CS6216 Advanced Topics in Machine Learning (Systems)

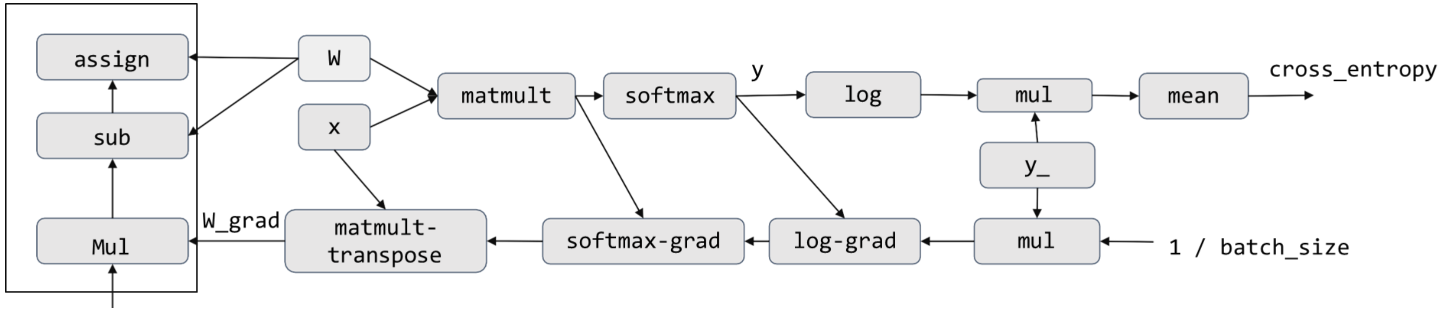
Parallelism and Training Techniques

Yao LU
11 Sep 2024

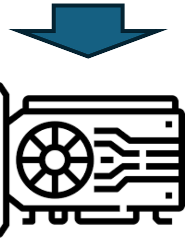
National University of Singapore
School of Computing

Mapping compute graph to actual runtime

- Key factors to consider:
  - Graph dependency
  - Parallelism & batching
  - Driver & API



- CPU, GPU, TPU, FPGA, etc.
  - Each architecture has corresponding libraries and APIs



- Optimizations:
  - Operator code-gen and fusion
  - Graph-level optimizations

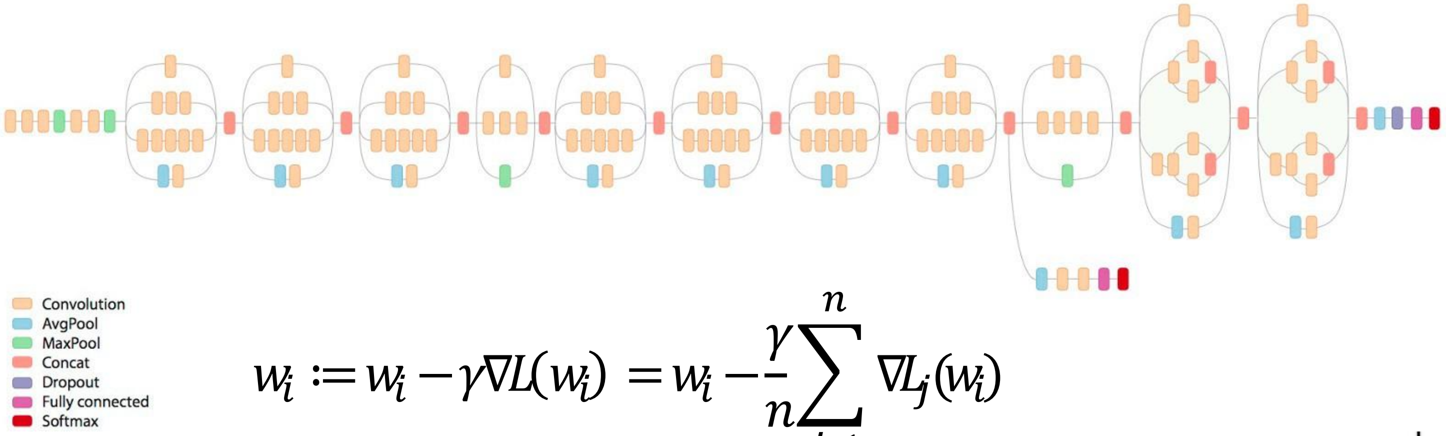
Algorithmic workflows: recap

Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

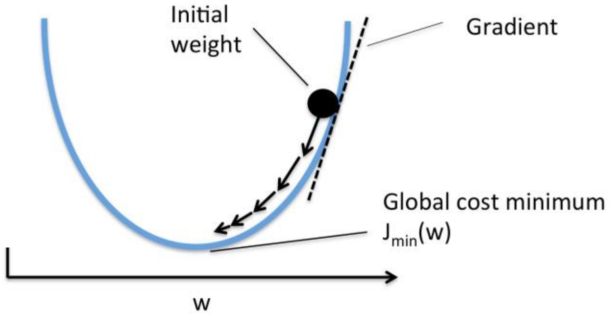
1. **Forward propagation:** apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights

Model inputs

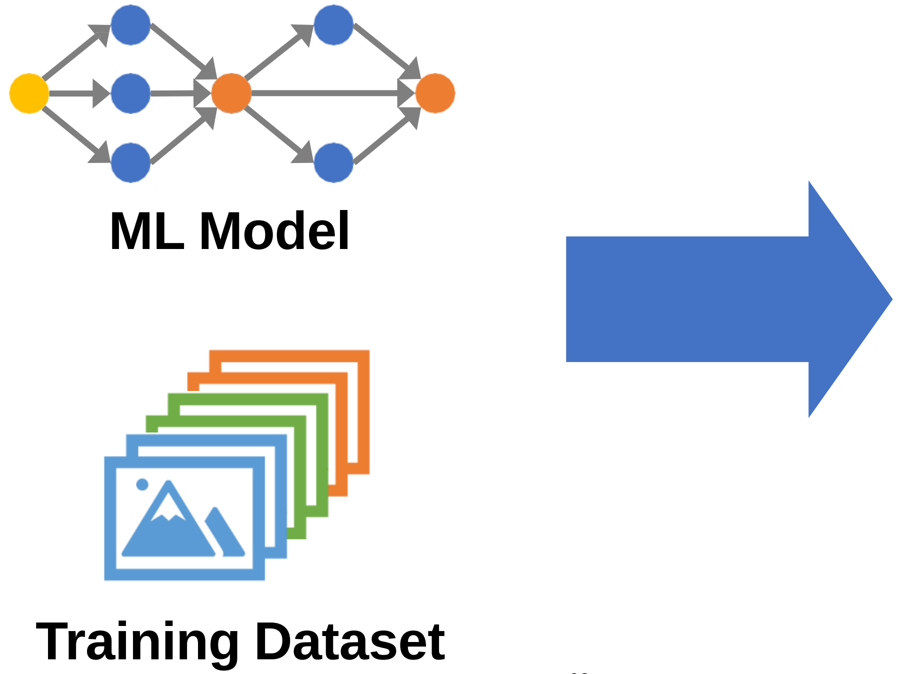


Model prediction

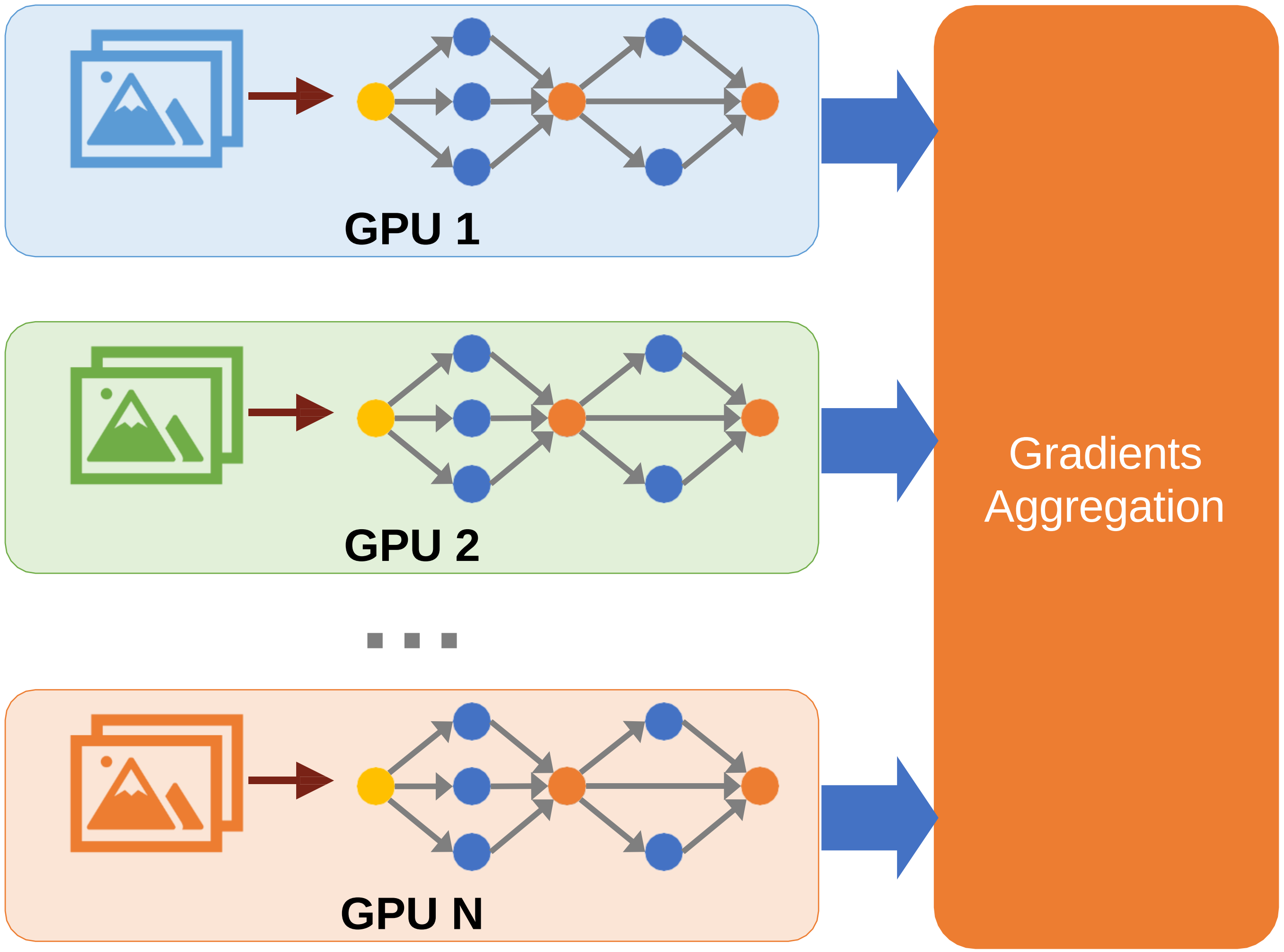
$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n}\sum_{j=1}^{n} \nabla L_j(w_i)$$



Execution of the compute graph: data parallelism



$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

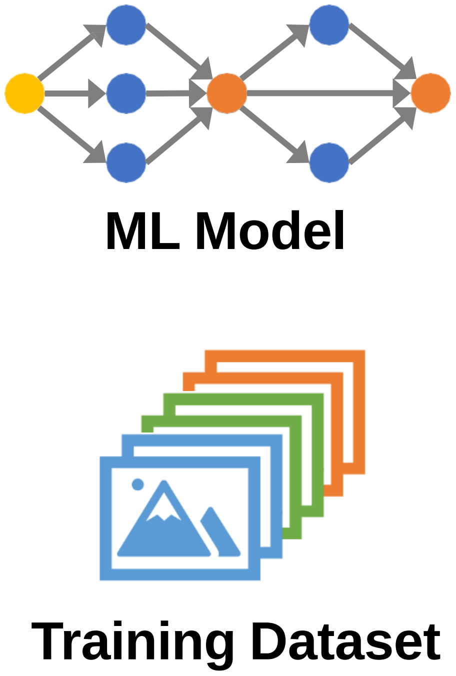


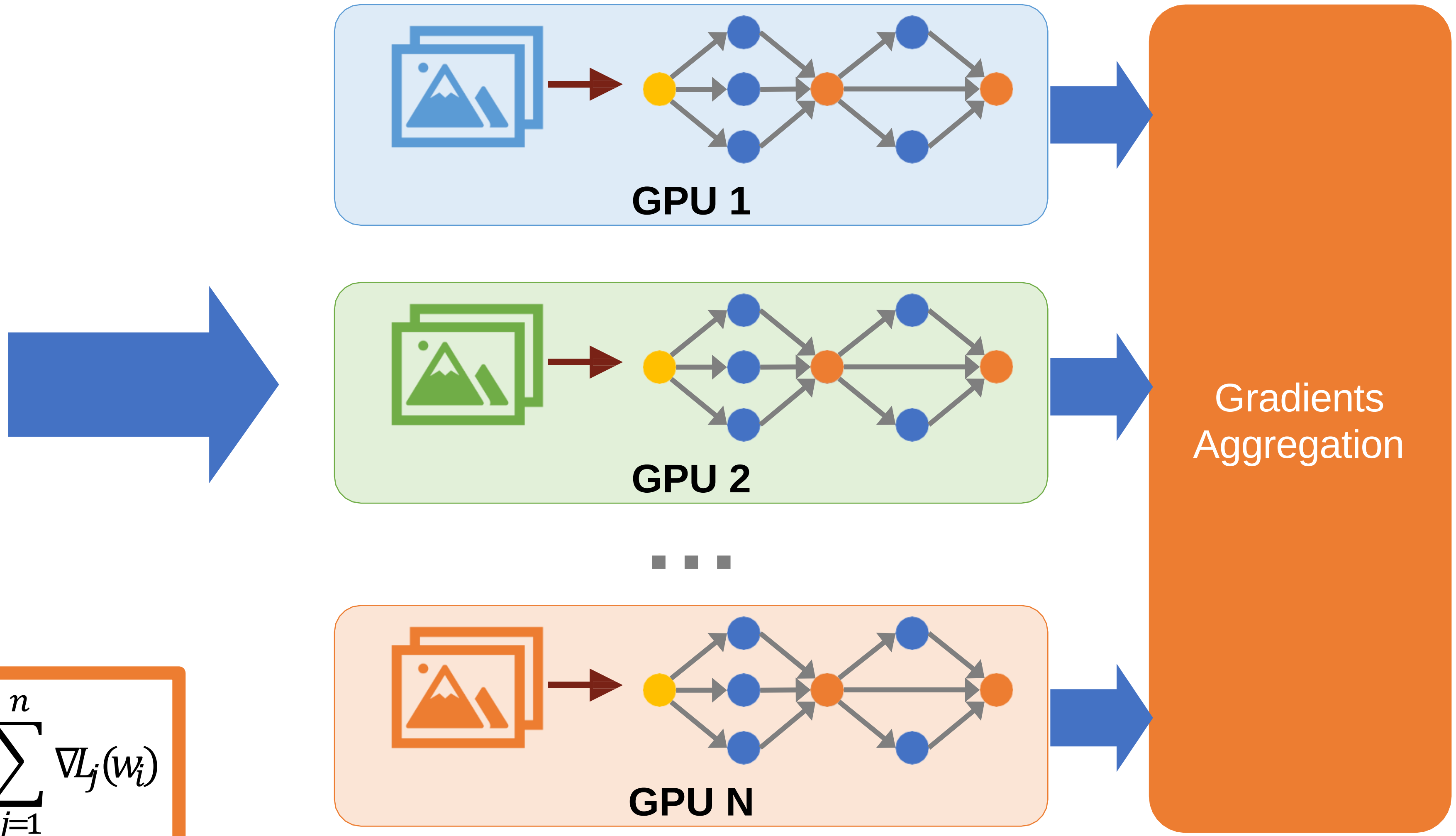
1. Partition training data into batches

2. Compute the gradients of each batch on a GPU

3. Aggregate gradients across GPUs

Execution of the compute graph: data parallelism





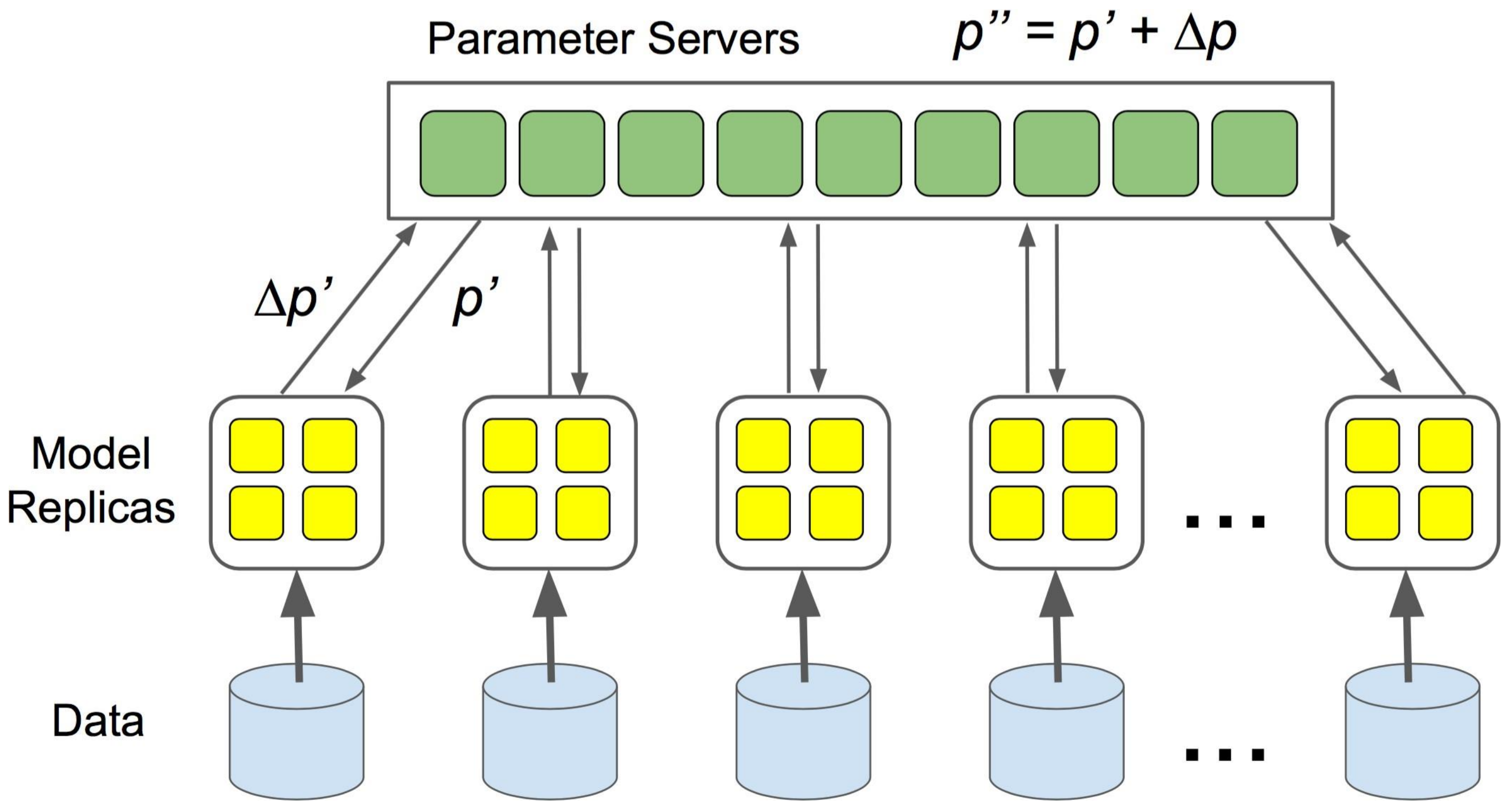
$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \boxed{\sum_{j=1}^{n} \nabla L_j(w_i)}$$

1. Partition training data into batches

2. Compute the gradients of each batch on a GPU

3. Aggregate gradients across GPUs

Data parallelism: Parameter Server (OSDI14)

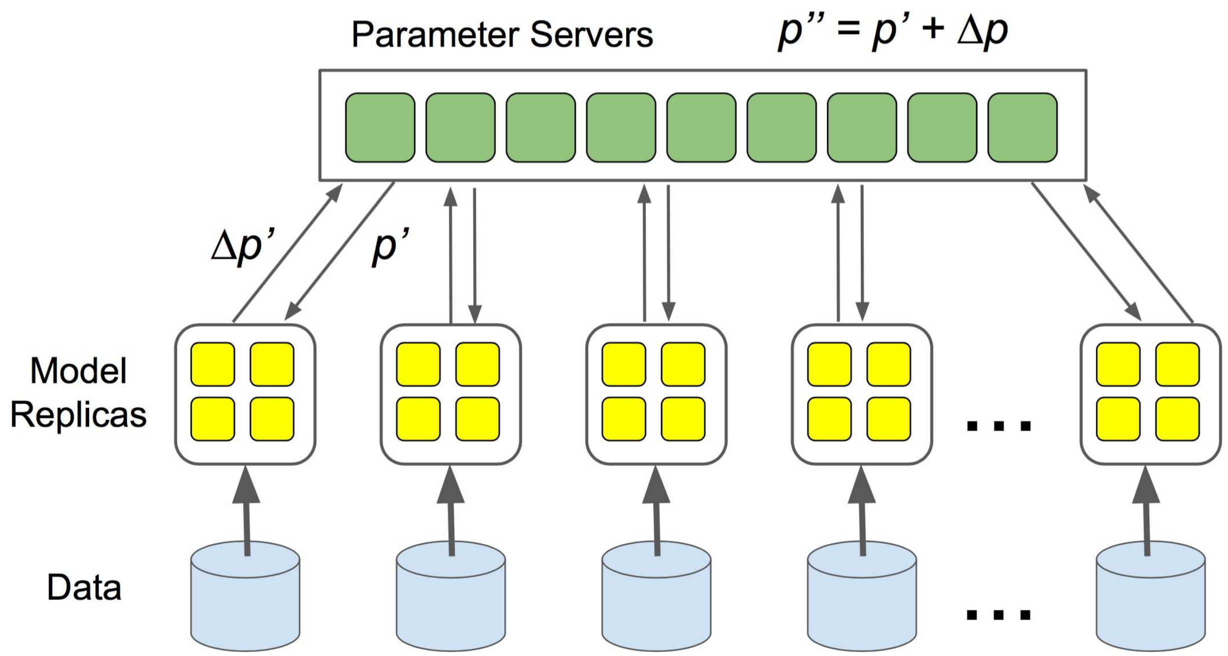


Workers push gradients to parameter servers and pull updated parameters back

Data parallelism: Parameter Server (OSDI14)

- **Centralized communication**: all workers communicate with parameter servers for weights update; cannot scale to large numbers of workers

- Can we decentralize communication in DNN training?



Data parallelism: Parameter Server (OSDI14)

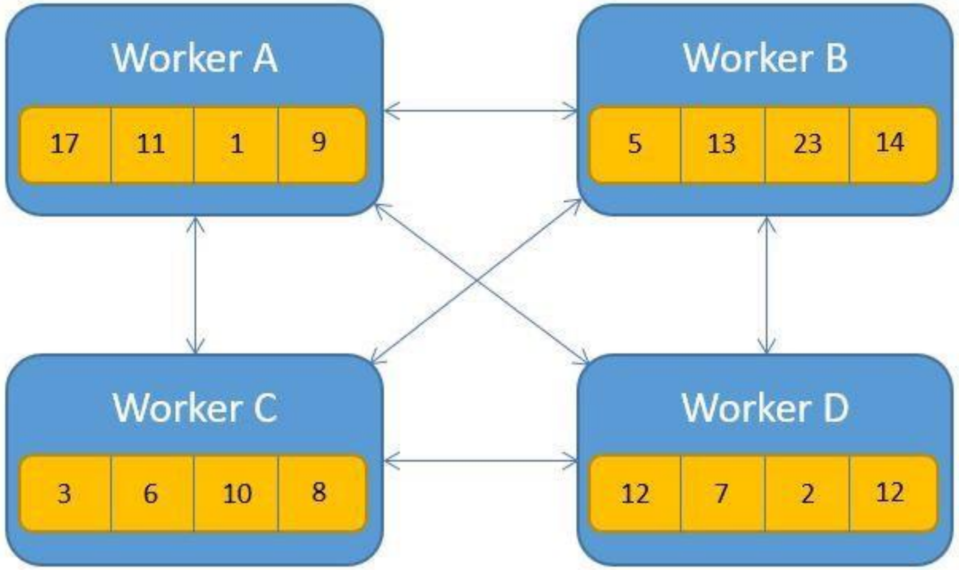
- **Centralized communication**: all workers communicate with parameter servers for weights update; cannot scale to large numbers of workers
- Can we decentralize communication in DNN training?
- **AllReduce**: perform element-wise reduction across multiple devices

Ways of AllReduce

- Naïve AllReduce
- Ring AllReduce
- Tree AllReduce
- Butterfly AllReduce

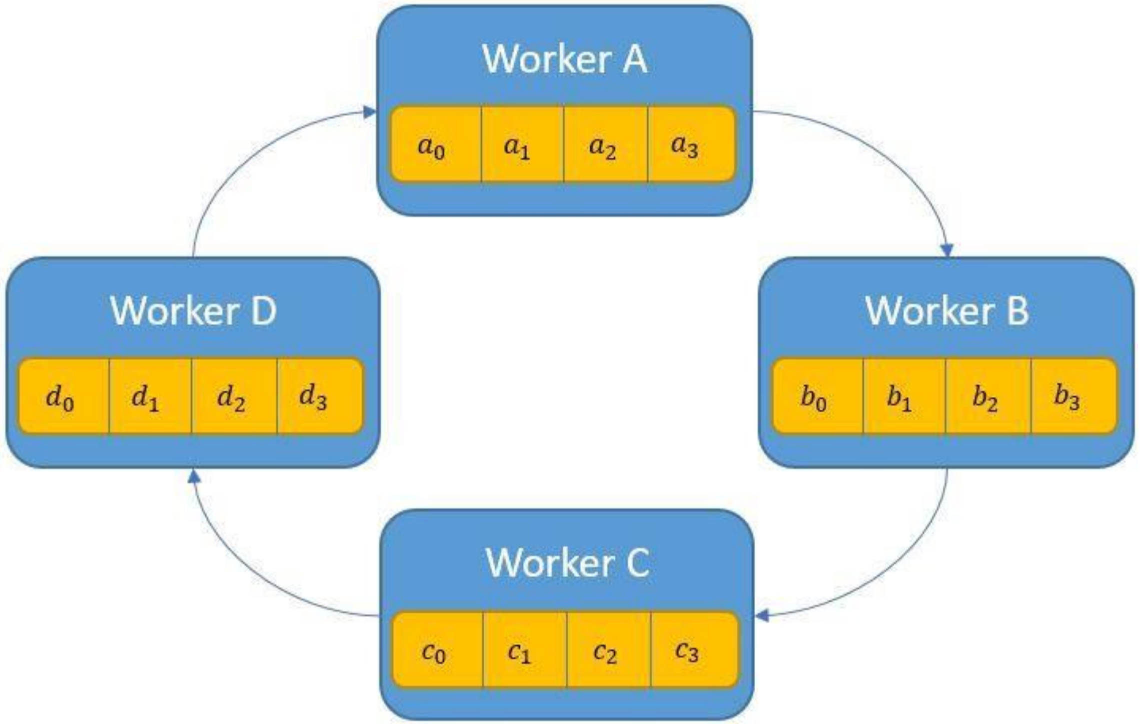
Naïve AllReduce

- Each worker can send its local gradients to all other workers
- If we have N workers and each worker contains M parameters
- Overall communication: N * (N-1) * M parameters
- **Issue**: each worker communicates with all other workers; same scalability issue as parameter server



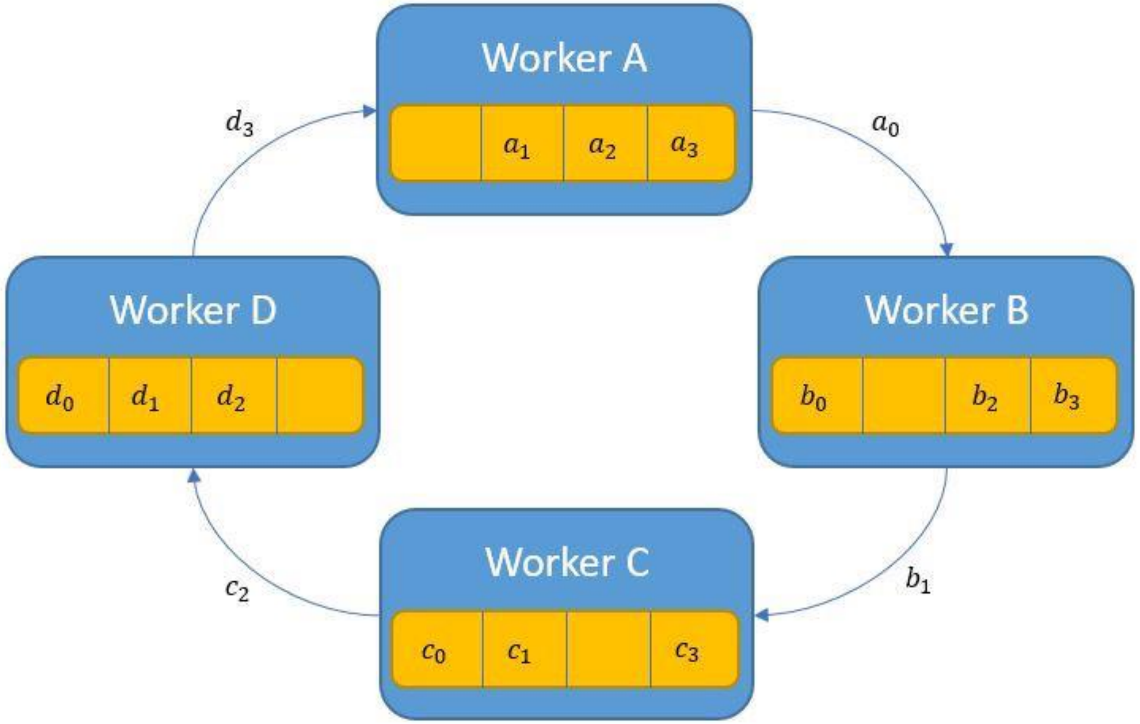
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times



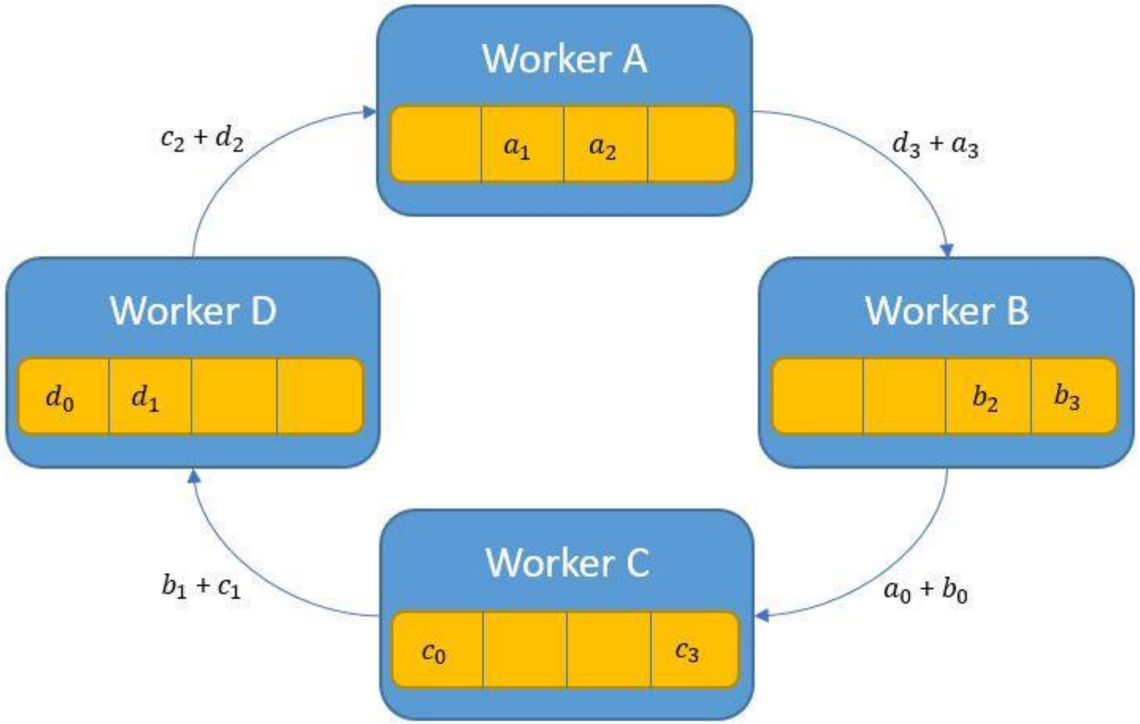
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times



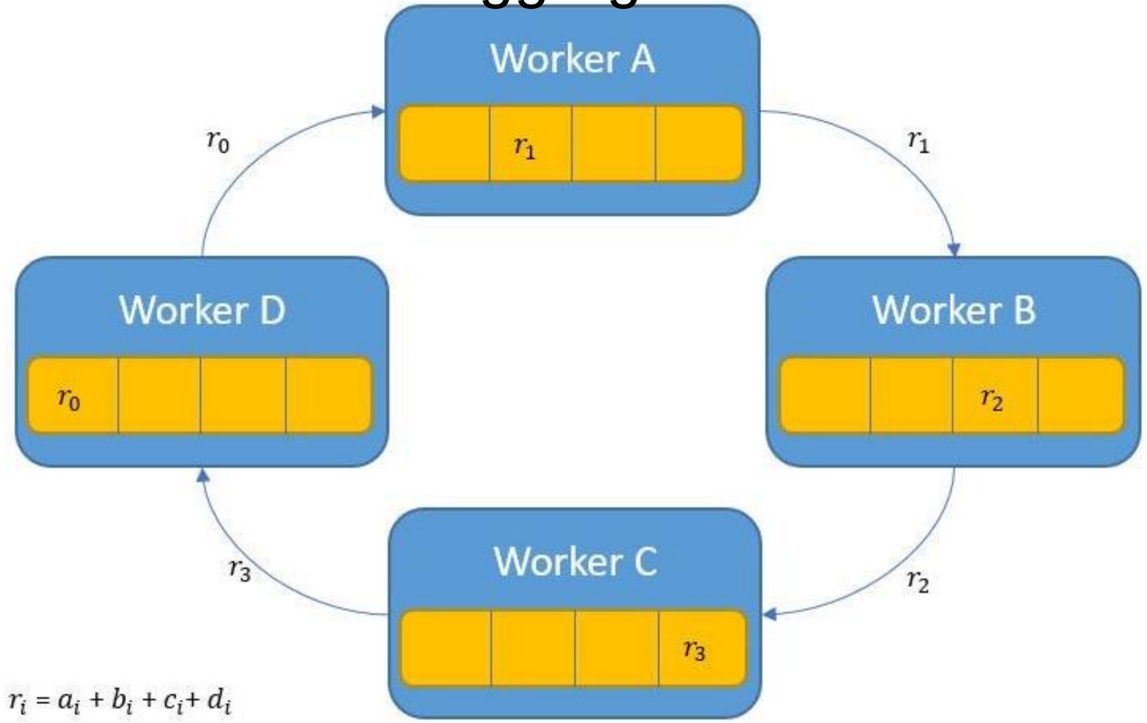
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times



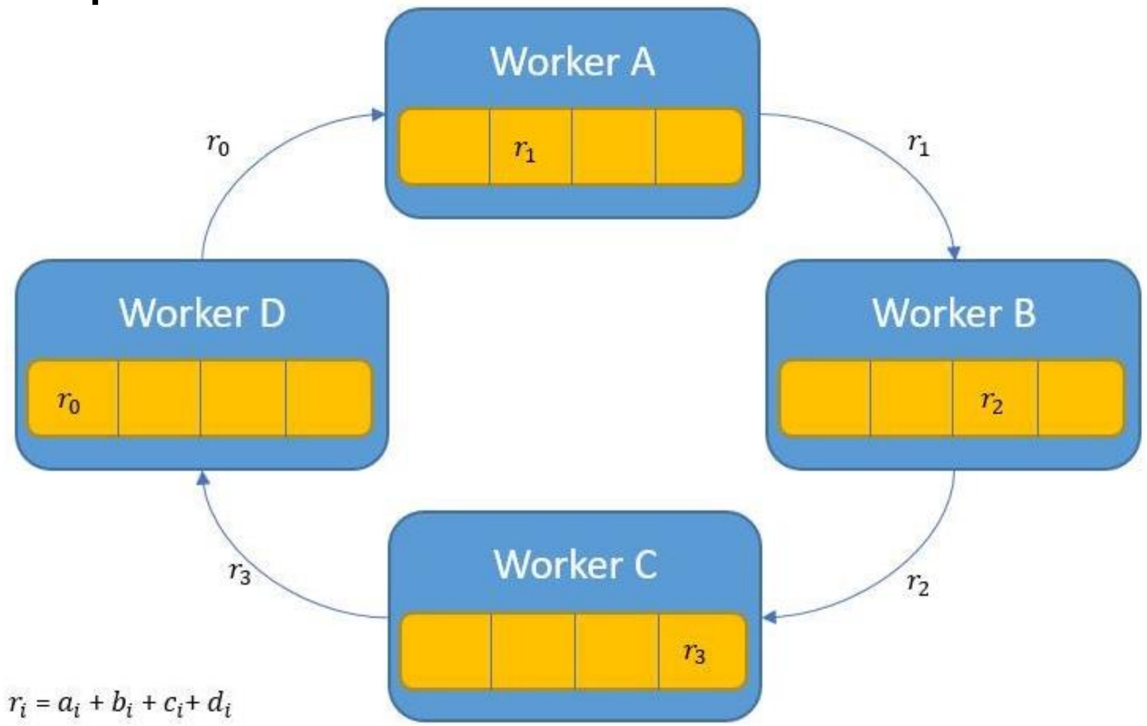
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- After step 1, each worker has the aggregated version of M/N parameters



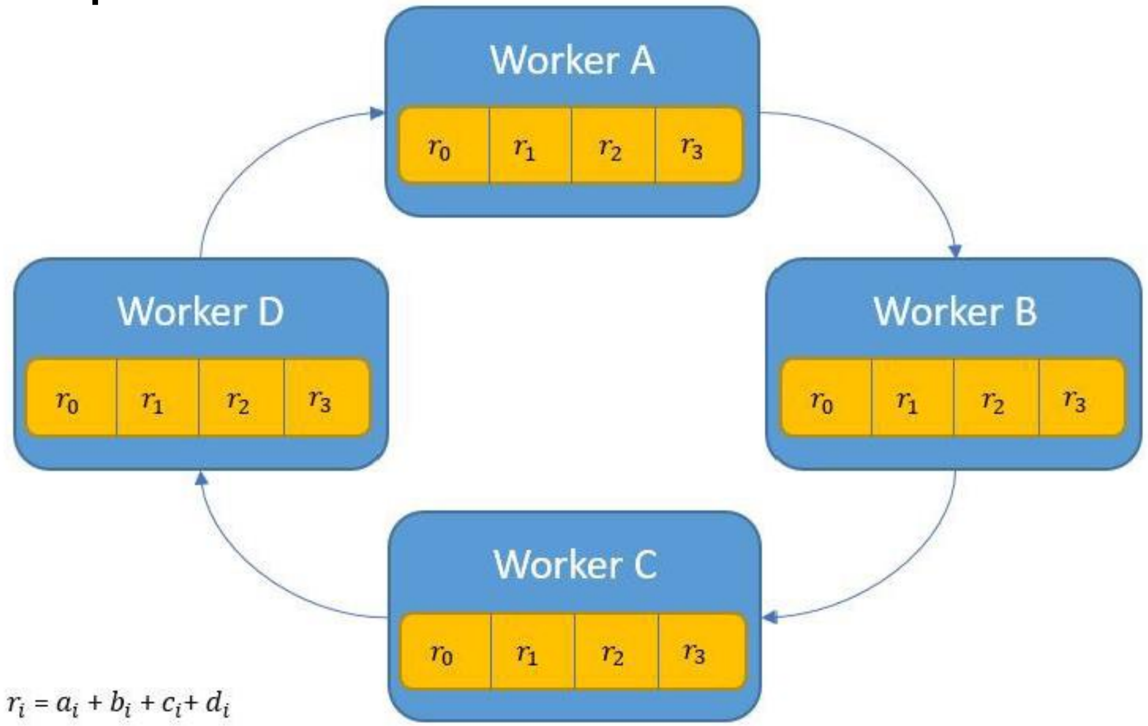
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times



Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times

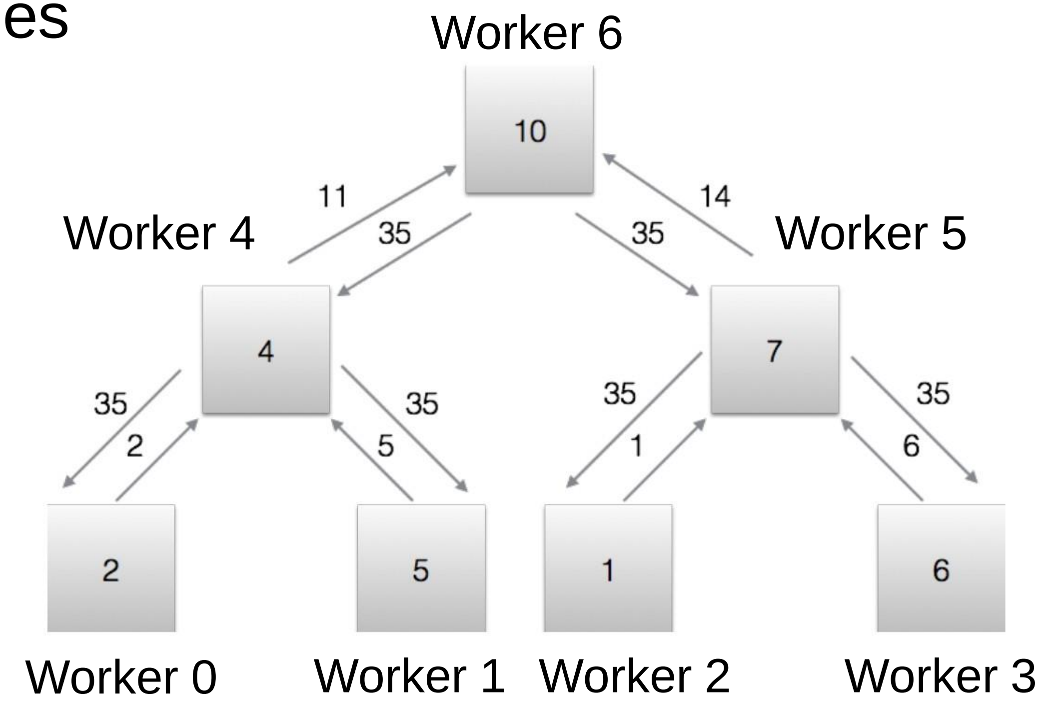


Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times
- Overall communication: 2 * M * N parameters
  - Aggregation: M * N parameters
  - Broadcast: M * N parameters

Tree AllReduce

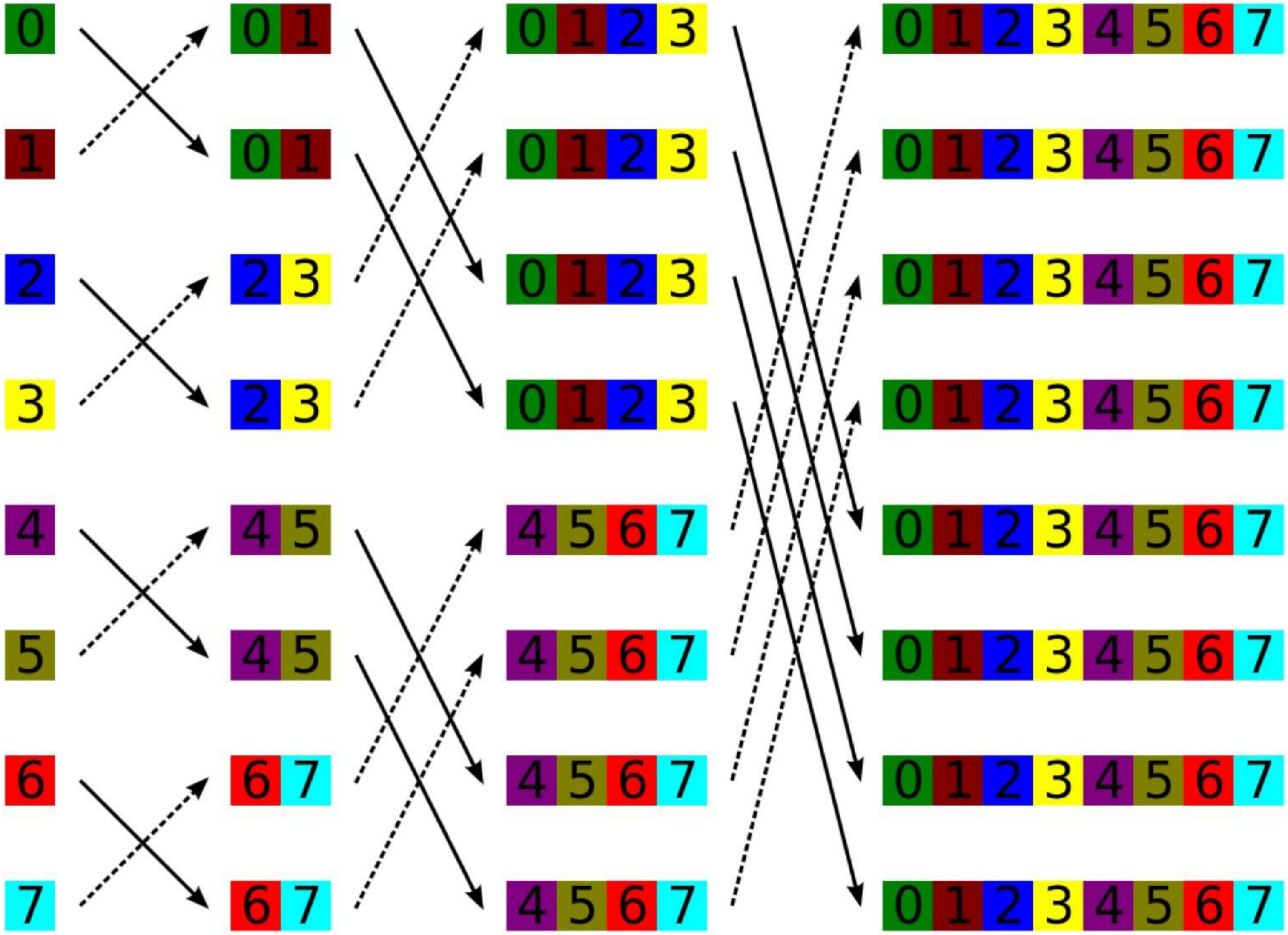
- Construct a tree of N workers;
- Step 1 (Aggregation): each worker sends M parameters to its parent; repeat log(N) times
- Step 2 (Broadcast): each worker sends M parameters to its children; repeat log(N) times



Tree AllReduce

- Construct a tree of N workers;
- Step 1 (Aggregation): each worker sends M parameters to its parent; repeat log(N) times
- Step 2 (Broadcast): each worker sends M parameters to its children; repeat log(N) times
- Overall communication: 2 * N * M parameters
  - Aggregation: M * N parameters
  - Broadcast: M * N parameters

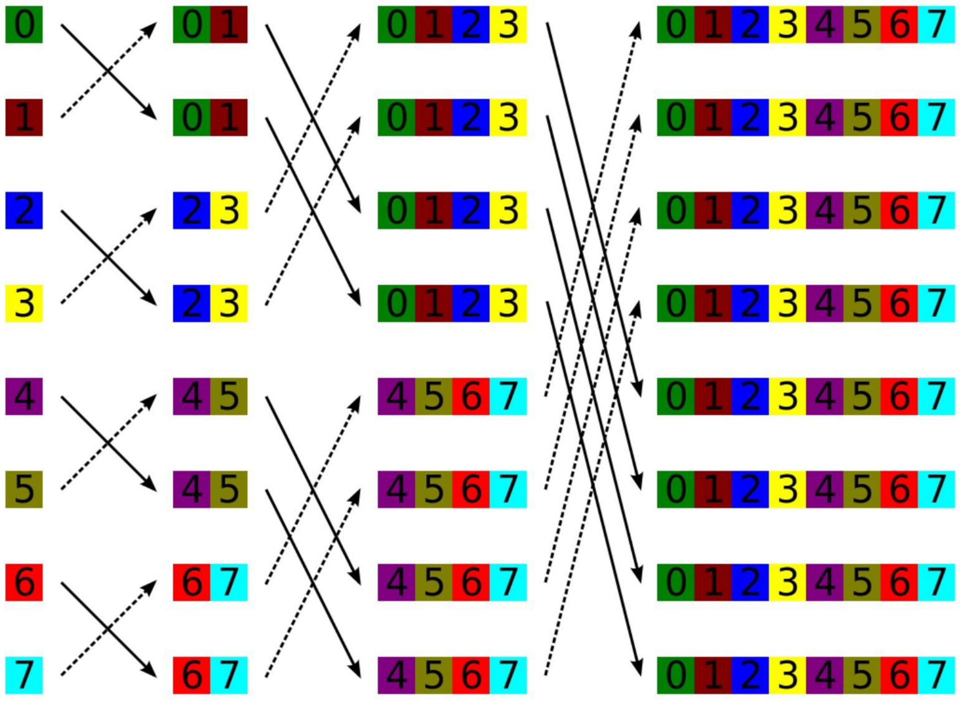
Butterfly AllReduce



Butterfly AllReduce

- Repeat log(N) times:
  1. Each worker sends M parameters to its target node in the butterfly network
  2. Each worker aggregates gradients locally

- Overall communication: N * M * log(N) parameters

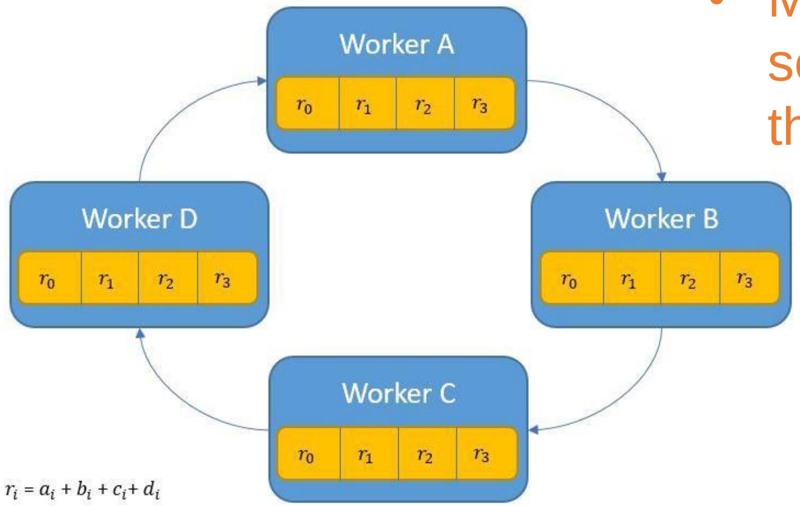


Comparing AllReduce methods

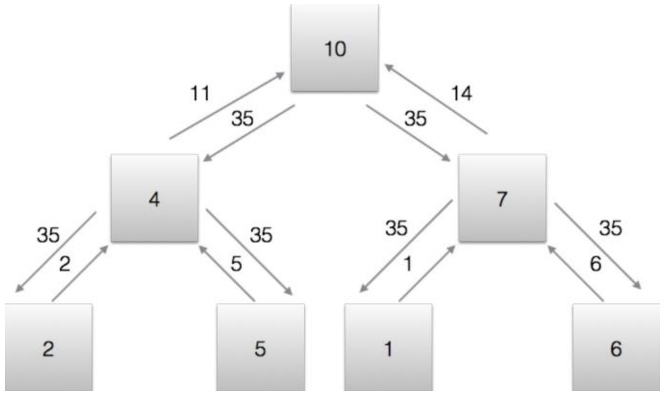
	Parameter Server	Naïve AllReduce	Ring AllReduce	Tree AllReduce	Butterfly AllReduce
Overall communication	$2 \times N \times M$	$N^2 \times M$	$2 \times N \times M$	$2 \times N \times M$	$N \times M \times \log N$

Ring AllReduce v.s. Tree AllReduce v.s. Parameter Server

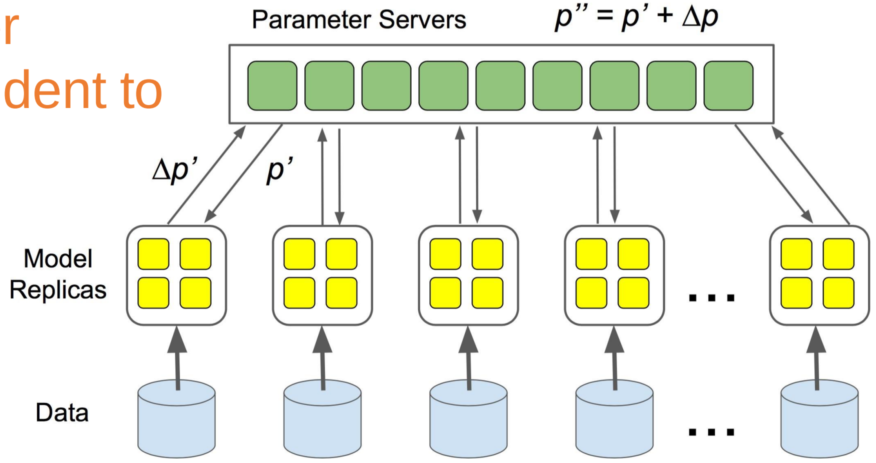
Ring AllReduce:
- Best latency
- Balanced workload across workers
- More scalable since each worker sends 2*M parameters (independent to the number of workers)



Each worker sends **M/N** parameters per iteration; repeat for **2*N** iterations
Latency: M/N * (2*N) / bandwidth



Each worker sends **M** parameters per iteration; repeat for **2*log(N)** iterations
Latency: M * 2 * log(N) / bandwidth



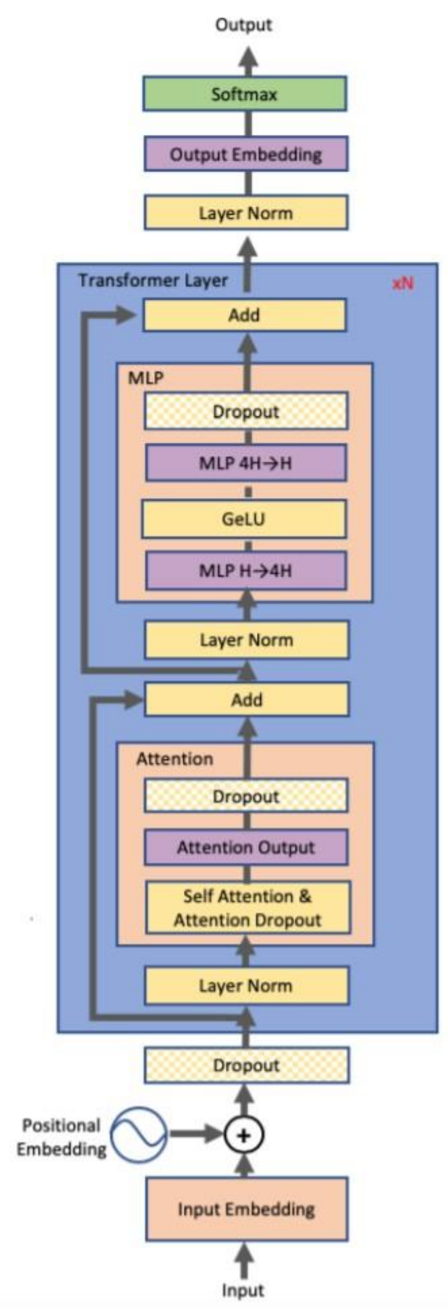
All workers send **M** parameters to parameter servers and receive **M** parameters from servers
Latency: M * N / bandwidth

Large model training challenges

	Bert-Large	GPT-2	Turing 17.2 NLG	GPT-3
Parameters	0.32B	1.5B	17.2B	175B
Layers	24	48	78	96
Hidden Dimension	1024	1600	4256	12288
Relative Computation	1x	4.7x	**54x**	**547x**
Memory Footprint	5.12GB	24GB	**275GB**	**2800GB**

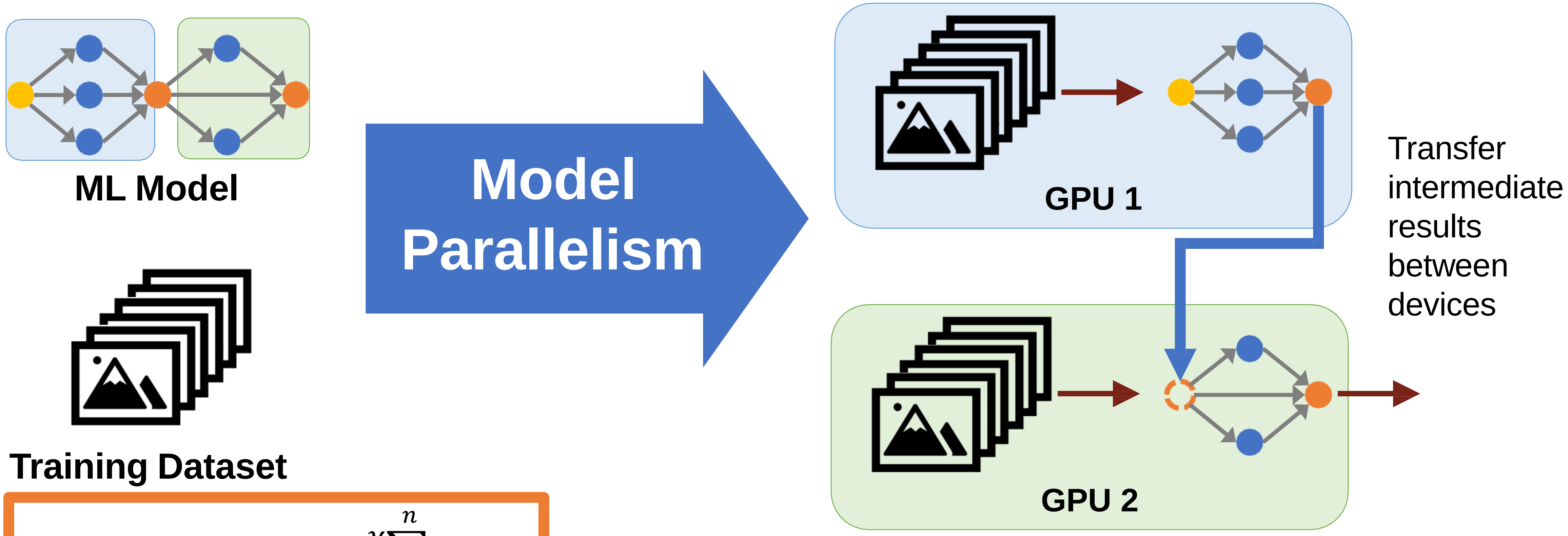
NVIDIA V100 GPU memory capacity: 16G/32G
NVIDIA A100 GPU memory capacity: 40G/80G

Out of Memory



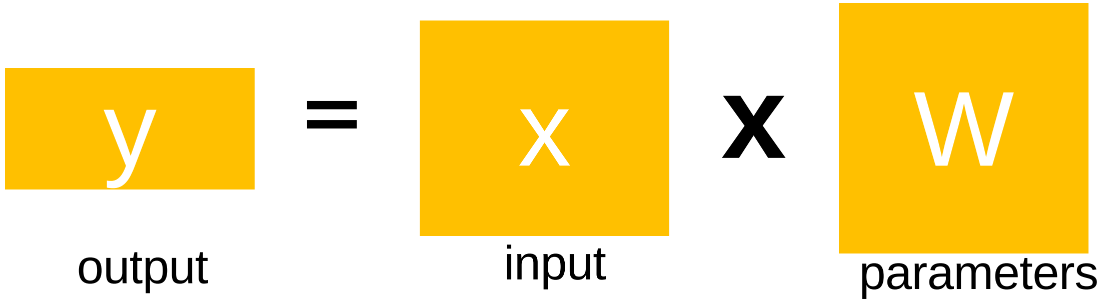
Execution of the compute graph: tensor / model parallelism

- Split a model into multiple subgraphs and assign them to different devices

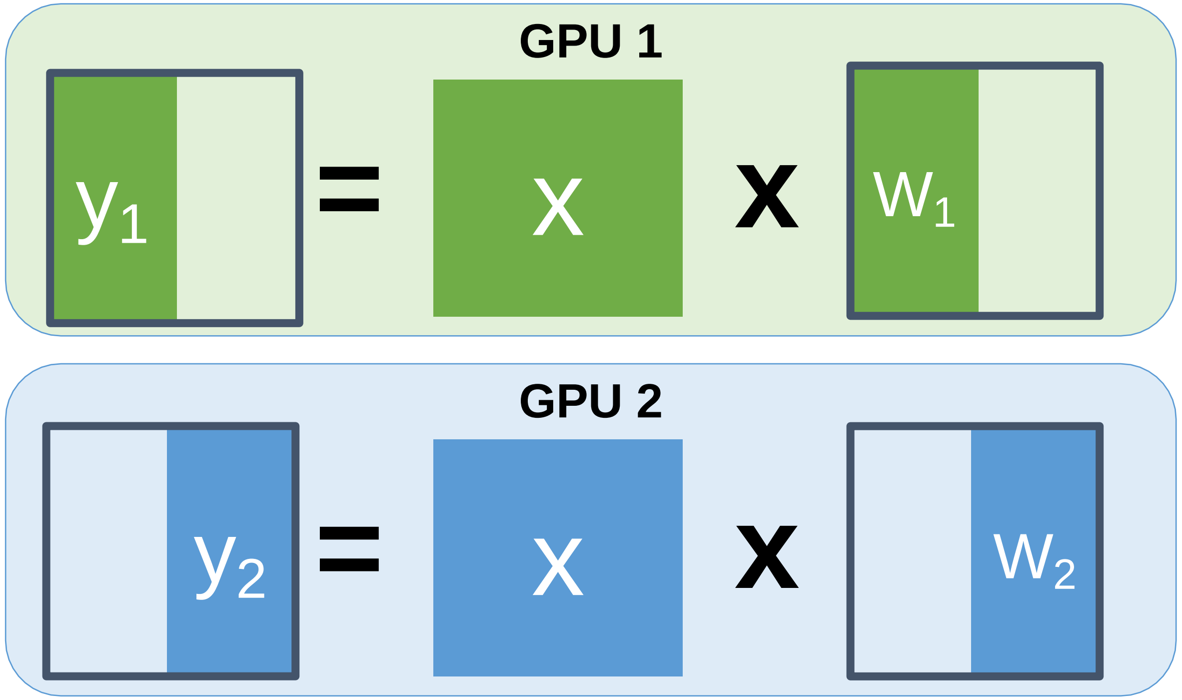


$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n}\sum_{j=1}^{n} \nabla L_j(w_i)$$

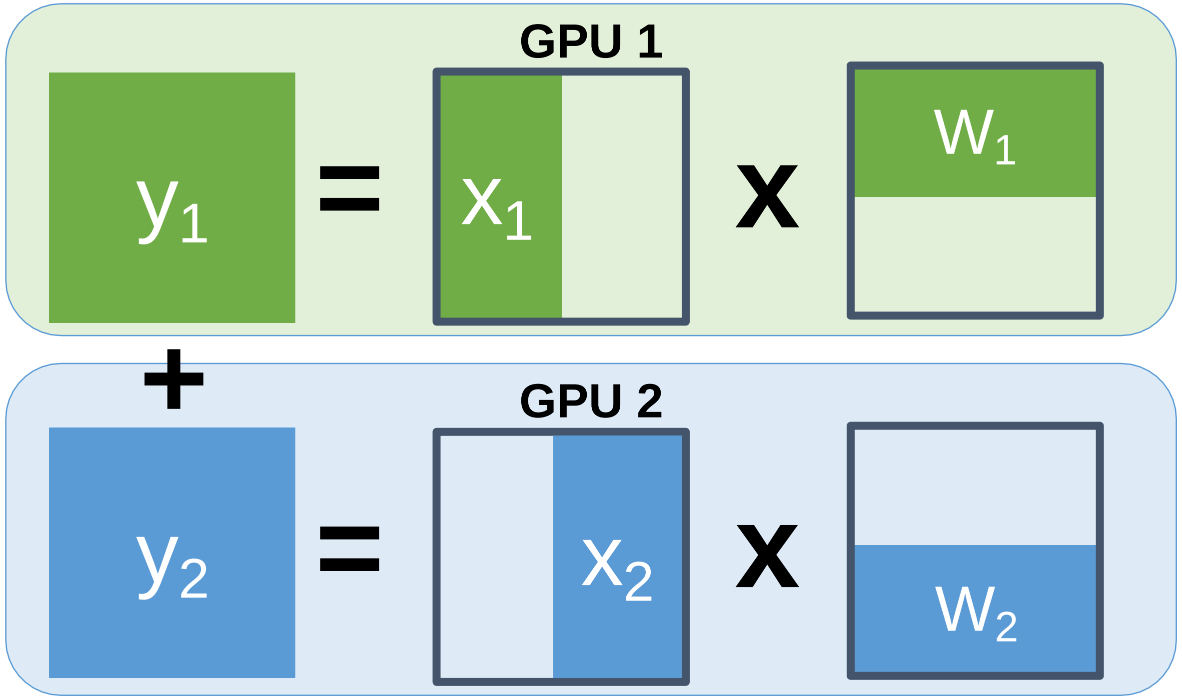
Tensor model parallelism



- Partition parameters/gradients ***within*** a layer



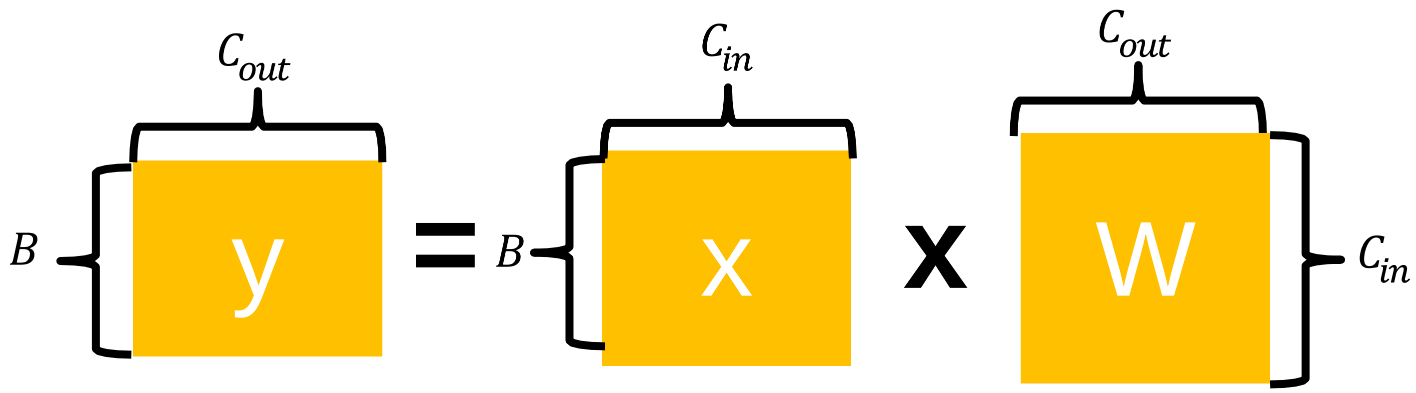
Tensor Model Parallelism (partition output)

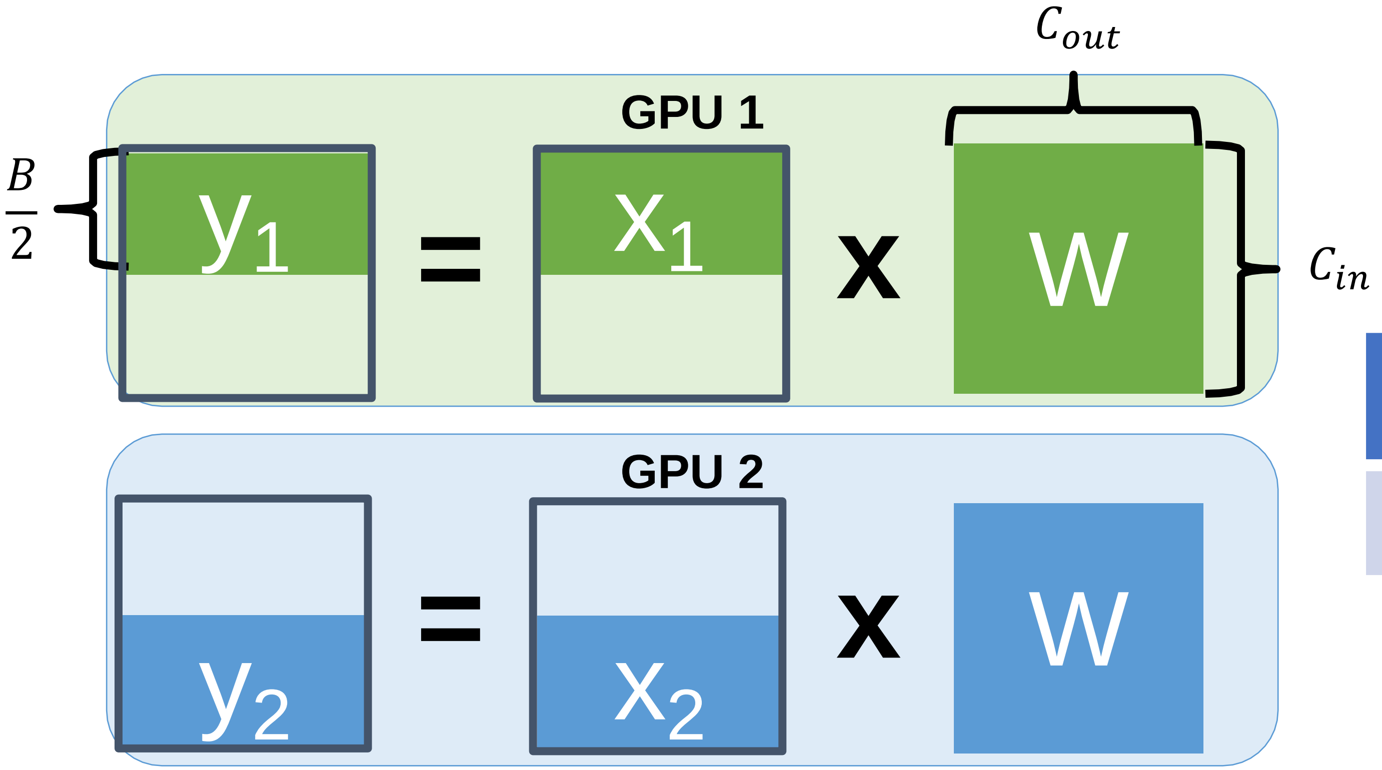


Tensor Model Parallelism (reduce output)

$y = y1 + y2$

Comparing data and tensor model parallelism



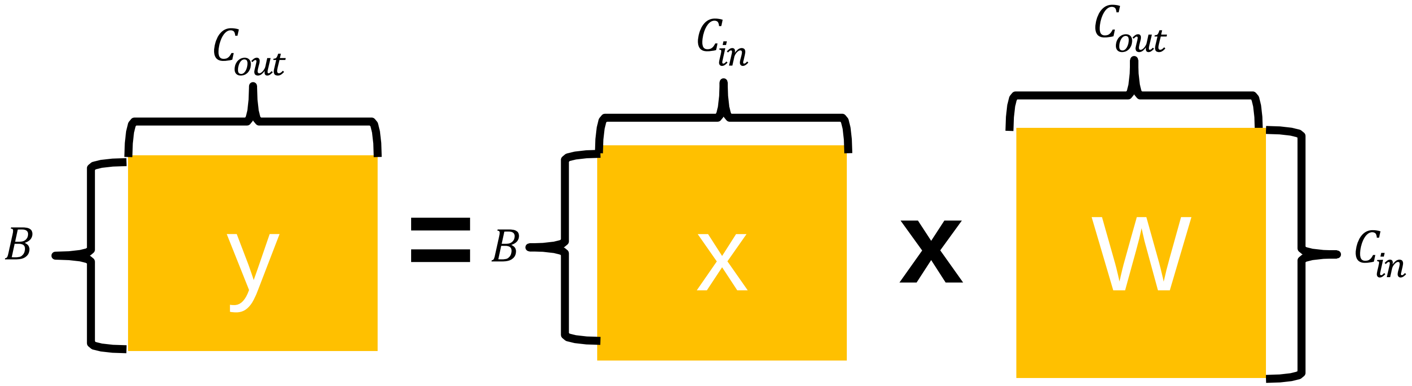


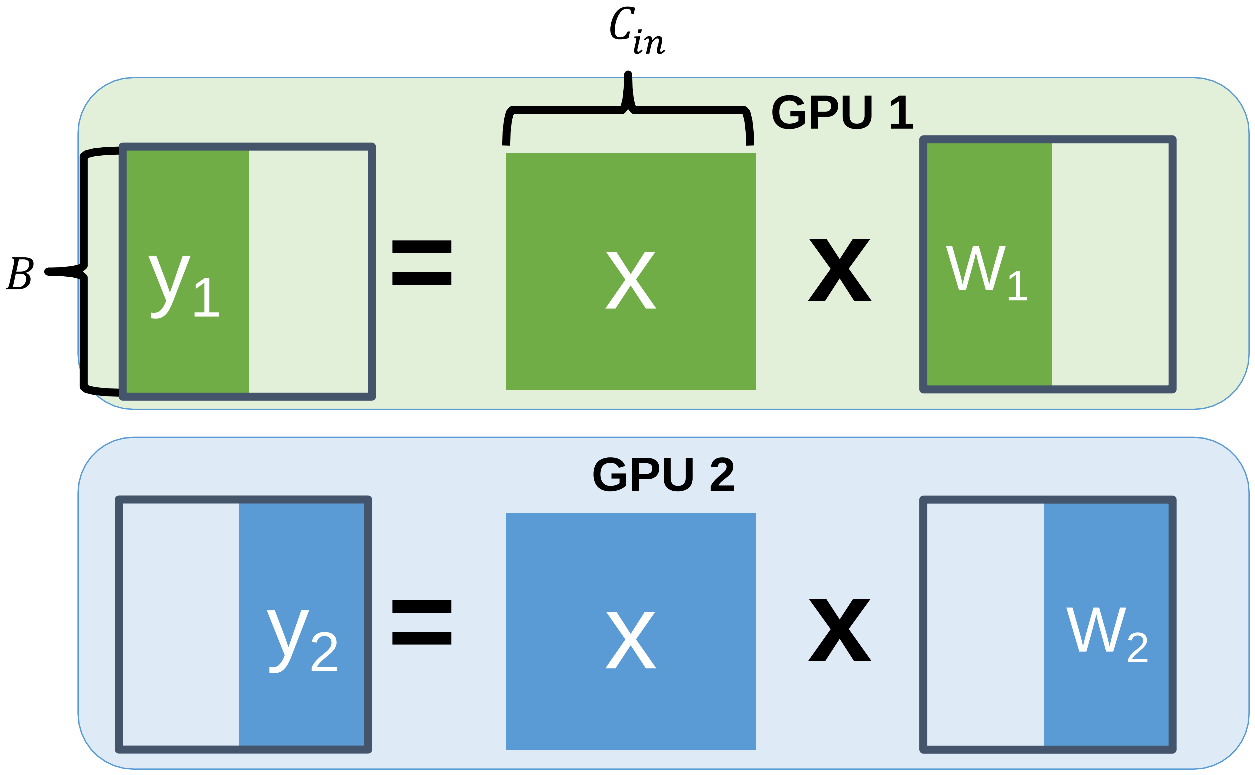
Data parallelism $y = Wx$

Forward Processing	Backward Propagation	Gradients Sync
0	0	$O(C_{out} * C_{in})$

Communication Cost of Data Parallelism

Comparing data and tensor model parallelism



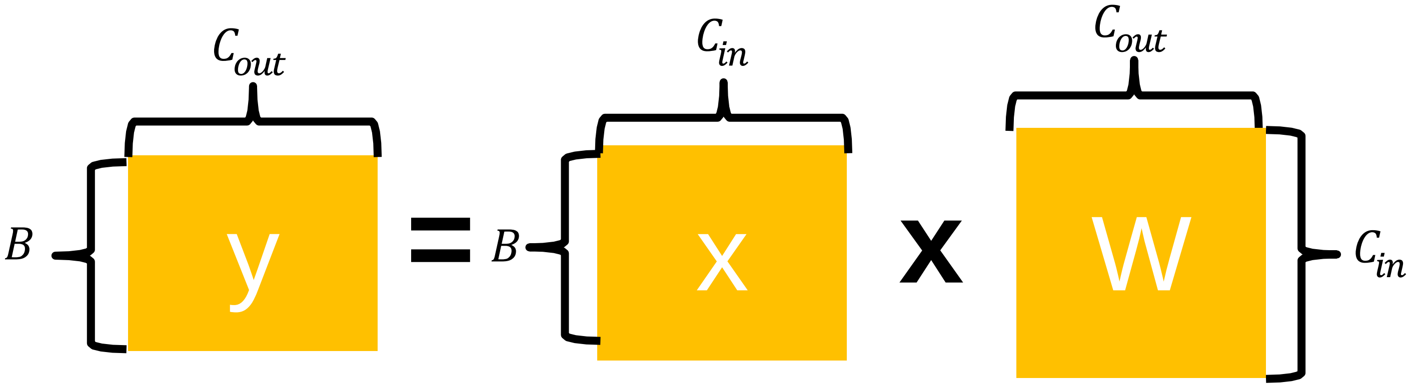


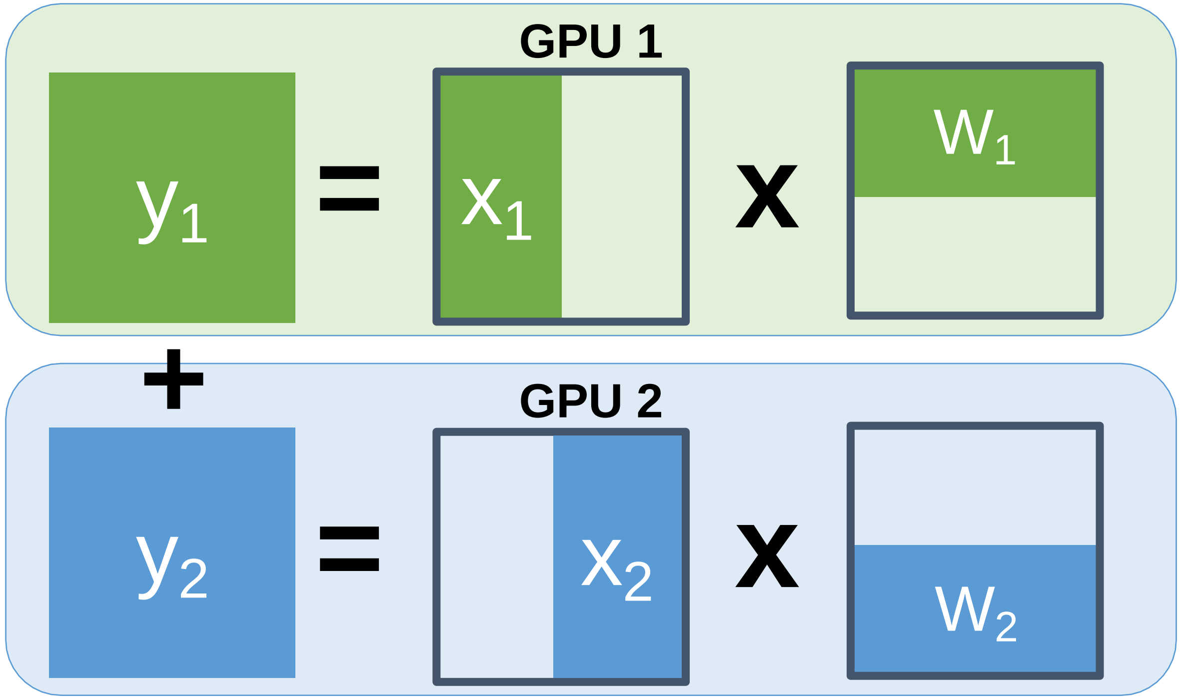
Tensor Model Parallelism (partition output)

Forward Processing	**Backward Propagation**	**Gradients Sync**
$O(B * C_{in})$	$O(B * C_{in})$	0

Communication Cost of Tensor Model Parallelism

Comparing data and tensor model parallelism





Tensor Model Parallelism (Reduce output)

$y = y1 + y2$

Forward Processing	Backward Propagation	Gradients Sync
$O(B * C_{out})$	$O(B * C_{out})$	0

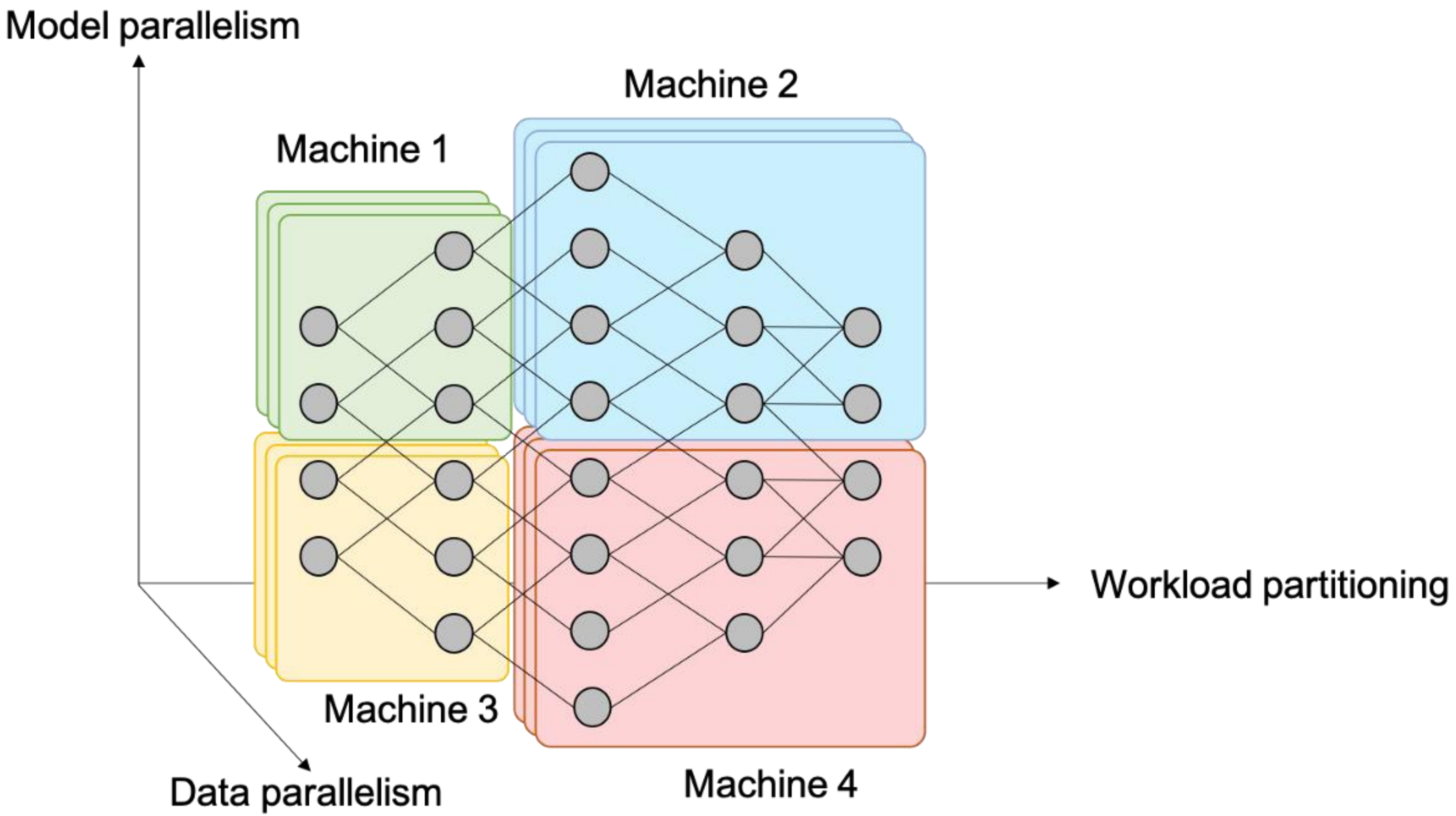
Communication Cost of Tensor Model Parallelism

Comparing data and tensor model parallelism

- Data parallelism: $O(C_{out} * C_{in})$
- Tensor model parallelism (partition output): $O(B * C_{in})$
- Tensor model parallelism (reduce output): $O(B * C_{out})$

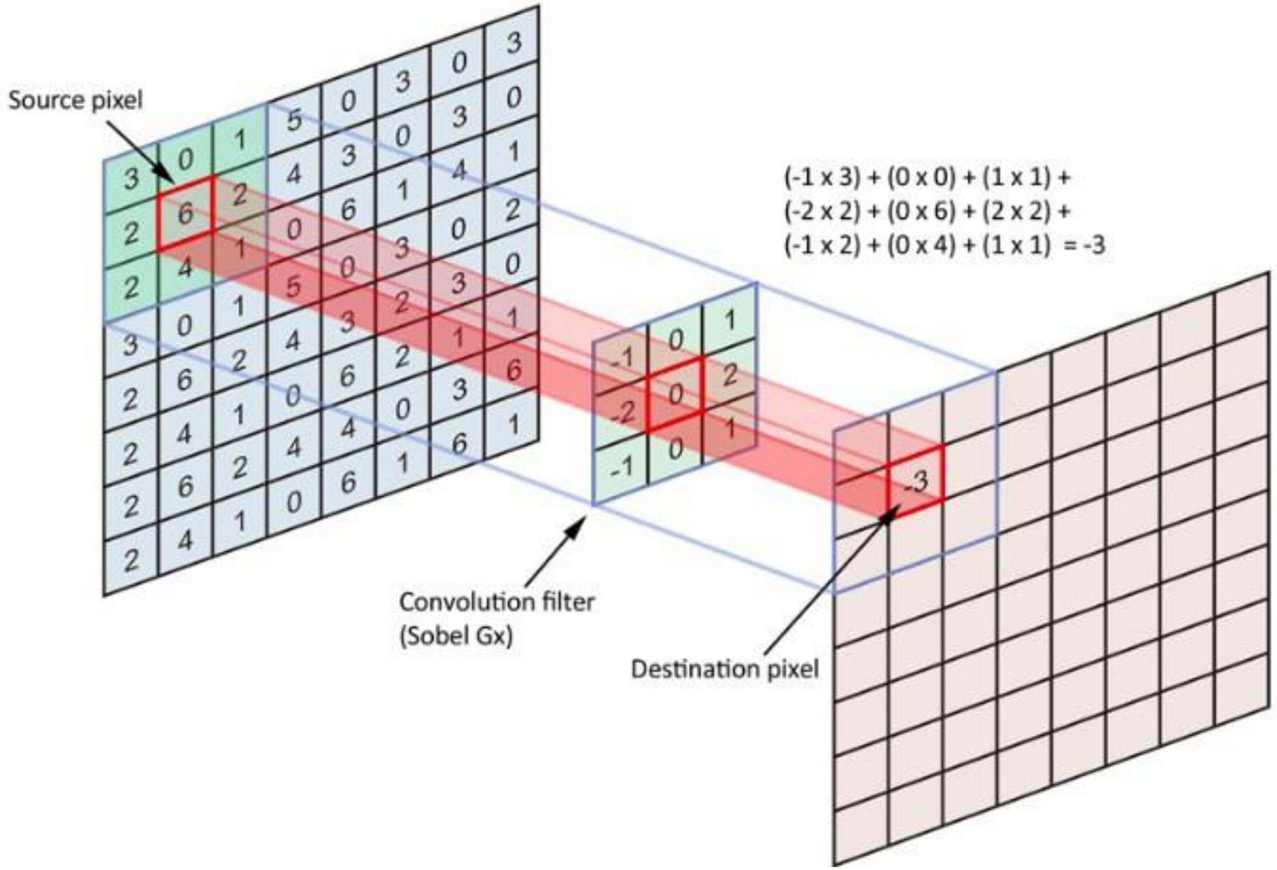
- **The best strategy depends on the model size and underlying infrastructures**

Combine data and model parallelism



Convolutional Neural Networks

- Convolve the filter with the image: slide over the image spatially and compute dot products



Parallelizing Convolutional Neural Networks

- Convolutional layers
  - 90-95% of the computation
  - 5% of the parameters
  - Very large intermediate activations

Data parallelism

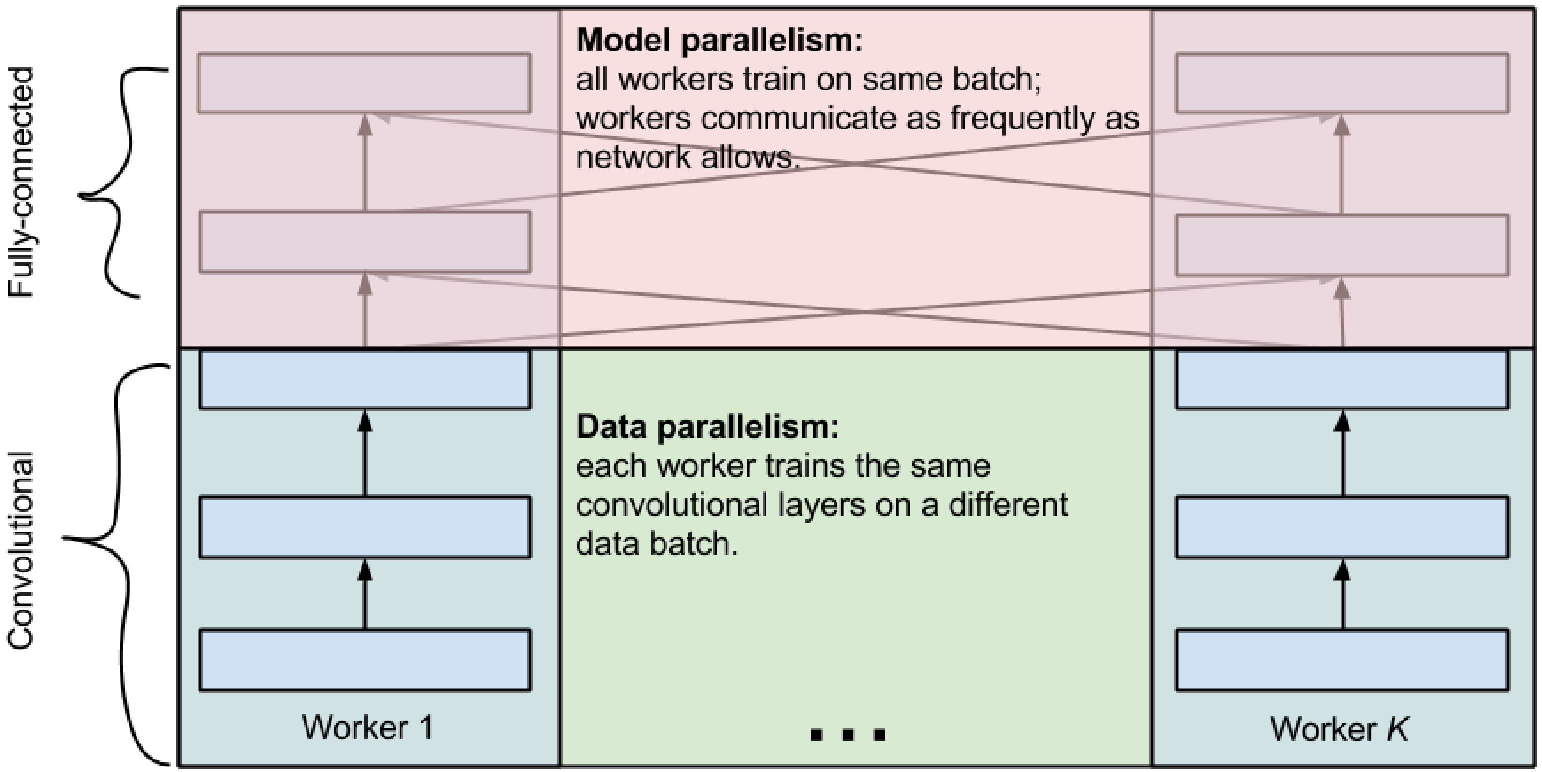
- Fully-connected layers
  - 5-10% of the computation
  - 95% of the parameters
  - Small intermediate activations

Tensor model parallelism

- **How to parallelize CNNs?**

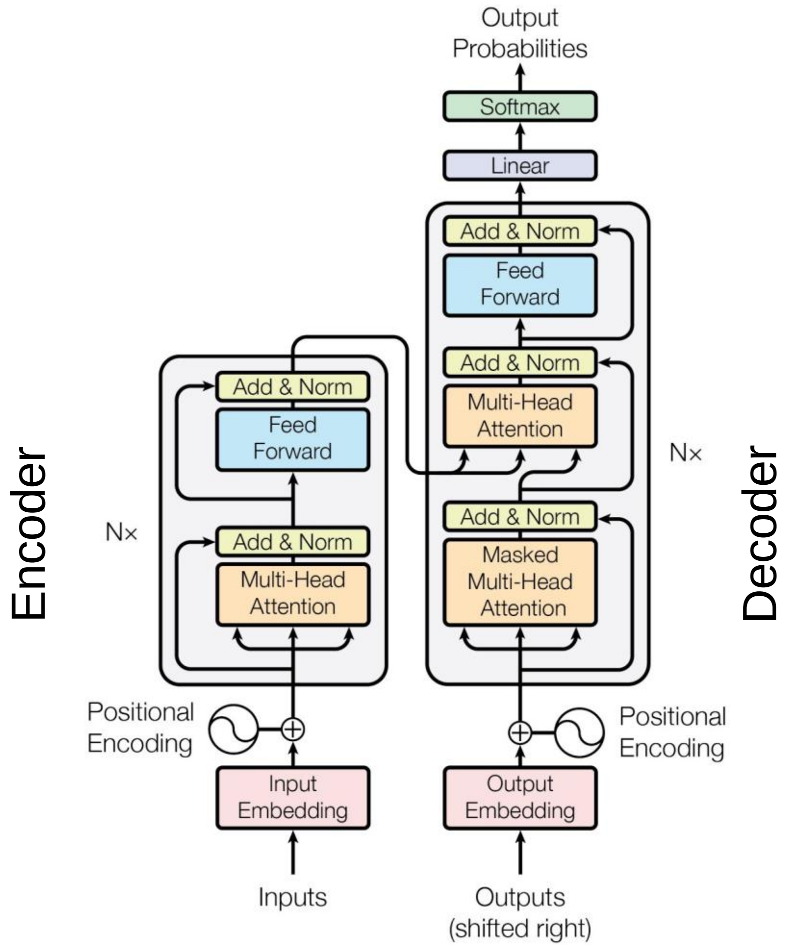
Parallelizing Convolutional Neural Networks

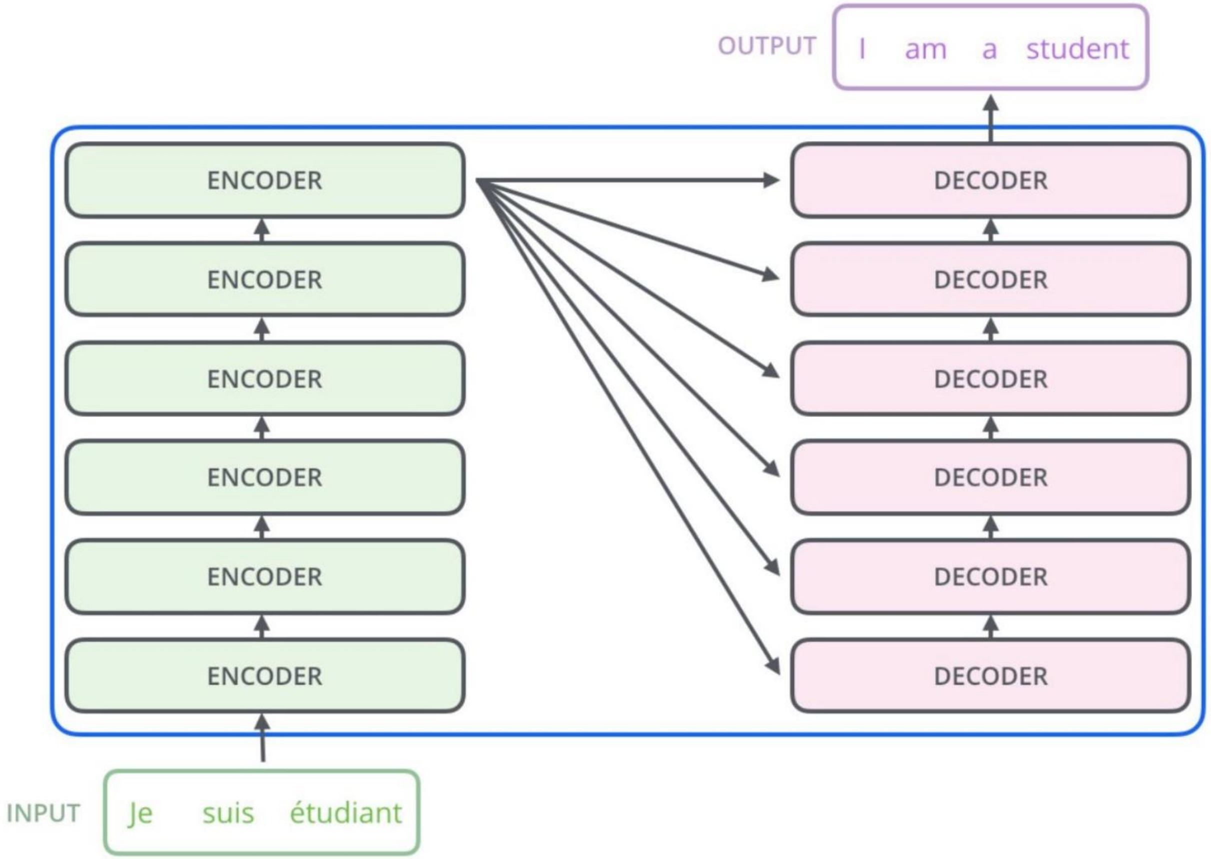
- Data parallelism for convolutional layers
- Tensor model parallelism for fully-connected layers



Parallelizing Transformers

- Transformer: attention mechanism for language understanding



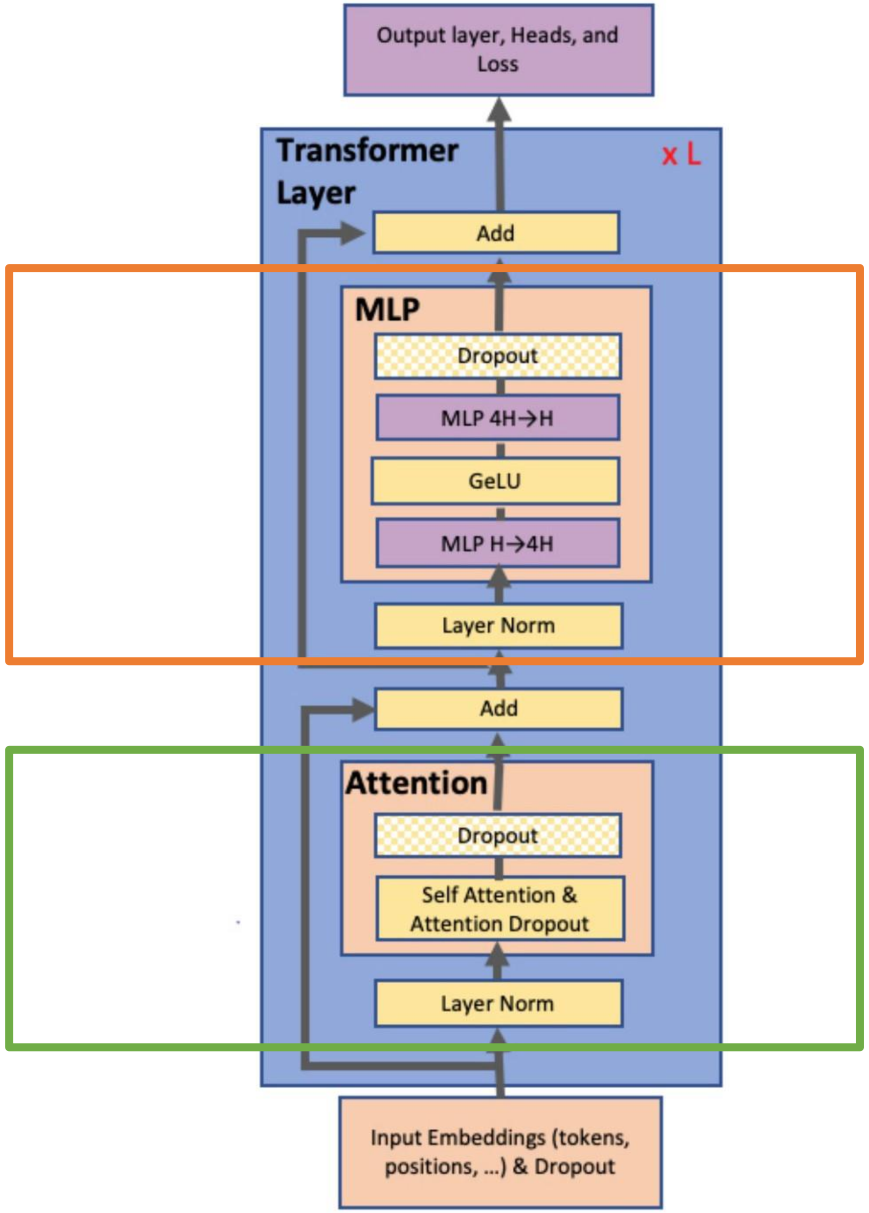


Ashish Vaswani et. al. Attention is all you need.

A Single Transformer Layer

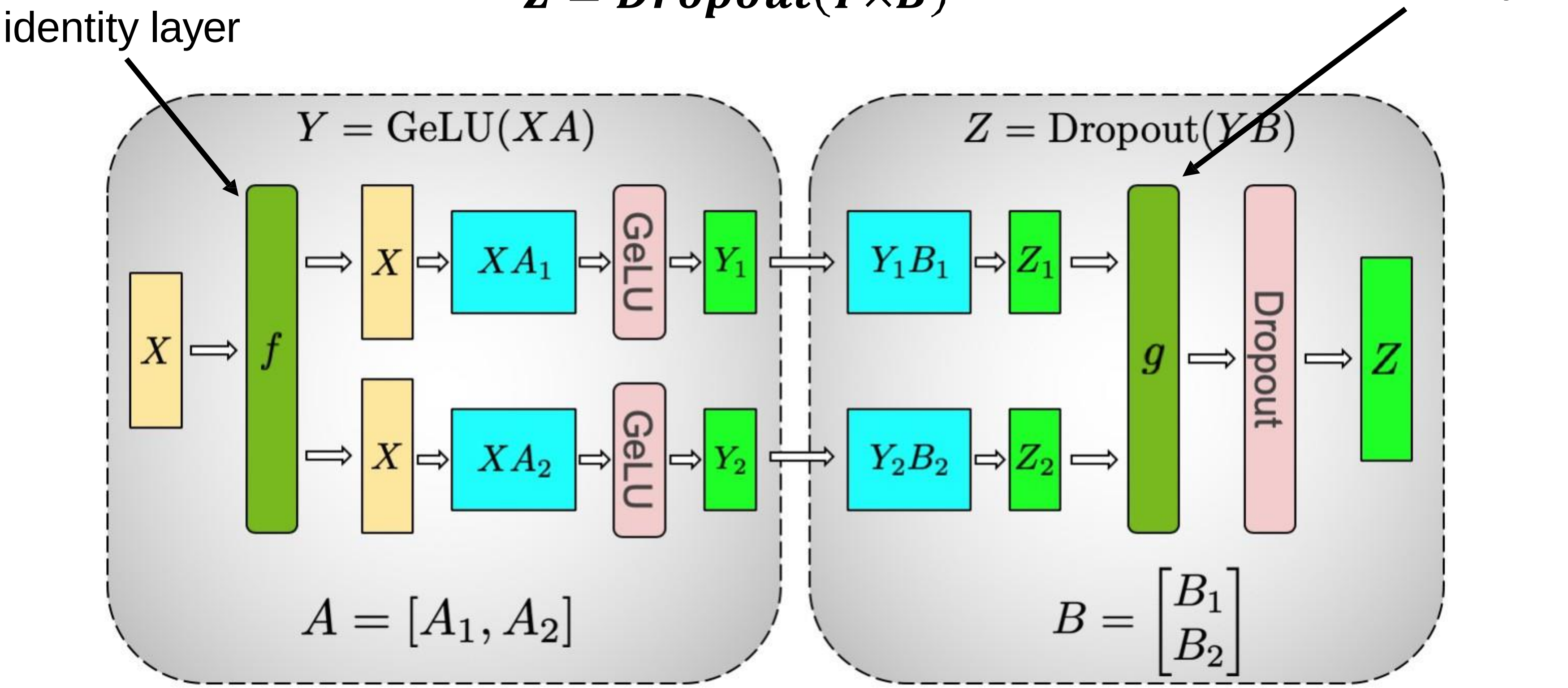
Fully-Connected Layers

Self-Attention Layers



Parallelizing Fully-Connected Layers in Transformers

$$Y = GeLU(X \times A)$$
$$Z = Dropout(Y \times B)$$

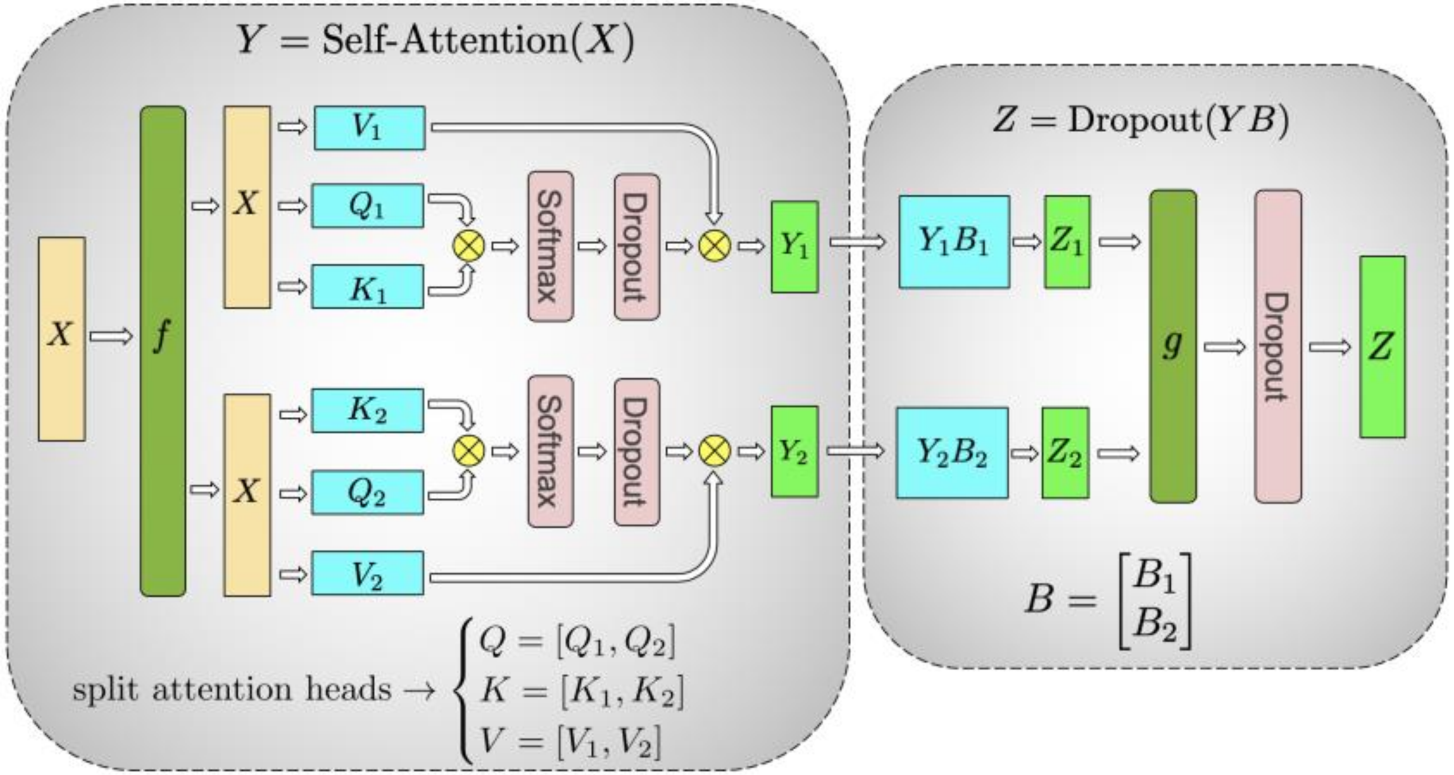


Tensor model parallelism (partition output)

Tensor model parallelism (reduce output)

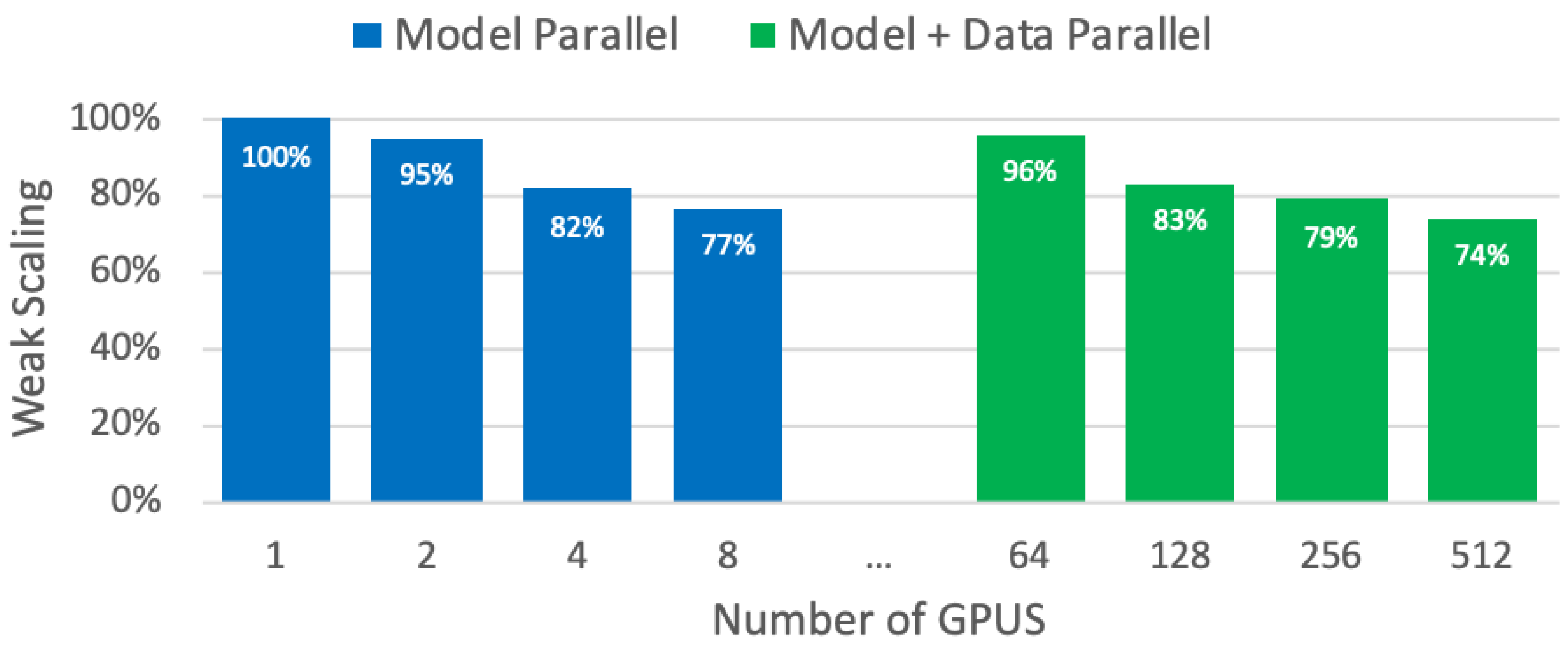
Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.

Parallelizing Self-Attention Layers in Transformers



Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.

Parallelizing Transformers

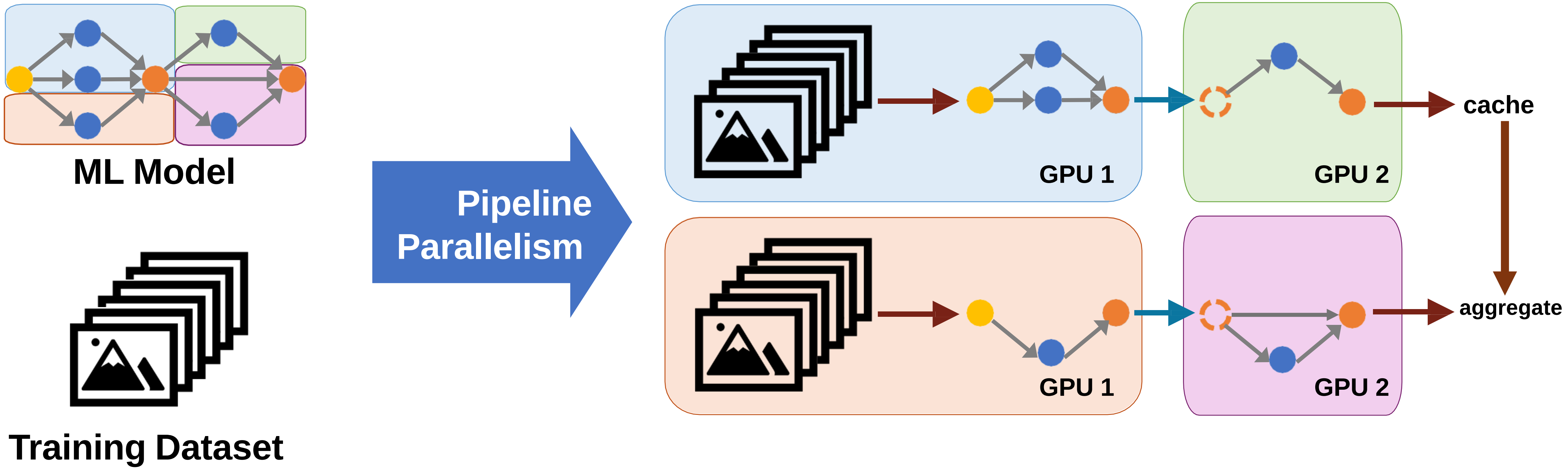


Scale to 512 GPUs by combining data and model parallelism

Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.

Execution of the compute graph: pipeline parallelism

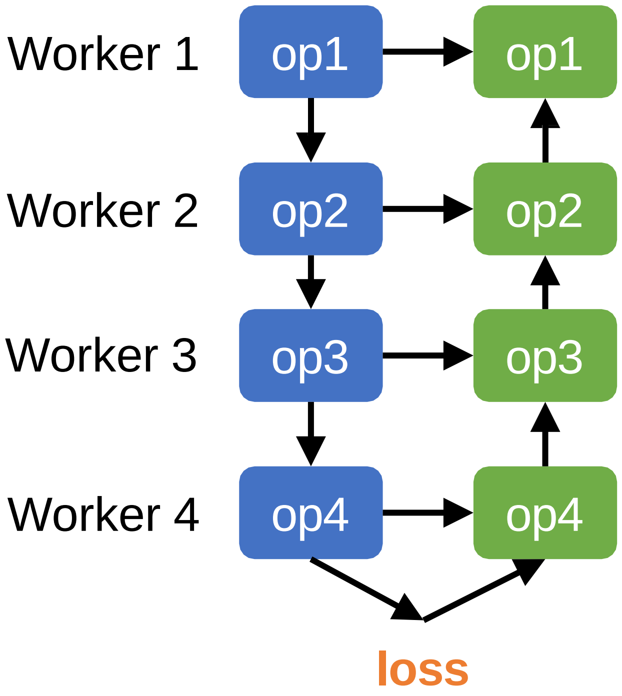
- Split a model into multiple subgraphs and assign them to different devices. Run them by proper scheduling.

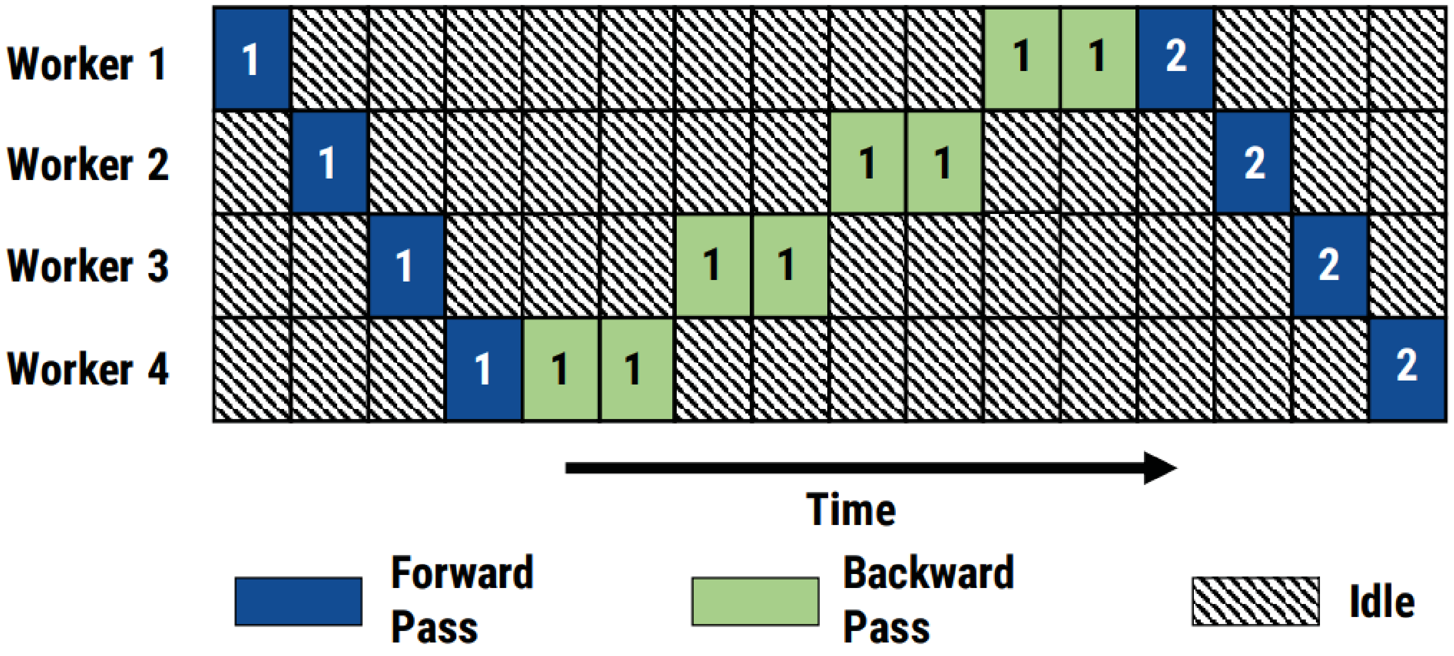


$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n}\sum_{j=1}^{n} \nabla L_j(w_i)$$

Issues with tensor / model parallelism

- Under-utilization of compute resources
- Low overall throughput due to resource utilization



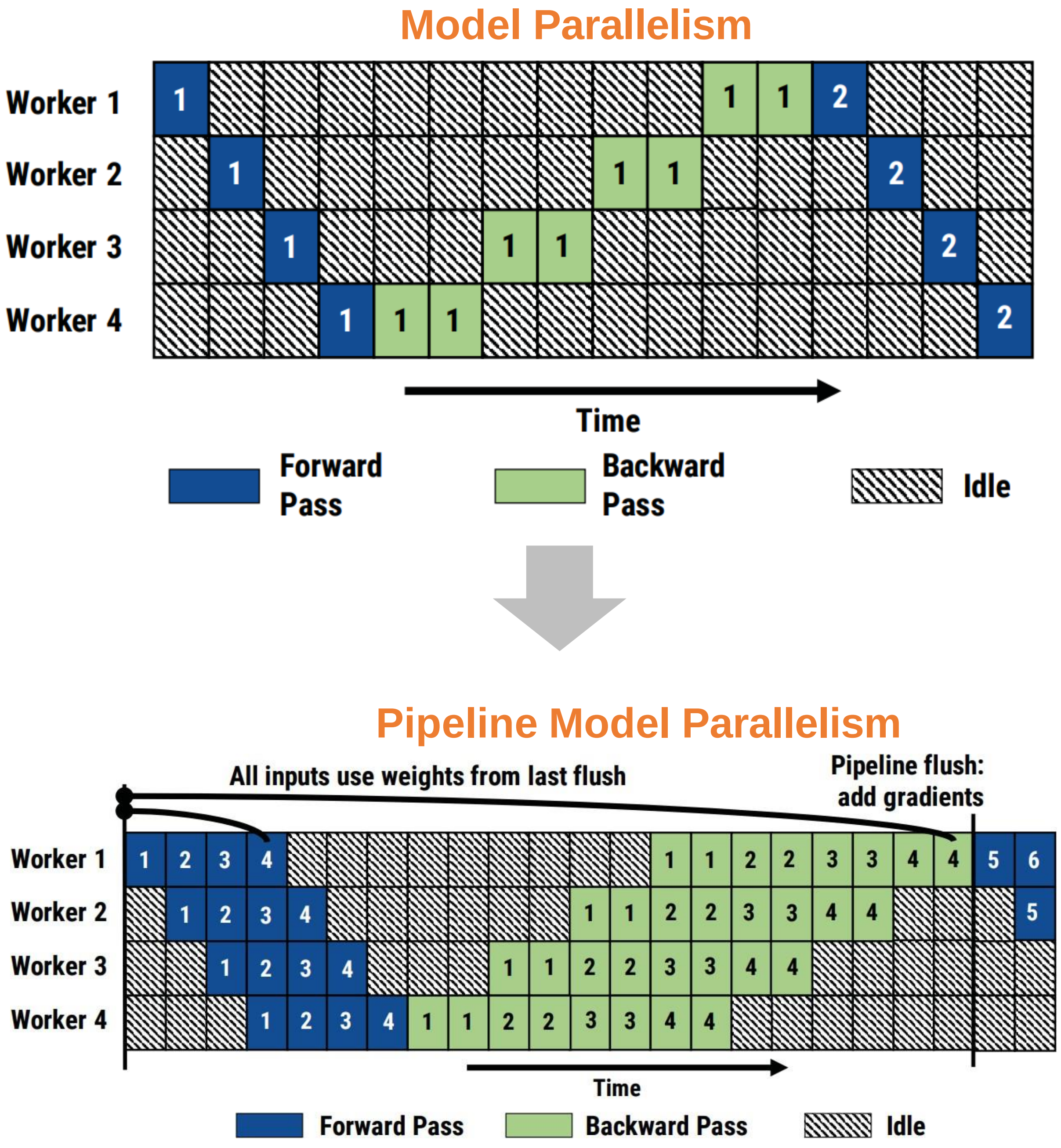


Pipeline parallelism

- **Mini-batch**: the number of samples processed in each iteration

- Divide a mini-batch into multiple **micro-batches**
  (by partitioning the compute graph)

- Pipeline the forward and backward computations across micro-batches



Pipeline parallelism

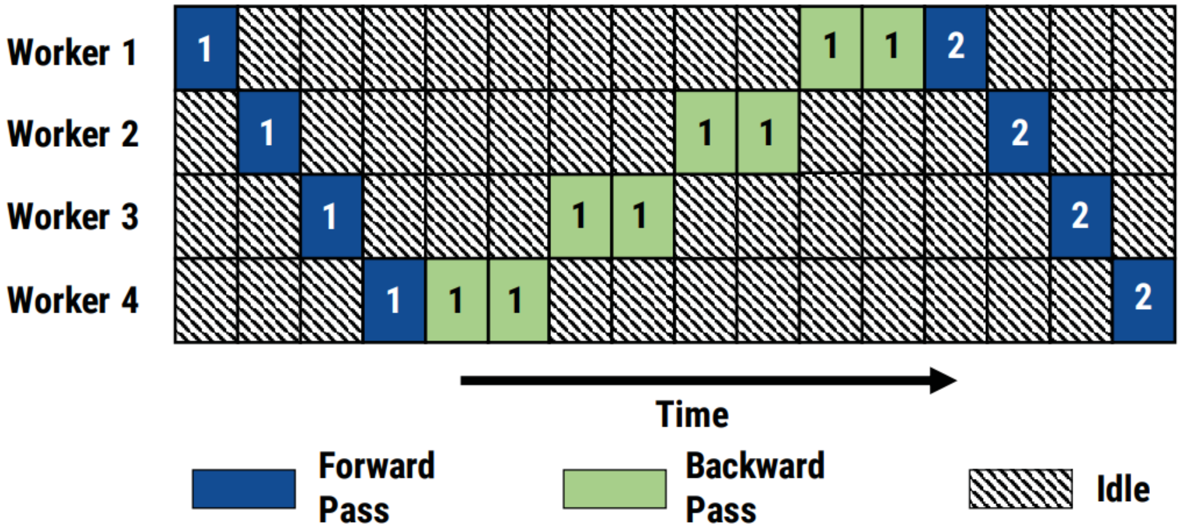
- **Mini-batch**: the number of samples processed in each iteration

- Divide a mini-batch into multiple **micro-batches**
  (by partitioning the compute graph)

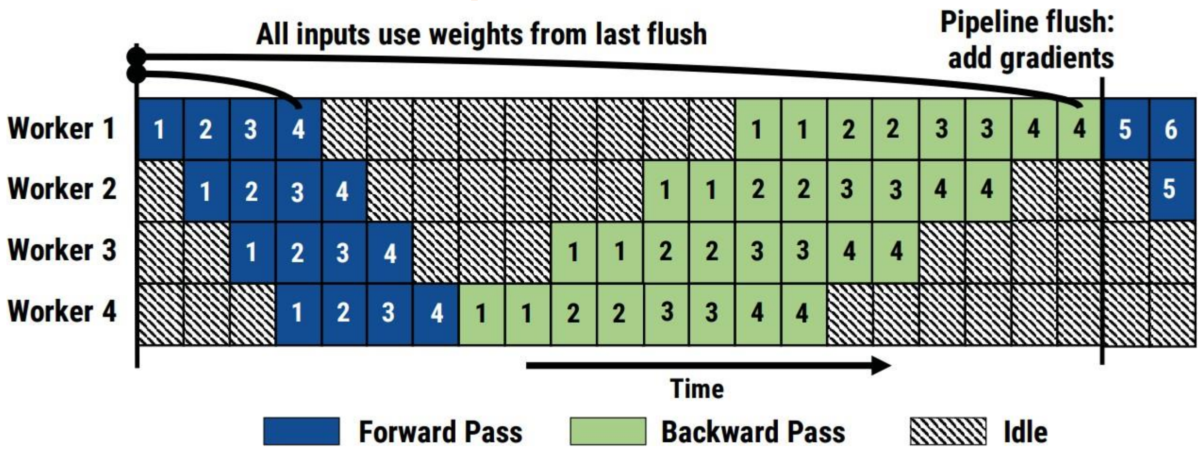
- Pipeline the forward and backward computations across micro-batches

Improving resource utilization

Model Parallelism

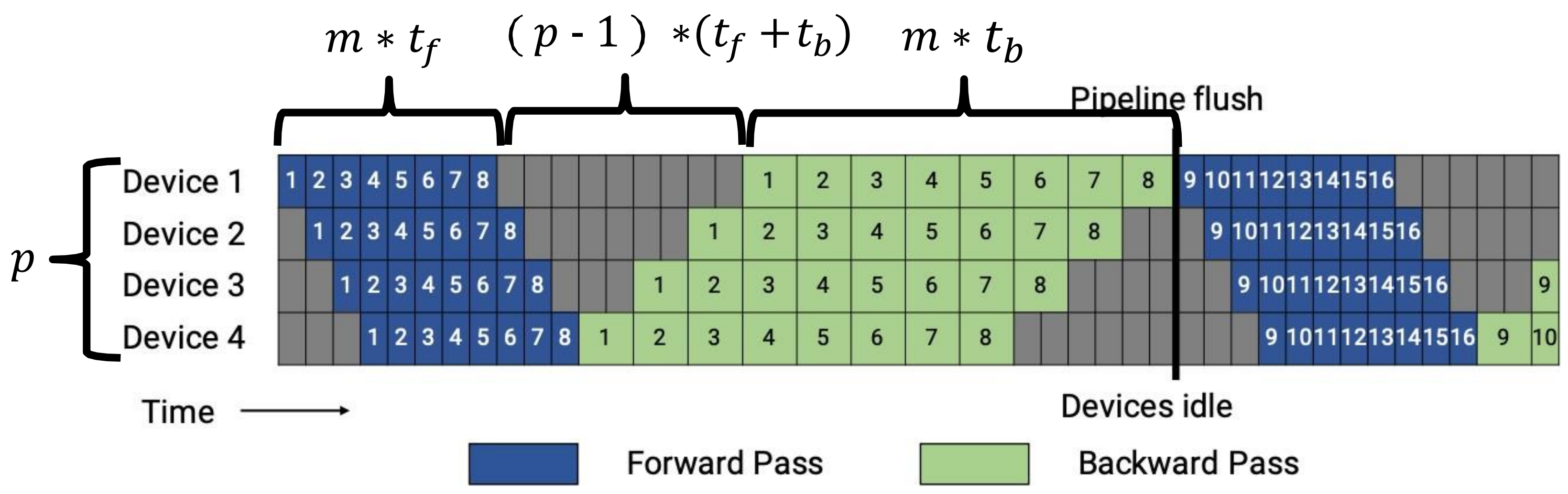


Pipeline Model Parallelism



Pipeline parallelism: device utilization

- m : micro-batches in a mini-batch
- p: number of pipeline stages
- All stages take t_f / t_b to process a forward (backward) micro-batch

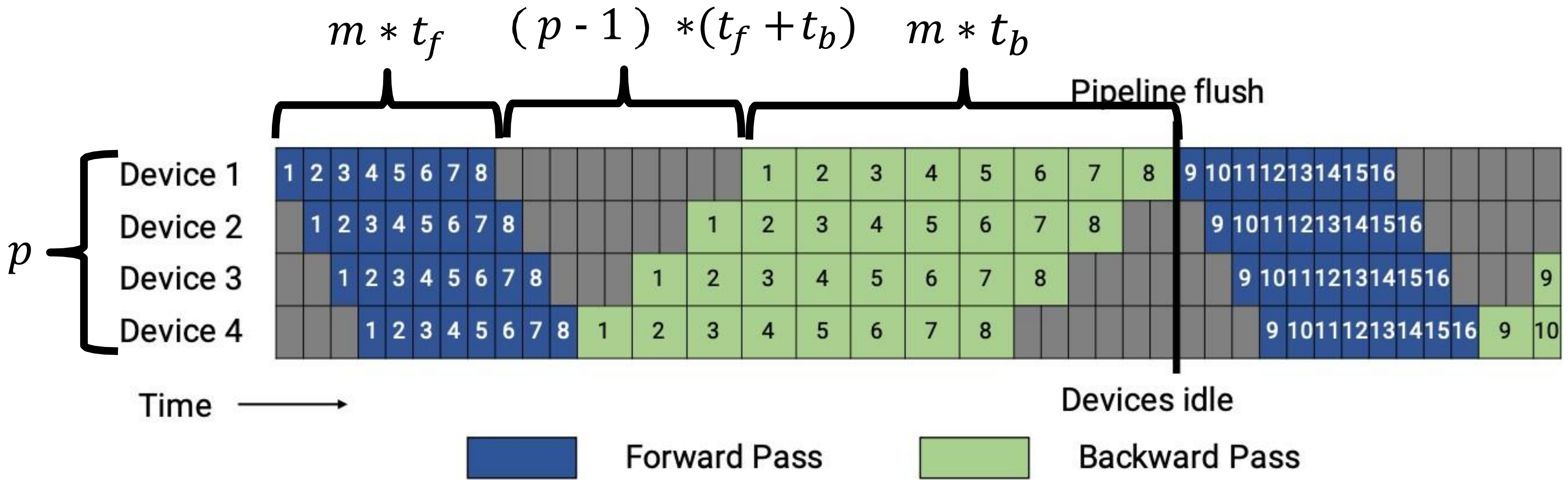


$$\boldsymbol{BubbleFraction} = \frac{(\boldsymbol{p}-\mathbf{1}) * (\boldsymbol{t_f}+\boldsymbol{t_b})}{\boldsymbol{m} * \boldsymbol{t_f} + \boldsymbol{m} * \boldsymbol{t_b}} = \frac{\boldsymbol{p}-\mathbf{1}}{\boldsymbol{m}}$$

GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

Improving pipeline parallelism efficiency

- m : number of micro-batches in a mini-batch
  - Increase mini-batch size or reduce micro-batch size
  - Caveat: large mini-batch sizes can lead to accuracy loss; small micro-batch sizes reduce GPU utilization
- p: number of pipeline stages
  - Decrease pipeline depth
  - Caveat: increase stage size

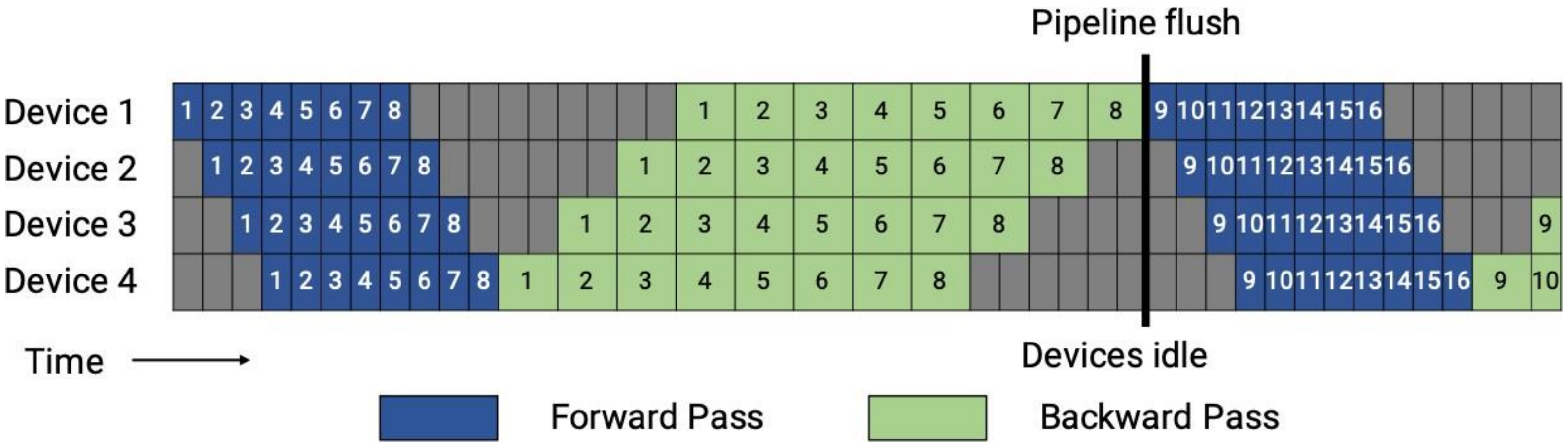


$$\boldsymbol{BubbleFraction} = \frac{(\boldsymbol{p}-\mathbf{1}) * (\boldsymbol{t_f}+\boldsymbol{t_b})}{\boldsymbol{m} * \boldsymbol{t_f} + \boldsymbol{m} * \boldsymbol{t_b}} = \frac{\boldsymbol{p}-\mathbf{1}}{\boldsymbol{m}}$$

GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

Pipeline parallelism: memory requirement

- We need to keep the intermediate activations of **all micro- batches** before back propagation



Can we improve the pipeline schedule to reduce memory requirement?

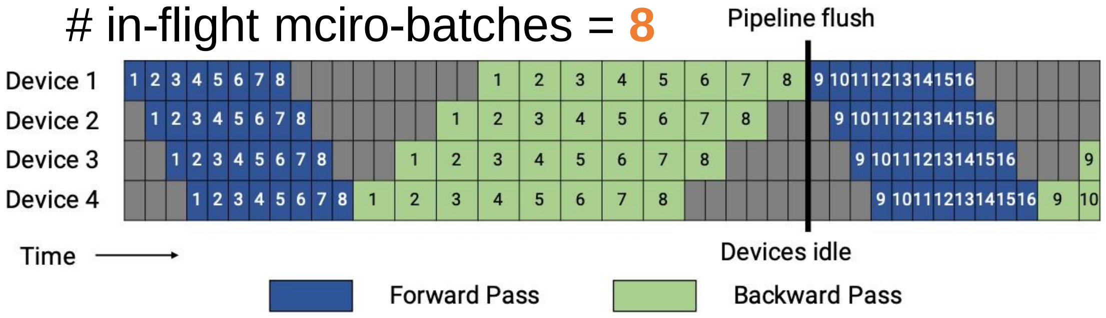
GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

Pipeline parallelism with 1F1B schedule

- One-Forward-One-Backward in the steady state
- Limit the number of in-flight micro-batches to the pipeline depth
- **Reduce memory footprint of pipeline parallelism**
- **Doesn't reduce pipeline bubble**

Can we reduce pipeline bubble?

in-flight mciro-batches = **8**



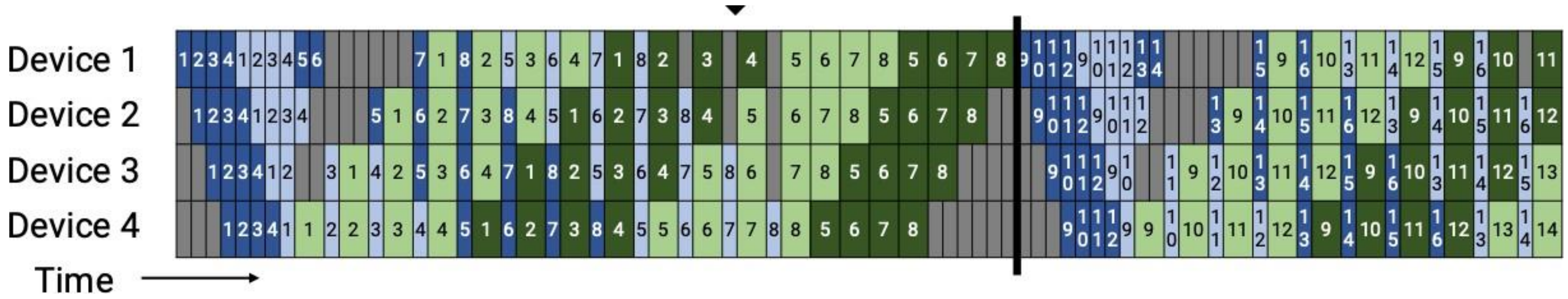
Pipeline parallelism with GPipe's schedule

in-flight mciro-batches = **4**

Pipeline parallelism with 1F1B schedule

Pipeline parallelism with interleaved 1F1B schedule

- Further divide each stage into v sub-stages
- The forward (backward) time of each sub-stage is $\frac{t_f}{v}$ $(\frac{t_b}{v})$



Each device is assigned two chunks. Dark colors show the first chunk and light colors show the second chunk.

$$\boldsymbol{BubbleFraction} = \frac{(p-1) * \frac{(t_f+t_b)}{v}}{m * t_f + m * t_b} = \frac{1}{v} * \frac{p-1}{m}$$

Reduce bubble time at the cost increased communication

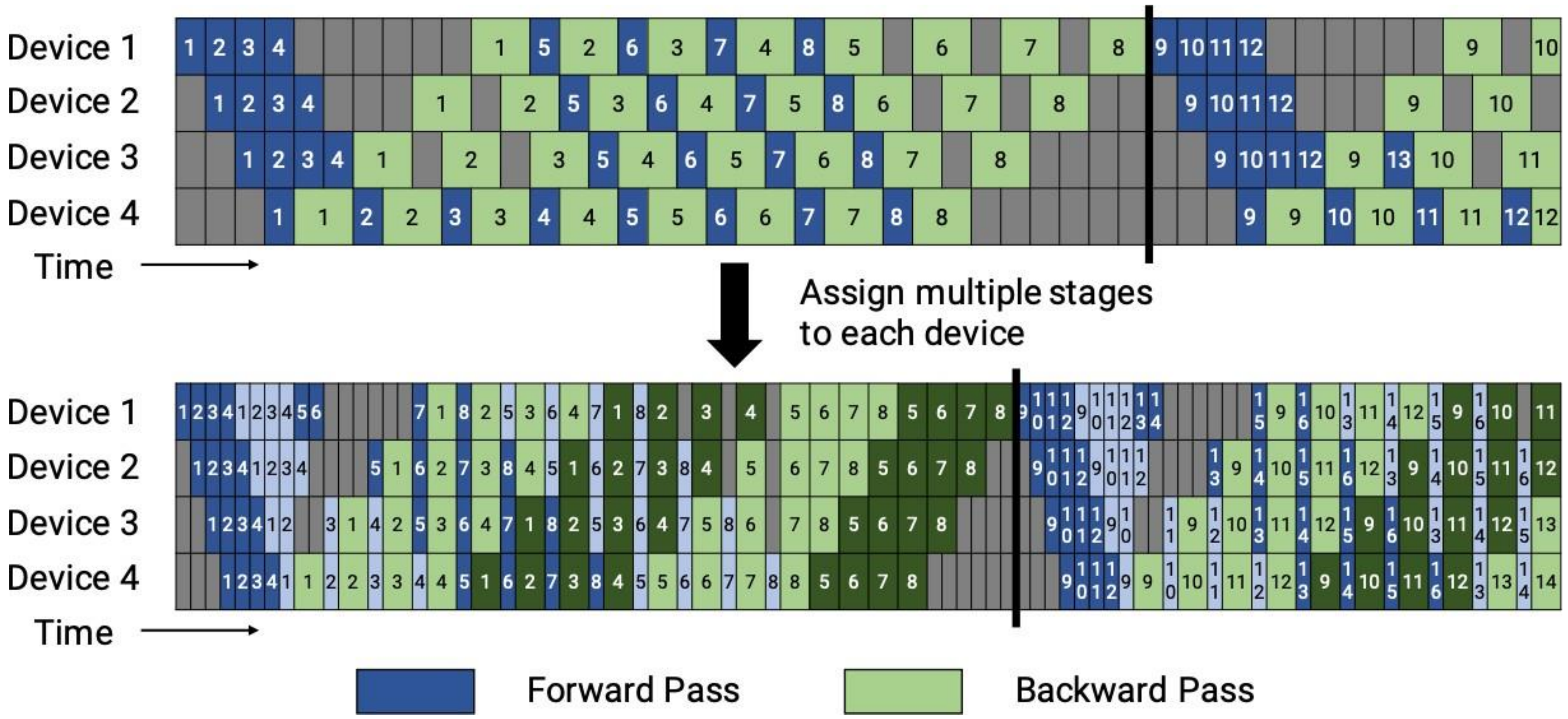
Pipeline parallelism with interleaved 1F1B schedule

Pipeline parallelism with 1F1B Schedule

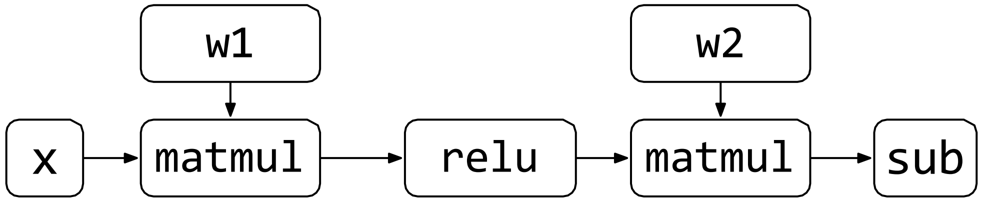
$$BubbleFraction = \frac{p-1}{m}$$

Pipeline parallelism with interleaved 1F1B Schedule

$$BubbleFraction = \frac{1}{v} * \frac{p-1}{m}$$



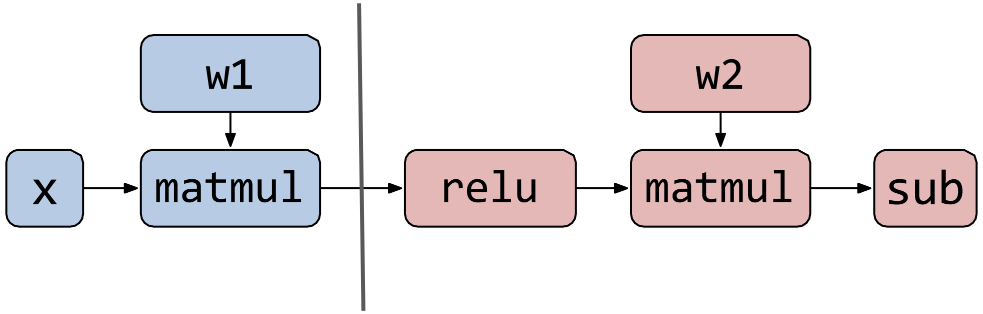
Pipeline parallelism by partitioning computational graphs



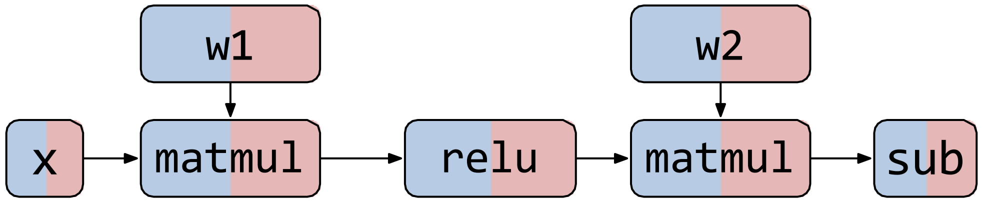
Device 1

Device 2

Strategy 1: Inter-operator Parallelism



Strategy 2: Intra-operator Parallelism



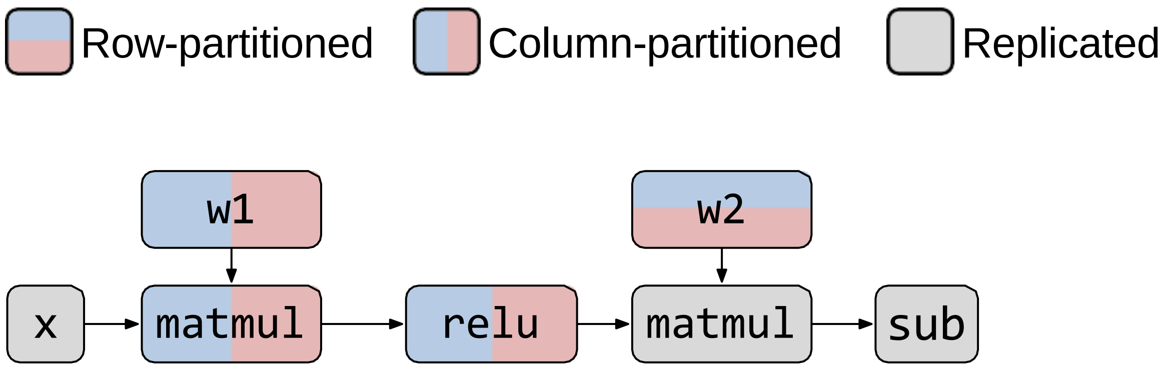
Trade-off

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

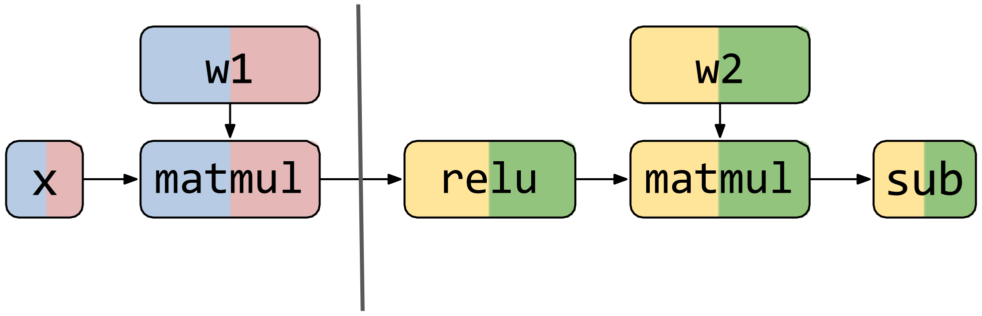
L. Zheng, et al. Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. OSDI 2022.

Pipeline parallelism by partitioning computational graphs

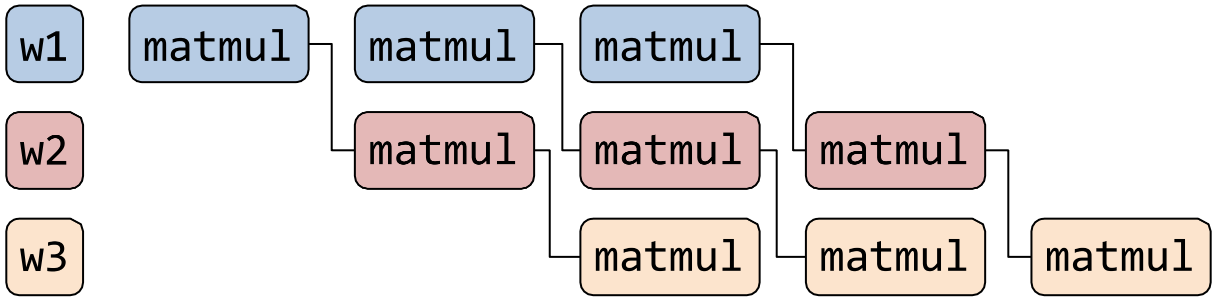
Multiple intra-op strategies for a single node



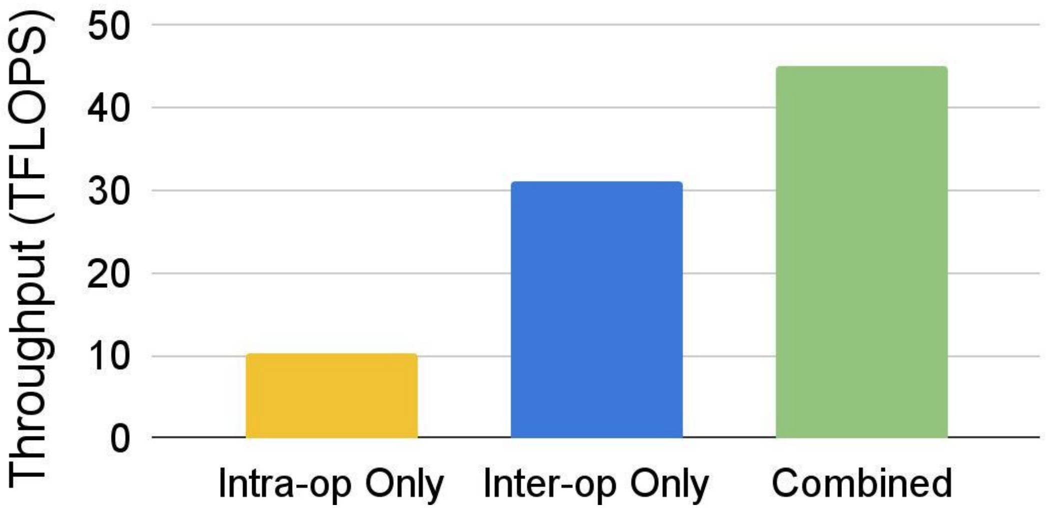
Combine Intra-op and Inter-op



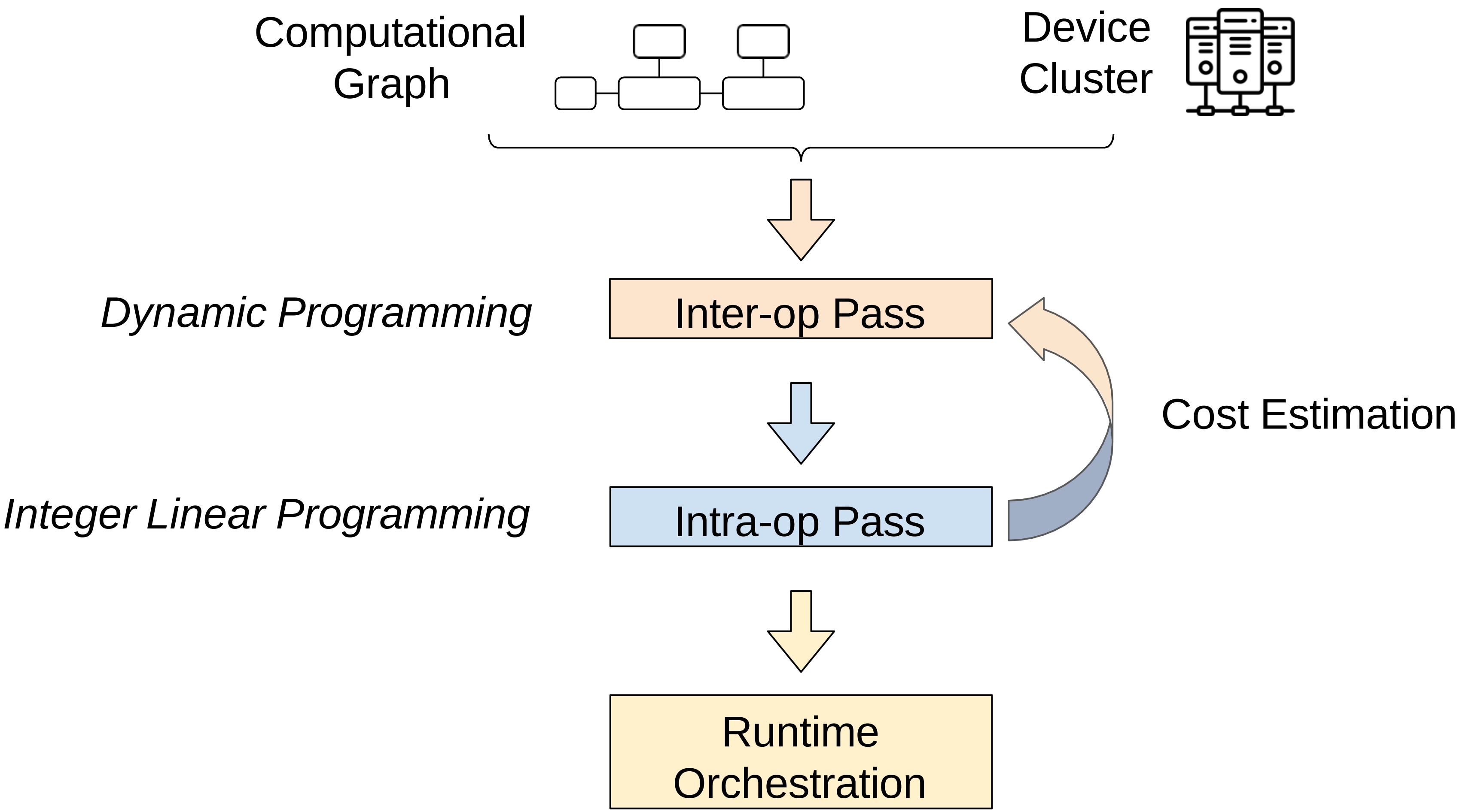
Pipeline the execution for inter-op parallelism



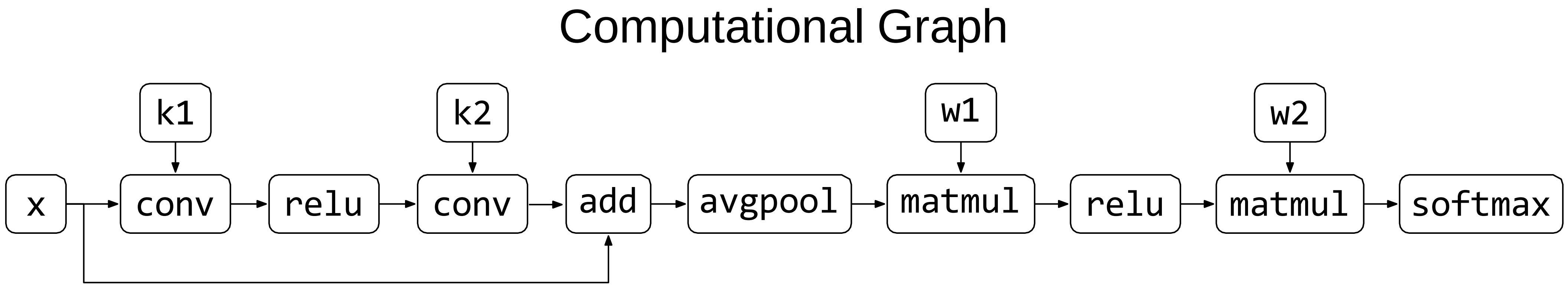
Training Throughput of an MoE Model



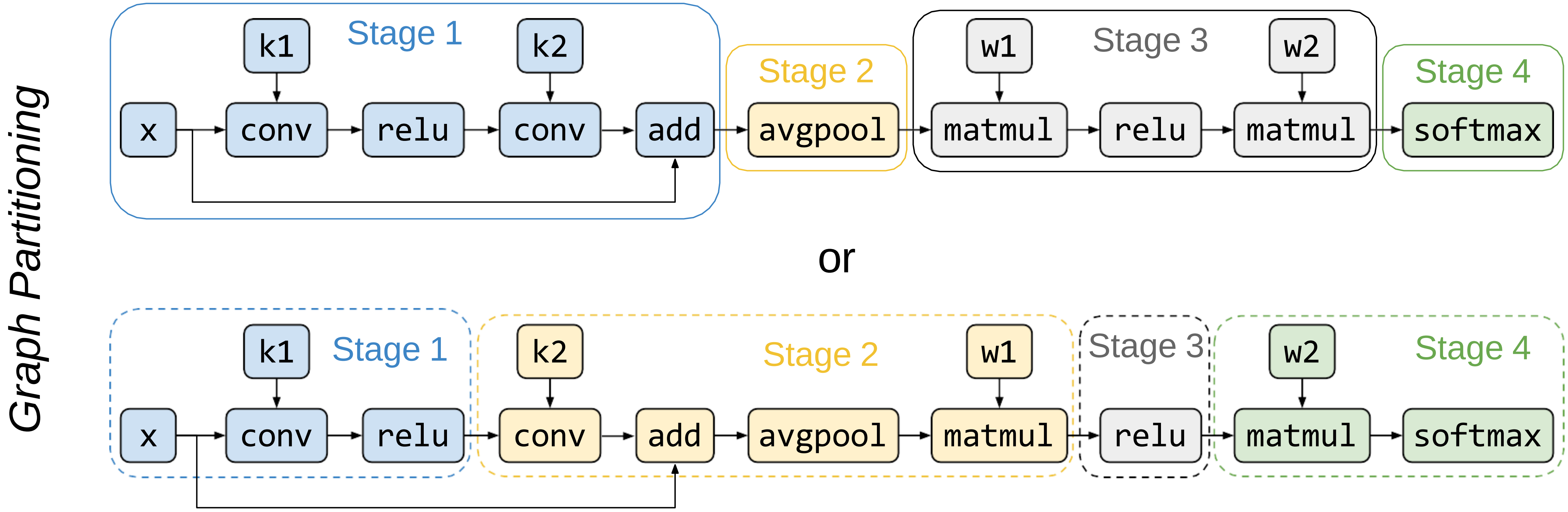
Alpa compiler: hierarchical optimization



Inter-op pass



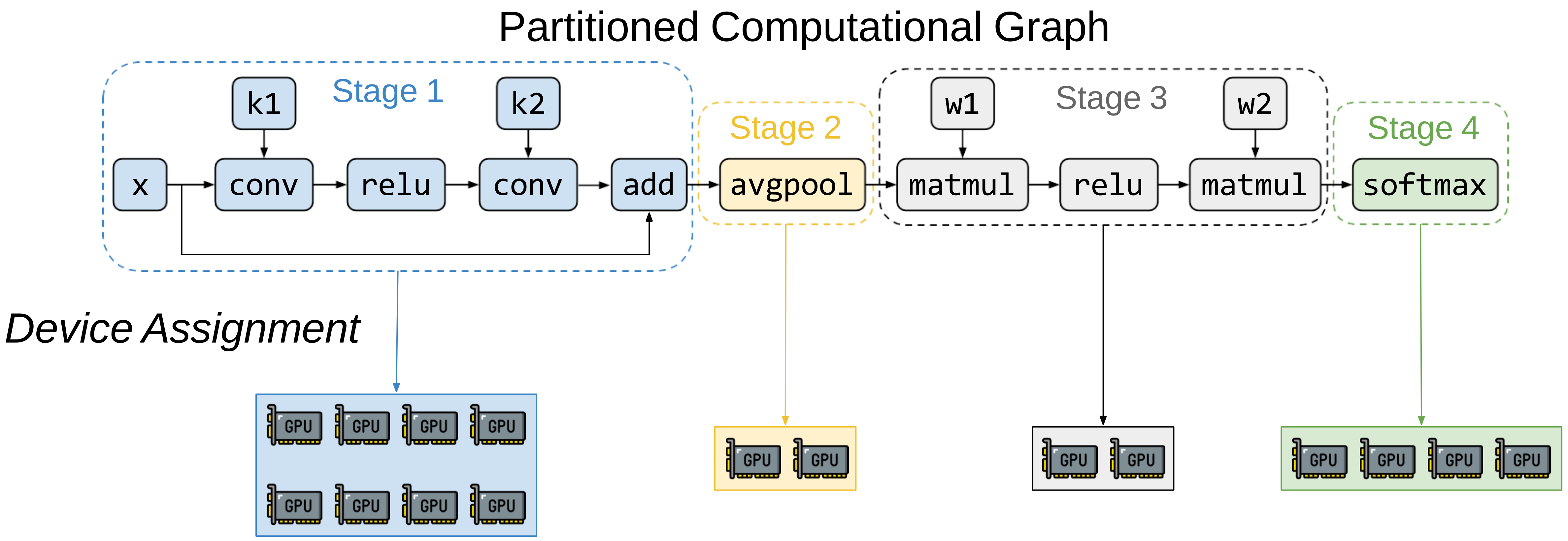
Inter-op pass



or

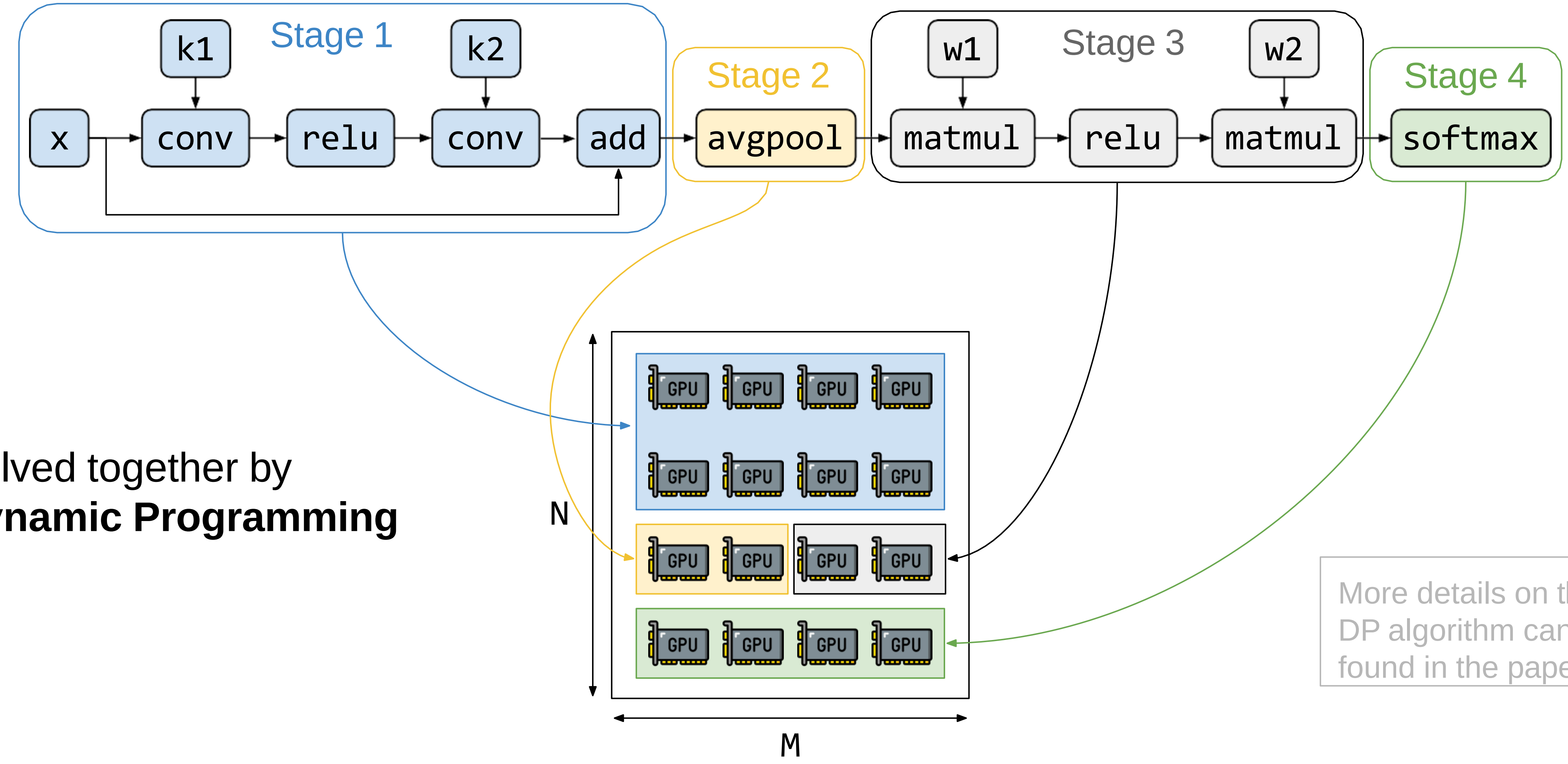
…

Inter-op pass



Improving resource utilization on heterogeneous (datacenter) infrastructures

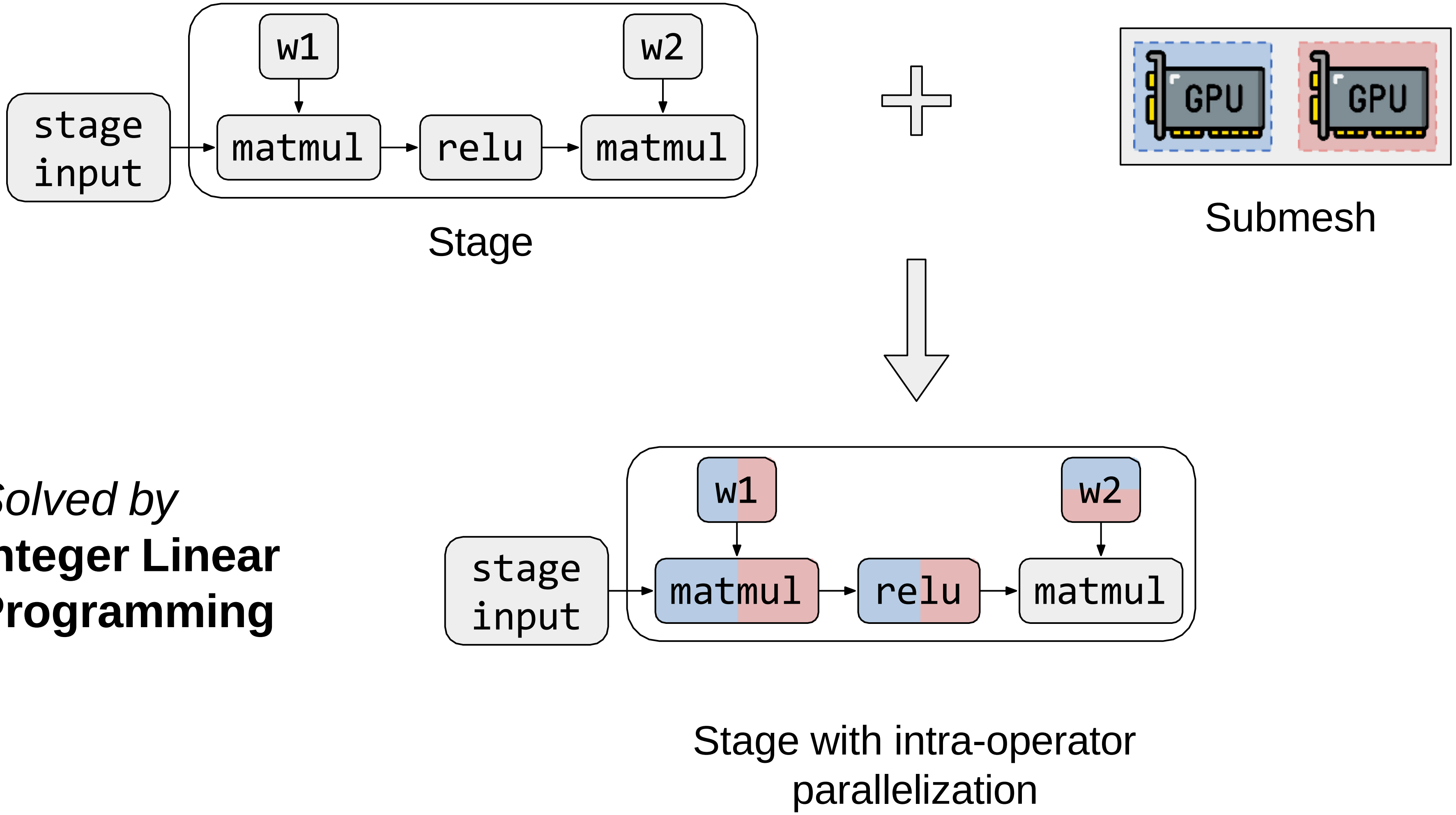
Inter-op pass



*S*olved together by
Dynamic Programming

More details on the DP algorithm can be found in the paper.

Intra-op pass

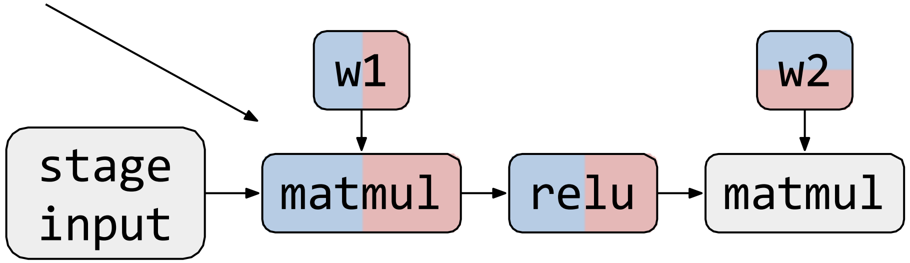


Solved by
Integer Linear Programming

Intra-op pass

Integer Linear Programming Formulation

Decision vector
Parallel strategies of each operator



Minimize Computation cost + Communication cost

More details on the ILP algorithm can be found in the paper.

Compilation time optimization

Communication-aware operator clustering in ILP & DP

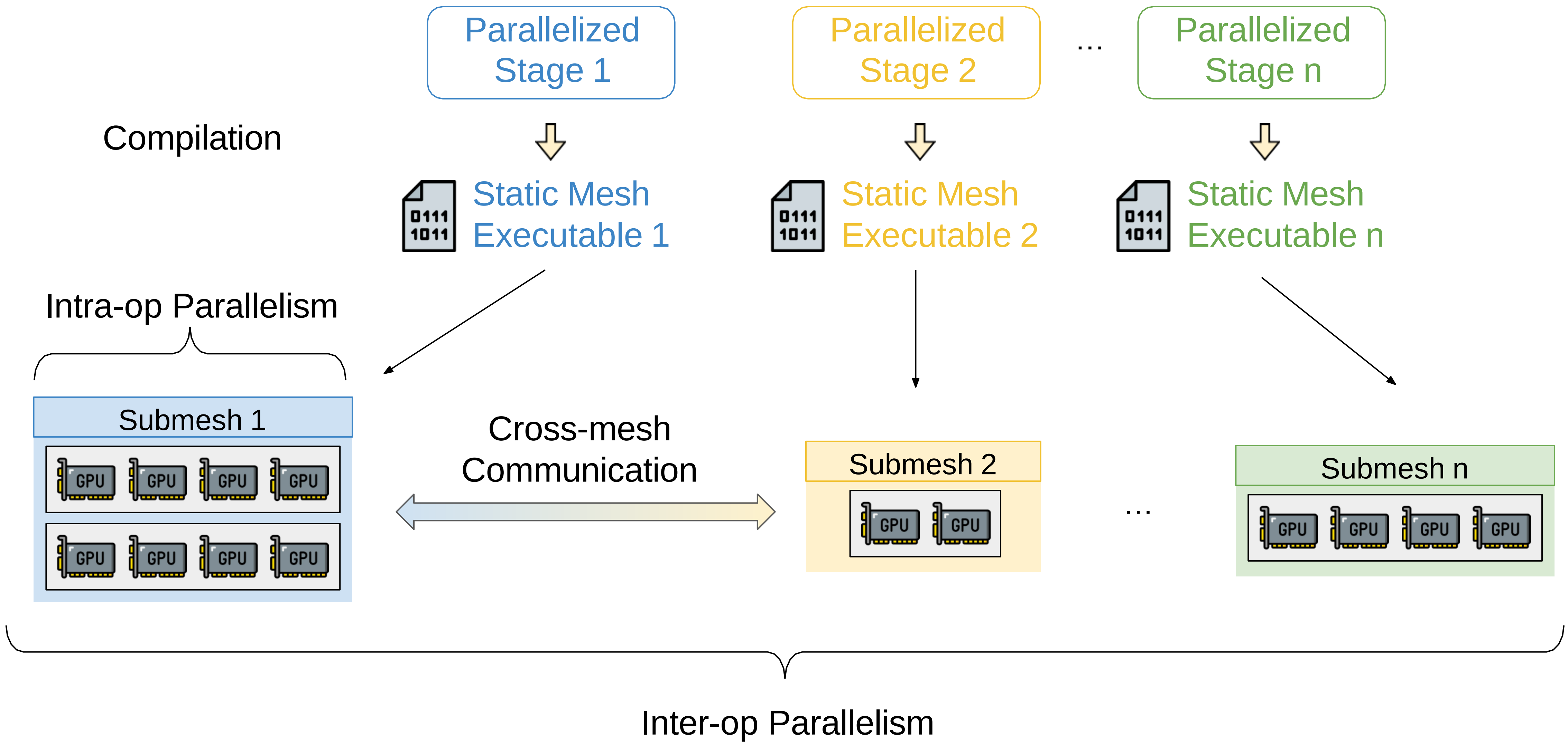
Early stopping in DP

Distributed Compilation

Alpa Compilation Time: < 40 min for the largest experiment.

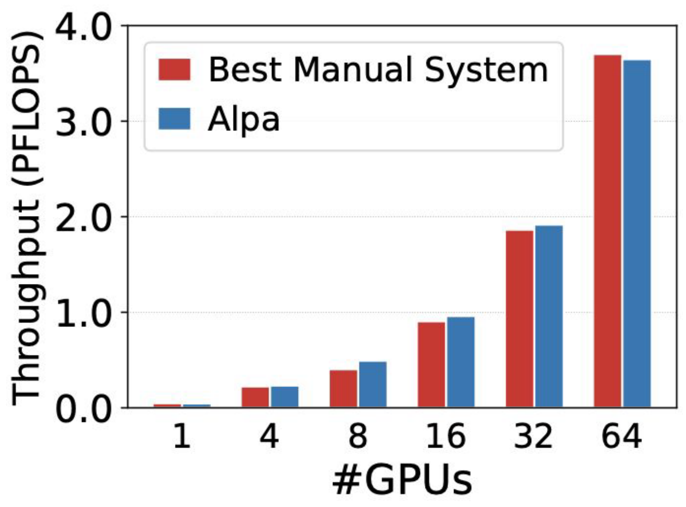
- Can be further reduced by at least 50% with search space pruning.

Runtime orchestration



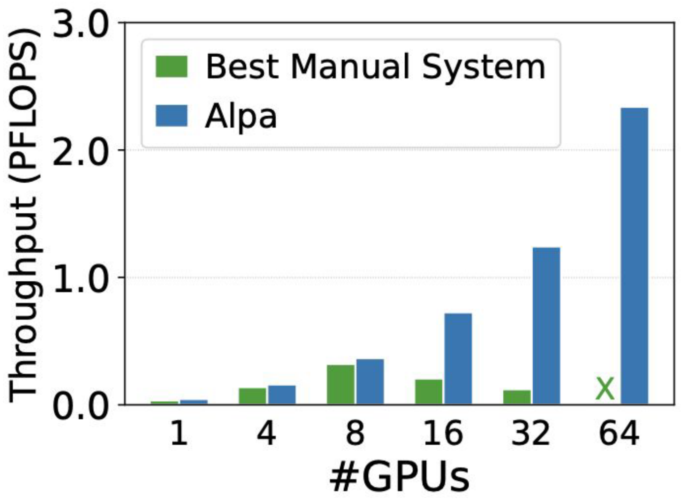
Evaluation of Alpa

GPT (up to 39B)



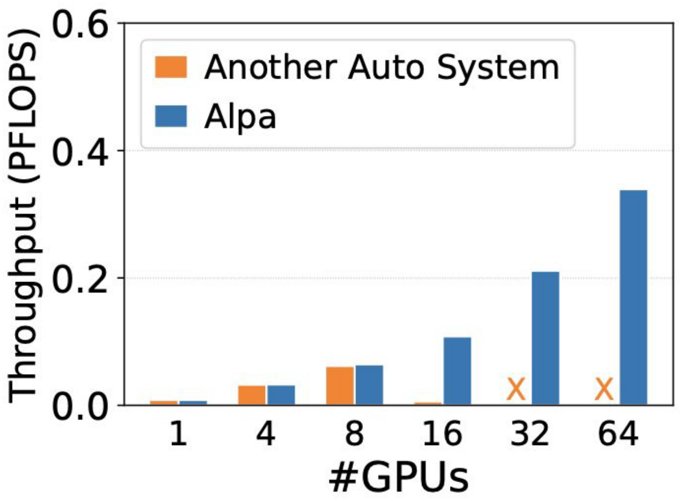
Match specialized manual systems.

GShard MoE (up to 70B)



Outperform the manual baseline by up to 8x.

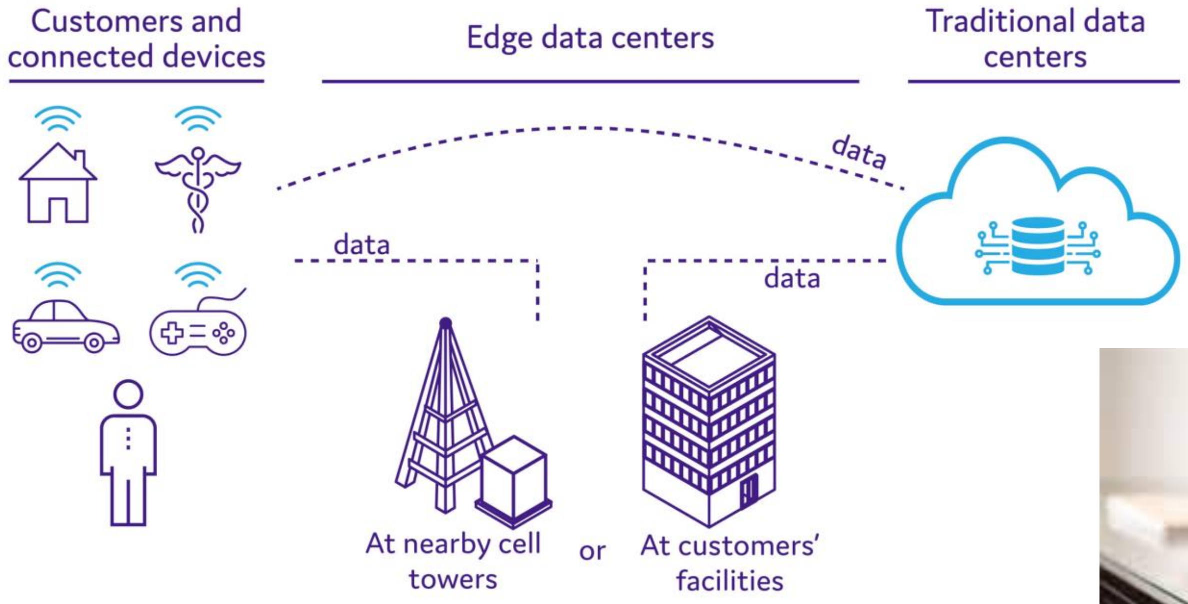
Wide-ResNet (up to 13B)



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).

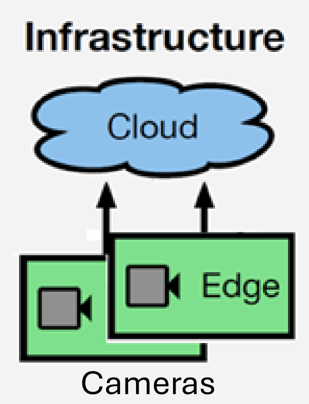
ML serving on heterogeneous (edge) infrastructures

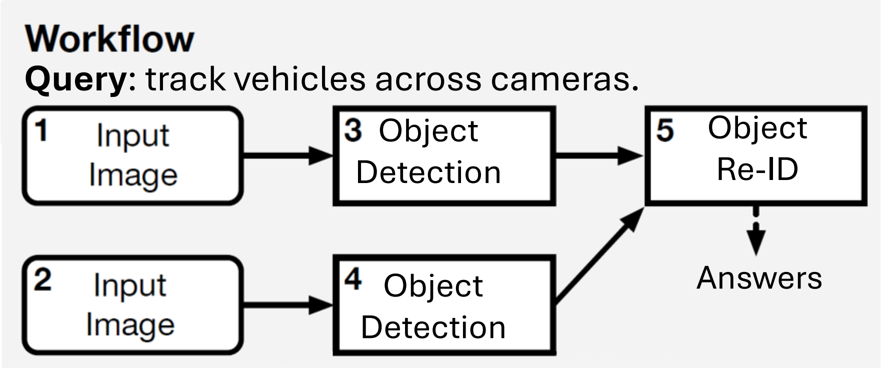


Data systems are growing into cloud + edge data centers.

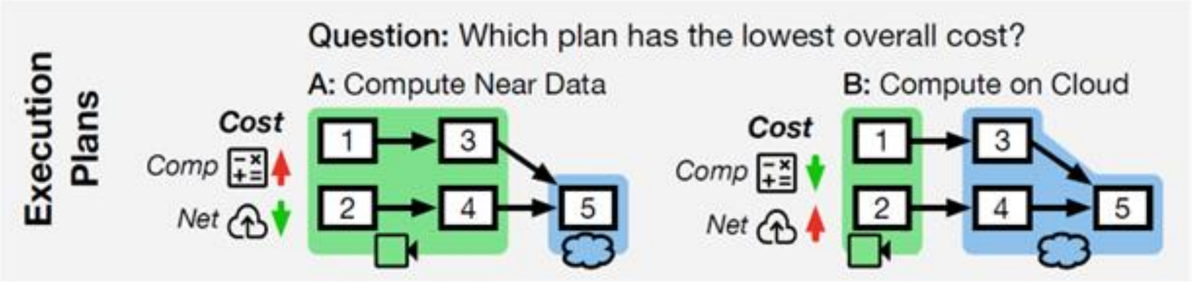
JellyBean: serving & optimizing ML workflows on hybrid cloud

- Maximize overall serving costs by solving:
  - model placement,
  - with estimated accuracy constraints.





- Prior IoT apps manually tune the plans.



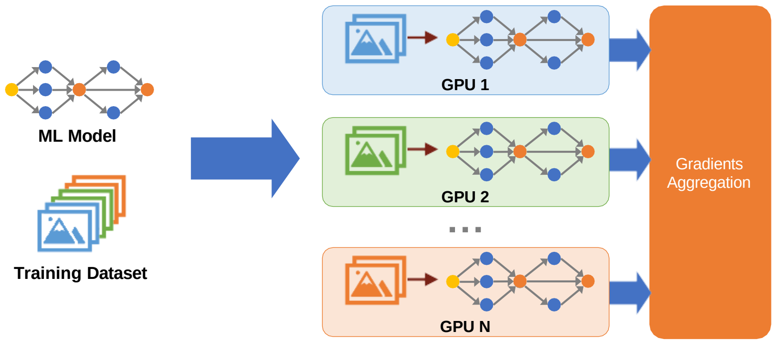
- Formulate as an optimization w/ a two-stage solver:
  - Model selection (beam search) + worker assignment (ILP).
  - Simplifying assumptions based on tiered infra & one-way data flow.

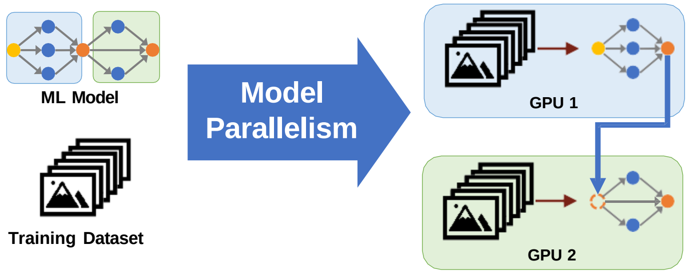
- Evaluation on [Nvidia AI city, Visual Question Answering] & different infra setups:

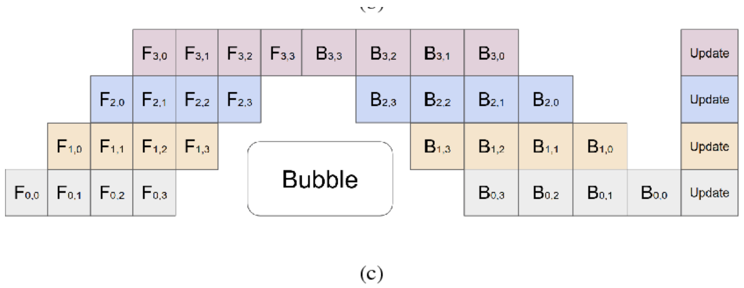
  At similar accuracy, improve serving costs by 30-60%.

Y. Wu, et al. Serving and Optimizing Machine Learning Workflows on Heterogeneous Infrastructures. VLDB 2023.

Summary: comparing different parallelisms

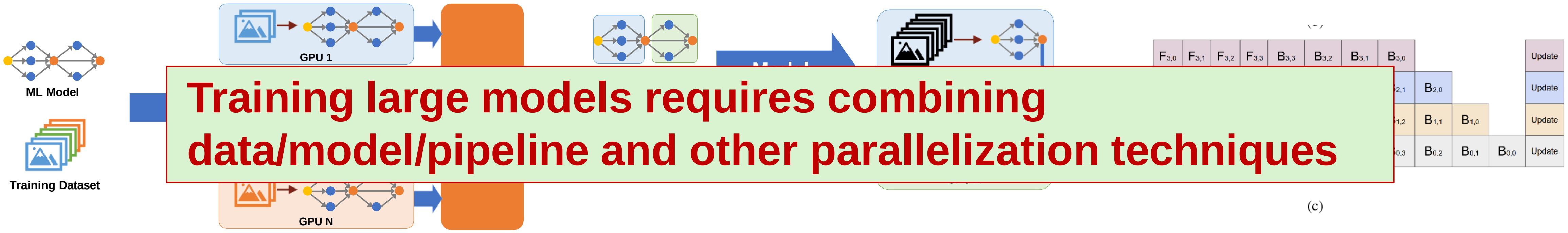






	Data parallelism	Tensor model parallelism	Pipeline model parallelism
Pros	✓ Massively parallelizable ✓ Require no communication during forward/backward	✓ Support training large models ✓ Efficient for models with large numbers of parameters	✓ Support large-batch training ✓ Efficient for deep models ✓ Dynamic cloud architecture
Cons	❖ Do not work for models that cannot fit on a GPU ❖ Do not scale for models with large numbers of parameters	❖ Limited parallelizability; cannot scale to large numbers of GPUs ❖ Need to transfer intermediate results in forward/backward	❖ Limited utilization: bubbles in forward/backward

Summary: comparing different parallelisms



Training large models requires combining data/model/pipeline and other parallelization techniques

	Data parallelism	Tensor model parallelism	Pipeline model parallelism
Pros	✓ Massively parallelizable ✓ Require no communication during forward/backward	✓ Support training large models ✓ Efficient for models with large numbers of parameters	✓ Support large-batch training ✓ Efficient for deep models ✓ Dynamic cloud architecture
Cons	❖ Do not work for models that cannot fit on a GPU ❖ Do not scale for models with large numbers of parameters	❖ Limited parallelizability; cannot scale to large numbers of GPUs ❖ Need to transfer intermediate results in forward/backward	❖ Limited utilization: bubbles in forward/backward