NASPipe: High Performance and Reproducible Pipeline Parallel Supernet Training via Causal Synchronous Parallelism

Shixiong Zhao
sxzhao@cs.hku.hk
The University of Hong Kong
Hong Kong, China

Fanxin Li
fxli@cs.hku.hk
The University of Hong Kong
Hong Kong, China

Xusheng Chen
xschen@cs.hku.hk
The University of Hong Kong
Hong Kong, China

Tianxiang Shen
txshen2@cs.hku.hk
The University of Hong Kong
Hong Kong, China

Li Chen
chen.li7@huawei.com
Huawei Technologies
Hong Kong, China

Sen Wang
wangsen31@huawei.com
Huawei Technologies
Hong Kong, China

Nicholas Zhang
nicholas.zhang@huawei.com
Huawei Technologies
Hong Kong, China

Cheng Li
chengli7@ustc.edu.cn
USTC
Anhui, China

Heming Cui*
heming@cs.hku.hk
The University of Hong Kong
Hong Kong, China

ABSTRACT

Supernet training, a prevalent and important paradigm in Neural Architecture Search, embeds the whole DNN architecture search space into one monolithic supernet, iteratively activates a subset of the supernet (i.e., a subnet) for fitting each batch of data, and searches a high-quality subnet which meets specific requirements. Although training subnets in parallel on multiple GPUs is desirable for acceleration, there inherently exists a race hazard that concurrent subnets may access the same DNN layers. Existing systems support neither efficiently parallelizing subnets' training executions, nor resolving the race hazard deterministically, leading to unreproducible training procedures and potentiallly non-trivial accuracy loss.

We present NASPipe, the first high-performance and reproducible distributed supernet training system via causal synchronous parallel (CSP) pipeline scheduling abstraction: NASPipe partitions a supernet across GPUs and concurrently executes multiple generated sub-tasks (subnets) in a pipelined manner; meanwhile, it oversees the correlations between the subnets and deterministically resolves any causal dependency caused by subnets' layer sharing. To obtain high performance, NASPipe's CSP scheduler exploits the fact that the larger a supernet spans, the fewer dependencies manifest between chronologically close subnets; therefore, it aggressively schedules the subnets with larger chronological orders into execution, only if they are not causally dependent on unfinished precedent subnets. Moreover, to relieve the excessive GPU memory burden for holding the whole supernet's parameters, NASPipe uses a context switch technique that stashes the whole supernet in CPU memory, precisely predicts the subnets' schedule, and pre-fetches/evicts a subnet before/after its execution. The evaluation shows that NASPipe is the only system that retains supernet training reproducibility, while achieving a comparable and even higher performance (up to 7.8X) compared to three recent pipeline training systems (e.g., GPipe).

CCS CONCEPTS

• **Computer systems organization** → **Pipeline computing**; *Grid computing*; • **Computing methodologies** → *Neural networks*.

KEYWORDS

Neural networks, neural architecture search, automatic machine learning, one-shot, parallel training, distributed training

ACM Reference Format:
Shixiong Zhao, Fanxin Li, Xusheng Chen, Tianxiang Shen, Li Chen, Sen Wang, Nicholas Zhang, Cheng Li, and Heming Cui. 2022. NASPipe: High Performance and Reproducible Pipeline Parallel Supernet Training via Causal Synchronous Parallelism. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507735

*Heming Cui is the corresponding author.

1 INTRODUCTION

Neural Architecture Search (NAS) has achieved transformational impact on building high-quality Deep Neural Networks (DNNs) for various applications [9, 18, 51] and devices [35, 36], and the supernet paradigm is the most widely adopted NAS paradigm due to its low computation cost and high quality. The supernet paradigm composes the whole search space as a monolithic supernet and trains the supernet in rounds. In each round, the paradigm activates a subset of the supernet (i.e., a subnet) directed by an exploration algorithm (e.g., SPOS [9]); then, the paradigm trains the subnet with a moderate batch (e.g., 256 [9]) of data samples as a sub-task for training the supernet.

Compared to traditional NAS paradigms that train each explored DNN to convergence [28, 34], the supernet paradigm [9, 18, 35, 36, 51] vastly reduces the NAS computational cost (e.g., by 500X [9]) from training tens of thousands of *standalone* DNNs to training a *monolithic* one, and at the same time, retains the quality of searched DNNs. The success of this paradigm has not only accelerated the adoption of NAS in the industry [4, 42, 45, 49], but also expedited the adoption of supernet in relevant fields such as dynamic models [15] and mixture-of-experts models [43]. A key determinant to the quality of a NAS-discovered DNN is the size of the search space, because a larger search space embeds more candidate architectures to search from. Therefore, data scientists are actively developing larger and larger search spaces; the largest search space (i.e., Evolved Transformer [34]) used by existing NAS algorithms has already included 15B parameters.

Unfortunately, although the industry and academia have developed systems for easily defining NAS supernets (i.e., Retiarii [45]) and training large standalone models (e.g., GPipe [12], Deepspeed [27], and Pipedream [21]), none of these systems are designed to efficiently train very large supernets. This is due to two major challenges that stem from the frequent switching of subnets.

First, as training each subnet is usually fast, a promising direction for accelerating supernet training is to train subnets in parallel. However, a major challenge stems from deterministically resolving the dependencies between subnets activated in parallel. Specifically, the exploration algorithms assume that the subnets are trained according to their order in an isolated and sequential way: if two subnets active the same layer, the later subnet has a *causal dependency* on the former subnet on this shared layer, thus the later subnet should use the updated layer parameters after the former one's training finishes. For adequate training accuracy and result reproducibility (defined in §2.1), it is crucial to retain the causal order provided by exploration algorithms during parallelization.

However, existing systems for large-scale DNN training are inherently designed for parallelizing the training of multiple batches within the same DNN model, so they are not designed to capture and enforce this causal dependency. For instance, Retiarii [45] adopts the bulk synchronous parallel (BSP) pattern that processes bulks of subnet training in parallel, each on a GPU, and performs parameter updates *in bulk*. This BSP pattern does not maintain the causal dependencies between subnets within a bulk (Figure 1) and cannot guarantee the training reproducibility as evaluated in §5.2.

The second challenge is to efficiently manage the extra-large supernet context among GPUs. To leave more cache space for larger-batch training (thus achieving higher GPU utilization), one has to keep only the activated subnets in GPUs. However, precisely prefetching subnets from CPU to GPU is very challenging because subnets are usually generated by the exploration algorithm at runtime; and the scheduling of parallel subnet executions is also unknown prior to run.

Although existing systems carry efficient DNN operator switching features [3, 11, 48], all these algorithms are designed for training a static DNN. Therefore, existing DNN operator switching designs often assume a pre-known DNN execution so that they [3] can predetermine a schedule that pipelines the DNN context switch

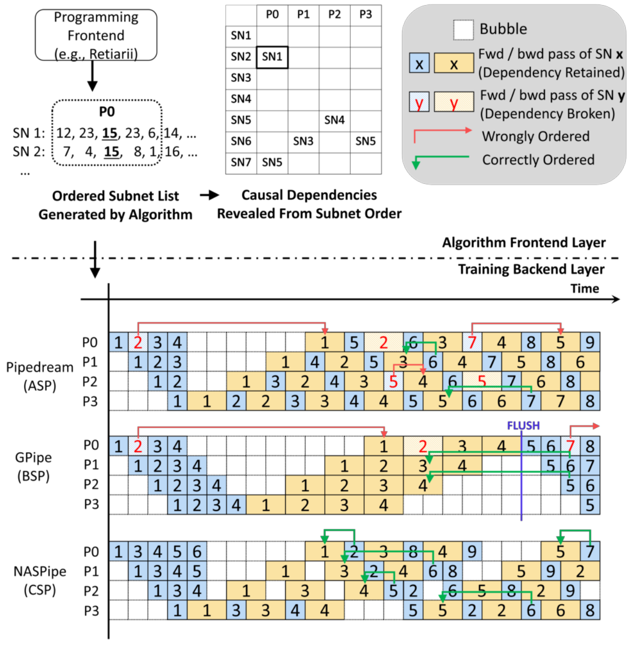

Programming Frontend (e.g., Retiarii)
P0
SN 1: 12, 23, 15, 23, 6, 14, ...
SN 2: 7, 4, 15, 8, 1, 16, ...
Ordered Subnet List Generated by Algorithm
Causal Dependencies Revealed From Subnet Order
P0 P1 P2 P3
SN1 SN2 SN3 SN4 SN5 SN6 SN7
Bubble
Fwd / bwd pass of SN x (Dependency Retained)
Fwd / bwd pass of SN y (Dependency Broken)
Wrongly Ordered
Correctly Ordered
Algorithm Frontend Layer
Training Backend Layer
Time
Pipedream (ASP)
GPipe (BSP)
NASPipe (CSP)
FLUSH

Figure 1: Comparison of ASP, BSP, and CSP pipeline on executing an ordered list of subnets with causal dependencies. A subnet's pipeline partition is causally dependent on a precedent subnet's partition, if they share the same layers. CSP pipeline (i.e., NASPipe) is the only method to retain all dependencies with an adequate pipeline efficiency (reasonable bubble rate).

and the DNN execution to overlap the switch cost. In contrast, in supernet training, the parallelized subnets involve different layers, and they are dynamically switched according to an order generated at runtime.

In this work, we build NASPipe, the first parallel supernet training system that efficiently tackles the aforementioned challenges, via causal synchronous parallel (CSP) pipelining, inspired by conventional CPU instruction pipeline problems [6]. As depicted in Figure 1, NASPipe concurrently executes the subnets generated by supernet-based exploration algorithms, oversees the correlation between subnets, and maintains any causal dependencies caused by layer sharing to enforce high-quality and reproducible supernet training.

To parallelize subnet executions, instead of placing each subnet task on one GPU (like Retiarii [45]), we partition each subnet into stages (i.e., a subset of layers), let each GPU process one stage, and form the subnet executions into a pipeline. This design choice brings two notable benefits. First, pipeline parallelism allows us to efficiently resolve causal dependencies and perform synchronizations in supernet training, *locally* on each GPU in a *decentralized* way. In comparison, Retiarii leverages an external *global* synchronization server, which is *neither scalable nor efficient*. Second, as DNNs are getting larger and larger, a subnet itself has already exceeded a single GPU's capacity, and pipeline parallelism is one of the most efficient approaches for training large models [12, 21, 41].

To improve pipeline efficiency, our *insight* is that, the larger the supernet spans, the fewer dependencies manifest between chronologically close subnet tasks; this allows us to design a pipeline scheduler that advances the subnet tasks with larger chronological order into execution, if they are not causally dependent on unfinished executions of precedent subnets.

To efficiently manage GPU memory and precisely swap in the context of subnets to be executed, we leverage the common insight that DNN computation time on GPUs is roughly deterministic. NASPipe forecasts the upcoming subnets with the highest chance to be scheduled in the next several steps by leveraging the status of the current stage and the status passed from other stages. NASPipe's forecast mechanism, most of the time (~ 90%, see Table 2), avoids an enforced idling to synchronously swap in a layer when the layer residing in CPU memory is requested by execution, with moderate GPU memory overhead.

We have prototyped NASPipe on PyTorch, one of the most well-accepted DNN execution frameworks. NASPipe serves as a training system behind a supernet-based NAS algorithm, which can be described by Retiarii [45] or any other NAS programming frameworks. We compared NASPipe to three recent pipeline training systems: Pipedream [21], GPipe [12], and VPipe [48]. The evaluation shows that:

- NASPipe is reproducible. NASPipe was able to produce the same training process and results, independent of the number of GPUs involved in training, while existing baseline systems cannot achieve this property.
- NASPipe achieves high performance even with causal dependencies among subnets enforced. NASPipe achieved comparable and even higher performance (up to 7.8X) compared to the baselines, while baselines did not enforce causal dependencies.
- NASPipe is scalable. With the number of GPUs increasing, NASPipe was able to provide a roughly linearly increased computation power measured by the total ALU utilization of all GPUs.

Our major contribution is the CSP pipeline scheduling, which maximally parallelizes a supernet training's ordered subnet tasks across GPUs, and deterministically resolves causal dependencies manifesting between concurrently executed subnets. We prototyped NASPipe, a pipeline parallel system geared towards supernet training with a bunch of system optimization techniques, including the CSP pipeline scheduler and the context forecast scheme, collectively making NASPipe the first parallel supernet training system that retains both high performance and reproducibility, an important feature pursued by researchers [42, 44], ML framework developers [1, 24], and hardware manufacturers [1]. We believe that leveraging NASPipe, the NAS communities are not only able to continuously and efficiently explore larger search spaces with more GPUs, but also can easily debug, reproduce, and analyze any supernet training procedures with a simple and deterministic training replay, on any number of GPUs according to their own budget. NASPipe's code is available to the community via github.com/hku-systems/naspipe.

2 BACKGROUND AND MOTIVATION

2.1 Supernet and Causal Dependencies

A typical DNN training process consists of iterations of forward and backward passes on a *standalone* DNN. Recently, a *composed* supernet [9, 18, 26, 35, 45], embedding thousands of DNNs, emerges as an unprecedented training workload. As depicted in Figure 2, a supernet embeds a DNN space, with hierarchical choice blocks of candidate layers (e.g., 1x1 conv, 3x3 conv, or maxpool), into a monolithic supernet; for each training input, a subnet, sampled by a list of layers choices, is activated to fit in the input data. Supernet training has two benefits. First, it embeds an exponential modeling space with its quadratic space complexity for holding execution context, e.g., a NAS supernet with 5 layers and 4 choices per layer can embed 4^5 candidate DNNs. Second, supernet enforces knowledge transferring across subnets by letting all subnets share weights, allowing a one-shot parameterization to a huge population of DNNs.

Leveraging these advantages, supernet is widely adopted and has vastly reduced the computation cost (e.g., by 500x [9]) in the field of Neural Architecture Search [5, 9, 18] (NAS). Specifically, NAS is to search out an optimal DNN architecture, dedicating to a certain application scenario, from an expert-defined search space; supernet connects and trains parameters of the whole search space in a one-shot way. This success in NAS has also expedited the supernet adoption in other emerging fields of Machine Learning such as dynamic slimmable models [15] and a mixture of experts models [43]. In this paper, we take the supernets used in NAS as our major workload of study, because most existing supernets are designed for NAS and have already covered all DNN layer types used in other fields.

However, accelerating a supernet training on parallel GPUs, just like the traditional DNN training task parallelizing (e.g., Parameter Server [16]), is challenging for existing systems. Specifically, a performance hazard stems from the causal dependencies between two subnet executions if they concurrently activate the same layer. NAS algorithms often assume that the subnets are trained according to their order in an isolated and sequential way: if two subnets access the same layer, the latter subnet has a *causal dependency* on the former subnet on this shared layer, and the later subnet should *read* this layer's parameters in its *forward pass* with the update made in the *backward pass* (with optimizer step) of the former subset. For reproducibility, it is crucial to retain all causal dependencies provided by the exploration algorithm, which sabotages parallelism.

Definition 1 (Reproducibility)**.** A supernet training process is *reproducible* if the training result (i.e., parameter weights of all layers) is bitwise equivalent (e.g., in floating-point 32 precision) when the training is repeated with the same dataset and the same random seeds, but potentially on a different cluster.

The reproducibility for supernet training contains two aspects: *intra-subnet reproducibility* and *inter-subnet reproducibility*. Intra-subnet reproducibility comprehends the deterministic computation for GPUs and training framework (given floating-point additions are not commutative), which is easily achievable through existing deterministic libraries [1]. Inter-subnet reproducibility, on the other hand, refers to the deterministic interleaving among reads and

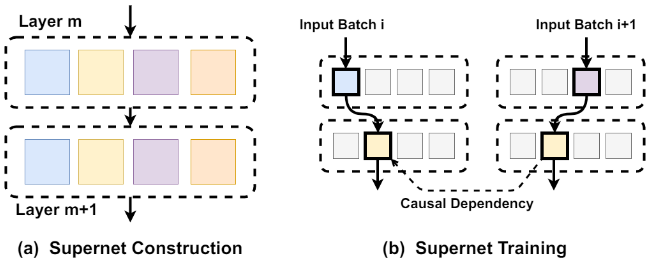


Figure 2: An example of supernet construction and supernet training with causal dependencies. A rounded rectangle with dashed lines means a choice block and colored rectangles mean candidate layers. Bold rectangles in subfigure (b) means activated subnet layers to be trained on each input batch.

writes of subnets' layers, especially when running in parallel, these subnets may read (during forward pass) and write (during backward pass) to intersecting sets of layers. In essence, inter-subnet reproducibility requires that the read-write interleaving among subnets is equivalent to training each subnet sequentially, one at a time, according to the order provided by the exploration algorithm (e.g., SPOS [9]).

Ensuring reproducibility is essential for three reasons. First, reproducibility facilitates knowledge transferability: a supernet training procedure of high quality should be exactly reproduced with different GPU clusters of various research budgets. Second, in NAS studies [42, 44], analysis (debugging) of supernet training procedures plays an important role. When an outstanding trial (with the best hyperparameters and specific random seed) of supernet training is identified from tremendous experimental trials, post-training analysis (e.g., training stability check) is often needed to reason about this trial. For example, the authors of GreedyNAS [42] needed to re-run the identified most outstanding trial and repeatedly inspected the quality-ranking information of subnets. With the training reproducibility, the re-runs are deterministic, including all the collected information (e.g., quality ranking), making supernet training much easier to inspect, analyze, and debug. Third, deterministic execution has constantly been a desirable feature for Machine Learning at different levels. For example, Nvidia launches an individual project [1] to enforce deterministic GPU execution at the ML operator level. NASPipe pursues reproducibility at the NAS supernet training level.

2.2 How to Generate and Parallel the Tasks?

NASPipe parallelizes a supernet's training with inter-subnet parallel task generation and pipeline parallelism.

Parallel Task Generation. Both Retiarii [45] (the notable system designed for supernet programming) and NASPipe advocate an *inter-subnet* [21] parallel task generation, which allows a launch of multiple subnets with each subnet processing one input batch. An alternative approach is the *intra-subnet* task generation [12] (a.k.a., intra-batch parallel in traditional DNN training) which splits a batch into multiple micro-batches, executes each micro-batch on the same subnet on each GPU, and synchronously flushes outputs when all split micro-tasks finish. Intra-subnet task generation is *non-general* as it is only efficient for large batch size training [45] to retain adequate utilization for all GPUs. However, the batch size used in many existing supernet-based algorithms [9, 18, 35] is relatively small and not suitable for intra-subnet task generation; in certain supernet [15, 43], each batch even could only contain one data item. Therefore, in this work, we assume that both NASPipe and all relevant systems are configured with *inter-subnet* parallel task generation.

Parallelism Selection. Different from Retiarii's selected parallelism (wrapped data parallelism), which assigns each parallel GPU one subnet execution and uses an external Parameter Server for synchronization, we embrace a pipeline parallelism design: we partition each subnet into stages (i.e., a subset of layers), let each GPU process one stage, and form parallel subnet executions into a pipeline. This design has two noteworthy advantages over Retiarii's choice. First, pipeline parallelism allows us to efficiently resolve dependencies and perform synchronizations in supernet training, *locally* on each GPU worker in a *decentralized* way. In contrast, Retiarii leverages an external *global* synchronization server, which is *neither scalable nor efficient* when the number of parallel GPUs is large, and subnets synchronize frequently. Second, as DNNs are getting larger and larger, a subnet itself is already beyond a single GPU's capacity. However, large DNNs and their search space are the major targeting workload of this paper, which is suitable for using pipeline parallelism [12, 21, 41]. Therefore, Retiarii's one subnet per GPU design does not apply to most workloads (e.g., Transformer-based NAS) targeted by this work.

2.3 Motivations

Challenge-1. No existing synchronization methods are designed to capture causal dependencies in parallel supernet training. Existing synchronization methods are mainly designed for single DNN training. The most notable one is the Bulk Synchronous Parallel (BSP) method, which inserts a synchronization barrier (flush) after finishing a bulk of parallel tasks. BSP methods have been widely adopted in traditional intra-batch parallel training of single DNNs (e.g., Parameter Server [24], GPipe [12]) to enforce *strong dependencies* (determined by the Stochastic Gradient Descent algorithm [50]) between different training batches. Retiarii also adopts a BSP method to inter-subnet parallel tasks and perform a parameter synchronization when a bulk of subnets is finished. However, BSP for inter-subnet parallel tasks often violates the causal dependencies between subnet within the same bulk, as illustrated in Figure 1. Other synchronization methods include Asynchronous Parallel [21] (ASP) and its variants (e.g., Stale Synchronous Parallel [17]) are not designed to tackle causal dependencies in supernet training.

Challenge-2. It is challenging to efficiently manage the extra-large supernet context and balance the pipeline load among GPUs. Existing work [28, 34] shows that a supernet's context can be much larger than standalone DNN's, and the community is actively building larger supernets for delivering better DNNs. For achieving reasonable GPU utilization, one has to keep only the activated subnets in GPUs. However, efficiently switching subnets between GPU and CPU memory is very challenging because the training of each subnet is usually very fast (e.g., 256 [9] samples), and the exploration schedule is generated by the exploration algorithm at

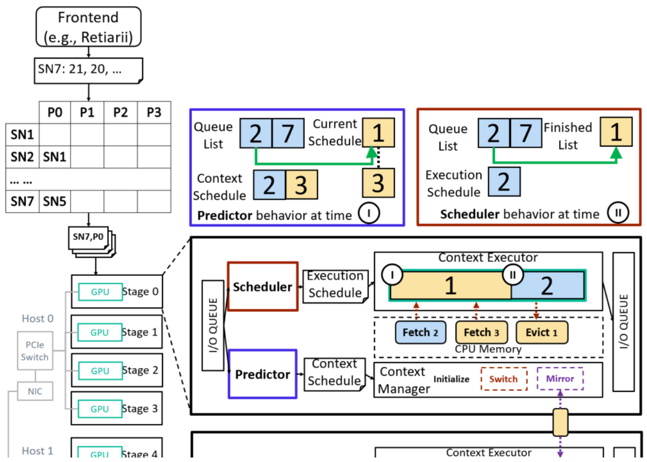


Figure 3: NASPipe's Architecture, details described in §3.1.

runtime. Existing optimizations [11, 22, 25, 30, 48] towards DNN training memory reduction or GPU-CPU memory switching are all not designed for NAS supernet to capture correlations between subnets.

Moreover, existing pipeline parallel systems all adopt a static partition, where operators reside on the assigned GPUs throughout the training. However, considering optimal (balanced) partitions for all subnet execution, an operator often belongs to different stages (GPUs). One approach [48] is to on-demand migrate an operator between stages when it is needed by another subnet's best partition. However, as the subnet switching of a NAS supernet training is often at second-level frequency, this design inevitably incurs high initialization and synchronization costs. Instead of on-demand migration, NASPipe mirrors these operators between stages and eliminates these costs (presented in §4.2).

3 NASPIPE DESIGN

In this section, we first deliver preliminaries used in this work, then formally define CSP, and at last present how NASPipe's design resolved the aforementioned challenges.

Preliminaries. In this work, same as [9, 45], we define a supernet as a sequence of m choice blocks, b_0, b_1, ..., b_m, and each choice block b_x is composed of a set of n candidate layers l_x^0, l_x^1, ..., l_x^i, ... l_x^n. We use SN_0, SN_1, ..., SN_y, where y is the subnet sequence ID, to denote the subnet list, whose sequence order is determined by the upper NAS supernet exploration algorithm. A subnet is an m-sized list of layers, with one layer selected from each choice block. All subnets are generated by a per choice block uniform sampling approach [9], which is the most representative method used in existing supernet practices [45].

During training, a subnet is split into D partitions, and each partition serves as a stage of a pipeline, where D is the number of GPUs; we denote a partition i of a subnet x as P_x^i. Each partition of a subnet has a forward pass (parameter READ) and a backward pass (parameter WRITE). Forward pass and backward pass are separately scheduled in the pipeline, and are defined as *tasks*, which are the minimal units of NASPipe's execution and scheduling.

Definition 2 (CSP). An inter-subnet parallel training process of a subnet is Causal Synchronous Parallel if the process satisfies the following two key properties.

Dependency Preservation. If subnet with sequence ID x (SN_x) and subnet with sequence ID y (SN_y) have the same choice of layer l, and $x < y$, the subnet y's access (READ and WRITE) to layer l must wait until subnet x's WRITE access on this layer finishes.

Concurrent computation and communication. Every parallel worker (e.g., GPU) may perform local computations, i.e., each worker can only make use of values stored in the local memory of the worker. The computations occur asynchronously of all the others but may overlap with communication. The worker exchange data to facilitate remote data storage and transfer.

3.1 Architecture Overview

NASPipe is a distributed pipeline parallel training system that lies behind a supernet programming frontend (e.g., Retiarii), as depicted in Figure 3.

Scheduler observes each candidate task in the forward pass queue and backward pass queue, and checks whether it satisfies the dependency preservation of CSP. If it meets CSP, it will be scheduled to run. Priorities are given according to the subnet sequence ID of each task (lower ID first) and its execution property (backward pass first), as explained in §3.2. For instance, at time point 2 of Figure 3, on stage 0 (P^0), the scheduler schedules SN_2 forward (SN_2 has the lowest ID), as its precedent constraint SN_1 backward has finished (SN_2 meets CSP). Scheduler can enforce dependency while trying best to parallel task executions, making NASPipe achieve adequate parallelism (at most 61% bubble elimination in §5.1).

Predictor forecasts the upcoming tasks (i.e., 2 in our configuration) that have the highest chance to be scheduled in the next several steps. Predictor leverages the status of the current stage and the status passed from other stages to simulate the pipeline run, explained in §3.3. For instance, at time point 1 of Figure 3, on stage 0 (P^0), the predictor predicts that SN_2 forward will be scheduled as the currently scheduled execution SN_1 backward will clear its dependency; the predictor predicts that SN_3 backward will be scheduled leveraging the pipeline status carried with SN_1 backward. We use the metric *cache hit rate* to measure the percentage of times of the event that, when a layer in a choice block is activated, the layer already resides in the GPU memory. Leveraging NASPipe's predictor design, NASPipe is able to achieve a cache hit rate as high as ∼ 90%, at the cost of a cache size ∼ 3X of a single subnet's context.

Context Executor is the main runtime *process* that iteratively fetches context schedule from the CSP scheduler and executes subnets (§3.2). Before execution, it checks whether the subnet context to be executed is ready in GPU for safety. Also, it updates its execution status, shared with the context manager, before and after a scheduled execution finishes. This is to make the context manager aware of the execution status.

Context Manager runs as a separate *process* that shares the supernet context (i.e., DNN modules) with the context executor, enabling totally asynchronous management of the supernet context. According to the context schedule generated by context predictor, NASPipe's context manager initializes new context modules, manages context switch, i.e., evicts finished or interrupted subnet context to CPU memory and pre-fetches the next possible subnet back to GPU, and handle the parameter synchronization for mirrored operators (§4.2).

3.2 Pipeline Scheduler with Dependency Preservation

Scheduling Unit. NASPipe's runtime involves partitioning the layers of a subnet into multiple stages. The partition is a D-partition (D is the number of GPUs) of a subnet's sequential list of layers, with each partition having roughly the same execution time, according to pre-profiled statistics of each layer. Each stage is mapped to a separate GPU (a runtime worker) that performs the forward pass (and backward pass) for all layers in that stage. The basic scheduling and execution unit in NASPipe's runtime is a *task*, which is defined as either a subnet stage i's forward pass or backward pass on processing one input batch. Each task is identified by a task property (forward or backward), subnet ID, and stage ID.

Scheduling Policy. Algorithm 1 and Algorithm 2 illustrate how to use the above scheduling unit to implement a scheduling policy to form a pipeline execution of subnets and enforce CSP in supernet learning. This policy maintains a Queue list (L_q), which stores the forward tasks to be executed, and a Finished list (L_f), which stores the finished task, and each runtime worker maintains a list of subnets (L_{SN}), with each subnet represented by a list of operator choices (§3) and asynchronously retrieved from the algorithm programming frontend in a producer-consumer manner ($retrieve()$ in line 16 in Algorithm 1). In all algorithms, *qidx* means the next scheduled index in the subnet queue, and *qval* means the subnet ID, the subnet's exact sequence ID in the total execution order.

Algorithm 1: NASPipe Runtime

```
1  Train (K)
       inputs : $total$, total steps; $K$, stage ID
2      $L_{SN}, L_q, L_f \leftarrow [], [], []$ ;
3      foreach $i \in [0, total - 1]$ do
4          if receiveBwd(recv) then
5              $bwd\_id \leftarrow recv.id$;
6              predictor($L_q$,$L_f$, $L_{SN}$, $K$, $recv$, $None$);
7              runBackward($bwd\_id$);
8              sendBackward($bwd\_id$);
9              flush($bwd\_id$);
10             $L_f$.append($bwd\_id$);
11             ctxt_manager($fwd\_id$, $EVICT$);
12         $qidx \leftarrow -1$ ;
13         $fwd\_id \leftarrow -1$ ;
14         $L_{SN}$.append(retrieve());
15         $qidx, fwd\_id \leftarrow$ schedule($L_q$,$L_f$, $L_{SN}$, $K$);
16         while $fwd\_id < 0$ and receiveFwd(recv) do
17             $L_q$.append($recv.id$);
18             $qidx, fwd\_id \leftarrow$ schedule($L_q$,$L_f$, $L_{SN}$, $K$);
19         if $fwd\_id \geq 0$ then
20             $L_q$.pop($qidx$);
21             predictor($L_q$,$L_f$, $L_{SN}$, $K$, $None$, $fwd\_id$);
22             runForward($fwd\_id$);
23             sendForward($fwd\_id$);
24             ctxt_manager($fwd\_id$, $EVICT$);
```

Algorithm 2: NASPipe Scheduler

```
1  Schedule ($L_q, L_f, L_{SN}, K$)
       inputs : $L_q$, queue LIST of subnet IDs ; $L_f$, finished LIST of
                subnet IDs; $L_{SN}$, subnet LIST with layer IDs; $K$, stage
                ID
       output : $qidx, qval$
2      foreach $qidx, qval \in L_q$ do
3          $scheduled \leftarrow True$;
4          foreach $wval \in [0, qval - 1]$ do
5              if $wval \in L_f$ then
6                  continue
7              foreach $l \in L_{SN}[qval][K]$ do
8                  if $l \in L_{SN}[wval]$ then
9                      $scheduled \leftarrow False$;
10                     break
11         if $scheduled$ then
12             return $qidx, qval$
13     return $-1, -1$;
```

The scheduling policy iteratively waits and selects the next task to run with the following heuristics. (1) Backward tasks are always preferred with the highest priority (lines 6-13 in Algorithm 1). The rationale is that backward tasks can remove the precedence constraints on the following tasks, making a larger scheduling search space. (2) Forward tasks in the queue are scheduled with Schedule() (Algorithm 2), which checks each forward task's causal dependency by comparing its layer choices with each layer in each subnet (line 7-10) that has a lower sequence ID (line 4) and does not show in the finished list (line 5-6). This ensures that the causal dependency preservation property of CSP (*definition-2*) is guaranteed.

Complexity Analysis. The time complexity of Algorithm 2 is $O(|L_q|(|L_f| + m^2)$, where $|L_q|$ is the queue list size, $|L_f|$ is the finished list size, and m is the number of layers in a subnet (i.e., number of choice blocks in a supernet). $|L_q|$ is usually not large (less than 30), restricted by causal dependencies. The number of simultaneously launched subnets is limited. L_f often has the same size with $|L_q|$, as we have an elimination scheme for the finished list that when subnets before a seq ID are all finished, we remove them both from the finished list and the dependencies check in line 4 of Algorithm 2. Overall, the time cost of our scheduler policy call is small (<0.01s) compared with a subnet execution (second level), which incurs a little penalty on NASPipe's performance.

3.3 Context Prediction

Context management on GPUs is key for both single DNN training [11, 12, 25, 48] and multiple DNN inference services multiplexed on the same group of GPUs [3] to reserve only necessary execution context to relieve the GPU memory burden, without notable performance penalty. Existing systems all assume a pre-known execution timeline of DNN operators, so that these systems can perform a pipeline *context switch* that asynchronously copies a DNN operator's parameters between CPU and GPU and overlaps the copy with DNN operator execution. However, in supernet training, the subnet generation is unknown prior to run. Worse, in CSP, the subnet executions are often reordered to reserve causal dependencies and high performance. These obstacles make a context prediction that forecasts the upcoming executed tasks is highly desirable but missing.

Opportunities. Fortunately, we reveal that the execution time of DNN operators on modern GPUs is roughly deterministic, so that given the status of the current stage and the whole pipeline, we are able to forecast the next few scheduled tasks in the near future with high prediction accuracy.

Prediction Policy. Algorithm 3 illustrates how NASPipe predicts the next few scheduled tasks. The Predictor() is called before a backward pass (line 23 in Algorithm 1) and before a forward pass (line 8 in Algorithm 1). The heuristics used for these two types of calls are different. As the backward pass will finish a subnet's write access, it will resolve the dependencies between itself and the following subnets in the queue list (L_q). Therefore, in line 5-6, we pre-add the backward pass to the finished list (L_f) re-run the Schedule(). The produced forward pass task has a high chance to be the next scheduled (line 7-9).

Moreover, in NASPipe, the received backward tasks are transferred from the latter stages of a pipeline, carrying the information of pending backward tasks from the last stage. The pending backward tasks are blocked as a forward pass has not arrived at the last stage in the presence of precedent dependencies. Therefore, during the backward data transfer between stage, each stage checks whether a forward pass's dependencies that block a backward pass is resolvable in this stage; if yes, it passes the pending backward pass to the precedent stages. Before a forward pass, if this forward pass will release a pending backward pass, NASPipe adds this backward pass task to the context fetch schedule (line 13-15). Meanwhile, the prediction before forward passes also re-run the Schedule() to predict the next scheduled forward pass (line 16-18). Predictor() re-runs Schedule and takes similar time cost complexity. Although negligible, this time cost can still be eliminated by asynchronous execution.

Algorithm 3: NASPipe Predictor

```
1  Predictor ($L_q, L_f, L_{SN}, K, recv, current$)
       inputs : $L_q$, queue LIST of subnet IDs; $L_f$, finished LIST of
                subnet IDs; $L_{SN}$, subnet LIST with layer IDs; $K$, stage
                ID; $recv$ backward recv object
2      $L_{blocked}$ ← [] //global variable ;
3      $L_{fetch}$ ← [];
4      if $recv$ not $None$ then //backward
5          $L'$ ← $L_f$;
6          $L'$.append($recv.id$) ;
7          _, $fwd\_id$ ← schedule($L_q$,$L'$, $L_{SN}$, $K$);
8          if $fwd\_id > 0$ then
9              ctxt_manager($fwd\_id$, $FETCH$);
10         foreach $bwd \in recv.next\_bwds$ do
11             $L_{blocked}$.append($bwd$);
12         return
13     foreach $bwd \in L_{blocked}$ do
14         if $current == bwd.precedence$ then
15             ctxt_manager($bwd.id$, $FETCH$);
16     _, $fwd\_id$ ← schedule($L_q$,$L_f$, $L_{SN}$, $K$);
17     if $fwd\_id > 0$ then
18         ctxt_manager($fwd\_id$, $FETCH$);
19     return
```

4 SYSTEM IMPLEMENTATION

NASPipe is implemented as a standalone Python library with about 4000 LoC. NASPipe uses PyTorch [24] for DNN computation and auto-differentiation because NASPipe's design leverages the imperative features from PyTorch: the supernet training needs dynamic DNN computation graph generation. Still, NASPipe's design is general for all imperative training frameworks (e.g., TensorFlow and MxNet). We leave NASPipe's implementation on these frameworks in future work.

4.1 Runtime Construction

NASPipe inherits the fashion of Retiarii's supernet construction with the PyTorch `Module`. Specifically, each supernet choice block with a set of candidate layers is constructed as a `Module List`, and each candidate layer is indexed by a block-wise choice id. Each supernet's choice block `Module` contains a unique `forward(inputs, choices)` function. In addition to training inputs, the forward function receives a list of `choices` that specifies which operators in the `Module List` should be activated for processing the inputs.

We modified Retiarii to generate subnets in a producer-consumer way, where NASPipe is the consumer. As each subnet has a unique balanced partition, NASPipe at runtime translates each subnet's K partitions (K is GPU number) into K `statements` in `string` objects. Each GPU is managed by a hot PyTorch context executor `process`. Each subnet's `forward()` function only contains a Python `exec_stmt()` utility call that executes the generated `statement`. NASPipe spawns a separate worker for each stage (i.e., GPU) and uses PyTorch `distributed` primitives to communicate between stages. NASPipe's runtime eliminates all factors that may break the deterministic execution properties (i.e., reproducibility) by fixing random seeds for PyTorch, Python, and DataLoader, and setting CUDA library deterministic configuration to true.

4.2 Context Management

NASPipe adopts activation recomputing techniques [12, 25] to eliminate the heavy activation memory context using PyTorch's `checkpoint` utility, thus further relieving the GPU memory burden. Note that all baseline systems (except for Pipedream) we evaluated are also enabled with this optimization. NASPipe takes CPU as secondary storage, and by default, the supernet's operators all reside in CPU storage. To handle asynchronous operator copy (i.e., copy the parameter tensors of an operator) between GPU world and CPU world, NASPipe uses PyTorch's native utility of tensor `copy_()` with `non_blocking = True`. Specifically, NASPipe's CPU storage is by default in `pinned` CPU memory. The reason is that in PyTorch, memory copies to devices can be asynchronous when they

Table 1: Default evaluation setup of seven search spaces. Each search space is configured by the number of choice blocks and the number of layers per choice block. A subnet is constructed by selecting a layer from each choice block.

Search Space	# Choice Blocks	# Layer/Block	Dataset
NLP.c0	48	96	WNMT
NLP.c1	48	72	WNMT
NLP.c2	48	48	WNMT
NLP.c3	48	24	WNMT
CV.c1	32	48	ImageNet
CV.c2	32	24	ImageNet
CV.c3	32	12	ImageNet

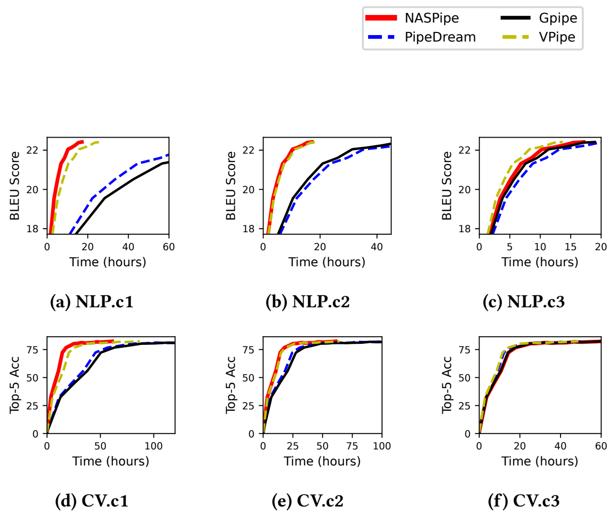
originate from `pinned` CPU (i.e., page-locked) memory. NASPipe invokes a GPU memory limit checking before it copies an operator to GPU (to wait for operators being evicted). If the limit is reached, NASPipe delays the operator copy. NASPipe catches runtime exception per stage execution and re-executes a stage in case that out-of-memory error happens.

Layer Mirroring. NASPipe by default initializes supernet layers with a partition based on their choice block hierarchy, with each partition initialized in each stage's `pinned` CPU storage. When a layer needs to be mirrored to another stage, NASPipe uses PyTorch's `add_module()` to mirror this layer. If a mirrored layer's parameters are updated, the new parameters are *actively* pushed to all the other mirrored ones via PyTorch `distributed` communication library and the corresponding parameter update is applied by the `load_dict()` member function of the layer's `Module` object.

5 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 of 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 were connected with 40 Gbps Ethernet, and the average ping latency was 0.17ms. In all our evaluations, the maximized network bandwidth across hosts used for both NASPipe and our baseline systems was 867MB/s, indicating that the network hardware was not the bottleneck of our experiments.

Baseline Systems. To evaluate the performance of NASPipe, we compared NASPipe with GPipe [12], PipeDream [21], and VPipe [48]. GPipe can achieve the most compact GPU memory utilization for pipeline parallelism with activation tensor rematerialization (§4). PipeDream interleaves the forward and backward computation on each GPU with asynchronous parameter update (i.e., ASP) to reduce the GPU idle time. VPipe further reduces the GPU memory consumption in the pipeline parallel training system by swapping the model parameters to CPU memory. GPipe and VPipe are all configured with BSP, sharing the same reproducibility with Retiarii's. We did not select Retiarii's performance as a baseline, because Retiarii's parallel execution cannot support training the large supernets as we evaluated. However, Retiarii [45]'s programming interface is essential for all these supernets to support NAS training. Therefore, NASPipe and all baseline systems are integrated with Retiarii (§4).



(a) NLP.c1 (b) NLP.c2 (c) NLP.c3
(d) CV.c1 (e) CV.c2 (f) CV.c3

Figure 4: The end-to-end training convergence comparison of NASPipe, GPipe, PipeDream, and VPipe. In all six search spaces, NASPipe can converge to a higher BLEU score (for NLP tasks) or higher top-5 accuracy (for CV tasks) than GPipe.

Datasets. We conducted our study on the two most representative datasets, ImageNet [7], and WNMT [31], targeting both computer vision and language processing tasks. ImageNet is a full-scale image dataset that contains 600k images from 1k classes. WNMT is the major translation dataset that has been used in recent natural language processing (NLP) studies.

Search Space and Search Strategy. We evaluated NASPipe on both language processing (e.g., NLP) and computer vision (e.g., CV) tasks. We adopted search space defined by Evolved Transformer [34] for the NLP task and search space defined by AmoebaNet [37] for the CV task. We derived a set of search spaces (4 for NLP tasks and 3 for CV tasks) by altering the number of candidate layers for choice blocks in the search space. The setup of the 7 search spaces is listed in Table 1.

Specifically, we migrated the initial open-source supernet implementation [37] to Retiarii for AmoebaNet. For Evolved Transformer, we extended their official implementation to Retiarii's supernet search space implementation. For all self-implemented and migrated supernet we evaluated, we achieved comparable search quality (i.e., the validation score of the final searched DNNs) compared with the reported results in their papers. For all NAS search spaces we evaluated, we used evolution [29] as the default search strategy.

Default Setting and Metrics. By default, we used 8 GPUs for each experiment with the default settings in Table 1. For training each search space's supernet, every sampled subnet was trained by one training step. Each training step means the training of one input batch. We measured the throughput by data samples per second. We used the GPU ALU to represent each GPU's average utilization, collected from Nvidia's GPU profiling tools. We used factors like 7.8X in Table 2 to make the total memory/ALU usage normalized to a single GPU's memory limit (e.g., 11GB) and ALU limit (100%) for easy comparison.

Our evaluation focuses on the following questions:
§5.1: How is NASPipe's end-to-end performance compared to baselines?
§5.2: How is NASPipe's reproducibility compared to baselines?
§5.3: How effective are NASPipe's components?
§5.4: How scalable is NASPipe's design to the number of GPUs?
§5.5: What are the lessons we learned?

5.1 NAS Supernet Training Performance

Figure 4 shows the resulting training convergence curve of six supernets (i.e., search space) and Figure 5 shows the normalized training throughput. Overall, NASPipe achieved $1.1X \sim 7.8X$ throughput compared to GPipe and $0.87X \sim 6.5X$ throughput compared to PipeDream on training the same supernets. Meanwhile, NASPipe has $0.77X \sim 1.5X$ throughput compared to VPipe. Both GPipe and PipeDream failed to run search space NLP.c0 because the supernet parameter size exceed the GPU memory capacity. Note that NASPipe is the only system that retains reproducibility, and we will further evaluate this in §5.2.

To further break down the performance gains that we collected, Table 2 shows the runtime statistics of each comparison. NASPipe's performance improvement over GPipe, PipeDream, and VPipe comes from three factors. First, as shown in the batch size (B.S.) column of Table 2, NASPipe supported $1.3X \sim 6X$ larger batch size than GPipe and $2.7X \sim 12X$ batch size than PipeDream. This is because NASPipe incurred less GPU memory burden for only stashing the layers of subnets being executed at the cost of extra CPU storage (the CPU memory column in Table 2). Note that, we measured the efficacy of our context prediction by the Cache Hit (rate) column, which was collected by checking whether an ML layer's parameter was in GPU memory before its execution. NASPipe's cache size was roughly 3X of a subnet's parameter memory (see Para. column in Table 2; NASPipe's cached parameter size was about 3X VPipe's; VPipe cached one subnet in GPU). The cache is composed of three parts: the current subnet being executed, the previous subnet to be evicted to CPU memory, and the next subnet to be pre-fetched into GPU memory.

Second, NASPipe enforced that each subnet was always executed with a balanced partition. Although the batch size support by VPipe is the same as NASPipe, the NASPipe's average execution time

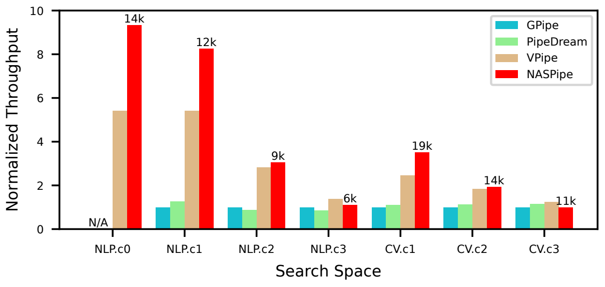


Figure 5: Normalized throughput of four systems on seven search spaces. The value on each red bar indicates NASPipe's average number of traversed (trained) subnets per hour.

(bubble eliminated) of a subnet (Exec. column of Table 2) for search space NLP.c1 is 1.13s, and the time of VPipe is 1.21s. A subnet's execution time of NASPipe is 9.6% less than VPipe on average. This indicates that NASPipe achieved a much more balanced and stable pipeline training than VPipe.

Third, NASPipe incurs a bubble time comparable to GPipe. The bubble time ratio (Bubble column of Table 2) of the BSP system GPipe is constantly 0.57 for all the search spaces. NASPipe achieved a bubble time ratio approximately the same as GPipe for search space NLP.c2 and CV.c2. For larger search spaces NLP.c1 and CV.c1, the bubble time ratio of NASPipe reached 0.39 and 0.43. The ASP system PipeDream interleaved the forward and backward computation and incurred a bubble time ratio of 0.1.

NASPipe's performance improvement over GPipe, PipeDream, and VPipe increases with the growth of search space size. From Table 2, NASPipe achieves merely $0.1X$ throughput improvement compared to GPipe for search space NLP.c3, while the improvement increases to $6.8X$ for a larger search space NLP.c1. With the growth of search space size, the bubble time ratio of NASPipe decreases and NASPipe's batch size enlargement over GPipe and PipeDream increases.

Note that, for NLP.c3 and CV.c3 (in Figure 5), we relieved the excessive supernet context burden in baselines by limiting the supernet's parameter size. In these cases, NASPipe's context management gained little advantage over the baselines. Meantime, we believe that, in such small search spaces, Retiarii's default parallel strategy (i.e., wrapped data parallelism with parameter server, see §2.2) could achieve comparable parallel efficiency with the baselines and NASPipe. Nevertheless, on large search spaces, NASPipe is the only system that achieves both high performance and reproducibility.

5.2 Reproducibility

We ran each of NASPipe, GPipe and PipeDream on 4, 8 and 16 GPUs to evaluate their reproducibility. Note that, all these systems were configured with deterministic execution (§4.1). We kept the random seed, batch size, and other hyperparameters (e.g., steps) the same when running a system on different numbers of GPUs. The results for six search spaces are listed in Table 3. NASPipe generated supernets with the same losses across 3 runs, and the scores (BLEU scores for NLP task and Top-5 accuracies for CV task) of the subnets derived from the supernet were also the same. As for GPipe and PipeDream, the generated supernets had different losses across the 3 runs, and a different subnet was derived in each run. To conclude, only the search result of the system adopted CSP was reproducible.

We then explain how the synchronization methods of NASPipe (CSP), GPipe (BSP), and PipeDream (ASP) affect the final NAS results. A layer's parameter is READ in the forward pass and updated (WRITE) in the backward pass. Table 4 demonstrates how the parameter of a randomly chosen layer in a supernet was accessed when using NASPipe, GPipe, and PipeDream on 4 and 8 GPUs. The chosen layer was sampled by the 2nd, 5th, and 7th subnets. The layer was then accessed and updated by the batch corresponding to each subnet. For NASPipe adopted CSP, the order a layer was accessed and updated remained the same when running on 4 and 8 GPUs, while the order differed for both GPipe and PipeDream. By

Table 2: Resource consumption and micro events. "P.S." indicates either the parameter size of a subnet in NASPIPE/VPipe or the whole supernet in GPipe/PipeDream; "Score" measures the quality of a converged supernet; "B.S." is the batch size of input to a pipeline; "GPU Mem (GPU ALU)" is the total used GPU memory (ALU utilization) across 8 GPUs; "CPU Mem" is the utilized CPU memory. Note that we do not incur CPU memory overhead in ShuffleNet because the size of ShuffleNet search space is small enough to fit into a single GPU's memory; "Exec." is the average execution time (bubble eliminated) of a subnet; "Bubble" is the bubble time ratio in the pipeline; "Cache Hit" is the hit rate of accessing a layer from GPU memory.

Space	System	Para.	Score	Batch	GPU Mem.	GPU ALU	CPU Mem.	Exec.(s)	Bub.	Cache Hit
NLP.c1	NASPipe	1327M	22.17	192	7.8x	3.9x	57.8G	1.13	0.39	86.4%
	PipeDream	14.8B	21.97	16	7.8x	0.6x	0	0.49	0.1	N/A
	GPipe	14.8B	22.09	32	7.7x	0.5x	0	0.54	0.57	N/A
	VPipe	474M	22.03	192	7.6x	2.6x	58.7G	1.21	0.57	1.1%
NLP.c2	NASPipe	1350M	21.58	192	7.8x	2.8x	40.0G	1.09	0.57	91.6%
	PipeDream	10.8B	20.43	24	7.9x	0.8x	0	0.51	0.1	N/A
	GPipe	10.8B	21.39	64	7.7x	0.9x	0	0.68	0.57	N/A
	VPipe	460M	20.74	192	7.5x	2.6x	40.9G	1.24	0.57	2.0%
NLP.c3	NASPipe	1327M	20.46	192	7.8x	2.0x	20.3G	1.12	0.68	94.3%
	PipeDream	5.8B	19.28	48	7.9x	1.6x	0	0.59	0.1	N/A
	GPipe	5.8B	20.17	128	7.8x	1.8x	0	0.96	0.57	N/A
	VPipe	440M	19.42	192	7.5x	2.6x	21.2G	1.20	0.57	4.1%
CV.c1	NASPipe	940M	82.4%	64	7.8x	3.5x	29.1G	0.87	0.43	92.6%
	PipeDream	7.4B	79.5%	12	7.7x	1.1x	0	0.51	0.1	N/A
	GPipe	7.4B	81.4%	24	7.9x	1.0x	0	0.63	0.57	N/A
	VPipe	337M	80.7%	64	7.6x	2.4x	29.6G	0.96	0.57	2.0%
CV.c2	NASPipe	983M	81.8%	64	7.9x	2.6x	14.6G	0.86	0.58	94.8%
	PipeDream	3.8B	78.3%	16	7.7x	1.5x	0	0.54	0.1	N/A
	GPipe	3.8B	81.0%	32	7.8x	1.3x	0	0.65	0.57	N/A
	VPipe	325M	79.1%	64	7.5x	2.4x	15.2G	0.95	0.57	4.1%
CV.c3	NASPipe	1021M	81.5%	64	7.6x	2.0x	6.1G	0.89	0.68	97.1%
	Pipedream	1.7B	77.8%	24	7.7x	2.3x	0	0.63	0.1	N/A
	GPipe	1.7B	81.2%	48	7.8x	1.9x	0	0.76	0.57	N/A
	VPipe	342M	78.6%	64	7.5x	2.4x	6.8G	0.96	0.57	7.9%

retaining this access order, NASPIPE achieved the reproducibility property on an arbitrary number of parallel GPUs.

5.3 Ablation Study of NASPIPE's Components

NASPIPE's performance gain primarily comes from its three key components: *scheduler* (§3.2) that deterministically resolves causal dependencies without harming much parallel efficiency, *predictor* (§3.3) that precisely switches subnets with little penalties and makes our supported batch size larger (higher GPU utilization), and *layer mirroring* (§4.2) that enforces each subnet to be executed at a more balanced partition (more efficient pipeline). In this subsection, we conducted an ablation study to evaluate the effectiveness of these three components by disabling each component, respectively. Overall, we evaluated four systems in this subsection: NASPIPE, NASPIPE w/o scheduler that disabled the scheduler of NASPIPE, NASPIPE w/o predictor that disabled the predictor of NASPIPE, and NASPIPE w/o mirroring that disabled the context manager of NASPIPE. Figure 6 shows the normalized throughput of the four systems. Overall, NASPIPE achieved higher throughput than the other three systems. This indicates that all the three key components of NASPIPE contribute to NASPIPE's improved performance.

When NASPIPE's predictor was disabled, the whole supernet was stored inside GPU memory. This made NASPIPE w/o predictor

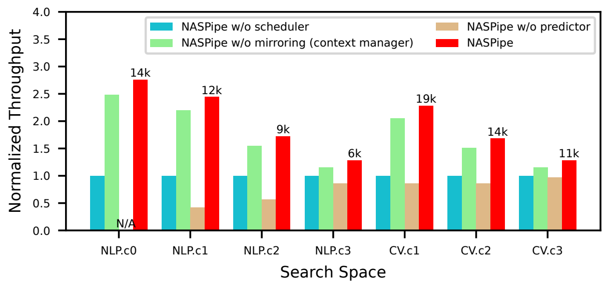


Figure 6: Normalized throughput of four systems on seven search spaces. The value on each red bar indicates NASPIPE's average number of traversed (trained) subnets per hour.

support a smaller batch size (same as GPipe, see Table 2). Still, NASPIPE w/o predictor incurred a higher throughput than GPipe, as the context manager made the execution of a pipeline on GPUs more balanced, and the scheduler reduced the bubble time ratio of NASPIPE.

When NASPIPE's context manager was disabled, it can be observed that the throughput of NASPIPE w/o mirroring slightly dropped. This was because the execution time of a subnet's partitions was no longer balanced without NASPIPE's context manager.

Table 3: Reproducibility. Search accuracy is converged accuracy of the DNN with the highest quality searched out from the trained supernet.

	Sync.	Supernet Loss			Search Accuracy		
		4GPU	8GPU	16GPU	4GPU	8GPU	16GPU
NLP.c1	CSP	5.9663	5.9663	5.9663	22.17	22.17	22.17
	BSP	6.0026	5.9772	5.9974	22.09	21.32	21.75
	ASP	6.7765	6.7835	6.8262	21.97	21.03	21.83
NLP.c2	CSP	5.7911	5.7911	5.7911	21.58	21.58	21.58
	BSP	5.9882	6.0825	6.0049	21.39	21.15	20.93
	ASP	6.5434	6.6661	6.5975	20.43	20.36	20.75
NLP.c3	CSP	5.4315	5.4315	5.4315	20.46	20.46	20.46
	BSP	5.7690	5.4091	5.5561	20.17	20.37	20.29
	ASP	6.2453	6.0856	6.4483	19.28	19.05	18.73
CV.c1	CSP	5.3086	5.3086	5.3086	82.4	82.4	82.4
	BSP	5.4136	5.6453	5.4674	81.4	82.1	82.3
	ASP	5.9086	6.1061	5.8269	79.5	80.7	80.1
CV.c2	CSP	5.2841	5.2841	5.2841	81.8	81.8	81.8
	BSP	5.4497	5.3986	5.5239	81.0	81.5	80.7
	ASP	5.8401	5.9625	5.6104	78.3	79.6	78.1
CV.c3	CSP	5.1193	5.1193	5.1193	81.5	81.5	81.5
	BSP	5.1358	5.2957	5.2829	81.2	80.9	81.2
	ASP	5.4509	5.5121	5.2788	77.8	78.5	78.8

Table 4: Access and update order of a layer in the supernet. *n*F means accessed by the *n*-th batch's forward pass and *n*B means updated by the *n*-th batch's backward pass.

	Access & Update Order	
	4 GPUs	8 GPUs
NASPipe	2F-2B-5F-5B-7F-7B	2F-2B-5F-5B-7F-7B
GPipe	2F-2B-5F-5B-7F-7B	2F-5F-7F-2B-5B-7B
PipeDream	2F-2B-5F-7F-5B-7B	2F-5F-7F-2B-5B-7B

Table 5: Comparison of computation and swap time for eight representative layers. Column Comp. shows a layer's forward/backward computation time in milliseconds. Column Swap shows time in milliseconds to swap a layer's parameters from CPU memory to GPU memory.

	Input Size	Layer	Comp.	Swap
NLP	(192, 1024)	Conv 3x1	5.0/10.0	1.76
		Sep Conv 7x1	4.2/5.7	0.56
		Light Conv 5x1	0.68/1.4	0.03
		8 Head Attention	7.9/13.8	2.07
CV	(64, 112, 112)	Conv 3x3	7.9/13.8	4.6
		Sep Conv 3x3	2.8/4.0	0.68
		Sep Conv 5x5	6.7/9.9	2.04
		Dil Conv 3x3	2.5/3.4	0.58

As a result, the partition with the longest execution time bottlenecked the subnet's whole execution.

When NASPipe's scheduler was disabled, the throughput of NASPipe w/o scheduler became 0.2*X* to 0.7*X* lower than NASPipe. This was because NASPipe w/o scheduler had to finish the execution of a pipeline before injecting the next pipeline. As a result, NASPipe w/o scheduler incurred a bubble time ratio of 0.75, which degraded the final throughput. In sum, Figure 6 indicates all three components of NASPipe are essential to make NASPipe's training performance high and stable.

5.4 Scalability

To evaluate whether NASPipe is scalable to larger search space (thus requires more GPUs), we ran NASPipe, GPipe, PipeDream, and VPipe on 4 to 16 GPUs for search space NLP.c1, which is the largest search space that all the four systems support.

In Figure 7, we used the total GPU ALU utilization to evaluate the scalability. Overall, NASPipe was able to scale sub-linearly. As we can see, NASPipe's scalability curve dropped when the GPU number became larger. This slowdown attributes to two factors. First, the communication time increases in a pipeline for a larger GPU number. As a GPU stays idle when awaiting inputs from other GPUs, the GPU utilization decreases with the increase of communication time ratio in a pipeline. Second, the causal dependency caused a larger bubble time ratio. We observed that the bubble time ratio is 0.39 for 8 GPUs, and the value increased to 0.42 for 16 GPUs.

GPipe, PipeDream, and VPipe achieved poorer scalability with the increase of GPU numbers. Although the bubble time ratio of BSP systems GPipe and VPipe stays the same for different numbers of GPUs, the problem of unbalanced GPU ALU utilization became more severe and degraded the performance. NASPipe maintained a balanced partition for the subnets when running on different numbers of GPUs. PipeDream incurred only a small portion of bubble time, but it still suffered from the unbalanced partition.

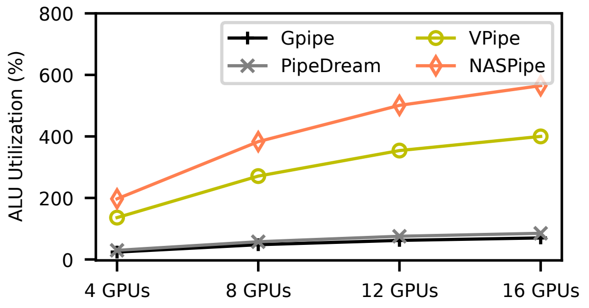


Figure 7: The total ALU utilization of different systems running with a scaled number of GPUs.

5.5 Lessons Learned

Limitations. NASPipe's design has two limitations. First, NASPipe's implementation requires the imperative feature of a training framework. Thus, training frameworks with static execution cannot be integrated with NASPipe. Still, most of the existing popular training frameworks, including Tensorflow [2], PyTorch [24], and MxNet [24], all support imperative programming. Second, NASPipe's major system optimizations are designed for the latest and popular NAS paradigms [9, 18, 26]. However, NAS is a hot and extensive research area containing many other heterogenous search methods. Thus, NASPipe's design is only limited to the supernet-based NAS algorithms that are programmable via Retiarii [45]'s abstractions.

Future Applications. We envision that NASPipe can facilitate two new training strategies for NAS supernet training, which can enable DNN architecture engineers to explore more valuable DNNs. First, NASPipe allows the hybrid traverse of multiple search spaces simultaneously as NASPipe's runtime design is flexible to hold any number of causal dependency relations. Second, NASPipe has the potential to be extended for supporting all supernet adoptions, including the dynamic networks [15] and a mixture of experts model [43]. We leave this for future work.

6 RELATED WORK

Prior GPU memory reduction systems [3, 11, 25, 30, 38] only capture layer dependencies inside a static DNN's forward and backward pass. These systems are not designed for NAS supernet training because NAS involves causal dependencies among subnets' layers. Unified Memory [22] enables a unified virtual memory space across CPUs and GPUs, but it is a black-box approach and unaware of the DNN training context at runtime.

Existing deep learning frameworks (e.g., Pytorch [24], Tensorflow [2]) are designed to describe and train a single DNN model. While some Automated Machine Learning (AutoML) systems (e.g., AutoGluon [8], AutoKeras [13]) can automate the model searching process, these systems are limited to domain-specific search space. Retiarii [45] introduces the mutator abstraction, which can describe any search space and facilitate model exploration. Our system NASPipe can be easily integrated with Retiarii.

There are Deep Learning scheduling systems [10, 14, 19, 20, 39, 40] that provide model-wise scheduling across a GPU cluster. Still, NASPipe solves domain-specific challenges stemming from NAS supernet training. There are tremendous studies [23, 32, 47] that explored the race hazard in various programs, NASPipe's is the first work to study the race hazard in parallelizing supernet training. There are studies [1, 24, 46] studying the deterministic ML training at hardware level or DNN operator level, and we study the deterministic ML training at the NAS supernet level. For parallel training of a static DNN, there are three parallel dimensions including data parallelism [17, 27], tensor model parallelism [33], and pipeline model parallelism [12, 21, 41]. NASPipe is a practical exercise that parallelizes a multi-subnet and dynamic training procedure across GPUs, which can be considered as a fourth parallel dimension (supernet/subnet parallelism); supernet/subnet parallelism can be integrated with other parallel dimensions.

7 CONCLUSION

We propose NASPipe, a full-fledged high-performance, and reproducible parallel training system via the CSP pipelining for supporting supernet training. NASPipe serves as a backend beneath a NAS programming frontend (e.g., Retiarii). Extensive results show NASPipe's design can provide a crucial property *reproducibility* for supernet training, with high training performance. NASPipe envisions a wide application scope for DNN experts to explore more valuable DNNs.

ACKNOWLEDGMENTS

We thank our shepherd, Chris Rossbach, and all anonymous reviewers for their helpful comments. This work is supported in part by a Huawei Flagship Research Grant in 2021, a HKU-SCF FinTech Academy R&D Funding Scheme in 2021, HK RGC GRF (17202318, 17207117), the National Natural Science Foundation of China (61802358), the Shanghai AI Lab, the HKU and IS-CAS Joint Lab for Intelligent System Software, and a Croucher Innovation Award.

A ARTIFACT APPENDIX

A.1 Abstract

This appendix describe the availability and functionality of the paper's artifact: a causal parallel training execution framework. The artifact requires a host with at least 100GB CPU RAM and 4 Nvidia GPUs, and each GPU requires at least 11GB memory. The runtime environment is installed by docker with a few command lines. The experiments contain a throughput evaluation and reproducible training evaluation. The artifact provides one-click shell scripts to conduct the experiments.

A.2 Artifact Check-list (Meta-information)

- **Program:** Shell scripts; Python program.
- **Model:** Transformer-based Neural Architecture Search model.
- **Data Set:** WMT 2014 English-German dataset.
- **Run-time Environment:** python 3.6.9, GCC 7.3.0, torch 1.9.0+cu102, torchvision 0.10.0+cu102, fairseq commit 0dfd6b6, cuda >= 10.2, nvidia driver 455.38, 18.04.3 LTS (Bionic Beaver).
- **Hardware:** 1 host with 4 Nvidia RTX 2080ti GPUs (or any Nvidia GPUs with >= 11GB memory) and 100GB CPU RAM
- **Execution:** Several command lines to run shell scripts.
- **Experiments:** Training throughput comparison between NLP.c0-c3 in Table 1 in four GPUs setting. Training reproducibility experiment between single GPU and four GPUs settings.
- **How Much Disk Space Required (Approximately)?:** 200GB
- **How Much Time is Needed to Prepare Workflow (Approximately)?:** 90 minutes.
- **How Much Time is Needed to Complete Experiments (Approximately)?:** 30 minutes.
- **Publicly Available?:** Yes.
- **Code Licenses?:** MIT license
- **Archived?:** https://doi.org/10.5281/zenodo.5739442

A.3 Description

A.3.1 How to Access. The artifact is accessible via the persistent DOI link. It is optionally accessible via the public Github repository link.
Persistent Link: https://doi.org/10.5281/zenodo.5739442
Optional Link: https://github.com/hku-systems/naspipe

A.4 Installation

You can also refer to README.md in the artifact with more detailed documentation.
Pull and run PyTorch official image:

```
docker pull pytorch/pytorch:1.9.0-cuda10.2-cudnn7-devel
cd naspipe
nvidia-docker run -it -v $PWD:/workspace –net=host –ipc=host
–name=naspipe pytorch/pytorch: 1.9.0-cuda10.2-cudnn7-devel
```

Inside docker, install dependencies:

```
apt-get update
```

```
    apt-get install -y git
    cd /
    git clone https://github.com/pytorch/fairseq
    cd fairseq
    pip install –editable ./
Fix fairseq compatibility issues:
    git am < /workspace/0001-compatible.patch
    cd /workspace
Fix deterministic execution issues:
    export CUBLAS_WORKSPACE_CONFIG=:4096:8
```

A.5 Evaluation and Expected Results

Experiment 1: Reproducible parallel training on single GPU and four GPUS settings. Search space NLP.c0. Comparing 500 training step outputs in full precision floating point.

Command: ./run_compare.sh

Output: All 500 training steps outputs in full precision floating point matches between settings.

Experiment 2: Training throughput comparison between NLP.c0-c3 in Table 1 in four GPUs setting.

Command: ./run_throughput.sh

Output: Training throughput of all settings with T(NLP.c0) > T(NLP.c1) > T(NLP.c2) > T(NLP.c3).

REFERENCES

[1] [n. d.]. NVIDIA/framework-determinism. https://github.com/NVIDIA/framework-determinism.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[3] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 499–514.

[4] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. 2018. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*. PMLR, 550–559.

[5] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).

[6] John Crawford. 1990. The execution pipeline of the intel i486 cpu. In *1990 Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*. IEEE Computer Society, 254–255. https://doi.org/10.1109/CMPCON.1990.63682

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[8] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505* (2020).

[9] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. 2020. Single path one-shot neural architecture search with uniform sampling. In *European Conference on Computer Vision*. Springer, 544–560. https://doi.org/10.1007/978-3-030-58517-4_32

[10] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2018. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288* (2018).

[11] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355. https://doi.org/10.1145/3373376.3378530

[12] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2018. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965* (2018).

[13] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1946–1956. https://doi.org/10.1145/3292500.3330648

[14] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning. *arXiv preprint arXiv:2012.02732* (2020).

[15] Changlin Li, Guangrun Wang, Bing Wang, Xiaodan Liang, Zhihui Li, and Xiaojun Chang. 2021. Dynamic slimmable network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8607–8617. https://doi.org/10.1109/CVPR46437.2021.00850

[16] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598. https://doi.org/10.1145/2640087.2644155

[17] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. 2013. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, Vol. 6. 2. https://doi.org/10.1145/2640087.2644155

[18] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[19] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 553–564. https://doi.org/10.1109/HPCA.2017.29

[20] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient {GPU} Cluster Scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 289–304.

[21] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15. https://doi.org/10.1145/3341301.3359646

[22] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. 2014. Unified memory in cuda 6.0. a brief overview of related data access and transfer issues. *SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09* (2014).

[23] Robert O'callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 167–178. https://doi.org/10.1145/966049.781528

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[25] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 891–905. https://doi.org/10.1145/3373376.3378505

[26] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*. PMLR, 4095–4104.

[27] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506. https://doi.org/10.1145/3394486.3406703

[28] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 33. 4780–4789. https://doi.org/10.1609/aaai.v33i01.33014780

[29] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. 2017. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*. PMLR, 2902–2911.

[30] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13. https://doi.org/10.1109/MICRO.2016.7783721

[31] Jean Senellart, Dakun Zhang, Bo Wang, Guillaume Klein, Jean-Pierre Ramatchandirin, Josep M Crego, and Alexander M Rush. 2018. OpenNMT system description for WNMT 2018: 800 words/sec on a single-core CPU. In *Proceedings of the 2nd Workshop on Neural Machine Translation and Generation*. 122–128. https://doi.org/10.18653/v1/W18-2715

[32] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. 62–71. https://doi.org/10.1145/1791194.1791203

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter

language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).
[34] David So, Quoc Le, and Chen Liang. 2019. The evolved transformer. In *International Conference on Machine Learning*. PMLR, 5877–5886.
[35] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. 2019. Single-path nas: Designing hardware-efficient convnets in less than 4 hours. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 481–497. https://doi.org/10.1007/978-3-030-46147-8_29
[36] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2820–2828. https://doi.org/10.1109/CVPR.2019.00293
[37] Mingxing Tan, Ruoming Pang, and Quoc V Le. 2020. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 10781–10790. https://doi.org/10.1109/CVPR42600.2020.01079
[38] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53. https://doi.org/10.1145/3200691.3178491
[39] Qiang Wang, Shaohuai Shi, Canhui Wang, and Xiaowen Chu. 2020. Communication Contention Aware Scheduling of Multiple Deep Learning Training Jobs. *arXiv preprint arXiv:2002.10105* (2020).
[40] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 595–610.
[41] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. 2021. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems* 3 (2021).
[42] Shan You, Tao Huang, Mingmin Yang, Fei Wang, Chen Qian, and Changshui Zhang. 2020. Greedynas: Towards fast one-shot nas with greedy supernet. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1999–2008. https://doi.org/10.1109/CVPR42600.2020.00207
[43] Zhao You, Shulin Feng, Dan Su, and Dong Yu. 2021. SpeechMoE: Scaling to Large Acoustic Models with Dynamic Routing Mixture of Experts. *arXiv preprint arXiv:2105.03036* (2021). https://doi.org/10.21437/Interspeech.2021-478
[44] Miao Zhang, Huiqi Li, Shirui Pan, Taoping Liu, and Steven Su. 2020. One-Shot Neural Architecture Search via Novelty Driven Sampling. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, Christian Bessiere (Ed.). International Joint Conferences on Artificial Intelligence Organization, 3188–3194. https://doi.org/10.24963/ijcai.2020/441 Main track.
[45] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. 2020. Retiarii: A Deep Learning Exploratory-Training Framework. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 919–936.
[46] Shixiong Zhao, Xusheng Chen, Cheng Wang, Fanxin Li, Qi Ji, Heming Cui, Cheng Li, and Sen Wang. 2020. HAMS: High Availability for Distributed Machine Learning Service Graphs. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 184–196. https://doi.org/10.1109/DSN48063.2020.00036
[47] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. 2018. Owl: Understanding and detecting concurrency attacks. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 219–230. https://doi.org/10.1109/DSN.2018.00033
[48] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, et al. 2021. VPIPE: A Virtualized Acceleration System for Achieving Efficient and Scalable Pipeline Parallel DNN Training. *IEEE Transactions on Parallel and Distributed Systems* (2021). https://doi.org/10.1109/TPDS.2021.3094364
[49] Yiyang Zhao, Linnan Wang, Yuandong Tian, Rodrigo Fonseca, and Tian Guo. 2021. Few-shot neural architecture search. In *International Conference on Machine Learning*. PMLR, 12707–12718.
[50] Martin Zinkevich, Markus Weimer, Alexander J Smola, and Lihong Li. 2010. Parallelized stochastic gradient descent.. In *NIPS*, Vol. 4. Citeseer, 4.
[51] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).