IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Enabling Large Dynamic Neural Network Training with Learning-based Memory Management

Jie Ren*, Dong Xu§, Shuangyan Yang§, Jiacheng Zhao‡, Zhicheng Li‡,
Christian Navasca†, Chenxi Wang‡, Harry Xu†, and Dong Li§
* William & Mary, ‡ University of Chinese Academy of Sciences,
† University of California, Los Angeles, § University of California, Merced
{jren03}@wm.edu, {zhaojiacheng, lizhicheng21s, wangchenxi}@ict.ac.cn,
{cnavasca253}@g.ucla.edu, {harryxu}@cs.ucla.edu, {dxu17, syang127, dli35}@ucmerced.edu

***Abstract*—Dynamic neural network (DyNN) enables high computational efficiency and strong representation capability. However, training DyNN can face a memory capacity problem because of increasing model size or limited GPU memory capacity. Managing tensors to save GPU memory is challenging, because of the dynamic structure of DyNN. We present DyNN-Offload, a memory management system to train DyNN. DyNN-Offload uses a learned approach (using a neural network called the *pilot model*) to increase predictability of tensor accesses to facilitate memory management. The key of DyNN-Offload is to enable fast inference of the pilot model in order to reduce its performance overhead, while providing high inference (or prediction) accuracy. DyNN-Offload reduces input feature space and model complexity of the pilot model based on a new representation of DyNN; DyNN-Offload converts the hard problem of making prediction for individual operators into a simpler problem of making prediction for a group of operators in DyNN. DyNN-Offload enables 8× larger DyNN training on a single GPU compared with using PyTorch alone (unprecedented with any existing solution). Evaluating with AlphaFold (a production-level, large-scale DyNN), we show that DyNN-Offload outperforms unified virtual memory (UVM) and dynamic tensor rematerialization (DTR), the most advanced solutions to save GPU memory for DyNN, by 3× and 2.1× respectively in terms of maximum batch size.**

I. Introduction

Deep learning (DL) is embracing dynamic neural network (NN) architectures where the NN structure changes across data samples [23]. Such dynamic neural networks (DyNN) are different from the traditional static NN where a network architecture (i.e., a dataflow graph) is defined using symbolic expressions before execution and fixed during execution. A DyNN model may select its model components (e.g., layers [28], channel [34] or sub-networks [66]) conditional on input samples, and change the structure and parameters in the dataflow graph accordingly. DyNN decouples the requirement for many parameters from computational costs, which leads to reduction of training cost. Previous works [10], [19], [60], [67], [73], [83] show that compared with static NN, DyNN reduces training cost yet improve model prediction performance. DyNNs have shown high computational efficiency over variable-length sequences [71], trees [72], and graphs [33]. They have also demonstrated strong representation capabilities and high adaptiveness in achieving desired tradeoffs between accuracy and efficiency on the fly [23]. As a result, DyNNs have been applied to many problems, such as speech recognition [78], language modeling [19], [60], [61], [82], image recognition [5], [17] and DL translation [6], [71], [73]. Recently, DyNNs are applied to large language models (such as GLaM from Google [18]), pushing the limit of scaling laws in the age of generative models. It is believed that the DyNN is one of a few techniques to improve efficiency and resource utilization of future large models [39].

Problems. DyNNs, as many other NNs, are often memory hungry [49], [57], [58], [86]. This is especially the case as large models are gaining increasing popularity. For example, AlphaFold [64], a DyNN model based on evoformers (a variant of transformer) recently making breakthrough in protein structure prediction, consumes 1,024 GB memory when using 128 amino acids sequences of 256 in length [1]. As another example, a switch-based mixture-of-expert (MoE) model with the similar parameter efficiency as T5-large (a static natural language processing model) consumes at least 320 GB memory [54]. Clearly, training of large models is fundamentally limited by GPU memory capacity. Distributed parallel training techniques such as pipeline parallelism [40], [41] and tensor model parallelism [65] go beyond the memory boundary of single GPU by splitting model states across multiple GPUs, enabling training of massive models that would otherwise not fit into a single GPU's memory. However, these techniques require enough GPUs to provide large aggregated GPU memory to store the model states necessary for training; these GPUs can be extremely expensive and beyond the affordability of many small companies and organizations [56], [58].

Exploiting CPU memory to reduce the need of GPU memory for large model training has been explored [26], [27], [49], [50], [57]–[59]. Although tensor offloading to CPU memory is effective in training static models, it is hard to be applied to DyNNs. In particular, effectively using heterogeneous memory (CPU and GPU memories) requires minimizing the amount of communication between CPU and GPU or hiding communication. To achieve this goal, existing efforts rely on profiling-guided optimization (PGO) to record tensor access orders using a few training iterations and plan tensor prefetech between CPU and GPU for remaining iterations. PGO has a fundamental assumption: the NN model must be invariant, i.e., using a static computation graph where tensor dimensions as well as data and control flows are statically fixed, and there are no complex data structures (such as graphs and trees) in the dataflow graph. Hence, profiling a few training iterations is enough to decide tensor prefetch for upcoming operators.

However, the above assumption does not hold for DyNNs due to their inherent dynamism. Depending on the input, the DyNN selectively activates model components, introducing irregular memory accesses and invalidating profiling results collected in training iterations. As a result, communications between CPU and GPU are largely exposed to the critical path, leading to training throughput loss.

This paper presents a memory (tensor) management system, *DyNN-Offload*, for training large DyNNs. DyNN-Offload uses a new approach to guide tensor migration between CPU and GPU to maximize GPU memory efficiency. In particular, we explore the extent to which a pilot model, such as an NN, can be used to increase predictability of tensor accesses during the training process of a large DyNN. We use the pilot model to timely prefetch tensors from CPU memory to GPU memory to hide communication overheads.

Insights. Key to our approach is *learning for learning*, i.e., using the learned knowledge proactively gained from other input problems and DyNNs, instead of using PGO which lacks flexibility to handle dynamism in DyNN training. Our work is driven by two major insights.

- The training process of an NN (regardless of whether it is static or dynamic) exhibits learnable patterns. For example, linear computation is commonly followed by nonlinear computation (e.g., ReLU activation function).
- In essence, the dynamic structure of a DyNN builds on a series of decision-making processes (i.e., control flows) to activate model components, which, in turn, impacts the access orders of tensors. The input sample to the DyNN provides indications on how such processes occur. A prediction model can be learned to enable automatic synthesis of the decision-making processes.

Research challenges. Developing a model for GPU memory management requires overcoming a number of challenges. The first is how to minimize the performance impact of querying the model (referred to as *pilot model*) for memory management. The inference using the pilot model introduces performance overheads to the critical path of DyNN training.

The second challenge is how exactly to use the pilot model. DyNN-Offload queries the pilot model to decide *when to prefetch tensors* from CPU to GPU memory with the goal to maximize the overlapping between tensor migration and DyNN training. Tensor prefetching is critical in minimizing the overheads incurred from tensor migration. A possible idea is to build the pilot model to predict the exact execution order of operators. If this can be done, we could come up with a prefetch plan in a similar way to using PGO-guided tensor prefetch for static NNs. However, this approach requires rich output from the pilot model and high prediction accuracy, which leads to high inference overhead of the pilot model. Hence, there is an important tradeoff between the usefulness (to guide tensor prefetch) and performance overhead.

DyNN-Offload. The design of the pilot model centers around how to enable efficient enforcement and yet provide high accuracy. We achieve this goal based on two observations: (1) operators in machine learning (ML), though rich in interfaces and algorithms, can be identified by a combination of *six pervasive and expressive memory access patterns*. (2) Tensors typically migrate in batches in order to fully utilize interconnect bandwidth. For those tensors that migrate together, there is no need to predict the exact execution order of the operators that reference the migrating tensors. This observation relaxes the requirement of using fine-grained execution order to plan tensor prefetch, which is the central technique used in all PGO-based solutions for static NNs [50], [57], [59], [75].

Based on the first observation, the input features and output of the pilot model can benefit from a compact representation based on six program idioms to *encode* the DyNN's architecture and indicate execution order of operators. This compact representation reduces the input feature space, leading to a simpler pilot model. Based on the second observation, the pilot model implicitly partitions a DyNN with resolved dynamism into multiple execution blocks, and only predicts the execution order of these blocks. This leads to an easier prediction task, and hence a lighter pilot-model and higher prediction accuracy. The above techniques address the challenge on the performance overhead of the pilot model.

To address the challenge in the planning of tensor prefetching, DyNN-Offload learns how to hide tensor migration through the training of the pilot model. During the pilot model training, the DyNN is transformed to a static one and then an existing PGO solution is used to decide execution blocks. Such transformation allows DyNN-Offload to create training samples with the knowledge of optimal DyNN partitioning for the pilot model to learn.

Results. DyNN-Offload supports a variety of DyNNs and works on real production datasets without the need of refactoring DyNNs. DyNN-Offload significantly improves GPU memory efficiency: given a constraint on GPU memory consumption, DyNN-Offload enables 8$\times$ larger DyNN training on a single GPU compared with using PyTorch alone (unprecedented with any existing solution); Evaluating with AlphaFold (a production-level, large-scale DyNN), we show that DyNN-Offload outperforms unified virtual memory (UVM) [44] and dynamic tensor rematerialization (DTR) [30], the most advanced solutions for DyNN, by 3$\times$ and 2.1$\times$ respectively in terms of maximum batch size. DyNN-Offload also reduces training time of the DyNN by 35% (up to 1.38$\times$) compared to UVM and DTR, while other solutions (e.g., ZeRO-Infinity [56]) cannot work for DyNNs. The pilot model causes only 30 μs inference overhead, much shorter than the training iteration time of large DyNNs ($O(100\mu s)$) for each training sample. Given $O(1000)$ training samples, the pilot model only mis-predicts tens of them.

II. Background

A. Dynamic Neural Networks

Static NN applies fixed-structured operations to all input samples. For example, convolutional NNs apply fixed network architecture to fixed-sized images, and are able to capture the spatial invariance common in computer vision. However, besides images, many forms of data (e.g., sequences of variable lengths and graphs) are structurally complex, and cannot be captured by fixed-structured NNs. DyNNs can adapt their structures or parameters to the input sample. Such dynamism is able to reflect the complex structures of input data, hence leading to high execution efficiency and accuracy. Such dynamism is often controlled by confidence-based criteria and gating functions using control flows.

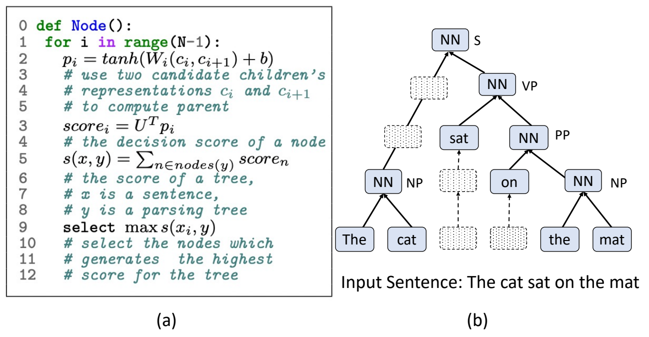
Figure 1 presents a DyNN example. This is a Tree-CNN based on constituency parsing and designed for the sentence embedding task in natural language processing. The Tree-CNN generates representations for the input sentence by constructing a parsing tree in a bottom-up manner and combining the representations of each subtree. In the Tree-CNN, each node represents a grammar type and has its own dedicated CNN. Figure 1.a illustrates the tree-building rule: each pair of neighboring nodes attempts to create a parent node, and only the parent node with the highest score is activated to form a subtree. Specifically, Line 9 in Figure 1.a demonstrates a control flow that determines the dynamic structure of the Tree-CNN. Therefore, the architecture of Tree-CNN varies based on the content of the input sentence. Figure 1.b shows the *resolved* tree structure for an input sentence. In the figure, the activated nodes are represented in blue, and the inactivated nodes are shown in shadow. Besides Tree-CNN, there are other types of DyNNs. Their commonality lies in the dynamism (i.e., the existence of the control flow as Line 9 in Figure 1.a). Their differences lie in the model components activated by the control flow: in Tree-CNN, the model components are CNN in tree nodes; in MoE, the model components can be multi-layer perceptron within a transformer block. DyNN-Offload can be generally applied to those DyNNs to address their dynamism.

We use the following terms throughout the paper.

- **DyNN training**, which is the target workload in this paper. DyNN training uses the pilot model for tensor prefetching.
- **Pilot model inference (or prediction)**, which means the pilot model is used to decide tensor prefetching;
- **Pilot model training**, which decides the pilot model parameters and happens offline.

B. Breaking Memory Capacity Wall

As state-of-the-art DL models continue to grow, training them within the capacity of GPU memory becomes increasingly challenging. Such a memory capacity wall limits ability to explore training techniques and memory-intensive model architectures. There are several solutions to reduce memory consumption and address this problem, such as using low-precision tensors [84], distributed training [41], [42], [56], [69], tensor redundancy removal [55], tensor migration on heterogeneous memory [26], [27], [49], [50], [57]–[59], and tensor rematerialization [11], [20], [29], [30], [70]. Among them, tensor migration and rematerialization are attractive, because they do not have risks of losing training convergence, do not change NN models, and are cost-effective (i.e., no need of extra GPU). However, they cannot work well for DyNNs.



```
0  def Node():
1  for i in range(N-1):
2    $p_i = tanh(W_i(c_i, c_{i+1}) + b)$
3    # use two candidate children's
4    # representations $c_i$ and $c_{i+1}$
5    # to compute parent
3    $score_i = U^T p_i$
4    # the decision score of a node
5    $s(x, y) = \sum_{n \in nodes(y)} score_n$
6    # the score of a tree,
7    # x is a sentence,
8    # y is a parsing tree
9    select $\max s(x_i, y)$
10   # select the nodes which
11   # generates  the highest
12   # score for the tree
```

Fig. 1. A DyNN example: (a) the implementation of each tree node in the DyNN, (b) the tree network where S, PP, NP and VP stand for sentence, preposition phrase, noun phrase, and verb phrase.

TABLE I
DISTRIBUTION OF JACCARD DISTANCE FOR ALL TRAINING SAMPLES.

	[0,0.2]	(0.2,0.4]	(0.4,0.6]	(0.6, 0.8]	(0.8, 1]
Percentage of training samples	5%	28%	25%	40%	2%

Tensor migration highly relies on the workload predictability to decide when tensor migration (prefetch) should happen. For static NN, such predictability is provided by PGO based on the assumption that the workload characteristics (including execution time and tensor accesses) is invariant across training samples, which is not held for DyNN.

To quantify the unpredictability of DyNN, we use Tree-LSTM [72] with 6,000 training samples as an example. We use Tree-LSTM as an example because of its rich control flows, but other DyNNs in our analysis (shown in Table II) show similar unpredictability. For each training sample, we build a binary vector and each element of it indicates if a specific control flow is taken or not. We use the first training sample as the baseline, and use the Jaccard distance [76] (JD) to quantify the execution similarity between the baseline and any other training sample. The JD is a common metric to measure how dissimilar two vectors are. For a given training sample, the analysis result (the JD) is a value falling into $[0, 1]$, with "1" indicating the training sample and the baseline take completely different control flows and "0" indicating opposite. A larger JD value indicates lower similarity. Table I shows the results, which shows a wide divergence of execution of the dataflow graph across training samples. This divergence fails the traditional PGO-guided tensor prefetch (in particular, profiling tensor accesses using the first few training samples cannot be used to guide prefetch for other training samples).

Tensor rematerialization frees some tensors (particularly activations) from GPU memory but recomputes them on demand. Tensor rematerialization uses checkpointing to store some tensors in GPU, in order to replay the parent operations to reproduce the freed tensors. Tensor rematerilization can work for DyNNs [21], [30].

However, tensor rematerialization has fundamental limitation. (1) Rematerialization can be recursive: if the arguments to a freed tensor's parent operation are freed too, then those arguments must first be rematerialized. There is no theoretical bound on depth of the recursiveness, leading to potentially large loss in training throughput. (2) Some tensors (e.g., constant tensors and weights) cannot be rematerialized, leading to a tighter bound on memory saving (compared with using tensor prefetch techniques). Our evaluation shows the inferior performance of using tensor rematerialization, compared with using the pilot-model guided tensor migration (Section VI-C).

C. *Using Machine Learning to Guide Tensor Migration*

Before we explored ML to guide tensor migration for DyNN, we asked whether simple heuristics would be accurate enough to resolve dynamism. For example, for a DyNN used in NLP and taking a sentence as input, one might measure the ratio of the number of verbs to the number of nouns in the input sentence, and assume that a larger ratio implies a higher possibility of taking some branch in the dataflow graph. However, we did not find a high correlation between the ratio and the decision of taking the branch, using Spearman's correlation or Pearson's correlation: at most 0.20 for Spearman's and 0.25 for Pearson's, which are known as the low degree of correlation [8]. Also, using this heuristics for prefetch, we find that var-BERT easily has 50% performance loss, compared to DyNN-Offload. Also, the heuristic is difficult to be generalized: different DyNNs require different heuristics. Hence, we decided to try ML models. Recent research on distributed and operating systems successfully employed ML for scheduling and resource allocation [24], [36], [53], [68], [79], [85]. A similar exploration to address the memory capacity problem can lead to a powerful result, as we show in this paper.

III. Overview

Usage scenario. DyNN-Offload is used to guide the decision for tensor prefetch based on operators execution order to save GPU memory without losing DyNN training throughput and accuracy. DyNN-Offload is transparent to data scientists and does not require DyNN refactoring. Before an input sample is fed into the DyNN for training, an NN model (i.e., the pilot model) is used to quickly resolve control flows in the DyNN and indicate when the tensor prefetch should happen. Based on the prediction of the pilot model, the runtime system in DyNN-Offload triggers tensor prefetch.

Overall architecture. Figure 2 shows the overall architecture of DyNN-Offload, consisting of three main components.

(1) The pilot model. The center of DyNN-Offload is a light NN (the pilot model). The pilot model's input features are the input sample to the DyNN and the DyNN architecture information collected through static analysis on the DyNN model script. The pilot model's output indicates how operators in the DyNN will be executed at the granularity of execution blocks. An execution block includes a group of operators. The pilot model indicates how the computation graph of the DyNN is partitioned into execution blocks, such that at the

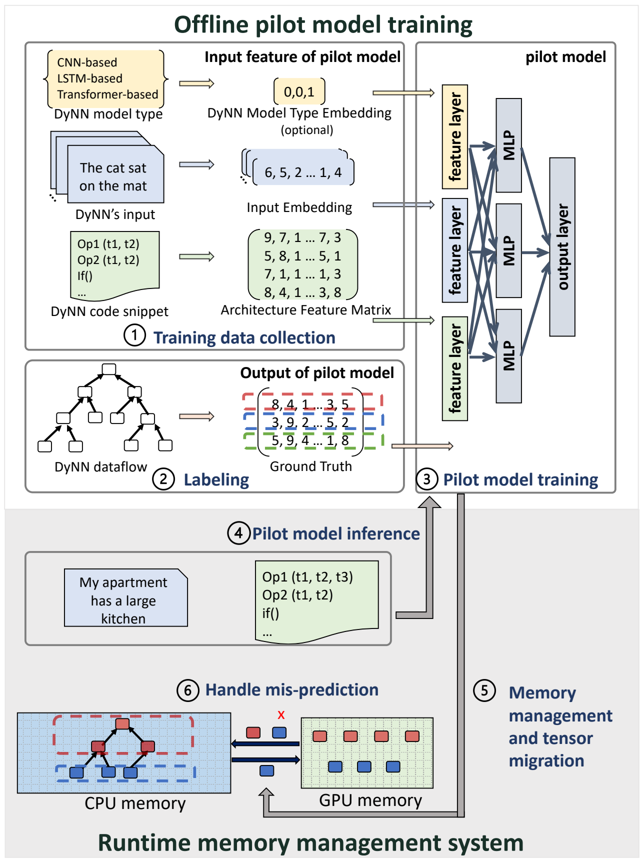


Fig. 2. The workflow of DyNN-Offload.

beginning of an execution block, the tensor migration for the next execution block is triggered to hide migration cost.

The input features and output of the pilot model use a program idiom-based representation to identify operators in the DyNN. We use six idioms defined in terms of memory access patterns, and each operator can be easily characterized with a combination of the six idioms. Using the idiom-based representation significantly reduces the complexity of input features and output, leading to the lightness of the pilot model.

(2) Runtime system. DyNN-Offload manages GPU memory for tensor migration and DyNN training based on double buffering. Based on the pilot model, the tensors can be prefetched from CPU memory to GPU memory at the beginning of each execution block. DyNN-Offload handles the mis-prediction of the pilot model by fetching tensors on demand and recording the input sample to improve the pilot model accuracy and avoid mis-prediction for future input.

(3) Training system for the pilot model. To generate training samples to train the pilot model, DyNN-Offload feeds a number of DyNN's input samples to different types of DyNN models, and records the execution traces of DyNNs for tensor profiling. Then, for each DyNN's input sample, DyNN-Offload partitions the resolved dataflow graph into execution blocks to maximize overlap between tensor migration and training computation. The information for those execution blocks plus DyNN's input samples and architectures become training samples for the pilot model.

IV. Design

This section presents our design and the principal intuitions.

A. Design of Input Features

The pilot model takes the following information as input features: (1) the input sample to the DyNN after embedding, (2) DyNN's static architecture, and (3) the basic NN type of DyNN. The static architecture is collected by static analysis on the DyNN code. The static architecture includes *all* model components in the DyNN whose execution is determined by the control flows. The static architecture is different from the dynamic architecture which is input-dependent. We include the static architecture as input features, such that the pilot model can be independent of DyNN architecture and hence more general. We discuss how to represent the static architecture as input features of the pilot model in this section.

1) Design Goals: When determining input features to represent the static architecture, we have two goals: (1) we cannot have too many features, because that leads to a large pilot model, causing large runtime overhead, and (2) the features should be informative to represent tensor accesses in operators.

We could use operator names (represented as numerical values) as the input features. In particular, treating the computation in the DyNN as a sequence of operators, we employ a vector and each element of the vector represents an operator in the sequence. The operator names (or operator types), indicating how tensors are accessed, are informative. However, there are three problems with this solution. (1) There are a large number of operator names, leading to a large feature space. Using an operator name-based vector as the input increases the complexity of the pilot model. (2) Some DyNNs have user-defined operators. Using the operator names as the input features lacks generality to handle a variety of DyNNs. (3) Some operators are the variant of the same operator (e.g., the ADAM optimizer) but with different names. These operators access the same tensors and have the same functionality. There is no need to distinguish them as the input features.

2) Idiom-based Representation: We use an abstract to represent DyNN's static architecture.

Idiom-based representation for operators. Each operator is characterized with six idioms. An idiom is a computation pattern, commonly found in numerical computation [9]. Using this idiom-based representation is based on our observation that the six idioms have wide coverage of computation in ML operators. We describe these idioms using the following examples, where A, B, and C are two-dimensional vectors, i and j are indexes, a is a scalar tensor, and operators step through tensors by enumerating i and j.

- Transpose: $A_{ij} = B_{ji}$
- Gather: $A_{ij} = B_{C_{ij}}$
- Scatter: $B_{C_{ij}} = A_{ij}$
- Reduction: $a = a + A_{ij}$
- Stream: $A_{ij} = A_{ij} + B_{ij}$

Fig. 3. An example of getting idiom-based representation for matmul

```
def matmul(a, b):
### the shape of input a and b are (ar, ac), (br, bc)
ar, ac = a.shape          # input [0,0,0,0,0,0,ar,ac,0]
br, bc = b.shape          # input [0,0,0,0,0,0,br,bc,0]
assert ac == br
c = torch.zeros(ar, bc)
### the shape of output c is (ar, bc)
for i in range(ar):
    c[i] = (a[i].unsequeezed(-1)
                           # transpose [1,0,0,0,0,0,0,0,0]
            *b             # stream [0,0,0,0,1,0,0,0,0]
            ).sum(dim=0)   # reduction [0,0,0,1,0,0,0,0,0]
return c
# The idiom-based representation of matmul(a,b) is
# [1,0,0,1,1,0,(ar+br),(ac+bc),0]
```

- Stencil: $A_{ij} = A_{(i-1)j} + A_{(i+1)j}$

The six idioms are pervasive and expressive. Among 300 common operators in PyTorch, all of them have these patterns [1]. These patterns are automatically identified (see Section V). We build a signature (a *nine-element vector*) for each operator using the idioms. The signature includes six elements that count the occurrence of each idiom in the operator and three elements that capture the operator input information. This results in a distinctive nine-element vector for each operator. The use of these idioms dominates DyNN execution time, and the nine-element vector is sufficient to build signatures, so other computation patterns are not considered. Figure 3 gives an example.

Idiom-based representation for DyNN. Given the source code of a DyNN, we collect the nine-element vectors for all operators in it. Those vectors are organized into a matrix (named *architecture feature matrix* or *AFM*). In AFM, each row corresponds to an operator. The row order in AFM corresponds to the operator order. In the case of a unresolved control flow, the operators in multiple branches are placed into AFM following the program order in the DyNN script. If an operator occurs multiple times, each occurrence has a row in AFM. Besides the operator representation, AFM has the control-flow representation, which is a row full of dummy values (all "0"s). The index for such a row in AFM corresponds to the control statement location in the DyNN source code. Hence, AFM captures the DyNN architecture. AFM is built offline and takes minor memory overhead (at most a few KB per DyNN).

AFM pays great attentions to reduce the complexity of the pilot model from two perspectives. *First*, the operators with the same tensor accesses and similar functionality are intentionally not distinguished to reduce the parameters of the pilot model. For example, the activation functions, `ReLU` and `Sigmoid`, use the same idioms (i.e., stream) and tensor shapes. Those operators are not distinguishable in AFM. However, this does not impact the prediction accuracy of the pilot model, because from the perspective of tensor usage, those operators have no difference. *Second*, the detailed tensor information (such as

[1]Idiom-based representation for each operator is in GitHub repo

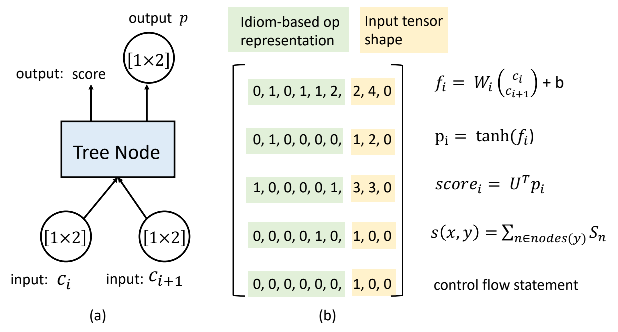


Fig. 4. An AFM example to show the static structure of the Tree-CNN shown in Figure 1. (a) A node in DyNN with input and output tensors; (b) AFM representation along with computation in operators.

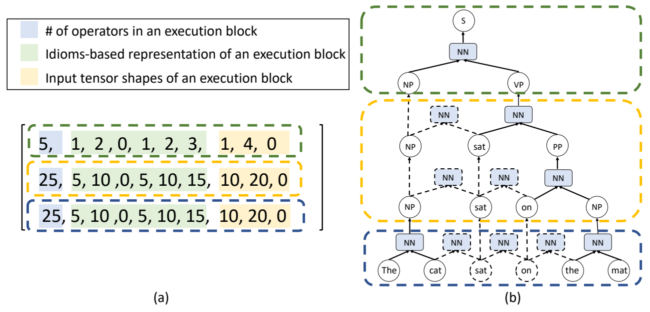


Fig. 5. An example of the pilot model output. (a) shows the output. This output partitions the DyNN execution into 3 execution blocks. (b) shows the dataflow graph of the DyNN for a given input and how the pilot model output maps back to the operators in the DyNN.

tensor association with operators) is not explicitly expressed in AFM. Instead, the tensor information is implicitly encoded into the operator information.

An example of AFM. Figure 4 shows the AFM for the Tree-CNN in Figure 1. Each node in the Tree-CNN has four operators, represented as the first four rows in AFM where the first six elements in each row correspond to the numbers of transpose, gather, scatter, reduction, stream, and stencil respectively. The last row in AFM corresponds to the control flow (Line 9 in Figure 1.a, showing six "0" as dummy values, as discussed before). The sequence of the operators and control flow follows the program order shown in Figure 1.a.

B. Output of Pilot Model

The pilot-model output is the information of operators grouped into *execution blocks*.

Execution blocks. The beginning of an execution block i is the point where tensor migration for the next execution block $i+1$ starts. To guide tensor prefetch, the output of the pilot model indicates how the operators should be organized into those execution blocks, such that tensor migration for the block $i+1$ can be overlapped with the prior block i. The knowledge of how to partition the training of the DyNN into execution blocks is learned through pilot model training (Sec. IV-D).

Output format. The pilot-model output is multiple vectors, each of which includes operator information for an execution block. Each vector has ten elements: (1) the total number of operators and control flows in the execution block (which is one number), (2) the numbers of the six idioms accumulated from all operators in the execution block (which are six numbers), and (3) the dimension sizes of input/output tensors accumulated from all operators in the execution block (which are three numbers). Given the above output and DyNN's static architecture, we can deterministically resolve dynamism (i.e., the control flows), discussed as follows.

Map pilot-model output to operators in DyNN. From the first operator, DyNN-Offload traverses DyNN's static architecture, meanwhile bookkeeping the number of operators, the number of idioms, and the dimension sizes of input/output tensors of the traversed operators. Whenever a control flow is encountered, DyNN-Offload enumerates and traverses each possible branch. The traverse continues till the last operator in the DyNN. Whenever any path leads to a bookkeeping record matching the pilot-model output, then that path (including the resolved control flows) is picked to decide tensor prefetch.

The pilot-model output should be able to be mapped to a traverse path, because during the pilot-model training, all training samples are created to have such a match. In our tests of 12,000 cases, all find an exact match. If the traverse cannot find a path matching the bookkeeping record, DyNN-Offload chooses a path whose bookkeeping record is the closest to the pilot-model output in terms of the number of operators. The above mapping process does not cause large runtime overhead, because a large DyNN does not have many control flows.

After resolving dynamism for a DyNN's training sample, the pilot-model output is stored on CPU memory to guide tensor prefetch for the training sample. The storage cost in CPU memory is small, because the ten-element vector for an execution block takes tens of bytes and the number of the blocks is typically $O(10)$.

An example of the pilot-model output. Figure 5 shows a pilot-model output to resolve dynamism in the example DyNN depicted in Figure 1. Figure 5.(b) shows the dataflow graph corresponding to the DyNN where dotted lines and solid lines show input-dependent dynamic architecture and static architecture respectively. For a given input sample in this example, the pilot-model output partitions the dataflow graph of the DyNN into three execution blocks.

C. Lightweight Pilot Model

Model topology. The pilot model consists of three input-feature layers, three parallel Multilayer Perceptrons (MLPs), and an output layer, shown in Figure 2 ③. Overall there are 3,260 trainable parameters in the pilot model. The inference time of the pilot model is small, only 30 μs.

The three input-feature layers process (1) the input sample to the DyNN, (2) the DyNN's static architecture, and (3) the basic NN type of DyNN, respectively. The basic NN type of DyNN (e.g., convolutional neural networks, LSTM, or transformer) is represented by a one-hot vector, while the DyNN's static architecture is represented by AFM.

Each MLP in the pilot model consists of four layers: an input layer, two hidden layers and an output layer. We use three small MLPs because of the following reason. There are a variety of DyNN architectures, built upon basic NN. Using a single bulk MLP in the pilot model to handle all types of DyNN significantly increases the pilot model complexity and inference time (by at least $100\times$ in our evaluation). To address this problem, we use three small MLPs in parallel in the pilot model. When using the basic NN type as an input feature to the pilot model, the pilot model activates only one of the MLPs, hence significantly reducing the inference time. We use three MLPs, since using more MLPs led to slow convergence without improving accuracy during pilot model training, and using less than three MLPs resulted in a loss of accuracy.

Embedding re-direction. We employ an embedding-redirection technique to embed the input sample to the pilot model. This technique reuses the embedding results from the DyNN. The embedding is commonly used at the beginning of the DyNN to convert each DyNN's input sample into a fixed-length vector for the following layers in the DyNN. The embedding works with input samples (training samples) with various sizes, and serves as a form of feature extraction. Using the embedding results from the DyNN, the pilot model can simplify its input layer to handle input samples with various sizes. To use the embedding from the DyNN, the embedding kernel is instrumented to copy (or re-direct) the embedding results from GPU memory to CPU memory for the pilot model to consume. The re-direction cost is smaller than using CPU for embedding, so we do not use CPU for embedding.

D. Training Pilot Model

A pilot-training sample is a pair of an input vector and an output vector. The input vector includes AFM of the DyNN and an input sample to the DyNN. The output vector (or *label*) uses the same format as the output of the pilot model (Section IV-B). We discuss collecting pilot-training samples.

Pilot-training sample collection. AFM of the DyNN is independent of DyNN's input and built using static analysis. In particular, we analyze the DyNN model script to record operator names, input tensor shapes of operators (maybe represented with variables) and control flows to build AFM. This procedure can be done by a static analysis tool for Python [51]. Pilot-training samples are collected from different types of DyNNs to make the samples representative.

DyNN-Offload collects execution information of the DyNN to generate labels for pilot-training samples. In particular, given an input sample to the DyNN, DyNN-Offload runs the DyNN and generates a dynamic execution trace. The trace includes execution order of operators, their names, input tensor shapes of each operator, and execution time of each operator. This execution trace is used to generate a pilot-training label.

Labeling. The execution trace of the DyNN gives enough profiling information for a traditional tensor-offloading method (for static NN) to decide the partition of a dataflow graph. We use the tensor-offloading method in Sentinel [57] because of its generality and short turnaround time. Using the GPU memory capacity, tensor profiling information (including execution order and time of operators), and NN topology as input, Sentinel partitions the dataflow graph to maximize the overlap between tensor migration and training computation without violating the GPU memory capacity. DyNN-Offload transforms the output of Sentinel into a representation compatible with output format of the pilot model. This representation is used as the output vector (or label) of a pilot-training sample.

E. Runtime Design

Pilot model inference. The pilot model runs on CPU. When a batch of DyNN-training samples is about to be transferred to GPU to train the DyNN, the pilot model is applied to each training sample (input sample) in the batch. The pilot-model output for each DyNN's training sample is sent to the runtime system of DyNN-Offload on CPU to trigger tensor prefetch.

Memory management and tensor migration. GPU memory is partitioned into two equal-sized buffers: one for working tensors referenced by the ongoing execution block (called *work buffer*), and the other (called *migration buffer*) for prefetching tensors for the next execution block. The two buffers switch roles once the ongoing execution block is done. The double buffering aims to hide tensor migration overhead. The two buffers have the same size to simplify management and to accommodate varying execution times across blocks. Striking a balance, the work buffer size is large enough to handle time variation in execution blocks, yet compact enough to reserve space for the migration buffer.

Since CPU triggers tensor migration for execution blocks and GPU performs execution-block-based DyNN-training, there must be a synchronization mechanism between CPU and GPU. We introduce an operator counter at CPU to record the number of operators (GPU kernels) launched on GPU. When the counter reaches the number of operators in the ongoing execution block i, CPU is aware that GPU starts to execute the block $i+1$, and starts to migrate tensors for $i+2$.

The migration buffer must evict unused tensors and prefetch tensors to be used. Eviction and prefetching could happen in parallel to better utilize interconnect bandwidth between CPU and GPU. However, we find this solution has difficulty to migrate tensors into a contiguous memory space in GPU, leading to memory fragmentation. Hence, DyNN-Offload evicts tensors first, and then prefetches tensors.

The memory management is implemented at the runtime of DyNN training. There is no need to change the DyNN.

Handling mis-prediction. The pilot model may mis-predict operator execution order. As a result, when an operator is about to be executed on GPU, tensors needed by the operator may not be on GPU memory. In this case, DyNN-Offload instruments the runtime error due to tensor missing and migrates the tensors on demand. Additionally, DyNN-Offload records the mis-prediction by recording the *resolved architecture* and input sample of the DyNN, in order to build a training sample to be used in the future offline training of the pilot model.

Furthermore, the mis-prediction case is directly used to avoid repeated mis-prediction for other DyNN-training samples. Some DyNN-training samples may lead to the same dataflow graph as the mis-prediction case. To identify such a DyNN-training sample, the output of the pilot model is compared with the output where there is mis-prediction. If the two outputs are exactly the same, then the correct execution block in the mis-prediction case is used to resolve the control flows for the new DyNN-training sample.

Impact of dynamic batching in DyNN. DyNN-training samples are often batched to improve GPU utilization [35], [81]: a batch is dynamically formed by batching operators from multiple dataflow graphs (each graph corresponds to one DyNN-training sample). Dynamic batching couples the execution of multiple dataflow graphs, but does not impact the effectiveness of DyNN-Offload, due to two reasons.

(1) Dynamic batching does not change the execution order of execution blocks in each dataflow graph of DyNN. Hence, the tensor prefetch guided by the pilot model is still useful. Also, the operator counter-based approach (Section IV-E) can still effectively set up synchronization between CPU and GPU for tensor prefetch. (2) Dynamic batching can extend the execution time of batched operators in the DyNN because of extra cache misses caused by thread block scheduling [77] and TLB misses [13]. But the extended execution time of execution blocks on GPU gives more opportunities to overlap with tensor migration. Hence the effectiveness of DyNN-Offload to hide tensor migration is not compromised.

V. Implementation

DyNN-Offload includes (1) a runtime system and (2) an offline training system to collect pilot-training samples. The runtime system is implemented on top of ONNX Runtime [45]. Since ONNX Runtime supports a variety of ML frameworks (e.g., PyTorch and TensorFlow), operating systems (e.g., Linux and Android), and hardware platforms, DyNN-Offload can benefit various DyNNs regardless of their environment. DyNN-Offload uses a LLVM-based static analysis tool [9] to count idioms in operators automatically. The runtime system has two components: tensor manager and runtime scheduler.

Tensor manager is in charge of tensor (de)allocation and handling of mis-prediction. DyNN-Offload intercepts tensor allocation API `AllocatorDefaultAlloc()` used for GPU memory allocation, and redirects it to CPU memory. DyNN-Offload initially allocates all tensors on CPU memory to avoid out-of-memory errors. To avoid memory leak, DyNN-Offload intercepts tensor `AllocatorDefaultFree()` to ensure memory space is freed regardless of the location of the tensor. Furthermore, DyNN-Offload implements a tensor fault handler leveraging the tensor hook mechanism in ONNX. The tensor fault handler is invoked when any tensor needed by GPU computation is missing in GPU memory (i.e., a mis-prediction happens) and a `cudaErrorInvalidAddressSpace` fault is reported. The handler fetches the missing tensor from CPU memory and records the mis-prediction information to a file.

Runtime scheduler is used for the pilot model inference and tensor prefetch. In particular, the pilot model is implemented as a user-defined operator `pilot_model_inference()`, which is used as the first operator in the dataflow graph for the DyNN. `pilot_model_inference()` takes a DyNN-training sample and runs the pilot model on CPU. The AFM generated from `pilot_model_inference()` is used by the ONNX runtime. Based upon operator-launching by the ONNX runtime, DyNN-Offload counts the number of launched operators and triggers tensor migration asynchronously. Within an execution block, DyNN-Offload migrates tensors without priority; DyNN-Offload waits for the completion of tensor migration and starts the computation for the next execution block.

```
0 import torch
1 import torch.nn as nn
2 from ort_support.offload as ort_support
3
4 device = ort_support.set_offload_device()
5 model = BuildModel(config)
6 model = ort_support.create_ort_trainer(device,model)
```

Fig. 6. An example of using DyNN-Offload.

Figure 5 illustrates how to use DyNN-Offload. *Only* Line 4 (deciding the offloading target device) and Line 6 (enabling training) need to be added.

Training systems for the pilot model include (1) an execution trace generator, (2) a partition simulator, and (3) a pilot-training sample generator.

The *execution trace generator* is based upon the existing tensor instrumentation infrastructures in PyTorch or TensorFlow to generate the dynamic execution trace in a Json-formatted file. The *partition simulator* consumes this file and implements the partition algorithm in Sentinel. The partition algorithm determines where tensor migration (prefetch) should be triggered. The partition simulator transforms the DyNN partition result into a representation compatible with the pilot-model output. The *pilot-training sample generator* pairs up outputs of the partition simulator, AFMs of DyNNs, and DyNN-training samples to generate pilot-training samples.

Our pilot model uses LeakyReLU as activation, SGD as optimizer, and 0.01 as learning rate. We fine-tune the hyper-parameters of the pilot model using a genetic algorithm [31]. The pilot model training happens offline, and we do not need to fine-tune it for each DyNN because the pilot is small and general, and fine-tuning it easily causes catastrophic forgetting.

VI. Evaluation

A. Methodology

Experimental setup. We use two environments: (1) four servers, each equipped with an NVIDIA RTX6000 GPU (GPU for desktop with 23GB memory) and dual Intel Xeon CPUs (totally 24 cores) and 186 GB CPU memory. This platform represents DL training environment for regular users. (2) Two high-end servers, each with four 80GB NVIDIA A100 GPUs, and dual Intel Ice Lake CPUs (totally 40 cores) and 500 GB CPU memory. This platform represents the environment for high-end users in data centers. The interconnect between CPU

TABLE II
DYNNS FOR EVALUATION. "BS" STANDS FOR BATCH SIZE.

DyNN name	DyNN (block) type	Dataset	# of blocks	BS
Tree-CNN [62]	CNN	CUB 200	128	32
UGAN [38]	RNN	circular gaussian	64	512
Tree-LSTM [72]	LSTM	SICK	128	512
var-LSTM [35]	LSTM	Reuters-21578	128	512
var-BERT [46]	transformer	wikitext-2-v1	48	32
AlphaFold [3]	evoformer	uniref90, pdb70	2,3,4	6
fixed-LSTM [71]	LSTM	Reuters-21578	128	512
fixed-Bert [12]	transformer	wikitext-2-v1	48	32

and GPU in both computing environments is 16-lane PCIe 3.0. We use CUDA Toolkit 11.2 and PyTorch 1.9.

Workloads. We evaluate six DyNNs, covering the major DyNN types, ranging from transformer-, LSTM-, and CNN-based ones. We also evaluate two static NNs (fixed-Bert and fixed-LSTM). See Table II. We build over 24,000 samples from six of the eight models in Table II to train the pilot model (to show the generalizability of the pilot model, training samples are not collected from var-LSTM and var-BERT). We evaluate 2,000 samples for each model in Table II for performance testing. Training and testing samples do not overlap.

Baselines for evaluation are summarized as follows.

- *DTR* [30] is a state-art-the-art solution based on tensor rematerialization for DyNNs. DTR frees memory space for activation tensors when the GPU memory is not large enough. DTR rematerializes the freed activation when needed.
- *Unified virtual memory (UVM)* [44] enables GPU memory oversubscription by using CPU memory, and migrates pages between CPU and GPU based on GPU's demands. UVM allows the programmer to use memory prefetch from CPU to GPU through `cudaMemPrefetchAsync()`. This mechanism must rely on the programmer to have a good knowledge on how tensors are accessed in DyNN and explicitly specify that using `cudaMemPrefetchAsync()`. However, this knowledge cannot be known a priori because of the dynamic and irregular nature of DyNN. Hence, we do not evaluate UVM with prefetch. We maximize the oversubscription rate of UVM (i.e., 2). This implies that when using UVM, the sum of CPU and GPU memory can be at most twice the size of the GPU memory, a standard that aligns with industrial practice [2], [52].
- *ZeRO-Offload* [58] is an industry-quality solution from Microsoft, aiming to optimize tensor offloading for *static* transformer models to save GPU memory based on PGO.

B. Breaking Memory Capacity Wall

We first test the largest trainable models on a single GPU with 80GB memory. We change the size of var-BERT by making it deeper (i.e., increasing the number of transformer layers) or wider (i.e., increasing the hidden size in each transformer layer). A similar method has been used in existing work [50], [56], [58], [69]. Without DyNN-Offload, PyTorch can only train var-BERT with 192 transformer layers (3.85B parameters and 1,024 as the hidden size) on A100 GPU. With DyNN-Offload, PyTorch can train var-BERT with 1,500 transformer layers (31.25B parameters and 1,024 as the hidden size), 8$\times$ *larger than without DyNN-Offload.* We also evaluate DyNN-Offload with wide var-BERT. DyNN-Offload enables 64-layer, wide var-BERT training (20.2B parameters and 8,192 as the hidden size), while without DyNN-Offload, the maximum trainable model has only 10 layers (3.2B parameters and 8,192 as the hidden size). *DyNN-Offload enables* 6.3$\times$ *larger wide-model training than without DyNN-Offload.*

TABLE III
MAX TRAINABLE BATCH SIZE (BS) AND MEMORY USAGE FOR DYNN TRAINING WITH UVM, DTR AND DYNN-OFFLOAD.

	UVM		DTR		DyNN-Offload	
	bs	mem usage	bs	mem usage	bs	mem usage
TreeCNN	48	30GB	72	36GB	96	39GB
UGAN	768	34GB	1120	38GB	1152	39GB
Tree-LSTM	352	34.5GB	560	39GB	640	44GB
var-LSTM	352	32.5GB	560	36GB	640	40GB
var-BERT	48	88GB	72	122GB	128	159GB
AlphaFold	10	94GB	20	144GB	24	192GB
fixed-LSTM	352	34GB	560	38GB	640	40GB
fixed-BERT	48	88GB	72	122GB	128	159GB

In comparison, UVM can train var-BERT with up to 7.5B parameters (consuming 2x GPU memory). Using UVM to train a larger model leads to more than 200% slow down in training time. DTR can only train var-BERT with 192 layers (3.85B parameters and 1,024 as the hidden size). Training a larger model with DTR suffers from system crashes because of DTR's internal mechanism to track tensor lifetime.

We further evaluate the largest batch size on a single GPU with 80GB memory, shown in Table III. We compare DyNN-Offload with UVM and DTR. To make the comparison fair, we set the same maximum-runtime-overhead (200%) for UVM, DTR, and DyNN-Offload. With this constraint, we evaluate which method leads to the largest batch size. Table III shows the result. With UVM, DTR and DyNN-Offload, the largest batch size is 1.17$\times$, 1.7$\times$, and 3.6$\times$ larger, compared with PyTorch without any memory saving solution. UVM is worse than DyNN-Offload, because of UVM's large runtime overhead – on-demand fetching limits memory saving. DTR is worse than DyNN-Offload, because DTR only works for activation tensors, limiting the memory saving opportunities.

C. DyNN Training Improvement

Figure 7 shows the training time. We deploy BERT and AlphaFold on an A100 GPU because of their high computing and memory demands, and deploy other DyNNs on RTX GPU. We report one-epoch time after warm-up run.

- UVM performs worst in almost all cases, because tensor migration largely happens on demand (see the discussion for Figure 8). UVM amplifies communication volume because of page-level (instead of tensor-level) migration. Also, as the batch size increases, the UVM's overhead increases significantly since more frequent GPU page faults happens when GPU memory oversubscription is more serious.

- As the batch size increases, the performance benefit of DyNN-Offload over DTR becomes larger. DyNN-Offload outperforms DTR by 35% on average.
- ZeRO-Offload only works for static NN, such as fixed-Bert. For static NN, DyNN-Offload outperforms ZeRO-Offload by 33% on average with three batch sizes, because of optimal partition decided by DyNN-Offload.

We break down training time to analyze performance. See Figure 8. We have 3 observations. (1) UVM spends a large portion of time on tensor migration (see Tree-CNN and UGAN where tensor migration takes 55% and 40% of training time). (2) Rematerialization is costly. To save memory space, DTR enables recursive tensor rematerilization, which introduces extra computation. In AlphaFold, DTR has $1.7\times$ computation time than DyNN-Offload. (3) By removing rematerialization and hiding migration, DyNN-Offload performs best.

We further compare the performance of DyNN-Offload and DTR under various GPU memory budgets. Figure 9 shows the results. In Figure 9, we include the performance of unmodified PyTorch. When the memory budget is the largest in Figure 9, all tensors can be allocated in GPU memory and the unmodified PyTorch shows the best performance, which provides a baseline to evaluate the overhead of DyNN-Offload (including the cost of tensor migration) and DTR (including the cost of recomputation) under various GPU memory budgets.

Figure 9 shows that as the memory budget becomes smaller, DyNN's training time becomes longer because of more frequent tensor-migration (in DyNN-Offload) or recomputation (in DTR). DyNN-Offload consistently outperforms DTR under various memory budgets by 12% on average (up to 28%).

Furthermore, DTR exhibits rapid performance degradation as the memory budget decreases because of its reliance on a lengthy recomputation chain for tensor recovery. The length of the computation chain increases superlinearly as the memory budget decreases. In contrast, DyNN-Offload's migration overhead is limited by the interconnect bandwidth between GPU and CPU. Until reaching the maximum hardware bandwidth, the migration overhead in DyNN-Offload increases almost linearly as the memory budget decreases.

Overhead analysis. The overhead of DyNN-Offload for a training iteration includes (1) the pilot model inference time and (2) the time of mapping the output of the pilot model to operators. For (1), the time is 30 μs. For (2), the time is about 10-15 μs. The overhead is much smaller than a single iteration time of large-scale DyNNs we evaluate (i.e., $O(100\mu s)$).

D. Scalability of DyNN-Offload

We use distributed training (see Figure 10). We employ data parallelism on 8 A100 GPUs on two nodes. The batch size per GPU is constant (20). As the scale becomes larger, the throughput increases proportionally until 4 GPUs. After that, the performance scaling slows down due to increase of inter-GPU communication, but the overhead of DyNN-Offload and on-demand tensor migration caused by mis-prediction remain stable at all scales, leading to good scalability.

TABLE IV
PILOT MODEL PERFORMANCE WITH DIFFERENT MODEL COMPLEXITY. "PM" STANDS FOR PILOT MODEL.

MLP Complexity (# of neurons)	PM Accuracy	PM Training Time	PM Inference Time	PM Memory Consumption
256	0.7	2h	5us	40KB
512	0.82	2.5h	12us	75KB
1024	0.88	3h	20us	140KB
2048	0.91	4h	42us	260KB
4096	0.93	6h	80us	530KB

E. Construction of Pilot Model

We study how to construct an effective and efficient pilot model. *We want to reduce inference time as much as possible* but with high modeling accuracy. This is especially important in use scenarios where DyNN has short iteration time. We change the number of neurons in each MLP layer of the pilot model and study its effect, shown in Table IV. When increasing neurons from 256 to 512, the model accuracy largely increases by 0.12. However, when we increase neurons beyond 512, the momentum of accuracy increase is significantly smaller, but the inference time continuously increases with a rate of approximately 2x. Hence we choose 512 for our pilot model, because of its good balance between accuracy and costs (including inference and training times).

Our pilot model can work for any transformer-, LSTM-, and CNN-based DyNN. For any new type of DyNN, the pilot model can be re-trained with incremental training technique such as curriculum learning [22], saving training efforts.

The number of mis-predictions. With 512 neurons in each MLP layer, we find less than 60 mis-predictions in each DyNN model with 3,000 testing samples.

F. Idiom-based Representation

We compare the idiom-based representation with another representation based on operator type. We give each operator type a unique ID and name this approach "global id-based representation". Using the two representations, we train two sets of pilot models. Figure 11 shows the accuracy. Given the same model complexity (in terms of the number of neurons), the idiom-based one outperforms the global id-based one in terms of accuracy by at least 19%. To reach the same accuracy, e.g. 0.88, the idiom-based one needs 2,048 neurons, while the global id-based one needs 4M neurons, which increases the inference time by $128\times$ and training time by $2.4\times$. Using the operator type largely increases the input feature space, and has to increase model complexity to improve accuracy.

G. Evaluation of DyNN Model Partition

DyNN-Offload partitions the dataflow graph of the DyNN into execution blocks to hide tensor migration. We compare it with 3 heuristics: (1) partitioning by evenly splitting the number of operators, (2) evenly splitting the training time, and (3) evenly splitting the size of all tensors. All partition methods use the same number of partitions. See Figure 12. DyNN-Offload outperforms other solutions by 14% - 24%, because

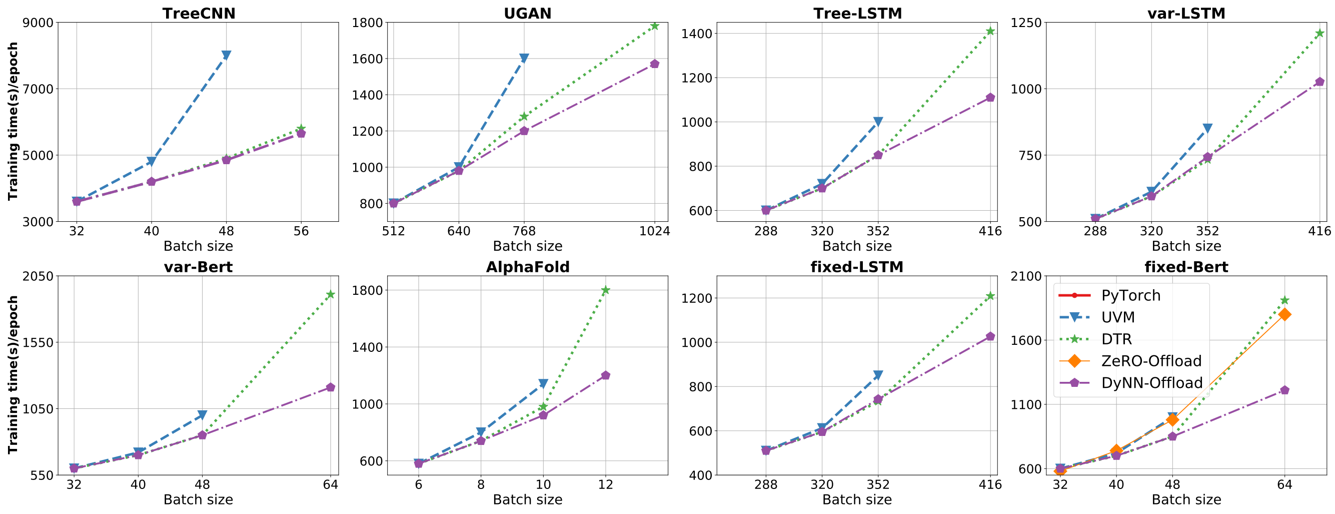


Fig. 7. Training time comparison between existing solutions and DyNN-Offload

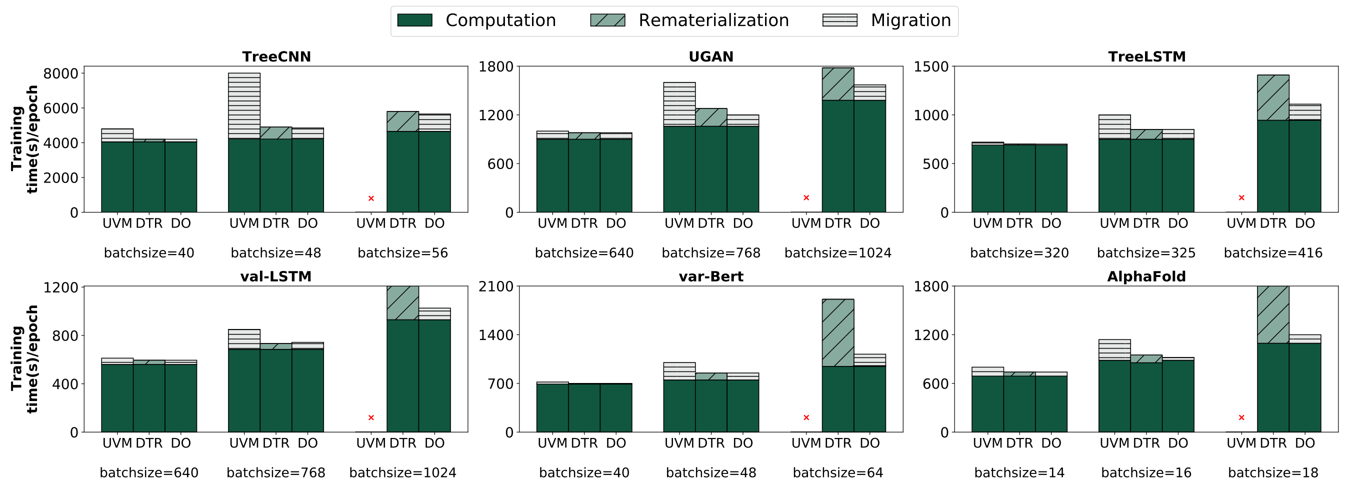


Fig. 8. Performance breakdown for DyNN-Offload, DTR and UVM. “DO” stands for DyNN-Offload.

DyNN-Offload can adaptively change the partition size to hide tensor migration (Section IV-D), while others cannot.

H. Impact of Handling Misprediction of Pilot Model

We study the number of mis-predictions without and with handling them. DyNN-Offload handles mis-predication to avoid repeated mis-prediction and improve the pilot model accuracy. Without handling mis-predication, the number of mis-prediction for Tree-CNN, Tree-LSTM, and var-BERT is 167, 109, 182, evaluated with 3,000 training samples. With handling of mis-prediction, the number of mis-prediction decreases to 59, 42, and 102 respectively. Despite causing on-demand data fetches from CPU memory, the rare occurrences of mis-predictions increase training time by less than 1%.

VII. Related Work

ML for memory and storage. LinnOS [24] uses a light NN to infer SSD performance at per-IO granularity for performance predictability. KML [4] and LearnedSSD [32] employ ML to tune storage configurations. LLAMA [36] introduces NN to predict object lifetime and avoid memory fragmentation. Cori [16] and Kleio [15] use ML to decide page migration frequency and granularity on hetero-memory.

Recent efforts use NN for prefetching. Voyager [68] builds hierarchical NN to learn address correlation and prefetch irregular sequences of memory accesses. Peled et al. [47] use a table-based reinforcement learning framework to explore the correlation between program contexts and memory addresses. Peled et al. [48] formulate prefetching as a regression problem and use a fully-connected feed-forward network as a prefetcher. Hashemi et al. [25] formulate prefetching as a classification problem and use LSTM as a prefetcher. DyNN-Offload is different from the above, because it focuses on a unique memory capacity problem for training DyNNs.

System supports for DyNNs focus on batching dynamic dataflow graphs to improve hardware utilization. Since differ-

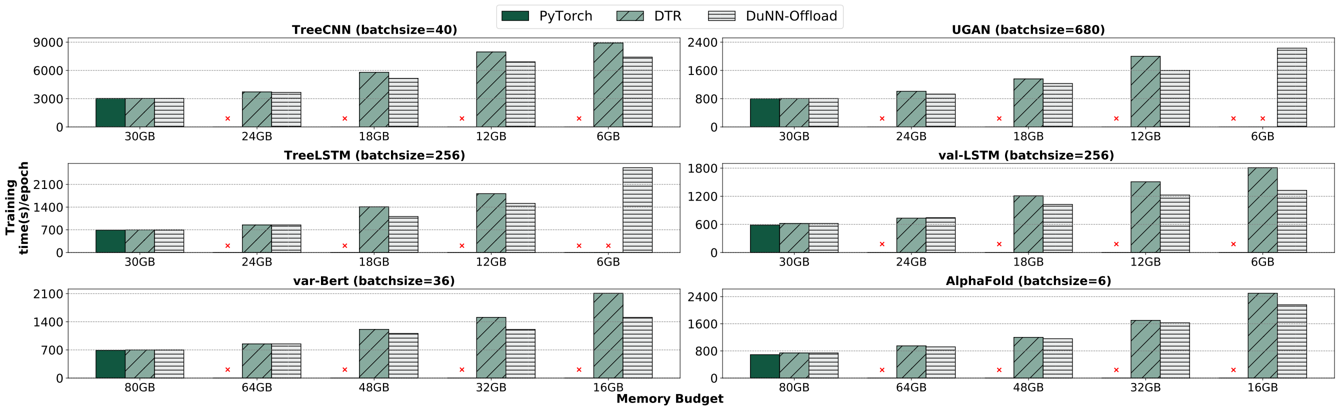


Fig. 9. Comparison between the existing solutions and DyNN-Offload with various DyNN models and GPU memory budgets. The red 'x' in the figure indicates that the DyNN model with a given GPU memory budget cannot be trained because of limited GPU memory budget.

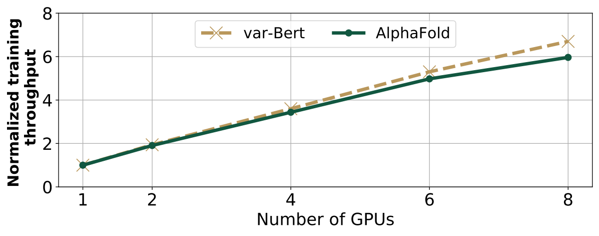


Fig. 10. Scalability evaluation of DyNN-Offload in terms of system scales (i.e., the number of GPUs).

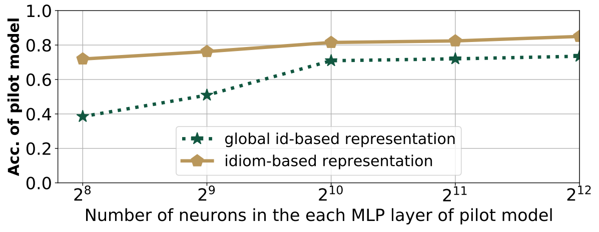


Fig. 11. Evaluation of the effectiveness of the idiom-based representation.

ent input samples use different dataflow graphs, batching them together with unresolved control flows is challenging.

TensorFlow Fold uses a depth-based batching [35], which dynamically batches nodes with the same depth and shapes in multiple dataflow graphs. However, this method misses batching opportunities (e.g., the loss functions in different dataflow graphs can be at different depths and cannot be batched using this method). DyNet [43] uses an agenda-based batching that dynamically tracks nodes with dependencies resolved for batching. However, DyNet focuses on individual nodes and is not open to dataflow graph level optimizations. Cavs [81] represents DyNN with a static vertex function and a dynamic instance-specific graph. The scheduling of the static function exposes batched execution opportunities over multiple input samples. However, Cavs needs many programming efforts. DyNN-Offload complements these works.

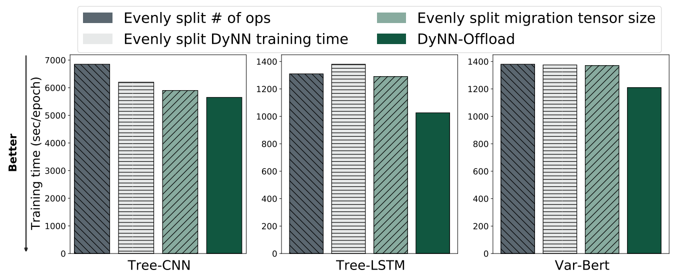


Fig. 12. The training time with DyNN-Offload and three heuristic solutions to partition DyNNs.

Input-aware performance optimization has been utilized for input-sensitive applications, e.g., streaming graph processing [7], Spark [87], sorting [14], and sparse matrix multiplication [80]. A common theme of the above work is to use input knowledge to determine how to optimize performance (e.g., deciding configurations for autotuning or computation granularity for aggregation). Also, input knowledge has been used for GPU code generation [37], [63], [74]. Different from the existing efforts, DyNN-Offload recognizes the implicit knowledge in input samples to save GPU memory.

VIII. Conclusions

DyNN-Offload is a memory management system enabling large DyNN training with limited GPU memory. Unlike the traditional PGO-based approach that lacks abilities to react to dynamism in DyNN, DyNN-Offload uses a learned approach to resolve dynamism and predict access order of tensors. We show that building a fast, accurate, and live ML model to guide performance optimization and analysis for DyNNs is feasible.

References

[1] "AlphaFold Performance: Molecule Size, Speed, Memory, and GPU," https://www.rbvi.ucsf.edu/chimerax/data/alphafold-jan2022/afspeed.html.
[2] "TensorFlow," https://github.com/tensorflow/tensorflow, 2022.

[3] G. Ahdritz, N. Bouatta, S. Kadyan, Q. Xia, W. Gerecke, T. J. O'Donnell, D. Berenberg, I. Fisk, N. Zanichelli, B. Zhang, A. Nowaczynski, B. Wang, M. M. Stepniewska-Dziubinska, S. Zhang, A. Ojewole, M. E. Guney, S. Biderman, A. M. Watkins, S. Ra, P. R. Lorenzo, L. Nivon, B. Weitzner, Y.-E. A. Ban, P. K. Sorger, E. Mostaque, Z. Zhang, R. Bonneau, and M. AlQuraishi, "Openfold: Retraining alphafold2 yields new insights into its learning mechanisms and capacity for generalization," *bioRxiv*, 2022. [Online]. Available: https://www.biorxiv.org/content/early/2022/11/22/2022.11.20.517210

[4] I. U. Akgun, A. S. Aydin, A. Shaikh, L. Velikov, and E. Zadok, "A Machine Learning Framework to Improve Storage System Performance," in *ACM Workshop on Hot Topics in Storage and File Systems*, 2021.

[5] M. Artetxe, S. Bhosale, N. Goyal, T. Mihaylov, M. Ott, S. Shleifer, X. V. Lin, J. Du, S. Iyer, R. Pasunuru, G. Anantharaman, X. Li, S. Chen, H. Akin, M. Baines, L. Martin, X. Zhou, P. S. Koura, B. O'Horo, J. Wang, L. Zettlemoyer, M. Diab, Z. Kozareva, and V. Stoyanov, "Efficient large scale language modeling with mixtures of experts," 2022.

[6] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in *International Conference on Learning Representations (ICLR)*, 2015.

[7] A. Basak, Z. Qu, J. Lin, A. R. Alameldeen, Z. Chishti, Y. Ding, and Y. Xie, "Improving streaming graph processing performance using input knowledge," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1036–1050.

[8] S. Boslaugh, *Statistics in a Nutshell*. O'Reilly Media, Inc., 2014.

[9] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole, "An Idiom-Finding Tool for Increasing Productivity of Accelerators," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2011.

[10] D. Chen, D. D. Lepikhin, H. Lee, M. Krikun, N. Shazeer, O. Firat, Y. Huang, Y. Xu, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," 2020.

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

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

[13] B. Di, D. Hu, Z. Xie, J. Sun, H. Chen, J. Ren, and D. Li, "TLB-pilot: Mitigating TLB Contention Attack on GPUs with Microarchitecture-Aware Scheduling," *Transactions on Architecture and Code Optimization*, vol. 19, no. 1, 2021.

[14] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 379–390, 2015.

[15] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence," in *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.

[16] T. D. Doudali, D. Zahka, and A. Gavrilovska, "Cori: Dancing to the Right Beat of Periodic Data Movements over Hybrid Memory Systems," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[17] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, B. Zoph, L. Fedus, M. Bosma, Z. Zhou, T. Wang, Y. E. Wang, K. Webster, M. Pellat, K. Robinson, K. Meier-Hellstern, T. Duke, L. Dixon, K. Zhang, Q. V. Le, Y. Wu, Z. Chen, and C. Cui, "Glam: Efficient scaling of language models with mixture-of-experts," 2022.

[18] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, B. Zoph, L. Fedus, M. Bosma, Z. Zhou, T. Wang, Y. E. Wang, K. Webster, M. Pellat, K. Robinson, K. Meier-Hellstern, T. Duke, L. Dixon, K. Zhang, Q. V. Le, Y. Wu, Z. Chen, and C. Cui, "GLaM: Efficient Scaling of Language Models with Mixture-of-Experts," in *International Conference on Machine Learning*, 2022.

[19] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022. [Online]. Available: http://jmlr.org/papers/v23/21-0998.html

[20] J. Feng and D. Huang, "Optimal gradient checkpoint search for arbitrary computation graphs," 2021.

[21] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-Efficient Backpropagation through Time," in *International Conference on Neural Information Processing Systems*, 2016.

[22] G. Hacohen and D. Weinshall, "On The Power of Curriculum Learning in Training Deep Networks," *CoRR*, vol. abs/1904.03626, 2019.

[23] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," *CoRR*, vol. abs/2102.04906, 2021.

[24] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi, "LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[25] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning Memory Access Patterns," in *International Conference on Machine Learning*, 2018.

[26] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[27] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[28] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger, "Multi-Scale Dense Networks for Resource Efficient Image Classification," in *International Conference on Learning Representations (ICLR)*, 2018.

[29] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica, "Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization," in *Proceedings of Machine Learning and Systems (MLSys)*, 2020.

[30] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock, "Dynamic tensor rematerialization," 2021.

[31] A. Lentzas, C. Nalmpantis, and D. Vrakas, "Hyperparameter Tuning Using Quantum Genetic Algorithms," in *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, 2019.

[32] D. Li and J. Huang, "A Learning-based Approach Towards Automated Tuning of SSD Configurations," *CoRR*, vol. abs/2110.08685, 2021.

[33] X. Liang, X. Shen, J. Feng, L. Lin, and S. Yan, "Semantic object parsing with graph LSTM," *CoRR*, vol. abs/1603.07063, 2016.

[34] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime Neural Pruning," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[35] M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig, "Deep Learning with Dynamic Computation Graphs," in *International Conference on Learning Representations (ICLR)*, 2017.

[36] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, "Learning-Based Memory Allocation for C++ Server Workloads," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[37] A. Magni, D. Grewe, and N. Johnson, "Input-aware auto-tuning for directive-based GPU programming," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013, pp. 66–75.

[38] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks," 2017.

[39] A. Mirhoseini, "Pushing the Limits of Scaling Laws in the Age of Generative Models," Keynote at Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2023.

[40] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in *Symposium on Operating System Principles*, 2019.

[41] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-Efficient Pipeline-Parallel DNN Training," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2021.

[42] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," 2021.

[43] G. Neubig, Y. Goldberg, and C. Dyer, "On-the-fly Operation Batching in Dynamic Computation Graphs," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[44] Nvidia, "Unified Memory," https://devblogs.nvidia.com/unified-memory-in-cuda-6/, 2019.

[45] ONNX, "ONNX Runtime," https://onnxruntime.ai/.

[46] R. Pappagari, P. Żelasko, J. Villalba, Y. Carmiel, and N. Dehak, "Hierarchical transformers for long document classification," 2019.

[47] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic Locality and Context-based Prefetching Using Reinforcement Learning," in *International Symposium on Computer Architecture*, 2015.

[48] L. Peled, U. Weiser, and Y. Etsion, "A Neural Network Prefetcher for Arbitrary Memory Access Patterns," *ACM Transactions on Architecture and Code Optimization*, 2019.

[49] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based GPU Memory Management for Deep Learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[50] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj, "Training large neural networks with constant memory using a new execution algorithm," *CoRR*, vol. abs/2002.05645, 2020.

[51] PyTorch, "PyTorch Profiler," https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html, 2021.

[52] PyTorch, "UVM in torchRec," https://github.com/pytorch/torchrec/blob/main/examples/sharding/uvm.ipynb, 2022.

[53] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[54] S. Rajbhandari, C. Li, Z. Yao, M. Zhang, R. Y. Aminabadi, A. A. Awan, J. Rasley, and Y. He, "DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale," *CoRR*, vol. abs/2201.05596, 2022.

[55] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations Toward Training Trillion Parameter Models," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[56] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning," *CoRR*, vol. abs/2104.07857, 2021.

[57] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[58] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "ZeRO-Offload: Democratizing Billion-Scale Model Training," in *USENIX Annual Technical Conference*, 2021.

[59] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[60] C. Riquelme, J. Puigcerver, B. Mustafa, M. Neumann, R. Jenatton, A. Susano Pinto, D. Keysers, and N. Houlsby, "Scaling vision with sparse mixture of experts," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 8583–8595. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/48237d9f2dea8c74c2a72126cf63d933-Paper.pdf

[61] S. Roller, S. Sukhbaatar, A. Szlam, and J. Weston, "Hash layers for large sparse models," 2021.

[62] D. Roy, P. Panda, and K. Roy, "Tree-cnn: A hierarchical deep convolutional neural network for incremental learning," 2019.

[63] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke, "Adaptive input-aware compilation for graphics engines," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 13–22.

[64] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. Nelson, A. Bridgland *et al.*, "Improved protein structure prediction using potentials from deep learning," *Nature*, vol. 577, no. 7792, pp. 706–710, 2020.

[65] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, "Mesh-TensorFlow: Deep Learning for Supercomputers," in *Neural Information Processing Systems (NeurIPS)*, 2018.

[66] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," in *International Conference on Learning Representations (ICLR)*, 2017.

[67] L. Shen, Z. Wu, W. Gong, H. Hao, Y. Bai, H. Wu, X. Wu, J. Bian, H. Xiong, D. Yu, and Y. Ma, "Se-moe: A scalable and efficient mixture-of-experts distributed training and inference system," 2023.

[68] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A Hierarchical Neural Model of Data Prefetching," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[69] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2019.

[70] J. M. Siskind and B. A. Pearlmutter, "Divide-and-conquer checkpointing for arbitrary programs with no user annotation," *Optimization Methods and Software*, vol. 33, no. 4-6, p. 1288–1330, Sep 2018. [Online]. Available: http://dx.doi.org/10.1080/10556788.2018.1459621

[71] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to Sequence Learning with Neural Networks," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2014.

[72] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," 2015.

[73] N. Team, M. R. Costa-jussà, J. Cross, O. Çelebi, M. Elbayad, K. Heafield, K. Heffernan, E. Kalbassi, J. Lam, D. Licht, J. Maillard, A. Sun, S. Wang, G. Wenzek, A. Youngblood, B. Akula, L. Barrault, G. M. Gonzalez, P. Hansanti, J. Hoffman, S. Jarrett, K. R. Sadagopan, D. Rowe, S. Spruit, C. Tran, P. Andrews, N. F. Ayan, S. Bhosale, S. Edunov, A. Fan, C. Gao, V. Goswami, F. Guzmán, P. Koehn, A. Mourachko, C. Ropers, S. Saleem, H. Schwenk, and J. Wang, "No language left behind: Scaling human-centered machine translation," 2022.

[74] P. Tillet and D. Cox, "Input-Aware Auto-tuning of Compute-bound HPC kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[75] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018.

[76] Wikipedia, "Jaccard Index." [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index

[77] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations," in *International Conference on Supercomputing (ICS)*, 2015.

[78] Z. Wu, D. Zhao, Q. Liang, J. Yu, A. Gulati, and R. Pang, "Dynamic sparsity neural networks for automatic speech recognition," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021.

[79] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "AntMan: Dynamic Scaling on GPU Clusters for Deep Learning," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[80] Z. Xie, G. Tan, W. Liu, and N. Sun, "IA-SpGEMM: An Input-Aware Auto-tuning Framework for Parallel Sparse Matrix-Matrix Multiplication," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2019.

[81] S. Xu, H. Zhang, G. Neubig, W. Dai, J. K. Kim, Z. Deng, Q. Ho, G. Yang, and E. P. Xing, "Cavs: An Efficient Runtime System for Dynamic Neural Networks," in *Proceedings of USENIX Conference on USENIX Annual Technical conference (ATC)*, 2018.

[82] F. Xue, Z. Shi, F. Wei, Y. Lou, Y. Liu, and Y. You, "Go wider instead of deeper," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 8, 2022, pp. 8779–8787.

[83] A. Yang, J. Lin, R. Men, C. Zhou, L. Jiang, X. Jia, A. Wang, J. Zhang, J. Wang, Y. Li, D. Zhang, W. Lin, L. Qu, J. Zhou, and H. Yang, "M6-t: Exploring sparse expert models and beyond," 2021.

[84] J. A. Yang, J. Huang, J. Park, P. T. P. Tang, and A. Tulloch, "Mixed-Precision Embedding Using a Cache," in *Conference on Machine Learning and Systems*, 2020.

[85] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training," in *USENIX Annual Technical Conference (ATC 21)*, 2021.

[86] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang, "Slimmable Neural Networks," in *International Conference on Learning Representations (ICLR)*, 2019.

[87] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware High Dimensional Configurations Auto-tuning of In-Memory Cluster Computing," in *International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2018.