Design and Evaluation of a Standardized Interface for AMD GPU Performance Metrics

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Dong Jun Woun

May 2026

With deepest gratitude to my parents, whose unwavering love and unconditional support have shaped who I am. Thank you for believing in me, cheering on my studies and journeys, and for your boundless love. This thesis is for you.

Acknowledgements

I am deeply grateful to the advisors and mentors whose guidance shaped this thesis and my development as a researcher.

I am grateful to Dr. Heike Jagode for a lab where inquiry and collaboration thrived and for steady guidance on my academics and thesis. Her trust in my efforts and the opportunities she provided were central to this work.

I am especially thankful to Dr. Catherine Schuman, who stepped in as my primary thesis advisor and provided consistent direction. Her mentorship, thoughtful feedback, and guidance have been invaluable throughout my studies.

I also thank Dr. James S. Plank, whose inspiring teaching strengthened my foundation in computer science and whose encouragement helped me pursue further opportunities.

I appreciate their collective support of my academic and professional development, including their willingness to advocate for me when needed. Any strengths in these pages reflect their example.

Abstract

Modern high-performance computing (HPC) systems increasingly rely on graphics processing unit(GPU) accelerators, making accurate and low-overhead monitoring of hardware behavior essential for performance analysis, energy optimization, and reliability studies. Differences and updates in vendor application programming interfaces(API) make cross-platform measurement difficult and force researchers to maintain vendor-specific code. The Performance API (PAPI) provides a common interface for accessing hardware performance metrics on central processing units(CPU), GPUs and interconnects. However, AMD's transition from ROCM System Management Interface (ROCm SMI) to the AMD System Management Interface (AMD SMI)—deprecating the former interface previously used by PAPI to access GPU metrics—underscores the need for a robust and vendor-neutral interface for AMD GPUs and accelerated processing units (APU). This thesis presents the design and evaluation of a portable performance monitoring interface for AMD GPUs that abstracts vendor-specific API differences while maintaining low-overhead. The proposed interface ensures measurement stability across software and hardware versions. The novelty lies in a probe-based, automatic discovery mechanism that ensures access to available metrics and consistency across GPU generations. The implementation is built on AMD SMI and integrated into the PAPI component model. The new component automatically detects which metrics are supported by a specific device and driver and exposes only those metrics. The interface provides access to metrics for monitoring power and energy, temperature, engine

activity, and interconnect bandwidth/link state. Experimental results confirm that the proposed interface maintains near-native measurement overhead and produces consistent results across different AMD GPU models and ROCm versions, validating its portability and reliability. In summary, this work enables portable, low-overhead access to AMD GPU/APU performance metrics, reducing the maintenance burden of vendor-specific instrumentation, and improving reproducibility in performance studies. It further enhances the portability and reliability of GPU/APU performance monitoring, providing a sustainable foundation for cross-platform performance analysis in heterogeneous HPC environments.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern high-performance computing (HPC) relies on graphics processing units(GPU) to meet the demands of scientific computing Atchley et al. (2022). Effective GPU performance engineering requires accurate, low-overhead measurement of hardware performance metrics. These metrics include compute utilization, interconnect traffic, and power consumption, which are used to analyze how optimizations impact execution time and hardware resource utilization Islam et al. (2024). Portable access to GPU hardware performance metrics with minimal overhead is challenging due to cross-vendor application programming interface(API) design differences and changes across software and firmware versions. These differences hinder cross-platform measurement and reproducibility.

The Performance API (PAPI) provides researchers with access to hardware counters through a standardized interface across central processing units (CPU) and GPUs Jagode et al. (2024). AMD is deprecating ROCM System Management Interface (ROCm SMI) in favor of AMD System Management Interface (AMD SMI), retiring the interface that PAPI previously used to read AMD GPU metrics Advanced Micro Devices, Inc. (2023). This change motivates the need for a robust, vendor-neutral interface for AMD GPUs and accelerated processing units (APU).

1.1 Research Objectives

This work is organized around three objectives:

1. **Design a portable, low-overhead interface.** Design a performance monitoring interface for AMD GPUs that abstracts vendor-specific API differences, maintains low per-call overhead, and provides consistent access to supported metrics across GPU generations and software versions.

2. **Validate overhead, portability, and practical sampling intervals.** Demonstrate that the interface design achieves near-native per-call overhead relative to direct AMD SMI calls. Confirm consistent access to supported metrics across the evaluated GPU generations and software versions. Identify polling intervals that yield new information without excessive cost.

3. **Characterize metric access and limitations.** Provide and document the supported metrics to enable repeatable and comparable studies.

1.2 Contributions

This thesis makes the following contributions:

1. **Portable, low-overhead, probe-based interface design for AMD GPUs.** Introduces a novel, portable interface design for AMD GPU hardware performance metrics that abstracts vendor-specific API differences and keeps per-call overhead low. Employs probe-based runtime discovery to identify device and driver supported metrics across GPU generations and software releases. Built on AMD SMI and realized within PAPI.

2. **Empirical validation of overhead, portability, and monitoring interval.** Establishes near-native per-call overhead relative to direct AMD SMI calls and confirms consistent access to supported metrics across the evaluated GPU

generations and software releases. Identifies effective polling intervals that yield new information without increasing measurement cost.

3. **Platform-scoped reference of supported metrics and limits.** Documents the supported metrics on the evaluated platform to enable reproducible and comparable studies.

4. **Integration with PAPI.** Seamless incorporation into the PAPI framework, extending its functionality to current AMD GPU/APUs and ensuring continuity for existing users and tools.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews related work. Chapter 3 presents the vendor-neutral, low-overhead interface design and the probe-based discovery method (Objective 1). Chapter 4 validates overhead and portability and studies usable monitoring intervals (Objective 2). Chapter 5 analyzes metric coverage and limitations in practice (Objective 3). Chapter 6 concludes with a summary of findings, discusses limitations, and outlines directions for future work.

Chapter 2

Related Work

This chapter reviews hardware performance monitoring in high-performance computing (HPC), the fragmentation of vendor-specific GPU monitoring interfaces, and statistical methodologies for validating abstraction layers.

2.1 Hardware Performance Monitoring in HPC

Hardware performance counters are the primary mechanism for correlating application execution with physical resource utilization in heterogeneous systems. In heterogeneous HPC environments, tools must aggregate metrics from CPUs, GPUs, and interconnects Jagode et al. (2024). The shift toward GPU-accelerated systems, exemplified by Frontier Atchley et al. (2022), extended monitoring requirements to include GPU metrics such as power consumption and temperatures. Accurate measurement of these metrics enables data-driven analysis to correlate software optimizations with hardware resource usage Islam et al. (2024).

2.2 Vendor-Specific GPU Monitoring Interfaces

Vendors expose performance metrics through proprietary system management interfaces. For AMD GPUs, the ecosystem is transitioning from the legacy ROCm System

Management Interface (ROCm SMI) to the AMD System Management Interface (AMD SMI) Advanced Micro Devices, Inc. (2023).

This transition introduces challenges for portability. The two interfaces differ in data structures, function signatures, and unit conventions. For example, ROCm SMI reports power in microwatts whereas AMD SMI standardizes reporting in watts, with similar discrepancies found in energy accounting Advanced Micro Devices, Inc. (2024b, 2025d, 2024a, 2025e). Furthermore, accessing PCIe throughput and XGMI link status requires navigating evolving API definitions across driver versions Advanced Micro Devices, Inc. (2025a,h). This fragmentation forces researchers to maintain version-specific instrumentation code, hindering the development of portable performance tools.

2.3 Standardized Abstraction Layers

Standardized abstraction layers mitigate the complexity of vendor-specific APIs. The Performance API (PAPI) provides a uniform interface for accessing hardware counters across CPUs, GPUs, and interconnects, decoupling analysis tools from the underlying hardware implementation Jagode et al. (2024). By encapsulating vendor libraries within a component-based architecture, PAPI enables tools to query metrics using consistent event names and units. Similarly, the LIKWID tool offers a lightweight, standardized approach to performance monitoring, abstracting hardware details across various architectures to facilitate cross-platform analysis Treibig et al. (2010).

However, static interface designs are brittle when underlying vendor libraries change. The deprecation of ROCm SMI necessitated a new component based on AMD SMI. Conventional static bindings often require code maintenance to accommodate the frequent API changes observed in the ROCm ecosystem. This thesis addresses this limitation by combining dynamic symbol resolution with a probe-based runtime

discovery mechanism. This approach ensures consistency across software versions without requiring manual adaptation to driver updates.

2.4 Methodologies for Low-Overhead Validation

Validating monitoring overhead requires statistical methods to ensure the abstraction introduces negligible latency. Systems research employs equivalence testing to verify performance equivalence. The Two One-Sided Tests (TOST) procedure determines if two measurement sets are statistically equivalent within a specified margin Schuirmann (1987). This work adopts TOST to quantify the overhead of the proposed standardized interface relative to direct vendor API calls.

Chapter 3

Design of a Portable, Low-Overhead Interface for AMD GPU Hardware Performance Metrics

3.1 Overview and Role of the AMD SMI Interface

This chapter describes the design of a portable, low-overhead interface for AMD GPU hardware performance metrics, built upon AMD's System Management Interface (AMD SMI). The design defines an abstraction layer that standardizes metric access and resolves metric availability across differing AMD architectures and API versions. Through this interface, AMD GPU/APU hardware performance metrics can be enumerated and sampled.

To support these capabilities, the design prioritizes three architectural objectives. First, it provides a vendor-neutral abstraction for AMD GPU/APU metrics that ensures consistency across hardware and software generations. Second, it validates metric support through a probe-based discovery mechanism that guarantees only valid, supported counters are exposed. Third, it keeps the read path minimal so that sampling overhead remains comparable to direct AMD SMI calls. To realize

these capabilities, the interface is incorporated within the PAPI framework. This integration enables PAPI to extend its functionality to current AMD GPUs and APUs and ensures continuity for existing users and tools.

3.2 Design Rationale and Key Decisions

Design Summary. The proposed interface was motivated by AMD's transition from ROCm SMI to AMD SMI, which disrupted the previously stable GPU monitoring functionality within PAPI. To restore portability and robustness, the design introduces a probe-based discovery mechanism that identifies and validates available GPU metrics at runtime. Concretely, the interface dynamically loads the AMD SMI library, discovers devices, and constructs an event table, based on runtime probing, that maps each exposed metric to specific AMD SMI functions. Furthermore, by executing metric queries through a short function-pointer path with strictly bounded accessors, the design maintains negligible measurement overhead. This architecture allows the system to remain resilient to frequent vendor API changes while abstracting underlying driver complexities.

Design Challenges. Developing a robust interface for AMD hardware metrics presented challenges. First, changes in function names and data structures across driver releases required the interface to resolve library symbols at runtime rather than relying on static definitions. Second, the absence of a metric enumeration API within AMD SMI prevented the interface from querying the library for a list of supported counters. These challenges motivated the probe-based discovery and validation mechanisms at the core of the design, enabling consistent metric access across environments.

Runtime Event Registration. Devices and driver revisions expose different metric subsets. At initialization, the interface builds an event table from library reports and function probes that verify availability of metrics, ensuring unsupported counters are not exposed. Each entry records a name, variant (for multi-field queries), a device selector, and a pointer to the accessor function.

Probe-Based Metric Discovery. Since AMD SMI lacks a native enumeration API, the interface constructs an event table by iterating over potential metrics. It validates each candidate metric using targeted runtime probes. Only metrics that successfully return data are registered, creating a guaranteed list of supported events. A compact sketch of this logic:

```
/* Pseudo: event table construction via probing */
for each metric_family F in predefined_list:
  for each device d and variant k in F:
    if probe_successful(d, F, k):
      add_event(name=F.base, descr=..., device=d,
                variant=k, access_func=F.read);
```

This strategy prevents runtime failures during measurement and provides an event table of valid events(metrics).

Handling API Version Divergence. Changes in symbol names and data structures across ROCm releases (6.4.0–7.2.0) pose compatibility challenges. To maintain stability, the interface employs a hybrid strategy: (i) compile-time guards adapt to structural definitions based on available headers, and (ii) a runtime primary/fallback mechanism resolves renamed function symbols:

```
/* Pseudo: primary/fallback binding */
void *sym(void *h, const char *primary, const char *fallback){
  void *p = dlsym(h, primary);
  return p ? p : (fallback ? dlsym(h, fallback) : NULL);
}
```

Low-Latency Bounded Accessors. Safe execution of vendor APIs requires validating inputs and initializing memory to prevent crashes, without compromising low latency. Consequently, accessors zero-initialize vendor structures, enforce buffer limits, and verify pointers. Failures return per-event error codes rather than aborting. Figure 3.1 shows an accessor that checks the access mode, validates the device index and handle, zero-initializes the PCIe bandwidth structure, and bounds-checks the vendor-provided rate index before returning PCIe bandwidth metrics. This design ensures robustness with negligible overhead, as quantified in Chapter 4.

Concurrency and Device Ownership. To ensure thread safety within a single process, the component enforces mutual exclusion on a per-device basis. Access is managed via a global, lock-protected 64-bit `device_mask`. When a context is initialized, the component attempts to atomically claim the target device bit. Other eventsets and threads in the same process is blocked from using the same GPU concurrently. Ownership is retained until the context is explicitly destroyed via `PAPI_cleanup_eventset`, preventing internal conflicts without restricting GPU access from external processes or tools.

3.3 Architecture and Core Lifecycle

Architectural Summary. The interface functions as a modular component connecting the standardized PAPI interface to the native AMD SMI library. It encapsulates vendor state—including device handles and function pointers—within a component-specific context, isolating the PAPI framework from driver details. The design maps events to specific accessor functions that execute the corresponding AMD SMI library calls to retrieve metric data.

```
int access_amdsmi_pci_bandwidth(int mode, void *arg)
{
  if (mode != PAPI_MODE_READ || !amdsmi_get_gpu_pci_bandwidth_p)
    return PAPI_ENOSUPP;

  native_event_t *event = (native_event_t *)arg;
  if (event->device < 0 || event->device >= device_count ||
      !device_handles || !device_handles[event->device])
    return PAPI_EMISC;

  amdsmi_pcie_bandwidth_t bw = (amdsmi_pcie_bandwidth_t){0};

  if (amdsmi_get_gpu_pci_bandwidth_p(device_handles[event->device], &bw) !=
      AMDSMI_STATUS_SUCCESS)
    return PAPI_EMISC;

  uint32_t cur = bw.transfer_rate.current;
  if (cur >= bw.transfer_rate.num_supported)   /* bounds check */
    return PAPI_EMISC;

  switch (event->variant) {
    case 0:
      event->value = bw.transfer_rate.num_supported;
      break;
    case 1:
      event->value = (int64_t)bw.transfer_rate.frequency[cur];
      break;
    case 2:
      event->value = bw.lanes[cur];
      break;
    default:
      return PAPI_ENOSUPP;
  }
  return PAPI_OK;
}
```

Figure 3.1: Excerpt of a bounded accessor (PCIe bandwidth metrics).

Core Lifecycle. The lifecycle comprises initialization, measurement, and shutdown. Initialization dynamically loads the vendor library and probes for metrics. During measurement, the component opens a context for the metrics measured, acquires device ownership via a global device mask, and iterates over requested events to invoke their accessor functions. Shutdown releases memory and library references. Figure 3.2 details this sequence.

Dynamic Library Loading and Symbol Resolution To decouple the interface from specific driver versions, the component dynamically loads the `libamd_smi.so` shared library at runtime via `dlopen`. Symbol resolution utilizes a primary/fallback strategy to accommodate function renames across software releases. Essential symbols are strictly validated during initialization; if missing, the component disables itself to prevent runtime failures. Figure 3.2 details this initialization sequence.

Initialization and Device Discovery Following library loading, the interface queries the system topology to enumerate sockets and devices. It then allocates state structures and builds the event table by probing each metric against the detected hardware. This validation filters unsupported capabilities, ensuring that only valid metrics are exposed.

Event Representation The interface defines metrics using a `native_event_t` structure that pairs an event name with an accessor function pointer. Variant indices handle parameterized queries, such as distinguishing between specific memory domains or sensor IDs. To support multi-GPU environments, device masks map the abstract event definition to specific hardware instances.

Runtime Measurement and Context Lifecycle For each metric sampled, the interface constructs a context holding per-event state and a device usage mask. The open operation decodes event codes, assigns device identifiers, and atomically acquires device ownership. Read iterates through events, invoking accessor functions to populate the output buffer. Close releases resources and returns devices to the available pool. This design ensures thread-safe sampling by preventing conflicting access to the same device. Figure 3.2 details this sequence.

Accessor Functions and Vendor Interaction Accessors execute the read path for individual metrics. The logic validates the device index, verifies the function pointer, and invokes the corresponding AMD SMI function to store the result. As

detailed in Figure 3.1, this flow relies on a memory safe call using zero-initialized structures.

PAPI Integration The `papi_vector_t` structure maps standard PAPI entry points—such as `init_component`, `start`, `read`, and `stop`—to the AMD SMI component. This structure registers the component with the PAPI framework.

3.4 Code Walkthrough of Key Modules

Module Organization. The source code separates the PAPI integration layer from the native hardware access logic. Core functionality such as dynamic loading and device discovery is contained within the core module, while event enumeration and mapping, concurrency management, and metric accessors are isolated in distinct source files. This structure decouples the generic component lifecycle from the specific AMD SMI library interactions.

amds.c: Core Component Logic and Global State This module manages the component lifecycle and maintains global state. It executes the initialization sequence detailed in Figure 3.2, handling dynamic library loading (`load_amdsmi_sym()`), device discovery, and probe-based event table construction (`init_event_table()`). The module also defines shared structures, including device handles and hardware counts, used by accessor functions.

amds_evtapi.c: Event Enumeration and Mapping Enumerating available metrics is a core functionality of PAPI. This module implements the native event interface to handle this requirement by iterating through the global event table. Metric lookups use a hash table to map event names to their corresponding integer codes. Event codes act as direct indices into the event table, retrieving event descriptions and accessor functions.

amds_ctx.c: Context Management and Concurrency This module implements the measurement lifecycle and concurrency control. It handles operations—such as start, stop, and read—for sampled metrics. To ensure thread safety, the module uses a global device mask to enforce hardware ownership during sampling. It iterates through selected events, invoking accessor functions to populate the per-context result buffer.

amds_accessors.c: Metric Accessor Functions This module implements metric accessor functions that map PAPI events to specific AMD SMI library calls. Each accessor retrieves the underlying vendor data structure, extracts the target field based on the event variant, and applies unit conversions where necessary. Figure 3.1 illustrates this retrieval and validation logic.

linux-amd-smi.c: PAPI Integration This module defines the `papi_vector_t` entry points—including `init_component`, `start`, `read`, and `stop`—that bind the PAPI framework to the component logic. To minimize startup overhead, the design implements lazy initialization. The component initially reports `PAPI_EDELAY_INIT`, delaying the loading of the AMD SMI library until the first event access. This access triggers `_amd_smi_check_n_initialize()`, which invokes `_amd_smi_init_private()` to dynamically load the vendor library and populate the event table.

Header File Organization The component source is structured across four headers. `amds.h` declares the internal interface used by the PAPI integration wrapper, including functions for initialization, shutdown, and context management. `amds_priv.h` defines private data structures and macros for accessing global device handles and hardware counts. `amds_funcs.h` uses macro expansion to generate function pointers for the AMD SMI library, facilitating dynamic symbol resolution. Finally, `htable.h` provides a lightweight hash table implementation for efficient event name lookups.

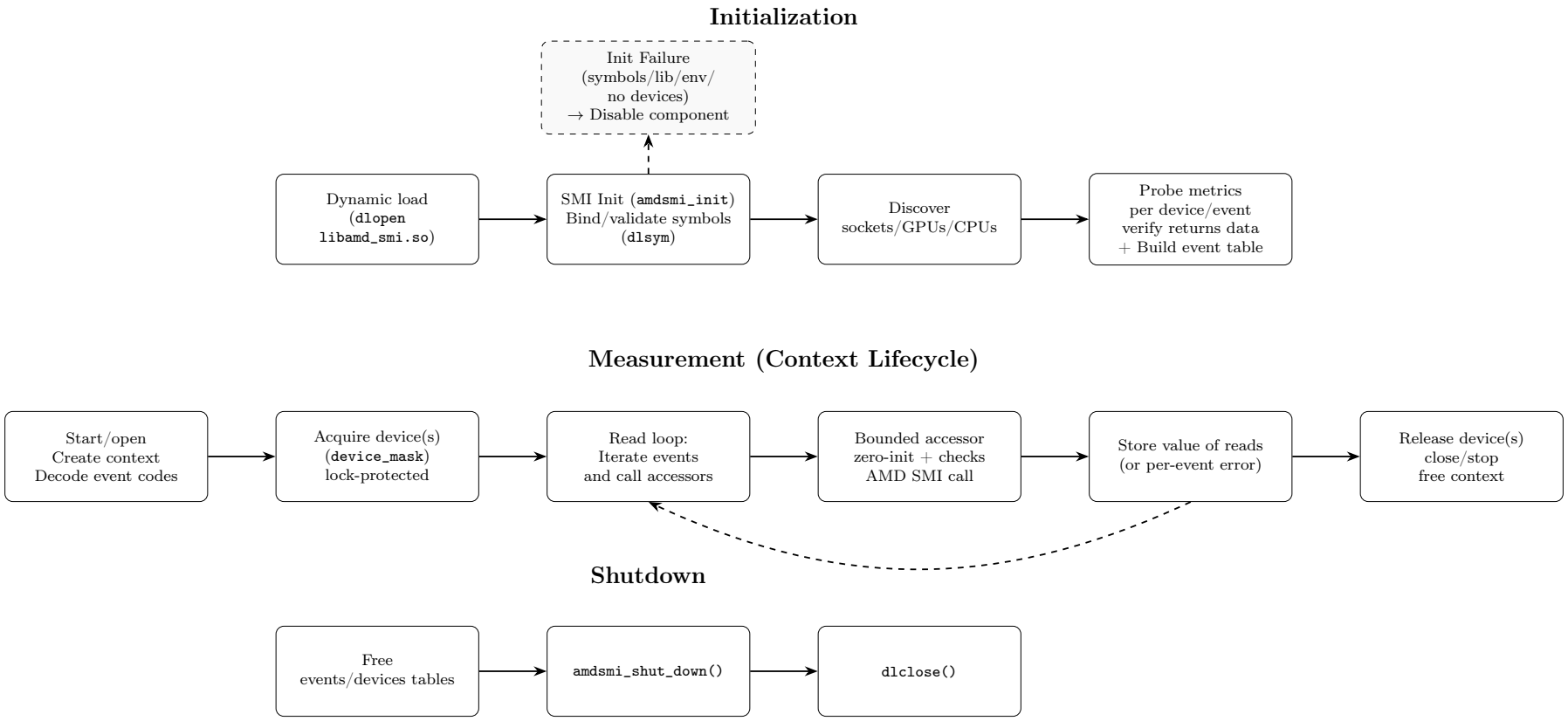

Figure 3.2: Initialization, measurement, and shutdown flows for the AMD SMI component.

3.5 Summary

This chapter presented the design of a robust, low-overhead interface for AMD GPU performance metrics. The architecture addresses API variations through dynamic symbol resolution and probe-based metric discovery, ensuring compatibility across diverse driver versions and hardware generations. By implementing efficient, bounded accessors and thread-safe concurrency controls, the design achieves safe, high-performance sampling suitable for continuous monitoring in heterogeneous HPC environments.

Chapter 4

Performance Validation, Measurement, and Analysis

This chapter evaluates the low-overhead, standardized interface design by monitoring AMD GPU metrics and examining its behavior across devices and ROCm versions. The chapter addresses three primary questions, namely:

(1) **Overhead:** Evaluate whether the interface read path introduces measurable per-call latency relative to direct AMD SMI calls.

(2) **Portability:** Verify that the interface builds, enumerates, and reads supported metrics across AMD GPU generations and ROCm releases without code changes.

(3) **Monitoring interval:** Determine whether shorter polling intervals yield additional distinct metric updates with limited measurement overhead.

Collectively, the experiments in this chapter validate low overhead, portability, and identify practical polling intervals.

4.1 Equivalence in Call Time

4.1.1 Experimental Setup

This experiment measures and compares the time required to read hardware metrics through the standardized interface and directly through the AMD SMI library. The test platform is an MI300A APU running ROCm 7.0.1. For each metric m in a representative set and for each source $s \in \{\text{standardized interface}, \text{direct AMD SMI API}\}$, a measurement loop runs on a pinned CPU core as follows:

1. Initialize source s (standardized interface or AMD SMI) for metric m.
2. Perform a short warmup of two reads (results discarded) for metric m with source s.
3. Record the elapsed time for 500 iterations of reading m from source s using a monotonic clock.

For each metric and source, the mean and standard deviation of 500 per-iteration call times are computed. The test uses a representative set of 32 metrics covering a broad range of categories: clock frequencies, power and energy, thermal sensors, utilization counters, and interconnect/link health. These metrics include both very fast calls (approximately 25–35 µs per read) and slower calls that can take on the order of a few milliseconds for certain queries. By keeping the code path and sampling routine identical for both the standardized interface and direct AMD SMI library, this setup isolates the contribution of the interface's read path to the per-call latency.

4.1.2 Statistical Methodology

Equivalence is assessed using Two One-Sided Tests (TOST) applied to the per-metric mean call times (Schuirmann, 1987). Since metric latencies span a wide range (25 μs to 1.6 ms), a simple difference analysis would be dominated by slower hardware queries.

Therefore, the log-ratio of execution times ($r_m = \log(\bar{t}_m^{SI}/\bar{t}_m^{SMI})$) is analyzed to normalize overhead impact across metrics. An equivalence margin of $\pm 2\%$ is defined, asserting that the interface overhead is negligible if the 98% confidence interval of the geometric mean ratio lies within $[0.98, 1.02]$.

4.1.3 Results

Applying the above equivalence test, the geometric-mean call-time ratio (standardized interface / direct AMD SMI) is

$$\mathbf{1.009} \quad \text{with a } 98\% \text{ confidence interval } [\mathbf{0.999}, \mathbf{1.019}].$$

This interval lies within the pre-specified $\pm 2\%$ equivalence bounds (Table 4.1), indicating that the per-call latency via the standardized interface is statistically indistinguishable from direct AMD SMI calls. In absolute terms, the cross-metric mean call time for direct AMD SMI calls is 371.70 μs, compared to 373.64 μs via the standardized interface—a difference of 1.95 μs (approximately +0.52%).

Table 4.2 and Figure 4.1 show that the 32 metrics cluster into three latency ranges based on the AMD SMI mean call time: a fast group around 25–35 μs (primarily clock-frequency and local temperature queries), an intermediate group around 300–380 μs (engine activity, PCIe counters, power-related metrics, and similar queries), and a slower group around 1.0–1.6 ms (ECC error counters and XGMI bandwidth). The symmetrical profile of Figure 4.1 across these ranges visually confirms that the standardized interface introduces negligible overhead. Across all three latency ranges, calls through the standardized interface have nearly the same duration as direct AMD SMI calls: the per-metric call-time ratio $\bar{t}_m^{SI}/\bar{t}_m^{SMI}$ lies between 0.95 and 1.05 for 29 of the 32 metrics, and even the remaining three metrics deviate by at most 7.4%. These results confirm that the abstraction layer does not introduce significant additional per-call latency relative to a direct AMD SMI call.

Table 4.1: Global equivalence summary pooled across 32 events (500 iterations each).

Quantity	**Estimate**	**98% CI**	**Interpretation**
Geometric-mean ratio (SI/AMD SMI)	1.009	$[0.999, 1.019]$	Within ±2% bounds
Cross-metric mean (AMD SMI)	371.70 μs	—	Absolute reference
Cross-metric mean (SI)	373.64 μs	—	+0.52% vs. SMI

Takeaway. Reads through the standardized interface are equivalent to direct AMD SMI calls within the 2% margin. The geometric mean ratio is 1.009 with a 98% confidence interval of [0.999, 1.019]. This indicates that the interface read path adds no practical per call latency for the tested metrics.

Table 4.2: Read latency per metric: means ± s.d. for the standardized counter interface vs AMD SMI (lower is better). Ratio close to 1 indicates near-native overhead.

Metric	**AMDSMI mean ± s.d.** (μs)	**Standardized interface mean ± s.d.** (μs)	**Ratio** (SI/AMDSMI)
clk_freq_df_current	31.47 ± 55.357	32.29 ± 58.421	1.026
clk_freq_gfx_current	31.55 ± 59.053	31.70 ± 57.642	1.005
clk_freq_mem_current	30.88 ± 51.869	32.61 ± 59.720	1.056
clk_freq_soc_current	30.62 ± 50.837	31.35 ± 55.723	1.024
clk_freq_sys_count	27.89 ± 36.326	29.95 ± 51.203	1.074
clk_freq_sys_current	30.87 ± 55.388	31.76 ± 59.959	1.029
ecc_total_correctable	1000.0 ± 9.8	999.9 ± 6.7	1.000
ecc_total_deferred	1000.0 ± 15.7	999.9 ± 8.8	1.000
ecc_total_uncorrectable	1000.0 ± 8.5	999.9 ± 8.4	1.000
energy_consumed	327.9 ± 151.2	330.4 ± 153.9	1.008
gfx_activity	347.4 ± 193.9	351.8 ± 202.5	1.013

Table 4.2: continued

Metric	**AMDSMI mean ± s.d.** (μs)	**Standardized interface mean ± s.d.** (μs)	**Ratio** (SI/AMDSMI)
gpu_throttle_status	317.9 ± 119.2	319.2 ± 127.5	1.004
mem_total_VRAM	54.39 ± 7.060	54.59 ± 1.898	1.004
mem_usage_GTT	22.14 ± 2.033	22.26 ± 2.922	1.006
mem_usage_VRAM	82.03 ± 9.094	82.08 ± 7.739	1.000
util_counter_gfx	347.0 ± 194.9	345.9 ± 191.7	0.997
pcie_bandwidth	376.4 ± 146.0	375.0 ± 138.9	0.996
pcie_nak_received_count	374.6 ± 142.0	374.7 ± 137.8	1.000
pcie_nak_sent_count	374.7 ± 142.2	374.5 ± 137.0	0.999
pcie_replay_count	375.6 ± 143.8	375.5 ± 139.2	1.000
perf_level	21.29 ± 2.424	21.39 ± 1.316	1.004
power_current	587.4 ± 245.6	616.5 ± 262.8	1.050
power_cap	286.0 ± 188.7	285.6 ± 186.8	0.999
temp_critical_sensor=7	308.5 ± 99.2	320.5 ± 132.8	1.039
temp_current_sensor=1	29.09 ± 42.110	29.33 ± 40.996	1.008
temp_current_sensor=2	29.18 ± 42.229	29.44 ± 42.812	1.009
temp_current_sensor=7	338.3 ± 176.0	340.1 ± 179.5	1.005
temp_emergency_sensor=7	326.2 ± 143.6	309.3 ± 100.4	0.948
temp_max_sensor=7	316.1 ± 120.6	318.5 ± 126.1	1.008
umc_activity	355.3 ± 209.2	345.3 ± 189.6	0.972
xgmi_read_kb	1556.8 ± 186.0	1587.6 ± 217.9	1.020
xgmi_write_kb	1556.8 ± 186.3	1557.7 ± 185.0	1.001

Geometric-mean of mean-ratios across metrics: 1.009.

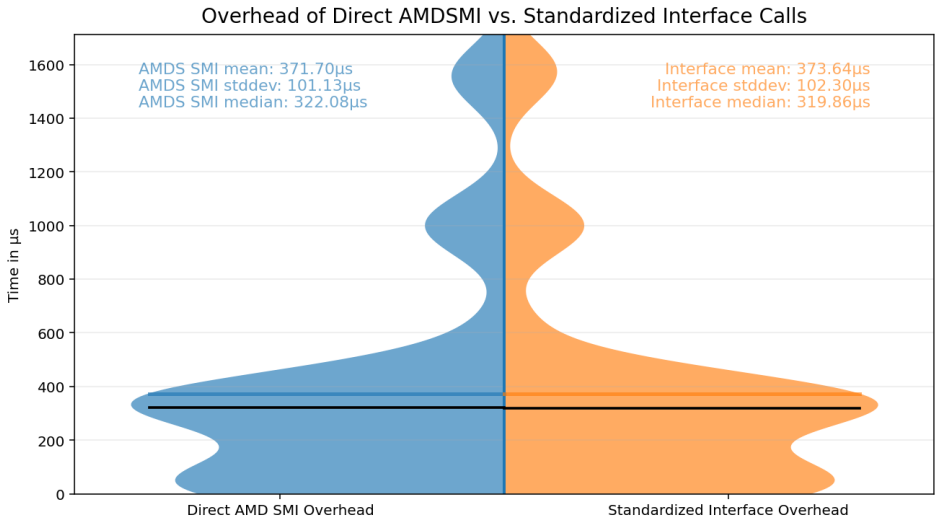

Figure 4.1: Call-time comparison of the Standardized interface versus direct AMD SMI across metrics. With 500 iterations, mirrored distributions remain nearly symmetric for both fast (tens of microseconds) and slower (hundreds to low thousands of microseconds) calls.

4.2 Portability: Devices and Software Versions

4.2.1 Scope and platforms

The evaluation verifies that the interface design and implementation handle differences in available metrics across GPU generations and ROCm releases by compiling, enumerating, and reading metrics on (i) 2 MI210, (ii) 8 MI250X, and (iii) 4 MI300A systems under ROCm 6.4.x, 7.0.x, 7.1.x, and 7.2.x. Only metrics supported by the specific device and ROCm release are counted. Metrics not supported on a given device or ROCm version are not exposed for enumeration or sampling. Table 4.3 shows exposed metric counts for MI300A across ROCm 6.4.0–7.2.0, MI210 on 6.4.0, 7.0.1, and 7.1.1, and MI250X on 6.4.1 and 7.0.2. Portability is achieved through probe-based runtime discovery of metrics, exposing only supported counters. This enumeration validates functional coverage; the next subsection examines how these metrics behave under a controlled workload on the same systems.

Table 4.3: Number of AMD SMI events enumerated by the component on each device/ROCm pair.

Device	ROCm 6.4.x				ROCm 7.0.x		ROCm 7.1.x	ROCm 7.2.x
	6.4.0	6.4.1	6.4.2	6.4.3	7.0.1	7.0.2	7.1.1	7.2.0
4 MI300A	259	259	259	259	342	341	342	342
8 MI250X	N/A	375	N/A	N/A	N/A	356	N/A	N/A
2 MI210	345	N/A	N/A	N/A	333	N/A	333	N/A

Workload response, precision, and consistency (GEMM)

Setup. Having established metric coverage across devices and ROCm versions in Table 4.3, this experiment evaluates workload response, precision, and consistency of the exposed metrics of the devices under GEMM workloads. A reference GEMM kernel (M=14592, K=65536, N=14592) executes three on/off cycles on each system. Metrics sampled are graphics engine (GFX) activity, unified memory

controller (UMC) activity, power, edge/junction/PCIe switch (PLX) temperatures, and compute unit (CU) occupancy, subject to support and system permissions. In addition to the reference kernel, a rocBLAS GEMM kernel with dimensions M=14592, K=131072, N=14592 is also run on the three systems. In this context, precision refers to low variability in metric values during steady-state phases and reproducibility of the traces across the three kernels. Consistency refers to the expected temporal ordering under load, where activity rises first, power follows within the same kernel, and temperatures respond last after a thermal lag. The resulting graphs are shown in Figs. 4.3–4.7.

MI300A (ROCm 7.0.1). For the reference kernel, GFX activity saturates near 100% with high-frequency modulation. Power follows and peaks around 550 W. Edge and junction temperatures rise and form plateaus. UMC activity remains low (about 0–5%). Compute unit occupancy climbs toward the device maximum of 228 compute units. The rocBLAS kernel shows the same sequence with heavier memory traffic (UMC about 45–55%), higher power (about 740–750 W), and modest compute unit occupancy (about 28 compute units). Temperature levels do not plateau within the rocBLAS kernels. The plateaus and repeated cycles indicate high measurement precision (stable values and repeatable traces), and the activity → power → temperature ordering confirms consistency (Figs. 4.2, 4.3).

MI210 (ROCm 7.0.1). For the reference kernel, GFX activity approaches 100% but shows frequent short drops within each kernel execution. Power rises and holds near 210 W. Edge and junction temperatures increase and approach plateaus only in later kernels. UMC activity varies widely between about 20–35%. Compute unit occupancy spans roughly 60–90 compute units with intermittent zeros. With the rocBLAS kernel, GFX activity again saturates, UMC rises to 40–55%, power peaks near 210 W, temperatures rise without clear plateaus, and compute unit occupancy is low at about 13 compute units. Relative to the MI300A and MI250X, MI210 temperatures plateau only after multiple kernels, and the per-metric traces show

larger fluctuations that reduce measurement precision. The activity → power → temperature ordering is consistent (Figs. 4.4, 4.5).

MI250X (ROCm 7.0.2). For the reference kernel, GFX activity holds near 100%. Power rises at kernel start and remains steady around 230 W. Temperatures rise and then plateaus after the first kernel. UMC activity fluctuates around 20–25%. For the rocBLAS kernel, the ordering is preserved while memory traffic increases (UMC about 60–70%) and power peaks near 250 W with decay between kernels. CU occupancy is unavailable on this platform. Repeated plateaus and matching sequences indicate high measurement precision and consistent activity → power → temperature ordering (Figs. 4.6, 4.7).

Takeaway. Across MI210, MI250X, and MI300A under ROCm 6.4.0–7.2.0, probe based discovery enables exposing only metrics supported by the current device and software (Table 4.3). This design requires no code changes as ROCm and hardware capabilities vary. For example, MI300A exposes 259 events under ROCm 6.4.x and 341–342 events under ROCm 7.0.x, while MI250X exposes 375 events under ROCm 6.4.1 and 356 events under ROCm 7.0.2.

Across devices and ROCm versions, the GEMM graphs repeat across the three on/off cycles and preserve the same activity → power → temperature sequence. The main difference is MI210, which shows frequent short GFX dropouts within each kernel and noisier UMC and CU-occupancy traces (including intermittent zeros). Its temperatures also stabilize later than MI250X and MI300A. Separately, under rocBLAS, UMC activity increases on all platforms, but CU occupancy remains low on the MI210 and MI300A even when GFX activity is near 100% and power rises. In contrast, the reference kernel shows higher CU occupancy on these platforms. MI250X does not report CU occupancy on this system. The key result is that the standardized interface captures both the shared response pattern and the device-specific differences in the same metric space, producing comparable, reproducible time series across platforms and software releases.

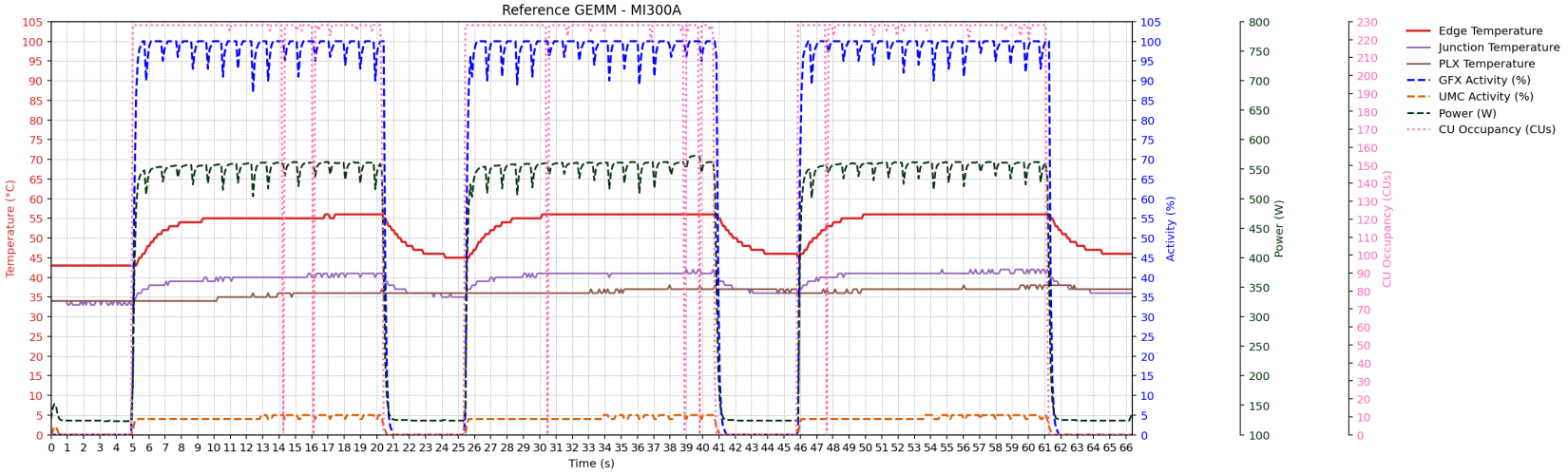

Figure 4.2: MI300A, reference GEMM kernel (M=14592, K=65536, N=14592) with ROCm 7.0.1. GFX activity (blue dashed) reaches $\approx 100\%$ in each kernel and shows a repeatable high-frequency oscillation. Power (green dashed) follows GFX activity and peaks near 550 W. Edge (red) and junction (purple) temperatures increase after kernel start and form plateaus. PLX temperature (brown) is nearly constant. UMC activity (orange dashed) remains low ($\sim$ 0–5%). CU occupancy (pink dotted) increases toward the device maximum of 228 CUs. Temporal ordering is activity $\rightarrow$ power $\rightarrow$ temperature.

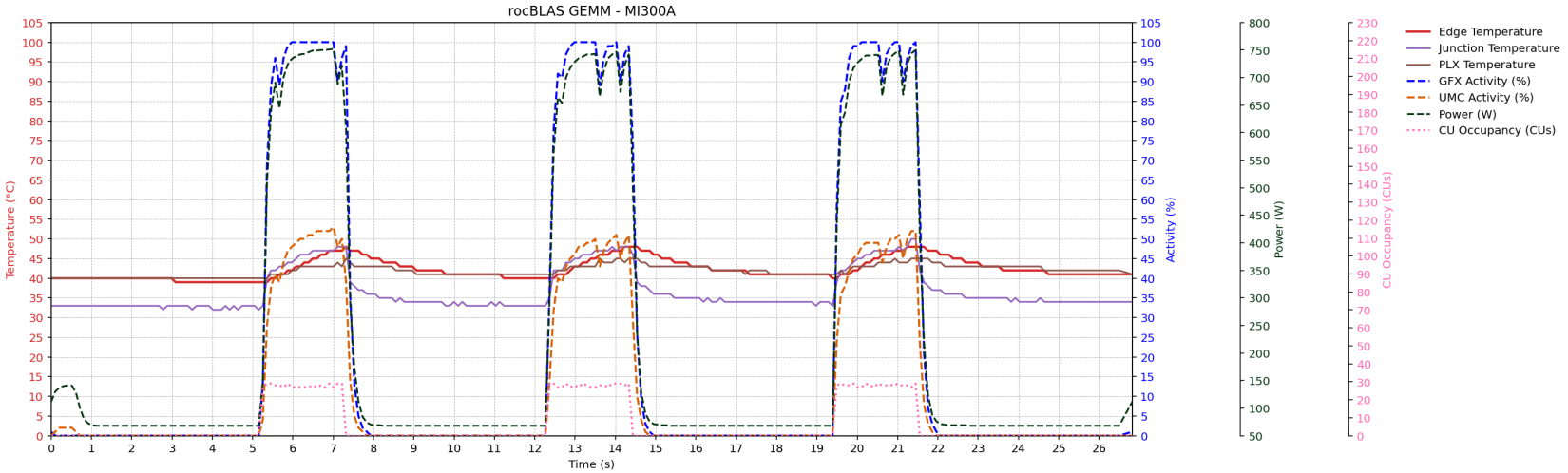

Figure 4.3: MI300A, rocBLAS GEMM kernel (M=14592, K=131072, N=14592) with ROCm 7.0.1. GFX activity saturates near 100% in each kernel. UMC activity rises to roughly 45–55%, indicating higher memory traffic than the reference kernel. Power follows GFX activity and peaks around 740–750 W. Edge and junction temperatures increase and remain below the levels in the reference kernel. CU occupancy (pink dotted) peaks at $\approx$ 28 CUs, well below the device maximum of 228 CUs. Temporal ordering is activity $\rightarrow$ power $\rightarrow$ temperature.

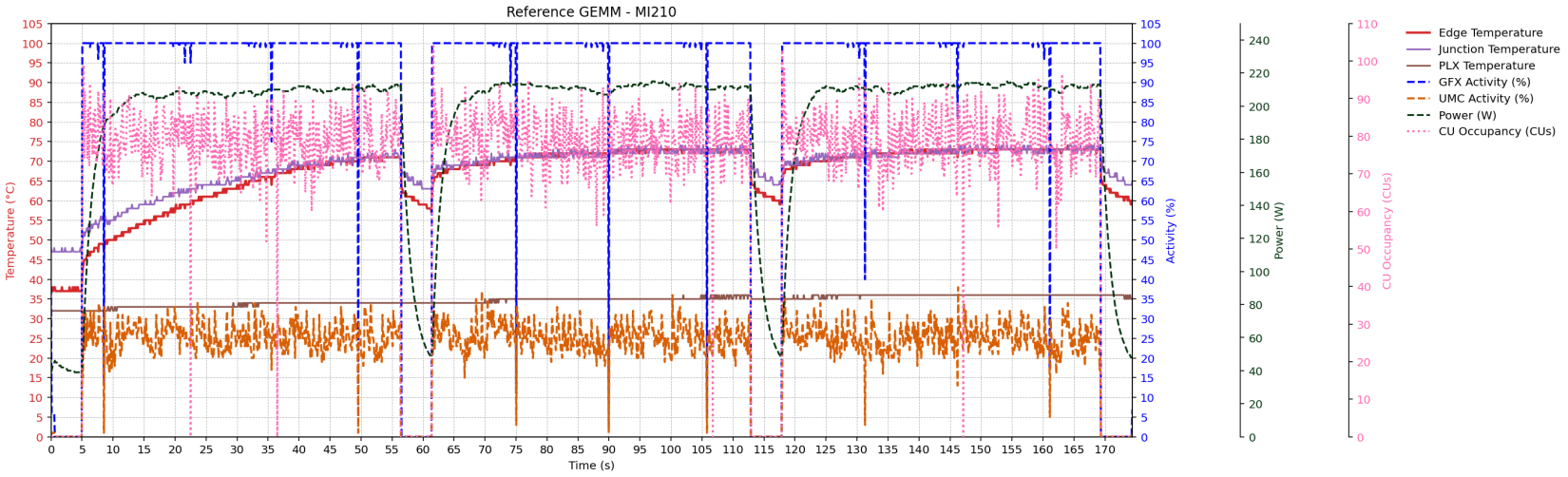

Figure 4.4: MI210, reference GEMM kernel (M=14592, K=65536, N=14592) with ROCm 7.0.1. GFX activity (blue dashed) reaches $\approx 100\%$ in each kernel but shows frequent short drops. Power (green dashed) rises at kernel start and plateaus near 210 W. Edge (red) and junction (purple) temperatures increase and approach plateaus only in later kernels; PLX temperature (brown) is nearly constant. UMC activity (orange dashed) varies widely ($\sim$ 20–35%). CU occupancy (pink dotted) fluctuates broadly, typically 60–90 CUs with intermittent zeros. Temporal ordering is activity $\rightarrow$ power $\rightarrow$ temperature.

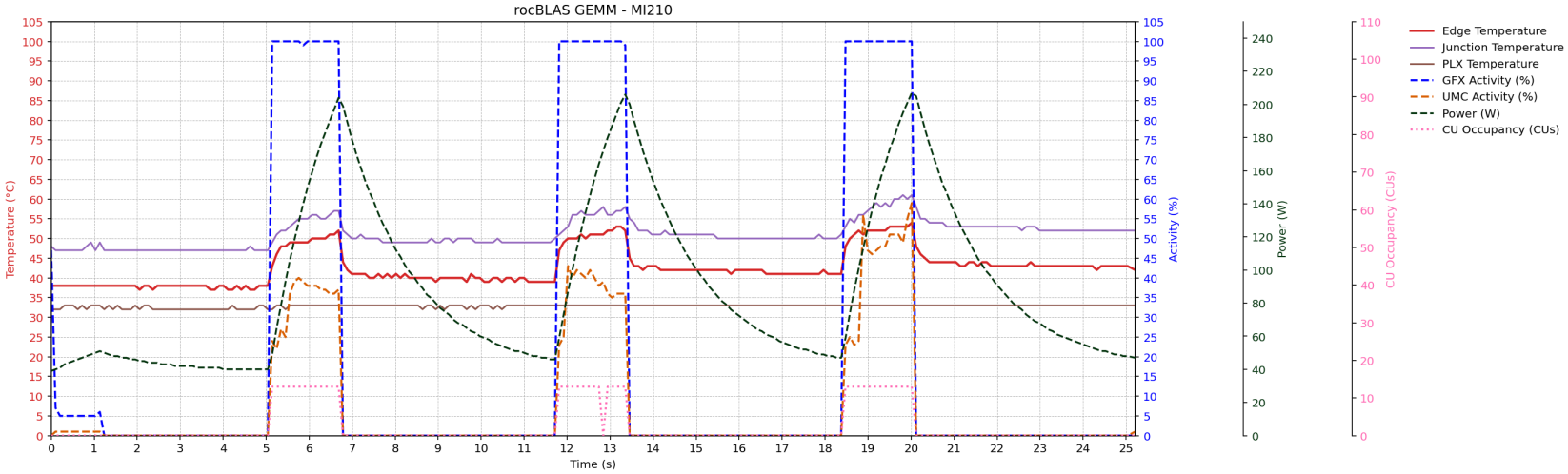

Figure 4.5: MI210, rocBLAS GEMM kernel (M=14592, K=131072, N=14592) with ROCm 7.0.1. GFX activity reaches $\approx 100\%$ in each kernel. UMC activity rises to 40–55%. Power increases gradually and peaks near 210 W. Edge and junction temperatures increase but do not form clear plateaus. CU occupancy (pink dotted) reaches and holds around 13 CUs during the kernels. Temporal ordering is activity $\rightarrow$ power $\rightarrow$ temperature.

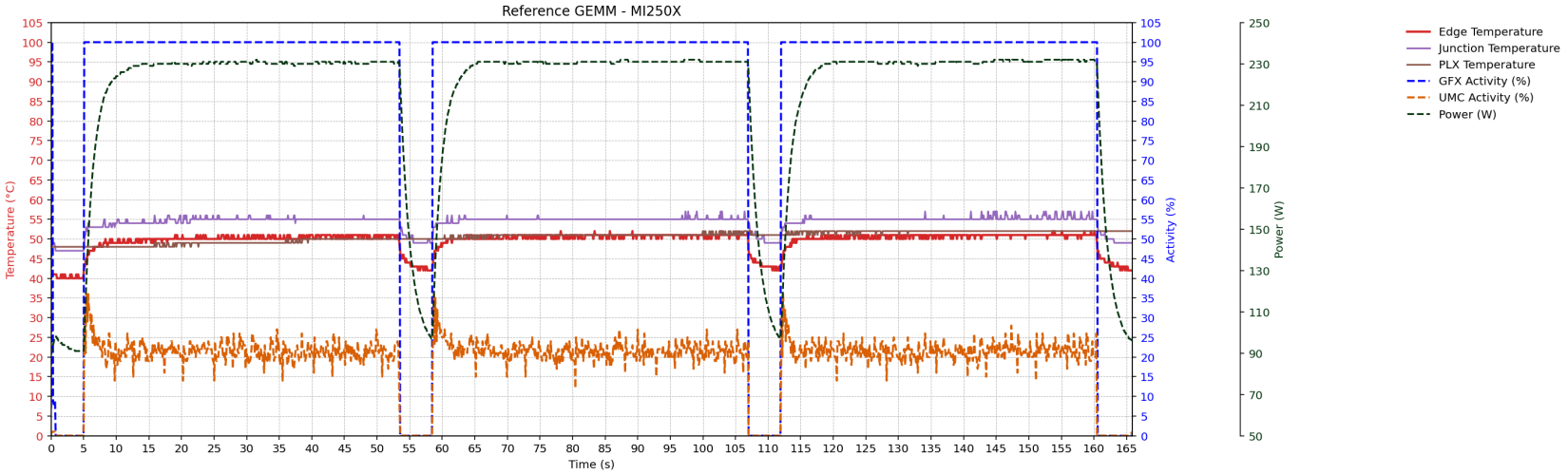

Figure 4.6: MI250X, reference GEMM kernel (M=14592, K=65536, N=14592) with ROCm 7.0.2. GFX activity (blue dashed) holds at $\approx 100\%$ for each kernel. Power (green dashed) rises at kernel start and remains steady for the duration. Edge (red) and junction (purple) temperatures increase on the first kernel and then plateaus. UMC activity (orange dashed) fluctuates around $\sim$ 20–25%. PLX temperature (brown) is nearly constant. Temporal ordering is activity $\rightarrow$ power $\rightarrow$ temperature.

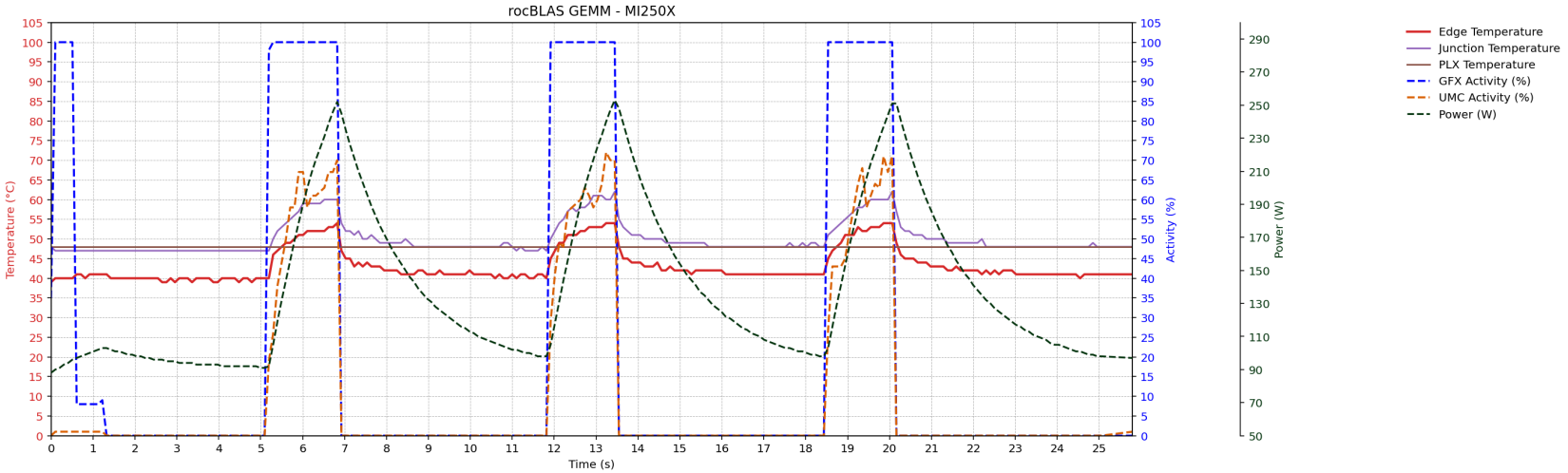

Figure 4.7: MI250X, rocBLAS GEMM kernel (M=14592, K=131072, N=14592) with ROCm 7.0.2. GFX activity (blue dashed) saturates near 100% in each kernel. UMC activity (orange dashed) rises to $\sim$ 60–70%, indicating heavier memory traffic than the reference kernel. Power (green dashed) increases with activity and peaks at $\approx$ 250 W before decaying between kernels. Edge (red) and junction (purple) temperatures rise during each kernel with no sustained plateau. PLX temperature (brown) remains consistent. Temporal ordering is activity $\rightarrow$ power $\rightarrow$ temperature.

4.3 Monitoring Interval and Usable Update Limit

4.3.1 Experimental setup

This experiment examines how the requested monitoring interval affects per-call overhead and the number of observable updates for a single power metric. The standardized interface samples `power_current` on an MI300A GPU under ROCm 7.0.1 at intervals $I \in \{1, 2, 5, 10, 20, 50, 100\}$ ms. For each I, the reference GEMM kernel runs for 100 s in repeating phases of approximately 5 s rest, 10 s load (GEMM), and 5 s rest (20 s per cycle). A sample is counted as a change only if it differs from the previous value by at least one unit in the metric's native unit (1 W for `power_current`).

4.3.2 Methodology

For each requested interval I, per-read wall time is defined as the duration of a single call that reads `power_current` from the standardized interface:

$$t_{\text{call}} = t_{\text{after}} - t_{\text{before}}.$$

The measurement uses a monotonic clock and excludes sleeping. For each I, Table 4.4 reports the total number of reads completed in 100 s, the mean ± s.d. per-call wall time, the total time spent inside read calls, number of changes, and the effective interval, which indicates how closely the sampler tracks the requested cadence.

4.3.3 Results

Table 4.4 shows that the mean per-call wall time lies between 0.59 and 0.85 ms across all requested intervals, with per-call standard deviations of 0.17–0.19 ms. At a requested 1 ms interval, the effective interval is 1.00 ms and about 58.93 s of the 100 s run are spent in read calls, leaving $\approx$ 41.07 s outside the measurement routine. As the requested interval increases, the time spent in reads decreases (e.g., 39.26 s at

Table 4.4: Monitoring-interval summary for `power_current` on MI300A (ROCm 7.0.1). For each requested interval, the table reports the number of reads completed in 100 s, the mean ± s.d. per-call wall time, the total time spent inside read calls, the effective interval (duration / reads), and the number of samples whose value differed from the previous sample by at least 1 W.

Interval (ms)	Reads	Mean ± s.d. (μs)	Time in reads (s)	Eff. interval (ms)	Changes (count)
1	100000	589.33 ± 193.69	58.93	1.00	6753
2	50000	785.27 ± 176.53	39.26	2.00	5854
5	20000	792.63 ± 186.40	15.85	5.00	3644
10	10000	799.71 ± 183.06	8.00	10.00	2360
20	5000	798.92 ± 177.15	3.99	20.00	1474
50	2000	841.12 ± 167.66	1.68	50.00	830
100	1000	854.48 ± 168.04	0.85	100.00	457

2 ms, 15.85 s at 5 ms, 8.00 s at 10 ms), while the per-call standard deviation remains in a narrow 0.17–0.19 ms band across all intervals.

To quantify the trade-off between sampling overhead and measurement granularity, Table 4.4 reports change counts for `power_current`. The number of observed changes decreases with longer intervals: 6,753 (1 ms), 5,854 (2 ms), 3,644 (5 ms), 2,360 (10 ms), 1,474 (20 ms), 830 (50 ms), and 457 (100 ms). Thus, 10–20 ms captures between 1,474 and 2,360 power updates over 100 s while using only 3.99–8.00 s inside read calls, whereas 50–100 ms still records hundreds of changes with less than 2 s of total sampling time.

Takeaway. For the monitored metric (`power_current`) on MI300A, 1–2 ms sampling yields the largest number of observed changes (6,753–5,854 over the run) but requires 58.93–39.26 s of elapsed time inside read calls. Requesting 10–20 ms reduces time in reads to 8.00–3.99 s while still capturing 2,360–1,474 changes, which is a practical trade-off between sampling frequency and sampling overhead on this platform. Shorter intervals provide higher sampling frequency at the cost of proportionally more time spent in the sampling routine, whereas longer intervals reduce sampling overhead but yield fewer distinct observations.

4.4 Summary

This chapter evaluated the standardized AMD GPU monitoring interface with respect to per-call overhead, portability, and usable monitoring intervals. On MI300A, a TOST equivalence test with a $\pm 2\%$ margin on the log ratio of mean call times showed that reads through the standardized interface and direct AMD SMI library calls have statistically equivalent latency. The geometric-mean ratio of call times is 1.009 with a 98% confidence interval of [0.999, 1.019].

Probe-based discovery allows the component to build, enumerate, and read metrics on MI210, MI250X, and MI300A across ROCm 6.4.0–7.2.0 while exposing only supported metrics with consistent names and units. Under reference and rocBLAS GEMM workloads, the interface produces repeatable activity, power, and temperature traces on all three devices. The monitoring-interval experiment on MI300A relates requested sampling intervals to both time spent in read calls and the number of distinct updates. Intervals of 1–2 ms yield the most changes but use a large share of the run inside the sampler, whereas 10–20 ms intervals reduce time in reads while still capturing thousands of updates. Intervals of 50–100 ms observe hundreds of updates with very low sampling overhead. Overall, the interface introduces no practically significant per-call overhead relative to AMD SMI, remains portable across AMD devices and ROCm versions, and supports monitoring intervals that can be chosen to balance sampling frequency against time spent in the measurement routine.

Chapter 5

AMD SMI and ROCm SMI Metric Coverage

This chapter compares the GPU monitoring capabilities of AMD's System Management Interface (AMD SMI) with the legacy ROCm SMI, as integrated in PAPI. The comparison evaluates coverage, granularity, and scope across seven domains: device identification and static properties, performance state and power management, thermal and memory subsystems, utilization and interconnects, process-level monitoring, reliability features, and low-level link diagnostics.

AMD SMI is the officially supported successor to ROCm SMI Advanced Micro Devices, Inc. (2023). On an MI300A platform with ROCm 7.0.1, the AMD SMI interface exposes 342 device-unique metrics compared to 80 in the ROCm SMI interface, a substantial expansion in observable hardware metrics. Beyond overlapping core metrics (temperature, power, clocks), AMD SMI provides detailed topology information, explicit throttle/violation signals, per-process metric attribution, and link-health diagnostics. The sections below pair brief interpretation notes with detailed comparison tables to indicate how the additional coverage can be applied in practice.

5.1 Device Identification and Static Hardware Properties

AMD SMI surfaces static architectural (e.g., cache sizes and instances; VRAM type, bus width, vendor, and capacity) and platform placement details (e.g., NUMA and KFD node IDs), enabling hardware-aware analysis and placement. It also reports static PCIe capabilities (maximum link speed/width and interface version), which provide baselines for subsequent link-health checks (Section 5.7). Both interfaces expose device names and driver version information. AMD SMI additionally provides string fields in event descriptions (e.g., `board_product_name_hash`, `driver_version_hash`).

In practice, these descriptors support: (i) correlating measurements with OS tooling (`lspci`, DRM minors) via BDF components; (ii) NUMA-aware scheduling and data placement using `numa_node`/`topo_numa_node`; and (iii) configuration auditing. For example, these metrics make it possible to verify that devices have trained to the expected PCIe interface version and link width. As summarized in Table 5.1, AMD SMI broadens static context beyond ROCm SMI's basic PCI identifiers.

Table 5.1: Comparison of Device and Driver Identification Events

Metric	**AMD SMI**	**ROCm SMI**	**Analysis & Key Differences**
Device and Driver Identification			
Number of Devices	`NUMDevices`	`NUMDevices`	Number of Available Devices.
Device/Vendor ID	`asic_device_id` `asic_vendor_id`	`device_id` `vendor_id`	Standard PCI hardware identifiers.

Table 5.1: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Vendor/Subsystem Names	`vendor_name_hash` `subsystem_name_hash`	`vendor_name` `device_subsystem_name`	Vendor and subsystem names.
Subsystem ID	`subsystem_id` `subsystem_vendor_id`	`subsystem_id` `subsystem_vendor_id`	Interfaces for identifying the specific board partner model.
Unique PCI ID	`gpu_bdfid` and components: `gpu_bdf_domain`, `gpu_bdf_bus`, etc.	`pci_id`	Unique Bus/Device/Function (BDF) identifier. AMD SMI also exposes its individual components.
Unique ID	`uuid_hash`, `uuid_length`	`unique_id`	Unique identifier for the device.
GPU Identifiers	`gpu_id` `gpu_revision` `gpu_subsystem_id`	No	AMD SMI provides specific device, revision, and GPU-centric subsystem IDs separate from the PCI identifiers.

Table 5.1: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
DRM Node	`enum_drm_render`, `enum_drm_card`	`drm_render_minor`	DRM (Direct Rendering Manager) render node minor number, a key identifier in Linux.
GPU Marketing Name	board_product_name_hash	`device_name`	Marketing name of the GPU.
GPU ASIC Revision	`asic_revision`	No	Specific silicon stepping/revision ID.
Compute/Partition Info	`compute_units` `compute_partition_hash` `memory_partition_hash`	No	Number of compute units and partition types.
Kernel Driver Version	driver_version_hash	`driver_version_str`	Version string of the loaded kernel driver.
Driver/Board Hashes	`board_serial_hash` `driver_name_hash` `driver_date_hash`	No	AMD SMI provides hashed strings for board serial and driver details.

Table 5.1: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
SMI Lib Version	`lib_version_major` `lib_version_minor` `lib_version_release`	`rsmi_version`	Library versions.
Platform Metadata	`kfd_id`, `kfd_node_id`, `kfd_current_partition_id`, `numa_node` / `topo_numa_node`, `gpu_virtualization_mode`, `enum_hsa_id`, `enum_hip_id`	No	AMD SMI provides metadata about the platform, including detailed Kernel Fusion Driver (KFD) IDs, NUMA topology, and virtualization status.

Table 5.1: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Static PCIe Capabilities	`pcie_max_-width`, `pcie_-max_speed`, `pcie_slot_-type`, `pcie_-interface_-version`, `pcie_max_-interface_-version`	No	Maximum supported PCIe link capabilities (width, speed, version) and physical slot type.
Cache Details	`L1_*_size` `L2_*_size` `L3_*_size`	No	Detailed sizes, instances, and sharing information for L1, L2, and L3 caches.
VRAM Details	`vram_bus_width` `vram_type` `vram_vendor_id` `vram_size_-bytes`	No	Low-level VRAM hardware details including bus width, type, vendor, and static capacity (`vram_size_bytes`).

5.2 Performance State and Power Management

AMD SMI and ROCm SMI both expose clock, performance state, and power/energy events. As shown in Table 5.2, both report SCLK (graphics), FCLK (data fabric), MCLK (memory), and SOC clocks, plus an overall performance level. AMD SMI additionally provides video/display clocks (VCLK/DCLK), per-clock details (minimum/maximum, lock, deep-sleep), optional overdrive monitoring (frequency bounds and voltage-curve points), and explicit throttling status that explains sub-maximal clocks.

For power, both interfaces report instantaneous power and configurable caps, but with different units (AMD SMI uses W Advanced Micro Devices, Inc. (2025d), ROCm SMI uses µW Advanced Micro Devices, Inc. (2024b)). Both expose cumulative energy (microjoules). AMD SMI further supplies energy-counter metadata (resolution and timestamp), power and thermal violation counters, and a power-management-enabled flag. In practice, these additions help (i) explain why clocks are below their maximum (throttle bits/violation counters), (ii) audit caps consistently across nodes, and (iii) combine power and energy for throughput-per-watt studies.

Table 5.2: Comparison of Performance State Management and Power Management Events

Metric	**AMD SMI**	**ROCm SMI**	**Analysis & Key Differences**
Clocks and Performance States			
Performance Level	`perf_level`	`perf_level`	Reading the power profile.

Table 5.2: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
System Clock	`clk_freq_sys_-current` `clk_freq_gfx_-current`	gpu_clk_freq_-System:current	Measures the main GPU graphics clock (SCLK).
Data Fabric Clock	`clk_freq_df_-current`	gpu_clk_-freq_DataFab-ric:current	Measures the interconnect clock (FCLK).
Memory Clock	`clk_mem_current` `clk_freq_mem_-current`	gpu_clk_freq_-Memory:current	Measures the VRAM clock frequency (MCLK).
SOC Clock	`clk_freq_SOC_-current`	gpu_clk_freq_-SOC:current	Measures the System-on-Chip clock frequency.
Video/Display Clocks	`clk_freq_-vclk0_current` `clk_freq_-vclk1_current` `clk_freq_-dclk0_current` `clk_freq_-dclk1_current`	No	Video (VCLK) and display (DCLK) clock frequencies.

Table 5.2: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Supported Clocks	`clk_freq_*_count` `clk_freq_*_level_*`	`*_:count` `*_:idx=*` `*_:mask`	Both list supported frequencies and selectors. AMD SMI provides explicit counts and per-level events, while ROCm SMI uses a more programmatic structure.
Detailed Clock Info	`clk_*_min/max`, etc.	No	Details like min/max, lock, and deep sleep status, available under both `clk_*` and `clk_freq_*` naming schemes.
Overdrive Clocks	`od_curr_sclk_level`	No	Monitoring "overdrive" (manual overclocking) settings.
Overdrive Freq. Bounds	`od_curr_sclk_min/max` `od_curr_mclk_min/max`	No	Current min/max bounds for both SCLK and MCLK under overdrive.

Table 5.2: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Overdrive Limits	`od_*_limit_-min/max`	No	Extends overdrive support by exposing the allowable min/max frequency limits for manual tuning.
Voltage Curve Points	`volt_curve_-point_*`	No	Frequency/voltage points from the GPU's voltage curve.
GPU Throttle Status	`gpu_throttle_-status` `gpu_indep_-throttle_status` `thermal_-throttle_events`	No	Detailed bitfields and notification events indicating GPU throttle reasons.
Power and Energy			
Power Management Status	`pm_enabled`	No	A flag indicating if power management is enabled.

Table 5.2: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Power Consumption	`power_current(MI300+)` `power_average(Navi + MI200)` `power_limit`	current_socket_power	Units vary: AMD SMI reports current and time-averaged power in watts; ROCm SMI reports current power in microwatts.
Power Cap	`power_cap`	power_cap:sensor=*	Both allow reading and setting the configurable power limit.
Power Cap Details	`power_cap_range_min/max` `power_cap_default` `power_cap_dpm`	power_cap_range_min/-max:sensor=*	Configurable range for the power cap. AMD SMI also exposes DPM and default caps.
Energy Consumed	`energy_consumed`	`energy_count`	Both provide a cumulative energy counter in microjoules.

Table 5.2: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Energy Counter Details	`energy_resolution` `energy_timestamp`	No	Metadata for the energy counter, including its resolution and last timestamp.
Power/Thermal Violations	`*_violation_acc` `*_violation_pct` `*_violation_active`	No	Counters for package power (ppt), socket thermal, and voltage regulator (vr) thermal violations.

5.3 Thermal and Memory Subsystem Monitoring

Both interfaces provide thermal sensors and memory usage metrics (Table 5.3). AMD SMI adds per-sensor hysteresis, offsets, and history (minimum/maximum), which can aid in control loops and diagnostics (e.g., identifying oscillatory behavior or persistent hotspots). Temperature is provided in degrees Celsius, converted from the default millicelsius value Advanced Micro Devices, Inc. (2025c).

For memory, both interfaces expose total and used VRAM. AMD SMI also includes MB-denominated convenience fields and a theoretical peak VRAM bandwidth. The latter enables quick checks for memory-bound kernels. Both interfaces provide GTT totals/usage for pinned host memory, which allows attributing memory consumption separately to VRAM and host memory.

Table 5.3: Comparison of Thermal and Memory Subsystem Events

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Temperature Monitoring			
Current Temperature	temp_current_-sensor=*	temp_current:sensor=*	Temperature in Celsius, converted from the default millicelsius value Advanced Micro Devices, Inc. (2025c). AMD SMI may expose more sensors.
Temperature Limits	temp_critical_-sensor=* temp_emergency_sensor=*	temp_critical:sensor=* temp_emergency:sensor=*	Report critical and emergency temperature thresholds.
Advanced Temp Metrics	`temp_max`, `temp_min`, `temp_*_hyst`, `temp_offset`, `temp_lowest`, `temp_highest`, `temp_shutdown_-sensor`	No	These advanced fields are sensor-dependent (e.g., available for PLX on sensor 7) and not guaranteed for every sensor.
Memory Monitoring			

Table 5.3: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences

VRAM Usage	`mem_total_VRAM`
	`mem_usage_VRAM`
	`vram_total_mb`
	`vram_used_mb`
		`mem_total_VRAM`
		`mem_usage_VRAM`
			Total and used VRAM in bytes. AMD SMI also provides MB-denominated convenience versions.

VRAM Max Bandwidth	`vram_max_bandwidth`	No	Theoretical maximum VRAM bandwidth.
Visible VRAM Usage	`mem_total_VIS_VRAM` `mem_usage_VIS_VRAM`	`mem_total_VIS_VRAM` `mem_usage_VIS_VRAM`	Visible portion of VRAM (the PCIe BAR).
GTT Memory Usage	`mem_total_GTT` `mem_usage_GTT`	`mem_total_GTT` `mem_usage_GTT`	Graphics Translation Table memory usage.

5.4 Utilization, Interconnects, and Device Metadata

ROCm SMI provides device-wide utilization metrics (`busy_percent`, `memory_busy_percent`). AMD SMI adds per-engine breakdowns—`gfx_activity` (compute/graphics), `umc_activity` (memory controller), `mm_activity` (multimedia)—and coarse-grain activity counters. These metrics allow monitoring of bottlenecks (compute vs. memory vs. multimedia) and can guide kernel scheduling or data movement strategies.

For interconnects, AMD SMI provides XGMI bandwidth with topology data (hops, weights, link types) and per-link status, enabling topology-aware placement (e.g., pairing ranks across low-hop links). PCIe bandwidth is available as instantaneous and accumulated forms (Mb/s or GB/s, depending on the event Advanced Micro Devices, Inc. (2025a)), which helps detect host-device transfer bottlenecks and validate overlap strategies. Firmware queries report per-block firmware versions, and metrics-header queries provide version information for the metrics data structures.

Table 5.4: Comparison of Utilization, Interconnect, and Firmware Events

Metric	**AMD SMI**	**ROCm SMI**	**Analysis & Key Differences**
Utilization and Links			
Engine-Level Activity	`gfx_activity` `umc_activity` `mm_activity`	`busy_percent` `memory_busy_percent`	Per-engine vs. device-wide: `gfx/umc/mm` give compute, memory-controller, and multimedia utilization; `busy_percent` is overall GPU busy, `memory_busy_percent` returns the percentage of time that any device memory is being used.

Table 5.4: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Coarse Grain Utilization	`util_counter_gfx` `util_counter_mem` `util_counter_dec`	No	Coarse-grained activity counters (including for decoder) for low-overhead utilization tracking.
Number of XCD	`xcd_counter`	No	Number of Accelerator Complex Die.
PCIe Bandwidth	`pcie_bandwidth` (inst., Mb/s) `pcie_bandwidth_inst` (inst., GB/s) `pcie_bandwidth_acc` (acc., GB)	No	PCIe bandwidth counters.

Table 5.4: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
XGMI Link Details	`xgmi_read_kb`, `xgmi_write_kb`, `xgmi_hive_id`, `xgmi_node_id`, `xgmi_lanes`, `xgmi_max_-bandwidth`, `xgmi_bit_rate`, `xgmi_min/max_-bandwidth_*`	min_xgmi_-internode_-bw:target=* max_xgmi_-internode_-bw:target=*	Detailed topology and throughput metrics (hops, link type, nearest-device sets; per-link bit rate and read/write counters in KB) Advanced Micro Devices, Inc. (2025b,g,f).

Table 5.4: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Topology & Fabric Details	`link_hops_-peer=*` `link_weight_-peer=*` `link_type_-peer=*` `xgmi_nearest_-count` `pcie_nearest_-count` `not_-applicable_-nearest_count` `unknown_-nearest_count` `xgmi_link_-status_link=*`	No	Topology information, including hops, weights, and link types between peer devices, nearest neighbor counts, and per-link XGMI status.
Firmware and Metadata			
Firmware Versions	`fw_version_*_*`	firmware_version:block=*	Firmware versions of various IP blocks on the GPU.

Table 5.4: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Metrics Header Info	`metrics_-header_*`	No	Exposes versioning information for the metrics data structures like header size, format revision, and header content revision.

5.5 Process-Level Monitoring

AMD SMI adds per-process monitoring that is absent in PAPI's ROCm SMI interface. This enables per-process resource accounting in shared GPU environments and application-level profiling: attributing memory by type (VRAM, GTT/pinned, CPU), engine time (GFX/ENC), and compute-unit occupancy to a specific process. In practice, these events help (i) identify noisy neighbors in shared environments, (ii) correlate application phases with GPU usage, and (iii) build job-level dashboards with explicit attribution (Table 5.5).

Table 5.5: Comparison of Per-Process Monitoring Events (AMD SMI Only)

Metric	AMD SMI	ROCm SMI	Analysis
Per-Process Monitoring			
Process Count	`process_count`	No	Number of Processes.

Table 5.5: continued

Metric	AMD SMI	ROCm SMI	Analysis
Process PID	`process_pid`	No	Enumerates active GPU-bound processes.
Process memory (total, bytes)	`process_mem`	No	Total GPU-attributed memory for the PID.
VRAM memory (bytes)	`process_vram_mem`	No	Per-process dedicated VRAM footprint.
GTT / host-pinned memory (bytes)	`process_gtt_mem`	No	Host-pinned (GTT) memory mapped by the GPU.
CPU memory (bytes)	`process_cpu_mem`	No	CPU-side memory attributed to the process by AMD SMI.
GFX engine time (ns)	`process_eng_gfx`	No	Accumulated wall-time on the graphics/compute engine.
ENC engine time (ns)	`process_eng_enc`	No	Accumulated wall-time on the encoder engine.
Compute unit occupancy	`process_cu_occupancy`	No	Instantaneous CU utilization for the process.

Table 5.5: continued

Metric	AMD SMI	ROCm SMI	Analysis
Process isolation	`process_isolation`	No	Indicates whether per-process isolation is enabled.

5.6 Reliability, Availability, and Serviceability (RAS) Features

ROCm SMI primarily exposes Error Correction Code (ECC) enablement via a block-level mask. AMD SMI extends this by surfacing overall and per-block error counters (correctable/uncorrectable/deferred), per-block ECC status codes and RAS block enable state, an ECC correction mask that controls which blocks are subject to correction, and RAS Electrically Erasable Programmable Read-Only Memory (EEPROM) version/validity (Tables 5.6 and 5.7). In practice, these metrics enable proactive stability management: trending correctable error rates, detecting hotspots at the block level (e.g., UMC vs. GFX), and scheduling maintenance before uncorrectable faults impact jobs.

Table 5.6: Comparison of General RAS and ECC Features

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
ECC Enabled Mask	`ecc_enabled_mask`	`ecc_enabled_get`	Mask indicating which hardware blocks have ECC enabled.

Table 5.6: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
ECC Correction Mask	`ecc_correction_mask`	No	AMD SMI specific mask for ECC correction features.
RAS EEPROM Status	`ras_eeprom_version` `ras_eeprom_valid`	No	Exposes EEPROM status for RAS, including version and validity.
RAS Block State	`ras_block_*_state`	No	Per-block state indicating if RAS is enabled for components like UMC, SDMA, and GFX.
ECC Block Status	`ecc_*_status`	No	Per-block status codes for ECC operation on components like UMC, GFX, and DF.

Table 5.7: Granular Per-Block RAS and ECC Error Counters (AMD SMI Only)

Hardware Block	Correctable Errors	Uncorrectable Errors	Deferred Errors
Overall GPU	`ecc_total_correctable`	`ecc_total_uncorrectable`	`ecc_total_deferred`
GFX Block	`ecc_gfx_correctable`	`ecc_gfx_uncorrectable`	`ecc_gfx_deferred`
SDMA Block	`ecc_sdma_correctable`	`ecc_sdma_uncorrectable`	`ecc_sdma_deferred`
MMHUB Block	`ecc_mmhub_correctable`	`ecc_mmhub_uncorrectable`	`ecc_mmhub_deferred`

5.7 PCIe Link Health Diagnostics

AMD SMI adds low-level PCIe link-health diagnostics not present in ROCm SMI (Table 5.8). Replay and NAK counters surface link-layer errors that reduce effective bandwidth, while L0 to recovery counts reveal retraining events that cause stalls. Combined with current link speed/width, these metrics help distinguish software bottlenecks from signal-integrity or platform issues. In practice, bandwidth and error counters can be recorded before and after runs and correlate spikes in replay/NAK/recovery with observed throughput drops.

Table 5.8: Low-Level PCIe Link Health Diagnostics Exclusive to AMD SMI

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
PCIe Link Health and Stability			

Table 5.8: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Replay Count	`pcie_replay_count` `pcie_replay_rollover_count` `pcie_replay_rollover_count_acc`	No	Counts packet retransmissions and rollovers.
NAKs Sent/Received	`pcie_nak_sent_count` `pcie_nak_received_count` (and `_acc` accumulated variants)	No	Counts Negative Acknowledgments. Direct indicator of link-layer errors being detected.
Link Recovery	`pcie_l0_to_recovery_count` `pcie_other_end_recovery_count` `pcie_l0_to_recovery_count_acc`	No	Counts transitions into the link recovery state. Only the L0 variant has an accumulated counter. Frequent recoveries cause performance stalls.

Table 5.8: continued

Metric	AMD SMI	ROCm SMI	Analysis & Key Differences
Current Link Speed/Width	`pcie_link_speed` (in 0.1 GT/s) `pcie_link_width` (lanes) *Also as:* `pcie_speed` (in MT/s) `pcie_width` (lanes)	No	Health metrics provide context for links; note units: `pcie_link_speed` is in 0.1 GT/s and `pcie_speed` is in MT/s Advanced Micro Devices, Inc. (2025a).

5.8 Summary

Across the seven domains, AMD SMI expands the AMD GPU hardware performance metrics available through PAPI beyond those provided by ROCm SMI. On the MI300A with ROCm 7.0.1 it exposes 342 device-unique metrics—compared to 80 via ROCm SMI—adding richer device and topology context, explicit throttle and violation data, engine and link metrics, per-process attribution, and RAS/PCIe health checks. Within PAPI these features are available through a portable, low-overhead interface for scientific workloads. Available metrics will change as future software and hardware releases. Overall, AMD SMI offers a stronger basis for GPU monitoring.

Chapter 6

Conclusion

This thesis presented the design and evaluation of a portable performance monitoring interface for AMD GPUs that abstracts vendor-specific API differences while maintaining low overhead. Motivated by the deprecation of ROCm SMI, the implementation is built on the AMD System Management Interface (AMD SMI) and integrated into the Performance API (PAPI) component framework. The design features a novel probe-based, automatic discovery mechanism that detects supported metrics at runtime, ensuring consistent access across different GPU models and driver versions.

Experimental results confirm that the interface maintains near-native measurement overhead relative to the vendor API. Statistical validation using Two One-Sided Tests (TOST) demonstrated a geometric mean call-time ratio of 1.009 between the standardized interface and direct AMD SMI calls, verifying that the abstraction introduces negligible latency. This work enables portable access to power, thermal, engine activity, and interconnect metrics for AMD accelerators, reducing the maintenance burden of vendor-specific instrumentation. By enhancing the stability of GPU monitoring, it provides a foundation for cross-platform performance analysis in heterogeneous HPC environments.

Bibliography

Advanced Micro Devices, Inc. (2023). ROCm 5.5.0 Release Notes: Deprecation of ROCm SMI. https://rocm.docs.amd.com/en/docs-5.5.0/release/release_notes.html. "The ROCm SMI tool is in maintenance mode and will be deprecated in a future ROCm release. Users are advised to transition to the AMD SMI tool." Accessed October 13, 2025. 1, 5, 35

Advanced Micro Devices, Inc. (2024a). ROCm SMI Library (RSMI) API Reference: Power Queries — `rsmi_dev_energy_count_get()` (resolution in μJ). https://rocm.docs.amd.com/projects/rocm_smi_lib/en/docs-6.4.2/doxygen/html/group__PowerQuer.html. "counter_resolution ... in micro Joules." Accessed 10 Oct. 2025. 5

Advanced Micro Devices, Inc. (2024b). ROCm SMI Library (RSMI) API Reference: Power Queries — `rsmi_dev_power_ave_get()` (power in μW). https://rocm.docs.amd.com/projects/rocm_smi_lib/en/docs-6.4.2/doxygen/html/group__PowerQuer.html. "writes the current average power consumption (in microwatts)." Accessed 10 Oct. 2025. 5, 41

Advanced Micro Devices, Inc. (2025a). AMD SMI API Reference: PCIe metrics units — `amdsmi_pcie_info_t::pcie_metric_` (MT/s, Mb/s) and `amdsmi_gpu_metrics_t` (0.1 GT/s). https://rocm.docs.amd.com/projects/amdsmi/en/docs-6.4.1/doxygen/docBin/html/structamdsmi_pcie_info_t_1_1pcie_metric_.html. See also https://rocm.docs.amd.com/projects/amdsmi/

en/amd-staging/doxygen/docBin/html/structamdsmi__gpu__metrics__t.html. Accessed 10 Oct. 2025. 5, 49, 59

Advanced Micro Devices, Inc. (2025b). AMD SMI C++ API: Hardware Topology Functions. https://rocm.docs.amd.com/projects/amdsmi/en/docs-7.0.1/doxygen/docBin/html/group__tagHWTopology.html. Includes `amdsmi_topo_get_link_type()` (hops, link type), `amdsmi_topo_get_link_weight()`, `amdsmi_get_link_topology_nearest()` (nearest sets), and `amdsmi_get_link_metrics()`. Accessed 10 Oct. 2025. 51

Advanced Micro Devices, Inc. (2025c). AMD SMI Library API Documentation: amdsmi_get_temp_metric(). <URLtotheROCmdocumentationpage>. Accessed: 2025-10-15. Documentation confirms the temperature output from the function is in Celsius. 46, 47

Advanced Micro Devices, Inc. (2025d). AMD SMI Library API Reference: GPU Monitoring — `amdsmi_get_power_info()` (power in W). https://rocm.docs.amd.com/projects/amdsmi/en/docs-6.1.0/doxygen/docBin/html/group__gpumon.html. "The voltage is in units of mV and the power in units of W." Accessed 10 Oct. 2025. 5, 41

Advanced Micro Devices, Inc. (2025e). AMD SMI Library API Reference: Power Queries — `amdsmi_get_energy_count()` (resolution in μJ). https://rocm.docs.amd.com/projects/amdsmi/en/latest/doxygen/docBin/html/group__tagPowerQuery.html. "counter_resolution ... in micro Joules." Accessed 10 Oct. 2025. 5

Advanced Micro Devices, Inc. (2025f). AMD SMI: `amdsmi_gpu_metrics_t` Struct Reference. https://rocm.docs.amd.com/projects/amdsmi/en/amd-staging/doxygen/docBin/html/structamdsmi__gpu__metrics__t.html. Includes PCIe/XGMI bandwidth fields (e.g., `xgmi_read_data_acc`/`xgmi_write_data_acc`

in KB, `pcie_bandwidth_inst`/`acc` in GB/s) and link-status arrays. Accessed 10 Oct. 2025. 51

Advanced Micro Devices, Inc. (2025g). AMD SMI: `amdsmi_link_metrics_t` (Bandwidth monitor). https://rocm.docs.amd.com/projects/amdsmi/en/docs-6.4.1/doxygen/docBin/html/group__tagBandwidthMon.html. Shows per-link fields: `bit_rate` (Gb/s), `max_bandwidth` (Gb/s), `link_type`, and `read`/`write` totals in KB. Accessed 10 Oct. 2025. 51

Advanced Micro Devices, Inc. (2025h). AMD SMI: `amdsmi_xgmi_link_status_t` Struct Reference. https://rocm.docs.amd.com/projects/amdsmi/en/latest/doxygen/docBin/html/structamdsmi__xgmi__link__status__t.html. Per-link XGMI up/down status array. Accessed 10 Oct. 2025. 5

Atchley, S., Baker, M., Bording, F., Droge, D., Venczel, T., Vose, B., Beck, M., Berrill, M., Bland, A. S., Dropps, F., Fahey, M. R., Jain, N., Kerbyson, D. J., Ladd, J., Levy, M., Moore, S., Murali, A., Rogers, T., Shpiner, V., Taffet, R., and Tallent, N. R. (2022). Frontier: The first us exascale system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '22)*, SC '22. ACM. Article No. 50. 1, 4

Islam, T. Z., Marathe, A., Schutte, H., and Zaeed, M. (2024). Data-driven analysis to understand gpu hardware resource usage of optimizations. *arXiv preprint arXiv:2408.10143*. 1, 4

Jagode, H., Danalis, A., Congiu, G., Barry, D., Castaldo, A., and Dongarra, J. (2024). Advancements of PAPI for the exascale generation. *The International Journal of High Performance Computing Applications*, 38(6):599–616. 1, 4, 5

Schuirmann, D. J. (1987). A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *Journal of Pharmacokinetics and Biopharmaceutics*, 15(6):657–680. 6, 18

Treibig, J., Hager, G., and Wellein, G. (2010). Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 International Conference on Parallel Processing Workshops (PSTI 2010)*, pages 207–216. IEEE. 5

Vita

Dong Jun Woun is an M.S. student in Computer Science at the University of Tennessee, Knoxville. He earned a B.S. in Computer Science (minors in Machine Learning and Cybersecurity) from UT Knoxville in May 2024. In August 2024, he joined the Innovative Computing Laboratory as a Graduate Research Assistant under the supervision of Dr. Heike Jagode; his primary thesis advisor is Dr. Catherine Schuman. His experience includes developing features for the PAPI performance-monitoring tool (C/C++); building a CI/CD pipeline and pytest automation for hardware-in-the-loop testing as a Software Development Engineer Intern with Amazon's Project Kuiper; and accelerating quantum SVM experiments at Oak Ridge National Laboratory (up to 11.1× speedups; see https://arxiv.org/abs/2401.12485). His research interests include high-performance computing, embedded programming, and artificial intelligence.