sdfg_generator.py

```python
import argparse
import collections
import functools
import itertools
import operator
import os
import re

import dace
import dace.codegen.targets.fpga
import numpy as np
from dace.graph.edges import InterstateEdge
from dace.memlet import Memlet
from dace.sdfg import SDFG
from dace.types import ScheduleType, StorageType, Language

import helper
from kernel_chain_graph import Kernel, Input, Output, KernelChainGraph

ITERATORS = ["i", "j", "k"]


def make_iterators(dimensions, halo_sizes=None):
    def add_halo(i):
        if i == len(dimensions) - 1 and halo_sizes is not None:
            return " + " + str(-halo_sizes[0] + halo_sizes[1])
        else:
            return ""
    if len(dimensions) > 3:
        return collections.OrderedDict(
            [("i" + str(i), "0:" + str(d) + add_halo(i))
             for i, d in enumerate(dimensions)])
    else:
        return collections.OrderedDict(
            [(ITERATORS[i], "0:" + str(d) + add_halo(i))
             for i, d in enumerate(dimensions)])


def make_stream_name(src_name, dst_name):
    return src_name + "_to_" + dst_name


def relative_to_buffer_index(buffer_size, index):
    return buffer_size - 1 - abs(index)


def generate_sdfg(name, chain):

    sdfg = SDFG(name)
```

```python
pre_state = sdfg.add_state("initialize")
state = sdfg.add_state("compute")
post_state = sdfg.add_state("finalize")

sdfg.add_edge(pre_state, state, InterstateEdge())
sdfg.add_edge(state, post_state, InterstateEdge())

def add_pipe(sdfg, edge):

    stream_name = make_stream_name(edge[0].name, edge[1].name)

    sdfg.add_stream(
        stream_name,
        edge[0].data_type,
        buffer_size=edge[2]["channel"]["delay_buffer"].maxsize,
        storage=StorageType.FPGA_Local,
        transient=True)

def add_input(node):

    # Host-side array, which will be an input argument
    sdfg.add_array(node.name + "_host", chain.dimensions, node.data_type)

    # Device-side copy
    sdfg.add_array(
        node.name,
        chain.dimensions,
        node.data_type,
        storage=StorageType.FPGA_Global,
        transient=True)
    access_node = state.add_read(node.name)

    iterators = make_iterators(chain.dimensions)

    # Copy data to the FPGA
    copy_host = pre_state.add_read(node.name + "_host")
    copy_fpga = pre_state.add_write(node.name)
    pre_state.add_memlet_path(
        copy_host,
        copy_fpga,
        memlet=Memlet.simple(
            copy_fpga,
            ", ".join(iterators.values()),
            num_accesses=functools.reduce(operator.mul, chain.dimensions,
                                          1)))

    entry, exit = state.add_map(
        "read_" + node.name, iterators, schedule=ScheduleType.FPGA_Device)
    # Sort to get deterministic output
    outputs = sorted([e[1].name for e in chain.graph.out_edges(node)])

    out_memlets = ["_" + o for o in outputs]

    tasklet_code = "\n".join(
        ["{} = memory".format(o) for o in out_memlets])

    tasklet = state.add_tasklet("read_" + node.name, {"memory"},
                                out_memlets, tasklet_code)

    state.add_memlet_path(
        access_node,
        entry,
        tasklet,
        dst_conn="memory",
        memlet=Memlet.simple(
            node.name, ", ".join(iterators.keys()), num_accesses=1))

    # Add memlets to all FIFOs connecting to compute units
    for out_name, out_memlet in zip(outputs, out_memlets):
        stream_name = make_stream_name(node.name, out_name)
        write_node = state.add_write(stream_name)
        state.add_memlet_path(
            tasklet,
            exit,
            write_node,
            src_conn=out_memlet,
            memlet=Memlet.simple(stream_name, "0", num_accesses=1))

def add_output(node):

    # Host-side array, which will be an output argument
    sdfg.add_array(node.name + "_host", chain.dimensions, node.data_type)

    # Device-side copy
    sdfg.add_array(
        node.name,
        chain.dimensions,
        node.data_type,
        storage=StorageType.FPGA_Global,
        transient=True)
    write_node = state.add_write(node.name)

    iterators = make_iterators(chain.dimensions)

    # Copy data to the FPGA
    copy_fpga = post_state.add_read(node.name)
    copy_host = post_state.add_write(node.name + "_host")
    post_state.add_memlet_path(
        copy_fpga,
        copy_host,
        memlet=Memlet.simple(
            copy_host,
            ", ".join(iterators.values()),
            num_accesses=functools.reduce(operator.mul, chain.dimensions,
                                          1)))

    entry, exit = state.add_map(
        "write_" + node.name, iterators, schedule=ScheduleType.FPGA_Device)

    src = chain.graph.in_edges(node)
    if len(src) > 1:
        raise RuntimeError("Only one writer per output supported")
    src = next(iter(src))[0]

    in_memlet = "_" + src.name

    tasklet_code = "memory = " + in_memlet

    tasklet = state.add_tasklet("write_" + node.name, {in_memlet},
                                {"memory"}, tasklet_code)

    stream_name = make_stream_name(src.name, node.name)
    read_node = state.add_read(stream_name)

    state.add_memlet_path(
        read_node,
        entry,
        tasklet,
        dst_conn=in_memlet,
        memlet=Memlet.simple(stream_name, "0", num_accesses=1))

    state.add_memlet_path(
        tasklet,
        exit,
        write_node,
        src_conn="memory",
        memlet=Memlet.simple(
            node.name, ", ".join(iterators.keys()), num_accesses=1))

def add_kernel(node):

    # Extract fields to read from memory and fields to buffer
    memory_accesses = []
    buffer_sizes = collections.OrderedDict()
    buffer_accesses = collections.OrderedDict()
```

```python
for field, relative in node.graph.accesses.items():
    # Deduplicate, as we can have multiple accesses to the same index
    abs_indices = helper.unique(
        [helper.dim_to_abs_val(i, node.dimensions) for i in relative] +
        ([0]
         if node.boundary_conditions[field]["type"] == "copy" else []))
    max_access = max(abs_indices)
    min_access = min(abs_indices)
    buffer_size = max_access - min_access + 1
    memory_accesses.append(field)
    buffer_sizes[field] = buffer_size
    # (indices relative to center, buffer indices, buffer center index)
    buffer_accesses[field] = (relative,
                              [i - min_access for i in abs_indices],
                              -min_access)

# Create a initialization phase corresponding to the highest distance
# to the center
init_sizes = [
    buffer_sizes[key] - 1 - val[2]
    for key, val in buffer_accesses.items()
]
init_size_max = np.max(init_sizes)

iterators = make_iterators(chain.dimensions)

# Manually add pipeline entry and exit nodes
pipeline_range = dace.properties.SubsetProperty.from_string(', '.join(
    iterators.values()))
pipeline = dace.codegen.targets.fpga.Pipeline(
    "read_" + node.name,
    list(iterators.keys()),
    pipeline_range,
    ScheduleType.FPGA_Device,
    False,
    init_size=init_size_max,
    init_overlap=False,
    drain_size=init_size_max,
    drain_overlap=True)
entry = dace.codegen.targets.fpga.PipelineEntry(pipeline)
exit = dace.codegen.targets.fpga.PipelineExit(pipeline)
state.add_nodes_from([entry, exit])

# Sort to get deterministic output
outputs = sorted(e[1].name for e in chain.graph.out_edges(node))

# Add nested SDFG to do 1) shift buffers 2) read from input 3) compute
nested_sdfg = SDFG(node.name, parent=sdfg)
nested_sdfg_tasklet = state.add_nested_sdfg(
    nested_sdfg,
    sdfg, ([name + "_in" for name in memory_accesses] +
           [name + "_buffer_in" for name, _ in buffer_sizes.items()]),
    ([name + "_out" for name in outputs] +
     [name + "_buffer_out" for name, _ in buffer_sizes.items()]),
    name=node.name)

# Shift state, which shifts all buffers by one
shift_state = nested_sdfg.add_state(node.name + "_shift")

# Update state, which reads new values from memory
update_state = nested_sdfg.add_state(node.name + "_update")

# TODO: data type per field
dtype = node.data_type

#########################################################################
# Implement boundary conditions
#########################################################################

boundary_code = ""
# Loop over each field
for (field, (accesses, accesses_buffer,
             center)) in buffer_accesses.items():
    # Loop over each access to this field
    for indices, offset_buffer in zip(accesses, accesses_buffer):
        # Loop over each index of this access
        cond = []
        for i, offset in enumerate(indices):
            if offset < 0:
                cond.append(ITERATORS[i] + " < " + str(-offset))
            elif offset > 0:
                cond.append(ITERATORS[i] + " >= " +
                            str(node.dimensions[i] - offset))
        if len(cond) == 0:
            boundary_code += "{} {}_{} = _{}_{};\n".format(
                dtype.ctype, field, offset_buffer, field,
                offset_buffer)
        else:
            if node.boundary_conditions[field]["type"] == "copy":
                boundary_val = "_{}_{}".format(field, center)
            elif node.boundary_conditions[field]["type"] == "constant":
                boundary_val = node.boundary_conditions[field]["value"]
            boundary_code += (
                "{} {}_{} = ({}) ? ({}) : (_{}_{});\n".format(
                    dtype.ctype, field, offset_buffer,
                    " || ".join(cond), boundary_val, field,
                    offset_buffer))
```

```
########################################################################
# Only write if we’re in bounds
########################################################################

write_code = (
    ("if (!{}) {{\n".format("".join(pipeline.init_condition()))
     if init_size_max > 0 else "") + ("\n".join([
         "write_channel_intel({}_inner_out, res);".format(output)
         for output in outputs
     ])) + ("\n}" if init_size_max > 0 else "\n"))

########################################################################
# Tasklet code generation
########################################################################

code = boundary_code + "\n" + "\n".join([
    dtype.ctype + " " + expr.strip() + ";"
    for expr in node.generate_relative_access_kernel_string(
        relative_to_center=False).split(";")
])
# Replace array accesses with memlet names
pattern = re.compile("(([a-zA-Z_][a-zA-Z0-9_]*)\[([^\]]+)\])")
for full_str, field, index in re.findall(pattern, code):
    buffer_index = relative_to_buffer_index(buffer_sizes[field],
                                            int(index))
    if int(index) > 0:
        raise ValueError("Received positive index " + full_str + ".")
    code = code.replace(full_str, "{}_{}".format(field, buffer_index))
code += "\n" + write_code

########################################################################
# Create DaCe compute state
########################################################################

# Compute state, which reads from input channels, performs the compute,
# and writes to the output channel(s)
compute_state = nested_sdfg.add_state(node.name + "_compute")
compute_inputs = list(
    itertools.chain.from_iterable(
        [["_{}_{}".format(name, offset) for offset in offsets]
         for name, (_, offsets, _) in buffer_accesses.items()]))
compute_tasklet = compute_state.add_tasklet(
    node.name + "_compute",
    compute_inputs, [name + "_inner_out" for name in outputs],
    code,
    language=Language.CPP)

# Connect the three nested states
nested_sdfg.add_edge(shift_state, update_state, InterstateEdge())
```

```python
nested_sdfg.add_edge(update_state, compute_state, InterstateEdge())

for in_name, (field_name, size), init_size in zip(
        memory_accesses, buffer_sizes.items(), init_sizes):

    stream_name = make_stream_name(in_name, node.name)

    # Outer memory read
    read_node_outer = state.add_read(stream_name)
    state.add_memlet_path(
        read_node_outer,
        entry,
        nested_sdfg_tasklet,
        dst_conn=in_name + "_in",
        memlet=Memlet.simple(stream_name, "0", num_accesses=-1))

    # Create inner memory pipe
    stream_name_inner = field_name + "_in"
    stream_inner = sdfg.arrays[stream_name].clone()
    stream_inner.transient = False
    nested_sdfg.add_datadesc(stream_name_inner, stream_inner)

    buffer_name_outer = "{}_{}_buffer".format(node.name, field_name)
    buffer_name_inner_read = "{}_buffer_in".format(field_name)
    buffer_name_inner_write = "{}_buffer_out".format(field_name)

    # Create buffer transient in outer SDFG
    # TODO: use data type from edge
    if size > 1:
        desc_outer = sdfg.add_array(
            buffer_name_outer, [size],
            dtype,
            storage=StorageType.FPGA_Local,
            transient=True)
    else:
        desc_outer = sdfg.add_scalar(
            buffer_name_outer,
            dtype,
            storage=StorageType.FPGA_Registers,
            transient=True)

    # Create read and write nodes
    read_node_outer = state.add_read(buffer_name_outer)
    write_node_outer = state.add_write(buffer_name_outer)

    # Outer memory read
    state.add_memlet_path(
        read_node_outer,
        entry,
        nested_sdfg_tasklet,
        dst_conn=buffer_name_inner_read,
        memlet=Memlet.simple(
            buffer_name_outer, "0:{}".format(size), num_accesses=-1))

    # Outer memory write
    state.add_memlet_path(
        nested_sdfg_tasklet,
        exit,
        write_node_outer,
        src_conn=buffer_name_inner_write,
        memlet=Memlet.simple(
            write_node_outer.data,
            "0:{}".format(size),
            num_accesses=-1))

    # Inner copy
    desc_inner_read = desc_outer.clone()
    desc_inner_read.transient = False
    desc_inner_read.name = buffer_name_inner_read
    desc_inner_write = desc_inner_read.clone()
    desc_inner_write.name = buffer_name_inner_write
    nested_sdfg.add_datadesc(buffer_name_inner_read, desc_inner_read)
    nested_sdfg.add_datadesc(buffer_name_inner_write, desc_inner_write)

    # Make shift state if necessary
    if size > 1:
        shift_read = shift_state.add_read(buffer_name_inner_read)
        shift_write = shift_state.add_write(buffer_name_inner_write)
        shift_entry, shift_exit = shift_state.add_map(
            "shift_{}".format(field_name),
            {"i_shift": "0:{} - 1".format(size)},
            schedule=ScheduleType.FPGA_Device,
            unroll=True)
        shift_tasklet = shift_state.add_tasklet(
            "shift_{}".format(field_name),
            {"{}_shift_in".format(field_name)},
            {"{}_shift_out".format(field_name)},
            "{field}_shift_out = {field}_shift_in".format(
                field=field_name))
        shift_state.add_memlet_path(
            shift_read,
            shift_entry,
            shift_tasklet,
            dst_conn=field_name + "_shift_in",
            memlet=Memlet.simple(
                shift_read.data, "i_shift + 1", num_accesses=1))
        shift_state.add_memlet_path(
            shift_tasklet,
                shift_exit,
                shift_write,
                src_conn=field_name + "_shift_out",
                memlet=Memlet.simple(
                    shift_write.data, "i_shift", num_accesses=1))

        # Begin reading according to this field's own buffer size, which is
        # translated to an index by subtracting it from the maximum buffer
        # size
        begin_reading = (init_size_max - init_size)
        end_reading = (functools.reduce(operator.mul, chain.dimensions, 1)
                       + init_size_max - init_size)

        update_read = update_state.add_read(stream_name_inner)
        update_write = update_state.add_write(buffer_name_inner_write)
        update_tasklet = update_state.add_tasklet(
            "read_wavefront", {"wavefront_in"}, {"buffer_out"},
            "if ({it} >= {begin} && {it} < {end}) {{\n"
            "buffer_out = read_channel_intel(wavefront_in);\n"
            "}} else {{\n"
            "buffer_out = -1000;\n"
            "}}\n".format(it=pipeline.iterator_str(), begin=begin_reading,
            end=end_reading),
            language=Language.CPP)
        update_state.add_memlet_path(
            update_read,
            update_tasklet,
            memlet=Memlet.simple(update_read.data, "0", num_accesses=-1),
            dst_conn="wavefront_in")
        update_state.add_memlet_path(
            update_tasklet,
            update_write,
            memlet=Memlet.simple(
                update_write.data,
                "{} - 1".format(size) if size > 1 else "0",
                num_accesses=1),
            src_conn="buffer_out")

        # Make compute state
        compute_read = compute_state.add_read(buffer_name_inner_read)
        for offset in buffer_accesses[field_name][1]:
            compute_state.add_memlet_path(
                compute_read,
                compute_tasklet,
                dst_conn="_" + field_name + "_" + str(offset),
                memlet=Memlet.simple(
                    compute_read.data, str(offset), num_accesses=1))

    for out_name in outputs:
            # Outer write
            stream_name_outer = make_stream_name(node.name, out_name)
            write_node_outer = state.add_write(stream_name_outer)
            state.add_memlet_path(
                nested_sdfg_tasklet,
                exit,
                write_node_outer,
                src_conn=out_name + "_out",
                memlet=Memlet.simple(
                    write_node_outer.data, "0", num_accesses=-1))

            # Create inner stream
            stream_name_inner = out_name + "_out"
            stream_inner = sdfg.arrays[stream_name_outer].clone()
            stream_inner.transient = False
            nested_sdfg.add_datadesc(stream_name_inner, stream_inner)

            # Inner write
            write_node_inner = compute_state.add_write(stream_name_inner)
            compute_state.add_memlet_path(
                compute_tasklet,
                write_node_inner,
                src_conn=out_name + "_inner_out",
                memlet=Memlet.simple(
                    write_node_inner.data, "0", num_accesses=-1))

    for link in chain.graph.edges(data=True):
        add_pipe(sdfg, link)

    for node in chain.graph.nodes():
        if isinstance(node, Input):
            add_input(node)
        elif isinstance(node, Output):
            add_output(node)
        elif isinstance(node, Kernel):
            add_kernel(node)
        else:
            raise RuntimeError("Unexpected node type: {}".format(
                node.node_type))

    return sdfg


if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument("stencil_file")
    parser.add_argument("--plot-graph", dest="plot-graph", action="store_true")
parser.add_argument("--plot-sdfg", dest="plot-sdfg", action="store_true")

args = parser.parse_args()

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

chain = KernelChainGraph(args.stencil_file)

if getattr(args, "plot-graph"):
    chain.plot_graph(name + ".pdf")

sdfg = generate_sdfg(name, chain)

if getattr(args, "plot-sdfg"):
    chain.plot_graph(name + ".pdf")

sdfg.compile()
```