vPIPE: A Virtualized Acceleration System for Achieving Efficient and Scalable Pipeline Parallel DNN Training

Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, Ping Luo, and Heming Cui, *Member, IEEE*

Abstract—The increasing computational complexity of DNNs achieved unprecedented successes in various areas such as machine vision and natural language processing (NLP), e.g., the recent advanced Transformer has billions of parameters. However, as large-scale DNNs significantly exceed GPU's physical memory limit, they cannot be trained by conventional methods such as data parallelism. Pipeline parallelism that partitions a large DNN into small subnets and trains them on different GPUs is a plausible solution. Unfortunately, the layer partitioning and memory management in existing pipeline parallel systems are fixed during training, making them easily impeded by out-of-memory errors and the GPU under-utilization. These drawbacks amplify when performing neural architecture search (NAS) such as the evolved Transformer, where different network architectures of Transformer needed to be trained repeatedly. vPIPE is the first system that transparently provides dynamic layer partitioning and memory management for pipeline parallelism. vPIPE has two unique contributions, including (1) an online algorithm for searching a near-optimal layer partitioning and memory management plan, and (2) a live layer migration protocol for re-balancing the layer distribution across a training pipeline. vPIPE improved the training throughput of two notable baselines (Pipedream and GPipe) by 61.4-463.4 percent and 24.8-291.3 percent on various large DNNs and training settings.

Index Terms—Machine learning, distributed systems, distributed artificial intelligence, pipeline, parallel systems, memory management

1 INTRODUCTION

In recent years, large deep neural networks (DNNs), including Transformer [52], BERT [10], AmoebaNet [39], and GNMT [58], are getting explosively deeper (i.e., more layers) and wider (i.e,. more parameters per layer) for higher modeling capacities. For instance, Transformer [52] has more than 600 layers (i.e., execution operators) and 6 billion parameters. This rising complexity of DNN models has also expedited the emergence of neural architecture search (NAS) (e.g., evolved Transformer [45]), where the layers of a model are dynamically activated/deactivated during training [39], [45] to search for a DNN architecture with high accuracy. This increasing complexity and dynamicity make it even more difficult for training a large DNN, considering that each GPU has only up to tens of gigabytes memory [18].

- *Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Ping Luo, and Heming Cui are with the Department of Computer Computer Science, The University of Hong Kong, Hong Kong 999077, China. E-mail: {sxzhao, fxli, xschen, xxguan, jyjiang, dhuang, yhqing, pluo, heming}@cs.hk.hk.*
- *Sen Wang, Peng Wang, and Gong Zhang are with Theory Lab, 2012 Labs, Huawei Technoloies, Co. Ltd, Shenzhen 518129, China. E-mail: {wangsen31, wang.peng6, nicholas.zhang}@huawei.com.*
- *Cheng Li is with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230052, China. E-mail: chengli7@ustc.edu.cn.*

Manuscript received 12 Nov. 2020; revised 21 Mar. 2021; accepted 24 Mar. 2021. Date of publication 2 July 2021; date of current version 5 Aug. 2021. (Corresponding author: Heming Cui.) Recommended for acceptance by J. Zola. Digital Object Identifier no. 10.1109/TPDS.2021.3094364

Pipeline parallelism is a promising approach to train large DNNs with lots of layers on multiple GPUs, where the DNN is partitioned into multiple stages, each containing a number of layers and running on a GPU. Existing pipeline parallel systems [14], [19], [33], [59] adopt a static partition policy, where the stage partition is fixed throughout the entire training process. A typical DNN training iteration contains a forward pass and a backward pass through all stages. The major memory consumption on each GPU (or stage) is for storing activations produced in a forward pass and reused in a backward pass [18], [37].

For high hardware efficiency (i.e., high GPU ALU utilization), a pipeline parallel system injects multiple batches of inputs and overlaps their forward and backward pass executions, forming a pipeline. Compared with a data parallel system [28], which needs to transfer enormous parameter updates among GPUs, a pipeline parallel system only needs to transfer intermediate data between layers across stages, significantly reducing the network consumption [33]. Therefore, more complex DNNs [19], [39], [45] are trained with pipeline parallel systems [14], [19], [33], [59].

An efficient pipeline parallel system should achieve two crucial design goals. First, as the system injects multiple input batches, it should carefully manage all stages' training memory to avoid exceeding the physical memory capacity on any GPU (**G1**). Otherwise, it will either cause out-of-memory errors or trigger synchronous paging events that significantly block the training execution of a DNN (discussed in Section 7). Second, to maximize the efficiency (i.e., high GPU ALU utilization and no stage stalls), the system

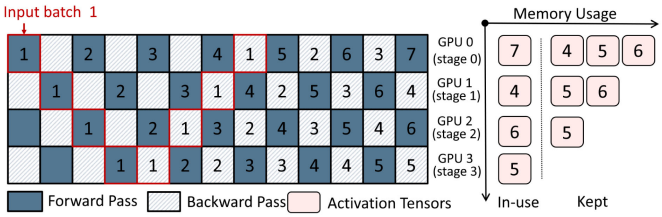

Fig. 1. A four-stage pipeline (Pipedream [33]). Stage 0 keeps four copies of activations, while stage 3 keeps only one copy.

should enforce a "balanced" partition (**G2**) such that all stages achieve roughly the same high throughput [19], [33]: data items processed per second by the pipeline. Unfortunately, despite much effort [14], [19], [33], [59] on building pipeline parallel systems, simultaneously realizing these two design goals for complex and dynamic DNNs is still an open problem.

Existing pipeline parallel systems fall into two categories. The first category (Pipedream [33] and XPipe [14]) keeps activation tensors produced during forward passes directly in GPU memory. However, due to the forward-then-backward nature of DNN training, activation tensors in the front stages reside longer in GPU memory than those in the rear stages (Fig. 1). Thus, when more input batches are injected, the front stages have to keep many more copies of activations than the rear stages.

To meet **G1** on the front stages, systems in the first category have to keep a moderate batch size [10], [39], [52], [58]. Still, a larger training batch size can lead to higher GPU ALU utilization and higher throughput [60]. In our evaluation (Section 6.1), when training Transformer with 8 GPUs, Pipedream [33] supported a batch size of only 32. Each GPU's ALU utilization rate was 42.3 percent on average, making the training throughput only 46.1 percent of the *ideal throughput*: the theoretical throughput supposing a system runs on GPUs with unlimited physical memory and utilizing all GPU ALUs (also defined in other systems [18]), and the stage partition is always balanced (**G2**).

The second category (GPipe [19] and PipeMare [59]) discards all activation tensors in the forward passes and *recomputes* them in the backward passes. This significantly alleviates the imbalanced GPU memory utilization between the front stages and rear stages, but at the cost of an extra forward pass. In our evaluation (Section 6.1), GPipe [19] supported a batch size of 128 when training the Transformer with 8 GPUs, and the each GPU's ALU utilization rate can be up to 95.6 percent. However, this all-recompute strategy inevitably leads to wasted ALU utilization of 29.4 percent, and GPipe incurred merely 66.2 percent *effective ALU utilization*: the useful GPU ALU utilization that contributes to the DNN training, but not the *recompute* utilization.

Moreover, both categories of pipeline parallel systems encounter even more severe throughput degradation when a DNN model enables NAS, where both the number and layout of the model's layers can be modified by a runtime algorithm (e.g., evolution algorithm [39], [45]). An evaluation (Section 6.3) is conducted by running a NAS-enabled Transformer [45] on one notable system in each category (i.e., Pipedream and GPipe). Compared with the defined ideal throughput, Pipedream's throughput dropped to 17.7 percent, and GPipe's throughput dropped to 25.3 percent.

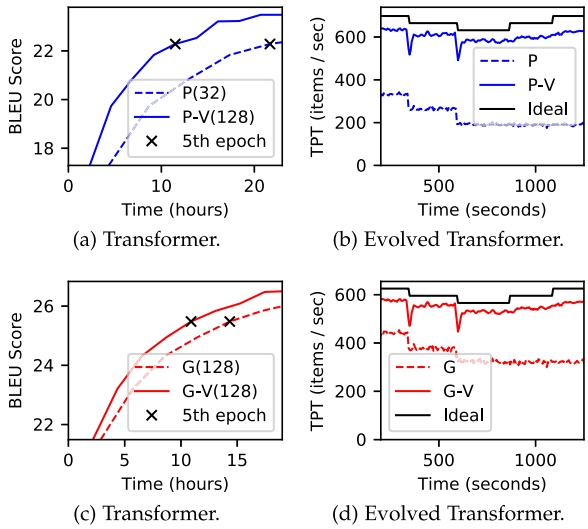

Fig. 2. (a)(b) With vPipe integrated, Pipedream-vPipe (P-V) and GPipe-vPipe (G-V) achieved faster convergence than Pipedream (P) and GPipe (G) when training Transformer [52] with 8 GPUs. (b)(d) When NAS was enabled in the Evolved Transformer [45], the training throughout (TPT) of Pipedream and GPipe further dropped, while Pipedream-vPipe and GPipe-vPipe could cope with this dynamicity.

Overall, despite great advances, existing pipeline parallel systems still incur suboptimal training efficiency on either static or dynamic (e.g., NAS enabled) DNN training. We believe the key reason is that these systems use static strategies for both memory management and layer partitioning. When stages become intense, caused by either GPU memory explosion or newly activated layers, these static strategies prevent themselves from using the available GPU resources in adjacent stages to alleviate these intense stages.

This paper presents vPipe, the first dynamic DNN layer partitioning and memory management system acting as a virtualized layer between a typical pipeline parallel system (e.g., Pipedream [33] or GPipe [19]) and its underlying execution engine (e.g., PyTorch [36] or Tensorflow [1]). vPipe automatically and transparently realizes both design goals (**G1** and **G2**) by automatically finding a globally near-optimal plan, which migrates layers among stages and relocates each layer's activations and parameters to its current stage's GPU or CPU memory. vPipe can significantly alleviate the intense stages of a pipeline and improve the pipeline's throughput in a balanced way (e.g., Fig. 2).

To achieve **G1**, instead of GPipe's all-recompute strategy, vPipe computes a hybrid plan of both *swap* and *recompute* for all layers on each stage. Specifically, *swap* asynchronously evicts activation tensors to CPU memory and prefetches them back to GPU memory before its corresponding backward usage starts. In pipeline parallelism, there usually exists an opportunity window, filled by other input batches' executions, between the forward pass and backward pass of each input batch. Leveraging this window, vPipe masks the *swap* time by precisely predicting the arrival time of the backward pass and overlapping the cost with other input batches' executions.

To achieve **G2**, instead of using a static partition strategy, vPipe online generates new partition plans and transparently *live* migrates layers from intense stages to their adjacent stages, both alleviating the memory burdens on intense

stage (**G1**) and achieving more balanced partitions with higher throughput (**G2**).

However, realizing these two goals in VPIPE must tackle two technical challenges. The first challenge is searching for a globally efficient *swap*, *recompute*, and *repartition* (*SRP*) strategy among all stages. We took the first step in the literature to model this challenge into a combinatorial optimization problem (Section 4.1). However, the problem is NP-hard due to its exponential search space [2], [3], [50].

To address this challenge, we created a fast-converging, near-optimal search algorithm using the powerful decomposition methodology [32], [47] via two observations. First, we can iteratively migrate layers from an intense stage to its adjacent stages, enabling new optimization space for a better hybrid plan of *swap* and *recompute* on each stage (Section 4.2). Second, the architecture (layout) of a typical complex DNN [39], [58] is usually constructed as a coarsened graph of repeated subgraphs, which are readily easy to be partitioned into an optimal plan [19], [33] that meets **G2**; VPIPE fast detects this coarsened graph by precisely distinguishing intra edges inside subgraphs and nested edges among subgraphs, leveraging the time series distance between each edge's two vertices (layers) collected at runtime execution.

The second challenge is how to *live* (i.e., no GPU stalls nor pipeline cleaning) migrate a layer while keeping VPIPE transparent [49] to general upper pipeline parallelism systems (i.e., VPIPE does not add nor reduce parameter staleness [14], [33], [59] to the upper system). Existing pipeline parallel systems [14], [33], [59] carefully designed various strategies to orchestrate (add or reduce) the staleness on parameter updates for higher training accuracy or throughput on specific DNNs.

VPIPE guarantees that a layer is migrated as if repartitioned by a non-live approach: stop injecting new input batches for the upper system, clean up the pipeline, migrate the layer, and reboot a new pipeline. To handle the migrated layer's unfinished backward passes, we present a new live migration protocol. Our key observation is that the time window between the activation generation (in a forward pass) and its final usage (in the corresponding backward pass) allows a subtle interleaving for VPIPE to live migrate a layer transparently without altering the parameter staleness of the upper system.

We implemented VPIPE in PyTorch [36] by adding 2782 LoC. We evaluated *all* six prevalent DNN models, including four complex DNNs Transformer [52], BERT [10], AmoebaNet [39], GNMT [58], and two simple DNNs ResNet50 [15], VGG16 [44], that are evaluated in all relevant systems Pipedream [33], GPipe [19], XPipe [14], and PipeMare [59]. The evaluation shows that:

- VPIPE was efficient in training complex DNNs. VPIPE improved Pipedream's and GPipe's throughput by 109.7 and 30.7 percent on average for four complex DNNs. VPIPE enlarged Pipedream's supported batch size by 3.75x. Within the same training time, VPIPE made Pipedream achieve higher training quality (e.g., BLEU [58]).
- VPIPE was scalable. When training the four complex DNNs on 4-16 GPUs, VPIPE's throughput increased roughly linearly with the GPU numbers. When running on 16 GPUs, VPIPE improved Pipedream's and GPipe's throughput by 323.3 and 20.7 percent.
- VPIPE was efficient in NAS workloads. When evaluated on Transformer [45] and AmoebaNet [39], the only two evaluated complex DNNs that support NAS features, VPIPE improved Pipedream's and GPipe's throughput by 421.3-463.4 percent and 245.4-291.3 percent.

Our main contribution is VPIPE, the first dynamic layer live partition and memory management system, serving as a transparent underlying acceleration layer for typical pipeline parallel systems (e.g., Pipedream and GPipe). Our major novelty is a fast and near-optimal stage-distributed search algorithm for finding a globally efficient *swap*, *recompute*, and *partition* strategy, greatly improving VPIPE's efficiency and scalability. Our secondary novelty is a transparent live migration protocol without stalling the executions or altering the upper system's parameter staleness. VPIPE's source code and evaluation framework are released at: github.com/hku-systems/vpipe.

In the rest of this paper, Section 2 presents the background; Section 3 gives an overview of VPIPE; Section 4 describes VPIPE's runtime design; Sections 5 and 6 present VPIPE's implementation and evaluation results; Section 7 discusses the related work, and Section 8 concludes.

2 BACKGROUND

2.1 DNN Training

DNN [10], [15], [29], [44], [46] is known to be the fundamental machine learning paradigm in deep learning. A DNN model typically contains hundreds of layers, and the goal of DNN training is to find an appropriate set of model parameters to fit a training dataset. Each DNN training process typically consists of millions of iterations, each containing a forward pass, a backward pass, and an optimization step.

The memory consumption of DNN training contains four parts: parameters of each layer; activations, i.e., feature maps produced by each layer in the forward pass; gradients, i.e., gradient maps produced by each layer in the backward pass; and scratch space for computation. Among these four parts, activations take the most significant portion (up to 73.3 percent) of the total memory consumption for DNN training. Activations are created in the forward pass and reused in the backward pass, so there exists a large time window between the two memory accesses. Activation memory is the major optimization target in previous work [18], [37].

2.2 Pipeline Parallel DNN Training

With the DNN training getting increasingly computation and memory intensive, distributed training systems across multiple GPUs become a must. Distributed training systems can be categorized as data parallel or model parallel. Data parallel systems [28] let each GPU maintain a copy of the complete model. In each iteration, each GPU trains on a small batch and synchronizes the parameter updates with other GPUs using all reduce [43] or parameter sever [28]. However, data parallelism is not designed to train large DNNs that cannot fit into a single GPU's memory.

Pipelined model parallelism (i.e., pipeline parallelism) aims to scale the supported DNNs to the number of GPUs

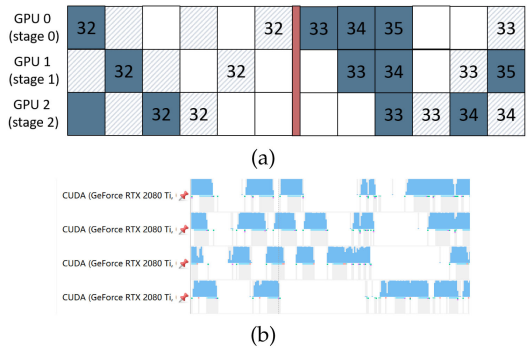

Fig. 3. Logical BSP pipeline (a) that demonstrates the bubble problem and a realtime nsys/nvprof GPU profiling (b) that verifies the bubble problem in BSP pipeline with four-stage GPipe training; red blocks are sync barriers.

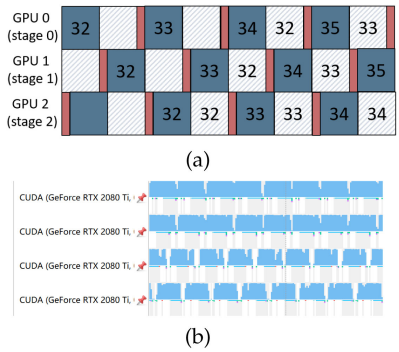

Fig. 4. Logical ASP pipeline (a) and a realtime nsys/nvprof GPU profiling (b) of ASP pipeline with four-stage Pipedream training; red blocks are sync barriers.

by partitioning a DNN model into multiple stages (a consecutive set of layers) and letting each GPU handle one stage. Pipeline parallelism is a pipeline version of model parallelism, where vanilla model parallelism leads to severe under-utilization due to the *bubble problem* caused by the sequential dependency between stages. Pipeline parallelism overlaps the computation and waiting time of different input batches, fills the bubbles, and improves the utilization. Based on how a pipeline parallel system handles synchronization of DNN parameters among input batches, the system falls into two categories: barrier synchronous parallel (BSP) systems and asynchronous parallel (ASP) systems.

BSP systems (e.g., GPipe [19]) let a set of training input batches work on the same version of model parameters, aggregate gradients computed by these iterations, and enforce a barrier that stops the pipeline to apply the gradients to the model parameter. BSP systems achieve almost the same statistical performance as vanilla model parallelism [19]. However, as shown in Fig. 3a, a BSP pipeline logically still incurs bubbles during each barrier synchronization, and we verified this in Fig. 3b by profiling the GPUs during a four-stage BSP pipeline training.

ASP systems (e.g., Pipedream [33] and PipeMare [59]) remove the sync barrier and let each input batch directly update the model parameters. Although bubbles are eliminated (as shown in Fig. 4), ASP systems suffer from parameter staleness in two aspects. First, the parameter version differs between a pipeline's forward pass and backward pass. Second, the parameter version differs among stages within the training of an input batch. Pipedream [33], XPipe [14], and PipeMare [59] provide various algorithm-level mitigation to the parameter staleness problem. vPipe is designed to be a transparent layer under either a BSP or an ASP pipeline parallelism algorithm; and vPipe's designs (Section 4.3) do alter the weight staleness in the upper systems.

Scheduling. One forward one backward (1F1B) scheduling is first introduced by Pipedream [33] and adopted by successive systems (e.g., PipeMare [59] and XPipe [14]). In 1F1B scheduling (e.g., Fig. 1), each stage alternates between performing forward pass for a current input batch and backward pass for an earlier input batch. 1F1B is widely adopted due to its high computational efficiency [33], [59] and low memory usage. Therefore, in this paper, we assume that the upper pipeline parallel systems adopt 1F1B scheduling.

2.3 Dynamic DNN Training

Recently, more and more developers have adopted dynamic DNN training where the number of layers varies with the training inputs (e.g., DyNet [34]) or the training is exploratory (e.g., neural architecture search [45], [55], [57], [62]). In such a case, a training workload (i.e., the GPU computation and memory required for training) varies as the training proceeds. Since the efficiency of pipeline parallelism highly depends on the workload partition among stages, this dynamicity exposes special requirements for pipeline parallel systems.

The variance of training workload usually happens very frequently. For example, a neural architecture search (NAS) process [39], [45] adopts an evolutionary algorithm that trains a set of models, fast eliminates those with low fitting scores, and initiates new ones. Thus, "bad" models can be eliminated within a few minutes [39], [45].

Existing pipeline parallel systems profile a static partition before the training starts. This static partition inherently cannot adapt to the dynamicity in the training process. vPipe copes with this dynamicity by a wait-free live layer migration protocol (Section 4.2) that transparently re-balances the training load when changed.

3 vPipe's Architecture

Fig. 5 shows vPipe's architecture, a virtualized layer between a typical pipeline parallel system and its underlying execution engine. On each host, there is a virtualized tensor manager, a training monitor, and a layer manager. On the host of the last stage, there is a global planner.

Virtualized Tensor Manager (VTM) provides fine-grained management to each parameter and activation tensor. VTM holds each layer's tensor (parameter or activation) information, including layer ID, stage ID, property (parameter or activation), training iteration ID, version, management policy (*vStatus*), storage status, and the pointer to the tensor's real storage constructs. An activation tensor's information is initialized in vPipe's tensor manager when created and deleted when released. For parameter tensors, vPipe creates tensor information as long as the model is initialized. The management policy of a layer's tensors is managed by the layer manager.

Training monitor monitors each stage's runtime statistics, including real-time memory usage of each GPU on these hosts, PCIe bandwidth usage, network usage, execution time, and recompute time. Along with forward passes of the normal training

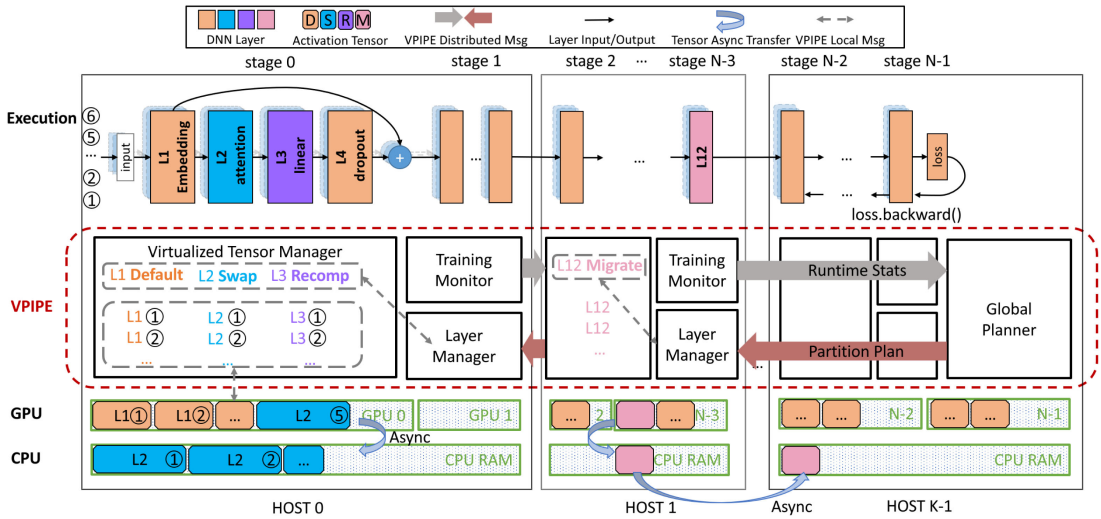

Fig. 5. Architecture of vPipe. vPipe is a virtualized layer between a typical pipeline parallel system (e.g., Pipedream [33] or GPipe [19]) and its underlying execution engine (e.g., PyTorch [36] or Tensorflow [1]). We use different colors to refer layers set by vPipe's operations including default (D), swap (S), recompute (R), and migrate (M).

iterations, the training monitor passes its own runtime statistics and the upstream stages' (if any) to its downstream stages.

Global planner collects the runtime statistics of all stages at the end of every forward pass. It produces new partition strategies (if needed) according to vPipe's SRP algorithm (Section 4.2). It resides on the last host for two reasons. First, in pipeline parallelism, rear stages usually have less computation and communication burdens. Second, as the runtime statistics are collected every training iteration, vPipe transfers the runtime statistics along with the forward pass and distributes the new partition (if any) along with the backward pass. By doing so, vPipe's global planner does not need extra distributed coordination.

Layer manager receives a new partition strategy from the global planner, diffs the new partition from its current partition to check whether a layer migration should be scheduled. For example, when a layer needs to be migrated, the migration manager of the source stage will coordinate with the tensor manager to asynchronously swap the layer's parameter tensors and activation tensors to the CPU memory; and then transfer the parameter and activation tensors to the migration manager of the target stage. The migration manager of the target stage will initialize the layer in the target GPU, receive the parameter and activation tensors from the source stage, and append the new layer to the forward pass and backward pass executions (Section 4.3). Layer manager also produces the local *swap* and *recompute* policies (Section 4.2).

Overall, vPipe's design is transparent to the upper pipeline parallel systems. We integrated vPipe into an ASP system Pipedream [33] and a BSP system GPipe [19]. For vanilla Pipedream, we set all layers' $vStatus$ to $default$; and for vanilla GPipe, we set all layers' $vStatus$ to $recompute$. vPipe can also be integrated into other pipeline parallel systems (e.g., PipeMare [59] and XPipe [14]) as long as they support an imperative programming model.

4 vPipe's Runtime

4.1 Problem Modeling

A major challenge for vPipe's design is to find an optimal strategy of swap, recompute, and partition (SRP) so that the steady-state throughput of the training pipeline can be maximized. Since there is no model to quantify the complexity of this SRP challenge, we take the first step in the literature to formalize the SRP challenge, transform it into a combinatorial optimization problem, and solve it by a decomposition algorithm (Section 4.2).

A DNN is a graph $G(N, E)$ with N layers (e.g., matrix operation) and E edges connecting the layers. In pipeline parallelism, a DNN model is partitioned to p stages, and each stage is placed on one GPU (p GPUs in total). To maximize the pipeline utilization, in a typical pipeline parallelism scheduling (Section 2), at least p input batches are simultaneously injected into the same pipeline. For each layer in the model, we denote it with (f_i, b_i, m_i, a_i), including a forward pass time f_i, a backward pass time b_i, a parameter memory m_i, and an activation memory a_i.

The major constraint for pipeline parallel training is **G1**: on each GPU, the training GPU memory usage should not exceed any GPU's physical memory limit (M). In pipeline parallelism, the memory consumption of all layers in each stage contains two parts. The first part is a constant memory consumption ($m_i^{constant}$) that does not vary with the number of injected input batches; the second part is the dependent memory consumption ($m_i^{dependent}$), which depends on the number of injected input batches and differs among stages: given a stage k, $p-k$ copies of $m_i^{dependent}$ should be kept in memory. In BSP systems, parameters are updated synchronously (Section 2), and all input batches in a pipeline share the same version of parameters, thus $m_i^{dependent}$ is a_i and $m_i^{constant}$ is m_i. In ASP systems, each training iteration in a pipeline may have an independent version of m_i, thus $m_i^{dependent}$ contains both a_i and m_i.

To reduce memory consumption, a pipeline parallel system can apply swap or recompute strategy to each layer's dependent tensors, which are the main memory burden in pipeline parallelism. Thus, for each tensor in a layer, we denote its memory management policy with (D_i, R_i, S_i), where $D_i, R_i, S_i = 0$ *or* $1, D_i + R_i + S_i = 1$. $D = 1$ means the tensor by default resides in the GPU memory; $S = 1$ means the tensor will be proactively swapped to CPU memory and

swapped back to GPU before usage; and $R = 1$ means the tensor will be dropped and recomputed by the backward pass. Thus, in pipeline parallelism, the memory constraint of each stage can be denoted as:

$$\sum_{l_k \le i \le r_k} m_i^{constant} + (p-k) * \sum_{l_k \le i \le r_k} D_i * m_i^{dependent} \le M. \quad (1)$$

Nevertheless, the $recompute$ of layers introduces extra computation time to the backward pass. Thus, a stage's backward time is the sum of the original backward pass time, the $recompute$ time (i.e., extra forward pass of recomputed layers), and the $swap$ time if the swap time cost is larger between the normal execution time (i.e., $max(0, swap\ time - execution\ time)$):

$$t^{bwd} = \Sigma(b_i + R_i * f_i) + max(0, (2 * \Sigma(S_i * m_i^d / P) - (t^{fwd} + t^{bwd}))). \quad (2)$$

Finally, we formalize the SRP challenge to a combinatorial optimization problem: given n layers and p GPUs, find a swap or recompute policy for each layer (meet **G1**), as well as a partition (meet **G2**), such that the pipeline throughput can be maximized. The throughput of a pipeline is the lowest throughput among all stages [22], [33]. All stages in a pipeline have the same request rate. Thus, the pipeline's throughput bottleneck is the stage that has the longest execution time (sum of the largest t^{fwd} and largest t^{bwd}). Therefore, we convert this problem to finding a partition and a swap/recompute policy such that the longest stage execution time can be minimized:

$$\begin{aligned} &\text{minimize} && \max_{1 \le k \le p} (t_k^{fwd} + t_k^{bwd}) \\ &\text{subject to} && (1)(2). \end{aligned} \quad (3)$$

This optimization problem is hard to solve for two reasons. First, the feasible set of this combinatorial optimization problem spans an extremely large search space ($O(3^{|N|}p^{|N|})$), as each of layers N can have three memory management policies and fall into p partitions. A graph partition problem itself is well-known to be NP-complete [50]. Second, constraint (2) indicate that both the memory management policy of all layers ((D_i, R_i, S_i), $for\ 1 \le i \le n$, denoted as Var^{sr}) and the stage partition plan (denoted as Var^{p}) can affect the optimization objective in (3), making this problem a multi-variable combinatorial optimization.

4.2 Swap, Recompute, and Repartition

We solve this multi-variable and combinatorial optimization problem by decomposition [32], [47] methodology. The idea of the decomposition methodology is to break a problem into smaller sub-problems coordinated by the master problem (i.e., the optimization problem). Inspired by the conventional decomposition method [32], [47], the key intuition is to iteratively migrate a layer from an intense stage where the GPU resource is exhausted to a relief stage and let the intense stage have more optimization space to search for a better hybrid plan of swap and recompute.

We decompose the master problem into two sub-problems. First, we assume that Var^{p} is constant, and each stage locally finds a swap and recompute plan (Var^{sr}) depending on its GPU resource to minimize the objective function (3). Second, we assume that Var^{sr} is constant, and stages should be repartitioned (i.e., find an optimal Var^{p}) to minimize (3). Algorithm 1 shows our decomposed algorithm by iteratively resolve these two sub-problems.

Algorithm 1. Decomposed SRP Algorithm

```
1: Stage 1,..., p:;
2: Function LayerManagerIterate():
3:   newPlan = receiveBwdProp() ;
4:   diff = compare(this.plan, newPlan);
5:   if diff! = null then
6:     migrating = True;
7:     for l in diff do set(l.vStatus, Migrate);
8:     stats = retrieveStats();
9:     optimizeSR(stats) ##Algorithm 2;
10:  return;
11: Function TrainingMonitorIterate()
12:  if ! migrating then
13:    stats = receiveFwdProp();
14:    mem = cudaMemStats();
15:    t^fwd, t^bwd = getExecTime();
16:    stats.append(this.meta, mem, t^fwd, t^bwd);
17:    fwdPropagate(stats);
18:  return;
19: Global Planner:;
20: Function: GlobalPlannerIterate()
21:  stats, migrating = receiveFwdProp();
22:  if migrating then
23:    return;
24:  unbalanced = checkBalanced(stats);
25:  if unbalanced then
26:    newPlan = layerRepartition() ##Algorithm 3;
27:    bwdPropagate(newPlan);
28:  return;
```

Swap and Recompute. For both $swap$ and $recompute$, the goal is to reduce the memory footprint with the lowest overhead. For the $swap$, our goal is to maximize the overlapping between $swap$ and the normal execution. For the $recompute$, our goal is to select the cheapest layer with maximized memory saving to recompute. It has been well studied in recent work (e.g., Capuchin [37]) that using a hybrid combination of $swap$ and $recompute$ of activation tensors can effectively reduce training memory on single GPU DNN training. However, applying $swap$ to a pipeline parallel system has to address two subtle points.

First, an efficient $swap$ plan should precisely *predict* when a tensor that has been swapped to CPU RAM will be reused in the backward pass. In single GPU training, an activation tensor is generated by the forward pass of an input batch training. The backward pass directly follows the forward pass. Thus, existing $swap$ techniques used in single GPU training systems (e.g., SwapAdvisor [18], Capuchin [37], vDNN [40], and SuperNeuron [56]) directly make predictions based on a DNN's graph (either profiled or runtime generated).

However, there usually exists a window in pipeline parallelism, filled by other input batches' executions, between the forward pass and backward pass of each input batch. To make a precise prediction, vPipe oversees the runtime statistics of each forward pass and its backward pass across all stages of a pipeline (line 21-28 in Algorithm 1), and let each vPipe's layer manager precisely predict the arrival time of each backward pass execution.

Algorithm 2. optimizeSR()

```
1: Input: layers in a stage, t^fwd, t^bwd, M, P, rank;
2: Foreach l in layers do
3:    if l.a/P > l.t^fwd then
4:       l.cost = l.t^fwd;
5:       l.op = Recompute;
6:    else
7:       l.cost = l.m^activation/P;
8:       l.op = Swap;
9:    l.gain = l.m^activation/l.cost;
10: window = t^fwd + t^bwd;
11: space = P * window;
12: sorted = sortByGain(layers);
13: while space ≥ 0 do
14:    l = sorted.pop();
15:    set(l.vStatus, S);
16:    space = space − rank * m^activation
17: while memConsume(layers) > M do
18:    l = sorted.pop();
19:    set(l.vStatus, l.op);
20: foreach l in layers do
21:    l = sorted.pop();
22:    set(l.vStatus, Default);
```

Second, in pipeline parallel systems, $swap$ and network communication impose severe burdens on the PCIe lanes, causing severe PCIe interference that is not addressed by single GPU training systems. In vPipe, both network communication and $swap$ that pass throughput PCIe are asynchronous streams [4]. To handle the PCIe interference, vPipe sets priorities to different asynchronous streams that pass through PCIe. vPipe sets a higher priority to network communication for not blocking the pipeline execution.

vPipe's swap and recompute algorithm (Algorithm 2) works as follows. For each stage, the algorithm takes a set of layers, a memory limit M, PCIe bandwidth P, stage $rank$ (p-k), t^{fwd} and t^{bwd} of this stage as input. vPipe first sort all layers by the potential memory saving gain of either $swap$ or $recompute$ (line 2-9). Until the PCIe is full, vPipe selects tensors according to their memory saving gains to be asynchronously swapped (line 13-16). After that, if the memory limit is still reached, vPipe chooses whether to swap or recompute an activation based on their swap/recompute cost and memory saving gain (line 17-19). For the rest of the layers, vPipe keeps them by default (line 20-22). Leveraging the first subtle point, vPipe can precisely overlap the async swap cost of these tensors with normal execution. With the second subtle point, the async swap will not block the network communication of normal training execution. Consequently, Algorithm 2 reduces the recompute overhead with async swap in existing pipeline parallel systems (e.g., GPipe [19]). vPipe swaps activation tensors first, as activation takes the most memory consumption; vPipe swap parameter tensors only if activation tensors are all swapped, which rarely happens in our evaluation.

Layer Partition. The problem of partitioning a graph $G(N, E)$ into p equal partitions with the lowest cross-partition communication cost is known to be NP-complete [3] and has extensive applications in many areas, including VLSI design [24], matrix factorization [7], and social network clustering [35]. Kernighan-Lin (KL) algorithm [25] is known to produce excellent partitions for a wide class of problems and is used quite extensively [17], [27]. To achieve a multi-partition, it recursively produces bi-partition of graph G and iteratively improves it by exchanging nodes in both partitions. KL algorithm is costly and takes $O(r|N|^2 \log |N|)$ [11] time (e.g., up to 16s to partition a complex DNN model into 16 stages), where r is the repeated cycles. There are many approximate algorithms [11], [12], [16], [48] that tend to be fast (near-linear) but often yield partitions that are worse than those obtained by KL algorithm [13], [23], [41] .

To make KL algorithm efficient, multi-level schemes reduce the size of the graph (i.e., coarsen the graph) by collapsing vertices and edges, partitioning the smaller graph, and then uncoarsening it [17], [23]. Multi-level scheme has been used in many areas, including matrix factorization [7] and VLSI design [24]. However, these algorithms assume domain-specific requirements for the graph (e.g., a sparse matrix [7] or a planar graph [24]), which are not applicable to a complex DNN graph (e.g., AmoebaNet [39]). Moreover, existing multi-level schemes all take multiple coarsen steps. In vPipe, leveraging the time series implied by the DNN's sequential executions, we identify two domain-specific heuristics to design a fast and online multi-level graph partition algorithm with a one-step coarsen scheme.

First, Deep Learning experts have already constructed the graphs of complex DNNs (e.g., Transformer, BERT, AmoebaNet, and GNMT), prevalently deployed with pipeline parallelism, as sequentially connected and repeated subgraphs of layers. Each subgraph is usually a basic block (e.g., a Transformer block) for constructing a large DNN. Inside each subgraph, there are intricate local edges (nested edges) forming multiple execution branches. Partitioning such a subgraph in two stages usually incurs huge network communication costs between two GPUs.

There are also sparse nested edges that form branches among blocks. However, network communication costs of partitioning these sparse nested edges are often static and do not vary with the partition plan. For example, in the BERT model, each block should take input from the first embedding layer, and it is necessary to pass the embedding output to all stages. Thus, under any partition plan, the network communication costs of transferring this input to all stages are persistent.

Second, different from conventional graphs in partitioning problems [2], [3], [50], in a DNN graph, vertices (i.e., layers) are executed by the training engine in time series. If a nested edge connects two vertices that have a gap that is larger than a stage's execution time in the time axis, the edge has a high chance to be a sparse nested edge. If a nested edge connected two vertices very close to each other in the time axis, the edge is likely to be part of a subgraph.

Based on these heuristics, vPipe's layer repartition algorithm (Algorithm 3) has three steps. First, vPipe (line 7-21)

coarsens the DNN graph. In this step, each edge in a DNN graph is classified with $O(|N|+|E|)$ cost to three categories: critical edges that construct the sequential backbone of the DNN graph, sparse nested edges, and subgraph edges. Then vPipe merges the subgraph edges to the sequential backbone edges by aggregating their execution time and communication. Second, vPipe partitions this merged graph by iteratively applying bipartition with KL algorithm [50] (line 22-26). Third, vPipe uncoarsens the merged graph to the original DNN graph and refines the partition to see if any potential better partition exists by KL refinement [17] (line 27-29).

Algorithm 3. layerRepartition()

```
1: Input: DNN Graph G(N, E), runtime statics of each layer
   (layers), e.g., invoke time (T) of each layer;
2: sorted = sortByTime(layers);
3: G^coarsened = coarsen(G(N, E));
4: bound = partition(G^coarsened);
5: G = uncoarsen(G^coarsened) ;
6: bound = refine(G, bound);
7: Function: coarsen(G(V, E))
8:    mean = sum(t)/p;
9:    E* = [];
10:   foreach l1, l2 in pairwise(sorted) do
11:      ##detect critical path edges;
12:      if e(l1, l2) in E then
13:         annotate e(l1, l2) as critical edge ;
14:         E*.append(e(l1, l2))
15:   foreach e in E − E* do
16:      ##distingush sparse and subgraph edges;
17:      if e.v2.T − e.v1.T > mean then
18:         annotate e(l1, l2) as sparse edge
19:      else
20:         annotate e(l1, l2) as subgraph edge
21:   merge(E, E*);
22: Function: parition(G(V, E), p)
23:   if p==1 then
24:      return
25:   bound, G1, G2 = KLParitition(G, cost);
26:   return bound, partition(G1, p2), parition(G2, p2);
27: unction refine(G(V,E), bound)
28:   foreach b in bound do
29:      KLRefine(G(V, E), b)
```

Analysis. vPipe's Algorithm 1 decomposes a master problem into two sub-problems [32], [47]. vPipe's Algorithm 2 is optimal as the sub-problem is a linear optimization with simple constraints (i.e., the memory limit and the PCIe limit). vPipe's Algorithm 3 is a successive algorithm of the Kernighan Lin (KL) algorithm. KL algorithm is a bipartition algorithm that starts from an initial bipartition of a graph and exchanges the vertices of the two partitions to see whether a better partition can be found [2], [3], [50].

The time complexity of the original KL algorithm is $O(r|N|^2 \log|N|)$, where r is the repeated cycles, and N is the total set of layers. The time cost of running KL algorithm on complex DNNs (e.g., AmoebaNet) is huge (up to 16s for each run). With our two heuristics on recent complex DNN graphs, vPipe's partition algorithm uses a coarsen phase of complexity $O(|N|+|E|)$ that coarsens a complex DNN graph (e.g., AmoebaNet graph with 4280 layers/vertices and 5080 edges) into a much smaller graph (e.g., coarsened AmoebaNet with 132 vertices and 142 edges). By doing so, the time cost of KL algorithm is greatly reduced. On partitioning various DNN model, evaluation (Section 6.4) shows that vPipe's partition algorithm speeds up the KL algorithm by 4x-32x and achieves 0.15s-0.46s time cost (less than the process time 1.21s-6.98s of one training input batch), fast enough to be deployed online.

4.3 Live Layer Migration

Existing pipeline parallel systems (e.g., Pipedream and GPipe) adopt a static layer partition before execution (Section 2). To migrate a layer in these systems, developers need to adopt a non-live approach: stop the runtime, modify the layer partition configuration, and reboot the whole training process. This process suffers from heavy bootstrap overhead, including runtime initialization, model initialization, and data loading (Section 2). Such a heavy overhead might dramatically decrease the training efficiency when layer migration is frequently triggered under a dynamic training process (Section 6.4).

In vPipe, we aim to design a live layer migration protocol for pipeline parallelism with a key technical requirement that the layer migration should remain transparent to the upper systems so that vPipe will not alter the upper systems' parameter staleness.

Existing pipeline parallel systems fall into two categories: BSP systems (GPipe [19]) and ASP systems (Pipedream [33], PipeMare [59], and XPipe [14]). BSP systems have no parameter staleness (Section 2.2). ASP systems adopt various parameter staleness strategies on different design goals. BSP and ASP systems have their own strengths on particular workloads. For instance, in Table 3, GPipe achieved better accuracy than Pipedream on training Transformer while achieved worse accuracy than Pipedream on training BERT. Thus, vPipe is designed to be transparent to the upper systems so that vPipe does not alter their parameter staleness. vPipe lets the programmer explicitly annotate the type of system.

However, it is challenging to transparently migrate a layer without losing liveness for both BSP and ASP systems. The reason is that at any time in a pipeline, a layer can always have multiple unfinished backward executions, and these backward passes will produce updates to the layer parameters. To avoid altering the parameter staleness, during the migration of a layer, no updates produced by these backward passes should be lost.

Moreover, in the typical scheduling of ASP systems (Section 2.2), layers on different stages have different pipeline execution interleaving. For example, in the last stage, the forward pass of an input batch directly works on the parameter updated by the last input batch, while in the first stage, the forward pass works on the parameter updated by a much earlier input bach. For BSP systems, forward passes on all stages work on the same version of parameters until a parameter synchronization occurs. To avoid altering the parameter staleness, during the migration of a layer, vPipe ensures that when a layer is migrated among stages, the execution interleaving of this layer should change

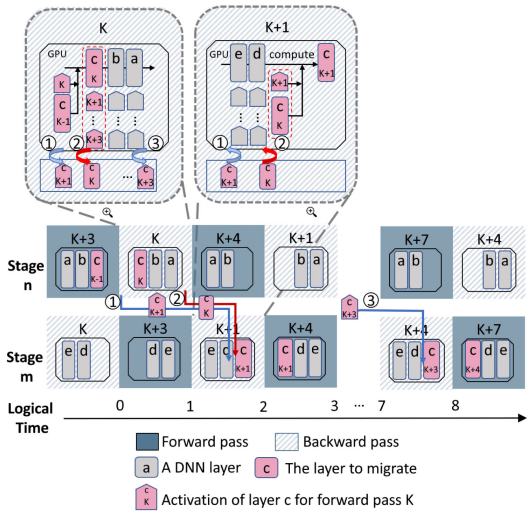

Fig. 6. A forward layer migration triggered after the ending of input batch k from stage n to stage m.

accordingly. By doing so, vPipe guarantees that a layer is migrated as if repartitioned by a non-live approach.

We formalize the above transparency requirements. Given a new input batch k, for q layers $\{l_1, l_2, ..., l_q\}$ in stage n of a training pipeline ($0 \leq n < p$, where p is the number of stages and the number of simultaneously injected input batches), each layer must have $p - n - 1$ unfinished backward passes. In ASP systems, in stage n, the forward pass of input batch k should work on the version (V_k) of layer parameters updated by $k - p + n$. In BSP systems, for all stages, if the parameter synchronization happens every $u * p$ input batches, the forward pass of input batch k should work on the same parameter version $k \bmod u * p$.

$$V_{fwd}^k = \begin{cases} k - p + n & \text{if ASP} \\ k \bmod u * p & \text{if BSP} \end{cases}. \quad (4)$$

When a set of layers $\{l_i, ..., l_j\}$ are going to be migrated from stage n to stage m, where $m = n \pm 1$, for each layer, vPipe should migrate $p - n - 1$ copy of activation tensors for unfinished backward passes. Meanwhile, for ASP systems, the V_k should be changed from $k - p + n$ to $k - p + m$.

A strawman stop-and-copy migration approach is to stop the execution, synchronously transfer parameter tensors and activation tensors, and resume the execution. However, on training complex DNNs, the tensors to be migrated can be up to several gigabytes, leading to a long stall.

In vPipe, we present a live runtime layer migration protocol. Without losing generality, to ease discussion, Figs. 6 and 7 shows an example of a forward layer migration in a four-stage (i.e., $p = 4$) pipeline, where $n = 0$ and $m = 1$. If Stage n is going to migrate layer c to Stage m after the ending of input batch k, the migration will work as follows. In prepare stage, Stage n sends a *prepare* message to stage m to inform the migration of layer c. Stage m initializes the layer module of layer c and moves the module to GPU memory. Then, stage m sends a *ready* to stage n.

Once stage n receives *ready*, the migration immediately starts in its next forward pass (i.e., forward pass of input batch $k + 4$ in Fig. 6). (1) Stage n immediately asynchronously

Fig. 7. Realtime nsys/nvprof GPU profiling of a forward layer migration. Pink blocks are GPU-to-CPU memory copy; green blocks are CPU-to-GPU memory copy. After migration, a higher utilization can be visually observed on the target GPU. We disabled *swap* to highlight the migration memory copies.

transfers activation tensors for backward pass of input batch $k + 1$ (denoted as backward $k + 1$). (2) After the next backward pass (i.e., backward k) finishes, stage n transfers the parameter tensors of layer c (updated by backward k) to stage m. Stage m will wait for the arrival of the parameter tensors of layer c and process layer c in its next backward pass (i.e., backward $k + 1$ is processed in stage m). (3) The subsequent layer c's activation tensors created by input batch $k + 2$, $k + 3$, ..., $k + p - n - 1$ (i.e., $k + 2$, $k + 3$ in Fig. 6) are continuously and asynchronously copied. vPipe ensures that the backward $k + 2$, $k + 3$, ..., $k + p - n - 1$ will not start at stage m until their corresponding activation tensors arrive. When vPipe is integrated into an ASP system, vPipe will transfer the activation tensors and the corresponding parameter tensors to migrate a layer.

Overall, vPipe's live layer migration merely affects the normal execution as in step (1) and (3), vPipe asynchronously transferred the activation tensors of migrated layers, and we verified this by profiling in Fig. 7. To avoid altering staleness, vPipe ensures that the V_{fwd}^k remains consistent when a layer is migrated from stage n to stage m. In vPipe, layer migrations can be triggered multiple times during a triggering of vPipe's Algorithm 1 (tens of seconds in Section 6.4). In our evaluation, each migration with a non-live migration approach stalls the pipeline execution by 1.1-6.8s, while vPipe's migration protocol remains live.

5 System Implementation

vPipe's design leverages the imperative features from PyTorch. The current popular deep learning frameworks are typically based on either imperative or declarative programming. The imperative programs are similar to Python or C++ programs, which perform computations during the execution. PyTorch adopts it as the default and only execution mode. Overall, vPipe is currently implemented by modifying 2782 LoC to PyTorch [36]. vPipe's design and implementation is common for all DNN training engines that follow an imperative programming style. In this section, we present three key points to implement vPipe in PyTorch: how to support distributed on-demand *swap* and *recompute*; how to migrate layers between stages; how to implement an NAS process [39], [45] in vPipe, as there is no existing literature that describes how to implement an NAS process in pipeline parallelism.

For the first point, to capture access patterns of tensors, vPipe intercepted PyTorch's activation creation in *forward* passes and reuse in *backward* passes. In PyTorch, an activation tensor is created and saved to an edge of an automatic gradient computation (*autograd*) graph in a data structure *SavedVariable*. vPipe intercepted the member functions of *SavedVariable* and saved the tensor pointers to vPipe's VTM

module (Section 3). In PyTorch, *SavedVariable* can refer to both a parameter tensor and an activation tensor. vPipe distinguished a parameter tensor and an activation tensor by assigning each a property upon their initialization (parameter tensors are initialized during a model initialization, i.e., *module* initialization in PyTorch). To precisely predict when to swap back a tensor, vPipe's VTM modules pass the captured access patterns of tensors to other stages (Section 4.2).

To support asynchronous and on-demand *swap* for activation tensors in PyTorch, vPipe added a tensor level asynchronous *swap* feature to PyTorch. PyTorch 1.5.0 currently only supports a synchronized *swap* for tensor implementation (i.e., the main thread will be blocked during the *swap*). Moreover, to accelerate the tensor *swap* from CPU memory to GPU memory, in vPipe, we stored the tensors that are *swapped* to CPU memory in a *pinned memory*. The technical reason is that in PyTorch, CPU memory to GPU memory copies are much faster when they originate from pinned (i.e., page-locked) memory. vPipe used the *pin_memory*() method for PyTorch's CPU tensor storage.

vPipe's *recompute* leverages PyTorch's *checkpoint* library, which is a builtin library for recomputing activations. A major implementation obstacle for on-demand *recompute* is to change the training statement at runtime. In vPipe, we used python's builtin feature *exec_stmt*, which takes a piece of statement as input and executes the statement, to to modify a stage's execution statement at runtime and on-demand decide whether to *recompute* a layer's activation.

To support layers migration between stages (thus, a stage of DNN is dynamic), vPipe maintains a DNN stage as a structured graph data and has a simple parser that switches between the graph description of DNNs and the PyTorch imperative statement (using *exec_stmt*). Thus, when a layer migration happens, on the target stage, vPipe modifies the graph description, initializes the corresponding layer module in PyTorch, overwrite the layer's state by the migrated layer's state, and adds the new layer to the stage's execution statement. On the source stage, vPipe removes the layer from the stage's execution statement and delete the layer from the GPU memory. vPipe both supports both branches in among stages and branches among layers.

To support NAS in pipeline parallelism, we implemented the NAS process on both Pipedream and GPipe (Section 6.3) based on the official description of the evolved Transformer [45] and AmoebaNet [39]. Overall, there are two key components for a NAS process: an evolution algorithm that iteratively explores new DNN architectures; and a just-in-time runtime that switches the training workload according to DNN generated by the evolution algorithm.

In an evolution algorithm, when a DNN switch occurs, our NAS implementation *deactivates* the differed layers in the existing DNN, *activates* the new layers, and reset parameters when a DNN switch finishes. The above implementation leverages PyTorch's imperative feature (i.e., *exec_stmt*) and fast switches between two DNNs without extra stop and initialization time.

6 Evaluation

Testbed. Our evaluation was conducted on a GPU farm with 8 hosts. Each host had 4 Nvidia 2080TI GPUs, 20 CPU cores, and 64 GB RAM. Each GPU had 11 GB physical memory and was connected to the host with PCIe 3.0 X16 that provided a total data transfer bandwidth of 15760 MB/s. Hosts are connected with 100 Gbps Ethernet, and the average ping latency is 0.17ms.

TABLE 1
Models and Datasets

Task	Model	Dataset
Image Classification	VGG16 [44]	ImageNet [9]
	Resnet50 [15]	ImageNet [9]
	AmoebaNet [39]	ImageNet [9]
Translation	GNMT [58]	WMT16 EN-DE [42]
	Transformer [52]	WMT16 EN-DE [42]
Language Modeling	BERT [10]	WMT16 EN-DE [42]

Workloads. We evaluated six well-studied DNN models (Table 1) that are widely used in the deep learning community. BERT [10], Transformer [52], AmoebaNet [39], and GNMT [58] are four large DNNs often trained by pipeline parallelism [19], [33]. Transformer [45] and AmoebaNet [39] are two typical workloads that have been applied with Neural Architecture Search. We used the open-source release of each model.

These models cover *all* prevalent DNNs evaluated in existing pipeline parallel systems, including Pipedream [33], GPipe [19], XPipe [14], and PipeMare [59]. For other models, including S2VT [53] and AWD LM [31] evaluated in these systems, they are surpassed by the DNNs we evaluated and no longer prevalent. We evaluated two well-known datasets: WMT16 [42] for NLP and ImageNet [9] for vision.

Baselines. We integrated vPipe to two baseline systems: the most notable ASP pipeline parallel system Pipedream [33] and the most notable BSP pipeline parallel system GPipe [19]. For Pipedream, we used its open-source release [33]; for GPipe, we implemented GPipe by applying a strong synchronization barrier (Section 2) on Pipedream's codebase because GPipe has no official release on PyTorch. Each integration of vPipe took only several LoC changes. For a baseline system (e.g., Pipedream), we used Pipedream-vPipe to represent Pipedream integrated with vPipe. We compared the throughput of Pipedream-vPipe with Pipedream alone to indicate vPipe's improvement on Pipedream. Overall, we evaluated four systems: Pipedream-vPipe, GPipe-vPipe, Pipedream, and GPipe.

There are also successive systems (i.e., XPipe [14] and PipeMare [59]) that mitigate Pipedream's parameter staleness. However, all these systems share the same performance model as either Pipedream or GPipe.

Batch Size and Training Setup. For all systems, we set the training batch sizes of each DNN to the largest batch size that can be supported without exceeding all GPU's physical memory limit. As Pipedream directly keeps all activation tensors in GPU memory, to avoid exceeding GPU memory limit on the front stages, the training batch size supported by Pipedream was 3.2x less than other evaluated systems (e.g., GPipe). For all systems, without specification, we evaluated them on 8 GPUs and set their default partition (shown in Table 2) by the static partition profiler provided by Pipedream [33], which is the only system that explicitly

TABLE 2
Default Settings of Baseline Systems

Model	layer #	l.r.	Default Partition
BERT	488	5×10^{-3}	[60, 62, 62, 62, 62, 61, 61, 58]
Trans.	332	5×10^{-4}	[41, 41, 42, 43, 43, 42, 42, 38]
Amoe.	2190	5×10^{-5}	[283, 238, 238, 238, 238, 286, 237, 432]
GNMT	86	6×10^{-2}	[11, 12, 11, 10, 8, 9, 13, 12]
VGG16	40	2×10^{-2}	[22, 18]
ResNet50	175	2×10^{-2}	[116, 59]

Baseline systems with vPipe start with the same default partition.

describes a partition scheme. In Section 6.2, when training with varied GPUs numbers, the default layer partition was also produced by Pipedream's static partition profiler. We also show the learning rate (l.r.) used by Adam optimizer in Table 2.

Metrics. We used the number of epochs processed per hour to measure each system's *throughput*. An epoch in DNN training is a traverse of the whole dataset. In Section 6.3, we used the number of data items processed per hour to measure each system's throughput because a model may be early-stopped before finishing one complete epoch.

We defined the *ideal throughput* as the training throughput supposing the system is running on GPUs with unlimited physical memory (also defined in other systems [18]), and the stage partition of the DNN model can seamlessly remain balanced. Same as previous work [18], we implemented the ideal throughput by directly reusing the GPU memory when out-of-memory exceptions were triggered.

We used *ALU utilization* to indicate the usage of GPU ALUs. We used *GPU memory utilization* and *GPU PCIe utilization* to indicate the GPU memory usage and PCIe bandwidth usage. Specifically, for GPU ALU utilization, we used *effective ALU utilization* to distinguish the effective ALU utilization that contributes to the training process and the wasted ALU utilization that are used for *recompute*.

Our evaluation focuses on the following questions:

Section 6.1: How was vPipe's efficiency on static DNN training, compared with the baseline systems?
Section 6.2: How was vPipe's scalability, compared with the baseline systems?
Section 6.3: How was vPipe's efficiency on dynamic DNN training, compared with the baseline systems?
Section 6.4: How effective were vPipe's runtime algorithms and protocol in Section 4?
Section 6.5: What are the limitations of vPipe?

6.1 Static DNN Training (i.e., NAS disabled)

We first give an overview of how much vPipe improved Pipedream and GPipe on training all DNNs. Fig. 8 shows the training curve that indicates how each model's training score improves as training time increases. Overall, in Fig. 8, to finish the same number of training epochs, vPipe shortened the training time of GPipe and Pipedream by 23.5 and 53.4 percent on average. Thus, within the same training time, vPipe allowed both GPipe and Pipedream to achieve better model fitting quality.

Fig. 9 shows the throughput of each system under the same setting as Fig. 8. These results were comparable to the evaluation results in Pipedream [33] and GPipe [19]. When training the four large DNNs, including BERT, Transformer, AmoebaNet, and GNMT, vPipe improved GPipe and Pipedream's throughput by 30.7 and 109.2 percent. To understand vPipe's improvement on GPipe and Pipedream, we looked into the runtime statistics of all GPUs, shown in Table 3, and per-GPU memory usage and ALU utilization when training Transformer in Figs. 10 and 11.

vPipe improved Pipedream most between the two baseline systems on training complex DNNs. In Pipedream, the front stages easily reached the GPU memory limits, as these stages needed to keep many more copies of activation tensors than the rear stages. For example, in Fig. 10, with Pipedream's default partition on Transformer, stage 0 consumed on average 10.3GB GPU memory, as itneeded to hold 8 copies of activation tensors, almost hitting the memory limit (11GB) of GPU 0; and stage 7 consumed only 4.8GB GPU memory, less than half of a GPU's capacity. When training Transformer with 8 GPUs, Pipedream only supported a batch size of 32, and this moderate batch size failed to fully utilize the GPU ALU units, making all GPUs' ALU utilization only 42.3 percent in Fig. 10.

Compared with Pipedream, on training four complex DNNs, Pipedream-vPipe supported 3.75x larger batch size and incurred 2.09x effective ALU utilization (Table 3). To accelerate Pipedream, vPipe alleviated the memory burdens of the front stages by *swap* and *recompute* and rebalanced the stages by repartition. In Fig. 10, vPipe made more *swap* and *recompute* operations on the front stages to reduce the memory burden. However, as the front stages incurred more computation overhead to reduce memory, the front stages took longer execution time, and execution time among stages was unbalanced.

In Fig. 10, when vPipe only enabled the *swap* and *recompute* optimization on each local stage (i.e., Pipedream-vPipe-SR, denoted as P-V-SR), we observed that although stage 0-3 had high total ALU utilization (87.6 percent-95.3 percent), stage 4-7 incurred low ALU utilization of only (61.4-81.7 percent). To make the pipeline more balanced, in vPipe's Algorithm 1, vPipe iteratively performed stage repartition that migrated layers from the front stages to the rear stages. This made the stage 4-7' ALU utilization high (89.7-95.6 percent) and further improved the pipeline's throughput.

vPipe's optimization space on GPipe was GPipe's overhead of an extra forward pass; in our study, an extra forward pass took 23.8-36.5 percent wasted computation on various DNNs [6], [33], [39], [45], [52] and training settings. When training complex DNNs on a large number of GPUs (>8), GPipe achieved better training efficiency than Pipedream because as shown in Table 3, although GPipe needed to process an extra forward pass, compared with Pipedream, GPipe supported 3.75x training batch size and incurred 1.59x total effective ALU utilization on all GPUs. Thus, vPipe had more improvement space on Pipedream.

Compared with GPipe, GPipe-vPipe used 73.2 percent less wasted GPU ALU utilization. The reason is that GPipe-vPipe invoked *swap* and provided a dynamic and efficient strategy to reduce GPipe's *recompute* overhead at runtime (Algorithm 2). In exchange, GPipe-vPipe used 7.9x more PCIe resource than GPipe for swapping. The PCIe resource was usually spare in GPipe's default setting except when

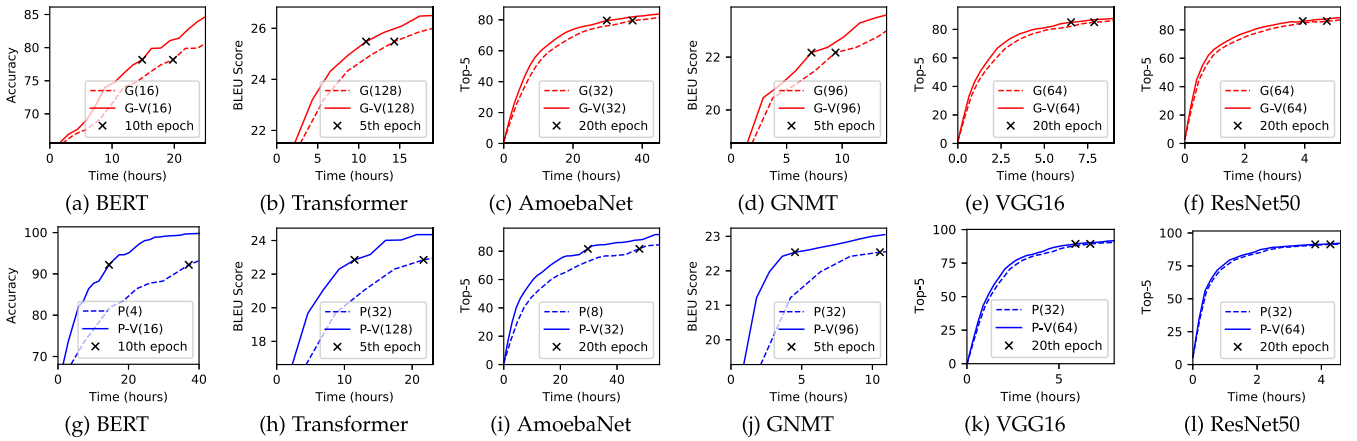

Fig. 8. Model fitting score versus time for training six models using 8 GPUs. For a-f, the models are training with GPipe (**G**) and GPipe-vPipe (**G+V**). For g-l, the models are training with Pipedream (**P**) and Pipedream-vPipe (**P+V**). For BERT, the score metric is next sentence prediction accuracy [10]. For Transformer and GNMT, the score metric is BLEU [58]. For AmoebaNet, VGG16, and ResNet 50, the score metric is top-5 accuracy [15], [33], [39], [44].

network communications was invoked, vPipe tackled the PCIe interference between *swap* and network communication in Section 4.2. Moreover, when NAS was enabled, vPipe improved GPipe by up to 291.3 percent, and we discuss it in Section 6.3.

When training "small" DNNs VGG16 and ResNet50, vPipe improved Pipedream and GPipe by merely 5.2 and 7.3 percent on average. The reason is that when we trained the VGG16 and ResNet50, following the setting of Pipedream [33], we partitioned both VGG16 and ResNet50 into two stages: a stage that contained convolution layers and a stage that contained fully connected layers. We used 7 GPUs to perform data parallelism on the former stage and uses 1 GPU to train the latter one. This two-stage setting limited the optimization space of vPipe's SRP algorithm.

We also evaluated the *ideal throughput* of GPipe and Pipedream, and both Pipedream-vPipe and GPipe-vPipe incurred a degradation from the ideal throughput. The reason is that due to limits of GPU memory capacity and PCIe bandwidth, to support sufficient large batch size that made all GPU's ALU units fully utilized, vPipe incurred inevitable *recompute* overhead on the front stage to avoid exceeding GPU physical memory limit (**G1**). In total, as shown in the GPU utilization column of Table 3, vPipe needed 6.7 percent inevitable wasted ALU utilization on average for *recompute*.

Overall, vPipe accelerated both Pipedream and GPipe on various complex DNNs under static training settings. vPipe's improvement stemmed from a higher utilization rate of all GPU resources, including the effective ALU utilization, memory, and PCIe usage.

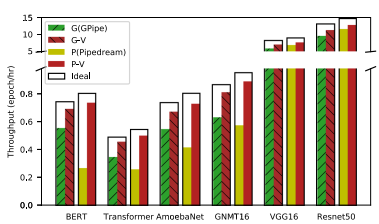

Fig. 9. Throughput of four systems with 8 GPU setting.

6.2 Scalability

To evaluate whether vPipe is scalable to large GPU clusters, we ran Pipedream-vPipe, GPipe-vPipe, Pipedream, and GPipe on different numbers (4-16) of GPU. In addition, an alternative approach to apply dynamic swap and recompute systems (i.e., Capuchin [37]) to distributed settings is to integrate Capuchin to each worker of data parallelism. We also evaluated Capuchin with data parallelism (parameter server) on a different number of GPUs. For pipeline parallelism, the motivation of using larger GPU clusters is often to train larger DNNs [19]. Thus, we made the DNN layer number proportional to the number of involved GPUs (e.g., DNNs used for 16 GPU setting had doubled layers comparing with DNNs used for 8 GPU setting). In Fig. 12, we used the total effective utilization of all GPUs to evaluate the scalability.

Pipedream achieved poor scalability. In pipeline parallelism, the number of simultaneously injected input batches are proportional to the GPU (Stage) number (Section 2.2); as Pipedream directly keeps activation tensors in GPU memory, an increasing GPU number makes the number of activation tensors kept by a single GPU (with a fixed memory) also increased. To avoid exceeding GPU memory limit, Pipedream needed to proportionally decrease the size of each input batch. For example, when training Transformer with 8 GPUs, the batch size supported by Pipedream was 32; when training Transformer with 16 GPUs, the batch size supported by Pipedream dropped to 16.

A larger training batch size can lead to higher GPU ALU utilization [60]; however, in the settings of Fig. 12, the batch size supported by Pipedream were often not high enough to fully utilize a GPU's ALU units. Therefore, when more GPUs were involved in Pipedream, the total effective ALU utilization increased little and even dropped when training AmoebaNet, as the batch size dropped to a very low number (e.g., 1 when training with 16 GPUs) and the parallel utilization of ALUs on all GPUs dropped significantly.

Compared with Pipedream, Pipedream-vPipe, GPipe-vPipe, and GPipe did not suffer from batch size degradation when more GPUs were involved. GPipe used $all - recompute$ strategy without keeping any activation tensors in GPU memory,

TABLE 3
Resource Consumption, Final Fitting Scores, and Micro Events of Training Four Large DNNs With Four Systems on 8 GPUs

Model		Sco.	Bat.	GPU	Mem.	PCIe	fwd	bwd		Sco.	Bat.	GPU	Mem.	PCIe	fwd	bwd
BE.	G+V	96.7%	16	6.4x/6.9x	6.8x	6.0x	0.45	0.76	P+V	98.1%	16	7.0x/7.6x	7.0x	6.3x	0.45	0.75
	G	96.8%	16	4.6x/6.8x	4.2x	0.8x	0.44	1.15	P	98.0%	4	2.7x/2.7x	5.5x	0.5x	0.29	0.48
TR.	G+V	26.4	128	6.3x/6.7x	6.8x	5.9x	0.51	0.96	P+V	24.3	128	6.9x/7.4x	6.9x	6x	0.53	0.97
	G	26.4	128	4.8x/6.6x	4.4x	0.8x	0.52	1.40	P	24.2	32	3.4x/3.4x	4.7x	0.4x	0.31	0.54
AM.	G+V	80.9%	32	6.0x/6.5x	7.1x	7.0x	1.05	2.03	P+V	83.4%	32	6.6x/7.3x	7.4x	7.3x	1.06	2.07
	G	80.8%	32	4.7x/6.6x	3.9x	1.0x	1.02	2.82	P	83.4%	8	2.9x/2.9x	6.3x	1.1x	0.51	0.97
GN.	G+V	24.3	96	5.8x/6.2x	5.8x	5.6x	2.39	4.59	P+V	23.1	96	6.5x/6.9x	6.1x	5.9x	2.38	4.62
	G	24.3	96	4.5x/6.3x	3.9x	0.4x	2.37	6.58	P	23.2	32	2.5x/2.5x	3.9x	0.3x	1.91	3.42

BE. *is BERT.* ***TR.*** *is Transformer.* ***AM.*** *is AmoebaNet.* ***GN.*** *is GNMT.* ***Sco.*** *is the final model fitting score when the training finishes, and score metric of each model is the same as Fig. 8.* ***Bat.*** *is training batch size.* ***GPU*** *is all GPUs' effective/total ALU utilization.* ***Fwd*** *and* ***bwd*** *mean forward pass time and backward pass time of each training iteration.*

and thus supported a sufficiently large batch size to fully utilize a GPU's ALU units. With vPipe integrated, Pipedream-vPipe and GPipe-vPipe supported the same large batch size as GPipe, while vPipe reduced the *recompute* overhead in GPipe. Thus, Pipedream-vPipe and GPipe-vPipe were as scalable as GPipe and achieved better total effective utilization than GPipe.

For both vanilla data parallelism (DP) and Capuchin with data parallelism (DP-C), the scalability was poor because for complex DNNs, the network communication cost for parameter synchronization was the major bottleneck (Section 2). However, DP-C still incurred better effective ALU utilization as Capuchin used *swap* and *recompute* to enlarge the training batch size supported by each GPU, making a high ALU utilization on each GPU worker.

To sum, with vPipe, both BSP (GPipe-vPipe) and ASP (Pipedream-vPipe) systems achieved almost linear scalability that is comparable to the scalable pipeline parallelism system GPipe, while vPipe achieved better total effective GPU utilization. These results indicate that vPipe is both efficient and scalable. As the emergence of more giant DNNs can be foreseen [6], the design of vPipe is able to remain efficient when more and more GPUs are involved.

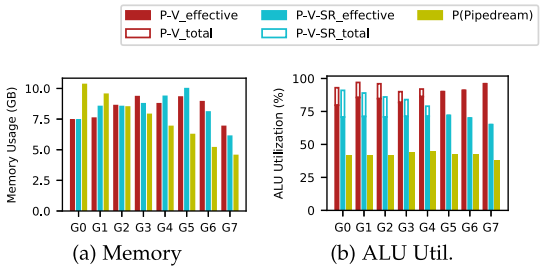

Fig. 10. Resource usage of each GPU when training (NAS-disabled) Transformer with Pipedream, Pipedream-vPipe, Pipedream-vPipe-SR on 8 GPUs. Unfilled bars are wasted GPU ALU utilization for *recompute*.

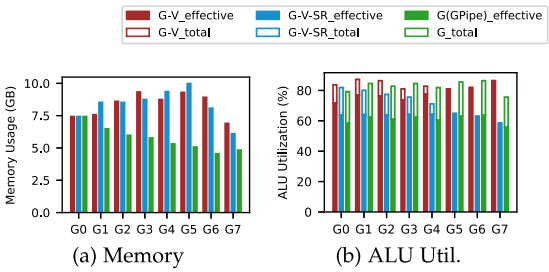

Fig. 11. Resource usage of each GPU when training (NAS-disabled) Transformer with GPipe, GPipe-vPipe, GPipe-vPipe-SR on 8 GPUs. Unfilled bars are wasted GPU ALU utilization for *recompute*.

6.3 Dynamic DNN Training (i.e., NAS Enabled)

To evaluate vPipe's efficiency on dynamic training workload, we conducted a case study of how vPipe performed on neural architecture search (NAS), one of the most prevalent dynamic training processes. We selected two models (Transformer [52] and AmoebaNet [39]) that have been pervasively used for neural architecture search. For both Transformer and AmoebaNet, we implemented the NAS process according to their published description [45] of an evolution algorithm: it creates a set of population DNN models, which have a similar architecture, and train them on a subset (around 1000 data entries) of their Dataset to fast eliminate those unqualified models. This elimination process often took the most time during a NAS process. To ensure fair evaluation, we made the evolution algorithm deterministic: i.e., for each NAS process, the population of models was trained in a determined sequence.

Overall, vPipe accelerated both GPipe and Pipedream on these two NAS-enabled DNN training by 245.4-291.3 percent and 421.3-463.4 percent, while vPipe made no impacts on the upper evolutionary algorithm and did not downgrade the quality of NAS.

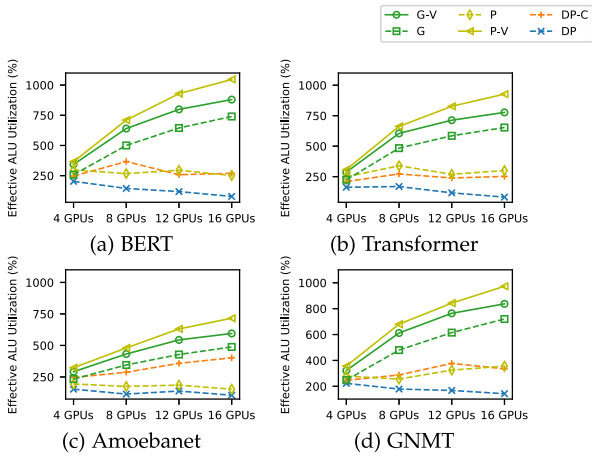

Fig. 12. Scalability. **DP** means pure data parallelism. **DP-C** means data parallelism + Capuchin [37].

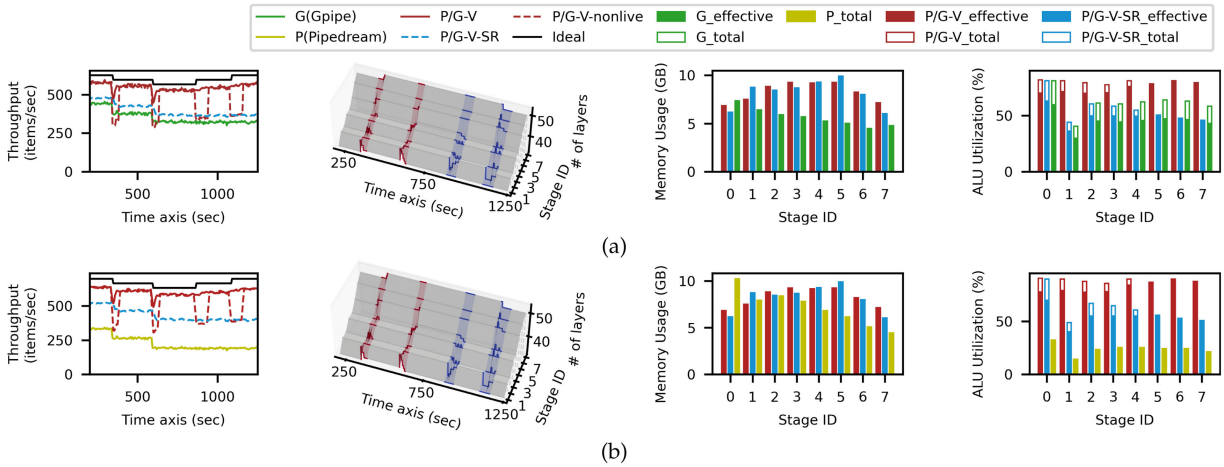

Fig. 13. Training profiling under dynamic training processes (Evolved Transformer). **V-SR** means vPipe with swap/recompute enabled and repartition disabled. In all sub-figures of (a) and (b), the 1st is training throughput collected at every input batch finished; the 2nd is real-time layer number of each stage (**red** means layer increase; **blue** means layer decrease); the 3rd and 4th are the resource utilizations of all GPUs at the end of each sub-figure's time axis.

We selected a snippet for each NAS-enabled model (Transformer and AmoebaNet) training on two baseline systems (Pipedream and GPipe), and Fig. 13 shows how vPipe improved both two systems on NAS-enabled model training. In Figs. 13a and 13b, 8 layers were added twice at 342s and 594s on the first stage, and 8 layers were deleted twice at 880s and 1123s on the second stage. In Figs. 14b and 14b, 46 layers were deleted twice at 921s and 1157s on the first stage, and 46 layers were added twice at 1265s and 1483s on the second stage.

For vanilla baseline systems without vPipe (Pipedream and GPipe), the static partition strategy used by both systems did not cope with this training dynamicity: taken the Transformer in Fig. 13a and 13b as an example, when layers were added on the first stage, both systems incurred a performance drop as the execution time of stage 0 suddenly increased, bottlenecking the whole pipeline; when layers were deleted on the second stage, the whole pipeline's throughput did not increase as the stage 0 was still the throughput bottleneck. In Figs. 13a and 13b, although the ALU utilization of stage 0 was high, other stages all incurred a low ALU utilization as these stages often needed to wait for the execution of stage 0.

When only vPipe's local *swap* and *recompute* optimization (Algorithm 2) on each stage (i.e., vPipe-SR) was enabled, although vPipe-SR improved the two baseline systems' throughput by enlarging the supported batch size (for Pipedream) or reducing the recompute overhead (for GPipe), vPipe-SR was also not able to cope with this training dynamicity. This implies that existing single GPU *swap* and *recompute* systems (e.g., Capuchin [37]) are not sufficient to achieved efficient pipeline parallelism in two folds: first, these systems do not support distributed memory management (Section 4.2); second, even if a distributed *swap* and *recompute* system (e.g., vPipe-SR) exists, it still incurs sub-optimal training efficiency.

In contrast, when vPipe with a full implementation of Algorithm 1 was integrated into Pipedream and GPipe, under training dynamicity, both systems (Pipedream-vPipe and GPipe-vPipe) adjusted its layer distribution on all stages to achieve a near-optimal training throughput. In Fig. 13, the second figure of each sub-figure shows how vPipe adjusted the layer distribution when layer activation/deactivation was suddenly triggered during a training process. For example, when layers were added on stage 0 at the

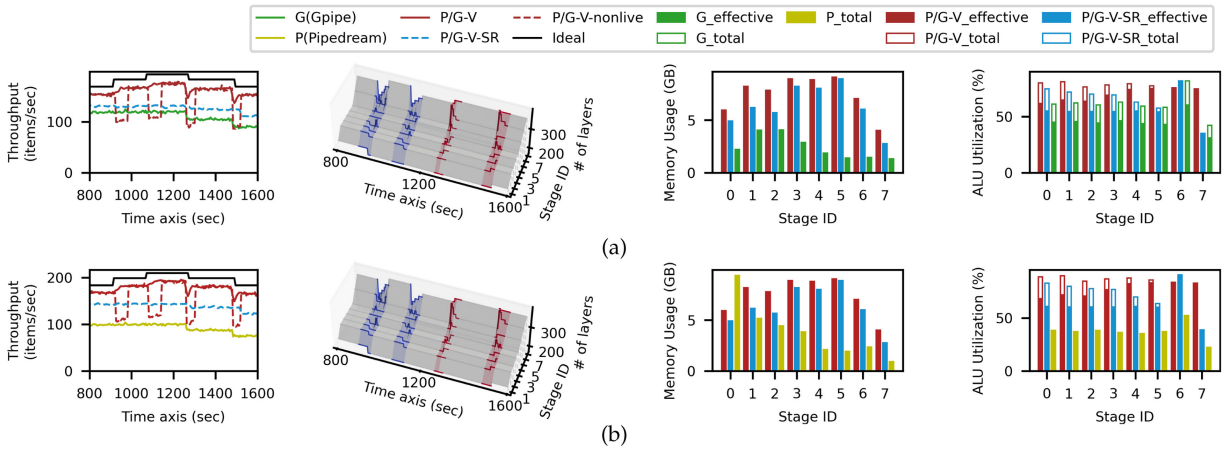

Fig. 14. Training profiling under dynamic training processes (AmoebaNet) with the same setting in Fig. 13.

TABLE 4
Performance of vPipe's Partition Algorithm versus Kernighan-Lin Algorithm [50]

Models	K-L			vPipe		
	O. G(N, E)	time(iter)	cost	C. G(N, E)	time(iter)	cost
BERT	(976,1262)	23.70s (5)	0.37	(124, 137)	2.15s (5)	0.37
Trans.	(662, 830)	21.18s (6)	0.74	(108, 146)	1.74s (6)	0.74
Amoe.	(4380, 5080)	61.82s (4)	5.64	(132, 142)	1.84s (4)	5.64
GNMT	(190, 228)	3.71s (5)	0.71	(46, 52)	0.75s (5)	0.71

O. G means the original graph with N layers and E edges. C. G means coarsened graph. Cost means the network communication cost caused by the partition algorithm (1^{e7} bytes per training sample). Each DNN models used is for 16 GPU training, and the algorithms partition each DNN into 16 stages.

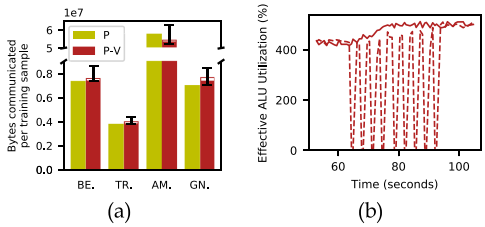

Fig. 15. (a) Network usage of Pipedream with and without vPipe. vPipe's network usage contains vPipe's network overhead (in unfilled red bars) including layer migration and control message costs. (b) Real time GPU ALU utilization statistics with vPipe's live migration and the non-live migration approach.

342s in Figs. 13a and 13b, vPipe's global planner collected the runtime statistics of all stages and noticed an imbalance of execution time among stages. vPipe then triggered Algorithm 3 to generate a new balanced partition. vPipe's layer manager immediately started to migrate layers from stage 0 to the subsequential stages (i.e., stage 3, 5, and 6). Then, vPipe's layer manager locally performed Algorithm 2 to find an optimized local memory management plan. After that, as described in Algorithm 1, vPipe iteratively performed Algorithm 3 and Algorithm 2 until no better SRP strategy was found.

In our evaluation, each iterative process of Algorithm 1 finished within 3-9 iterations (Section 6.1) without performance downgrade thanks to vPipe's fast SRP algorithm and live layer migration protocol. We will further discuss this in Section 6.3. We also evaluated the ideal throughput in Fig. 13, and vPipe incurred a degradation from the ideal throughput for the same reason as we discussed in Section 6.1.

To sum, with vPipe, both Pipedream-vPipe and GPipe-vPipe transparently changed their layer distribution along with the training dynamicity; and by doing so, both systems kept their training throughput close to the ideal throughput during an extremely dynamic training. Both forward and backward layer migrations were triggered frequently during a NAS training process, making both vPipe's forward and backward layer migration designs desirable.

6.4 Effectiveness of vPipe's Algorithms

Effectiveness of vPipe *'s SRP Algorithm.* vPipe's SRP algorithm (Algorithm 1) is a decomposition method that iteratively optimizes two sub-problems: a local search of swap and recompute (Algorithm 2); and a global search of stage partition (Algorithm 3).

We first summarize how vPipe's SRP algorithm improved the baseline systems. For both static training processes (Section 6.1) and dynamic training processes (Section 6.3), vPipe made the training throughput of both Pipedream and GPipe always close to ideal throughput; vPipe's throughput degradation from the ideal throughput was caused by the inevitable *recompute* overhead to make all GPU's total effective ALU utilization high (e.g., Figs. 10 and 11). From Table 3, compared with bare-metal baseline systems Pipedream and GPipe, vPipe's SRP algorithm essentially well utilized *all* available resources of *all* GPUs.

We then examined how fast vPipe's SRP algorithm was. Overall, each invoking of SRP algorithm finished within 10 iterations. The major time cost of each iteration is taken by the graph partition sub-algorithm (Algorithm 3), which solves the NP-hard graph partitioning problem (Section 4.1). In Table 4, we compared the runtime cost of vPipe's partition algorithm (Algorithm 3) with the original KL-algorithm [50] on partitioning four complex DNNs. The results show that vPipe speeded up the KL algorithm by 4x-32x. The reason is that vPipe's coarsen step greatly reduced the complexity of the graph used in the partitioning (Section 4.2). On average, vPipe reduced the number of graph nodes by 3x-32x and the number of graph edges by 3x-35x. This time cost is negligible comparing with the training time. The final edge cuts (i.e., total network communication costs across partitions) produced by vPipe and KL algorithm were equal, as vPipe used KL-refinement to ensure that no better partition on the original graph was missed.

In Fig. 15a, we collected the network communication costs of Pipedream-vPipe and Pipedream using the same setting in Fig. 9. Overall, Pipedream-vPipe achieved comparable network communication costs with Pipedream when training the four complex DNNs. vPipe's layer migration costs and control message costs incurred little overhead as the these costs were amortized over the long training time (up to hundreds of hours). During a layer migration process, vPipe's peak data transfer rate was about 432MB/s, far from blocking both the network connection and the PCIe connection across stages.

To sum, these results indicate that vPipe's SRP algorithm is both fast converging and can achieve a near-optimal plan that well utilizes all GPU resources to achieve efficient pipeline parallel training.

Effectiveness of Live Layer Migration Protocol. vPipe's live layer migration protocol 4.3 transparently migrates a layer to realize a new partition without degrading the training throughput. This guarantees that vPipe can iteratively search for a better SRP plan (Section 4.2) with a negligible training performance penalty.

To examine the necessity of vPipe's live layer migration protocol, we compared it with a non-live layer migration approach (Section 4.2): stop injecting new input batches for the upper system, clean up the pipeline, manually migrate the layer to a new stage, and reboot a new pipeline. In Fig. 13, the red dashed line is the training throughput using a non-live layer migration. The non-live migration degraded the training throughput by up to 60.3 percent because, in each iteration of vPipe's Algorithm 1, a repartition would be triggered, and the pipeline would be cleaned up. Fig. 15b shows the real-time ALU utilization comparison between vPipe's live migration approach and the non-live migration approach, during an iterative Algorithm 1 that triggers 9 stage repartition. In each repartition, the total ALU utilization dropped to zero as the

pipeline was clean up. In comparison, vPipe live-migrated a layer without notable throughput degradation and GPU stall.

6.5 Discussions

vPipe has two limitations. First, vPipe assumes that for any DNN workload trained with vPipe, a single layer fits within the memory limits of a single GPU. This is also assumed by other pipeline parallel systems (e.g., Pipedream and GPipe). In reality, for all recent complex DNNs evaluated by vPipe, the layers can all fit in a single GPU. Second, vPipe's layer migration protocol (Section 4.3) remains live when the time cost of transferring a layer's tensors can overlap with the computation time of DNN training. There might exist special DNNs where the execution time of all layers is extremely short, while a layer holds a non-negligible amount of data to transfer. In all the models we studied and literature, DNNs are both computation intensive and memory intensive [18], [37], making vPipe's off-the-critical-path data transfer realizable, verified in Section 6.4.

In future work, we envision three applications of vPipe. First, vPipe has the unique strength to support more dynamic training paradigms (e.g., DyNet [34]) other than NAS, as DyNet enabled dynamic DNNs (e.g., LSTM [31]) are prevalent and powerful in handling input data with varying lengths (e.g., sentences). Second, existing NAS algorithms produce DNN evolvement with the assumption that GPU memory is unlimited. However, when these NAS algorithms are deployed with pipeline parallelism, they may produce DNN evolvements that cannot be realized with pipeline parallelism, leading to poor search quality. Leveraging vPipe's pipeline statistics, researchers can let NAS algorithms be aware of the underlying pipeline resources, making NAS both highly accurate and feasible under limited hardware resources. Third, as DNNs today are deployed with various training framework, in addition to PyTorch, vPipe can also augment other imperative training engines (e.g., MxNet [8] and Tensorflow [1]).

7 Related Work

Data Parallel Systems. Data parallelism [28] has been widely adopted in DNN training to support large batch size training. In data parallelism, inputs are partitioned across workers. Each worker maintains a local copy of the model parameters and trains on its own partition of inputs while periodically synchronizing weights with other workers. Typical data parallelism systems assume that a DNN model can fit into a single GPU. Nevertheless, the size of recent DNNs has grown far beyond a single GPU's capacity, driving researchers to conduct studies [19], [21] on model parallelism. To support large DNN training with data parallelism, DeepSpeed [38] partitions a DNN's status of parameters and optimizers to each worker, and on-demand transfers the status during the training. DeepSpeed [38] reported a 1.5x network communication volume compared with a typical data parallel system (e.g., Parameter Server). Compared with data parallelism, pipeline parallelism (e.g., vPipe) incurs much less network communication volume [19], [33] and better scalability during large DNN training [19] (see Section 6.2). Overall, data parallelism is complementary to pipeline parallelism systems and can be integrated to vPipe as mixed parallelism to support large batch size training.

Pipeline Parallel Systems. Pipeline (model) parallelism is a special type of model parallel system. Model parallel systems are designed to train complex DNN models that cannot fit into a single GPU's memory. Despite Pipedream [33] and GPipe [19], there are many successive pipeline parallel systems that try to address Pipedream's parameter staleness problem. XPipe [14] uses parameter prediction to mitigate the staleness issues incurred by the ASP pipeline parallel systems (i.e., Pipedream). XPipe directly keeps the activation memories in GPU and have the same performance model as Pipedream. PipeMare [59] adopts the GPipe's all recompute strategy to ASP systems and has a similar model to GPipe's performance and memory. However, PipeMare shares the same limitations as GPipe.

Hybrid Parallel Systems. Existing pipeline parallel systems [14], [19], [33], [59] assume that GPU resource consumptions of layers are roughly evenly distributed. In most recent large DNNs like Transformer [52], BERT [10], GPT-3 [6], AmoebaNet [39], DNN layers are usually homogenous and even in training resource consumption. Nevertheless, in some DNNs like ResNet50 [15] and VGG16 [44], convolution layers usually take much more computation time than the fully connected layers. Hybrid parallelism systems, including OWT [26], FlexFlow [30], etc, are designed to improve the training efficiency of such heterogenous DNNs. Specifically, these systems apply data parallelism to convolution layers and apply model parallelism to fully connected layers. These systems are orthogonal to vPipe, and we leave the support of hybrid parallelism as vPipe's future work.

Training Memory Reduction. DNN training is memory intensive. Training memory reduction has been widely studied by existing work [18], [37]. Existing memory reduction approaches mainly fall into two categories: transparent approaches including swap [18] and recompute [37] that do not affect the training accuracy; and opaque approaches such as low precision training [20] and mixed-precision training that trade-off training accuracy with training memory. vPipe aims to act as a transparent layer so that vPipe's memory reduction will not affect the upper systems. Thus, opaque memory reduction approaches are orthogonal to vPipe. There are many transparent memory reduction systems that are designed for single GPU training. vDNN [40] and SwapAdvisor [18] focus only on swap. SuperNeuron [56] and Capuchin [37] coherently combine swap and recompute to dynamically reduce the memory consumption of DNN training on a single GPU. However, these single GPU systems are not designed to cope with challenges stemming from pipeline parallelism (Section 2). A recent study [54] partially offloads the recompute overhead to the CPU processors. This work is complementary to vPipe and can be integrated into vPipe to further reduce the recompute overhead.

Nvidia proposes Unified Memory [51], a general unified memory address space accessible from both CPU and GPU, so that a process can allocate a memory space larger than a GPU's physical capacity. Nvidia Zero-Copy [61] allows integrated GPU (GPU and CPU physically share memory devices, common in mobile devices) to directly access pinned memory on CPU. VPipe focuses on discrete GPUs (GPU has its own memory devices) in data centers. If a training process exceeds a GPU's physical capacity, Unified Memory automatically migrates tensors (e.g., activations) from GPU to CPU. When

these tensors are accessed later by the GPU ALUs, Unified Memory page fault is triggered, and tensors needed are synchronously moved back from CPU to GPU. Such per-host on-demand moving back significantly blocks a Deep Learning application's execution (e.g., Unified Memory can slow down a DNN's execution by more than 1x [5]). Compared with Unified Memory, vPipe's distributed runtime (Section 4.2) enables vPipe to predict when tensors in CPU will be needed and asynchronously pre-fetches these tensors back to GPU before they are accessed, which prevents blocking the normal execution; vPipe's async swap has an overall negligible overhead on the training performance (Section 4.2). Besides swap, vPipe's distributed memory management also contains features like recompute and migrate.

8 Conclusion

In this paper, we present vPipe, the first dynamic memory and layer partition management system for pipelined parallelism, acting as a virtualized layer between a typical pipeline parallel system and its underlying execution engine. vPipe can accelerate existing pipeline parallel systems under both static and dynamic training of complex DNNs, making them both efficient and scalable. vPipe's source code is released at: github.com/hku-systems/vpipe.

Acknowledgments

The authors would like to thank all reviewers for their valuable comments. This work was funded in part by Huawei Innovation Research Program (HIRP) Flagship, under Grants HK RGC ECS 27200916, HK RGC GRF 17207117, 17202318, and 27208720, in part by Croucher Innovation Award, in part by National NSF China under Grant 61802358, and in part by the USTC Research Funds of Double First-Class Initiative, under Grant YD2150002006. Shixiong Zhao and Fanxin Li contributed equally to this work.

References

[1] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.
[2] S. Areibi, "An integrated genetic algorithm with dynamic hill climbing for VLSI circuit partitioning," in *Proc. Genet. Evol. Comput. Conf.*, 2000, pp. 97–102.
[3] S. Areibi and A. Vannelli, "Distributed advanced search techniques for circuit partitioning," in *Proc. IEEE Can. Conf. Elect. Comput. Eng.*, 1998, pp. 553–556.
[4] PyTorch cuda streams. Accessed: Nov. 12, 2020. [Online]. Available: https://pytorch.org/docs/stable/notes/cuda.html#cuda-streams
[5] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "PipeSwitch: Fast pipelined context switching for deep learning applications," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 499–514.
[6] T. B. Brown *et al.*, "Language models are few-shot learners," 2020, *arXiv:2005.14165*.
[7] T. N. Bui and C. Jones, "A heuristic for reducing fill-in in sparse matrix factorization," Soc. Ind. Appl. Math., Philadelphia, PA, USA, *Tech. Rep.*, 1993.
[8] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*.
[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.
[11] S. Dutt, "New faster Kernighan-Lin-type graph-partitioning algorithms," in *Proc. Int. Conf. Comput. Aided Des.*, 1993, pp. 370–377.
[12] C. Farhat, E. Wilson, and G. Powell, "Solution of finite element systems on concurrent processing computers," *Eng. Comput.*, vol. 2, no. 3, pp. 157–165, 1987.
[13] P.-O. Fjällström, *Algorithms for Graph Partitioning: A Survey*, vol. 3. Linköping, Sweden: Linköping University, Electronic Press, 1998.
[14] L. Guan, W. Yin, D. Li, and X. Lu, "XPipe: Efficient pipeline model parallelism for multi-GPU DNN training," 2019, *arXiv:1911.04610*.
[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[16] M. T. Heath and P. Raghavan, "A cartesian parallel nested dissection algorithm," *SIAM J. Matrix Anal. Appl.*, vol. 16, no. 1, pp. 235–253, 1995.
[17] B. Hendrickson and R. Leland, "A multi-level algorithm for partitioning graphs," in *Proc. ACM/IEEE Conf. Supercomputing*, 1995, pp. 28–es.
[18] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 1341–1355.
[19] Y. Huang *et al.*, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 103–112.
[20] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, 2017.
[21] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," 2018, *arXiv:1807.05358*.
[22] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," *ACM SIGARCH Comput. Architect. News*, vol. 47, no. 4, pp. 223–234, 2012.
[23] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in *Proc. Conf. ICPP (3)*, 1995, pp. 113–122.
[24] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *Proc. ACM/IEEE Conf. Supercomputing*, 1995, pp. 29–es.
[25] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, Feb. 1970.
[26] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014, *arXiv:1404.5997*.
[27] C.-H. Lee, M. Kim, and C. I. Park, "An efficient k-way graph partitioning algorithm for task allocation in parallel computing systems," in *Proc. 1st Int. Conf. Syst. Integration*, 1990, pp. 748–751.
[28] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 583–598.
[29] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi,"A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.
[30] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Architect.*, 2017, pp. 553–564.
[31] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing LSTM language models," 2017, *arXiv:1708.02182*.
[32] J. M. Mulvey and A. Ruszczyński, "A new scenario decomposition method for large-scale stochastic optimization," *Operations Res.*, vol. 43, no. 3, pp. 477–490, 1995.
[33] D. Narayanan *et al.*, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.
[34] G. Neubig *et al.*, "DyNet: The dynamic neural network toolkit," 2017, *arXiv:1701.03980*.
[35] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen, "Hermes: Dynamic partitioning for distributed social network graph databases," in *Proc. 18th Int. Conf. Extending Database Technol.*, 2015, pp. 25–36.
[36] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.
[37] X. Peng *et al.*, "Capuchin: Tensor-based GPU memory management for deep learning," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 891–905.

[38] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations toward training trillion parameter models," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–16.
[39] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 4780–4789.
[40] 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 *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitect.*, 2016, pp. 1–13.
[41] I. Safro, P. Sanders, and C. Schulz, "Advanced coarsening schemes for graph partitioning," *J. Exp. Algorithmics*, vol. 19, pp. 1–24, 2015.
[42] R. Sennrich, B. Haddow, and A. Birch, "Edinburgh neural machine translation systems for WMT 16," 2016, *arXiv:1606.02891*.
[43] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in tensorflow," 2018, *arXiv:1802.05799*.
[44] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[45] D. R. So, C. Liang, and Q. V. Le, "The evolved transformer," 2019, *arXiv:1901.11117*.
[46] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," 2019, *arXiv:1906.02243*.
[47] Y. Sun, M. Kirley, and S. K. Halgamuge, "A recursive decomposition method for large scale continuous optimization," *IEEE Trans. Evol. Comput.*, vol. 22, no. 5, pp. 647–661, Oct. 2018.
[48] S. H. Teng and P. Spheres, "Unified geometric approach to graph separators," in *Proc. 31st Ann. Symp. Foundations Comput. Sci.*, 1991, pp. 538–547.
[49] F. Teraoka, Y. Yokore, and M. Tokoro, "A network architecture providing host migration transparency," in *Proc. Conf. Commun. Architecture Protoc.*, 1991, pp. 209–220.
[50] J. L. Träff, "Direct graph k-partitioning with a Kernighan–Lin like heuristic," *Operations Res. Lett.*, vol. 34, no. 6, pp. 621–629, 2006.
[51] Nvidia unified memory. Accessed: Mar. 21, 2021. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/
[52] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[53] S. Venugopalan, M. Rohrbach, J. Donahue, R. Mooney, T. Darrell, and K. Saenko, "Sequence to sequence – Video to text," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 4534–4542.
[54] M. Wahib *et al.*, "Scaling distributed deep learning workloads beyond the memory capacity with KARMA," 2020, *arXiv:2008.11421*.
[55] L. Wang, S. Xie, T. Li, R. Fonseca, and Y. Tian, "Sample-efficient neural architecture search by learning action space," 2019, *arXiv:1906.06832*.
[56] L. Wang *et al.*, "Superneurons: Dynamic GPU memory management for training deep neural networks," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 41–53.
[57] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, and R. Fonseca, "AlphaX: Exploring neural architectures with deep neural networks and Monte Carlo tree search," 2019, *arXiv:1903.11059*.
[58] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.
[59] B. Yang, J. Zhang, J. Li, C. Ré, C. R. Aberger, and C. De Sa, "PipeMare: Asynchronous pipeline parallel DNN training," 2019, *arXiv:1910.05124*.
[60] E. Yang, S.-H. Kim, T.-W. Kim, M. Jeon, S. Park, and C.-H. Youn, "An adaptive batch-orchestration algorithm for the heterogeneous GPU cluster environment in distributed deep learning system," in *Proc. IEEE Int. Conf. Big Data Smart Comput.*, 2018, pp. 725–728.
[61] Nvidia CUDA zero-copy. Accessed: Mar. 21, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#zero-copy
[62] Y. Zhao, L. Wang, Y. Tian, R. Fonseca, and T. Guo, "Few-shot neural architecture search," 2020, *arXiv:2006.06863*.