ODTÜ METU

TÜBİTAK

EURO CC 4SEE

Optimizing Deep Learning Systems for Hardware

Assoc. Prof. Erdem AKAGÜNDÜZ, METU

ncc@ulakbim.gov.tr

Part IV : System-level Optimization

NCC TÜRKİYE EURO CC 4SEE

- PIV.a : Parallelism
- PIV.b : Mixed-Precision Training
- PIV.c : Other

Part IV.a: Parallelism, Why?

- Large models (e.g., GPT, BERT) don’t fit on a single GPU
- Training takes days or weeks on one machine
- Solution: break up the work across multiple devices.

Types of Parallelism

- We’ll cover four types of parallelism:
  - Data Parallelism
  - Model Parallelism
  - Pipeline Parallelism
  - Tensor (Sharded) Parallelism

Data Parallelism

- Data Parallelism (Concept)
  - Copy the entire model to multiple devices
  - Each device gets a different batch of training data
  - Gradients are averaged & synced across devices
  - no 🧠er:
    - Easy to implement
    - Widely supported

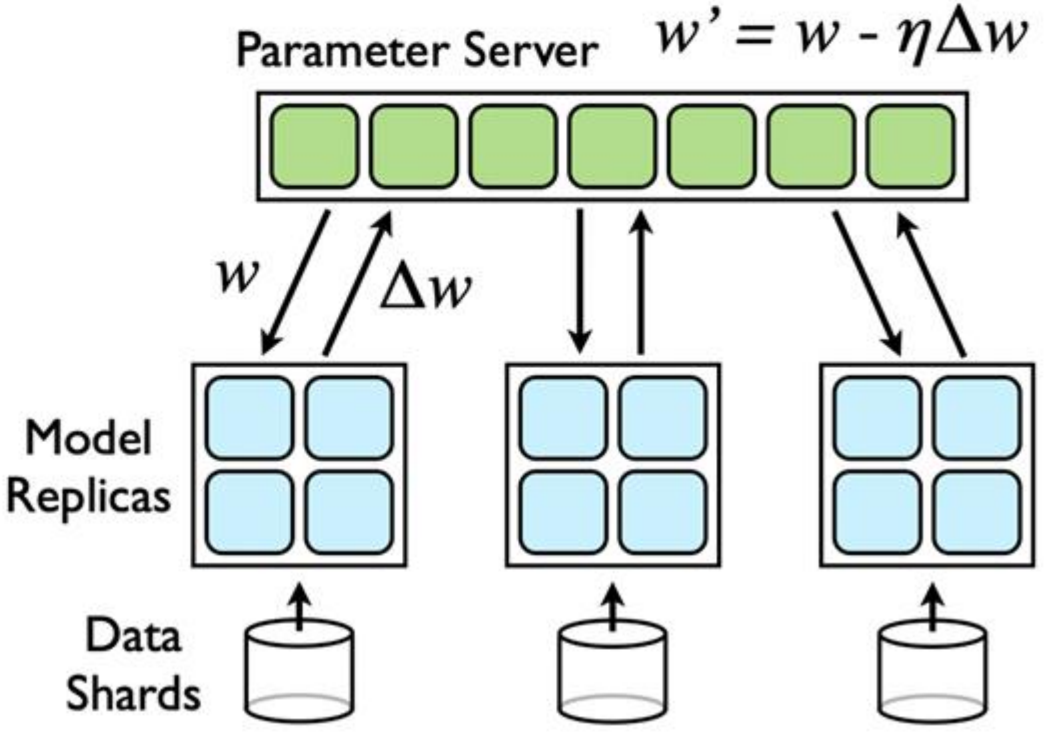


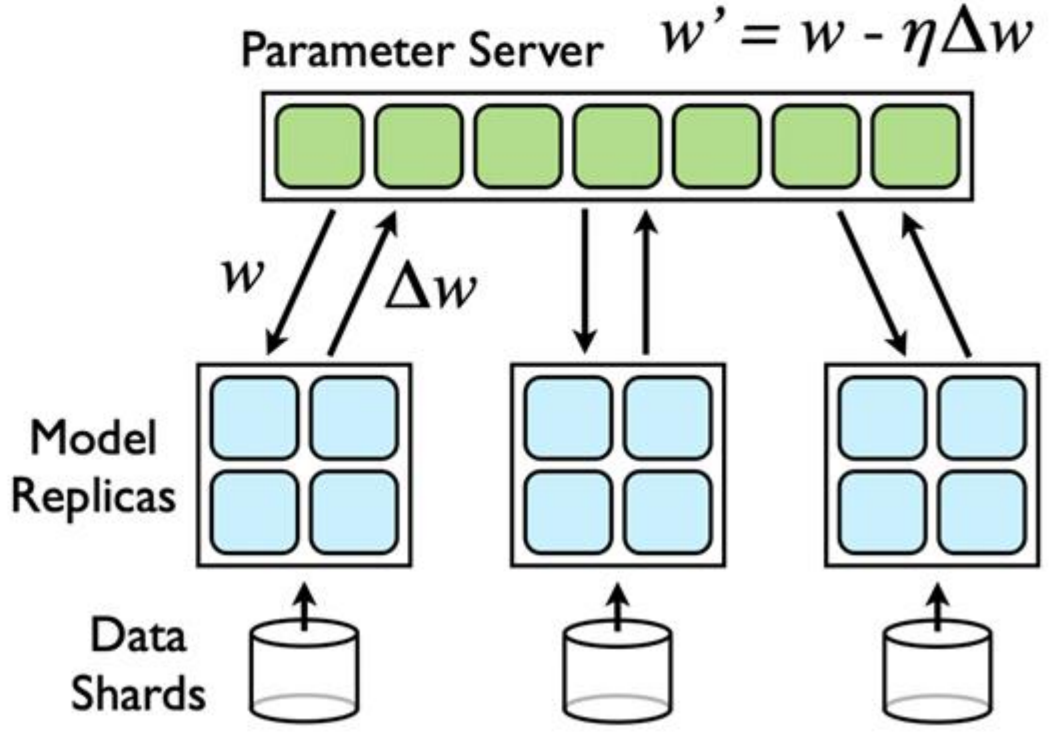
figure: https://insujang.github.io/2022-06-11/parallelism-in-distributed-deep-learning/

Data Parallelism

NCC TÜRKİYE EURO 4SEE

- Data Parallelism (Concept)
  - Copy the entire model to multiple devices
  - Each device gets a different batch of training data
  - Gradients are averaged & synced across devices
  - no 🧠er:
    - Easy to implement
    - Widely supported

Gradient synchronization at each step

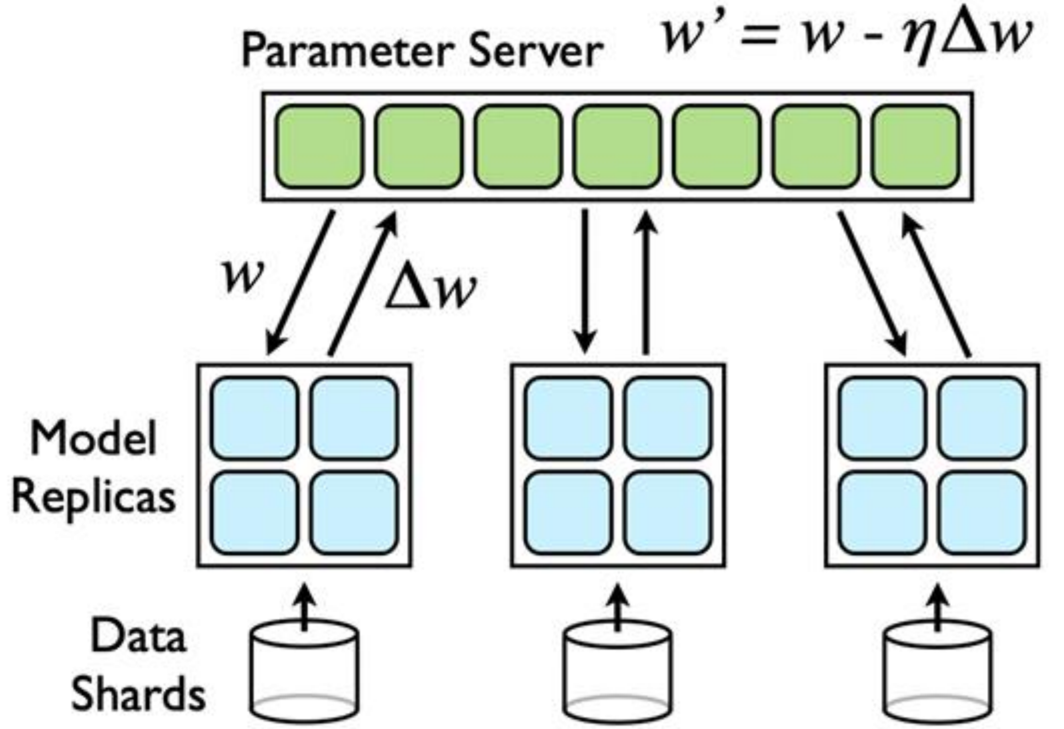


Data Parallelism - Pros & Cons

Pros:
- Simple to implement
- Scales well for many tasks

Cons:
- Doesn't help if model is too big to fit on one GPU
- Communication overhead (the gradient sync)

Gradient synchronization at each step



Model Parallelism

- Model Parallelism
  - Split the model across multiple devices
  - Each device holds a different part of the model
  - Useful when the model is too large for one GPU

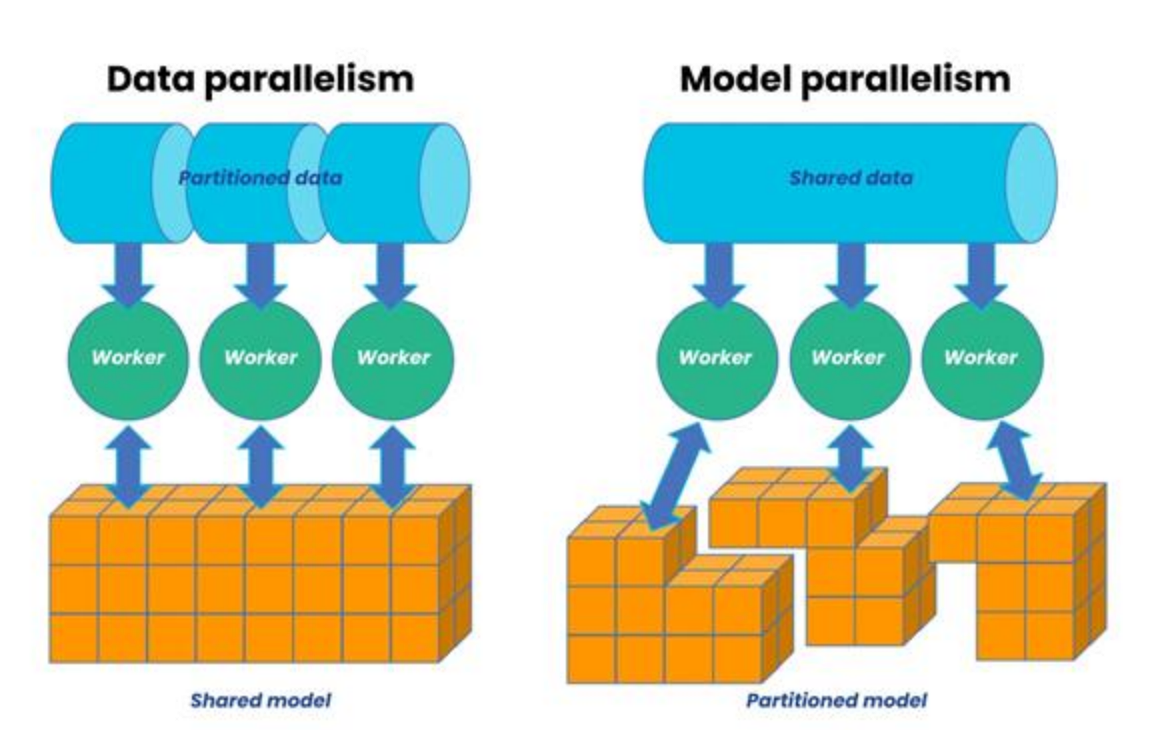


figure: https://www.anyscale.com/blog/what-is-distributed-training

Model Parallelism: Case AlexNet!

- When AlexNet was introduced in 2012, GPUs had much less memory than today. The model was too large to fit on a single GPU at the time, so the authors split the model across two GPUs.
- In AlexNet, each GPU handled a different subset of the convolutional filters (and corresponding feature maps).
- This was implemented using **grouped convolutions**, where each group was assigned to a different GPU.

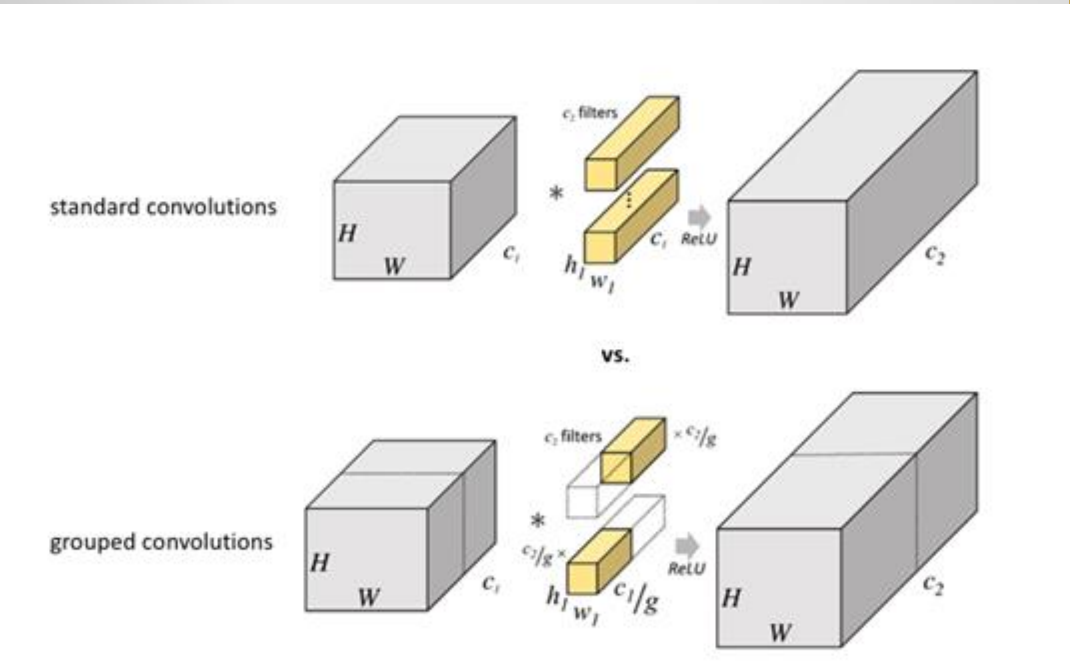


figure: https://www.jeremyjordan.me/convnet-architectures/

Model Parallelism

Pros:
- Enables training very large models

Cons:
- Harder to implement and debug
- Communication between devices can slow things down

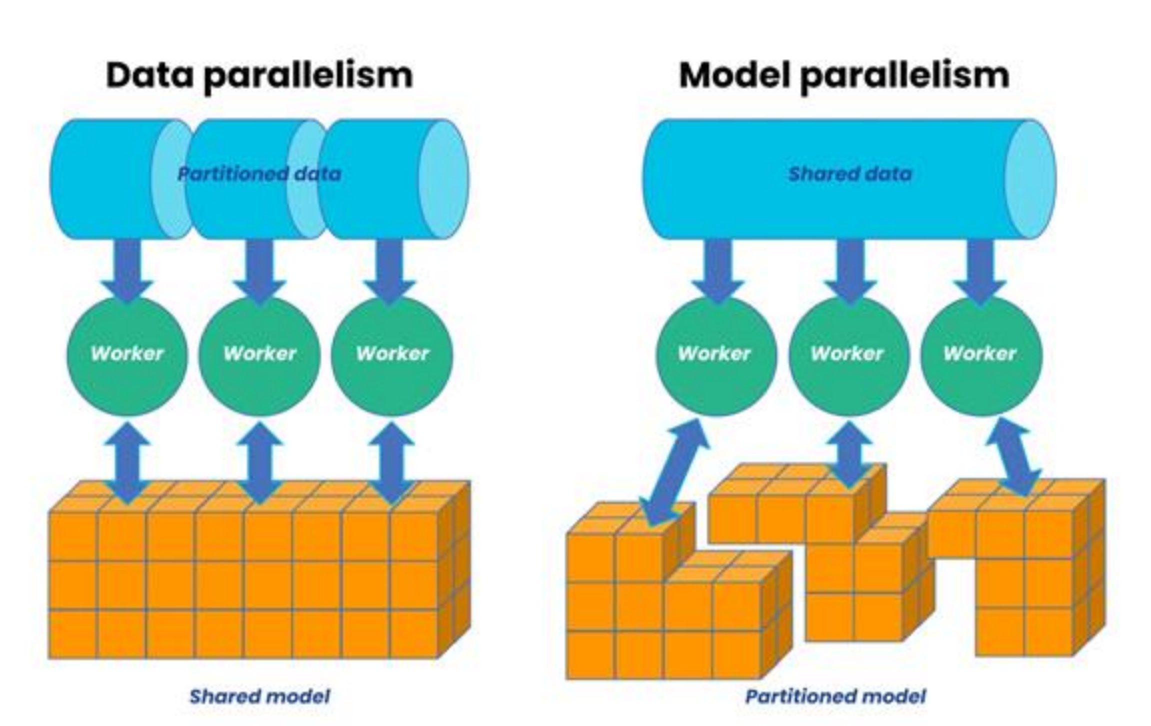


figure: https://www.anyscale.com/blog/what-is-distributed-training

Pipeline Parallelism

- Pipeline Parallelism
  - Break model into “stages” and run mini-batches in a pipeline across them
  - Like an assembly line
  - Helps keep all devices busy (how?)

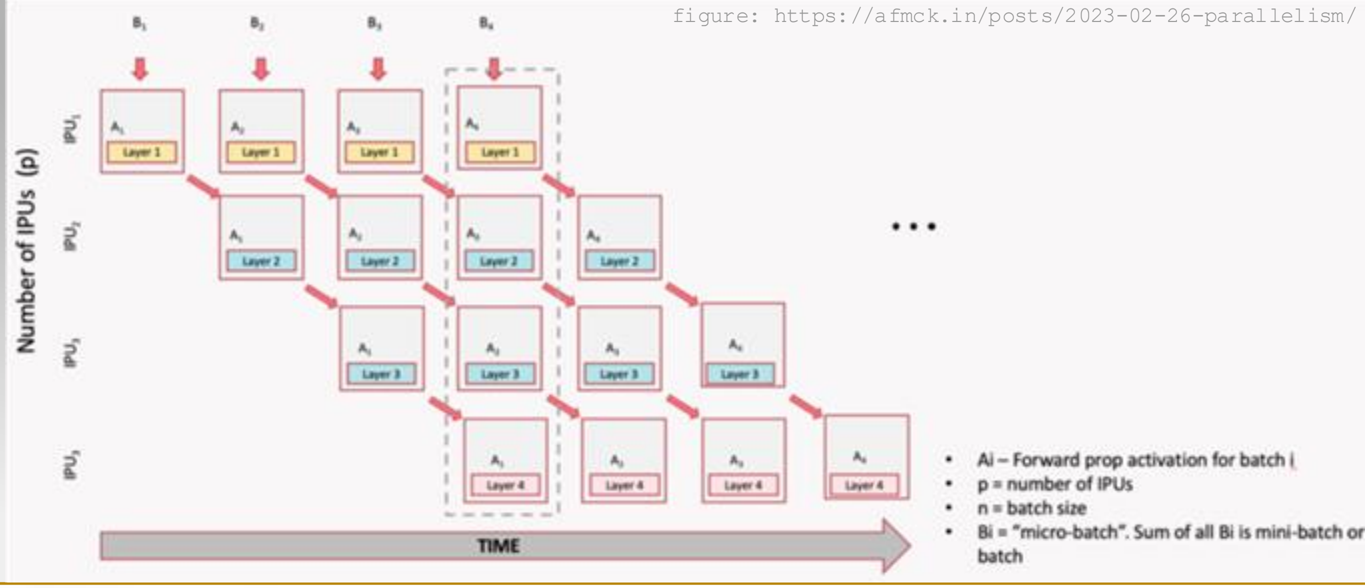
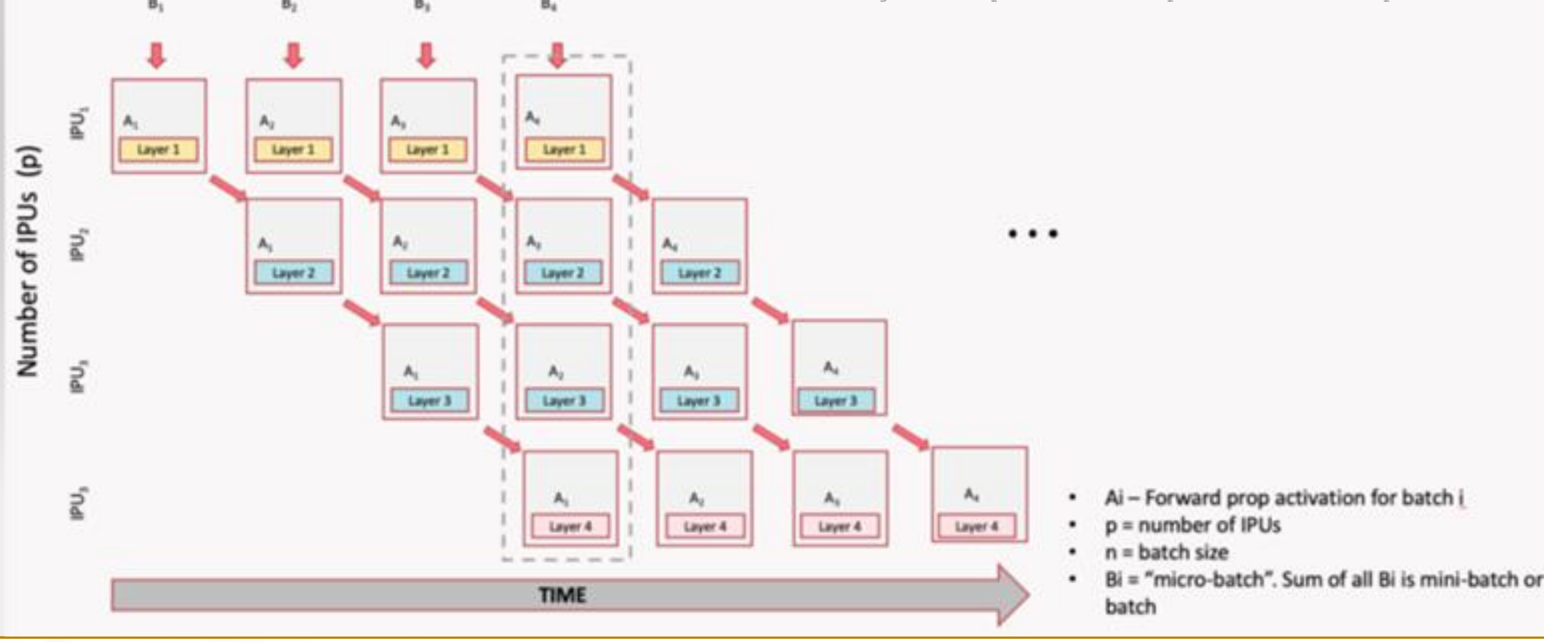


figure: https://afmck.in/posts/2023-02-26-parallelism/

Pipeline Parallelism

- Stage 1 on GPU 1, Stage 2 on GPU 2, etc.
- Micro-batches flow through the pipeline
- Have to make scheduling perfect!
  - i.e. The load at each GPU should take similar amounts of time

figure: https://afmck.in/posts/2023-02-26-parallelism/



Pipeline Parallelism

Pros:

- Higher hardware utilization than basic model parallelism

Cons:

- Requires careful scheduling (e.g., bubble overhead)
- More complex training logic
- Requires a framework! (like DeepSpeed)

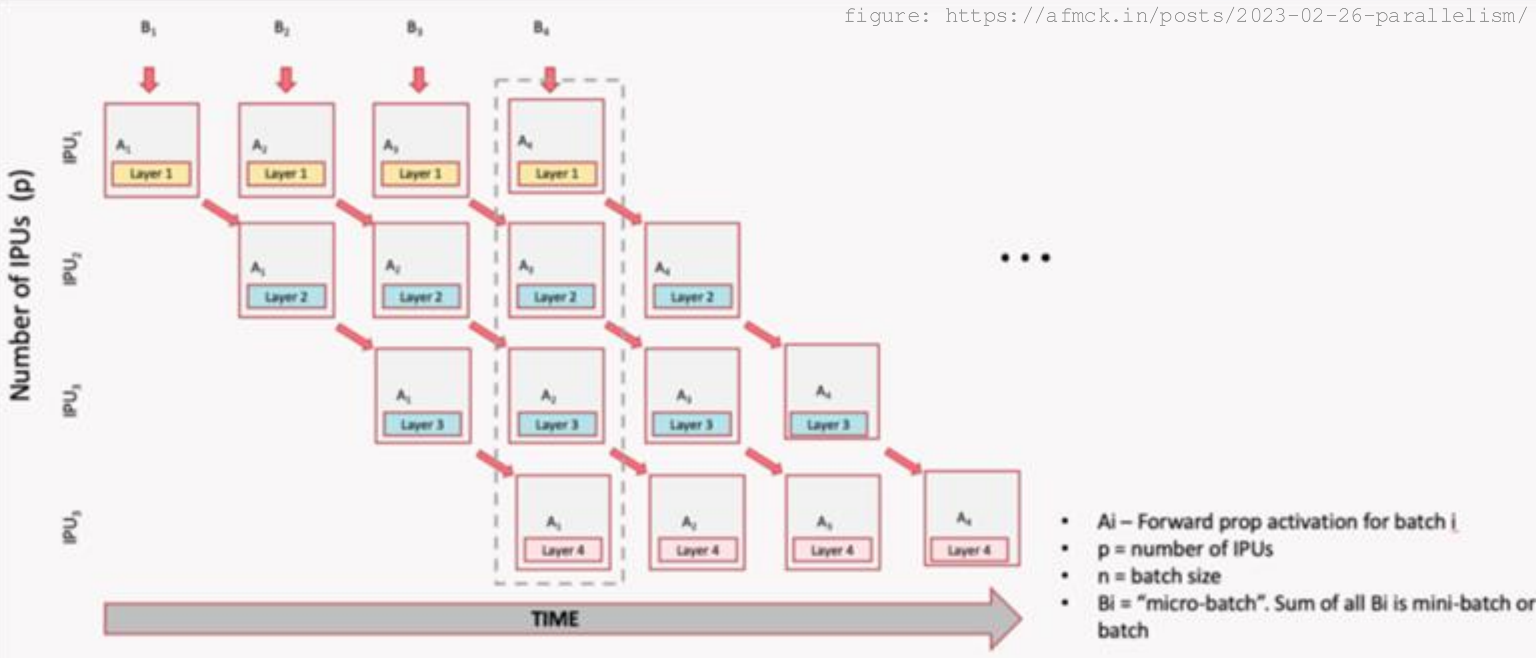


figure: https://afmck.in/posts/2023-02-26-parallelism/

Tensor Parallelism

- Tensor (sharded) Parallelism
  - Split large tensors (e.g., weights) across devices at a fine-grained level
  - e.g., split matrix multiplication across GPUs
  - Common in LLM training frameworks

figure: https://www.anyscale.com/blog/what-is-distributed-training

Tensor (Sharded) Parallelism

TÜRKİYE EURO 4SEE

- Instead of splitting the model by layers (like model parallelism), we split the individual tensors (weights, activations) across devices.
    - Imagine a large matrix too big for one GPU.
    - Slice it into smaller chunks.
    - Each GPU holds and computes only its piece.
    - Together, they perform the full operation.

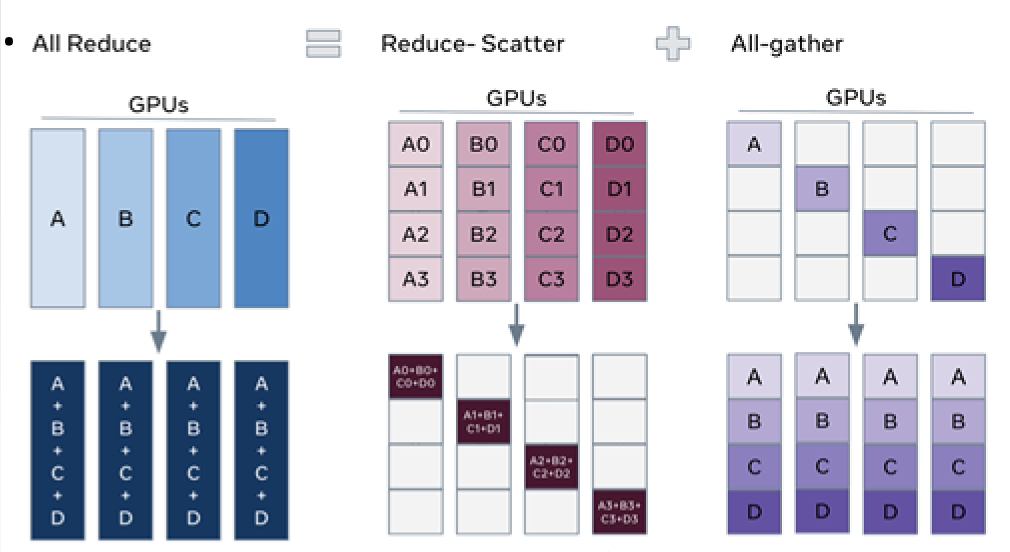


figure: https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Tensor (Sharded) Parallelism

Large tensors are split across multiple GPUs.

- Let's say we have a large weight matrix W in a linear (fully connected) layer:
  - $Y = X \times W^T$
- If W is too big to fit on one GPU, we shard it column-wise:

- GPU 0 holds part A of W
- GPU 1 holds part B …
- GPU 2 holds part C …
- GPU 3 holds part D …

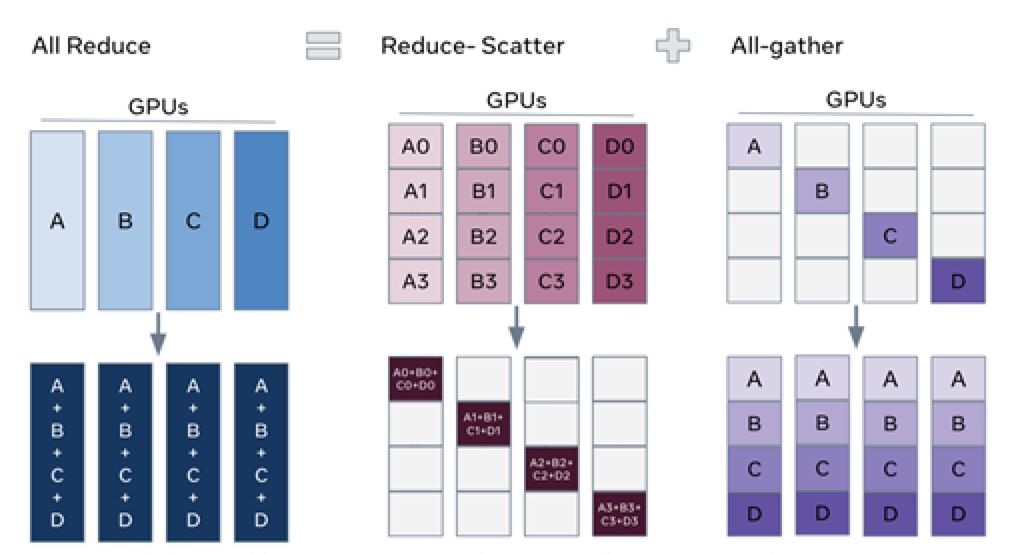


figure: https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Tensor (Sharded) Parallelism

If you sharded W across GPUs:

- Each GPU computes only its local part of dL/dW (say A, B, C, or D)
- To update weights correctly (e.g., using SGD), you may need to sum or sync these parts across all GPUs
- That's why **All-Reduce** is used: to combine these partial gradients into a full gradient.

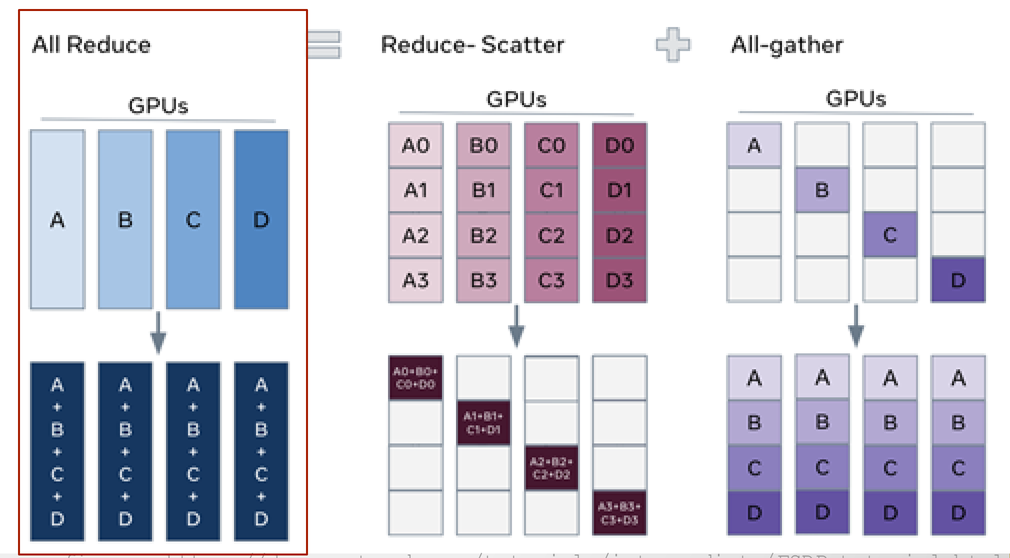


figure: https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Tensor (Sharded) Parallelism

Reduce-Scatter is another communication pattern used in tensor parallelism during distributed training.

- It combines two steps:
  - Reduce: Aggregate data (e.g., sum) across GPUs.
  - Scatter: Distribute chunks of the result to different GPUs.

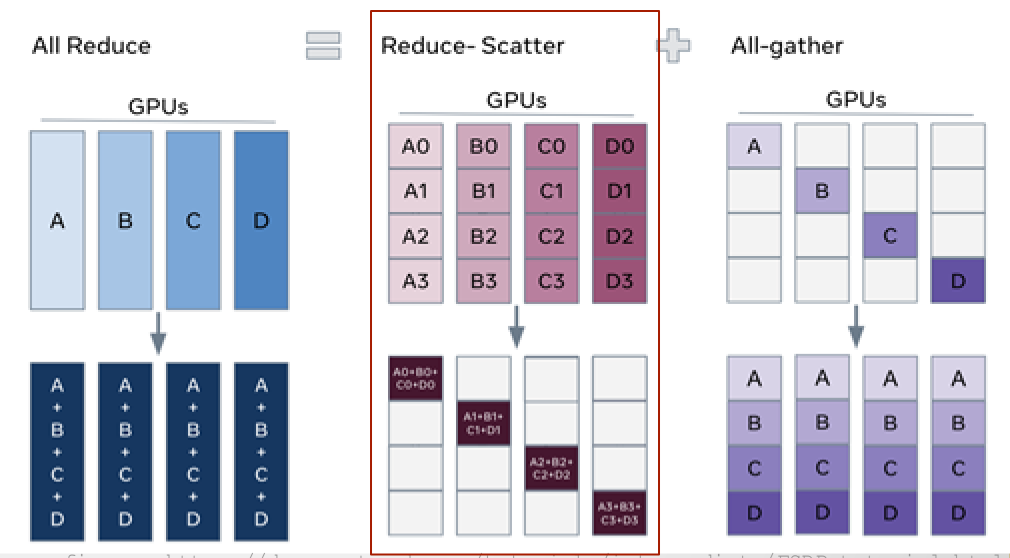


figure: https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Tensor (Sharded) Parallelism

Now each GPU has one reduced piece.

- To get the full result (A+B+C+D), they need to share their pieces.

- In the All-Gather step:
  - GPUs exchange their chunks
  - So that all GPUs end up with every piece

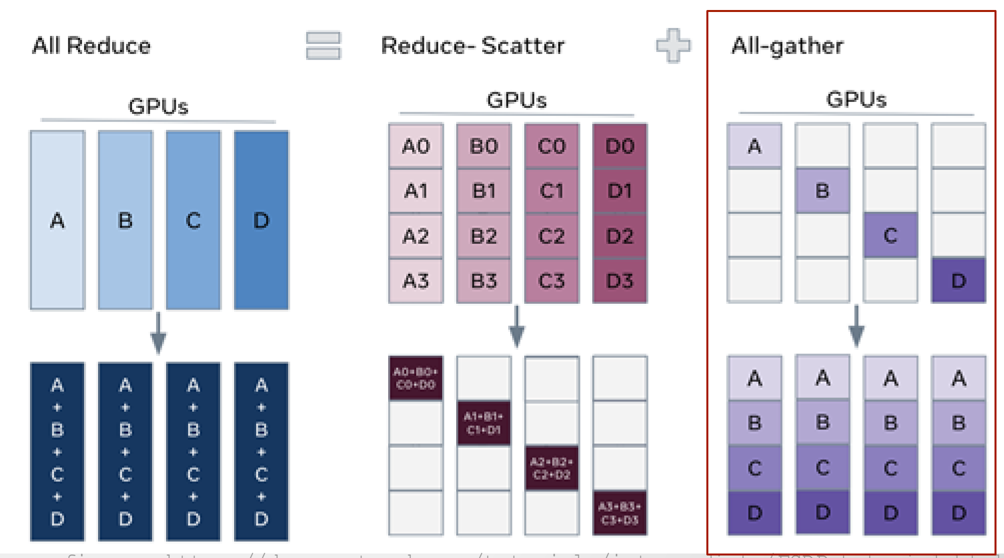


figure: https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Communication Patterns

- When you split a tensor (like weights or activations) across GPUs, each GPU does part of the computation.
- But to get the final result (e.g., output or gradient), the GPUs need to communicate and combine their partial results.
- All-Reduce = Compute + Share + Combine
  - Every GPU:
    - Computes its part
    - Shares it with other GPUs
    - Receives others' parts
    - Combines everything into a final result (e.g., a full gradient)

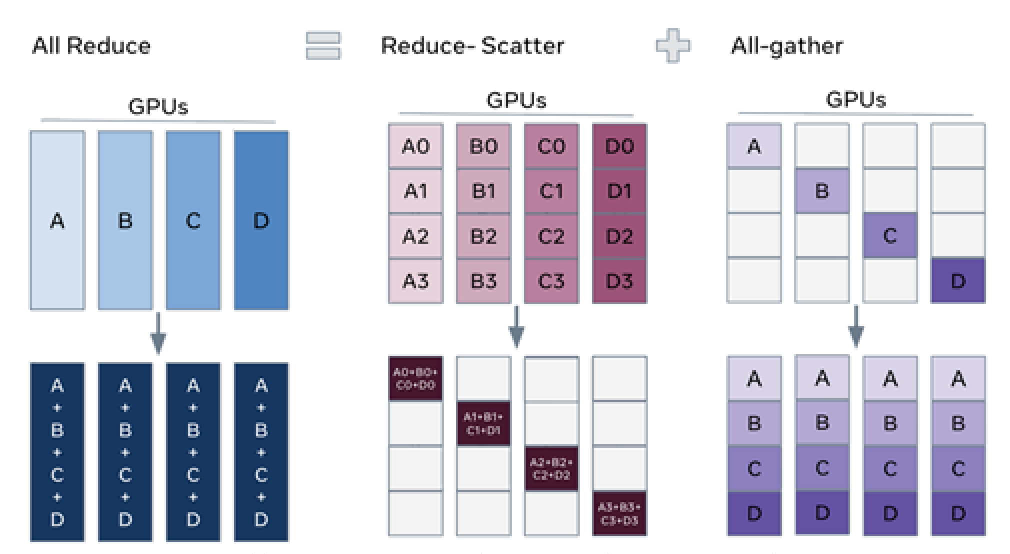


figure: https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Tensor (Sharded) Parallelism

Pros

- Scalability: Excellent (sometimes a must) for huge models
- Lower memory usage per GPU (can do it with cheaper GPUs)

Cons

- You must have many GPUs!
- Complexity: High implementation effort
- Significant communication overhead
- Tooling / Debugging
    - Requires advanced frameworks *which exist*

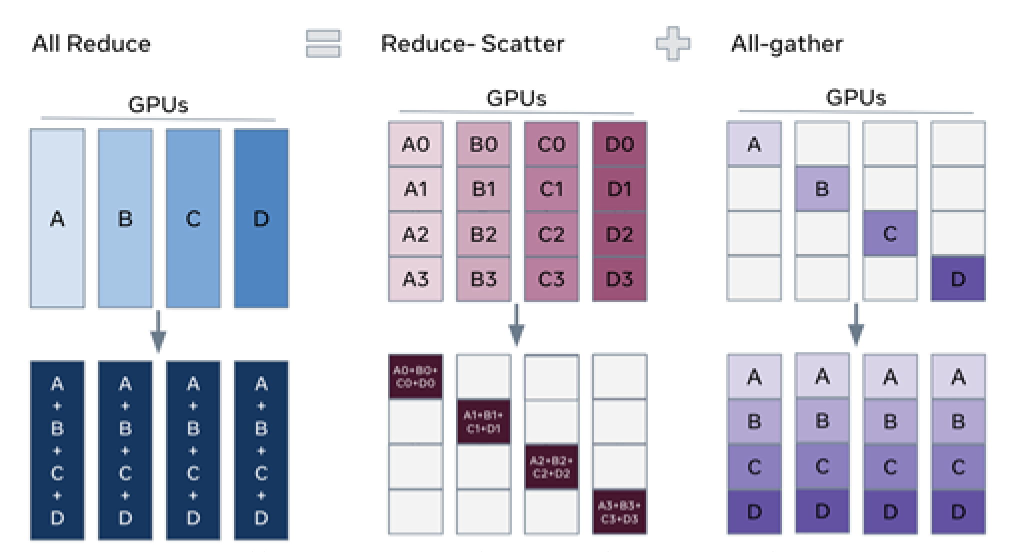


figure: https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Next: Part IV.b

TÜRKİYE EURO 4SEE

- PIV.a : Parallelism
- **PIV.b : Mixed-Precision Training**
- PIV.c : Other

EURO 4SEE

Thanks!

Co-funded by the European Union

EuroHPC Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101191697. The JU receives support from the Digital Europe Programme and Germany, Türkiye, Republic of North Macedonia, Montenegro, Serbia, Bosnia and Herzegovina.