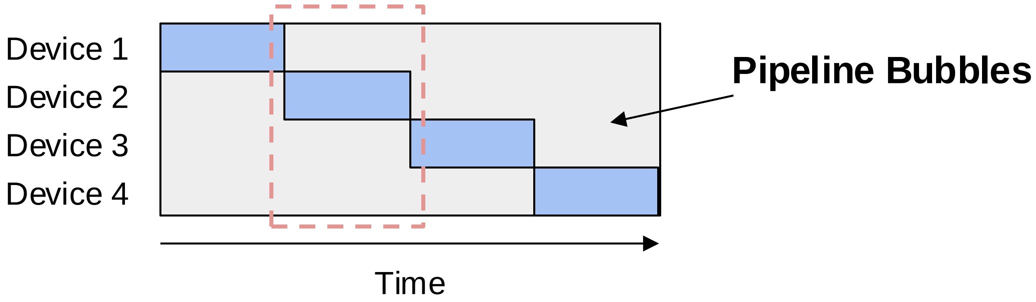
Where We Are

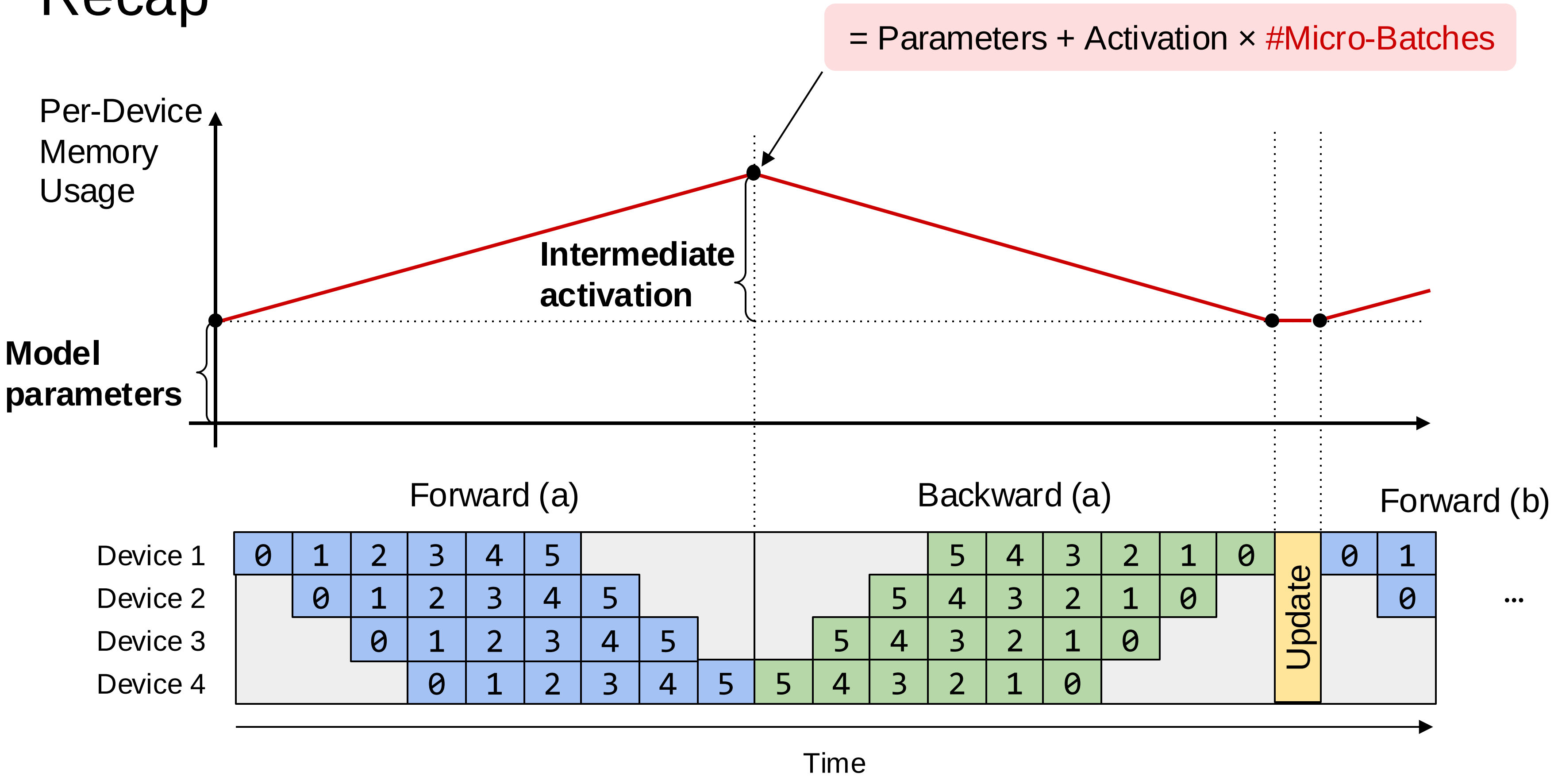
- Motivation
- History
- Parallelism Overview
- Data parallelism
- Model parallelism
  - **Inter-op parallelism**
  - Intra-op parallelism
- Auto-parallelization

Recap

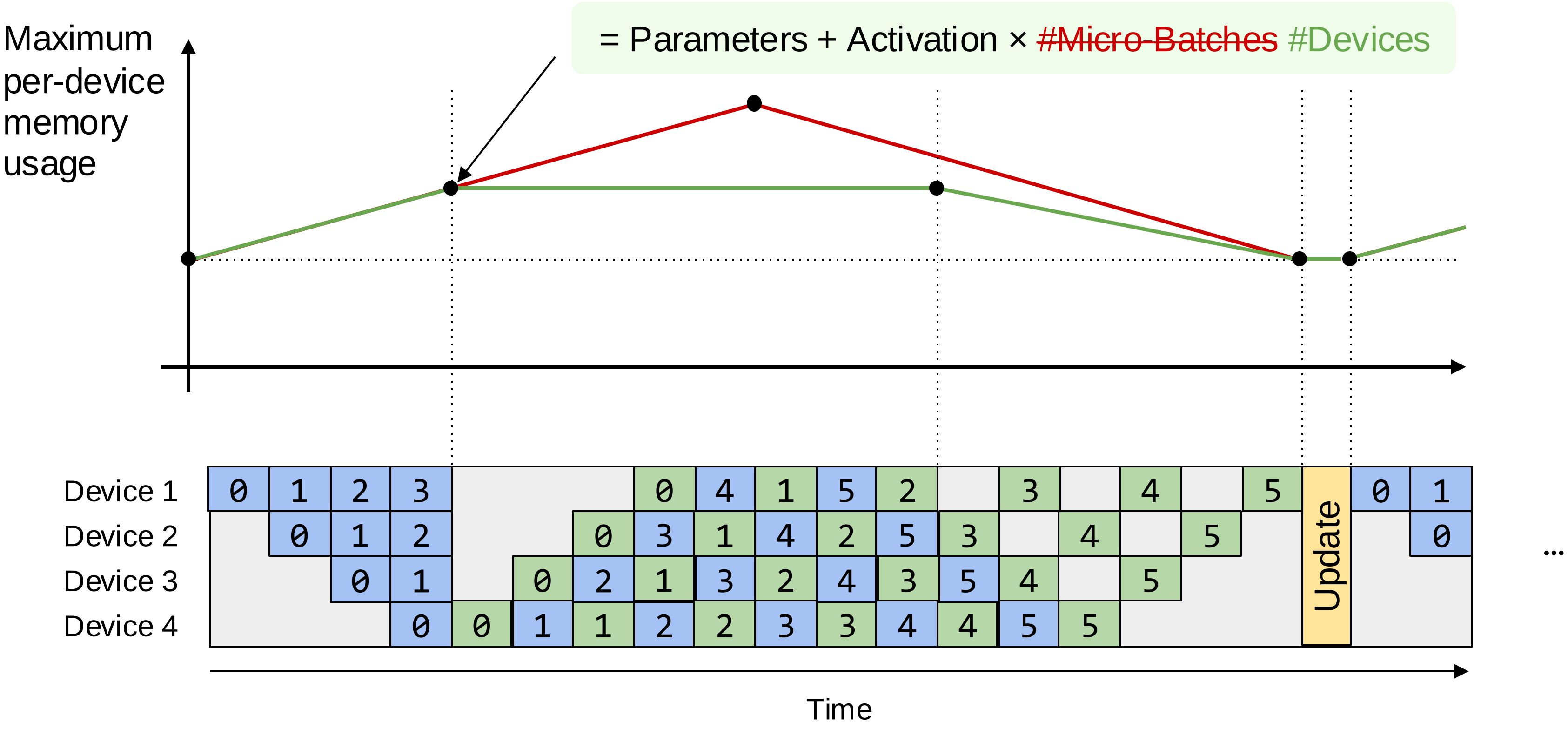


- Gray area ( indicates devices being idle (a.k.a. Pipeline bubbles).
- Only 1 device activated at a time.
- **Pipeline bubble percentage** `= (D - 1) / D`, assuming `D` devices.

Recap


= Parameters + Activation × #Micro-Batches
Per-Device Memory Usage
Intermediate activation
Model parameters
Forward (a)
Backward (a)
Forward (b)
Device 1
Device 2
Device 3
Device 4
Update
...
Time

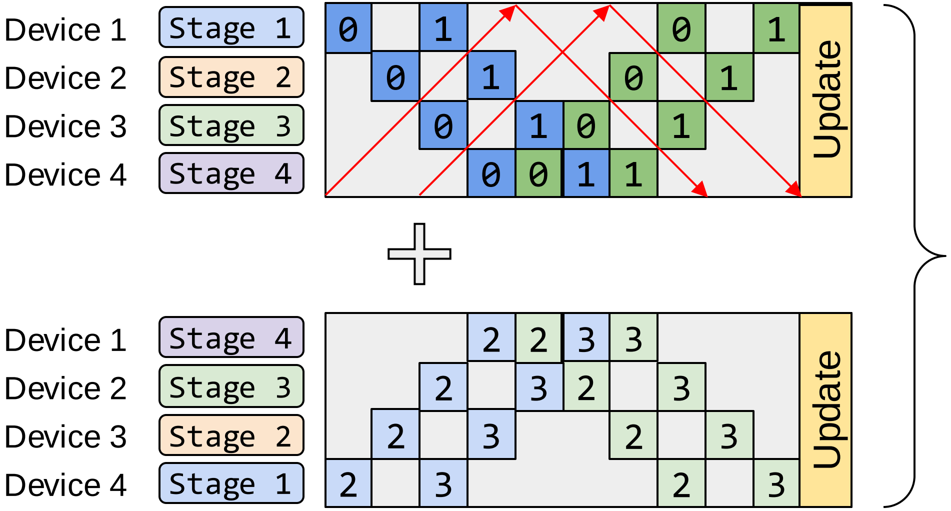
Recap



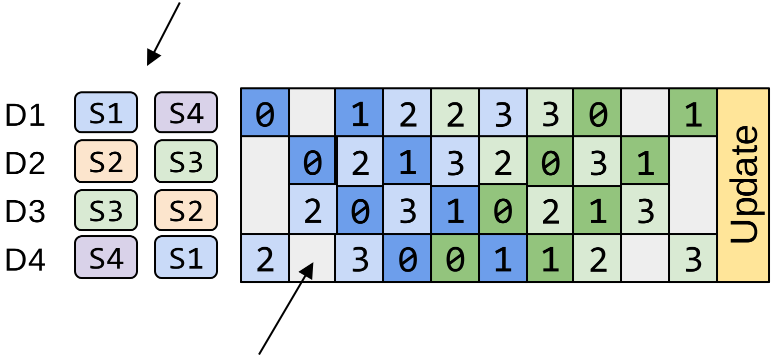
Fan, Shiqing, et al. "DAPPLE: A pipelined data parallel approach for training large models." *PPoPP 2021.*

Recap: Chimera

Idea: Store bi-directional stages and combine bidirectional pipeline to further reduce pipeline bubbles.



Extra copy of parameters & extra synchronization.



Pipeline bubbles percentage

= (D - 2) / (D - 2 + 2N)

with D devices and N micro-batches.

Li, Shigang, and Torsten Hoefler. "Chimera: efficiently training large-scale neural networks with bidirectional pipelines." *SC 21.*

Synchronous Pipeline Schedule Summary

✅ **Pros:**

- Keep the convergence semantics. The training process is exactly the same as training the neural network on a single device.

❌ **Cons:**

- Pipeline bubbles.
- Reducing pipeline bubbles typically requires splitting inputs into smaller components, but too small input to the neural network will reduce the hardware efficiency.

Asynchronous Pipeline Schedules

Idea: Start next round of forward pass before backward pass finishes.

✅ **Pros:**

- No Pipeline bubbles.

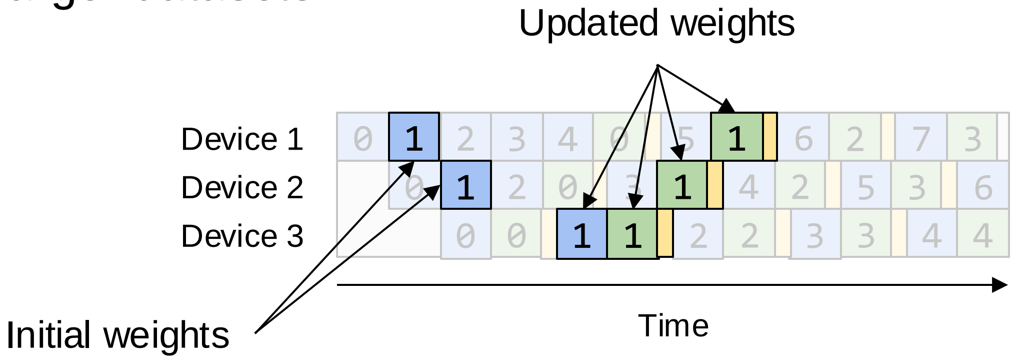
❌ **Cons:**

- Break the synchronous training semantics. Now the training will involve stalled gradient.
- Algorithms may store multiple versions of model weights for consistency.

AMPNet

Idea: Fully asynchronous. Each device performs forward pass whenever free and updates the weights after every backward pass.

Convergence: Achieve similar accuracy on small datasets (MNIST 97%), hard to generalize to larger datasets.



PipeMare: modify the optimizer to improve AMPNet convergence

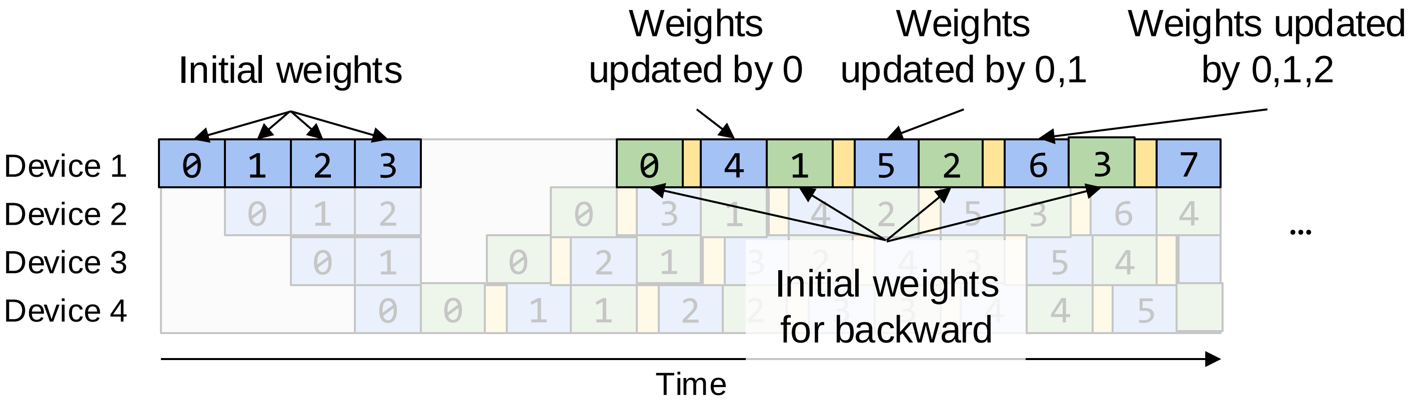
Gaunt, Alexander L., et al. "AMPNet: Asynchronous model-parallel training for dynamic neural networks." *arXiv 2017.*
Yang, Bowen, et al. "Pipemare: Asynchronous pipeline parallel dnn training." *MLSys 2021.*

Pipedream

Idea: Enforce the same version of weight for a single input batch by storing multiple weight versions.

Convergence: Similar accuracy on ImageNet with a 5x speedup compared to data parallel.

Con: No memory saving compared to single device case.

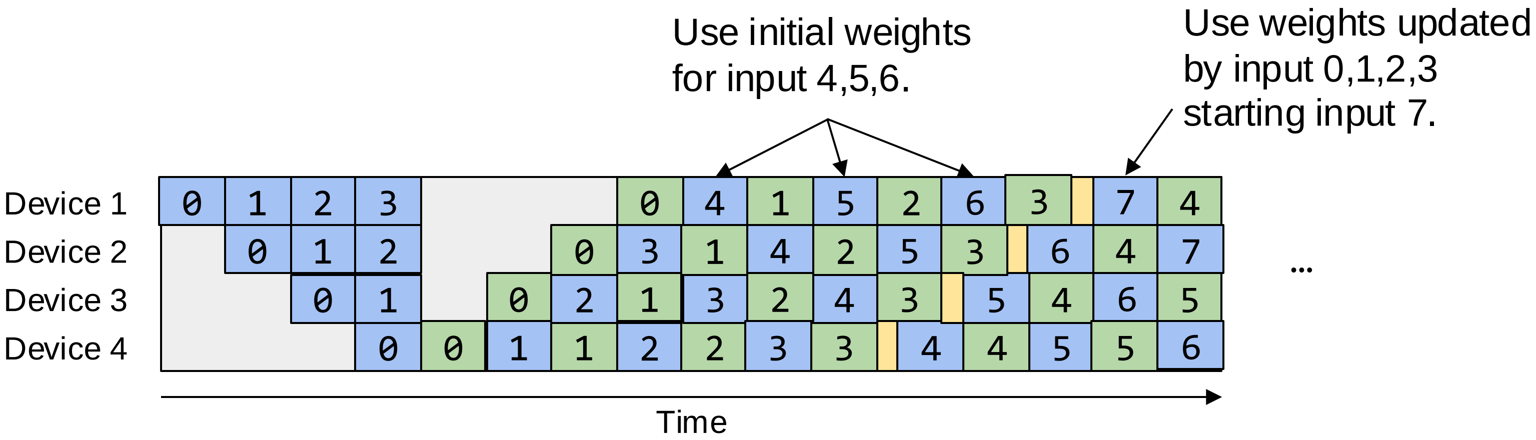


Narayanan, Deepak, et al. "PipeDream: generalized pipeline parallelism for DNN training." *SOSP 2019.*

Pipedream-2BW

Idea: Reduce Pipedream's memory usage (only store 2 copies) by updating weights less frequently. Weights always stalled by 1 update.

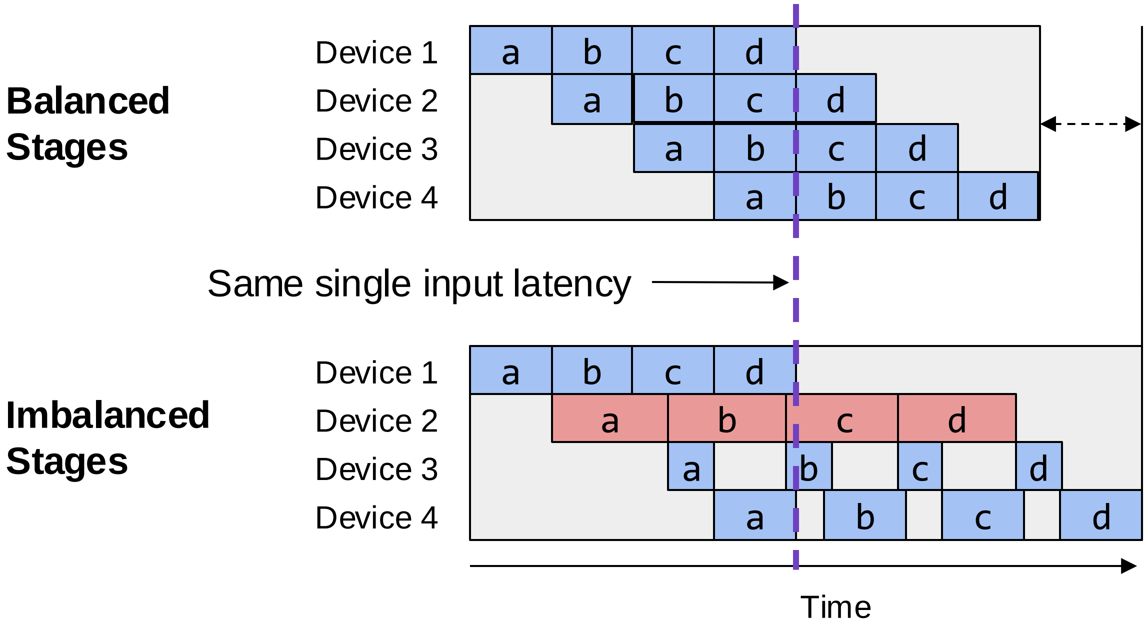
Convergence: Similar training accuracy on language models (BERT/GPT)



Narayanan, Deepak, et al. "Memory-efficient pipeline-parallel dnn training." *ICML 2021.*

Imbalanced Pipeline Stages

Pipeline schedules works best with balanced stages:



Frontier: Automatic Stage Partitioning

Goal: Minimize maximum stage latency & maximize parallelization

Reinforcement Learning Based (mainly for device placement):

1. Mirhoseini, Azalia, et al. "Device placement optimization with reinforcement learning." *ICML 2017.*
2. Gao, Yuanxiang, et al. "Spotlight: Optimizing device placement for training deep neural networks." *ICML 2018.*
3. Mirhoseini, Azalia, et al. "A hierarchical model for device placement." *ICLR 2018.*
4. Addanki, Ravichandra, et al. "Placeto: Learning generalizable device placement algorithms for distributed machine learning." *NeurIPS 2019.*
5. Zhou, Yanqi, et al. "Gdp: Generalized device placement for dataflow graphs." *Arxiv 2019.*
6. Paliwal, Aditya, et al. "Reinforced genetic algorithm learning for optimizing computation graphs." *ICLR 2020.*
7. *…*

Optimization (Dynamic Programming/Linear Programming) Based:

1. Narayanan, Deepak, et al. "PipeDream: generalized pipeline parallelism for DNN training." *SOSP 2019.*
2. Tarnawski, Jakub M., et al. "Efficient algorithms for device placement of dnn graph operators." *NeurIPS 2020.*
3. Fan, Shiqing, et al. "DAPPLE: A pipelined data parallel approach for training large models." *PPoPP 2021.*
4. Tarnawski, Jakub M., Deepak Narayanan, and Amar Phanishayee. "Piper: Multidimensional planner for dnn parallelization." *NeurIPS 2021.*
5. Zheng, Lianmin, et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning." *OSDI 2022.*
6. *…*

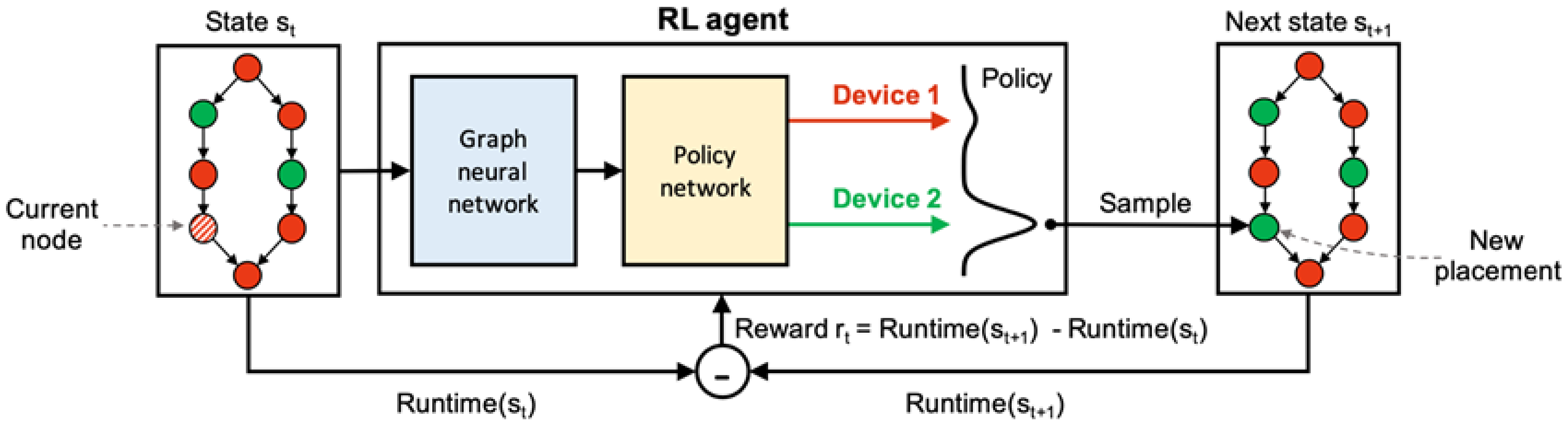
RL-Based Partitioning Algorithm

State: Device assignment plan for a computational graph.

Action: Modify the device assignment of a node.

Reward: Latency difference between the new and old placements.

Trained with **policy gradient** algorithm.



Addanki, Ravichandra, et al. "Placeto: Learning generalizable device placement algorithms for distributed machine learning." NeurIPS 2019.

Inter-operator Parallelism Summary

Idea: Assign different operators of the computational graph to different devices and executed in a pipelined fashion.

Method	General computational graph	No pipeline bubbles	Same convergence as single device
Device Placement	❌	❌	✅
Synchronous Schedule	✅	❌	✅
Asynchronous Schedule	✅	✅	❌

Stage Partitioning: Imbalance stage → More pipeline bubble

RL-Based / **Optimization-Based** Automatic Stage Partitioning

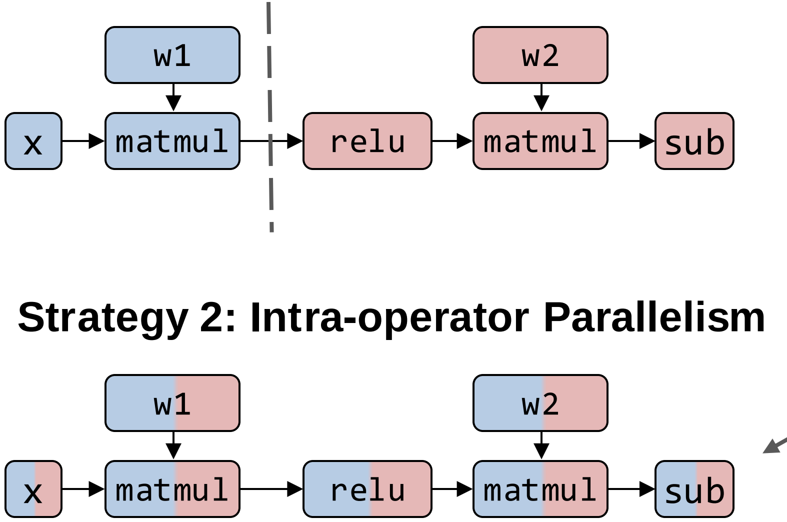
Where We Are

- Motivation
- History
- Parallelism Overview
- Data parallelism
- Model parallelism
    - Inter-op parallelism
    - **Intra-op parallelism**
- Auto-parallelization

Recap: Intra-op and Inter-op

Strategy 1: Inter-operator Parallelism

Strategy 2: Intra-operator Parallelism



This section:

1. How to parallelize an **operator** ?
2. How to parallelize a **graph** ?

Parallelize One Operator

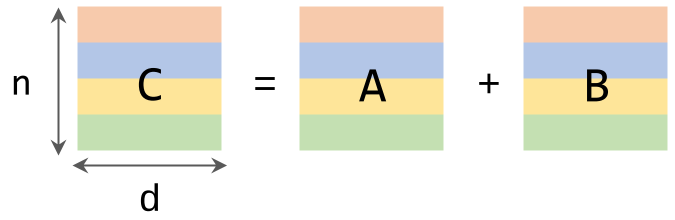
Element-wise operators

```
for n in range(0, N):
  for d in range(0, D):
    C[n,d] = A[n,d] + B[n,d]
```

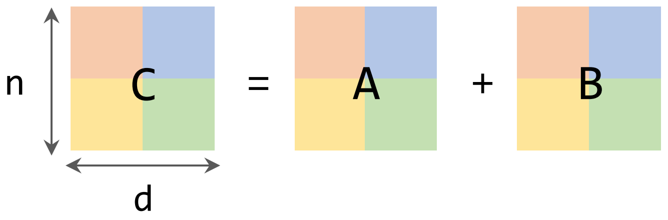
No dependency on the two for-loops.
Can arbitrarily split the for-loops on different devices.

device 1 device 2 device 3 device 4

Parallelize loop n



Parallelize both loop n and loop d



a lot of
other variants
…

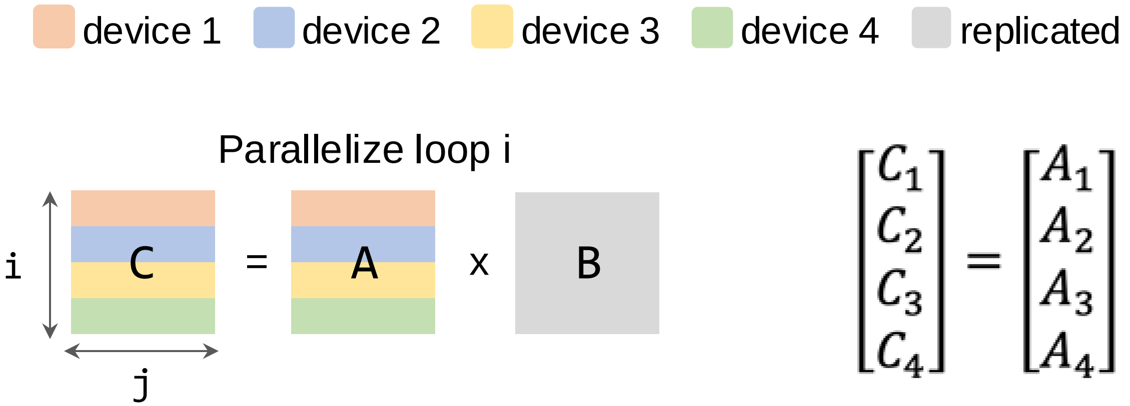
Parallelize One Operator

Matrix multiplication

No dependency on the two spatial for-loops.
Can arbitrarily split the for-loops on different devices.

```
for i in range(0, N):
  for j in range(0, M):
    for k in range(0, K):
      C[i,j] = C[i,j] + A[i,k] x B[k,j]
```

Accumulation on this reduction loop.
Have to accumulate partial results if we split this for-loop



$$\begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \times B$$

Parallelize One Operator

Matrix multiplication

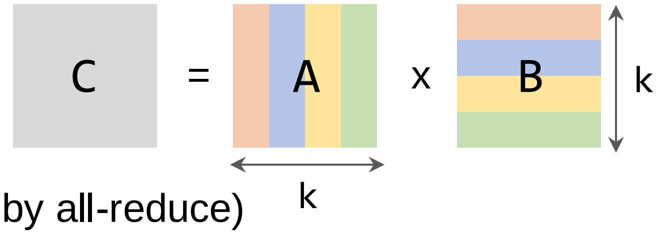
No dependency on the two spatial for-loops.
Can arbitrarily split the for-loops on different devices.

```
for i in range(0, N):
  for j in range(0, M):
    for k in range(0, K):
      C[i,j] = C[i,j] + A[i,k] x B[k,j]
```

Accumulation on this reduction loop.
Have to accumulate partial results if we split this for-loop

device 1 device 2 device 3 device 4 replicated

Parallelize loop k



(got by all-reduce)

$$C = [A_1 \;\; A_2 \;\; A_3 \;\; A_4] \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{bmatrix} = A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4$$

Parallelize One Operator

Matrix multiplication

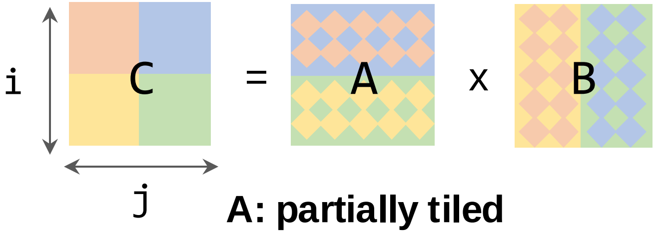
No dependency on the two spatial for-loops.
Can arbitrarily split the for-loops on different devices.

```
for i in range(0, N):
  for j in range(0, M):
    for k in range(0, K):
      C[i,j] = C[i,j] + A[i,k] x B[k,j]
```

Accumulation on this reduction loop.
Have to accumulate partial results if we split this for-loop

device 1 device 2 device 3 device 4

Parallelize loop i and j

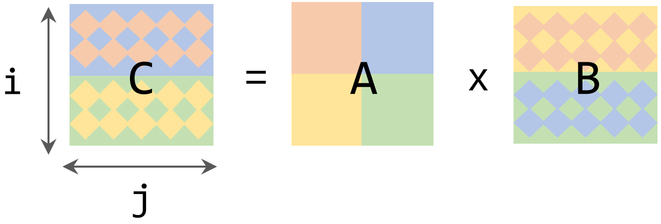


A: partially tiled

Device 1 and 2 hold a replicated tile

Device 3 and 4 hold a replicated tile

Parallelize loop i and k



C: got by all-reduce

a lot of other variants
…

Parallelize One Operator

2D Convolution

Simple spatial loops. Can be arbitrarily split.

Stencil computation loops. Splitting these requires careful boundary handling.

Reduction loop. Need to accumulate partial results.

Reduction loops. But usually too small (<= 5) for parallelization.

```
for n in range(0, N):
  for co in range(0, CO):
    for h in range(0, H):
      for w in range(0, W):
        for ci in range(0, CI):
          for kh in range(0, KH):
            for kw in range(0, KW):
              C[n,co,h,w] += A[n,co,h+kh,w+kw] x B[kh,kw,co,ci]
```

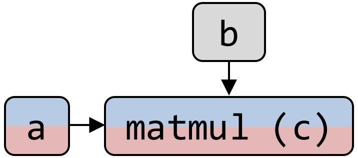
Simple case: Parallelize loop n, co, ci, then the parallelization strategies are almost the same as matmul's.

Complicated case: Parallelize loop h and w

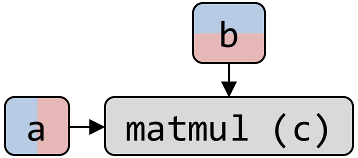
Data Parallelism as A Case of Intra-op Parallelism

Replicated Row-partitioned Column-partitioned

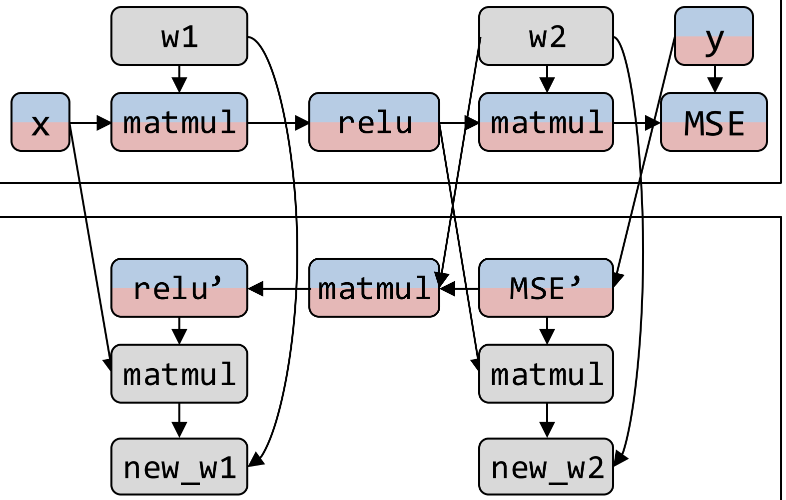
Matmul Parallelization Type 1
communication cost = 0



Matmul Parallelization Type 2
communication cost = all-reduce(c)



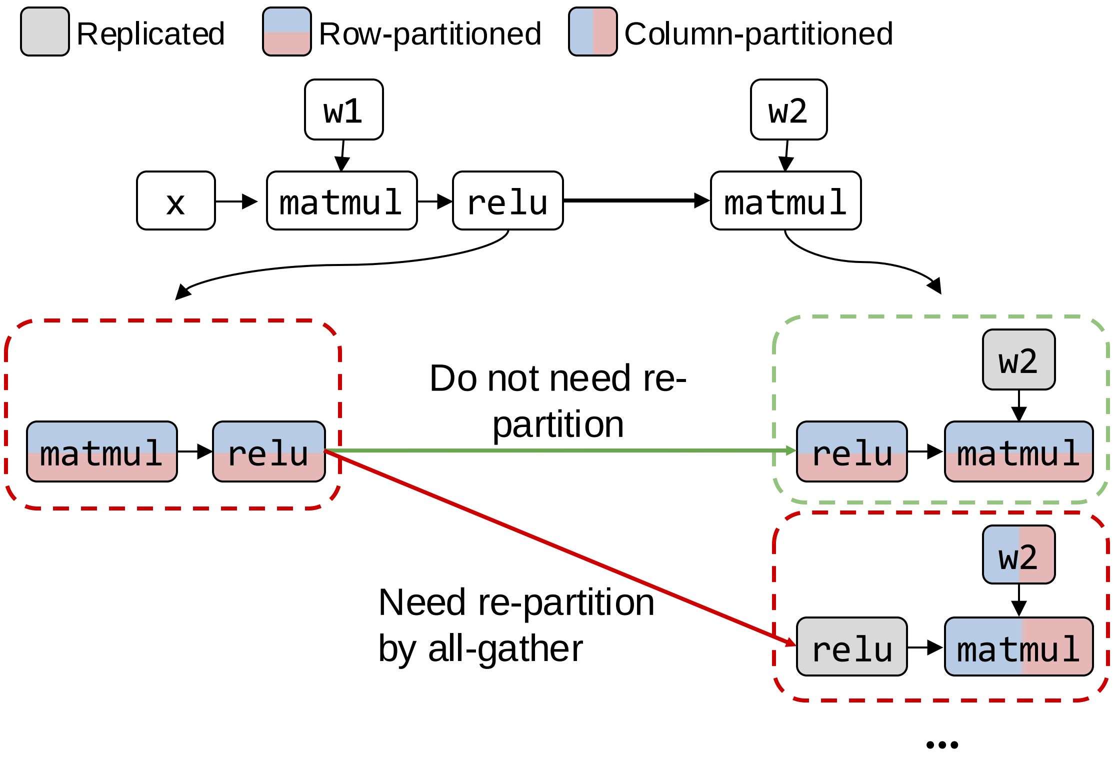
Forward Pass
Two “Type 1” matmuls: no communication



Backward Pass
One “Type 1” matmul: no communication
Two “Type 2” matmuls: require all-reduce

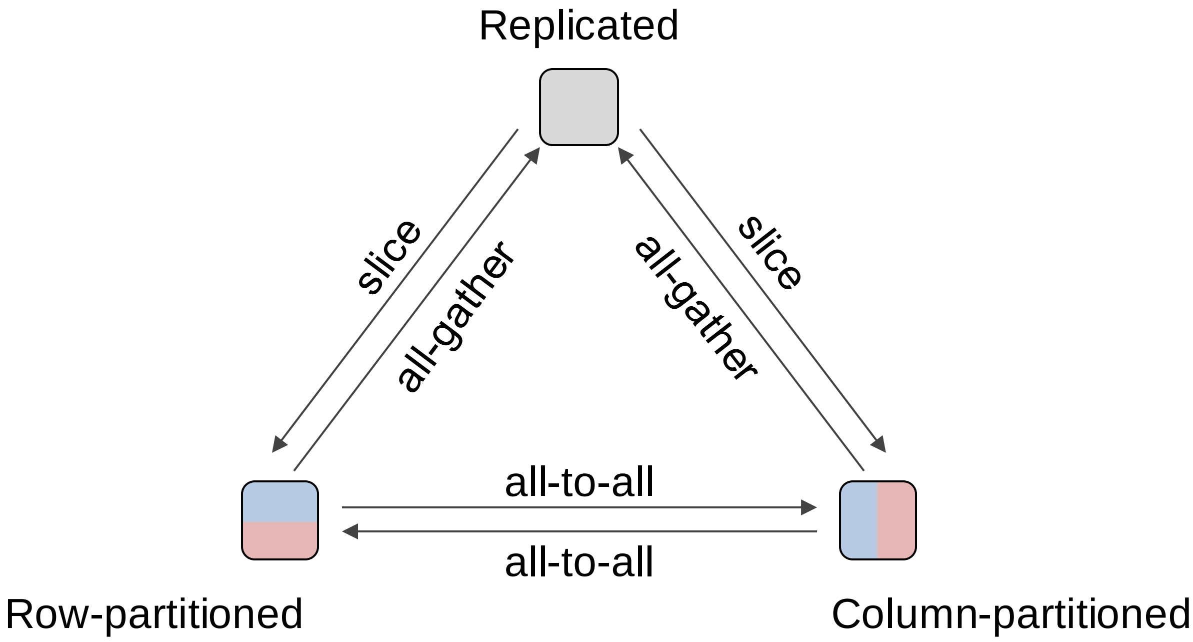
Re-partition Communication Cost

Different operators' parallelization strategies require different partition format of the same tensor



Re-partition Communication Cost

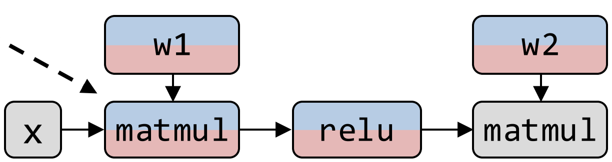
Different operators' parallelization strategies require different partition format of the same tensor



Parallelize All Operators in a Graph

Problem

Pick a parallel strategy
of each operator



Minimize Node costs (computation + communication) + Edge costs (re-partition communication)

Solution

Manual design
Randomized search
Dynamic programming
Integer linear programming

Important Projects

Model-specific Intra-op Parallel Strategies

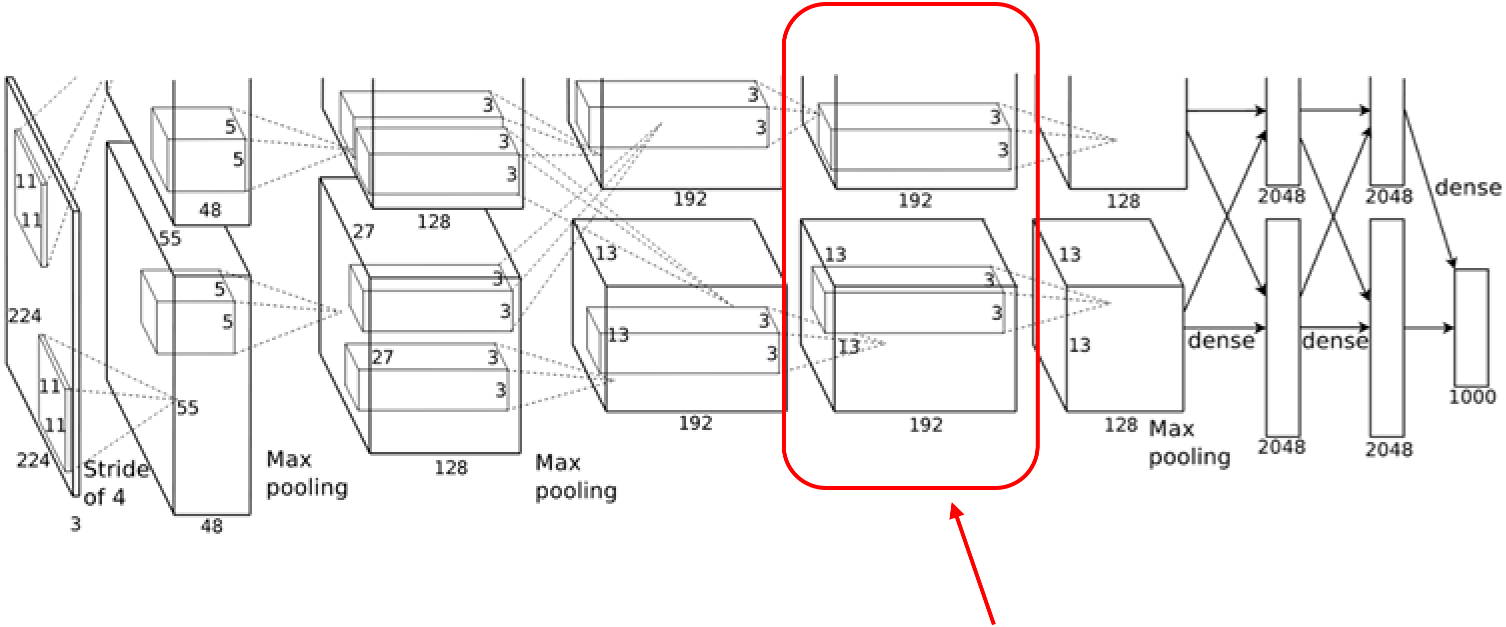
- AlexNet
- Megatron-LM
- GShard MoE

Systems for Intra-op Parallelism

- ZeRO
- Mesh-Tensorflow
- GSPMD
- Tofu
- FlexFlow

AlexNet

Result: increase top-1 accuracy by 1.7%



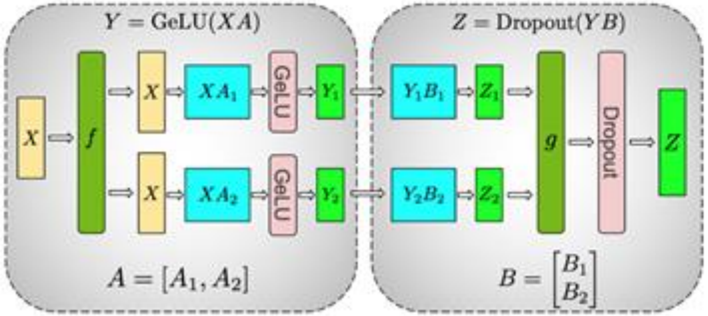
Assign a group convolution layer to 2 GPUs

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *NeurIPS 2012*

Megaton-LM

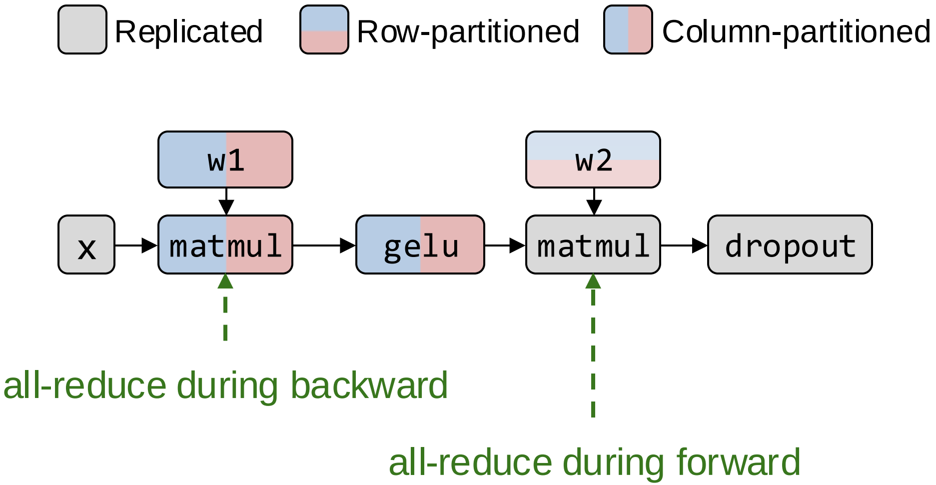
Result: a large language model with 8.3B parameters that outperforms SOTA results

Figure 3 from the paper：
How to partition the MLP in the transformer.



(a) MLP

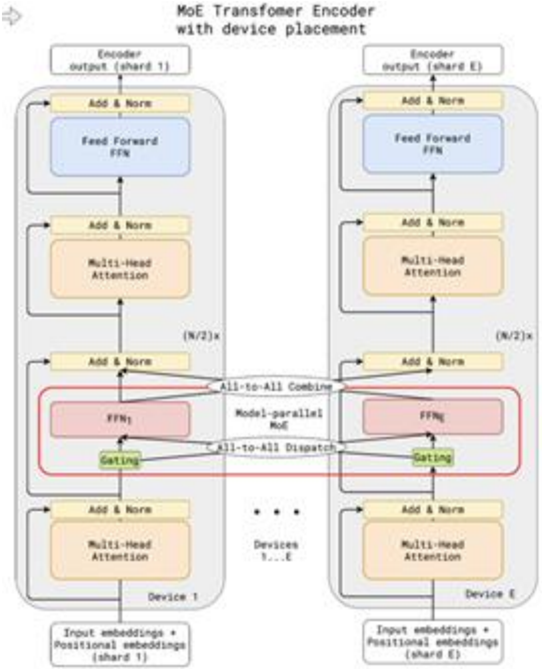
Illustrated with the notations in this tutorial



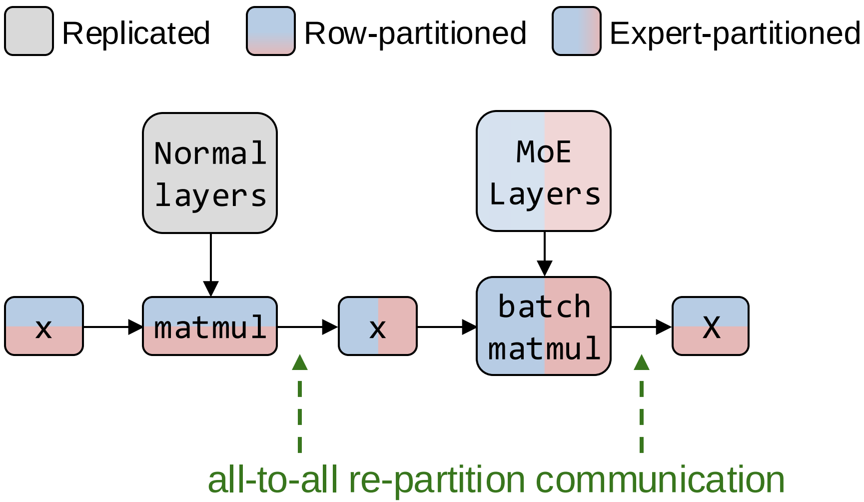
Shoeybi, Mohammad, et al. "Megatron-LM: Training multi-billion parameter language models using model parallelism."

GShard MoE

Result: a multi-language translation model with 600B parameters that outperforms SOTA



Illustrated with the notations in this class



Lepikhin, Dmitry, et al. "GShard: Scaling giant models with conditional computation and automatic sharding." *ICLR 2021*

ZeRO Optimizer

Problem
Data parallelism replicates gradients, optimizer states and model weights on all devices.

Idea
Partition gradients, optimizer states and model weights.

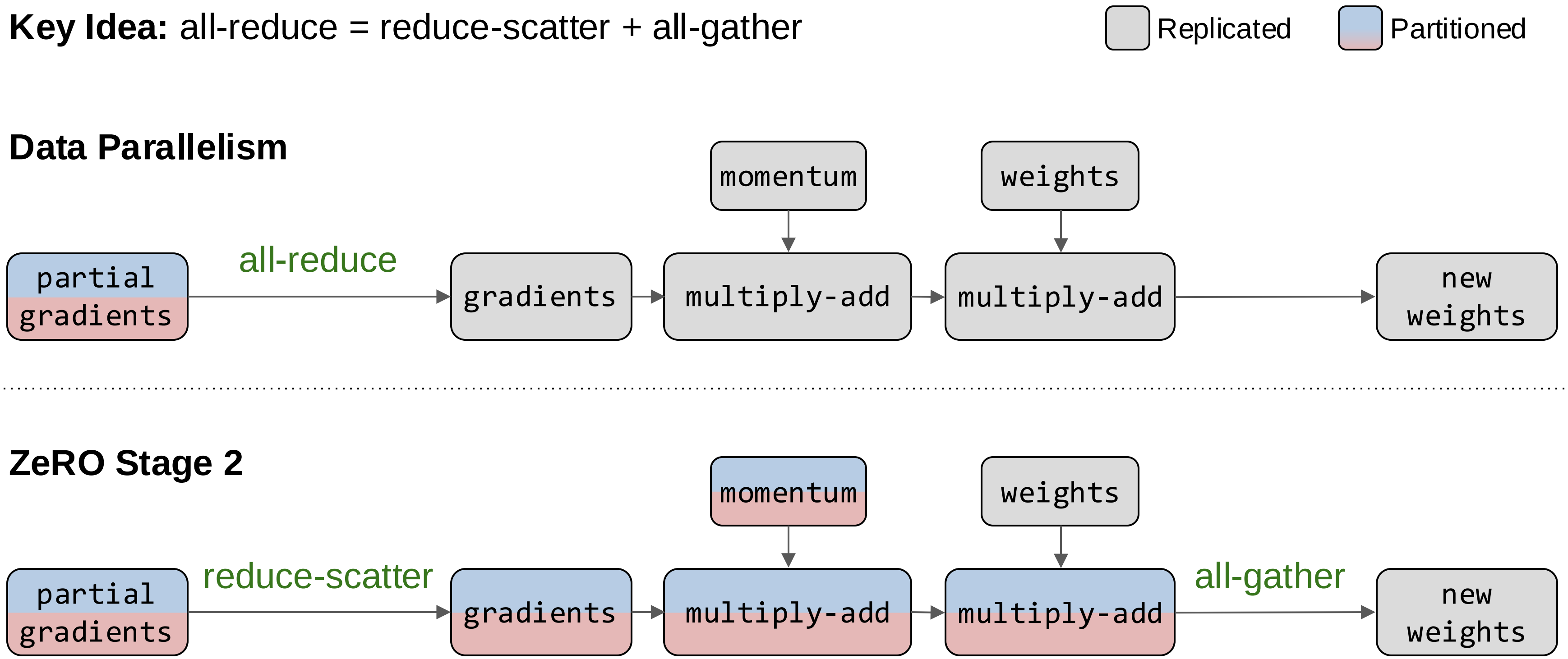
M is the number of parameters, N is the number of devices.

	Optimizer States (12M)	Gradients (2M)	Model Weights (2M)	Memory Cost	Communication Cost
Data Parallelism	Replicated	Replicated	Replicated	$16M$	all-reduce(2M)
ZeRO Stage 1	Partitioned	Replicated	Replicated	$4M + \frac{12M}{N}$	all-reduce(2M)
ZeRO Stage 2	Partitioned	Partitioned	Replicated	$2M + \frac{14M}{N}$	all-reduce(2M)
ZeRO Stage 3	Partitioned	Partitioned	Partitioned	$\frac{16M}{N}$	1.5 all-reduce(2M)

Rajbhandari, Samyam, et al. "Zero: Memory optimizations toward training trillion parameter models." *SC 2020*

ZeRO Stage 2

Key Idea: all-reduce = reduce-scatter + all-gather



Same communication cost but save memory by partitioning more tensors

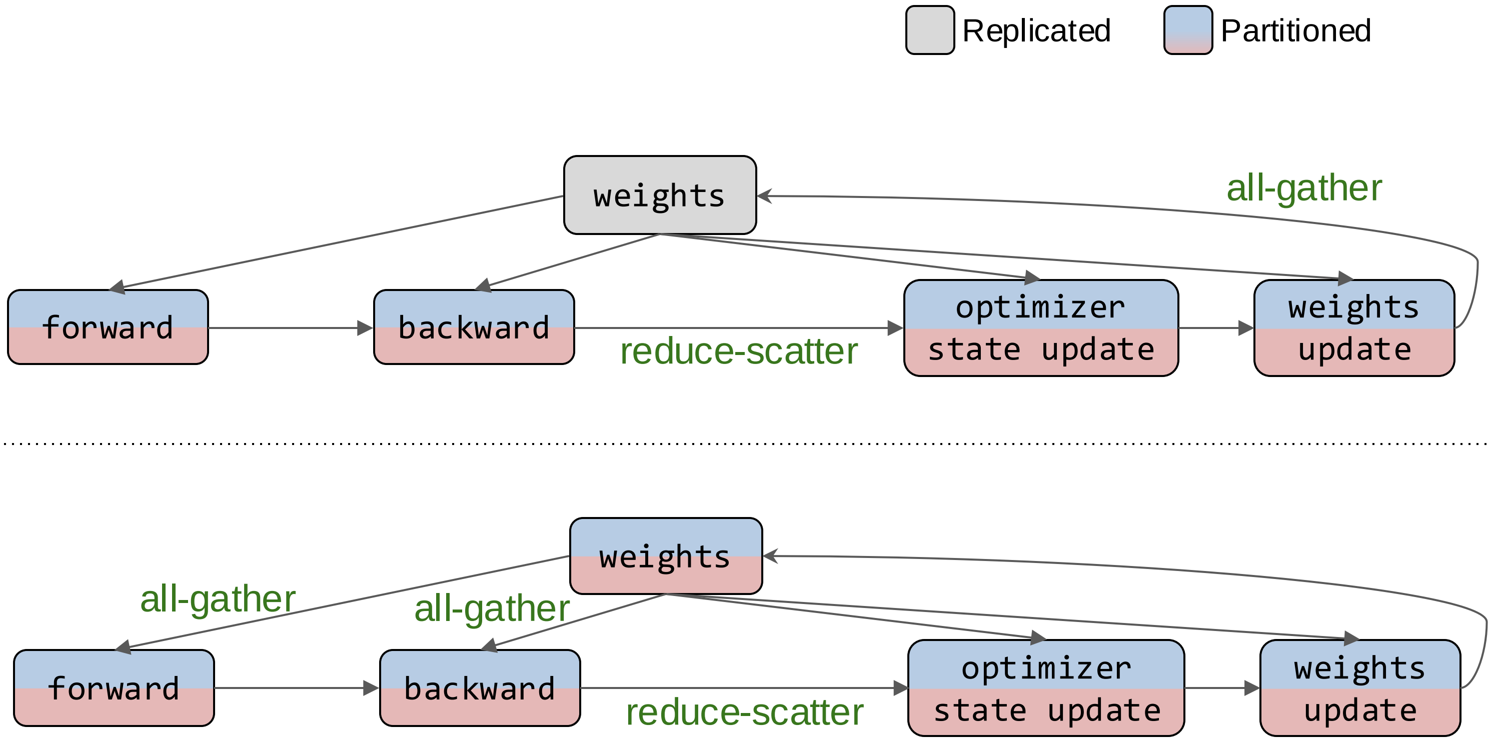
ZeRO Stage 3

ZeRO Stage 2

communication cost
= all-reduce

ZeRO Stage 3

communication cost
= 1.5 all-reduce



ZeRO Optimizer

Problem
Data parallelism replicates gradients, optimizer states and model weights on all devices.

Idea
Partition gradients, optimizer states and model weights.

M is the number of parameters, N is the number of devices.

	Optimizer States (12M)	Gradients (2M)	Model Weights (2M)	Memory Cost	Communication Cost
Data Parallelism	Replicated	Replicated	Replicated	$16M$	all-reduce(2M)
ZeRO Stage 1	Partitioned	Replicated	Replicated	$4M + \frac{12M}{N}$	all-reduce(2M)
ZeRO Stage 2	Partitioned	Partitioned	Replicated	$2M + \frac{14M}{N}$	all-reduce(2M)
ZeRO Stage 3	Partitioned	Partitioned	Partitioned	$\frac{16M}{N}$	1.5 all-reduce(2M)

Rajbhandari, Samyam, et al. "Zero: Memory optimizations toward training trillion parameter models." *SC 2020*

Mesh-Tensorflow

Map tensor dimension to mesh dimension for parallelism

```
...
batch = mtf.Dimension("batch", b)
io = mtf.Dimension("io", d_io)
hidden = mtf.Dimension("hidden", d_h)
# x.shape == [batch, io]
w = mtf.get_variable("w", shape=[io, hidden])
bias = mtf.get_variable("bias", shape=[hidden])
v = mtf.get_variable("v", shape=[hidden, io])
h = mtf.relu(mtf.einsum(x, w, output_shape=[batch, hidden]) + bias)
y = mtf.einsum(h, v, output_shape=[batch, io])
...
```

Tensor dimension

```
mesh_shape = [("rows", r), ("cols", c)]
computation_layout = [("batch", "rows"), ("hidden", "cols")]
```

Mesh dimension

Mapping

Shazeer, Noam, et al. "Mesh-tensorflow: Deep learning for supercomputers." *NeurIPS* 2018.

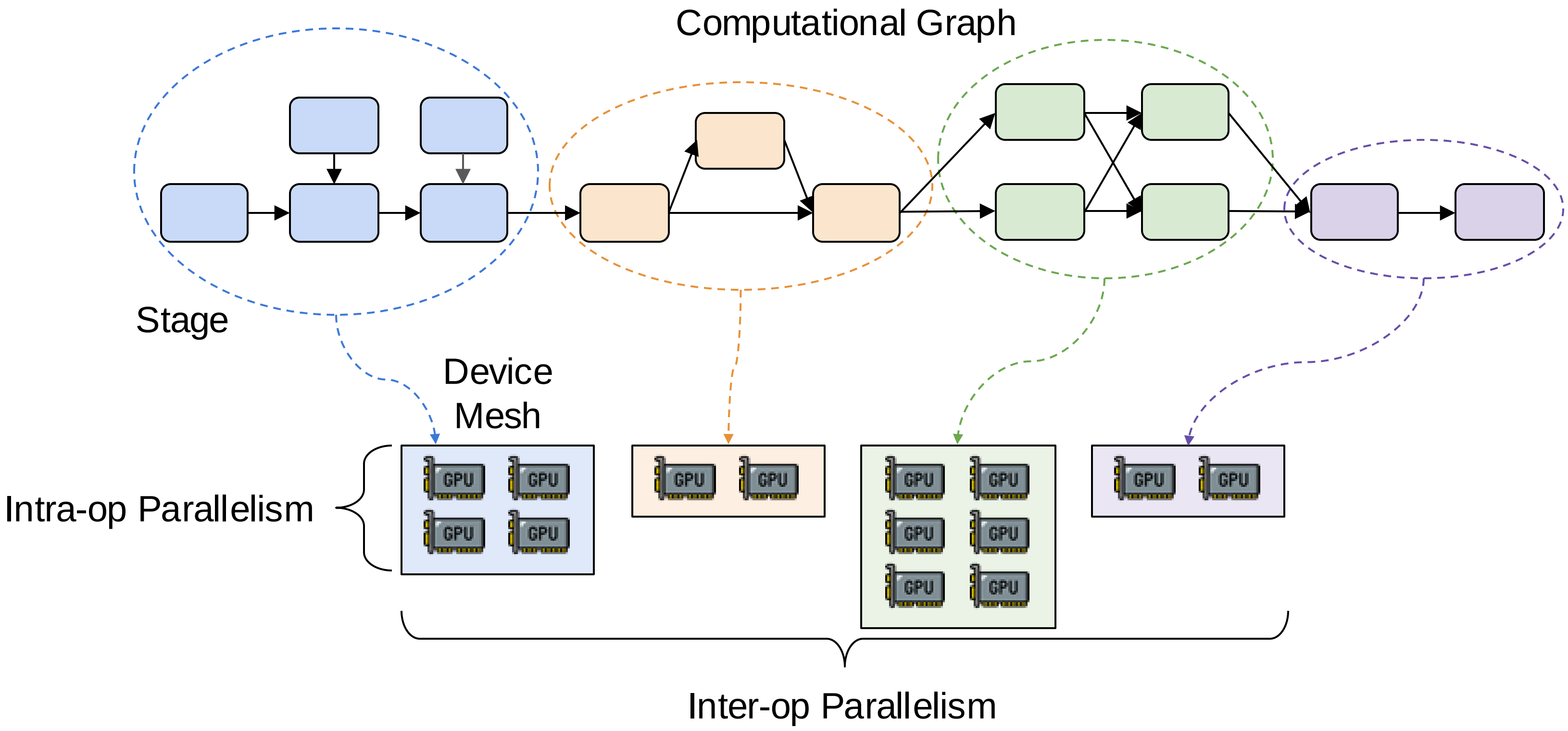
GSPMD

- Use annotations to specify partition strategy
- Propagate the annotations to whole graph
- Use compiler to generate SPMD (Single Program Multiple Data) parallel executables

```
  # Partition inputs along group (G) dim.
+ inputs = split(inputs, 0, D)
  # Replicate the gating weights
+ wg = replicate(wg)
  gates = softmax(einsum("GSM,ME->GSE", inputs, wg))
  combine_weights, dispatch_mask = Top2Gating(gating_logits)
  dispatched_expert_inputs = einsum(
    "GSEC,GSM->EGCM", dispatch_mask, reshaped_inputs)
  # Partition dispatched inputs along expert (E) dim.
+ dispatched_expert_inputs = split(dispatched_expert_inputs, 0, D)
  h = einsum("EGCM,EMH->EGCH", dispatched_expert_inputs, wi)
  ...
```

Xu, Yuanzhong, et al. "GSPMD: general and scalable parallelization for ML computation graphs." *arXiv 2021*

Combine Intra-op Parallelism and Inter-op Parallelism



Narayanan, Deepak, et al. "Efficient large-scale language model training on gpu clusters using megatron-lm." *SC 2021*
Zheng, Lianmin, et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning." *OSDI 2022*

Intra-operator Parallelism Summary

- We can parallelize a single operator by exploiting its internal parallelism

- To do this for a whole computational graph, we need to choose strategies for all nodes in the graph to minimize the communication cost

- Intra-op and inter-op can be combined

Other Techniques for Training Large Models

System-level Memory Optimizations

- Rematerialization/Gradient Checkpointing
- Swapping

ML-level Optimizations

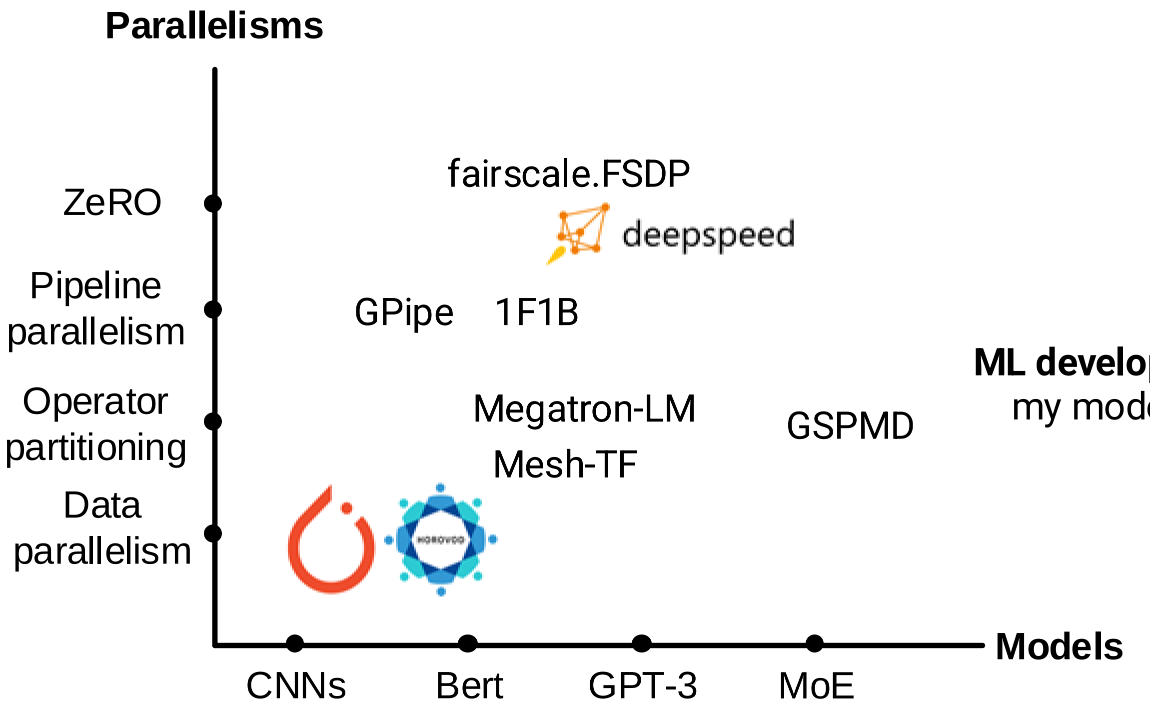
- Quantization
- Sparsification
- Low-rank approximation

Chen, Tianqi, et al. "Training deep nets with sublinear memory cost." *arXiv 2016*
Rajbhandari, Samyam, et al. "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning." *SC* 2021.
Tang, Hanlin, et al. "1-bit adam: Communication efficient large-scale training with adam's convergence speed." *ICML* 2021.
Shazeer, Noam, and Mitchell Stern. "Adafactor: Adaptive learning rates with sublinear memory cost." *ICML* 2018.

Where We Are

- Motivation
- History
- Parallelism Overview
- Data parallelism
- Model parallelism
  - Inter-op parallelism
  - Intra-op parallelism
- **Auto-parallelization**

Auto-parallelization: Motivation



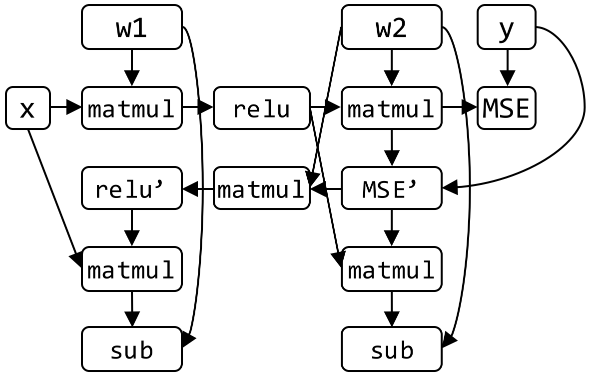
ML developer: which one is for my model and my cluster?

Auto-parallelization: Problem

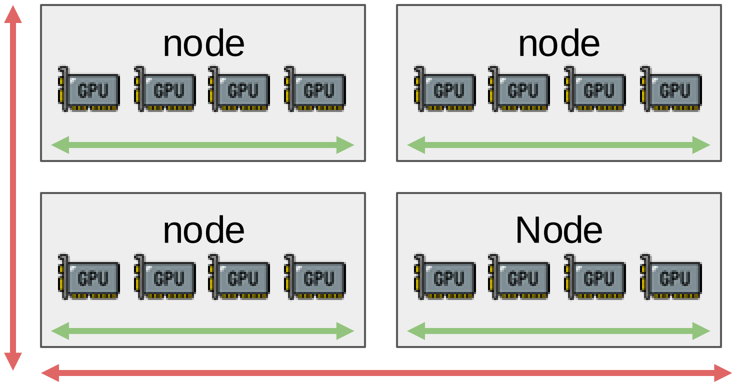
$$\max_{\text{strategy}} \ \text{Performance(Model, Cluster)}$$
$$s.t. \ \text{strategy} \in \text{Inter-op} \cup \text{Intra-op}$$

Auto-parallelization: Problem

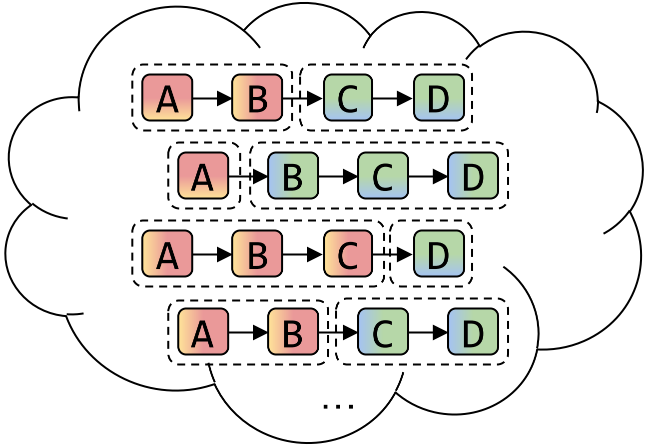
Model



Cluster



Strategy



The Search Space is Huge

#ops in a real model (nodes to color)	#op types (type of nodes)	#devices on a cluster (available colors)
100 - 10K	**80 - 200+**	**10s - 1000s**

Automatic Parallelization Methods

Search-based methods

- MCMC:
  - ➔ [Jia et al., 2018]
  - ➔ [Jia et al., 2019]
- Heuristics
  - ➔ [Fan et al., 2021]

Learning-based methods

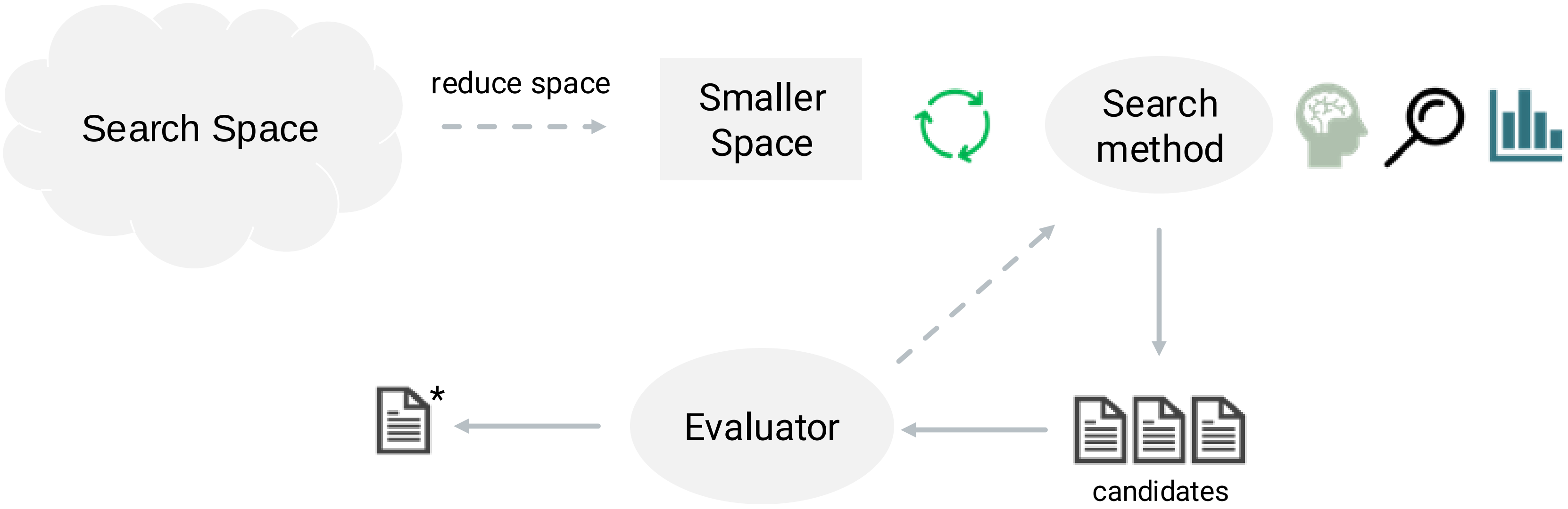
- Reinforcement Learning:
  - ➔ [Mirhoseini et al., 2017]
  - ➔ [Mirhoseini et al., 2018]
  - ➔ [Addanki, et al., 2019]
- ML-based cost model:
  - ➔ [Chen et al., 2018],
  - ➔ [Zhou et al., 2020],
  - ➔ [Zhang, 2020]
- Bayesian optimization:
  - ➔ [Sergeev et al., 2018],
  - ➔ [Peng et al., 2019]

Optimization-based methods

- Dynamic programming
  - ➔ [Wang, et al., 2018]
  - ➔ [Narayanan, et al., 2019]
  - ➔ [Li, et al., 2021]
  - ➔ [Narayanan, et al., 2012]
  - ➔ [Tarnawski, et al., 2020]
  - ➔ [Tarnawski, et al., 2021]
- Integer linear programming
  - ➔ [Tarnawski, et al., 2020]
- Hierarchical Optimization
  - ➔ [Zheng, et al., 2022]

The complete list of references is available on the tutorial website

General Recipe



Automatic Parallelization Methods

Search-based methods

- MCMC:
  - ➔ [Jia et al., 2018]
  - ➔ [Jia et al., 2019]
- Heuristics
  - ➔ [Fan et al., 2021]

Learning-based methods

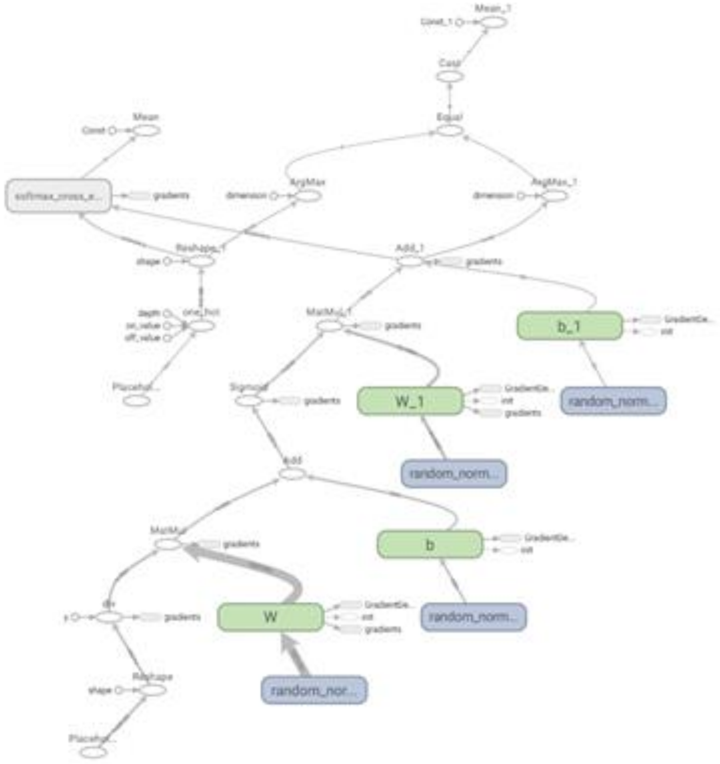
- Reinforcement Learning:
  - ➔ **[Mirhoseini et al., 2017]**
  - ➔ [Mirhoseini et al., 2018]
  - ➔ [Addanki, et al., 2019]
- ML-based cost model:
  - ➔ [Chen et al., 2018],
  - ➔ [Zhou et al., 2020],
  - ➔ [Zhang, 2020]
- Bayesian optimization:
  - ➔ [Sergeev et al., 2018],
  - ➔ [Peng et al., 2019]

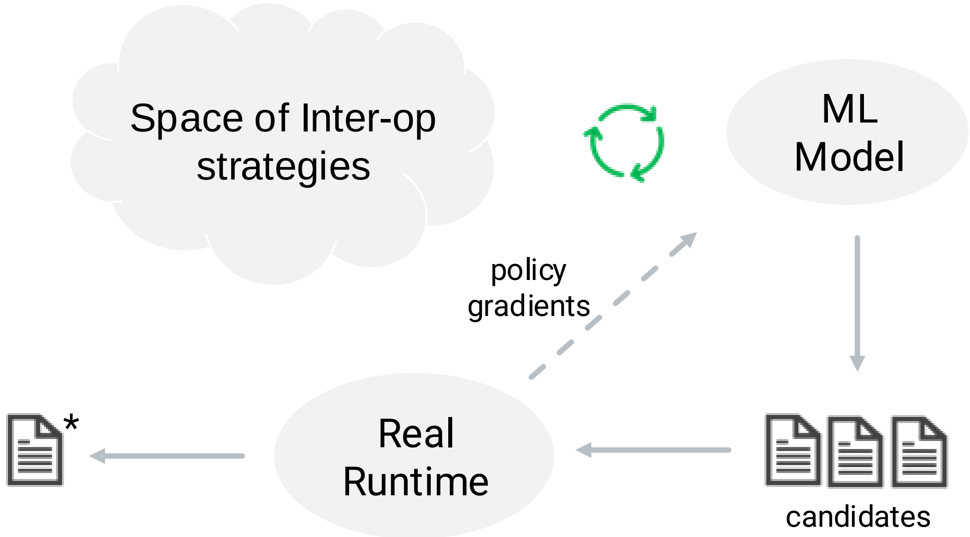
Optimization-based methods

- Dynamic programming
  - ➔ [Wang, et al., 2018]
  - ➔ [Narayanan, et al., 2019]
  - ➔ [Li, et al., 2021]
  - ➔ [Narayanan, et al., 2012]
  - ➔ [Tarnawski, et al., 2020]
  - ➔ [Tarnawski, et al., 2021]
- Integer linear programming
  - ➔ [Tarnawski, et al., 2020]
- Hierarchical optimization
  - ➔ **[Zheng, et al., 2022]**

The complete list of references is available on the tutorial website

ColocRL (a.k.a. Device Placement Optimization)





Mirhoseini, et al. "Device Placement Optimization with Reinforcement Learning." *ICML* 2017.

ColocRL: Model

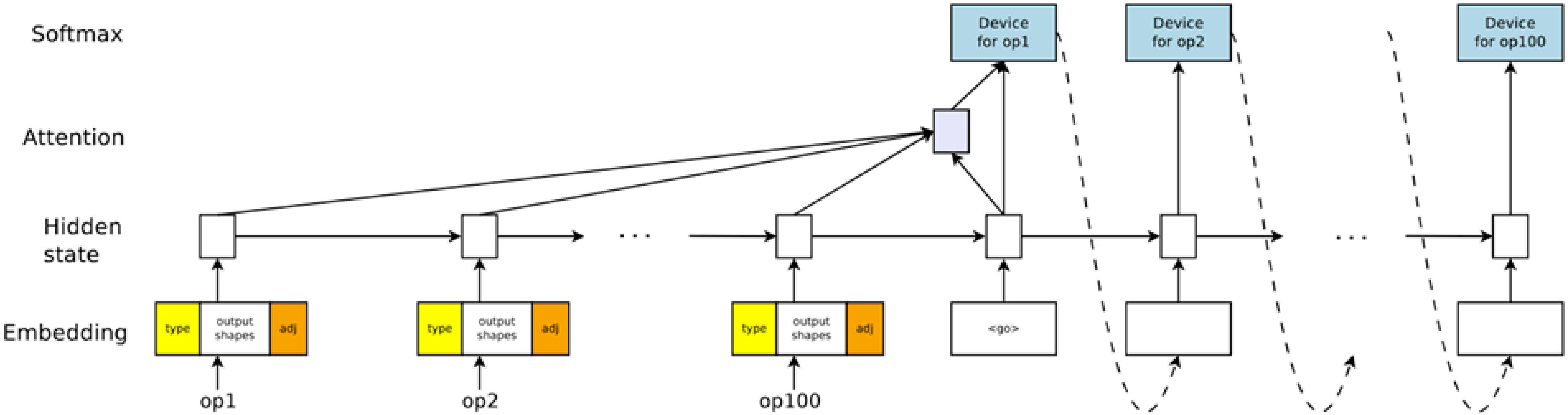


Figure from [Mirhoseini et al., ICML 2017]

Mirhoseini, et al. "Device Placement Optimization with Reinforcement Learning." *ICML* 2017.

ColocRL: Training

$$\mathbb{E}_{\mathcal{P}\sim\pi(\mathcal{P}\mid\mathcal{G};\theta)}[R(\mathcal{P})||\mathcal{G}]$$

$\mathcal{G}$: computational graph

$\mathcal{R}(\mathcal{P})$: Real runtime of a placement

$\pi(\cdot)$: output distributed of the RNN

Mirhoseini, et al. "Device Placement Optimization with Reinforcement Learning." *ICML* 2017.

ColocRL: Other Improvement

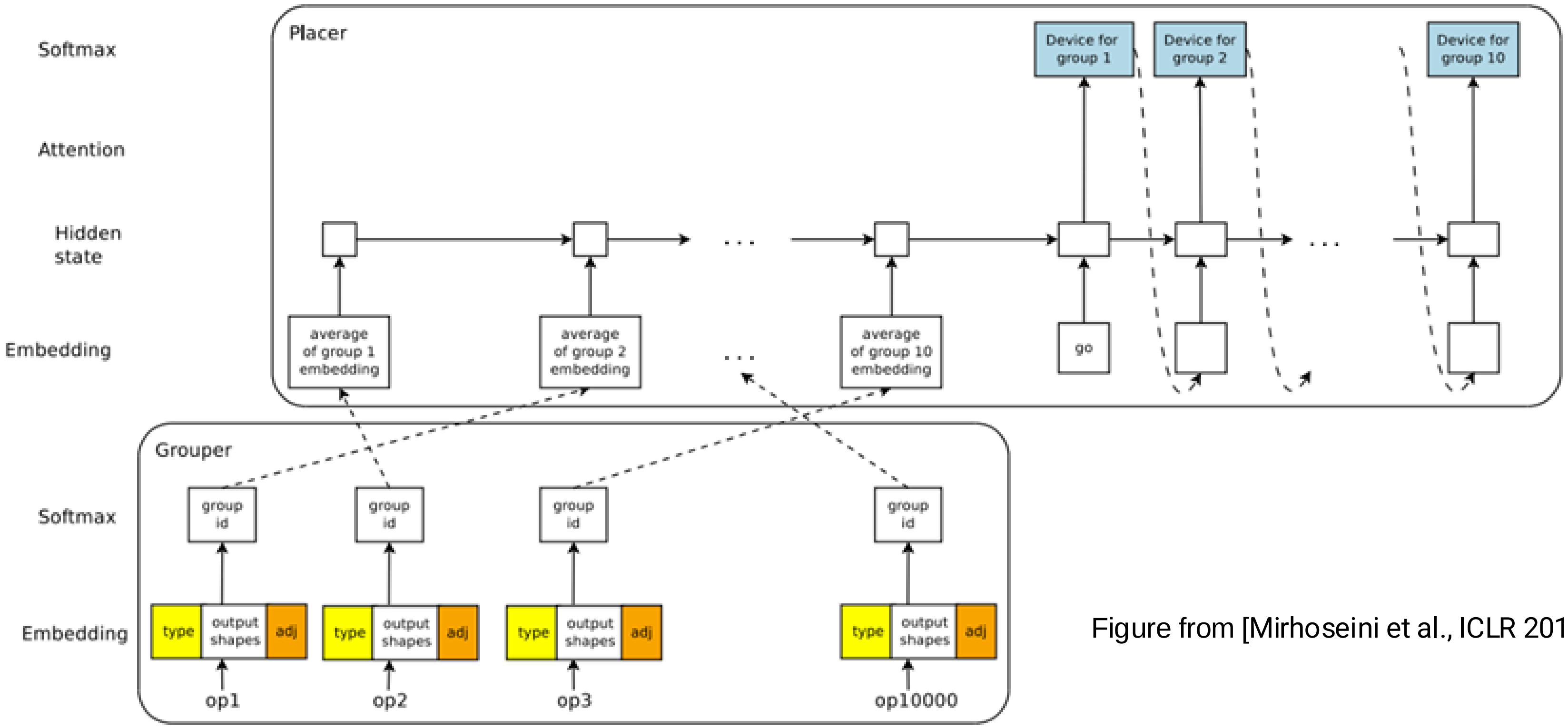
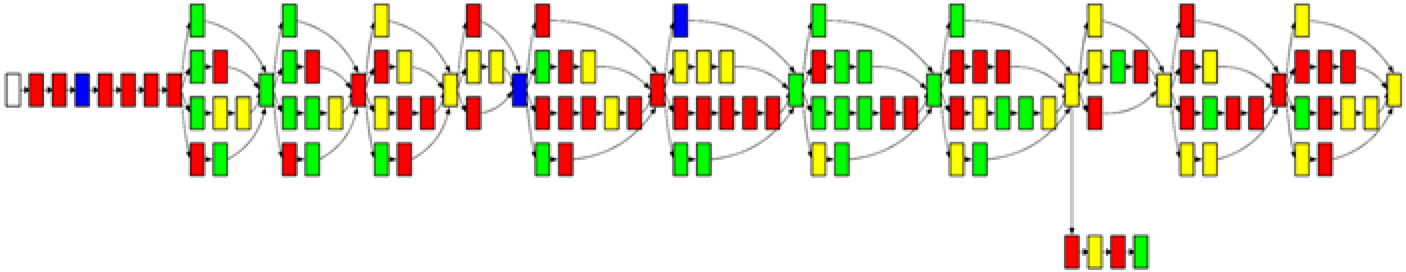


Figure from [Mirhoseini et al., ICLR 2018]

Mirhoseini, et al. "A Hierarchical Model for Device Placement." *ICLR* 2018.

Results Discussion



Tasks	Single-CPU	Single-GPU	#GPUs	Scotch	MinCut	Expert	RL-based	Speedup
RNNLM (batch 64)	6.89	**1.57**	2	13.43	11.94	3.81	**1.57**	0.0%
			4	11.52	10.44	4.46	**1.57**	0.0%
NMT (batch 64)	10.72	OOM	2	14.19	11.54	4.99	**4.04**	23.5%
			4	11.23	11.78	4.73	**3.92**	20.6%
Inception-V3 (batch 32)	26.21	**4.60**	2	25.24	22.88	11.22	**4.60**	0.0%
			4	23.41	24.52	10.65	**3.85**	19.0%

Figure and table from [Mirhoseini et al., ICML 2017]

Automatic Parallelization Methods

Search-based methods

- MCMC:
  - ➔ [Jia et al., 2018]
  - ➔ [Jia et al., 2019]
- Heuristics
  - ➔ [Fan et al., 2021]

Learning-based methods

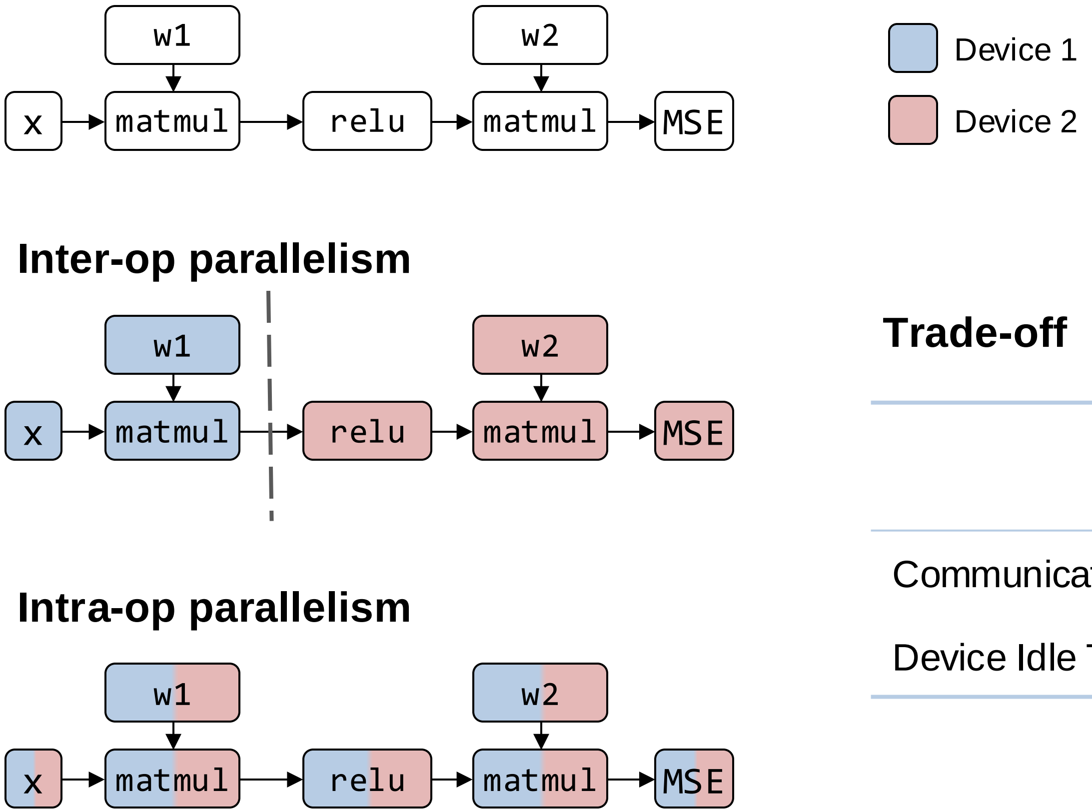
- Reinforcement Learning:
  - ➔ [Mirhoseini et al., 2017]
  - ➔ [Mirhoseini et al., 2018]
  - ➔ [Addanki, et al., 2019]
- ML-based cost model:
  - ➔ [Chen et al., 2018],
  - ➔ [Zhou et al., 2020],
  - ➔ [Zhang, 2020]
- Bayesian optimization:
  - ➔ [Sergeev et al., 2018],
  - ➔ [Peng et al., 2019]

Optimization-based methods

- Dynamic programming
  - ➔ [Wang, et al., 2018]
  - ➔ [Narayanan, et al., 2019]
  - ➔ [Li, et al., 2021]
  - ➔ [Narayanan, et al., 2012]
  - ➔ [Tarnawski, et al., 2020]
  - ➔ [Tarnawski, et al., 2021]
- Integer linear programming
  - ➔ [Tarnawski, et al., 2020]
- Hierarchical optimization
  - ➔ **Alpa [Zheng, et al., 2022]**

The complete list of references is available on the tutorial website

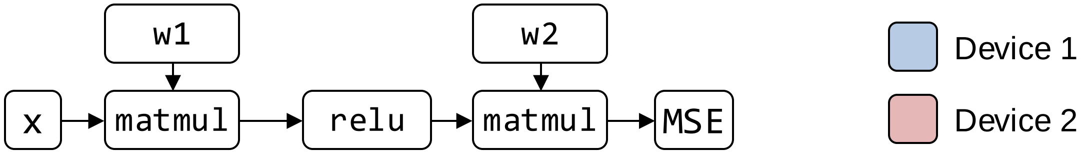
Optimization-based Method: Alpa



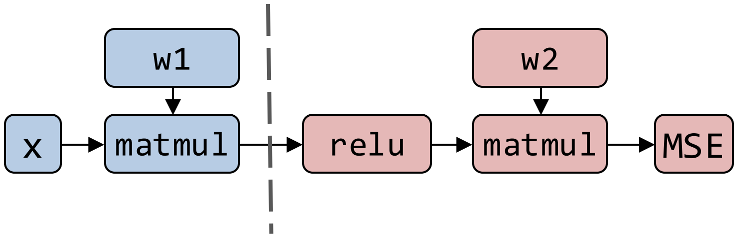
Trade-off

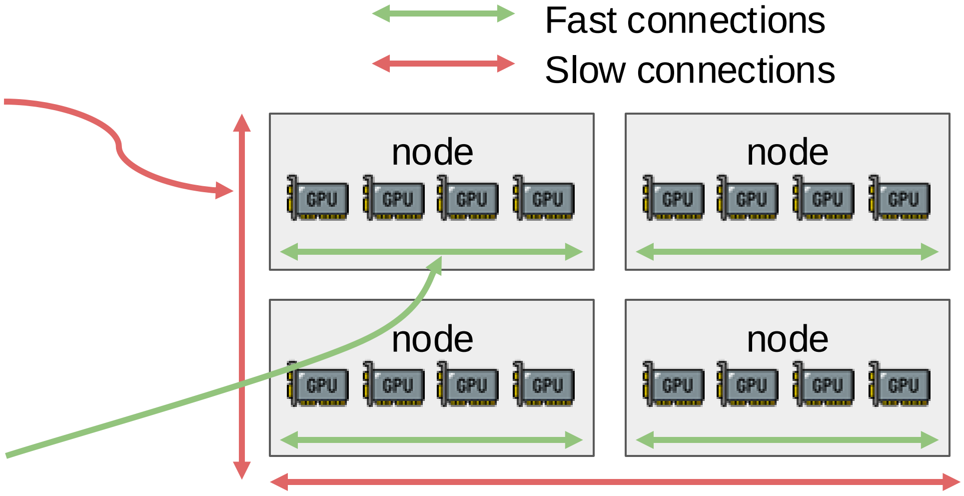
	Inter-operator Parallelism	**Intra-operator Parallelism**
Communication	Less	More
Device Idle Time	More	Less

Alpa Rationale

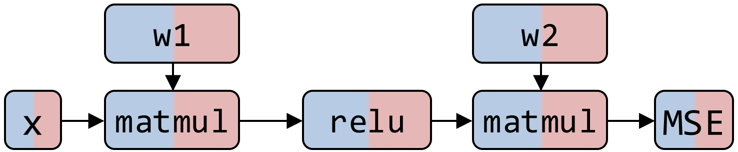


Inter-op parallelism



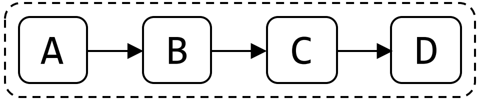


Intra-op parallelism

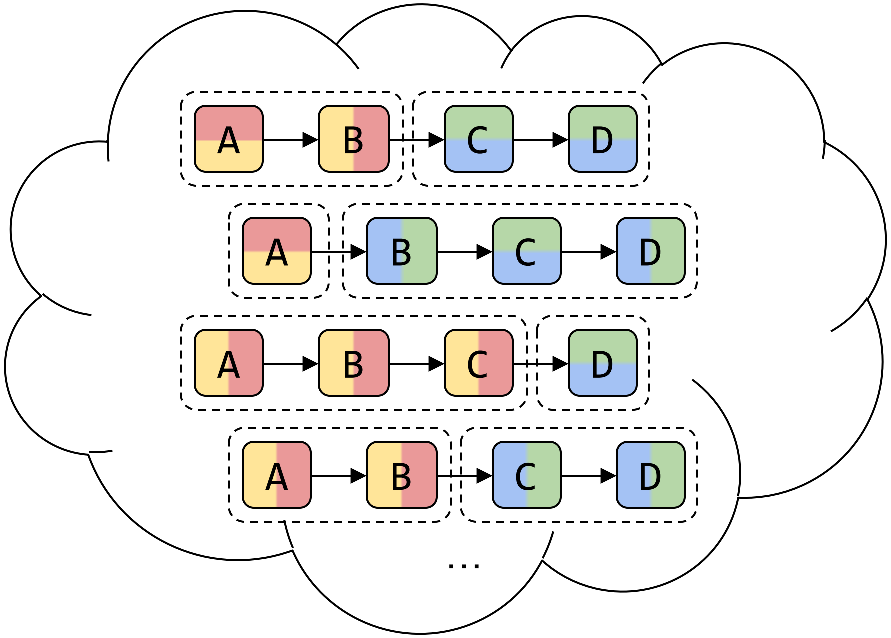


Search Space

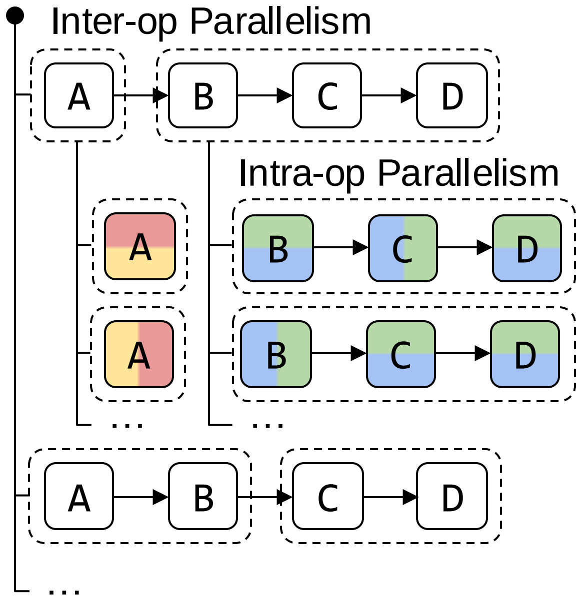
Computational Graph



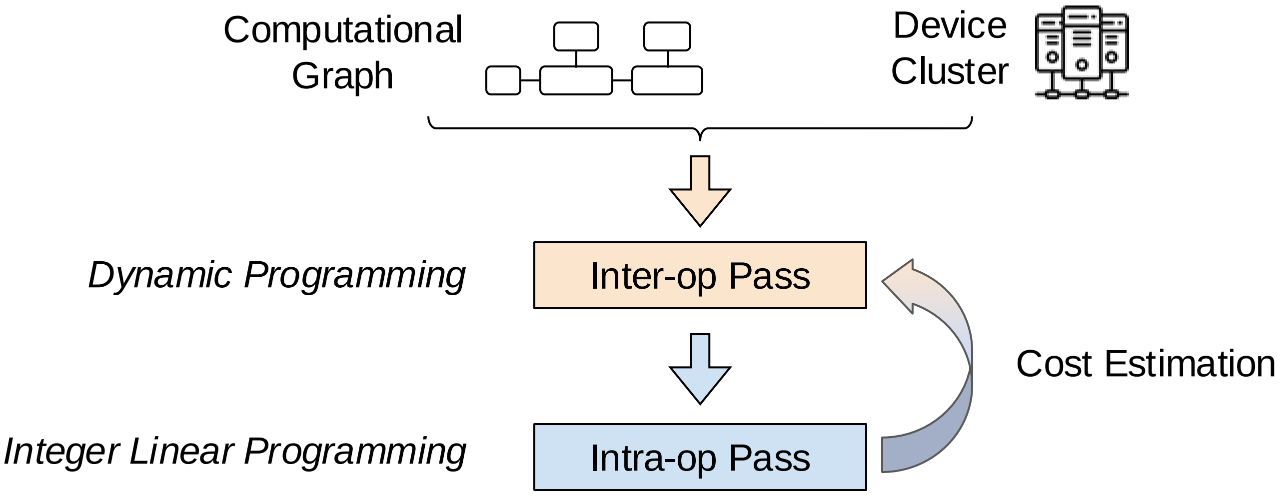
Whole Search Space



Alpa Hierarchical Space

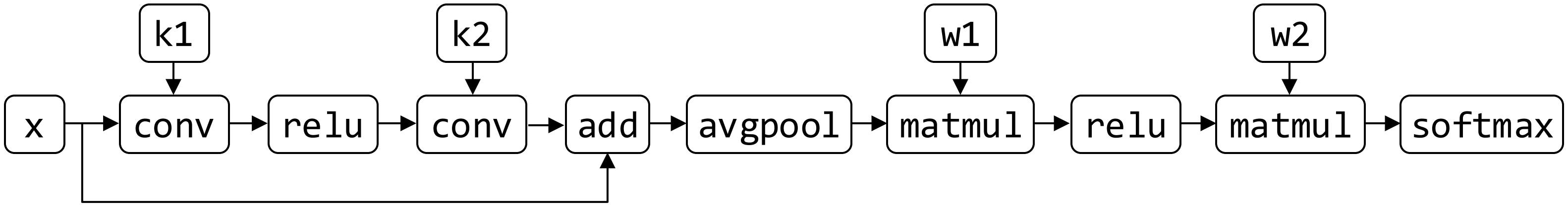


Alpa Compiler: Hierarchical Optimization

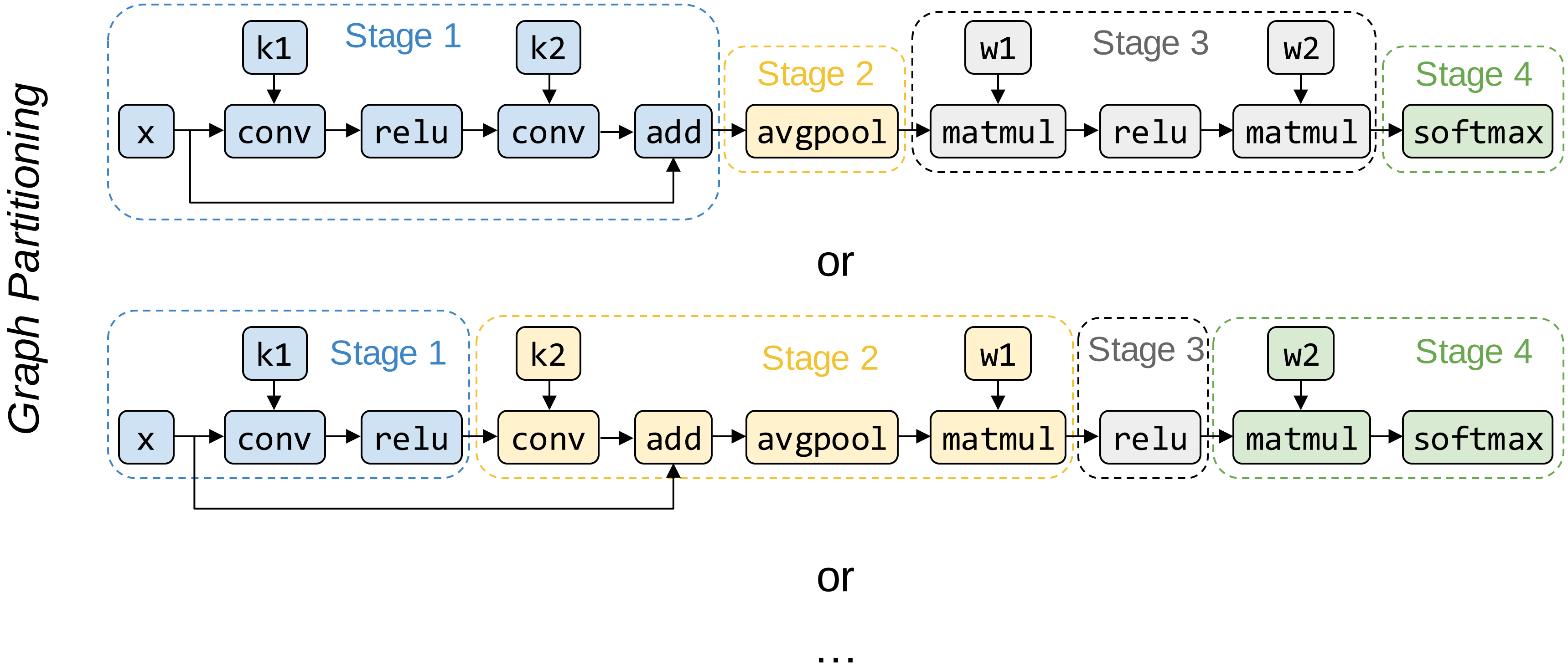


Inter-op Pass

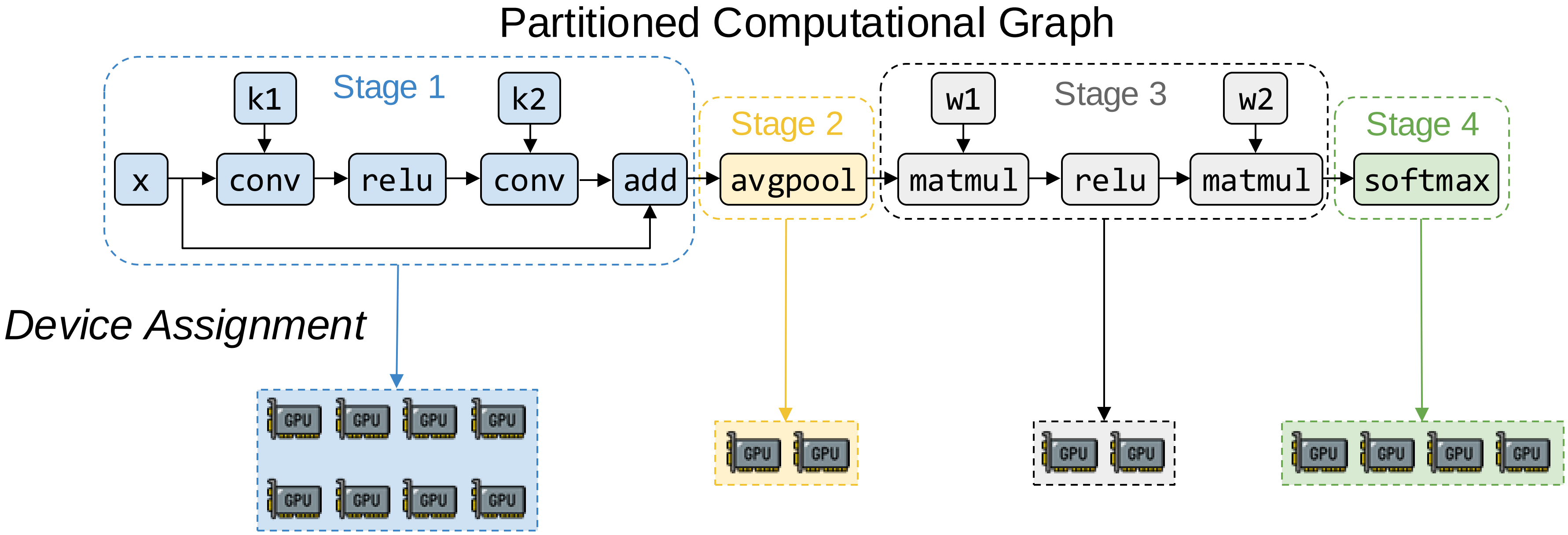
Computational Graph



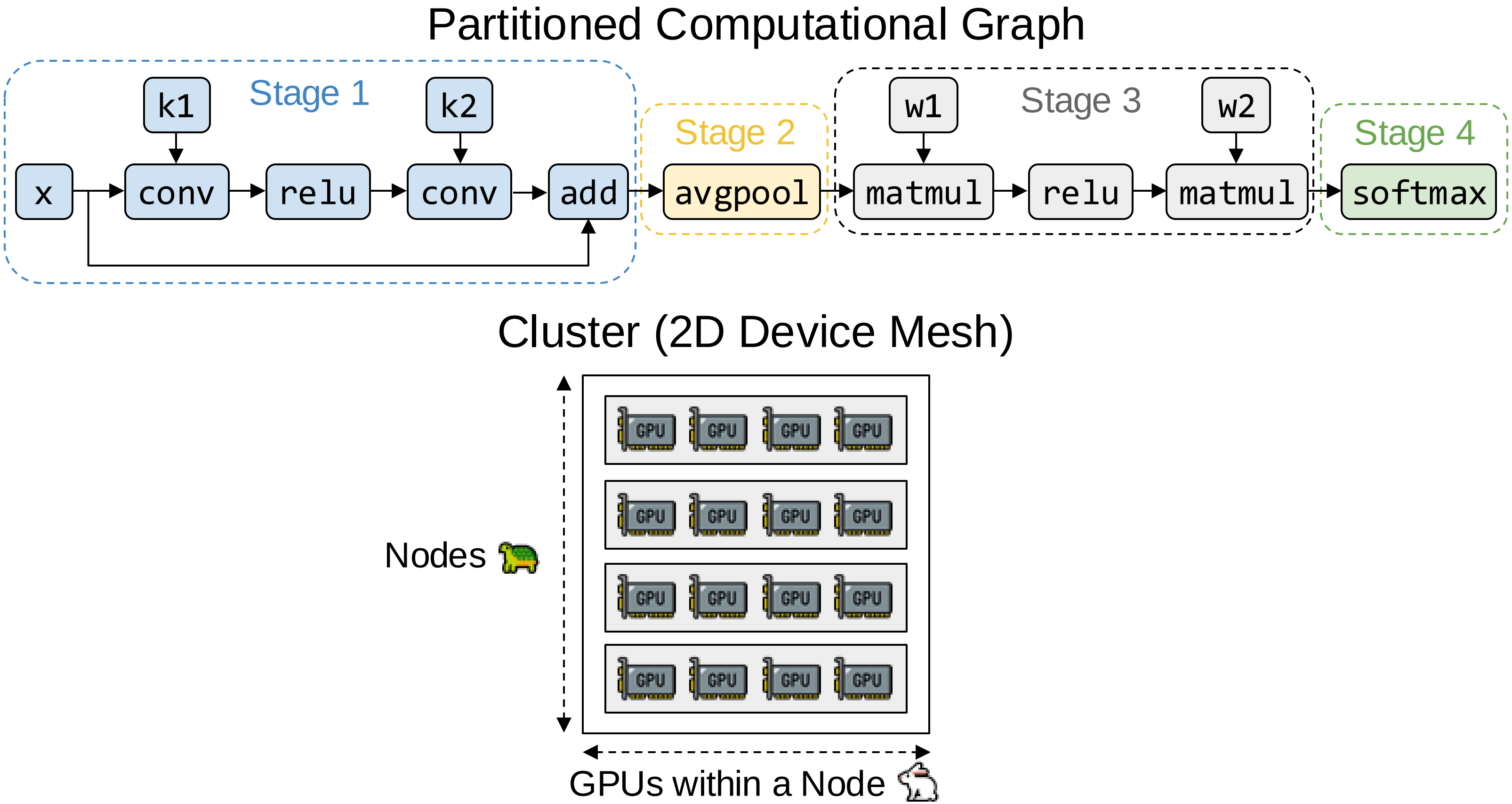
Inter-op Pass



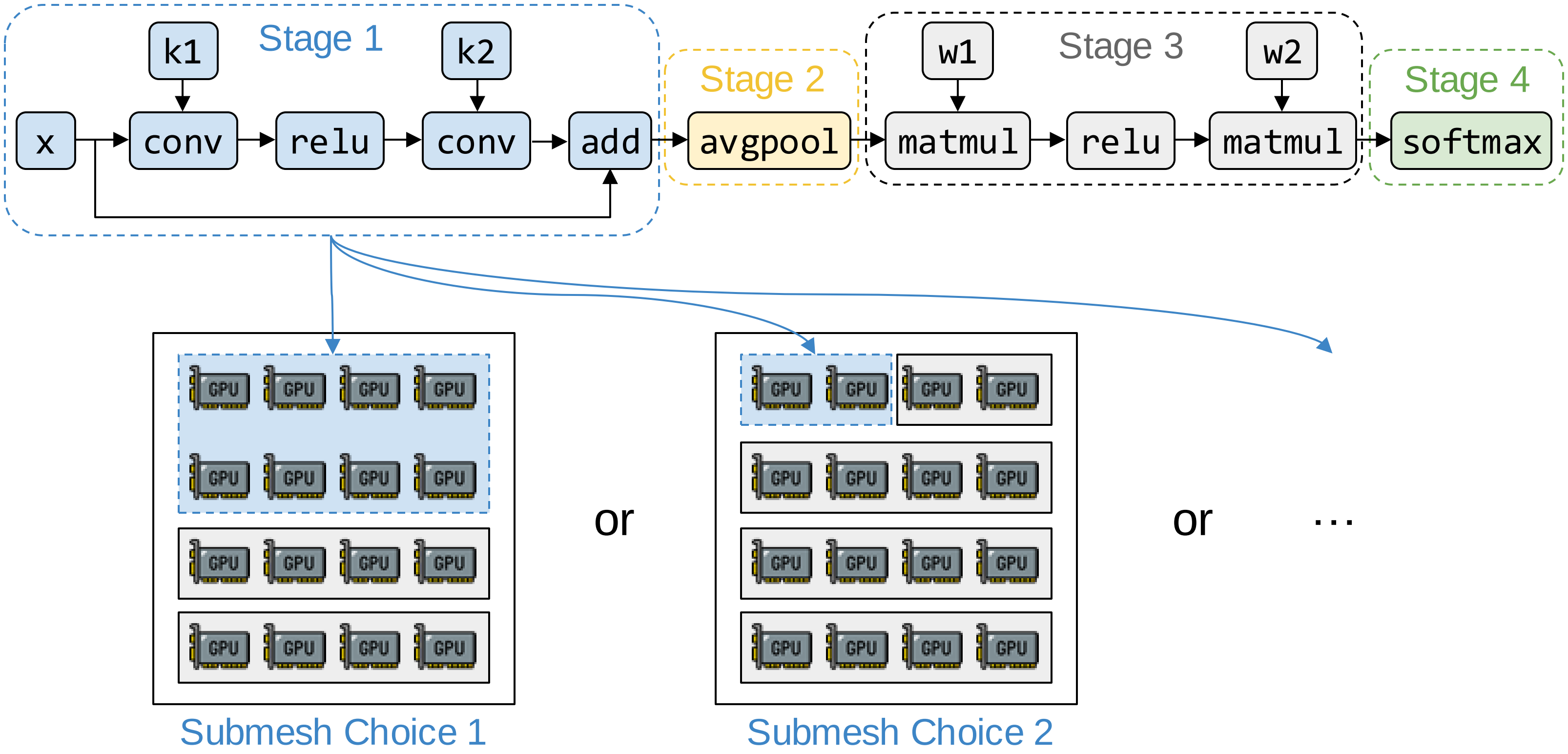
Inter-op Pass



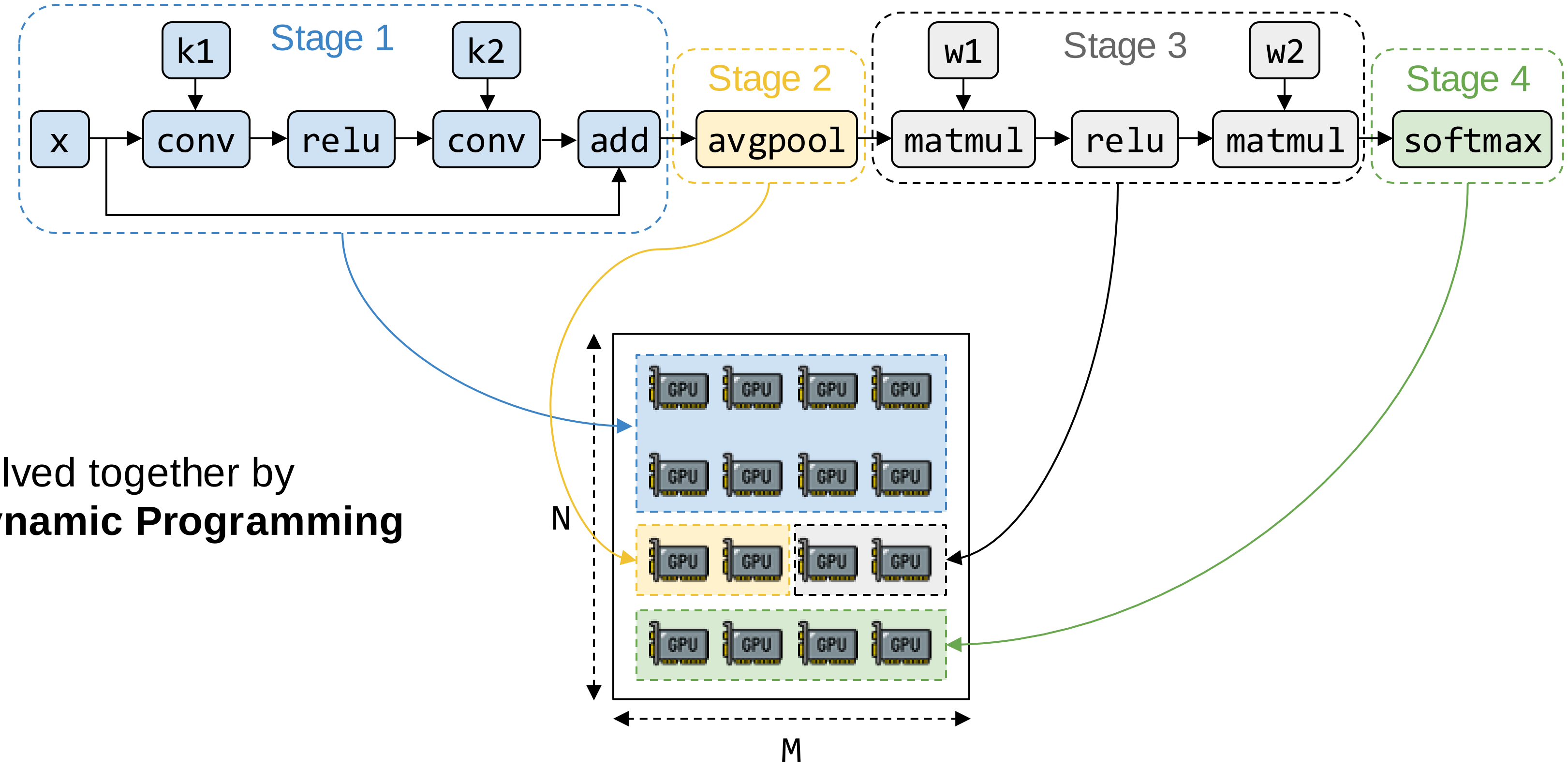
Inter-op Pass



Inter-op Pass



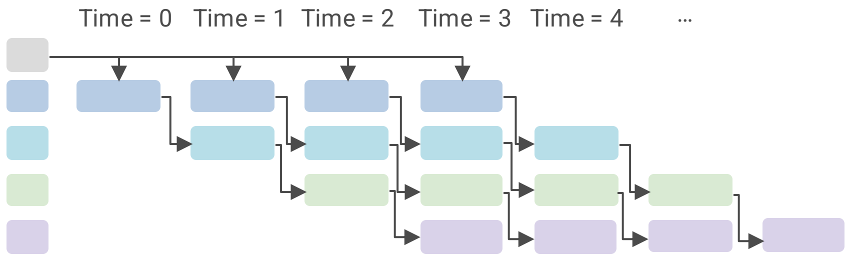
Inter-op Pass

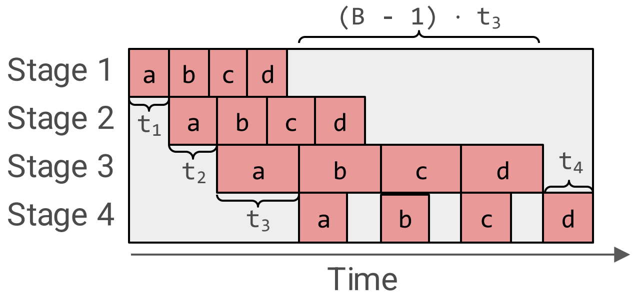


*S*olved together by
Dynamic Programming

Pipeline Execution Latency

Inter-op Pass





warmup phase stable phase

$$T = \sum_{i}^{S} t_i + (B-1) \cdot \max_{1 \le j \le S} \{t_j\}$$

Inter-op Pass: Dynamic Programming

Optimization objective: Find the optimal $(\text{stage}, \text{mesh})$ pairs that minimize T.

$$T = \underbrace{\sum_{i}^{S} t_i}_{\text{warmup phase}} + \underbrace{(B-1) \cdot \max_{1 \le j \le S}\{t_j\}}_{\text{stable phase}}$$

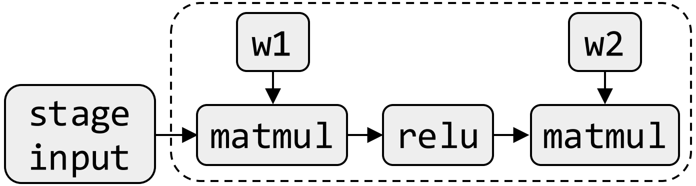
the **optimal** latency of executing stage i on its assigned mesh i :

$$t_i = t^*_{\text{intra}}(\text{stage}_i, \text{mesh}_i)$$

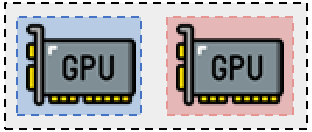
Solution:

Enumerate all possible $\max_{1 \le j \le S}\{t_j\}$ (stable phase) and convert the first term $\sum_i^S t_i$ (warmup phase) into a *2-dimensional knapsack problem*.

Intra-op Pass

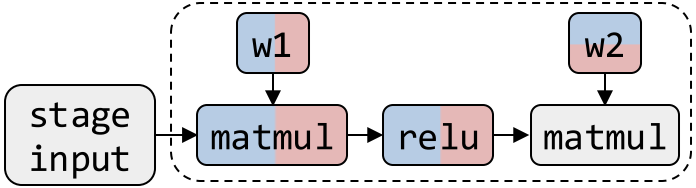


Stage



Submesh

Solved by
Integer Linear Programming

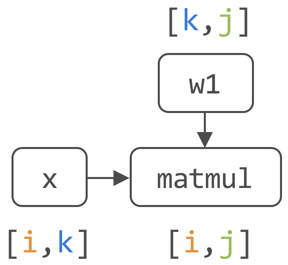


Stage with intra-operator parallelization

Intra-op Pass: Computation

Device 1
Device 2

Row-partitioned | Column-partitioned | Replicated



matmul[i, j] = Σ_k x[i, k] × w1[k, j]   Cost

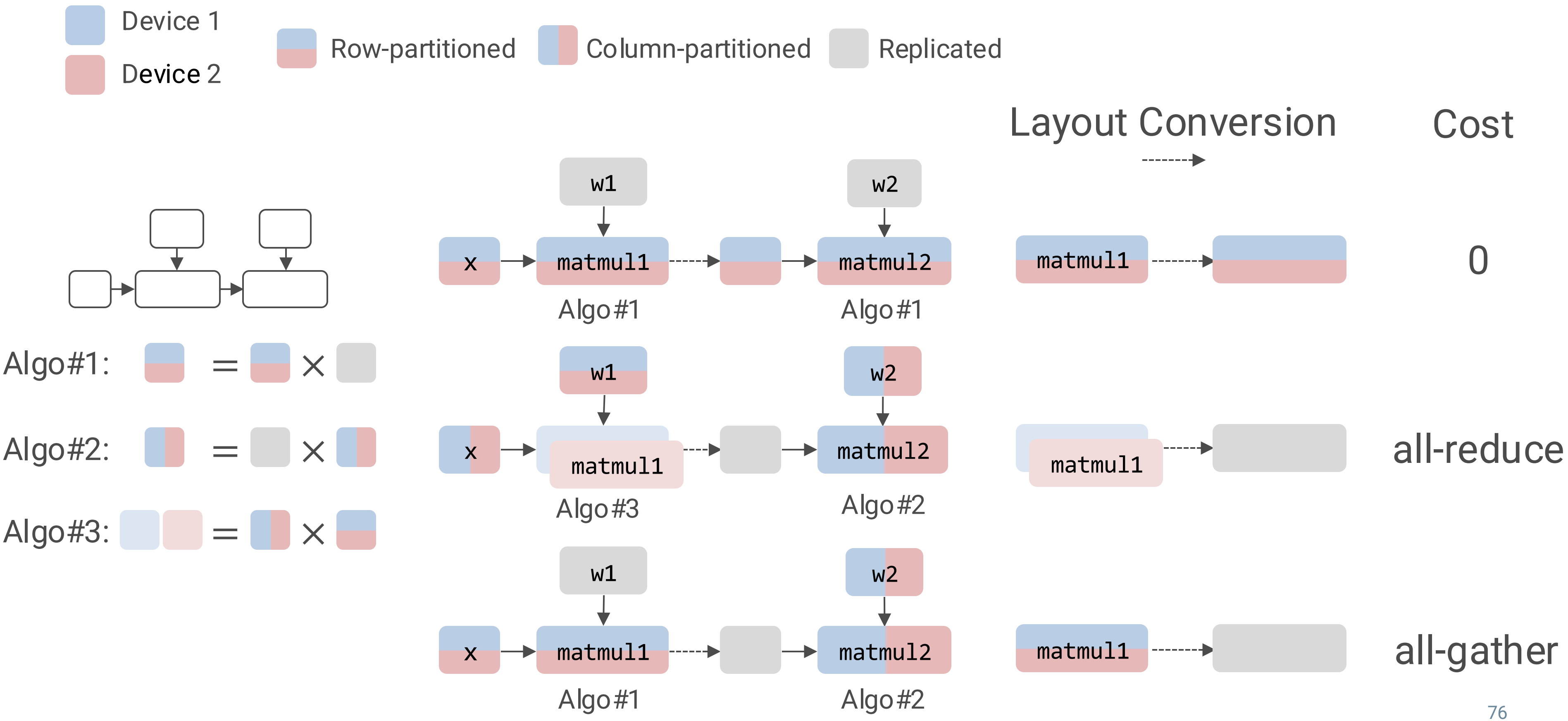
Algo#1: loop i   =   ×   Cost1

Algo#2: loop j   =   ×   Cost2

Algo#3: loop k   =   ×   Cost3

Algo#4: …

Intra-op Pass: Communication

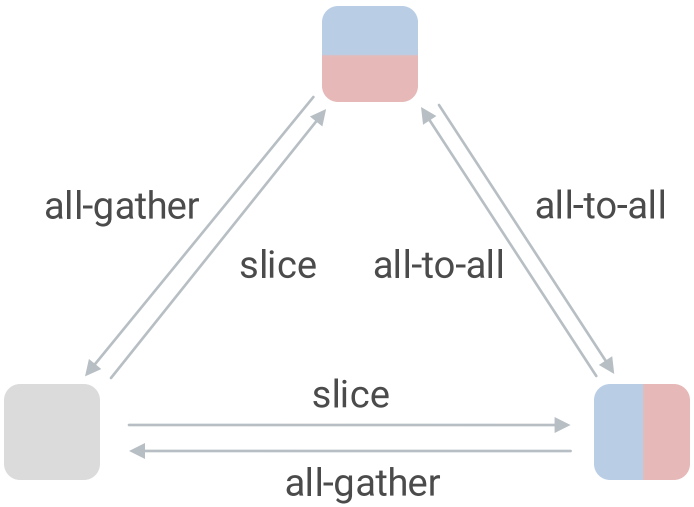


Intra-op Pass: Layout Conversion

Intra-op Pass

Device 1
Device 2

Row-partitioned Column-partitioned Replicated

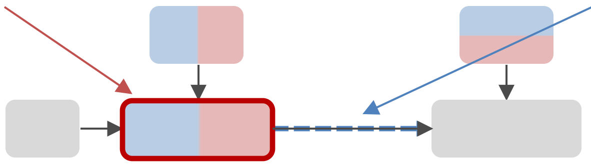


Intra-op Pass: ILP Formulation

Goal: Within each stage, “color” every node in the stage, so the execution latency of this stage on its assigned mesh is minimized.

For every node (op), enumerate all possible parallel algorithms

For every edge, infer the cost due to layout conversion



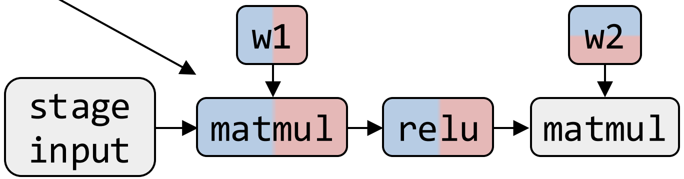
Minimize node-cost + edge-cost

s.t. peak memory usage < memory budget

Intra-op Pass

Integer Linear Programming Formulation

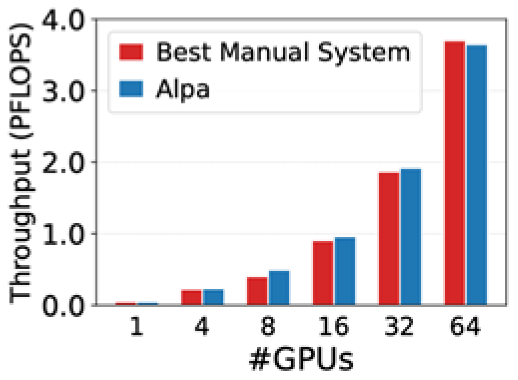
Decision vector
Parallel strategies of each operator



Minimize Computation cost + Communication cost

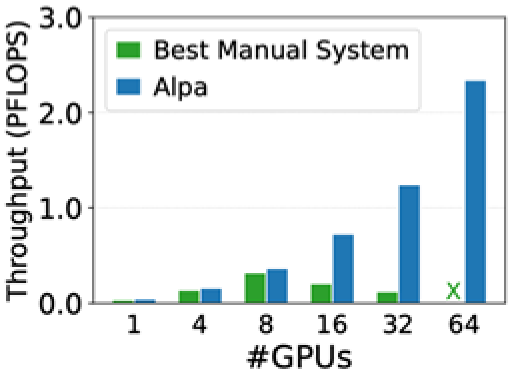
Evaluation: Comparing with Previous Works

GPT (up to 39B)



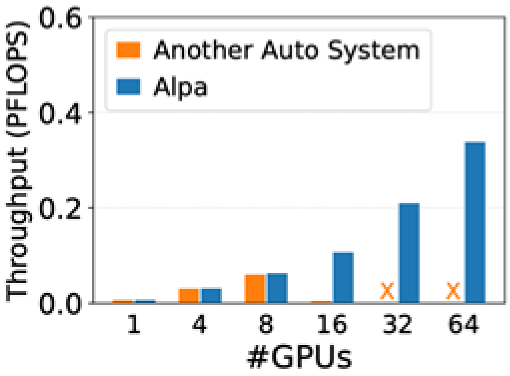
Match specialized manual systems.

GShard MoE (up to 70B)



Outperform the manual baseline by up to 8x.

Wide-ResNet (up to 13B)



Generalize to models without manual plans.

Weak scaling results where the model size grow with #GPUs.
Evaluated on 8 AWS EC2 p3.16xlarge nodes with 8 16GB V100s each (64 GPUs in total).

Automatic Parallelization Methods

Search-based methods

- ✅ Easy to extend the search space
- ✅ No training cost
- ❌ High inference cost
- ❌ Not explainable
- ❌ No optimality guarantee

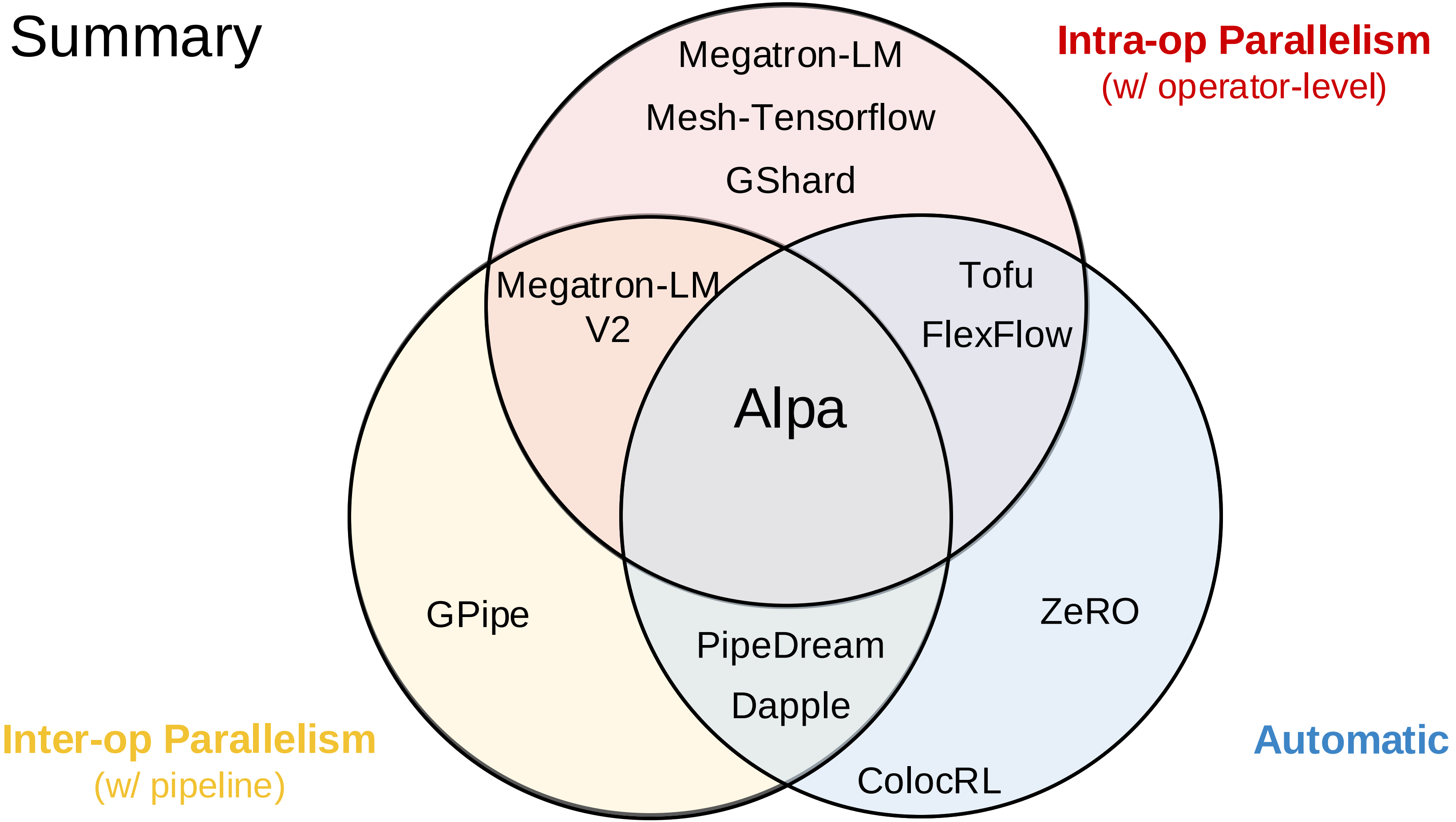
Learning-based methods

- ✅ Easy to extend the search space
- ❌ High training cost
- ✅ Low inference cost
- ❌ Not explainable
- ❌ No optimality guarantee

Optimization-based methods

- ❌ Non-trivial to extend the search space
- ✅ No training cost
- ✅ Medium inference cost
- ✅ Explainable
- ✅ Some optimality guarantee

Summary



Intra-op Parallelism
(w/ operator-level)
Megatron-LM
Mesh-Tensorflow
GShard
Megatron-LM V2
Tofu
FlexFlow
Alpa
GPipe
PipeDream
Dapple
ZeRO
ColocRL
Inter-op Parallelism
(w/ pipeline)
Automatic

Summary: How to Choose Parallelism

1. Use automatic compiler if not transformer
2. Manual parallelism search for transformers:

- Factors to consider
  - #GPUs you have
  - Model size
  - JCT (Job completion time)
  - Communication bandwidth
  - etc.

Hao's Ultimate Guide