# Dilu: Enabling GPU Resourcing-on-Demand for Serverless DL Serving via Introspective Elasticity

Cunchi Lv
lvcunchi21s@ict.ac.cn
ICT, CAS
UCAS
Zhongguancun Lab
Beijing, China

Xiao Shi*
shixiao@ict.ac.cn
ICT, CAS
Nanjing Institute of InforSuperBahn
Beijing, China

Zhengyu Lei
leizhengyu20s@ict.ac.cn
ICT, CAS
UCAS
Zhongguancun Lab
Beijing, China

Jinyue Huang
huangjinyue22s@ict.ac.cn
ICT, CAS
UCAS
Beijing, China

Wenting Tan
tanwenting@ict.ac.cn
ICT, CAS
Beijing, China

Xiaohui Zheng
zhengxiaohui@ict.ac.cn
ICT, CAS
Beijing, China

Xiaofang Zhao
zhaoxf@ict.ac.cn
ICT, CAS
IICT, Suzhou, CAS
UCAS, Nanjing
Zhongguancun Lab
Beijing, China

## Abstract

Serverless computing, with its ease of management, auto-scaling, and cost-effectiveness, is widely adopted by deep learning (DL) applications. DL workloads, especially with large language models, require substantial GPU resources to ensure QoS. However, it is prone to produce GPU fragments (e.g., 15%-94%) in serverless DL systems due to the dynamicity of workloads and coarse-grained static GPU allocation mechanisms, gradually eroding the profits offered by serverless elasticity.

Different from classical serverless systems that only scale horizontally, we present introspective elasticity (IE), a fine-grained and adaptive two-dimensional co-scaling mechanism to support GPU resourcing-on-demand for serverless DL tasks. Based on this insight, we build Dilu, a cross-layer and GPU-based serverless DL system with IE support. First, Dilu provides multi-factor profiling for DL tasks with efficient pruning search methods. Second, Dilu adheres to the resourcing-complementary principles in scheduling to improve GPU utilization with QoS guarantees. Third, Dilu adopts an adaptive 2D co-scaling method to enhance the elasticity of GPU provisioning in real time. Evaluations show that it can dynamically adjust the resourcing of various DL functions with low GPU fragmentation (10%-46% GPU defragmentation), high throughput (up to 1.8× inference and 1.1× training throughput increment) and QoS guarantees (11%-71% violation rate reduction), compared to the SOTA baselines.

***CCS Concepts:*** • **Computer systems organization** → **Cloud computing**.

***Keywords:*** Serverless Deep Learning, GPU Resourcing-on-Demand, Introspective Elasticity, Co-scaling

**ACM Reference Format:**
Cunchi Lv, Xiao Shi, Zhengyu Lei, Jinyue Huang, Wenting Tan, Xiaohui Zheng, and Xiaofang Zhao. 2025. Dilu: Enabling GPU Resourcing-on-Demand for Serverless DL Serving via Introspective Elasticity. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3669940.3707251

## 1 Introduction

Serverless computing has been widely used in DL serving. Many end-to-end DL platforms, like AWS SageMaker [5], Alibaba PAI [4] and Microsoft ACI [33], have adopted serverless concepts to provide ease-of-use experience and reduce management efforts of developers. Studies investigate the integration of elastic training [18, 26, 50] and inference [19, 25, 51] tasks within serverless, offering numerous benefits such as low resource consumption, automatic deployment, and auto-scaling. Propelled by the emergence of Large Language Models (LLMs), GPU-based serverless DL systems become more popular and notable [7, 15, 19, 51].

However, GPU fragmentation tends to occur in serverless DL systems due to various factors, such as the dynamicity of DL task workloads, static GPU allocation and keep-alive strategies to balance the overheads of cold starts. Further, it leads to several issues, such as low resource utilization, and high cost, which significantly undermine elasticity and

*Corresponding author.



*ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands*

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707251

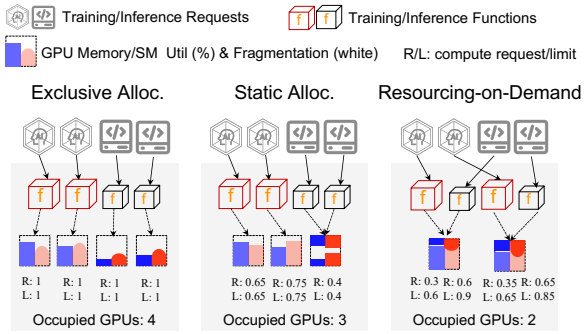



**Figure 1.** GPU provisioning of serverless DL systems.

cost-efficiency (i.e., pay-as-you-go) brought by serverless computing. Specifically, as shown in Figure 1, most systems [7, 15, 18] adopt the exclusive GPU allocation method (left), resulting in a significant waste of GPU resources. Works like [10, 19, 51], leverage slightly fine-grained GPU provisioning via MPS [38], allowing several instances to share a single GPU with a fixed resource quota (medium), which lacks efficient dynamic resourcing adjustment. We argue that resourcing-on-demand GPU provisioning is the ideal status to improve GPU utilization in serverless computing (right), maximizing the potential benefits of serverless elasticity.

Constrained by the aforementioned exclusive or static GPU provisioning, existing serverless DL systems [18, 19, 51] merely scale horizontally (i.e., scaling in/out at the inter-instance level) reactively in response to workload changes. In contrast, we present introspective elasticity (IE), which means fine-grained and dynamic GPU provisioning for serverless DL functions, supporting vertical scaling (scale up/down GPU compute cores at the intra-instance level) and horizontal scaling in a coordinated manner to minimize GPU fragmentation. However, it is non-trivial to design such a cross-layer system. It faces several challenges, including accurately profiling basic resource requirements of DL tasks to identify fragments, scheduling with consideration of multiple factors (e.g., Quality of Service/QoS, heterogeneous workloads), and performance interference caused by disordered resource consumption between instances in real time.

In this paper, we build Dilu, a serverless DL system designed to achieve GPU resourcing-on-demand with introspective elasticity. First, to determine multiple resourcing requirements to facilitate identifying GPU fragments, especially GPU compute resources, we introduce efficient pruning search strategies for heterogeneous DL tasks. Second, we design a resourcing-complementary scheduling policy to minimize GPU consumption while ensuring QoS of collocated tasks. Most importantly, we present an adaptive two-dimensional co-scaling strategy, combining fast scaling-up/down with lazy scaling-out/in. It can dynamically adjust GPU provisioning and smoothly transition between vertical scaling and horizontal scaling, promoting DL serving performance. The evaluation shows that Dilu successfully delivers GPU resourcing-on-demand for serverless DL tasks, reducing fragmentation by 10-46% and boosting inference and training throughputs by 1.8× and 1.1× compared to the SOTA baseline. It also guarantees QoS by reducing 11-71% violation rate.

In summary, we make the following major contributions:

- We present an efficient binary-search-based GPU resource profiling method for training and a Hybrid Growth Search Strategy for inference tasks, where the latter speeds searching efficiency up to 3.3× compared to the SOTA method.
- We introduce a resourcing-complementary scheduling method to defragment GPU while considering QoS, significantly improving GPU utilization and increasing function deployment density.
- We co-design fast scaling-up/down and lazy scaling-in/out to achieve introspective elasticity in a cross-layer manner. It reduces the cold start rate of inference functions by 91% at most, while maintaining the lowest serving violation rate.
- We develop a prototype system of Dilu on Kubernetes and Docker, which is publicly available[1]. The cluster evaluations show that Dilu boosts inference and training throughputs by 1.8× and 1.1× compared to INFless.

## 2 Background and Motivation

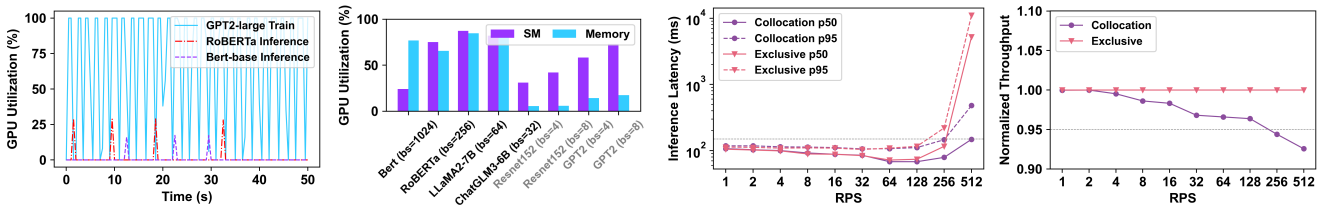
### 2.1 Serverless DL Serving

Considering manifold drawbacks (e.g., high resource consumption, complicated server operations) of server-centric DL serving, cloud providers and studies promote the serverless DL serving patterns and deliver explicit progress. Nowadays, propelled by LLMs, serverless DL serving becomes more promising to provide elastic GPU provisioning.

**Serverless Training**. Providers or developers benefit from the serverless paradigm to build ready-to-use online training services, automatic deployment of training workflows (e.g., LambdaML [26], Siren [46], Cirrus [6], FuncPipe [29], Hydrozoa [22]), and elastic scaling for training workers (e.g., ElasticFlow [18], $\lambda$DNN [50] and Dorylus [45]). Specifically, FuncPipe [29] and Hydrozoa [22] automate model partitioning for dynamic hybrid parallel training, and ElasticFlow [18] adaptively adjust serverless DP workers according to residual GPU resources.

**Serverless Inference**. Unlike training, inference is mainly online with fluctuated workloads, naturally suitable with the serverless paradigm. Since inference is more time-sensitive, studies [3, 25, 27, 35, 53, 55] pay attention to guaranteeing Service Level Objectives (SLOs, e.g., inference latency within 100ms). Additionally, studies like INFless [51] and BATCH [2] introduce batching execution to serverless to improve GPU utilization. Others [15, 42, 51] leverage layered caches and keep-alive strategies to reduce cold start overhead.

[1] https://github.com/sigserverless/Dilu.




(a) Temporal workload characteristics (b) Spatial occupying characteristics (c) Inference latency (d) Training throughput

**Figure 2.** Observations on serverless DL serving. (a)(b): GPU fragmentation in temporal and spatial dimensions. The black-color models represent training and the gray is for inference models. (c)(d): The RoBERTa-large inference latency and Bert-base training throughput comparisons under the co-scaling mechanism using 3 GPUs, relative to the Exclusive mode using 4 GPUs.

**Trends**. The serverless DL functions are more compute- or memory-bound due to the larger model sizes. Though studies [7, 15, 18, 19, 51] devote to building GPU-based serverless DL systems, they suffer from coarse-grained GPU resourcing techniques, which hinders their ability to deliver highly efficient, elastic, and cost-effective DL services.

## 2.2 GPU Resourcing

**GPU Device.** With the LLM emergence, GPUs are increasingly prominent for DL. GPUs consist of several Streaming Multiprocessors (SMs), each equipped with numerous Tensor Cores, CUDA Cores and scarce memory. High-level DL programs (e.g., based on PyTorch [31], TensorFlow [17]) are first transformed into CUDA kernels and dispatched to run on SMs. However, DL tasks vary significantly in types and sizes of kernels. For example, forward and backward propagations in training are compute-bound, while gradient synchronization is usually memory-bound caused by communication. The prefilling in LLM inference is compute-bound while decoding is memory-bound [56]. Thus, it may directly lead to the underutilization of specific GPU resources with improper task assignment at the high level.

**GPU Allocation in Cloud.** A common method is exclusively allocating the whole GPU to each DL instance, commonly seen in many serverless DL systems [7, 15, 18, 22, 54]. To further improve GPU utilization, studies leverage GPU-sharing methods to multiplex GPUs, including MPS-based spatial partition used by [10, 19, 51], virtual GPU [35], temporal methods used by [19, 20, 47] and rCUDA [54]. Studies have also explored spatio-temporal sharing methods [8, 19, 23], all of which are based on MPS. These methods require frequent adjustments of partition sizes at the process level to accommodate the highly fluctuating workloads, leading to significant time overhead. Moreover, due to the static allocation enforced by MPS, they are unable to exceed the resource limits of a single instance to handle burst workloads instantaneously. Thus, a non-negligible gap exists to support fine-grained GPU provisioning on demand in current serverless DL systems.

## 2.3 Fragmented GPU Resourcing in Serverless

With current monotonous elasticity, existing serverless DL systems are prone to produce GPU fragments. We make the following observations of the fragment sources:

**Observation-1: GPU overprovisioning.** With static GPU allocation for DL functions, each instance is easy to be overprovisioned since the resource quotas are often set empirically high or assigned in a coarse-grained manner to ensure training job completion times or meet inference SLOs. As Table 1 shows, taking the basic resource definition with *<request, limit>* in Kubernetes as an example, both Exclusive- and MPS-based allocation mechanisms can be regarded as an equal setting of *request* and *limit* quotas. Specifically, as shown in Figure 2(a), INFless allocates a constant 30% SM rate to handle RoBERTa inference with a batch size of 4 at maximum, while this SM quota cannot be reallocated to other instances dynamically, though the workload is low. Figure 2(b) shows the average GPU utilization in terms of SM and memory, which is much lower than the actual allocated GPU resource quotas, especially under the exclusive allocation [7, 18, 22].

**Observation-2: GPU idling of serverless serving.** In distributed training, each worker needs to communicate to synchronize gradients. Since this process does not consume GPU compute resources, it results in significant GPU idling time. As shown in Figure 2(a), the GPU idling time exceeds 40%, in which a 4-worker GPT2-large training task utilizes PyTorch.DDP [32] and NCCL [36]. We also fine-tune LLaMA2-7B with pipeline parallelism via DeepSpeed [34] and each worker yields nearly 20% GPU idling SM as shown in Figure 2(b). As for distributed inference, works like [15, 25, 29, 53] introduce model parallelism to serverless, which undoubtedly leads to substantial bubble time due to pipeline execution characteristics.

**Observation-3: Keep-alive strategy for serverless inference tasks.** Keep-alive is adopted in serverless DL systems [19, 27, 42, 51] to balance cold start overheads, and also becomes an important source to exacerbate fragmentation. As shown in Figure 2(a), we simulated the workload traces according to Faaswap[54], which indicates that less than 85% of functions are invoked per minute on average in *Alibaba*.

**Table 1.** Comparison of GPU provisioning in serverless. Dilu allows the definition of unequal *request* and *limit* quotas during profiling and dynamically adjusts resource provisioning based on real-time demand.

| Mechanism | Req/Lim | Allocation | Resourcing |
|---|---|---|---|
| Exclusive | equal,1 | - | static |
| MPS | equal,<1 | profiling-based | static |
| Dilu | unequal,<1 | profiling-based | On-Demand |

Two keep-alive inference function instances only handle 3-4 requests within a nearly-50s lifecycle. It means the keep-alive strategy brings over 95% of resource waste in the time dimension.

**Implications.** The GPU fragmentation and allocation limitation pull down the elasticity and deployment density [11] of serverless DL functions, increasing both user and provider costs. We argue that a proper GPU resourcing-on-demand mechanism is imperative and essential for current serverless DL serving.

### 2.4 Motivation and Challenges

**Insight: Introspective Elasticity.** Introspective Elasticity (IE) is a specialized mechanism for GPU resourcing-on-demand. It refers to a holistic and novel GPU provisioning paradigm tailored for serverless DL functions, which provides fine-grained, continuous and adaptive GPU resources. Unlike current horizontal-only elasticity in serverless [19, 51], which merely provides discrete GPU provisioning and focuses solely on eliminating external resource fragments, IE expands it by dynamically multiplexing internal GPU fragments of instances according to real-time kernel-level workloads, to maximize GPU utilization.

IE requires the system to identify static GPU fragments, which provides the opportunity to eliminate them with the collocation method. It also emphasizes the need to efficiently reuse dynamic fragments generated by varying workloads. More importantly, these two capabilities ought to be coordinated to effectively manage bursty workloads.

**Preliminary Verification.** We verify this idea through a toy experiment using Exclusive and Collocation setups with serverless functions: the Exclusive setup involves 4 GPUs, 3 GPUs for training and 1 GPU for inference. In contrast, the Collocation mode occupies only 3 GPUs, each collocating a training worker and an inference worker (vertical scaling). For the collocation case, requests are loaded balance to 3 inference workers (horizontal scaling). The results in subfigures 2(c) and 2(d) show that on top of saving 25% of GPU resources, the co-scaling mechanism effectively improves inference throughput by 46% with QoS guarantees while merely decreasing throughout of the collocated training task by 5.2% at RPS=256.

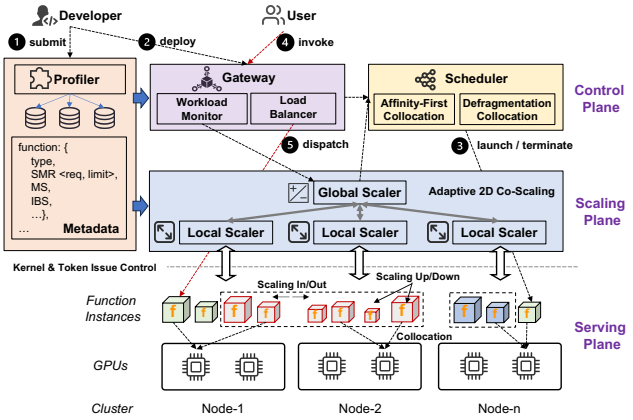



**Figure 3.** The system architecture of Dilu.

**Challenges.** IE is carried out by essential co-scaling but builds on indispensable profiling and scheduling. However, building such a serverless DL system with IE support is non-trivial. ***Challenge 1:*** It is costly and difficult to precisely measure the basic GPU resource quotas required to guarantee QoS for each DL function. The profiling result helps prevent overprovisioning and converts previously internal GPU fragments of DL functions into external resources for potential reassignment. Diverse factors, including DL function types or priorities, varying model sizes, and complex execution patterns (e.g., batching for inference), exacerbate the profiling sampling spaces. ***Challenge 2:*** It requires wise scheduling to reuse GPU fragments and make collocation decisions efficiently. The real-time fluctuations of function workloads and the cluster's resource status expand the scheduling and collocation search space, making it challenging to achieve multiple objectives simultaneously (e.g., defragmentation, QoS guarantees). ***Challenge 3:*** A deliberate and cross-layer coordination mechanism is needed to handle resource contention of collocated tasks. For example, under the high RPS (e.g., 256 and 512), there exist serious inference SLO violations in Figure 2(c) and training throughput decrease in Figure 2(d), caused by blunt high-level horizontal scaling and disordered low-level vertical scaling.

## 3 System Design

This section outlines the design of Dilu. We first describe the system's execution workflow (Section 3.1), followed by detailed explanations of key components. Section 3.2 profiles resources to identify available GPU fragments. Section 3.3 involves GPU allocation principles from the global scheduler's perspective. Lastly, Section 3.4 presents the core 2D co-scaling mechanism for GPU resourcing-on-demand.

### 3.1 Architecture Overview

Dilu is a serverless DL system with introspective elasticity support for GPU resourcing-on-demand. It collocates DL functions with resource-complementarity concerns to reduce

GPU fragments, and adaptively adjusts GPU provisioning to guarantee QoS. The system architecture is depicted in Figure 3. It consists of a control plane, a scaling plane, and a serving plane.

**The Control Plane.** It takes charge of DL task profiling, deploying, scheduling and request dispatching. Users submit DL function programs with pre-defined QoS descriptions to the system ❶. Specifically, we generally consider training throughput and inference latency as QoS objectives. The profiler acquires resource plans with pruning-search trials, as resourcing metadata referred by scheduling. After profiling, developers deploy functions to the gateway ❷, which then forwards them to the scheduler. The scheduler manages instance deployment requirements of DL tasks adhering to several principles. The gateway dispatches all internal and external API requests to the target modules.

**The Scaling Plane.** It mainly provides an adaptive 2D co-scaling service in horizontal and vertical dimensions, delivering a practical introspective elasticity for GPU resourcing-on-demand. The global scaler informs the scheduler to carry out horizontal scaling of DL functions, including launching and terminating instances ❸. The local scaler is distributed in each GPU server. It dynamically adjusts the compute resource of functions by resizing up or down the SM quotas to ensure QoS and improve GPU utilization. Putting them together, as the external invocation workloads increase ❹, the two-layer scaling will initiate a fast scaling-up and a lazy scaling-out process to deal with bursty workloads and reduce cold starts. Conversely, while the workloads decrease, a fast scaling-down joint with lazy scaling-in will be triggered.

**The Serving Plane.** The DL functions are running as instances in the serving plane with shared GPUs and other cloud resources, to handle dispatched requests ❺. The functions may employ multiple GPUs or servers for large-scale LLM computations.

### 3.2 Multi-Factor Profiling

Dilu profiles three key resourcing factors of DL tasks, including the GPU SM rate (SMR), memory size and inference batch size (IBS), for ensuring QoS. The SMR directly influences training throughput and inference latency. The memory size is usually a constant due to the memory pool management in DL frameworks. The IBS plays a major role as the request batching can improve throughput significantly [2, 3, 8, 51].

Inspired by the resource quota definition like CPU and host memory in Kubernetes[1], Dilu adopts a similar *<request, limit>* mechanism for SMR quotas, while memory size remains the same due to its steady demand. The *request* quota denotes the minimum compute resource requirement to ensure QoS (e.g., 80% exclusive training throughput, <100ms inference SLO) and the *limit* indicates the optimal cost-effective quota value, approximately reaching marginal effect points (e.g., near-exclusive training throughput with least compute resource, adaptive for burst inference workloads with SLO ensurance). Leveraging this mechanism, actual allocated SMR can be continuously adjusted between *request* and *limit*, effectively avoiding task starvation and GPU overprovisioning.

**Table 2.** Inference function profiling comparison for models (a)-(d) as illustrated in Figure 4. The number represents profiling iterations, approximately 30s per trial.

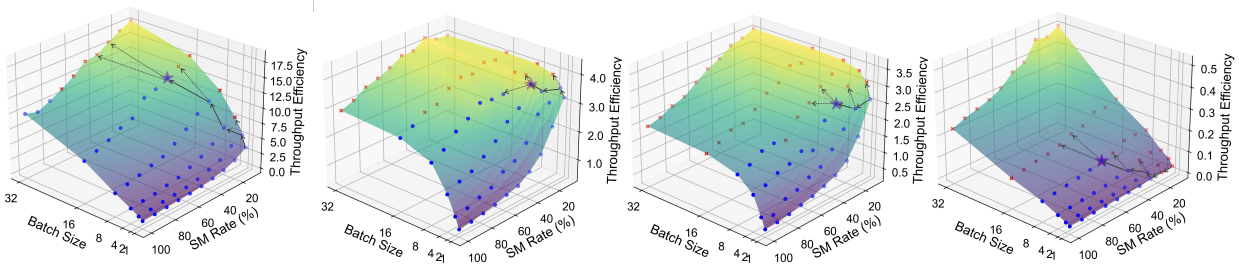
| Baseline | a | b | c | d | Method |
|---|---|---|---|---|---|
| Traversal | 60 | 60 | 60 | 60 | pre-running |
| INFless[51] | 20 | 40 | 40 | 30 | prediction |
| GPUlet[8] | 16 | 16 | 16 | 16 | pre-running |
| **Dilu** | **8** | **6** | **6** | **9** | pre-running |

**Training Profiling.** Dilu employs a binary search method to iteratively seek the *request* and *limit* quotas of SM. The profiler records exclusive throughput $T_1$ with *high* (=100%, firstly) SMR and $T_2$ with *mid* (=50%) SMR quota ($low = 0$ indicates zero throughput and is therefore omitted). If $T_2$ is less than $T_1 * p$, indicating underprovisioned GPU compute resources, the *low* value is set to *mid*. Otherwise, the *high* value is set to *mid*. The profiling ends until the $T_i$ satisfies $T_1 * p \pm 2\%$. The SMR for $T_i$ serves as the *request* quota when $p$ is set to 80% and the *limit* quota when $p = 100\%$.

**Inference Profiling.** We adopt a novel Hybrid Growth Search Strategy to search the most cost-efficient settings of <SMR, IBS>. Although SMR is positively related to throughput, we observe marginal effects when increasing it, as shown in Figure 4. For example, there is merely a 2% throughput boost for RoBERTa-large model with IBS = 4, while increasing SMR doublely from 50% to 100%. Thus, we introduce the throughput efficacy (TE) metric, defined as $\text{TE} = \frac{\text{Throughput}}{\text{SMR}} = \frac{\text{IBS}}{t_{\text{exec}} \cdot \text{SMR}}$[2] to denote throughput per SM unit. As illustrated in Figure 4, all the tested models form a convex surface in the three-dimensional space of $\langle \text{IBS}, \text{SMR}, \text{TE} \rangle$. With this strategy, IBS iteratively increases by doubling during profiling, while SMR increases linearly by a fixed rate (i.e., 10 units). While some untested models may not conform to the expected convex surface, the QoS is assured since the search process is established on the premise of meeting SLOs. Finally, the star in each subfigure of Figure 4 contains the optimal SMR, marked as the *request* quota. We empirically set the *limit* quota at twice of *request* to accommodate bursty workloads.

**Profiling Efficiency**. As Table 2 demonstrates, Dilu outstands all baselines in search iteration times, 0.7-1.7× speedup compared with the traversal method, and 1-3.3× to the SOTA GPUlet [8]. INFless [51] may sustain lower accuracy due to model decomposition and operator time prediction.

[2]We adhere to $t_{\text{exec}} = \text{SLO}/2$ like INFless [51], to account for additional overheads caused by communication, batching waiting, preprocessing, etc.




(a) Resnet152 (b) RoBERTa-large (c) GPT2-large (d) LLaMA2-7B

**Figure 4.** Throughput efficiency under varying SM rates and batch sizes for inference models with specific SLO targets. The star indicates the optimal configuration pair <IBS, SMR>, blue points denote configurations meeting SLOs, red crosses indicate SLO violations, and the black solid line shows the forward path with the dashed line representing blocked paths.

### 3.3 Resourcing-Complementary Scheduling

The scheduler manages the GPU allocation plans of DL functions at the cluster level, and makes collocation decisions. The primary goal is to improve GPU cluster utilization and aggregate throughput, increasing the deployment density [11] of DL tasks with QoS guarantees. The objective is formalized in Equation 1, minimizing the number of GPUs used, where $n$ represents the number of GPUs in the cluster, and $g_i = 1$ indicates GPU $i$ is occupied, otherwise idle. Constraint 2 ensures that each function instance $f_j$ is allocated to at least one GPU, where $m$ denotes the number of all instances and $f_{ij} = 1$ indicates $f_j$ occupies GPU $i$. Constraint 3 ensures that the execution time of $f_j$ (compute resource requirements) meets the corresponding QoS $Q_j$. Constraint 4 ensures that the total memory usage of collocated instances remains within the limit of a single GPU card. Constraint 5 ensures that GPU $i$ is marked as occupied if it is assigned any $f_j$.

$$min \quad \sum_{i=1}^{n} g_i \tag{1}$$

$$s.t. \quad \sum_{i=1}^{n} f_{ij} \geq 1, \ \forall j = 1, \ldots, m, \tag{2}$$

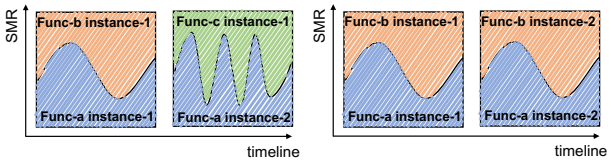
$$t(f_j) \leq Q_j, \ \forall j = 1, \ldots, m, \tag{3}$$

$$\sum_{j=1}^{m} M(f_{ij}) <= M(g_i), \tag{4}$$

$$g_i = 1 \ \text{ if } \ \sum_{j=1}^{m} f_{ij} \geq 1 \ \text{ else } \ g_i = 0, \tag{5}$$

$$f_{ij} \in \{0, 1\}, \ \forall i = 1, \ldots, n, \forall j = 1, \ldots, m. \tag{6}$$

The scheduling can be regarded as a 2D bin-packing problem, by coordinating SMR (with IBS) and memory size. It is a well-known NP-complete problem, hence Dilu adopts a heuristic greedy Algorithm 1 to reduce complexity, which adheres to the following three principles.




(a) Without workload affinity (b) With workload affinity

**Figure 5.** Workload-affinity effect comparison.

**Principle-1: Reducing laggers with affinity-first collocation.** Each DL function may differ in its characteristics and workload, and impact the SM consumption accordingly. Random scheduling may lead to situations shown in Figure 5(a), where the resources allocated to instances of training Func-a vary differently. It results in a severe barrel effect since the training speed depends on the lagger instance with the least compute resources. Considering the runtime affinity of functions, Dilu strategically collocates instances with similar workloads (line 11-12) to reduce the impact of the barrel effect. Specifically, instances of the same function (e.g., instance-1/2 of Func-a) often keep analogous loads or SM consumption trends. Thus, as Figure 5(b) shows, Dilu tends to collocate the running instance-2 of Func-a with newly launched instance-2 of Func-b, instead of conflicting instances of Func-c. Thus, it mitigates the impact of the barrel effect and prevents elastic scheduling failures.

**Principle-2: Defragmentation through Resource Complementarity.** Dilu considers both SM and memory resources to maximize GPU utilization (the *SelectOptGPU* function in lines 19-29). The *bestScore* corresponds to the GPU with the minimum weighted fragmentation. For models that fit within a single GPU fragment, we employ a best-fit strategy. For larger models (e.g., LLMs) that exceed a single GPU fragment, we adopt a memory-based worst-fit strategy which prioritizes choosing GPUs with the most remaining memory, to minimize pipeline stages and reduce end-to-end latency. If no GPU fragments are available, a new GPU instance will be allocated (line 15-16).

**Algorithm 1** Heuristic GPU Scheduling Algorithm

1: **Input:**
2: $G_{act}$: Active GPUs with at least one deployed function.
3: $m_j, sm_j^{req}, sm_j^{lim}, n_j$: Resource requirements and number of GPUs needed for function $F_j$.
4: $\Omega, \gamma, M_i$: Max allowable sums of SM requests and limits ratios, and memory on each GPU.
5: **Output:**
6: $I^*$: Set of optimal GPUs for deployment.
7: **function** ScheduleInstances
8: **while** True **do**
9: Accept the deployment request for $F_j$ and initialize $I^* \leftarrow \emptyset$.
10: **for** $k \leftarrow 1$ **to** $n_j$ **do**
11: $G_{WA} \leftarrow$ GPUs hosting instances with high workload-affinity.
12: $i^* \leftarrow$ SelectOptGPU($G_{WA}, sm_j^{req}, sm_j^{lim}, \Omega, \gamma$)
13: **if** $i^* == -1$ **then** ▷ Select from the GPUs without WA
14: $i^* \leftarrow$ SelectOptGPU($G_{act} \setminus G_{WA}, sm_j^{req}, sm_j^{lim}, \Omega, \gamma$)
15: **if** $i^* == -1$ **then** ▷ No available active GPU
16: $i^* \leftarrow$ Start a new GPU instance
17: Update resource info of $G_{i^*}$ for $sm_j^{req}$ and $sm_j^{lim}$
18: $I^* \leftarrow I^* \cup \{i^*\}$
**return** $I^*$
19: **function** SelectOptGPU($G_{candidates}, sm_j^{req}, sm_j^{lim}, \Omega, \gamma, m_j$)
20: bestScore, $i^* \leftarrow \infty, -1$
21: **for** each $i$ in $G_{candidates}$ **do**
22: $newReqSum \leftarrow \sum_{k \in \text{Funcs on } i} sm_k^{req} + sm_j^{req}$
23: $newLimSum \leftarrow \sum_{k \in \text{Funcs on } i} sm_k^{lim} + sm_j^{lim}$
24: $newMemUsage \leftarrow \sum_{k \in \text{Funcs on } i} m_k + m_j$
25: $score \leftarrow \alpha \cdot \left(1 - \frac{newReqSum}{SM_i^{total}}\right) + \beta \cdot \left(1 - \frac{newMemUsage}{M_i}\right)$
26: **if** $newReqSum \leq \Omega$ and $newLimSum \leq \gamma$ and $newMemUsage \leq M_i$ and $score <$ bestScore **then**
27: bestScore $\leftarrow score$
28: $i^* \leftarrow i$
29: **return** $i^*$

**Principle-3: Balancing the oversubscription and QoS guarantees.** Higher oversubscription leads to increased function density but will cause severe performance interference. Given the QoS guarantees, two parameters, $\Omega$ and $\gamma$ (line 26), are used respectively to restrict the maximum sum of *request* and *limit* quotas provided per GPU. We conservatively set these hyperparameters (e.g., $\Omega = 1$, $\gamma = 1.5$) to minimize the effects of oversubscription.

## 3.4 Adaptive 2D Co-Scaling

Dilu introduces an adaptive two-dimensional co-scaling mechanism, which provides dynamic and fast vertical provisioning (i.e., scaling up/down) elasticity and cost-effective and lazy horizontal elasticity (i.e., scaling in/out). Compared with horizontal-only scaling in classic serverless computing, it strengthens GPU resourcing-on-demand and provides a smooth transition to handle fluctuating workloads, minimizing the impact of cold starts. Specifically, GPU compute provisioning quotas have shifted from traditional discrete integers to continuous decimals (unit is # of GPU).

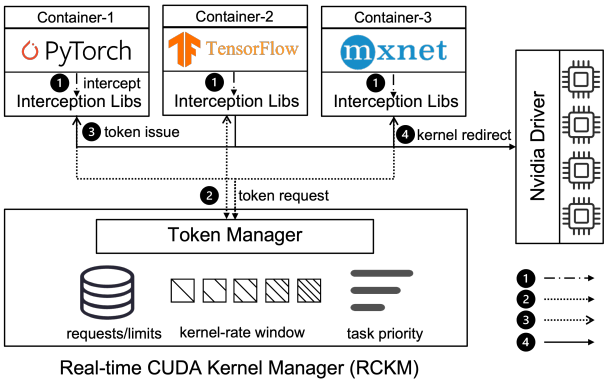



**Figure 6.** The vertical scaling workflow in Dilu.

### 3.4.1 Dynamic and Fast Scaling Up/Down.

The vertical scaling is responsible for throttling compute resources (i.e., SMs), and has two objectives: to support GPU resourcing-on-demand at the intra-GPU level, and to facilitate a transition to bulky horizontal scaling, as discussed in Section 3.4.2.

Since GPU drivers (e.g., NVIDIA) are often closed-source, direct SM management is unfeasible. Inspired by the monitor-and-control mechanisms used in [19, 20, 47, 52], Dilu overall adopts a **server** (Real-time CUDA Kernel Manager, RCKM) **-client** (Interception Library, IL within each container) architecture to *indirectly* manage SM consumption of each instance, as illustrated in Figure 6. They cooperate to monitor and restrict the launched CUDA kernels of collocated instances that shares a single GPU. The workflow mainly consists of the following processes:

- **Kernel Intercept**: CUDA kernel calls emitted by each instance from the host CPU are first intercepted by the Interception Library into respective queues via Linux's LD_PRELOAD mechanism.
- **Token Request**: Token represents available GPU time for each co-located instance per period. To forward kernels, IL first asks for tokens from the RCKM server periodically (e.g., 5ms).
- **Token Issue**: RCKM issues tokens to IL based on several factors, including<*request, limit*> quotas, task priority, and continuous execution cycles. We discuss it on Algorithm 2 in detail.
- **Kernel Redirect**: IL determines to block (# tokens < current kernel counts) or release (# tokens >= current kernel counts) CUDA kernels for execution.

The Algorithm 2 details introspective vertical elasticity, explaining how dynamic GPU provisioning works for each DL instance. The motivation stems from observations via Nsight System [39], where we note that SM contention would prolong the kernel launching cycle (KLC) time within an iteration[3], e.g., from 25ms to 50ms for RoBERTa-large inference.

[3]We consider one iteration to encompass a forward and backward propagation for training tasks, as well as a single batch execution for inference tasks.

Therefore, the core idea is to dynamically allocate GPU tokens for collocated instances on the same GPUs based on KLC changes of high-priority task instances, without monitoring reactive metrics from the application layer.

Specifically, when the calculated KLC increases significantly (line 14), it implies either a bursty workload of itself or overly aggressive GPU provisioning for its collocated instances. Then RCKM sets the global variable *state* to *EMERGENCY* and **fast** resizes up its issued tokens (line 15). Accordingly, the collocated instances are temporarily resized down based on the KLC variation (line 26-27). Only the current instance can reset or modify the *EMERGENCY* state. Next, we explain the **dynamicity**. If no kernels have been launched recently, then the instance will scale down (line 16-17). And if its collocated instances have not launched kernels recently, its quotas will gradually be increased (line 18-19). Otherwise, RCKM will maintain the current provisioning plans in the relatively stable *CONTENTION* state.

**Discussion**. Different from previous spatio-temporal GPU sharing methods in [8, 19, 23], which all depend on static MPS [38] allocation technique, Dilu enables basic spatial sharing via collocation and fine-grained token issuing management based on pre-profiled *<request, limit>* meta-quota to avoid task starvation and GPU overprovisioning. In the temporal dimension, it dynamically allocates tokens between *<request, limit>* from a global perspective (RCKM), ensuring high utilization of the entire GPU. Moreover, Dilu relies on the isolation properties of containers for security, providing reliable protection for multi-tenant services. As for fault tolerance, Dilu leverages the classic restart strategy in serverless.

**Algorithm 2** Fast Scale-up/down Control Algorithm

```
1:  Input:
2:    MaxTokens: Maximum number of tokens that can be issued.
3:    RW: Kernel rate windows for instances.
4:    R_current: Current kernel execution rate.
5:    request, limit: Request/limit rate for the instance.
6:    Type: Type of the instance (e.g., SLO-sensitive).
7:    T_current, T_min: Current and minimum recorded KLCs.
8:  Output:
9:    R_issue: Issued tokens of the instance during this cycle.
10: function IssueToken
11:   Shift Rate Window RW[current] with R_current.
12:   if Type is SLO-sensitive then
13:     ΔT = (T_current − T_min) / T_min          ▷ Calculate the relative change
14:     if ΔT > η_violation then                  ▷ Trigger protective logic, scale up
15:       state, R_issue ← EMERGENCY, MaxTokens * limit
16:     else if sum(RW[current]) == 0 then        ▷ Scale down
17:       state, R_issue ← RECOVERY, MaxTokens * request
18:     else if sum(RW[others]) == 0 then         ▷ Scale up
19:       state, R_issue ← RECOVERY, R_last * η_increase
20:     else
21:       state, R_issue ← CONTENTION, MaxTokens * request
22:   else
23:     switch state do
24:       case NONE:                              ▷ Without collocation instances
25:         R_issue ← MaxTokens * limit
26:       case EMERGENCY:                         ▷ Scale down
27:         R_issue ← min(MaxTokens * request, R_last)/ΔT
28:       case RECOVERY:                          ▷ Scale up
29:         R_issue ← min(R_last * η_increase, MaxTokens * limit)
30:       case CONTENTION:
31:         R_issue ← R_last
```

**3.4.2 Lazy Scaling Out/In.** Dilu also facilitates introspective horizontal scaling, effectively integrating it with vertical scaling to amplify system resilience.

Classic horizontal-only scaling (e.g., FaST-GS [19]) launches or terminates function instances *reactively* as workloads increase or drop. Advanced approaches (e.g., INFless [51] and Azure Serverless [42]) calculate pre-warming and keep-alive duration based on prior knowledge to associate workload prediction with pre-provisioning. However, due to short-term unpredictability, these strategies still incur significant cold start overheads caused by the slow and bulky deployment of large DL functions. It also leads to severe SLO violations. Instead, Dilu strengthens horizontal scaling in cooperation with fast vertical scaling to manage the burst workloads.

Specifically, Dilu handles sudden workload increases by first triggering fast scaling-up, adaptively delaying the scaling-out execution and avoiding cold starts caused by few-second-level bursty requests. The global scaler maintains a sliding window (i.e., size=40s) for each function to guide the horizontal scaling. If at least $\phi_{\text{out}}$ RPS values within the window exceed the serving throughput of deployed instances (acquired from profiling), the scaler informs the scheduler to launch new instances. Here, $\phi_{\text{out}}$ (i.e., 20) indicates a relatively stable high workload. Conversely, to avoid frequent restarts of new instances, a scaling-in decision is only triggered if more than $\phi_{\text{in}}$ (i.e., 30) RPS values in the window fall below the serving throughput of (# of instances - 1). Notably, the lazy scaling-out only occupies a small portion of GPU memory (as shown in Figure 2(b)). Due to the fast scaling-down mechanism mentioned above, the idle SMs can be dynamically reallocated to other collocated instances.

## 4 System Implementation

**Prototype system**. We have implemented a prototype system based on Docker and Kubernetes, with 5k lines of code, including 2k+ python LOC for the scheduler and 3k+ C LOC for the scaler. Additionally, we have developed 3k+ lines of code for simulation, evaluation and scripts. The profiler, scheduler, and global scalers are all deployed within containers. Dilu is compatible with any CUDA-based programming framework like [14, 17, 31].

**Serverless DL Functions**. In the system, we build a DL function with model-parameter files, an execution entry script and groups of DL runtimes, including Pytorch, transformers, or deepspeed libraries. We also pack IL with each DL function and add its path to the */etc/ld.so.preload* file.

**Profiler**. It receives DL function images provided by developers. The profiling script varies SMR via *CUDA_MPS_ACTIVE-_THREAD_PERCENTAGE* of MPS [38] API to allocate different compute resources for each pre-running instance.

**Scaler**. The local vertical scaler on each node establishes a *Unix domain socket* with IL hooked into each instance. It then receives accumulated kernel counts and sends tokens. IL intercepts *cuLaunchKernel*,*cuLaunchCooperativeKernel*, and other related APIs. Notably, both tokens and kernels are measured in units of *CUDA kernel blocks*. Each GPU device is managed by a separate *POSIX thread*. The global horizontal scaler periodically (e.g., every 1 second) retrieves workload information from the Application Layer Gateway.

**Scheduler**. The scheduler accesses the resource requirement metadata of each function from the profiler and allocates *ip_address, GPU index, port* for launched instances. For distributed DL function instances, we leverage *NCCL [36]* to communicate and adopt the *accelerate [13]* library for the LLM model partition. All instances run on the *NVIDIA Docker* runtime.

# 5 Evaluation

## 5.1 Methodology

**Experiment testbed**. We set up an environment of a self-hosted, five-worker Kubernetes (v1.23.4) cluster, each equipped with 4 NVIDIA A100-40GB-PCIE GPUs. Each node is based on PyTorch v1.11, DeepSpeed v0.11.1, NCCL v2.10.3, NVIDIA driver v515.105.01, CUDA v11.7 and Docker v24.0.5.

**Workloads**. Several popular DL models are selected for evaluation, including ResNet152 [24], VGG19 [43] from computer vision, and BERT-base [9], RoBERTa-large [28], GPT2-large [40] from natural language processing, and LLAMA2-7B [30], ChatGLM3-6B [12] of the thriving LLM family. The model parameters range from 0.2GB to 12.6GB.

For training, we adopt the torch.DDP [32] for medium- and small-size models, and DeepSpeed pipeline-parallelism [34] for LLM fine-tuning. For inference, several workload patterns are considered, including Poisson distribution (used by [2, 3, 44, 56]), Gamma distribution (used by FastServe [48]), and three typical traces from Azure Function's Production Traces [42]: Bursty, Sporadic, and Periodic (used by INFless [51]).

To study the large-scale performance (Section 5.5), we simulate a cluster of 1,000 nodes, each equipped with 4 GPUs. We generate 3,200 DL instances of varying types in the cluster, with the distribution of training, LLM inference, and non-LLM inference instances by a ratio of 2:2:6.

**Metrics.** Training throughput (e.g. images/s of computer vision models, tokens/s of natural language processing models), latency (e.g., p50/p95) and SLO violation rate (SVR), cold start count (CSC) of inferences are measured. For the LLM inferences, the average time-per-output-token of requests is regarded as LLM latency.

**Baselines**. They are chosen for both GPU- and cluster-level comparisons. At the GPU level, the baselines include:

- **Exclusive**: All GPUs are allocated exclusively to DL function instances via pass-through.
- **NVIDIA MPS** [38]: The official NVIDIA GPU sharing mechanism is widely used in serverless DL systems [10, 19, 51]. **MPS-l** (MPS with *limit* quotas from Dilu profiling) and **MPS-r** (MPS with *request* quotas from Dilu profiling) are used for comparisons.
- **FaST-GS** [19]: A typical spatio-temporal GPU sharing method, specifically designed for serverless DL inference, relies on MPS. For fairness, we allocate the same amount of SMR spatially as MPS-l.
- **TGS** [47]: A transparent GPU sharing method, improving opportunistic job throughput while guaranteeing productive jobs.

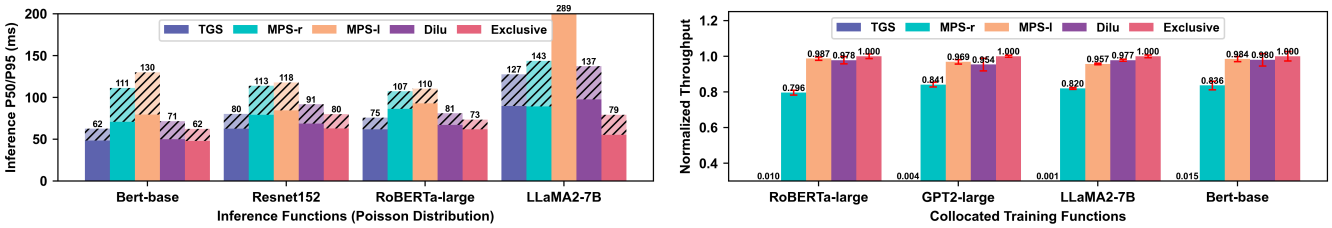
At the cluster level, the baselines include:

- **Exclusive**: It is used by [7, 18, 22, 33] to allocate GPUs exclusively for DL functions, which is a common basic scheme in Kubernetes.
- **FaST-GS+** [19], **INFless+** [47]: Two serverless inference systems based on MPS improve serving throughputs. We extend them to support training scheduling as FaST-GS+ and INFless+.

## 5.2 Vertical Scaling Performance

**High GPU utilization and aggregate throughput**. We analyze the vertical scaling performance in Dilu on three typical collocation cases. Experiments show that Dilu effectively achieves GPU resourcing-on-demand, dynamically adjusting resource provisioning while ensuring QoS. Dilu outperforms all the GPU sharing baselines, and is close to the Exclusive mode which occupies more GPUs and generates an amount of resource fragmentation.

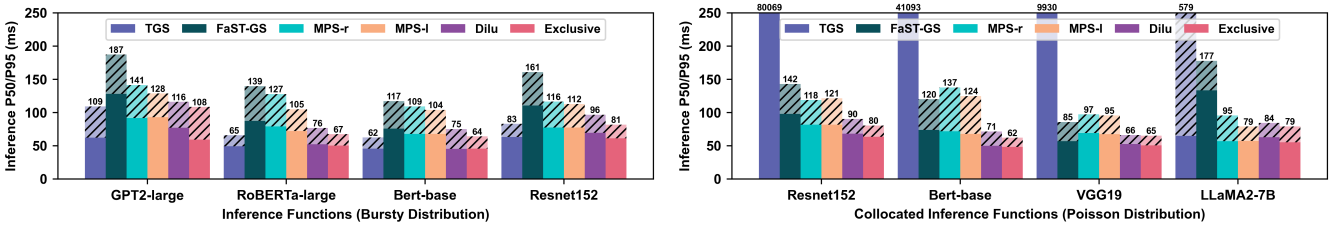
**Training-inference collocation**. As shown in Figure 7, compared to the Exclusive, Dilu achieves only 1.24× p50 and 1.28× p95 latencies on average along with 97.2% Exclusive' throughput, while saving 50% of GPU resources. TGS shows similar inference performance and sub-optimal p95 latency of all, but it nearly stops the collocated training functions. Because TGS simply prioritizes the high-priority inference instance to execute first, slowly and incrementally increases execution opportunities for another low-priority instance through trial. It may not lead to starvation, but it lacks the mechanism of adapting to highly hybrid and fluctuated DL workloads. MPS-r results in higher inference tail latencies and lower training throughput due to its static and conservative resource provisioning. Compared to the average p50 and p95 latencies of MPS-l and MPS-r, Dilu obtains reductions of 35% and 25%, and 13% and 21%, respectively. Especially, the LLaMA2-7B inference instance is deployed using four fragmented GPUs, except the Exclusive baseline. We can see




(a) Inference latency: the dark and light bars resprsent p50/p95, and the mean RPS from left to right is 35, 20, 10 and 3, respectively.

(b) Collocated training throughput: error bars represent standard deviation.

**Figure 7.** Training-Inference collocation performance.




(a) Inference latency under the bursty distribution, and the scaling factor of the initial burst workload from left to right is 4,6,6,4 respectively.

(b) Collocated inference latency under the Poisson distribution, and the mean RPS from left to right is 20, 30, 20 and 3, respectively.

**Figure 8.** Inference-Inference collocation performance.

that Dilu balances training throughput and inference latency, achieving the highest aggregate performance.

**Inference-inference collocation**.

In this case, Dilu still keeps the best performance and consistently attains higher GPU utilization than other baselines. The inference performance in Figure 8(a) almost aligns with Figure 7(a), with the only difference that MPS-l performs better than MPS-r due to less resource contention in current collocation mode. In Figure 8(b), TGS's average p50 and p95 latencies are 442× and 405× of Dilu respectively, due to its conservative and speculative sharing mechanism previously mentioned. Compared to MPS-l in Figure 8(b), Dilu reduces the average p95 by 25% through fast vertical scaling to manage bursty workloads. In Figure 8(b), the p95 latency of LLaMA2-7B inference with Dilu is slightly higher (less than 6%) than MPS-l, due to the relatively fair token issuing to ensure SLOs of both collocated instances. Additionally, we include comparisons with FaST-GS, a spatio-temporal sharing mechanism. Since it relies on MPS, its performance upper limit matches that of MPS-l only when compute resources are not temporally shared. However, frequent collection of CUDA Event time statistics and the prioritized dequeuing mechanism for temporal sharing introduce significant overhead, leading to higher latency than MPS-l in most cases, as shown by the green bars. This gap becomes negligible for smaller models, such as Bert-base and VGG19.

**Training-training collocation**. On average, Dilu achieves 176% of the aggregate training throughput of Exclusive, outperforming all other baselines. As shown in Figure 9, Dilu is 10%-14% and 3%-14% higher than MPS-l and MPS-r. We

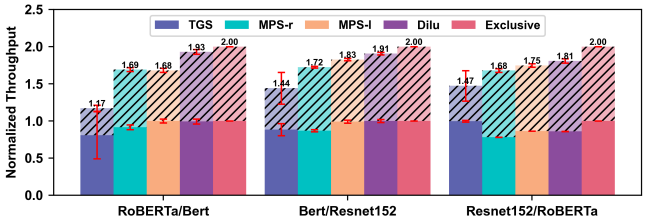
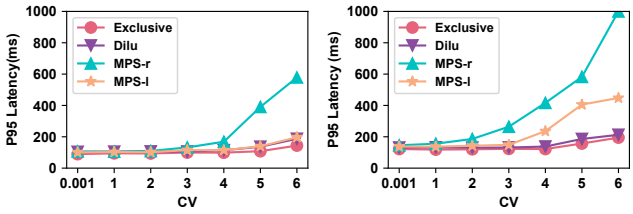



**Figure 9.** Training-Training collocation performance: the bottom bars represent the left models.

also observe intense SM contention in the RoBERTa and BERT collocation case, resulting in lower throughput for MPS-l compared to MPS-r. TGS still prioritizes high-priority tasks, but the performance of collocated low-priority tasks is severely affected.

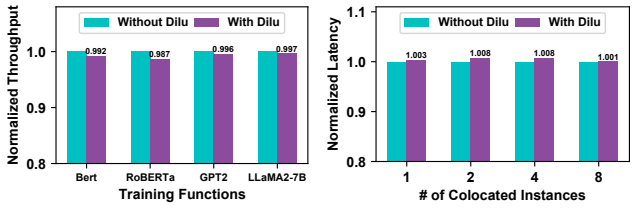
**Fast adaptivity.** Under fluctuating Gamma distribution workloads, Dilu guarantees SLOs like the Exclusive setup and surpasses other baselines, thanks to the fast scaling-up capability of RCKM. The p95 latencies of Dilu on the smaller RoBERTa-large model are only 6%-29% higher than the Exclusive in Figure 10(a), and 7%-9% higher on the bigger GPT2-large model in Figure 10(b). However, MPS-l and MPS-r both show an exponential growth trend as the Coefficients of Variation (CV) increase due to static resource provisioning. Specifically, at CV=6, the p95 of MPS-l and MPS-r are 2.08× and 4.76× higher respectively than Dilu, while Dilu is only 9% higher than Exclusive.

**Adaptive kernel issuing**. In Figure 13, we explore kernel issuing traces of <RoBERTa-large, LLaMA2-7B> case in Figure 7 and <GPT2-large, RoBERTa-large> case when CV = 5




(a) RoBERTa-large with RPS=64 (b) GPT2-large with RPS=48

**Figure 10.** Inference latency under gamma distributions, collocated with Bert-base and RoBERTa-large training instances, respectively.




(a) Training Overhead (b) Inference Overhead

**Figure 11.** Vertical scaling overhead.

in Figure 10(b). When the workload is low (average 10 req/s, as shown in Figure 13(a)), Dilu maintains a low normalized kernel issuing ratio for the inference instance, allowing the collocated training to utilize more SMs. However, MPS-r still keeps a relatively high ratio for inference, resulting in an overall training throughput decrease of 15% than Dilu. Figure 14 further illustrates total issuing kernel counts in this case, where the purple trace indicates that Dilu maintains the highest GPU utilization. Under the fluctuating workload shown in Figure 13(b), Dilu consistently provides more tokens than MPS-r, which explains why the p95 of MPS-r is 3.1× higher than Dilu.

**Negligible vertical scaling overhead.** Figure 11(a) demonstrates that the average throughput loss for different training models with vertical scaling is below 1%. Similarly, Figure 11(b) reveals negligible overhead in inference performance, even as the number of managed instances on a single GPU increases.

**Sensitivity analysis**. Figure 18(b) shows the impact of the *MaxToken* size in RCKM on DL performance. A conservative setting severely affects the collocated instances, while an excessively high setting causes interference, particularly in inference tasks.

### 5.3 Co-Scaling Performance

**Low SLO violation rate and few cold starts**. As shown in Table 3, Dilu stands out with the co-scaling support, achieving the lowest SVR and the fewest CSC, while significantly saving GPU costs and ensuring QoS. Specifically, Dilu achieves an average SVR of 4.7%, reducing CSC by 75%-77% and

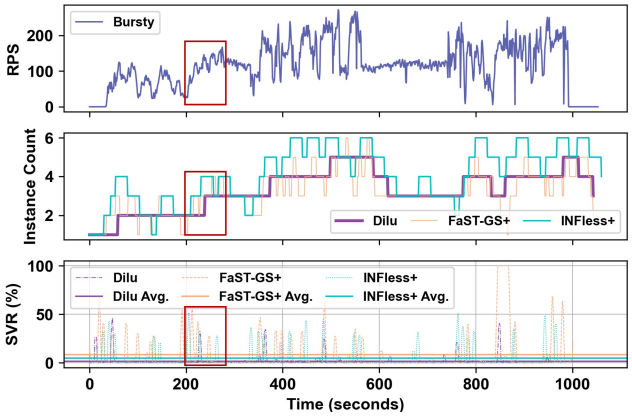



**Figure 12.** Trace analysis on co-scaling performance. SVR denotes the SLO violation rate.

**Table 3.** Horizontal scaling performance. CSC represents cold start counts. SVR denotes SLO violation rate. SGT means saved GPU time.

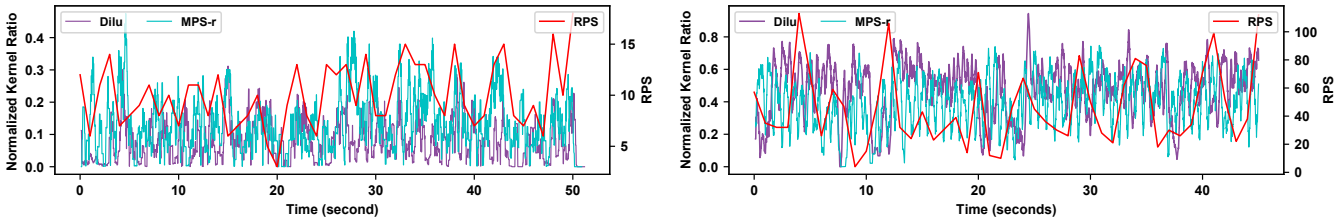
| Trace | Baseline | CSC | SVR(%) | SGT |
|---|---|---|---|---|
| Bursty | FaST-GS+ | 40 | 10.79 | 715.4s |
| | INFless+ | 27 | 6.28 | 433.6s |
| | Dilu | **7** | **1.79** | - |
| Periodic | FaST-GS+ | 41 | 19.25 | 650.4s |
| | INFless+ | 27 | 11.09 | 346.8s |
| | Dilu | **11** | **9.85** | - |
| Sporadic | FaST-GS+ | 4 | 7.57 | 65.0s |
| | INFless+ | 11 | 5.17 | 216.8s |
| | Dilu | **1** | **2.33** | - |

SVR by 46%-67%, compared to INFless+ and FaST-GS+, respectively. FaST-GS+'s eager scaling-out strategy and static scaling-up capability result in a high SVR. INFless+ slightly reduces CSC using prior knowledge but maintains a group of keep-alive instances, leading to substantial GPU waste.

**Smooth transition.** The co-scaling mechanism brings a smooth transition to guarantee inference SLOs. To investigate further, we record and analyze the serving trace under bursty workloads, as shown in Figure 12.

At certain periods, e.g., framed 200-240 seconds, there exists a workload surge as the top subfigure shows. The fast scaling-up capability provided by RCKM ensures sufficient resource provisioning for these large number of instantaneous requests, thereby securing enough response time to scale out new instances, as indicated by the increase of instance count around 225 seconds in the medium subfigure.

**Low horizontal scaling overhead.** Regarding the scheduling overhead, Dilu generates scheduling decisions for 3,200 instances concurrently within 1.12 seconds. For real-world




(a) Case-1: low inference workloads (b) Case-2: fluctuating inference workloads

**Figure 13.** Kernel issuing traces analysis. The normalized kernel ratio denotes inference kernel counts divided by the total kernel counts of all collocated tasks.

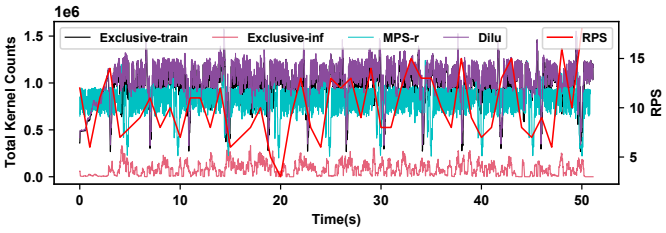



**Figure 14.** Total kernel counts comparison.

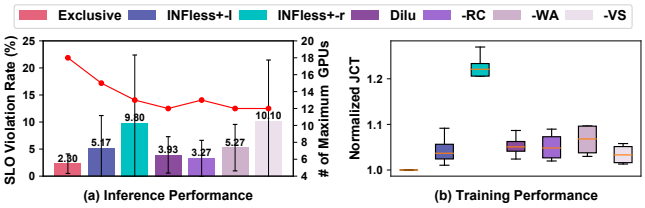



**Figure 15.** End-to-end performance comparison and component analysis of Dilu in local cluster. The error bar in (a) represents the standard deviation of all inference function's SVRs and each boxplot of (b) is constructed based on all normalized training JCT relative to the Exclusive. -RC: without resource complementarity and multi-GPU LLM deployment; -WA: without workload affinity; -VS: without vertical scaling.

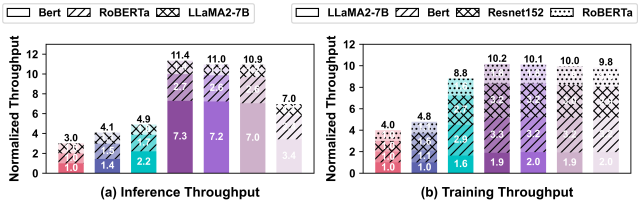



**Figure 16.** Aggregate throughput performance comparison.

workloads, the scaling overhead of each instance is less than 1 ms.

### 5.4 End-to-End Scheduling Performance

**Few resource fragments and high throughput**. To demonstrate the effectiveness of our scheduling mechanism, we submit 4 training functions at different times, including 2 with 2-workers and 2 with 4-workers, along with three inference functions with varying workloads (specifically, bursty, periodic, and Poisson distributions). Figure 15 shows the end-to-end results. Although Exclusive achieves the best DL performance, it requires 1.5× GPUs compared to Dilu. INFless+-r performs poorly due to its lower-bound resource allocation. INFless+-l achieves comparable training performance to Dilu but occupies three more GPUs. Figure 16 reveals that Dilu obtains the highest aggregate throughput (i.e., the inference RPS or training throughput divided by the resources they occupy, following a similar definition as INFless [51]). Specifically, it achieves 3.8×, 2.8×, and 2.3× the performance of Exclusive, INFless+-l, and INFless+-r in inference, and 2.5×, 2.1×, and 1.2× in training.

**Ablation study.** Without *RC*, though there is a slight gain in SVR, it requires one additional GPU due to the lack of a distributed deployment strategy for LLM instances. Without *WA*, a slight decline in both training and inference performance occurs due to the barrel effect. Without *VS*, as training infringes on more compute resources, a slight reduction in overall training Job Completion Time (JCT) is observed. However, the average and maximum inference SVR increase by 158% and 203%, respectively, compared to Dilu.

### 5.5 Large Scale Cluster Simulation

**Minimal GPU occupancy and least resource fragmentation**. The results in Figure 17 indicate that Dilu maintains the lowest memory and SM fragmentation, thereby minimizing GPU occupancy. Compared to Exclusive and INFless+-l, Dilu reduces costs by 30% and 23% respectively, at a scale of 3,200 instances. The bottom of Figure 17 presents the variation in GPU count over time, corresponding to the dynamic launching and termination of instances. The consistently lower purple line indicates that Dilu keeps the most efficient GPU occupying. Meanwhile, Exclusive and INFless+-l exhibit an increasing gap in GPU usage compared to Dilu as the number of instances grows.

**Sensitivity analysis.** Figure 18(a) is based on the 3,200 instances. As the oversubscription coefficient increases, resource fragments and GPU occupancy gradually decrease,

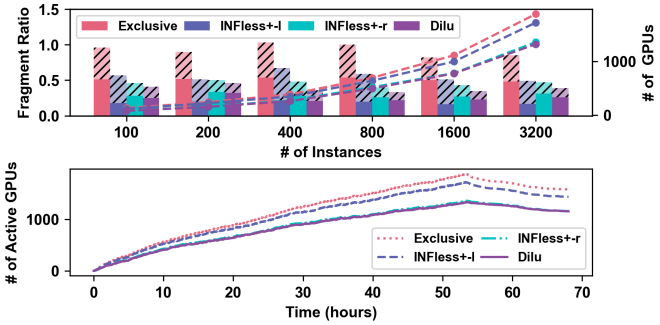
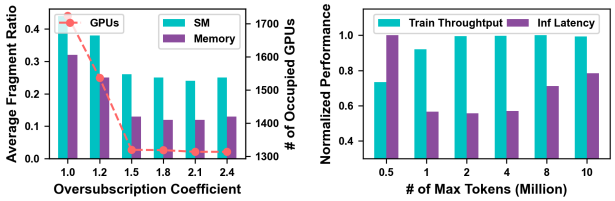



**Figure 17.** GPU provisioning efficiency in large-scale simulations. The dark bottom and light-striped bars represent SM and memory fragments respectively.




(a) Oversubscription coefficient (b) Max tokens

**Figure 18.** Sensitivity analysis on oversubscription coefficient and max tokens. The oversubscription coefficient denotes the sum of *limit* for collocated instances.

with diminishing returns beyond 1.5. Since excessive oversubscription can degrade QoS, we set this parameter to 1.5 in our experiments.

## 6 Related Work

**Serverless DL Systems.** In general, studies aim to simplify deployment and reduce costs for both DL training and inference tasks through the classic horizontal elasticity of serverless computing. LambdaML [26], Siren [46] and Cirrus [6] build data-parallel workers with serverless functions, while FuncPipe [29] and Hydrozoa [22] extend it to the hybrid-parallelism pattern. Works like $\lambda$DNN [50] and ElasticFlow [18] explore to leverage serverless elasticity to accelerate training. For inference, studies such as INFless [19, 51] focus on improving throughput. Amps [25] and Gillis [53] enable distributed inference in serverless context. MArk [55] and Tetris [27] enhance CPU and host memory utilization separately to optimize resource usage. ServerlessLLM [15] leverages memory locality to decrease the launching time of LLM-inference functions. Considering the granularity of serverless elasticity, these studies inevitably produce GPU resource fragments due to the adoption of static GPU allocation methods and classic horizontal scaling patterns. In contrast, Dilu enables GPU resourcing-on-demand for serverless DL serving, leveraging fine-grained and introspective elasticity.

**GPU Sharing**. Previous works attempt to enable GPU sharing in either spatial or temporal dimensions. NVIDIA MIG [37] supports physical partitioning of GPU compute resources, and MPS [38] offers logical partitioning by limiting the active CUDA threads percentage per process, as adopted by [8, 10, 19, 51]. Orion [44] designs a kernel-scheduling mechanism to share GPU with local threads, but not suitable in the cloud. TGS [47] and Antman [49] prioritize productive jobs to occupy GPUs in timeshare. GPUlet [8] and FaST-GS [19] enable spatio-temporal sharing to improve inference throughput, while depending on static MPS and are hard to adjust quickly in serverless context. Dilu introduces dynamic GPU provisioning through *<request, limit>* quota control and high-level scheduling to support spatio-temporal sharing.

**DL Scheduling**. With the rise of deep learning, many studies focus on scheduling DL tasks to improve resource utilization. Antman [49], Tiresias [21] and Pollux [41] concentrate on reducing the average JCT for training tasks while ElasticFlow [18] and Chronus [16] are deadline-aware. For inference workloads, although INFless [51] and FaST-GS [19] optimize inference throughput and guarantee QoS through existing horizontal-scaling mechanisms, they fail to fully utilize temporal resource fragmentation. Moreover, these works cannot adjust GPU provisioning at the 5ms granularity, whereas Dilu can.

## 7 Conclusion and Discussion

Given the rising trends in serverless DL serving and apparent inefficiencies in current GPU provisioning methods, we present Dilu, a cross-layer and introspective design to improve GPU utilization and extend the elastic scaling dimensions of serverless DL systems. The adaptive 2D co-scaling mechanism not only dynamically adjusts GPU provisioning at the intra-instance level to minimize fragmentation, but also enhances the capability to handle sudden workloads, and reduces cold starts at the inter-instance level while guaranteeing QoS. To our knowledge, Dilu is the first serverless DL system with dynamic and omultidimensional elasticity for heterogeneous DL functions.

In the future, we plan to further extend Dilu to explore more elastic serverless training and LLM serving. Additionally, GPU-sharing protection will be aligned with low-level security mechanisms.

## Acknowledgement

We thank all anonymous reviewers for their valuable feedback and comments. We also thank our colleague Xiaohong Wang for her kind support in this study. This work is supported by the Innovation Funding of ICT, CAS under Grant No. E361060, No. E461040, and the Pilot for Major Scientific Research Facility of Jiangsu Province of China under Grant No. BM2021800.

## References


[1] Kubernetes resource quotas. https://kubernetes.io/docs/concepts/policy/resource-quotas/, 2024.

[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment*, 15(10), 2022.

[4] Alibaba. Alibaba pai. https://www.aliyun.com/product/bigdata/learn, 2024.

[5] Amazon. Amazon sagemaker. https://aws.amazon.com/pm/sagemaker/, 2024.

[6] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[7] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. Sla-driven ml inference framework for clouds with heterogeneous accelerators. *Proceedings of Machine Learning and Systems*, 4:20–32, 2022.

[8] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-gpu servers with spatio-temporal sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC'22)*, pages 199–216, 2022.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[10] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 492–506, 2020.

[11] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM international conference on architectural support for programming languages and operating systems*, pages 797–813, 2022.

[12] Tsinghua University Knowledge Engineering and Data Mining Group. Thudm chatglm3. https://github.com/THUDM/ChatGLM3, 2024.

[13] Hugging Face. Huggingface accelerate. https://pypi.org/project/accelerate/, 2024.

[14] Apache Software Foundation. Apache mxnet. https://github.com/apache/mxnet, 2024.

[15] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, 2024.

[16] Wei Gao, Zhisheng Ye, Peng Sun, Yonggang Wen, and Tianwei Zhang. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 609–623, 2021.

[17] Google. Tensorflow. https://www.tensorflow.org/, 2024.

[18] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 266–280, 2023.

[19] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 635–644, 2023.

[20] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. Gaiagpu: Sharing gpus in container clouds. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 469–476. IEEE, 2018.

[21] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A gpu cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 485–500, 2019.

[22] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. *Proceedings of Machine Learning and Systems*, 4:779–794, 2022.

[23] Ziyi Han, Ruiting Zhou, Chengzhong Xu, Yifan Zeng, and Renli Zhang. Inss: An intelligent scheduling orchestrator for multi-gpu inference with spatio-temporal sharing. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[25] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. Amps-inf: Automatic model partitioning for serverless inference with cost efficiency. In *Proceedings of the 50th International Conference on Parallel Processing*, pages 1–12, 2021.

[26] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, pages 857–871, 2021.

[27] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC'22)*, 2022.

[28] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[29] Yunzhuo Liu, Bo Jiang, Tian Guo, Zimeng Huang, Wenhao Ma, Xinbing Wang, and Chenghu Zhou. Funcpipe: A pipelined serverless framework for fast and cost-efficient training of deep learning models. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(3):1–30, 2022.

[30] Meta. Meta llama2. https://llama.meta.com/llama2/, 2024.

[31] Meta. Pytorch. https://pytorch.org/, 2024.

[32] Meta. Pytorch ddp. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html, 2024.

[33] Microsoft. Microsoft aci. https://azure.microsoft.com/zh-tw/products/container-instances, 2024.

[34] Microsoft. Microsoft deepspeed. https://github.com/microsoft/DeepSpeed, 2024.

[35] Diana M Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. Accelerated serverless computing based on gpu virtualization. *Journal of Parallel and Distributed Computing*, 139:32–42, 2020.

[36] NVIDIA. Nvidia collective communications library. https://developer.nvidia.com/nccl, 2024.

[37] NVIDIA. Nvidia mig. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2024.

[38] NVIDIA. Nvidia mps. https://docs.nvidia.com/deploy/mps/, 2024.

[39] NVIDIA. Nvidia nsight systems. https://developer.nvidia.com/nsight-systems, 2024.

[40] OpenAI. Openai gpt2-large. https://huggingface.co/openai-community/gpt2-large, 2024.

[41] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation OSDI'21)*, 2021.

[42] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC'20)*, pages 205–218, 2020.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[44] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1075–1092, New York, NY, USA, 2024. Association for Computing Machinery.

[45] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: Affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, pages 495–514, 2021.

[46] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1288–1296. IEEE, 2019.

[47] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent gpu sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, pages 69–85, 2023.

[48] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[49] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on gpu clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 533–548, 2020.

[50] Fei Xu, Yiling Qin, Li Chen, Zhi Zhou, and Fangming Liu. $\lambda$dnn: Achieving predictable distributed dnn training with serverless architectures. *IEEE Transactions on Computers*, 71(2):450–463, 2021.

[51] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022.

[52] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud. In *Proceedings of the 29th international symposium on high-performance parallel and distributed computing*, pages 173–184, 2020.

[53] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS'21)*, pages 138–148. IEEE, 2021.

[54] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping. *arXiv preprint arXiv:2306.03622*, 2023.

[55] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 1049–1062, 2019.

[56] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.