ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, Yuxiong He

{samyamr, olruwase, jerasley, shsmit, yuxhe}@microsoft.com

ABSTRACT

In the last three years, the largest dense deep learning models have grown over 1000x to reach hundreds of billions of parameters, while the GPU memory has only grown by 5x (16 GB to 80 GB). Therefore, the growth in model scale has been supported primarily though system innovations that allow large models to fit in the aggregate GPU memory of multiple GPUs. However, we are getting close to the GPU memory wall. It requires 800 NVIDIA V100 GPUs just to fit a trillion parameter model for training, and such clusters are simply out of reach for most data scientists. In addition, training models at that scale requires complex combinations of parallelism techniques that puts a big burden on the data scientists to refactor their model.

In this paper we present ZeRO-Infinity, a novel heterogeneous system technology that leverages GPU, CPU, and NVMe memory to allow for unprecedented model scale on limited resources without requiring model code refactoring. At the same time it achieves excellent training throughput and scalability, unencumbered by the limited CPU or NVMe bandwidth. ZeRO-Infinity can fit models with tens and even hundreds of trillions of parameters for training on current generation GPU clusters. It can be used to fine-tune trillion parameter models on a single NVIDIA DGX-2 node, making large models more accessible. In terms of training throughput and scalability, it sustains over 25 petaflops on 512 NVIDIA V100 GPUs (40% of peak), while also demonstrating super linear scalability. An open source implementation of ZeRO-Infinity is available through DeepSpeed[1].

1 EXTENDED INTRODUCTION

Deep learning (DL) has made tremendous advances in recent years, allowing it to become an integral part of our lives from powering our search engines to our smart home virtual assistants. Increased model size is at the center of these advancements [1–3], and multiple studies have shown that this trend will continue [4, 5]. As a result, there has been significant investment in training huge models.

In the last three years, the largest trained dense model in deep learning has grown over 1000x, from one hundred million parameters (ELMo [6]) to over one hundred *billion* parameters (GPT-3 [4]). In comparison, the single GPU memory has increased by a meager 5x (16 GB to 80 GB). Therefore, the growth in model size has been made possible mainly through advances in system technology for training large DL models, with parallelism technologies such as
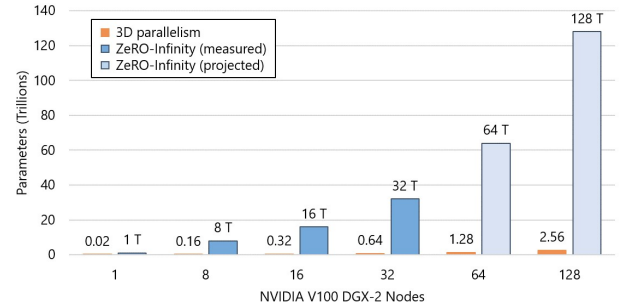

Figure 1: ZeRO-Infinity can train a model with 32 trillion parameters on 32 NVIDIA V100 DGX-2 nodes (512 GPUs), 50x larger than 3D parallelism, the existing state-of-the-art.

model parallelism [7], pipeline parallelism [8–10], and ZeRO [11, 12] creating a path to training larger and more powerful models.

The current state-of-the-art in large model training technology is three-dimensional parallelism (3D parallelism [13, 14]), which combines model (tensor-slicing) and pipeline parallelism with data parallelism to efficiently scale DL training to trillions of parameters on hundreds or thousands of GPUs. For example, the DeepSpeed implementation of 3D parallelism can scale to over a trillion parameters on 800 NVIDIA V100 GPUs by fully leveraging the aggregate GPU memory of a cluster [15].

Despite the capabilities of 3D parallelism for large model training, we are now arriving at the GPU memory wall [16]. The aggregate GPU memory is simply not large enough to support the growth in model size. Even with the newest NVIDIA A100 GPUs with 80 GB of memory, 3D parallelism requires 320 GPUs just to fit a trillion-parameter model for training, and scaling to a hundred trillion parameter model of the future would require over 6K GPUs even if we assume a 5x increase in GPU memory in the next few years. We can no longer sustain the continuous growth in the model scale with GPU memory as the bottleneck.

The GPU memory wall also limits data scientists from accessing even the large models of today, especially for fine tuning. Large models are first pretrained on large amounts of generic data, and through fine tuning the same model can be specialized for a wide variety of applications. While pretraining a model with hundreds of billions of parameters can require millions of GPU compute hours, fine-tuning it is much cheaper, requiring significantly fewer GPU compute hours, and could be done on a single compute node with a handful of GPUs. While such compute resources are accessible to many businesses and users, they are unfortunately restricted by the memory available on these compute nodes, which in turn limits the size of the model that can be fine tuned. It makes large model fine

---

[1]DeepSpeed (https://www.deepspeed.ai/) is a deep learning optimization library designed to make distributed training easy, efficient, and effective. DeepSpeed has been extensively adopted by the DL community.

tuning inaccessible to most researchers and companies that do not have access to massive GPU clusters. For example fine-tuning GPT-3 would require over 8 DGX-2 nodes(128 GPUs) with 3D parallelism to just fit the model for training, even though a single DGX-2 node (16-GPUs) has enough compute to fine-tune it in a reasonable time.

In addition to the GPU memory wall, state-of-the-art for training massive models is also limited in terms of usability and flexibility. As discussed above, 3D parallelism requires combining data, model, and pipeline parallelism in sophisticated ways to get to hundreds of billions or trillions of parameters. While such a system can be very efficient, it requires data scientists to perform major model code refactoring, replacing single GPU operators with tensor-sliced versions, and splitting the model into load-balanced pipeline stages. This also makes 3D parallelism inflexible in the types of models that it can support. Models with complex dependencies cannot be easily converted into a load-balanced pipeline.

Given the landscape of large model training, 3 questions arise:

- Looking ahead, how do we support the next 1000x growth in model size, going from models like GPT-3 with 175 billion parameters to models with hundreds of *trillions* of parameters?
- How can we make large models of today accessible to more data scientists who don't have access to hundreds to GPUs?
- Can we make large model training easier by eliminating the need for model refactoring and multiple forms of parallelism?

In this paper, we take a leap forward from 3D parallelism and present ZeRO-Infinity, a novel system capable of addressing all the aforementioned challenges of large model training.

Unprecedented Model Scale ZeRO-Infinity extends the ZeRO family of technology [11, 12] with new innovations in heterogeneous memory access called the *infinity offload engine*. This allows ZeRO-Infinity to support massive model sizes on limited GPU resources by exploiting CPU and NVMe memory simultaneously. In addition, ZeRO-Infinity also introduces a novel GPU memory optimization technique called *memory-centric tiling* to support extremely large individual layers that would otherwise not fit in GPU memory even one layer at a time. With the infinity offload engine and memory-centric tiling, ZeRO-Infinity not only supports the next 1000x growth in model size, but also makes large models accessible to data scientists with limited GPU resources.

Excellent Training Efficiency ZeRO-Infinity introduces a novel data partitioning strategy for leveraging aggregate memory bandwidth across all devices, which we refer to as *bandwidth-centric partitioning*, and combines it with powerful communication *overlap-centric design*, as well as optimizations for high performance NVMe access in the infinity offload engine. Together, ZeRO-Infinity offers excellent training efficiency, despite offloading data to CPU or NVMe, unencumbered by their limited bandwidth.

Ease of Use With ZeRO-Infinity, data scientists no longer have to adapt their model to multiple forms of parallelism like in 3D parallelism. This is possible due to *memory-centric tiling* in ZeRO-Infinity discussed above aimed at reducing GPU memory requirements of large individual layers that would otherwise require model parallelism (tensor-slicing) to fit the layers in GPU memory. In addition, ZeRO-Infinity eliminates the need for manual model code refactoring, even when scaling to trillions of parameters via an *ease inspired*

implementation that automates all of the communication and data partitioning required for training arbitrary model architectures.

The main contributions of this paper are as follows:

- Memory and performance characterization for large model training that describes the memory requirements (Sec. 3) for different components of a large model training as well as their bandwidth requirements (Sec. 4) for the training to be efficient.
- ZeRO-Infinity (Sec. 5, 6 & Sec. 7): A novel DL training system technology consisting five innovative technologies to address the memory and bandwidth requirements for offering unprecedented model scale that is accessible and easy to use while achieving excellent training efficiency: i) **infinity offload engine** to fully leverage heterogeneous architecture on modern clusters by simultaneously exploiting GPU, CPU and NVMe memory, and GPU and CPU compute, ii) **memory-centric tiling** to handle massive operators without requiring model parallelism, iii) **bandwidth-centric partitioning** for leveraging aggregate memory bandwidth across all parallel devices, iv) **overlap-centric design** for overlapping compute and communication, v) **ease-inspired implementation** to avoid model code refactoring.
- An extensive evaluation of ZeRO-Infinity demonstrating: i) unprecedented scale running 32 trillion parameters on 32 NVIDIA DGX-2 nodes (512 V100 GPUs), ii) excellent training efficiency achieving over 25 petaflops in throughput on the same hardware, iii) superlinear scalability of a trillion parameter model, iv) accessibility and ease-of-use: fine-tune up to a trillion parameter model on a single DGX-2 node, without using any model parallelism or model code refactoring, and v) impact of different technologies in ZeRO-Infinity on model-scale and efficiency (Sec. 8).
- A discussion of ZeRO-Infinity and its potential implications of future hardware system design (Sec. 9)
- An open source implementation of ZeRO-Infinity in DeepSpeed[2], a deep learning optimization library for making distributed training easy, efficient, and effective training that has been extensively adopted by the DL community.

2 BACKGROUND AND RELATED WORK

Data, Model, Pipeline and 3D Parallelism Parallelization is an important strategy for training large models at scale. For a model that fits in the device memory for training, data parallelism (DP) can be used to scale training to multiple devices. When models do not fit in device memory, model parallelism[3] (MP) [7, 17, 18] and pipeline parallelism (PP) [7–9] can split the model among processes, vertically and horizontally, respectively. 3D parallelism [14, 15] combines data, model, and pipeline parallelism to leverage the merits of each, allowing it to scale to trillions of parameters efficiently. While 3D parallelism can be highly efficient, it requires i) significant model code refactoring to split the model into model and pipeline parallel components, ii) models with complex dependency graphs are difficult to be expressed into load-balanced pipeline stages and iii) the model size is limited by the total available GPU memory. We refer the reader to Ben-Nun and Hoefler [19] for a thorough survey on parallelism in DL.

ZeRO: Zero Redundancy Optimizer ZeRO [11] removes the memory redundancies across data-parallel processes by partitioning the three model states (i.e., optimizer states, gradients, and parameters) across data-parallel processes instead of replicating them. By doing so, it boosts memory efficiency compared to classic data parallelism while retaining its computational granularity and communication efficiency. There are three stages in ZeRO corresponding to three model states: the first stage (ZeRO-1) partitions only the optimizer states, the second stage (ZeRO-2) partitions both the optimizer states and the gradients, and the final stage (ZeRO-3) partitions all three model states. In ZeRO-3, the parameters in each layer of the model are owned by a unique data parallel process. During the training, ZeRO-3 ensures that the parameters required for the forward or backward pass of an operator are available right before its execution by issuing broadcast communication collectives from the owner process. After the execution of the operator, ZeRO-3 also removes the parameters as they are no longer needed until the next forward or backward pass of the operator. Additionally, during the parameter update phase of training, ZeRO-3 ensures that each data-parallel process only updates the optimizer states corresponding to the parameters that it owns. Thus, ZeRO-3 can keep all the model states partitioned throughout the training except for the parameters that are required by the immediate computation.

Heterogeneous Training Approaches Out of several heterogeneous CPU memory based training approaches [20–26], ZeRO-Offload [12] is the state-of-the-art (SOTA) for large model training on multi-GPUs. ZeRO-Offload is built on top of ZeRO-2 and stores the gradients and the optimizer states in CPU memory. ZeRO-Offload leverages CPU memory in the absence of enough GPU devices to store the optimizer states and gradients. However, it still requires the parameters to be stored in GPU memory and replicated across all devices. Thus, the model scale with ZeRO-Offload is limited to the total number of parameters that the memory on a single GPU device can host. ZeRO-Offload also requires a large batch size to remain efficient due to suboptimal data partitioning and limited PCIe bandwidth. We address these limitations of ZeRO-Offload with ZeRO-Infinity. In terms of NVMe based approaches, Zhao et al. [27] use a hierarchical parameter server-based design to offload sparse parameters to SSD for creating a massive scale DL Ads System. In contrast, ZeRO-Infinity is designed to be a generic DL system for training massive dense models.

Reducing Activation Memory Activations are the intermediate results produced during the forward propagation that need to be retained to compute the gradients during backward propagation. Multiple efforts have focused on reducing the memory required by activations through compression [28], activation checkpointing [29, 30], or live analysis [31]. ZeRO-Infinity works together with activation checkpointing to reduce activation memory.

Adam Optimizer and Mixed Precision Training Adaptive optimization methods [32–35] are crucial to achieving SOTA performance and accuracy for effective model training of large models. Compared to SGD, by maintaining fine-grained first-order and second-order statistics for each model parameter and gradient at the cost of significant memory footprint. Adam [33] is the optimizer used most prominently in large model training.

Large model training is generally trained in mixed precision, where the forward and backward propagation are done in FP16 and

the parameter updates in FP32 [36]. This leverages the performance acceleration of the tensor core units available on modern GPUs [37].

3 MEMORY REQUIREMENTS

This section characterizes the memory requirements for DL training. While our methodology is generic, we focus the concrete analysis on Transformer [38] based architectures since all of the SOTA models with over a billion parameters follow that. Our analysis assumes mixed precision training with the Adam optimizer since this recipe is the de facto standard for training Transformer based models.

The memory required for training can be categorized into two components: i) Model states including optimizer states, gradients, and model parameters, ii) Residual states primarily referring to activation memory. To study training on heterogeneous resources, we also characterize the GPU working memory, describing the minimum amount of memory that must be available on the GPU to support training, assuming the model and residual states can be successfully offloaded from GPU memory.

Memory for Model States: The model states are comprised of optimizer states, gradients, and parameters. For mixed precision training with Adam optimizer, the parameters and gradients are stored in FP16 while the optimizer states consist of FP32 momentum, variance, parameters, and gradients. In total, each parameter requires 20 bytes of memory. The total number of parameters in a Transformer based model primarily depends on the hidden dimension (hd) and the number of Transformer layers (nl). Nearly all the parameters in a Transformer block come from four linear layers within each block with sizes: (hd, $3hd$), (hd, hd), (hd, $4hd$) and ($4hd$, hd), respectively. Thus, the total parameters in a Transformer based model and can be approximated as

$$12 \times nl \times hd^2 \qquad (1)$$

requiring a total memory

$$240 \times nl \times hd^2 \qquad (2)$$

in bytes to store the model states.

Figure 2a column 5 shows the memory required to store the model states of a GPT-3 like Transformer based model with 100 billion to a 100 trillion parameters created by varying hidden dimension and number of layers. To put the memory requirements in context, Figure 2b column 3 shows the aggregate GPU memory available on a single NVIDIA V100 DGX-2 box as well as a DGX-2 SuperPOD cluster. Note that it requires 64 GPUs to just fit the model states for a 100B parameter model. Fitting a trillion parameter model requires over 512 GPUs, while a 10 trillion parameter model is beyond the scope of even a massive 1536 GPU cluster.

Memory for Residual States: The residual states primarily consist of the activation memory, which depends on the model architecture, batch size (bsz) and sequence length (seq), and it can be quite large. On the positive side, the memory required for activation can be significantly reduced via activation checkpointing [29], which trades off activation memory at the expense of 0.33x additional recomputation. Large models such as Turing-NLG 17.2B and GPT-3 175B were all trained using activation checkpointing. The memory required to store activation checkpoints is estimated as

$$2 \times bsz \times seq \times hd \times nl/ci \qquad (3)$$

Params (Trillions)	Layers	Hidden Size	Attn Heads	Model States (TB/Model)	TB/Node		Working Mem. per GPU (GB)	
					Act.	Act. Ckpt.	Model State	Act.
0.10	80	10K	128	1.83	2.03	0.05	1.95	1.63
0.50	100	20K	160	9.16	3.91	0.12	6.25	2.50
1.01	128	25K	256	18.31	7.13	0.20	9.77	3.56
10.05	195	64K	512	182.81	24.38	0.76	64.00	8.00
101.47	315	160K	1024	1845.70	88.59	3.08	400.00	18.00

(a)

Nodes	GPUs	Aggregate Memory (TB)			GPU-GPU Bandwidth (GB/s)	Memory Bandwidth/GPU (GB/s)		
		GPU	CPU	NVMe		GPU	CPU	NVMe
1	1	0.032	1.5	28.0	N/A	600-900	12.0	12.0
1	16	0.5	1.5	28.0	150-300	600-900	3.0	1.6
4	64	2.0	6.0	112.0	60-100	600-900	3.0	1.6
16	256	8.0	24.0	448.0	60-100	600-900	3.0	1.6
64	1024	32.0	96.0	1792.0	60-100	600-900	3.0	1.6
96	1536	48.0	144.0	2688.0	60-100	600-900	3.0	1.6

(b)

Figure 2: (a) Memory requirements for massive models. (b) Available memory and achievable bandwidth on NVIDIA V100 DGX-2 Cluster (The reported bandwidths represent per GPU bandwidth when all GPUs are reading data in parallel from the designated memory).

bytes where ci is the number of Transformer blocks between two activation checkpoints, and $bsz \times seq \times hd$ is the size of the input to each Transformer block. Figure 2a column 7 shows the memory required to store activation checkpoints for batch size of 32 and sequence length of 1024 assuming we store one activation per Transformer block. Many modern GPU clusters have 8-16 GPUs per node, and so we chose a batch size of 2-4 per GPU, resulting in a batch size of 32 as a conservative estimate of activation within each node. While the resulting activation checkpoints are orders of magnitude smaller than the full set of activations (column 6) , beyond a trillion parameters they still get too large to fit in GPU memory for the batch size and sequence length under consideration.

Model State Working Memory (MSWM) is the minimum amount of GPU memory required to perform forward or backward propagation on the largest single operator in the model after all the model states have been offload to CPU or NVMe. This is approximately given by the size of the parameters and gradients of that operator in the model, since there must be at least enough memory to hold the parameter and its gradient for backward propagation.

For a Transformer based model, the largest operator is a linear layer that transforms hidden states from hd to $4hd$. The size of the parameter and gradients of this linear layer in bytes is

$$4 \times hd \times 4hd \qquad (4)$$

Note that MSWM (Figure 2a Column 8) grows significantly beyond a 100 billion parameters, requiring multiple gigabytes in contiguous memory, which can result in running out of memory during training due to lack of enough contiguous memory to satisfy these requirements. State-of-art approaches like 3D Parallelism, addresses this issue via model parallelism, by splitting individual operator across multiple GPUs. In Sec. 5.1.3, we discuss a novel approach for addressing these massive model state working memory without requiring model parallelism.

Activation Working Memory (AWM) is the memory required in the backward propagation for recomputing the activations before performing the actual backward propagation. This is the size of the activations between two consecutive activation checkpoints. For example, if we create one activation checkpoint per Transformer block, the memory is given by the size of the total activation per Transformer block. This is given in bytes by approximately

$$bsz \times seq \times ci \times (16 \times hd + 2 \times attn\_heads \times seq). \qquad (5)$$

Figure 2a column 8 shows that AWM gets large beyond 10 trillion parameters, even with $ci = 1$. Unlike MSWM that is only composed of a single parameter and gradient, AWM is composed of

dozens of activations, and does not cause memory issues due to lack of contiguous memory as long as the total AWM can fit in GPU memory.

4 BANDWIDTH REQUIREMENTS

A critical question of offloading to CPU and NVMe memory is whether their limited bandwidth will hurt training efficiency. This section characterizes the impact of bandwidth on training efficiency.

We start from defining an efficiency metric. Assuming a workload execution without any compute and communication overlap, we can use the peak computational throughput ($peak_{tp}$), data movement bandwidth (bw) and its arithmetic intensity (ait) to estimate the training efficiency.

The arithmetic intensity (AIT) of a workload is the ratio between the total computation and the data required by the computation. It describes the amount of computation per data movement. Higher AIT means a lower requirement on the data movement bandwidth, since for each data loaded the accelerator can do more computations. The efficiency metric can derived as follows:

$$compute\_time \quad = \frac{total\_computation}{peak_{tp}}$$

$$ait \quad = \frac{total\_computation}{total\_data\_movement}$$

$$communication\_time \quad = \frac{total\_data\_movement}{bw}$$

$$= \frac{total\_computation}{ait \times bw}$$

$$efficiency \quad = \frac{compute\_time}{compute\_time + communication\_time}$$

The efficiency can be written as a function of $peak_{tp}$, bw and ait:

$$efficiency = \frac{ait \times bw}{ait \times bw + peak_{tp}} \qquad (6)$$

We will use this simple efficiency equation to characterize the data movement bandwidth required for training massive models. But before that, we will first quantify ait for DL training workloads.

4.1 Quantifying AIT in DL training

Model states and activation checkpoints can have varying ait. We can quantify them by first identifying the total computation in each iteration of DL training, and then identifying the data movement volume for each of the model states and activations.

Total Computation per Iteration The total computation per iteration is dominated by the computation in the linear layers of

(a) Parameter and Gradient Bandwidth

(b) Optimizer States bandwidth

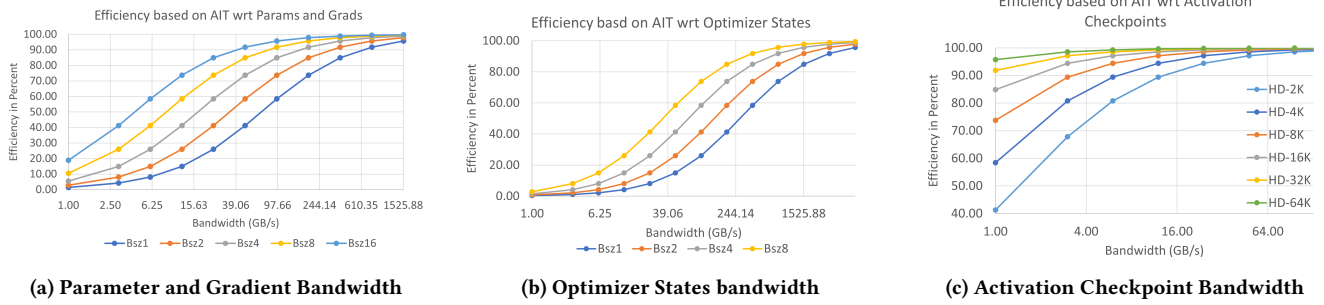**(c) Activation Checkpoint Bandwidth**

Figure 3: Impact of bandwidth on efficiency assuming an accelerator with 70 TFlops of single GPU peak achievable throughput.

the Transformer. For the forward propagation this can be approximated as a function of the number of parameters, sequence length, and batch size, given by $2 \times bsz \times seq \times params$. The cost of backward propagation is approximately twice that of forward propagation. Additionally, activation checkpointing requires an additional forward computation as part of recomputation during backward propagation. Therefore, the total computation per iteration is:

$$computation\_per\_iter = 2 \times 4 \times bsz \times seq \times parameters \quad (7)$$
$$= 2 \times 4 \times 12 \times bsz \times seq \times nl \times hd^2 \quad (8)$$

AIT w.r.t. Parameters and Gradients During forward and back propagation, model *parameters* must be loaded from the source location to GPU registers at least twice, i) during forward, ii) during the actual backward, resulting in a data movement of $2 \times parameters$. In presence of activation checkpointing, the parameters may be loaded one additional time for re-computation during the backward pass, adding another $1 \times parameters$. Furthermore, the *gradients* must be stored from the GPU registers to its final location at least once, adding a final $1 \times parameters$ in data movement.

Therefore, assuming that parameters and gradients are stored at the same final location, the total data movement during the forward and backward pass would be $4 \times parameters$, i.e. $2 \times 4 \times parameters$ in bytes. The total computation per iteration is given by Sec. 4.1. Therefore the *ait* w.r.t *parameter and gradients* is

$$seq \times bsz. \quad (9)$$

AIT w.r.t. Optimizer States During the optimizer step, the optimizer states must be read at least once, and the optimizer states must be written at least once. So the total data movement is $2 \times optimizer\_states$, which is approximately $2 \times 16 \times parameters$ bytes. The total computation per iteration is given by Sec. 4.1. Therefore *ait* w.r.t *optimizer states* during a full training iteration is

$$seq \times bsz/4. \quad (10)$$

AIT w.r.t. Activation Checkpoints During the forward propagation activation checkpoints must be saved to their final location, and must be retrieved during the backward propagation. Therefore, the total data movement w.r.t activation checkpoints in bytes is given by $2 \times total\_activation\_checkpoints\_in\_bytes$ which is given by $4 \times nl/ci \times hd \times seq \times bsz$ from Eq. (3). The total computation per iteration is given by Sec. 4.1. So the *ait* w.r.t activation checkpoints is given by

$$24 \times hd \times ci. \quad (11)$$

4.2 Bandwidth Requirements

Due to the variation in the AIT, model states and activation checkpoints have very different bandwidth requirements to achieve good efficiency. The former only depends on the batch size and sequence length, while the latter only depends on the frequency of activation checkpoints and hidden dimension size of the model.

Besides AIT, the bandwidth requirement for efficiency also depends on $peak_{tp}$, as shown in Eq. (6). Using $peak_{tp}$, and *ait* we first show how efficiency varies with bandwidth w.r.t to different model and residual states, and then discuss the bandwidth requirements on these states for DL training to be efficient. Our methodology is generic and can be applied to understanding the bandwidth requirements on any current or future generation clusters. Here, we use NVIDIA V100 DGX-2 SuperPOD cluster as our example platform.

Using the *ait* expression from Sec. 4.1 and efficiency metric based on Eq. (6), Figure 3 shows the relationship between efficiency and available bandwidth w.r.t. parameter and gradients, optimizer states, and activation checkpoints. To produce these plots, we computed the *ait* based on expressions derived in Sec. 4.1, for varying batch sizes, sequence length and model configurations. More specifically, we use a sequence length of 1024, the same sequence length used for GPT-2 [2], Megatron-LM [7], and Turing-NLG [39]. We vary batch size range from 1 to 16 to capture large GPU and small GPU experiments, respectively. A small batch size per GPU is used when running on large number of GPUs, while a large batch size per GPU is used when training on relatively fewer GPUs to maintain a reasonable effective batch size for training. Our hidden size ranges from 8K-64K representing models with hundreds of billions of parameters, to tens of trillions of parameters as shown in Figure 2a.

To identify $peak_{tp}$ for this analysis, we use an empirical approach[4]. We ran models with aforementioned configurations on a single NVIDIA V100 DGX-2 box with all non-GPU communication turned off to simulate a virtually unlimited bandwidth scenario. The performance achieved ranged from 62-78 TFlops/GPU based on the hidden size of 8K-64K, respectively. We used the average of 70 TFlops/GPU to represent $peak_{tp}$ for the purpose of this analysis[5].

---

[4]Note that $peak_{tp}$ is not the theoretical hardware peak, but instead the achievable peak in the absence of any communication bottleneck.

[5]Results will vary based on the value of $peak_{tp}$ used, and this analysis is a single data point, meant as a guide for understanding relationship between efficiency and bandwidth for DL workloads specifically on the NVIDIA V100 DGX-2 clusters. Furthermore, the result only considers the relationship between efficiency and bandwidth of model states and activations, one at a time, assuming infinite bandwidth for others to isolate the bandwidth requirement for each state separately.

Bandwidth w.r.t. Parameter and Gradients Figure 3a shows that with a bandwidth of over 70 GB/s for parameter and gradients, we can achieve over 50% efficiency for even the smallest batch size. At this bandwidth, the data movement in theory can be completely overlapped with the computation to achieve a 100% efficiency.

Bandwidth w.r.t. Optimizer States Figure 3b shows that optimizer states require nearly 4x higher bandwidth to achieve 50% efficiency compared to parameters and gradients. Furthermore, the optimizer states are updated at the end of the forward and backward propagation and cannot be overlapped with the computation. As a result they require significantly larger bandwidth to keep the overall DL workload efficient. For example achieving 90% efficiency with batch size of 2 per GPU requires nearly 1.5 TB/s of effective bandwidth, which is greater than even the GPU memory bandwidth.

Bandwidth w.r.t. activation memory Figure 3c also shows that with activation checkpointing enabled, a meager bandwidth of 2 GB/s is able to sustain over 50% efficiency even for a hidden size of $2K$. The bandwidth requirement drops down to less than 1 GB/s once the hidden size grows over $8K$.

5 ZERO-INFINITY DESIGN OVERVIEW

In this section we present an overview of the design choices in ZeRO-Infinity that enable it to achieve unprecedented model scale while offering excellent training efficiency and ease of use. A bird's eye view of ZeRO-Infinity is illustrated in Figure 4 and discussed below.

5.1 Design for Unprecedented Scale

Modern GPU clusters are highly heterogeneous in terms of memory storage. In addition to the GPU memory, they have CPU memory as well as massive NVMe storage that is over 50x larger than the GPU memory and nearly 20x larger than CPU memory (See Fig. 2b).

We developed ZeRO-Infinity, a parallel system for DL training that can transcend the GPU memory wall by exploiting these heterogeneous memory systems in modern GPU clusters. Figure 1 compares the maximum achieved model size of 3D parallelism and ZeRO-Infinity. ZeRO-Infinity supports one trillion parameters per NVIDIA V100 DGX-2 node, a 50x increase over 3D parallelism.

5.1.1 Infinity offload engine for model states. ZeRO-Infinity is built on top of ZeRO-3 [11] which partitions all model states to remove memory redundancy as discussed in Sec. 2. Unlike any of the existing ZeRO family of technology, ZeRO-Infinity is designed with a powerful offload mechanism called the infinity offload engine which can offload all of the partitioned model states to CPU or NVMe memory, or keep them on the GPU based on the memory requirements. Note from Fig. 2a and Fig. 2b, even the model states required by a 100 trillion parameter model can fit in the aggregate NVMe memory of a DGX-2 cluster with 96 nodes (1536 GPUs). Therefore, the infinity offload engine allows ZeRO-Infinity to fit model states of models with hundreds of trillions of parameters. See Sec. 6 for more details.

5.1.2 CPU Offload for activations. In addition to model states, ZeRO-Infinity can offload activation memory to CPU memory, when necessary. Note that the activation checkpoints (0.76 TB) required by a 10 trillion parameter model can easily fit in the 1.5TB
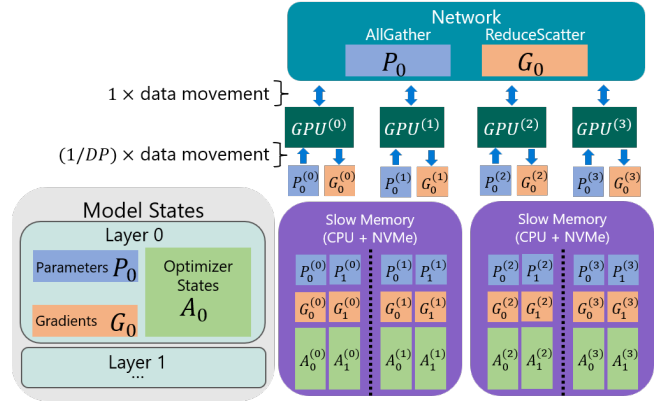

Figure 4: A snapshot of ZeRO-Infinity training a model with two layers on four data parallel (DP) ranks. Communication for the backward pass of the first layer is depicted. Partitioned parameters are moved from slow memory to GPU and then collected to form the full layer. After gradients are computed, they are aggregated, repartitoned, and then offloaded to slow memory. Layers are denoted with subscripts and DP ranks are denoted with superscripts. For example, $P_0^{(2)}$ is the portion of layer 0's parameters owned by $GPU^{(2)}$.

of CPU memory available on a DGX-2 system, while the 3 TBs of activation checkpoints required by a 100 trillion parameter is within reach of the CPU memory of the next generation hardware. Therefore, by offloading activation checkpoints to CPU memory, ZeRO-Infinity can fit the activation checkpoints of models with hundreds of trillions of parameters.

5.1.3 Memory-centric tiling for working memory. To reduce the working memory requirements of DL training for large models, ZeRO-Infinity introduces a novel technique called *memory-centric tiling* that exploits the data fetch and release pattern of ZeRO-3 to reduce the working memory requirements by breaking down a large operator into smaller tiles that can be executed sequentially.

For example, to reduce the working memory for a large linear operator, ZeRO-Infinity represents the operator as a mathematically equivalent sequence of smaller linear operators consisting of tiles of parameters from the original operator, and executes them sequentially. When combined with ZeRO-3, the parameter and gradients of each tile can be fetched and released one at a time, reducing the working memory proportional to the number of tiles. Therefore, ZeRO-Infinity can support operators of arbitrary sizes, without relying on model parallelism to fit them in limited GPU memory.

5.2 Design for Excellent Training Efficiency

Offloading all model states and activations to CPU or NVMe is only practical if ZeRO-Infinity can achieve high efficiency despite the offload. In reality this is extremely challenging since CPU memory is an order of magnitude slower than GPU memory bandwidth, while the NVMe bandwidth is yet another order of magnitude slower than the CPU memory bandwidth. Furthermore, reading and writing to these memory from GPU is even slower (see Fig. 2b).

On a system like the DGX-2, the bandwidth must be greater than 70GB/s, 1.5TB/s, and 1-4 GB/s w.r.t. parameter and gradients,

optimizer states, and activation checkpoints, respectively for DL training to be efficient, based on our analysis in Sec. 4. Here we discuss how ZeRO-Infinity achieves the necessary bandwidths to achieve excellent efficiency.

5.2.1 Efficiency w.r.t Parameter and Gradients. The data movement bandwidth for parameters and gradients must be greater than 70GB/s, close to the GPU-GPU bandwidth available on DGX-2 clusters [40]. Therefore, a DL parallel training solution like ZeRO-3 [11] where parameters are broadcasted from the owner GPU to the rest before using them in forward or backward propagation can run efficiently as long as the communication is overlapped.

On the contrary, a meager 12 GB/s PCIe bandwidth from a single GPU to CPU memory or NVMe (see Fig. 2b) or vice-versa is simply not sufficient to support heterogeneous training at scale[6]. Therefore, existing heterogeneous solutions like ZeRO-Offload where the parameters must be first moved from CPU to owner GPU before broadcasting requires significantly large batch sizes per GPU to achieve enough *ait* necessary to be efficient under the limited bandwidth. This poses two problems: i) for massive models the activation memory will get too large to fit even in CPU memory, and ii) the effective batch size becomes too large when scaling to hundreds or thousands of GPUs for effective convergence.

ZeRO-Infinity addresses these challenges in two ways: i) *bandwidth-centric partitioning*: a novel data mapping and parallel data retrieval strategy for offloaded parameters and gradients that allows ZeRO-Infinity to achieve virtually unlimited heterogeneous memory bandwidth (details in Sec. 6.1), and ii) an *overlap centric design* that allows ZeRO-Infinity to overlap not only GPU-GPU communication with computation but also NVMe-CPU and CPU-GPU communications over the PCIe (details in Sec. 5.1.3).

5.2.2 Efficiency w.r.t Optimizer States. Unlike parameters and gradients that are consumed and produced sequentially during the forward and backward propagation, optimizer states can be updated in parallel, all at once. This property is leveraged by both ZeRO-3 and ZeRO-Offload, that store and update the optimizer states in GPU and CPU memory, respectively, in parallel across all available GPUs and CPUs. As a result the aggregate GPU or CPU memory bandwidth can get much higher than the required 1.5TB/s with increase in GPU or CPU count.

Since ZeRO-Infinity is built upon ZeRO-3, it can also leverage the aggregate GPU and CPU memory bandwidth as well as the aggregate CPU compute for optimizer step, when offloading optimizer states to CPU memory. However, with NVMe offload, it is necessary to bring the data from NVMe to CPU memory and back in chunks that can fit in the CPU memory to perform the optimizer step, one chunk at a time. The optimizer step is therefore limited by the NVMe-CPU memory bandwidth: while ZeRO-Infinity can achieve aggregate NVMe bandwidth across multiple nodes, it is crucial to achieve near peak NVMe bandwidth per node, to allow supporting the necessary bandwidth of over 1.5 TB/s with as few nodes, and as small batch size as possible. Furthermore, the process of bringing data in and out of NVMe to CPU memory, or from CPU memory to GPU memory can cause CPU memory fragmentation

in both GPU and CPU that can result in out of memory even with plenty of memory still available.

The *infinity offload engine* can not only achieve near peak NVMe bandwidth, it can also allows ZeRO-Infinity to overlap NVMe to CPU reads with CPU to NVMe writes, as well as the CPU computation for the optimizer step at the same time to allow ZeRO-Infinity to remain efficient with a modest batch size on small number of GPUs and with small batch sizes on large numbers of GPUs. At the same time, it minimizes memory fragmentation by carefully reusing temporary buffers for data movement. We discuss the optimizations in infinity offload engine and in detail in Sec. 6.

5.2.3 Efficiency w.r.t Activations. On a DGX-2 node, each GPU can read and write data at about 3 GB/s to CPU memory in parallel over the PCIe allowing activation checkpoints to be offloaded to CPU memory while retaining over 80% efficiency for hidden size larger $8K$ or larger. To also allow for high efficiency at smaller hidden sizes, ZeRO-Infinity can decrease the frequency of activation checkpoints as well as effectively overlap the communication of activation checkpoints both to and from CPU memory with the forward and backward computation on the GPU.

5.3 Design for Ease of Use

With ZeRO-Infinity, data scientists no longer have to adapt their model to multiple forms of parallelism like in 3D parallelism. This is possible due to *memory-centric tiling* in ZeRO-Infinity discussed in Sec. 5.1.3 aimed at reducing GPU memory requirements of large individual layers that would otherwise require model parallelism (tensor-slicing) to fit the layers in GPU memory.

In addition, ZeRO-Infinity is implemented in PyTorch in a way that eliminates the need for manual model code refactoring even when scaling to trillions of parameters. This is made possible through an *ease-inspired implementation* with two automated features:

i) *automated data movement* to gather and partition parameters right before and after they are required during the training. ZeRO-Infinity does this by injecting i) pre forward/backward hooks into PyTorch submodules that trigger allgather collectives to collect the parameters required before its forward/backward pass and ii) post forward/backward hooks that trigger parameter/gradient partiitoning and optionally offloading them to CPU or NVMe (see Sec. 7.1 for details).

ii) *automated model partitioning during initialization* such that models that can not fit within single GPU or CPU memory can still be initialized without requiring manual partitioning of the model across data parallel processes. ZeRO-Infinity achieves this by wrapping the constructor of all module classes so that parameters of each submodule are partitioned and offloaded immediately after they are created during initialization. The entire model is never fully instantiated on a single data parallel process (see Sec. 7.2 for details).

6 EFFICIENCY OPTIMIZATIONS

In this section, we deep dive into the optimizations introduced in Sec. 5 that allow ZeRO-Infinity to achieve excellent efficiency.

---

[6]CPU and NVMe bandwidth are in the order of 100 GB/s and 25 GB/s, respectively, but reading data from CPU or NVMe to a single GPU is limited by the achievable PCIe bandwidth which is around 10-12 GB/s

6.1 Bandwidth-Centric Partitioning

ZeRO-Infinity implements a novel data mapping and retrieval strategy to address the NVMe and CPU memory bandwidth limitations. Unlike ZeRO [11] and ZeRO-Offload [12], where parameters of each layer are owned by a single data parallel process, which broadcasts them to the rest when needed, ZeRO-Infinity partitions individual parameters across all the data parallel process, and uses an allgather instead of a broadcast when a parameter needs to be accessed. Note that both broadcast and allgather communication collectives have the same communication cost when it comes to data movement volume if the data is located on the GPU. Therefore, this makes no difference for a GPU-only training. However, this is a game changer when the data is located in NVMe or CPU.

In the broadcast-based approach, since each parameter is fully owned by one of the data parallel processes, the parameter must be first communicated from its source location to the GPU memory via the PCIe before the broadcast can happen. Note that only a single PCIe can be active for this process, while all the PCIe links connected to all the other GPUs are idle. On the contrary, with the partitioned parameter and allgather based approach in ZeRO-Infinity, all PCIe links are active in parallel, each bringing in $1/dp^{th}$ portion of the parameter where dp is the data parallel degree. As a result, the effective communication bandwidth between NVMe or CPU to the GPU, increases linearly with the dp degree.

For example, with broadcast-based approach, the CPU/NVMe to GPU bandwidth stays constant at about 12 GB/s with PCIe Gen 3, even with 16-way data parallelism on the DGX-2 box. However, with the all-gather-based approach, the effective achievable bandwidth increases to about 48/25 GB/s (3.0/1.6 GB/s per GPU), respectively (see Fig. 2b), limited only by the max aggregate PCIe bandwidth and max NVMe bandwidth per DGX-2 node. From here, the bandwidth grows linearly with more nodes. When training a massive model at massive scale, ZeRO-Infinity can therefore offer significantly more heterogeneous memory bandwidth than necessary (virtually unlimited) for the training to remain efficient. For example, on 64 DGX-2 nodes, ZeRO-Infinity has access to over 3TB/s of CPU memory bandwidth and over 1.5TB/s of NVMe bandwidth.

6.2 Overlap Centric Design

While ZeRO-Infinity can leverage sufficient heterogeneous memory bandwidth on a multi-node setup, the bandwidth can still be a bottleneck on a single GPU or single node setup. Even the GPU-GPU allgather communication has a big impact on efficiency when running with a small batch size (Fig. 3). Furthermore, accessing NVMe memory requires a three step process: i) read data from NVMe to CPU memory (nc-transfer), ii) copy the data from CPU memory to GPU memory (cg-transfer), iii) execute allgather to construct the full parameter on all GPUs (gg-transfer). The sequential nature of these data movements means that if done naively, the total communication time would be the sum of each of these three data movement cost, resulting in poor efficiency even if the bandwidth for data movement at each of these stages is individually sufficient.

To address these issues, ZeRO-Infinity has an overlap engine that not only overlaps GPU-GPU communication with GPU computation, but also overlaps the NVMe to CPU, and CPU to GPU communication, all at the same time. The overlap engine has two components: i) A dynamic prefetcher for overlapping the data movement required to reconstruct parameters before they are consumed in the forward or backward pass, and ii) a communication and offload overlapping mechanism for executing the data movement required by gradients in parallel with the backward computation.

The dynamic prefetcher in ZeRO-Infinity traces the forward and backward computation on that fly, constructing an internal map of the operator sequence for each iteration. During each iteration, the prefetcher keeps track of where it is in the operator sequence and prefetches the parameter requires by the future operators. The prefetcher is aware of the three step communication process, and therefore can overlap the nc-transfer for one parameter, with cg-transfer and gg-transfer of other parameters. For instance, before executing the ith operator, the prefetcher can invoke nc, cg, and gg-transfer for parameters required by $i+3$, $i+2$, and $i+1$ operators, respectively. Note that all of these data movement can happen in parallel with the execution of the ith operator. Furthermore, ZeRO-Infinity can update the operator sequence map in case of dynamic workflow, allowing for appropriate prefetching even when the forward and backward propagation changes across iterations.

Similarly, in the backward pass, ZeRO-Infinity can overlap the reduce-scatter for gradients of the parameters in $(i + 1)^{th}$ operator with the computation of the ith operator, while simultaneous transferring the partitioned gradients from the reduce-scatter of the gradients of the $(i + 2)^{th}$ operator to the CPU or NVMe.

With this powerful overlap centric design, ZeRO-Infinity hides significant portions of data movement even when training with a small number of GPUs and small batch size per GPU.

6.3 Infinity Offload Engine

The infinity offload engine is composed of two main components:

DeepNVMe, a powerful C++ NVMe read/write library in the infinity offload engine that supports bulk read/write requests for asynchronous completion, and explicit synchronization requests to flush ongoing read/writes. The support for asynchrony allows ZeRO-Infinity to overlap these requests with GPU/GPU or GPU/CPU communication or computation.

Most importantly, DeepNVMe is capable of achieving near peak sequential read and write bandwidths on the NVMe storage device. It achieves this high performance through a number of optimizations, including aggressive parallelization of I/O requests (whether from a single user thread or across multiple user threads), smart work scheduling, avoiding data copying, and memory pinning.

Pinned memory management layer To ensure high performance tensor reads (or writes) from (to) NVMe/CPU storage, the source (or destination) tensors must reside in pinned memory buffers. However, pinned memory buffers are scarce system resources, and their oversubscription by a single process can degrade overall system performance or cause system instability. This layer manages the limited supply of pinned memory by reusing a small amount (tens of GBs) for offloading the entire model states (up to tens of TBs) to CPU or NVMe. The reuse of memory buffer prevents memory fragmentation in CPU and GPU memory. This layer also provides PyTorch tensors with pinned memory data, allowing in-place computation of the tensors so that they can then be written to NVMe without any further copies to improve bandwidth.

7 EASE INSPIRED IMPLEMENTATION

ZeRO-Infinity is implemented on top of PyTorch, and is designed to be used without any model code refactoring similar to standard data-parallel training in PyTorch. This section details some of the challenges faced in implementing such a system.

7.1 Automating Data Movement

ZeRO-Infinity must coordinate the movement of tensors comprising the model parameters, gradients, and optimizer states. When a tensor is not in active use, it remains partitioned among workers and potentially offloaded to CPU or NVMe memory. The system must ensure that the tensors are resident in GPU memory in time for use and then later re-partitioned.

PyTorch models are expressed as a hierarchy of modules that represent the layers of a neural network. For example, Transformer architectures [38] contain submodules such as self-attention and feedforward networks. The self-attention submodules are further comprised of linear transformations and other submodules.

ZeRO-Infinity recursively injects hooks into the submodules of a model to automate the required data movement. At the start of a submodule's forward pass, these hooks ensure that the submodule's parameters are available for computations, otherwise it will execute the appropriate allgather collectives and block until the parmaeters become available. The overlap-centric design detailed in Sec. 6.2 is critical to minimizing stalls due to parameter communication. At the end of the submodule's forward pass, we partition the parameters again and optionally offload them. The backward pass is handled in a similar fashion.

7.1.1 Auto Registration of External Parameters. In the ideal case, a submodule's parameters and gradients are only accessed within its own forward and backward passes, making it straight forward to identify and automate the data movement as discussed in section above. However, some model architectures are exceptions, where parameters defined and allocated in a submodule is used in forward and backward propagation of a different submodule. For example, language models such as GPT [41] share the weights of the embedding layer at both the beginning and the end of the network to map words to vectors and vice versa. We refer to the parameters that are used across module boundaries as *external parameters*. In presence of external parameters, it is difficult to know which parameters to gather at the beginning of a submodule's forward and backward pass.

One way to address this is to *register* external parameters with ZeRO-Infinity so that they are collected for the forward and backward passes of the submodule that access them. After registration, an external parameter is treated like all others and will be included in the prefetching system as described in Sec. 6.2. We provide APIs for manual registration of external parameters.

In order to improve user experience, we also provide mechanisms to detect these scenarios and automatically register external parameters so the the user does not have to make any code change:

Intercepting partitioned parameter accesses PyTorch modules store their tensor parameters in a hash table. At the initialization time, we replace the hash table with a subclassed type that overrides the tensor accesses. When a partitioned parameter is accessed, we do a blocking allgather on the parameter, register it as an external parameter, and then return the gathered parameter.

Activation introspection A submodule may return a parameter from its forward pass to be consumed by another submodule's forward and backward passes. For example, Megatron-LM returns bias vectors from the forward pass of linear layers and they are consumed by the parent Transformer layer modules. We inspect the activation outputs returned from each submodule's forward pass for partitioned parameters. If one is discovered, we collect and register it as an external parameter.

7.2 Automatic Model Partitioning during Initialization

If the model is large, then it may not be possible to fully initialize the model with traditional data parallel approach, replicating it on each data parallel process before it can be partitioned for ZeRO-Infinity. For example, a 500 billion parameter model will occupy 1 TB of memory in half precision, and thus a system with 8 GPUs per node requires 8 TB of aggregate CPU or GPU memory just for the initial data parallel allocation step. This is beyond the GPU or CPU memory available on a node.

To address this limitation, the parameters corresponding to each layer of the model must be partitioned at the time of initialization, and not after the entire model is initialized. To do this, we provide a Python ZeRO-Infinity *context* which decorates the `__init__` method of `torch.nn.Module`, so that parameters allocated under each module/sub-module are partitioned immediately after its initialization among the group of data parallel processes.

As a result, only individual sub-modules are fully initialized before they are partitioned, and the full model is never replicated on all the data parallel process. In the example above, the 500 billion parameter model can therefore be fully partitioned during its initialization requiring only 1 TB of aggregate CPU memory regardless of the total number of data parallel process.

8 EVALUATION

This section evaluates ZeRO-Infinity, demonstrating that it achieves excellent training efficiency and scalability for models with tens of trillion parameters. We also show the impact of various technologies within ZeRO-Infinity on model scale and performance.

8.1 Methodology

Hardware. We conducted our experiments on a cluster of up to 512 V100 SXM3 32 GB GPUs (32 DGX-2 nodes) with 800 Gbps internode communication bandwidth.

Baseline. For experiments without model parallelism (mp), we use torch's distributed data parallel (DDP [42]) as a baseline. For experiments with model parallelism, we use Megatron-LM [7]. As a baseline for each experiment we use the relevant state-of-the-art method among 3D Parallelism [13], ZeRO [11], or ZeRO-Offload [12].

Model Configurations. We use GPT-like Transformer based models. We fix the sequence length to 1024 and vary the hidden dimension and number of layers to obtain models with different number of parameters. Table 1 provides the specific model configurations

# nodes	# params	hidden dim	# layers	batch/GPU	mp	fp16 param	Opt State
1	10 B	4K	50	8	1	GPU	GPU
1	50, 100 B	8K	62, 125	26, 24	1	CPU	NVMe
1	0.5, 1 T	18K, 25K	124, 128	8, 7	1	NVMe	NVMe
32	0.5, 1 T	18K, 25K	124, 128	7, 5	4	GPU	GPU
32	5, 10, 20 T	48K, 64K, 88K	174, 200, 205	3, 2, 1.25	4, 4, 8	NVMe	NVMe

Table 1: Experiment configurations. Sizes are expressed in B for billions, T for trillions, and K for 1024.

(a) ZeRO-Infinity efficiently trains 40x larger models than 3D parallelism on 512 GPUs.

(b) ZeRO-Infinity exceeds linear scaling from 64 to 512 GPUs for a 1T parameter model.

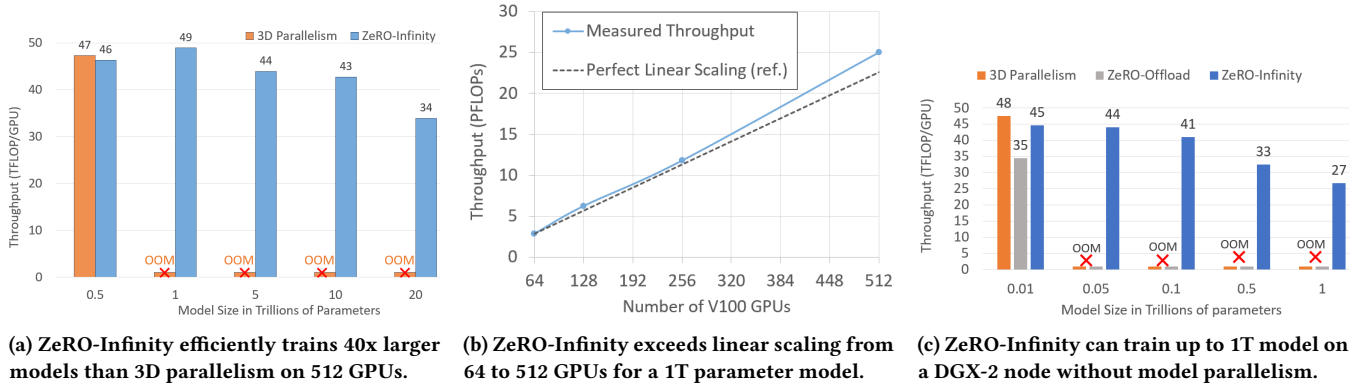(c) ZeRO-Infinity can train up to 1T model on a DGX-2 node without model parallelism.

Figure 5: Efficiency and scalability of ZeRO-Infinity for training multi-trillion parameter models.

used throughout our evaluation, (see Appendix A) for additional configurations.

8.2 Model Size and Speed

Model Size ZeRO-Infinity trains models over 32 *trillion* parameters compared to about 650B parameters with 3D parallelism, state of the art, offering a leap of 50x in model scale (Figure 1).

Model Speed Figure 5a shows the performance of ZeRO-Infinity on up to 20 trillion parameter models on 512 GPUs. For the 500B model (close to the largest that 3D parallelism can run on these resources), ZeRO-Infinity and 3D parallelism achieve nearly identical throughput, indicating ZeRO-Infinity is on par with the training efficiency of the state-of-the-art. When increasing the model size further, 3D parallelism simply runs out of memory, while ZeRO-Infinity trains up to 20 trillion parameter models (40x larger) with excellent throughput of up to 49 TFlops/GPU. At the extreme-scale, Figure 5a shows a performance drop from 10T (43 TFlops/GPU), and 20T (34 TFlops/GPU). This drop is not due to NVMe bandwidth as both model sizes use NVMe offload, but instead due to an extremely small batch size per GPU (Table 1) at 20T scale as a result of limited CPU memory to store activation checkpoints. This can be improved by increasing the CPU memory or offloading activation checkpoints to NVMe in a future implementation.

8.3 Superlinear Scalability

Figure 5b shows that ZeRO-Infinity achieves super-linear scalability from 4 nodes (64 GPUs) to 32 nodes (512 GPUs) when training a 1T model. This is a weak scaling result where we keep batch size per node as constant and increase total batch size with increased number of nodes. ZeRO-Infinity exceeds perfect linear scaling by effectively leveraging the linear increase in aggregate PCIe and

NVMe bandwidth to accelerate the offloading of parameters and optimizer states, and leveraging CPU compute from additional nodes for parameter update. In addition, ZeRO-Infinity already achieves over 2.8 petaflops (44 Tflops/GPU) with just 4 nodes, demonstrating that the aggregated NVMe bandwidth is sufficient to achieve good efficiency even at a modest scale.

8.4 Democratizing Large Model Training

Figure 5c shows performance of training 10B to 1T models on a single node (16 GPUs) with ZeRO-Infinity without any model parallelism. With models up to 100 billion parameters, ZeRO-Infinity achieves excellent performance of over 40 TFlops/GPU, making it possible to fine-tuning models such as GPT-3 with just a single DGX-2 box. In contrast, 3D parallelism is unable to scale to models with over 20 billion parameters.

These results demonstrate two aspects of ZeRO-Infinity: i) Accessibility to fine-tuning large models with up to a trillion parameters on a single NVIDIA DGX-2 node, empowering users without access to large GPU clusters. ii) Ease-of-Use: Models at this scale can be trained using ZeRO-Infinity without combining model or pipeline parallelism, or requiring model code refactoring, making it easy for data scientists to scale up their models.

8.5 Impact of System Features on Model Scale

We show the impact of different device placement strategies on model scale and impact of memory-centric tiling (Sec. 5.1.3) on maximum hidden size using a single DGX-2 system (16 GPUs).

Maximum model size Figure 6a shows the effect of different device placement and partitioning strategies (see Table 2) on maximum model size. By using data parallelism alone we're limited to only 1.4B parameters, due to limited GPU memory and significant
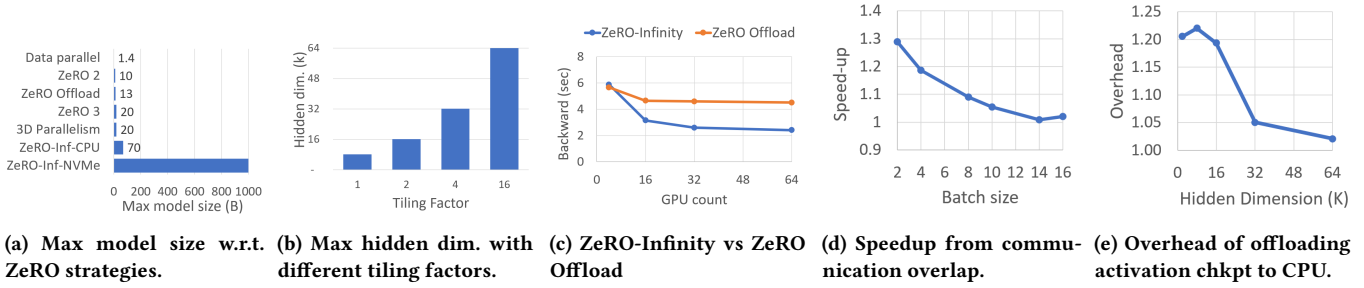
(a) Max model size w.r.t. ZeRO strategies. **(b) Max hidden dim. with different tiling factors.** **(c) ZeRO-Infinity vs ZeRO Offload** **(d) Speedup from communication overlap.** **(e) Overhead of offloading activation chkpt to CPU.**

Figure 6: Impact of system features on model scale and performance.

Name	Optimizer + Grad (devices/partitioned)	Parameters (devices/partitioned)
Data parallel	[GPU] / ✗	[GPU] / ✗
ZeRO 2	[GPU] / ✓	[GPU] / ✗
ZeRO-Offload	[CPU,GPU] / ✓	[GPU] / ✗
3D Parallelism	[GPU] / ✓	[GPU] / ✓
ZeRO 3	[GPU] / ✓	[GPU] / ✓
ZeRO-Inf-CPU	[CPU, GPU] / ✓	[CPU,GPU] / ✓
ZeRO-Inf-NVMe	[NVMe,CPU,GPU] / ✓	[NVMe,CPU,GPU] / ✓

Table 2: Device placement options and partitioning strategies for optimizer, gradient, and parameter states.

model state redundancies. As we introduce optimizer/gradient partitioning and offloading to CPU with ZeRO-2 and ZeRO-Offload, we are able to scale up 9x to 13B parameters on a single node. Partitioning and offloading parameter states to CPU in ZeRO-Infinity allows us to almost reach 100B parameters. However, the final major jump in scale comes from offloading model states to NVMe which finally gets us to 1T parameters, resulting in a 700x increase in model size relative to data parallelism alone.

Maximum Hidden Size We evaluate the impact of memory-centric tiling in enabling large hidden sizes in the presence of memory fragmentation. We train a single layer transformer model with different hidden sizes and tiling factors to identify the largest hidden size that can be trained with and without tiling. To keep memory fragmentation consistent across all the experiments, we pre fragment the total GPU memory into 2 GB contiguous chunks so that all memory allocation requests larger than 2GB will fail.

Figure 6b shows the largest hidden size that can be trained without memory-centric tiling is 8K, while we can even train a massive hidden size of 64K using memory-centric tiling factor of 16. With memory-centric tiling, ZeRO-Infinity greatly simplifies DL system stack by avoiding the need for model parallelism, making it easy for data scientists to train with large hidden sizes.

8.6 Impact of System Features on Performance

We evaluate the effects of the infinity offload engine (Sec. 5), bandwidth-centric partitioning (Sec. 6.1), overlap-centric design (Sec. 6.2), and activation checkpoint offloading (Sec. 4.1) on training speed.

ZeRO-Infinity vs ZeRO-Offload Figure 6c shows the impact of offloading gradients to CPU memory with ZeRO-Infinity vs

ZeRO-Offload on the back propagation time of an 8B parameter model. ZeRO-Infinity leverages the aggregate PCIe bandwidth across GPUs to offload the gradients, resulting in a speedup of nearly 2x at 64 GPUs compared to ZeRO-Offload which is limited by single PCIe bandwidth.

Prefetching and Overlapping Figure 6d shows the relative throughput difference with communication overlapping and prefetching turned on and off for an 8B parameter model with 64 GPUs. The figure shows that prefetching and overlapping are crucial to achieving good performance at small batch sizes per GPU, while its impact diminishes at large batch sizes.

Activation checkpoint offload Figure 6e shows that CPU offloading of activation checkpoints in ZeRO-Infinity reduces the training throughput by up to 1.2x for small hidden sizes, but for hidden sizes 32K and 64K, the impact is minimal, demonstrating that it is possible to offload activation checkpoints to CPU memory without impacting efficiency for large hidden sizes.

9 CONCLUSION & FUTURE IMPLICATIONS

	V100	10x	100x
Total devices	512	512	512
Achievable peak (pflops/device)	0.07	0.70	7.00
Slow memory bw requirement (GB/s per device)	3.0	30.0	300.0
Slow memory aggregate bw (TB/s)	1.5	15.0	150.0
GPU-to-GPU bw (GB/s)	70.0	700.0	7000.0

Table 3: Bandwidth (bw) requirements for ZeRO-Infinity to remain efficient on a cluster of 512 accelerator devices with 10x and 100x more achievable compute than NVIDIA V100 GPUs.

In this paper, we presented ZeRO-Infinity, a novel heterogeneous system technology that leverages GPU, CPU, and NVMe memory to allow for unprecedented model scale that is accessible and easy to use while achieving excellent efficiency. It offers a paradigm shift in how we think about memory for large model training. It is no longer necessary to fit DL training on ultra-fast but expensive and limited memory like HBM2. ZeRO-Infinity demonstrates that it is possible to transcend the GPU memory wall by leveraging cheap and slow, but massive, CPU or NVMe memory in parallel across multiple devices to achieve the aggregate bandwidth necessary for efficient training on current generation of GPU clusters.

As we look into the future, the GPUs and other accelerators will become more powerful, and this aggregate bandwidth required for

efficient training will also increase. Table 3 shows that even when the compute of the accelerators increases by 10x compared to the NVIDIA V100 GPUs, on a cluster with 512 of them, ZeRO-Infinity only requires a bandwidth of 30 GB/s between each accelerator and the slow memory to remain efficient. In fact, this is already possible with today's technology by connecting accelerators to slow memory via NVLink [43]. For example, the Summit Supercomputer launched in 2018 [44] connects NVIDIA V100 GPUs with the CPU memory at 40GB/s per GPU.

It is clear that with ZeRO-Infinity, accelerator device memory is no longer a limitation on model scale or training efficiency. However, training models with tens or hundreds of trillions of parameters in a reasonable time still requires massive leaps in compute, and running efficiently on these future devices requires a proportional leap in device-to-device bandwidth (Table 3).

We hope that, with device memory no longer a limitation, ZeRO-Infinity will inspire a more compute and device-device bandwidth focused innovations of ultra-powerful accelerator devices and super-computing clusters in the future to support the next 1000x growth in model scale and the advancements that they can offer.

ACKNOWLEDGEMENT

REFERENCES

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[2] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[3] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019.

[4] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[5] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[6] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[7] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019.

[8] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. Pipedream: Fast and efficient pipeline parallel DNN training. *CoRR*, abs/1806.03377, 2018.

[9] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *ArXiv*, abs/1811.06965, 2018.

[10] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. *arXiv preprint arXiv:2006.09503*, 2020.

[11] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[12] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyang Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training, 2021.

[13] Microsoft. DeepSpeed: Extreme-scale model training for everyone. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/, 2020.

[14] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Software repository. https://github.com/NVIDIA/Megatron-LM, 2021.

[15] DeepSpeed Team and Rangan Majumder. DeepSpeed: Extreme-scale model training for everyone. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/, 2020.

[16] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall. *RiseLab Medium Post*, 2021.

[17] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. *CoRR*, abs/1811.02084, 2018.

[18] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[20] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 875–890, 2020.

[21] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[22] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):1–26, 2018.

[23] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[24] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *IEEE International Symposium on High Performance Computer Architecture*, 2021.

[25] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[26] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018.

[27] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems, 2020.

[28] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *International Symposium on Computer Architecture (ISCA 2018)*, 2018.

[29] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[30] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *ArXiv*, abs/1910.02653, 2019.

[31] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic GPU memory management for training deep neural networks. *CoRR*, abs/1801.04380, 2018.

[32] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159, July 2011.

[33] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[34] Yang You, Igor Gitman, and Boris Ginsburg. Scaling SGD batch size to 32k for imagenet training. *CoRR*, abs/1708.03888, 2017.

[35] Yang You, Jing Li, Jonathan Hseu, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. Reducing BERT pre-training time from 3 days to 76 minutes. *CoRR*, abs/1904.00962, 2019.

[36] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2017.

[37] NVIDIA Tensor Cores. https://www.nvidia.com/en-us/data-center/tensor-cores/, 2018. [Online, accessed 5-April-2021].

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[39] turing-nlg: A 17-billion-parameter language model by microsoft.

[40] NVIDIA. NVIDIA DGX SuperPOD delivers world record supercomputing to any enterprise. https://developer.nvidia.com/blog/dgx-superpod-world-record-supercomputing-enterprise/, 2019.

[41] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[42] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[43] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[44] Oak Ridge National Laboratory. ORNL Launches Summit Supercomputer. https://www.ornl.gov/news/ornl-launches-summit-supercomputer, 2018. [Online; accessed 08-April-2021].

A APPENDIX

Figure 6(a)							
Model size	Number of GPUs	MP	Layers	Hidden Size	Attention head	Batch size	Total batch size
1.4B	16	1	40	1536	16	1	16
10B	16	1	50	4096	16	1	16
13B	16	1	64	4096	16	1	16
20B (ZeRO-3)	16	1	98	4096	32	1	16
20B (3D Par.)	16	4	98	4096	32	1	16
70B	16	1	125	8192	32	1	16
1000B	16	4	128	25600	256	5	20

Table 4: Model configurations for Figure 6(a)

Figure 6(b)							
Hidden size	Number of GPUs	MP	Layers	Model size	Attention head	Batch size	Total batch size
8192	16	1	1	900M	16	1	16
16384	16	1	1	3B	16	1	16
32768	16	1	1	13B	16	1	16
65536	16	1	1	50B	32	1	16

Table 5: Model configurations for Figure 6(b)

Figure 6(c)							
Number of GPUs	Hidden size	MP	Layers	Model size	Attention head	Batch size	Total batch size
[4,16,32,64]	8192	1	10	8B	16	2	[8,32,64,128]

Table 6: Model configurations for Figure 6(c)

Figure 6(d)							
Batch size	Number of GPUs	Hidden size	MP	Layers	Model size	Attention head	Total batch size
[2,4,8,10,14,16]	64	8192	1	10	8B	16	[128,256,512,640,896,1024]

Table 7: Model configurations for Figure 6(d)

Figure 6(e)								
Hidden size	Number of GPUs	Opt Device	MP	Layers	Model size	Attention head	Batch size	Total batch size
2048	32	CPU	1	5	275M	16	4	128
8192	32	CPU	1	5	4B	16	4	128
16384	32	CPU	1	5	16B	16	4	128
32768	32	CPU	1	5	64B	16	4	128
65536	64	NVMe	1	5	260B	16	4	128

Table 8: Model configurations for Figure 6(e)