ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

Samyam Rajbhandari*, Jeff Rasley*, Olatunji Ruwase, Yuxiong He
{samyamr, jerasley, olruwase, yuxhe}@microsoft.com

ABSTRACT

Large deep learning models offer significant accuracy gains, but training billions to trillions of parameters is challenging. Existing solutions such as data and model parallelisms exhibit fundamental limitations to fit these models into limited device memory, while obtaining computation, communication and development efficiency. We develop a novel solution, Zero Redundancy Optimizer (*ZeRO*), to optimize memory, vastly improving training speed while increasing the model size that can be efficiently trained. *ZeRO* eliminates memory redundancies in data- and model-parallel training while retaining low communication volume and high computational granularity, allowing us to scale the model size proportional to the number of devices with sustained high efficiency. Our analysis on memory requirements and communication volume demonstrates: *ZeRO* has the potential to scale beyond 1 *Trillion* parameters using today's hardware.

We implement and evaluate ZeRO: it trains large models of over 100B parameter with super-linear speedup on 400 GPUs, achieving throughput of 15 Petaflops. This represents an 8x increase in model size and 10x increase in achievable performance over state-of-the-art. In terms of usability, *ZeRO* can train large models of up to 13B parameters (e.g., larger than Megatron GPT 8.3B and T5 11B) without requiring model parallelism which is harder for scientists to apply. Last but not the least, researchers have used the system breakthroughs of *ZeRO* to create *Turing-NLG*, the world's largest language model at the time (17B parameters) with record breaking accuracy.

I. EXTENDED INTRODUCTION

Deep Learning (DL) models are becoming larger, and the increase in model size offers significant accuracy gain. In the area of Natural Language Processing (NLP), Transformers [1] have paved way for large models like Bert-large (0.3B) [2], GPT-2 (1.5B) [3], Megatron-LM (8.3B) [4], T5 (11B) [5]. To enable the continuation of model size growth from 10s of billions to trillions of parameters, we experience the challenges of training them - they clearly do not fit within the memory of a single device, e.g., GPU or TPU, and simply adding more devices will not help scale the training.

Basic data parallelism (DP) does not reduce memory per device, and runs out of memory for models with more than 1.4B parameters on GPUs with 32 GB memory when trained using common settings like mixed precision and ADAM optimizer [6]. Other existing solutions such as Pipeline Parallelism (PP), Model Parallelism (MP), CPU-Offloading, etc, make trade-offs between functionality, usability, as well as memory and compute/communication efficiency, all of which are crucial to training with speed and scale.

Among different existing solution for training large models, MP is perhaps the most promising one. The largest models in the current literature, the 11B T5 model [5], and Megatron-LM 8.3B [4], were both powered by model parallelism, implemented in Mesh-Tensorflow [7] and Megatron-LM[4], respectively. However, MP cannot scale much further beyond these models sizes. MP splits the model vertically, partitioning the computation and parameters in each layer across multiple devices, requiring significant communication between each layer. As a result, they work well within a single node where the inter-GPU communication bandwidth is high, but the efficiency degrades quickly beyond a single node [4]. We tested a 40B parameter model using Megatron-LM across two DGX-2 nodes and observe about $5Tflops$ per V100 GPU (less than 5% of hardware peak) compared to near $30Tflops$ for models that can be trained within a node.

So, how can we overcome the limitations of existing solutions and train large models more efficiently? To answer this question, we first analyze the full spectrum of memory consumption of the existing systems on model training and classify it into two parts: 1) For large models, the majority of the memory is occupied by *model states* which include the optimizer states (such as momentum and variances in Adam [6]), gradients, and parameters. 2) The remaining memory is consumed by activation, temporary buffers and unusable fragmented memory, which we refer to collectively as *residual* states. We develop *ZeRO*— Zero Redundancy Optimizer — to optimize memory efficiency on both while obtaining high compute and communication efficiency. As these two parts face different challenges, we develop and discuss their solutions correspondingly.

Optimizing Model State Memory Model states often consume the largest amount of memory during training, but existing approaches such as DP and MP do not offer a satisfying solution. DP has good compute/communication efficiency but poor memory efficiency while MP can have poor compute/communication efficiency. More specifically, DP replicates the entire model states across all data parallel process resulting in redundant memory consumption; while MP partition these states to obtain high memory efficiency,

*Equal Contribution

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00

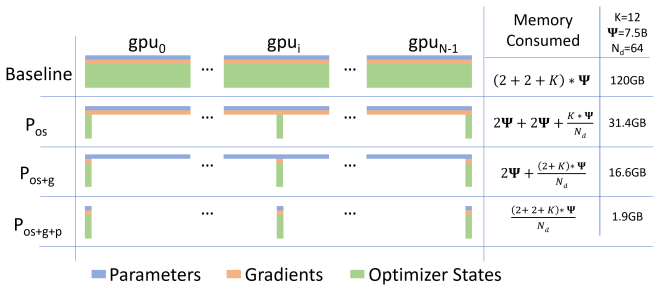

Fig. 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

but often result in too fine-grained computation and expensive communication that is less scaling efficient. Furthermore, all of these approaches maintain all the model states required over the entire training process statically, even though not all model states are required all the time during the training. Based on these observations, we develop *ZeRO*-DP, ZeRO-powered data parallelism, that achieves the computation/communication efficiency of DP while achieving memory efficiency of MP. *ZeRO*-DP removes the memory state redundancies across data-parallel processes by *partitioning* the model states instead of replicating them, and it retains the compute/communication efficiency by retaining the computational granularity and communication volume of DP using a dynamic communication schedule during training.

ZeRO-DP has three main optimization stages (as depicted in Figure 1), which correspond to the partitioning of optimizer states, gradients, and parameters (Sec. V). When enabled cumulatively:

1) Optimizer State Partitioning (P_{os}): 4x memory reduction, same communication volume as DP;

2) Add Gradient Partitioning (P_{os+g}): 8x memory reduction, same communication volume as DP;

3) Add Parameter Partitioning (P_{os+g+p}): Memory reduction is linear with DP degree N_d. For example, splitting across 64 GPUs (N_d = 64) will yield a 64x memory reduction. There is a modest 50% increase in communication volume.

ZeRO-DP eliminates memory redundancies and makes the full aggregate memory capacity of a cluster available. With all three stages enabled, ZeRO can train a trillion-parameter model on just 1024 NVIDIA GPUs. A trillion-parameter model with an optimizer like Adam [6] in 16-bit precision requires approximately 16 terabytes (TB) of memory to hold the optimizer states, gradients, and parameters. 16TB divided by 1024 is 16GB, which is well within a reasonable bound for a GPU (e.g., with 32GB of on-device memory).

Optimizing Residual State Memory After *ZeRO*-DP boosts memory efficiency for model states, the rest of the memory consumed by activations, temporary buffers, and unusable memory fragments could become a secondary memory bottleneck. We develop *ZeRO*-R to optimize the residual memory consumed by these three factors respectively.

1) For activations (stored during the forward pass in order to perform backward pass), we noticed checkpointing [8] helps but is insufficient for large models. Thus *ZeRO*-R optimizes activation memory by identifying and removing activation replication in existing MP approaches through activation partitioning. It also offloads activations to CPU when appropriate.

2) *ZeRO*-R defines appropriate size for temporary buffers to strike for a balance of memory and computation efficiency.

3) We observe fragmented memory during training due to variations in the lifetime of different tensors. Lack of contiguous memory due to fragmentation can cause memory allocation failure, even when enough free memory is available. *ZeRO*-R proactively manages memory based on the different lifetime of tensors, preventing memory fragmentation.

ZeRO-DP and *ZeRO*-R combined together forms a powerful system of memory optimizations for DL training that we collectively refer to as *ZeRO*.

***ZeRO* and MP**: Since *ZeRO* eliminates the memory inefficiency in DP, it is natural to ask: Do we still need MP, and when? How does *ZeRO* work with MP? With *ZeRO*, MP becomes a less attractive option for the purpose of fitting large models alone. *ZeRO*-DP is at least as effective on reducing per-device memory footprint as MP, or more effective sometimes when MP cannot divide the model evenly. It also has comparable or better scaling efficiency. Furthermore, data parallelism is so easy to use that it is widely applicable across different workloads, while MP approaches today often need some work from model developers to revise their model, system developers to work out distributed operators, and existing work like Megatron-LM only supports a limited set of operators and models.

That being said, there are still cases where we want to leverage MP: i) When used with *ZeRO*-R, MP can reduce activation memory footprint for very large models. ii) For smaller models where activation memory is not an issue, MP can also have benefits when aggregated batch size using DP alone is too big to have good convergence.[1] In those case, one can combine *ZeRO* with MP to fit the model with an acceptable aggregated batch size.

We show that *ZeRO* can be combined with MP, resulting in a max theoretical memory reduction of $N_d \times N_m$ times on each device with a DP degree of N_d and MP degree of N_m. This could allow us to fit a trillion parameter model on 1024 GPUs with 16-way model parallelism (within each DGX2 node) and 64-way data parallelism across nodes, and run it efficiently using a modest batch size.

Implementation & Evaluation The complete set of optimizations in *ZeRO* could allow us to run models with trillion

[1]Prior work [9] shows, very large batch size could slow down convergence. For given model and data, there is a measure of critical-batch size, where increasing batch size further slows down convergence. The detailed discussion of this topic is beyond the scope of the paper.

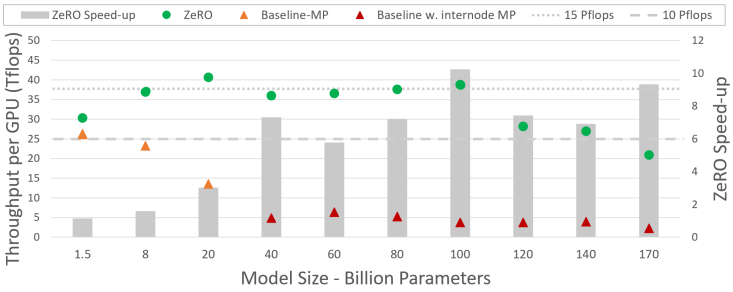

Fig. 2: *ZeRO* training throughput and speedup w.r.t SOTA baseline for varying model sizes. For *ZeRO*, the MP always fit in a node, while for baseline, models larger than 20B require MP across nodes.

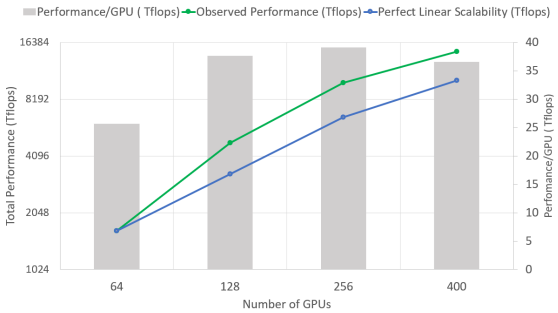

Fig. 3: Superlinear scalability and per GPU training throughput of a 60B parameter model using *ZeRO*-100B.

parameters on the high-end hardware cluster today (e.g., with 1024 V100 GPUs), however, the hardware compute capacity is still too limited and training time can be impractically long (>1 year). Therefore, our focus for this implementation is to efficiently support models with 10x parameters (∼100B parameters) than state-of-the-art (SOTA) while still being within reach of the compute capabilities of current hardware. We implement and evaluate a subset of optimizations in *ZeRO* called *ZeRO*-100B — P_{os+g} of *ZeRO*-DP plus ZeRO-R — that allow us to achieve this goal. The results show:

Model Size Combined with MP, *ZeRO*-100B runs 170B parameter models efficiently, while the existing system like using Megatron alone cannot scale efficiently beyond 20B parameters, as shown in Figure 2. This is an over 8x increase in model size compared to SOTA.

Speed Improved memory efficiency powers higher throughput and faster training. As shown in Figure 2, *ZeRO* runs 100B parameter models on a 400 Nvidia V100 GPU cluster with over 38 TFlops per GPU, and aggregate performance over 15 Petaflops. This is more than 10x improvement in training speed compared to SOTA for the same model size.

Scalability We observe super linear speedup in the regime of 64-400 GPUs, where the performance more than doubles when we double the number of GPUs. This is a property of *ZeRO*-DP which reduces the memory footprint of the model states as we increase the DP degree allowing us to fit larger batch sizes per GPU resulting in better performance. We expect this behaviour to continue further as we increase the number of GPUs beyond 400.

Democratization of Large Model Training *ZeRO*-100B powers data scientist to train models with up to 13B parameters without any MP or PP that requires model refactoring, where 13B is more parameters than the largest model in literature (T5 with 11B parameters). Data scientists can thus experiment freely with large models without worrying about parallelism. In comparison, existing systems (e.g., PyTorch Distributed Data Parallel) runs out of memory with 1.4B parameter models.

New SOTA Model *ZeRO* powers *Turing-NLG* [10], the largest language model at the time with 17B parameters and record-breaking accuracy.

We share *ZeRO* as a part of our open source DL training optimization library — DeepSpeed[2]. We have released all implementations described in this paper and we will extend it further to support 1 trillion parameters by enabling *ZeRO*-DP stage 3 partitioning parameters (P_{os+g+p}). We make *ZeRO* fully accessible to the DL community to catalyze the evolution and democratization of large model training at scale. While we evaluate *ZeRO* in the context of NLP in this paper, our approach is model agnostic and is applicable across different tasks and DL domains.

II. Related Work

A. Data, Model and Pipeline Parallelism

Parallelization is a key strategy on training large models at scale. For a model that fits in the device memory for training, data parallelism (DP) is used to scale training to multiple devices. In DP, model parameters are replicated on each device. At each step, a mini-batch is divided evenly across all the data parallel processes, such that each process executes the forward and backward propagation on a different subset of data samples, and uses averaged gradients across processes to update the model locally.

When a model does not fit in the device memory, model parallelism (MP) [7], [4], [11] and pipeline parallelism (PP) [12], [13] split the model among processes, in vertical and horizontal way respectively. Sec. I discussed how *ZeRO* relates to DP and MP. We now discuss PP and how it relates to reducing memory consumption.

PP splits a model horizontally across layers running each partition on a different device and use micro-batching to hide the pipeline bubble [12], [13]. Model functionalities such as tied-weights and batch-normalization are difficult to implement due to horizontal splitting and micro-batching, respectively. Popular PP implementation such as G-pipe [12] partitions both model parameters and total activations but requires a batch size proportional to number of pipeline partitions to hide the pipeline bubble. The large batch size can affect the convergence rate, while also requiring significant memory to store activations. A different implementation of PP in PipeDream [14] keeps multiple copies of stale parameters

[2]GitHub Repository: https://github.com/microsoft/deepspeed, DeepSpeed Website: https://www.deepspeed.ai/

to hide the pipeline bubble without increasing the batch size significantly, making it less memory efficient. Additionally, the implementation is not equivalent to the standard DL training and has implications on training convergence. In contrast, *ZeRO* obtains the same or better memory efficiency than PP without incurring functionality, performance and convergence related restrictions of PP.

B. Non-parallelism based approach to reduce memory

In addition to MP and PP, there are multiple lines of work that target reducing memory overheads of DL training.

1) Reducing Activation Memory: Multiple efforts have focused on reducing the memory footprint of activations through compression [15], activation checkpointing [8], [16], or live analysis [17]. These efforts are complimentary and can work together with *ZeRO*. In fact, activation memory reduction in *ZeRO*-R works in parallel with activation checkpointing.

2) CPU Offload: [18], [19] exploit heterogeneous nature of today's compute nodes, offloading model states to CPU memory through algorithmic design, virtualized memory, respectively. Up to 50% of training time can be spent on GPU-CPU-GPU transfers [18]. *ZeRO* differs in that it reduces the memory consumption significantly without storing the model states to CPU memory whose bandwidth is severely constrained due to PCI-E. On rare cases, *ZeRO*-R may offload just the activation checkpoints for very large models to improve performance (see Sec. VI-A for details). [20] uses graph rewriting to offload activations to CPU memory, but does not reduce memory required by model states.

3) Memory Efficient Optimizer: [21], [22] focus on reducing memory consumption of adaptive optimization methods by maintaining coarser-grained statistics of model parameters and gradients, with potential impact on model convergence guarantees. *ZeRO* is orthogonal to these efforts, and its optimizations do not change the model optimization method or affect model convergence, but effectively reduce memory footprint of optimizer states and gradients per device.

C. Training Optimizers

Adaptive optimization methods [23], [6], [24], [25] 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. *ZeRO* can reduce the memory footprint of these optimizers by orders of magnitude, making these sophisticated optimization methods practical for training large models on hardware with modest device memory. It also makes it possible to develop and use even more complex and memory hungry optimizers that may have better convergence.

III. Where Did All the Memory Go?

Let's take a step back to examine the memory consumption of the current training system. For example, a 1.5B parameter GPT-2 model requires 3GB of memory for its weights (or parameters) in 16-bit precision, yet, it cannot be trained on a single GPU with 32GB memory using Tensorflow or PyTorch. One may wonder where all the memory goes. During model training, most of the memory is consumed by *model states*, i.e., tensors comprising of optimizer states, gradients, and parameters. Besides these model states, the rest of the memory is consumed by activations, temporary buffers and fragmented memory which we call *residual states*. We look at the memory consumption from both in details.

A. Model States: Optimizer States, Gradients and Parameters

The majority of the device memory is consumed by model states during training. Consider for instance, Adam [6], one of the most popular optimizers for DL training. Adam requires storing two optimizer states, i) the time averaged momentum and ii) variance of the gradients to compute the updates. Therefore, to train a model with ADAM, there has to be enough memory to hold a copy of both the momentum and variance of the gradients. In addition, there needs to be enough memory to store the gradients and the weights themselves. Of these three types of the parameter-related tensors, the optimizer states usually consume the most memory, specially when mixed-precision training is applied.

Mixed-Precision Training The state-of-the-art approach to train large models on the current generation of NVIDIA GPUs is via mixed precision (fp16/32) training [26], where parameters and activations are stored as fp16, enabling the use of the high throughput tensor core units [27] on these GPUs. During mixed-precision training, both the forward and backward propagation are performed using fp16 weights and activations. However, to effectively compute and apply the updates at the end of the backward propagation, the mixed-precision optimizer keeps an fp32 copy of the parameters as well as an fp32 copy of all the other optimizer states.

Let's take Adam as a concrete example. Mixed precision training of a model with Ψ parameters using Adam requires enough memory to hold an $fp16$ copy of the parameters and the gradients, with memory requirements of 2Ψ and 2Ψ bytes respectively. In addition, it needs to hold the optimizer states: an $fp32$ copy of the parameters, momentum and variance, with memory requirements of 4Ψ, 4Ψ, and 4Ψ bytes, respectively. Let's use K to denote the memory multiplier of the optimizer states, i.e., the additional memory required to store them is $K\Psi$ bytes. Mixed-precision Adam has $K = 12$. In total, this results in $2\Psi+2\Psi+K\Psi = 16\Psi$ bytes of memory requirement. For a model such as GPT-2 with 1.5 Billion parameters, this leads to a memory requirement of at least $24\,GB$, which is significantly higher than the meager $3\,GB$ of memory required to hold the $fp16$ parameters alone.

B. Residual Memory Consumption

Activations can take up a significant amount of memory [8] during training. As a concrete example, the 1.5B parameter GPT-2 model trained with sequence length of 1K and batch

size of 32 requires about 60 GB of memory[3]. Activation checkpointing (or activation recomputation) is a common approach to reduce the activation memory by approximately the square root of the total activations at the expense of 33% re-computation overhead [8]. This would reduce the activation memory consumption of this model to about 8 GB.

Despite the significant reduction, the activation memory can grow quite large for bigger models even with activation checkpointing. For example, a GPT-like model with 100 billion parameters requires around 60 GB of memory for batch size 32, even when using activation checkpointing.

Temporary buffers used for storing intermediate results consumes non-trivial amount of memory for large models. Operations such as gradient all-reduce, or gradient norm computation tend to fuse all the gradients into a single flattened buffer before applying the operation in an effort to improve throughput. For example, the bandwidth of all-reduce across devices improves with large message sizes. While the gradient themselves are usually stored as fp16 tensors, the fused buffer can be an fp32 tensor depending on the operation. When the size of the model is large, these temporary buffer sizes are non-trivial. For example, for a model with 1.5B parameters, a flattened fp32 buffer would required $6\,GB$ of memory.

Memory Fragmentation: So far we have discussed the actual memory consumption during training. Additionally, it is possible to run out of usable memory even when there is plenty of available memory. This can happen with memory fragmentation. A request for a memory will fail if there isn't enough contiguous memory to satisfy it, even if the total available memory is larger than requested. We observe significant memory fragmentation when training very large models, resulting in out of memory issue with over 30% of memory still available in some extreme cases.

IV. *ZeRO*: Insights and Overview

ZeRO has two sets of optimizations: i) *ZeRO*-DP aimed at reducing the memory footprint of the model states, and ii) *ZeRO*-R targeted towards reducing the residual memory consumption. We present an overview of the optimizations and the insights behind, which allows *ZeRO* to reduce memory footprint while remaining efficient. Please note efficiency is a key here: without this constraint, trivial solutions like moving all the parameter states to the CPU memory, or increasing the MP degree arbitrarily can reduce memory footprint.

A. Insights and Overview: ZeRO-*DP*

ZeRO powered DP is based on three key insights:

a) DP has better scaling efficiency than MP because MP reduces the granularity of the computation while also increasing the communication overhead. Beyond a certain point, lower computational granularity reduces the efficiency per GPU, while the increased communication overhead, hiders the scalability across GPUs, especially when crossing node boundaries. On the contrary, DP has both higher computational granularity and lower communication volume, allowing for much higher efficiency.

b) DP is memory inefficient as model states are stored redundantly across all data-parallel processes. On the contrary, MP partitions the model states to obtain memory efficiency.

c) Both DP and MP keep all the model states needed over the entire training process, but not everything is required all the time. For example, parameters corresponding to each layer is only needed during the forward propagation and backward propagation of the layer.

Based on these insights, *ZeRO*-DP retains the training efficiency of DP while achieving the memory efficiency of MP. *ZeRO*-DP *partitions* the model states instead of replicating them (Section V) and uses a dynamic communication schedule that exploits the intrinsically temporal nature of the model states while minimizing the communication volume (Section VII). By doing so, *ZeRO*-DP reduces per-device memory footprint of a model *linearly* with the increased DP degree while maintaining the communication volume close to that of the default DP, retaining the efficiency.

B. Insights and Overview: ZeRO-*R*

1) Reducing Activation Memory: Two key insights are:

a) MP partitions the model states but often requires replication of the activation memory. For example, if we split the parameters of a linear layer vertically and compute them in parallel across two GPUs, each GPU requires the entire activation to compute its partition

b) For models such as GPT-2 or larger, the arithmetic intensity (ratio of the amount of computation per iteration to amount of activation checkpoints per iteration) is very large ($\geq 10K$) and increases linearly with hidden dimension making it possible to hide the data-movement cost for the activation checkpoints, even when the bandwidth is low.

ZeRO removes the memory redundancies in MP by *partitioning* the activations checkpoints across GPUs, and uses all-gather to reconstruct them on demand. The activation memory footprint is reduced proportional to the MP degree. For very large models, *ZeRO* can even choose to move the activation partitions to the CPU memory, while still achieving good efficiency due to large arithmetic intensity in these models.

2) Managing Temporary buffers: ZeRO-R uses constant size buffers to avoid temporary buffers from blowing up as the model size increases, while making them large enough to remain efficient.

3) Managing fragmented Memory: Memory fragmentation is a result of interleaving between short lived and long lived memory objects. During the forward propagation activation checkpoints are long lived but the activations that recomputed are short lived. Similarly, the backward computation, the activation gradients are short lived while the parameter gradients are long lived. Based on this insight, *ZeRO* performs on-the-fly memory defragmentation by moving activation checkpoints and gradients to pre-allocated contiguous memory buffers.

[3]The activation memory of a transformer-based model is proportional to the number of transformer layers × hidden dimensions × sequence length × batch size. For a GPT-2 like architecture the total activations is about $12 \times hidden\_dim \times batch \times seq\_length \times transformer\_layers$.

```
for_parallel rank in range(world_size):
  allocate_layers()
  for batch, label in dataset:
    x = forward(batch)
    backward(compute_loss(x, label))
    step()

def _is_owner(i):
  return True if rank owns i else False

# Only allocate device memory for layers owned by rank
def allocate_layers():
  for i in range(num_layers):
    ALLOCATE layers[i].param if _is_owner(i) else pass

def forward(x):
  for i in range(num_layers):
    BROADCAST(layers[i].param, src_rank=_owner_rank(i))
    x = layers[i].forward(x)
    del layers[i].param if not _is_owner(i) else pass
  return x

def backward(loss):
  dx = loss.backward()
  for i in range(num_layers, 0, -1):
    BROADCAST(layers[i].param, src_rank=_owner_rank(i))
    dx = layers[i].backward(dx)
    REDUCE(layers[i].grad, dst_rank=_owner_rank(i))
    if not _is_owner(i):
      del layers[i].grad, layers[i].param

def step():
  for i in range(num_layers):
    layers[i].step() if _is_owner(i) else pass
```

Fig. 4: Pseudo-code representing *ZeRO*-DP with all three phases, P_{os+g+p}, enabled.

This not only increases memory availability but also improves efficiency by reducing the time it takes for the memory allocator to find free contiguous memory.

V. Deep Dive into *ZeRO*-DP

While the existing DP approach replicates the model states at each device and introduces significant memory overhead, *ZeRO*-DP eliminates this memory redundancy by partitioning them — optimizer states, gradients and parameters — across data parallel processes. Figure 1 quantifies and visualizes the memory requirement with and without *ZeRO*-DP. The figure shows the memory footprint after partitioning (1) optimizer state, (2) gradient and (3) parameter redundancies accumulatively. We refer to them as the three optimization phases of *ZeRO*-DP: P_{os}, P_g, and P_p, which we elaborate below.

A. P_{os} : Optimizer State Partitioning

For a DP degree of N_d, we group the optimizer states into N_d equal partitions, such that the i^{th} data parallel process only updates the optimizer states corresponding to the i^{th} partition. Thus, each data parallel process only needs to store and update $\frac{1}{N_d}$ of the total optimizer states and then only update $\frac{1}{N_d}$ of the parameters. We perform an all-gather across the data parallel process at the end of each training step to get the fully updated parameters across all data parallel process.

DP	7.5B Model (GB)			128B Model (GB)			1T Model (GB)		
	P_{os}	P_{os+g}	P_{os+g+p}	P_{os}	P_{os+g}	P_{os+g+p}	P_{os}	P_{os+g}	P_{os+g+p}
1	120	120	120	2048	2048	2048	16000	16000	16000
4	52.5	41.3	**30**	896	704	512	7000	5500	4000
16	35.6	**21.6**	7.5	608	368	128	4750	2875	1000
64	**31.4**	16.6	1.88	536	284	**32**	4187	2218	250
256	30.4	15.4	0.47	518	263	8	4046	2054	62.5
1024	30.1	15.1	0.12	513	257	2	4011	2013	**15.6**

TABLE I: Per-device memory consumption of different optimizations in *ZeRO*-DP as a function of DP degree . Bold-faced text are the combinations for which the model can fit into a cluster of 32GB V100 GPUs.

Memory Savings: As shown in Figure 1, the memory consumption after optimizing state partition reduces from $4\Psi + K\Psi$ to $4\Psi + \frac{K\Psi}{N_d}$. As the concrete example depicted in Figure 1, a 7.5 B parameter model requires 31.4GB of memory using P_{os} with 64-way DP ($N_d = 64$), while requiring 120 GB with standard DP. Furthermore, when N_d is large, the memory requirement on model states reduces from $4\Psi + 12\Psi = 16\Psi$ bytes to $4\Psi + \frac{12\Psi}{N_d} \approx 4\Psi$ bytes, leading to a 4x reduction.

B. P_g: Gradient Partitioning

As each data parallel process only updates its corresponding parameter partition, it only needs the reduced gradients for the corresponding parameters. Therefore, as each gradient of each layer becomes available during the backward propagation, we only reduce them on the data parallel process responsible for updating the corresponding parameters. After the reduction we no longer need the gradients and their memory can be released. This reduces the memory footprint required to hold the gradients from 2Ψ bytes to $\frac{2\Psi}{N_d}$.

Effectively this is a Reduce-Scatter operation, where gradients corresponding to different parameters are reduced to different process. To make this more efficient in practice, we use a bucketization strategy, where we bucketize all the gradients corresponding to a particular partition, and perform reduction on the entire bucket at once. This is similar in spirit to how NVIDIA's AMP [28] optimizer bucketizes the all-reduce gradient computation to overlap communication and computation. In our case we perform a reduction instead of an all-reduce at the partition boundaries to reduce memory footprint and overlap computation and communication.

Memory Savings: By removing both gradient and optimizer state redundancy, we reduce the memory footprint further down to $2\Psi + \frac{14\Psi}{N_d} \approx 2\Psi$. As the example in Figure 1, a 7.5 B parameter model requires only 16.6 GB of memory using P_{os+g} with 64-way DP ($N_d = 64$), while requiring 120 GB with standard DP. When N_d is large, the memory requirement of model states reduces from $2\Psi + 14\Psi = 16\Psi$ bytes to $2\Psi + \frac{14\Psi}{N_d} \approx 2\Psi$ bytes, leading to a 8x reduction.

C. P_p: Parameter Partitioning

Just as with the optimizer states, and the gradients, each process only stores the parameters corresponding to its partition. When the parameters outside of its partition are required for forward and backward propagation, they are received from the appropriate data parallel process through broadcast. While

this may seem to incur significant communication overhead at first glance, we show that this approach only increases the total communication volume of a baseline DP system to 1.5x, while enabling memory reduction proportional to N_d. Therefore, the memory savings using P_{os+p+g} is equivalent to the memory savings from MP, however at a fraction of the communication volume as elaborated in Sec. VII-C.

Memory Savings: With parameter partitioning, we reduce the memory consumption of an Ψ parameter model from 16Ψ to $\frac{16\Psi}{N_d}$. As the example in Figure 1, a 7.5 B parameter model requires 1.9 GB of model-state memory using P_{os+p+g} with 64-way DP ($N_d = 64$), while requiring 120 GB with standard DP. This has a profound implication: ZeRO *powers DP to fit models with arbitrary size— as long as there are sufficient number of devices to share the model states*. We show the pseudo-code for training a model with P_{os+p+g} in Figure. 4.

D. Implication on Model Size

The three phases of partitioning P_{os}, P_{os+g}, and P_{os+g+p} reduces the memory consumption of each data parallel process on model states by up to 4x, 8x, and N_d respectively. Table I analyzes model-state memory consumption of a few example models under the 3 stages of *ZeRO*-DP optimizations for varying DP degree. Without *ZeRO*, the memory consumption is equal to the first row in the table, regardless of the DP degree. Note that, with $N_d = 64$, *ZeRO* can train models with up to 7.5B, 14B, and 128B parameters using P_{os}, P_{os+g}, and P_{os+g+p}, respectively. When $N_d = 1024$, *ZeRO* with all of its optimizations enabled (P_{os+g+p}) could train models with 1 trillion parameters. Without *ZeRO*, the largest model DP alone can run has less than 1.5 Billion parameters.

VI. Deep Dive into *ZeRO*-R

A. P_a: Partitioned Activation Checkpointing

As discussed in IV-B, MP by design requires a replication of the activations, resulting in redundant copies of the activations across model parallel GPUs. *ZeRO* eliminates this redundancy by partitioning the activations, and only materializes them in a replicated form one activation layer at a time, right before the activation is used in computation. More specifically, once the forward propagation for a layer of a model is computed, the input activations are partitioned across all the model parallel process, until it is needed again during the backprogation. At this point, *ZeRO* uses an all-gather operation to re-materialize a replicated copy of the activations. We refer to this optimization as P_a. It works in conjunction with activation checkpointing [8], storing partitioned activation checkpoints only instead of replicated copies. Furthermore, in the case of very large models and very limited device memory, these partitioned activation checkpoints can also be offloaded to the CPU reducing the activation memory overhead to nearly zero at an additional communication cost, which we will discuss in VII. We refer to this as P_{a+cpu}.

Memory Saving With partitioned activation checkpointing, *ZeRO* reduces the activation footprint by a factor proportional to the MP degree. Consider training a 100B model shown in Table IV with a batch size of 32, sequence length of 1024 and a MP degree of 16. If we checkpoint a single activation for each transformer layer, it would require about 33 GB of memory per GPU just to store the activation checkpoints. But with P_a in *ZeRO*, it can be reduced to about 2 GB per GPU. Furthermore, this 2GB can be offloaded to the CPU reducing the memory footprint for activations to nearly zero.

B. C_B: Constant Size Buffers

ZeRO carefully selects the sizes of the temporal-data buffers to balance memory and compute efficiency. During training, the computational efficiency of some operations can be highly dependent on the input size, with larger inputs achieving higher efficiency. For example, a large all-reduce operation achieves much higher bandwidth than a smaller one. Hence, to get better efficiency, high performance libraries such as NVIDIA Apex or Megatron fuses all the parameters into a single buffer before applying these operations. However, the memory overhead of the fused buffers is proportional to the model size, and can become inhibiting. For example, for a 3B parameter model, a 32-bit fused buffer will require 12 GB of memory. To address this issue, we use a performance-efficient constant-size fused buffer when the model becomes too large, similar to [29]. By doing so, the buffer size does not depend on the model size, and by keeping the buffer size large enough, we can still achieve good efficiency.

C. M_D: Memory Defragmentation

Memory fragmentation in model training occurs as a result of activation checkpointing and gradient computation. During the forward propagation with activation checkpointing, only selected activations are stored for back propagation while most activations are discarded as they can be recomputed again during the back propagation. This creates an interleaving of short lived memory (discarded activations) and long lived memory (checkpointed activation), leading to memory fragmentation. Similarly, during the backward propagation, the parameter gradients are long lived, while activation gradients and any other buffers required to compute the parameter gradients are short lived. Once again, this interleaving of short term and long term memory causes memory fragmentation.

Limited memory fragmentation is generally not an issue, when there is plenty of memory to spare, but for large model training running with limited memory, memory fragmentation leads to two issues, i) OOM due to lack of contiguous memory even when there is enough available memory, ii) poor efficiency as a result of the memory allocator spending significant time to search for a contiguous piece of memory to satisfy a memory request.

ZeRO does memory defragmentation on-the-fly by pre-allocating contiguous memory chunks for activation checkpoints and gradients, and copying them over to the pre-allocated memory as they are produced. M_D not only enables *ZeRO* to train larger models with larger batch sizes, but also improves efficiency when training with limited memory.

VII. Communication Analysis of *ZeRO*-DP

As *ZeRO* boosts model size by removing memory redundancy, it is only natural to ask if we are trading communication volume for memory efficiency. In other words, what is the communication volume of *ZeRO*-powered DP approach compared to a baseline DP approach? The answer is in two parts: i) *ZeRO*-DP incurs no additional communication using P_{os} and P_g, while enabling up to 8x memory reduction, ii) *ZeRO*-DP incurs a maximum of 1.5x communication when using P_p in addition to P_{os} and P_g, while further reducing the memory footprint by N_d times. We present the analysis in this section. We begin by first presenting a brief overview of the communication volume for standard DP.

A. Data Parallel Communication Volume

During data parallel training, gradients across all data parallel processes are averaged at the end of the backward propagation before computing the updates for the next step. The averaging is performed using an all-reduce communication collective. For a large model size, the all-reduce communication is entirely communication bandwidth bound, and therefore, we limit our analysis to the total communication volume send to and from each data parallel process.

State-of-art implementation of all-reduce uses a two-step approach, where the first step is a reduce-scatter operation, which reduces different part of the data on different process. The next step is an all-gather operation where each process gathers the reduced data on all the process. The result of these two steps is an all-reduce.

Both reduce-scatter and all-gather are implemented in a pipelined fashion, that results in a total data movement of $\Psi \times \frac{N_d-1}{N_d}$ elements (for N_d processes and Ψ data elements) for each step. Therefore, the standard DP incurs $2 \times \Psi \times \frac{N_d-1}{N_d}$ data movement in each training step. For large value of N_d, this quantity is approximately equal to $2 \times \Psi$.

B. ZeRO-*DP Communication Volume*

1) Communication Volume with P_{os+g}: With gradient partitioning, each process only stores the portion of the gradients, that is required to update its corresponding parameter partition. As such, instead of an all-reduce, *ZeRO* only requires a scatter-reduce operation on the gradients, incurring communication volume of Ψ. After each process updates the partition of the parameters that it is responsible for, an all-gather is performed to collect all the updated parameters from all the data parallel process. This also incurs a communication volume of Ψ. So the total communication volume per training step is $\Psi + \Psi = 2\Psi$, exactly the same as the baseline DP.

2) Communication Volume with P_{os+g+p} : After parameter partitioning, each data parallel process only stores the parameters that it updates. Therefore, during the forward propagation it needs to receives the parameters for all the other partitions. However, this can be pipelined to avoid the memory overhead. Before computing the forward propagation on the part of the model corresponding to a particular partition, the data parallel process responsible for that partition can broadcast the weights to all the data parallel processes. Once the forward propagation for that partition is done, the parameters can be discarded. The total communication volume is thus $\frac{\Psi \times N_d}{N_d} = \Psi$. In other words, we reschedule the parameter all-gather by spreading it across the entire forward propagation, and discarding the parameters once they have been used. Note however that this all-gather needs to happen once again for the backward propagation in the reverse order.

The total communication volume is therefore the sum of the communication volumes incurred by these all-gathers in addition to the communication volume incurred by the reduce-scatter of the gradients. The total volume is therefore 3Ψ which is 1.5x compared to the baseline. Both gradient and parameter partitioning leverage the insight that — not all states of gradients and parameters are needed all the time — to optimize memory by communicating the states judiciously.

C. Comparison with MP

MP incurs significantly higher communication compared to *ZeRO*-DP. The communication volume of MP per mini-batch is proportional to the number of samples in the batch while communication for *ZeRO*-DP is a constant w.r.t mini-batch. We show that beyond a very small batch size, MP incurs significantly higher communication volume compared to *ZeRO*-DP.

MP splits the computation of each data sample across GPUs at an operator level by partitioning both the computation and the parameters corresponding to each forward and backward operator in the model. For example, MP may partition a linear layer across multiple GPUs each producing partial outputs on each GPU. As a result, the outputs need to be combined together for each sample, incurring a communication overhead.

The exact communication volume depends on model size, model architecture, activation checkpointing and MP strategy. To share concrete insights, we perform the communication volume analysis in the context of transformer based models implemented using SOTA MP approach, Megatron-LM.

In Megatron-LM with activation checkpointing, each transformer block performs two all-reduce operations of size $batch \times seq\_length \times hidden\_dim$ in the forward propagation, two all-reduce for forward re-computation and two more in the backward propagation. The total communication per batch is $12 \times batch \times seq\_length \times hidden\_dim \times num\_layers$ since communication volume of an all-reduce is $2 \times message\_size$.

In comparison, the max communication volume for *ZeRO*-DP is $3 \times model\_parameters$, as described in Sec. VII-B2, which for a transformer based model is approximately $3 \times 12 \times hidden\_dim \times hidden\_dim \times num\_layers$, where the factor of 12 is due to 4x for QKV and self-output linear layers and 8x for intermediate linear layers within each transformer layer. Therefore, the ratio between communication volume of MP w.r.t. *ZeRO*-DP is $\frac{batch \times seq\_length}{3 \times hidden\_dim}$. For language models, seq_length is generally greater than 1024, the largest $hidden\_dim$ used in literature at the time of submission is less than $5K$ [4]. Therefore, even for models with massive

$hidden\_dim$, MP incurs higher communication volume beyond a small batch size of a few dozen. LM models are generally trained with batch size over 512, and at those batch size MP would incur over an order-of-magnitude in communication volume overhead compared to *ZeRO*-DP.

VIII. Communication Analysis of *ZeRO*-R

We compare the communication volume of partitioned activation checkpointing (P_a) in *ZeRO*-R with baseline MP, and show that P_a incurs a communication volume increase that is in general less than one tenth of the baseline MP. Furthermore, we analyze the communication overhead of P_a in relation to DP communication volume to identify scenarios when P_a improves efficiency by allowing for a larger batch size and reducing DP communication. We leverage such analysis to decide if and when to apply P_a as well as P_{a+cpu}.

Communication volume trade-off of partitioning activation checkpoints depends on the model size, checkpointing strategy and the MP strategy. Once again, to provide concrete insights, we perform the analysis in the context of transformer based models implemented using SOTA MP approach, Megatron-LM.

In Megatron-LM with activation checkpointing, each transformer block performs two all-reduce operations of size $batch \times seq\_length \times hidden\_dim$ in the forward propagation, two all-reduce for forward re-computation and two more in the backward propagation. The total communication per block is $12 \times batch \times seq\_length \times hidden\_dim$ since communication volume of an all-reduce is $2 \times message\_size$.

When *ZeRO*-R partitions activation checkpoints, it requires an additional all-gather operation before the forward recomputation of the back-propagation on each activation checkpoint. In general, we checkpoint the input activation for each transformer block, requiring one all-gather per transformer block. The communication overhead P_a is therefore $batch \times seq\_length \times hidden\_dim$, since the communication volume of an all-gather is $message\_size$. Therefore, the total communication overhead of P_a is less than 10% of the original communication volume for model parallelism.

When MP is used in conjunction with DP, P_a can be used to increase the batch size and reduce the DP communication volume by an order of magnitude at the expense of a 10% increase in MP communication volume, This can significantly boost efficiency when DP communication is a performance bottleneck. More concretely, P_a reduces the activation memory consumption by the MP degree allowing for a proportional increase in batch size. For large models, MP can be as large as 16 (#GPUs on a DGX-2 node), allowing for up to 16x increase in the batch size. The communication volume during the end-to-end DP training for a fixed number of training samples is inversely proportional to the batch size. Therefore, an order of magnitude increase in batch size due to P_a could result in an order-of-magnitude decrease in DP communication volume.

Finally if P_{a+cpu} is applied, partitioned activation checkpoints are offloaded to CPU, reducing the activation memory requirement to nearly zero at the expense of 2x added data movement to and from CPU memory compared to P_a. In extreme cases where DP communication volume is the major bottleneck due to a small batch size even with P_a, P_{a+cpu} can improve efficiency by increasing the batch size as long as the CPU data transfer overhead is less than the DP communication volume overhead, which is generally true for small batch sizes.

Given model and hardware characteristics, we leverage the above analysis to decide if and when to apply P_a and P_{a+cpu}.

MP	GPUs	Max Theoretical Model Size				Measured Model Size	
		Baseline	P_{os}	P_{os+g}	P_{os+g+p}	Baseline	*ZeRO*-DP (P_{os})
1	64	2B	**7.6B**	14.4B	128B	1.3B	**6.2B**
2	128	4B	**15.2B**	28.8B	256B	2.5B	**12.5B**
4	256	8B	**30.4B**	57.6B	0.5T	5B	**25B**
8	512	16B	**60.8B**	115.2B	1T	*10B*	**50B**
16	1024	32B	**121.6B**	230.4B	*2T*	20B	**100B**

TABLE II: Maximum model size through memory analysis (left) and the measured model size when running with *ZeRO-OS* (right). The measured model size with P_{os} matches the theoretical maximum, demonstrating that our memory analysis provides realistic upper bounds on model sizes.

IX. Step Towards 1 Trillion Parameters

The largest published models today are in the range of 10 billion parameters, which are already challenging to train. Getting to a trillion parameters, 3-orders of magnitude larger, will inevitably happen, but the road will be full of hurdles, surprises and innovations. While we do not claim knowing or addressing all of them, *ZeRO* addresses one of the most fundamental challenges from a system perspective: the ability to fit a model of this scale on current hardware while allowing it to train with good system scalability.

A Leap from State-of-Art The largest model that the state-of-art framework, Megatron, can train with acceptable throughput is a 16 - 20B parameter model in a DGX-2 system. Scaling further by having model parallelism across multiple DGX nodes results in significant efficiency drop due to limited internode bandwidth.

ZeRO vastly increase the efficiently-runnable model size. It enables the current generation of hardware to run significantly larger models without requiring fine-grained model parallelism to go across the node boundaries. As demonstrated in Table I, *ZeRO*, with all optimizations turned on (P_{os+g+p}), could fit more than 1 *Trillion* parameters on 1024 GPUs using DP only. Alternatively, when combined with model parallelism (as shown in Table II), *ZeRO* could fit more than 1 *Trillion* parameters on 1024 GPUs with 16-way model parallelism (within each DGX2 node) and 64-way data parallelism across nodes. Running a model with a trillion parameters efficiently is no longer impossible!

Compute Power Gap Training a trillion parameter model end-to-end within an acceptable time range, however, could still require significant amount of compute power, which is lacking in today's AI clusters.

To understand the resource requirement, we present a brief comparison with Bert-Large. Bert-Large can be trained in 67 minutes on a 1024 GPU DGX-2H cluster [30]. A 1 Trillion Parameter model can easily contain 3000x (1 trillion / 330

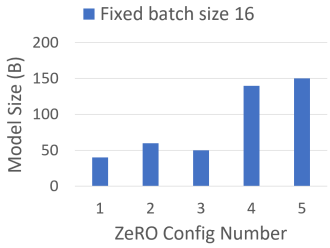

Fig. 5: Max model size

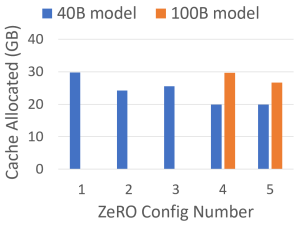

Fig. 6: Max cache allocated.

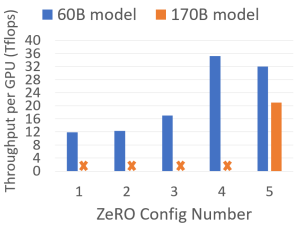

Fig. 7: Throughput per GPU.

	ZeRO-DP	*ZeRO*-R
1	P_{os}	C_B+M_D
2	P_{os}	C_B+M_D+P_a
3	P_{os+g}	C_B+M_D
4	P_{os+g}	C_B+M_D+P_a
5	P_{os+g}	C_B+M_D+P_{a+cpu}

TABLE III: *ZeRO* configurations

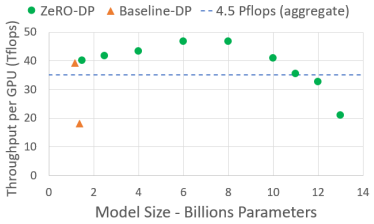

Fig. 8: Max model throughput with *ZeRO*-DP.

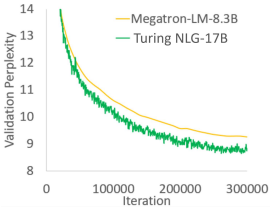

Fig. 9: Turing-NLG 17B enabled by *ZeRO*.

Figure 2			Figures 3, 8		
	Layers	HD		Layers	HD
1.5B	48	1600	1.16B–2.5B	24,34,54	1920
8B	72	3072	4B	64	2304
20B	98	4096	6B-8B	52,72	3072
40B–60B	88, 132	6144	10B–13B	50,54,58,62	4096
80B–170B	100,125,150, 175,212	8192	60B	75	8192

TABLE IV: Configurations for different model sizes, number of layers, and hidden dimensions (HD) across Figures 2, 3, 8.

million) more computation than a Bert-Large model for a data sample. Even if we assume the same sequence length and the total number of samples required to train the model, training a 1T model would take 140 days, assuming the same hardware and similar computational efficiency. In practice, both data samples and sequence length are likely to increase with the increased model size requiring over a year to train. It would require an exa-flop system to train a 1T parameter model in a reasonable time. But when such compute capacity becomes available, we hope *ZeRO* will provide the system technology to run the 1T models efficiently.

X. Implementation and Evaluation

We focus our implementation on supporting efficient training of models with ∼100B parameters, which are an order-of-magnitude larger than the largest published models today (e.g., T5-11B [5]) while trainable within a reasonable time frame on current hardware (e.g., with 1K V100 GPUs). We implement and evaluate a subset of optimizations in *ZeRO*—P_{os+g} in *ZeRO*-DP plus ZeRO-R — that allows us to achieve this goal. We will refer to this implementation as *ZeRO*-100B. Our results show that *ZeRO*-100B can efficiently train models with up to 170B parameters, 8x bigger than SOTA, up to 10x faster and with improved usability. *ZeRO*-100B powers *Turing-NLG*, the largest published model of the time with new SOTA accuracy.

A. Implementation and Methodology

a) Implementation: We implemented *ZeRO*-100B in PyTorch including the full set of optimizations in P_{os+g} and *ZeRO*-R. Its interface is compatible with any model implemented as `torch.nn.module`. Users can simply wrap their models using this interface and leverage *ZeRO*-powered DP as they use the classic DP. Users do not need to modify their model. *ZeRO*-powered DP can be combined with any form of MP including Megatron-LM.

b) Hardware: We conducted all of our experiments on a cluster of 25 NVIDIA DGX-2 nodes. Each node consisted of 16 NVIDIA Tesla V100-SXM3-32GB GPUs, 8x 100 Gbps Mellanox InfiniBand (ConnectX-5) cards, and dual Intel Xeon Platinum 8168 2.7 GHz CPUs (24-cores). All 25 nodes are connected together with InfiniBand using a 648-port Mellanox MLNX-OS CS7500 switch.

c) Baseline: For experiments without MP, we use torch's distributed data parallel (DDP) as baseline. For experiments with MP, we use Megatron-LM because it is, to our knowledge, the state-of-art. The most recent Megatron-LM results report the ability to scale up to 16B parameter models using 32 DGX-2 nodes (total of 512 32GB V100 GPUs) [4].

d) ZeRO*:* Experiments without MP use *ZeRO*-100B only, while experiments with MP combine *ZeRO*-100B with MP of Megatron-LM.

e) Model Configurations: The models presented in this section are GPT-2 [3] like transformer based models. We vary the hidden dimension and the number of layers to obtain models with different number of parameters. Table IV shows the configuration parameters used in our experiments with additional details in AE Appendix.

B. Speed and Model Size

ZeRO-100B efficiently run models with up to 170B parameters on 400 GPUs, more than 8x bigger than Megatron-LM. Figure 2 shows throughput per GPU for varying model sizes using *ZeRO*-100B with MP versus using Megatron MP alone. *ZeRO*-100B achieves a sustained throughput of 15 PetaFlops (over 30% of the peak) on average for models with 8B to 100B parameters. In comparison, the baseline MP performance degrades quickly with the increase in model size: MP incurs high communication volume between GPUs, and going beyond a single node to fit larger models causes a communication bandwidth drop from 300GB/sec per link (NVSwitch) to 12.5 GB/sec per link (Infiniband EDR), resulting in a significant performance drop as demonstrated by

the performance difference between 20B and 40B baseline models. *ZeRO*-100B achieves up to 10x speedup over baseline, significantly outperforming on large models.

For *ZeRO*-100B, the slight reduction in performance beyond 100B is due to lack of enough memory to run larger batch sizes. We expect the performance to improve as we increase the number of GPUs due to super-linear speedup of *ZeRO*-100B as we discuss next.

C. Super-Linear Scalability

ZeRO-100B demonstrates super-linear scalability for very large model sizes. Figure 3 shows scalability results for a 60B parameter model going from 64 to 400 GPUs and we expect this trend to continue further for more GPUs. P_{os+g} reduces per GPU memory consumption of *ZeRO*-100B with increase in DP degree, allowing *ZeRO*-100B to fit larger batch sizes per GPU, which in turn improves throughput as a result of increasing arithmetic intensity.

Please note that the throughput scalability shown here do not represent strong or weak scaling in the traditional sense, however, it is a proxy for the end-to-end training time, which is more meaningful in the context of DL training. In DL training, the amount of computation per iteration changes with different batch sizes, while the total end-to-end computation for a fixed total training samples is the same. Therefore, despite the change in amount of computation per iteration from an increased batch size, the end-to-end training time is inversely proportional to the absolute throughput number shown in Figure 3.

It is also worth pointing out that a significant batch size increase without increasing the total training samples can lead to poor convergence. However, this is not the case for our results. The largest batch size used in this experiment is 1.6K, which is well within range of batch sizes used to train LM models (512-2K) [3], [31]. In fact, a growing body of literature show that convergence rate for LM is resilient to batch sizes that are much larger than the ones used in this experiments [32], [25].

D. Democratizing Large Model Training

Using MP and PP is challenging for many data scientists, which is a well-known hurdle to train large models. *ZeRO* does not require any changes to the model itself and it can be used as simple as baseline DP while delivering significantly boosted model size and speed. Fig. 8 shows that *ZeRO*-100B can train models with up to 13B parameters without MP on 128 GPUs, achieving throughput over 40 TFlops per GPU on average. In comparison, without *ZeRO*, the largest trainable model with DP alone has 1.4B parameters with throughput less than 20 TFlops per GPU. Furthermore, in the absence of the communication overhead from MP, these models can be trained with lower-end compute nodes without very fast intra-node interconnect such as NVLINK or NVSwitch, which is required to achieve good efficiency with MP.

E. Memory and Performance Analysis

We look into the benefits and impact of different optimizations on maximum model size, memory consumption and performance. These optimizations are referred to as Config 1 to 5 (C1-C5) in Table. III.

a) Maximum Model Size: Figure 5 shows the largest trainable model by enabling different *ZeRO* optimizations for a fixed batch size and MP of 16. The model size increase from 40B to 60B when trained with C1 vs C2 due to a 16x (MP degree) reduction in activation memory from using P_a, while the jump to 140B using C4 is from enabling P_{os+g} which halves the memory requirement by the model states compared to P_{os} in C2. The increase to 150B using C5 is solely due to further reduction in activation memory from offloading the partitioned activation checkpoints to the CPU memory.

b) Max Cached Memory: Figure 6 shows the maximum memory cached by PyTorch during each training iteration for a 40B and a 100B parameter model. The decrease of the cached memory size is as expected from C1 to C2. The difference in memory consumption between C2 and C3 depends on the size of the model states in comparison to the activation memory, and can increase when activation memory is larger, or decrease when the model states are larger. It is note worthy that the cached memory does not decrease from C4 to C5 for 40B but it does for 100B. This is simply because the activation memory for 100B is much larger for the decrease to be noticeable. This makes P_{a+cpu} a valuable tool to fit a larger batch size when we get to very large models. In Figure 7, P_{a+cpu} is needed for 170B model to execute without running out of memory.

c) Max Achievable Performance: Figure 7 shows the best achievable performance for different set of optimizations. Notice that performance improvement corresponds to decrease in memory consumption between the optimizations. As mentioned earlier, lower memory consumption allows for larger batch size which improves performance. The only caveat is the performance drop between C4 and C5 for 60B parameter model. Despite lower memory consumption, C5 incurs activation movement to and from the CPU, this will result in worse performance in most cases, except for a few where the model is so large that the model simply cannot run without C5 or the batch size that can run without C5 is very small (such as model with 170B parameters in Figure 7). During training, P_{a+cpu} is turned on only when it is beneficial.

F. Turing-NLG, SOTA language model with 17B parameters

As of April 22, 2020, *Turing-NLG* [10] was the largest model in the world with over 17B parameters. It achieved the new SOTA for language models with Webtext-103 perplexity of 10.21. *Turing-NLG* was trained end-to-end using *ZeRO*-100B and Fig. 9 shows the validation perplexity over 300K iterations compared to previous SOTA, Megatron-LM 8.3B parameter model. *ZeRO*-100B achieves a sustained throughput of 41.4 TFlops/GPU for this model.

XI. Concluding Remarks

From a HPC and system perspective, we believe that *ZeRO* represents a revolutionary transformation in the large model training landscape. While our implementation, *ZeRO*-100B, enables 8x increase in model sizes, over 10x in throughput improvement, achieves super-linear speedups on modern GPU clusters, and trains the largest model in the world, it is still just a tip of the iceberg. *ZeRO* in its entirety has the potential to increase the model size by yet another order of magnitude, enabling the training of trillion parameter models of the future.

Perhaps, what we feel most optimistic about *ZeRO* is that it imposes no hurdles on the data scientists. Unlike existing approaches such as MP and PP, no model refactoring is necessary, and it is as easy to use as standard DP, making *ZeRO* a prime candidate for future investigations on large model training. Through open sourcing and community feedback, we plan to make *ZeRO* fully accessible to the DL community to catalyze the evolution and democratization of large model training at scale.

Acknowledgement

We thank Junhua Wang for his valuable support and advice. We thank Minjia Zhang, Elton Zheng, Shaden Smith, Reza Yazdani Aminabadi, Arash Ashari, and Niranjan Uma Naresh for their great feedback and help on evaluating the work. We thank Brandon Norick, Corby Rossett, Gopi Kumar, Jack Zhang, Jing Zhao, Payal Bajaj, Rangan Majumder, Saksham Singhal, Saurabh Tiwary, and Xia Song for many helpful discussions and suggestions.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *CoRR*, abs/1812.06162, 2018.

[10] Microsoft. Turing-nlg: A 17-billion-parameter language model by microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/, 2020.

[11] 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.

[12] 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.

[13] 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.

[14] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Granger, Phil Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *ACM Symposium on Operating Systems Principles (SOSP 2019)*, October 2019.

[15] 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.

[16] 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.

[17] 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.

[18] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. *ArXiv*, abs/2002.05645, 2020.

[19] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. 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, 2016.

[20] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. TFLMS: large model support in tensorflow by graph rewriting. *CoRR*, abs/1807.02037, 2018.

[21] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. *CoRR*, abs/1804.04235, 2018.

[22] Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer. Memory-efficient adaptive optimization for large-scale learning. *ArXiv*, abs/1901.11150, 2019.

[23] 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.

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

[25] 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.

[26] 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.

[27] NVIDIA Tesla V100 GPU architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. [Online, accessed 22-April-2020].

[28] NVIDIA. Automatic mixed-precision. https://developer.nvidia.com/automatic-mixed-precision, 2019.

[29] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[30] Shar Narasimhan. NVIDIA Clocks World's Fastest BERT Training Time ... https://devblogs.nvidia.com/training-bert-with-gpus/, 2019. [Online; accessed 25-September-2019].

[31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov.

Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[32] 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.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Our implementation of ZeRO was written in Python on top of PyTorch 1.2. We used torch.distributed and NVIDIA NCCL 2.4.8 for inter-GPU communication. We used NVIDIA Megatron-LM as our baseline, and integrated it with our DeepSpeed Library and ZeRO implementation for all of our experiments. All of our code is open source and available with detailed tutorials on how to use it, see ARTIFACT AVAILABILITY for details.

To offer full reproducibility of our results, we provide the model configurations (number of layers, hidden size, and attention heads), batch sizes, and number of GPUs we used for every experiment in Tables 1-6 below.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: No author-created artifacts are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/microsoft/DeepSpeed
Artifact name: DeepSpeed
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: All experiments were performed on NVIDIA DGX-2 hardware. Each node contains: 16x NVIDIA Tesla V100-SXM3-32GB; Dual Intel Xeon Platinum 8168, 2.7 GHz (24-cores); 24x 64GB DDR4 Micron 72ASS8G72LZ-2G6D2, 2666MT/s 288-UDIMM; 8x 100Gb/s InfiniBand Mellanox MT27800 Family (ConnectX-5); 2x NVME SSD Samsung 960GB (MZ1LW960HMJP); Mellanox MLNX-OS CS7500 648-ports

Operating systems and versions: a. Ubuntu 18.04.2 LTS; Linux ff422fc538c049ed804224fc761b952b-master-0 4.15.0-96-generic #97-Ubuntu SMP Wed Apr 1 03:25:46 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux

Compilers and versions: g++ (Ubuntu 7.4.0-1ubuntu1 18.04) 7.4.0; nvcc release 10.0, V10.0.130

Applications and versions: NVIDIA Megatron-LM from September 2019

Libraries and versions: Python 3.6.7; DeepSpeed 0.2; PyTorch 1.2; NVIDIA NCCL 2.4.8; NVIDIA Apex from February 2020

Key algorithms: ZeRO, ADAM

Input datasets and versions: OpenWebText

Figure 2								
Model size	ZeRO/Baseline	Number of GPUs	MP	Layers	Hidden size	Attention head	Batch size	Total batch size
1.5B	ZeRO	400	1	48	1600	16	24	9600
1.5B	Baseline	400	2	48	1600	16	16	3200
8B	ZeRO	400	4	72	3072	24	64	6400
8B	Baseline	400	8	72	3072	24	8	400
20B	ZeRO	400	4	98	4096	32	32	3200
20B	Baseline	400	16	98	4096	32	4	100
40B	ZeRO	400	4	88	6144	32	12	1200
40B	Baseline	384	32	88	6144	64	4	48
60B	ZeRO	400	16	132	6144	32	64	1600
60B	Baseline	384	64	132	6144	64	4	24
80B	ZeRO	400	16	100	8192	64	32	800
80B	Baseline	384	128	100	8192	128	4	12
100B	ZeRO	400	16	125	8192	64	32	800
100B	Baseline	384	128	125	8192	128	2	6
120B	ZeRO	400	16	150	8192	64	24	600
120B	Baseline	384	128	150	8192	128	2	6
140B	ZeRO	400	16	175	8192	64	16	400
140B	Baseline	384	128	175	8192	128	2	6
170B	ZeRO	400	16	212	8192	64	12	300
170B	Baseline	256	256	212	8192	256	2	2

Table 1: Model parameters and batch sizes to reproduce results in Figure 2 related to ZeRO throughput compared with baseline.

Figure 3								
Model size	ZeRO/Baseline	Number of GPUs	MP	Layers	Hidden size	Attention head	Batch size	Total batch size
60B	ZeRO	64,128,256,400	16	75	8192	32	16,48,48,64	64,384,768,1600

Table 2: Model parameters and batch sizes to reproduce results in Figure 3 related to superlinear scalability.

Figure 4								
Model size	ZeRO/Baseline	Number of GPUs	MP	Layers	Hidden size	Attention head	Batch size	Total batch size
40B	ZeRO	400	16	50	8192	32	16	400
60B	ZeRO	400	16	132	6144	64	16	400
140B	ZeRO	400	16	175	8192	64	16	400
150B	ZeRO	400	16	187	8192	64	16	400
50B	ZeRO	400	16	62	8192	32	16	400

Table 3: Model parameters and batch sizes to reproduce results in Figure 4 related to max model size with different ZeRO configurations.

Figure 5								
Model size	ZeRO/Baseline	Number of GPUs	MP	Layers	Hidden size	Attention head	Batch size	Total batch size
40B	ZeRO	400	16	50	8192	32	16	400
100B	ZeRO	400	16	125	8192	64	32	800

Table 4: Model parameters and batch sizes to reproduce results in Figure 5 related to memory allocated with different ZeRO configurations.

Figure 6								
Model size	ZeRO/Baseline	Number of GPUs	MP	Layers	Hidden size	Attention head	Batch size	Total batch size
60B	ZeRO	128	16	75	8192	64	2,4,32,32,8	16,32,256,256,64
170B	ZeRO	400	16	212	8192	64	12	300

Table 5: Model parameters and batch sizes to reproduce results in Figure 6 related to throughput with different ZeRO configurations.

Figure 7								
Model size	ZeRO/Baseline	Number of GPUs	MP	Layers	Hidden size	Attention head	Batch size	Total batch size
1.5B	ZeRO	128	1	34	1920	16	24	3072
2.5B	ZeRO	128	1	54	1920	16	24	3072
4B	ZeRO	128	1	64	2304	24	16	2048
6B	ZeRO	128	1	52	3072	24	12	1536
8B	ZeRO	128	1	72	3072	24	8	1024
10B	ZeRO	128	1	50	4096	32	6	768
11B	ZeRO	128	1	54	4096	32	4	512
12B	ZeRO	128	1	58	4096	32	4	512
13B	ZeRO	128	1	62	4096	32	2	256
1p16B	Baseline	128	1	24	1920	16	8	1024
1p38B	Baseline	128	1	40	1536	16	1	128

Table 6: Model parameters and batch sizes to reproduce results in Figure 7 related to evaluating maximum model sizes vs throughput while using only data-parallelism.