compute_graph.py

```
import ast
from typing import List, Dict, Set

import networkx as nx

import helper
from base_node_class import BaseOperationNodeClass
from compute_graph_nodes import Name, Num, Binop, Call, Output, Subscript, Ternary, Compare, UnaryOp


class ComputeGraph:
    """
        The ComputeGraph class manages the inner data flow of a single computation respectively a single kernel
        including its properties e.g. latency, internal buffer sizes and field accesses.

        Notes:
            - Creation of a proper graph representation for the computation data flow graph.
            - Credits for node-visitor: https://stackoverflow.com/questions/33029168/how-to-calculate-an-equation-in-a-string-
python
            - More info: https://networkx.github.io/

        Note about ast structure:
            tree.body[i] : i-th expression
            tree.body[i] = Assign: of type: x = Expr
            tree.body[i].targets          ->
            tree.body[i].value = {BinOp, Name, Call}
            tree.body[i] = Expr:
            tree.body[i].value = BinOp    -> subtree: .left, .right, .op {Add, Mult, Name, Call}
            tree.body[i].value = Name     -> subtree: .id (name)
            tree.body[i].value = Call     -> subtree: .func.id (function), .args[i] (i-th argument)
            tree.body[i].value = Subscript -> subtree: .slice.value.elts[i]: i-th parameter in [i, j, k, ...]
    """

    def __init__(self,
                 verbose: bool = False) -> None:
        """
        Create new ComputeGraph with given initialization parameters.
        :param verbose: flag for console output logging
        """
        # set parameter variables
        self.verbose = verbose
        # read static parameters from config file
        self.config: Dict[str, int] = helper.parse_json("compute_graph.config")
        # initialize internal data structures
        self.graph: nx.DiGraph = nx.DiGraph()  # networkx (library) compute graph with compute_graph_nodes as nodes
        self.tree: type(ast) = None  # abstract syntax tree (python) data structure
        self.max_latency: int = -1  # (non-valid) initial value for the maximum latency (critical path) of the
        # computational tree
        self.inputs: Set[BaseOperationNodeClass] = set()  # link to all nodes that feed input into this computation
```

```
        self.outputs: Set[BaseOperationNodeClass] = set()  # link to all nodes this computation feeds data to
        self.min_index: Dict[str, List] = dict()  # per input array the last access index of the stencil
        self.max_index: Dict[str, List] = dict()  # per input array the furthest access index of the stencil
        self.buffer_size: Dict[str, List] = dict()  # size (dimensional) from the last to the first access (determines
        # the internal buffer size)
        self.accesses: Dict[str, List[List]] = dict()  # dictionary containing all field accesses for a specific
        # resource e.g. {"A":{[0,0,0],[0,1,0]}} for the stencil "res = A[i,j,k] + A[i,j+1,k]"

    @staticmethod
    def create_operation_node(node: ast,
                              number: int) -> BaseOperationNodeClass:
        """
        Create operation node of the correct type.
        :param node: abstract syntax tree node
        :param number: tree numbering
        :return: corresponding operation node
        """
        if isinstance(node, ast.Name):  # variables or array access
            return Name(node, number)
        elif isinstance(node, ast.Num):  # static value
            return Num(node, number)
        elif isinstance(node, ast.BinOp):  # binary operation
            return Binop(node, number)
        elif isinstance(node, ast.Call):  # function (e.g. sin, cos,..)
            return Call(node, number)
        elif isinstance(node, ast.Assign):  # assign operator (var = expr;)
            return Output(node, number)
        elif isinstance(node, ast.Subscript):  # array access (form: arr[i,j,k])
            return Subscript(node, number)
        elif isinstance(node, ast.IfExp):  # if/else clause of ternary operation
            return Ternary(node, number)
        elif isinstance(node, ast.Compare):  # comparison of ternary operation
            return Compare(node, number)
        elif isinstance(node, ast.UnaryOp):  # negation of value ('-' sign)
            return UnaryOp(node, number)
        else:
            raise Exception("Unknown AST type {}".format(type(node)))

    def setup_internal_buffers(self,
                               relative_to_center=True) -> None:
        """
        Set up minimum/maximum index and accesses for the internal data structures.
        :param relative_to_center: if true, the center of the stencil is at position [0,0,0] respecively 0, if false,
        the furthest element is at position [0,0,0] and all other accesses on the same input field are relative to that
        (i.e. negative)
        """
        # init dicts
        self.min_index = dict()  # min_index["buffer_name"] = [i_min, j_min, k_min]
        self.max_index = dict()
```

```
        self.buffer_size = dict()  # buffer_size["buffer_name"] = size
        # find min and max index
        for inp in self.inputs:
            if isinstance(inp, Subscript):  # subscript nodes only
                if inp.name in self.min_index:
                    if inp.index < self.min_index[inp.name]:  # check min
                        self.min_index[inp.name] = inp.index
                    if inp.index >= self.max_index[inp.name]:  # check max
                        self.max_index[inp.name] = inp.index
                else:  # first entry
                    self.min_index[inp.name] = inp.index
                    self.max_index[inp.name] = inp.index
                if inp.name not in self.accesses:  # create initial list
                    self.accesses[inp.name] = list()
                self.accesses[inp.name].append(inp.index)  # add entry
        # set buffer_size = max_index - min_index
        for buffer_name in self.min_index:
            self.buffer_size[buffer_name] = [abs(a_i - b_i) for a_i, b_i in zip(self.max_index[buffer_name],
                                                                                self.min_index[buffer_name])]
        # update access to have [0,0,0] for the max_index (subtract it from all)
        if not relative_to_center:
            for field in self.accesses:
                updated_entries = list()
                for entry in self.accesses[field]:
                    updated_entries.append(helper.list_subtract_cwise(entry, self.max_index[field]))
                self.accesses[field] = updated_entries

    def determine_inputs_outputs(self) -> None:
        """
        Fill up internal input and output data structures with the corresponding nodes.
        """
        # create empty sets
        self.inputs = set()
        self.outputs = set()
        # idea: do a tree-walk: all nodes with cardinality(predecessor)=0 are inputs, all nodes with cardinality(
        # successor)=0 are outputs
        for node in self.graph.nodes:
            if len(self.graph.pred[node]) == 0:
                self.inputs.add(node)
            if len(self.graph.succ[node]) == 0:
                self.outputs.add(node)

    def contract_edge(self,
                      u: BaseOperationNodeClass,
                      v: BaseOperationNodeClass) -> None:
        """
        Contract node v into node u.
        :param u: contractor node
        :param v: contracted node
```

```
        """
        # add edges of node v to node u
        for edge in self.graph.succ[v]:
            self.graph.add_edge(u, edge)
        for edge in self.graph.pred[v]:
            self.graph.add_edge(edge, u)
        # remove node v
        self.graph.remove_node(v)

    def generate_graph(self,
                       computation_string: str) -> nx.DiGraph:
        """
        Create networkx graph of the mathematical computation given in the computation_string.
        :param computation_string:
        :return networkx (library) graph of the computation with nodes from compute_graph_nodes
        """
        # generate abstract syntax tree
        self.tree = ast.parse(computation_string)
        # iterate over all equations (e.g. res=a+b;b=c+d; ...)
        for equation in self.tree.body:
            # check if base node is of type Expr or Assign
            if isinstance(equation, ast.Assign):
                lhs = self.create_operation_node(equation, 0)  # left hand side equation
                rhs = self.ast_tree_walk(equation.value, 1)  # right hand side of equation
                self.graph.add_edge(rhs, lhs)
        # merge ambiguous variables in tree (implies: merge of ast.Assign trees into a single tree)
        outp_nodes = list(self.graph.nodes)
        for outp in outp_nodes:
            if isinstance(outp, Name):
                inp_nodes = list(self.graph.nodes)
                for inp in inp_nodes:
                    if isinstance(outp, Subscript) and outp is not inp and outp.name == inp.name and \
                            outp.index == inp.index:
                        # only contract if the indices and the names match
                        self.contract_edge(outp, inp)
                    elif isinstance(outp, Name) and outp is not inp and outp.name == inp.name:
                        # contract nodes if the names match
                        self.contract_edge(outp, inp)
        # test if graph is now a single component (for directed graph: each non-output must have at least one successor)
        for node in self.graph.nodes:
            if not isinstance(node, Output) and len(self.graph.succ[node]) == 0:
                raise RuntimeError("Kernel-internal data flow is not single component (must be connected in the sense "
                                   "of a DAG).")
        return self.graph

    def ast_tree_walk(self,
                      node: ast,
                      number: int) -> BaseOperationNodeClass:
        """
```

```
Recursively walk through the abstract syntax tree structure.
:param node: current node
:param number: tree numbering
:return: reference of current node
"""
# create node
new_node = self.create_operation_node(node, number)
# add node to graph
self.graph.add_node(new_node)
# node type specific implementation of the tree walk
if isinstance(node, ast.BinOp):
    # do tree-walk recursively and get references to children (to create the edges to them)
    lhs = self.ast_tree_walk(node.left, ComputeGraph.child_left_number(number))  # left hand side
    rhs = self.ast_tree_walk(node.right, ComputeGraph.child_right_number(number))  # right hand side
    # add edges from parent to children
    self.graph.add_edge(lhs, new_node)
    self.graph.add_edge(rhs, new_node)
elif isinstance(node, ast.Call):
    # do tree-walk for all arguments
    if len(node.args) > 2:
        raise NotImplementedError("Current implementation does not support more than two arguments due"
                                  " to the binary tree numbering convention")
    # process first argument
    first = self.ast_tree_walk(node.args[0], ComputeGraph.child_left_number(number))
    self.graph.add_edge(first, new_node)
    # check if second argument exist
    if len(node.args) >= 2:
        second = self.ast_tree_walk(node.args[1], ComputeGraph.child_right_number(number))
        self.graph.add_edge(second, new_node)
elif isinstance(node, ast.Name):
    # nothing to do
    pass
elif isinstance(node, ast.Num):
    # nothing to do
    pass
elif isinstance(node, ast.Compare):
    # do tree-walk recursively and get references to children (to create the edges to them)
    lhs = self.ast_tree_walk(node.left, ComputeGraph.child_left_number(number))  # left hand side
    rhs = self.ast_tree_walk(node.comparators[0], ComputeGraph.child_right_number(number))  # right hand side
    # add edges from parent to children
    self.graph.add_edge(lhs, new_node)
    self.graph.add_edge(rhs, new_node)
elif isinstance(node, ast.IfExp):
    # do tree-walk recursively and get references to children (to create the edges to them)
    test = self.ast_tree_walk(node.test, 0)  # test clause
    true_path = self.ast_tree_walk(node.body, ComputeGraph.child_left_number(number))
    false_path = self.ast_tree_walk(node.orelse, ComputeGraph.child_right_number(number))
    # add edges from parent to children
    self.graph.add_edge(true_path, new_node)
```

```
            self.graph.add_edge(false_path, new_node)
            self.graph.add_edge(test, new_node)
        elif isinstance(node, ast.UnaryOp):
            # do tree-walk recursively and get references to child
            operand = self.ast_tree_walk(node.operand, ComputeGraph.child_right_number(number))
            # add edges form parent to child
            self.graph.add_edge(operand, new_node)
        return new_node

    @staticmethod
    def child_left_number(n: int) -> int:
        """
        Default tree numbering (number from root right to left downward per level).
        :param n: parent number
        :return: left child number
        """
        return 2 * n + 1

    @staticmethod
    def child_right_number(n: int) -> int:
        """
        Default tree numbering (number from root right to left downward per level).
        :param n: parent number
        :return: right child number
        """
        return 2 * n

    def plot_graph(self,
                   save_path: str = None) -> None:
        """
        Plot the compute graph graphically.
        :param save_path: filename of the output image, if none: do not save to file
        """
        # create drawing area
        import matplotlib.pyplot as plt  # import matplotlib only if graph plotting is set to true
        plt.figure(figsize=(20, 20))  # define drawing size. NOTE: must probably be a function of the number of nodes at
        # some point (for large graphs)
        plt.axis('off')
        # generate positions
        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()
        comp = list()
        for node in self.graph.nodes:
            if isinstance(node, Num):  # numerals
                nums.append(node)
```

```
    elif isinstance(node, Name) or isinstance(node, Subscript):  # variables
        names.append(node)
    elif isinstance(node, Binop) or isinstance(node, Call) or isinstance(node, UnaryOp):  # operations
        ops.append(node)
    elif isinstance(node, Output):  # outputs
        outs.append(node)
    elif isinstance(node, Ternary) or isinstance(node, Compare):  # comparison
        comp.append(node)
# create dictionary of the labels and add all of them
labels = dict()
for node in self.graph.nodes:
    labels[node] = node.generate_label()
# add nodes and edges
# name nodes
nx.draw_networkx_nodes(G=self.graph,
                       pos=positions,
                       nodelist=names,
                       node_color='orange',
                       node_size=3000,
                       node_shape='s',  # square
                       edge_color='black')
# output nodes
nx.draw_networkx_nodes(G=self.graph,
                       pos=positions,
                       nodelist=outs,
                       node_color='green',
                       node_size=3000,
                       node_shape='s')
# numeral nodes
nx.draw_networkx_nodes(G=self.graph,
                       pos=positions,
                       nodelist=nums,
                       node_color='#007acc',
                       node_size=3000,
                       node_shape='s')
# ternary operator nodes
nx.draw_networkx_nodes(G=self.graph,
                       pos=positions,
                       nodelist=comp,
                       node_color='#009999',
                       node_size=3000,
                       node_shape='o')  # circle
# operation nodes and edges between all nodes
nx.draw_networkx(G=self.graph,
                 pos=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)
    # add labels
    nx.draw_networkx_labels(G=self.graph,
                            pos=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:
        plt.savefig(save_path)
    # plot it
    plt.show()

def try_set_max_latency(self,
                        new_val: int) -> bool:
    """
    Update the maximum latency of the compute graph.
    :param new_val: new maximum latency candidate
    :return: whether or not the candidate is the new maximum
    """
    if self.max_latency <= new_val:
        self.max_latency = new_val
        return True
    else:
        return False

def calculate_latency(self) -> None:
    """
    Find critical path in the computation tree.
    """
    # idea: do a longest-path tree-walk (since the graph is a DAG (directed acyclic graph) we can do that
    for node in self.graph.nodes:
        if isinstance(node, Output):  # start at the output nodes and walk the tree up to the input nodes
            node.latency = 1
            self.try_set_max_latency(node.latency)
            self.latency_tree_walk(node)

def latency_tree_walk(self,
                      node: BaseOperationNodeClass) -> None:
    """
    Computation tree walk for latency calculation.
```

```
        :param node: current node
        """
        # check node type
        if isinstance(node, Name) or isinstance(node, Num) or isinstance(node, Subscript):  # variable or numeral:
            # no additional latency
            # copy parent latency to children
            for child in self.graph.pred[node]:
                child.latency = node.latency
                self.latency_tree_walk(child)
        elif isinstance(node, Binop) or isinstance(node, Call):  # function calls: additional latency of the function
            # added
            # get op latency from config
            op_latency = self.config["op_latency"][node.name]
            # add latency to children
            for child in self.graph.pred[node]:
                child.latency = max(child.latency, node.latency + op_latency)
                self.latency_tree_walk(child)
        elif isinstance(node, Output):  # output: no additional latency
            # copy parent latency to children
            for child in self.graph.pred[node]:
                child.latency = node.latency
                self.latency_tree_walk(child)
        elif isinstance(node, Ternary):  # function calls: additional latency of the conditional operator added
            # get op latency from config
            op_latency = self.config["op_latency"]["conditional"]
            # add latency to children
            for child in self.graph.pred[node]:
                child.latency = max(child.latency, node.latency + op_latency)
                self.latency_tree_walk(child)
        elif isinstance(node, Compare):  # comparison: additional latency of the comparison operator added
            # get op latency from config
            op_latency = self.config["op_latency"]["comparison"]
            # add latency to children
            for child in self.graph.pred[node]:
                child.latency = max(child.latency, node.latency + op_latency)
                self.latency_tree_walk(child)
        elif isinstance(node, UnaryOp):  # unary operator
            # get op latency from config
            op_latency = self.config["op_latency"][node.name]
            # add latency to children
            for child in self.graph.pred[node]:
                child.latency = max(child.latency, node.latency + op_latency)
                self.latency_tree_walk(child)
        else:
            raise NotImplementedError("Node type {} has not been implemented yet.".format(type(node)))
        self.try_set_max_latency(node.latency)


if __name__ == "__main__":
```

compute_graph.py

```
"""
    simple debugging example
"""
computation = "res = -a if (a+1 > b-c) else b; b = d + e"
graph = ComputeGraph()
graph.generate_graph(computation)
graph.calculate_latency()
# graph.plot_graph("compute_graph_example.png")  # write graph to file
graph.plot_graph()
```