FLASH: Fault-Led, Attested, Secure Hybrid-Key Checkpoint for Confidential Inference

Jonghyun Lee[1], Taewoon Kang[2], Yongqin Wang[3], Rachit Rajat[4], Mengyuan Li[1], Gunjae Koo[2], Daniel Wong[5], Murali Annavaram[1]
{leejongh,mli49061,annavara}@usc.edu,{taewoon_kang,gunjaekoo}@korea.ac.kr
yongqinwang@roblox.com,rrajat@nvidia.com,danwong@ucr.edu
[1]University of Southern California, [2]Korea University, [3]Roblox, [4]NVIDIA, [5]University of California, Riverside

Abstract

GPU Confidential Computing (CC) enables secure LLM inference in production, combining a CPU TEE with NVIDIA's GPU TEE to protect models and user data from a malicious cloud operator. As long-running LLMs are failure-prone, traditional cloud systems rely on backups, and live migration using checkpoint and restore to deal with failures. In particular, during LLM serving, GPUs build up a massive key-value (KV) cache that accelerates the decode process. But when using GPU CC, these recovery methods fail, as the GPU's memory state, such as the KV cache, can't be efficiently checkpointed. Checkpointing GPU memory requires encrypting the GPU data and transmitting to the CPU over the PCIe, which the CPU then decrypts using its shared key with the GPU. This encryption and decryption sequence over the PCIe leads to a 5.5× bandwidth penalty. The decrypted data is then re-encrypted with a new set of keys shared by the CPU and a destination server which adds overhead to the already-long checkpoint latency. LLM serving is stalled for the entire duration of checkpointing; a secure full-dump of an 80 GB H100 memory takes roughly 11 s. Alternatively, tracking write operations and selectively checkpointing these written pages is a preferable path as only a small fraction of the KV cache is updated after each decode round. Current GPUs, however, lack the ability to track dirty pages.

We present FLASH, a checkpoint/recovery (C/R) system for GPU CC that is both secure (providing confidentiality, integrity, and freshness against a malicious operator) and low latency. FLASH eliminates needless decryption of GPU state during checkpoint creation. FLASH preserves the GPU-encrypted HBM data as is without decryption. The CPU then creates a heterogeneous encryption package: (1) the CPU's own memory state and the key used to encrypt the GPU CC is encrypted into a single bundle using keys shared by the CPU and a destination server that stores the checkpoint. (2) The GPU-encrypted HBM data is preserved as a second bundle within this heterogeneous package. This approach eliminates redundant decrypt-then-reencrypt overheads for GPU CC state. FLASH also presents a novel approach to track dirty pages to enable incremental checkpointing. FLASH enhances the GPU's unified virtual memory (UVM) driver to downgrade any GPU page that is already checkpointed to a read-only page. It then uses the write fault mechanism to track modified pages at a 2 MB granularity. This approach requires no source code modifications to the LLM code base. Lastly, FLASH classifies never-written allocations, such as model weights, and removes them from repeated checkpoint copies while continuing to track them for correctness. On an H100 with Intel TDX running vLLM, FLASH reduces checkpoint stall latency up to 9.4× over state-of-the-art GPU C/R systems.

ACM Reference Format:
Jonghyun Lee[1], Taewoon Kang[2], Yongqin Wang[3], Rachit Rajat[4], Mengyuan Li[1], Gunjae Koo[2], Daniel Wong[5], Murali Annavaram[1]. 2026. FLASH: Fault-Led, Attested, Secure Hybrid-Key Checkpoint for Confidential Inference. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857076

1 Introduction

GPU Confidential Computing (CC) is increasingly adopted for AI inference workloads that handle sensitive models and data on untrusted cloud infrastructure. By combining a CPU Trusted Execution Environment (TEE), such as Intel TDX [19] or AMD SEV-SNP [40], with GPU TEE, such as NVIDIA Confidential Computing [13, 30] available on Hopper (H100) [33] and Blackwell (B200, RTX PRO6000) [2, 4] architectures, an end-to-end confidential stack protects both the host memory and GPU memory from a malicious hypervisor. This confidential CPU+GPU stack is now commercially available on major cloud platforms, such as Microsoft Azure and Google Cloud [15, 29]. Meta's Private Processing system uses NVIDIA CC to run LLM inference for WhatsApp AI features with hardware-enforced guarantees that neither Meta nor the infrastructure provider can access user data in the clear [28]. As confidential LLM inference moves from



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGOPS ATC 2026, Hong Kong, Hong Kong
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857076

early adoption to production serving, the question of how to operate these deployments reliably becomes urgent.

Reliability is a major hurdle for LLM deployments. LLM failure probability scales with deployment duration [11, 16, 31, 39]. CC deployments may worsen, or at best preserve,these failure rates as they add a cryptographic layer on top of the GPU and CPU runtime stacks. Checkpoint/restore (C/R) is a time tested approach for recovering quickly from a failure. C/R allows the system state to be saved at regular intervals and restored whenever a failure is encountered. In the context of LLM serving, key-value (KV) cache is a GPU-generated data structure that grows rapidly with context length. Without checkpoint/restore (C/R), any fault results in a full cold restart. Cold restart forces every request in flight to be dropped, discarding tokens generated for the current round, removing the KV cache corresponding to the currently evaluated context. This cost grows with context length as agentic workloads accumulate long-lived sessions with massive KV cache contexts. Recomputing them from the retained prompt repeats the full prefill process. Even with the repetition of prefill, the restart fails to reproduce the original session, as tokens are sampled in a stochastic manner during decode. Hence a resumed request diverges from the original path. C/R is therefore critical for fault tolerance.

C/R Challenges with CC: GPU CC makes C/R categorically harder than in conventional deployments. In a non-CC setting, the hypervisor can inspect GPU memory, snapshot process state, or migrate the workload to a healthy node. In CC mode, these mechanisms are unavailable by design, as the same isolation that prevents a malicious operator from reading the tenant's model weights equally prevents the operator from saving GPU state during a fault [13, 17, 30]. Cloud providers acknowledge this gap explicitly, as Azure and Google Cloud disclaim backup, site recovery, and live migration for confidential VMs with GPU CC, instructing customers to build their own high-availability solutions [15, 29].

The main obstacle to practical checkpointing for confidential computing is the CPU-GPU PCIe bandwidth. In existing GPU CC threat models, the PCIe, connecting CPU TEE and GPU TEE is untrusted. Hence, any GPU state that is checkpointed and transferred to the host must be encrypted first. This encryption is on the critical path and reduces the PCIe bandwidth by 5.5× – reducing the bandwidth to 7.3 GB/s from the default 40 GB/s (Section 4). Hence, stunning the GPU and dumping its memory, also referred to as *stop-the-world* checkpoint, leads to a 11 sec stun time for an 80 GB HBM, an order of magnitude latency penalty than most inference systems can tolerate.

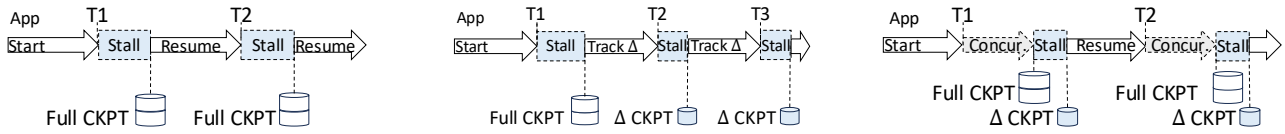
Furthermore, any checkpoint image needs to be stored in a remote server for availability reasons. Hence, the CPU must first decrypt the GPU memory state and append the DRAM state to create a complete system-wide checkpoint. This system-wide checkpoint is then re-encrypted with a new key that is shared between the source and a remote server. Thus, there is a full re-encryption cost on top of the already prohibitive cost of transferring GPU memory over the PCIe channel. Copying the entire system memory footprint (CPU and GPU memory state) on every checkpoint is therefore prohibitively expensive.

Challenges with Incremental Checkpointing in CC: Two mechanisms avoid the full system checkpointing overhead by tracking memory that has been dirtied and taking only a snapshot of the dirty pages. (1) *Incremental checkpoint* copies the dirty delta accumulated since the previous checkpoint. (2) *Concurrent checkpoint*, on the other hand, overlaps the GPU memory state checkpointing and copying to the host while continuing to service the LLM requests and then re-copies only the pages dirtied at the end of the main checkpoint copy window.

To overcome this limitation, prior works relied on software-based tracking of dirty data. Buffer-granularity tracking [46] conservatively marks an entire allocation (buffer) dirty whenever a kernel receives a mutable pointer into it. A buffer here could be a programmer defined data structure. vLLM's KV cache kernels typically allocate large buffers yet write only a few entries per call. Conservative dirty page marking would need to checkpoint the entire KV cache memory buffer on each checkpoint interval due to this coarse-grain tracking. Our measurements show that such an approach inflates the effective write rate to ~2.8 GB/s. An alternative approach is to use byte granularity [48] that captures the actual 1 GB/s dirty data rate at a fine granularity. But this approach requires offline PTX analysis that is tied to specific kernel implementations to track which bytes are modified. The analysis must be re-derived whenever a framework updates its kernel, and cannot handle the runtime-compilation that inference stacks increasingly adopt.

We need a checkpoint system that is both fine-grained and robust to arbitrary, evolving kernels. Additionally, we must consider the case of malicious hosts and thus, provide protection for the created checkpoint image. We present FLASH, an efficient checkpoint/restore system designed for LLM inference on GPU CC systems. FLASH is composed of the following key contributions.

- We show that GPU CC's 5.5× PCIe bandwidth penalty makes checkpointing a significant bottleneck to the reliable deployment of LLMs.
- We propose *hybrid-key checkpoint format*, which eliminates the redundant decrypt-and-then-re-encrypt cycle between GPU and CPU. GPU data delivered to the CPU is already encrypted under the Secure Protocols and Data Model (SPDM) session key (D2H). FLASH retains this ciphertext directly in the checkpoint image rather than decrypting it on the CPU and re-encrypting under a single checkpoint key. This approach requires 2 different keys to be stored alongside the checkpoint, which is a small penalty to pay for removing the redundant decrypt-encrypt cycle.



(a) Stop-the-world checkpoint. **(b)** Incremental checkpoint. **(c)** Concurrent checkpoint.

Figure 1. Various checkpoint methods.

- We adopt concurrent checkpoint with page-granularity dirty tracking through *UVM write-fault handling mechanism*. We repurpose the UVM driver to exploit the driver's page-permission mechanism to track GPU page modifications. We minimize the fault handling overhead with (1) 2MB-granularity tracking and (2) proactive faulting. The optimization is tailored to KV cache usage, where it is preallocated as a contiguous memory pool and a request is assigned with sequential blocks. Our method requires no kernel source code modifications or offline analysis.
- FLASH proposes a *static allocation classification* that identifies memory that is never written (such as model weights) across checkpoint intervals and drops it from subsequent checkpoints, with no framework integration or knowledge of memory layout.
- We implement FLASH on NVIDIA H100 with Intel TDX and evaluate it on vLLM inference, reducing stop-and-copy stall latency to 1.2 s on average; 9.4× over PhoenixOS and 7.8× over GCR, with bounded serving impact.

2 Background

2.1 GPU Checkpoint/Restore (C/R)

Checkpoint/restore captures the execution state of a running process as a checkpoint image and later restores it to resume execution. A GPU checkpoint typically consists of both **data state** (*e.g.* CPU memory and GPU buffers) and **control state** (*e.g.* program counter and execution contexts); capturing the latter relies on driver support such as cuda-checkpoint [34], which is unavailable under CC. FLASH focuses on the data state that governs inference recovery and reconstructs the execution context by re-initializing the serving framework at restore. This suffices for inference, where the costly, irrecoverable state is the KV cache and its indexing metadata rather than the execution context that can be rebuilt. The checkpoint path itself requires no application or kernel source changes beyond interposing on device allocation/synchronization calls.

Extending C/R to GPUs is challenging: GPUs expose no OS-visible dirty bits or copy-on-write support, so memory updates cannot be tracked cheaply. Conventional GPU checkpointing typically follows a *stop-the-world* model (Figure 1a) [21, 34, 41, 43] where CPU and GPU execution are fully quiesced, then the entire memory state is copied to the host, incurring stall time proportional to the full GPU footprint and the PCIe bandwidth.

Two methodologies reduce checkpointing stall times. Incremental checkpoint (Figure 1b) transfers only memory modified since the previous checkpoint, requiring persistent dirty tracking across intervals. For example, GCR [48] supports incremental checkpointing via shadow execution of kernel PTX. Concurrent checkpointing (Figure 1c) overlaps the copy with execution (concurrent copy) and quiesces only briefly to recopy the memory written during the concurrent copy window, the dirty delta (stop-and-copy). As this dirty delta is far smaller than the full footprint, PhoenixOS [46] adopts concurrent checkpointing using copy-on-write and recopy protocols to minimize the stall time.

2.2 GPU Confidential Computing.

A Trusted Execution Environment (TEE) protects code and data from privileged software (*e.g.* a malicious host/hypervisor) and physical attackers via memory encryption and hardware-enforced enclave isolation [7, 9, 19, 22–24, 40]. Confidential Virtual Machines (CVMs) extend protection to the entire guest OS and its applications [8, 9, 19, 23, 24, 40], supporting TBs of memory for data-intensive workloads. Our implementation uses Intel Trust Domain Extensions (TDX) [19] as the CPU TEE, which encrypts CVM memory via Intel Total Memory Encryption (TME) [18] and hardware-rooted attestation. NVIDIA Confidential Computing (CC) [3, 35] extends the trust boundary to the GPU on Hopper (H100) [33] and Blackwell (B200, RTX Pro 6000) [2, 4], adding device-rooted attestation [36] and isolating GPU memory behind a Compute Protected Region (CPR) firewall [17]. Together, they form an end-to-end confidential stack that protects ML and HPC workloads from a malicious hypervisor.

To isolate GPU workloads from the hypervisor, NVIDIA CC pairs the PCIe-side memory firewall with end-to-end encrypted CPU-GPU channel, bootstrapped via Secure Protocols and Data Model (SPDM) [17, 35] attestation at GPU initialization. The GPU and the in-CVM-memory driver then derive directional asymmetric keys - K_{H2D} (host-to-device) and K_{D2H}(device-to-host) - and encrypts any PCIe transfer with Advanced Encryption Standard in Galois/Counter Mode (AES-GCM). The GPU's Copy Engine (CE) performs the device side encryption inside the CPR before data leaves the GPU. Since neither side can access the other's private memory, data transfers route through unprotected bounce buffers.

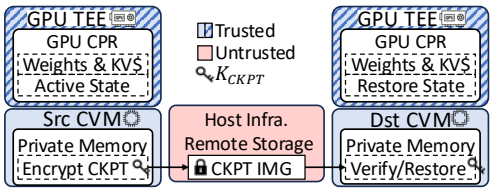


Figure 2. Confidential GPU C/R Architecture and cryptographic boundaries.

Table 1. Copy Engine MemCopy throughput on H100 PCIe across Legacy and TDX (TD) VMs.

	H2D (GB/s)		D2H (GB/s)	
Platform	**Sync**	**Async**	**Sync**	**Async**
Legacy	40.88	40.88	40.81	40.81
TD (CC enabled)	7.31	7.26	7.37	7.29
Reduction	**5.59×**	**5.63×**	**5.54×**	**5.60×**

The source *pushes* ciphertext in, while the destination *pulls* it via PCIe DMA and decrypts into its own protected memory, leaving the hypervisor unable to read or alter the data in transit over untrusted PCIe.

3 Threat Model and Baseline C/R Model

Threat model. Our threat model follows the standard TEE model for multi-tenant clouds [1, 17]. We assume the cloud provider, hypervisor and host OS are malicious. The adversary has physical or remote access to the host machine which includes the system memory and the untrusted CPU-GPU PCIe link. We assume the TEE hardware protections of Section 2.2 hold, which include the CPR firewall, CE-encrypted transfers, SPDM and TME, and exclude physical decapsulation attacks against HBM or the GPU package.

These protections cover data in use, but not a checkpoint image once it leaves the CVM. In transit to remote storage or destination CVM, the image crosses host-controlled infrastructure the adversary fully controls, exposing it to tampering and rollback (replay) attacks. Established TEE state-continuity techniques [10, 27, 37, 42] addresses by using authenticated encryption for tampering and monotonic freshness counter for rollback; we inherit them as our baseline. FLASH targets confidentiality, integrity and freshness; availability is out of scope. A malicious cloud operator controlling the infrastructure can always deny service, declining to schedule the CVM, dropping PCIe traffic, or withholding the RAS. Thus, we do not attempt to guarantee liveness. When required state is unavailable, checkpointing fails and closes rather than proceeding.

Baseline confidential GPU C/R model. To satisfy these security guarantees, we assume a stop-the-world baseline C/R design (Figure 2). It fully pauses execution, utilizes standard CUDA memory copy APIs (*e.g.* `cudaMemcpy`) to dump GPU HBM into CVM private memory (which automatically decrypts the PCIe-side ciphertext), and subsequently uses a software library (*e.g.* OpenSSL) to re-encrypt the entire dump under a dedicated checkpoint key (K_{CKPT}) before writing it to host storage. K_{CKPT} is deposited to a *Remote Attestation Service* (RAS), a separate, third-party trusted CVM outside the adversary's control. RAS retains K_{CKPT} in plaintext in its private memory. At restore time, the destination and RAS perform mutual attestation, establish a secure channel, and RAS sends K_{CKPT} through this channel.

While encryption and authentication via K_{CKPT} detect tampering, the baseline model is still vulnerable to a rollback attack. A stale image that is validly encrypted and carries the correct authentication tag can be restored. The baseline maintains a monotonic counter ctr in CVM private memory, incremented at each checkpoint, and binds it into a report:

$$\text{Report} = ctr \parallel H(\mathcal{M}) \parallel H(CVM_{id}) \parallel ts, \tag{1}$$

where $H(\cdot)$ denotes SHA-256, $\mathcal{M}$ the checkpoint manifest, CVM_{id} the CVM identity, and tsthe checkpoint timestamp. The report is signed by the TDX HW Root of Trust (RoT) inside the CVM, so neither the hypervisor nor any privileged software. The RAS verifies the signature and stores the latest $CVM_{id} \rightarrow \{ctr, H(\mathcal{M})\}$ mapping, which restore side checks against the image it is given. Any image whose counter or manifest hash does not match the stored entry is rejected.

Unless otherwise stated, this paper considers C/R at per-GPU granularity: each checkpoint instance captures and restores the state of a single confidential GPU, while coordinating a globally consistent checkpoint across multiple GPUs is outside the scope of this work.

4 Challenges in Confidential GPU C/R

Checkpoint with GPU TEE faces a critical system bottleneck. Creating a checkpoint image requires transferring the entire GPU memory state across encrypted PCIe, resulting in a structural bandwidth penalty for GPU TEE that makes GPU memory dump prohibitively expensive.

4.1 Bandwidth Penalty in GPU TEE.

The TEE encrypted transfer path imposes a significant bandwidth penalty over standard PCIe data movement. Table 1 depicts the copy engine (CE) host-to-device (H2D) and device-to-host (D2H) transfer throughput across Legacy and TD VM configurations on H100, sweeping transfer sizes from 20 GB to 50 GB in 10 GB increments. Regardless of transfer API, GPU TEE imposes a consistent ~5.5× bandwidth reduction on both H2D and D2H transfers, confirming this is a structural property of TEE encrypted transfer path rather than a software overhead of the benchmark [17, 47]. A CVM-to-GPU-TEE transfer can be decomposed into four components:

(1) CPU must use AES-GCM to encrypt the data (hardware accelerated on both ends with GPU CE and CPU AES-NI). (2) Once the data is encrypted, it needs to be placed in a bounce buffer and read by the GPU driver, which we refer to as a push-pull sequence. (3) The encrypted data must be transferred over PCIe. (4) GPU decrypts the data to be placed in HBM. As TEE isolates its private memory, any outside entity, even another TEE cannot directly access (DMA) each other's memory. The bounce buffer at Step (2), located in unprotected memory region, plays the role of man-in-the-middle allowing the sender to push the data and the receiver to pull the data. The bandwidth constraint is a structural limitation and cannot be eliminated without compromising GPU TEE's security guarantees (Section 8). In our system setting, Step (1) accounts for 49.6% of the transfer time, Step (2) accounts for 30.6%, Step (3) and (4) account for 19.8%. Roughly 80.2% of the cost is on the host side, which we need to avoid in order to achieve practical C/R. Table 1 measures the bandwidth of a single large contiguous copy, 7.3 GB/s, which is a *best* case scenario. Smaller, fragmented transfers issue more calls and realize lower effective bandwidth (Section 7.2). If we were to save GPU state via a stop-the-world checkpoint, an 80 GB H100 using 7.3 GB/s bandwidth stalls serving for 10.9 s. Additionally, with our baseline confidential GPU C/R model (Section 3), K_{CKPT}-encryption must be added. Our simple analysis rules out the stop-the-world checkpoint and motivates the need to minimize the dump footprint by utilizing dirty memory tracking.

Both incremental and concurrent checkpoint provide smaller footprint via dirty tracking on write operations. However, their tracking mechanism makes one more favorable to the other in LLM inference setting. Incremental checkpoint tracks write operations between every checkpoint interval after the first initial full checkpoint. Intuitively, even in low dirty data rate, incremental checkpoint requires to keep smaller interval (*e.g.* ≤1 m) to keep the accumulated dirty delta bounded. Otherwise, a longer interval produces a dirty delta approaching the entire memory. This contradicts with LLM deployment manager who wants reliability through C/R but minimize its impact (stall time) on user experience. In contrast, concurrent checkpoint's dirty delta is proportional only to dirty data rate and transfer bandwidth not the checkpoint interval. Thus, concurrent checkpoint is more favorable in LLM serving scenarios where checkpoint intervals can be manually set by the deployment manager.

However, as GPU inherently does not provide hardware dirty bit per page, prior GPU C/R works [46, 48] rely on software analysis to track writes at buffer granularity (PhoenixOS (PhOS) [46]) or byte-granularity (GCR [48]). We quantify the dirty data rate of each granularity in realistic LLM serving.

4.2 Dirty Data Rate of LLM Serving

We evaluate on two workloads with high memory load: *Chatbot* (high request per second (RPS), small context requests) and *Reasoning* (low RPS, long context requests). See Table 2 for workload-specific configurations. Figure 3 reports the dirty data rate (GB/s) for different tracking granularities: byte, page, and buffer. We modify NVIDIA Unified Virtual Memory (UVM) driver to track writes in 2MB pages and add fine-grained page-granularity tracking (Section 5.3).

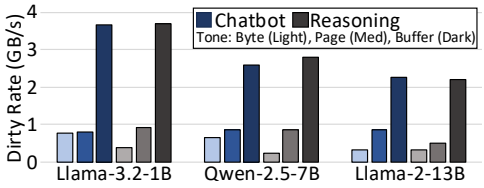


Figure 3. Dirty data rate measured during LLM inference.

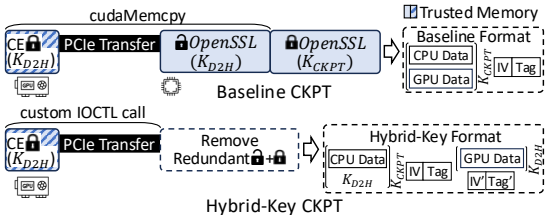


Figure 4. Baseline checkpoint vs hybrid-key checkpoint.

Buffer-granular tracking records an average of 2.8 GB/s write activity, while both byte- and page-granular trackings stay ≤ 1 GB/s dirty rate. However, buffer granularity defeats the purpose of concurrent checkpoint: even at the best-case PCIe bandwidth (7.3 GB/s), a PhOS-style concurrent copy dirties ~30 GB during its 10.9 s window, leaving a best-case of 4.2 s stop-and-copy stall. Accounting for fragmentation in KV cache allocation, the stall can be pushed higher (Section 7.2). Between the fine-grained options, GCR's byte tracking is ~1.9× finer than our 2MB-page tracking, but it relies on offline PTX analysis and falls back to buffer granularity whenever that analysis breaks, requiring updated kernels, runtime compilation, or indirection-dependent writes. Also, GCR takes an incremental checkpoint approach which, as discussed before, is not suitable for LLM serving.

4.3 Towards TEE-aware GPU Checkpoint

Together, these insights demonstrate that while the TEE encrypted path heavily penalizes bulk data transfers (5.5×), the dirty data rate of LLM serving ($\leq 1GB/s$) ensures that the final stop-and-copy phase can be restricted strictly to a minimal dirty delta rather than a costly full memory dump. Resolving this asymmetry requires two optimizations: (1) increasing the bandwidth of the dump path, and (2) tracking dirty memory in a finer granularity at the driver boundary rather than relying on brittle API-level analysis. FLASH realizes both, as detailed in Section 5.

5 FLASH: Fault-Led, Attested, Secure, Hybrid-Key Checkpoint for Conf. GPUs

5.1 System Overview

Building on the baseline of Section 3, FLASH addresses the performance challenges of Section 4 with three optimizations. First, hybrid-key checkpoint (Section 5.2) eliminates the redundant decrypt-then-reencrypt cycle by retaining the CE-delivered K_{D2H} ciphertext directly in the image. Second, UVM write-fault dirty tracking (Section 5.3) operates below the CUDA API, handles all kernel write patterns uniformly, and limits the stop-and-copy phase to actual dirty delta. Finally, static allocation classification (Section 5.4) identifies pages never written after loading, *i.e.* the model weights, captures them once, and excludes them from subsequent concurrent-copy phases.

5.2 Hybrid-Key Checkpoint

As explained in Section 2.2, any data leaving GPU HBM via the CE DMA path is first encrypted inside the GPU CPR before crossing the untrusted PCIe where it arrives in the CPU-side bounce buffer as K_{D2H}-ciphertext. The baseline then decrypts this into CVM private memory as plaintext and immediately re-encrypts it under K_{CKPT}, requiring a full, redundant AES-GCM cycle on the critical path for every GPU page transferred.

FLASH's Hybrid-Key Checkpoint (Figure 4) eliminates this redundant cycle by retaining the K_{D2H}-ciphertext directly in the checkpoint image without decrypting it. A custom IOCTL call in the NVIDIA driver reads the bounce buffer to retrieve the HBM ciphertext together with its AES-GCM metadata, such as IV counters and authentication tags, ensuring that all GPU pages can be unwrapped upon restore. CPU-resident state (process state, and other non-GPU buffers) continues to be encrypted under K_{CKPT} as in the baseline, since it never traverses the CE DMA path.

Asynchronous Producer-Consumer Architecture. Removing the decrypt-then-reencrypt cycle shifts the bottleneck from `cudaMemcpy` to the storage subsystem. The baseline's crypto-throttled producer can be handled by a single-threaded writer, but the accelerated IOCTL path now outruns a sequential writer and results in back-pressure of the PCIe data-path. FLASH therefore decouples the two stages into a producer-consumer pipeline where (1) a high-priority producer thread issues the custom IOCTLs to retrieve K_{D2H}-ciphertext and metadata from the bounce buffer into a shared ring buffer, and (2) a pool of worker threads serializes and streams the image to disk, so storage I/O never stalls the hardware data-path.

Hybrid-Key Checkpoint Format. Hybrid-Key checkpointing result is a hybrid image (Figure 4) with two distinct protection domains: CPU-side data under K_{CKPT} and GPU-side data under K_{D2H}. We later show how this structure preserves confidentiality, integrity, and freshness in Section 6.2.

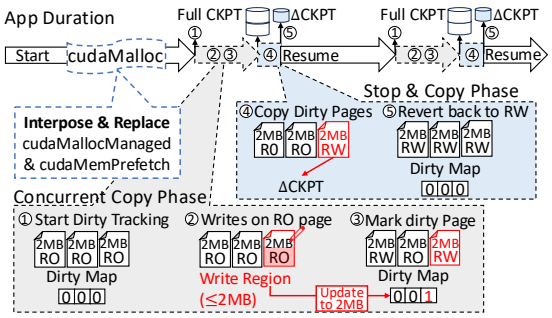


Figure 5. UVM-based page-granular dirty tracking.

To make the GPU domain restorable, the image carries metadata sufficient to reconstruct the K_{D2H} decryption context, such as the AES-GCM authentication tag, the per-channel IV counters at the time of the checkpoint, and the source CVM's K_{D2H} wrapped under K_{CKPT} [1]. At restore, the destination CVM retrieves K_{CKPT} from the RAS CVM (Section 3), unwraps the source CVM's K_{D2H}, and manually AES-GCM-decrypts the GPU pages with the recovered key and the stored IV counters. The CE then re-encrypts them under the destination's session K_{H2D} as they re-enter the GPU. The retained-ciphertext optimization, thus eliminates the cycle on the *frequent* checkpoint path, which runs on the serving hot path. In our hybrid-key checkpoint, restore pays the cost once per fault and it occurs off the critical path.

***K_{D2H}* IV Recording.** Unlike K_{CKPT}, whose IV is managed explicitly by FLASH, the K_{D2H} IV counters are maintained and incremented autonomously by the CE hardware after every encryption operation within the SPDM session. Because the GPU pages are transferred across both the pre-copy and stop-and-copy phases while the IV counter advances continuously, FLASH records the IV counter value associated with each transferred page alongside the page data in the hybrid checkpoint metadata. At restore time, the restore agent uses the recorded IV counters along with the recovered K_{D2H} to reconstruct the exact AES-GCM decryption context for each CE channel, performing manual decryption of GPU pages using the recovered K_{D2H}.

5.3 Fine-Grained Dirty Tracking via UVM

The hybrid-key checkpoint reduces the per-byte cryptographic overhead, but the stop-and-copy stall time remains proportional to the dirty data delta transferred. Therefore, dirty-tracking granularity becomes the dominant factor in application pause time. FLASH revisits NVIDIA Unified Virtual Memory (UVM) write-fault mechanism (Figure 5) to achieve dirty tracking in page granularity. By initially marking GPU pages read-only, it allows the concurrent-copy phase to record exactly which pages are written during the

[1] GPU Travelling [49] demonstrates K_{D2H} export from the session's cryptographic context

tracking phase and limits the stop-and-copy phase to only those dirtied pages. Unlike application-level serialization such as `torch.save`, FLASH requires no framework integration or knowledge of memory layout.

Enabling Write-Fault through UVM. UVM manages GPU memory through a fault-driven model. When a kernel touches a page not resident on the device, the GPU Memory Management Unit (MMU) raises a fault and the driver migrates the page on demand. Similarly, page permission control over already-resident pages is an overlooked mechanism in prior GPU checkpointing works due to its fault-induced slowdown. A GPU-resident page can be marked read-only (RO) where any subsequent CUDA kernel write raises a fault that the UVM driver intercepts before the store commits. We repurpose this write-fault path as a tracking primitive.

Enabling UVM by API Interposition. Frameworks like PyTorch [38] allocate device memory with `cudaMalloc` (Copy-on-Execute (CoE) model) by default, outside UVM's control. To bring all device allocations under UVM's permission control, FLASH interposes on allocation calls by substituting `cudaMallocManaged` for `cudaMalloc` [45, 50] (Figure 5) and then issues `cudaMemPrefetchAsync` immediately to make each region resident on the GPU before the first access. This keeps pages UVM-managed without paying the migration fault costs at kernel launch. This substitution is the only change to the application with no kernel logic or control flow modifications. Purely changing the allocation does not incur runtime overhead (Section 7.3).

Checkpoint Flow. Tracking runs on top of the concurrent-copy/stop-and-copy protocol. On checkpoint start, **(1)** the agent sets all UVM-managed pages to Read-Only (RO), opening the tracking window, and begins a single concurrent-copy pass that streams pages from GPU VRAM to a host buffer over the ioctl-driven CE DMA path while the application runs. **(2)** Each write during this window generates write faults on RO pages; **(3)** the driver records in the dirty 2 MB region in the dirty bitmap and restores it to RW, so it never faults again. **(4)** At the next application quiesce point, which are identified by interposing on `cudaDeviceSynchronize` where in-flight stream work has drained and memory is consistent, the stop-and-copy phase copies only the accumulated dirty delta. **(5)** All page permissions revert to Read-Write (RW). The application pause is thus bounded to a small dirty working set rather than the full footprint.

Minimizing Fault Amplification. Tracking at 4 KB is precise but too costly since each KV-write raises individual 4 KB faults. However, for UVM driver to coalesce individual faults to 2MB granularity, it requires all 512 contiguous 4 KB pages to fault within the fault handling window, which is rarely satisfied during inference. This results in inference creating a "fault amplification". FLASH instead proactively upgrades the first fault in each 2 MB region to a single 2 MB fault, replacing up to 512 fault events with one and amortizing the driver cost across the region. This is deliberate trade-off behind our page granularity: it accepts the ~1.9× over-marking versus byte-level tracking (Section 4) in exchange for eliminating the amplified faults and needing no PTX analysis. Paged-Attention [25] keeps that over-marking acceptable by pre-allocating the KV cache as a contiguous memory pool and assigning blocks sequentially. The contiguous token generation writes cluster within the same 2 MB region and the pages we mark dirty are largely written anyway. Finally, FLASH tracks only during the concurrent-copy window, not the whole inference lifetime, to bound the overhead.

5.4 Static Allocation Classification

The dirty bitmap reveals not only which pages were written but which were never written. In LLM inference, the model weights are loaded once during initialization and never modified during serving. Therefore, they never appear in the bitmap across checkpoint intervals. FLASH classifies pages incrementally on this signal. While all pages begin as static candidates, these pages are permanently promoted to dynamic upon their first appearance in the dirty bitmap, and after a few intervals the classification converges as the dynamic set reaches a steady state. Once classified, FLASH excludes static pages from both concurrent-copy and stop-and-copy phase of each checkpoint interval. These static pages are captured once and retained as K_{D2H}-ciphertext in the image. In consecutive checkpoints, concurrent-copy phase streams only dynamic pages, shortening the phase and the dirty delta it accumulates.

Correctness. Static classification avoids the needless copying but never stops tracking. Consider an example where Page#X is classified as static and Page#Y as dynamic. Page#Y is copied into the image at every concurrent-copy phase, whether or not it was modified. Page#X is excluded from copying, but it is not excluded from tracking. Unlike dynamic pages, static pages are not reverted to RW at the end of the tracking window, so their read-only permission persists for the lifetime of the process and Page#X is monitored at every interval. If Page#X is written at any point, the write faults is issued, the page is marked dirty, and permanently promoted to dynamic. Page#X is captured at the next checkpoint, and restore applies the static region first and the dynamic region over it, so the only a single copy of Page#X is restored. Correctness, therefore, is preserved, and convergence is a performance property, not a correctness issue.

At restore, FLASH transfers both static and dynamic pages from the authenticated image into GPU VRAM, so the recovered state derives entirely from the verified checkpoint. It also re-injects the saved block table and per-request metadata into the restarted framework, so in-flight requests resume from the checkpoint positions. This is the only framework-specific step, while tracking remains framework-agnostic.

6 Security Analysis

FLASH inherits the security model of the baseline Confidential GPU C/R model (Section 3). In Section 6.1, we provide the security guarantees of the baseline C/R models: confidentiality, integrity and freshness. Section 6.2 extends the baseline security guarantees to FLASH's three optimizations: hybrid-key checkpoint format, UVM write-fault-based dirty tracking and static allocation classification.

6.1 Baseline Security Guarantees

The baseline guarantees rest on the trust assumptions of the confidential-computing standards. We trust the CVM and its guest kernel, protected by TDX RoT. We trust the GPU TEE, including its CE and the SPDM-derived session keys K_{D2H} and K_{H2D}. We trust the RAS, a separate trusted service that persists K_{CKPT} and the freshness counter across CVM teardown and operates outside the adversary's control (Section 3). Under these assumptions, the baseline provides confidentiality, integrity, and freshness.

Confidentiality. Checkpoint data that traverses outside the CVM boundary is always encrypted. The baseline restoration decrypts K_{D2H}-ciphertext into CVM private memory and reencrypts under K_{CKPT} before writing to remote storage (Section 6.1). K_{CKPT} is generated inside the CVM, transmitted to the RAS only over a mutually attested secure channel. Thus, it is never exposed to the malicious cloud provider as plaintext. Under standard AES-GCM assumptions, the checkpoint image is indistinguishable from a random blob to an adversary without K_{CKPT}.

Integrity. Any modification to the checkpoint image by the adversary, whether to CPU/GPU data or metadata, is detected at restore time via AES-GCM authentication tag verification under K_{CKPT}; the adversary cannot produce a valid tag without possessing K_{CKPT}.

Freshness. The monotonic counter `ctr`, maintained in CVM private memory and incremented at each checkpoint, is embedded in a TDX attestation report and signed by the hardware root of trust (Section 3). The RAS stores the latest $\{ctr, H(\mathcal{M})\}$ per CVM identity; the restore agent rejects any candidate image whose counter or manifest hash deviates from the RAS-held record. Since the adversary cannot forge TDX attestation reports or increment the counter outside the CVM, substitution of a stale image is detected.

IV Uniqueness. AES-GCM security requires that no (key, IV) pair is reused across encryptions. Within a single CVM lifetime, the checkpoint agent maintains a local IV counter in CVM private memory, incremented by a mutex for each chunk of data encrypted under K_{CKPT} using AES-GCM. The counter persists across multiple checkpoint rounds within the same CVM lifetime, ensuring IV uniqueness across successive checkpoints. At CVM teardown, IV reuse is precluded by construction: the destination CVM generates a fresh K_{CKPT} at restore time rather than reusing the source CVM's key, resetting the IV counter to zero under a new key. Each K_{CKPT} is therefore scoped to exactly one CVM lifetime, and no IV coordination across CVM restarts is required.

6.2 FLASH Security Analysis

FLASH introduces three optimizations: a hybrid-key checkpoint format (Section 5.2), UVM write-fault-based dirty tracking (Section 5.3), and static allocation classification (Section 5.4). We argue that each preserves the confidentiality, integrity, and freshness properties established above.

Hybrid-Key Checkpoint. Retaining the K_{D2H}-ciphertext in the image changes which key protects the GPU domain, not whether it is protected. **Confidentiality** holds because GPU data never appears as plaintext outside the GPU CPR. The CE encrypts inside the CPR, and the ciphertext flows through the bounce buffer into the image without decryption, exactly as on the DMA path. However, the format adds one dependency where K_{D2H} must be recoverable at restore and the K_{D2H} must be encrypted under K_{CKPT}. Thus, unless K_{CKPT} leaks, the adversary gains nothing over the baseline. The image now keeps track of two separate IV counters of each key. K_{CKPT} IVs are FLASH-managed as in the baseline, while K_{D2H} IVs are the CE's per-channel counters, incremented monotonically inside the CPR. No (key,IV) pair repeats within either domain. **Integrity** of CPU memory is preserved by K_{CKPT} and integrity of the GPU domain is preserved by K_{D2H}. K_{D2H} exists within an SPDM session established after the CVM verifies the GPU attestation reports against the device certificate chain and NVIDIA RIM measurement. A GPU that fails attestation never derives the key, so it is authentic at creation. The authenticity of K_{D2H} carries into the image because K_{D2H} is wrapped under K_{CKPT}. Substituting ciphertext under an attacker-chosen key requires wrapping that key under K_{CKPT}. An adversary who can do that already holds K_{CKPT}, at which point confidentiality and integrity are lost regardless. Eq (1) binds H(CVM_id), a measurement covering the C/R code that performs the attestation check, so provenance is carried through the measured identity of the verifying entity. Thus, tampering with either the GPU or CPU data is caught independently at restore. **Freshness** is unchanged, as the monotonic counter and TDX-signed report cover the whole image.

UVM-Based Dirty Tracking. Tracking runs entirely inside the CVM kernels. We execute the allocation interposition, RO permission changes, fault handling, and dirty bitmap in the guest under TDX protection, and none expose a new host-visible interface. Transferred pages still traverse the same CE DMA path with the same K_{D2H} encryption and GMAC authentication, so tracking neither exposes additional data nor alters the cryptographic properties of any page.

Static Allocation Classification. Static pages receive the same protection as the dynamic ones, so excluding them from the per-interval copy does not weaken confidentiality

Table 2. Workload Profiles and Model Configurations

Workload	Model	Tokens	Conc.	RPS	Max Len
Chatbot (Short)	Llama-3.2-1B	512	512	30.0	4096
	Qwen-2.5-7B	512	256	20.0	4096
	Llama-2-13B	512	64	5.0	4096
Reasoning (Long)	Llama-3.2-1B	8192	200	1.0	16384
	Qwen-2.5-7B	8192	64	0.5	10000
	Llama-2-13B	4096	32	0.5	4096

nor integrity. When the CE rotates K_{D2H} on IV counter exhaustion, FLASH regenerates their ciphertext under the new key, preserving IV uniqueness via the CE counter. At restore, both static and dynamic pages come from the authenticated image, so the recovered state derives entirely from the verified checkpoint. The partition itself is observable through the image which we discuss as a source of side-channel leakage in the next subsection.

Manifest Binding. Each checkpoint carries an authenticated manifest $\mathcal{M}$ recording, each page, its VRAM offset, IV counter, K_{D2H} key, and authentication tag. $\mathcal{M}$ is encrypted and authenticated under K_{CKPT} and $H(\mathcal{M})$ is bound into the TDX-signed freshness report (Section 6.1). This closes integrity gaps that per-page tags alone do not. Reordering or splicing valid pages into wrong positions, mixing dynamic pages across intervals, and stale-image substitution all change $H(\mathcal{M})$ and are rejected at restore. Because $\mathcal{M}$ records each page's epoch and the image wraps every referenced K_{D2H}, a checkpoint spanning a rare CE key rotation stays fully decryptable.

6.3 Side-Channel Considerations

FLASH's deltas expose additional metadata of the checkpoint state. The image size and the set of dirty pages reveal the dirty working set and the static/dynamic partition gathered across intervals. While these signals leak coarse workload intensity and memory layout, GPU memory contents are never leaked, as they remain encrypted. Like the underlying CPU and GPU TEEs, FLASH does not defend volume, access-pattern, or timing channels, much of which is already visible in bounce-buffer traffic.

6.4 Trusted Computing Base

Beyond the trusted components named in Section 6.1, FLASH adds two elements to the TCB logic which is a separate privileged agent that runs C/R through those IOCTLs, performs the encryption, and generates hybrid-key checkpoint images. Under CC, the entire guest already lies within the trust boundary, so these additions enlarge the trusted code base, and the observable surface with it.

Table 3. Comparison of checkpointing methodologies. SC denotes static classification (Sec. 5.4).

System	Checkpointing	Tracking Granularity	Dump Mechanism
Phos	Concurrent	Buffer-level	`cudaMemcpy`
GCR	Incremental	Byte-level	`cudaMemcpy`
FLASH–SC	Concurrent	Page-level	Custom IOCTL
FLASH	Concurrent	Page-level	Custom IOCTL

7 Evaluation

7.1 Methodology

Hardware Platform and System Setup. Experiments run on a single-node server with one NVIDIA H100 GPU in Confidential Computing mode and TDX-enabled Intel Xeon Gold 6548Y+ processors (Emerald Rapids) with 32 physical / 64 logical cores. We extend NVIDIA Driver 550.163.01 with UVM IOCTL that reads UVM-managed GPU pages, toggle permissions (RO↔RW), marking pages dirty on write-fault, and transfer D2H pages without CPU-side AES-GCM decryption. We use CUDA 12.4 (version that supports NVIDIA CC) which constrains us to vLLM [25] v0.8.5. To mimic production serving while isolating checkpoint overhead, we drive vLLM's in-process `AsyncLLMEngine` with a Poisson load generator bypassing the HTTP layer. Checkpoint images are staged in tmpfs (DRAM-backed).

Workload Designs and LLM Models. LLM serving stresses memory along several axes - request arrival rates (RPS), context lengths, model sizes, and attention architectures - so no single scenario captures the full range of pressure. We therefore use two contrasting workloads (Table 2): 1) Chatbot (high RPS, short-context) represents short-context conversational AI which stresses spatial fragmentation, and 2) Reasoning (low-RPS, long-context) represents agentic AI usage which stresses KV cache volume (heavy decode stage). We span three models with differing size and attention design: Llama-3.2-1B (Llama-1B) [16] and Qwen-2.5-7B (Qwen-7B) [6] (grouped-query attention), and Llama-2-13B (Llama-13B) [44] (multi-head attention).

Checkpoint Mechanisms. We compare FLASH and FLASH-SC (FLASH without static allocation classification, isolating that optimization) against two state-of-the-art GPU checkpointing systems, PhoenixOS (PhOS) [46] and GCR [48], as summarized in Table 3. Neither published artifact runs as-shipped on our hardware and driver stack: PhOS targets CUDA 11.3 with Ubuntu 20.04 on a non-TEE driver stack, and GCR depends on NVIDIA's cuda-checkpoint [34] for control-state and page-table preservation, which returns `CUDA_ERROR_NOT_SUPPORTED` under NVIDIA CC. We therefore re-implement each system's data path as an `LD_PRELOAD` library intercepting the CUDA runtime and driver allocation, copy, and kernel-launch APIs, and execute each system's

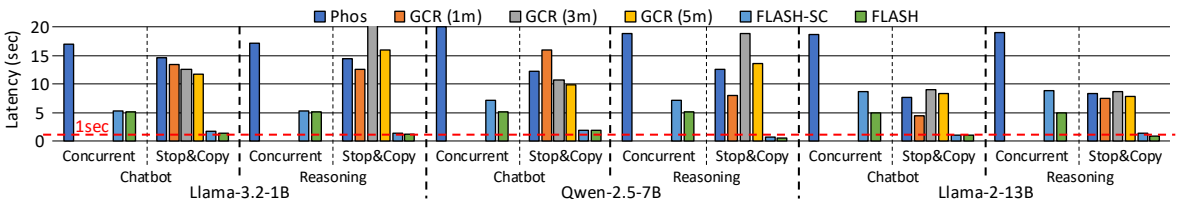


Figure 6. Concurrent-copy and stop-and-copy phase latency (sec).

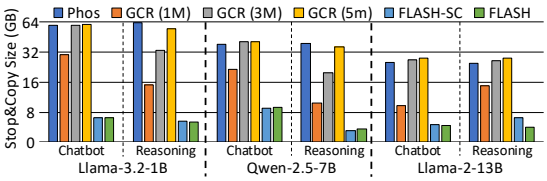


Figure 7. Dirty memory size in concurrent-copy phase.

published checkpoint protocol. For PhOS, we implement its four-phase protocol (quiesce / background concurrent copy / re-quiesce / stop-and-copy) with *buffer-granularity dirty tracking* via kernel-argument scanning, an approximation of PhOS' fatbin-derived metadata. For GCR, we implement shadow execution using the released, unmodified dirty-template kernel with byte-granularity dirty tracking on hits. PhOS and GCR dump using cudaMemcpy, while FLASH and FLASH-SC dump via custom-IOCTL path that implements the hybrid-key checkpoint (Section 5.2). To isolate the effect of tracking granularity, all systems share the same vLLM benchmark, same Poisson arrival generator, locked GPU clocks, AES-256-GCM encryption, and preallocated image backing. As the checkpoint interval matters most for incremental schemes like GCR where longer window dirties more KV cache and enlarges the delta processed during the pause, we sweep the interval over 1, 3, and 5 minutes.

Evaluation Metrics. Our primary metrics are system-level observations on the latency and captured volume of the concurrent-copy and stop-and-copy phases (Section 7.2). Additionally, to capture serving impact, we report Time-To-First-Token (TTFT) and Time-Between-Tokens (TBT) against Service Level Objectives (SLOs) of 100 ms (Section 7.3).

Implementation. Our modifications on the code consist of 2,876 LoC in the NVIDIA open-source driver and have been made available along with tools and libraries for reproduction at https://zenodo.org/records/23199897.

7.2 Checkpoint Latency and Volume.

The primary design objective of FLASH is to minimize the synchronous "stop-and-copy" phase that stuns GPU execution. Figure 6 shows both concurrent-copy and stop-and-copy phase latency while Figure 7 shows captured dirty memory during concurrent-copy phase, which determines the stop-and-copy phase latency.

Checkpoint Mechanism Comparison. FLASH and FLASH-SC report an average of 1.2 s stop-and-copy latency with its dirty volume of 12.0, 11.1 and 9.6 GB for Llama-1B, Qwen-7B and Llama-13B, respectively; stop-and-copy latency shows 9.4× improvement over PhOS, and 7.8× over GCR (1M).

PhOS experiences an average of 11.3 s of stop-and-copy latency, with copying 62.7, 52.0, and 42.1 GB for Llama-1B, Qwen-7B and Llama-13B, respectively. The measured dirty memory reflects the entire HBM memory allocated for KV cache as the application is configured to utilize 85% of 80GB HBM, 68GB excluding model weights. As PhOS lacks fine-grained tracking, long concurrent phase of average 18.4 seconds and dirty rate of average 2.8 GB/s results in dumping the entire KV-cache-allocated memory during stop-and-copy phase. We conclude that buffer-granular tracking used by PhOS is too coarse-grained to be used as checkpoint mechanism for inference workloads as it is more effective to simply apply stop-the-world checkpoint.

GCR shows different checkpoint characteristics over different intervals due to difference in dirtied memory volume across interval. While GCR (1M) captures less dirty memory compared to PhOS, the stop-and-copy latency is heavily penalized by D2H bandwidth limitation for Llama-1B (37.5 GB, 13.0 s), and Qwen-7B (28.4 GB, 11.3 s). GCR (1M) bandwidth is limited to an average of 2.75 GB/s in Llama-1B and Qwen-7B (compared to 4.41 GB/s in PhOS) which is induced by dirty memory being scattered over each buffer, and thus, resulting more `cudaMemcpy` APIs being launched to copy the byte-granular dirty memory. However, GCR (1M) Llama-13B shows dirty memory of 24.2 GB copied over 5.8 s of stop-and-copy phase (4.28 GB/s). Llama-1B and Qwen-7B uses grouped query attention (GQA) compared to multi-head attention of Llama-13B. While all three models exhibit the same fragmentation pattern from paged attention, the GQA models' smaller per-range payloads (a consequence of fewer KV heads) leave a larger fraction of each `cudaMemcpy` consumed by fixed CopyEngine setup overhead, dropping effective D2H bandwidth. A coalescer over the dirty-range list could merge writes to physically adjacent slots within the same block and lift GCR's effective bandwidth. We do not add such an optimization as it is not present in the published artifact.

GCR (3M) and (5M) have longer intervals, which result in copying near the entire GPU memory allocated for KV cache during stun-and-copy resulting in Llama-1B (58.0 GB, 16.0

Table 4. Ratio of concurrent copy time to the total time.

Workload	Llama-1B Chatbot/ Reasoning	Qwen-7B Chatbot / Reasoning	Llama-13B Chatbot / Reasoning
FLASH-SC	0.31 / 0.43	0.34 / 0.86	0.79 / 0.34
FLASH	0.31 / 0.42	0.24 / 0.74	0.52 / 0.34

s), Qwen-7B (48.0 GB, 14.2 s), and Llama-13B (44.0 GB, 8.9 s). Our results indicate that for any incremental checkpoint to be effective in LLM serving scenario, the checkpoint interval must be small enough so that dirty region accumulated over the intervals does not exceed the entire mutable memory.

FLASH-SC and FLASH, the stop-and-copy latency is 1.3 s and 1.1 s across models, copying only (12.1 GB, 12.0 GB) for Llama-1B, (10.7 GB, 11.5 GB) for Qwen-7B, and (10.9 GB, 8.5 GB) for Llama-13B. Comparing with PhOS, the 9.4× stop-and-copy latency reduction comes from three factors. First, our custom IOCTL implementation achieves an effective D2H bandwidth of 8.9 GB/s (2× improvement over average 4.2 GB/s measured in PhOS). In addition to encryption path optimization presented in Section 5.2, we minimize the number of memory transfer calls by designing our custom IOCTL to iterate through every dirty region and copy all dirty pages in a single invocation. This overcomes the limitation of `cudaMemcpy` that can only be called for one contiguous region per call[2] Second, we reduce dirty rate by 3.6× using page-granularity tracking compared to buffer-granularity. Lastly, we reduce the tracking period by 1.3×. Together, all three factors contribute to the 9.4× improvement over PhOS. Comparing with GCR (1M), while the page-granularity increase the dirty data rate by 80%, the tracking period reduces to 4.7×, contributing to 7.8× improvement.

Shorter Checkpoint Interval. While shortening the checkpoint interval for GCR below 1M would reduce the checkpoint image size, the fraction of time spent in stun-and-copy does not change. On Llama-1B, GCR stalls for 13.3 s per 60 s; halving the interval halves the stall to 6.7 s per 30 s. While the frequency may change, the fraction stays at 22% in both intervals. FLASH's fraction on the same workload is 2.1%, set by tracking granularity and effective bandwidth rather than by the interval. Thus, our methodology does not favor FLASH by depriving GCR's operating regime.

Tracking Period for FLASH. Intuitively, the volume of dirty memory accumulated prior to stop-and-copy phase might be modeled as the product of the concurrent copy latency and the application's dirty rate. However, this linear assumption does not hold in practice due to the execution dynamics of the underlying LLM serving. With high RPS and high concurrency, inference frameworks batch a massive number of compute kernels, which delays explicit host-device synchronization points (*i.e.* `cudaDeviceSynchronize`). Because FLASH does not directly issue these synchronization points into the application's execution context, the overall tracking period is dictated by the application's native synchronization schedule and thus, extends significantly beyond the concurrent-copy phase.

Table 4 highlights this asymmetry between concurrent-copy latency and the total tracking period. Across all models, the concurrent-copy latency is only 43.2% of the total tracking period on average. While forceful synchronization could theoretically truncate this extended tracking window and reduce the accumulated dirty memory, doing so would destroy kernel overlap and degrade steady-state serving throughput. FLASH accepts this extended window by design, reliably processing the additional dirty pages to maintain stable stop-and-copy time across workloads. Moreover, in scenarios where the application's native synchronization naturally aligns more closely with the background concurrent copy completion, such as Qwen-7B Reasoning workload, where the active copy time occupies up to 86% of the window, FLASH achieves outstanding performance. In this optimal case, the stop-and-copy stall drops to a negligible 0.66 seconds to serialize only 6.73 GB of memory, proving that exceptional checkpoint latency can be achieved entirely within the inference engine's unmodified native execution flow.

Workload: Chatbot vs Reasoning. For Llama-1B, both workloads show similar trend in stop-and-copy latency and volume, as the ratios of the concurrent-copy latency to the total tracking period are 31% for Chatbot and 43% for Reasoning (Table 4). For Qwen-7B running Reasoning workload, the ratio reaches up to 86%, indicating that the number of batched kernels is smaller and thus resulting in closer synchronization point. Consequently, the stop-and-copy latency is only average of 0.75 s for FLASH-SC and 0.58 s for FLASH. For Llama-13B running Chatbot, the dirty rates are 863 MB/s with the accumulated dirty volumes of 9.16 GB for FLASH-SC and 9.0 GB for FLASH. While Reasoning has longer dirty tracking window as shown in Table 4, with lower dirty rate of 501 MB/s, the accumulated dirty volumes are 12.9 GB (FLASH-SC) and 8.0 GB (FLASH), and the stop-and-copy latency stays consistent across workloads.

7.3 Impact on Inference Latency: TTFT and TBT

We focus on reasoning workloads shown in Table 5, which represent agentic AI use where GPU C/R is operationally critical for fault tolerance and live migration. Results of Chatbot workload are provided at the end of this subsection.

While we evaluate the checkpoint intervals of 1, 3, 5 minute, we concentrate our TTFT/TBT analysis on 1-minute interval, which represents the most aggressive checkpointing frequency and thus the highest-stress operating point. At

[2]While `cudaMemcpyBatchAsync` (introduced in CUDA 12.8) and `cudaMemcpy2D` accept multi-region or strided copies in a single call, they do not remove the redundant decryption-then-reencryption when gather dirty pages—a critical property that our custom kernel path uniquely provides. Furthermore, both APIs postdate the CUDA 12.4 runtime required by our Confidential Computing (CC) driver environment.

Table 5. TTFT/TBT measured with 60 s checkpoint interval running **Reasoning** workload.

	Llama-3.2-1B						Qwen-2.5-7B						Llama-2-13B					
	TTFT			TBT			TTFT			TBT			TTFT			TBT		
Method	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean
Baseline	57.4	60.6	50.4	28.3	30.6	25.2	51.5	56.8	42.7	21.8	22.6	20.1	107251.3	107251.3	25466.9	42.0	165.8	54.5
Baseline (UVM)	58.7	62.3	49.3	29.0	30.8	24.5	51.3	57.0	42.7	21.9	22.8	20.2	104886.3	124415.9	25279.4	42.4	166.6	54.5
PhOS	282.8	14364.9	503.9	30.7	32.6	30.1	52.2	59.4	45.0	21.7	23.4	24.8	105177.7	156155.5	27181.9	42.3	167.0	58.0
GCR	59.8	6385.6	187.2	28.8	30.2	25.8	54.5	61.4	42.7	22.0	29.3	21.3	108077.6	128499.5	26634.7	42.8	165.5	55.0
FLASH-SC	70.8	77.5	75.3	35.0	37.2	34.1	65.8	73.6	54.9	30.2	32.5	28.0	120863.0	214655.5	27660.3	49.4	169.2	69.8
FLASH	70.1	75.8	73.9	34.6	36.8	33.5	65.3	70.8	54.9	28.7	31.4	27.6	123475.4	159194.5	28028.3	49.9	169.1	63.9

this interval the checkpoint mechanism is exercised most frequently relative to serving time, amplifying per-checkpoint overhead and providing the most stringent test of each system's impact on serving quality. Additionally, we provide no-checkpoint Baseline and Baseline (UVM) that applies the allocator substitution (Section 5.3, Section 7.3).

UVM Substitution Overhead. FLASH's only change to the application is replacing `cudaMalloc` with `cudaMalloc Managed` with an immediate `cudaMemPrefetchAsync` (Section 5.3). Baseline (UVM) configuration that applies this substitution only confirms this is not costly. For Llama-1B and Qwen-7B, Baseline (UVM) tracks Baseline within ∼1 ms on both TTFT and TBT, within run-to-run variance. Llama-13B runs near saturation, where TTFT is dominated by queuing rather than allocation. TBT which represents the steady-state signal, is unchanged (mean 54.5 ms in both). The allocator substitution therefore adds no measurable overhead, and FLASH's serving impact reported below stems from write-fault tracking during checkpoint windows.

Llama-3.2-1B: Tail Latency. At P95, FLASH's TTFT (70.1 ms) is modestly higher than GCR (59.8 ms). However, P99 reveals a dramatic difference. PhOS's P99 TTFT reaches 14,364 ms, a 50× blowup from P95, while GCR's P99 of 6,385 ms shows similar catastrophic tail behavior. FLASH maintains P99 of 75.8 ms, within 8% of its P95: 189× lower than PhOS and 84× lower than GCR at P99. The Mean/95 ratio confirms the following: FLASH's ratio of 1.05 indicates a tight, predictable distribution, versus 1.78 (PhOS) and 3.13 (GCR) where catastrophic outliers pull the mean above P95. PhOS and GCR concentrate checkpoint disruption into severe stall events affecting a small fraction of requests catastrophically. In contrast, FLASH's write-fault-based tracking distributes uniform overhead across the tracking window, eliminating tail explosions at the cost of modestly elevated P95 over the no-checkpoint baseline of 57.4 ms (13.4% of SLO).

Qwen2.5-7B: Steady-State Overhead. Operating within comfortable steady state (baseline P95 TTFT: 51.5 ms), Qwen-7B provides the cleanest overhead measurement. All systems maintain well-behaved distribution, as P99 does not reveal a tail advantage for FLASH here (P99: 70.8 ms vs PhOS 59.4 ms, GCR 61.4 ms). Under sufficient serving headroom, checkpoint stalls are absorbed without queuing cascades, so the tail explosion observed in Llama-1B is not shown. Qwen isolates FLASH's intrinsic write-fault-based mechanism costs: 14 ms P95 TTFT overhead (13.8% of SLO).

Llama-2-13B: Queuing-Dominated Regime. With baseline P95 TTFT of 107 seconds, this model operates near saturation where queuing amplifies any checkpoint overhead non-linearly. FLASH's P99 (159,194 ms) is comparable to PhOS (156,155 ms). GCR achieves the lowest P99 (128,499 ms). In this regime, differences reflect interactions between checkpoint timing and queue state rather than mechanism characteristics. TBT P99 converges across all systems (167-169 ms), confirming that extreme-tail decode latency is dominated by the serving dynamics, not checkpoint design.

TBT Overhead. TBT overhead is consistent across models: FLASH adds approximately 6-8 ms at P95 without the extreme P99 divergence observed in TTFT. Decode steps are shorter and less likely to fully overlap with checkpoint stalls.

Overhead Analysis. FLASH imposes 14 ms P95 TTFT overhead but maintains bounded predictable latency (P99 within 8-10% of P95). PhOS and GCR achieve lower P95 but exhibit catastrophic P99 under load (50-107 × P95 blowup for Llama-1B). The tail explosion severity is prominent under moderate-to-high load (Llama-1B), is absent larger models (Qwen-7B), and is subsumed by queuing (Llama-13B). TBT shows a consistent increase across all models. The steady-state overhead in TTFT and TBT results from the stalls caused by UVM's write-fault handling mechanism. While we optimize UVM's fault handling mechanism by proactively upgrading individual sub-2MB-page faults to a single 2MB fault on the first fault, each fault reports 25μs round-trip latency [26]. Combined with FLASH's ∼9.4× stop-and-copy latency reduction (Section 7.2), the results demonstrate that FLASH enables frequent checkpointing with bounded serving impact, eliminating the catastrophic tail events that existing systems produce under realistic loads.

Overhead on Chatbot. Under conversational Chatbot workloads characterized by short-context inputs, high requests-per-second (RPS), and dense concurrency, a localized tail-latency outlier emerges for the edge-scale Llama-1B model

Table 6. TTFT/TBT measured with 60 s checkpoint interval running **Chatbot** workload

	Llama-3.2-1B						Qwen-2.5-7B						Llama-2-13B					
	TTFT			TBT			TTFT			TBT			TTFT			TBT		
Method	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean
Baseline	43.3	48.4	26	14.5	16.5	8.4	74.6	87	55.4	31.4	39.3	22.6	59.4	74.7	37.4	19.9	24.9	18.9
PhOS	6630.6	12750.9	711.1	20.3	27.3	19	160.8	655.8	151.7	33.1	72.8	31.6	8094.4	11872.6	1072.3	27.4	30	29.9
GCR	8549	13030.5	900.9	22.3	30.4	16.7	135.1	182	115.5	37.4	51.9	29.5	73.8	5725.2	189.7	20.7	31.6	21
FLASH-SC	14128.3	22034.9	1308.8	27.9	34	24.4	86.7	111.9	88.4	40.9	47.6	35.9	101.2	124.9	82.4	39.7	58.6	39.2
FLASH	13314.8	18855.8	1195.2	28.4	34	24.8	91.1	110.1	87.6	40.8	47.7	35.7	100.2	122	78.6	39.5	58	38.6

at the most aggressive 60-second checkpointing interval. As shown in Table 6, both FLASH-SC and FLASH experience significant performance degradation under this extreme configuration. We analyze the root cause as *Absence of Compute Cushion*. Llama-1B model processing short-context input runs lightweight execution kernel. In contrast, larger production-scale models (*e.g.* Qwen-7B or Llama-13B) spend a much higher proportion of cycles executing dense matrix multiplication, providing a natural "compute cushion" that effectively hides background asynchronous data management and UVM write-fault overhead. Llama-1B model lacks this cushion, leaving its execution transparent to underlying overhead. Consequently, all FLASH-induced overheads, which include write-fault serving latency, TLB invalidation, and permission changes, are directly exposed in the critical path of token generation. For larger models, where a single decode step involves tens of milliseconds of dense computation, a few milliseconds of checkpoint-related overhead represents a small fractional increase.

We note that this outlier is *not unique to FLASH*. Under the same 60-second-interval Chatbot configuration, PhOS exhibits P95 TTFT of 6,630.6 ms and GCR reaches 8,549.0 ms, both orders of magnitude above baseline. All checkpoint systems suffer high overhead on Llama-1B Chatbot workload because the model's minimal compute footprint leaves no margin to absorb any checkpoint disruption. FLASH's higher absolute P95 TTFT (13,314.8 ms) compared to PhOS and GCR in this specific configuration reflects the overhead from permission-based dirty tracking where execution kernels are lightweight to amortize per-fault costs.

For Qwen-7B and Llama-13B under Chatbot workloads, FLASH's TTFT overhead is well bounded. Qwen-7B reaches P95 TTFT of 81.1,ms against a 74.5 ms baseline, a 6.5% increase, and *outperforms* both PhOS (160.8 ms) and GCR (135.1 ms). Llama-13B follows the same pattern, with FLASH at 100.2 ms versus a 59.4 ms baseline while PhOS degrades to 8,094.4 ms; GCR is lowest here at 73.8 ms. The larger compute footprint of these models provides enough execution depth to absorb tracking overhead within normal decode variation.

FLASH shows higher P95 TBT than PhOS and GCR across all Chatbot configurations: 28.4 ms versus 20.3 and 22.3 ms on Llama-1B, 47.7 versus 33.1 and 37.4 ms on Qwen-7B, and 39.5 versus 27.4 and 20.7 ms on Llama-13B. This gap is the per-decode-step cost of write-fault tracking when decode overlaps the active tracking window. Because decode steps are short, they rarely coincide with the concentrated stalls that drive PhOS and GCR's TTFT tails, so TBT differences stay moderate in absolute terms.

7.4 Restore Overhead.

Restore is a one-time, post-fault operation off the serving hot path, so its cost is amortized over the lifetime of the recovered session. Unlike checkpoint, it cannot reuse the hybrid-key shortcut: the retained K_{D2H}-ciphertext cannot be injected into the GPU directly, since the inbound path re-encrypts under the new session's K_{H2D}. FLASH instead decrypts the retained ciphertext in a custom IOCTL, which recovers the plaintext from the checkpoint image in the CVM private memory. Then, the plaintext is loaded into GPU VRAM with a standard cudaMemcpy, where the CE re-encrypts it under K_{H2D} (Section 5.2). Static classification affects only the checkpoint path, so FLASH and FLASH-SC restore the full footprint (both model and KV cache must be loaded for correctness), and thus we report only FLASH.

Restore moves the full ~66–67 GB footprint and is dominated by the penalized H2D transfer (Section 4). End-to-end restore takes 18.7, 17.1, and 18.1 s for Llama-1B, Qwen-7B, and Llama-13B (all under 19 s). Restore pays the full transfer cost once per fault, off the hot path.

7.5 Optimal Deployment Strategy

For unannounced failure, the Young-Daly optimum [12] gives a checkpoint period of $\sqrt{2 \cdot C \cdot M}$ with wasted wall time $\sqrt{2 \cdot C/M}$, where C is the stall time, and M is the mean time between failures. The deployment depends only on C, not M, so FLASH at C=1.1s is 3.2x less than PhOS at C=11.3s regardless of failure rate. At M=3 days, FLASH's optimum is 12 m with 0.29% waste, against 40 m at 0.93% for PhOS. Our 1–5m sweep sits below this optimum, so the reported overheads are upper bounds for crash recovery.

8 Related Work

Trusted I/O (TIO) Trusted I/O, supported by both Intel TDX Connect [20] and AMD SEV-TIO [5], extends the CVM trust

boundary to an attested device so that the device can DMA directly into CVM's private memory (vice versa) and thus removes the software-managed bounce buffer. Trusted I/O is compatible in NVIDIA's Blackwell generation [2]. With trusted I/O, the data transfer is still protected with cryptography, but the (en/de)cryption is relocated to inline hardware on the PCIe link instead of the software-based path that relies on the bounce buffers.

Trusted I/O affects our hybrid-key checkpoint (Section 5.2). In the current bounce-buffer-based transfer, FLASH eliminates the redundant decrypt-then-reencrypt process by retaining the GPU-encrypted ciphertext directly in the checkpoint image. However, with trusted I/O, this redundancy is structurally absent, as the GPU directly delivers plaintext into CVM's private memory, leaving no intermediary buffer whose ciphertext we could reuse. This optimization targets the broadly deployed hardware we evaluate, while FLASH's remaining contributions of page-granular dirty tracking and static allocation classification stay with trusted I/O.

TEE State Continuity and Rollback Protection. Memori [37] provides practical state continuity for protected modules using an attested counters and a trusted external service. ROTE [27] extends rollback protection to distributed Intel Software Guard Extensions (SGX) enclaves by establishing a mutual-verification protocol among a group of enclaves, removing the need for a single trusted third party. Burke et al. [10] demonstates rollback attacks on confidential cloud applications more broadly, analyzing how serialized CVM state can be replayed or forged by a malicious host. Swidowski et al. [42] identify exploitable vulnerabilities in Intel TDX's live migration path. FLASH inherits the monotonic-counter and attested-report design but extends the problem to GPU TEE, where checkpoint data traverses an encrypted PCIe transfer path before reaching CVM, introducing a dual-key domain that the hybrid-key checkpoint format addresses.

GPU Checkpoint/Restore Early GPU C/R systems such as CheCUDA [43] and CRAC [21] established the stop-the-world approach where they quiesce GPU execution, copy the full device memory to the host, and resume. NVIDIA's CUDA-checkpoint [34] integrates this capability into the driver, providing transparent checkpointing via CRIU, but retains the stop-the-world model and is explicitly unsupported when CC mode is enabled [32, 34]. CRIUgpu [41] extends CRIU with GPU plugins for both NVIDIA and AMD devices, achieving fully transparent checkpointing but similarly following the full-dump model with stall time proportional to GPU memory footprint. CRUM [14] introduces fork-based copy-on-write checkpointing for UVM-managed memory, overlapping storage write with application execution, but targets HPC workloads on pre-CC hardware and does not address dirty tracking or security.

PhoenixOS (PhOS) [46] is state-of-the-art OS-level concurrent GPU C/R, overlapping full footprint dump with execution via copy-on-write and recopy. Lacking hardware dirty bits, it infers write sets from GPU API calls and kernel arguments, tracking at buffer granularity. As Section 4 shows, this badly over-marks LLM inference, where the KV cache is a single large allocation but each serving interval writes only a few pages.

GCR [48] adopts incremental checkpoint through shadow execution, tracking dirty buffers on the CPU from pregenerated, PTX-derived templates. Two limitations exist with their method. (1) Byte granularity falls back to buffer granularity when dirty addresses depend on memory indirection, and (2) its offline analysis cannot handle the runtime compilation.

FLASH differs from all prior GPU C/R systems. First, FLASH is the first to provide confidentiality, integrity, and freshness guarantees for GPU checkpoint images under a malicious-operator threat model. Second, FLASH's UVM write-fault mechanism tracks modified pages at 2 MB granularity without application source code or offline analysis, remaining effective regardless of write pattern. Lastly, the hybrid-key checkpoint optimizes a redundant AES-GCM decrypt-then-reencrypt cycle imposed by CC's encrypted transfer path.

Multi-GPU support. Multi-GPU CC requires Blackwell's TEE-I/O and NVLink inline-encryption, so it is unavailable on our H100 testbed, and we evaluate single-GPU only. FLASH's dirty tracking is per-device by construction. The one mechanism that does not transfer directly is NVLink peer writes, which bypass the destination GMMU; we exclude peer-mapped and IPC-exported pages from the static set, so lifetime write-protection remains sound under parallelism.

9 Conclusion

FLASH is the first checkpoint/restore system designed for GPU Confidential Computing; the first work to make secure checkpointing practical under CC's structural 5.5× PCIe bandwidth penalty, which renders a naive full dump prohibitively slow. Three optimizations deliver this practicality. A hybrid-key checkpoint format retains GPU-encrypted data directly in the checkpoint image, eliminating a redundant AES-GCM decrypt-then-reencrypt cycle. UVM write-fault-based dirty tracking identifies modified pages at 2MB granularity without kernel source code or offline analysis. Static allocation classification excludes never-written memory regions from the critical path. On an H100 with Intel TDX running vLLM, FLASH reduces stop-and-copy latency to an average of 1.2 s, a 9.4× over PhOS and 7.8× over GCR. It achieves this by transferring only the dirty working set rather than the full memory footprint, with bounded serving impact. By bringing secure checkpointing within the cost that inference serving can absorb, FLASH makes C/R a practical option for confidential LLM inference rather than one that CC's constraints rule out.

Acknowledgment

We sincerely thank all the reviewers for their time and constructive comments. This material is based upon work supported by Broadcom gift. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] [n. d.]. NVIDIA Confidential Computing. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf. Accessed: 2024-10-07.

[2] 2024. NVIDIA Blackwell Architecture Technical Brief. https://resources.nvidia.com/en-us-blackwell-architecture. Accessed: 2024-03.

[3] 2025. NVIDIA Confidential Computing. https://docs.nvidia.com/nvidia-secure-ai-with-blackwell-and-hopper-gpus-whitepaper.pdf. Accessed: 2025-08-14.

[4] 2025. NVIDIA RTX PRO Blackwell Architecture. https://blogs.nvidia.com/blog/rtx-pro-6000-blackwell-server-edition/. Accessed: 2025-03.

[5] Advanced Micro Devices, Inc. 2023. *AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization.* Whitepaper. AMD Developer. https://www.amd.com/content/dam/amd/en/documents/developer/sev-tio-whitepaper.pdf Accessed: 2026-06-04.

[6] Imtiaz Ahmed, Sadman Islam, Partha Protim Datta, Imran Kabir, Md Naseef Ur Rahman Chowdhury, and Ahshanul Haque. 2025. Qwen 2.5: A comprehensive review of the leading resource-efficient llm with potentioal to surpass all competitors. (2025).

[7] Tiago Alves. 2004. Trustzone: Integrated hardware and software security. *Information Quarterly* 3 (2004), 18–24.

[8] AMD. 2019. SEV API Version 0.22.

[9] ARM. 2021. ARM CCA Security Model 1.0.

[10] Quinn Burke, Anjo Vahldiek-Oberwagner, Michael Swift, and Patrick McDaniel. 2025. It's a Feature, Not a Bug: Secure and Auditable State Rollback for Confidential Cloud Applications. *arXiv preprint arXiv:2511.13641* (2025).

[11] Xiaoyu Chu, Sacheendra Talluri, Qingxian Lu, and Alexandru Iosup. 2025. An empirical characterization of outages and incidents in public services for large language models. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*. 69–80.

[12] John Daly. 2003. A model for predicting the optimum checkpoint interval for restart dumps. In *International Conference on Computational Science*. Springer, 3–12.

[13] Gobikrishna Dhanuskodi, Sudeshna Guha, Vidhya Krishnan, Aruna Manjunatha, Rob Nertney, Michael O'Connor, and Phil Rogers. 2023. Creating the first confidential gpus. *Commun. ACM* 67, 1 (2023), 60–67.

[14] Rohan Garg, Apoorve Mohan, Michael Sullivan, and Gene Cooperman. 2018. CRUM: Checkpoint-restart support for CUDA's unified memory. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 302–313.

[15] Google Cloud. 2025. Create a Confidential VM Instance with GPU. https://docs.cloud.google.com/confidential-computing/confidential-vm/docs/create-a-confidential-vm-instance-with-gpu. Accessed: April 2026.

[16] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[17] Zhongshu Gu, Enriquillo Valdez, Salman Ahmed, Julian James Stephen, Michael Le, Hani Jamjoom, Shixuan Zhao, and Zhiqiang Lin. 2025. NVIDIA GPU confidential computing demystified. *arXiv preprint arXiv:2507.02770* (2025).

[18] Intel. 2017. Intel Hardware Shield- Intel Total Memory Encryption. https://www.intel.com/content/www/us/en/architecture-and-technology/total-memory-encryption-security-paper.html.

[19] Intel. 2020. Intel Trust Domain Extensions Whitepaper. https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-final9-17.pdf.

[20] Intel Corporation. 2023. *Intel® TDX Connect TEE-IO Device Guide.* Technical Report. Intel Corporation. https://intel.com

[21] Twinkle Jain and Gene Cooperman. 2020. Crac: Checkpoint-restart architecture for cuda with streams and uvm. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[22] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, Frank Mckeen, and Intel Corp. 2016. Intel software guard extensions: EPID provisioning and attestation services. *White Paper* 1, 1-10 (2016), 119.

[23] David Kaplan. 2017. Protecting VM register state with SEV-ES. *White paper* (2017).

[24] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[25] Woosuk Kwon, Zhuowei Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*.

[26] Kyushick Lee, Michael B Sullivan, Siva Kumar Sastry Hari, Timothy Tsai, Stephen W Keckler, and Mattan Erez. 2019. Gpu snapshot: checkpoint offloading for gpu-dense systems. In *Proceedings of the ACM International Conference on Supercomputing*. 171–183.

[27] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. {ROTE}: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*. 1289–1306.

[28] Meta. 2026. *Private Processing for WhatsApp: Technical White Paper and Security Guide.* Technical Report. V2 updated March 16, 2026.

[29] Microsoft. 2024. Azure Confidential GPU Options. https://learn.microsoft.com/en-us/azure/confidential-computing/gpu-options. Accessed: April 2026.

[30] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. 2024. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 3, Article 36 (Dec. 2024), 42 pages. doi:10.1145/3700418

[31] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. {CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 203–216.

[32] Rob Nertney. 2026. Inquiry on Checkpointing on Confidential Computing. NVIDIA Developer Forums. Available at: https://forums.developer.nvidia.com/t/inquiry-on-checkpointing-on-confidential-computing/361667.

[33] NVIDIA. 2023. Confidential Compute on NVIDIA Hopper H100. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf.

[34] Nvidia. 2024. NVIDIA Cuda Checpoint. https://developer.nvidia.com/blog/checkpointing-cuda-applications-with-criu/.

[35] NVIDIA. 2025. AI Security with Confidential Computing. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/.

[36] Nvidia. 2025. NVIDIA Attestation. https://docs.nvidia.com/attestation/index.html.

[37] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. 2011. Memoir: Practical state continuity for protected modules. In *2011 IEEE Symposium on Security and Privacy*. IEEE, 379–394.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia

Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[39] Shangshu Qian, Kipling Liu, PC Sruthi, Lin Tan, and Yongle Zhang. 2026. Towards Resiliency in Large Language Model Serving with KevlarFlow. *arXiv preprint arXiv:2601.22438* (2026).

[40] AMD Sev-Snp. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* 53, 2020 (2020), 1450–1465.

[41] 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 preprint arXiv:2502.16631* (2025).

[42] Kirk Swidowski, Daniel Moghimi, Josh Eads, Erdem Aktas, and Jia Ma. 2026. Security Assessment of Intel TDX with support for Live Migration. *arXiv preprint arXiv:2602.11434* (2026).

[43] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. 2009. CheCUDA: A checkpoint/restart tool for CUDA applications. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 408–413.

[44] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[45] Jacob Wahlgren, Gabin Schieffer, Ruimin Shi, Edgar A León, Roger Pearce, Maya Gokhale, and Ivy Peng. 2025. Dissecting CPU-GPU unified physical memory on AMD MI300A APUs. In *2025 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 368–380.

[46] 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*. 996–1013.

[47] Yang Yang, Mohammad Sonji, and Adwait Jog. 2025. Dissecting performance overheads of confidential computing on GPU-based systems. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 1–16.

[48] Shaoxun Zeng, Tingxu Ren, Jiwu Shu, and Youyou Lu. 2026. {GPU}{Checkpoint/Restore} Made Fast and Lightweight. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*. 239–254.

[49] Shixuan Zhao, Zhongshu Gu, Salman Ahmed, Enriquillo Valdez, Hani Jamjoom, and Zhiqiang Lin. 2025. GPU Travelling: Efficient Confidential Collaborative Training with TEE-Enabled GPUs. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 2653–2667.

[50] Weixi Zhu, Guilherme Cox, Jan Vesely, Mark Hairgrove, Alan L Cox, and Scott Rixner. 2022. Uvm discard: Eliminating redundant memory transfers for accelerators. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 27–38.