Alpa : Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning

Lianmin Zheng*, Zhuohan Li*, and Hao Zhang*, UC Berkeley; Yonghao Zhuang, Shanghai Jiao Tong University; Zhifeng Chen and Yanping Huang, Google; Yida Wang, Amazon Web Services; Yuanzhong Xu, Google; Danyang Zhuo, Duke University; Eric P. Xing, MBZUAI and Carnegie Mellon University; Joseph E. Gonzalez and Ion Stoica, UC Berkeley

Presented by Khoa Pham and Julian Yu

Background - Parallel Training

Parallel Training can boost the training of large-scale model

Many parallelism strategy have been proposed: DP, PP, TP, ZeRO, etc.

Some works try to **combine different parallelism**: Megatron-LM, etc.

- Most of them heavily rely on **manual tuning** and requires system expert experiences

Background - Google Training Stack

- XLA
  - ML Compiler that can take models written in TF, PyTorch, Jax and optimizes them for high-performance execution across GPUs, TPUs, Trainium, …
- GSPMD
  - Implements at XLA-level, can infer tensor sharding configuration based on users' annotations.
    - `mesh_split(tensor, device_mesh, dims_mapping)`
  - GSPDM automatically generate parallel instructions and insert communication collective.
  - Natively support intra-op parallelism.
  - Alpa intra-op sharding spec take inspiration from and build heavily on it. See more later!

Target & Challenges

Target: **Auto-parallelization**

It can significantly accelerate ML research by freeing developers from struggling with underlying system challenges

Main challenge: It requires navigating **a complex space of plans** that **grows exponentially** with the dimensions of parallelism and the size of the model and cluster:

1. how many data-parallel **replicas**
2. which **axis** to be partitioned
3. how to split the model into pipeline **stages**
4. how to **map** devices to the resulting parallel executables

Target & Challenges (Cont.)

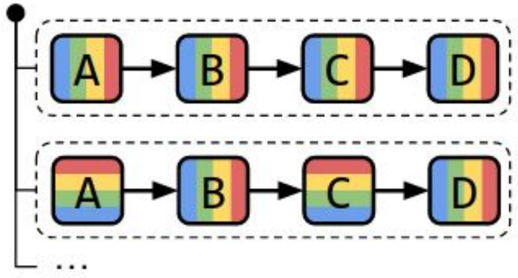
Existing Works For Auto-Parallelization:

1. **Dapple:** only for DP + PP
2. **PipeDream:** only for PP
3. **Autosync:** only for DP
4. **Tofu:** only support single node, no PP
5. **FlexFlow:** randomized search, can’t find optimal/near-optimal plan

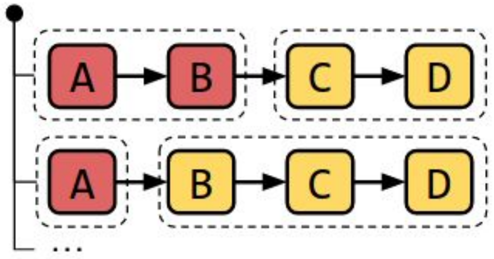
Design Overview - Recategorizing Parallelism

Re-categorizing parallelism as **intra-operator** and **inter-operator**:

1. **Intra-op**: data/operator parallelism
    a. Higher utilization
    b. Higher communication volume
    c. Fit devices with faster network connectivity



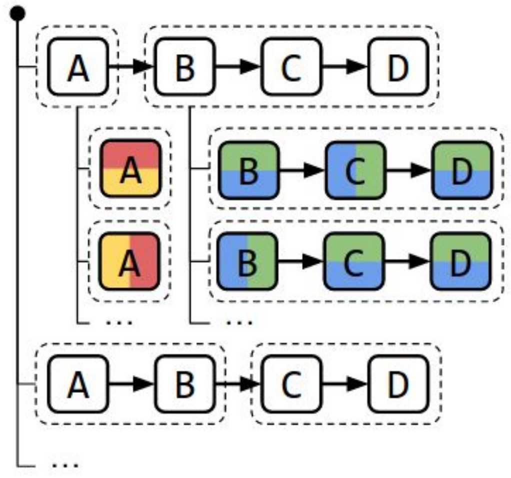
2. **Inter-op**: pipeline parallelism
    a. Lower communication volume
    b. Idle time
    c. Fit devices with slower network connectivity



Design Overview - Problem Formulation

Hierarchically optimizing the parallel plan **at two levels**: intra-op and inter-op.

```
total cost = inter-op cost + intra-op cost
```

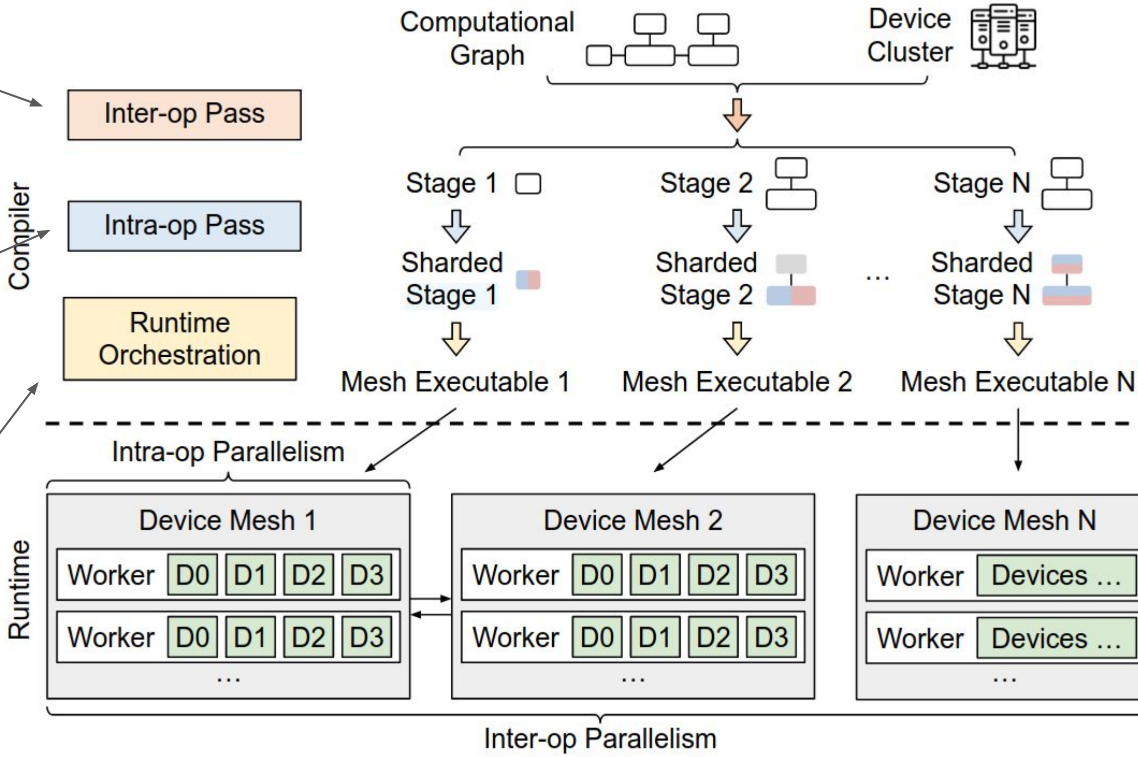


Design Overview - Compilation Passes

- **Partition** graph and cluster into disjoint stages
- **Optimize** total cost
- **Invoke intra-op pass** to query the exec cost of this stage

- **Optimize intra-op** parallel exec plan on assigned mesh
- **Report** the cost back to inter-op pass

Fulfill the **communication requirement** between two adjacent stages



Design Overview - API

Annotate `train_step()` by `@parallelize`

Upon the first call to `train_step()`:

1. Traces the whole function to get the **model IR**
2. Invokes the **compilation passes** to converts the function to a optimized parallel version

```python
# Put @parallelize decorator on top of the Jax functions
@parallelize
def train_step(state, batch):
    def loss_func(params):
        out = state.forward(params, batch["x"])
        return jax.numpy.mean((out - batch["y"]) ** 2)

    grads = grad(loss_func)(state.params)
    new_state = state.apply_gradient(grads)
    return new_state

# A typical training loop
state = create_train_state()
for batch in data_loader:
    state = train_step(state, batch)
```

Intra-Op Parallelism - Goal

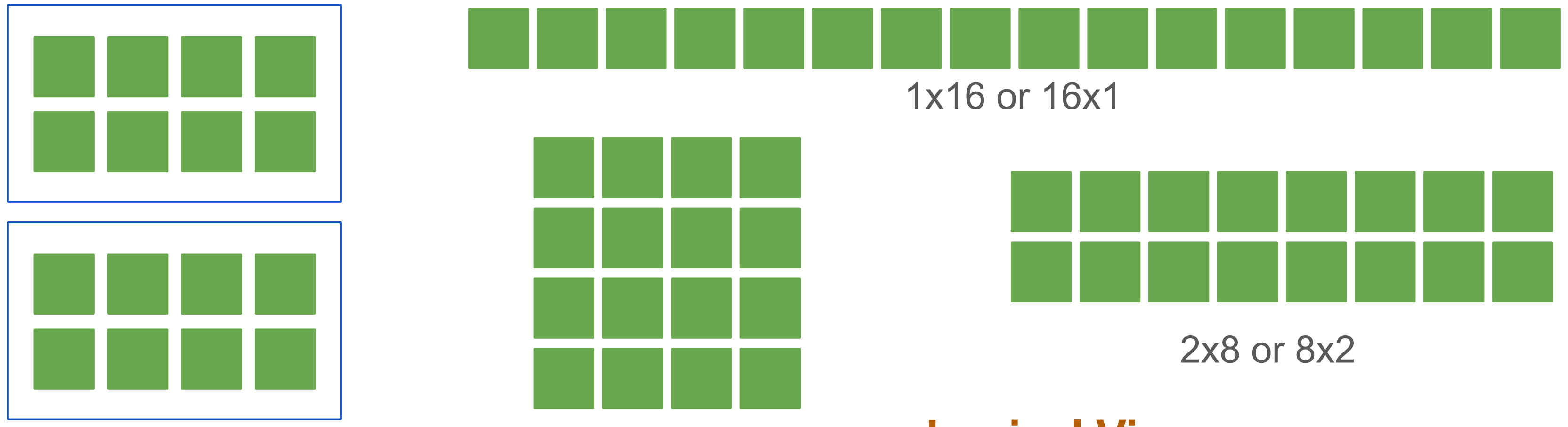
Goal: find a **intra-op parallel plan** to minimize the **intra-op cost**

How:

- **Building the searching space**: device mesh, sharding spec, resharding
- **Formulating the cost**
- **Optimizing the cost**

Intra-Op Parallelism - Device Mesh

Device mesh is the **logical 2D mesh view** of a set of GPUs



Physical View:
2 node, 8 GPUs per node

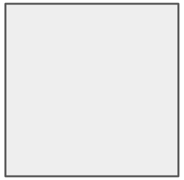
Logical View

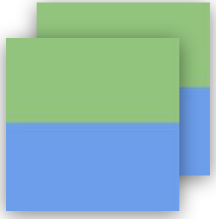
Which mapping? optimized by inter-op pass!

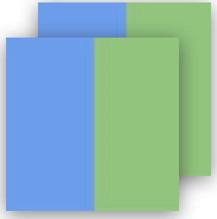
Intra-Op Parallelism - Sharding Spec

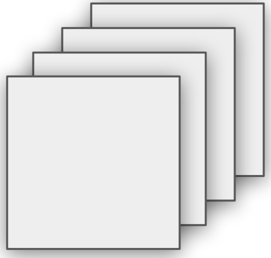
Sharding spec is to define the **layout of a tensor**

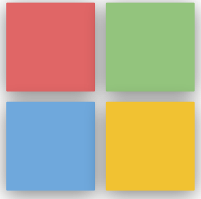
N-dimensional matrix: $\mathbf{X_0X_1\ldots X_{n-1}}$, where $\mathbf{X_i \in \{S, R\}}$, means **sliced/replicated** on i-th dimension


2D matrix


SR:
row-partitioned

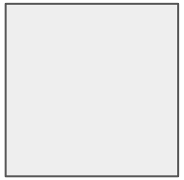

RS:
column-partitioned


RR:
no partitioning

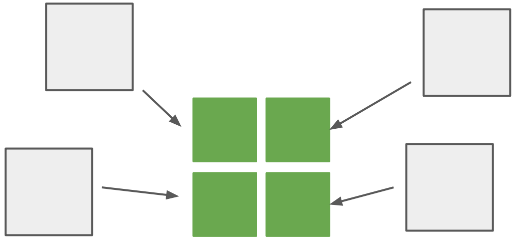

SS:
row- and column-partitioned

Intra-Op Parallelism - Sharding Spec (Cont.)

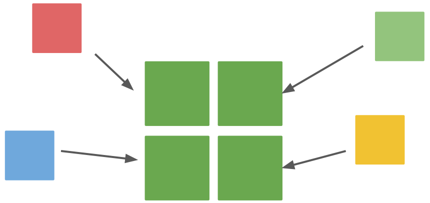
Mapping **tensor axes** to **device mesh axes:** add **superscript** to S



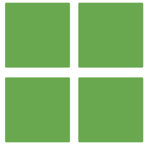
2D matrix



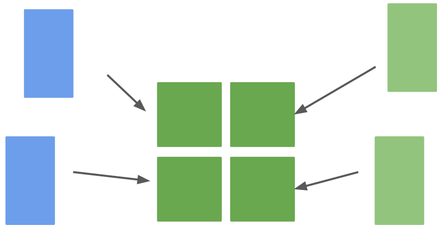
RR



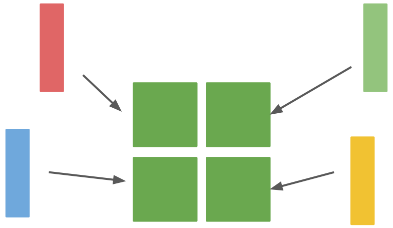
S^0 S^1



2x2 mesh



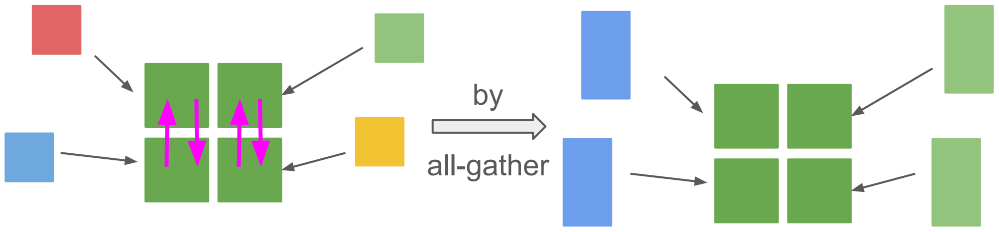
RS^1



RS^{01}

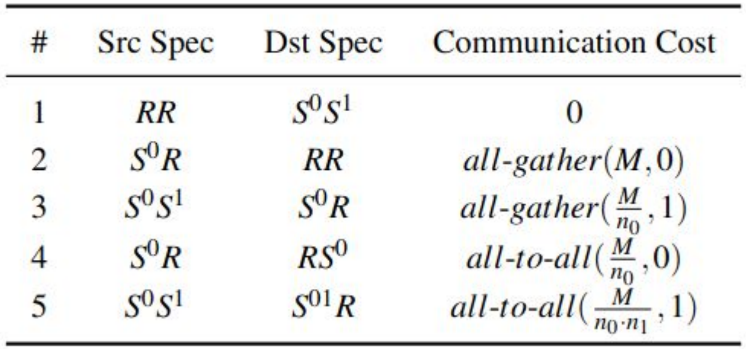
Intra-Op Parallelism - Resharding

Means **layout conversion**, when an input tensor does not satisfy the sharding spec of the chosen parallel plan for the operator. It will introduce **communication cost**



S^0 S^1:
output sharding spec of **previous op**

RS^1
sharding spec of **this op**



#	Src Spec	Dst Spec	Communication Cost
1	RR	S^0S^1	0
2	S^0R	RR	$\mathit{all\text{-}gather}(M,0)$
3	S^0S^1	S^0R	$\mathit{all\text{-}gather}(\frac{M}{n_0},1)$
4	S^0R	RS^0	$\mathit{all\text{-}to\text{-}all}(\frac{M}{n_0},0)$
5	S^0S^1	$S^{01}R$	$\mathit{all\text{-}to\text{-}all}(\frac{M}{n_0 \cdot n_1},1)$

several cases of communication cost

Intra-Op Parallelism - Parallel Algorithms of An Operator

Means map the **loop axes** to **mesh axes**, introducing **communication cost**

C=AB $\Longrightarrow$ $C_{b,i,j} = \sum_k A_{b,i,k} B_{b,k,j}$

mesh shape: (n_0, n_1)

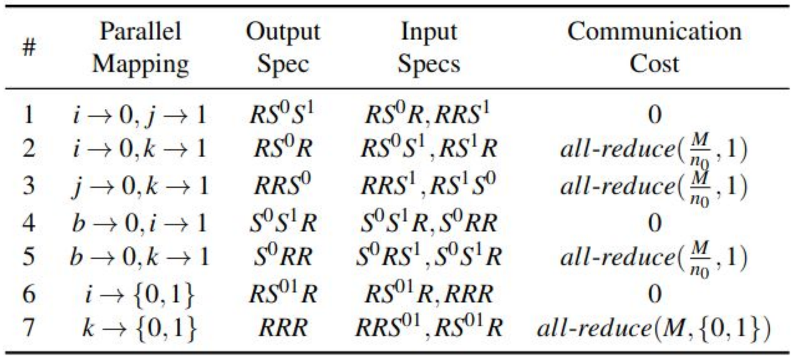
loop axes: b, i, j, k

mesh axes: 0, 1

If using i→0, k→1 mapping

Input spec: **R S^0 S^1, R S^1 R**
Output spec: **R S^0 R**

Communication cost: $all\text{-}reduce(\frac{M}{n_0}, 1)$



#	Parallel Mapping	Output Spec	Input Specs	Communication Cost
1	$i \to 0, j \to 1$	RS^0S^1	RS^0R, RRS^1	0
2	$i \to 0, k \to 1$	RS^0R	RS^0S^1, RS^1R	$all\text{-}reduce(\frac{M}{n_0}, 1)$
3	$j \to 0, k \to 1$	RRS^0	RRS^1, RS^1S^0	$all\text{-}reduce(\frac{M}{n_0}, 1)$
4	$b \to 0, i \to 1$	S^0S^1R	S^0S^1R, S^0RR	0
5	$b \to 0, k \to 1$	S^0RR	S^0RS^1, S^0S^1R	$all\text{-}reduce(\frac{M}{n_0}, 1)$
6	$i \to \{0,1\}$	$RS^{01}R$	$RS^{01}R, RRR$	0
7	$k \to \{0,1\}$	RRR	$RRS^{01}, RS^{01}R$	$all\text{-}reduce(M, \{0,1\})$

Intra-Op Parallelism - ILP Formulation

Formulating the **total intra-op cost** and **optimizing it** by an Integer Linear Programming (ILP) solver: on graph G=(V, E), e∈E, u, v ∈ V

Comp. and **comm. cost** of **node *v***:
number of parallel plan: k_v
comp. cost vector of plans: $c_v \in \mathbb{R}^{k_v}$
comm. cost vector of plans: $d_v \in \mathbb{R}^{k_v}$
choice of parallel: **one hot** vec $s_v \in \{0,1\}^{k_v}$

Resharding cost of **edge e**:
number of parallel plan: $k_v \;\; k_u$
resharding cost matrix: $R_{vu} \in \mathbb{R}^{k_v \times k_u}$

Total Inta-op Cost $$\sum_{v \in V} s_v^\intercal (c_v + d_v) + \sum_{(v,u) \in E} s_v^\intercal R_{vu} s_u$$

optimize
S_u S_v !

Intra-Op Parallelism - ILP Formulation (Cont.)

How to get c_v, d_v, R_{uv} ?

$$\sum_{v \in V} s_v^\intercal (c_v + d_v) + \sum_{(v,u) \in E} s_v^\intercal R_{vu} s_u$$

By profiling? too much cases!

By estimating for simplicity:

- **comp. cost** c_v: set as 0
  - heavy ops (e.g. matmul): no replication, so arithmetic complexity is same for all parallel plans
  - light ops (e.g. element-wise): negligible
- **comm. cost** d_v and **resharding cost** R_{uv}: communication bytes

Inter-op Parallelism - Goal

Goal: Slice computation graph and device cluster to *stage-mesh* pair such that

Pipeline execution latency is <u>minimized</u> and model is <u>fit into memory</u>

$$T^* = \min_{\substack{s_1,\ldots,s_S;\\(n_1,m_1),\ldots,(n_S,m_S)}} \left\{ \sum_{i=1}^{S} t_i + (B-1) \cdot \max_{1 \le j \le S} \{t_j\} \right\}. \quad (2)$$

We want to solve (2), under these additional constraints

- Colocate forward with corresponding backward operator on the same submesh
- The sliced submesh $(n_1, m_1), \ldots, (n_S, m_S)$ must fully cover the N x M cluster mesh (use all compute devices)

Inter-op Parallelism - Goal

Goal: Slice computation graph and device cluster to *stage-mesh* pair such that

Pipeline execution latency is <u>minimized</u> and model is <u>fit into memory</u>

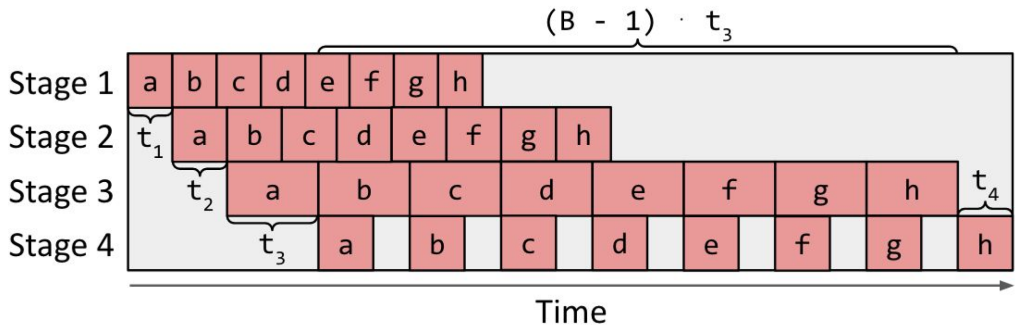
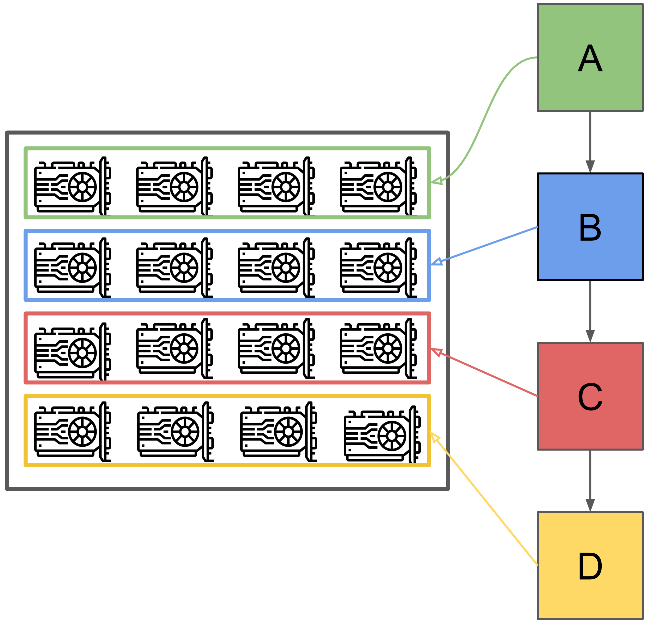
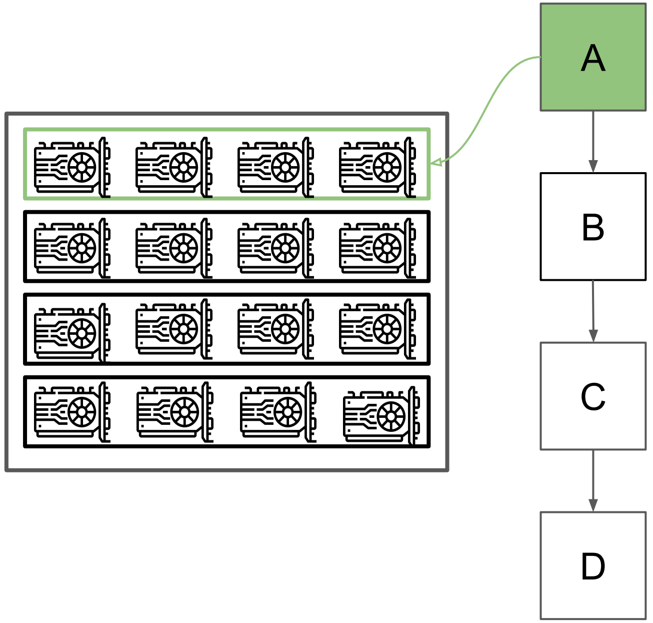


Figure 5: Illustration of the total latency of a pipeline, which is determined by two parts: the total latency of all stages ($t_1 + t_2 + t_3 + t_4$) and the latency of the slowest stage ($(B-1) \cdot t_3$).

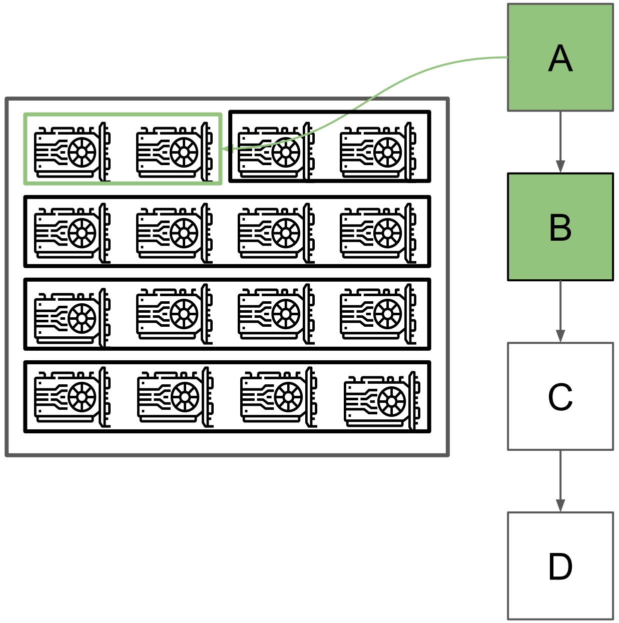


Inter-op Parallelism - Challenges

Challenges: There are many ways to slice computation graph and device cluster to stage-mesh pair. How do we know which stage-mesh mapping is the best?



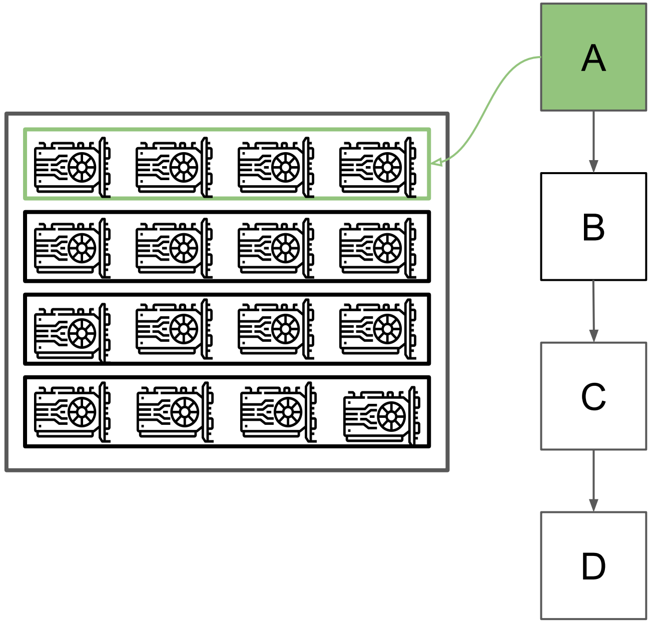
or



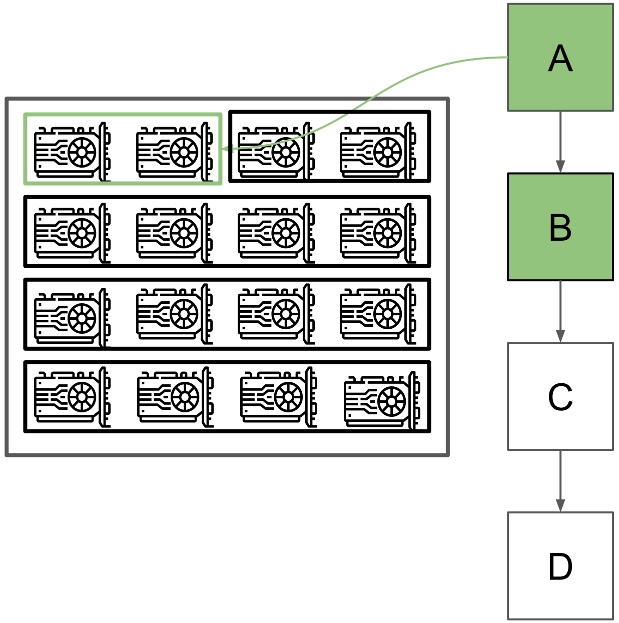
… and many more

Inter-op Parallelism - Challenges

Challenges: There are many ways to slice computation graph and device cluster to stage-mesh pair. How do we know which stage-mesh mapping is the best?



or



… and many more

Solution? DP!

Inter-op Parallelism - DP Formulation

Our submesh spaces $(n_1, m_1), ..., (n_S, m_S)$ consists of two options

- One-dimensional submeshes $(1, 1), (1, 2), (1, 4), ..., (1, M)$
  - I.e use 1, 2, 4, 8, … devices in a single node
- Two-dimensional submeshes $(2, M), (3, M), ..., (N, M)$
  - i.e use multiple nodes and all the devices of those nodes

Other choices, such as (n, m) where n > 1 and m < M, (i.e use multiple nodes but not all devices on those nodes) leads to inferior result. The above two choices can fully cover the device mesh N x M (proof in paper)

Inter-op Parallelism - DP Formulation

$$F(s,k,d;t_{max}) \tag{3}$$
$$= \min_{\substack{k \le i \le K \\ n_s \cdot m_s \le d}} \left\{ \begin{array}{l} \boxed{t_{intra}((o_k,\ldots,o_i), Mesh(n_s,m_s), s)} \\ \quad + F(s-1, i+1, d - n_s \cdot m_s; t_{max}) \\ \mid t_{intra}((o_k,\ldots,o_i), Mesh(n_s,m_s), s) \le t_{max} \end{array} \right\},$$

Lowest latency to run $(o_k,\ldots,o_i)$ on $Mesh(n_s,m_s)$

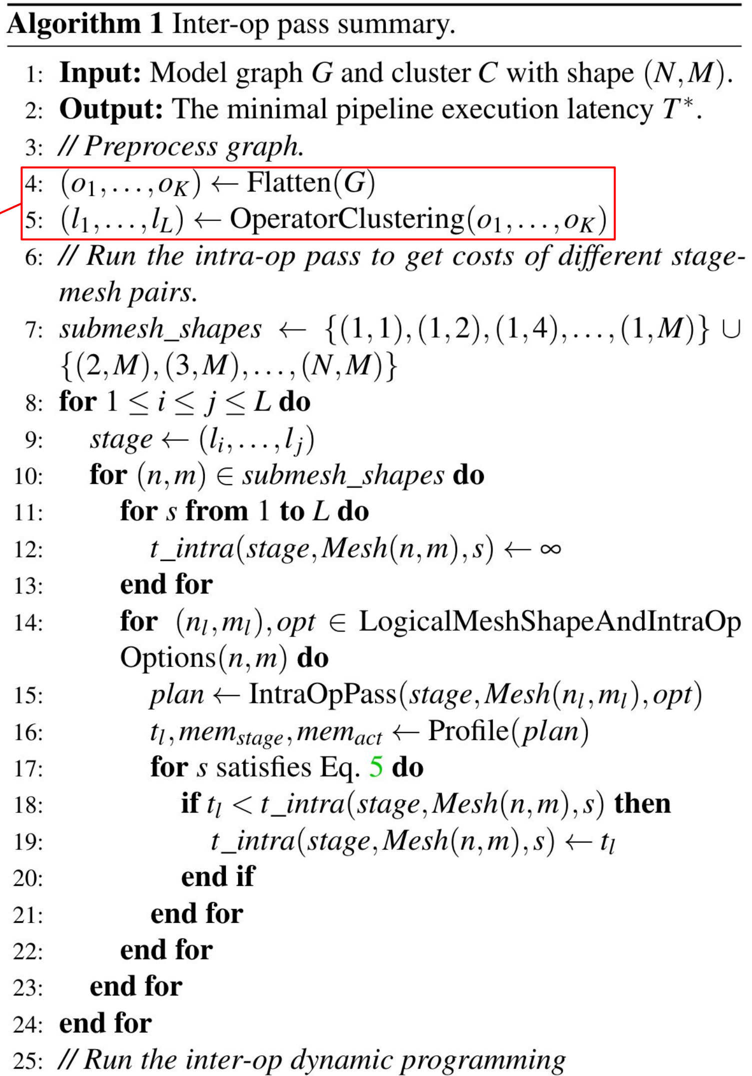
Set to infinity if OOMs

Represents the minimal total latency when slicing operators ok to oK into s stages and putting them onto d devices so that the latency of each stage is less than tmax

$$T^*(t_{max}) = \min_{s}\{F(s,0,N \cdot M; t_{max})\} + (B-1) \cdot t_{max}. \tag{4}$$

Inter-op Parallelism - Putting it all together

Flatten the computation graph and condense the operators into layers



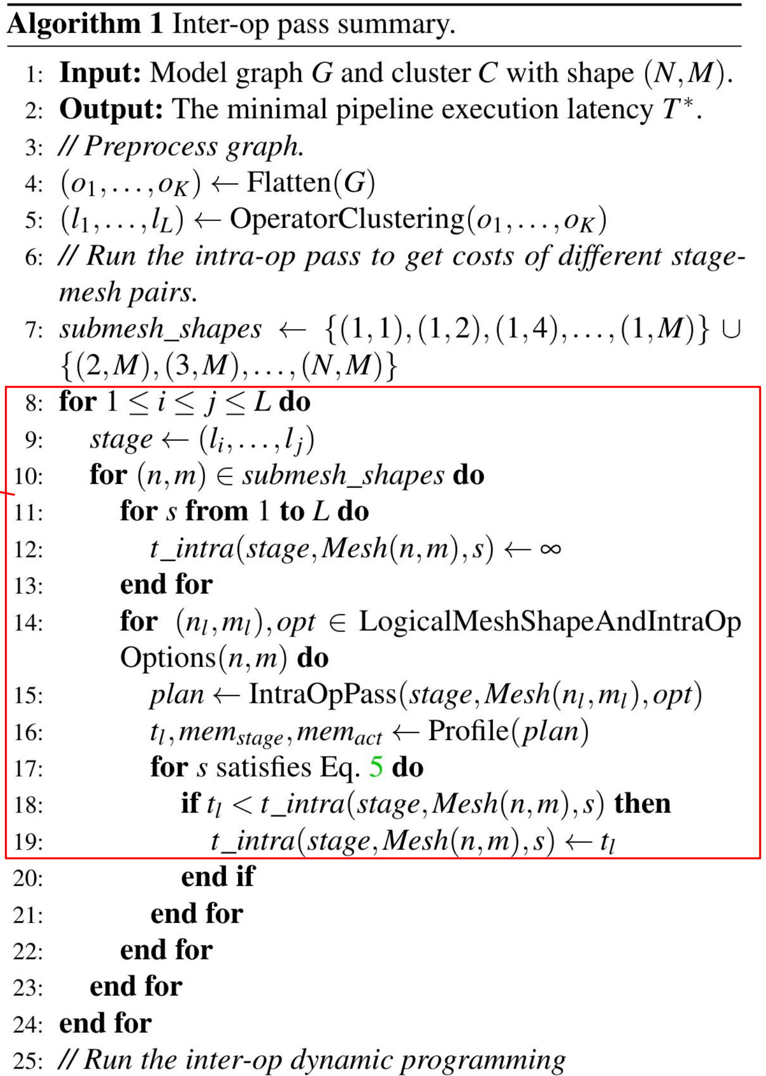
Algorithm 1 Inter-op pass summary.

```
1:  Input: Model graph $G$ and cluster $C$ with shape $(N, M)$.
2:  Output: The minimal pipeline execution latency $T^*$.
3:  // Preprocess graph.
4:  $(o_1, \ldots, o_K) \leftarrow \text{Flatten}(G)$
5:  $(l_1, \ldots, l_L) \leftarrow \text{OperatorClustering}(o_1, \ldots, o_K)$
6:  // Run the intra-op pass to get costs of different stage-mesh pairs.
7:  $submesh\_shapes \leftarrow \{(1,1), (1,2), (1,4), \ldots, (1,M)\} \cup \{(2,M), (3,M), \ldots, (N,M)\}$
8:  for $1 \le i \le j \le L$ do
9:      $stage \leftarrow (l_i, \ldots, l_j)$
10:     for $(n,m) \in submesh\_shapes$ do
11:         for $s$ from 1 to $L$ do
12:             $t\_intra(stage, Mesh(n,m), s) \leftarrow \infty$
13:         end for
14:         for $(n_l, m_l), opt \in$ LogicalMeshShapeAndIntraOpOptions$(n,m)$ do
15:             $plan \leftarrow \text{IntraOpPass}(stage, Mesh(n_l, m_l), opt)$
16:             $t_l, mem_{stage}, mem_{act} \leftarrow \text{Profile}(plan)$
17:             for $s$ satisfies Eq. 5 do
18:                 if $t_l < t\_intra(stage, Mesh(n,m), s)$ then
19:                     $t\_intra(stage, Mesh(n,m), s) \leftarrow t_l$
20:                 end if
21:             end for
22:         end for
23:     end for
24: end for
25: // Run the inter-op dynamic programming
```

Inter-op Parallelism - Putting it all together

Precalculate lowest execution latency for every stage-mesh pair using

IntraOpPass



Algorithm 1 Inter-op pass summary.

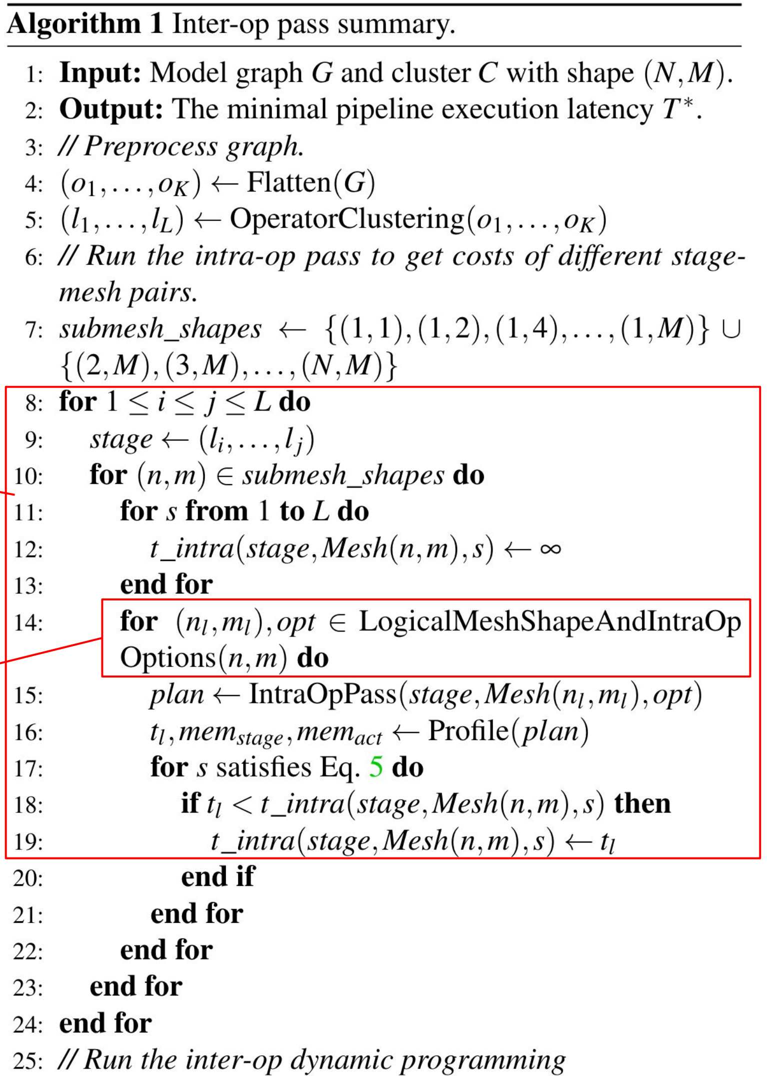
```
Input: Model graph $G$ and cluster $C$ with shape $(N,M)$.
Output: The minimal pipeline execution latency $T^*$.
// Preprocess graph.
$(o_1,\dots,o_K) \leftarrow \text{Flatten}(G)$
$(l_1,\dots,l_L) \leftarrow \text{OperatorClustering}(o_1,\dots,o_K)$
// Run the intra-op pass to get costs of different stage-mesh pairs.
$submesh\_shapes \leftarrow \{(1,1),(1,2),(1,4),\dots,(1,M)\} \cup \{(2,M),(3,M),\dots,(N,M)\}$
for $1 \le i \le j \le L$ do
    $stage \leftarrow (l_i,\dots,l_j)$
    for $(n,m) \in submesh\_shapes$ do
        for $s$ from 1 to $L$ do
            $t\_intra(stage, Mesh(n,m), s) \leftarrow \infty$
        end for
        for $(n_l,m_l), opt \in$ LogicalMeshShapeAndIntraOpOptions$(n,m)$ do
            $plan \leftarrow \text{IntraOpPass}(stage, Mesh(n_l,m_l), opt)$
            $t_l, mem_{stage}, mem_{act} \leftarrow \text{Profile}(plan)$
            for $s$ satisfies Eq. 5 do
                if $t_l < t\_intra(stage, Mesh(n,m), s)$ then
                    $t\_intra(stage, Mesh(n,m), s) \leftarrow t_l$
                end if
            end for
        end for
    end for
end for
// Run the inter-op dynamic programming
```

Inter-op Parallelism - Putting it all together

Precalculate lowest execution latency for every stage-mesh pair using

IntraOpPass

Can interpret a (n, m) physical mesh as any (n’, m’) virtual mesh such that n’m’ = nm



Algorithm 1 Inter-op pass summary.

```
**Input:** Model graph $G$ and cluster $C$ with shape $(N, M)$.
**Output:** The minimal pipeline execution latency $T^*$.
// Preprocess graph.
$(o_1, \ldots, o_K) \leftarrow \text{Flatten}(G)$
$(l_1, \ldots, l_L) \leftarrow \text{OperatorClustering}(o_1, \ldots, o_K)$
// Run the intra-op pass to get costs of different stage-mesh pairs.
$submesh\_shapes \leftarrow \{(1,1),(1,2),(1,4),\ldots,(1,M)\} \cup \{(2,M),(3,M),\ldots,(N,M)\}$
for $1 \le i \le j \le L$ do
    $stage \leftarrow (l_i, \ldots, l_j)$
    for $(n,m) \in submesh\_shapes$ do
        for $s$ from 1 to $L$ do
            $t\_intra(stage, Mesh(n,m), s) \leftarrow \infty$
        end for
        for $(n_l, m_l), opt \in$ LogicalMeshShapeAndIntraOpOptions$(n,m)$ do
            $plan \leftarrow \text{IntraOpPass}(stage, Mesh(n_l, m_l), opt)$
            $t_l, mem_{stage}, mem_{act} \leftarrow \text{Profile}(plan)$
            for $s$ satisfies Eq. 5 do
                if $t_l < t\_intra(stage, Mesh(n,m), s)$ then
                    $t\_intra(stage, Mesh(n,m), s) \leftarrow t_l$
                end if
            end for
        end for
    end for
end for
// Run the inter-op dynamic programming
```

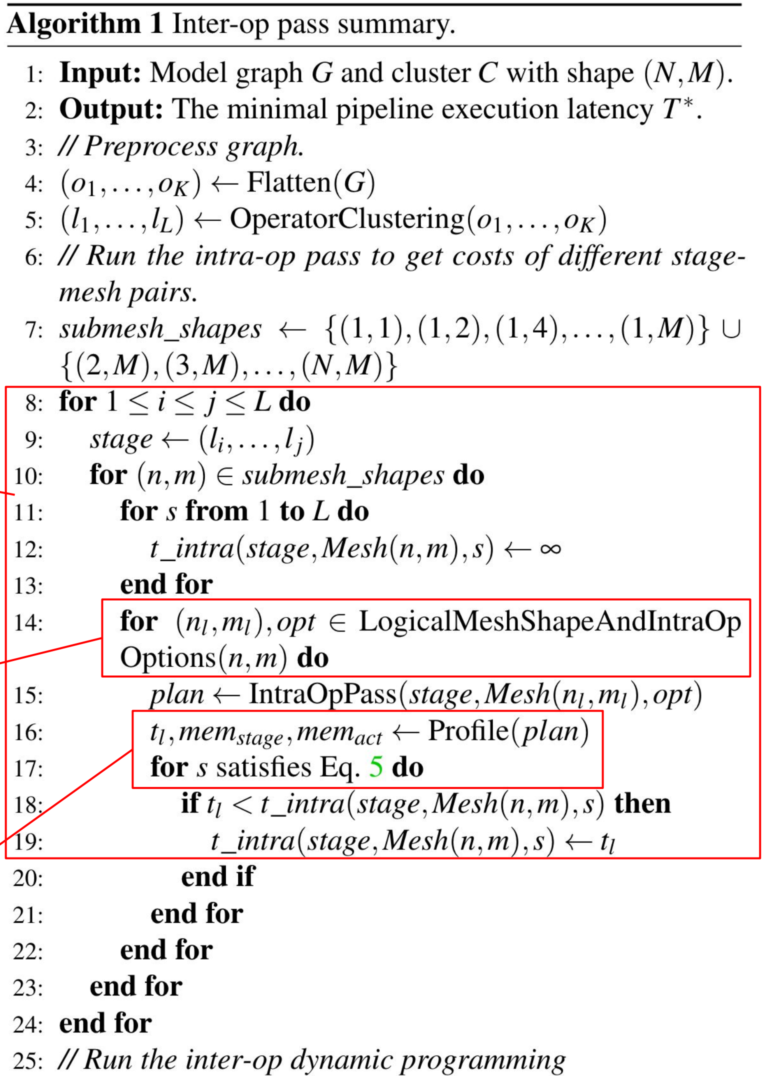
Inter-op Parallelism - Putting it all together

Precalculate lowest execution latency for every stage-mesh pair using

IntraOpPass

Can interpret a (n, m) physical mesh as any (n’, m’) virtual mesh such that n’m’ = nm

Profile memory usage and only keep the Intra-op parallelism plans that do not result in OOM

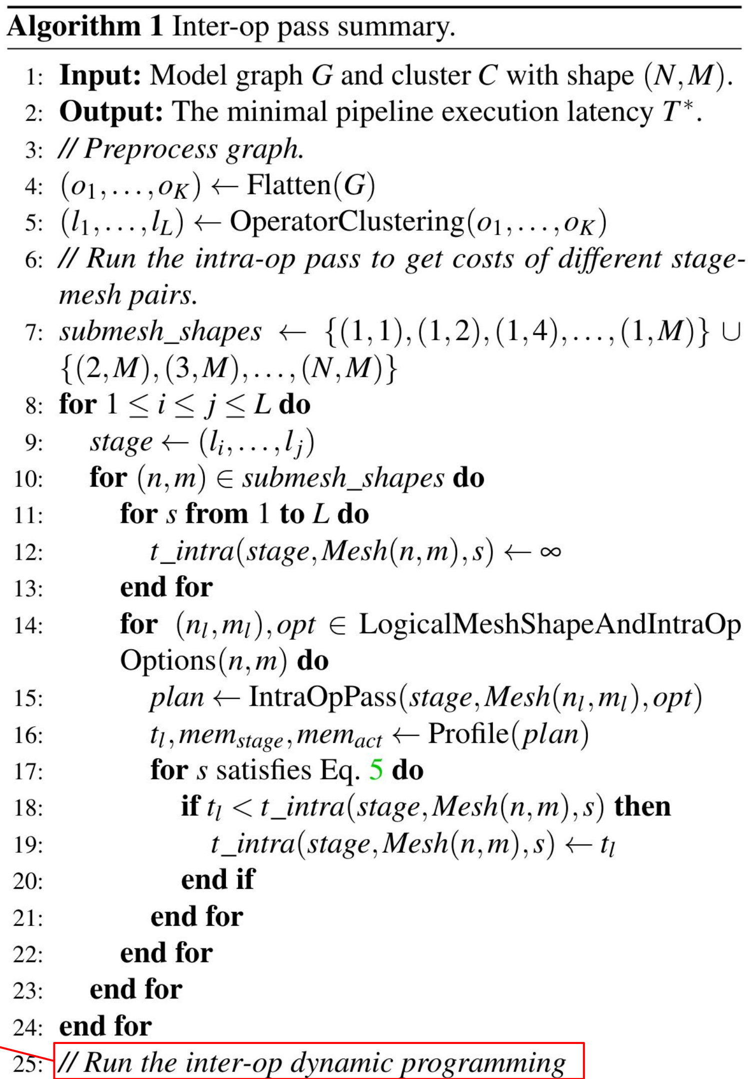


Algorithm 1 Inter-op pass summary.

```
**Input:** Model graph $G$ and cluster $C$ with shape $(N, M)$.
**Output:** The minimal pipeline execution latency $T^*$.
// Preprocess graph.
$(o_1, \ldots, o_K) \leftarrow \text{Flatten}(G)$
$(l_1, \ldots, l_L) \leftarrow \text{OperatorClustering}(o_1, \ldots, o_K)$
// Run the intra-op pass to get costs of different stage-mesh pairs.
$submesh\_shapes \leftarrow \{(1,1),(1,2),(1,4),\ldots,(1,M)\} \cup \{(2,M),(3,M),\ldots,(N,M)\}$
**for** $1 \le i \le j \le L$ **do**
    $stage \leftarrow (l_i, \ldots, l_j)$
    **for** $(n, m) \in submesh\_shapes$ **do**
        **for** $s$ **from** 1 **to** $L$ **do**
            $t\_intra(stage, Mesh(n,m), s) \leftarrow \infty$
        **end for**
        **for** $(n_l, m_l), opt \in$ LogicalMeshShapeAndIntraOpOptions$(n,m)$ **do**
            $plan \leftarrow \text{IntraOpPass}(stage, Mesh(n_l, m_l), opt)$
            $t_l, mem_{stage}, mem_{act} \leftarrow \text{Profile}(plan)$
            **for** $s$ satisfies Eq. 5 **do**
                **if** $t_l < t\_intra(stage, Mesh(n,m), s)$ **then**
                    $t\_intra(stage, Mesh(n,m), s) \leftarrow t_l$
                **end if**
            **end for**
        **end for**
    **end for**
**end for**
// Run the inter-op dynamic programming
```

Inter-op Parallelism - Putting it all together

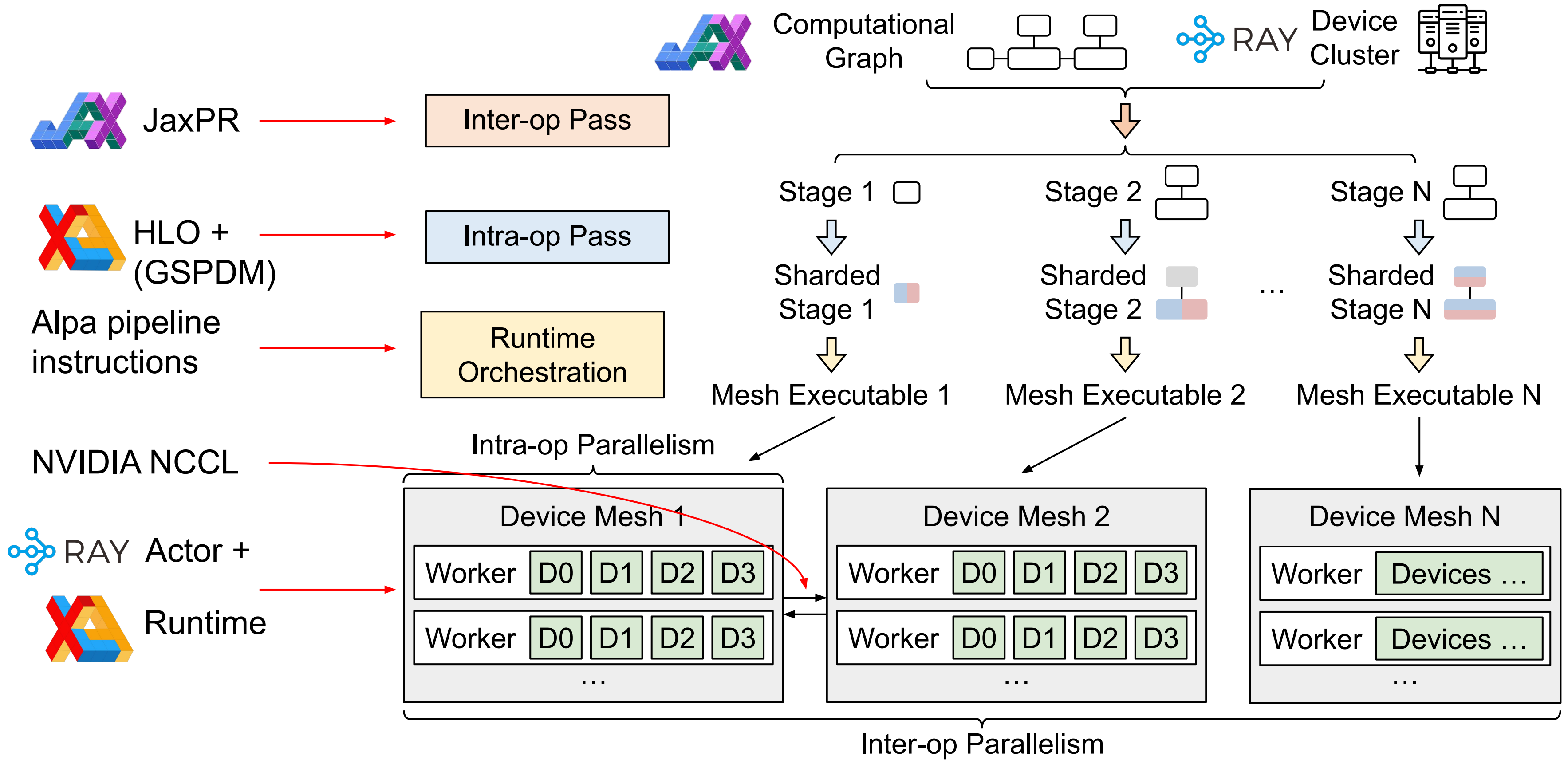
Run the inter-op dynamic programming



Algorithm 1 Inter-op pass summary.

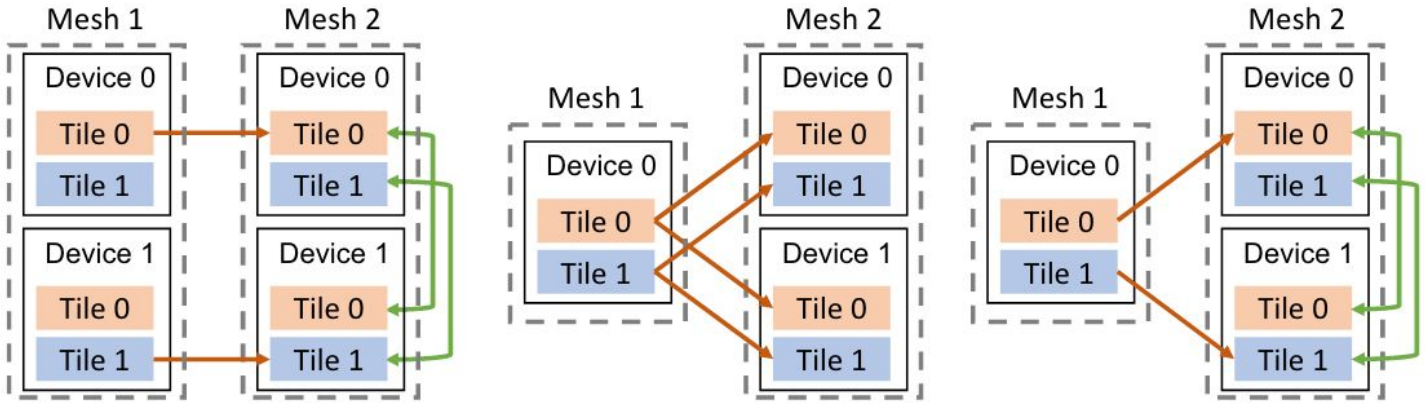
```
1:  Input: Model graph $G$ and cluster $C$ with shape $(N, M)$.
2:  Output: The minimal pipeline execution latency $T^*$.
3:  // Preprocess graph.
4:  $(o_1, \ldots, o_K) \leftarrow$ Flatten($G$)
5:  $(l_1, \ldots, l_L) \leftarrow$ OperatorClustering($o_1, \ldots, o_K$)
6:  // Run the intra-op pass to get costs of different stage-mesh pairs.
7:  $submesh\_shapes \leftarrow \{(1,1),(1,2),(1,4),\ldots,(1,M)\} \cup \{(2,M),(3,M),\ldots,(N,M)\}$
8:  for $1 \leq i \leq j \leq L$ do
9:      $stage \leftarrow (l_i, \ldots, l_j)$
10:     for $(n, m) \in submesh\_shapes$ do
11:         for $s$ from 1 to $L$ do
12:             $t\_intra(stage, Mesh(n,m), s) \leftarrow \infty$
13:         end for
14:         for $(n_l, m_l), opt \in$ LogicalMeshShapeAndIntraOpOptions($n, m$) do
15:             $plan \leftarrow$ IntraOpPass($stage, Mesh(n_l, m_l), opt$)
16:             $t_l, mem_{stage}, mem_{act} \leftarrow$ Profile($plan$)
17:             for $s$ satisfies Eq. 5 do
18:                 if $t_l < t\_intra(stage, Mesh(n,m), s)$ then
19:                     $t\_intra(stage, Mesh(n,m), s) \leftarrow t_l$
20:                 end if
21:             end for
22:         end for
23:     end for
24: end for
25: // Run the inter-op dynamic programming
```

Parallelism Orchestration



Parallelism Orchestration - Cross-mesh resharding

- In Megatron-LM, each pipeline stages have same degrees of data and tensor parallelism. Point-to-point communication between correspondent devices
- For Alpa, device meshes holding two consecutive stages may have different shapes.

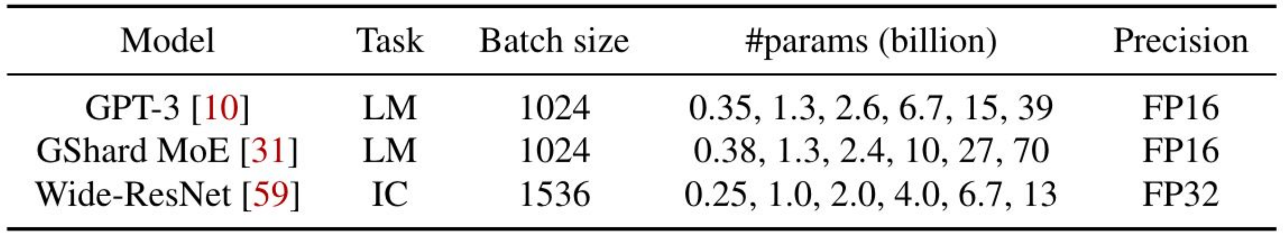


(a) Megatron-LM (b) Naïve send/recv (c) Local all-gather

Evaluation - Setup

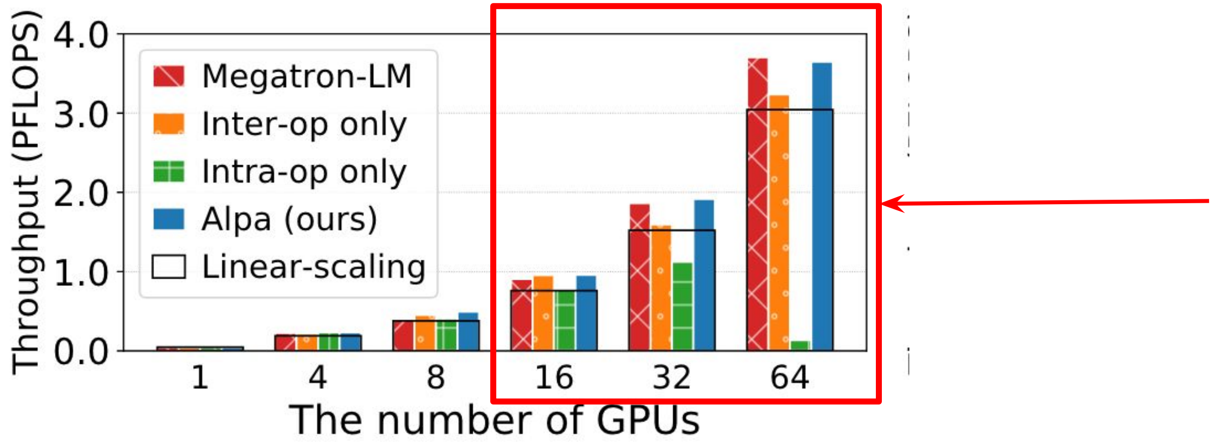
- Each node is an Amazon EC2 p3.16xlarge instance with 8 NVIDIA. V100 16 GB GPUs, 64 vCPUs, and 488 GB memory.
    - The 8 GPUs in a node are connected via NVLink. 25Gbps cross-node bandwidth
- Respects the semantics of synchronous gradient descent, thus does not affect model convergence
- Evaluate weak scaling by increasing model size along with number of GPUs

Table 4: Models used in the end-to-end evaluation. LM = language model. IC = image classification.



Model	Task	Batch size	#params (billion)	Precision
GPT-3 [10]	LM	1024	0.35, 1.3, 2.6, 6.7, 15, 39	FP16
GShard MoE [31]	LM	1024	0.38, 1.3, 2.4, 10, 27, 70	FP16
Wide-ResNet [59]	IC	1536	0.25, 1.0, 2.0, 4.0, 6.7, 13	FP32

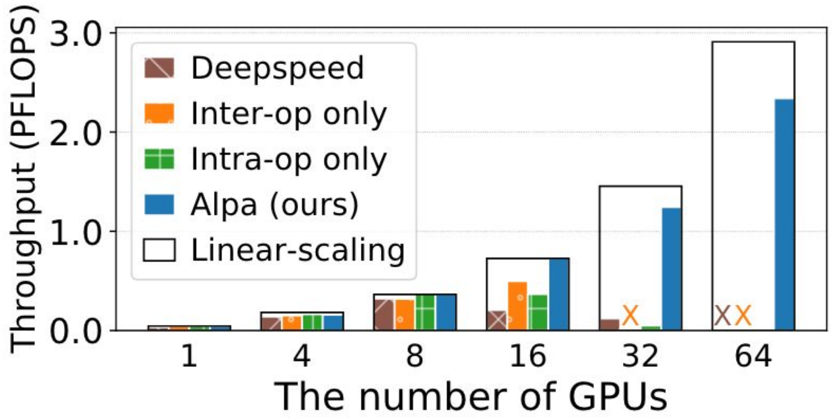
Evaluation - End-to-end (Weak Scaling)



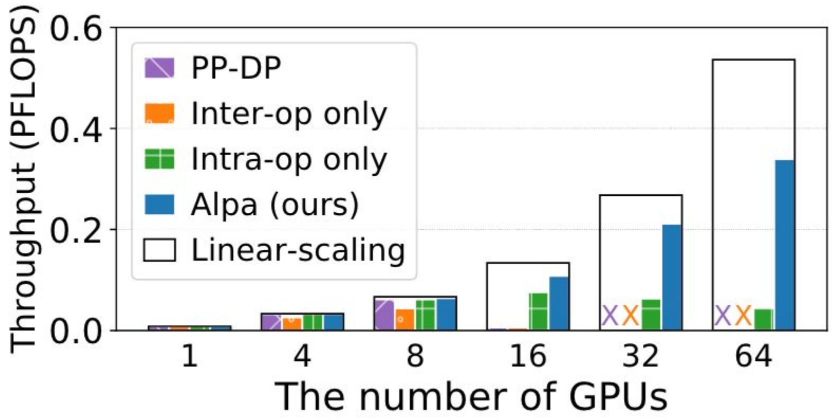
(a) GPT

- Alpa generated parallelism plan closely resembles Megatron-LM best-performed plans
- Key diff: Alpa also partitions weight-update operation when DP exists => slight improvement to Megatron-LM in some config

Evaluation - End-to-end (Weak Scaling)



(b) MoE

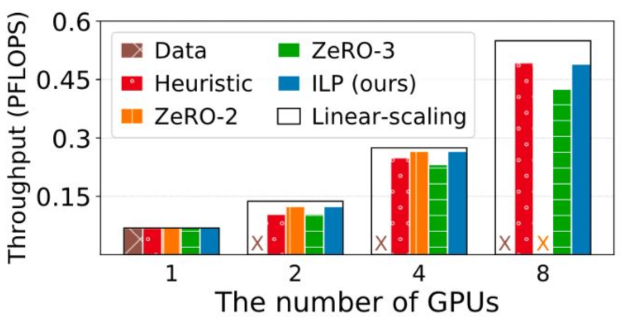


(c) Wide-ResNet

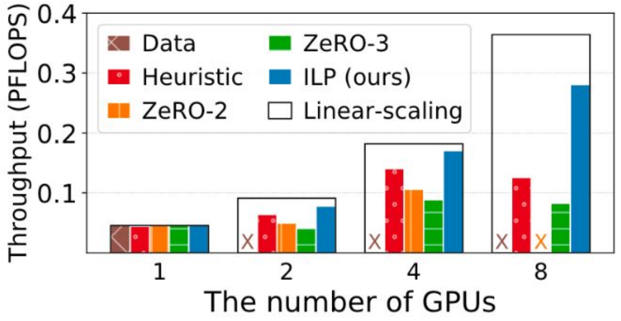
- Slightly better/matches DeepSpeed for single node performance
- DeepSpeed MoE does not have PP. Alpa performs 3.5x on 2 nodes and 9.7x on 4 nodes

- Heterogeneous architecture. Very hard for manual parallelism plan
- Alpa still manage to find 80% scaling parallelism plan

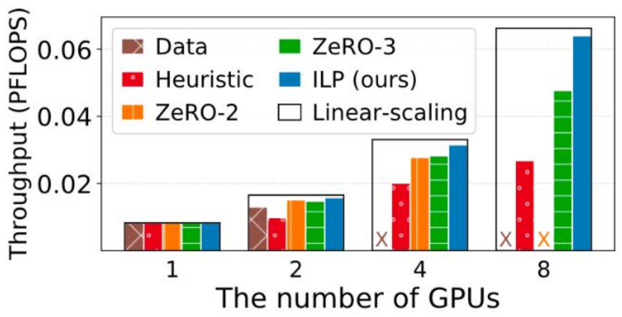
Evaluation - Intra-op only study



(a) GPT



(b) MoE



(c) Wide-ResNet

- ZeRO optimizes for memory but not communication overhead
- Alpa's ILP always figure out the correct plan that minimize communication overhead in all cases, achieving near linear-scaling, while making sure the model fits into memory
- For MoE, Alpa ILP managed to find and combine expert parallelism and ZeRO-flavour data parallelism

Case Study: Wide-ResNet

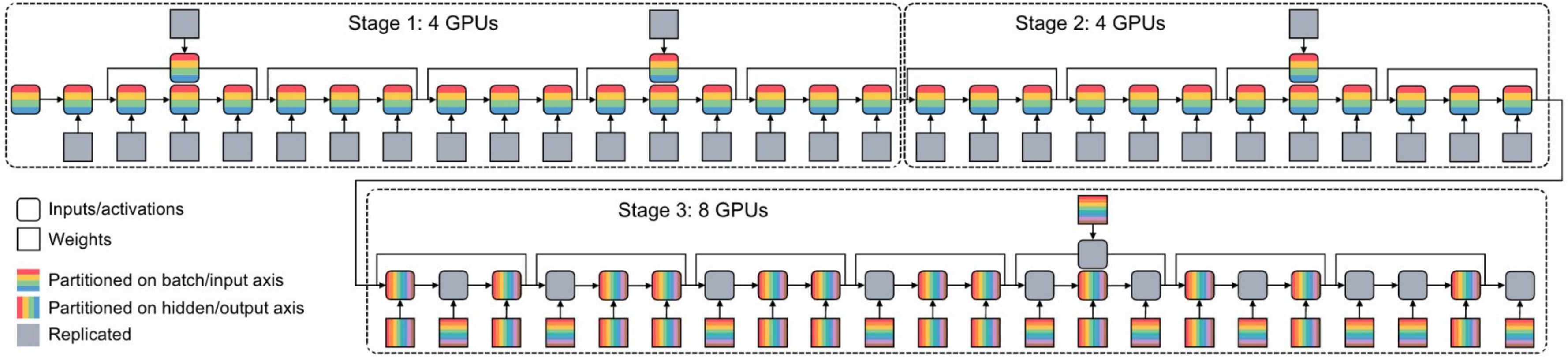


Figure 12: Visualization of the parallel strategy of Wide-ResNet on 16 GPUs. Different colors represent the devices a tensor is distributed on. Grey blocks indicate a tensor is replicated across the devices. The input data and resulting activation of each convolution and dense layer can be partitioned along the batch axis and the hidden axis. The weights can be partitioned along the input and output channel axis.

Compilation Overhead (Runtime of Algorithm 1)

- Most of the time is spent on enumerating and profiling stage-mesh (preprocessing)
- Speedup profiling by a simple cost model built at XLA instruction level
- Compile executable for each stage in parallel with distributed workers

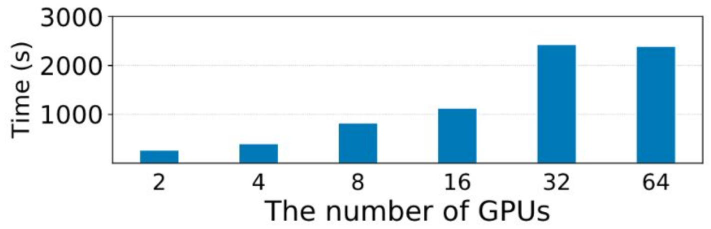


Figure 10: Alpa's compilation time on all GPT models. The model size and #GPUs are simultaneously scaled.

Table 5: Compilation time breakdown of GPT-39B.

Steps	Ours	w/o optimization
Compilation	1582.66 s	> 16hr
Profiling	804.48 s	> 24hr
Stage Construction DP	1.65 s	N/A
Other	4.47 s	N/A
Total	2393.26 s	> 40hr

Alpa Present and Future

- Alpa project is no longer actively maintaining
- Instead, integrating into XLA's autosharding , idea is to compile model code (Torch, Jax, TensorFlow) to automatic parallelism executable without relying on users' annotation unlike GSPDM

Thoughts

- The only functional open-source automatic parallelism framework as of today!
- Works for any model without user code changes
- Built automatic support for GSPMD intra-op parallelism
  - Generalizable view of parallelism
  - All about choosing what dim to replicate/shard
- Matches performance of Megatron-LM in GPT and search results closely resembles Megatron-LM best-performed plans
- Cross-mesh resharding is not optimal (also acknowledged in the paper)
  - Follow up work MLSys 23’