Device Placement for Distributed Deep Learning

Advanced Topics in Distributed Systems

Sina Sheikholeslami

March 18, 2020

What is Device Placement?

The goal of automated device placement is to find the optimal assignment of operations to devices such that the end-to-end execution time for a single step is minimized and all device constraints like memory limitations are satisfied.

(From the GDP Paper)

Reviewed Papers

- Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning (NeurIPS ‘19)
- GDP: Generalized Device Placement for Dataflow Graphs (2019, Google Brain)
- Tofu: Supporting Very Large Models using Automatic Dataflow Graph Partitioning (EuroSys ‘19)

Placeto

Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning

Ravichandra Addanki, **Shaileshh Bojja Venkatakrishnan**, **Shreyan Gupta**,
Hongzi Mao, **Mohammad Alizadeh**

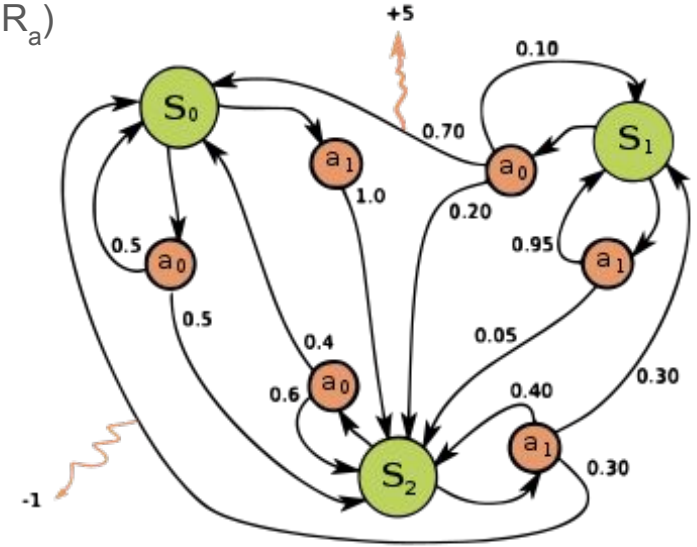
MIT Computer Science and Artificial Intelligence Laboratory

Key Ideas

- Partition operators (ops) into predetermined groups, place ops from the same group on the same device
- Learns an iterative policy and executes it on a given input graph
  - this policy has been learned using RL on a set of structurally similar computation graphs
- Markov Decision Process Formalism
- Graph Embeddings

Markov Decision Process (MDP)

- Decision making framework in situations where outcomes are partly random and partly under the control of a decision maker.
  - a 4-tuple (S, A, P_a, R_a)

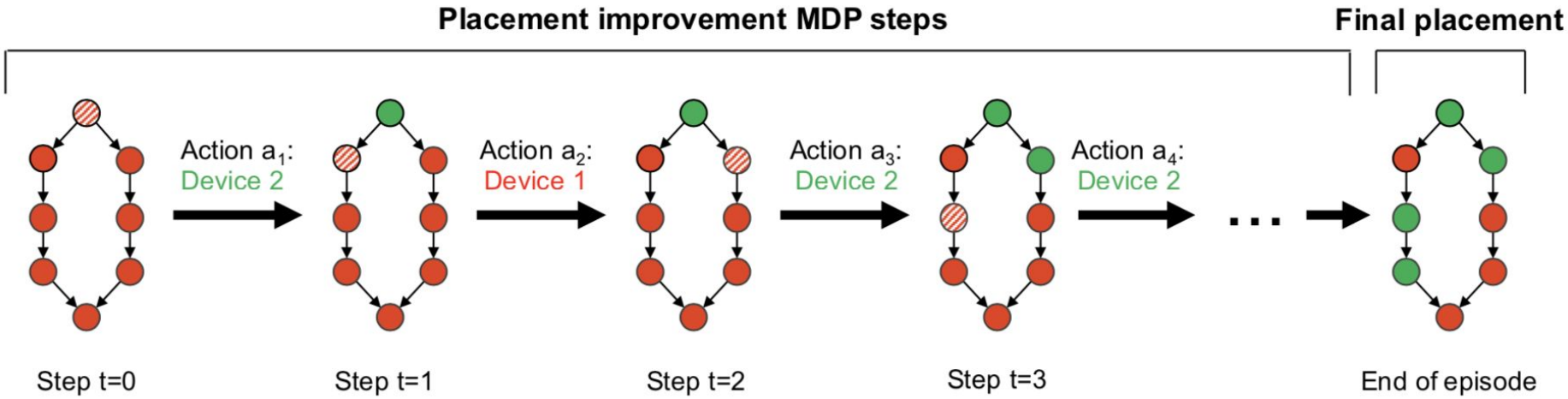


(Wikipedia)

MDP Formulation (1/2)

- $\mathscr{G}$ is a family of computation graphs for which we want to learn an efficient placement policy
- A **state** observation ***s*** comprises of a graph $G(V, E) \in \mathscr{G}$ with the following features for each node $v \in V$ which is an op group:
  - estimated runtime of *v*, total size of tensors output by *v*,
    current device placement of *v*, a flag indicating whether *v* has been visited before,
    a flag indicating whether *v* is the current node for which the placement has to be updated
- At the **initial state** $\boldsymbol{s_0}$ the nodes are assigned to devices arbitrarily, the visit flags are all 0, and an arbitrary node is selected as the current node
- An **action** is the placement of *v* on a device

MDP Formulation (2/2)



Reward Assignment

- A zero reward at each intermediate step in MDP
- A reward equal to the **negative run time of the final placement** at the terminal step

Approach (1)

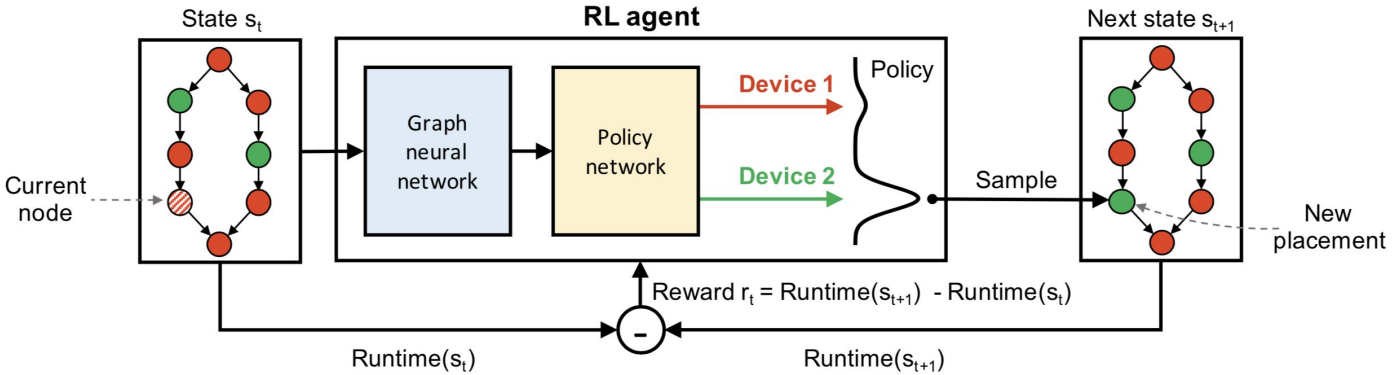
- An **intermediate** reward of
$$r_t = p(s_t) - p(s_{t-1})$$
at the t-th step for each $t = 1, \ldots, |V|$

Approach (2)

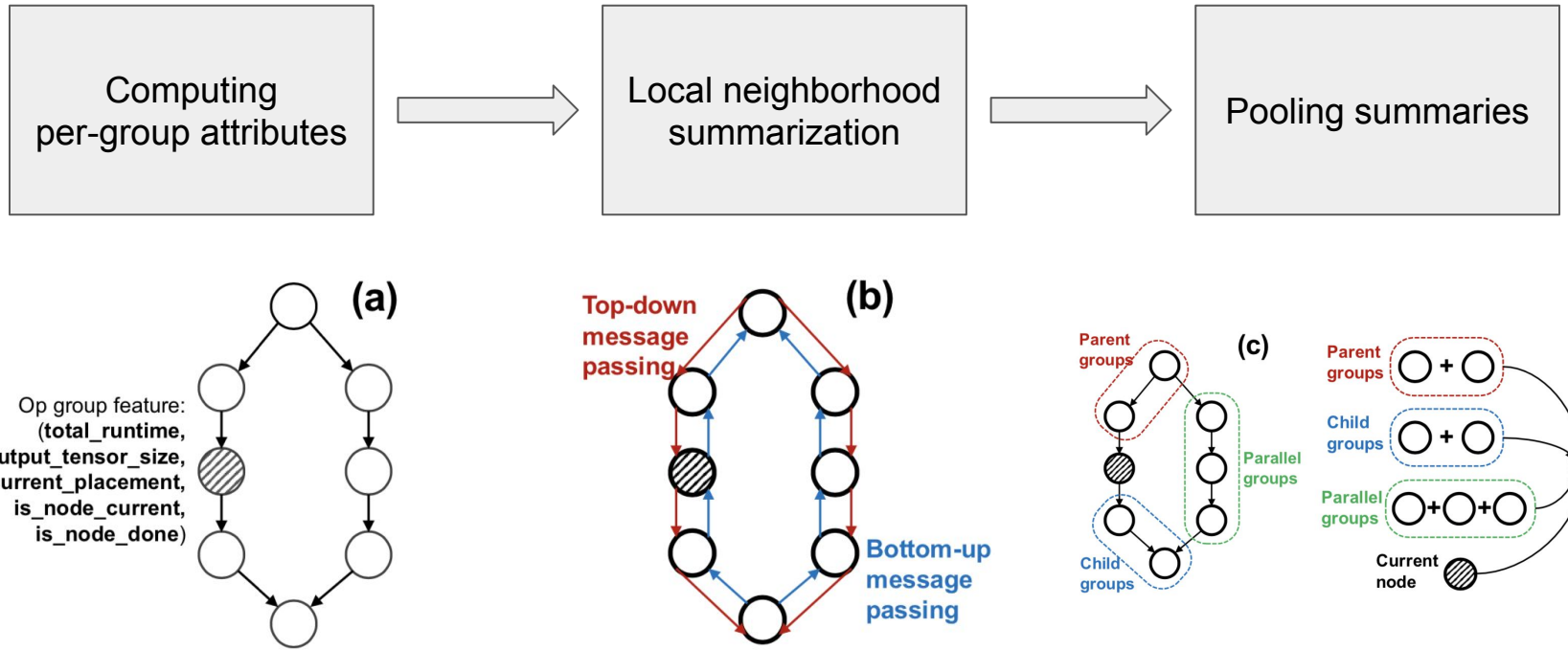
A **penalty** in the reward proportional to the peak memory utilization if it crosses a certain threshold M

Policy Network Architecture

- The MDP policy is parametrized using a neural network
  - input: graph configuration in state s_t
  - output: updated placement for the t-th node
- The graph-structured information of the state is vectorized via a graph embedding procedure
  - implemented using a graph neural network, learned jointly with the policy



Graph Embedding Architecture



Experimental Setup

- Main evaluation on Inception-V3, NMT, and NASNet
- Further evaluation on three synthetic datasets each comprising of 32 graphs
  - cifar10, ptb: each graph has 128 nodes, synthesized by ENAS
  - nmt: each graph has 160 nodes, constructed by varying RNN length and batch size hyperparameters of NMT

- Single GPU
- Scotch
- Human Expert
- RNN-based approach

Training

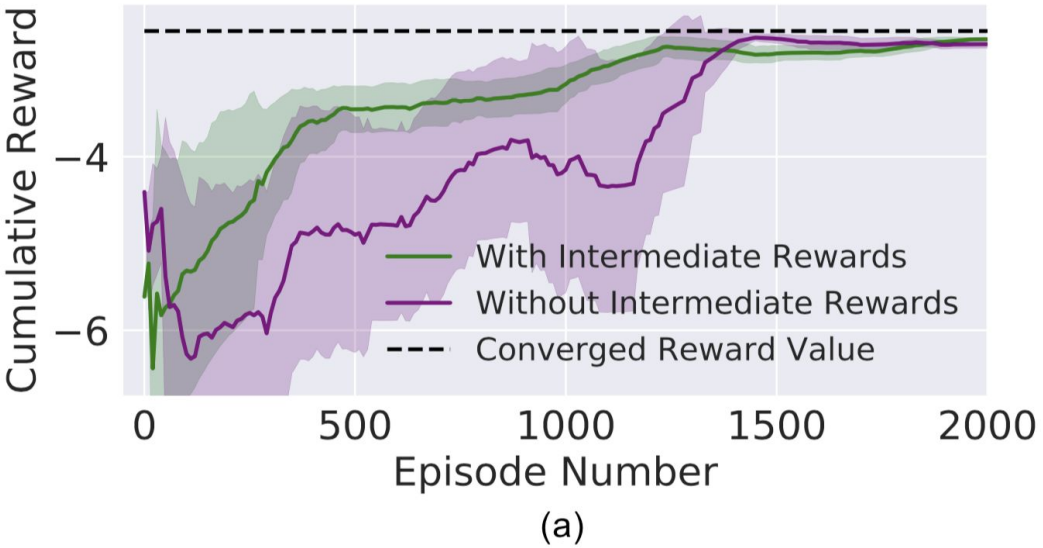
- Op-groups are formed by the strategy from ColocRL (Mirhoseini et al., 2017)
  - few leftover op-groups are iteratively merged with their neighbors
  - this is a shortcoming, later addressed by GDP
- A simulator is used to predict the runtime of any given placement for a given device configuration (only for training)

Results: Performance

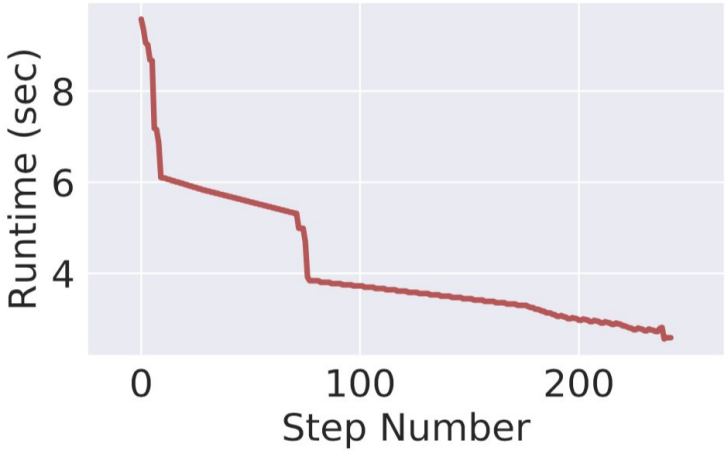
- Performance is quantified along two axes: 1) runtime of the best placement found, and 2) time taken to find the best placement.

	Placement runtime (sec)							Training time (# placements sampled)		Improvement	
Model	CPU only	Single GPU	#GPUs	Expert	Scotch	Placeto	RNN-based	Placeto	RNN-based	Runtime Reduction	Speedup factor
Inception-V3	12.54	1.56	2	1.28	1.54	1.18	1.17	1.6 K	7.8 K	- 0.85%	**4.8** ×
			4	1.15	1.74	1.13	1.19	5.8 K	35.8 K	5%	**6.1** ×
NMT	33.5	OOM	2	OOM	OOM	2.32	2.35	20.4 K	73 K	1.3 %	**3.5** ×
			4	OOM	OOM	2.63	3.15	94 K	51.7 K	**16.5 %**	0.55 ×
NASNet	37.5	1.28	2	0.86	1.28	0.86	0.89	3.5 K	16.3 K	3.4%	**4.7** ×
			4	0.84	1.22	0.74	0.76	29 K	37 K	2.6%	**1.3** ×

Results: Comparing Reward Approaches



(a)

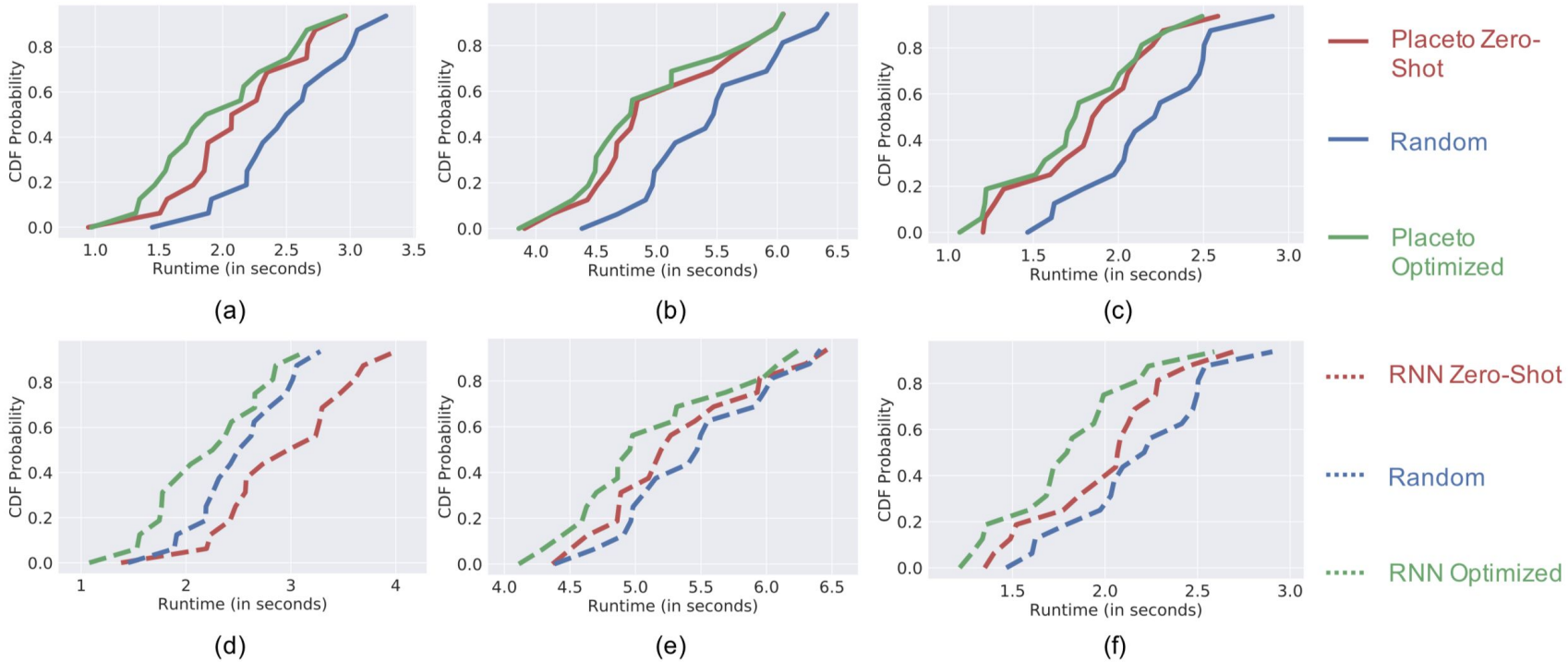


(b)

Results: Generalizability (1/2)

- For unseen graphs from a family of graphs for which a placement policy is learned, how good are the placements predicted by the same policy?
- Dataset: three family of graphs (cifar10, nmt, and ptb)
- Placeto Zero-Shot: uses the policy trained over graphs from the dataset, without further re-training
- Placeto Optimized: a policy is trained specifically over a test graph
- Random: uses a policy that generates placement for each node by random
- The same categorization is also defined for the RNN-based approach

Results: Generalizability (2/2)



GDP

GDP: GENERALIZED DEVICE PLACEMENT FOR DATAFLOW GRAPHS

Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi,* Daniel Wong,† Peter C. Ma, Qiumin Xu

Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, James Laudon

Google Brain

GDP: Key Ideas

- An end-to-end device placement network that can generalize to arbitrary graphs
- A scalable placement network with an efficient recurrent attention mechanism which eliminates the need for an explicit grouping stage before placement
- Batch pre-training and superposition based fine-tuning mechanism

Contributions

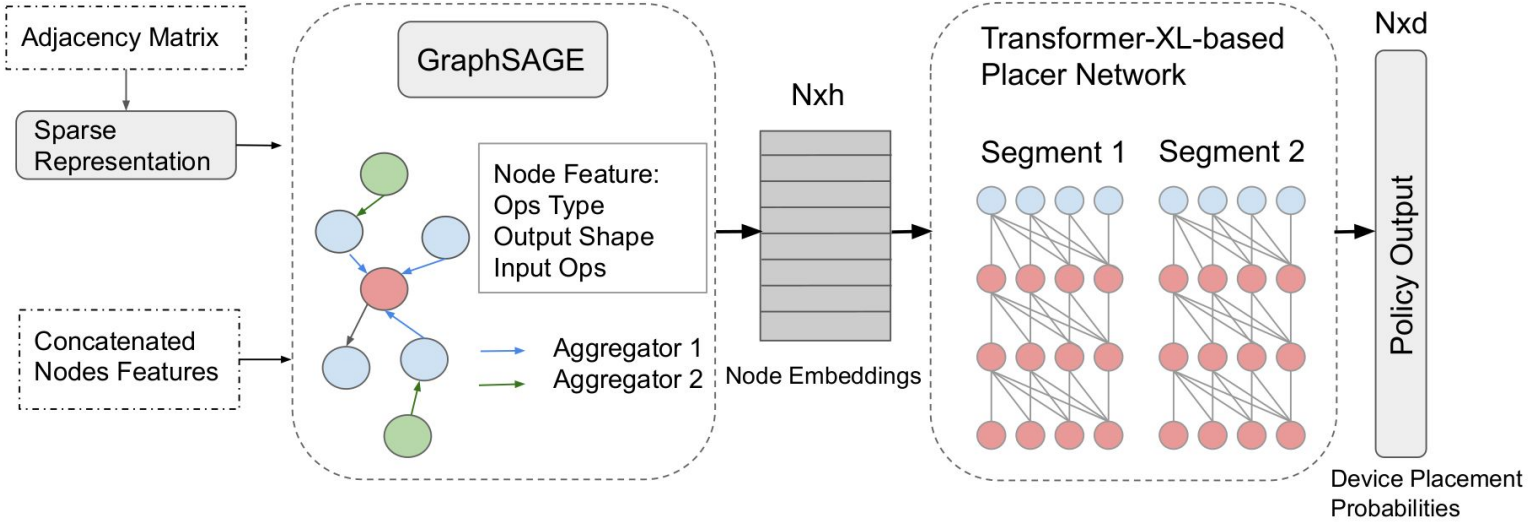
- The policy network consists of a graph-embedding network that encodes operation features and dependencies into a trainable graph representation, followed by a scalable sequence-to-sequence placement network based on an improved Transformer
- Removes complex constraints such as hierarchical grouping of ops, or colocation heuristics
- Can scale to graphs with over 50,000 nodes
- The policy is trained jointly over a set of dataflow graphs (instead of one at a time) and then fine-tuned on each graph individually

End-to-end Policy Formulation

- $G(V, E) \in \mathcal{G}$
- Goal: learn a policy $\boldsymbol{\pi} : \mathcal{G} \rightarrow D$ that maximizes the reward $r_{G, D}$ defined based on the runtime
- $\boldsymbol{\pi}_{\boldsymbol{\theta}}$ is a neural network parametrized by $\boldsymbol{\theta}$, and is our policy network
- The RL objective is optimized using proximal policy optimization (for improved sample efficiency)

$$J(\theta) = \mathbb{E}_{G \sim \mathcal{G}, D \sim \pi_\theta(G)}[r_{G,D}] \approx \frac{1}{N} \sum_{G} \mathbb{E}_{D \sim \pi_\theta(G)}[r_{G,D}]$$

Overview of the Process



Graph Embedding (1/2)

- Graph Neural Networks (GNNs) are used to capture the topological information encoded in the dataflow graph
- Adopts the feature aggregation scheme proposed in GraphSAGE to model the dependencies between the operations and build a generalizable placement policy
- Nodes and edges in the dataflow graph are represented as the concatenation of their meta features (e.g., operation type, output shape, adjacent node ids)
- The graph embedding process consists of multiple iterations

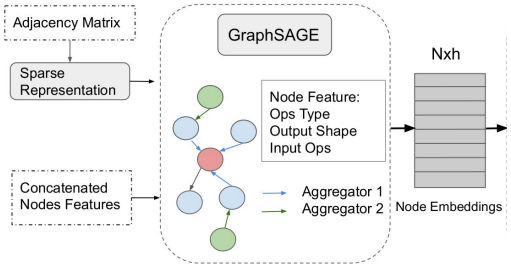
Graph Embedding (2/2)

1. Each node *v* aggregates the feature representation of its neighbors, $\{h_u^{(l)}, \forall u \in \mathcal{N}(v)\}$, into a single vector (with max pooling):

$$h_{\mathcal{N}(v)}^{(l)} = \max(\sigma(W^{(l)} h_u^{(l)} + b^{(l)}), \forall u \in \mathcal{N}(v))$$

2. Concatenate the node's current representation with the aggregated neighborhood vector and feed it to a fully connected layer:

$$h_v^{(l+1)} = f^{(l+1)}(\text{concat}(h_v^{(l)}, h_{\mathcal{N}(v)}^{(l)}))$$



Placement Network

- A Transformer-based attentive network to generate operator placements in an end-to-end fashion, to capture the long term dependencies in very large dataflow graphs
- The positional embedding in the original transformer is removed to prevent overfitting, as the graph embedding already contains spatial information for each node
- Segment-level recurrence: hidden states computed for the previous set of nodes are cached and reused as an extended context during the training of the next segment

Batch Training with Parameter Superposition

- Challenge: graphs from different tasks (computer vision, language, speech, etc.) have drastically different architectures and GDP aims to generalize over them
- Main idea: train a single shared policy, but condition its parameters based on the input features to mitigate the potentially undesirable interference among different input graphs
- A feature conditioning layer is added as an additional transformer layer to the placement network:

$$x^{(l+1)} = g^{(l)}(c(x^{(0)}) \odot x^{(l)})$$

Experiments

- Machines with one Intel Broadwell CPU and up to 8 Nvidia P100 GPUs
- Comparison with Human expert Placement (HP), TensorFlow METIS, and Hierarchical Device Placement (HDP)
- Evaluation on: RNN Language Modeling, GNMT, Transformer-XL, Inception, AmoebaNet, and WaveNet
- Run time: training step time of the best placement found, in seconds
- Reward: negative square root of the run time
- Penalty for invalid placements: -10

Results: GDP-one (Separate Graph Training)

Model (#devices)	**GDP-one** (s)	HP (s)	METIS (s)	HDP (s)	Run time speed up over HP / HDP	Search speed up
2-layer RNNLM (2)	0.234	0.257	0.355	0.243	9.8% / 4%	2.95x
4-layer RNNLM (4)	0.409	0.48	OOM	0.490	17.4% / 19.8%	1.76x
2-layer GNMT (2)	0.301	0.384	OOM	0.376	27.6% / 24.9%	30x
4-layer GNMT (4)	0.409	0.469	OOM	0.520	14.7% / 27.1%	58.8x
8-layer GNMT (8)	0.649	0.610	OOM	0.693	-6% / 6.8%	7.35x
2-layer Transformer-XL (2)	0.386	0.473	OOM	0.435	22.5% / 12.7%	40x
4-layer Transformer-XL (4)	0.580	0.641	OOM	0.621	11.4% / 7.1%	26.7x
8-layer Transformer-XL (8)	0.748	0.813	OOM	0.789	8.9% / 5.5%	16.7x
Inception (2)	0.405	0.418	0.423	0.417	3.2% / 3%	13.5x
AmoebaNet (4)	0.394	0.44	0.426	0.418	26.1% / 6.1%	58.8x
2-stack 18-layer WaveNet (2)	0.317	0.376	OOM	0.354	18.6% / 11.7%	6.67x
4-stack 36-layer WaveNet (4)	0.659	0.988	OOM	0.721	50% / 9.4%	20x
GEOMEAN	-	-	-	-	**16% / 9.2%**	**15x**

Results: GDP-batch (Mixing Redundant Tasks)

- Mixing redundant tasks in the batches results in both faster and better learning for RNNLM and GNMT with large number of layers

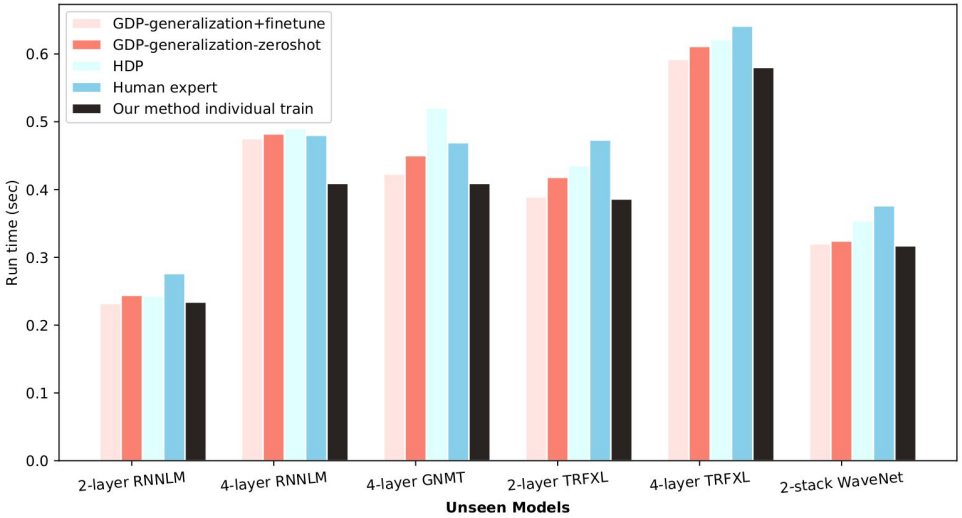
Batch Setting	Model	speed up (s)
Batch 2	Inception	0
	AmoebaNet	-4.5%
	2-layer RNNLM	0
	2-layer GNMT	0
	2-layer Transformer-XL	6.5%
	2-stack 18-layer Wavenet	4%
Batch 3	2-layer RNNLM	0
	4-layer RNNLM	0
	8-layer RNNLM	4.5%
	2-layer GNMT	0
	4-layer GNMT	0
	8-layer GNMT	8%
Batch 4	3x8-layer GNMT	5.1%
Batch 5	3x8-layer RNNLM	4.5%

Results: GDP-batch (Batch Training)

Model	Speed up	Model	Speed up
2-layer RNNLM	0	Inception	0
4-layer RNNLM	5%	AmoebaNet	-5%
2-layer GNMT	0	4-stack 36-layer WaveNet	3.3 %
4-layer GNMT	0	2-stack 18-layer WaveNet	15%
2-layer Transformer-XL	7.6%	8-layer Transformer-XL	1.5%
4-layer Transformer-XL	3%		

Run time comparison on GDP-batch vs. GDP-one

Results: GDP-generalization(+finetune/-zeroshot)



Tofu

Supporting Very Large Models using Automatic Dataflow Graph Partitioning

Minjie Wang
New York University

Chien-chin Huang
New York University

Jinyang Li
New York University

Motivation

- Limited GPU device memory puts a constraint on the size of the neural networks that can be explored
  - compared to CPU memory (RAM), GPU memory has much higher bandwidth but also smaller capacity
    - e.g., 12 GB (K80) and 16 GB (Tesla V100)’
- Partitioning should be completely **transparent to the user**: the same program written for a single device can also be run across devices without change.

Research Questions

Can we support **generalizable** fine-grained tensor partitioning on one of the general purpose DL frameworks (e.g., MXNet)?

Research Questions - Breakdown

- How to partition the input/output tensors and parallelize the execution of an **individual operator**?
  - What are the viable partitioning dimensions?
  - What makes it harder: a dataflow framework supports a large number of operators
    - MXNet: 139 (from the paper)
    - TensorFlow: 341 (from the paper)
- How to optimize the partitioning of different operators for **the overall graph**?
  - the graph of operators is more complex and an order of magnitude larger than the graph of layers
    - e.g., ResNet-152 implementation in MXNet has > 1500 operators
- How to make this partitioning **transparent** to the user?

Contributions

- TDL (Tensor Description Language) for a high-level specification of operators: the way an output tensor is derived from its input
  - an operator's TDL description is separate from its implementation
  - TDL specifications can be statically analyzed
- Tofu, a system that partitions very large DNN models across multiple GPU devices to reduce per-GPU memory footprint
  - Prototype built for MXNet, but they claim it can potentially be applied to other dataflow systems such as TensorFlow

Solution

- Tofu automatically partitions a dataflow graph of fine-grained tensor operators (used by e.g., TensorFlow and MXNet)
  - to enable automatic discovery of an operator's partition dimension, the semantics of an operator are described in a simple language inspired by Halide, called TDL
- Several techniques to shrink the search space of the overall dataflow graph partitioning
  - a **recursive** search algorithm that partitions the graph **among only two workers** at each recursive step
  - **graph coarsening** by grouping related operators

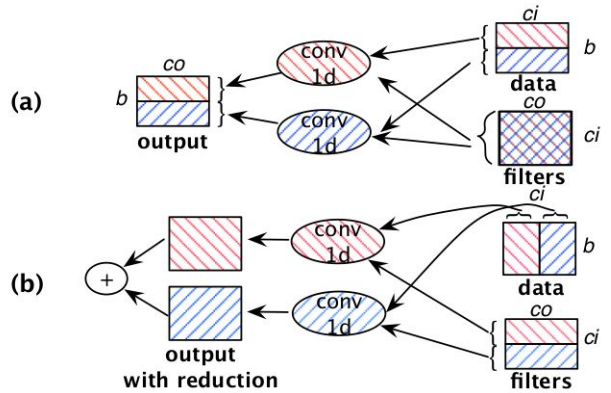
Partitioning a Single Operator: Partition-n-reduce

- Suppose operator *c* computes output tensor *O*. Under partition-n-reduce, *c* can be parallelized across two workers by executing the same operator on each worker using **smaller inputs**. The final output tensor *O* can be obtained from the output tensors of both workers, (*O1* and *O2*):
  - *O* is the **concatenation** of *O1* and *O2* along some dimension
  - *O* is the **element-wise reduction** of *O1* and *O2*

Partitioning a Single Operator: Partition-n-reduce

- A naive implementation of **conv1d** in Python, and two of the possible ways for parallelizing it using partition-n-reduce
  - on the 5th line, *di* should be *ci*

```
def conv1d(data, filters):
 for b in range(output.shape[0]): #b is batch dimension
   for co in range(output.shape[1]): #co is output channel
    for x in range(output.shape[2]): #x is output pixel
     for ci in range(filters.shape[0]): #di is input channel
      for dx in range(filters.shape[2]): #dx is filter window
       output[b, co, x] += data[b, ci, x+dx]
                         * filters[ci, co, dx]
```



TDL: Tensor Description Language

- Tofu analyzes the **access pattern** of an operator to determine all viable partition strategies, using specifications in TDL
  - the description specifies (at a high-level) how the output tensor is derived from its input
- index variables (arguments of the lambda function)
- tensor elements (e.g., filters[ci, co, dx])
- arithmetic operations involving constants, index variables, tensor elements, or TDL expressions
- reduction over a tensor along one or more dimensions
  - reducers are commutative and associative functions that aggregate elements of a tensor
  - supported built-in reducers: Sum, Max, Min, and Prod.

```
@tofu.op
def conv1d(data, filters):
  return lambda b, co, x:
    Sum(lambda ci, dx: data[b, ci, x+dx]*filters[ci, co, dx])
```

Analyzing TDL Descriptions

- Tofu analyzes the TDL description of an operator to discover its **basic partition strategies**
  - a partition strategy specifies the input tensor regions required by each worker to perform its share of the computation
  - a basic partition strategy parallelizes an operator for 2 workers only
- The search algorithm uses basic strategies recursively to optimize partitioning for more than two workers

Example: Input Regions from Concrete Ranges

```
# TDL for shift_two operator

def shift_two(A): B = lambda i : A[i+2]; return B
```

- Suppose we want to partition along **output dimension** *i*. If we have *i*'s concrete range, e.g., [0, 9], we can compute that if one of the workers wants to compute *B* over [0, 4], it will need *A* over [2, 6].
  - *A*[2] -> *B*[0], *A*[3] -> *B*[1], ...
- Hugely inefficient in practice!
  - dataflow graphs can contain thousands of operators that are identical except for their index ranges (tensor shapes)

Symbolic Interval Analysis (for each operator)

- Suppose the output tensor of an operator has *n* dimensions:
  - `lambda x1, ..., xn : ...`
- We consider the range of index variable `xi` to be $[0, X_i]$
- We now symbolically execute the lambda function to calculate symbolic intervals
- We represent symbolic interval I as an affine transformation of all symbolic upper bounds:

$$I \triangleq [\sum_i l_i X_i + c, \sum_i u_i X_i + c] \qquad l_i, u_i, c \in \mathbb{R}$$

- In other words, I can be represented as $\langle l_1, \ldots, l_n, u_1, \ldots, u_n, c \rangle$
- By default initialize `xi` to $ZV[u_i = 1]$

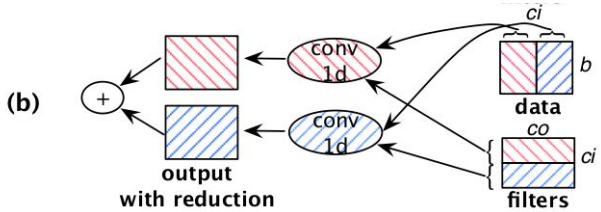
Discover Operator Partition Strategies

- Infer the input regions required by each of the 2 workers for every partitionable dimensions
- Case 1: no reduction, where each partition strategy corresponds to some output dimension
- Example: partitioning **conv1d** output tensor along dimension *b*
  - We use two different intervals for lambda variable *b* (in two separate analysis runs)
    - $ZV[u_b = \frac{1}{2}]$ and $ZV[l_b = \tilde{\frac{1}{2}}, u_b = 1]$
  - each run calculates the input regions needed to compute half of the output tensor

Discover Operator Partition Strategies (Cont'd)

- Case 2: with reduction, so we partition along a reduction dimension
- Example: *ci* and *dx* in the following:

```
@tofu.op
def conv1d(data, filters):
  return lambda b, co, x:
    Sum(lambda ci, dx: data[b, ci, x+dx]*filters[ci, co, dx])
```



- Out of 47 non-element-wise MXNet operators describable by TDL, 11 have at least one reduction dimension

Partitioning the Dataflow Graph

- Combinatorially many choices to partition each tensor
  - since each operator has several partition strategies
- Tofu uses an existing Dynamic Programming algorithm (Jia et al., 2018) alongside techniques to make it practical
- Dataflow graph coarsening to shrink the searchspace and make it linear
  - e.g, grouping the forward and backward operations, coalescing element-wise operations
- Recursive search
  - in each recursive step, partition each tensor in the coarsened graph among two worker groups

---

Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks

---

Zhihao Jia [1] Sina Lin [2] Charles R. Qi [1] Alex Aiken [1]

What are We Looking For?

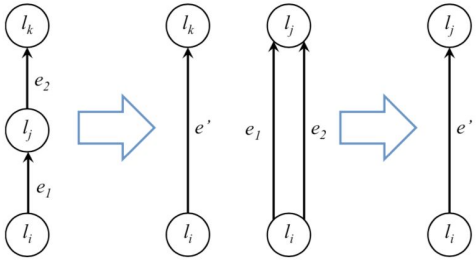
- Ideally, we want our partitioning to be optimized in terms of both **end-to-end execution time** and the **per-worker memory consumption**
- Tofu chooses to minimize the **total communication cost** based on two observations
  - GPU kernels for very large DNN models process large tensors and thus have similar execution time no matter which dimension its input/output tensors are partitioned on, so reducing communication between them would lead to reduction in end-to-end exec. time
  - the memory consumed at each GPU worker is in two areas: 1) storing a worker's share of tensor data, and 2) buffering data for communication between GPUs, which is proportional to the amount of communication

The Base DP Algorithms

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 $\mathcal{S}^{(0)}$

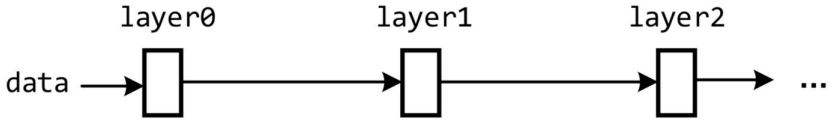


(a) Node elimination. (b) Edge elimination.

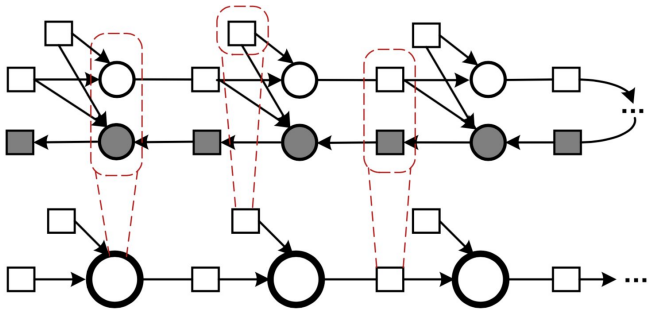
1. The algorithm first iteratively uses node and edge eliminations to simplify an input computation graph until neither eliminations can be applied.
2. After the elimination phase, the algorithm enumerates all potential strategies for the final graph to choose a strategy that minimizes the cost function, and then decides the configuration for the eliminated nodes by going back in the reversed order, eventually leading to an optimal parallelization strategy for the original graph.

Graph Coarsening: Motivation

- The Base DP Algorithm is only applicable for linear graphs, such as DNN layer graphs:



- However, dataflow graphs of fine-grained operators are usually non-linear, so we need coarsening:



Graph Coarsening: Approach

- Grouping forward and backward operations
  - each forward operator and its auto-generated backward operators form a group
  - each forward tensor (e.g., weights) and its gradient tensor form a group
- Coalescing operators: some operators should share the same strategy
  - element-wise operators (in gradient-based optimizers)
  - unrolled timesteps (in RNNs)
- Perform the DP algorithm on the coarsened graph
- However, coarsening alone is not enough: at each step, the DP algorithm has to consider all possible configurations of an operator group, leading to an explosion of the search space, hence the search time

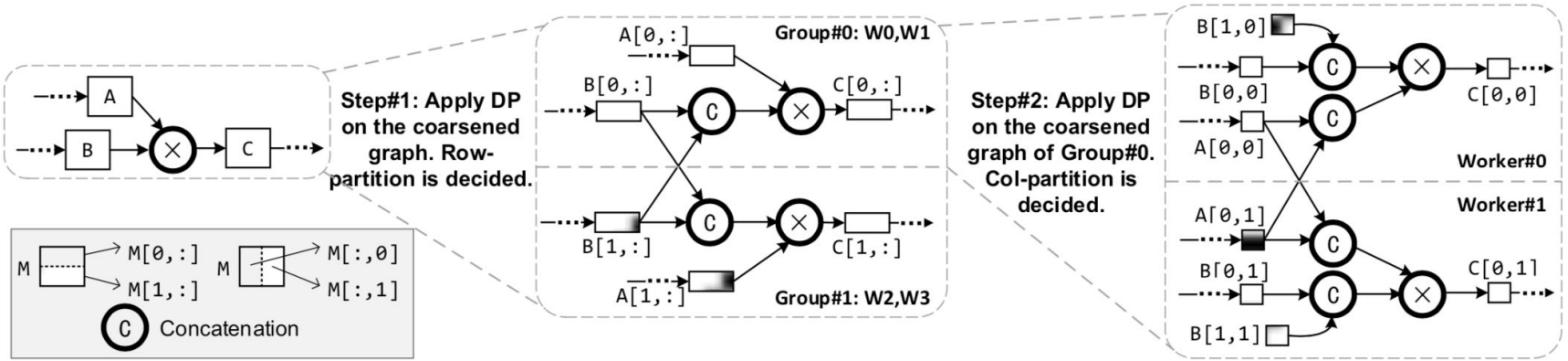
Recursive Partitioning

- Given $k = 2^m$ GPUs and a graph G:
- Run the DP algorithm with coarsening to partition G for two worker groups, each consisting of 2^{m-1} workers.
  - the partitioned dataflow graph has two halves, G_0 and G_1
  - if data is needed from other groups, add them as extra input tensors
- Repeat the first step on G_0 and apply the partition result to G_1 until there is only one worker per group.

	Search Time	
	WResNet-152	RNN-10
Original DP [14]	n/a	n/a
DP with coarsening	8 hours	>24 hours
Using recursion	8.3 seconds	66.6 seconds

(Best partitioning for 8 workers)

Recursive Partitioning Example

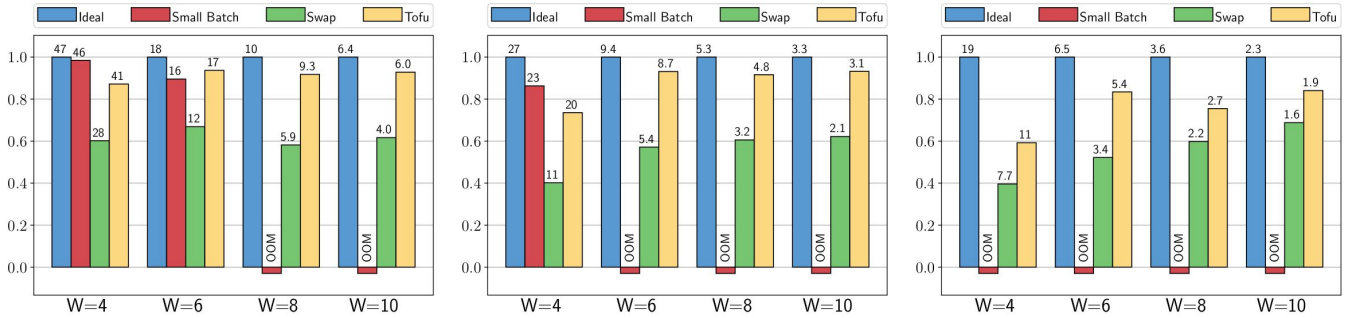


Experiments

- Tofu prototype implementation in MXNet
- Testbed: A single EC2 p2.8xlarge instance
  - 8 K80 GPUs, 12 GB memory each
  - GPUs are connected via PCIe bus with 21 GB/s peer-to-peer bandwidth
  - CPU-GPU bandwidth: 10 GB/s
  - 32 virtual CPU cores
  - 488 GB RAM
- Models: Wide ResNet and Multi-layer RNNs (most of which do not fit in a single GPU's memory)
  - But the DP algorithm paper (Jia et al., 2018) evaluates on Inception-v3, AlexNet, VGG-16
- Search time only compared with the original DP Algorithm
- Can find non-trivial partitioning strategies

Results - Training Throughput

- 25% - 400% speedup (training throughput) over alternative approaches to train very large models



(a) Wide ResNet-50 **(b)** Wide ResNet-101 **(c)** Wide ResNet-152

Limitations (1/2)

- Some operators cannot be expressed in TDL (e.g., Cholesky decomposition)
  - TDL does not support loops or recursion
- The automatically discovered partitioning strategies do not exploit the underlying communication topology
- Tofu is designed for very large DNN models (models that do not fit in the memory of a single GPU) and for moderately sized models, the resulting strategies are no better than data parallelism.

Limitations (2/2)

- The evaluation has been performed on a single machine with multiple GPUs
  - Their justification: for training very large DNNs on fast GPUs, the aggregate bandwidth required far exceeds the network bandwidth in deployed GPU clusters (e.g., Amazon's EC2 GPU instances have only 25 Gbps aggregate bandwidth).
- Tofu always partitions all operators across all workers
  - For moderately sized DNNs, this may lead to small GPU kernels and underutilization
- Tofu does not support non-uniform partitioning
  - consider a heterogeneous mix of GPUs in a cluster

Discussion Time