15-849 Paper Discussion

PipeDream

Presenter: Yuxuan Zheng

Feb. 23, 2022

PipeDream: Generalized Pipeline Parallelism for DNN Training

Deepak Narayanan[‡*], Aaron Harlap[†*], Amar Phanishayee[⋆],
Vivek Seshadri[⋆], Nikhil R. Devanur[⋆], Gregory R. Ganger[†], Phillip B. Gibbons[†], Matei Zaharia[‡]
[⋆]*Microsoft Research* [†]*Carnegie Mellon University* [‡]*Stanford University*

Carnegie Mellon University

Executive Summary

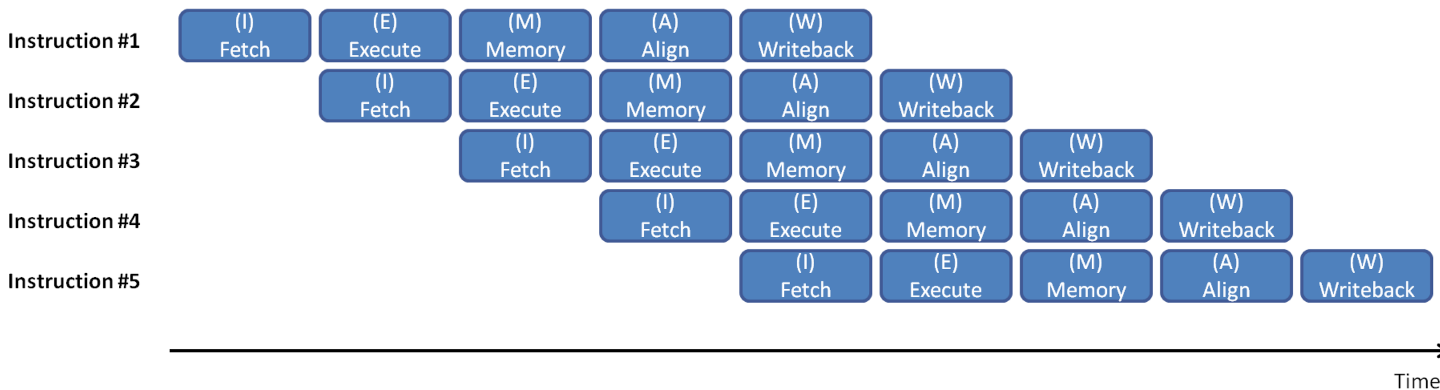
1. Pipeline parallelism is another parallelism strategy that overlaps computation and communication of workers.
2. PipeDream provides a more flexible and efficient pipeline parallelism approach compared to GPipe.
3. PipeDream combines model parallelism and data parallelism in a pipelined fashion to partition and schedule the training job into stages over a network of devices.
4. To support pipelining for DNN jobs, PipeDream resolves 3 challenges that traditional hardware pipeline doesn't see.
    1. Dynamic Programming algorithm to partition workload evenly
    2. Static scheduling policy to coordinate forward and backward pass
    3. Weight stashing to improve statistical efficiency
5. Able to achieve around 2x higher throughput than GPipe; 1.9x faster than FlexFlow on AlexNet

Background

1. Data parallelism: synchronous all-reduce
    1. High communication overhead due to all-to-all communication
    2. Need to fit the entire model in a single worker
    3. Waste memory due to duplication
2. Model parallelism: split operators to different workers
    1. Under-utilize compute resources
    2. Programmer needs to decide how to partition the model
3. Hybrid intra-batch parallelism
4. Pipeline parallelism: inter-batch
    1. Try to find optimization opportunity between mini-batch iterations of a training loop

Background

Traditional Hardware Pipeline



Carnegie Mellon University

Problem of Model Parallelism and GPipe

Idle device

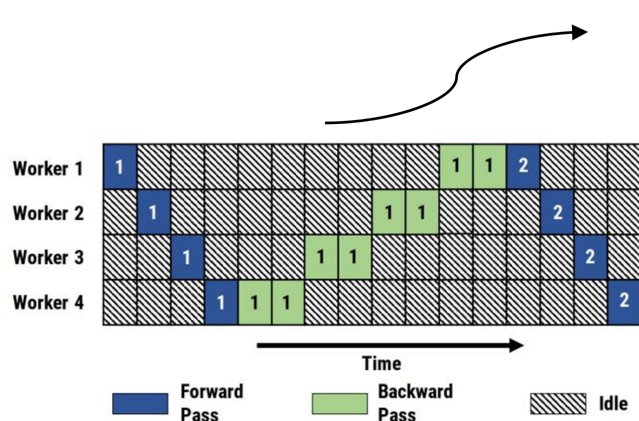


Figure 2: Model parallel training with 4 workers. Numbers indicate batch ID, and backward passes takes twice as long as forward passes. For simplicity, we assume that communicating activations/gradients across workers has no overhead.

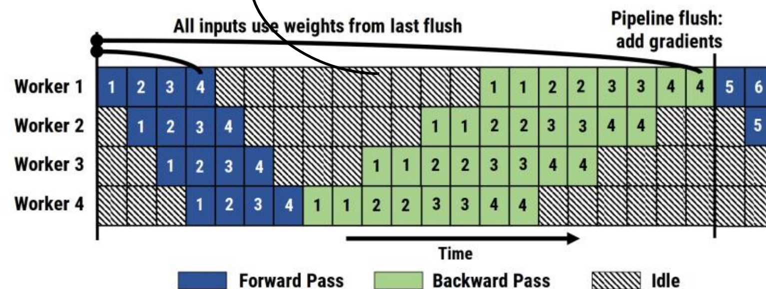


Figure 3: GPipe's inter-batch parallelism approach. Frequent pipeline flushes lead to increased idle time.

Problem of GPipe

1. Assumes a partitioned model as input
    1. No auto-partition algorithm
2. Splits a mini-batch into micro-batches
3. May suffer from reduced hardware efficiency due to re-computation overheads and frequent pipeline flushes

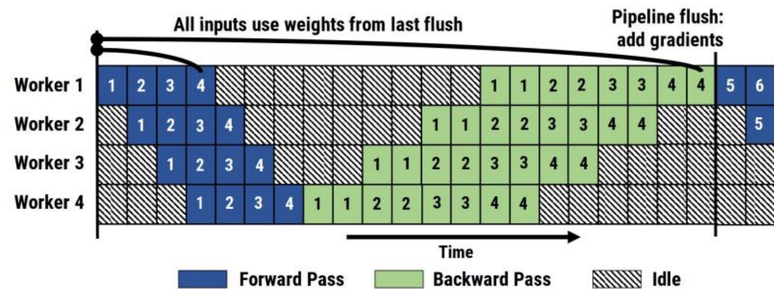


Figure 3: GPipe's inter-batch parallelism approach. Frequent pipeline flushes lead to increased idle time.

Carnegie Mellon University

Main Idea of PipeDream

1. Partition the layers of a DNN model into multiple stages
    1. Each stage has a consecutive set of layers
    2. Each stage is distributed to a worker (GPU, etc.)
2. Multiple mini-batches are fed into the stages one after another
3. Each worker can choose to run:
    1. The forward pass given the activation from the previous stage
    2. The backward pass given the gradient from the stage after itself
4. Once the computation is done, the worker sends the result (activation or gradient) to the next worker who should process it
5. At the same time, the worker starts to compute the next mini-batch (either forward or backward)

Figure 4: An example PipeDream pipeline with 4 workers, showing startup and steady states. In this example, the backward pass takes twice as long as the forward pass.

Main Idea of PipeDream

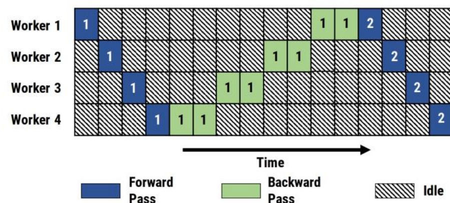


Figure 2: Model parallel training with 4 workers. Numbers indicate batch ID, and backward passes takes twice as long as forward passes. For simplicity, we assume that communicating activations/gradients across workers has no overhead.

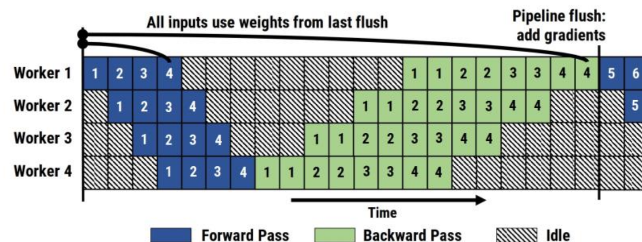


Figure 3: GPipe's inter-batch parallelism approach. Frequent pipeline flushes lead to increased idle time.

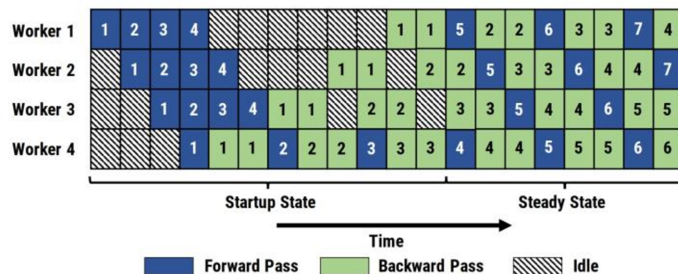
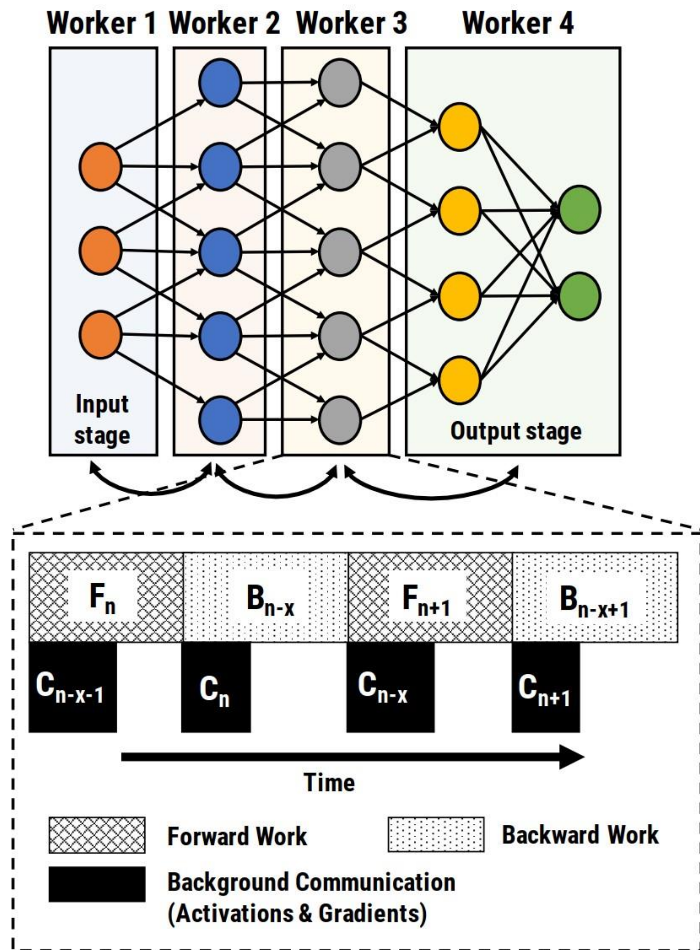


Figure 4: An example PipeDream pipeline with 4 workers, showing startup and steady states. In this example, the backward pass takes twice as long as the forward pass.

Carnegie Mellon University

Why Pipelining Is Good?

1. Less communication than data parallelism
   1. No need to aggregate and broadcast the gradients
   2. Only communicates a subset of gradients/activations to only a single worker
   3. All-to-all -> peer-to-peer
2. Overlaps computation and communication
   1. Communication of current results overlaps next minibatch's computation
   2. Because communication and computation have no dependency



Challenges

1. Partition DNN layers into the stages evenly
    1. Throughput depends on the slowest stage in pipeline
    2. Allow a stage to be replicated (data parallelism)
    3. Has optimal sub-problem structure -> use DP
2. Schedule forward and backward computation on the same worker
    1. DNN training needs forward and backward passes
    2. Workers need to decide what to run and how to coordinate replicated workers for data parallelism
3. Mismatch between the version of weights used in the forward and backward pass
    1. Fundamental issue due to pipelining

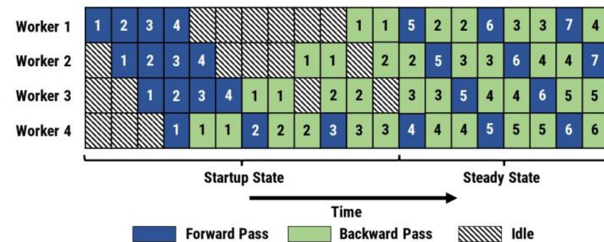


Figure 4: An example PipeDream pipeline with 4 workers, showing startup and steady states. In this example, the backward pass takes twice as long as the forward pass.

Carnegie Mellon University

Challenge 1: Work/Stage Partition

Profile the training job and find the optimal partition with DP

Goal: Get partitioning of layers, replication factor, optimal num of in-flight mini-batches

Figure 7: An example 2-level hardware topology. Solid green boxes represent GPUs. Each server (dashed yellow boxes) has 4 GPUs connected internally by links of bandwidth B_1; each server is connected by links of bandwidth B_2. In real systems, $B_1 > B_2$. Figure best seen in color.

Carnegie Mellon University

Challenge 1: Work/Stage Partition

Goal: Get partitioning of layers, replication factor, optimal num of in-flight mini-batches

find $A^L(0 \rightarrow N, m_L)$

Profile the training job and find the optimal partition with DP

Total time of a stage for a single input from layer i to j for both forward and backward passes, replicated over m workers with bandwidth B_k

$$T^k(i \rightarrow j, m) = \frac{1}{m} \max \begin{cases} A^{k-1}(i \rightarrow j, m_{k-1}) \\ \frac{2(m-1)\sum_{l=i}^{j} |w_l|}{B_k} \end{cases}$$

comp time at level k-1

comm time of data-parallel

Time of m inputs of comp and comm overlapped

Time of slowest stage in the optimal pipeline between layers i and j using m workers at device level k

comm time of activations and gradients of size a_s between layer s and s+1

$$A^k(i \rightarrow j, m) = \min_{i \leq s < j} \min_{1 \leq m' < m} \max \begin{cases} A^k(i \rightarrow s, m - m') \\ 2a_s/B_k \\ T^k(s+1 \rightarrow j, m') \end{cases}$$

Challenge 2: Forward/Backward Scheduling

1F1B-RR: static scheduling policy to run forward and backward alternatively, and handle stage replications with round-robin

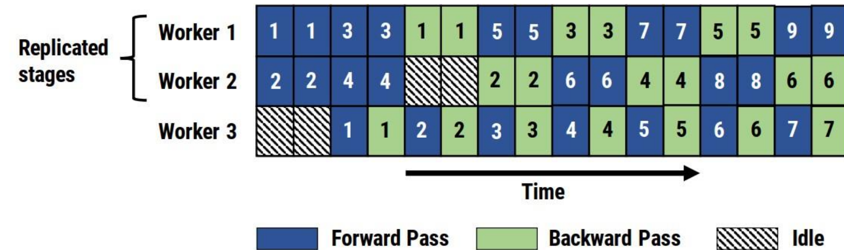


Figure 8: An example PipeDream pipeline with 3 workers and 2 stages. We assume that forward and backward passes in the first stage take two time units, while forward and backward passes in the second stage take only a single time unit. The first stage in this pipeline is replicated twice so that each stage sustains roughly the same throughput. Here, we assume that forward and backward passes take equal time, but this is not a requirement of our approach.

Carnegie Mellon University

Challenge 3: Effective Training

Weight Stashing

- Maintains multiple versions of the weights, one for each active mini-batch
- Each stage processes a minibatch using the latest version of weights available in the forward pass
- After forward pass, store the weights used for that minibatch
- The same version of weights is then used to compute in backward pass
- Only ensure the weights used in forward/backward pass are the same
- Cannot ensure the weights are the same across stages (workers)
- Doesn't increase memory overhead significantly compared to data-parallel training

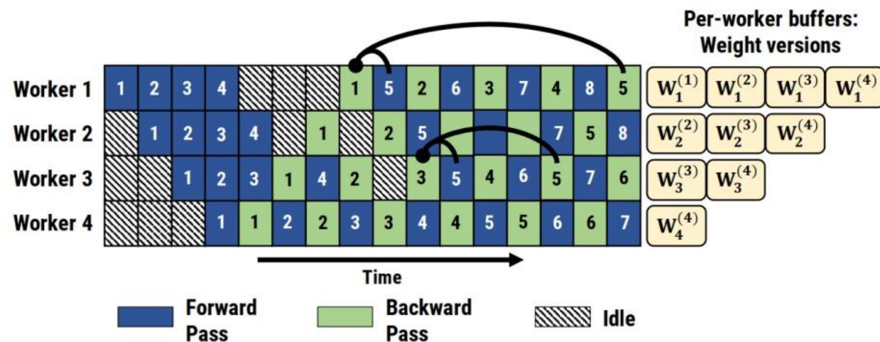
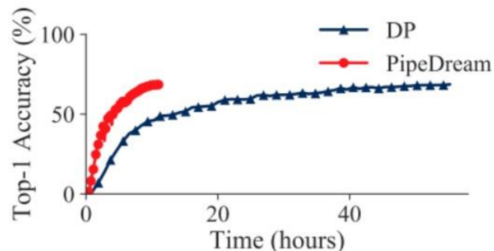


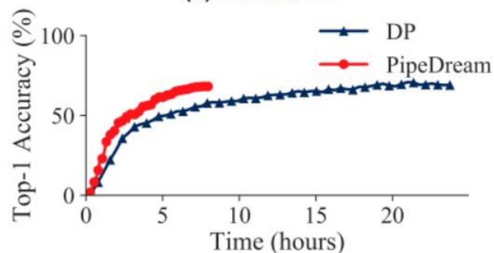
Figure 9: Weight stashing as minibatch 5 flows across stages. Arrows point to weight versions used for forward and backward passes for minibatch 5 at the first and third stages.

Carnegie Mellon University

Evaluation

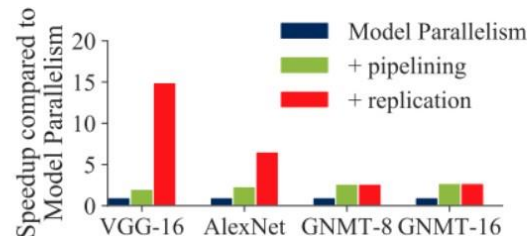


(a) Cluster-A.

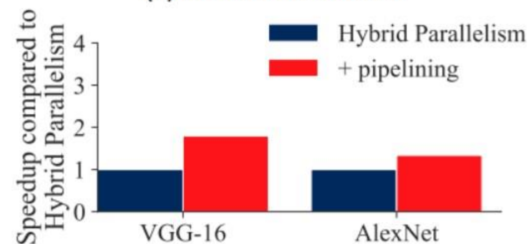


(b) Cluster-B.

Figure 10: Accuracy vs. time for VGG-16 using 16 GPUs. Each circle or triangle represents two epochs of training.



(a) Model Parallelism.



(b) Hybrid Parallelism.

Figure 14: Comparison of PipeDream (red) to non-DP intra-batch techniques for 4-GPU configurations on Cluster-A.

Evaluation

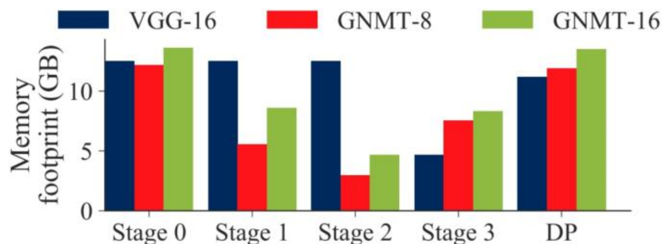


Figure 16: Memory footprint for various models using 4 GPUs. Per-GPU memory footprint is shown for data parallelism, and is identical on all GPUs.

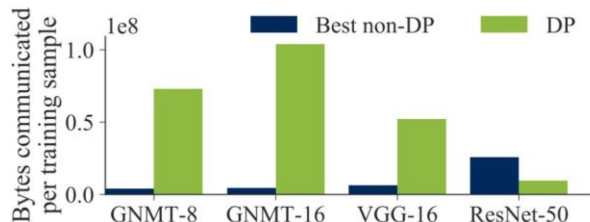
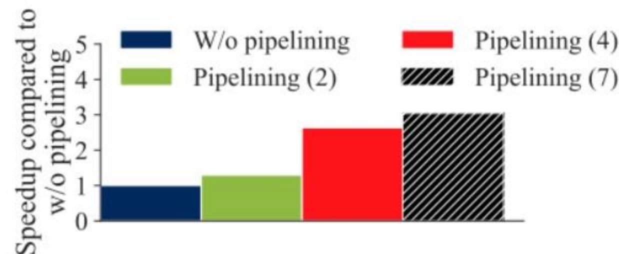
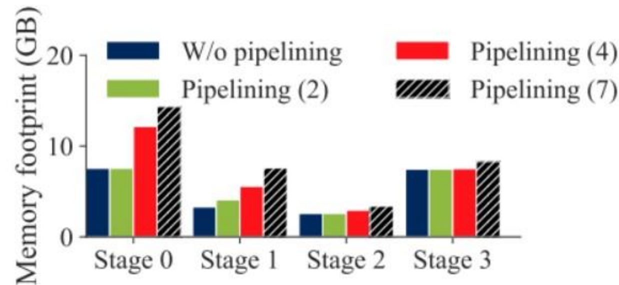


Figure 17: Bytes communicated per training sample by data-parallel (DP) and the best non-DP configurations for 4 GPUs on Cluster-A.



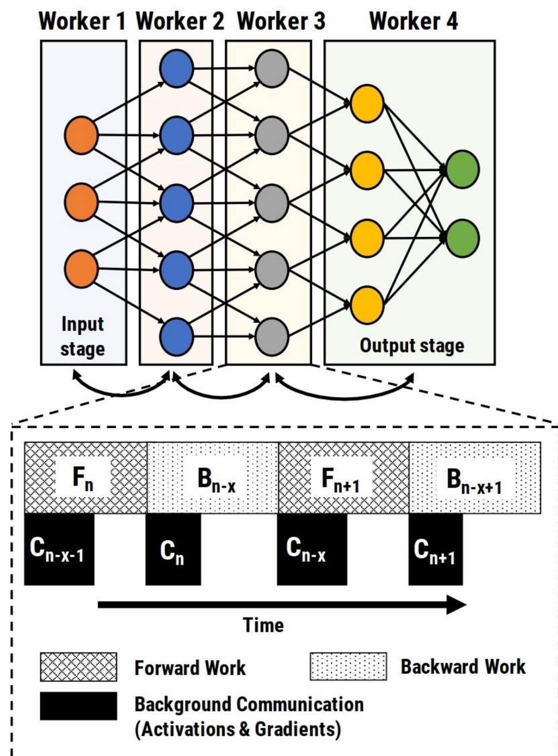
(a) Throughput.



(b) Memory Overhead.

Figure 18: Effect of pipeline depth on throughput and memory overhead for GNMT-8 on 4 V100s in Cluster-A.

Discussion of PipeDream



- Better partition computational graph
  - Profile & DP for "optimal" solution
- Better utilize resources with pipelined mini-batches
- Overlap computation & communication
- Combine data & model & pipeline parallelism "naturally"

Carnegie Mellon University

Discussion Questions

1. Do you see any **possible improvement of the work/stage partition algorithm**? Especially, consider what factors are not included in the current DP formulation?
2. Could you discuss the similarities and differences between **PipeDream and Spark-style** multi-worker computation?
3. What are some advantages & limitations of the "**profile**-then-partition" approach? Is there any improvement potential for the "**1F1B-RR**" scheduling?

Thank You!

GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

---

Yanping Huang, et al.

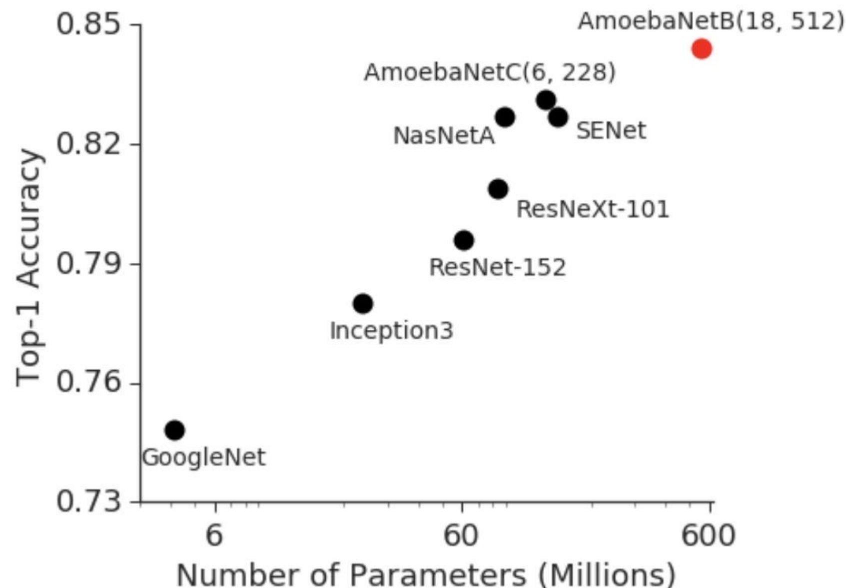
NIPS 2019

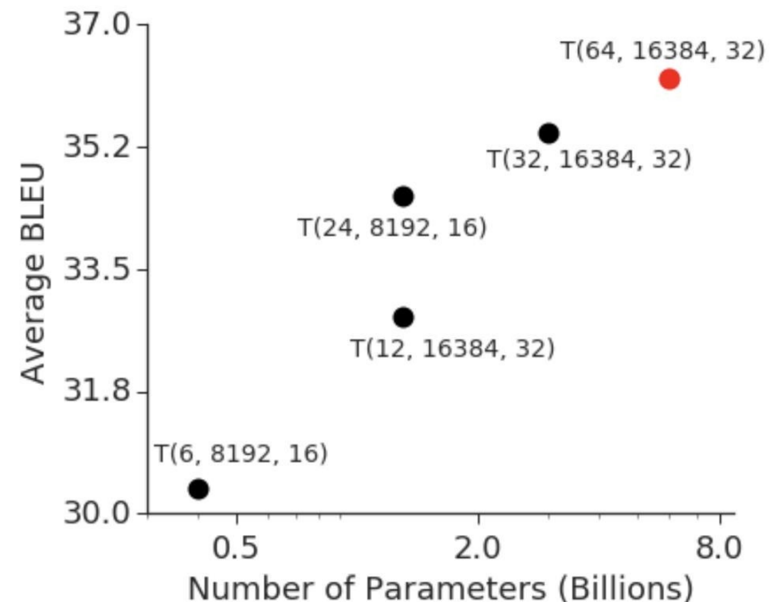
Presenter: Weizheng Xu

Feb.23. , 2022

Introduction: Becoming more larger

- **Bring remarkable quality improvements to several fields**
  - Trend: **accuracy improvements** on ImageNet with increase in **model capacity**
  - Similar phenomenon be overserved in **context of natural language processing**

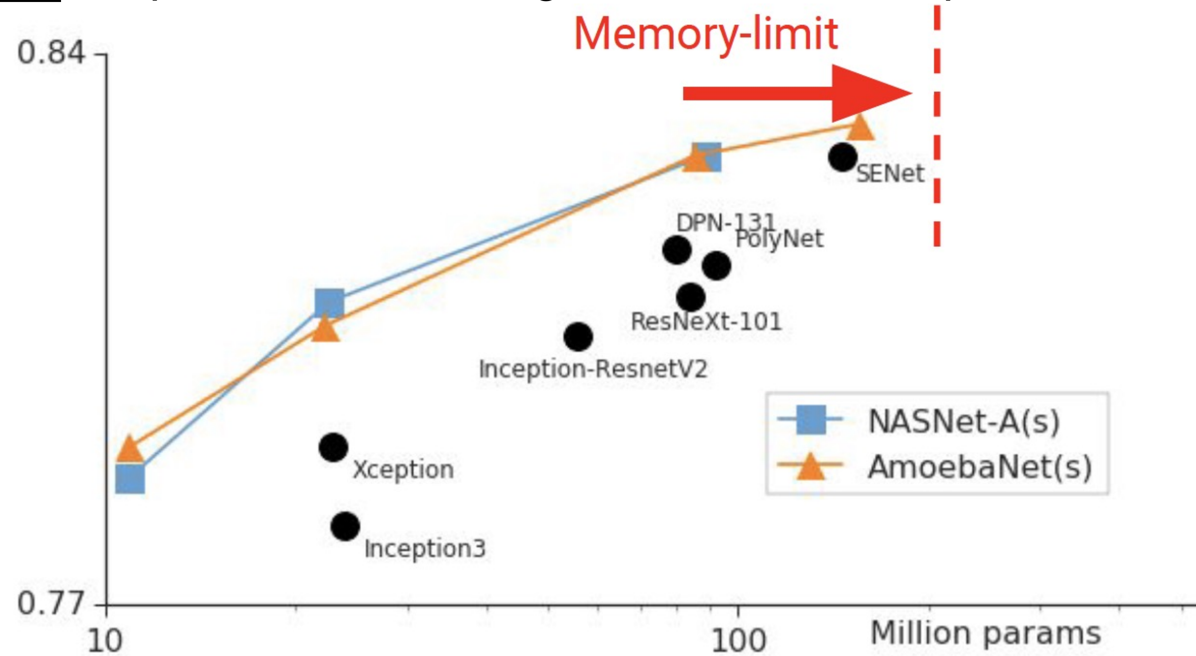




[1] https://www.synopsys.com/designware-ip/technical-bulletin/building-efficient-deep-learning-dwtb_q318.html

Introduction: Becoming more larger

- **Practical challenges with larger models**
  - HW constraints (**memory limitation**, communication bandwidth on accelerator)
  - **Dividing larger models** into partitions and assignment of different partitions to different accelerators[1]

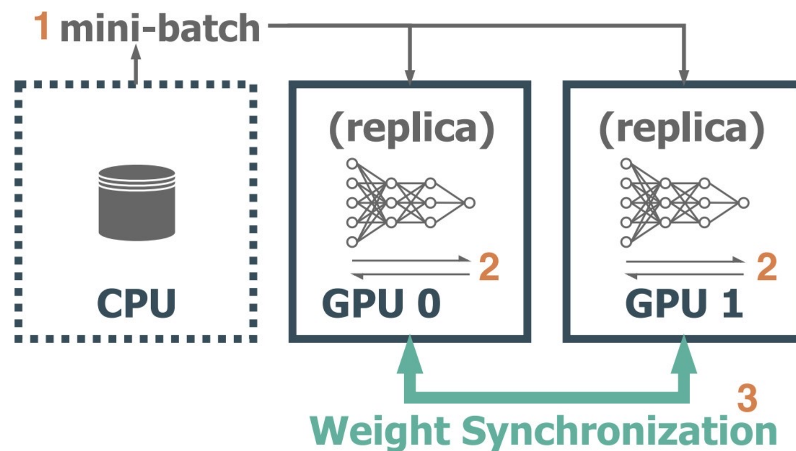


35x bigger models → 6% accuracy increase

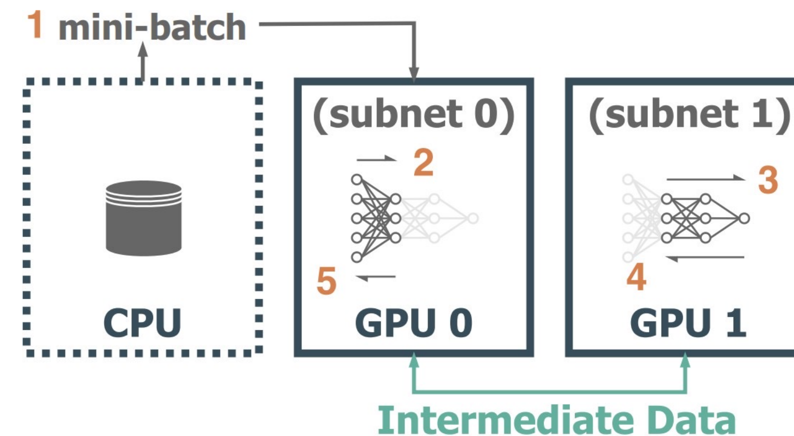
[1] https://www.microsoft.com/en-us/research/blog/pipedream-a-more-effective-way-to-train-deep-neural-networks-using-pipeline-parallelism/

Introduction: Becoming more larger

- **Model parallelism**
  - Hard to design and implement **efficient model parallelism algorithm**[2]
  - **Using multiple GPUs,** Each GPU is responsible for weight updates of assigned model layers
  - Architecture and task-specific and Increasing demand for reliable and flexible infrastructure



(a) Data Parallelism.

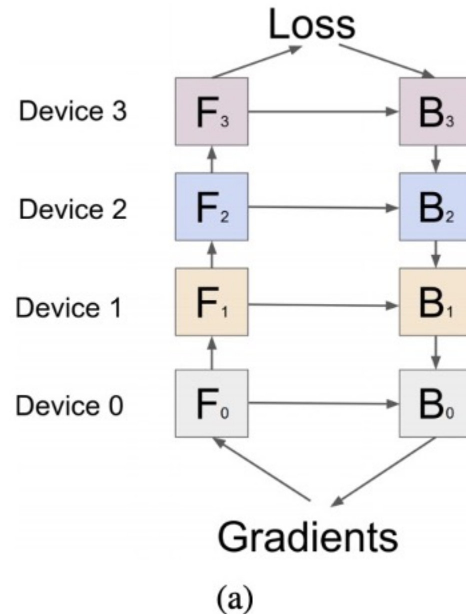


(b) Model Parallelism.

[2] https://arxiv.org/pdf/1809.02839.pdf

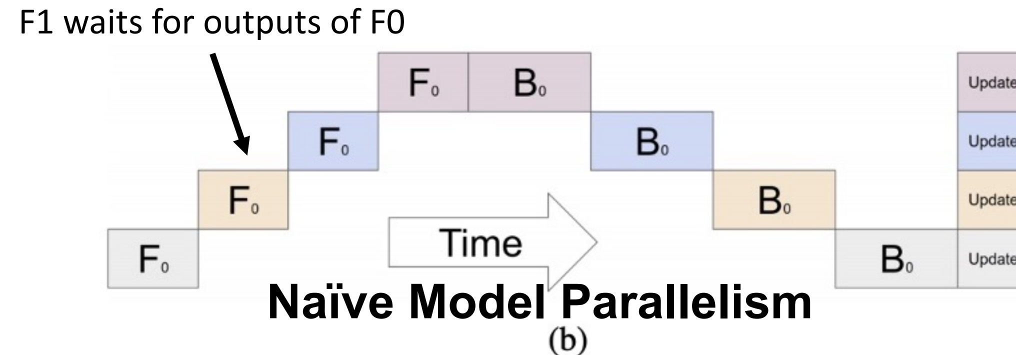
Introduction: Model Parallelism

- ❑ Scaling arbitrary DNN architecture by partitioning models
- ❑ Layers can be partitioned into cells and each cell is then placed on separate accelerator.
  - (a) Sequential partitioning
  - (b) Naïve Model Parallelism (severely under-utilized)



(a)

F1 waits for outputs of F0



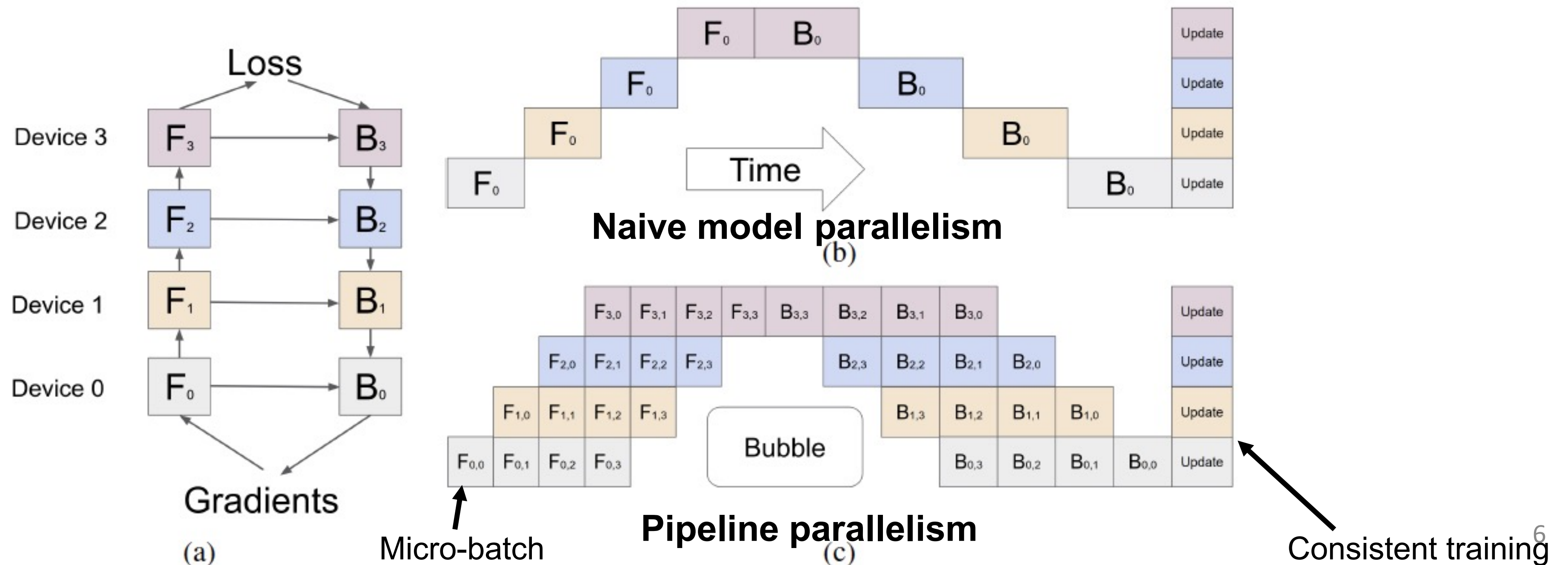
Naïve Model Parallelism
(b)

Only one accelerator is active when the model is distributed across the accelerators
Resource is severely under-utilized!

GPipe: Enhancing efficiency with pipeline parallelism

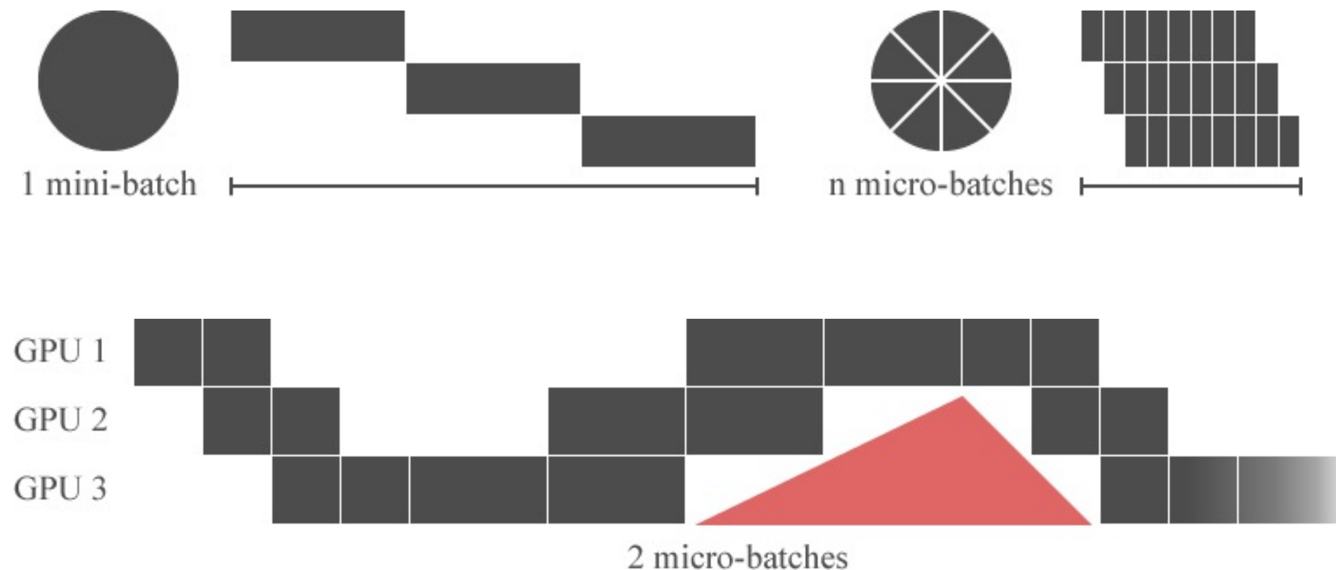
Flexible library for efficient training of large neural networks

- GPipe partitions the network into K cells and places the k-th cell on the k-th accelerator.
- **Forward:** Dividing a minibatch of size N into M equal **micro-batches** (pipelined through K accelerators)
- **Backward:** Gradients for each micro-batch are computed on same model parameters used for forward
- **Update:** To keep consistency, gradients are updated at the end after all micro-batches are processed



GPipe: Enhancing efficiency with pipeline parallelism

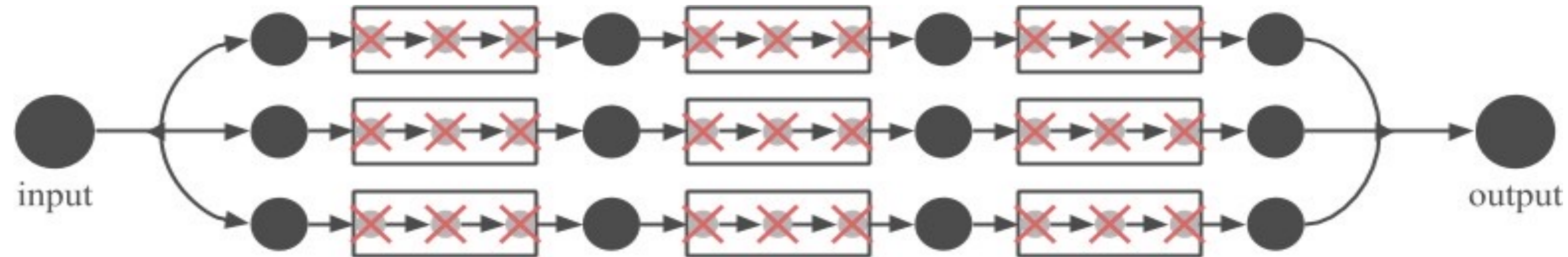
- **Pipeline parallelism**
  - Amount of overlapped processing increases, allowing same amount of data to be processed in less time[3]
  - As size of Micro-batch gets smaller, next GPU can start working faster (idle time gradually decreases)
  - However, size is too small, efficiency is reduced (Not enough to compute workload for GPU)



[3] https://www.kakaobrain.com/blog/66

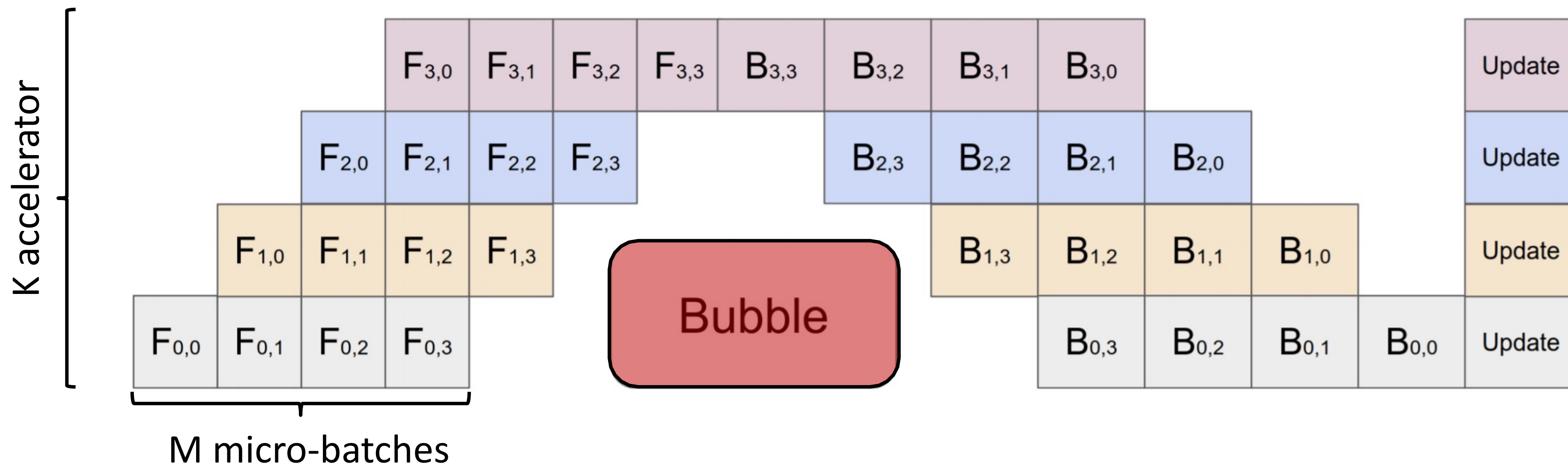
GPipe: Performance optimization

- **Re-materialization: to reduce activation memory requirement**
  - Forward, each accelerator only stores output activations at the partition boundaries
    - **Only hidden layers connecting separate models to each other is placed in memory**
  - Backward, the k-th accelerator recompute forward function



GPipe: Performance optimization

- **Re-materialization: to reduce activation memory requirement**
  - Forward, each accelerator only stores output activations at the partition boundaries
  - Backward, the k-th accelerator recompute forward function
  - Bubble overhead: some idle time per accelerator



Bubble time is amortized over number of micro-steps

(to be negligible when M≥4*K)

GPipe: Performance analysis --- Scalability

- **Performance with two different types of models: AmoebaNet**
  - **AmoebaNet** (convolutional model), **Transformer** (sequence-to-sequence model)
  - Both re-materialization and pipeline parallelism **to benefit memory utilization**
  - **Biggest model size GPipe can support under reasonably large input size**
  - **25x** more memory than what is possible w/o GPipe

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

GPipe: Performance analysis --- Scalability

- **Performance with two different types of models: Transformer**
  - **AmoebaNet** (convolutional model), **Transformer** (sequence-to-sequence model)
  - Both re-materialization and pipeline parallelism **to benefit memory utilization**
  - **Biggest model size GPipe can support under reasonably large input size**
  - **298x** more memory than what is possible w/o GPipe

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

GPipe: Performance analysis --- Efficiency

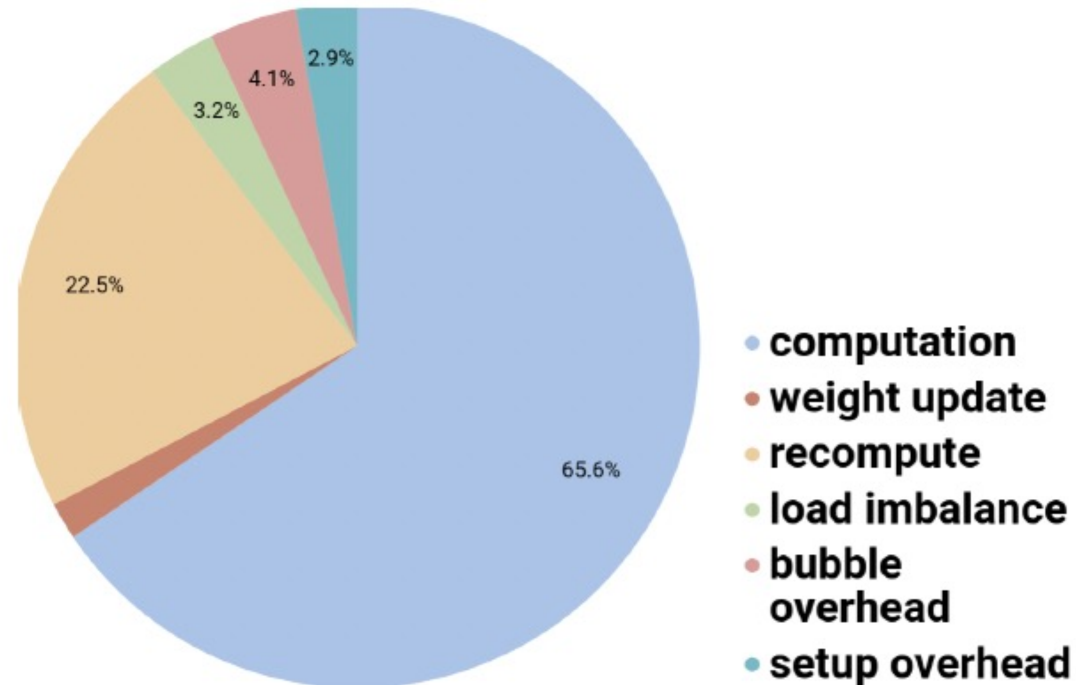
- **Normalized training throughput of AmoebaNet and Transformer**
  - In Transformer, for M=32 (# micro-batch per minibatch), the speed up from k=2 to k=8 is **3.5x**. The improvement is almost **linear**. (k= #partitions)
  - In AmoebaNet, the improvement is **sublinear** due to imbalanced computation distribution.

TPU Throughputs

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: Performance analysis --- Overhead Breakdown

- **There are some overheads…**



Recompute introduces performance overhead in exchange for smaller memory footprint

GPipe: Case Study

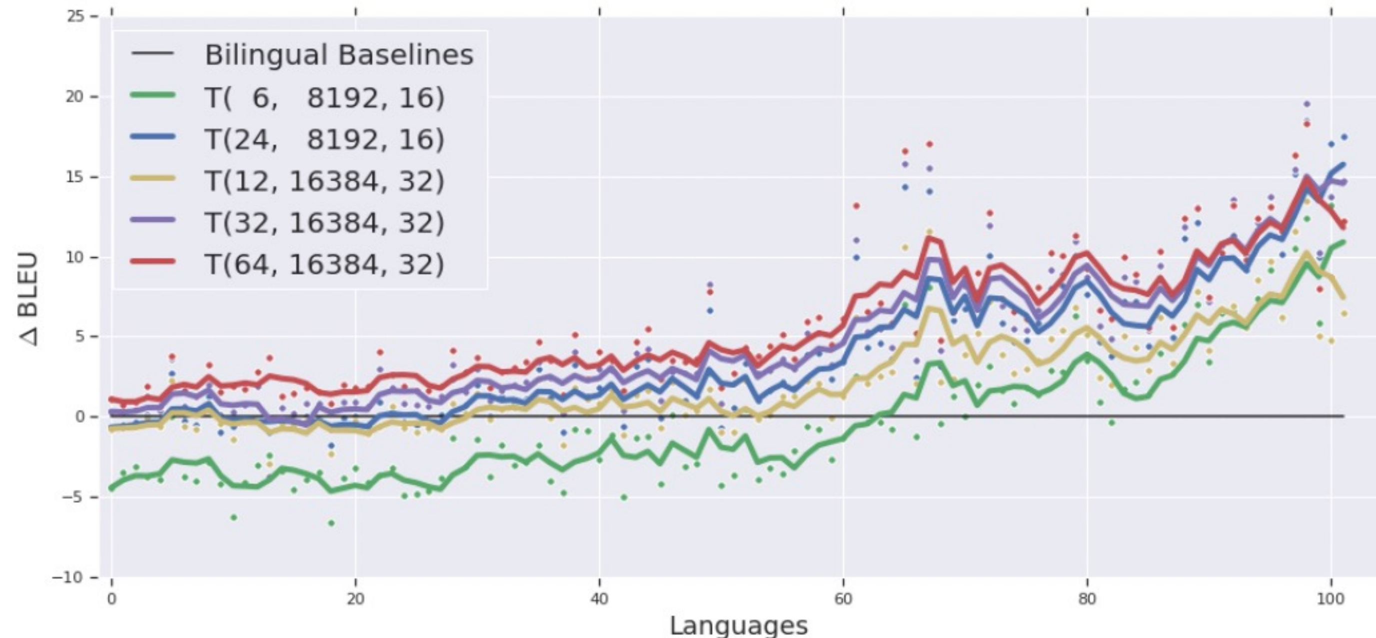
- **Image classification**
  - AmoebaNet with scaled input image size (480x480)
  - Single model: 4 partitions, 84.4$ (top-1), 97% (top-5)
  - Effectiveness of giant convolution networks on other image datasets through transfer learning

Table 5: Image classification accuracy using AmoebaNet-B (18, 512) first trained on ImageNet 2012 then fine-tuned on others. Please refer to the supplementary material for a detailed description of our training setup. Our fine-tuned results were averaged across 5 fine-tuning runs. Baseline results from Real *et al*. [12] and Cubuk *et al*. [26] were directly trained from scratch. *Mahajan *et al*.'s model [27] achieved 85.4% top-1 accuracy but it was pretrained on non-public Instagram data. Ngiam *et al*. [28] achieved better results by pre-training with data from a private dataset (JFT-300M).

Dataset	# Train	# Test	# Classes	Accuracy (%)	Previous Best (%)
ImageNet-2012	1,281,167	50,000	1000	**84.4**	83.9 [12] (85.4* [27])
CIFAR-10	50,000	10,000	10	**99.0**	98.5 [26]
CIFAR-100	50,000	10,000	100	**91.3**	89.3 [26]
Stanford Cars	8,144	8,041	196	94.6	**94.8*** [26]
Oxford Pets	3,680	3,369	37	**95.9**	93.8* [29]
Food-101	75,750	25,250	101	**93.0**	90.4* [30]
FGVC Aircraft	6,667	3,333	100	92.7	**92.9*** [31]
Birdsnap	47,386	2,443	500	**83.6**	80.2* [32]

GPipe: Case Study

- **Massive Massively Multilingual Machine Translation**
  - Scaled transformer: (1) depth by increasing number of layers, (2) width by increasing hidden dimensions
  - 1.3B wide model with (12,16384,32), 1.3B deep model (24,8192,16)
  - Quality of both model is similar, deeper model outperforms by huge margins on low-resource languages (suggesting that increasing model depth might be better for generalization)



*BLEU (Bilingual Evaluation Understudy) score

GPipe: Conclusion

- Training time is reduced by **parallelism and pipelining** to minimize bubble time.
- **Activation memory requirement** is reduced by micro-batching and re-computing the forward activations.
- Gpipe with 8 partitions of Nvidia 8GB GPU each, can support a AmoebaNet-D of up to **1.5B parameters.**
- **Training throughput** can be increased by tuning #partitions and #micro-batches.
- While powerful and giant models can be supported, the **overheads** are not negligible.

Questions to Discuss?

- Memory requirements and computation flops of different layers are often quite imbalanced (although transformer is a relatively balanced model), how could we automatically find ”optimal” partitioning (the scheduling algorithm)?

- How possible node failures are taken into account?

- Gpipe assumes that a single layer is able to fit the memory requirements of a single accelerator. How to combine the partitioning of intra-layer and inter-layer and use the resource efficiently?