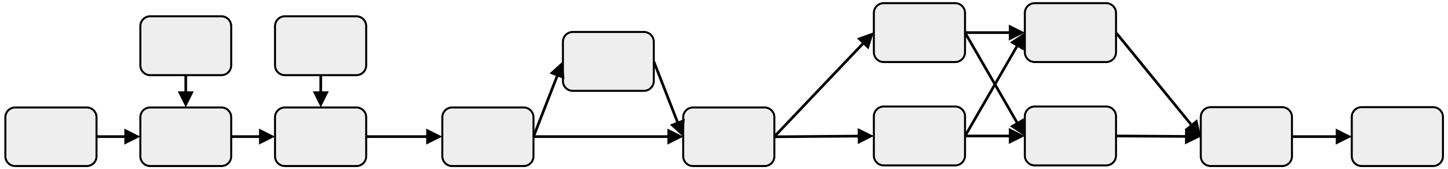
Where We Are

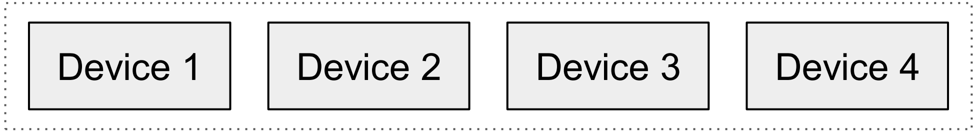
- Motivation
- History
- Parallelism Overview
- Data parallelism
- **Model parallelism**
  - Inter-op parallelism
  - Intra-op parallelism
- Auto-parallelization

Computational Graph (Neural Networks) → Stages

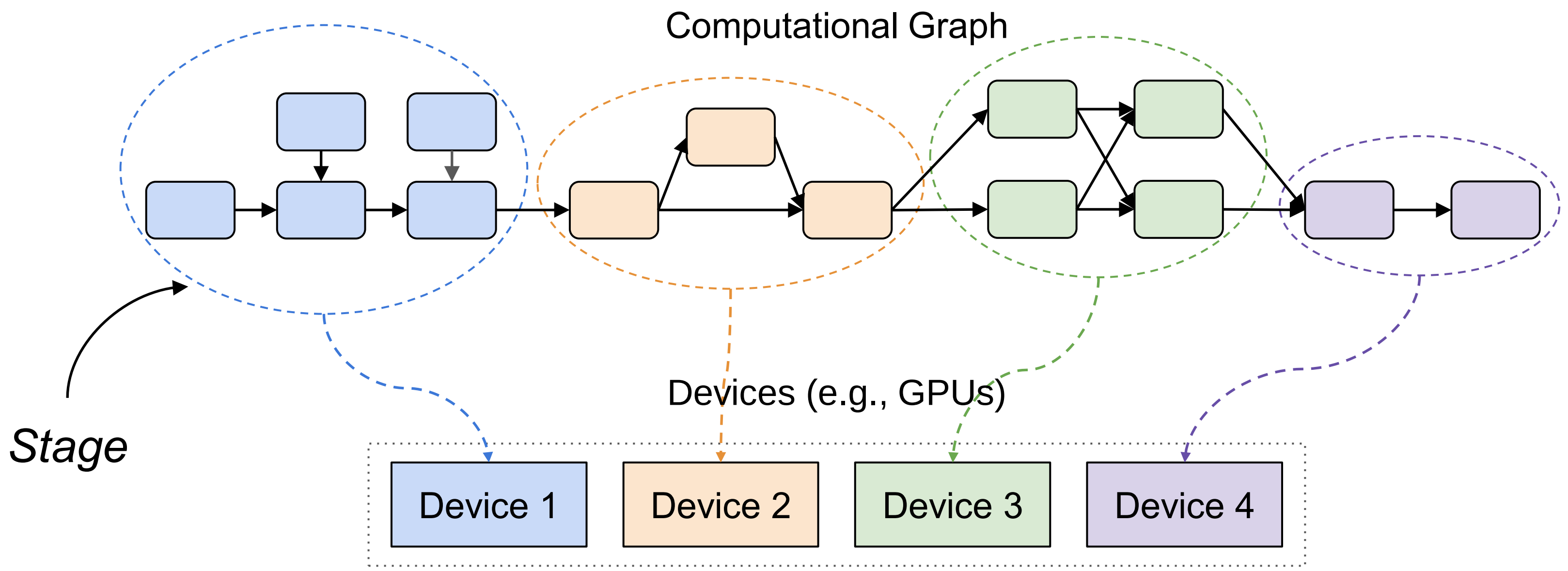
Computational Graph



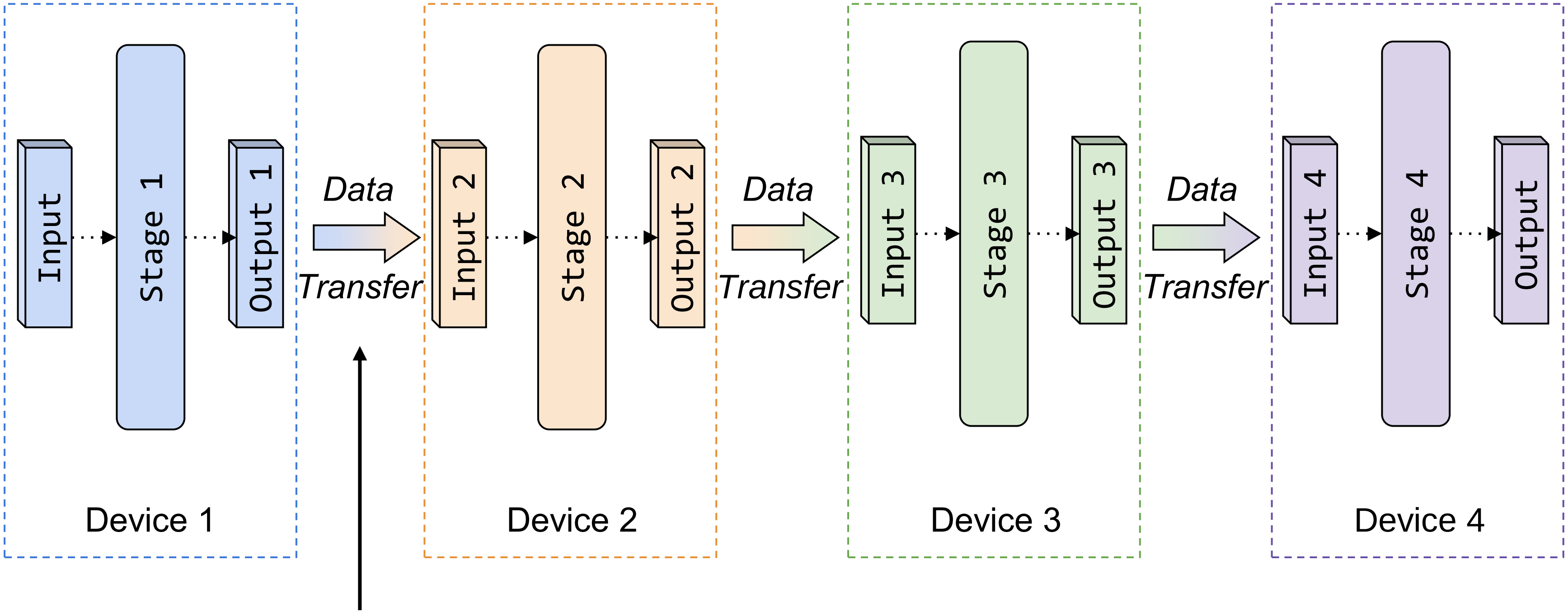
Devices (e.g., GPUs)



Computational Graph (Neural Networks) → Stages

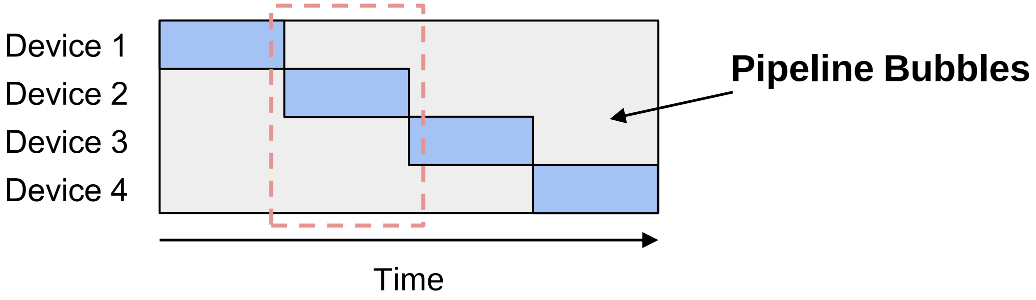


Execution & Data Movement



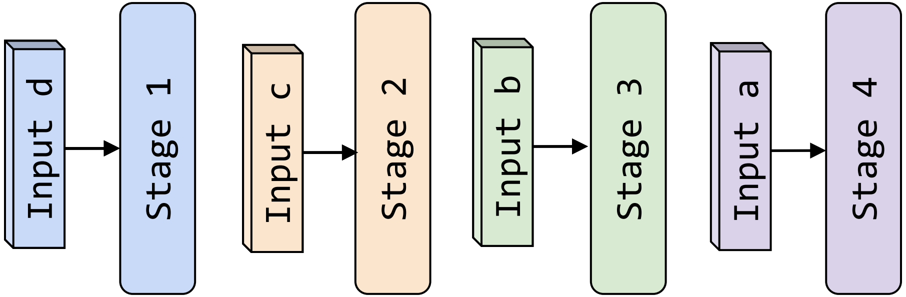
Note: The time spent on data transfer is typically **small,** since we only communicates stage outputs at stage boundaries between two stages.

Timeline: Visualization of Inter-Operator Parallelism

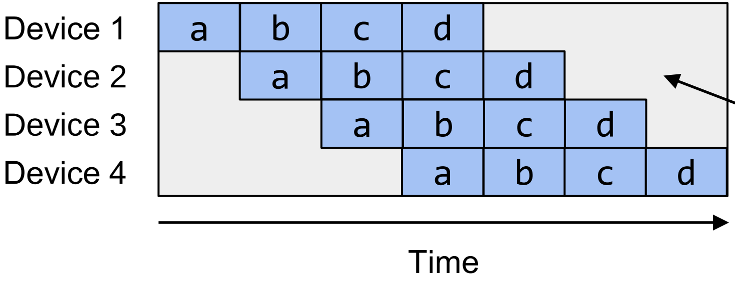


- Gray area ( indicates devices being idle (a.k.a. Pipeline bubbles).
- Only 1 device activated at a time.
- **Pipeline bubble percentage** = `bubble_area / total_area`
  = `(D - 1) / D`, assuming `D` devices.

Reduce Pipeline Bubbles via Pipelining Inputs

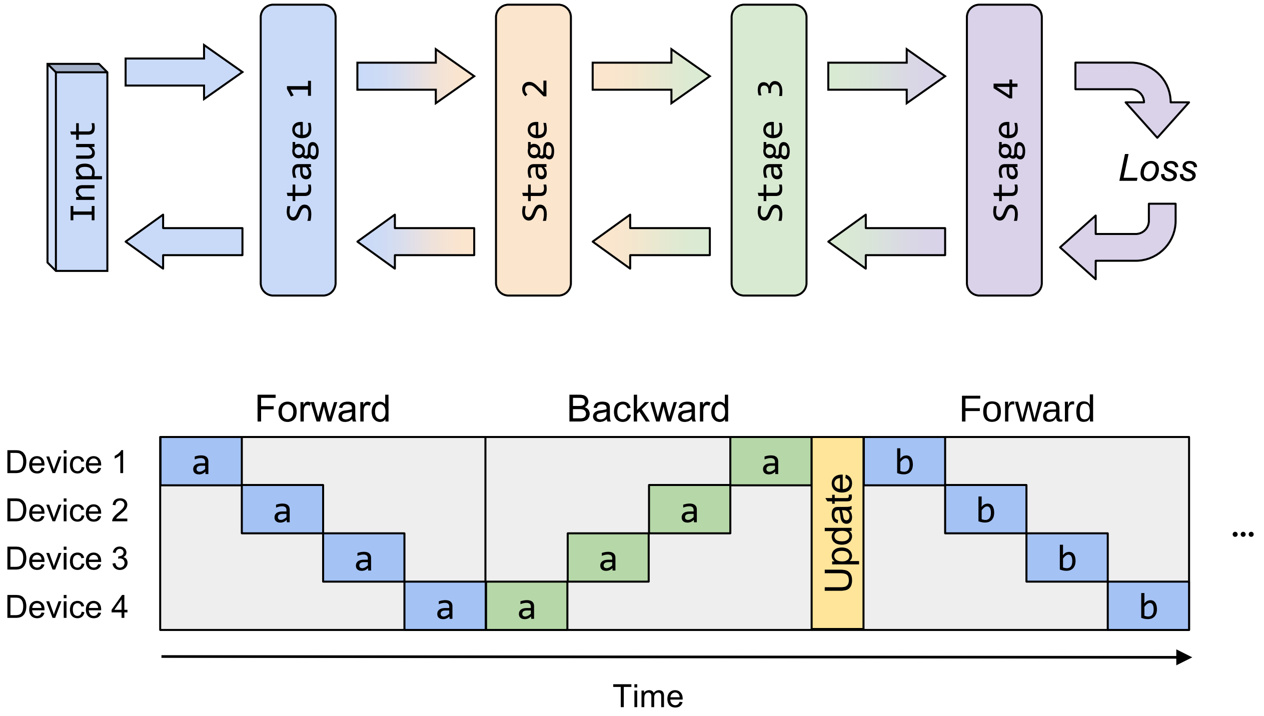


Used in inference.



Pipeline bubbles percentage
`= (D - 1) / (D - 1 + N)`
with `D` devices and `N` inputs.

Training: Forward & Backward Dependency

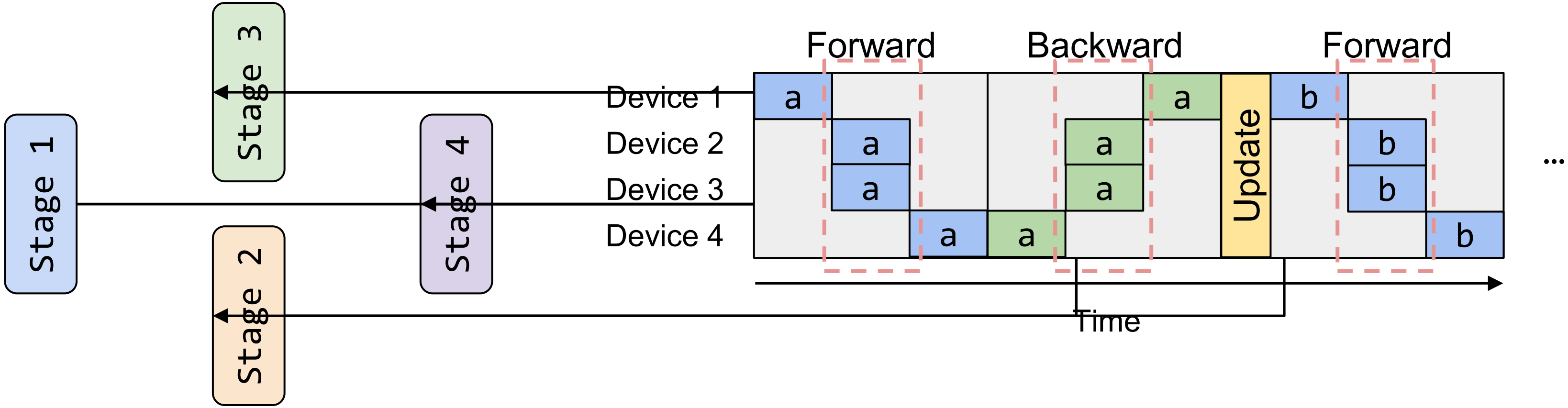


How to Reduce Pipeline Bubbles for Training?

- Device Placement
- Synchronous Pipeline Parallel Algorithms
  - GPipe
  - 1F1B
  - Interleaved 1F1B
  - TeraPipe
  - Chimera
- Asynchronous Pipeline Parallel Algorithms
  - AMPNet
  - Pipedream/Pipedream-2BW

Device Placement

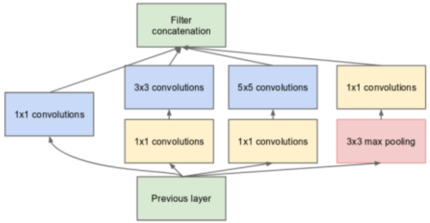
Idea: Slice the branches of a neural network into multiple stages so they can be calculated concurrently.



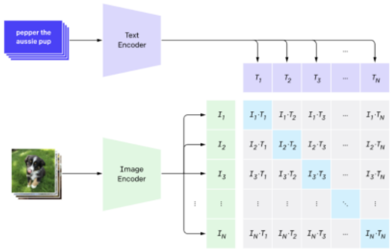
Mirhoseini, Azalia, et al. "Device placement optimization with reinforcement learning." *ICML 2017.*

Device Placement: Limitations

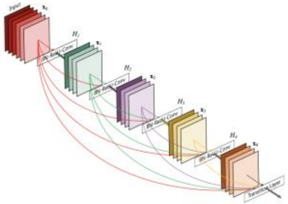
Only works for specific NNs with branches:



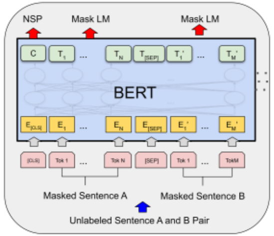
✅ Inception Module



✅ Contrastive Model

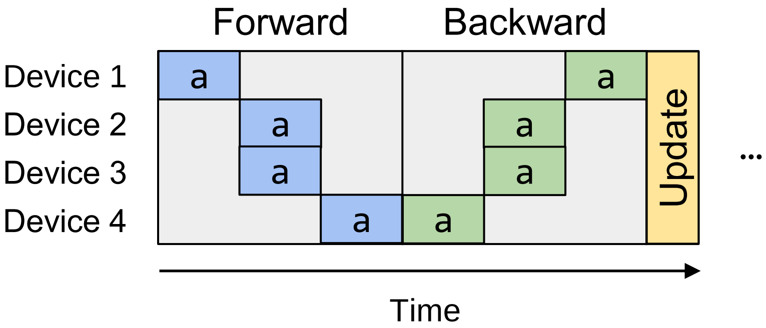


❌ Other ConvNets



❌ Transformers

Device Utilization is still low:



Note: device placement needs to be combined with the other pipeline schedules discussed later to further improve device utilization.

Synchronous Pipeline Parallel Schedule

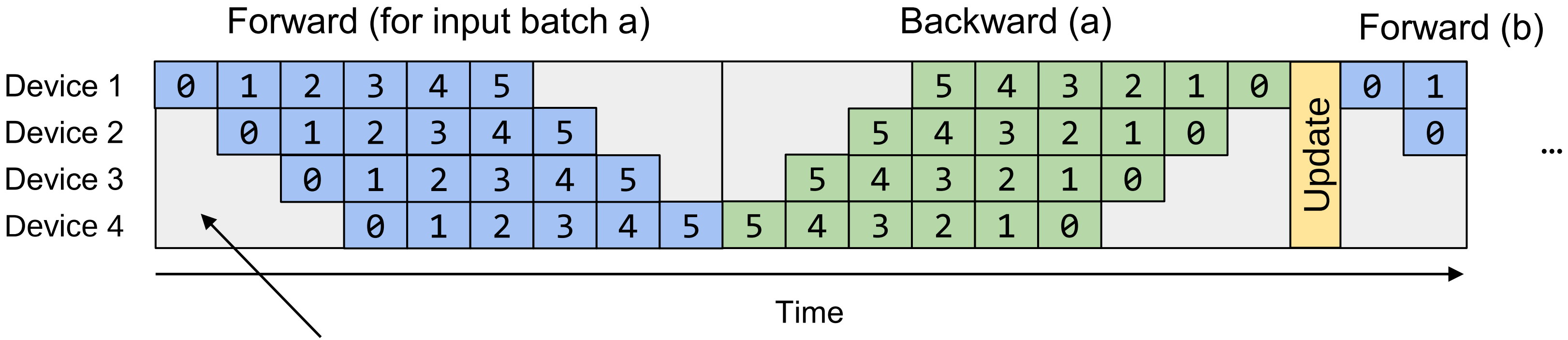
Idea: Modify pipeline schedule to improve efficiency, but keep the computation and convergence semantics exactly the same as if training with a single device.

GPipe

Idea: Partition the input batch into multiple "*micro-batches*". Pipeline the micro-batches. Accumulate the gradients of the micro-batches:

$$\nabla L_\theta(x) = \frac{1}{N} \sum_{i=1}^{N} \nabla L_\theta(x_i)$$

Example: Slice each input batch into 6 micro-batches:



Pipeline bubbles percentage `= (D - 1) / (D - 1 + N)` with `D` devices and `N` micro-batches.

Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." *NeurIPS 2019.*

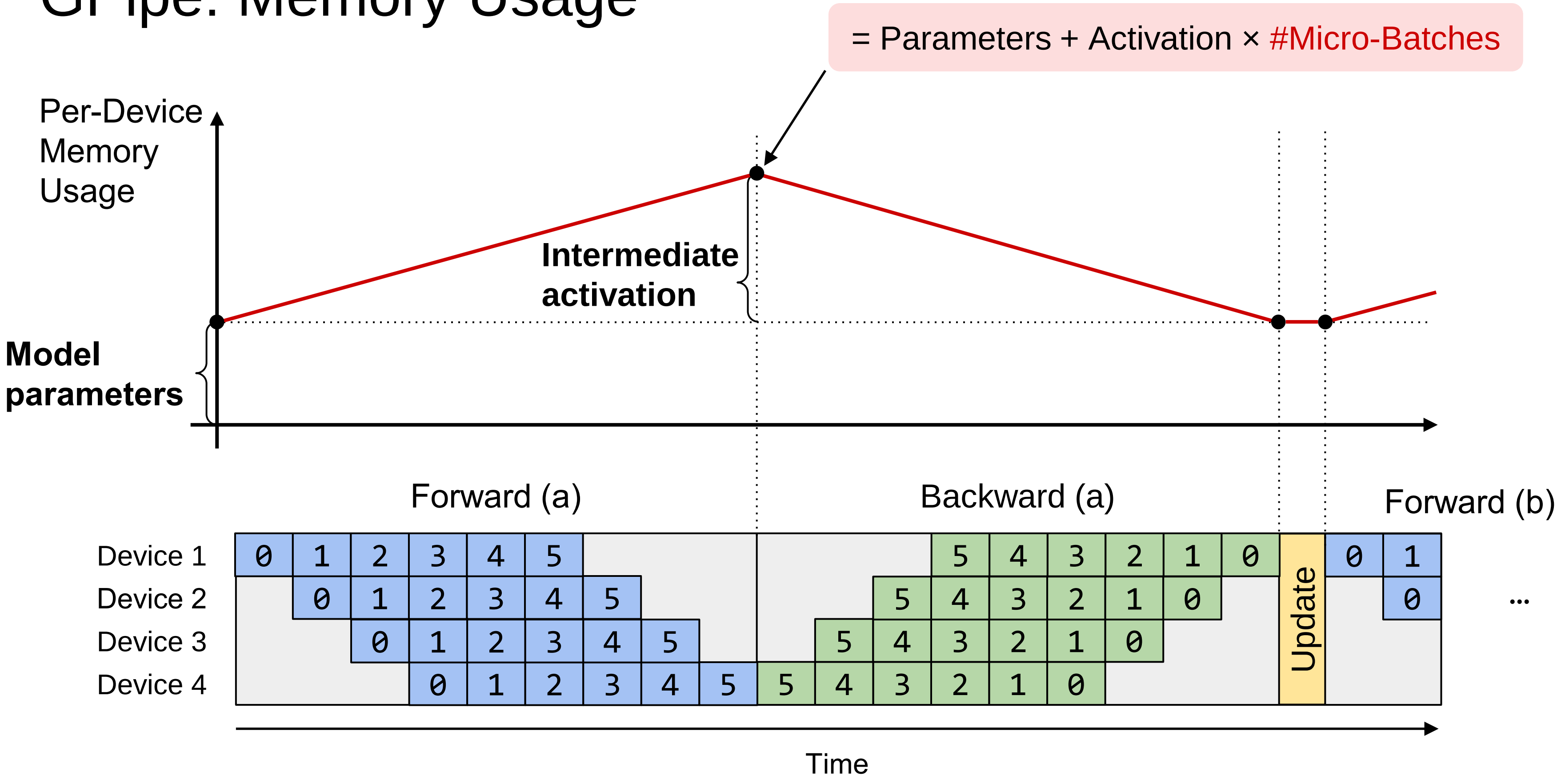
GPipe: Experimental Results

Table: Normalized training throughput using GPipe with different number of devices (stages) and different number of micro-batches M on TPUs.

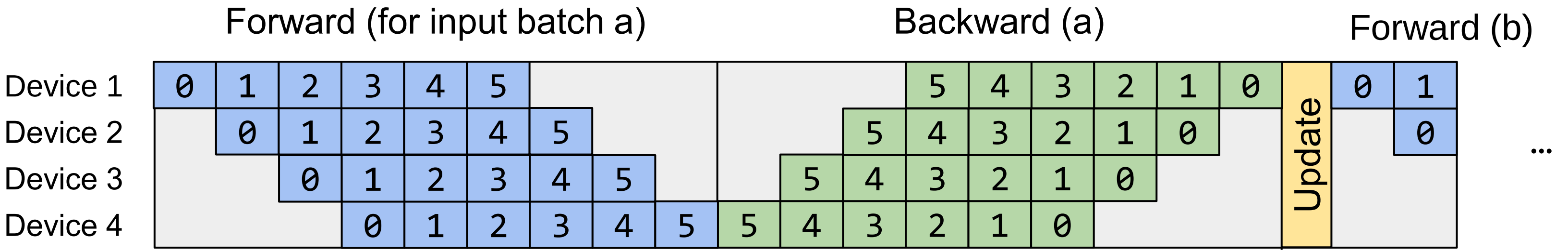
	#TPUs = 2	**#TPUs = 4**	**#TPUs = 8**
#Micro-batches = 1	1	1.07	1.3
#Micro-batches = 4	1.7	3.2	4.8
#Micro-batches = 32	1.8	3.4	6.3

Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." *NeurIPS 2019.*

GPipe: Memory Usage

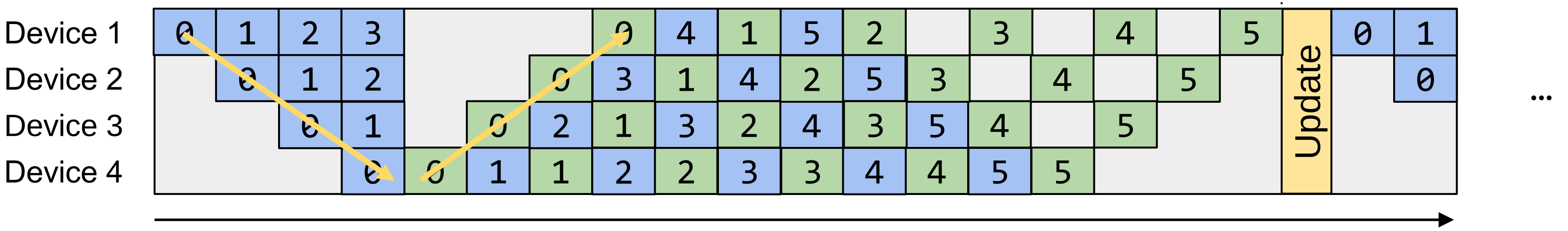


GPipe Schedule:



1F1B (1 Forward 1 Backward) Schedule:

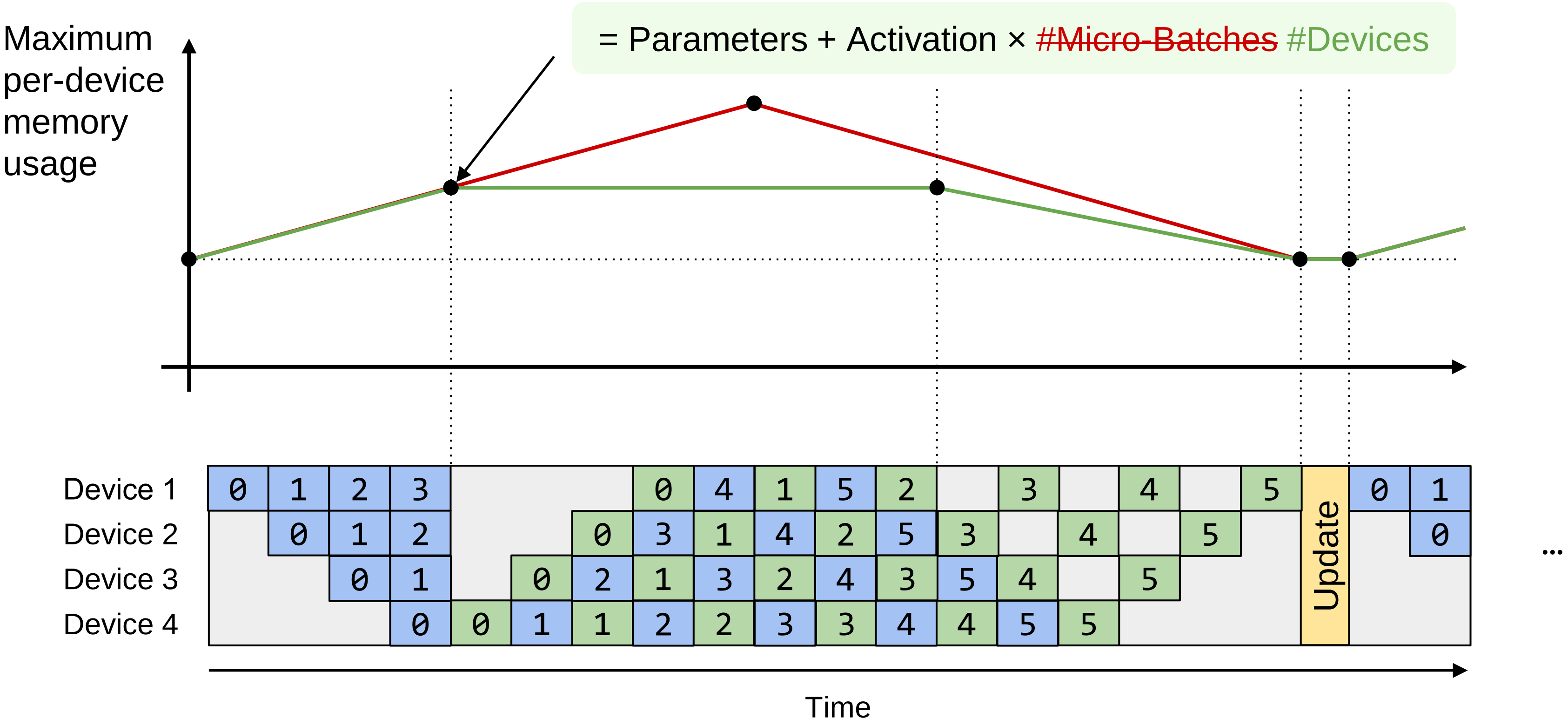
Same Latency



Perform backward as early as possible

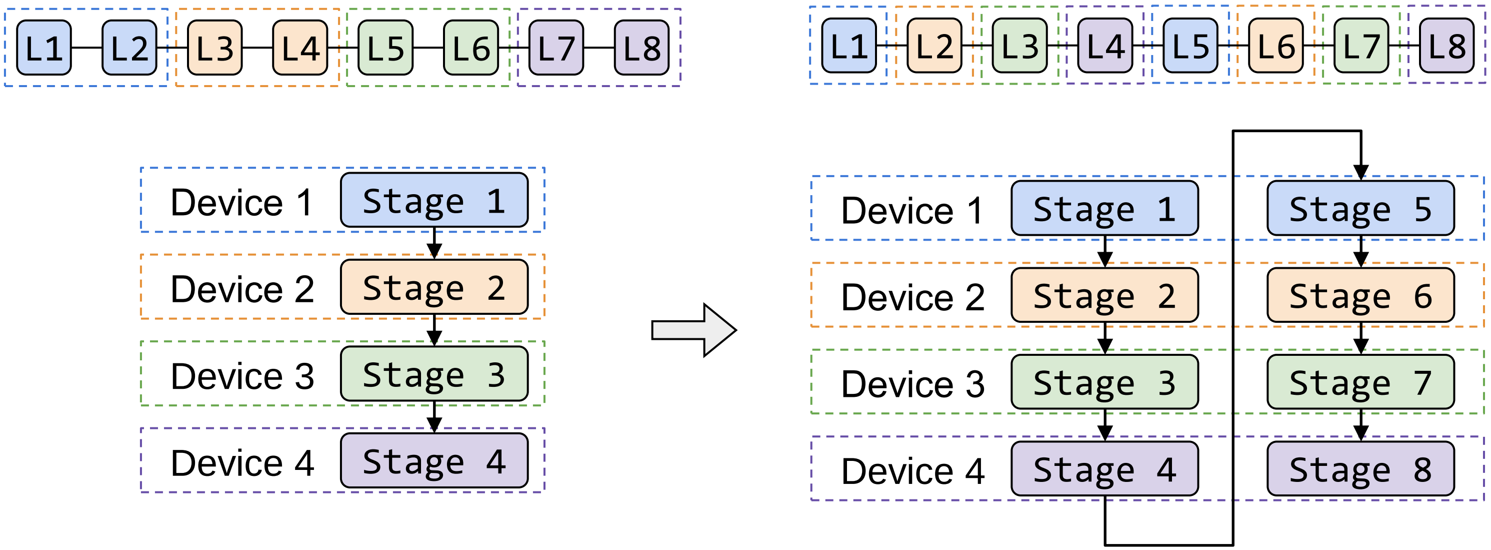
Fan, Shiqing, et al. "DAPPLE: A pipelined data parallel approach for training large models." *PPoPP 2021.*

1F1B Memory Usage



Interleaved 1F1B

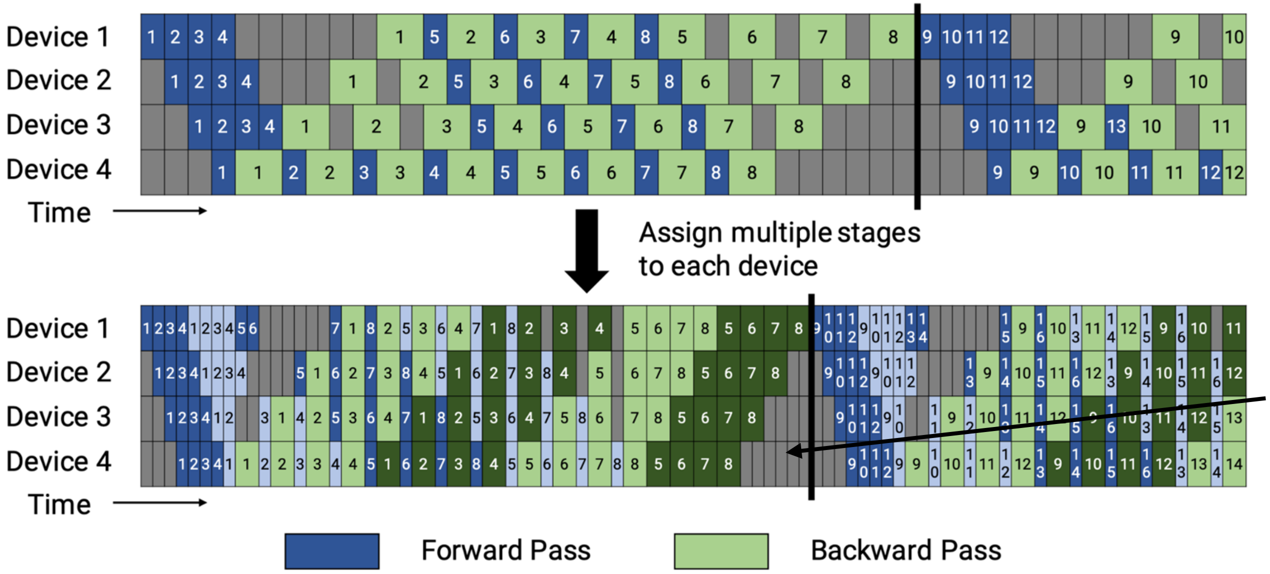
Idea: Slice the neural network into more fine-grained stages and assign multiple stages to reduce pipeline bubble.



Interleaved 1F1B

Pro:
Higher pipeline efficiency with fewer pipeline bubbles.

Con:
More communication overhead between stages.

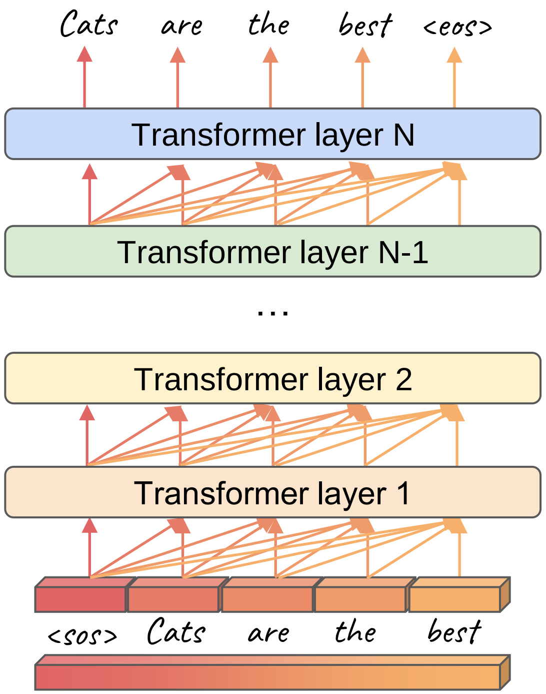


Pipeline bubbles percentage $= (D - 1) / (D - 1 + KN)$ with D devices, K stages on each device, and N micro-batches.

TeraPipe

Idea: The computation of an input token only depends on previous tokens but not future tokens for autoregressive models.

Further reduce the bubble size by pipelining within a sequence.

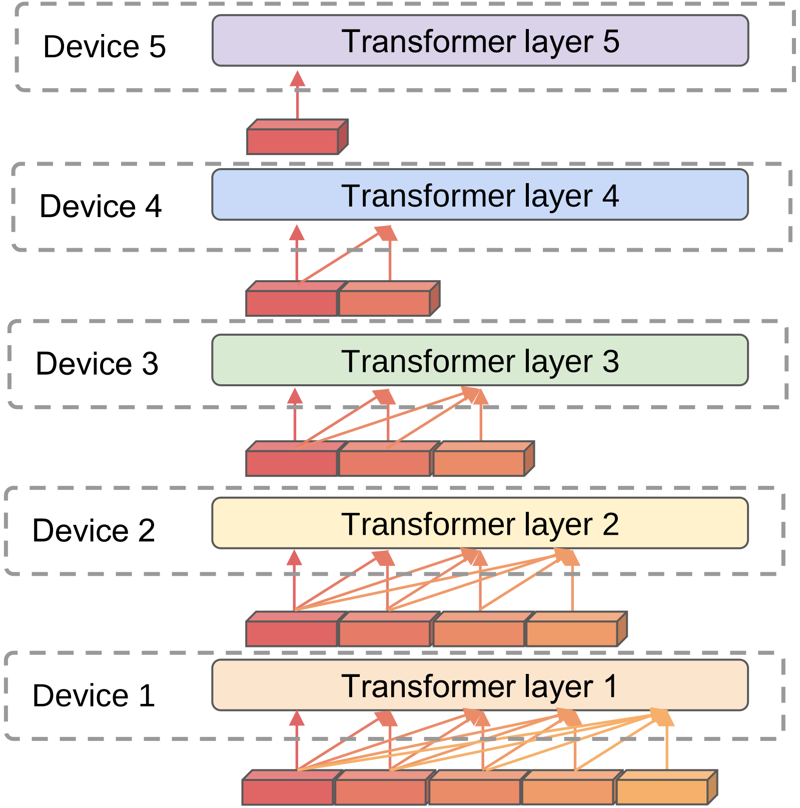


Li, Zhuohan, et al. "TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models." *ICML 2021.*

TeraPipe

Idea: The computation of an input token only depends on previous tokens but not future tokens for autoregressive models.

Further reduce the bubble size by pipelining within a sequence.

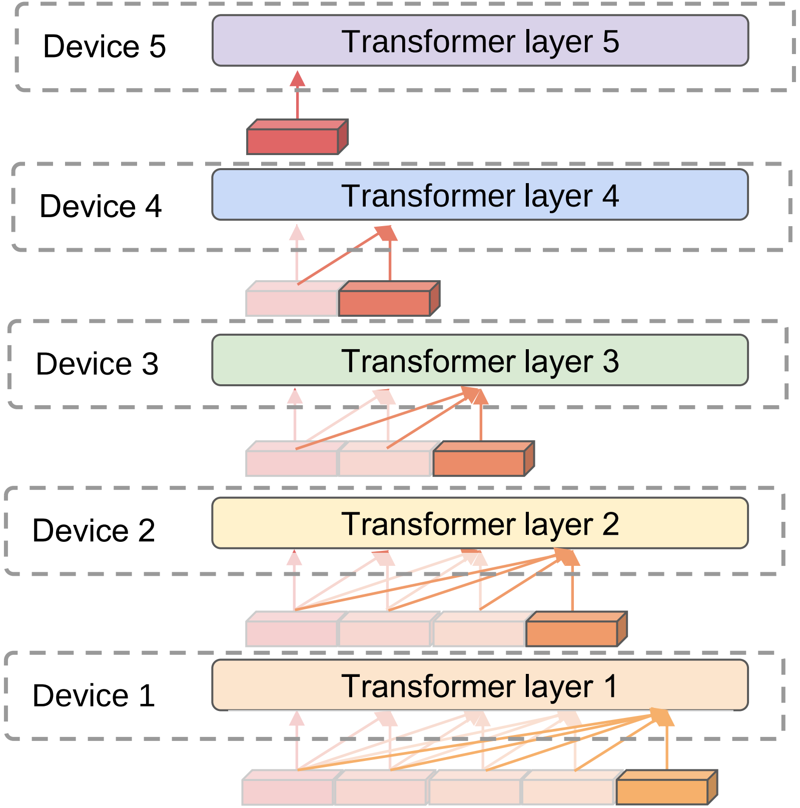


Li, Zhuohan, et al. "TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models." *ICML 2021.*

TeraPipe

Idea: The computation of an input token only depends on previous tokens but not future tokens for autoregressive models.

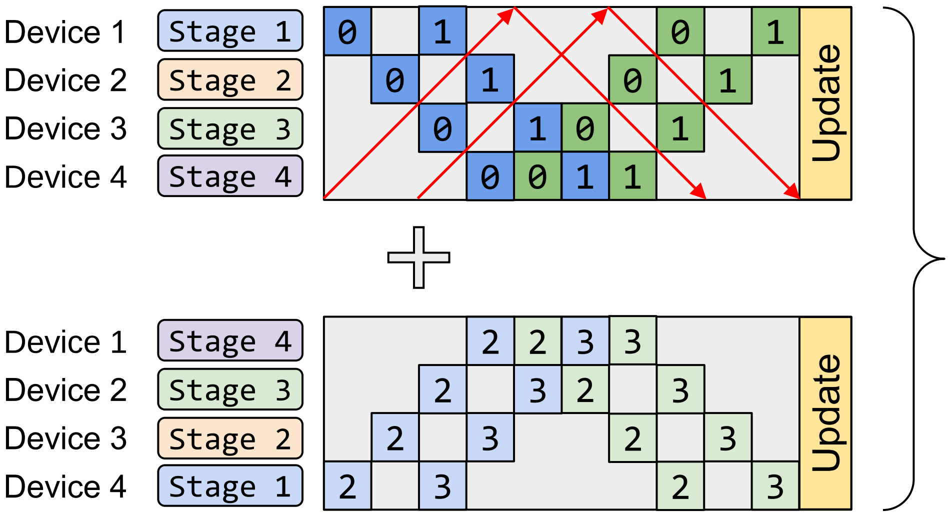
Further reduce the bubble size by pipelining within a sequence.

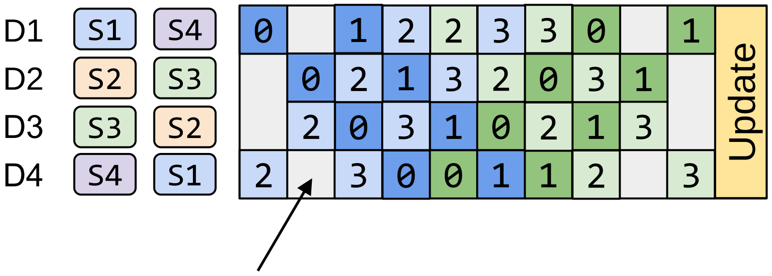


Li, Zhuohan, et al. "TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models." *ICML 2021.*

Chimera

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





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.

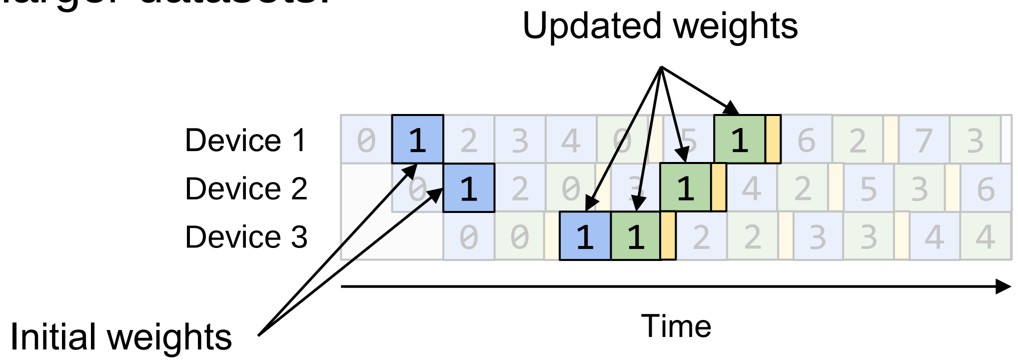
❌ **Cons:**

- 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

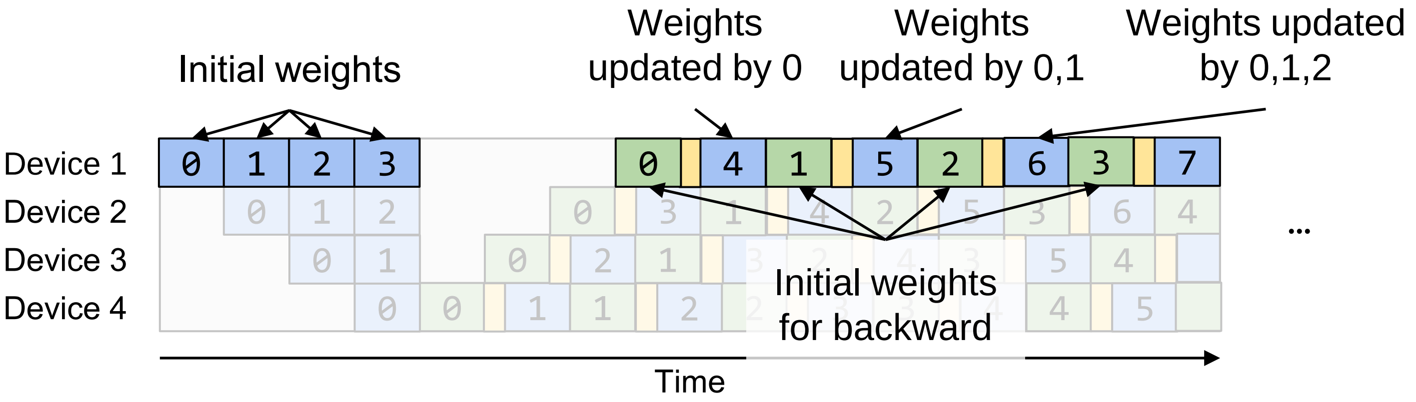
Gaunt, Alexander L., et al. "AMPNet: Asynchronous model-parallel training for dynamic neural networks." *arXiv 2017.*
Yang, Bowen, et al. "Pipemare: Asynchronous pipeline parallel dnn training." *MLSys 2021.*

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.

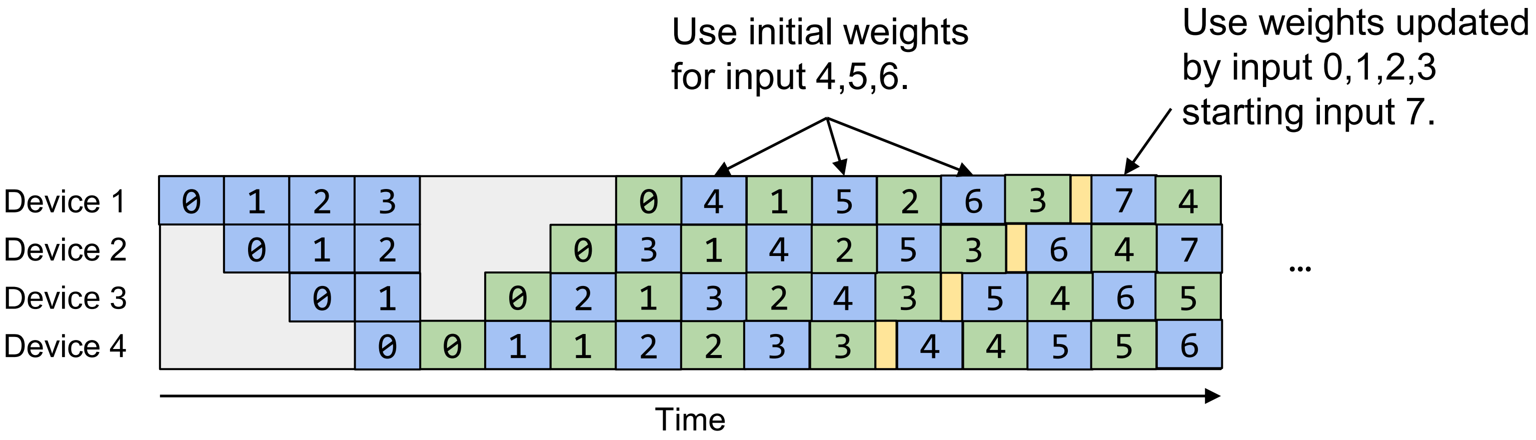


Narayanan, Deepak, et al. "PipeDream: generalized pipeline parallelism for DNN training." *SOSP 2019.*

Pipedream-2BW

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

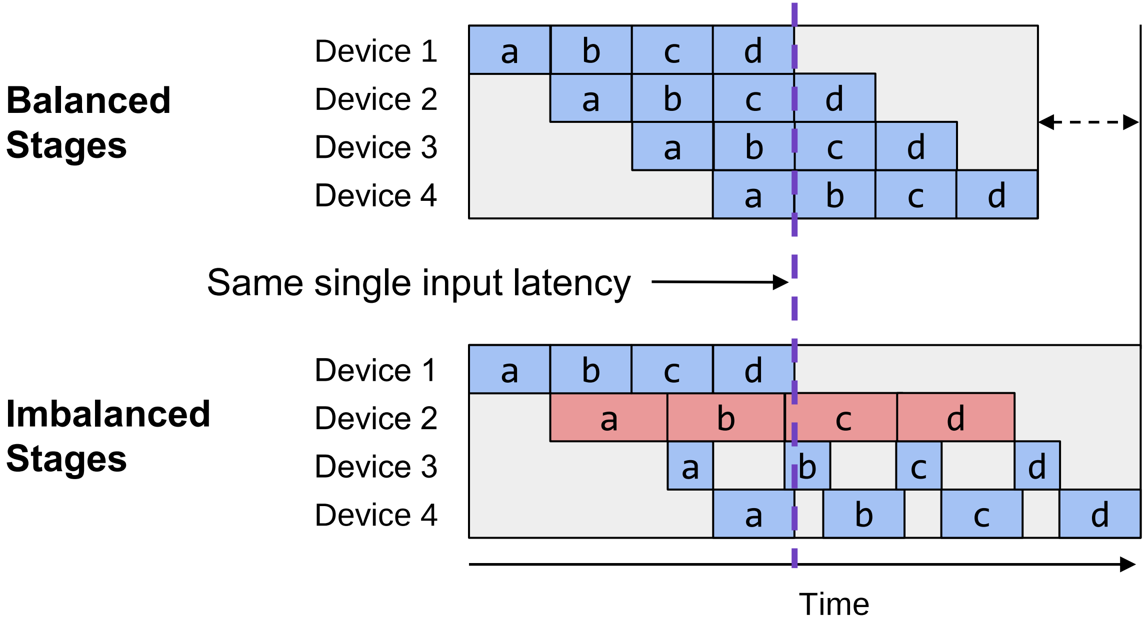
Convergence: Similar training accuracy on language models (BERT/GPT)



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. *…*

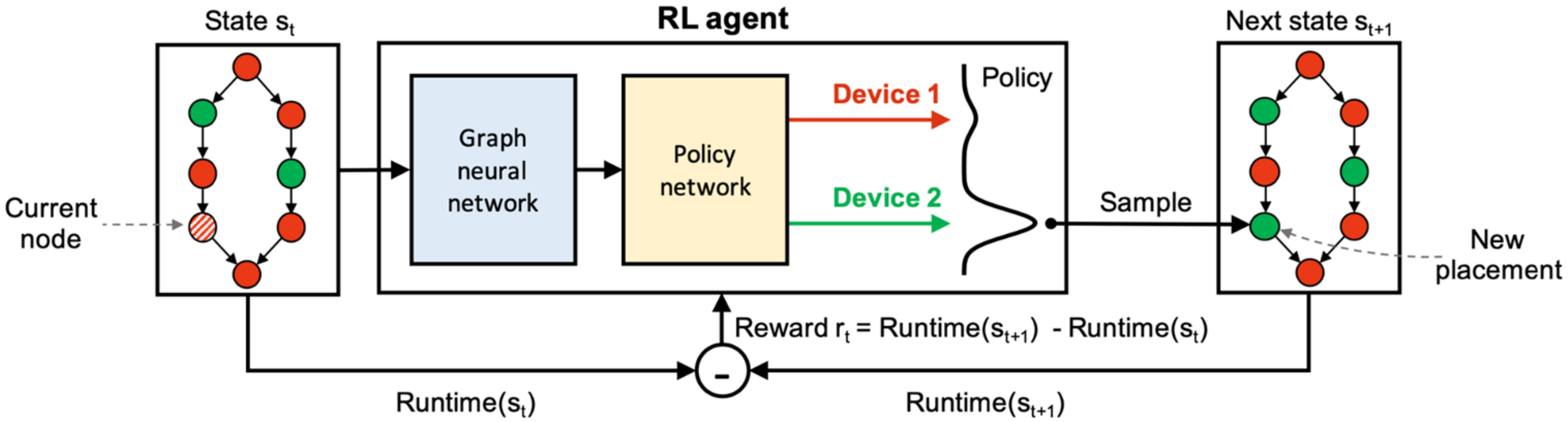
RL-Based Partitioning Algorithm

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.

Optimization-Based Partitioning Algorithm

Integer Linear Programming:

Variable: Decision variable vector for each operator, representing device assignment.

Minimize: Maximum finishing time of all operators.

Constraint: Execution dependency & memory capacity of each device.

$$
\begin{aligned}
\min \quad & \text{TotalLatency} \\
\text{s.t.} \quad & \sum_{i=0}^{k} x_{vi} = 1 \\
& \text{subgraph } \{v \in V : x_{vi} = 1\} \text{ is contiguous} \\
& M \geq \sum_{v} m_v \cdot x_{vi} \\
& \text{CommIn}_{ui} \geq x_{vi} - x_{ui} \\
& \text{CommOut}_{ui} \geq x_{ui} - x_{vi} \\
& \text{TotalLatency} \geq \text{Latency}_v \\
& \text{SubgraphStart}_i \geq \text{Latency}_v \cdot \text{CommIn}_{vi} \\
& \text{SubgraphFinish}_i = \text{SubgraphStart}_i + \sum_{v} \text{CommIn}_{vi} \cdot c_v \\
& \qquad + \sum_{v} x_{vi} \cdot p_v^{\text{acc}} + \sum_{v} \text{CommOut}_{vi} \cdot c_v \\
& \text{Latency}_v \geq x_{v0} \cdot p_v^{\text{cpu}} \\
& \text{Latency}_v \geq x_{v0} \cdot p_v^{\text{cpu}} + \text{Latency}_u \\
& \text{Latency}_v \geq x_{vi} \cdot \text{SubgraphFinish}_i \\
& x_{vi} \in \{0, 1\}
\end{aligned}
$$

Tarnawski, Jakub M., et al. "Efficient algorithms for device placement of dnn graph operators." NeurIPS 2020.

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

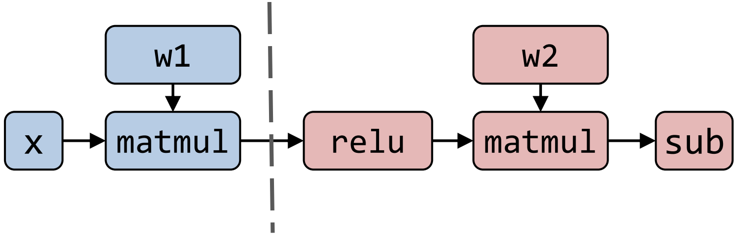
RL-Based / **Optimization-Based** Automatic Stage Partitioning

Where We Are

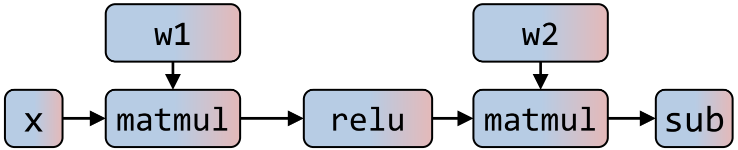
- Motivation
- History
- Parallelism Overview
- Data parallelism
- **Model parallelism**
  - Inter-op parallelism
  - Intra-op parallelism
- Auto-parallelization

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

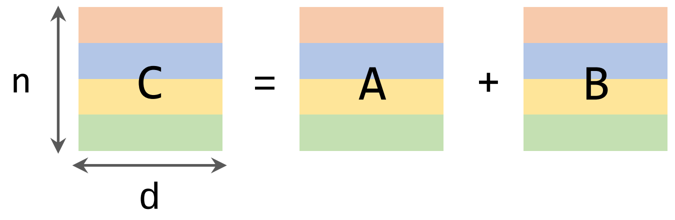
Element-wise operators

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

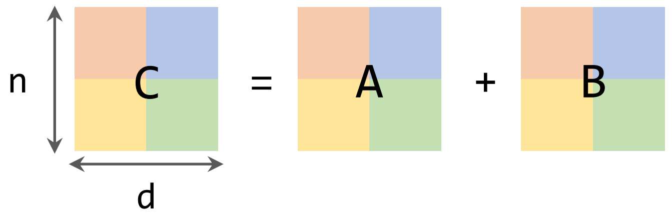
No dependency on the two for-loops.
Can arbitrarily split the for-loops on different devices.

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

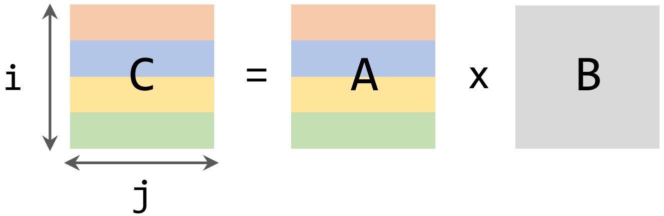
No dependency on the two spatial for-loops.
Can arbitrarily split the for-loops on different devices.

```
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

device 1 device 2 device 3 device 4 replicated

Parallelize loop i



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

Parallelize One Operator

Matrix multiplication

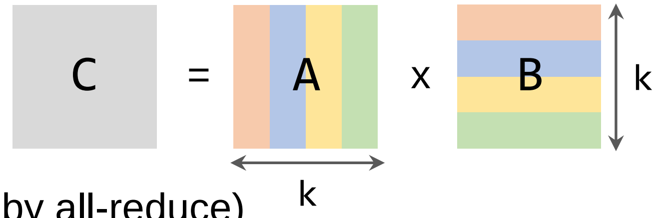
No dependency on the two spatial for-loops.
Can arbitrarily split the for-loops on different devices.

```
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

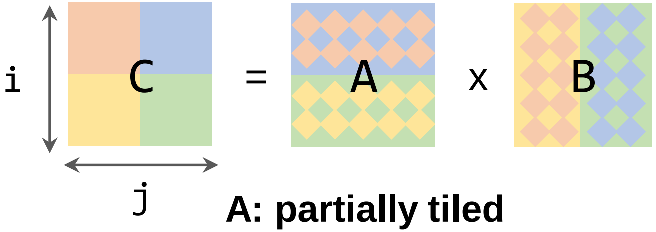
```
for i in range(0, N):
  for j in range(0, M):
    for k in range(0, K):
      C[i,j] = C[i,j] + A[i,k] x B[k,j]
```

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

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

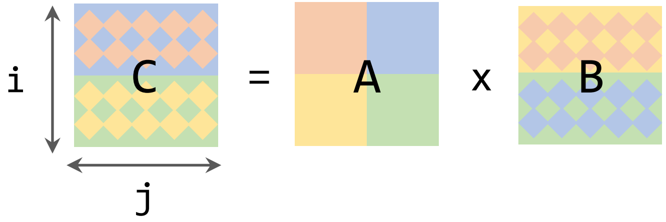


A: partially tiled

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