Accelerating Distributed DNN Training with BytePS

Chuanxiong Guo

ByteDance

HPC Interconnects Forum, HPC China
September 30 2020

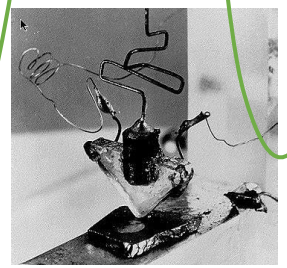
Outline

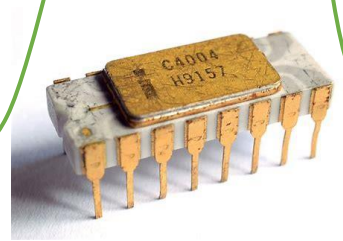
- DNN Background
- DNN Training Acceleration with BytePS
  - RDMA
  - ByteDance Tensor Scheduler: ByteScheduler
  - Summation Service
- Summary

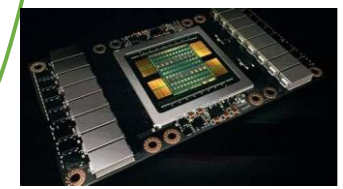
Application

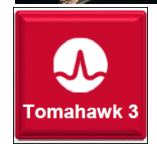
PSTN Cellular TCP/IP Windows Linux Web Browser Mobile Internet Cloud Computing *ML systems*

System



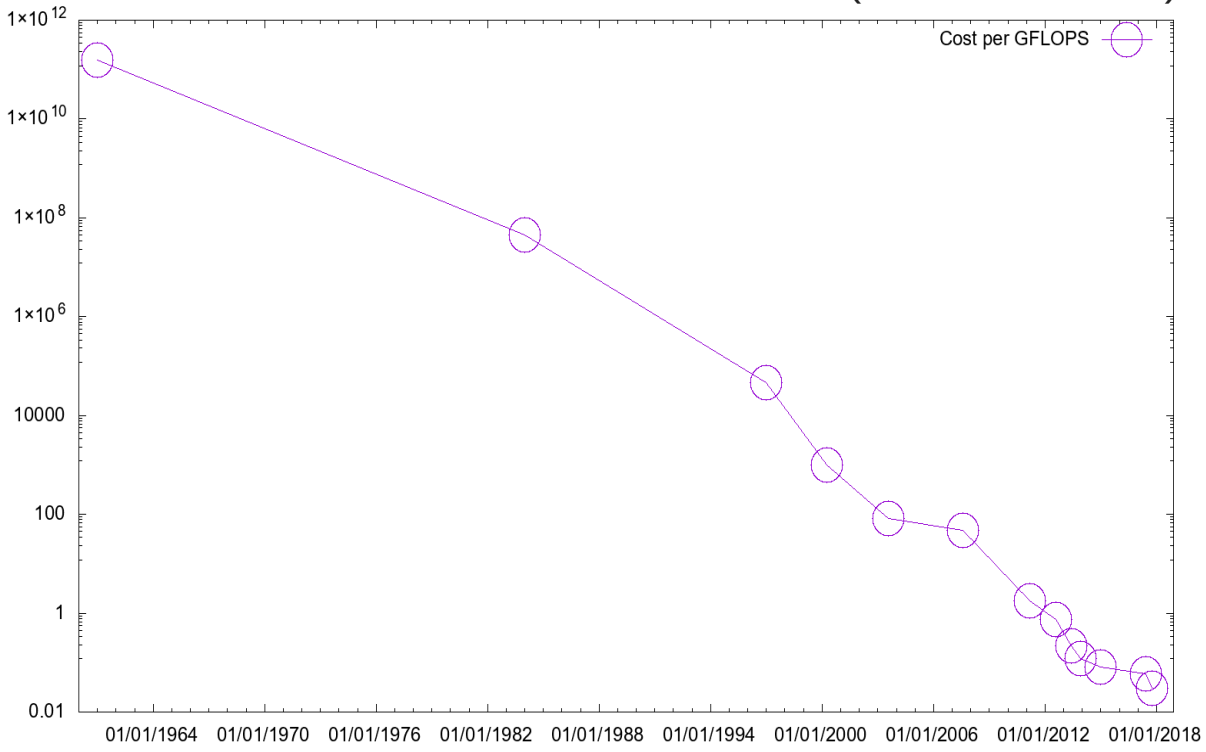


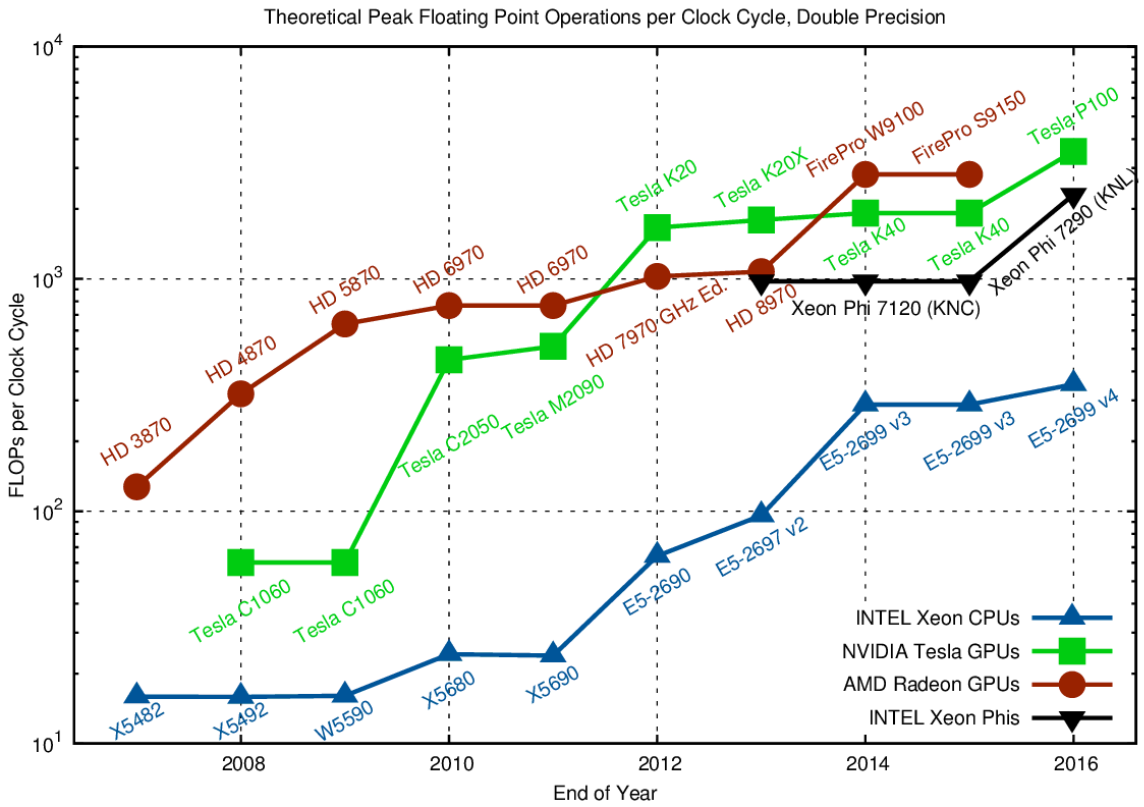




Hardware

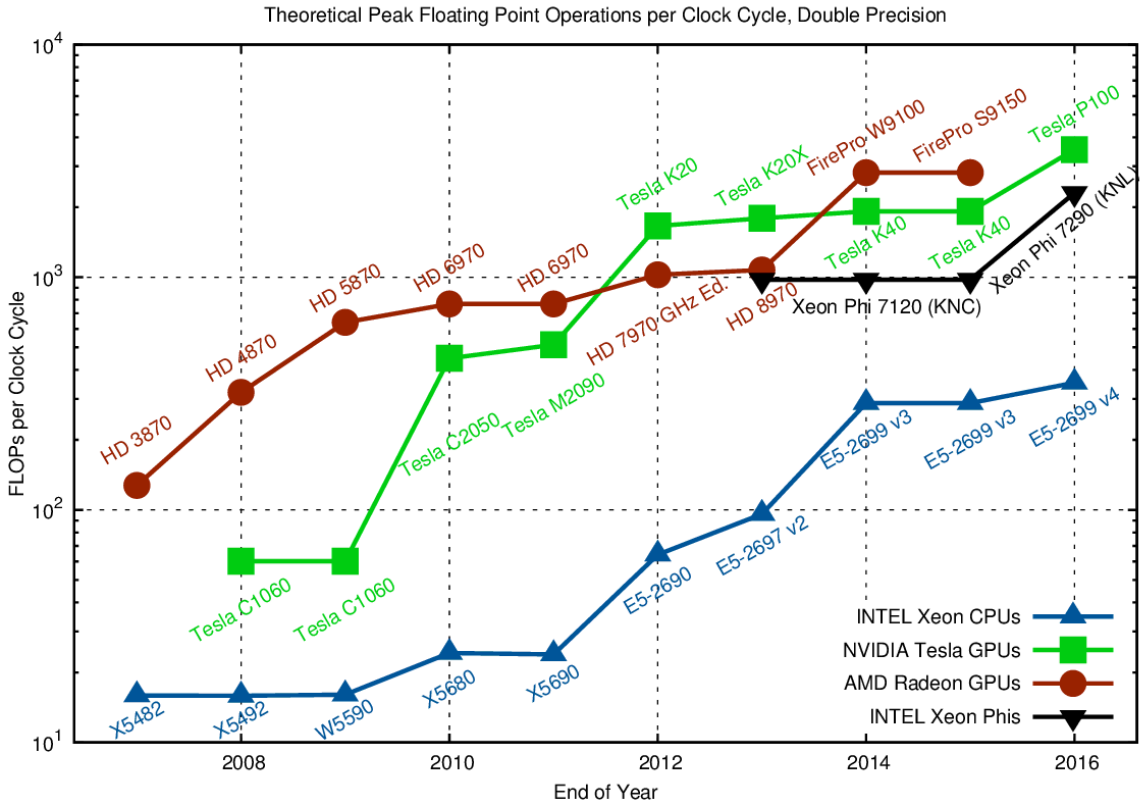
(2018 US dollars)



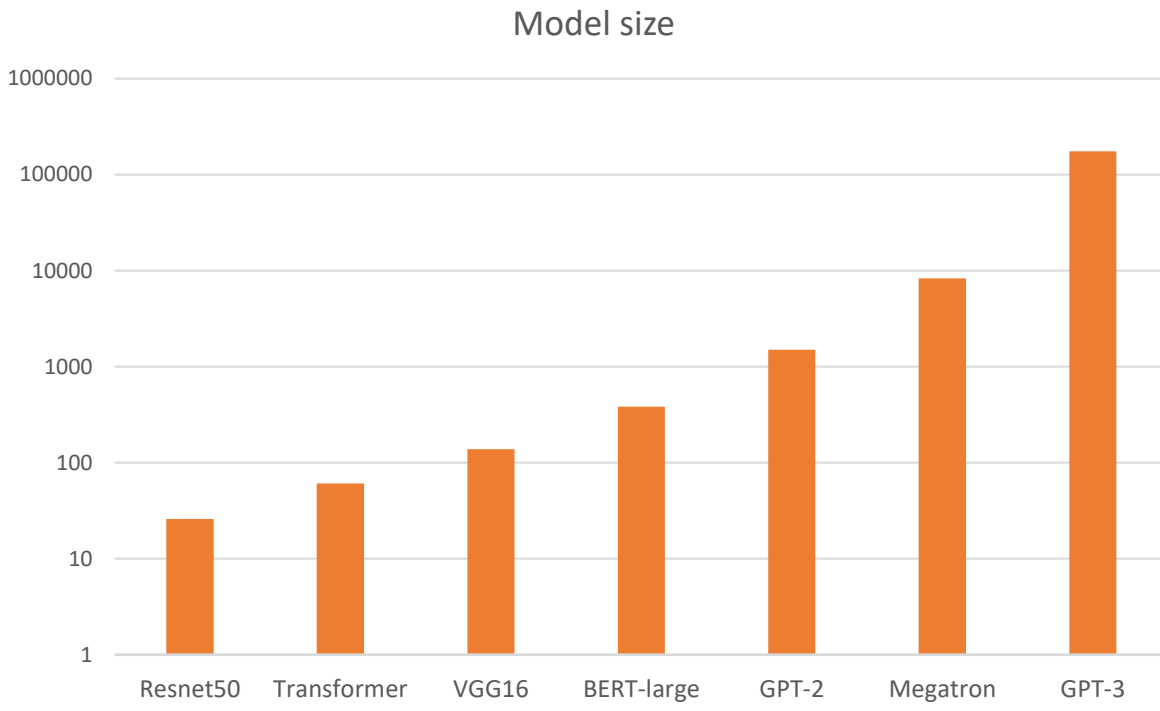


https://en.wikipedia.org/wiki/FLOPS

https://www.karlrupp.net/2016/08/flops-per-cycle-for-cpus-gpus-and-xeon-phis/

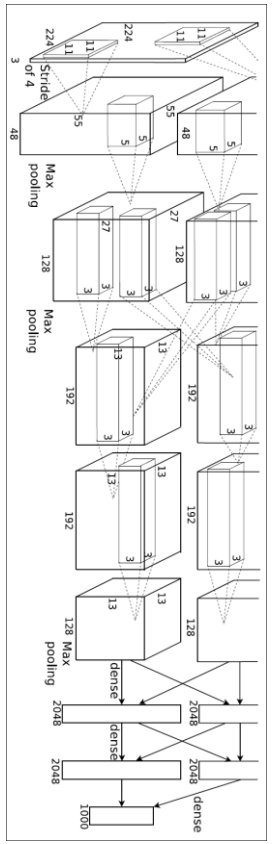


10 Years

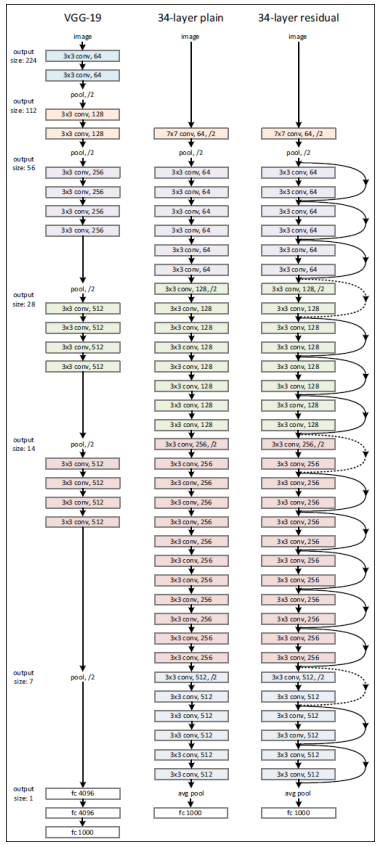


5 Years!

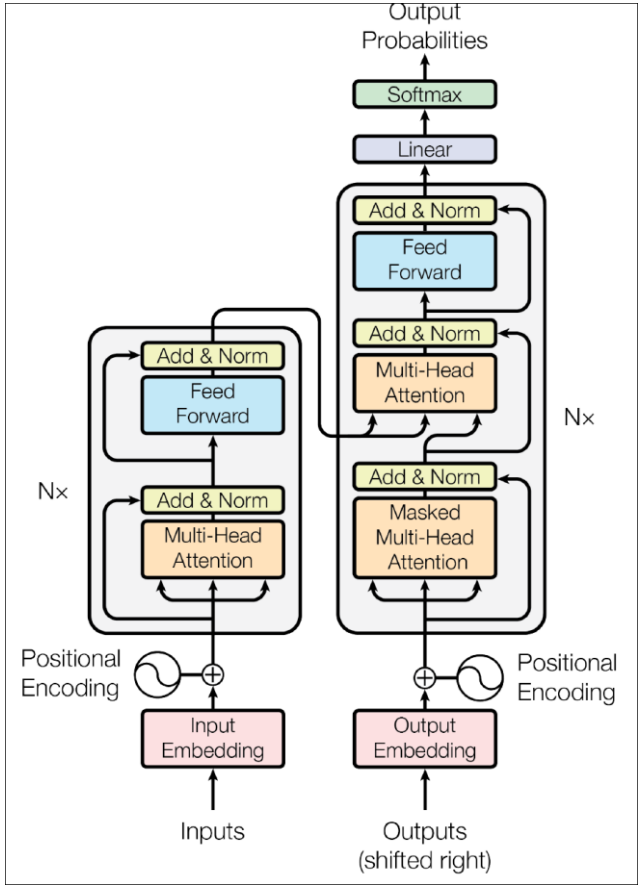
DNN Models



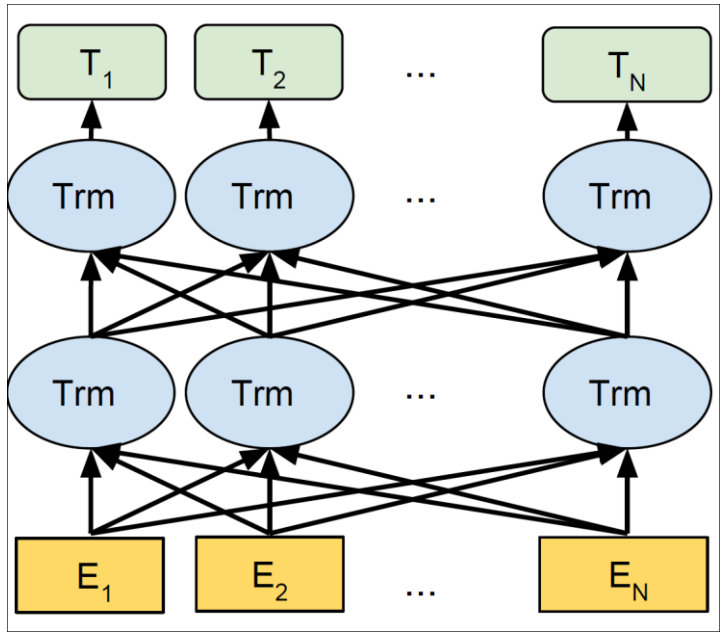
AlexNet



VGG ResNet



Transformer



Bert pre-training

DNN Models

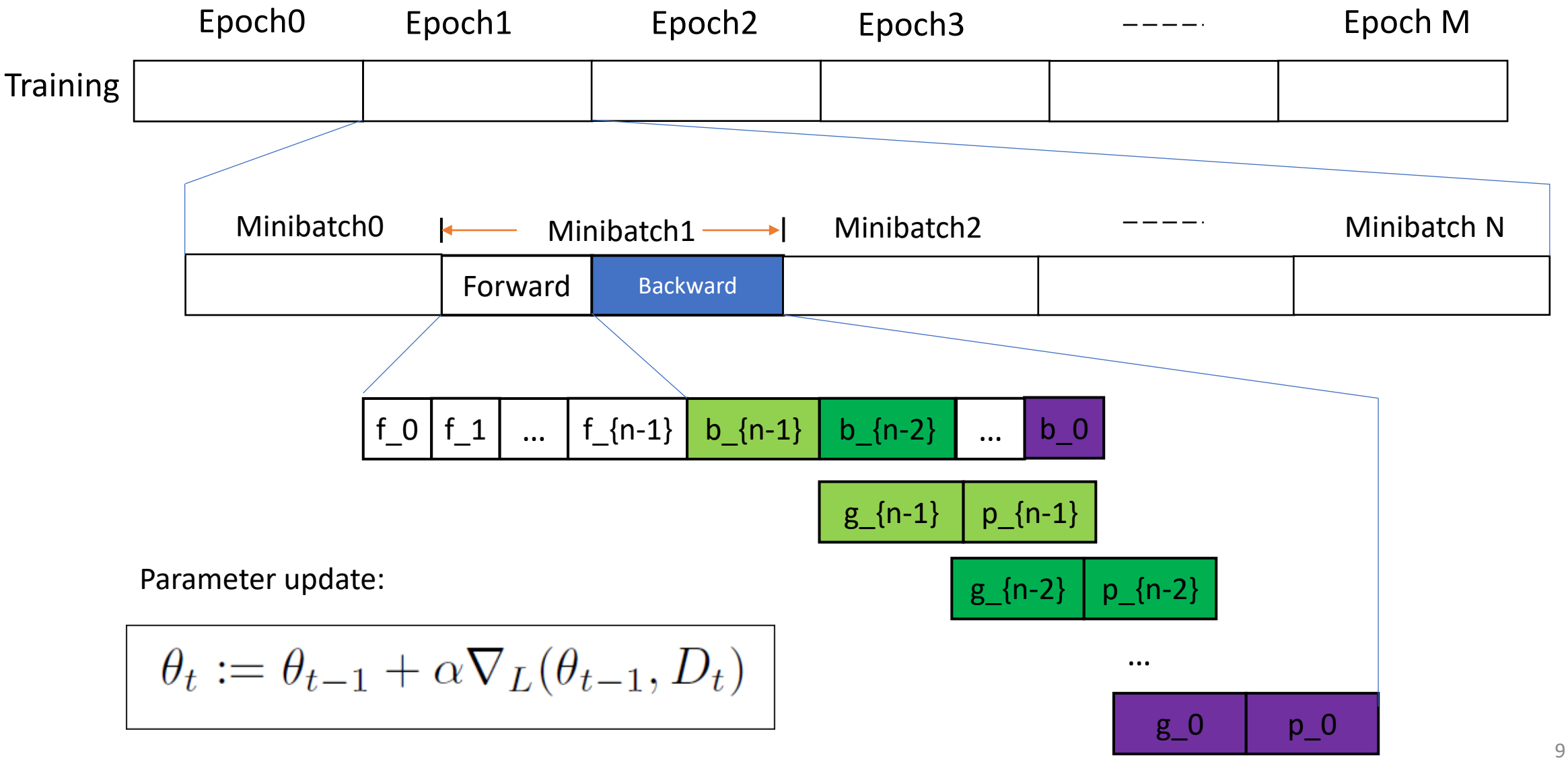
Model (dataset)	Resnet50(imagenet)	VGG16(imagenet)	Transformer(WMT17)	Bert base	Bert large
Model param# (M)	26	138	61	148	387
GFlops FP (batch size 1)	3.9	16	37	52	170
GFlops BP (batch size 1)	7.8	32	74	97	324
Time for FP (ms)	7 (batch1) 32 (batch32)	5 (batch1) 49 (batch32)	5 (batch1) 95 (batch16)	13 (batch1) 188 (batch85)	22 (batch1) 339 (batch35)
Time for BP (ms)	14 (batch1) 64 (batch32)	11 (batch1) 100 (batch32)	14 (batch1) 114 (batch16)	42 (batch1) 281 (batch85)	102 (batch1) 508 (batch35)
Ideal comm. time (ms), 100GbE	8.32	44.6	19.52	23.68	61.92
#Iter. to converge	450k(total batch size 256)	370k (total batch size 256)	100k (total batch size 4096)	125k (64 V100)	125k (64 V100)
Data shuffled for convergence (TB)	46.8 (fp32)	204 (fp32)	24 (fp16)	37 (fp16)	97 (fp16)

DNN Models

Model (dataset)	Resnet50(imagenet)	VGG16(imagenet)	Transformer(WMT17)	Bert base	Bert large
Model param# (M)	26	138	61	148	387
GFlops FP (batch size 1)	3.9	16	37	52	170
Ideal comm. time (ms), 100GbE	8.32	44.6	19.52	23.68	61.92
#Iter. to converge	450k(total batch size 256)	370k (total batch size 256)	100k (total batch size 4096)	125k (64 V100)	125k (64 V100)
Data shuffled for convergence (TB)	46.8 (fp32)	204 (fp32)	24 (fp16)	37 (fp16)	97 (fp16)

- Training is a mechanical, iterative process
- Models need more and more computational power
- DNN training needs to shuffle huge amount of data

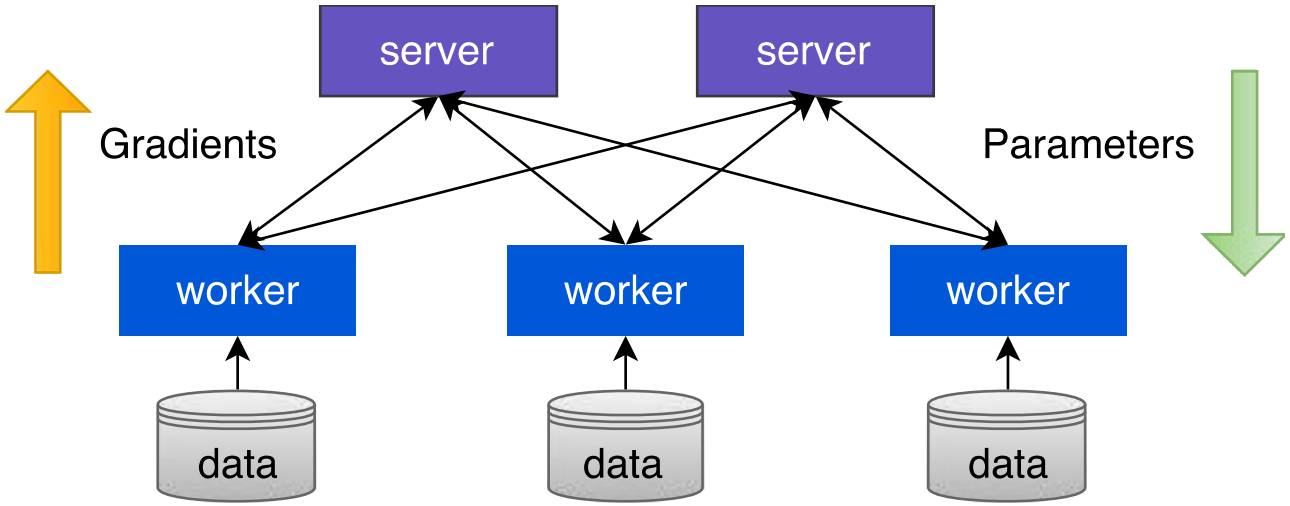
Back Prop based DNN Training



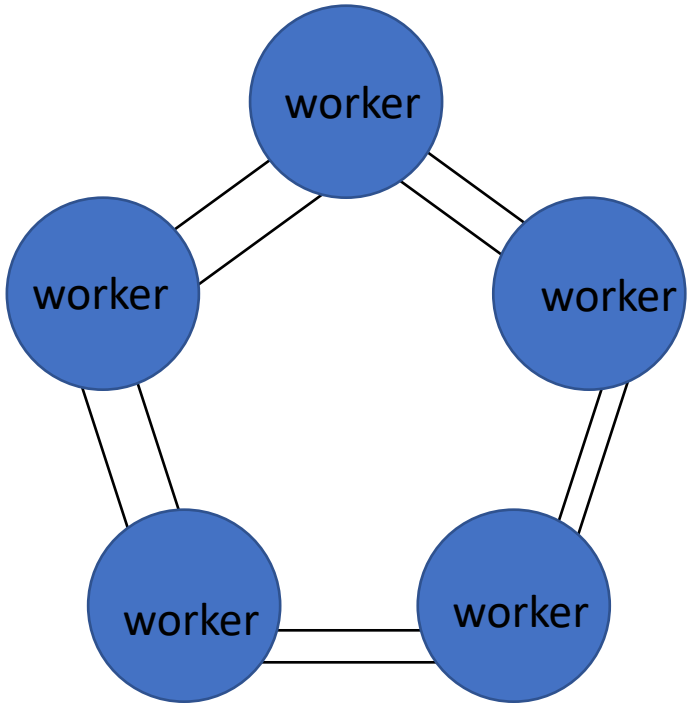
Parameter update:

$$\theta_t := \theta_{t-1} + \alpha \nabla_L(\theta_{t-1}, D_t)$$

Distributed DNN Training



Parameter Server



All-reduce

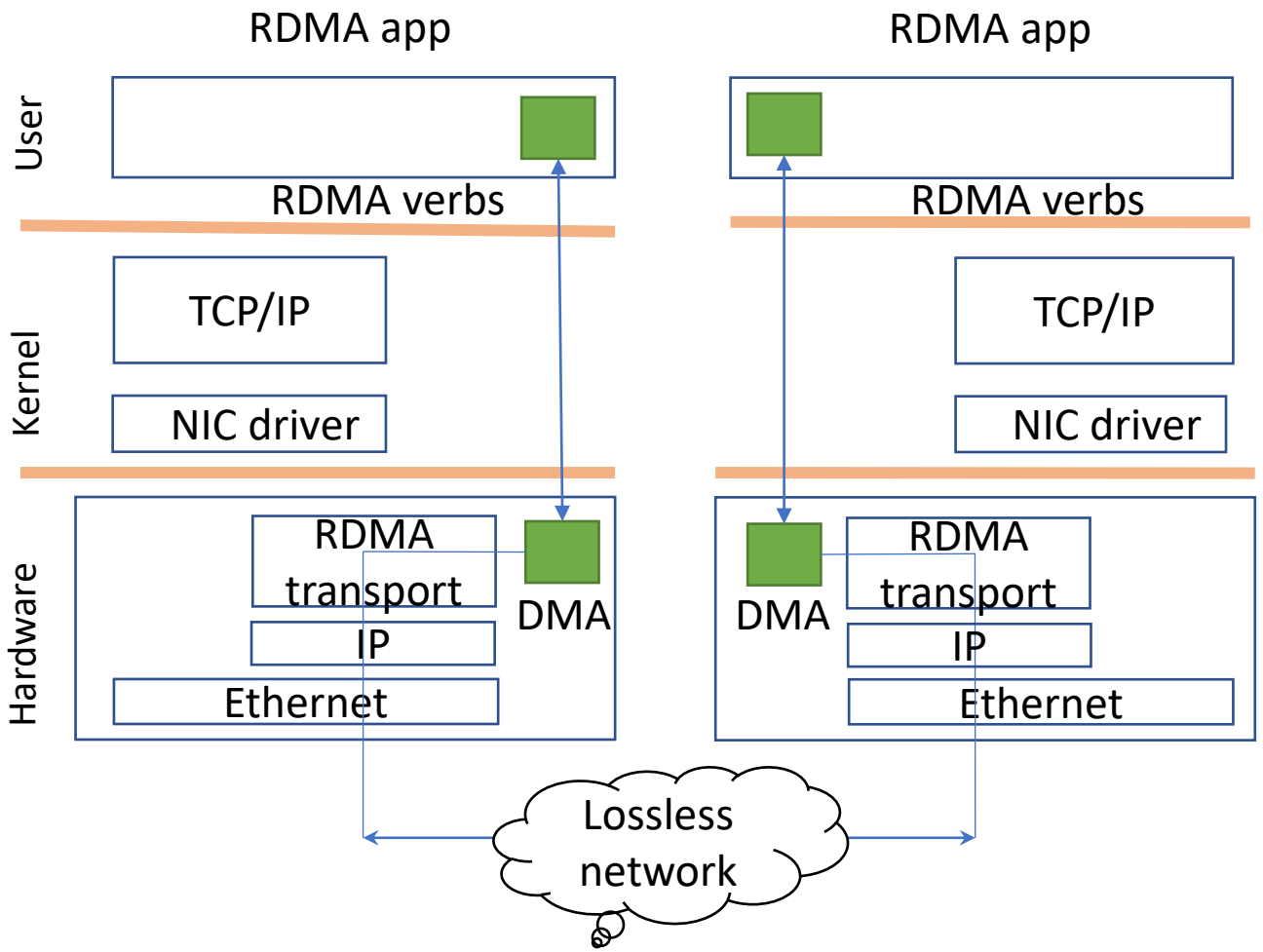
Outline

- DNN Background
- DNN Training Acceleration with BytePS
  - *RDMA*
  - ByteScheduler
  - Summation Service
- Summary

RDMA

- Remote Direct Memory Access (RDMA): Method of accessing memory on a remote system ***without*** interrupting the processing of the CPU(s) on that system
- RDMA offloads packet processing protocols to the NIC
- IBVerbs/NetDirect for RDMA programing
- RDMA in Ethernet based data centers (RoCEv2)

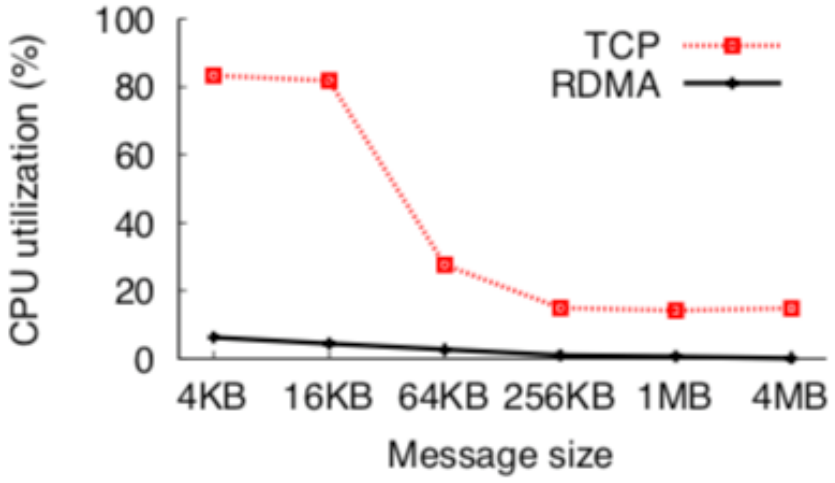
RoCEv2: RDMA over Commodity Ethernet



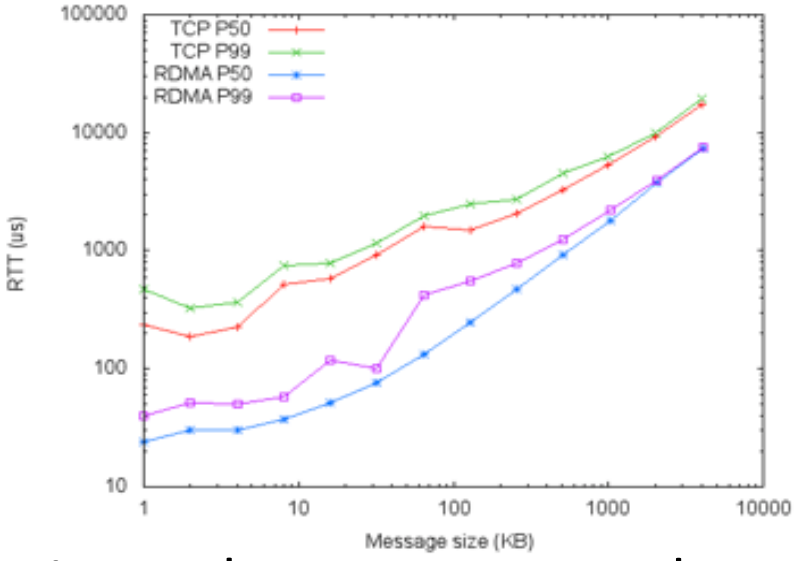
- RoCEv2 for Ethernet based data centers
- DSCP-based PFC for lossless networking
- DCQCN for flow-based congestion control
- Huge engineering efforts to make it scalable and safe

Using RDMA for Communication

- RDMA NIC Offloading: ~0% CPU usage
- RDMA Latency benefits (lower latency than TCP)
  - Kernel bypassing
  - Fast start / Lossless networking / ACK consolidation



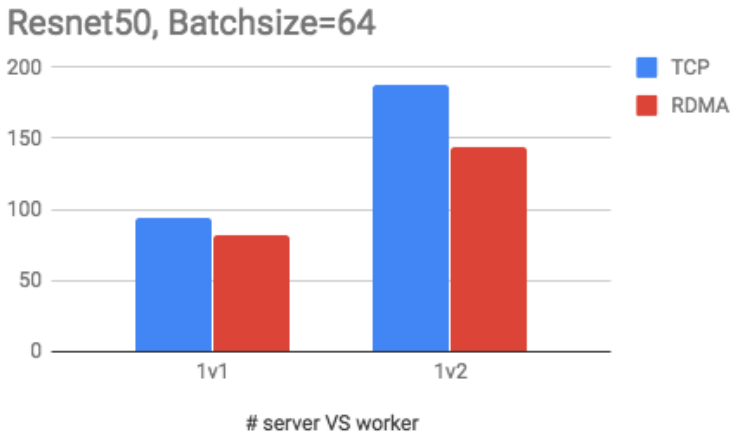
~0% CPU Usage

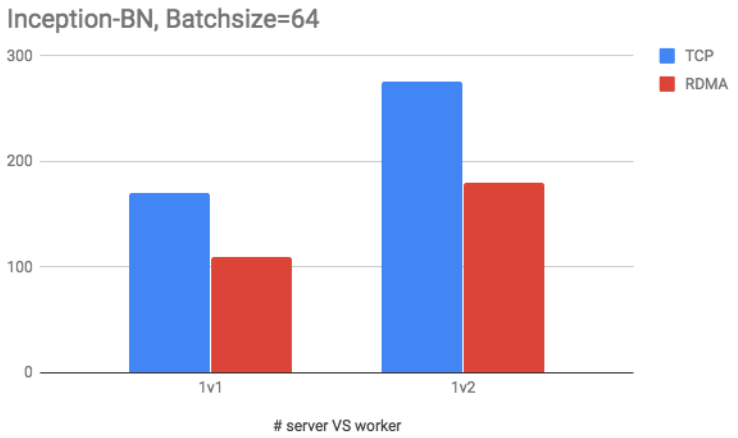


Lower latency compared to TCP

Vanilla RDMA Does Not Perform Well

- For large models, vanilla RDMA outperforms TCP
  - VGG19 has some large tensors (hundreds of MB), making the overhead not evident
- For small models, vanilla RDMA is worse than TCP
  - Resnet50 and Inception-BN have a lot of small tensors (most of them no more than 1MB)
  - The overhead is larger than the benefit that RDMA brings





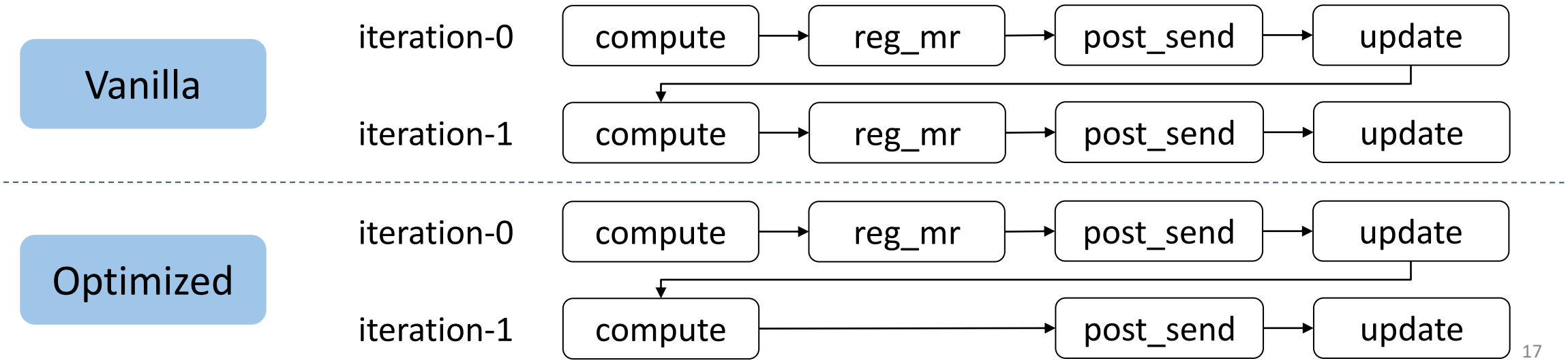
Optimizing RDMA for DNN Training

Our optimizations on RDMA performance

- Tensor buffer memory reuse
- One-way RDMA write
- Zero-copy RDMA data path
- Address the Slow-Receiver Symptom

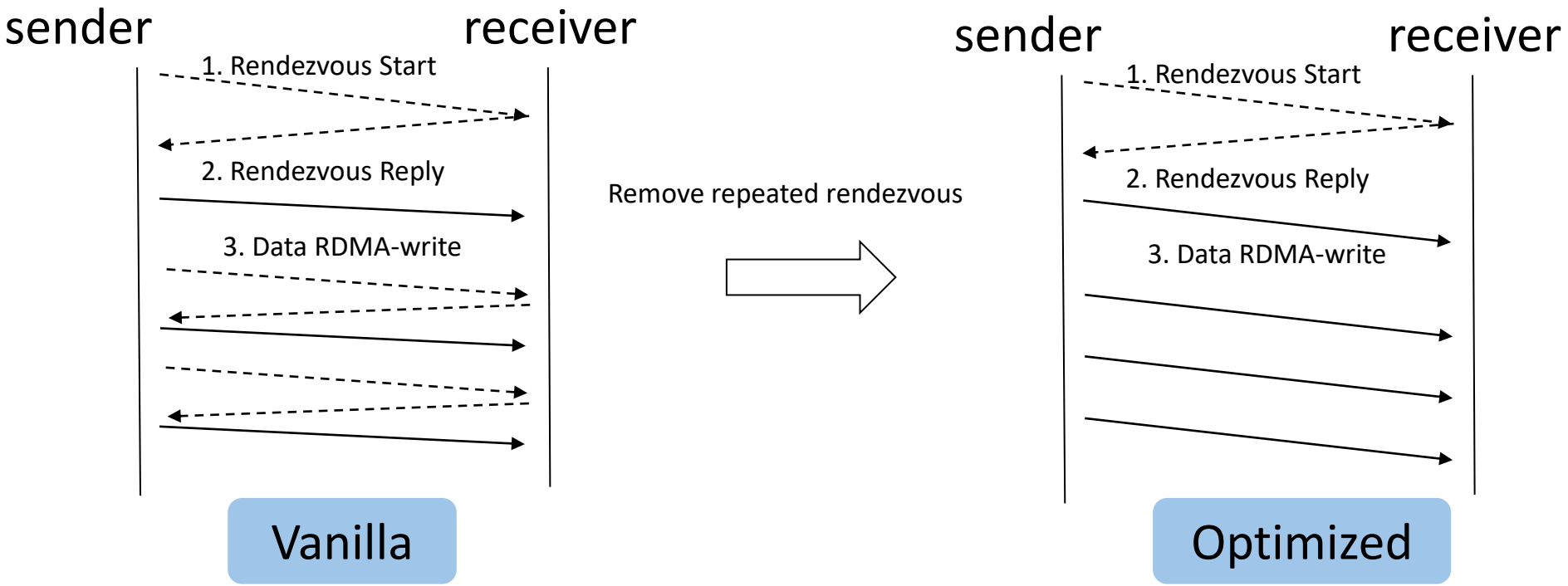
Tensor Buffer Memory Reuse

- RDMA message needs to be registered before sent
- Reuse the tensor memory region
  - i.e., the sending buffer memory address can be reused
  - we only need to do *reg_mr* once in the first training iteration, and reuse the memory in all the following iterations
  - can benefit frameworks that decouple memory allocation with its communication library (e.g., MXNet & ps-lite)



One-way RDMA-write

- Reduce redundant rendezvous latency
  - Rendezvous: get the remote buffer address to write the data
  - With memory reuse, we only need Rendezvous once, and remove redundant rendezvous because the remote memory address does not change



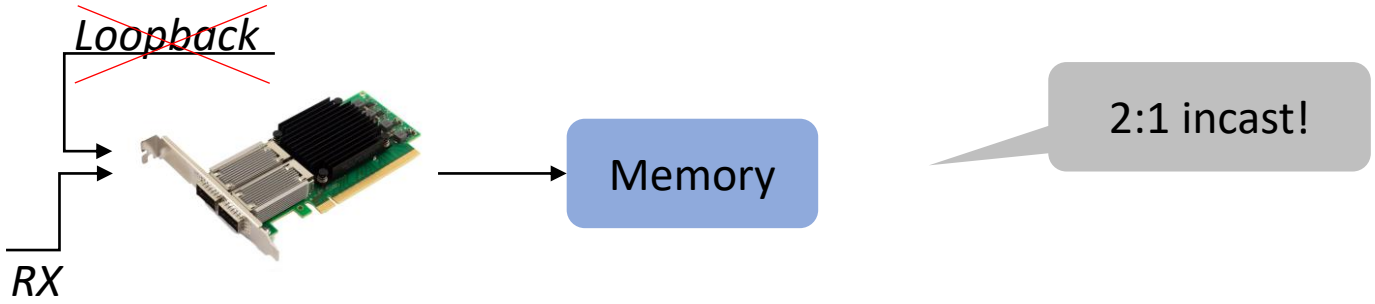
Zero-copy RDMA Data Path

- There exists several memory copy ops on the data path
  - Worker sends push requests to server, server copies the result to CPU buffer
  - Server sends pull responses to worker, worker copies to CPU buffer
- Memory copy decreases performance
  - Single-threaded memcpy bandwidth: ~40Gbps
  - Memcpy significantly decreases RDMA bandwidth utilization
- We remove all memcpy on RDMA data-path, and achieve zero-copy communication

Address the Slow-Receiver Symptom

Problem 1: RDMA loopback traffic creates internal incast

Use shared memory to eliminate loopback traffic



Problem 2: the DMA write can be slow

- Use *page-aligned* buffers to reduce the # pages to be written

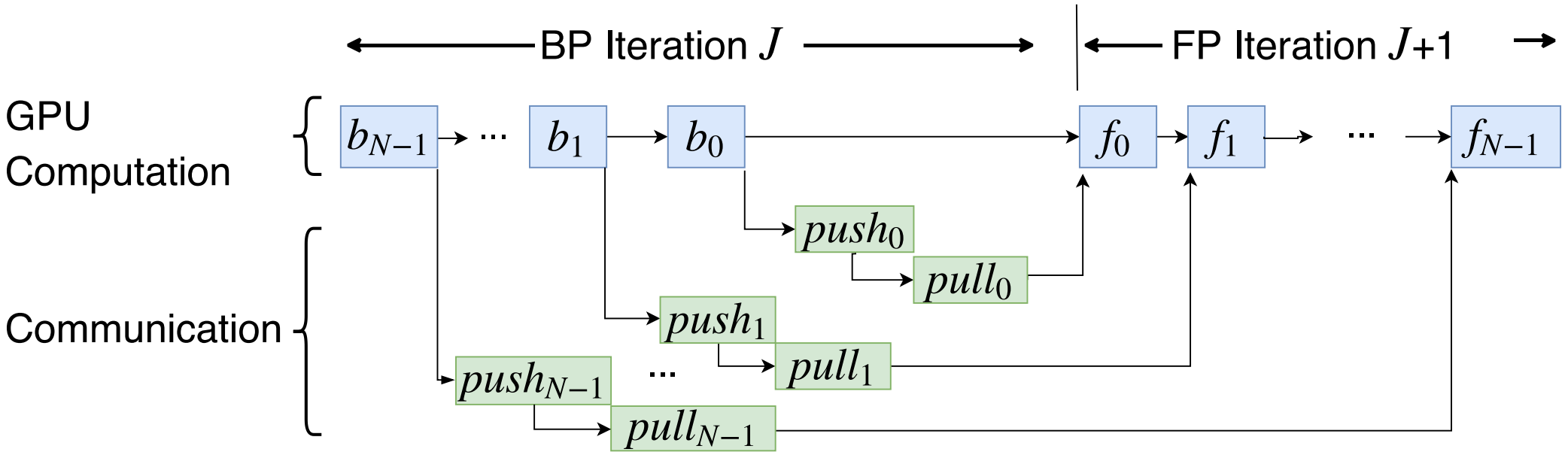
Problem 3: the RX can be impacted by the concurrent TX (not duplex)

- Without TX, the RX works well; With TX, the RX cannot reach line rate
- Use *page-aligned* sending buffers and enforce only one *SGE* per RDMA write

Outline

- DNN Background
- DNN Training Acceleration with BytePS
  - RDMA
  - *ByteScheduler*
  - Summation Service
- Summary

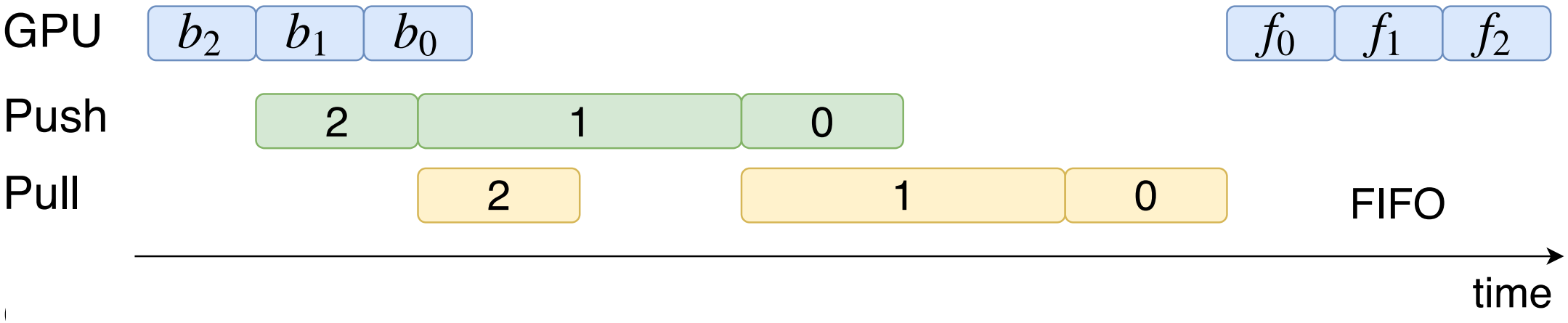
Dependency Graph



Dependency:

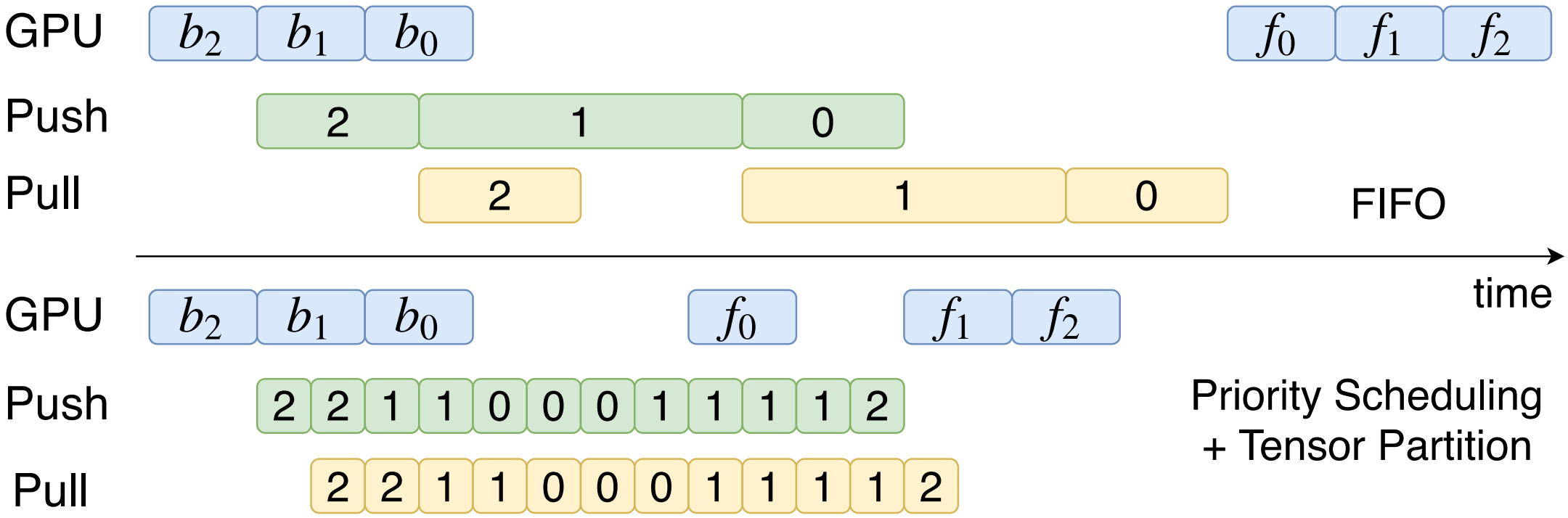
- Backward depends on forward
- Push depends on backward
- Pull depends on push
- Forward depends on pull

Scheduling the Tensor Transmission Order



- ML framework executes communication operations in FIFO order
- Problem: FIFO strategy delays push and pull of layer 0 and hence delays the start of next iteration
- *Priority scheduling*: layer i with higher priority than j for i < j

Priority Scheduling and Tensor Partitioning



- In this example, priority scheduling and tensor partition result in 44% speedup

Priority Scheduling and Tensor Partitioning

- Tensor partitioning reduces an NP-hard problem to an easy problem
- Tensor partitioning was first introduced in P3 and TicTac

Theorem 1. *The following priority queuing scheduler:*

- *in a PS architecture, prioritize* $pull_i$ *over* $pull_k$*, and* $push_i$ *over* $push_k$*,* $\forall i < k$
- *in an all-reduce architecture, prioritize* $allreduce_i$ *over* $allreduce_k$*,* $\forall i < k$

is the optimal solution for minimizing the time for each training iteration, if the following conditions are met:

1. *(Sequential GPU computation operations) The subgraph of DAG containing only* f_i *and* b_i *is a chain (i.e., no parallel layers in the DNN).*
2. *(Optimal GPU scheduling) Whenever the dependencies of* f_i *or* b_i *are satisfied, GPU will run the computation operation without preemption. This is the optimal GPU scheduler because the GPU computation operations are in a chain.*
3. *(Tensor partition) Tensors in each DNN layer are partitioned, such that flow preemption can happen; with PS, if the push flow in a layer is only partially done before being preempted, the done part can be pulled.*
4. *(Infinitely small partition) Suppose the partition size is* δ *and* $\delta \to 0$*.*
5. *(Flow preemption) Higher priority push/pull or all-reduce can preempt lower priority push/pull or all-reduce immediately without extra overhead.*

ByteScheduler: One Unified Scheduler for ALL

- Dependency graphs have similar structure for different ML frameworks, DNN models and communication methods (PS or all-reduce, TCP or RDMA)
- We aim to design a *generic* scheduler, no matter which ML framework and communication method.

Challenges addressed in ByteScheduler:

- Many training frameworks: TensorFlow, PyTorch, MxNet, etc.
- Imperative engines (e.g., PyTorch) and declarative engines (e.g., TensorFlow)
- Global barrier between iterations (e.g., TensorFlow, PyTorch)

Outline

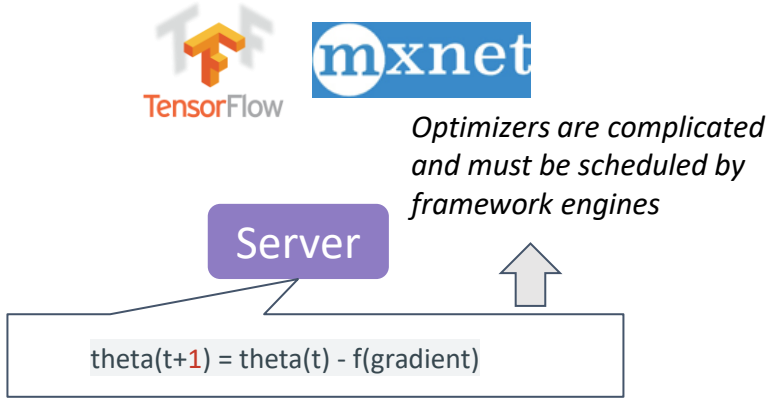
- DNN Background
- DNN Training Acceleration with BytePS
  - RDMA
  - ByteScheduler
  - *Summation Service*
- Summary

Summation Service (SS)

Why design SS, rather than re-using the conventional PS process?

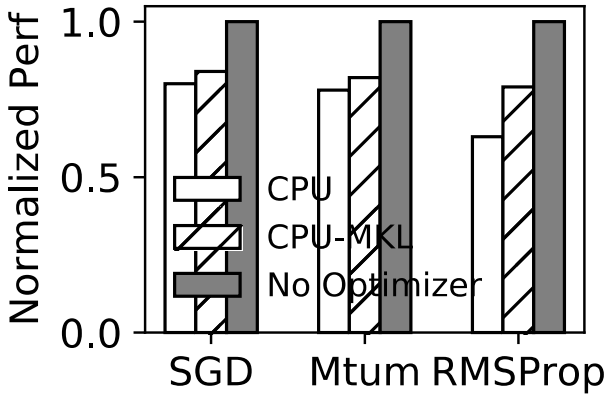
- In PS, the optimizer in placed on the servers, leading to two problems

Problem 1: Optimizer update on server requires framework-specific implementation, hindering the cross-framework requirement



f(gradient) *needs to be implemented in framework specific syntax*

Problem 2: Using CPU-based servers for optimizer update cannot match the 100 Gbps network bandwidth, causing ***CPU bottleneck***



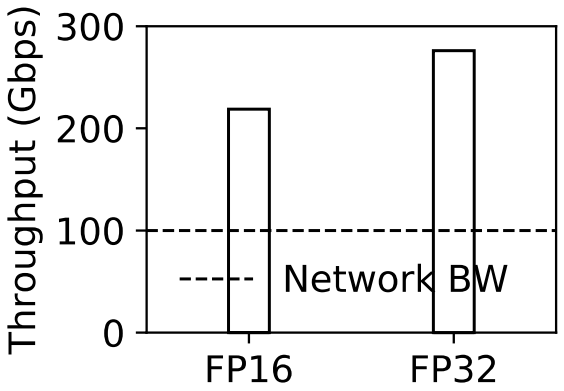
CPU for Gradient Summation

Optimizer update can be divided into *gradient summation* and *parameter update*

Though not efficient to run optimizer, CPU is good at summation

- Summation on modern x86 CPUs can be optimized using AVX instructions

Our key idea: **put optimizer update on GPU, and sum gradient on CPU**

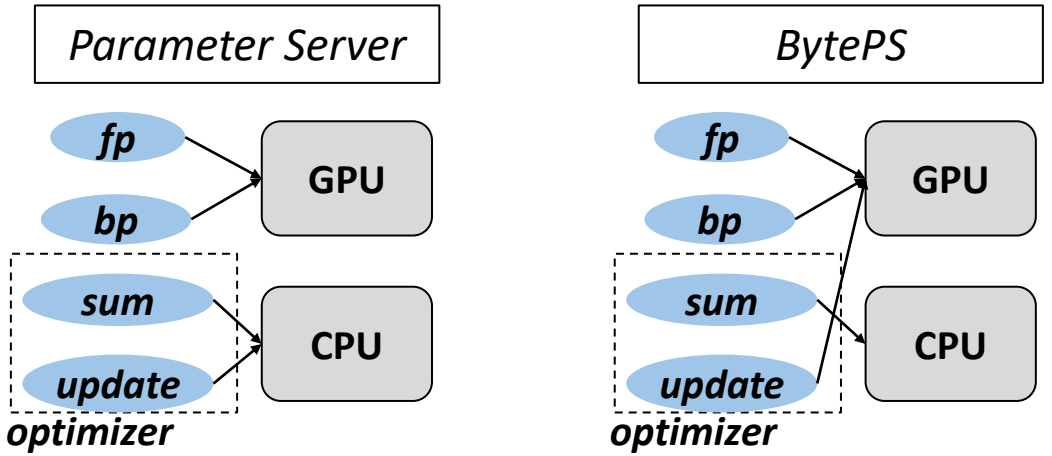


Parameter Server V.S. Summation Service

Advantages of Summation Service

- The server only sums up gradient, which is framework-independent (**generic**)
- The heavy optimizer update is now performed by GPU (**high performance**)

A high level comparison:



Evaluation

- GPU servers, each with 8 V100 GPUs and a 100Gbps NIC
- RDMA/RoCEv2 network, full bisection bandwidth
- Benchmarks: BERT-Large, ResNet, Transformer, GPT-2, VGG, UGATIT-GAN
- Baselines:
  - Naïve PS
  - All-reduce
  - BytePS w/o CPU servers
  - Linear scaling
- Metric: training speed (samples/sec or tokens/sec)

BERT-Large (TF)

Without BytePS, the entire training takes about 18-19 days!

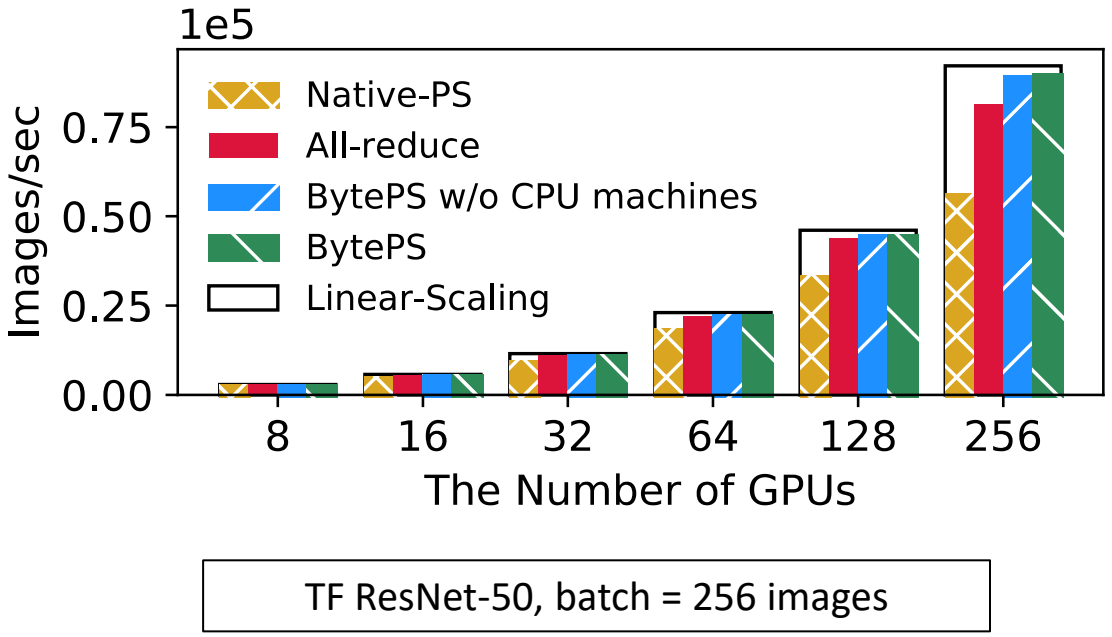
We reduce the training time to about 10 days.

steps/sec	32 GPUs	256 GPUs
Baseline (Horovod)	1.00	0.71
BytePS	1.35	1.34

Near 89%+ performance gain for 256 GPUs

Video Classification (MxNet)

ResNet-alike, image classification model



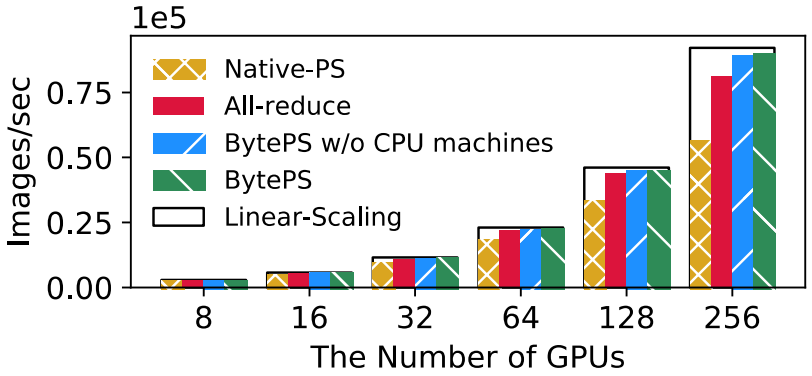
TF ResNet-50, batch = 256 images

97.5% scaling efficiency for 256 GPUs

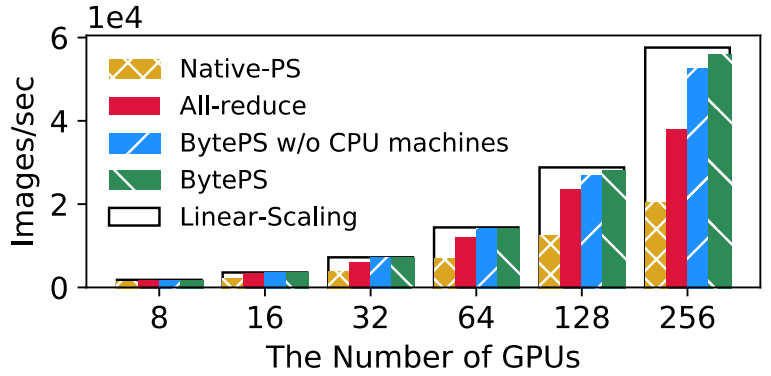
Scalability

BytePS scales well on almost all models

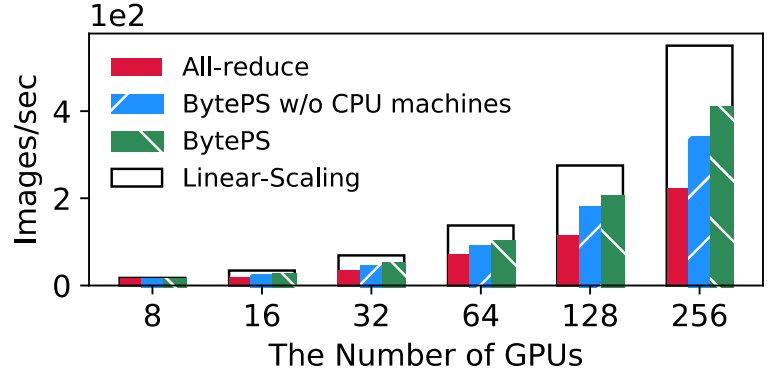
BytePS outperforms PS & All-reduce on all CV and NLP models



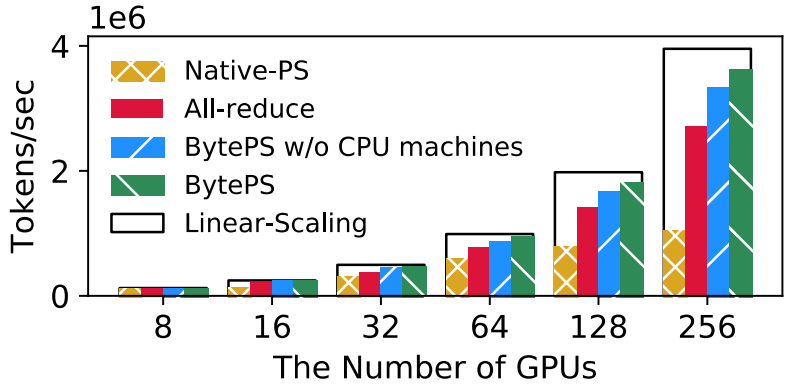
TF ResNet-50, batch = 256 images



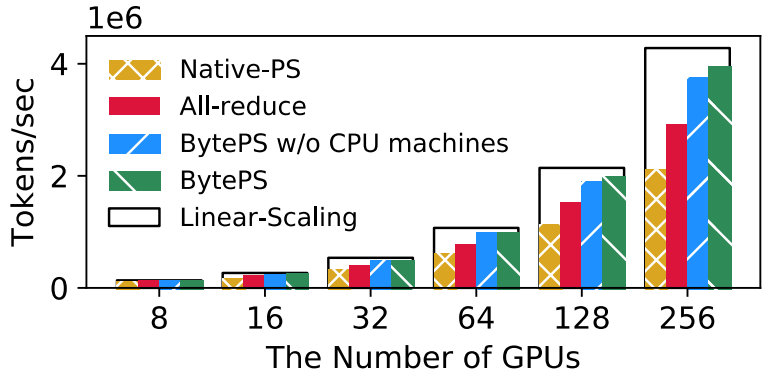
MxNet VGG-16, batch = 96 images



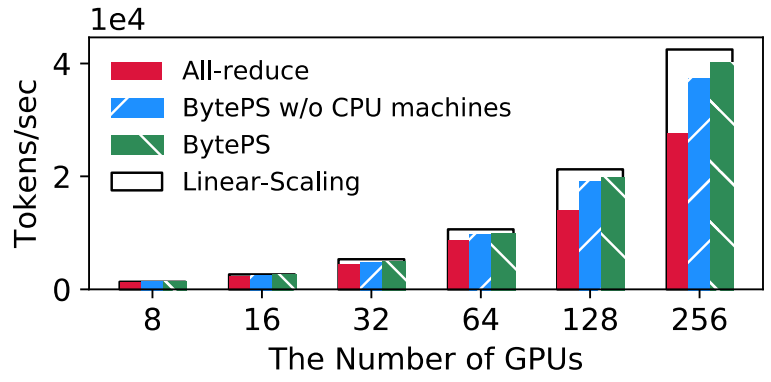
PyTorch UGATIT-GAN, batch = 2 images



TF Transformer, batch = 3072 tokens



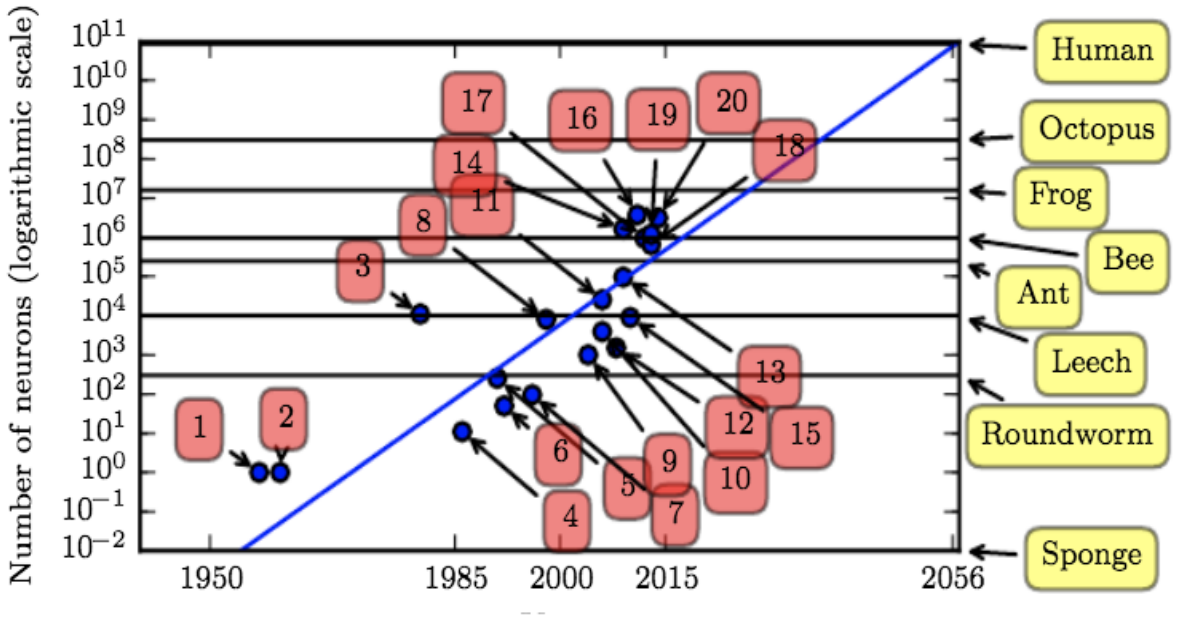
MxNet BERT-Large, batch = 8192 tokens



PyTorch GPT-2, batch = 80 tokens

Summary

- BytePS
  - RDMA for tensor communication acceleration
  - ByteScheduler: optimal communication and computation overlapped tensor scheduler
  - Summation Service: better computation partitioning
- ML systems opportunities
  - Distributed, parallel HPC
  - Scale out and up
  - Reduce cost



PSTN Cellular TCP/IP Windows Linux Web Browser Mobile Internet Cloud Computing ML system

System

Q&A

- Thanks the interns and members of the Machine Learning Systems Group at ByteDance AILab!
- We are hiring!
  - Both ByteDance Networking, and Machine Learning Systems groups
  - guochuanxiong@bytedance.com