Scalable Data Management on Hybrid Memory System for Deep Neural Network Applications

Wei Rang
Department of Computer Science
UNC at Charlotte
Charlotte, USA
wrang@uncc.edu

Donglin Yang
Department of Computer Science
UNC at Charlotte
Charlotte, USA
dyang33@uncc.edu

Zhimin Li
School of Computing and Information
University of Pittsburgh
Pittsburgh, USA
zhl157@pitt.edu

Dazhao Cheng
Department of Computer Science
UNC at Charlotte
Charlotte, USA
dazhao.cheng@uncc.edu

2021 IEEE International Conference on Big Data (Big Data) | 978-1-6654-3902-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/BigData52589.2021.9671309

***Abstract*—Memory resource is a critical bottleneck for large-scale Deep Neural Network (DNN) applications. Hybrid Memory System (HMS) provides a promising solution to increase memory capacity in an affordable way. However, to release the powerful performance of HMS, data migration plays an important role. A typical DNN application has a couple of execution layers ,and each requires distinct data objects. Deploying DNN on HMS imposes enormous challenges on data migration strategy and inspires us to pursue smart solutions. To tackle the data migration problem on HMS for DNN applications, we propose a runtime system for HMS that automatically optimizes scalable data migration and exploits domain knowledge on DNN to decide data migrations between the fast and slow memories in HMS. To achieve a better performance in data migrations for DNN training, we introduce a reference distance and location based data management strategy (ReDL) that treats short-lived and long-lived data objects with Idle and Dynamic migration methods, respectively. Using ReDL, DNN training on HMS with a smaller fast memory size can achieve similar performance to the fast memory-only system. The experimental results demonstrate that with configured the size of fast memory to be 20% of each workload's peak memory consumption, our work achieves a similar performance (at most 9.6% performance difference) to the fast memory-only system. It further achieves an average of 19% and 11% improvement in data locality against the state-of-the-art solutions.**

***Index Terms*—Hybrid Memory System, Scalable Data Management Strategy, DNN Applications, Idle Migration, Dynamic Migration**

I. Introduction

Hybrid Memory System (HMS) is an emerging memory architecture consisting of heterogeneous memory components, e.g., DRAM, non-volatile memory (NVM). Different memory components are equipped with various technical attributes; for example, DRAM has a faster I/O speed but a smaller capacity; NVM provides more memory space while it is not as fast as DRAM; HMS brings a promising solution to improve memory capacity, increase I/O bandwidth, and avoid some existing memory limitations. Given that HMS consists of multiple types of memory components with different attributes (e.g., capacity, bandwidth, speed and, latency), the memory management strategy for HMS should take advantage of these attributes discrepancies when conducting data placements and migrations.

Deep Neural Network (DNN) has been dramatically successful over the past decade across many academic and industrial domains, including recommendation systems [1], real-time strategic game control [2], and computer vision [3]. In order to speed up the training process of DNN, in-memory computing is introduced. However, the demands for higher model qualities come with more training data and larger model sizes, which result in larger memory footprints. For example, the state-of-the-art-models about language translation have hundreds of billions of parameters [4], which requires hundreds of GB of active memory to hold the training network. The recommendation system developed by Facebook [5] contains orders of magnitude more parameters than the traditional neural networks, which means tremendous memory space is demanded to guarantee the system's normal running.

To host the large DNN models and ensure a smooth training process, HMS profiles the memory access pattern of DNN training, bridges the performance gaps caused by different memory components and eliminates the adverse impact of data migrations. However, memory management is more complicated when we consider memory size and data scalability. In HMS, the fast memory size tends to be much smaller than the slow ones due to its high price. The average price of DRAM DDR4 (a common fast memory) increases by 2.3x from 2016 to 2020 [6], motivating researchers to find an efficient memory management strategy for HMS. Moreover, data migrations between the fast and slow memories can be detrimental to the application performance due to the current training process has to suspend until the requested data is

This work is supported by the National Science Foundation under Grant CCF-1908843 and CNS-2008265.

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.

moved into the fast memory. Ideally, the data that is frequently accessed by DNN should be placed in the fast memory to ensure wider bandwidth and lower latency. In contrast, other less-used data is stored in the slow memory, which guarantees a better training performance and decreases data migrations.

Our work focuses on scalable data objects management on HMS for DNN training. We leverage the repeatability and predictability of the DNN computing graph to optimize data placement and migration between the fast and slow memory components. To flexibly characterize the data access pattern of each DNN, we use tensor as the basic unit for data profiling and management. We also introduce a metric named liveness to describe how many layers a tensor can survive based on the profiling information on tensors. Accordingly, short-lived and long-lived definitions are adopted to represent data objects with longer and shorter liveness. To achieve a better performance in data migrations for DNN training, we propose a reference distance and location based data management strategy (ReDL) that controls short-lived and long-lived data objects with Idle and Dynamic migration methods, respectively. ReDL uses scalable data migration periods to realize the overlap between data migrations and training processes so that the requested data is proactively hosted in the fast memory as much as possible. It further increases the training performance by introducing Direct Slow Memory Access (DSMA), which breaks the barrier between computing units and slow memory by allowing computing units to directly access data objects in the slow memory. In general, conducting data placement and migration across different memory components with less performance loss is a critical optimization target for HMS, especially to achieve similar performance to the fast memory-only system.

In this paper, we propose a runtime system that automatically optimizes data migrations and exploits domain knowledge on deep learning to conduct data management on HMS. The key contributions of our work are as follows:

- We empirically analyze the performance of DNN with different memory hardware and indicate the performance differences from hosting data objects in different memory.
- We propose a reference distance and location based data management strategy (ReDL) that treats short-lived and long-lived data objects with idle and Dynamic migration methods, respectively. ReDL uses scalable data migration periods to realize the overlap between data migration and the training process to achieve better training performance and memory utilization.
- We design a runtime system running on HMS with ReDL that automatically bridges the performance gaps between the fast and slow memories, avoids unnecessary data migration, and allows computing units to directly access data objects in the slow memory.
- We evaluate ReDL by with TensorFlow. The experimental results demonstrate that with configured the size of fast memory to be 20% of peak memory consumption of each workload, ReDL achieves a similar performance (at most 9.6% difference) to the fast memory-only system. It also achieves average of 19% and 11% improvements in data locality against the state-of-the-art solutions.

The rest of this paper is organized as follows. Section II gives background and motivations. Section III describes the detailed system design and data migration strategy. Section IV presents the experimental results. Section V reviews related work. Section VI concludes the paper.

II. Background and Motivation

A. Graph-Based DNN Training

DNN is often training with a backward propagation algorithm and an optimizer such as stochastic gradient descent. A typical DNN model usually contains a stack of layers, which is comprised of a group of neurons. Each neuron in a layer computes a non-linear function of neurons' outputs in the preceding layer, using a set of weights [7]. Training DNN models often include many iterative steps, which involve a batch of training data objects fed into DNN. Training DNN with some popular deep learning frameworks such as TensorFlow [8] implements DNN as a computation graph composed of a set of nodes or vertex, which represent some computational kernel. Each kernel has its attributes, such as the number of inputs, number of outputs, computation time, and computational complexity. Data dependencies and control between kernels are denoted as directed edges in the computation graph. Edges representing data dependencies are assigned with a tensor that takes contiguous fixed memory space size. Moreover, the computation graph is static and repeated in the whole training process. These features make profiling the memory access pattern of data objects possible and then optimize data migration in terms of the static DNN computation graph.

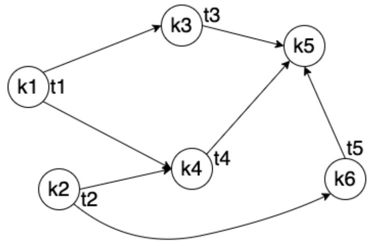


Fig. 1. A sample of DNN computation graph.

Figure 1 depicts a simple example of a computation graph with 6 kernels and 5 tensors. Nodes in the graph represent computing kernels and edges mean tensors consumed by kernels. Nodes are denoted with zero or more inputs and outputs. The inputs and outputs of each kernel are tensors. Each tensor is displayed with its producing kernel, each consumer of the tensor, and its last consumer. After the last consumer, a tensor's memory can be freed for other coming tensors. For example, tensor *t2* is produced by kernel *k2* and consumed by kernels *k4*, *k6*. Besides, *k6* is computed after *k4* in the computation graph. The memory space occupied by *t2* cannot be released

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.

until *k6* finished its computation, which means *k6* is actually the last consumer of *t2*.

We focus on the scenario that the computation graph describes a static DNN training process. That is, the computation graph has no data dependence control and the static graph structure is known at compile time. Moreover, the sizes of all intermediate data are obtained with the compile process. In this scenario, we can predict and record each tensor's memory access behavior and proactively migrate it into the corresponding memory component to achieve better performance.

B. Case Study

We characterize the memory access pattern in DNN and use the analysis results to drive our work. We adopt Persistent Memory Block Driver (PMBD) [9] to simulate the slow memory. PMBD is a PCIe NVM device simulation based on DRAM. 200ns delay for the read/write operation and 19GB/s bandwidth are set up to simulate slow memory media. We use the DRAM as the fast memory, which is configured with 34GB/s bandwidth and 90ns latency.

We use the ResNet50 V2 to study data objects (tensor) and their access patterns. Only one training step is used for profiling this information. ResNet50 V2 is fed with the CIFAR-10 data set with 128 batch sizes and 64 layers in a forward and backward pass. Besides, in this case study, we only use the regular DRAM which means there is no fast or slow memory included. We regard a data object as alive after it is allocated and before it is freed. The liveness of a data object is defined based on the number of layers the data object is alive.

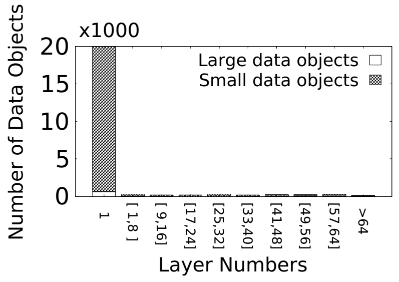


Fig. 2. Distribution of liveness of data objects and their sizes from ResNet50 V2.

1) Memory Access Pattern: Figure 2 displays the distribution of data objects' liveness and the ratio of data sizes. In this case, the small data object is smaller than 4KB and the large data object is no smaller than 4KB. The X-axis denotes how many layers (liveness) a data object can survive in the model training step; the last label ">64" means that the data object stays more than one forward and backward pass. Figure 2 shows that more than 90% of data objects' liveness are no longer than one layer and among those data objects, nearly 97% of them are small data objects. In this paper, we define the data object whose liveness is no longer than one layer as a short-lived data object and the data object can survive more than one layer as a long-lived data. By hosting short-lived data objects in the fast memory, we can decrease unnecessary data migrations, increase memory utilization and improve DNN training performance.

Figure 3 shows the distribution of memory access at the layer distance level. The figure shows that a large number of data objects are accessed in the first 24 layers. Among those data objects, nearly 77% of them are accessed in Layer 9-24. Those are the frequently used data objects that should be placed in the fast memory. On the other hand, some data objects are less accessed. For example, in the layer range [57, 64], there is almost no data access occurred. The uneven distribution of data objects and their unbalanced access pattern in DNN provide opportunities for data management.

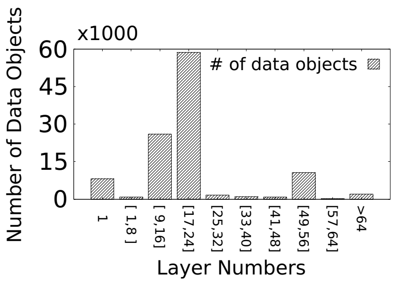
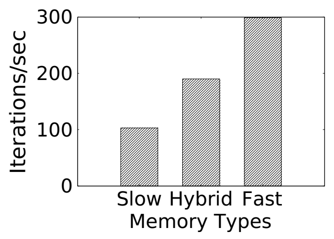


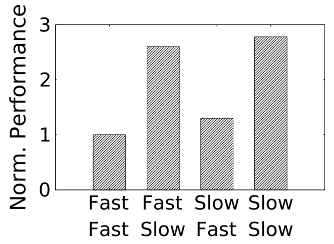
Fig. 3. Distribution of memory access at the layer distance level.

2) Impact of Different Memory: We conduct a further case study on the performance of ResNet50 V2 with 128 batch size under different memory settings. PMBD [9] simulates a slow memory while DRAM serves as the fast memory. Three memory settings are configured: all slow memory, hybrid memory and all fast memory. We set up 10GB in slow-only and fast-only memory cases, 2GB fast memory and 8GB slow memory in the hybrid case.

Figure 4(a) shows the training performance for three different memory settings: the slow-only memory, the fast-only memory and the hybrid memory running with NUMA [10]. The Slow bar displays that simply replacing fast memory with slow memory results in poor performance (about 3x slowdown) for training DNN models. The Hybrid bar shows



(a) Performance of iteration.



(b) I/O performance of memory.

Fig. 4. (a) shows the iteration performance of ResNet50 V2 with a batch size of 128. (b) plots normalized performance to all I/O committed in fast memory.

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.

that some specific optimized data management for hybrid memory is necessary, providing only a small improvement over the Slow case. The Fast case brings the best performance with almost 3x and 1.5x speedup compared with Slow and Hybrid cases, respectively, but the cost is more expensive than the first two settings. Thus, a hybrid memory system equipped with a smart data management strategy could reduce the performance gap between slow and fast memory.

The read/write speed of memory has implications on the performance of kernels with inputs and outputs in slow or fast memory hardware. We analyze the performance impact on a single CONV kernel from ResNet50 V2. Figure 4(b) demonstrates the execution time of this kernel with input (upper label) and output (lower label) saved into Slow and Fast memory hardware. We observe that when the input to the CONV kernel is saved in Slow memory and output is in Fast memory, its performance is very similar to the case when both input and output are in Fast memory. However, in the case when the output is saved in Slow memory, the kernel's performance decreased dramatically with over 2x slower. This behavior inspires us to consider the I/O latency differences when deciding where to place a data object to get an optimal runtime with a fast memory size constraint.

Summary: The above study has demonstrated the uneven distribution of short-lived and long-lived data objects and the unbalance access pattern across DNN training layers. Besides, the read/write speed asymmetry from the fast and slow memory results in large implications on kernel's performance, especially when a kernel places its outputs in the slow memory. Given that the DNN computation graph is static and its memory access pattern is stable, we can profile each tensor's data behavior to build a scalable memory management strategy for HMS, which is smart in controlling data placements and migrations. In the ideal case, short-lived data objects are to be placed in the fast memory while long-lived ones are in the slow memory. Specifically, the input data location should be considered because it has a huge effect on DNN training performance.

III. System Design

A. Architecture Overview

In this section, we present a runtime system that implements ReDL, a thin middleware layer between applications and backend deep learning frameworks.

An overview of the proposed architecture is shown in Figure 5. The key contribution relies on a novel reference distance and location based data management strategy (ReDL) on a hybrid memory system for DNN applications. ReDL can place data objects (tensors) into fast or slow memory and migrating data objects among them based on the liveness of data objects. More details on the data management strategy would be discussed below. As Figure 5 shows, our proposed architecture mainly consists of three components: *Kernel Controller*, *Data Objective Controller*, and *Memory Controller*.

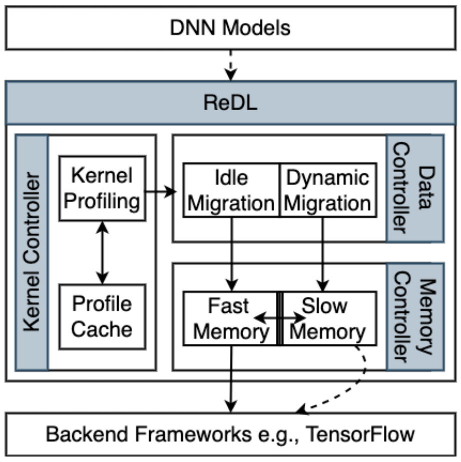


Fig. 5. Architecture Overview.

Kernel Controller has two functional modules. The Kernel Profiling collects memory access information of each kernel, records the related data objects (tensors), calculates the kernel's execution time in each layer, and decides the liveness of the kernel. The profiling process only uses one training step to collect the information and then update the profile information in Profile Cache. The Profile Cache is a software cache reserved to record profiled kernels. Since profiling of some DNN models may be time-consuming and DNNs typically contain many identical kernels, Profile Cache is necessary. Besides, by reserving a profile cache, the profiling process for any given DNN model only needs to be performed once.

Data Objective Controller is driven by the profiling information and implements the ReDL method we proposed. Short-lived and long-lived data objects are processed by *Idle Migration* and *Dynamic Migration*, respectively. Short-lived data objects are placed in contiguous memory space in fast memory by Idle Migration, and some of them will be freed when the space is in a tension status. This method decreases some unnecessary data movement caused by the short liveness of data objects. We adopt a Dynamic migration strategy for long-lived data objects, which conducts data migration between fast and slow memories periodically. In a migration period, the Dynamic Migration module migrates data objects demanded for the next period based on the static DNN computation graph. To handle the case that requested data objects are not timely migrated in the fast memory, we propose **Direct Slow Memory Access (DSMA)**, which breaks the barrier between computing units and slow memory by allowing computing units to directly access data objects in the slow memory.

Memory Controller consists of two modules. The *Fast Memory* manages data placed in fast memory while the *Slow Memory* maintains data objects in slow memory space. Besides, *Fast Memory* and *Slow Memory* communicate with each other to finish data migration between them. Data Objective Controller sends out all the data management information with a periodical heartbeat connection protocol. ReDL controls data

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.

migrations among fast and slow memories that overlap DNN application execution, such that the application performance is not affected.

B. Kernel Profiling

Kernel Profiling inspects the DNN computation graph in its compiling stage to extract the order and types of kernels in the graph, the producer(s) and consumer(s) of each kernel, and the data objects (tensors) it involves. It collects the profiling information of the kernel and tensor, such as its liveness, accessing time, and execution time by varying its inputs and outputs in fast and slow memories. Since the selection of input and output tensor location is tentative, it might take a couple of training steps to decide the best execution time for each kernel. We use the *Profiling Cache* to record the profiled kernels so that the profiling step for any given DNN graphs only needs to be performed once.

We need to minimize the execution time of the DNN computation graph with limited memory size. In the static DNN computation graph, computation kernels are executed sequentially. Therefore, there is not data object movement during the training step. We formulate the objective function as:

$$min \sum_{k\in\kappa, t\in\tau} \rho_{k,t}$$

$$\text{s.t.} \begin{cases} \sum_{t\in\tau} f_{(t,k)} + \sum_{t\in\tau} s_{(t,k)} \geq \tau & (1) \\ \sum_{t\in\tau} f_{(t,k)} \leq FS & (2) \\ \sum_{t\in\tau} s_{(t,k)} \leq SS & (3) \end{cases}$$

where *k* is a kernel in the set of all kernels of a computation graph and ρ_k is the expected execution of kernel *k*. τ is the set of all tensors for a kernel *k* and *t* is a tensor in τ. Note that ρ_k highly depends on the locations of input and output tensors for kernel *k*. The selection of the related input and output is important due to the dependencies among kernels. As shown in Figure 4(b), the optimal case is that the inputs are placed in the slow memory and outputs in the fast memory. The main constraints are on the amount of memory used by the computation graph. Constraint (1) denotes the total memory usage in the fast and slow memory is no smaller than the total size of tensors for a kernel. Constraints (2) and (3) represent the memory space used in the fast and slow memory, respectively.

To solve the objective function, we use the linear and integer programming theory [11], which uses the profiling information to automatically optimize the placed locations of data objects with memory size as the constraints.

C. Idle Migration

Idle Migration manages short-lived data objects in the fast memory aiming to guarantee a smooth training process by decreasing unnecessary suspense for data fetching. During the DNN model training steps, the short-lived data objects are not accessed too frequently compared with the long-lived data objects. However, Figure 2 shows a large number of short-lived data objects throughout the whole DNN training process, which indeed has a non-negligible impact on the training performance. Furthermore, the size of a short-lived data object is small, huge page in and out operations will be caused if these short-lived data are saved in the slow memory.

1) Managing Data Objects: A continuous space is allocated in the fast memory to host short-lived data objects. This space is reserved for short-lived and some newly migrated long-lived data objects. Short-lived data objects in the fast memory are not considered for migration when a memory tension occurs because their quantity is large. The migration of short-lived data in other memory can take a lot of time and is detrimental to the training process. However, long-lived data objects are supposed to be migrated to the slow memory if there is no enough space in the fast memory because these data objects have a higher probability of being accessed again throughout the training steps. The continuous fast memory space is allocated at the beginning of each migration period to accommodate data objects for the current training iteration. With this operation, ReDL guarantees that there is always enough space for short-live data objects. The access information of data objects is also collected during the migration period to achieve efficient memory utilization and better application performance.

When a new short-lived data object is required, ReDL first checks the free size of the fast memory. If there is enough size to hold the data objects, it is directly placed. If not, some short-lived data objects are selected as the victims to be freed, and then the migration of new data objects is processed. The selection of victims is based on the liveness. Short-lived data objects with shorter liveness are chosen. By doing this, the side-effects on application performance are lightened. At the end of the current migration period, ReDL updates the metric information such as liveness, accessing times for data objects and free space size, memory utilization, and data locality for the fast memory used in the next migration period.

2) Data Movement Implementation: Algorithm 1 depicts the main procedure of *Idle Migration*, mainly including *AllocFastMem*, *ParseProf*, *FastDOCtrl*, and *UpdateInfo* functions. (1) The fast memory is allocated at the beginning of each data migration period. Function *AllocFastMem* (Line 1-5) finishes smoothly applying for a continuous space from the fast memory if the available space satisfies the requested memory size and updates the size of free fast memory. (2) Then *ParseProf* function parses the profiles of data objects generated by profiling modules. In this step, the basic accessing information of each data object (tensor) is abstracted, such as liveness, execution time, input (producer), and output (consumer). According to the parsed information, the short-lived data objects are preferentially placed in the just allocated fast memory space. *ParseProf* also handles the migrations of the long-lived data objects from the slow memory. (3) When a new data object request arrives and is not in the fast memory, data migration is triggered. *FastDOCtrl* first checks whether the fast memory space is empty or there is enough space for the new data object. If the checking result is true, this data object is directly placed in fast memory.

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Idle Migration

```
Input: $profiling\_info$, $fast\_mem\_size$, $migration\_seq$
Output: $data\_info$, $mem\_info$
 1: function ALLOCFASTMEM($fast\_mem\_size$)
 2:     if $free\_size > fast\_mem\_size$
 3:         $mem\_pts$ = fastMalloc($fast\_mem\_size$)
 4:         $free\_size$ = $free\_size$ - $fast\_mem\_size$
 5:     return $mem\_pts$
 6: end function
 7: function PARSEPROF($profiling\_info$)
 8:     $data\_info$ = parseInfo($profiling\_info$)
 9:     return $data\_info$
10: end function
11: function FASTDOCTRL($mem\_pts$, $data\_info$)
12:     //New Data Paged in
13:     if $mem\_pts$ is empty or enough space
14:         $mem\_pts$ = placeData($data\_info$)
15:     else freeVictimData()
16:     return $mem\_pts$, $data\_info$
17: end function
18: function UPDATEINFO($profiling\_info$)
19:     $profiling\_info$ = updateInfo($mem\_pts$,$data\_info$)
20:     return $profiling\_info$
21: end function
```

Otherwise, *FastDOCtrl* traverses all the live data objects in the fast memory to find victims that are supposed to be freed. Victims are the data objects with much shorter liveness and on longer accessed in the current migration period. (4) The *UpdateInfo* function updates the metric information of the fast memory size and data objects. And this information helps make data management decisions in the next migration period.

The idle migration policy above tackles the issue of migrating short-lived data objects to the slow memory even if they are no longer accessed. Idle Migration decreases unnecessary data migrations, which result in performance loss and waste memory bandwidth. It is a huge waste of the fast memory space if short-lived data objects stay longer time in this memory. Furthermore, deciding the migrations of short-lived data objects is time-consuming but cannot always guarantee accuracy since collecting memory accessing information takes time, and counting the number of memory access information for data objects can be inaccurate. Idle Migration overcomes the limitations with the DNN domain knowledge and all the essential information for making migrations has been obtained in the profiling process.

D. Dynamic Migration

Dynamic Migration controls migrations of long-lived data objects in the slow memory. It uses the scalable migration period to decide the amount of long-lived data objects placed in the slow memory and the frequency of data movements between fast and slow memories. Besides, Dynamic Migration supports directly accessing the slow memory in the case that requested data is not timely migrated in the fast memory. The critical operation in Dynamic Migration is determining an optimal migration period size that brings the best DNN training performance. In ReDL, a training step is divided into equal-sized migration periods to guarantee flexible control on long-lived data objects migrations. We use the layers in the static DNN computation graph as the metrology to define the migration period. The layer-based migration period usually ensures the completion of kernel operation at the end of each period because no operations are running across layers. The static DNN graph structure is fixed at the compiling step, which provides the probability of finding the optimal layer-based migration period. Besides, each layer is associated with a computation phase that implies a memory access pattern. The layer-based migration period leverages the memory access pattern collected in the profiling phase to lead data migrations.

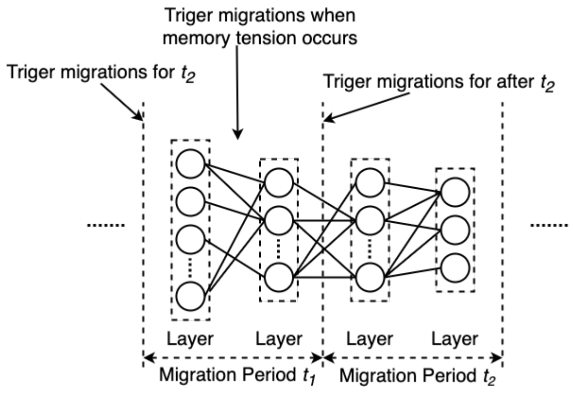

Triger migrations when memory tension occurs
Triger migrations for t_2
Triger migrations for after t_2
....
....
Layer
Layer
Layer
Layer
Migration Period t_1
Migration Period t_2

Fig. 6. Layer-based data migration period.

1) Determining Migration Periods: Figure 6 depicts a general example of a layer-based data migration period. In this example, two randomly consecutive migration periods, t_1, t_2, are displayed. Data objects migration for period t_2 is triggered at the beginning of the t_1, aiming to migrate most requested shored-lived data objects to the fast memory and long-lived ones in the slow memory before the second period starts. This operation occurs during the whole period so that data migration overlaps with DNN training and the overhead of data migration is further neutralized. Given the fact that the size of fast memory space is limited, memory tension is inevitable, which is shown as ”Trigger migrations when memory tension occurs” in Figure 6. When this case happens, some unused short-lived data objects are first freed, and then the long-lived ones that are not accessed in the current period are migrated back to the slow memory. Such a strategy is applied to save the fast memory space as much as possible. At the end of migration period t_1, another data object migration is triggered for the period after t_2. A similar operation is conducted until the training phase is finished.

Determining an optimal migration period is a dilemma. If the migration period is too large, the amount of data objects to migrate can be larger than the available memory space, especially for the fast memory. If the migration is small, then the possible execution time to overlap with DNN

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.

training is shorten. So the migration period cannot be too short; otherwise, the data objects to migrate cannot be timely migrated to the right memory space before the next migration period begins. A trade-off is to make between large and small migration periods.

To tackle this problem, we formulate the objective function as follows:

$$min \sum_{l \in \gamma} L_l(I)$$

where γ is the set of all layers l in the DNN computation graph, L_l is the expected execution time for layer l, and I is the optimal migration period for the DNN. We wish to *minimize* the execution time of the whole computation graph with limited memory space. Note that L_l depends on the memory size and data objects migration period.

For L_l, we can express it as:

$$L_l(I) = p_l(I) + m_l(I) \qquad (1)$$

where $p_l(I)$ is the computation time and $m_l(I)$ is data objects migration time. To find the optimal migration period *I*, we find that $p_l(I)$ and $m_l(I)$ have positive correlations with the migration period *I*. The large the migration period is, the longer time takes by $p_l(I)$ and $m_l(I)$. Equation (1) is subject to the following constraints:

$$\text{s.t.} \begin{cases} p_l(I) \leqslant S - F(I) & (4) \\ m_l(I) \geqslant p_l(I)/BW & (5) \end{cases}$$

where *S* is the total fast memory size, *F* is the fast memory space taken by the short-lived data objects, and *BW* is the migration speed determined by bandwidth between the fast and slow memories. *F* is a function of the migration period *I*. Different migration periods have different *F*s. According to the profiling results, *F* is relatively stable. *S* is a constant, so $S - F(I)$ is close to constant. $p_l(I)$ and $m_l(I)$ are also monotonically increasing functions of *I*. Hence, constraints (4) and (5) build the upper and lower bounds to the migration periods.

Although Equation (1) and its constraints reveal the inherent trade-off between large and small migration periods, it still needs a dedicated algorithm to find the optimal migration period. In ReDL, we adopt an iterated greedy algorithm [12] [13] to determine the optimal period at runtime. When the profiling step is finished, we start with the migration period of the median of the total layers and then test if this period satisfies the constraints. If the test result is positive, the optimal migration period is determined. Otherwise, we will first repeat this process with a new migration period by adding 1 layer to the current period size. And then test the case by deducting 1 layer from the median number of total layers. During this optimizing round, three training iterations are included. In the next round, the algorithm tests the migration periods by adding or deducting 2 to the median number. A similar round is repeated until the optimal migration period is found. We measure the performance from different migration periods and use the best one in the remaining training steps. We must ensure that data placement in any optimizing round is the same in order to obtain accurate comparison results. The same data placement is easily guaranteed due to the repetitive execution pattern in DNN model training. It might take a couple of rounds to determine the optimal migration period and result in some performance loss, but the total overhead is not large because this case does not often happen and performance loss is compensated in the remaining training steps.

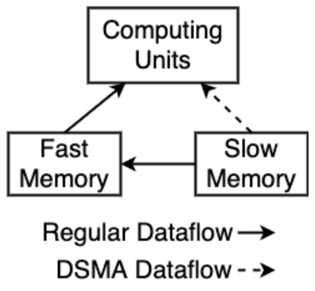


Fig. 7. Direct Slow Memory Access (DSMA).

2) Direct Slow Memory Access: Another case to consider is the optimal migration period fails to fit some of the computation phases because there is no enough time for migrating requested data objects from the slow memory to the fast memory before the upcoming period starts. To tackle this problem, we propose **Direct Slow Memory Access (DSMA)**, which breaks the barrier between computing units and slow memory by allowing computing units to directly access data objects in the slow memory. Figure 7 shows a simple dataflow of DSMA. In the regular dataflow, data objects in the fast memory are directly accessed by computing units such as CPU, GPU, and FPGA to execute DNN model training. Data objects saved in the slow memory are first migrated in the fast memory and then computing units can use them. With DSMA, computing units can directly access data objects in the slow memory without first migration in the fast memory. Application performance from DSMA cannot compete with the regular access mode, but DSMA does not occur frequently and its adverse impacts are negligible.

Algorithm 2 depicts the main modules of *Dynamic Migration*. The functionalities *AllocSlowMem*, *ParseProf* (Line 1-10) and *UpdateInfo* (Line 26-29) are similar to *Idle Migration*. *SlowDOCtrl* manages the long-lived data objects. The newly coming long-lived data object is placed in the slow memory (Line 16-20). The migration to the fast memory happens only when it is during a valid migration period and there is enough free space in the fast memory. We implement our proposed iterated greedy algorithm in *FindOptPerid* (Line 11-14), aiming to find the optimal migration period. As discussed in the above context, determining the optimal migration period might take a couple of rounds and cause some performance loss. We implement Direct Slow Memory Access (DSMA) in *accessSlowMem* function (Line 14) in *SlowDOCtrl* to handle the case that data migration to the fast memory cannot be finished due to the migration period does not fit to some layers. With DSMA, the training step still continues by accessing long-lived data objects in the slow memory and does not

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 Dynamic Migration

```
Input: $profiling\_info$, $fast\_mem\_size$, $migration\_seq$
Output: $data\_info$, $mem\_info$
 1: function AllocSlowMem($fast\_mem\_size$)
 2:     if $free\_size > fast\_mem\_size$
 3:         $mem\_pts$ = fastMalloc($fast\_mem\_size$)
 4:         $free\_size$ = $free\_size$ - $fast\_mem\_size$
 5:     return $mem\_pts$
 6: end function
 7: function ParseProf($profiling\_info$)
 8:     $data\_info$ = parseInfo($profiling\_info$)
 9:     return $data\_info$
10: end function
11: function FindOptPerid($data\_info$)
12:     $opt\_period$ = findPerid($data\_info$)
13:     return $opt\_period$
14: end function
15: function SlowDOCtrl($opt\_period$, $mem\_pts$, $data\_info$)
16:     if $mem\_pts$ is empty or enough space
17:         $mem\_pts$ = placeData($data\_info$)
18:     else freeVictimData()
19:     return $mem\_pts$, $data\_info$
20:     //Data Migrated to fast memory
21:     if $opt\_period$ is valid
22:         $mem\_pts$ = migrateData($data\_info$)
23:     else accessSlowMem($mem\_pts$)
24: end function
25: function UpdateInfo($profiling\_info$)
26:     $profiling\_info$ = updateInfo($mem\_pts$,$data\_info$)
27:     return $profiling\_info$
28: end function
```

need to wait for the completion of data migration. Overall, Algorithm 2 does not bring a large overhead because these special cases do not happen often. Hence a large number of training steps or testing rounds to determine the optimal migration period is not too needed.

In ReDL, since the long-lived data objects are placed in the slow memory and their total size is usually much larger than the short-lived ones, the size of this memory is allocated tens of GB to host data objects for the whole DNN model training.

IV. Evaluation

A. Experiment Setup

We study the performance of ReDL on a single machine, which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR4 RAM. Table I shows the testbed environment we use. Persistent Memory Block Driver (PMBD) [9] simulates a slow memory while DRAM serves as the fast memory so that a hybrid memory system is built. We compare the performance of ReDL with NUMA [10] and OS-Integrated Multi-level memory management system(OIM) [14]. In NUMA [10], when new memory space is allocated, it will occur in the fast memory if free space is available; otherwise, it will only occur in the slower memory node. OIM [14] improves page migration performance by launching four threads for paralleling page copy and eight threads for concurrent page migration, and it also optimizes page locations every five seconds. Unless specified otherwise, all the experiments are conducted on a pure CPU platform and the size of fast memory is configured to be 20% of the peak memory consumption of each DNN model. We will leave the CPU and GPU computation in future work.

TABLE I
Overview of Experimental Environment

CPU	Intel Xeon(R) CPU E5-2630v4
DRAM	64 GB DDR4
Fast Memory	Bandwidth: 34 GB/s Latency: 90ns
Slow Memory	Bandwidth: 19 GB/s Latency: 200ns

TABLE II
Summary of selective benchmarks.

Benchmark	Dataset	Batch size	Model size
ResNet50 V2	CIFAR-10	128	98 MB
LSTM	PTB	20	106 MB
VGG 19	CIFAR-10	64	549 MB
Inception V3	MNIST	64	92 MB

We select four popular DNN workloads to benchmark our work, ReDL is also working on other frameworks.. A summary of the benchmarks is described in Table II. TensorFlow [8] is used to implement ReDL and test its performance with these four workloads: ResNet50 V2 [15], LSTM [16], VGG 19 [14], and Inception V3 [17]. Each workload is run until its execution time in a migration period is stable. Since these workloads have no data dependent relations, performance will be constant after the first couple of computation iterations. For Tensorflow, we set its inter-op parallelism and intra-op parallelism to be 20, which is the number of physical cores in our testbed, so that all the experiments are conducted on the CPU with one thread per physical core. The performances of our work are mainly evaluated in (i) training throughput, (ii) training speedup, (iii) data locality, and (iv) overhead and scalability.

B. Training Throughput

Figure 8 depicts the performance of DNN model training throughputs. We compare ReDL with the following cases: FastMem (fast memory only system), OIM, NUMA and SlowMem (slow memory only system). The size of fast memory is configured to be 20% of each workload's peak memory consumption. The figures show that the performance difference between ReDL and all the fast memory cases is very small. The maximum difference is 9.6% from ResNet50 V2 while the minimum is 3.9% from VGG 19. Overall, ReDL has an average 6.9% performance improvement. ReDL averagely outperforms OIM with 8.4% (up to 11% in Inception V3) and is better than NUMA by 20% on average (up to 34% in ResNet50 V2). The improvements are bought with the

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.

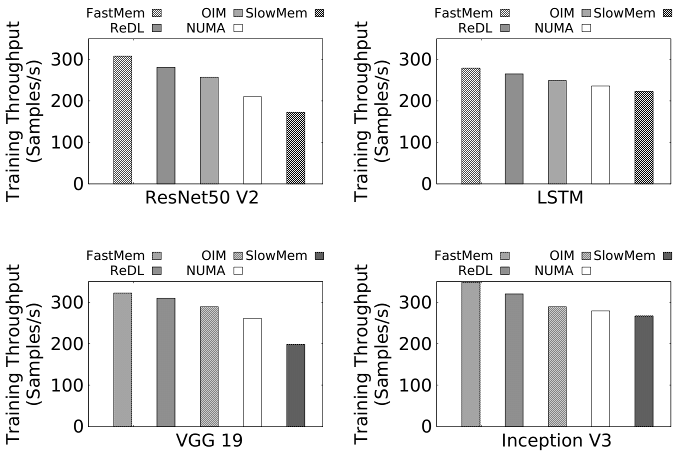


Fig. 8. Training throughputs from FastMem, ReDL, OIM, NUMA and SlowMem

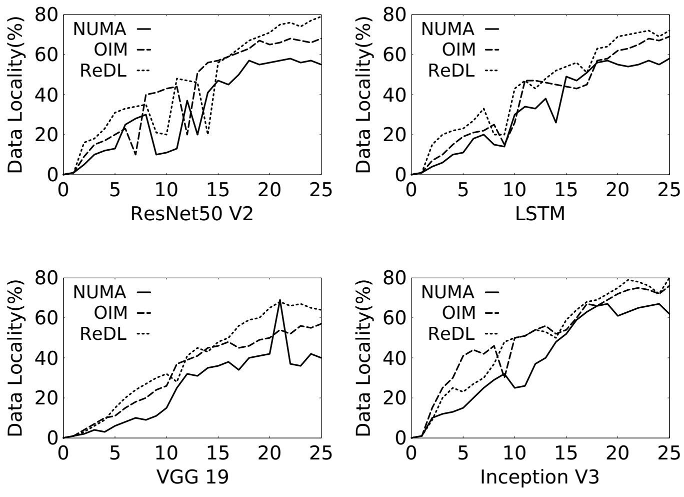


Fig. 10. Data localities from NUMA, OIM and ReDL.

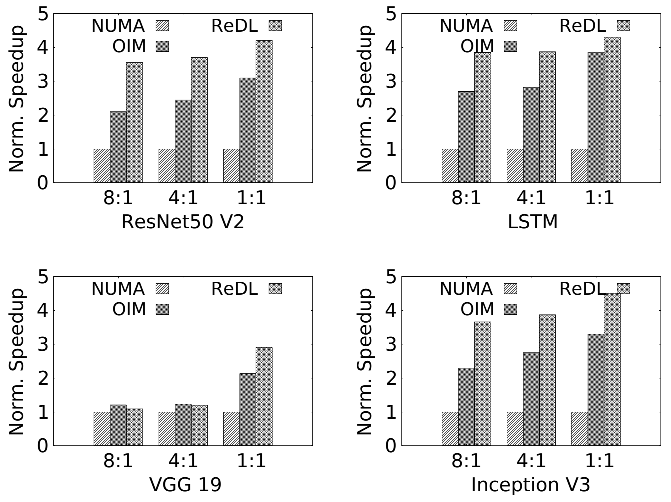


Fig. 9. Training speedup normalized to NUMA under different ratios between fast and slow memory.

fact that ReDL optimizes data migrations between fast and slow memory in tensors and conducts migrations only within migration periods. ReDL also has an average 20% increase compared with NUMA.

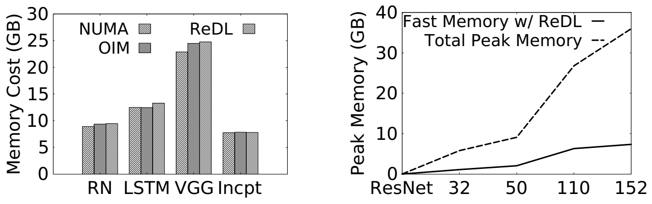
C. Training Speedup

Figure 9 shows the speedup achieved by ReDL normalized to training with NUMA. The x-axis denotes the ratio of slow to fast memory used to train the four workloads. In this experiment, we apply 40GB from the system memory and use PMBD [9] to simulate it as the fast and slow memories following different ratios above. We notice that NUMA's performance is poor compared with OIM and ReDL in all ratios: 8:1, 4:1, and 1:1. With more memory space is allocated as fast memory, performance improvements by NUMA are not so dramatic. This is because NUMA first allocates data objects in the fast memory without considering the computation sequence until the fast memory is full. Besides, in NUMA, the long-live data objects are placed in the fast memory resulting in more data migrations. OIM tackles this issue via whole page migration and parallel migration strategy, but its performance improvement is not as good as ReDL. ReDL has better performance under any memory ratios because it is aware of the liveness of data objects and based on them to optimize data migration between fast and slow memory. In Figure 9, VGG 19 is an outlier due to its extremely large second convolution layer, which demands more memory space to host its related data objects. When the fast memory is small (i.e., in the cases with ratios 8:1 and 4:1), some of the data objects must be placed in the slow memory, incurring a performance loss. When there is enough fast memory, we note a huge performance improvement, as it is illustrated in the figure, the performance jumps high from the 4:1 ratio to the 1:1 ratio.

D. Data Locality

Figure 10 illustrates the data localities from NUMA, OIM and ReDL. In this experiment, we define data locality as the percentage of requested data objects in the fast memory. Higher data locality implies fewer data migrations and, in turn, reflects the efficiency of the data management strategy. In Figure 10, we monitor the workloads' activities until their performance is stable, which is reflected by a relatively smooth line. We can observe that the stable time for each workload differs due to the differences in their computation graphs. Furthermore, a couple of peaks and valleys in each line, which displays locality fluctuations. The valleys between any two consecutive peaks illustrate data migration. Note that our proposed work, ReDL is always above NUMA and OIM with an average of 19% and 11% improvements, respectively. ReDL also delays the occurrence times and frequency of valleys because it can timely migrate data objects in the fast memory. There is an outlier in the experimental results of VGG with NUMA; a spike occurs around the time points 20. We can

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.



(a) Memory Overhead (b) Scalability.

Fig. 11. Memory Overhead and Scalability of ReDL.

explain this exception with its special large convolution layer, which demands more memory while the memory management policy in NUMA is not so efficient.

E. Overhead and Scalability

To analyze the overhead of ReDL, we use NUMA as the baseline and conduct experiments to measure peak memory consumption in NUMA, OIM and ReDL, which scales how much memory resource it takes to conduct DNN model training. Figure 11(a) shows ReDL consumes 6% and 3% more memory compared with NUMA, OIM due to ReDL needs some extra memory to conduct profiling and optimizing steps. Inception V3's peak memory consumption is relatively stable in all platforms. Because the model size of Inception is small and its computation graph is simple so that the memory demand is not large. The overall average memory overhead introduced is about 5% (about 3GB), which is negligible to the whole system-level memory (64GB in our testbed).

Figure 11(b) shows the peak fast memory consumption and the fast memory size for different ResNets. We use four different ResNets with different computation layers to illustrate ReDL's scalability. Figure 11(b) shows that with more layers added to ResNet, its peak memory consumption increases from 5.8G (ResNet 32) to 36GB (ResNet 152) and the fast memory size increases from 1.1GB to 7.35GB. In total, when increasing ResNet's layer from 32 to 152, the fast memory size rises slower than the peak memory. This behavior demonstrates the scalability in saving fast memory size and memory management effectiveness by using ReDL.

V. Related Works

Many new memory technologies have been proposed to build Hybrid Memory System (HMS), such as high-bandwidth memory [18], non-volatile memory [19]. Facebook adopts SSDs to reduce the memory footprint of databases [5]. Similarly, Bandana proposes an SSD based persistence memory system to hold DL training models with system memory serving as a small cache [20]. Recent works [21] [6] study data management on HMS with hardware-based methods. In [21], the authors propose a novel Hardware/Software Co-operative Caching (HSCC) mechanism that organizes NVM and DRAM in a flat address space while logically supporting a cache/memory hierarchy, which simplifies the hardware design and offers several optimization opportunities for cache management in software layers. [6] proposes an FPGA-based hybrid memory system emulation platform by leveraging the onboard hard IP ARM processors to improve simulation performance while enhancing the results' accuracy. The main goal of these works is to guarantee high memory performance with fast memory. ReDL is orthogonal to these systems and complemental to them with supporting HMS and achieving a better memory service rate.

Recently, many studies explore page placement policies in hybrid memory systems. Thermostat [22] proposes an application-transparent huge-page-aware mechanism to place pages in a dual-technology hybrid memory system. It uses an online page classification mechanism to classify both 4KB and 2MB pages as hot or cold while incurring no observable performance overhead across several representative cloud applications. FileMR [23] modifies the RDMA driver of the CPU to expose files in its local storage to external RDMA applications. Their main idea is that a remote user can access one of the local files without interrupting the local processor by using a special RDMA NIC with file access capabilities. However, FileMR needs to modify the RDMA standard by adding new functions, which results in less flexibility in implementation. OIM [14] guides page placement based on an existing Linux page replacement mechanism. It improves page migration performance by launching multi-threaded migration for single pages and concurrent migration for multiple pages. However, using this design to decide page migration for common short-lived data objects in DNN can be slow and lacks a global view. Autotm [24] profiles the execution time of kernels by placing input and output tensors in different combinations of fast (DRAM) and slow (NVRAM) memory media and manages data placement and movement to minimize execution time with the limited fast (DRAM) size. ReDL complements these works and mainly focuses on data object migrations which use tensor as the basic unit.

VI. Conclusion

In this paper, we propose a runtime system that automatically optimizes scalable data migration and exploits domain knowledge on DNN to decide data migrations between the fast and slow memories in HMS. To achieve a better performance in data migrations for DNN training, we introduce a reference distance and location based data management strategy (ReDL) that treats short-lived and long-lived data objects with Idle and Dynamic migration methods, respectively. Using ReDL, DNN training on HMS with a smaller fast memory size can achieve similar performance to the fast memory-only system. Our experimental results demonstrate that with configured the size of fast memory to be 20% of each workload's peak memory consumption, ReDL achieves a similar performance (at most 9.6% performance difference) to the fast memory-only system. It further achieves an average of 19% and 11% improvement in data locality against the state-of-the-art solutions.

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.

References

[1] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016, pp. 7–10.

[2] G. A. DeepMind, "Mastering the real-time strategy game starcraft ii," 2019.

[3] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[4] A. Vaswani, Y. Zhao, V. Fossum, and D. Chiang, "Decoding with large-scale neural language models improves translation," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 2013, pp. 1387–1392.

[5] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing dram footprint with nvm in facebook," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–13.

[6] F. Wen, M. Qin, P. V. Gratz, and A. Reddy, "Fpga-based hyrbid memory emulation system," *arXiv preprint arXiv:2011.04567*, 2020.

[7] J. Ren, J. Luo, K. Wu, M. Zhang, and D. Li, "Sentinel: Runtime data management on heterogeneous main memorysystems for deep learning," *arXiv preprint arXiv:1909.05182*, 2019.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[9] F. Chen, M. P. Mesnier, and S. Hahn, "A protected block device for persistent memory," in *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2014, pp. 1–12.

[10] C. Lameter, "Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors." *Queue*, vol. 11, no. 7, pp. 40–51, 2013.

[11] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[12] D. Lei, Y. Yuan, J. Cai, and D. Bai, "An imperialist competitive algorithm with memory for distributed unrelated parallel machines scheduling," *International Journal of Production Research*, vol. 58, no. 2, pp. 597–614, 2020.

[13] W. Rang, D. Yang, D. Cheng, and Y. Wang, "Data life aware model updating strategy for stream-based online deep learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2571–2581, 2021.

[14] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 331–345.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[16] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[17] X. Xia, C. Xu, and B. Nan, "Inception-v3 for flower classification," in *2017 2nd International Conference on Image, Vision and Computing (ICIVC)*. IEEE, 2017, pp. 783–787.

[18] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 432–433.

[19] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, S. Seo *et al.*, "A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta 2 o 5- x/tao 2- x bilayer structures," *Nature materials*, vol. 10, no. 8, pp. 625–630, 2011.

[20] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 160–167.

[21] H. Liu, Y. Chen, X. Liao, H. Jin, B. He, L. Zheng, and R. Guo, "Hardware/software cooperative caching for hybrid dram/nvm memory architectures," in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1–10.

[22] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 631–644.

[23] J. Yang, J. Izraelevitz, and S. Swanson, "Filemr: Rethinking {RDMA} networking for scalable persistent memory," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 111–125.

[24] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 875–890.

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:16:28 UTC from IEEE Xplore. Restrictions apply.