```python
import os
import sys
from dataclasses import dataclass
from typing import Any, Callable, Dict, List, Optional, Tuple
from functools import wraps
import inspect
import numpy as np
import psutil
import torch
from loguru import logger
from tenacity import (
    retry,
    retry_if_exception_type,
    stop_after_attempt,
    wait_fixed,
)
import time


# Configure Loguru logger
LOG_LEVEL = os.getenv("LOG_LEVEL", "INFO")
LOG_FORMAT = (
    "<green>{time:YYYY-MM-DD HH:mm:ss.SSS}</green> | "
    "<level>{level: <8}</level> | "
    "<cyan>{name}</cyan>:<cyan>{function}</cyan>:<cyan>{line}</cyan> | "
    "<level>{message}</level>"
)
```

```
# Remove default logger and add custom configuration

logger.remove()

logger.add(

    sys.stderr,

    format=LOG_FORMAT,

    level=LOG_LEVEL,

    backtrace=True,

    diagnose=True,

    enqueue=True,

)

logger.add(

    "resource_manager_{time}.log",

    format=LOG_FORMAT,

    level=LOG_LEVEL,

    rotation="100 MB",

    retention="30 days",

    compression="zip",

)




@dataclass

class ResourceRequirements:

    """Data class to store predicted resource requirements."""



    cpu_memory_bytes: int
    gpu_memory_bytes: int

    requires_gpu: bool

    estimated_runtime_seconds: float



@dataclass

class ResourceAllocation:

    """Data class to store actual resource allocation."""


    cpu_memory_reserved: int

    gpu_memory_reserved: int

    gpu_id: Optional[int]

    cpu_cores: List[int]

    allocation_time: float = time.time()



class InsufficientResourcesError(Exception):

    """Raised when there are not enough resources to execute the function."""


    pass



class MemoryPredictor:

    """Predicts memory requirements for function execution."""


    def __init__(self):
        self.history: Dict[str, List[ResourceRequirements]] = {}
        self.sample_size = 3
        logger.info(
            "Initialized MemoryPredictor with sample size {}",
            self.sample_size,
        )

    def update_history(
        self, func_name: str, requirements: ResourceRequirements
    ) -> None:
        """
        Updates the execution history for a function with new resource requirements.

        Args:
            func_name: Name of the function
            requirements: Resource requirements from the latest execution
        """
        try:
            with logger.contextualize(function=func_name):
                if func_name not in self.history:
                    self.history[func_name] = []

                self.history[func_name].append(requirements)

                # Keep only recent history
                if len(self.history[func_name]) > self.sample_size:
```

```
    self.history[func_name] = self.history[func_name][
        -self.sample_size :
    ]

logger.debug(
    "Updated history for function {}. Total records: {}",
    func_name,
    len(self.history[func_name]),
)

# Calculate and log average requirements
avg_cpu = sum(
    r.cpu_memory_bytes
    for r in self.history[func_name]
) / len(self.history[func_name])
avg_gpu = sum(
    r.gpu_memory_bytes
    for r in self.history[func_name]
) / len(self.history[func_name])

logger.debug(
    "Average requirements - CPU: {} bytes, GPU: {} bytes",
    int(avg_cpu),
    int(avg_gpu),
)
```

```
    except Exception as e:
        logger.exception(
            "Error updating history for function {}: {}",
            func_name,
            str(e),
        )

def get_historical_requirements(
    self, func_name: str
) -> Optional[ResourceRequirements]:
    """
    Gets average historical resource requirements for a function.

    Args:
        func_name: Name of the function

    Returns:
        Optional[ResourceRequirements]: Average requirements or None if no history
    """
    try:
        if (
            func_name not in self.history
            or not self.history[func_name]
        ):
            logger.debug(
                "No history found for function {}", func_name
    )
    return None

history = self.history[func_name]
avg_requirements = ResourceRequirements(
    cpu_memory_bytes=int(
        sum(r.cpu_memory_bytes for r in history)
        / len(history)
    ),
    gpu_memory_bytes=int(
        sum(r.gpu_memory_bytes for r in history)
        / len(history)
    ),
    requires_gpu=any(r.requires_gpu for r in history),
    estimated_runtime_seconds=sum(
        r.estimated_runtime_seconds for r in history
    )
    / len(history),
)

logger.debug(
    "Retrieved historical requirements for {}: {}",
    func_name,
    avg_requirements,
)
return avg_requirements
```

```
    except Exception as e:
        logger.exception(
            "Error retrieving historical requirements for {}: {}",
            func_name,
            str(e),
        )
        return None

def _estimate_object_size(self, obj: Any) -> int:
    """
    Estimates the memory size of a Python object.

    Args:
        obj: Any Python object

    Returns:
        int: Estimated size in bytes
    """
    try:
        with logger.contextualize(object_type=type(obj).__name__):
            if isinstance(obj, (np.ndarray, torch.Tensor)):
                size = obj.nbytes
                logger.debug("Array/Tensor size: {} bytes", size)
                return size
            elif isinstance(obj, (list, tuple, set, dict)):
                size = sys.getsizeof(obj) + sum(
                    self._estimate_object_size(item)
                    for item in obj
                )
                logger.debug("Container size: {} bytes", size)
                return size

            size = sys.getsizeof(obj)
            logger.debug("Object size: {} bytes", size)
            return size
    except Exception as e:
        logger.exception("Error estimating object size: {}", e)
        return 0

def _analyze_function_memory(
    self, func: Callable, *args, **kwargs
) -> ResourceRequirements:
    """
    Analyzes function's memory requirements based on its arguments and source code.

    Args:
        func: Function to analyze
        *args: Function arguments
        **kwargs: Function keyword arguments

    Returns:
```

```
    ResourceRequirements: Predicted resource requirements
"""
logger.info(
    "Analyzing memory requirements for function: {}",
    func.__name__,
)

with logger.contextualize(function=func.__name__):
    try:
        # Estimate memory for input arguments
        args_size = sum(
            self._estimate_object_size(arg) for arg in args
        )
        kwargs_size = sum(
            self._estimate_object_size(value)
            for value in kwargs.values()
        )

        logger.debug("Arguments size: {} bytes", args_size)
        logger.debug(
            "Keyword arguments size: {} bytes", kwargs_size
        )

        # Analyze function source code for GPU operations
        source = inspect.getsource(func)
        requires_gpu = any(
    keyword in source
    for keyword in ["torch.cuda", "gpu", "GPU"]
)
logger.debug(
    "GPU requirement detected: {}", requires_gpu
)

# Estimate GPU memory if needed (using a heuristic based on input size)
gpu_memory = args_size * 2 if requires_gpu else 0

# Add buffer for intermediate computations (50% of input size)
cpu_memory = (args_size + kwargs_size) * 1.5

requirements = ResourceRequirements(
    cpu_memory_bytes=int(cpu_memory),
    gpu_memory_bytes=int(gpu_memory),
    requires_gpu=requires_gpu,
    estimated_runtime_seconds=1.0,
)

logger.info(
    "Memory requirements analysis complete: {}",
    requirements,
)
return requirements
```

```
        except Exception as e:
            logger.exception(
                "Error analyzing function memory requirements: {}",
                e,
            )
            raise


class ResourceManager:
    """Manages resource allocation for function execution."""

    def __init__(self):
        self.predictor = MemoryPredictor()
        self.reserved_cpu_memory = 0
        self.reserved_gpu_memory = {}
        logger.info("Initialized ResourceManager")

        # Log initial system resources
        self._log_system_resources()

    def _log_system_resources(self):
        """Logs current system resource status."""
        cpu_info = {
            "total_cpu_memory": psutil.virtual_memory().total,
            "available_cpu_memory": psutil.virtual_memory().available,
            "cpu_percent": psutil.cpu_percent(interval=1),
        "cpu_count": psutil.cpu_count(),
    }

    gpu_info = {}
    if torch.cuda.is_available():
        for i in range(torch.cuda.device_count()):
            gpu_info[f"gpu_{i}"] = {
                "name": torch.cuda.get_device_name(i),
                "total_memory": torch.cuda.get_device_properties(
                    i
                ).total_memory,
                "allocated_memory": torch.cuda.memory_allocated(
                    i
                ),
            }

    logger.info("System resources - CPU: {}", cpu_info)
    if gpu_info:
        logger.info("System resources - GPU: {}", gpu_info)

def _get_available_resources(self) -> Tuple[int, Dict[int, int]]:
    """
    Gets available CPU and GPU memory.

    Returns:
        Tuple[int, Dict[int, int]]: Available CPU memory and GPU memory per device
    """
    with logger.contextualize(operation="resource_check"):
        cpu_available = psutil.virtual_memory().available
        gpu_available = {}

        if torch.cuda.is_available():
            for i in range(torch.cuda.device_count()):
                gpu_available[i] = (
                    torch.cuda.get_device_properties(
                        i
                    ).total_memory
                    - torch.cuda.memory_allocated(i)
                )

        logger.debug(
            "Available CPU memory: {} bytes", cpu_available
        )
        logger.debug("Available GPU memory: {}", gpu_available)

        return cpu_available, gpu_available

@retry(
    stop=stop_after_attempt(3),
    wait=wait_fixed(1),
    retry=retry_if_exception_type(Exception),
)
```

```
def allocate_resources(
    self, requirements: ResourceRequirements
) -> ResourceAllocation:
    """
    Allocates required resources for function execution.

    Args:
        requirements: ResourceRequirements object with predicted requirements

    Returns:
        ResourceAllocation: Allocated resources

    Raises:
        InsufficientResourcesError: If required resources are not available
    """
    with logger.contextualize(operation="resource_allocation"):
        logger.info(
            "Attempting to allocate resources: {}", requirements
        )

        cpu_available, gpu_available = (
            self._get_available_resources()
        )

        # Check CPU memory availability
        if requirements.cpu_memory_bytes > cpu_available:
```

```
    logger.error(
        "Insufficient CPU memory. Required: {}, Available: {}",
        requirements.cpu_memory_bytes,
        cpu_available,
    )
    raise InsufficientResourcesError(
        "Insufficient CPU memory"
    )

# Select GPU if required
gpu_id = None
if requirements.requires_gpu:
    for gid, available in gpu_available.items():
        if available >= requirements.gpu_memory_bytes:
            gpu_id = gid
            logger.info("Selected GPU {}", gpu_id)
            break
    else:
        logger.error(
            "No GPU with sufficient memory. Required: {}",
            requirements.gpu_memory_bytes,
        )
        raise InsufficientResourcesError(
            "Insufficient GPU memory"
        )
```

```
# Allocate CPU cores based on memory requirements
total_cores = psutil.cpu_count()
cores_needed = max(
    1,
    min(
        total_cores,
        requirements.cpu_memory_bytes
        // (cpu_available // total_cores),
    ),
)
allocated_cores = list(range(cores_needed))

allocation = ResourceAllocation(
    cpu_memory_reserved=requirements.cpu_memory_bytes,
    gpu_memory_reserved=(
        requirements.gpu_memory_bytes
        if gpu_id is not None
        else 0
    ),
    gpu_id=gpu_id,
    cpu_cores=allocated_cores,
    allocation_time=time.time(),
)

logger.success(
    "Resource allocation successful: {}", allocation
```

```
        )
        return allocation


def release_resources(self, allocation: ResourceAllocation):
    """Releases allocated resources."""
    with logger.contextualize(operation="resource_release"):
        try:
            self.reserved_cpu_memory -= (
                allocation.cpu_memory_reserved
            )
            if allocation.gpu_id is not None:
                self.reserved_gpu_memory[
                    allocation.gpu_id
                ] -= allocation.gpu_memory_reserved

            allocation_duration = (
                time.time() - allocation.allocation_time
            )
            logger.info(
                "Resources released. Duration: {:.2f}s, CPU: {} bytes, GPU: {} bytes",
                allocation_duration,
                allocation.cpu_memory_reserved,
                allocation.gpu_memory_reserved,
            )
        except Exception as e:
            logger.exception("Error releasing resources: {}", e)
```

```
                raise


def with_resource_management(func: Callable) -> Callable:
    """
    Decorator that handles resource prediction and allocation for a function.

    Args:
        func: Function to be wrapped

    Returns:
        Callable: Wrapped function with resource management
    """
    resource_manager = ResourceManager()

    @wraps(func)
    def wrapper(*args, **kwargs):
        with logger.contextualize(
            function=func.__name__, execution_id=time.time_ns()
        ):
            logger.info("Starting execution with resource management")
            start_time = time.time()

            try:
                # Check historical data first
                historical_requirements = resource_manager.predictor.get_historical_requirements(
    func.__name__
)

if historical_requirements is not None:
    logger.info(
        "Using historical requirements: {}",
        historical_requirements,
    )
    requirements = historical_requirements
else:
    # Predict new requirements
    requirements = resource_manager.predictor._analyze_function_memory(
        func, *args, **kwargs
    )

# Allocate resources
allocation = resource_manager.allocate_resources(
    requirements
)

# Execute function with allocated resources
if allocation.gpu_id is not None:
    with torch.cuda.device(allocation.gpu_id):
        result = func(*args, **kwargs)
else:
    result = func(*args, **kwargs)
    # Update execution history
    execution_time = time.time() - start_time
    requirements.estimated_runtime_seconds = (
        execution_time
    )
    resource_manager.predictor.update_history(
        func.__name__, requirements
    )

    logger.success(
        "Function execution completed successfully in {:.2f}s",
        execution_time,
    )

    return result

except Exception as e:
    logger.exception(
        "Error during function execution: {}", e
    )
    raise

finally:
    if "allocation" in locals():
        resource_manager.release_resources(allocation)
    return wrapper



if __name__ == "__main__":

    @with_resource_management
    def process_large_matrix(size: int = 10000) -> np.ndarray:
        # Create a large matrix and perform memory-intensive operations
        data = np.random.random((size, size))
        result = np.matmul(data, data.T)  # Matrix multiplication

        # Additional operations to increase memory usage
        for _ in range(3):
            result = np.exp(result)  # Element-wise exponential
            result = np.sin(result)  # Element-wise sine
            result = result @ result  # Another matrix multiplication

        return result

    # Process a 10k x 10k matrix with intensive operations
    result = process_large_matrix()
    print(f"Result shape: {result.shape}")
    print(f"Peak memory usage: {result.nbytes / 1e9:.2f} GB")
```