TSPLIT: Fine-grained GPU Memory Management for Efficient DNN Training via Tensor Splitting

Xiaonan Nie† Xupeng Miao† Zhi Yang† Bin Cui†§
†*School of Computer Science & Key Laboratory of High Confidence Software Technologies (MOE), Peking University*
§*Institute of Computational Social Science, Peking University(Qingdao), China*
{*xiaonan.nie, xupeng.miao, yangzhi, bin.cui*}*@pku.edu.cn*

2022 IEEE 38th International Conference on Data Engineering (ICDE) | 978-1-6654-0883-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDE53745.2022.00241

***Abstract*—Since Deep Neural Networks (DNNs) are deeper and larger, performing DNNs training on existing accelerators (e.g., GPUs) is challenging due to their limited device memory capacity. Existing memory management systems reduce the memory footprint via tensor offloading and recomputing. However, this coarse-grained, one-tensor-at-a-time memory management often incurs high peak GPU memory usage and cannot fully utilize available hardware resources (e.g., PCIe). In this paper, we propose TSPLIT, a fine-grained DNN memory management system that breaks apart memory bottlenecks while maintaining the efficiency of DNNs training. TSPLIT achieves this by proposing a model-guided approach to holistically exploit the tensor-split and its joint optimization with out-of-core execution methods (via offload and recompute). We further provide an efficient implementation of TSPLIT with proposed splittable tensor abstraction, profiling-based planner, and optimized DNN runtime. Evaluations on 6 DNN models show that compared to vDNN and SuperNeurons, TSPLIT can achieve maximum model scale up to $10.5\times$ and $3.1\times$ and throughput improved up to $4.7\times$ and $2.7\times$ under the same memory over-subscription, respectively.**

***Index Terms*—Deep Learning System, Memory Management, Large Model Support.**

I. INTRODUCTION

Enabled by the availability of enormous data, deep neural networks (DNNs) have achieved great success in various domains, such as computer vision, graph mining, and natural language processing [1, 2, 3, 4, 5]. Recently, there is a trend for deep learning community to use larger DNNs [6, 7, 8, 9, 10, 11] to analyze massive volumes of data and solve more complex tasks, such as high resolution image segmentation and large-scale machine translation [12]. Also, empirical evidence shows that the size of state-of-the-art NLP models has been increasing at a rate of $240\times$ every 2 years [13]. The exponential growth of model scale consists of expansions in multiple dimensions, such as data sample dimension (e.g., batch size, sample length) and model parameter dimension (e.g., hidden size in Transformer, channel size in CNNs). It brings large amounts of memory requirements for the model training, which is significantly challenging to these expensive AI accelerators (e.g., GPU). For example, we increase both the dimension of *data samples* and *model parameters* (e.g., hidden size) to enlarge the model scale of BERT-Large models to show their memory requirement in Figure 1 and present four mainstream GPUs to show the trainable model scale (below the corresponding black line). The gap between the increasing size of DNNs and considerably small device memory limits the exploration of more advanced DNN architectures.

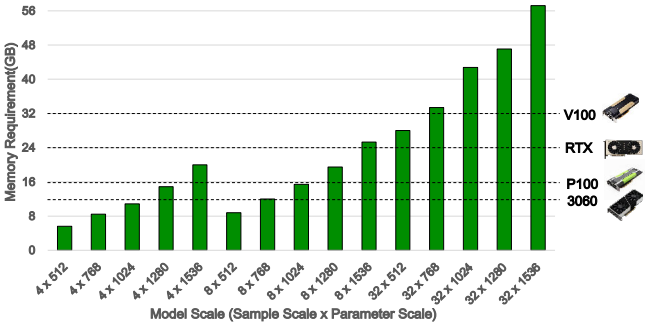


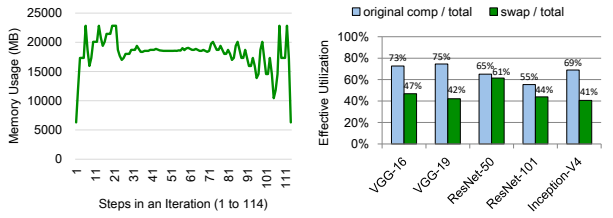
Fig. 1. The memory requirement of BERT-Large (24-layer Transformer) under different model scale (*sample scale* $\times$ *parameter scale*). Specifically, the sample scale refers to batch size as well as the parameter scale refers to hidden size in Transformer. On the right, we present the max trainable model scale with mainstream NVIDIA GPUs, e.g., 4×1280 for P100 .

TABLE I
THE FINAL PERFORMANCE ON MRPC DATASET OF BERT AT DIFFERENT SAMPLE SCALE AND PARAMETER SCALE, WHERE WE FIX THE PARAMETER SCALE AS 1024 AND THE SAMPLE SCALE AS 64, RESPECTIVELY [10].

Model Scale	**Sample**				**Parameter**	
	4	8	16	32	768	1024
Accuracy	68.38%	70.10%	82.11%	83.82%	86.7%	86.9%

Typically, there is an optimal value or range of values for sample size regarding each DNN model. Unfortunately, the range of possible sample sizes is limited by GPU memory. For example, the recommended batch size for BERT-large is 32 [10] and the accuracy could be further improved with larger batch size [14]. But the maximum supportable batch size is only 9 on P100 and 24 on V100, respectively, far from the optimal value. To demonstrate this, we finetune BERT on the MRPC dataset at different model scale with TSPLIT. The results in Table I present the necessity to support larger batch sizes to get better final accuracy. Recent contrastive learning methods have also shown that larger batches help learn better representations [15, 16]. Therefore, TSPLIT and other GPU memory management systems [17, 18, 19], attempt to break the GPU memory boundaries which prevent from choosing the optimal sample scale and parameter scale, making it more attractive for users who cannot access more than a single GPU, or users who want to minimize resource usage.

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00241



(a) Memory usage for VGG-16 (b) Effective resource utilization

Fig. 2. Illustrations of SuperNeurons on different DNNs with maximum batch size (i.e., in Table IV) on TITAN RTX GPU. Figure 2(a) SuperNeurons generates multiple high memory peaks, which bottleneck the trainability when executing the VGG model. Figure 2(b) demonstrates that even combining swap and recompute, SuperNeurons still incurs significant performance overhead of 25% $\sim$ 45% across the 5 DNN models, with a low PCIe resource utilization of 45.6% on average.

Distributed systems can mitigate above problems by involving multiple GPUs in training, while it brings high communication costs and increases the system complexity [20, 21, 22]. Meanwhile, the memory capacity on a single GPU still cannot be efficiently utilized. The memory footprint of DNN training is mainly occupied by parameters, feature maps, and their gradients [19]. And various memory optimizations are proposed to reduce the footprint. One approach is to adopt model compression [23, 24, 25] techniques such as *quantization* or *sparsification* [26] to compress tensors' size during training. However, this approach usually affects the final accuracy of models and requires heavy hyper-parameter tuning.

Another approach is to evict feature map tensors in the forward pass and regenerate them by *recompute* or *swap* in the backward pass, which is more promising because of no accuracy loss. Existing approaches manage to make strategic decisions about *swap* and *recompute* for each tensor to reduce the extra introduced time-costs. For example, TFLMS [27] and vDNN [19] only utilize *swap* for feature map tensors according to their execution order. SuperNeurons [17] determine *swap* and *recompute* strategies based on the layer type, e.g. activations of convolution layers are swapped out while batch normalization layers are recomputed.

However, the tensor-wise GPU memory strategies (such as *swap* and *recompute*) lead to two major inefficiencies: (1) The **trainability** would be restricted by operations producing the largest intermediate tensors, which generates high memory peak and pressure. Figure 2(a) shows SuperNeurons generates multiple high memory peaks, which bottleneck the trainability when executing the VGG model. Similar patterns can also be observed in other DNN models. (2) The coarse-grained, one-at-a-time tensor swap/recompute limits the **training efficiency**. A large tensor must be entirely swapped from the GPU before releasing memory for executing operations blocked under memory pressure, which hinders the scheduling ability of GPU memory managers and incurs large performance overheads. Figure 2(b) demonstrates that even combining swap and recompute, SuperNeurons still incurs a significant performance overhead of 25%$\sim$ 45% across the 5 DNN models, with a low PCIe recourse utilization of 45.6% on average. The above limitation worsens with increasingly deeper and wider DNNs.

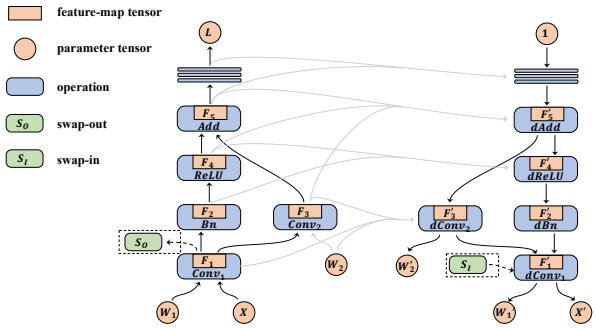
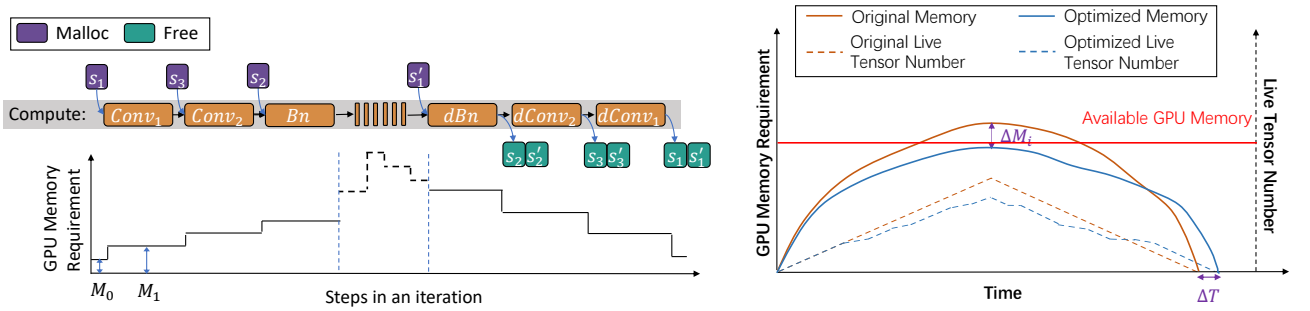


Fig. 3. A computation graph, includes operations, dependency (edge) and tensors. $Conv_1$ and $dConv_1$ represent the forward and backward computation operations. F_1 is the outputs tensor of $Conv_1$ and $F_1{}'$ is the gradient tensor of F_1. S_o and S_i represent two optional operations under memory-constrained DNN training, where S_o represents offloading tensors from GPU to CPU and S_i represents the tensor movement in the opposite direction.

In this paper, we present TSPLIT, a deep learning system that provides fine-grained tensor memory management. The key novelty of TSPLIT lies in both the mechanism design and system implementation. In terms of mechanism design, TSPLIT breaks the operation boundary of a tensor with the *tensor-splitting* primitive, which allows performing memory operations (e.g., swap or evict) on the fine-granularity of micro-tensors. The combination of splitting operations (what and how to split) and out-of-core operations (what and when to swap or generate) provides the chance of reducing peak memory usage and improving the overlap between GPU computation and memory transfer. To efficiently cope with a large search space from micro-tensors, we propose a model-guided search mechanism driven by the observation that most DNN's dataflow graph is available before execution and exhibits predictable performance characteristics. In terms of system implementation, we provide a splittable tensor abstraction called *sTensor*, and a computation graph profiler and executor for efficient DNN training. We conduct evaluations on 6 popular DNN models. The results show that compared to vDNN and SuperNeurons, TSPLIT can achieve maximum model scale up to 10.5$\times$ and 3.1$\times$ and throughput up to 4.7$\times$ and 2.7$\times$ under the same memory over-subscription.

To summarize, our main contributions are:

- We target GPU memory footprints in DNN training and propose the tensor splitting approach to improve trainability and efficiency with fine-grained memory optimization.
- We propose a model-guided planning algorithm to efficiently search the optimal configuration information of each tensor.
- We build a prototype of deep learning memory optimization system, TSPLIT, to implement the fine-grained tensor operations.
- Evaluations on various DNN workloads show that TSPLIT can significantly outperform state-of-the-art baselines on both training ability and efficiency.



(a) Execution operation schedule and its memory requirement each operation.

(b) Memory requirement and live tensor number during training with and without memory optimization.

Fig. 4. Figure 4(a) represents a possible computation schedule of Figure 3, where S_O and S_I are excluded. Figure 4(b) represents the memory requirement and live tensor number during training with and without memory optimization. To reduce the peak memory, the memory-optimized execution involves re-generation operations (i.e., swap-in/recompute) that delay the computation towards the tail, thus leading to more live tensors.

Algorithm 1: Construct Execution Schedule for DNNs

Data: $\mathcal{G}$: Computation Graph
Result: $\mathcal{O}$: Operation Schedule
Function *Execution_Scheduler(layer):*
  /*Topo Sort of DFS Manner*/ ;
  $\mathcal{O}$.push(layer) ;
  for *next_layer in layer* $\rightarrow$ *outputs()* **do**
    next_layer $\rightarrow$ ref_cnt -= 1 ;
    if *next_layer* $\rightarrow$ *ref_cnt = 0* **then**
      Execution_Scheduler(next_layer);

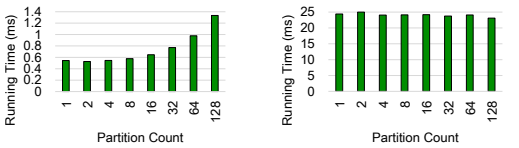
II. Background

Deep Neural Network Training. Deep neural networks consist of multiple mathematical functions as layers. Each function takes the outputs of functions in the previous layer as the inputs and produces an output as a function of the inputs. Such functions naturally translate into a series of matrix or *tensor* operations, such as matrix algebra, convolution, pooling, etc. Thus, the computation of DNNs is typically expressed as a dataflow graph (DFG) representation [28, 29, 30], where the nodes are operations, and the edges are tensors. Figure 3 represents an identical computation dataflow graph, and the memory footprint is mainly consumed by feature maps (i.e., F_1), gradient maps (i.e., $F_1{}'$), and model parameters (i.e., W_1). Due to the dependency, the feature maps can't be deleted until their gradients are computed completely, which accounts for the major memory usage. After users define DNN models, deep learning systems first utilize a scheduler to construct an execution order according to the computation graph and then execute the operations one by one. The execution scheduler of TSplit is shown a Algo 1, which takes the first layer as input and recursively searches the subsequent layers in the Depth-First-Search (DFS) manner. For example, Figure 4(a) shows the execution schedule and its corresponding memory usage of Figure 3, and the malloc or free of tensors only happen at the beginning or end of each operation. DNNs are trained on multiple feed-forward and backward-propagation passes iteratively to minimize the prediction error of labeled datasets. The feed-forward pass takes a batch of training input (e.g., a set of images for an image classification task), and executes the forward computation graph to get the model outputs Y (e.g., prediction labels). The following loss function L is used to measure the difference between Y and the ground truth (e.g., true labels) as the error or loss of the network. The error values are then propagated back in the back-propagation pass, which executes the backward graph to obtain the gradients $\frac{\partial L}{\partial w}$ of each model parameter w for updating.

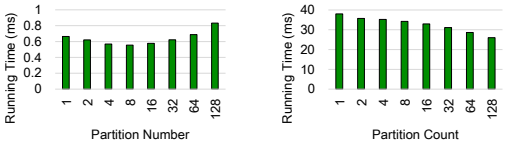
TABLE II
The distribution of tensors' size in BERT-Large

Size(MB)	< 1	$1 \sim 10$	$10 \sim 50$	$50 \sim 100$	$100 \sim 500$	> 500
Percentage	5.03%	18.25%	8.94%	45.25%	9.12%	13.41%

GPU Memory Management. GPUs have become a de facto standard for DNN training. However, recent advances in deep learning emphasize the importance of using large DNN models to improve the model quality and accelerate convergence [12, 31, 32]. Such challenging trends have driven away from computation bound and more towards memory bound. Prior works [33, 34, 35] reduce memory footprint by evicting tensors in the forward phase and adopt *swap* or *recompute* strategies to regenerate in the backward phase. Thanks to the large time gap between feature maps used in forward and backward phases, *swap* feature map tensors between CPU and GPU is beneficial, which considers CPU memory as a temporary cache. However, synchronization between the computation and data-transfer may sometimes cause inefficiencies. *Recompute* reduces memory footprint by recomputing the corresponding sub-graph to generate feature maps in the backward pass, which would cause extra computation time. Figure 4(b) shows the memory requirement curve and the live tensor number curve with and without memory optimizations at different time. After applying memory optimization techniques, several tensors are temporarily released from GPU and then the peak memory usage is reduced by ΔM_i while the overall execution time is increased by ΔT. vDNN [19] first proposes *swap* strategy and employs a layer-wise memory management. Layup [34] and SuperNeurons [17] take the characteristic of each operation into consideration and further combines *swap* with *recompute* [36] . SwapAdvisor [33] first searches the operation schedule and memory allocation policies and



(a) Conv2d: (512, 3, 32, 32) (b) Conv2d: (512, 3, 224, 224)



(c) Relu: (512, 3, 32, 32) (d) Relu: (512, 3, 224, 224)

Fig. 5. The impact of split number on the operator performance, the caption of each sub-graph is the size of input tensor, the x-axis represents the split number and the y-axis represents the time of all split operations in total.

TABLE III
NOTATIONS

Symbols	Definitions
$\mathcal{G}$	Computation Graph
Op_i	The ith operation
s_i	The output sTensor of i-th operation
OOM	Out of Memory
size(s_i)	The size of tensor s_i
c_i	Memory management strategy configuration for tensor s_i
$\mathcal{C}$	Strategy configurations for all tensors
$\Delta M_i[s_j, c]$	Memory reduction of applying c on s_j
$\Delta T_i[s_j, c]$	Extra time cost of applying c on s_j
B	the bandwidth between CPU and GPU

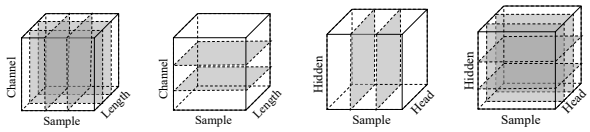
then makes *swap* decisions. KARMA [35] combines these efforts with model parallelism to support distributed DL model training. Overall, existing approaches [33, 37, 38] are concentrating on tensor-wise memory management, which limits the swapping policy we can explore and leads to low hardware utilization and efficiency.

III. MOTIVATION

A. Tensor Splitting

The minimum granularity of a memory operation in existing GPU memory management is the entire tensor. We find such coarse-grained, one-tensor-at-a-time memory operation restricts the full performance potential due to the execution primitive: For operators of large input and output tensors, their input tensor cannot be *evicted* and output tensors cannot be *swapped* out before the completion of operators, resulting in resource under-utilization and undesirable memory usage peaks. Although model designers often avoid having large tensors in their models, these tensors are inevitable and account for a large proportion. We analyze the distribution of tensors' size in a popular model, BERT-Large, and present the results in Table II. We can see that the model has many large tensors, e.g., tensors of size $>$ 500MB accounts for 13.41%, demonstrating the reason for the inefficiency of existing methods operating at the full tensor granularity.

To mitigate these limitations, our basic idea is to introduce **tensor splitting** to split a tensor into multiple independent



(a) Sample (Conv) (b) Parameter (Conv) (c) Parameter (Multi-Head Attention) (d) Parameter (Multi-Head Attention)

Fig. 6. Illustrations of Tensor-Partition along the sample/ parameter dimension. For convolution, typical layer of CNNs, we can split the sample size (e.g., number of images) and length (e.g., height and width for images) for sample splitting, and channel size for parameter splitting. For multi-head attention, typical layer of Transformer, we can split the sample size(e.g., number of sentences and length of sentences) for sample splitting, as well as hidden size and numbers of heads for parameter splitting.

micro-tensors. Each is a fine-grained unit for a single memory operation (e.g., allocate/evict and swap/recompute). With tensor-partitioning, we could allow early swapping of output tensors at micro-tensor granularity, improving the PCIe utilization and execution efficiency. Further, we can evict an input micro-tensor to make room for executing the blocked micro-tensor operator, reducing the peak memory usage. Partitioning also brings an additional benefit of reducing the workspace of operators (e.g., FFT-based convolution operator).

B. Challenges & Opportunities.

However, the benefits of partition come at the better utilization of memory transfer bandwidth and reducing unnecessary recomputation. Meanwhile, the partition imposes different impacts on different operators in terms of execution time. As demonstrated in Figure 5, we see that the operator execution time changes along with the partition number, and different operators exhibit different patterns. As shown by Figure 6, we could exploit tensor split in the sample or parameter dimension, providing a more comprehensive memory optimization space. Therefore, a key challenge of introducing graph-partitioning for reducing memory consumption is to find an efficient combination of partition strategy and memory management strategy (e.g., swap or recompute). We must address (1) how to partition and manage the memory of input/output tensors for a single operator, and (2) how to optimize the partitioning and memory management of tensors for different operators over the dataflow graph. Both problems are made difficult and distinct from the partition in DNN parallelization (over multi-GPUs) by the much larger *joint* search space of tensor splitting and memory management. Fortunately, most DNNs' dataflow graph is usually known prior to execution, and the operators often exhibit deterministic performance, therefore, their execution times and memory usage can be obtained through profiling. This allows us to search for the best combination of partition and memory management strategies prior to execution to maximize performance and trainability.

IV. TSPLIT MEMORY MANAGEMENT

TSPLIT proposes a fine-grained DNN memory management system that breaks apart memory bottlenecks of training and greatly improves the efficiency of large DNNs training. In this section, we first formulate the problem of memory-constrained

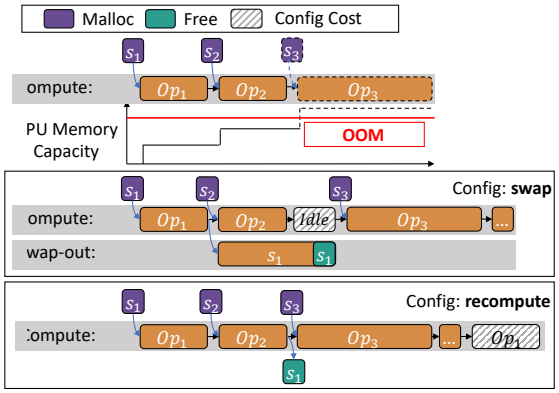


Fig. 7. Illustrations of cost estimation on the non-split strategy and split strategy. The memory bottleneck is appeared at Op_3, the *swap* or *recompute* option is used on s_1 to reduce memory by $size(t_1)$ respectively.

deep learning training, then analyze the benefit and cost of each memory optimization technique in detail and finally propose the model-guided algorithm to design optimization for each tensor.

A. System Objective

According to the dependencies of the computation graph, TSPLIT builds the execution operation schedule as Algo 1, as a manner of Depth-First-Search (DFS). Multi-branch neural networks may have different topological order, and Figure 4(a) represents a possible computation schedule of Figure 3, where S_O and S_I are excluded. Meanwhile, we describe GPU memory requirement at each operation according to the schedule, and tensors' allocation and de-allocation only happen at the beginning and end of operations. Tensors reside in GPU during their lifetime, which is defined as the interval between its allocation and de-allocation.

The overall execution time T of regular graphs can be predicted by summing the execution time T_i of each operation Op_i: $T = \sum_{i=1}^{N} T_i$. The corresponding GPU memory requirement M_i when executing Op_i can also be predicted by summing the total size of live tensors t_i: $M_i = \sum_{t_i \ is \ live} size(t_i)$ For example, the initial memory requirement m_0 in Figure 4(a) is total size of $\{X, W_1, W_2\}$, which only contains model parameters and input data.

When executing the first operation $Conv_1$, M_0 turns into M_1, the total size of $\{X, W_1, W_2, s_1\}$, and M_{N-2} = $\{X, W_1, W_2, s_1, s_3, s_1', s_3', W_2'\}$ when executing $dConv_2$. However, the peak memory requirement may exceed GPU available memory (Out-Of-Memory, OOM), which incurs *memory bottleneck*. In this case, TSPLIT employs memory management strategies to break apart this bottleneck, including *recompute*, *swap* and *split*, as we shall detail in Sec. IV-B.

Figure 4(b) shows the memory requirement curve and the live tensor number curve with and without memory optimizations at different time. By applying several strategies on tensors, we could reduce memory requirement by $\Delta M(S)_i$ when executing Op_i, at cost of increasing the overall execution time by $\Delta T(S)$. As shown by Equation 1, the memory-

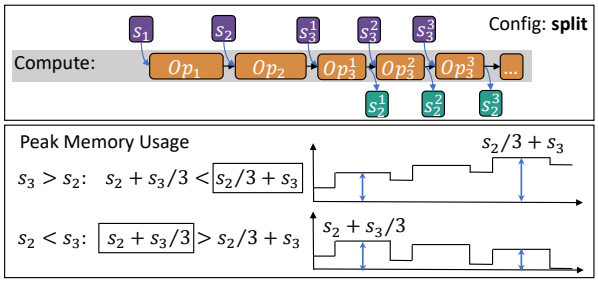


Fig. 8. Illustrations of cost estimation on the split strategy. The *split* option is used on Op_3 to make memory reduction by $\frac{2}{3} * \min\{size(s_2), size(s_3)\}$, where we release s_2 in three steps and consider the situation according to the size relationship between s_2 and s_3.

constrained deep learning training problem can be formulated as an optimization problem, where our goal is to find appropriate planning $\mathcal{C}$ specifying the memory management strategy configuration $c_i \in \{$*swap*, *recompute*, *split* $\}$ for tensor s_i to reduce the max training memory requirement under available GPU memory $\mathcal{M}$ while minimizing the incurred time cost on the performance.

$$\begin{aligned} &\min_{\mathcal{C}} \ T + \Delta T(\mathcal{C}) \\ &s.t. \quad M_i - \Delta M(\mathcal{C})_i \leq \mathcal{M}, \forall i \in \{1, ..., N\} \end{aligned} \tag{1}$$

Once the strategy of tensors is decided as $\mathcal{C}$, $\Delta M(\mathcal{C})_i$ and $\Delta T(\mathcal{C})$ are the memory reduction when executing Op_i and the extra execution time cost of the whole graph. We formulate the optimization problem by Equation 1, but finding the optimal parallelization strategy is NP-hard, by an easy reduction from the Minimum Makespan problem [39]. Our key insight is that a DNN training job is a predictable workload that declares its computation in terms of dataflow graph. So we could build a cost model to estimate the execution time with given memory management strategy configurations, and use a greedy search algorithm preferring to move towards lower cost guided by our analytic model.

B. Cost Models

By exploiting the predictability of DNNs' iterative execution, we derive the analytic model for the memory reduction $\Delta M_i[s_j, c]$ and extra time cost $\Delta T_i[s_j, c]$ at the current operation Op_i when applying the memory optimization strategy c on tensor s_j, where $c \in \{swap, recompute, split\}$. Assuming that encountering a memory bottleneck when executing Op_i, we will formulate the $\Delta M_i[s_j, c]$ and $\Delta T_i[s_j, c]$ for each candidate strategy on each possible sTensor as following.

Swap/Recompute. For the live tensors that are neither inputs nor outputs of the current operation, e.g., s_1 and s_2 in the Figure 7, *split* would not bring additional benefits in terms of memory reduction beyond *swap* or *recompute*. For each of them, we only need to determine the swap and recompute options based on its costs from extra memory transfer or computation. Equation 2 formulates the memory reduction of ΔM when applying *swap* or *recompute* on s_j. Figure 7 illustrates the cost introduced by non-split strategies. The memory reduction $\Delta M_3[s_1, swap]$ and $\Delta M_3[s_1, recompute]$

are both $size(s_1)$, which is achieved by swapping or recomputing tensor s_1 respectively.

$$\begin{aligned} \Delta M_i[s_j, c] &= size(s_j) \\ c \in \{\text{swap}, \text{recompute}\} &\wedge i < j \end{aligned} \tag{2}$$

As for the *swap* strategy, the cost comes from the idle time on the computation stream, since the inserted swap-out operation might block the execution of the following operation due to the limited GPU memory (e.g., there is no space for Op_3 to allocate its output tensor). The same situation may also happen at the execution of the swap-in operation. Equation 3 formulates the forward and backward time cost of setting *swap* for s_j, which aims to measure whether swap operation can be overlapped by computation and the gap between them. B represents the hardware bandwidth (e.g., PCIe) between CPU and GPU, and Oc_u represents the percentage of PCIe occupied during the execution of Op_u. As described above, i is the current memory bottleneck position and j is index of the target tensor. q is the index of Op_q which use s_i as inputs in the backward phase and p is the index where the swap-in operation begins to execute. Oc_u is key to calculate the potential overlap ratio and we keep an array to simulate and store the status of each Op as for implementation.

$$\begin{aligned} \Delta T_i[s_j, \text{swap}] = &\underbrace{\max\left\{\frac{size(s_j)}{B} - \sum_{u=j+1}^{i-1}(1 - Oc_u) * T_u, 0\right\}}_{\text{Forward cost}} \\ &+ \underbrace{\max\left\{\frac{size(s_j)}{B} - \sum_{u=p}^{q-1}(1 - Oc_u) * T_u, 0\right\}}_{\text{Backward cost}} \\ s.t. \quad & M_i > \mathcal{M} \quad (i > j+1) \\ & M_k + size(s_j) < \mathcal{M} \quad (\forall k \in [p, q], q >= p) \end{aligned} \tag{3}$$

Unlike the *swap*, the *recompute* strategy directly evicts $tensor_i$ but needs to execute corresponding sub-graph to generate s_i before required again (e.g., the gray version of Op_1). Equation 4 formulates the time cost of setting *recompute* for s_i, where I is the operations set for recomputing s_i. Because sometimes the input tensor of operation i is also set *recompute*, the input tensor's operation is also needed to inserted into I. For example, because s_2 is already set *recompute* option, $\mathcal{I}$ will be $\{Op_1, Op_2\}$ if *recompute* s_3.

$$\Delta T_i[s_j, \text{recompute}] = \sum_{u \in \mathcal{I}} T_u \tag{4}$$

Split. For the input tensors of the current operation, e.g., s_2 of Op_3 in Figure 7, by introducing the *split* option, we could reduce current peak memory usage by performing fine-grained micro tensor eviction. Note that the *split* enables fine-grained memory manangement, and it can be combined with *swap* and *recompute*. So for each split tensor, the option should be set as (c, p_num, dim), where c is the option between $\{swap, recompute, reside\}$, p_num is the split number and dim is the target dimension. Taking Figure 8 as an example, we split the input tensor s_2 into 3 micro-tensors ($p\_num = 3$), and evict each micro-tensor by the *swap* and *recompute* option once the the corresponding operation Op_3^i has finished. The memory saving ΔM_i is brought by the memory reuse between input tensor s_j and output tensor s_{j+1}, shown as Equation 5.

$$\begin{aligned} &\Delta M_i[s_j, (c, p\_num, dim)] = \\ &\qquad \frac{p\_num - 1}{p\_num} * \min\{size(s_j), size(s_{j+1})\} \\ &c \in \{\text{swap}, \text{recompute}\} \wedge i = j + 1 \end{aligned} \tag{5}$$

The time cost incurred by the *split* operation could be classified into three categories: (1) The costs of swap/recompute on the micro-tensors, which are similar with the above analysis. (2) The extra memory copy overheads incurred by the `split` operation and `merge` operation. (3) The performance degradation of the GPU kernels (e.g., the kernel launch time, the GPU under-utilization of micro-tensor operations). Equation 6 formulates the time cost of split memory option, where $\Delta T_{s_i, split}(p\_num, dim)$ represents the last two cost categories above and is related with the split number.

$$\begin{aligned} \Delta T_i[s_j, (c, p\_num, dim)] = &\underbrace{\sum_{u=1}^{p\_num} \Delta T_i[s_j^u, c]}_{\text{swap/recompute cost}} \\ &+ \underbrace{\Delta T_{s_j, split}(p\_num, dim)}_{\text{split overheads}} \\ c \in \{\text{swap}, \text{recompute}\} &\wedge i == j \end{aligned} \tag{6}$$

C. Model-guided Planning

After the cost model of each strategy on each tensor is defined, TSPLIT adopts a model-guided planning algorithm to search strategy combination for the trainability as well as high training throughput. Meanwhile, thanks to the predictability and iterative characteristic of deep learning training, TSPLIT profiles the training process of the given model before actual execution and uses these profiling data to calculate the cost of each candidate strategy.

To shrink the significantly expanded searching space from splitting, we divide the decision for split strategy and non-split strategy, which aims at live tensors in GPU and current input tensor. We make consistent memory options (e.g., swap/recompute or not) for the micro-tensors inside a sTensor and rule out the complex combinatorial decision among multiple sTensors. The following key observation also verifies our design: Given continuous tensor access, swapping out an earlier generated tensor should be prioritized. This is because the swap operation of such tensor can start the transfer from GPU to CPU earlier but is possibly used in backward pass later, which maintains a longer time memory reduction for GPU and a higher utilization rate for CPU-GPU bandwidth.

As shown in Algorithm 2, given an operation execution schedule $\mathcal{O}$ and a GPU with available memory $\mathcal{M}$, TSPLIT simulates the memory requirement M_i at each op_i (line 3). When encountering a memory bottleneck (line 5), TSPLIT iteratively selects strategy for the tensor with the smallest $\frac{\Delta T}{\Delta M}$

Algorithm 2: Profiling Based Planning Algorithm

Data: Operation schedule: $\mathcal{O}$, GPU available memory $\mathcal{M}$
Result: Strategy combination: $\mathcal{S} = \{s_i \mid i \in [1, N]\}$

1 $\mathcal{S} = \{\}$
2 **for** $op_i \in \mathcal{O}$ **do**
3 $M_i \leftarrow current\ memory\ requirement$
4 *// Suffer from memory bottleneck at* op_i
5 **while** $M_i - \Delta M(\mathcal{S})_i > \mathcal{M}$ **do**
6 *// Step 1: select best non-split strategy from previous*
7 $j, c \leftarrow \arg\min \frac{\Delta T_i[s_j,c]}{\Delta M_i[s_j,c]} := \{s_j, c | c_j \in \mathcal{C},$
8 c_j *is* reside,
9 $c \in \{\text{swap}, \text{recompute}\}\}$;
10 *// Step 2: select best split strategy from current*
11 $p\_num, c', dim \leftarrow \arg\min \frac{\Delta T_i[s_j,(c',p\_num,dim)]}{\Delta M_i[s_j,(c',p\_num,dim)]} :=$
12 $\{p\_num, dim, c' |$
13 $c' \in \{\text{swap}, \text{recompute}\}\}$;
14 *// Step 3: select better strategy from non-split and split*
15 **if** $\frac{\Delta T_i[s_j,c]}{\Delta M_i[s_j,c]} \leq \frac{\Delta T_i[s_j,(c',p\_num,dim)]}{\Delta M_i[s_j,(c',p\_num,dim)]}$ **then**
16 $\mathcal{C} \leftarrow c_j = \{c, 1, -1\}$;
17 **else**
18 $\mathcal{C} \leftarrow \mathcal{C} \cup c_i = \{c', p\_num, dim\}$;
19 *// No strategy for current input yet*
20 **if** $c_i \notin \mathcal{C}$ **then**
21 $\mathcal{C} \leftarrow \mathcal{C} \cup c_i = \{reside, 1, -1\}$;

to reduce the memory requirement in 3 steps (line 6-16). *Step 1*: As for the live tensors in GPU, we assign non-split strategies $\{swap, recompute\}$ on them to get their costs and propose the tensor and strategy with the smallest $\frac{\Delta T}{\Delta M}$ (line 6-8). *Step 2*: By introducing the *split* strategy, we can reuse memory between inputs and outputs. We iterate the strategy c', split number p_num, and split dimension dim on them to get their costs and propose the tensor and *split* strategy with the smallest $\frac{\Delta T}{\Delta M}$ (line 9-11). *Step 3*: We select better strategy proposed by non-split strategy in *step 1* and split strategy in *step 2* for our decision and insert it into $\mathcal{S}$ (line 12-16). If there exists no bottleneck here, we just set $reside$ for the current input tensor (line 17-19). The planning will terminate as soon as all the bottlenecks are eliminated or fail because of no more available tensors.

V. TSPLIT IMPLEMENTATION

TSPLIT is designed and implemented on a python-based DL framework, where the computation operations are accelerated by using NVIDIA cuBLAS and cuDNN [40]. Note that the idea of fine-grained abstraction, i.e., sTensor, and the design of TSPLIT are not limited to our DL platforms. In other words, our techniques can also be adopted in other platforms, such as TensorFlow. Our implementation consists of 14.5K LOC in C/C++/CUDA with a Python front-end (20.7K LOC).

A. sTensor Abstraction

TSPLIT takes a computation graph as input and redefines each tensor in the graph as a *splittable tensor* or *sTensor*, which enables tensor-split primitive.

sTensor configuration. Each sTensor contains not only the basic tensor information, but also the configuration information to enable the tensor splitting primitive of sTensor. For a certain sTensor t, its configuration `cfg` contains the memory option `opt` (i.e., reside, swap and recompute) and the splitting settings (i.e., split number `p_num` and dimension `dim`).

```
struct sTensor {
  //tensor information
  size_t tensor_id;
  Vector <Tensor*> inputs;

  //config information
  struct config{
      // memory option (reside/swap/recompute)
      size_t opt;
      size_t p_num;
      size_t dim;
  }cfg;

  void set_config(config cfg);
  //split-n-merge
  void split(size_t dim, size_t p_num);
  void merge(size_t dim);
};
```

Fig. 9. The sTensor interfaces.

sTensor interfaces. As shown in Figure 9, *sTensor* provides a set of interfaces and helps to manage the size of the joint search space of tensor split and memory management. sTensors utilize the `split` primitive to break the operation boundary of tensors, allowing single sTensor to be split as `p_num` fine-grained micro-tensors executed by the operation and memory operations (e.g., such as allocate/free and evict/regenerate). It can be split along the `dim` of sample, model parameter, or attribute (e.g., batch, channel, height, and width dimensions for images). The `merge` interface allows transforming multiple micro-tensors to a entire tensor in one of the two ways: either the concatenation along one `dim` or element-wise reduction (e.g., sum). During the training, the `merge` of micro-tensors can be required by the operation (e.g., batch normalization) or the need of tensor re-split (e.g., different `p_num` for inputs and outputs).

sTensor graph generation Based on the sTensor configuration, TSPLIT can easily transform a tensor-based dataflow graph into the corresponding augmented sTensor-based graph, which includes extra partition, memory optimization operations and additional control flow edges (e.g., timing).

Figure 10 gives an example to show the the augmented graph generation process. The `config` of each sTensor is shown in the left, and then TSPLIT utilizes `split(config.dim, config.p_num)` to split the corresponding operations, e.g., the splitting of s_1 also splits the operation o_2 into $\{o_1^1, o_1^2, o_1^3, o_1^4\}$. Further, $merge\&split$ operations are inserted for tensor re-split (e.g., different `p_num`). For example, an $merge\&split$ operation is inserted between o_2 and s_2 because o_2's input (s_1) and output (s_2) have different `config.p_num`. At last, memory-option operations are inserted for sTensors according to their `config.opt`.

B. Profiling-based Estimation

As assumed by [20, 41, 42], the execution time of each operation is predictable with low variance and is unrelated to data. TSPLIT profiles every single operation before training, which is helpful for models that can't fit in GPU.

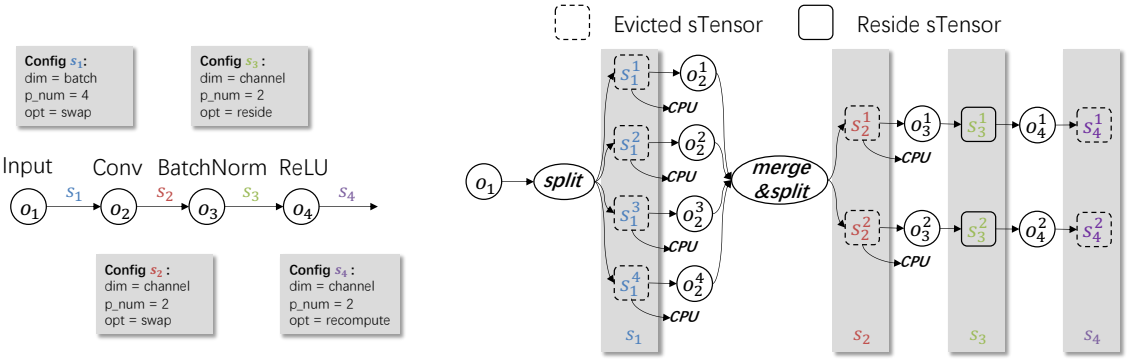


Fig. 10. An example of augmented dataflow graph $\mathcal{G}$ for sTensor configuration $\mathcal{S}$. The `config` of each sTensor is shown in the left, and then TSPLIT utilizes `split(config.dim, config.p_num)` to split the corresponding operations, e.g., the splitting of s_1 also splits the operation o_2 into $\{o_1^1, o_1^2, o_1^3, o_1^4\}$. Further, $merge\&split$ operations are inserted for tensor re-split (e.g., different `p_num`). For example, an $merge\&split$ operation is inserted between o_2 and s_2 because o_2's input (s_1) and output (s_2) have different `config.p_num`. At last, memory-option operations are inserted according to their `config.opt`.

TSPLIT utilizes `cudaEvent` to measure the execution time of computation operation T_u and obtain transfer time of *swap* as $\frac{size(t_i)}{bandwidth}$ by the full utilization of PCI-e bandwidth. Meanwhile, the profiling procedure should monopolize the hardware, e.g., GPU cores and PCIe bandwidth. Under the current strategy combination $\mathcal{S}$, TSPLIT predict the Ocu of PCIe when executing op_u through simulation. Specifically, TSPLIT first assigns the ideal swap-out begin time and swap-in begin time for each *swap* tensors as the generation time and the previous computation operation begin time and then simulate the PCI-e occupancy status.

By introducing *split* strategy, we enlarge the tensor search space by inserting the current input tensor into the candidate list. The cost model of *split* strategy can be divided into the cost of *swap* or *recompute* and $\Delta T_{i,split}(p\_num, dim)$, which is the cost of tensor *split* on hardware . $\Delta T_{i,split}(p\_num, dim)$ is consist of the cost of split kernel and the cost of $split\&merge$ operation. TSPLIT splits the original tensor into p_num micro-tensors along dimension dim and then sums total computation execution time on these micro-tensors to calculate the extra execution time as the cost of splitting kernel. The cost of $split\&merge$ is ignored in TSPLIT, which accounts for less than 1% of total execution time.

C. Data Layout Management

While swapping full tensors allow to maintain the same data layout, splitting will have consequences on the layout of tensors, contiguousness, etc, which in turn could affect performance. TSPLIT avoids the negative effects in the following ways: First, during the planning phase, we will use profiling data to calculate the cost of splitting in Equation 6, and still resort to swapping/recomputing full tensors if the cost outweighs the benefits (see line 15 in Algo 2). Second, to enforce contiguousness, we use best-fit memory allocation strategy in the implementation to store micro-tensors in contiguous chunks of memory. Also, if we find the split and merge actions are not necessary to be executed physically, we will perform a in-place split or merge instead of extra memory copy to alleviates the consequences on data layout. For example, p_num changes from 2 to 4 on batch dimension will share the same tensor with different pointer address.

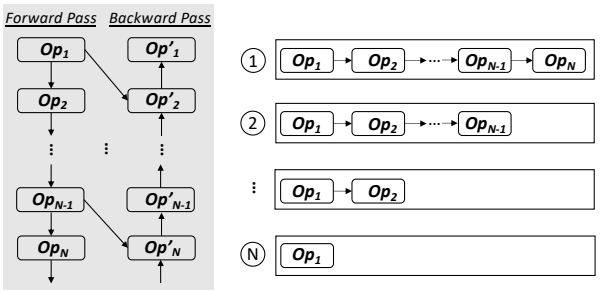


Fig. 11. The memory-centric recomputation strategy. Left: The continuous recomputation sequence is $\{op_1 \rightarrow op_2 \rightarrow \ldots \rightarrow op_N\}$ and the input tensor of op_1 is stored as the checkpoint. Right: N rounds of recomputation sequence to get the output of the corresponding op_i respectively.

D. Deep Learning Runtime

Graph Executor. All GPU operations will be scheduled into the GPU compute stream, and be launched and executed asynchronously to avoid block the CPU threads. *Swap* operations are scheduled in another two streams, including the D2H stream (GPU to CPU) and the H2D stream (CPU to GPU). The synchronization between *swap* and their corresponding computation operations are implemented by inserting CUDA *event*s. Based on these, TSPLIT could perform asynchronous memory copy, i.e., `cudaMemcpyAsync()`, and guarantee the specific execution order. TSPLIT provides fine-grained tensor memory scheduling which also involves frequent allocation and de-allocation. However, such intensive memory allocations incur unnegligible overheads if using the native `cudaMalloc` and `cudaFree`. To alleviate this issue, we pre-allocate a large piece of GPU memory and implement a runtime memory pool to manage the GPU usage for TSPLIT.

Recomputation Implementation. For multiple continuous operations which require to be recomputed, there are two optional strategies for the recomputation [17]. The speed-centric strategy directly recomputes all N ancestor sTensors in one-pass and stores all intermediate results, which involves

TABLE IV
THE LARGEST **SAMPLE SCALE** (BATCH SIZE) THAT EACH POLICY CAN REACH WITH A 24GB TITAN RTX

Models	Base	vDNN conv	vDNN all	Check points	Super Neurons	**TSPLIT**
VGG-16	67	176	413	316	462	**661**
VGG-19	61	164	411	293	441	**661**
ResNet-50	76	201	783	192	591	**1278**
ResNet-101	51	138	783	124	361	**1096**
InceptionV4	36	131	864	220	882	**1372**
Transformer	62	x	452	117	x	**730**

TABLE V
THE LARGEST **PARAMETER SCALE** THAT EACH POLICY CAN REACH WITH A 24GB TITAN RTX. THE BATCH SIZE OF EACH MODEL IS FIXED AT 16 AND WE SCALE CHANNEL NUMBER IN CNNS AND HIDDEN SIZE IN TRANSFORMER RESPECTIVELY.

Models	Base	vDNN conv	vDNN all	Check points	Super Neurons	**TSPLIT**
VGG-16	4	10	25	19	28	**40**
VGG-19	3	10	25	18	27	**40**
ResNet-50	4	12	48	12	36	**79**
ResNet-101	3	8	48	7	22	**68**
InceptionV4	2	8	54	13	55	**23**
Transformer	3	x	18	5	x	**30**

$\mathcal{O}(N)$ computation costs and $\mathcal{O}(N)$ additional memory consumption. To fully exploits the memory-saving opportunity, we adopt the memory-centric strategy that recomputes forward dependencies every time for each backward layers, which involves $\mathcal{O}(N^2)$ computation costs and $\mathcal{O}(1)$ additional memory consumption. For example, as illustrated in Figure 11, the recompute sequence is $\{op_1 \rightarrow op_2 \rightarrow \ldots \rightarrow op_N\}$ and the input tensor of op1 is stored as the checkpoint. In the recomputation phase, $\{op_1 \rightarrow op_2 \rightarrow \ldots \rightarrow op_N\}$, $\{op_1 \rightarrow op_2 \rightarrow \ldots \rightarrow op_{N-1}\}$, $\ldots$, $\{op_1 \rightarrow op_2\}$ and $\{op_1\}$ will be executed, which totally involves $N \times (N+1)/2$ extra operations. The detailed explanation can be found in SuperNeurons [17]. We further adopt an LRU-based recomputation optimization to combine the advantages of both strategies with limited memory, i.e., execute as the speed-centric manner and abandon the least recently used intermediate tensor once the available memory is not enough.

VI. EVALUATION

In this section, we conduct detailed evaluations to demonstrate the effectiveness of TSPLIT. We compare TSPLIT with other state-of-the-art baselines on large DNN model training.

A. Experimental Setup

Machine Environment. Our experiments are conducted on two different hardware environments. The first server is equipped with NVIDIA Titan RTX GPU 24 GB, Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz, 256 GB RAM, PCIe 3.0, running Ubuntu 16.04. The second server is equipped with NVIDIA GTX 1080ti GPU with 11 GB of RAM, Intel(R) Xeon(R) E5-2650 v4 CPU @ 2.20GHz, 128 GB RAM and PCIe 3.0. In both servers, the CUDA Toolkit version is 10.0, and the cuDNN is 7.5.0. Compared to 1080ti, Titan RTX GPU has larger GPU memory capacity and more computing units.

Baselines. We evaluate and compare TSPLIT with other state-of-the-art memory-optimized polices, including vDNN [19] (swap), Checkpoints [36] (recompute) and SuperNeurons [17] (swap & recompute). To show the effects of memory optimizations, we create a baseline called Base that represents common DL systems (e.g., TensorFlow, PyTorch) which store all the feature maps and parameters during the training process. vDNN [19] virtualizes the memory usage by swapping fixed feature maps to the CPU, where vDNN-conv only swaps inputs of convolution and vDNN-all swaps all tensors. Checkpoints [36] optimizes the memory with the cost of extra forward computation. Superneurons [17] combines *swap* and *recompute* to optimize memory, which swaps the outputs of convolution to CPU memory and recomputes the outputs of other cheap-to-recompute operations such as pooling. We also implement TSplit on PyTorch and compare it with recently proposed methods, including Zero-Offload [43] and FairScale Offload [44]. Zero-Offload offloads the parameter gradients to CPU at the backward phase, conduct the optimizer updates computation in CPU and then swap the updated parameters to GPU. The CPU in Zero-Offload is responsible for updating the parameters and holding onto the optimizer state. Fairscale-Offload also involves CPU with optimizer updating and shards models almost equally based on the number of parameters, which is moved between CPU and GPU at each iteration. Moreover, it copies intermediate activations between CPU and GPU in training. We have not included SwapAdvisor [33] because it is not open-sourced and it mainly focuses on scheduling optimization which is orthogonal to and compatible with our work.

Benchmarks and Datasets. We evaluate TSPLIT on representative models of different architectures such as CNN and Transformer. The CNN models contain VGG, ResNet and InceptionV4, which are the classic CNN architectures and widely used for DL system benchmarks [45]. We also evaluate Transformer [46] which is the basic module of the current state-of-the-art large NLP models (e.g., BERT, GPT-3). We take the ImageNet [1] dataset for CNN models and the IWSLT2016 [32] dataset for Transformer

B. Results Analysis

This section answers the following research questions:

1) How much model scale can TSPLIT obtain comparing with the other baselines?
2) How much throughput can TSPLIT maintain while performing out-of-GPU training?
3) How much performance gain comes from TSPLIT's tensor split mechanism?
4) How does the hardware and workload affect the scheduling decision of TSPLIT?

Model Scale We demonstrate the memory footprint of different models by scaling models along sample dimensions and parameter dimensions. As for *Sample Scale*, in order to scale models, we fix the parameter size and increase the

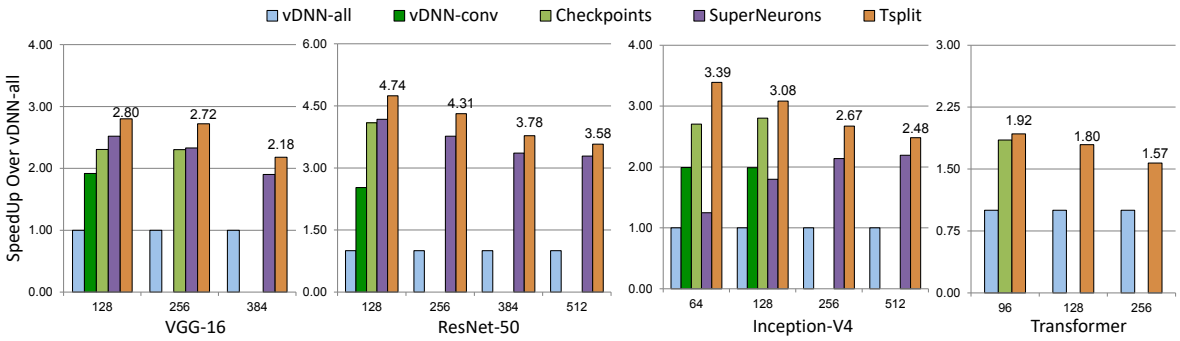


Fig. 12. Performance comparison on TITAN RTX GPU under different sample size, the x-axis represents the sample-size. Experiments are conduct on four famous models, including CNNs and Tansformer. With the increased sample size, the execution of previous approaches begins to fail, i.e., the related performance bars are missing in the figure. TSPLIT significantly outperforms these designs and supports much larger sample size with minor throughput loss.

sample size in a batch, e.g., number of images in CNNs and number of language sequences in Transformer. Table IV shows the largest sample scale by each approach on a TITAN RTX.

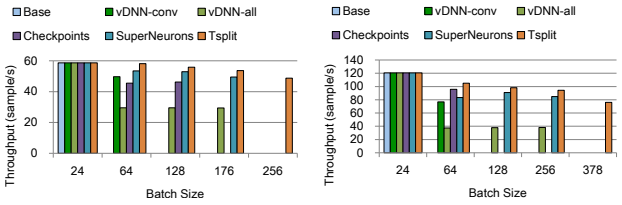
In summary, TSPLIT achieves the maximum sample scale among all these six models. Compared with Base, TSPLIT increases the maximum sample scale by $18.15\times$ on average. Due to the complexity of multi-branch model architecture, TSPLIT could obtain up to $21.50\times$ and $38.11\times$ for ResNet-101 and InceptionV4, respectively. Among the previous state-of-the-art approaches, vDNN-all presents the best in ResNet-50 and ResNet-101 while SuperNeurons presents the best in other models. Compared to the best combination of previous designs, TSPLIT still increases the max sample scale up to $1.52\times$ on average. As for *Parameter Scale*, in order to scale models, we fix the batch size at 16 and increase the parameter dimensions, e.g., channels of convolution kernels in CNNs and hidden size in Transformer. We expand the number of channel in different kernels proportionally. Specifically, if the original channel size is c_1 and the parameter scale number is k, it has $c_1 \times k$ channels after scaling. TSPLIT also achieves the maximum parameter scale among all these six models. Note that vDNN-conv and SuperNeurons cannot help Transformer since it doesn't have convolution layers. Specifically, vDNN-conv has no layers to offload and SuperNeurons has no layers as checkpoints for recomputation. For both *Sample Scale* and *Parameter Scale*, TSPLIT achieve the best results thanks to *split* strategy.

Based on Table IV and Table V, we conduct the in-depth analysis. The max sample scale on VGG-16 and VGG-19 under vDNN-all are almost equal, because the bottleneck of VGG under vDNN-all is caused by the largest layer, i.e., the second block, which exists in both VGG-16 and VGG-19. In SuperNeurons, it performs swap-in operations and recompute operations in the backward at the same time to improve the throughput, while both of them require extra memory allocation and causes the memory bottleneck. The key problem here is the tensor-wise memory management, which causes high memory usage during training and leads to low hardware utilization.

Throughput. We also evaluate the efficiency of TSPLIT by measuring the running throughput against other approaches. Figure 12 presents the relationship between the sample size (x-axis) and the speedup over vDNN (y-axis) among 4 popular models, including CNNs and Transformer. We highlight that among all four workloads, TSPLIT achieve throughput improved up to $4.7\times$ and $2.7\times$ under the same memory oversubscription, respectively.

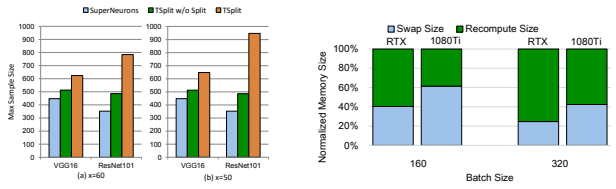
The speedup over vDNN is because vDNN-all swaps all layers instead of on demand which seriously exacerbates the overhead and leads the most serious performance loss. Meanwhile, vDNN-all swaps fixed layers without considering the actual memory requirement, so the throughput (samples/second) almost remains the same. Taking Figure 12 (VGG-16) as an example, it shows that training VGG-16 with sample size 128, the throughput of TSPLIT is $1.46\times$ of vDNN-conv, $2.80\times$ of vDNN-all, $1.21\times$ of Checkpoints and $1.11\times$ of SuperNeurons. Although vDNN-conv tries to overlap the computation and communication by swapping the input of convolution layers, the bottleneck of slow PCIe bandwidth compared with high computational ability of GPU, and layer-wise synchronization overhead leads to performance loss. Checkpoints only involves recomputation without synchronization overheads and always presents better computation throughput than vDNN. But it also shows weaker scalability of sample size, similar to vDNN-conv. By combining swap and recompute, SuperNeurons outperform other previous baselines. However, SuperNeurons still performs worse than our TSPLIT since the tensor granularity based design can not fully utilize GPU resources. We clarify that TSplit offers the following improvement over SuperNeurons: (1) SuperNeurons only swaps data for convolution operations and cannot support Transformer model as TSplit. (2) TSplit outperforms significantly over SuperNeurons for large CNN models (e.g., Inception-V4). (3) Besides the throughput, we show that TSplit could promote both the maximum batch size and parameter size by up to $3\times$, as compared to SuperNeurons.

When sample size increases to 384, TSPLIT outperforms vDNN-all and SuperNeurons by $2.18\times$ and $1.15\times$, respectively. vDNN-conv and Checkpoints fail to run because the evicted tensors' space is not enough to eliminate the mem-



(a) VGG-16 (b) ResNet-50

Fig. 13. Performance comparison on 1080Ti GPU, whose FP32 FLOPS is about 70% of TITAN RTX.



(a) x% of Base throughput (b) Different Devices

Fig. 14. Figure 14(a) Max sample size under x% of Base throughput. TSPLIT can train larger model scale with the cost of throughput degradation. Figure 14(b) The strategy combination of VGG-16 on RTX and 1080Ti. Because of the profiling-based cost estimation, TSPLIT apply different strategy for the same model under different hardware.

ory bottleneck. The throughput of SuperNeurons decreases because more tensors should be evicted, where extra recomputation and the idle time is introduced as communication volume increases. Due to the fine-grained memory management, TSPLIT still achieves better throughput performance. TSPLIT gets closer to vDNN performance with the increasing sample size because it would make planning get closer to split and swap full layers, which can be reached when even a single layer might exceed the GPU memory due to the very large sample/parameter size.

C. Breakdown Analysis

Effect of Tensor-Split To evaluate the impact of the tensor split mechanism, we compare TSPLIT with TSPLIT w/o Split. We set the expected throughput as 60% and 50% of the baseline throughput and then compare the max trainable sample size. Experiments are conducted on VGG-16 and ResNet-101. In Figure 14(a), under 60% of the basic throughput, the max trainable sample size of TSPLIT w/o Split and TSPLIT outperforms SuperNeurons on ResNet-101 by 38% and 123%, respectively. When decreasing x to 50, TSPLIT further outperforms SuperNeurons by 169%, shown as Figure 14(a). Similar experimental results are shown on VGG-16. Compared with SuperNeurons which makes decisions based on the static information (e.g. layer type), TSPLIT w/o Split searches for better swap/recompute policies based on cost models, which reduces unnecessary memory eviction, improves the overlap between computation and communication, and alleviates redundant computation introduced by memory-centric recomputation.

Performance Comparison on 1080ti. We further evaluate TSPLIT on 1080Ti (11 GB) which has less GPU memory than RTX (24 GB). Meanwhile, the FP32 computation performance of 1080Ti (11.34 TFLOPS) is around 70% of RTX (i.e., 16.3 TFLOPS). We report the actual runtime speed (images/second) as the metric and show the throughput comparsion in Figure 13. TSPLIT still achieves the best among all previous approaches. Compared to RTX, 1080ti has lower computation ability, which increases the operation computation time and therefore improves the overlap between computation and communication. With the increased sample size, the performance loss of vDNN in 1080ti is less than in RTX.

TABLE VI
THE LARGEST **SAMPLE SCALE** (BATCH SIZE) THAT EACH POLICY CAN REACH WITH A 24GB TITAN RTX

Models	Base	Zero-Offload	FairScale	TSPLIT(PyTorch)
VGG-16	48	176	383	**485**
ResNet-50	32	162	502	**920**
Transformer	34	130	440	**540**

TABLE VII
THE LARGEST **PARAMETER SCALE** THAT EACH POLICY CAN REACH WITH A 24GB TITAN RTX. THE BATCH SIZE OF EACH MODEL IS FIXED AT 16 AND WE SCALE CHANNEL NUMBER IN CNNS AND HIDDEN SIZE IN TRANSFORMER RESPECTIVELY.

Models	Base	Zero-Offload	FairScale	TSPLIT(PyTorch)
VGG-16	3	14	26	**32**
ResNet-50	2	12	23	**55**
Transformer	2	18	15	**24**

Configuration Comparison on GTX 1080ti. For the same DNN model, the strategy combination selected by TSPLIT could be different when the underlying platform changes. Different hardware could lead to distinct decisions and should be taken into consideration and TSPLIT utilizes the profiling data for decisions. Figure 14(b) shows the total memory size of swapped tensors and recomputes tensors decided by TSPLIT in different GPUs. The results indicate that TSPLIT chooses more tensors to swap, rather than recompute on GTX 1080ti due to the larger recomputation overheads. This verifies that TSPLIT can capture the differences in varied GPU characteristics.

D. Applicability for existing DNN Frameworks

Given a dataflow graph, TSplit performs a model-guided search based on the graph, and outputs an augmented dataflow graph which includes extra split/swap/regenerate operators and additional control flow edges (as illustrated in Figure 10). The additional edges ensure the final execution order adheres to the timing of TSplit's searched plan. Although TSPLIT provides a lightweight runtime to execute the augmented dataflow graph, TSPLIT will not result in changes in existing operator implementation, making it compatible with existing DNN frameworks: we could convert TSplit augmented dataflow graph into PyTorch or TensorFlow model that can be executed in existing DNN frameworks. After adding the extra split/swap/regenerate operators in existing frameworks, the augmented dataflow graph of TSPLIT can be converted into the executable model in PyTorch or TensorFlow [47].

We have implemented the TSplit to PyTorch conversion and compare it with Zero-Offload [43] and FairScale-Offload [44] on PyTorch. Table VI and Table VII show that TSPLIT TSPLIT achieves maximum model scale up to 4.6× and 2.4×

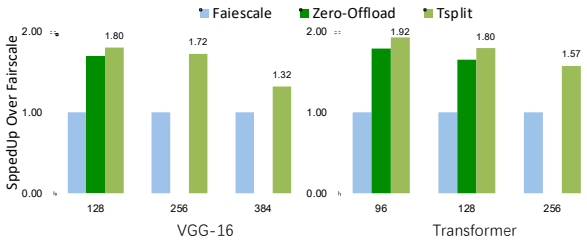


Fig. 15. Performance comparison with Fairscale and Zero-offload on PyTorch in terms of throughput.

and throughput improved up to 1.9× and 1.2× (Figure 15), under the same memory over-subscription, respectively. Zero-Offload [43] only offloads the gradients of parameters and optimizer state to CPU. For CNN-based models with small-scale parameters but large-scale hidden states, we see that Zero-offload achieves almost the least sample scale. FairScale Offload [44] only utilizes the swapping techniques, where the limited PCIe bandwidth would slow down the training.

VII. Related Work

GPU Memory Management. Many memory management optimizations have been proposed for GPU, inluding paging [48], replacement caching [49], unified memory address [50] and memory pool [51]. Mosaic [48] provides application-transparent support for multiple page sizes to page in and out. MultiQx-GPU [49] designs the cost-driven replacement policy for data swapping. Pichai et al. [50]. Zhang et al. [51] propose *CNMeM* to exploit the variable's lifetime and size information to optimize memory allocation position. However, studies about paging, replacement caching and unified memory address are not designed for DL training and don't utilize the special nature of tensor access patterns, while others about memory pool don't consider CPU memory.

Out-of-Core Training. Out-of-core training utilizes extra memory (e.g., CPU) or extra computation (e.g., re-forward-propagation) to free tensors out from GPU. vDNN [19] first virtualizes the memory usage of DNNs against both GPU and CPU and employs a layer-wise memory management strategy. Layup [34] and SuperNeurons [17] further change several cheap-to-compute operations from swapping to re-computation [36] to improve the overall throughput. SwapAdvisor [33] adopts a genetic algorithm to simultaneously search for operator scheduling, memory allocation and swap decisions. Capuchin [18] further explores the dynamic tensor access patterns during the training. KARMA [35] formulates the policy decision problem as a two-stage Integer Linear Programming problem and first combine it with model parallelism to support distributed training. Zero-offload [43] offloads both data and compute to CPU, and work together with model parallelism. Note that, our approach is orthogonal to the model parallelism studies [20] and TSplit can also be adopted in multi-GPU to further increase the scalability. Overall, existing approaches are concentrating on tensor-wise memory management, which limits the swapping policy and leads to low hardware utilization and efficiency.

Fine-Grained Scheduling. Lookup Tables [52] co-locates the related individual tuples in fine granularity and designs a large *lookup table* as database. Squall [53] utilizes the presence of transactions, data and high throughput client workloads to re-partition the databases. Split-CNN [54] proposes to split the local receptive fields of CNN and reduce the GPU memory requirements, which hurts the model quality. moDNN [37] splits the batch of samples into several mini-batches and uses the accumulated gradients to compute the final updates. The split dimension keeps the batch dimension and the split number keeps constant for all layers, which may incur the serious efficiency problem of low computational cost layers. The small sub-batch size reduces the memory usage for all tensors in the computation graph by the same ratio but the memory management is still trivial.

Micro-batches and Model Parallelism. Existing DNN frameworks adopt to avoid gigantic tensors. For example, they can adopt pipeline parallelism (i.e., micro-batches) to partition gigantic tensors in the sample dimension (e.g., GPipe [55], PipeDream [56]). In addition, they can adopt model parallelism to partition the tensors in the parameter dimension (e.g., Megatron-LM [57], Mesh-TensorFlow [58] and GSPMD [59]). However, pipeline/model parallelism and TSplit address the memory challenges of DNNs in two orthogonal directions and settings: Their focus, however, is on strategies for parallelizing the actual execution across multiple GPUs, which jointly optimize tensor partition and device placement. By contrast, our focus is on the out-of-memory strategies (via offload and recompute), which jointly optimizes tensor partition and swap/recompute operations. Compared to DNN parallelization over multi-GPUs, TSplit has different challenges and search space, and is more attractive for users who cannot access more than a single GPU, or users who want to minimize resource usage.

VIII. Conclusion

Existing DNNs memory system suffers from unnecessary overheads due to the minimum granularity of management today is the entire tensor. TSplit addresses this issue with a holistic optimization solution that (1) provides a sTensor abstraction that exposes the system fine-grained memory operations capability, (2) leverages the predictability of DNN computation to build the cost model for each strategy, and (3) proposes a model-guided planning algorithm to explore the enriched search space for joint optimization of tensor split and swap/recompute strategies. Our evaluations show that TSplit can achieve significant improvements compared to existing tensor-based memory management baselines. This positions TSplit as a new enhancement to the existing DNNs memory management infrastructure.

IX. Acknowledgement

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004403), the National Natural Science Foundation of China (No. 61832001), PKU-Tencent joint research Lab. Zhi Yang and Bin Cui are the corresponding authors.

REFERENCES

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009, pp. 248–255.

[2] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *CVPR*, 2017, pp. 1492–1500.

[3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[4] X. Miao, N. M. Gürel, W. Zhang, Z. Han, B. Li, W. Min, S. X. Rao, H. Ren, Y. Shan, Y. Shao, Y. Wang, F. Wu, H. Xue, Y. Yang, Z. Zhang, Y. Zhao, S. Zhang, Y. Wang, B. Cui, and C. Zhang, "Degnn: Improving graph neural networks with graph decomposition," in *KDD*, 2021, pp. 1223–1233.

[5] X. Bai, Y. Pang, and G. Zhang, "Special focus on deep learning for computer vision," *Sci. China Inf. Sci.*, vol. 63, no. 2, p. 120100, 2020.

[6] S. L. Smith, P. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," in *ICLR*, 2018.

[7] M. A. Gordon, K. Duh, and J. Kaplan, "Data and parameter scaling laws for neural machine translation," in *EMNLP*, 2021, pp. 5915–5922.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[10] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019, pp. 4171–4186.

[11] X. Nie, S. Cao, X. Miao, L. Ma, J. Xue, Y. Miao, Z. Yang, Z. Yang, and B. Cui, "Dense-to-sparse gate for mixture-of-experts," *CoRR*, vol. abs/2112.14397, 2021.

[12] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," in *ICLR*, 2021.

[13] A. Gholami, Z. Yao, S. Kim, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *RiseLab Medium Post*, 2021.

[14] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv*, 2019.

[15] L. Gao, Y. Zhang, J. Han, and J. Callan, "Scaling deep contrastive learning batch size under memory limited setup," *arXiv*, 2021.

[16] X. Xie, F. Sun, Z. Liu, S. Wu, J. Gao, B. Ding, and B. Cui, "Contrastive learning for sequential recommendation," *arXiv*, 2020.

[17] 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 *PPoPP*, 2018, pp. 41–53.

[18] 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 *ASPLOS*, 2020, pp. 891–905.

[19] 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 *MICRO*, 2016, pp. 1–13.

[20] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," in *MLSys*, 2019.

[21] X. Miao, H. Zhang, Y. Shi, X. Nie, Z. Yang, Y. Tao, and B. Cui, "HET: scaling out huge embedding model training via cache-enabled distributed framework," *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 312–320, 2021.

[22] X. Miao, X. Nie, Y. Shao, Z. Yang, J. Jiang, L. Ma, and B. Cui, "Heterogeneity-aware distributed machine learning training via partial reduce," in *SIGMOD*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds., 2021, pp. 2262–2270.

[23] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[24] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *NeurIPS*, vol. 28, pp. 1135–1143, 2015.

[25] F. Fu, Y. Hu, Y. He, J. Jiang, Y. Shao, C. Zhang, and B. Cui, "Don't waste your bits! squeeze activations and gradients for deep neural networks via tinyscript," in *ICML*, vol. 119, 2020, pp. 3304–3314.

[26] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *HPCA*, 2018, pp. 78–91.

[27] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya, "Tflms: Large model support in tensorflow by graph rewriting," *arXiv*, 2018.

[28] 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 *OSDI*, 2016, pp. 265–283.

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019, pp. 8024–8035.

[30] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv*, 2015.

[31] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv*, 2020.

[32] M. Cettolo, N. Jan, S. Sebastian, L. Bentivogli, R. Cattoni, and M. Federico, "The iwslt 2016 evaluation campaign," in *ISWLT*, 2016.

[33] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *ASPLOS*, 2020, pp. 1341–1355.

[34] W. Jiang, Y. Ma, B. Liu, H. Liu, B. B. Zhou, J. Zhu, S. Wu, and H. Jin, "Layup: Layer-adaptive and multi-type intermediate-oriented memory optimization for gpu-based cnns," *TACO*, vol. 16, no. 4, pp. 1–23, 2019.

[35] M. Wahib, H. Zhang, T. T. Nguyen, A. Drozd, J. Domke, L. Zhang, R. Takano, and S. Matsuoka, "Scaling distributed deep learning workloads beyond the memory capacity with karma," in *SC*, 2020.

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

[37] X. Chen, D. Z. Chen, and X. S. Hu, "modnn: Memory optimal DNN training on gpus," in *DATE*, 2018, pp. 13–18.

[38] 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 *OSDI 20*, 2020, pp. 533–548.

[39] S. Lam and R. Sethi, "Worst case analysis of two scheduling algorithms," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 518–536, 1977.

[40] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014.

[41] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in accelerating convolutional neural networks," in *ICML*, 2018, pp. 2274–2283.

[42] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with rtasks," in *OSDI 20*, 2020, pp. 881–897.

[43] 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 ATC*, 2021, pp. 551–564.

[44] M. Baines, S. Bhosale, V. Caggiano, N. Goyal, S. Goyal, M. Ott, B. Lefaudeux, V. Liptchinsky, M. Rabbat, S. Sheiffer, A. Sridhar, and M. Xu, "Fairscale: A general purpose modular pytorch library for high performance and large scale training," https://github.com/facebookresearch/fairscale, 2021.

[45] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters," in *OSDI*, 2020, pp. 463–479.

[46] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, 2017, pp. 5998–6008.

[47] H. Dai, X. Peng, X. Shi, L. He, Q. Xiong, and H. Jin, "Reveal training performance mystery between tensorflow and pytorch in the single gpu environment," *Sci. China Inf. Sci.*, vol. 65, no. 1, pp. 1–17, 2022.

[48] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: a GPU memory manager with application-transparent support for multiple page sizes," in *MICRO*, 2017, pp. 136–150.

[49] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang, "Concurrent analytical query processing with gpus," *Proc. VLDB Endow.*, vol. 7, no. 11, pp. 1011–1022, 2014.

[50] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: designing memory management units for cpu/gpus with unified address spaces," in *ASPLOS*, 2014, pp. 743–758.

[51] J. Zhang, S. Yeung, Y. Shu, B. He, and W. Wang, "Efficient memory management for gpu-based deep learning systems," *CoRR*, vol. abs/1903.06631, 2019.

[52] A. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden, "Lookup tables: Fine-grained partitioning for distributed databases," in *ICDE*, 2012, pp. 102–113.

[53] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. E. Abbadi, "Squall: Fine-grained live reconfiguration for partitioned main memory databases," in *SIGMOD*, 2015, pp. 299–313.

[54] T. Jin and S. Hong, "Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization," in *ASPLOS*, 2019, pp. 835–847.

[55] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *NeurIPS*, vol. 32, 2019.

[56] 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 *SOSP*, 2019, pp. 1–15.

[57] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *SC*, 2021, pp. 1–15.

[58] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," *NeurIPS*, vol. 31, 2018.

[59] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni *et al.*, "Gspmd: General and scalable parallelization for ml computation graphs," *arXiv*, 2021.