Efficient Function-as-a-Service for Large Language Models with TIDAL

Weihao Cui^{1,2}^{*}, Ziyi Xu¹^{*}, Han Zhao¹, Quan Chen¹[†], Zijun Li¹, Bingsheng He², Minyi Guo¹ ¹Shanghai Jiao Tong University,²National University of Singapore

Abstract

Large Language Model (LLM) applications have emerged as a prominent use case for Function-as-a-Service (FaaS) due to their high computational demands and sporadic invocation patterns. However, serving LLM functions within FaaS frameworks faces significant GPU-side cold start. A fundamental approach involves leveraging a template with function state saved on GPUs to bypass the cold start for new invocations. Yet, this approach struggles with the high GPU footprint, dynamic initialization behaviors, and lazy GPU kernel loading inherent in LLM functions, primarily due to a lack of insight into the underlying execution details. In this paper, we introduce TIDAL, an optimized FaaS framework for LLM applications that achieves fast startups by tracing fine-grained execution paths. By utilizing the traced execution details, TIDAL generates adaptive function templates, effectively breaking startup barriers for LLM functions. Extensive evaluations demonstrate that TIDAL reduces cold start latency by $1.79 \times 2.11 \times$ and improves the 95%-ile time-tofirst-token by 76.0%, surpassing state-of-the-art methods.

1 Introduction

Function-as-a-Service (FaaS) [1] has emerged as the leading serverless paradigm in cloud platforms [2, 3]. By adopting functions as the basic unit of scheduling, FaaS provides distinct benefits for both application developers and cloud providers. FaaS allows developers to focus on core function logic without caring infrastructure management. Additionally, its pay-as-you-go model lowers costs, particularly for functions with low invocation frequency [4]. For cloud providers, FaaS enhances resource utilization by enabling more effective resource management, ensuring efficiency and scalability.

Large language model (LLM) applications are emerging as a prominent use case for FaaS. These applications require both



Figure 1: Cold-start invocation using Llama2-13B [11] on an Nvidia RTX A6000. The input length is 2k.

rapid innovation and significant GPU resources but may encounter low invocation frequencies, particularly during early deployment phases. The high computational demands and unpredictable workloads make FaaS an ideal solution for hosting LLM functions. Recognizing these benefits, many companies, such as Hugging Face and RunPod [5–7], now offer FaaS services designed for LLM applications.

We observe that the inherent long cold startup problem in FaaS becomes much worse, exacerbated by the complexity of LLM execution environments. From memory stack aspect, executing LLMs requires managing a large footprint across a three-tier memory hierarchy: storage, host memory, and GPU memory. From application aspect, different from traditional FaaS applications, costly dynamic model initialization is often required. For instance, multilingual functions [8] adapt to individual requests by dynamically attaching language-specific LoRA adapters [9] to a shared base LLM [10–12]. The dynamics makes many cache-based optimizations inefficient. From context preparing aspect, inference on GPUs necessitates preparing the execution environment, including creating CUDA contexts [13] and other setup operations.

Figure 1 illustrates the cold start process of an LLM function: initializing the model on the host (loading the checkpoint into host memory), transferring the model from the host to the GPU, and performing the inference on GPU. The first two steps are the primary contributors to cold start. Recent studies [14, 15] have proposed techniques to mitigate the startup latency associated with host-side initialization. In the upper half of Figure 1, state-of-the-art frameworks [14] overlap storage-to-host model loading with host-to-GPU data transfer and cache the models in a host-side memory pool. However, even if the storage-to-host transfer (the first step) is optimized

^{*}Weihao Cui and Ziyi Xu contributed equally to this work.

[†]Quan Chen is the corresponding author

to negligible level [14], the time-to-first-token (TTFT) remains $2.74 \times$ greater than inference latency, primarily due to host-to-GPU data transfer (the second step).

To entirely eliminate cold start, a fundamental solution is template-start [16, 17]. A template consists of a pre-initialized function state cached in memory. Template-start allows new invocations to launch directly from the template, bypassing all initialization steps. To prevent data conflicts, the template must be request-agnostic, and a copy-on-write mechanism is critical for safe template sharing. As shown in the bottom right of Figure 1, by saving the initialized LLM in GPU memory as a template, we could directly launch a new invocation, entirely excluding the cold-start latency. Unfortunately, we find that template-start, which blindly reuses an existing function state without understanding the execution details across function invocation, is generally ineffective for LLM functions.

Firstly, template-start struggles to reconcile the need for fast startup with high template density, due to the substantial footprint of LLMs. E.g., an Nvidia RTX A6000 GPU (48 GB) can accommodate only one template for the function in Figure 1. With such a low template density, template-start is impractical within a FaaS framework, where multi-tenancy is essential. Based on the fact that model weights in the template are accessed sequentially (kernel by kernel) during inference [18], we identify the opportunity to overlap host-to-GPU data transfers with inference. Such overlap potentially reduces the template footprint and increases template density without impacting latency. However, such overlapping cannot be achieved without knowing the order of weights are accessed in a model during inference.

Secondly, template-start fails to reduce cold start latency for functions with dynamic initialization. As previously discussed, LoRA adapters, which are dynamically attached to the base model, are request-specific [19–21]. Although these adapters account for less than 1% of the base model, the initialization of such LLM functions does not meet the criteria for being saved in the template. Ideally, it is possible to mitigate cold start for dynamic LLM functions, by creating a template with the most reusable initialization. Such design currently cannot be achieved, lacking a mechanism that identifies and excludes dynamically initialized components.

Thirdly, starting new invocations from a template with model initialized still suffers from cold start overhead. Our study in §2.2 also reveals that the inference time during a cold start is still higher than that within a fully warm function state, where the model has been loaded and executed once. The key factor is that the code segments for kernel execution are lazily loaded onto GPU during the first inference [22]. Such overhead can be eliminated if these code segments could be proactively loaded when pre-warming processes for new invocations. However, without prior knowledge of the specific GPU kernels launched during inference, such proactive code loading cannot be achieved.

To this end, we introduce TIDAL, an efficient FaaS frame-

work tailored for LLM applications with fast cold start. The key insight of TIDAL is that a detailed understanding of the fine-grained execution paths of an LLM function-spanning initialization and inference-unlocks new opportunities to optimize cold start. Unfortunately, the fine-grained execution paths required for startup optimizations are implicit within each invocation and cannot be manually exposed by function developers. TIDAL integrates a lightweight tracing mechanism to automatically extract these fine-grained execution paths at runtime. Using adaptive function templates generated from traced execution paths, TIDAL tackles the cold-start problem for LLM functions by two optimizations: proactive code segment loading and adaptive state forking. While proactive code segment loading ensures a fully pre-warmed GPU context, adaptive state forking efficiently reuses static components and overlaps model loading with inference.

We implement TIDAL by extending PyTorch to serve as the runtime for LLM functions. TIDAL transparently supports LLM functions wrapped with a wide range of LLM models. We conducted extensive evaluations of TIDAL using representative LLMs [10–12, 23–25] of various sizes, both with and without LoRA enabled. Experimental results show that TIDAL achieves $1.79 \times \sim 2.11 \times$ speedup in cold start latency compared to state-of-the-art solutions. Furthermore, under real-world workloads, TIDAL reduces the 95%-ile of TTFT by 76.0%. The key contributions are as follows:

- We present a detailed analysis of the cold-start overhead of LLM functions on GPUs and highlight the obstacles of directly applying template-start for optimization.
- We identify that the primary challenge in optimizing the cold start of LLM functions lies in the unawareness of the fine-grained execution paths underlying invocations.
- We build up a lightweight, weight-centric tracing mechanism to automatically expose the fine-grained execution paths without manual effort.
- We consolidate optimizations based on the traced finegrained execution paths to minimize the startup latency of cold LLM function invocations.

2 Background and Motivation

2.1 LLMs and Function-as-a-Service

Function-as-a-Service (FaaS) products [5–7] are popular among cloud vendors to support large language model (LLM) applications. These frameworks provide scalable and costefficient solutions, allowing developers to invoke custom LLM functions tailored to specific use cases.

Under this context, application developers encapsulate the LLM inference logic within function code. Figure 2 illustrates such an example written in Python, adhering to the coding logic employed in several state-of-the-art research works and industry products [5–7, 14, 26]. The code is uploaded

to the framework, which automatically invokes the function instance in response to triggered events (e.g., a RESTful http request). Once the function is triggered, the framework loads the function code, initializes the corresponding LLM models on GPUs, and executes the function handler for inference. For high request rates, the initialized model is reused to efficiently process subsequent requests . However, at low request rates, function is started from scratch, requiring time-consuming model initialization for each request This results in the common issue known as cold startup in serverless computing.

2.2 Cold Start of LLM Functions

Researchers have recently focused on alleviating the cold start issues of functions on GPU [14, 26–28]. Figure 3 presents the lifecycle of a cold-start LLM invocation on GPU, utilized to better demonstrate these works and define the target scope of TIDAL. In the figure, we omit the CPU-side initialization processes, such as container creation, as these have been extensively explored in CPU-only studies [17, 29–31].

Warming of model weight on host. Stage-1 in Figure 3 primarily involves loading model checkpoints into host memory. State-of-the-art solutions [14] tailored for LLMs utilize a pinned host memory pool to efficiently load and cache model weights from local storage. With host memory offering significantly greater capacity than GPU memory ($10.7 \times$ in our testbed), advanced pre-loading policies [26,27,32] can further reduce the impact of Stage-1 to a negligible level. Throughout this paper, unless otherwise indicated, we assume that model weights are pre-cached in host memory by existing solutions.

Warming of CUDA context on GPU. Stage-2 in Figure 3 involves the creation of the CUDA context, which initializes essential data structures for GPU-enabled processes. While this stage is notoriously time-consuming, it is functionagnostic, meaning it can be pre-warmed independently of the function invocations. As a result, pre-warming the CUDA context with a process pool allows it to be excluded entirely from the cold start of LLM function invocations.

```
---initialization start---
   import torch
     load checkpoints to host memory
    #
3
   llama_weight= torch.load("llama2-13b")
Δ
     creat CUDA context
5
   torch.cuda.set_device("cuda:0")
6
      init model
                  and load it to cuda
   llama
         = Llama()
    llama.load_state_dict(llama_weight)
10
   llama = llama.cuda()
         -initialization end---
11
12
   def handler(event, context):
13
        # model inference on GPU
14
       output = llama(event["input"])
15
        return {"output": output}
16
```

Figure 2: An example of an LLM function encapsulating LLaMA 2-13B with two parts: initialization and handler.



Figure 3: Lifecycle of a cold-start invocation using Llama2-13B and highlighting the optimizing targets of TIDAL.



Figure 4: Breakdown of GPU cold start and fully-warmed invocation latencies for 2 Llama-family models [10, 11] with varied inputs. For instance, "13B-512" denotes a Llama with 13 billions parameters evaluated using an input length of 512.

Warming of function state on GPU. We collectively term Stages-3&4 in Figure 3 as the "GPU cold start", that remains largely unexplored. It involves loading the initialized model into GPU memory and launching GPU kernels for the firsttime execution. To dissect the GPU cold start, we conduct experiments with representative LLMs [10, 11] on an Nvidia RTX A6000 GPU. In the experiment, we change the length of input sequences from 512 to 4k for LLMs for simulating various tasks [33, 34]. Figure 4 illustrates the latency of a cold-start invocation compared to that of an invocation within a fully warmed function state.

Observed from Figure 4, Stage-3 requires $2.11 \times$ more time than Stage-4 on average. The long time of Stage-3 is dominated by GPU-side modeling loading through PCIe. The issue worsens with shorter input sequences and larger model sizes. Moreover, the latency of Stage-4 exceeds that of a fully warmed invocation by an average of 76.1%, equivalent to 179 ms in absolute terms, primarily due to the loading of kernel-related code segments into the GPU during the *firsttime* kernel execution. The long GPU cold start emphasizes the critical need for optimization.

Existing works [26] have tried to optimize Stage-3 by employing pre-warming policy. The pre-warming method is inherently load-dependent, failing to fundamentally address the cold start caused by model loading over PCIe. Moreover, to the best of our knowledge, we find no work on eliminating the overhead in Stage-4. *Consequently, this paper focuses on systematically optimizing the warming of GPU function state to mitigate the cold start overhead in LLM function invocation.*

2.3 Limitations of a Strawman Solution

A strawman solution of achieving warm GPU function states for cold invocations is adopting template-start [16, 17], a



Figure 5: Strawman solution based on CPU-only templatestart, implemented via CUDA IPC.

Table 1: Memory footprint of weights with varied model.

Model	ResNet-101 [36]	Llama Family-8B/13B/34B [10–12	
Size (B)	170M	15.7G/24.3G/60G	

method proven effective in the traditional CPU-only FaaS to resolve cold start. As shown in Figure 5-(a), this method saves only request-agnostic initializations in a template and leverages the system call to launch new invocations from the template within 10ms. With data sharing via CUDA inter process communication (CUDA IPC) [35], a similar template for LLM functions can be prepared by first loading the model into GPU. Figure 5-(b) illustrates this approach and summarizes its limitations, which stem from GPU or LLM-specific factors and are further elaborated below.

Fast startup but low template density. The strawman solution requires warming the entire model. After the warm-up phase, the GPU memory utilized by the template primarily consists of two components: the CUDA context (around 500 MB), and the model weights. Table 1 compares the memory footprint of traditional small models with that of popular LLMs. While the footprint of small models can be even smaller than the requirement for the CUDA context, the situation becomes significantly worse for LLMs due to their substantial memory demands. As mentioned before, an Nvidia RTX A6000 GPU can support only a single template for Llama2-13B. Such low template density for fast startup is impractical for a FaaS framework, which is designed to accommodate numerous tenants' functions.

Vulnerable request-agnostic template. Employing the strawman solution requires that the saved function state within the template is request-agnostic. Failure to do so would result in incorrect outputs and undermine the statelessness principle of FaaS [37]. Nonetheless, the model initialization for LLM functions inherently exhibits request-specific variability. Application developers may utilize parameter-efficient finetuning techniques [38], like LoRA [9], to better meet individual user needs [19–21]. As shown in Figure 6, LoRA [9] allows the base model to remain unchanged across invocations while



Figure 6: Initializing models with different LoRA adapters for different invocations.



Figure 7: Fine-grained execution path of model weight-1 and intrinsic dependency of an LLM function invocation.

loading user-specific LoRA adapters for each request. Although adapters are significantly smaller than the base model (<1%), the initialization of such dynamic LLM functions is not request-agnostic and cannot benefit from template-start. This limitation reduces the general applicability of templatestart for resolve cold start.

Non-shareable and lazy-loading code segments. Not all GPU data are shareable through CUDA IPC, as only memory explicitly allocated by the process is accessible through this mechanism. Code segments used for launching kernels during inference fall into this category. The CUDA runtime implicitly and lazily loads these segments during the first invocation of the related kernels. Thus, the strawman solution still suffers from cold kernel calls (around 180*ms* in Figure 5). A straightforward mitigation is to eagerly load all code segments during pre-warming of CUDA context. Our experiments show that this approach incurs an additional 1.12 GB of GPU memory consumption and increases the pre-warm time for a process from 830 ms to 3050 ms. This significant overhead makes it impractical to maintain a process pool for LLM invocations.

2.4 **Opportunities & Challenges**

Fine-grained execution paths help. Figure 7-(a) depicts a fine-grained execution path for a specific model weight. In a framework like PyTorch [39] for building LLM functions, weights are loaded from host to the GPU as tensors during initialization and subsequently utilized by operators through GPU kernel launches during inference. Moreover, model inference is also commonly represented as a data flow graph consisting of interconnected operators. By aligning the execution paths of all weights with the topological order of operators in inference, we derive the intrinsic dependencies of cold-start LLM invocation, as shown in Figure 7-(b).

Figure 7-(b) reveals three key optimizations to address the limitations. 1) Kernel execution can overlap with weights loading required for subsequent kernels, removing the need to store the entire model in template. Since model loading via PCIe and inference latencies are comparable (hundreds of milliseconds), carefully managed overlapping would avoid extra latency while increasing template density. 2) Dynamic



Figure 8: Design overview of TIDAL and workflow of function invocation in TIDAL.

elements of model initialization, such as LoRA adapters, can be removed by comparing execution paths across multiple invocations. As these elements constitute only a small fraction, and the base model's weights are still saved within the template, dynamic LLM functions would benefit from templatestart. 3) Kernels specific to an LLM function can be profiled during inference and loaded proactively. Since each LLM function typically uses a small subset of kernels from the kernel libraries, these kernels can be pre-warmed with minimal overhead but excluded from cold start.

Challenges. There are several challenges in achieving the above optimizations based on fine-grained execution paths. 1) Fine-grained execution paths are implicit, making them difficult to be exposed manually by developers; 2) LLM functions have per-invocation dynamic behaviors, requiring runtime extraction of execution paths; 3) General support of these optimizations for various customized LLM functions is hard within a FaaS framework.

3 Design Overview

To address these challenges, we introduce TIDAL, an efficient FaaS framework tailored for LLM applications. TIDAL first proposes a lightweight mechanism for transparently tracing fine-grained execution paths. By leveraging these paths, TIDAL resolves cold start issues across diverse LLM functions, achieving high template density, robust templates, and pre-warmed kernel calls.

Figure 8 depicts the design of TIDAL, along with a streamlined function invocation workflow. TIDAL comprises three key modules: a *runtime tracer*, a *template server*, and a *process pool*. Each function invocation runs atop a runtime tracer that traces the fine-grained execution paths with low overhead (§4.1). It closely interacts with the template server to generate the function templates and launch new invocations. The template server stores the function templates and retrieves model weights from a pinned host memory pool for each invocation. The process pool provides pre-warmed processes for new function invocations.

Invocation workflow. For each LLM function, a function template is prepared either offline or online, based on finegrained execution paths. TIDAL can adjust the size of model weights cached on the GPU within the template, while the

import tidal @tidal.init(static=False) def initializer(event, context): llama_weight = tidal.load(event["llama2-13b"]) lora_weight = tidal.load(event["lora"]) # dynamic all_weights = llama_weight + lora_weight llama_lora = LlamaLoRA() llama_lora.load_state_dict(all_weights) 10 llama_lora = llama_lora.cuda() 11 12 return llama_lora 13 **def** handler(event, context): 14 llama_lora = initializer(event, context) 15 output = llama_lora(event["input"]) 16 17 return {"output": output}

Figure 9: Programming interface of TIDAL.

remaining weights are loaded concurrently with inference. The template is updated by excluding dynamic components at runtime (§4.2). Once the template is ready, TIDAL accepts user requests. Using the template, it pre-warms processes by initializing CUDA contexts and loading kernel code segments proactively (§5.1). Upon invocation, TIDAL adaptively forks the saved LLM from the function template as follows: CPU operations for static model architectures are skipped, dynamic components are initialized as required, and weights not cached on the GPU are loaded asynchronously. When overlapping model loading with inference, TIDAL provides strict correctness guarantees. (§5.2).

Programming interface. TIDAL adheres closely to the programming model of modern FaaS frameworks, as shown in Figure 2. To express an LLM function in a traceable manner, TIDAL imposes minimal requirements on application developers. Figure 9 illustrates an example of defining a function using Llama2-13b with LoRA enabled in TIDAL. The most noticeable difference is that model developers need to wrap the function initialization within a method decorated with tidal.init and explicitly invoke it within the function handler, This requirement arises because an LLM model with dynamics must be initialized for each invocation. The tidal.init decorator accepts an optional argument, static, to indicate whether the initialization is static or dynamic. Without annotation, TIDAL defaults to treating it as dynamic.



Figure 10: Weight-centric two-phase tracing for weight-1.

4 Template Based on Fine-grained Tracing

In this section, we first introduce TIDAL's lightweight tracing mechanism, which transparently extracts fine-grained execution paths. Additionally, we show details of generating function templates for startup optimizations.

4.1 Tracing Fine-grained Execution Paths

Traceability of LLM functions. Fundamentally, LLMs are large-scale deep learning models targeting natural language processing tasks. Therefore, LLMs follows the generic deep learning paradigms, where model inference is abstracted as a data flow graph composed of tensors and operators [40]. While prior works [41] have leveraged this property to trace inference, TIDAL further extends it to model initialization. This extension is driven by the observation that, once loaded into host memory, a weight is represented as a tensor, and subsequent operations performed on it–such as transferring weights from CPU to GPU–are all implemented as deep learning operators. Fortunately, these traceable operations correspond to the fine-grained execution paths that TIDAL needs to optimize the startup process. TIDAL leaves other non-traceable CPU operations during initialization to be executed as normal.

Weight-centric two-phase tracing. Blindly tracing function invocation involves going through all tensors and operators, even those used for temporary storage or reshaping during inference. This leads to considerable runtime overhead. Our preliminary results show that the overhead can be several times greater than that of a standard cold-start invocation. Since TIDAL focuses on optimizing cold starts in LLM function invocations, and model weight loading dominates the initialization, there is no need to treat initialization and inference with the same tracing mechanism.

Therefore, TIDAL adopts a weight-centric two-phase tracing approach, as illustrated in Figure 10. This method focuses on execution paths related to weights. During model initialization, TIDAL employs strict tracing to construct data flow graphs (DFGs) for generating weight tensors. Each weight tensor's DFG specifies the shape used for initialization, the checkpoint for loading weight data, and other relevant details. Using the DFG, TIDAL identifies weights that are dynamically initialized. For example, in an LLM function with LoRA enabled, the adapters are flagged as dynamic because they are sourced from different checkpoints, although their shape



Figure 11: Adaptive function template composed of three key components. The numerical values indicates the sequence in which the weights are initialized.

remains the same. During inference, TIDAL employs lax tracing, capturing only the access patterns of weights, including their order and associated GPU kernels. The tidal.init decorator in Figure 9 enables TIDAL to differentiate between the two phases, activating the appropriate tracing mechanism.

4.2 Generating Adaptive Function Template

With the traced fine-grained execution paths, TIDAL generates the function templates for LLM functions managed by the template server. The function template generated is adaptive in several key aspects: first, it perceives the necessary kernels for proactive code segment loading with minimal overhead; second, it adapts its saved state to accommodate dynamic LLM initialization; third, its GPU memory consumption could be adjusted with loading efficiency guarantee. Such adaptiveness enables TIDAL to address cold starts for various LLM functions, regardless of whether models are dynamically initialized, or what workloads are processed (e.g., model sizes, request rates, input lengths, etc.). As shown in Figure 11, the template for a specific LLM function is generated as follows.

Firstly, information about GPU kernels required for proactive loading of code segments is stored in the template. Since LLM models often consist of multiple identical transformer blocks, TIDAL scans all traced operators and filters them by removing duplicates. GPU kernels associated with these operators are then identified as candidates for proactive loading.

Secondly, model weights are stored in the template with a re-ordered memory layout. TIDAL reorganizes the weights based on the traced access order. Without this reorganization, our evaluation in §7.4 reveals that overlapping efficiency can be easily compromised by misordered weight initialization, highlighting the necessity of TIDAL's tracing mechanism. By leveraging the traced access order, TIDAL also retains a subset of model weights on the GPU while preserving only the memory layouts of others, thereby optimizing the template size. Weights stored as memory layouts in the template are efficiently loaded into the GPU during inference.

Thirdly, data flow graphs for generating each model's weights are stored in the template to facilitate the exclusion of dynamic components at runtime. Since TIDAL cannot identify all dynamic components of an LLM model in a single tracing pass, its low-overhead tracing mechanism enables the incremental exclusion of these components during runtime.

5 Optimizations of Function Startup

In this section, we introduce the key startup optimizations in TIDAL based on insights from fine-grained tracing: proactive code segment loading, and adaptive state forking.

5.1 Proactive Code Segment Loading

Avoiding cold kernel call startup requires knowledge of the kernels to be launched during inference. As discussed in §2.3, template-start incurs a 180-millisecond cold start due to the absence of this information. TIDAL addresses it by leveraging traced kernels stored in its template to proactively load the relevant code segments during process pre-warming. As a result, new invocations do not suffer cold kernel calls.

However, this process requires launching the corresponding kernels on the GPU to trigger proactive loading, which also occupies computational resources. To minimize interference with other ongoing invocations, TIDAL reduces the input dimensions of the triggering kernels. Additionally, as GPU kernels are already deduplicated during template generation, the kernel code segments are efficiently loaded during prewarming with minimal computational and memory overhead, as evaluated in §7.4.

Loading policy. Another issue arises from the multitenancy of FaaS, where a single GPU instance serves multiple LLM functions. Proactively loading code segments for all deployed functions within a single process may lead to excessive GPU memory usage and increased triggering latency. To address this, TIDAL employs a proactive loading policy that aligns with the set of LLM functions currently cached in the host memory of the GPU instance. During the process pool's pre-warming phase, TIDAL triggers the loading of all the deduplicated kernels corresponding to the templates for these cached functions. This policy ensures that the code segments of frequently accessed kernels are readily available.

5.2 Adaptive State Forking

With the process pre-warmed with loaded code segment, TIDAL employs adaptive state forking to efficiently initialize function states from the template for new invocations.

Adaptive template state reusing. When an LLM is dynamically initialized based on user requests, the function initialization cannot be saved into the template for fast startup in template-start. As shown in Figure 7, TIDAL maximizes reuse of static elements from the existing state within the template by leveraging its traced fine-grained execution paths. Since dynamic elements typically account for only a small portion of the entire model (barely 1%), TIDAL achieves fast startup by reusing the majority of the initialized LLM.

Specifically, TIDAL supports both dynamic initialization and template-based startup through its runtime tracing. In-



Figure 12: Adaptively forking an LLM function invocation and performing inference with overlapped data loading. The function is dynamically initialized with LoRA enabled.

stead of directly reusing function state from template, TIDAL executes function initialization atop of runtime tracer. For model weight identified as static initialization, TIDAL reuses it from the template server. For model weight identified as dynamic initialization, TIDAL initializes it within user code.

The left part of Figure 12 presents an example of how TIDAL forks a new invocation from its function template. During initialization, TIDAL traces the operators involved in GPU weight initialization, generating a data flow graph for comparison with the pre-saved graphs in the function template. For weights (e.g., weight-1, weight-5, weight-4, and weight-3) whose data flow graphs match the template, TIDAL skips operator execution and directly initializes the tensors by forking GPU memory pointers from the template. In contrast, weight-2, loaded from a different adapter, has a data flow graph inconsistent with the template. As a result, TIDAL replays the operators for weight-2 to dynamically initialize it.

Dynamic elements, such as LoRA adapters, are handled within user code and can be loaded either from host memory or storage. A function with LoRA enabled may be equipped with thousands of request-specific adapters [19–21]. Due to this specificity, the caching policies of FaaS frameworks often fail to effectively cache checkpoints for these adapters. To ensure a fair comparison in §7, TIDAL currently loads these dynamically initialized adapters directly from storage.

Efficient overlapping with correctness ensuring. On the left side of Figure 12, when weights such as weight-3 and weight-4 are forked from the template, only their GPU memory addresses are allocated initially. At this point, their actual data temporarily resides in the host memory pool and TIDAL's template server is loading these weights into the GPU asynchronously in the background. In this case, TIDAL is capable of overlapping model loading with inference, thus reducing TTFT of an cold-start LLM invocation to the latency of either loading or inference, whichever is longer.

While template server enables overlapping, two key issues remain: efficiency and correctness. The misalignment between the order of weight initialization and their access order reduces overlapping efficiency when weights are loaded arbitrarily. TIDAL leverages the traced access order to ensure weights are loaded in the correct sequence. As shown in the right of Figure 12, although weight-3 is initialized before weight-4, the template server loads weight-4 ahead of weight-3 because weight-4 is consumed by a kernel first during inference, according to the traced access pattern.

For correctness, TIDAL addresses two key aspects. Firstly, dependencies between weights and kernels must be preserved, as data is transferred to the GPU using asynchronous copy operations. To ensure the required data is available on the GPU before kernel execution, TIDAL injects synchronization events based on the traced execution paths. Secondly, TIDAL employs a copy-on-write mechanism to prevent modifications to weights forked for a new invocation. The runtime tracing in TIDAL inspects the read-write properties of operators before launching the underlying kernels. If an operator attempts to write to a forked weight, TIDAL copies the weight into a new tensor to maintain its read-only property during inference.

Adapting template size for less cold start. As mentioned earlier, TIDAL reduces the TTFT of a cold-start LLM invocation to the greater of the loading or inference latency. Ideally, this latency could be reduced to match inference, as inference is the only step that cannot be performed in advance. TIDAL could adapt the template size on GPU-the model weights prefetched on the GPU within the template-based on traced model access order to achieve this. As highlighted in prior works [18], this is fundamentally a trade-off between computation and data transfer. To improve overlapping, TIDAL dynamically adjusts the template size based on the following principles. First, TIDAL need to analyze the function workloads to determine the average input length and batch size of LLM function requests. Using the analyzed data, it profiles the warm execution to obtain an average TTFT for the current LLM function. TIDAL then adapts the template size according to Equation 1,

$$M_{prefetch} = \max(M_{model} - T_{TTFT} \times B_{PCIe}, 0)$$
(1)

where, $M_{prefetch}$ denotes the template size, M_{model} the entire LLM weight footprint, T_{TTFT} the analyzed average TTFT, and B_{PCIe} the bandwidth of PCIe. Notably, $M_{prefetch}$ represents the maximum required template size for optimal performance. TIDAL dynamically adapts this value using its runtime tracing to balance cold start reduction with high template density.

Keep-alive of dynamic function. FaaS frameworks typically keep a launched function instance alive for a predefined interval to handle subsequent requests and avoid cold starts. However, for dynamic functions that initialize different models per request, the initialized models cannot be reused for subsequent requests. With adaptive fork, TIDAL retains static model weights on the GPU during initialization while only re-initializing dynamic components. This allows dynamic functions in TIDAL to benefit from the performance enhancements of keep-alive. To enable adaptive support for keep-alive across all LLM functions, TIDAL requires application developers to specify whether a function is dynamically or statically

initialized, as mentioned in Figure 9. During the keep-alive interval, this information allows TIDAL to skip initialization for static function invocations and leverage adaptive state forking for dynamic ones. Without this specification, TIDAL assumes all functions are dynamic, leading to inefficient keep-alive management for static functions.

6 Implementation

TIDAL is implemented in Python and C++ with approximately 5,800 lines of code. Its function runtime and programming interface are built by extending PyTorch, utilizing 3,900 lines of C++ and 1,050 lines of Python. Although TIDAL primarily addresses the cold start challenge for individual LLM functions by tracing their fine-grained execution paths, we have also developed a FaaS scheduler prototype to evaluate its performance with real-world workloads. This scheduler, consisting of 840 lines of Python, supports essential features required in a FaaS cluster, such as keep-alive for functions with high invocation rates and early-reject mechanisms for timeout requests, etc.

Dispatch-based runtime tracing. TIDAL's tracing mechanism leverages the dispatch mechanism of PyTorch [42], which enables the injection of tracing code through its Python frontend before the execution of operators. Although the weight-centric two-phase tracing is designed to capture only the fine-grained execution paths needed for startup optimization, a naive implementation in Python end introduces substantial overhead due to frequent context switches between Python and C++. During inference, this overhead can reach as high as 100%. To mitigate this issue, TIDAL registers its runtime tracer as a custom backend in PyTorch, operating entirely within C++ context. As a result, TIDAL eliminates the overhead caused by thousands of context switches, significantly improving the efficiency of tracing and execution.

Tailored memory pool in template server. TIDAL's template server is designed to efficiently transfer model weights from host memory to the GPU. When overlapping host-to-GPU data transfers with kernel execution, a single LLM may require the transfer of numerous weight tensors. Transferring these tensors individually, as stored in the template, could saturate the command queue for memory copy operations [43], leading to significant overhead during function initialization. To address this, TIDAL's template server automatically merges weight tensors into fewer tensors when their total number exceeds a threshold.

7 Evaluation

7.1 Experimental Setup

Testbed. We evaluate TIDAL with two setups. The first testbed has four servers, each equipped with an AMD EPYC



Figure 13: TTFT of LLM functions across different LLMs with input length fixed at 2048 and batch size fixed at 1, including variants with LoRA enabled.

7R32 CPU and two Nvidia RTX A6000 (48GB) GPUs. Each server has 512GB of host memory and communicates with the GPUs via PCIe 4.0, providing a bandwidth of 32GB/s. The second testbed is a server with an Intel Xeon Platinum 83698 CPU with 8 Nvidia Amper 100 (80GB) GPUs. This server has 1TB of host memory, and GPU communication occurs through PCIe 3.0, offering a bandwidth of 16GB/s. Both servers run CUDA-12.5 and PyTorch-2.4. We use the first testbed for most evaluations including startup of single LLM function and application of TIDAL within a FaaS cluster. The second sever is used for evaluating the scalability of TIDAL for distributed LLM functions.

Benchmarks. We evaluate TIDAL using models from four representative LLM families: GPT-2 [25], OPT [23], Gemma [24], and Llama [10–12], covering parameter sizes ranging from 1.5 billions to 70 billions. For single-GPU LLM function evaluations, we include GPT-2-1.5B, OPT6.7B, Gemma-9B, Llama3-8B, and Llama2-13B. For evaluation in a distributed inference environment, we utilize Llama2-13B, Llama2-34B, and Llama3-70B. LoRA adapters are attached to evaluated LLMs for constructing dynamic LLM functions, following the previous works [19, 20]. Across all evaluated LLM functions, we report their time-to-first-token (TTFT), which reflects the startup latency corresponding to Stages 3&4 in Figure 3, the primary optimization target of TIDAL.

7.2 Startup of LLM Functions

7.2.1 TTFT across Various LLM Functions

To the best of our knowledge, TIDAL is the first approach to introduce template-start in GPU-related applications. No prior work has demonstrated the capability to launch an invocation directly from a template residing on a GPU. Consequently, we evaluate TIDAL against the following three baselines for cold-start LLM function invocation. PyTorch-pin assumes that the model has been pre-initialized in host pinned memory, representing the upper bound of latency for a cold-start LLM invocation. ServerlessLLM, state-of-the-art FaaS solution for LLM, caches model weights in a separate host-side pinned memory pool for launching a LLM invocation. Execution assumes the model has already been loaded into GPU memory and executed once, representing the lower bound of latency for a cold-start LLM invocation. In this experiment, TIDAL prefetches none of weights within the template generated for all evaluated LLM functions for fair comparison (Tidal-0G). Each LLM is tested in two versions: the original and a LoRAenabled dynamic variant. The input length is fixed at 2048, and the batch size is set to 1.

Figure 13 shows the TTFT of all evaluated LLM functions. All latencies are normalized to Pytorch-pin. Compared to PyTorch-pin and ServerlessLLM, Tidal-OG achieves $1.96 \times$, $2.00 \times$ speedup of TTFT on average, respectively. While none of the systems occupy GPU memory before function invocation, TIDAL minimizes startup latency by skipping the initialization of static model elements, overlapping model loading and inference, and proactively loading code segments using fine-grained execution paths traced during runtime. In contrast, PyTorch-pin and ServerlessLLM require the model to be fully initialized and loaded into GPU memory before inference can begin. Additionally, their model inference processes are impacted by cold kernel calls, further contributing to latency. When LoRA is disabled, TIDAL delivers an average TTFT speedup of 2.07× compared to ServerlessLLM. With LoRA enabled, the speedup decreases to $2.0 \times$ on average. The performance slowdown is expected, as TIDAL's runtime tracer cannot trace or save the dynamic initialization of LoRA adapters into the function template. The initialization of dynamic elements is fully managed by LLM functions themselves without acceleration. Despite this, by reusing the majority of the initialized model (99%), TIDAL still achieves significant reductions for dynamic functions.

Compared to Execution, Tidal-0G remains 22%~84% slower across all LLM functions. This slowdown can be attributed to two main factors. First, all model weights in Tidal-0G are loaded from host memory. By prefetching a portion of the model weights into TIDAL's template (template size), TIDAL can further approach the lower bound. The impact of template size will be discussed later. Second, TIDAL is unable to trace pure CPU-based operations during initialization, which fall outside its acceleration capabilities. This limitation is noticeable in the case of GPT2-1.5B, whose initialization involves numerous CPU operations.

Notably, ServerlessLLM fails to execute LLM functions wrapped with GPT2-1.5B. This limitation stems from the fact that ServerlessLLM is not a native FaaS framework tailored for LLM functions. ServerlessLLM requires manual adaptation of the LLM model initialization for efficient host-to-GPU data transfer. In contrast, TIDAL leverages its runtime tracer to transparently enable its optimizations without the need for manual intervention.

7.2.2 Ablation Study

To further analyze TIDAL's effectiveness in reducing startup latency for LLM functions, we conduct ablation studies. The



Figure 14: TTFT of Llama-family functions with varied template sizes, including LoRA-enabled variants. Input length and batch size are fixed at 2048 and 1, respectively.



Figure 15: TTFT of Llama-family functions with varied input lengths and template sizes (0G, 4G, and entire model), including LoRA-enabled variants. The batch size is set to 1.

following experiments focus on Llama-family models, as they are among the most widely used pretrained LLMs for customization. Experimental results with other LLMs demonstrate similar trends. We omit the results of PyTorch-pin and ServerlessLLM from most experiments, as they are significantly slower than all configurations of TIDAL.

TTFT with varied template sizes. In this experiment, we evaluate TIDAL's effectiveness by varying the template size on the GPU. Figure 14 illustrates the TTFT of the evaluated LLM functions as the template size varies from 0Gto the entire model size. Specifically, Tidal-Warm refers to the configuration where the template size equals the entire model size. Compared to Tidal-OG, Tidal-Warm achieves a 14%~48% TTFT speedup across all evaluated functions. Generally, TIDAL's performance improves with larger template sizes, up to the point where model loading fully overlaps with model inference. From the figure, we also observe that the template size required for dynamic LLM functions with LoRA enabled to achieve the best TTFT is smaller than that of static functions. This is because the initialization of dynamic functions takes longer than their static counterparts, allowing TIDAL to use this time to overlap more model loading.

TTFT with varied input lengths and batch sizes. In this experiment, we explore the impact of function workloads on TIDAL's performance. To evaluate this, we create different workloads by varying function input lengths and batch sizes. Figure 15 illustrates the TTFT of the evaluated functions with varying input lengths, while Figure 16 presents the TTFT of the evaluated functions with increased batch sizes. Both



Figure 16: TTFT of Llama-family functions with varied batch sizes and template sizes (0G, 4G, and entire model), including LoRA-enabled variants. The input length is set to 2048.



Figure 17: Breakdown of TIDAL under different conditions.

figures include three variants of TIDAL, each configured with a different template size. From these figures, we observe a turning point for Tidal-0G and Tidal-4G, where their TTFT converges with that of Tidal-Warm when the input length or batch size exceeds a certain threshold. This behavior occurs because higher workloads provide TIDAL with greater capacity to overlap model loading and inference.

Improvement breakdown in TIDAL. Figure 17 summarizes TIDAL's improvement breakdown using the example of Llama3-8B with LoRA enabled. The optimizations include three key steps: kernel code segments are proactively loaded before invocation, only the LoRA adapters are initialized during invocation, and model loading is overlapped with both adapter initialization and model inference. Three distinct cases are observed: when the input sequence length is 2k, the TTFT is 632ms, dominated by model loading; increasing the template size to 4GB reduces the TTFT to 571ms, dominated by model inference; and increasing the input sequence length to 4096 raises the TTFT to 927ms, also dominated by model inference. These results highlight how workload (input length) and template size influence performance, with TTFT bottlenecks shifting between model loading and inference.

TTFT with distributed inference. We also evaluate the performance of TIDAL in a distributed inference setting. Functions are defined using Llama-family models with 13 billion, 34 billion, and 70 billion parameters, running on 2, 4, and 8 A100 GPUs, respectively, on our second testbed. Each LLM is parallelized using tensor parallelism [44]. We compare TIDAL against PyTorch-pin and Execution, while ServerlessLLM is excluded as it does not natively support tensor parallelism. Figure 18 shows the TTFT of three distributed LLM functions. Compared to PyTorch-pin, Tidal-0G, Tidal-4G, Tidal-8G, and Tidal-Warm achieve TTFT speedups of $1.76 \times -2.01 \times$, $2.33 \times -2.66 \times$, $3.15 \times -4.24 \times$, and $3.19 \times -5.16 \times$, respectively. This consistent performance demonstrates TIDAL's scalability in a distributed



Figure 18: TTFT of three distributed functions using Llamafamily models, with input length = 4096 and batch size = 1.

Table 2: LLM tasks with their average input length provided, as it significantly impacts the TTFT of each LLM invocation.

Tasks	Mail [45]	Conv [33]	Code [33]	LongBench [34]
Input len. (Avg.)	867	1154	2048	6101

inference environment.

7.3 Evaluation with Real-world Traces

We further evaluate TIDAL using real-world workloads on 4 servers of first testbed. Specifically, we generate 16 LLM function traces by combining real-world serverless workloads [46] with LLM tasks [33, 34, 45]. Out of the 16 function traces, we include four replications each of Llama3-8B, Llama3-8B-LoRA, Llama2-13B, and Llama2-13B-LoRA. Each replication is associated with a specific task listed in Table 2. The function traces represent low, medium, and high invocation rates, covering a range of input sequence lengths. Across all traces, the batch size is fixed at 1. To ensure feasibility, the traces are scaled and accelerated to complete within 6 hours, as the original traces, sourced from a large cluster, span a duration of 7 days. For evaluation purposes, the request timeout during scheduling is set to 60 seconds. As detailed in §2.2, all models are pre-loaded into a host-side pinned-memory pool.

We compare TIDAL with ServerlessLLM [14], the only publicly available end-to-end FaaS solution. We first set the keep-alive interval to the model loading time following ServerlessLLM. Figure 19-(a) presents the results for Serverless-LLM alongside three variants of TIDAL: Tidal sets the template size of all functions to zero. Tidal-DK extends Tidal by enabling keep-alive for dynamic functions. Tidal-DK-6G further enhances Tidal-DK by selecting function 4 traces and increasing their template size to 6GB of memory on 2GPUs. Since 8 GPUs are used for evaluation, 6GB of each selected GPU's memory is allocated for caching the function templates. The adjustments of template size are guided by Equation 1.

Compared to ServerlessLLM, Tidal reduces the 95%-ile latency of TTFT by 76.0%, as TIDAL significantly reduces GPU-side cold start across all LLM functions. As illustrated in the amplified CDF, the variants of TIDAL progressively reduce function latency, with each variant outperforming the previous one. With keep-alive enabled for dynamic LLM functions, Tidal-DK effectively avoids the cold starts of dynamic LLM invocations with high invocation rates. In Tidal-DK-6G, 6GB



Figure 19: (a) CDF of under real-world workloads (b) Average latency of different percentile stages.



Figure 20: (a) TTFT of LLM invocations with different weight loading orders; (b) Decoding latency of LLM invocations with 99 output tokens.

of GPU memory is allocated for storing templates. However, this memory usage does not impact the invocation of other functions, indicating a high template density. Additionally, by increasing the template size for four functions, Tidal-DK-6G further reduces their cold start latency. Overall, Tidal-DK-6G achieves the best performance.

To further evaluate TIDAL's cluster-wide performance, we increased the keep-alive interval to 10 seconds. Figure 19-(b) shows the average latency across various percentile stages. Under different keep-alive configurations, TIDAL consistently outperforms ServerlessLLM at all percentile stages, demonstrating its robustness as a FaaS framework for LLMs.

7.4 **Optimizations**

Loading order of model weights. We evaluate TIDAL under different weight-loading orders, with the results shown in Figure 20-a. Tidal denotes the access order traced by TIDAL. Reverse represents the reverse of Tidal, while Default corresponds to the initialization order of the weights. Compared to Reverse and Default, Tidal achieves performance improvements of $1.55 \times$ and $1.54 \times$, respectively. The similar performance of Default and Reverse attributes to the fact that many LLMs, such as Llama2 [11], share the weights of their embedding layer (the first layer) with the final output layer. Moreover, this weight tensor is initialized and loaded by the last layer but accessed first during inference. In contrast, TIDAL automatically extracts such fine-grained execution paths, thus maximizing overlapping efficiency.

Overhead reduction in TIDAL. The overhead in TIDAL arises from two aspects. First, proactive code segment load-ing increases the memory consumption of each pre-warmed

Table 3: TTFT (ms) of Llama2-70B on 8 A100 GPUs, with and without weight tensor merging.

Input len.	512	1024	2048	4096	8192	16384
No Merge	1366	1370	1371	1541	2203	4084
Merge	1363	1364	1366	1381	1539	3406

process for LLM functions. Experimental results indicate that TIDAL raises memory usage per pre-warmed process from 270MB to 350MB and extends process pre-warming time from 830ms to 1070ms. Such overhead is acceptable relative to the substantial requirements of an LLM function invocation. Second, TIDAL's runtime tracing could reduce inference performance. In Figure 20-(b), we compare the LLM decoding latency of TIDAL with native PyTorch to evaluate this impact. During this phase, the model weights are already loaded into memory, but TIDAL's runtime tracing remains active to ensure correctness guarantees. Compared to native PyTorch, TIDAL incurs an overhead of less than 1.2%.

Memory pool in template server. When there are too many weight tensors, TIDAL's template server merges them into fewer weight tensors to improve overlapping efficiency. For instance, while Llama2-70B initializes 1,200 weight tensors, TIDAL merges them into just 300 tensors. Table 3 compares the TTFT of Llama2-70B with and without weight tensor merging. Without tensor merging, the overhead increases with input length but stabilizes at 600 milliseconds. By leveraging tensor merging, TIDAL achieves strong scalability, even as LLMs scale to extremely large sizes.

7.5 Security Analysis in TIDAL

TIDAL shares static initialization across different invocations, ensuring that this initialization does not include any requestspecific content. Furthermore, TIDAL's runtime tracer guarantees that all weight tensors are shared in a copy-on-write manner during inference. A potential data leak could occur if application developers' customized GPU kernels bypass TIDAL's runtime tracer. To address this, TIDAL requires developers to register their customized GPU kernels in PyTorch for tracing. A static code analyzer could be employed to identify and mitigate such issues.

8 Related Work

Optimizations of Cold Start. Several studies [26–28] have explored optimizing cold starts in FaaS frameworks for deep learning inference. Tetris [27] reduces memory footprints through tensor sharing to warm more functions, while Optimus [28] reuses model structures across functions to optimize function loading. However, thet primarily target deep learning models [36, 47], and focus exclusively on CPU-based inferences. In contrast, TIDAL addresses GPU-side cold starts for

LLM functions, a critical and overlooked issue.

InstaInfer [26] addresses GPU-side challenges by employing preloading techniques but falls short of fully eliminating cold starts due to its load-dependent design. Serverless-LLM [14] optimizes data transfers from storage to host memory and employs a locality-driven scheduler for reducing cold start for LLM functions. However, ServerlessLLM neglects GPU-side cold starts, missing opportunities to overlap hostto-GPU data transfers with model inference, proactively load critical code segments, and tackle the cold starts of dynamic LLM functions.

Template-start in CPU-only FaaS Template-start techniques [16, 17] have been widely adopted in CPU-only FaaS frameworks to eliminate cold starts for new invocations. Among these, Catalyzer [17] was the first to propose launching new invocations from an existing template, effectively bypassing function initialization. Rund [16] further improves template deployment density by introducing lightweight secure container runtime. Inspired by these works, TIDAL goes further by generating adaptive function templates based on traced fine-grained execution paths, rather than simply reusing an existing template.

Serverless Scheduling for Deep Learning. Substantial works [48–52] have explored integrating deep learning applications into serverless environments with advanced scheduling techniques. Some efforts [48, 49, 51, 53] enable dynamic batching for deep learning inferences within serverless frameworks. INFaaS [52] further introduces the concept of modelless architectures for automated management of model variants. These advanced scheduling techniques are complementary to TIDAL and could be integrated to further enhance the performance of LLM functions.

9 Conclusion

In this paper, we presented TIDAL, an optimized Functionas-a-Service (FaaS) framework designed to address the challenges of serving Large Language Model (LLM) applications within FaaS environments. By tracing fine-grained execution paths to generate adaptive function templates, TIDAL effectively overcomes GPU-side cold start issues that hinder existing frameworks. Our extensive evaluations highlight TIDAL's ability to significantly reduce cold start latency by $1.79 \sim 2.11 \times$ and improve the 95%-ile TTFT by 76.0%, compared to state-of-the-art solutions. These results confirm TIDAL's potential as a robust and efficient FaaS framework for the growing demands of LLM workloads.

References

[1] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

- [2] Microsoft Corporation. Azure Functions. https:// functions.azure.com/, 2024. Accessed: 2024-11-14.
- [3] Amazon Web Services, Inc. AWS Lambda. https: //aws.amazon.com/lambda/, 2024. Accessed: 2024-11-14.
- [4] Akash Tayal, Eric Lam, Diganto Choudhury, Meghan Dickerson, Ganesh Moovera, , and Gary Arora. Determining the total cost of ownership of serverless technologies when comparing aws lambda and amazon ec2. https://pages.awscloud.com/rs/112-TZM-766/ images/AWS_MAD_Deloitte_TCO_paper.pdf, 2021. Accessed: 2024-11-30.
- [5] Hugging Face. Create custom inference handler. https://huggingface.co/docs/ inference-endpoints/guides/custom_handler, 2024. Accessed: 2024-11-29.
- [6] RunPod. Build a serverless application on runpod. https://docs. runpod.io/serverless/get-started# build-a-serverless-application-on-runpod, 2024. Accessed: 2024-11-29.
- [7] Beam. Running functions on gpu in beam. https://docs.beam.cloud/v2/environment/ gpu#running-tasks-on-gpu, 2024. Accessed: 2024-11-29.
- [8] Annamalai Chockalingam and Kedar Potdar. Deploy diverse ai apps with multilora support on rtx ai pcs and workstations. https://developer.nvidia.com/blog/ deploy-diverse-ai-apps-with-multi-lora-support/ -on-rtx-ai-pcs-and-workstations/, 2024. Accessed: 2024-11-29.
- [9] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. (arXiv:2106.09685), October 2021.
- [10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, and et al. The llama 3 herd of models. (arXiv:2407.21783), August 2024.

- [11] Hugo Touvron, Louis Martin, and Kevin Stone. Llama2: Open foundation and fine-tuned chat models.
- [12] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. (arXiv:2302.13971), February 2023.
- [13] NVIDIA Corporation. CUDA Driver API: Context Management, 2024. Accessed: 2024-11-30.
- [14] 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.
- [15] Yi Cheng, Cade Daniel, Chen Shen, and Liguang Xie. Loading llama-2 70b 20x faster with anyscale endpoints. https://www.anyscale.com/blog/ loading-llama-2-70b-20x-faster-with-anyscale/ -endpoints, 2023. Accessed: 2024-11-29.
- [16] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. {RunD}: A lightweight secure container runtime for high-density deployment and highconcurrency startup in serverless computing. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 53–68, 2022.
- [17] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 467–481, Lausanne Switzerland, March 2020. ACM.
- [18] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 499–514, 2020.
- [19] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 911–927, 2024.

- [20] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-Tenant LoRA Serving. In *Proceedings of Machine Learning and Systems*, volume 6, pages 1–13, May 2024.
- [21] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph Gonzalez, and Ion Stoica. SLoRA: Scalable Serving of Thousands of LoRA Adapters. *Proceedings of Machine Learning* and Systems, 6:296–311, May 2024.
- [22] NVIDIA Corporation. Cuda module management. https://docs.nvidia.com/cuda/ cuda-driver-api/group_CUDA_MODULE.html, 2024. Accessed: 2024-11-23.
- [23] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. arXiv preprint arXiv:2205.01068, 2022.
- [24] Thomas Mesnard and 106 others. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.
- [25] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
- [26] Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. Pre-warming is not enough: Accelerating serverless inference with opportunistic pre-loading. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 178–195, Redmond WA USA, November 2024. ACM.
- [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] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. Optimus: Warming serverless ML inference via inter-function model transformation. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1039– 1053, Athens Greece, April 2024. ACM.
- [29] Alexander Fuerst and Prateek Sharma. FaasCache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming*

Languages and Operating Systems, pages 386–400, Virtual USA, April 2021. ACM.

- [30] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, pages 335– 350, La Jolla CA USA, April 2024. ACM.
- [31] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 57–70, 2018.
- [32] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), pages 315–328, 2023.
- [33] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), pages 118–132, June 2024.
- [34] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. arXiv preprint arXiv:2308.14508, 2023.
- [35] NVIDIA Corporation. Cuda c++ programming guide - interprocess communication. https://docs.nvidia.com/cuda/ cuda-c-programming-guide/index.html? highlight=ipc#interprocess-communication, 2024. Accessed: 2024-11-23.
- [36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 770–778, June 2016.
- [37] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys*, 54(10s):1–34, January 2022.
- [38] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. (arXiv:2403.14608), April 2024.

- [39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- [40] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 881–897, 2020.
- [41] PyTorch. Dynamo overview. https://pytorch. org/docs/stable/torch.compiler_dynamo_ overview.html#dynamo-overview, 2024. Accessed: 2024-11-30.
- [42] PyTorch. Extending all torch api with modes. https: //pytorch.org/docs/stable/notes/extending. html#extending-all-torch-api-with-modes, 2024. Accessed: 2024-11-30.
- [43] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 539–558, 2022.
- [44] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. (arXiv:1909.08053), March 2020.
- [45] Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, Vancouver, Canada, July 2017. Association for Computational Linguistics.
- [46] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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.
- [47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [48] 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.

- [49] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment*, 15(10), 2022.
- [50] 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, Lausanne Switzerland, February 2022. ACM.
- [51] 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.
- [52] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 397–411, 2021.
- [53] 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, highthroughput inference. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pages 768–781, 2022.