```
import operator
from functools import reduce
from typing import List, Dict

import helper
from kernel import Kernel
from log_level import LogLevel


class Optimizer:
    """
        This is the Optimizer class to optimize the usage of fast on-chip memory vs slow ddr memory vs bandwidth between
        them.

        Optimization strategy:
            - initial state: all buffers are in fast memory, there is no communication volume used for transferring data
              between slow and fast memory
            - optimize for:
                - minimize_comm_vol
                - minimize_fast_mem
                -
    """

    def __init__(self,
                 kernels: Dict[str, Kernel],
                 dimensions: List[int],
                 log_level: int = 0):
        """
        Create new BoundedQueue with given initialization parameters.
        :param kernels: all kernels
        :param dimensions: global dimensions / problem size (i.e. size of the input array
        :param verbose: flag for console output logging
        """
        if log_level >= LogLevel.BASIC.value:
            print("Initialize Optimizer.")
        # save params
        self.kernels = kernels
        self.dimensions: List[int] = dimensions
        self.log_level = log_level
        # init local fields
        self.fast_memory_use: int = 0
        self.slow_memory_use: int = 0
        self.metric_data: List[Dict] = list()
        self.config = helper.parse_json("stencil_chain.config")
        self.eps = self.config["eps"]  # machine precision (i.e. used for division by (almost) zero)
        # run init methods
        if self.log_level >= LogLevel.MODERATE.value:
            print("Add all buffers to the metric.")
        self.add_buffers_to_metric()
```

```
    if self.log_level >= LogLevel.MODERATE.value:
        print("Reset old state.")
    self.reset()

def reinit(self):
    """
    Clean old state to run a new optimization round.
    """
    # reinit local fields
    self.fast_memory_use: int = 0
    self.slow_memory_use: int = 0
    self.metric_data: List[Dict] = list()
    # run init methods
    self.add_buffers_to_metric()
    self.reset()

def minimize_comm_vol(self,
                      fast_memory_bound: int,
                      slow_memory_bound: int) -> None:
    """
    This optimization strategy optimizes the problem to minimize the communication volume between fast and slow
    memory. In other words, it uses as few bandwidth as possible by a given amount of fast and slow memory.
    :param fast_memory_bound: maximum available fast on-chip memory
    :param slow_memory_bound: maximum available slow dram memory
    """
    if self.log_level >= LogLevel.BASIC.value:
        print(
            "Run optimizer in mode: Minimize Communication Volume with fast memory bound: {} and slow memory "
            "bound: {}.".format(fast_memory_bound, slow_memory_bound))
    self.reinit()
    # optimize for minimal communication volume use / maximal fast memory use
    opt = self.max_metric()
    while not self.empty_list(self.metric_data) and self.fast_memory_use > fast_memory_bound:
        self.fast_memory_use -= opt["queue"].maxsize * opt["datatype_size"]
        self.slow_memory_use += opt["queue"].maxsize * opt["datatype_size"]
        opt["queue"].swap_out = True
        self.update_neighbours(opt)
        self.metric_data.remove(opt)
        opt = self.max_metric()
    if self.slow_memory_use > slow_memory_bound:
        raise Exception("Optimization failed, slow memory bound: {}, slow memory necessary to hold fast memory "
                        "constraint: {}".format(slow_memory_bound, self.slow_memory_use))
    if self.log_level >= LogLevel.MODERATE.value:
        self.report()

def minimize_fast_mem(self,
                      communication_volume_bound: int) -> None:
    """
    This optimization strategy minimizes the usage of fast memory by a given amount of communication volume from/to
```

```
    slow memory.
    :param communication_volume_bound: maximum available data volume between fast and slow memory
    """
    if self.log_level >= LogLevel.BASIC.value:
        print("Run optimizer in mode: Minimize Fast Memory Usage with comm volume bound: {}".format(
            communication_volume_bound))
    self.reinit()
    # optimize for minimal fast memory use / maximum communication volume use
    opt = self.max_metric()
    while not self.empty_list(self.metric_data) and opt["comm_vol"] < communication_volume_bound:
        communication_volume_bound -= opt["comm_vol"]
        self.fast_memory_use -= opt["queue"].maxsize * opt["datatype_size"]
        self.slow_memory_use += opt["queue"].maxsize * opt["datatype_size"]
        opt["queue"].swap_out = True
        self.update_neighbours(opt)
        self.metric_data.remove(opt)
        opt = self.max_metric()
    if self.log_level >= LogLevel.MODERATE.value:
        self.report()

def optimize_to_ratio(self,
                      ratio: float) -> None:
    """
    This optimization strategy optimizes for the given ratio value between the fast memory usage and the amount of
    available communication volume.
    :param ratio: ratio = #fast_mem / #comm_vol
    """
    if self.log_level >= LogLevel.BASIC.value:
        print("Run optimizer in mode: Optimize to Ratio with ratio: {}".format(ratio))
    self.reinit()
    # optimize for the ratio of #fast_memory/communication_volume
    opt = self.max_metric()
    while not self.empty_list(self.metric_data) and self.ratio() > ratio:
        self.fast_memory_use -= opt["queue"].maxsize * opt["datatype_size"]
        self.slow_memory_use += opt["queue"].maxsize * opt["datatype_size"]
        opt["queue"].swap_out = True
        self.update_neighbours(opt)
        self.metric_data.remove(opt)
        opt = self.max_metric()
    if self.log_level >= LogLevel.MODERATE.value:
        self.report()

@staticmethod
def empty_list(lst: List) -> bool:
    """
    Check if collection is empty.
    :param lst: collection
    """
    return len(lst) == 0
```

```
def ratio(self):
    """
    Get the ratio between fast and the communication volume of the current optimization state.
    :return: ratio: fast_mem / comm_vol
    """
    total_com = 0
    for item in self.metric_data:
        total_com += item["comm_vol"]
    return self.fast_memory_use / (total_com + self.eps)

def reset(self) -> None:
    """
    Reset the internal optimizer state to start a new round of optimization.
    """
    # initially put all buffers into fast memory
    for item in self.metric_data:
        item["queue"].swap_out = False
    # count fast memory usage
    self.fast_memory_use = 0
    for item in self.metric_data:
        if not item["queue"].swap_out:
            self.fast_memory_use += item["queue"].maxsize
    # reset slow memory usage
    self.slow_memory_use = 0

def max_metric(self) -> Dict:
    """
    Return the entry with the highest ratio between its size and the necessary communication volume (i.e. the "worst"
    node).
    :return: worst node (best candidate to swap out to slow memory)
    """
    if self.empty_list(self.metric_data):
        return dict()  # empty dict
    else:
        return max(self.metric_data, key=lambda x: x["queue"].maxsize / x["comm_vol"])

def update_neighbours(self,
                      buffer: Dict) -> None:
    """
    After moving the buffer from/to slow/fast memory, the communication volume to the direct neighbours must get
    updated.
    :param buffer: node that is being moved
    """
    if buffer["prev"] is not None:
        self.update_comm_vol(buffer["prev"])
    if buffer["next"] is not None:
        self.update_comm_vol(buffer["next"])
```

```
def update_comm_vol(self,
                    buffer: Dict) -> None:
    """
        Recompute the communication volume.

        How to determine the necessary communication volume?
        (predecessor, successor):
        case (fast, fast): 2C
        case (fast, slow): C
        case (slow, fast): C
        case (slow, slow): 0
            where C:= communication volume to stream single data array in or out of fast memory (=SIZE(data array))
        Note:
        pred of delay buffer is always fast memory
    :param buffer: node that has to be recomputed
    """
    # determine predecessor fast/slow
    if buffer["type"] == "delay":
        pre_fast = True
    elif buffer["prev"]["queue"].swap_out:
        pre_fast = False
    else:
        pre_fast = True
    # determine successor fast/slow
    if buffer["next"] is None:
        succ_fast = True
    elif buffer["next"]["queue"].swap_out:
        succ_fast = False
    else:
        succ_fast = True
    # set comm vol accordingly
    if pre_fast and succ_fast:  # case (fast, fast)
        buffer["comm_vol"] = 2 * self.single_comm_volume(buffer["datatype_size"])
    elif (pre_fast and not succ_fast) or (not pre_fast and succ_fast):  # case (fast, slow) or (slow, fast)
        buffer["comm_vol"] = 1 * self.single_comm_volume(buffer["datatype_size"])
    else:  # case (slow, slow)
        buffer["comm_vol"] = self.eps

def add_buffers_to_metric(self):
    """
    Create buffer data structure for optimization (contain buffers, type of buffer as well as predecessor/successor.
    """
    # loop over all kernels
    for kernel in self.kernels:
        # loop over all delay buffers
        for buf in self.kernels[kernel].delay_buffer:
            # get delay buffer first
            del_buf = {
                "queue": self.kernels[kernel].delay_buffer[buf],
```

```
                "comm_vol": 2 * self.single_comm_volume(self.kernels[kernel].data_type.bytes),
                "type": "delay",
                "datatype_size": self.kernels[kernel].data_type.bytes,
                "prev": None,
                "next": None}
            self.fast_memory_use += del_buf["queue"].maxsize * del_buf["datatype_size"]
            self.metric_data.append(del_buf)
            # get internal buffers next
            prev = del_buf
            for entry in self.kernels[kernel].internal_buffer[buf]:
                curr = {
                    "queue": entry,
                    "comm_vol": 2 * self.single_comm_volume(self.kernels[kernel].data_type.bytes),
                    "type": "internal",
                    "datatype_size": self.kernels[kernel].data_type.bytes,
                    "prev": prev,
                    "next": None}
                prev["next"] = curr
                self.fast_memory_use += curr["queue"].maxsize * curr["datatype_size"]
                self.metric_data.append(curr)
                prev = curr

def single_comm_volume(self,
                       datatype_size: int):
    """
    # Returns the number of bytes necessary to copy a whole array from or to the fpga.
    :param datatype_size: size in bytes of the kernel data type e.g. 4 for float32
    :return:
    """
    return reduce(operator.mul, self.dimensions) * datatype_size

def report(self):
    print("Optimization report:")
    # sum up data values
    total_fast, total_slow, total_comm = 0, 0, 0
    for kernel in self.kernels:
        # loop over all delay buffers
        for buf in self.kernels[kernel].delay_buffer:
            # get delay buffer
            if self.kernels[kernel].delay_buffer[buf].swap_out:
                print("Delay buffer: {} {}: swapped out to slow memory".format(kernel, buf))
                total_slow += self.kernels[kernel].delay_buffer[buf].maxsize * self.kernels[kernel].data_type.bytes
            else:
                print("Delay buffer: {} {}: kept in fast memory".format(kernel, buf))
                total_fast += self.kernels[kernel].delay_buffer[buf].maxsize * self.kernels[kernel].data_type.bytes
            # get internal buffers
            for entry in self.kernels[kernel].internal_buffer[buf]:
                if entry.swap_out:
                    print("Internal buffer: {} {} index {}: swapped out to slow memory"
```

```
                                .format(kernel, buf, self.kernels[kernel].internal_buffer[buf].index(entry)))
                        total_slow += entry.maxsize * self.kernels[kernel].data_type.bytes
                    else:
                        print("Internal buffer: {} {} index {}: kept in fast memory".format(kernel, buf, self.kernels[
                            kernel].internal_buffer[buf].index(entry)))
                        total_fast += entry.maxsize * self.kernels[kernel].data_type.bytes
        for item in self.metric_data:
            total_comm += self.metric_data["comm_vol"]
        print("Total fast memory usage: {} bytes".format(total_fast))
        print("Total slow memory usage: {} bytes".format(total_slow))
        print("Total communication volume usage: {} bytes".format(total_comm))


if __name__ == "__main__":
    opt = Optimizer()
    print()
```