FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks

Jonghyun Bae, *Seoul National University;* Jongsung Lee, *Seoul National University and Samsung Electronics;* Yunho Jin and Sam Son, *Seoul National University;* Shine Kim, *Seoul National University and Samsung Electronics;* Hakbeom Jang, *Samsung Electronics;* Tae Jun Ham and Jae W. Lee, *Seoul National University*

https://www.usenix.org/conference/fast21/presentation/bae

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings of the 19th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks

Jonghyun Bae[†] Jongsung Lee[†‡] Yunho Jin[†] Sam Son[†] Shine Kim[†‡] Hakbeom Jang[‡]
Tae Jun Ham[†] Jae W. Lee[†]
[†]*Seoul National University* [‡]*Samsung Electronics*

Abstract

Deep neural networks (DNNs) are widely used in various AI application domains such as computer vision, natural language processing, autonomous driving, and bioinformatics. As DNNs continue to get wider and deeper to improve accuracy, the limited DRAM capacity of a training platform like GPU often becomes the limiting factor on the size of DNNs and batch size—called *memory capacity wall*. Since increasing the batch size is a popular technique to improve hardware utilization, this can yield a suboptimal training throughput. Recent proposals address this problem by offloading some of the intermediate data (e.g., feature maps) to the host memory. However, they fail to provide robust performance as the training process on a GPU contends with applications running on a CPU for memory bandwidth and capacity. Thus, we propose FlashNeuron, the *first* DNN training system using an NVMe SSD as a backing store. To fully utilize the limited SSD write bandwidth, FlashNeuron introduces an offloading scheduler, which selectively offloads a set of intermediate data to the SSD in a compressed format without increasing DNN evaluation time. FlashNeuron causes minimal interference to CPU processes as the GPU and the SSD directly communicate for data transfers. Our evaluation of FlashNeuron with four state-of-the-art DNNs shows that FlashNeuron can increase the batch size by a factor of 12.4× to 14.0× over the maximum allowable batch size on NVIDIA Tesla V100 GPU with 16GB DRAM. By employing a larger batch size, FlashNeuron also improves the training throughput by up to 37.8% (with an average of 30.3%) over the baseline using GPU memory only, while minimally disturbing applications running on CPU.

1 Introduction

Deep neural networks (DNNs) are the key enabler of emerging AI-based applications and services such as computer vision [19, 22, 38, 53, 54], natural language processing [2, 11, 13, 51, 67], and bioinformatics [46, 73]. With a relentless pursuit of higher accuracy, DNNs have become wider and deeper to increase network size [65]. It is because even a 1% accuracy loss (or gain) potentially affects the experience of millions of users if the AI application serves a billion of people [47].

DNNs must be *trained* before deployment to find optimal network parameters that minimize the error rate. Stochastic Gradient Descent (SGD) is the dominant algorithm used for DNN training [15]. In SGD, the entire dataset is divided into multiple (mini-)batches, and weight gradients are calculated and applied to the network parameters (weights) for each batch via backward propagation. Unlike inference, the training algorithm reuses the intermediate results (e.g., feature maps) produced by a forward propagation during the backward propagation, thus requiring a lot of memory space [55].

This GPU *memory capacity wall* [33] often becomes the limiting factor on DNN size and its throughput. Specifically, such a large memory capacity requirement forces a GPU device to operate at a relatively small batch size, which often adversely affects its throughput. The use of multiple GPUs can partially bypass the memory capacity wall because a careful use of multiple GPUs can achieve near-linear improvements in throughput [27, 28, 59]. However, such a throughput improvement comes with the linear increase in the GPU cost, which is often a major component of the overall system cost. As a result, the use of multiple GPUs often ends up with sub-optimal cost efficiency (i.e., throughput/system cost) as it does not change the fact that each GPU is not operating at its full capacity due to the limited per-GPU batch size.

This memory capacity problem in DNN training has drawn much attention from the research community. The most popular approach is to utilize the host CPU memory as a backing store to offload some of the tensors that are not immediately used [8, 9, 24, 42, 55, 62]. However, this *buffering-on-memory* approach fails to provide robust performance as the training process on the GPU contends with applications running on the CPU for memory bandwidth and capacity (e.g., data augmentation tasks [5, 41, 57, 61] to boost training accuracy). Moreover, these proposals focus mostly on increasing batch size but less on improving training throughput. Therefore, they often yield a low training throughput as the cost of CPU-GPU data transfers outweighs a larger batch's benefits.

Thus, we propose FlashNeuron, the first DNN training system using a high-performance SSD as a backing store. While NVMe SSDs are a promising alternative to substitute or augment DRAM, they have at least an order of magnitude

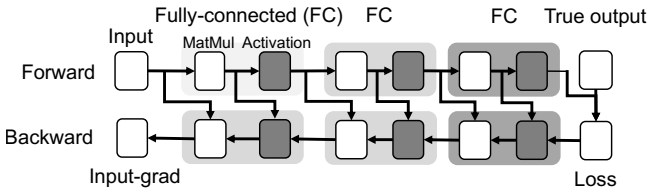

Figure 1: DNN training iteration and data reuse pattern.

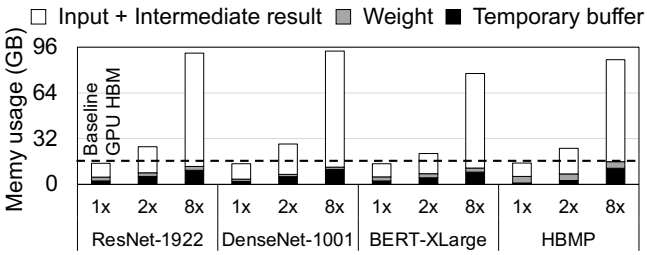

Figure 2: Breakdown of GPU memory usage in DNN training.

lower bandwidth than both HBM DRAM on GPU and DDRx DRAM on CPU. Therefore, it is critical to effectively smooth bandwidth utilization to minimize bandwidth waste while overlapping GPU computation with GPU-SSD data transfers. To this end, FlashNeuron introduces an offloading scheduler, which judiciously selects a set of tensors to offload to the SSD. On the host side, FlashNeuron is realized by a lightweight user-level I/O stack, which leaves a minimal resource footprint on CPU cycles and memory usage as the GPU and the SSD directly communicate for tensor data transfers utilizing GPUDirect [17] technology.

We prototype FlashNeuron on PyTorch [50], a popular DNN framework, and evaluate it using four state-of-the-art DNN models. Our evaluation with the state-of-the-art NVIDIA V100 GPU with 16GB DRAM shows that FlashNeuron can scale the batch size by a factor of 12.4× to 14.0× over the maximum allowable batch size using the GPU memory only. By selecting the optimal batch size, FlashNeuron improves the training throughput by 30.3% on average over the baseline with no offloading, with a maximum gain of 37.8%. At the same time, FlashNeuron also provides excellent isolation between CPU and GPU processes. Even under an extreme condition of CPU applications utilizing 90% host memory bandwidth, the slowdown of the DNN training on GPU falls within 8% of standalone execution, while the slowdown of buffering-on-memory can be as high as 67.8% (i.e., less than one-third of the original throughput).

Our contributions can be summarized as follows:

- We identify a bandwidth contention problem in recent buffering-on-memory proposals [8, 9, 24, 42, 55, 62] and propose FlashNeuron, the first *buffering-on-SSD* solution to overcome this problem.
- We introduce a novel offloading scheduler to fully utilize the scarce SSD write bandwidth.
- We implement a lightweight user-space I/O stack customized for DNN training, which orchestrates SSD-GPU direct data transfers with minimal CPU intervention.
- We prototype FlashNeuron on PyTorch, a popular DNN framework, and evaluate it using four state-of-the-art DNNs to demonstrate its effectiveness for increasing batch size and hence training throughput, while minimally disturbing applications on CPU.

2 Background and Motivation

2.1 DNN Training

Deep Neural Networks (DNN) are widely used for many machine learning tasks such as computer vision [19, 22, 38, 53, 54], natural language processing [2, 11, 13, 51, 67], bioinformatics [46, 73] and so on. For a DNN to effectively perform a target task, it has to learn optimal network parameters using a large amount of labeled data — a process called *training*. DNN training is often performed using mini-batch stochastic gradient descent (SGD) algorithm [15]. In this algorithm, a training process is divided into multiple *epochs*, where a single epoch processes the entire dataset exactly once. Then, a single epoch is further divided into multiple *iterations*, where each iteration processes a single partition of the dataset, called *(mini-)batch*, to update network parameters.

As shown in Figure 1, an iteration consists of two steps: forward pass — a process of computing error for the given input batch, and backward pass — a process of back-propagating errors and updating network weights. A forward pass starts from the very first layer. Given input data, it simply performs the computation associated with the first layer using the layer's current weight and then passes the outcome to the next layer. This process is repeated until the last layer is reached. At that point, the error (also called loss) of the model is computed by comparing the last layer's outcome with the correct output. Then, the backward propagation starts from the last layer. During this step, i) the gradient of a layer's inputs to the final error is computed (using the gradient of the next layers' inputs to the final error) and passed to the next layer, and ii) the gradient of the layer's weights to the final error is computed using i) and stored. Once the backward propagation finishes in the first layer, the weights of all layers are updated accordingly based on the weight gradient computed during the backward pass.

2.2 Memory Capacity Wall in DNN Training

This training process exhibits an interesting data reuse pattern. Specifically, during a forward pass, inputs and intermediate computation results (e.g., products of weights and inputs in the feed-forward layer, followed by activation) of each layer should be buffered. Then, during the backward pass, i) the buffered intermediate computation results of a

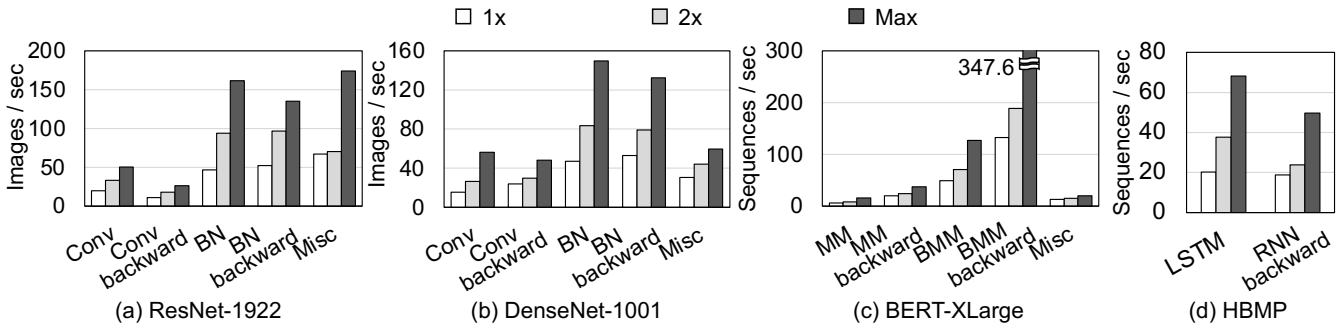

Figure 3: Per-layer throughput of key layers in various DNN models. (`Conv`: Convolution, `BN`: BatchNorm, `MM`: Matrix Multiplication, `BMM`: Batched Matrix Multiplication). *1*× represents the maximum batch size of the baseline using GPU memory only. *Max* represents the batch size that saturates the training throughput with an idealized assumption of zero offloading overhead.

layer are used to compute the layer's gradient, and ii) the buffered inputs are used to compute the gradient of the layer's weights. Arrows in Figure 1 illustrates this data reuse pattern. This data buffering does not cause a problem when the network is shallow. However, a recent trend in deep learning is to utilize networks with a large number of layers (i.e., *deep* networks). With this trend, the amount of memory capacity required to buffer data (i.e., inputs and intermediate computation results for each layer) becomes much larger. It is not feasible in these deep networks to train the network using a large batch size as the required memory capacity for data buffering exceeds the available GPU memory size.

Figure 2 shows the required memory capacity of the DNN models across different batch sizes. Specifically, for each model, the figure shows the minimum memory capacity required to perform training for a certain batch size successfully. Here, 1× represents the maximum batch size that this model can run on a state-of-the-art GPU (i.e., Tesla V100) with 16GB memory. 2× and 8× represent the 2× and 8× batch sizes of the base (1×) batch size. To run on these large batch sizes, we offload all the tensors to host CPU memory except for those of the layer currently being executed. On the base batch size (1×), the required capacity is just below 16GB, indicating that this model almost fully utilizes the provided GPU memory. However, this model cannot be run on a GPU with 16GB memory when we set the batch size to be 2× or 8× as the required memory size far exceeds the available memory size. This figure also shows that most of the memory capacity is occupied by the inputs and the intermediate computation results for each layer. Other memory objects such as weights or temporary buffers (e.g., temporary workspace for convolution operations) take a relatively small portion of these models' total memory consumption.

GPU memory capacity bottleneck described above significantly limits the per-GPU batch size of the DNN models. Using a small batch size often results in the GPU's lower utilization, which leads to lower throughput [3, 16, 68, 69]. Figure 3 presents the per-layer throughput of key layers (i.e., layers accounting for a significant fraction of the total time) in various DNN models. We run each layer in isolation for this exploratory experiment without considering the overheads of tensor offloading, host-GPU communication, etc. The figure shows that there is still significant room for additional throughput by increasing the batch size. GPU resources are being underutilized even at the maximum per-GPU batch size if only GPU memory is used. In this scenario, a GPU throughput can be improved by ameliorating the GPU memory capacity bottleneck. One potential concern is that larger batch size can sometimes negatively affect the model accuracy [20, 30, 40]. However, for extremely deep neural network models, the base batch size is relatively small, and thus an increase in batch size is expected not to affect the final model accuracy severely.

2.3 Overcoming GPU Memory Capacity Wall

A popular approach to overcome GPU memory capacity bottleneck is to buffer data in the host CPU memory. For example, both vDNN [55] and SuperNeurons [62] (selectively) offload activation tensors to the CPU memory. These buffering-on-memory solutions can interfere with the CPU processes for memory bandwidth and capacity to pay a significant opportunity cost. For example, running *data augmentation* on CPU at every iteration of DNN training is a common practice [32, 41, 66] to prevent overfitting to the training data set. A typical data augmentation pipeline consists of image loading, decoding, and a sequence of geometric transformations [5, 41, 57, 61], requiring high memory bandwidth.

Figure 4 shows the GPU training throughput of a buffering-on-memory system while the CPU is continuously running a multi-threaded data augmentation task composed of rotation, transposition, and color conversion. By adjusting the number of data augmentation threads, we control the amount of memory bandwidth consumed by the CPU task (i.e., *50%*: 21GB/s, *70%*: 29GB/s, *90%*: 36GB/s). The throughput of DNN training with buffering-on-memory is noticeably degraded due to memory bandwidth contention. To avoid this problem, we

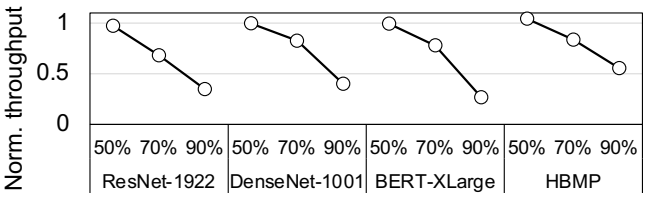

Figure 4: Normalized throughput of buffering-on-memory system (vDNN [55]-like) when the host CPU is running a data augmentation with varying degrees of contention.

propose a new solution that buffers inputs and intermediate data (tensors) to SSDs. Specifically, we leverage direct peer-to-peer communication between the GPU and NVMe SSD devices so that data buffering does not consume either host CPU cycles or memory bandwidth. This *buffering-on-SSD* approach complements the popular buffering-on-memory approach, hence improving overall resource utilization over various use cases.

3 FlashNeuron Design

3.1 Overview

FlashNeuron is a library that can be integrated into popular DNN execution frameworks. Figure 5 shows the system overview of FlashNeuron. FlashNeuron consists of three components: offloading scheduler, memory manager, and peer-to-peer direct storage access. Specifically, the offloading scheduler identifies a set of tensors (i.e., multidimensional matrices) to offload and generates an offloading schedule by considering multiple factors such as tensor sizes, tensor transfer times, and forward/backward pass runtime. Once the schedule is determined, the memory manager orchestrates data transfers between the GPU memory and the SSDs using peer-to-peer direct storage access to minimize performance overheads from offloading.

3.2 Memory Manager

Tensor Allocation/Deallocation. Instead of buffering all input and intermediate data in memory, FlashNeuron chooses to buffer selected tensors in SSDs, which requires extra tensor allocations and deallocations. Since frequent GPU memory allocations and deallocations using runtime (e.g., CUDA) incur noticeable performance overheads, FlashNeuron employs a custom memory allocator. The custom memory allocator first reserves the whole GPU memory space initially and manages memory allocation/deallocation itself. In FlashNeuron, tensors are allocated when i) a tensor is first created during the forward propagation or ii) an offloaded tensor is prefetched from the SSD to the memory during a backward pass. On the other hand, tensors are deallocated when i) a tensor is completely offloaded from the memory to the SSD, or ii) a tensor is no longer used by any layer during the iteration. To track the lifetime of a tensor, a reference counting mechanism

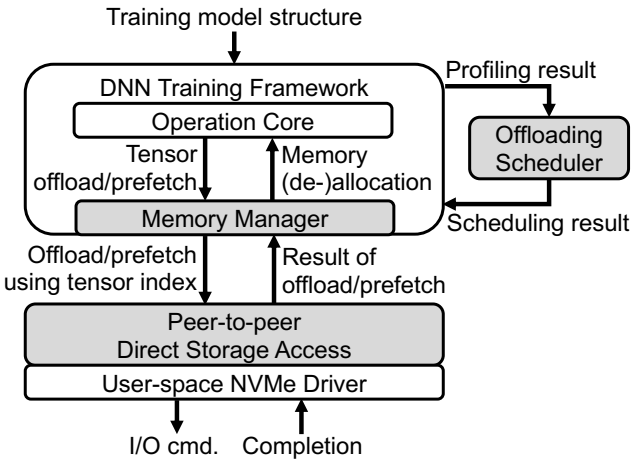

Figure 5: System overview of FlashNeuron.

(used in PyTorch [50] and TensorFlow [1]) is utilized. For DNN frameworks employing a static computational graph like Caffe, the memory manager traverses the computational graph and tracks the tensor lifetime through pointer chasing of tensors attached to each layer.

One crucial issue in the tensor allocation and deallocation is fragmentation. If we allocate memory addresses for all tensors from the beginning, severe memory fragmentation can occur since only some tensors are offloaded to the SSDs, effectively making holes in the GPU physical memory address space. To avoid this issue, we allocate memory-resident (i.e., not offloaded) tensors from the lowest end of the memory address space and allocate ephemeral (i.e., offloaded) tensors from the highest end of the memory address space. Since the ephemeral data has a very short lifetime during the forward pass, only a tiny portion of the memory address space is utilized for such data, and thus the amount of fragmented memory space becomes negligible.

Managing Offloading and Prefetching. The memory manager interacts with peer-to-peer direct storage access (P2P-DSA) to perform offloading and prefetching. It initiates an offloading request of a tensor to P2P-DSA during the forward pass immediately after its use by the next layer. At the end of each layer's execution, the memory manager checks whether the offloading request is completed (i.e., the tensor is wholly offloaded to the SSD). Then, the tensor is deallocated from the GPU memory at this point.

The memory manager issues prefetch requests to the SSD during the backward pass. At the beginning of the backward pass, it first allocates memory and initiates prefetch requests for the set of tensors that are soon to be used. Then, whenever those tensors are used and freed, the memory manager eagerly prefetches additional tensors using the available memory space while reserving enough memory for execution to run the largest layer.

Augmented Compressed-Sparse Row (CSR) Compression and Decompression. When offloading a tensor, the

memory manager applies CSR compression if the compression ratio estimated during the profiling iteration is greater than one. The CSR compression is only applied to output tensors of ReLU. We observe that ReLU outputs have a high sparsity, ranging from 43% up to 75% during the training process. Since a tensor is a multi-dimensional matrix, we cast the tensor into a two-dimensional matrix whose column has 128 entries. Then, we apply a slightly different CSR format where we replace a vector storing the column index of each element (often called JA vector) to a set of bit-vectors where each bit vector represents a set of nonzero elements for a row. By doing so, the size of CSR format representation decreases by *8 bits (to represent the column index)* $\times$ *the number of nonzero elements in the matrix* and increases by *1 bit* $\times$ *the total number of elements in the matrix*. This representation is beneficial when more than one-eighth of all the elements are nonzero. Since this is the typical case for input and intermediate tensors, we apply this technique to improve the compression ratio. We implement a specialized routine to perform this augmented-CSR compression/decompression in GPU. According to our evaluation, the runtime overhead of these compression/decompression operations is negligible.

Use of a Half-precision Floating Points (FP16) for Offloaded Tensors. To further reduce the traffic between the GPU and the SSD, the memory manager exploits the fact that neural network can tolerate a certain level of precision loss without significantly degrading the final model accuracy. Specifically, during a forward path, the memory manager first converts the offloaded tensor to FP16 format (from FP32) and then stashes them in the SSD. Later, during a backward propagation, the offloaded FP16 tensor is prefetched, padded to FP32, and reused. Naturally, the use of a lower-precision tensor comes with the potential degradation in the final model accuracy [44, 63]. However, a previous study demonstrates that the technique of selectively utilizing FP16 for the offloaded tensors incurs less accuracy degradation than processing all tensors in FP16 [25]. The key observation is that the FP32 tensor is utilized during forward propagation, and the stashed FP16 version of the same tensor is only utilized during a backward propagation (Delayed Precision Reduction in [25]).

3.3 Offloading Scheduler

The offloading scheduler in FlashNeuron takes a DNN model as input and derives an optimal tensor offloading schedule, which is designed with the following rationale. First, it should offload enough tensors so that the GPU can correctly run at the target batch size without triggering an out-of-memory error. Second, it should avoid excessive data transfers from the GPU to the SSD (and vice versa), thus minimizing the increase of the iteration time induced by tensor offloading.

The offloading scheduler finds an optimal schedule for a given target batch size. It first performs a profiling iteration. At this iteration, all the tensors that are buffered during the forward pass (i.e., input and intermediate data for each layer)

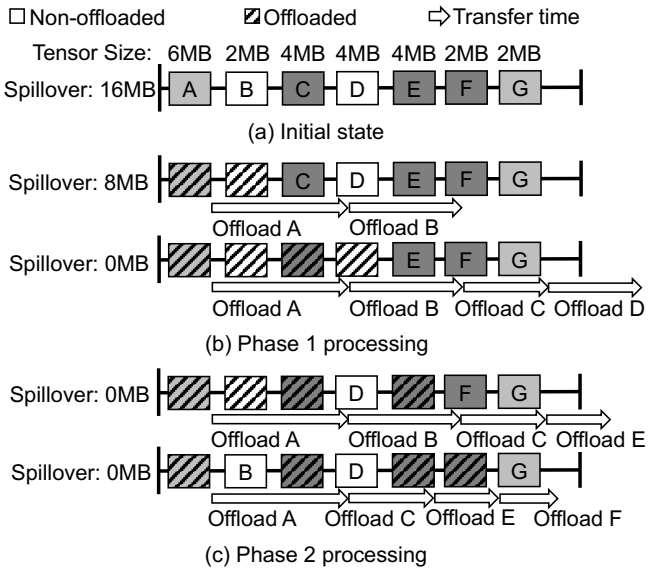

Figure 6: Tensor selection walk-through. Darker boxes indicate tensors with a higher compression ratio, whereas lighter boxes those with lower compression ratio.

are offloaded to the SSD so that the system can run with a large target batch size without causing an out-of-memory error. Profiling iteration collects i) the size of each buffered tensor, ii) the time it takes to offload each buffered tensor, iii) the expected compression ratio of a tensor using CSR (Compressed Sparse Row) format and half-precision floating-point conversion, iv) the execution time for the forward pass and the backward pass (excluding tensor offloading time), and v) the total size of the other memory-resident objects (e.g., weights, temporary workspace). Once this profiling iteration completes, information collected during this iteration is passed to the offloading scheduler.

Phase 1: Linear Tensor Selection. The first phase of the scheduler is to iteratively select a certain number of tensors from the beginning until the total size of the unselected tensors plus the total size of the other memory-resident objects (i.e., weights and temporary workspace) becomes smaller than the total GPU memory size. Figure 6 illustrates this process, where the forward-pass with seven buffered tensors (labeled A through G) is to run on a hypothetical GPU with 8MB physical memory. Figure 6(b) shows the example selection process of Phase 1. At this point, the scheduler checks whether the total data transfer time, which is computed by summing up the individual tensor offloading times, is smaller than the total execution time of all layers in the forward pass. If this condition is satisfied, the scheduler adopts this schedule and stops because it can fully overlap tensor offloading with layer computation via scheduling. If not, the offloading scheduler enters the second phase.

Phase 2: Compression-aware Tensor Selection. The second phase of the scheduler is run only when a satisfactory

schedule is not found in the first phase. This indicates that the current schedule spends too much time offloading the tensors, and such the transfer time has now become the new bottleneck. To solve the issue, our scheduler replaces the already selected tensors with compression-friendly tensors expected to have high compression ratios with CSR and FP16 conversion illustrated in Figure 6(c). Specifically, the scheduler performs the following steps in an iterative way to refine the existing schedule. First, the scheduler excludes the last uncompressible tensor selected from Phase 1. It is replaced with one or more tensors having the highest expected compression ratios among the tensors that are not yet selected, such that the size of the newly selected tensors exceeds the excluded tensor size. Then, it recomputes the expected total data transfer time, assuming that the compressed tensor takes a fraction (inversely proportional to the compression ratio) of the original offloading time. If this total transfer time does not exceed the forward pass's total execution time, the scheduler stops. Otherwise, it repeats this process until the condition is satisfied or there exist no compression-friendly tensors (i.e., tensors whose size does not decrease after compression).

If a *satisfactory* schedule is found, the corresponding batch size is likely not to increase the iteration time and achieve higher throughput than the baseline. On the other hand, if our scheduler stops as it cannot find more compression-friendly tensors, the generated schedule is expected to incur some delay from tensor transfers. However, this schedule can still be used to run DNN training at a larger batch size (but likely at a lower throughput).

3.4 Peer-to-Peer Direct Storage Access

Peer-to-peer direct storage access (P2P-DSA) enables direct memory access between a GPU and NVMe SSDs without using the host DRAM buffer to minimize host intervention during SSD read/write. P2P-DSA builds on GDRCopy [14] and SPDK [58] to communicate tensors from/to a GPU to/from NVMe SSDs. GDRCopy is a fast GPU memory copy library based on NVIDIA GPUDirect [17], a technology that exposes the GPU memory to be accessed directly by other PCIe peripherals. Intel SPDK exposes a block-level I/O interface directly to the user-space software. P2P-DSA is a lightweight layer that leverages these two technologies to enable direct offloading/prefetching tensors from GPUs to SSDs. To maintain each tensor's metadata offloaded to SSDs, P2P-DSA contains a metadata table consisting of a `long` LBA value and a `boolean` value to check the I/O completion.

Example Walk-through of a Transfer Request. Figure 7 illustrates the operations of the transfer request (offloading from the GPU to the SSD) in greater details. When `P2PDSA_issue` is called, ❶ P2P-DSA get the `index`, `buffer`, and `direction(write)` information from the transfer request. Then, ❷ the logical block address (LBA) allocator is called to allocate a set of contiguous blocks on a single SSD device or multiple SSD devices (when multiple SSDs are utilized

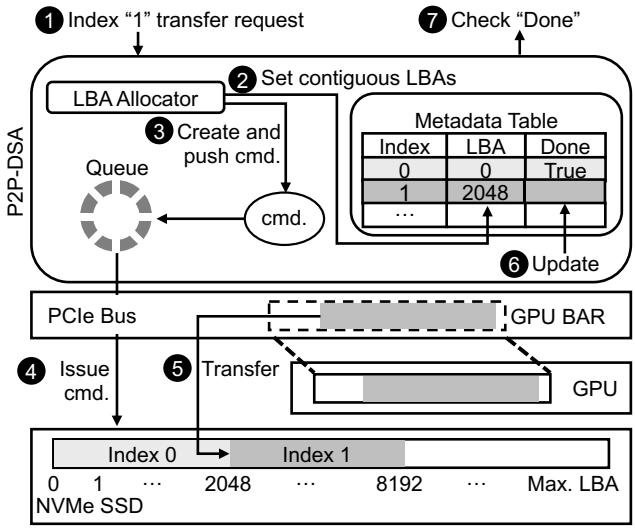

Figure 7: Overall structure of P2P-DSA with an example walk-through (write).

to boost offloading/prefetching bandwidth). The LBA of the first block allocated from the LBA allocator is updated at the metadata table's appropriate location. After this point, ❸ P2P-DSA creates a command for each logical block and then enqueues it to the command queue. Here, an NVMe command contains i) the source address (GPU memory address is translated to PCIe bus address by GPUDirect), and ii) the device address (computed using the LBA in metadata table).

When `P2PDSA_update` is called, ❹ commands queued in the software command queue are fetched and issued to NVMe SSD as long as the NVMe device submission queue has space. Then, ❺ NVMe SSD devices will execute these requests and perform direct data transfers between SSD devices and the GPU. Sometime later, these transfer requests will be completed, and their status will be updated in the NVMe device completion queue. When the `P2PDSA_update` is called once again, ❻ `P2PDSA_update` will clear the completion queues and updates the metadata table by setting corresponding done bits. At this point, ❼ if the application calls `P2PDSA_is_done` for the already offloaded tensor, it will return true. The reverse-path (prefetching data from the SSD to the GPU memory) is performed similarly except that i) LBAs are read from the metadata table instead of being allocated, and ii) read commands are issued instead of write commands. In both offloading and prefetching cases, most data accesses are sequential accesses, which are more advantageous than random accesses in throughput and endurance.

Implications on SSD lifetime. The endurance of an SSD depends on the program and erase (P/E) cycles for the NAND blocks. Therefore, for write-intensive workloads, a primary concern for the flash-based SSDs is the endurance degradation by wear-out of NAND blocks. We estimate the guaranteed (minimum) endurance of the SSD when running the P2P-DSA workload, using drive writes per day (DWPD) (i.e., 3 DWPD

Table 1: System configurations.

CPU	Intel Xeon Gold 6244 CPU 8 cores @ 3.60GHz
GPU	NVIDIA Tesla V100 16GB PCIe
Memory	Samsung DDR4-2666 64GB (32GB × 2)
Storage	Samsung PM1725b 8TB PCIe Gen3 8-lane × 2 (Seq. write: 3.3GB/s, Seq. read: 6.3GB/s)
OS	Ubuntu server 18.04.3 LTS
Python	Version 3.7.3
PyTorch	Version 1.2

for five years of Samsung PM1725b SSD [49], assuming a 50% write-50% read workload like P2P-DSA). If the training workload is running 24×7, the endurance is estimated to be about 7,374 hours, which is 307 days (i.e., 3 DWPD × 5 years × 365 days × 8,000 GB × 2 (50% write) / 3.3 GB/s / 86,400 seconds/day). While a longer lifetime would be desirable, we note that our estimation is conservative as P2P-DSA only performs sequential writes to sustain the write amplification factor (WAF) of (nearly) one to maximize endurance. SSD manufacturers typically use 4KB random write [49] to report the endurance number, which has a higher WAF than sequential writes at least by 3.5× [6, 21]. Furthermore, if P2P-DSA uses the emerging low-latency SSDs, such as Intel Optane SSD [45] and Samsung Z-SSD [70], the endurance can be further improved by a factor of 5× to 10×. Finally, we can further extend the SSD lifetime by leveraging tradeoffs between cell retention time and P/E cycles [7, 26]. An offloaded object has a very short lifetime and hence requires a much lower retention time (i.e., one training iteration time, which is in order of seconds and minutes at most, rather than years as required by modern SSDs). This can improve the P/E cycles by 46× or more [7]. With these optimizations, the expected SSD lifespan can increase by multiple orders of magnitude.

4 Evaluation

4.1 Methodology

System Configurations. We evaluate FlashNeuron on a Gigabyte R281-3C2 rack server with NVIDIA Tesla V100 and two Samsung NVMe PM1725b SSDs. The details of hardware and software configurations are summarized in Table 1.

Workloads. Among various DNN models, we choose four state-of-the-art models and scale them up to represent the future DNN models with very deep structures: ResNet-1922 [19] and DenseNet-1001 [22, 74] are state-of-the-art deep CNN models for image processing. BERT-XLarge [13] and HBMP [60] are two of the top-performing models for natural language processing tasks. ResNet-1922 and DenseNet-1001 [74] are selected based on the deepest network of ResNet and DenseNet models. The depths of BERT-XLarge and HBMP are increased by 2× and 4× from the maximum size stated in the original papers. Note that these naively scaled models do not necessarily improve accuracy. Our purpose is to use them as proxies for future DNN models requiring a much larger capacity for efficient training. The specifics are summarized in Table 2.

Table 2: Suite state-of-the-art DNN models and datasets used, major layer types and counts.

Network	Dataset	# of layers	Structure
ResNet-1922 [19]	ImageNet [12]	1922	(Conv1→BN1→ ReLU→Conv2→ BN2→ReLU)^{n}
DenseNet-1001 [22, 74]	ImageNet [12]	1001	(Conv1→BN1→ ReLU)$^{n-1}$→ Conv2→BN2→ ReLU
BERT-XLarge [13]	SQuAD 1.1 [52]	48 blocks	(Embd1→Embd2 →Embd3→FC1 →Attn→FC2 →LNorm)^{n}
HBMP [60]	SciTail [31]	24 hidden layers	FC^{m}→LSTM^{n}

4.2 Performance Evaluation

Overview. Figure 8 shows the training throughput over varying batch sizes. The baseline uses GPU memory only. FlashNeuron (SSD) and FlashNeuron (Memory) offload tensors to SSD and CPU memory, respectively, with no interference from CPU processes. Note that FlashNeuron (Memory) represents a state-of-the-art buffering-on-memory scheme. The dotted line shows the best throughput that can be achieved by the baseline. To demonstrate the effectiveness of FlashNeuron, we mark with an arrow the maximum batch size for which the proposed offloading scheduler is able to find an effective schedule (i.e., a schedule that does not increase the estimated forward-pass time). The training throughput indeed peaks at the batch size marked with the arrow. As we further increase the batch size, the throughput gets degraded as the cost of tensor offloading outweighs the benefits of the increased batch size.

FlashNeuron (SSD) improves the training throughput by up to 37.8% by selecting the optimal batch size and can increase the batch size by up to 5.0× while achieving at least the same throughput as the baseline (or higher). In some cases, increasing batch size may give additional benefits to reduce total training time further. For example, the effectiveness of batch normalization is known to diminish for small batches, and increasing the batch size can yield higher accuracy, faster convergence, or both [36]. However, when the batch size is too large, the limited bandwidth between the GPU and the SSD becomes the bottleneck and offsets the higher utilization benefits. Some configurations in Figure 8 (e.g., batch size of 8+ in ResNet-1922 and 10+ in DenseNet-1001) represent these cases. The performance gap between FlashNeuron (SSD) and FlashNeuron (Memory) is attributed to the difference in sustainable write throughput. While FlashNeuron (Memory) can utilize the nearly full PCIe write bandwidth (13.0GB/s), FlashNeuron (SSD) is limited by the write throughput of the SSD device (3.0GB/s×2). Thus, FlashNeuron (Memory)

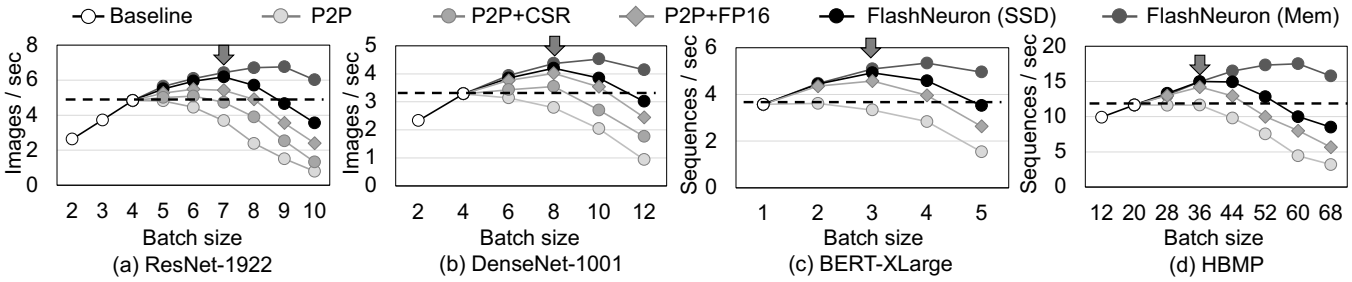

Figure 8: Throughput of FlashNeuron with varying batch sizes (`P2P`: Baseline with P2P, `P2P+CSR`: With P2P and CSR compression, `P2P+FP16`: With P2P and FP16 conversion). The arrow shows the maximum throughput of FlashNeuron (SSD).

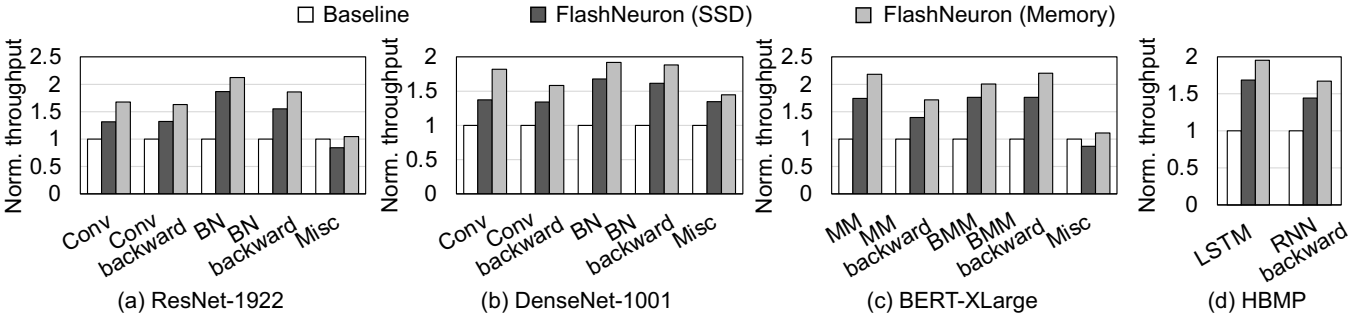

Figure 9: Normalized per-layer throughput of key layers across training scenarios (`Conv`: Convolution, `BN`: BatchNorm, `MM`: Matrix Multiplication, `BMM`: Batched Matrix Multiplication).

achieves up to 49.1% throughput gain (with an average of 43.9%) over the baseline. This performance gap can be closed by FlashNeuron (SSD) employing additional SSDs to saturate the PCIe channel bandwidth.

Source of Performance Improvement. Overall, FlashNeuron benefits from its two optimizations: CSR and offloading tensors using FP16 representation. Figure 8 shows the improvements from each optimization. Here, `P2P` represents the configuration where tensors are offloaded using P2P-DSA, but without any other optimization (e.g., CSR compression or FP16). This configuration enables the use of a larger batch size beyond the GPU memory capacity limit. However, the limited bandwidth between the GPU and the SSD limits the performance. P2P with CSR compression (`P2P+CSR`) improves the baseline performance by 7.14%. Note that the tensor compression does not improve the performances of BERT-XLarge and HBMP because those models do not utilize a ReLU layer. P2P with FP16 conversion (`P2P+FP16`) improves the performance by 21.41% over the baseline. The improvement is greater than that of P2P+CSR because the use of the FP16 format cuts the traffic by half, while the CSR compression is only applied for a limited set of tensors (e.g., output tensors of ReLU).

Figure 9 shows the per-layer throughput of FlashNeuron at the optimal batch size normalized to the baseline using GPU memory only. By employing a larger batch size, FlashNeuron substantially increases the throughput for the key layers. Batch normalization (BN) and LSTM layers benefit the most from the increase in batch size, whereas convolution (Conv) layers demonstrate relatively modest improvements. It is because the Conv layer is known to be compute-intensive and already has a high resource utilization even for the baseline.

Table 3: Batch sizes achieving maximum throughput and the maximum batch size FlashNeuron can run.

Network	Baseline	Maximum throughput		Runnable maximum	
	Batch	Batch	Ratio	Batch	Ratio
ResNet-1922	4	7	1.75×	56	14.0×
DenseNet-1001	4	8	2.00×	52	13.0×
BERT-XLarge	1	3	3.00×	14	14.0×
HBMP	20	36	1.80×	248	12.4×

Since the bandwidth of the PCIe channel and SSD writes is the limiting factor for performance, future scaling of both PCIe and SSD write bandwidth will further improve the throughput. For example, PCIe 5.0 interconnects [48] will provide 4× higher bandwidth than PCIe 3.0 used in this work, enabling FlashNeuron (SSD) to utilize 4× larger batch size. Figure 3 demonstrates that there is still substantial room for further throughput improvement, and higher bandwidth interconnects in the future will close this performance gap.

Maximum Batch Size. Table 3 shows the largest batch size for different configurations. The first column ("*Baseline*") is the maximum batch size using GPU memory only (i.e., without using FlashNeuron). The second column ("*Maximum throughput*") shows the batch size for which FlashNeuron (SSD) yields the highest throughput, which is marked by

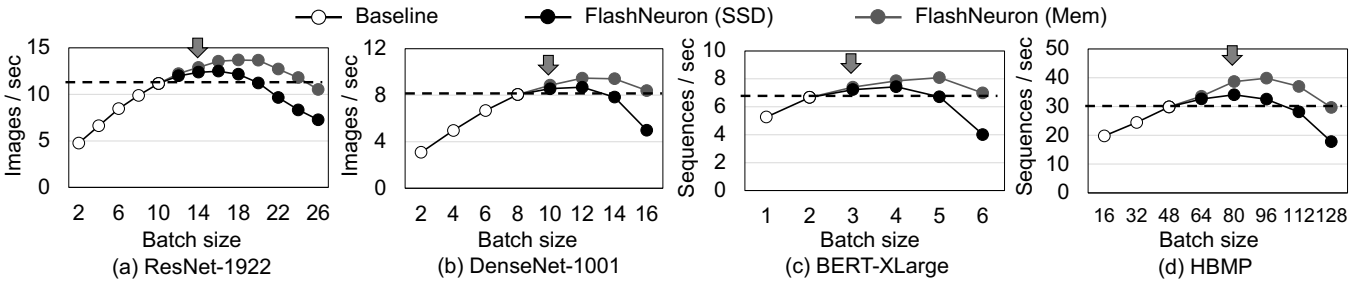

Figure 10: Throughput of FlashNeuron with half-precision. The arrow shows maximum throughput of FlashNeuron (SSD).

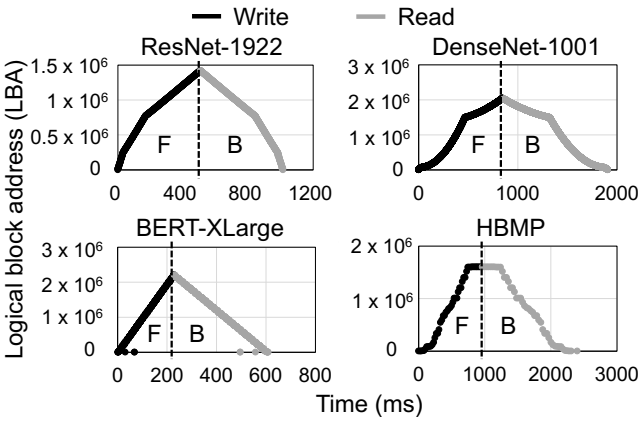

Figure 11: LBA access pattern during a single iteration (F: Forward propagation, B: Backward propagation).

an arrow in Figure 8. Finally, the third column ("*Runnable maximum*") shows the maximum batch size that FlashNeuron can run to completion. On average, FlashNeuron uses 2.09× larger batch size to maximize the training throughput and increases the maximum runnable batch size by a factor of 13.4×. When FlashNeuron is operating with the runnable maximum batch size, FlashNeuron offloads 59.2GB of tensors on average (up to 68.6GB for DenseNet-1001) and occupies 33.2GB (up to 48.9GB for HBMP) storage space.

Half-precision Training. Based on the observation that many neural network models can still maintain the competitive accuracy using FP16 representations, FP16 training is gaining popularity. For example, the high-end NVIDIA GPUs come with Tensor Cores, which are specialized functional units for FP16 computations. Unfortunately, when the tensor is already represented in FP16, FlashNeuron does not benefit from converting the offloaded tensors to FP16. However, our experiments still demonstrate that FlashNeuron can result in extra speedup as well as an increase in the per-GPU batch size. Figure 10 shows the throughput of FlashNeuron over varying batch sizes when FP16 values are used for both weights and activations. FlashNeuron (SSD) enables the use of a 1.8× larger batch size while preserving the training throughput compared to the baseline. By employing larger batch sizes, FlashNeuron (SSD) and FlashNeuron (Memory) achieve 8.04% and 22.98% throughput improvement over the baseline, respectively. This FP16 training requires less memory/storage capacity per batch and thus enables even larger batch sizes. The speedup from FlashNeuron is smaller in this scenario than full-precision as the overall iteration time becomes shorter, thus having a much narrower window for tensor offloading.

I/O Pattern. Ensuring the sequential read/write by P2P-DSA is important for both performance and endurance. Figure 11 shows the SSD's logical block address (LBA) access pattern during a single iteration. During a forward propagation, offloaded tensors are allocated sequentially in the LBA space (on the left side of the figure's dotted line). In a backward propagation, the most recently written tensors are read first, and the least recently written tensors are read last, as shown on the right side of the dotted line. Each offloaded/prefetched tensor's size ranges from 2MB to 310MB, and it is sufficiently large to saturate the SSD's read/write bandwidth thoroughly.

Cost Efficiency. As of September 2020, DDR4 DRAM on the host CPU costs about $3.6/GB on average and NAND flash SSD about $0.102/GB [29, 43, 56]. Assuming the same capacity, FlashNeuron (SSD) achieves 35.3× higher cost-efficiency. HBM2 DRAM has much higher $/GB than DDR4, and thus scaling its capacity will be much more costly.

4.3 Case Studies

Our premise is that the common practice of leaving CPU (mostly) idle while running DNN training on GPU is suboptimal. Thus, we envision co-locating CPU jobs with DNN training to improve resource utilization substantially. To not degrade DNN training throughput running at large batch size, it is crucial to provide performance isolation between the co-located CPU and GPU processes. FlashNeuron is a unified framework that flexibly supports both SSD and memory offloading to minimize resource contention for a wide range of co-located CPU workloads. Superior performance isolation of FlashNeuron can enable consolidation of CPU applications and DNN training jobs. The two case studies in this section are presented *not* to claim that they are common use cases today, *but* to demonstrate that even memory-intensive CPU workloads can be effectively co-located with DNN training using FlashNeuron (SSD). For I/O-intensive workloads, one can opt to use FlashNeuron (Memory) to avoid I/O contention.

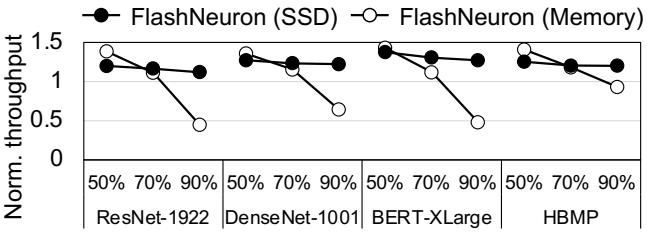

Figure 12: Normalized throughput of FlashNeuron (SSD) and FlashNeuron (Memory) when the host CPU is running a memory-intensive image transformation workload [23].

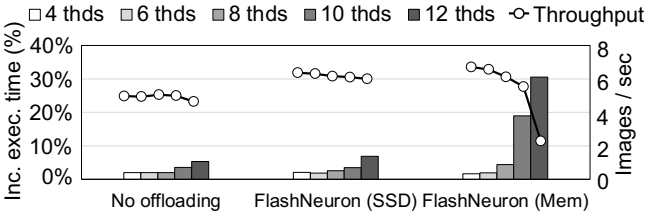

Figure 13: Increase in execution time of data augmentation tasks processing 256 2K (2048×1080) resolution images on CPU and training throughput of ResNet-1922 on GPU.

4.3.1 Co-locating Bandwidth-Intensive Tasks on CPU

The first use case is data augmentation tasks [5, 41, 57, 61] running on CPU while executing DNN training on GPU. Employing a data augmentation for DNN training is a common practice to prevent the model from overfitting to the data set, hence providing more robustness. Our example data augmentation transforms 2K (2048×1080) resolution images with a sequence of geometric operators such as rotation and transposition, as well as re-coloring operators such as color conversion. These operators are commonly used in data augmentation [10, 34, 37]. Note that the actual DNN model works with smaller images, but the data augmentation often works with the original image, and then the augmented image is resized to the model's input image size (e.g., 224×224).

Throughput of DNN Training on GPU. Buffering-on-memory can potentially achieve higher throughput than buffering-on-SSD for the higher write bandwidth of the CPU DRAM than the SSDs. However, the DNN training throughput with buffering-on-memory can be heavily affected by CPU processes' characteristics due to the memory bandwidth contention between CPU and GPU processes. Figure 12 shows the impact of CPU workload on the DNN training throughput on GPU for both FlashNeuron (SSD) and FlashNeuron (Memory). By controlling the number of data augmentation threads, we make the CPU process consume a certain portion of the CPU memory bandwidth. In particular, we use three configurations according to the portion of the CPU DRAM bandwidth consumed by the data augmentation task: *50%* (21GB/s), *70%* (29GB/s), *90%* (36GB/s).

When the CPU consumes 50% of the available memory bandwidth, the training throughput is still at least 35% higher than the baseline for both FlashNeuron (SSD) and FlashNeuron (Memory). However, when the CPU workload is more memory bandwidth-intensive (*75%*), FlashNeuron (Memory) yields only 14.0% throughput gains. This performance loss becomes even worse when the CPU workload utilizes nearly all of the available memory bandwidth (*90%*), where the training throughput degrades by 40.2% on average compared to baseline. On the other hand, FlashNeuron (SSD) still achieves 22.6% and 20.2% throughput gains over the baseline even if the CPU consumes 75% and 90% of the available memory bandwidth, respectively. Even in the worst case, the throughput loss of FlashNeuron (SSD) falls just within 8% of standalone execution, whereas that of FlashNeuron (Memory) can be as high as 67.8% (i.e., having lower than one-third of the original training throughput).

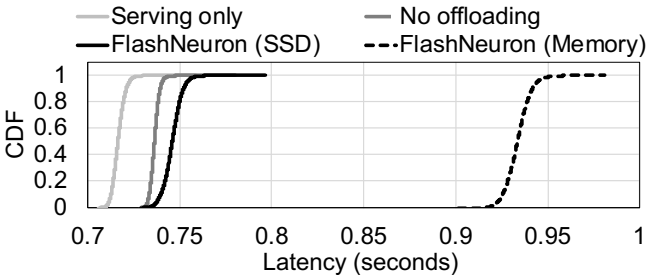

Figure 14: Query latency CDF of CPU inference across the various training scenario.

Execution Time of Data Augmentation Task on CPU. Figure 13 shows both the increase in execution time of the data augmentation task on CPU (bar graph) and DNN training throughput of ResNet-1922 using FlashNeuron on GPU (line graph). All bars are normalized to the baseline, which is standalone execution of the data augmentation pipeline with no co-located GPU processes. *No offloading* represents the case when GPU is running DNN training with no tensor offloading to either host memory or SSDs. FlashNeuron (SSD) only utilizes a minimal amount of the host memory bandwidth (mostly for PyTorch application code) to incur a comparable degree of the slowdown with *No offloading*. In contrast, FlashNeuron (Memory) consumes a large amount of the host CPU memory bandwidth (roughly equal to the maximum bandwidth of a 16-lane PCIe interface) to incur a substantial performance slowdown. The figures show that this memory bandwidth contention can break performance isolation between the CPU and GPU processes to make it much more challenging to deploy them in a consolidated environment.

4.3.2 Co-locating Latency-Critical Tasks on CPU

For the second case study, we select a *DNN inference* task, which is latency-critical; according to Facebook, inference tasks are mostly running on CPUs while requiring a large memory space for users and contents data [18]. We run a BERT-as-service [64] on CPU, which takes user-provided

Table 4: 50%, 95%, and 99% percentile of query latency and delay time ratio compared to *Serving only*.

	No offloading	FlashNeuron (Memory)		FlashNeuron (SSD)	
	Latency	Latency	Delay	Latency	Delay
50%	0.736s	0.933s	30.3%	0.746s	4.11%
95%	0.740s	0.944s	30.7%	0.754s	4.35%
99%	0.743s	0.950s	30.9%	0.758s	4.45%

sentences as input and invokes BERT to return their embedding, while concurrently running a BERT training on GPU.

Figure 14 shows the cumulative distribution function (CDF) of the CPU inference. *Serving only* is a case when there is no process running on GPU, whereas *No offloading* is when BERT is training but using GPU memory only. As shown in this figure and Table 4, FlashNeuron (SSD) incurs less than 5% and 2% slowdown compared to *Serving only* and *No offloading*. In contrast, over 30% latency increase is observed for FlashNeuron (Memory) compared to *Serving only* due to memory bandwidth contention. As for training throughput, FlashNeuron (SSD) experiences only a 1.8% slowdown, whereas FlashNeuron (Memory) as much as 27.5%. This slowdown is not sensitive to the model or dataset and is largely attributed to the bandwidth consumption to offload tensors.

5 Related Work

Augmented GPU Memory for DNN Training. Many proposals build on NVIDIA Unified Virtual Memory (UVM) [42], which enables transparent data sharing over both GPU and CPU memory. However, its performance is often limited due to its excessive page fault handling overhead [39]. To address this problem, several specialized schemes that do not rely on demand-fetching have been proposed to accelerate DNN training [8, 9, 24, 55, 62]. Similar to vDNN [55], moDNN [9] offloads and prefetches tensors in convolution layers in addition to accumulating gradients.

Alternatively, Chen et al. [8] propose to mark the outputs of convolution layers and free unmarked tensors. The freed data is recomputed during a backward pass. Merging the two ideas, SuperNeurons [62] offloads the marked tensors to host memory and saves device memory space. Ooc_cuDNN [24] divides the data in a single layer and performs for a piece of data at a time. The unused data is prefetched from the host memory concurrently with computation. Such mechanisms, however, experience substantial performance degradation when the host CPU is running memory-intensive workloads. To complement this, FlashNeuron offloads tensors directly to SSDs, and thus do not suffer performance degradation even under the presence of memory-intensive processes on the CPU.

Data Transfer Methods between GPU and Storage Devices. Several proposals introduce effective data transfer methods between GPU and storage devices [4, 39, 71]. Dragon [39] leverages the page-faulting mechanism of CPU and read-ahead operation of OS. Upon page fault, page cache in host memory is used as a bridge between GPU memory and NVM storage. SPIN [4] and NVMMU [71] take a step further, removing the usage of the host side buffer, thus allowing direct access from GPU to SSD. However, they are more general-purpose solutions, which perform sub-optimally for DNN training as they do not sequentially read/write.

Reducing Memory Footprint of DNN Models. Another way to relieve the capacity limitations of GPU memory is to optimize the DNN model without compromising the accuracy [25, 35, 72]. Echo [72] reduces the memory footprint by stashing small input values of the attention layers and recomputing the feature maps during the backward passes. Gist [25] applies various footprint reduction techniques by compressing feature maps, especially for ReLU-convolution and ReLU-pooling layers, as well as lower-precision representations (FP8/10/16). Likewise, FlashNeuron exploits sparse matrix representations such as CSR and FP16 representation on offloaded tensors to reduce the traffic to SSD devices.

6 Conclusion

With a relentless pursuit of higher accuracy, DNNs are continuously getting deeper and wider. One significant constraint in scaling trainable DNNs is the limited capacity of the GPU memory. This problem is exacerbated by emerging DNN applications required to handle large inputs. There have been previous attempts to overcome this GPU memory capacity wall through the use of host memory as a buffer for intermediate data generated during the forward pass of DNN training for reuse during the backward pass. However, these approaches experience substantial performance degradation as the host CPU contends for the limited host memory bandwidth. Thus, we propose FlashNeuron, the first buffering-on-SSD approach to offload intermediate data to high-performance NVMe SSDs instead of the host DRAM. FlashNeuron enables large-batch training of very deep and wide neural networks of today and the future to achieve high training throughput. Furthermore, it flexibly supports both SSD and memory offloading to provide excellent performance isolation between GPU training jobs and a wide range of co-located CPU workloads, including memory- and I/O-intensive ones.

Acknowledgments

We extend our thanks to Randal Burns for shepherding this paper. We also thank Jin-Soo Kim and Jaehoon Sim for valuable discussions and their help with P2P-DSA in an early phase of this work. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) (NRF-2020R1A2C3010663) and Samsung Electronics. The source code is available at `https://github.com/SNU-ARC/flashneuron.git`. Jae W. Lee is the corresponding author.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.

[2] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Vaino Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 173–182. PMLR, 2016.

[3] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys*, 52(4), 2019.

[4] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 167–179. USENIX Association, 2017.

[5] Alexander Buslaev, Vladimir I. Iglovikov, Eugene Khvedchenya, Alex Parinov, Mikhail Druzhinin, and Alexandr A. Kalinin. Albumentations: Fast and flexible image augmentations. *Information*, 11(2), 2020.

[6] Zydan Bybin, Mohammed Khandaker, Monika Sane, and Graham Hill. Over-provisioning NAND-based intel SSDs for better endurance. *Intel White Paper*, 2019.

[7] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich Haratsch, Adrian Cristal, Osman Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design*, pages 94–101. IEEE, 2012.

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174v2*, 2016.

[9] Xiaoming Chen, Danny Chen, and Xiaobo S. Hu. moDNN: Memory optimal DNN training on GPUs. In *Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition*, pages 13–18, 2018.

[10] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, June 2020.

[11] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988. Association for Computational Linguistics, 2019.

[12] Jia Deng, Wei Dong, Richard Socher, Li jia Li, Kai Li, and Li Fei-fei. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2009.

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

[14] NVIDIA GDRCopy: A low-latency GPU memory copy library based on GPUDirect RDMA. https://github.com/NVIDIA/gdrcopy.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677v2*, 2018.

[17] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*, pages 620–629. IEEE, 2018.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778. IEEE, 2016.

[20] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: Closing the generalization gap in large batch training of neural networks. In *Proceedings of the Advances in Neural Information Processing Systems 30*, pages 1731–1741. Curran Associates, Inc., 2017.

[21] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of the International Systems and Storage Conference*, pages 10:1–10:9. ACM, 2009.

[22] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Weinberger. Densely connected convolutional networks. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2261–2269. IEEE, 2017.

[23] Yermalayeu Ihar, Antonenka Mikhail, Radchenko Andrey, Dmitry Fedorov, Kirill Matsaberydze, Artur Voronkov, and Facundo Galan. SIMD library. http://ermig1979.github.io/Simd/.

[24] Yuki Ito, Ryo Matsumiya, and Toshio Endo. ooc_cuDNN: Accommodating convolutional neural networks over GPU memory capacity. In *Proceedings of the 2017 IEEE International Conference on Big Data*, pages 183–192. IEEE, 2017.

[25] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, pages 776–789, 2018.

[26] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 61–74. USENIX Association, 2014.

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[28] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 463–479. USENIX Association, November 2020.

[29] McCallum John C. Price and performance changes of computer technology with time. http://www.jcmit.net/.

[30] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *Proceedings of the 5th International Conference on Learning Representations*, 2017.

[31] Tushar Khot, Ashish Sabharwal, and Peter Clark. SciTaiL: A textual entailment dataset from science question answering. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pages 5189–5197, 2018.

[32] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, pages 283–296. USENIX Association, February 2020.

[33] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric HPC system for deep learning. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 148–161. IEEE, 2018.

[34] Yonggang Li, Guosheng Hu, Yongtao Wang, Timothy Hospedales, Neil M. Robertson, and Yongxin Yang. DADA: Differentiable automatic data augmentation. *arXiv preprint arXiv:2003.03780*, 2020.

[35] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. *arXiv preprint arXiv:2002.11794*, 2020.

[36] Xiangru Lian and Ji Liu. Revisit batch normalization: New understanding and refinement via composition optimization. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, pages 3254–3263, 2019.

[37] Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. Fast autoaugment. In *Proceedings of the Advances in Neural Information Processing Systems 32*, pages 6665–6675. Curran Associates, Inc., 2019.

[38] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. *arXiv preprint arXiv:1805.00932*, 2018.

[39] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. DRAGON: Breaking GPU memory capacity limits with direct NVM access. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 32:1–32:13. IEEE, 2018.

[40] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[41] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *arXiv preprint arXiv:2007.06775*, 2020.

[42] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. Unified memory in CUDA 6.0: a brief overview of related data access and transfer issues. *Tech. Rep. TR-2014–09, University of Wisconsin-Madison*, 2014.

[43] Newegg.com. https://www.newegg.com/.

[44] NVIDIA. Training with mixed precision. https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html.

[45] Intel Optane SSD 905P series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-905p-series.html.

[46] Yongjin Park and Manolis Kellis. Deep learning for regulatory genomics. *Nature Biotechnology*, 33(8):825, 2015.

[47] David A. Patterson. Lecture 20: Domain-specific architectures and the google TPU, UC Berkeley CS152 Computer Architecture and Engineering. http://www-inst.eecs.berkeley.edu/~cs152/sp19, 2019.

[48] PCI-SIG. PCI-SIG® member companies announce support for the PCI express® 5.0 specification. https://pcisig.com.

[49] Samsung PM1725b NVMe SSD. http://image-us.samsung.com/SamsungUS/PIM/Samsung_1725b_Product.pdf.

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

[51] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019.

[52] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250v3*, 2016.

[53] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, volume 33, page 4780–4789. Association for the Advancement of Artificial Intelligence, 2019.

[54] Jerome Revaud, Minhyeok Heo, Rafael S. Rezende, Chanmi You, and Seong-Gyun Jeong. Did it change? Learning to detect point-of-interest changes for proactive map updates. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4081–4090. IEEE, 2019.

[55] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 18:1–18:13. IEEE, 2016.

[56] Samsung Semiconductor. http://www.samsung.com/semiconductor/.

[57] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

[58] Intel storage performance development kit. http://www.spdk.io/.

[59] Peng Sun, Yonggang Wen, Ruobing Han, Wansen Feng, and Shengen Yan. GradientFlow: Optimizing network performance for large-scale distributed DNN training. *IEEE Transactions on Big Data*, pages 1–1, 2019.

[60] Aarne Talman, Anssi Yli-Jyrä, and Jörg Tiedemann. Sentence embeddings in NLI with iterative refinement encoders. *Natural Language Engineering*, 25(4):467–482, 2019.

[61] TensorFlow. Data augmentation. https://www.tensorflow.org/tutorials/images/data_augmentation.

[62] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. SuperNeurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 41–53. ACM, 2018.

[63] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, page 7686–7695. Curran Associates Inc., 2018.

[64] Han Xiao. Bert-as-service. https://github.com/hanxiao/bert-as-service, 2018.

[65] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1(4):216–222, 2018.

[66] Chih-Chieh Yang and Guojing Cong. Accelerating data loading in deep neural network training. In *Proceedings of the 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics*, pages 235–245. IEEE Press, 2019.

[67] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237v2*, 2020.

[68] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-batch training for LSTM and beyond. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019.

[69] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2018.

[70] Samsung Z-SSD SZ985. https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD_SZ985_1804.pdf.

[71] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. NVMMU: A non-volatile memory management unit for heterogeneous GPU-SSD architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation Techniques*, pages 13–24. IEEE, 2015.

[72] Bojian Zheng, Abhishek Tiwari, Nandita Vijaykumar, and Gennady Pekhimenko. Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training. *arXiv preprint arXiv:1805.08899v5*, 2019.

[73] Jingbo Zhou, Qi Guo, H. V. Jagadish, Lubos Krcal, Siyuan Liu, Wenhao Luan, Anthony Tung, Yueji Yang, and Yuxin Zheng. A generic inverted index framework for similarity search on the GPU. In *Proceedings of the 2018 IEEE 34th International Conference on Data Engineering*, pages 893–904. IEEE, 2018.

[74] Ligeng Zhu, Ruizhi Deng, Michael Maire, Zhiwei Deng, Greg Mori, and Ping Tan. Sparsely aggregated convolutional networks. In *Proceedings of the European Conference on Computer Vision*, pages 186–201, 2018.