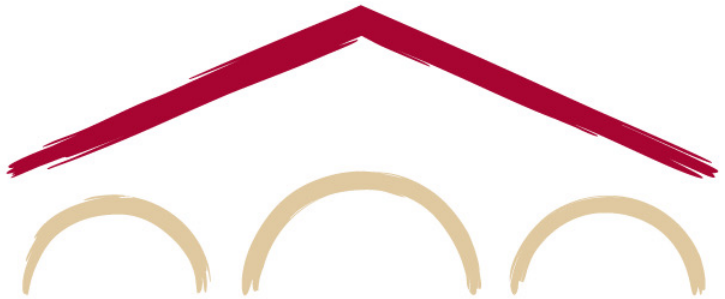
# Natural Language Processing with Deep Learning CS224N/Ling284



Shikhar Murty

Lecture 12: Efficient Neural Network Training

# Lecture Plan

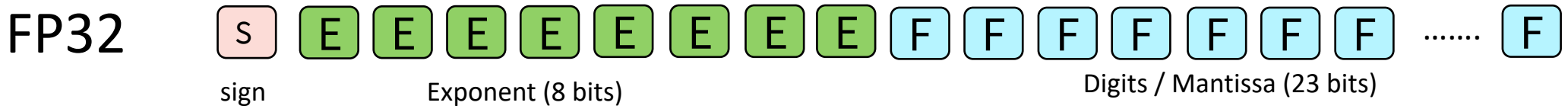
**Lecture 12: Efficient Neural Network Training - hopefully useful for final projects!**

1. Mixed Precision Training [20 mins]
2. Multi-GPU Training with DDP / FSDP [40 mins]
3. Parameter Efficient Finetuning: LoRA [20 mins]

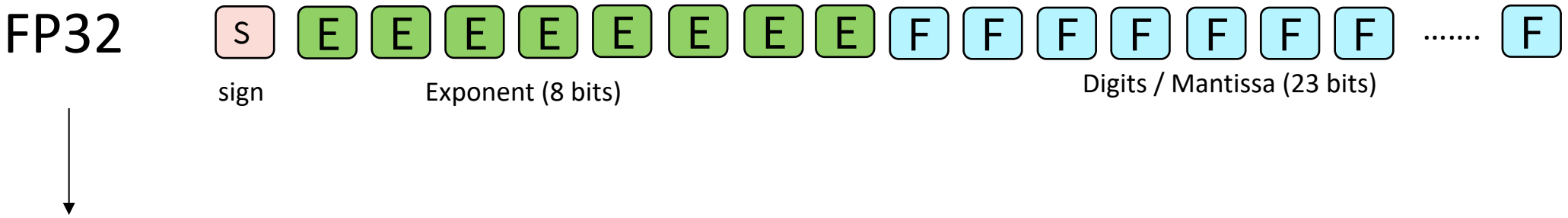
- Announcements
  - Proposal grades coming out today
  - Final project milestone details is out today!
    - Worth 5% of overall grade
    - Due on May 21$^{st}$ (12 days to work on this)
    - Max 2 pages
    - **Use this as a forcing function to get work done for your final project!**

# Mixed Precision Training

# Floating Points 101



# Floating Points 101




Memory requirement: 4 bytes

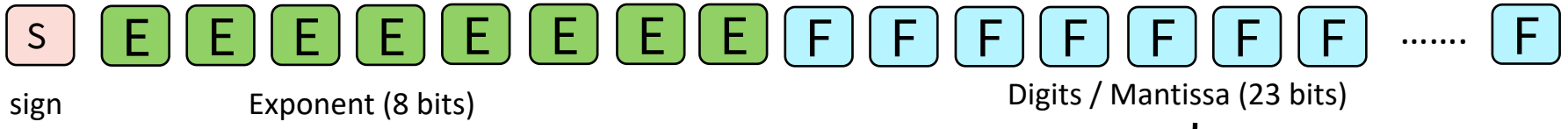
# Floating Points 101

FP32


$$(-1)^B \times 2^{E-127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

# Floating Points 101

FP32




$$(-1)^B \times 2^{E-127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

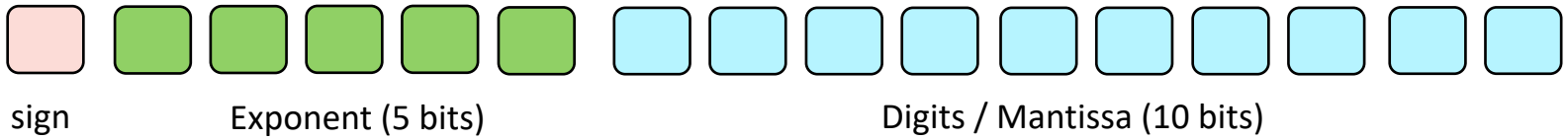
range

precision

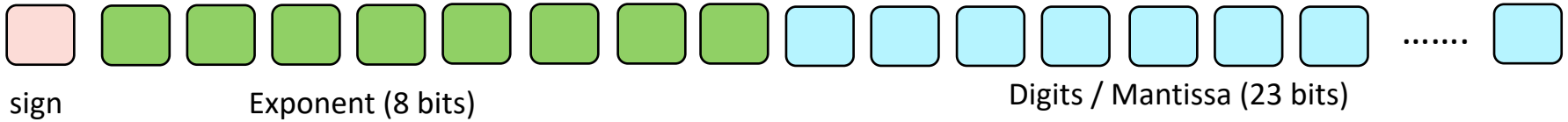
Can represent $[2^k, 2^k\,(1\,+\epsilon), 2^k(1\,+\,2\,\epsilon)\,, \ldots, 2^{k+1}]$ where $\epsilon = 2^{-23}$

# Floating Points 101

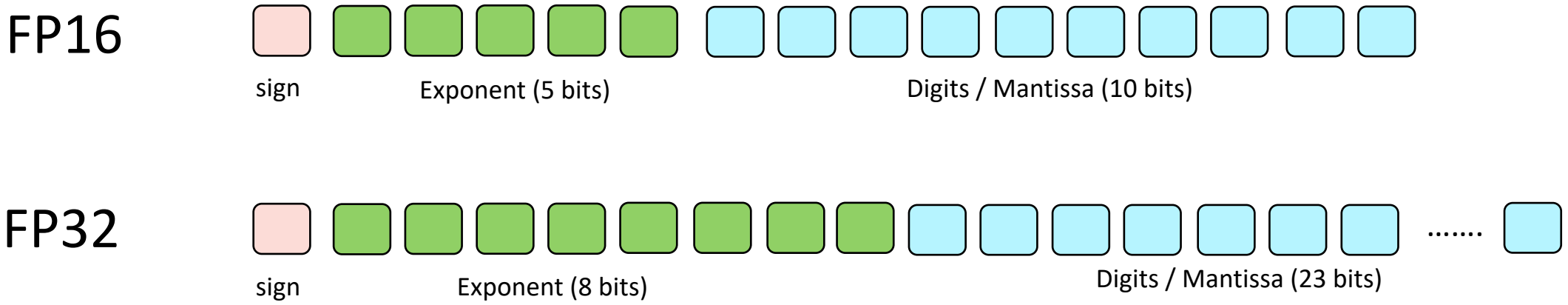
FP16




FP32



# Training Neural Networks in Half-Precision?



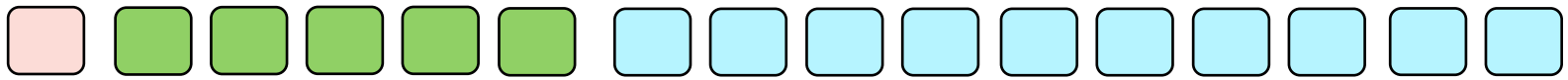

- Standard Neural Network Training: Model parameters and gradients represented in FP32 (CUDA OOM errors with large models).
- Possible solution: Use FP16!

# Training Neural Networks in Half-Precision?

- Standard Neural Network Training: Model parameters and gradients represented in FP32 (CUDA OOM errors with large models).
- Possible solution: Use FP16!
  - Less range: Roughly 2e-14 to 2e15 on both sides
  - Smaller precision leads to rounding errors: 1.0001 is 1 in half precision

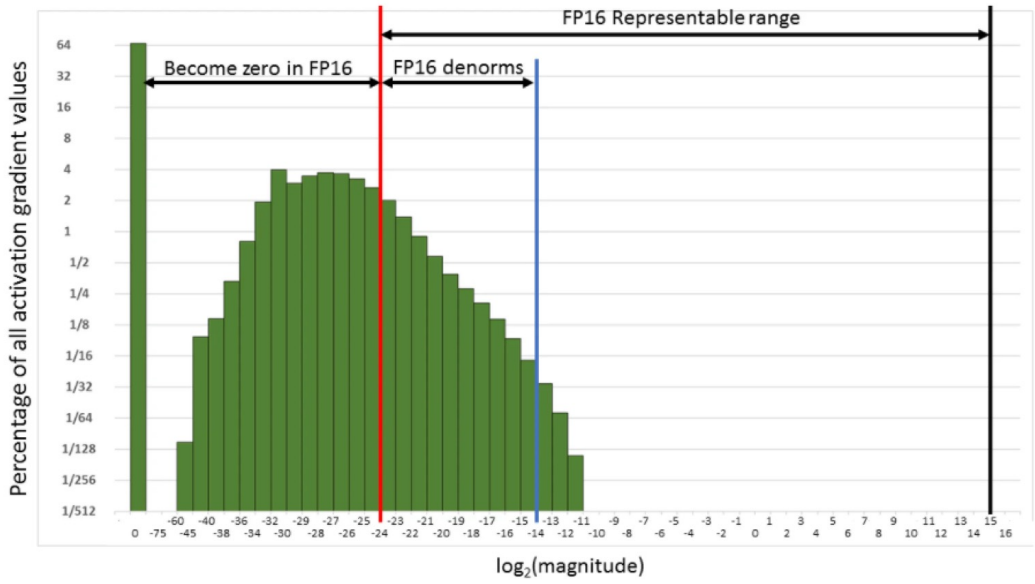
```
>>> torch.finfo(torch.float16)

finfo(resolution=0.001, min=-65504, max=65504,
eps=0.000976562, smallest_normal=6.10352e-05,
tiny=6.10352e-05, dtype=float16)
```

FP16



# Training Neural Networks in Half-Precision?

- Standard Neural Network Training: Model parameters and gradients represented in FP32 (CUDA OOM errors with large models).
- Possible solution: Use FP16!
  - Less range: Roughly 2e-14 to 2e15 on both sides
  - Smaller precision leads to rounding errors: 1.0001 is 1 in half precision
  - For Neural Net training:
    - Gradients can underflow



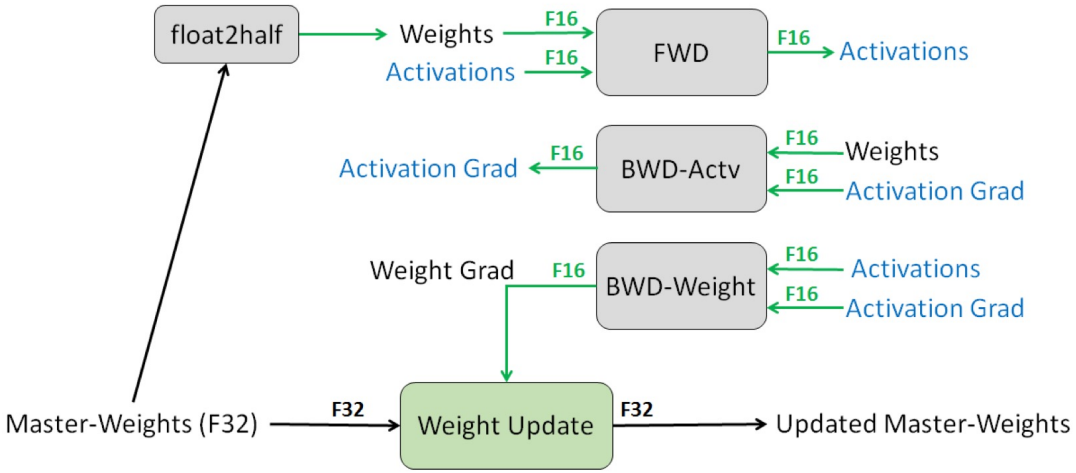
# Training Neural Networks in Half-Precision?

- Standard Neural Network Training: Model parameters and gradients represented in FP32 (CUDA OOM errors with large models).
- Possible solution: Use FP16!
  - Less range: Roughly 2e-14 to 2e15 on both sides
  - Smaller precision leads to rounding errors: 1.0001 is 1 in half precision
  - For Neural Net training:
    - Gradients can underflow
    - Weight updates are imprecise

# *Solution*: Mixed Precision Training

- Still use FP16, but use FP32 for neural network updates!

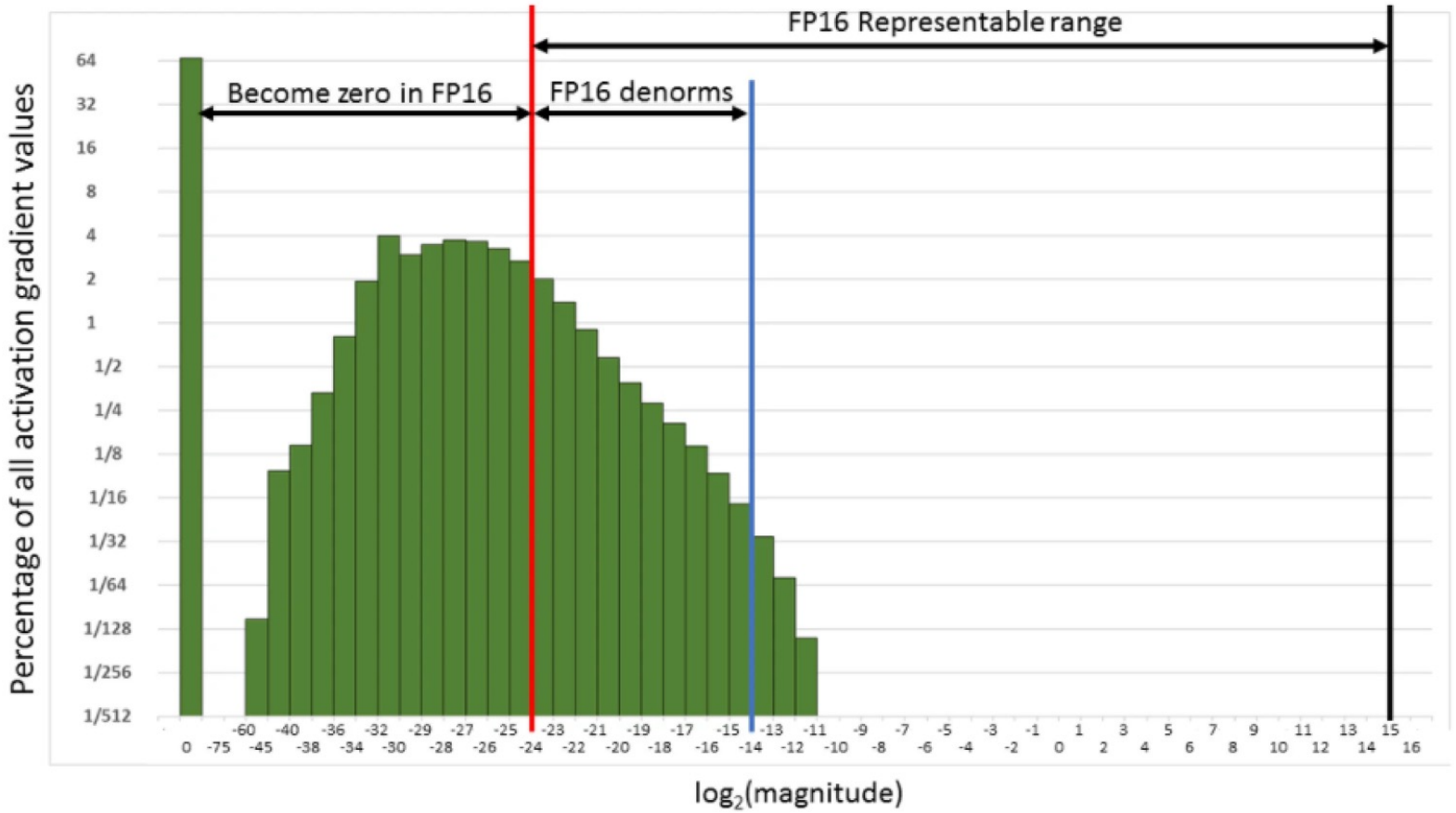
Sharang et al. 2018




## Take-2

1. Maintain a copy of model parameters in FP32 (Master weights)
2. Run forward pass in FP16
3. Compute gradient in FP16
4. Copy gradient into FP32
5. Update master weights in FP32 *[fixes weight update issue!]*
6. Copy into FP16 version

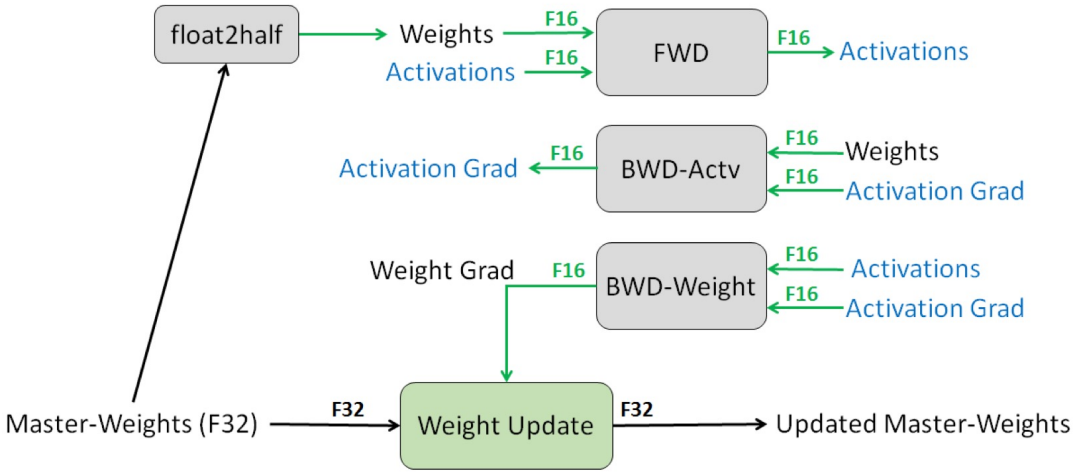
# *Solution*: Mixed Precision Training



# *Solution*: Mixed Precision Training

- Still use FP16, but use FP32 for neural network updates!

Sharang et al. 2018




## Take-2

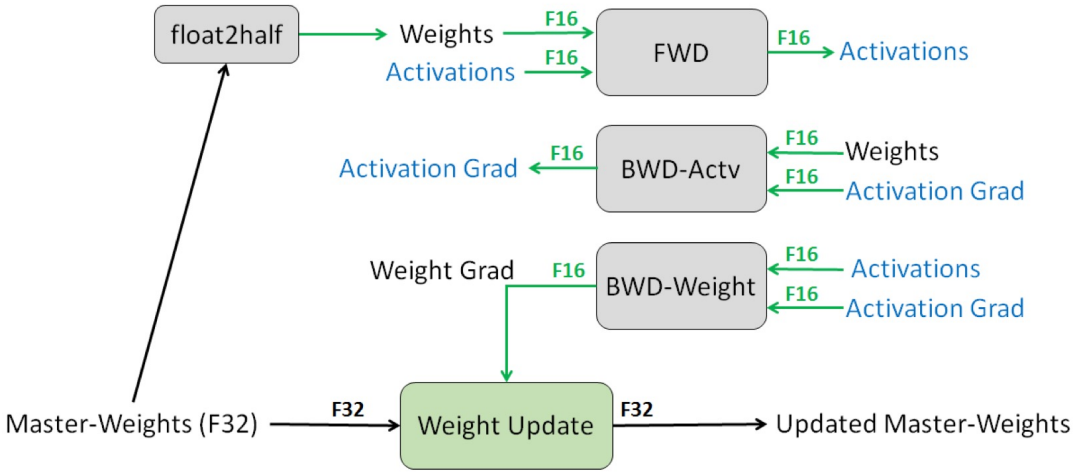
1. Maintain a copy of model parameters in FP32 (Master weights)
2. Run forward pass in FP16
3. Compute gradient in FP16
4. Copy gradient into FP32
5. Update master weights in FP32 *[fixes weight update issue!]*
6. Copy into FP16 version

Here, gradients can still underflow (small gradients will become exactly 0).

# *Solution*: Mixed Precision Training

- Still use FP16, but use FP32 for neural network updates!

Sharang et al. 2018




## Recipe for Mixed-Precision Training

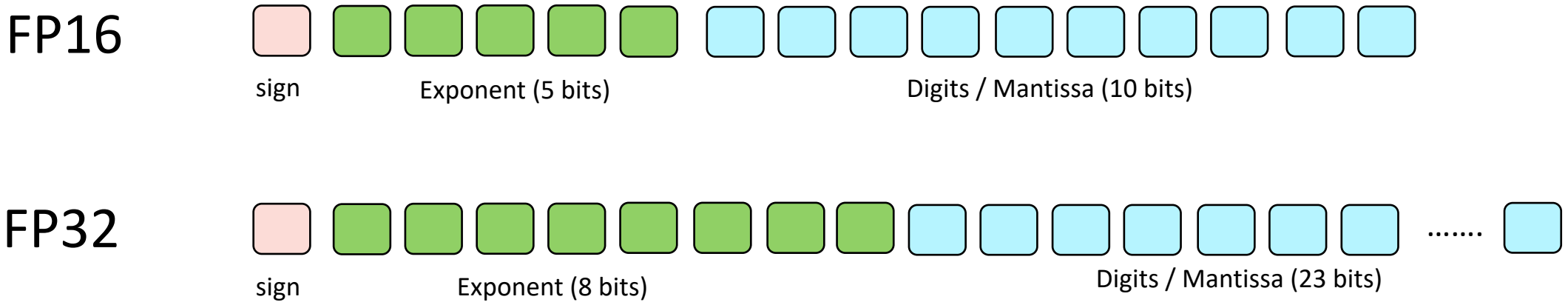
1. Maintain a copy of model parameters in FP32 (Master weights)
2. Run forward pass in FP16
3. Scale loss by a large value (to artificially increase gradient)
4. Compute gradient in FP16
5. Copy gradient into FP32 and divide by scale factor
6. Update master weights in FP32 *[fixes weight update issue!]*
7. Copy into FP16 version

# Mixed Precision Training in PyTorch

```
# Creates model and optimizer in default precision
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

# Creates a GradScaler once at the beginning of training.
scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()

        # Runs the forward pass with autocasting.
        with autocast(device_type='cuda', dtype=torch.float16):
            output = model(input)
            loss = loss_fn(output, target)

        # Scales loss.  Calls backward() on scaled loss to create scaled gradients.
        # Backward passes under autocast are not recommended.
        # Backward ops run in the same dtype autocast chose for corresponding forward ops.
        scaler.scale(loss).backward()

        # scaler.step() first unscales the gradients of the optimizer's assigned params.
        # If these gradients do not contain infs or NaNs, optimizer.step() is then called,
        # otherwise, optimizer.step() is skipped.
        scaler.step(optimizer)

        # Updates the scale for next iteration.
        scaler.update()
```

Source: https://pytorch.org/docs/stable/notes/amp_examples.html

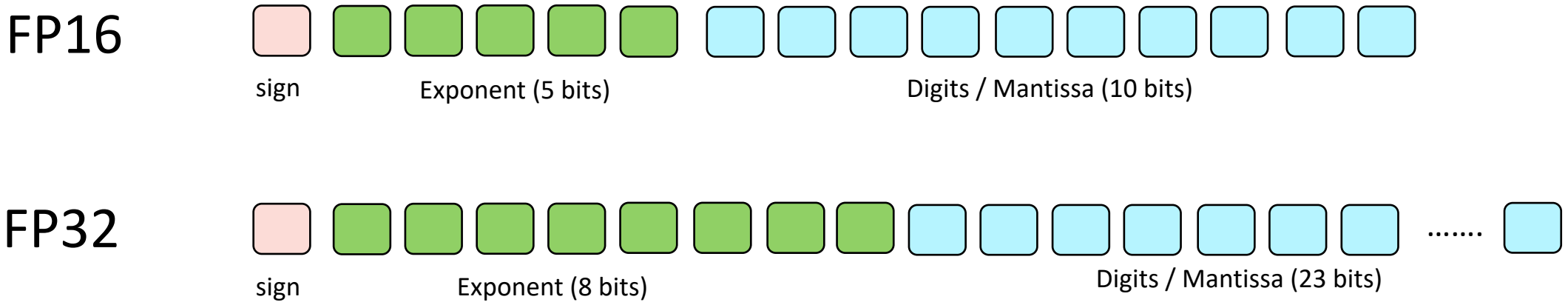
# Can we get rid of gradient scaling?




We need scaling because FP16 has a small range compared to FP32

```
>>> torch.finfo(torch.float16)

finfo(resolution=0.001, min=-65504, max=65504,
eps=0.000976562, smallest_normal=6.10352e-05,
tiny=6.10352e-05, dtype=float16)
```

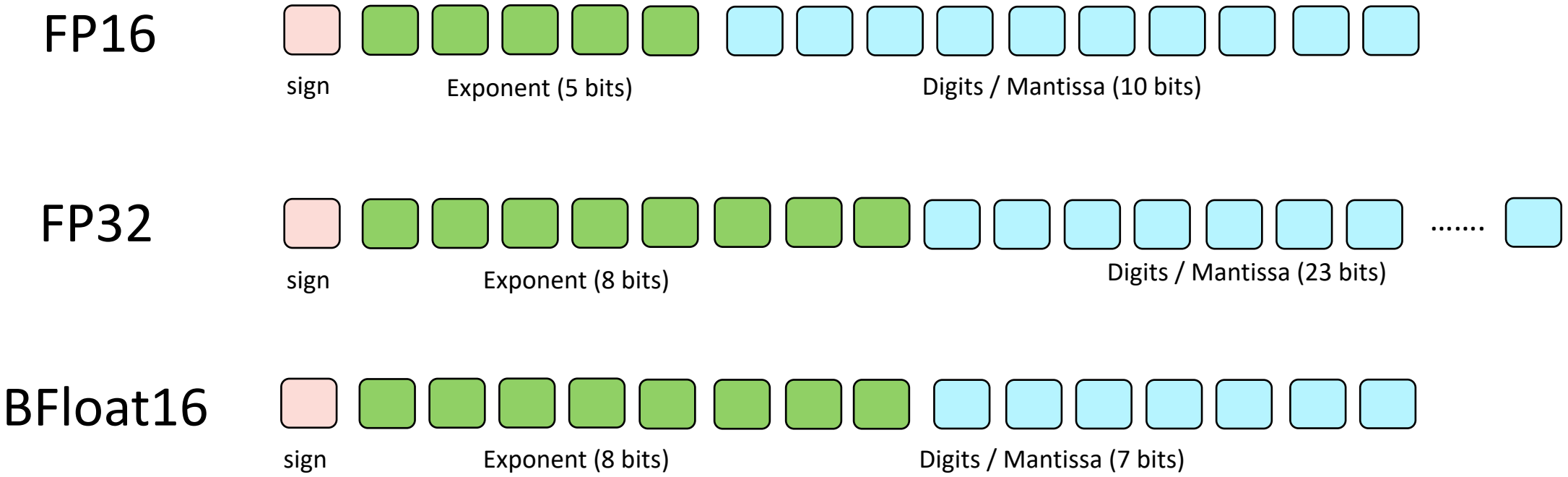
# Can we get rid of gradient scaling?




We need scaling because FP16 has a small range compared to FP32

💡Can we allocate 8 bits for exponent (same range) while sacrificing precision?

# Greater Dynamic Range with Bfloat16




Greater Dynamic Range with Bfloat16:

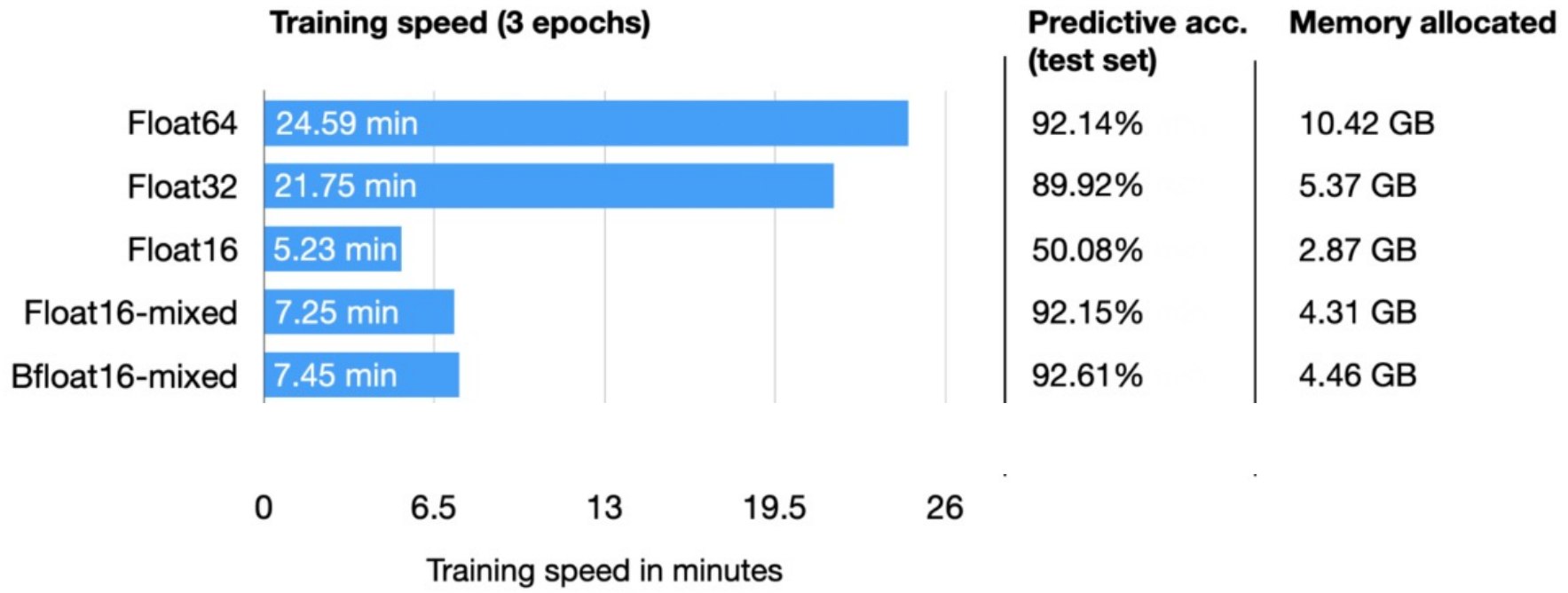
can represent much smaller numbers and much larger numbers (no INF / NaNs)

# Bfloat16 does not need GradScalars

```
# Creates model and optimizer in default precision
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

for input, target in data:
    optimizer.zero_grad()

    # Enables autocasting for the forward pass (model + loss)
    with torch.autocast(device_type="cuda"):
        output = model(input)
        loss = loss_fn(output, target)

    # Exits the context manager before backward()
    loss.backward()
    optimizer.step()
```

Source: https://pytorch.org/docs/stable/amp.html#torch.autocast

# Greater Dynamic Range with Bfloat16

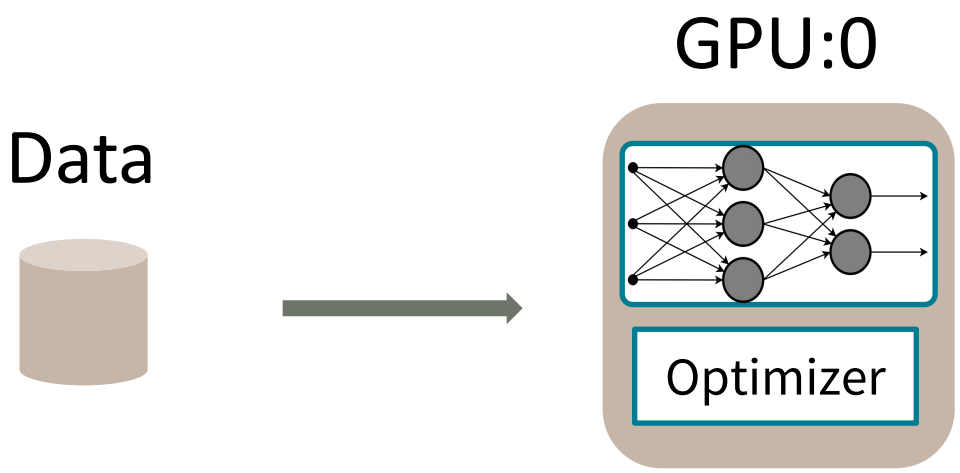



Results from finetuning DistilBERT for sentiment classification on a single A100 GPU.

Source: https://sebastianraschka.com/blog/2023/llm-mixed-precision-copy.html

# Multi-GPU Training

# Multi-GPU Training




What's stored on GPU VRAM?

NN: Model parameters (in FP16)

Optimizer:
Master weights (FP32) +
Adam momentum (FP32) +
Adam variance (FP32) +

## Adam Optimizer

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla w_t$$

minibatch gradient

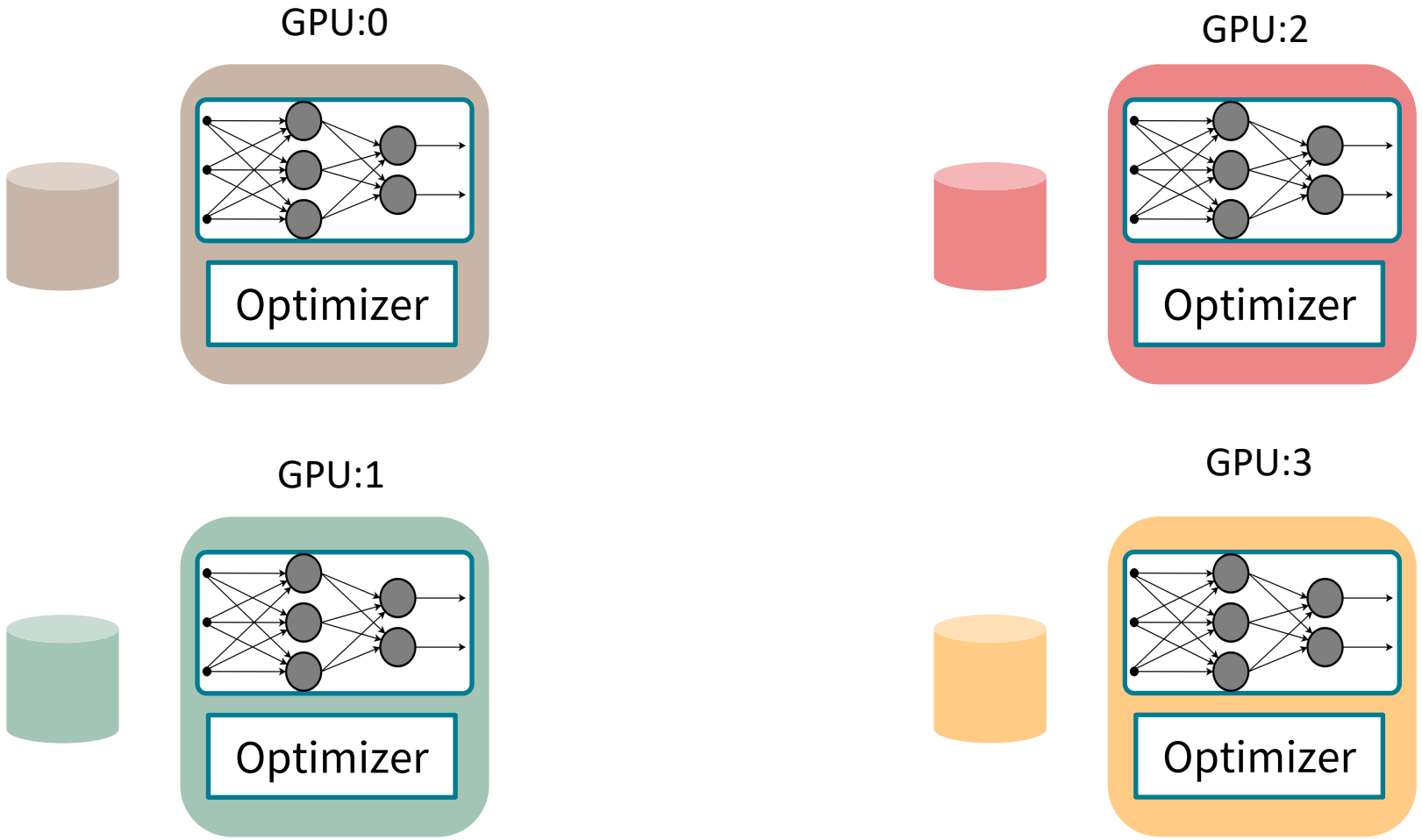
$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * (\nabla w_t)^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

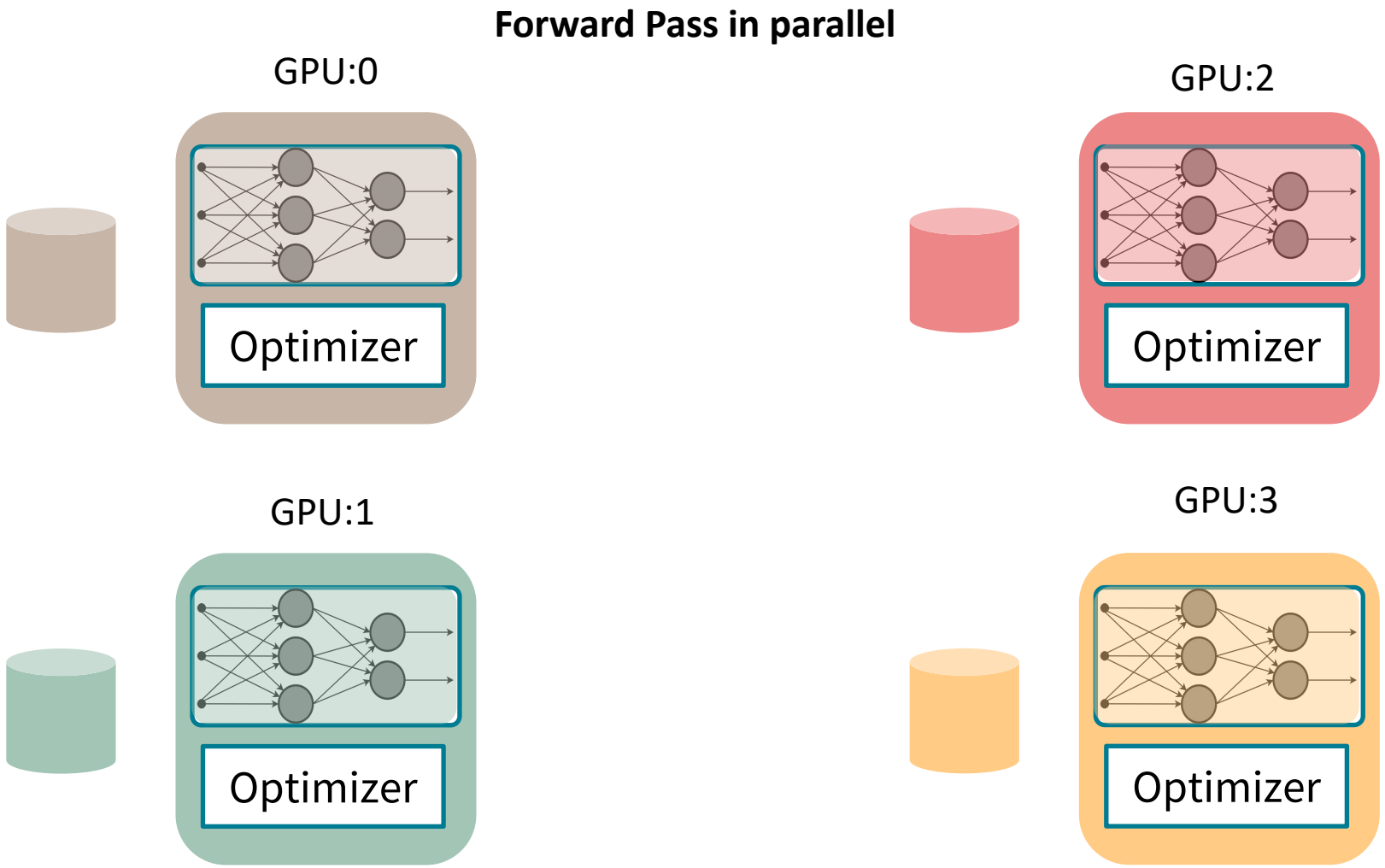
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

# The Basics: Distributed Data Parallel (DDP)

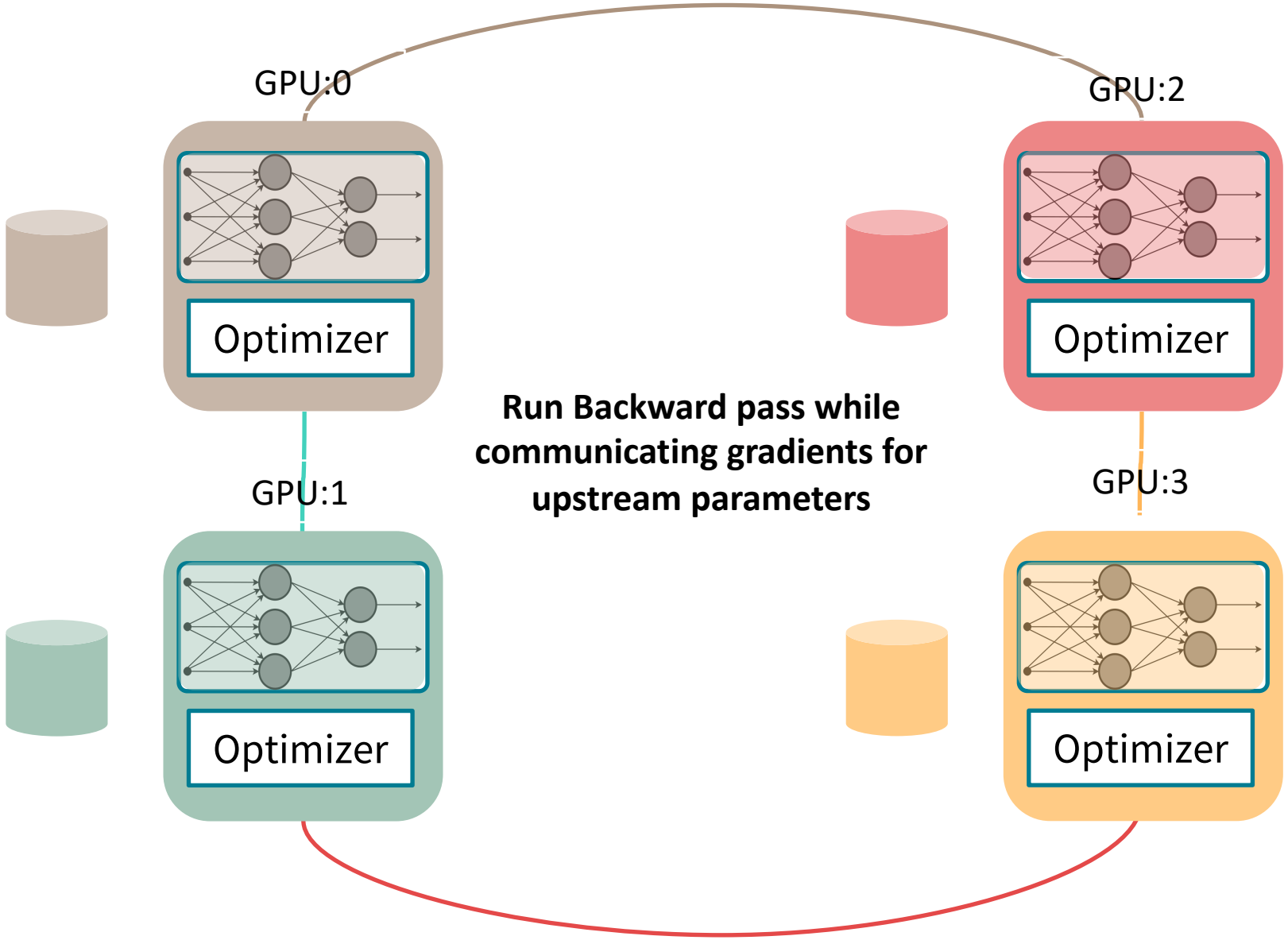
**Each GPU has a synchronized copy of the model with its own slice of the data**



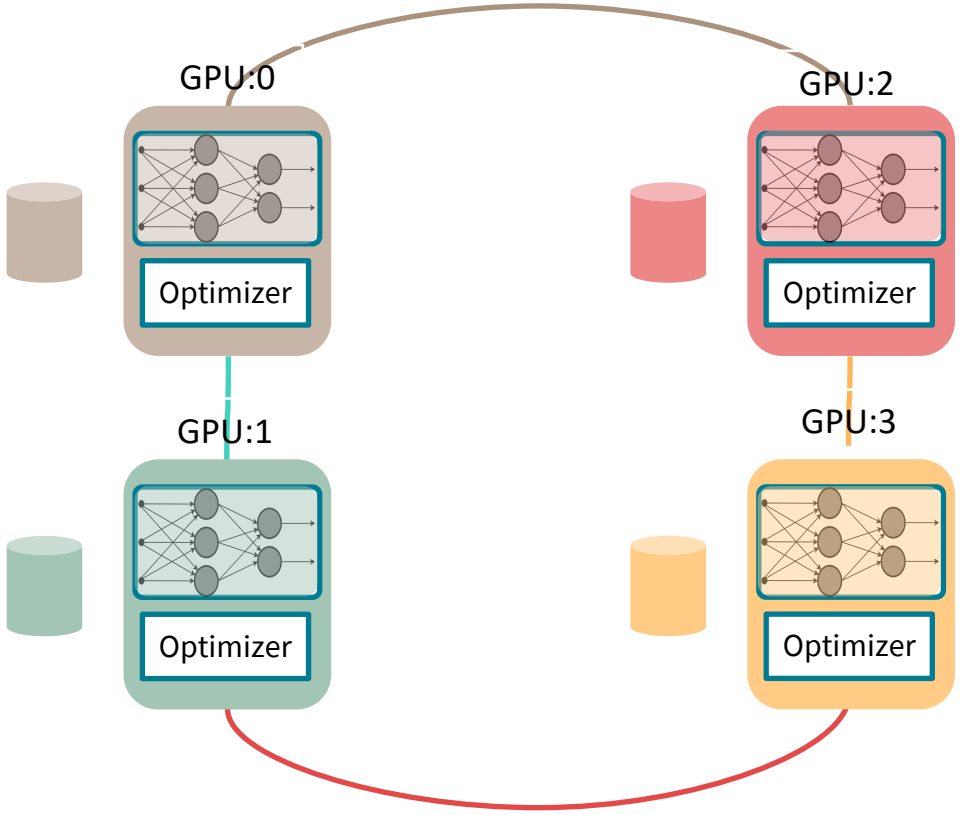
# The Basics: Distributed Data Parallel (DDP)



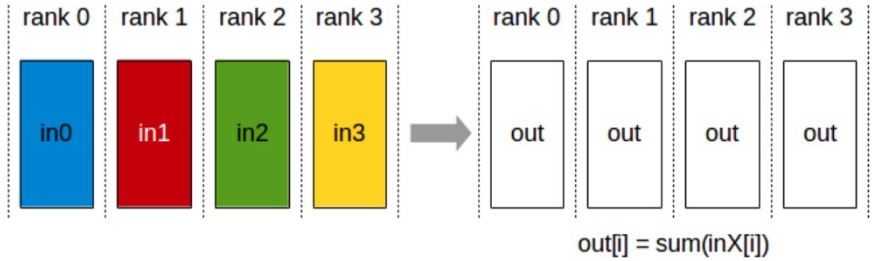
# The Basics: Distributed Data Parallel (DDP)



# The Basics: Distributed Data Parallel (DDP)

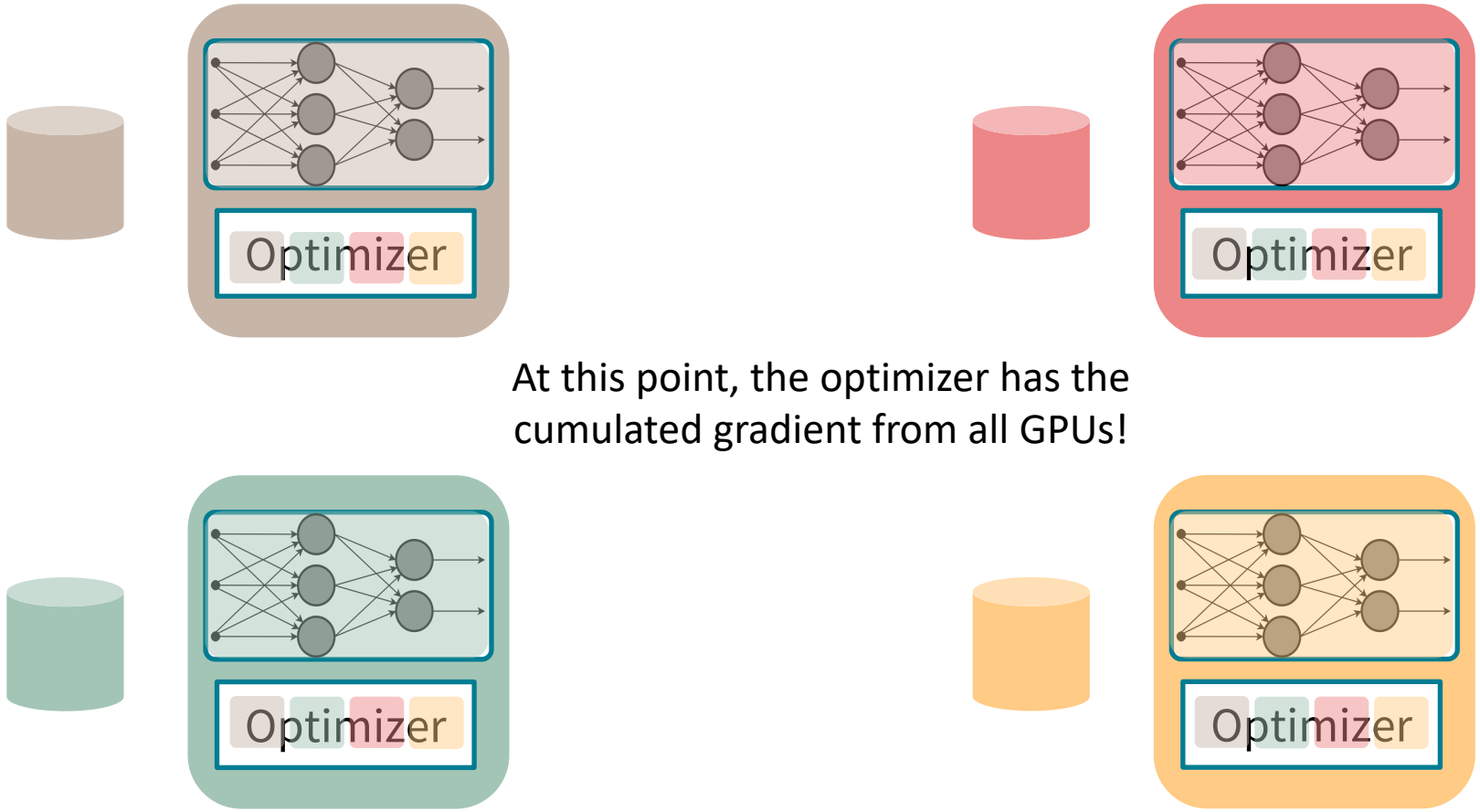



**“All Reduce” Operation:**



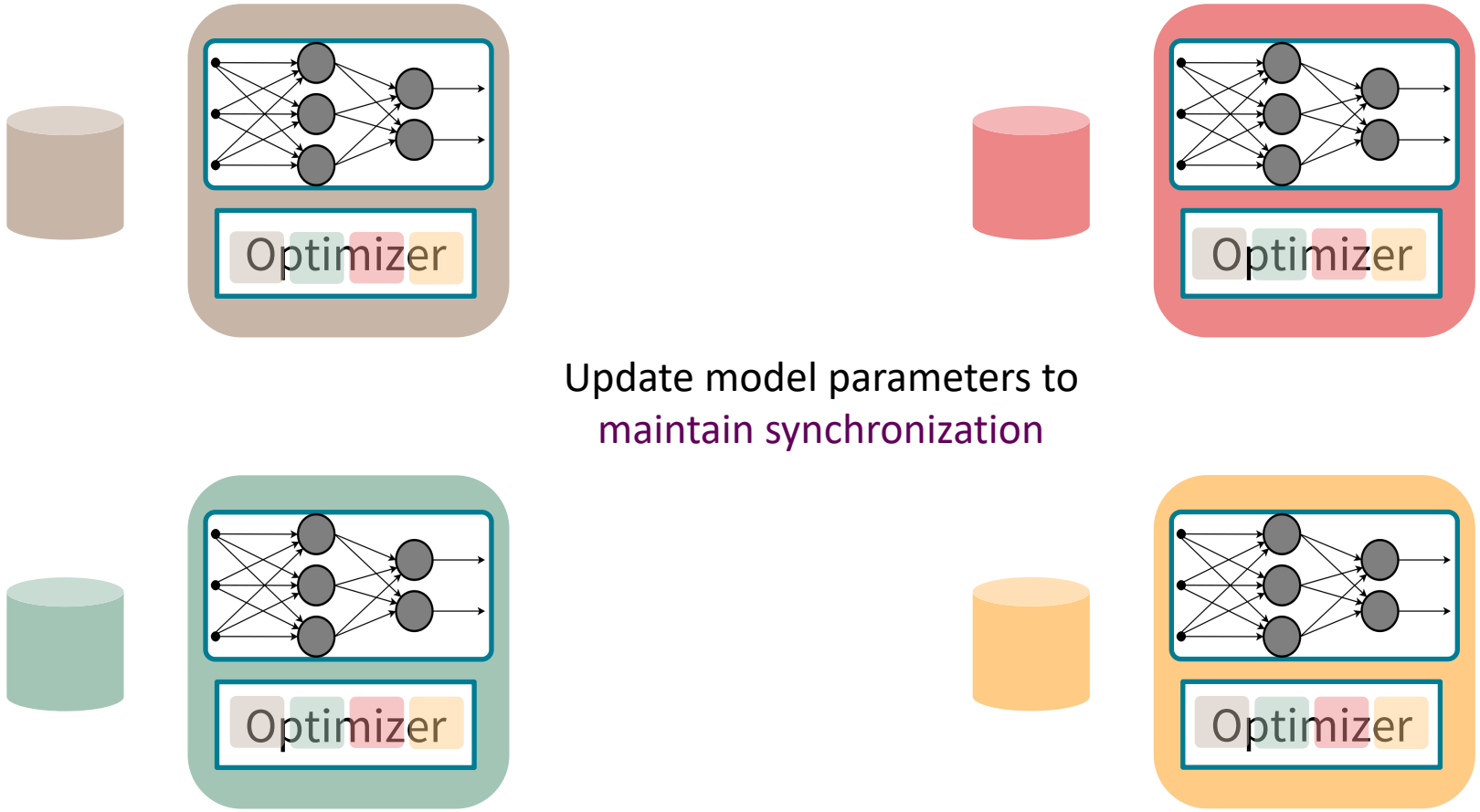

Communication overhead: 2 bytes per parameter (gradients in FP16)

# The Basics: Distributed Data Parallel (DDP)

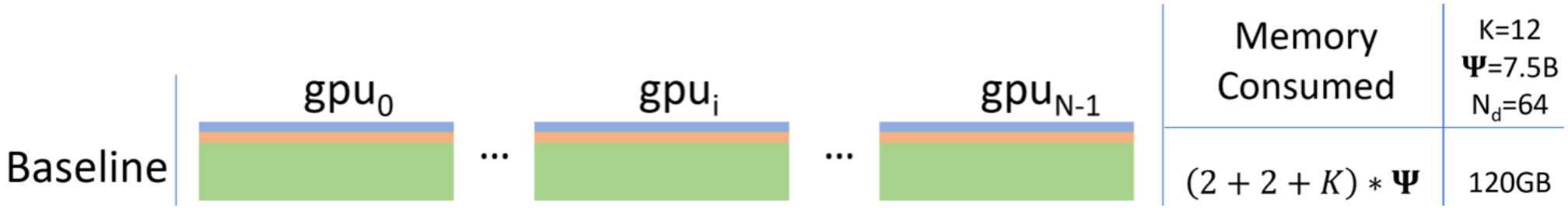



At this point, the optimizer has the cumulated gradient from all GPUs!

# The Basics: Distributed Data Parallel (DDP)



# Unfortunately, Naive DDP has poor memory scaling




2 bytes for FP16 parameters

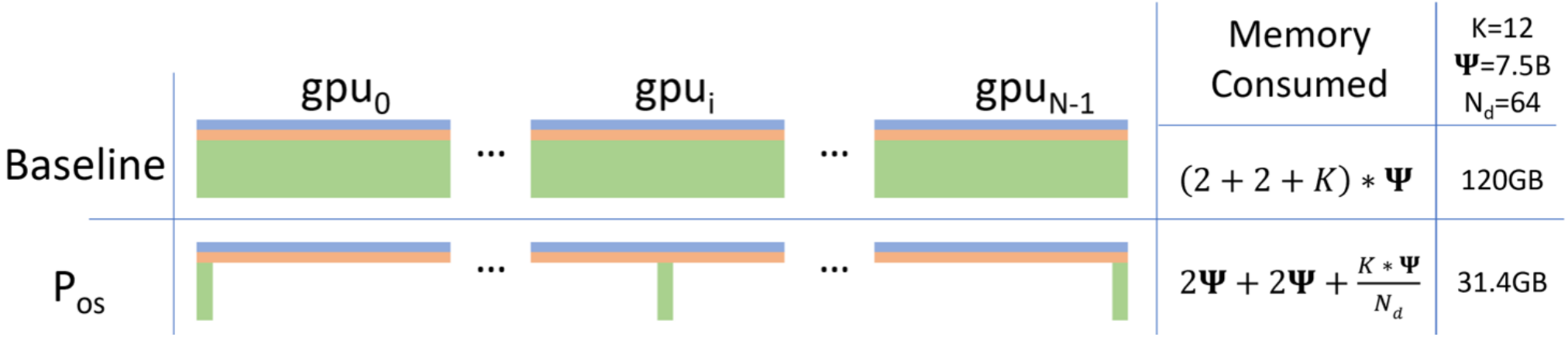
2 bytes for FP16 backward pass gradients

4 bytes for FP32 master weights

4 bytes for FP32 Adam momentum

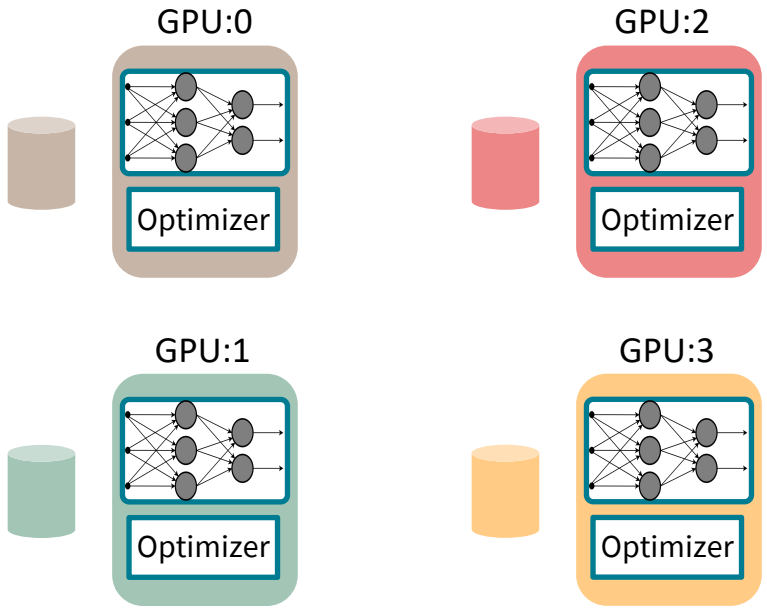
4 bytes for FP32 Adam variance

# ZeRO Stage-1: Optimizer State Sharding ($P_{os}$)

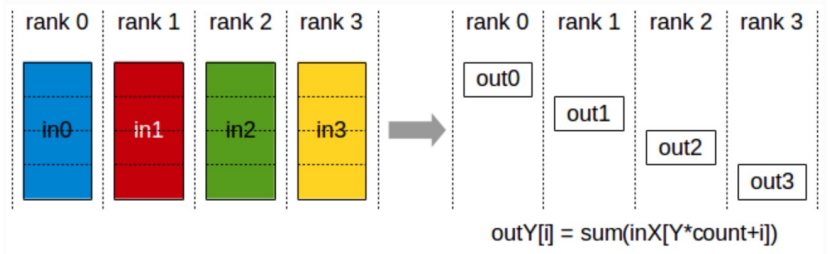



- Each GPU has the full set of FP16 model parameters, and computes the gradient on its subset of data
- Each GPU has a sharded copy of the full optimizer state.
- Each GPU is responsible for updating a shard of the full parameters

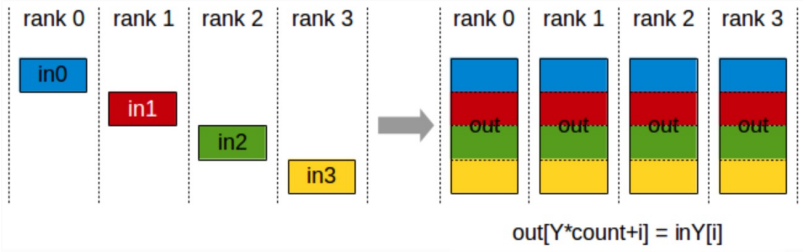
# ZeRO Stage-1: Optimizer State Sharding ($P_{os}$)



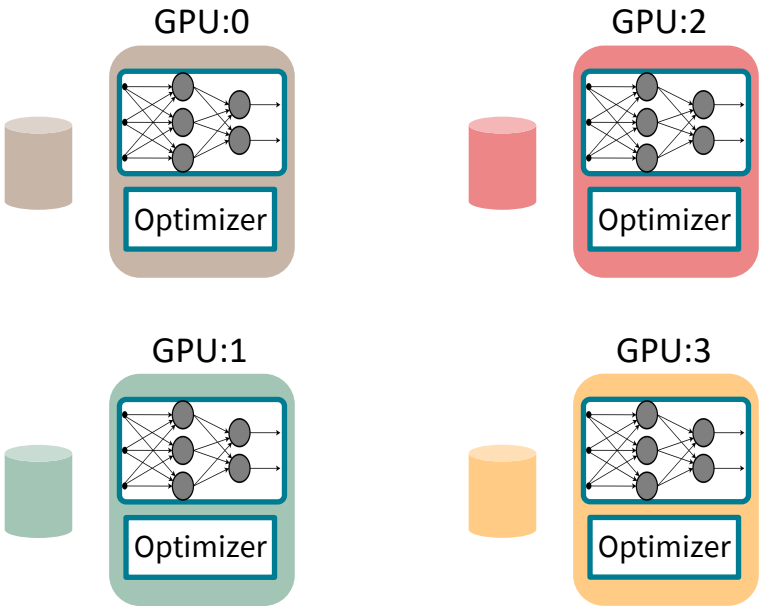

- Each worker computes gradient on its subset of data.
- Perform a reduce-scatter so that each worker gets the full gradient corresponding to their parameter shard:

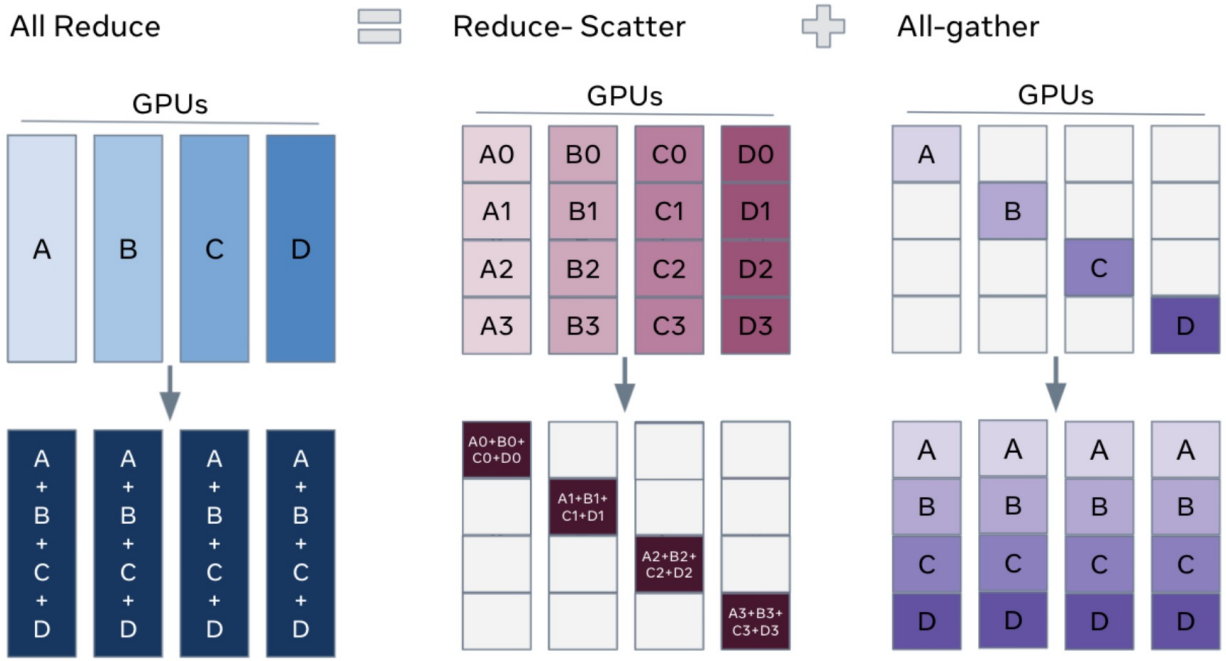



- Each worker updates its parameters
- Perform an all-gather to synchronize params



# ZeRO Stage-1: Optimizer State Sharding ($P_{os}$)



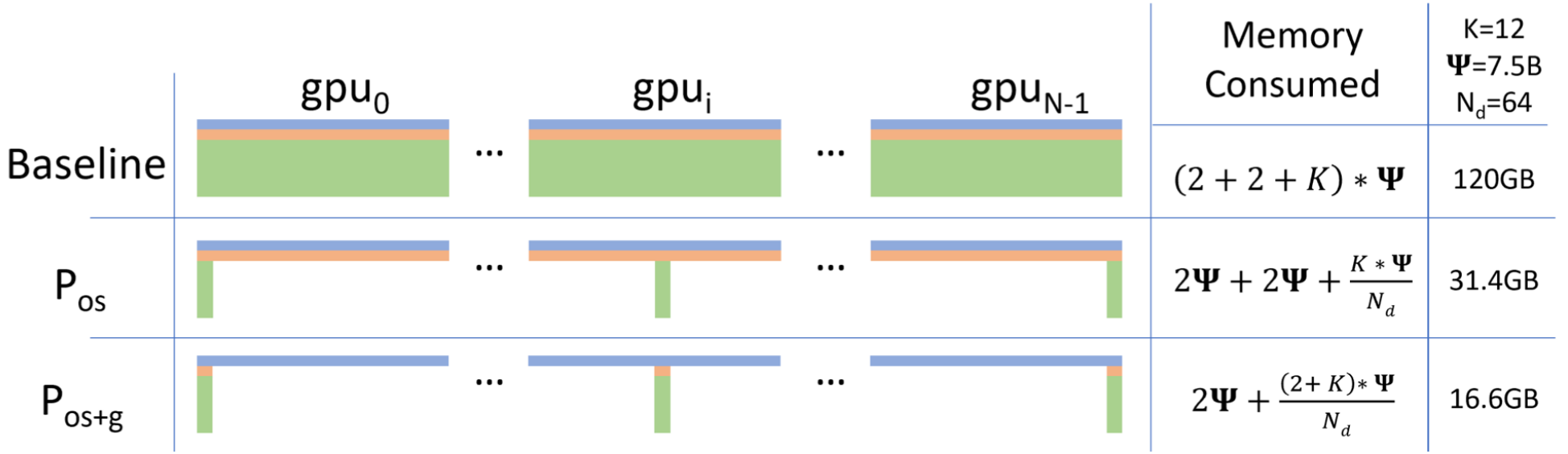

Source: https://engineering.fb.com/2021/07/15/open-source/fsdp/attachment/fsdp-graph-2a/

We used all-reduce for DDP

Communication overhead: 2 bytes per parameter (gradients in FP16)

We saved memory for free!

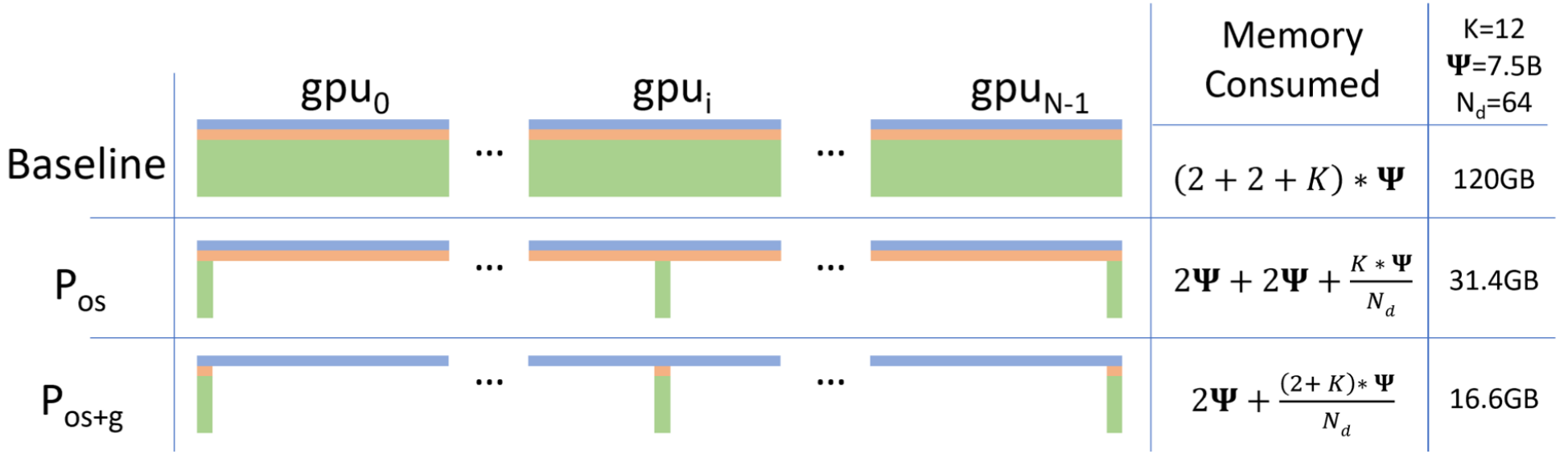
# ZeRO Stage-2: Optimizer State + gradient sharding ($P_{os+g}$)




| | Memory Consumed | K=12 $\Psi$=7.5B $N_d$=64 |
|---|---|---|
| Baseline | $(2+2+K)*\Psi$ | 120GB |
| $P_{os}$ | $2\Psi+2\Psi+\frac{K*\Psi}{N_d}$ | 31.4GB |
| $P_{os+g}$ | $2\Psi+\frac{(2+K)*\Psi}{N_d}$ | 16.6GB |

Along with sharing optimizer state, can we also shard gradients? 🤔
Complexity: We still need the full gradient for the worker's data slice!

# ZeRO Stage-2: Optimizer State + gradient sharding ($P_{os+g}$)



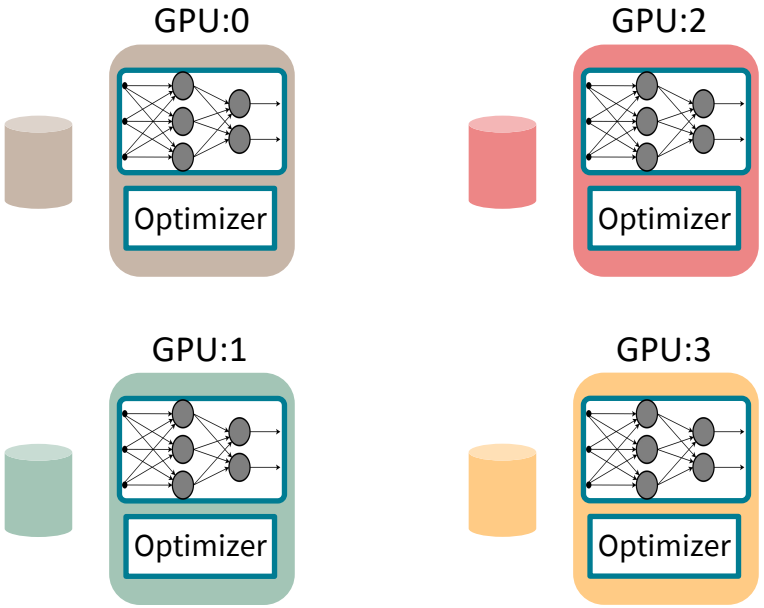

| | Memory Consumed | K=12 $\Psi$=7.5B $N_d$=64 |
|---|---|---|
| Baseline | $(2+2+K)*\Psi$ | 120GB |
| $P_{os}$ | $2\Psi+2\Psi+\frac{K*\Psi}{N_d}$ | 31.4GB |
| $P_{os+g}$ | $2\Psi+\frac{(2+K)*\Psi}{N_d}$ | 16.6GB |

Along with sharing optimizer state, can we also shard gradients? 🤔
Complexity: We still need the full gradient for the worker's data slice!

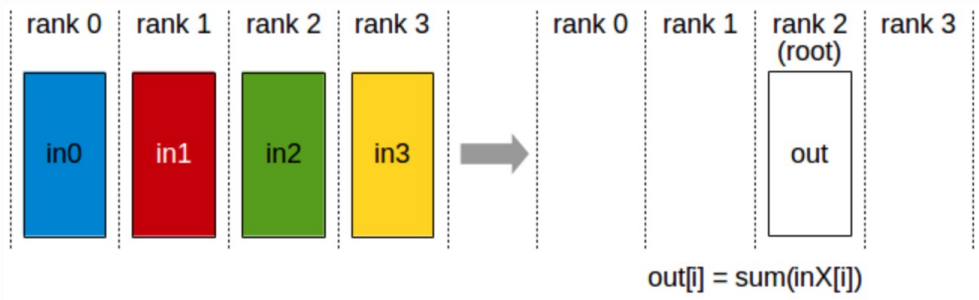
Solution:

- Never instantiate the full gradient vector!
- Send gradient to the "GPU in charge" as soon as the gradient for a shard is made available in the backward pass

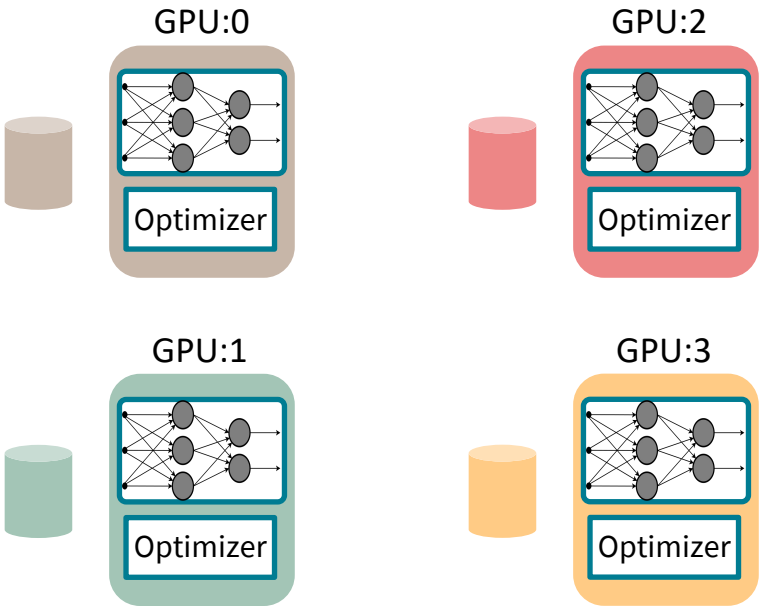
# ZeRO Stage-2: Optimizer State + gradient sharding ($P_{os+g}$)




- Worker performs a backward pass layer-by-layer in the computation graph
- Suppose worker is at layer-j:
  - Take upstream gradient, compute gradient for parameters at layer-j,
  - immediately send the gradients to the correct worker (reduce)
  - deallocate memory for parameter gradient.

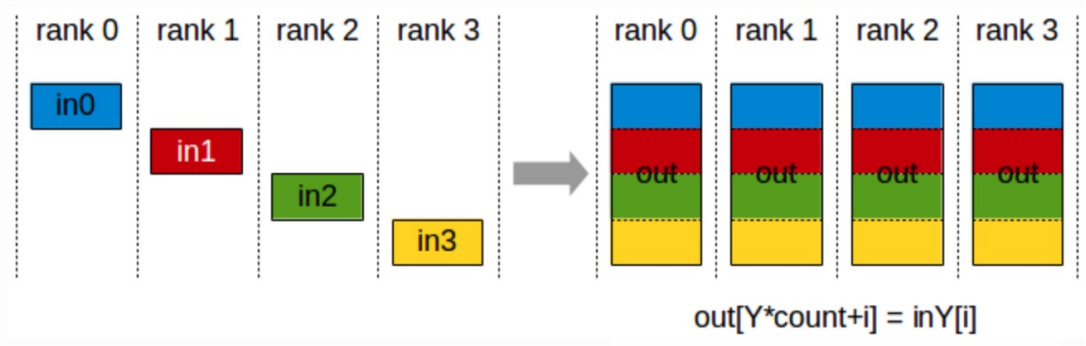


# ZeRO Stage-2: Optimizer State + gradient sharding ($P_{os+g}$)

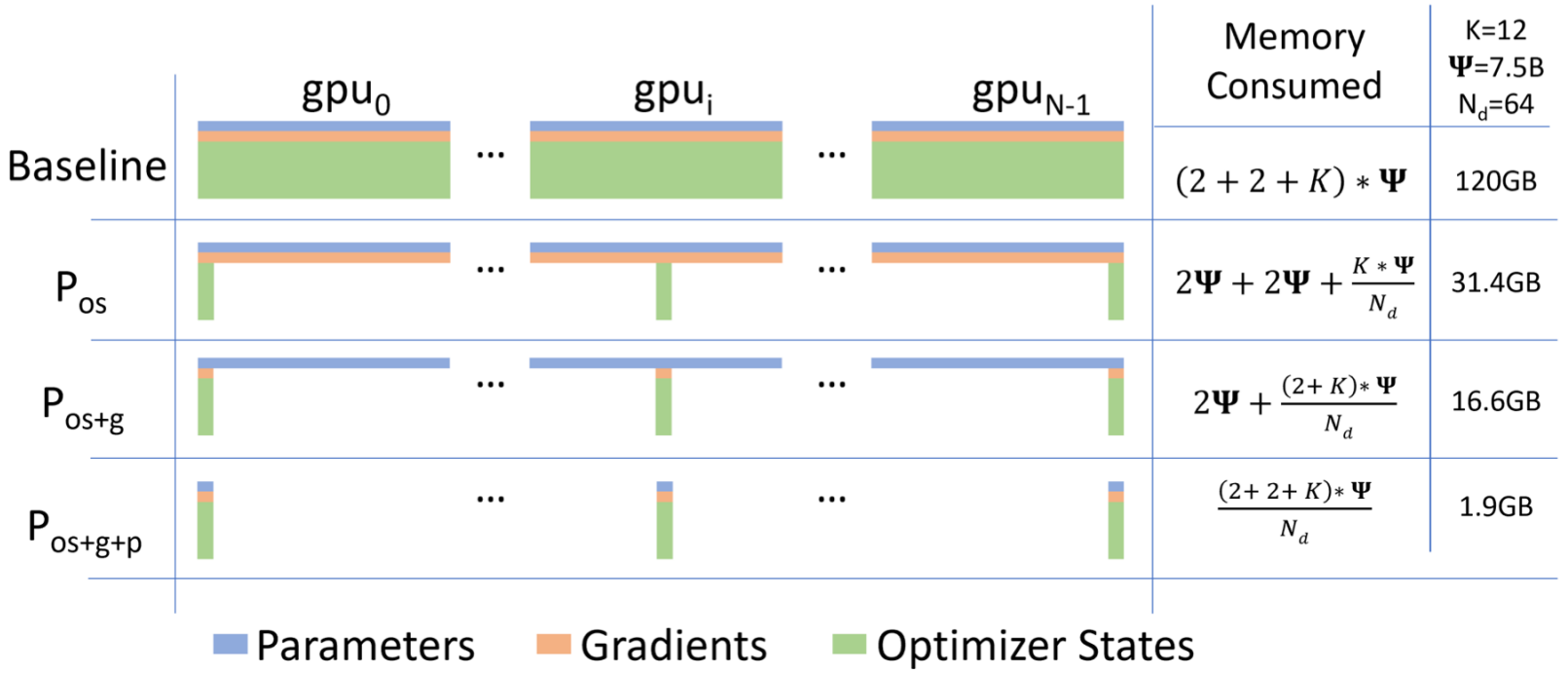



- Worker performs a backward pass layer-by-layer in the computation graph
- Suppose worker is at layer-j:
  - Take upstream gradient, compute gradient for parameters at layer-j,
  - immediately send the gradients to the correct worker (reduce)
  - deallocate memory for parameter gradient.

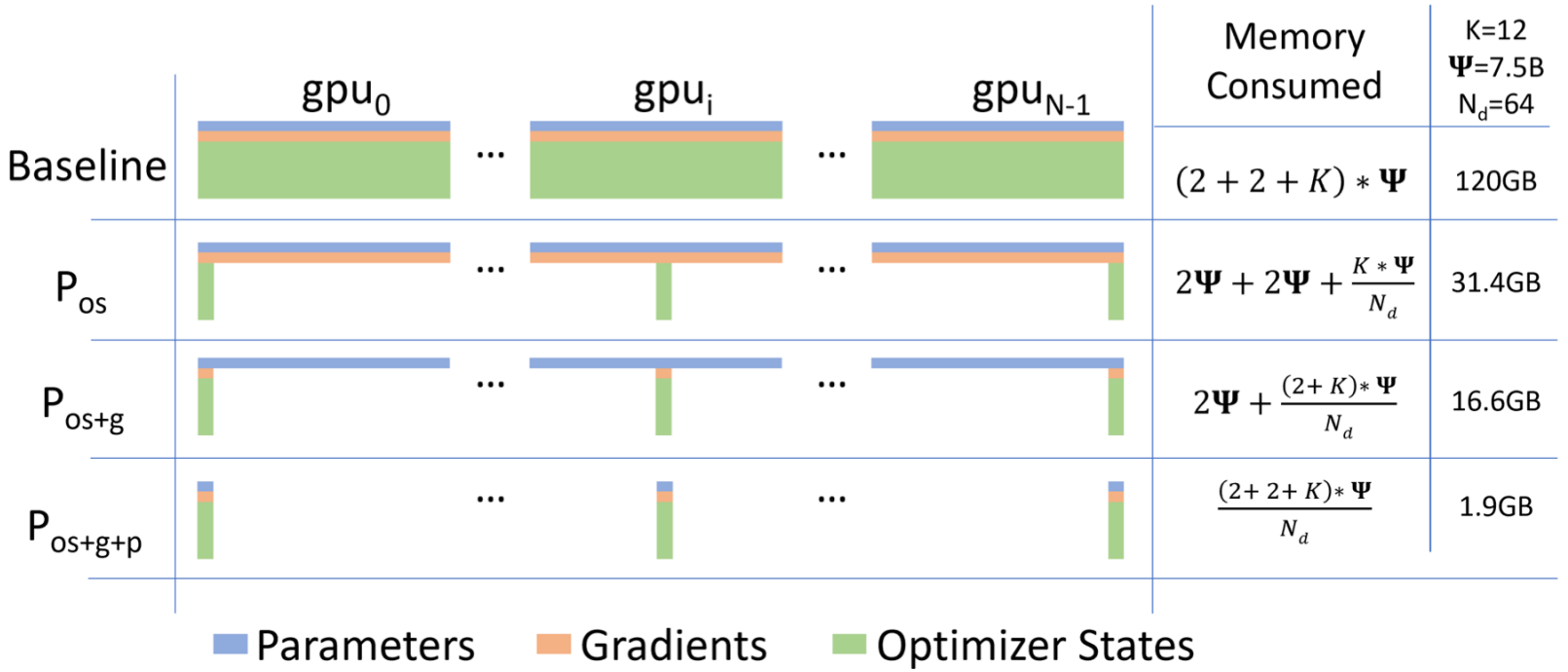
- Worker updates its param shard using corresponding gradient + state
- Perform an all-gather to synchronize



# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit

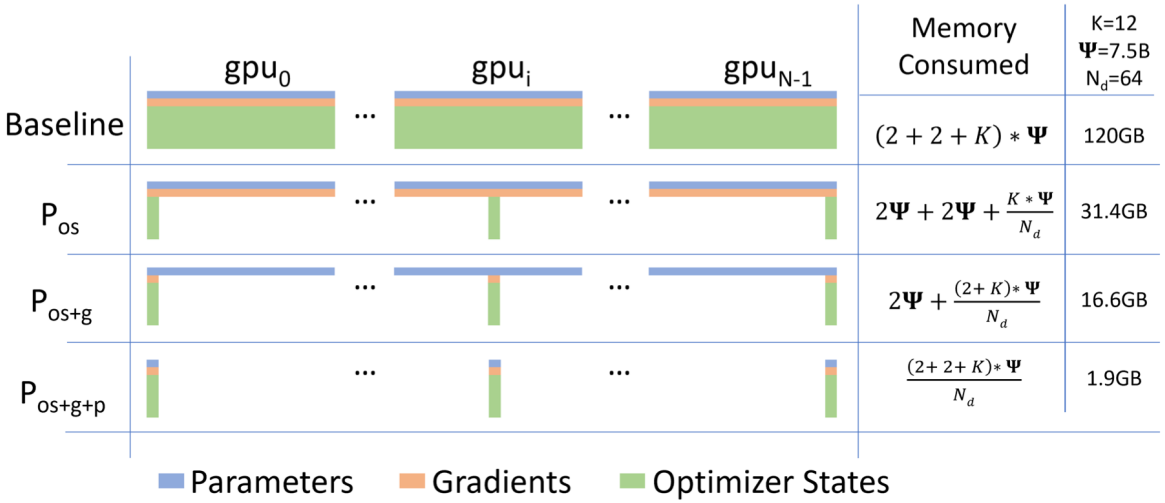


# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit




Caveat: So far, communication overhead was "free".
With Full FSDP, this is no longer the case.

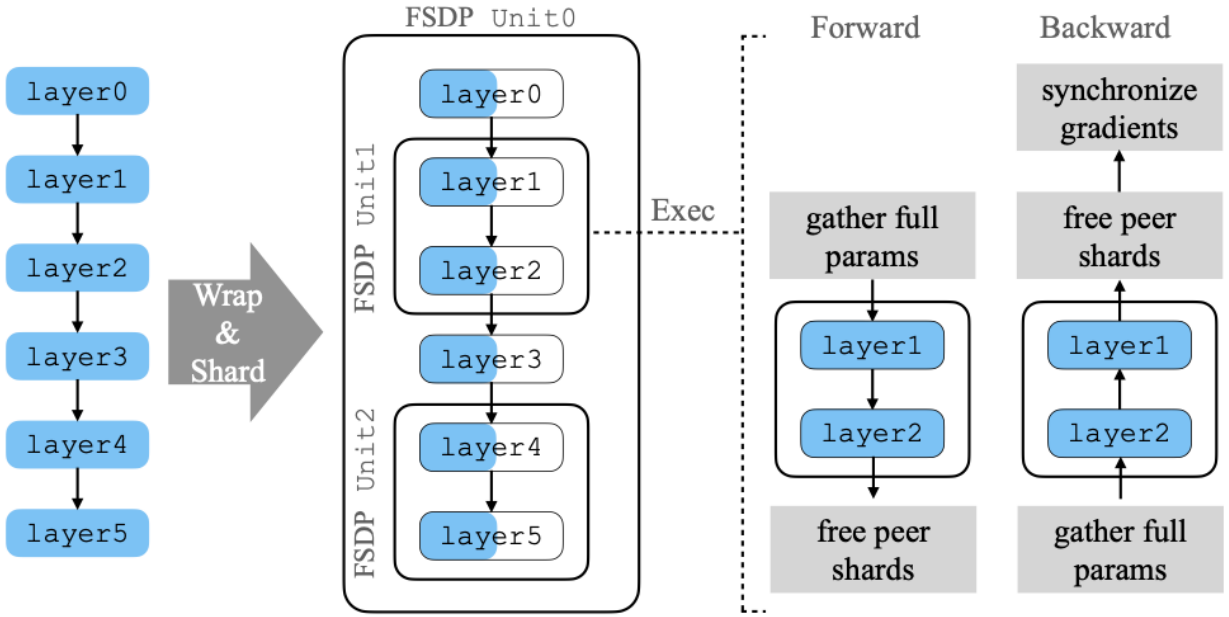
# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit




High-level sketch:

1. Divide model parameters into FSDP units

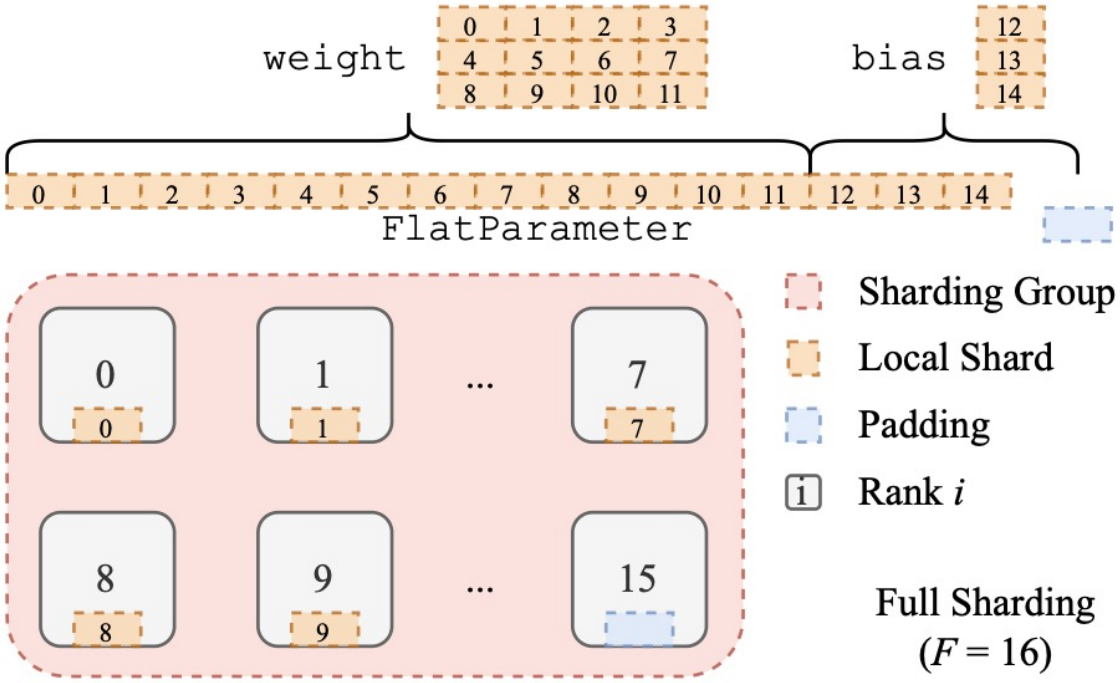
# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit




High-level sketch:

**1. Divide model parameters into FSDP units**

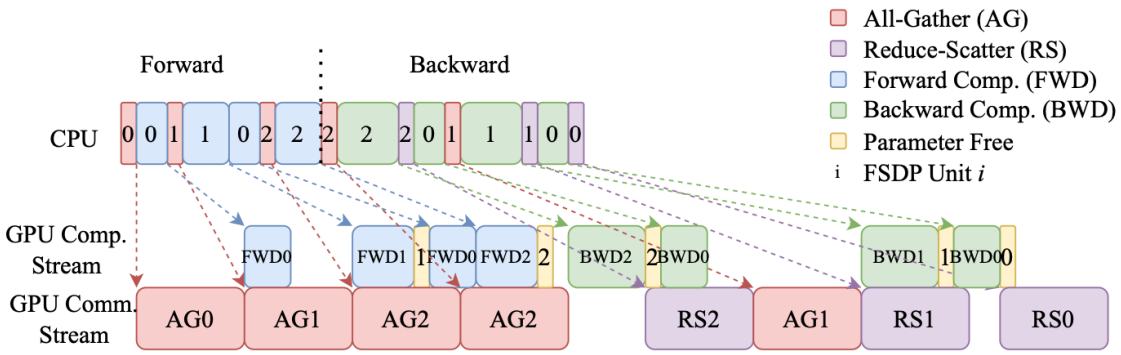
# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit

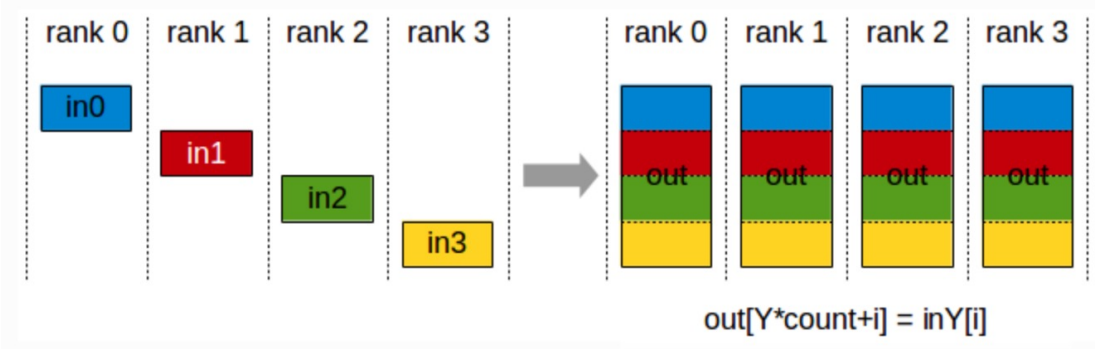



High-level sketch:

1. Divide model parameters into FSDP units
2. **Shard each unit across multiple GPUs**

# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit



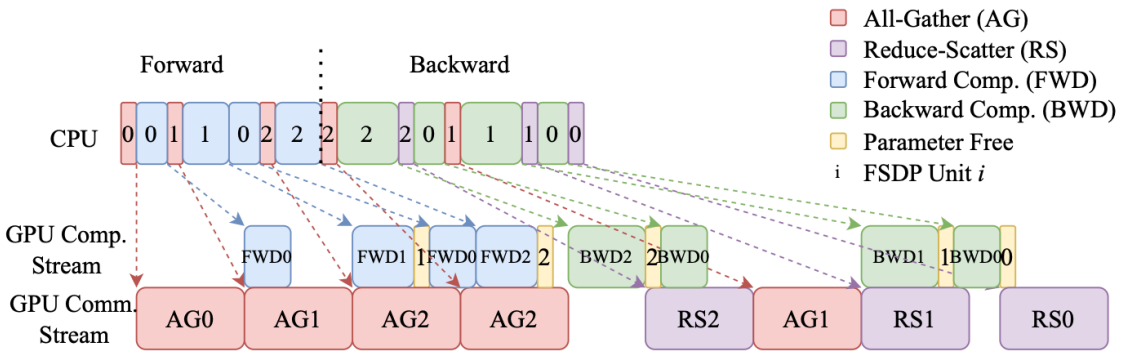



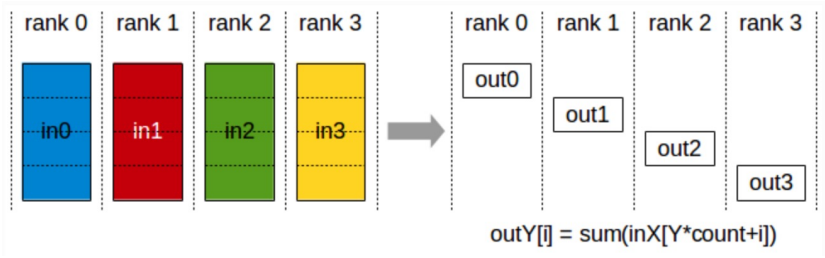

all-gather

High-level sketch:

1. Divide model parameters into FSDP units
2. Shard each unit across multiple GPUs
3. **Run forward pass:**
   - perform an all-gather so each GPU gets all pieces of a module.
   - Run forward pass
   - Discard param shards

# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit




High-level sketch:

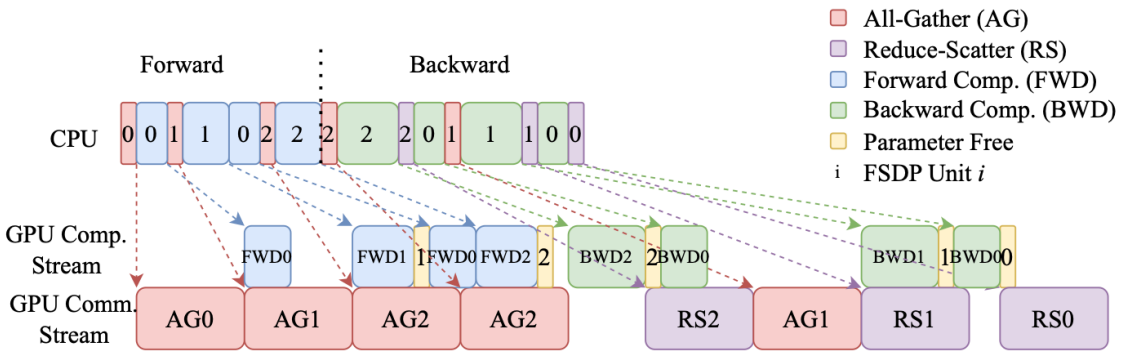
1. Divide model parameters into FSDP units
2. Shard each unit across multiple GPUs
3. Run forward pass

**4. Run backward pass:**

- perform an all-gather to get all pieces of module,
- Each GPU computes gradient for its data chunk
- Do a reduce-scatter to send full gradient piece to the right GPU

reduce-scatter

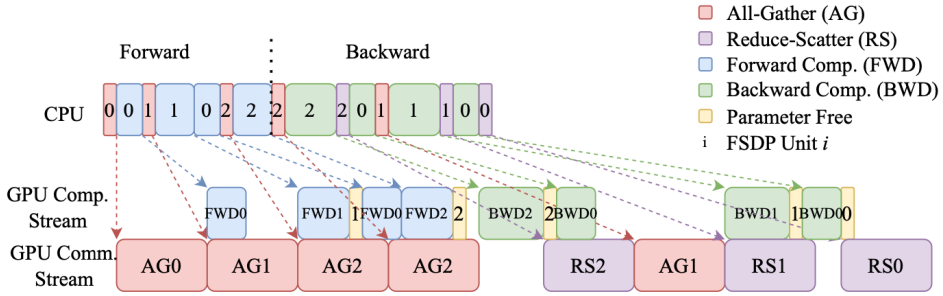
# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit

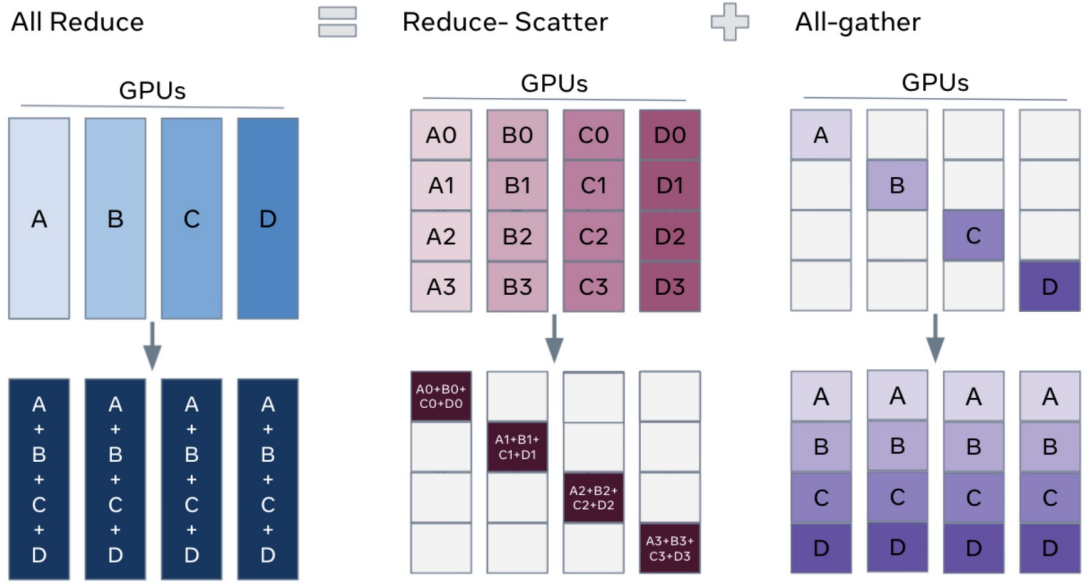



High-level sketch:

1. Divide model parameters into FSDP units
2. Shard each unit across multiple GPUs
3. Run forward pass
4. Run backward pass
5. **Each GPU updates its own shard using the full gradient received earlier.**

# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit

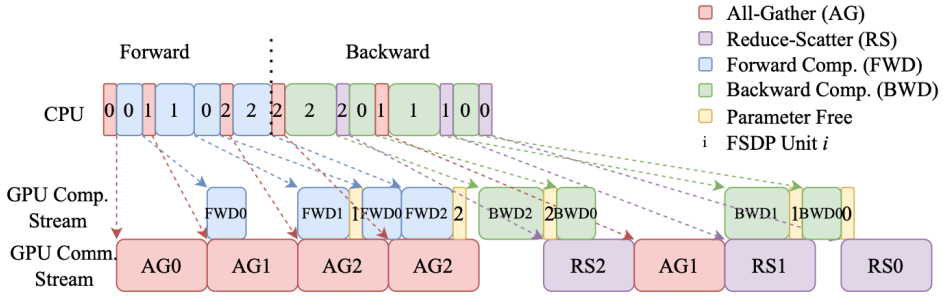



Communication overhead recap:

1. DDP: All reduce
2. ZeRO Stage-1 / 2: Reduce-Scatter + All-gather [memory saved for free]

# ZeRO Stage-3 (Full FSDP): When even the model parameters won't fit



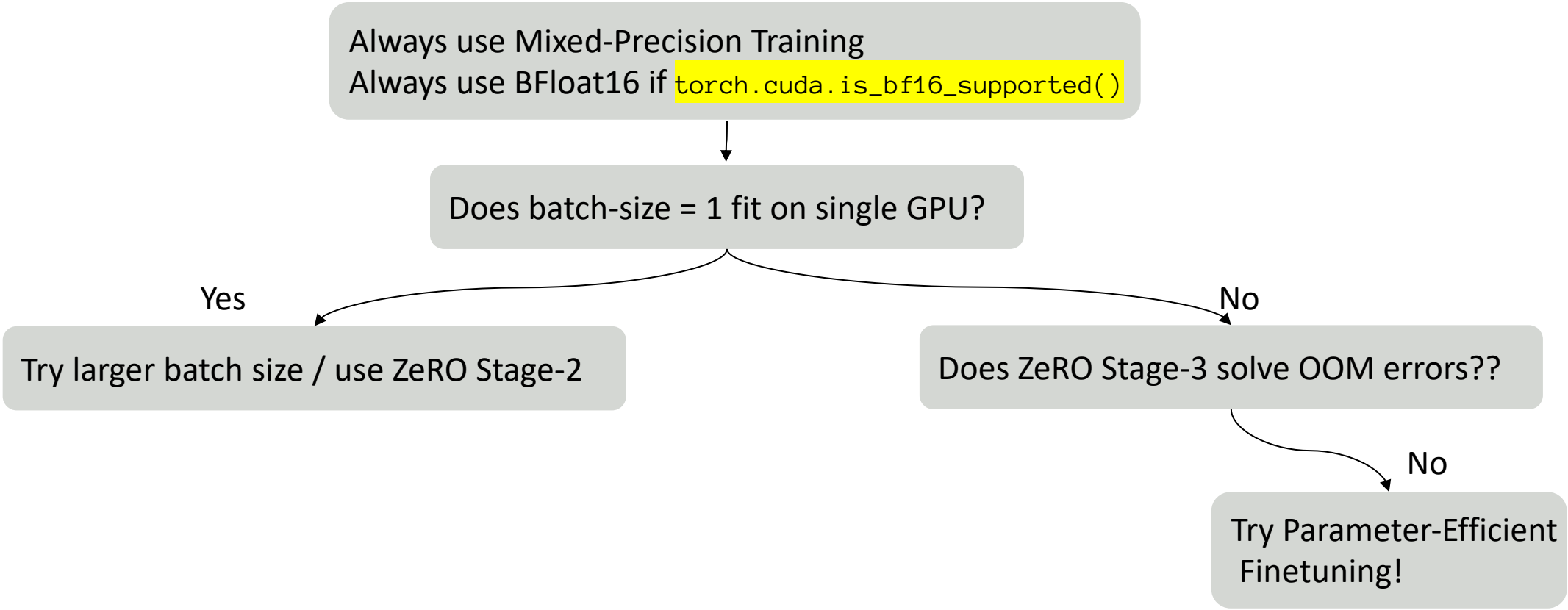

Communication overhead recap:

1. DDP: All reduce
2. ZeRO Stage-1 / 2: Reduce-Scatter + All-gather [memory saved for free]
3. ZeRO Stage-3: All-gather + Reduce-scatter + All-gather [More overhead!]

# Revisiting GPU memory calculation

GPU memory consumption :=
Model parameters (in FP16) +
Gradients (FP16) +
Master weights (FP32) +
Adam momentum (FP32) +
Adam variance (FP32) +
Model Activations (This scales with the batch size)!

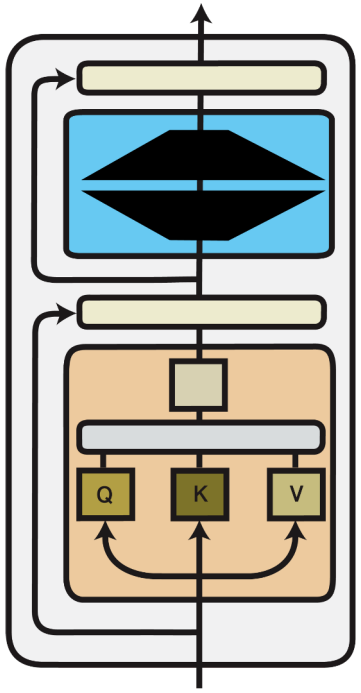
# Multi-GPU Training Optimizations Recap



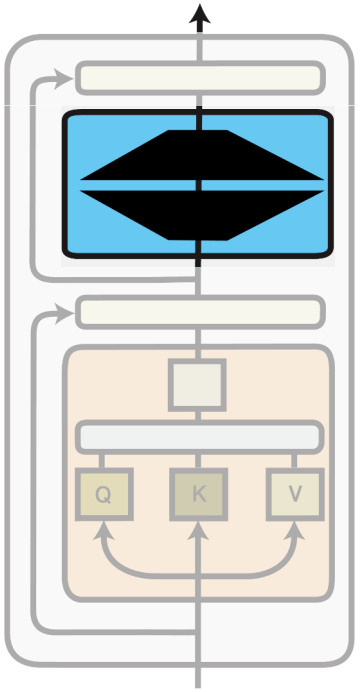
# Parameter-Efficient Finetuning

*Adapted from slides by Diyi Yang*

# From fine-tuning to parameter-efficient fine-tuning (PEFT)




Full Fine-tuning
Update **all model parameters**




Parameter-efficient Fine-tuning
Update a **small subset** of model parameters

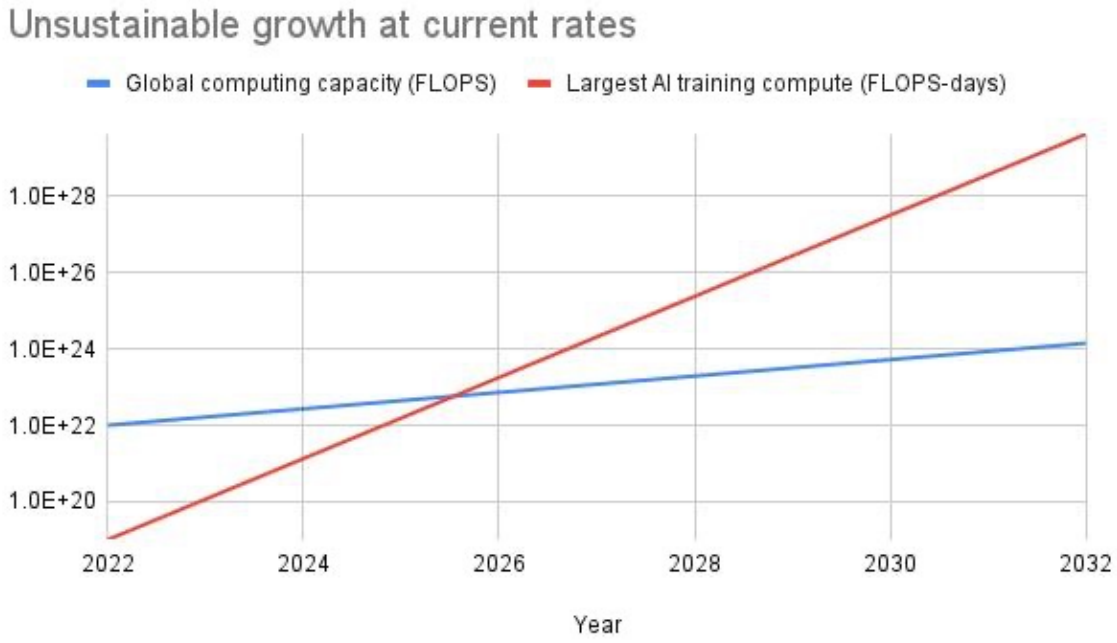
Why fine-tune *only some* parameters?

1. Fine-tuning all parameters is impractical with large models

2. State-of-the-art models are massively over-parameterized
   → Parameter-efficient fine-tuning matches performance of full fine-tuning

# Why do we need efficient adaptation?

- Exponential growth in maximum training compute for largest AI models (~2x every 3.4 months) vs. global compute capacity (~2x every 1.5 years)
- Clearly unsustainable rate of growth in AI computing scale, forecasted to slow a lot in the next few years.
- **As costs of training go up, AI development becomes concentrated in only the most well-funded organizations, especially in industry.**
    - Concentrating market power could lead to only a few dominant interests controlling a global technology – ***whose value systems*** are embedded in the AI of tomorrow?

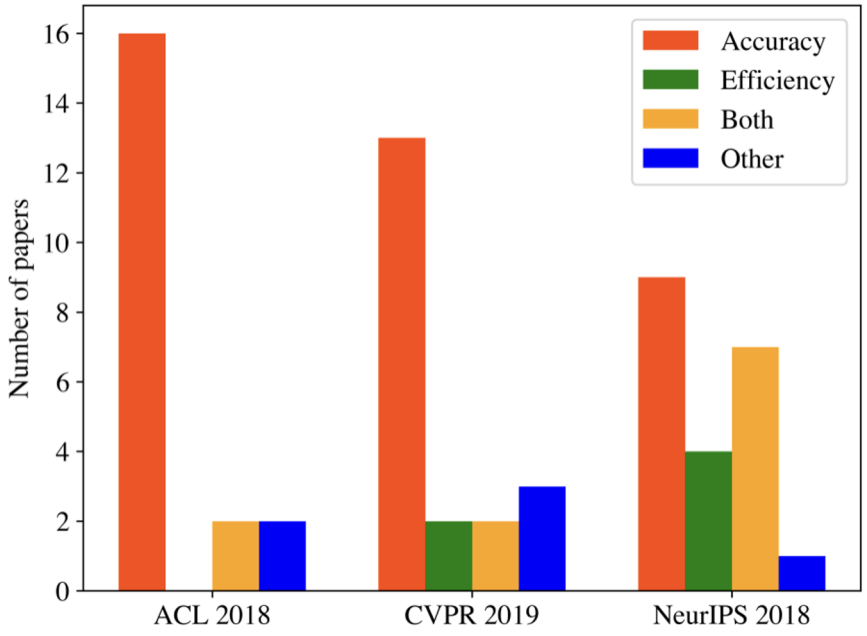



*Source: How much of AI progress is from scaling compute? And how far will it scale? (by 'jack', AI Progress Essay Contest)*

Slides credit to Benji Xie, Regina Wang and Pranav Gurusankar

# Accuracy vs Efficiency: What are we focusing on?

1. Emphasis on accuracy over efficiency in current AI paradigm
   - Is the tradeoff between efficiency and accuracy linear? It's quite that simple… [Ang et al., 2022]
2. **Hidden environmental costs of training (and fine tuning) LLMs**
   - *Most large players are non-transparent about the costs of training their models.*
   - *Cornell scientists in 2021 estimated that training GPT-3 was equivalent in carbon emissions to running a coal power plant for 10 straight hours.*




AI papers tend to target accuracy rather than efficiency. The figure shows the proportion of papers that target accuracy, efficiency, both or other from a sample of 60 papers from top AI conferences (Green AI)

Slides credit to Benji Xie, Regina Wang and Pranav Gurusankar

“At Stanford, for example, more than 200 students in a class on reinforcement learning were asked to implement common algorithms for a homework assignment. Though two of the algorithms performed equally well, one used far more power.

If all the students had used the more efficient algorithm, the researchers estimated they would have reduced their collective power consumption by 880 kilowatt-hours — **about what a typical American household uses in a month.”**

An example using CS234 in Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning.

Slides credit to Benji Xie, Regina Wang and Pranav Gurusankar

# Full Finetuning

- Assume we have a pre-trained autoregressive language model $P_\phi(y|x)$
  - E.g., GPT based on Transformer

- Adapt this pretrained model to downstream tasks (e.g., summarization, NL2SQL, reading comprehension)
  - Training dataset of context-target pairs $\{(x_i, y_i)\}_{i=1,\ldots,N}$

- During full fine-tuning, we update $\phi_o$ to $\phi_o + \Delta\phi$ by following the gradient to maximize the conditional language modeling objective

$$\max_{\phi} \sum_{(x,y)} \sum_{t=1}^{|y|} \log(P_\phi(y_t|x, y_{<t}))$$

# Full-Finetuning

- For each downstream task, we learn a different set of parameters $\Delta\phi$
  - $|\Delta\phi| = |\phi_o|$
  - GPT-3 has a $|\phi_o|$ of 175 billion
  - Expensive and challenging for storing and deploying many independent instances
- Can we do better?

# Full-Finetuning

- For each downstream task, we learn a different set of parameters $\Delta\phi$
  - $|\Delta\phi| = |\phi_o|$
  - GPT-3 has a $|\phi_o|$ of 175 billion
  - Expensive and challenging for storing and deploying many independent instances
- Can we do better?
- Key idea: encode the task-specific parameter increment $\Delta\phi = \Delta\phi(\Theta)$ by a smaller-sized set of parameters $\Theta, |\Theta| \ll |\phi_o|$

- The task of finding $\Delta\phi$ becomes optimizing over $\Theta$

$$\max_{\Theta} \sum_{(x,y)} \sum_{t=1}^{|y|} \log(P_{\phi_o+\Delta\phi(\Theta)}(y_t|x, y_{<t}))$$

# Low-rank-parameterized update matrices

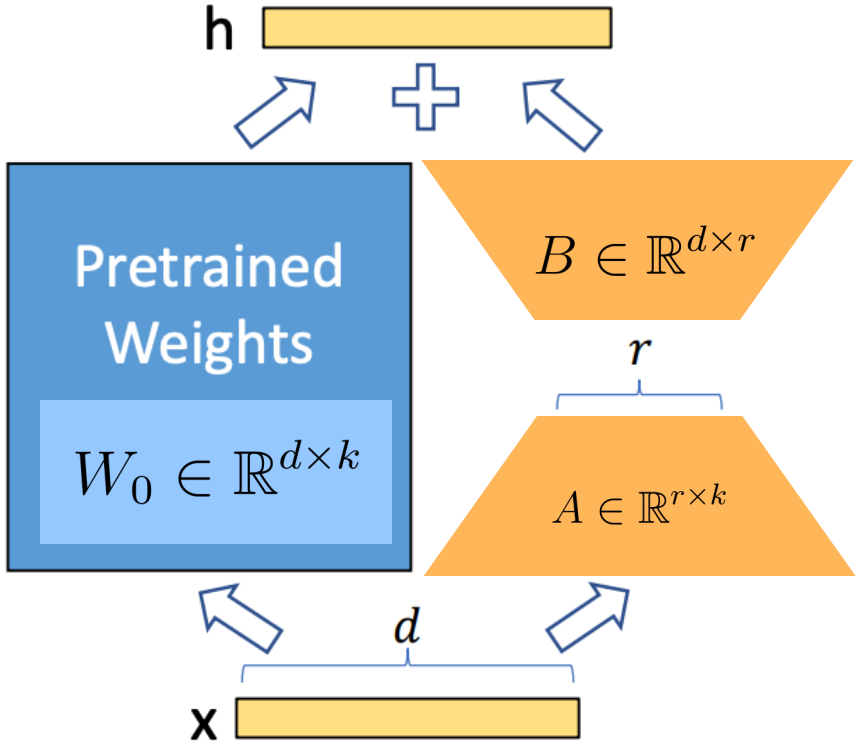
- Updates to the weights have a low "intrinsic rank" during adaptation (Aghajanyan et al. 2020)

- $W_0 \in \mathbb{R}^{d \times k}$: a pretrained weight matrix
- Constrain its update with a low-rank decomposition:

$$W_0 + \Delta W = W_0 + \alpha BA$$

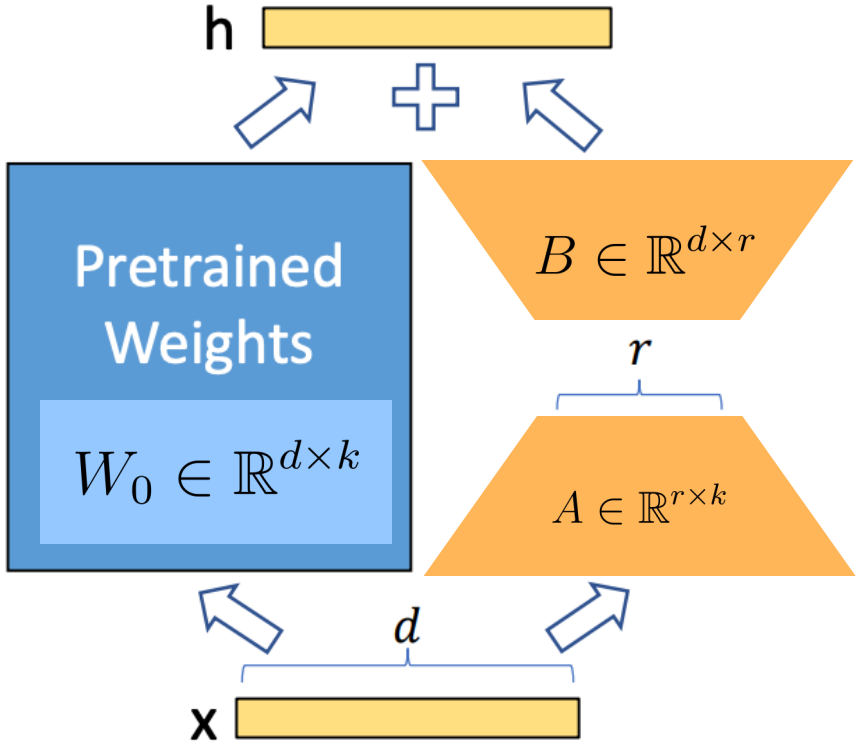
where $B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}, r \ll \min(d, k)$

- $\alpha$ is the tradeoff between pre-trained "knowledge" and task-specific "knowledge"
- Only A and B contain **trainable** parameters



# Low-rank-parameterized update matrices

- As one increase the number of trainable parameters, training LoRA converges to training the original model
- **No additional inference latency:** when switching to a different task, recover $W_0$ by subtracting $BA$ and adding a different $B'A'$
- Often LoRA is applied to the weight matrices in the self-attention module



# Low-rank-parameterized update matrices

```
input_dim = 768  # e.g., the hidden size of the pre-trained model
output_dim = 768  # e.g., the output size of the layer
rank = 8  # The rank 'r' for the low-rank adaptation

W = ... # from pretrained network with shape input_dim x output_dim

W_A = nn.Parameter(torch.empty(input_dim, rank)) # LoRA weight A
W_B = nn.Parameter(torch.empty(rank, output_dim)) # LoRA weight B

# Initialization of LoRA weights
nn.init.kaiming_uniform_(W_A, a=math.sqrt(5))
nn.init.zeros_(W_B)

def regular_forward_matmul(x, W):
    h = x @ W
return h

def lora_forward_matmul(x, W, W_A, W_B):
    h = x @ W  # regular matrix multiplication
    h += x @ (W_A @ W_B)*alpha # use scaled LoRA weights
return h
```

Copy

Source: https://lightning.ai/pages/community/article/lora-llm/

# LoRA in practice

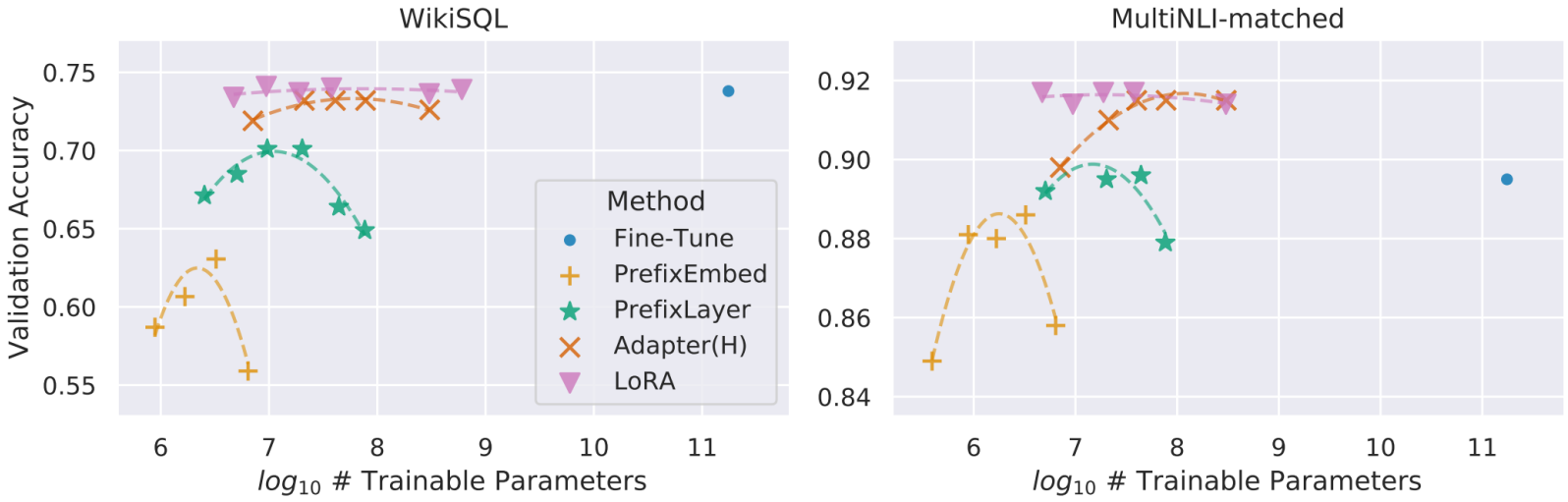
| Model & Method | # Trainable Parameters | E2E NLG Challenge BLEU | NIST | MET | ROUGE-L | CIDEr |
|---|---|---|---|---|---|---|
| GPT-2 M (FT)* | 354.92M | 68.2 | 8.62 | 46.2 | 71.0 | 2.47 |
| GPT-2 M (Adapter$^{L}$)* | 0.37M | 66.3 | 8.41 | 45.0 | 69.8 | 2.40 |
| GPT-2 M (Adapter$^{L}$)* | 11.09M | 68.9 | 8.71 | 46.1 | 71.3 | 2.47 |
| GPT-2 M (Adapter$^{H}$) | 11.09M | $67.3_{\pm.6}$ | $8.50_{\pm.07}$ | $46.0_{\pm.2}$ | $70.7_{\pm.2}$ | $2.44_{\pm.01}$ |
| GPT-2 M (FT$^{Top2}$)* | 25.19M | 68.1 | 8.59 | 46.0 | 70.8 | 2.41 |
| GPT-2 M (PreLayer)* | 0.35M | 69.7 | 8.81 | 46.1 | 71.4 | 2.49 |
| GPT-2 M (LoRA) | 0.35M | $\mathbf{70.4}_{\pm.1}$ | $\mathbf{8.85}_{\pm.02}$ | $\mathbf{46.8}_{\pm.2}$ | $\mathbf{71.8}_{\pm.1}$ | $\mathbf{2.53}_{\pm.02}$ |
| GPT-2 L (FT)* | 774.03M | 68.5 | 8.78 | 46.0 | 69.9 | 2.45 |
| GPT-2 L (Adapter$^{L}$) | 0.88M | $69.1_{\pm.1}$ | $8.68_{\pm.03}$ | $46.3_{\pm.0}$ | $71.4_{\pm.2}$ | $\mathbf{2.49}_{\pm.0}$ |
| GPT-2 L (Adapter$^{L}$) | 23.00M | $68.9_{\pm.3}$ | $8.70_{\pm.04}$ | $46.1_{\pm.1}$ | $71.3_{\pm.2}$ | $2.45_{\pm.02}$ |
| GPT-2 L (PreLayer)* | 0.77M | 70.3 | 8.85 | 46.2 | 71.7 | 2.47 |
| GPT-2 L (LoRA) | 0.77M | $\mathbf{70.4}_{\pm.1}$ | $\mathbf{8.89}_{\pm.02}$ | $\mathbf{46.8}_{\pm.2}$ | $\mathbf{72.0}_{\pm.2}$ | $2.47_{\pm.02}$ |

GPT-2 medium (M) and large (L) with different adaptation methods on the E2E NLG Challenge. For all metrics, higher is better. LoRA outperforms several baselines with comparable or fewer trainable parameters

Hu, Edward J., Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. "Lora: Low-rank adaptation of large language models." *arXiv preprint arXiv:2106.09685* (2021).

# LoRA in practice

| Model&Method | # Trainable Parameters | WikiSQL Acc. (%) | MNLI-m Acc. (%) | SAMSum R1/R2/RL |
|---|---|---|---|---|
| GPT-3 (FT) | 175,255.8M | **73.8** | 89.5 | 52.0/28.0/44.5 |
| GPT-3 (BitFit) | 14.2M | 71.3 | 91.0 | 51.3/27.4/43.5 |
| GPT-3 (PreEmbed) | 3.2M | 63.1 | 88.6 | 48.3/24.2/40.5 |
| GPT-3 (PreLayer) | 20.2M | 70.1 | 89.5 | 50.8/27.3/43.5 |
| GPT-3 ($\text{Adapter}^{\text{H}}$) | 7.1M | 71.9 | 89.8 | 53.0/28.9/44.8 |
| GPT-3 ($\text{Adapter}^{\text{H}}$) | 40.1M | 73.2 | **91.5** | 53.2/29.0/45.1 |
| GPT-3 (LoRA) | 4.7M | 73.4 | **91.7** | **53.8/29.8/45.9** |
| GPT-3 (LoRA) | 37.7M | **74.0** | **91.6** | 53.4/29.2/45.1 |




LoRA matches or exceeds the fine-tuning baseline on all three datasets

LoRA exhibits better scalability and task performance.

# Understanding low-rank adaptation

## Which weight matrices in Transformers should we apply LoRA to?

| | # of Trainable Parameters = 18M | | | | | | |
|---|---|---|---|---|---|---|---|
| Weight Type | $W_q$ | $W_k$ | $W_v$ | $W_o$ | $W_q, W_k$ | $W_q, W_v$ | $W_q, W_k, W_v, W_o$ |
| Rank $r$ | 8 | 8 | 8 | 8 | 4 | 4 | 2 |
| WikiSQL ($\pm$0.5%) | 70.4 | 70.0 | 73.0 | 73.2 | 71.4 | **73.7** | **73.7** |
| MultiNLI ($\pm$0.1%) | 91.0 | 90.8 | 91.0 | 91.3 | 91.3 | 91.3 | **91.7** |

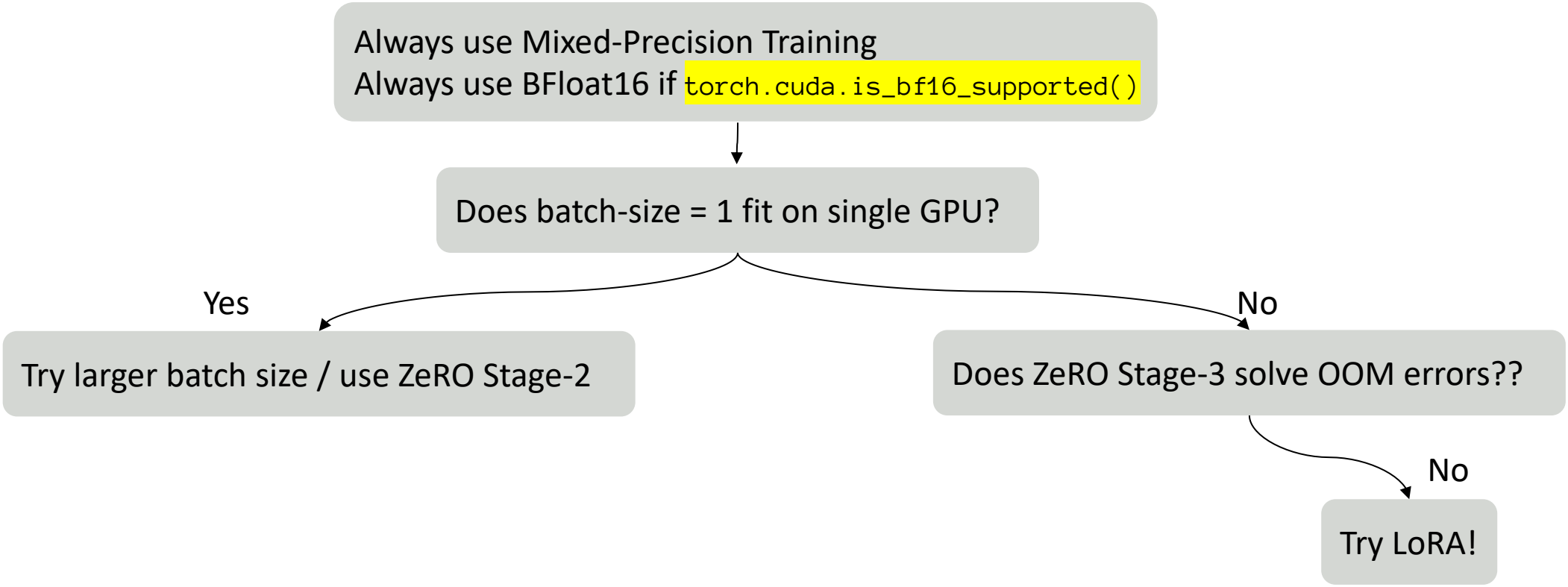
Adapting both Wq and Wv gives the best performance overall.

## What is the optimal rank $r$ for LoRA?

| | Weight Type | $r = 1$ | $r = 2$ | $r = 4$ | $r = 8$ | $r = 64$ |
|---|---|---|---|---|---|---|
| WikiSQL($\pm$0.5%) | $W_q$ | 68.8 | 69.6 | 70.5 | 70.4 | 70.0 |
| | $W_q, W_v$ | 73.4 | 73.3 | 73.7 | 73.8 | 73.5 |
| | $W_q, W_k, W_v, W_o$ | 74.1 | 73.7 | 74.0 | 74.0 | 73.9 |
| MultiNLI ($\pm$0.1%) | $W_q$ | 90.7 | 90.9 | 91.1 | 90.7 | 90.7 |
| | $W_q, W_v$ | 91.3 | 91.4 | 91.3 | 91.6 | 91.4 |
| | $W_q, W_k, W_v, W_o$ | 91.2 | 91.7 | 91.7 | 91.5 | 91.4 |

LoRA already performs competitively with a very small $r$

# Summarizing everything:




**Even simple: start with Llama 7B + bfloat16 + ZeRO Stage-3 (or FSDP) + LoRA** 👈