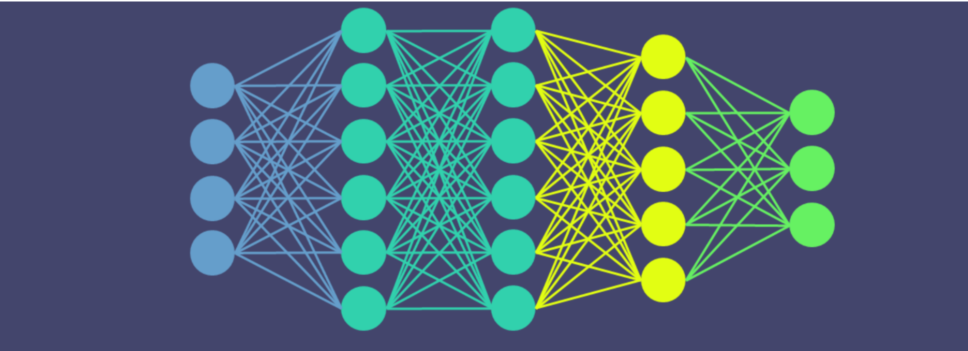
KTH ROYAL INSTITUTE
OF TECHNOLOGY

BigDL: A Distributed Deep Learning Framework for Big Data

ACM Symposium on Cloud Computing 2019 [Acceptance Rate: 24%]
Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, Jiao Wang, Shengsheng Huang, Zhongyuan Wu, Yang Wang, Yuhao Yang, Bowen She, Dongjie Shi, Qi Lu, Kai Huang, Guoqiong Song. [Intel Corporation]

Presenter: Ezgi Korkmaz



Outline

- Deep learning frameworks
- BigDL applications
- Motivation for end-to-end framework
- Drawbacks of prior approaches
- BigDL framework
- Experimental Setup and Results of BigDL framework
- Critique of the paper

Deep Learning Frameworks

- Big demand from organizations to apply deep learning to big data
- Deep learning frameworks:
  - **Torch** [2002 Collobert et al.] [C, Lua]
  - **Caffe** [2014 Berkeley BAIR] [C++]
  - **TensorFlow** [2015 Google Brain] [C++, Python, CUDA]
  - **Apache MXNet** [2015 Apache Software Foundation] [C++]
  - **Chainer** [2015 Preferred Networks] [Python]
  - **Keras** [2016 Francois Chollet] [Python]
  - **PyTorch** [2016 Facebook] [Python, C++, CUDA]
- Apache Spark is an open-source distributed general-purpose cluster-computing framework.
  - Provides interface for programming clusters with data parallelism

BigDL

- A library on top of Apache Spark
- Provides integrated data-analytics within a unifed data analysis pipeline
- Allows users to write their own deep learning applications
- Running directly on big data clusters
- Supports similar API to Torch and Keras
- Supports both large scale distributed training and inference
- Able to run across hundreds or thousands servers efficiently by uses underlying Spark framework

BigDL

- Developed as an open source project
- Used by
  - Mastercard
  - WorldBank
  - Cray
  - Talroo
  - UCSF
  - JD
  - UnionPay
  - GigaSpaces
- Wide range of applications: transfer learning based image classification, object detection, feature extraction, sequence-to-sequence prediction, neural collaborative filtering for reccomendation etc.

Motivation

- Normally in research established datasets and benchmarks

What if we had dynamic data?

- We need to care about compatibility and efficieny in another level
  - End-to-end integrated data analytics and deep learning frameworks
- Real world data is dynamic, messy and implicitly labeled
- Requires more complex data processing
- Morevoer, it is not single shot i.e. ETL (extract, transform, load)
- It is iterative and recurrent (back-and-forth development and debugging)

Prior Approach Drawbacks

- It is highly inefficient to run on big data and deep learning systems separately
- Connector approach:
  - TFX
  - CaffeOnSpark
  - TensorFlowOnSpark
  - SageMaker
    - provides proper interface to connect data processing and deep learning frameworks
- Results in very large overheads in practice
  - inter-process communication
  - data serialization
  - impedance matching (crossing boundaries between heterogenous components)
  - persistency etc.

Prior Approach

- If a Spark worker fails, Spark system relaunch the worker
  - Highly incompatible with TensorFlow execution model
  - Causing the entire workflow to be blocked indefinetly
- BigDL
  - Directly implements the distributed deep learning in the big data
  - End-to-end single framework
  - Eliminates the impedance mismatch
  - Efficient

Data-parallel training in BigDL

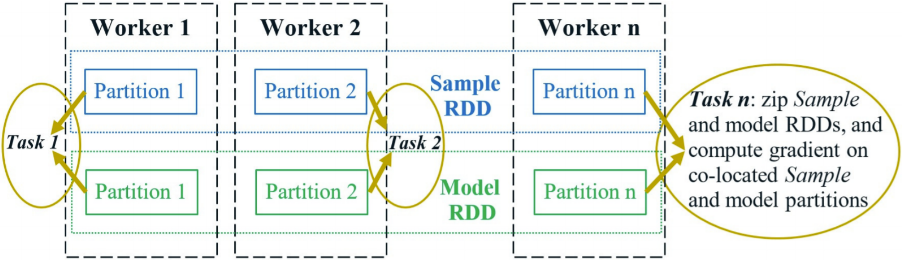


Figure: The “model forward-backward” spark job computing the local gradients for each model replica in parallel.

Parameter Synchronization in BigDL

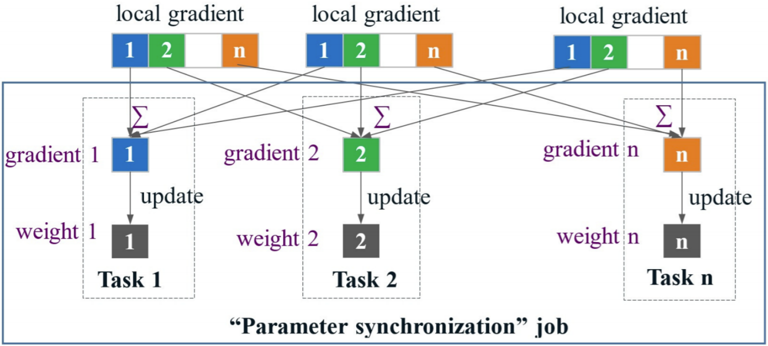
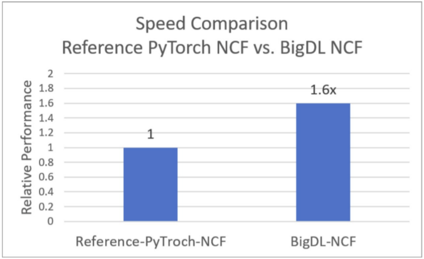


Figure: Parameter synchronization in BigDL. Each local gradient is evenly divided in N partitions; then each task n in the "parameter synchronization" job aggregates these local gradients and updates the weights for the n^{th} partition.

Evaluation

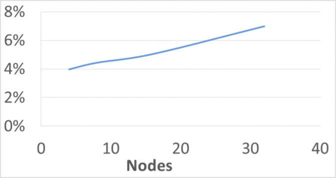
- In their experimental setup authors use
  - Neural Collaborative Filtering (NCF)
    - NCF on MovieLens 20Million dataset
    - 20 million ratings
    - 465000 tags
    - 27000 movies
    - 138000 users
  - Convolutional Neural Networks (CNNs)
    - Inception-v1 on ImageNet

Experiments for NCF



- NCF using BigDL trained on dual-socket Intel Skylake [29.8 min]
  - 56 cores and 384 GB memory
- NCF using PyTorch [Reported by MLPerf]
  - Single Nvidia P100 GPU

Experiments on ImageNet



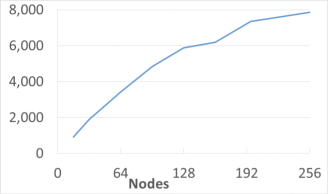


Figure: Left: Overhead of parameter synchronization measured as a fraction of the average model computation time. Right: Throughput of ImageNet Inception-v1 training in BigDL.

- ImageNet Inception-v1 using BigDL
  - Each node dual-socket Intel Broadwell 2.1GHz
  - Syncronization overhead is small (less than 7%)
  - Scales linearly up to 96 nodes.

Experiments on ImageNet Task Scheduling

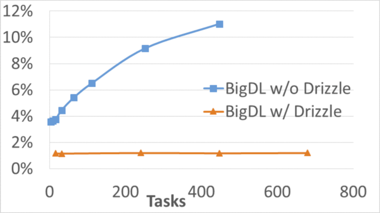


Figure: Overheads of task scheduling and dispatch for ImageNet Inception-v1 in BigDL.

- Needs to schedule very large number of tasks across cluster
  - Low for 100-200 tasks per iteration
  - Grows to over 10% close to 500 tasks per iteration
  - Using group scheduling introduced by Drizzle (low latency execution engine) can reduce this overhead

Experiments on Feature Extraction

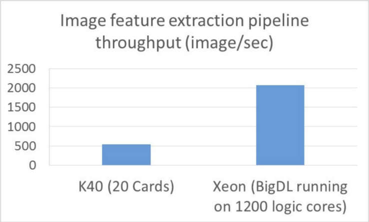


Figure: Throughput of GPU clusters and Xeon clusters for the image features extraction pipeline benchmarked by JD.

- GPU cluster consists of 20 NVIDIA Tesla K40 cards
- Xeon Cluster consists of 1200 logical cores running 50 logical cores

Critique

- Some metric to actually quantify the discussions in the experiment section
  - Server utilization?
  - Cost for equipment?
  - Network traffic?
  - Power dissipation?
- As it stands hard to interpret the actual contribution
- Explicit explanation needs to be added on speed comparison with proper metrics

PyTorch Distributed: Experiences on Accelerating Data Parallel Training

Shen Li et al

VLDB Endowment 2020

Presented by: Steven W. D. Chien for FID3024 VT20

Contributions

- Describes the design and implementation of a widely adopted industrial state-of-the-art distributed training solution
- Highlights real-world caveats
- Share performance tuning experiences collected from serving internal teams and open-source community users and summarized several directions for future improvements.

Design Principles

- API
  - Non-intrusive
  - Allow the implementation to intercept signals and trigger appropriate algorithms promptly

```
import torch
import torch.nn as nn
import torch.nn.parallel as par
import torch.optim as optim

# initialize torch.distributed properly
# with init_process_group

# setup model and optimizer
net = nn.Linear(10, 10)
net = par.DistributedDataParallel(net)
opt = optim.SGD(net.parameters(), lr=0.01)

# run forward pass
inp = torch.randn(20, 10)
exp = torch.randn(20, 10)
out = net(inp)

# run backward pass
nn.MSELoss()(out, exp).backward()

# update parameters
opt.step()
```

Design Principles

- Gradient Reduction
  - Allreduce
    - Performs poorly on small tensors
    - No opportunity to overlap computation with communication (since everyone must join)
  - Gradient bucketing
    - Still Allreduce
    - Concat small tensors into bigger chunks
    - Do not reduce all parameters in single shot
    - Triggered by autograd hook

Design Principles

- Communication Collectives
  - MPI
  - NCCL
  - Gloo
  - Use ProcessGroups to manage parallelisms (communication, CUDA, etc)

Evaluation platform

- 4 GPU servers
- Mellanox MT27700 ConnectX-4 100GB/s NIC
- 8 NVIDIA Tesla V100 GPUs per node

Latency improvement

- Overlapping approach on ResNet and BERT on 32 GPUs
  - NCCL attains 38.0% and 35.2% speedup
  - GLOO attains 26.8% and 21.5% speedup

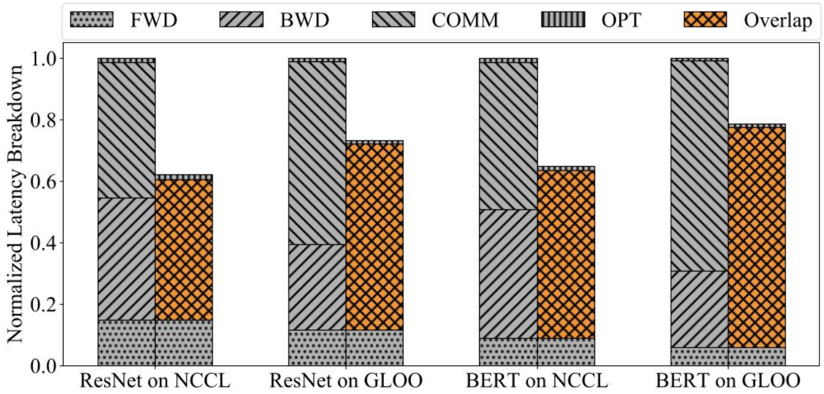
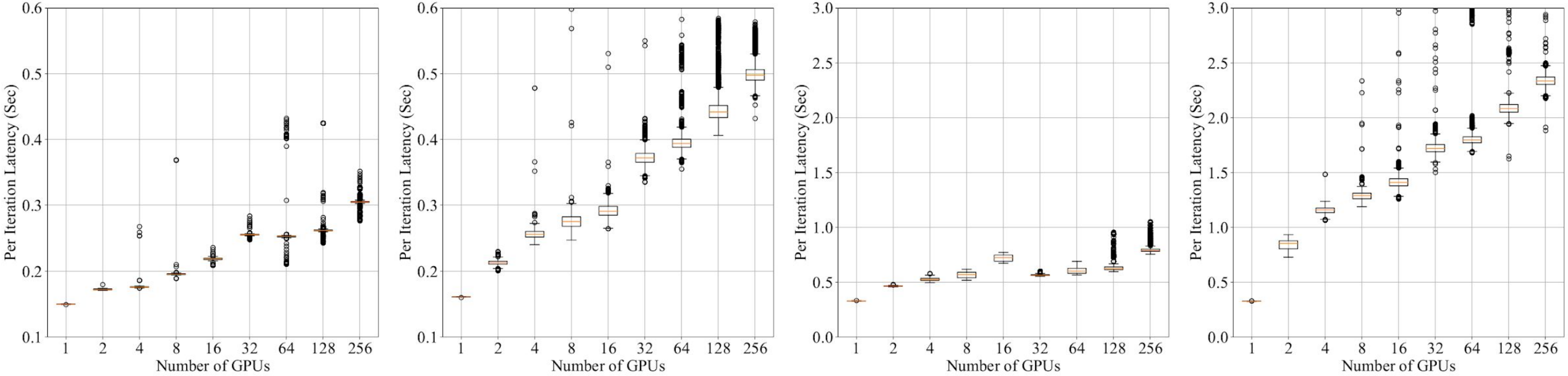


Figure 6: Per Iteration Latency Breakdown

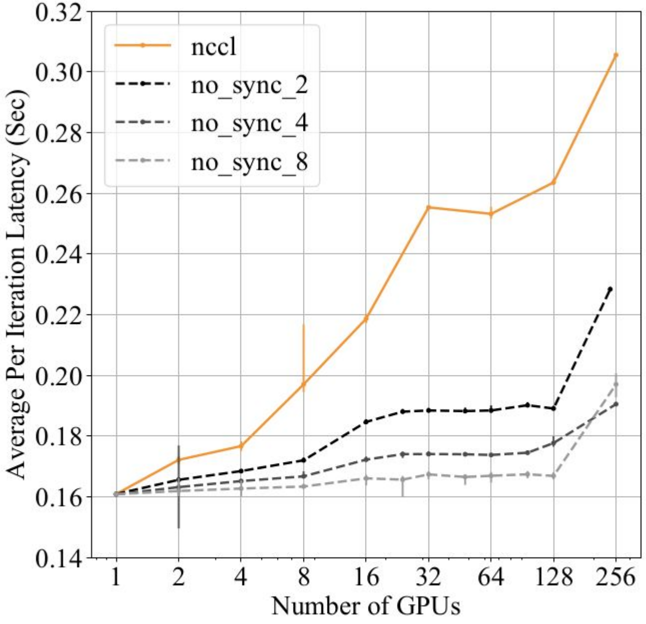
Scalability (Weak Scaling!)



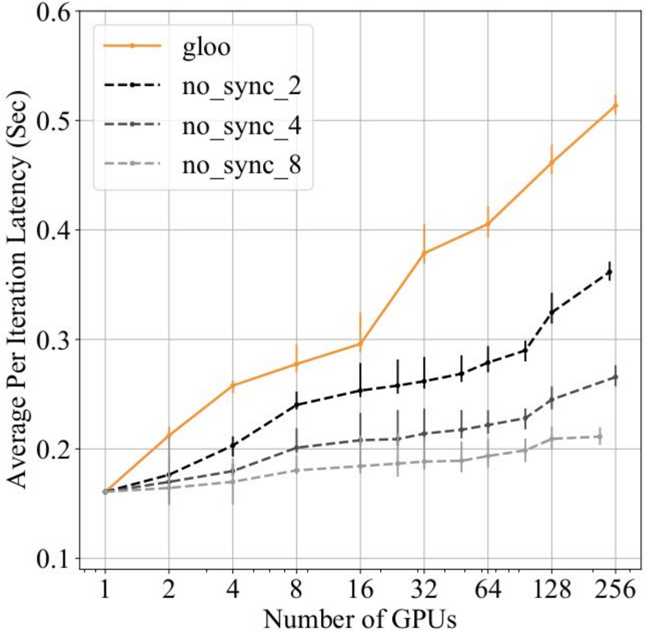
(a) ResNet50 on NCCL (b) ResNet50 on Gloo (c) BERT on NCCL (d) BERT on Gloo

Figure 9: Scalability

Scalability ("Async" update")



(a) ResNet50 on NCCL



(b) ResNet50 on Gloo

Figure 10: Skip Gradient Synchronization

Conclusion

- Communication backends
- Bucket size (transfer size)
- Resource allocation (out-of-core training)

Discussion

- What are the limitation of hook based communication trigger?
- Can this scheme impact progress? (e.g. stall for bucket 0)

Beyond Data and Model parallelism For Deep Neural Networks

Zhihao Jia, Matei Zaharia, Alex Aiken

MLSys'19

Presented by: Steven W. D. Chien for FID3024 VT20

- Background and Research Question
- Hybrid Parallelism
- Solution
- Optimization
- Related work and Conclusion

Background

- Data parallelism
- Model parallelism

Data parallelism

- Batches as finest granularity
- Synchronize every certain intervals
- Care required in synchronization and parameter tuning

Model parallelism

- Operations in model as lowest granularity
- All operations execute ayshcnrounsly, subject to dependency
- Difficult placement
    - Computation power
    - Communication cost
    - Degree of overlapping

What if….

The operations themselves can be parallelized?

FlexFlow: SOAP

- **S**ample
- **O**perator
- **A**ttribute
- **P**arameter

FlexFlow: SOAP

- **S**ample
  - E.g. sample based data parallelism
- **O**perator
  - E.g. Conv / gemm
- **A**ttribute
  - E.g. Length / Width
- **P**arameter
  - E.g. Channels

FlexFlow: SOAP

Table 2. Parallelizable dimensions for different operators. The *sample* and *channel* dimension index different samples and neurons, respectively. For images, the *length* and the combination of *height* and *width* dimensions specify a position in an image.

Operator	**Parallelizable Dimensions**		
	(S)ample	**(A)ttribute**	**(P)arameter**
1D pooling	sample	length, channel	
1D convolution	sample	length	channel
2D convolution	sample	height, width	channel
Matrix multiplication	sample		channel

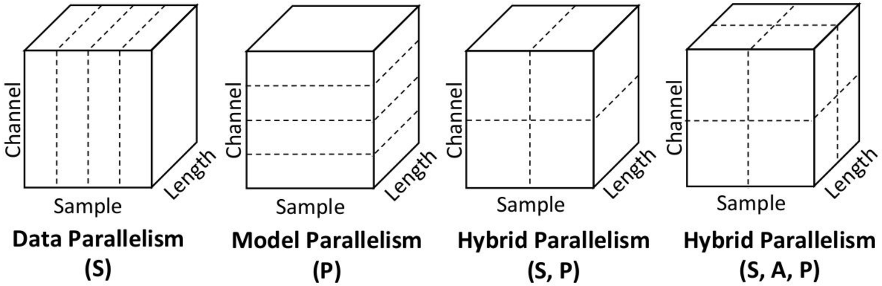


Figure 2. Example parallelization configurations for 1D convolution. Dashed lines show partitioning the tensor.

Jia, Z., Zaharia, M. and Aiken, A., 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*.

How to compute a matrix multiplication?

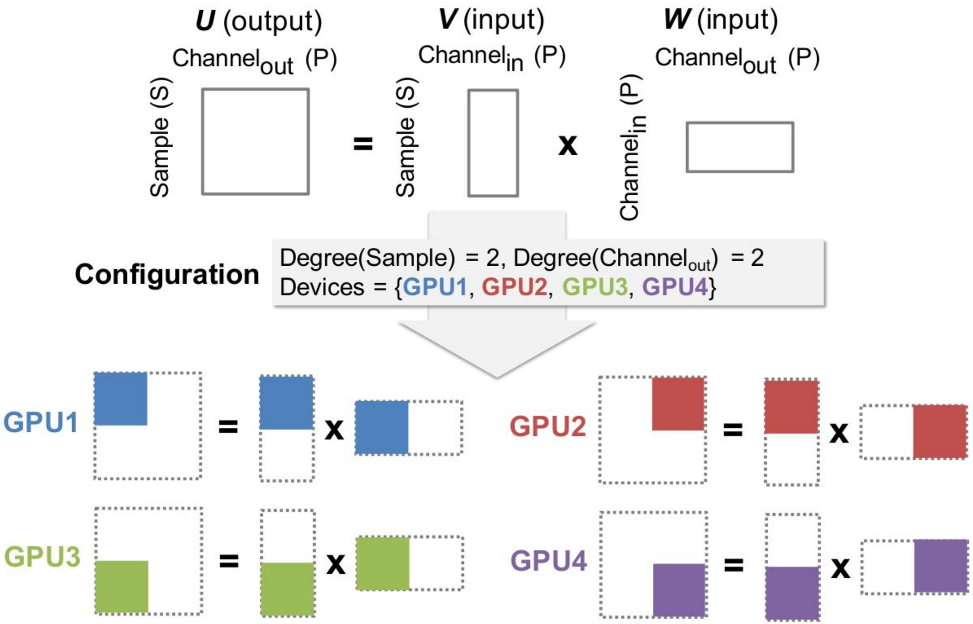
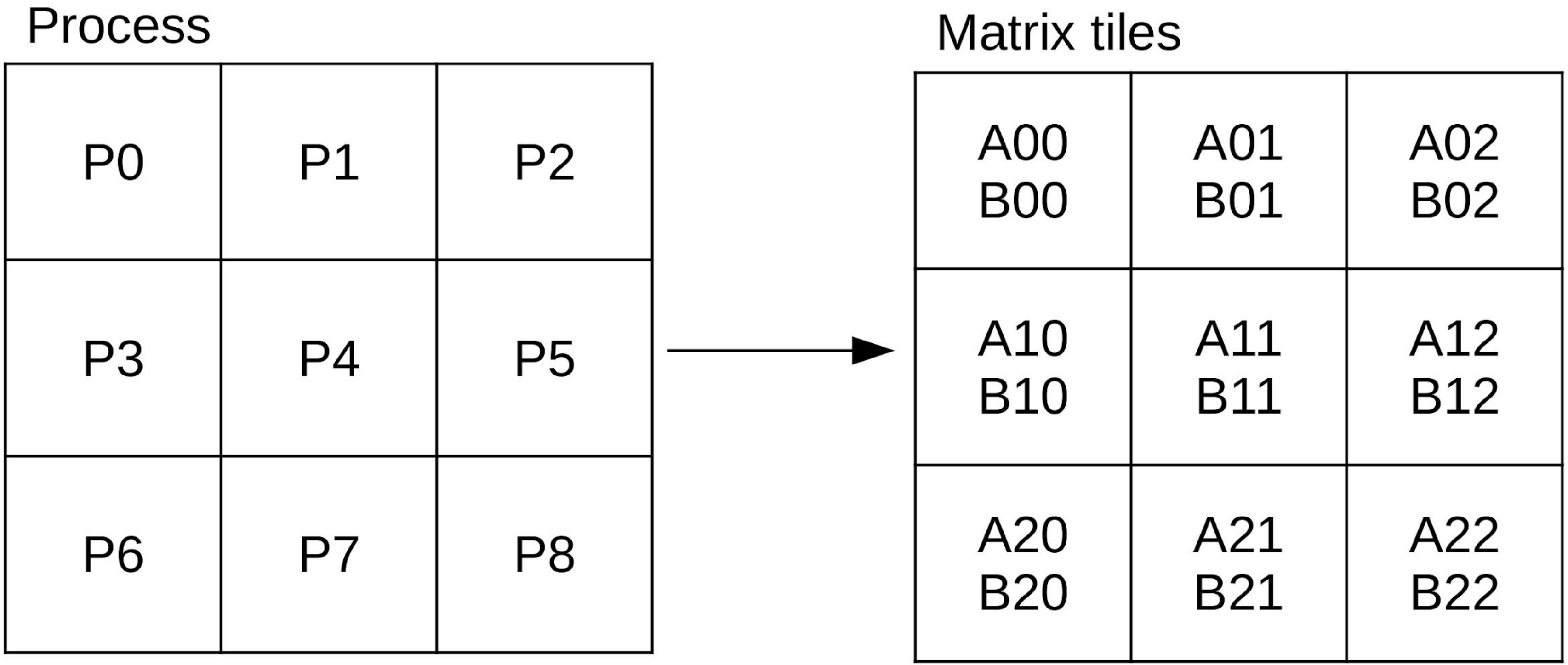


Figure 3. An example parallelization configuration for a matrix multiplication operator.

Jia, Z., Zaharia, M. and Aiken, A., 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*.

How to compute a matrix multiplication?



How to compute a matrix multiplication?

Buffer A

A00	A01	A02
A10	A11	A12
A20	A21	A22

Buffer B

B00	B01	B02
B10	B11	B12
B20	B21	B22

Buffer C

A00 B00	A00 B01	A00 B02
A11 B10	A11 B11	A11 B12
A22 B20	A22 B21	A22 B22

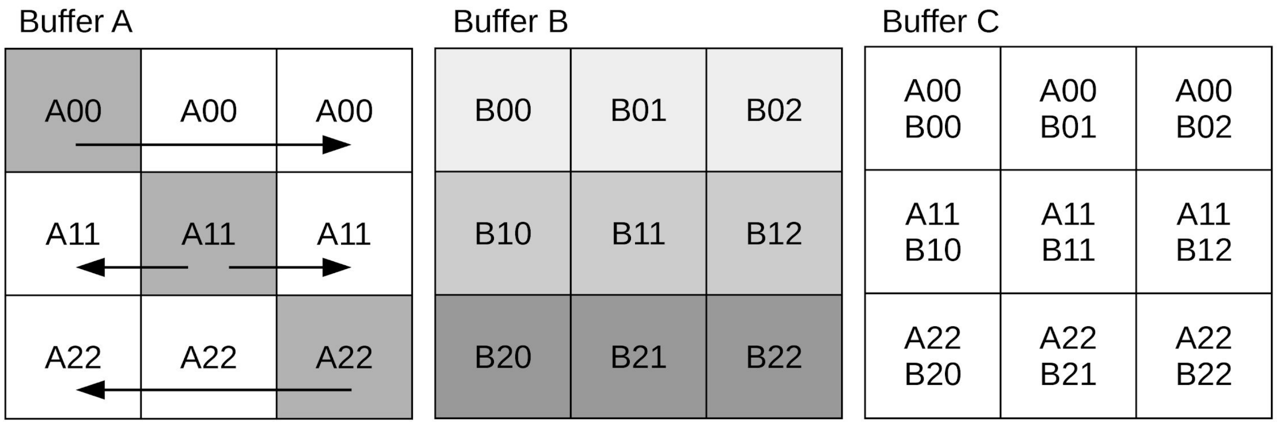
#iterations = √#tiles = √9 = 3

C00 = A00 x B00 + A01 x B10 + A02 x B20
C10 = A10 x B00 + A11 x B10 + A 12 x B20
C20 = A20 x B00 + A21 x B10 + A22 x B20

C01 = A00 x B01 + A01 x B11 + A02 x B21
C11 = A10 x B01 + A11 x B11 + A12 x B21
C21 = A20 x B01 + A21 x B11 + A22 x B21

C02 = A00 x B02 + A01 x B12 + A02 x B22
C12 = A10 x B02 + A11 x B12 + A12 x B22
C22 = A20 x B02 + A21 x B12 + A22 x B22

How to compute a matrix multiplication?

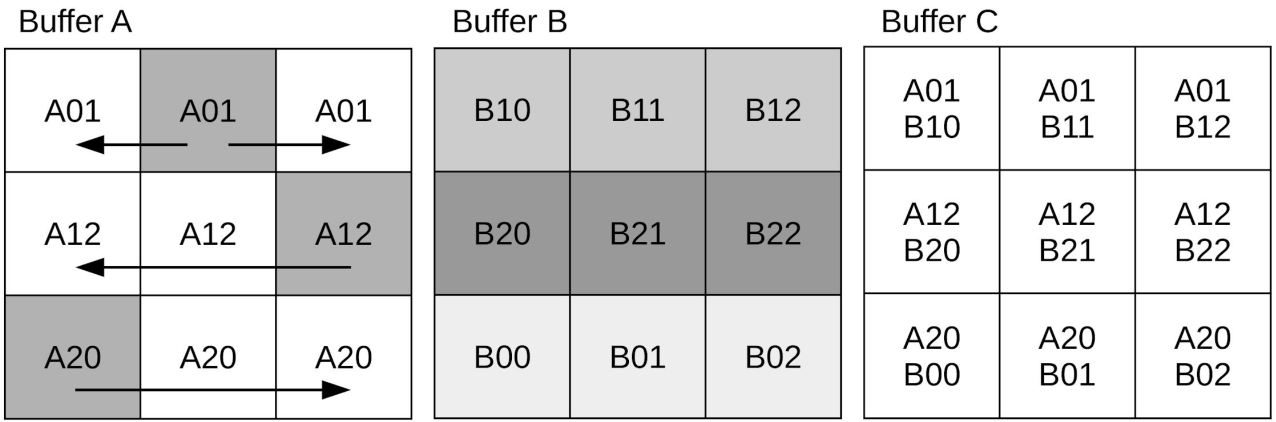


C00 = A00 x B00 + A01 x B10 + A02 x B20
C10 = A10 x B00 + A11 x B10 + A 12 x B20
C20 = A20 x B00 + A21 x B10 + A22 x B20

C01 = A00 x B01 + A01 x B11 + A02 x B21
C11 = A10 x B01 + A11 x B11 + A12 x B21
C21 = A20 x B01 + A21 x B11 + A22 x B21

C02 = A00 x B02 + A01 x B12 + A02 x B22
C12 = A10 x B02 + A11 x B12 + A12 x B22
C22 = A20 x B02 + A21 x B12 + A22 x B22

How to compute a matrix multiplication?

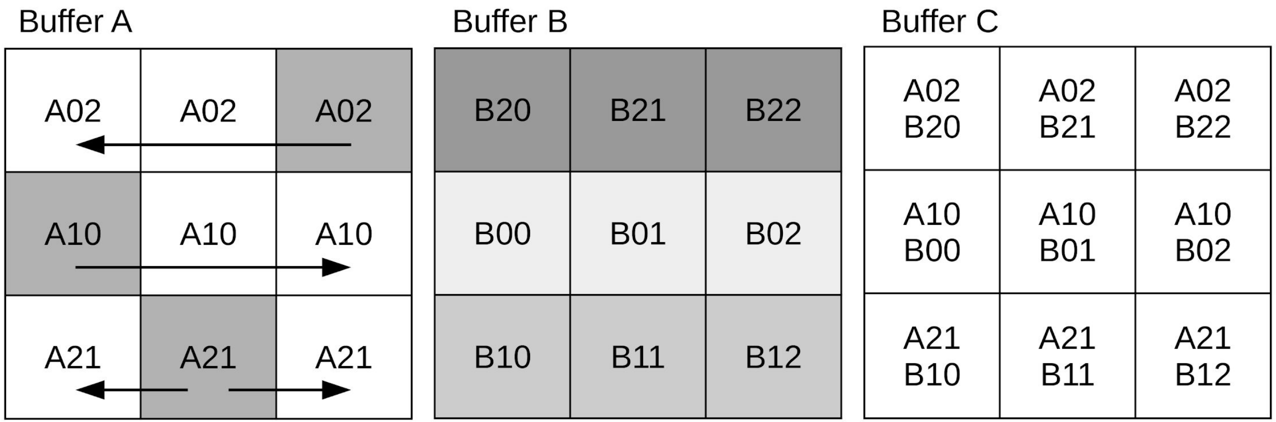


C00 = A00 x B00 + A01 x B10 + A02 x B20
C10 = A10 x B00 + A11 x B10 + A 12 x B20
C20 = A20 x B00 + A21 x B10 + A22 x B20

C01 = A00 x B01 + A01 x B11 + A02 x B21
C11 = A10 x B01 + A11 x B11 + A12 x B21
C21 = A20 x B01 + A21 x B11 + A22 x B21

C02 = A00 x B02 + A01 x B12 + A02 x B22
C12 = A10 x B02 + A11 x B12 + A12 x B22
C22 = A20 x B02 + A21 x B12 + A22 x B22

How to compute a matrix multiplication?



C00 = A00 x B00 + A01 x B10 + A02 x B20
C10 = A10 x B00 + A11 x B10 + A 12 x B20
C20 = A20 x B00 + A21 x B10 + A22 x B20

C01 = A00 x B01 + A01 x B11 + A02 x B21
C11 = A10 x B01 + A11 x B11 + A12 x B21
C21 = A20 x B01 + A21 x B11 + A22 x B21

C02 = A00 x B02 + A01 x B12 + A02 x B22
C12 = A10 x B02 + A11 x B12 + A12 x B22
C22 = A20 x B02 + A21 x B12 + A22 x B22

Wait… But this is not as simple as ScaLAPACK?! 😱

Challenges

- What we know
  - Configurations of tensors (S, A, P)
- What we do not know
  - How to optimally split an operation (O) subject to constraints
  - Resources
  - Communication
  - Overlapping

Parallelization Strategy

- Partitions an operator o_i into $|c_i|$ independent tasks $t_{i:1}, \ldots, t_{i:|c_i|}$
- Configuration also includes the device assignment for each task
  - $t_{i:k} (1 \leq k \leq |c_i|)$
- Infer the necessary input tensors for each task using the output Tensors

Parallelization Strategy

- Define S that describes one possible parallelization for each operation
- Configurations of each operator can be randomly chosen

Choosing strategy: Execution simulator

- Assumptions
    - Input independent and predictable execution time with low variance
    - Communication bandwidth can be fully utilized
    - FIFO scheduling policy
    - Negligible runtime overhead

Construct a task graph

1. Two kinds of tasks: Normal (compute, communicate, Edge (dependency)
2. Place all tasks of an operation in the graph
3. Connect the input and output tensors (device placement)
4. If two connected tasks are on the same device, add an edge
5. Else, add a communication task

Simulate

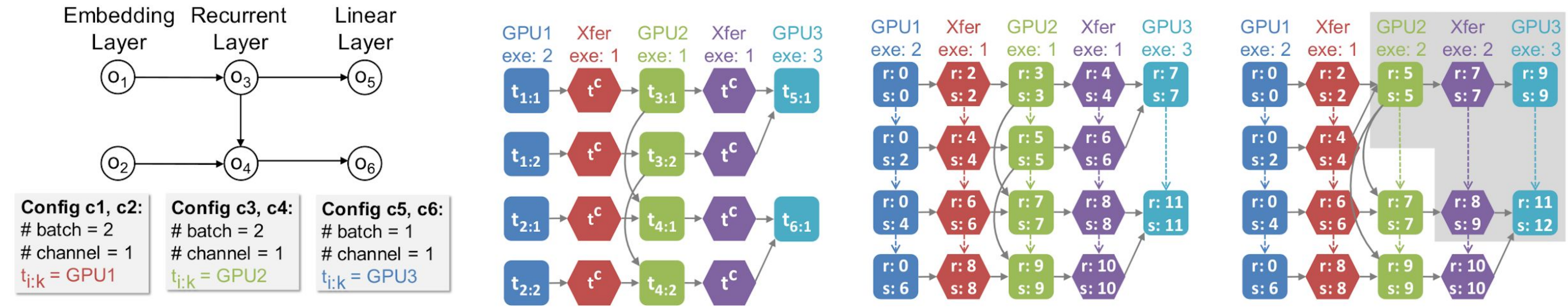
- Fill in a number of properties
  - Device
  - exeTime
  - …
- Run a variant of Dijkstra's algorithm
- Dequeue in order of ready time

Table 3. Properties for each task in the task graph.

Property	**Description**
Properties set in graph construction	
`exeTime`	The elapsed time to execute the task.
`device`	The assigned device of the task.
$\mathcal{I}$(t)	$\{t_{in}\|(t_{in}, t) \in \mathcal{T}_E\}$
$\mathcal{O}$(t)	$\{t_{out}\|(t, t_{out}) \in \mathcal{T}_E\}$
Properties set in simulation	
`readyTime`	The time when the task is ready to run.
`startTime`	The time when the task starts to run.
`endTime`	The time when the task is completed.
`preTask`	The previous task performed on `device`.
`nextTask`	The next task performed on `device`.
Internal properties used by the full simulation algorithm	
`state`	Current state of the task, which is one of `NOTREADY`, `READY`, and `COMPLETE`.

Optimize task graph

- Introduces “Delta simulation algorithm”
  - Change configuration of one operator at a time
  - Only resimulate the affected operations
- Search
  - Use existing strategy (only data parallel, expert tuned) as initial condition
  - Replace one operator configuration at a time (randomly) by a random config
  - Use expected execution as cost function for minimization



(a) An example parallelization strategy. (b) The corresponding task graph. (c) The task graph after the full simulation algorithm. (d) The task graph after the delta simulation algorithm.

Figure 4. Simulating an example parallelization strategy. The tasks' `exeTime` and `device` are shown on the top of each column. In Figure 4c and 4d, the word "r" and "s" indicate the `readyTime` and `startTime` of each task, respectively, and the dashed edges indicate the `nextTask`.

Implementation

- FlexFlow (Not the one in HPCA'17 ?)
- cuDNN
- cuBLAS
- Legion (Task based runtime, SC'12)

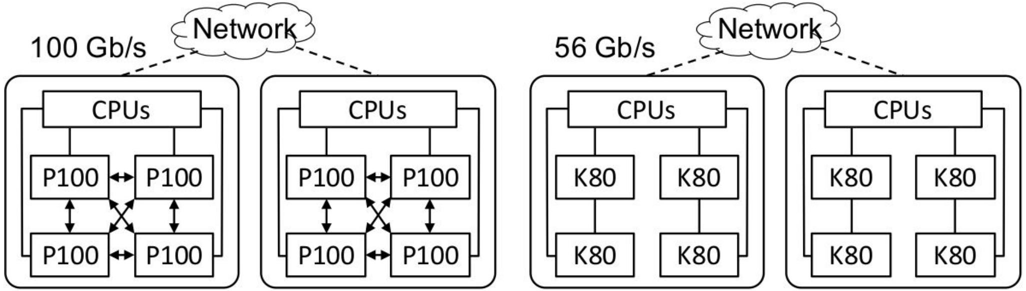
Evaluation: Applications

Table 4. Details of the DNNs and datasets used in evaluation.

DNN	**Description**	**Dataset**	**Reported Acc.**	**Our Acc.**
Convolutional Neural Networks (CNNs)				
AlexNet	A 12-layer CNN	Synthetic data	-	-
Inception-v3	A 102-layer CNN with Inception modules (Szegedy et al., 2014)	ImageNet	78.0%[a]	78.0%[a]
ResNet-101	A 101-layer residual CNN with shortcut connections	ImageNet	76.4%[a]	76.5%[a]
Recurrent Neural Networks (RNNs)				
RNNTC	4 recurrent layers followed by a softmax layer	Movie Reviews (Movies)	79.8%	80.3%
RNNLM	2 recurrent layers followed by a softmax layer	Penn Treebank (Marcus et al.)	78.4[b]	76.1[b]
NMT	4 recurrent layers followed by an attention and a softmax layer	WMT English-German (WMT)	19.67[c]	19.85[c]

[a] top-1 accuracy for single crop on the validation dataset (higher is better).
[b] word-level test perplexities on the Peen Treebank dataset (lower is better).
[c] BLEU scores (Papineni et al., 2002) on the test dataset (higher is better).

Evaluation: Platforms



(a) The P100 Cluster (4 nodes). (b) The K80 Cluster (16 nodes).

Figure 5. Architectures of the GPU clusters used in the experiments. An arrow line indicates a NVLink connection. A solid line is a PCI-e connection. Dashed lines are Infiniband connections across different nodes.

Evaluation: Configuration search

- Data parallelism as initial condition
- 30 minutes search budget

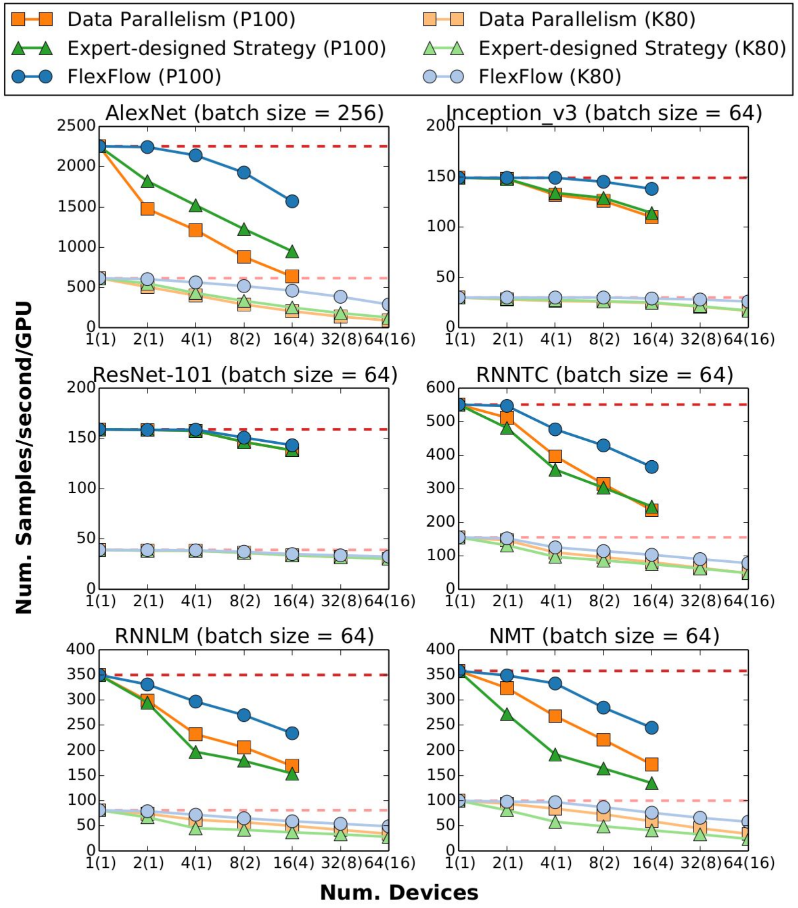
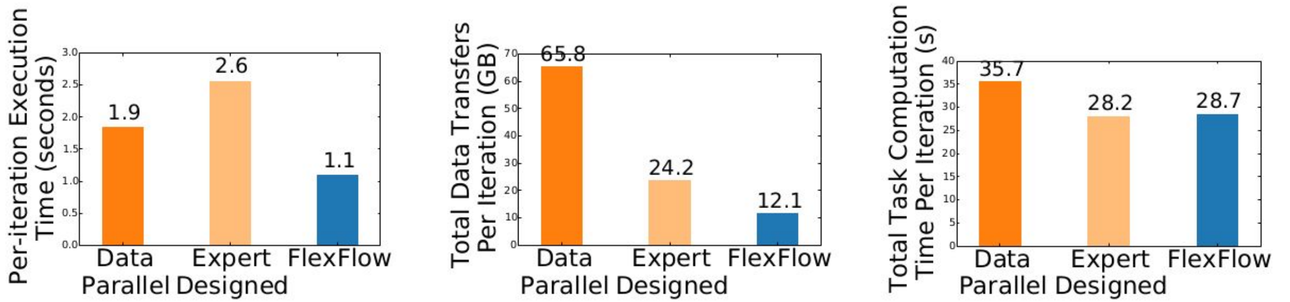


Figure 6. Per-iteration training performance on six DNNs. Numbers in parenthesis are the number of compute nodes used in the experiments. The dash lines show the ideal training throughput.

Three figure of merits



(a) Per-iteration execution time.

(b) Overall data transfers per iteration.

(c) Overall task run time per iteration.

Figure 7. Parallelization performance for NMT on 64 K80 GPUs (16 nodes). FlexFlow reduces per-iteration execution time by 1.7-2.4× and data transfers by 2-5.5× compared to other approaches. FlexFlow achieves similar overall task computation time as expert-designed strategy, which is 20% fewer than data parallelism.

End-to-end training performance

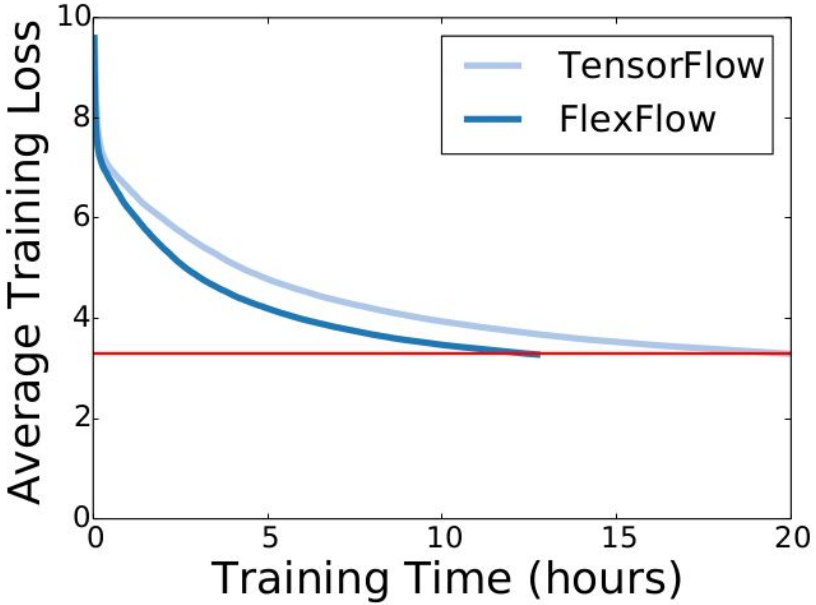


Figure 8. Training curves of Inception-v3 in different systems. The model is trained on 16 P100 GPUs (4 nodes).

Speedup of configuration search time using Delta

Table 5. The end-to-end search time with different simulation algorithms (seconds).

Num. GPUs	AlexNet Full	AlexNet Delta	AlexNet Speedup	ResNet Full	ResNet Delta	ResNet Speedup	Inception Full	Inception Delta	Inception Speedup	RNNTC Full	RNNTC Delta	RNNTC Speedup	RNNLM Full	RNNLM Delta	RNNLM Speedup	NMT Full	NMT Delta	NMT Speedup
4	0.11	0.04	**2.9**×	1.4	0.4	**3.2**×	14	4.1	**3.4**×	16	7.5	**2.2**×	21	9.2	**2.3**×	40	16	**2.5**×
8	0.40	0.13	**3.0**×	4.5	1.4	**3.2**×	66	17	**3.9**×	91	39	**2.3**×	76	31	**2.5**×	178	65	**2.7**×
16	1.4	0.48	**2.9**×	22	7.3	**3.1**×	388	77	**5.0**×	404	170	**2.4**×	327	121	**2.7**×	998	328	**3.0**×
32	5.3	1.8	**3.0**×	107	33	**3.2**×	1746	298	**5.9**×	1358	516	**2.6**×	1102	342	**3.2**×	2698	701	**3.8**×
64	18	5.9	**3.0**×	515	158	**3.3**×	8817	1278	**6.9**×	4404	1489	**3.0**×	3406	969	**3.6**×	8982	2190	**4.1**×

Simulation Accuracy

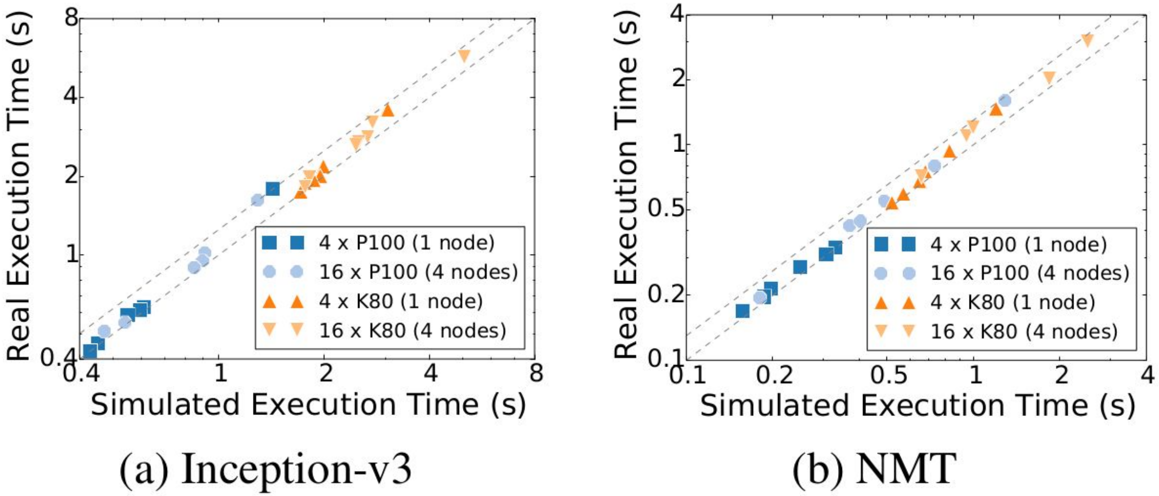


Figure 10. Comparison between the simulated and actual execution time for different DNNs and device topologies.

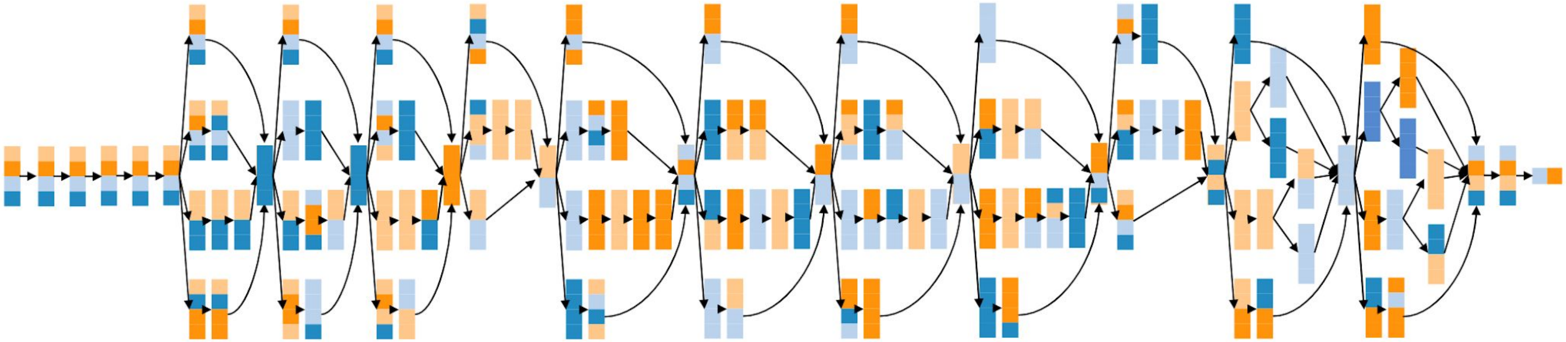


Figure 12. The best discovered strategy for parallelizing Inception-v3 on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each GPU is denoted by a color.

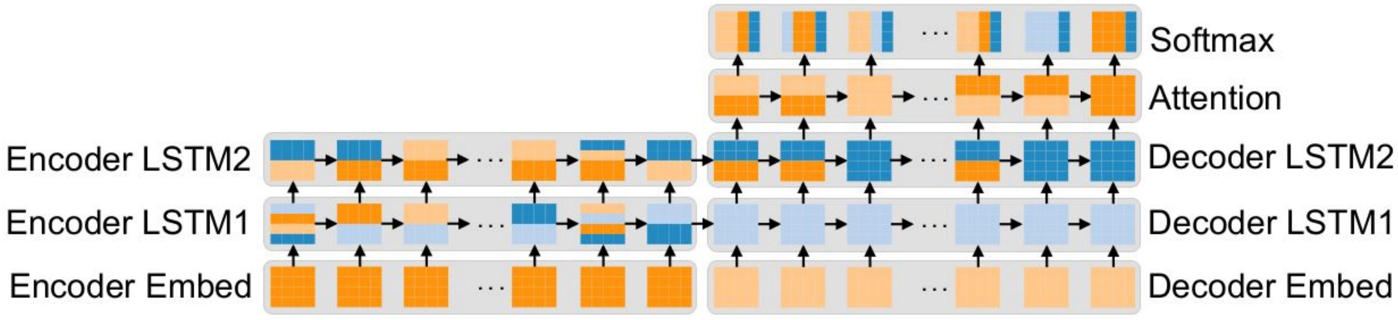


Figure 13. The best discovered strategy for parallelizing NMT on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each grey box denotes a layer, whose operators share the same network parameters. Each GPU is denoted by a color.

Conclusion

- FlexFlow that uses SOAP: A lower granularity of parallelization
- Transforming into a “task” runtime problem
- Uses traditional optimization techniques

Discussion

- What are the alternatives to task based runtime in this context?
- Can this scheme be modeled as a traditional parallel application?
- Is the full bandwidth utilization assumption over optimistic?

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, Yuxiong He

2019, Microsoft

Data Parallelism (DP)

- Compute/communication efficiency
- Poor memory efficiency

Redundant memory allocation

Model Parallelism (MP)

- Favourable memory efficiency
- Poor compute/communication efficiency

Expensive communication

Both keep all the model states over entire training process

Zero Redundancy Optimizer (ZeRO)

Goal: Achieve the best of both worlds

Make full aggregate memory capacity of a cluster available while remaining efficient

Contribution:

Reduce per-device memory footprint linearly with the increased degree of parallelism while keeping communication close to that of default DP

1. Improved training speed for large models
2. Independence of model size

An Example

1.5B parameter GPT-2 trained with ADAM

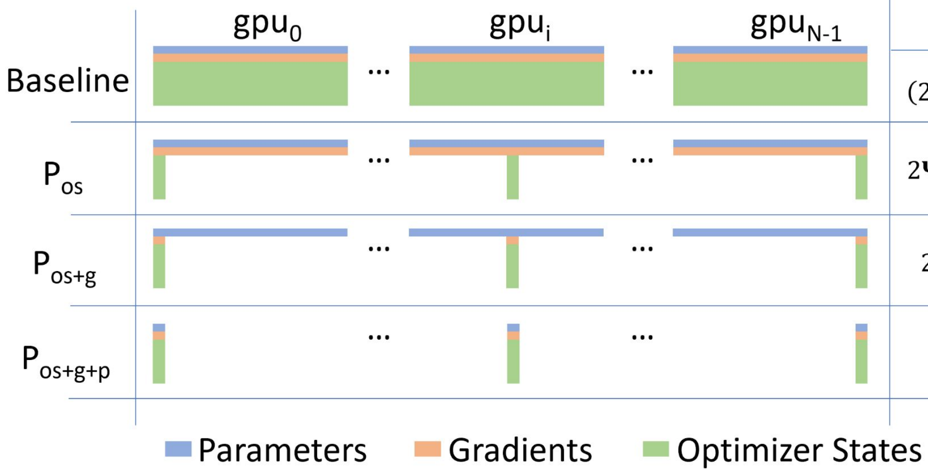
- Weights/parameters: 3GB with fp16 (2Ψ)
- Gradients: 3GB with fp16 (2Ψ)
- Optimizer state: fp32 copy of parameters, momentum, variance -> 18GB ($4\Psi+4\Psi+4\Psi$)

Residual memory:

- Activations: for a GPT like model *12 * hidden_dim * batch * seq_length * transformer_layers* (60 GB)
    - Checkpointing: trading-off memory for computation
- Temporary buffers: Gradient fusion for improved device throughput (6GB)
- Memory fragmentation: long-lived vs. short-lived memory
    - Reduces practically available amount of memory
    - OOM with over 30% memory still available

Optimizing Model State Memory (ZeRO-DP)

Assumption: For large models, the majority of the memory is occupied by model states which includes **optimizer states** (momentum, variances), **gradients** and **parameters.**



	Memory Consumed	K=12 $\boldsymbol{\Psi}$=7.5B N_d=64	Memory Reduction
Baseline	$(2+2+K)*\boldsymbol{\Psi}$	120GB	
P_{os}	$2\boldsymbol{\Psi}+2\boldsymbol{\Psi}+\frac{K*\boldsymbol{\Psi}}{N_d}$	31.4GB	4x
P_{os+g}	$2\boldsymbol{\Psi}+\frac{(2+K)*\boldsymbol{\Psi}}{N_d}$	16.6GB	8x
P_{os+g+p}	$\frac{(2+2+K)*\boldsymbol{\Psi}}{N_d}$	1.9GB	Linear with Nd

Optimizing Rrsidual State Memory (Zero-R)

That is: **Activation, temporary buffers** and unusable **fragmented memory**.

1. Activation partitioning and CPU offloading
2. Constant size temporary buffer size
3. Proactive management of memory with respect to tensor lifetime.

Do we still need MP, and when?

ZeRO-DP is at least as effective in reducing memory as MP, or even more effective when MP cannot divide model evenly. + scales better.

1. MP can be extended with ZeRO-R
2. Smaller models, MP might have better convergence due to large batch size in Zero-DP.

Relation to Other Optimizations

Pipeline Parallelism:

Often incurs functionality, performance and convergence

Activation Memory Optimization:

Compression, checkpointing or live analysis (complementary)

CPU Offloading:

Can be avoided due to memory reduction

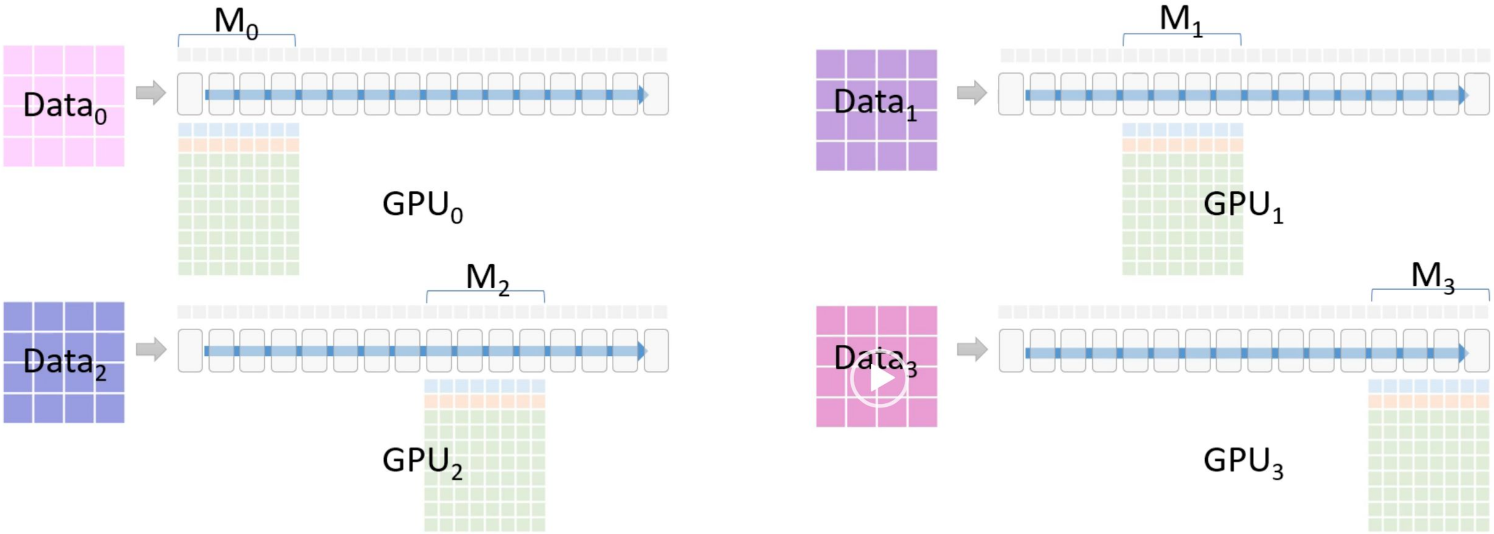
Memory efficient optimizers:

Impact convergence (orthogonal, ZeRO does not change the optimizer)

Training Optimizers:

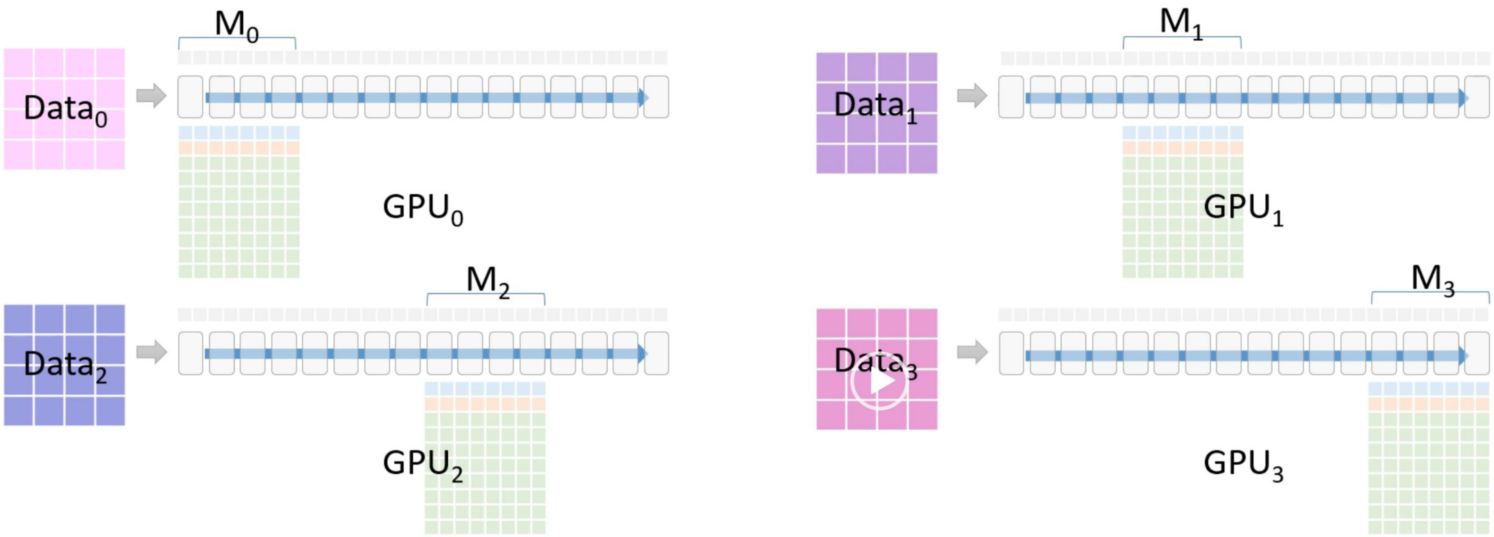
ZeRO makes more sophisticated optimizers possible

ZeRO-DP

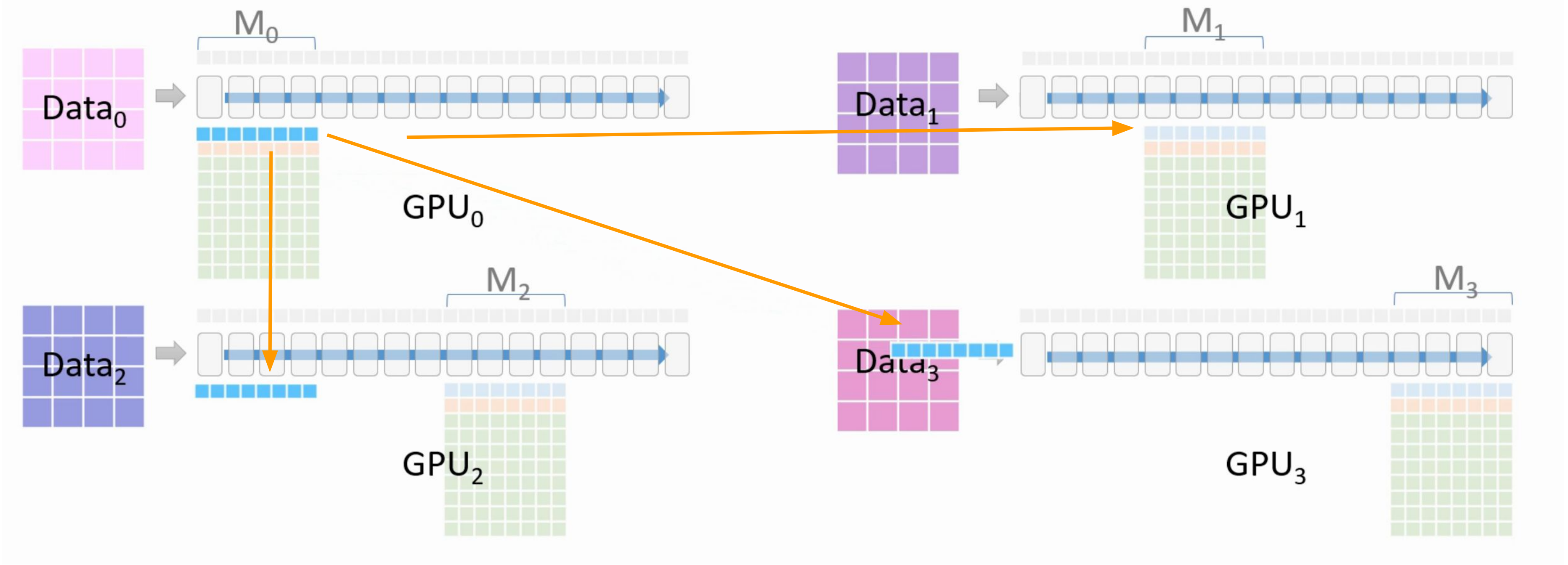


https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

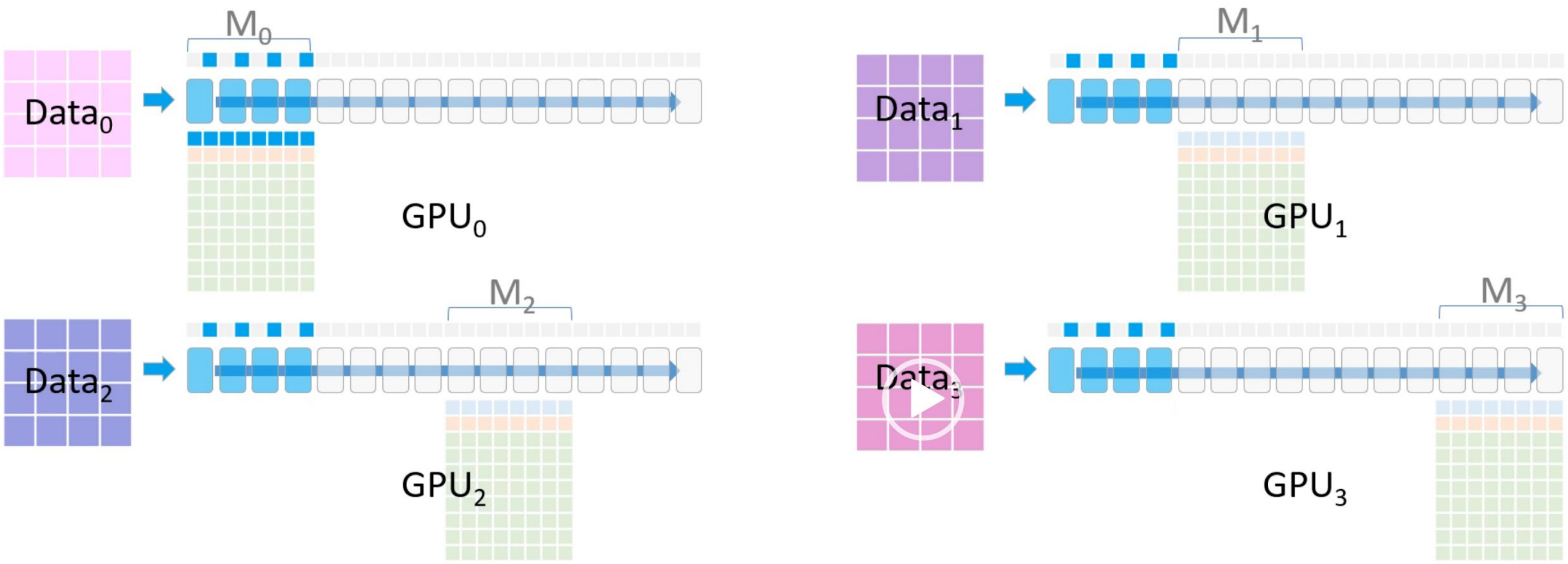
Zero-DP



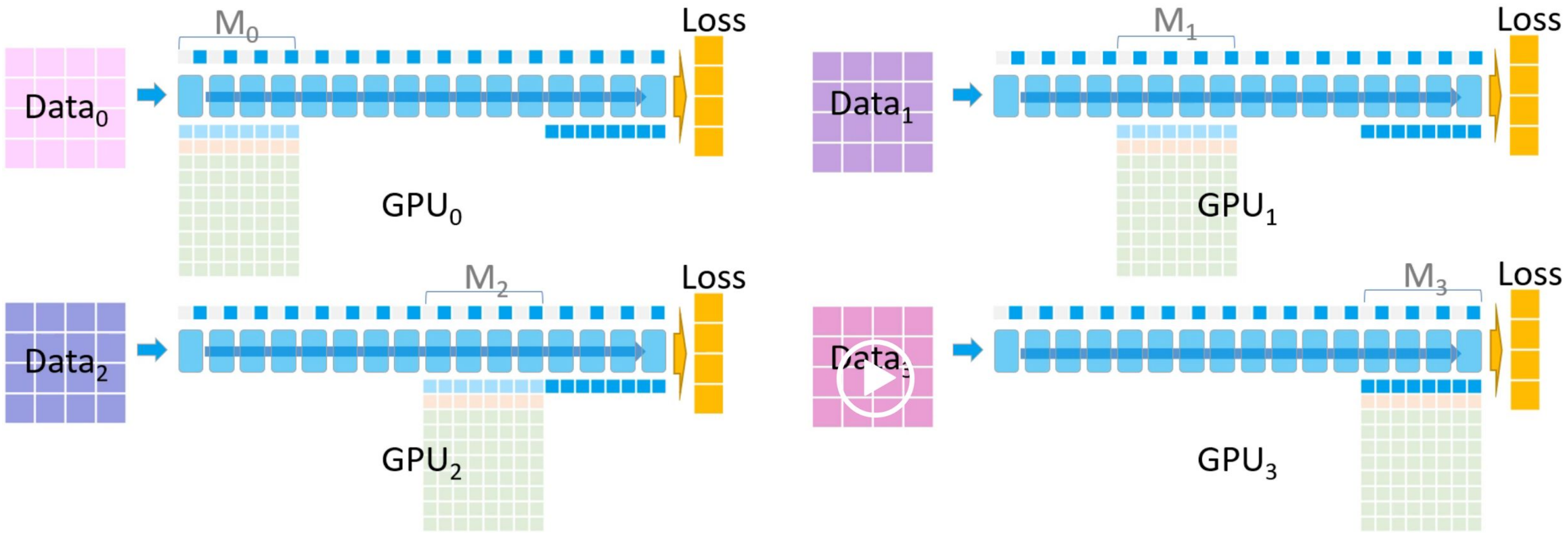
GPU0 initially has parameters of M0 -> broadcast



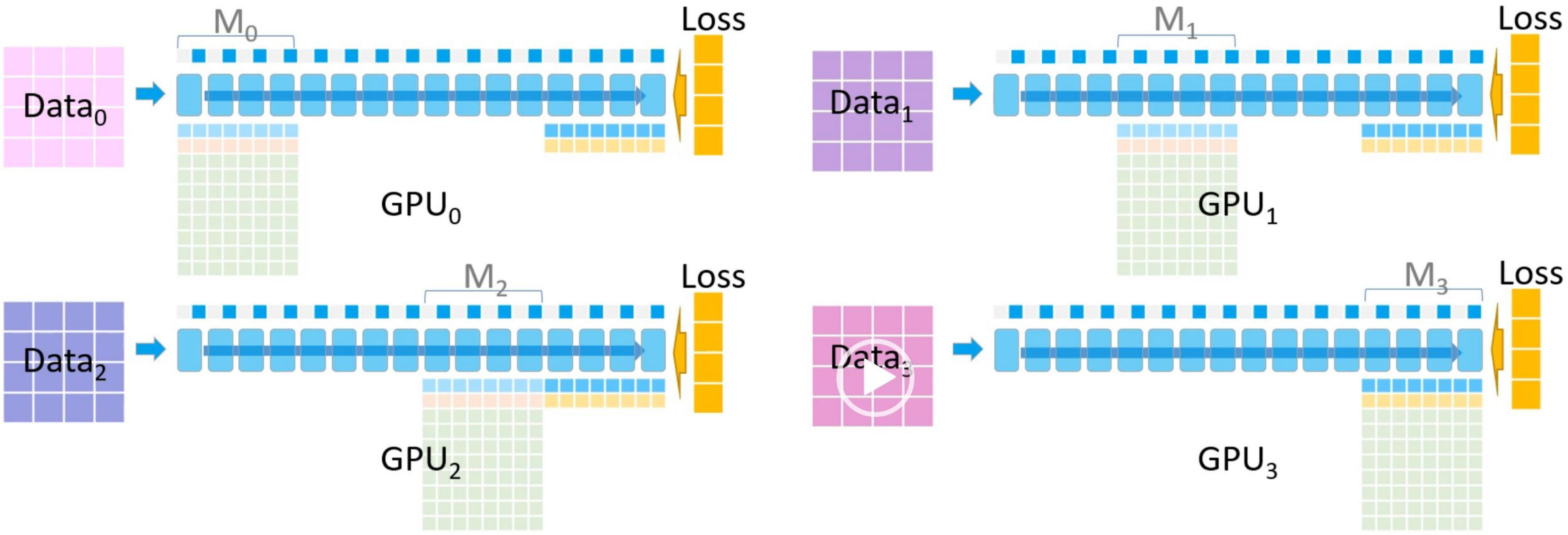
1,2,3 delete parameters and 1 continues broadcasting parameters of M1



Forward Pass complete -> Loss

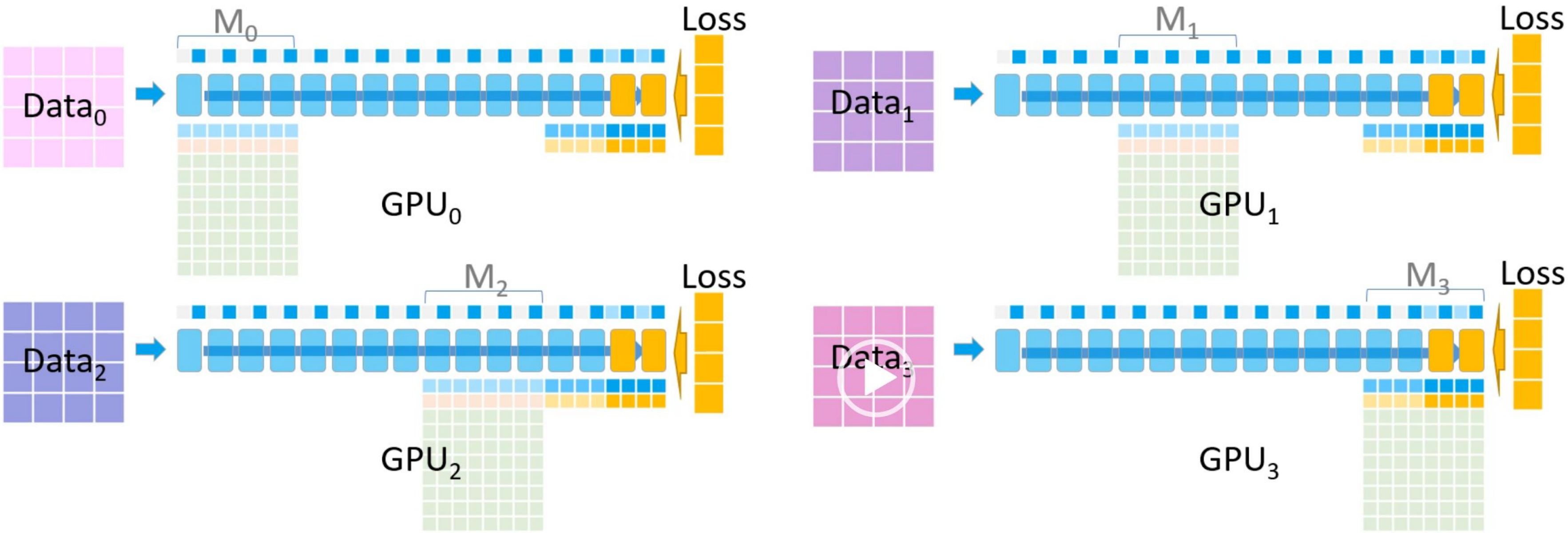


The Backward Pass starts on M3



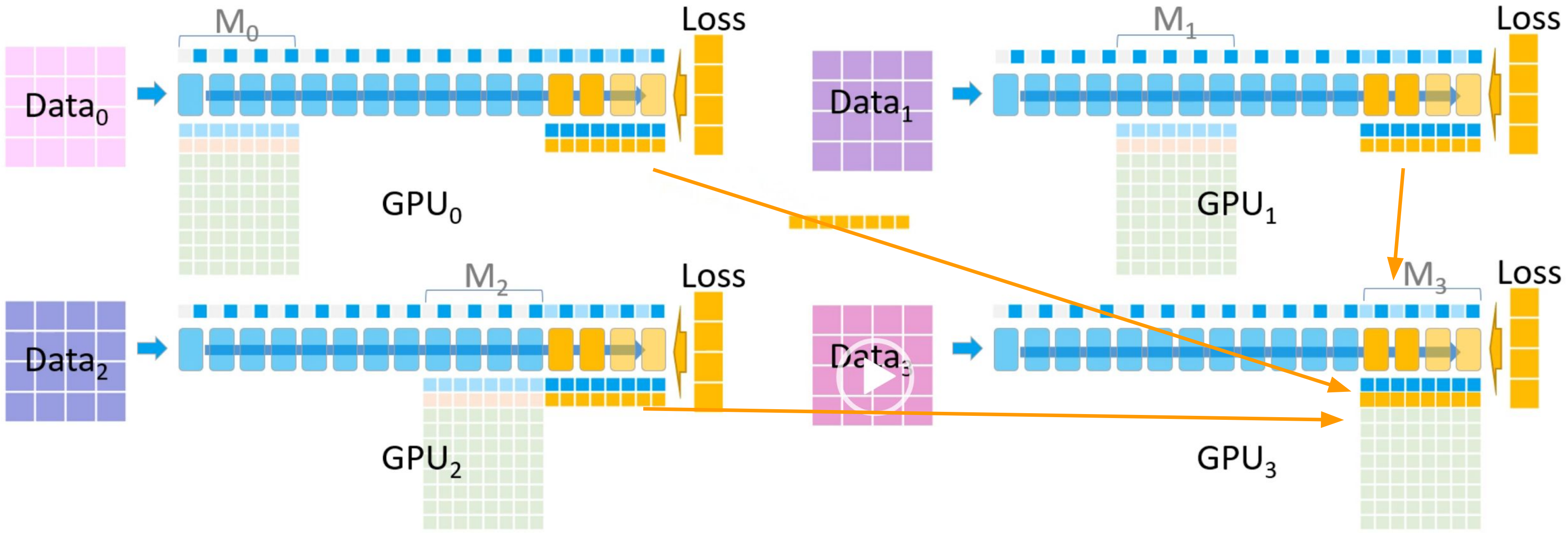
The backwards pass starts.
$GPU_{0,1,2}$ will hold a temporary buffer M_3 gradients on $Data_{0,1,2}$

M3 on all devices



The backwards pass proceeds on M_3

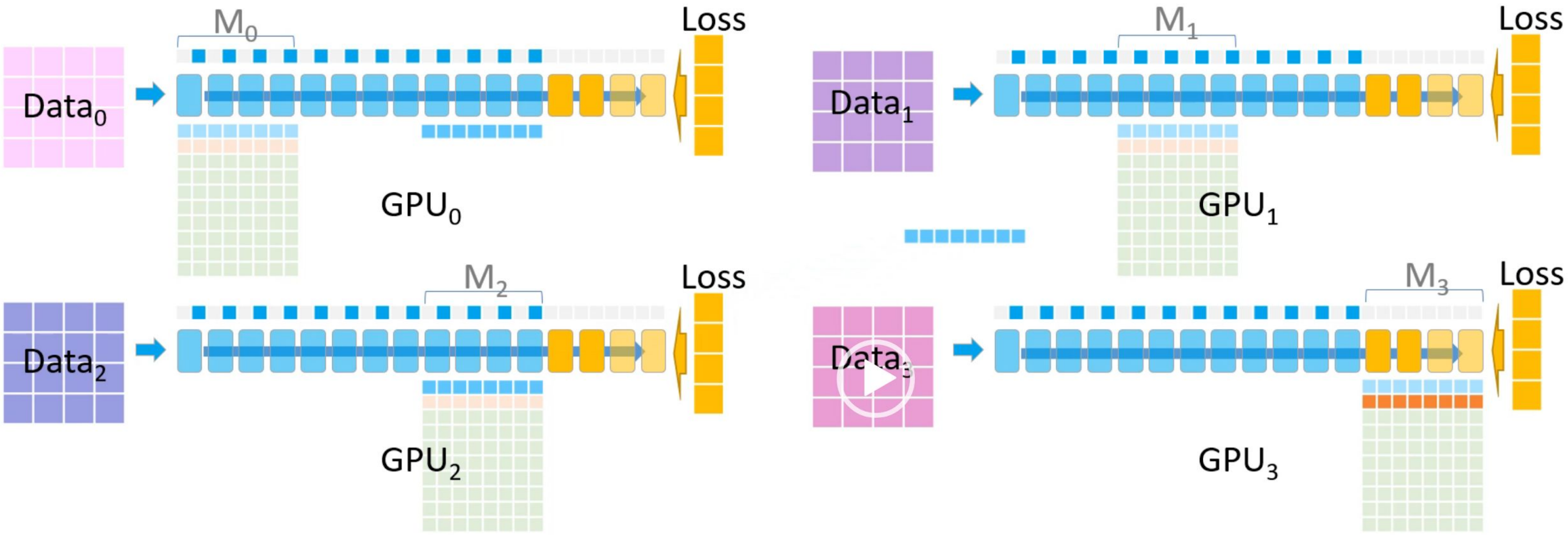
The activations for M_3 are recomputed from the saved partial activations



$GPU_{0,1,2}$ pass their M_3 gradients to GPU_3

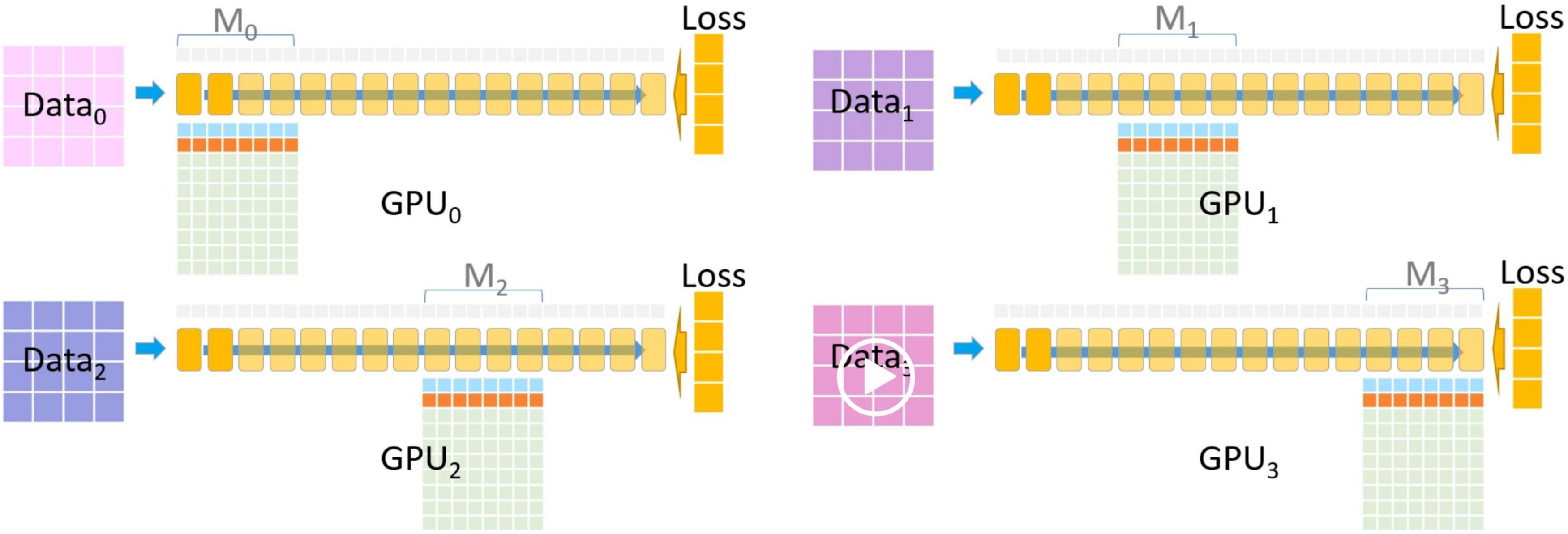
GPU_3 performs gradient accumulation and holds final M_3 for all Data

To Start M2, GPU2 broadcasts parameters



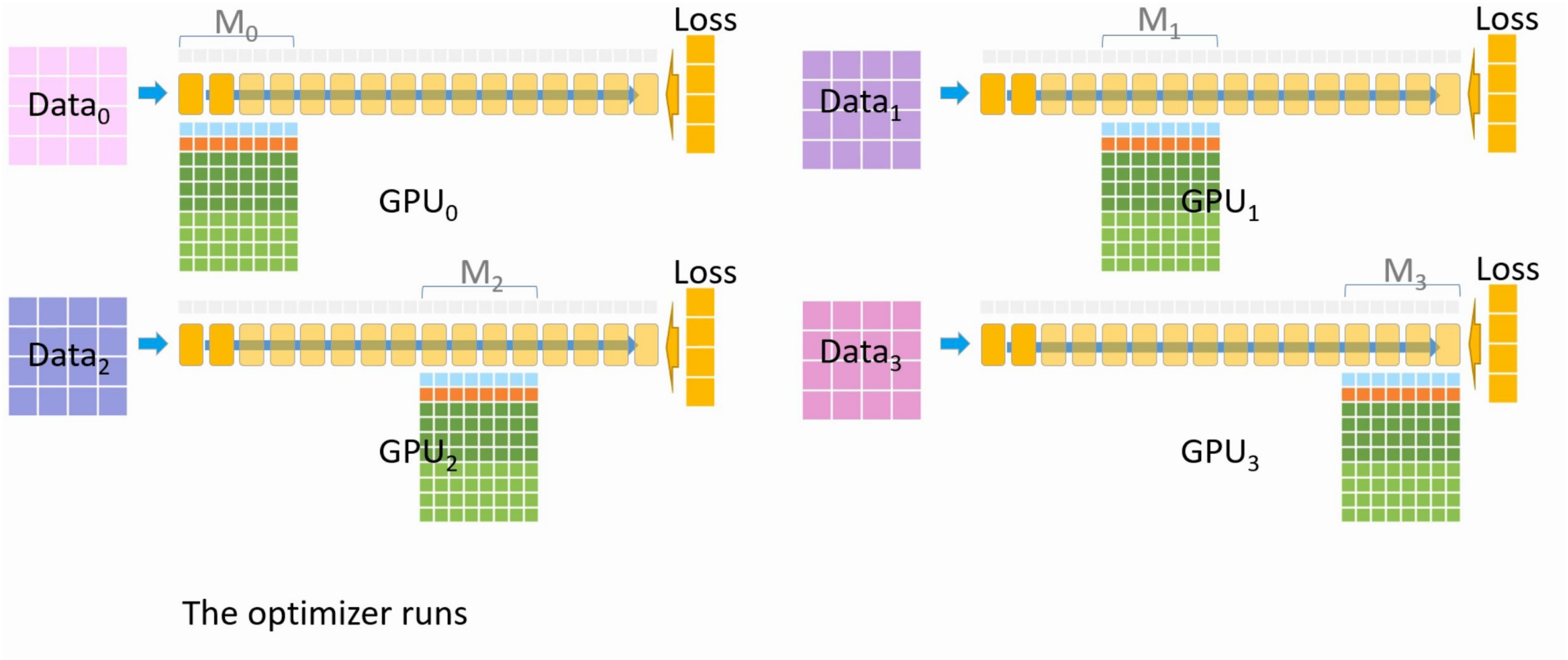
GPU_2 passes M_2's parameters to $GPU_{0,1,3}$ so they can run the backwards pass and compute gradients for M_2

Update Parameters For Local Partition

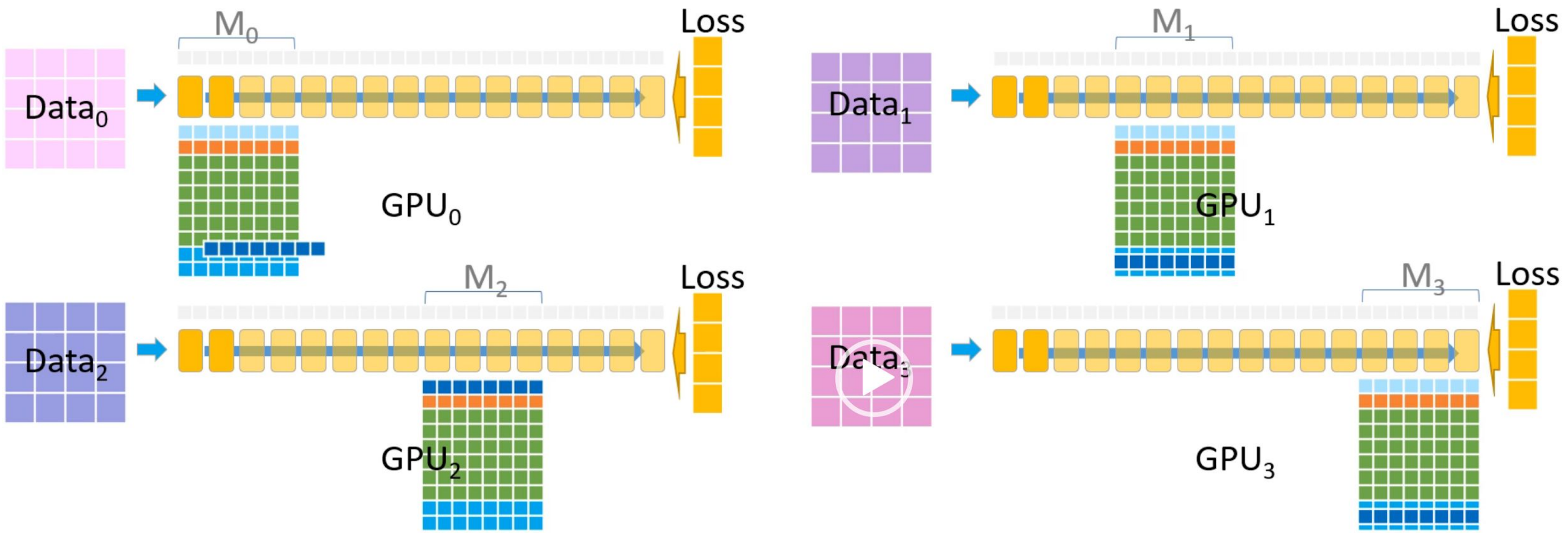


Now every GPU has its respective gradients (accumulated from all datasets)
We can compute the updated parameters

Optimizer Runs in Parallel

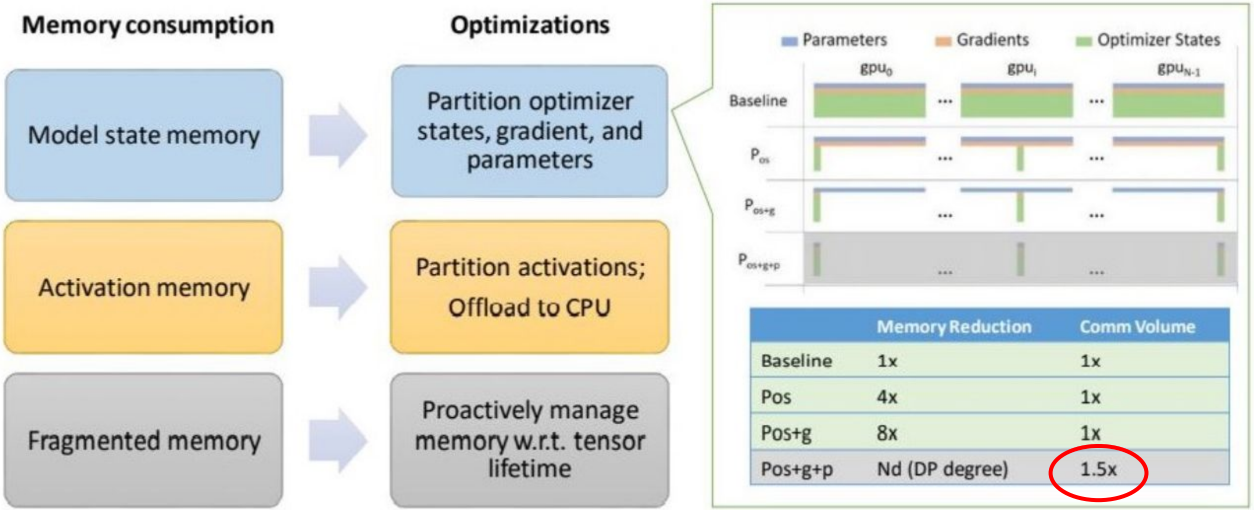


Iteration Complete



The fp16 weights become the model parameters for the next iteration
Training iteration complete!

Analysis

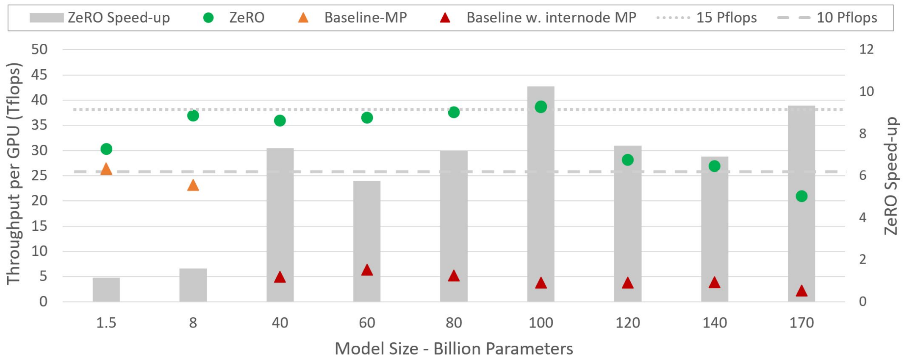


All-gather over parameters is spread over entire forward pass, but needs to happen again for backward pass as parameters are discarded.

Results

ZeRO vs. Megatron-LM (MP) and PyTorch Distributed Data Parallel (Baseline without MP)

400 V100 GPUs



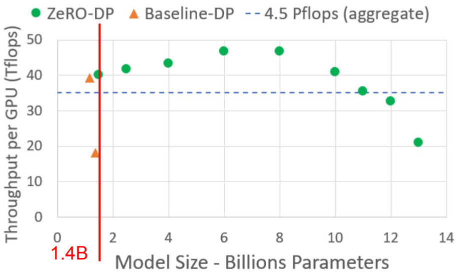


Figure 4: Max model throughput with *ZeRO*-DP.

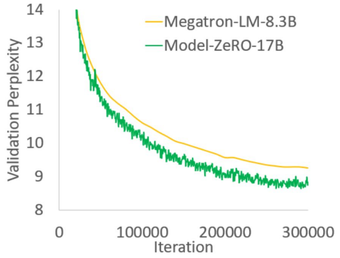
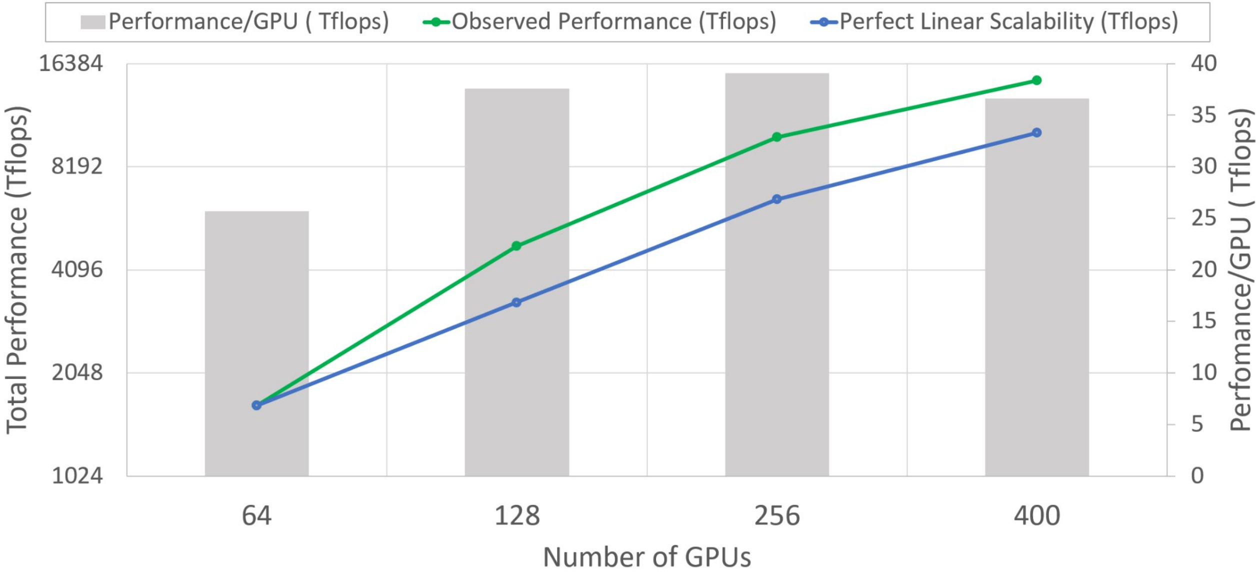


Figure 5: SOTA Turing-NLG enabled by *ZeRO*.

No MP, up to 13B parameters on 128 GPUs

Super-Linear Scalability - 60B parameters



Trillion PArameters Possible?

Theoretically yes, when combined with MP and with 1024 GPUs

- 16-way model parallelism (intra DGX-2 node)
- 64-way data parallelism

Turing-NLP: 17B

GPT-3: 175B

https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/