```
import os

import sys

import time

from typing import Any, Callable, Dict, Optional, Union



import GPUtil

import psutil

from loguru import logger



# Configurable environment variables with validation

LOG_LEVEL = os.getenv("LOG_LEVEL", "INFO")

if LOG_LEVEL.upper() not in [

    "DEBUG",

    "INFO",

    "WARNING",

    "ERROR",

    "CRITICAL",

]:

    LOG_LEVEL = "INFO"



RETRY_COUNT = max(

    1, int(os.getenv("RETRY_COUNT", 3))

)  # Minimum 1 retry

RETRY_DELAY = max(

    0.1, float(os.getenv("RETRY_DELAY", 1.0))

)  # Minimum 0.1s delay
```

```
CPU_THRESHOLD = min(
    100, max(0, int(os.getenv("CPU_THRESHOLD", 90)))
)  # 0-100 range


# Configure Loguru logger for detailed logging
logger.remove()
logger.add(
    sys.stderr,
    level=LOG_LEVEL.upper(),
    format="{time} | {level} | {message}",
)



def monitor_resources(
    cpu_threshold: Optional[int] = None,
    gpu_threshold: Optional[int] = 90,
    interval: float = 1.0,
) -> Dict[str, Any]:
    """
    Continuously monitors CPU and GPU resources and logs alerts when thresholds are crossed.


    Args:
        cpu_threshold (Optional[int]): CPU usage percentage threshold for alerts (0-100).
            If None, uses CPU_THRESHOLD from env vars.
        gpu_threshold (Optional[int]): GPU memory usage percentage threshold for alerts (0-100).
            If None, monitoring of GPUs is disabled.
        interval (float): Time interval in seconds between measurements.


    Returns:

        Dict[str, Any]: Resource usage statistics including:

            - cpu_usage: Current CPU usage percentage

            - gpu_stats: List of dicts with GPU stats (id, memory_used, memory_total)

            - alerts: List of any threshold violations


    Raises:

        ValueError: If thresholds are not in valid range 0-100

        RuntimeError: If resource monitoring fails
    """
    if cpu_threshold is not None and not 0 <= cpu_threshold <= 100:

        raise ValueError("CPU threshold must be between 0 and 100")

    if gpu_threshold is not None and not 0 <= gpu_threshold <= 100:

        raise ValueError("GPU threshold must be between 0 and 100")


    stats = {"cpu_usage": 0.0, "gpu_stats": [], "alerts": []}


    try:

        # Monitor CPU

        stats["cpu_usage"] = psutil.cpu_percent(interval=interval)

        threshold = (

            cpu_threshold

            if cpu_threshold is not None

            else CPU_THRESHOLD
        )
        if stats["cpu_usage"] > threshold:
            alert = f"CPU usage exceeds {threshold}%: Current usage {stats['cpu_usage']}%"
            stats["alerts"].append(alert)
            logger.warning(alert)

        # Monitor GPUs if threshold provided
        if gpu_threshold is not None:
            gpus = GPUtil.getGPUs()
            for gpu in gpus:
                memory_usage = 100 * (
                    1 - gpu.memoryFree / gpu.memoryTotal
                )
                gpu_stat = {
                    "id": gpu.id,
                    "memory_used": gpu.memoryUsed,
                    "memory_total": gpu.memoryTotal,
                    "usage_percent": memory_usage,
                }
                stats["gpu_stats"].append(gpu_stat)

                if memory_usage > gpu_threshold:
                        alert = f"GPU {gpu.id} memory usage exceeds {gpu_threshold}%: Current usage
    {memory_usage:.1f}%"
                    stats["alerts"].append(alert)
                    logger.warning(alert)
        logger.info("Resource monitoring completed successfully")

        return stats


    except Exception as e:

        error_msg = f"Error monitoring resources: {str(e)}"

        logger.error(error_msg)

        raise RuntimeError(error_msg) from e




def profile_execution(

    func: Callable,

    *args: Any,

    collect_gpu_metrics: bool = True,

    **kwargs: Any,

) -> Dict[str, Union[Any, float, Dict]]:

    """

    Profiles the execution of a task, collecting metrics like execution time and resource usage.


    Args:

        func (Callable): The function to profile

        *args (Any): Arguments for the callable

        collect_gpu_metrics (bool): Whether to collect GPU metrics. Default True.

        **kwargs (Any): Keyword arguments for the callable


    Returns:
```

```
    Dict containing:
        - result: Return value from the function
        - metrics: Dict of execution metrics including:
            - execution_time: Time taken in seconds
            - cpu_usage: Dict of CPU usage before/after
            - gpu_usage: Dict of GPU memory usage before/after (if enabled)

Raises:
    RuntimeError: If profiling or function execution fails
"""
metrics = {
    "execution_time": 0.0,
    "cpu_usage": {},
    "gpu_usage": {},
}

try:
    start_time = time.time()

    # Get initial resource usage
    metrics["cpu_usage"]["initial"] = psutil.cpu_percent()

    if collect_gpu_metrics:
        gpus = GPUtil.getGPUs()
        metrics["gpu_usage"]["initial"] = {
            gpu.id: {
```

```
            "free": gpu.memoryFree,
            "used": gpu.memoryUsed,
            "total": gpu.memoryTotal,
        }
        for gpu in gpus
    }

# Execute function
result = func(*args, **kwargs)

# Collect final metrics
metrics["execution_time"] = time.time() - start_time
metrics["cpu_usage"]["final"] = psutil.cpu_percent()

if collect_gpu_metrics:
    gpus = GPUtil.getGPUs()
    metrics["gpu_usage"]["final"] = {
        gpu.id: {
            "free": gpu.memoryFree,
            "used": gpu.memoryUsed,
            "total": gpu.memoryTotal,
        }
        for gpu in gpus
    }

# Log metrics
```

```
        logger.info(
            f"Task execution time: {metrics['execution_time']:.2f}s"
        )
        logger.info(
            f"CPU usage: {metrics['cpu_usage']['initial']}% -> {metrics['cpu_usage']['final']}%"
        )

        if collect_gpu_metrics:
            for gpu_id, usage in metrics["gpu_usage"][
                "final"
            ].items():
                initial = metrics["gpu_usage"]["initial"][gpu_id]
                logger.info(
                    f"GPU {gpu_id} memory: {initial['free']}MB free -> {usage['free']}MB free"
                )

        return {"result": result, "metrics": metrics}

    except Exception as e:
        error_msg = f"Error during profiled execution: {str(e)}"
        logger.error(error_msg)
        raise RuntimeError(error_msg) from e


# # Example function to run
# def sample_task(n: int) -> int:
```

```
#     return n * n



# # Monitor resources during execution

# monitor_resources()


# # Profile task execution and collect metrics

# profile_execution(sample_task, 10)


# # Execute distributed across multiple GPUs

# distributed_execute_on_gpus([0, 1], sample_task, 10)
```