kernel_chain_graph.py

```python
import argparse
import functools
import operator
import os
import re
from typing import List, Dict

import networkx as nx

import helper
from bounded_queue import BoundedQueue
from input import Input
from kernel import Kernel
from log_level import LogLevel
from output import Output


class KernelChainGraph:
    """
    The KernelChainGraph class represents the whole pipelined data flow graph consisting of input nodes (real data input
    arrays, kernel nodes and output nodes (storing the result of the computation).
    """

    def __init__(self,
                 path: str,
                 plot_graph: bool = False,
                 log_level: int = 0) -> None:
        """
        Create new KernelChainGraph with given initialization parameters.
        :param path: path to the input file
        :param plot_graph: flag indication whether or not to produce the graphical graph representation
        :param log_level: flag for console output logging
        """
        if log_level >= LogLevel.BASIC.value:
            print("Initialize KernelChainGraph.")
        # set parameters
        # absolute path
        self.path: str = os.path.join(os.path.dirname(os.path.dirname(os.path.realpath(__file__))), path)  # get valid
        self.log_level: int = log_level
        # init internal fields
        self.inputs: Dict[str, Dict[str, str]] = dict()  # input data
        self.outputs: List[str] = list()  # name of the output fields
        self.dimensions: List[int] = list()  # global problem size
        self.program: Dict[str, Dict[str, Dict[str, Dict[
            str, str]]]] = dict()  # mathematical stencil expressionos:program[stencil_name] = stencil expression
        self.kernel_latency = None  # critical path latency
        self.channels: Dict[str, BoundedQueue] = dict()  # each channel is an edge between two nodes
        self.graph: nx.DiGraph = nx.DiGraph()  # data flow graph
        self.input_nodes: Dict[str, Kernel] = dict()  # Input nodes of the graph
```

```
        self.output_nodes: Dict[str, Kernel] = dict()  # Output nodes of the graph
        self.kernel_nodes: Dict[str, Kernel] = dict()  # Kernel nodes of the graph
        self.config = helper.parse_json("stencil_chain.config")
        self.name = os.path.splitext(os.path.basename(self.path))[0]  # name
        self.kernel_dimensions = -1  # 2: 2D, 3: 3D
        # trigger all internal calculations
        if self.log_level >= LogLevel.BASIC.value:
            print("Read input config files.")
        self.import_input()  # read input config file
        if self.log_level >= LogLevel.BASIC.value:
            print("Create all kernels.")
        self.create_kernels()  # create all kernels
        if self.log_level >= LogLevel.BASIC.value:
            print("Compute kernel latencies.")
        self.compute_kernel_latency()  # compute their latencies
        if self.log_level >= LogLevel.BASIC.value:
            print("Connect kernels.")
        self.connect_kernels()  # connect them in the graph
        if self.log_level >= LogLevel.BASIC.value:
            print("Compute delay buffer sizes.")
        self.compute_delay_buffer()  # compute the delay buffer sizes
        if self.log_level >= LogLevel.BASIC.value:
            print("Add channels to the graph edges.")
        self.add_channels()  # add all channels (internal buffer and delay buffer) to the edges of the graph
        # plot kernel graphs if flag set to true
        if plot_graph:
            if self.log_level >= LogLevel.BASIC.value:
                print("Plot kernel chain graph.")
            # plot kernel chain graph
            self.plot_graph()
            # plot all compute graphs
            if self.log_level >= LogLevel.BASIC.value:
                print("Plot computation graph of each kernel.")
            for compute_kernel in self.kernel_nodes:
                self.kernel_nodes[compute_kernel].graph.plot_graph()
        # print sin/cos/tan latency warning
        for kernel in self.program:
            if "sin" in self.program[kernel]['computation_string'] or "cos" in self.program[kernel][
                'computation_string'] or "tan" in self.program[kernel]['computation_string']:
                print("Warning: Computation contains sinusoidal functions with experimental latency values.")
        # print report for moderate and high verbosity levels
        if self.log_level >= LogLevel.MODERATE.value:
            self.report(self.name)

    def plot_graph(self,
                   save_path: str = None) -> None:
        """
        Draw the networkx (library) KernelChainGraph graphically.
        :param save_path: path to save the image
```

```
"""
# create drawing area
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
fig.set_size_inches(25, 25)
ax.set_axis_off()
# generate positions of the node (for pretty visualization)
positions = nx.nx_pydot.graphviz_layout(self.graph, prog='dot')
# divide nodes into different lists for colouring purpose
nums = list()
names = list()
ops = list()
outs = list()
# add nodes to the corresponding list
for node in self.graph.nodes:
    if isinstance(node, Kernel):
        ops.append(node)
    elif isinstance(node, Input):
        names.append(node)
    elif isinstance(node, Output):
        outs.append(node)
# create dictionary of labels
labels = dict()
for node in self.graph.nodes:
    labels[node] = node.generate_label()
# add nodes and edges with distinct colours and shapes
nx.draw_networkx_nodes(
    self.graph,
    positions,
    nodelist=names,
    node_color='orange',
    node_size=3000,
    node_shape='s',
    edge_color='black')
nx.draw_networkx_nodes(
    self.graph,
    positions,
    nodelist=outs,
    node_color='green',
    node_size=3000,
    node_shape='s')
nx.draw_networkx_nodes(
    self.graph,
    positions,
    nodelist=nums,
    node_color='#007acc',
    node_size=3000,
    node_shape='s')
nx.draw_networkx(
```

```
            self.graph,
            positions,
            nodelist=ops,
            node_color='red',
            node_size=3000,
            node_shape='o',
            font_weight='bold',
            font_size=16,
            edge_color='black',
            arrows=True,
            arrowsize=36,
            arrowstyle='-|>',
            width=6,
            linwidths=1,
            with_labels=False)
        nx.draw_networkx_labels(
            self.graph,
            positions,
            labels=labels,
            font_weight='bold',
            font_size=16)
        # save plot to file if save_path has been specified
        if save_path is not None:
            fig.savefig(save_path)
        else:
            # plot it
            fig.show()

    def connect_kernels(self) -> None:
        """
        Connect the nodes to a directed acyclic graph by matching the input name with kernel names.
        """
        # loop over all node tuples
        for src in self.graph.nodes:
            for dest in self.graph.nodes:
                if src is not dest:  # skip src == dest case
                    if isinstance(src, Kernel) and isinstance(dest, Kernel):  # case: KERNEL -> KERNEL
                        for inp in dest.graph.inputs:
                            if src.name == inp.name:
                                # add edge
                                self.graph.add_edge(src, dest, channel=None)
                                break
                    elif isinstance(src, Input) and isinstance(dest, Kernel):  # case: INPUT -> KERNEL
                        for inp in dest.graph.inputs:
                            if src.name == inp.name:
                                # add edge
                                self.graph.add_edge(src, dest, channel=None)
                                break
                    elif isinstance(dest, Output):  # case: INPUT/KERNEL -> OUTPUT
```

```
                    if src.name == dest.name:
                        # add edge
                        self.graph.add_edge(src, dest, channel=None)
                else:
                    # pass all other source/destination pairs
                    pass

    def add_channels(self) -> None:
        """
        Assemble channels (internal buffers and delay buffer) and add them to src, dest and the global dict.
        """
        # create initial empty dictionary
        self.channels = dict()
        # loop over all node tuples
        for src in self.graph.nodes:
            for dest in self.graph.nodes:
                if src is not dest:  # skip src == dest
                    if isinstance(src, Kernel) and isinstance(dest, Kernel):  # case: KERNEL -> KERNEL
                        for inp in dest.graph.inputs:
                            if src.name == inp.name:
                                # create channel
                                name = src.name + "_" + dest.name
                                channel = {
                                    "name": name,
                                    "delay_buffer": self.kernel_nodes[dest.name].delay_buffer[src.name],
                                    "internal_buffer": dest.internal_buffer[src.name],
                                    "data_type": src.data_type
                                }
                                # add channel reference to global channel dictionary
                                self.channels[name] = channel
                                # add channel to both endpoints
                                src.outputs[dest.name] = channel
                                dest.inputs[src.name] = channel
                                # add channel to edge
                                self.graph[src][dest]['channel'] = channel
                                break
                    elif isinstance(src, Input) and isinstance(dest, Kernel):  # case: INPUT -> KERNEL
                        for inp in dest.graph.inputs:
                            if src.name == inp.name:
                                # create channel
                                name = src.name + "_" + dest.name
                                channel = {
                                    "name": name,
                                    "delay_buffer": self.kernel_nodes[dest.name].delay_buffer[src.name],
                                    "internal_buffer": dest.internal_buffer[src.name],
                                    "data_type": src.data_type
                                }
                                # add channel reference to global channel dictionary
                                self.channels[name] = channel
```

```
                    # add channel to both endpoints
                    src.outputs[dest.name] = channel
                    dest.inputs[src.name] = channel
                    # add to edge
                    self.graph[src][dest]['channel'] = channel
                    break
            elif isinstance(dest, Output):  # case: INPUT/KERNEL -> OUTPUT
                if src.name == dest.name:
                    # create channel
                    name = src.name + "_" + dest.name
                    channel = {
                        "name": name,
                        "delay_buffer": self.output_nodes[dest.name].delay_buffer[src.name],
                        "internal_buffer": {},
                        "data_type": src.data_type
                    }
                    # add channel reference to global channel dictionary
                    self.channels[name] = channel
                    # add channel to both endpoints
                    src.outputs[dest.name] = channel
                    dest.inputs[src.name] = channel
                    # add to edge
                    self.graph[src][dest]['channel'] = channel
            else:
                # pass all other source/destination pairs
                pass

def import_input(self) -> None:
    """
    Read all sections of the program input file.
    """
    inp = helper.parse_json(self.path)
    # get dimensions
    self.kernel_dimensions = len(inp["dimensions"])
    if self.kernel_dimensions == 1:  # 1D
        self.program = inp["program"]
        for entry in self.program:
            self.program[entry]["computation_string"] = \
                self.program[entry]["computation_string"].replace("[", "[i,j,")  # add two extra indices
        self.inputs = inp["inputs"]
        self.outputs = inp["outputs"]
        self.dimensions = [1, 1] + inp["dimensions"]  # add two extra dimensions
    elif self.kernel_dimensions == 2:  # 2D
        self.program = inp["program"]
        for entry in self.program:
            self.program[entry]["computation_string"] = self.program[entry]["computation_string"]\
                .replace("[","[i,")  # add extra index
        self.inputs = inp["inputs"]
        self.outputs = inp["outputs"]
```

```
            self.dimensions = [1] + inp["dimensions"]  # add extra dimension
        else:  # 3D
            self.program = inp["program"]
            self.inputs = inp["inputs"]
            self.outputs = inp["outputs"]
            self.dimensions = inp["dimensions"]

    def total_elements(self) -> int:
        """
        Reduction of the global problem size to a single scalar.
        :return: the global problem size as a scalar value
        """
        return functools.reduce(operator.mul, self.dimensions, 1)  # foldl (*) 1 [...]

    def create_kernels(self) -> None:
        """
        Create the kernels and add them to the networkx (library) graph.
        """
        # create all kernel objects and add them to the graph
        self.kernel_nodes = dict()
        for kernel in self.program:
            new_node = Kernel(name=kernel,
                              kernel_string=str(self.program[kernel]['computation_string']),
                              dimensions=self.dimensions,
                              data_type=self.program[kernel]['data_type'],
                              boundary_conditions=self.program[kernel]['boundary_condition'])
            self.graph.add_node(new_node)
            self.kernel_nodes[kernel] = new_node
        # create all input nodes (without data, we will add data in the simulator if necessary)
        self.input_nodes = dict()
        for inp in self.inputs:
            new_node = Input(name=inp,
                             data_type=self.inputs[inp]["data_type"],
                             data_queue=BoundedQueue(name=inp,
                                                     maxsize=self.total_elements(),
                                                     collection=[None] * self.total_elements()))
            self.input_nodes[inp] = new_node
            self.graph.add_node(new_node)
        # create all output nodes
        self.output_nodes = dict()
        for out in self.outputs:
            new_node = Output(name=out,
                              data_type=self.program[out]["data_type"],
                              dimensions=self.dimensions,
                              data_queue=BoundedQueue(name="dummy", maxsize=0))
            self.output_nodes[out] = new_node
            self.graph.add_node(new_node)

    def compute_kernel_latency(self) -> None:
```

```
    """
    Fill global dictionary of the individual kernel critical computation paths.
    """
    # create dict
    self.kernel_latency = dict()
    # compute kernel latency of each kernel
    for kernel in self.kernel_nodes:
        self.kernel_latency[kernel] = self.kernel_nodes[kernel].graph.max_latency

def at_least_one(self,
                 value: int) -> int:
    """
    This function returns the input value or at least one if it is less.
    :param value: input value
    :return: at least 1
    """
    return value if value > 0 else 1

def compute_delay_buffer(self) -> None:
    """
    Computes the delay buffer sizes in the graph by propagating all paths from the input arrays to the successors in
    topological order. Delay buffer entries should be of the format: kernel.input_paths:{
                                                                        "in1": [[a,b,c, pred1], [d,e,f, pred2],
                                                                        ...],
                                                                        "in2": [ ... ],
                                                                        ...
                                                                    }
    where inX are input arrays to the stencil chain and predY are the kernel predecessors/inputs
    """
    # get topological order for top-down walk through of the graph
    try:
        order = nx.topological_sort(self.graph)
    except nx.exception.NetworkXUnfeasible:
        raise ValueError("Cycle detected, cannot be sorted topologically!")
    # go through all nodes
    for node in order:
        # process delay buffer (no additional delay buffer will appear because of the topological order)
        for inp in node.input_paths:
            # compute maximum delay size per input
            max_delay = max(node.input_paths[inp])
            max_delay[2] += 1  # add an extra delay cycle for the processing in the kernel node
            # loop over all inputs and set their size relative to the max size to have data ready at the exact
            # same time
            for entry in node.input_paths[inp]:
                name = entry[-1]
                max_size = helper.convert_3d_to_1d(self.dimensions,
                                                   helper.list_subtract_cwise(max_delay[:-1], entry[:-1]))
                node.delay_buffer[name] = BoundedQueue(name=name, maxsize=max_size)
                node.delay_buffer[name].import_data([None] * node.delay_buffer[name].maxsize)
```

```
        # set input node delay buffers to 1
        if isinstance(node, Input):
            node.delay_buffer = BoundedQueue(name=node.name, maxsize=1, collection=[None])
        # propagate the path lengths (from input arrays over all ways) to the successors
        for succ in self.graph.successors(node):
            # add input node to all as direct input (=0 delay buffer)
            if isinstance(node, Input):
                # add emtpy list dictionary entry for enabling list append()
                if node.name not in succ.input_paths:
                    succ.input_paths[node.name] = []
                successor = [0] * len(self.dimensions)
                successor = successor + [node.name]
                succ.input_paths[node.name].append(successor)
            # add kernel node to all, but calculate the length first (predecessor + delay + internal, ..)
            elif isinstance(node, Kernel):  # add KERNEL

                # add latency, internal_buffer, delay_buffer
                internal_buffer = [0] * 3
                for item in node.graph.accesses:
                    internal_buffer = max(node.graph.accesses[item]) if max(
                        node.graph.accesses[item]) > internal_buffer else internal_buffer
                # latency
                latency = self.kernel_nodes[node.name].graph.max_latency
                # compute delay buffer and create entry
                for entry in node.input_paths:
                    # the first entry has to initialize the structure
                    if entry not in succ.input_paths:
                        succ.input_paths[entry] = []
                    # compute the actual delay buffer
                    delay_buffer = max(node.input_paths[entry][:])
                    # merge them together
                    total = [
                        i + d
                        for i, d in zip(internal_buffer, delay_buffer)
                    ]
                    # add the latency too
                    total[-1] += latency
                    total.append(node.name)
                    # add entry to paths
                    succ.input_paths[entry].append(total)

            else:  # NodeType.OUTPUT: do nothing
                continue

    def compute_critical_path_dim(self) -> List[int]:
        """
        Computes the max latency critical path through the graph in dimensional format.
        Note: Since we know the output nodes as well as the path lengths the critical path is just
        max { latency(node) + max { path_length(node) | node in output nodes }
```

```
        :return:
        """
        # init critical path length with zero
        critical_path_length = [0] * len(self.dimensions)
        # loop through all and update if our path with the extra kernel latency is larger then the largest that is
        # already stored
        for output in self.outputs:
            a = self.kernel_nodes[output].graph.max_latency
            b = max(self.kernel_nodes[output].input_paths)
            c = max(self.kernel_nodes[output].input_paths[b])
            c[2] += a
            critical_path_length = max(critical_path_length, c)
        # return final result
        return c[:-1]

    def compute_critical_path(self) -> int:
        """
        Computes the max latency critical path through the graph in scalar format.
        """
        return helper.dim_to_abs_val(self.compute_critical_path_dim(), self.dimensions)

    def report(self, name):
        print("Report of {}\n".format(name))

        print("dimensions of data array: {}\n".format(self.dimensions))

        print("channel info:")
        for u, v, channel in self.graph.edges(data='channel'):
            if channel is not None:
                print("internal buffers:\n {}".format(channel["internal_buffer"]))
                print("delay buffers:\n {}".format(channel["delay_buffer"]))
        print()

        print("field access info:")
        for node in self.kernel_nodes:
            print("node name: {}, field accesses: {}".format(node, self.kernel_nodes[node].graph.accesses))
        print()

        print("internal buffer size info:")
        for node in self.kernel_nodes:
            print("node name: {}, internal buffer size: {}".format(node,
                                                                    self.kernel_nodes[node].graph.buffer_size))
        print()

        print("internal buffer chunks info:")
        for node in self.kernel_nodes:
            print("node name: {}, internal buffer chunks: {}".format(node,
                                                                      self.kernel_nodes[node].internal_buffer))
        print()
```

```
print("delay buffer size info:")
for node in self.kernel_nodes:
    print("node name: {}, delay buffer size: {}".format(node, self.kernel_nodes[node].delay_buffer))
print()

print("path length info:")
for node in self.kernel_nodes:
    print("node name: {}, path lengths: {}".format(node, self.kernel_nodes[node].input_paths))
print()

print("latency info:")
for node in self.kernel_nodes:
    print("node name: {}, node latency: {}".format(node, self.kernel_nodes[node].graph.max_latency))
print()

print("critical path info:")
print("critical path length is {}\n".format(self.compute_critical_path()))

print("total buffer info:")
total = 0
for node in self.kernel_nodes:
    for u, v, channel in self.graph.edges(data='channel'):
        if channel is not None:
            total_delay = 0
            for item in channel["internal_buffer"]:
                total_delay += item.maxsize
            total_internal = 0
            total_delay += channel["delay_buffer"].maxsize
            total += total_delay + total_internal
print("total buffer size: {}\n".format(total))

print("input kernel string info:")
for node in self.kernel_nodes:
    print("input kernel string of {} is: {}".format(node, self.kernel_nodes[node].kernel_string))
print()

print("relative access kernel string info:")
for node in self.kernel_nodes:
    print("relative access kernel string of {} is: {}".format(node, self.kernel_nodes[node].
                                                                   generate_relative_access_kernel_string()))

print("instantiate optimizer...")
from optimizer import Optimizer
opt = Optimizer(self.kernel_nodes, self.dimensions)
bound = 12001
opt.minimize_fast_mem(communication_volume_bound=bound)
print("optimize fast memory usage with comm volume bound= {}".format(bound))
print("single stream comm vol for float32 is: {}".format(opt.single_comm_volume(4)))
```

```
        print("total buffer info:")
        total = 0
        for node in self.kernel_nodes:
            for u, v, channel in self.graph.edges(data='channel'):
                if channel is not None:
                    total_fast = 0
                    total_slow = 0
                    for entry in channel["internal_buffer"]:
                        if entry.swap_out:
                            print("internal buffer slow memory: {}, size: {}".format(entry.name, entry.maxsize))
                            total_slow += entry.maxsize
                        else:
                            print("internal buffer fast memory: {}, size: {}".format(entry.name, entry.maxsize))
                            total_fast += entry.maxsize
                    entry = channel["delay_buffer"]
                    if entry.swap_out:
                        print("delay buffer slow memory: {}, size: {}".format(entry.name, entry.maxsize))
                        total_slow += entry.maxsize
                    else:
                        print("delay buffer fast memory: {}, size: {}".format(entry.name, entry.maxsize))
                        total_fast += entry.maxsize
        print("buffer size slow memory: {} \nbuffer size fast memory: {}".format(total_slow, total_fast))


if __name__ == "__main__":

    """
        simple test stencil program for debugging

        usage: python3 kernel_chain_graph.py -stencil_file stencils/simulator12.json -plot -simulate -report -log-level 2
    """
    # instantiate the argument parser
    parser = argparse.ArgumentParser()
    parser.add_argument("-stencil_file")
    parser.add_argument("-plot", action="store_true")
    parser.add_argument("-log-level")
    parser.add_argument("-report", action="store_true")
    parser.add_argument("-simulate", action="store_true")
    args = parser.parse_args()
    # instantiate the KernelChainGraph
    chain = KernelChainGraph(path=args.stencil_file,
                             plot_graph=args.plot,
                             log_level=int(args.log_level))
    # simulate the design if argument -simulate is true
    if args.simulate:
        from simulator import Simulator

        sim = Simulator(input_config_name=re.match("[^\.]+", os.path.basename(args.stencil_file)).group(0),
```

```
                    input_nodes=chain.input_nodes,
                    input_config=chain.inputs,
                    kernel_nodes=chain.kernel_nodes,
                    output_nodes=chain.output_nodes,
                    dimensions=chain.dimensions,
                    write_output=False,
                    log_level=int(args.log_level))
    sim.simulate()

# output a report if argument -report is true
if args.report:
    chain.report(args.stencil_file)
    if args.simulate:
        sim.report()
```