PR #2696 完整报告

radixark/miles

[dashboard for amd gpu]: add AMD SMI GPU telemetry

合并时间：2026-08-22 09:09

原文链接：http://prhub.com.cn/radixark/miles/pull/2696

执行摘要

- 一句话：Dashboard 新增 AMD SMI 遥测，NVML 优先自动回退
- 推荐动作：值得精读，尤其是 Provider 抽象、单位差异处理与降级语义的设计：_AmdSmiProvider 与 _NvmlProvider 共享同一接口而实现细节差异被干净隔离，_amd_socket_power 对多代 ROCm 字段兼容的思路可复用于其他硬件平台。创建类似硬件适配层时可参考本 PR 的 importlib 延迟导入与 sys.modules 打桩测试方式。

功能与动机

在 AMD/ROCm 节点上，原实现只有 NVML 一条采集路径。PR body 指出，在 4 x MI350X devbox 上跑 Megatron Dashboard 任务时日志出现 NVML Shared Library Not Found，/api/timeline/gpu 返回空 lanes，导致 AMD 集群的 GPU 利用率、显存与功耗在 Dashboard 上完全不可见。该 PR 的目标是在不改变 Dashboard 既有 schema 与前端的前提下，用 AMD SMI 补齐这条遥测链路。

实现拆解

1. 双 Provider 抽象与自动检测：miles/dashboard/gpu_sampler.py 把原先内联在 GpuSampler 里的 NVML 调用整体抽为 _NvmlProvider，并新增同接口的 _AmdSmiProvider，两者统一实现 initialize() -> (handles, uuids)、read_device(handle) -> (util, mem_mb, power_w)、read_processes(handle) -> [(pid, name, mem_mb)]。GpuSampler.__init__ 新增 amdsmi 注入参数，并断言 nvml 与 amdsmi 不能同时注入；_init_nvml 重构为 _init_provider，按“显式注入 > 自动检测”构造候选列表，自动检测时先 NVML 后 AMD SMI，api 为 None 时用 importlib.import_module 延迟加载可选依赖，避免 CPU 节点强依赖 pynvml / amdsmi。
2. AMD 指标归一化与 ROCm 版本兼容：新增 _text 统一 bytes/str 转换；_AmdSmiProvider.read_device 直接取 amdsmi_get_gpu_vram_usage()["vram_used"]（AMD SMI 已是 MiB，不再右移 20 位）；_amd_socket_power 按 socket_power → current_socket_power → average_socket_power 的优先级取值，识别 "N/A" 与 0xFFFF（ROCm 6.0 uint16 哨兵）等不可用标记；read_processes 过滤 vram_mem == 0 的 KFD 旁观进程，使按进程显存与 NVML 的 compute-process-only 语义对齐。
3. 失败隔离与降级语义：_init_provider 逐个尝试候选后端并收集失败原因，区分“单后端失败”与“双后端失败”的 warning 日志；设备级读取异常沿用 RateLimitedWarner 跳过该 tick；AMD 功耗读取失败只把 power_w 降为 0 W，不拖累 util/mem 上报。

4. 测试配套：tests/fast/dashboard/test_gpu_sampler.py 新增 FakeAmdSmi，并把 FakeNvml 的 UUID 与进程名改为返回 bytes 以对齐真实库；新增 _amd_socket_power 参数化用例、自动检测回退（用 sys.modules 打桩）、显式注入不探测另一后端、零设备禁用、缺失可选依赖禁用、KFD 过滤、单设备降级跳过等用例。新增 tests/fast-gpu/test_gpu_sampler_hw.py，通过 register_cuda_ci / register_rocm_ci 分别挂到 stage-b-2-gpu-h200 与 stage-c-4-gpu-mi350，在真实 GPU 上验证自动检测选择匹配后端并上报全部设备。

关键文件：

- miles/dashboard/gpu_sampler.py（模块 遥测采集；类别 source；类型 core-logic；符号 _text, _NvmlProvider, _AmdSmiProvider, _amd_socket_power）：核心改动文件：将单一 NVML 采样路径重构为 NVML 优先、AMD SMI 回退的双 Provider 架构，并新增 _amd_socket_power、_text 等兼容逻辑，是本次功能的全部源码主体。
- tests/fast/dashboard/test_gpu_sampler.py（模块 采样测试；类别 test；类型 test-coverage；符号 FakeAmdSmi, test_amd_sample_once_preserves_native_units_and_uuids, test_amd_socket_power_prefers_current_then_falls_back, test_production_auto_detection_falls_back_from_nvml_to_amdsmi）：单元测试主阵地：新增 FakeAmdSmi 完整模拟 amdsmi 的 dict 形态返回，覆盖功耗字段选择、自动检测回退、KFD 过滤、单设备降级跳过等关键路径，是保证两个后端行为等价的主要手段。
- tests/fast-gpu/test_gpu_sampler_hw.py（模块 硬件测试；类别 test；类型 test-coverage；符号 test_auto_detection_picks_the_backend_matching_the_hardware, test_every_device_reports_telemetry_and_processes, PushSpy）：新增的真实硬件 smoke test：CPU-only 的 fast 测试无法触达真实 vendor 库，此文件注册到 CUDA 与 ROCm 两套 GPU CI fleet，保证 auto-detection 在两个平台上都真实验证。

关键符号：GpuSampler._init_provider, _NvmlProvider.initialize, _NvmlProvider.read_device, _NvmlProvider.read_processes, _AmdSmiProvider.initialize, _AmdSmiProvider.read_device, _AmdSmiProvider.read_processes, _amd_socket_power, _text

关键源码片段

miles/dashboard/gpu_sampler.py

核心改动文件：将单一 NVML 采样路径重构为 NVML 优先、AMD SMI 回退的双 Provider 架构，并新增 _amd_socket_power、_text 等兼容逻辑，是本次功能的全部源码主体。

以下是 GpuSampler 自动检测与 AMD 适配的核心实现。

```
# Provider 层把两家厂商的 SMI 调用统一成三个方法，输出 Dashboard 约定单位：
# initialize() -> (handles, uuids)，read_device(handle) -> (util %, mem MiB, power W)，
# read_processes(handle) -> [(pid, name, mem MiB)]。initialize() 失败即抛异常，
# 供 _init_provider 自动回退到下一个后端。

class _AmdSmiProvider:
    name = "AMD SMI"
    module = "amdsmi"
```

```
    def __init__(self, api):
        self._api = api

    def initialize(self) -> tuple[list, list[str]]:
        self._api.amdsmi_init()
        handles = self._api.amdsmi_get_processor_handles()
        if not handles:
            raise RuntimeError("no AMD SMI devices")
        # SMI 槽位顺序与 Ray 的可见顺序一致；partitioned MI300+ 卡可能重复物理 GPU UUID
        return list(handles), [_text(self._api.amdsmi_get_gpu_device_uuid(handle)) for handle in
        handles]

    def read_device(self, handle) -> tuple[int, int, int]:
        util = int(self._api.amdsmi_get_gpu_activity(handle)["gfx_activity"])
        # 与 NVML 不同，AMD SMI 的 vram_used 已经是 MiB，不能再右移 20 位
        mem_mb = int(self._api.amdsmi_get_gpu_vram_usage(handle)["vram_used"])
        try:
            power_w = _amd_socket_power(self._api.amdsmi_get_power_info(handle))
        except Exception:
            power_w = 0 # 功耗传感器不可用时只降级功耗，不影响 util/mem 继续上报
        return util, mem_mb, power_w

    def read_processes(self, handle) -> list[tuple[int, str, int]]:
        processes = []
        for proc in self._api.amdsmi_get_gpu_process_list(handle):
            # KFD 会把不持有 VRAM 的旁观进程也列出来；NVML 只上报 compute
            # processes，这里过滤 0 显存行以对齐两个后端的语义
            mem_mb = int((proc.get("memory_usage") or {}).get("vram_mem") or 0) >> 20
            if mem_mb == 0:
                continue
            pid = int(proc["pid"])
            name = proc.get("name")
            processes.append((pid, str(name) if name and name != "N/A" else f"pid {pid}", mem_mb))
        return processes


def _amd_socket_power(power: dict) -> int:
    # 兼容多代 ROCm：优先 MI300+ 统一 socket_power 字段，再退回 current/average；
    # 不可用传感器以 "N/A" 或 0xFFFF（ROCm 6.0 uint16 哨兵）表达
    for field in ("socket_power", "current_socket_power", "average_socket_power"):
        value = power.get(field, "N/A")
        if value != "N/A" and 0 <= value < 0xFFFF:
            return int(value)
    raise ValueError(f"AMD SMI socket power unavailable: {power!r}")


# GpuSampler 初始化入口：根据注入参数或自动检测决定使用哪个 vendor backend。
# 生产环境不注入任何 backend 时，先试 NVML，初始化失败再回退 AMD SMI。
def _init_provider(self, *, nvml, amdsmi) -> bool:
```

```
    if nvml is not None:
        candidates = [(_NvmlProvider, nvml)]
    elif amdsmi is not None:
        candidates = [(_AmdSmiProvider, amdsmi)]
    else: # 自动检测：优先 NVML，失败后尝试 AMD SMI
        candidates = [(_NvmlProvider, None), (_AmdSmiProvider, None)]

    failures = []
    for cls, api in candidates:
        try:
            # api 为 None 时按模块名延迟导入，避免在 CPU 节点强制引入 vendor 库
            provider = cls(api if api is not None else importlib.import_module(cls.module))
            self._handles, self._uuids = provider.initialize()
        except Exception as error:
            failures.append((cls.name, str(error)))
            logger.debug("%s unavailable on %s", cls.name, self.node, exc_info=True)
            continue
        self._provider = provider
        return True

    # 单后端与双后端失败分开记录，方便区分“未安装库”与“驱动不匹配”
    if len(failures) == 1:
        name, error = failures[0]
        logger.warning("%s unavailable on %s (%s); GPU utilization will not be collected", name,
        self.node, error)
    else:
        detail = "; ".join(f"{name}: {error}" for name, error in failures)
        logger.warning("GPU telemetry unavailable on %s (%s); GPU utilization will not be
        collected", self.node, detail)
    return False
```

tests/fast/dashboard/test_gpu_sampler.py

单元测试主阵地：新增 FakeAmdSmi 完整模拟 amdsmi 的 dict 形态返回，覆盖功耗字段选择、自动检测回退、KFD过滤、单设备降级跳过等关键路径，是保证两个后端行为等价的主要手段。

以下为 FakeAmdSmi 及其驱动测试：通过 sys.modules 打桩验证生产环境的自动检测回退路径。

```
class FakeAmdSmi:
    """模拟 amdsmi API（ROCm >= 6.1 dict 形态）；handle == 设备索引。"""

    def __init__(self, count=2, *, fail_init=False, failing_devices=()):
        self.count = count
        self.fail_init = fail_init
        self.failing_devices = set(failing_devices)

    def amdsmi_init(self):
        if self.fail_init:
            raise RuntimeError("AMD SMI initialization failed")
```

```
    def amdsmi_get_processor_handles(self):
        return list(range(self.count))

    def amdsmi_get_gpu_device_uuid(self, handle):
        return f"GPU-amd-{handle}"

    def amdsmi_get_gpu_activity(self, handle):
        if handle in self.failing_devices:
            raise RuntimeError("GPU is lost")
        return {"gfx_activity": 70 + handle, "umc_activity": 10, "mm_activity": 0}

    def amdsmi_get_gpu_vram_usage(self, handle):
        # AMD SMI 的 vram_used 直接是 MiB，而不是字节
        return {"vram_total": 192 * 1024, "vram_used": (handle + 1) * 2048}

    def amdsmi_get_power_info(self, handle):
        # 让三个功耗字段取值互不相同，以便捕获选错字段的回归
        return {
            "socket_power": 500 + handle,
            "current_socket_power": 475 + handle,
            "average_socket_power": 250 + handle,
        }

    def amdsmi_get_gpu_process_list(self, handle):
        if handle in self.failing_devices:
            raise RuntimeError("GPU is lost")
        return [
            {"pid": 2000 + handle, "name": f"amd-proc-{handle}" if handle == 0 else "N/A",
             "memory_usage": {"vram_mem": (handle + 1) * 768 * 1024 * 1024}},
            # 第二个进程模拟 KFD 旁观者：不持有 VRAM，应被 read_processes 过滤
            {"pid": 9000 + handle, "name": "kfd-bystander", "memory_usage": {"vram_mem": 0}},
        ]


def test_production_auto_detection_falls_back_from_nvml_to_amdsmi(monkeypatch):
    # 生产环境不注入任何 backend：pynvml 初始化失败时应自动切到 amdsmi
    monkeypatch.setitem(sys.modules, "pynvml", FakeNvml(fail_init=True))
    monkeypatch.setitem(sys.modules, "amdsmi", FakeAmdSmi(count=1))

    sampler = GpuSampler(PushSpy(), node="n")

    assert sampler.available
    assert sampler.gpu_uuids() == ["GPU-amd-0"]
```

tests/fast-gpu/test_gpu_sampler_hw.py

新增的真实硬件 smoke test：CPU-only 的 fast 测试无法触达真实 vendor 库，此文件注册到 CUDA 与 ROCm 两套 GPU CI fleet，保证 auto-detection 在两个平台上都真实验证。

```
from tests.ci.ci_register import register_cuda_ci, register_rocm_ci
```

```
# 挂到两套 GPU fleet：H200 跑 CUDA 路径，MI350 跑 ROCm 路径
register_cuda_ci(est_time=60, suite="stage-b-2-gpu-h200", labels=["short"])
register_rocm_ci(est_time=60, suite="stage-c-4-gpu-mi350", labels=["amd"])

import torch

from miles.dashboard.gpu_sampler import GpuSampler

class PushSpy:
    def __init__(self):
        self.calls = []

    def __call__(self, node, batch):
        self.calls.append((node, batch))


def test_auto_detection_picks_the_backend_matching_the_hardware():
    # 不注入任何 backend，让 sampler 自己按硬件选择 vendor 路径
    sampler = GpuSampler(PushSpy(), node="ci")
    assert sampler.available, "no GPU telemetry backend initialized on a GPU runner"
    expected = "AMD SMI" if torch.version.hip else "NVML"
    assert sampler._provider.name == expected


def test_every_device_reports_telemetry_and_processes():
    push = PushSpy()
    push_processes = PushSpy()
    sampler = GpuSampler(push, node="ci", push_processes=push_processes)
    assert sampler.available

    uuids = sampler.gpu_uuids()
    count = len(uuids)
    assert count >= 2, f"expected a multi-GPU CI runner, saw {count} device(s)"
    assert all(uuids)

    assert sampler.sample_once(ts=1.0) == count
    assert sampler.sample_processes_once(ts=1.0) >= 0
    sampler.flush()
    [(_, batch)] = push.calls
    assert [sample.gpu for sample in batch] == list(range(count))
    # 真机断言只做合法性检查，单位换算的正确性由 fake 单测保障
    for sample in batch:
        assert 0 <= sample.util <= 100
        assert sample.mem_mb >= 0 and sample.power_w >= 0
    if push_processes.calls:
        for process_sample in push_processes.calls[0][1]:
            assert process_sample.pid > 0 and process_sample.mem_mb >= 0 and process_sample.
            name
```

评论区精华

这轮 review 的主体是作者 yushengsu-thu 的自审：他先是要求“clean these code”，只保留两个 Provider 类、压缩多行注释、删除冗余测试，随后在 24389786e 中移除了 _GpuProvider Protocol 与 _import_nvml / _import_amdsmi 包装，理由是“Protocol 只是类型注解、从未被实现”，测试改为直接 monkeypatch.setitem(sys.modules, ...) 打桩。74d800d 中有一个关键的上游对齐决策：drop provider shutdown 生命周期——因为生产环境用 ray.kill 销毁 sampler、从不调用 stop()，且 base 代码从未调用 nvmlShutdown。功耗降级语义也在此轮敲定：AMD 功耗传感器不可用（"N/A" 或 0xFFFF 哨兵）时只把 power_w 置 0，util/mem 继续上报。最终 yueming-yuan 给出 APPROVED。

- 精简为两个 Provider 类并移除 Protocol (design): commit 24389786e 完成精简，文件从 300 行降到 277 行；provider 接口改由注释文档化。
- 功耗降级语义 (design): power_w 降级为 0 W，util/mem 继续上报；_amd_socket_power 按 socket_power → current → average 顺序取字段并过滤哨兵值。
- KFD 旁观进程过滤 (correctness): _AmdSmiProvider.read_processes 过滤 vram_mem == 0 的行，并配套测试覆盖。
- 注释与测试冗余清理 (style): commit 883ffde66 完成全部清理，所有评论均答复 "Done in 883ffde66"。
- 真实硬件 CI 覆盖 (testing): 新增 tests/fast-gpu/test_gpu_sampler_hw.py，分别注册 stage-b-2-gpu-h200 与 stage-c-4-gpu-mi350，验证后端选择与全设备合法遥测。

风险与影响

- 风险：
  1. AMD SMI API 版本漂移：_AmdSmiProvider 依赖 ROCm >= 6.1 的 dict 形态返回（gfx_activity、vram_used、socket_power 等字段），不同 ROCm 版本的字段名与哨兵值已有差异，未来版本升级仍需跟进。
  2. 缺少 shutdown 生命周期：作者明确说明上游用 ray.kill 销毁、从不调用 stop()，与 base NVML 行为一致；但长生命周期 Ray actor 内若反复初始化 AMD SMI 可能累积资源，这一点未处理。
  3. partitioned 卡 UUID 重复：代码注释指出 partitioned MI300+ 卡可能重复物理 GPU UUID，若 Dashboard 时间线以 UUID 为 key 可能出现 lane 合并或覆盖。
  4. 真机测试断言偏宽：tests/fast-gpu/test_gpu_sampler_hw.py 只断言 util 在 0-100、mem/power 非负与 name 非空，无法捕获单位换算类回归；单位正确性主要靠 fake 单测保障。 - 影响：对 AMD/ROCm 集群用户，Dashboard 首次能看到 GPU util/mem/power 时间线，/api/timeline/gpu 不再返回空 lanes；对 NVIDIA 用户无感知，NVML 路径保持默认行为；对 Dashboard 团队，新增了一个可复用的硬件 vendor 适配模式，未来可扩展更多 SMI 后端。CI 侧新增 tests/fast-gpu 硬件测试，同时覆盖 CUDA 与 ROCm 两套 GPU fleet，tests/fast/dashboard 保持 CPU-only。PR body 报告在 4 x MI350X devbox 上的 1-GPU Qwen3-0.6B Megatron Bridge run 中，训练 GPU 达到 90% 利用率、187695 MiB VRAM、356 W，浏览器 UI 显示四条利用率 lane 且无警告。 - 风险标记：AMD SMI API 版本漂移，缺少 shutdown 生命周期，

partitioned 卡 UUID 重复，真机测试断言偏宽

关联脉络

- PR #2606 update doc & readme: 同为 Dashboard 模块的文档调整，此前刚把未稳定的 dashboard 长文档重定向到监控页；本 PR 是 Dashboard 功能的后续推进。
- PR #2674 [AMD] Let the ROCm suite honour the ci-sglang-pr and ci-megatron-pr directives: 同属 AMD/ROCm 平台支持主线，构建 ROCm CI 能力；本 PR 的 tests/fast-gpu 也利用了 ROCm fleet 注册机制。
- PR #2671 [AMD] Enable four CI tests on ROCm: AMD CI 测试启用系列的延续，与本 PR 在 ROCm 镜像与 CI 套件上有共同的依赖面。