ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

XXX *
Microsoft

Abstract

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.

1 Extended Introduction

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 [1].

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 [2]), 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.

Optimizing Model State Memory: 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, 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.

ZeRO-DP removes the memory state redundancies across data-parallel processes by partitioning the model states instead of replicating them, and it retains

*The original paper is located at `https://arxiv.org/abs/1910.02054`. This version has been modified by LuYF-Lemon-love <`luyanfeng_nlp@qq.com`> for personal study.

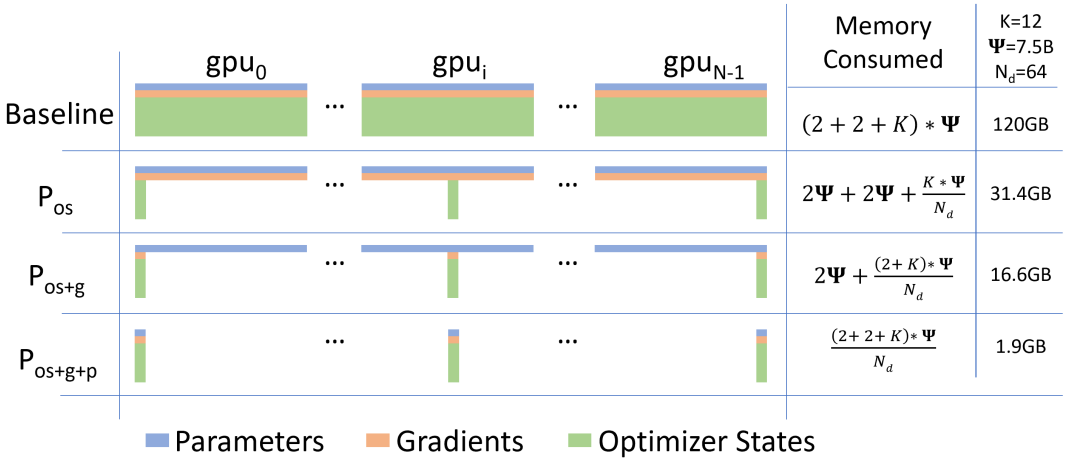

Figure 1: Comparing the per-device memory consumption of model states, with three stages of **ZeRO**-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and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

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. 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.

Optimizing Residual State Memory: We develop **ZeRO**-R to optimize the residual memory consumed by these three factors respectively.

1) For activations (stored from forward pass in order to perform backward pass), we noticed checkpointing [3] helps but not sufficient 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 optimiza-

tions for DL training that we collectively refer to as **ZeRO**.

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 share **ZeRO** as a part of our open source DL training optimization library called DeepSpeed[2].

2 Related Work

2.1 Data, Model and Pipeline Parallelism

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) [5, 1] and pipeline parallelism (PP) [6, 7] split the model among processes, in vertical and horizontal way respectively.

PP splits a model horizontally across layers running each partition on a different device and use micro-batching to hide the pipeline bubble [6, 7]. 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 [6] 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 [8] keeps multiple copies of stale parameters 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.

2.2 Non-parallelism based approach to reduce memory

2.2.1 Reducing Activation Memory

Multiple efforts have focused on reducing the memory footprint of activations through compression [9], activation checkpointing [3, 10], or live analysis [11]. These efforts are complimentary and can work together with **ZeRO**. In fact, activation memory reduction in **ZeRO**-R works in parallel with activation checkpointing.

[1] Prior work [4] 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.

[2] `https://github.com/microsoft/deepspeed`

2.2.2 CPU Offload

[12, 13] exploit heterogeneous nature of today's compute nodes, offloading model states to CPU memory through algorithmic design or virtualized memory, respectively. Up to 50% of training time can be spent on GPU-CPU-GPU transfers [12]. **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. 6.1 for details).

2.2.3 Memory Efficient Optimizer

[14, 15] 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.

2.3 Training Optimizers

Adaptive optimization methods [16, 2, 17, 18] 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.

3 Where Did All the Memory Go?

During model training, most of the memory is consumed by **model states**, i.e., tensors comprising of pptimizer 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**.

3.1 Model States: Optimizer States, Gradients and Parameters

Majority of the device memory is consumed by model states during training.

Consider for instance, Adam [2], 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 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.

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.

3.2 Residual Memory Consumption

Activations can take up a significant amount of memory [3] 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 [3]. This would reduce the activation memory consumption of this model to about 8 GB.

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: 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.

4 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.

4.1 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.

[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 $12 \times hidden\_dim \times batch \times seq\_length \times transformer\_layers$.

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 5) and uses a dynamic communication schedule that exploits the intrinsically temporal nature of the model states while minimizing the communication volume (Section 7). 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.

4.2 Insights and Overview: ZeRO-R

4.2.1 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 allgather 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.

4.2.2 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.

4.2.3 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. 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.

5 Deep Dive into ZeRO-DP

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.

5.1 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.

Memory Savings: As shown in Figure 1, the memory consumption after optimizing state partition reduces from $4\Psi + K\Psi$ to $4\Psi + \frac{K\Psi}{N_d}$.

5.2 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. 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 $2\Psi + \frac{14\Psi}{N_d} \approx 2\Psi$.

5.3 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.

Memory Savings: With parameter partitioning, we reduce the memory consumption of an Ψ parameter model from 16Ψ to $\frac{16\Psi}{N_d}$. 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.

5.4 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 1 analyzes model-state memory consumption of a few example models under the 3 stages of **ZeRO**-DP optimizations for varying DP degree.

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 1: 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.

6 Deep Dive into ZeRO-R

6.1 P_a: Partitioned Activation Checkpointing

As discussed in 4.2, 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 [3], 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 7. 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.

6.2 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. To address this issue, we simply use a performance-efficient constant-size fused buffer when the model becomes too large. 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.

6.3 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.

7 Communication Analysis of ZeRO-DP

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.

7.1 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 using a pipelined approach, that results in a total data movement of Ψ elements (for a data with Ψ elements) for each. Therefore, the standard DP incurs 2Ψ data movement during each training step.

7.2 ZeRO-DP Communication Volume

7.2.1 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.

7.2.2 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.

8 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.

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.

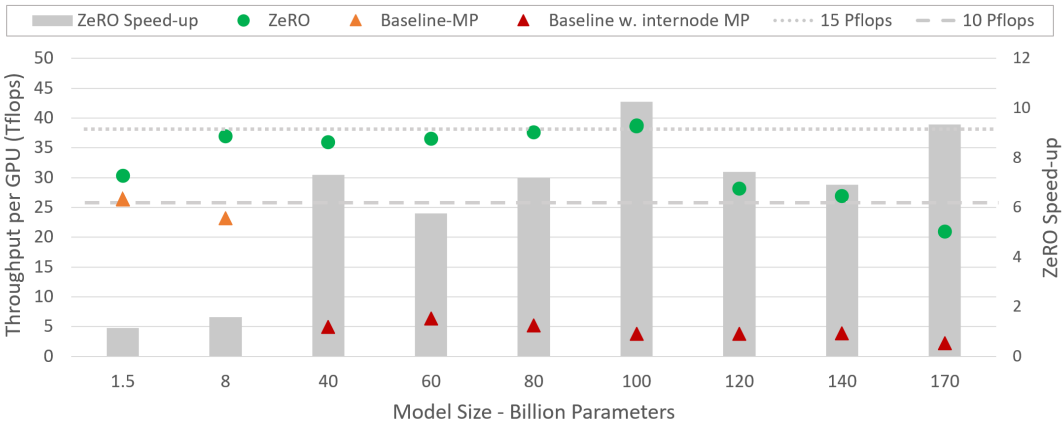

Figure 2: **ZeRO** training throughput and speedup w.r.t SOTA baseline for varying model sizes. For **ZeRO**, the MP always fit in a node, while for baseline, models larger than 40B require MP across nodes.

9 Implementation and Evaluation

9.1 Implementation and Methodology

Its interface is compatible with any model implemented as an `torch.nn.module`. Users can simply wrap their models using this interface and leverage **ZeRO**-powered DP as they use classic DP. Users do not need to modify their model. **ZeRO**-powered DP can be combined with any form of MP including Megatron-LM [4].

9.2 Speed and Model Size

Figure 2 shows throughput per GPU for varying model sizes using **ZeRO**-100B with MP versus using Megatron MP alone.

For **ZeRO**-100B, the slight reduction in performance beyond 100B is due to lack of enough memory to run larger batch sizes.

9.3 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.

9.4 Democratizing Large Model Training

Figure 4 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.

[4] `https://github.com/nvidia/Megatron-LM`

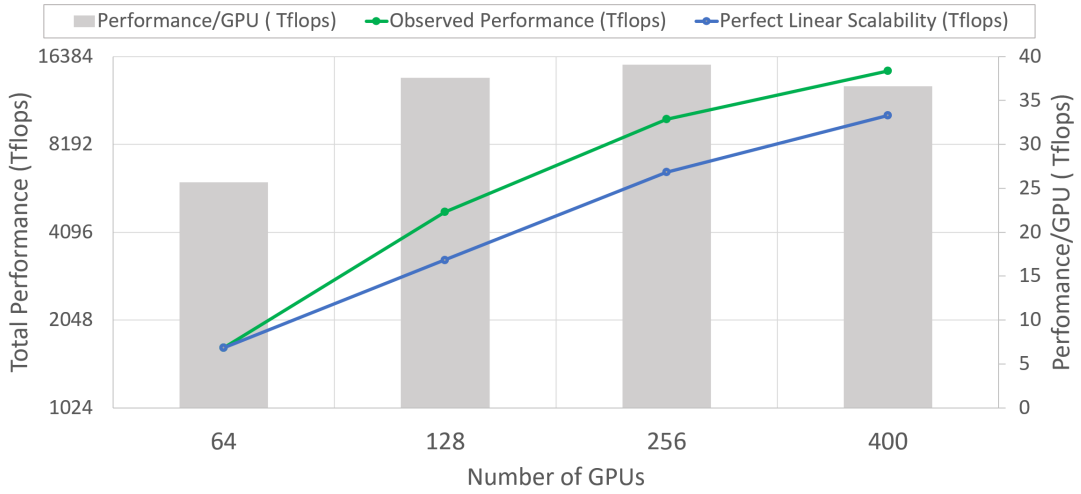

Figure 3: Superlinear scalability and per GPU training throughput of a 60B parameter model using **ZeRO**-100B.

9.5 Turing-NLG, the SOTA language model with 17B parameters

Turing-NLG was trained end-to-end using **ZeRO**-100B and Figure 5 shows the validation perplexity over 300K iterations compared to previous SOTA, Megatron-LM 8.3B parameter model.

References

[1] 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.

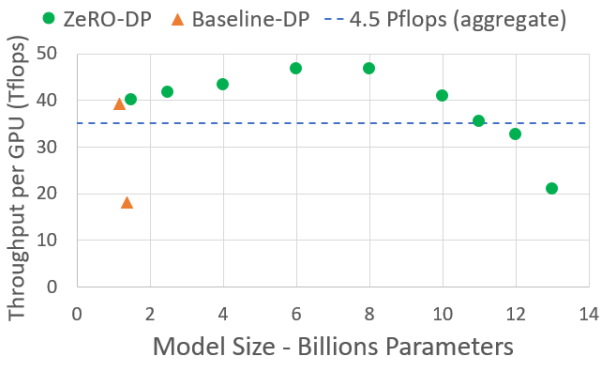

Figure 4: Max model throughput with **ZeRO**-DP.

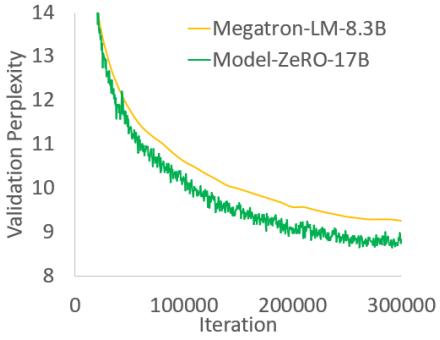

Figure 5: SOTA Turing-NLG enabled by **ZeRO**.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

This is because the total number of GPUs must be a product of the number of MP, and we only had access to a total of 400 GPUs. There exist a handful of additional constraints in model configuration values, such as hidden size must be divisible by attention heads, hidden size divisible by MP, and attention heads divisible by MP.