MEDUSA: Accelerating Serverless LLM Inference with Materialization

Shaoxun Zeng
Tsinghua University
Beijing, China
zsx21@mails.tsinghua.edu.cn

Minhui Xie
Tsinghua University
Beijing, China
xmh19@mails.tsinghua.edu.cn

Shiwei Gao
Tsinghua University
Beijing, China
gsw23@mails.tsinghua.edu.cn

Youmin Chen
Tsinghua University
Beijing, China
chenyoumin@tsinghua.edu.cn

Youyou Lu*
Tsinghua University
Beijing, China
luyouyou@tsinghua.edu.cn

Abstract

Serverless is a promising paradigm to provide scalable, cost-efficient, and easy-to-use model inference services. However, the cold start of model inference functions requires loading models to the devices, which incurs high latencies and undermines the benefits of serverless computing. In LLMs, things get even worse since two extra stages are introduced: a *KV cache initialization stage* that profiles and anticipates memory reservation for KV cache, and a *capturing stage* which dynamically constructs CUDA graphs for different batch sizes. Both stages are paramount to the inference performance, but become the main culprit of cold start latency.

This paper proposes MEDUSA to mitigate the long cold start latency through *state materialization*. Instead of dynamic profiling and construction in the runtime, MEDUSA materializes the CUDA graphs as well as the information needed by the KV cache initialization in the offline phase, and restores them efficiently in the online phase. MEDUSA further introduces two novel techniques – *offline-online cooperated parameters restoration* and *triggering-kernels enhanced kernel address restoration* – to tackle non-deterministic issues in CUDA graphs. MEDUSA successfully materializes and restores CUDA graphs across 10 models (with a total of 139364 CUDA graph nodes), and reduces the latency of model loading by 42.5%. Under real-world LLM inference workloads, MEDUSA reduces the tail latency of the time to first token (TTFT) by 53.0%.

CCS Concepts: • **Information systems** → **Information storage systems**.

*Youyou Lu is the corresponding author.

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

ACM ISBN 979-8-4007-0698-1/25/03…$15.00
https://doi.org/10.1145/3669940.3707285

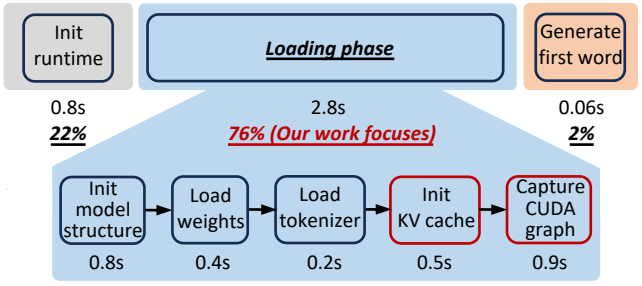

Figure 1. Cold start timeline when serving Qwen1.5 4B.

Keywords: LLM; serverless computing; machine learning system

ACM Reference Format:
Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, Youyou Lu. 2025. MEDUSA: Accelerating Serverless LLM Inference with Materialization. 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, 16 pages. https://doi.org/10.1145/3669940.3707285

1 Introduction

The widespread adoption of large language models (LLMs) has led to a phenomenal increase in demand for serving LLM inference at a large scale [11, 16, 17, 35]. Serverless computing [22, 41] presents a promising paradigm for providing scalable, cost-efficient, and easily accessible LLM inference services. LLM inference requests are highly bursty, potentially fluctuating by 10-20 times within a 30-second window [38]. Accommodating the highest request rates would require massive, expensive GPU resources. Consequently, the pay-as-you-go nature of the serverless paradigm is particularly well-suited for serving LLMs at scale, reducing operational costs for users significantly. As a result, several major cloud providers offer serverless LLM inference services, including Amazon SageMaker [9], Microsoft Azure [26], Alibaba PAI [5], and among others.

Despite the benefits of the serverless paradigm, scaling to accommodate bursty user requests presents a notorious challenge known as the *cold start* problem. This occurs when launching a new serving instance from scratch, which typically takes orders of magnitude [13, 34, 53, 54] longer time than serving a request on an already-running instance. The long cold start latency can significantly impair the *time-to-first-token* (TTFT), a critical metric in LLM inference serving [1, 14]. The cold start latency stems primarily from the *runtime initialization phase* (e.g., user codes and execution environment) and *loading phase* (e.g., loads model-specific data and performs warm-up profilings), as shown in Figure 1. Existing work [2, 15, 25, 27, 34, 40, 42, 54, 57] focuses on eliminating the overhead introduced by runtime initialization. However, we find that in the serverless LLM inference, the loading phase accounts for a major part (76%) of the cold start time. While some recent work proposes to speed up the loading phase through efficient model weights loading [14] and model structure initialization [19], they overlook unique stages introduced by the LLM inference, i.e., KV cache initialization and CUDA graph capturing. These stages account for 50% of the loading phase duration.

Both above stages are introduced for improving the serving throughput. KV cache [36] is a de facto solution employed by inference frameworks [23, 45, 46] to reduce recomputation overhead. By caching the context of earlier tokens, KV cache eliminates repeated computation when generating new output tokens in the sequence. Inference frameworks [23, 45, 46] also employ CUDA graphs [30, 32] to reduce the CPU-induced kernel launching overhead. Launching each kernel individually by the CPU incurs non-negligible overhead, as kernel execution on the GPU can be as fast as microseconds. CUDA graph groups multiple GPU kernels into a single graph, enabling the submission of multiple kernels via a single CPU launching instead of launching each kernel individually. By leveraging the CUDA graph, we observe the performance acceleration can be up to 2.4× in our experiments (Figure 3).

However, we find that the two stages are the main culprit of cold start latency; both of them require time-consuming profiling and construction during runtime, accounting for 50% of the loading phase duration (Figure 1). In the KV cache initializaiton stage, the primary task is to determine the available free GPU memory that can be allocated to the KV cache. A typical solution [23] is *profiling forwarding*, where the framework would run a model forwarding with a maximum sequence length and a maximum batch size, and profiles the residual free GPU memory space. In the capturing stage, CUDA graphs are constructed through *capturing forwarding*. Specifically, the framework launches normal model forwarding surrounded by the CUDA graph capturing API calls for different batch sizes [23]. This process subsequently generates a low-level kernel execution graph. In this graph, nodes record parameters (e.g., data pointers) and kernels' addresses, while edges represent execution dependencies. Simply removing these stages would introduce unacceptable performance degradation, placing us in a dilemma between long cold start latency and low inference serving throughput.

We propose *state materialization* to mitigate the prolonged cold start latency. This approach involves estimating the available free GPU memory allocated for the KV cache and saving the CUDA graph states offline. During a cold start, these states are restored and reused, significantly reducing latency. While available KV cache space can be derived via profiling, materializing CUDA graphs raises the following challenges. First, the data pointers in CUDA graph node parameters are subject to change, as there is no guarantee that addresses returned by malloc remain constant between different launches. This variability makes a correct restoration of data pointers complicated. Second, to materialize and restore the correct kernels, we can traverse the symbol tables of dynamic-link libraries using the kernel's mangled name. However, not every kernel is present in the symbol tables (e.g., some cuBLAS kernels), making it challenging to locate these kernels during runtime.

To address the first challenge, we observe that, in contrast to the non-deterministic nature of data pointers, the control flow exhibits strict determinism. Such a deterministic control flow forms a strong correlation between the data pointers and the buffer allocation sequence. Based on this observation, we identify the corresponding indexes of data pointers in the buffer allocation sequence offline and restore the correct data pointers by replaying the buffer allocation sequence during runtime (§4). To address the second challenge, we leverage *triggering-kernels* to trigger kernels loading at the CUDA module level, and use the kernel's name to locate the corresponding kernel's address (§5).

We build Medusa, a serverless LLM inference system to apply state materialization. We evaluate Medusa on 10 popular LLM models including Falcon [6], Llama2 [49], Qwen1.5 [47], and Yi [56] from huggingface [46]. For a given model, Medusa materializes and restores CUDA graphs for 35 different batch sizes (same as in vLLM [23]) with a total number of CUDA graph nodes of 139364. Medusa reduces the latency of the loading phase by 42.5%. Under real-world LLM inference workload (ShareGPT [48]), Medusa reduces the tail latency of the time to first token (TTFT) by 53.0%.

2 Background and Motivation

The cold start overhead is a notorious issue for serverless inference in practical applications. Despite a rich body of existing efforts [2, 8, 13, 15, 25, 34, 40, 43, 50, 54, 57] dedicated to optimizing it, this issue in the scenario of LLM has not yet been thoroughly explored. In this section, we first provide an overview of all stages involved in LLM inference cold start, along with a quantitative breakdown of the cost

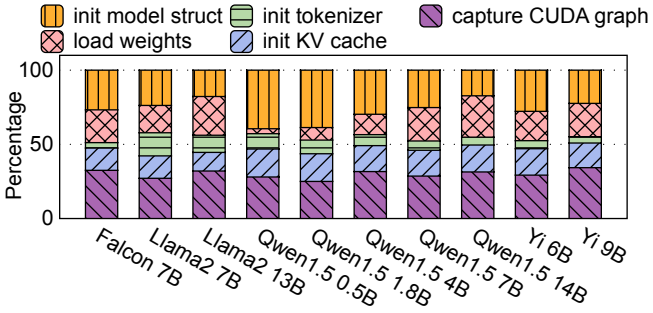

Figure 2. Breakdown of the loading phase.

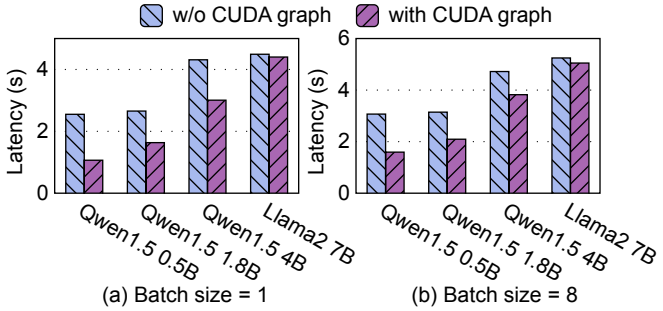

Figure 3. Acceleration brought by the CUDA graph.

in each stage (§2.1). According to our breakdown, the capturing stage constitutes a significant portion; therefore, we introduce CUDA graphs (§2.2) and analyze the reasons why their capturing costs a long latency (§2.3). Finally, we discuss the opportunities and challenges for accelerating cold starts in serverless LLM inference.

2.1 Cold Start of Serverless LLM Inference

As shown in Figure 1, we break down the cold start procedure in the vanilla vLLM [23] system using Qwen1.5 4B as an example. It mainly consists of three phases: 1) initializing runtime phase (22%), which includes loading Python code and packages, as well as setting up the execution environment; 2) **loading phase (76%)**, which loads model-specific data and performs warm-up profilings; and 3) generating the first token (2%).

With the help of existing works [15, 27, 40, 42], the initializing runtime phase can be totally eliminated. For example, we can leverage a pool of warm containers which have already loaded the user codes and Python packages [2, 33, 34]. Thus, our work focuses on the **loading phase**, whose overhead predominates. We further breakdown the loading phase into the following five stages. ❶ Model structure initialization stage. This stage involves instantiating the structure of the model, e.g., the number of layers and each kernel within the layers. It would also allocate tensors' buffers on the GPU to store model weights. ❷ Model weights loading stage. This stage loads model weights from persistent storage media into the pre-allocated tensors in GPU. ❸ Tokenizer loading stage. In this stage, the framework loads the tokenizer specified by each model into memory, which would be used to convert user input strings into a series of token IDs. ❹ KV cache initialization stage. For better GPU memory management, the framework manages the KV cache at the granularity of blocks [23]. In this stage, the framework should first get the available GPU memory for KV cache blocks, which is achieved by a *profiling forwarding*, i.e., launching a model forwarding with a maximum sequence length and maximum batch size, and profiles the residual free GPU memory space. Then, the framework allocates KV cache blocks based on the available free GPU memory space. ❺ CUDA graph capturing stage. The CUDA graphs are used to accelerate the model forwarding during serving inference. This phase creates CUDA graphs for different batches, which is done by first launching *warm-up forwarding*, followed by *capturing forwarding*. The warm-up and capturing are repeated several times given different batch sizes.

Figure 2 further quantitatively shows each stage's latency in the loading phase across ten different LLM models, using vanilla vLLM [23]. We can see that the following two stages stand out. First, the KV cache initialization stage is time-consuming (18%) since it requires profiling forwarding with a long sequence and large batch size. Second, CUDA graphs play a pivotal role in performance optimization (§2.2), however, capturing them introduces notable overhead, i.e., 32% of the loading phase. Overall, these two stages account for 47% on average among different models in the vanilla vLLM. Note that the remaining stages, such as the model structure initialization stage and the model weights loading stage, can be further optimized with existing works [14, 19], which will make the proportion of the KV cache initialization stage and CUDA graph capturing stage even more significant.

2.2 CUDA Graph

CUDA graph is a feature released in CUDA 10 to improve the model forwarding performance by reducing the CPU-induced kernel launching overhead. It is widely adopted by many LLM inference frameworks [23, 44, 45]. Fast advances in GPU architectures make GPU performance grow rapidly, resulting in kernel execution in GPU even as fast as microseconds. As a result, launching each kernel individually by the CPU incurs non-negligible overhead. CUDA graph organizes multiple GPU kernels into a graph, which describes the kernels to execute and their dependencies. Instead of launching each kernel individually, the CUDA graph enables launching a graph of kernels via a single CPU launching.

The benefits of CUDA graphs. CUDA graph can significantly reduce inference latency. We conducted an experiment to evaluate inference latency with and without CUDA graph, as shown in Figure 3. For this experiment, the model was

preloaded onto the GPU and prepared for inference, allowing us to assess the pure acceleration provided by the CUDA graph during the inference process. The detailed experimental settings are described in §7. We used a prompt length of 161 tokens and an output length of 338 tokens, which are the average lengths in the popular ShareGPT dataset [48]. Inference latency was measured as the time duration from the beginning of prefilling to the end of the last token's generation. Our results show that CUDA graph can achieve performance acceleration of up to 2.4×.

Two ways to build CUDA graphs. There are two ways to build CUDA graphs. The first is with explicit API calls, which require experts to build a CUDA graph by adding each kernel node individually using the low-level APIs, such as `cudaGraphAddKernelNode()`. However, this method is not supported by high-level frameworks like PyTorch [7] due to its impracticality. For example, some highly optimized but closed-source kernels, such as cuBLAS kernels [29] are hidden from user applications and cannot be explicitly added to the CUDA graphs because their kernel addresses are unknown.

As a result, it is more efficient to build CUDA graphs in a second way, i.e., stream capture, which only requires running a model forwarding wrapped by the capturing API calls provided by CUDA. The CUDA driver would record all the kernels as well as their dependencies within the capture range, and automatically build the CUDA graphs without handcrafting. vLLM as well as other popular inference frameworks [23, 45, 46] leverage this approach by default.

A typical workflow of capturing the CUDA graph is shown in Figure 4(a), where capturing is done by model forwarding. During capturing, the CPU launches the underlying kernels as shown in Figure 4(b), and the CUDA driver records all the launched kernels as well as their execution dependencies. After the capturing, a CUDA graph is built automatically by the CUDA driver with the nodes representing the kernels, and the edges representing the execution dependencies between kernels (shown in Figure 4(c)). The nodes of the CUDA graph could be inspected by the CUDA API to get its kernel address and parameters. As shown in Figure 4(d), a CUDA graph node contains the kernel address, the pointer to the array of all parameters, the number of parameters, and the size of each of them. The dependency represented by edges forces the correct execution order from the source node to the destination node.

The limitations and characteristics of capturing. There are also some limitations when using capturing. For example, capturing multiple CUDA graphs within a process at the same time would cause CUDA-related errors. As a result, each CUDA graph should be captured one by one.

After capturing the CUDA graph, the subsequent model forwarding could be performed by replaying the CUDA graph, which causes all kernels recorded in the CUDA graphs to be executed and produces the output identical to a normal

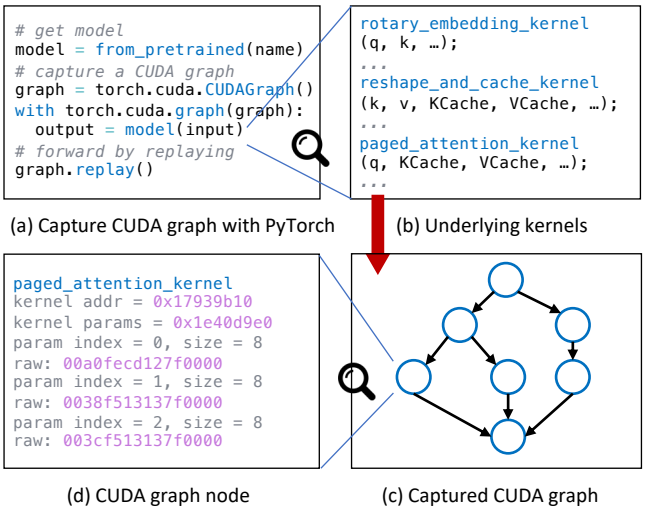

Figure 4. CUDA graph capturing and the node contents.

model forwarding. The CUDA driver takes care of the correct execution order of all kernels, and the kernels are executed according to their given parameters, which we refer to as *self-replaying*.

Across different self-replaying of the CUDA graph, kernels read and write using the given data pointers recorded during graph capture. Therefore, one must ensure that these pointers consistently reference the same buffers as they did during the initial capture [37]. To this end, in the capturing stage, PyTorch [7] reserves these allocated buffers and never frees or returns them to the CUDA driver afterwards; otherwise, replaying of CUDA graph might cause illegal memory accesses or data corruption.

2.3 Analysis of CUDA Graph Capturing Overhead

Capturing offers a more efficient method for building CUDA graphs. However, capturing the model forwarding process introduces delays, which contradicts the need to minimize cold start latency. Moreover, two factors exacerbate the overhead associated with capturing CUDA graphs.

Warm-up. First, warm-up forwarding is required [21, 28] before the actual capturing process. This necessity arises because certain CUDA API calls are prohibited during the capturing, such as device (or stream) synchronization [20, 31]. However, the initial launch of some kernels requires initializing the runtime or context, which internally triggers these CUDA APIs. For instance, the first call to cuBLAS functions in PyTorch would internally trigger cuBLAS initialization, which invokes synchronization, causing the capturing to fail.

Multiple batch sizes. Second, a CUDA graph is completely bound up with a specific batch size. Since changes in input result in variations in the argument shapes of the underlying kernels, different batch sizes necessitate capturing the CUDA graph multiple times, further worsening cold start latency.

For example, in vLLM [23], capturing 35 different batch sizes accounts for 30% of the loading phase on average (Figure 2).

2.4 Potential Solutions and Limitations

Hot spares are unaffordable. Existing works employ hot spares to mitigate cold-starts [9, 51, 55]. This approach provisions a collection of ready-to-serve instances for bursty request rates. While applicable to serverless LLM inference, it leads to resource wastage during periods of low request rates. Furthermore, the diversity of model types makes it unaffordable to over-provision for every type of model.

Asynchronous execution of stages is insufficient. An intuitive approach to reduce the latency is executing different stages simultaneously in the loading phase, thus hiding the latency. However, a naive asynchronous execution could not hide all the latency due to the following reasons. First, the dependencies between different stages necessitate a ordered execution of them. For example, the capturing stage requires model forwarding which in turn relies on the KV cache. This dependency mandates that the KV cache initialization stage precede the capturing stage. Second, the model weights loading involves disk I/O, which could be executed in parallel with the other stages, i.e., the tokenizer loading (CPU involves) and KV cache initialization (CPU + GPU involves) stages. However, our analysis reveals that it is insufficient to fully conceal the latencies of the two stages. As illustrated in Figure 2, for 60% of the models, the model weights loading stage does not fully cover the combined latency of the tokenizer loading and KV cache initialization stages[1].

Deferring the capturing stage is ineffective. Instead of capturing all CUDA graphs for different batch sizes in the loading phase, we could defer this process until serving inference requests. This involves running warm-up and capturing CUDA graphs of a given batch size when serving the corresponding request batches. However, this method does not fundamentally eliminate the capturing latency; it merely delays and disperses it across different requests in the inference serving phase.

2.5 Our Approach and Challenges

To reduce the latency of the KV cache initialization stage and the capturing CUDA graphs stage, we identify the long latency is attributable to the *dynamic* profiling or construction in the runtime. Specifically, the KV cache initialization acquires the available free GPU memory space through a dynamic profiling forward; the capturing stage requires multiple model forwarding to construct the CUDA graphs. To mitigate the overhead incurred by runtime dynamic operations, we propose to *materialize* the size of the available free GPU memory used by KV cache (abbreviated as materialize KV cache initialization), and CUDA graphs in an offline phase and restore them efficiently in the online phase, which introduces less latency compared with dynamic profiling or construction in the runtime.

The process of materializing the KV cache primarily involves saving and restoring the available free GPU memory size. As for the materialization of CUDA graphs, we observe that the model structure remains unchanged during every cold start, resulting in a near identical CUDA graph, i.e., only the address of the kernels and parameters may change. So it is also applicable to materialize the CUDA graphs conceptually. However, it is nontrivial due to the low-level and ready-to-execute nature of the CUDA graph. Specifically, the CUDA graph records the raw pointers address of the kernel and the parameters, which are subject to changes in every launch, making blindly dumping and loading the memory address space infeasible. We detail the challenges as follows.

Challenge I: Non-deterministic data pointers in parameters of CUDA graphs. The kernel parameters fall into two categories, including the *constants*, e.g., 4-byte integers, and the *data pointers*, e.g., the address of the input and output buffers. The constants remain unchanged in the CUDA graph between different launches, so it is straightforward to materialize and restore them. However, the data pointers require more sophisticated considerations since there is no guarantee that `cudaMalloc()` returns the same address across different cold starts. Such non-determinism requires us to amend the data pointers in the restored CUDA graph nodes to the correct address which hold the required data of the kernel parameters.

Challenge II: Randomized and even hidden kernel address of CUDA graphs. In the offline phase, we can get a CUDA graph after capturing and inspecting the kernel address in the graph nodes' information. However, the address is subject to change since the process address space is randomized in every launch. So, instead of saving the kernel address in the offline phase, MEDUSA materializes the *kernel's mangled names*. During the online phase, we can load the kernel mangled names, and traverse the symbols tables of dynamic-link libraries to locate the corresponding kernel address. However, it does not always work since not every kernel is present in the symbol tables. For example, we find that cuBLAS kernels can not be found in the symbol table of the cuBLAS libraries.

3 MEDUSA: Design Overview

To accelerate the serverless LLM inference cold start, we propose MEDUSA to reduce the overhead of the KV cache initialization and capturing CUDA graphs in the runtime by materializing and restoring the CUDA graphs.

MEDUSA includes two phases: *offline phase* and *online phase*. The goal of the offline phase is to materialize the

[1]Note that the breakdown shows the vanilla time series where the loading weights stage is synchronously executed.

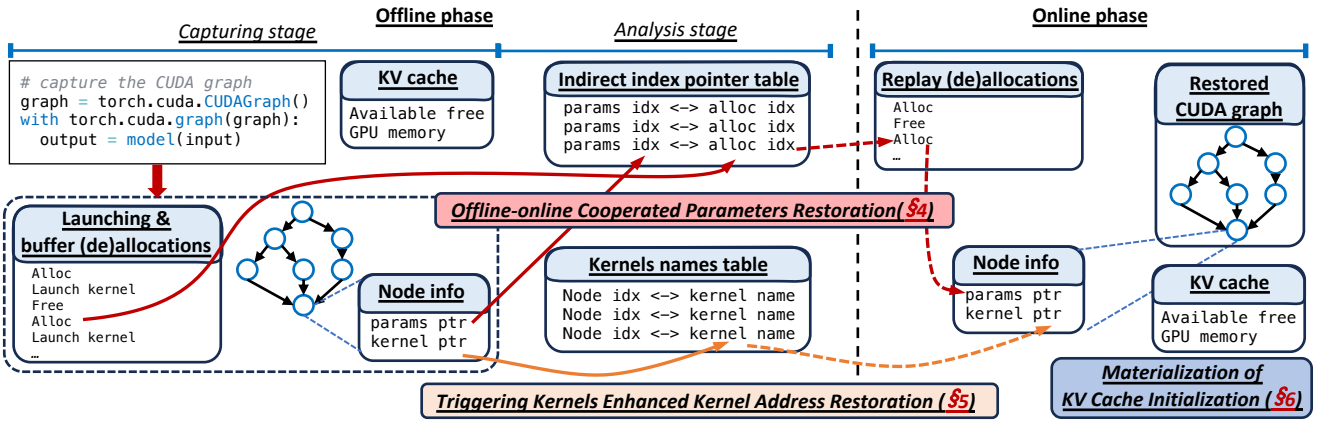

Figure 5. Overview of MEDUSA.

KV cache initialization and CUDA graphs, and the online phase is a cold start where MEDUSA restores the previously materialized KV cache initialization and CUDA graphs instead of dynamically profiling or constructing them. Notably, the offline phase is executed only once for each unique combination of <GPU type, model type>.

Key ideas. To overcome the indeterminacy of data pointers in parameters (*Challenge I*), we observe that, compared with the non-deterministic nature of data pointers, the control flow exhibits strict determinism. Each time an inference service starts, the control flow follows a strict order defined by the user's code – first allocate data buffers for storing the data, e.g., the model parameters, and then launches the kernels based on the returned address of the buffers, i.e., the data pointers. In terms of the data buffer allocations, the control flow would also allocate each layer's data buffers in order, due to the layers being initialized sequentially. Such a deterministic control flow forms a strong correlation between the data pointers and the buffer allocation sequence, i.e., the *i*-th data pointer is returned by the *i*-th buffer allocation.

Leveraging this property, MEDUSA introduces an intermediate representation, called the *indirect index pointer* (§4.1), which points to the index in the buffer allocation sequence. For example, if a pointer is returned by the *i*th buffer allocation, its corresponding indirect index pointer is *i*. In the online phase, MEDUSA replays the buffer (de)allocation[2] sequence and restores the buffer pointers address in the CUDA graph nodes using these indirect index pointers. Given an indirect index pointer *i*, MEDUSA uses the address returned by the *i*-th buffer allocation as the pointer and restores it to the CUDA graph nodes. MEDUSA also introduces a *copy-free buffer contents restoration* (§4.3) to reduce the overhead of restoring the buffer contents.

To overcome the randomized and even hidden kernel addresses (*Challenge II*), MEDUSA leverages *triggering-kernels* (§5) to trigger the loading of kernels by forcing the CUDA driver to load their *modules*, which contains the kernels we need to restore the CUDA graph. MEDUSA then uses the materialized kernel's name to traverse the modules and restore the correct kernel address.

Workflow. As shown in Figure 5, the offline phase further splits into two stages, capturing stage and analysis stage. In the capturing stage, MEDUSA captures the CUDA graphs via a model forwarding, during which MEDUSA intercepts the kernel's launching as well as the buffer (de)allocation sequence for further analysis. At the end of the capturing stage, MEDUSA saves the CUDA graphs information including the nodes and the dependencies. In this stage, MEDUSA also performs a profiling forwarding to materialize the KV cache initialization by saving the available free GPU memory.

In the analysis stage, MEDUSA materializes the CUDA graphs by synthesizing the output of the capturing stage, i.e., the kernels' launching and the buffer allocation sequence, as well as each node's information. The analysis process finds the correlation between the pointers of the parameters and the corresponding index in the buffer allocation sequence, and records such correlation in the indirect index pointer table, which could be used in the online phase to restore the pointers of parameters (§4). Also, MEDUSA uses kernels name table to record the node's kernel name, which could be used to restore the kernel's address (§5).

During the cold start (online phase), MEDUSA loads the materialized CUDA graphs, and restores them to the ready-to-execute ones. MEDUSA first replays the buffer (de)allocation sequence and records the returned addresses. For the pointers of parameters, MEDUSA leverages the indirect index pointer table to find the corresponding pointers returned by the buffer allocations, and fills them to the CUDA graph node's parameters. For the kernels' address, MEDUSA uses triggering-kernels to force the module loading, and leverage the kernels' name to match the corresponding address. Worth noticing, the dependencies could be restored with the information

[2]Buffer deallocation is also replayed to prevent GPU memory overflow.

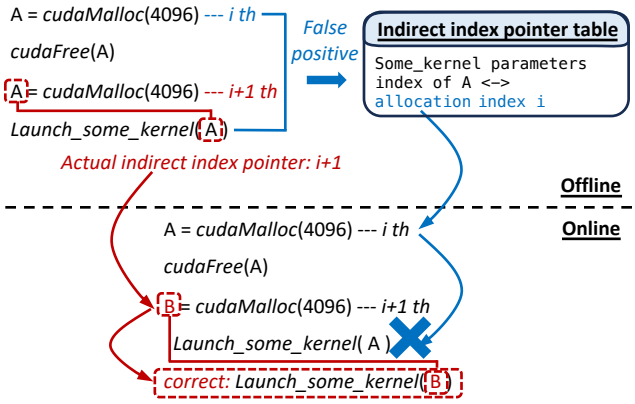

Figure 6. A false positive example (in blue), and the correct one (in red) in indirect index pointers substitution.

of the source node's index and the destination node's index, which have already been saved during the offline phase. MEDUSA also accelerates KV cache initialization by restoring the pre-saved available free GPU memory space (§6), hence eliminating the profiling forwarding in the online phase.

4 Offline-online Cooperated Parameters Restoration

The kernel's parameters fall into two categories: *constants* and *data pointers*. Constants, for example, are 4-byte integers. Data pointers are 8-byte values that point to the buffers the kernel accesses during execution. The constants and data pointers can be easily distinguished, as the pointers are 8 bytes long and usually begin with a high address prefix. However, since the high address prefix may also contain false positive candidates (which are rare), MEDUSA validates and corrects these exceptions afterwards. To validate, we run a model forwarding and compare the outputs of the original and speculative versions of CUDA graphs. MEDUSA can then validate the speculations and report errors when the results mismatch.

To materialize and restore constants, we directly save their plain values, and restore them in the online phase. In contrast, materialize and restore *data pointers* is non-trivial, as the data pointers recorded in the CUDA graph node are subject to change across different cold starts due to the non-deterministic addresses returned by malloc. Consequently, simply using the data pointers materialized in the offline phase would lead to invalid memory access. We observe that, the deterministic control flow forms a strong correlation between the data pointers and the buffer allocation sequence. We propose to use the *indirect index pointers*, i.e., the corresponding index in the allocation sequence. MEDUSA employs an offline-online cooperative parameters restoration approach, which analyzes and constructs an indirect index pointer table in the offline phase and restores the parameters with the help of the indirect index pointer table in the online phase.

4.1 Indirect Index Pointers Analysis

In the offline phase, MEDUSA analyzes and transforms all data pointers to their corresponding index in the allocation sequence, i.e., indirect index pointers. To do so, MEDUSA intercepts all allocation calls, and records the returned addresses. Given a data pointer in kernel parameters, MEDUSA searches it in the returned addresses to find a match, and then materializes it as the indirect index pointer.

Note that simply matching data pointers may cause false positives. Multiple matches could be found since the buffers could be allocated and deallocated multiple times, and the following allocation may return the same address which is deallocated before, resulting in a false positive as matching the wrong indirect index.

As shown in Figure 6, in the offline phase, the i-th and $(i+1)$-th allocations return the same address 'A'. Then, 'some_kernel' uses the $(i+1)$-th returned address 'A' as its parameter. Given the kernel's parameter 'A', since there are two matches in the allocation sequence, false positive indirect index pointers to i would cause data corruptions as follows. In the online phase, since the allocation returned address in a non-deterministic way, the i-th allocation may differ from the $(i+1)$-th allocation, i.e., 'A' and 'B' respectively. However, the false positive indirect index pointer would restore the data pointer to the ith allocation returned address 'A', resulting in data corruptions.

Trace-based indirect index pointers analysis. We observe that, although the buffers are allocated and deallocated, the kernels would always use the allocated buffers before they have been deallocated. So, MEDUSA further intercepts the `cudaLaunchKernel()` to analyse along with the allocation sequence in the offline phase. With the sequence of the allocation and `cudaLaunchKernel()`, MEDUSA matches the pointers address of a kernel backwards from its corresponding `cudaLaunchKernel()`, and records the first match as the indirect index pointers. Note that, the pointer addresses are matched when the addresses are identical or within the range of the allocated buffer.

4.2 Replaying-Based Data Pointers Restoration

In the online cold start phase, MEDUSA restores the data pointers based on the indirect index pointers. The online phase follows an identical control flow as in the offline phase, except it eliminates CUDA graph capturing. Thus, MEDUSA replays the buffer allocation (and deallocation) sequence and records all the returned addresses. Then, for a given kernel, MEDUSA leverages the indirect index pointers to obtain the corresponding correct pointer addresses and fills them into the CUDA graph node.

4.3 Copy-Free Buffer Contents Restoration

Besides restoring the pointers, Medusa should also make sure the buffer's contents are restored correctly. The intuitive is dumping all the buffer pointed by the CUDA graph nodes' parameters in the offline phase and restoring them by reading and copying in the online phase. However, *saving the buffer contents of every pointer is both expensive and unnecessary*. There are two types of buffers whose contents could be skipped from saving and restoration. First, the buffers of model parameters contain the model weights loaded from the storage before capturing the CUDA graphs, so there is no need to save these buffers again. Second, the temporary buffers are allocated to store the intermediate results between several kernels, and would be deallocated after the relevant kernels are finished. These temporary buffers could also be discarded since the contents are managed by the CUDA graph's kernels automatically. Identifying the model parameters buffers and temporary buffers to skip their contents restoration could greatly reduce the cost of data transfer.

Medusa identifies model parameters buffers with the following observation: the model parameters are already prepared before capturing the CUDA graphs. As a result, Medusa filters the pointers of model parameters out if they are allocated before the CUDA graphs capturing.

Medusa identifies temporary buffers by leveraging the *self-replaying* characteristics of the CUDA graphs. The kernels in the CUDA graph automatically read from and write to the buffers using recorded pointers. Consequenctly, if the buffers are deallocated after replaying is finished, it indicates that their contents can be discarded and will not be accessed in subsequent CUDA graph replays. This characteristic distinguishes temporary buffers from permanent buffers used across different launches of CUDA graphs. Temporary buffers can be identified if they are deallocated after capturing. In practice, we find that only 9.0% kernels require permanent buffers. Each of these kernels needs two permanent buffers, with each buffer contains only 4 bytes holding a magic number for launching. Therefore, restoring only the contents of the permanent buffers can significantly reduce the overhead.

5 Triggering-Kernels Enhanced Kernel Address Restoration

The goal of kernel address restoration is to fill the correct kernel addresses into CUDA graph nodes. These kernel addresses can vary across different cold starts; therefore, Medusa uses the kernels' mangled names for materialization and restoration. In the offline phase, Medusa materializes the kernels' mangled names and construct a mapping between each kernel's mangled name and its dynamic-link library. In the online phase, given the materialized kernel's mangled name and its dynamic-link library, we first `dlopen()` the library, then use `dlsym()` to get its function handler. Finally, we use `cudaGetFuncBySymbol()` to load it.

Most of the kernels (e.g. 69.2% in the Llama2 13B with the batch size of 1) can be restored in such a way. However, there are kernels whose addresses cannot be found in the dynamic-link library. For example, we find kernels in the cuBLAS library can not be found by in the symbol table of the dynamic-link libary (`dlsym()`). These kernels are close-source and are somehow hidden by not being shown in the symbol tables.

Instead, we resort to another way to find their kernel address by traversing the *module*. The module is a set of kernels, and the CUDA driver loads kernels in the granularity of the module, i.e., when loading a specific kernel, the CUDA driver would load the whole module containing the specific kernel as well as others within the module. As a result, by traversing a module, we can resolve multiple kernels' addresses at a time. Specifically, we utilize `cuModuleEnumerateFunctions()` to enumerate all kernel addresses within the module. Subsequently, we employ `cuFuncGetName()` to retrieve each kernel's name and compare it with the target kernel's name. Upon finding a match, we get the corresponding kernel's address and record it in the CUDA graph node for kernel address restoration.

In order to trigger the module loading, Medusa leverages *triggering-kernels*. There are some kernels within the target module which contain the target kernels we want to resolve. Once the triggering-kernels are executed by the CUDA driver, they force the CUDA driver to load the module we want, and then we can enumerate all kernels in the module and get the target kernel address by comparing the kernel names. At last, we fill the kernel address to the corresponding CUDA graph nodes.

5.1 Handwriting triggering-kernels

At first, we manually find triggering-kernels (usually matrix multiplication). For example, we find a certain kind of matrix multiplication can trigger to load the module which contains several cuBLAS kernels in the CUDA graph nodes. This simple method works well at first. It requires less than five kernels to trigger the loading of all the modules a CUDA graph needs. However, such human efforts become unacceptable because, it requires finding some new trigging kernels given different batch sizes, which makes us resort to a smarter way to trigger the module loading.

5.2 First layer as triggering-kernels

We find that the first layer of models is inherently a triggering-kernel. Most of the LLM models are composed of multiple structurally identical layers. For example, Llama2 13B has 40 layers, each containing the same kernels and some size of the input and output buffers. This feature inspires us that

Table 1. Models and their parameter sizes as well as the number of CUDA graph nodes.

model	Falcon 7B	Llama2 7B	Llama2 13B	Qwen1.5 0.5B	Qwen1.5 1.8B	Qwen1.5 4B	Qwen1.5 7B	Qwen1.5 14B	Yi 6B	Yi 9B
parameter size	13.4GB	12.6GB	24.2GB	1.2GB	3.4GB	7.4GB	14.4GB	26.4GB	11.3GB	16.4GB
CUDA graph nodes	14406	12518	16150	9118	9550	16150	12902	16350	12902	19318

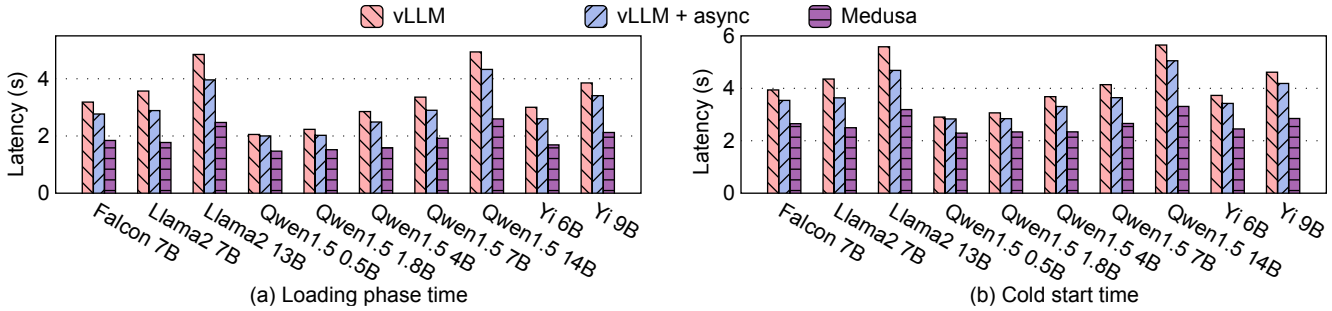

Figure 7. Overall loading phase time and cold start time.

the kernels of the first layer are inherent in the triggering-kernels, which could relieve us of the labour of finding the suitable triggering-kernels manually.

We now describe how we leverage the first layer kernels as the triggering-kernels to enhance our kernel address restoration process. During the online phase, we run and capture the first layer of the model, which is fast compared with capturing the whole model forwarding since it only accounts for a small proportion of the whole model (e.g., only 1/40 in Llama2 13B). After the CUDA graph of the first layer has been built, we enumerate all the CUDA graph nodes to get the kernel's corresponding address. Based on these addresses, we can fill all kernel addresses in the CUDA graph nodes since the kernels are repetitive in the following layers.

6 Materialization of KV Cache Initialization

The KV cache initialization requires the available free GPU memory to reserve a continuous buffer for the KV cache. In the vanilla implementation of vLLM [23], this is achieved through dynamic profiling forwarding at runtime. Notably, the "continuous" in this context refers to "a continuous chunk of GPU buffer." The KV blocks are still managed as separate "block bases" and reside within the continuous KV cache buffer, which is exactly the default implementation in vLLM [23].

We now present how to reduce the latency of the KV cache initialization stage through materialization. The invariance here is that given the same model and GPU type, the profiling forwarding would result in the same available free GPU memory space. In the offline phase, MEDUSA runs a profiling forwarding and records the available free GPU memory. In the online phase, MEDUSA directly take the offline recorded value as available free GPU memory, hence eliminating the runtime dynamic profiling overhead.

7 Evaluation

Platform setup. We use one server equipped with 4 A100-40GB SXM4 GPUs interconnected via NVLink. The host has 2 AMD EPYC 7642 CPUs and 256GB DDR4 memory. Storage devices include 4 Optane P5800X SSDs (400GB each). The software environment includes PyTorch 2.2.0 and CUDA version 12.4. While application trace experiments utilize all 4 A100 GPUs, other experiments are conducted on a single A100 GPU.

Compared strategies.

- vLLM [23] is an efficient inference framework that manages KV cache at the granularity of blocks to reduce memory fragmentation. Each stage in the loading phase of vLLM operates in a synchronized manner.
- vLLM + ASYNC represents the vLLM plus naive asynchronous model weights loading. The model weights loading stage is asynchronous with the tokenizer loading stage and the KV cache initialization stage.
- MEDUSA enables all optimizations including the materialization of CUDA graphs, and the materialization of the KV cache initialization.

7.1 Models

We use 10 popular LLM models from huggingface [46], including Falcon [6], Llama2 [49], Qwen1.5 [47], and Yi [56]. Table 1 shows their parameter sizes and the total number of the CUDA graph nodes with 35 different batch sizes. Capturing CUDA graphs of 35 different batch sizes is a default strategy as in vLLM [23], and we keep it in all experiments.

7.2 Overall Performance

We evaluate the loading phase latency using three different strategies in Figure 7(a), and the overall cold start latency in Figure 7(b). We make the following observations.

First, Medusa can reduce the loading phase latency effectively (Figure 7(a)). The average loading phase latency reduction is 42.5% compared with vLLM, and 34.4% compared with vLLM + async. This demonstrates that the naive asynchronous model weight loading is limited in the aspect of hiding all the latency. Instead, Medusa further enables the materialization of the CUDA graph, which reduces the latency of the capturing stage (including warm-up and capturing forwarding). Medusa also employs the materialization of KV cache initialization to eliminate the dynamic profiling of available GPU memory to allocate KV cache. By reordering the KV cache initialization stage before the model weights loading stage, Medusa also makes the warm-up of the capturing stage be executed in parallel with the model weights loading stage, further reducing the loading phase latency.

Second, by reducing the latency of the loading phase, Medusa can reduce the overall cold start latency effectively (Figure 7(b)), with 34.9% on average. The loading phase constitutes the predominant portion of the overall cold start latency. As a result, by reducing the overhead of the loading phase, Medusa eventually reduces the overall latency.

Third, Medusa is effective across different types of models with different parameter sizes. Among all models, Medusa demonstrates the most significant latency reduction when applied to Llama2 13B, with 42.9% compared with vLLM. The least latency reduction is observed in Qwen1.5 0.5B, which is 21.1% compared with vLLM.

7.3 Breakdowns of Medusa

To better understand the benefit of our designs, we measure the latency of each stage in the loading phase and report them in Figure 8. The example model is Qwen1.5 4B, with the total model parameters size of 7.4GB.

Figure 8(a) shows the latency breakdown of vLLM, where each phase is executed in a synchronous manner. The total latency is 2.85 seconds. The model structure initialization stage takes 0.85 seconds, followed by the model weights loading stage of 0.39s. After the model weights loading stage, it takes 0.21 seconds to load the tokenizer, and another 0.50 seconds to initialize the KV cache. At last, the capturing

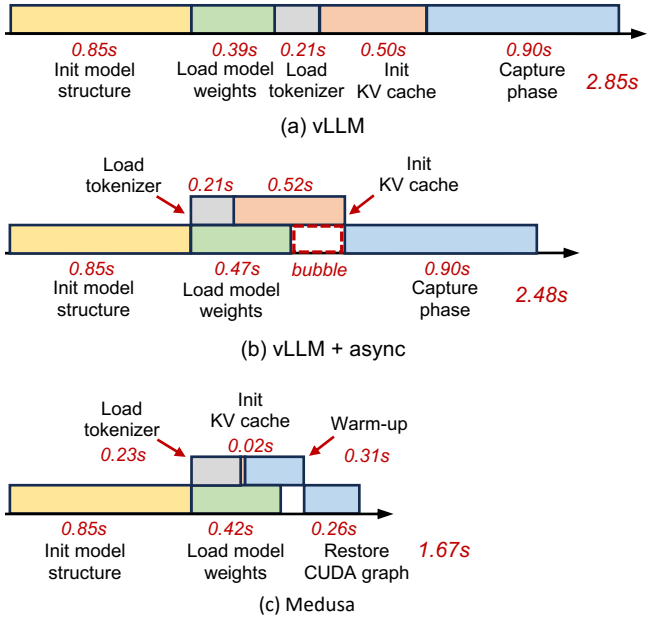

Figure 8. Breakdown of different strategies.

phase (including the warm-up and capturing forward) takes 0.9 seconds.

With the naive asynchronous weight loading (Figure 8(b)), the model weights loading is asynchronous with the tokenizer loading as well as the KV cache initialization, reducing the latency of loading phase by 13.0% compared with vLLM. We make two observations here.

First, model weights loading and the KV cache initialization exhibit mutual interference, increasing longer latency in model weights loading stage, i.e., 0.08 seconds. We observe similar interference as in other model's. The reason is that the KV cache initialization requires a profiling forwarding with the maximum sequence length and the maximum batch sizes, during which, we find [3] that some of the asynchronous CUDA memory copies issued by the model weights loading stage are blocked.

Second, the model weights loading stage could not cover all of the tokenizer loading and the KV cache initialization stages, leaving a 0.26 seconds bubble. We also confirm that 6 out of 10 models have such bubbles. This demonstrates the naive asynchronous execution could not hide all the latency.

Medusa further reduces the loading phase latency as shown in Figure 8(c) in the following two ways. First, Medusa enables the materialization in the KV cache initialization stage and CUDA graph capturing stage. As a result, the latency of KV cache initialization stage is greatly reduced since the profiling forward is eliminated, i.e. from 0.50 seconds to 0.02 seconds. With the materialization of CUDA graphs, Medusa reduces the capturing stage from 0.90 seconds to

[3]We use Nsight Systems tool, which is provided by NVIDIA.

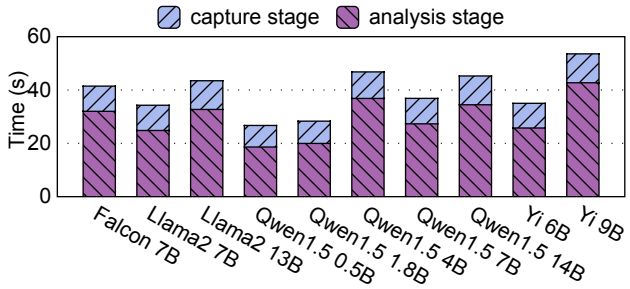

Figure 9. Overhead of offline phase.

0.57 seconds. This is because MEDUSA only needs to warm up and capture the first layer of the model, while the warm-up and capturing of subsequent layers can be skipped, as we can restore their CUDA graphs from materialization. This optimization reduces the capturing stage latency from 0.90 seconds to 0.57 seconds. Second, by reducing the KV cache initialization stage, MEDUSA gains the opportunity to execute the warm-up asynchronously with the model weights loading stage, further reducing overall latency. In summary, with the materialization of KV cache initialization and CUDA graphs, MEDUSA successfully reduces the bubble within the asynchronous execution, reducing the latency by 41.4% compared with vLLM, and 32.7% compared with vLLM + ASYNC.

7.4 Offline Phase Overhead Analysis

We now demonstrate the overhead associated with the offline phase. The offline phase consists of the capturing stage and analysis stage. In the capturing stage, MEDUSA runs a cold start process to record the memory space for allocating the KV cache block. Most importantly, MEDUSA captures the CUDA graphs for multiple batch sizes and materializes them in the analysis stage.

Figure 9 shows the breakdown of offline phase. It takes 39.2 seconds on average to perform an offline phase including a capturing stage and an analysis stage. The duration of the capturing stage is relatively constant, with an average of 9.7 seconds. The analysis stage needs to analyse 35 different CUDA graphs, accounting for a major part of the offline phase. In summary, the offline phase incurs relatively low overhead, requiring less than one minute to complete.

7.5 Application Traces

We further evaluate MEDUSA using real-world workloads. We use the ShareGPT dataset [48] as the real-world trace, which contains a collection of user-generated interactions with ChatGPT. The user request arrival pattern follows a Poisson distribution, and we vary the requests per second (RPS), using values of 2 and 10, to simulate the different levels of frequency.

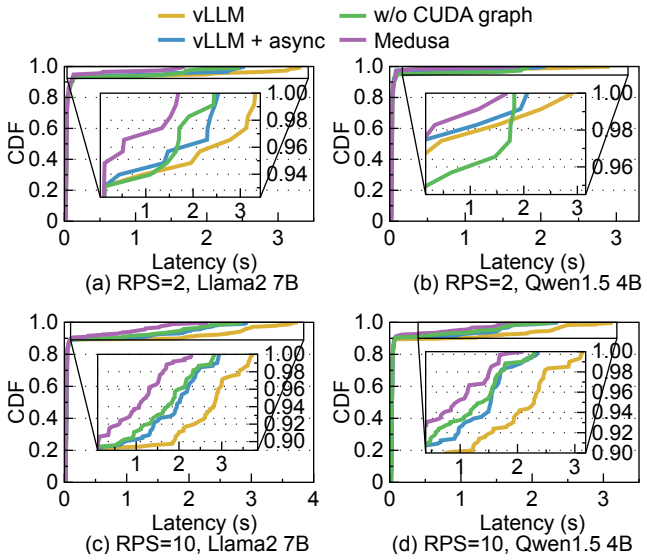

Figure 10. 99th-percentile tail latency of time to first token (TTFT) under different RPS with real-world traces.

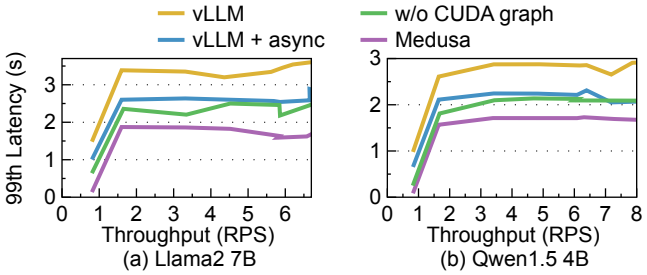

Figure 11. 99th-percentile tail latency of TTFT under different system overall throughput.

We leverage a pool of warm execution environments which have loaded the user codes and Python packages as the state-of-the-art works [2, 33, 34], so the runtime initialization is eliminated, and the time required to launch an inference serving instance is equal to the duration of the loading phase. Since cold starts primarily affect the time to first token (TTFT), we collect the TTFT and overall system throughput as the metrics. To understand the performance without CUDA graph, we add a new compared strategy w/o CUDA GRAPH. In this strategy, we remove the capturing stage in the cold start process, and the other parts of the cold start process follows the same as in vLLM. While this strategy reduces the cold-start latency, it comes at the cost of subsequent inference serving being unable to benefit from the acceleration provided by the CUDA graphs. We evaluate all the strategies with two models, namely Llama2 7B and Qwen1.5 4B.

As shown in Figure 10, MEDUSA can reduce the tail latency of TTFT under all settings. For example, in Figure 10(a), MEDUSA reduces the 99th-percentile latency by 50.5% compared with vLLM. Similarly, in Figure 10(b), MEDUSA achieves

a 53.0% reduction compared with vLLM. This is because Medusa can reduce the latency of cold start, and hence shortens the request waiting time in the pending queue, eventually resulting in a shorter tail latency of TTFT. We also observe similar results when changing the model to Qwen1.5 4B.

Compared with w/o CUDA graph, Medusa achieves lower 99th-percentile tail latency for two reasons. First, the end-to-end cold start latency of w/o CUDA graph is longer than Medusa. Simply removing the capturing stage fails to achieve the same latency as Medusa's, because the KV cache initialization stage still accounts for a significant part. Instead, Medusa also reduces the KV cache initialization latency by materialization. Second, removing the CUDA graphs results in increased latency for serving user requests, which also hurts the tail latency.

We incrementally increase the RPS to simulate more frequent client interactions and monitor two key performance metrics: the system's overall serving throughput and the corresponding 99th-percentile TTFT tail latency. The results are shown in Figure 11. Across all tested scenarios, Medusa consistently achieves the lowest tail latency compared to other approaches. At lower request rates, the tail latency remains relatively low for all systems, as a single inference serving instance can adequately handle the client's requests without launching additional instances. As the request rate increases, we observe a rise in tail latency across all approaches, attributed to the latency involved in launching new inference serving instances to meet the growing demand. Medusa outperforms the compared strategies (Figure 11(a)). When the system's overall serving throughput is around 4.5 QPS with Llama2 7B, the 99th-percentile tail latency of Medusa is 43.0%, 29.9%, and 27.0% lower than vLLM, vLLM + async, and w/o CUDA graph, respectively. Subsequently, when the request rate exceeds the system's processing capacity, the tail latency increases as the queueing time becomes dominant.

8 Discussion

Device-side memory allocations and indirect pointers. Memory allocations may occur within kernels (device-side), and data pointers could be indirect, such as pointers to arrays of pointers, complicating parameter restoration. Currently, Medusa only intercepts allocations from host-side APIs and handles direct pointers exclusively. Our empirical analysis of 10 models (comprising 139,364 CUDA graph nodes) revealed no instances of device-side allocations or indirect pointers in CUDA kernels. Despite their rarity, we maintain validation to ensure correctness (§4). Simple solutions could be employed to address these potential issues in future work; for example, intercepting device-side allocations could be implemented by adding a pass during compilation.

Multi-GPU support. Currently, Medusa supports the models that fit on a single GPU, which covers a wide range of use cases as a single 40GB A100 GPU can support Llama2 13B or smaller models. Regarding multi-GPU support, Medusa's core concepts remain applicable, particularly regarding KV cache and CUDA graph materialization. One potential future exploration is constructing the indirect index pointer table across multiple GPU instances. We make it future work.

9 Related Work

Serverless ML inference systems. There are lots of existing works [3, 4, 12, 24, 39, 55, 58, 59] aim to optimize the throughput of serverless ML inference through requests batching or scheduling. Medusa differs from these works by concentration on reducing the overhead associated with cold start. Phantom [14] is a recent work proposed to reduce the startup latency through fast model weights loading and locality-driven migration. However, it overlooks two critical stages in the cold start process: KV cache initialization stage and CUDA graph capturing stage. Medusa, in contrast, specifically addresses and optimizes the latency of these stages through materialization.

Optimizations for serverless cold start. A line of works [2, 15, 25, 27, 34, 40, 42, 57] focuses on speeding up the cold start of serverless through a set of warm containers. The warm containers have already undergone some runtime initialization, e.g., the codes or Python packages have been loaded, which turns a cold start into a warm start, thus reducing the latency. MITOSIS [54] argues that pre warm-up containers may not be sufficient to serve bursty requests, so it proposes to leverage remote forking to reduce the cold start latency. These works focus on eliminating the runtime initialization (such as VMs or containers), instead, Medusa focuses on the loading phase, and is orthogonal to those of works. Optimus [19] proposes to accelerate the model structure initialization stage by transforming an existing model with a similar structure in warm containers. On the other hand, Medusa focuses on optimizing other parts in the loading phase, i.e., KV cache initialization and CUDA graphs capturing.

Another line of works propose to accelerate cold start through checkpoint and restore [8, 10, 13, 18, 43, 50, 52]. They persist the complete states of a already launched instance into storage as checkpoints, and restore the checkpoints when a new invocation arrives. By leveraging checkpoints, the cold start latency could be shortened since once the restoration finished, the instance is ready to serve. Instead of taking a full checkpoint of the instance, Medusa only materializes the CUDA graphs and the information needed by the KV cache initialization, which is more lightweight and could be combined with these previous works.

10 Conclusion

This paper introduces Medusa, which mitigates cold start latency in serverless LLM inference by materializing KV

cache initialization and CUDA graphs. Its core concept can be generalized to other scenarios where dynamic profiling and construction can be eliminated through materialization. Medusa also presents the materialization of CUDA graphs, which could be applied to other systems leveraging CUDA graphs. Overall, Medusa significantly reduces cold start latency, enhancing serverless LLM inference efficiency.

Acknowledgements

We sincerely thank our shepherd Hubertus Franke for helping us improve the paper. We also thank the anonymous reviewers for their feedback. This work is supported by the National Natural Science Foundation of China (Grant No. 62332011).

A Artifact Appendix

A.1 Abstract

This appendix describes the workflow of Medusa. The source code and scripts for all experiments are publicly available at: https://github.com/thustorage/Medusa.

A.2 Artifact check-list (meta-information)

- **Model: 10 models listed in Table 1**
- **Data set: public ShareGPT dataset**
- **Run-time environment: CUDA version 12.4; root access**
- **Hardware: we will provide GPU and SSD**
- **Metrics: time-to-first-token (TTFT)**
- **Output: numerical results**
- **Experiments: automated scripts, and all results are within 5% variation**
- **How much disk space required (approximately)?: less than 150GB**
- **How much time is needed to prepare workflow (approximately)?: 3-4 hours**
- **How much time is needed to complete experiments (approximately)?: 2-3 hours**
- **Publicly available?: yes**
- **Code licenses?: Apache-2.0 license**
- **Data licenses?: Apache-2.0 license**

A.3 Description

A.3.1 How to access.

- https://github.com/thustorage/Medusa.

A.3.2 Hardware dependencies.

- 2 AMD EPYC 7642 CPUs
- 256GB DDR4 memory
- 4 Optane P5800X SSDs (400GB each)
- 4 A100-40GB SXM4 GPUs interconnected via NVLink

A.3.3 Software dependencies.

- PyTorch 2.2.0
- CUDA version 12.4

A.4 Installation

Run command MAX_JOBS=32 python setup.py develop to install Medusa.

A.5 Experiment workflow

Several directories should be created for logs and other intermediate data:

Run ./scripts/mktmpfs.sh to create directories for storing logs.

Run python scripts/serverless_llm.py –save_tensor to save model parameters into the SSDs.

Run python scripts/serverless_llm.py –offline to save CUDA Graph.

After creating these directories, follow the steps below to reproduce all results.

A.6 Evaluation and expected results

- **Figure 7 and Figure 2:**
  Run mkdir experiments/overall to create the directory for logs.
  Run python scripts/overall.py > results/Figure7 for Figure 7. This experiment shows the overall loading phase time and cold start time.
  Run python scripts/breakdown.py > results/Figure2 for Figure 2. This experiment shows the breakdown of the loading phase given all evaluated models.
- **Figure 3:**
  Run mkdir experiments/cuda_graph to create the directory for logs.
  Run python scripts/cuda_graph.py > results/Figure3 for Figure 3. This experiment shows the acceleration brought by the CUDA graph given four models.
- **Table 1:**
  Run python scripts/calculations.py > results/Table1 for Table 1. This experiment shows the models and their parameter sizes as well as the number of CUDA graph nodes.
- **Figure 9:**
  Run mkdir experiments/offline to create the directory for logs.
  Run python scripts/offline.py > results/Figure9 for Figure 9. This experiment shows the overhead of the offline phase.
- **Figure 10:**
  Run mkdir experiments/traces;
  mkdir experiments/traces/qps2;
  mkdir experiments/traces/qps10 to create the directory for logs.
  Run python scripts/traces.py > results/Figure10 for Figure 10. This experiment shows the 99th-percentile tail latency of time to first token (TTFT) under different RPS with real-world traces.
- **Figure 11:**
  Run mkdir experiments/traces_throughput to create the directory for logs.
  Run python scripts/traces_throughput.py
  > results/Figure11 for Figure 11. This experiment shows the 99th-percentile tail latency of TTFT under different system overall throughput.
- **Figure 8 and Figure 1:**
  Run python scripts/breakdown_Qwen.py

> results/Figure8_Figure1 for Figure 8 and Figure 1. This experiment shows the breakdown of different strategies.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 69. https://doi.org/10.1109/SC41405.2020.00073

[4] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing Inference Serving on Serverless Platforms. *Proc. VLDB Endow.* 15, 10 (2022), 2071–2084. https://doi.org/10.14778/3547305.3547313

[5] Alibaba. 2024. *Platform for AI (PAI)-Elastic Algorithm Service.* https://www.alibabacloud.com/product/machine-learning

[6] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. The Falcon Series of Open Language Models. *CoRR* abs/2311.16867 (2023). https://doi.org/10.48550/ARXIV.2311.16867 arXiv:2311.16867

[7] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM. https://doi.org/10.1145/3620665.3640366

[8] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS made fast using snapshot-based VMs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 730–746. https://doi.org/10.1145/3492321.3524270

[9] AWS. 2024. *AWS SageMaker.* https://aws.amazon.com/pm/sagemaker/

[10] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 32:1–32:15. https://doi.org/10.1145/3342195.3392698

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, and Greg Brockman. et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[12] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[13] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 467–481. https://doi.org/10.1145/3373376.3378512

[14] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 135–153. https://www.usenix.org/conference/osdi24/presentation/fu

[15] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 386–400. https://doi.org/10.1145/3445814.3446757

[16] Github. 2024. *Github Copilot.* https://github.com/features/copilot

[17] Google. 2024. *Google GEMINI.* https://gemini.google.com

[18] gVisor team. 2024. *Checkpoint/Restore in gVisor.* https://gvisor.dev/docs/user_guide/checkpoint_restore/

[19] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. 2024. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 1039–1053. https://doi.org/10.1145/3627703.3629567

[20] Github issue. 2024. *CUDA Graph error caused by synchronization.* https://github.com/microsoft/onnxruntime/issues/15002

[21] Github issue. 2024. *Warm-up before CUDA graph capturing.* https://github.com/pytorch/pytorch/issues/99397

[22] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626. https://doi.org/10.1145/3600006.3613165

[24] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association. https://www.usenix.org/conference/atc22/presentation/li-jie

[25] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[26] Microsoft. 2024. *Azure ML*. https://learn.microsoft.com/en-us/azure/machine-learning

[27] Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*, Christina Delimitrou and Dan R. K. Ports (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[28] NVIDIA. 2024. *Accelerating PyTorch with Native CUDA Graphs Support*. https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41965/

[29] NVIDIA. 2024. *cuBLAS*. https://docs.nvidia.com/cuda/cublas/

[30] NVIDIA. 2024. *CUDA graph API*. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html

[31] NVIDIA. 2024. *CUDA graphs programming model*. https://www.olcf.ornl.gov/wp-content/uploads/2021/10/013_CUDA_Graphs.pdf

[32] NVIDIA. 2024. *Getting Started with CUDA Graphs*. https://developer.nvidia.com/blog/cuda-graphs/

[33] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Pipsqueak: Lean Lambdas with Large Libraries. In *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino (Eds.). IEEE Computer Society, 395–400. https://doi.org/10.1109/ICDCSW.2017.32

[34] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[35] OpenAI. 2024. *OpenAI ChatGPT*. https://openai.com/index/chatgpt/

[36] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. *CoRR* abs/2211.05102 (2022). https://doi.org/10.48550/ARXIV.2211.05102 arXiv:2211.05102

[37] PyTorch. 2024. *PyTorch CUDA Graphs*. https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/#api-example

[38] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. *CoRR* abs/2407.00079 (2024). https://doi.org/10.48550/ARXIV.2407.00079 arXiv:2407.00079

[39] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[40] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 753–767. https://doi.org/10.1145/3503222.3507750

[41] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Comput. Surv.* 54, 11s (2022), 239:1–239:32. https://doi.org/10.1145/3510611

[42] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Irfan Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[43] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[44] Huggingface team. 2024. *text-generation-inference*. https://github.com/huggingface/text-generation-inference

[45] MLC team. 2023. *MLC-LLM*. https://github.com/mlc-ai/mlc-llm

[46] Huggingface teams. 2024. *Huggingface*. https://huggingface.co

[47] Qwen teams. 2024. *Introducing Qwen1.5*. https://qwenlm.github.io/blog/qwen1.5/

[48] Sharegpt teams. 2023. *Sharegpt*. https://sharegpt.com

[49] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

[50] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 559–572. https://doi.org/10.1145/3445814.3446714

[51] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[52] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 39:1–39:16. https://doi.org/10.1145/3302424.3303978

[53] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[54] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 497–517. https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[55] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 768–781. https://doi.org/10.1145/3503222.3507709

[56] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Tao Yu, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. 2024. Yi: Open Foundation Models by 01.AI. *CoRR* abs/2403.04652 (2024). https://doi.org/10.48550/ARXIV.2403.04652 arXiv:2403.04652

[57] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. 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, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 335–350. https://doi.org/10.1145/3617232.3624871

[58] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. 2021. Gillis: Serving Large Neural Networks in Serverless Functions with Automatic Model Partitioning. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*. IEEE, 138–148. https://doi.org/10.1109/ICDCS51616.2021.00022

[59] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang