DeepSpeed Under the Hood: Revolutionising AI with Large-Scale Model Training

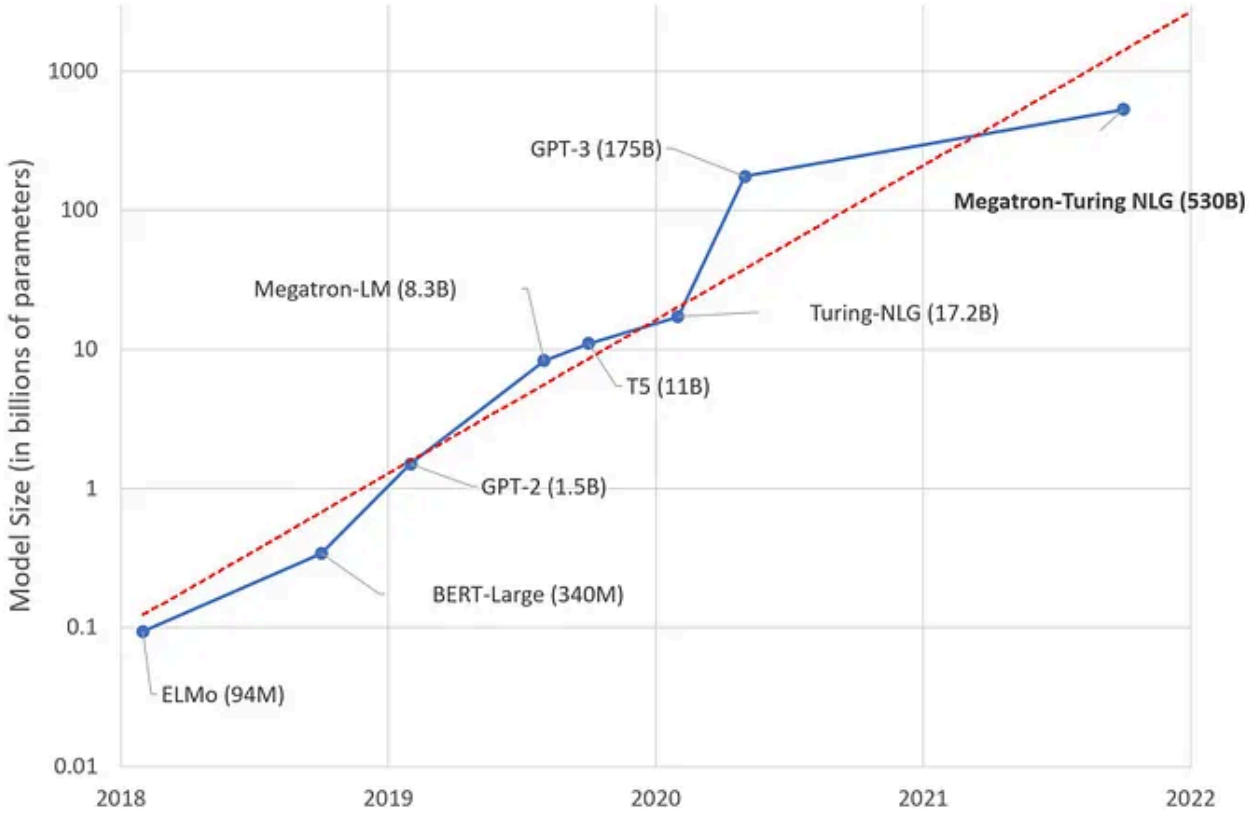
Srikaran · Follow
15 min read · Apr 9, 2024

DeepSpeed, a groundbreaking deep learning optimization library from Microsoft, is reshaping the future of AI. Designed to efficiently train massive, complex models, DeepSpeed handles billions of parameters with unprecedented efficiency. This blog post delves into the intricate workings of DeepSpeed and its impact on the world of deep learning.

Introduction to DeepSpeed

DeepSpeed is an open-source deep learning optimization library tailored for PyTorch. It enhances training efficiency and effectiveness while reducing computational resources and memory usage. This is achieved through innovative techniques in model parallelism, memory optimization, and communication efficiency. Challenges of training large deep learning models

Large models offer significant accuracy gains, as you see from the below graph, the number of parameters in the models is exponentially increasing.



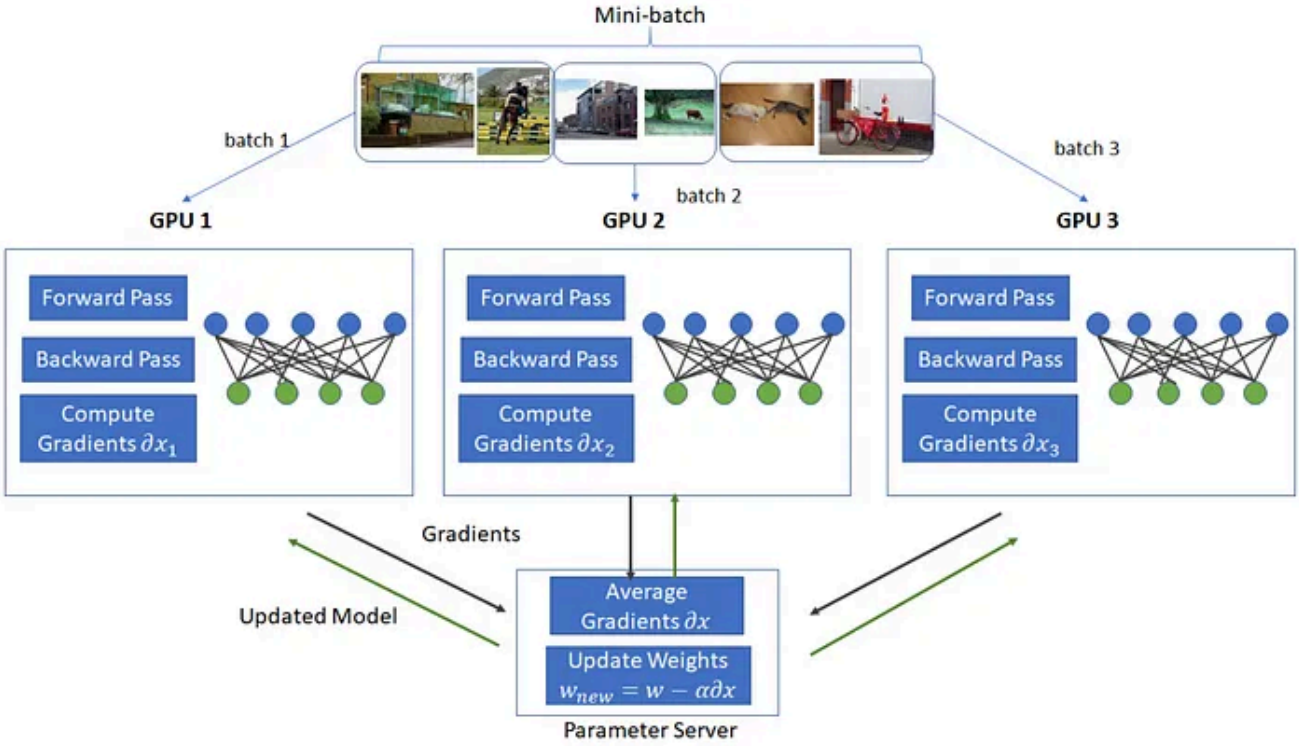
Challenges of Training Large Deep Learning Models

Training models with billions to trillions of parameters often hits hardware limitations. Existing solutions, like data parallelism, don't reduce per-device memory footprint, and model parallelism doesn't scale efficiently beyond a single node. For instance, NVIDIA's Megatron-LM, while setting a record with 8.3 billion parameters, faces performance degradation when scaling across nodes.

Lets try to understand data parallelism and model parallelism is detail in order to get the context now!!

Data Parallelism

Most users with just a few GPUs are likely to be familiar with `DistributedDataParallel` (DDP) PyTorch documentation. In this method the model is fully replicated to each GPU and then after each iteration all the models synchronize their states with each other. This approach allows training speed up but throwing more resources at the problem, but it only works if the model can fit onto a single GPU.



Tensor Parallelism

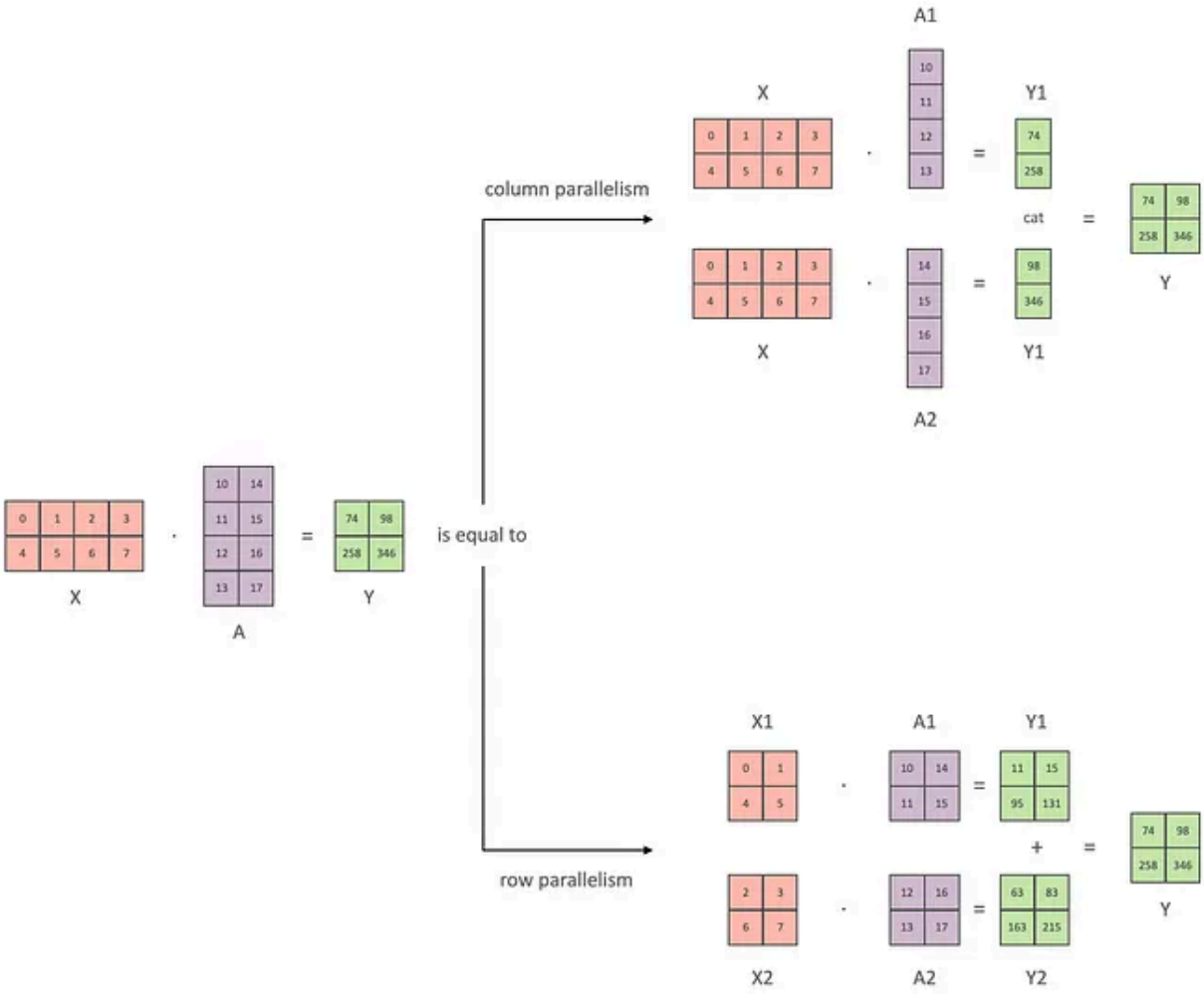
In Tensor Parallelism (TP) each GPU processes only a slice of a tensor and only aggregates the full tensor for operations that require the whole thing.

In this section we use concepts and diagrams from the Megatron-LM paper: Efficient Large-Scale Language Model Training on GPU Clusters.

The main building block of any transformer is a fully connected `nn.Linear` followed by a nonlinear activation `GeLU`.

Following the Megatron paper's notation, we can write the dot-product part of it as `Y = GeLU(XA)`, where `X` and `Y` are the input and output vectors, and `A` is the weight matrix.

If we look at the computation in matrix form, it's easy to see how the matrix multiplication can be split between multiple GPUs:

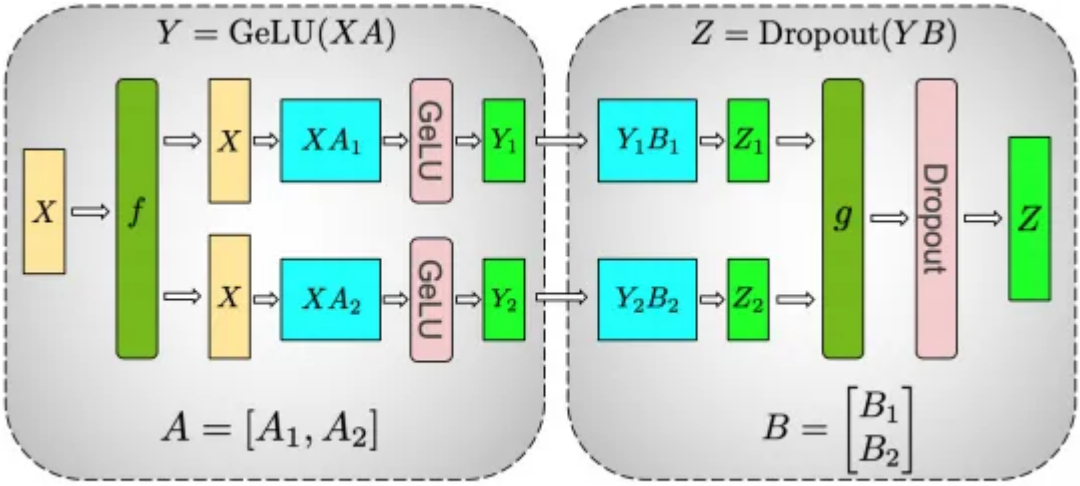


If we split the weight matrix `A` column-wise across `N` GPUs and perform matrix multiplications `XA_1` through `XA_n` in parallel, then we will end up with `N` output vectors `Y_1, Y_2, ..., Y_n` which can be fed into `GeLU` independently:

$$[Y_1, Y_2] = [\text{GeLU}(XA_1), \text{GeLU}(XA_2)]$$

Notice with the Y matrix split along the columns, we can split the second GEMM along its rows so that it takes the output of the GeLU directly without any extra communication.

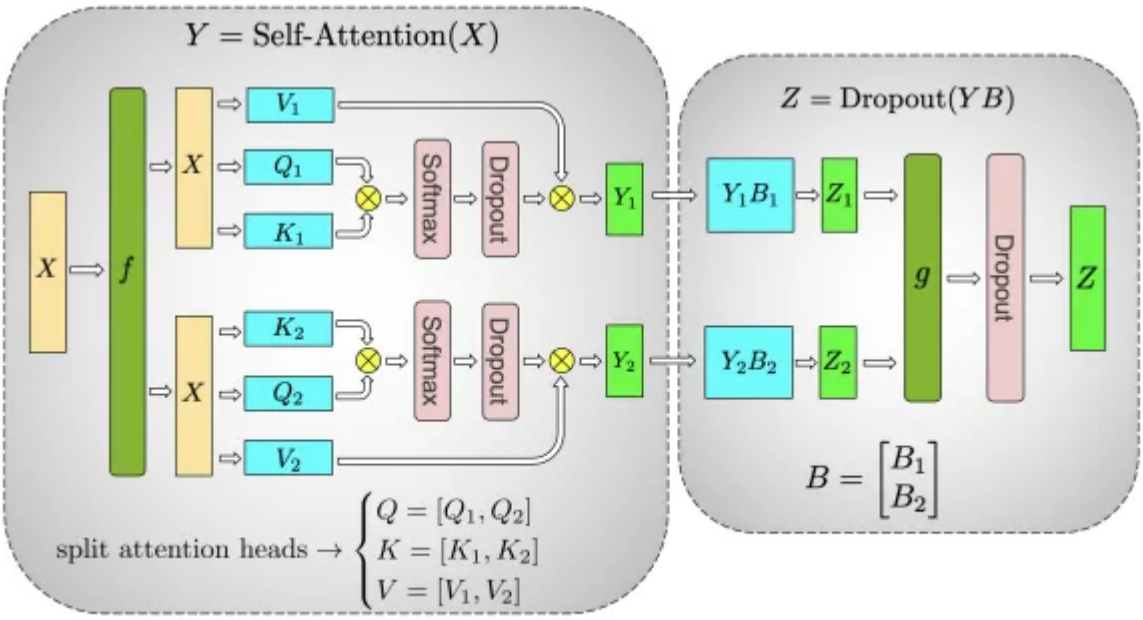
Using this principle, we can update an MLP of arbitrary depth, while synchronising the GPUs after each row-column sequence. The Megatron-LM paper authors provide a helpful illustration for that:



(a) MLP

Here `f` is an identity operator in the forward pass and all reduce in the backward pass while `g` is an all reduce in the forward pass and identity in the backward pass.

Parallelizing the multi-headed attention layers is even simpler, since they are already inherently parallel, due to having multiple independent heads!



(b) Self-Attention

Special considerations: Due to the two all reduces per layer in both the forward and backward passes, TP requires a very fast interconnect between devices. Therefore it's not advisable to do TP across more than one node, unless you have a very fast network. In our case the inter-node was much slower than PCIe. Practically, if a node has 4 GPUs, the highest TP degree is therefore 4. If you need a TP degree of 8, you need to use nodes that have at least 8 GPUs.

This component is implemented by Megatron-LM. Megatron-LM has recently expanded tensor parallelism to include sequence parallelism that splits the operations that cannot be split as above, such as LayerNorm, along the sequence dimension. The paper Reducing Activation Recomputation in Large Transformer Models provides details for this technique.

Pipeline Parallelism

Naive Pipeline Parallelism (naive PP) is where one spreads groups of model layers across multiple GPUs and simply moves data along from GPU to GPU as if it were one large composite GPU. The mechanism is relatively simple — switch the desired layers `.to()` the desired devices and now whenever the data goes in and out those layers switch the data to the same device as the layer and leave the rest unmodified.

This performs a vertical model parallelism, because if you remember how most models are drawn, we slice the layers vertically. For example, if the following diagram shows an 8-layer model:

```
===================  ===================
|  0 | 1 | 2 | 3  |  |  4 | 5 | 6 | 7  |
===================  ===================
        GPU0                 GPU1
```

we just sliced it in 2 vertically, placing layers 0–3 onto GPU0 and 4–7 to GPU1.

Now while data travels from layer 0 to 1, 1 to 2 and 2 to 3 this is just like the forward pass of a normal model on a single GPU. But when data needs to pass from layer 3 to layer 4 it needs to travel from GPU0 to GPU1 which introduces a communication overhead. If the participating GPUs are on the same compute node (e.g. same physical machine) this copying is pretty fast, but if the GPUs are located on different compute nodes (e.g. multiple machines) the communication overhead could be significantly larger.

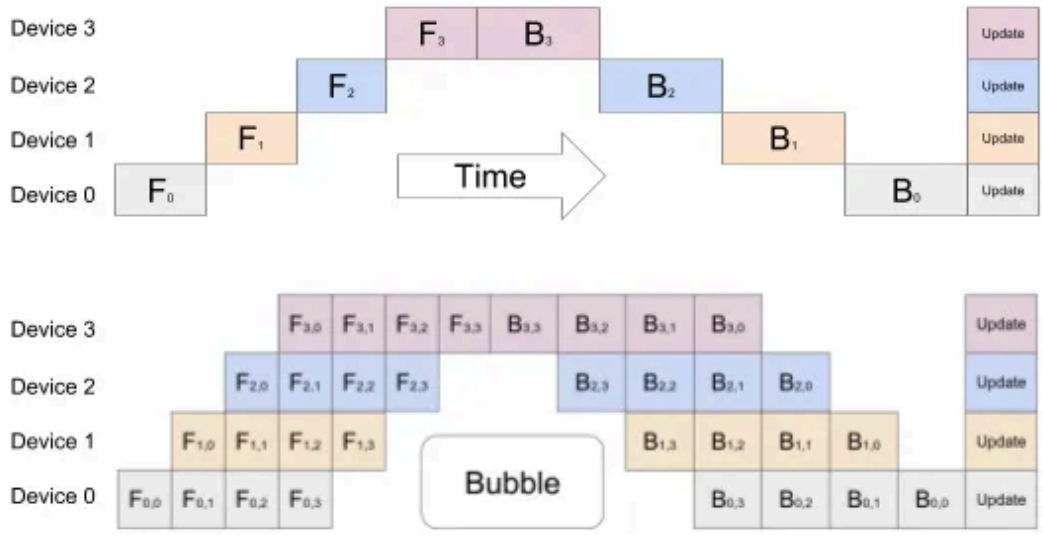
Then layers 4 to 5 to 6 to 7 are as a normal model would have and when the 7th layer completes we often need to send the data back to layer 0 where the labels are (or alternatively send the labels to the last layer). Now the loss can be computed and the optimizer can do its work.

Problems:

- the main deficiency and why this one is called “naive” PP, is that all but one GPU is idle at any given moment. So if 4 GPUs are used, it’s almost identical to quadrupling the amount of memory of a single GPU, and ignoring the rest of the hardware. Plus there is the overhead of copying the data between devices. So 4x 6GB cards will be able to accommodate the same size as 1x 24GB card using naive PP, except the latter will complete the training faster, since it doesn’t have the data copying overhead. But, say, if you have 40GB cards and need to fit a 45GB model you can with 4x 40GB cards (but barely because of the gradient and optimizer states).
- shared embeddings may need to get copied back and forth between GPUs.

Pipeline Parallelism (PP) is almost identical to a naive PP described above, but it solves the GPU idling problem, by chunking the incoming batch into micro-batches and artificially creating a pipeline, which allows different GPUs to concurrently participate in the computation process.

The following illustration from the GPipe paper shows the naive PP on the top, and PP on the bottom:



Top: The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one accelerator is active at a time. Bottom: GPipe divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on separate micro-batches at the same time.

It's easy to see from the bottom diagram how PP has fewer dead zones, where GPUs are idle. The idle parts are referred to as the "bubble".

Both parts of the diagram show parallelism that is of degree 4. That is 4 GPUs are participating in the pipeline. So there is the forward path of 4 pipe stages F0, F1, F2 and F3 and then the return reverse order backward path of B3, B2, B1 and B0.

PP introduces a new hyper-parameter to tune that is called `chunks`. It defines how many chunks of data are sent in a sequence through the same pipe stage. For example, in the bottom diagram, you can see that `chunks=4`. GPU0 performs the same forward path on chunk 0, 1, 2 and 3 (F0,0, F0,1, F0,2, F0,3) and then it waits for other GPUs to do their work and only when their work is starting to be complete, does GPU0 start to work again doing the backward path for chunks 3, 2, 1 and 0 (B0,3, B0,2, B0,1, B0,0).

Note that conceptually this is the same concept as gradient accumulation steps (GAS). PyTorch uses `chunks`, whereas DeepSpeed refers to the same hyper-parameter as GAS.

Because of the chunks, PP introduces the concept of micro-batches (MBS). DP splits the global data batch size into mini-batches, so if you have a DP degree of 4, a global batch size of 1024 gets split up into 4 mini-batches of 256 each (1024/4). And if the number of `chunks` (or GAS) is 32 we end up with a micro-batch size of 8 (256/32). Each Pipeline stage works with a single micro-batch at a time.

To calculate the global batch size of the DP + PP setup we then do: `mbs*chunks*dp_degree` (`8*32*4=1024`).

Let's go back to the diagram.

With `chunks=1` you end up with the naive PP, which is very inefficient. With a very large `chunks` value you end up with tiny micro-batch sizes which could be not very efficient either. So one has to experiment to find the value that leads to the highest efficient utilization of the GPUs.

While the diagram shows that there is a bubble of "dead" time that can't be parallelized because the last `forward` stage has to wait for `backward` to complete the pipeline, the purpose of finding the best value for `chunks` is to enable a high

concurrent GPU utilization across all participating GPUs which translates to minimizing the size of the bubble.

This scheduling mechanism is known as `all forward all backward`. Some other alternatives are one forward one backward and interleaved one forward one backward.

While both Megatron-LM and DeepSpeed have their own implementation of the PP protocol, Megatron-DeepSpeed uses the DeepSpeed implementation as it's integrated with other aspects of DeepSpeed.

One other important issue here is the size of the word embedding matrix. While normally a word embedding matrix consumes less memory than the transformer block, in our case with a huge 250k vocabulary, the embedding layer needed 7.2GB in bf16 weights and the transformer block is just 4.9GB. Therefore, we had to instruct Megatron-Deepspeed to consider the embedding layer as a transformer block. So we had a pipeline of 72 layers, 2 of which were dedicated to the embedding (first and last). This allowed to balance out the GPU memory consumption. If we didn't do it, we would have had the first and the last stages consume most of the GPU memory, and 95% of GPUs would be using much less memory and thus the training would be far from being efficient.

Overcoming limitations of data parallelism and model parallelism with ZeRO

ZeRO(Zero Redundancy Optimizer) was developed to conquer the limitations of data parallelism and model parallelism while achieving the merits of both. ZeRO removes the memory redundancies across data-parallel processes by partitioning the model states — parameters, gradients, and optimizer state — across data parallel processes instead of replicating them. It uses a dynamic communication schedule during training to share the necessary state across distributed devices to retain the computational granularity and communication volume of data parallelism.

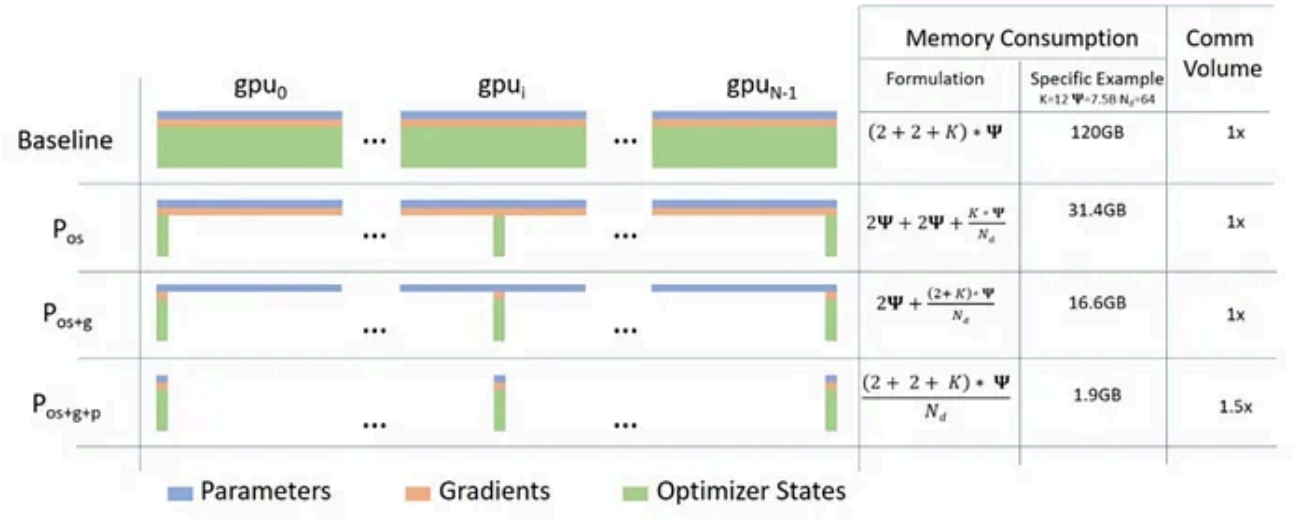
1. **Memory Efficiency and Data Parallelism:**

- **ZeRO** leverages the aggregate computation and memory resources of **data parallelism** to reduce the memory and compute requirements of each device (GPU) used for model training.

- It achieves this by partitioning various model training states (weights, gradients, and optimizer states) across available devices (GPUs and CPUs) in the distributed training hardware.
- The key appeal of ZeRO is that **no model code modifications are required**.
- ZeRO is implemented as incremental stages of optimizations, where optimizations in earlier stages are available in the later stages 2.

1. **Stages of ZeRO Optimization**:

- **Stage 1**:
- Optimizer states (e.g., for Adam optimizer, 32-bit weights, and the first and second moment estimates) are partitioned across processes. Each process updates only its partition.
- **Stage 2**:
- Reduced 32-bit gradients for updating the model weights are also partitioned. Each process retains only the gradients corresponding to its portion of the optimizer states.
- **Stage 3**:
- The 16-bit model parameters are partitioned across processes. ZeRO-3 automatically collects and partitions them during forward and backward passes.
- ZeRO-3 also includes the **infinity offload engine** to form **ZeRO-Infinity**, which can offload to both CPU and NVMe memory for huge memory savings 2.



	Memory Consumption		Comm Volume
	Formulation	Specific Example (K=12 Ψ=7.5B N_d=64)	
Baseline	$(2+2+K)*\Psi$	120GB	1x
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

Figure 1: Memory savings and communication volume for the three stages of ZeRO compared with standard data parallel baseline. In the memory consumption formula, Ψ refers to the number of parameters in a model and *K* is the optimizer specific constant term. As a specific example, we show the memory consumption for a 7.5B parameter model using Adam optimizer where *K*=12 on 64 GPUs. We also show the communication volume of ZeRO relative to the baseline.

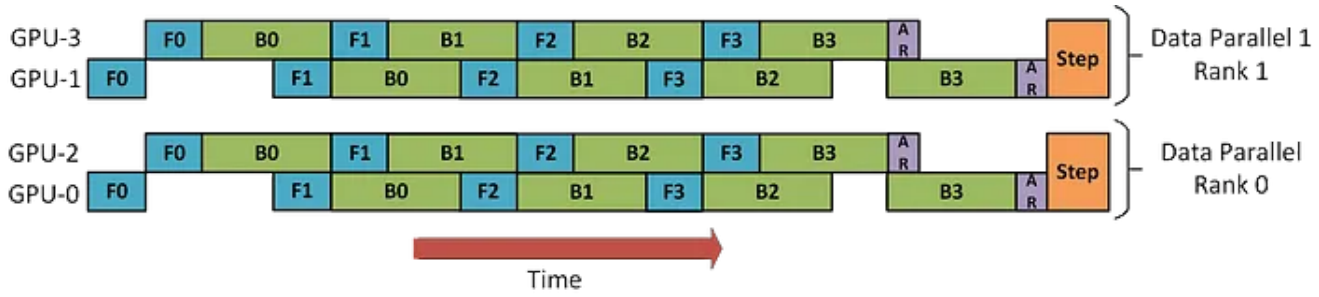
For more information, you can refer to this video.

3D Parallelism in DeepSpeed

3D Parallelism in DeepSpeed is an innovative approach that integrates three parallelization strategies to efficiently train extremely large AI models. It combines ZeRO-powered data parallelism, which replicates the model across multiple GPUs and processes different data slices in parallel, with pipeline parallelism, where model layers are divided into stages for parallel processing, enhancing memory and compute efficiency. Additionally, tensor-slicing model parallelism slices tensors along specific dimensions, distributing computation and reducing memory usage across devices. This triad of parallelism techniques allows DeepSpeed to adapt to varying workload requirements, achieving near-perfect memory and throughput scaling. As a result, it enables the training of models with over a trillion parameters, marking a significant advancement in the capabilities of large-scale AI model training.

DP+PP

The following diagram from the DeepSpeed pipeline tutorial demonstrates how one combines DP with PP.

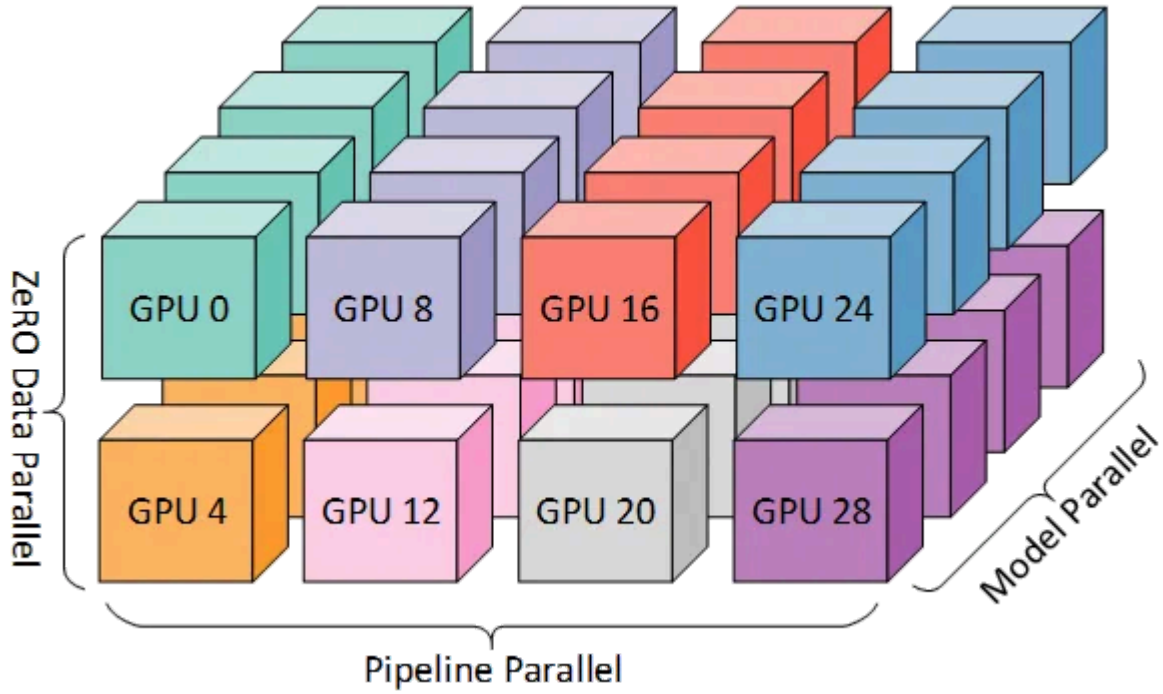


Here it's important to see how DP rank 0 doesn't see GPU2 and DP rank 1 doesn't see GPU3. To DP there are just GPUs 0 and 1 where it feeds data as if there were just 2 GPUs. GPU0 "secretly" offloads some of its load to GPU2 using PP. And GPU1 does the same by enlisting GPU3 to its aid.

Since each dimension requires at least 2 GPUs, here you'd need at least 4 GPUs.

DP+PP+TP

To get an even more efficient training PP is combined with TP and DP which is called 3D parallelism. This can be seen in the following diagram.



This diagram is from a blog post 3D parallelism: Scaling to trillion-parameter models, which is a good read as well.

Since each dimension requires at least 2 GPUs, here you'd need at least 8 GPUs for full 3D parallelism.

ZeRO DP+PP+TP

One of the main features of DeepSpeed is ZeRO, which is a super-scalable extension of DP. It has already been discussed in ZeRO Data Parallelism. Normally it's a standalone feature that doesn't require PP or TP. But it can be combined with PP and TP.

When ZeRO-DP is combined with PP (and optionally TP) it typically enables only ZeRO stage 1, which shards only optimizer states. ZeRO stage 2 additionally shards gradients, and stage 3 also shards the model weights.

While it's theoretically possible to use ZeRO stage 2 with Pipeline Parallelism, it will have bad performance impacts. There would need to be an additional reduce-scatter collective for every micro-batch to aggregate the gradients before sharding, which adds a potentially significant communication overhead. By nature of Pipeline Parallelism, small micro-batches are used and instead the focus is on trying to balance arithmetic intensity (micro-batch size) with minimizing the Pipeline bubble (number of micro-batches). Therefore those communication costs are going to hurt.

In addition, there are already fewer layers than normal due to PP and so the memory savings won't be huge. PP already reduces gradient size by `1/PP`, and so gradient sharding savings on top of that are less significant than pure DP.

ZeRO stage 3 can also be used to train models at this scale, however, it requires more communication than the DeepSpeed 3D parallel implementation. After careful evaluation it was found Megatron-DeepSpeed 3D parallelism performed best. Since then ZeRO stage 3 performance has dramatically improved and if we were to evaluate it today perhaps we would have chosen stage 3 instead.

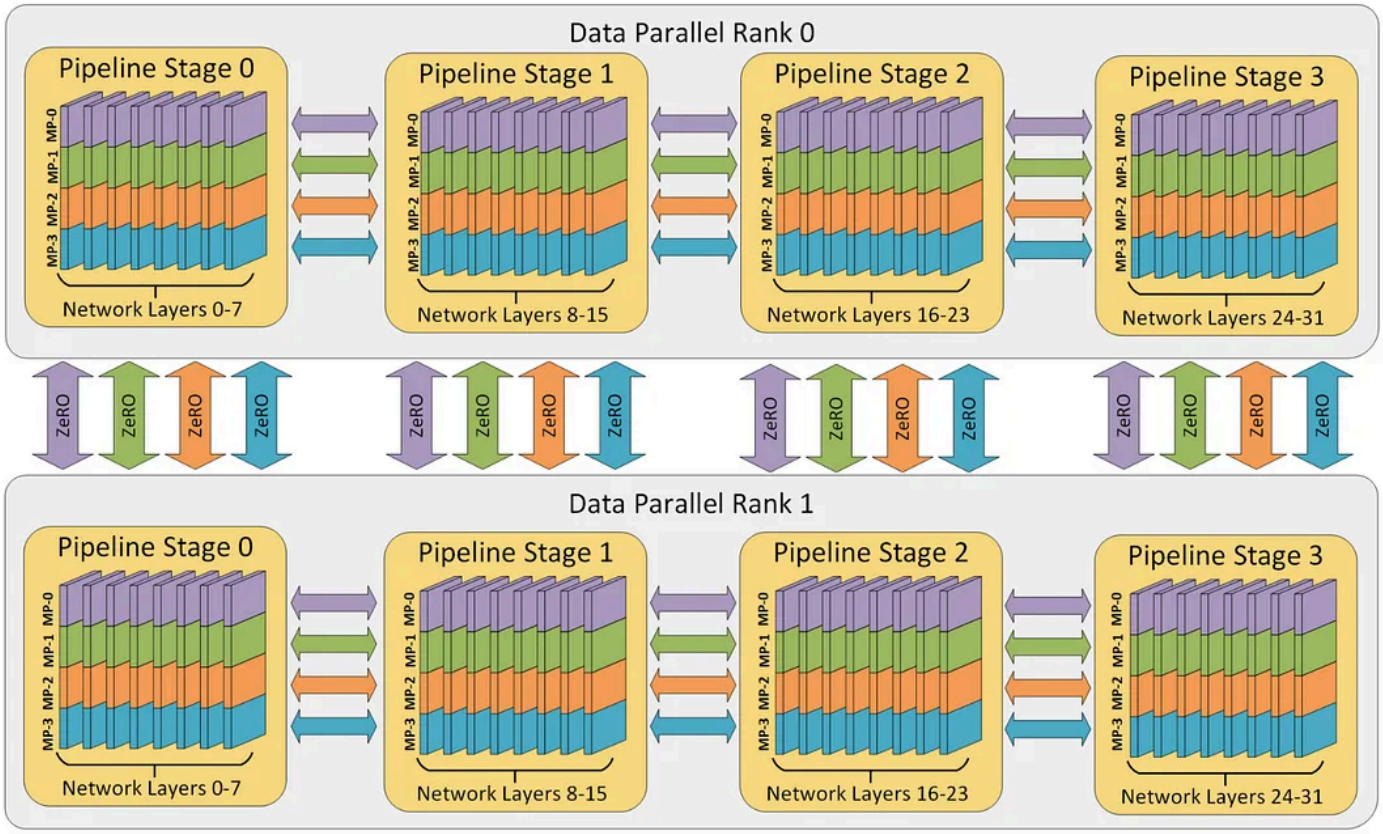
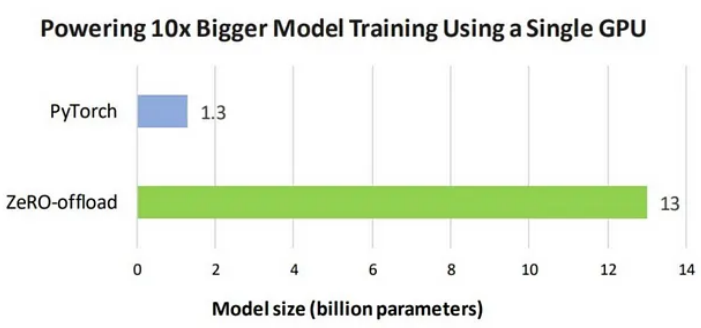
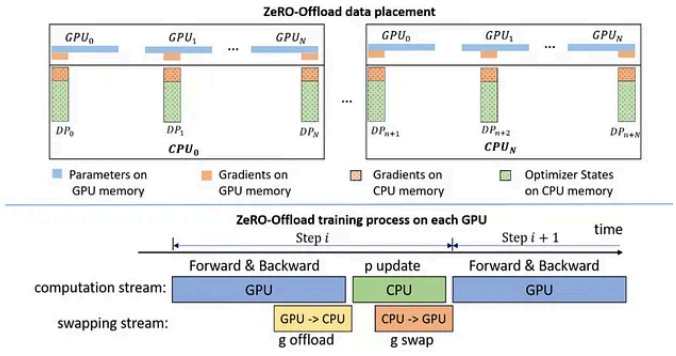


Figure 1: Example 3D parallelism with 32 workers. Layers of the neural network are divided among four pipeline stages. Layers within each pipeline stage are further partitioned among four model parallel workers. Lastly, each pipeline is replicated across two data parallel instances, and ZeRO partitions the optimizer states across the data parallel replicas.

ZeRO-Offload: 10x bigger model training using a single GPU

ZeRO-Offload pushes the boundary of the maximum model size that can be trained efficiently using minimal GPU resources, by exploiting computational and memory resources on both GPUs and their host CPUs. It allows training up to 13-billion-parameter models on a single NVIDIA V100 GPU, 10x larger than the state-of-the-art while retaining high training throughput of over 30 teraflops per GPU.





The figure below shows the architecture of ZeRO-Offload.

DeepSpeed Sparse Attention: Powering 10x longer sequences with 6x faster execution

Sparse attention is a crucial component in DeepSpeed, Microsoft's advanced deep learning optimization library. It's designed to address the computational and memory challenges posed by the attention mechanisms in large transformer models, such as those used in natural language processing.

The Challenge with Standard Attention

In traditional attention mechanisms, like those in Transformer models, the computational and memory requirements grow quadratically with the sequence length. This means that as the input sequence (like a text) gets longer, the resources needed to process it increase dramatically, limiting the ability to handle long sequences effectively.

What is Sparse Attention?

Sparse attention is an innovative solution that reduces these compute and memory demands. Instead of attending to every element in the sequence, sparse attention selectively focuses on a subset of relevant elements. This approach dramatically decreases the number of computations and the amount of memory required.

How Sparse Attention Works

- Selective Focus: Sparse attention mechanisms only compute attention for a subset of the input sequence, choosing elements based on certain criteria or patterns.
- Block-Sparse Computation: DeepSpeed implements sparse attention using block-sparse computation. This method divides the attention matrix into smaller blocks and only computes the blocks that are necessary.
- Different Patterns: Sparse attention can be designed to focus locally (nearby tokens), globally (tokens far apart), or in a combination of patterns. This flexibility allows it to be tailored to the specific requirements of different tasks and datasets.

Benefits of Sparse Attention

- Handling Longer Sequences: With sparse attention, DeepSpeed can process much longer sequences than traditional attention mechanisms. This capability is crucial for tasks involving lengthy documents or conversations.
- Increased Efficiency: Sparse attention reduces the computational load, enabling faster processing and training times. This efficiency is especially beneficial

when scaling up to large models and datasets.

- Memory Savings: By reducing the number of computations, sparse attention also cuts down the memory usage, allowing for training on hardware with more limited resources.

Applications and Impact

Sparse attention in DeepSpeed opens up new possibilities for training large-scale transformer models, particularly in fields like natural language processing, where long sequence lengths are common. It allows for more efficient and effective training of models, pushing the boundaries of what's possible in AI and deep learning.

In summary, sparse attention is a key feature in DeepSpeed that addresses the scalability challenges of traditional attention mechanisms. By enabling the processing of longer sequences with reduced computational and memory requirements, it significantly enhances the capabilities of transformer-based models.To address this limitation, **DeepSpeed offers a suite of sparse attention kernels** — an instrumental technology that can reduce the compute and memory requirement of attention computation by orders of magnitude via block-sparse computation. The suite not only alleviates the memory bottleneck of attention calculation, but also performs sparse computation efficiently. Its APIs allow convenient integration with any transformer-based models. Along with providing a wide spectrum of sparsity structures, it has the flexibility of handling any user-defined block-sparse structures.

Efficient implementation on GPUs: While a basic implementation of sparse attention may show a benefit of memory savings, computationally it can be even worse than full computation. This is mainly due to the divergence and un-coalesced memory access that sparse data adds to the full picture. In general, developing efficient sparse kernels, particularly on GPUs, is challenging. DeepSpeed offers efficient sparse attention kernels developed in Triton. These kernels are structured in block-sparse paradigm that enables aligned memory access, alleviates thread divergence, and balances workloads on processors.

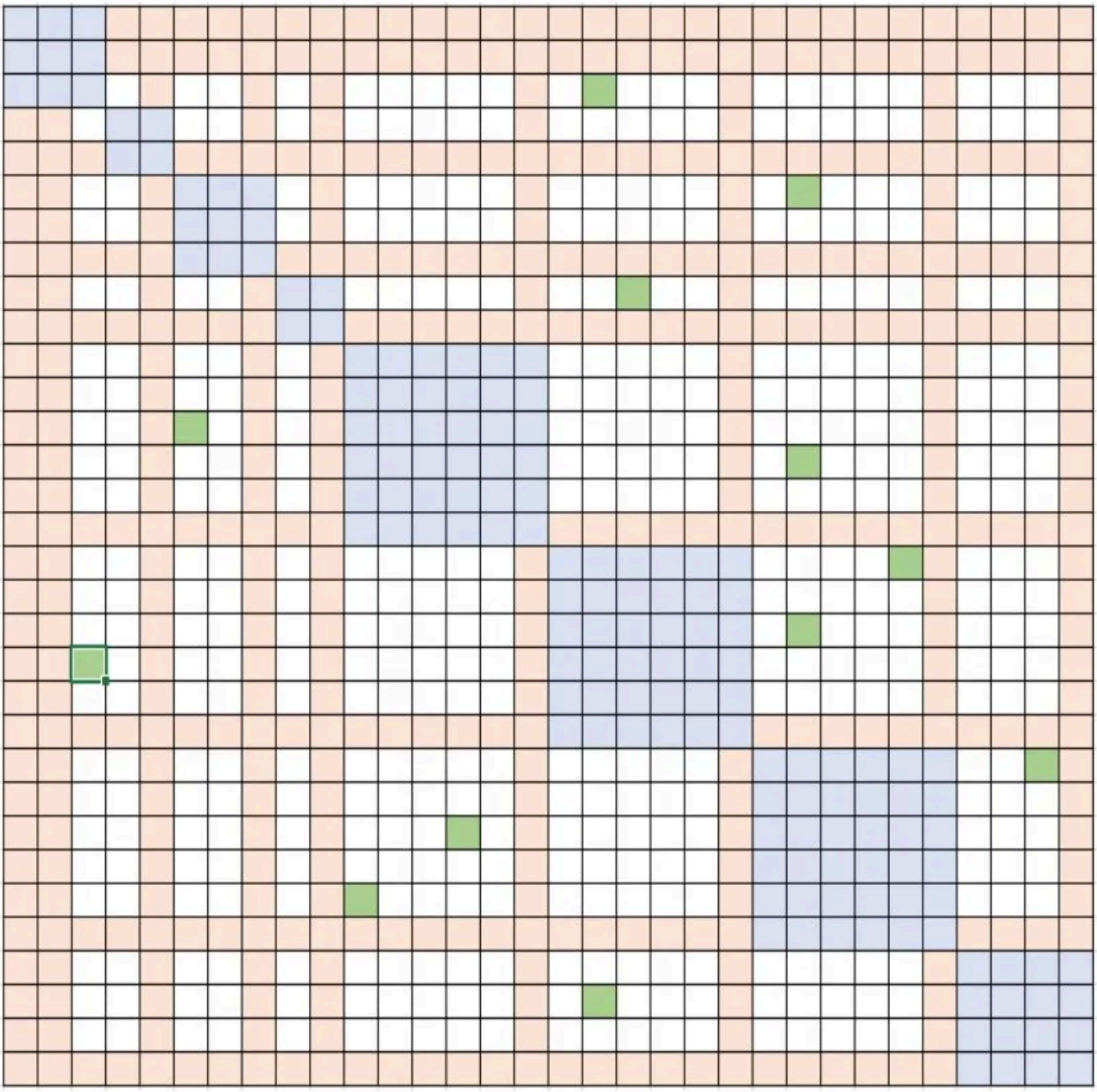


Figure 10: Variable Sparsity structure

System performance: SA powers over **10x longer sequences** and **up to 6.3x faster computation** as shown in Figure 11. The left figure shows the longest sequence length runnable in BERT-Base and BERT-Large models under three settings: dense, dense with activation checkpoint, and sparse (SA) with activation checkpoint. SA empowers 10x and 16x longer sequences when compared with dense for BERT-Base and BERT-Large, respectively. Furthermore, SA reduces total computation compared

with dense and improves training speed: the boost is higher with increased sequence length, and it is up to 6.3x faster for BERT-Base and 5.3x for BERT-Large.

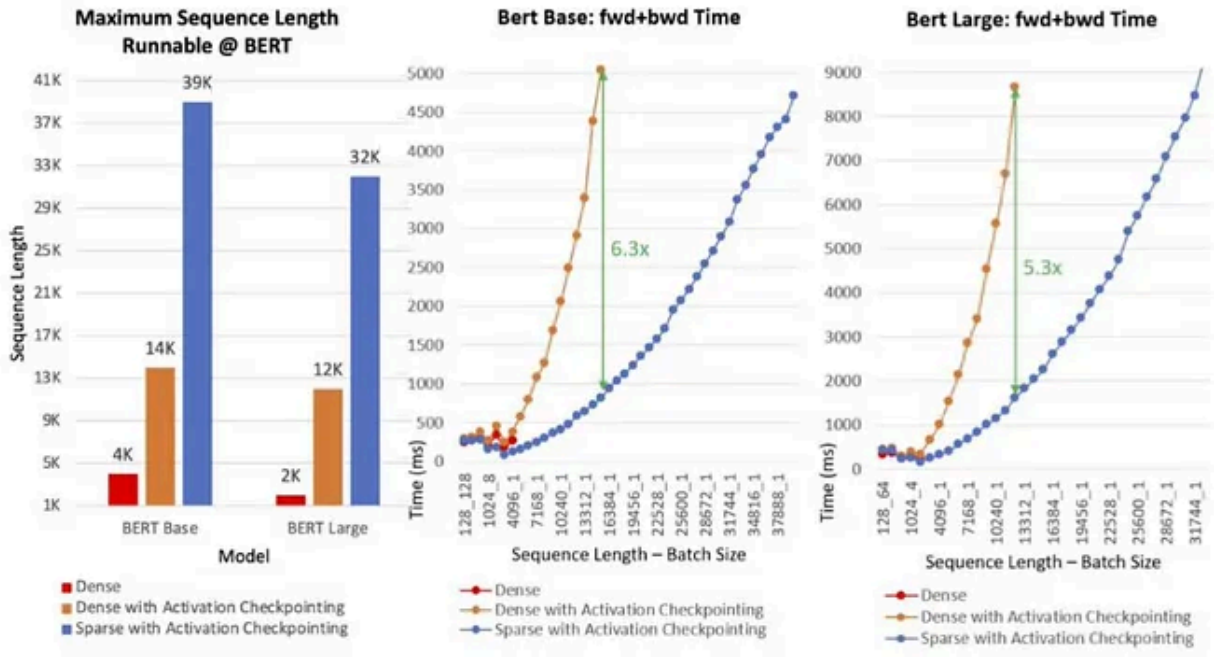
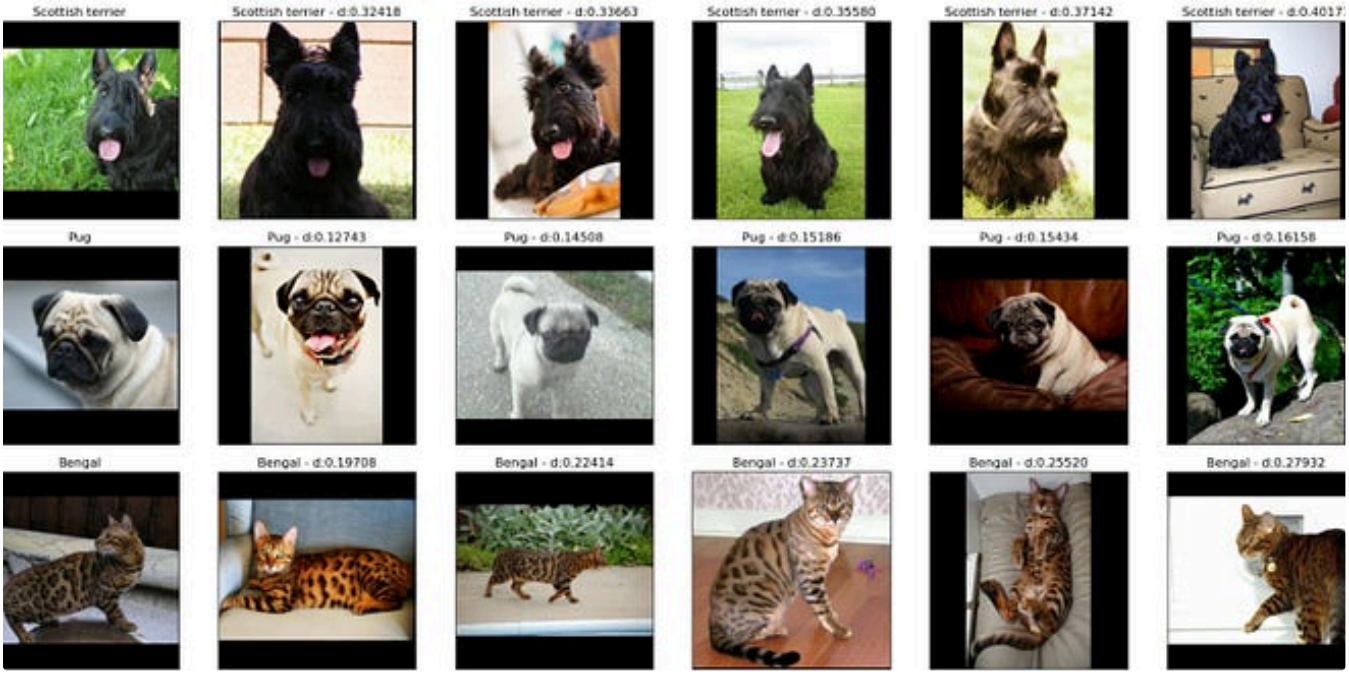


Figure 11: Maximum possible sequence length for BERT models (left); Training time of BERT-Base (center) and BERT-Large (right) on a single NVIDIA V100 GPU with varying sequence length.

Conclusion

DeepSpeed is revolutionizing the field of AI by enabling the training of large-scale models with efficiency and scalability. Its innovative features like ZeRO, 3D Parallelism, ZeRO-Offload, and Sparse Attention are key to managing memory constraints and computational challenges, democratizing large model training. As AI continues to evolve, DeepSpeed is poised to be at the forefront, driving progress and unlocking new potentials in deep learning.

More from Srikaran



Srikaran

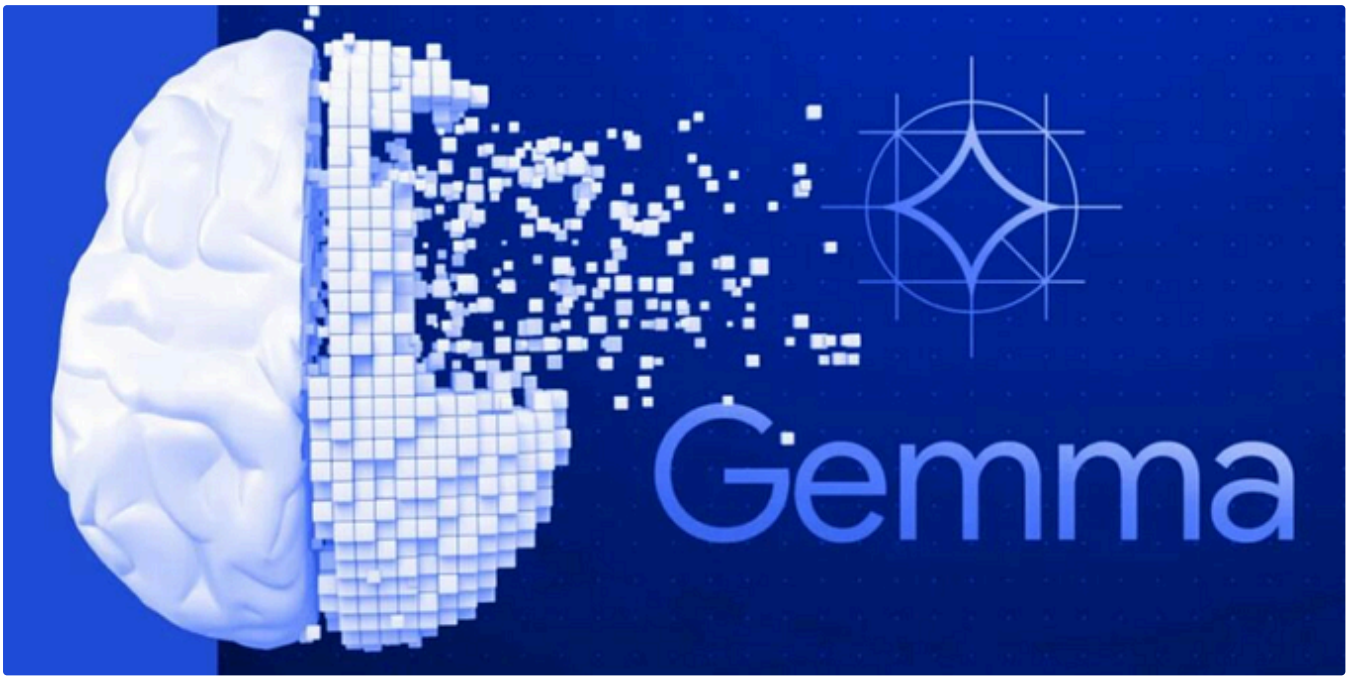
Exploring Siamese Networks for Image Similarity using Contrastive Loss

Aug 9, 2023 62 2

Srikaran

LoRA And QLoRA: An Efficient Approach to Fine-tuning Large Models Under the hood

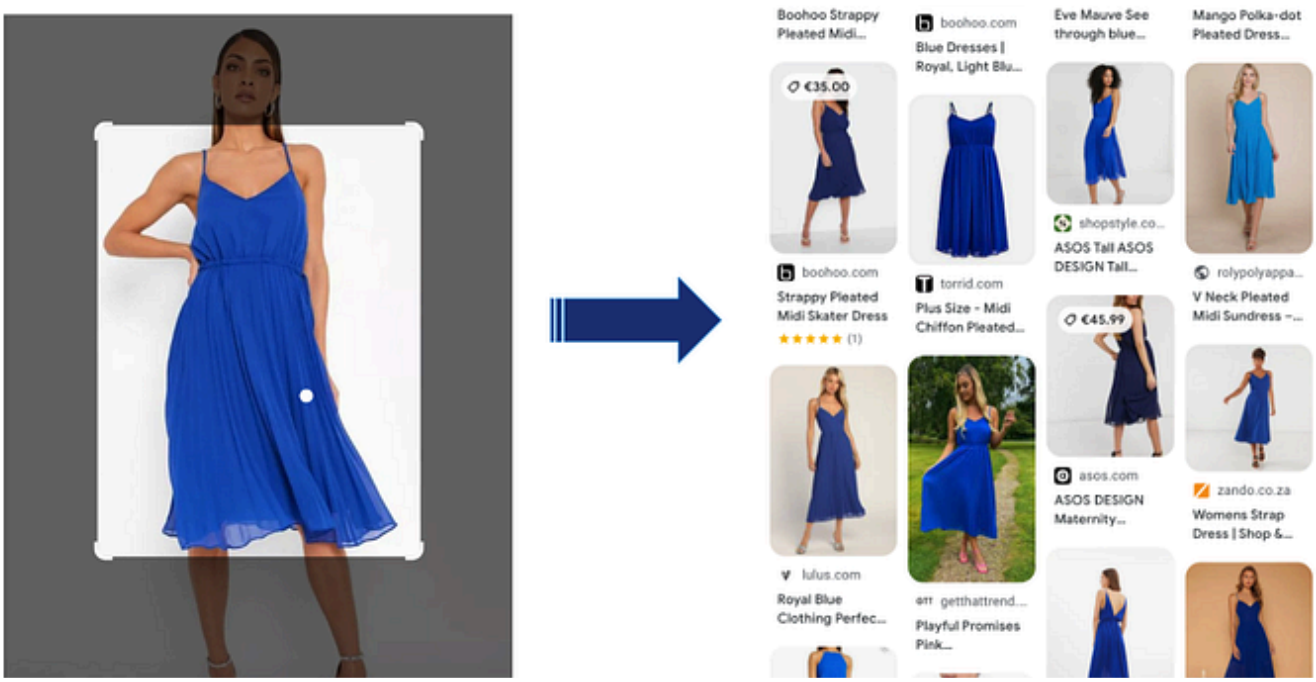
Aug 29, 2023 82



Srikaran

Finetune Gemma-2b for Text to SQL

May 16 3



Srikaran

“Pixel Affinity: Unlocking Image Similarity with ViTForImageClassification and Faiss”

Aug 9, 2023 5

See all from Srikaran

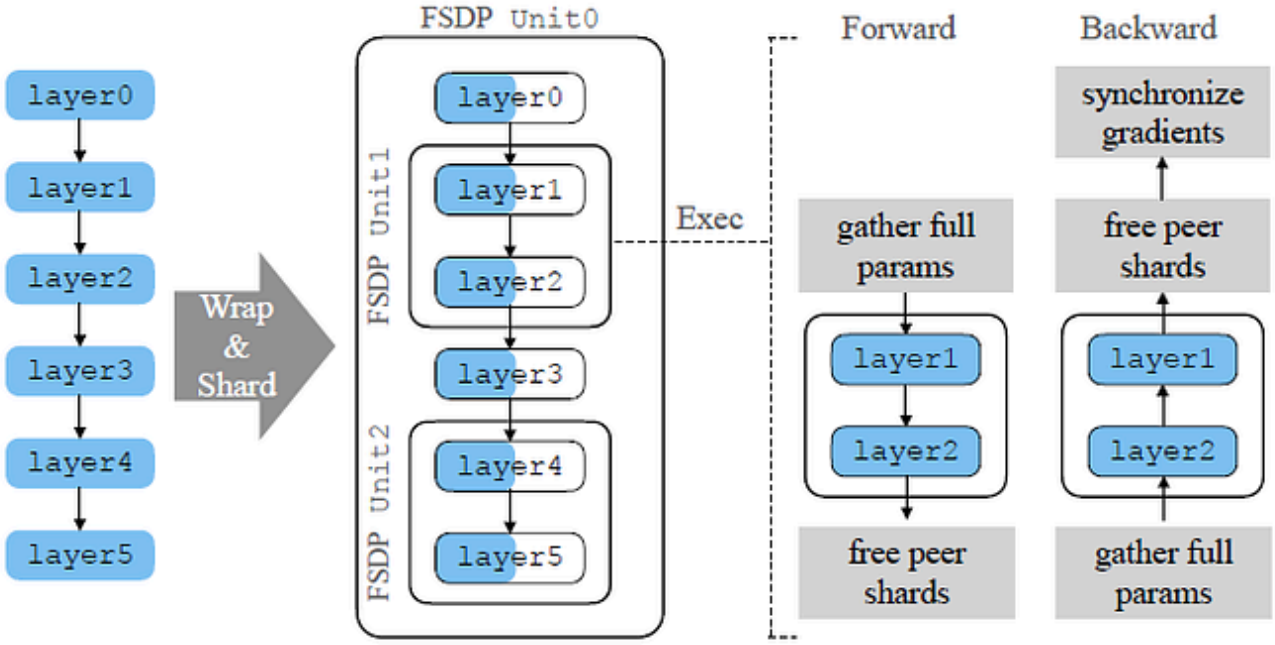
Recommended from Medium

Liana Napalkova

Fine-tuning Small Vision Language Models: Phi-3-vision

In this article, we will explore the process of fine-tuning Phi-3-vision, a small vision-language model developed by Microsoft. We will use...

Jul 12 40



Don Moon in Byte-Sized AI

LLM Training—Fully Sharded Data Parallel (FSDP): An Efficient Distributed Training Technique in...

Overview of Pytorch's Fully Sharded Data Parallel (FSDP)

May 31 14

Lists

Staff Picks
745 stories · 1352 saves

Stories to Help You Level-Up at Work
19 stories · 823 saves

Self-Improvement 101
20 stories · 2828 saves

Productivity 101
20 stories · 2417 saves

azhar in azhar labs

Fine-Tuning the Qwen2-VL Model: A Comprehensive Guide

The growing landscape of AI and machine learning has seen significant advancements, especially in the domain of multimodal models—those...

Sep 12 12

Nitin Tiwari in Google Developer Experts

[ML Story] Fine-tune Vision Language Model on custom dataset

Introduction

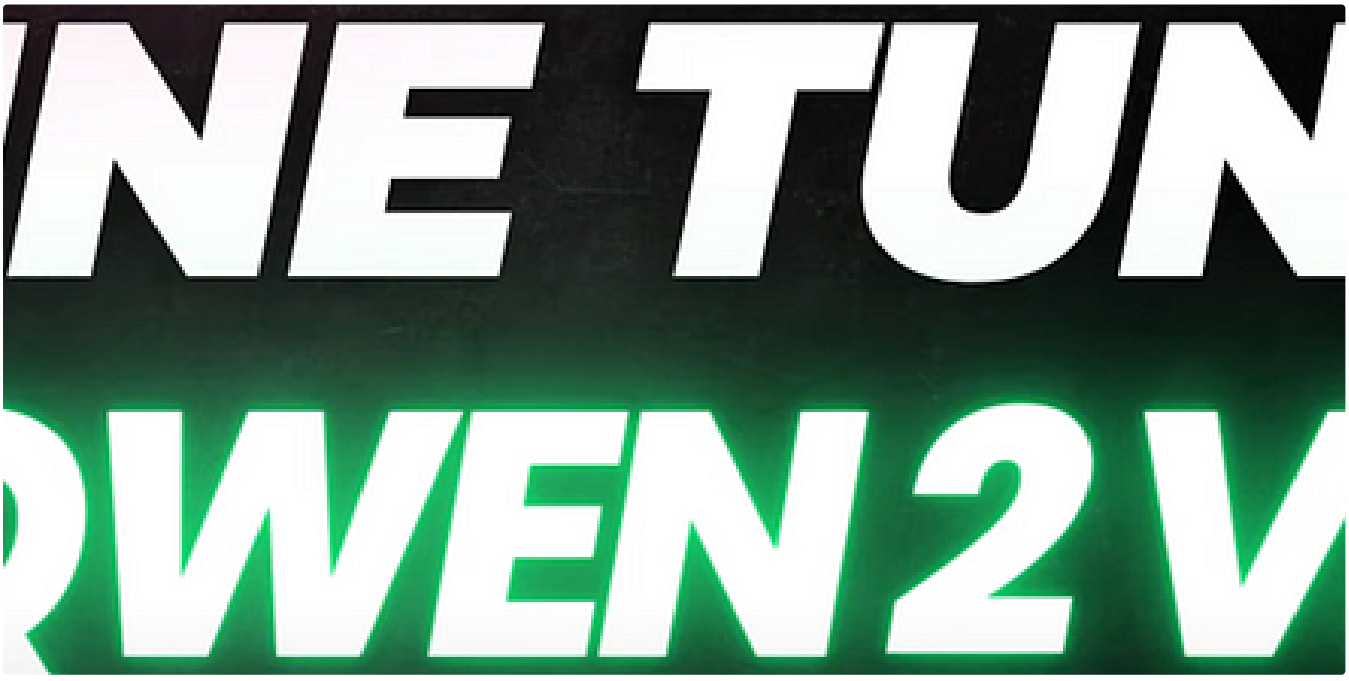
Apr 27 59

Suresh Pawar

Maximizing Efficiency: A Comprehensive Guide to GPU and Memory Selection for Training, Tuning, and...

Introduction:

Apr 15 65



Richardson Gunde

Fine-Tuning the Multimodal Marvel: Qwen-2 VL with LlamaFactory

Hey there, AI enthusiasts! Today we're diving deep into the exciting world of multimodality with Qwen-2 VL, a cutting-edge open-source...

Sep 8 6