清華大學电子工程系
Department of Electronic Engineering, Tsinghua University

大规模模型并行训练的自动并行化

宁雪妃

2022.04.21

目录 Contents

1 背景和基础

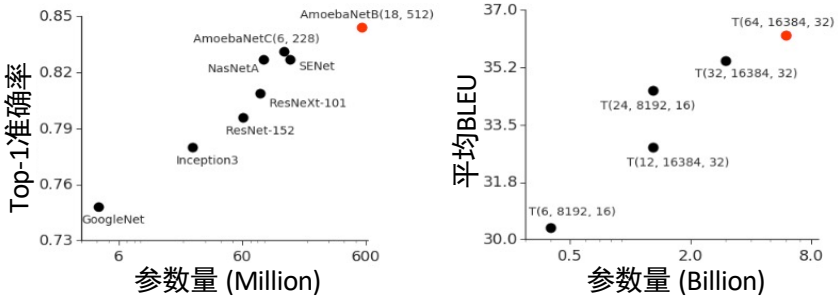
2 并行映射策略介绍

3 自动并行化

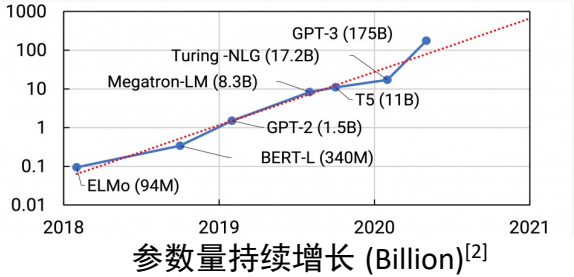
4 总结

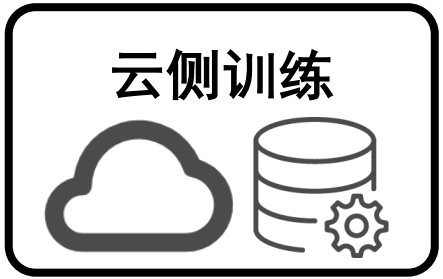
大模型趋势和并行训练的必要性

应用事实1: 深度学习模型放大效果变好[1]



应用事实2: 大数据时代，模型持续变大[2]





以GPT-3为例

- 高内存: 1.75×10^{11}参数量 - 典型GPU内存 10^{10} Byte
- 高计算: 训练数据数十TB，需要3640 PetaFlop/s-day (1PetaFlop/s-day=10^20 ops) - 约需64个A100训练1年

必然需要并行训练

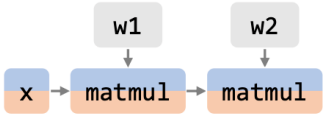
[1] Huang et al., Gpipe: Easy Scaling with Micro-Batch Pipeline Parallelism, NeurIPS'19.
[2] NVIDIA blog, https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/

并行映射策略、优化策略一览

大模型不能放到一个设备上，且其训练需要大算力资源 ➡ 需要并行化映射至集群上

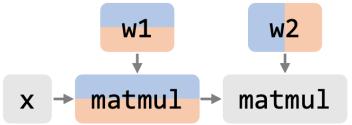
其它计算-通信-内存间做Trade-off的优化策略

数据并行 (Data)



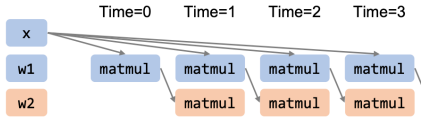
- 易用性高
- 不能降低内存占用
- 通信代价大
- 硬件利用率高

操作内并行 (Intra-Op Model)



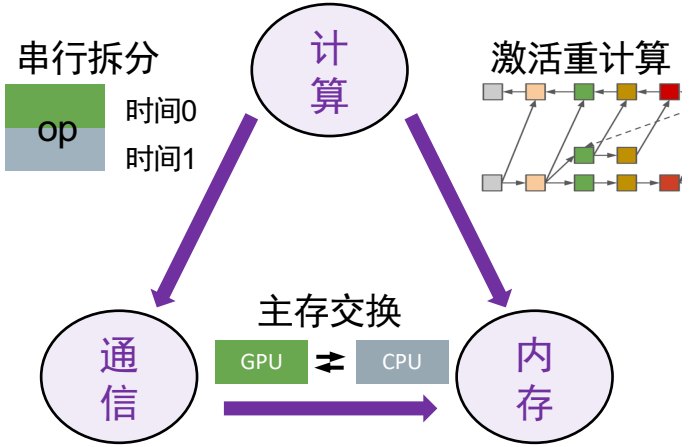
- 能降低内存占用
- 通信代价大
- 硬件利用率高

操作间并行 (Inter-Op Model)



一般指**流水线并行**

- 能降低内存占用
- 通信代价小
- 硬件利用率受到限制

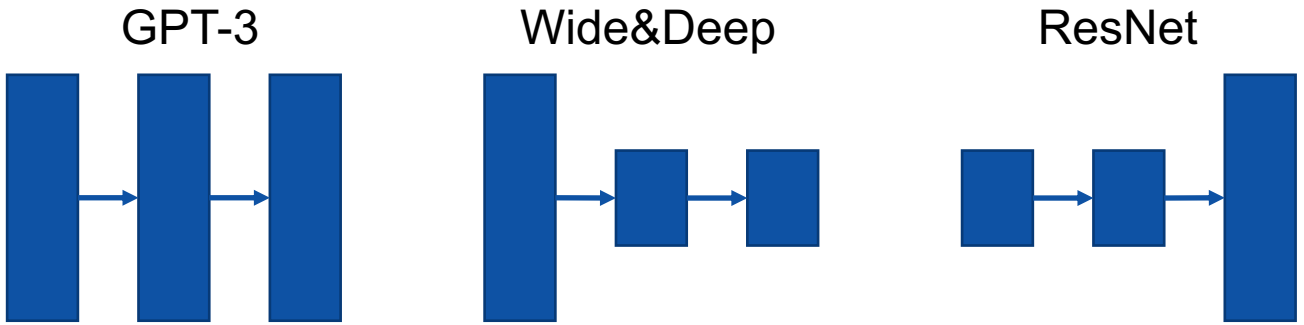


- 映射/优化策略着有不同Trade-off, 适用于不同模型&操作, 需混合达到最优效果
- 混合映射策略中, 决策间互相影响

[1] Zheng et al., Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning, arXiv'22.01. (图借用)

例：不同模型/操作适用不同策略

- NLP GPT-3: 超大参数量，训练数据量也很大 -> 数据并行+模型并行混用
- Rec Wide&Deep: 大型嵌入(Embedding)层，内存开销大，必须模型并行；后端模型小，可用数据并行
- CV ResNet: 分类类别数多，尾部全连接层参数量巨大 -> 在数万分类场景应对FC层采用模型并行，因为模型并行通信量正比于feature大小 (activation)，而数据并行通信量正比于参数大小 (gradient)



目录 Contents

1 背景和基础

2 并行映射策略介绍

3 自动并行化

4 总结

映射策略介绍

- 模型并行-流水线并行 Inter-Op Model Parallelism
  - GPipe@NIPS'19 by Google
  - PipeDream@SOSP'19 by Microsoft&CMU&Stanford
  - PipeDream-2BW@ICML'21 by Microsoft&Stanford
  - Chimera@SC'21 by ETH, Megatron-LM@SC'21 by NVIDIA&Stanford&Microsoft
- 模型并行-算子并行
  - GShard@ICLR'21 -> GSPMD@arXiv21 by Google
- 数据并行改进
  - Zero@SC'20 by Microsoft
  - Zero-Offload@ATC'21 by Microsoft
- 其它优化策略: 激活重计算 (在GPipe稍微介绍一下，不展开了)
  - Training Deep Nets with Sublinear Memory Cost@arXiv16 by Tianqi Chen
  - Checkmate@MLSys'20 by Microsoft

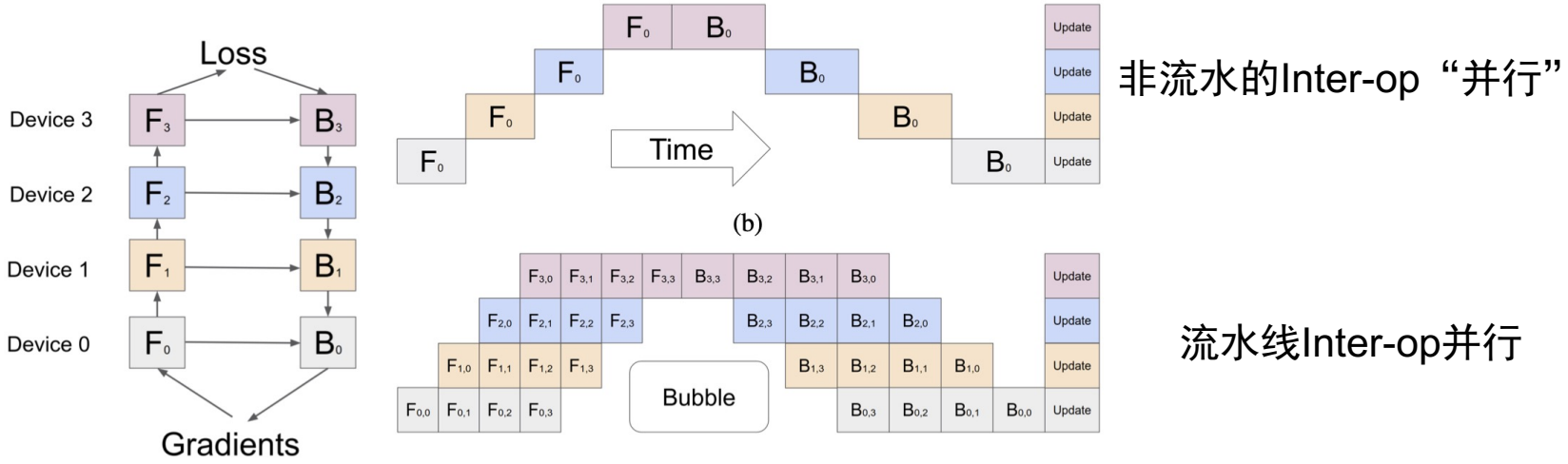
模型并行是什么

- 什么是模型并行，和常用的数据并行区别是什么
- Data Parallelism将模型**重复**到多个device上，每个device都计算整个模型、输入不同数据（一个minibatch切成K份）、并行计算
  - 通信代价：所有参数的梯度在每个minibatch需要一次all-reduce
  - 内存限制：不能降低每个device上的内存占用
- Intra-Op Model Parallelism将操作**划分**到多个device上，每个device负责计算部分op
  - 通信代价：常在op粒度需要all-reduce，例如，$C_{ij} = \sum_k A_{ik} B_{kj}$矩阵乘，如果在k loop维度做划分，需要all-reduce通信才能完成C计算
  - 内存限制：能降低每个device上的内存占用
- Inter-Op Model Parallelism将模型**划分**到多个device上，每个device只负责计算部分模型
  - 通信代价：point-to-point (across划分boundary) 的activation和activation gradient传输
  - 内存限制：能降低每个device上的内存占用
  - 虽然能降低内存限制，但是朴素的Inter-Op Model Parallelism里能并行计算的只有本身模型DAG里就并行的op节点，如果考虑线性情况，负责某层的device必须等前面的层算完才能算，不能同时计算

Inter-Op Model Parallelism可以帮助满足内存限制、通信代价低，但并行性不高、硬件利用效率低

流水线并行: GPipe@NIPS'19

- 使用pipeline并行的思想来提升Inter-Op Model Parallelism的硬件利用率。
- 将数据minibatch进一步切分成microbatch，不同device并行地对不同microbatch计算该device所负责的stage



- Minibatch size为N, 进一步切成M个microbatch。总共L layers，在K个device上partition成K个stage并pipeline并行。一个minibatch里的M个microbatch算完做一个gradient update。

流水线并行: GPipe@NIPS'19

- 切分方法: 1) 模型序列化，2) 根据用户可以定义per-layer cost，均分几个stage
- 权衡
  - 为了优化每个minibatch的latency，关于**microbatch数量M**的设置有一个权衡（由于在SGD训练有一个限制：minibatch size N不能超过一个安全值才能收敛，所以我们假设N固定）
  - microbatch越多，Bubble 相对比例$O(\frac{K-1}{M+K-1})$越小， 也就是说device们完全空闲等待的比例少。但是N固定，M大micro batch size就会小，导致 1) 每个计算的utilization低，(GPU可能不会充分利用)，整体计算时间不一定更短；2) training-mode的BN statistics计算带来影响

流水线并行: GPipe@NIPS'19

- **激活重计算 / Gradient Checkpointing / Rematerialization**
- Gpipe的主要贡献是引入了microbatch-based流水线，它实验中还用了一些其它常用优化，比如激活重计算降低内存开销（计算换内存）
  - 在前向过程中，每个device只存partition boundary的activation (checkpointing)
  - 在反向过程中重计算这个partition的前向过程、得到所想要的这个partition的activation。激活的峰值内存占用(peak activation memory) 从 降低到$O(N+\frac{L}{K}\times\frac{N}{M})$, $\frac{N}{M}$ 是microbatch size, $\frac{L}{K}$ 是每个stage的layer个数
    - 如果没有pipeline和rematerialization，网络需要存储一个minibatch里所有N个数据的所有feature，所以是$O(N\times L)$
    - 如果用上重计算，那么只要在计算一个partition里一个microbatch的时候online算出这个partition这个microbatch的中间activation，所以就是 $O\left(\frac{L}{K}\times\frac{N}{M}\right)$, +N指的是存boundary activation
- **“计算换内存”** → 由于降低了存储activation tensor的memory需求，可以增加batch size，增加throughput（所以也不一定会变“慢”）
- **对bubble比例的影响**：发现当 M ≥ 4K 时可以忽略bubble，因为backward pass里的recomputation可以被提前schedule，占据中间bubble的空间

流水线并行: GPipe@NIPS'19

➢结果

能放下的模型大小

可以放下的最大模型：

1. 单个device，不划分模型（没有流水线），只是通过引入micro batch splitting和gradient checkpointing降低peak memory使用，可以从支持82M amoebanet到318M。
2. 随着pipeline stage和partition数的同时增加 (k)，能支持的模型越来越大

Table 1: Maximum model size of AmoebaNet supported by GPipe under different scenarios. Naive-1 refers to the sequential version without GPipe. Pipeline-k means k partitions with GPipe on k accelerators. AmoebaNet-D (L, D): AmoebaNet model with L normal cell layers and filter size D. Transformer-L: Transformer model with L layers, 2048 model and 8192 hidden dimensions. Each model parameter needs 12 bytes since we applied RMSProp during training.

NVIDIA GPUs (8GB each)	Naive-1	Pipeline-1	Pipeline-2	Pipeline-4	Pipeline-8
AmoebaNet-D (L, D)	(18, 208)	(18, 416)	(18, 544)	(36, 544)	(72, 512)
# of Model Parameters	82M	318M	542M	1.05B	1.8B
Total Model Parameter Memory	1.05GB	3.8GB	6.45GB	12.53GB	24.62GB
Peak Activation Memory	6.26GB	3.46GB	8.11GB	15.21GB	26.24GB
Cloud TPUv3 (16GB each)	**Naive-1**	**Pipeline-1**	**Pipeline-8**	**Pipeline-32**	**Pipeline-128**
Transformer-L	3	13	103	415	1663
# of Model Parameters	282.2M	785.8M	5.3B	21.0B	83.9B
Total Model Parameter Memory	11.7G	8.8G	59.5G	235.1G	937.9G
Peak Activation Memory	3.15G	6.4G	50.9G	199.9G	796.1G

吞吐

当$M >> K$时，随着K的增加有almost linear speed up

Table 2: Normalized training throughput using GPipe with different # of partitions K and different # of micro-batches M on TPUs. Performance increases with more micro-batches. There is an almost linear speedup with the number of accelerators for Transformer model when $M \gg K$. Batch size was adjusted to fit memory if necessary.

TPU	AmoebaNet			Transformer		
$K =$	2	4	8	2	4	8
$M = 1$	1	1.13	1.38	1	1.07	1.3
$M = 4$	1.07	1.26	1.72	1.7	3.2	4.8
$M = 32$	1.21	1.84	3.48	1.8	3.4	6.3

GPipe's Limitations

Limitations

1. GPipe Pipeline最重要的limitation是Low pipeline utilization。虽然GPipe相对原始Model Parallelism提升了utilization，但是还是有着较大的bubble。
2. GPipe的切分算法(partition algorithm)非常简单：一是假设模型的序列化（对于拓扑复杂的模型，先拓扑排序转成序列再只在序列上切不一定优）；二是没有考虑通信代价，只考虑了一维per-layer目标。

后续关注切分算法的改进主要从两个方面

1. 从依赖序列化到在**general DAG**上的切分
   a. Efficient Algorithms for Device Placement of DNN Graph Operators Tarnawski@NIPS20 在DAG上identify了optimal sub-structure做DP
2. 综合考虑计算&通信&memory开销来**优化Training Throughput**，特别是当联合考虑其它维度data & intra-op model parallelism时怎么做切分，就更难一点了 (Hybrid parallelism)
   a. 各种auto parallelism文章

后续的Pipeline Parallelism改进

Pipeline设计优化目标/限制: 利用率(Bubble ratio), 内存占用, 同步更新语义限制

后续关注pipeline parallelism的工作从几个方面改进

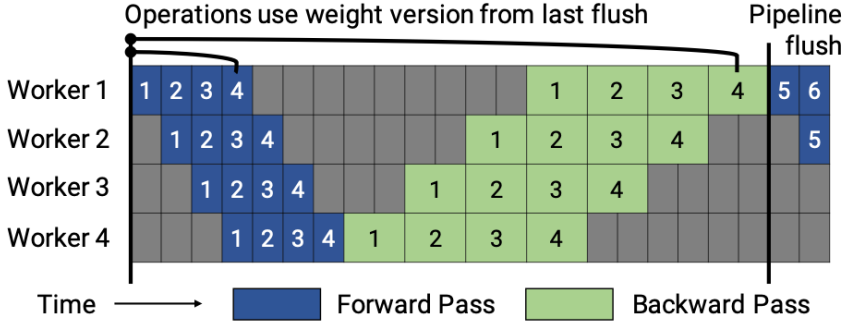
1. 放松Synchronized约束 → 引入*statistical efficiency*、memory的开销，但是大大提升utilization（基本可以用满）
   a. PipeDream@SOSP19
   b. 进一步，PipeDream-2BW@ICML21，PipeMare@MLSys21，2GW@ICLR22从缓解async pipeline的memory、statistical efficiency上面提出了一些方法
2. 更好的pipeline schedule方式 → 1F1B，哪怕保持synchornize没有bubble的提升，也能可以降低memory开销
   a. PipeDream-2BW-Flush@ICML21
3. 更灵活的partition assignment + 相应的pipeline schedule设计 → 引入通信或者内存开销，提升utilization
   a. Megatron-LM@SC21 更多通信 interleaved assignment
   b. Chimera@SC21 更多内存/通信，bidirectional pipeline
4. 在特定应用场景下，找到除了microbatch新的pipeline维度 → 语言模型的时间（或者说Token）维度（更细粒度）可以pipeline起来
   a. TeraPipe@ICML21

schedule

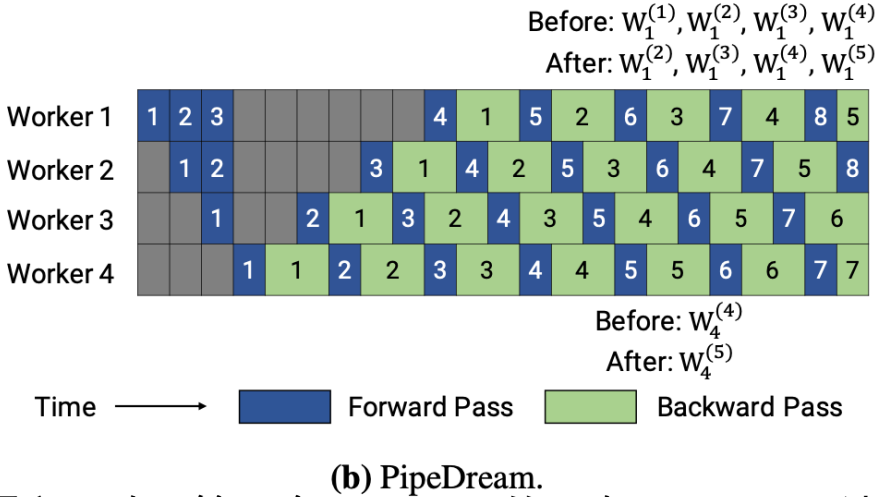
assignment+schedule

pipeline-parallel dimension

Async Pipeline: PipeDream@SOSP'19



(a) GPipe.



(b) PipeDream.

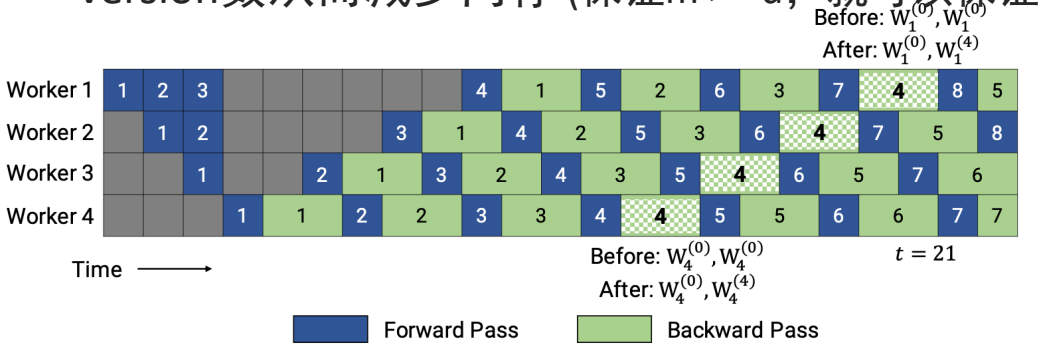
- GPipe是同步流水线(synchronized)，跟SGD语义一致：等一个minibatch的所有microbatch计算完gradient更新完权重，再进行下一个minibatch的计算，导致流水线利用率低
- PipeDream是异步流水线(async)，没有flush或者分microbatch
  - 在稳定状态下，每个worker交替计算它负责的stage的1F&1B (只是在不同minibatch)
  - 不能完全async(例如某个stage的gradient算出来马上更新weight)，为了保证一定的statistical efficiency, 需要存n份versioned model weights (n是in-flight batch数=d, stage数)

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t)}, w_2^{(t)}, \ldots, w_n^{(t)})$$

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t-n+1)}, w_2^{(t-n+1)}, \ldots, w_n^{(t-n+1)})$$

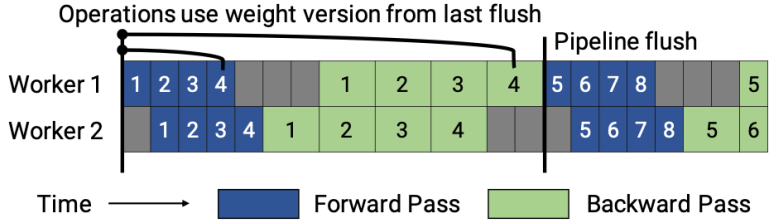
Reduce Memory: PipeDream-2BW@ICML'21

- PipeDream-2BW@ICML'21
  - 从n份versioned weights到2份：不是PipeDream里的每个batch都更新，而是过m个batch才用累计梯度更新一份新权重，降低更新频次就可以减少PD里的weight version数从而减少内存 (保证m >= d，就可以保证只存两份)

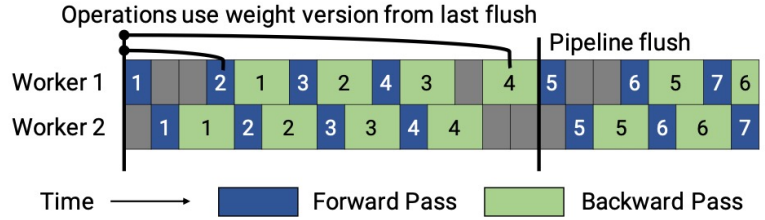


从第9个microbatch (第三个minibatch)可以开始使用$W_1^{(4)}$做forward

GPipe



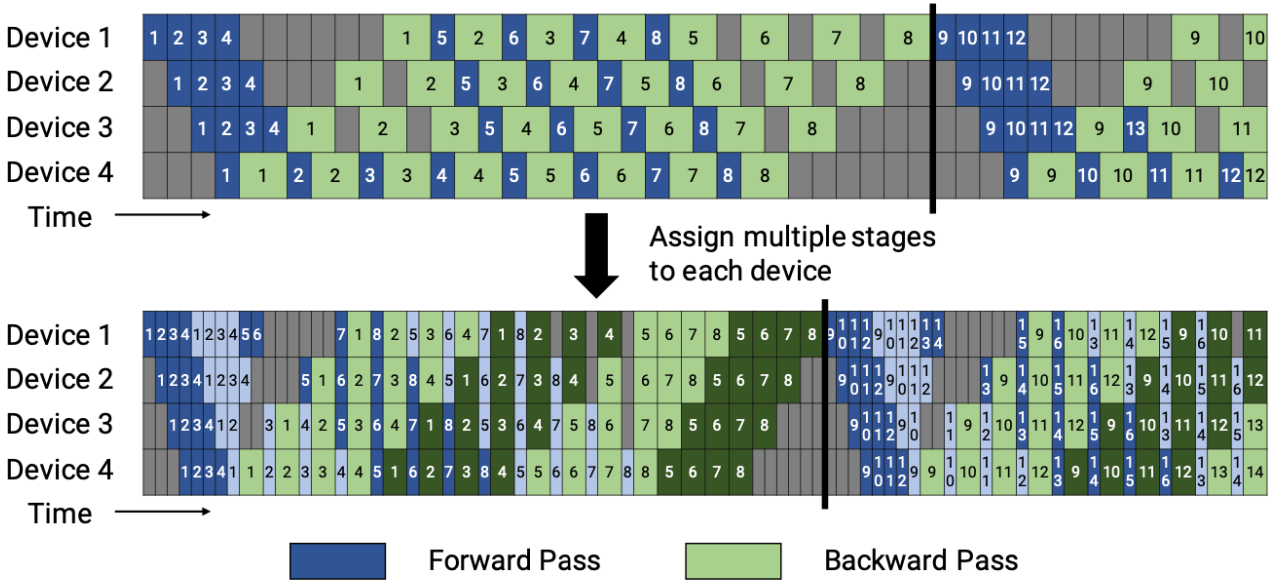
PipeDream-2BW-Flush同步流水线变种 (1F1B)



提升Sync Pipeline利用率

Megatron-LM@SC'21 by NVIDIA & Stanford & Microsoft Research

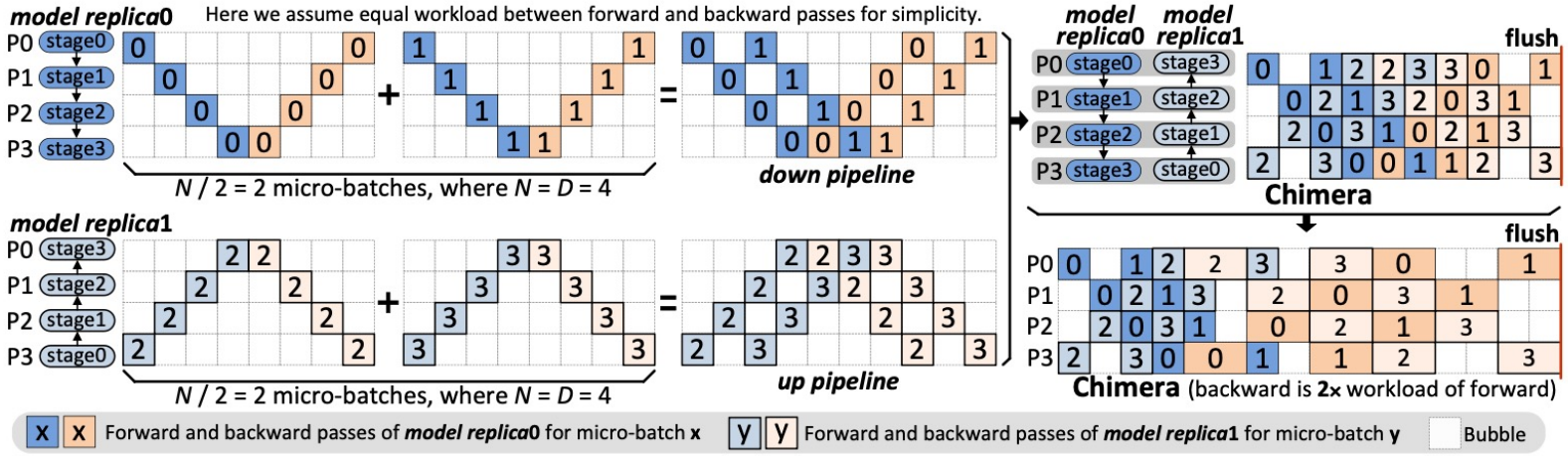
- Stage切的更细, interleave的assign给device，也就是说一个device不再限制只assign一个continuous的stage，而是assign两个continuous stage
- 因为stage切得更细了，bubble比例降低v倍，自然通信代价也提升了v倍



提升Sync Pipeline利用率

Chimera@SC'21 by ETH

- 把两个流水线（复用同一组devices，相反方向assign）拼起来提升利用率，两个流水线分别处理不同的microbatches
- 也就是说每个device会需要计算两个stage（一个是前向流水线assign的，一个是反向流水线assign），如下图所示
- 每个device上的权重存储提升了2倍



映射策略介绍

- 模型并行-流水线并行 Inter-Op Model Parallelism
  - GPipe@NIPS'19 by Google
  - PipeDream@SOSP'19 by Microsoft&CMU&Stanford
  - PipeDream-2BW@ICML'21 by Microsoft&Stanford
  - Chimera@SC'21 by ETH, Megatron-LM@SC'21 by NVIDIA&Stanford&Microsoft
- 模型并行-算子并行
  - GShard@ICLR'21 -> GSPMD@arXiv21 by Google
- 数据并行改进
  - Zero@SC'20 by Microsoft
  - Zero-Offload@ATC'21 by Microsoft

算子并行

- 算子内并行：对一个算子进行切分，切分后的sub算子在各个device上计算，聚合结果是整个算子的计算结果
  - 决定算子并行映射方式，就是决定operands和results tensor的sharding方式（怎么切分split/tiling或怎么重复replicate）
  - 算子间operand/results的sharding方式不match，可能会引入一些resharding / redistribution开销

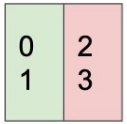
算子并行: GSPMD

- 生成SPMD程序，所有partition都执行一样的代码
- 基于用户的部分sharding annotation实现半自动化算子并行

Sharding三种类型

1. Replicated:
   - Every partition has full data

2. Tiled:
   - Every partition has one ¼ data
   - Device order can be specified
   - Device mesh tensor [[0,2],[1,3]]
   - mesh_split(t, device_mesh, dims_mapping=[0, 1])



3. Partially Tiled:
   - Replicated in subgroups
   - Each subgroup has a different subset of data
   - Device mesh tensor [[[0,1],[2,3]]], inner-most dim is replication
   - mesh_split(t, device_mesh, dims_mapping=[-1, 0])

推理示例

bf = matmul(bd, df)

```
bd = mesh_split(bd, mesh, [0, -1])
df = mesh_split(df, mesh, [-1, 1])
```

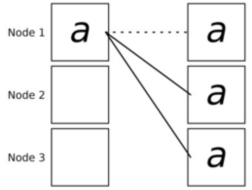
自动推理输出bf的sharding为
mesh_split(bf, mesh, [0,1])

- 实现了两个XLA compiler pass
  - sharding completion: 迭代根据heuristic/经验推理补全所有tensor的sharding
  - per-operator partitioning：生成SPMD 计算code， 插入必要的集合通信原语做activation、gradients的聚合，以及在必要的时候做resharding

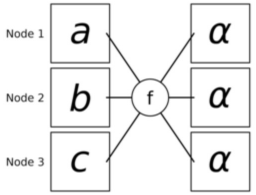
补充简介：集合通信

➢集合通信原语 (collective operations)

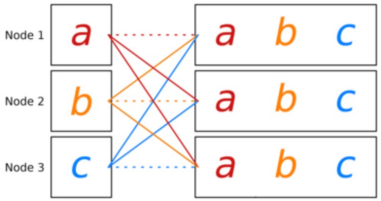
- Broadcast



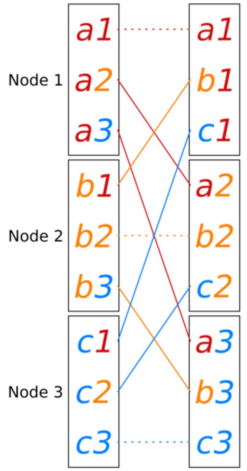
- AllReduce



- AllGather



- AllToAll



- ReduceScatter: reduce后接scatter
  - a,b,c -> f -> α -> 切成三块

映射策略介绍

- 模型并行-流水线并行 Inter-Op Model Parallelism
  - GPipe@NIPS'19 by Google
  - PipeDream@SOSP'19 by Microsoft&CMU&Stanford
  - PipeDream-2BW@ICML'21 by Microsoft&Stanford
  - Chimera@SC'21 by ETH, Megatron-LM@SC'21 by NVIDIA&Stanford&Microsoft
- 模型并行-算子并行
  - GShard@ICLR'21 -> GSPMD@arXiv21 by Google
- 数据并行改进
  - Zero@SC'20 by Microsoft
  - Zero-Offload@ATC'21 by Microsoft

数据并行改进: Zero@SC'20 & Zero-Offload@ATC'21

➢Zero@SC'20

- 普通数据并行不能降低per-device权重内存消耗，因为整个模型需要复制(replicate)在不同device上, 每个training step在backward的时候会把不同worker上所有层的梯度all-reduce到所有的worker上
- Zero-DP改进了DP的内存消耗: optimizer states和gradients这种东西可以按“model parallelism”的思想participate，每个worker只存一部分，gradient backward的时候要的通信量还变少了。最后，在gradient算&传完+update之后，需要一次对weight的all-gather。**通信量不变、内存消耗减小**
- 进一步，还可以把weight也切开，前向/后向计算要用到其它device上的引入通信原语传。相比普通DP1.5x通信开销，**通信换内存**

➢Zero-Offload@ATC'21

- 用上CPU的内存和计算能力！
- 把gradients、optimizer states放到CPU上，把optimizer computation用CPU计算 (memory bottleneck的主要来源), 参数和F/B计算放到GPU上

目录 Contents

1 背景和基础

2 并行映射策略介绍

3 自动并行化

4 总结

问题定义

并行种类	内存友好	通信友好	硬件利用率 (计算)
Data (DP)	低 (有一些优化e.g., Zero@SC2020)	低 (gradient sync)	高
Intra-Op Model	高	低 (op粒度的通信)	高
Inter-Op Model	高	高	因为pipeline bubble受到限制

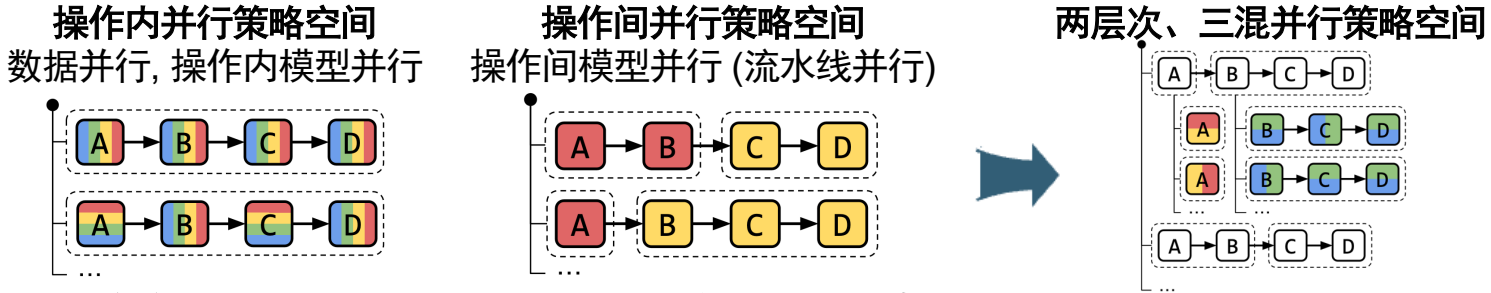
- 映射/优化策略着有不同Trade-off, 适用于不同模型&操作, 需混合达到最优效果
- 混合映射策略中, 决策间互相影响

为了达到最优效果，混合映射策略的优化空间是指数级的

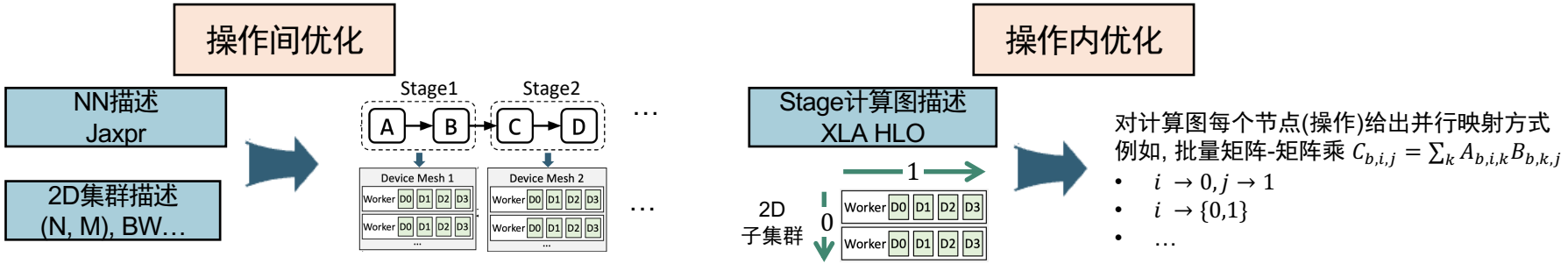
自动并行化：给定NN模型和集群描述, 考虑内存约束, 优化训练吞吐, 自动化决定并行映射&优化策略 (即如何**切分计算，**如何**映射切分后计算到集群上**)

Alpa[1]: 贡献

1. 将数据并行作为操作内并行的一种，把并行策略分为操作间和操作内两类，构建操作间 – 操作内的两层次并行策略空间



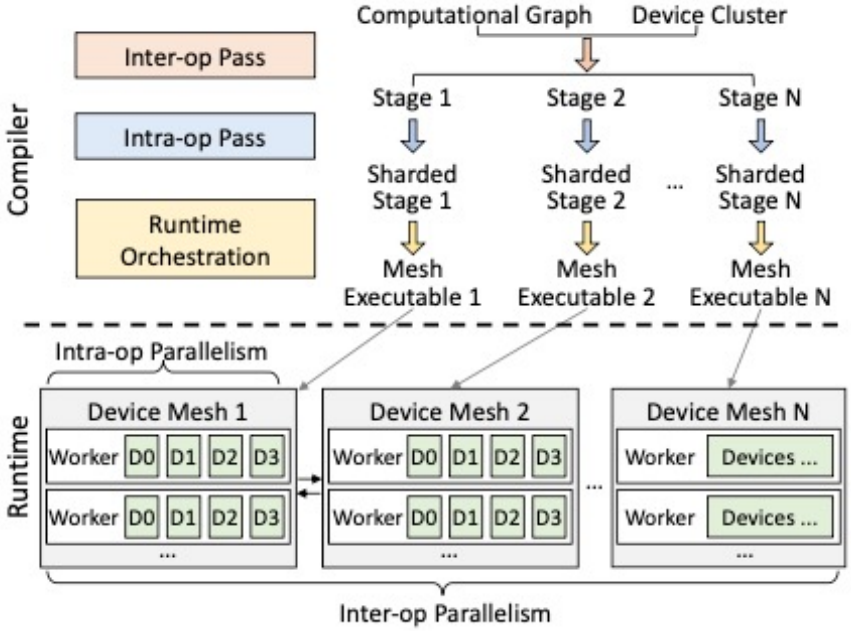
2. 将三混空间的优化问题划分为两个层次的规模更小的子问题 (缩减优化空间: 操作内并行对通信要求更高, 应该assign到物理更近的submesh)



[1] Zheng et al., Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning, arXiv'22.01.

Alpa: 贡献

3. Sophiscated的实现&优化，并且用户易用性高



1.编译时：使用Jax来trace模型，inter-op pass在jaxpr上进行，因为inter-op需要forward&backward这种NN语义信息；intra-op pass在XLA HLO上以NN-agnostic op为粒度进行。编译对每个stage得到一个不同的executable。

2.运行时： Ray来负责在多个服务器上起worker，执行编译得到的executable。使用XLA runtime来实际上GPU做计算。

3.写了个通信lib，做cross-mesh communication加速

Alpa: 用户API

3. Sophiscated的实现&优化，并且用户易用性高

```
# One-line auto-parallelization:
# 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)
```

Alpa: 用户API

```
1  ray.init(address="auto")
   jax.config.update('jax_platform_name', 'cpu')

   device_mesh = DeviceCluster().get_virtual_physical_mesh()
2  set_parallelize_options(devices=device_mesh,
                           strategy="pipeshard_parallel",
                           pipeline_stage_mode="auto_gpipe",
                           logical_mesh_search_space="default"
   )

   layer_num = 6
   state, batch = create_train_state_and_batch()

   def train_step(state, batch):

3      @automatic_layer_construction(layer_num=layer_num)
       def loss_func(params):
           out = state.apply_fn(params, batch['x'])
           return jnp.mean((out - batch['y'])**2)

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

   # Serial execution
   expected_state = train_step(state, batch)

4  # Parallel execution
   global_config.num_micro_batches = 2
   parallel_train_step = parallelize(train_step)
   actual_state = parallel_train_step(state, batch)

   # Check results
   assert_allclose(expected_state.params, actual_state.params)
```

1. Ray初始化，拿到ray cluster大小信息为device_mesh
2. 设置Options
   1. strategy表示支持什么类型并行映射: pipeshard是三混, shard是二混
   2. pipeline_stage_mode表示pipeline映射里stage切分是怎么做的: auto_gpipe (alpa), uniform_layer_gpipe, manual_gpipe
3. 文中Performance optimization #2: operator clustering
   - 流程：make_jaxpr -> per-layer cost -> cluster
   - 重计算：如果传入remat_layer=True, 把每个layer的jaxpr的equation替换成primitive是jax的remat_call的equation，原equation作为参数传入，这个primitive的autodiff的transpose函数里会做重计算
   - 结果：自动把jaxpr的equations cluster成layer_num (文中L)个layer，加入pipeline marker eqn：”pipeline ”primitive, 计算语义是identity，带有mark_type ”start”, “end”。后续代码靠这个分layers
4. 把一个train step给parallelize化, parallel_train_step运行alpa算法

Alpa: 具体算法 Inter-Op Pass

输入输出: 输入Jaxpr，cluster configuration；需要决定stage个数S, 把Jaxpr分成S Stages, 把mesh分成S sliced meshes, 并给出Jax stage到sliced mesh的map关系

优化空间:先序列化整个模型DAG成op序列, $o_1, \dots o_K$。要将这个序列切成S 个连续不相交的stages，并且每个stage需要assign一个 $n \times m$ 的device mesh。需要决定S, 划分点, $\{n_i, m_i\}_{i=1,\dots S}$。

优化目标:假设PipeDream-2BW-Flush（synchronized pipeline，1F1B schedule），优化目标是每个training iteration的latency。

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

优化方法: 动态规划 (DP)

$$F(s,k,d;t_{max}) = \min_{\substack{k \le i \le K; \\ (n_s,m_s)}} \begin{cases} t_{intra}((o_k,\dots,o_i),Mesh(n_s,m_s)) \\ \quad + F(s-1,i+1,d-n_s \cdot m_s;t_{max}) \\ \quad | \, t_{intra}((o_k,\dots,o_i),Mesh(n_s,m_s)) \le t_{max} \end{cases}$$

使用操作内优化得到的cost matrix里的
将Stage $(o_k, \dots, o_i)$部署到$n_s \times m_s$子集群最优时间

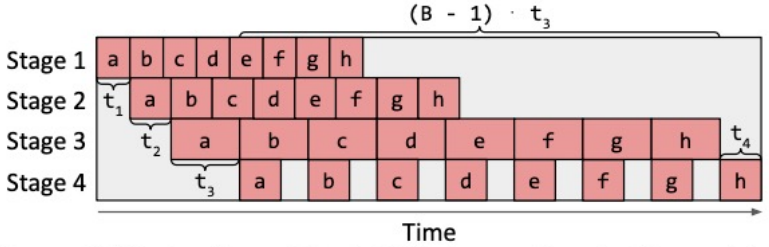


Figure 6: Illustration of the total latency of a pipeline, which is determined by two parts: the total latency of all stages ($t_1 + t_2 + t_3 + t_4$) and the latency of the slowest stage ($(B-1) \cdot t_3$).

Alpa: 具体算法 Intra-Op Pass

输入输出: 输入一个stage和mesh配置(比如2x8 GPUs)，给出这个stage里每个op节点的algorithm选择、返回t_intra

优化空间: algorithm选择是什么?

操作内并行映射空间表述

示例: 批量矩阵-矩阵乘操作的一些并行映射方式

#	并行映射	输出 sharding	输入 sharding	引入通信开销
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^0RS^1	S^0RS^1, 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	$RRS^{01}, RS^{01}R$	$all\text{-}reduce(\frac{M}{n_0}, \{0,1\})$

R: replicate
S: split

优化目标:

compute cost + communication cost

Communication cost 包括op node之间的resharding communication cost $\sum_{v,u} s_v^T R s_u$ ，和单个node里某些algorithm所需要的all-reduce的comm. cost $\sum_v s_v^T d_v$

$s_v \in \{0,1\}^{k_v}$ 独热向量，代表节点v选择的映射方式

$$\min_s \sum_{v \in V} s_v^T c_v + d_v + \sum_{(v,u) \in E} s_v^T R s_u$$

单操作计算并行引入的通信开销

操作间Resharding引入的通信开销

优化方法: 整数线性规划 (ILP)

Alpa: 结果

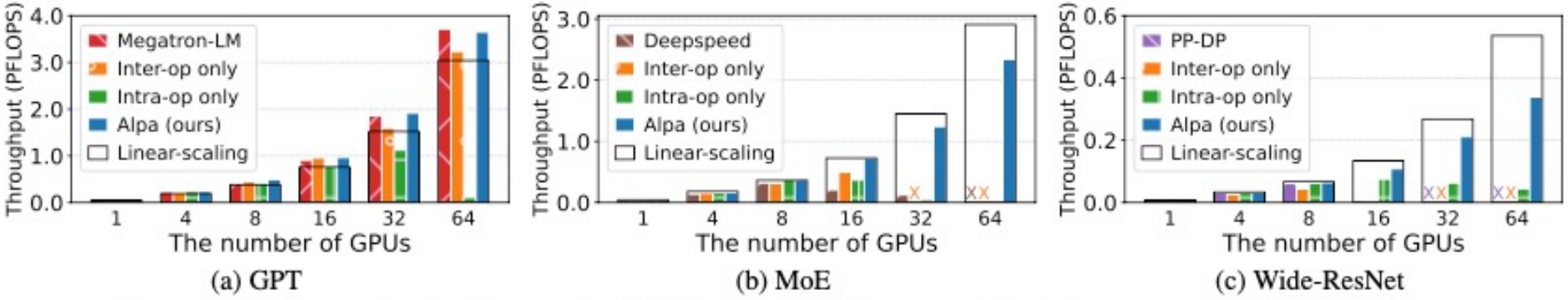


Figure 8: End-to-end evaluation results. "×" denotes out-of-memory. Black boxes represent linear scaling.

自动并行化工作一览

类型	方法	机构	并行空间	优化方法	备注
单混 双混	DPO-RL@ICML'17	Google Brain	操作间 (无流水线)	强化学习	
	FlexFlow@ICML'17	Stanford	数据, 操作内, 操作间 (无流水线)	MCMC搜索	
	PipeDream@SOSP'19	MSR, CMU, Stanford	数据, 操作间	动态规划	
	Tarnawski@NIPS'20	MSR, Amazon, MS	操作间	动态规划	
	PaSE@IPDPS'21	Baidu	数据, 操作内	图搜索+动态规划	
	PipeDream-2BW@ICML'21	Stanford, MSR, MS	数据, 操作间	暴力搜索	将搜索空间设计的非常小, 只适用规整模型 e.g., GPT-3
	Automap@NIPS'21	DeepMind	数据, 操作内	Learning-based + MCTS	
	DAPPLE@PPoPP'21	Alibaba	数据, 操作间	动态规划	
三混 (支持三种映射并行策略)	Piper@NIPS'21	MSR	数据, 操作内, 操作间, 其它优化	动态规划	操作内并行、其它优化是黑盒建模的选择，需要预先设计好作为输入
	DistIR@EuroSys'21	Stanford, Microsoft	数据, 操作内, 操作间	暴力搜索	
	Alpa@OSDI'22	UCB, AWS, Google	数据, 操作内, 操作间	动态规划+整数线性规划	

目录 Contents

1 背景和基础

2 并行映射策略介绍

3 自动并行化

4 总结

总结

- 三种并行策略
  - 流水线并行
    - GPipe: 同步流水线，数据minibatch进一步切分microbatch，流水线并行
    - PipeDream & PipeDream-2BW & 2GW: 异步流水线，牺牲一定语义，高流水线利用率。2BW和2GW减小异步流水线内存占用 (weight stashed versions)
    - PipeDream-2BW-Flush: 同步流水线，利用率与GPipe相同，但其1F1B schedule使其比GPipe的内存占用小
    - Megatron-LM & Chimera: 同步流水线，灵活化stage-device assignment对应设计schedule，达到高流水线利用率
  - 算子并行
    - GSPMD: 半自动生成算子并行的SPMD程序，根据少量sharding annotation做sharding completion和per-operator partitioning (per-device compute, insert communication primitives, handle irregular pattern)
  - 数据并行改进
    - Zero: 做weight和optimizer state拆分，通信换内存
    - Zero-Offload: 一些低计算高内存操作(optimizer computation)放在CPU上

总结

- 自动并行化
  - 问题定义: 给定NN模型和集群描述, 考虑内存约束, 优化训练吞吐, 自动化决定并行映射&优化策略 (即如何**切分计算**，如何**映射切分后计算到集群上**)
  - 寻优空间；寻优方法；加速寻优/减小寻优空间的heuristics
  - Alpa: 三混寻优空间，外层pipeline，内层算子+数据；外层DP、内层ILP；分层的mesh assign空间设计减少了寻优空间大小，operator clustering等策略做加速
- 其它
  - 激活重计算

Thanks and Q&A

(其它补充) IR格式: Jaxpr

```
jaxpr ::= { lambda Var* ; Var+.
            let Eqn*
            in  [Expr+] }
```

- Var*: constvars
- Var+: invars，input of traced python function
- Eqn*: a list of equations, 中间变量计算和定义
- Expr+: a list of output atomic expressions (literals or variables)

示例，三层MLP构成的一个forward stage的Jaxpr

```
ipdb> p jax_pipeline_stages[0].closed_jaxpr()
{ lambda ; a:f32[16,16] b:f32[16] c:f32[16,16] d:f32[16] e:f32[16,
    h:f32[16,16] i:f32[16] j:f32[16,16] k:f32[16] l:f32[16,16] m:
      mark_type=start
      name=0
    ] a b c d e f g
    o:f32[8,16] = dot_general[
      dimension_numbers=(((1,), (0,)), ((), ()))
      precision=None
      preferred_element_type=None
    ] n h
    p:f32[1,16] = reshape[dimensions=None new_sizes=(1, 16)] i
    q:f32[8,16] = add o p
    r:f32[8,16] = dot_general[
      dimension_numbers=(((1,), (0,)), ((), ()))
      precision=None
      preferred_element_type=None
    ] q j
    s:f32[1,16] = reshape[dimensions=None new_sizes=(1, 16)] k
    t:f32[8,16] = add r s
    u:f32[8,16] = dot_general[
      dimension_numbers=(((1,), (0,)), ((), ()))
      precision=None
      preferred_element_type=None
    ] t l
    v:f32[1,16] = reshape[dimensions=None new_sizes=(1, 16)] m
    w:f32[8,16] = add u v
    x:i32[] = add 0 0
    y:f32[8,16] z:f32[8,16] ba:f32[8,16] bb:i32[] = pipeline[
      mark_type=end
      name=0
    ] w t q x
  in (y, z, ba, bb) }
```

F **foxfi-ning** Apr 15 (edited)
n: 一个micro batch (mbs=8)的输入数据
F Reply...

F **foxfi-ning** Apr 15 (edited)
h/j/l: 第1/2/3层weight; i/k/m: 第1/2/3层bias
F Reply...

F **foxfi-ning** Apr 15

(其它补充) IR格式: XLA HLO

- HLO module (可看作一个编译单元, 一个可运行程序)
  - 包含多个HLO computations
  - 每个module有且只有一个入口计算是entry_computation (可以看作main函数)
  - 除了入口计算以外的computation都称为nested computation
- HLO computation (可看作一个函数)
  - 包含一个 HLO instructions的DA
  - 每个computation有且只有一个root instruction

```
ENTRY main {
  a = f32[] parameter(0)
  b = f32[10] parameter(1)
  ROOT root = (f32[], f32[10]) tuple(%a, %b)
}
```