Distributed DNN Training

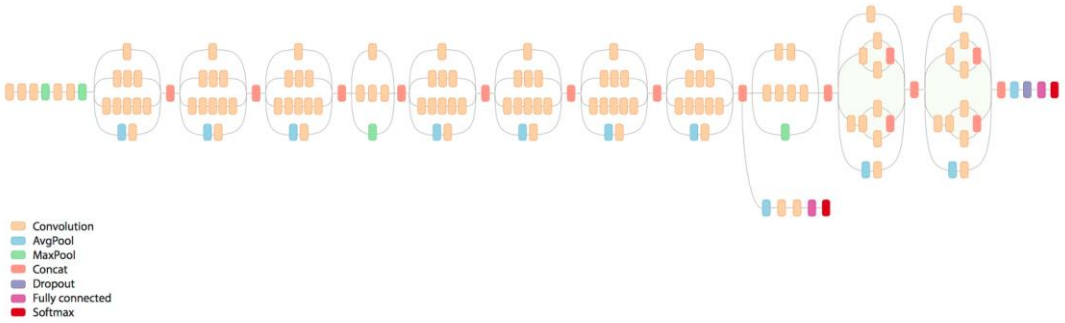
Advanced Topics in Distributed Systems

Tianze Wang (tianzew@kth.se)

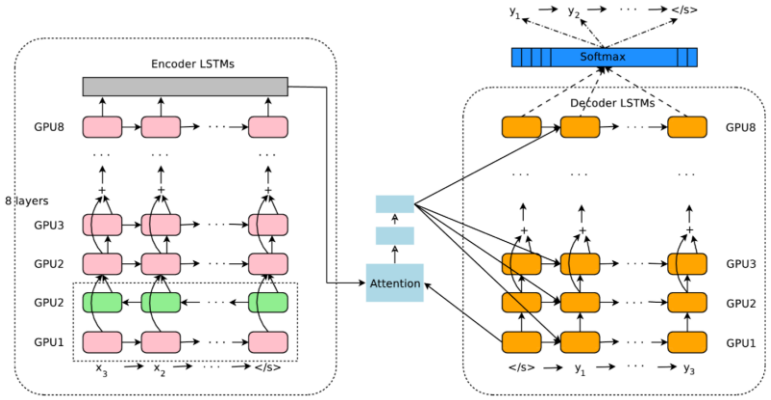
Outline

- A brief background
- Some paper reading
  - OptCNN
  - FlexFlow
  - P3
- A brief summary

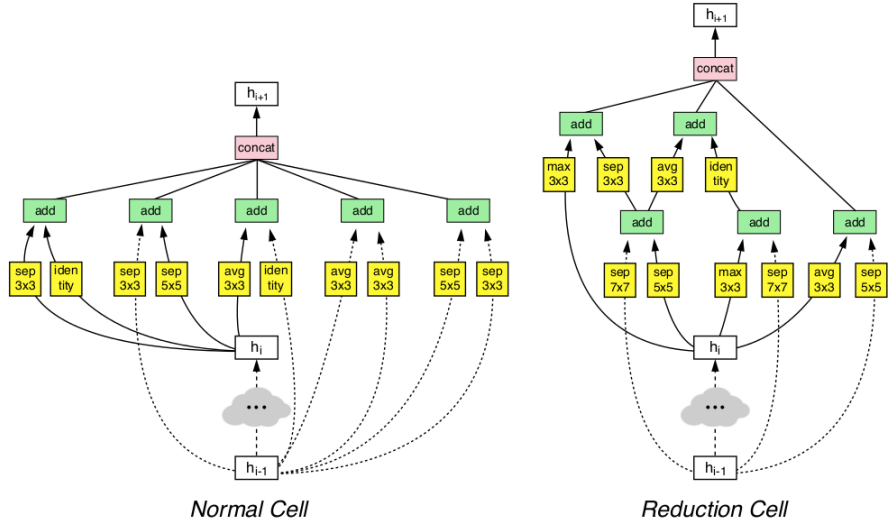
Deep Learning is everywhere



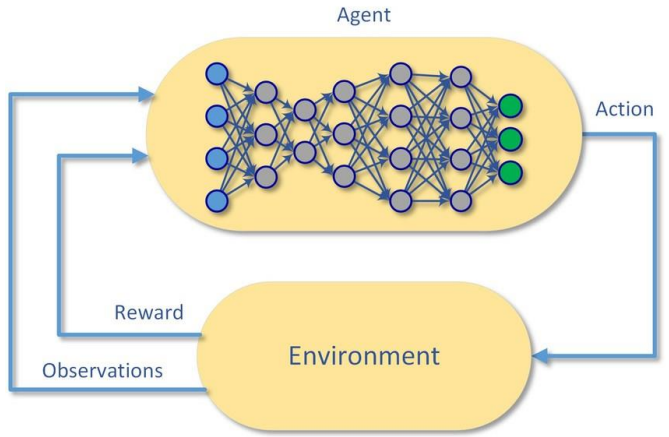
Convolutional Neural Networks



Recurrent Neural Networks



Neural Architecture Search



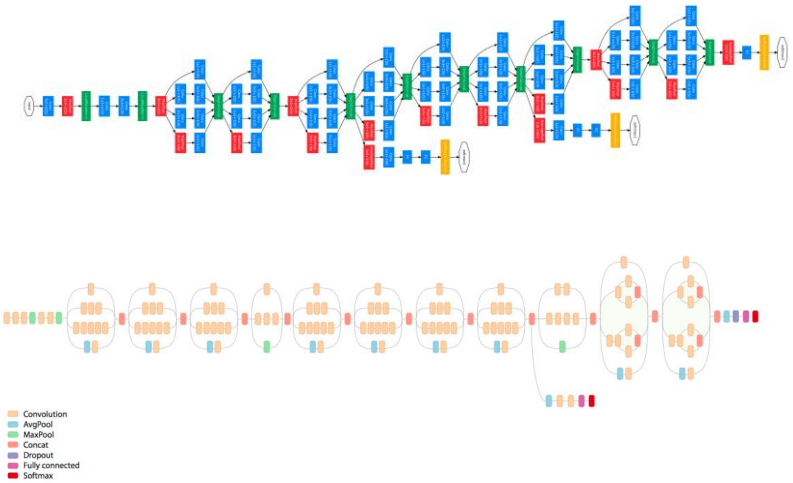
Reinforcement Learning

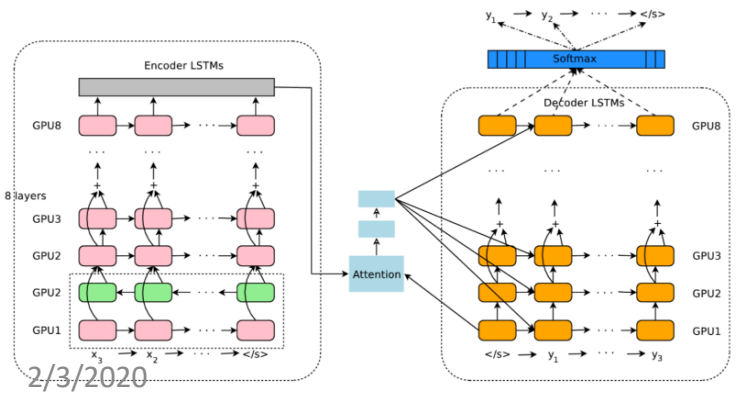
- Comes together with the “fancy magic” with Deep Learning models are the computational requirements.

- How to satisfy the computation requirements?
  - Build a strong single machine?
  - Parallelize the computation across multiple devices, nodes, clusters?

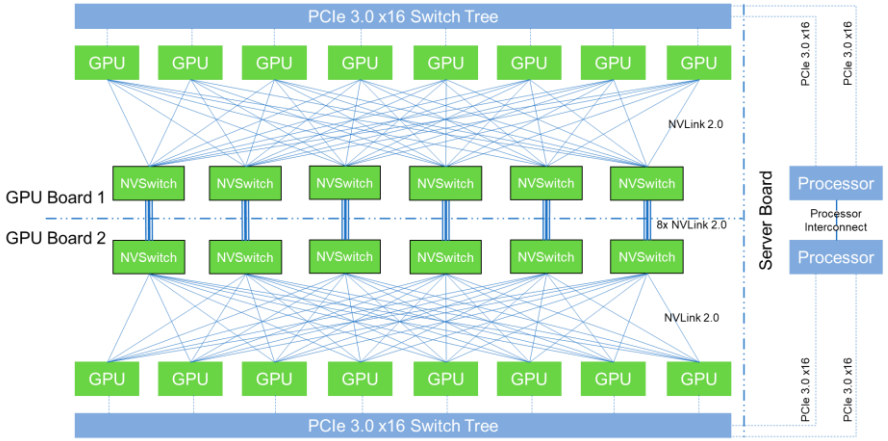
Deep learning deployment is challenging

Diverse and Complex DNN Models



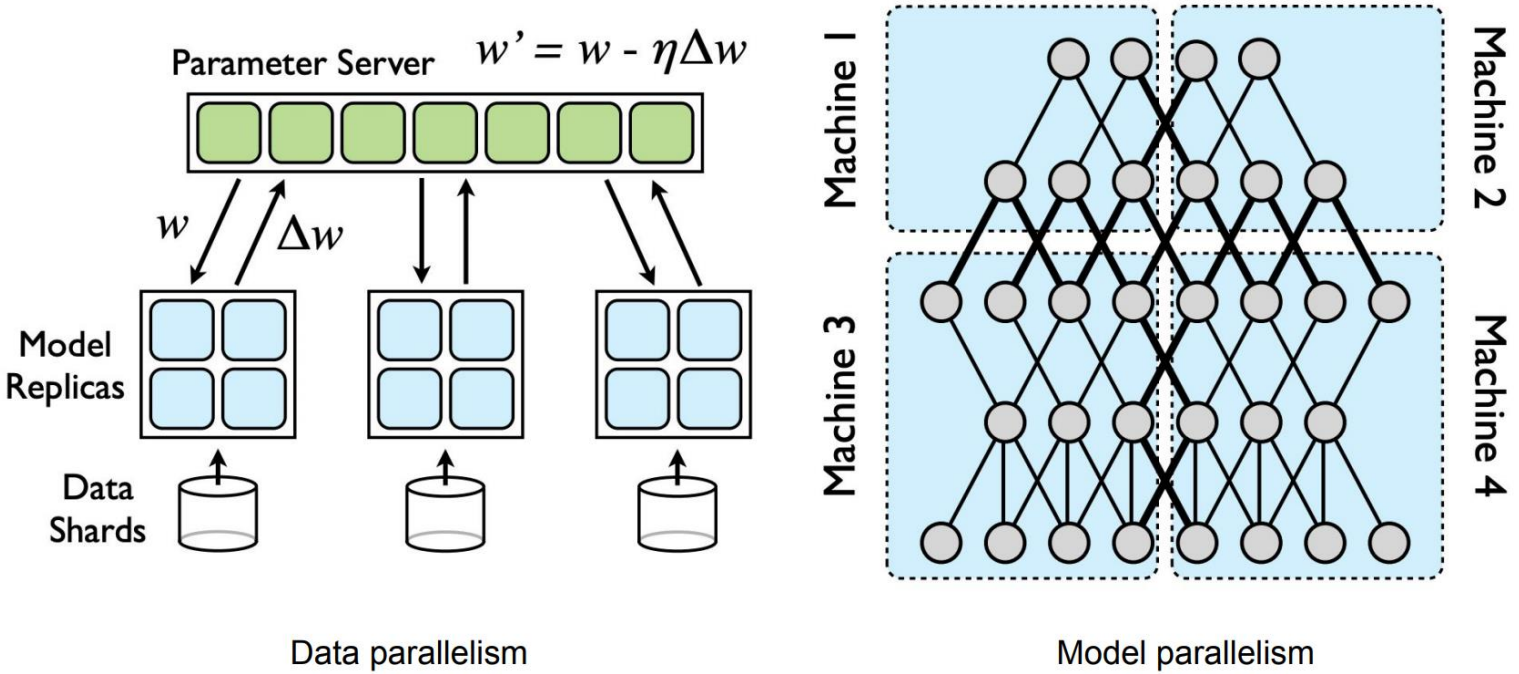


Distributed Heterogenous Hardware Platforms



Popular methods

- Data parallelism
  - efficient for compute-intensive operators with few parameters (e.g., convolution)
  - suboptimal for operators with a large number of parameters (e.g., embedding)
- Model parallelism
  - eliminates parameter synchronization between devices
  - but requires data transfer between operators



Images from **Large Scale Distributed Deep Networks** (Dean et al., 2012)

Popular methods

- System optimizations: (based on data parallelism)
  - Allreduce operation to optimize communication
  - Overlaps gradient synchronization with back propagation
- Network parameter reduction:
  - Weight pruning method that removes weak connection;
- Gradient compression:
  - Lossy compression like gradient quantization and sparse parameter synchronization.

Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks

Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken (ICML 2018)

Motivation of the paper

- Applying a single parallelization strategy to all layers in a network results in suboptimal runtime performance in large scale distributed training;
- Different layers in a network **may prefer** different parallelization strategies.
- Densely-connected layers with millions of parameters:
  - prefer model parallelism to reduce communication cost
- Convolutional layers:
  - typically prefer data parallelism to eliminate data transfers from the previous layers

Layer-wise parallelism

- The paper propose layer-wise parallelism:
  - Enable each layer in a network to use an individual parallelization strategy;
  - Perform the same computation for each layer as the original network;
  - A more comprehensive search space which includes data and model parallelism;
- The goal is:
  - to find the parallelization strategies for individual layers to jointly achieve the **best possible runtime performance** while maintaining the original network accuracy

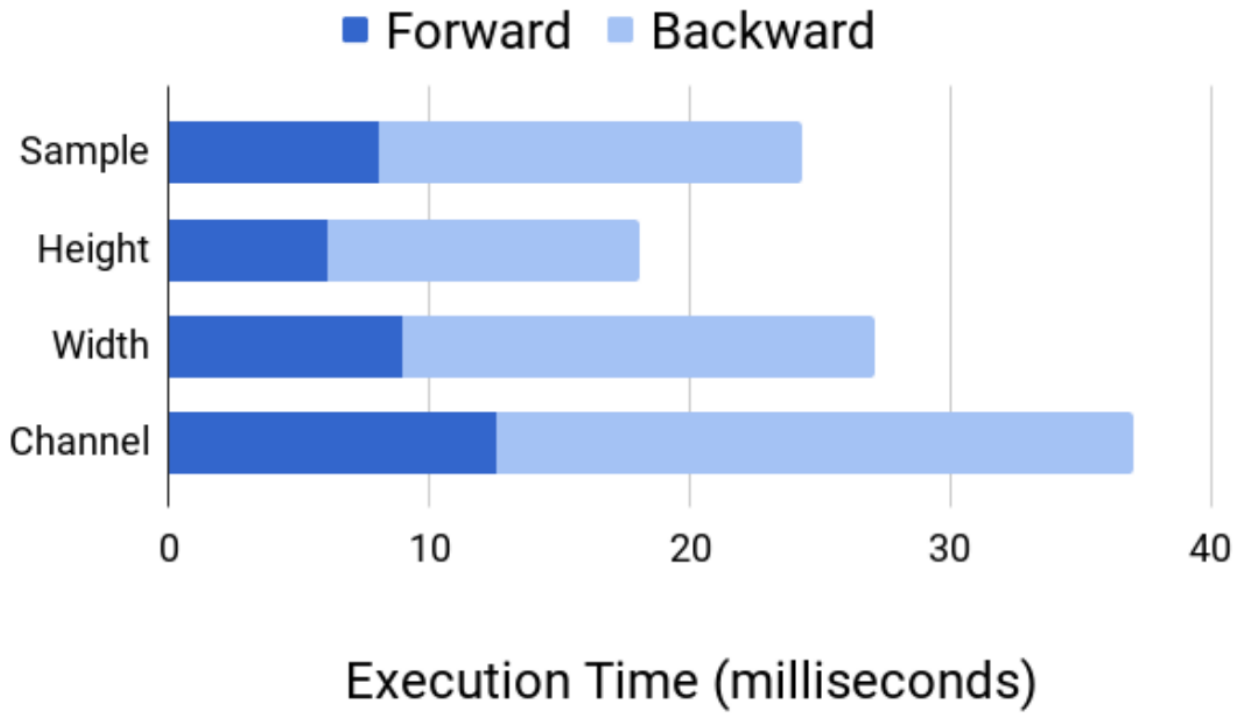
Hidden dimensions

- In standard CNNs for 2D images, data is commonly organized as 4-dimensional tensors (i.e., sample, height, width, and channel);
  - Sample dimension: partition the training dataset;
  - Height and width dimensions: partition the image/feature map;
  - Channel dimensions: different neurons for different output channel.

- To parallelize a layer we should:
  - Select from all dimensions;
  - Consider the degree of parallelism in each dimension.

Advantages of exploring hidden dimensions

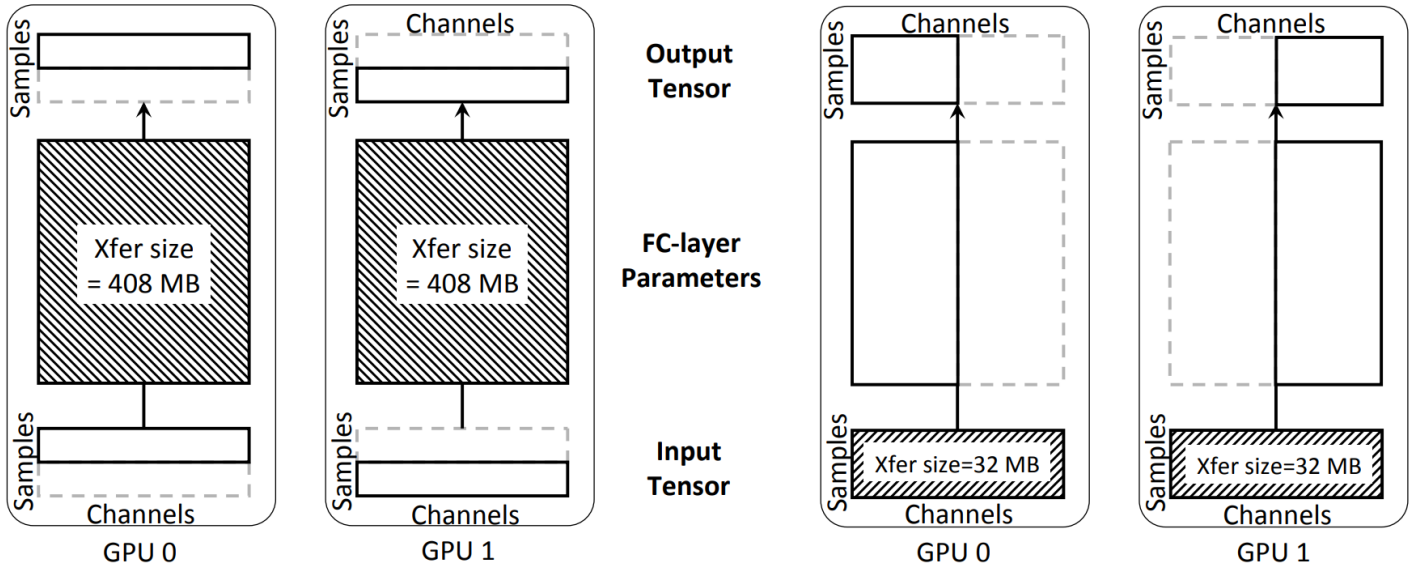
- Can reduce execution time.



Execution time for parallelizing a convolutional layer (Conv8 in VGG-16 (Simonyan & Zisserman, 2014)) on 4 GPUs by using different dimensions.

Advantages of exploring hidden dimensions

- Can reduce communication cost.



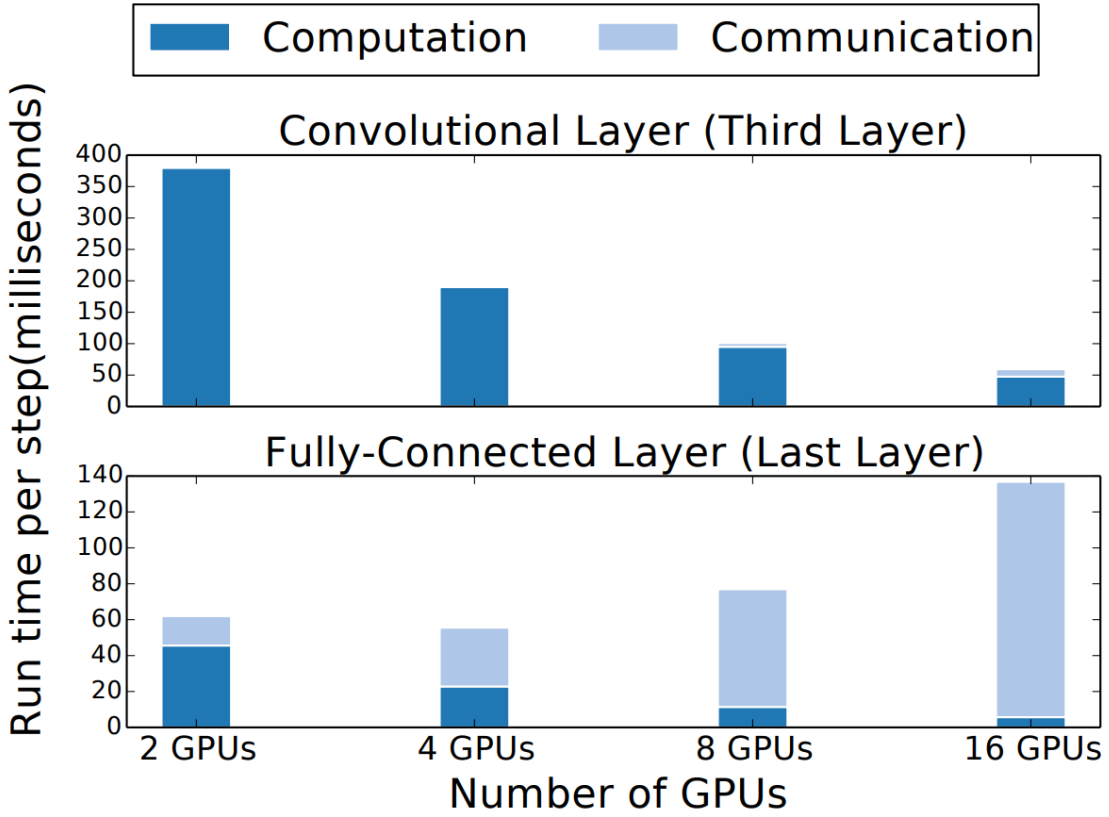
(a) Parallelism in the sample dimension.

(b) Parallelism in the channel dimension.

Different ways to parallelize the first fully-connected layer of VGG-16.
Rectangles with solid lines indicate tensors managed by the local GPU, while rectangles with dotted lines are tensors managed by a remote GPU.
The shadow rectangles indicate data transfers in each step.

Advantages of exploring hidden dimensions

- Can find the degree of parallelism that minimize the runtime.



Computation and communication time to process the third layer and the last layer of Inception-v3 using data parallelism

Problem definition

- Define the parallelization problem with two graphs:
  - Device graph D :
    - models all available hardware devices and the connections between them;
    - each node d_i is a device (e.g., a CPU or a GPU);
    - each edge (d_i, d_j) is an connection between d_i and d_j with communication bandwidth $b(d_i, d_j)$.
  - Computation graph G :
    - defines the neural network to be mapped onto the device graph.
    - each node $l_i \in G$ is a layer in the neural network;
    - Each edge $(l_i, l_j) \in G$ is a tensor that is an output of layer l_i and input of layer l_j.

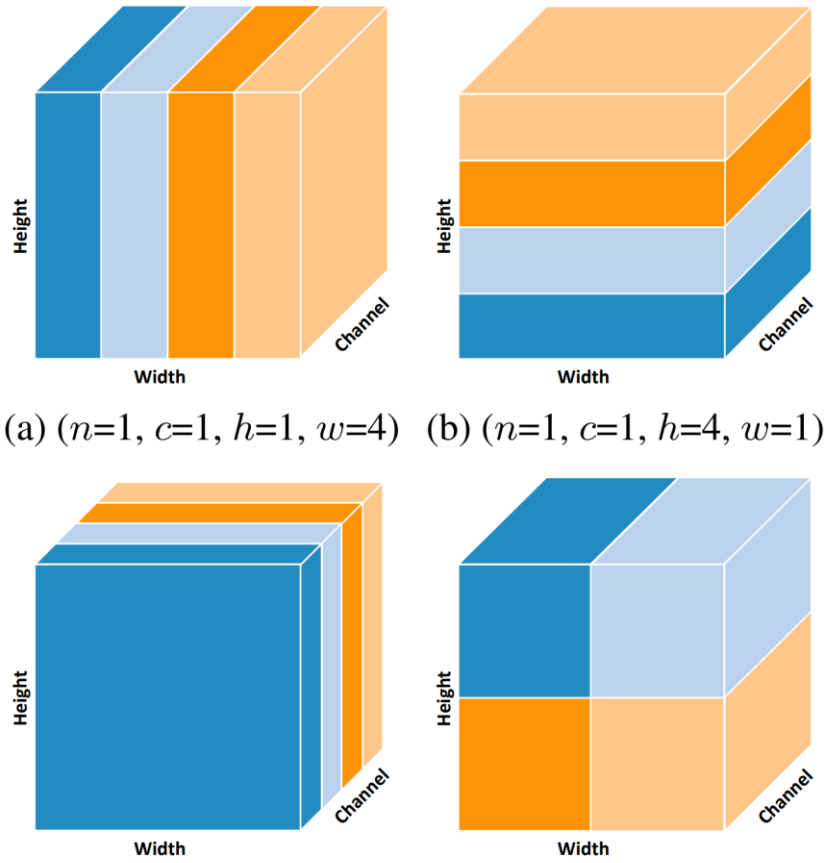
Solution definition

- The parallelization of a layer:
  - Assume that different devices can process the layer in parallel without any dependencies (compute **disjoin subsets** of the layer's output tensor);
  - The parallelization of a layer is defined by how its **output tensor** is partitioned.
  - For a layer l_i, the parallelizable dimensions P_i is the set of all divisible dimensions in its output tensor.

Layer	Parallelizable dimensions
Fully-connected	{sample, channel}
1D convolution/pooling	{sample, channel, length}
2D convolution/pooling	{sample, channel, height, width}
3D convolution/pooling	{sampe, channel, height, width, depth}

Solution definition

- A parallelization configuration c_i of a layer l_i defines how l_i is parallelized across different devices.
- For each parallelizable dimension in P_i , c_i includes a positive integer that describes the degree of parallelism in that dimension.
- We assume equal partitioning in each parallelizable dimension.
- A parallelization strategy S includes a configuration ci for each layer $l_i \in G$.
- Let $t(G, D, S)$ denote the per-iteration execution time to parallelize the computation graph G on the device graph D by using strategy S.



Example configurations to parallelize a 2D convolutional layer in a single dimension or combinations of multiple dimensions. The figure shows how each training sample is partitioned.

Cost model

Assumptions:

- The execution time of each layer is **predictable** with **low variance** and is largely independent of the contents of the input data;
- For each connection (d_i, d_j) between device d_i and d_j with bandwidth b, **transferring** a tensor of size s from d_i to d_j takes s/b time;
- The runtime system has **negligible overhead**. A device begins processing a layer as soon as its input tensors are available and the device has finished previous tasks.

Cost model

- The cost model:

$$t_O(G, D, S) = \sum_{l_i \in G} \{t_C(l_i, c_i) + t_S(l_i, c_i)\} + \sum_{e=(l_i, l_j) \in G} t_X(e, c_i, c_j)$$

the estimated per-step execution time for parallelization strategy S

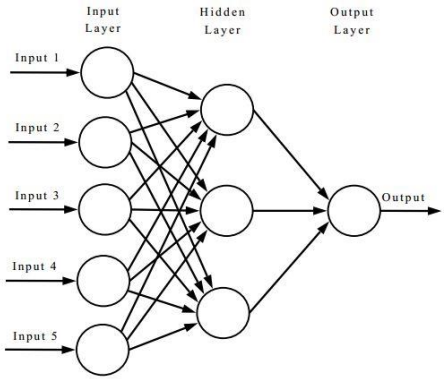
the time to process layer l_i under configuration c_i (includes both the forward and back propagation time)

the time to synchronize the parameters in layer l_i after back propagation

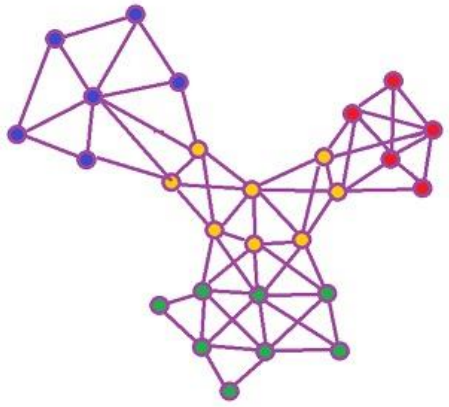
the time to transfer the input tensors to the target devices

Graph Search

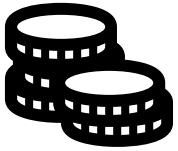
Computation Graph



Device Graph



Graph Search



Cost Model

Parallelization strategy

Graph Search

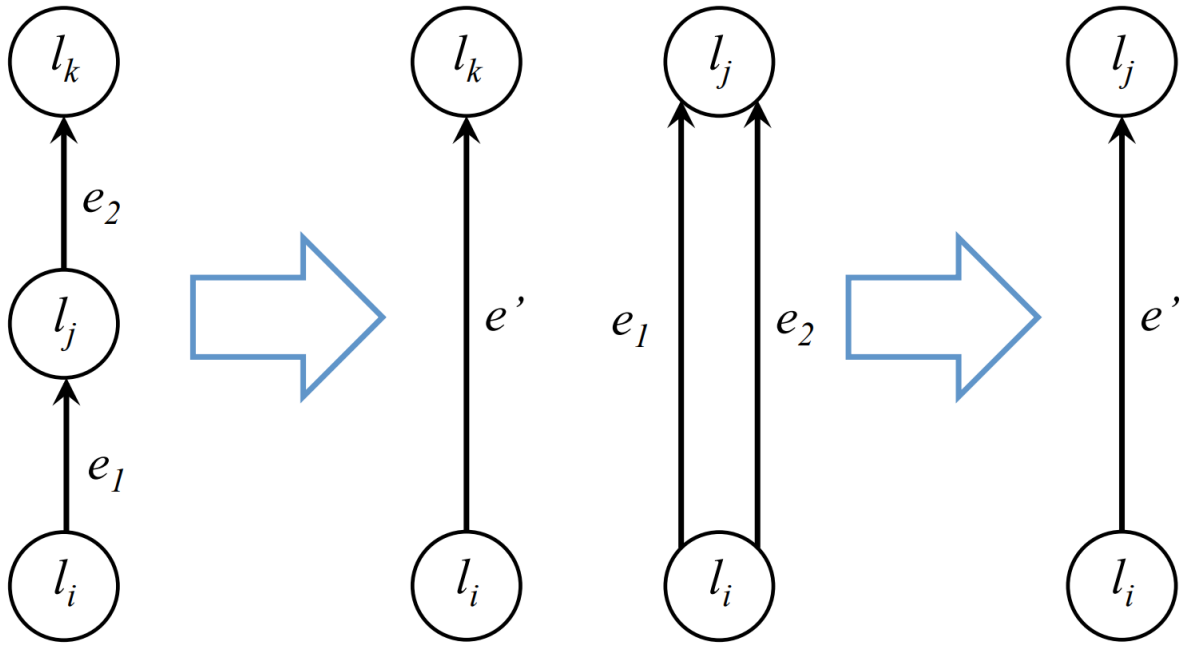
- Node elimination

$$t_X(e', c_i, c_k) = \min_{c_j}\{t_C(l_j, c_j) + t_S(l_j, c_j) + t_X(e_1, c_i, c_j) + t_X(e_2, c_j, c_k)\} \quad (2)$$

- Edge elimination

$$t_X(e', c_i, c_j) = t_X(e_1, c_i, c_j) + t_X(e_2, c_i, c_j) \quad (3)$$

- Theorem 1 & 2 (see the paper) show that given an optimal parallelization strategy for the modified graph, one can easily construct an optimal strategy for the original graph.



(a) Node elimination. (b) Edge elimination.

Graph Search

Algorithm 1 Finding Optimal Parallelization Strategy $\mathcal{S}$.

Input: A computation graph $\mathcal{G}$, a device graph $\mathcal{D}$, and precomputed cost functions (i.e., $t_c(\cdot)$, $t_s(\cdot)$ and $t_x(\cdot)$)
Output: A parallelization strategy $\mathcal{S}$ minimizing $t_o(\mathcal{G}, \mathcal{D}, \mathcal{S})$

$\mathcal{G}^{(0)} = \mathcal{G}$
$m = 0$
while true **do**
    $\mathcal{G}^{(m+1)}$ = NODEELIMINATION($\mathcal{G}^{(m)}$)
    $\mathcal{G}^{(m+2)}$ = EDGEELIMINATION($\mathcal{G}^{(m+1)}$)
    if $\mathcal{G}^{(m+2)} = \mathcal{G}^{(m)}$ **then**
        break
    end if
    $m = m + 2$
end while
Find the optimal strategy $\mathcal{S}^{(m)}$ for $\mathcal{G}^{(m)}$ by enumerating all possible candidate strategies
for $i = m$-1 to 0 **do**
    if $\mathcal{G}^{(i+1)}$ = NODEELIMINATION($\mathcal{G}^{(i)}$) **then**
        ▷ Assume l_j is the node eliminated from $\mathcal{G}^{(i)}$
        Find c_j that minimizes Equation 1
        $\mathcal{S}^{(i)} = \mathcal{S}^{(i+1)} + c_j$
    else
        $\mathcal{S}^{(i)} = \mathcal{S}^{(i+1)}$
    end if
end for
return $\mathcal{S}^{(0)}$

Iteratively simplify an input computation graph

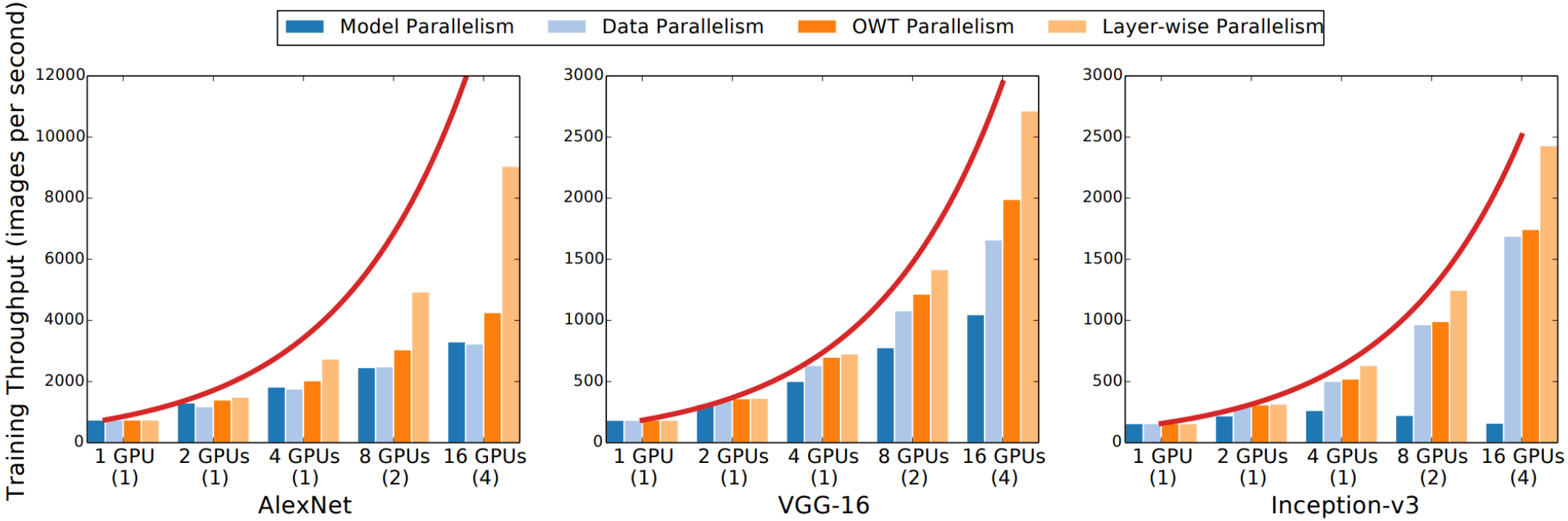
Find the optimal strategy of the simplified graph

Iteratively undo the elimination to decide the configurations for the eliminated nodes

Experiments

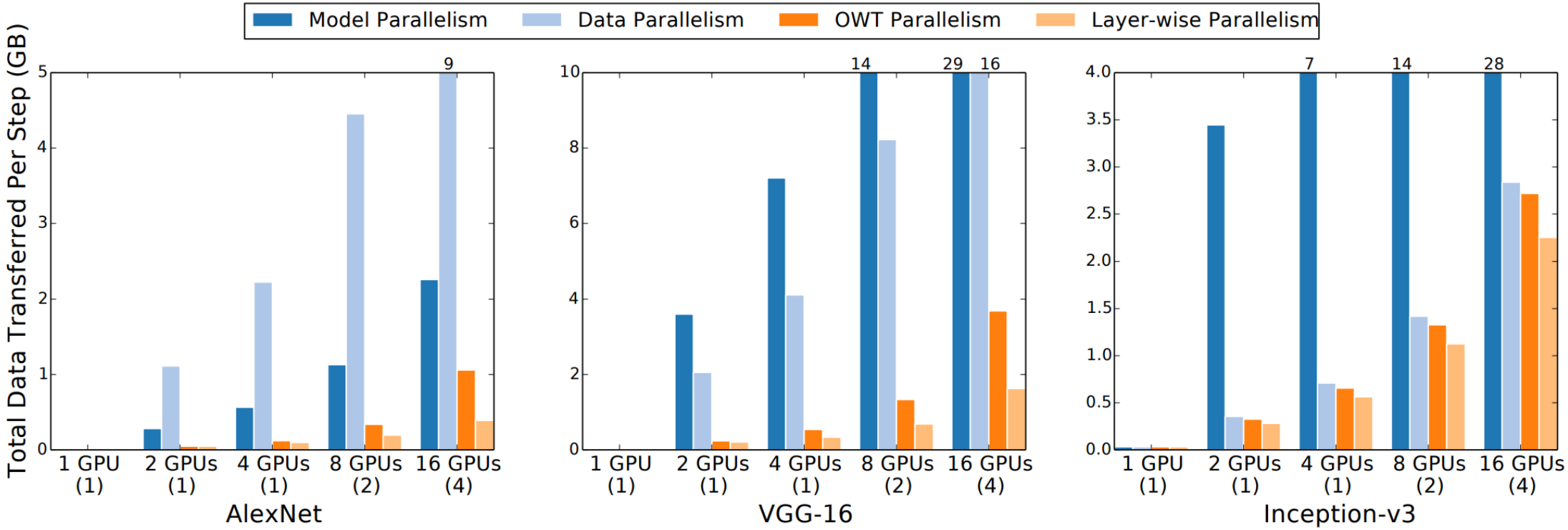
- Implemented in
  - Legion, cuDNN, and cuBLAS.
- Benchmarked on
  - AlexNet, VGG-16, and Inception-v3.
- Used the ImageNet-1K dataset.
- Compared between:
  - Data parallelism, model parallelism, OWT parallelism, and layer-wise parallelism.
- Hardware:
  - a GPU cluster with 4 compute nodes, each of which is equipped with two Intel 10-core E5-2600 CPUs, 256G main memory, and four NVIDIA Tesla P100 GPUs. GPUs on the same node are connected by NVLink, and nodes are connected over 100Gb/s EDR Infiniband. We use synchronous training and a per-GPU batch size of 32 for all experiments.

Evaluation – training throughput



Training throughput (i.e., number of images processed per second) with different parallelization strategies (higher is better). Numbers in parenthesis are the number of compute nodes used in the experiments. The red lines show the training throughput in linear scale (ideal case).

Evaluation – communication cost



Communication cost (i.e., data transferred in each step) with different parallelization strategies (lower is better).

Analysis of Optimal Parallelization Strategies

- For the beginning layers of a CNN with large height/width dimensions and a small channel dimension => data parallelism on all available devices;
- Deeper layers in a CNN tend to have smaller height/width dimensions and a larger channel dimension. => An optimal strategy adaptively reduces the number of devices for these layers and opportunistically uses parallelism in the height/width dimensions;
- For densely-connected layers, => model parallelism on a small number of devices.

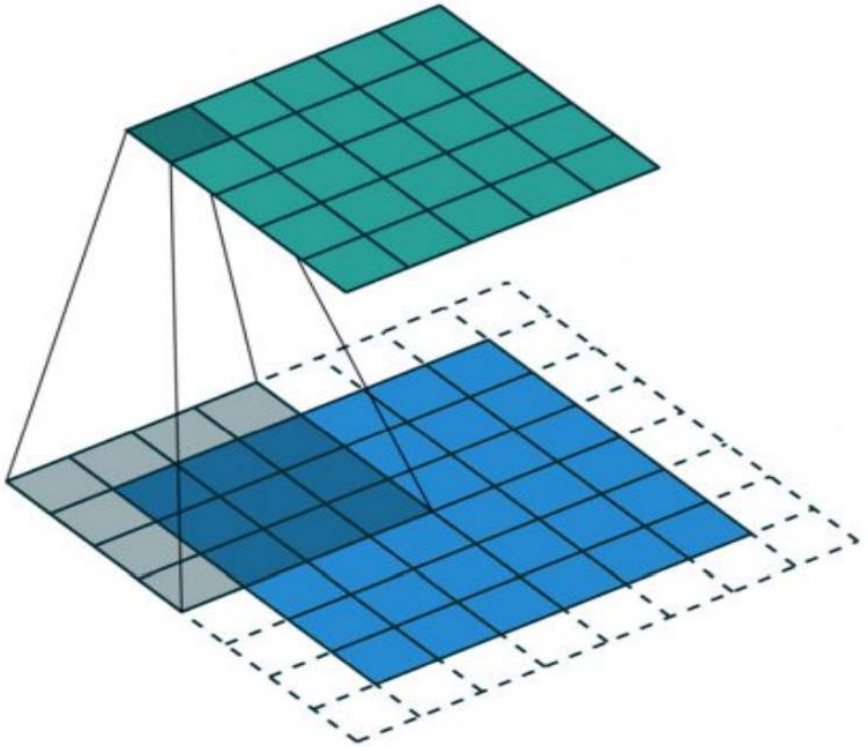
Beyond Data and Model Parallelism for Deep Neural Networks

- Search-Based Approaches to Accelerate Deep Learning

Zhihao Jia, Matei Zaharia, and Alex Aiken (SysML 2019)

Types of parallelism

- Tensorflow, PyTorch, Caffe2 are mainly based on data and model parallelism
- Something deep learning frameworks don’t exploit is operation level parallelism. The convolution operation can be distributed along the channel or spatial dimensions.



Current Approaches: Data and Model Parallelism

- **Data parallelism** is the default strategy in existing DNN frameworks
- Manually-designed strategies [1, 2]
  - **Combine data and model parallelism** to accelerate specific DNNs
- Automatic generated strategies
  - ColocRL [3] uses RL to find device placement for **model parallelism**

Exploring dimensions beyond data and model parallelism can further accelerate DNN training (by up to 3.3x)

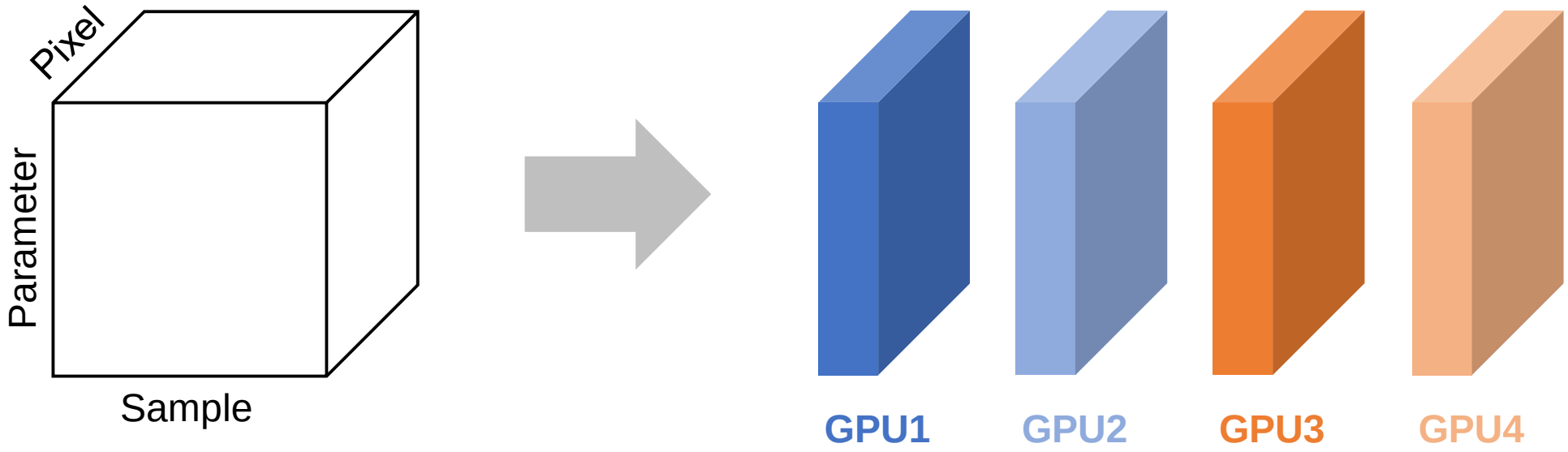
[1] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. 2014
[2] Wu et. al. Google's neural machine translation system: Bridging the gap between human and machine translation. 2016
[3] Mirhoseini et. al. Device placement optimization with reinforcement learning. 2017

The SOAP Search Space

- **S**amples
- **O**perators
- **A**ttributes
- **P**arameters

The SOAP Search Space

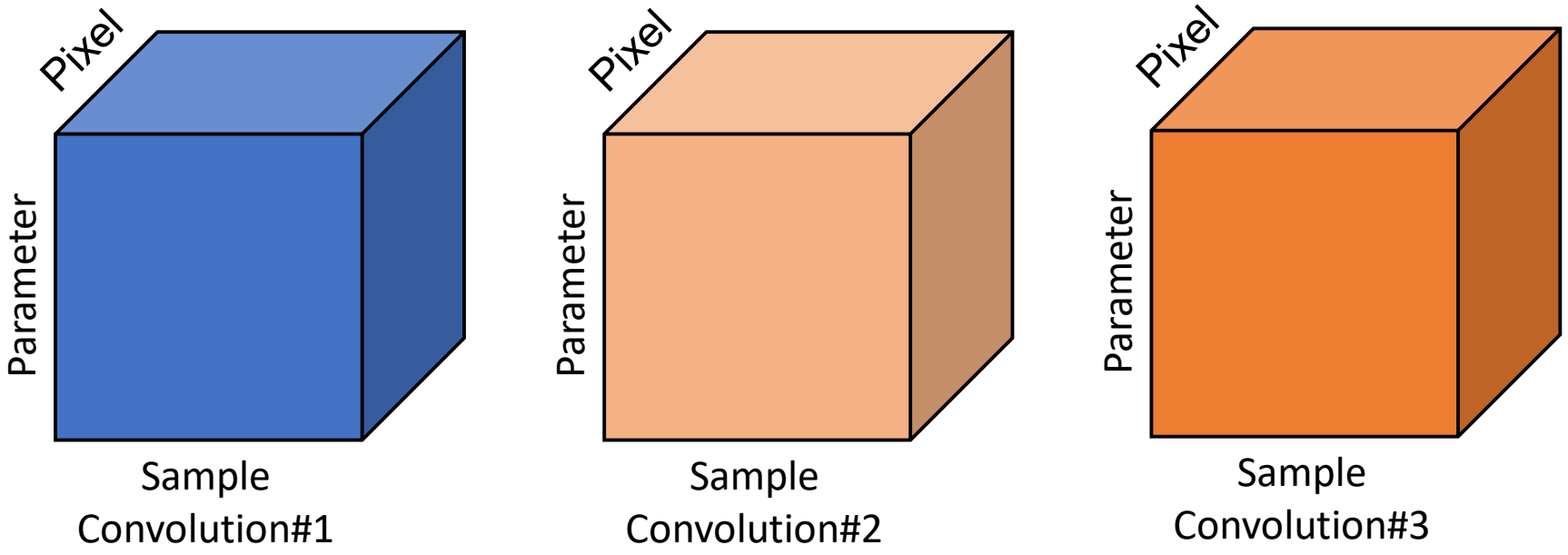
- **S**amples: partitioning training samples (Data Parallelism)
- **O**perators
- **A**ttributes
- **P**arameters



Parallelizing a 1D convolution

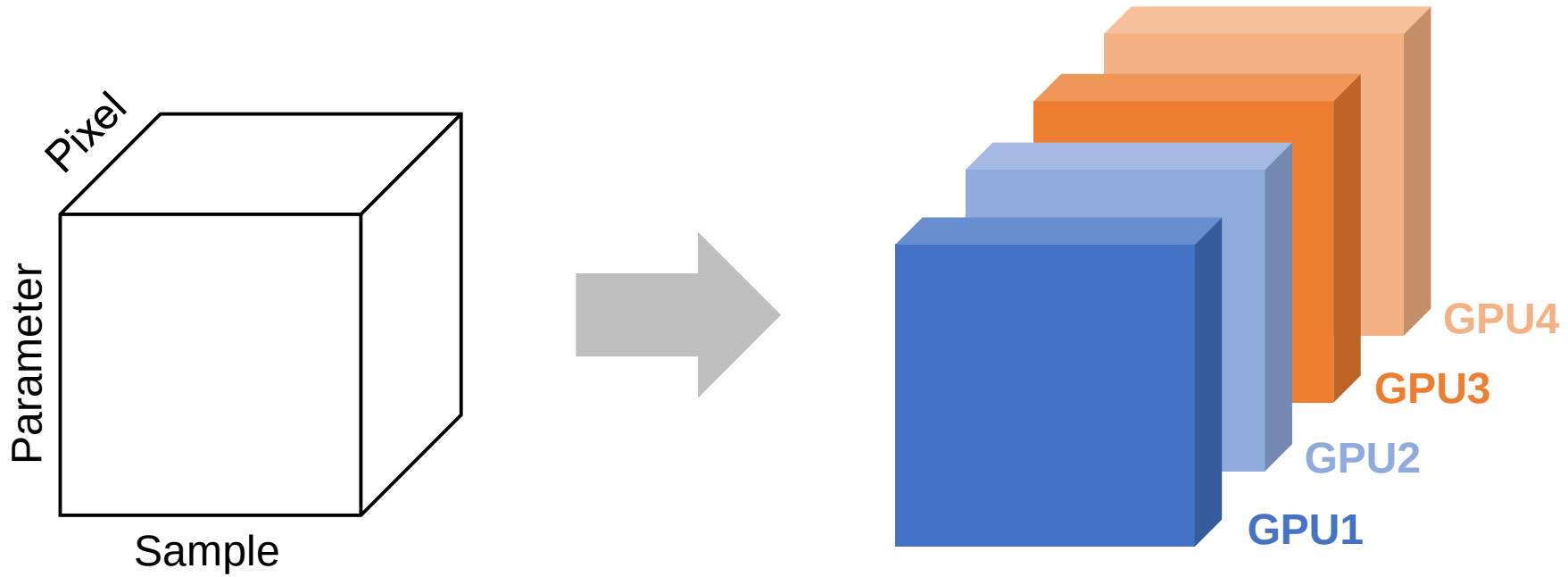
The SOAP Search Space

- **S**amples: partitioning training samples (Data Parallelism)
- **O**perators: partitioning DNN operators (Model Parallelism)
- **A**ttributes
- **P**arameters



The SOAP Search Space

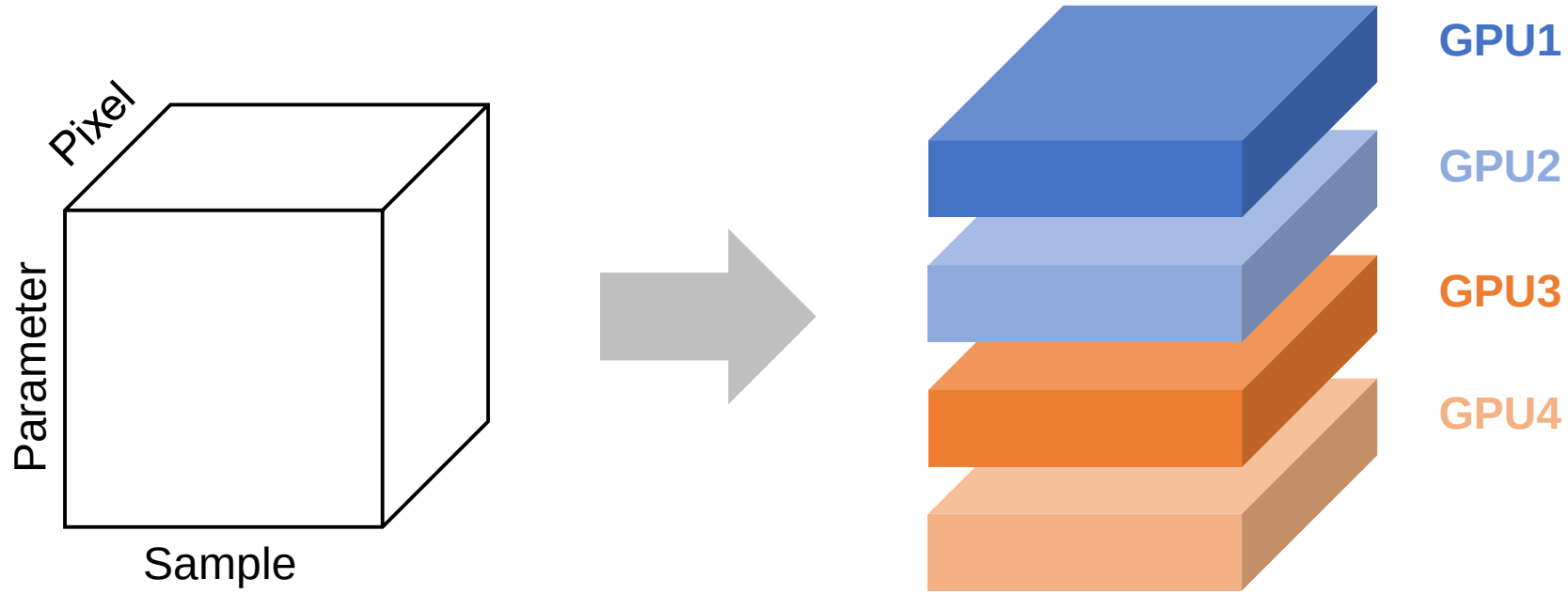
- **S**amples: partitioning training samples (Data Parallelism)
- **O**perators: partitioning DNN operators (Model Parallelism)
- **A**ttributes: partitioning attributes in a sample (e.g., different pixels)
- **P**arameters



Parallelizing a 1D convolution

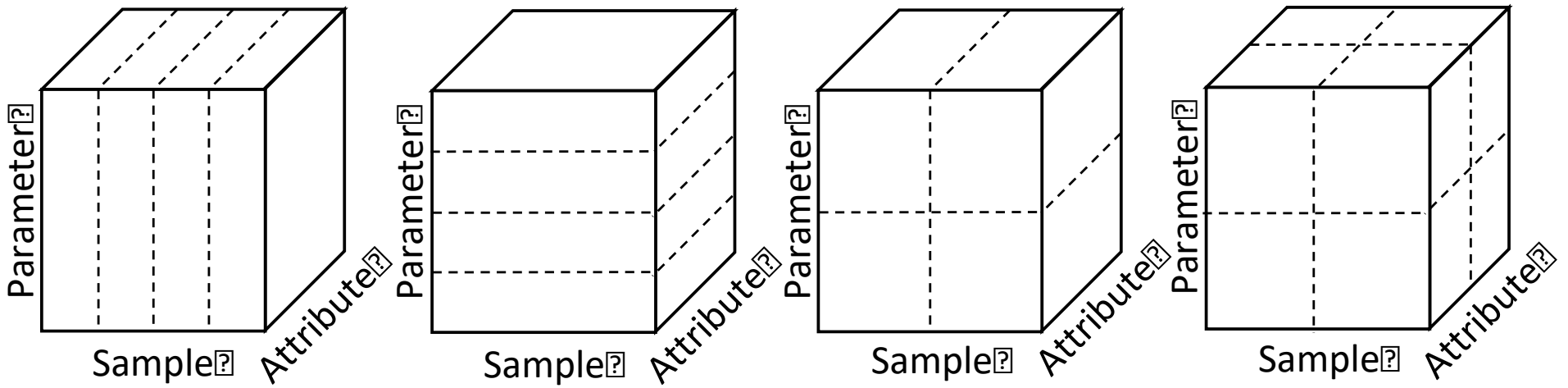
The SOAP Search Space

- **S**amples: partitioning training samples (Data Parallelism)
- **O**perators: partitioning DNN operators (Model Parallelism)
- **A**ttributes: partitioning attributes in a sample (e.g., different pixels)
- **P**arameters: partitioning parameters in an operator



Parallelizing a 1D convolution

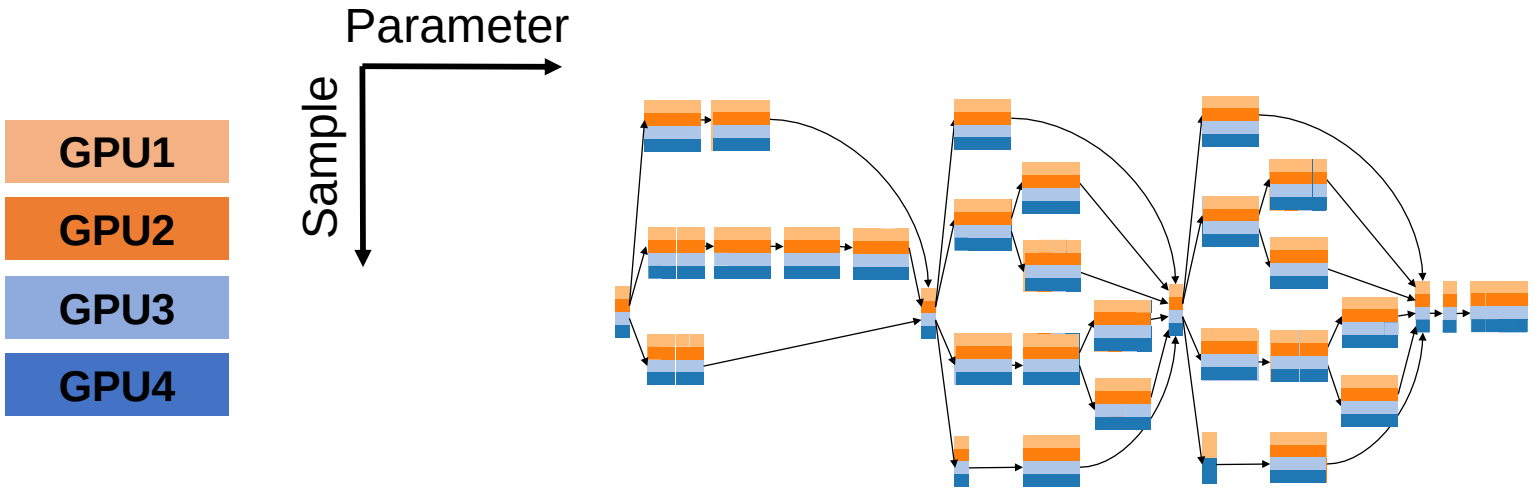
Hybrid Parallelism in SOAP



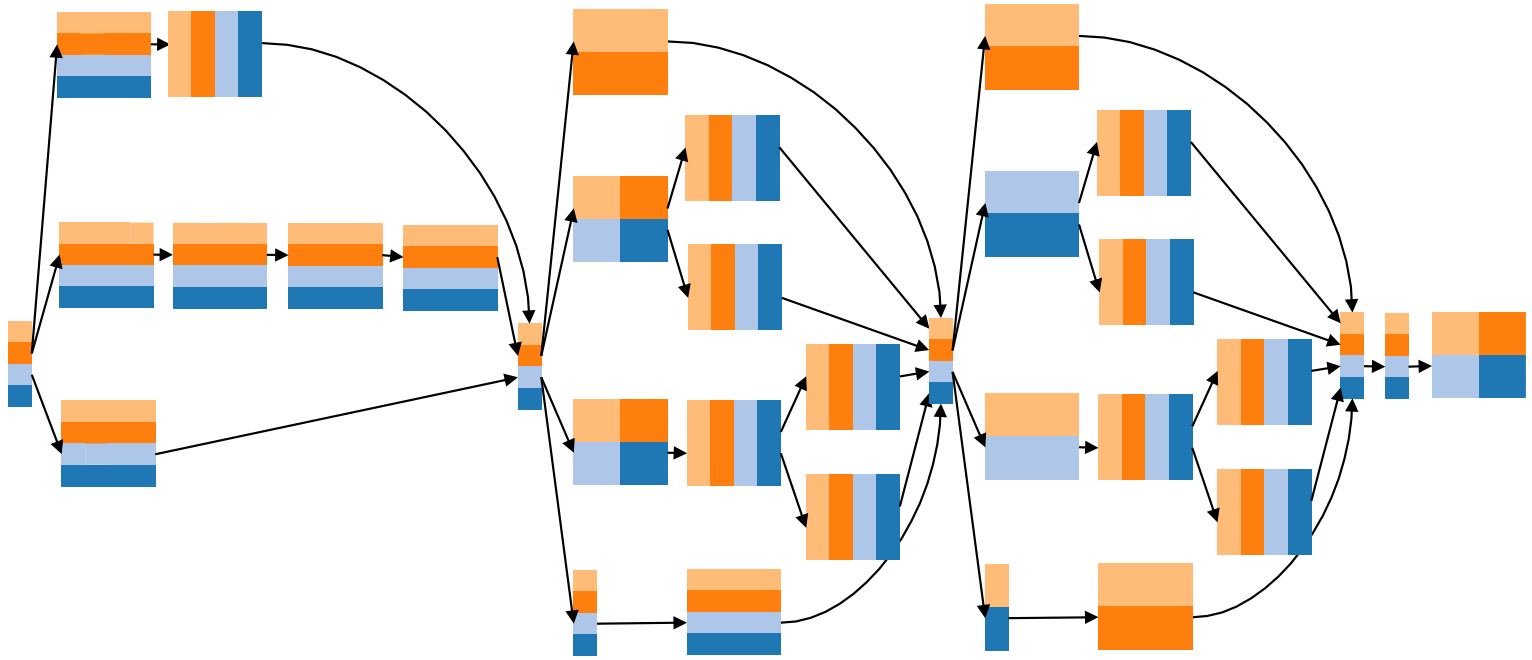
Example parallelization strategies for 1D convolution

Different strategies perform the same computation.

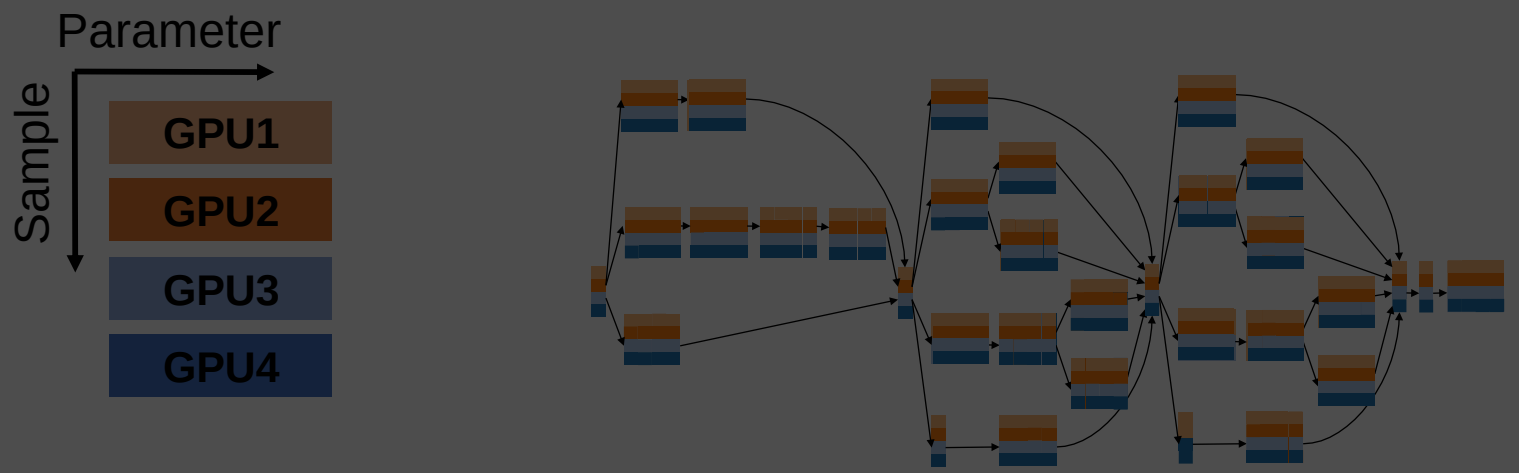
Parallelization Approach	Sample	Operator	Attribute	Parameter
Data Parallelism	✓			
Model Parallelism		✓		✓
Manually-designed strategies				
(Krizhevsky, 2012)	✓			✓
(Wu et al., 2014)	✓	✓		
Mesh-TensorFlow	✓			✓
Automatic generated strategies				
ColocRL		✓		
Tofu and SoyBean	✓			✓
Gpipe and PipeDream	✓	✓		
The SOAP search space				
Our work	✓	✓	✓	✓



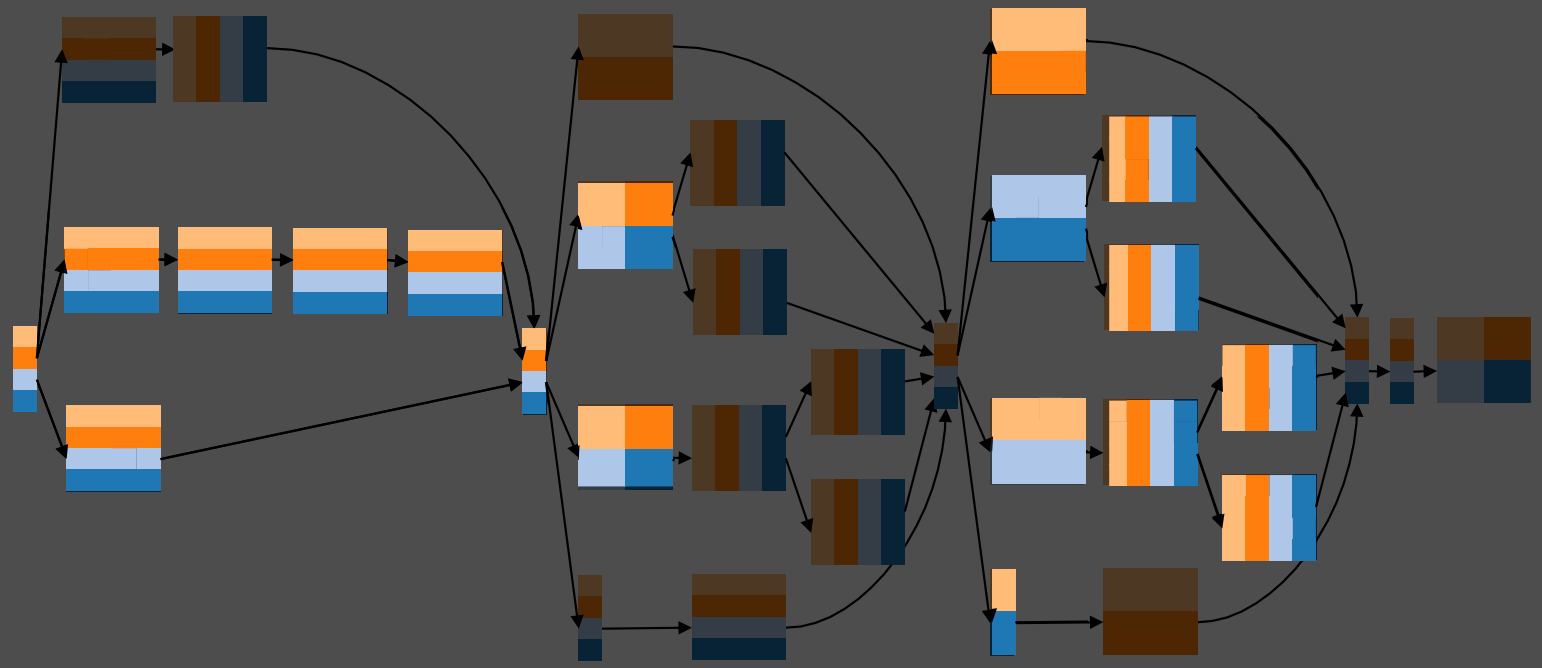
Data parallelism



A possible parallelization strategy in the SOAP search space



Data parallelism



A possible parallelization strategy in the SOAP search space

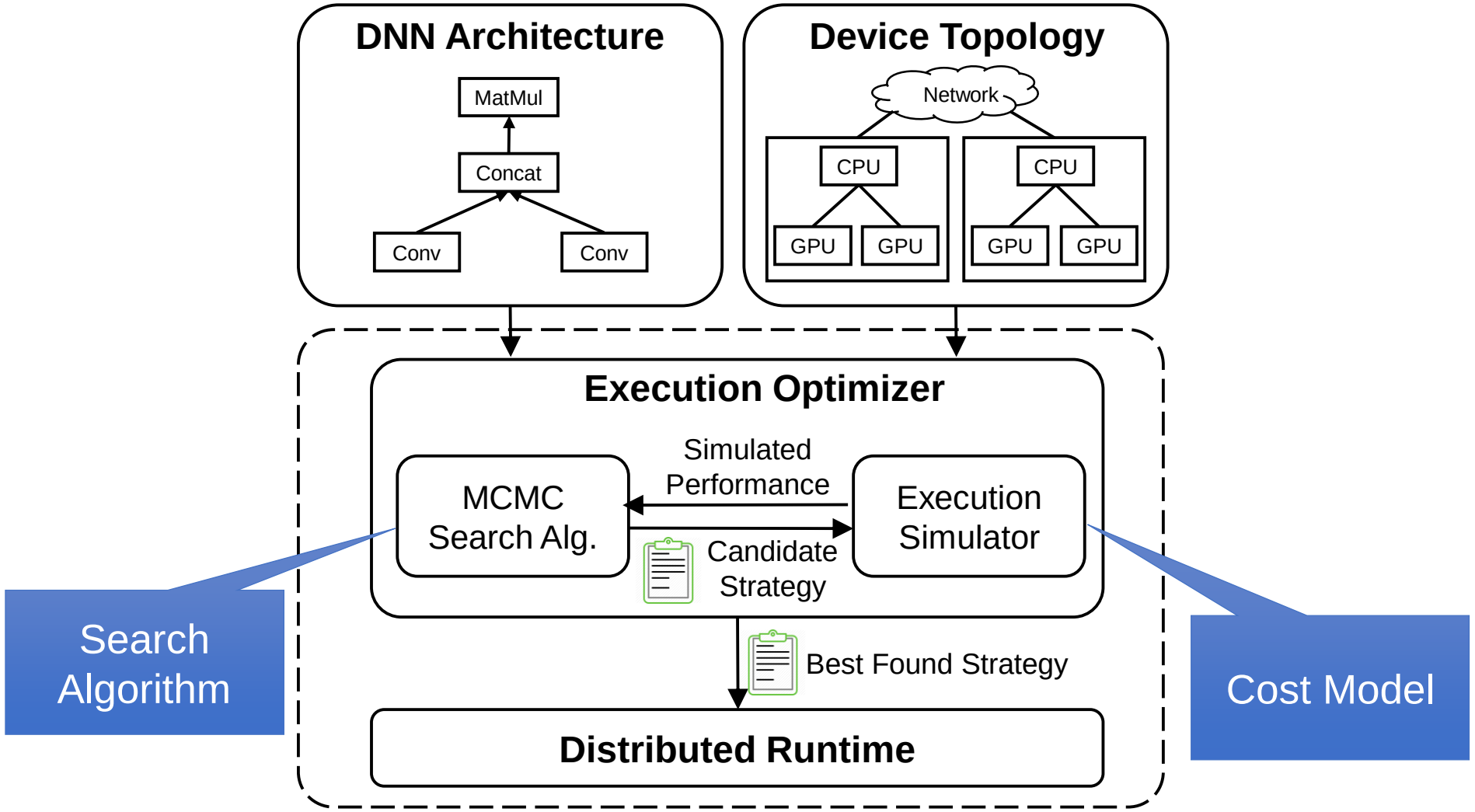
Search-Based Optimizations

A **search space** of possible strategies + A **cost model** and a **search algorithm** = Optimized strategies

Challenge 1:
How to build a search space including optimized strategies?

Challenge 2:
How to efficiently explore the search space?

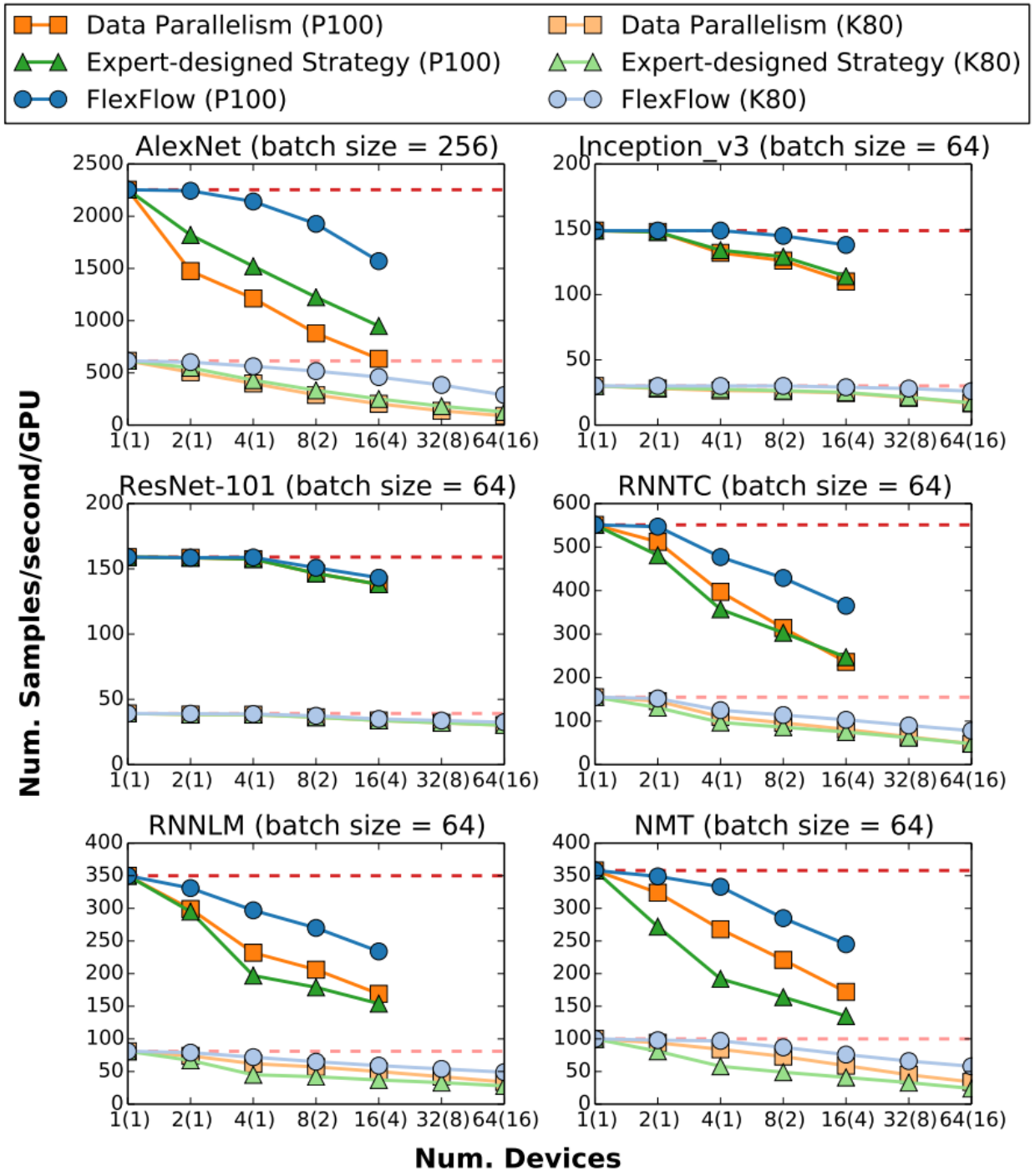
FlexFlow


DNN Architecture
MatMul
Concat
Conv
Conv
Device Topology
Network
CPU
CPU
GPU
GPU
GPU
GPU
Execution Optimizer
Simulated Performance
MCMC Search Alg.
Execution Simulator
Candidate Strategy
Best Found Strategy
Distributed Runtime
Search Algorithm
Cost Model

Evaluation: per GPU

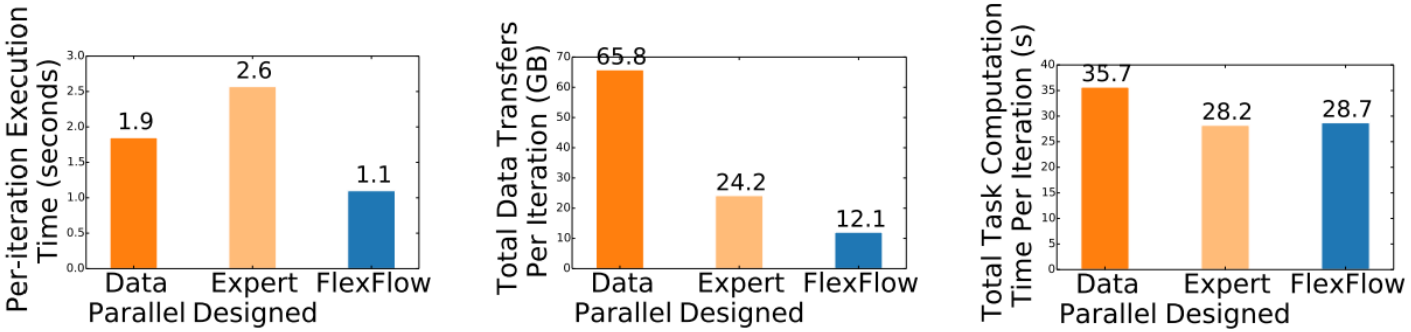
- AlexNet, Inception-v3, and ResNet-101 are three CNNs that achieved the best accuracy in the ILSVRC competitions.
- RNNTC, RNNLM and NMT are sequence-to-sequence RNN models for text classification, language modeling, and neural machine translation, respectively.

- For ResNet-101, FlexFlow finds strategies similar to data parallelism (except using model parallelism on a single node for the last fully-connected layer) and there fore achieves similar parallelization performance.
- For other DNN benchmarks, FlexFlow finds more efficient strategies than the baselines and achieves 1.3-3.3x speedup.



Evaluation:

- Reduces the overall communication costs. (Figure 7b)
- Reduces overall task computation time. (Figure 7c)
- The expert-designed strategy achieves slightly better total task computation time than FlexFlow. However, this is achieved by using model parallelism on each node, which disables any parallelism within each operator and results in imbalanced workloads. As a result, the expert-designed strategy achieves even worse execution performance than data parallelism (see Figure 7a).



(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.

Evaluation: End-to-End Performance

- Train Inception-v3 on the ImageNet dataset until the model reaches the single-crop top-1 accuracy of 72% on the validation set.
- FlexFlow reduces the training time by **38%** compared to TensorFlow

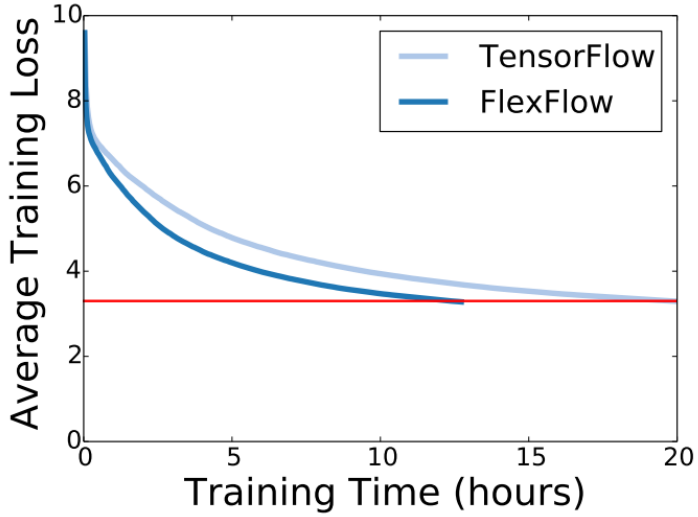
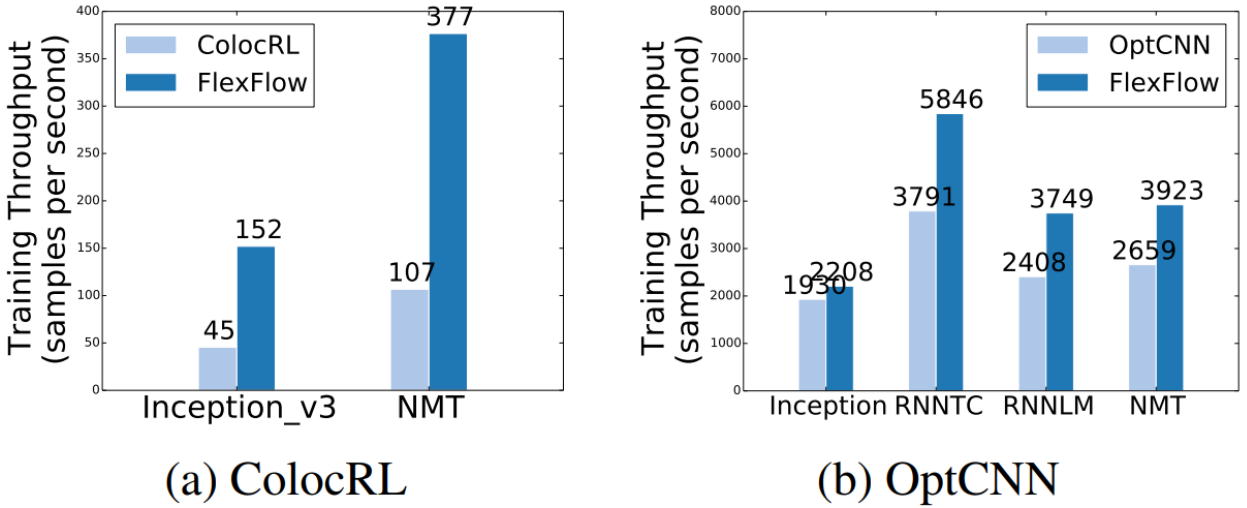


Figure 8. Training curves of Inception-v3 in different systems. The model is trained on 16 P100 GPUs (4 nodes).

Evaluation: other frameworks

- The parallelization strategies found by FlexFlow achieve **3.4 - 3.8×** speedup compared to ColocRL.
- FlexFlow and OptCNN found the same parallelization strategies for AlexNet and ResNet with linear operator graphs.
- For DNNs with non-linear operator graphs, FlexFlow achieves **1.2-1.6×** speedup compared to OptCNN by using parallelization strategies that exploit parallelism across different operators



(a) ColocRL (b) OptCNN

Figure 9. Comparison among the parallelization strategies found by different automated frameworks.

Assumptions of the simulator

The simulator depends on the following assumptions:

A1. The execution time of each task is predictable with low variance and is independent of the contents of input tensors.
A2. For each connection (di , dj) between device di and dj with bandwidth b, transferring a tensor of size s from di to dj takes s/b time (i.e., the communication bandwidth can be fully utilized).
A3. Each device processes the assigned tasks with a FIFO (first-in-first-out) scheduling policy. This is the policy used by modern devices such as GPUs.
A4. The runtime has negligible overhead. A device begins processing a task as soon as its input tensors are available and the device has finished previous tasks

One more thing

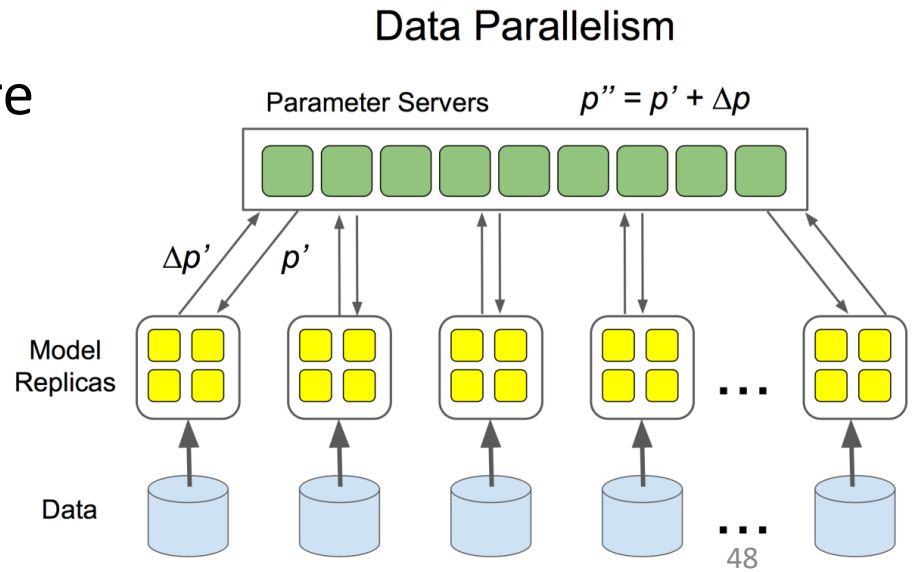
- "… the underlying implementation of Legion automatically and systematically **overlaps** communication with computation and optimizes the path and pipelining of data movement across the machine …"
- "… Similar to existing deep learning systems, the FlexFlow distributed runtime supports **overlapping** data transfers with computation to hide communication overheads. …"
- What is this overlap?

Priority-based Parameter Propagation for Distributed DNN Training

Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko (SysML 2019)

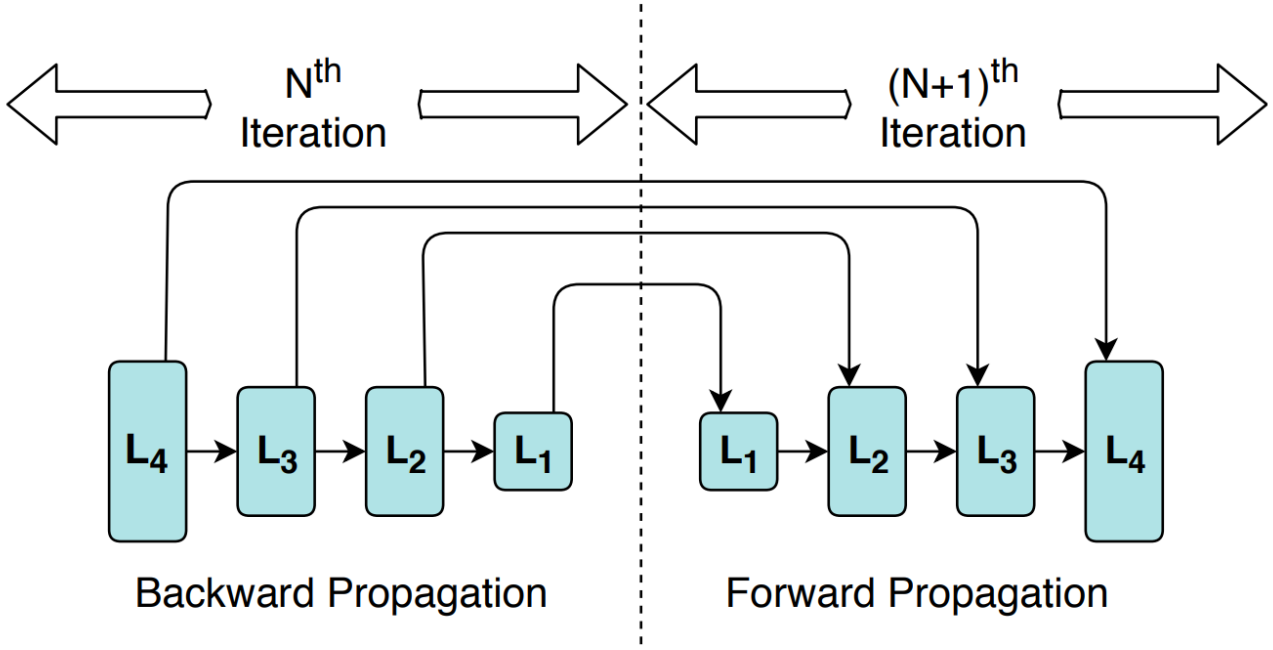
Motivation

- The paper overlap parameter synchronization with computation to improve the training performance:
  - the optimal data representation granularity for the communication may differ from that used by the underlying DNN model implementation;
  - different parameters can afford different synchronization delay;
  - The traffic generated by the training processes are generally bursty.
- The paper propose a new synchronization mechanism called **Priority-based Parameter Propagation (P3)**.



Gradient consumption

- Better schedule parameter synchronization based on:
  - when the gradients are generated;
  - when *the data* is consumed.
  - e.g., During training, the gradients of the layers are generated from final to initial layers and subsequently consumed in the reverse order in the next iteration.

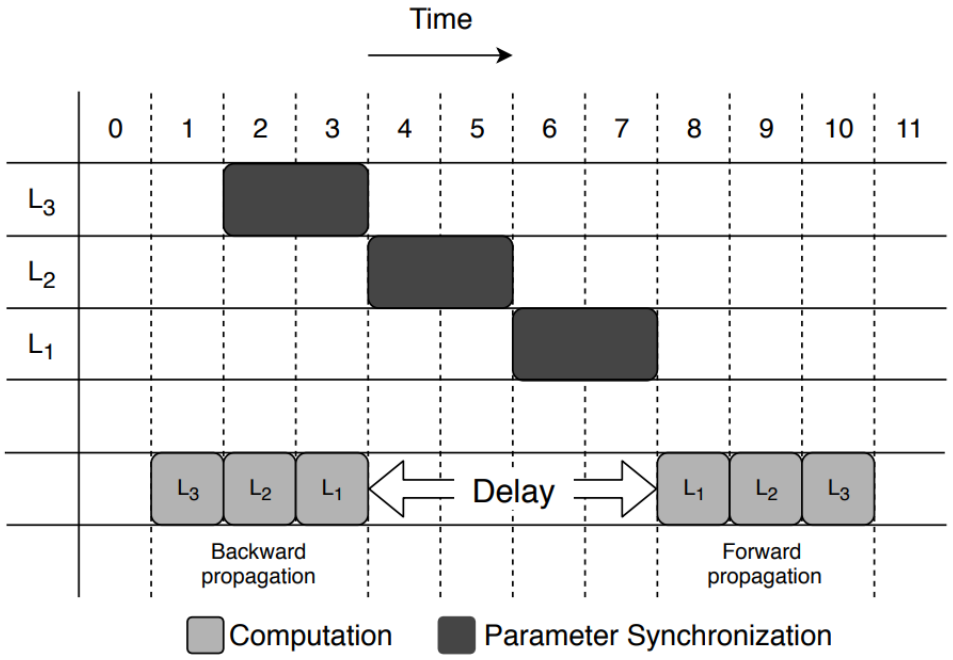


Synchronization granularity

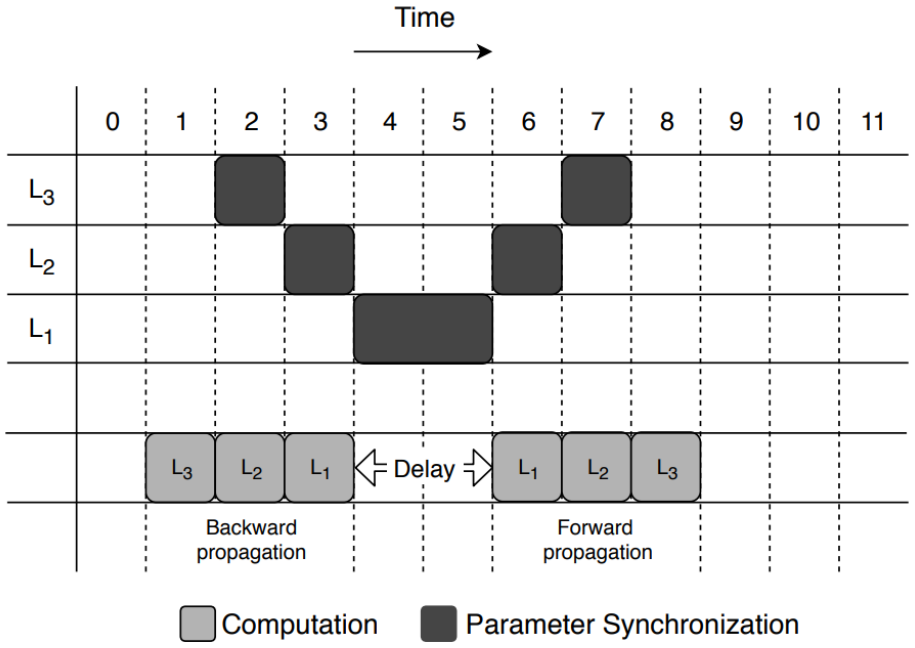
- The layer-wise granularity may not always be optimal for parameter synchronization.
- In their experiments, for certain heavy models (e.g., VGG (Simonyan & Zisserman, 2014), Sockeye (Hieber et al., 2017)), parameter synchronization at a finer granularity improves the network utilization and reduces the communication delay.

Limitations of parameter synchronization

- Aggressive synchronization



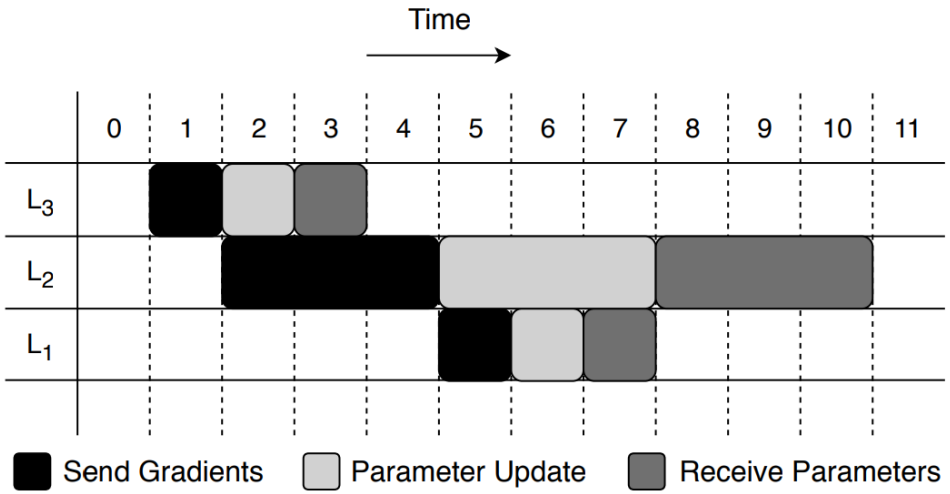
(a) Aggressive synchronization



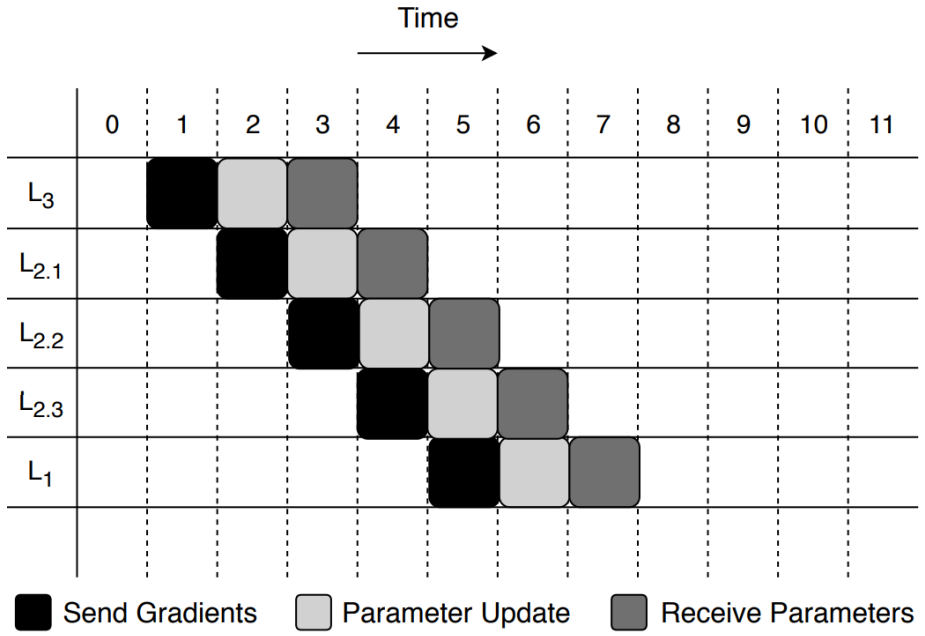
(b) Priority based synchronization

Limitations of parameter synchronization

- Synchronization at layer-level granularity.



(a) Layer level granularity



(b) Fine granularity

Approach - Priority-based Parameter Propagation (P3)

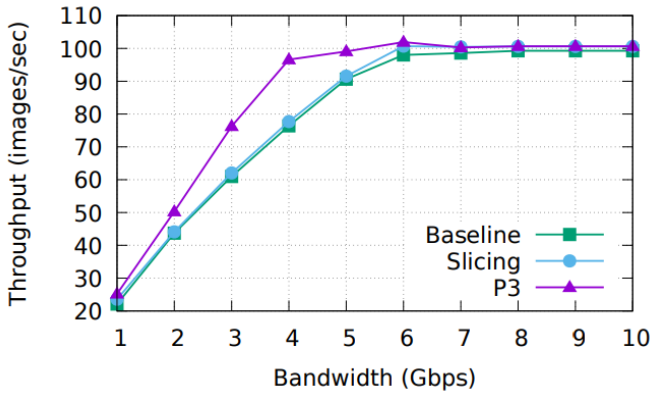
- P3 has two main components:
  - Parameter Slicing: split layers into smaller slices and synchronize independently;
  - Priority-based Update: synchronizes parameter slices based on their priority.
- Layers' priorities are based on the order in which they are processed in the forward propagation.
- During back propagation, P3 always allocates network cycles to the highest priority slices in the queue, preempting synchronization of the slices from a previous lower priority layer if necessary (through a producer-consumer mechanism).

Experiment

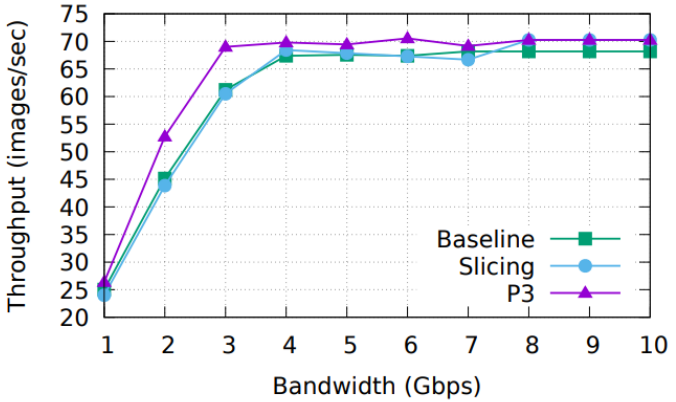
- Baseline system: KVStore (standard MXNet implementation) .
- P3: Implementation by modify KVStore.
- Evaluated on:
  - ResNet-50, Inception-v3, VGG-19, and Sockeye.
- Hardware differs based on the types of evaluation.

Evaluation - throughput

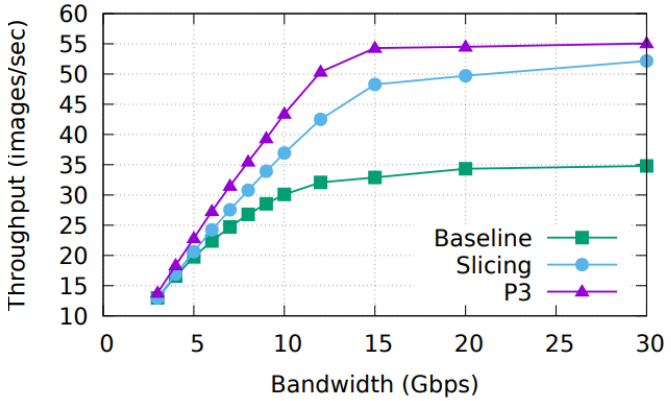
- measure the training throughput of ResNet-50, InceptionV3, VGG-19, and Sockeye on a tightly controlled four-machine cluster by setting different transmission rates on the network interface on all the machines



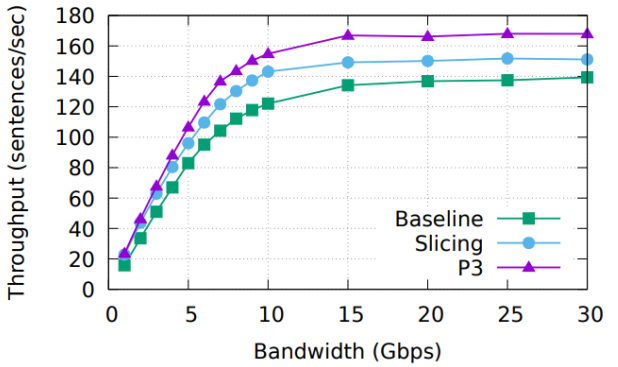
(a) ResNet-50



(b) InceptionV3

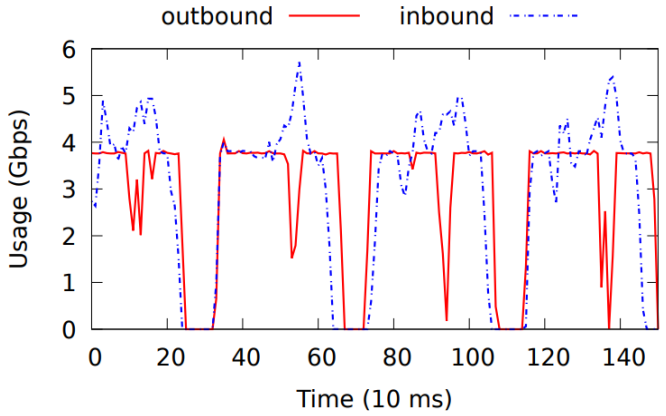


(c) VGG-19

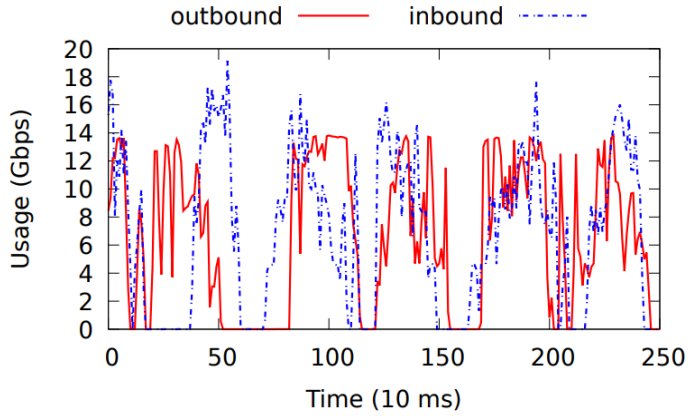


(d) Sockeye

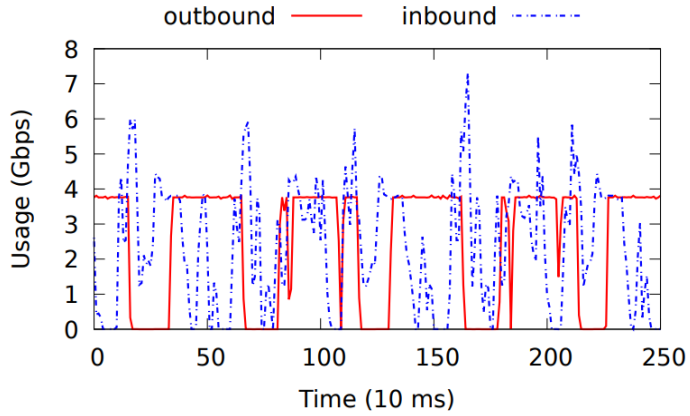
Evaluation – network utilization



(a) ResNet-50 at 4Gbps

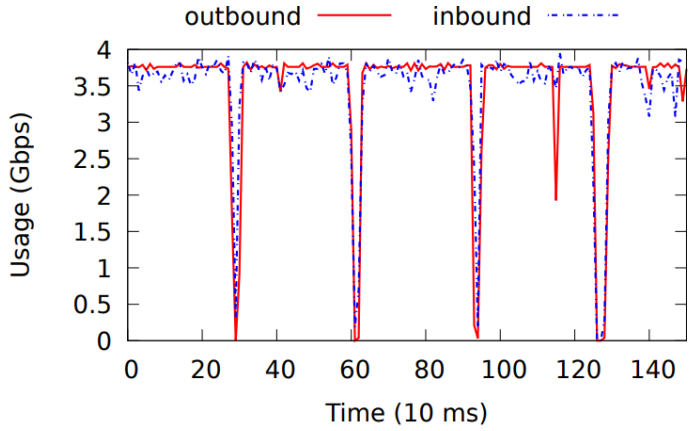


(b) VGG-19 at 15Gbps

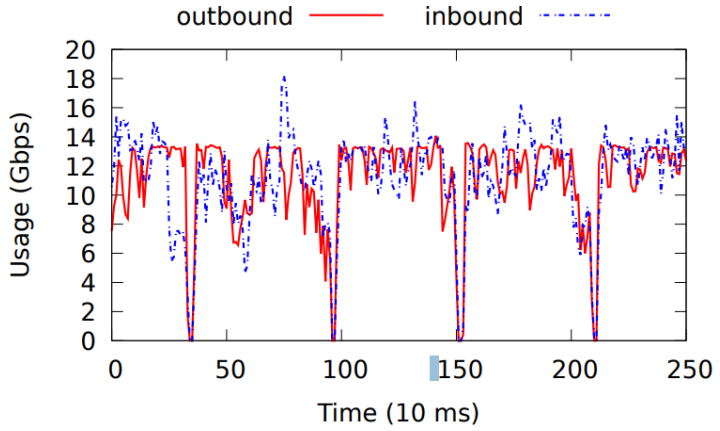


(c) Sockeye at 4Gbps

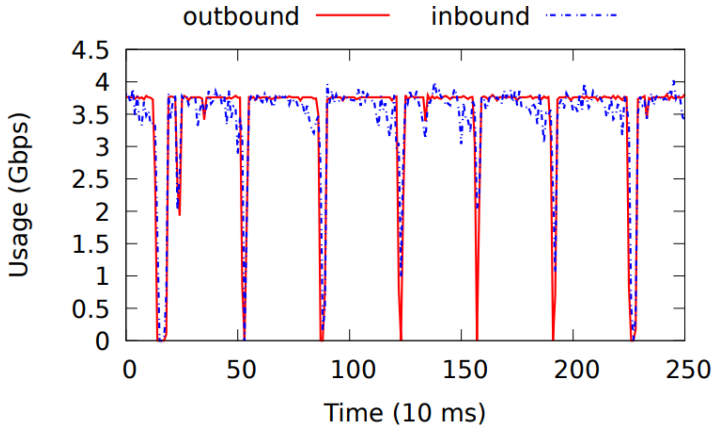
Figure 8. Network utilization of the baseline system



(a) ResNet-50 at 4Gbps



(b) VGG-19 at 15Gbps



(c) Sockeye at 4Gbps

Figure 9. Network utilization of *P3*

Evaluation – scalability

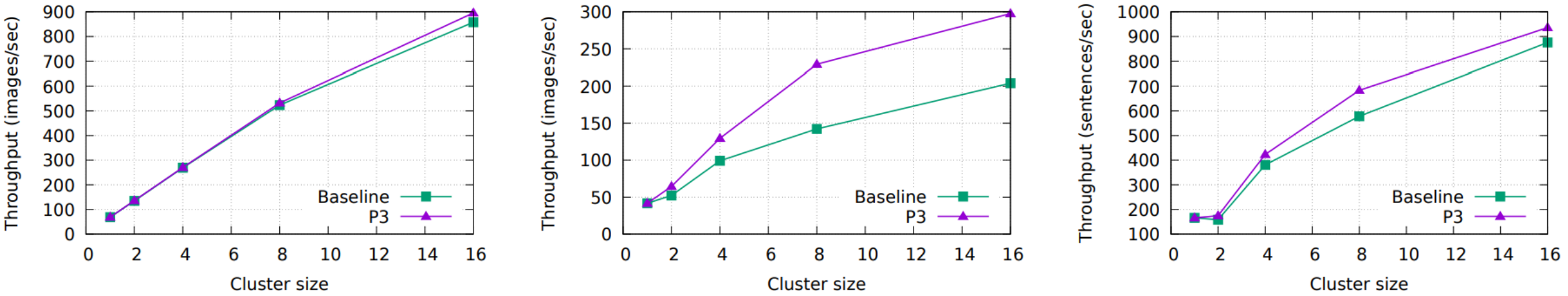
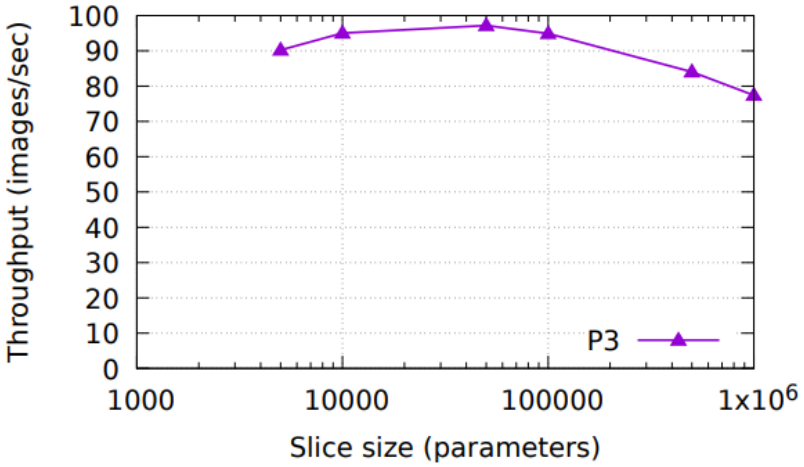
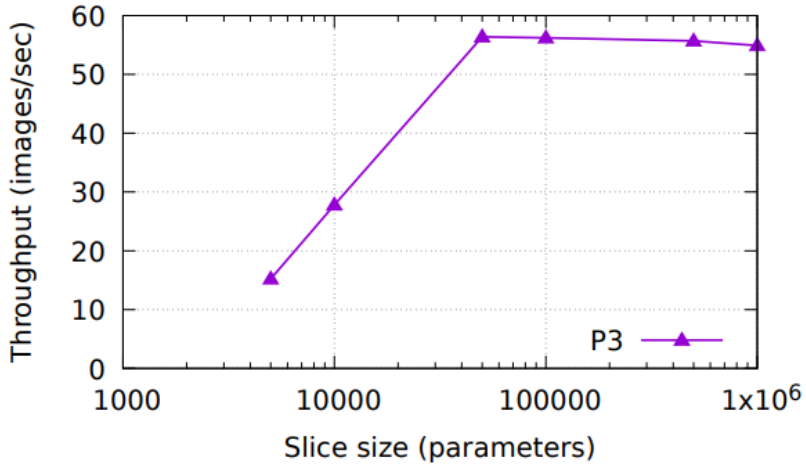


Figure 10. Throughput scaling with different number of machines

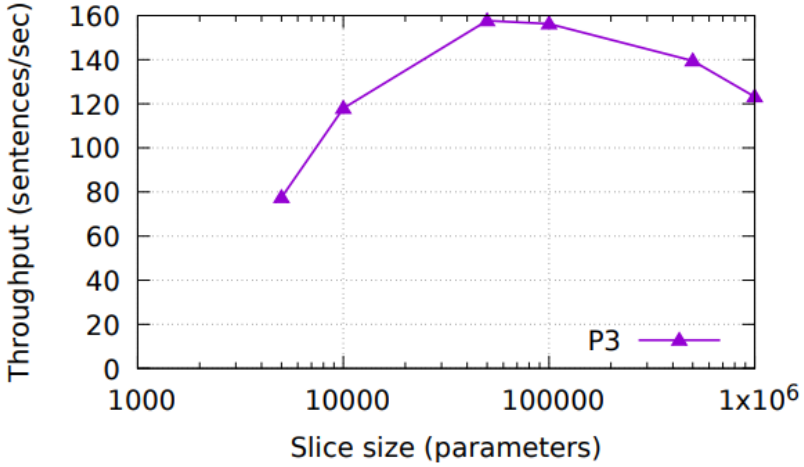
Evaluation – slice size selection



(a) ResNet-50



(b) VGG-19



(c) Sockeye

Figure 12. Granularity v.s. Throughput

A journey to parallelizing DNN training

- Data and model parallelism
- One weird trick
- Search through multiple dimensions:
  - OptCNN (the first paper)
  - Flexflow (the second paper)
- Better schedule the parameter sync:
  - P3 (the third paper)
- Other methods to help parameter sync:
  - Model compression
  - Gradient compression


THE END