```
import os
import psutil
from typing import Callable, Any
import functools
from loguru import logger


def run_on_cpu(func: Callable) -> Callable:
    """
    Decorator that ensures the function runs on all available CPU cores,
    maximizing CPU and memory usage to execute the function as quickly as possible.

    This decorator sets the CPU affinity of the current process to all available CPU cores
    before executing the function. After the function completes, the original CPU affinity is restored.

    Args:
        func (Callable): The function to be executed.

    Returns:
        Callable: The wrapped function with CPU affinity settings applied.

    Raises:
        RuntimeError: If the CPU affinity cannot be set or restored.
    """

    @functools.wraps(func)
```

```
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        # Get the current process
        process = psutil.Process(os.getpid())

        # Check if the platform supports cpu_affinity
        if not hasattr(process, "cpu_affinity"):
            logger.warning(
                "CPU affinity is not supported on this platform. Executing function without setting CPU affinity."
            )
            return func(*args, **kwargs)

        # Save the original CPU affinity
        original_affinity = process.cpu_affinity()
        logger.info(f"Original CPU affinity: {original_affinity}")

        try:
            # Set the CPU affinity to all available CPU cores
            all_cpus = list(range(os.cpu_count()))
            process.cpu_affinity(all_cpus)
            logger.info(f"Set CPU affinity to: {all_cpus}")

            # Set process priority to high
            try:
                process.nice(psutil.HIGH_PRIORITY_CLASS)
                logger.info("Set process priority to high.")
```

```
    except AttributeError:
        logger.warning(
            "Setting process priority is not supported on this platform."
        )

    # Pre-allocate memory by creating a large array (optional step)
    memory_size = int(
        psutil.virtual_memory().available * 0.9
    )  # 90% of available memory
    try:
        logger.info(
            f"Pre-allocating memory: {memory_size} bytes"
        )
        _ = bytearray(memory_size)
    except MemoryError:
        logger.error(
            "Failed to pre-allocate memory, continuing without pre-allocation."
        )

    # Run the function
    result = func(*args, **kwargs)

except psutil.AccessDenied as e:
    logger.error(
        "Access denied while setting CPU affinity",
        exc_info=True,
```

```
    )
    raise RuntimeError(
        "Access denied while setting CPU affinity"
    ) from e

except psutil.NoSuchProcess as e:
    logger.error("Process does not exist", exc_info=True)
    raise RuntimeError("Process does not exist") from e

except Exception as e:
    logger.error(
        "An error occurred during function execution",
        exc_info=True,
    )
    raise RuntimeError(
        "An error occurred during function execution"
    ) from e

finally:
    # Restore the original CPU affinity
    try:
        process.cpu_affinity(original_affinity)
        logger.info(
            f"Restored original CPU affinity: {original_affinity}"
        )
    except Exception as e:
```

```
            logger.error(
                "Failed to restore CPU affinity", exc_info=True
            )
            raise RuntimeError(
                "Failed to restore CPU affinity"
            ) from e

        return result

    return wrapper


# # Example usage of the decorator
# @run_on_cpu
# def compute_heavy_task() -> None:
#     # An example task that is CPU and memory intensive
#     data = [i**2 for i in range(100000000)]
#     sum(data)
#     print("Task completed.")


# compute_heavy_task()
```