Google Developer Experts

UNIVERSITY OF AMSTERDAM

Training LLMs at Scale

On Parallelization Strategies and Efficiency

ESSIR Summer School

Phillip Lippe
PhD student, GDE JAX/Flax

July 1, 2024

About me

- 4th year PhD student
  - Generative modeling, causality, reasoning
- Intern at Google DeepMind (Gemini) and Microsoft Research (AI4Science)
- Google Developer Expert for JAX/Flax
  - Practical implementation tutorials

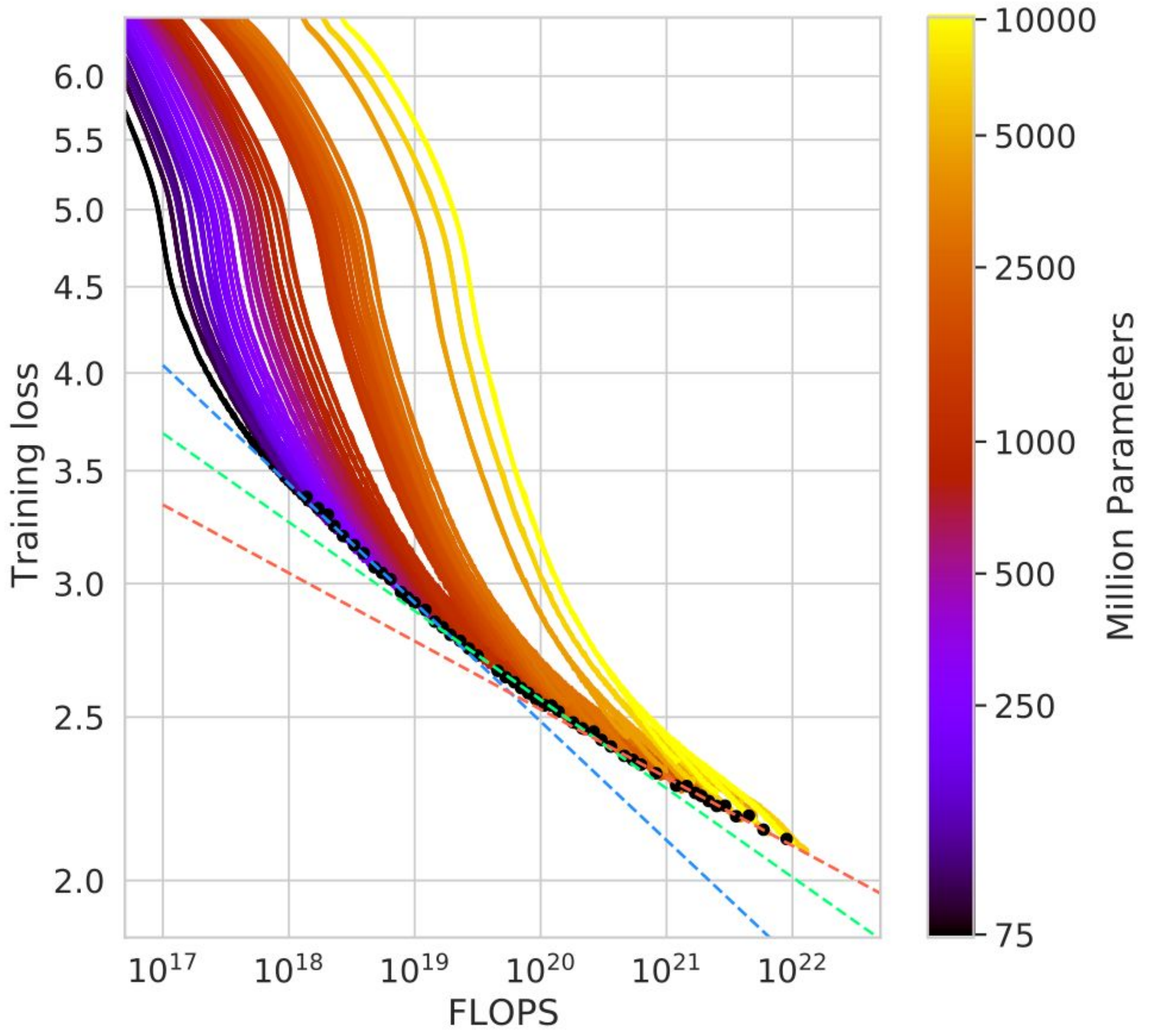
Scaling

- More FLOPs = better performance
- LLM training requires thousands of accelerators
- How do you train massive models efficiently?

Today's focus:
- Concepts and scaling strategies
- Framework agnostic

Implementation Tutorials:
[UvA-DL Notebooks]

Hoffman et al., 2022 - Chinchilla scaling laws



Scaling Challenges

- Limited resources: compute and memory
- Naive training does not fit in memory
  - How do we reduce memory usage?
- Single GPU/TPU would take years to train an LLM
  - How do we utilize maximum FLOPs of all devices?
  - Metric: MFU - Model FLOP/s utilization

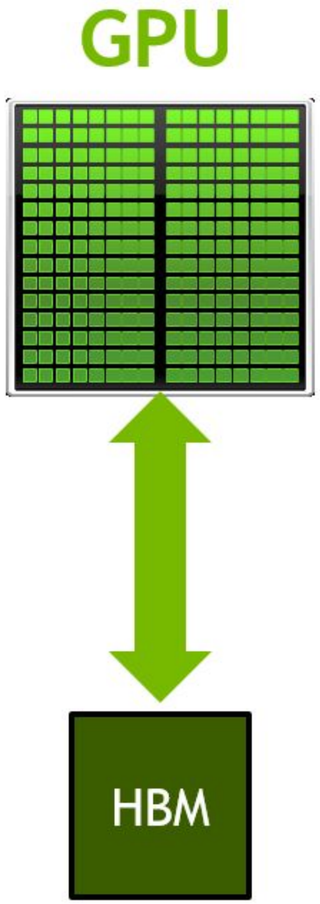
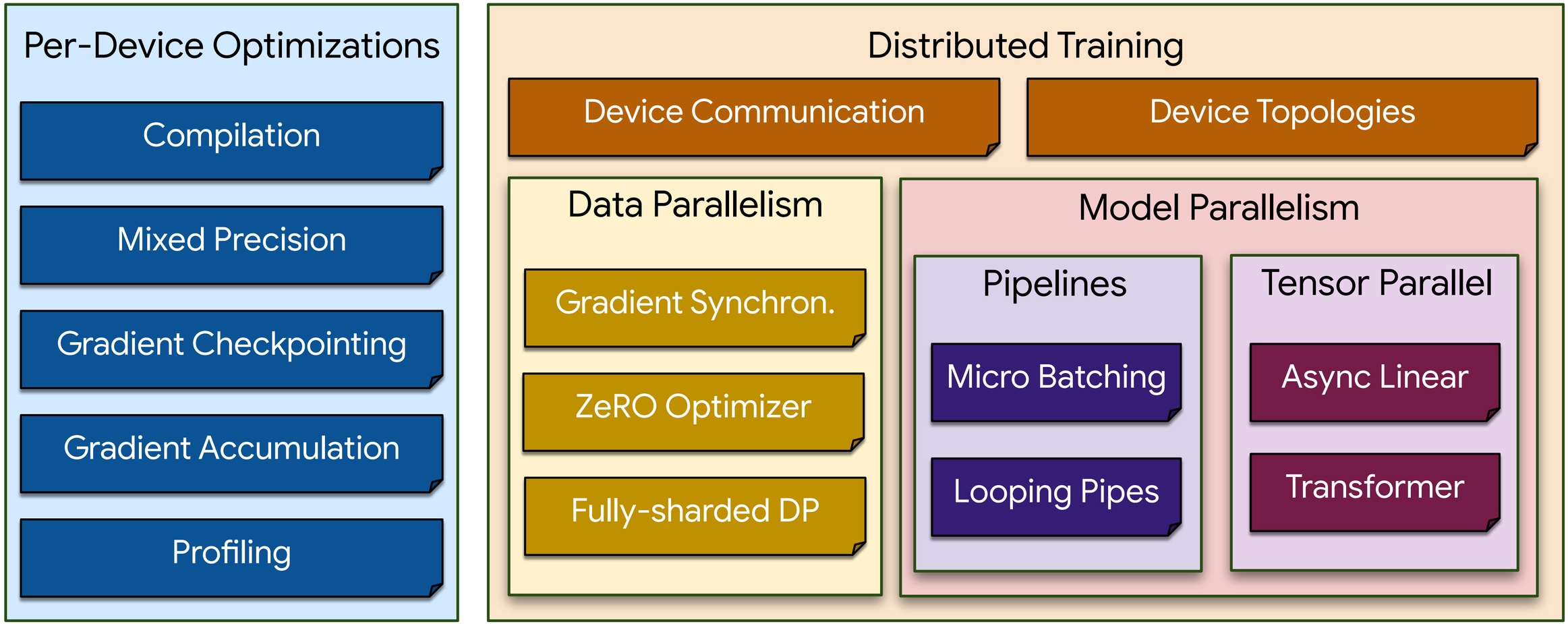


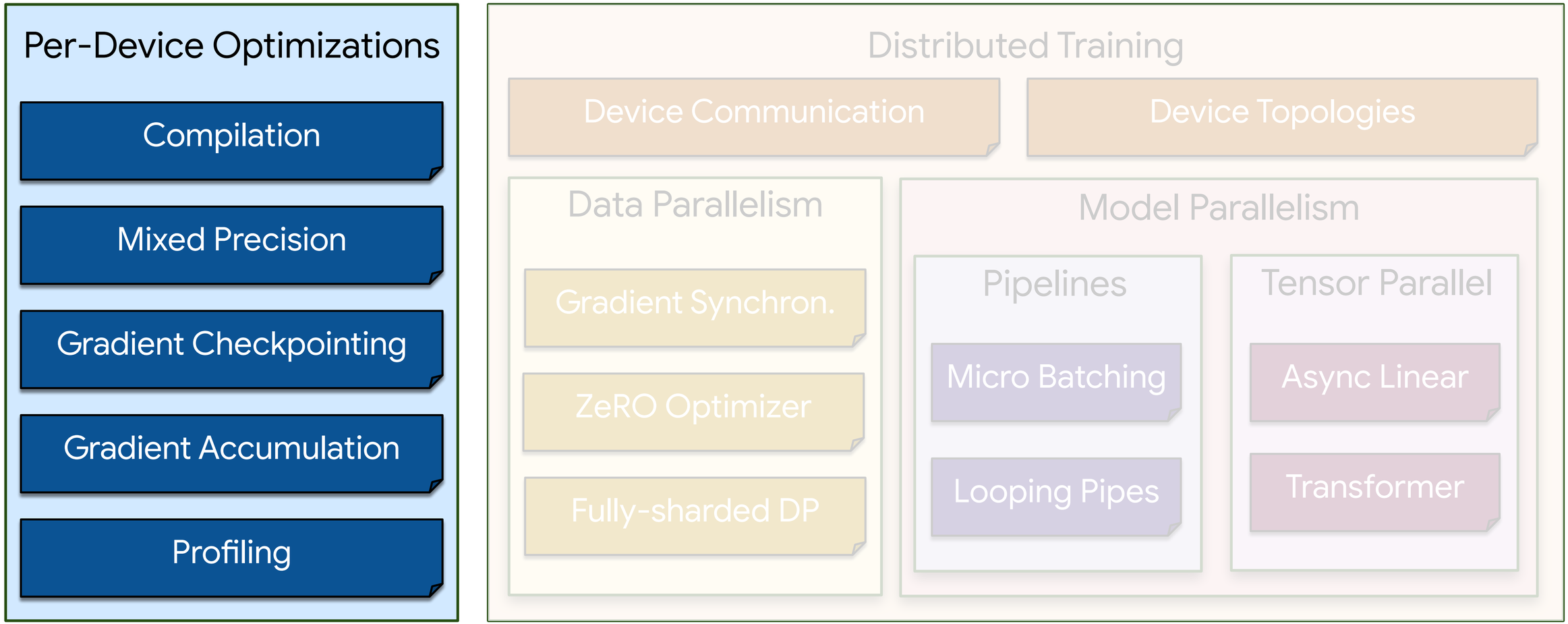
Figure credit: NVIDIA

Overview



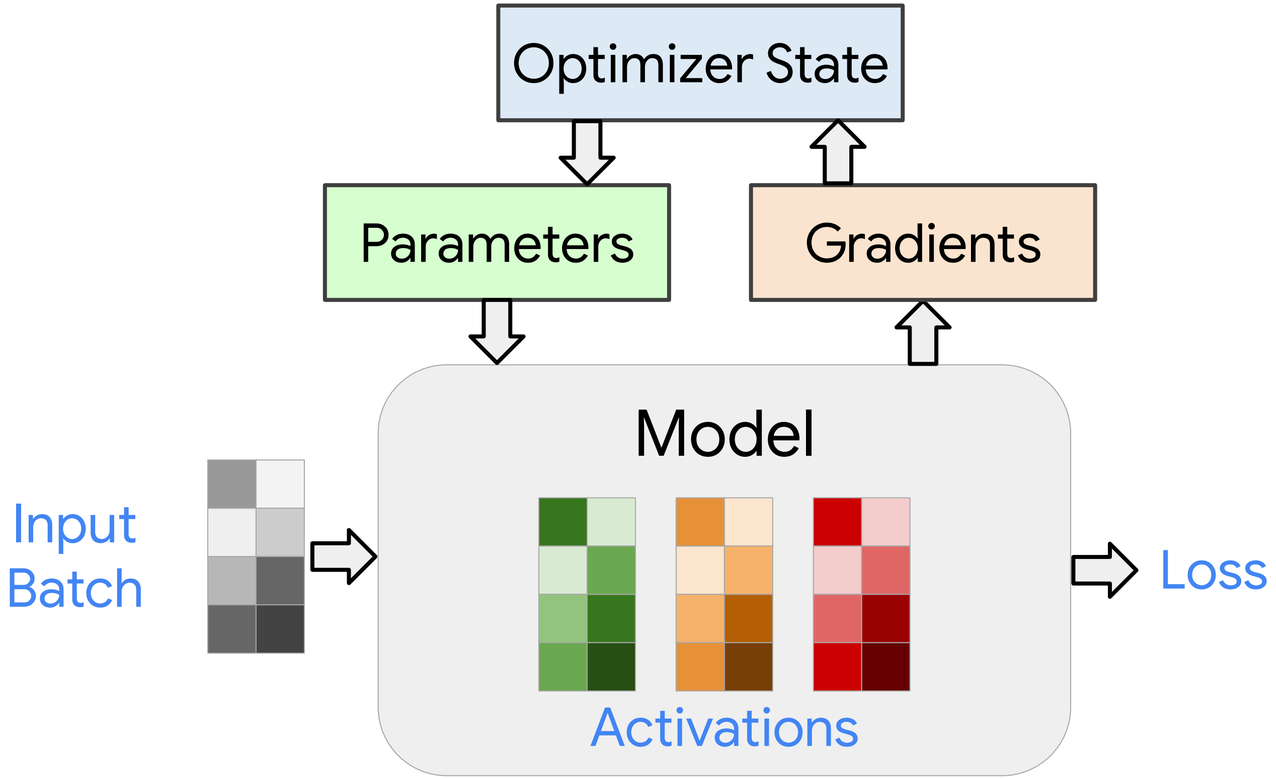
And a lot more out there...

Overview



And a lot more out there...

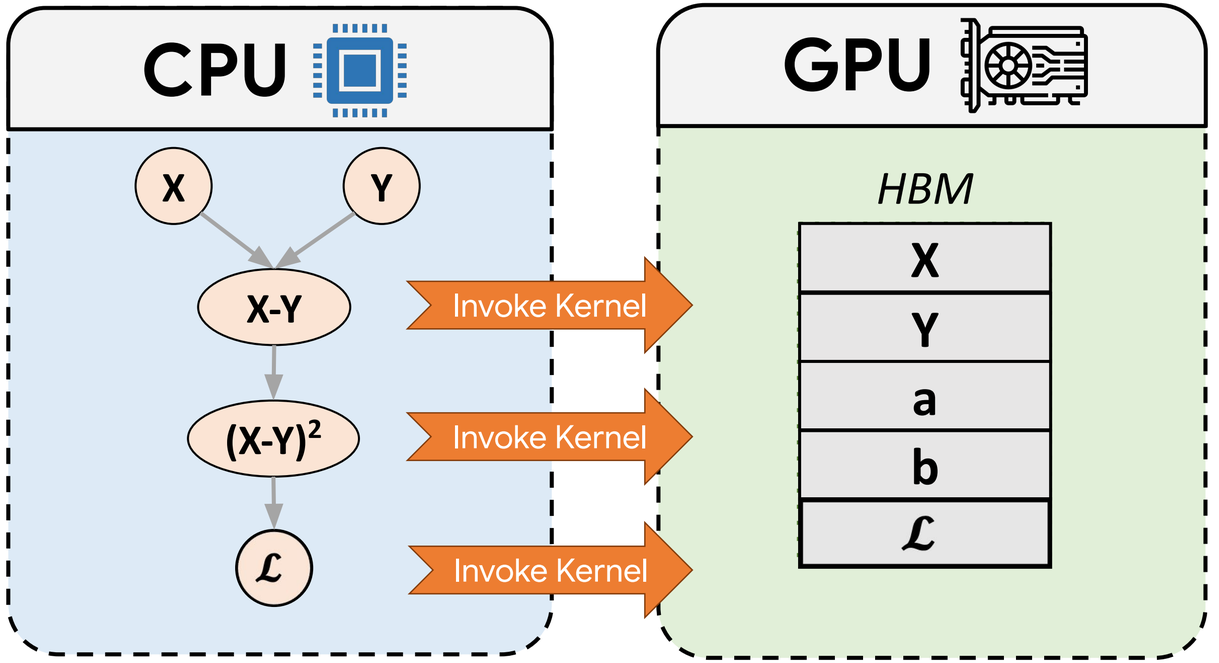
Model Training - As usual



Just-In-Time Compilation

- Execution of operations on accelerator happen asynchronously
- Eager execution requires considerable overhead in CPU ↔ GPU communication

```
def mse_loss(x, y):
    return ((x - y) ** 2).mean()
```



Just-In-Time Compilation

- Compilation fuses operations and creates fewer kernel calls
- Compiler performs accelerator-based optimizations for maximum efficiency

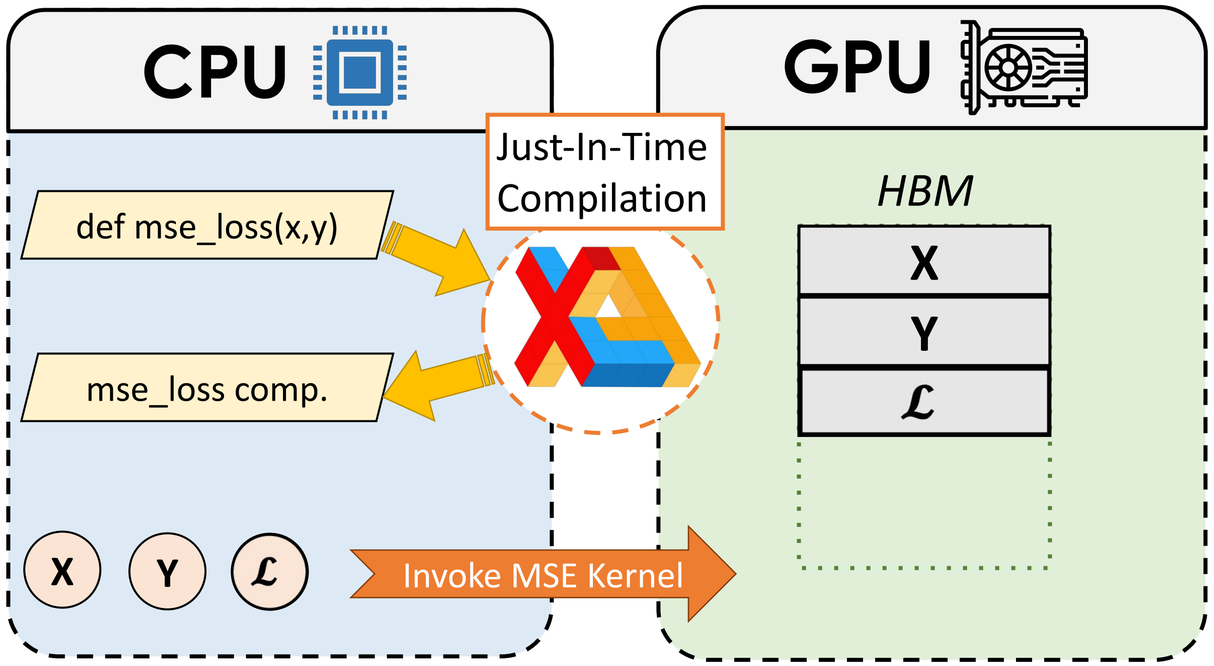
PyTorch:

- torch.compile (link)
- torch.cuda.graph (link)

JAX:

- jax.jit (link)

```
@jax.jit
def mse_loss(x, y):
    return ((x - y) ** 2).mean()
```



Low-Level Kernels

- Compilation finds general optimizations, but may not be optimal across multiple complex operations
- Most GPUs are memory-bandwidth bound (compute outspeeds memory)
- Significant gains can be achieved by minimizing memory bottleneck

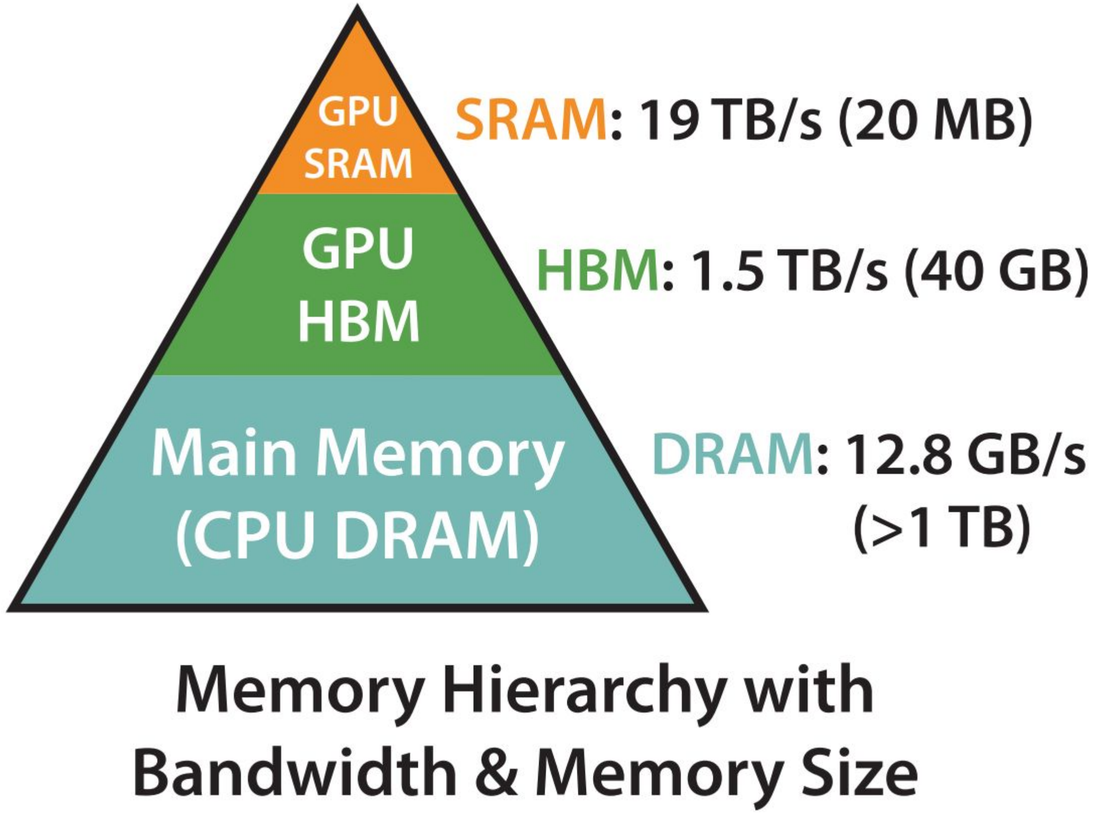
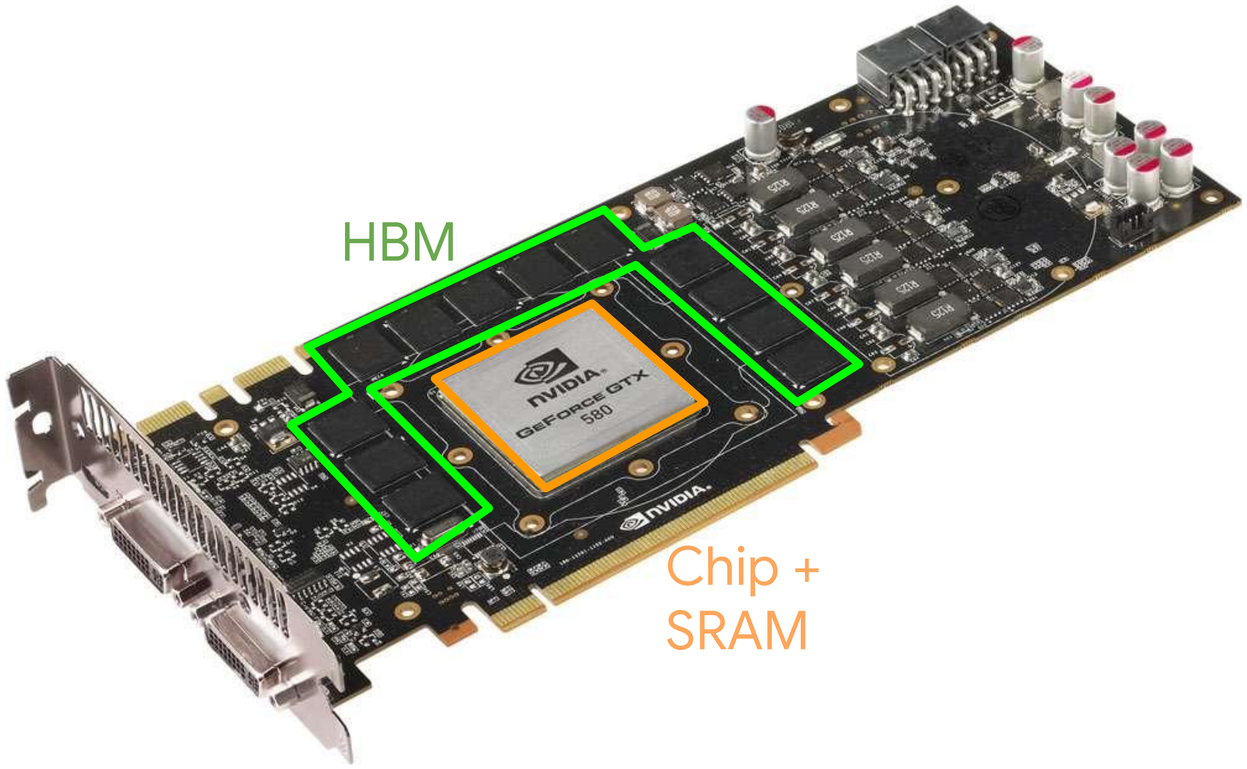
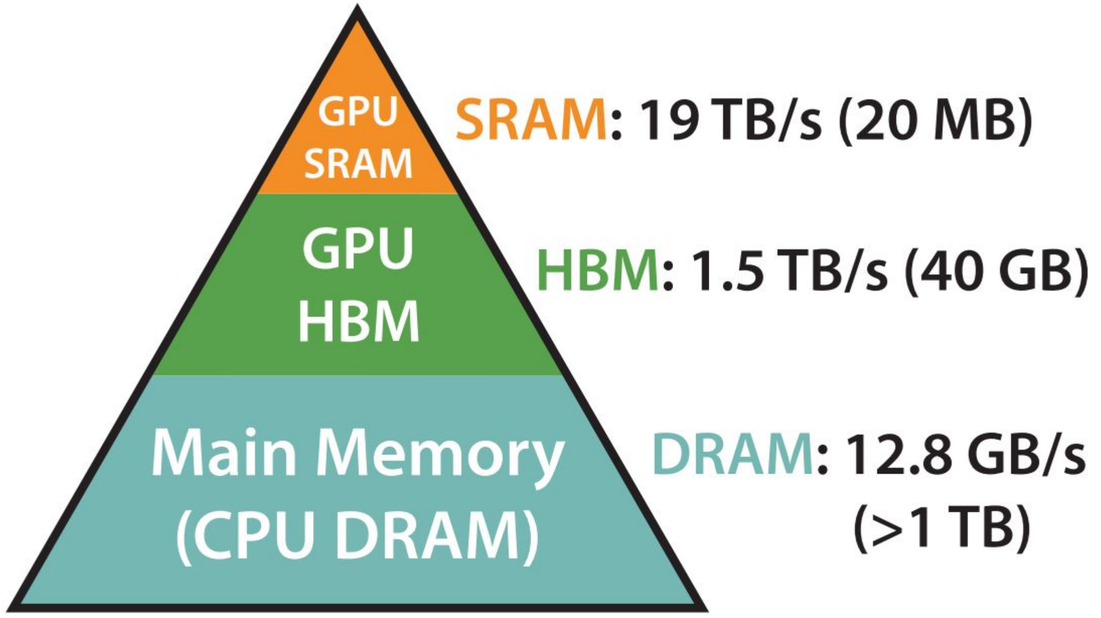


Figure credit: Dao et al., 2022

Low-Level Kernels





Memory Hierarchy with Bandwidth & Memory Size

Figure credit: Dao et al., 2022

Low-Level Kernels

- Example: Attention
- Default eager implementation:

Algorithm 0 Standard Attention Implementation

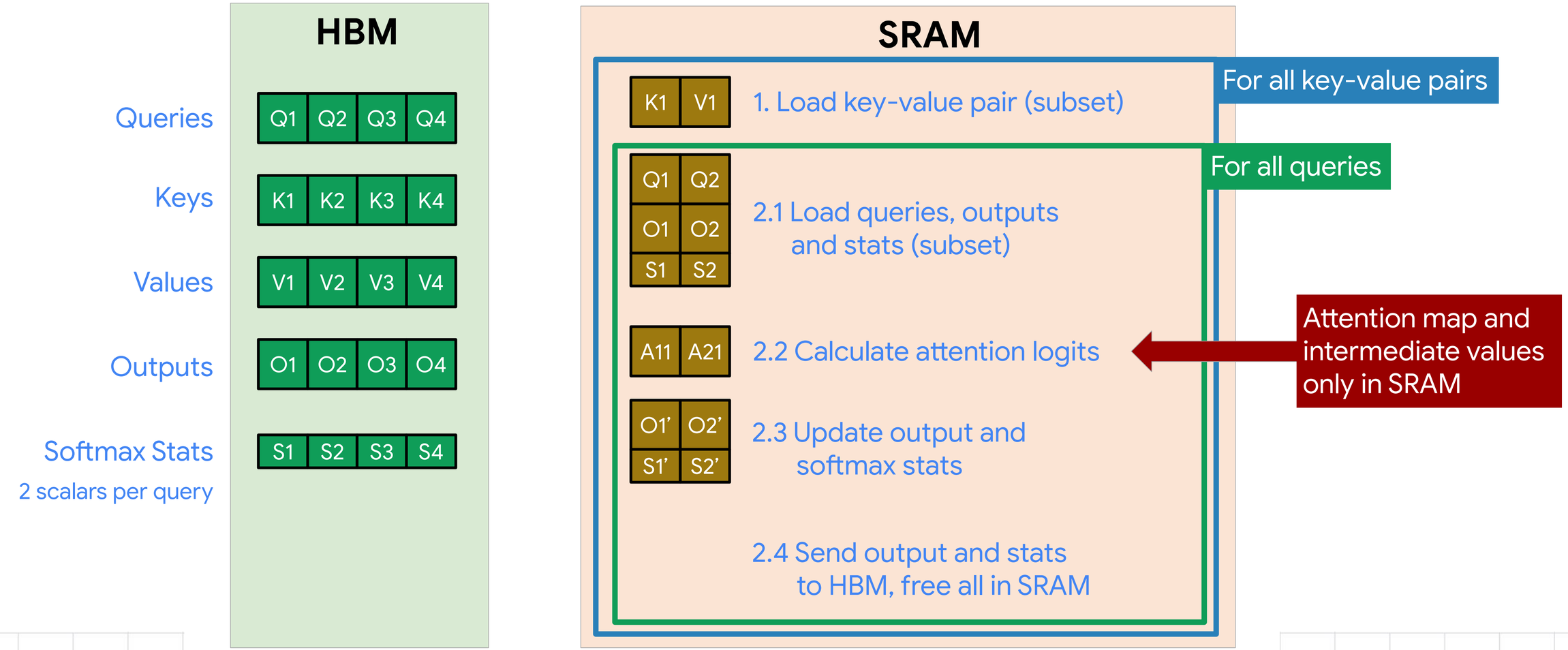
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Figure credit: Dao et al., 2022

- Several slow HBM calls - can we do better?

Flash Attention



Flash Attention

- Example: Flash Attention
- Do not materialize QK^T in HBM, but only communicate final output to HBM
- Calculate one key-value pair at a time for several queries with decomposed softmax
- Recompute attention matrix in backward
- Speedup of 3x over eager implementation
  - Compiling (esp JAX) can be similarly fast
  - Default kernel in PyTorch

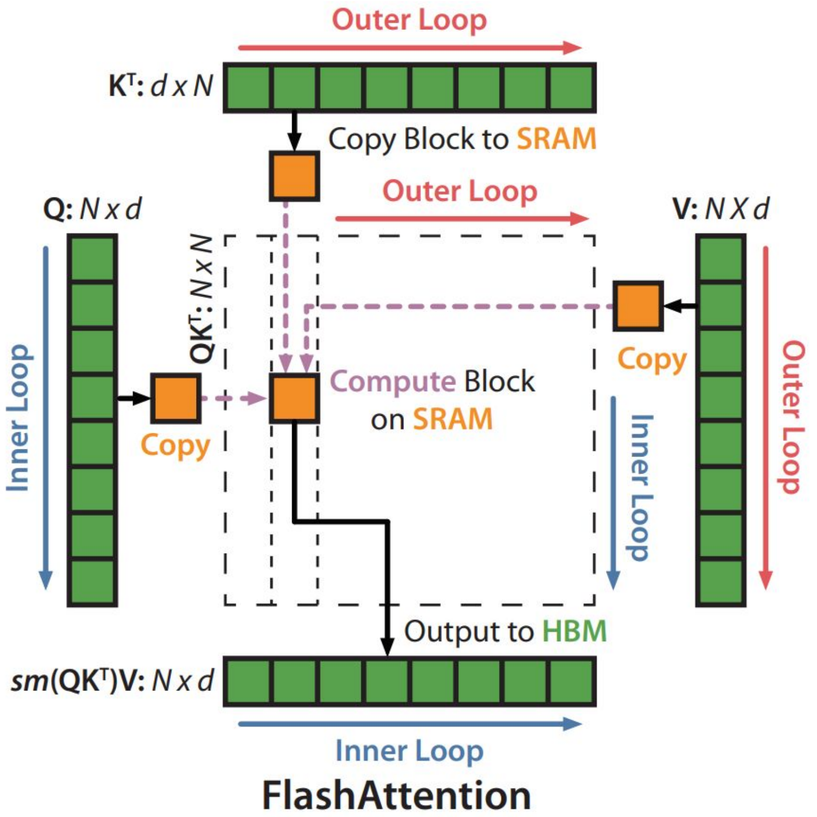


Figure credit: Dao et al., 2022

Low-Level Kernels

- Implementing low-level kernels makes new architectures viable again
- Opens up the “hardware lottery”

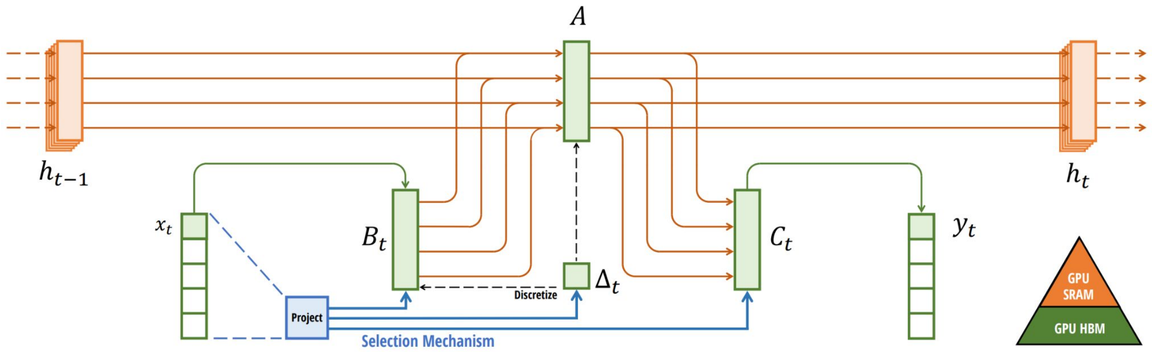


Figure credit: Gu et al., 2023

Mixed Precision

- By default, models train in float32 precision
- Modern accelerators support lower precision with significantly higher throughput

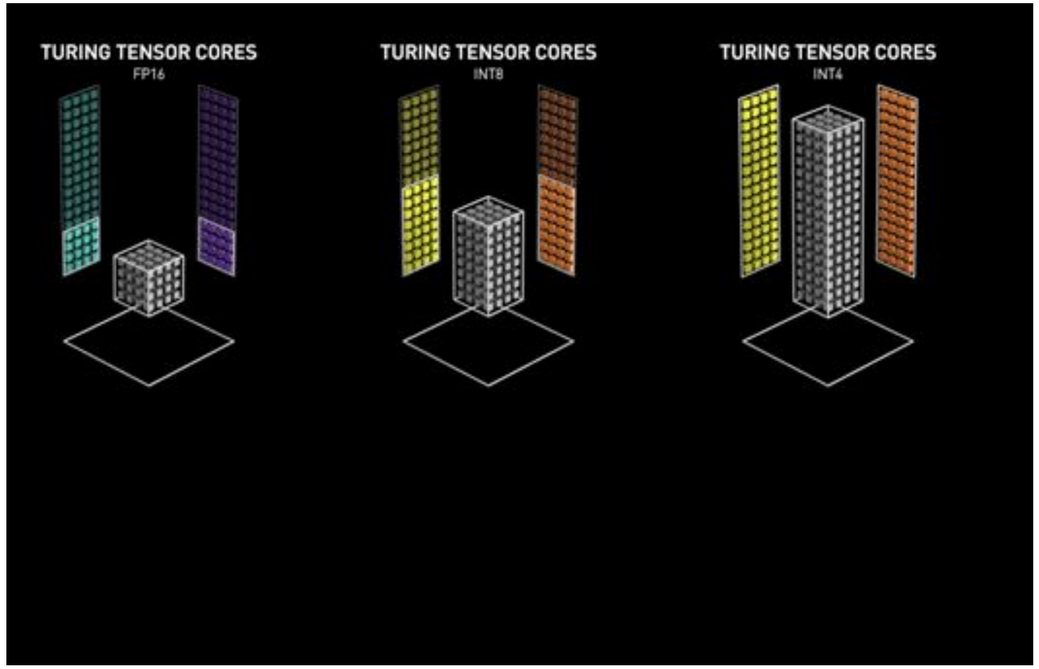


Figure credit: NVIDIA, Tensor Cores

Mixed Precision

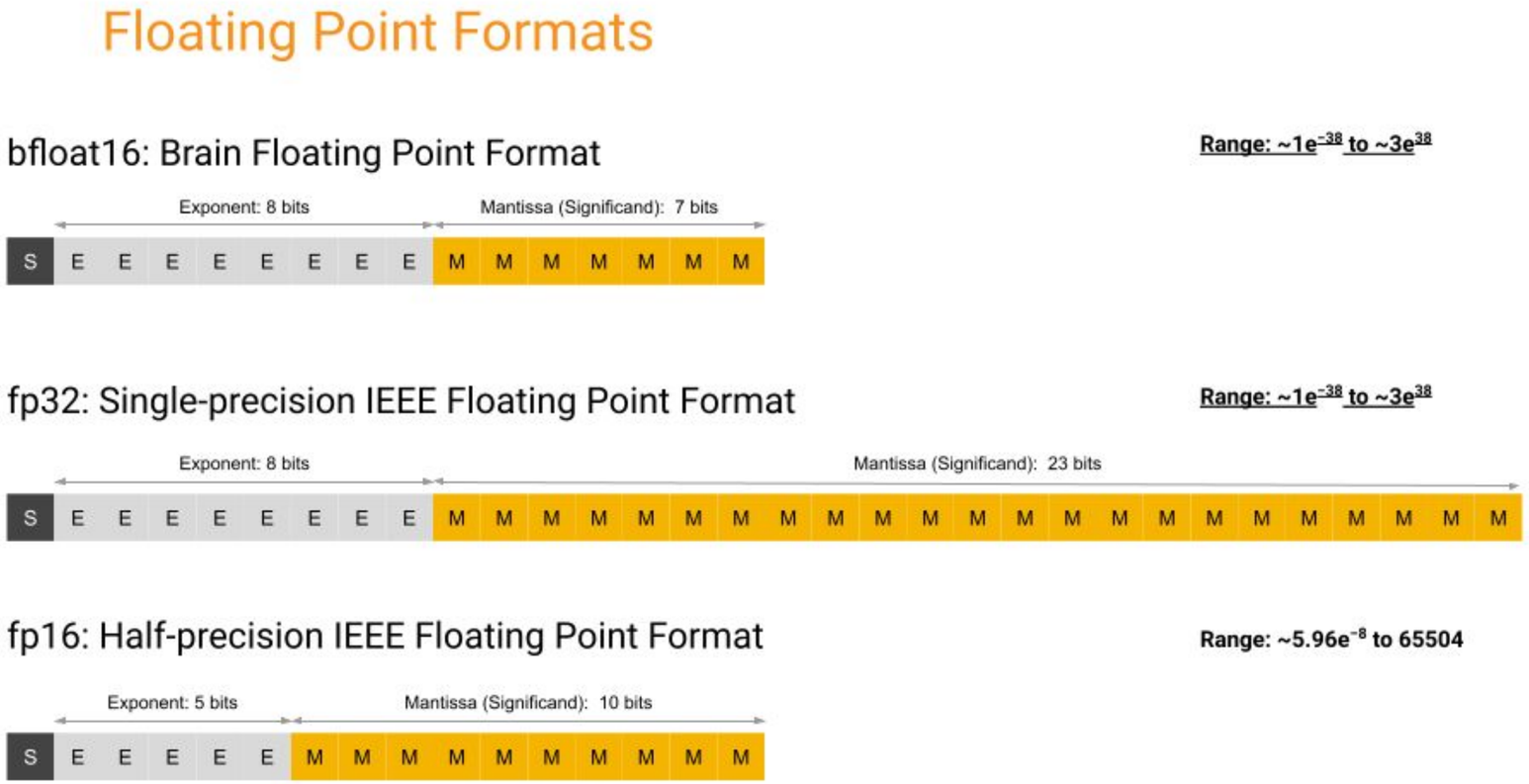
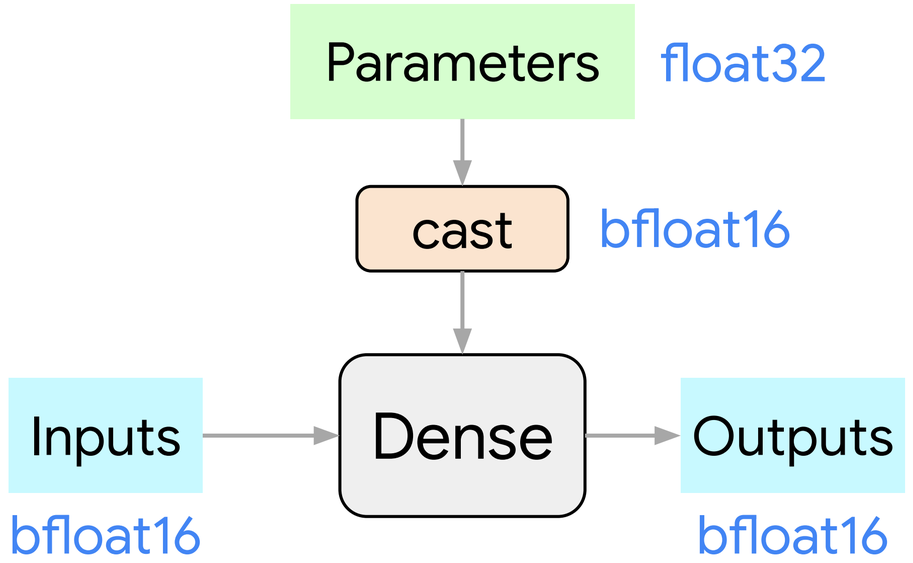


Figure credit: Google Cloud Doc

Mixed Precision

- Key idea: lower the precision of network **activations**
- Keep original parameters etc. in high precision for accurate updates
- Different precision types:
  - float16 (small range, higher precision)
  - bfloat16 (float32 range, low precision)



PyTorch:

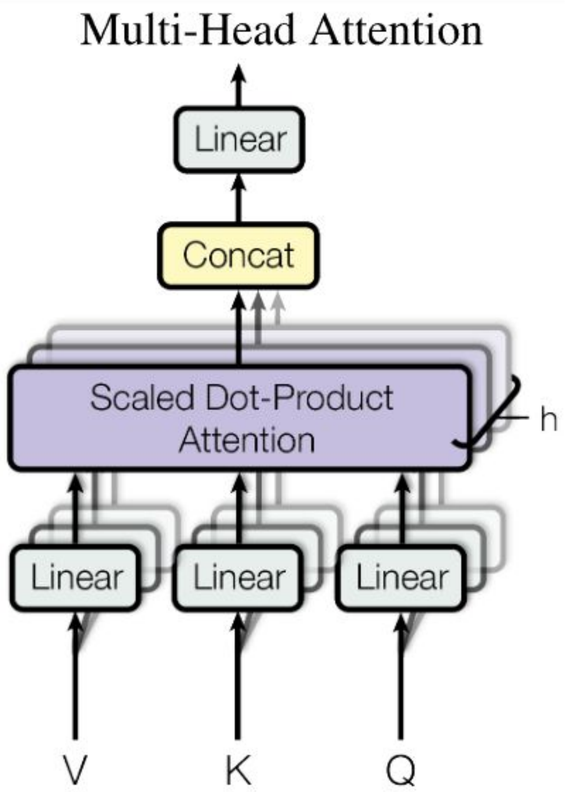
torch.autocast (link)

JAX/Flax:

Module dtype (link)

Mixed Precision

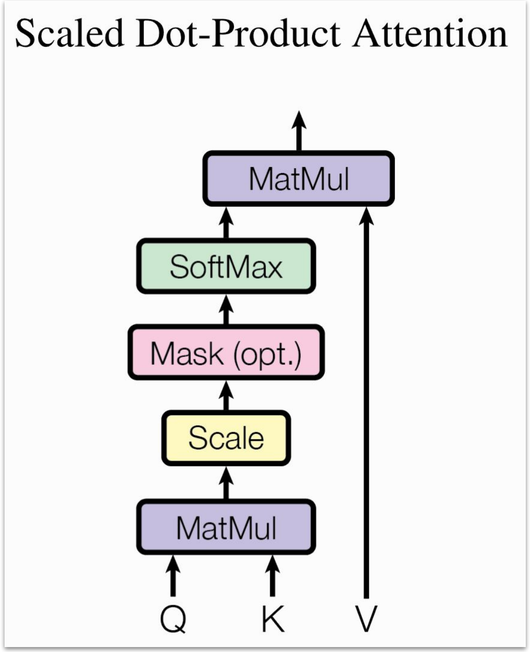
- Case study: LLM Transformer (12 layers, 155M)
- Float32: 20.6GB memory, 2.1 seconds per step
- Bfloat16: 14.6GB memory, 1.1 seconds per step
  - **30% memory reduction**, **2x speed-up**
  - Activations are 50% smaller and accelerator operates faster
  - Equivalent performance



Mixed Precision

Caveats

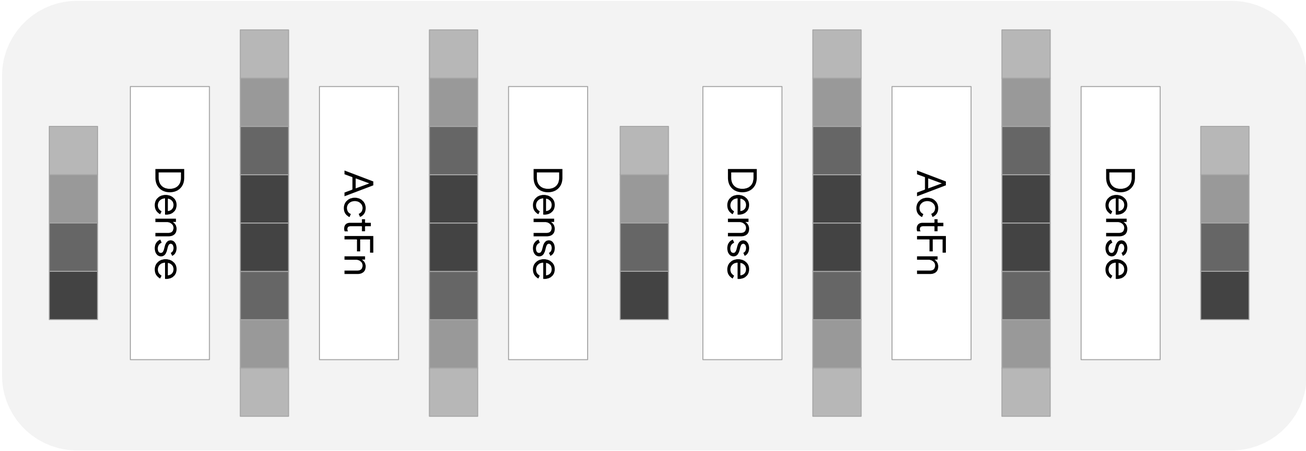
- Large reductions should be kept in high precision for numerical stability
  - Output softmax
  - Attention softmax
    - Possible with bfloat16 in smaller seq lengths
- Lower precision parameters and optimizer possible
  - More tricky, performance might suffer



Gradient Checkpointing

- Storing all activations in memory is very expensive
- Alternative: recompute activations on demand

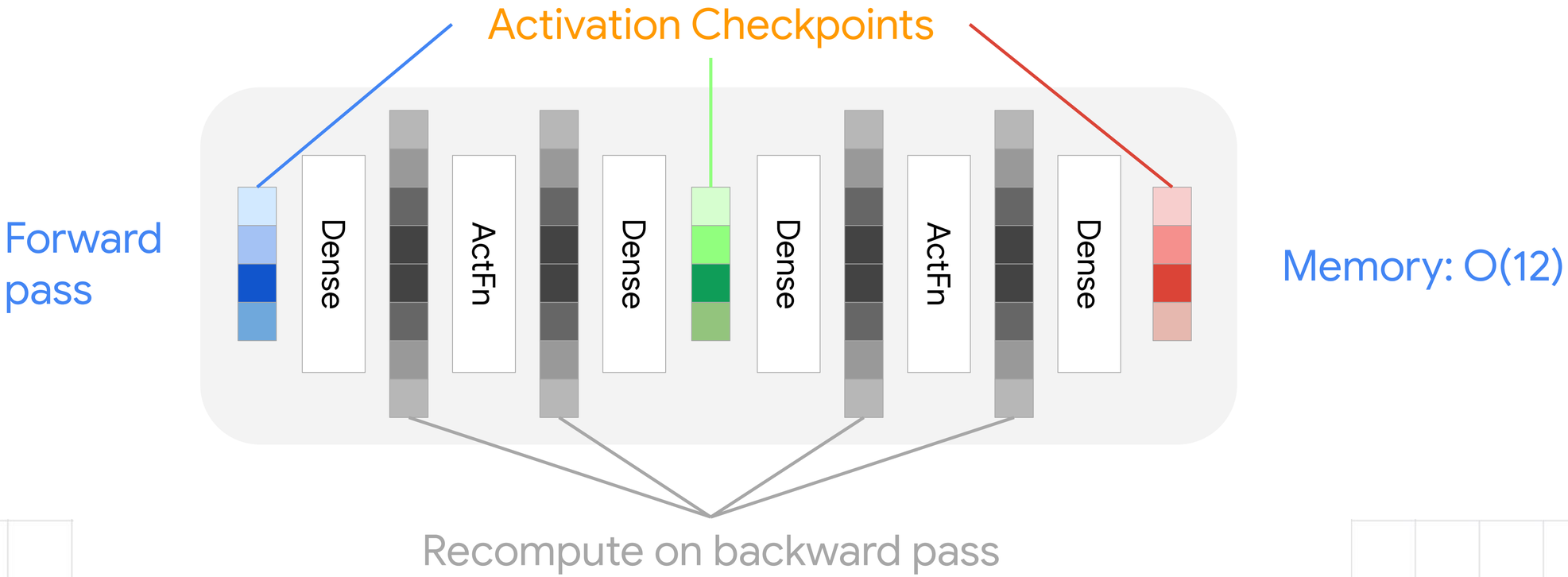
Forward pass



Memory: O(40)

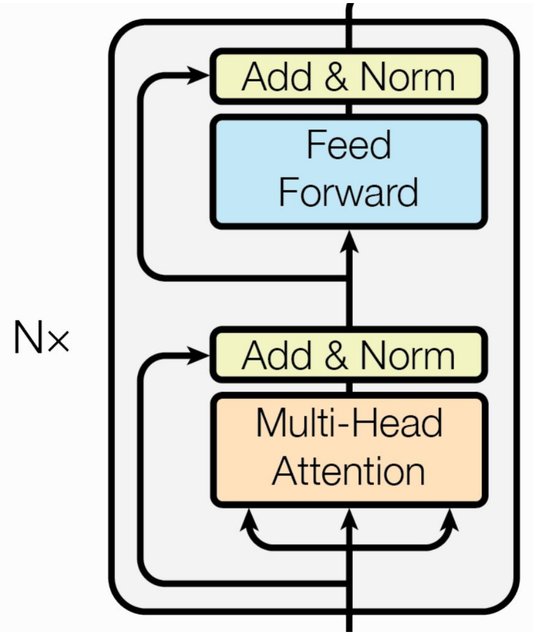
Gradient Checkpointing

- Storing all activations in memory is very expensive
- Alternative: recompute activations on demand



Gradient Checkpointing

- Case study: LLM Transformer (12 layers, 155M)
- Bfloat16 (optimized): 8.8GB memory, 0.7 seconds per step
- Gradient Checkpointing: 3.9GB memory, 0.9 seconds per step
  - **55% memory reduction, 1.25x slow down**
  - Often 90% memory reduction in activations possible
  - Picking the right layers to checkpoint/remat is important



PyTorch:

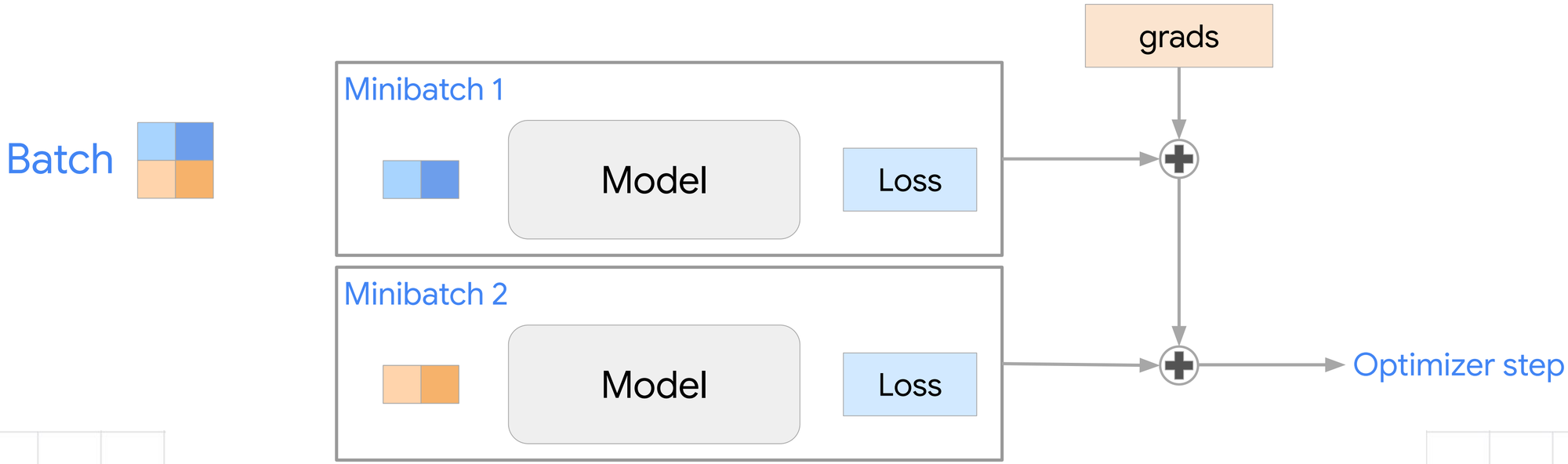
torch.utils.checkpoint (link)

JAX/Flax:

- jax.remat (link)
- flax.nn.remat (link)

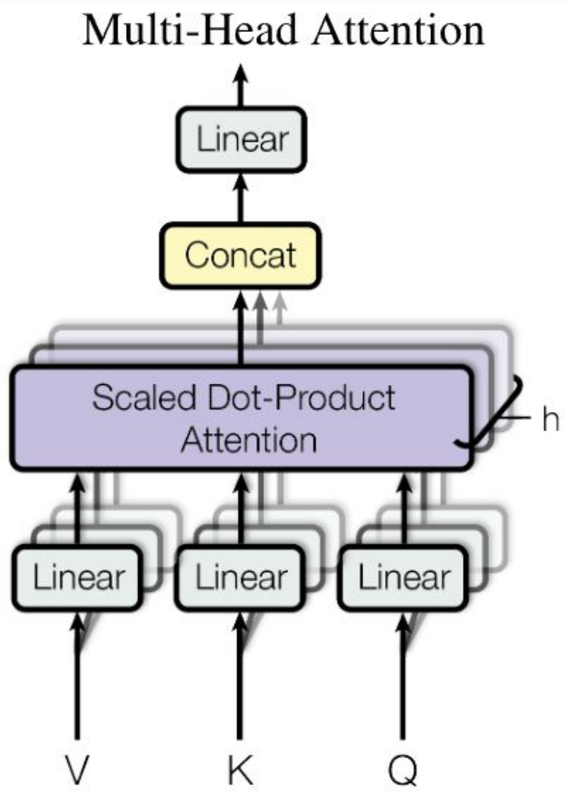
Gradient Accumulation

- Large-model optimizations often require large batch sizes
- Memory only fits small batches
- Idea: accumulate gradients of multiple smaller batches before optimizing



Gradient Accumulation

- Case study: LLM Transformer (12 layers, 155M)
- Gradient Accumulation: 3.9GB memory, 0.91 seconds per step
- Single batch: 6.2GB memory, 0.86 seconds per step
  - **40% memory reduction**, **1.06x slow down**
  - Simple to implement
  - Not always possible to fully utilize the accelerator



CPU Offloading

- If model parameters are larger than GPU HBM, can offload into CPU RAM
  - Also for optimizer parameters
- Load layers one-by-one as needed
- Very slow transfer speed, only as final gamble if nothing else works

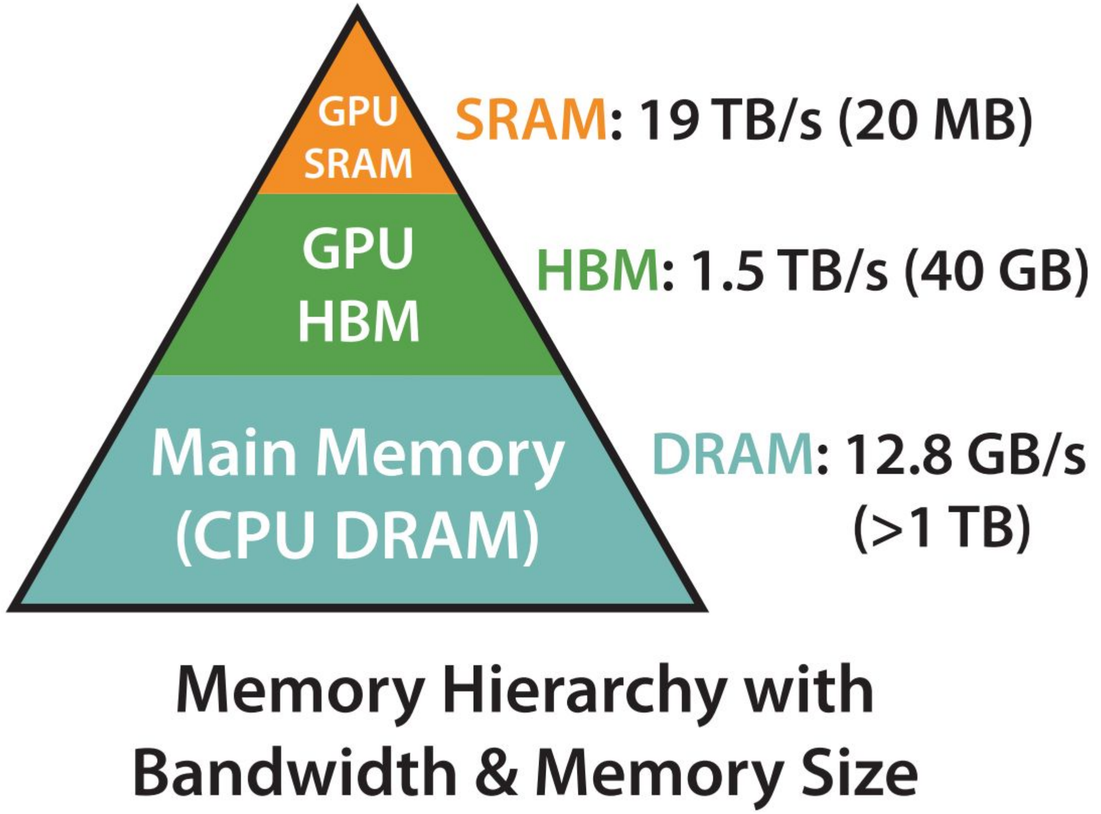


Figure credit: Dao et al., 2022

Profiling

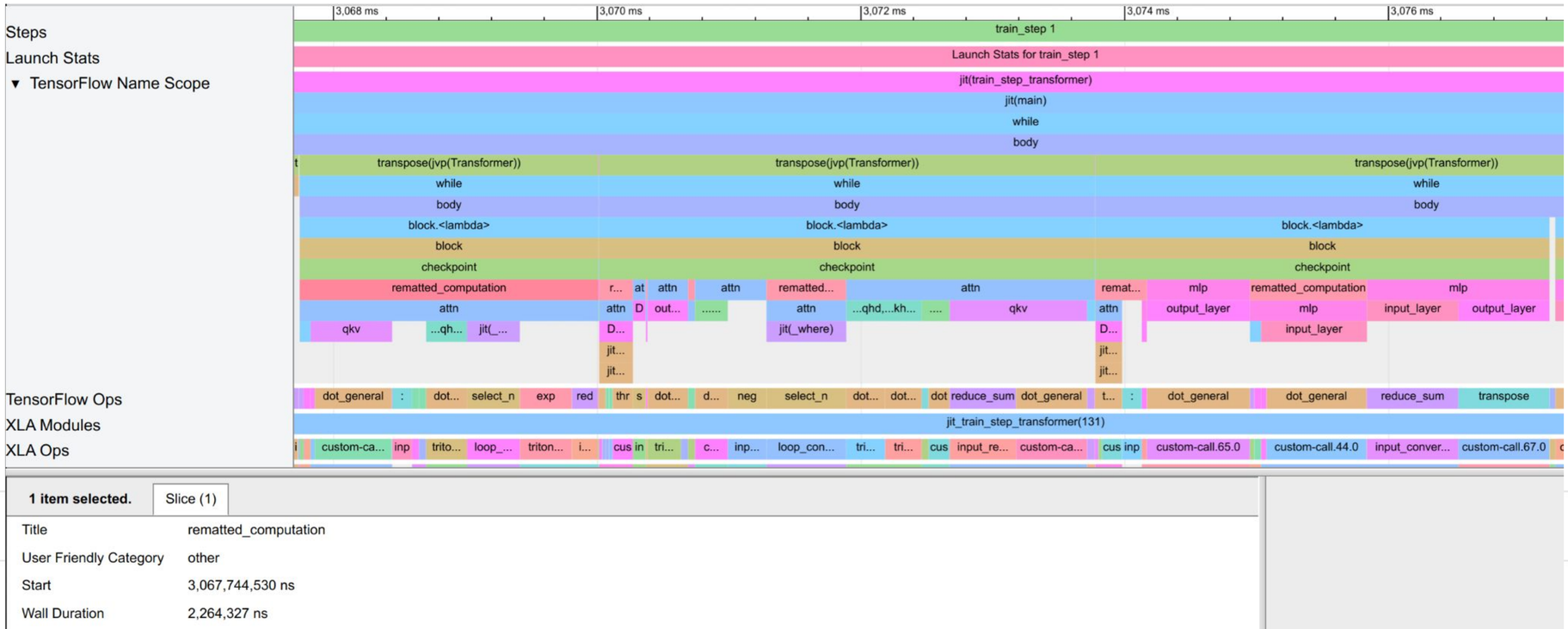
- Crucial to understand the execution of your model
  - Find bottlenecks, memory limitations, “silent” bugs, etc.
- Profiling model execution

PyTorch:

torch.profiler.profile (link)

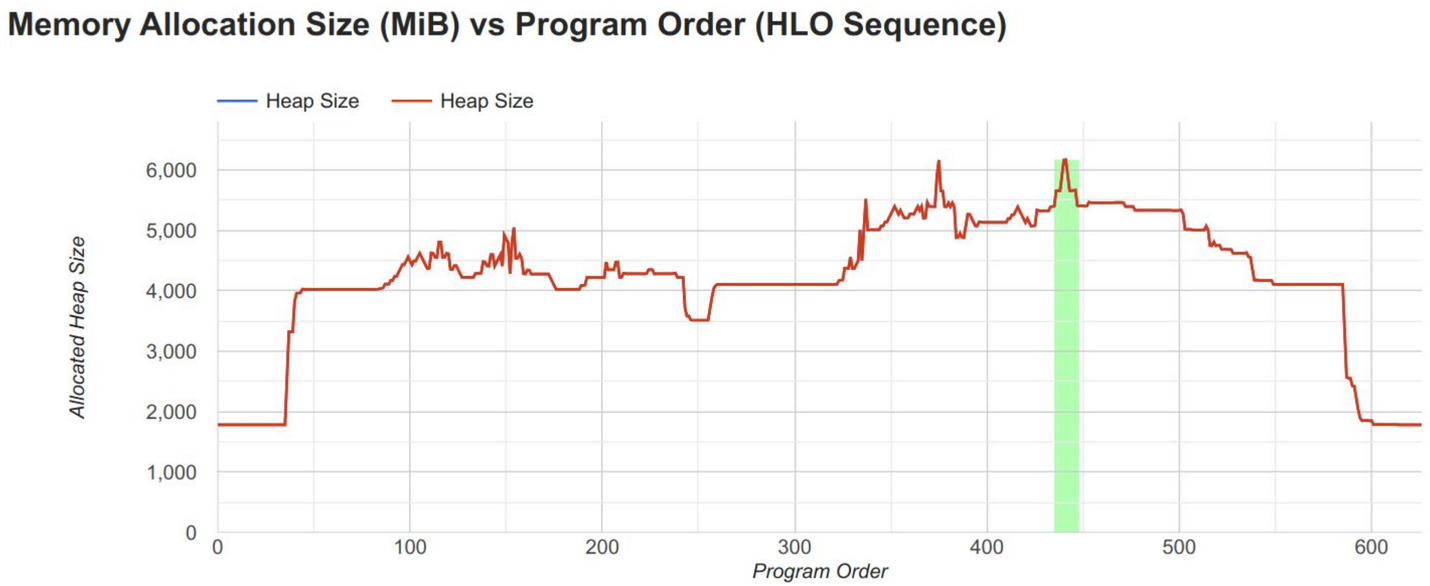
JAX:

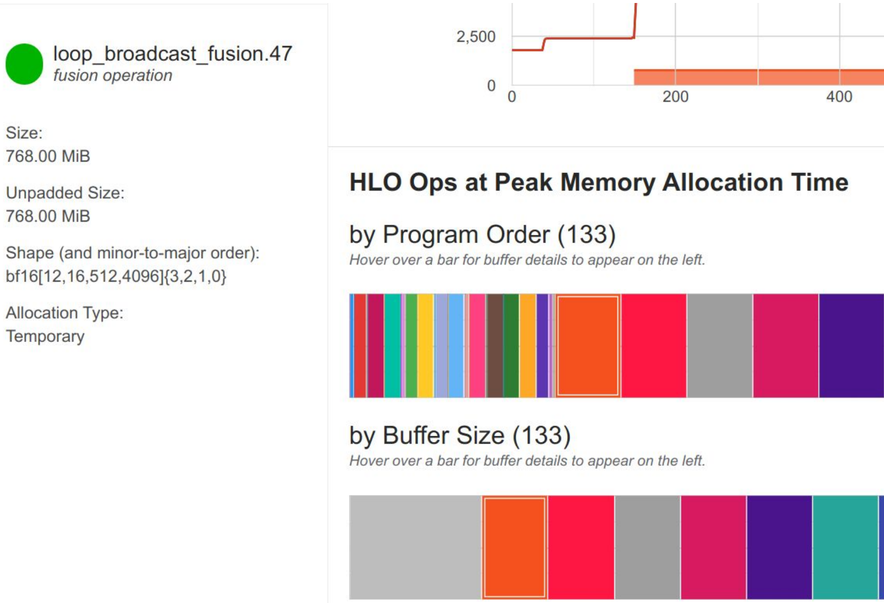
jax.profiler.start_trace (link)



Profiling

- Memory profiling
  - Biggest tensors, when created/free'd, check precision, etc.





Tutorial: [How to read the profile]

Per-Device Optimizations

Summary

Speed-Ups

- Compilation
- Mixed Precision
- Optimized kernels

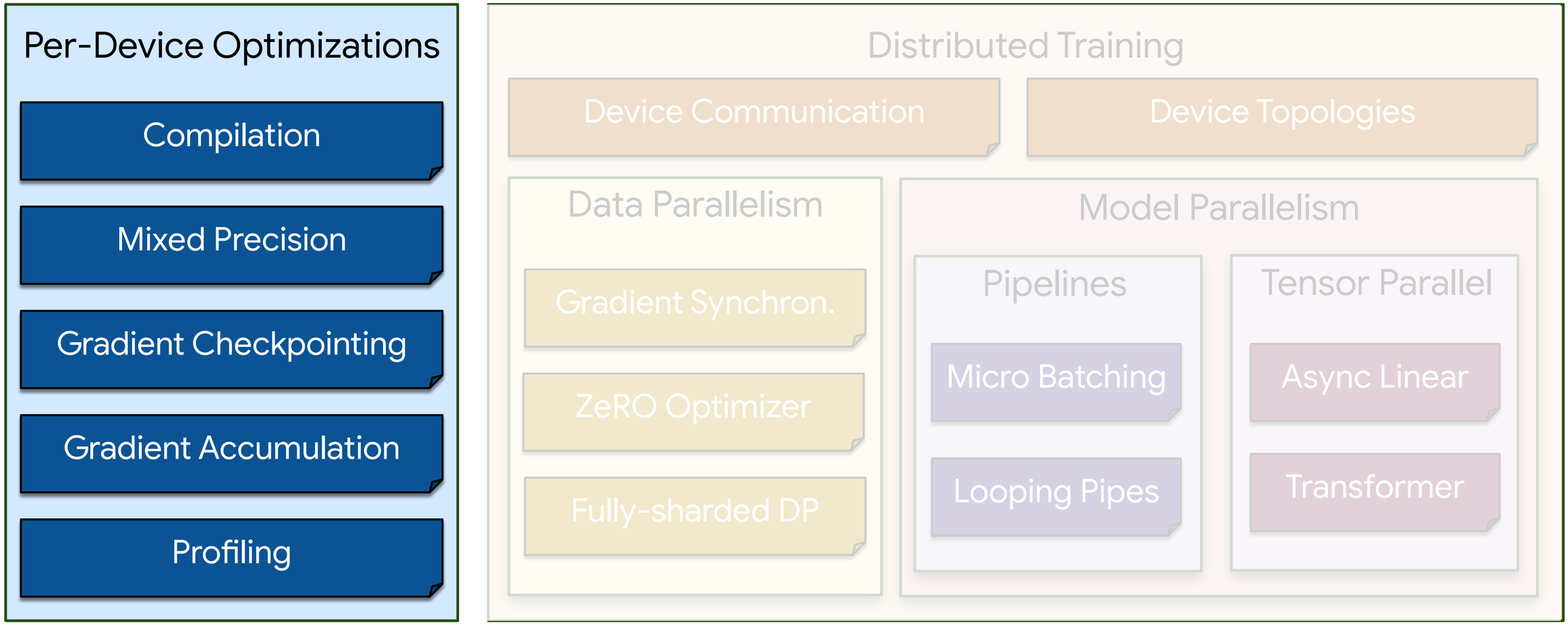
Memory Reduction

- Compilation
- Mixed Precision
- Gradient Checkpointing
- Gradient Accumulation
- CPU Offloading
- Lighter Optimizers

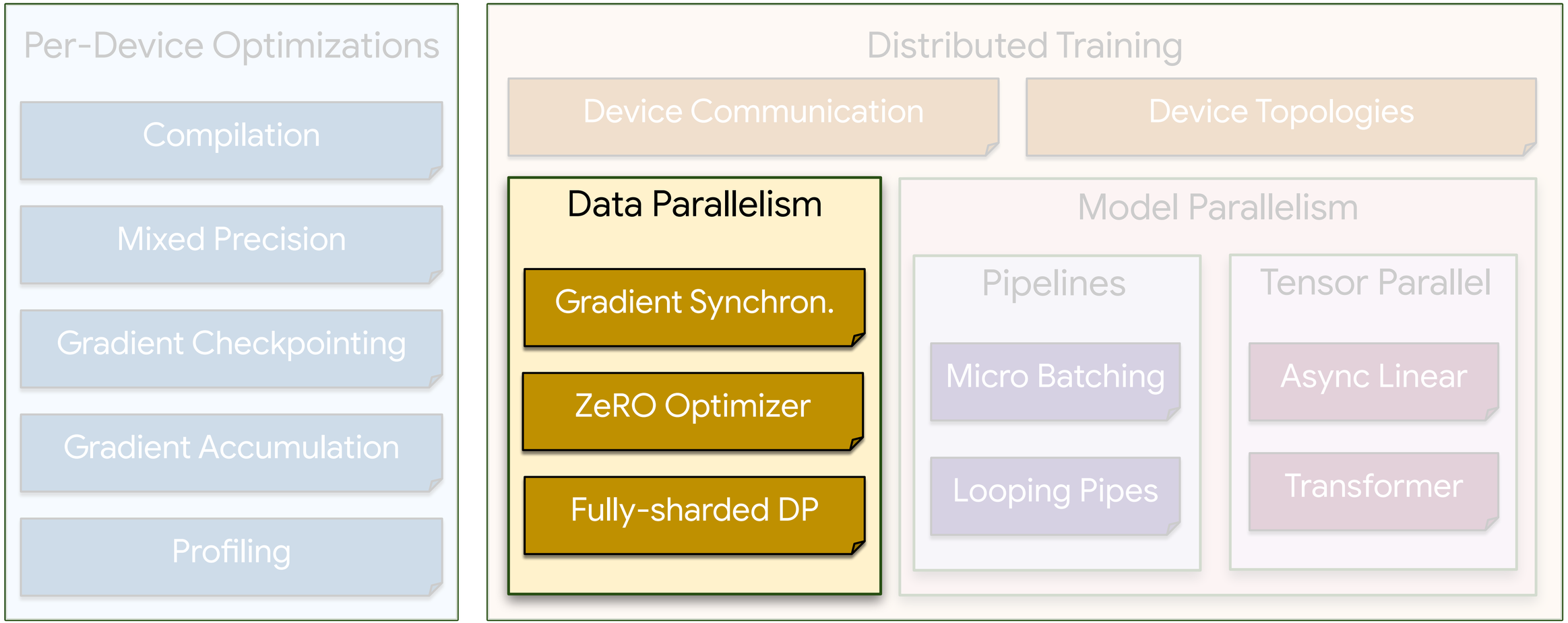
Model Inspection

- Profiling

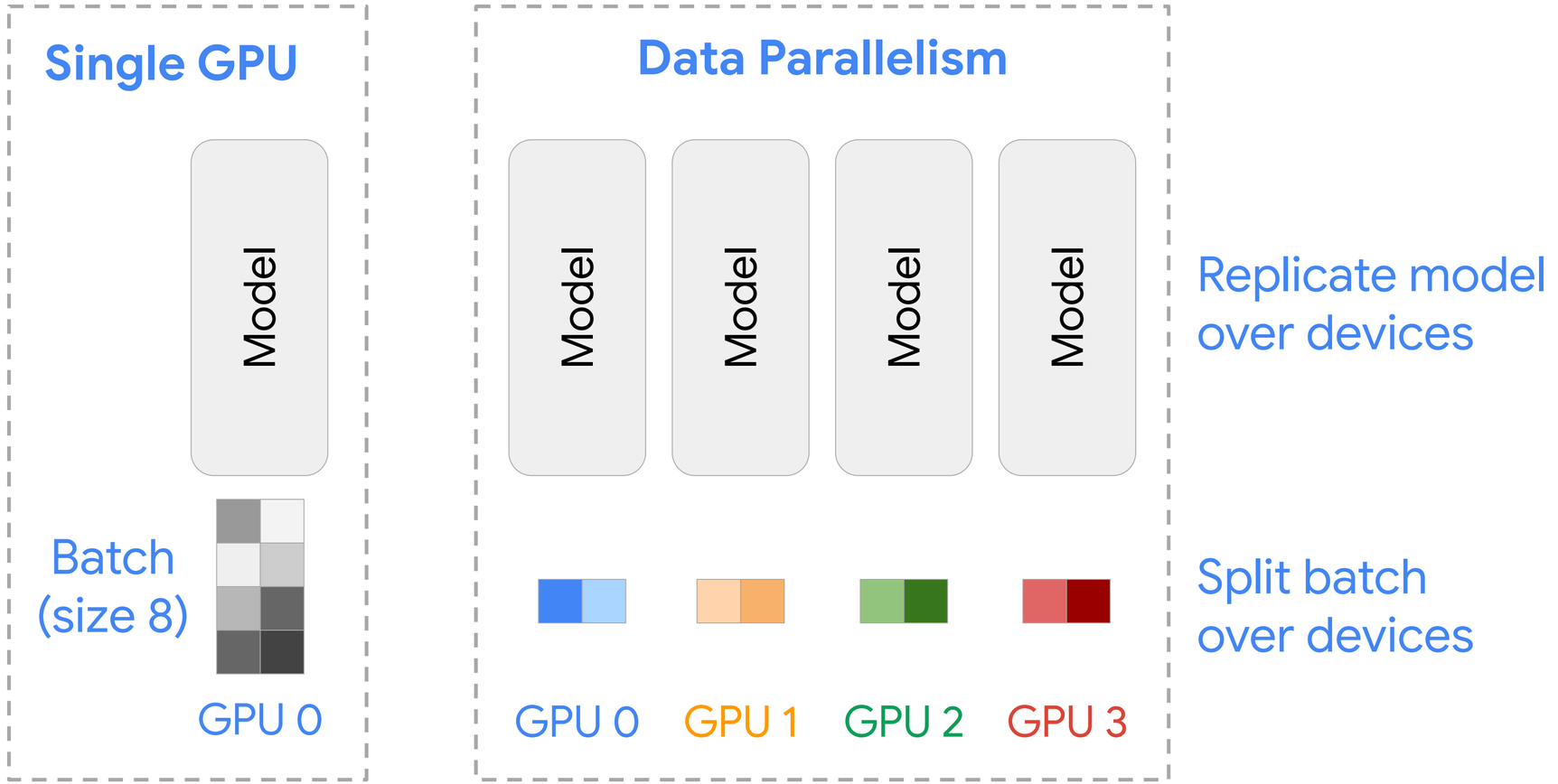
Overview


Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

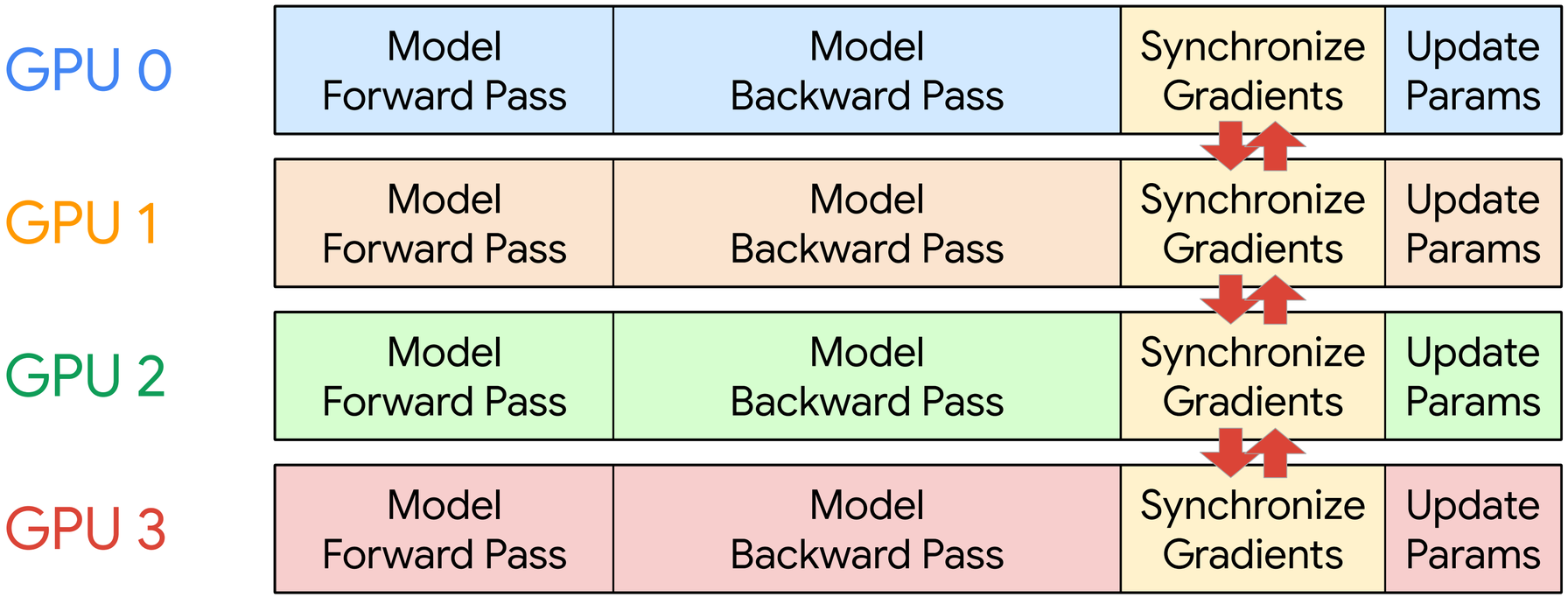
Overview


Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

Data Parallelism

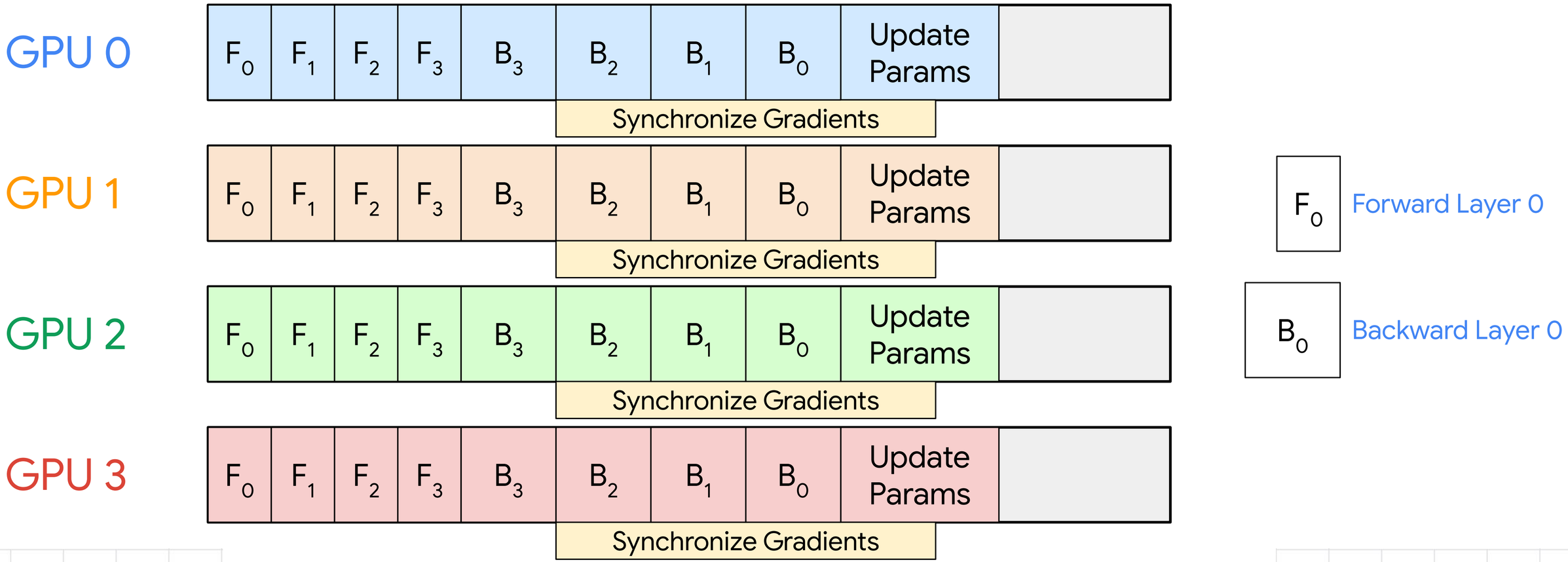


Data Parallelism



Data Parallelism

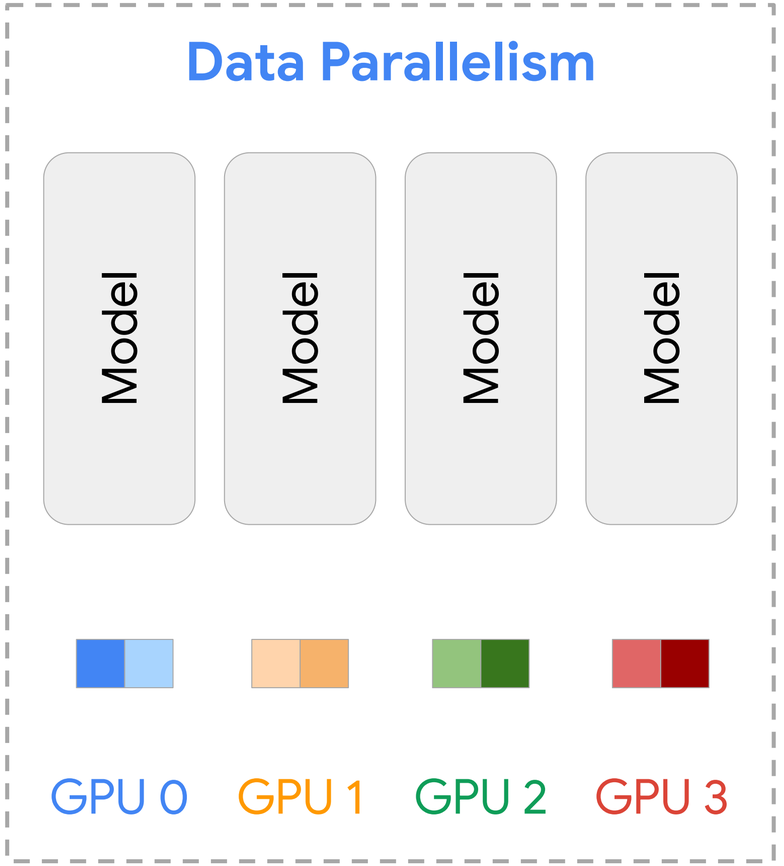
Asynchronous Communication



Data Parallelism

Sharding

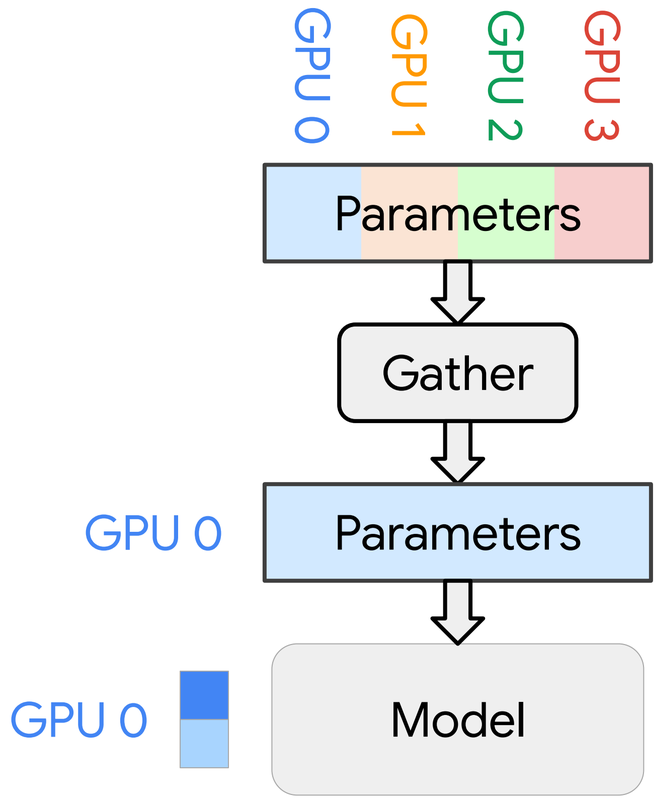
- At 1 billion parameters, replicating model is expensive
  - 16 bytes per parameter ⇒ 16GB on each device



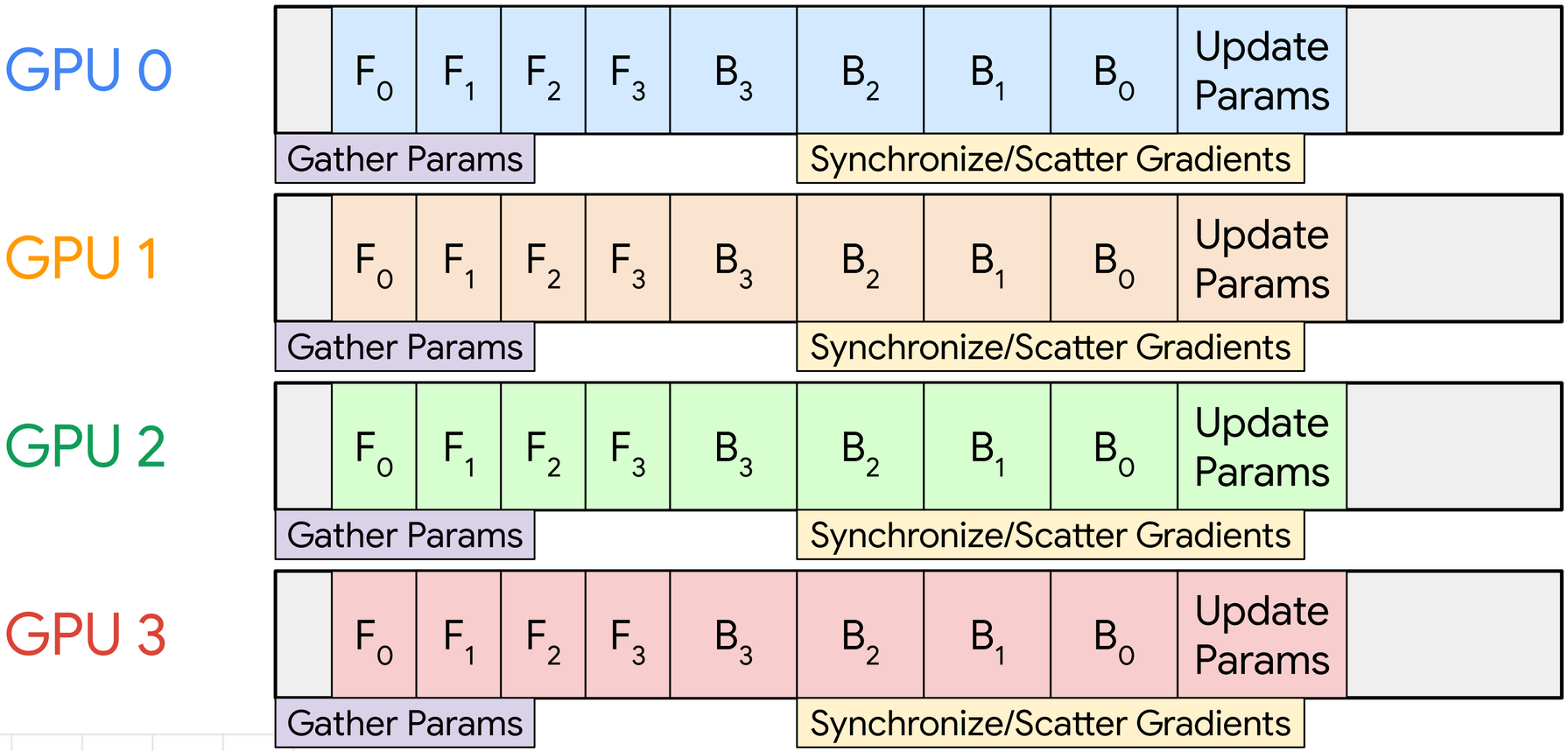
Data Parallelism

Sharding

- At 1 billion parameters, replicating model is expensive
  - 16 bytes per parameter ⇒ 16GB on each device
- Solution: fully-sharded data parallelism
  - Each device holds a part of the model parameters
  - Before model execution, we gather the parameters and continue as usual

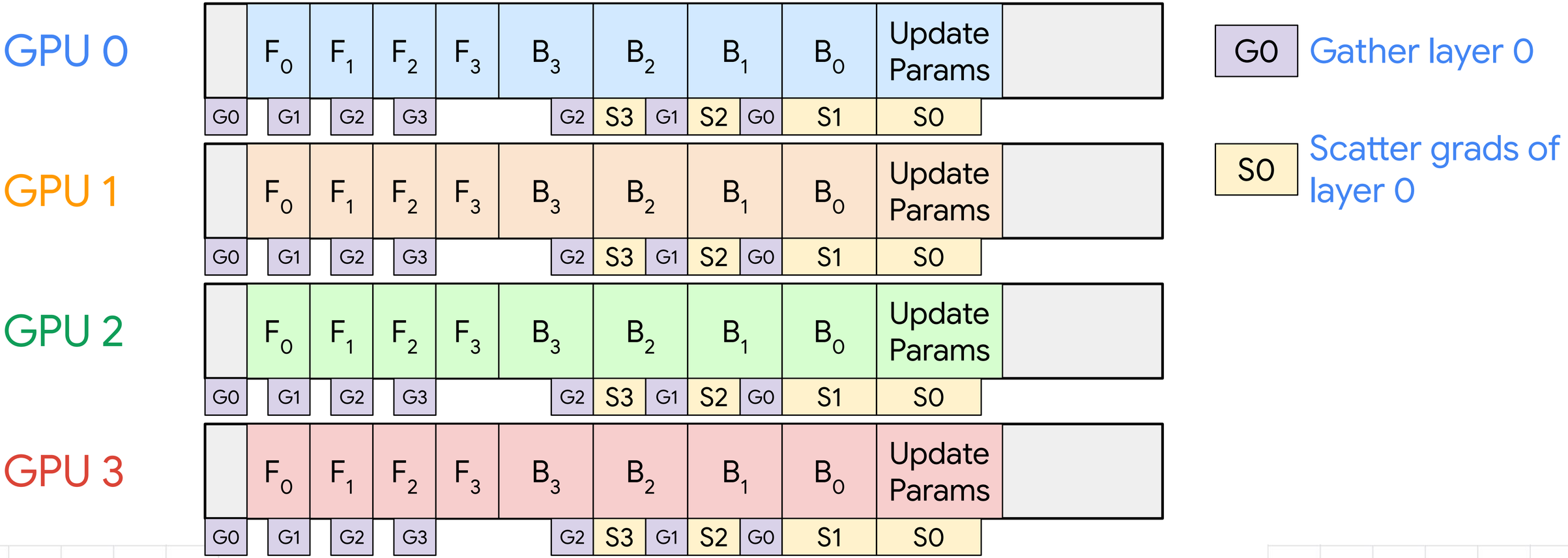


Fully-Sharded Data Parallelism

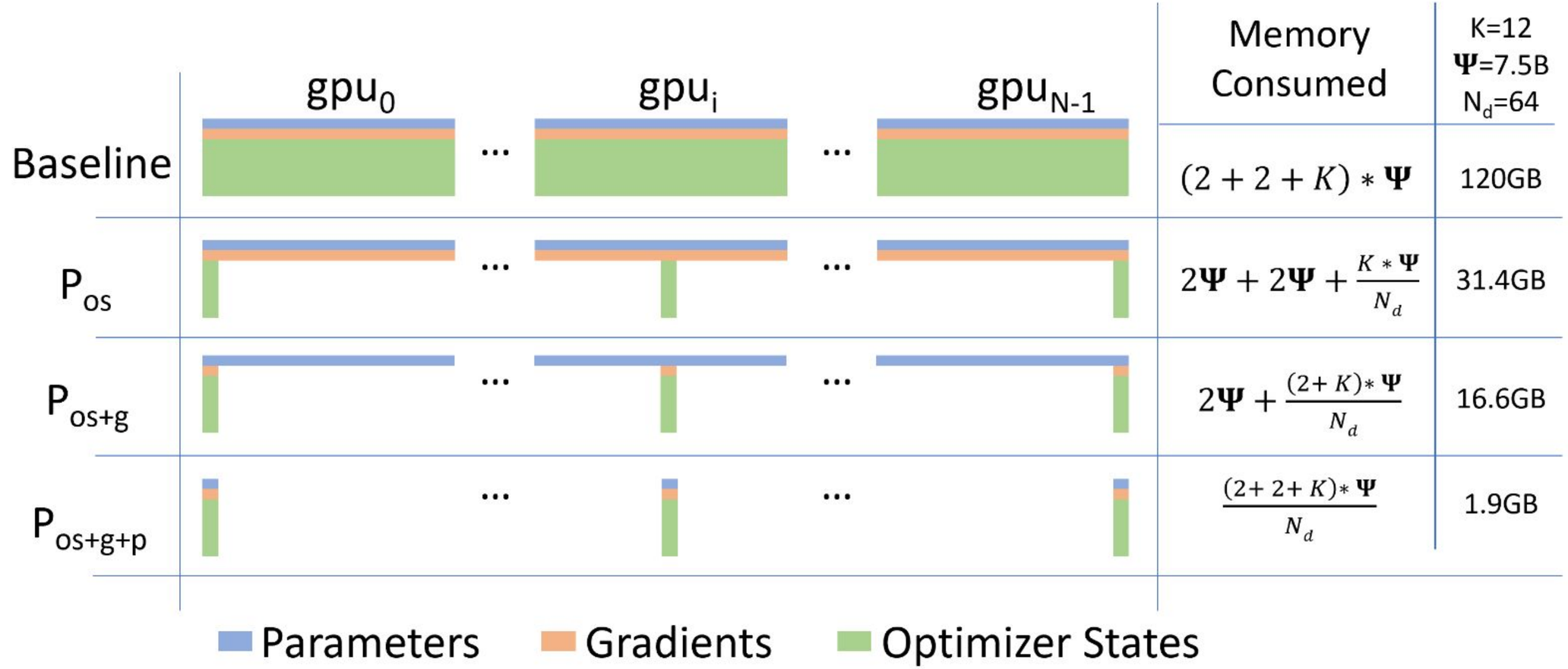


Fully-Sharded Data Parallelism

Gradient Checkpointing



ZeRO Optimizer



Rajbhandari et al., 2019

Data Parallelism

Summary

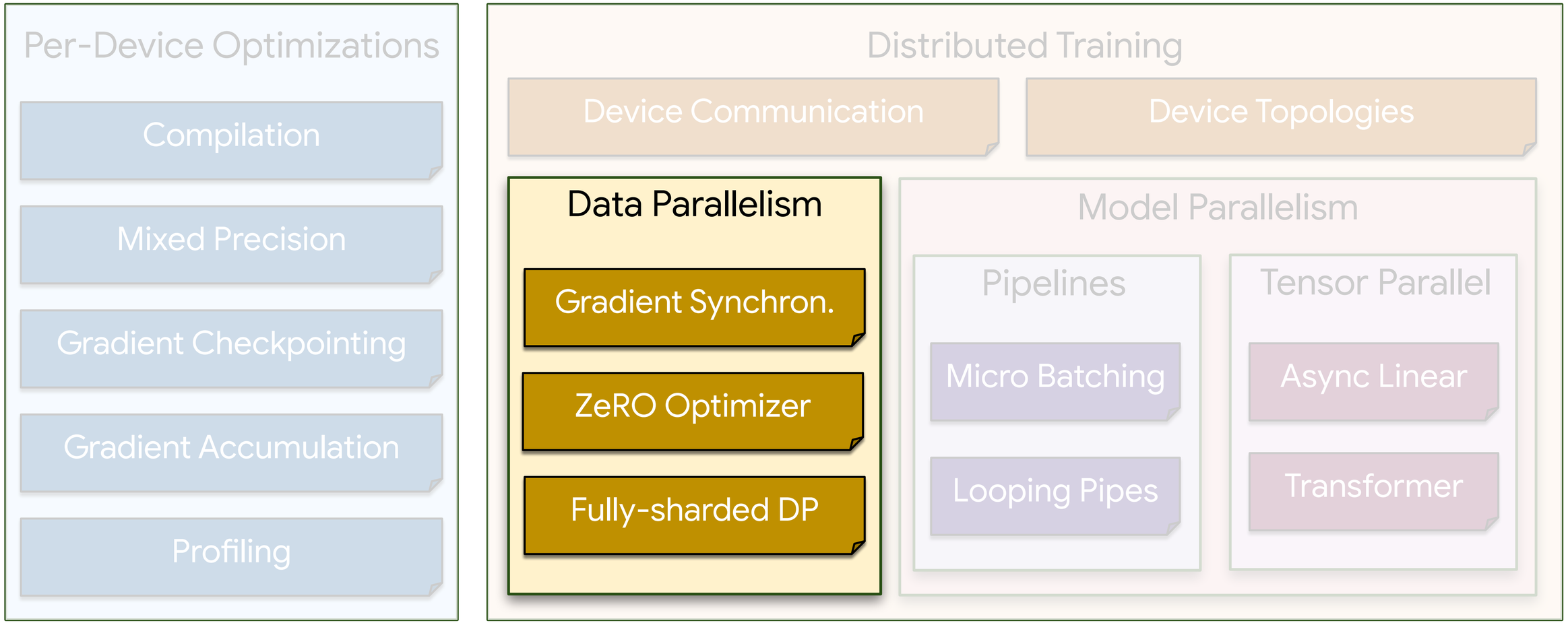
Benefits

- Easy to implement
- Low communication costs
- Parameter sharding reduces memory footprint

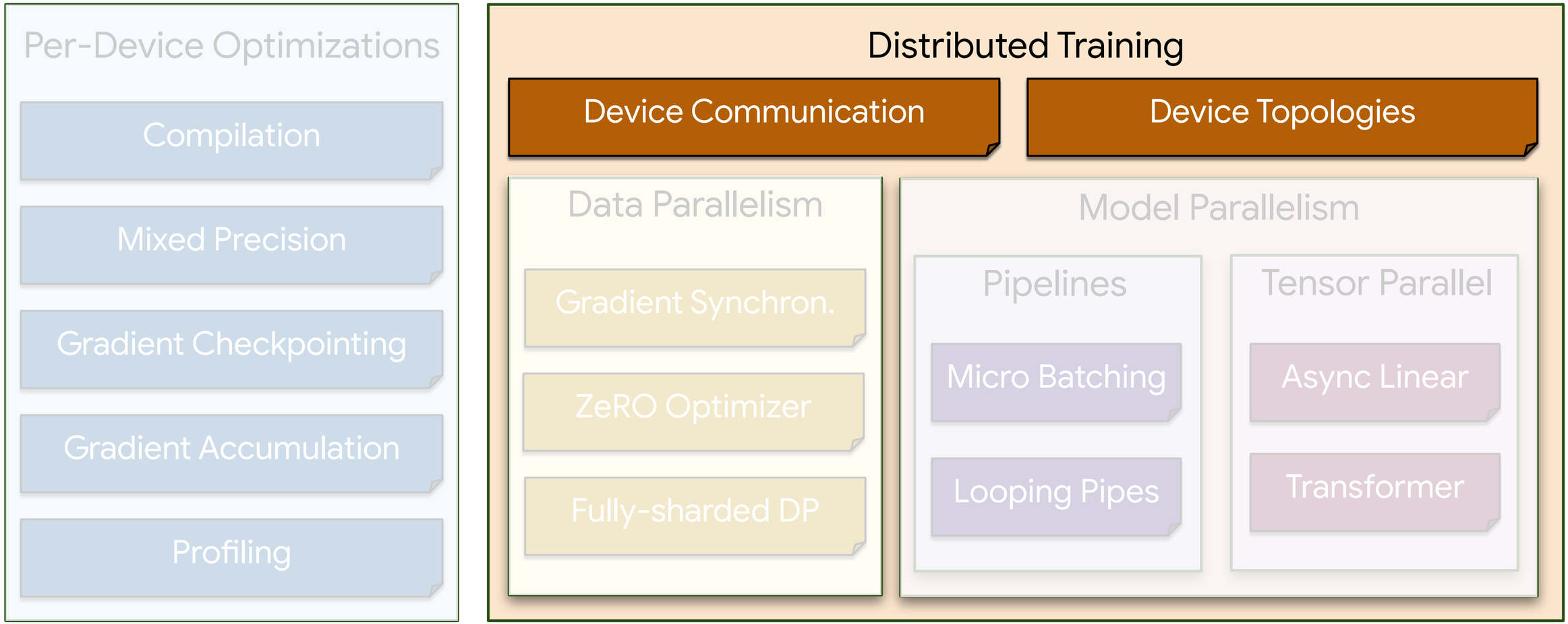
Drawbacks

- Each device runs the full model
- Parameter sharding becomes inefficient for slow/large clusters

Overview


Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

Overview


Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

Device Communications

- Three key communication operations:
  - all_gather
  - reduce_scatter
  - ppermute

There exist more, but we can reduce most of them to these basic blocks

Device Communications

all_gather

- Replicate across devices
- Forward pass FSDP

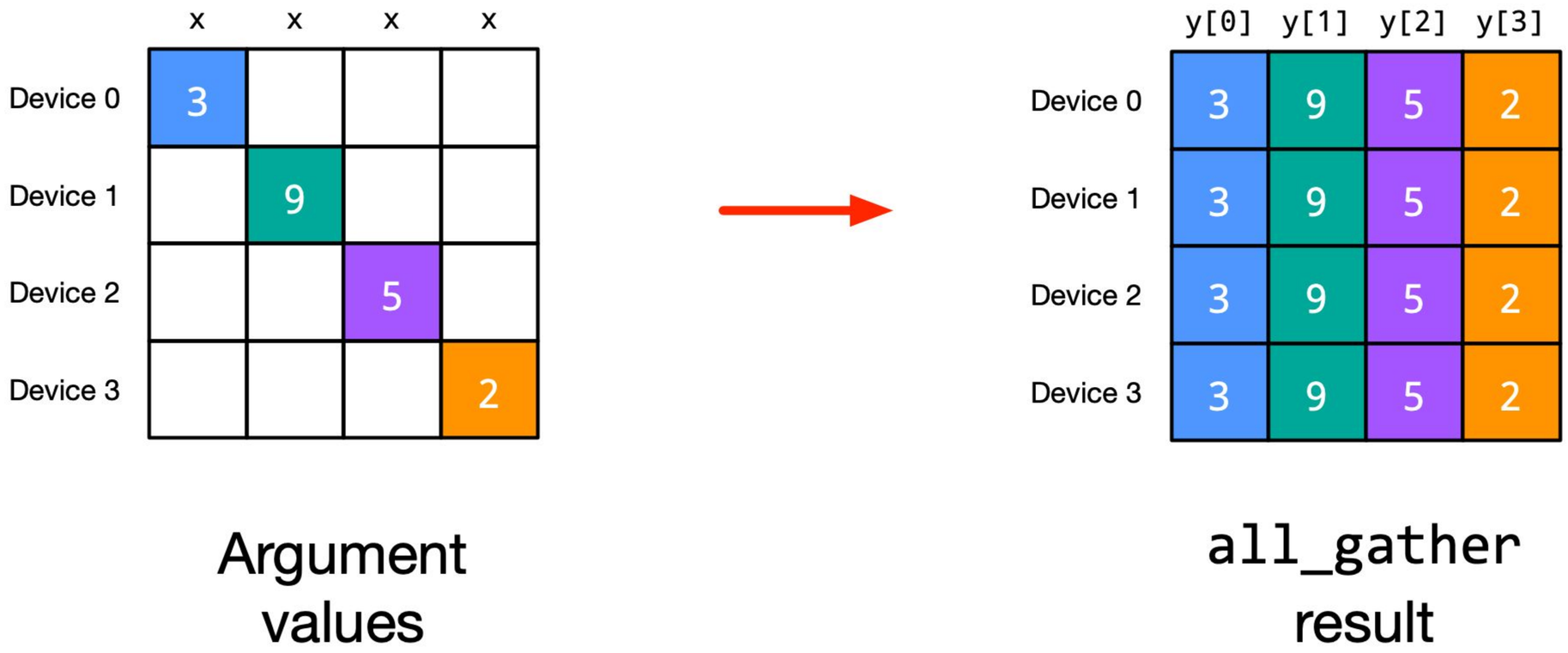


Figure credit: JAX Team

Device Communications

reduce_scatter

- Gradient fn of all_gather
- Backward pass FSDP
- Other reduce ops possible

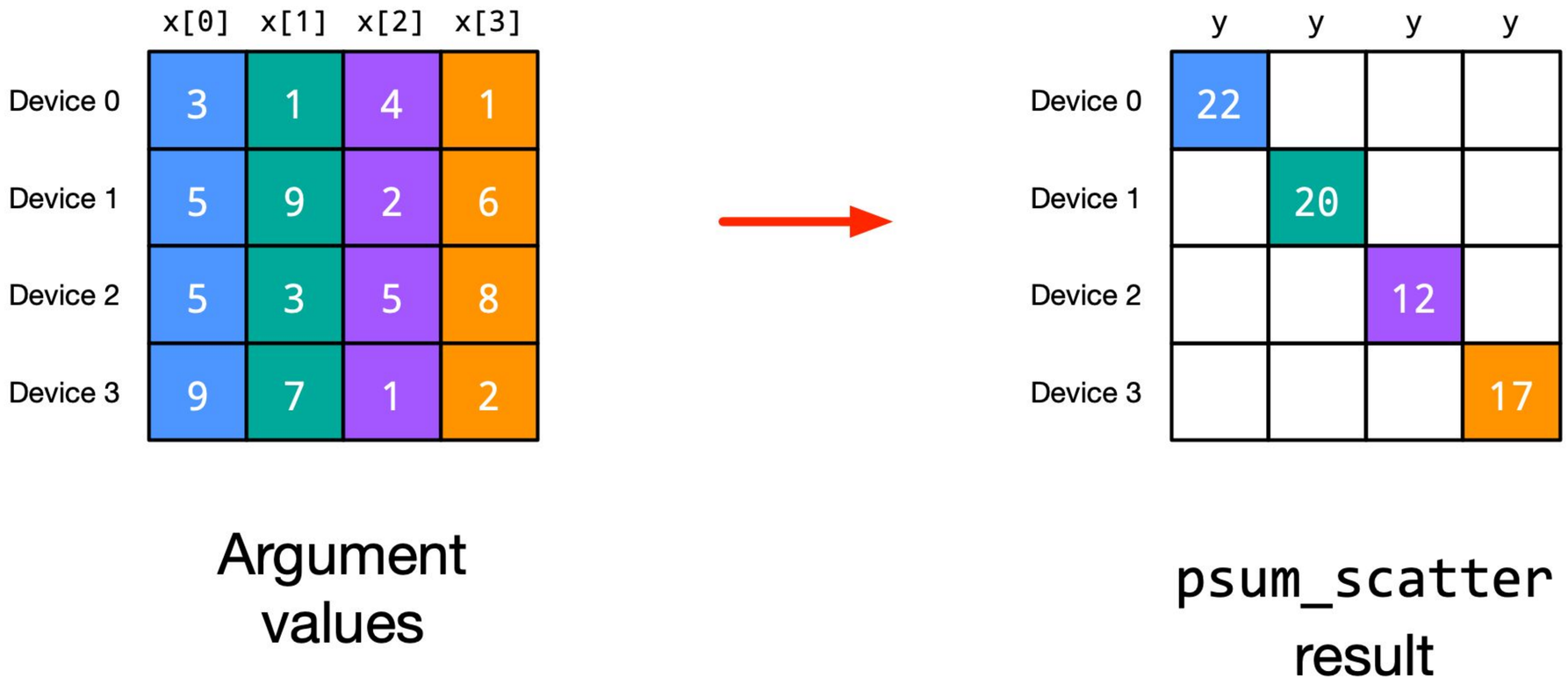


Figure credit: JAX Team

Device Communications

ppermute

- Round-robin communication
- Ideal on a ring
- Can implement other comms. (more later)

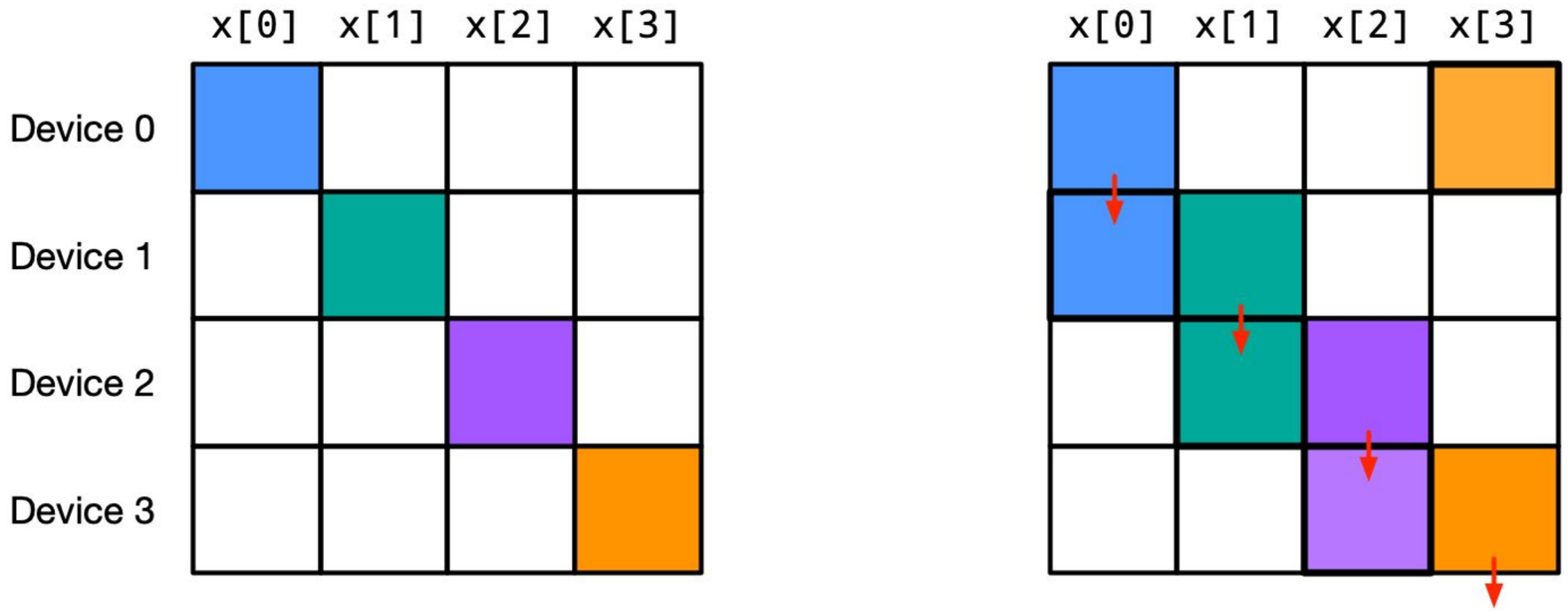
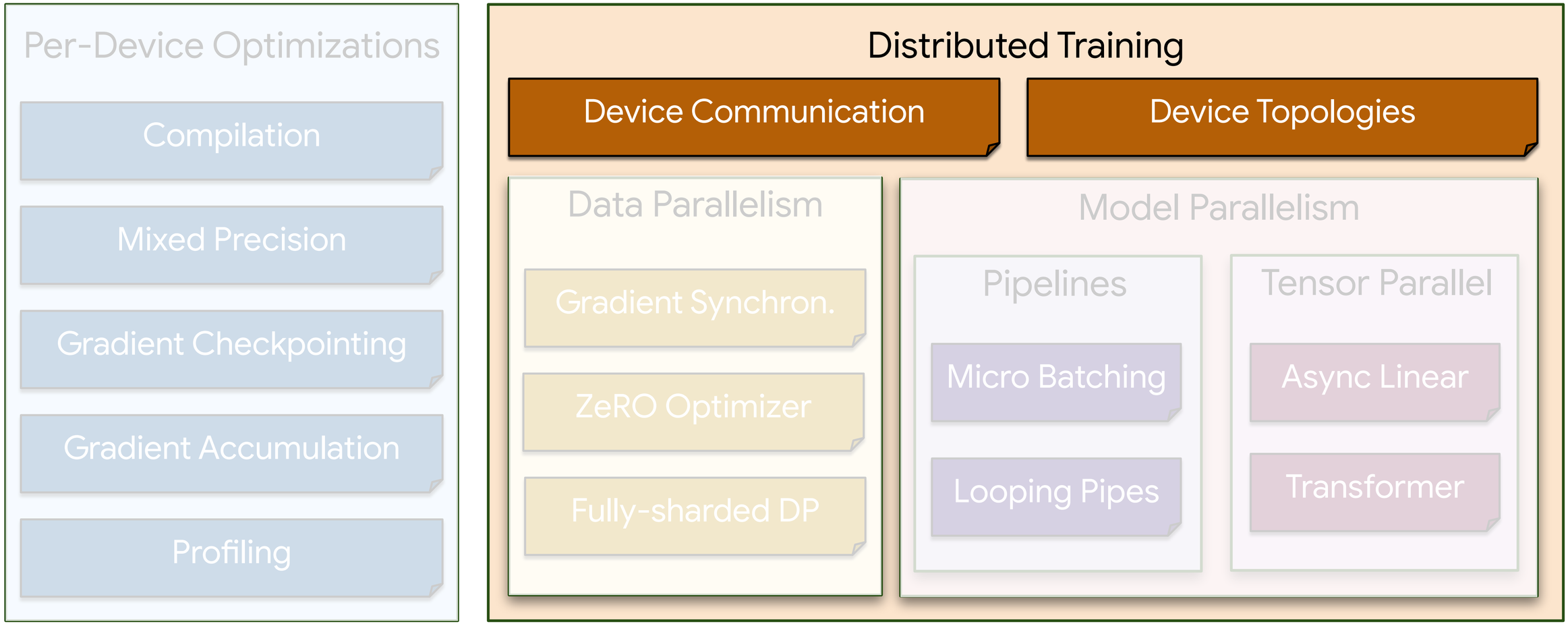
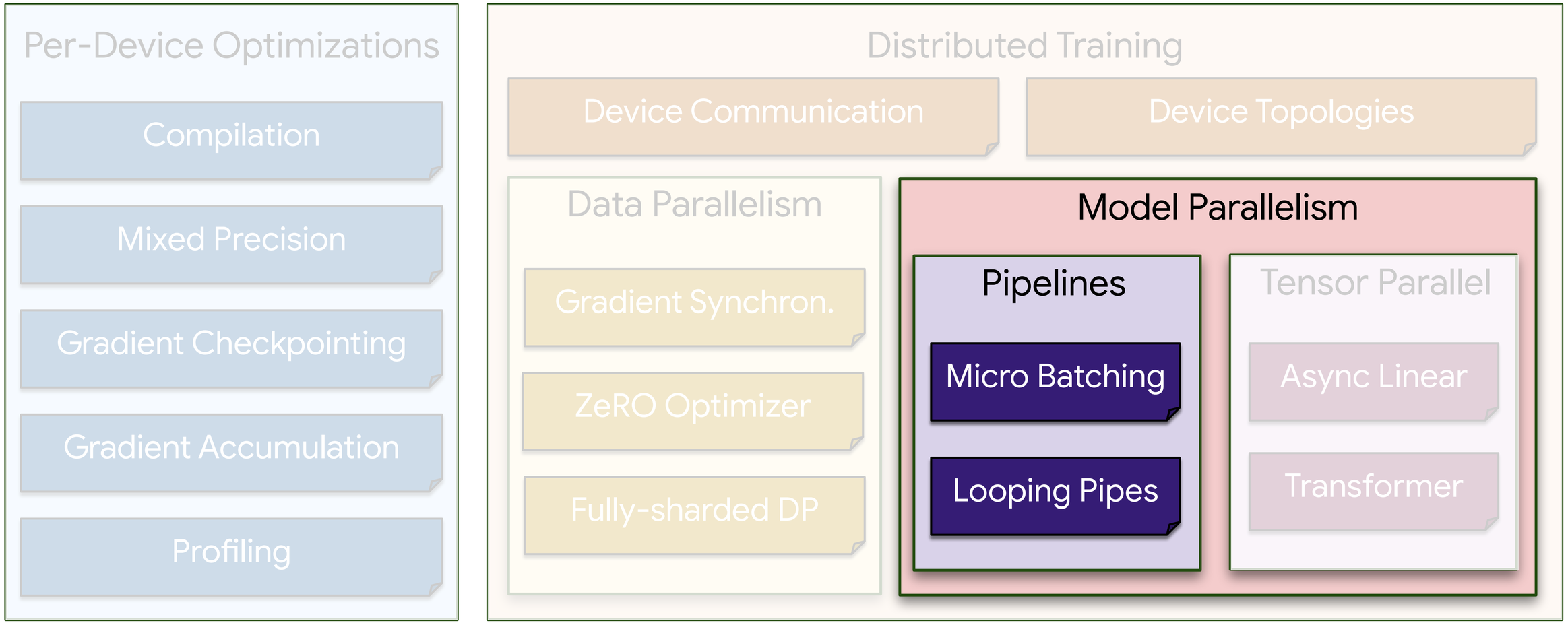


Figure credit: JAX Team

Overview

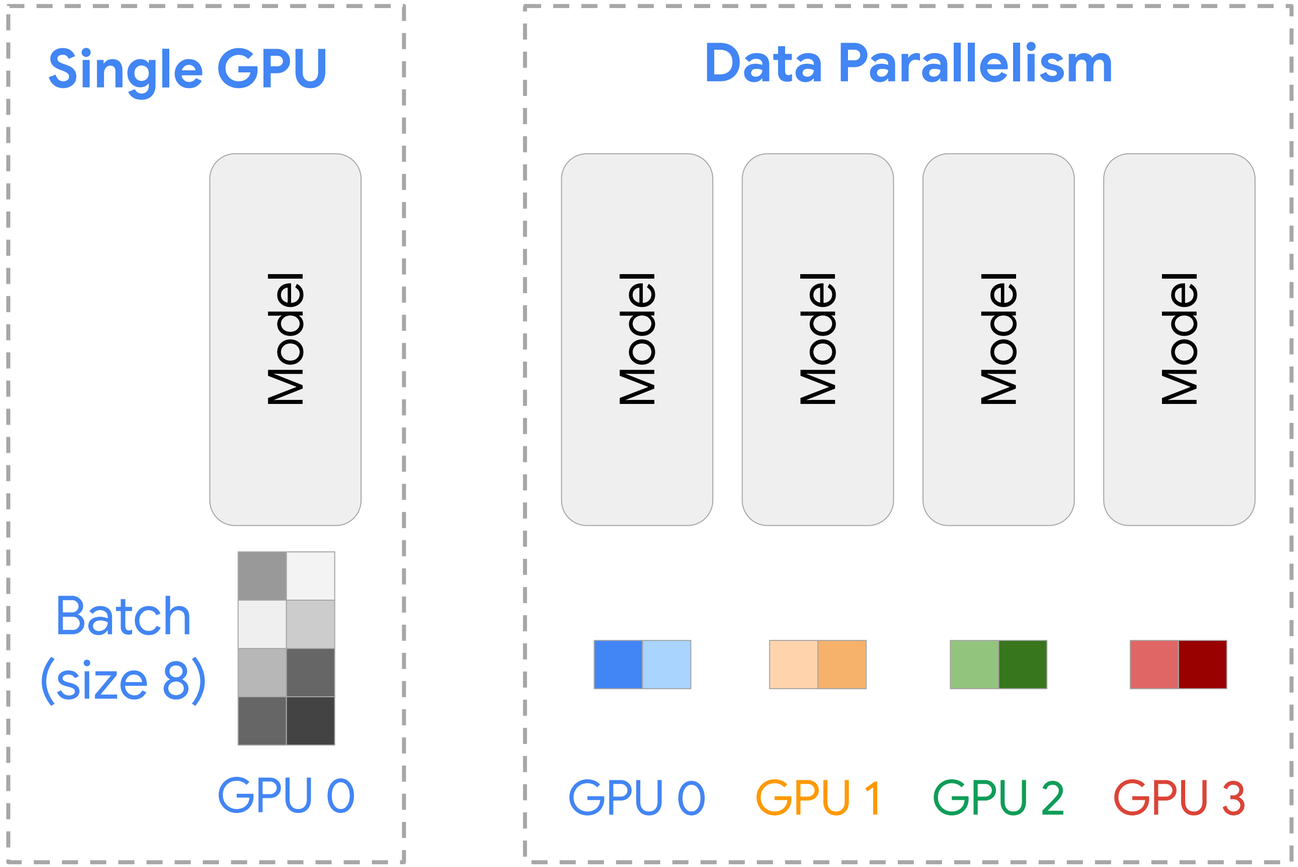

Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

Overview


Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

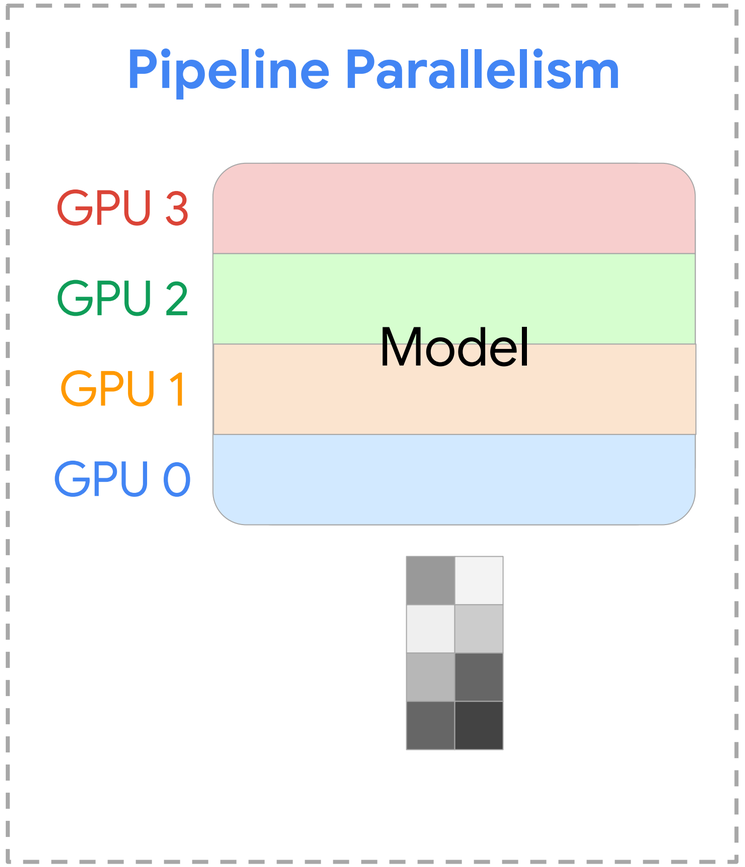
Model Parallelism

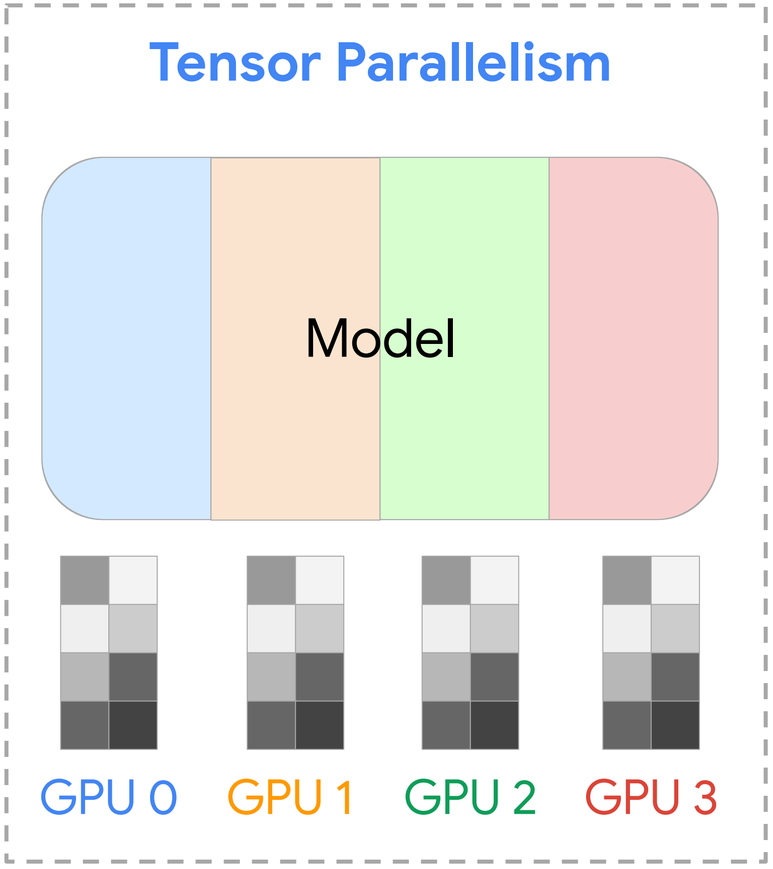
- Fully-sharded data parallelism becomes inefficient over many nodes
  - Slow communication
  - Tiny batch sizes can't achieve max utilization
- Alternative: parallelize the model itself



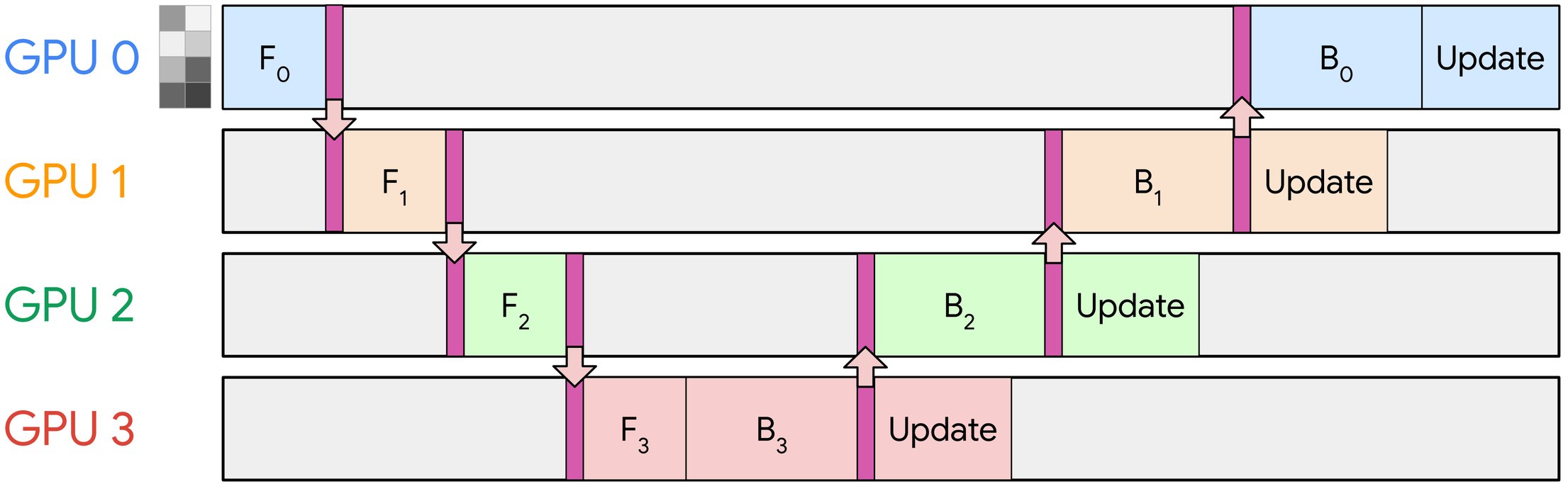
Model Parallelism

- **Pipeline** parallelism distributes over **layers**
- **Tensor** parallelism distributes over **features**

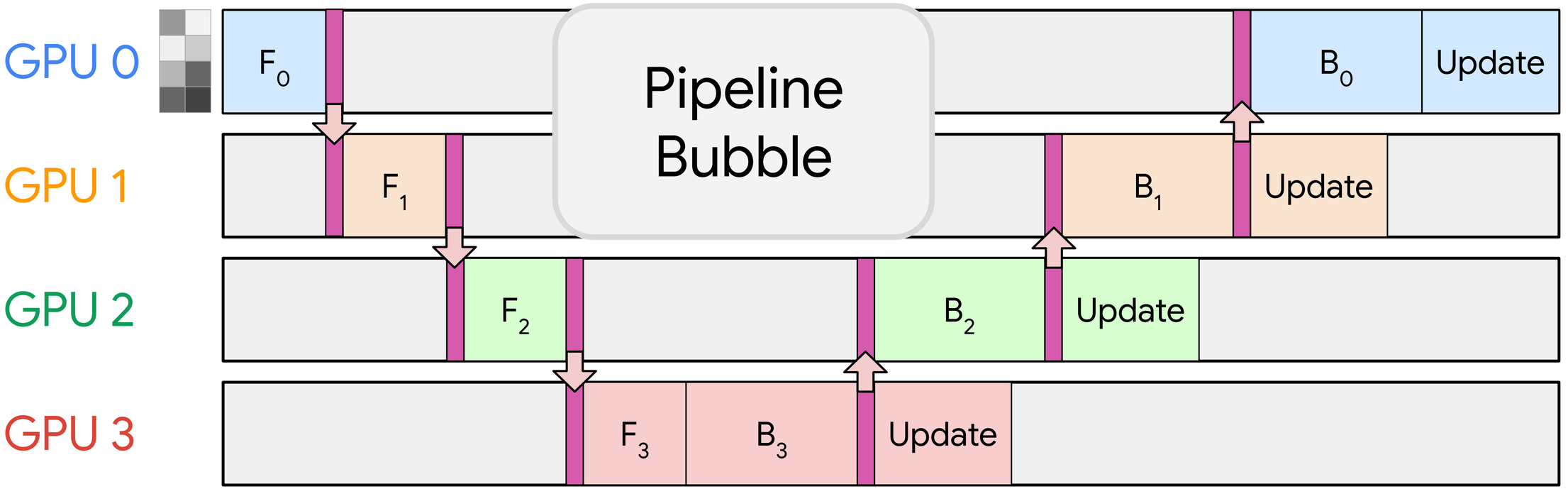




Pipeline Parallelism



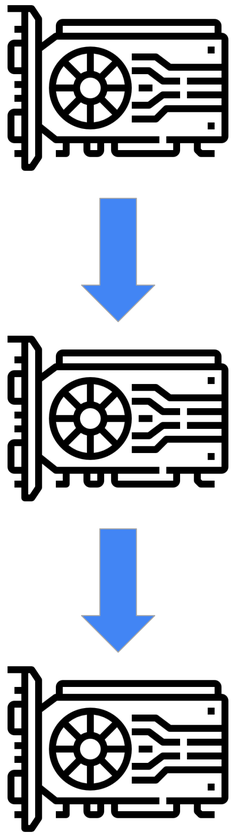
Pipeline Parallelism



Pipeline Parallelism

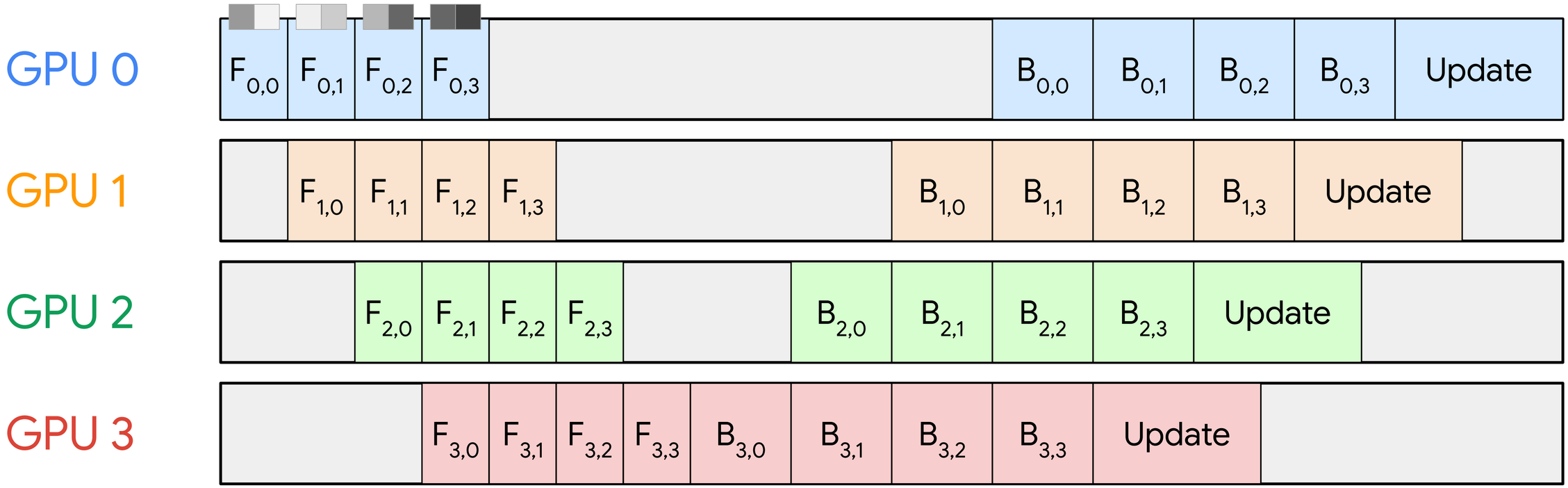
Pipeline Bubble

- Pipelines struggle with keeping devices busy
- Simple idea: Micro-Batching
  - Split batch into multiple smaller batches
  - Run batches one at a time
  - Communicate as soon as one microbatch finishes



Pipeline Parallelism

Micro-Batching



$F_{i,j}$ Forward pass, layer i, micro-batch j

Handcrafted Pipeline Schedules

- More complex schedules for increased speed and memory efficiency
  - Combining with gradient accumulation to free memory of early microbatches
- Increased implementation complexity
- More toolkits out there for [PyTorch]

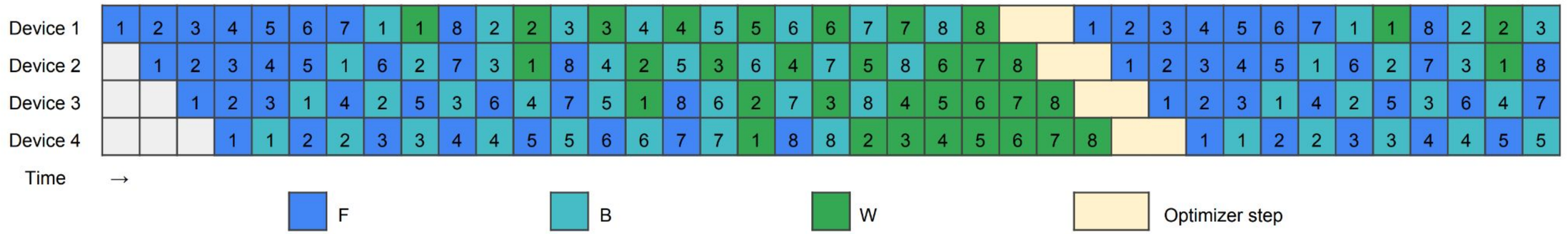
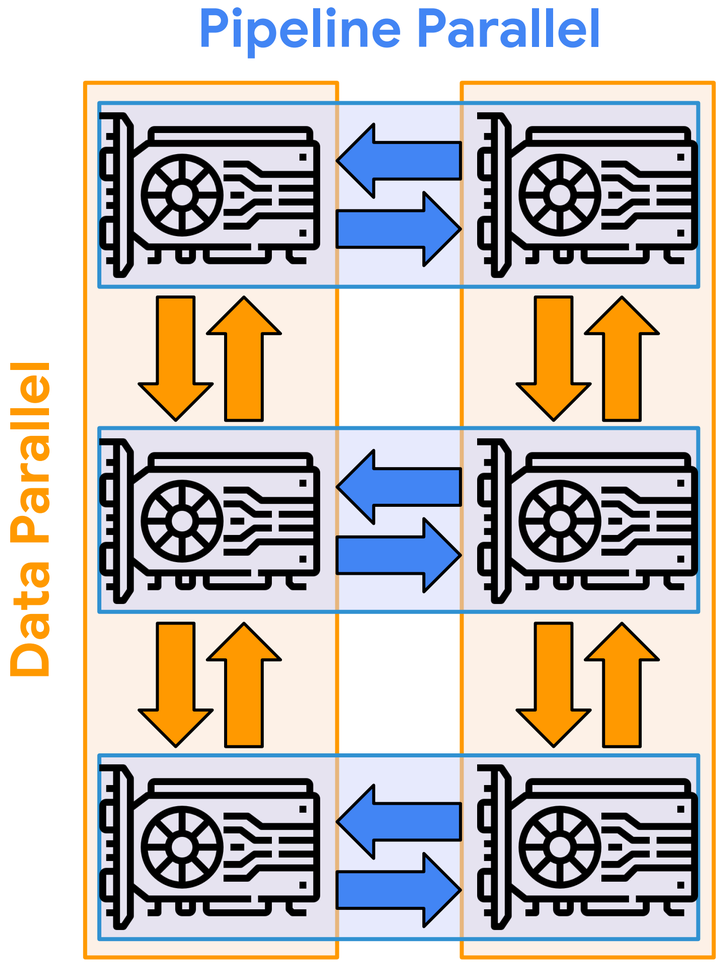


Figure credit: [Qi et al., 2023]

2D Parallelism

- Parallelism strategies complement each other well
  ⇒ combine over device mesh
- Use pipeline parallel over fast connections
  - Allows for larger models
- Use data parallel over slower connections
  - Allows for larger batch size



Pipeline Parallelism

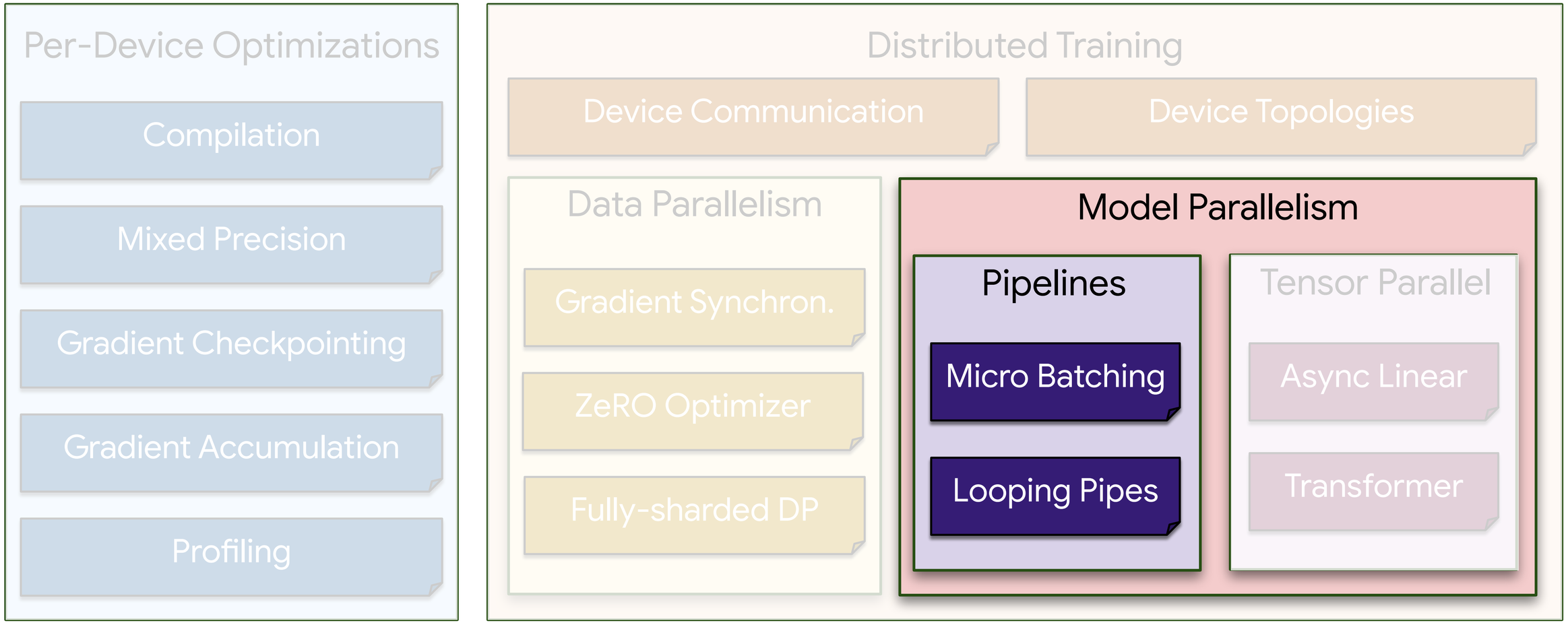
Benefits

- Scales to extremely large models
- Only few communications per execution needed
- Can be generally adapted to any DL model

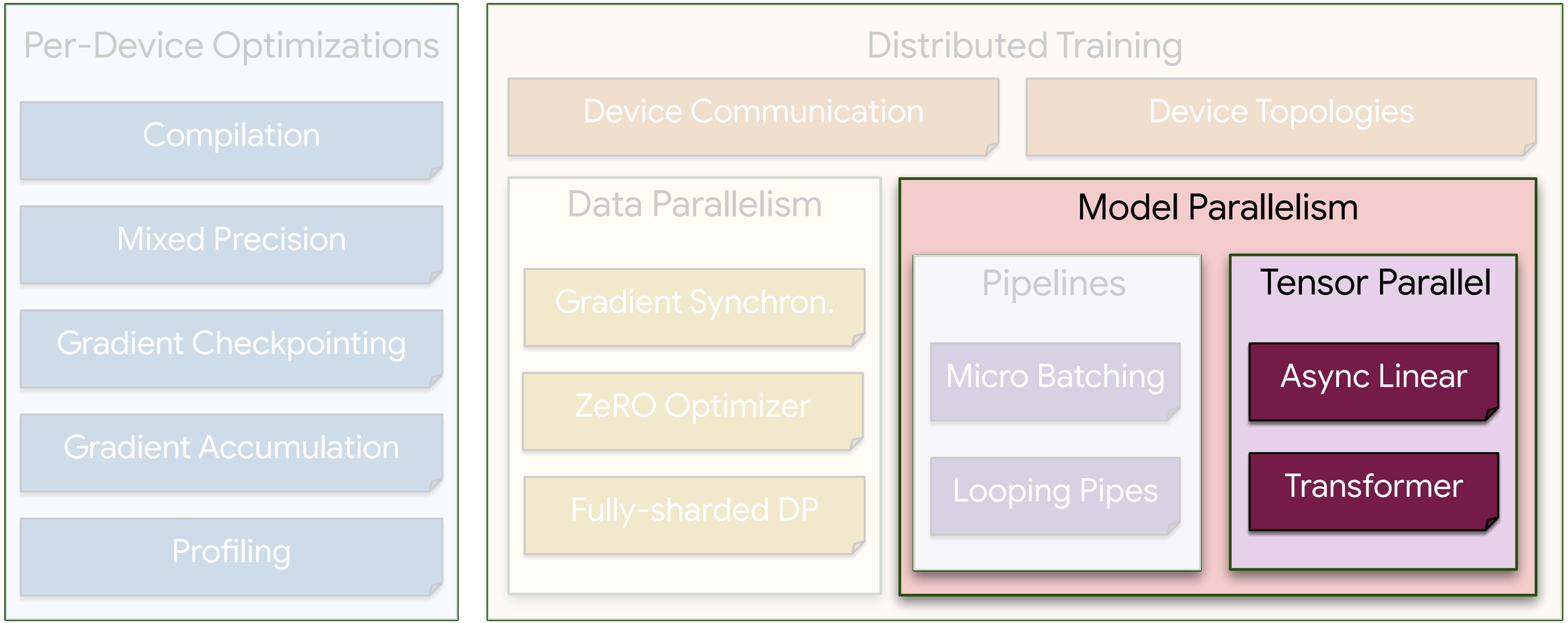
Drawbacks

- Pipeline bubble reduces efficiency
- Requires many micro-batches
- Communication non-async
- Complex handcrafted schedules

Overview

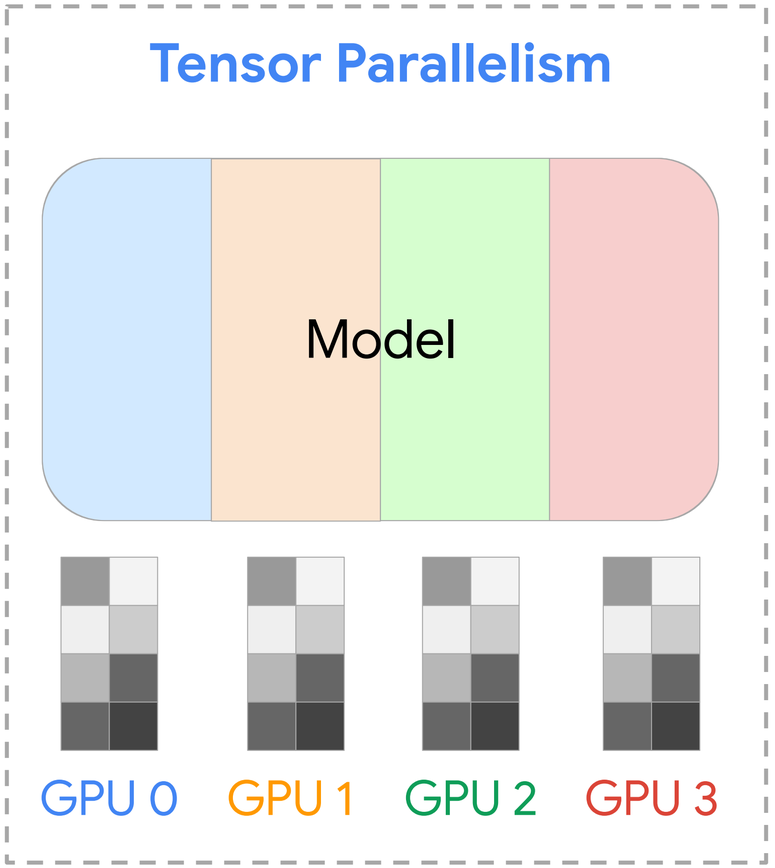

Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

Overview


Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

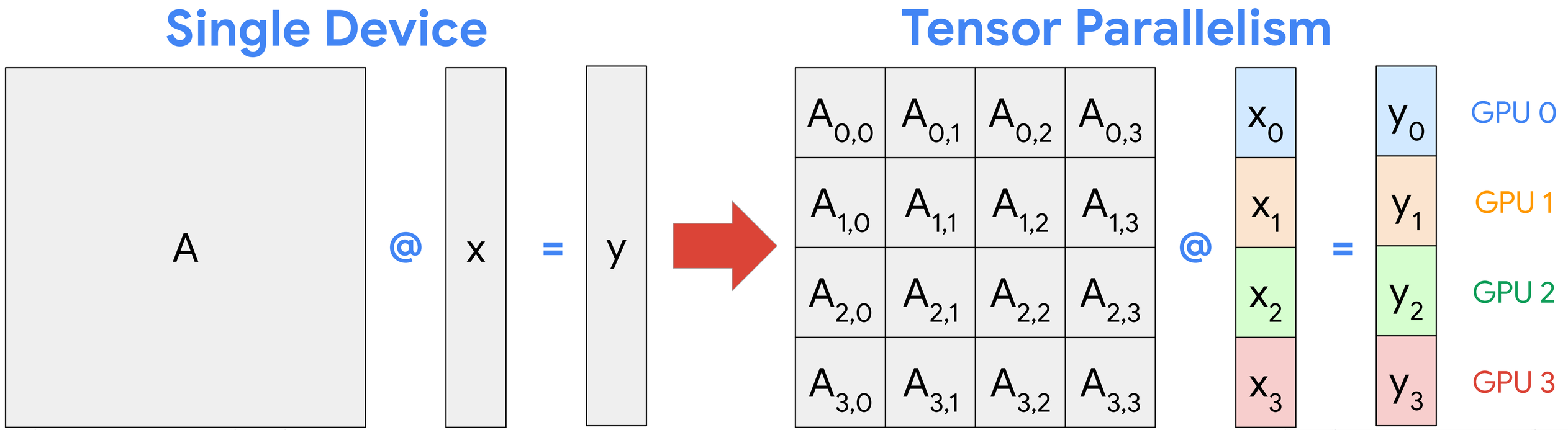
Tensor Parallelism

- **Tensor** parallelism distributes over **features**
- Each layer will be distributed over multiple devices
- Key technology behind ViT-22b, Megatron-LM, OPT-175b, and more transformer-based models



Tensor Parallelism

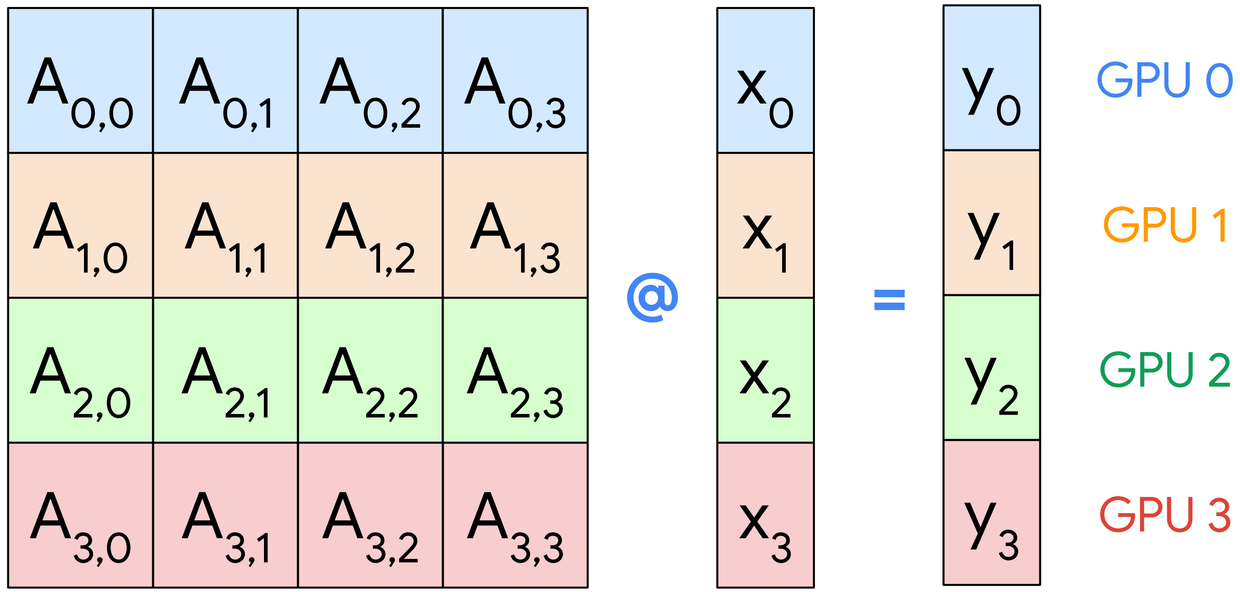
Linear Layer



Tensor Parallelism

Linear Layer - Gather

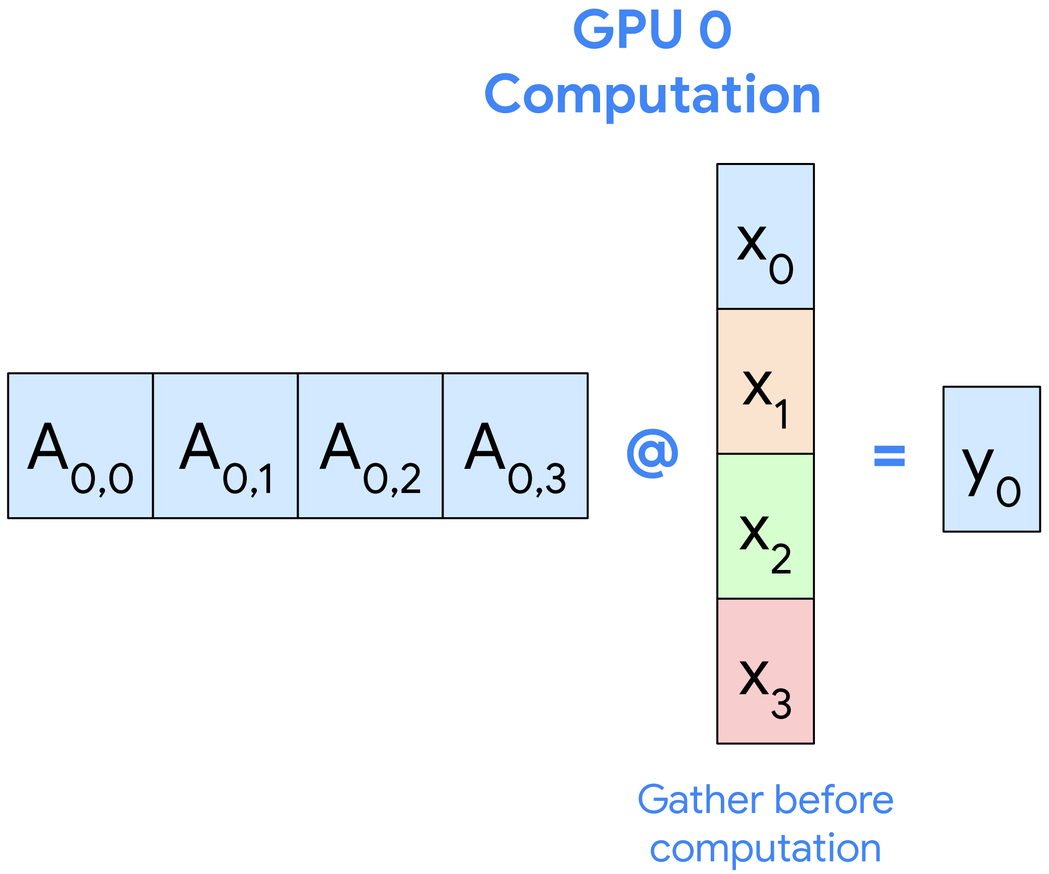
- First, each device **gathers** the features from all devices
- Matmul with device-own parameters gives device-own output



Tensor Parallelism

Linear Layer - Gather

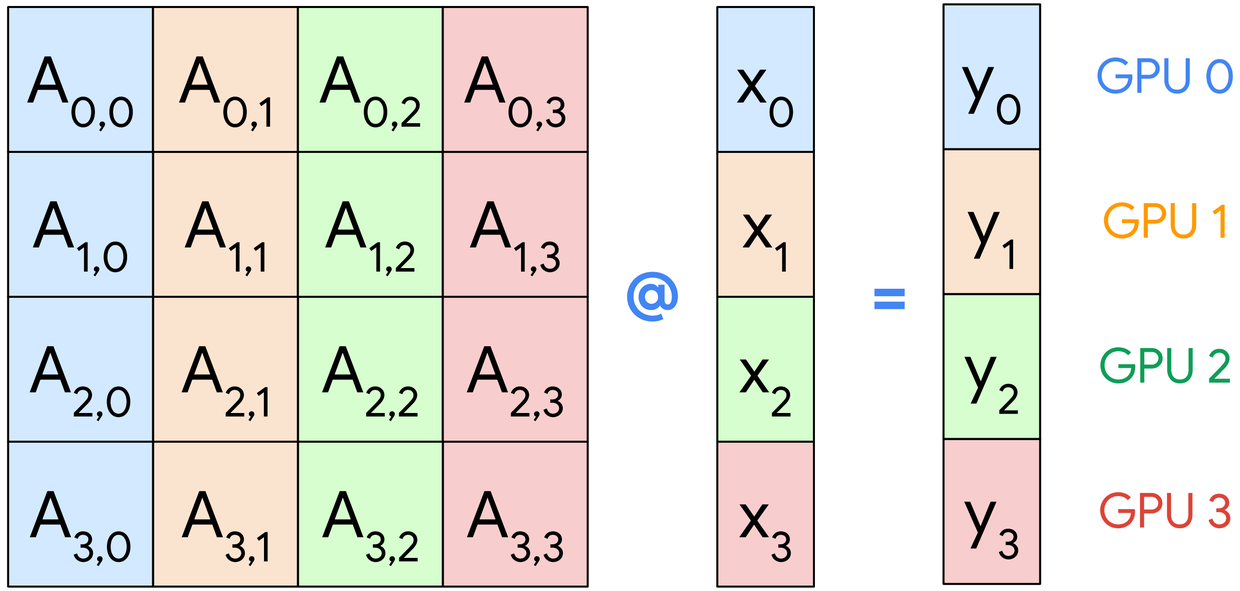
- First, each device **gathers** the features from all devices
- Matmul with device-own parameters gives device-own output



Tensor Parallelism

Linear Layer - Scatter

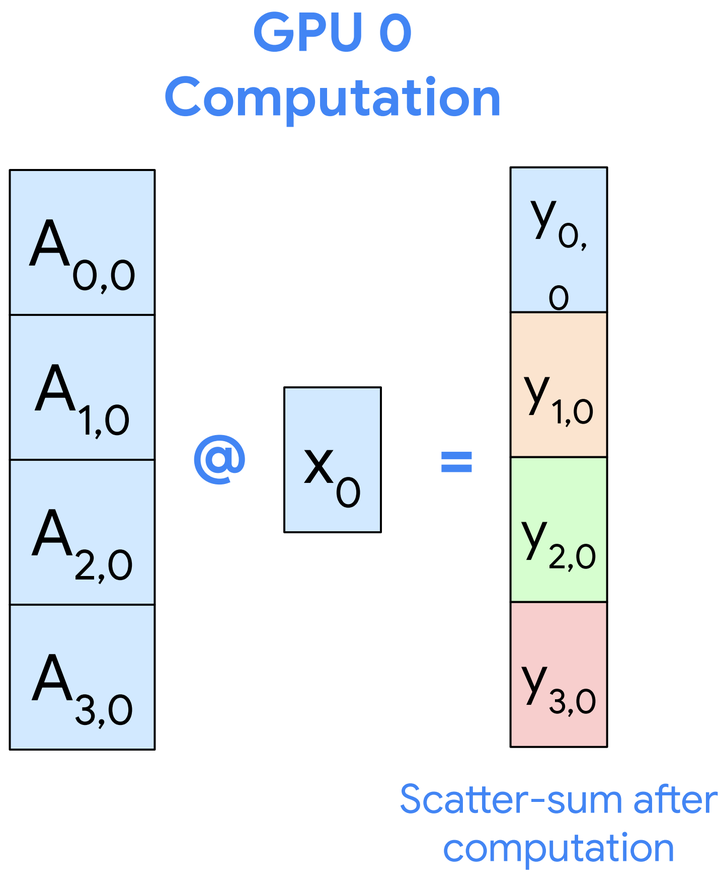
- Matmul of device-own features with device-own parameters give
- Afterwards, **scatter-sum** results to obtain device-own output
- $y_0 = A_{0,0}@x_0 + A_{0,1}@x_1 + A_{0,2}@x_2 + A_{0,3}@x_3$



Tensor Parallelism

Linear Layer - Scatter

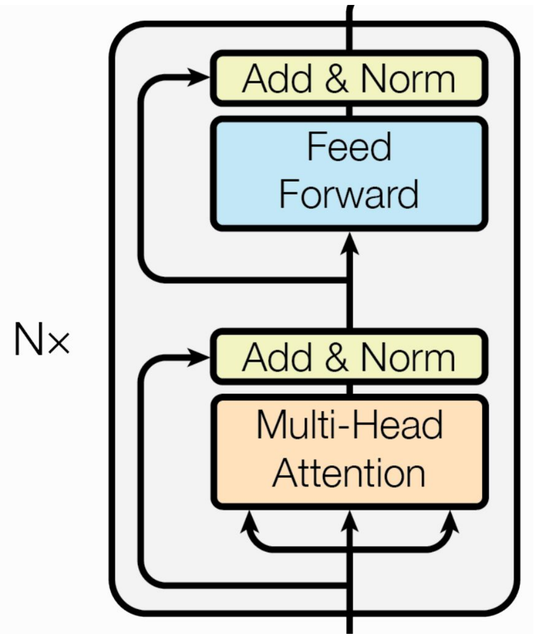
- Matmul of device-own features with device-own parameters give
- Afterward, **scatter-sum** results to obtain device-own output
- $y_0 = y_{0,0} + y_{0,1} + y_{0,2} + y_{0,3}$



Tensor Parallelism

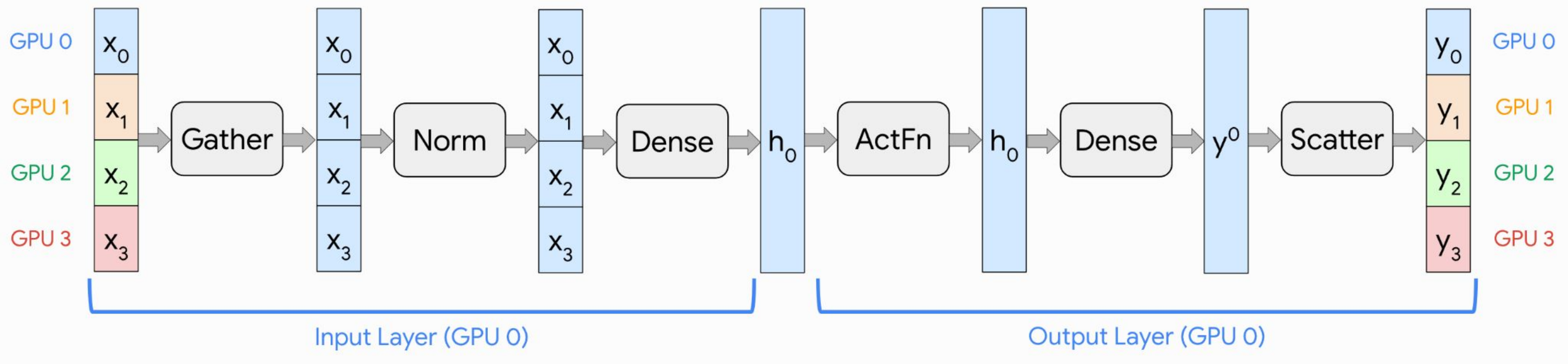
MLP in Transformers

- Select version that communicates the **smallest arrays**
- Example: MLP Block in Transformer
  - Hidden size scales by 4-16x
  - Gather on input layer, scatter on output layer



Tensor Parallelism

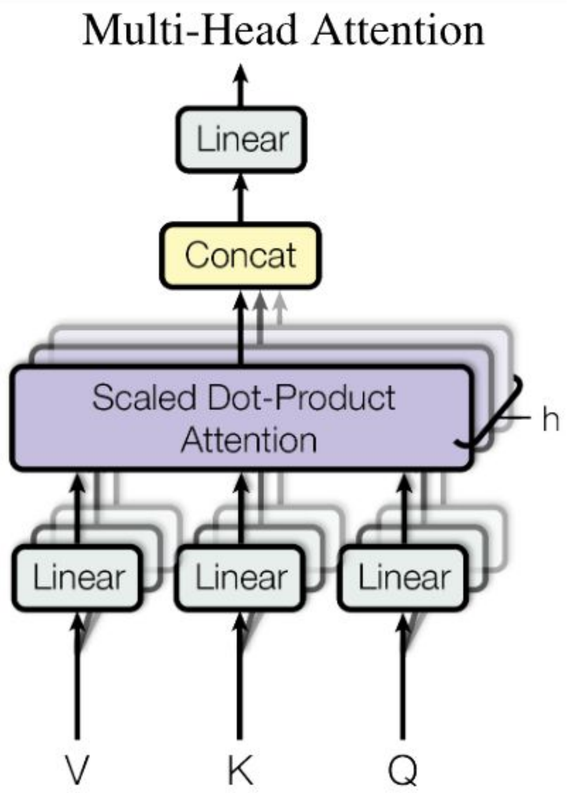
MLP in Transformers



Tensor Parallelism

MHA in Transformers

- Distribute over heads
  a. Gather inputs
  b. Calculate Q, K, V per device-own heads
  c. Calculate self-attention output of device-own heads
  d. Gather/scatter output linear



Tensor Parallelism

Communication Reduction

- Big limitation so far: communication is blocking
  - Devices are idle while waiting for communication
- Reduce by parallelizing Attn and MLP
  - For deep models, similar performance
- Next: Asynchronous Communication

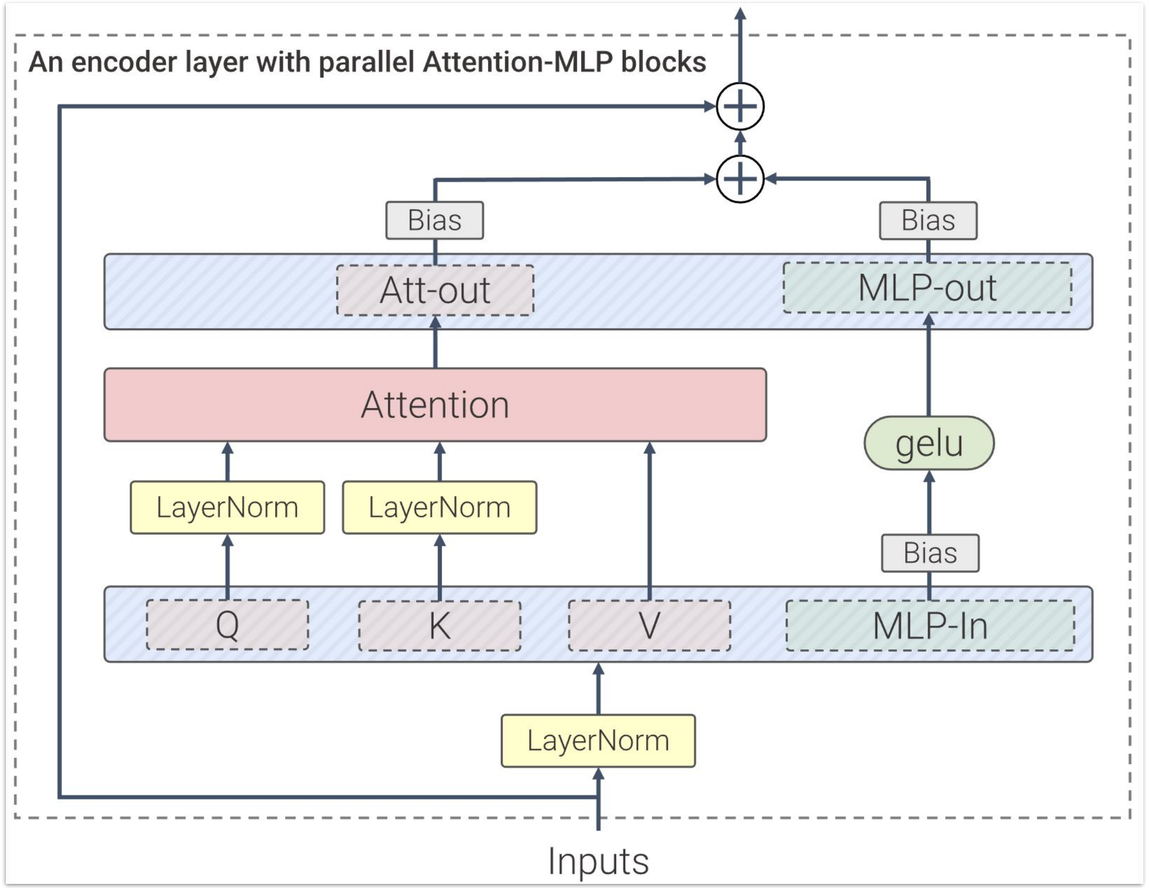


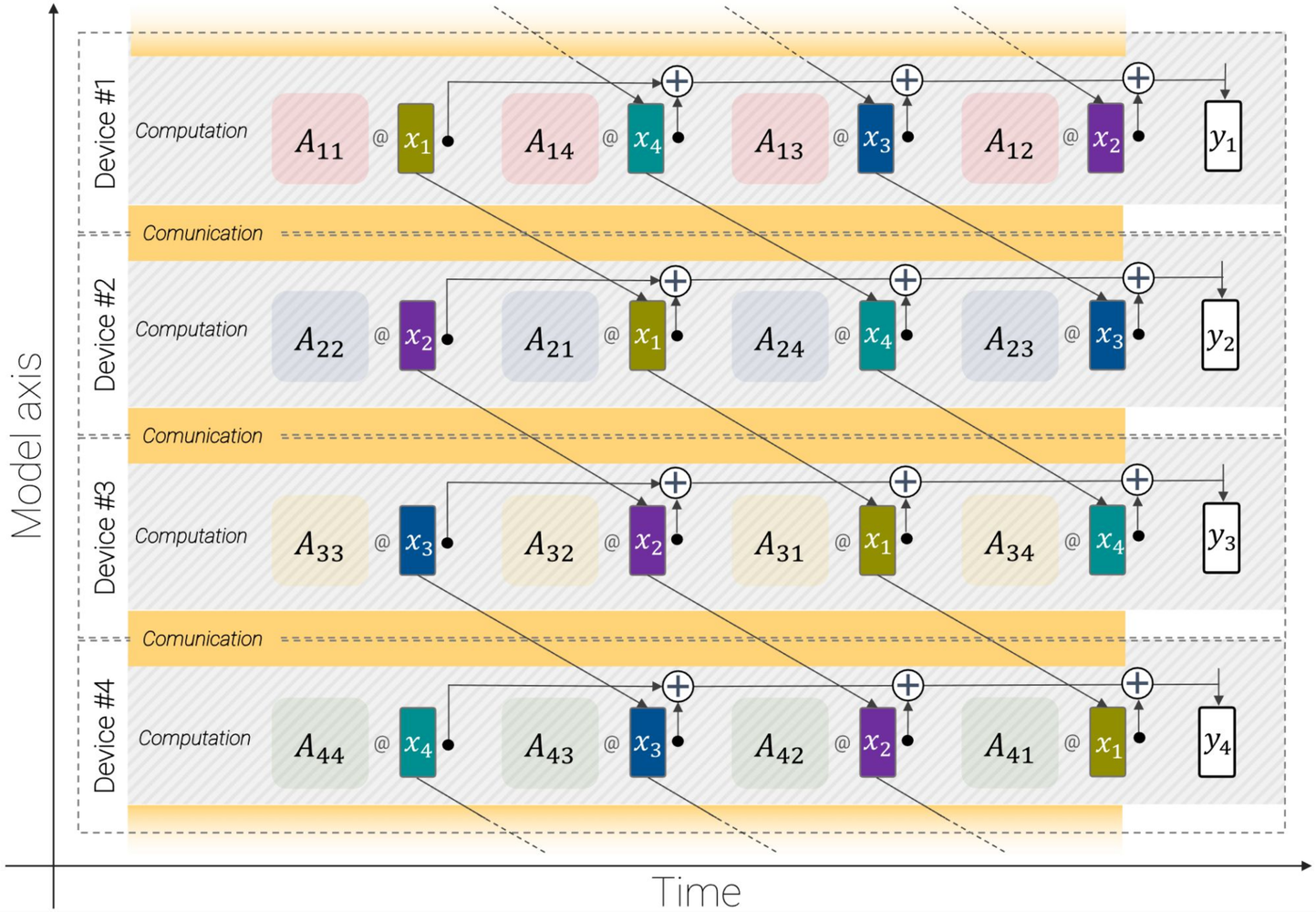
Figure credit: Dehghani et al., 2023

Tensor Parallelism

Asynchronous Layers - Gather

Figure credit: Dehghani et al., 2023

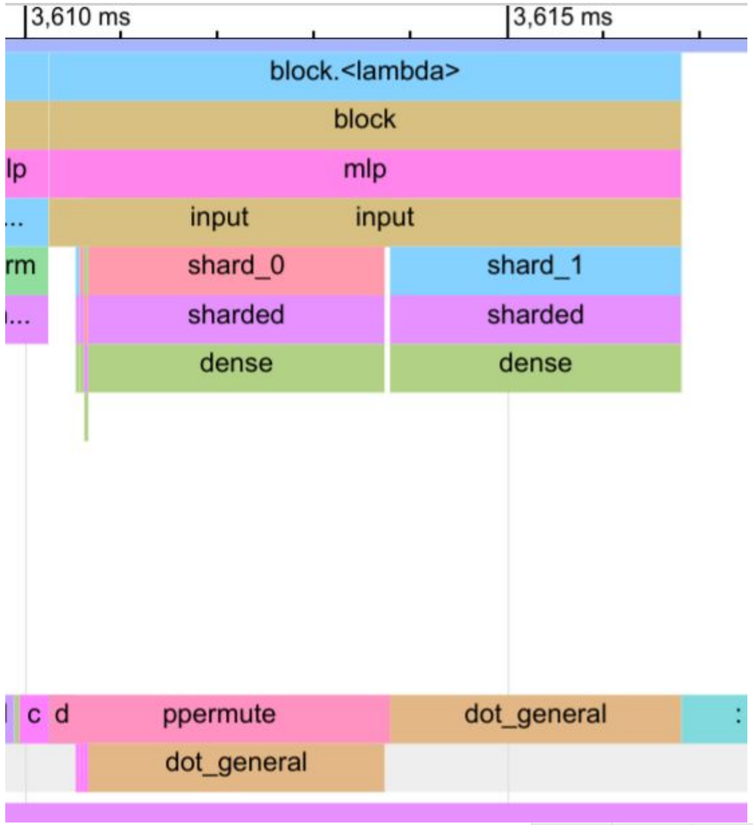
- **Communicate** next feats **while computing** current
- Uses ppermute for fast, ring-based communication
- In ideal case, completely removes communication bottleneck



Tensor Parallelism

Asynchronous Layers - Gather

- Asynchronous communication can be well observed in profile
- 1B LLM trained on 2 A5000 with NVLink 4 (60GB/s)
- Communication blocking negligible

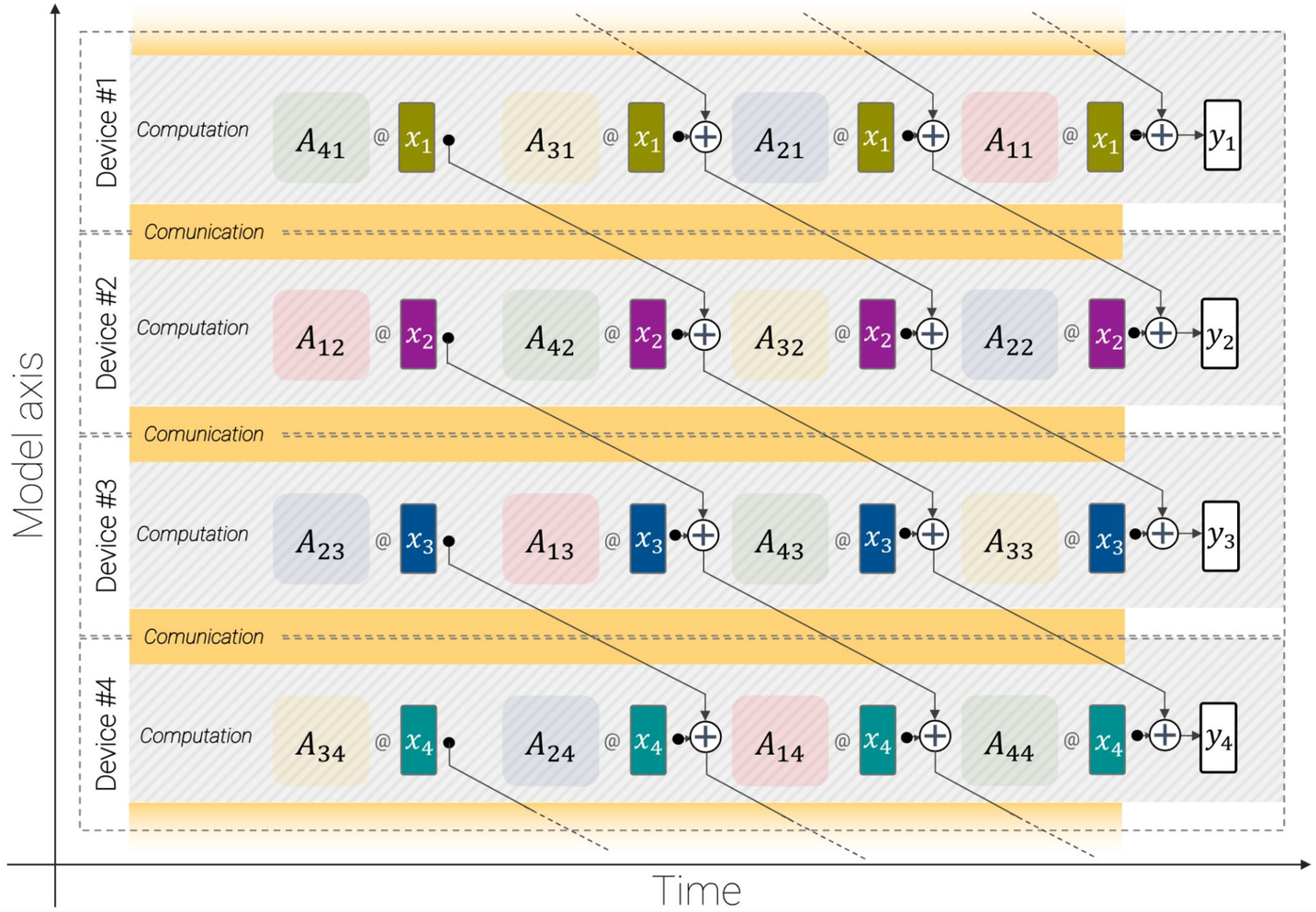


Tensor Parallelism

Asynchronous Layers - Scatter

- First calculate outputs that communicate furthest
- Last compute matmul for device-own output

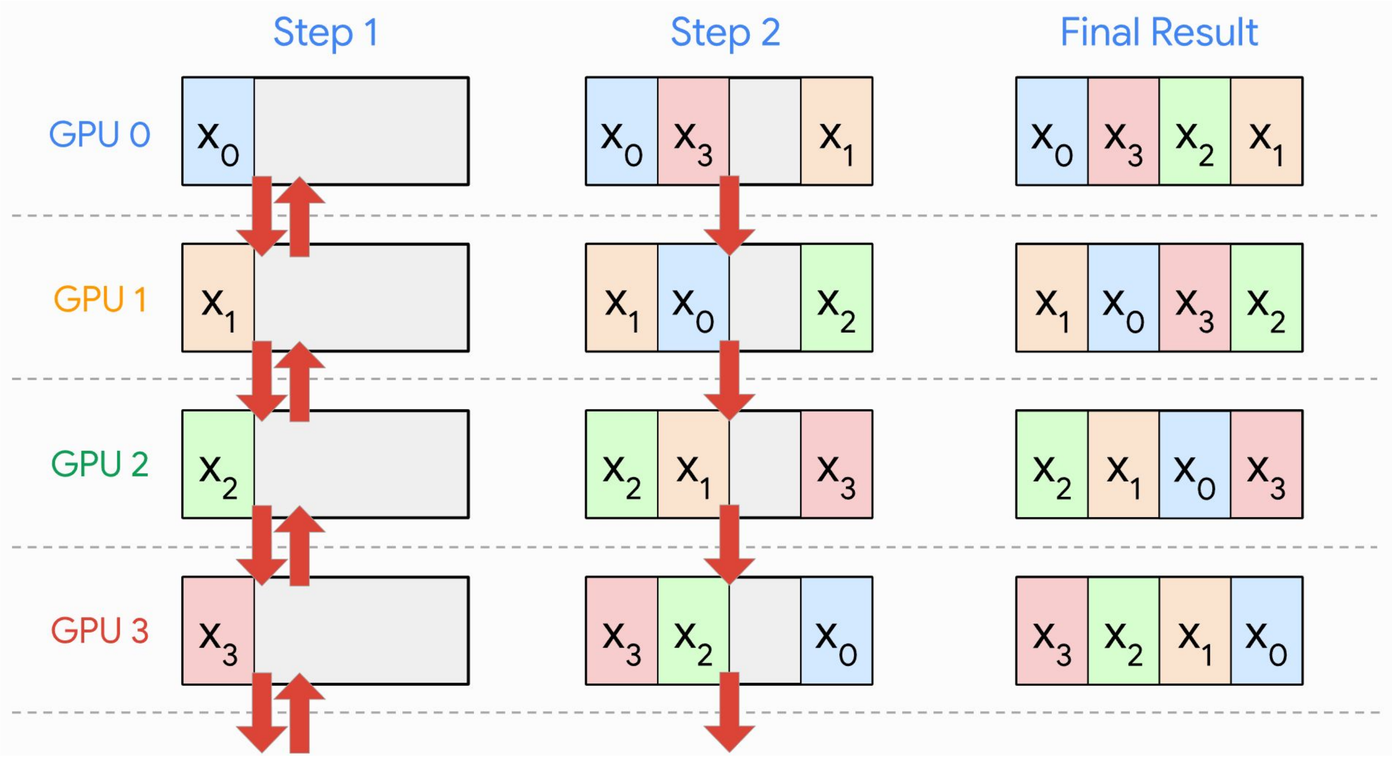
Figure credit: Dehghani et al., 2023



Tensor Parallelism

Asynchronous Layers - Bidirectional

- Device connections are often bidirectional (NVLink, TPUs)
- Can exploit full bandwidth by performing bidirectional communication
- Alternative: split features and comm. half in each direction



Tensor Parallelism

Benefits

- Scales to extremely large models
- Achieves maximum performance with async communication
- Fits well in Single-Program Multiple-Data (SPMD) regime
- Fits well with Transformers

Drawbacks

- Requires high-bandwidth inter-device communications
- Layer-/Model-specific optimization
- More tricky to get model back on single device

3D Parallelism

- Combine all strategies for trillion-parameter scale
- Grouped on bandwidth:
  - Tensor (fastest)
  - Pipeline (intermediate)
  - Data / ZeRO (slow)

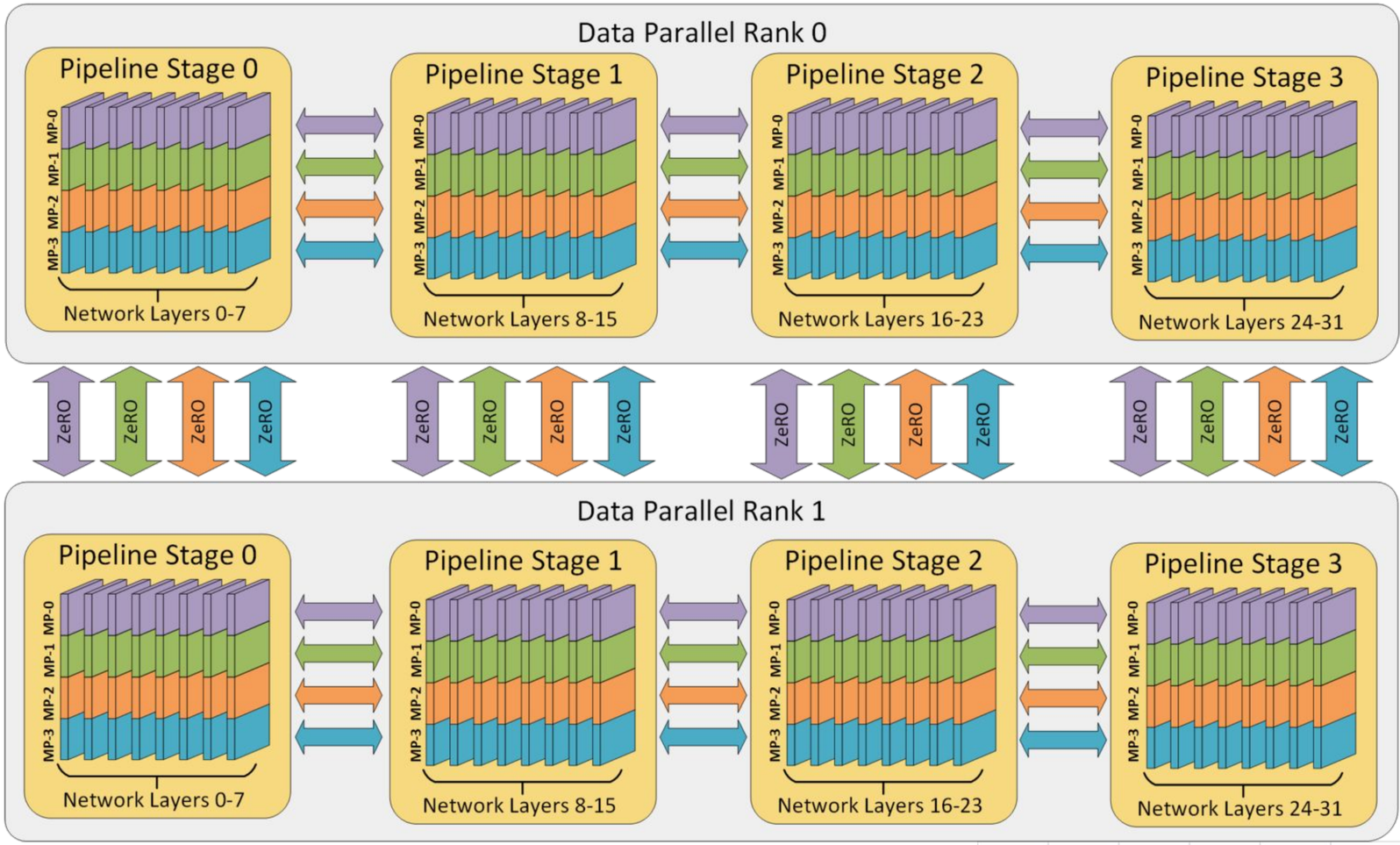


Figure credit: DeepSpeed

3D Parallelism - Nemotron

- Recent NVIDIA Nemotron 340B model trained on 768x8 H100s
  - TP axis size: 8
  - PP axis size: 12
  - DP axis size: 64
    - Batch size 36 per DP element

Figure credit: von Werra

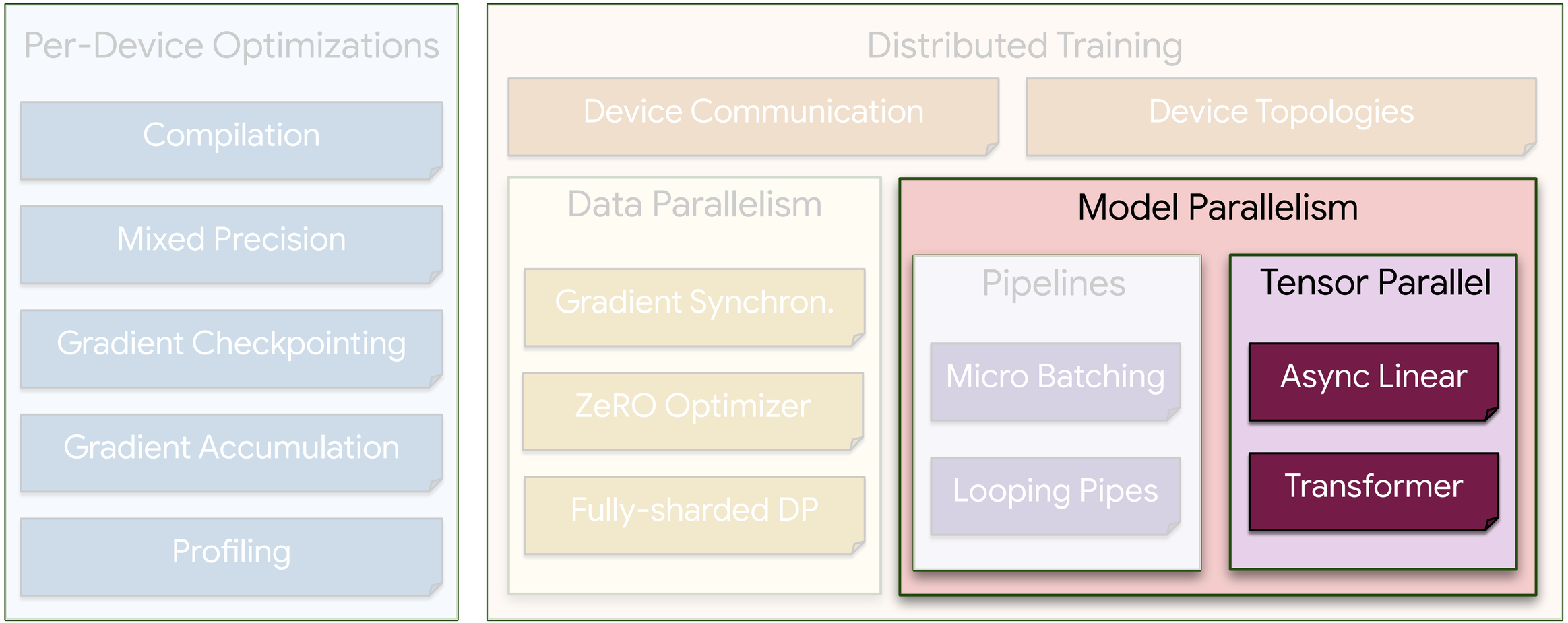
3D Parallelism - Nemotron

- Iteration time multiple seconds
- MFU - Model FLOP/s Utilization

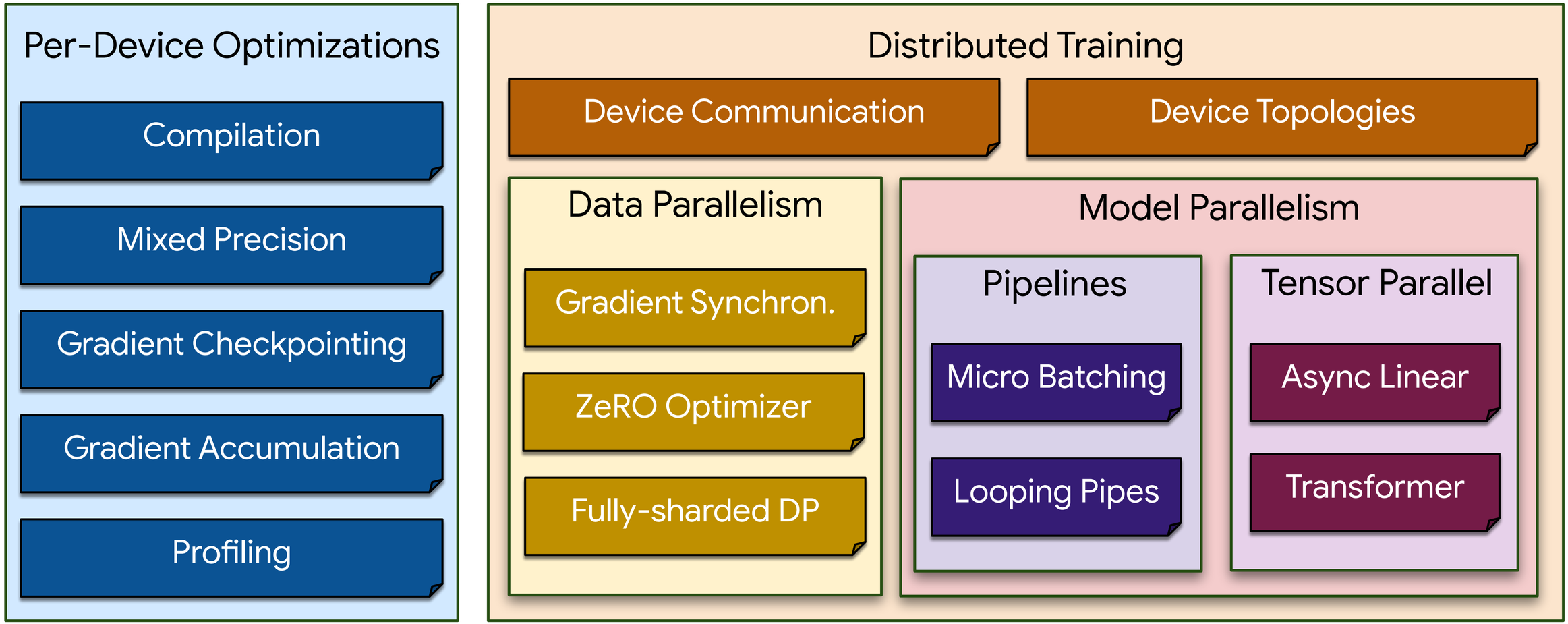
Data-parallel size	GPUs	Iteration time (secs)	MFU (%)	Batch size	Tokens (B)
16	1536	10.3	42.4%	768	200
32	3072	10.3	42.3%	1536	200
64	6144	8.0	41.0%	2304	7600

Figure credit: Nemotron Report

Overview


Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

Overview


Per-Device Optimizations
Compilation
Mixed Precision
Gradient Checkpointing
Gradient Accumulation
Profiling
Distributed Training
Device Communication
Device Topologies
Data Parallelism
Gradient Synchron.
ZeRO Optimizer
Fully-sharded DP
Model Parallelism
Pipelines
Micro Batching
Looping Pipes
Tensor Parallel
Async Linear
Transformer

Summary

Single-Device

- Memory-Compute trade-off on single device
- Mixed precision and compilation to speed-up for “free”
- Profile for bottlenecks

Distributed Training

- Overlapping communication with computation is the key for efficiency
- Small scale: (Fully-Sharded) Data Parallel
- Large scale: Model + Data Parallel
  - Pipeline for intermediate bandwidths
  - Tensor (async) for fast bandwidths
- Tools: DeepSpeed, MaxText
  - Compiler can do a lot of the work

More details:
UvA-DL Notebooks

What we didn’t get to cover

- Sequence parallelism
- Mixture-of-Experts (MoE)
- Implementing own kernel optimizations
- Low-precision optimizers
- Fine-tuning techniques
- Hyperparameter search at scale
- Data loading at scale
- Checkpointing
- Challenges of thousands of GPUs
- And much more...

References and Resources

Per-Device Optimization

[Chen et al., 2016] Chen, T., Xu, B., Zhang, C. and Guestrin, C., 2016. Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174. Paper link

[Micikevicius et a., 2018] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G. and Wu, H., 2018, February. Mixed Precision Training. In International Conference on Learning Representations. Paper link

[Bulatov, 2018] Bulatov, Y., 2018. Fitting larger networks into memory. Blog post link

[Kalamkar et al., 2019] Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D.T., Jammalamadaka, N., Huang, J., Yuen, H. and Yang, J., 2019. A study of BFLOAT16 for deep learning training. arXiv preprint arXiv:1905.12322. Paper link

[Ahmed et al., 2022] Ahmed, S., Sarofeen, C., Ruberry, M., et al., 2022. What Every User Should Know About Mixed Precision Training in PyTorch. Tutorial link

[Weng et al., 2022] Weng, L., Brockman, G., 2022. Techniques for training large neural networks. Blog link

[Raschka, 2023] Raschka, S., 2023. Optimizing Memory Usage for Training LLMs and Vision Transformers in PyTorch. Tutorial link (gives more details for the topics here in PyTorch)

[HuggingFace, 2024] HuggingFace, 2024. Performance and Scalability: How To Fit a Bigger Model and Train It Faster. Tutorial link

[NVIDIA, 2024] NVIDIA, 2024. Mixed Precision Training. Documentation link

[NVIDIA, 2024] NVIDIA, 2024. Performance Guide for Training. Documentation link

[Google, 2024] JAX Team Google, 2024. Control autodiff's saved values with jax.checkpoint (aka jax.remat). Tutorial link

[Google, 2024] JAX Team Google, 2024. Profiling JAX programs. Tutorial link

[Google, 2024] JAX Team Google, 2024. GPU peformance tips. Tutorial link

References and Resources

Data Parallelism

[Rajbhandari et al., 2020] Rajbhandari, S., Rasley, J., Ruwase, O. and He, Y., 2020. Zero: Memory optimizations toward training trillion parameter models. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-16). Paper link

[Wang and Komatsuzaki, 2021] Wang, B., and Komatsuzaki, A., 2021. Mesh transformer jax. GitHub link

[Beyer et al., 2022] Beyer, L., Zhai, X., and Kolesnikov, A., 2022. Big Vision. GitHub link

[PyTorch, 2024] PyTorch Team, 2024. Getting Started with Distributed Data Parallel. Tutorial link

[PyTorch, 2024] PyTorch Team, 2024. What is Distributed Data Parallel. Tutorial link

[Google, 2024] JAX Team Google, 2024. Distributed arrays and automatic parallelization. Notebook link

[Google, 2024] JAX Team Google, 2024. SPMD multi-device parallelism with shard_map. Notebook link

[Google, 2024] JAX Team Google, 2024. Using JAX in multi-host and multi-process environments. Notebook link

[DeepSpeed, 2024] DeepSpeed, 2024. Zero Redundancy Optimizer. Tutorial link

References and Resources

Pipeline Parallelism

[Huang et al., 2019] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q.V. and Wu, Y., 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. Advances in neural information processing systems, 32. Paper link

[Narayanan et al., 2021] Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B. and Phanishayee, A., 2021, November. Efficient large-scale language model training on gpu clusters using megatron-lm. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-15). Paper link

[Lamy-Poirier, 2023] Lamy-Poirier, J., 2023. Breadth-First Pipeline Parallelism. Proceedings of Machine Learning and Systems, 5. Paper link

[Qi et al., 2023] Qi, P., Wan, X., Huang, G. and Lin, M., 2023. Zero Bubble Pipeline Parallelism. arXiv preprint arXiv:2401.10241. Paper link

[McKinney, 2023] McKinney, A., 2023. A Brief Overview of Parallelism Strategies in Deep Learning. Blog post link

[PyTorch, 2024] PyTorch Team, 2024. Pipeline Parallelism. Documentation link

[Huggingface, 2024] Huggingface, 2024. Model Parallelism. Documentation link

[DeepSpeed, 2024] DeepSpeed, 2024. Pipeline Parallelism. Documentation link

References and Resources

Tensor Parallelism

[Shoeybi et al., 2019] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J. and Catanzaro, B., 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053. Paper link

[Wang and Komatsuzaki, 2021] Wang, B., and Komatsuzaki, A., 2021. Mesh transformer jax. GitHub link

[Xu et al., 2021] Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M. and Pang, R., 2021. GSPMD: general and scalable parallelization for ML computation graphs. arXiv preprint arXiv:2105.04663. Paper link

[Dehghani et al., 2022] Dehghani, M., Gritsenko, A., Arnab, A., Minderer, M. and Tay, Y., 2022. Scenic: A JAX library for computer vision research and beyond. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 21393-21398). Paper link

[Yoo et al., 2022] Yoo, J., Perlin, K., Kamalakara, S.R. and Araújo, J.G., 2022. Scalable training of language models using JAX pjit and TPUv4. arXiv preprint arXiv:2204.06514. Paper link

[Chowdhery et al., 2023] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., Schuh, P., et al., 2023. Palm: Scaling language modeling with pathways. Journal of Machine Learning Research, 24(240), pp.1-113. Paper link

[Anil et al., 2023] Anil, R., Dai, A.M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z. and Chu, E., 2023. Palm 2 technical report. arXiv preprint arXiv:2305.10403. Paper link

[Dehghani et al., 2023] Dehghani, M., Djolonga, J., Mustafa, B., Padlewski, P., Heek, J., Gilmer, J., Steiner, A.P., Caron, M., Geirhos, R., Alabdulmohsin, I., Jenatton, R., et al., 2023. Scaling vision transformers to 22 billion parameters. In International Conference on Machine Learning (pp. 7480-7512). PMLR. Paper link

[McKinney, 2023] McKinney, A., 2023. A Brief Overview of Parallelism Strategies in Deep Learning. Blog post link

[Huggingface, 2024] Huggingface, 2024. Model Parallelism. Documentation link

[Google, 2024] JAX Team Google, 2024. SPMD multi-device parallelism with shard_map. Notebook link

[OpenAI, 2024] OpenAI, 2024. GPT-4. Technical Report

[Google, 2024] Gemini Team Google Deepmind, 2024. Gemini. Technical Report

Experts

Thank you.

For questions, suggestions, etc.,
feel free to reach out.

Phillip Lippe
p.lippe@uva.nl

Slides