SC22: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-6654-5444-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/SC41404.2022.00076

# STRONGHOLD: Fast and Affordable Billion-Scale Deep Learning Model Training

Xiaoyang Sun[1,2], Wei Wang[2], Shenghao Qiu[1], Renyu Yang[1], Songfang Huang[2§], Jie Xu[1], Zheng Wang[1§]
[1]University of Leeds, UK [2]Alibaba Group, China
{scxs, sc19sq, r.yang1, j.xu, z.wang5}@leeds.ac.uk; {robot.sxy, hebian.ww, songfang.hsf}@alibaba-inc.com

***Abstract*—Deep neural networks (DNNs) with billion-scale parameters have demonstrated impressive performance in solving many tasks. Unfortunately, training a billion-scale DNN is out of the reach of many data scientists because it requires high-performance GPU servers that are too expensive to purchase and maintain. We present STRONGHOLD, a novel approach for enabling large DNN model training with no change to the user code. STRONGHOLD scales up the largest trainable model size by dynamically offloading data to the CPU RAM and enabling the use of secondary storage. It automatically determines the minimum amount of data to be kept in the GPU memory to minimize GPU memory usage. Compared to state-of-the-art offloading-based solutions, STRONGHOLD improves the trainable model size by 1.9x∼6.5x on a 32GB V100 GPU, with 1.2x∼3.7x improvement on the training throughput. It has been deployed into production to successfully support large-scale DNN training.**

***Index Terms*—Deep learning, Distributed training, DNNs training acceleration**

## I. INTRODUCTION

We are seeing exponential growth of deep neural network (DNN) sizes. For example, in a short span of 3 years, model size has grown by over 1000 folds from around 100 million model weights in 2018 for ELMo [1] or BERT [2] to 175 billion for GPT-3 [3] in 2020 and 530B for MT-NLG [4] in 2021. At the same time, the GPU memory capacity is increased by less than 3 folds per GPU generation. For instance, the latest high-end NVIDIA A100 GPU has 80 GB of memory, which is a 2.5x improvement on the GPU memory capacity over the 32 GB V100 predecessor. As a larger model tends to provide a better learning ability over the smaller counterparts, many future DNNs will continue to integrate many more parameters and consume more GPU memory. This will further increase the gap between the memory demand of DNN training and the available GPU memory, making entry into large model training out of reach for many data scientists and academics.

Large DNN training can be achieved by utilizing aggregated computing resources of multiple GPUs through parallel and distributed computation. Examples of such strategies include data [5], model [6] and pipeline [7], [8] parallelisms and by placing the model parameters, gradients and optimizer states across computing devices [9]. However, all these distributed training schemes require sufficient computing resources, which can incur significant infrastructure and operational costs [10]. For example, to efficiently train a 10B parameter model currently requires a computing system with 16 NVIDIA V100 GPUs, costing over $100K to purchase [11]. While such cost may not concern big tech firms, it can place a big financial burden on small businesses and academic organizations.

Efforts have been made to reduce the GPU memory pressure and computing resource requirement for large DNN training. This is done by either trading precision for lower storage space [12] or leveraging CPU memory [13], [14], [15], [16]. The former uses low or mixed precision representations for model states (e.g., model parameters, gradients and optimizer states) to reduce GPU memory consumption, but it can slow down the model convergence speed. The latter approaches reduce the GPU memory requirement by either implementing a software-managed cache or using a dedicated memory allocation scheme. However, most of the techniques for using CPU memory are designed for convolutional neural networks (CNNs), where the memory consumption during training is dominated by the dynamically generated activations rather than the optimizer (e.g., SGD) states. Unfortunately, these techniques are ill-suited for the latest attention-based language models (e.g., Transformer-based DNNs) that have become the de facto approach for building state-of-the-art DNNs [17], [2], [3], where the model and optimizer (e.g., Adam) states rather than activation memory is the memory bottleneck.

ZeRO-Offload [11] and L2L [18] were among the first attempts to leverage CPU memory to train large Transformer-based DNNs. ZeRO-Offload moves optimizer states from the GPU to the CPU memory while keeping the entire model parameters in the GPU memory. It is able to train a model with 13B parameters on a 32GB V100 GPU. As ZeRO-Offload requires storing the entire model parameters in the GPU, the trainable model size is limited by the smallest part of the available GPU and CPU memory capacity. L2L is specifically designed for Transformer-based models, only keeping one Transformer block in the GPU memory by dynamically offloading the model parameters. ZeRO-Infinity [19] adopts a dynamic strategy to leverage the secondary storage (e.g., NVMe) and partitions the model parameters and optimizer states across heterogeneous memory hierarchy. While these approaches can increase the trainable model size, as we will show later in the paper, they come with significant overhead, leading up to 1.76x slowdown (up to 29.2x when NVMe is used). Such magnitude of overhead greatly reduces the practicability of these techniques in training and fine-tuning large DNNs where the training time is a major concern [9].

§Corresponding authors. This work was conducted primarily during the first author's internship at Alibaba Group.

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

We present STRONGHOLD, a new approach for leveraging heterogeneous memory resources to scale up the trainable model size on GPUs without significantly compromising the training efficiency. The key insight of STRONGHOLD is to keep just a sufficient number of layers and their model parameters in the GPU to avoid GPU stalls during data offloading. Doing so can increase the trainable model size because only a small part of the model states are presented at the GPU memory at any time. By keeping the GPU computation going with overlapped data transfer, this strategy can also minimize and even hide the CPU-GPU communication overhead. To this end, STRONGHOLD implements a software pipeline between the GPU and CPU to dynamically offload the required model states between two memory spaces, *without user code refactoring*. Unlike ZeRO-Offload, STRONGHOLD does not store all model parameters in the GPU. Instead, it implements a dynamic working window to only store a few model layers in the GPU. It then dynamically moves the required layers and their parameters and the generated gradients between the CPU and the GPU. The STRONGHOLD runtime automatically determines a suitable working window size to ensure that asynchronous CPU-GPU data transfer can be overlapped with GPU computation to hide the data transfer latency.

A key challenge of STRONGHOLD is to decide how many layers are kept in the GPU. Having an unnecessarily large working window would waste the precious GPU memory with little performance gain while using an insufficient working window makes it difficult to overlap the CPU-GPU data transfer with GPU computation to minimize the overhead. We address this challenge by modeling the GPU computation with data transfer and using an analytical model to derive the right working window size based on profiling information collected from a few initial iterations during the warm-up phase.

STRONGHOLD runs multiple concurrent optimizers on the multi-core CPU to parallelize the model parameter update process. Since parameter update, data transfer, and GPU computation are asynchronous processes, they run in parallel to utilize the hardware parallelism. To reduce the GPU memory management overhead, STRONGHOLD employs a user-level GPU memory management scheme for the working window to avoid frequent invocations of the expensive GPU memory operations. We show that by reducing the GPU memory footprint, STRONGHOLD also opens up new opportunities to implement data parallelism within a single GPU. This is achieved by running multiple training workers across concurrently running streaming multiprocessors (SM) of a GPU while keeping one copy of model parallelism. This optimization leads to comparable or even better training performance over expert-tuned implementation when training a billion-scale model.

We implemented STRONGHOLD in PyTorch [20] and evaluated it in single and distributed GPU environments. We compare STRONGHOLD to three state-of-the-art offloading solutions [18], [11], [19]. Experimental results show that STRONGHOLD can support larger DNNs with a higher training throughput than competing schemes. Specifically, STRONGHOLD supports the training of a model with 39.5B and 82.1B parameters on a single 32GB V100 GPU and eight distributed 24GB A10 GPUs respectively without significantly compromising the training efficiency.

We envision STRONGHOLD to be attractive in two scenarios. It is useful in fine-tuning a large pre-trained DNN or using a pre-trained DNN to guide the training of a small model (a.k.a. knowledge distillation [21]) using limited GPU resources. This feature makes large DNN fine-tuning more accessible and affordable to small organizations and data scientists. It is also useful for accelerating model training by reducing cross-node communications or utilizing fine-grained GPU parallelization. For this use case, STRONGHOLD has been deployed to the production environment of Alibaba to support the training of DNNs with hundreds of billions and trillions of parameters.

This paper makes the following contributions:

- A new CPU-GPU offloading framework to scale up the trainable model size (Section III);
- An analytic model to determine the right working window for dynamic DNN training offloading (Section III-C);
- A new fine-grained GPU parallelism to speed up DNN training (Section IV-A).

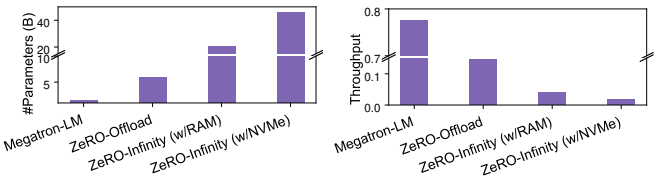
## II. BACKGROUND AND MOTIVATION

### A. Deep Learning Model Training

DNN training typically consists of millions of iterations performed across multiple training epochs. Each iteration mainly involves three stages: *forward propagation* (`FP`), *backward propagation* (`BP`) and *parameter update*. In the FP stage, a *batch* of the training samples are passed through the DNN model to compute a loss based on an objective function. In the BP stage, the loss value is propagated reversely through model layers to compute the *gradients*. In the last stage, an *optimizer* uses the aggregated gradients to update parameter weights of individual model layers.

The memory consumption during DNN training largely stems from model states and residual states. Model states include model parameters, gradients and optimizer states (i.e., Adam optimizer [22] stores momentums and variances for parameter updating). Residual states include activations (i.e., the intermediate tensors saved for BP stage to produce gradients) and other temporary buffers. When training large DNNs, model states dominate the memory consumption, which can account for 87.5% of the GPU memory footprint when low-precision (i.e., 16-bit precision) is used [9].

Large DNN models are trained with parallelization techniques. For a model that can fit into the device memory for training, data parallelism is commonly used to distribute the training samples across multiple devices to improve the training throughput. When the model cannot fit into the device memory, model parallelism [23], [24] and pipeline parallelism [25], [7] can be leveraged to split the model layers or parameters to make the best use of the memory across multiple devices. While these three parallelism strategies can




(a) Trainable model size (b) Throughput

Figure 1: Performance of Megatron-LM and ZeRO-based solutions measured on a 32GB V100 GPU by the trainable model size (a) and throughput on a 1.7B model (b).

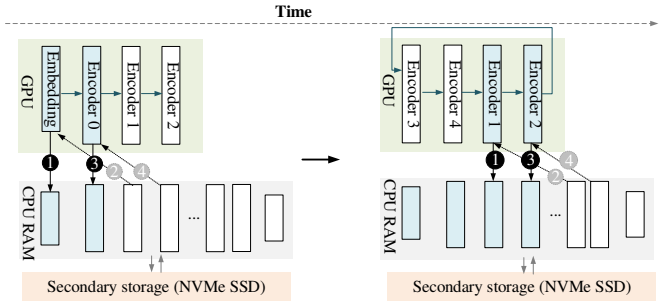



Figure 2: Dynamic model state offloading of STRONGHOLD. STRONGHOLD stores some DNN layers in the GPU memory and swapping out the finished layer states to the CPU RAM. Actions ❶ and ❸ indicate offloading layers that have been used during FP and BP from GPU memory to CPU RAM. These actions also trigger ❷ and ❹ to prefetch the future-used layers from CPU RAM to GPU memory.

produce a synergy, model and pipeline parallelism often require additional code refactoring to split a DNN into model and pipeline components.

### B. CUDA Streams

Modern GPUs consist of a large number of processing units, which are organized as streaming multiprocessors (SMs). For example, the NVIDIA V100 GPU supports 80 SMs, where each SM has a fixed number of cores. In the CUDA programming model, instructions placed within a single CUDA stream are executed sequentially. However, operations placed in different CUDA streams can be executed *concurrently* in different hardware SMs. STRONGHOLD uses multiple CUDA streams to accelerate DNN training when possible.

### C. Motivation

As a motivation example, consider Figure 1 that shows the performance given by ZeRO-Offload [11] and ZeRO-Infinity [19] - the state-of-the-art offloading solutions for large DNNs. This experiment was conducted on a GPU server with a 32GB NVIDIA V100 GPU and 755GB of DDR4 RAM (see Section V-A). As a reference, we use Megatron-LM, NVIDIA's heavily-optimized library for Transformer-based model training.

Figure 1a shows the largest trainable model size for each approach, while Figure 1b shows the throughput (i.e., the number of samples processed per second) on a common 1.7B Transformer-based model (the largest model supported by Megatron-LM on our platform). Although techniques like ZeRO-Offload and ZeRO-Infinity can scale up the trainable model size, this comes at the cost of significantly lower throughput and poor efficiency. For example, ZeRO-Offload enables training of a model that is 3x larger than Megatron-LM, but the training throughput on the 1.7B model is 6.7x less than Megatron-LM. By offloading some of the model parameters and states to the secondary NVMe SSD, ZeRO-Infinity (w/ NVMe) scales up the trainable model size by 29x over Megatron-LM, but its throughput drops by over 800x compared to Megatron-LM on a 1.7B model. This poor training efficiency of existing offloading solutions makes them impractical to train large models due to the long training time. STRONGHOLD aims to avoid this pitfall.

## III. OUR APPROACH

STRONGHOLD is our open-source framework designed to enable efficient training of large DNNs on single or distributed GPUs. This is achieved by implementing dynamic software prefetching and offloading techniques to only store part of the model states - the main GPU memory consumer for large DNN training - in the GPU memory. STRONGHOLD utilizes the CPU memory and secondary storage to reduce the GPU memory pressure for training large DNNs. By doing so, the trainable model size is no longer bounded by the GPU device memory but the system's storage capacity. STRONGHOLD is designed to be used without any user code refactoring similar to standard data-parallel training in PyTorch.

STRONGHOLD advances ZeRO-Offload [11], the state-of-the-art static CPU-GPU DNN training offloading framework, by offloading optimizer states onto the CPU side. Like ZeRO-Infinity [19], STRONGHOLD can also leverage secondary storage, but it delivers a higher throughput than ZeRO-Infinity.

### A. Overview of STRONGHOLD

Figure 2 gives a high-level overview of the STRONGHOLD dynamic offloading scheme. The idea is to store model states (parameters, gradients and optimizer states) for selected DNN layers in the GPU memory. This is achieved by managing a *working window* in the GPU, where layer states are dynamically moved between the GPU and CPU memory. Precisely, the STRONGHOLD runtime swaps the already computed layer states (blue shaded boxes in Figure 2) from GPU memory to the CPU RAM (and potentially between the CPU RAM and secondary storage). It then adaptively prefetches the parameters of the subsequent layers in the FP or BP processing pipeline into the working window. STRONGHOLD leverages *asynchronous* data transfer to hide the CPU-GPU communication overhead by overlapping data transmission with GPU computation. By doing so, STRONGHOLD greatly reduces the CPU-GPU communication overhead, and in many cases, it can completely hide the data transfer overhead. Therefore, STRONGHOLD only causes modest slowdown in the training speed. Crucially, the *asynchronous* operations do not introduce stale model updates nor not affect the training precision.

STRONGHOLD is a low-level runtime library. It automatically identifies offloading sequence and determines the working window size. It then dynamically partitions the tensor

graph and manages BP, FP and parameter updates without user code refactoring.

### B. Preprocessing

During the model loading stage, STRONGHOLD extracts DNN layers and their execution order from the tensor graph. Most Transformer-based models follow a sequential layer execution order by stacking multiple Transformer blocks (Figure 3a), resulting in a static relationship. Extracting the layer execution order of such model architectures is straightforward. However, there are also other model structures with residual components [26] or gating mechanisms [27], [28] like mixture of experts (MoE) models [29], where the execution path can change dynamically at *inference time*. For these non-linear structures, STRONGHOLD either offloads all units/layers directly connected to a branch to the GPU working window (if possible), or delays the layer movement until it knows which layer will be computed to avoid GPU out-of-memory (OOM) errors. This can be further improved by leveraging techniques that pre-compute the activated layers [15] to proactively determine which layers to be moved to the GPU working window ahead of time. When loading a model, for each DNN layer, STRONGHOLD also computes the required storage size for the parameter tensors and the associated gradients and optimizer states. This storage size is then used to determine the GPU working window size during FP and BP[1].

During the first few iterations (5 by default) of model training (i.e., the warm-up phase), the STRONGHOLD runtime profiles the GPU computation time and the data transfer time of model states of each layer. It then uses this information to derive the working window size for later training iterations. At the warm-up phase, STRONGHOLD ensures that the chosen GPU working window size does not cause out-of-memory (`OOM`) errors by using the storage size information of the DNN layers to compute the right working window size. Despite the initial working window size may not lead to the optimal GPU memory use, the overhead is negligible since the profiling is only performed on the first few iterations. The dynamic working window size derived for later training iterations (see Section III-C) is designed to overlap the CPU-GPU data transfer with the GPU computation while minimizing the GPU memory consumption. We note that the computation performed in the warm-up phase also contribute to the final training outcome, so no computation cycle is wasted.

### C. Dynamic GPU Offloading

As a working example, we use a simplified Transformer-based model shown in Figure 3a to illustrate the dynamic GPU working window mechanism of STRONGHOLD. In this subsection, we assume STRONGHOLD does not use secondary storage (which will be discussed later in Section III-G).

The STRONGHOLD runtime maintains a GPU working window with the layer-specific inputs, model parameters, and gradients (for BP). Within STRONGHOLD, the basic offloading unit under data and pipeline parallelism is an entire DNN layer. However, under tensor parallelism, this can be a sliced layer on the GPU defined by the user code. The working window essentially contains GPU buffers for the tensor operator implementations (kernels) and the data that the kernel operates on. CPU-GPU data movement is automatically handled by the STRONGHOLD runtime, which registers callback functions for each layer through the hooking mechanism provided by mainstream deep learning frameworks. STRONGHOLD supports activation checkpointing as long as the working window size is larger than the number of layers between two consecutive checkpoints. Conceptually, this mechanism resembles applying a sliding window to the DNN model along the FP or BP direction, described as follows.

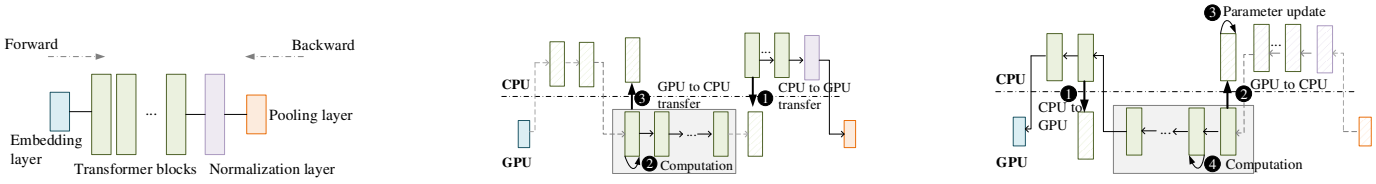
**FP stage.** As shown in Figure 3b, before executing each layer in the working window, the $pre\_forward$ hook function is called to issue an asynchronous load operation to fetch the next layer right outside the current working window from the CPU RAM to the GPU memory (step ❶). Next, the GPU performs FP computation as normal on the first layer in the current working window (step ❷). At the end of the layer computation, the $post\_forward$ hook function invokes another asynchronous operation to move model parameters of the *already computed* layer back to the CPU RAM (step ❸). The computation result will then be passed to the next layer in the working window. At the end of the GPU-CPU transfer, the GPU buffer used by a computed layer will be recycled by a newly fetched layer. After these steps, the working window moves toward the successive layer of the DNN, following the direction of FP. The asynchronous CPU-GPU data transfer takes place concurrently with the GPU computation and will not block the STRONGHOLD runtime.

**BP stage.** As depicted in Figure 3c, this step moves the working window along the BP direction. Before computing a layer in the working window, the $pre\_backward$ hook function invokes an asynchronous operation to fetch the parameters of the layer that is just outside the current working window in the BP direction (step ❶). The $pre\_backward$ function also issues an asynchronous operation to move the model parameters (and gradients) of the last computed layer in the working window to the CPU (step ❷), followed by a call to the optimizer to update the majority of the model parameters on the CPU (step ❸). Finally, the GPU computes gradients for the second-last layer of the working window (step ❹). Once again, the CPU-GPU data communications run concurrently with the GPU computation and the windows moves towards the BP direction.

### D. Modeling Offloading Parameters

STRONGHOLD uses an analytical model to automatically find a suitable GPU working window size during FP and BP. The key here is to find the right window size where the asynchronous CPU-GPU data transfer can overlap with GPU computation to hide the data transfer latency, without oversubscribing the GPU memory.

[1]The current implementation of STRONGHOLD stores most of the optimizer states in the CPU RAM.




(a) Transformer-based DNN (b) Working window during FP (c) Working window during BP

Figure 3: Dynamic offloading of STRONGHOLD during the FP and BP stages using a simplified Transformer-based architecture as an example. STRONGHOLD keeps the first and the last layers of the DNN (the embedding and pooling layers) in the GPU memory to reduce the initialization overhead.

**Notations.** We use the following notations to model CPU-GPU offloading. We use $s^i_{fp}$ and $s^i_{bp}$ to denote the memory consumption of layer $i$ during FP and BP respectively, and $t^i_{fp}$ and $t^i_{bp}$ [2] to denote the GPU computation time on layer $i$ during FP and BP respectively, $t^i_{c2g}$ and $t^i_{g2c}$ to denote the CPU to GPU and GPU to CPU data transfer respectively for layer $i$, and $t_{async}$ and $t_{opt_{gpu}}$ to denote the overhead on an asynchronous function call and one layer's parameter update respectively. We let $N = \sum_{k=0}^{m} S_k$ be the GPU working window size, where $S_k$ is the model state (parameters and gradients) of layer $k$ within the $m$-layer window.

**FP offloading.** We use the following formulation, $\mathbb{P}_1$, to ensure layer fetching does not become a bottleneck of FP.

$$\mathbb{P}_1: \min\ m \tag{1a}$$
$$s.t.\ \sum_{i=0}^{m} t^i_{fp} \geq t^j_{c2g}, \tag{1b}$$
$$\sum_{i=0}^{m} s^i_{fp} + s^j_{fp} \leq S_{avail}, \tag{1c}$$
$$\sum_{i=0}^{m} t^i_{fp} \geq \sum_{i=0}^{m} t^i_{c2g} + \sum_{i=0}^{m} t^i_{g2c}, \tag{1d}$$

where $m$ is the number of layers in the GPU working window, layer $j$ is the layer outside the current working window along the FP direction, $s^{m+1}_{fp}$ is the buffer size required for fetching the layer outside the current working window (step ❶ in Figure 3b), and $S_{avail}$ is the available GPU memory (by excluding the runtime memory consumption). Here, terms (1b) and (1c) are hard constraints to ensure that the data transfer time is less than the layer computation time, and no GPU OOM will happen. The FP computation time for one layer is $t^i_{fp} + 2\ t_{async}$. Term (1d) is a soft constraint to ensure that we can recycle the buffer of the computed layer (steps ❷ and ❸ in Figure 3b) to further reduce the GPU memory consumption.

**BP offloading.** Like FP, we use the following formula, $\mathbb{P}_2$, to ensure gradient and parameter offloading does not become the bottleneck during BP:

$$\mathbb{P}_2: \min\ m \tag{2a}$$
$$s.t.\ \sum_{i=0}^{m-1} t^i_{bp} \geq t^j_{g2c}, \tag{2b}$$
$$\sum_{i=0}^{m} s^i_{bp} \leq S_{avail}, \tag{2c}$$
$$\sum_{i=0}^{m-1} t^i_{bp} \geq \sum_{i=0}^{m-1} t^i_{g2c} + \sum_{i=0}^{m-1} t^i_{c2g}, \tag{2d}$$

where layer $j$ is the layer outside the current working window along the BP direction. The time for BP computation of one layer is $t^i_{fp} + 3\ t_{async}$. The soft constraint in (2d) is used to further reduce the GPU memory consumption when possible.

**Parameter update.** With conventional DNN training, the GPU performs the parameter update layers by layers. This gives a total parameter update time of $\sum_{i=0}^{n} t^i_{opt_{gpu}}$, where $n$ is the total number of layers of the DNN model. STRONGHOLD utilizes the CPU cores to perform most parameter update, which runs concurrently with the GPU gradient computation. As a result, the parameter update time in STRONGHOLD is $\sum_{i=0}^{m} t^i_{opt_{gpu}} + \sum_{i=m}^{n} t^i_{opt_{cpu}}$. To hide the CPU computation overhead, the CPU-directed parameter update time should satisfy:

$$t^k_{opt_{cpu}} \leq \sum_{i=0}^{k} (t^i_{fp} + t^i_{bp}) + \sum_{i=0}^{m} t_{opt^i_{gpu}}, k \in [m, n]. \tag{3}$$

To avoid introducing additional overhead during a training iteration, we need to ensure the computation time incurring by STRONGHOLD during FP and BP is not greater than the time for conventional training, i.e.,

$$\sum_{i=0}^{n} t^i_{fp} + n * 2\ t_{async} + \sum_{i=0}^{n} t^i_{bp} + n * 3\ t_{async} + \sum_{i=0}^{m} t^i_{opt_{gpu}} \leq \sum_{i=0}^{n} t^i_{fp} + \sum_{i=0}^{n} t^i_{bp} + \sum_{i=0}^{n} t^i_{opt_{gpu}}$$

which gives us:

$$5\ n\ t_{async} \leq \sum_{i=m}^{n} t^i_{opt_{gpu}} \tag{4}$$

When applying into a DNN where most of the layers are homogeneous with the same number of parameters, e.g.,

[2]Note that $t^i_{bp}$ also includes the FP re-computation time with activation checkpointing.

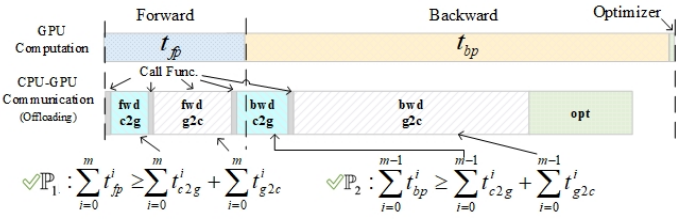



Figure 4: Real GPU computation and offloading profiling trace when applying STRONGHOLD to train a 4B model on a 32GB V100 GPU. The profiling measurement supports our analytical modeling of computation-communication overlapping optimization. Here, GPU computation and communication are overlapped when $\mathbb{P}_1$ and $\mathbb{P}_2$ are satisfied.

Transformer-based models, GPU computation time can be approximated as $\sum_{i=m}^{n} t^i_{opt_{gpu}} \approx (n-m)\ t_{opt_{gpu}}$ and hence:

$$5\ n\ t_{async} \leq (n-m)\ t_{opt_{gpu}} \tag{5}$$

Since the overhead of asynchronous function calls is largely constant regardless of the DNN model size, we can easily satisfy (5) with a deep (i.e., a large $n$) or a wide network (i.e., a large $t_{opt_{gpu}}$ due to more parameters that a layer has).

**Determining the working window size.** For an $n$-layer DNN, STRONGHOLD automatically finds a suitable working window size $m$ that meets all $\mathbb{P}_1$ and $\mathbb{P}_2$ across all layers during a training iteration. It is possible that there is not enough GPU memory to find an optimal $m$ to meet all the constraints. In this scenario, STRONGHOLD still uses the largest possible $m$ layers permitted by the avaiable GPU memory to train a large DNN (that would not be possible using conventional training methods) but the training efficiency may be sub-optimal. By default, STRONGHOLD finds an available GPU buffer for $m$ layers. This strategy improves the GPU cache locality for Transformer-based models that have a large number of identical layer structures (and computation kernels). However, STRONGHOLD also supports having a fixed-size GPU buffer where the number of DNN layers stored can dynamically change, which can be turned on by users to improve GPU memory utilization for DNN models with a heterogeneous layer structure.

As a working example, Figure 4 shows the profiling data of one training iteration when applying STRONGHOLD to train a 4B model on a 32GB V100 GPU. The profiling results show that the CPU-directed offloading is largely overlapped by the GPU computation when criteria $\mathbb{P}_1$ and $\mathbb{P}_2$ set in our analytical models are met. In this case, the communication overhead can be hidden by the GPU computation, suggesting the effectiveness of our analytical model.

### *E. Offloading Optimization*

STRONGHOLD utilizes the gRPC module of Ray [30] and concurrent library for communications among parallel CPU workers. As a result, STRONGHOLD can support concurrent and asynchronous parameter updates and data transfer. To this end, STRONGHOLD maintains a thread pool. All workers are initialized with the model code, and a worker remains idle until a task has been assigned to it through a callback function. By default, STRONGHOLD uses all available CPU cores, but the user can change this.

*1) Concurrent parameter update:* Unlike conventional training schemes (including ZeRO-Offload) that employ a single optimizer for parameter update, STRONGHOLD creates multiple optimizers during the model initialization stage. It then dispatches several optimizers to run as asynchronous actors to perform parameter updates on multiple layers simultaneously (step ❸ in Figure 3c). This optimization leverages multiple CPU cores to process the parameter updates of multiple layers simultaneously, reducing the chance for the CPU becoming a bottleneck. As parameter updates are performed by the CPU, this process runs concurrently with the GPU computation during BP. By default, STRONGHOLD keeps the first few layers of the model (i.e., layers of the first working window) in the GPU memory. Since the last $m$ layers in BP (i.e., the first $m$ layers of the model) remain in the GPU working window before the start of FP, there is no GPU stall when computing BP of the last layers in Figure 3c ❷.

*2) Heterogeneous collective communications:* For distributed training involved multiple computing devices, gradients communications are realized through collective communication operations like all-scatter and all-gather. With a native deep learning framework (e.g., PyTorch and Tensorflow), only one type of tensors (CPU or CUDA) can participate in collective communications at a time. STRONGHOLD lifts this restriction to support concurrent heterogeneous collective communications on CPU and CUDA tensors. This feature is essential for STRONGHOLD to support concurrent CPU and GPU processing. This is achieved by extending the low-level collective communication libraries, NVIDIA NCCL [31] and Gloo [32] for GPU and CPU communications, respectively.

*3) Runtime memory management:* During DNN training, many temporary tensors will be allocated and deallocated. Frequent device memory operations using the native CUDA memory (de)allocation API can result in expensive runtime due to explicit and implicit synchronizations. Frameworks like PyTorch and Tensorflow avoid this issue by reusing the previously allocated buffers through a software caching mechanism. For an $n$-layer DNN, where each layer has $k$ tensors, such a caching mechanism incurs up to $n \times k$ GPU memory allocation operations. After the first training iteration, these $n*k$ GPU buffers are then reserved by the runtime, which can then be reused for future training iterations. This strategy is ill-suited for our scenarios when the model is too large to be fit into the GPU memory (i.e., the $n*k$ buffers are beyond the GPU memory capacity). STRONGHOLD addresses this issue by employing a user-level software memory management scheme on the CPU and GPU. For an $m$-layer GPU working window, STRONGHOLD only needs to incur a one-off $m*k$ CUDA memory operation at the warm-up stage. Since the working window size, $m$, is smaller than the number of layers (i.e., $m < n$) of the DNN, STRONGHOLD reduces the GPU memory footprint while incuring fewer memory allocation operations than existing caching mechanisms.

Specifically, when loading the DNN, STRONGHOLD allocates pinned memory on the CPU for each DNN layer. The pinned (or page-locked) memory permits STRONGHOLD to asynchronously transfer the CPU data to the GPU using an idle CUDA stream so that the GPU will not be blocked during data transfer. At the same time, STRONGHOLD also reserves GPU buffers for layers of the first working window. The reserved buffers will be managed by STRONGHOLD in future training iterations. The reserved GPU buffer may grow (but not shrink) if larger buffers are needed once STRONGHOLD has determined the working window size after the warm-up stage. When prefetching a layer from the CPU memory to the GPU (e.g., step ❶ in Figure 3b), STRONGHOLD first allocates a free GPU buffer from the reserved GPU memory in a round-robin manner. It then copies the corresponding `data` content to the corresponding GPU tensor (e.g., through the PyTorch `tensor.copy_()` API). Similarly, when offloading a layer from the GPU to the CPU memory, STRONGHOLD copies the `data` property back to the corresponding CPU buffer. It then returns the GPU buffer to the STRONGHOLD managed GPU buffer queue. Whenever STRONGHOLD requests or releases device memory, the STRONGHOLD runtime always reuses the reserved GPU memory by overwriting the in-place memory management methods of the layer implementation.

### F. Cross-server Communication Optimization

Another benefit of STRONGHOLD is that it can eliminate the cross-server communications introduced by traditional model parallelism in certain cases. For example, if a model cannot fit into the GPU memory under a traditional training method, model parallelism is typically adopted to break the model layers (and their parameters) across multiple GPUs. In contrast, if the same model can fit into the same GPU under STRONGHOLD, we can then use the additional GPUs to run data parallelism training without incurring the synchronization and communication overhead of model parallelism. The reduction of cross-server communications when converting model parallelism to data parallelism for Transformer-based models can be estimated as follows. The communication volume for an $n$-layer Transformer model is $V_{dp} = (w-1)w \times (12 \times n \times hd^2 + hd \times vs)$ for $w$-way data parallelism, and $V_{mp} = (w-1)w \times n \times bs \times seq \times hd$ for $w$-way model parallelism. Here, $hd$, $bs$, $seq$, and $vs$ are the the hidden size, batch size, sequence length and vocabulary size, respectively. Furthermore, we obtain the constant number 12 summing $4 \times hd^2$ for attention and $2 \times 4 \times hd^2$ in the feed-forward network in one Transformer block. By converting $w$-way model parallelism to $w$-way data parallelism, STRONGHOLD reduces the communication volume by $\frac{V_{mp}}{V_{dp}}$.

Using a typical training setup where the training sentence sequence length is set to 1024 ($seq = 1024$) and vocabulary size is set to $30k$ ($vs = 30K$), we can simplify $\frac{V_{mp}}{V_{dp}}$ as $\frac{V_{mp}}{V_{dp}} = \frac{bs}{3 \times hd/256 + 30/n}$. Let $k = \frac{1}{3 \times hd/256 + 30/n}$, we now have $V_{mp}/V_{dp} = k \times bs$. Here, the saving in cross-node communication depends on $n, hd, w$, and $bs$. By increasing the trainable model size, STRONGHOLD allows one to use a $w$-way data parallelism to replace the traditional $w$-way model parallelism. For a 20B model with a typical $bs = 16$, $n = 50$, $hd = 4K$, STRONGHOLD halfs the communication traffics by comparing to model parallelism (see also Section VI-D2).

### G. Utilizing Secondary Storage

Like ZeRO-Infinity, STRONGHOLD provides an option to use NVMe SSDs to further increase the trainable model size. This is achieved by memory-mapping a swap file on the secondary storage to the CPU memory space and using the read/write library to optimize asynchronous bulk read/write requests between the CPU and the device. The support for asynchrony allows STRONGHOLD to overlap the I/O requests with CPU-GPU communication or computation. Since the I/O bandwidth between the CPU and an NVMe SSD (up to 7GB/s for PCIe 4.0) is an order of magnitude slower than the CPU-GPU bandwidth, we do not expect the user to train a large DNN from scratch with this option. It is also not advised to train a large model with this strategy because frequent random reads and writes can increase the chance of NVMe disk failure [33]. However, this option can be useful in fine-tuning a pre-trained model with fewer training iterations.

## IV. STRONGHOLD-ENABLED OPTIMIZATIONS

By reducing the GPU memory footprint, STRONGHOLD not only permits training larger DNNs but also opens up new optimization opportunities. For example, STRONGHOLD allows training a large DNN that previously was would otherwise only possible using model parallelism to split the DNN layers across GPU servers. With STRONGHOLD, the distributed GPU servers can be used to run data-parallel training workers, where each computing node holds the entire model parameters. This can massively reduce the communication and synchronization overhead imposed by model parallelism (see also Section VI-D2). Another use case is to support teacher-student based knowledge distillation [21] by supporting large DNN inference on a single GPU (Section VI-D3). The third interesting optimization is to support data parallel training within a single GPU by utilizing multiple CUDA streams (see also Section II-B) to use the GPU parallelism to improve the training throughput, which was not attempted in accelerating DNN training before. This is described in the next subsection.

### A. Multi-streamed GPU Execution

As depicted in Figure 5, STRONGHOLD enables the use of multiple CUDA streams to accelerate training if there is enough GPU memory to store the gradients and model inputs for at least two training workers.

Conceptually, this is achieved through applying data parallelism within a single GPU by partitioning the training batch into mini-batches. To this end, STRONGHOLD introduces 'executors' within its runtime to manage the GPU kernel execution context across training workers. An executor is a process of the STRONGHOLD runtime that manages the working window for a training worker. The executor dispatches kernels to run on a CUDA stream. Note that only one copy of the model

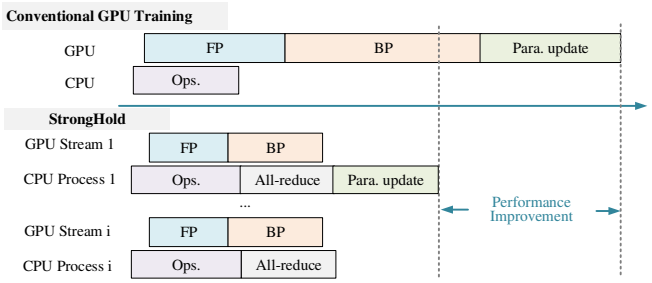



Figure 5: Utilizing GPU SMs to speedup training, where each SM processes a microbatch of the training data.

Table I: Transformer-based model configurations.

| Model Size (B) | #Layers | Hidden Size | #Heads | Model Parallelism |
|---|---|---|---|---|
| 1.7, 4.0, 5.9, 6.0, 6.6, 20.5, 23.7, 39.4 | 20, 50, 74, 75, 83, 260, 300, 500 | 2,560 | 16 | 1 |
| 4.0 | 19 | 4,096 | 16 | 1 |
| 6.2, 10.0 | 19, 31 | 5,120 | 16 | 1 |
| 3.4, 4.7, 7.8, 23.2, 63.2, 75.7, 82.0, 103.2, 367.6, 524.5 | 10, 12, 24, 72, 200, 240, 260, 328, 1174, 1676 | 5,120 | 16 | 8 |
| 19.8, 25.4 | 24, 31 | 8,192 | 16 | 8 |
| 28.7, 32.1, 66.7 | 31 | 8,704, 9,216, 13,312 | 16 | 8 |

parameters and kernel code is stored in the GPU memory, despite that there may be more than one training worker and working window on the GPU. Furthermore, STRONGHOLD's strategy of dividing a larger batch into micro-batches to be processed by concurrently running does not affect the model consistency since parameter update takes after the entire batch has been processed, i.e., data-parallel training.

During runtime, we bind each executor to an available CUDA stream and allocate dedicated buffers on the CPU to allow the executor to manage the host-device communications for the corresponding GPU working window. Each executor runs as a multi-staged pipeline to process a *mini-batch*, and each pipeline stage of an executor can execute a kernel for FP, BP, computation, communication or another user-defined kernel. As depicted in Figure 5, by mapping different concurrent executors onto multiple CUDA streams, we can allow different processing pipelines to run in parallel on a single GPU to improve the throughput. STRONGHOLD uses an all-reduce operation to synchronize the gradients among parallel training workers before performing parameter updates, similar to data parallel training across *multiple* GPUs (but STRONGHOLD has the advantage of only keeping one copy of model parameters). The number of concurrent streams used is determined during the warm-up phase, where STRONGHOLD computes the GPU memory consumption of the GPU working window size to determine how many CUDA streams to use so that multi-streamed execution does not cause GPU OOM.

STRONGHOLD essentially creates a lightweight parallel execution environment for individual executors within a single user program. This is different from existing GPU virtualization schemes such as NVIDIA MIG (Multi-Instance GPU) [34], which aim to provide an isolated execution environment for different user programs running on the same GPU. MIG is ill-suited for our purpose because CUDA streams or processes in different virtualized environments cannot directly communicate with each other. To implement our idea within MIG would also require each GPU process holds a copy of model parameters, resulting in multiple model parameters being stored on a single GPU, increasing the GPU memory pressure. Our lightweight approach can avoid such pitfalls by only storing one copy of the model parameters on the GPU and enabling CUDA streams to directly communicate with each other through the GPU hardware communication scheme. Since our goal is to accelerate a single user program, we do not need a heavy and strongly isolated execution environment.

## V. EVALUATION SETUP

### A. Evaluation Platforms

We evaluate STRONGHOLD on two hardware platforms. Our main evaluation platform is a V100 GPU server with one 32GB NVIDIA Tesla V100 GPU, 2x 24-core Intel Xeon Platinum 8163 CPU at 2.50GHz and 755GB of DDR4 RAM. We also evaluate STRONGHOLD on an 8-node A10 GPU cluster where servers are connected through a 800 Gbps network. Each computing node of the cluster has 2x 64-core Intel Xeon Platinum 8369B CPUs at 2.90 GHz with 1TB of DDR4 RAM, and a 24GB GPUDirect-RDMA-enabled A10 GPU based on the latest Ampere architecture. The platforms run Ubuntu 20.04 operating system with Linux kernel 4.19.91. We use CUDA 11.2 and PyTorch 1.10.2.

### B. Workloads

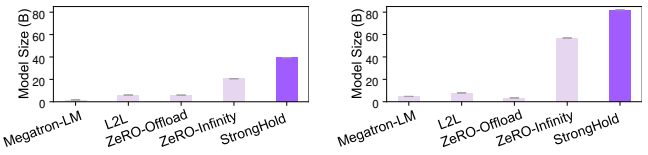
We evaluate STRONGHOLD on GPT-like Transformer-based models [35], [3] – the current de facto approach to building large-scale models [2], [3]. Following the evaluation setup of ZeRO-Offload [11], we vary the hidden dimension of a layer to increase the model width and the number of layers to scale the model depth. Table I lists the model parameters. For each model, we consider a batch size of 2, 4, 8, and 16 per GPU. Note that scaling the depth alone is often not sufficient because it would make training harder to converge [36]. Unless stated, we use training hyperparameters like weight decay and learning rate from [23], [11].

### C. Competing Baselines

We compare STRONGHOLD against the following billion-scale model training solutions:

**Megatron-LM** [37] is NVIDIA's optimizing library for Transformer-based models. We use Megatron-LM v2.6 as a reference model for the training throughput and trainable model size.

**L2L** [18] keeps one Transformer layer in the GPU at a time, by *sequentially* offloading parameters between the GPU and CPU memory. Since L2L still stores the optimizer states on the GPU, it is largely limited by the GPU memory.




(a) Single V100 GPU (b) A10 cluster (8 A10 GPUs)

Figure 6: The largest trainable model size on a 32GB V100 GPU (a) and the A10 cluster with 8 degrees of model parallelism (b).

**ZeRO-Offload** [11] statically stores the model states in GPU memory and optimizer states in the CPU RAM. It also utilizes the CPU computation cycle to update the model parameters through a CPU-tuned optimizer.

**ZeRO-Infinity** [19] utilizes GPU, CPU and NVMe memory. By default, we compare STRONGHOLD against ZeRO-Infinity with CPU RAM instead of using NVMe due to the expensive I/O overhead. In Section VI-C, we compare STRONGHOLD with ZeRO-Infinity when using a 2TB PCIe4.0 NVMe SSD.

**ZeRO-2 and ZeRO-3** [9] partition the model states across distributed machines but with a data parallel strategy. We compare STRONGHOLD to the ZeRO-2 and -3 solutions (which are the core of the DeepSpeed [38] framework) for distributed training in Section VI-D2.

### D. Performance Report

We consider *trainable model size* and *throughput* in the evaluation. We report the trainable model size by counting the number of model parameters (using FP32 representation) that can be trained without incurring GPU OOM, and report throughput by measuring the number of training samples processed per second. We run each test case 10 times on unloaded servers to measure the metrics, and then report the geometric mean across different runs. In our evaluation, we use layer-wised activation checkpointing [39] that is widely used in large-scale training. We note that the profiling overhead of STRONGHOLD at the warm-up phase accounts for less than 0.5% of the total training time on our evaluation setup. This overhead will be much smaller (and negligible) in a real-life training scenario with a higher number of training iterations. We have included this overhead when computing the throughout of STRONGHOLD. Finally, the throughput and efficiency variance across runs is small, less than 3%.

## VI. EXPERIMENTAL RESULTS

In this section, we first show that STRONGHOLD outperforms all the offloading baselines by enabling the training of a 39.5B model on a V100 GPU and an 82.1B model across 8 distributed A10 GPUs using model parallelism (Section VI-A) and giving a higher training throughput (Section VI-B). We then perform further analysis on STRONGHOLD (Section VI-C) - including the use of NVMe - before showcasing the new optimizations enabled by STRONGHOLD (Section VI-D).

### A. Trainable Model Size

Figure 6 compares the trainable model size when using CPU RAM only. The min-max bar on the diagram gives the range of measurements for different DNN configurations when varying the hidden dimension and model depth.

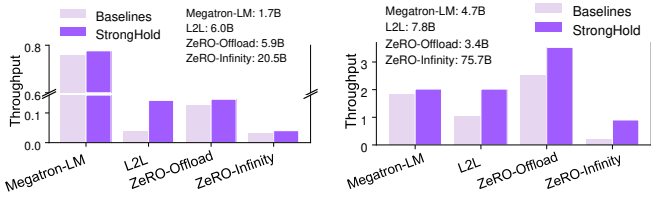
*1) Single V100 GPU:* Figure 6a gives the largest trainable model on a single 32GB V100 GPU. Megatron-LM allows one to train a model with up to 1.7B parameters on a V100 GPU before incurring GPU OOM. L2L and Zero-Offload can expand the trainable size by 3.5x over Megatron-LM, to support a model with around 6B of parameters through CPU-GPU offloading. Moreover, the fine-grained parameter partitions in ZeRO-Infinity (with CPU RAM only) can support a 20.6B model[3]. STRONGHOLD outperforms all baselines, supporting model training with 39.5B parameters and giving comparable training efficiency compared to other offloading scheme. STRONGHOLD is limited by the CPU and GPU memory and the offloading unit size. Therefore, it can support a larger model with larger CPU RAM and GPU memory (or reducing the size of the offloading unit through tensor parallelism). The trainable model size of STRONGHOLD translates to a 6.5x improvement over L2L and Zero-Offload, and an 1.9x improvement over Zero-Infinity. ZeRO-Infinity requires moving the parameters, gradients, and optimizer states to the GPU for runtime model refactoring. This operation requires making a copy of the refactored model parameters, incurring extra GPU memory overhead. STRONGHOLD does not have this overhead, leading to a larger trainable model size. Furthermore, while Zero-Infinity could assure large DNN training, as we will show in Section VI-B, it comes at the cost of poor training efficiency.

*2) Distributed GPUs:* Figure 6b shows the largest trainable model size across 8 distributed A10 GPU servers using model parallelism. All offloading approaches benefit from additional GPU resources. However, L2L and Zero-Offload give limited improvement on the trainable model size as they are largely constrained by a single GPU memory. By partitioning the model states across heterogeneous devices, ZeRO-Infinity and STRONGHOLD demonstrate stronger scalability by scaling the trainable model size to 56.9B and 82.1B parameters respectively, with STRONGHOLD supports the largest model. In all test cases, STRONGHOLD gives a nearly 100% GPU utilization, with 80% utilization of the theoretical peak bandwidth of the CPU-GPU PICe or communication network.

### B. Training Throughput

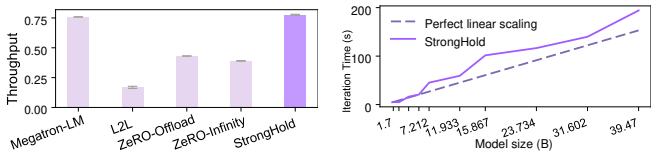
Figure 7 compares the training throughput on the largest trainable model size supported by each baseline. STRONGHOLD runs the same model of its counterpart. STRONGHOLD outperforms all baselines (including Megatron-LM thanks to STRONGHOLD's multi-streamed optimization), achieving 42 $\sim$ 57% hardware performance by delivering

[3]In [19], Zero-Infinity was reported to support a 1T model using either a DGX-2 node with NVMe or 32 DGX-2 nodes (512 V100 GPUs), and FP16 for model parameters, for which the computation resources and parameter settings differ from our evaluation setup.




(a) Single V100 GPU (b) A10 cluster (8 A10 GPUs)

Figure 7: Throughput (#samples/second) on a single 32GB V100 GPU (a) and the A10 cluster (b) when training the largest trainable model of each baseline.




(a) Throughput (b) Scaling w/ model size

Figure 8: The throughput given by different strategies on a common 1.7B model (a). STRONGHOLD delivers nearly linear scaling performance (lower-is-better) as we increase the model size on a V100 GPU (b).

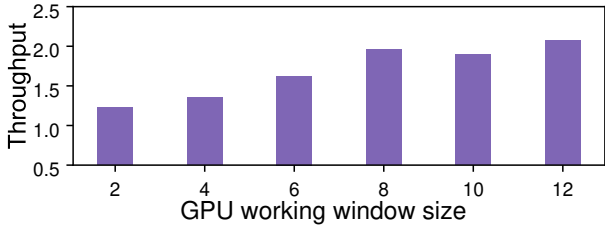



Figure 9: Impact of the *GPU working window size* on performance.

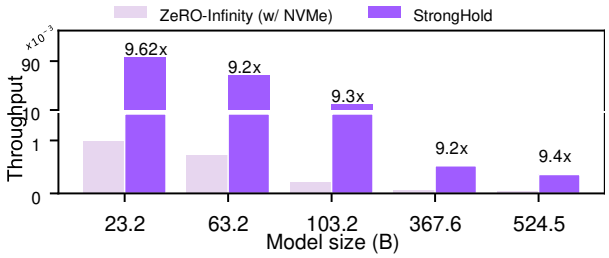



Figure 10: STRONGHOLD improves ZeRO-Infinity when using secondary storage (NVMe).

6∼9 TFlops on a V100 GPU. The TFLOPS given by STRONGHOLD far exceeds the one delivered by L2L (1.88), ZeRO-Offload (0.59) and ZeRO-Infinity (0.53). It improves the training throughput by at least 1.1x (up to 3.7x) by better overlapping the CPU-GPU communication. Figure 8a shows the throughput obtained when a common 1.7B model (the largest trainable model supported by Megatron-LM) on a V100 GPU. For this case, L2L delivers only 22.2% of the Megatron-LM throughput because it simply serializes computation with data transfer for each DNN layer. ZeRO-Offload and ZeRO-Infinity achieve less than 57% of the Megatron-LM training efficiency because a large portion of the CPU-GPU data transfer and computation cannot overlap due to their CPU optimizer implementation. STRONGHOLD is the only offloading solution that gives an improvement over Megatron-LM. The results show that STRONGHOLD can scale up the trainable model size and accelerate DNN training.

### C. Further Analysis

*1) Training efficiency:* Figure 8b shows that STRONGHOLD delivers nearly linear training efficiency on a single V100 GPU, using a 1.7B model as the starting point. The training efficiency is a *lower-is-better* metric, measured by the averaged time in performing one training iteration. STRONGHOLD's performance is on par with a perfect linear scaling projection, albeit there are some fluctuations in the scaling trend due to the impact of the GPU working window size on the GPU cache performance. Using the same resources, STRONGHOLD can train 25x, 4x, and 20x bigger models in a single-GPU-single-node, multiple-GPU-single-node and multiple-GPU-multiple-node environment, respectively; and it achieves these without significantly compromising the training efficiency. Therefore, STRONGHOLD can reduce the number of GPUs required by at least 4x compared to the traditional training method. The resource-saving can also be used to increase data parallelism by 4x to speed up the training time. Based on this scalability projection, we estimate that STRONGHOLD can complete the training of 175B GPT-3 using a quarter of the GPUs with a similar training time, or it can speed up the training process by 4x when using the same number of GPUs compared to the conventional distributed training method. This capability can reduce the cost by using either smaller-scale GPU resources or fewer GPU hours, making the training of large DNN more accessible and affordable.

*2) Impact of working window size:* Figure 9 shows how the GPU working window size affects the throughput by running a 1.7B and a 39.5B model on a V100 GPU. In our cases, initially, a larger window can better overlap GPU computation with data transfer, which leads to a higher training throughput. However, the improvement reaches a plateau with a window size of 8. Using a window size greater than 8 does not justify the gain in throughput but will increase the GPU memory pressure. Using the analytical method described in Section III-D, STRONGHOLD automatically determines to use a window size of 8 for this model.

*3) Using NVMe:* Figure 10 shows the throughput improvement over ZeRO-Infinity when NVMe is used to scale up the model size on the V100 GPU server. When using NVMe, STRONGHOLD and Zero-Infinity can support the training of a model with half trillions of parameters on our V100 GPU server. Compared to Zero-Infinity, STRONGHOLD can also better overlap the disk I/O requests with GPU computation, improving the throughput by over 8x.

### D. STRONGHOLD-*enabled Optimizations*

*1) Multi-streamed optimization:* Figure 11 shows performance improvement given by STRONGHOLD over Megatron-LM when varying the batch size. By only keeping part of the DNN layers and gradients in the GPU, STRONGHOLD

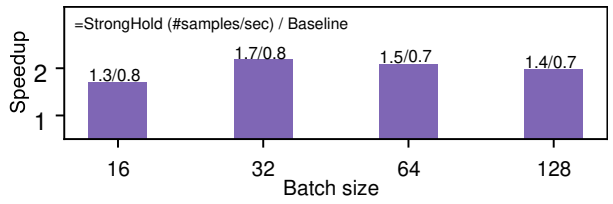



Figure 11: Speedup over Megatron-LM under different training batch sizes when the STRONGHOLD multi-stream optimization is enabled.

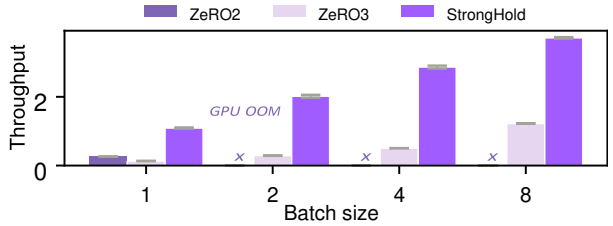



Figure 12: Performance on the 8-node A10 cluster (using the largest trainable model supported by ZeRO-2). STRONGHOLD outperforms ZeRO-based distributed training solutions.

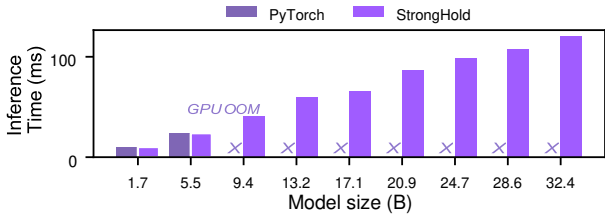



Figure 13: Using STRONGHOLD to support DNN inference for knowledge distillation on a single 32GB V100 GPU.

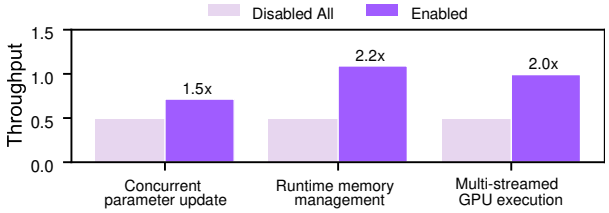



Figure 14: Improvement given by individual optimizations when applying STRONGHOLD to train a 4B model with NVMe enabled (see also Figure 10).

reduces the GPU memory footprint by 60%. The reduced memory footprint permits STRONGHOLD to use multiple CUDA streams speedup the training process, leading to at least 1.7x (up to 2.1x) speedup over Megatron-LM.

*2) Accelerating distributed training:* Figure 12 compares STRONGHOLD against ZeRO-2 and ZeRO-3 in a distributed training setup on the A10 GPU cluster. ZeRO-2 partitions the optimizer states and gradients across parallel processes running on multiple servers while ZeRO-3 partitions the model parameters on top of ZeRO-2 across GPU servers [9]. We apply all approaches to the largest model (3B parameters) that can be supported by ZeRO-2 with a batch size of 1. ZeRO-2 and ZeRO-3 have to partition the model/optimizer states across GPUs due to the GPU memory restrictions, but they introduce extra communication overhead across GPUs and serve nodes. By reducing the GPU consumption, STRONGHOLD does not need to partition the model across GPU servers. Instead, it can run the entire model on a single server to exploit data parallelism across GPU servers. As a result, STRONGHOLD reduces the cross-server communications, leading to over 2.6x throughput improvement over ZeRO. For example, STRONGHOLD reduces the cross-server communication by around 50% on a 20B model (see also Section III-F). This experiment demonstrates another advantage of STRONGHOLD in speeding up distributed deep learning training.

*3) Knowledge distillation:* STRONGHOLD can also support knowledge distillation [21]. This strategy is widely used to accelerate DNN inference by using a *trained* large model to guide the training of a smaller but faster DNN. Under this setting, the large DNN only needs to perform FP on training samples to provide layer-wised activations to guide the student model training. Inferencing frameworks like TensorRT [40] are not suitable for this scenario because they do not produce activations of intermediate layers. As can be seen from Figure 13, STRONGHOLD can effectively support large DNN inference for knowledge distillation. Note that as STRONGHOLD only needs to support FP in this scenario, it can support a larger model than when it is used for training that is involved with both FP and BP. It gives similar performance for small DNN inference compared to Pytorch but delivers linear scalability for large DNNs where PyTorch give an OOM error.

*4) Optimization breakdown:* Figure 14 reports the performance improvement on the V100 GPU platform when turning on a single optimization to a baseline offloading scheme without optimization. By utilizing the multi-core CPU, concurrent parameter update with heterogeneous collective communication (Sections III-E1 and III-E2) give 1.5x throughput improvment. By minimizing the GPU tensor allocation overhead, our memory management optimization (Section III-E3) alone gives a 2.2x throughput improvement. Similarly, by launching multiple kernels concurrently, our multi-streamed optimization (see Section IV-A) offers up to 2x improvement.

## VII. RELATED WORK

Data and model parallelisms are two dominant parallelization techniques for large DNN training [41], [38], [42]. The former partitions the training data across multiple GPUs [5], [43], and the latter splits the model layers vertically and then distributes different layers onto GPUs to reduce the memory pressure of the model states on a single GPU [6]. ZeRO [9] partitions the training batch across multiple GPUs, similar to data parallelism, but it further splits the model states across GPUs and uses collective operations to gather the required model parameters. Parallelization can also be achieved by horizontally partitioning a tensor operator across multiple GPUs [23], [44]. STRONGHOLD complements the existing data-parallel approaches by using multiple GPU SMs to achieve fine-grained data parallelism while just keeping one copy of the model parameters across parallel training workers.

A recent line of work adds pipelines into model parallelism by partitioning model layers into parallel stages [7], [8], [45], [46], [47], [48], [49]. In this way, each training batch is divided into micro-batches to be processed by pipeline stages across computing devices. Although STRONGHOLD also follows pipelining, tasks on a STRONGHOLD pipeline are finer-grained, where the model layers stored in a GPU memory can change dynamically during execution.

All the aforementioned strategies utilize the aggregated GPU memory of multiple GPUs to meet the memory requirement of large DNNs. STRONGHOLD is among the recent attempts in scaling up the trainable model size on a single GPU. This line of research includes activation check-pointing methods, which trade computation for GPU memory saving [39]. This strategy drops the activations after FP and recomputing them from checkpoints during BP. Unfortunately, this comes as the cost of huge overhead for large DNNs because a large number of activations to be recomputed. Compression techniques, such as using low or mixed precision representations for model states, can reduce the memory footprint [12]. However, compression techniques can reduce the training accuracy and slowdown model convergence.

Our work falls under a third approach to utilize external memory like the CPU RAM and NVMe to expand the memory capacity during training [50], [13], [51], [15], [52], [16]. ZeRO-Offload [11] and ZeRO-Infinity [19] are the most closely related work. ZeRO-Offload uses the CPU memory to store gradients and optimizer states. Unlike STRONGHOLD, ZeRO-Offload stores the entire model parameters in the GPU memory. As such, ZeRO-Offload is limited by the GPU memory, rather than the external memory like STRONGHOLD. ZeRO-Infinity extends ZeRO-Offload through fine-grained model state partitioning and utilizing the secondary storage. When using NVMe, ZeRO-Infinity can greatly increase the trainable model size, but this comes at the cost of prohibitively long training time. By carefully overlapping computation and communication, STRONGHOLD significantly improves the training efficiency over ZeRO-Infinity. L2L [18] is a Transformer-specific offloading scheme. It keeps an encoder layer in the GPU memory, by *synchronously* moving the model parameters the CPU memory to mimic layer-by-layer computation. Unlike STRONGHOLD, L2L requires model refactoring and offers poor efficiency due to extensive communication overhead and frequent GPU stalls. The M6 model [53] keeps a fixed number of layers on GPUs, which is specific to the current model, and requires code refactoring.

The work presented in [54] combines rematerialization to trade memory for computation time and offloading to trade memory for data movement. It employs a dynamic programming heuristic to determine the optimal offloading sequence. AxoNN [55] exploits asynchronous and message-driven execution for scheduling parallel training workers to improve GPU utilization and system throughput. Varuna [56] leverages low-priority virtual machines and pipelining to enable low-cost model training over commodity networking. These techniques are complementary to STRONGHOLD.

Dorylus exploits the workload characteristics of graph neural networks (GNNs) to use parallel serverless CPU threads for model training [57]. Dorylus and other memory [58] or computation [59] optimization techniques can be used in combination with STRONGHOLD to utilize low-cost CPU threads to train GNNs. Furthermore, STRONGHOLD can also be used together with asynchronous training [60] to further reduce the waiting time across training epochs, but care must be taken to avoid slowing down model convergence [61].

## VIII. CONCLUSION

We have presented STRONGHOLD, a new offloading framework to lower the GPU memory consumption for training billion-scale DNNs. STRONGHOLD utilizes heterogeneous resources to scale the trainable model size. Compared to existing offloading solutions, STRONGHOLD reduces the GPU memory footprint with lower computation overhead. It achieves this by maintaining a compact GPU working window and using data prefetching techniques to overlap data transfer and GPU computation. By reducing the GPU memory consumption, we demonstrate that STRONGHOLD enables new optimization to utilize the GPU hardware parallelism to improve performance. We show that STRONGHOLD allows the training of a larger DNN model with better training efficiency than the state-of-the-art offloading techniques. Specifically, it enables the training of a DNN with 39.5 billion parameters on a single V100 GPU without changing user code and supports faster model training in a distributed GPU environment.

## ACKNOWLEDGEMENT

This project was supported in part by the Alibaba Research Intern Program and an Alibaba Innovative Research Program between Alibaba and the University of Leeds. Zheng Wang was also supported in part by a Meta research award. For any correspondence, please contact Songfang Huang (E-mail: songfang.hsf@alibaba-inc.com) and Zheng Wang (E-mail: z.wang5@leeds.ac.uk).

## REFERENCES

[1] M. E. Peters, W. Ammar, C. Bhagavatula, and et al, "Semi-supervised sequence tagging with bidirectional language models," *arXiv*, 2017.

[2] J. Devlin, M.-W. Chang, K. Lee, and et al, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv*, 2018.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[4] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti *et al.*, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," *arXiv preprint arXiv:2201.11990*, 2022.

[5] S. Li, Y. Zhao *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[6] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao *et al.*, "Large scale distributed deep networks," *NeurIPS*, vol. 25, 2012.

[7] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[8] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *SOSP*, 2019, pp. 1–15.

[9] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[10] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?" in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 2021, pp. 610–623.

[11] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, I. Calciu and G. Kuenning, Eds. USENIX Association, 2021, pp. 551–564.

[12] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," in *International Conference on Learning Representations*, 2018.

[13] H. Jin, B. Liu, W. Jiang, Y. Ma, X. Shi, B. He, and S. Zhao, "Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, pp. 37:1–37:26, 2018.

[14] M. Hildebrand *et al.*, "Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *ASPLOS*, 2020.

[15] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1341–1355. [Online]. Available: https://doi.org/10.1145/3373376.3378530

[16] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 598–611.

[17] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 10 012–10 022.

[18] B. Pudipeddi *et al.*, "Training large neural networks with constant memory using a new execution algorithm," *arXiv*, 2020.

[19] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[20] PyTorch. https://pytorch.org.

[21] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *IJCV*, vol. 129, no. 6, pp. 1789–1819, 2021.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015.

[23] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv*, 2019.

[24] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *EuroSys*, 2019.

[25] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv*, 2018.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[27] W. Fedus, B. Zoph *et al.*, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *arXiv*, 2021.

[28] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang *et al.*, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv preprint arXiv:2006.16668*, 2020.

[29] C. Riquelme, J. Puigcerver, B. Mustafa, M. Neumann, R. Jenatton, A. Susano Pinto, D. Keysers, and N. Houlsby, "Scaling vision with sparse mixture of experts," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[30] Ray. https://github.com/ray-project/ray.

[31] NCCL. https://developer.nvidia.com/nccl.

[32] Gloo. https://github.com/facebookincubator/gloo.

[33] 4 causes of ssd failure and how to deal with them. https://www.techtarget.com/searchstorage/tip/4-causes-of-SSD-failure-and-how-to-deal-with-them.

[34] NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-sg/technologies/multi-instance-gpu/.

[35] A. Radford, J. Wu, R. Child, D. Luan *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[36] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child *et al.*, "Scaling laws for neural language models," *arXiv*, 2020.

[37] Megatron-LM. https://github.com/NVIDIA/Megatron-LM.

[38] DeepSpeed. https://www.deepspeed.ai.

[39] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv*, 2016.

[40] NVIDIA TensorRT. https://docs.nvidia.com/deeplearning/tensorrt/.

[41] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *CSUR*, vol. 52, no. 4, pp. 1–43, 2019.

[42] L. Lin, S. Qiu, Z. Yu, L. You, X. Long, X. Sun, J. Xu, and Z. Wang, "Aiacc-training: Optimizing distributed deep learning training through multi-streamed and concurrent gradient communications," in *Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022.

[43] Q. Xu, S. Li, C. Gong, and Y. You, "An efficient 2d method for training super-large deep learning models," *arXiv preprint arXiv:2104.05343*, 2021.

[44] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *SC*, 2021.

[45] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 431–445.

[46] A. Kosson, V. Chiley, A. Venigalla, J. Hestness, and U. Koster, "Pipelined backpropagation at scale: training large models without batches," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 479–501, 2021.

[47] C. He, S. Li, M. Soltanolkotabi, and S. Avestimehr, "Pipetransformer: Automated elastic pipelining for distributed training of transformers," *arXiv preprint arXiv:2102.03161*, 2021.

[48] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "Terapipe: Token-level pipeline parallelism for training large-scale language models," in *ICML*. PMLR, 2021, pp. 6543–6552.

[49] D. Narayanan, A. Phanishayee *et al.*, "Memory-efficient pipeline-parallel dnn training," in *ICML*. PMLR, 2021, pp. 7937–7947.

[50] 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 *MICRO*. IEEE, 2016, pp. 1–13.

[51] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *PPoPP*, 2018, pp. 41–53.

[52] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based gpu memory management for deep learning," in *ASPLOS*, 2020, pp. 891–905.

[53] J. Lin, A. Yang, J. Bai *et al.*, "M6-10t: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining," *arXiv*, 2021.

[54] O. Beaumont, L. Eyraud-Dubois, and A. Shilova, "Efficient combination of rematerialization and offloading for training dnns," *Advances in Neural Information Processing Systems*, vol. 34, pp. 23 844–23 857, 2021.

[55] S. Singh and A. Bhatele, "Axonn: An asynchronous, message-driven parallel framework for extreme-scale deep learning," *arXiv preprint arXiv:2110.13005*, 2021.

[56] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, "Varuna: scalable, low-cost training of massive deep learning models," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 472–487.

[57] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei *et al.*, "Dorylus: Affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads," in *OSDI*, 2021.

[58] N. Namashivayam, B. Cernohous, K. Kandalla, D. Pou, J. Robichaux, J. Dinan, and M. Pagel, "Symmetric memory partitions in openshmem:

A case study with intel knl," in *Workshop on OpenSHMEM and Related Technologies*. Springer, 2017, pp. 3–18.

[59] S. Qiu, L. You, and Z. Wang, "Optimizing sparse matrix multiplications for graph neural networks," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2022, pp. 101–117.

[60] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[61] K. Vora, S. C. Koduru, and R. Gupta, "Aspire: exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm," in *OOPSLA*, 2014.

# Appendix: Artifact Description/Artifact Evaluation

## SUMMARY OF THE EXPERIMENTS REPORTED

### Summary

StrongHold aims to increase the trainable model size through dynamic offloading. We evaluate StrongHold on the largest trainable deep neural network (DNN) size on a single GPU (NVIDIA V100) and a distributed environment with 8 A10 GPU servers. We also report the training throughput and compare StrongHold against Megatron-LM, L2L, ZeRO-offload and ZeRO-Infinity, and the ZeRO training schemes (in the distributed training setup).

### Artifact Checklist

This artifact describes the tools and code used in our evaluation. The main component is a docker image that includes a detailed README, scripts to generate and run all experiments, and the required installed libraries. It requires Docker Engine to run.

Evaluating this artifact requires an Intel multicore processor and an NVIDIA GPU. The scripts will generate the main results from our paper evaluation and produce the diagrams.

**Hardware:** Our work was tested on two GPU platforms. The first is a single node GPU server with 2x 24-core Intel Xeon Platinum 8163 CPU at 2.50GHz and 755GB of DDR4 RAM, and a 32GB NVIDIA Tesla V100 GPU. The second is an 8-node distributed GPU cluster, where each node has a 24GB NVIDIA Ampere A10 GPU and 2x 64-core Intel Xeon Platinum 8369B CPUs at 2.90 GHz with 1TB of DDR4 RAM. It is possible to evaluate our approach on a different NVIDIA GPU platform, but the results may deviate from the ones reported in the paper.

**Operating System:** Ubuntu 20.04 with Linux kernel 4.19.91. Our Docker image was tested on a host machine running Linux with CUDA>=11.06

**Software:** Running the artifact requires Docker, all other software dependencies are included in the Docker image or installed when running the included scripts.

**How much disk space required (approximately)?:** 500 GB

**How much time is needed to prepare workflow (approximately)?:** 2 hours

**How much time is needed to complete experiments (approximately)?:** 6 hours

**Publicly available?:** Yes

**Code licenses (if publicly available)?:** MIT License

**GitHub Link:** https://github.com/strongh2/sc22-ae, with DOI:10.5281/zenodo.6887214

### Prepared Notebook

In addition to the docker image, we also provide a preconfigured server with a 32G V100 GPU to make it easier to evaluate the AE. This server allows the use of an interactive Jupyter notebook to work through the main evaluation results. We recommend using the preconfigured notebook for AE evaluation. The notebook can be accessed using the following link (via a web browser) and password:
URL: http://47.111.26.83:8888/notebooks/sc22ae.ipynb
Password: wE6jszc

### Installation

We provide an "out-of-box" Docker image to run the AE scripts. We recommend the installation of Docker Community Edition (CE). Instructions for various platforms can be found here: https://docs.docker.com/get-docker/. This section provides instructions for setting up the host machine for Ubuntu, on which we tested our artifact.

Run the following command in a Linux terminal:

```
$ sudo apt-get update && sudo apt-get install -y docker.io
```

Running the Docker command requires root privileges. It can be run by a user without root privileges by adding the username to the Docker group:

```
$ sudo usermod -a -G docker $USER
```

### Setup GPU Environment on the Host Machine

Run the following commands to enable GPU in Docker environment with NVIDIA container toolkit on the host machine. Details are from https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#docker.

```
$ distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
```

```
$ curl -fsSL \
https://nvidia.github.io/libnvidia-container/gpgkey | \
sudo gpg –dearmor -o \
/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg
```

```
$ curl -s -L https://nvidia.github.io/libnvidia-container/\
$distribution/libnvidia-container.list | \
sed ‘sdeb https://deb [signed-by=/usr/share/keyrings/\
nvidia-container-toolkit-keyring.gpg] https://g’ | \
tee /etc/apt/sources.list.d/nvidia-container-toolkit.list
```

Then, install the nvidia-docker2 package after updating the package listing:

```
$ sudo apt-get update && sudo apt-get install -y
nvidia-docker2 && sudo systemctl restart docker
```

Next, test if the GPU running environment is successfully configured. A working setup can show the GPU information by running a base CUDA container:

```
$ sudo docker run –rm –gpus all
nvidia/cuda:11.0.3-base-ubuntu20.04 nvidia-smi
```

### Setup Docker Image on a Single Node

In this section, we describe how to set up the docker image on a single GPU server.

Our docker image can be downloaded using the following command:

```
$ docker pull strongh/sc22-ae:latest
```

After obtaining the docker image, run:

```
$ docker run -it -P -w /home/sys/STRONGHOLD –name=stronghold –network=host –gpus=all –ipc=host strongh/sc22-ae /bin/bash
```

**Download data:** We do not ship the training data to reduce the docker image size. However, this can be downloaded as:

```
$ wget -c https://dumps.wikimedia.org/enwiki/ latest/enwiki-latest-pages-articles.xml.bz2
```

Unpack the data using:

```
$ python3 -m wikiextractor.WikiExtractor -o text –json ./enwiki-latest-pages-articles.xml.bz2
```

## Experiment Workflow

The evaluation script is located at the *examples* folder, which should be executed from the parent directory. The script contains a number of configurable arguments to change the DNN setup. The script itself should self explain these parameters, which are summarized as follow:

(1) number of layers, (default=16)
(2) hidden size, (default=2048)
(3) number of heads, (default=16)
(4) sequence length, (default=1024)
(5) batch size, (default=4)
(6) window size, (default=4) (only for StrongHold)

The script can be executed by:

```
$ ./examples/run.sh -m [METHOD] -l [NUM_LAYERS] -h [HIDDEN_SIZE] -b [BATCH_SIZE] -w [WINDOW_SIZE]
```

where the [METHOD] argument takes the following values: `megatron-lm`, `l2l`, `zero-offload`, `zero-infinity`, `stronghold` and `all`. Using `all` to automatically evaluate all approaches; the [NUM_LAYERS], [HIDDEN_SIZE], [BATCH_SIZE] and [WINDOW_SIZE] take integers, indicating the number of transformer layers, the hidden size of each layer, the training batch size and offloading window size, separately.

**Results:** Evaluation results will be saved into the `results` folder, formatted as log_[METHOD]_l-[NUM_LAYERS]_h-[HIDDEN_SIZE]_bs-[BATCH_SIZE]_ws-[WINDOW_SIZE]_TIMESTAMP.txt.

**Cases:** Here lists the core testing cases. After each execution is completed, run `./examples/case*_extract.sh` for log analysis, and `./examples/case*_draw.py` for auto-generated diagrams. * means the order of the case.

(1). The largest trainable model size (Fig 6a in Sec VI.A)

- to increase the layers or hidden size before 'OOM' error occurs.
- `./examples/fig6a.sh`

(2). Throughput on the largest trainable model size supported by each baseline (Fig 7a in Sec VI.B)

- to execute this script based on the fig6a's values explored.
- `./examples/fig7a.sh`

(3). Throughput on the largest trainable model size of Megatron-LM (Fig 8a in Sec VI.B)

- to measure the degree of performance loss, compared with the existing offloading works.
- `./examples/fig8a.sh`

(4). Nearly linear scaling as model size increases (Fig 8b in Sec VI.B)

- to measure the degree of the offloading overhead.
- `./examples/fig8b.sh`

(5). Impact of working window size (Fig 9 in Sec VI.C)

- `./examples/fig9.sh`

## Evaluation and Expected Results

We measure the throughput by running each DNN for 10 iterations after the warmup phase to reduce the experimental overhead. We run each evaluation setup 10 times and report the geometric mean across 10 runs.

Generating results for all experiments takes approximately 24 hours, depending on the evaluation hardware. The results directory includes the author's results used in the paper submission. However, the results can vary depending on GPUs, CPU RAM, network bandwidth, and delay.

## Notes

For more detailed information about the artifact and its evaluation process, read the `README.md` located in the docker image, which will also be published on the project website before the AE process.

## Methodology

The artifact of this paper was reviewed according to the following guidelines: http://cTuning.org/ae/reviewing20190109.html, https://www.acm.org/publications/policies/artifact-review-badging.

## AUTHOR-CREATED OR MODIFIED ARTIFACTS:

**Artifact 1**

Persistent ID: `DOI:10.5281/zenodo.6887214;https://github.com/strongh2/sc22-ae`

Artifact name: stronghold-sc22-ae

*Reproduction of the artifact with container:*

1.1 download this docker image from the official hub website.

$ docker pull strongh/sc22-ae:latest

1.2 create a new docker container based on this image.

$ docker run -it -P -w /home/sys/STRONGHOLD –name=aetesting –network=host –gpus=all –ipc=host strongh/sc22-ae:latest /bin/bash

1.3 check the runtime environment.

At this point, we believe the docker container is launched successfully, and the current terminal focus should be at the {/home/sys/STRONGHOLD} folder with {(py3.9.10)} virtual python environment, shown as

$ (py3.9.10) root@??:/home/sys/STRONGHOLD#

If not at the STRONGHOLD folder, please use the cd {/home/sys/STRONGHOLD} command to change the current location.

If not at {(py3.9.10)} virtual python environment, please use {pyenv activate py3.9.10} to enter into py3.9.10 virtual env.

Note: if the host server's CUDA driver is lower than 11.4 or errors about torch version occur, please see the {README} in GitHub to reinstall torch, torchvision, apex libraries.

2.1 Follow the {Experiment Workflow} section to run the testing cases scripts.

(1). The largest trainable model size (Fig 6a in Sec VI.A)

$ ./examples/fig6a.sh && ./examples/case1_extract.sh && python ./examples/case1_draw.py

(2). Throughput on the largest trainable model size supported by each baseline (Fig 7a in Sec VI.B)

$ ./examples/fig7a.sh && ./examples/case2_extract.sh && python ./examples/case2_draw.py

(3). Throughput on the largest trainable model size of Megatron-LM (Fig 8a in Sec VI.B)

$ ./examples/fig8a.sh && ./examples/case3_extract.sh && python ./examples/case3_draw.py

(4). Nearly linear scaling as model size increases (Fig 8b in Sec VI.B)

$ ./examples/fig8b.sh && ./examples/case4_extract.sh && python ./examples/case4_draw.py

(5). Impact of working window size (Fig 9 in Sec VI.C)

$ ./examples/fig9.sh && ./examples/case5_extract.sh && python ./examples/case5_draw.py