Smart-Infinity: Fast Large Language Model Training using Near-Storage Processing on a Real System

Hongsun Jang[†], Jaeyong Song[†], Jaewon Jung[†], Jaeyoung Park[‡,*], Youngsok Kim[§], Jinho Lee[†,¶]
[†]*Department of Electrical and Computer Engineering, Seoul National University*
[‡]*Department of Electrical and Computer Engineering, University of Texas at Austin*
[§]*Department of Computer Science, Yonsei University*
{hongsun.jang, jaeyong.song, jungjaewon}@snu.ac.kr, jaeyoung@utexas.edu, youngsok@yonsei.ac.kr, leejinho@snu.ac.kr

***Abstract*—The recent huge advance of Large Language Models (LLMs) is mainly driven by the increase in the number of parameters. This has led to substantial memory capacity requirements, necessitating the use of dozens of GPUs just to meet the capacity. One popular solution to this is storage-offloaded training, which uses host memory and storage as an extended memory hierarchy. However, this obviously comes at the cost of storage bandwidth bottleneck because storage devices have orders of magnitude lower bandwidth compared to that of GPU device memories.**

Our work, Smart-Infinity, addresses the storage bandwidth bottleneck of storage-offloaded LLM training using near-storage processing devices on a real system. The main component of Smart-Infinity is SmartUpdate, which performs parameter updates on custom near-storage accelerators. We identify that moving parameter updates to the storage side removes most of the storage traffic. In addition, we propose an efficient data transfer handler structure to address the system integration issues for Smart-Infinity. The handler allows overlapping data transfers with fixed memory consumption by reusing the device buffer. Lastly, we propose accelerator-assisted gradient compression/decompression to enhance the scalability of Smart-Infinity. When scaling to multiple near-storage processing devices, the write traffic on the shared channel becomes the bottleneck. To alleviate this, we compress the gradients on the GPU and decompress them on the accelerators. It provides further acceleration from reduced traffic. As a result, Smart-Infinity achieves a significant speedup compared to the baseline. Notably, Smart-Infinity is a ready-to-use approach that is fully integrated into PyTorch on a real system. The implementation of Smart-Infinity is available at https://github.com/AIS-SNU/smart-infinity.

I. Introduction

Transformer [118] based models currently dominate the natural language processing (NLP) field, effectively addressing the gradient vanishing problem that plagued recurrent neural networks (RNNs). The transformer-based models tend to have large training parameters, still not showing overfitting problems to the training corpus [85]. Therefore, the recent transformer-based model [17], [26], [85], [95] has been linearly scaling the model size for the past few years, forming a group of large language models (LLMs). As the models become larger, the major limiting factor becomes the GPU memory capacity. Often, dozens of GPUs are required, just to keep the training data on the memory, even when such computing power is not necessary such as fine-tuning [120].

One promising way to handle such a problem in a GPU memory-limited environment is storage-offloaded training [97], [113]. For LLM training, it is known that the most memory-consuming factor is optimizer states and gradients [62], [92], [97], [99], [113], followed by model parameters and the activations. Using this fact, the most widely adopted form of storage-offloaded training [99] is to store optimizer states and gradients in storage, while activations and model parameters are stored in host memory. Because the memory capacities typically differ by orders of magnitude (e.g., ∼80G, ∼2TB, ∼100TB for GPU memory, host memory, and storage devices, respectively), they essentially serve as an extended memory hierarchy for GPUs.

Predictably, the use of large-capacity storage devices causes a severe bandwidth bottleneck because SSD storage devices only provide up to a few GB/s of bandwidth. According to our study, more than 88% of the total training time is consumed by transferring data from/to the storage. At a glance, combining multiple storage devices using RAID0 solution could alleviate the problem. However, such a method would fundamentally be bottlenecked by the limited number of PCIe lanes (or IO pins) of the host processor. Moreover, PCIe lanes from CPUs nowadays are very valuable resources, which are shared by GPUs, FPGAs, NICs, or even system memories [30].

In such circumstances, we aim to address these issues using computational storage device (CSD) products. Based on decades of research [24], [35], [56], [74], [117], there are several commercial CSD products that are available off-the-shelf [1], [6], [7], [83]. By placing computational engines near storage, they aim to offload computation of the host, and utilize the internal bandwidth of storage. When more such devices are added, the available internal bandwidths linearly increase.

Utilizing these, we propose Smart-Infinity, a fast LLM training system using CSDs. To address the storage bottleneck problem, we mainly suggest moving the update task to the near-storage accelerator. We identify that the optimizer states are only used in the update phase of the training but consume 75% of the total storage bandwidth. By moving the update task to the custom accelerator inside CSDs, only the gradients and model parameters are required to be transferred, reducing 75% of the entire traffic.

There are several challenges to realizing Smart-Infinity on an actual system. One issue is the CSD-internal data transfer,

[*]Work performed while at Yonsei University. [¶]Corresponding author.

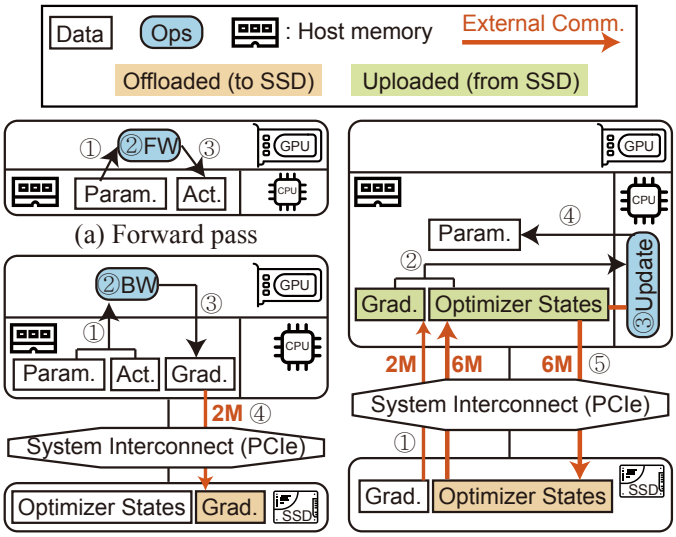


Fig. 1: A conceptual diagram of the storage-offloaded LLM training. Overview of (a) the forward pass, (b) the backward pass, and (c) the update (step) procedure.

which causes detrimental effects on the system performance. To address this, we propose internal transfer handler optimization with buffer pre-allocation and swap overlapping techniques for drawing maximum throughput.

Additionally, we propose a CSD-assisted gradient compression/decompression. When we scale the system with multiple CSDs, the bottleneck is at the host to storage traffic through the shared channel for sending gradients. To reduce this traffic, we compress the gradients on GPUs, and decompress them on the accelerator in CSD. This greatly reduces the traffic toward better performance, without compromising the final model accuracy.

Most notably, Smart-Infinity is integrated into PyTorch on a real system and is a ready-to-use framework. Our evaluation shows that Smart-Infinity achieves up to 2.11× speedup over the baseline. The contributions are summarized as follows.

1) We propose Smart-Infinity, a method to perform the update phase of LLM training in custom CSD accelerators. This greatly reduces the storage bandwidth bottleneck.
2) We propose an efficient data transfer handler structure to utilize storage bandwidth and hide the latency of CSDs.
3) We suggest a CSD-assisted gradient compression method to enhance the scalability of Smart-Infinity.
4) We integrate Smart-Infinity on a real system with PyTorch to achieve up to 2.11× speedup on mixed-precision LLM training.

II. Background

As discussed in Section I, using aggregated GPU memory for entire LLM training is an expensive way. A popular alternative is offloading solutions [97], [99], [113] utilizing the host memory or the storages in training LLMs with limited resources. We will describe them and discuss their limitations.

A. Overview of Dataflow in Storage-Offloaded LLM Training

ZeRO [96] analyzes the memory usage while LLM training and suggests methods to split the optimizer states (e.g., momentum and variance of Adam) for minimizing memory usage in distributed LLM training with multiple GPUs. It points out the considerable additional memory requirements for optimizer states during LLM training.

The FP32 optimizer states occupy $6M$ capacity (model parameter, momentum, and variance in FP32) under mixed precision training [84], regarding the FP16 model parameter size as M. Considering that maintaining optimizer states in FP32 is an almost essential option for mixed precision LLM training [44], [85], [88], this provides an insight that management of the optimizer state is a critical issue for LLM training in a resource-limited environment (e.g., lack of enough GPU memory).

Motivated by the above insight, many approaches [92], [97], [99], [113] try to offload optimizer states of LLM training to host memory or storages, when GPU cannot hold the whole optimizer states due to memory capacity. Host memory-offloaded training [92], [99] provides the baseline concept of training LLMs using the host memory.

Because the computational intensity of updating optimizer states (element-wise operations) is low, it becomes attractive to offload the optimizer states to host memory and let the host CPU update parameters. However, to train larger LLMs with model sizes of 345M [94], [108] to 530B [17], [85], which cannot be trained even with the host memory-offloaded training, some works [62], [97] provide solutions to further utilize storage devices.

On top of the host memory-offloaded training, storage-offloaded training [97] suggests additionally using storage (usually NVMe) devices for training LLMs with limited hardware resources. They offload the optimizer states to storages and make host memory hold only activations and mixed precision parameters. It breaks the wall of the host memory size so that it can train even larger models on a single machine.

Figure 1 provides an overview of storage-offloaded training methods using mixed precision training. These methods split an LLM model into multiple blocks (e.g., layers), which have sizes that a GPU (or host memory) can handle at a time. Before the training starts, the whole optimizer states for the model are initially stored in the storage (i.e., SSDs), as depicted at the bottom of Figure 1(b) and (c). In the forward pass (Figure 1(a)), ① GPU loads the mixed precision parameters of a block. ② After forward processing of the block in GPU, ③ GPU checkpoints the activation of block to host memory. For all blocks, the host iteratively conducts ①-③ and checkpoints all activations of the whole model. In the backward pass (Figure 1(b)), for each block, ① GPU loads mixed precision parameters and activations, and ② conducts backward processing of the block in GPU, ③ whose resulting gradients are sent to the host memory. To minimize the usage of host memory space, ④ the host offloads the created gradients to the storage device.

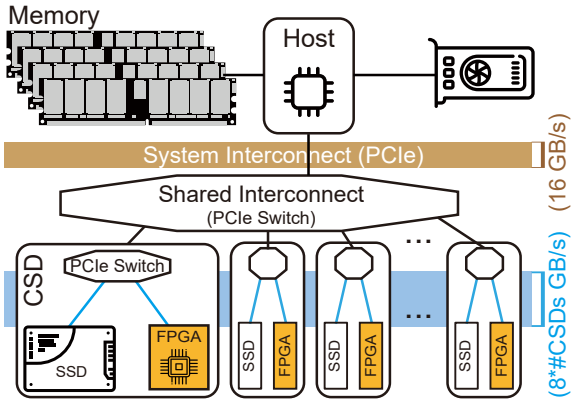
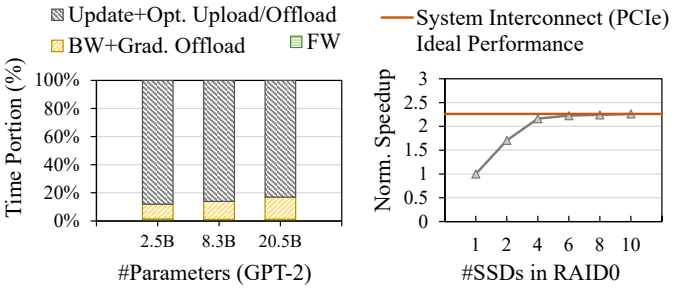


Fig. 2: An example environment with CSDs (e.g., SmartSSDs).



(a) Breakdown of baseline (b) RAID0 performance

Fig. 3: (a) LLM storage-offloaded training time breakdown with various model sizes. (b) Speedup from the increasing numbers of SSDs using RAID0 solution.

When the forward and backward passes are finished, the host starts the update procedure for the parameters of a model (Figure 1(c)) in a block-wise manner. ① The gradients and optimizer states are uploaded from the storage to the host memory. Then, ② The CPU loads the uploaded gradients and optimizer states, and ③ updates them. After the block update, ④ the CPU replaces the mixed precision parameters with updated ones and ⑤ offloads the optimizer states to the NVMe device. The host keeps updating all blocks by repeatedly conducting ①-⑤, and a forward pass of the next iteration follows after updating is finished.

From the help of storage devices, prior works enable the training of extreme-size LLMs when there is not enough GPU memory to hold the whole optimizer states. However, as illustrated in Figure 1(b) and (c), storage-offloaded training suffers from a significant amount of upload/offload data transfers at every iteration, including the optimizer states (model parameter, momentum, and variance of $6M$ size in total) and gradients handled in 32 bits by using a highly optimized offloading engine of [97] ($2M$). Such data transfers pass through the system interconnect (PCIe) depicted with the red arrows in Figure 1. Smart-Infinity targets to reduce the data transfer through the system interconnect with a computational storage device (CSD). Smart-Infinity enjoys the aggregated internal CSD bandwidth instead of limited system interconnect bandwidth. To retain such bandwidth with multiple CSDs, Smart-Infinity utilizes the accelerators in CSDs while addressing some real system challenges towards efficient training.

B. Computational Storage Devices

Computational storage devices, or near-storage processing (NSP) has been studied for years [24], [35], [56], [117]. By placing a computational engine closer to the storage devices, latency and bandwidth benefits can be obtained, in addition to reducing the host workload. Among many CSD types [112], we specifically target ones shown in Figure 2 where a lightweight FPGA accelerator is connected to SSDs via a switch inside the product, which is widely used for commercial products [1], [7], [18], [83]. Figure 2 provides an example of a host environment with multiple such CSDs.

Generally, CSDs have two unique features compared to plain storage. First, it has an accelerator (e.g., a lightweight FPGA) to compute data near storage. Second, it has its own internal PCIe switch, providing a private inner path for direct peer-to-peer (P2P) communication between the SSD and accelerator without redundant storage-to-host and host-to-storage data traffic through the system interconnect. A single CSD provides no bandwidth boost, as storage-to-FPGA and storage-to-host traffic transfers pass through the same number of PCIe lanes. However, when multiple CSD devices are on the system (Figure 2), the aggregated internal bandwidth linearly increases according to the number of CSDs, while the shared system interconnect bandwidth remains the same. In Smart-Infinity, we use SmartSSD [83], a representative commercially available CSD on the market.

III. Motivation

We discussed the data transfer bottleneck between host memory and storage devices in storage-offloaded LLM training. In this section, we directly analyze such an overhead with actual data and figure out why it is hard to be mitigated.

Data transfer overhead. Figure 3(a) is the training time breakdown of LLM training with ZeRO-Infinity [97], state-of-the-art storage-offloaded training framework. We conduct training in the environment with a single NVMe device (SSD). For the detailed experimental environment, please refer to Section VII-A. As we described in Section II, we break down the storage offloaded training into forward (FW), backward (BW + Gradients Offload), and update (Update + Optimizer states Upload/Offload). Contrary to conventional training, the most time is spent on the update phase of over 80% the training time, due to the storage access overhead. The data transfer portion is significant regardless of model sizes, so it is a critical issue to be addressed in storage-offloaded LLM training.

System interconnect bottleneck. One way to mitigate the data transfer overhead of NVMe devices is to make a higher communication bandwidth using RAIDs. Figure 3(b) shows the normalized speedup of storage-offloaded training when increasing the number of SSDs using RAID0. Unfortunately, the speedup saturates after using more than four SSDs. Shared

TABLE I
SYSTEM INTERCONNECT TRAFFIC FOR STORAGE-OFFLOADED TRAINING WITH ADAM OPTIMIZER.

Type	Optimizer States		Gradients	
SSD Operation	Read	Write	Read	Write
ZeRO-Inf [97]	$6M$	$6M$	$2M$	$2M$
SmartUpdate	$2M$	–	–	$2M$
SmartComp (c%)	$2M$	–	–	$c\% \times 2M$

system interconnect becomes a new bottleneck when using more than four SSDs. Therefore, using the RAID solution for storage-offloaded LLM training has a limitation.

The motivational study demonstrates that data transfer overhead in storage-offloaded LLM training cannot be easily mitigated due to the limited resource of the existing system structure. Therefore, when we use CSDs as an alternative, our primary goal is **minimizing the data transfer between storage devices and host memory through the shared system interconnect**, and we need to fully utilize 1) the aggregated internal bandwidth from multiple CSDs and 2) the computational ability of the accelerator in each CSD.

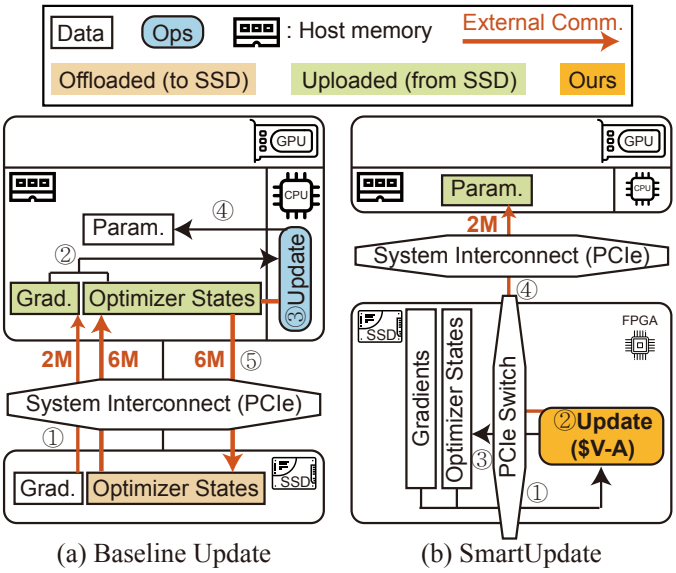
IV. SMART-INFINITY

Table I provides an overview of changes in the system interconnect traffic from applying Smart-Infinity. First, to minimize the communication between storage devices and host memory, we offload the update computation from the CPU to the accelerator in CSDs (SmartUpdate). From this, we can benefit from the fast aggregate bandwidth of CSDs, reducing the communication overhead of existing storage-offloaded LLM training methods from $(6+2)M$ to $2M$. Second, we propose a new internal data transfer handler structure with buffer preallocation and swap overlapping, which are directly applicable to SmartUpdate. The handler structure can be applied when integrating CSD to popular host codes (C++, Python), which is critical to the throughput of CSD applications in real systems. Third, we propose a new method to further reduce remaining storage write traffic through the shared system interconnect from $2M$ to $c\% \times 2M$ (c: compression ratio) by compressing the gradients using the computational capability of FPGA in CSDs (SmartComp).

A. SmartUpdate: Near-storage Update with CSD

As discussed in Section II, the communication volume of the optimizer states in the update procedure is significant in storage-offloaded LLM training. With conventional storage devices, this communication through the system interconnect (PCIe) is essential because the CPU conducts the update computation. However, SmartUpdate offloads the updating procedure from the CPU to the CSDs to reduce the interconnect communication volume. The total data read from the storage is the same, but the aggregate bandwidth of multiple CSDs is utilized instead of the shared system interconnect.

Figure 4 illustrates the detailed process of SmartUpdate compared to the baseline [97]. We use Adam optimizer [63] as an example because it is the primary choice for modern



(a) Baseline Update (b) SmartUpdate

Fig. 4: Update procedure of the storage-offloaded training with (a) baseline [97] and (b) SmartUpdate.

LLM training [17], [34], [94]. As illustrated in Figure 4(a), the baseline updates parameters using the host CPU. The gradients and optimizer states should be uploaded from SSDs to conduct the update procedure with the CPU. After the update is finished, the optimizer states are offloaded to SSDs. On the other hand, SmartUpdate updates the parameters with the accelerator (i.e., FPGA) in a CSD. Instead of uploading the gradients and optimizer states to host memory, ① SmartUpdate directly loads them to FPGA through direct P2P communication which is possible because of the existence of an internal PCIe switch in each CSD. After the loaded gradients and optimizer states are stored in the accelerator memory, ② the accelerators update the parameters using an update module (orange box). We discuss the detailed architecture in a separate section (Section V-A). ③ While the baseline offloads the optimizer states from the host memory to SSDs after updating, SmartUpdate directly sends back the updated optimizer states to SSD through internal P2P communication. ④ The updated weight parameters ($2M$) are transferred from SSD to the host memory. This traffic did not exist in the baseline, but it is comparably small overhead considering the size of the previous traffic volume was upload/offload optimizer states ($6M$, three single-precision variables per parameter). In addition, this upstream traffic can be overlapped with update steps for other parameters.

When quantifying the total communication volume, the baseline requires $8M$ (gradients + three optimizer states) for both read and write. SmartUpdate minimizes the total communication volume through the system interconnect to only $2M$ read during the update phase for the updated parameters and $2M$ write during backward for the gradients.

When SmartUpdate is used with a single CSD, the bottleneck simply moves from the system interconnect to the CSD-

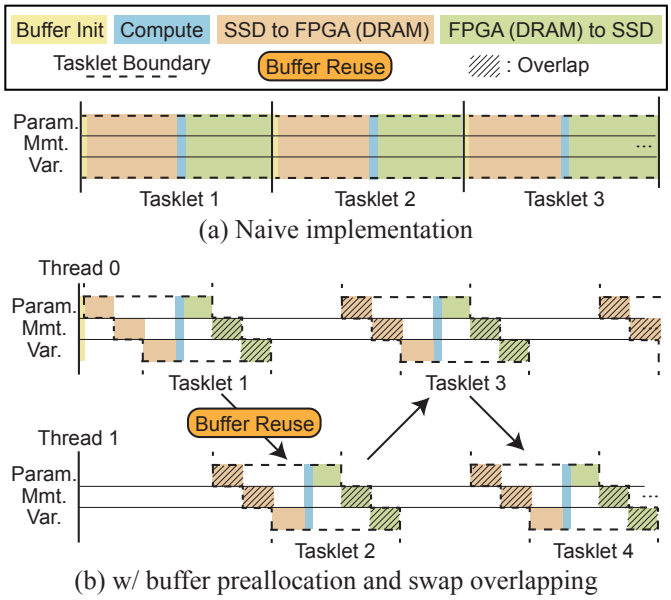


Fig. 5: (a) Naive implementation of SmartUpdate. (b) SmartUpdate with internal data transfer handler optimization.

internal switch. However, the real advantage appears when scaling the number of CSDs. When the number of CSDs increases, the aggregate bandwidth between the FPGA and SSD increases linearly, while that of the system interconnect (PCIe) to the host stays constant. While there is a slight benefit of increased computational capability from the increased number of FPGAs, the aggregated internal bandwidth is the main driver for the speedup of Smart-Infinity.

B. Internal Data Transfer Handler for SmartUpdate

As described in Section II, a single iteration of storage-offloaded training splits the model into multiple blocks and processes a single block at a time. The number of parameters in each block is decided based on the estimated memory requirement. Similarly, SmartUpdate also sets the size of a *subgroup* of model parameters according to the accelerator's device memory capacity, and each subgroup is processed with a single *tasklet*. In a naive implementation shown in Figure 5(a), the tasklets are executed sequentially because each tasklet will occupy the entire memory space. Fortunately, we found that there is room for throughput enhancement by overlapping data transfer between subgroups. However, naive overlapping of data transfers between subgroups requires additional device memory space to direct P2P communication, which leads to out-of-memory (OOM) errors in device memory. Therefore, we propose an internal data transfer handler optimization which leads to better throughput of SmartUpdate, addressing the device memory consumption issue.

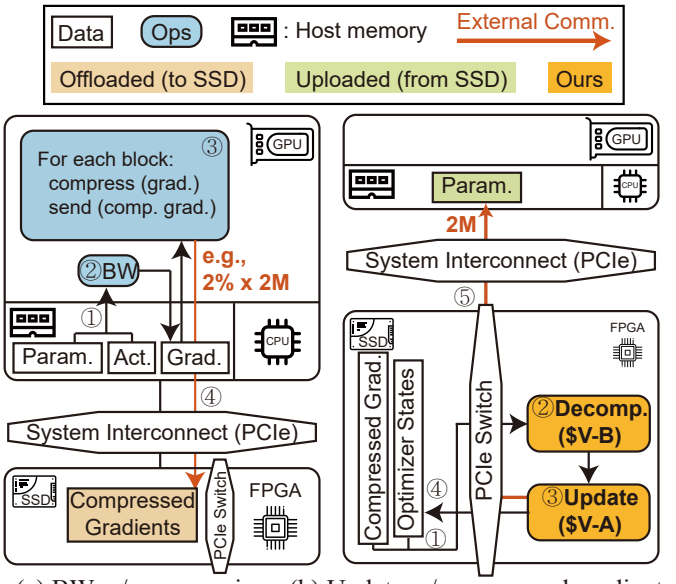
The key idea of the proposed handler is to separate the buffer for variables and lazily transfer non-urgent variables. In a naive implementation (Figure 5(a)), the procedure is as follows: the buffers for current subgroup are allocated, the variables are read from the SSD, parameters are updated (light blue bars), and the variables are written back to the SSD. Afterward, the buffer is deallocated, so a buffer for the next subgroup can be allocated. With our optimization technique (Figure 5(b)), the device memory buffer is preallocated at initialization of SmartUpdate, separately for the largest possible size of each optimizer state variable. We dedicate two threads (thread 0 and 1) running at the CPU to manage the allocated buffers. Initially, thread 0 owns the buffers. When it finishes the update computation, it immediately writes the model parameters back to the SSD, because it is needed by GPUs to conduct forward/backward passes. However, the other variables (e.g., momentum and variance) are not urgent to be written back, because they will be used only at the update phase of the next training iteration. Thus, thread 0 defers the writeback of the remaining variables and signals thread 1 to start loading the model parameters of the next subgroup. Because we have preallocated the buffer for the largest subgroup size, thread 1 can directly reuse the same buffer. For the remaining variables, the writeback of thread 0 and loading of thread 1 are performed similarly by reusing the buffers, but in a lower priority because it is not critical for the forward/backward phases. Through the optimization, we obtain several benefits: 1) avoid reallocation of device memory, 2) the GPU can start forward/backward phases earlier, and 3) the data transfers to the SSDs are overlapped.

C. SmartComp: CSD-aided Gradient Compression

SmartUpdate reduces a significant amount of the SSD traffic by removing the transfer of optimizer states to the host. Given multiple CSDs in the system, the optimizer states ($6M$) are transferred within the CSDs through the internal bandwidth linearly scaled by the number of CSDs. However, gradients ($2M$) still go through the system interconnect, which becomes a new bottleneck when the number of CSDs increases. Unfortunately, it is difficult to overlap gradient offloading with the update step because there are some constraints for mixed precision LLM training before starting the update step. First, not-a-number (NaN) and infinity value (Inf) due to the limited range of half-precision must be checked before the update for loss scaling [84]. Second, the norm of total gradients from the whole model is required for gradient clipping before the update phase. Moreover, the write bandwidth is often far lower than that of the read of SSDs, which aggravates the issue.

Fortunately, the fact that bottleneck is caused by gradients provides an interesting opportunity to Smart-Infinity: compressing gradients. It is widely known that DNN training is tolerable to some degree of errors, especially the gradients. Gradient compression methods [20], [21], [77], [111], [119] are widely used to mitigate communication overhead in neural network training, and it is almost a norm to train modern larger models in distributed settings [98], [111].

With the aid of the accelerator in CSDs, we propose SmartComp that applies gradient compression on top of SmartUpdate: compress the gradients using GPU, and decompress them before updating on CSDs. While there are many variants of algorithms, we implement the magnitude-based compression



(a) BW w/ compression (b) Update w/ compressed gradients

Fig. 6: An overview of SmartComp. (a) The remaining gradient offloading overhead in SmartUpdate is further reduced by gradient compression. (b) FPGA in CSD conducts the decompression of the compressed gradients.

method [20], [77] for SmartComp. In the method, the gradients are first sorted by their magnitude, and higher ones are chosen. The lower-magnitude gradients are replaced by zero because they have a relatively small impact on the original gradient direction compared to the higher-magnitude ones. The compression results are indices list and values list, each representing positions of the high magnitude gradients and corresponding values. This is a viable choice because the relatively heavier job or sorting can be done in powerful GPUs, and the lightweight FPGAs in CSD perform the decompression part that only involves scattering the values according to the index. Another approach, such as low-rank decomposition [23], [119], could be applied to SmartComp. However, tuning the floating-point matrix multiplication performance is challenging and non-trivial [28], [29], [122]. The magnitude-based compression provides comparable model quality at low DSP cost, so we settled on the magnitude-based compression.

Figure 6 illustrates the overall process of SmartComp. After the backward pass generates the gradients (① and ② of Figure 6(a)), ③ GPU compresses the gradients. For example, GPU selects the higher magnitude (e.g., 1%) elements by sorting each block. Instead of offloading the full gradients vector to SSD, ④ only the compressed gradients are offloaded to SSDs. In the update procedure (Figure 6(b)), through the internal CSD P2P communication, ① the compressed gradients and optimizer states are loaded to FPGA. Because the full matrices are required for the element-wise aspect of the update operation (e.g., Adam [63]), we first ② decompress the compressed gradients to the same size as the original gradient vector using index information before the update. For this process, we designed the decompression module (orange box), which will be described in a separate section (Section V-B). After the decompression, ③-⑤ SmartComp follows the same procedure as illustrated in Section IV-A. With SmartUpdate and SmartComp, the remaining data transfers through the system interconnect now become the offloading compressed volume of gradients and upstream updated weight parameters, which are essential information to proceed to the next forward and backward pass in GPU. Combining the facts that the weight parameters are more urgently passed with our internal data transfer handler (Section IV-B) and SSDs are faster on reads, SmartComp greatly helps to scale to the larger number of CSDs on Smart-Infinity.

D. Workload Distribution to Multiple CSDs

In the previous sections, the details of Smart-Infinity were provided using a single CSD for easy understanding. However, Smart-Infinity achieves acceleration from using multiple CSDs, utilizing the high internal bandwidth. Therefore, it is meaningful to discuss how Smart-Infinity manages them.

The key idea of allocating the workload of each CSD is that all the operations in updating optimizer states are conducted in an element-wise manner. Using this, Smart-Infinity flattens the model parameters and equally distributes them to the CSDs, where each CSD takes the responsibility to update the owned parameters. Then, the optimizer states are allocated and initialized at CSDs that own the associated parameters. After a backward pass, the generated gradients are offloaded to the owner CSD, who conducts update computation via P2P communication between its attached FPGA and SSD.

Because of using flattened model parameters, the distribution procedure is agnostic to the model architecture. This allows for a simple adoption of Smart-Infinity, where end users do not need to consider the model architecture information such as the layers, hidden dimensions, or number of heads.

V. Accelerator Architecure

In this section, we introduce the accelerator architecture implemented in the FPGA to support SmartUpdate and SmartComp. Figure 7 shows the microarchitecture of the updater for SmartUpdate (Section IV-A) and decompressor for SmartComp (Section IV-C). Even though Smart-Infinity mainly describes using Adam optimizer and magnitude-based compression method, the accelerator is designed to support various algorithms of user's choice.

Overall, Smart-Infinity's updater and decompressor processes the model in units of a subgroup that fits into the DRAM size of the accelerator (D). Therefore, when processing subgroup i, the parameters with the indices from $i \times D$ to $(i+1) \times D - 1$ are the target parameters. S indicates the processing chunk size that fits into an internal BRAM buffer of the accelerator.

A. General Updater

The lower part of Figure 7 shows the example updater for Adam optimizer. For the updater, Smart-Infinity ① loads

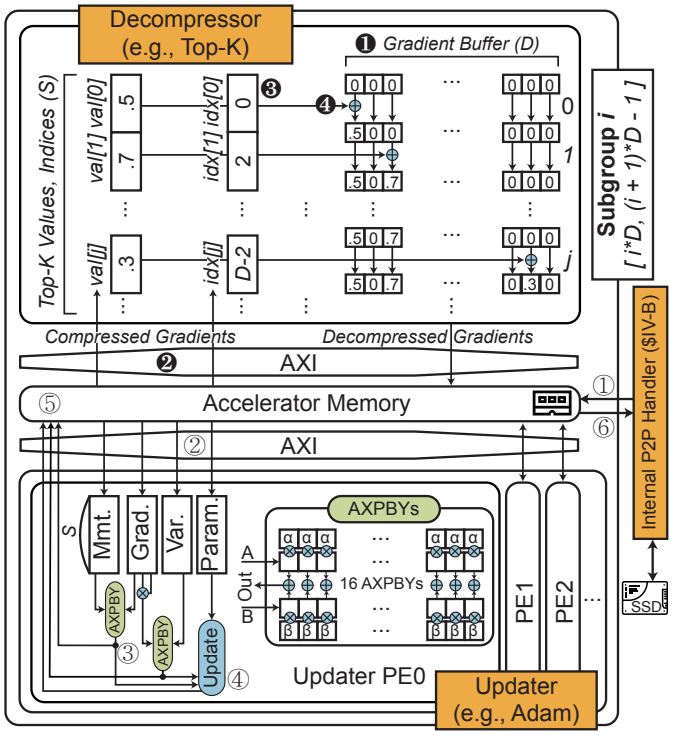


Fig. 7: Smart-Infinity microarchitecture.

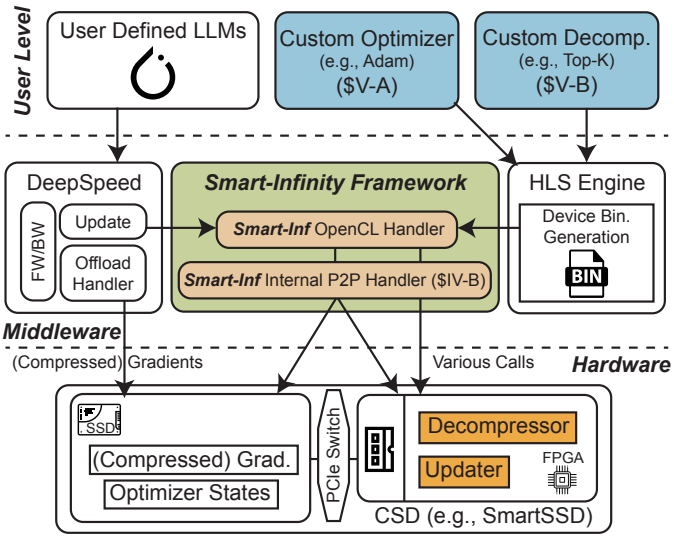


Fig. 8: An overview of Smart-Infinity, which is a flexible ready-to-use framework. Users can apply their own optimizers and decompressor HLS modules with minimal modifications.

the subgroup size (D) amount of the gradients, optimizer states, and target parameters using the optimization techniques illustrated in Section IV-B. The updater consists of multiple updater processing elements (PEs), which update the target parameters in parallel. ② Each PE gets the gradients, the optimizer states, and the target parameters from the accelerator memory. Generally, optimizer algorithms require various types of moving averages. Therefore, we placed SIMD *AXPBY* units, which can handle general averaging operations. It takes two input vectors (i.e., A and B) and multiplies the dedicated coefficients (i.e., α and β) to them. After multiplication, it conducts the element-wise addition of two vectors. The above procedure can be used for various averaging operations by changing the coefficients. ③ Using the AXPBY units, the updater incorporates the optimizer states (e.g., momentum and variance) into the gradients. ④ After all the optimizer states are applied to gradients, the updater updates the target parameter using them. ⑤ The new optimizer states, and the updated parameters are passed to the accelerator memory. ⑥ The updated data are swapped out to the storage of CSD via the data transfer handler of Section IV-B. Note that the updater can be extended to other optimizers such as SGD, AdaGrad [39], and AdamW [78] because most optimizers use variations of moving averaging. We further implemented and tested other updaters in Section VII-F.

B. General Decompressor

As a representative compression method, Smart-Infinity applies a magnitude-based (i.e., Top-K) algorithm. The upper part of Figure 7 is the microarchitecture of the example decompressor for Top-K compression. ❶ At the first iteration, the *gradient buffer* for the target parameters is initialized with zero. ❷ The decompressor gets the compressed gradients from the accelerator memory. A typical Top-K compression algorithm compresses the gradients into two parts, indices and values. Therefore, the decompressor loads the buffer size (S) amount of indices and values iteratively. For each iteration, ❸ the decompressor figures out the target position using the value of indices ($idx[j]$) to map the values ($val[j]$). ❹ In the output vector, the partial Top-K elements of the target parameters selected in this iteration j are filled with their values, and the others are filled with zero. Until decompression finishes, the decompressor repeats ❷-❹. When all Top-K elements are processed, the gradient buffer contains the vector filled with Top-K gradient values dedicated to the target parameters of subgroup i. After decompression completes, decompressed gradient values are now ready in the accelerator memory, so that the updater (Section V-A) can use them.

VI. Implementation

This section describes the main software components of Smart-Infinity on top of DeepSpeed [2], an open-source framework that is highly optimized and widely used for LLM training with PyTorch. We mainly focus on replacing the parameter update feature of DeepSpeed ZeRO-Infinity [97]. To build Smart-Infinity into a callable module, we exploit *disutils* [3] to build C/C++ implementations as additional modules of Python. Because of this, Smart-Infinity can be enabled by simply specifying an option, and automatic compilation starts when the DeepSpeed engine is initialized. This indicates that any LLM training codes implemented with DeepSpeed can be run with Smart-Infinity with no modification, making Smart-Infinity a practical and powerful framework. Note that wrapping any

native PyTorch LLM training code with DeepSpeed is also straightforward and requires little effort.

Figure 8 *User Level* shows the design flow for users to customize Smart-Infinity decompressor/updater. Users can freely modify the customized logic for Smart-Infinity via high-level synthesis (HLS) codes for the decompressor/updater. Smart-Infinity provides HLS templates for implementing their own compression or weight update logic. The templates also consist of a performance analyzer and a sanity checker of logic.

Figure 8 *Middleware* describes interaction between Smart-Infinity and DeepSpeed runtime engine. DeepSpeed supports forward/backward execution in mixed precision. The generated gradients are offloaded during backward execution. We carefully modify the gradient offloading path because the gradients should be located in the corresponding SSD to execute the update in FPGA via internal P2P communication. After updating parameters with Smart-Infinity, the updated parameters should be passed to the DeepSpeed runtime engine. Smart-Infinity engine can directly communicate *'torch.Tensor'* type with PyTorch application using pybind11 [9], a lightweight library that exposes Python types into C++.

Figure 8 *Hardware* illustrates how Smart-Infinity engine interacts with the hardware components. Smart-Infinity uses the Xilinx OpenCL extension [10] to interact with the attached FPGA in runtime. When initializing the device with OpenCL, it is critical to figure out which FPGA device is directly connected via its internal PCIe switch to the specific SSD. As described in Section IV-B, the internal data transfer handler calculates the required device buffer size and initially pre-allocates OpenCL buffer using *'CL_MEM_EXT_PTR_XILINX'* flag at once. Then, the buffer can be re-used for direct internal P2P communication with the attached NVMe SSD. Some standard file access Linux system calls are supported for direct P2P data transfer between SSD and FPGA device memory. We use *pread/pwrite* system call to the P2P buffer, which operates the direct internal P2P communication feature of SmartSSD.

VII. Evaluation

A. Experimental Environment

The overall experimental environments are shown in Table II. To evaluate the efficiency of utilizing CSD, we use at most 10 SAMSUNG SmartSSD [83] devices. Each SmartSSD has a 4TB NVMe SSD which directly communicates with Kintex UltraScale+ KU15P FPGA through a PCIe Gen3.0 x4 bus. The attached FPGA has approximately 522K LUTs, 984 BRAMs, 1968 DSPs, and DDR4 4GB DRAM. We use the NVMe SSD of SmartSSD for a fair comparison with the baseline. We compose software RAID via Linux *mdadm*. We connect SmartSSDs via a PCIe expansion [4], which is helpful to increase the physical slots of the whole system when the PCIe lanes are limited. We further discuss the storage expansion in Section VIII-C. For LLM training, we equipped RTX A5000 (24GB), Tesla A100 (40GB), and RTX A4000 (16GB), which are widely used GPUs to train DNNs. We use RTX A5000 as a default if it is not stated otherwise.

TABLE II
Experimental environment.

HW	GPU	NVIDIA A5000, A100 (40GB), A4000
	CPU	Xeon(R) Gold 6342, 2×48C 96T
	Memory	32×32GB DDR4-3200
	SSD	SAMSUNG SmartSSD, 4TB
	PCIe Expansion	H3 Falcon 4109
SW	OS	Ubuntu 20.04 LTS
	Python / PyTorch	3.9 / 1.12.1
	CUDA / OpenCL	11.6.2 / 2.2
	Vitis / XRT	2023.1 / 2.12.427
	Model	GPT-2, BERT, BLOOM, ViT
	Deepspeed	0.9.3

TABLE III
Resource utilization of implementation results for Adam updater and Adam updater with Top-K decompressor.

Module	LUT (522K)	BRAM (984)	URAM (128)	DSP (1968)
Adam	33.66%	27.13%	34.38%	11.03%
Adam w/ Top-K	34.12%	27.13%	35.94%	11.03%

As described in Section II, DeepSpeed ZeRO-Infinity [97] is highly optimized for overlapping data transfer in storage-offloaded training and widely used in mixed-precision training. Additionally, it has internal implementation with AVX operations to provide faster updates with CPUs. Therefore, we set DeepSpeed ZeRO-Infinity [97] with software RAID0 as the baseline (**BASE**). In the following sections, we notate SmartUpdate as **SU** (§Section IV-A), SmartUpdate with the optimization techniques of internal data transfer handler as **SU+O** (optimized SU, §Section IV-B)), and optimized SmartUpdate with SmartComp as **SU+O+C** (§Section IV-C). If not stated, the default compression ratio is 2% (1% gradients with indices), which indicates that it communicates only 2% of the original communication volume. And we used batch size 4 as our default setting, since we targeted the situation where GPU memory size is limited. In addition, we mainly chose two different language models (GPT2 [94], BERT [34]), which are representative decoder/encoder-only models to measure speedup and accuracy. Further, we conduct experiments in Figure 13 for the other two models (BLOOM [103], ViT [37])

B. Implementation Results

Table III shows the resource utilization of the implementation results of the microarchitectures on the FPGA, for the Adam optimizer and Top-K decompressor. Note that in the actual usage of Smart-Infinity framework, the user does not always need to implement the modules manually except when they need customized logic. Smart-Infinity provides general templates for generating the device binary, as discussed in Section VI. For implementing Adam, the updater consumes roughly a quarter of the available resources for implementing floating point AXPBY vector arithmetic and pipeline registers. On the other hand, the decompressor consumes a small amount

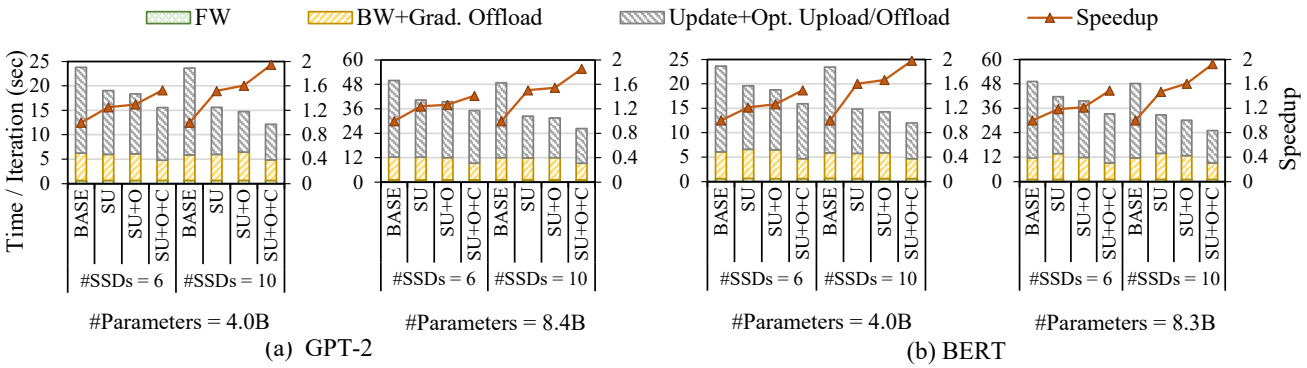


Fig. 9: Training time breakdown and speedup of Smart-Infinity over the baseline (BASE). SU indicates SmartUpdate, and SU+O means the optimized SmartUpdate with internal data transfer handler. SU+O+C uses SmartComp on top of SU+O.

of logic, because the Top-K decompressor only requires routing the value to the right location without any arithmetic. There is much room left for extra logic despite the FPGA being lightweight, possibly allowing other extensions of applications.

C. Performance Comparison and Ablation

Figure 9 shows the training time breakdown and the speedup of Smart-Infinity compared to the baseline with ablation. We break down the training time into three parts, forward time, backward with gradient offload time, and update with the optimizer states upload and offload time. Overall, the baseline training time results suffer from the severe communication overhead as discussed in Section II and Section III. In GPT-2 8.4B model with 6 SSDs, the update time with the optimizer states communication time consumes 75.57% of the training time of the baseline. Even when the number of SSDs increases to 10, the training time stays constant due to the system interconnect bottleneck, and this trend persists for all test cases. SmartUpdate (SU) reduces this communication volume through the system interconnect, providing 1.18×∼1.24× speedup with 6 SSDs, and 1.54×∼1.60× speedup with 10 SSDs. Further applying the transfer handler optimization (SU+O), the speedup from SmartUpdate is boosted to up to 1.60×∼1.66× with 10 SSDs. This result suggests that the optimization techniques successfully overlap the SSD-FPGA communications. Additionally, on the top of the optimized SmartUpdate, SmartComp provides 1.22×∼1.31× additional speedup over SmartUpdate, showing 1.85×∼1.98× speedup over the baseline. Through the breakdown results, we can find out that SmartComp successfully reduces the remaining gradient offloading time of the optimized SmartUpdate further. For all models tested, the speedup trend is almost identical. This is because the modern LLM models are all based on Transformers [118] and only differ in some model design parameters. Furthermore, in all cases, the main bottleneck is the storage bandwidth, making the model size or structure relatively less important for the training time. This explains the constant speedup trends.

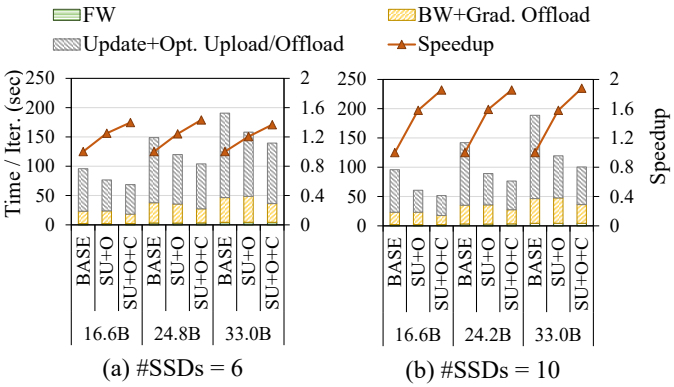


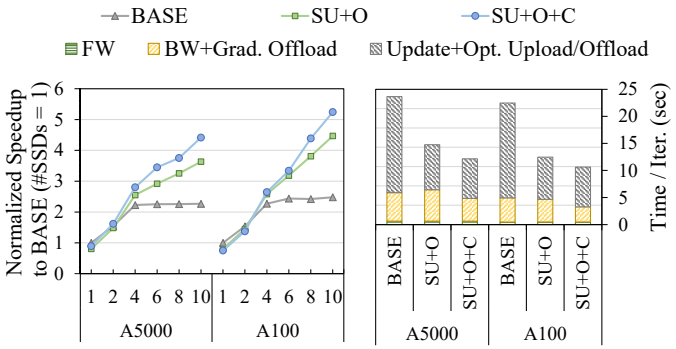
Fig. 10: Scalability on larger model sizes (16.6B to 33.0B) of Smart-Infinity compared to the baseline.

D. Scalability to Larger Models

Storage-offloaded training allows training much larger models using limited resources. Therefore, it is essential to check whether Smart-Infinity brings stable speedup over the baseline on the various large model sizes. Figure 10 is the scalability test of Smart-Infinity on larger GPT models with various sizes compared to the baseline. The results show that Smart-Infinity achieves consistent speedup with larger models. Even with GPT-2 33.0B model, Smart-Infinity provides 1.37× and 1.88× speedup over the baseline using 6 SSDs and 10 SSDs, respectively. The speedups are stable on large models because the communication portion of the training time is maintained. It is natural because the communication volume is proportional to the number of model parameters in transformer model training. Additionally, using more number of CSDs still brings speedup to the larger models. Therefore, Smart-Infinity is scalable on larger models while the baseline suffers from the huge uploading and offloading overheads.

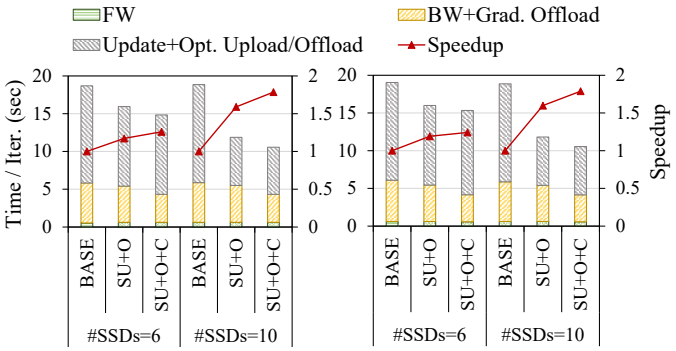
E. Experiments on the Number of CSDs and GPU Grade

Smart-Infinity benefits from the aggregated bandwidth of CSDs, so it is essential to check the sensitivity of speedup on the number of CSDs. Additionally, it is helpful to check the



(a) Scalability with #SSDs (b) Breakdown

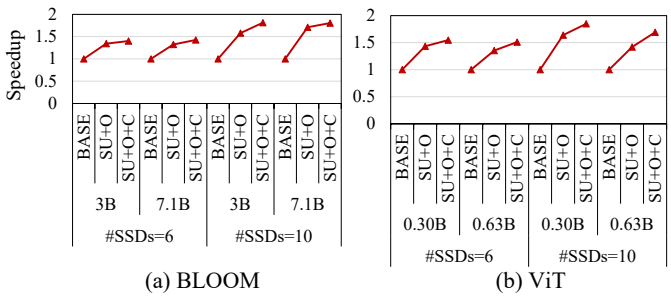
Fig. 11: (a) Scalability with #SSDs of Smart-Infinity compared to the baseline in A5000 and A100 GPU settings. (b) Training time breakdown of Smart-Infinity and the baseline in both GPU settings with ten SSDs.



(a) SGD (b) AdaGrad

Fig. 12: Applying SmartUpdate to other optimizers.

speedup when using a higher-end GPU. In Figure 11(a), we scaled the number of CSD from one to 10 while keeping the model as GPT-2 4.0B with the default A5000 GPU and the higher-end A100 GPU. In both GPU setups, the baseline does not scale beyond 4 SSDs, whose aggregate SSD bandwidth reaches roughly that of the PCIe system interconnect bandwidth. On the other hand, Smart-Infinity mainly depends its speedup on the aggregate internal bandwidth, not the system interconnect. Because of this, Smart-Infinity shows almost linear speedups along with increasing the number of CSDs. On a single CSD, there is a slight slowdown, which is expected. It is because there is no bandwidth increase with a single CSD, and the base system overhead to utilize the FPGA adds a slight overhead. When utilizing a higher-end GPU (i.e., A100), the portion of computation (FW and BW) becomes smaller. Therefore, the portion of data transfer gets larger, and this causes the speedups in the A100 GPU setting to be generally higher than the ones in the default A5000 setting. Additionally, Figure 11(b) breaks down the training time when using ten SSDs in both settings, and it shows that Smart-Infinity still provides up to 2.11× speedup in the A100 GPU setting.



(a) BLOOM (b) ViT

Fig. 13: Application of Smart-Infinity to BLOOM and Vision Transformer (ViT).

F. Extension to Other Optimizers

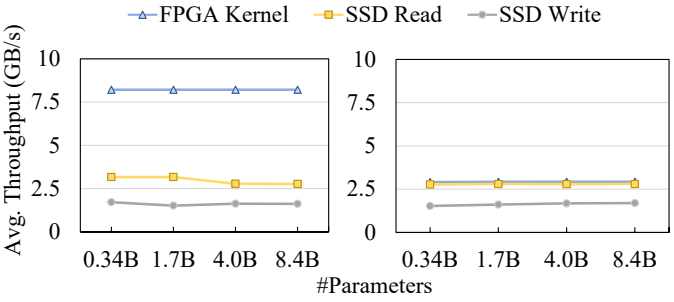
Smart-Infinity provides a general updater that can support various optimizers. Besides the default Adam updater, we additionally implemented the updaters for stochastic gradient descent (SGD) with momentum and AdaGrad [39]. Figure 12 shows the speedup of Smart-Infinity over the baseline when changing the optimizer type in GPT-2 4.0B. SGD requires $3/4\times$ less optimizer states because it uses three states (parameter, gradient, and momentum) instead of four. Therefore, the speedup when using SGD becomes slightly lower than that of Adam. AdaGrad also incurs $3/4\times$ offloading volume (parameter, gradient, and variance) compared to Adam, so the speedup is similar to the SGD. Recent optimizers generally use a similar or larger number of optimizer states, indicating Smart-Infinity would be an efficient way for other optimizers.

G. Applying to Other Models

Smart-Infinity can be generally applied to other transformer-based models with heavy memory overhead. To check such applicability, we tested the speedup of Smart-Infinity on BLOOM [103] and vision transformer (ViT [37]) in Figure 13. BLOOM is another open-sourced multilingual LLM. ViT is a representative transformer-based model for vision-related tasks. They are chosen to demonstrate that Smart-Infinity is not limited to specific model sensitive aspects. On such models, Smart-Infinity stably shows $1.32\times\sim1.85\times$ speedup, similar to the results we have shown for GPT-2 and BERT.

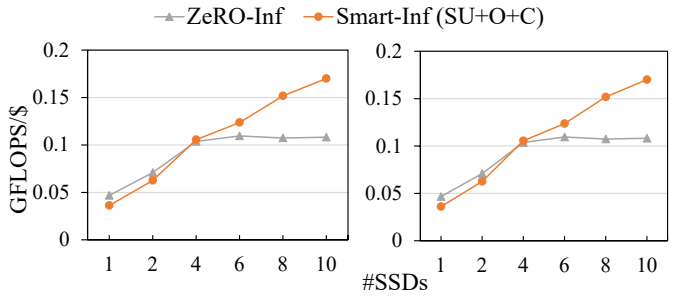
H. Accelerator Throughput Analysis

For achieving high performance with Smart-Infinity, it is crucial that the throughput of the updater and the decompressor modules keep up with the SSD bandwidth. Therefore, we compared the throughput of updater, decompressor, and SSD read/write in Figure 14. The throughput of the updater is above 7GB/s, which is sufficiently higher than the SSD read/write. The decompressor also slightly surpasses the throughput of the SSD read. In principle, it could be possible to obtain higher decompression speed by deploying more such engines considering the FPGA resource utilization. However, we chose to save the FPGA resources for later extensions (Section VIII-B) as it was not slowing down the system.



(a) Update (b) Decompress + Update

Fig. 14: Computational throughput of Smart-Infinity's modules compared to NVMe SSD read and write performance.



(a) A5000 (b) A100

Fig. 15: Performance/$ of the baseline and Smart-Infinity.

I. System Cost Efficiency Analysis

It is meaningful to directly compare the system cost efficiency of Smart-Infinity with the baseline storage-offloaded training [97]. In Figure 15, we analyze their system cost compared to GFLOPS/$ in the setting used for Figure 9 (GPT-2 4.0B). The system costs include the server cost (around $45,000, including CPU, RAM, PCIe expansion, etc.), storage cost, CSD (i.e., SmartSSD) cost, and GPU cost (around $2,000, $7,000 for A5000 and A100, respectively). The cost of a SmartSSD is around $2,400, which is 6× more expensive than the same capacity SSD storage ($400), so Smart-Infinity with 1-3 CSDs shows lower GFLOPS/$ than the baseline. However, when using more than four SSDs, the speedup over the baseline makes Smart-Infinity more efficient. When scaling the number of SmartSSDs, the GFLOPS/$ keeps increasing, showing the efficiency of Smart-Infinity.

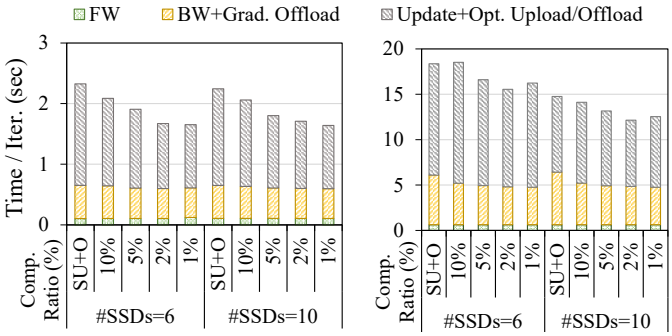
J. Application Case Study: Fine-tuning

Fine-tuning is a representative application where Smart-Infinity can be applied because it requires loading the entire model and the optimizer states but has a relatively small training time. Thus, we demonstrate fine-tuning results with pre-trained LLMs on various datasets using Smart-Infinity.

For experiments, publicly available pre-trained weights were used to conduct fine-tuning tasks. We obtained pre-trained weights of BERT-345M from Megatron-LM GitHub [108], and GPT-2 (774M and 1.6B) from the huggingface hub [5]. Table IV shows the development set results of four datasets

TABLE IV
FINETUNING ACCURACY AND SPEEDUP COMPARISON.

Model	Method	Speedup (#SSDs=6)	Accuracy (%) MNLI m/mm	QQP	SST-2	QNLI
BERT-0.34B	Baseline	1×	89.60/89.46	91.95	93.98	94.23
	SU+O	1.10×	89.60/89.46	91.95	93.98	94.23
	SU+O+C (10%)	1.23×	89.42/89.55	91.69	94.95	94.16
	SU+O+C (5%)	1.34×	89.24/89.38	91.90	95.41	93.96
	SU+O+C (2%)	1.38×	89.50/89.39	91.80	95.41	94.30
	SU+O+C (1%)	1.40×	89.61/89.33	91.75	95.53	94.03
GPT2-0.77B	Baseline	1×	85.95/86.60	90.96	94.27	91.60
	SU+O	1.11×	85.95/86.60	90.96	94.27	91.60
	SU+O+C (10%)	1.23×	85.71/86.10	90.71	94.27	91.21
	SU+O+C (5%)	1.29×	85.31/85.87	90.38	94.38	91.30
	SU+O+C (2%)	1.35×	85.05/85.25	90.00	94.27	90.83
	SU+O+C (1%)	1.44×	84.57/85.39	89.77	94.27	90.66
GPT2-1.6B	Baseline	1×	87.32/87.00	91.45	95.18	92.22
	SU+O	1.29×	87.32/87.00	91.45	95.18	92.22
	SU+O+C (10%)	1.45×	86.71/87.23	91.15	94.84	91.74
	SU+O+C (5%)	1.54×	86.88/87.16	91.04	94.38	91.65
	SU+O+C (2%)	1.54×	86.67/86.71	90.83	94.61	91.62
	SU+O+C (1%)	1.53×	86.58/86.69	90.56	94.72	91.45



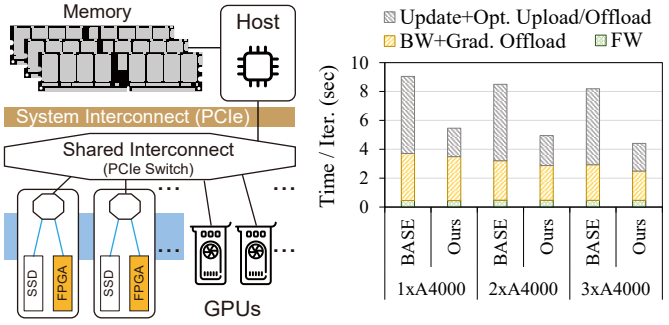
(a) BERT-0.34B (b) GPT 4.0B

Fig. 16: Training time sensitivity on Top-K compression ratio of Smart-Infinity.

included in the GLUE benchmark [120]. For BERT-345M, we trained the model with 10 epochs for MNLI and 12 epochs for QQP following [108]. The model was trained with 3 epochs for SST-2 and QNLI to follow hyperparameters of [34]. We fixed the batch size as 4, which is the same batch size as the default experimental setup in Section VII-A. For GPT models, the model was finetuned for 3 epochs. To use the mixed precision training and pre-trained weights from the huggingface, the accuracy was measured with the autocast feature, which performs FP32 operations for the numerically unstable operations (e.g., softmax). The results show that Smart-Infinity can stably achieve fine-tuning accuracy for various datasets. SmartUpdate is algorithmically identical to the baseline training, so the accuracy is exactly the same as the baseline. SmartComp adopts lossy compression, but achieves comparable accuracy in all datasets.

K. Sensitivity of Compression Ratio

As a representative gradient compression, SmartComp uses a magnitude-based one. We used 2% compression (top 1% selection, an index-value pair per element) as a default because it is the usual practice in various works [23], [77]. To further



(a) Congested topology (b) Breakdown (#SSDs = 10)

Fig. 17: (a) An example of multi-GPU congested topology. (b) Training time breakdown of Smart-Infinity and the baseline on such environment with 1∼3 GPUs (GPT-2 1.16B).

investigate how Smart-Infinity is sensitive to compression ratio, we experimented on various compression ratios from 1∼10% in Table IV and Figure 16. There exists some trade-off between speedup and model quality. However, it does not significantly harm model quality even with much more compression than the default, but the speedup almost gradually increases.

VIII. Discussion

A. An Alternative Scenario: Multi-GPU Congested Topology

When PCIe lanes of a system are more limited, GPUs and storages could be installed in the same PCIe expansion sharing the same PCIe switch, so the peer-to-peer communication topology changes. To test such a scenario, we additionally equipped a congested topology with up to three GPUs, as illustrated in Figure 17(a). Due to the chassis limit of PCIe expansion, we inserted single-slot RTX A4000 GPUs into the expansion. Figure 17(b) shows the training time and speedup of Smart-Infinity over the baseline when using 1∼3 A4000 GPUs. We adopted tensor parallelism for the multi-GPU strategy because it is widely selected in a single server setting [108]. After gradients are calculated, each GPU identifies CSDs that own the corresponding parameters and performs updates with Smart-Infinity.

Using tensor parallelism slightly reduces 'FW' and 'BW' time when utilizing more GPUs. However, in this alternative scenario, the remotely placed GPUs incur some extra traffic of model and activation transfer, which has to share the same PCIe interconnect with the CSDs. This causes some overhead to the 'BW+Grad. Offload' phase compared to the default topology, while not greatly affecting the time of 'Update+Opt. Upload/Offload' phase where the GPUs are idle. Because of this, the observed speedup is smaller than the default setup, which indicates that the performance is affected by how PCIe topologies are structured. However, Smart-Infinity still provides 1.66×∼1.86× speedup with ten CSDs, which shows that it could be extended to a more PCIe lane-limited environment with multiple GPUs.

B. Applying Smart-Infinity to Model Compression

In this work, we have used fine-tuning to demonstrate the usefulness of Smart-Infinity. Various types of model compression methods can be usecases of Smart-Infinity, such as quantization [25], [36], pruning [45], [46], or low-rank decomposition [121] because they require some fine-tuning in compressed form to recover accuracy drop from compression.

Interestingly, those applications are expected to bring even more speedup to Smart-Infinity. As discussed in Section IV-C, the current bottleneck of Smart-Infinity is on the upstream model transfer from the CSD to the host. When Smart-Infinity is used for model compression, Smart-Infinity can perform compression and upload the compressed model, further reducing the bottleneck.

However, achieving further speedup from this would bring some non-trivial issues. For example, quantization often performs backpropagation with straight-through estimator (STE) [14] with variable (floating point) quantization intervals per layer. This means that the training GPUs use floating point models instead of integers, rather counter-intuitively. To address the issue, the CSDs would have to derive the per-layer quantization intervals, convert the models to integers, and send the parameters upstream along with the interval values. Then, the GPUs convert the integer weights into floating points using the intervals, so that it can perform STE. For pruning and low-rank decomposition, similar issues can be found. This indicates that the CSD now has to handle a more computationally complex job of compression using a lightweight FPGA, and an efficient GPU kernel for decompression has to be developed to not introduce additional bottlenecks. We leave this as a future work, and foresee more interesting approaches for model compression on top of Smart-Infinity.

C. Storage Expansion and Pooling

Smart-Infinity utilizes a storage expansion system using PCIe switches to accommodate multiple CSDs. As modern workloads such as LLMs, recommendation, big data analytics [43], or bioinformatics demand more memory and storage capacity, new proposals are being made on multiple servers sharing storages [4] and memories [30]. Especially when a large capacity is in need, often the choice is to equip such switches to multiply available slots for memory or storage. These are often pooled among multiple servers to adapt to dynamic capacity requirements over time, and some provide direct GPU-initiated NVMe accesses [93]. We believe Smart-Infinity is a great fit for such a movement. Expansion using switches increases the number of physical slots for storage devices, but not the raw link bandwidth to the host. When more devices are used for more capacity, then the link bandwidth bottleneck will only get severer. Using techniques such as Smart-Infinity with CSDs, more capacity means more internal bandwidth and computational capability to utilize them. Along with such evolving system architecture toward sharing more resources, these types of solutions are expected to prevail.

IX. Related Work

A. Near-Data Processing

The idea of near-data processing has been studied, ranging from DRAM, SRAM, NVRAM to storage. The early research on near-data processing was led by integrating DRAM with logic around 90's [38], [57], [64], [65], [79], [90]. Later, with the surge of 3D stacked memories [91], several ideas were suggested to utilize the logic dies stacked with memory dies for various applications [8], [11], [16], [32], [41], [51], [60], [86], [127], [128], [130]. Recently, driven by the discontinuity of the 3D stacked memory, both academia and industry started turning back to the DDR variants, sometimes by modifying the die-internal circuitry [33], [47], [59], [68], [70], [73], [75], [105], [106], using buffer chips on DIMMs [13], [40], [58], [69], or both [61], [89]. While the principles are similar, these are backed by commercial products [33], [68], [73].

Such movements have also been made on storage devices. Starting from ideas of separate accelerators near the IO subsystem [109], [123], earlier ideas were to utilize the embedded cores for computational approaches, led by database or big data workloads [35], [56], [117]. There were many follow-up approaches targeting various issues [52], [54], [55], [66], [82], [124]. However, the embedded cores often turned out to be low in computational power for many applications [24], [100], [117], and more approaches started having dedicated accelerators. ASICs were used for genome sequence analysis [81], private information retrieval [76], or ML queries [80]. Some approaches employ FPGAs to enhance flexibility targeting similar issues [53], [71], [100], [104], [114], [125]. It is worth noting GradPIM [61] and OptimStore [62] as they both target DNN training similar to this work. However, they assume dedicated memory/storage die, making those solutions less practical. Moreover, they were only implemented on simulators and do not consider real system integration issues.

Recently, several commercial products came out, equipped with an FPGA accelerator in the SSD package [1], [7], [18], [83] for near-storage processing. SmartSSD [83] is a representative one, which was especially used for DB workloads [72] and sorting [101], [102]. A similar platform was deployed in a commercial cloud for DB scans [18]. There are some works [50], [74], [110] utilizing CSD for near-storage processing to accelerate training neural networks, but they mainly focus on preprocessing training data or embedding tables to reduce storage overhead. To the best of our knowledge, Smart-Infinity is the first work using multiple CSDs to mitigate system interconnect bottlenecks in DNN training. Our work builds on SmartSSD but is not limited to certain products.

B. DNN Training Acceleration

As deep learning models grow and training time becomes longer [34], [85], [94], [115], many previous works aimed to reduce training time. Distributed training is one attractive way of using multiple workers. Data parallelism [27], [31], [42], [129] replicates the entire model, and each worker holds it to process training batch independently. However, when the model size exceeds the worker memory limit, the model needs to be split across workers, and model parallelism [19], [22], [67] enables training such large models. Pipelining approach [48], [49], [87], [126] has been proposed to address a low utilization of workers in model parallelism. Each worker concurrently participates by dividing a minibatch into smaller microbatches and overlapping them.

In this distributed training, communication among servers becomes a bottleneck. Gradient compression is a popular approach to reduce such communication overhead. It effectively reduces training time while achieving comparable accuracy, even in large models [98], [111]. It can largely be categorized into two groups: gradient sparsification [15], [21], [77] and low-rank decomposition [23], [119]. The former sends some portion of gradients, usually using the magnitude criteria. Its convergence was proven in previous works [12], [107], and it showed comparable accuracy. Low-rank decomposition factorizes a gradient matrix into two small low-rank matrices to reduce the communication volume. [119] used the power iteration method to reduce the decomposition overhead. Both approaches use error compensation, which memorizes errors from the compression and adds them to gradients at the next step before compression. [116] found that error compensation does not apply to the Adam optimizer due to its nonlinearity. Therefore, it preconditioned the variance term after a warm-up period to make it equivalent to momentum SGD.

X. Conclusion

We propose Smart-Infinity, a novel CSD-based framework to accelerate storage-offloaded LLM training. By utilizing the proposed FPGA accelerators in each CSD, the transfer bottleneck from LLM training is efficiently addressed. We further provide transfer optimizations and CSD-assisted gradient compression to boost the speedup of the proposed system. Smart-Infinity has been fully integrated into PyTorch only with off-the-shelf products, making itself a ready-to-use solution. Experimental results show that Smart-Infinity achieves a significant speedup, and shows good scalability to the increased number of CSDs attached to the system. Our implementation is available at https://github.com/AIS-SNU/smart-infinity.

Acknowledgements

This work was supported by Samsung Electronics Co., Ltd (IO221213-04119-01), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2022R1C1C1011307, 2022R1C1C1008131), and Institute of Information & communications Technology Planning & Evaluation (IITP) under the artificial intelligence semiconductor support program to nurture the best talents (IITP-2023-RS-2023-00256081) grant funded by the Korea government (MSIT). The experimental environment was provided by Samsung Memory Research Center (SMRC). Hongsun Jang, Jaewon Jung, Youngsok Kim, and Jinho Lee were partly supported by the BK21 FOUR (Fostering Outstanding Universities for Research) funded by the Ministry of Education (MOE) and the National Research Foundation (NRF) of Korea.

REFERENCES

[1] "CSD 3000." [Online]. Available: https://scaleflux.com/products/csd-3000/

[2] "Deepspeed." [Online]. Available: https://github.com/microsoft/DeepSpeed

[3] "Distutils." [Online]. Available: https://docs.python.org/3.9/library/distutils.html

[4] "Falcon 4109." [Online]. Available: https://www.h3platform.com/product-detail/overview/25

[5] "Hugging Face." [Online]. Available: https://huggingface.co/docs/hub/index

[6] "NGD systems newport computational storage platform." [Online]. Available: https://www.ssdcompute.com/Newport-Platform.asp

[7] "NoLoad computational storage processor." [Online]. Available: https://www.eideticom.com/products.html

[8] "PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture."

[9] "pybind11." [Online]. Available: https://github.com/pybind/pybind11

[10] "Xilinx OpenCL extension." [Online]. Available: https://xilinx.github.io/XRT/master/html/opencl_extension.html

[11] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.

[12] D. Alistarh, T. Hoefler, M. Johansson, S. Khirirat, N. Konstantinov, and C. Renggli, "The convergence of sparsified gradient methods," in *NeurIPS*, 2018.

[13] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems," in *MICRO*, 2016.

[14] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[15] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar, "SignSGD with majority vote is communication efficient and fault tolerant," *arXiv preprint arXiv:1810.05291*, 2018.

[16] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *ASPLOS*, 2018.

[17] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *NeurIPS*, 2020.

[18] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, F. Zhu, and T. Zhang, "POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database," in *FAST*, 2020.

[19] C.-C. Chen, C.-L. Yang, and H.-Y. Cheng, "Efficient and robust parallel DNN training through model parallelism on multi-gpu platform," *arXiv preprint arXiv:1809.02839*, 2018.

[20] C.-Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, and K. Gopalakrishnan, "AdaComp : Adaptive residual gradient compression for data-parallel distributed training," in *AAAI*, 2018.

[21] C.-Y. Chen, J. Ni, S. Lu, X. Cui, P.-Y. Chen, X. Sun, N. Wang, S. Venkataramani, V. V. Srinivasan, W. Zhang, and K. Gopalakrishnan, "ScaleCom: Scalable sparsified gradient compression for communication-efficient distributed training," in *NeurIPS*, 2020.

[22] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[23] M. Cho, V. Muthusamy, B. Nemanich, and R. Puri, "Gradzip: Gradient compression using alternating matrix factorization for large-scale deep learning," in *NeurIPS*, 2019.

[24] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: A case for intelligent ssds," in *ICS*, 2013.

[25] J. Choi, S. Venkataramani, V. V. Srinivasan, K. Gopalakrishnan, Z. Wang, and P. Chuang, "Accurate and efficient 2-bit quantized neural networks," in *MLSys*, 2019.

[26] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "PaLM: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[27] V. Codreanu, D. Podareanu, and V. Saletore, "Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train," *arXiv preprint arXiv:1711.04291*, 2017.

[28] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS today: Successes, challenges, and opportunities," *ACM TRETS*, vol. 15, no. 4, pp. 1–42, 2022.

[29] J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *ICCAD*, 2018.

[30] C. Consortium, "Compute express link." [Online]. Available: https://www.computeexpresslink.org

[31] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng, "Large scale distributed deep networks," in *NeurIPS*, 2012.

[32] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "DrAcc: A DRAM based accelerator for accurate CNN inference," in *DAC*, 2018.

[33] F. Devaux, "The true processing in memory accelerator," in *HCS*, 2019.

[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[35] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," in *SIGMOD*, 2013.

[36] Z. Dong, Z. Yao, A. Gholami, M. Mahoney, and K. Keutzer, "HAWQ: Hessian aware quantization of neural networks with mixed-precision," in *ICCV*, 2019.

[37] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *ICLR*, 2021.

[38] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The architecture of the DIVA processing-in-memory chip," in *ICS*, 2002.

[39] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of machine learning research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[40] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *HPCA*, 2015.

[41] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *ASPLOS*, 2017.

[42] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[43] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," in *ISCA*, 2016.

[44] Y. Gu, X. Han, Z. Liu, and M. Huang, "PPT: Pre-trained prompt tuning for few-shot learning," in *ACL*, 2022.

[45] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[46] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *NeurIPS*, 2015.

[47] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. Vijaykumar, "Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning," in *MICRO*, 2020.

[48] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *NeurIPS*, 2019.

[49] H. Jang, J. Jung, J. Song, J. Yu, Y. Kim, and J. Lee, "Pipe-BD: Pipelined parallel blockwise distillation," in *DATE*, 2023.

[50] Y. Jang, S. Kim, D. Kim, S. Lee, and J. Kung, "Deep partitioned training from near-storage computing to DNN accelerators," *IEEE CAL*, vol. 20, no. 1, pp. 70–73, 2021.

[51] L. Jiang, M. Kim, W. Wen, and D. Wang, "XNOR-POP: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 DRAMs," in *ISLPED*, 2017.

[52] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "BlueDBM: An appliance for big data analytics," in *ISCA*, 2015.

[53] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "GraFBoost: Using accelerated flash storage for external graph analytics," in *ISCA*, 2018.

[54] L. Kang, Y. Xue, W. Jia, X. Wang, J. Kim, C. Youn, M. J. Kang, H. J. Lim, B. Jacob, and J. Huang, "Iceclave: A trusted execution environment for in-storage computing," in *MICRO*, 2021.

[55] S. Kang, J. An, J. Kim, and S.-W. Jun, "Mithrilog: Near-storage accelerator for high-performance log analytics," in *MICRO*, 2021.

[56] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *MSST*, 2013.

[57] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an advanced intelligent memory system," in *ICCD*, 1999.

[58] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, K. Kim, J. Jung, I. Yun, S. J. Park, H. Park, J. Song, J. Cho, K. Sohn, N. S. Kim, and H.-H. S. Lee, "Near-memory processing in action: Accelerating personalized recommendation with AxDIMM," *IEEE Micro*, vol. 42, no. 1, pp. 116–127, 2022.

[59] B. Kim, J. Chung, E. Lee, W. Jung, S. Lee, J. Choi, J. Park, M. Wi, S. Lee, and J. H. Ahn, "Mvid: Sparse matrix-vector multiplication in mobile DRAM for accelerating recurrent neural networks," *IEEE TC*, vol. 69, no. 7, pp. 955–967, 2020.

[60] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," in *ISCA*, 2016.

[61] H. Kim, H. Park, T. Kim, K. Cho, E. Lee, S. Ryu, H.-J. Lee, K. Choi, and J. Lee, "GradPIM: A practical processing-in-DRAM architecture for gradient descent," in *HPCA*, 2021.

[62] J. Kim, M. Kang, Y. Han, Y.-G. Kim, and L.-S. Kim, "OptimStore: In-storage optimization of large scale DNNs with on-die processing," in *HPCA*, 2023.

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

[64] G. Kirsch, "Active memory: Micron's yukon," in *IPDPS*, 2003.

[65] P. M. Kogge, "EXECUBE-A new architecture for scaleable MPPs," in *ICPP*, 1994.

[66] G. Koo, K. K. Matam, T. I, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: Trading communication with computing near storage," in *MICRO*, 2017.

[67] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[68] Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Kim, J. Lee, I. Kim, J. Park, C. Park, Y. Song, B. Yang, H. Lee, S. Kim, D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, M. Lee, M. Shin, M. Shin, J. Cha, C. Jung, K. Chang, C. Jeong, E. Lim, I. Park, J. Chun, and S. Hynix, "System architecture and software stack for GDDR6-AiM," in *HCS*, 2022.

[69] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *MICRO*, 2019.

[70] J. Lee, J. H. Ahn, and K. Choi, "Buffered compares: Excavating the hidden parallelism inside DRAM architectures with lightweight logic," in *DATE*, 2016.

[71] J. Lee, H. Kim, S. Yoo, K. Choi, H. P. Hofstee, G.-J. Nam, M. R. Nutter, and D. Jamsek, "ExtraV: Boosting graph processing near storage with a coherent accelerator," *pVLDB*, vol. 10, no. 12, p. 1706–1717, 2017.

[72] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "SmartSSD: FPGA accelerated near-storage data analytics on SSD," *IEEE CAL*, vol. 19, no. 2, pp. 110–113, 2020.

[73] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product," in *ISCA*, 2021.

[74] Y. Lee, J. Chung, and M. Rhu, "SmartSAGE: Training large-scale graph neural networks using in-storage processing architectures," in *ISCA*, 2022.

[75] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *MICRO*, 2017.

[76] J. Lin, L. Liang, Z. Qu, I. Ahmad, L. Liu, F. Tu, T. Gupta, Y. Ding, and Y. Xie, "INSPIRE: In-storage private information retrieval via protocol and architecture co-design," in *ISCA*, 2022.

[77] Y. Lin, S. Han, H. Mao, Y. Wang, and B. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," in *ICLR*, 2018.

[78] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *ICLR*, 2019.

[79] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz, "Smart Memories: a modular reconfigurable architecture," in *ISCA*, 2000.

[80] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. De Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu, "Deepstore: In-storage acceleration for intelligent queries," in *MICRO*, 2019.

[81] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alserr, R. Ausavarungnirun, N. Vijaykumar, M. Alser, and O. Mutlu, "GenStore: A high-performance in-storage processing system for genome sequence analysis," in *ASPLOS*, 2022.

[82] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "GraphSSD: graph semantics aware SSD," in *ISCA*, 2019.

[83] P. Mehra, "Samsung smartssd: Accelerating data-rich applications," *Flash Memory Summit*, 2019.

[84] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *ICLR*, 2018.

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

[86] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *HPCA*, 2017.

[87] 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.

[88] 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.

[89] J. Park, B. Kim, S. Yun, E. Lee, M. Rhu, and J. H. Ahn, "Trim: Enhancing processor-memory interfaces with scalable tensor reduction in memory," in *MICRO*, 2021.

[90] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[91] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *HCS*, 2011.

[92] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj, "Training large neural networks with constant memory using a new execution algorithm," *arXiv preprint arXiv:2002.05645*, 2020.

[93] Z. Qureshi, V. S. Mailthody, I. Gelado, S. Min, A. Masood, J. Park, J. Xiong, C. J. Newburn, D. Vainbrand, I.-H. Chung, M. Garland, W. Dally, and W.-m. Hwu, "GPU-initiated on-demand high-throughput storage access in the BaM system architecture," in *ASPLOS*, 2023.

[94] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, 2019.

[95] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv preprint arXiv:1910.10683*, 2019.

[96] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations toward training trillion parameter models," in *SC*, 2020.

[97] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning," in *SC*, 2021.

[98] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *ICML*, 2021.

[99] 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 *USENIX ATC*, 2021.

[100] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing in-storage computing system for emerging high-performance drive," in *USENIX ATC*, 2019.

[101] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, "NASCENT: Near-storage acceleration of database sort on SmartSSD," in *FPGA*, 2021.

[102] S. Salamat, H. Zhang, Y. S. Ki, and T. Rosing, "NASCENT2: Generic near-storage sort accelerator for data analytics on SmartSSD," *ACM TRETS*, 2022.

[103] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," *arXiv preprint arXiv:2211.05100*, 2022.

[104] R. Schmid, M. Plauth, L. Wenzel, F. Eberhardt, and A. Polze, "Accessible near-storage computing with fpgas," in *EuroSys*, 2020.

[105] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization," in *MICRO*, 2013.

[106] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *MICRO*, 2017.

[107] S. Shi, X. Chu, K. C. Cheung, and S. See, "Understanding Top-k sparsification in distributed deep learning," *arXiv preprint arXiv:1911.08772*, 2019.

[108] 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 preprint arXiv:1909.08053*, 2019.

[109] M. Singh and B. Leonhardi, "Introduction to the IBM Netezza warehouse appliance," in *CASCON*, 2011.

[110] M. Soltaniyeh, V. Lagrange Moutinho Dos Reis, M. Bryson, X. Yao, R. P. Martin, and S. Nagarakatte, "Near-storage processing for solid state drive based recommendation inference with smartssds®," in *ICPE*, 2022.

[111] J. Song, J. Yim, J. Jung, H. Jang, H.-J. Kim, Y. Kim, and J. Lee, "Optimus-CC: Efficient large NLP model training with 3D parallelism aware communication compression," in *ASPLOS*, 2023.

[112] Storage Networking Industry Association, "What is computational storage?" [Online]. Available: https://www.snia.org/sites/default/files/SSSI/Computational_Storage_What_Is_Computational_Storage_final_PDF.pdf

[113] X. Sun, W. Wang, S. Qiu, R. Yang, S. Huang, J. Xu, and Z. Wang, "Stronghold: Fast and affordable billion-scale deep learning model training," in *SC*, 2022.

[114] X. Sun, H. Wan, Q. Li, C.-L. Yang, T.-W. Kuo, and C. J. Xue, "RM-SSD: In-storage computing for large-scale recommendation inference," in *HPCA*, 2022.

[115] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *ICML*, 2019.

[116] H. Tang, S. Gan, A. A. Awan, S. Rajbhandari, C. Li, X. Lian, J. Liu, C. Zhang, and Y. He, "1-bit Adam: Communication efficient large-scale training with Adam's convergence speed," in *ICML*, 2021.

[117] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *FAST*, 2013.

[118] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, 2017.

[119] T. Vogels, S. P. Karimireddy, and M. Jaggi, "PowerSGD: Practical low-rank gradient compression for distributed optimization," in *NeurIPS*, 2019.

[120] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *ICLR*, 2019.

[121] H. Wang, S. Agarwal, and D. Papailiopoulos, "Pufferfish: Communication-efficient models at no extra cost," in *MLSys*, 2021.

[122] J. Wang, L. Guo, and J. Cong, "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA," in *FPGA*, 2021.

[123] R. Weiss, "A technical overview of the oracle exadata database machine and exadata storage server," *Oracle White Paper*, 2012.

[124] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, "RecSSD: near data processing for solid state drive based recommendation inference," in *ASPLOS*, 2021.

[125] W. Xiong, L. Ke, D. Jankov, M. Kounavis, X. Wang, E. Northup, J. Yang, B. Acun, C. Wu, P. P. Tang, G. E. Suh, X. Zhang, and H. S. Lee, "SecNDP: Secure Near-Data Processing with Untrusted Memory," in *HPCA*, 2022.

[126] B. Yang, J. Zhang, J. Li, C. Re, C. Aberger, and C. De Sa, "PipeMare: Asynchronous Pipeline Parallel DNN Training," in *MLSys*, 2021.

[127] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: throughput-oriented programmable processing in memory," in *HPDC*, 2014.

[128] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing communication for pim-based graph processing with efficient data partition," in *HPCA*, 2018.

[129] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep Learning with Elastic Averaging SGD," in *NeurIPS*, 2015.

[130] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable pim-based graph processing," in *MICRO*, 2019.