Behemoth: A Flash-centric Training Accelerator for Extreme-scale DNNs

Shine Kim, *Seoul National University and Samsung Electronics;* Yunho Jin, Gina Sohn, Jonghyun Bae, Tae Jun Ham, and Jae W. Lee, *Seoul National University*

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

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.

Behemoth: A Flash-centric Training Accelerator for Extreme-scale DNNs

Shine Kim[†‡*] Yunho Jin[†*] Gina Sohn[†] Jonghyun Bae[†] Tae Jun Ham[†] Jae W. Lee[†]
[†]*Seoul National University* [‡]*Samsung Electronics*

Abstract

The explosive expansion of Deep Neural Networks (DNN) model size expedites the need for larger memory capacity. This movement is particularly true for models in natural language processing (NLP), a dominant application of AI along with computer vision. For example, a recent extreme-scale language model GPT-3 from OpenAI has over 175 billion parameters. Furthermore, such a model mostly consists of FC layers with huge dimensions, and thus has a relatively high arithmetic intensity. In that sense, an extreme-scale language model does not suit well to the conventional HBM DRAM-based memory system that lacks capacity and offers extremely high bandwidth. For this reason, we propose to pair the neural network training accelerator with the flash-based memory system instead of the HBM DRAM-based memory system. To design the effective flash-based memory system, we optimize the existing SSD design to improve the SSD bandwidth as well as endurance. Finally, we evaluate our proposed platform, and show that Behemoth achieves 3.65× cost saving over TPU v3 and 2.05× training throughput improvement over the accelerator attached to a commercial SSD.

1 Introduction

Deep Neural Networks (DNNs) have become pervasive in various application domains. Early DNN models demanded only high computation, but recent models additionally require increasing memory capacity with continued scaling of DNNs. This is especially true for Natural Language Processing (NLP) models [5, 18, 39, 40, 44, 54], targeting problems including language translation [2, 50, 65], text generation [5, 53, 59] and summarization [35, 41, 69], and sentiment analysis [18, 40].

This advent of extreme-scale NLP models (with more than several billion parameters) is one of the most important recent breakthroughs in DNNs. Transformer [66] introduced in 2017 demonstrated that neural networks could substantially outperform existing NLP techniques, and the introduction of BERT [18] showed that the concept of training the Transformer language model with a large corpus could produce a versatile language model that can be utilized for various natural language processing tasks. Following BERT, many Transformer-based NLP models [4, 5, 16–18, 31, 34, 39, 40, 44, 54, 59, 66, 68, 72] have emerged. Specifically, GPT-3 [5], one of the most recent language models, has established state-of-the-art performance in various NLP tasks.

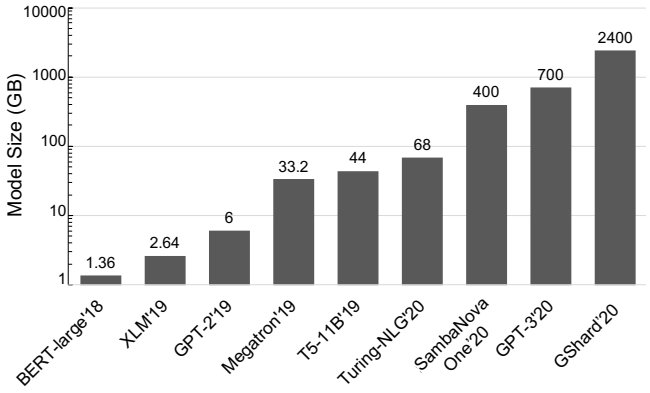

Figure 1: Trends of model size scaling with large NLP models

These NLP models explosively expand their sizes, taking hundred billions of parameters. Figure 1 shows that the size of the model has increased by more than 1000× over the last two years. For example, GShard [39] from Google contains roughly 2.4TB of parameters. This demand for memory capacity has been partly satisfied by simply supplying more memory. However, relatively stagnant DRAM scaling cannot keep pace with this increase in the DNN model size. Therefore, the solution of merely augmenting the system with extra DRAM is impractical.

It is impossible to process these models in a data parallel manner on the conventional hardware because it would require each device to hold the entire model [60]. There are mainly two solutions to this capacity problem. The first approach is to simply discard some computation results in the forward path and recalculate them during the backward path [8]. Unfortunately, this approach can incur a substantial

*These authors contributed equally to this work.

amount of extra computations. The other approach is the utilization of the model parallelism. This technique divides the model into multiple partitions and distributes them across multiple devices so that the system as a whole can accommodate the model. Following the GShard example from the previous paragraph, at least 75 devices with 32GB of memory [27, 64] are needed to run this model. Unfortunately, model parallelism comes with its inherent drawbacks. The dimensions and types of layers in a model are not identical. Thus careful load balancing of partitioned models is required. Additionally, stalls due to dependency may arise, and extra inter-node communication may be required to exploit pipeline parallelism.

To tackle this capacity problem differently, we first analyze the characteristics of those emerging NLP models. Our analysis reveals that, unlike conventional models, these extreme-scale NLP models consume huge memory proportional to the parameter size, but do not fully utilize the bandwidth of high-performance DRAM (e.g., high bandwidth memory (HBM)) because of a much higher degree of data reuse. This high arithmetic intensity stems from the huge model sizes, as parameters are shared by much more input elements. Thus, we have identified the opportunity to use high-capacity, low-performance NAND flash instead of low-capacity, high-performance HBM to train these extreme-scale NLP models.

Thus, we propose Behemoth, a NAND flash-centric training accelerator targeting extreme-scale NLP models. Behemoth allows those NLP models to be trained in a *data parallel* manner, where the training data set (not the model) is partitioned across multiple devices. To satisfy the computation, memory, and bandwidth requirements simultaneously, Behemoth integrates one Weight Node with multiple Activation Nodes. Like the parameter server in a data-parallel distributed system, Weight Node is responsible for feeding the Activation Nodes with weight data for each layer and reducing the weight gradients produced from them. Activation Nodes are the actual worker nodes processing each layer of the model. These nodes are composed of a DNN-specific Compute Core and enhanced large NAND flash memory, which can feed the core in time and store the data generated during the training process. The need for enhanced high-bandwidth flash memory is engendered by large data size and high-throughput Compute Cores. The NAND flash memory bandwidth can be scaled by increasing the number of channels and chips. However, in order to deliver the required performance of the DNN training workload, for example, tens of GB/s or more, the bottleneck caused by firmware must be resolved. Behemoth provides a performance scalable flash memory system for DNN training by hardware automation of the write datapath in a controller. Furthermore, Behemoth drastically extends the endurance of NAND by leveraging tradeoffs between retention time and endurance (P/E cycles) of NAND flash.

To summarize, our contributions are listed as follows:

- We carefully analyze the memory capacity problem that arises when training extreme-scale DNN models and identify new opportunities to leverage NAND flash devices, replacing expensive DRAM devices.
- For efficient training of these models, we present Behemoth, a novel flash-centric training accelerator targeting those models.
- To satisfy the bandwidth and endurance requirements of DNN training, we propose Flash Memory System (FMS) for Behemoth, which provides both high bandwidth and high endurance.

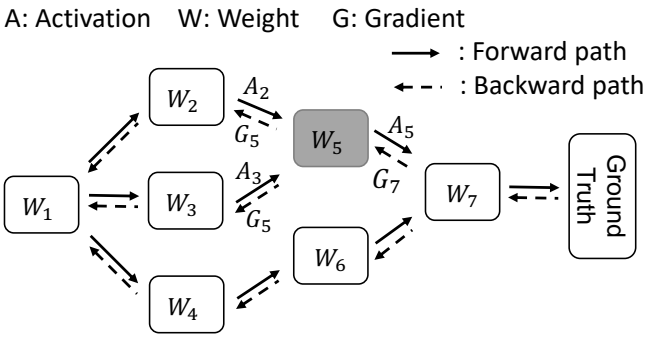

Figure 2: DNN training process and dataflow. The solid line represents the forward path in Layer 5 where A_2 and A_3 are provided as inputs. The dotted line depicts the backward path of the same layer, where A_5 is retained with W_5 and G_7 is received to compute G_5.

2 Background and Motivation

2.1 DNN Training

DNN training is a process where a neural network model utilizes a training dataset to improve its performance (e.g., accuracy) by updating its parameters. It is essentially a repetitive process of matrix operations. Both activations and weights, represented as matrices, are multiplied and added in every layer. Figure 2 describes an *iteration*, where activations follow the path predefined by the model, repeating the forward and the backward path. The training process is deterministic as the process follows a predefined forward and ensuing backward path. It is also iterative in that the paths are repeated until the desired result is generated. One set of a forward and backward path is called, surprisingly, *iteration*, and the set of inputs being processed in the iteration is termed *batch*.

In the forward path, input activations are passed to a layer, as shown in Figure 2. Upon receiving the activations A_2 and A_3, they are multiplied with the weight W_5 and generate output activation A_5. The output activation is retained in the layer for use in the backward path and sent as input to the next layer in the forward path. This process is repeated until the last layer. The output of the last layer is compared with the ground truth to calculate the error. This error is fed back to the last layer, thus starting the backward path. In this path, the gradients

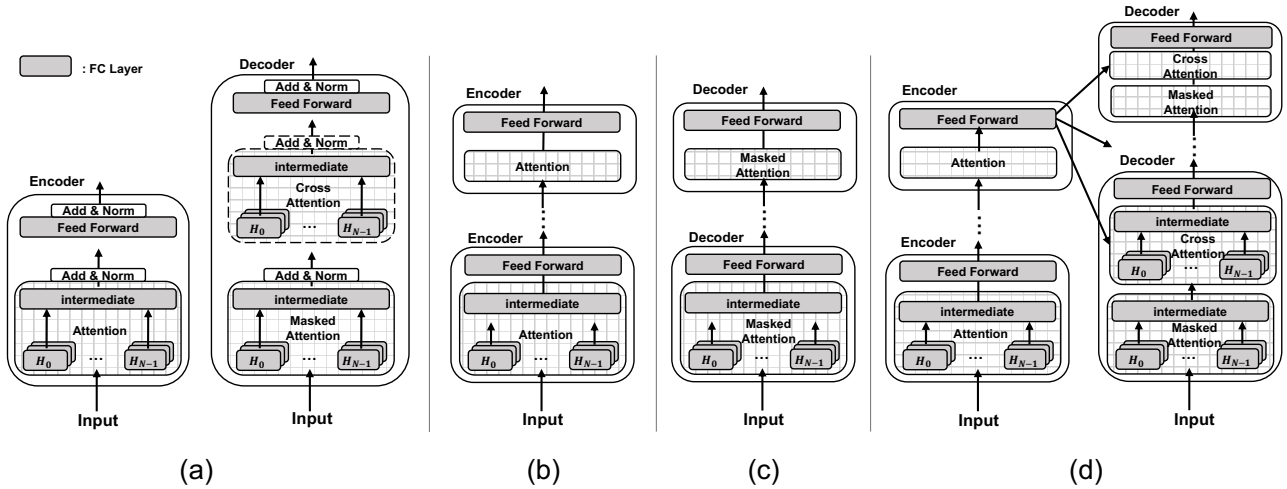

Figure 3: Simplified view of Transformer-based models. (a) Transformer block as a whole which consists of an encoder block(left) and a decoder block(right). Each H_n represents the n^{th} head in the attention layer which performs matrix multiplication to calculate a vector which is concatenated in the n^{th} position in the resulting vector. (b) BERT-like models where only the encoder part of the transformer is used. (c) GPT-like models where only the decoder part of the transformer is used. The main difference from (b) is that these models typically adopt masked attention, where the attention operation can only attend tokens appearing before the current one. (d) T5-like models where both encoders and decoders are used. Unlike those illustrated in (b) and (c) they have cross-attention layers in the decoder modules. The final output of encoders in T5 is distributed across all cross-attention layers in their decoders.

of both activations and weights with respect to the error are created. Input gradients are then propagated backward, generating weight gradients and updating weights along the way. Every DNN model training follows this pattern, regardless of which application it targets. However, the size and structure of the models differ by their usage. There are diverse application domains of DNNs, such as natural language processing (NLP), reinforcement learning (RL), computer vision (CV), and so on. This work primarily focuses on NLP.

2.2 Extreme-scale Language Models

Many of the emerging, extreme-scale NLP models share the same internal structure. Although the details may vary, all of these NLP models essentially consist of Transformer blocks shown in Figure 3(a). Specifically, some NLP models (e.g., BERT [18], RoBERTa [44], BART [40]) are constructed by stacking the encoder blocks of the Transformer model as shown in Figure 3(b), and some other NLP models (e.g., GPT-2 [53], GPT-3 [5]) are constructed by stacking the decoder blocks of the Transformer model as shown in Figure 3(c). Finally, as in the original Transformer, some models(e.g., T5 [54], Transformer-XL [17]) have stacked encoder blocks followed by extended decoder blocks as shown in Figure 3(d). Conceptually, encoder blocks can focus on relevant parts of the input sentence through attention layers. As a result, encoder-only models are mostly utilized for comprehension tasks (e.g., sentiment analysis,question-answering). Decoder blocks contain a masked attention layer, which is identical to the encoder's attention layer except that words coming after the current position are masked. Based on such characteristics, many decoder-only models are utilized for text generation tasks. In models that exploit both encoder and decoder blocks, a cross-attention layer is added to the decoders, which helps the decoder focus on the input sentence's related positions by utilizing the encoder block's output. Such models are often utilized for tasks like translation.

One notable characteristic of these Transformer-based language models is their gigantic sizes. For example, GPT-3 has 175 billion parameters, and the parameter size has been scaled by over 1000× over the past two years as shown in Figure 1. This implies that the model size will scale even further in the future. The gigantic size of these emerging language models brings many unique challenges to the existing neural network processing system. One of the most notable ones is the *memory capacity wall*, explained in the following.

2.3 Challenges for Extreme-scale Language Model Training

As stated above, the GPT-3 model has 175 billion parameters, where each parameter is represented in the FP16 format. In such a case, it takes 350GB of storage to store the parameters for this model. The storage cost is worse in training since training requires extra storage to buffer each layer's output activations as well as weight gradients. Here, the size of the weight gradients is identical to the weights themselves and thus cost 350GB. The sum of output activation sizes for the GPT-3 model is 43.7GB for a single input sequence with 2048 tokens, and this linearly increases with the number of input sequences in a single batch (i.e., batch size). For example, if one wants to train the GPT-3 with a batch size of 32, it will

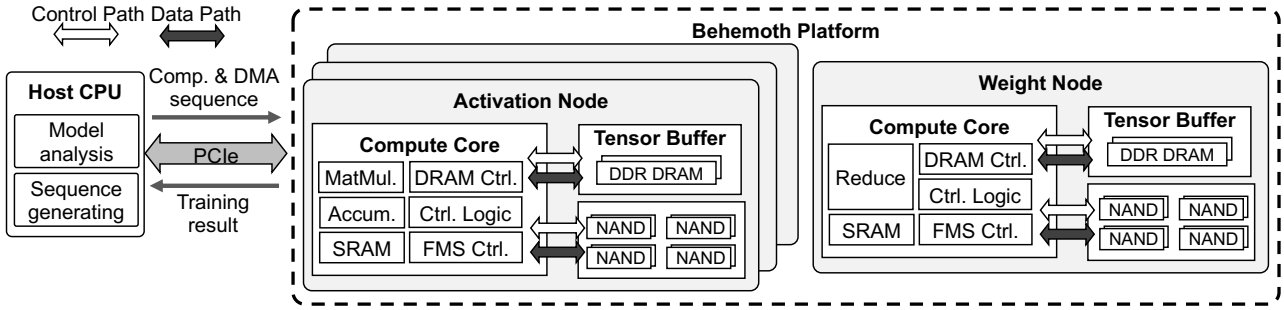

Figure 4: Behemoth architecture

require around 2.1TB storage space.

Unfortunately, conventional DNN training platforms such as NVIDIA GPUs or Google TPUs are equipped with HBM, which offers very limited storage capacity. For example, a single TPU chip, as well as a single V100 GPU only has (a maximum of) 32GB memory space. In order to train the GPT-3 model on these platforms, a minimum of 66 devices are required. In terms of computation, this is not a significant problem since training an extreme-scale model like GPT-3 in a reasonable time frame requires a very large computational capability, which often exceeds that of the 66 GPUs or TPU chips. However, the inefficiency here is that existing platforms such as TPU chips or GPU comes with an unnecessarily expensive memory system that is inadequate for the large-scale language model training.

As shown in Figure 3, large-scale, Transformer-based language models are mostly piles of fully-connected (FC) layers. Here, the dimensions of the FC-layers are very large. For example, a feedforward layer in the encoder/decoder block includes matrix multiplication between a 2048×12288 matrix and a 12288×49152 matrix to process a single input. In such cases where the matrix dimensions for the fully-connected layer are large, each value in the matrix is reused many times, and thus the layer ends up requiring a relatively small number of memory accesses compared to the amount of computation (i.e., the arithmetic intensity is low). Specifically, if the input matrix size for the FC layer is $m \times n$ and the weight matrix for the FC layer is $n \times k$, the amount of required multiply-accumulate (MAC) operations is mnk, and the amount of data that needs to be communicated from/to memory is $mn + nk + mk$. Thus, a larger m, n, or k increases the ratio of mnk to $mn + nk + mk$. When $m = 2048 * 32, n = 12288, k = 49152$ as in the feedforward layer of the encoder/decoder block processing 32 inputs (i.e., sequence length = 2048, batch size = 32), the operation requires 73.728 TFLOP for this matrix multiplication, and the total amount of data transfer from/to memory is 9.26GB, assuming FP16 datatype. For a single TPU chip, which can perform 105 TFLOP per second, this matrix multiplication takes 0.7 seconds. Since the TPU v3 is equipped with two HBM memory channels whose aggregate bandwidth is 600GB/s, this is enough time for the chip to transfer 420 GB. However, the operation only requires 9.26GB data transfer with the memory, grossly underutilizing the memory bandwidth. Thus, it is critical to match the arithmetic intensity of the models and the compute-to-memory (disk) bandwidth ratio of accelerators. Note that the released pretrained versions of smaller-scale Transformer-based models such as BERT and GPT-2 do not support processing of such long sequences, thus featuring lower arithmetic intensities.

This analysis of bandwidth underutilization indicates that HBM is not the ideal system for this workload. A similar argument applies for NVIDIA V100 GPU, which pairs some 112 TFLOPS with an HBM memory system having 900GB/s aggregate bandwidth.

Our Work. Observing this significant memory bandwidth underutilization, we propose to utilize the *flash memory system* (FMS) to design a more cost-effective large-scale language model training platform. However, naively replacing the HBM memory-system to a commodity SSD is not the right solution. In order to architect an efficient FMS for language model training, several challenges need to be addressed. First, an SSD has extremely-low bandwidth, especially when the access pattern is not sequential. Even if the access pattern is sequential, the sustained write bandwidth is often substantially lower than the peak bandwidth due to SSD garbage collection operations (GC) [33, 55]. The second challenge is endurance. Because SSDs can only sustain a limited number of writes, utilizing SSD as a memory for the DNN training can significantly reduce the lifetime of SSDs. This problem becomes exacerbated when the access pattern is not sequential because random writes tend to increase the write amplification factor (WAF). Our work proposes solutions for these challenges and demonstrates that FMS can be effectively utilized for DNN training.

3 Overview of Behemoth

We design Behemoth to fully accommodate extreme-scale DNN models in a single node enabling data-parallel training. As illustrated in Figure 4, Behemoth consists of Compute Core, Tensor Buffer, and FMS. Compute Core is the computing substrate for training. Control Logic in Compute Core receives a sequence of commands from a host CPU and gener-

ates both computation and data transfer commands by parsing the sequence. The imminent weight and activation tensors are kept in the DRAM buffer (named Tensor Buffer) serving as a staging area. Upon receiving the read/write commands issued by the control logic, FMS controller executes the commands, retrieving and storing data in the NAND chips.

3.1 Training DNN Models on Behemoth

Most of the recent DNN frameworks employ a Python model. This model is pre-processed on the host CPU for analysis, extraction of layer information, and then generation of a sequence of commands that Behemoth can execute. This command sequence is communicated to Behemoth for execution.

Model Analysis. A user can define a DNN model to train using the PyTorch [52] format. In this model analysis step, information is collected about each layer's order, arguments used in the layer's operation, and input/output tensors to use. This step works similarly to the process of creating a static computation graph in Caffe [3] and TensorFlow [1].

Generating Command Sequences. Based on the collected model data, this step generates two types of command sequences: computation command sequence and direct memory access (DMA) command sequence. The DMA command sequence controls data transfers between Tensor Buffer and NAND flash devices. A DMA command includes fields about the direction of transfer (read/write), logical block address (LBA) of the NAND device, and Tensor Buffer address. The computation command sequence lists operation commands to perform on Compute Core. A computation command includes fields about the type of the layer (e.g., Fully-connected (FC), Convolution (Conv)), and the address in Tensor Buffer where the layer's input and output tensors will be stored. Both command sequences are transferred to Behemoth to be saved in the region for the non-volatile stream (NV-Stream in Section 4.1).

3.2 Hardware Components of Behemoth

Behemoth platform consists of a single Weight Node and multiple Activation Nodes to fully utilize the bandwidth between Tensor Buffer and NAND flash. Each node consists of Compute Core, Tensor Buffer, and NAND flash. Weight Node stores the weights of the target training model in the NAND flash. The Activation Nodes create activation tensors during forward propagation and store them locally in the NAND flash for reuse during backward propagation.

Compute Core. Compute Core is not bound to a specific DNN accelerator architecture. Thus, we assume a generic DNN accelerator that abstracts popular commercial/academic accelerators [9,10,24,27,58]. The DNN accelerator is specialized for DNN processing and performs matrix multiplication and addition for weights and activations. It consists of a 2D array of processing elements (PEs), where each PE can perform a single MAC operation every clock cycle. Behemoth assumes a weight stationary dataflow architecture [9], where weights are directly loaded from Tensor Buffer and kept in the local registers inside the PE. Every cycle, a new input is provided to the PEs from the SRAM buffer of Compute Core. This input is multiplied by the corresponding weight in the PE, and the result is accumulated. Once the computation is done, the output values are transferred to the SRAM buffer, and eventually to Tensor Buffer.

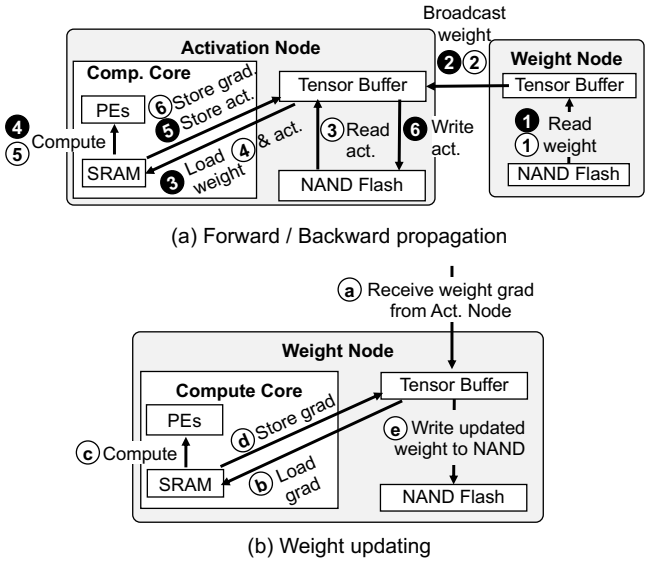

Figure 5: Example walk-through of Behemoth (a) Forward and backward propagation (b) Weight updating

Control Logic. Control Logic is responsible for sequencing computation commands and orchestrating data transfers between SRAM buffer, Tensor Buffer, and FMS. Specifically, it decodes the commands provided by the host CPU and inspects if this command can be scheduled (i.e., satisfies all dependencies). If so, Control Logic dispatches this command to Compute Core (if it is a computation command) or DRAM Controller or FMS Controller (if it is a DMA command) to initiate the requested DMA. Note that this is a very simple logic, which sequences the commands in order.

Tensor Buffer. Tensor Buffer is a DRAM region that serves as a staging area between Compute Core and FMS. The primary role of his buffer is to smooth the traffic between FMS and Compute Core simply. Thus, Tensor Buffer only stores temporal data and does not require persistence.

Flash Memory System (FMS). FMS is the main storage in Behemoth replacing the HBM in the conventional DNN accelerators (e.g., TPU). As in SSDs, this component includes a set of NAND chips. However, unlike the conventional SSD, it has a hardware-based FMS controller that replaces the flash translation layer (FTL) running on general-purpose cores. This component interfaces with Control Logic and transfers data to Tensor Buffer. The details of this component are explained in Section 4.

3.3 Example Execution Walk-Through

Figure 5(a) and (b) illustrate the process of forward and backward propagation in Behemoth. In what follows, we explain this process in greater detail.

Forward Propagation. Executing forward propagation in Behemoth consists of 6 steps, which can be overlapped. While computation is being performed on Activation Node, weights on Weight Node are prefetched. ❶ Executing a layer starts with reading the weights stored in the NAND flash of Weight Node into Tensor Buffer. ❷ The weight tensor is broadcasted and transferred to Tensor Buffer of Activation Node. ❸ Then, the weight tensor is loaded into on-chip SRAM, and ❹ computation begins. When computation is completed on Activation Node, ❺ the activation tensor stored in SRAM is copied to Tensor Buffer. ❻ Finally, the activation tensor in Tensor Buffer is written to NAND flash for reuse during backward propagation, and the weight tensor is deallocated from Tensor Buffer.

Backward Propagation. Figure 5(a) and (b) show the process of backward propagation (labeled with empty circles). It is divided into two parts. The first part is executed for each layer, and the second one only once at the end of each iteration. Like forward propagation, all steps are pipelined and operated in parallel. The processes of ① and ② are the same as forward propagation. Once the weight tensor is received from Weight Node, ③ Activation Node reads the activation tensor of the corresponding layer stored in the NAND flash during the forward propagation into Tensor Buffer. ④ Upon completion of loading the activation tensor, both activation and weight tensors are loaded into SRAM, and then ⑤ computation is started. When the operation is completed, ⑥ the resulting gradient tensor is stored in Tensor Buffer.

After the calculation of all layers is completed, ⓐ the final weight gradient tensor is transferred from Activation Node to Tensor Buffer of Weight Node. After confirming that all weight gradients have been received, ⓑ Weight Node loads the weight gradients into SRAM and ⓒ updates the training results of the iteration to the weights. ⓓ The updated weights are stored to SRAM and ⓔ written to the NAND flash for the next iteration.

3.4 DNN Model Coverage

Behemoth targets training workloads for extreme-scale models whose memory bandwidth requirement does not exceed the sustainable bandwidth of FMS. Suitability for other models can be determined by analyzing their arithmetic intensity. Our model analyzer can compare the arithmetic intensity of a model with Behemoth's compute-to-bandwidth ratio (i.e., FLOPS / GB/s) to check whether the model is provided with enough bandwidth from FMS [49].

The key enabler of FMS as a storage medium of tensors is a higher degree of data reuse resulting from long sequence length. State-of-the-art models that are capable of handling such long sequences have an extremely large size. Small-sized models exhibit much lower arithmetic intensity, hence not being the primary target of Behemoth. For example, small NLP models [18, 40, 53] have limited sequence length (e.g., 512). This is much smaller than 2048 for GPT-3. Conventional vision models such as ResNet heavily utilize convolution layers. Converting convolutions into matrix multiplications through convolution lowering [12] results in a dimension m that is much smaller than those of FC layers, hence failing to provide enough bandwidth from FMS. This issue can be mitigated to a certain extent by increasing the batch size, which in turn increases the degree of reuse for weight parameters.

Table 1: DNN training data types and multi-stream support

#: Stream name (Act. Node / Weight Node)	Persistency	Retention	Access permission	
			Host	Behemoth
1: NV-Stream (Training inputs / –)	Non-volatile	Years	Append-only seq. write	Read only
2: V-Stream (Activations / Interm. weights)	Volatile	Minutes	N/A	Read & Append-only seq. write
3: NV-Stream (– / Trained weights)	Non-volatile	Years	Read only	Read & Append-only seq. write

4 Architecting Specialized Flash Memory System (FMS) for DNN Training

As stated in Section 2, the main challenges in adopting NAND flash memories for the language model training is the limited bandwidth and the endurance of the NAND flash memories. This section presents our solution to the two challenges and explains FMS's implementation in detail.

4.1 Improving Effective Bandwidth of FMS

Modern NAND flash memory-based storage adopts a number of flash channels and ways to increase bandwidth and capacity. A host interface for the storage has also run a neck and neck race with the storage's internal bandwidth to meet the user's performance requirement. In terms of hardware bandwidth of the NAND flash-based storage, the interface speed and the number of NAND channels, as well as the number of NAND chips attached to a channel, define the maximum reachable speed of a NAND flash-based storage.

Technically, the bandwidth of a flash-based memory system can be improved by utilizing a large number of NAND channels as well as the sufficient number of NAND chips per channel to saturate the channel bandwidth. Indeed, some recent proposals [11, 25] demonstrate that it is possible to build a high-bandwidth NAND system by increasing the number of channels or the channel bandwidth itself. However, to fully utilize the high peak bandwidth of such a NAND device, one needs to i) make writing sequential as much as it can and ii) prevent the slow NAND firmware running on a general-purpose processor from being a bottleneck [11, 71].

Data type Separation. Generally, it is challenging to identify

Table 2: NAND block layout for a chip and multi-stream attributes of Activation Node

NAND Block Layout					Stream attributes	
Plane / PBN	0	1	. . .	7	Capacity	P/E cycle/ Retention
0 – 9	FTL Metadata (LBN2PBN map, PB metadata, etc)					
10 – 92	1: NV-Stream (training input)				249 GiB	50K / 1 year
93 – 671	2: V-Stream (activation data)				1737 GiB	2M / 1 day
672 – 682	Reserved blocks for bad block replacement					

a workload's data access patterns before it is executed. However, a DNN training accelerator (or NPU) has a deterministic data access pattern that can be statically analyzed. The DNN training accelerator accesses three types of data, each having a very specific characteristic as listed in Table 1.

First, Activation Node's FMS houses two types of data: training input data and the activation data. Here, training input data is a set of text data used as training inputs of the DNN model. This data is written by a host before the training starts and then discarded once the training finishes. Activation data are written by Compute Core of the FMS platform during a forward path of the training and then consumed during a backward path of the training. The data is not written or read by the host, and the life cycle of these data is very short (in order of seconds, or minutes at most) as they are lived only within a single iteration.

Similar to Activation Node, FMS of Weight Node also houses two types of data. First, it holds the final model weights, that is only updated at the end of the training (or after a certain number of iterations to checkpoint the intermediate weights), and then later read by the host CPU. Second, it stores intermediate model weights that are updated at the end of each iteration.

Since two data types housed in the same device (i.e., training input data vs. activation data in Activation Node; final trained weights vs. intermediate model weights in Weight Node) have completely different characteristics, it is beneficial to separate them to two logically isolated spaces as in multi-stream SSDs [30, 33, 55]. In particular, we employ two streams: the non-volatile stream (NV-Stream) and the volatile stream (V-Stream). Table 2 summarizes a block layout for a single NAND chip and capabilities of each data stream of Activation Node. Weight Node layout is the same, but only the capacity and physical block number (PBN) division are different from storing the weight result. The streams are physically separated by block address boundary, hence able to function as if each stream were an individual storage space, and thus each single stream can have their own logical address space, access permission, and allowed P/E cycle based on retention requirement, which is determined by characteristics of the DNN training data. This separation of different data types enables several useful optimizations as follows.

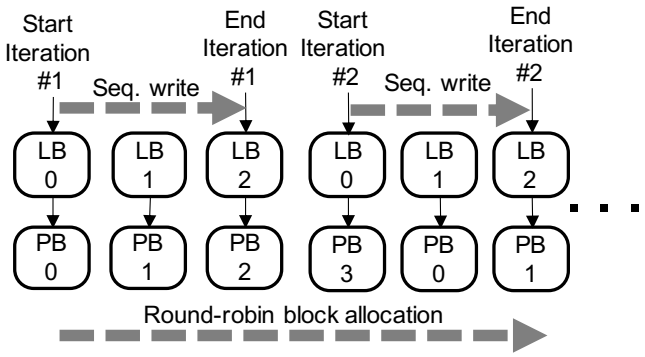

Figure 6: Sequential append-only writes with RR allocation

Lightweight Flash Translation Layer. Two major functionalities of the flash translation layer (FTL) are garbage collection (GC) and wear-leveling. However, since writes to each data for our FMS is guaranteed to be sequential, complicated garbage collection and wear-leveling are mostly unnecessary. Thus, we remove the FTL's garbage collection functionality and then replace the wear-leveling block allocator with a simple round-robin block allocator shown in Figure 6. For example, as shown in the figure, assume that FMS has four Physical Blocks (PBs), and the host writes three Logical Blocks (LBs) sequentially during a single DNN training iteration. During the first training iteration, FMS uses PB 0, 1, and 2 by mapping to LB 0, 1, and 2. And then, in the second training iteration, FMS allocates PBs according to Round-Robin (RR) policy from PB 3 to PB 0, 1 for writing to LB 0, 1, 2 of the host. This simple RR block allocation policy strictly levels wear of all NAND blocks. The utilization of this simple wear-leveling scheme as well as the removal of the garbage collection greatly simplified the FTL.

Hardware Automation of Write Path. Most modern commercial SSD controllers adopt a read automation feature that accelerates the read operation exploiting specialized hardware that substitutes (part of) the read path of the SSD firmware [11]. On the other hand, the write data path still relies on the firmware with high overhead or is merely partially replaced by hardware logic with substantial functional restrictions [28, 70]. This is mostly because the write data path is much more complex than the read path. For example, the write path needs to perform many additional operations compared to the read path. Specifically, it needs to i) reserve NAND blocks for the GC operations, ii) perform wear-leveling to ensure that all NAND blocks are used evenly, iii) guarantee data consistency among the internal R/W operations generated by the GC, the wear-leveling, and the user write commands, iv) manage metadata necessary for recovery from expected or unexpected power-reset, and v) handle exceptions for P/E failures.

However, we note that our FMS's common write data path does not need to perform many additional operations than the read path. It does not require a garbage collection and utilizes a very simple wear-leveling block allocator. Metadata

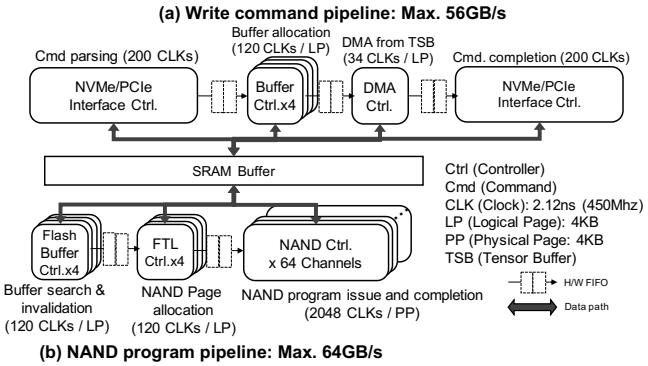

Figure 7: Automated write data path of FMS

management is not on a critical path and unnecessary for temporary data such as activation data and intermediate weight data. Finally, the exception handling is a rare event. Thus, it becomes relatively easy to automate the write data path by utilizing specialized hardware. By doing so, it is possible to prevent the firmware from being a bottleneck.

Figure 7 shows hardware pipeline stages for the write path of FMS and a timing of each pipeline stage. The automated write path is composed of (a) write command pipeline that transfers data from Tensor Buffer to an SRAM buffer in the FMS controller and (b) NAND program pipeline that programs data in the SRAM to NANDs. As shown in Figure 7, we carefully design each pipeline stage to meet a memory bandwidth requirement for DNN training. In particular, a buffer search/invalidation and a NAND page allocation stage of the NAND program pipeline, which was handled by the firmware of an existing SSD product [11], have been completely replaced with FMS controller logic.

Note that the hardware pipeline does not update the metadata necessary for persistence. For temporary data that accounts for the most portion of FMS, persistency support is not performance-critical as the iteration can be re-executed from the last checkpoint. For the data that needs the storage to be persistent, a user can make an explicit request (e.g., flush command [48] after writes) that initiates the firmware to ensure that the data is persistent.

4.2 Improving Endurance of FMS

The endurance of a NAND flash based storage relies on the program and erase (P/E) cycle for NAND blocks. The P/E operation wears the NAND block, accelerating the leakage of the electrons in the NAND cells. Additionally, such damage from the P/E cycles is cumulative and irreversible and gives rise to a myriad of read error bits, which cannot be corrected by an ECC engine of a storage controller.

FMS essentially utilizes a flash as a temporary buffer for the activation and intermediate weights. At a glance, it may seem like such a frequently re-programmed value will substantially affect the lifetime of the SSDs, which are often defined as the number of P/E cycles that a NAND cell can sustain. However, we argue that this is not the case (and present a quantitative analysis in Section 5.3).

Typically, each P/E cycle damages a NAND cell, and such a damage keeps reducing the retention time of the cell. Once the retention time falls below the guaranteed retention time (e.g., 1 year in consumer-grade SSDs [14]), the cell is considered having failed. At that point, the cell may not be suitable for storing the data for a long time; however, it is likely to be still sufficient to store the data that will only last for a few minutes. In light of device physics, the programmed NAND flash cells gradually lose their electrons from a floating gate over time, and in case a cell is damaged by the repetitive P/E cycles, the cell loses charge faster [26, 57]. However, with the low retention requirement, the cell can still maintain sufficient level of charges until the end of the retention time. In fact, many studies [6, 43, 45] already demonstrated that the SSD endurance (# of P/E cycles) is larger when the retention requirement is relaxed. Note that the benefits of reduced retention does not require additional hardware resources (e.g., more complex ECC engines or an extra over-provisioning space).

Considering that FMS (V-Stream data) requires only a few minutes (e.g., 5 minutes) of retention time that is almost five orders of magnitude smaller than a typical consumer-grade SSD, it is expected that the cell can sustain a substantially large number of P/E cycles before a cell's minimum retention time to fall below a few minutes.

5 Evaluation

5.1 Methodology

We evaluate our platform's effectiveness by i) comparing our platform's memory cost to conventional TPU-based DNN training system, and ii) comparing our platform with the specialized FMS to the hypothetical platform with conventional SSDs.

Simulation Framework. To model the performance of the Behemoth platform, we utilize MAESTRO [36] for Compute Core and MQSim [63] for modeling our FMS. Specifically, we utilize PyTorch [52] to obtain the layer dimensions of the large-scale language models, and then use that information on MAESTRO [36] to obtain the number of cycles that Compute Core (PE arrays with weight-stationary dataflow) needs for the computation of a specific layer in the language model. Then, based on this information, we generated the traces for our FMS and fed these traces to the MQSim (modified to support our proposed changes detailed in Section 4) to obtain the flash memory-related statistics. Both simulators are validated with NPU hardware RTL [9, 37] and a commercial SSD product [13], exhibiting an average of 5% errors [32, 36].

Workloads. We evaluate twelve workloads representing three types of widely adopted transformer models. Table 3 lists these models. As listed in Table 3, we evaluate our work with two types of models: (a) BERT/GPT-like and (b) T5-like. We

Table 3: DNN models evaluated with Behemoth. We use a sequence length of 2048 (tokens) for each model.

Model	Size	Total act. (GB)	Total weight (GB)	PFLOP
BERT/GPT3-like [5, 18]	1×1	44	350	2.15
	1×2	88	698	4.42
	1×4	175	1393	8.56
	2×1	88	1395	8.56
	2×2	175	2786	17.12
	2×4	349	5569	34.21
T5-like [54]	1×1	40	305	0.62
	1×2	80	609	1.25
	1×4	160	1218	2.49
	2×1	80	1218	2.49
	2×2	160	2436	4.99
	2×4	319	4871	9.97

enlarge the dimensions in FC layers of the models and/or stack more encoders/decoders, respectively, for diverse comparison. $W \times D$ notation is defined as W-fold enlarged FC layers dimension (width) and D times increased depth was implemented by stacking more encoders/decoders or transformers blocks. Our workloads present various Transformer-based models. Transformer is a key enabling primitive for DNN to advance the state-of-the-art in the domains of NLP [4, 5, 16–18, 31, 34, 39, 40, 44, 54, 59, 66, 68, 72], image detection [7, 19], point cloud [21], and recommendation systems [62]. All of these models can be classified into either BERT/GPT-like or T5-like.

The rationale behind binding BERT/GPT-like models is as follows. They do not use a combined transformer but separately utilize encoders and decoders. While they have certain different characteristics, such as in the computation process, their structure is identical, as described in Figure 3. Thus, the two systems' total activation and weight are equal when having the same number of parameters. The structure of the T5-like models makes it difficult to exactly match the number of parameters with the encoder-only or decoder-only models. In turn, they show the different sizes of activation and weight. The sequence length of 2048 tokens can be interpreted as roughly 2048 words in a sequence. The sequence of 2048 tokens comprises a batch size of 1, and the activation size is calculated for one batch.

5.2 Memory Cost Evaluation

Baseline NPU with HBM DRAM. In order to train very large scale language models like GPT3, existing HBM-based neural processing accelerators need to be configured in a model-parallel manner. In such a configuration, each TPU is assigned a portion of the model (i.e., a distinct set of consecutive layers). Once a neural processing accelerator finishes the computation for the layers it is assigned to, it passes its outputs to the other neural processing accelerator in charge of the following layers. This model parallelism makes it possible to train a very large model that does not fit in a single neural processing accelerator's HBM-based memory. One notable drawback of this approach is the difficulty in load-balancing.

Table 4: Platform configurations for the cost evaluation of Behemoth.

NPU Parameters		
Number of cores	16 cores (52.5 TFLOPs per core)	
Number of PEs	524,288	
Peak throughput	840 TFLOPs	
Host I/F conf.	PCIe Gen4 × 32 lane [51]	
Memory Parameters		
	Resembled TPU [27]	**Behemoth**
Buffer conf.	16GB HBM	16GB DDR4 DRAM + 2TB NAND flash
Peak bandwidth	300GB/s	50GB/s
Compute Parameters		
Parallel comp. method	Model parallelism	Data parallelism

For example, training GPT-3 requires a minimum of 393GB storage, which translates to a 24-stage pipeline assuming that a hardware corresponding to each stage has a 16GB HBM memory system. Depending on the nature of the model, partitioning the model into 24 slices in a load-balanced manner may be very difficult, if not impossible. For the cost comparison, we configure an NPU resembling the structure of TPU, as shown in Table 4. A single device here has 16 compute cores, each having 52.5TFLOPS peak throughput. Each of these cores is attached to a single, 16GB HBM memory whose peak bandwidth is 300 GB/s. To achieve sufficiently high throughput, multiple copies of these devices are utilized in parallel.

Behemoth with FMS. Unlike the baseline, Behemoth platform utilizes data parallelism, which enables the complete model to be trained on a single device, and thus does not suffer from load imbalance issues. We configure the single device for the cost comparison to having 16 compute cores, each having 52.5 TFLOPS peak throughput as in the baseline NPU. However, instead of HBM, 16 computation cores in Behemoth device share a single FMS with 2 TB capacity and 50 GB/s peak bandwidth. In addition to this device utilized as an Activation Node (see Figure 5), there is a separate device utilized as a Weight Node. However, since there will be many Activation Nodes that share a single Weight Node, the cost of the Weight Node is amortized. Note that we carefully configured 16 cores to share a single FMS. To determine the number of cores to share a FMS, we selected the maximum number of cores that satisfies the following criteria: (a) the size of the data fits inside our storage, and (b) the data transfer between the FMS and compute cores can be completely hidden.

Cost Evaluation. Figure 8 demonstrates the difference of memory cost between Behemoth and TPU v3 [27], a popular training accelerator deployed by Google. We assumed that the user utilizes the 432 Behemoths and 864 TPUs that are just enough to train each workload in 10 days. For this calculation, we assumed HBM device cost to be $20/GB [23], DDR4 DRAM device cost to be $4/GB [47], and flash device cost to be $0.67/GB. Note that the cost of Behemoth SSD, using 128Gb V-NAND based SLC, was set as four times a commercial SSD price (0.167$/GB) which uses 512Gb V-

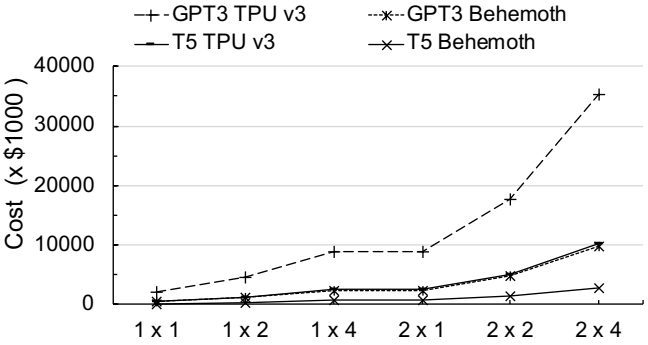

Figure 8: Memory cost comparison between TPU v3 [27] and Behemoth. $W \times D$ in the figure illustrates that the dimension of each layer is increased by W times and the number of layers is increased by D times.

NAND TLC [15]. It can be seen from the figure that the cost gap between the two systems increases commensurate with model size. The maximum difference of $25.7M for BERT/GPT3-like models and $7.5M for T5-like models.

5.3 FMS Evaluation

Configuration. We compare a DNN training platform utilizing BehemothFMS and the other utilizing the conventional storage using commodity SSDs. The configuration details are tabulated in Table 5.

Impact of FMS on Training Throughput. We compare the training throughput of a DNN training platform utilizing BehemothFMS and one that utilizes the commodity SSDs. As shown in Figure 9, the DNN training platform with BehemothFMS performance is close to the ideal case where there is zero overhead from memory system accesses. On the other hand, a DNN training platform with commodity SSDs often achieves much lower training throughput in many workloads. This is because the baseline SSD achieves limited throughput bottlenecked by an SSD firmware. To be exact, since a write data path of the baseline SSDs requires a minimum of 1.45μs to write a 4KB page, a single commodity SSD device's write throughput is limited to about 2.75GB/s despite its high aggregate channel bandwidth or external interface bandwidth. It aggregates four SSD device's throughput results in about 11.0GB/s bandwidth, which is substantially smaller than the 50GB/s bandwidth that Behemoth can provide. Note that the speedup of BehemothFMS is a little lower on wider models (e.g., 2×1, 2×2, 2×4). This is because wider models have even higher data reuse (see Section 2), thus requiring even less memory or storage bandwidth.

Endurance for Training Workloads. Figure 10 shows lifespan of tensors that are generated during DNN training. As shown in the figure, all tensors created during the DNN training have a lifespan of up to a single iteration period [22]. Therefore, the longest lifespan of tensors equals the retention time necessary for the V-Stream of FMS: 41 sec. Based on the previous studies [6, 43, 45] that demonstrate the number of P/E cycle of NAND can be increased by at least 40× [6] (up-to 600× [43]) if 1-year retention is reduced to 3 days, we also conservatively assume that the P/E cycle of our SSD is improved by 40 times, despite our retention requirement (i.e., less than a minute) is much shorter than three days. The Samsung Z-SSD [13] has 50K P/E cycles, and improving its P/E cycles by 40× results in the 2M P/E cycles. Two million P/E cycles on 1.85TB storage for V-Stream of FMS translates to the 3,700,000 TBW (TeraBytes Written). Considering that Behemoth FMS can sustain up to 17.6GB/s write bandwidth on average for T5-like models, Behemoth FMS is guaranteed to function for 6.6 years (i.e., 3.7M TBW / (17.6GB/s) = 210M seconds = 6.6 years). As shown in Figure 11, this is an even longer period than the 5-year warranty of typical commercial SSDs. Here, note that we assume the write amplification factor (WAF) of one because Behemoth only performs monotonic sequential writes and reads during the entire DNN training iteration without GC operations [29] as shown in Figure 12.

Table 5: FMS and conventional storage configuration.

Storage Parameters		
	Behemoth FMS	**Baseline SSD**
NAND Configurations	2TB, 64 channels, 2 chips/channel, 1 die/chip	500GB, 16 channels, 2 chips/channel, 1 die/chip
Channel Speed Rate	1200MT/s (MT/s: Mega Transfers per Second [20])	
NAND Structure	128Gb SLC / die: 8 planes / die, 683 blocks / plane, 768 pages / block, 4KB page	
NAND Latency	Read: 3μs, Program: 100μs, Block erase: 5ms	
Buffer Configurations	SRAM 16MB: 6MB for FTL metadata, 10MB for I/O buffer	DRAM 512GB: FTL metadata SRAM 8MB: I/O buffer, GC Buffer
FTL Schemes	Block mapping	Page mapping, Preemtible GC [38]
OP ratio	N/A	7%
Firmware Latency	N/A	Write: 1.45μs / a page (4KB)
Contoller Latency	Read: 1.93μs / an NVMe Cmd, Write: 1.18μs / an NVMe Cmd	Read: 1.93μs / an NVMe Cmd

6 Related Work

Heterogeneous Memory System for Tensor Management. Due to the memory capacity wall, researchers train the model with a limited number of parameters and batch sizes that the memory capacity allows, or parallelize the model by distributing the data needed for computation across multiple DNN training devices. However, training on a small model shows low accuracy, and the distributed learning of large models through multiple devices is a waste of memory bandwidth compared to memory capacity usage in several cases. To address the memory capacity wall, several proposals [22, 56, 67]

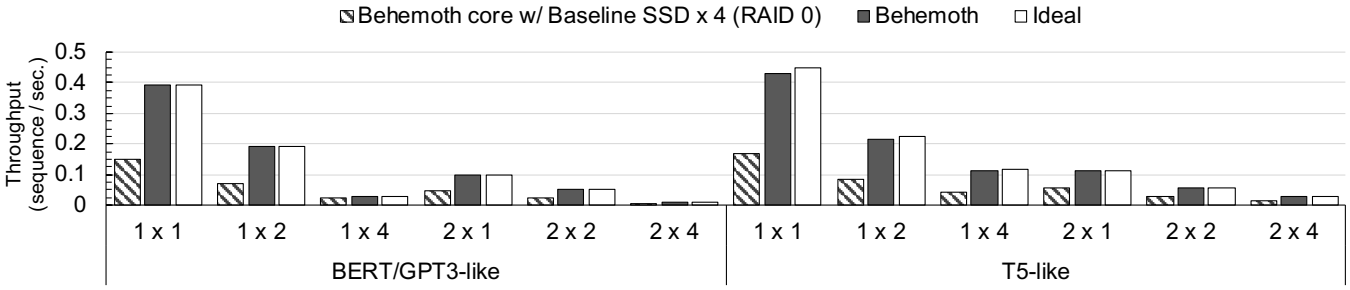

Figure 9: DNN training throughput of 432 Behemoths over various model sizes.

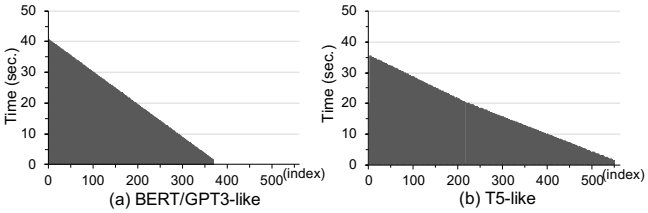

Figure 10: Tensor lifespan

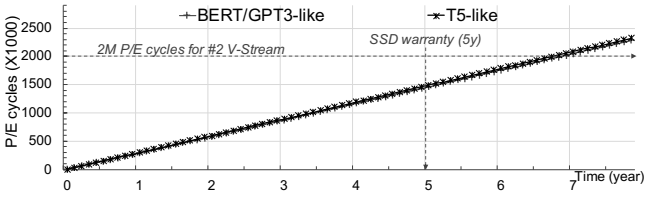

Figure 11: Behemoth FMS endurance

introduce effective memory management techniques that migrate tensors in a heterogeneous memory system. HALO [22] analyzes the hotness and lifetime of tensors and constructs an offloading schedule for each tensor, which focuses on how to migrate and place tensors across the heterogeneous memory nodes. vDNN [56] offloads tensors to the host memory during the forward pass and prefetches tensors from the host memory to be sent to the GPU during the backward pass. SuperNeuron [67] partially adopts the idea of vDNN by only offloading/prefetching marked tensors and recomputing unmarked tensors during the backward pass. These researches utilize another DRAM memory as offloading media. However, this is more of a one-time solution since the large language model does not fit in the host main memory. Behemoth adopts the idea of offloading and prefetching tensors according to their lifetime and dependency, but completely overcomes the memory capacity wall by using a dense NAND-based flash memory system configured for the high bandwidth.

Storage-Centric Machine Learning System. Several proposals [42, 46, 61, 70] use SSD as a secondary storage in the compute core to run large-scale applications. BLAS-on-flash [61] builds a library to achieve efficient flash memory speed in computing machine learning algorithms (e.g., ISLE, XML) that are used on large datasets. Cognitive SSD [42] constructs an engine designed for unstructured data retrieval involving DNN inference in flash memory devices. In contrast, Behemoth proposes the flash-based memory system for the extreme-scale neural-network language models with over hundreds of billions of parameters.

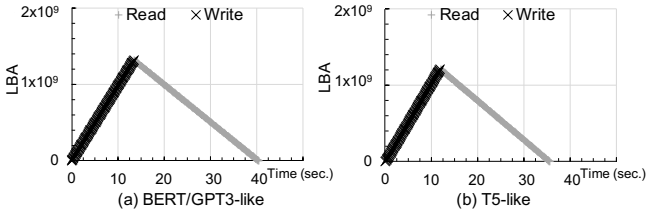

Figure 12: Block access pattern of DNN training workloads running on Behemoth FMS

7 Conclusion

Recent DNN models are getting wider and deeper, increasing the memory requirements for training. This trend is especially obvious in NLP with extreme-scale models showing exponential growth in its size. However, conventional DNN training platforms such as NVIDIA GPUs or Google TPUs provide insufficient storage capacity, which leads to excessive cost and memory bandwidth underutilization. To address this problem, we propose Behemoth, a flash-based memory system for a cost-effective training platform targeting extreme-scale DNN models. Behemoth overcomes the low-bandwidth and endurance problem of SSDs by separating data according to their characteristics. This enables a simplified firmware and hardware automation of the write path, which significantly improves the bandwidth. Furthermore, by exploiting the much shorter required retention time, we also showed that the SSD could be safely utilized for over six years. In the end, this Behemoth flash memory system based DNN training platform achieves a much smaller memory system cost than the conventional DNN training platform utilizing HBM devices.

Acknowledgments

We thank Suparna Bhattacharya for shepherding this paper. We also thank Young H. Oh for his help with DNN training. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2020R1A2C3010663). 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] Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. Unsupervised Neural Machine Translation. *arXiv e-prints*, page arXiv:1710.11041, October 2017.

[3] BAIR. Caffe: Deep learning framework. https://caffe.berkeleyvision.org.

[4] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv e-prints*, April 2020.

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

[6] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrian Cristal, Osman S. 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, 2012.

[7] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-End Object Detection with Transformers. *arXiv e-prints*, page arXiv:2005.12872, May 2020.

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv e-prints*, page arXiv:1604.06174, April 2016.

[9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, page 367–379. IEEE Press, 2016.

[10] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DianNao family: Energy-efficient hardware accelerators for machine learning. *Commun. ACM*, 59(11):105–112, October 2016.

[11] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel DG Lee, Jin-Hyeok Choi, and Jaeheon Jeong. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 338–340. IEEE, 2018.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv e-prints*, page arXiv:1410.0759, October 2014.

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

[14] Samsung SSD 980 PRO. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/980pro/.

[15] SSD price: Samsung SSD 970 EVO. https://www.amazon.com/Samsung-970-EVO-1TB-MZ-V7E1T0BW/dp/B07BN217QG/ref=sr_1_2?dchild=1&keywords=970+evo&qid=1600645757&sr=8-2.

[16] Zihang Dai, Guokun Lai, Yiming Yang, and Quoc V. Le. Funnel-Transformer: Filtering out Sequential Redundancy for Efficient Language Processing. *arXiv e-prints*, June 2020.

[17] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv e-prints*, January 2019.

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

[19] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and

Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv e-prints*, page arXiv:2010.11929, October 2020.

[20] Alan Freedman. MT/sec. *The Computer Desktop Encyclopedia*. https://www.computerlanguage.com/results.php?definition=MT/sec.

[21] Meng-Hao Guo, Jun-Xiong Cai, Zheng-Ning Liu, Tai-Jiang Mu, Ralph R. Martin, and Shi-Min Hu. PCT: Point Cloud Transformer. *arXiv e-prints*, page arXiv:2012.09688, December 2020.

[22] Myeonggyun Han, Jihoon. Hyun, Seongbeom. Park, and Woongki. Baek. Hotness- and lifetime-aware data placement and migration for high-performance deep learning on heterogeneous memory systems. *IEEE Transactions on Computers*, 69(3):377–391, 2020.

[23] Radeon VII 16GB HBM 2 memory cost around $320. https://www.fudzilla.com/news/graphics/48019-radeon-vii-16gb-hbm-2-memory-cost-around-320.

[24] Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. Intel nervana neural network processor-t (NNP-T) fused floating point many-term dot product. In *Proceedings of the 2020 IEEE 27th Symposium on Computer Arithmetic*, pages 133–136, 2020.

[25] Jae-Woo Im, Woo-Pyo Jeong, Doo-Hyun Kim, Sang-Wan Nam, Dong-Kyo Shim, Myung-Hoon Choi, Hyun-Jun Yoon, Dae-Han Kim, You-Se Kim, Hyun-Wook Park, Dong-Hun Kwak, Sang-Won Park, Seok-Min Yoon, Wook-Ghee Hahn, Jin-Ho Ryu, Sang-Won Shim, Kyung-Tae Kang, Sung-Ho Choi, Jeong-Don Ihm, Young-Sun Min, In-Mo Kim, Doo-Sub Lee, Ji-Ho Cho, Oh-Suk Kwon, Ji-Sang Lee, Moo-Sung Kim, Sang-Hyun Joo, Jae-Hoon Jang, Sang-Won Hwang, Dae-Seok Byeon, Hyang-Ja Yang, Ki-Tae Park, Kye-Hyun Kyung, and Jeong-Hyuk Choi. A 128Gb 3b/cell V-NAND flash memory with 1Gb/s I/O rate. In *Proceedings of the 2015 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 1–3, 2015.

[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] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, page 1–12. ACM, 2017.

[28] Myoungsoo Jung. OpenExpress: Fully hardware automated open research framework for future fast NVMe devices. In *Proceedings of the 2020 USENIX Annual Technical Conference*, pages 649–656. USENIX Association, July 2020.

[29] Behemoth FMS storage trace. https://github.com/SNU-ARC/storage-trace.

[30] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. USENIX Association, June 2014.

[31] Douwe Kiela, Suvrat Bhooshan, Hamed Firooz, and Davide Testuggine. Supervised multimodal bitransformers for classifying images and text. *arXiv e-prints*, September 2019.

[32] Shine Kim, Jonghyun Bae, Hakbeom Jang, Wenjing Jin, Jeonghun Gong, Seungyeon Lee, Tae Jun Ham, and Jae W. Lee. Practical erase suspension for modern low-latency SSDs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, page 813–820. USENIX Association, July 2019.

[33] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for multi-streamed SSDs using program contexts. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, pages 295–308. USENIX Association, February 2019.

[34] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient Transformer. *arXiv e-prints*, January 2020.

[35] Wojciech Kryściński, Nitish Shirish Keskar, Bryan McCann, Caiming Xiong, and Richard Socher. Neural Text Summarization: A Critical Evaluation. *arXiv e-prints*, page arXiv:1908.08960, August 2019.

[36] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 754–768. ACM, 2019.

[37] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 461–475, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, and Jongman Kim. Preemptible I/O scheduling of garbage collection for solid state drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(2):IEEE, 247–260, 2013.

[39] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. *arXiv e-prints*, June 2020.

[40] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv e-prints*, October 2019.

[41] Piji Li, Wai Lam, Lidong Bing, and Zihao Wang. Deep Recurrent Generative Decoder for Abstractive Text Summarization. *arXiv e-prints*, page arXiv:1708.00625, August 2017.

[42] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 395–410. USENIX Association, 2019.

[43] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, page 11. USENIX Association, 2012.

[44] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv e-prints*, July 2019.

[45] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. WARM: Improving NAND flash memory lifetime with write-hotness aware retention management. In *2015 31st Symposium on Mass Storage Systems and Technologies*, pages 1–14, 2015.

[46] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. DeepStore: In-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 224–238. ACM, 2019.

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

[48] NVM Express 1.4. https://nvmexpress.org.

[49] Young H. Oh, Seonghak Kim, Yunho Jin, Sam Son, Jonghyun Bae, Jongsung Lee, Yeonhong Park, Dong Uk Kim, Tae Jun Ham, and Jae W. Lee. Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[50] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling Neural Machine Translation. *arXiv e-prints*, page arXiv:1806.00187, June 2018.

[51] PCI Express 4. https://pcisig.com.

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

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

[54] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text Transformer. *arXiv e-prints*, October 2019.

[55] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. FStream: Managing flash streams in the file system. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 257–264. USENIX Association, 2018.

[56] 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.

[57] C. Sandhya, Apoorva B. Oak, Nihit Chattar, Udayan Ganguly, C. Olsen, S. M. Seutter, L. Date, R. Hung, Juzer Vasi, and Souvik Mahapatra. Study of P/E cycling endurance induced degradation in SANOS memories under NAND (FN/FN) operation. *IEEE Transactions on Electron Devices*, 57(7):1548–1558, July 2010.

[58] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 14–27. ACM, 2019.

[59] Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. CTRL: A conditional transformer language model for controllable generation. *arXiv e-prints*, September 2019.

[60] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv e-prints*, September 2019.

[61] Suhas Jayaram Subramanya, Harsha Vardhan Simhadri, Srajan Garg, Anil Kag, and Venkatesh Balasubramanian. BLAS-on-flash: An efficient alternative for large scale ML training and inference? In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, pages 469–484. USENIX Association, 2019.

[62] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations from Transformer. *arXiv e-prints*, page arXiv:1904.06690, April 2019.

[63] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 49–66. USENIX Association, 2018.

[64] NVIDIA tesla V100 architeture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[65] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2Tensor for Neural Machine Translation. *arXiv e-prints*, page arXiv:1803.07416, March 2018.

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv e-prints*, June 2017.

[67] 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.

[68] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized autoregressive pretraining for language understanding. *arXiv e-prints*, June 2019.

[69] Haoyu Zhang, Jianjun Xu, and Ji Wang. Pretraining-Based Natural Language Generation for Text Summarization. *arXiv e-prints*, page arXiv:1902.09243, February 2019.

[70] Jie Zhang and Myoungsoo Jung. ZnG: Architecting GPU multi-processors with new flash for scalable data analysis. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, pages 1064–1075, 2020.

[71] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, pages 121–136. USENIX Association, February 2020.

[72] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. PEGASUS: Pre-training with extracted gap-sentences for abstractive summarization. *arXiv e-prints*, December 2019.