

Enabling Hardware-Software Co-Design for Large-Scale AI Systems via Fine-Grained GPU Memory Tracing

Palak Mishra¹, Rajat Bhardwaj¹, Yash Verma¹, Ramanjeet Singh², Rinku Shah¹

¹IIIT-Delhi, India ²Marvell Technology Inc., India

{palakm, rajat24071, yash23610}@iiitd.ac.in, ramanjeets@marvell.com, rinku@iiitd.ac.in

CCS Concepts

• **Computer systems organization** → *Parallel architectures*; • **Computing methodologies** → *Modeling and simulation*; • **Networks**;

Keywords

GPU memory tracing, large-scale AI systems, HW-SW co-design

ACM Reference Format:

Palak Mishra¹, Rajat Bhardwaj¹, Yash Verma¹, Ramanjeet Singh², Rinku Shah¹. 2026. Enabling Hardware-Software Co-Design for Large-Scale AI Systems via Fine-Grained GPU Memory Tracing. In *ACM SIGCOMM Workshop on Memory-Semantic Networking for AI-System Systems (MemNetAI '26)*, August 17–21, 2026, Denver, CO, USA. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3789240.3822570>

1 Introduction

Memory traces capturing GPU operations over modern scale-up fabrics (e.g., load/store accesses, ordering events, synchronization, and communication behavior) enable a broad set of research directions for next-generation AI infrastructure.

First, such traces provide a foundation for evaluating emerging disaggregated memory architectures, including GPU memory pooling, HBM–CXL tiered memory systems, remote KV-cache systems for LLM serving, and disaggregated MoE expert storage. Trace-driven analysis enables realistic characterization of locality, burstiness, remote access granularity, latency tolerance, migration efficiency, and cacheability — all of which are critical for designing scalable memory disaggregation.

Second, these GPU memory operation traces are valuable for protocol and fabric co-design efforts targeting emerging industry specifications designed for AI scale-up networking, such as UAL [14] and ESUN [11]. Real workload traces enable evaluation of packetization strategies, ordering semantics, transport protocols, switch buffering policies, and collective communication optimizations under realistic GPU-generated traffic patterns.

Coarse-grained profiles breed inaccurate low-level simulations. Chakra [13] defines schemas for execution graphs and dependencies, but lacks fine-grained memory tracing. For example, Chakra does not support per-kernel warp-level traces, access distribution across memory spaces, and memory operation characterization. Nsight Systems [9] and PyTorch’s Kineto-based profiler [12]



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

SIGCOMM '26, August 17–21, 2026, Denver, CO, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2467-1/26/08

<https://doi.org/10.1145/3789240.3822570>

Table 1: Comparing Profiling Tools.

Profiler	Approach	Memory Access Pattern	Granularity	Overhead
NVBit [15]	Assembly	Yes	Warp	High
Nsight Compute [8]	CUPTI APIs	Partial	Warp	High
Nsight Systems [9]	CUPTI APIs	No	Warp	Medium
eGPU [16]	eBPF	No	–	Low
eInfer [1]	eBPF	Partial	Kernel	Medium
CUPTI [6]	–	PM Sample	Warp	Low
GPA [17]	PM Sample	Yes	Warp	Low
Our Tool	PC Sampling + SASS	Yes	Warp	Low

use CUPTI [6] to expose GPU, PCIe, NVLink, and operator-level performance metrics. eInfer [1] profiles distributed GPU workloads at a coarse granularity, while eGPU [16] extends eBPF [2] observability to GPUs by translating eBPF programs into PTX [10] for lightweight kernel instrumentation. However, these systems do not focus on fine-grained memory tracing or on the bottlenecks that arise from such memory access patterns (see Table 1).

Fine-grained, instruction-level tracing fails to scale, and often perturbs workload behavior. NVBit [15] rewrites GPU SASS [5] to enable instruction-level instrumentation of memory operations. However, its in-band tracing perturbs program behavior and can incur up to 100× runtime overhead, limiting scalability [15]. Similarly, SnooPie [4] relies on NVBit to trace GPU memory accesses and inherits similar reliability and overhead issues. Nsight Compute [8] provides instruction-level profiling but lacks warp-level memory tracing.

Achieving non-invasive, instruction-level precision at scale. NVIDIA CUPTI’s Program Counter (PC) sampling [7] periodically samples executing instructions out-of-band and attributes events such as memory accesses and stalls to program counters, enabling low-overhead memory analysis without binary modification. Compared to instrumentation-based approaches like NVBit and eGPU, it offers better scalability and lower execution perturbation, but sacrifices temporal resolution and requires post-processing to reconstruct memory traces. GPA [17] supports causal “what-if” analysis to identify bottlenecks, but does not generate benchmarks.

Our proposal. Design an execution trace generator that captures and replays AI/ML workload memory behavior using PC sampling during kernel execution. We collect temporal samples of memory accesses, instruction classes, and warp stall reasons across kernels at warp-level granularity. Using these traces, we characterize workload behavior and plan to build realistic memory benchmarks for distributed training and inference.

Challenges. (1) PC sampling records stall reasons and program counters at warp-level granularity. However, there is no direct correspondence between the PC reported by the sampler and the PC of the target SASS instruction. Consequently, the sampled PC cannot

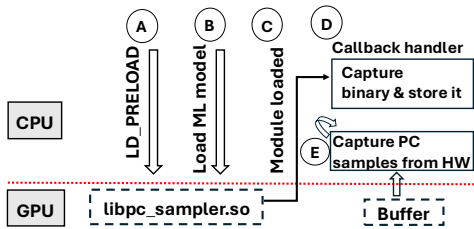


Figure 1: Pipeline for Online Runtime Trace Generation. Components B–E run throughout the execution.

be directly used to identify stalls caused by a specific SASS instruction. We engineer a correlation pipeline that maps sampled PCs to decoded SASS instructions, binding hardware events to specific instruction types for precise execution profiling. (2) Modern ML frameworks use asynchronous CUDA streams to launch thousands of overlapping kernels, making accurate temporal tracing difficult. We propose using an optimal PC sampling frequency within the hardware buffer limit constraint and capture sufficiently fine-grained memory access patterns, while ensuring that instructions aren't dropped. We achieve partial temporal ordering of memory operations and analyze kernel dominance, memory hierarchy usage, operation types, and stall reasons.

2 Design and Implementation

Figure 1 shows the overall design of our memory tracing tool.

Phase-I: Online Low-Overhead Trace Generation. The profiler (`pc_sampler_v15a.cu`) is compiled into a shared library and injected via `LD_PRELOAD`, initializing the CUPTI PC sampling session before any CUDA kernel launches occur. At each sampling interval (every 1M cycles), the hardware samples the PC and the associated stall reason for each active warp across all streaming multiprocessors. The stall reason encodes the microarchitectural condition that prevents a warp from issuing an instruction.

Our tool transparently extracts assembly from embedded CUDA binaries (CUBINs) via an `LD_PRELOAD` library (`libpc_sampler.so`) that intercepts driver module-loading APIs (e.g., `cuModuleLoad`) and writes the assembly to disk. This interception layer adds minimal latency to the CUDA driver's critical path while dumping raw binary payloads to disk. The sampling approach captures aggregate warp behavior instead of individual instruction events, yielding a statistically representative view of execution bottlenecks with low overhead while preserving temporal structure via monotonic timestamps associated with each sample batch.

Why 1M cycle sampling rate? PC sampling employs a two-stage buffering pipeline in which per-SM (Streaming Multiprocessor) buffers feed into a shared hardware buffer with a capacity of 512MB. We chose a sampling interval of 1M cycles because we observed that higher-frequency PC sampling (e.g., 2^{10} and 2^6) causes SM buffers to overflow, resulting in dropped instructions (see Table 2).

Phase-II: Offline High-Fidelity Correlation and Mapping. The mapping phase semantically enriches the pipeline by joining sampled PC observations with instruction-level SASS disassembly using a memory-efficient, two-level index. The outer index maps sanitized kernel names to .sass file paths, while the inner index maps PC offsets to SASS mnemonic strings. For each record, the pipeline

sanitizes the kernel name, resolves the file path, and extracts the SASS mnemonic. Mismatches are systematically logged: SASS-level misses capture unextracted JIT or dynamic kernels, while offset-level misses capture sampling jitter or version-specific PC alignment variations. Successfully mapped records are enriched with architectural attributes from a classification database, which we built from existing NVIDIA documentation [5], including operation category, target memory space, uniformity, and Tensor Core involvement. Concurrently, a width extraction utility uses regular expressions to parse bit-width tokens (e.g., .64, .128), assigning a transaction width and confidence rating (EXACT, DEFAULT, UNKNOWN). The resulting enriched dataset provides the foundational per-sample ground truth required for simulator validation and temporal workload analysis.

3 Evaluation

Setup. We evaluated our pipeline for the NanoGPT-10M training workload (200 iterations) across two NVIDIA H100 GPUs and $4 \times$ H200 GPUs, connected via PCIe and NVLink, respectively, for data parallelism.

Can our pipeline help construct a real-world memory benchmark? With the sampling time frames, the multi-kernel behavior was captured by transforming raw data into timestamp vectors. By correlating the stall reasons (Figure 2), memory operations (Figure 3), and memory hierarchy (Figure 4), it is possible to attribute the sample time frame(s) to the corresponding template memory abstraction, and construct memory benchmark. We see that the dominant memory operation is MEMORY (Load/Store), stall reason is `tex_throttle`, and memory hierarchy is GLOBAL (off-chip memory access), indicating a bottleneck due to busy queue at TMU. Therefore, the benchmark can pick from the TMU template operations.

Can our pipeline identify dynamic NCCL communication protocols? NCCL dynamically selects the communication protocols (i.e., LL, LL128, and SIMPLE) at the start of every individual collective operation [3]. Using SASS disassembly and PC sampling mapping, we could observe the exact communication protocol temporally, which can feed into our memory benchmark.

4 Conclusion and Future Work

In this paper, we presented a GPU memory tracing tool. To realize a comprehensive execution trace generator for distributed ML workloads, we plan to extend the tool with low-overhead support for capturing inter-operation memory dependencies and scale-up memory access patterns.

While the Chakra [13] DAG defines the standard execution trace format for representing AI/ML workloads, including memory operations, the captured memory traces are coarse-grained and lack information about stall reasons that occur due to memory access patterns. We plan to extend our tool as a feeder pipeline that maps PC samples to decoded SASS and groups them into accurate Memory Nodes within Chakra's execution graph.

References

- [1] Xexin Chu, Jianchang Su, Yifan Zhang, Chenxingyu Zhao, Yiwei Yang, Yusheng Zheng, Shengkai Lin, Shizhen Zhao, and Wei Zhang. 2025. eInfer: Unlocking

- Fine-Grained Tracing for Distributed LLM Inference with eBPF. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25)* (Coimbra, Portugal). Association for Computing Machinery, New York, NY, USA, 76–83. <https://doi.org/10.1145/3748355.3748372>
- [2] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunstrom. 2024. The eBPF Runtime in the Linux Kernel. *arXiv preprint arXiv:2410.00026* (2024). <https://arxiv.org/abs/2410.00026>
 - [3] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jauegy, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. 2025. Demystifying NCCL: An in-depth analysis of GPU communication protocols and algorithms. In *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 48–59.
 - [4] Mohammad Kefah Taha Issa, Muhammad Aditya Sasongko, Ilyas Turimbetov, Javid Baydamirli, Doğan Sağbılı, and Didem Unat. 2024. SnooPie: A Multi-GPU Communication Profiler and Visualizer. In *Proceedings of the 38th ACM International Conference on Supercomputing*. ACM, Kyoto Japan, 525–536. <https://doi.org/10.1145/3650200.3656597>
 - [5] NVIDIA. 2026. CUDA Binary Utilities. <https://docs.nvidia.com/cuda/cuda-binary-utilities/> Documentation for NVIDIA SASS (Streaming Assembler) disassembly and instruction set reference.
 - [6] NVIDIA. 2026. CUDA Profiling Tools Interface (CUPTI). <https://developer.nvidia.com/cupti>
 - [7] NVIDIA. 2026. CUPTI PC Sampling API. https://docs.nvidia.com/cupti/api/group__CUPTI_PCSAMPLING__API.html
 - [8] NVIDIA. 2026. NVIDIA Nsight Compute. <https://developer.nvidia.com/nsight-compute>
 - [9] NVIDIA. 2026. NVIDIA Nsight Systems. <https://developer.nvidia.com/nsight-systems>
 - [10] NVIDIA. 2026. Parallel Thread Execution ISA. <https://docs.nvidia.com/cuda/parallel-thread-execution/> NVIDIA CUDA Toolkit Documentation.
 - [11] Open Compute Project. 2026. OCP ESUN - Network Operator Requirements - Base Specification. <https://www.opencompute.org/documents/ocp-esun-network-operator-requirements-base-specification-rev-1-0-final-pdf>
 - [12] PyTorch Contributors. 2021. Kineto: Profiling Library for PyTorch. <https://github.com/pytorch/kineto>
 - [13] Srinivas Sridharan, Theodor-Adrian Badea, Andy Balogh, Bradford M. Beckmann, Brian Coutinho, Louis Feng, Sheng Fu, Sanshan Gao, Mehryar Garakani, Taekyung Heo, David Kanter, Josh Ladd, Ziwei Li, Winston Liu, Changhai Man, Dan Mihailescu, Spandan More, Joongun Park, Ashwin Ramachandran, Vinay Ramakrishnaiah, Saeed Rashidi, Vijay Janapa Reddi, Puneet Sharma, Phio Tian, William Won, Hanjiang Wu, Huan Xu, Jinsun Yoo, and Tushar Krishna. 2026. MLCommons Chakra: Advancing Performance Benchmarking and Co-design using Standardized Execution Traces. *arXiv:2605.11333 [cs.DC]* <https://arxiv.org/abs/2605.11333>
 - [14] UALink. 2026. Ultra Accelerator Link (UALink). <https://ualinkconsortium.org/specification/>
 - [15] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 372–383. <https://doi.org/10.1145/3352460.3358307>
 - [16] Yiwei Yang, Tong Yu, Yusheng Zheng, and Andrew Quinn. 2025. eGPU: Extending eBPF Programmability and Observability to GPUs. In *4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25)* (Rotterdam, Netherlands). Association for Computing Machinery, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3723851.3726984>
 - [17] K. Zhou, X. Meng, R. Sai, and J. Mellor-Crummey. 2021. GPA: A GPU Performance Advisor Based on Instruction Sampling. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Seoul, Korea (South), 115–125. <https://doi.org/10.1109/CGO51591.2021.9370339>

A Selecting a Sampling Frequency Under Buffer Constraints

Table 2: Impact of PC sampling frequency on instruction collection. S1: 4×H200 GPUs connected via NVLink. S2: 2×H100 GPUs connected via PCIe. Hardware PC-sampling buffer size = 512 MB. Buffer size for S1: 100000; S2: 1000000.

Server	Sampling Freq.	# Instr. Sampled	Drops
S1	2 ²⁰	153K	No
	2 ⁶	700	Yes
S2	2 ²⁰	1.54M	No
	2 ¹⁰	109K	Partial
	2 ⁶	11K	Yes

B Evaluation Plots

Temporal Distribution of Stall Reasons. Figure 2 shows the temporal evolution of warp stall reasons obtained from CUPTI PC sampling over the NanoGPT training workload. We observe that the dominant stall reason throughout execution is `tex_throttle`, indicating the instruction L1 queue is full with texture operations.

Stall distributions evolve gradually rather than exhibiting abrupt transitions, suggesting that statistical PC sampling preserves stable temporal execution characteristics despite not recording every instruction. Less frequent events such as `warpgroup_arrive`, `selected_not_issued`, and `wait_not_issued` remain comparatively rare, indicating that memory subsystem pressure dominates over scheduling-related stalls for this workload.

Temporal Distribution of Memory Operation Categories.

Figure 3 shows the temporal evolution of semantic instruction categories obtained after correlating PC samples with decoded SASS instructions. MEMORY (load/store) operations consistently dominate execution, demonstrating that memory accesses constitute the primary activity throughout training. UNIFORM and TMA instructions contribute a smaller but persistent fraction, reflecting scalar operations and Hopper Tensor Memory Accelerator transfers, respectively, while ATOMIC and SYNC instructions occur infrequently.

Temporal Distribution of Memory Spaces. Figure 4 shows the temporal distribution of memory spaces accessed by sampled SASS instructions. CONST memory space with grid-level scope, resides on the HBM memory throughout the application lifetime, and is read-only to the GPU kernel. GLOBAL memory space is accessible to all threads in the kernel and resides in the HBM, whereas LOCAL resides in the HBM but is thread-local. SHARED memory sits on the L1 cache, and is accessible by all the threads in a thread-block. We observe that CONST and GLOBAL memory dominate throughout execution, followed by SHARED memory, while LOCAL memory usage remains minimal, indicating limited register spilling for the evaluated workload.

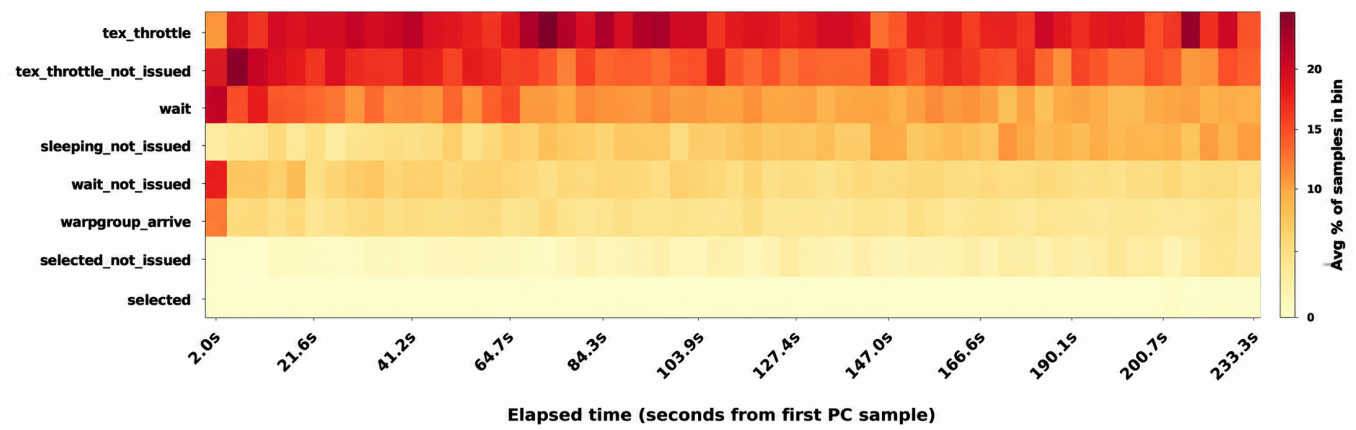


Figure 2: Temporal samples of Stall reason. `tex_throttle`: Texture Mapping Unit is busy, and its queue is full, `wait` in L1-tex; `wait`: Warp stalled on fixed latency execution dependency.

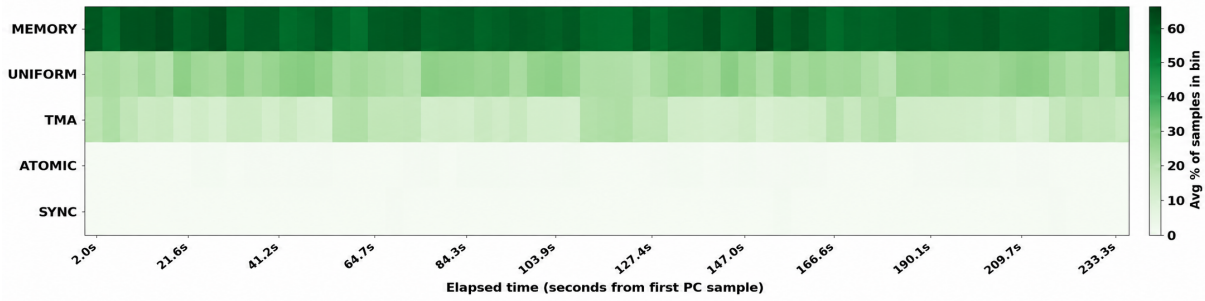


Figure 3: Temporal samples of Memory Operations. `MEMORY`: load/store; `Uniform`: scalar register shared by threads in a warp; `TMA`: load from HBM to shared bypassing registers; `ATOMIC`: in-memory operation; `SYNC`: Synchronization and memory barrier operations.

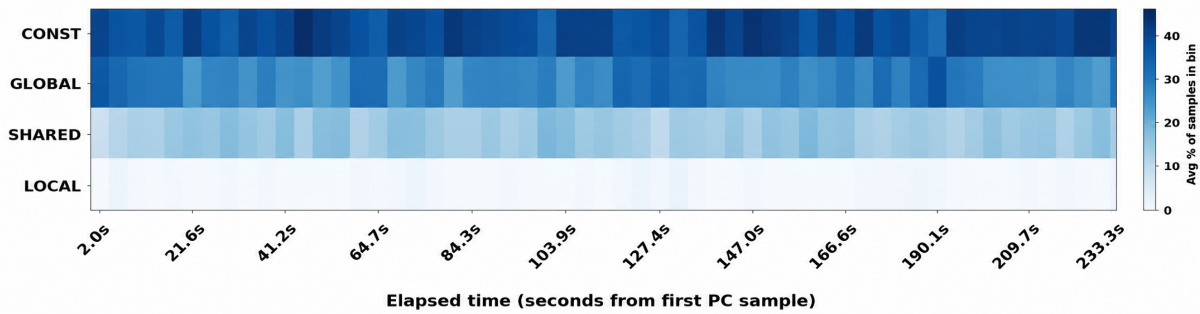


Figure 4: Temporal samples of Memory Spaces. `Constant`: Grid|Device ; `Global`: Grid|Device ; `Shared`: Block|SM ; `Local`: Thread|Device