PIPELINE PARALLELISM

FOR REAL-TIME ONLINE LEARNING

Enrico Meloni

SAILab, University of Siena

Introduction

MOTIVATIONS

- Ongoing increase in **model complexity**
- **Scaling up** network capacity enhances performances
- Hardware capabilities are **unable to scale** as fast as required
- Need for alternative **parallel computations**
- Take advantage of **multiple** accelerators

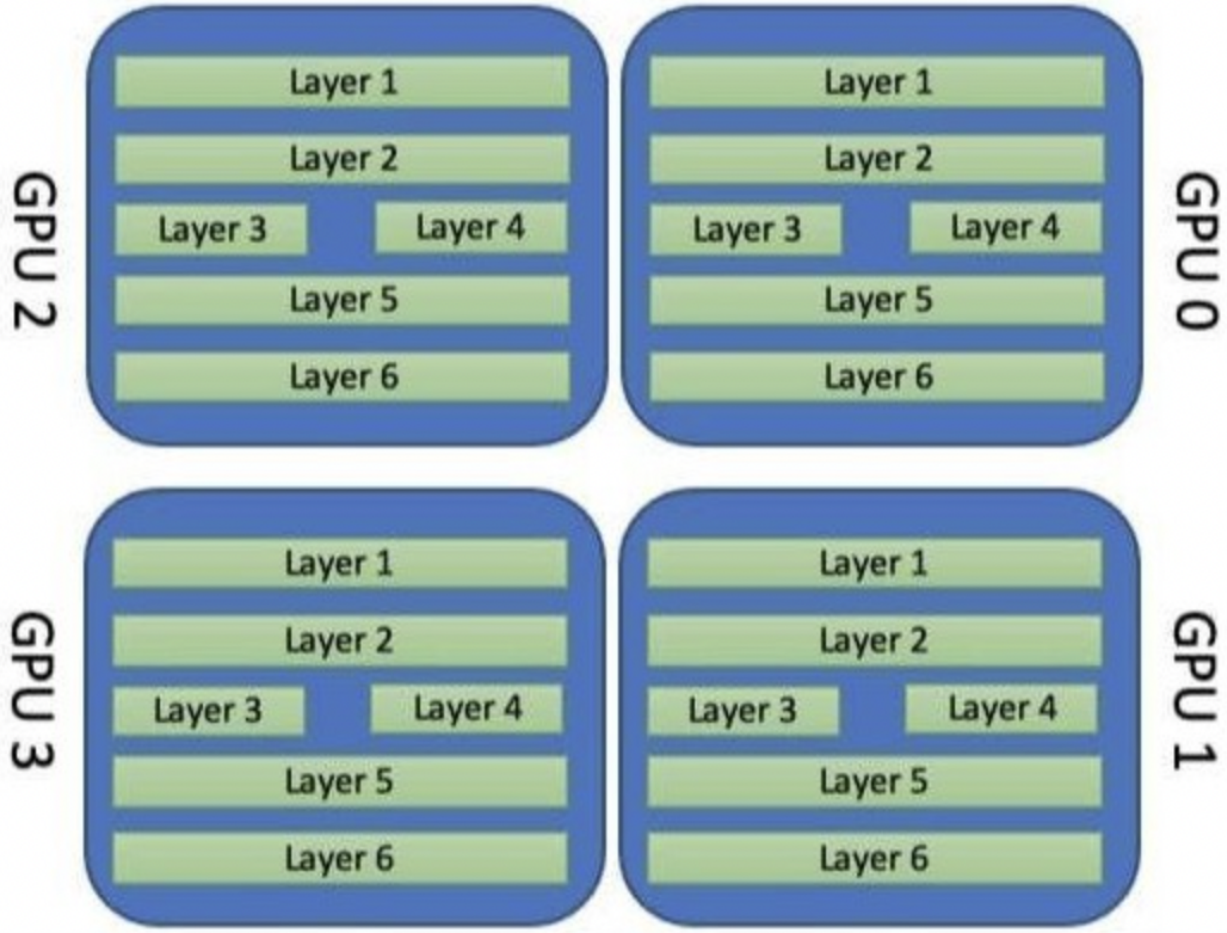
PARALLEL ALGORITHMS

- Usually they are **tailored** to the task at hand
- Characterized by a **difficult trade-off** among:
  - Flexibility
  - Scaling capacity
  - Achievable performances
- Two main categories:
  - **Data** Parallelism
  - **Model** Parallelism

DATA PARALLELISM

- Dataset is **split** into N parts, one for each available GPU
- The model is **replicated** into each GPU
- Each portion of data is **processed independently** by a GPU
- The result of each part is **aggregated at the end**
- Gradients are similarly computed in the **backward pass**
- Pros:
  - Almost **linear speed-up** in training
  - Little to **no need to adapt the model**
- Cons:
  - The model need to **fit in every GPU**
  - Really **useful only in batch** processing

DATA PARALLELISM VISUALIZED

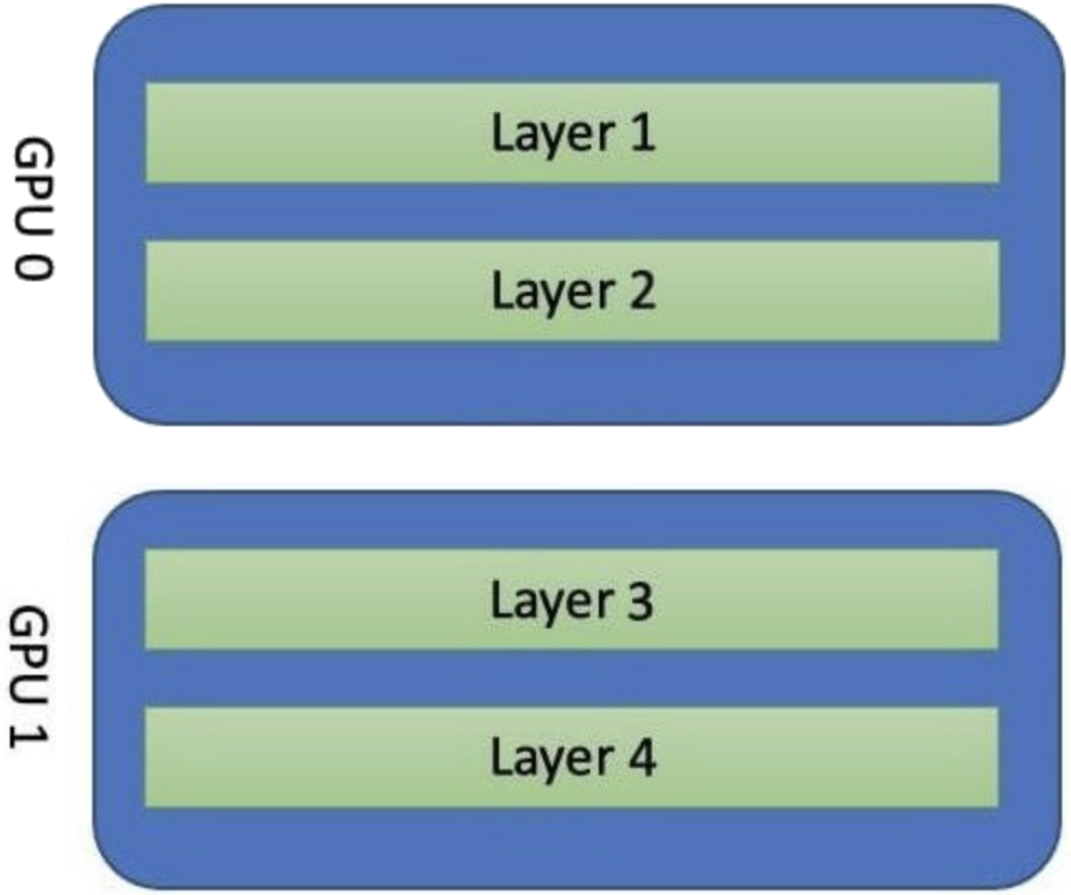


How Data Parallelism works (Source: Deep Learning on Supercomputers)

MODEL PARALLELISM

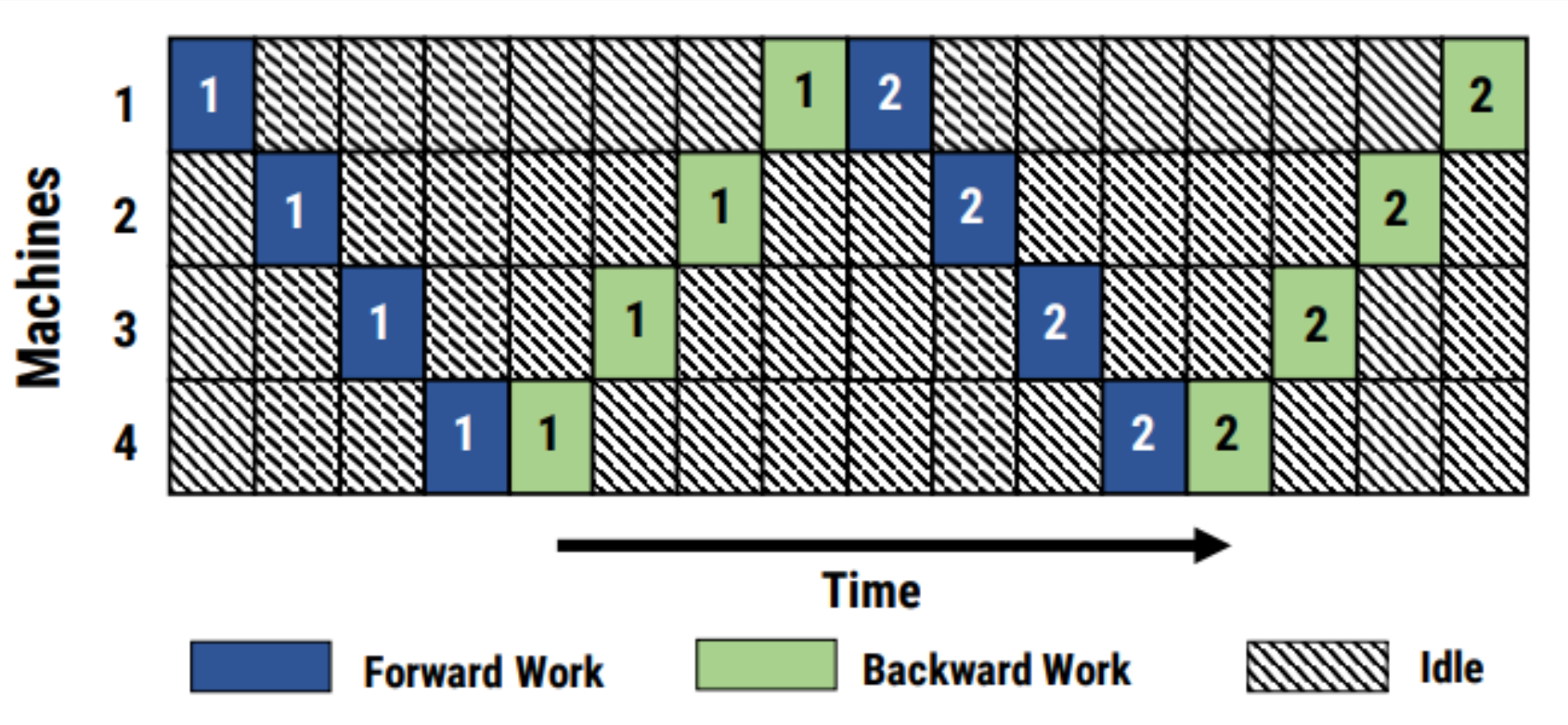
- The model is **split** into N parts, one for each available GPU
- Each part of the model **runs on its GPU**
- Pros:
  - Model **is not required to entirely fit** into each GPU
  - Allows to **parallelize indepedent computations**
- Cons:
  - Splitting the model into parts is **not always straightforward**
  - GPUs for the first parts **need to wait** the latest parts (**idle**)

MODEL PARALLELISM VISUALIZED



How Model Parallelism works (Source: Deep Learning on Supercomputers)

MODEL PARALLELIM TIMELINE



ASYNCHRONOUS MODEL PARALLELISM

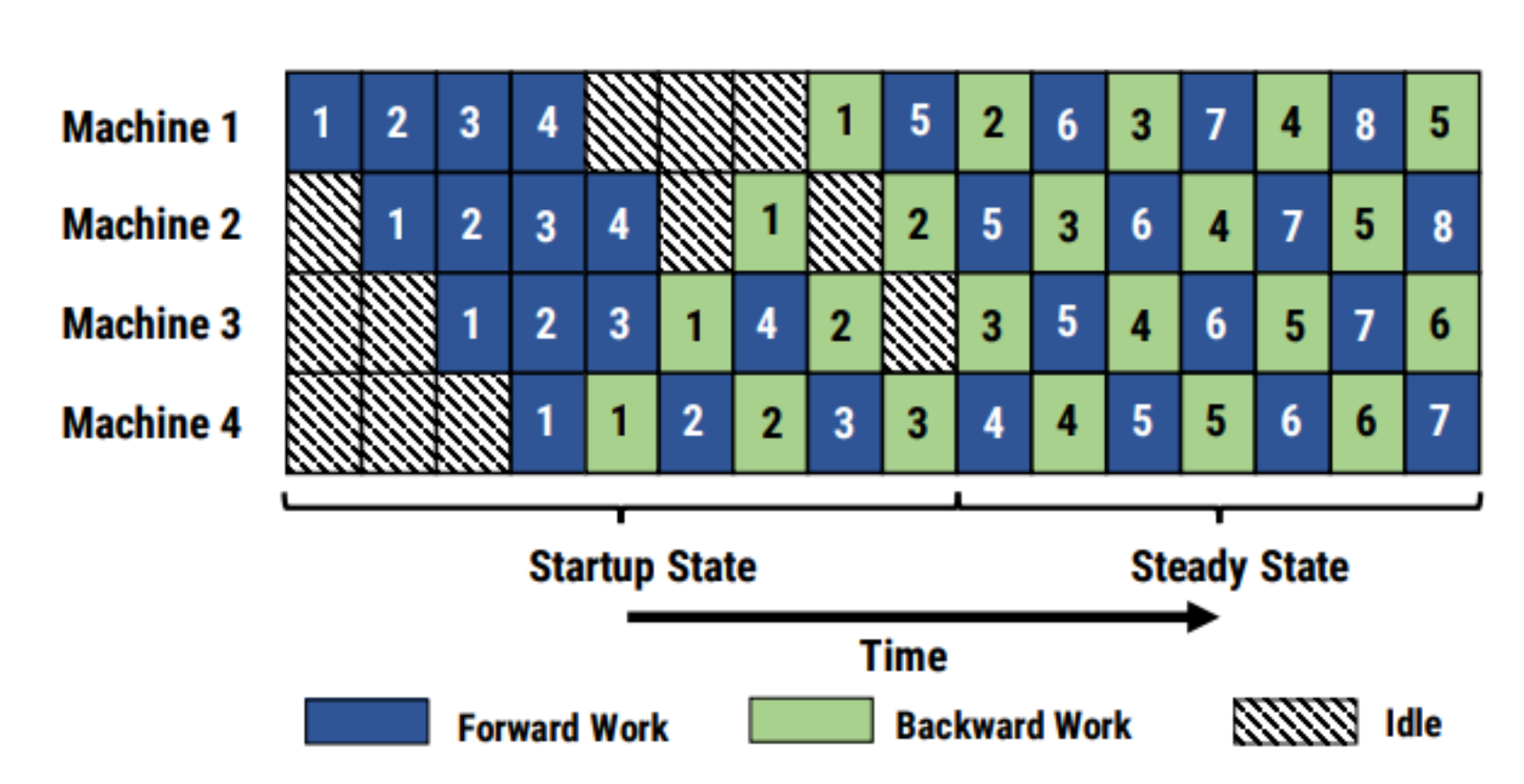
- As soon as a part of the model is ready, **start processing new data**
- This clearly **reduces idle times** for accelerators
- Thus, it can **greatly speed-up** operations
- But introduces other problems such as **weight staleness**
- Also **weight version mismatch** between forward and backward pass

Pipeline Parallelism

PIPELINE PARALLELISM

- An example of **asynchronous model parallelism**
- A sequential network is **split into N stages**
- Each stage is **assigned to a GPU**
- At each step, the **output of a stage is fed to the next stage**
- If done **naively**, the weight version mismatch problem may occur.

PIPELINE PARALLELISM VISUALIZED

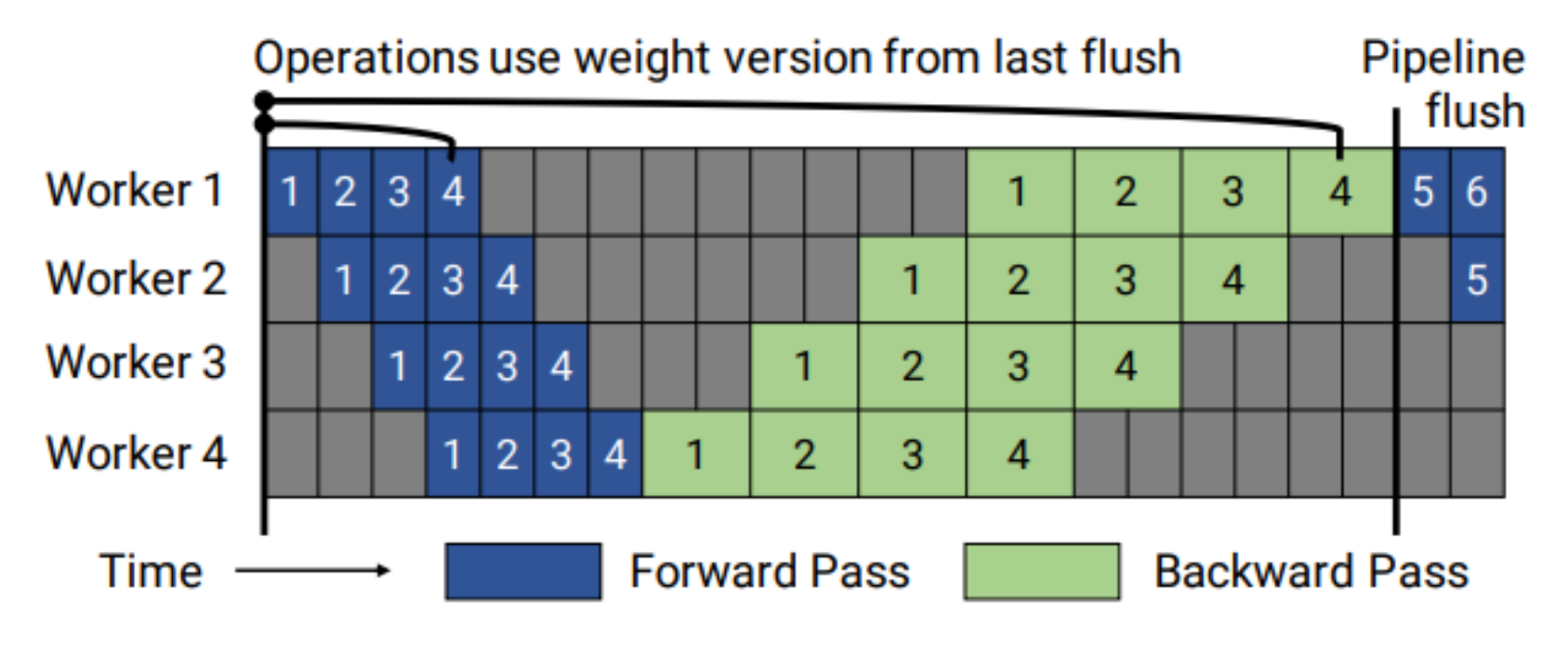


GPIPE

- **GPipe**[1] is an implementation of Pipeline Parallelism
- It splits a minibatch into M **microbatches**
- It keeps a **single version** of the weights
- The weights are updated when the minibatch is **processed entirely**
- This creates a **bubble** where the devices are idle

[1] Yanping Huang et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism". In: *Advances in neural information processing systems* 32 (2019).

GPIPE VISUALIZED

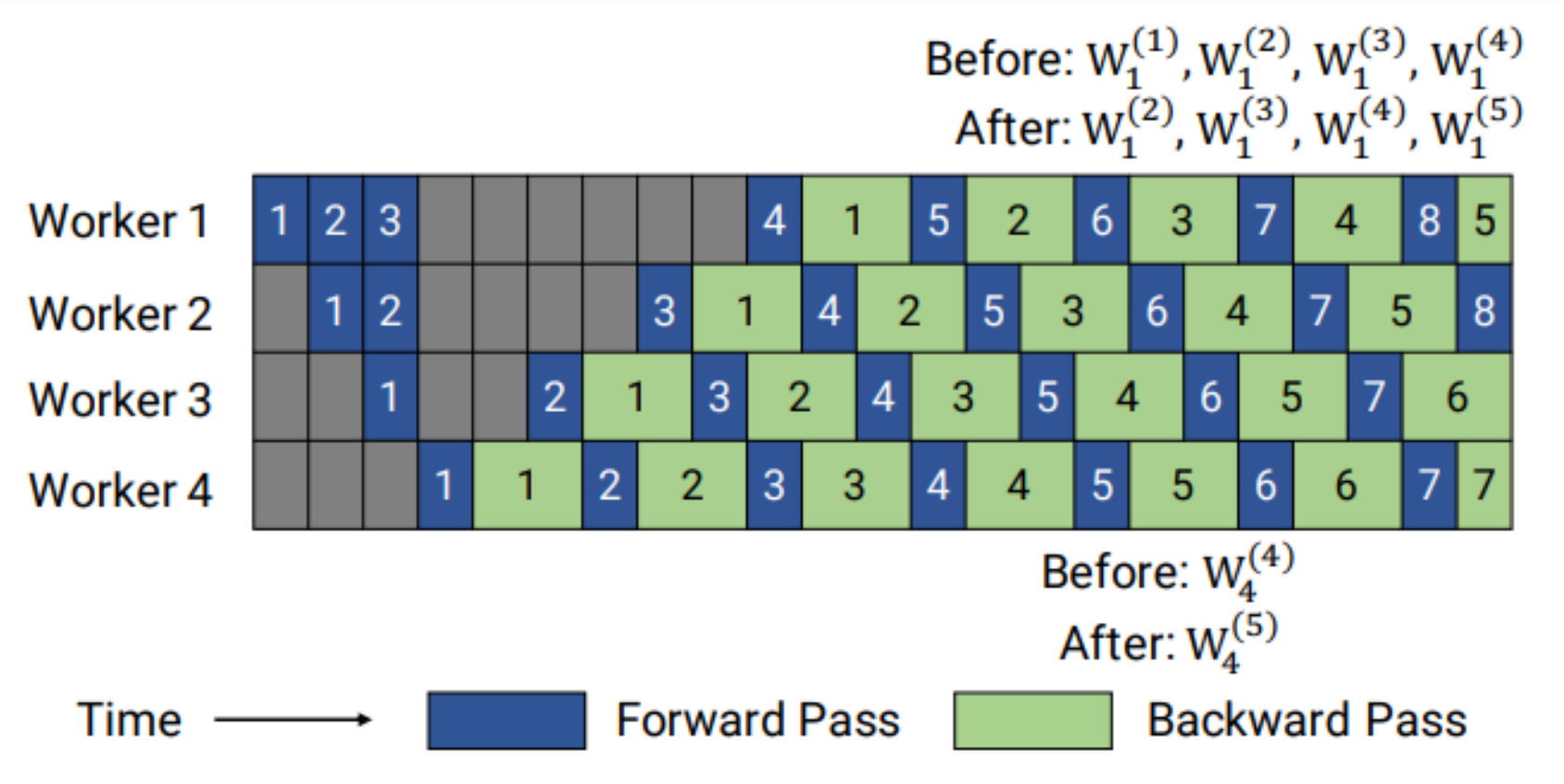


PIPEDREAM

- **PipeDream**[2] is an alternative implementation
- It also splits minibatches into M **microbatches**
- It keeps N **versions of the weights** in a stash
- The stashed versions are used to compute **correct gradients**
- Devices are **never idle at steady-state**
- But **memory consumption is too high** for deep models

[2] Deepak Narayanan et al. "PipeDream: generalized pipeline parallelism for DNN training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 1–15.

PIPEDREAM VISUALIZED



Pipeline Parallelism for Online Learning

PIPELINE PARALLELISM FOR ONLINE LEARNING

- Both of the implementations above assume **batch training**
- Pipeline Parallelism is **applied over a split minibatch**
- There is a **clear distinction** between **start-up** and **steady-state**
- When doing **online learning**, applying this is not straightforward

ISSUES IN REAL-TIME ONLINE LEARNING

- Let us assume that data is provided by a **live source stream**
- We want to process it in **real-time**
- Thus, we need to process a sample **before the next one arrives**
- **Collecting a batch** to apply GPipe or PipeDream would not satisfy real-time requirements
- Thus, we need to start the processing **as soon as the samples arrive**

PARTIME: ONLINE PIPELINE PARALLELISM

- We are working on a PyTorch implementation for **online real-time tasks**
- It allows to process a **live input stream** as soon as the new sample is ready
- Only a **single version of the weights** is kept in memory
- The processing is **very similar to steady-state PipeDream**

PARTIME PROCESSING

- Outputs of the stages are propagated forward
- Gradients are propagated backward and used to update weights **at every step**
- Gradients are computed using the current input, **even if it is not the same** that produced that gradient
- We can assume that given a real video stream, consecutive samples will not differ greatly
- It is an approximation, but necessary to keep **high performances with low memory footprint**

REAL-TIME PROCESSING

- Let T be the **processing time** of the original network
- Let F be the **framerate** of the input stream
- The network can process the stream **in real-time** if $T < \frac{1}{F}$
- Let us consider a **balanced split** of the network into N stages
- The processing time of **each stage** will approx. be $\frac{T}{N}$
- Thus, the **requirement for real-time** becomes $\frac{T}{N} < \frac{1}{F}$
- This requirement is clearly **easier to accomplish**

ISSUES WITH IMPLEMENTATION

- The considerations above do not consider the **communications overheads**
- In a concrete implementation, the **speed-up is not linear**
- Time for a stage would be $\frac{T}{K}$ with $K \geq N$
- For batch processing, most of the overheads are amortized over a great T
- For online real-time processing, T is much lower and **overheads are more impactful**

SOLUTION: CUDA GRAPH

- A recent PyTorch API that allows to **dramatically reduce communications overheads**
- It also **generally optimizes** PyTorch computations
- Basically, it records PyTorch operations **issued by the CPU**
- It also records **synchronization ops** between GPUs
- Then, it **optimizes the operation graphs** and produces **a callable**
- This callable will use a **static pool of memory (no new allocations)**
- This callable is called **by the CPU** and will not **synchronize with it until it finishes**

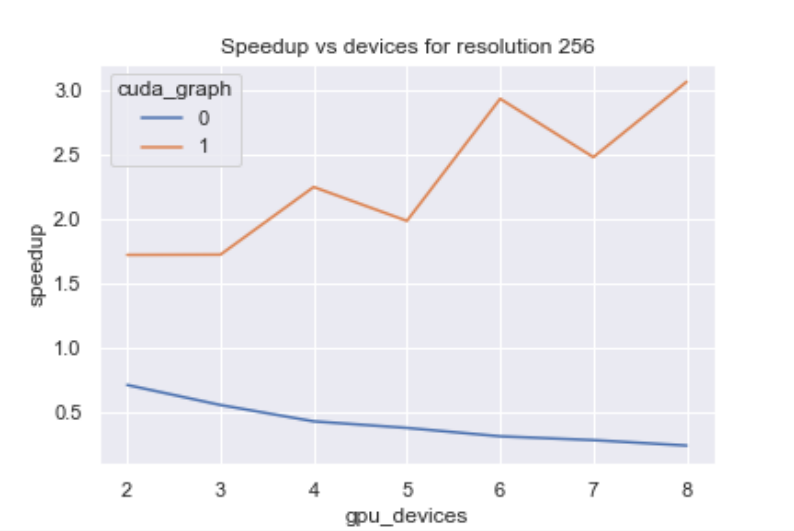
PRELIMINARY RESULTS

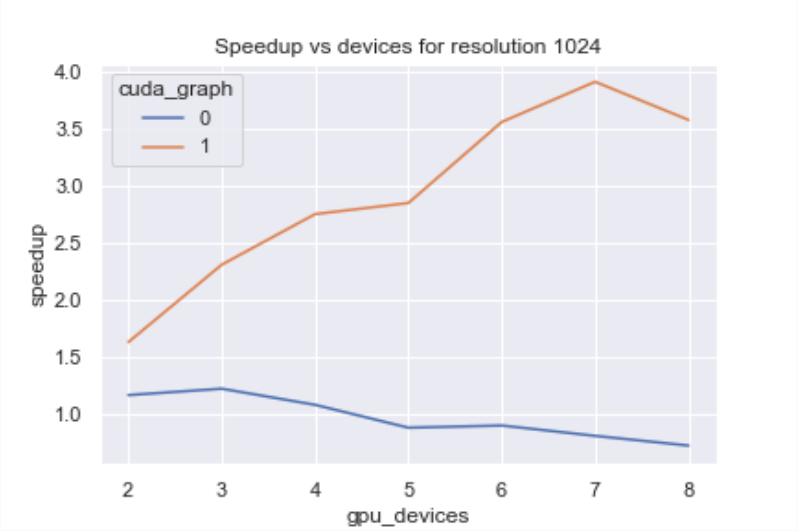
- We have tested the implementation on a **8-GPU Node**[3]
- We have considered a CNN **with no downscaling** (for ease of balancing)
- We have considered different setups in: depth, kernel size, number of output features for each layer
- We considered pipelines with 2 to 8 stages

[3] https://www.hpc.cineca.it/services/iscra

BASIC CONFIGURATION

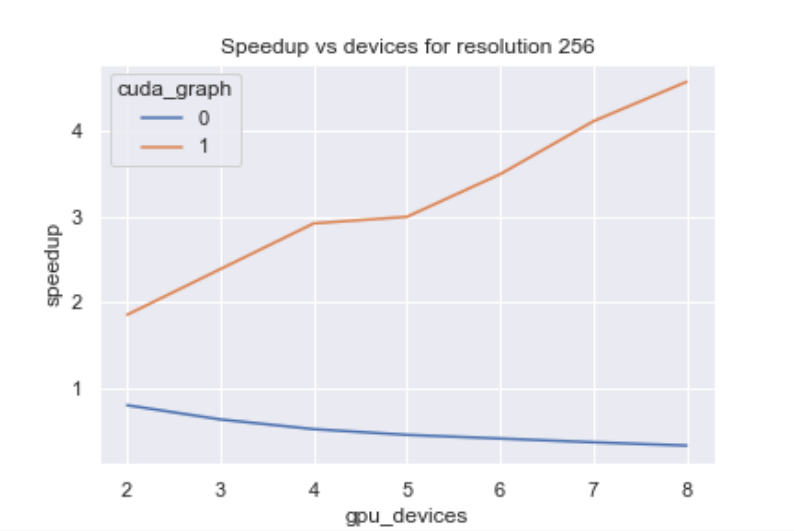
- Layers: 10; features per layer: 16; kernel size: 5

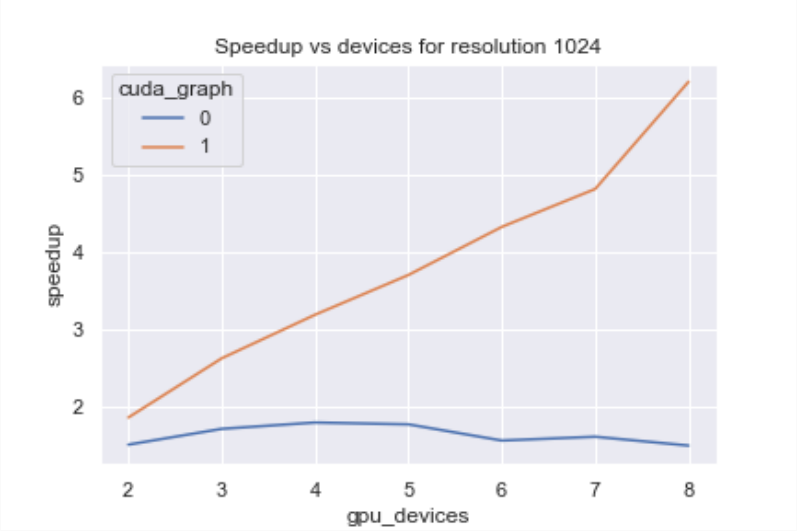




LAYERS X3 CONFIGURATION

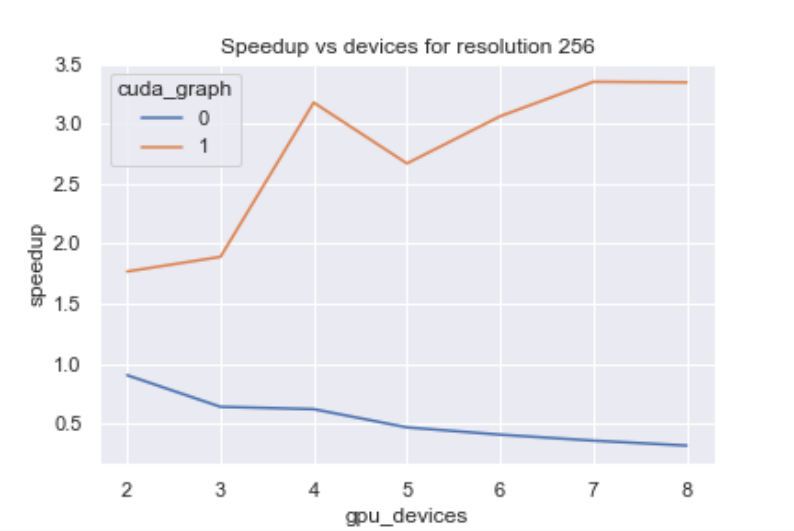
- Layers: 30; features per layer: 16; kernel size: 5

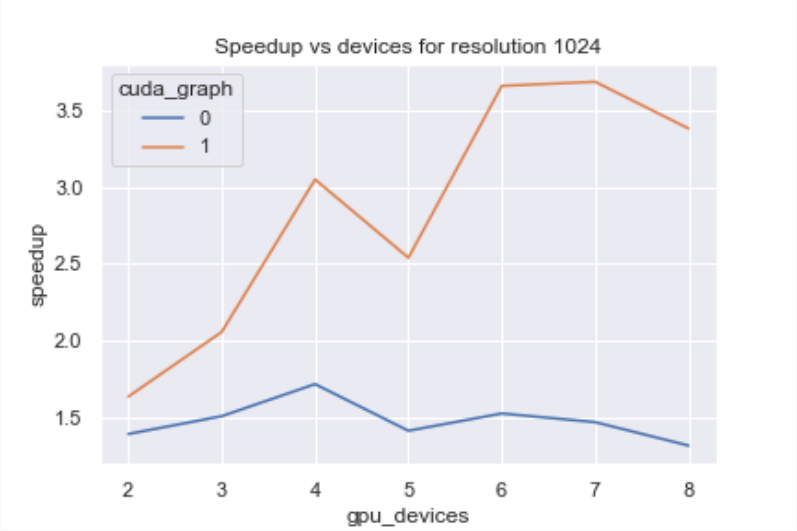




FEATS X3 CONFIGURATION

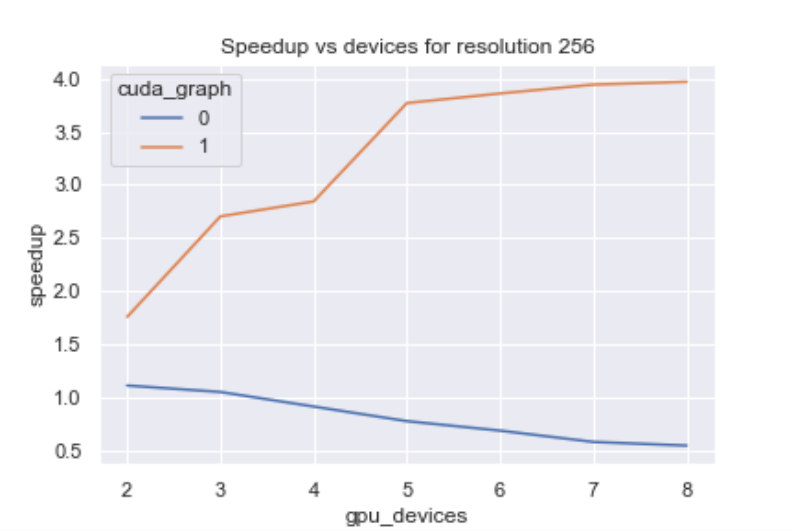
- Layers: 10; features per layer: 48; kernel size: 5

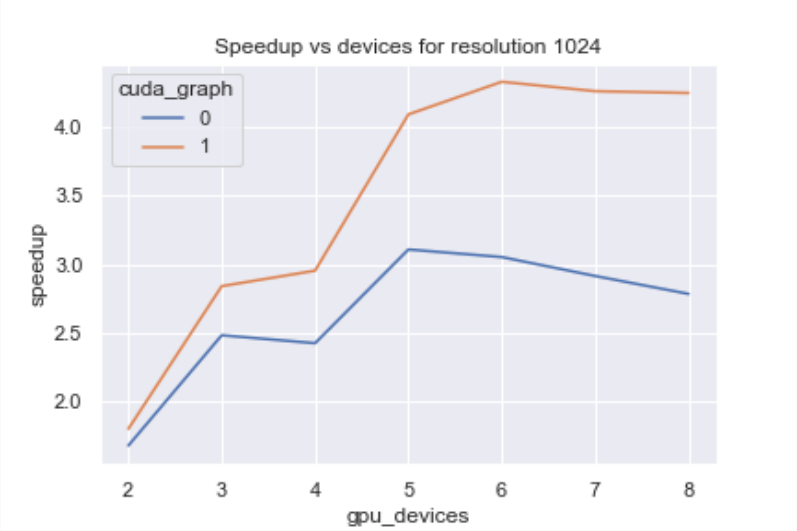




KERNEL SIZE X3 CONFIGURATION

- Layers: 10; features per layer: 16; kernel size: 15





DISCUSSION

- The **deeper** the network, the **more linear** the speed-up
- At **higher resolutions**, the **speed-up is higher**
- In general, **without CUDA Graph**, overheads are **too high** to achieve significant speed-ups
- Exception: with greater kernel size and high reosolution, even without CUDA Graph there is a significant speed-up
- With shallow networks there is a **plateau** even with many GPUs

Conclusions

CONCLUSIONS

- We talked about different parallelism techniques in deep learning
- Model Parallelism seems the most apt for Deep Learning
- Still, it can be difficult to use it at its best
- We talked about Pipeline Parallelism, a straightforward parallelism for sequential networks
- We talked about two implementations, their pros and cons
- We introduced PARTIME to tackle the issue of online real-time learning

References

- Huang, Yanping et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism". In: *Advances in neural information processing systems* 32 (2019).
- Narayanan, Deepak et al. "PipeDream: generalized pipeline parallelism for DNN training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 1–15.

Thank you for listening!