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

import helper
from kernel_chain_graph import KernelChainGraph

"""
    Intro:
        This is a buffer size and bandwidth estimate for the whole dynamical core of the COSMO weather model. With only
        a few actual kernel chains ported and having the shape of the whole (production) dynamical core, we try to
        estimate if it the buffer space and bandwidth requirements are in the order of resources we have available on a
        single or an cluster of at most 32 Intel Stratix 10 FPGAs.

    Assumptions:
        - data type: float (32bit IEEE 754)
        - iteration over the smaller two dimensions possible (e.g. 1024x1024x64 -> iterate over 64x1024)
        - estimate for critical path length (#cycles):
            Stencil Chains (automatic output):
                fastwaves:     [3, 1, 54]
                diffusion_min: [3, 0, 24]
                advection_min: [7, 4, 26]
                -> use the mean of the three: [4, 2, 35]
        - clock frequency: 200Mhz
        - fast memory (Stratix 10): 25MB
        - bandwidth to slow memory (Stratix 10): 86.4GB/s
        - we assume that as soon as the pipeline is saturated, we can produce one result per cycle

"""
# datatype: for the moment, we assume float (32bit) are precise enough
_SIZEOF_DATATYPE: int = 4  # in bytes

# we assume that we can iterate over the smaller dimension by transposition of all data arrays, if not, change this to
# [64, 1024, 1024] (would lead to an additional factor 16 of buffer space growth)
_DIMENSIONS: List[int] = [1024, 1024, 64]  # longitude x latitude x altitude

#  FPGA clock frequency: 200Mhz
_FPGA_CLOCK_FREQUENCY: int = int(2e8)

# available FGPA fast memory (Stratix 10): 25MB
_FPGA_FAST_MEMORY_SIZE: int = int(25e6)

# available bandwidth to slow memory (Stratix 10): 86.4GB/s
_FPGA_BANDWIDTH: int = int(86.4e9)

# we assume that as soon as the pipeline is saturated, we can produce one result per cycle
_CYCLES_PER_OUTPUT: int = 1
```

```
def do_estimate():
    """
    This function is meant to programmatically go through the current 'best-estimate' calculation of our model.
    :return: None
    """

    # estimate for critical path length (#cycles): mean of the tree stencils: fastwaves, diffusion, advection
    critical_paths: List[List[int]] = list(list())
    # instantiate fastwaves and add critical path
    critical_paths.append(KernelChainGraph(path="input/fastwaves.json",
                                           plot_graph=False,
                                           verbose=False).compute_critical_path_dim())
    # instantiate diffusion and add critical path
    critical_paths.append(KernelChainGraph(path="input/diffusion.json",
                                           plot_graph=False,
                                           verbose=False).compute_critical_path_dim())
    # instantiate advection and add critical path
    critical_paths.append(KernelChainGraph(path="input/advection.json",
                                           plot_graph=False,
                                           verbose=False).compute_critical_path_dim())
    # calculate mean of the three
    critical_path_sum = functools.reduce(lambda x, y: helper.list_add_cwise(x, y), critical_paths, [0] * 3)
    mean: List[int] = list(map(lambda x: x / len(critical_paths), critical_path_sum))
    _MEAN_CRITICAL_PATH_KERNEL: List[int] = mean
    print("Mean critical path length of the three stencils is: {}".format(mean))
    # print header
    print("################################################################")
    print("COSMO dynamical core buffer size estimate report:\n")
    print("################################################################")
    # instantiate the dummy-dycore (modified to fixed latency per kernel of 4*latency(addition) to get full analysis
    chain = KernelChainGraph("input/dycore_upper_half.json")
    # assumption: since we implemented ~1/2 of the dycore in the dummy input file, we assume the critical path is 2x
    # longer
    _DYCORE_CRITICAL_PATH_LENGTH = 2 * chain.compute_critical_path()
    # compute total critical path
    critical_path_dim = [x * _DYCORE_CRITICAL_PATH_LENGTH for x in _MEAN_CRITICAL_PATH_KERNEL]
    print("total critical path length (dimensionless) = _MEAN_CRITICAL_PATH_KERNEL * _DYCORE_CRITICAL_PATH_LENGTH = "
          "{} * {} = {}".format(_MEAN_CRITICAL_PATH_KERNEL, _DYCORE_CRITICAL_PATH_LENGTH, critical_path_dim))
    critical_path_cyc = helper.dim_to_abs_val(critical_path_dim, _DIMENSIONS)
    print("total critical path length (cycles) = {} cycles\n".format(critical_path_cyc))
    # compute maximum possible communication volume
    run_time_cyc = critical_path_cyc + reduce(operator.mul, _DIMENSIONS)
    print("total run time (cycles) = latency + dimX*dimY*dimZ = {}".format(run_time_cyc))
    run_time_sec = run_time_cyc / _FPGA_CLOCK_FREQUENCY
    print("total run time (seconds) = total run time (cycles) / _FPGA_CLOCK_FREQUENCY = {}".format(run_time_sec))
    comm_vol = _FPGA_BANDWIDTH * run_time_sec
    print("maximum available communication volume (to slow memory) = _FPGA_BANDWIDTH * total run time (seconds) = {}"
          .format(comm_vol))
```

```
if __name__ == "__main__":
    do_estimate()
```