Tencent 腾讯

Training ImageNet in 4 minutes with Tencent Jizhi

Yangzihao Wang, Haidong Rong
Cloud Architecture Platform Dept. TEG at Tencent

Agenda

- Motivations
- System implementation and optimizations
- Introduction of Jizhi Platform
- Case Studies
- Problems and Countermeasures

1 Motivations

Problems we try to solve

- *Academic:*
  - *Difficult to train with large-batch and on clusters*
- *Industrial:*
  - *Complex/arbitrary training pipelines for models from different fields*
  - *Separation between experimental models and distributed trained industrial-level models*

Problems for academic research:

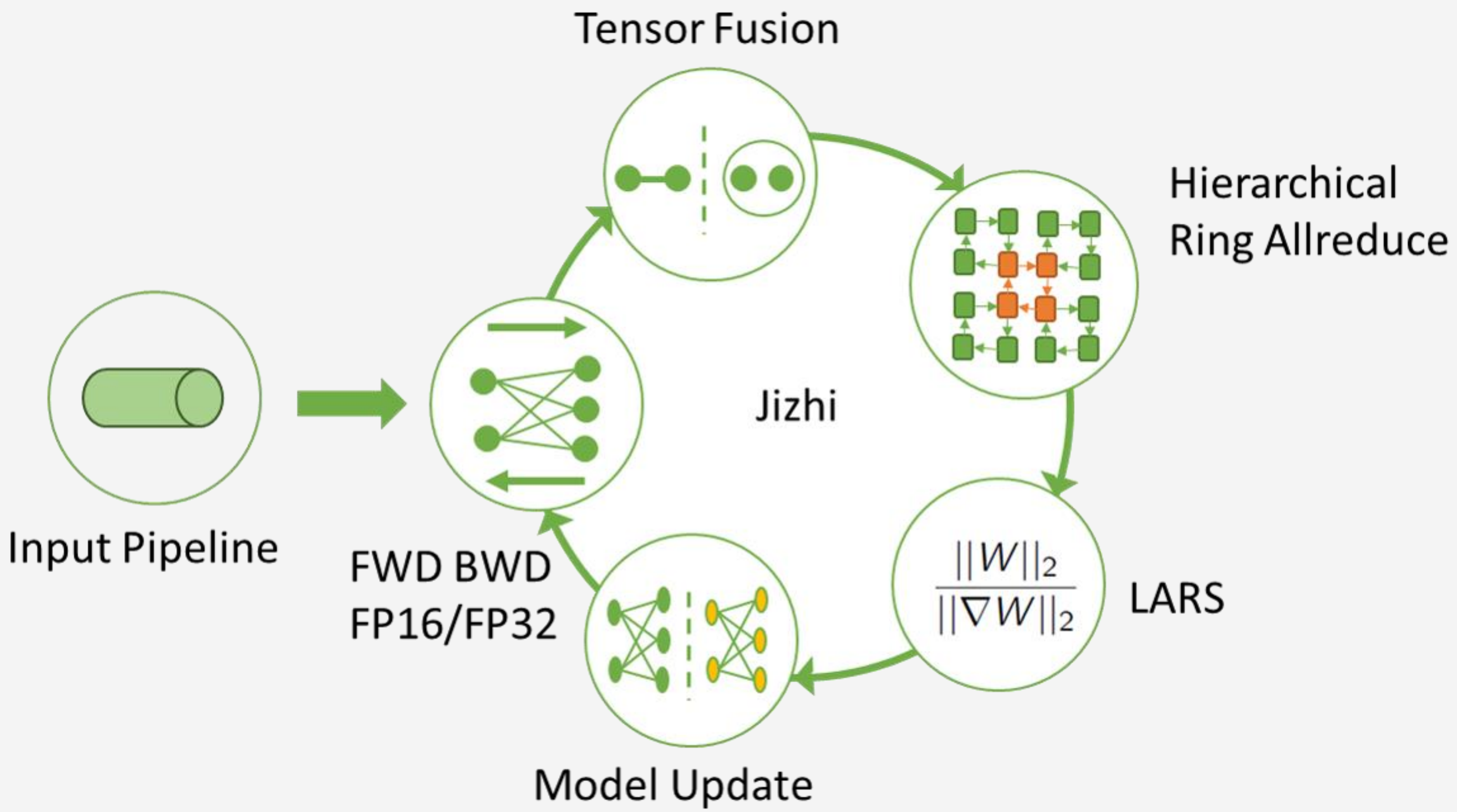
Two Challenges in large-batch distributed training:

How to maintain the same accuracy with large mini-batch training with SGD?

How to achieve near-linear scalability on large clusters?

Solution: Tencent Jizhi

A High Performance Distributed Deep Learning Training Platform



Goals for industrial applications:

High-Performance: Integration of general optimization strategies

Efficiency: Modular feature combinations and flexible resource management

Usability: Automate/Standardize Stages in ML pipeline

2 System Implementation and Optimizations

Can we train ImageNet using 1024 GPUs with a batch size of 64K?

How fast can we do it?

Optimization Techniques for Large-Batch Training

Mixed-precision training with LARS

Improvements on model architecture

Improvements on communication strategies

Mixed-precision Training with LARS

Layer-wise Adaptive Rate Scaling:

Set local learning rate per layer to stabilize the rate scaling:

$$\Delta w_t^l = \gamma \cdot \eta \cdot \frac{||w^l||}{||\nabla L(w^l))||} \cdot \nabla L(w_t^l)$$

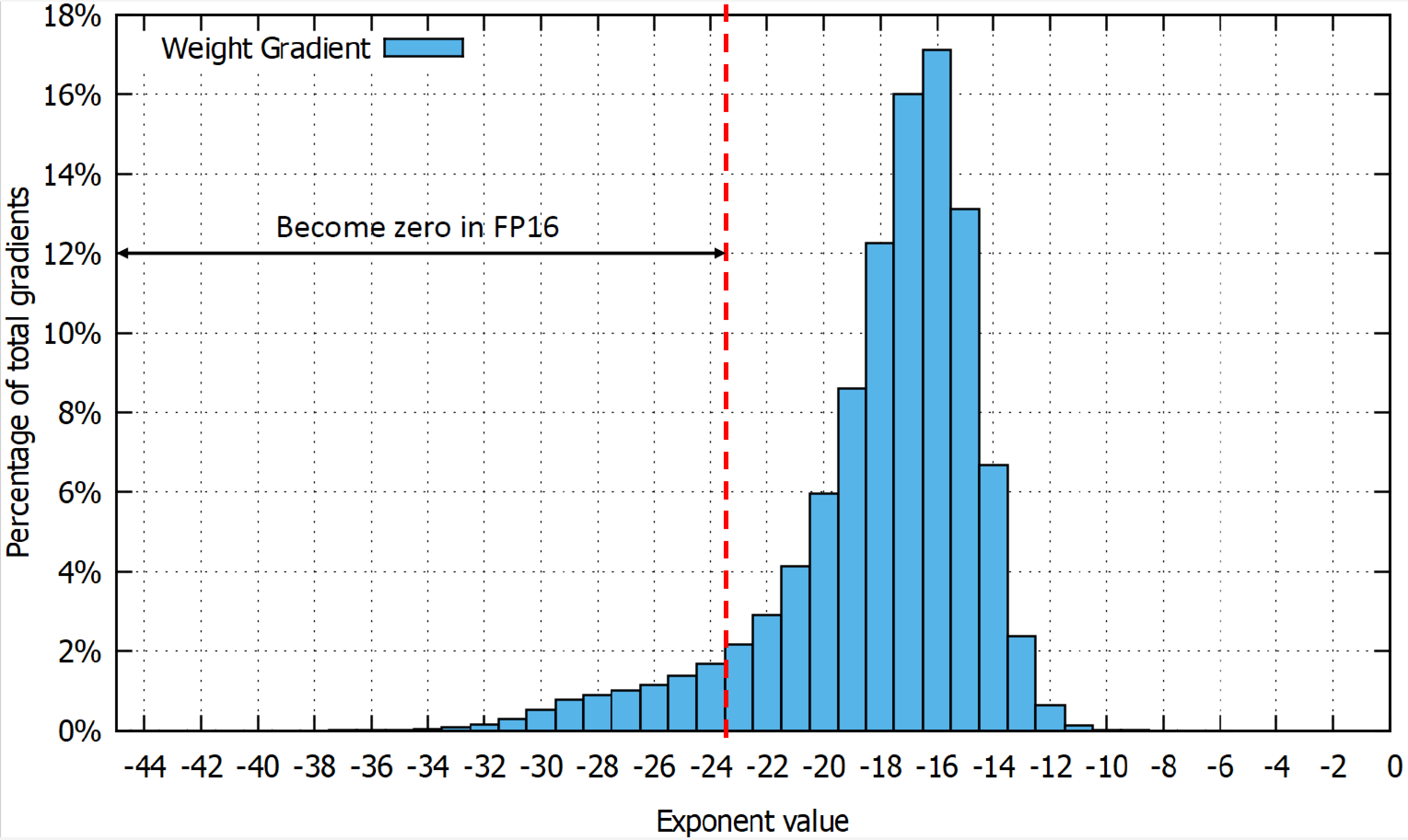
Can be used with momentum and weight decay in SGD:

Algorithm 1 SGD with LARS. Example with weight decay, momentum and polynomial LR decay.

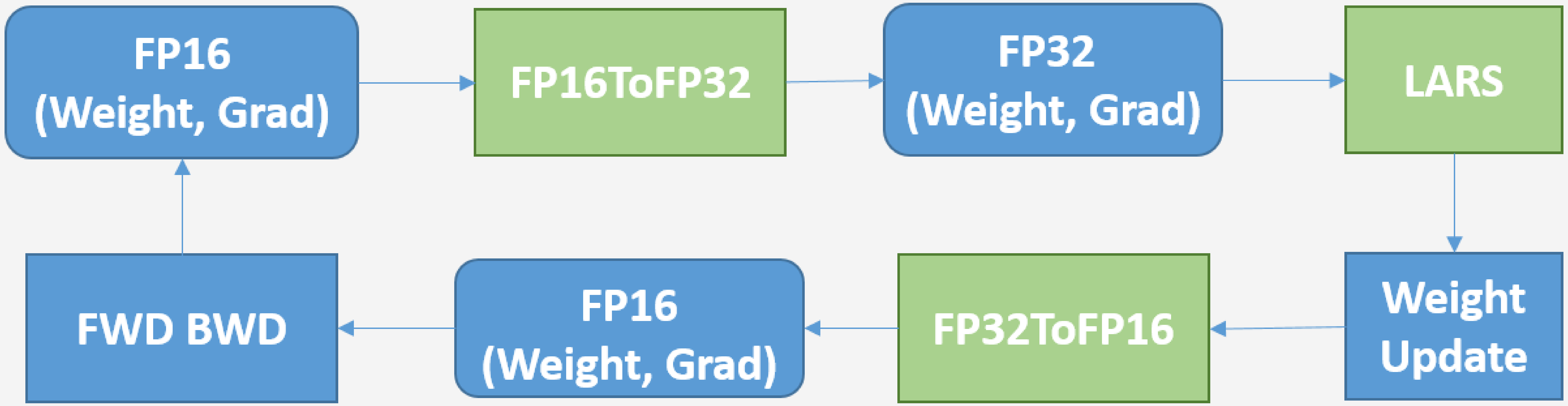
Parameters: base LR γ_0, momentum m, weight decay β, LARS coefficient η, number of steps T
Init: $t = 0, v = 0$. Init weight w_0^l for each layer l
while $t < T$ for each layer l **do**
  $g_t^l \leftarrow \nabla L(w_t^l)$ (obtain a stochastic gradient for the current mini-batch)
  $\gamma_t \leftarrow \gamma_0 * \left(1 - \frac{t}{T}\right)^2$ (compute the global learning rate)
  $\lambda^l \leftarrow \frac{||w_t^l||}{||g_t^l|| + \beta ||w_t^l||}$ (compute the local LR λ^l)
  $v_{t+1}^l \leftarrow m v_t^l + \gamma_{t+1} * \lambda^l * (g_t^l + \beta w_t^l)$ (update the momentum)
  $w_{t+1}^l \leftarrow w_t^l - v_{t+1}^l$ (update the weights)
end while

Mixed-precision Training with LARS

A different story for FP16:



Mixed-precision Training with LARS



Mixed-precision Training with LARS

Table 1: Effectiveness of using LARS on ResNet-50

Mini-Batch Size	Number of Epochs	LARS	Top-1 Accuracy
64K	90	NO	73.2%
64K	90	YES	76.2%

Improvements on Model Architecture

Do not do weight decay on bias and \beta, \gamma in BN:

A typical practice to penalize only the weights of the affine transformation at each layer and leaves the biases unregularized:

$$w_i^{t+1} = w_i^t - \eta \frac{\partial E}{\partial w_i^t} - \lambda w_i^t$$

Do not penalize the trainable params \beta, \gamma in BN:

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$$

Improvements on Model Architecture

Do not do weight decay on bias and \beta, \gamma in BN:

Effect of Regularization with b, β and γ for AlexNet

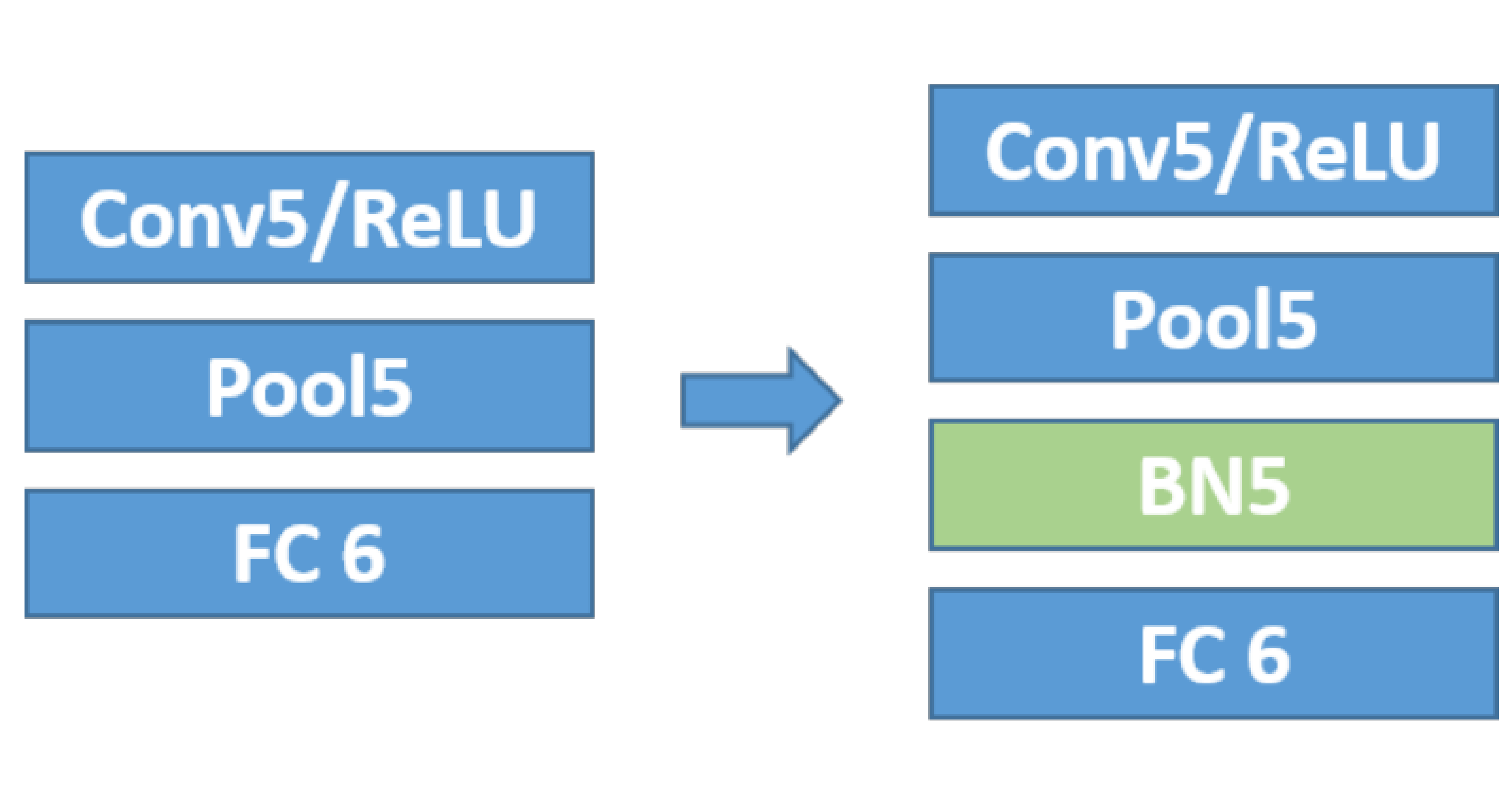
Batch	Epochs	Regularize b, β and γ	Top1
64K	95	Yes	55.8%
64K	95	No	57.1%

Effect of improvements to ResNet-50 Training

Batch	No Decay BN	Top1
64K	×	71.9%
64K	✓	76.2%

Improvements on Model Architecture

Insert BN layer after Pool5 in AlexNet



Improvements on Model Architecture

Insert BN layer after Pool5 in AlexNet

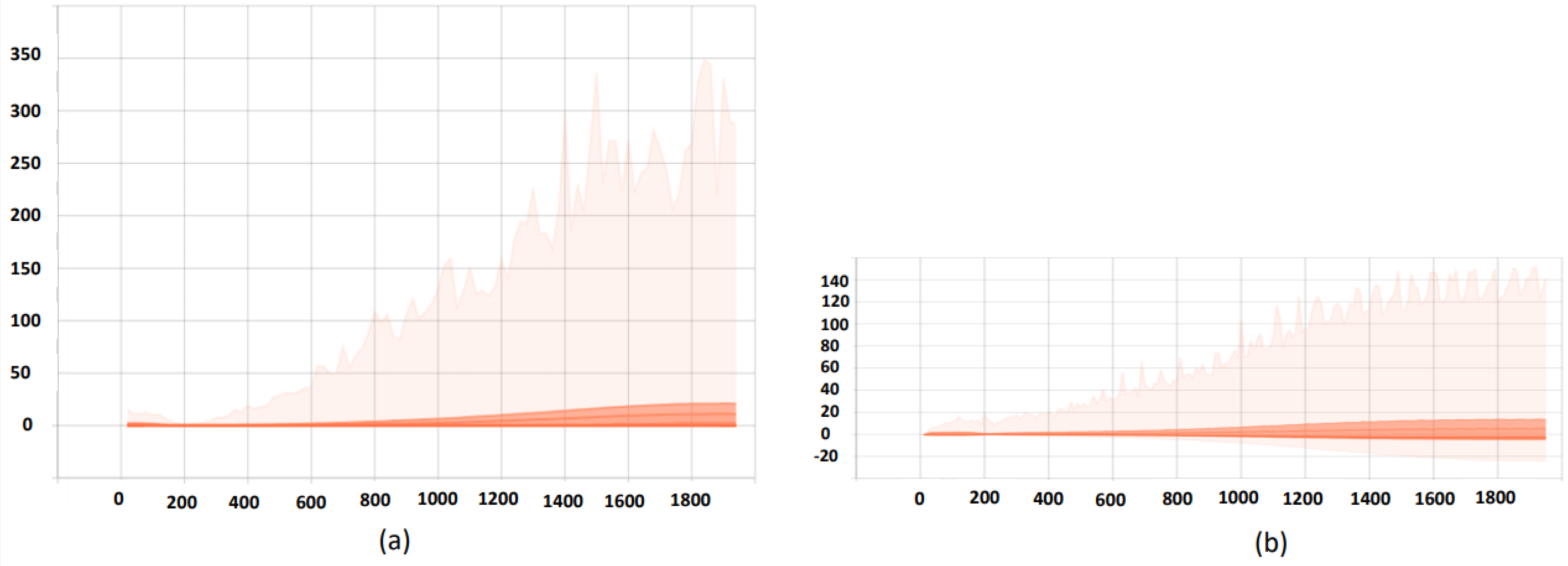


Figure 4: Feature Map Distribution of Pool5(a) and Pool5-BN5(b) of AlexNet as shown in Figure 3. (the horizontal axis is the training steps, the vertical axis is the feature map distributions.)

Improvements on Model Architecture

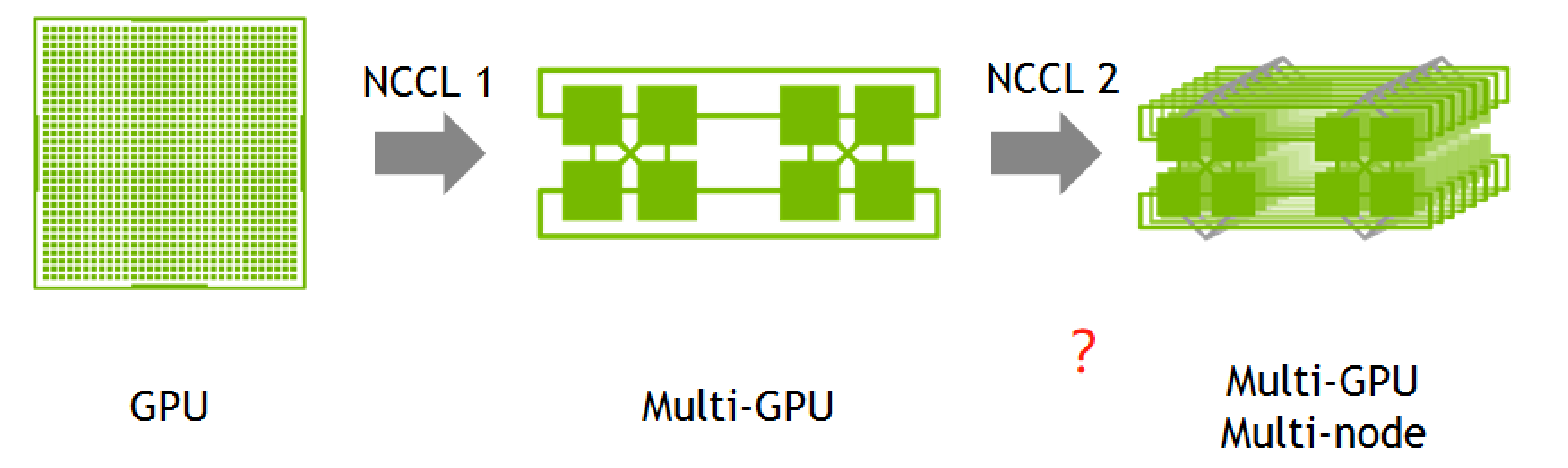
Insert BN layer after Pool5 in AlexNet

Batch	No Decay Bias	No Decay BN	pool5 BN	Top-1 Accuracy
64K	×	×	×	55.8%
64K	×	✓	×	56.3%
64K	✓	×	×	56.4%
64K	✓	✓	×	57.1%
64K	✓	✓	✓	58.8%

Improvements on Communication

For large batch training with distributed synchronized SGD, efficient gradients aggregation across all GPUs after each iteration is crucial to the training performance

Improvements on Communication



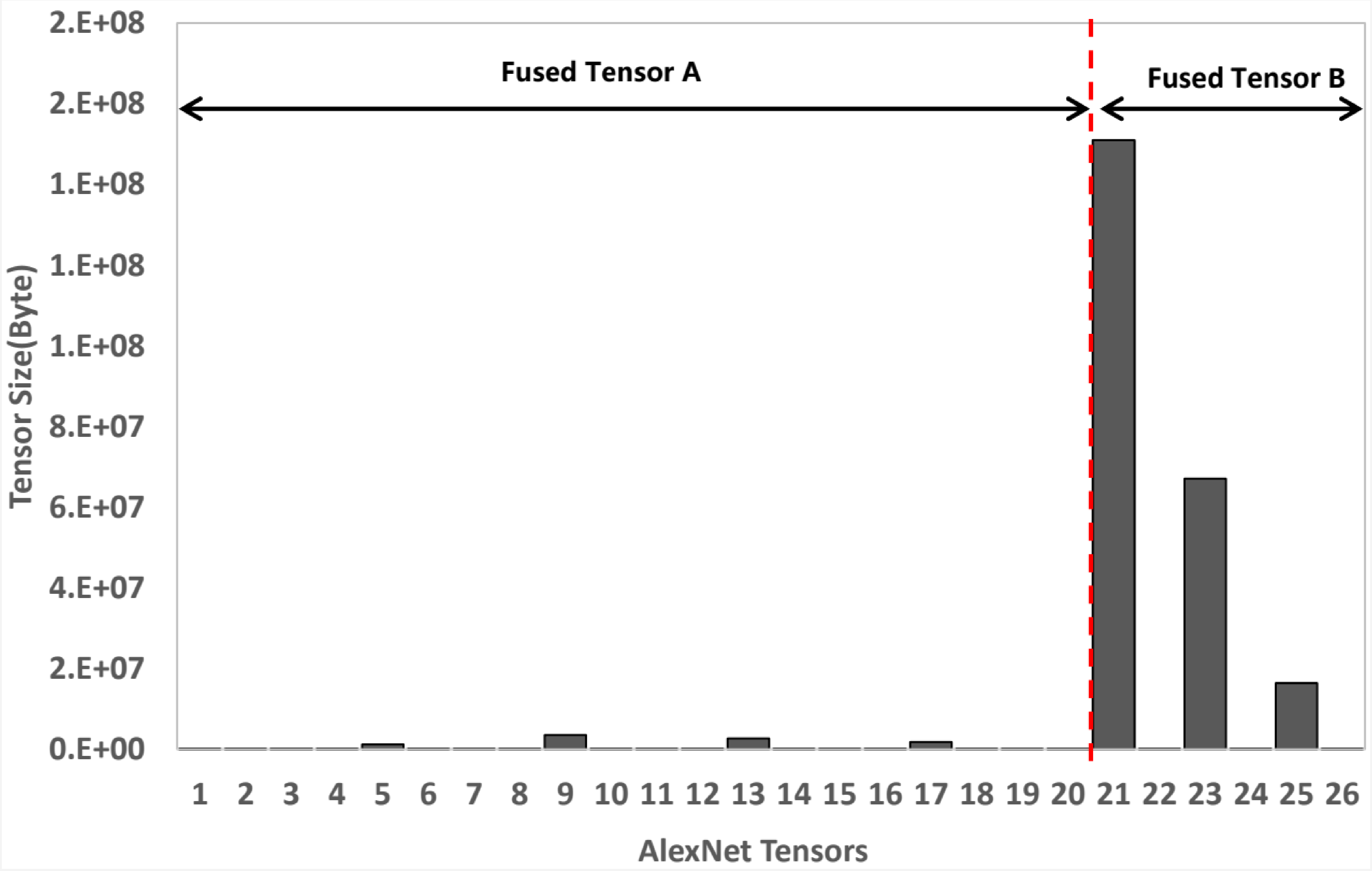
NCCL 2.0 alone cannot solve the problem:

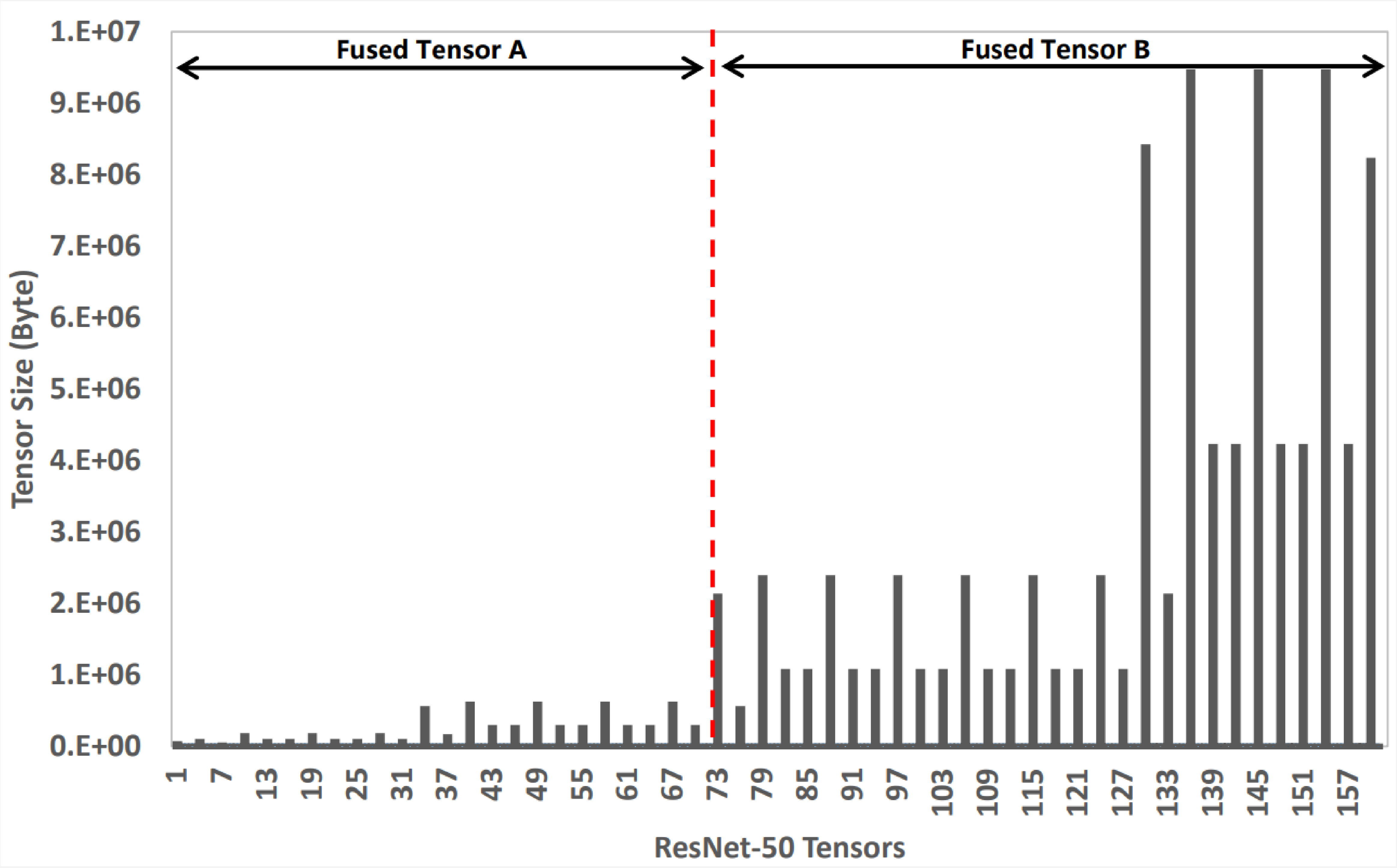
In a cluster with k GPUs, Ring all-reduce will split the data on each GPU into k chunks and do the reduce in $k-1$ iterations

When k gets larger, the messages passing between nodes will become smaller and fail to utilize the full bandwidth of the network

Improvements on Communication

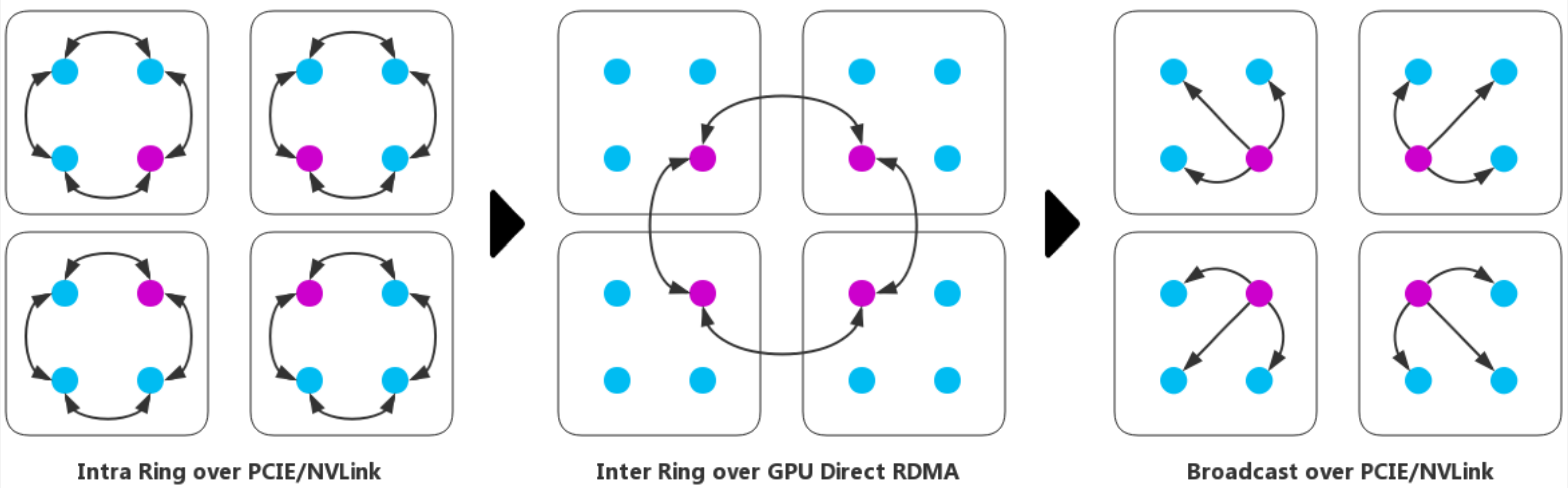
Tensor Fusion:





Improvements on Communication

Hierarchical All-Reduce:

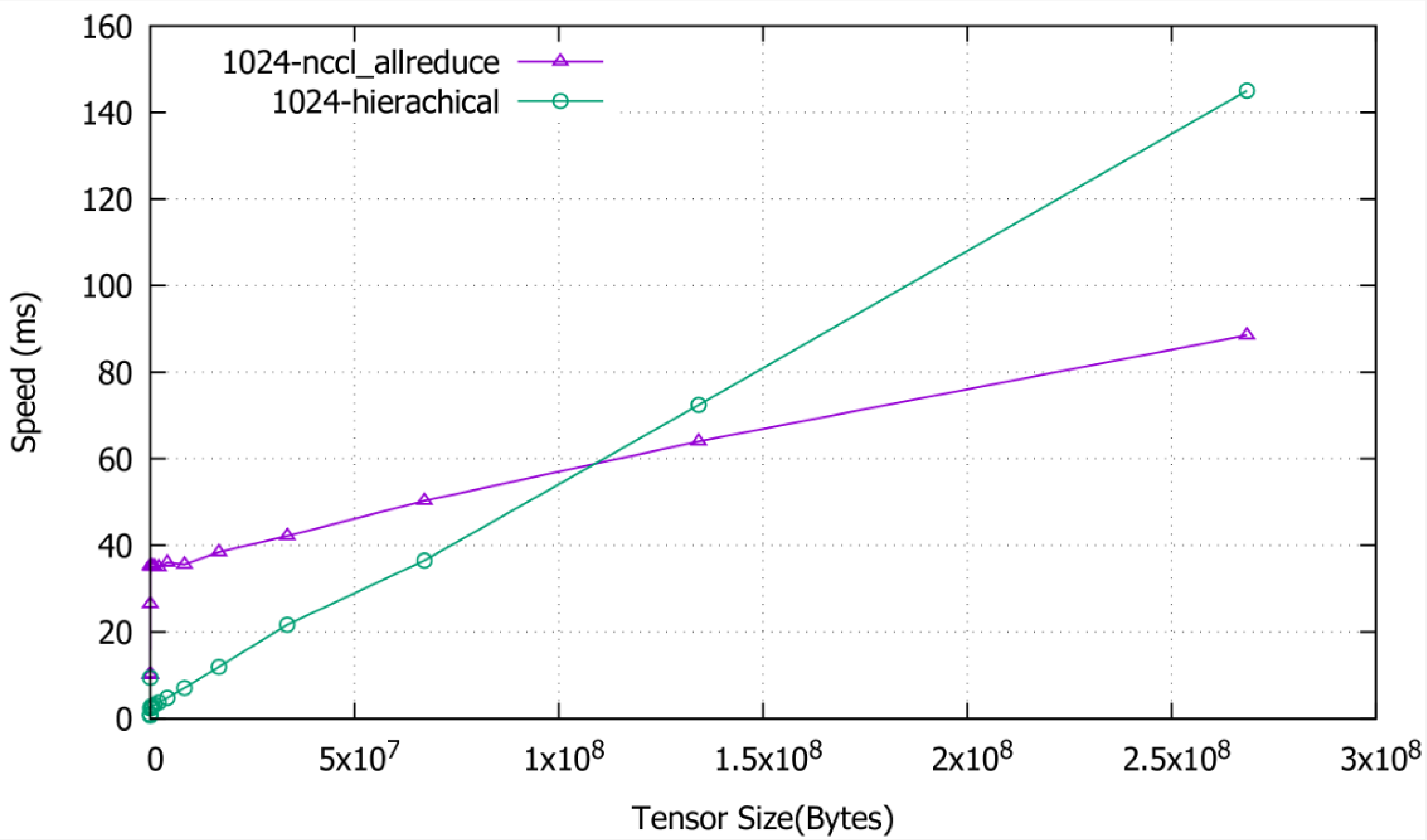


p GPUs, p/k groups:

Ring All-Reduce #steps: 2(p-1)
Hierarchical All-Reduce #steps: 4(k-1)+2(p/k-1)
In our case: p=1024, k=16 achieves the best performance

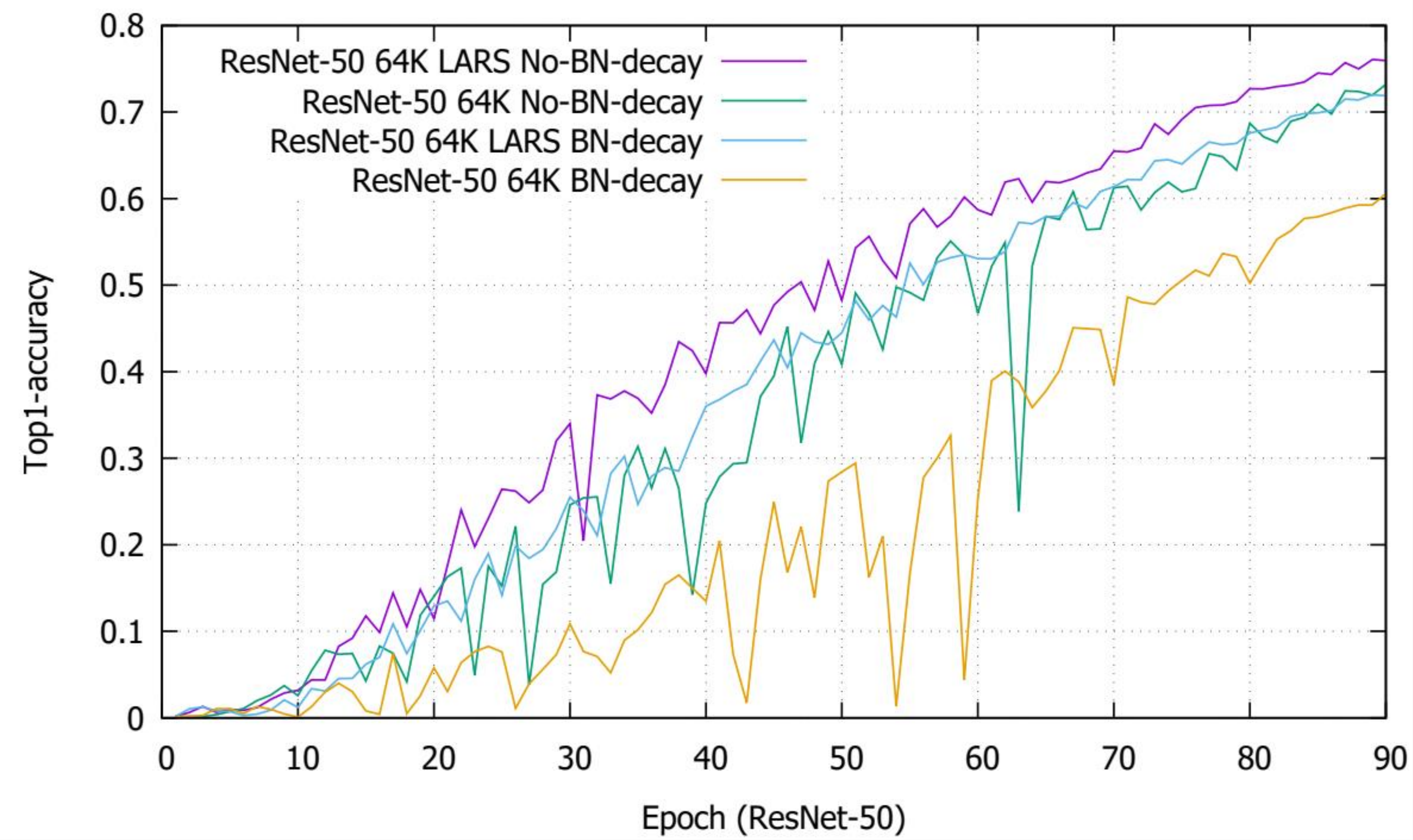
Improvements on Communication

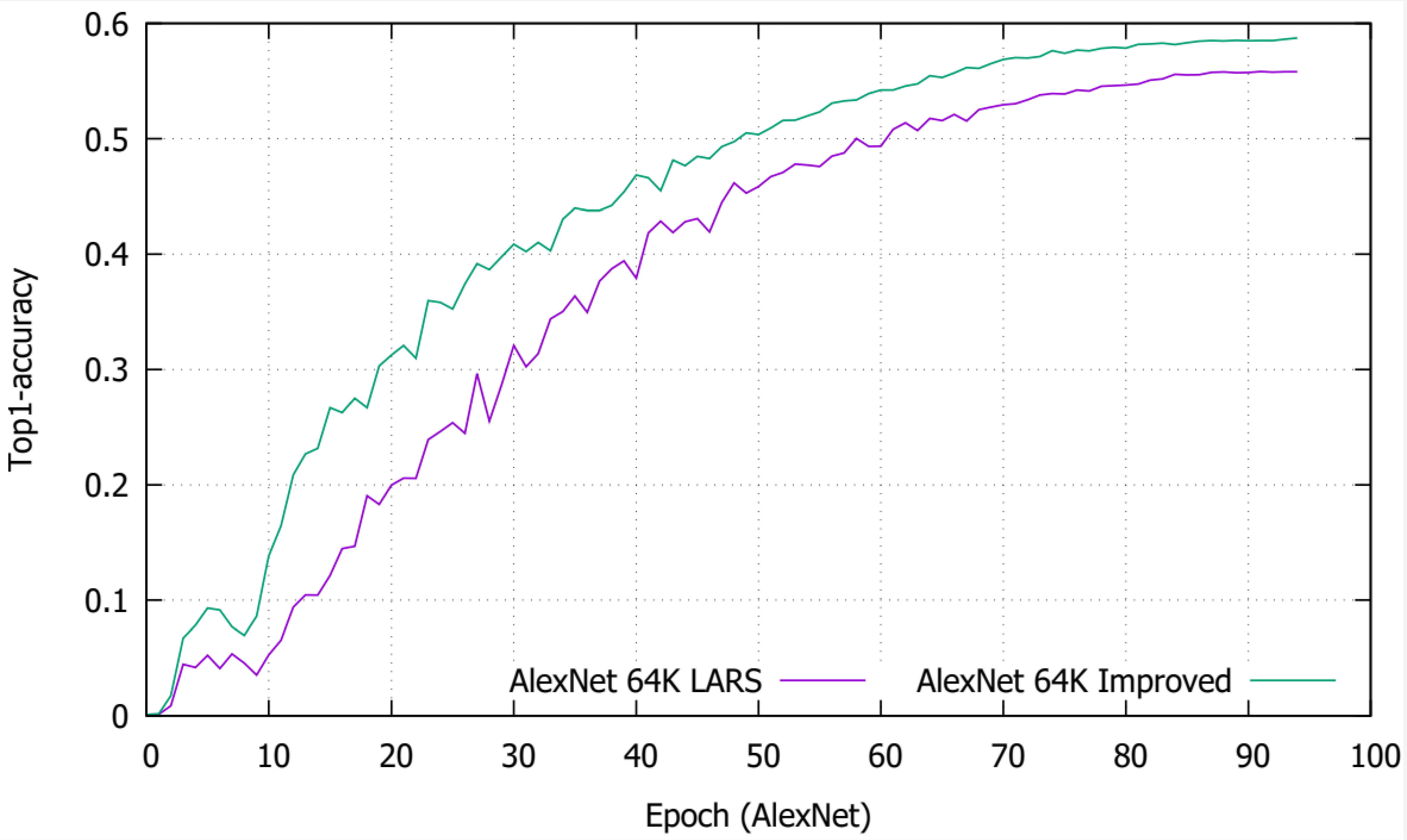
Hybrid All-Reduce:



Results

Both Models Converge to >= baseline accuracy:





Results

Both Models Converge to >= baseline accuracy, and fast!

Table 4: Compare AlexNet training with different teams

Team	Batch	Hardware	Software	Top-1 Accuracy	Time
You et al. [27]	512	DGX-1 station	NVCaffe	58.8%	6h 10m
You et al. [27]	32K	CPU × 1024	Intel Caffe	58.6%	11min
This work	**64K**	Tesla P40 × 512	TensorFlow	**58.8%**	**5m**
This work	**64K**	Tesla P40 × 1024	TensorFlow	**58.7%**	**4m**

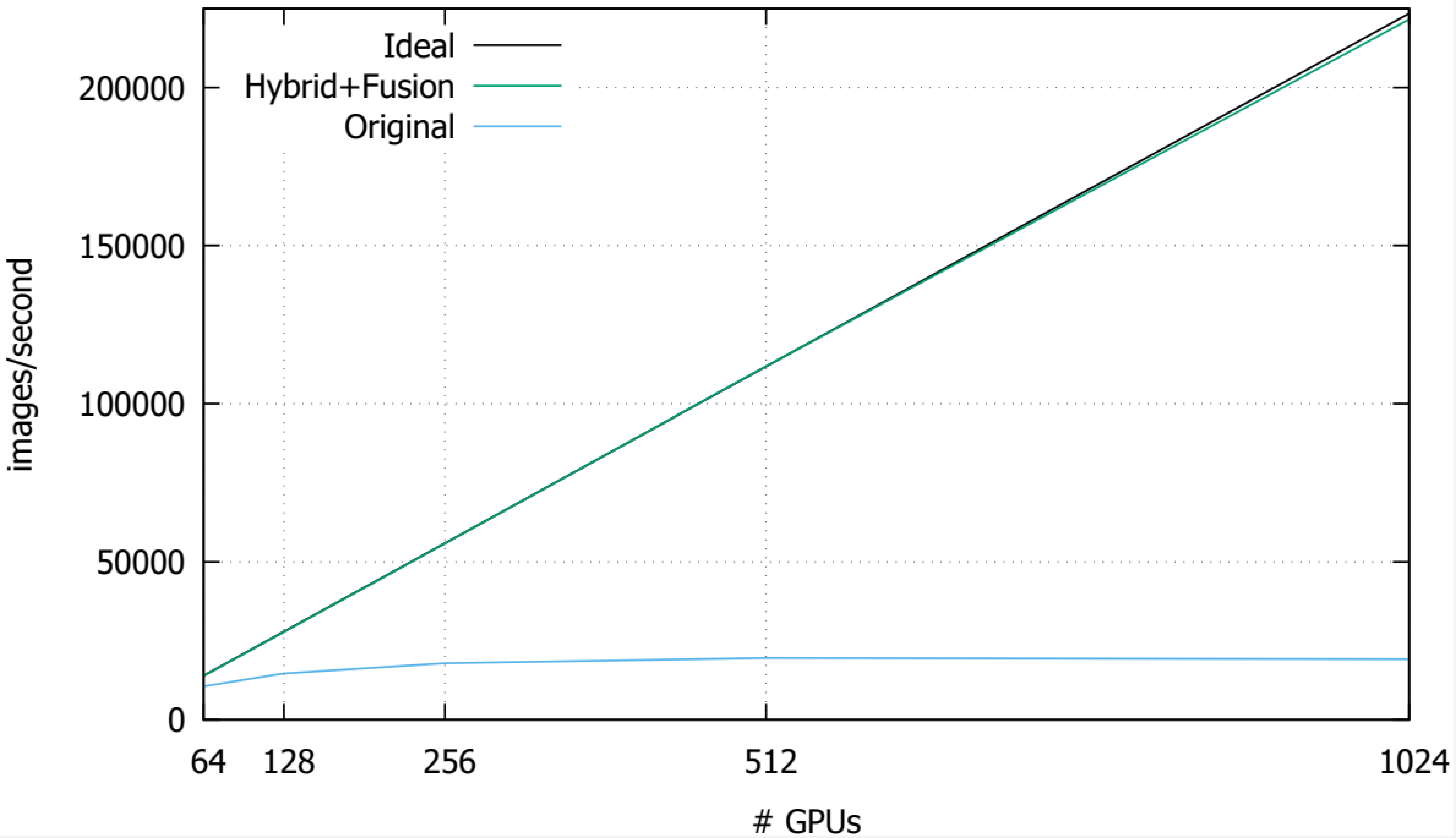
Table 5: Compare ResNet-50 training with different teams

Team	Batch	Hardware	Software	Top-1 Accuracy	Time
He et al. [13]	256	Tesla P100 × 8	Caffe	75.3%	29h
Goyal et al. [12]	8K	Tesla P100 × 256	Caffe2	76.3%	1h
Cho et al. [4]	8K	Tesla P100 × 256	Torch	75.0%	50min
Codreanu et al. [5]	32K	KNL × 1024	Intel Caffe	75.3%	42min
You et al. [27]	32K	KNL × 2048	Intel Caffe	75.4%	20min
Akiba et al. [2]	32K	Tesla P100 × 1024	Chainer	74.9%	15min
This work	**64K**	Tesla P40 × 1024	TensorFlow	**76.2%**	**8.7m**
This work	**64K**	Tesla P40 × 2048	TensorFlow	**75.8%**	**6.6m**

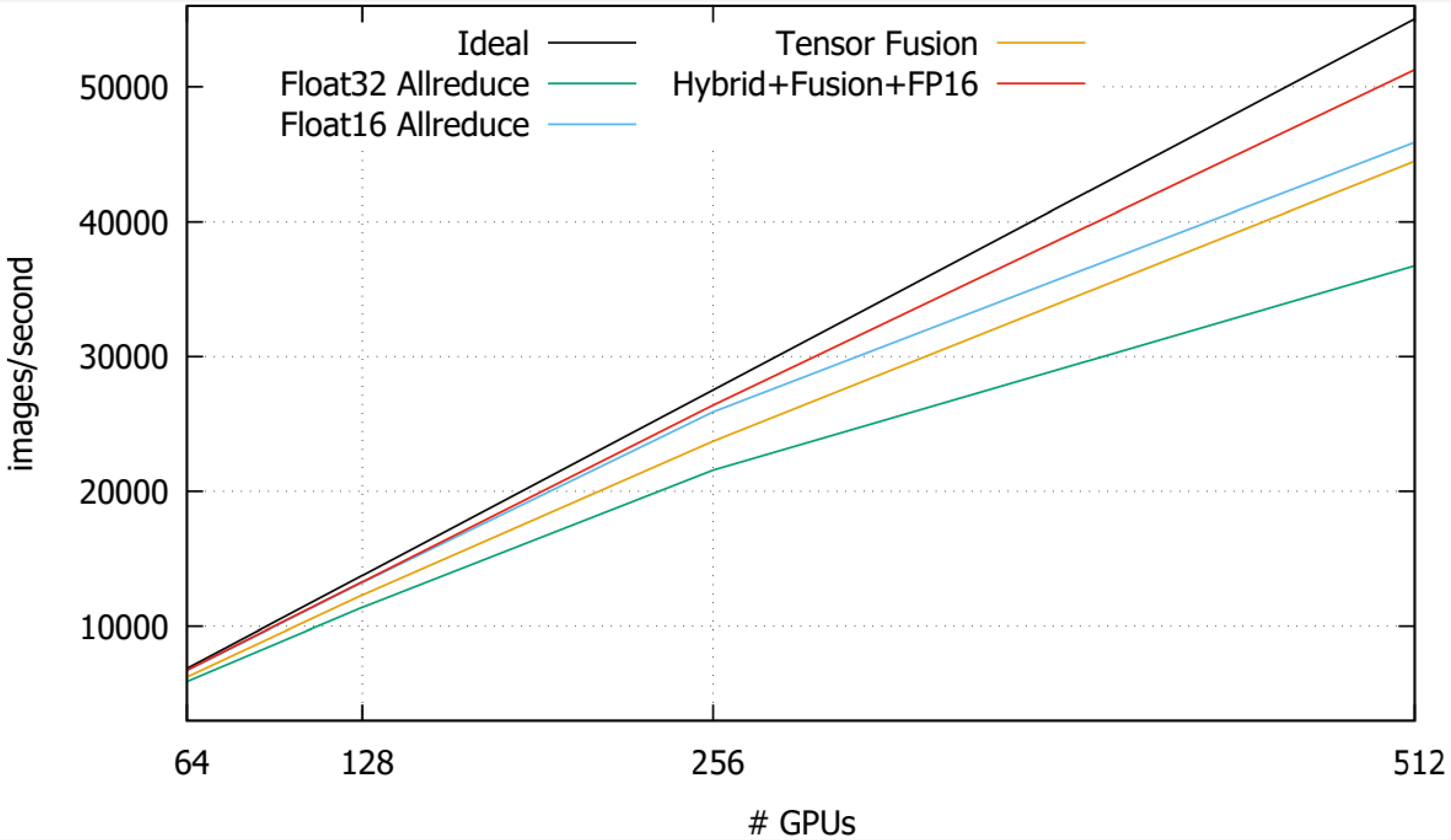
Results

Also we scale (almost) linearly:

ResNet-50



AlexNet



Future Works

AutoML and Network Optimizations

More Communication Optimizations

More Optimizers: Second-Order Optimization

More Execution Methods: Asynchronized Training

Agenda

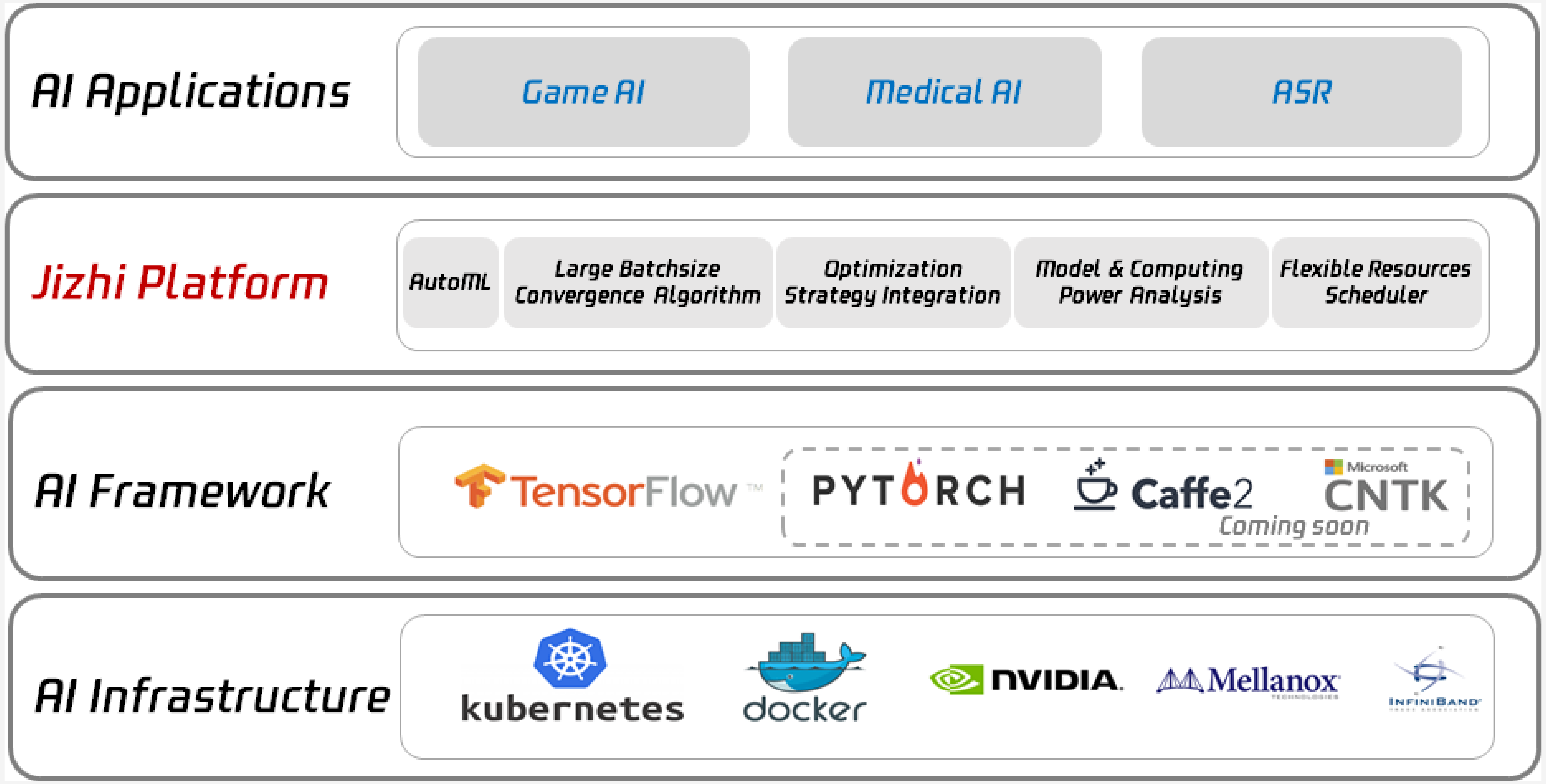
- Motivations
- System implementation and optimizations
- Introduction of Jizhi Platform
- Case Studies
- Problems and Countermeasures

3 Introduction of Jizhi Platform

What is the Jizhi Platform?

- A General-purpose AI Platform
- A Distinctive AI Accelerating Platform

Architecture of Jizhi Platform



The Platform has served for more than 5 internal businesses and created the practical commercial value.

Features of Tencent Jizhi

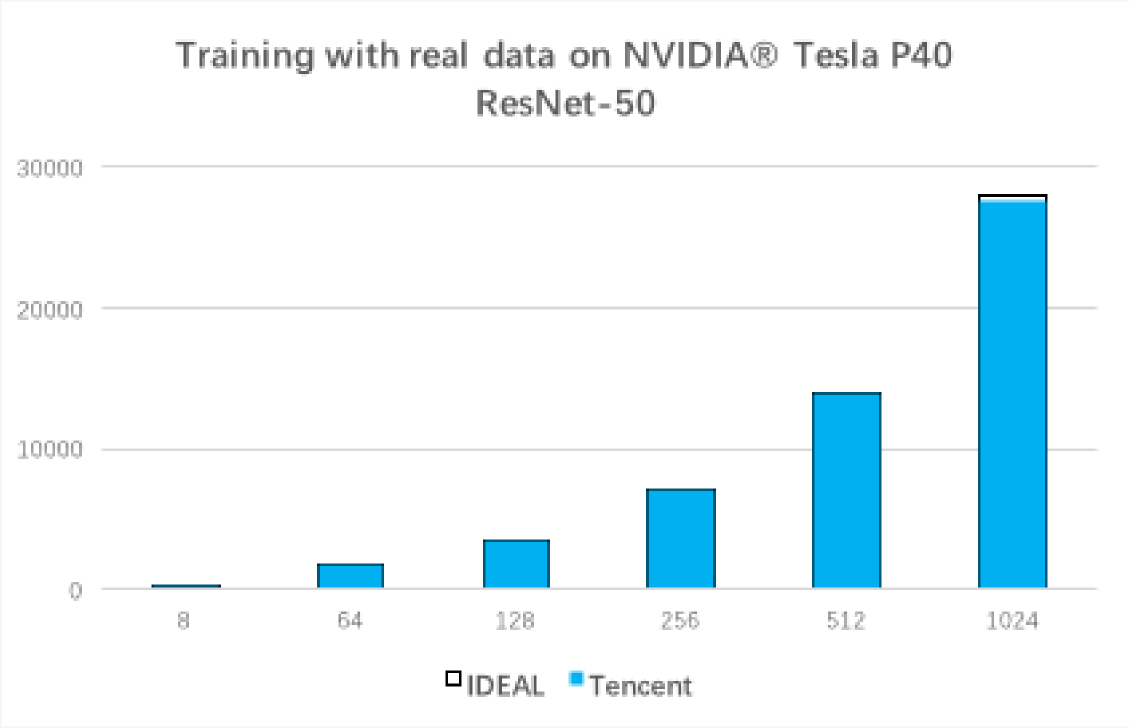
- High Efficiency
  - Integrating more than 60% computing resources of Tencent into one unified pool
  - Increasing Utilization Rate through Flexible Strategy
    - High priority : Exclusive with budget
    - Low priority : Shared once idle

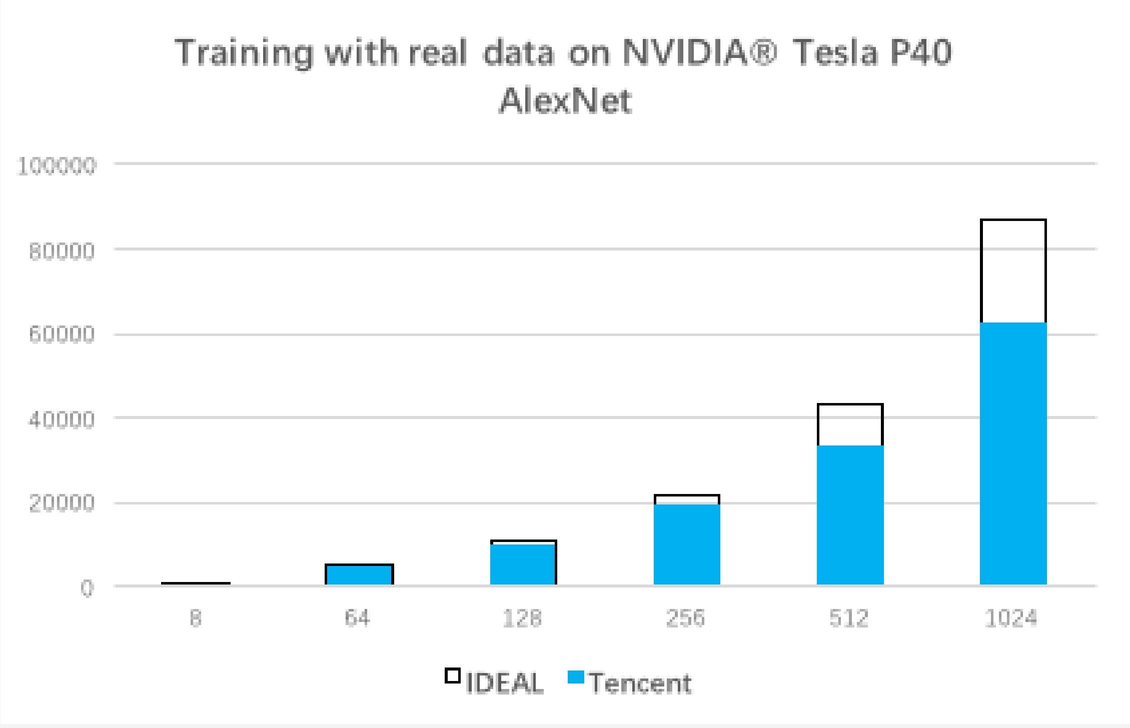
Features of Tencent Jizhi

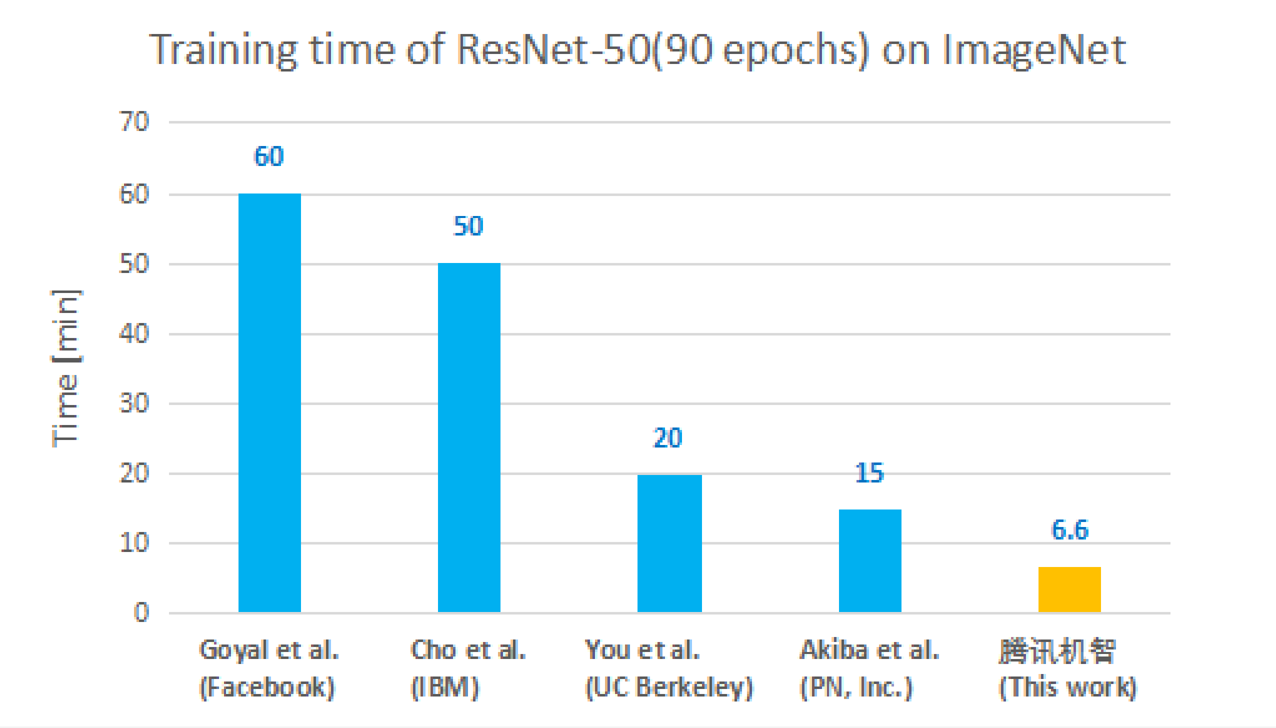
- High Performance:
  - Model level optimization (LARS, Mix precision training, etc.)
  - Framework level optimization (State-of-Art optimizer and loss function, etc.)
  - Platform level optimization (Tensor fusion, Hierarchical allreduce, etc.)

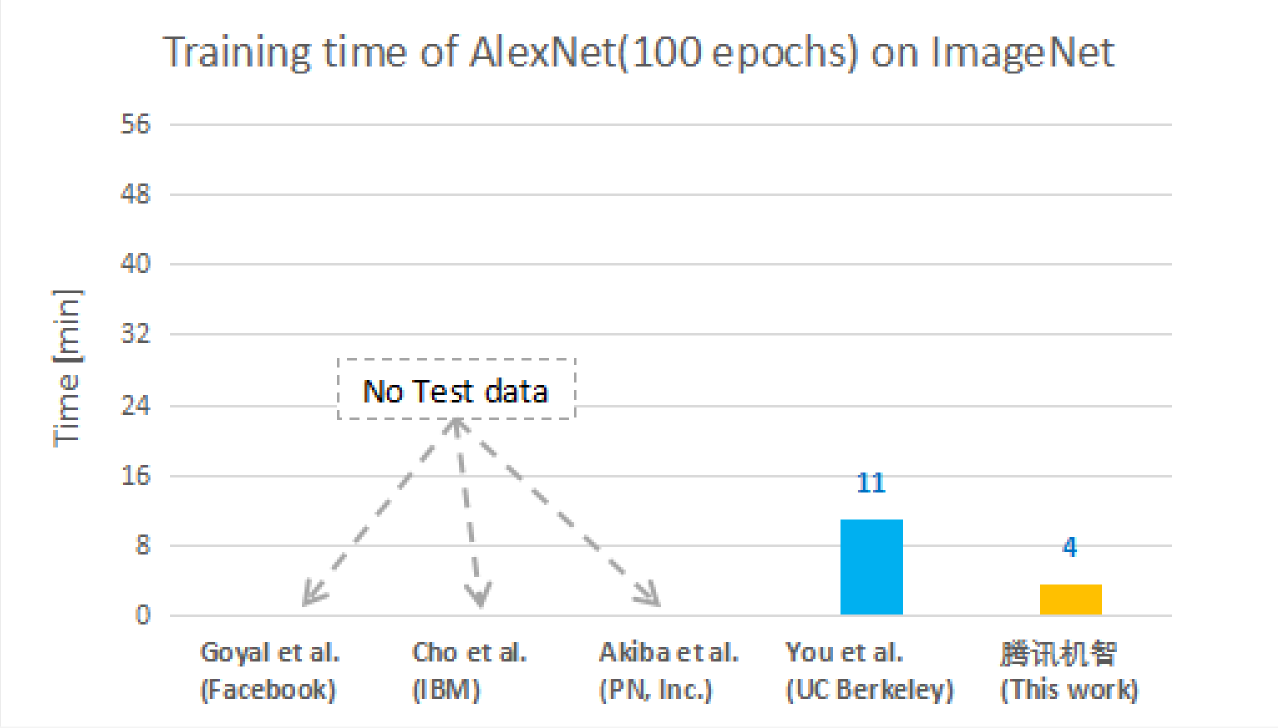
Features of Tencent Jizhi

- *High Performance*









- *Allocating more than 1000 GPUs for a single task*
- *Running on Super large-scale cluster with near-linear speedup*
- *And Supporting large batch size without significant accuracy loss*

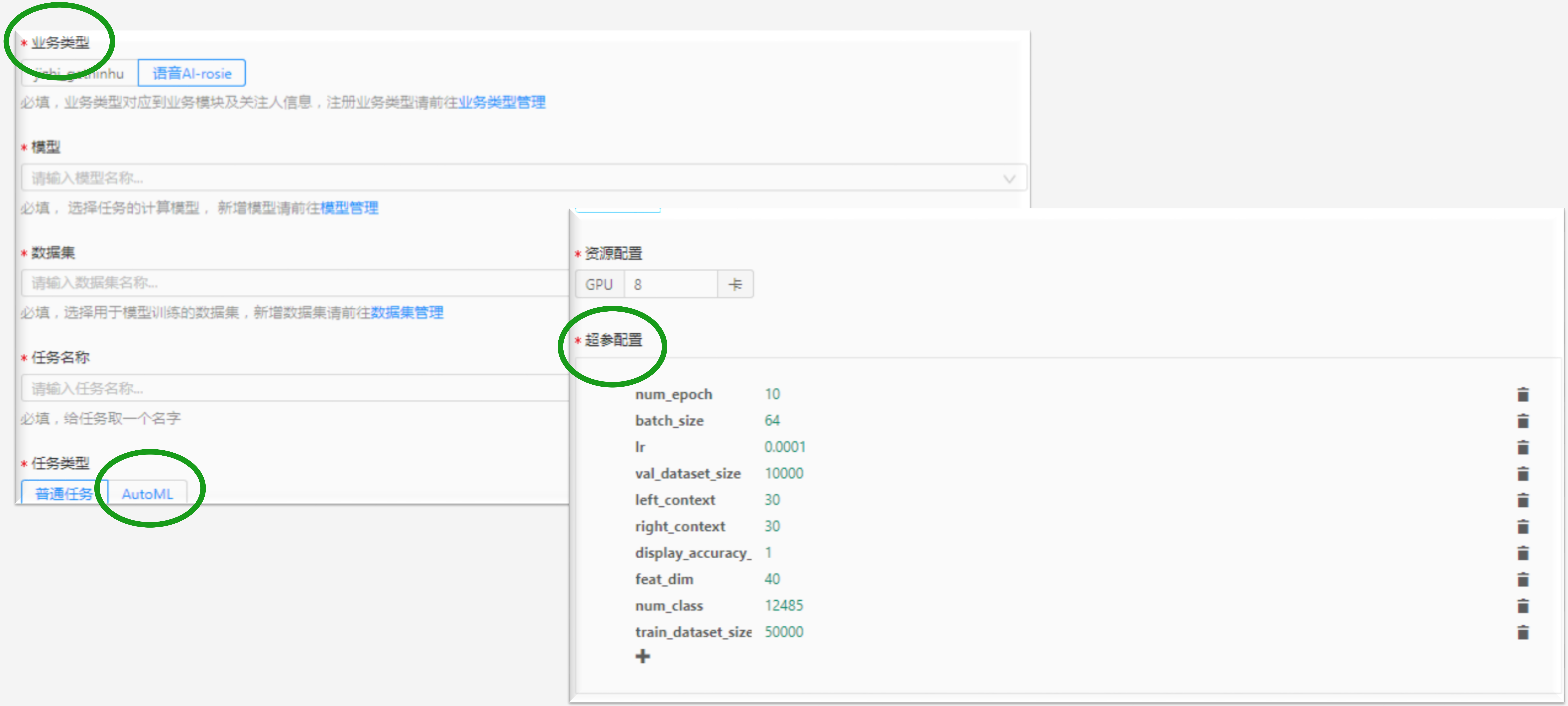
Features of Tencent Jizhi

- *High Usability*
  - *Automatic parallelization / Transparent to Model Engineers*
  - *Simple High-level API (network, dataset)*
  - *Support secondary development through an Open API set*

```python
class Network(object):
    def __init__(self, params):
        self.params = params

    def inference(self, images):...

    def cal_loss_accuracy_and_others(self, input_data):...

    def cal_accuracy_and_others(self, input_data):...
```

```python
def get_train_filename_list(params, path):...


def get_valid_filename_list(params, path):...


def get_samples(params, path, sample_file_tuple, queue):...
```

UI of Tencent Jizhi

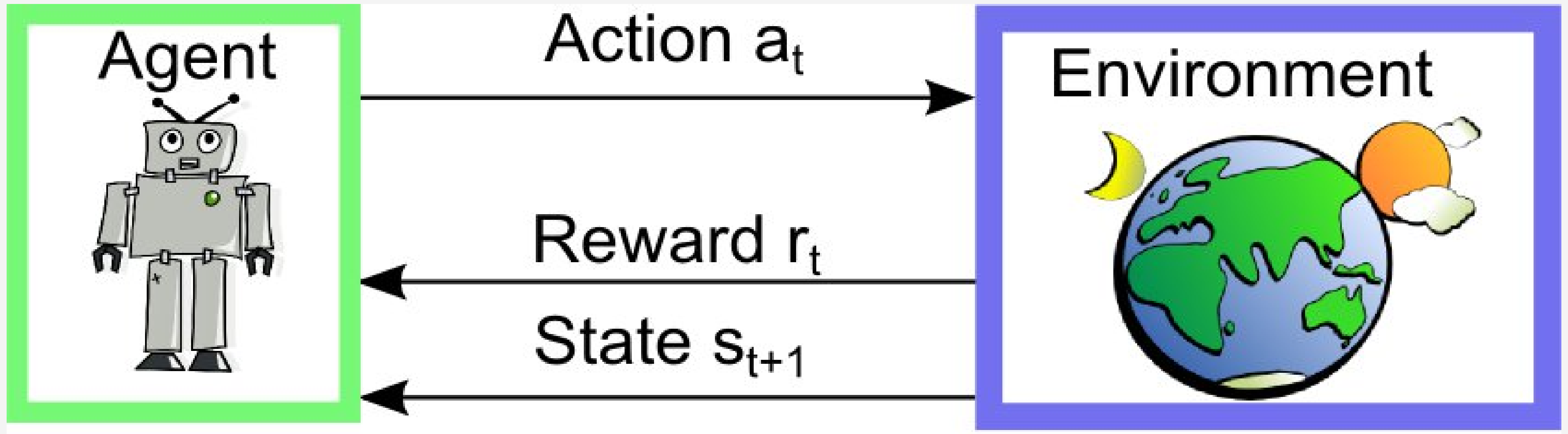


UI of Tencent Jizhi

4 Case Studies

Appliance on Game AI

- a Dota2-like MOBA Game
- Reinforcement Learning



Reinforcement Learning Setup

Appliance on Game AI

- 128k batch size for 5v5 on 128 GPUs
- Exceeding OpenAI (1M vs ~560K rounds/day) on the same number of GPUs

Game Type	GPU	Batch size	Speedup
5v5	Baseline 8 GPUs	8k	--
	Jizhi 128 GPUs	128K	13.6

Appliance on Automatic Speech Recognition

- LSTM model/ DNN model
- Super Large-scale Dataset
  - 100 thousand or even hundreds of thousands hours corpus
  - Larger than 10 TB
- More Than 3 months/epoch on 4 * Tesla M40

Appliance on Automatic Speech Recognition

- Training time reduced from more than 3 months to 20 hours

	samples/s	1 epoch	Speedup
Baseline 4 GPU	210	2194h	--
Jizhi 4 GPU	956	482h	4.55
Jizhi 120 GPU	753	20h	107

5 Problems and Countermeasures

Node affinity is not always satisfied

- The assigned nodes are not always under the same switch
- Imbalance Bandwidth between different nodes

How to maximize performance in this case?

Counter measure

- *Reducing bandwidth requirements by*
  - *Asynchronous training algorithm*
    - *Such as BMUF(Kai et al. 2016)[1]*
  - *Gradient compression algorithm*
    - *Such as Deep Gradient Compression (Yujun Lin et al. 2017)[2]*

Algorithm 1: BlockMomentumSGD with Nesterov Block Momentum:

Input:
- The initial model w_0;
- Training data with labels $\mathcal{X}$;
- Block momentum η_B and learning rate ε_B ;
- Synchronization period n ;
- Number of workers K ;

Initialization: $v_0 = 0$

for $t = 1, \ldots, T$ **do**
- **for** $k \in 1, \ldots, K$ ***parallel*** **do**
  1. Initialize the local models: $w_0^{(k)} = w_{t-1} - \eta_B v_{t-1}$
  2. Update local models using SGD:

     for $\tau = 1, \ldots, n$ **do**
     - Draw a mini-batch from $\mathcal{X}$;
     - Calculate gradient on the current mini-batch;
     - (optionally) Additional gradient processing, e.g., SGD momentum or adagrad
     - Update model parameters to $w_\tau^{(k)}$

     end
  3. Send block gradient $g_k = w_0^{(k)} - w_n^{(k)}$ to the master;

  end
- Aggregate and filter block gradients:

$$v_t = \eta_B v_{t-1} + (1 - \eta_B)\varepsilon_B \sum_k g_k$$

$$w_t = w_{t-1} - v_t$$

end

[1] https://www.microsoft.com/en-us/research/publication/scalable-training-deep-learning-machines-incremental-block-training-intra-block-parallel-optimization-blockwise-model-update-filtering/
[2] https://arxiv.org/abs/1712.01887

Hyper-parameter tuning can be expensive

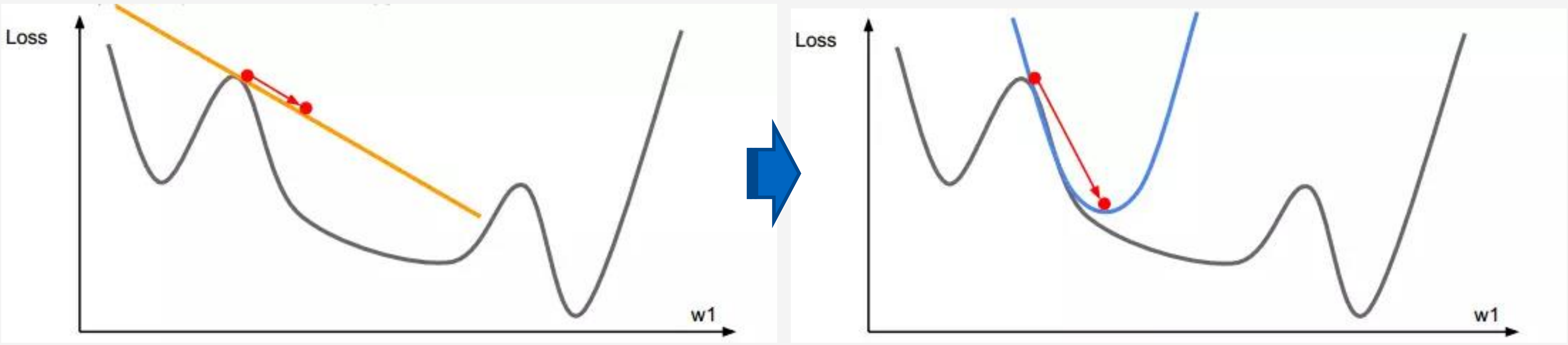
- Too many tunable parameters
- Training with one hyper-parameter set takes too long (e.g. ASR)
- Limited budget for GPU time

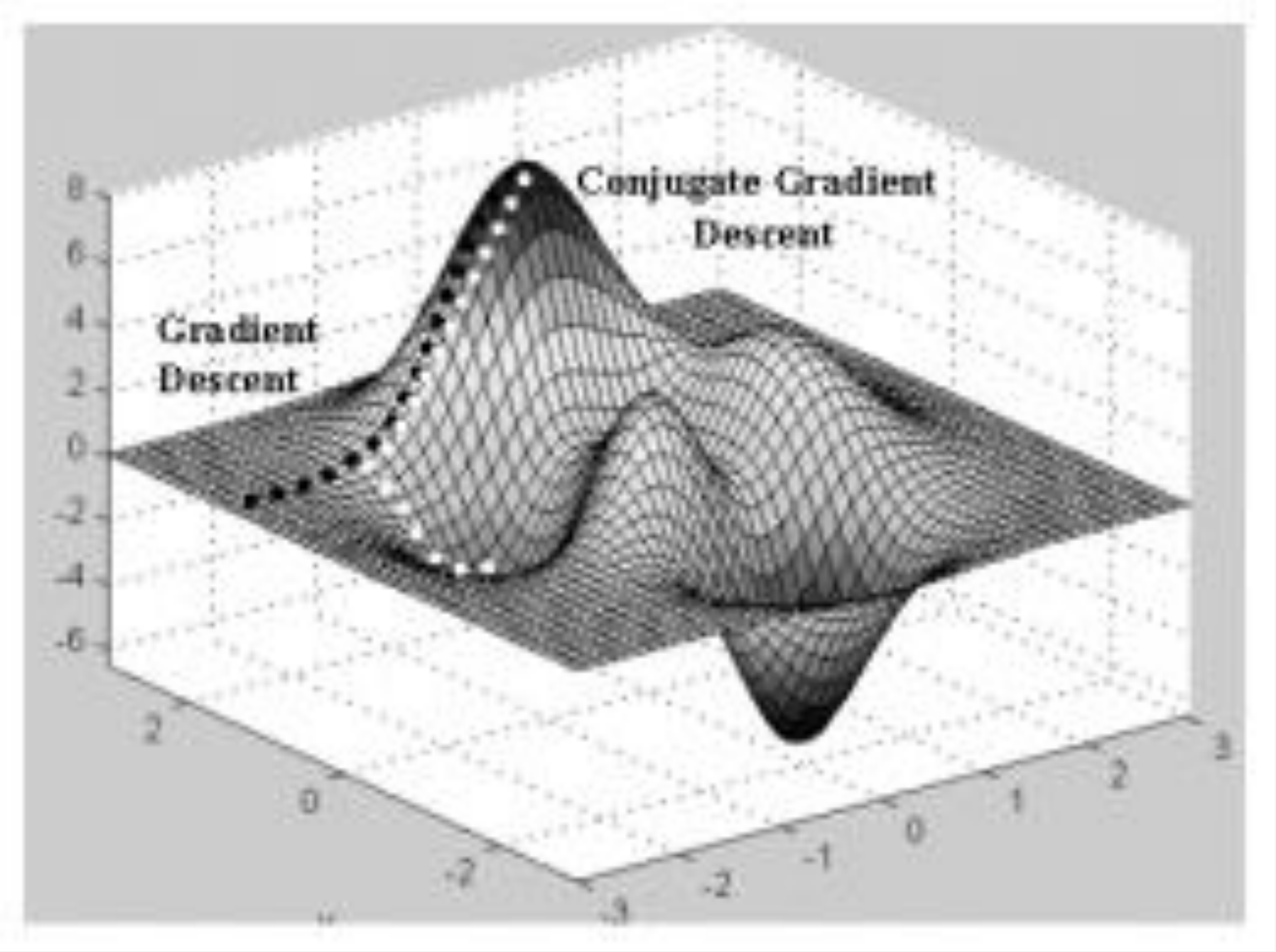
Counter measure

- *Second-order Optimization*
  - *Hessian-free algorithm(James et al. 2012)[1]*
  - *Distributed K-FAC (Jimmy et al. 2017)[2]*

Algorithm 1 The Hessian-free optimization method

for $n = 1, 2, \ldots$ **do**
  $g_n \leftarrow \nabla f(\theta_n)$
  compute/adjust λ by some method
  define the function $B_n(d) = \mathbf{H}(\theta_n)d + \lambda d$
  $p_n \leftarrow$ CG-Minimize$(B_n, -g_n)$
  $\theta_{n+1} \leftarrow \theta_n + p_n$
end for





[1] http://www.cs.toronto.edu/~jmartens/docs/HF_book_chapter.pdf
[2] https://jimmylba.github.io/papers/nsync.pdf

Imbalance Computing Resource

- Computing resource should be evenly distributed across batches
- But sometimes there is not enough GPUs with the same type in pool

Counter measure

- *Asynchronous Decentralized Training Algorithm*
  - *Such as AD-PSGD(Xiangru et al. 2018)* [1]

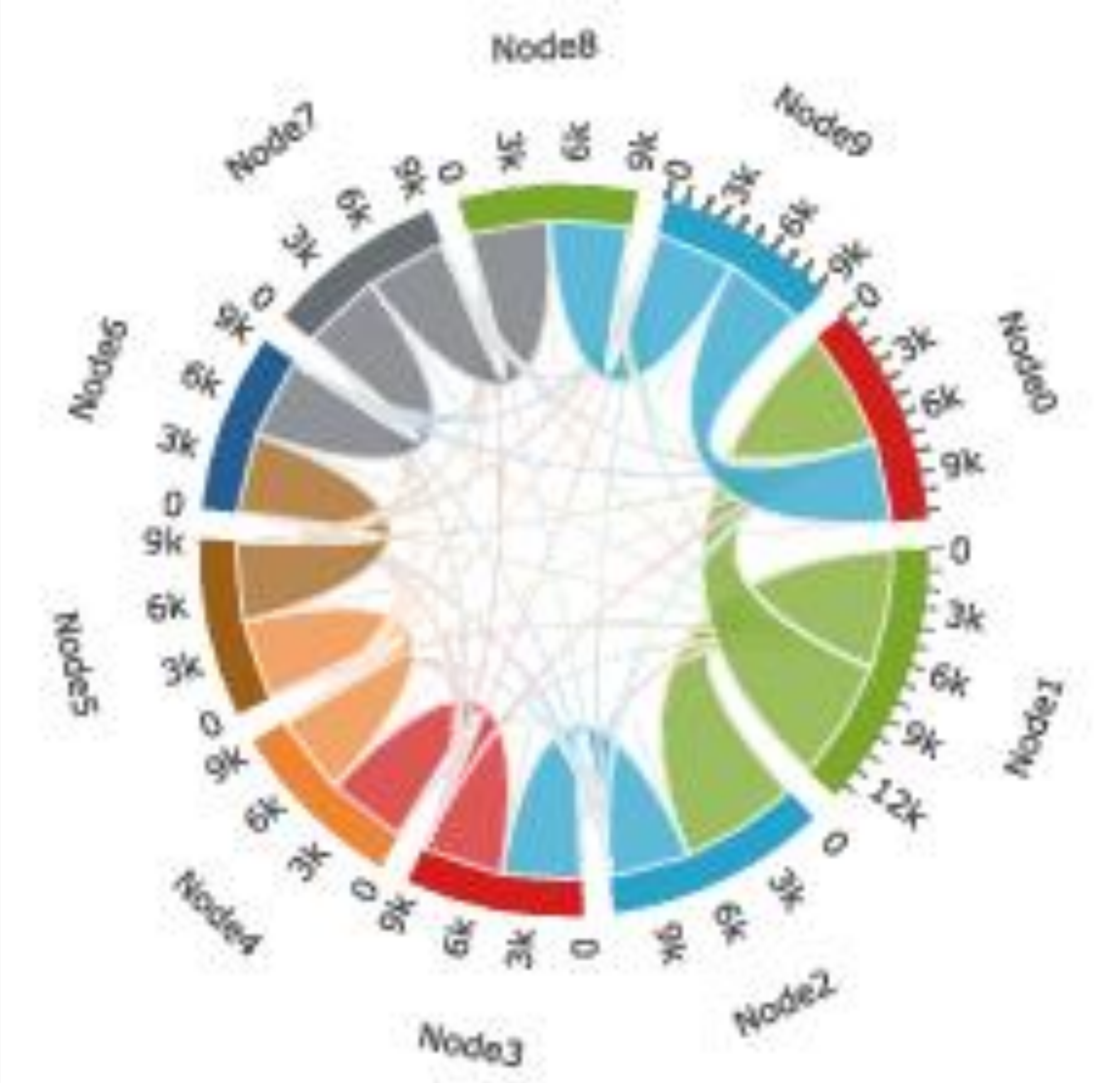
Algorithm 1 AD-PSGD (logical view)

b

Require: Initialize local models $\{x_0^i\}_{i=1}^n$ with the same initialization, learning rate γ, batch size M, and total number of iterations K.

1: **for** $k = 0, 1, \ldots, K-1$ **do**
2: Randomly sample a worker i_k of the graph G and randomly sample an averaging matrix W_k which can be dependent on i_k.
3: Randomly sample a batch $\xi_{k,i_k} := (\xi_{k,1}^{i_k}, \xi_{k,2}^{i_k}, \ldots, \xi_{k,M}^{i_k})$ from local data of the i_k-th worker.
4: Compute the stochastic gradient locally $g_k(\hat{x}_k^{i_k}; \xi_k^{i_k}) := \sum_{j=1}^{M} \nabla F(\hat{x}_k^{i_k}; \xi_{k,j}^{i_k})$.
5: Average local modes by [a] $[x_{k+1/2}^1, x_{k+1/2}^2, \ldots, x_{k+1/2}^n] \leftarrow [x_k^1, x_k^2, \ldots, x_k^n] W_k$
6: Update the local model $x_{k+1}^{i_k} \leftarrow x_{k+1/2}^{i_k} - \gamma g_k(\hat{x}_k^{i_k}; \xi_k^{i_k})$ and $x_{k+1}^j \leftarrow x_{k+1/2}^j, \forall j \neq i_k$.
7: **end for**
8: Output the average of the models on all workers.

[a] Note that Line 4 and Line 5 can run in parallel.



[1] https://arxiv.org/abs/1710.06952

Q&A

Thanks