Towards On-the-Fly Snapshot Memory Compression for Low-Latency Elastic Inference Serving Systems

Radostin Stoyanov
radostin.stoyanov@eng.ox.ac.uk
University of Oxford
Oxford, United Kingdom

Viktória Spišaková
spisakova@ics.muni.cz
Masaryk University
Brno, Czech Republic

Adrian Reber
areber@redhat.com
Red Hat
Stuttgart, Germany

Andrei Vagin
avagin@gmail.com
CRIU
Seattle, Washington, USA

Rodrigo Bruno
rodrigo.bruno@tecnico.ulisboa.pt
INESC-ID, IST, Univ. of Lisbon
Lisbon, Portugal

Abstract

In-memory model caching and startup latency are key bottlenecks in large-scale AI serving systems, especially for GPU-accelerated large language model (LLM) inference in elastic, serverless environments. While container checkpointing enables hot starts, it introduces new challenges in memory footprint, storage bandwidth, and restore latency. Existing offline snapshot compression methods reduce snapshot size but add extra I/O, storage duplication, and decompression overhead. In this paper, we present CRIU-LZ4, a restore-optimized method for on-the-fly compression integrated directly into the CPU–GPU checkpoint and restore pipelines. Built atop CRIUgpu, CRIU-LZ4 performs page-level compression during memory transfer, eliminating intermediate artifacts and minimizing the latency on the restore critical path. Our evaluation results show that CRIU-LZ4 reduces cold-start latency by 46–59% and achieves up to 6× smaller snapshots compared to uncompressed GPU-aware checkpointing, while eliminating the decompression bottleneck of offline compression, significantly reducing both end-to-end restore time and peak disk usage.

CCS Concepts

• **Software and its engineering → Checkpoint / restart**; **Cloud computing**; • **Theory of computation → Data compression**.

EuroMLSys '26, Edinburgh, Scotland UK

ACM ISBN 979-8-4007-2605-7/2026/04
https://doi.org/10.1145/3805621.3807612

Keywords

Container Checkpoint/Restore, GPU Memory Compression, CRIU, Cold-start Latency, LLM Inference Serving

ACM Reference Format:
Radostin Stoyanov, Viktória Spišaková, Adrian Reber, Andrei Vagin, and Rodrigo Bruno. 2026. Towards On-the-Fly Snapshot Memory Compression for Low-Latency Elastic Inference Serving Systems. In *Sixth European Workshop on Machine Learning and Systems (EuroMLSys '26), April 27–30, 2026, Edinburgh, Scotland UK*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3805621.3807612

1 Introduction

The rapid adoption of generative and agentic Artificial Intelligence (AI) workloads has led to significant efforts from both industry and academia to develop efficient inference serving platforms [1, 2, 4, 13, 18, 20]. These platforms increasingly operate at scales where jobs span hundreds or thousands of GPUs and multiple terabytes of memory. To make hosting AI models economically viable, cloud providers rely on serverless platforms that combine multi-tenancy and "scale-to-zero" policies, aggressively evicting inactive models to improve hardware utilization. However, aggressive policies introduce a severe cold-start penalty: loading large models from storage into GPU memory and initializing the inference engine can take tens of seconds to minutes, directly increasing request latency for inference serving workloads. For example, initializing a 7–9B-parameter model with vLLM takes over 90 seconds, during which the GPU remains idle and requests queue.

To reduce cold-starts, state-of-the-art AI platforms increasingly rely on container snapshots to accelerate the startup of inference services [14, 19]. By capturing the execution state of a warmed-up model and inference engine, platforms bypass initialization entirely [24, 27], making restore latency the dominant recurring cost. However, snapshots store the

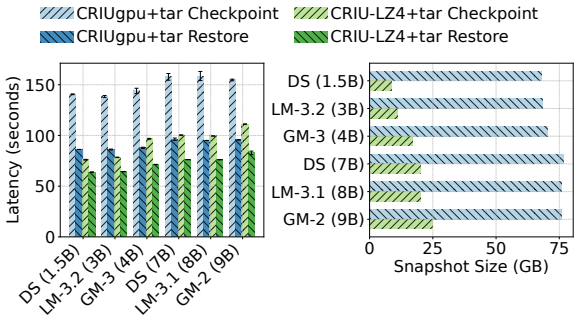

Figure 1: C/R latency (left) and snapshot size (right) for vLLM containers serving DeepSeek-R1 (DS), Gemma (GM), and LLaMA (LM) on an NVIDIA H100 80 GB GPU.

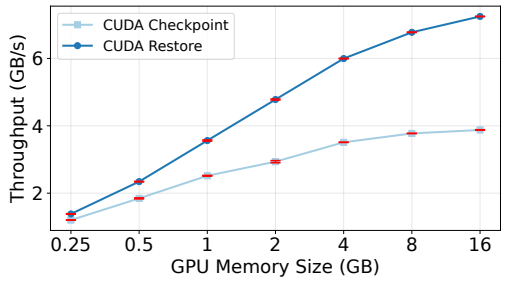

Figure 2: Throughput of the CUDA C/R API on an NVIDIA RTX 5090 GPU (PCIe 5.0 x16). Error bars show standard error over 10 runs.

entire memory footprint, shifting the bottleneck to host memory capacity and storage bandwidth [16, 26]. As LLM sizes grow, saving and restoring large snapshots introduces significant I/O overheads, limiting cluster efficiency and increasing operational costs [15].

A natural solution to reducing snapshot size is compression. Prior approaches apply generic compression methods after a snapshot is fully created (e.g., `gzip`, `zstd`) [21–23]. However, after-checkpoint-compression (we use the term offline for simplicity) introduces fundamental inefficiencies. The system must either (i) temporarily store the entire uncompressed snapshot in memory before compression completes, increasing peak memory usage, or (ii) write the uncompressed snapshot to storage and then re-read it for compression, incurring additional I/O and storage overhead.

Inspired by operating systems' in-memory compression mechanisms such as zram and zswap, we propose CRIU-LZ4, a restore-path optimized method for on-the-fly memory page compression, integrated directly into the checkpoint and restore pipeline. Built atop CRIUgpu [28], a unified CPU–GPU checkpointing framework extending CRIU [8], CRIU-LZ4 performs page-granularity compression during device-to-host memory transfer, eliminating large intermediate buffers and avoiding post-processing compression passes, without requiring additional runtime components.

The CRIU-LZ4 design is restore-path optimized in that compression and decompression are structured to minimize end-to-end container restore latency, and prioritize fast service resumption for inference workloads. In particular, CRIU-LZ4 pipelines compress and decompress memory pages as they are saved and restored, avoiding expensive archive compression stages and reducing both total restore time and peak disk usage.

Our evaluation shows that CRIU-LZ4 reduces cold-start latency by 46–59% compared to reinitializing inference engines from scratch, while reducing snapshot size by up to 6× relative to uncompressed GPU-aware checkpointing (see Figure 1). Compared to archive-based offline compression, CRIU-LZ4 restores containers up to 3× faster while using up to 3.7× less disk space.

In summary, this paper makes the following contributions:

- We analyze the initialization phase of state-of-the-art inference engines and the fundamental limitations of post-checkpoint (offline) compression when applied to accelerate container startup (§ 2 and § 3).
- We design and implement CRIU-LZ4, an on-the-fly, page-granularity compression integrated into the CPU–GPU checkpoint and restore pipeline, explicitly optimized to reduce end-to-end restore latency for model inference serving (§ 4).
- We evaluate CRIU-LZ4 with inference and training workloads using a set of Gemma, LLaMA, and DeepSeek models with different architectures and sizes (§ 5).

2 Background and Motivation

Serverless Inference Hot-Starts. Serverless inference platforms abstract the management of underlying compute resources by dynamically provisioning and reclaiming execution environments in response to demand. In container-based serverless platforms, the lifecycle of an inference instance typically follows four phases: (i) image fetch and container creation, (ii) model weights loading and initialization, (iii) memory profiling and key-value cache setup, and (iv) steady-state request serving. When an inference engine instance is terminated due to scale-down events, node failures, or rolling updates, the associated containers are restarted, re-executing this initialization sequence and incurring repeated cold-start overheads that increase latency and degrade overall system responsiveness [11].

Scaling decisions in serverless platforms aim to achieve a balance between maximizing hardware utilization and minimizing request tail latency, which is mostly impacted by how frequent *cold starts* are. In particular, cold starts occur

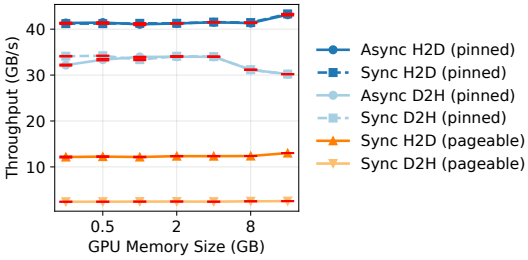

Figure 3: CUDA memory transfer throughput between host (H) and GPU (D) for pinned and pageable memory using synchronous and asynchronous copies.

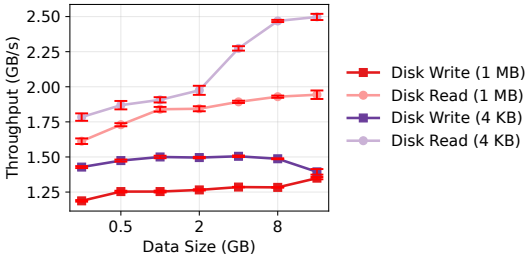

Figure 4: Sequential disk I/O throughput for reads and writes at 4 KB and 1 MB block sizes on an NVMe SSD. Error bars show standard error over 10 runs.

when a new inference engine instance must be provisioned from scratch, requiring container image loading, dependency initialization, model weight loading, GPU context creation, and memory allocation [3]. These steps typically dominate end-to-end latency, especially when serving large models, often taking from tens of seconds up to minutes. In contrast, restoring from snapshot of the pre-initialized execution environment, allows the system to start serving requests with minimal delays.

Snapshots as a Hot-Start Mechanism. Container checkpointing systems are a promising approach to transform cold starts into hot starts. Instead of re-initializing a model-serving engine from scratch, the Checkpoint/Restore (C/R) engine captures a snapshot of a fully initialized instance, including its memory state, open file descriptors, and runtime context, and later restores it on demand. Upon restoring from a snapshot, the instance resumes execution as if it had never been terminated, bypassing costly initialization steps such as model deserialization and memory pool setup.

This technique has gained significant traction in containerized AI inference stacks and Kubernetes-based deployments, where restoring a pre-warmed serving process can significantly reduce pod startup time [5, 19]. By checkpointing after model weights are loaded and GPU contexts are initialized, restore latency is reduced to only reconstructing process execution state and CPU–GPU memory, rather than repeating full model initialization.

C/R shifts the performance bottleneck from application initialization to host memory capacity and storage bandwidth (Figure 3), as snapshots must be stored and retrieved efficiently. Serverless environments amplify these challenges: restore must meet tight latency objectives, storage is often ephemeral or network-backed, and container-level memory limits leave little headroom for temporary buffers. Efficient memory representation and scalable restore mechanisms are therefore essential for practical checkpoint-based hot starts.

To quantify the snapshot restore pipeline bottleneck, we measure the disk and GPU read/write bandwidths. While storage may be remote, we ignore network bandwidth as it is consistently higher in modern data center networks. Figure 2 measures the throughput of the CUDA checkpoint and restore API in isolation. Checkpoint throughput reaches 3.9 GB/s and restore 7.2 GB/s at 16 GB. This is an order of magnitude below the theoretical PCIe peak of 63 GB/s per direction (Figure 3).

Disk I/O is, however, a more constraining bottleneck. Figure 4 shows sequential read and write throughput on an NVMe SSD (PCIe 4.0) at two block sizes: 4 KB, matching CRIU's default page-granular I/O, and 1 MB, representing batched writes. Writes saturate at approximately 1.5 GB/s regardless of block size, while reads reach up to 2.5 GB/s for 4 KB blocks at 16 GB. These rates are well below the CUDA restore throughput shown in Figure 2, indicating that storage bandwidth, not GPU transfer, is the dominant bottleneck on the restore critical path. Reducing the volume of data written to and read from disk through compression can therefore lower end-to-end checkpoint and restore latency.

Inference Workload Checkpointing. Public serverless platforms manage large-scale multi-tenant deployments hosting a variety of inference engines on different devices. To cope with such heterogeneous environments, operators rely on transparent platform-level checkpointing engines that capture the entire state of the inference engine (including the internal GPU state). As such, for this work, we build on top of CRIUgpu [28], a recent work proposing a fully transparent and unified checkpointing engine that extends CRIU to checkpoint the GPU device state in addition to the host CPU state. Other AI-enabled C/R works [10, 25, 31] offer transparent C/R but rely on complex API interception between application and the GPU driver, forcing platform operators to carefully track how user applications interact with the underlying GPU drivers.

C/R In Userspace [8] (CRIU) is a widely used framework for transparent checkpointing of Linux processes. CRIU freezes a target process, serializes its kernel-managed resources and user-space memory into image files, and later restores the

process from these images. Memory contents are stored page-by-page in `pages.img` files, with a `pagemap` image recording per-page metadata. On restore, CRIU recreates the address space and repopulates memory from the stored page images.

CRIU provides the fundamental technology for checkpointing of many container checkpointing engines, including those used in Kubernetes-based runtimes. Because containers are implemented as Linux processes with namespaces and cgroups, CRIU can be applied to checkpoint entire containers transparently, without requiring application-level modifications. This makes it particularly relevant for serverless ML deployments, where model-serving processes run inside containers or lightweight virtual machines. However, CRIU's default memory handling is page-granular but uncompressed, leading to large `pages.img` files proportional to the memory footprint of the process. For ML workloads with multi-gigabyte model weights, snapshot size directly affects checkpoint latency, storage overhead, and restore time.

3 Challenges with Snapshot Compression

While platform-level checkpointing engines such as CRIUgpu can be used with offline compression, it introduces additional data-copying steps or require intermediate buffers during both checkpoint and restore operations. The overheads of these additional steps can easily negate the latency benefits expected from snapshot-based hot-starts.

Offline Checkpoint Compression. A straightforward approach to reducing checkpoint size is to apply compression after the checkpoint has been written to disk. In this *post-checkpoint compression* pipeline, the runtime first generates a full, uncompressed snapshot (e.g., CRIU image files), flushes it to persistent storage, and subsequently invokes an external compression utility (e.g., `gzip`, `lz4`, or `zstd`) to reduce its footprint. While simple to implement, this approach introduces two main inefficiencies. First, the checkpoint pause time remains dominated by writing the full uncompressed memory image to disk since compression does not overlap with memory dumping, and therefore does not reduce the I/O volume during the latency-critical checkpoint phase. Second, the compression step itself incurs additional CPU overhead and elongates the overall snapshot pipeline, increasing time-to-readiness for subsequent restores or migrations.

Storage Duplication. Offline compression also leads to temporary storage duplication as the uncompressed snapshot must first be fully materialized on disk before compression begins. During compression, both the uncompressed and compressed versions coexist, effectively doubling peak storage requirements. This duplication becomes particularly problematic for large-scale AI workloads. GPU-accelerated training or inference jobs commonly maintain memory footprints ranging from tens to hundreds of gigabytes per replica. In distributed settings, aggregate snapshot size can easily reach terabytes. Provisioning additional memory or disk capacity to accommodate both compressed and uncompressed artifacts increases infrastructure cost and may exceed the limits imposed in containerized environments.

Restore-time Overhead. Restore performance is also affected by post-checkpoint compression. Restoring a compressed snapshot requires decompressing the memory image before repopulating the process address space, temporarily doubling the storage footprint. In containerized environments, this expansion can exceed filesystem quotas or cgroup memory limits. Moreover, the full decompression pass delays application readiness, diminishing the latency benefits of snapshot-based hot starts for inference workloads with strict service-level objectives.

Incremental Checkpoint Deduplication. Incremental checkpointing reduces snapshot overhead by storing only changed pages across checkpoint iterations. However, offline compression applied to the entire snapshot as a monolithic artifact obscures page boundaries, preventing page-granular deduplication. In addition, compressed representations may vary across snapshots even when underlying pages are identical, breaking cross-snapshot deduplication.

In summary, offline compression is fundamentally misaligned with low-latency snapshot pipelines and incremental checkpoint designs. Compression must be integrated at the page level to preserve granularity and compatibility with these mechanisms.

4 CRIU-LZ4

CRIU-LZ4 extends CRIUgpu [28] to provide restore-path optimized, page-granular memory compression that minimizes cold-start latency for containerized inference workloads. As shown in Section 3, offline compression is fundamentally misaligned with low-latency restore pipelines due to storage duplication and decompression overhead. CRIU-LZ4 addresses these limitations by integrating compression directly into the memory checkpoint and restore pipeline. CRIU-LZ4 is implemented as an optional, backward-compatible extension to CRIUgpu where compression is enabled at checkpoint time. During restore, compressed snapshots are detected automatically from the checkpoint metadata and do not require additional configuration. CRIU-LZ4's design aims to reduce total snapshot size, while avoiding intermediate data copies or temporary artifacts, and maintaining compatibility with page-granular optimizations such as incremental checkpointing.

CRIU-LZ4 has four main objectives. First, CRIU-LZ4 preserves page boundaries to ensure that memory compression remains fully compatible with the existing incremental checkpointing and page deduplication mechanisms. Second,

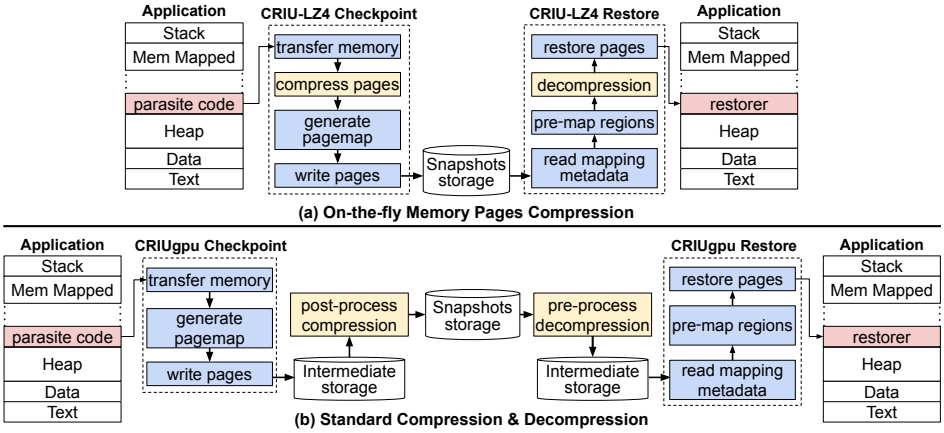

Figure 5: Overview of (a) CRIU-LZ4 and (b) standard file-based compression mechanisms for container checkpointing systems with an intermediate local storage. CRIU-LZ4 eliminates the need for intermediate storage.

to minimize memory overhead and checkpointing latency, CRIU-LZ4 adopts a streaming approach that performs compression and decompression inline within the existing checkpoint and restore data paths. Third, CRIU-LZ4 prioritizes restore-path optimization by minimizing the latency on the restore critical path, i.e., the time from when a container begins restoring to when it is ready to serve inference requests. Finally, CRIU-LZ4 is designed to reuse the existing C/R architecture by minimizing changes into pagemap abstraction, memory processing, the Position-Independent Executable (PIE) parasite code, and restorer context.

4.1 Memory Checkpointing

As illustrated in Figure 5, the checkpointing operation creates a snapshot of the target application address space through PIE *parasite code*. The checkpointing engine uses the `smaps`, `map_files`, and `pagemap` kernel interfaces to generate a list of memory mappings for private, shared, and copy-on-write pages. The checkpointing engine then communicates with the parasite code over a Unix socket, which transfers individual memory pages (4 KB) into a set of PIPEs (*page-pipe buffers*) using the `vmsplice` system call. These pages are then transferred into a set of image files using `splice` system calls. This approach minimizes the memory overhead and checkpointing latency by leveraging zero-copy mechanisms that avoid redundant data copies between user space and kernel space.

4.2 On-the-Fly Memory Pages Compression

CRIU-LZ4 applies lightweight, low-latency LZ4 compression to each memory page as it is streamed from the address space of the target application. Instead of saving raw pages directly to disk, each page is classified into one of three categories: (i) *zero-pages (fast path)* – avoids both compression overhead and I/O by detecting zero-filled pages; (ii) *compressible pages* – stored as LZ4-compressed blocks of data along with metadata that contains compressed size; and (iii) *incompressible pages* – where the LZ4 output is at least as large as the input (e.g., high-entropy optimizer state), so the original page is stored uncompressed to avoid wasting CPU cycles on data that cannot benefit from compression.

Per-page compressed sizes are stored as a repeated 32-bit unsigned integer array within the pagemap protobuf entry, alongside a cumulative compressed size field that records the total. This metadata enables random-access restore, where any page can be located by summing the compressed sizes of the preceding pages, without requiring a separate index.

CRIU-LZ4 saves only the compressed data and its size into the corresponding checkpoint images. As compression is applied per page, no large intermediate buffers are required and page boundaries are preserved. By compressing pages on-the-fly, the amount of data written to disk is reduced proportionally to the achieved compression ratio. In addition, this method allows the compressed snapshots to remain compatible with incremental checkpointing and page deduplication. Compression is applied uniformly to both private and shared memory pages.

4.3 Restoring Compressed Pages

File offsets of compressed pages. While page-level compression preserves alignment with the existing checkpointing engine abstractions and functionality, it also changes how memory page offsets are calculated during the restore phase. In particular, the order in which pages are restored is different from the order in which they are written during

checkpointing. As a result, the restore phase uses the memory mapping entries to calculate file offsets used to locate and read individual memory pages from the snapshot. With raw memory data, offset calculation is a simple multiplication of the page size by the number of pages. With compressed data, however, each compressed page has a different size, and the system must sum the compressed sizes of all preceding pages. CRIU-LZ4 addresses this challenge during the checkpointing operation by keeping track of the compressed sizes and their sum, and storing these values with memory mappings metadata. This approach requires O(1) additional storage per memory mapping entry and avoids sequential scans during restore.

Restorer context. By preserving the compressed sizes of pages and their sums, CRIU-LZ4 can efficiently locate, read, and decompress individual pages. However, the restorer context is responsible for recreating the target process's virtual memory and populating its contents through `preadv()` system calls. This component must remain a minimal position-independent executable that cannot be linked against external compression libraries. To address this requirement, CRIU-LZ4 introduces a lightweight helper process running outside the restorer context that performs read and decompression operations for requested pages (see Figure 6). To preserve process tree semantics and correctly reconstruct the process address space, memory pages are restored in a different order than they were written during checkpointing. Consequently, CRIU-LZ4 locates and decompresses pages independently. In particular, the restorer communicates compressed offsets and page metadata via PIPEs, and the helper writes decompressed pages into the target address space using `process_vm_writev()`. This approach allows CRIU-LZ4 to perform batched I/O reads with inline decompression. This separation ensures that the restorer remains minimal and free of external library dependencies, preserving CRIU's security and correctness guarantees for address space reconstruction.

Restore-path decompression. CRIU supports multiple restore modes depending on how page data is stored and transferred. In the default mode, pages are read from local disk. In addition, CRIU can receive pages over the network from a *page server*, which acts as a remote store for memory images. A third mode uses CRIU *image streamer* [7], where all checkpoint images are transferred over a UNIX socket to an external process without intermediate local storage. CRIU-LZ4 integrates decompression into each of these paths while preserving their distinct I/O characteristics.

In the *local* restore path, pages are read from disk using `pread()`. For compressed images, CRIU-LZ4 introduces a compressed variant of the local reader that resolves the file offset of each page by summing the per-page compressed

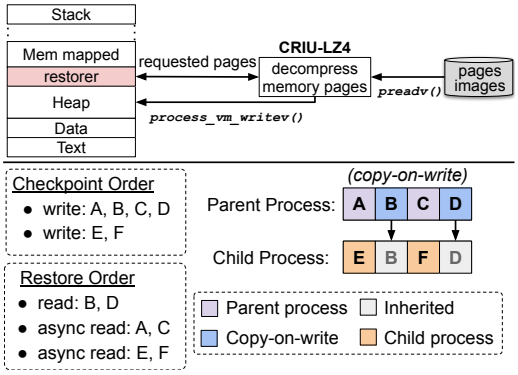

Figure 6: Helper process restoring compressed memory pages for the PIE-compiled restorer context (top), and an illustrative example of saving and restoring memory pages for parent and child processes (bottom).

sizes stored in the pagemap metadata, reads the compressed block, and decompresses it inline. To amortize I/O overhead, CRIU batches multiple page requests into asynchronous read queues. CRIU-LZ4 extends this mechanism by capturing per-page compressed sizes alongside each queued request and capping the batch size to avoid allocating a single decompression buffer proportional to the entire checkpoint. During batch processing, a single `pread()` reads the concatenated compressed blocks, which are then decompressed page-by-page into the target memory regions.

In the *streaming* restore path, pages are read sequentially and cannot be accessed at arbitrary file offsets. The compressed streaming reader consumes the variable-length compressed blocks in order, handling short reads across pipe buffer boundaries, and decompresses each block inline as it is received. For entries inherited from a previous uncompressed checkpoint, both paths transparently fall back to the default uncompressed reader.

In the *page server* path, CRIU transfers memory pages over the network to a remote node for storage. With CRIU-LZ4, the source compresses each page before transmission, reducing the volume of data sent over the network. On the receiving end, the page server receives pre-compressed pages, where each page is preceded by a 32-bit compressed size header, and writes the compressed blocks directly to the local image without re-compressing, preserving the pagemap metadata. The subsequent local restore proceeds through the compressed local reader described above. This design avoids redundant compression on the destination while reducing network transfer time proportionally to the achieved compression ratio. Across all three paths, decompression is performed inline as pages are consumed, ensuring that no fully materialized uncompressed snapshot is required at any point during restore.

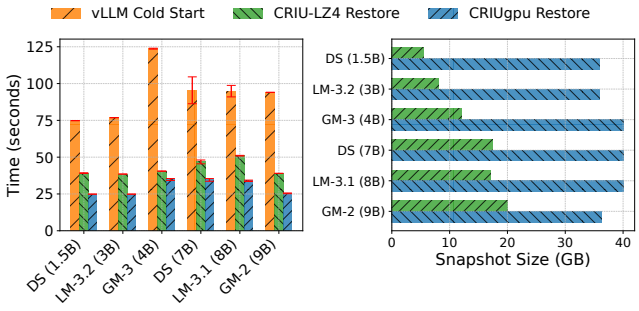

Figure 7: vLLM cold-start latency, CRIU-LZ4 and CRIUgpu restore times (left) and snapshot sizes (right) for vLLM containers serving Gemma (GM), LLaMA (LM), and DeepSeek-R1 (DS) models. The container rootfs and checkpoint image files are stored on a local in-memory filesystem.

5 Evaluation

We evaluate CRIU-LZ4 to answer the following questions:

- How does the cold-start latency engines such as vLLM compare to restoring with CRIU-LZ4? (§ 5.1)
- What are the trade-offs between CRIU-LZ4 and offline compression methods in terms of C/R latency, snapshot size, and storage overhead? (§ 5.2)
- How does CRIU-LZ4 perform on training jobs? (§ 5.3)

Methodology. We evaluate the runtime overhead and storage footprint of CRIU-LZ4's compression mechanism using both process-level and container-level checkpointing, and measure the performance impact when checkpoints are saved to local storage (Figure 7) as well as packaged into archive files (tarballs) (Figure 1, Figure 8). Unless otherwise stated, all reported results are averaged over 3 independent runs.

Experimental Setup. To evaluate CRIU-LZ4, we use three Ubuntu 24.04 servers equipped with NVIDIA GPUs, running Linux kernel v6.8, CUDA 13.1, NVIDIA driver version 590.48, Podman v5.8, and CRIU v4.2. For the LLM inference experiments in Figure 1, we deploy vLLM containers on a server equipped with an NVIDIA H100 SXM5 80 GB (HBM3) GPU, an Intel Xeon Platinum 8480+ CPU (13 cores, 26 threads), 221 GB of ECC memory, and 2.8 TB of local storage. The cold-start and C/R latency experiments shown in Figure 7 and Figure 8 are conducted on a server with an NVIDIA A100 SXM4 40 GB GPU, 30 vCPUs (AMD EPYC 7J13), 216 GB of ECC memory, and 512 GB of SSD storage. For the fine-tuning experiments in Figure 9, we use LLaMA models running on an NVIDIA B200 SXM6 180 GB GPU, an Intel Xeon Platinum 8592 CPU (13 cores, 26 threads), 360 GB of ECC memory, and 2.8 TB of storage.

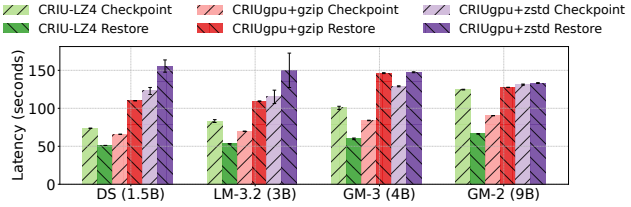

Figure 8: C/R latency of CRIU-LZ4 and CRIUgpu using `gzip` and `zstd` on a vLLM container serving FP16 Gemma (GM), LLaMA (LM), and DeepSeek-R1 (DS).

5.1 Cold Start and Restore Latency

In Figure 7, we first characterize the cold-start latency of vLLM [17], a state-of-the-art inference engine, with several DeepSeek-R1 [9], LLaMA [30], and Gemma [29] models.

Across models from 1.5B to 9B parameters, vLLM cold-start latency ranges from 74.8–123.7 s, with time-to-first-token (TTFT) at 46–58 ms (except Gemma-3 4B at 1.37 s, likely due to its larger context window). In comparison, CRIU-LZ4 completes restore in 39–51 s, reducing startup time by 46–59% while preserving identical runtime state and TTFT. Although CRIUgpu achieves lower restore latency (25–35 s), uncompressed snapshots occupy 36–40 GB. CRIU-LZ4 reduces snapshot sizes to 5.4–20 GB (up to 6× reduction). Smaller models achieve higher compression ratios due to the larger proportion of zero-filled pages (e.g., pre-allocated KV cache buffers) relative to dense model weights. These results demonstrate the practical trade-off of CRIU-LZ4 for latency-sensitive inference deployments where cold starts directly impact tail latency.

5.2 Snapshot Compression Trade-offs

We compare CRIU-LZ4 with two offline snapshot compression methods used in container runtimes: `gzip` and `zstd`, which package checkpoints as compressed tarballs [28].

Figure 8 and Table 1 compare checkpoint latency, restore latency, and storage overhead. The total checkpoint time for CRIU-LZ4 ranges from 73–125 s. Although archive-based methods report lower raw checkpoint times (45–58 s), total latency increases to 65–131 s once tarball creation and compression are included. At restore time, CRIU-LZ4 completes in 51–66 s, whereas archive-based methods require 109–146 s with `gzip` and 133–156 s with `zstd`, dominated by tarball extraction and decompression.

Despite restoring up to 3× faster, CRIU-LZ4 achieves compressed sizes within 5–20% of `gzip` and `zstd`. As shown in Table 1, archive-based methods require 40–55 GB of peak disk space, while CRIU-LZ4 reduces this to 11–40 GB (up to 3–4× reduction). Overall, CRIU-LZ4 combines lower peak storage with the fastest end-to-end restore performance.

Model	CRIU-LZ4	gzip	zstd
DeepSeek-R1-1.5B (3.3 GB)	11	40.6	40.4
LLaMA-3.2-3B (6 GB)	16.2	43.1	43
Gemma-3-4B (8 GB)	14.6	48	50
Gemma-2-9B (17.2 GB)	40	55.2	55.2

Table 1: Peak storage space (GB) for C/R for compressed snapshots using CRIU-LZ4 and CRIUgpu with offline-compression for vLLM containers.

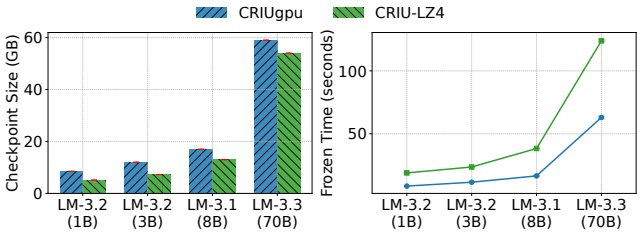

Figure 9: Supervised fine-tuning training of LLaMA models running on the NVIDIA B200 180 GB SXM6.

5.3 Model Training Snapshot Compression

We evaluate CRIU-LZ4 on training workloads to assess its effectiveness for larger memory states. Compared to inference, training snapshots include optimizer states, gradients, and larger activation buffers, increasing memory footprint and checkpoint cost. Training snapshots also exhibit lower compressibility than inference snapshots: optimizer states (e.g., Adam momentum and variance) and gradients contain high-entropy floating-point values that compress poorly at page granularity, whereas inference snapshots contain large zero-filled KV cache allocations that compress well.

Figure 9 shows that CRIU-LZ4 consistently reduces snapshot size across all models, achieving 1.1–1.6× reductions (e.g., from 12 GB to 7.1 GB for LLaMA-3.2 3B, and from 59 GB to 54 GB for LLaMA-3.3 70B). As CRIU-LZ4 compresses inline while the application is frozen, checkpoint freeze time increases by 2–3× depending on model size. This overhead is acceptable for training workloads that checkpoint infrequently, where reduced storage footprint and faster failure recovery outweigh the per-checkpoint cost. Restore latency increases by 1.5–1.7× (e.g., 5.7 s to 8.3 s for 3B, 26.3 s to 41.5 s for 70B), but the smaller snapshots translate to lower storage costs and faster transfers on remote storage.

6 Related Work

Engine-Agnostic Inference Checkpointing. Snapshot-based model swapping and hot starts have been explored across inference engines and serving architectures. Systems such as ServerlessLLM [11] and SwapServeLLM [27] demonstrate that restoring pre-initialized model instances can significantly reduce cold-start latency. As CRIU-LZ4 operates at the platform level, transparently capturing and restoring the full CPU–GPU execution state without modifying the inference framework, it is applicable to heterogeneous serving stacks and future inference engines beyond vLLM. Our evaluation shows that CRIU-LZ4 achieves consistent compression improvements across distinct model families (DeepSeek-R1, Gemma, LLaMA) and multiple GPU platforms (A100, H100), without requiring any model-specific configuration.

Compression-aware Checkpointing. Coordinated Restore at Checkpoint (CRaC) [6] is an OpenJDK feature that provides offline compression with intermediate storage for Java-based applications. Container runtimes such as gVisor [12] provide similar C/R mechanisms with compression; however, they do not support incremental checkpointing or memory deduplication. In addition, parallel GPU restore mechanisms, such as gCROP [32], are complementary to CRIU-LZ4 on-the-fly snapshot compression, as they target GPU memory restore parallelism rather than reducing snapshot size or storage overhead. By contrast, CRIU-LZ4 integrates compression directly into the checkpoint data path while preserving compatibility with incremental snapshots and GPU C/R.

7 Conclusion

We presented CRIU-LZ4, an on-the-fly, restore-optimized compression mechanism integrated directly into the CPU–GPU checkpoint and restore pipelines that eliminates intermediate data copies and minimizes storage overhead and peak disk usage while preserving compatibility with incremental checkpointing. Our evaluation results with inference workloads demonstrate that CRIU-LZ4 reduces snapshot size by up to 6× and avoids the I/O and storage overhead of archive-based compression, achieving the fastest end-to-end restore times among all evaluated methods while reducing cold-start latency by 46–59% relative to reinitializing inference engines from scratch. For training workloads, CRIU-LZ4 reduces snapshot sizes by up to 1.6× with moderate checkpoint overhead, making it suitable for fault-tolerance scenarios where storage efficiency and fast recovery are prioritized over checkpoint speed.

Acknowledgments

We thank Steven Gurfinkel for his insightful feedback on the CUDA checkpointing functionality. This work was supported in part by FCT under projects UID/PRR/50021/2025, LISBOA2030-FEDER-00748300, UID/50021/2025, and by the EU's Horizon Europe research and innovation programme under Grant Agreement No. 101189689.

References

[1] BentoML. 2026. BentoML. BentoML Documentation. https://www.bentoml.com/ Accessed: 2026-02-16.

[2] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, et al. 2022. On the Opportunities and Risks of Foundation Models. arXiv:2108.07258 [cs.LG] https://arxiv.org/abs/2108.07258

[3] Luis Capelo and Colin Weld. 2025. GPU Memory Snapshots: Supercharging Sub-second Startup. Modal Blog. https://modal.com/blog/gpu-mem-snapshots Accessed: 2026-02-16.

[4] Cedana. 2026. Cedana. Cedana Documentation. https://cedana.com/ Accessed: 2026-02-16.

[5] Cedana. 2026. Cedana Checkpoint/restore basics. Cedana Documentation. https://docs.cedana.ai/daemon/checkpoint-restore/cr Accessed: 2026-02-16.

[6] CRaC Project. 2026. Coordinated Restore at Checkpoint. https://crac.github.io/. Accessed: 2026-02-25.

[7] CRIU. 2020. Image Streamer. GitHub repository. https://github.com/checkpoint-restore/criu-image-streamer Accessed: 2026-04-07.

[8] CRIU Project. 2026. Checkpoint/Restore In Userspace. https://criu.org/. Accessed: 2026-02-24.

[9] DeepSeek-AI, Aixin Liu, Bei Feng, et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[10] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. 2022. Cricket: A virtualization layer for distributed execution of CUDA applications with checkpoint/restart support. *Concurrency and Computation: Practice and Experience* 34, 14 (2022), e6474. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6474 doi:10.1002/cpe.6474

[11] 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

[12] Google. 2018. gVisor: Application Kernel for Containers. https://github.com/google/gvisor.

[13] Google. 2026. Vertex AI. Vertex AI Documentation. https://cloud.google.com/vertex-ai Accessed: 2026-02-16.

[14] Google Cloud. 2026. GKE Pod snapshots. Google Kubernetes Engine. https://docs.cloud.google.com/kubernetes-engine/docs/concepts/pod-snapshots Accessed: 2026-02-16.

[15] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[16] Ekin Karabulut and Yoed Ginzburg. 2025. *Cut Model Deployment Costs While Keeping Performance With GPU Memory Swap*. https://developer.nvidia.com/blog/cut-model-deployment-costs-while-keeping-performance-with-gpu-memory-swap/ NVIDIA Developer Blog.

[17] 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* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[18] Modal. 2026. Modal. Modal Documentation. https://modal.com/ Accessed: 2026-02-16.

[19] NVIDIA. 2026. Dynamo Checkpointing for Kubernetes. NVIDIA Dynamo Documentation. https://docs.nvidia.com/dynamo/dev/kubernetes-deployment/deployment-guide/checkpointing/integration-with-dynamo Accessed: 2026-02-16.

[20] NVIDIA. 2026. NVIDIA Dynamo. NVIDIA Dynamo Documentation. https://developer.nvidia.com/dynamo Accessed: 2026-02-16.

[21] Adrian Reber. 2019. Add support to migrate containers. https://github.com/containers/podman/pull/2272

[22] Adrian Reber. 2022. Provide support for Checkpoint and Restore. https://github.com/cri-o/cri-o/pull/4199

[23] Adrian Reber. 2024. Wire through CRI checkpoint RPC. https://github.com/containerd/containerd/pull/6965

[24] Brandon Royal. 2025. Introducing Agent Sandbox: Strong guardrails for agentic AI on Kubernetes and GKE. Google Cloud Blog. https://cloud.google.com/blog/products/containers-kubernetes/agentic-ai-on-kubernetes-and-gke Accessed: 2026-02-16.

[25] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. 2022. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. arXiv:2202.07848 [cs.DC] https://arxiv.org/abs/2202.07848

[26] Snowflake. 2024. Snowflake LLM Inference: Model Hotswapping. https://www.snowflake.com/engineering-blog/llm-interference-model-hotswapping/

[27] Radostin Stoyanov, Viktória Spišaková, Adrian Reber, Wesley Armour, Marcin Copik, and Rodrigo Bruno. 2025. Engine-Agnostic Model HotSwapping for Cost-Effective LLM Inference. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*. Association for Computing Machinery, New York, NY, USA, 114–125. doi:10.1145/3731599.3767354

[28] Radostin Stoyanov, Viktória Spišaková, Jesus Ramos, Steven Gurfinkel, Andrei Vagin, Adrian Reber, Wesley Armour, and Rodrigo Bruno. 2025. CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads. arXiv:2502.16631 [cs.DC] https://arxiv.org/abs/2502.16631

[29] Gemma Team, Morgane Riviere, Shreya Pathak, et al. 2024. Gemma 2: Improving Open Language Models at a Practical Size. arXiv:2408.00118 [cs.CL] https://arxiv.org/abs/2408.00118

[30] Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[31] Xingda Wei, Zhuobin Huang, Tianle Sun, Yingyi Hao, Rong Chen, Mingcong Han, Jinyu Gu, and Haibo Chen. 2025. PhoenixOS: Concurrent OS-level GPU Checkpoint and Restore with Validated Speculation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 996–1013. doi:10.1145/3731569.3764813

[32] Yanning Yang, Dong Du, Haitao Song, and Yubin Xia. 2024. On-demand and Parallel Checkpoint/Restore for GPU Applications. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 415–433. doi:10.1145/3698038.3698510