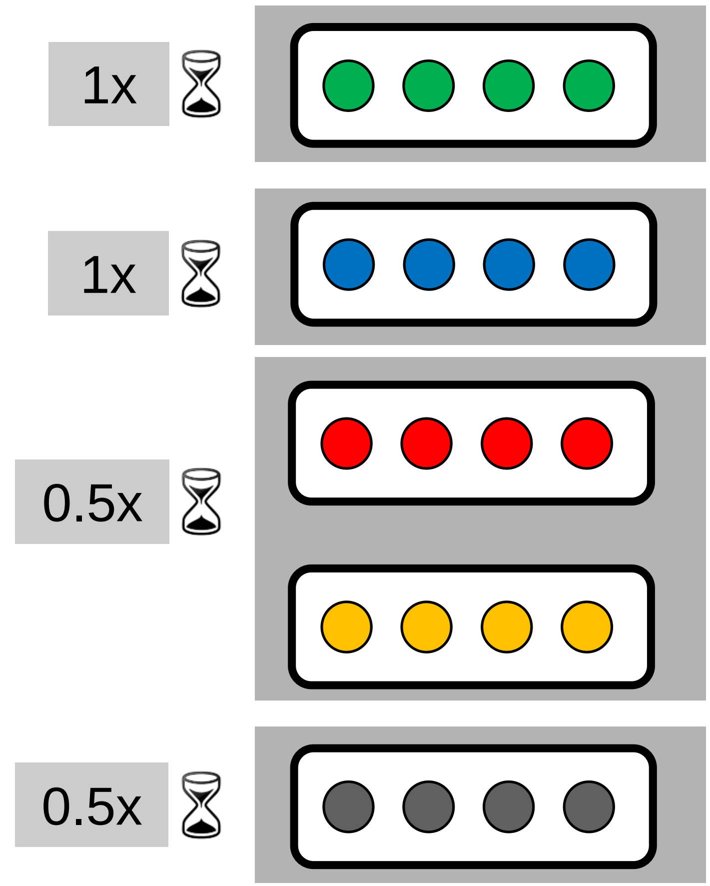
$$l_{ij} = max\,\{\sum_{k=i}^{j} r_k, t_j\}$$

$$\min_{P} \max_{[i,j]\in P} l_{[i,j]}$$



Challenges

- No bubbles in pipeline
  - How should we divvy work across machines?

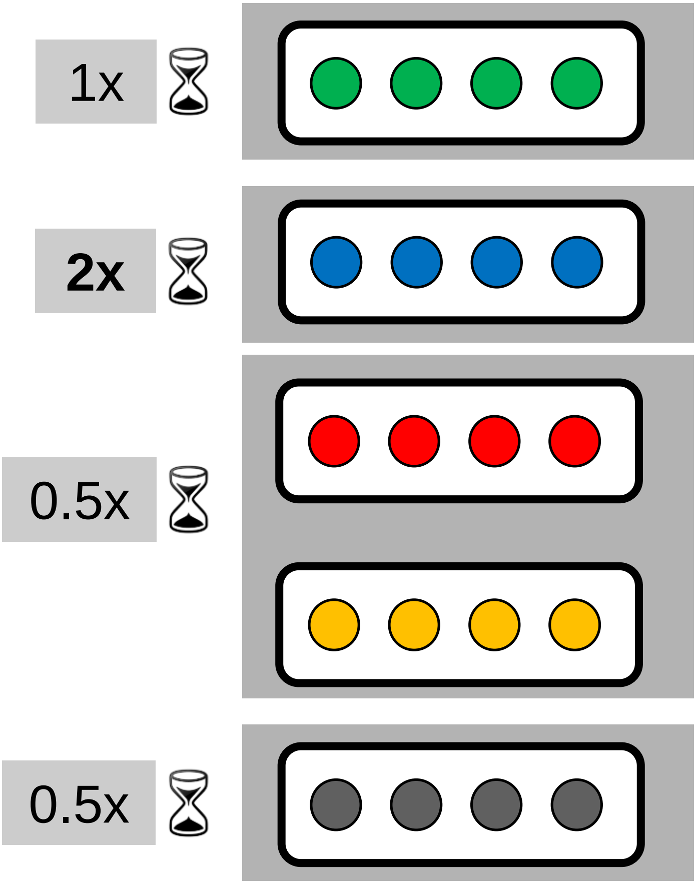


Challenges

- No bubbles in pipeline
  - How should we divvy work across machines?

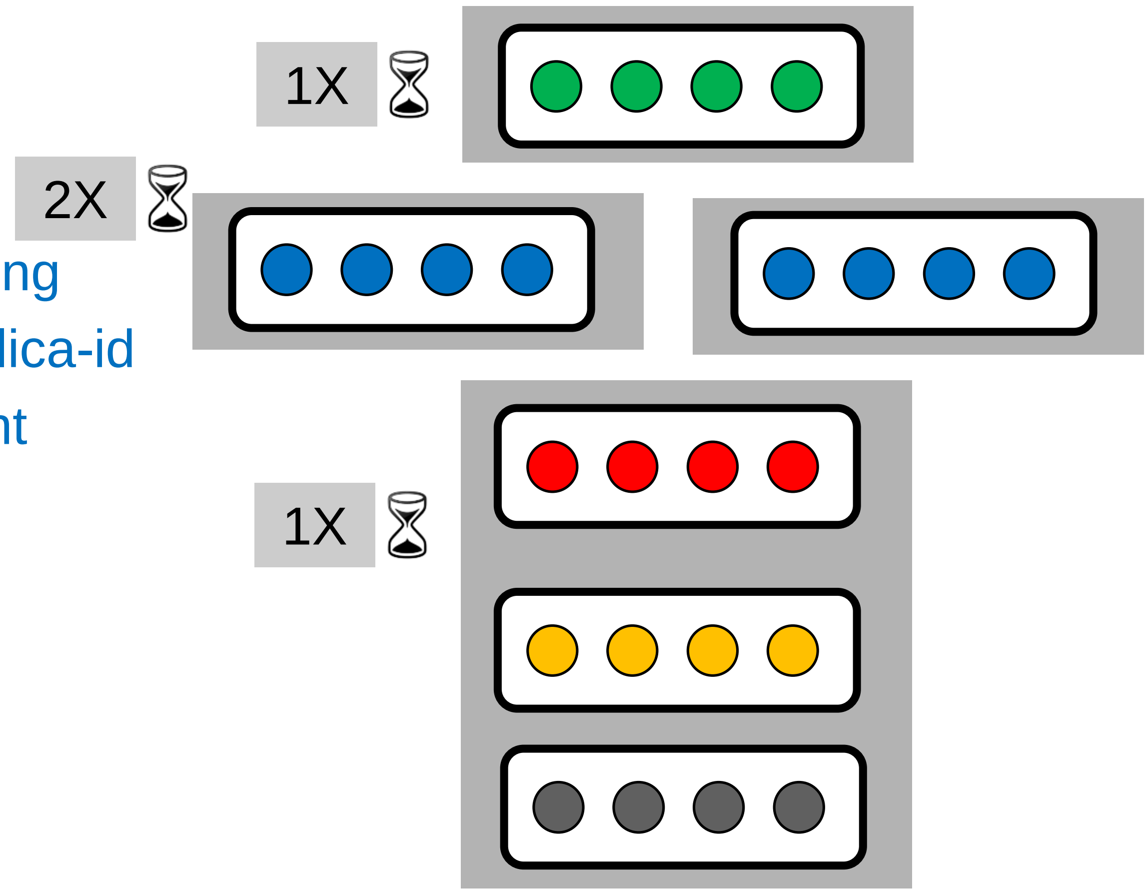
Running at speed of the slowest layer

- Idle Cycles



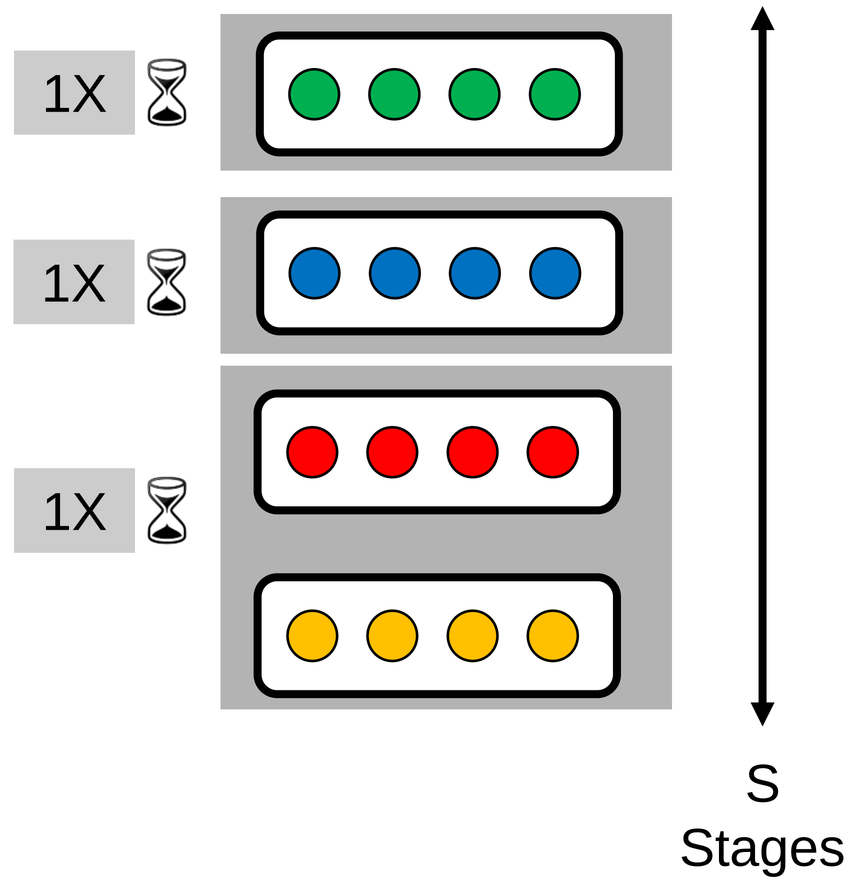
Challenges

- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages (if needed)
    - Static load balancing mini-batch-id % stage-replica-id
    - F/B paths consistent



Challenges

- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages (if needed)
    - Static load balancing mini-batch-id % stage-replica-id
    - F/B paths consistent
  - How many items to admit?
    - Wild, S, less?

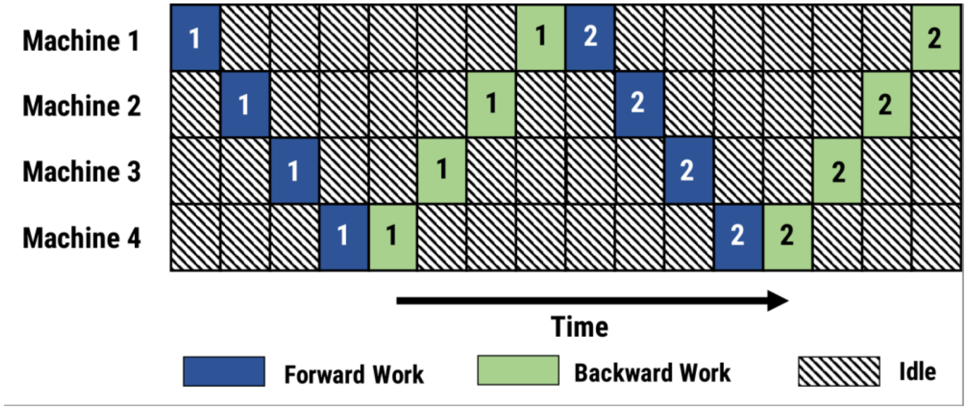


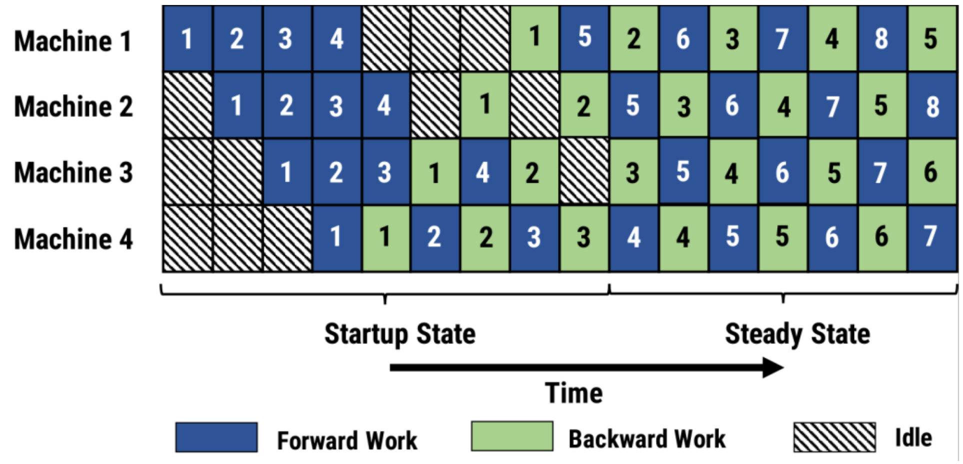
Challenges

- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages

    (if needed)
    – Static load balancing

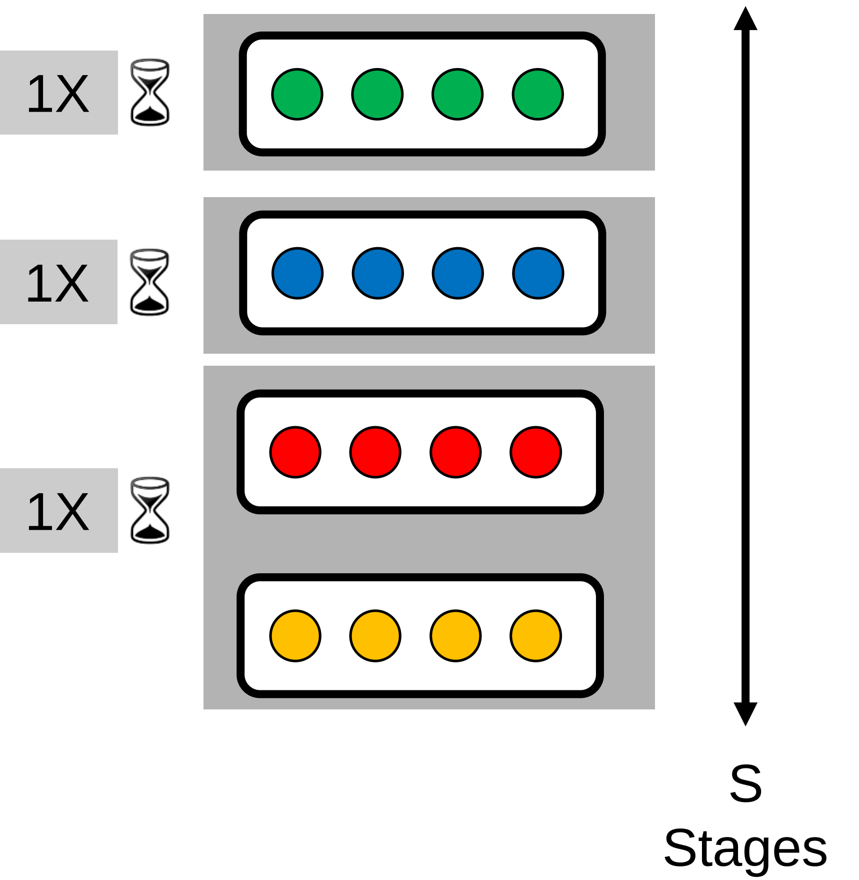
  mini-batch-id % stage-replica-id
    – F/B paths consistent
  - How many items to admit?
    – Wild, S, less?





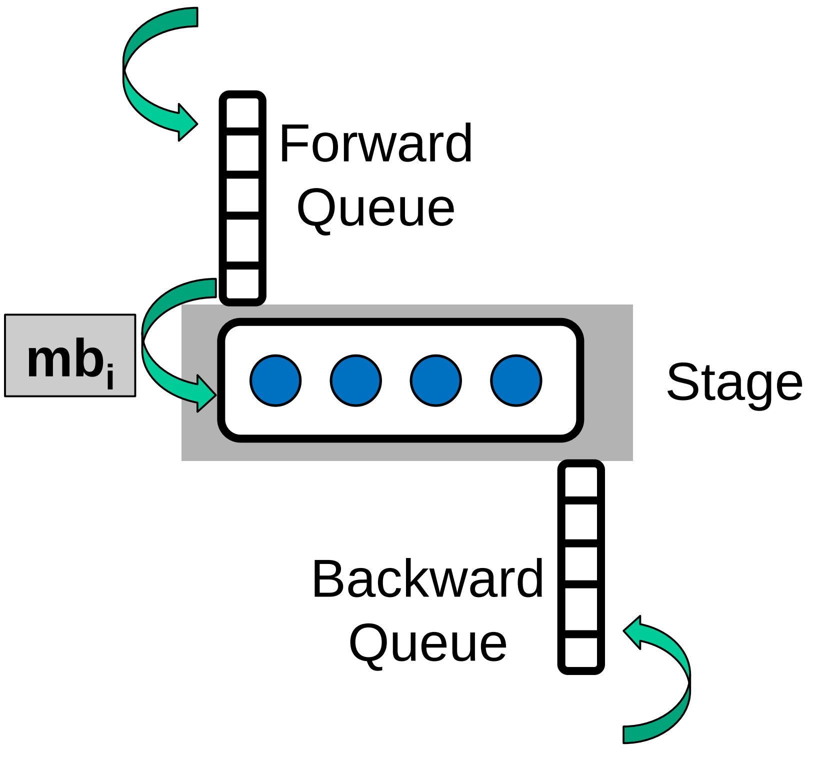
Challenges

- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages
    (if needed)
    - Static load balancing
    mini-batch-id % stage-replica-id
    - F/B paths consistent
  - How many items to admit?
    - Wild, S, less?
- In **steady** state:
Each new item learns from a previous (new) mini-batch



Challenges

- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages
    (if needed)
    - Static load balancing
    mini-batch-id % stage-replica-id
    - F/B paths consistent
  - How many items to admit?
    - Wild, S, less?
- In **steady** state:
  Each new item learns from a previous (new) mini-batch
  - Stage alternates between F/B



Challenges

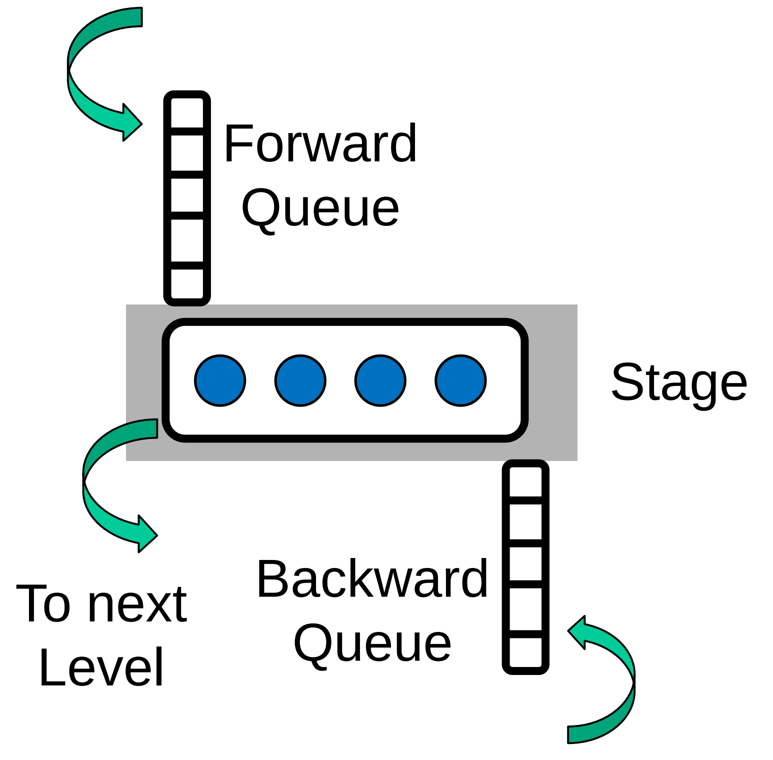
- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages

    (if needed)
    - Static load balancing

  mini-batch-id % stage-replica-id
    - F/B paths consistent
  - How many items to admit?
    - Wild, S, less?
- In **steady** state:

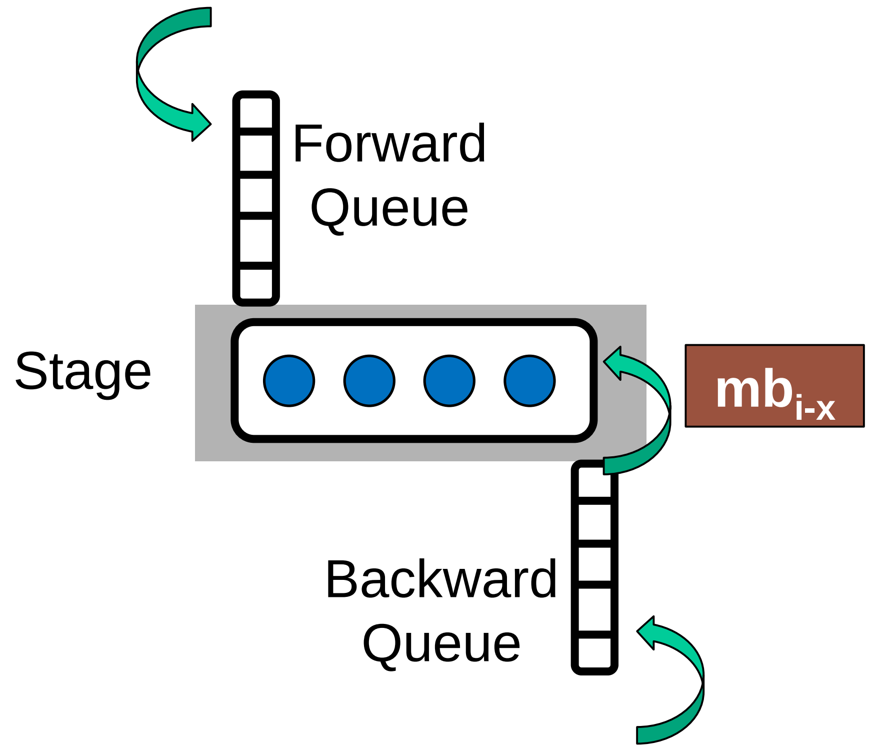
Each new item learns from a previous (new) mini-batch

  - Stage alternates between F/B



Challenges

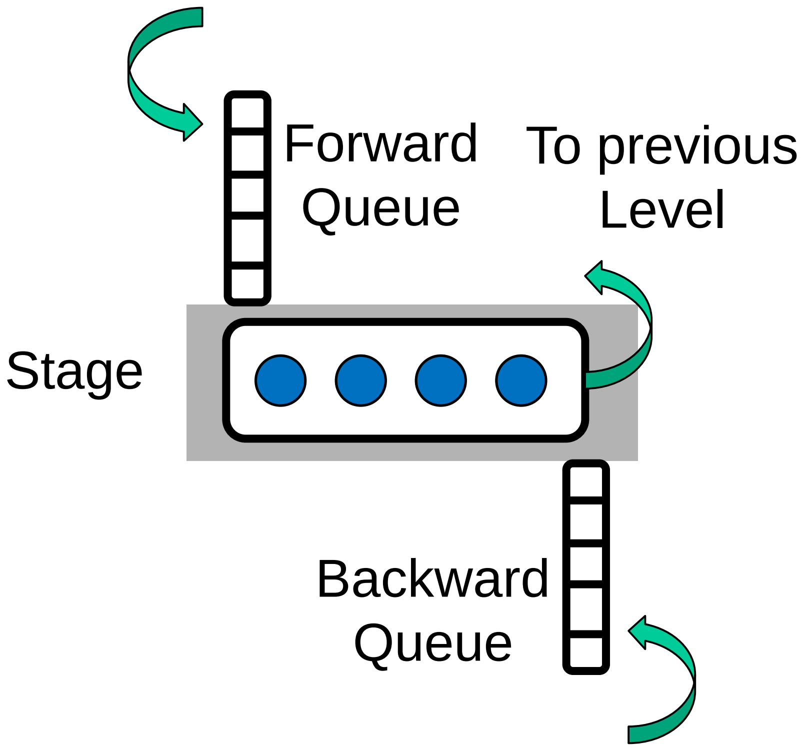
- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages (if needed)
    - Static load balancing mini-batch-id % stage-replica-id
    - F/B paths consistent
  - How many items to admit?
    - Wild, S, less?
- In **steady** state: Each new item learns from a previous (new) mini-batch
  - Stage alternates between F/B



Challenges

- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages
    (if needed)
    - Static load balancing
  mini-batch-id % stage-replica-id
    - F/B paths consistent
  - How many items to admit?
    - Wild, S, less?
- In **steady** state:

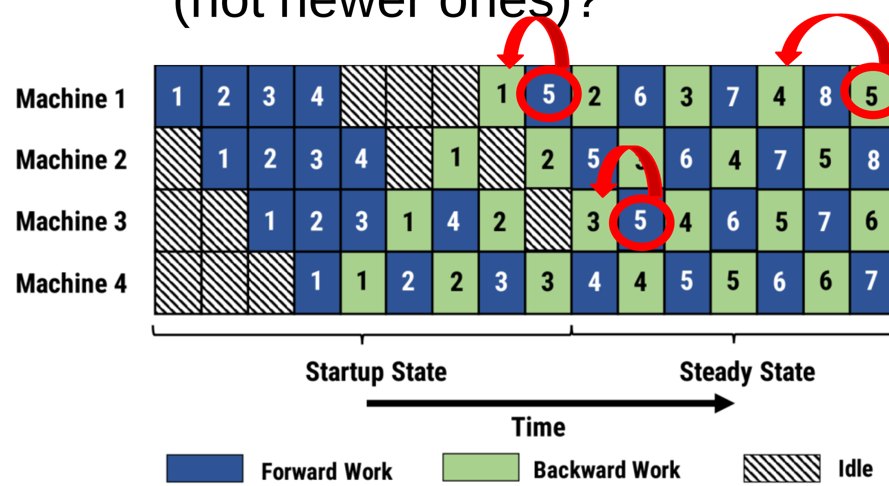
Each new item learns from a previous (new) mini-batch
  - Stage alternates between F/B



Challenges

- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages (if needed)
    - Static load balancing mini-batch-id % stage-replica-id
    - F/B paths consistent
  - How many items to admit?
    - Wild, S, less?
- In **steady** state: Each new item learns from a previous (new) mini-batch
  - Stage alternates between F/B

- Statistical Efficiency
  - Affected by feature map and model parameter versions
  - Each incomplete mini-batch needs to stash feature maps between F and B passes
  - Should admitted items see the updates of the same set of minibatches across levels (not newer ones)?

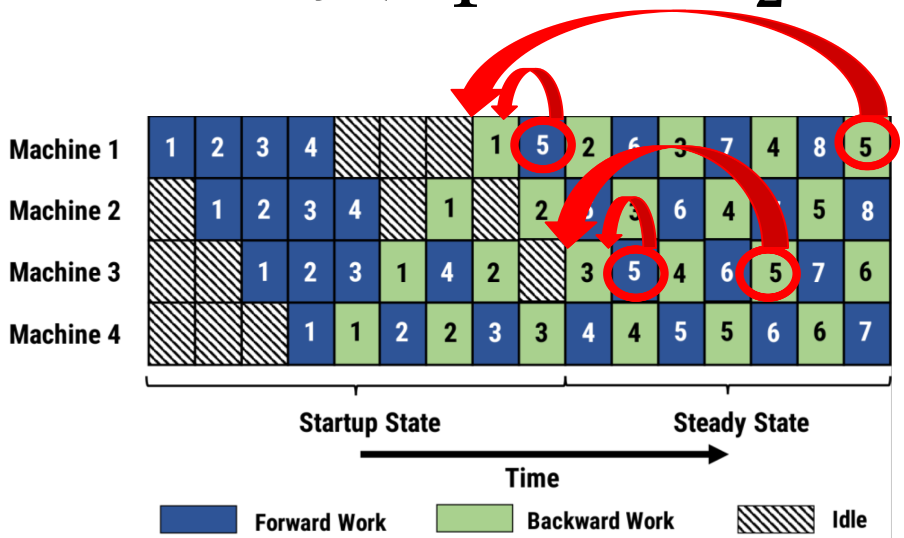


Valid Gradients Or Bust

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{\mu} * \nabla \boldsymbol{f}(\boldsymbol{w}_1^{(t)}, \boldsymbol{w}_2^{(t)}, \ldots, \boldsymbol{w}_n^{(t)})$$

PipeDream with wt stashing is critical for valid gradients (use same wts across F and B *within* every stage)

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{\mu} * \nabla \boldsymbol{f}(\boldsymbol{w}_1^{(t-n+1)}, \boldsymbol{w}_2^{(t-n+2)}, \ldots, \boldsymbol{w}_n^{(t)})$$



Valid Gradients Or Bust

$$\boldsymbol{w^{(t+1)} = w^{(t)} - \ \mu * \nabla f(w_1^{(t)}, w_2^{(t)}, \ldots, w_n^{(t)})}$$

PipeDream with **wt stashing** is critical for valid gradients
(use same wts across F and B *within* every stage)

$$\boldsymbol{w^{(t+1)} = w^{(t)} - \ \mu * \nabla f(w_1^{(t-n+1)}, w_2^{(t-n+2)}, \ldots, w_n^{(t)})}$$

PipeDream with **vertical sync** maps to bounded staleness
(use same wts across all stages)

$$\boldsymbol{w^{(t+1)} = w^{(t)} - \ \mu * \nabla f(w_1^{(t-n+1)}, w_2^{(t-n+1)}, \ldots, w_n^{(t-n+1)})}$$

Challenges

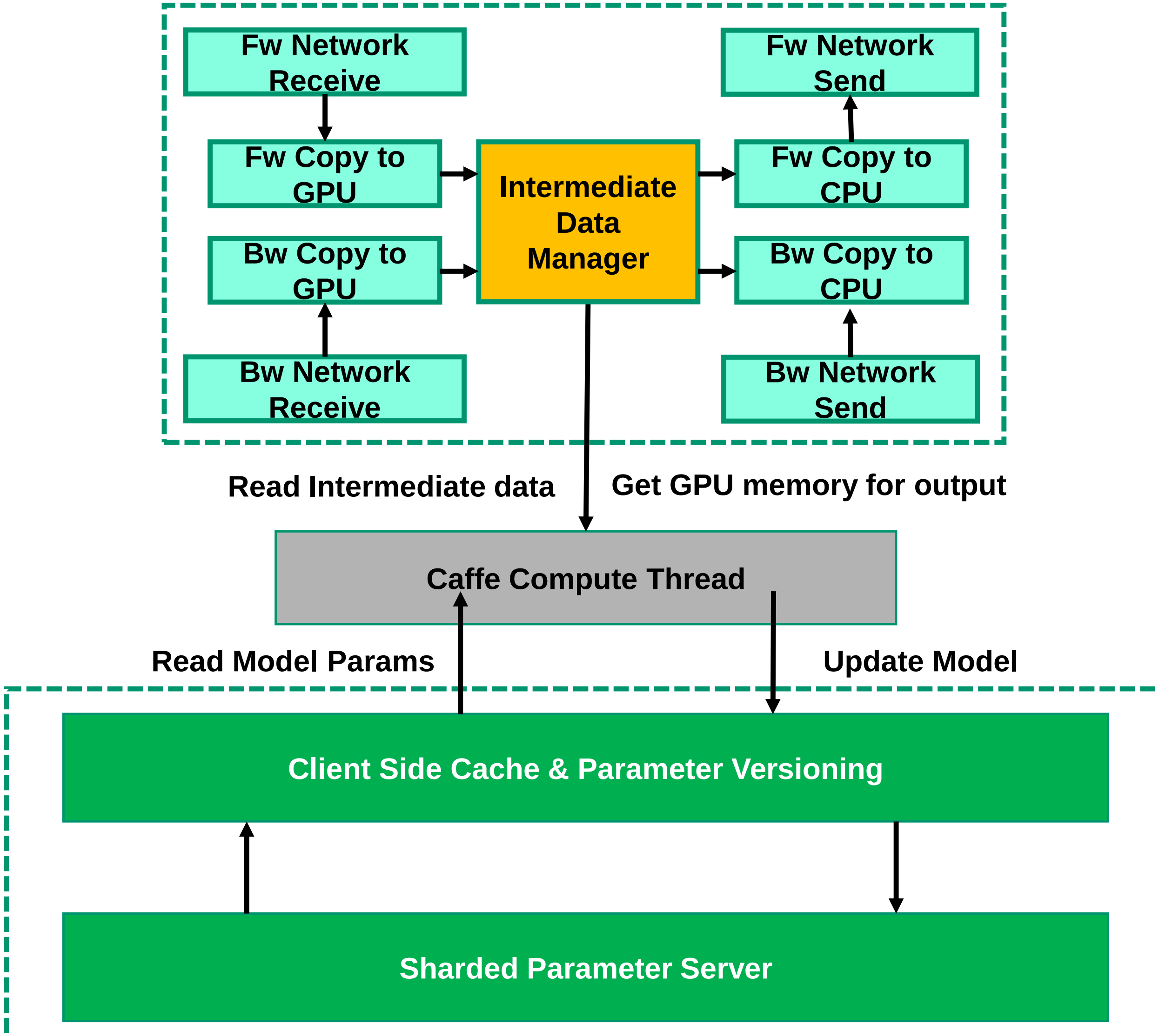
- No bubbles in pipeline
  - How should we divvy work across machines?
  - Replicate stages
    (if needed)
    – Static load balancing
  mini-batch-id % stage-replica-id
    – F/B paths consistent
  - How many items to admit?
    – Wild, S, less?
- In **steady** state:

Each new item learns from a previous (new) mini-batch

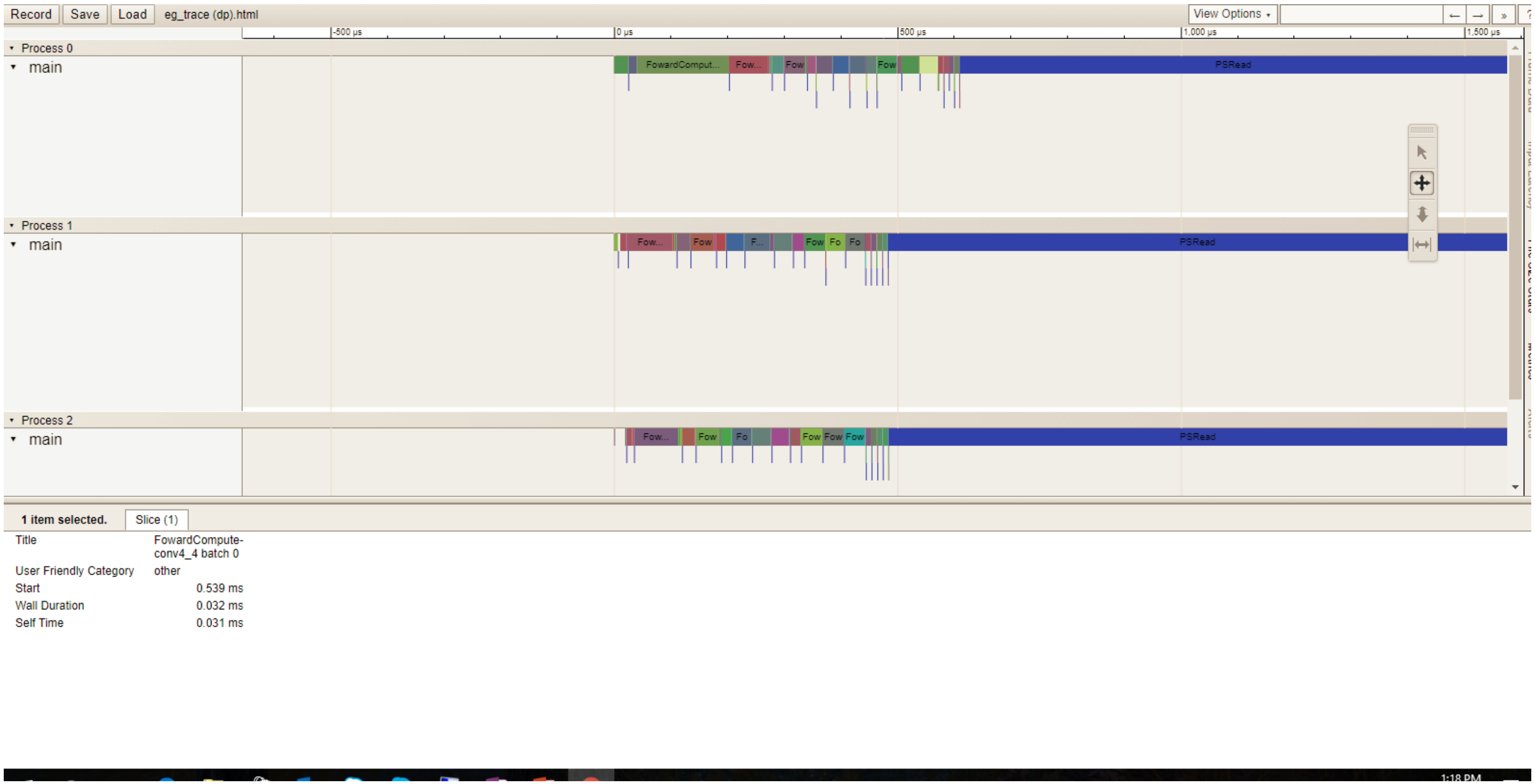
  - Stage alternates between F/B

- Statistical Efficiency
  - Affected by feature map (FM) and model parameter versions
  - Each incomplete mini-batch needs to stash feature maps between F and B passes
  - Should admitted items see the updates of the same set of minibatches across levels (not newer ones)?
- Memory Manager
  - Stashing: Static memory buffer pool (FMs, wts, wt-updates)
  - Use same wts across F/B within every stage (**valid gradient**)
  - Vertical Sync

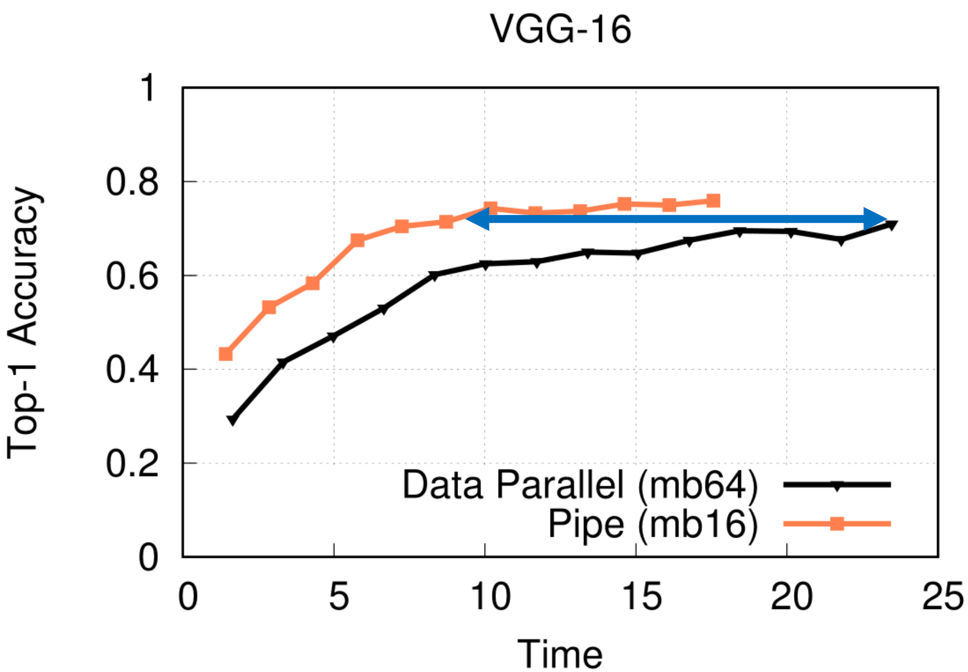
Stage Runtime Architecture

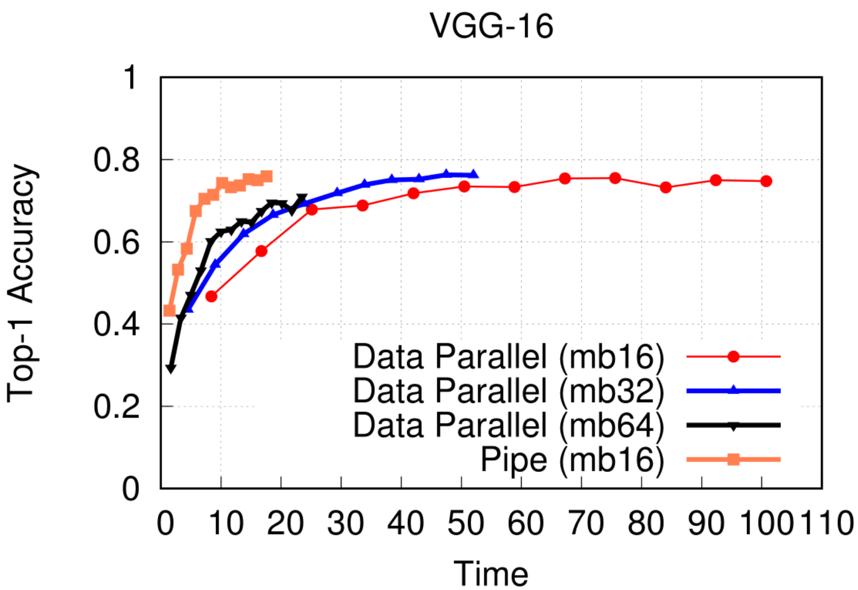


Visualization / Debugging



Encouraging results

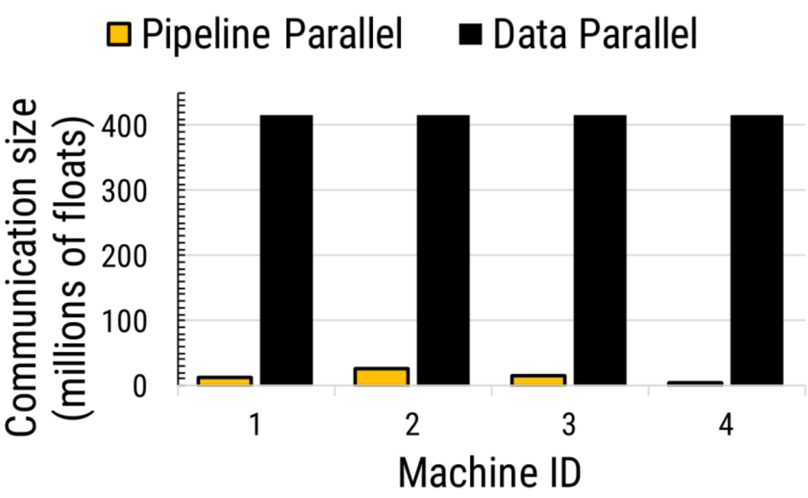


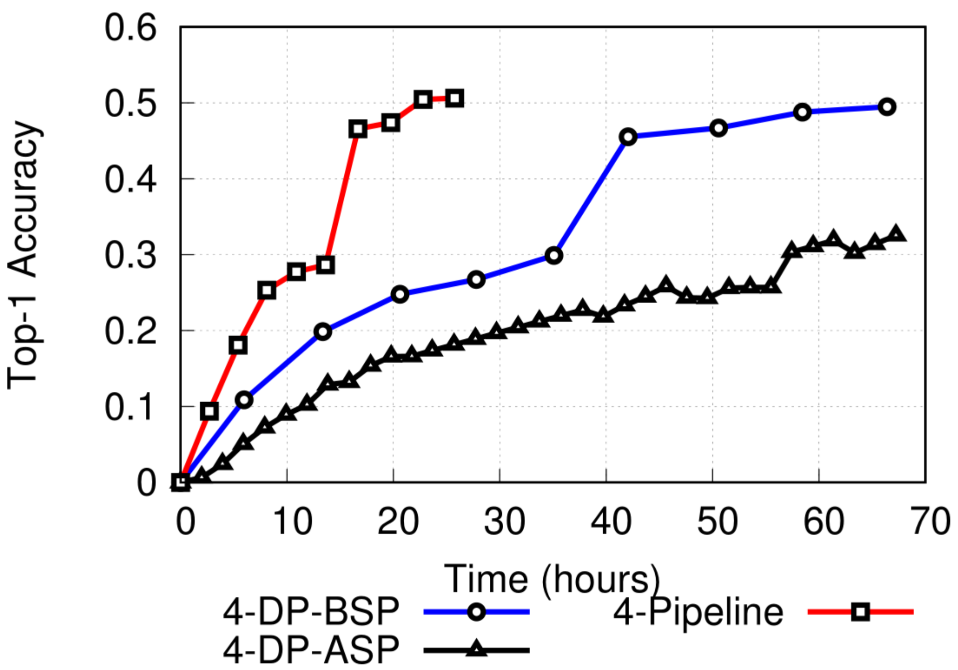


Up to 3x faster, more efficient

Effect of smaller minibatches on hardware efficiency and stat efficiency in data parallel runs

Restructure computation to push a lot less data on the network





	Single Machine Training	Multi-Machine Training
Memory, Computation	**Gist** Train larger networks on a *single* GPU by reducing memory footprint. **Up to 2x compression ratio**	**PipeDream** Efficiently scale training. New way to parallelize DNN computation. **3x faster training**
Interconnects	**Blink** Blazingly fast transfers over PCIe + NVLink + IB **Collective transfer primitives 8x faster than NCCL**	**Hub** Fast parameter servers **Up to 3x speedup**

	Single Machine Training	Multi-Machine Training
Memory, Computation	**Gist**	**PipeDream**
Interconnects	**Blink**	**Hub**

TBD

Training **B**enchmark for **D**NNs

Diversity in workloads, hardware, and frameworks.

Fin