FlowGPU: Transparent and Efficient GPU Checkpointing and Restore

Zehua Yang[1], Xiao Zheng[2], Yonghao Zou[1], Junyang Zhang[1], Zhisheng Ye[1], Feng Xie[2], Xiaolin Wang[1], Yingwei Luo[1], Zhenlin Wang[3], and Diyu Zhou[1]

[1] Peking University, Beijing, China
{yzh182_,junyang,zouyonghao}@stu.pku.edu.cn,
{yezhisheng,wxl,lyw,diyu.zhou}@pku.edu.cn
[2] Alibaba Group, China
{zhengxiao.zx,stephen.xf}@alibaba-inc.com
[3] Michigan Technological University, Houghton, MI, USA
zlwang@mtu.edu

Abstract. GPU checkpointing and restore promise to enable emerging tasks, such as deep learning, to benefit from functionalities like task scheduling and fault tolerance. However, existing GPU checkpointing/restore solutions suffer from runtime overhead, bloated checkpoint images, and correctness issues. This paper presents FlowGPU, a system-level GPU checkpointing/restore mechanism that overcomes all aforementioned limitations. Our key insight is that the limitations of prior mechanisms implicitly stem from their architectural design, which tightly couples checkpointing/restore with a legacy virtualization technique: API forwarding. In response, the design of FlowGPU decouples checkpointing/restore from virtualization with two key techniques: per-task interception and ghost process, thereby overcoming these limitations. Furthermore, FlowGPU comes with a set of novel techniques to further improve performance and ensure correctness under complex scenarios, such as a task operating on multiple GPUs. Our evaluation shows that FlowGPU outperforms prior mechanisms by up to 4.5×.

Keywords: GPU checkpointing · checkpoint/restore · deep learning · task migration

1 Introduction

Checkpointing (*i.e.*, saving program state as a checkpoint image in memory) and restore (*i.e.*, recreating program state from a checkpoint image) are a pair of basic system primitives that enables various core functionalities, such as task scheduling and migration [21,23], fault tolerance and elastic GPU scaling [17]. Given their importance, checkpointing and restore have been extensively studied for conventional CPU applications [3].

Recent years have also seen a rapid advancement in GPU computing, driven by emerging highly parallel tasks, such as deep learning (DL) and scientific

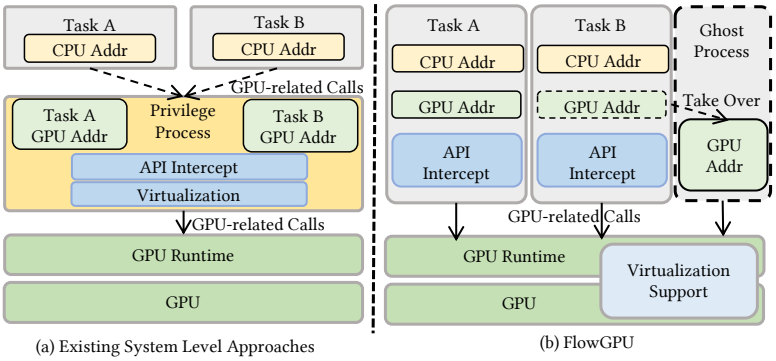

Fig. 1: Existing mechanisms vs. FlowGPU.

computation [2,9,8]. Supporting checkpointing/restore for these emerging GPU tasks enables them to benefit from various useful functionalities discussed above.

Previous mechanisms [14,7,17,10] fulfill checkpointing/restore outside the task to maintain transparency, but suffer from key limitations, including ❶ runtime overhead during normal execution (*i.e.*, outside checkpoint/restore); ❷ incurring correctness issues upon GPU sharing; ❸ violating transparency by preventing tasks from using certain GPU features; ❹ imposing high checkpointing/restore time.

This paper presents FlowGPU, a system-level GPU checkpointing/restore mechanism that overcomes all aforementioned limitations of previous mechanisms. FlowGPU identifies that limitations ❶ to ❸ stem from coupling checkpointing/restore with a legacy GPU virtualization technique: API forwarding [17,10] (detailed in §3). As shown in Figure 1(a), API forwarding routes all GPU operations through a central privileged process, incurring IPC overhead (❶), placing all task GPU memory in the shared central process which causes address conflicts upon restore (❷), and preventing features like Unified Memory [19] (❸).

As shown in Figure 1(b), FlowGPU decouples checkpointing/restore from GPU virtualization with two key techniques. First, a per-task intercept library intercepts GPU operations within each task privately, eliminating IPC overhead. Second, upon checkpointing, FlowGPU creates a ghost process that takes over the task's GPU state, enabling state separation and parallel checkpointing/restore without API forwarding during normal operation. Thanks to decoupling, FlowGPU leverages modern virtualization (*e.g.*, MPS [13], MIG [12]) upon GPU sharing, or forgoes virtualization entirely for exclusive GPU access.

FlowGPU further proposes general techniques to overcome limitation ❹ and address correctness issues. A key performance technique accurately identifies active memory in DL tasks: DL frameworks implement allocators that almost never return memory to the GPU runtime, causing naively saved checkpoint images to be bloated (up to 137×, §3). FlowGPU exploits the stable memory management interfaces of DL framework backends to identify active memory without modifying task code. We implemented FlowGPU with 19,700+ lines of code; our evaluation shows it outperforms prior mechanisms by up to 4.5×.

In summary, we make the following contributions: **Insights:** We reveal significant limitations of existing mechanisms and found that they implicitly stem from coupling checkpointing/restore with API forwarding. **Architectural design:** We propose an architecture decoupling checkpointing/restore from GPU virtualization, overcoming limitations of prior mechanisms while maintaining the benefits of API forwarding. **General techniques:** We propose general optimizations including active memory identification, resolving performance and correctness issues unaddressed by prior work.

2 Background

GPU virtualization GPU virtualization enables multiple tasks to share a single GPU simultaneously. This improves GPU utilization rate (by 34% as reported in [22]), leading to an increase in aggregate cluster throughput (by 17% [22]).

One legacy way to virtualize GPUs is API forwarding [5,16,18], proposed before hardware/runtime support for GPU virtualization is available. With API forwarding, all GPU operations performed by tasks (*e.g.*, launching kernels, allocating/ freeing GPU memory) are forwarded to a central privileged process, as shown in Figure 1.

However, due to the limitations of API forwarding, GPU vendors introduce virtualization support into the GPU runtime and/or hardware. For example, back in 2012, Nvidia introduced Multi-Process Service (MPS) [13] and Multi-Instance GPU (MIG) [12] for sharing GPU among tasks. AMD GPUs also provide similar mechanisms [1].

Active Memory in Deep Learning Framework Deep learning (DL) frameworks (*e.g.*, PyTorch, TensorFlow) often have their own memory allocators to improve performance. This is because allocating and freeing GPU memory from the GPU runtime is slow, often taking hundreds of microseconds. Framework allocators improve performance by directly serving task memory requests and only asking for a large memory block from the GPU runtime when it runs out of memory. In addition, allocators in a framework never return memory to the GPU runtime, even if such memory is unused by any task. Framework allocators bring a special characteristic to DL tasks. That is, memory allocated to and thus used by a task, which we term **active memory**, is much smaller than the total amount of memory allocated by the GPU runtime.

Furthermore, active memory fluctuates during training: the maximum size (after a forward pass, storing activations) can be 3× to 124× larger than the minimum (end of iteration).

3 Motivation

Framework-provided checkpointing [20,15] (*e.g.*, in PyTorch and TensorFlow) requires tasks to pre-define checkpoints, which is misaligned with cloud scheduling: pre-defined checkpoints cannot match arbitrary scheduling events, cloud providers

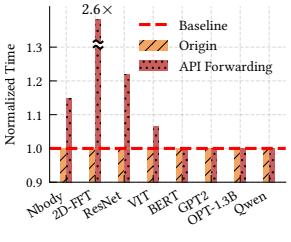

Fig. 2: Runtime overhead caused by API forwarding.

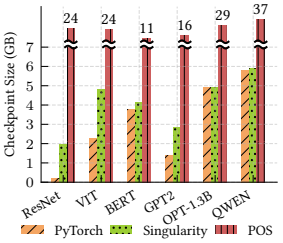

Fig. 3: A comparison of checkpoint image sizes.

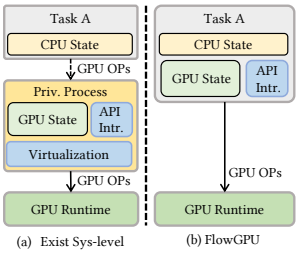

Fig. 4: Existing mechanisms vs FlowGPU.

cannot mandate tenants modify code, and framework-specific tools exclude non-framework tasks.

System-level Checkpointing and Restore Pioneering work includes rCUDA [14] and Cricket [7]; Singularity [17] adds DL-domain optimizations; POS [10] supports concurrent execution and C/R.

Coupling C/R with API Forwarding. Prior system-level mechanisms [17,10] couple checkpointing/restore with API forwarding: as shown in Figure 4, all C/R functionality resides in the central privileged process, so all mechanisms require API forwarding to be enabled. Such coupling is deliberate [17,10]: API forwarding simplifies interception (since existing mechanisms directly reuse its central process for recording GPU operations), and cleanly separates GPU state from non-GPU state, enabling CRIU [3] reuse for non-GPU state and parallel C/R.

Limitations in the coupled design. Even when a single task runs a GPU (no sharing needed), the coupling forces API forwarding, imposing the following limitations: **Runtime overhead.** Figure 2 shows that with POS, forwarding overhead reaches 1.2× for DL tasks and 2.6× for general GPU tasks, due to large IPC data transfers. **Correctness.** API forwarding maps all tasks' GPU memory into the shared central process, making it inherently difficult to ensure that restore places memory at the same addresses (*e.g.*, if task A held `0x1000` before checkpoint, another task may claim `0x1000` in the meantime). **Transparency.** Certain GPU features (*e.g.*, Unified Memory [19]) require the task to directly access the GPU, which is at odds with API forwarding's exclusive central-process model.

Bloated Checkpoint Image Another limitation of prior mechanisms is that they fail to accurately identify active memory (§2) to minimize the size of checkpoint images for DL tasks, thereby increasing checkpointing/restore time. Figure 3 shows the issue discussed above. POS [10] (as well as rCUDA [14] and Cricket [7]) simply checkpoint all memory allocated by the GPU runtime, producing checkpoint images 6×-137× larger than PyTorch.

Singularity [17] leverages domain knowledge about PyTorch, examining the call stack to predict whether allocated memory will be active. This approach must be re-tuned per PyTorch version, and making such predictions correctly is extremely difficult. Even so, Singularity still produces images up to 10× larger

than PyTorch (Figure 3) and increases migration latency by 2× (§5.3), as it marks whole allocated blocks as active memory even when only part is used.

4 Design

Design Goals We design FlowGPU to meet four goals: ❶ **Transparency**: function correctly without modifying task code; ❷ **Minimal runtime overhead**: no overhead during normal operation (§3); ❸ **Minimal C/R time**: minimize checkpoint image size (§3); ❹ **Correctness**: handle all scenarios including GPU sharing and multi-GPU tasks.

4.1 Overview

The Standalone Architecture. Coupling checkpointing/restore with API forwarding, as employed in prior work (§3), violates the *transparency*, *minimal runtime overhead*, and *correctness* design goals. Therefore, FlowGPU decouples checkpointing/restore from any specific virtualization technique, enabling tasks to use any technique or even completely forgo it.

The key challenge in decoupling is maintaining the benefits offered by API forwarding (§3). FlowGPU overcomes this with two key techniques: per-task API interception and ghost process, as detailed next.

To maintain the benefit of API interception, our observation is that such interception can and should be done *individually* for each task, rather than in a *shared* central process as in API forwarding. A per-task interception achieves the same for checkpointing/restore, while avoiding IPC overhead. Therefore, as shown in Figure 5, FlowGPU employs a per-task intercept library that intercepts GPU operations within each task, eliminating one key reason for using API forwarding. We name the architecture enabled by intercept library as "standalone architecture", since it contrasts with the client-server architecture in API forwarding.

Ghost process. Another reason for prior mechanisms to couple API forwarding with checkpointing/restore is to separate non-GPU and GPU state, thereby enabling two benefits (§3). FlowGPU demonstrates that this advantage is not inherent, and addresses it with the ghost process technique.

The key insight behind ghost process is that separating GPU and non-GPU state is only required during checkpointing/restore, not normal operation. Therefore, upon checkpointing, FlowGPU creates a ghost process that takes over the task's GPU state via GPU shared memory support, then unmaps GPU memory from the task, and cleaning the CUDA related state, making it a conventional process. As shown in Figure 6, this enables the same benefits as API forwarding: CRIU reuse for non-GPU state and parallel C/R.

Benefits of the decoupling. Thanks to the intercept library and ghost process, FlowGPU eliminates API-forwarding limitations. In particular, during normal operation, GPU operations go directly to the GPU without IPC, thereby eliminating runtime overhead. Furthermore, in both cases, the virtual address

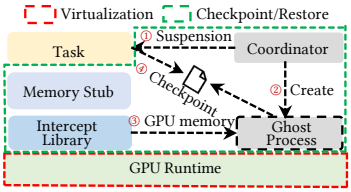

Fig. 5: Overview of FlowGPU.

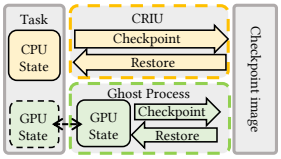

Fig. 6: Parallel checkpointing/restore with the ghost process.

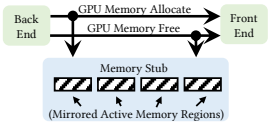

Fig. 7: Intercepting memory operations to identify active memory.

of each GPU memory resides in the private address space of each task, thereby avoiding GPU address conflicts during checkpointing/restore.

Other components. As shown in Figure 5, the standalone architecture further includes an optional per-task memory stub that identifies active memory for DL tasks (§4.2), and a coordinator for global synchronization.

Workflow. During normal operation, the intercept library records GPU operations for later replay. Upon checkpointing, the coordinator ❶ pauses the task, ❷ creates the ghost process, ❸ separates GPU/non-GPU state, and ❹ in parallel: checkpoints non-GPU state with CRIU [3] and saves GPU memory via the ghost process. Restore is the reverse; the intercept library replays intercepted functions to reconstruct GPU state.

4.2 Minimal checkpointing/restore time

Identifying Active Memory Unlike Singularity (§3), FlowGPU's approach generalizes across DL frameworks, requires no per-version re-tuning, and is more accurate. FlowGPU's approach is based on a common property of DL frameworks. DL frameworks have a modular performance-critical C++ backend as independent shared libraries. The backend takes over the memory management with an allocator (§2) with a set of stable interfaces to allocate and free memory. As shown in Figure 7, FlowGPU inserts a memory stub that intercepts these interfaces to maintain the active memory set. Our evaluation confirms near-framework-level accuracy (§5.2). FlowGPU further minimizes checkpoint size by waiting until active memory reaches its minimum (§2) before checkpointing (within 5% threshold), with a configurable maximum wait time (default 500 ms) after which it checkpoints immediately.

Fine-Grained Deduplication For distributed tasks, model parameters are replicated across GPUs and must be deduplicated. Singularity [17] deduplicates at the granularity of full GPU runtime memory blocks, but this is a rare match (§5.2); in practice, only *parts* of blocks are identical.

Therefore, FlowGPU performs deduplication at a finer granularity at the level of a fixed-size region (32 KB in the current implementation). If a memory block consists of multiple fixed-size regions, FlowGPU calculates a hash for each region. If two memory regions from different GPUs share the same hash, FlowGPU eliminates the duplicate. This deduplication is performed within the ghost process, which, during checkpointing, has access to GPU memory. As shown

in §5.2, this fine-grained deduplication reduces the checkpoint image size by 5× compared to Singularity.

4.3 Correctness Enhancements

Restoring GPU Memory To ensure correct recovery, the addresses of GPU memory regions in a task must remain identical across checkpointing and restore, as discussed in §3. Beyond GPU sharing, existing record-and-replay mechanisms fail to guarantee identical addresses even for single-task scenarios, since GPU runtimes do not guarantee deterministic address allocation. FlowGPU resolves this using VMM [4]: it intercepts GPU allocations and uses `cuMemAddressReserve`/`cuMemCreate`/`cuMemMap` to obtain virtual+physical regions, recording virtual addresses in a log. Upon restore, FlowGPU re-reserves the same virtual addresses before mapping, ensuring address identity.

Refined Pause Mechanism Distributed tasks (*e.g.*, synchronous training) require pausing all GPU instances for a consistent checkpoint. The coordinator signals all tasks; each pauses when the intercept library intercepts a GPU operation and sends an ack. However, Singularity's mechanism deadlocks when two GPUs use NCCL: GPU A calls blocking `send()` while GPU B is paused before its `receive()`, blocking both. FlowGPU resolves this by resuming all instances after 300 ms of incomplete pause, breaking the circular dependency.

4.4 Implementation Details

We implement FlowGPU with 19700+ lines of C++ code on CUDA runtime 12.4 and CUDA driver 550.90.07. We implement the optimization for identifying active memory (§4.2) on PyTorch 1.13.0 and TensorFlow 2.8. We package FlowGPU as dynamic libraries and inject them into the task using LD_PRELOAD. We focus on dynamic linking, since most tasks use it by default and generally support it.

Speed up memory copy with pinned memory. Following prior system-level mechanisms [17,10], FlowGPU uses pinned memory to speed up memory copy. During checkpointing, FlowGPU allocates pinned memory, copies GPU memory to the pinned memory, and then frees the pinned memory. Upon contiguous checkpointing (*i.e.*, checkpointing at fixed intervals for, *e.g.*, fault tolerance), to minimize the overhead of setting up pinned memory, FlowGPU keeps pinned memory until the user cancels checkpointing.

Handling Opaque Runtime Objects. GPU tasks rely on opaque runtime objects, such as CUDA context, CUDA streams, CUDA events for, *e.g.*, synchronization or workflow control. CUDA-related libraries such as cuBLAS and cuDNN introduce opaque runtime objects, including cuBLAS handles and cuDNN handles. These objects are opaque since the GPU runtime does not allow a task to directly read their state. Following Singularity and POS, FlowGPU uses record and replay to save/restore these objects. Specifically, FlowGPU intercepts and records GPU operations that create or modify opaque runtime objects (*e.g.*, `cudaSetDevice`), and then replays these operations upon recovery.

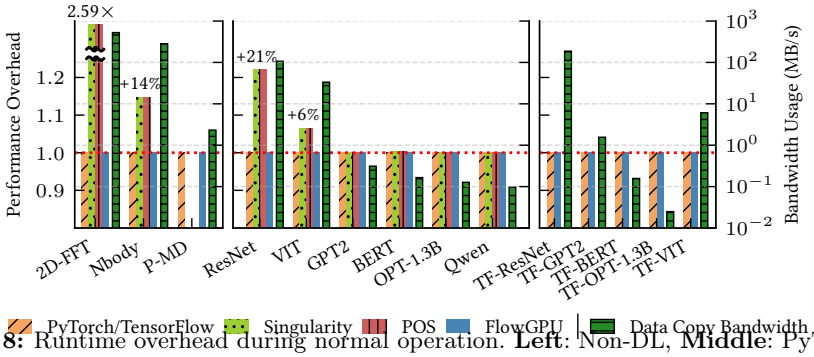

Fig. 8: Runtime overhead during normal operation. **Left**: Non-DL, **Middle**: PyTorch DL, **Right**: TensorFlow DL.

5 Experiment

5.1 Experimental Setup

Our evaluation machine has a 32-core Intel Xeon Gold 6462C, 500GB RAM, four 48GB NVIDIA L20 GPUs (PCIe 4.0, 24 GB/s, measured with CUDASamples [11]), running Ubuntu 22.04.5 LTS. We evaluate non-DL workloads (2D-FFT [8], Nbody [9], ProteinMD [6]) and DL workloads (ResNet50, ViT-L-16, GPT-2, BERT, Qwen2.5-1.5B, OPT-1.3B and OPT-6.7B) on single and 4-GPU configurations (OPT-6.7B across 4 GPUs). DL models run on both PyTorch and TensorFlow (except Qwen).

Baseline. Singularity [17] and POS [10] are the system-level baselines; PyTorch/TensorFlow serve as reference points. Singularity was not open-sourced so we re-implemented it, applying all applicable optimizations and using an oracle for its active-memory prediction (best case). POS does not support ProteinMD, TensorFlow tasks, or distributed tasks.

5.2 Checkpointing and Restore Performance

Runtime Overhead During Normal Operation Figure 8 shows runtime overhead with a single GPU during normal operation. FlowGPU incurs no overhead since it forgoes virtualization, enabling direct GPU access. Singularity and POS still require API forwarding, incurring up to 2.6× overhead from IPC.

C/R Time of Tasks Operating on a Single GPU Checkpointing time. Figure 9 shows pause time during checkpointing. For DL tasks, FlowGPU reduces pause time by 6.2× to 15× over POS and up to 10.4× over Singularity, since they fail to accurately identify active memory. For tasks with a small image (ResNet50, GPT-2, ViT), FlowGPU incurs longer pause time since CRIU is the bottleneck; PyTorch/TensorFlow skip non-GPU state by reinitializing upon restore. For other tasks (ViT, OPT-1.3B, Qwen), FlowGPU matches or outperforms PyTorch/TensorFlow since it uses pinned memory (§4.4).

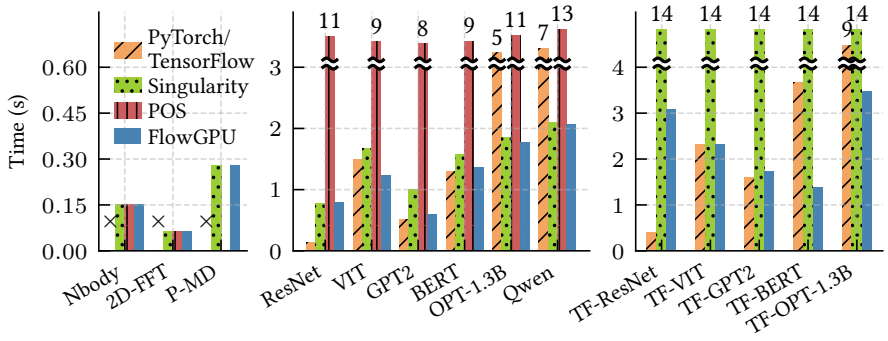

Fig. 9: Pause time during checkpointing for tasks operating on a single GPU. **Left**: Non-DL, **Middle**: PyTorch DL, **Right**: TensorFlow DL.

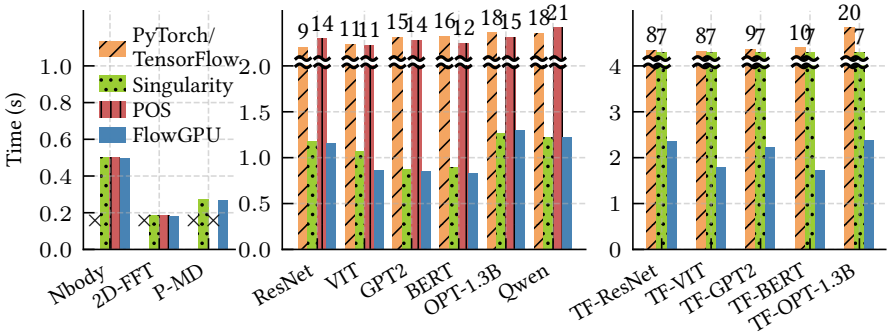

Fig. 10: Restore time for tasks operating on a single GPU. **Left**: Non-DL, **Middle**: PyTorch DL, **Right**: TensorFlow DL.

Restore time. Figure 10 shows restore time. FlowGPU reduces it by 12× to 18× over POS and up to 4.1× over Singularity, due to active memory identification. PyTorch/TensorFlow suffer from long restore time as they reinitialize the entire task (*e.g.*, reload datasets and models).

Breakdown. Identifying active memory reduces checkpointing time by 3.3× to 14.8× and restore time by 2.1× to 4.7× (Figures 11, 12). Parallel C/R via ghost process further reduces checkpointing by up to 1.9× and restore by 1.3× to 2×.

Checkpoint image size. FlowGPU reduces checkpoint image size by up to 137× vs. Singularity and 6.1× to 137× vs. POS, achieving near-framework-level sizes (Figure 13).

C/R Time of Distributed Tasks Figures 14 to 16 show results for 4-GPU tasks (TensorFlow omitted; similar to PyTorch). Compared to Singularity, FlowGPU reduces checkpointing time by up to 2.4× and restore time by up to 1.8×, due to smaller checkpoint images. Fine-grained deduplication (§4.2) reduces checkpointing by 1.1× to 1.6× and restore by 1.1× to 1.7×; image size further reduces by 1.2× to 5× vs. coarse-grained.

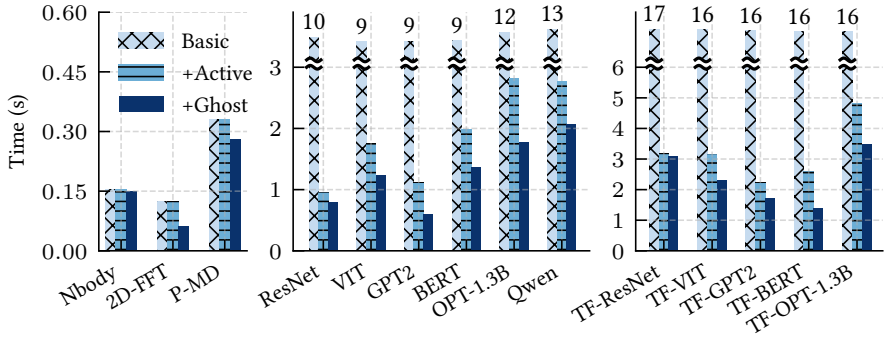

Fig. 11: How each optimization reduces the checkpointing time for tasks operating on a single GPU. **Left**: Non-DL, **Middle**: PyTorch DL, **Right**: TensorFlow DL. *active*: identify active memory. *ghost*: ghost process.

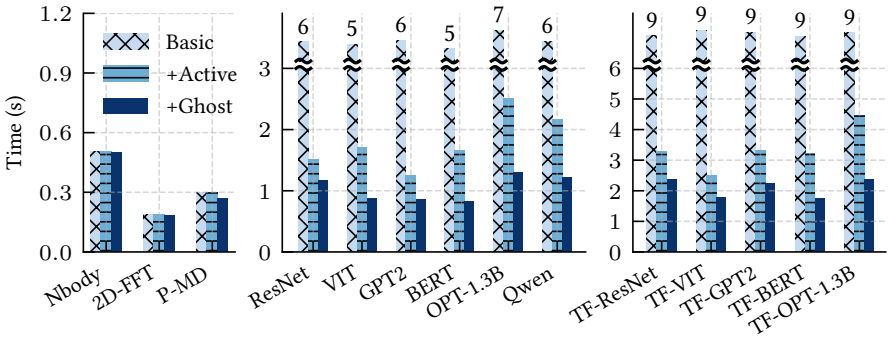

Fig. 12: How each optimization reduces the restore time for tasks operating on a single GPU. **Left**: Non-DL, **Middle**: PyTorch DL, **Right**: TensorFlow DL. *active*: identify active memory. *ghost*: ghost process.

5.3 End-to-End Use Cases

Task Migration Figure 18 shows migration time (checkpointing, image transmission, restore). FlowGPU outperforms Singularity by up to 2.1× and PyTorch by 1.7× to 4.5×. FlowGPU outperforms Singularity via smaller checkpoint images; it outperforms PyTorch via faster restore (§5.2).

6 Related Work

Section 3 already presents the most relevant work to FlowGPU. We would like to further note that FlowGPU is orthogonal to POS [10]. POS realizes a soft copy-on-write mechanism to reduce pause time during checkpointing. This is achieved during checkpointing by replacing GPU memory regions that a task operates on with fresh memory regions. Such a mechanism is orthogonal to FlowGPU; FlowGPU benefits from this design, while POS benefits from other design aspects in FlowGPU as well. Moreover, CriuGPU[3] shares a motivation similar to ours.

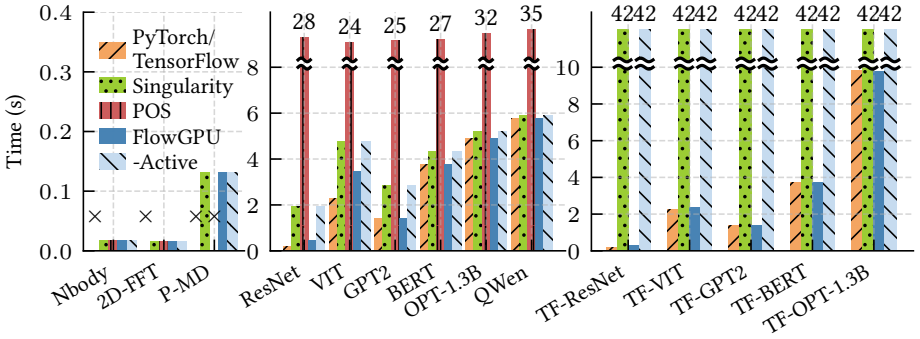

Fig. 13: Checkpoint image size (single GPU). Lower is better. -*active*: FlowGPU without active memory identification. **Left**: Non-DL, **Middle**: PyTorch DL, **Right**: TensorFlow DL.

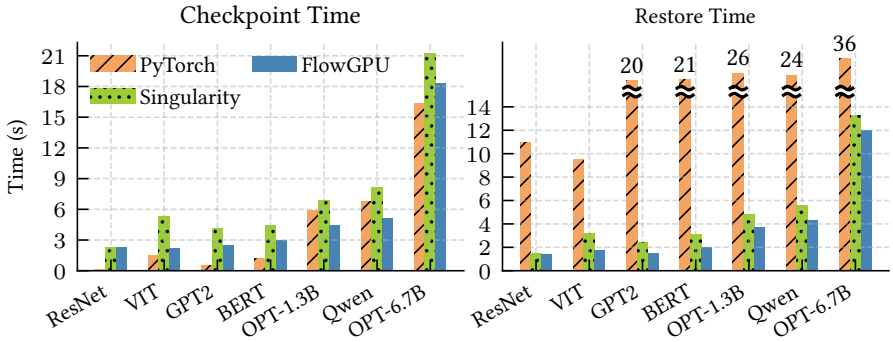

Fig. 14: Checkpointing and restore time for distributed tasks. POS does not support distributed tasks.

However, it relies on NVIDIA's closed-source binary tools, forgoing potential optimizations, such as active memory, parallel checkpointing and restore and cannot support multi-GPU tasks with NCCL.

7 Conclusion

This paper presents FlowGPU, a system-level GPU checkpointing/restore mechanism that advances the state of the art by overcoming key limitations of prior mechanisms, including runtime overhead, long checkpointing/restore time, compromised correctness, and violating task transparency.

The key insight behind FlowGPU is that these limitations stem from coupling checkpointing/restore with a legacy virtualization technique: API forwarding, Therefore, FlowGPU decouples these two aspects by proposing two key techniques: 1) per-task API interception and 2) the ghost process. These two techniques enable FlowGPU to maintain the benefits of API forwarding and overcome the limitations of API forwarding with the decoupling. FlowGPU further comes with various techniques to improve correctness and performance, including a

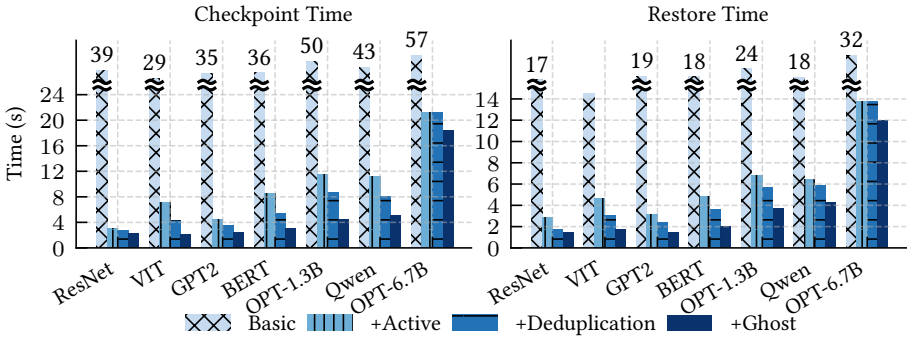

Fig. 15: How each optimization reduces the checkpointing time for tasks operating on multiple GPUs. *active*: identify active memory. *ghost*: ghost process.

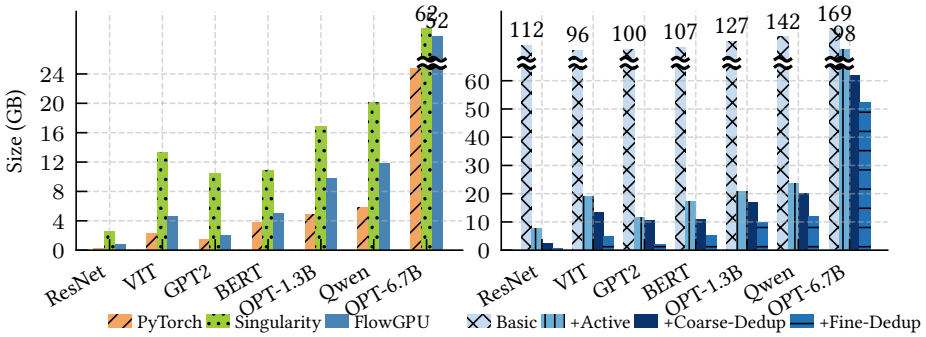

Fig. 16: Checkpoint image size (**Left**) and breakdown (**Right**) for distributed tasks.

key technique to identify active memory, which is more stable, generalized, and accurate than the prior proposal in Singularity. Our evaluation shows that FlowGPU incurs no overhead during normal operation, produces a minimal checkpoint image, and, upon task migration and fault tolerance, outperforms prior mechanisms by 4.5×.

References

1. AMD ROCm: (2024), https://rocm.docs.amd.com/en/latest/what-is-rocm.html
2. Cornell, W.D., Cieplak, P., Bayly, C.I., Gould, I.R., Merz, K.M., Ferguson, D.M., Spellmeyer, D.C., Fox, T., Caldwell, J.W., Kollman, P.A.: A second generation force field for the simulation of proteins, nucleic acids, and organic molecules. Journal of the American Chemical Society **117**(19), 5179–5197 (1995). https://doi.org/10.1021/ja00124a002, https://doi.org/10.1021/ja00124a002
3. CRIU: (2025), https://criu.org/Main_Page
4. CUDA VMM: (2024), https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html
5. Duato, J., Peña, A.J., Silla, F., Mayo, R., Quintana-Ortí, E.S.: rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In: Smari, W.W.,

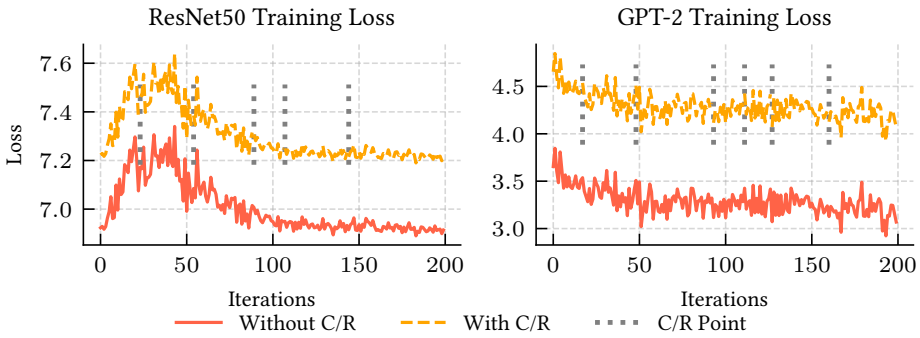

Fig. 17: Loss curves under checkpointing/restore vs. baseline.

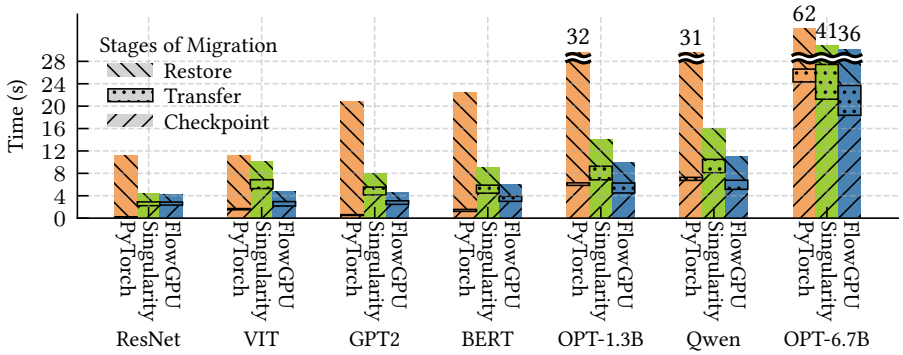

Fig. 18: Migration time for different tasks.

McIntire, J.P. (eds.) HPCS 2010. pp. 224–231. IEEE (2010). https://doi.org/10.1109/HPCS.2010.5547126, https://doi.org/10.1109/HPCS.2010.5547126

6. Eastman, P., Pande, V.: Openmm: A hardware-independent framework for molecular simulations. Computing in Science and Engineering **12**(4), 34–39 (2010). https://doi.org/10.1109/MCSE.2010.27
7. Eiling, N., Baude, J., Lankes, S., Monti, A.: Cricket: A virtualization layer for distributed execution of CUDA applications with checkpoint/restart support. Concurr. Comput. Pract. Exp. **34**(14) (2022). https://doi.org/10.1002/cpe.6474, https://doi.org/10.1002/cpe.6474
8. Frigo, M., Johnson, S.G.: FFTW: an adaptive software architecture for the FFT. In: Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98, Seattle, Washington, USA, May 12-15, 1998. pp. 1381–1384. IEEE (1998). https://doi.org/10.1109/ICASSP.1998.681704, https://doi.org/10.1109/ICASSP.1998.681704
9. Heggie, D., Hut, P.: INTRODUCTIONS, p. 1–2. Cambridge University Press (2003)
10. Huang, Z., Wei, X., Hao, Y., Chen, R., Han, M., Gu, J., Chen, H.: Parallelgpuos: A concurrent os-level gpu checkpoint and restore system using validated speculation (2024), https://arxiv.org/abs/2405.12079
11. NVIDIA CUDA samples: (2024), https://github.com/NVIDIA/cuda-samples
12. Nvidia MIG: (2025), https://www.nvidia.com/en-us/technologies/multi-instance-gpu

13. Nvidia MPS: (2025), https://docs.nvidia.com/deploy/mps/index.html
14. Prades, J., Silla, F.: Gpu-job migration: The rcuda case. IEEE Trans. Parallel Distributed Syst. **30**(12), 2718–2729 (2019). https://doi.org/10.1109/TPDS.2019.2924433, https://doi.org/10.1109/TPDS.2019.2924433
15. Shahzad, F., Thies, J., Kreutzer, M., Zeiser, T., Hager, G., Wellein, G.: CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance. IEEE Trans. Parallel Distributed Syst. **30**(3), 501–514 (2019). https://doi.org/10.1109/TPDS.2018.2866794, https://doi.org/10.1109/TPDS.2018.2866794
16. Shi, L., Chen, H., Sun, J., Li, K.: vcuda: Gpu-accelerated high-performance computing in virtual machines. IEEE Trans. Computers **61**(6), 804–816 (2012)
17. Shukla, D., Sivathanu, M., Viswanatha, S., Gulavani, B.S., Nehme, R., Agrawal, A., Chen, C., Kwatra, N., Ramjee, R., Sharma, P., Katiyar, A., Modi, V., Sharma, V., Singh, A., Singhal, S., Welankar, K., Xun, L., Anupindi, R., Elangovan, K., Rahman, H., Lin, Z., Seetharaman, R., Xu, C., Ailijiang, E., Krishnappa, S., Russinovich, M.: Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads. CoRR **abs/2202.07848** (2022), https://arxiv.org/abs/2202.07848
18. Tian, K., Dong, Y., Cowperthwaite, D.: A full GPU virtualization solution with mediated pass-through. In: Gibson, G., Zeldovich, N. (eds.) Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014. pp. 121–132 (2014)
19. Unified Memory in CUDA: (2013), https://developer.nvidia.com/blog/unified-memory-in-cuda-6/
20. Wang, Z., Jia, Z., Zheng, S., Zhang, Z., Fu, X., Ng, T.S.E., Wang, Y.: Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In: Proceedings of the 29th Symposium on Operating Systems Principles. pp. 364–381. SOSP '23 (2023)
21. Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., Zhou, L.: Gandiva: Introspective cluster scheduling for deep learning. In: Arpaci-Dusseau, A.C., Voelker, G. (eds.) 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018. pp. 595–610. USENIX Association (2018), https://www.usenix.org/conference/osdi18/presentation/xiao
22. Xiao, W., Ren, S., Li, Y., Zhang, Y., Hou, P., Li, Z., Feng, Y., Lin, W., Jia, Y.: AntMan: Dynamic scaling on GPU clusters for deep learning. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). pp. 533–548. USENIX Association (Nov 2020), https://www.usenix.org/conference/osdi20/presentation/xiao
23. Ye, Z., Sun, P., Gao, W., Zhang, T., Wang, X., Yan, S., Luo, Y.: Astraea: A fair deep learning scheduler for multi-tenant GPU clusters. IEEE Trans. Parallel Distributed Syst. **33**(11), 2781–2793 (2022). https://doi.org/10.1109/TPDS.2021.3136245, https://doi.org/10.1109/TPDS.2021.3136245