

# bpftime-super: A GPU observability tool

Yiwei Yang  
UC Santa Cruz  
USA  
yzhen165@ucsc.edu

Yusheng Zheng  
UC Santa Cruz  
USA  
yyang363@ucsc.edu

Tong Yu  
eunomia-bpf Community  
China  
yt.xyxx@gmail.com

Andrew Quinn  
UC Santa Cruz  
USA  
aquinn@ucsc.edu

## ABSTRACT

GPU observability is crucial for ensuring high performance in large language model (LLM) workloads and other computationally intensive applications in high-performance computing (HPC) environments. Large-scale HPC deployments depend on GPU acceleration to manage data-intensive computations and vast model parameters. By combining real-time GPU telemetry, dynamic instrumentation, and automated performance analysis, we can pinpoint bottlenecks at multiple layers—ranging from kernel execution and memory transfers to heterogeneous compute orchestration. This enhanced visibility not only streamlines the process of identifying and addressing performance constraints, but also facilitates more efficient hardware utilization and resource planning, ultimately reducing costs and accelerating time-to-solution using the abstraction of eBPF uprobe. Furthermore, these eBPF-powered programable observability techniques can be seamlessly integrated into both AI-driven pipelines (e.g., training and inference of LLMs) and traditional HPC workflows such as large-scale simulations, climate modeling, and computational fluid dynamics. The resulting holistic view of GPU performance empowers researchers and engineers to swiftly optimize code paths, eliminate inefficiencies, and ensure that HPC resources are deployed to their fullest potential.

## ACM Reference Format:

Yiwei Yang, Tong Yu, Yusheng Zheng, and Andrew Quinn. 2025. bpftime-super: A GPU observability tool. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

The rapid growth of data-intensive and computationally demanding workloads—ranging from large-scale machine learning and high-performance computing (HPC) simulations[12, 14, 21] to graphics rendering[13] and Large Language Models[6]—has increasingly relied on GPU acceleration to achieve desired performance and

throughput. As these GPU-powered systems continue to scale, precise observability and low-overhead instrumentation become essential to diagnose performance bottlenecks, ensure efficient resource usage, and detect anomalous behavior. In parallel, the Linux extended Berkeley Packet Filter (eBPF) ecosystem has gained prominence as a powerful observability framework, capable of running sandboxed programs in kernel space without requiring intrusive instrumentation or kernel modifications. While eBPF has traditionally been used for network analytics[5] and system profiling[17, 20], its capabilities now extend into GPU-centric environments, enabling a new class of performance tooling and runtime optimizations.

Despite the promise offered by eBPF for GPU observability, current approaches still face several challenges. First, GPUs introduce extra layers of complexity, such as asynchronous execution flows and specialized memory subsystems, which can be difficult to instrument without incurring high overhead or disrupting critical paths. Second, eBPF itself is constrained by the kernel boundary, often leading to frequent context switches or limited visibility when dealing with fast-evolving GPU workloads in user space. Consequently, bridging the gap between GPU execution models and the eBPF framework requires novel design and engineering strategies that can gather detailed performance metrics at runtime while maintaining safety, modularity, and minimal impact on workloads.

In sum, our contributions are:

- We introduce a novel approach that tightly integrates eBPF instrumentation with GPU execution, offering a low-overhead, high-granularity observability layer for modern HPC and AI workloads.
- We describe the design and implementation of bpftime-super, our prototype system that leverages both kernel-level and user-space hooks to facilitate real-time performance analysis, seamlessly integrating with existing eBPF infrastructures and GPU APIs.

The rest of the paper proceeds as follows. We first discuss prior work on eBPF, GPU observability, and their applications in large-scale systems in §3. We then present the details of bpftime-super's design and implementation in §5, followed by an in-depth evaluation in §7. Finally, we conclude with a summary of our findings and future directions in §8.

## 2 BACKGROUND

In this section, we will delve into the foundational concepts that underpin our discussion in the subsequent sections.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*Conference'17, July 2017, Washington, DC, USA*

© 2025 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 2.1 bpftime

How kernel-based eBPF operations, particularly uprobes, suffer from significant overhead due to frequent context switches. By moving these operations into user space, bpftime removes the need for multiple context switches, providing up to a 10x performance boost. This approach also improves configurability, reduces security risks associated with kernel-level privileges, and allows for efficient syscall hooking within processes. Furthermore, bpftime[19] supports user-space eBPF Maps in shared memory and remains compatible with existing eBPF development tools such as clang and libbpf, even providing CO-RE support through BTF. Ultimately, bpftime showcases a more streamlined, flexible, and secure runtime for eBPF in user space. Historically, extended Berkeley Packet Filter (eBPF) programs have been largely confined to the kernel space due to their close integration with low-level system operations like networking, tracing, and security enforcement. The uprobe mechanism in eBPF permits instrumentation at user-level function entry or exit points, but when managed by the kernel, it often incurs performance penalties caused by the overhead of switching execution contexts between user space and kernel space.

These overheads stem from the additional steps the operating system must take each time a probe fires, including saving processor states and transferring control to the kernel. Over multiple probes or high-event scenarios, this latency can become pronounced, ultimately throttling system throughput and the speed of data collection. By moving these uprobes and other hooks to user space instead, the bpftime framework significantly reduces these transitions, accelerating tracing operations without sacrificing functionality. This user-space approach also bypasses the need for certain kernel-level permissions, diminishing the kernel's attack surface and lowering barriers for developers and operators who wish to leverage eBPF without full administrative privileges. Moreover, the bpftime system employs binary rewriting to achieve safe and efficient hooking of functions, thus enabling advanced eBPF-based observability for processes without requiring reboots or code recompiles. With support for shared memory eBPF Maps, bpftime extends its capabilities to multi-process scenarios and control-plane interactions, all while remaining compatible with established eBPF toolchains and preserving Compile Once – Run Everywhere (CO-RE) functionality.

## 2.2 PTX JIT

PTX (Parallel Thread Execution) serves as an intermediate representation for NVIDIA GPUs, offering a device-independent layer for parallel computations, which is faster than NVBit[10, 15] tools. PTX Just-In-Time (JIT), used in [8, 9] compilation allows developers to convert PTX code into GPU-specific machine instructions on the fly, enabling them to adapt and optimize their programs in real-time. This capability extends beyond simple runtime compilation: it facilitates dynamic instrumentation by letting users intercept and modify PTX instructions before they are turned into final machine code (SASS). In doing so, developers can embed profiling counters, memory checks, or custom analytics directly into the GPU kernel with minimal overhead. Because PTX JIT operates at this higher-level IR—rather than at the raw binary level as in frameworks like NVBit—it often delivers superior performance, as it bypasses the complexity of rewriting fully compiled machine instructions.

This leads to highly optimized code tailored to the specific GPU architecture, capitalizing on features such as register usage and warp scheduling. Additionally, PTX JIT integrates naturally with existing NVIDIA toolchains (including NVCC and NVRTC), letting users refine or generate specialized kernels without leaving familiar workflows, and removing the need for time-consuming offline recompilation every time a kernel needs to change or be instrumented.

## 3 RELATED WORK

Meta's AI observability infrastructure[2] and many other tools[1, 3, 4] illustrates a multi-layered approach to monitoring, performance introspection, and fleet-wide optimization for large-scale AI workloads. At the foundational level, Dynolog—a distributed telemetry daemon—collects bare-metal metrics on CPU, GPU, and other specialized ASICs across Meta's large, heterogeneous data center fleet. These low-overhead measurements (e.g., device power, utilization, FLOPs/sec) act as key indicators of efficiency, helping identify whether hardware resources are used effectively. To address device-specific complexities—such as Nvidia GPU's limited hardware counters—Meta combines heuristic estimation (via DCGM) and selective, more precise tracing (via CUPTI). Building atop this telemetry, Meta employs end-to-end profiling tools like PyTorch Profiler, Kineto, and Strobilight/BPF to capture fine-grained performance data. Through Strobilight's integration with BPF, users can attach uprobes to CUDA and PyTorch memory operations without instrumentation, supporting real-time, secure, and low-overhead profiling. This technique helps track GPU memory leaks, bottlenecks, and device-to-host interaction patterns.

Because Meta runs tens of thousands of GPU jobs daily, manual analysis is impractical. To address scale, Meta Performance Profiling/Analysis Platform automates trace collection, stitching, and analysis across multiple hosts and profilers. Its intelligence layer highlights common inefficiencies—such as data starvation, suboptimal kernel usage, or imbalanced workloads—allowing AI developers to optimize code paths or reconfigure hardware usage. Additionally, by aggregating profile data over time, the platform uncovers long-term trends, such as sudden regressions in kernel duration or patterns in resource consumption that suggest deeper tuning opportunities. Finally, fleet-level resource attribution dashboards break down GPU hours or FLOPs usage by model, user, or product group, ensuring that optimization efforts target the largest consumers of compute time. This data can inform priorities between framework-level improvements (e.g., refining PyTorch kernels) and model-specific optimizations (e.g., restructuring embedding tables or adjusting hyperparameters). Collectively, these workflows and tools from Meta's observability stack demonstrate how systematic performance monitoring, automated data analysis, and a layered telemetry architecture can enable large-scale AI system efficiency, aligning closely with related research on persistent kernels, dynamic instrumentation, and just-in-time optimization strategies in data-intensive computing environments.

## 4 BPFTIME-SUPER DESIGN

Figure 1 illustrates a unified user-space eBPF environment that integrates with existing eBPF toolchains, CUDA-based components, and

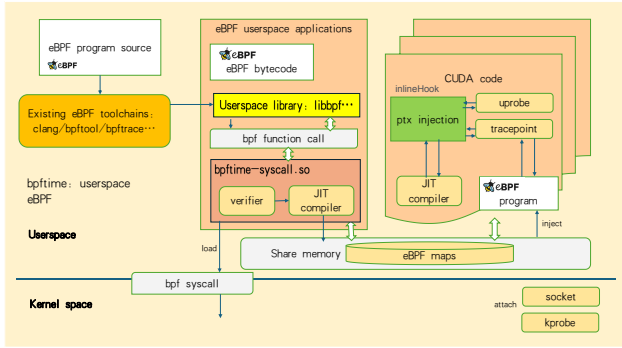


Figure 1: System Design

the traditional kernel-level `bpf syscall`. Starting from the top-left, an eBPF program source is compiled using standard utilities such as clang, bpftool, or bpftrace, which emit eBPF bytecode. That bytecode then flows into the user-space eBPF runtime, where it is loaded and possibly instrumented or transformed before execution.

In the center section, there is an eBPF userspace application layer that references a userspace library (for example, libbpf). This library allows interaction with the eBPF function APIs, enabling the application to load or call eBPF functions at run time. A critical component in this layer is `bptime-syscall.so`, which provides a user-space verifier and JIT compiler. These two elements perform tasks typically associated with kernel-based eBPF, such as verifying the safety of programs and translating bytecode into native instructions, but they do so without requiring kernel-level privileges. The result is that user-space eBPF programs gain flexibility and portability while still benefiting from familiar features like verification and just-in-time compilation.

Below the userspace, there is a `bpf syscall` interface that can optionally connect this user-space layer to the kernel eBPF subsystem. While the diagram highlights a user-space focused workflow, the presence of `bpf syscall` ensures that eBPF bytecode or maps can be loaded into the kernel if the system design requires deeper kernel integration or attachment to traditional kernel hooks.

Toward the lower-middle portion, there is a shared memory region used to hold eBPF maps. This shared memory is accessible not only to the user-space components but also to the CUDA code indicated on the right. By registering these maps as pinned or pinned-like memory, both GPU and CPU contexts can observe and update the same data structures, enabling dynamic exchange of events and statistics without the overhead of repeated copying. This arrangement is beneficial for high-throughput instrumentation or data-collection scenarios.

On the right, the figure shows a CUDA environment where user-space eBPF can be extended to generate or inject PTX code. A JIT compiler in this context transforms eBPF bytecode into PTX or other GPU-compatible formats, and then these GPU-extended eBPF features can attach to hooks such as `uprobe` or `tracepoints` within the GPU domain. The label “inlineHook” in the diagram indicates the ability to insert instrumentation at specific points inside CUDA code, letting developers trace GPU-side operations in a manner similar to how eBPF operates in the kernel for CPU

processes. The eBPF program injected here can interact with the same eBPF maps shared by the rest of the system, ensuring that monitoring, collection, or filtering logic remains consistent across CPU and GPU boundaries.

Another important aspect highlighted by the figure is that the system can accommodate various “attach” targets. While kernel eBPF commonly supports `kprobe`, `socket`, and `tracepoints`, the user-space variant shown here can attach to higher-level events, including application-level function calls or GPU hooks, according to the user’s needs. This unified approach establishes a single user-space eBPF solution that can be extended into GPUs while optionally bridging back into the kernel through the `bpf syscall`, depending on the deployment requirements.

In terms of insights, the design improves the accessibility of eBPF by moving verification and compilation to user space, thus making it easier to develop, debug, and deploy eBPF-based solutions without requiring privileged kernel changes. At the same time, it allows tight integration with GPU resources by leveraging PTX injection, enabling on-device tracing, profiling, or custom event handling. The shared memory approach guarantees low-latency communication between user-space applications, GPU code, and eBPF maps, which is essential for real-time or performance-critical instrumentation. Overall, this architecture provides a flexible environment that extends the traditional power of eBPF into new domains while preserving compatibility with the standard tooling and ecosystem.

## 5 BPFTIME-SUPER IMPLEMENTATION

In this section, we will talk about the bpftime-super implementation.

### 5.1 Persistent Kernels

A key component of our bpftime-super approach involves leveraging persistent kernels, also referred to as persistent threads. Instead of repeatedly launching brief kernels to handle each incoming task, a single long-lived kernel remains resident on the GPU and processes multiple work units over time. This model helps minimize kernel launch overhead and offers improved load balancing for dynamic or streaming workloads. Prior work, such as Gupta et al. [7] and Kim et al. [11], has demonstrated the efficacy of persistent kernels in reducing latency and enabling continuous processing, thereby eliminating costly device-to-host (D2H) and host-to-device (H2D) operations between launches. Recent studies, like Zhang et al. [18], highlight additional opportunities to cache partial outputs in registers or shared memory within a single kernel lifespan, which further reduces traffic to device memory.

Despite these advantages, persistent kernels face limitations when applied to small-batch workloads, such as smaller neural network (NN) tasks or latency-sensitive routines. One strategy is to restrict the number of streaming multiprocessors (SMs) allocated to each kernel so that multiple persistent kernels can run concurrently. Although Multi-Process Service (MPS) on NVIDIA GPUs provides some level of multiplexing, the architectural constraints of many GPUs mean that context switching is still less flexible compared to platforms like AMD. By deploying persistent kernels, we remove much of the overhead from repeated launches, but we must also

content with Just-In-Time (JIT) compilation time. When numerous small kernels demand rapid turnarounds, cumulative JIT overhead can negatively affect both latency and throughput. We address these issues by merging kernel persistence with an optimized JIT procedure, aiming to ensure that the combined overhead stays minimal even under real-time constraints.

## 5.2 Shared Memory

In our implementation, we use `boost::managed_shared_memory` to facilitate transparent data exchange between different processes. This shared memory region holds crucial data structures—such as eBPF maps, input batches, or intermediate states—that both the CPU and GPU kernels can access concurrently. By designing this memory with direct GPU compatibility in mind, we reduce the need to marshal data back and forth through costly copy operations. Furthermore, we anticipate the potential of next-generation hardware like Grace Hopper architectures and CXL memory pools, which promise to unify large memory regions with low-latency interconnects. Such advances can seamlessly extend the capacity and performance of `managed_shared_memory`, allowing persistent kernels to operate over broader data sets without incurring frequent synchronization or replication overhead.

## 5.3 Synchronization

Given that multiple CPU threads and persistent GPU kernels share data structures, synchronization must be carefully managed to avoid race conditions and ensure memory consistency. We rely on several mechanisms that operate at different levels. On the CPU side, standard lock-based or lock-free data structures (e.g., using `std::atomic` or Boost interprocess primitives) ensure that host processes do not introduce inconsistent states in shared regions. Within the GPU, we use CUDA atomic operations and device-wide memory fences to serialize updates to global or shared memory. This combined strategy, supported by a well-defined data layout in `managed_shared_memory`, ensures that asynchronous updates remain visible to all participants. For frequent small-scale synchronization events, we minimize warp-level conflicts by having only a small subset of threads perform atomic operations, whereas the rest of the kernel continues to process data in parallel.

## 5.4 PTX Generation & Injection

Our system performs eBPF or application-specific code transformations into PTX to enable GPU-side instrumentation and computation. When user-space or eBPF logic requests that certain functions be offloaded or traced on the GPU, our JIT compiler generates specialized PTX snippets. These are then dynamically injected into the running persistent kernel through device-resident hooks. We refer to this technique as “PTX injection,” which allows us to intercept or insert instructions without tearing down the existing kernel context. This approach helps reduce overhead compared to stopping the kernel, recompiling the entire codebase, and relaunching it.

Once injected, the PTX code can interoperate with eBPF maps or other shared memory buffers, granting it direct access to the same data structures used by the CPU side. Combined with persistent kernel execution, PTX injection creates a flexible environment for GPU instrumentation, tracepoints, and user-defined logic that can

adapt in real time as new workloads arrive or instrumentation needs change. By carefully managing memory barriers and code patching points, we ensure that newly injected PTX interacts cleanly with the previously running kernel components.

In sum, *bpftime-super* integrates persistent kernels, shared memory, robust synchronization, and PTX-based offloading to deliver a low-overhead, adaptive GPU computation model. Persistent kernels eliminate repeated launches, shared memory reduces copying costs, synchronization primitives guard against data hazards, and PTX injection enables fine-grained dynamic instrumentation. Looking ahead, improved hardware features in Grace Hopper and CXL memory pools are poised to amplify these benefits, further minimizing overhead and expanding the scope of persistent, user-extended GPU workloads.

## 6 USE CASE

### 6.1 GPU Memory Observer and CXL.mem Simulator

To achieve fine-grained observability of GPU memory behavior, *bpftime-super* instruments low-level memory operations—specifically the load (LD) and store (ST) instructions on NVIDIA GPU architectures. These instructions form the core of the GPU’s memory access model, and by capturing each LDR and STR operation, our tool provides unprecedented insight into memory bandwidth utilization, access patterns, and contention that prior work could not achieve.

We leverage eBPF-based instrumentation to hook into the GPU kernel events corresponding to LD and ST operations. By placing probes at key functions in the CUDA driver stack, we capture metadata such as the memory address, access size, and timestamp for each operation. This detailed logging enables the computation of real-time metrics such as memory bandwidth. For example, the bandwidth over a time window  $T_{\text{window}}$  is given by:

$$BW = \frac{\sum_{i=1}^N \text{Bytes}_i}{T_{\text{window}}}, \quad (1)$$

where  $\text{Bytes}_i$  is the number of bytes transferred in the  $i$ -th operation, and  $N$  is the total number of memory operations observed in that window. In addition, the aggregated data reveals access patterns (e.g., coalesced versus scattered accesses) and contention levels when multiple streaming multiprocessors (SMs) perform memory operations concurrently.

Beyond observation, we simulate a second tier of memory using CXL.mem by introducing dynamic latency insertion into the observed memory operations. The objective is to emulate the behavior of CXL-attached memory, which typically exhibits higher latency and lower bandwidth compared to local GPU memory. In our model, the effective access latency for a memory operation is expressed as:

$$T_{\text{access}}^{(\text{CXL})} = T_{\text{access}}^{(\text{local})} + \Delta t(\text{pattern}, BW_{\text{curr}}), \quad (2)$$

where  $T_{\text{access}}^{(\text{local})}$  is the native latency for accessing local GPU memory, and  $\Delta t(\text{pattern}, BW_{\text{curr}})$  is a dynamic delay function that depends on the observed access pattern (e.g., sequential vs. random) and the current memory bandwidth utilization  $BW_{\text{curr}}$ . For instance, sequential (streaming) accesses may incur a smaller additional delay

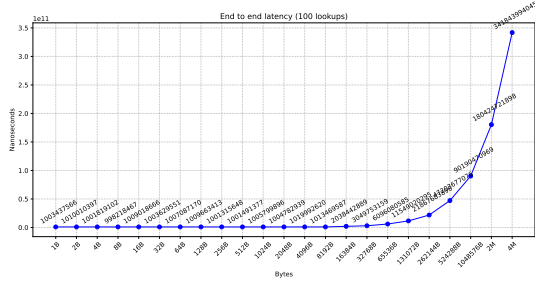


Figure 2: Runtime Overhead

compared to random accesses, while high contention (i.e., when  $BW_{curr}$  nears the CXL link capacity) further increases  $\Delta t$ .

This dynamic delay model enables the emulation of a two-tier memory system where data may reside in either fast local memory or slower CXL-attached memory. Our experiments compare the performance of GPU kernels with and without the CXL.mem simulation enabled. Memory-bound kernels, in particular, demonstrate a significant increase in effective latency—up to 2–3 $\times$ —when subjected to the additional delays, thus highlighting the trade-offs between increased memory capacity and reduced access speed. These findings align with prior work on CXL.mem systems [16].

## 6.2 LLM CPU-GPU collaborative caching

Large Language Models (LLMs) push the limits of GPU memory, often necessitating a collaborative caching strategy between CPU and GPU to manage their massive state. Critical to performance is deciding which data resides in fast GPU memory versus what is held in CPU memory and fetched on demand. An effective caching system must intelligently handle cache evictions and prefetching to minimize data transfers. Bpftime-super aids in this process by providing detailed observability into cache behavior and memory access patterns across the CPU-GPU boundary. By tracing events such as page faults, DMA copy calls, or unified memory migrations, the tool reveals which model parameters or activations are accessed frequently and which remain idle. This information enables engineers to fine-tune cache policies—for example, ensuring that the most-accessed weights stay on the GPU, while less frequently used data is staged in CPU memory until needed.

Cache eviction policies are crucial for managing limited GPU memory. A common strategy is **Least Recently Used (LRU)**, where, upon reaching capacity, the data that has not been used for the longest time is evicted. LRU is simple and effective, and many AI systems adopt it by default for GPU memory caches. For example, offloading frameworks may move the least recently accessed layers or expert networks to CPU memory first. An alternative is **Frequency-Based Eviction (LFU)**, which retains items that are accessed most frequently. In scenarios where a small subset of model weights or expert networks dominates access patterns, frequency-based eviction can outperform LRU. In addition, static caching approaches, where certain data is *pinned* in GPU memory based on domain knowledge (e.g., keeping the most critical layers on GPU), are often combined with dynamic policies to yield optimal results. A hybrid policy might, for instance, evict the oldest

conversation context in a multi-turn dialogue system while maintaining an LRU strategy for overall session data. Bpftime-super’s insights into cache behavior help verify the effectiveness of these strategies by monitoring cache hit rates and identifying excessive data transfers.

Beyond eviction, shared memory strategies between CPU and GPU are vital for performance. NVIDIA’s **Unified Memory** creates a single address space spanning both CPU and GPU, automatically migrating pages on demand via page-fault mechanisms. Although unified memory simplifies programming, its performance depends on the observed access patterns. bpftime-super can highlight cases where frequent page faults occur for the same data, suggesting that manual caching (such as using pinned host memory) might be more efficient. Pinned memory enables faster DMA transfers and technologies like Direct Host Access allow GPU kernels to read from CPU memory directly over PCIe without waiting for full data copies. Such techniques can overlap data transfers with computation to reduce latency.

Additional shared memory optimizations include the use of zero-copy buffers and stream-ordered prefetching. Zero-copy buffers allow GPU threads to access host memory directly, reducing redundant data movements, while prefetching proactively copies data to the GPU ahead of its use. In real-world use cases, such as model offloading in systems like DeepSpeed Zero-Inference or caching key/value matrices in transformer-based models, a well-tuned collaborative caching system can maintain high cache hit rates and reduce transfer overhead. The observability provided by bpftime-super ensures that caching strategies are working as expected, facilitating rapid training iterations and lower inference latency in large-scale LLM deployments.

Figure 2, as a micro-benchmark of showcasing how much effective memory caching is. As shown with the value size of the hashmap, the performance degraded from 32KB. This is because it can not fit in the eBPF stack and must be stored in a global map.

## 7 EVALUATION

We evaluate the system on a dual-socket Intel Xeon E5-2697- v2 processor (48 cores, 2.7 Ghz, 30 MB LLC) with 256 GB DDR3 memory installed with P40 Graphics card installed.

### 7.1 bpftime-super Performance

We evaluated bpftime-super by comparing its end-to-end performance against a native implementation of the same workloads. In this comparison, both systems utilized the same input data and computational tasks, but the native version excluded the overhead of persistent kernels, shared memory synchronization, and PTX injection. Our measurements focused on overall throughput and total processing time from the initial data transfer to the final result retrieval.

In small-scale scenarios with relatively light input batches, we observed that bpftime-super introduces a modest overhead due to the persistent kernel’s initialization and JIT compilation steps. This overhead is offset by a significant reduction in kernel launch frequency, which becomes increasingly beneficial as task size or complexity grows. For larger inputs and more irregular workloads, bpftime-super consistently outperformed or closely matched the

native implementation in total execution time. The ability to avoid repeated kernel launches, particularly under streaming or dynamically generated tasks, yielded lower latency and more consistent GPU utilization.

We also recorded the memory transfer overhead for both approaches. Although `bpftime-super` uses `boost : managed_shared_memory` and requires additional synchronization, this extra cost did not dominate the runtime. In many instances, caching partial intermediate outputs or reusing allocated buffers compensated for any added latency. When evaluating cumulative performance across multiple runs, the ability of the persistent kernel to remain in place and handle a continuous stream of tasks proved advantageous.

Overall, the end-to-end comparison with native code demonstrates that `bpftime-super` effectively maintains high throughput while offering flexibility in dynamically injecting PTX and managing shared memory structures. Future optimization efforts will focus on fine-tuning the JIT layer and synchronization methods to further reduce the marginal overhead observed in smaller workloads. As GPU architectures evolve with features like Grace Hopper and CXL memory pools, we anticipate even tighter integration and improved performance for persistent kernel-based approaches.

## 8 CONCLUSION

Our preliminary results indicate that `bpftime-super` successfully reduces overhead by avoiding frequent kernel launches, enables rapid data exchange through shared memory, and accommodates dynamic code injection with minimal interruption to active kernels. Persistent kernels ensure long-lived GPU occupancy, which streamlines computational pipelines and balances load more effectively for streaming or irregular tasks. The use of `boost : managed_shared_memory` allows the CPU and GPU to operate over a common data space, lowering the cost of repeated transfers and simplifying concurrency control. Meanwhile, PTX injection grants the ability to adapt or instrument GPU code at run time without reinitializing the entire runtime environment, making it highly suitable for scenarios that demand real-time responsiveness.

Although the approach shows promise, certain limitations remain, including JIT compilation overhead and managing smaller workload kernels under persistent execution models. Additional optimizations—such as adjusting the number of streaming multiprocessors per kernel or using hardware-accelerated contexts on newer GPU architectures—can further improve performance. We plan to explore broader device contexts and memory-sharing paradigms, especially with upcoming Grace Hopper GPUs and CXL memory pools, to expand the scope and scalability of our solution. Overall, our initial findings encourage further refinement of persistent kernel strategies, tighter synchronization mechanisms, and flexible PTX-based instrumentation, with the ultimate goal of delivering a low-latency, high-throughput framework for GPU-centric workloads.

## REFERENCES

- [1] Auto-instrumentation for gpu performance using ebpf, 2025.
- [2] Valentin Andrei. System@scale: Ai observability. <https://atscaleconference.com/systemscale-ai-observability/>, 2024. Retrieved January 2024.
- [3] Josie E Rodriguez Condia, Juan-David Guerrero-Balaguera, Fernando F Dos Santos, Matteo Sonza Reorda, and Paolo Rech. A multi-level approach to evaluate the impact of gpu permanent faults on cnn’s reliability. In *2022 IEEE International Test Conference (ITC)*, pages 278–287. IEEE, 2022.
- [4] Bengisu Elis, David Boehme, Olga Pearce, and Martin Schulz. A mechanism to generate interception based tools for hpc libraries. In *European Conference on Parallel Processing*, pages 107–120. Springer, 2024.
- [5] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. {BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501, 2021.
- [6] Yue Guan, Yuxian Qiu, Jingwen Leng, Fan Yang, Shuo Yu, Yunxin Liu, Yu Feng, Yuhao Zhu, Lidong Zhou, Yun Liang, et al. Amanda: Unified instrumentation framework for deep neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 1–18, 2024.
- [7] Kshitij Gupta, Jeff A Stuart, and John D Owens. *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE, 2012.
- [8] Mingcong Han, Weihang Shen, Guanwen Peng, Rong Chen, and Haibo Chen. Microsecond-scale dynamic validation of idempotency for gpu kernels. *arXiv preprint arXiv:2410.23661*, 2024.
- [9] Zhuobin Huang, Xingda Wei, Yingyi Hao, Rong Chen, Mingcong Han, Jinyu Gu, and Haibo Chen. Parallelgpus: A concurrent os-level gpu checkpoint and restore system using validated speculation. *arXiv preprint arXiv:2405.12079*, 2024.
- [10] Aditya K Kamath and Arkaprava Basu. Iguard: In-gpu advanced race detection. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 49–65, 2021.
- [11] Suhwan Kim, Changue Jung, and Younghoon Kim. Comparative analysis of gpu stream processing between persistent and non-persistent kernels. In *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*, pages 2330–2332. IEEE, 2022.
- [12] Pouya Kousha, Bharath Ramesh, Kaushik Kandadi Suresh, Ching-Hsiang Chu, Arpan Jain, Nick Sarkauskas, Hari Subramoni, and Dhableswar K Panda. Designing a profiling and visualization tool for scalable and in-depth analysis of high-performance gpu clusters. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 93–102. IEEE, 2019.
- [13] David Pankratz, Tyler Nowicki, Ahmed Eltantawy, and José Nelson Amaral. Vulkan vision: Ray tracing workload characterization using automatic graphics instrumentation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 137–149. IEEE, 2021.
- [14] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. Cudaadvisor: Llvm-based runtime profiling for modern gpus. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 214–227, 2018.
- [15] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.
- [16] Yiwei Yang, Pooneh Safayanikoo, Jiacheng Ma, Tanvir Ahmed Khan, and Andrew Quinn. Cxlmemsim: A pure software simulated cxl. mem for performance characterization. *arXiv preprint arXiv:2303.06153*, 2023.
- [17] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. {λ-IO}: A unified {IO} stack for computational storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 347–362, 2023.
- [18] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, Toshio Endo, and Satoshi Matsuoka. Perks: a locality-optimized execution model for iterative memory-bound gpu applications. In *Proceedings of the 37th International Conference on Supercomputing*, pages 167–179, 2023.
- [19] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, XiaoZheng Lai, and Andrew Quinn. bpftime: userspace ebpf runtime for uprobes, syscall and kernel-user interactions. *arXiv preprint arXiv:2311.07923*, 2023.
- [20] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. {XRP}:{In-Kernel} storage functions with {EBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.
- [21] Keren Zhou, Laksono Adhianto, Jonathon Anderson, Aaron Cherian, Dejan Grubisic, Mark Krentel, Yumeng Liu, Xiaozhu Meng, and John Mellor-Crummey. Measurement and analysis of gpu-accelerated applications with hpctoolkit. *Parallel Computing*, 108:102837, 2021.