Synergistic Approach for Systems with AXDIMMs, GPUs, and NVMe Devices

Heehoon Kim
Dept. of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
heehoon@aces.snu.ac.kr

Daeyoung Park
Dept. of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
daeyoung@aces.snu.ac.kr

Jinpyo Kim
Dept. of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
jinpyo@aces.snu.ac.kr

Junsik Shin
Dept. of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
junsik@aces.snu.ac.kr

Jaejin Lee
Dept. of Data Science
Dept. of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
jaejin@snu.ac.kr

Abstract

The existing Deep Learning frameworks, such as PyTorch and TensorFlow, allocate all tensors on the GPU memory. Since state-of-the-art DNN models, such as GPT-3, already have larger tensors than the GPU memory, the GPU memory oversubscription problem occurs. One emerging solution is utilizing the CPU memory or storage devices to free up the GPU memory by swapping. In this paper, we propose a platform with an AXDIMM, a near-memory acceleration memory module developed by Samsung, to tackle the problem with high performance. Our platform consists of a hardware implementation of the Adam optimizer in the AXDIMM and a software runtime to enable the AXDIMM to be used with GPUs and NVMe SSDs. The platform offloads the parameter and optimizer states to the AXDIMM. The overflowed tensors from the AXDIMM are evicted to the NVMe SSDs and restored from them as needed. A PyTorch-compatible library is built on top of the platform, fully exploiting the synergy between the GPU, AXDIMM, and NVMe SSDs to train a large DNN model. The evaluation result with GPT models of various sizes indicates that the AXDIMM outperforms the normal DIMM. Our platform achieves up to 1.63× speedup on the platform with a normal DIMM in training the large DNN models.

1 Introduction

Near-Memory Computing (NMC) is a paradigm that aims to perform computation near the memory[39]. While traditional processors like general-purpose CPUs try to provide higher bandwidth via memory hierarchy, the memory bandwidth becomes a significant performance bottleneck for applications with low locality. We can achieve a higher memory bandwidth and energy efficiency by placing the cores as near as possible.

Among others, an AXDIMM[20] is a DDR4-compatible near-memory acceleration module developed by Samsung. An AXDIMM has an FPGA equipped with ARM cores and multiple ranks of DRAM inside it. While the CPU can access only one rank at a time, the FPGA can access multiple ranks in parallel, resulting in higher effective bandwidth.

Since the invention of Transformer[40], the number of parameters of state-of-the-art DNN models has grown to trillions. While GPUs are de-facto standard accelerators for training DNN models, training such large DNN models on a single GPU is impossible[9, 27, 28]. GPT-3[9], for example, has 175 billion parameters whose size is 700GB in single-precision floating-point format. However, even the state-of-the-art high-end GPUs have less than 100GB of memory. Note that training a DNN model requires more memory than the size of the model parameters because there exist other data, such as activations, gradients, and optimizer states.

By default, the existing Deep Learning (DL) frameworks, such as PyTorch[25] and TensorFlow[6], allocate all tensors of a DNN model in the GPU memory. If the maximum memory footprint of the model exceeds the GPU memory capacity, the training process halts with the out-of-memory error.

A standard solution to such a memory capacity problem is parallelizing the DNN model across multiple GPUs. The most common parallelization techniques are exploiting various data, model, and pipeline parallelism[17, 29, 36].

Another emerging solution is utilizing the main memory or storage devices as a backup space of the GPU memory[30, 33], i.e., offloading some tensors to the host memory or storage devices to free up the GPU memory space until the DNN model reaccesses the tensors. Because only a small portion of the GPU memory is accessed at a time, we can offload the data in the remaining GPU memory space to slower but larger devices until the GPU reaccesses them. In this approach, the

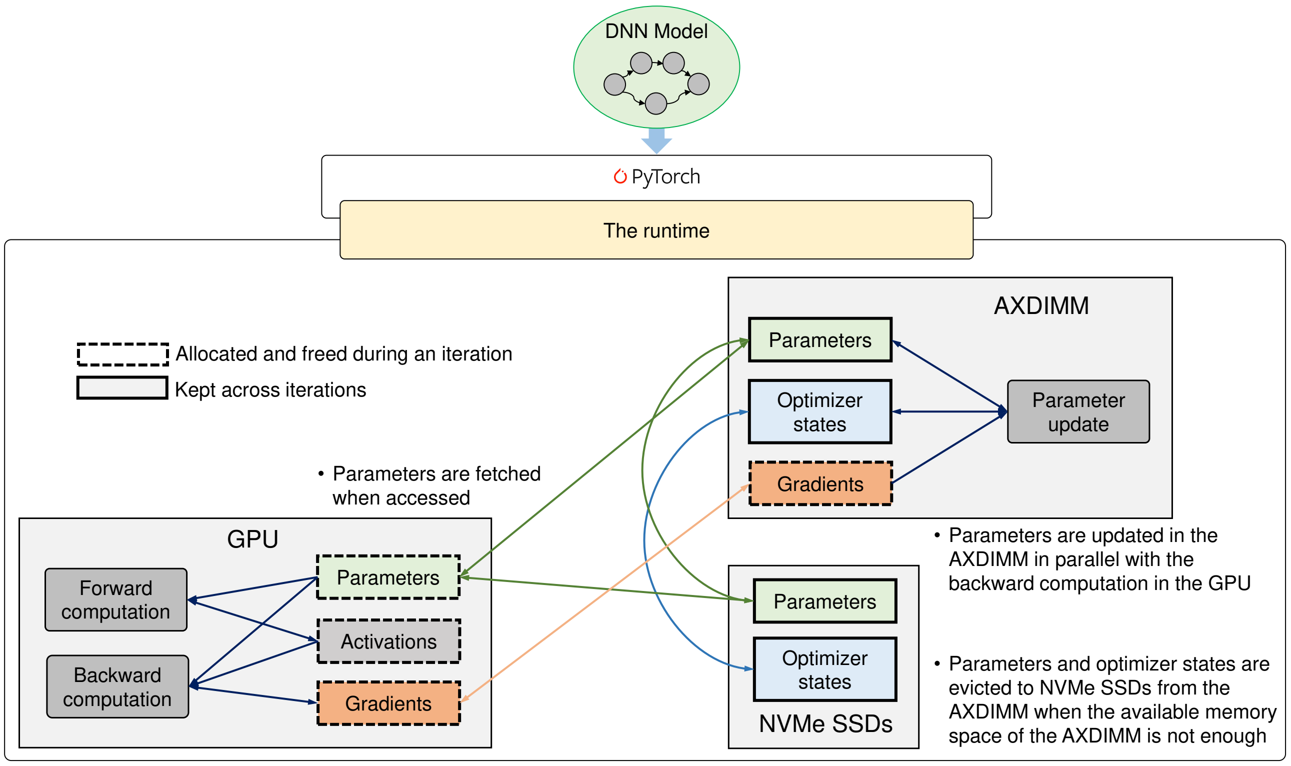

DNN Model
PyTorch
The runtime
Allocated and freed during an iteration
Kept across iterations
• Parameters are fetched when accessed
GPU
Forward computation
Backward computation
Parameters
Activations
Gradients
AXDIMM
Parameters
Optimizer states
Gradients
Parameter update
• Parameters are updated in the AXDIMM in parallel with the backward computation in the GPU
Parameters
Optimizer states
NVMe SSDs
• Parameters and optimizer states are evicted to NVMe SSDs from the AXDIMM when the available memory space of the AXDIMM is not enough

Figure 1. Overall workflow of the proposed platform.

communication between the GPU and the backup storage devices should be minimized to achieve high performance.

For example, ZeRO-Offload[33] shows that placing the model parameters and optimizer states on the host memory and updating them by the CPU achieve the minimum communication volume between the CPU and the GPU. When the CPU computes the optimizer states, the model parameters can be updated on the CPU side instead of transferring optimizer states back to the GPU. However, those computations may be memory-intensive and interfere with the data transfer, resulting in significant performance degradation.

In this paper, we propose a platform that synergistically exploits AXDIMM, a GPU, and NVMe SSDs to train large DNN models that do not fit into the GPU memory. The proposed platform consists of a hardware implementation of the parameter update pass on the AXDIMM, a software runtime to control the AXDIMM and the AXDIMM-GPU or AXDIMM-NVMe SSD communication, and a PyTorch[25]-compatible library to support the training.

Figure 1 shows the overall workflow of the proposed platform. Its primary difference from ZeRO-Offload[33] is that it offloads the DNN parameters and optimizer states to the AXDIMM instead of the host memory. Then, the FPGA in the AXDIMM updates the parameters instead of the CPU. Moreover, the excessive parameters and optimizer states in the AXDIMM are saved in the NVMe SSDs and restored from them when necessary.

Because the FPGA is located close to the DRAM in the AXDIMM, it has a much higher memory bandwidth and updates the parameters faster. It performs the parameter update in parallel with the communication to the GPU as the parameter update does not use the DDR4 link between the CPU and the AXDIMM, resulting in better computation-communication overlapping and shorter training time.

However, it is required to know the time when the parameters are accessed, and the gradients are computed to implement the proposed workflow. To obtain the information, we replace the functions in PyTorch for the layer computation with our implementation of the functions or utilize the hooking mechanism provided by PyTorch to obtain the information.

The major contributions of this paper are summarized as follows:

- We provide an FPGA implementation on the AXDIMM. The kernel in the FPGA implementation is easily replaced with another kernel to accelerate various applications.
- We design and implement a runtime that enables the AXDIMM to be used with the GPU and NVMe SSDs. To the best of our knowledge, our approach is the first

work that exploits the AXDIMM synergistically with the GPU and the NVMe SSDs.
- We propose offloading the parameter update step to the AXDIMM in training large DNN models to improve performance. To support this idea, we provide an easy-to-use PyTorch extension. The proposed technique is orthogonal to the typical parallelization approaches across GPUs. It can be adapted to the parallelization approaches to train much larger DNN models.
- Experimental results show that the AXDIMM is 5.6x faster on the parameter update and 1.26× to 1.63× faster in training a GPT-based DNN model over the normal host memory.

2 Background

In this section, we briefly overview Deep Learning (DL) and describe the input and output for the Adam optimizer[21].

2.1 Deep Learning

DL is a subset of Machine Learning. It trains a Deep Neural Network (DNN) model to make the model learn relationships between the input and output. A DNN is usually modeled by a data-flow graph where nodes are DNN layers and edges represent the data dependences between the layers.

A layer in a DNN model is modeled as a function L as follows:

$$Y = L(X_1, \cdots, X_N). \tag{1}$$

The inputs $X_1, \cdots, X_N$ and the output Y of the layer are called *tensors*, in the sense that they are multi-dimensional arrays. While some tensors (e.g., *activations*) are discarded after being used as an input to a layer, the others (e.g., *model parameters*) are stored and reused throughout the lifetime of the model.

DNNs operate in two phases: *inference* and *training*. In the inference phase, input data are fed into the DNN, and output data are produced, assuming that DNN model parameters have already been trained. The data are propagated through the paths in the data-flow graph of the DNN model. This process is called *forward propagation* or *forward pass*.

The training phase evaluates the DNN model output and modifies the model parameters to make the DNN produce a more accurate output. To evaluate the output, a *loss function* is required. The loss function computes a scalar value, called the *loss* or *error* that becomes smaller as the DNN model output is closer to the expected target.

While minimizing the loss can be performed in many ways, the simplest is Stochastic Gradient Descent (SGD)[35]. SGD optimizes the DNN model by updating its parameters θ as follows:

$$\theta \leftarrow \theta + \gamma * \nabla_\theta l, \tag{2}$$

where $\nabla_\theta l$ is the *gradient* of the loss function with respect to θ, and γ is a hyper-parameter called the *learning rate* that adjusts the amount by which the weights are updated.

To compute the gradients, each layer of the DNN model should compute its input gradients from its output gradients as follows:

$$\nabla_X l = \mathbf{J}_L \nabla_Y l, \tag{3}$$

where $\nabla_X l$ is the input gradient, $\nabla_Y l$ is the output gradient, and $\mathbf{J}_L$ is the Jacobian matrix of L. The gradients are propagated through the DNN backward; hence this is called *backpropagation* or *backward pass*. In summary, the training phase of a DNN model consists of the forward pass, backward pass, and weight updates.

```
**Input:** $\nabla_\theta(\theta_{t-1}), \theta_{t-1}, m_{t-1}, v_{t-1}, \gamma, \beta_1, \beta_2, \lambda, t$
**Output:** $\theta_t, m_t, v_t$

$g_t \leftarrow \nabla_\theta(\theta_{t-1}) + \lambda\theta_{t-1}$
$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) {g_t}^2$
$\widehat{m_t} \leftarrow m_t / (1 - {\beta_1}^t)$
$\widehat{v_t} \leftarrow v_t / (1 - {\beta_2}^t)$
$\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$
```

Algorithm 1: The Adam algorithm at a training step t.

2.2 Adam Optimizer

Due to the slow convergence rate of SGD, other stochastic gradient-based optimizers have been invented. Adam [21] is the most popular algorithm and widely used to train large DNN models, such as BERT[10] and GPT[28].

SGD is stateless because it does not require any information other than the gradients to update the DNN model parameters. However, the Adam optimizer should maintain two additional values m and v per parameter, which are the moving average of its gradients and the square of its gradient, respectively. The tensors for m and v represent the current state of the optimizer and are called *optimizer states*.

In the aspect of memory accesses, Adam needs four loads per parameter for the parameter θ_{t-1} itself, a gradient $\nabla_\theta(\theta_{t-1})$, and optimizer states m_{t-1} and v_{t-1}. In addition, it needs three stores per parameter for the updated parameter θ_t and updated optimizer states m_t and v_t. Thus, the Adam optimizer require a high memory bandwidth.

3 Implementations in the AXDIMM

AXDIMM[20] is a near-memory acceleration module developed by Samsung. AXDIMM has the DDR4 DIMM form factor and can be placed in the memory slot of a mainboard. With the reference design loaded in the AXDIMM's FPGA, the host CPU sees the AXDIMM as a 32GB dual-rank DDR4-800 DIMM.

Table 1. AXDIMM Specification

FPGA	Xilinx Zynq Ultrascale+ (xczu19eg-ffvb1517-2LV-e)
DRAM	10 x K4ABG165WA-MCTD (32Gb, DDR4, Up to 2666 Mbps)
Interfaces	DDR4 DIMM, JTAG

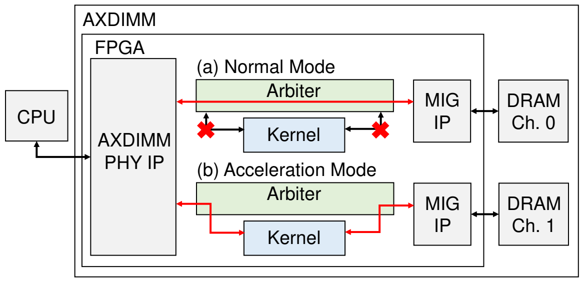


Figure 2. The proposed architecture in the AXDIMM.

3.1 Proposed Architecture in the AXDIMM

Figure 2 shows the proposed architecture in the AXDIMM. The AXDIMM reference design consists of an AXDIMM PHY IP and two Memory Interface Generator(MIG) IPs. The AXDIMM PHY IP deserializes the 400MHz DDR4 interface from the host into the 200MHz interface with 512-bit data width so that a computation kernel can be implemented in the FPGA fabrics of the AXDIMM with a reasonable frequency (The kernel in Figure 2). The MIG IP converts the frequency to 800MHz and connects to the internal DRAM chips. The DRAM chips are grouped into two independent 16GB channels. The AXDIMM PHY IP maps the ranks seen by the host to the channels.

FCFS arbitration is problematic. Because both the host and the kernel in the FPGA access the DRAM channels, they require arbitration. However, the typical FCFS (First Come, First Serve) arbitration of DDR4 transactions is problematic because of the following two reasons:

- The host cannot change the read latency after the system boots.
- The DRAM channels have a state for each bank.

Consider the following scenario. Suppose that the host activates a row R_0 of a bank and requests a read for the row, and then the kernel accesses another row R_1 of the same bank. The arbiter precharges R_0 after serving the host's read request and activate R_1 to serve the kernel request. Sometime later, the host requests a read for R_0 again, thinking that R_0 is still active. However, R_0 is not active at this point because of the kernel request. Thus, the host has to wait until the precharging and activation finish, in addition to the read latency. This worst-case scenario implies that the host's read latency should be set to a somewhat high value, resulting in significantly reduced memory throughput between the host and the AXDIMM module.

3.2 Arbiter Implementation in the FPGA

Thus, instead of having an FCFS-based arbiter, we implement a simple arbiter with two exclusive modes: *normal* and *acceleration*. In the normal mode, the host exclusively uses the internal DRAM channels in the AXDIMM. All requests from the kernel are blocked. In contrast, all requests from the host are routed to the kernel in the acceleration mode. The kernel exclusively accesses the internal DRAM channels.

To poll the current arbiter mode and switch it to the other, requests to a specific address range are always routed to the arbiter's control and status registers (CSRs). Instead of having the kernel generate DDR4 transactions, the arbiter and the kernel connect through AXI4[1] interfaces for easy kernel implementation. The arbiter converts the host's DDR4 transaction to an AXI4 transaction with a fixed read latency and the kernel's AXI4 transaction to a DDR4 transaction.

Exploiting channel parallelism. Each DRAM channel has its own arbiter and computation kernel. The arbiters and kernels are controlled independently. The CPU can keep reading from or writing to the internal DRAM by setting one of the channels to the normal mode, even if the kernel uses the other channel.

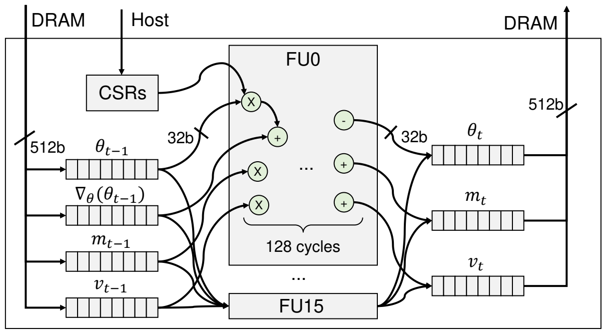


Figure 3. Adam kernel architecture.

3.3 Adam Kernel Implementation in the FPGA

Figure 3 shows the architecture of our Adam kernel implementation in the FPGA. There are a total of sixteen functional units (FU0, FU1, ..., FU15) for the Adam algorithm with a single-precision floating-point format (FP32). Each unit consists of 31 DSPs and is fully pipelined with a 128-cycle latency. Because the data width of the interface to the DRAM is 512 bits (= 32bits × 16), we implement 16 FUs to process the inputs to the kernel in parallel. We use a FIFO queue implemented in the FPGA on-chip memory to handle each of the four inputs: θ_{t-1}, $\nabla_\theta(\theta_{t-1})$, m_{t-1}, v_{t-1} and three outputs: θ_t, m_t, v_t of the kernel.

Hereafter, a block refers to a contiguous section of memory loaded or stored by the kernel each time. The block size is a design parameter. The kernel keeps loading the blocks of θ_{t-1}, $\nabla_\theta(\theta_{t-1})$, m_{t-1}, and v_{t-1} from the internal DRAM in a round-robin manner and enqueues them into the corresponding queues. Results (θ_t, m_t, and v_t) of the kernel are also enqueued to the corresponding output queues, and the kernel writes them back to the DRAM later.

The kernel has CSRs that have execution status and kernel parameters, such as the addresses of the inputs and outputs, constants used by the Adam algorithm, and the number of parameters to update. After setting the arbiter to the acceleration mode, the host writes kernel parameters and signals to the kernel to start. The host waits for the kernel to finish execution by polling the status.

3.4 Effect of the Block Size

The size of the block determines the tradeoff between performance and resource usage. Small block sizes may result in precharging and activating many DRAM banks depending on the address of the tensors, while large block sizes consume more on-chip memory spaces. We set the block size to 16KB, which spans the two rows of all DRAM chips. Because interleaved accesses to two different bank groups are faster than consecutive accesses to a single bank group, reading the entire contents of two rows from different bank groups may improve performance.

Double buffering. To hide computation latency, we also apply double buffering. Instead of waiting for the computation result from one block, the kernel loads another block during the computation simultaneously. The separate input and output queues in the kernel architecture make it possible.

4 Runtime for the AXDIMM

In this section, we describe the design and implementation of the proposed runtime system for the AXDIMM.

The memory space of the AXDIMM is reserved when the system boots. However, the system should map it to the virtual address space for an application. To do so, we implement a Linux kernel module that creates a character device that supports *mmap* call. During initialization, the runtime calls *mmap* on the character device. The rest of the runtime is implemented in the user space.

4.1 Cache Coherence

While address mapping is easy, we must carefully choose the mapped pages' cache mode. Since the DDR4 protocol opted for the cache line size that is 64 bytes in the target system, even a four-byte read results in a 64-byte read to the AXDIMM if a page is set non-cacheable, significantly reducing the effective memory throughput. On the other hand, the host may read inconsistent values if the cache is enabled because there is no way for the AXDIMM to inform the cache that the value on its internal DRAM has changed. Moreover, writes by the host may end up in the cache, making it impossible to send a signal, such as a kernel launch, to the AXDIMM.

Guaranteeing cache coherence. Thus, applications should always access the AXDIMM through read and write functions provided by the runtime for correct execution that guarantees cache coherence. The runtime first enables the host-side write-back cache for the memory space of the AXDIMM to improve performance. For read transactions by the host, the runtime issues a cache line flush instruction (*CLFLUSH*) followed by two 32-byte load instructions (*VMOVDQA*) to ensure that the value is always served by the AXDIMM, not by the cache. Note that the size of a cache line is 64 bytes. For write transactions, the runtime uses two 32-byte store instructions with a non-temporal memory hint (*VMOVNTDQ*). It makes the write go into a specialized write-combining buffer instead of the cache. Thus, the two stores are likely to be combined into a 64-byte data item and will reach the AXDIMM as soon as possible. A vector instruction set, such as AVX512, will perform even better, but our target system does not support it.

4.2 Runtime API Functions

The runtime also contains helper functions to use the AXDIMM, such as managing DRAM spaces in the AXDIMM, launching a kernel, switching the arbiter mode, and copying the memory contents to other devices. Table 2 shows some examples of the runtime API functions. The platform runtime is implemented as a Linux shared library (*libaxdimm.so*) so that an application can be compiled and linked with it.

4.3 Pinning Pages

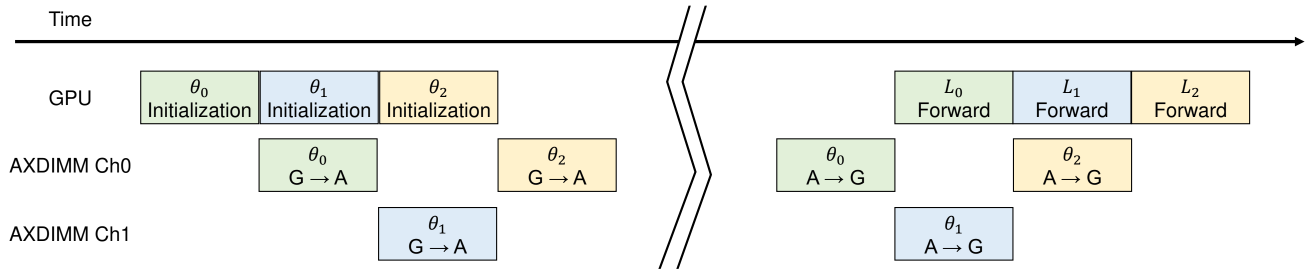
For high CPU-GPU communication bandwidth, a range of pages to be transferred should be *pinned* before the communication. Pinning is the process of marking the pages in the address space as non-pageable. Non-pageable pages are guaranteed to have the same physical addresses. Thus, the GPU can directly access only pinned pages without CPU intervention.

Technically, the virtual address range mapped to the memory space of the AXDIMM can inherently be considered pinned because their corresponding physical addresses inside the AXDIMM never change. However, the naïve implementation of Linux *mmap* marks the virtual address range of the AXDIMM physical address space as that of an IO device. In Linux, the I/O-mapped address range is treated differently from a physical address range in many ways. Especially, the Linux kernel does not allow to pin the I/O-mapped pages. To circumvent this, we mark page table entries for the pages in the AXDIMM physical address space to be pinned as if they resided in a *hot-plugged* physical memory unit[4]. Moreover,

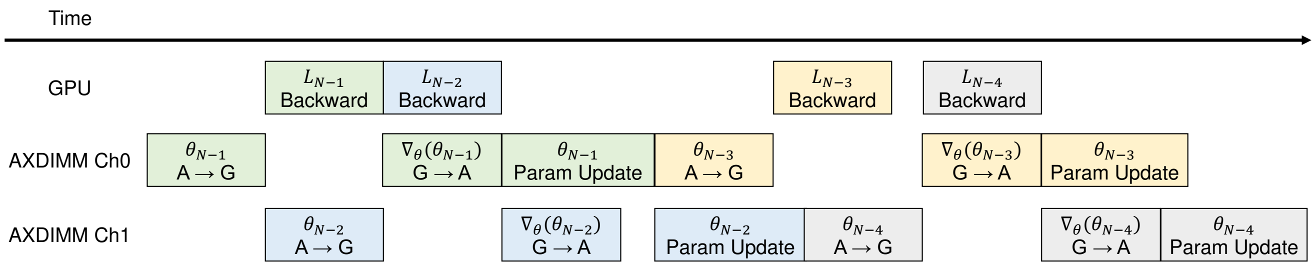
Table 2. Examples of runtime API functions.

Type	Functions
Resource Management	`Axdimm(device);` Initialize the AXDIMM runtime with the AXDIMM module pointed by `device` (e.g., `/dev/axdimm0`).
	`RegisterStorage(device);` Register a new NVMe SSD pointed by `device`, for use by the runtime (e.g., `/dev/nvme0n1`).
Memory Management[†]	`MallocX(nbytes, channel);` Allocate `nbytes` on the device `X` and returns a pointer to the allocated space.
	`FreeX(ptr);` Free the memory allocation on the device `X` pointed by `ptr`.
	`MemcpyX2Y(dst, src, nbytes);` Copy `nbytes` from the address `src` of the device `X` to the address `dst` of the device `Y`.
AXDIMM Management	`SetArbiterAcc(channel);` Set the arbiter of the specified channel to the acceleration mode.
	`RunAdamOnAxdimmAsync(p, m, v, g, nparams, ..., t, channel);` Run the Adam computation kernel asynchronously on the specified channel.

[†] The device X and Y can be the host CPU, the AXDIMM, the GPU, or an NVMe SSD.



(a) Initialization and forward propagation.


Time
GPU
AXDIMM Ch0
AXDIMM Ch1
L_{N-1} Backward
L_{N-2} Backward
L_{N-3} Backward
L_{N-4} Backward
θ_{N-1} A → G
$\nabla_\theta(\theta_{N-1})$ G → A
θ_{N-1} Param Update
θ_{N-3} A → G
$\nabla_\theta(\theta_{N-3})$ G → A
θ_{N-3} Param Update
θ_{N-2} A → G
$\nabla_\theta(\theta_{N-2})$ G → A
θ_{N-2} Param Update
θ_{N-4} A → G
$\nabla_\theta(\theta_{N-4})$ G → A
θ_{N-4} Param Update

(b) Backpropagation and parameter updates.

Figure 4. A timeline of the initialization and a single training iteration with our framework. G and A denotes the GPU and the AXDIMM, respectively. Blocks with the same color belong to the same layer, thus they should be accessed sequentially.

we mark the AXDIMM pages as reserved ones to prevent the AXDIMM from being allocated for Linux kernel use.

Such an approach allows us to manage the AXDIMM as a typical DIMM and to use well-known communication runtime APIs.

4.4 Data Transfers between the GPU and NVMe SSDs

In a few cases, it is beneficial for GPU memory contents to be directly offloaded to storage devices. However, a naïve GPU-storage communication is staged through a bounce buffer in the main memory. For higher communication bandwidth, we exploit *GPUDirect Storage*[5] for the GPU to access the NVMe SSDs directly. GPUDirect Storage provides direct memory accesses (DMAs) between the GPU and the storage device on the same PCIe root. The DMA improves communication bandwidth and prevents wasting the DRAM bandwidth due to the staged copy. To support GPUDirect Storage, we implement I/O operations using the cuFile APIs[3].

5 DL Framework for the AXDIMM

Tensors in training a DL model can be classified into four groups: parameters, activations, gradients, and optimizer states. Widely used DL frameworks, such as PyTorch and TensorFlow, allocate all the tensors in the GPU memory in the training process. If the total size of the tensors exceeds the GPU memory capacity, the training process causes an Out-Of-Memory (OOM) error.

5.1 Proposed DL Framework

To overcome this problem, we provide a library compatible with the existing DL framework PyTorch[25], one of the most popular DL frameworks, to support training large DNN models with the AXDIMM. The library runs on top of PyTorch, and no source code modification of PyTorch itself is required. The library internally uses the runtime API functions described in Section 4.2.

Before the training starts, our library replaces PyTorch layers with our own layer implementations. The replacing layers have essentially the same functionalities as the replaced ones but have some extra functionalities. For example, they check if the input is a parameter tensor and prefetch the tensor into the GPU memory before computation starts.

PyTorch also provides a mechanism to register hooks that are called when the tensors are accessed during the backpropagation phase or after the gradient computation has finished. We exploit the hooking mechanism to prefetch tensors, send the gradients to the AXDIMM, and execute the parameter update kernel.

5.2 Tensor Offloading and Prefetching

We observe that not all tensors are accessed simultaneously during the training process. For example, a parameter tensor is accessed only by the layers that own the tensor. Optimizer states, on the other hand, are accessed only in the parameter update step during the training process. We can train large DNN models that do not fit into the GPU memory without OOM errors by offloading tensors that are not accessed currently to a location other than the GPU.

Initializing a large DNN model. Figure 4 shows the timeline of the initialization and a single training iteration of a DNN model on our framework. The parameter tensors are allocated and randomly initialized on the GPU during the DNN model initialization. For a large DNN model, the total size of the parameter tensors alone may already exceed the GPU memory capacity, resulting in an OOM error even before the training phase starts.

Thus, we offload each tensor to the AXDIMM as soon as it has been initialized. Moreover, the corresponding optimizer states are also offloaded to the AXDIMM unit. For optimization, we allocate the parameter tensors on the channels in the AXDIMM in a round-robin manner. Note that even though each channel has independent internal DRAM, the data transfer between the host and the AXDIMM should be sequential because they share the same DDR4 link.

Tensor prefetching. During the forward propagation phase, the GPU cannot start the computation until the parameters of the current layer are copied back. Thus, these tensors should be prefetched before the GPU accesses them to prevent underutilization of the GPU. PyTorch manages its own CUDA stream[2] and enqueues computation into the stream. A CUDA stream is a FIFO queue bound to a thread that keeps processing requests in the queue. Since the stream is usually full of computations, a computation request enqueued by PyTorch will be served later. Thus, if our platform starts to fetch the associated tensors when PyTorch enqueues a computation, it has a net effect of prefetching tensors. Since our platform runtime enqueues its prefetch requests into a separate CUDA stream dedicated to data transfer, the tensors are fetched in parallel with the computation of the previous layer, maximizing computation-communication overlap.

Gradient offloading. The backward computation of each layer produces a gradient tensor passed to the previous layer, or it is the gradient of the parameter tensor. When it is the gradient of the parameter tensor, it is transferred to the AXDIMM, and its memory space in the GPU is freed immediately.

Advantages over the existing DL frameworks. In the existing DL frameworks, such as PyTorch and TensorFlow [6, 25], the parameter update step is usually done after the backpropagation step has completed because both steps involve computations on the GPU and cannot execute in parallel. In contrast, our AXDIMM architecture has two independent channels. Thus, the parameter update kernel can execute by accessing one channel while it uses the other channel to transfer data from the GPU. Thus, our framework overlaps the backward computation on the GPU, the data transfer between the GPU and the AXDIMM, and the parameter update step to hide the data transfer overhead.

5.3 Exploiting NVMe SSDs

For large models that do not even fit into the memory space of the AXDIMM, the NVMe SSDs are used as secondary backup storage. At the initialization, the runtime sends the remaining parameters to the NVMe SSDs after the memory space of the AXDIMM is fully occupied.

Whenever the GPU requires the parameters saved in the NVMe SSDs, the runtime initiates a set of DMA operations to transfer them directly from the NVMe SSDs to the GPU. After the computation of the gradients completes, they and their corresponding parameters and optimizer states need to be gathered in the AXDIMM to update parameters. In this case, the runtime evicts the tensors in the memory space of the AXDIMM until enough space is secured. The tensors that have already been used in the parameter update step

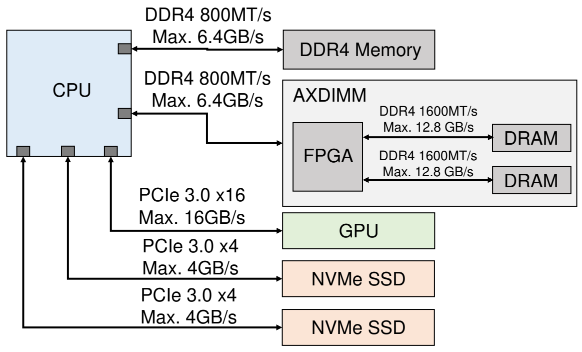


Figure 5. Target system components and the bandwidth of each link.

of the current iteration have a higher priority on eviction because they will not be used until the next iteration.

6 Evaluation

In this section, we evaluate our AXDIMM platform.

6.1 System Configuration and DNN Models Used

System configuration. The platform consists of a single Intel multicore CPU, 32GB DDR4 DIMM for the host memory, AXDIMM, and NVIDIA GPU. In addition, it has two 2TB NVMe SSDs. The system configuration is summarized in Table 3 and shown in Figure 5.

DNN models used. To evaluate the proposed AXDIMM platform, we use the Megatron[36] implementation of GPT[9, 28]. We modify it to run on a single GPU. GPT is a series of language models that consist mostly of the decoder layers of the Transformer model[40]. The number of the decoder layers can be adjusted to obtain models with a different number of parameters. Table 4 shows the configurations of different GPT models used for the evaluation. It also shows the hyperparameters of each model in the training phase. n_{layers}, d_{model}, n_{heads}, d_{head}, and n_{ctx} are the number of decoder layers, the number of hidden units in the linear layers, the number of heads in the multi-attention layer, the size of each attention head, and the size of the context window in the number of words, respectively. The model configurations are slightly different from the original GPT because of the limitation of Megatron. n_{params}, m_{params}, m_{os} are the total number of parameters in the model, the total size of parameters, and the total size of optimizer states, respectively. Note that we choose all the configurations large enough to cause the OOM error if they are trained with the original PyTorch.

Table 3. Target system configuration.

Mainboard	Intel DBS2600CW2
CPU	Intel Xeon E5 2698 v4
Memory	DDR4 32GB 800MT/s, AXDIMM (See Table 1)
GPU	NVIDIA Tesla V100 32GB
NVMe	2 x Seagate FireCuda 520 SSD

6.2 Performance of the Adam Kernel

We first evaluate the performance of our Adam kernel implementation in the AXDIMM. In this experiment, we assume that the inputs to the kernel already reside in the AXDIMM internal DRAM. We measure the time taken until the Adam kernel finishes processing all the inputs and writes the result back to the internal DRAM.

We compare the AXDIMM Adam kernel to the manually optimized Adam kernel with SIMD instructions on the CPU (say CPU baseline). The inputs reside in the host memory for the CPU baseline, and the outputs are written back to the host memory. We execute the CPU baseline with a single core because we observe no performance improvement when we exploit multiple cores. The reason is that Adam is memory bound.

We measure the throughput in the units of the number of processed parameters (millions) per second (MP/s). Adam needs four loads and three stores to update each parameter. They are summed to `(4 + 3) * (the size of one parameter)` = 28 bytes. Thus, the theoretical maximum throughput is the theoretical memory bandwidth divided by 28 bytes.

The experimental result in Table 5 shows that the AXDIMM Adam kernel has 5.6X higher throughput over the CPU baseline. There are two reasons for the significant performance difference. One is that the AXDIMM has direct cycle-accurate control over its DRAM. The AXDIMM Adam kernel achieves over 86% of the theoretical maximum throughput. In contrast, the CPU base accesses the data through its memory hierarchy, limiting the throughput when many misses occur. Thus, the CPU base achieves only 61% of the theoretical maximum throughput. The other reason is that the AXDIMM has two independent DRAM channels, each of which runs at twice the frequency of the host memory. It results in another four-fold throughput increase.

6.3 Training the GPT Models

Next, we evaluate the performance of our AXDIMM platform to train the GPT models in Table 4. We compare our AXDIMM platform with a baseline platform. The baseline platform uses the same AXDIMM runtime and PyTorch as those used in the AXDIMM platform. Instead of offloading tensors to the AXDIMM, the baseline offloads them to the host memory, and the CPU updates parameters, not the AXDIMM. If the host memory overflows in the baseline platform, it saves the evicted tensors in the NVMe SSDs. These evicted tensors should be fetched from the NVMe SSDs in the next iteration. The same thing is true for the AXDIMM memory in the AXDIMM platform.

Table 4. Configurations of the GPT models and Hyperparameters used in the training.

Name	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	n_{ctx}	n_{params}	$m_{params}(GB)$	$m_{os}(GB)$
GPT-3 Large	24	1,536	16	96	3	1,152	760M	3.0	6.1
GPT-3 XL	24	2,064[†]	24	86[†]	2	1,024	1.3B	5.2	10.4
GPT-3 2.7B	32	2,560	32	80	1	1,024	2.7B	10.8	20.2
GPT-3 6.7B	32	4,096	32	128	1	1,024	6.7B	26.8	53.6
GPT-3 13B	40	5,120[†]	40	128	1	512	13.0B	52.0	104.0

[†] We use a different value from the original because Megatron supports only models with $d_{model} = n_{heads} * d_{head}$.

Table 5. Performance (throughput = million parameters per second) comparison for Adam between the AXDIMM and the CPU.

Device	Theoretical	Measured	Efficiency
CPU baseline	228.57 MP/s	140.32 MP/s	61.4%
AXDIMM	914.29 MP/s	789.38 MP/s	86.3%

Since the operating system and PyTorch are running on the host memory, we cannot assign the entire 32GB of the host memory space to the training process. We let the baseline utilizes 16GB of the host memory in all experiments. Thus, we limit the AXDIMM to use only 16GB (8GB per channel) of its DRAM for a fair comparison. The baseline uses a single dedicated CPU core for Adam, while PyTorch and the AXDIMM runtime utilize other cores.

In both platforms, the GPU performs forward and backward computations. Thus, the amount of memory and storage space consumed by both platforms will be the same. By comparing them under this setup, we can see the net effect of the AXDIMM on performance.

We measure the time and the peak NVMe storage usage for each training iteration of the GPT models in Table 4. Table 6 shows the experimental result. It shows the breakdown of a single training iteration into the forward and backward passes. The forward pass shows almost no performance difference because both platforms have exactly the same tensor access patterns and locations.

In the backpropagation pass, however, the AXDIMM platform outperforms the baseline up to 1.74×. The main reason is that, unlike the CPU, the AXDIMM fully overlaps the parameter update with the data transfer. As the model becomes large, the data transfers between the AXDIMM (or the host memory in the baseline) and the NVMe SSDs become a more dominant factor in performance; thus, the speedup gradually decreases to 1.28× from 1.63×.

The result indicates that the AXDIMM outperforms the normal DIMM in training large DNN models. It is a promising application area of the AXDIMM.

7 Related Work

In this section, we briefly describe the related work to the proposed approach.

7.1 Near-Memory Computing and AXDIMM

The *memory wall*[14], is one of the long-lasting and still important problems, not yet solved, in modern computer architectures even though it has been quite mitigated by the saturation of the processor frequency due to the power wall[14]. Main memory accesses are much slower than the CPU computation, so the data transfer between the CPU and the main memory becomes the performance bottleneck.

Near-Memory computing (NMC) is an architectural model that combines DRAM and processor logic on a single die. The processors near the memory access data at a low cost and mitigate the memory wall problem especially for data-intensive tasks[13, 19, 22, 37, 43].

There are three types of NMC implementations[37]: *CPU-based*, *ASIC-based*, and *reconfigurable*. The CPU-based implementation integrates a general-purpose CPU and DRAM together[11]. It is the most flexible but requires significant hardware resources and consumes considerable energy. The ASIC-based implementation integrates a special-purpose ASIC and DRAM together[41]. It saves hardware resources and energy using the ASIC but is restricted to specific applications. Finally, the reconfigurable implementation uses reconfigurable processors[18], such as FPGAs and CGRAs[12].

An AXDIMM developed by Samsung is an FPGA-based reconfigurable NMC device that integrates reconfigurable processors and DRAM together[20]. It exploits both the flexibility of the CPU-based NMC implementations[11] and the performance of the ASIC-based NMC implementations[41].

Ke et al.[20] offload the memory-intensive embedding layers of the Deep Learning Recommendation Model (DLRM)[24] to an AXDIMM, which significantly improves the inference throughput on the CPU. Their evaluation shows a significant improvement in the latency, energy savings, and throughput over a regular DIMM. Unlike Ke et al.[20], we exploit an AXDIMM for the Adam optimizer, which is a memory-intensive task and one of the most expensive components in training DNN models.

Programming models. There are two types of programming models for NMC devices. One is to use the near-memory processor as an accelerator or helper processor for the specific tasks offloaded from the host. For example, Solihin et al.[38] propose a User-Level Memory Thread (ULMT) running on a helper processor near the main memory for

Table 6. Performance (execution time in seconds) comparison for training the GPT models between the AXDIMM and the baseline.

Model	Baseline			AXDIMM			Speedup	Speedup	Peak NVMe
	Forward	Backward	Total	Forward	Backward	Total	(Backward)	(Total)	Usage (GB)
GPT-3 Large	0.78	8.01	8.79	0.80	4.60	5.40	1.74	1.63	0.9
GPT-3 XL	2.12	19.31	21.42	2.14	13.28	15.43	1.45	1.39	9.8
GPT-3 2.7B	4.78	43.09	47.87	4.79	30.45	35.23	1.42	1.36	26.1
GPT-3 6.7B	13.12	115.17	128.29	13.55	88.17	101.72	1.31	1.26	78.4
GPT-3 13B	25.24	221.62	246.86	26.21	166.25	192.45	1.33	1.28	153.1

correlation prefetching. Their proposal prefetches data from the main memory based on the correlations between memory accesses determined by the user-level thread running on the helper processor. The other is explicitly separating the host system and the NMC system and connecting them using a network interface. Alian et al.[7] propose the Memory Channel Network (MCN) architecture. Their approach modifies the Linux device driver and mimics the message passing interface (MPI) over Ethernet between the host and NMC systems. Our platform uses the AXDIMM as an accelerator for the Adam optimizer used in DNN models.

7.2 Training Large DNN Models

Parallelization. A common approach to training a large DNN model by overcoming the GPU memory capacity problem is to parallelize the model across multiple GPUs using various parallelization strategies, such as data, model, and pipeline parallelism[17, 29, 31, 36]. Among others, ZeRO[29] proposes a new type of data parallelism that evenly distributes the tensors to all the GPUs only at the cost of 1.5× communication. These approaches still allocate all tensors in the GPU memory. Thus the model size is eventually bounded by the sum of the memory capacities of all the GPUs.

Swapping tensors. Another solution is to use the main memory or storage devices to store tensors temporarily using the swapping mechanism. It swaps in/out GPU memory objects to the CPU memory or NVMe SSDs[8, 15, 16, 23, 26, 29–34, 42].

Among others, ZeRO-Offload[33] searches for an optimal offloading strategy for training a large DNN model with the help of the host memory. ZeRO-Offload proposes the following:

- Offloading the parameters and the optimizer state to the CPU memory.
- Executing the forward propagation and the backpropagation on the GPU.
- Updating the parameters on the CPU can achieve the minimum communication volume between the GPU and the CPU.

ZeRO-Infinity[30] is an extension to ZeRO's data parallelism. If the GPU memory overflows, it offloads the parameter, gradient, and optimizer-state tensors to the CPU memory. Then, it further offloads those tensors to the NVMe SSDs when the CPU memory becomes full.

8 Conclusions

This paper introduces an AXDIMM platform specialized for training large DNN models. Specifically, it solves the GPU memory oversubscription problem caused by the large DNN models with high performance. The target platform consists of a CPU, an AXDIMM, a GPU, and two NVMe SSDs. It also has a normal DIMM. Among the tensors used in the DNN training with the GPU, the parameters and optimizer states are offloaded to the AXDIMM. The Adam optimizer implemented in the FPGA of the AXDIMM updates the parameters. The parameters and the optimizer states are evicted to the NVMe SSDs and restored from them as needed. In addition, the parameters are prefetched to the GPU before they are accessed.

We build a platform runtime that enables the AXDIMM to be used with the GPU and NVMe SSDs. We also provide a PyTorch-compatible library on top of the runtime to facilitate training the real-world DNN models. The evaluation result shows that the parameter update on the AXDIMM outperforms the CPU baseline by 5.6×. In training large GPT models with various sizes, the AXDIMM platform achieves speedups from 1.26× to 1.63× over the baseline that uses the normal DIMM. The proposed technique is orthogonal to the parallelization of DNN models, so it can be applied to large-scale distributed training to break the limit on the DNN model size.

References

[1] 2013. *AMBA® AXI™ and ACE™ Protocol Specification.* https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification

[2] 2022. *CUDA Runtime API.* https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[3] 2022. *cuFile API Reference Guide.* https://docs.nvidia.com/gpudirect-storage/api-reference-guide/index.html

[4] 2022. *Memory Hot(Un)Plug.* https://www.kernel.org/doc/html/latest/admin-guide/mm/memory-hotplug.html

[5] 2022. *NVIDIA Magnum IO GPUDirect Storage Overview Guide.* https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16).* 265–283.

[7] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Kai Wang, Thomas Roewer, Adam McPadden, Oliver O'Halloran, Deming Chen, Jinjun Xiong, et al. 2018. Application-transparent near-memory processing architecture with memory channel network. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 802–814.

[8] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2021. FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21).* USENIX Association, 387–401.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. 2017. The mondrian data engine. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 639–651.

[12] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 283–295.

[13] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. 2019. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development* 63, 6 (2019), 3–1.

[14] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach.* Elsevier.

[15] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20).* Association for Computing Machinery, New York, NY, USA, 875–890.

[16] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20).* Association for Computing Machinery, New York, NY, USA, 1341–1355.

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[18] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent distributed storage. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1202–1213.

[19] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. 2020. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 790–803.

[20] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, et al. 2021. Near-memory processing in action: Accelerating personalized recommendation with AxDIMM. *IEEE Micro* 42, 1 (2021), 116–127.

[21] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[22] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 740–753.

[23] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. 2018. TFLMS: Large Model Support in TensorFlow by Graph Rewriting. *ArXiv* abs/1807.02037 (2018).

[24] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[26] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-Based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20).* Association for Computing Machinery, New York, NY, USA, 891–905.

[27] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

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

[29] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–16.

[30] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–14.

[31] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining (KDD '20).* Association for Computing Machinery, New York, NY, USA, 3505–3506.

[32] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 598–611.

[33] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[34] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Article 18, 13 pages.

[35] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[36] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[37] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. 2018. A review of near-memory computing architectures: Opportunities and challenges. In *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 608–617.

[38] Yan Solihin, Jaejin Lee, and Josep Torrellas. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 171–182.

[39] Harold S Stone. 1970. A logic-in-memory computer. *IEEE Trans. Comput.* 100, 1 (1970), 73–78.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[41] Erik Vermij, Leandro Fiorin, Christoph Hagleitner, and Koen Bertels. 2017. Sorting big data on heterogeneous near-data processing systems. In *Proceedings of the Computing Frontiers Conference*. 349–354.

[42] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 41–53.

[43] Salessawi Ferede Yitbarek, Tao Yang, Reetuparna Das, and Todd Austin. 2016. Exploring specialized near-memory processing for data intensive operations. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1449–1452.