kernel.py

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

import dace.types

import helper
from base_node_class import BaseKernelNodeClass, BaseOperationNodeClass
from bounded_queue import BoundedQueue
from calculator import Calculator
from compute_graph import ComputeGraph
from compute_graph import Name, Num, Binop, Call, Output, Subscript, Ternary, Compare, UnaryOp


class Kernel(BaseKernelNodeClass):
    """
        The Kernel class is a subclass of the BaseKernelNodeClass and represents the actual kernel node in the
        KernelChainGraph. This class is able to read from predecessors, process it according to the stencil expression
        and write the result to the successor channels. In addition it analyses the buffer sizes and latencies of the
        computation according  to the defined latencies.
    """

    def __init__(self,
                 name: str,
                 kernel_string: str,
                 dimensions: List[int],
                 data_type: dace.types.typeclass,
                 boundary_conditions: Dict[str, Dict[str, str]],
                 plot_graph: bool = False,
                 verbose: bool = False) -> None:
        """

        :param name: name of the kernel
        :param kernel_string: mathematical expression representing the stencil computation
        :param dimensions: global dimensions / problem size (i.e. size of the input array
        :param data_type: data type of the result produced by this kernel
        :param boundary_conditions: dictionary of the boundary condition for each input channel/field
        :param plot_graph: flag indicating whether the underlying graph is being drawn
        :param verbose: flag for console output logging
        """
        # initialize the superclass
        super().__init__(name, BoundedQueue(name="dummy", maxsize=0), data_type)
        # store arguments
        self.kernel_string: str = kernel_string  # raw kernel string input
        self.dimensions: List[int] = dimensions  # input array dimensions [dimX, dimY, dimZ]
        self.boundary_conditions: Dict[str, Dict[str, str]] = boundary_conditions  # boundary_conditions[field_name]
        self.verbose = verbose
        # read static parameters from config
        self.config: Dict = helper.parse_json("kernel.config")
```

```
        self.calculator: Calculator = Calculator()
        # set simulator initial parameters
        self.all_available = False
        self.not_available = set()
        # analyze input
        self.graph: ComputeGraph = ComputeGraph()
        self.graph.generate_graph(kernel_string)  # generate the ast computation graph from the mathematicl expression
        self.graph.calculate_latency()  # calculate the latency in the compuation tree to find the critical path
        self.graph.determine_inputs_outputs()  # sort out input nodes (field accesses and constant values) and output
        # nodes
        self.graph.setup_internal_buffers()
        # set plot path (if plot is set to True)
        if plot_graph:
            self.graph.plot_graph(name + ".png")
        # init sim specific params
        self.var_map: Dict[
            str, float] = dict()  # mapping between variable names and its (current) value: var_map[var_name] =
        # var_value
        self.read_success: bool = False  # flag indicating if read has been successful from all input nodes (=> ready
        # to execute)
        self.exec_success: bool = False  # flag indicating if the execution has been successful
        self.result: float = float('nan')  # execution result of current iteration (see program counter)
        self.outputs: Dict[str, BoundedQueue] = dict()
        # output delay queue: for simulation of calculation latency, fill it up with bubbles
        self.out_delay_queue: BoundedQueue = BoundedQueue(name="delay_output",
                                                          maxsize=self.graph.max_latency + 1,
                                                          collection=[None] * self.graph.max_latency)
        # setup internal buffer queues
        self.internal_buffer: Dict[str, BoundedQueue] = dict()
        self.setup_internal_buffers()
        # this method takes care of the (falsely) executed kernel in case of not having a field access at [0,0,0]
        # present and the implication that there might be only fields out of bound s.t. there is a result produced,
        # but there should not be a result yet (see paper example ref# TODO)
        self.dist_to_center: Dict = dict()
        self.set_up_dist_to_center()
        self.center_reached = False
        # add performance metric fields
        self.max_del_buf_usage = dict()
        # for mean
        self.buf_usage_sum = dict()
        self.buf_usage_num = dict()
        self.init_metric = False
        self.PC_exec_start = helper.convert_3d_to_1d(self.dimensions, self.dimensions)  # upper bound
        self.PC_exec_end = 0  # lower bound

    def print_kernel_performance(self):
        """
        Print performance metric data.
        """
```

```
        print("##############################")
        for input in set(self.inputs).union(set(self.outputs)):
            print("##############################")
            print("input buffer name: {}".format(input))
            print("max buffer usage: {}".format(self.max_del_buf_usage[input]))
            print("average buffer usage: {}".format(self.buf_usage_sum[input] / self.buf_usage_num[input]))
        print("total execution time (from first exec to last): {}".format(self.PC_exec_end - self.PC_exec_start))

    def update_performance_metric(self):
        """
        Update buffer size values for performance evalution purpose.
        """
        # check if dict has been initialized
        if not self.init_metric:
            # init all keys
            for input in self.inputs:
                self.max_del_buf_usage[input] = 0
                self.buf_usage_num[input] = 0
                self.buf_usage_sum[input] = 0
            for output in self.outputs:
                self.max_del_buf_usage[output] = 0
                self.buf_usage_num[output] = 0
                self.buf_usage_sum[output] = 0
        # update maximum delay buf usage
        # inputs
        for input in self.inputs:
            buffer = self.inputs[input]
            self.max_del_buf_usage[input] = max(self.max_del_buf_usage[input],
                                                len([x for x in buffer['delay_buffer'].queue if x is not None]))
            self.buf_usage_num[input] += 1
            self.buf_usage_sum[input] += len([x for x in buffer['delay_buffer'].queue if x is not None])
        # outputs
        for output in self.outputs:
            buffer = self.outputs[output]
            self.max_del_buf_usage[output] = max(self.max_del_buf_usage[output],
                                                 len([x for x in buffer['delay_buffer'].queue if x is not None]))
            self.buf_usage_num[output] += 1
            self.buf_usage_sum[output] += len([x for x in buffer['delay_buffer'].queue if x is not None])

    def set_up_dist_to_center(self):
        """
        Computes for all fields/channels the distance from the furthest field access to the center of the stencil
        ([0,0,0,]).
        """
        for item in self.graph.accesses:
            furthest = max(self.graph.accesses[item])
            self.dist_to_center[item] = helper.dim_to_abs_val(furthest, self.dimensions)

    def iter_comp_tree(self,
```

```
                node: BaseOperationNodeClass,
                index_relative_to_center=True,
                replace_negative_index=False,
                python_syntax=False) -> str:
"""
Iterate through the computation tree in order to generate the kernel string (according to some properties
e.g. relative to center or replace negative index.
:param node: current node in the tree
:param index_relative_to_center: indication wheter the zero index should be at the center of the stencil or the
furthest element
:param replace_negative_index: replace the negativ sign '-' by n in order to create variable names that are not
being split up by the python expression parser (Calculator)
:return: computation string of the subgraph
"""
# get predecessor list
pred = list(self.graph.graph.pred[node])
# differentiate cases for each node type
if isinstance(node, Binop):  # binary operation
    # extract expression elements
    lhs = pred[0]  # left hand side
    rhs = pred[1]  # right hand side
    # recursively compute the child string
    lhs_str = self.iter_comp_tree(lhs, index_relative_to_center, replace_negative_index, python_syntax)
    rhs_str = self.iter_comp_tree(rhs, index_relative_to_center, replace_negative_index, python_syntax)
    # return formatted string
    return "({} {} {})".format(lhs_str, node.generate_op_sym(), rhs_str)
elif isinstance(node, Call):  # function call
    # extract expression element
    expr = pred[0]
    # recursively compute the child string
    expr_str = self.iter_comp_tree(expr, index_relative_to_center, replace_negative_index, python_syntax)
    # return formatted string
    return "{}({})".format(node.name, expr_str)
elif isinstance(node, Name) or isinstance(node, Num):
    # return formatted string
    return str(node.name)  # variable name
elif isinstance(node, Subscript):
    # compute correct indexing according to the flag
    if index_relative_to_center:
        dim_index = node.index
    else:
        dim_index = helper.list_subtract_cwise(node.index, self.graph.max_index[node.name])
    # break down index from 3D (i.e. [X,Y,Z]) to 1D
    word_index = self.convert_3d_to_1d(dim_index)
    # replace negative sign if the flag is set
    if replace_negative_index and word_index < 0:
        return node.name + "[" + "n" + str(abs(word_index)) + "]"
    else:
        return node.name + "[" + str(word_index) + "]"
```

```
        elif isinstance(node, Ternary):  # ternary operator of the form true_expr if comp else false_expr
            # extract expression elements
            compare = [x for x in pred if type(x) == Compare][0]  # comparison
            lhs = [x for x in pred if type(x) != Compare][0]  # left hand side
            rhs = [x for x in pred if type(x) != Compare][1]  # right hand side
            # recursively compute the child string
            compare_str = self.iter_comp_tree(compare, index_relative_to_center, replace_negative_index, python_syntax)
            lhs_str = self.iter_comp_tree(lhs, index_relative_to_center, replace_negative_index, python_syntax)
            rhs_str = self.iter_comp_tree(rhs, index_relative_to_center, replace_negative_index, python_syntax)
            # return formatted string
            if python_syntax:
                return "(({}) if ({}) else ({}))".format(lhs_str, compare_str, rhs_str)
            else:  # C++ ternary operator syntax
                return "(({}) ? ({}) : ({}))".format(compare_str, lhs_str, rhs_str)
        elif isinstance(node, Compare):  # comparison
            # extract expression element
            lhs = pred[0]
            rhs = pred[1]
            # recursively compute the child string
            lhs_str = self.iter_comp_tree(lhs, index_relative_to_center, replace_negative_index, python_syntax)
            rhs_str = self.iter_comp_tree(rhs, index_relative_to_center, replace_negative_index, python_syntax)
            # return formatted string
            return "{} {} {}".format(lhs_str, str(node.name), rhs_str)
        elif isinstance(node, UnaryOp):  # unary operations e.g. negation
            # extract expression element
            expr = pred[0]
            # recursively compute the child string
            expr_str = self.iter_comp_tree(node=expr, index_relative_to_center=index_relative_to_center,
                                           replace_negative_index=replace_negative_index, python_syntax=python_syntax)
            # return formatted string
            return "({}{})".format(node.generate_op_sym(), expr_str)
        else:
            raise NotImplementedError("iter_comp_tree is not implemented for node type {}".format(type(node)))

    def generate_relative_access_kernel_string(self,
                                               relative_to_center=True,
                                               replace_negative_index=False,
                                               python_syntax=False) -> str:
        """
        Generates the relative (either to the center or to the furthest field access) access kernel string which
        is necessary for the code generator HLS tool.
        :param relative_to_center: if true, the center is at zero, otherwise the furthest access is at zero
        :param replace_negative_index: if true, all negative access signs e.g. arrA_-20 gets replaced by n e.g.
        arrA_n20 in order to be correctly recognised as a single variable name.
        :return: the generated relative access kernel string
        """
        # format: 'res = vdc[index1] + vout[index2]'
        res = []
        # treat named nodes
```

```python
        for n in self.graph.graph.nodes:
            if isinstance(n, Name):
                res.append(n.name + " = " + self.iter_comp_tree(
                    list(self.graph.graph.pred[n])[0], relative_to_center, replace_negative_index, python_syntax))
        # treat output node(s)
        output_node = [
            n for n in self.graph.graph.nodes if isinstance(n, Output)
        ]
        if len(output_node) != 1:
            raise Exception("Expected a single output node")
        output_node = output_node[0]
        # concatenate the expressions
        res.append("res = " + self.iter_comp_tree(node=list(self.graph.graph.pred[output_node])[0],
                                                  index_relative_to_center=relative_to_center,
                                                  replace_negative_index=replace_negative_index,
                                                  python_syntax=python_syntax))
        return "; ".join(res)

    def reset_old_compute_state(self) -> None:
        """
        Reset the internal kernel simulator state in order to be prepared for the next iteration.
        """
        self.var_map = dict()
        self.read_success = False
        self.exec_success = False
        self.result = None

    def convert_3d_to_1d(self,
                         index: List[int]) -> int:
        """
        Convert [i,j,k] to flat 1D array index using the given dimensions [dimX, dimY, dimZ]
        :param index: index array to be converted to 1D
        :return: scalar value of the computation i*dimY*dimZ + j*dimZ + k = (i*dimY + j)*dimZ + k
        """
        # do computation: index = i*dimY*dimZ + j*dimZ + k = (i*dimY + j)*dimZ + k if the array is not empty
        if not index:
            return 0  # empty list
        return helper.dim_to_abs_val(index, self.dimensions)

    def remove_duplicate_accesses(self,
                                  inp: List) -> List:
        """
        Remove duplicate accesses of the given input array.
        :param inp: List with duplicates.
        :return: List without duplicates.
        """
        tuple_set = set(tuple(row) for row in inp)
        return [list(t) for t in tuple_set]
```

kernel.py

```python
def setup_internal_buffers(self) -> None:
    """
    Create and split the internal buffers according to the pipline model (see paper example ref# TODO)
    :return:
    """
    # remove duplicate accesses
    for item in self.graph.accesses:
        self.graph.accesses[item] = self.remove_duplicate_accesses(self.graph.accesses[item])
    # slice the internal buffer into junks of accesses
    for buf_name in self.graph.buffer_size:
        # create empty list and sort the accesses according to their relative position
        self.internal_buffer[buf_name]: List[BoundedQueue] = list()
        list.sort(self.graph.accesses[buf_name], reverse=True)
        # split according to the cases
        if len(self.graph.accesses[buf_name]) == 0:  # empty list
            pass
        elif len(self.graph.accesses[buf_name]) == 1:  # single entry list
            # this line would add an additional internal buffer for fields that only have a single access
            self.internal_buffer[buf_name].append(BoundedQueue(name=buf_name, maxsize=1, collection=[None]))
        else:  # many entry list
            # iterate through all of them and split them into correct sizes
            itr = self.graph.accesses[buf_name].__iter__()
            pre = itr.__next__()
            for item in itr:
                curr = item
                # calculate size of buffer
                diff = abs(helper.dim_to_abs_val(helper.list_subtract_cwise(pre, curr), self.dimensions))
                if diff == 0:  # two accesses on same field
                    pass
                else:
                    self.internal_buffer[buf_name].append(
                        BoundedQueue(name=buf_name, maxsize=diff, collection=[None] * diff))
                pre = curr

def buffer_position(self,
                    access: BaseKernelNodeClass) -> int:
    """
    Computes the offset position within the buffer list
    :param access: the access index we want to know the buffer position
    :return: the offset from the access
    """
    return self.convert_3d_to_1d(self.graph.min_index[access.name]) - self.convert_3d_to_1d(access.index)

def index_to_ijk(self,
                 index: List[int]):
    """
    Creates a string of the access (for variable name generation).
    :param index: access
    :return: created string
```

```
    """
    # current implementation only supports 3 dimension (default)
    if len(index) == 3:
        """
        # v1:
        return "[i{},j{},k{}]".format(
            "" if index[0] == 0 else "+{}".format(index[0]),
            "" if index[1] == 0 else "+{}".format(index[1]),
            "" if index[2] == 0 else "+{}".format(index[2])
        )
        # v2:
        return "_{}_{}_{}".format(index[0], index[1], index[2])
        """
        # compute absolute index
        ind = helper.convert_3d_to_1d(self.dimensions, index)
        # return formatted string
        return "_{}".format(ind) if ind >= 0 else "_n{}".format(abs(ind))
    else:
        raise NotImplementedError(
            "Method index_to_ijk has not been implemented for |indices|!=3, here: |indices|={}".format(len(index)))

def buffer_number(self,
                  node: Subscript):
    """
    Computes the index within the internal buffer array for accessing the input node.
    :param node: input node
    :return: index (-1: delay buffer, >= 0: internal buffer index)
    """
    # select all matching inputs
    selected = [x.index for x in self.graph.inputs if x.name == node.name]
    # remove duplicates
    selected_unique = self.remove_duplicate_accesses(selected)
    # sort them to have them ordered by the access
    ordered = sorted(selected_unique, reverse=True)
    # get the position within the sorted list
    result = ordered.index(node.index)
    return result - 1

def get_global_kernel_index(self) -> List[int]:
    """
    Return the current position (simulator, program counter) within the comutation as a list of the form
    [i,j,k].
    :return: current global kernel position as [i,j,k]
    """
    # get dimensions and PC
    index = self.dimensions
    number = self.program_counter
    # convert the absolute value (PC) to its corresponding position in the given 3D space.
    n = len(index)
```

```
    all_dim = functools.reduce(operator.mul, index, 1) // index[0]  # integer arithmetic
    output = list()
    for i in range(1, n + 1):
        output.append(number // all_dim)
        number -= output[-1] * all_dim
        if i < n:
            all_dim = all_dim // index[i]
    return output

def is_out_of_bound(self,
                    index: List[int]) -> bool:
    """
    Checks whether the current access is within bounds or not.
    :param index: access index
    :return: true: within bounds, false: otherwise
    """
    # check all dimensions boundary
    for i in range(len(index)):
        if index[i] < 0 or index[i] >= self.dimensions[i]:
            return True
    return False

def get_data(self,
             inp: Subscript,
             global_index: List[int],
             relative_index: List[int]):
    """
    Returns data of current stencil access (could be real data or boundary condition)
    :param inp: array field access
    :param global_index: center location of current stencil
    :param relative_index: offset from center of stencil
    :return: data
    """
    # get the access index
    access_index = helper.list_add_cwise(global_index, relative_index)
    """
        Boundary Condition
    """
    # check if it is within bounds
    if self.is_out_of_bound(access_index):
        if self.boundary_conditions[inp.name]["type"] == "constant":
            return self.boundary_conditions[inp.name]["value"]
        elif self.boundary_conditions[inp.name]["type"] == "copy":
            raise NotImplementedError("Copy boundary conditions have not been implemented yet.")
        else:
            raise NotImplementedError("We currently do not support boundary conditions of type {}".format(
                self.boundary_conditions[inp.name]["type"]))
    """
        Data Access
```

```
        """
        # get index position within the buffers
        pos = self.buffer_number(inp)
        if pos == -1:  # delay buffer
            return self.inputs[inp.name]["delay_buffer"].try_peek_last()
        elif pos >= 0:  # internal buffer
            return self.inputs[inp.name]["internal_buffer"][pos].try_peek_last()

    def test_availability(self):
        """
        Check if all accesses are available (=> ready for execution). In addition to that, the method delivers all
        accesses that are not available yet.
        :return: true: all available, false: otherwise
        """
        # set initial value and init set
        all_available = True
        self.not_available = set()
        # iterate through all inputs
        for inp in self.graph.inputs:
            # case split for types
            if isinstance(inp, Num):  # numerals are always available
                pass
            elif len(self.inputs[inp.name]['internal_buffer']) == 0:  # no internal buffer
                pass
            elif isinstance(inp, Subscript):  # normal subscript access
                # get current internal state position in [i,j,k] format
                gki = self.get_global_kernel_index()
                # check bound, out of bound is handled by the boundary condition automatically (always available for
                # constant)
                if self.is_out_of_bound(helper.list_add_cwise(inp.index, gki)):
                    pass
                else:  # within bounds
                    # get position and check if the value (not None) is available
                    index = self.buffer_number(inp)
                    if index == -1:  # delay buffer
                        if self.inputs[inp.name]['delay_buffer'].try_peek_last() is None or \
                                self.inputs[inp.name]['delay_buffer'].try_peek_last() is False:
                            all_available = False
                            self.not_available.add(inp.name)
                    elif 0 <= index < len(self.inputs[inp.name]['internal_buffer']):  # internal buffer
                        if self.inputs[inp.name]['internal_buffer'][index].try_peek_last() is False \
                                or self.inputs[inp.name]['internal_buffer'][index].try_peek_last() is None:
                            all_available = False
                            self.not_available.add(inp.name)
                    else:
                        raise Exception("index out of bound: {}".format(index))

        return all_available
```

```python
def move_forward(self,
                 items: Dict[str, Dict]) -> None:
    """
    Move all items within the internal and delay buffer one element forward.
    :param items:
    :return:
    """
    # move all forward
    for name in items:
        if len(items[name]['internal_buffer']) == 0:  # no internal buffer
            pass
        elif len(self.inputs[name]['internal_buffer']) == 1:  # single internal buffer
            items[name]['internal_buffer'][0].dequeue()
            items[name]['internal_buffer'][0].enqueue(items[name]['delay_buffer'].dequeue())
        else:  # many internal buffers
            # iterate over them and move all one forward
            index = len(items[name]['internal_buffer']) - 1
            pre = items[name]['internal_buffer'][index - 1]
            next = items[name]['internal_buffer'][index]
            next.dequeue()
            while index > 0:
                next.enqueue(pre.dequeue())
                next = pre
                index -= 1
                pre = items[name]['internal_buffer'][index - 1]
            items[name]['internal_buffer'][0].enqueue(items[name]['delay_buffer'].dequeue())

def decrement_center_reached(self):
    """
    Decrement counter for reaching the center. As soon as this counter reaches zero, the computed output values
    are valid and should be forwarded to the successors channels.
    """
    # decrement all
    for item in self.dist_to_center:
        if self.inputs[item]['delay_buffer'].try_peek_last() is not None:
            self.dist_to_center[item] -= 1

def try_read(self) -> bool:
    """
    This is the implementation of the kernel reading functionality of the simulator. It tries to read from all
    input channels and indicates if this has been done with success.
    """
    # check if all inputs are available
    self.all_available = self.test_availability()
    # get all values and put them into the variable map
    if self.all_available:
        for inp in self.graph.inputs:
            # read inputs into var_map
            if isinstance(inp, Num):  # case numerals
```

```
                self.var_map[inp.name] = float(inp.name)
            elif isinstance(inp, Name):  # case variable names
                # get value from internal_buffer
                try:
                    # check for duplicate
                    if not self.var_map.__contains__(inp.name):
                        self.var_map[inp.name] = self.internal_buffer[inp.name].peek(self.buffer_position(inp))
                except Exception as ex:  # do proper diagnosis
                    self.diagnostics(ex)
            elif isinstance(inp, Subscript):  # case array accesses
                # get value from internal buffer
                try:
                    name = inp.name + self.index_to_ijk(inp.index)
                    if not self.var_map.__contains__(name):
                        self.var_map[name] = self.get_data(inp=inp,
                                                           global_index=self.get_global_kernel_index(),
                                                           relative_index=inp.index)
                except Exception as ex:  # do proper diagnosis
                    self.diagnostics(ex)
        # set kernel flag indicating the the read has been successful
        self.read_success = self.all_available
        # test center reached
        self.decrement_center_reached()
        self.center_reached = True
        for item in self.dist_to_center:
            if self.dist_to_center[item] >= 0:
                self.center_reached = False
        # either move all inputs forward or those that are not available yet
        if self.center_reached:
            if self.all_available:
                self.move_forward(self.inputs)
            else:
                not_avail_dict = dict()
                for item in self.not_available:
                    not_avail_dict[item] = self.inputs[item]
                self.move_forward(not_avail_dict)
        else:
            not_reached_dict = dict()
            for item in self.dist_to_center:
                if self.dist_to_center[item] >= 0:
                    not_reached_dict[item] = self.inputs[item]
            self.move_forward(not_reached_dict)
        return self.all_available

    def try_execute(self):
        """
        This is the implementation of the kernel execution functionality of the simulator. It executes the stencil
        computation for the current variable mapping that was set up by the try_read() function.
        """
```

```
    # check if read has been succeeded
    if self.center_reached and self.read_success and \
            0 <= self.program_counter < functools.reduce(operator.mul, self.dimensions, 1):
        # execute calculation
        try:
            # get computation string
            computation = self.generate_relative_access_kernel_string(relative_to_center=True,
                                                                       replace_negative_index=True,
                                                                       python_syntax=True) \
                .replace("[", "_").replace("]", "").replace(" ", "")
            # compute result and
            self.result = self.data_type(self.calculator.eval_expr(self.var_map, computation))
            # write result to latency-simulating buffer
            self.out_delay_queue.enqueue(self.result)
            # update performance metric
            self.PC_exec_start = min(self.PC_exec_start, self.program_counter)
            self.PC_exec_end = max(self.PC_exec_end, self.program_counter)
            # increment the program counter
            self.program_counter += 1
        except Exception as ex:  # do proper diagnosis upon an exception
            self.diagnostics(ex)
    else:
        # write bubble to latency-simulating buffer
        self.out_delay_queue.enqueue(None)

def try_write(self):
    """
    This is the implementation of the kernel write functionality of the simulator. It writes the output element to
    its successor channels.
    """
    # read last element of the delay queue
    data = self.out_delay_queue.dequeue()
    # write result to all output queues
    for outp in self.outputs:
        try:
            self.outputs[outp]["delay_buffer"].enqueue(data)  # use delay buffer to be consistent with others,
            # delay buffer is used to write to the output data queue here
        except Exception as ex:  # do proper diagnosis upon an exception
            self.diagnostics(ex)

def diagnostics(self,
                ex: Exception) -> None:
    """
    Interface for error overview reporting (gets called in case of an exception)

    - goal:
            - get an overview over the whole stencil chain state in case of an error
                - maximal and current size of all buffers
                - type of phase (saturation/execution)
```

```
            - efficiency (#execution cycles / #total cycles)
        :param ex: the exception that arose
        """
        print("######################################")
        print("Diagnosis output of kernel {}".format(self.name))
        print("Program Counter: {}".format(self.program_counter))
        print("All inputs available? {}".format(self.all_available))
        print("Center reached? {}".format(self.center_reached))
        print("Exception traceback:")
        if ex is not None:
            import traceback
            try:
                raise ex
            except Exception:
                print(traceback.format_exc())  # inputs
        for input in self.inputs:
            buffer = self.inputs[input]
            print("Buffer info from input {}".format(input))
            # delay buffer
            print("Delay buffer max size: {}, current size: {}".format(buffer['delay_buffer'].maxsize,
                                                                     buffer['delay_buffer'].size()))
            print("Delay buffer data: {}".format(buffer['delay_buffer'].queue))
            # internal buffer
            data = list(map(lambda x: x.queue, buffer['internal_buffer']))
            print("Internal buffer data: {}".format(data))
        # latency sim buffer
        print("Latency simulation buffer data: {}".format(self.out_delay_queue.queue))
        # output
        for output in self.outputs:
            buffer = self.outputs[output]
            print("Buffer info from output {}".format(output))
            # delay buffer
            print("Delay buffer max size: {}, current size: {}".format(buffer['delay_buffer'].maxsize,
                                                                     buffer['delay_buffer'].size()))
            print("Delay buffer data: {}".format(buffer['delay_buffer'].queue))
            # internal buffer
            data = list(map(lambda x: x.queue, buffer['internal_buffer']))
            print("Internal buffer data: {}".format(data))


if __name__ == "__main__":
    """
        simple test kernel for debugging
    """
    # global dimensions
    dim = [100, 100, 100]
    # instantiate kernel
    kernel = Kernel(name="dummy",
                    kernel_string="res = a[i+1,j+1,k+1] + a[i+1,j,k] + a[i-1,j-1,k-1] + a[i+1,j+1,k] + (-a[i,j,k])",
```

```
                dimensions=dim,
                boundary_conditions={"a": {
                    "type": "constant",
                    "value": 0.0}},
                data_type=dace.types.float64)
print("Kernel string conversion:")
print("dimensions are: {}".format(dim))
print(kernel.kernel_string)
print(kernel.generate_relative_access_kernel_string(relative_to_center=False))
print()
```