SC24: International Conference for High Performance Computing, Networking, Storage and Analysis | 979-8-3503-5291-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/SC41406.2024.00100

Efficient Tensor Offloading for Large Deep-Learning Model Training based on Compute Express Link

Dong Xu
University of California, Merced
dxu17@ucmerced.edu

Yuan Feng
University of California, Merced
yfeng44@ucmerced.edu

Kwangsik Shin
SK hynix
kwangsik.shin@sk.com

Daewoo Kim
SK hynix
daewoo022.kim@sk.com

Hyeran Jeon
University of California, Merced
hjeon7@ucmerced.edu

Dong Li
University of California, Merced
dli35@ucmerced.edu

***Abstract*—The deep learning models (DL) are becoming bigger, easily beyond the memory capacity of a single accelerator. The recent progress in large DL training utilizes CPU memory as an extension of accelerator memory and offloads tensors to CPU memory to save accelerator memory. This solution transfers tensors between the two memories, creating a major performance bottleneck. We identify two problems during tensor transfers: (1) the coarse-grained tensor transfer creating difficulty in hiding transfer overhead, and (2) the redundant transfer that unnecessarily migrates value-unchanged bytes from CPU to accelerator. We introduce a cache coherence interconnect based on Compute Express Link (CXL) to build a cache coherence domain between CPU memory and accelerator memory. By slightly extending CXL to support an update cache-coherence protocol and avoiding unnecessary data transfers, we reduce training time by 33.7% (up to 55.4%) without changing model convergence and accuracy, compared with the state-of-the-art work in DeepSpeed [62].**

I. Introduction

The deep learning (DL) models are becoming larger. In the last four years, the model size increases by at least 1,000x. In 2018, Bert [24], a DL model with over 300M parameters, was the largest model. Just within one year, Turing-NLG [61], T5 [77], GPT-2 [12], and Megatron-LM [92], a set of transformer-based models, increase the number of parameters to tens of billions. In 2022, BLOOM, one of the largest models, has 176B parameters [44]. Utilizing large models leads to gains in model quality [10] and resource utilization [35]. The momentum of using them may continue.

However, training those large models faces a memory capacity wall. The model states (such as parameters, gradients, and optimizer states) and intermediate results (such as activation) can easily go beyond the memory capacity of an accelerator (e.g., up to 180GB in a NVIDIA Blackwell GPU [65]). Distributed model-training technologies, such as pipeline model parallelism [32], tensor parallelism [92], and ZeRO data parallelism [78], are able to split the model states across multiple accelerators or reduce data redundancy across accelerators, hence addressing the memory capacity problem faced by a single accelerator. In practice, using a combination of multiple distributed training technologies enables large model training by leveraging aggregated accelerator memory on multiple accelerators. However, the distributed training relies on expensive hardware resources. For example, training 175B-parameter GPT-3 requires 112 NVIDIA A100 GPUs (each with 80GB HBM) to accommodate memory consumption, easily causing millions of dollars, which is not affordable by many users. Even using half-precision and loss-scaling to save memory and using eight V100 GPU on the cloud, it still takes hundreds of thousands of dollars [82]. Large production costs and pervasive usage of large DL models motivate various hardware- and software-based solutions [9], [38], [47], [51], [69], [104], [120].

Use of heterogeneous memory for model training breaks the memory capacity wall without adding accelerators. By using the CPU memory as an extension of the accelerator memory, this technology offloads tensors (such as parameters [30], [73], [74], [79], [85], [90], [103], optimizer states [79], [87], and/or activation memory [79]) to the CPU memory.

Problems. Using heterogeneous memory (HM) introduces data movement overhead: Tensors offloaded to the CPU memory must be transferred back to accelerator for model training, and the transfer must be timely to avoid the reduction of computation efficiency on accelerator. Existing efforts prefetch tensors from the CPU memory to a GPU buffer and overlap tensor transfers with computation [30], [31], [74], [85], [103] based on the knowledge of the DL topology. To maximize the saving of the accelerator memory, the recent effort [79], [87] offloads optimizer computation to CPU to completely remove optimizer states from GPU. However, to make those solutions effective, one must use a large batch size or large layer-wise computation in the DL model to provide overlapping opportunities, because of suboptimal data partitioning and limited PCIe bandwidth. Given a model with billions of parameters (or hundreds of millions of parameters per accelerator with model parallelism), the existing efforts transfer hundreds of MB or even a few GB tensors for each layer, easily taking

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

$\sim$10 or $\sim$100 of milliseconds on a PCIe 3.0 [71] (or PCIe 5.0 [72]) interconnect, which is typically longer than layer-wise computation time. The mismatch between tensor transfer time and computation time brings challenges to hide the overhead of tensor transfers. Hiding or reducing this overhead is the key to use HM for large model training.

Major insights. We identify two fundamental problems during tensor transfers for large DL model training: the *coarse-grained tensor transfer* and *redundant tensor transfer*. Identifying the two problems provides a new angle to examine how to hide or reduce the tensor transfer overhead.

The coarse-grained tensor transfer means that the tensor transfer between CPU and accelerator occurs at the granularity of a group of variables (e.g., a group of parameters aligned with the block size in a transformer). Such a tensor transfer is caused by software management that periodically transfers a group of variables (instead of individual variables) to make the best usage of interconnect bandwidth. However, this transfer method leads to long transfer time per transfer, which is difficult to be overlapped with computation.

The redundant tensor transfer means unnecessary data transfer causing communication overhead. Our profiling on Bert shows that 44.5% of parameters do not change values across two consecutive training steps throughout some training steps; 80% of the updated parameters are modified only in the least significant two bytes. This limited byte update is a common phenomenon in most DL training. This phenomenon is more common in large DL models using fine-tuning. During the fine-tuning phase, the weights and parameters do not significantly change to avoid catastrophic forgetting [41]. Since CPU and accelerator hold the gradient and parameter tensors in a former training step, the unmodified parts in these tensors do not need to be transferred. The unmodified bytes consume PCI-e bandwidth and significantly delay communication (69.5% in our experiments in Section II).

Solution. To address the above problems, we introduce *Te*nsor-*C*XL-*O*ffload (or TECO), an HM system based on a cache-coherent interconnect — Compute Express Link (CXL), aiming to hide the tensor transfer overhead to train large DL.

To address the problem of coarse-grained tensor transfer, TECO puts the CPU cache and a part of the accelerator memory into the same CXL coherent domain. That part of the accelerator memory is used as a giant cache of the CPU memory. We recognize the limitation of CXL to hide communication time because of an invalidation-based cache-coherence. This protocol best fits for large-scale communications where it is hard to keep track of all data sharers. For the small-scale communications that have a clear data producer and a consumer (e.g., CPU-GPU DL training), sending an invalidation message and the updated data in separate CXL packets only demands more PCI-e bandwidth and unnecessarily delays the data transfer. Therefore, TECO extends CXL to be configurable to support either update- or invalidation-based coherence based on the execution scheme. Using the new protocol, TECO decomposes coarse-grained tensor transfers into cache line-grained ones to maximize the overlap between tensor transfers and computation.

To address the problem of the redundant tensor transfer, TECO excludes useless bytes from the transfer. Based on the statistics of the number of bytes per data (e.g., DL parameters) updated during application execution, we configure the CXL host module to collect only the updated bytes from each parameter and pack multiple (trimmed) parameters in a CXL packet. The number of bytes to drop from each 4-byte unit data is given by the user and is passed to the accelerator-side CXL module via a special API. With this information, the accelerator parses each CXL packet and merges the updated data with the original 4-byte data residing in the accelerator memory. We call this process, *dirty-byte aggregation*, and design an aggregator (to encapsulate multiple data into one CXL packet) on the sender device and a disaggregator (to parse individual data from a packet) on the receiver device.

We summarize the major contributions as follows.

- We use a cache coherent HM consisting of the memories of accelerator and CPU to train large DL, based on which we provide a new method to reduce data movement overhead;
- We characterize tensor transfers during large model training with an industry-quality, HM-based training solution (ZeRO-Offload [87] from DeepSpeed [62]), and identify two problems that make tensor transfers the major performance bottleneck for the HM-based training;
- We leverage and extend CXL to address the two tensor-transfer problems by reducing tensor transfer granularity and avoiding useless transfer;
- To our best knowledge, this is the first study demonstrating the effectiveness of the extended CXL giant cache model.
- Evaluating with large transformers and a graphic neural network, we show that compared with ZeRO-Offload, TECO reduces training time by 33.7% on average (up to 55.4%) without changing model convergence and accuracy; TECO reduces communication overhead by 93.7% on average (up to 100%).

II. Background

A. Training Large Deep Learning Models on HM

Using the CPU memory as an extension to the accelerator memory, the existing solutions use various strategies to decide which tensors should be offloaded and how to hide transfer overhead. We review a representative and state-of-the-art effort, ZeRO-Offload. ZeRO-Offload represents a class of work (including ZeRO-Infinity [79], [95]) selectively offloading tensors to the CPU memory but stores all parameters on accelerator. ZeRo-Offload performs better than Unified Virtual Memory [74], [85].

ZeRO-Offload stores the gradients and optimizer states in the CPU memory, and stores the parameters in the GPU memory. GPU also has a small gradient buffer. There are five phases in a training step in ZeRO-Offload (see Figure 1).

- **Phase 1:** Forward propagation on GPU using recently updated parameters on the GPU memory. These parameters are transferred from the CPU memory;

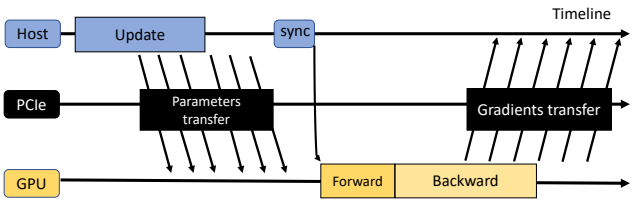


Fig. 1: Overview of ZeRO-Offload.

- **Phase 2:** Backward propagation on GPU, generating gradients in the gradient buffer on the GPU memory.
- **Phase 3:** During the backward propagation, the gradient buffer is periodically filled and flushed (i.e., transferring gradients) from GPU to the CPU memory.
- **Phases 4 and 5:** After collecting all gradients at the end of a training step, the gradients are clipped to be bounded within a certain range on CPU. The new gradients are used in the ADAM optimizer on CPU to generate new parameters on the CPU memory. The new parameters are then transferred from CPU to the GPU memory for the next training step.

In ZeRO-Offload, the gradients transfers from GPU to CPU are partially exposed to the critical path. For the parameters, ZeRO-Offload uses a double-buffer technique on CPU to hide the transfer overhead: while CPU fills one buffer with new parameters, the other is used for parameter transfers from CPU to GPU. However, the buffer filling is much faster than the parameter transfer. As a result, the parameter transfer is largely exposed to the critical path. Although using the "one-step delayed parameter update" technique (DPU) [87], the parameter transfer can overlap with GPU computation, the effectiveness of this technique requires significantly large batch sizes to achieve enough arithmetic intensity on GPU. However, the batch size per GPU is limited by the maximum global batch size that can be used during the training without sacrificing convergence efficiency [5], [36], [121]: since the global batch size cannot be increased indefinitely without slowing down model convergence, training on GPUs (especially in a large scale) often forces the batch size per GPU to be very small [102], which invalidates DPU effectiveness. Also, DPU itself raises the risk of changing DL model convergence.

B. Compute Express Link

CXL is proposed to be used as interconnect between processors, memory expansion, and accelerators. CXL is built on top of PCIe, and develops a custom higher-level protocol stack. CXL intends to get lower latency of fine-grained memory accesses (i.e., cache line-level accesses), while provides sufficient peak bandwidth (e.g., about 90% of the underlying serial bus protocol bandwidth). CXL includes three sub-protocols. One of them, `CXL.cache`, is related to this paper. `CXL.cache` is an agent coherency protocol permitting device caching of the host memory. CXL uses the MESI protocol managed by hardware to ensure coherence for `CXL.cache`.

Giant cache model in CXL. The CXL specification includes a giant-cache model [21]. When the memory footprint of a workload exceeds the capacity of accelerator memory, this memory can work as a giant cache. In particular, the full dataset resides in the CPU memory, but the subsets of the full dataset are cycled through the giant cache as the computation proceeds. The cache in the host system and the giant cache in the accelerator are in the same coherence domain and work as distributed caches for the CPU memory.

The recent surge of domain-specific accelerators (especially for AI workloads) [25], [27]–[29], [52], [75], [80], [111], makes the implementation of TECO highly feasible. This feasibility is further enabled by the recent initiative from NVIDIA for customization of AI chips to meet users' growing needs [48].

TABLE I: Percentage of training time used for communication. We report the communication time exposed to the critical path.

Batch size	4	8	16	20
Overhead (ZeRO-Offload)	42.24%	37.87%	28.65%	25.95%

III. Motivation

We study the training performance with ZeRO-Offload. We use pre-trained Bert-large-cased [2], a large transformer model with 334M parameters and 24 transformer blocks. We fine-tune Bert-large-cased on the IMDB dataset [54]. We use a system with Intel Xeon 6120 CPU, one Tesla V100 GPU with 32GB memory, PyTorch 1.10.0, and PCIe 3.0 for data transfer between CPU and GPU. We use ZeRO-Offload 0.3.15. We vary the batch size to study its impact on tensor offloading.

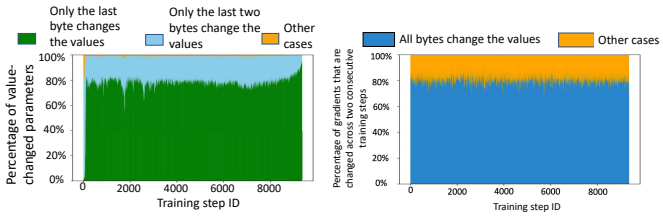
Quantification of communication time. We measure tensor transfer time exposed to the critical path. See Table I.

Observation 1: the communication overhead takes a rather large portion of total training time.

For example, when the batch size is 4, the communication overhead takes 42.2% and 59.7% of total training time with ZeRO-Offload. This large overhead comes from (1) some gradient transfers from GPU to CPU are exposed to the critical path and the CPU computation must wait for the gradient transfers to finish before it starts, and (2) the parameter transfers from CPU to GPU are largely exposed to the critical path, because the double-buffer is not effective and DPU partially fails because of low arithmetic intensity on GPU.

Furthermore, we notice that when the batch size becomes larger, the communication overhead takes a smaller portion of the training time. For example, for ZeRO-Offload, the overhead reduces from 42% to 26% when the batch size increases from 4 to 20. This is because GPU computation complexity becomes larger while the communication time remains the same, leading to smaller communication overhead. Nevertheless, the communication overhead still takes a large portion of training time. Note that increasing the batch size increases memory consumption, hence limiting the model size we can train and deteriorating the memory capacity problem.

Variance of parameter/gradient values across training steps. We examine the values of parameters and gradients in each training step. Each parameter or gradient is a 4-byte floating point number. We study that in a training step i, among



(a) Parameters (b) Gradients

Fig. 2: The distribution of value-changed bytes in parameters (a) and gradients (b) across two consecutive training steps.

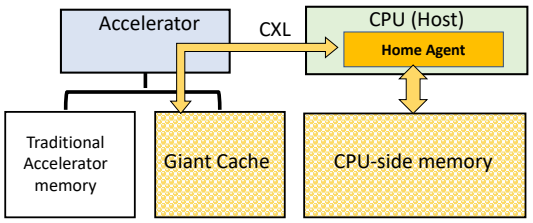


Fig. 3: Memory organization in TECO. The yellow-shadowed components are in the same coherent domain.

the 4 bytes in a parameter (or a gradient) how many of them change values, compared with the prior step $i-1$. In other words, we count the number of *value-changed* bytes in each parameter (or each gradient) across two consecutive training steps. Given four bytes in a parameter (or a gradient), we classify the distribution of value-changed bytes in those four bytes into three cases: (1) only the last byte changes the value, (2) only the last two bytes change, and (3) other distributions. We train (fine-tune) a pre-trained Bert-large-cased to converge using 9,870 training steps. Figure 2.(a)-(b) show that among those value-changed parameters (or gradients), how the above three cases are distributed in training steps.

Observation 2: Across two consecutive training steps, among those parameters that change values, most of them change values only in the last two bytes.

Figure 2 (a) shows that about 80% of parameters belong to Case (1) (i.e., only the last byte changes values). An FP32 parameter has 4 bytes where the 1st byte plus one bit are for exponent and sign, and the remaining bits are for mantissa. For most of the parameters, there is few value-change in exponent and sign across training steps, but there are frequent value-changes in mantissa, leading to frequent value-changes in the last two bytes. Also, we notice that the first two cases become more common when the training is close to converge. Figure 2 (b) shows the distribution of value-changed bytes in gradients. Different from parameters, all bytes in gradients frequently change values across training steps.

IV. TECO: Tensor-CXL-Offload

A. Overall Architecture

1) Memory Organization: A part of the accelerator's global memory is a cache of CPU memory and is mapped to the CXL coherent domain using the giant cache model in CXL. This cache is used for parameters and gradients transferred between CPU and accelerator. The other tensors are allocated to the remaining accelerator memory which is used as the traditional non-coherent memory. DL training on accelerator is executed against both parts of the memory. See Figure 3.

The giant cache size is configured to be large enough to accommodate tensors transferred between accelerator and CPU, and there is no cache capacity (or conflict) miss during accelerator computation.

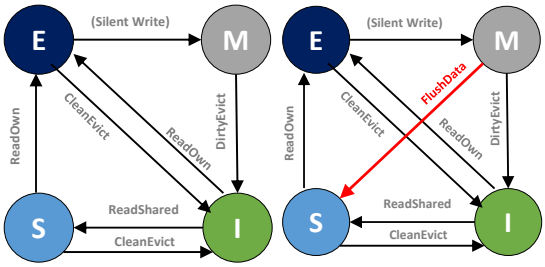
Before starting the DL training, once the size of batch, DL model, and training data are determined, the giant cache size is configured by the user and does not change during the DL training. For Zero-Offload, this size is the size of parameters in the accelerator plus the size of the gradient buffer (the size of the gradient buffer is a configurable parameter in Zero-Offload).

There can be various ways to configure giant cache size. We use *resizable Base Address Register (BAR)* [70], which enables faster communication between host CPU and PCIe devices by mapping configurable memory regions of the devices to the system memory map. Once the size is set, that amount of space is separately marked as the giant cache. Then, the CXL controller and home agent handle the coherent tensor movement between the giant cache and CPU memory (Section IV-A2).

2) Coherence and Caching Mechanisms: **Limitation of CXL.** CXL uses invalidation-based coherence protocol (MESI). Upon every cache line update, only *invalidation message* is sent to the shared opposite party (CPU cache for the giant cache or vice versa). Later, when the shared party issues a memory load for the data, the updated value is transferred via PCIe. Thus, the expensive PCIe transfer time is included in the critical path. According to our evaluation, this on-demand data transfer increases training time by 56.6% on average (up to 99.7% in the case of T5-large model with 737M parameters), compared to when the updated data is sent at the time of invalidation. This motivates us to optimize the CXL coherence protocol.

CXL extension. We extend CXL to use an update-based cache coherence protocol. This means that cache lines are transferred at the updating time between CPU cache and accelerator's giant cache by `CXL.cache`. As the CPU cache is typically smaller than the giant cache, only the lines residing in the CPU cache are updated by the update protocol. For the other lines, the CPU cache simply ignores the update messages. Our proposed extension enables cache line level tensor transfer over time. Thus, we can support fine-grained migration and better compute and data transfer overlapping.

Figure 4 illustrates the revised protocol transitions. This figure is based on the state transition figure in the CXL spec-



(a) MESI in CXL (b) The new protocol

Fig. 4: Minor revision of the MESI state transition.

ification [23]. As highlighted with a red arrow, `Modified` state can be immediately transitioned to `Shared` state upon the CXL home agent's approval. When a cache line is updated on CPU, CPU sends a message to the home agent to see if this cache line is in the giant cache domain (see the first two red arrows in Figure 5). The above is the only change to the invalidation protocol, and the other transitions remain the same.

Figure 5 depicts the state transition in detail when a parameter cache line is updated. The accelerator's giant cache and CPU cache are treated as peer caches. The home agent, as depicted in the CXL specification, manages the coherence between the peer caches of CXL device. C_S and G_S represent the state of CPU cache line and accelerator cache line respectively. `C` is the cache line that CPU updates. When the training starts, the giant cache has a copy of the parameters. At the beginning, the CPU state of `C` (i.e., C_S) is I, since `C` is not in the CPU cache. The accelerator state of `C` (i.e., G_S) is E. When ① happens, C_S becomes E after sending a ReadOwn CXL coherence message. When ② happens, C_S transits to M from E, and then transits to S after receiving the Go-Flush message. ② has such a transition because of the revised protocol: `C` is in the giant cache domain and updated by CPU. ① to ③ represents the process of updating the parameters. If the CPU evicts `C` or flushes all the cache lines, C_S transits to I from S and G_S transits to E from S. The flush happens only once at each training iteration to guarantee all the updated parameters are sent out. When the accelerator reads `C`, G_S remains E. The accelerator only reads the parameters and never updates them, and this means that the CPU and accelerator never concurrently update the parameters.

We introduce a function, `CXLFENCE()`, to ensure the completion of in-flight CXL cache coherent traffic. CXLFENCE is used to enforce the memory consistency. `CXLFENCE()` is implemented by extending accelerator synchronization API such as `cudaDeviceSynchronize()` in GPU, which is commonly used to check the completion of data transfer or CUDA kernel call. `cudaDeviceSynchronize()` is supported by the existing GPU driver to check the DMA-based copy engine and detect if there is incomplete data transfer in PCIe. `CXLFENCE()` works similar to `cudaDeviceSynchronize()` but it only guarantees

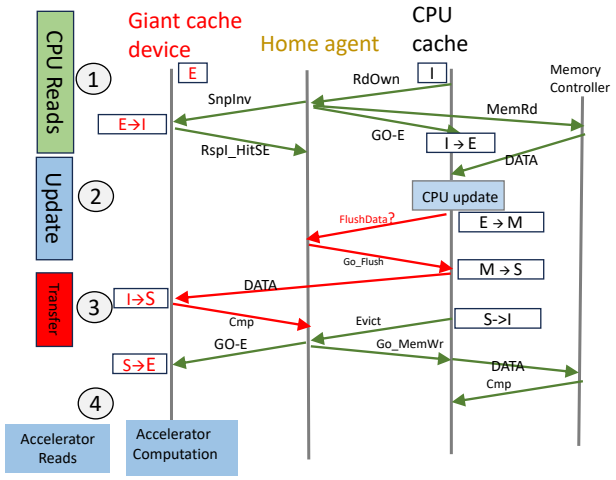


Fig. 5: Transition flow when updating a parameter cache line.

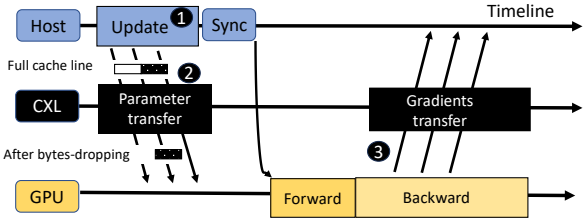
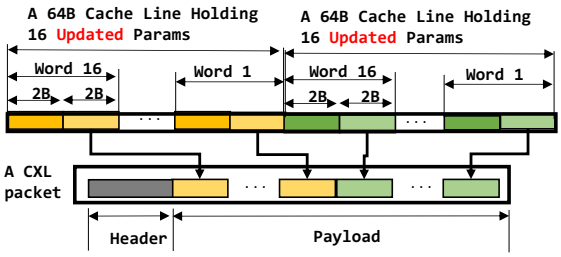


Fig. 6: Dataflow when applying TECO to ZeRO-Offload.

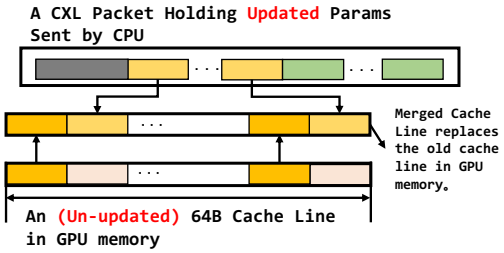
the CXL coherence traffic by checking the status of CXL controller and home agent.

One challenge to designing a giant cache is the large size of snoop filter (or coherence directory) as the sharer information of individual cache lines should be maintained in the filter. TECO does not have the snoop filter design problem. As the giant cache size is configured to afford gradients and parameters, and CPU and accelerator have a clear producer-consumer relationship for each data, it is not necessary to keep track of data sharers and their coherence status with a snoop filter. In particular, when CPU updates a tensor element cached on the giant cache, based on the DL domain knowledge we know that the tensor element must be a parameter, and the parameter copy on the accelerator must be updated due to the update protocol and hence in `Shared` state. Hence, there is no need to apply the snoop filters. Therefore, TECO does not maintain the snoop filters for the giant cache, which saves memory space and avoids long snoop latency.

For the application that does not have a clear producer-consumer relationship (e.g., having more than two sharers) or multiple sharers updating the cache line concurrently, TECO goes back to using the invalidation protocol and snoop filter. This can be implemented by modifying the CXL home agent. As shown in Figure 5, all the transitions are managed by the home agent. By disabling the immediate FlushData transition upon data update, the update-based transitions can be disabled.



(a) Aggregator in the CPU-side CXL Module



(b) Disaggregator in the accelerator-side CXL Module

Fig. 7: Aggregator and disaggregator

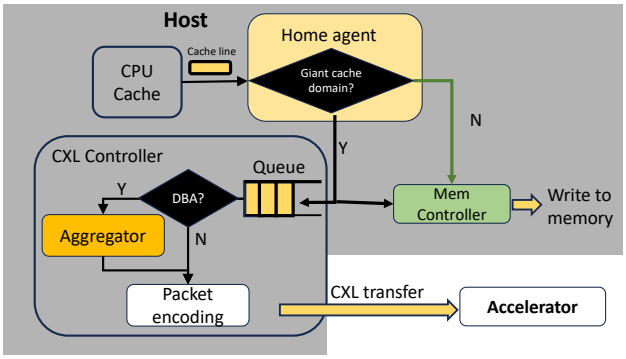


Fig. 8: Workflow of updating a cache line.

B. Applying TECO

Applying TECO to ZeRO-Offload. With TECO, parameters and gradients use the giant cache. Before training, the CPU memory has a whole copy of parameters and gradients. Figure 6 depicts the dataflow. TECO reduces the parameter transfer overhead. With TECO, once any parameter is updated on CPU, it is transferred to the GPU memory according to MESI-update (❶ and ❷). There is no need to use the double-buffer technique. Therefore, we can avoid the frequent synchronization between the two buffers and reduce software complexity. Also, with TECO, there is no need to explicitly call parameter transfers. Only when all parameters are updated, `CXLFENCE()` is called once to enforce the completion of coherent parameter updates to the giant cache.

The above method does not increase the communication volume. In particular, when multiple parameters fall into the same cache line, any update to a parameter in the cache line can cause a transfer of the cache line from CPU. This means a cache line containing multiple parameters may be transferred multiple times especially when there is a long time interval between the updates of parameters, while the original ZeRO-Offload transfers the whole cache line only once. However, the computation for parameter update is commonly based on vectorization. As a result, multiple parameters are updated at the same time, causing only one transfer of the cache line.

TECO also reduces gradient transfer overheads. With TECO, the gradient transfer does not need to wait for the gradient buffer to be fully filled. Instead, the gradient transfer is overlapped with the backward propagation (❸). After the buffer is full with gradients, `CXLFENCE()` must be called to ensure in-flight data transfer is done.

V. Reduction of Communication

This section discusses the dirty-byte aggregation (DBA) in detail. We introduce two components, *Aggregator* and *Disaggregator*, depicted in Figure 7. The figures illustrate (a) how the updated least-significant two bytes of each 4-byte parameter are aggregated from two cache lines into a CXL packet in CPU, and (b) how the aggregated dirty bytes are parsed from the CXL packet in the accelerator and merged with the un-updated cache lines in the giant cache to reconstruct the updated cache lines. Note that the un-updated cache lines remain in the accelerator memory after being used by the prior training step. The gradients transfers from the accelerator to CPU cannot apply DBA, because there is no common byte-update pattern in gradients as shown in Section III.

A. Activation of Dirty-Byte Aggregation

The DBA is activated at runtime. TECO determines the activation of DBA after a specific number of training steps (specified with *act_aft_steps* by the user in an AI model configuration file). *act_aft_steps* is a model dependent hyperparameter. Like other hyperparameters (such as the learning rate, batch size, and number of steps delayed for parameter update [87]), *act_aft_steps* is determined by the user, while the default value is 500 (Section VIII-E). *act_aft_steps* can be tuned using the Bayesian optimization [17], [94].

To indicate the dirty-byte length, we introduce another model-dependent hyperparameter, *dirty_bytes*. *dirty_bytes* is configured by the user. For DL training, it is set to 2, as the parameter-value change happens mostly in the least significant two bytes in consecutive training steps (Section III).

B. Aggregator

When CPU updates a cache line of parameters mapped to the giant cache, the line is transferred to the accelerator according to MESI-update. For each FP32 parameter in 64-byte cache lines, the Aggregator takes the last N bytes (N = *dirty_bytes*), packs them into a CXL packet, and sends it to the accelerator (Figure 7.(a)). The Aggregator and its configuration registers are implemented in the CPU CXL module.

The DL framework (such as PyTorch) uses the configuration register, *DBA register*, in the CXL module through a CXL configuration interface to activate the DBA. The DBA register has four bits: the most significant bit for indicating the activation and the remaining three bits for setting the dirty byte length (0 to 4 bytes). For example, suppose that this register is set with dirty bytes as 2 bytes (i.e., the DBA register is set to 1010_2). For each 64-byte cache line to be sent to the accelerator, the Aggregator takes the least significant two bytes of each 4-byte parameter, aggregates them into a 32-byte payload, and passes it with the cache line address to the CXL Link Layer to create a CXL packet. The CXL Link Layer combines one or multiple 32-byte payloads into one CXL packet depending on the CXL transfer size. We indicate the size of payloads (32-byte aggregated cache lines or a 64-byte unaggregated cache line) by reserving an unused bit in the CXL packet header (the packet header has at least six unused bits). When the DBA register is not set (i.e., the most significant bit is 0), the Aggregator logic is bypassed and all bytes in each cache line are sent through CXL.

Figure 8 shows the workflow of updating a cache line. When the cache line is flushed or evicted from the last level cache on CPU, the CXL home agent on CPU first checks if this cache line is mapped in the giant cache. If not, this cache line is directly written to CPU memory. If yes, this cache line is sent to the transmission queue maintained by the CXL root port. This queue is like the transmit buffer in PCIe [3]. Then, the CXL root port checks if DBA is activated, and sends out the aggregated or full cache line through the CXL link accordingly.

As the only function that the Aggregator does is collecting dirty bytes from a given cache line, it can be implemented with simple logic gates. We implement it by designing a logic that takes the lower N bytes from each 4-byte parameter, where N is indicated by the lower three bits of DBA register, and concatenates them together.

To indicate memory regions mapped to the giant cache on the accelerator, the Aggregator has two registers ("address registers") per cached region, which are set when a tensor is allocated and checked by the CXL host agent when triggering coherent data transfer. Note that the implementation of registers does not change the CPU architecture. TECO leverages unused registers in the CXL specification [19]. The Aggregator utilizes the internal buffers of CXL Link Layer and does not need extra buffer.

C. Disaggregator

During the DL training, the accelerator retains the parameter values used by the prior training step. Once CPU updates the parameters and sends only the dirty bytes through the DBA in a MESI-update message, the accelerator reconstructs the updated parameters by overwriting the corresponding bytes of each parameter in its memory with the dirty bytes sent by CPU. For this task, we design the Disaggregator in the CXL module on the accelerator.

Once the DBA register is set in the CPU CXL module, the CXL host agent asks the accelerator's CXL module to activate the disaggregation by sending the DBA-register value to the accelerator. Once an MESI-update message is delivered to the accelerator, the cache line to update is read from the giant cache in the accelerator memory. If the DBA-register value is 1010_2, from the MESI-update packet, the Disaggregator takes 32 bytes from the payload, and uses every two bytes to overwrite every other two bytes in the 64-byte cache line (see Figure 7.(b)).

This approach leads to one extra read operation per cache line update to merge the dirty bytes with the un-updated cache line in the giant cache. However, the accelerator-side DRAM architecture does not need to be changed because disaggregation and merge are handled in the CXL module, not in the memory. Also, the extra read operations incur almost negligible performance overhead according to our evaluation (Section VIII), because there is a large bandwidth gap between PCIe and the accelerator memory (e.g., GDDR5) and the accelerator memory is not the performance bottleneck.

The Disaggregator's function can be implemented with negligible overhead. For example, we implement it by designing a logic that (1) resets N bytes indicated by the DBA register per 4 bytes of a cache line, (2) shifts the payload by taking every N bytes from the payload and shifting them by $((4 - N) \times W)$ bytes, where W is the corresponding word index in the cache line, and (3) runs OR operations between the cache line and the shifted payload. The Disaggregator assumes that there is an old copy of the parameters in the accelerator memory in order to perform merging, which is true in TECO.

The Aggregator and Disaggregator require little hardware modifications: (1) adding the address and DBA registers, (2) a few logic gates to take the dirty bytes per word in the Aggregator, and (3) merging the disaggregated dirty bytes and the old copy in the accelerator memory in the Disaggregator. The space and power overheads are evaluated in Section VIII-D.

About mixed-precision training. Using mixed-precision to train large DL models is common. In those cases, parameters often have two copies, one in FP32 and the other in FP16 [79], [87]. Using gradients in FP32 and optimizer states in FP32, the parameters in FP32 are updated on CPU, and then converted to FP16 for (back)forward propagation on GPU. Such a conversion does not impact the effectiveness of TECO, because the conversion happens on GPU. As a result, the parameters transfer from CPU to GPU still uses FP32, creating opportunities for the DBA. The conversion must happen on GPU instead of CPU to reduce conversion overhead.

VI. Using TECO in DL Models in Practice

The use of TECO includes minor changes to the DL implementation. Listing 1 gives an example of how to use TECO with ZeRO-Offload for Bert. Using TECO, the user just needs to call `check_activation()` after the backward propagation (Line 5). `check_activation()` determines if the DBA should be activated according to Section V-A. *Only* two lines of code (Lines 1 and 6) are needed.

Listing 1: Using TECO with ZeRO-Offload for Bert. The code from TECO is highlighted in blue.

```
1 from TECO import check_activation
2 . . . # import other packages and preparing the training
3 for i in range(training_steps): # main loop for training
4     . . . # feed the model and compute loss
5     loss.backward()    # CXLFENCE() is called in this function
6     check_activation(i)
7     optimizer.step()  # CXLFENCE() is called at the end in step()
```

Mapping parameters and gradients to the giant cache is hidden from the user. Those tensors are allocated on CPU as usual. `CXLFENCE()`, used after parameter transfers from CPU and gradient transfers from GPU, is also hidden from the user. `CXLFENCE()` is called within the implementation of ZeRO-Offload and PyTorch (Lines 2 and 4). `CXLFENCE()` is called only twice in a training step. Since `CXLFENCE()` can be based on `cudaDeviceSynchronize()`, we measure the time of `cudaDeviceSynchronize()` which takes less than 1% of training time.

VII. TECO Generality

TECO relies on minor hardware changes. This is reasonable, because DL workloads become so important such that recent efforts introduce DL-specific hardware [16], [34], [38], [39], [46], [83], [97]–[99], [101]. Nevertheless, TECO can be generally applicable to other applications. The application that can benefit from TECO should have three characteristics: (1) the application has an iterative structure, and the computation results are iteratively fine-tuned; (2) the application can tolerate computation approximation. Many applications have the above characteristic, including common numerical solvers (e.g., multi-grid solver [13] and conjugate gradient solver [91]); (3) the data transfer between CPU and the accelerator is a performance bottleneck, and *data producer and consumer are explicit*. (1) and (2) are needed to apply the DBA, and (3) is needed to benefit from the extended CXL.

To demonstrate the generality, **we apply TECO to LAMMPS [1] (a molecular dynamics (MD) simulation code)**. We study 3D Lennard-Jones melting simulation with LAMMPS where the accelerator is used for force calculation for a set of molecules. After accelerator computation, the force data is sent to CPU. CPU then updates the molecules' positions and sends them to the accelerator. This code meets the requirement of (1) and (2), and the data transfer takes 27% of the application time with explicit data producer and consumer, meeting the requirement of (3). Applying TECO, LAMMPS gets 21.5% performance improvement, and the communication volume is reduced by 17% by DBA. Among the performance improvement, CXL contributes 78% and DBA contributes 22%. This improvement leads to 5-hour saving in the simulation time.

VIII. Evaluation

A. Experimental Setup

Evaluation method. We design an evaluation infrastructure by interfacing a CPU simulator (gem5-avx [105] v20.0.0) and a GPU simulator (Accel-Sim [37]) via CXL emulation, as illustrated in Figure 9. We use GPU as an example accelerator.

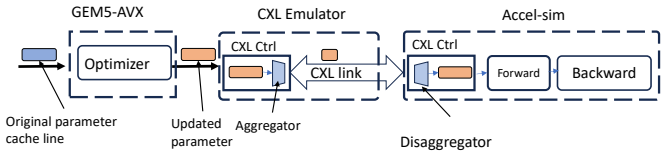


Fig. 9: Overview of the simulation platform.

Using GPU with TECO is possible because of the recent initiative from NVIDIA for customization of AI chips to meet the growing needs of users [48], and the hardware support for the giant cache model is expected to be available in the near future. We emulate PCIe 3.0 with 16 lanes with 16GB/s bandwidth. All data transfer times over the CXL protocol are emulated by assuming to consume 94.3% of PCIe bandwidth [20], [106]. Table II shows the configurations of gem5-avx. Accel-Sim is configured as NVIDIA V100 GPU. The communications over CXL are controlled by a CXL controller with a pending queue of 128 entries. We extend gem5-avx and Accel-Sim to model the CXL controllers.

We use Ubuntu 18.04, PyTorch 1.10.0, CUDA 10.2, DeepSpeed 0.3.15 (including ZeRO-Offload), and Python 3.6.7. Unless indicated otherwise, act_aft_steps and $dirty\_bytes$ are set as 500 and 2 respectively. We change the batch size to evaluate the effectiveness of TECO. Even though we use tensor offloading, the batch sizes are chosen to be within a certain range such that out-of-memory does not happen. Pytorch 2.0 provides an API called torch.compile [59] to optimize the computation graph. However, it can't be used for CPU+GPU training, and only supports computation graph-based optimization on CPU only or GPU only.

Parameter transfer over CXL. CPU runs the parameter update (using the ADAM optimization and AVX512 intrinsic) during the training. TECO uses the update-based cache coherence protocol. Thus, our simulation transfers a cache line when multiple parameters in the cache line are updated using a vectorized instruction (see Section IV-B) and the cache line is written back to the main memory. To measure the transfer time of these cache lines over CXL, we collect the timing and amount of these writebacks by generating a trace of main memory accesses during CPU simulation. The trace contains the timings and addresses of memory loads/stores [40].

The CXL emulator measures the transfer time that does not overlap with CPU computation time and adds it to the gem5-avx simulation time as the final CPU time. Note that the CXL emulator measures the transfer time by considering the aforementioned CXL bandwidth. For TECO-Reduction (using both CXL and DBA), the Aggregator delay is added, which is 1 ns as explained in detail in Section VIII-D. Our emulator is designed based on the fact that CXL is a serial bus [20], [72]. The updated cache lines with different parameters are going through the link one after another in a stream manner.

Gradient transfer over CXL. To simulate the update-based cache coherence protocol, Accel-Sim is modified to transfer the updated gradients over CXL whenever the corresponding cache line is written back to the giant cache region in GPU memory. When TECO-Reduction is used, the Disaggregator's

TABLE II: Gem5-avx configuration.

Processor	48 DerivO3CPU cores, 3.7GHz
Memory systems	8 memory controllers 32GB DDR4-2600Mhz
Cache	I-cache: 8KB/64B-line/8-way L1cache: 8KB/64B-line/8-way L2cache: 64KB/64B-line/16-way L3cache: shared/16MB/64-line/64-way

delay (1 ns) is added as specified in Section VIII-D.

We use ZeRO-Offload as the baseline. Without CXL, the baselines explicitly trigger gradient transfer by the DL model rather than CXL. In evaluation, any technique to hide CPU computation overhead and tensor transfer overhead in the DL training framework (such as DPU in ZeRO-Offload) is employed to enable a fair comparison.

We do not evaluate ZeRO-Infinity [79] (a ZeRO-Offload-like work), because ZeRO-Infinity uses main memory and NVMe SSD based on the assumption that the main memory capacity is not large enough. ZeRO-Infinity regresses to ZeRO-Offload when memory capacity is large enough. CXL memory provides sufficiently large capacity, hence ZeRO-Offload is more appropriate for evaluation.

Workloads. Table III lists DL models we evaluate. They are pre-trained models and we fine-tune them on various datasets.

B. Overall Performance

Speedup. We use two versions of TECO: TECO-CXL (using CXL without DBA), and TECO-Reduction (using both CXL and DBA). For GCNII, we do not change the batch size, because GCNII only supports full-graph training. We cannot evaluate T5-large with ZeRO-Offload when the batch size is 16, because it leads to an out-of-memory error. Figures 11 show the training time speedup of TECO-Reduction over Zero-Offload.

For ZeRO-Offload in Figure 11 and Table IV, we have three observations. **(1)** TECO-Reduction outperforms ZeRO-Offload by 1.08x-1.82x. **(2)** Albert-xxlarge-v1 shows less speedup than the other models. That is because Albert has 4x more attention heads than GPT-2, Bert-large-cased, and T5-large, hence the computation (forward and backward) takes a larger portion of the total training time. Thus, there are fewer opportunities for TECO to take effects. **(3)** For all the models, TECO-Reduction consistently outperforms TECO-CXL by up to 21% because of DBA. Table IV has the results for TECO-reduction.

Impacts on training accuracy and convergence. We study the impact of DBA on training accuracy with ZeRO-Offload. Table V shows the final training accuracy (using the model specific metrics). There is no TECO-CXL's result, because it does not use DBA and has no impact on training accuracy. In general, we see small impact on training accuracy. Figure 10 shows the training loss curves. Without and with TECO-Reduction, the training loss curves show the similar trend and we use the same number of steps to reach convergence. The impact on the convergence is minor. Figure 10 only shows GPT-2 and Albert because of space limitation, but the above conclusion is valid for Bert and T5.

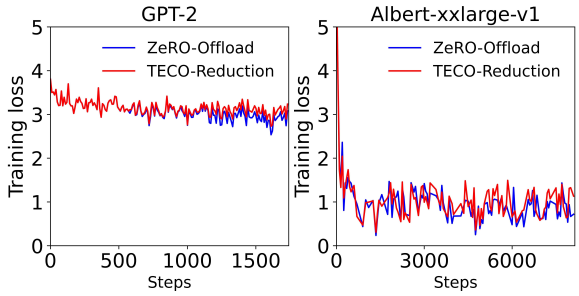


Fig. 10: The training loss curves.

Performance breakdown and analysis. To understand the performance benefit of TECO better, we break down the training time into forward-backward time, gradient transfer time exposed to the critical path, gradient optimizer, parameter optimization (using the ADAM optimizer), and parameter transfer time exposed to the critical path. See Figure 12.

For the gradients, the transfer time is completely hidden by TECO when the batch size is 8. When the batch size is smaller, the gradient transfer time is exposed to the critical path, even with TECO, but TECO hides it by at least 69%.

For the parameters, when the batch size is 4, TECO-CXL reduces transfer time by 76%. When applying DBA, the transfer time is completely hidden.

C. Communication Volume and DBA Contribution

Applying TECO-Reduction to parameters, the volume is reduced by 50% after applying DBA. For gradients, there is no reduction on the volume, because we do not apply DBA. Nevertheless, using CXL, TECO is able to hide the gradient transfer time. Reduction of communication volume by DBA leads to 0.8%-7.3% performance improvement (compared with the original time without TECO). It has been reported that in an AWS data center, the AI training takes 20% of GPU cycles [14]. Assume a data center with 256 A100 GPU and 50% utilization of GPUs. 7% of saving in training time leads to a reduction of roughly $900K in production cost in a year. (The cost estimation is based on AWS p4de.24xlarge instance [96]).

D. Overhead Analysis

We evaluate area, power, and latency overhead of the Aggregator and Disaggregator using Xilinx Vivado ML Design Suite [116] and Ramulator [40]. We implement the required logic on Xilinx UltraScale architecture (KU035-FFVA1156) with 406K FFs and 203K LUTs in 20 nm technology [115]. The FPGA-to-ASIC area, power and delay conversion ratios are 1:33, 1:14, and 1:3.5, respectively [42]. The latency is for processing a 64-byte cache line. The scaled power overhead of the Aggregator and Disaggregator is 0.0127W and 0.017W respectively. The latency of them are 1.28 ns and 1.126 ns.

The Aggregator and Disaggregator have insignificant performance overhead. When considering the limited bandwidth of CXL interfaces, each cache line takes around 4 ns latency and hence the added latency is amortized through pipelined

TABLE III: DL models configurations and training datasets.

Models	Model type	# Parameters	# Layers, Hidden size, # Attention head (if transformers)	# Dataset	Tasks	Metrics	Giant cache size
GPT-2 [76]	Transformer (decoder)	122M	12, 1024, 12	Wikitext	Lanuage modeling	Perplexity	324MB
Albert-xxlarge-v1 [43]	Transformer (encoder)	223M	12, 4096, 48	Squad-v2	Question-awsering	F1/EM	547MB
Bert-large-cased [2]	Transformer (encoder)	334M	24,1024,12	IMDB	Text Classification	Accuracy	817MB
T5-large [77]	Transformer (encoder-decoder)	737M	48,1024,12	Wiki-summary	Summarization	Gen-length	2069MB
GCNII [15]	Graph neural network	156M	64,1560, N/A	Wisconsin	Link prediction	Accuracy	400MB

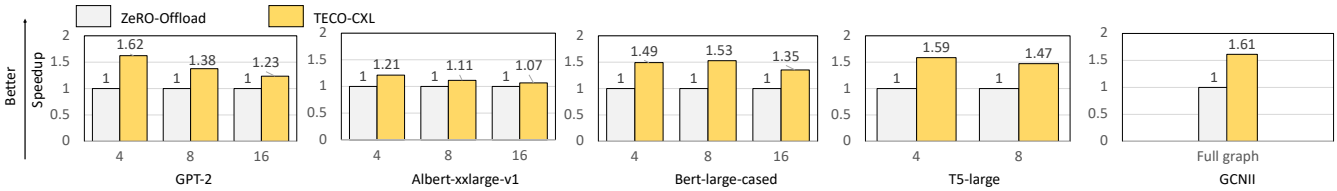


Fig. 11: Training time speedup of TECO-CXL over ZeRO-Offload. The x axis is batch sizes.

TABLE IV: TECO-reduction over ZeRO-Offload.

Batch-size	4	8	16
GPT2	1.82x	1.52x	1.32x
Albert-xxlargev1	1.25x	1.23x	1.08x
Bert-large-cased	1.6x	1.62x	1.41x
T5-large	1.73x	1.58x	N/A

TABLE V: The final model accuracy.

Models	Metrics	Results	
		Original	TECO-Reduction
GPT-2	Perplexity	21.05	21.54
Albert-xxlarge-v1	F1/EM	84.38/81.40	83.69/79.87
Bert-large-cased	Accuracy	93.13	91.99
T5-large	Gen-length	22.95	21.11
GCNII	Accuracy	54.90	N/A

TABLE VI: Impact of model size on TECO effectiveness.

Models	ZeRO-Offload	TECO-CXL	TECO-Reduction
GPT2	1x	1.55x	1.82x
GPT2-Medium	1x	1.54x	1.64x
GPT2-Large	1x	1.67x	1.79x
GPT2-11B	1x	1.29x	1.41x

transfers, such as processing cache lines while transferring earlier cache lines. However, to understand the performance overhead, we add 1 ns overhead for end-to-end performance evaluation. For the Disaggregator, we need to read the target

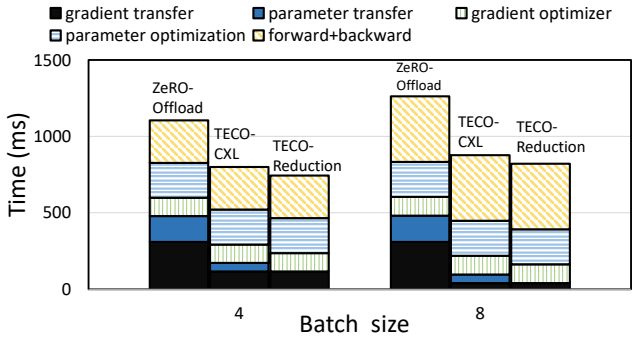


Fig. 12: Time breakdown. We use T5-large.

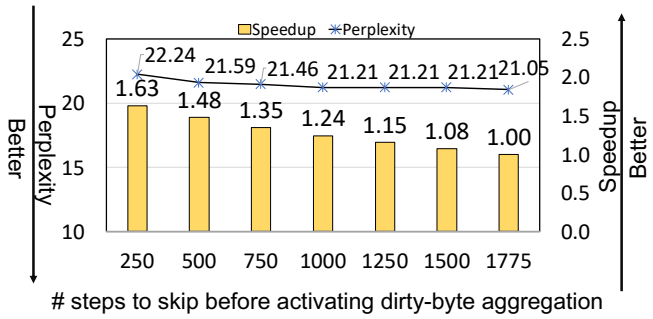


Fig. 13: Impact of DBA on DL model accuracy at different training steps. We use GPT-2.

cache line from DRAM and then the merged cache line should be written to DRAM. We simulate these extra read operations with Ramulator on the memory access traces extracted from our evaluation. The total simulated DRAM cycle increases by 2.48× and 1.9× for sequential and shuffled accesses, respectively. Considering the bandwidth gap between GDDR5 (total 900GB/s with 8 memory controllers) and PCIe 3.0 (16 GB/s) that NVIDIA V100 GPU has, this latency does not incur perceivable performance overhead.

E. Sensitivity Study

We use GPT-2, because it has multiple model scales provided by OpenAI [9] (GPT2-medium with 356M parameters and GPT2-large with 778M parameters). We continue to increase the model size to billion-scale(11 billion parameters) by changing the GPT-2 configurations.

Model size and model configuration. Table VI shows the performance of the original ZeRO-Offload, ZeRO-Offload with TECO-CXL and TECO-Reduction. As the model size is changed, TECO consistently brings performance benefits (44.9%, 38.9%, 44.2% and 29% for GPT-2, medium, large, 11B). For GPT-2 with 11 billion parameters, the performance improvement is smaller compared to the other three models. This is because the computation time (including forward pass, backward pass, gradient optimization and parameter

TABLE VII: Training time. Zero-Quant compresses model parameters. The compression ratio is 75%.

System	Task	Model	Time(Hours))
Zero-Quant	GLUE-MNLI [100]	Bert-base-uncased	5.8
TECO-Reduction	GLUE-MNLI [100]	Bert-base-uncased	2.03

TABLE VIII: Performance impact of lossless compression

Models	GPT2	Albert-xxlarge-v1	Bert-large	T5-large
Compression ratio	5%	0%	0%	36%
Normalized training time	4.51	1.95	3.03	2.04

optimization) already accounts for 63.4% of the total time. The computation time cannot be optimized further by TECO.

When to activate DBA. Figure 13 shows model accuracy (using the model-specific metric "perplexity") when TECO skips different numbers of steps to start DBA. We train the model to converge and use the same number of training steps (1775 steps). Figure 13 shows that compared with no DBA where accuracy is 21.05, activating DBA differently changes the accuracy (22.50-21.21), while the performance speedup varies (1.63-1.15). Choosing the 500th step strikes a balance.

F. Comparison with Model Compression

The model compression saves memory.

Lossy compression. We evaluate ZeRO-Quant [60] (a state-of-the-art lossy compression) in DeepSpeed. ZeRO-Quant takes 2.87x longer training time than TECO, because it requires a teacher model (a full-precision model) during the quantized model training to ensure training accuracy. Such a teacher model introduces extra training latency. Some works [67], [81] apply lossy compression to the post-training models for reducing inference time. Although they reduce communication time, they significantly increase training time.

Lossless compression. We evaluate the performance of a lossless compression algorithm (LZ4) [53] when transferring the parameters. LZ4 is a high-performance and byte-oriented compression algorithm commonly used in recent work [50], [63], [79]. We use LZ4's multi-threaded version [58] for CPU and NVIDIA's LZ4 solution [64] for GPU. Table VIII shows the training time normalized to TECO-Reduction. We see that compression and decompression incur large performance overhead (at least $2\times$). With this large overhead, a replacement of DBA with the lossless compression in TECO is impractical. The compression ratio is also very low, demonstrating the ineffectiveness of reducing communication volume.

IX. Related Work

Large DL model training on HM. Existing efforts [4], [30], [31], [73], [74], [79], [85], [87], [90], [93], [103] study DL models on HM, but without cache coherent interconnect. Others works [26], [33], [45], [49], [55], [57], [84], [88], [89], [107]–[110], [112]–[114], [117], [119], explore the HM optimization with the context of CPU. Betty [118] and Sentinel [86] uses HM to enable large graph neural network training.

Cache coherence interconnect. The recent emergence of cache coherent interconnect attracts attentions recently [7], [11], [106], [122]. COARSE [106] is a distributed parameter synchronization scheme based on disaggregated memory and cache coherent interconnect for training distributed DL models. Kona [11] reduces dirty data amplification and improves network utilization and performance based on cache coherent interconnect between FPGA and CPU. ORCA [122] leverages cache coherent interconnect to provide efficient notification of communication requests to accelerators and allow accelerators to directly process requests received by NIC. Different from them, TECO leverages cache coherent interconnect to build a cache coherent domain between CPU and GPU memories. OpenCAPI [68], CCIX [18] and Gen-Z [22] are previously proposed cache coherent connection protocols. Recently, these protocol has been merged into the CXL specification. NVLink [66] can also support coherent interconnection, but it can only used by NVIDIA GPUs. CXL has an open specification enabling cache coherent communication across hosts and a variety of accelerators (including GPUs). With the open CXL specification, the users can explore more application-optimal solutions such as giant cache models.

Peer-to-Peer Direct Memory Access (P2P). Existing works [6], [8], [56] use P2P to allow GPU to access SSD and perform direct file I/O. TECO is different from them, because it focuses on CPU memory and uses finer-grained data transfer.

X. Conclusions

Training large DL models without expensive hardware is the key to make them approachable. Using HM with tensor offloading is promising. Its effectiveness relies on whether tensor transfers in HM are efficient. Extending an emerging cache-coherent interconnect CXL, we study how CXL can enable efficient tensor offloading and what changes in hardware and software should be provided. Extending CXL, we significantly reduce training time by 33.7% and communication overhead by 93.7%, compared with the state-of-the-art industry solution.

Acknowledgement

This work was partially supported by U.S. National Science Foundation (2104116 and 2316202) and the Chameleon Cloud. This paper is a result of a research project sponsored by SK hynix Inc. We would like to thank the anonymous reviewers for their feedback on the paper.

References

[1] LAMMPS Molecular Dynamics Simulator . http://lammps.sandia.gov/.

[2] Bert, roberta, distilbert, xlnet — which one to use? https://towardsdatascience.com/bert-roberta-distilbert-xlnet-which-one-to-use-3d5ab82ba5f8, 2019.

[3] AMD. Pcie transmit buffers. https://docs.xilinx.com/r/en-US/pg054-7series-pcie/Transmit-Buffers.

[4] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient Combination of Rematerialization and Offloading for Training DNNs. *Advances in Neural Information Processing Systems*, 34, 2021.

[5] Tal Ben-Nun and Torsten Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Computing Surveys*, 52(4), 2019.

[6] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. Spin: Seamless operating system integration of peer-to-peer dma between ssds and gpus. *ACM Transactions on Computer Systems (TOCS)*, 36(2):1–26, 2019.

[7] Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Tobias Vinçon, Andreas Koch, and Ilia Petrov. Cache-Coherent Shared Locking for Transactionally Consistent Updates in Near-Data Processing DBMS on Smart Storage. In *International Conference on Extending Database Technology*, 2022.

[8] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. {GAIA}: An {OS} page cache for heterogeneous systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 661–674, 2019.

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[10] Sébastien Bubeck and Mark Sellke. A Universal Law of Robustness via Isoperimetry. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[11] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[12] Yu Cao, Wei Bi, Meng Fang, and Dacheng Tao. Pretrained Language Models for Dialogue Generation with Multiple Input Sources. *CoRR*, abs/2010.07576, 2020.

[13] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. Fault Resilience of the Algebraic Multi-grid Solver. In *Proceedings of the international conference on Supercomputing (ICS)*, 2012.

[14] Bryan Catanzaro. Language Models: the Most Important Compute Challenge of Our Time. Keynote at Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2023.

[15] Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and deep graph convolutional networks. *CoRR*, abs/2007.02133, 2020.

[16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[17] Hyunghun Cho, Yongjin Kim, Eunjung Lee, Daeyoung Choi, Yongjae Lee, and Wonjong Rhee. Basic enhancement strategies when using bayesian optimization for hyperparameter tuning of deep neural networks. *IEEE Access*, 8:52588–52608, 2020.

[18] CCIX Consortium. CCIX. https://www.ccixconsortium.com/".

[19] CXL Consortium. Chapter 8: Control and Status Registers. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf.

[20] CXL Consortium. ComputeExpressLink. https://www.computeexpresslink.org.

[21] CXL Consortium. Giant cache model. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf.

[22] Gen-Z Consortium. The Gen-Z Consortium. https://genzconsortium.org.

[23] CXL Consortium. Compute express link™ (cxl™): Exploring coherent memory and innovative use cases. https://www.youtube.com/watchv=It1_mHsor9g&ab_channel=CXLConsortium.

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

[25] Renhao Fan, Yikai Cui, Qilin Chen, Mingyu Wang, Youhui Zhang, Weimin Zheng, and Zhaolin Li. Maicc: A lightweight many-core architecture with in-cache computing for multi-dnn parallel inference. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 411–423, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Yuan Feng, Hyeran Jeon, Filip Blagojevic, Cyril Guyot, Qing Li, and Dong Li. Attmemo : Accelerating transformers with memoization on big memory systems, 2023.

[27] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. Spade: A flexible and scalable accelerator for spmm and sddmm. ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Yu Gong, Miao Yin, Lingyi Huang, Jinqi Xiao, Yang Sui, Chunhua Deng, and Bo Yuan. Ette: Efficient tensor-train-based computing engine for deep neural networks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[31] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing

Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[32] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism*. 2019.

[33] Yingchao Huang and Dong Li. Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems. In *IEEE International Conference on Cluster Computing*, 2017.

[34] Norm Jouppi. Google supercharges machine learning tasks with tpu custom chip. *Google Blog, May*, 18(1), 2016.

[35] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *CoRR*, abs/2001.08361, 2020.

[36] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[37] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486. IEEE, 2020.

[38] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. NPE: an fpga-based overlay processor for natural language processing. *CoRR*, abs/2104.06535, 2021.

[39] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. *ACM SIGARCH Computer Architecture News*, 44(3):380–392, 2016.

[40] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. 15(1):45–49, 2016.

[41] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming Catastrophic Forgetting in Neural Networks. In *Proceedings of the National Academy of Sciences*, 2017.

[42] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. 26(2):203–215, 2007.

[43] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. *CoRR*, abs/1909.11942, 2019.

[44] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. 2022.

[45] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Cedric Lichtenau, Alper Buyuktosunoglu, Ramon Bertran, Peter Figuli, Christian Jacobi, Nikolaos Papandreou, Haris Pozidis, Anthony Saporito, Andrew Sica, and Elpida Tzortzatos. Ai accelerator on ibm telum processor: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1012–1028, New York, NY, USA, 2022. Association for Computing Machinery.

[47] Jiahuang Lin, Xin Li, and Gennady Pekhimenko. Multi-node bert-pretraining: Cost-efficient approach. *CoRR*, abs/2008.00177, 2020.

[48] Linkedin. Nvidia plans custom chips: Report. https://www.linkedin.com/news/story/nvidia-plans-custom-chips-report-5935428/.

[49] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. Sparta: high-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 318–333, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Weiqiang Liu, Faqiang Mei, Chenghua Wang, Maire O'Neill, and Earl E Swartzlander. Data compression device based on modified lz4 algorithm. *IEEE Transactions on Consumer Electronics*, 64(1):110–117, 2018.

[51] Zejian Liu, Gang Li, and Jian Cheng. Hardware acceleration of fully quantized bert for efficient natural language processing. In *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 513–516, 2021.

[52] Yun-Chen Lo and Ren-Shuo Liu. Bucket getter: A bucket-based processing engine for low-bit block floating point (bfp) dnns. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 1002–1015, New York, NY, USA, 2023. Association for Computing Machinery.

[53] LZ4. Lz4. https://lz4.github.io/lz4/.

[54] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[55] Suyash Mahar, Hao Wang, Wei Shu, and Abhishek Dhanotia. Workload behavior driven memory subsystem design for hyperscale, 2023. https://arxiv.org/abs/2303.08396.

[56] Pak Markthub, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Satoshi Matsuoka. Dragon: breaking gpu memory capacity limits with direct nvm access. In *SC18: International Conference for High*

Performance Computing, Networking, Storage and Analysis, pages 414–426. IEEE, 2018.

[57] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[58] Takayuki Matsuoka. Multi-threading LZ4. https://github.com/t-mat/lz4mt.git".

[59] Meta. Introduction to torch.compile. https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html.

[60] Microsoft. DeepSpeed Compression: A Composable Library for Extreme Compression and Zero-Cost Quantization. https://www.microsoft.com/en-us/research/blog/deepspeed-compression-a-composable-library-for-extreme-compression-and-zero-cost-quantization/.

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

[62] Microsoft. DeepSpeed. https://github.com/microsoft/DeepSpeed, 2021.

[63] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[64] NVIDIA. nvCOMP. https://github.com/NVIDIA/nvcomp.git.

[65] NVIDIA. Nvidia dgx b200 specifications. https://www.nvidia.com/en-us/data-center/dgx-b200/.

[66] NVIDIA. Nvlink and nvlink switch. https://www.nvidia.com/en-us/data-center/nvlink/.

[67] Anton Obukhov, Maxim Rakhuba, Stamatios Georgoulis, Menelaos Kanakis, Dengxin Dai, and Luc Van Gool. T-basis: a compact representation for neural networks. In *International Conference on Machine Learning*, pages 7392–7404. PMLR, 2020.

[68] OpenCAPI. Ecosystem for developers using opencapi, a new high performance coherent bus interface. https://github.com/OpenCAPI.

[69] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. Demystifying BERT: implications for accelerator design. *CoRR*, abs/2104.08335, 2021.

[70] PCI. osdev.org. . Bar address registers. https://en.wikipedia.org/wiki/PCI_configuration_space.

[71] PCI-SIG. PCI Express Base Specification Revision 3.1a. https://pcisig.com/specifications.

[72] PCI-SIG. PCI Express Base Specification Revision 5.0. hhttps://pcisig.com/specifications.

[73] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

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

[75] Yubin Qin, Yang Wang, Dazheng Deng, Zhiren Zhao, Xiaolong Yang, Leibo Liu, Shaojun Wei, Yang Hu, and Shouyi Yin. Fact: Ffn-attention co-optimized transformer architecture with eager correlation prediction. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[76] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[77] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *CoRR*, abs/1910.10683, 2019.

[78] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations Toward Training Trillion Parameter Models. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[79] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *CoRR*, abs/2104.07857, 2021.

[80] Bahador Rashidi, Chao Gao, Shan Lu, Zhisheng Wang, Chunhua Zhou, Di Niu, and Fengyu Sun. Unico: Unified hardware software co-optimization for robust neural network acceleration. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 77–90, New York, NY, USA, 2023. Association for Computing Machinery.

[81] Brandon Reagan, Udit Gupta, Bob Adolf, Michael Mitzenmacher, Alexander Rush, Gu-Yeon Wei, and David Brooks. Weightless: Lossy weight encoding for deep neural network compression. In *International Conference on Machine Learning*, pages 4324–4333. PMLR, 2018.

[82] Reddit. The cost of training gpt-3. https://www.reddit.com/r/MachineLearning/comments/hwfjej/d_the_cost_of_training_gpt3/.

[83] Enrico Reggiani, Cristóbal Ramírez Lazo, Roger Figueras Bagué, Adrián Cristal, Mauro Olivieri, and Osman Sabri Unsal. Bison-e: A lightweight and high-performance accelerator for narrow integer linear algebra computing on the edge. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 56–69, New York, NY, USA, 2022. Association for Computing Machinery.

[84] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. Optimizing large-scale plasma simulations on persistent memory-based heterogeneous

memory with effective data placement across memory hierarchy. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 203–214, New York, NY, USA, 2021. Association for Computing Machinery.

[85] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[86] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 598–611. IEEE, 2021.

[87] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *USENIX Annual Technical Conference*, 2021.

[88] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. MTM: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 803–817, New York, NY, USA, 2024. Association for Computing Machinery.

[89] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[90] 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 *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[91] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution. In *ACM International Conference on Supercomputing (ICS)*, 2012.

[92] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR*, abs/1909.08053, 2019.

[93] SB Shriram, Anshuj Garg, and Purushottam Kulkarni. Dynamic memory management for gpu-based training of deep neural networks. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 200–209. IEEE, 2019.

[94] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms, 2012.

[95] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. *arXiv preprint arXiv:2312.12456*, 2023.

[96] vantage. https://instances.vantage.sh/aws/ec2/p4d.24xlarge.

[97] Swagath Venkataramani, Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2013.

[98] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, et al. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 13–26, 2017.

[99] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, Ankur Agrawal, Monodeep Kar, Shubham Jain, Alberto Mannari, Hoang Tran, Yulong Li, Eri Ogawa, Kazuaki Ishizaki, Hiroshi Inoue, Marcel Schaal, Mauricio Serrano, Jungwook Choi, Xiao Sun, Naigang Wang, Chia-Yu Chen, Allison Allain, James Bonano, Nianzheng Cao, Robert Casatuta, Matthew Cohen, Bruce Fleischer, Michael Guillorn, Howard Haynie, Jinwook Jung, Mingu Kang, Kyu-hyoun Kim, Siyu Koswatta, Saekyu Lee, Martin Lutz, Silvia Mueller, Jinwook Oh, Ashish Ranjan, Zhibin Ren, Scot Rider, Kerstin Schelm, Michael Scheuermann, Joel Silberman, Jie Yang, Vidhi Zalani, Xin Zhang, Ching Zhou, Matt Ziegler, Vinay Shah, Moriyoshi Ohara, Pong-Fei Lu, Brian Curran, Sunil Shukla, Leland Chang, and Kailash Gopalakrishnan. Rapid: Ai accelerator for ultra-low precision training and inference. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 153–166. IEEE Press, 2021.

[100] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[101] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[102] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. Zero++: Extremely efficient collective communication for giant model training. *arXiv preprint arXiv:2306.10209*, 2023.

[103] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018.

[104] Yu Wang, Gu-Yeon Wei, and David Brooks. A systematic methodology for analysis of deep learning hardware and software platforms. *Proceedings of Machine Learning and Systems*, 2:30–43, 2020.

[105] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. Stream Floating: Enabling Proactive and Decentralized Cache Optimizations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 640–653. IEEE,

2021.

[106] Zixuan Wang, Joonseop Sim, Euicheol Lim, and Jishen Zhao. Enabling efficient large-scale deep learning training with cache coherent disaggregated memory systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2022.

[107] K. Wu, Y. Huang, and D. Li. Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[108] Kai Wu, Jie Ren Ivy Peng, and Dong Li. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *USENIX Conference on File and Storage Technologies*, 2021.

[109] Kai Wu, Jie Ren, and Dong Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 401–413, 2018.

[110] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey Vetter, and Sparsh Mittal. Algorithm-directed data placement in explicitly managed non-volatile memory. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[111] Yannan Nellie Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel Emer. Highlight: Efficient and flexible dnn acceleration with hierarchical structured sparsity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 1106–1120, New York, NY, USA, 2023. Association for Computing Machinery.

[112] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-Exclusive memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, Santa Clara, CA, July 2024. USENIX Association.

[113] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. Md-hm: memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 215–226, New York, NY, USA, 2021. Association for Computing Machinery.

[114] Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. Merchandiser: Data placement on heterogeneous memory for task-parallel hpc applications with load-balance awareness. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 204–217, New York, NY, USA, 2023. Association for Computing Machinery.

[115] Xilinx. UltraScale Architecture and Product Data Sheet: Overview. https://docs.xilinx.com/v/u/en-US/ds890-ultrascale-overview".

[116] Xilinx. Vivado ML Overview. https://www.xilinx.com/products/design-tools/vivado.html".

[117] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. FlexMem: Adaptive page profiling and migration for tiered memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 817–833, Santa Clara, CA, July 2024. USENIX Association.

[118] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. Betty: Enabling large-scale gnn training with batch-level graph partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 103–117, New York, NY, USA, 2023. Association for Computing Machinery.

[119] Shuo Yang, Kai Wu, Yifan Qiao, Dong Li, and Jidong Zhai. Algorithm-Directed Crash Consistence in Non-Volatile Memory for HPC. In *IEEE International Conference on Cluster Computing*, 2017.

[120] 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.

[121] 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. *arXiv preprint arXiv:1904.00962*, 12:2, 2019.

[122] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan RK Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. Orca: A network and architecture co-design for offloading us-scale datacenter applications. *arXiv preprint arXiv:2203.08906*, 2022.

Appendix: Artifact Description

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 We characterize tensor transfers during large model training with an industry-quality, HM-based training solution (ZeRO-Offload from DeepSpeed), and identify two problems that make tensor transfers the major performance bottleneck for the HM-based training;

C_2 Evaluating with large transformers and a graphic neural network, we show that compared with ZeRO-Offload, TECO reduces training time by 33.7% without changing model convergence and accuracy; TECO reduces communication overhead by 93.7% on average

B. Computational Artifacts

A_1 https://github.com/luckyq/ADSC-24

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Tables I,IV,V,VI
	C_2	Figures 2,10, 11, 12, 13

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact includes the scripts to get the communication overhead in ZeRO-Offload, the value changes between continuous iterations and the experiments in evaluation. Through these tests, we can prove that 1): communication still takes a large portion of the total training time in previous offloading system; 2) TECO can outperform previous systems by extending CXL. TECO can reduce most of the communication overhead.

Expected Results

The communication overhead takes a rather large portion of total training time. Across two consecutive training steps, among those parameters that change values, most of them change values only in the last two bytes. TECO can hide the majority of the communication time. By this, TECO can TECO reduces training time by 33.7% without changing model convergence and accuracy; TECO reduces communication overhead by 93.7% on average.

Expected Reproduction Time (in Minutes)

The expected computational time of this artifact is 10 hours.

Artifact Setup (incl. Inputs)

Hardware: Intel Xeon 6120 CPU, two sockets, each socket with 186GB DRAM. One Tesla V100 GPU with 32GB memory. The link between the GPU and CPU is PCIe3.0x16 lanes.

Software:

- PyTorch, 1.10.0, https://pytorch.org/get-started/previous-versions/
- DeepSpeed, 0.3.15,https://github.com/microsoft/DeepSpeed
- DeepSpeedExamples, latest version, https://github.com/microsoft/DeepSpeedExamples.git
- Transformer, latest version,https://huggingface.co/docs/transformers/en/index
- Datasets, latest version, https://huggingface.co/docs/datasets/en/index
- Gem5-AVX, latest version, https://github.com/seanzw/gem5-avx
- Accel-Sim, latest version, https://accel-sim.github.io/
- LZ4, laster version, https://github.com/lz4/lz4.git
- nvCOMP, latest version, https://github.com/NVIDIA/nvcomp.git

Datasets / Inputs:

- IMDB, https://huggingface.co/datasets/stanfordnlp/imdb
- Wikitext, https://huggingface.co/datasets/wikitext
- Wiki-summary, https://github.com/m3hrdadfi/wiki-summary
- Squad-v2, https://huggingface.co/datasets/rajpurkar/squad_v2
- Wisconsin, https://paperswithcode.com/dataset/wisconsin-48-32-20-fixed-splits

Installation and Deployment: All the packages(excluding Gem5−AVX and Accel−sim) can be installed by pip install package_name==version. For Gem5−AVX and Accel−sim, we can use git to download it. For the installation, we can just follow the instructions in READ.ME of each repository.

Artifact Execution

For the motivation test, it contains two Python scripts, *communication.py* and *valuechanges.py*. *communication.py* and *valuechanges.py* are independent. The first one is used to get the communication time exposed to the critical path. The second one is used to get the Variance of parameter/gradient values across training steps. The dataset used in these two scripts is IMDB. There is no specific command-line parameter to run the scripts. For the speedup tests, four scripts are used for each model. *model_name.sh* run the model with ZeRO-Offload. *model_name_gem5_avx.sh* is to simulate the parameter updating on the CPU and get the memory access trace. *process.py* is going to replay the trace and get the time of CXL data transfer. *gpu_backward.sh* is going to simulate the data transfer during the backward phase. For the accuracy and loss tests, we will use the scripts in *transformers* package (path_to_transformers/examples/pytorch/). To get results of apply compression method in the training, we will run the LZ4 and nvCOMP to compress and decompress the same amount of parameters.

Artifact Analysis (incl. Outputs)

The expected results are detailed breakdown time of each phase. A simple sum up to get the end-to-end performance. To get the performance of TECO, the simulated results can replace the corresponding part. For accuracy, it can directly be read. The expected training time is reduced by 33.7% on average. Communication overhead will be reduced 93.7% on average. These results will prove that TECO is effective to reduce the communication overhead in tensor-offloading Deep learning systems.