**run_program.py**

```
#!/usr/bin/env python3
import argparse
import itertools
import os
import re
import subprocess as sp

import dace
import numpy as np

import helper
from kernel_chain_graph import KernelChainGraph
from sdfg_generator import generate_sdfg
from simulator import Simulator

parser = argparse.ArgumentParser()
parser.add_argument("stencil_file")
parser.add_argument("mode", choices=["emulation", "hardware"])
parser.add_argument("-log-level", choices=["0", "1", "2", "3"], default=3)
parser.add_argument("-plot", action="store_true")
parser.add_argument("-simulation", action="store_true")
parser.add_argument("-skip-execution", dest="skip_execution", action="store_true")
parser.add_argument("--print-result", dest="print_result", action="store_true")
args = parser.parse_args()

# Load program file
program_description = helper.parse_json(args.stencil_file)
name = os.path.basename(args.stencil_file)
name = re.match("[^\.]+", name).group(0)

# Create SDFG
print("Create KernelChainGraph")
chain = KernelChainGraph(path=args.stencil_file,
                         plot_graph=args.plot,
                         log_level=int(args.log_level))

# do simulation
if args.simulation:
    print("Run simulation.")
    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()
    simulation_result = sim.get_result()
```

```
print("Generate sdfg")
sdfg = generate_sdfg(name, chain)

# Configure and compile SDFG
print("Configure sdfg")
dace.config.Config.set("compiler", "fpga_vendor", value="intel_fpga")
# dace.config.Config.set("compiler", "use_cache", value=True)
dace.config.Config.set("optimizer", "interface", value="")
if args.mode == "emulation":
    dace.config.Config.set("compiler", "intel_fpga", "mode", value="emulator")
else:
    dace.config.Config.set("compiler", "intel_fpga", "mode", value="hardware")
print("Compile sdfg")
program = sdfg.compile()

# Load data from disk
print("Loading input arrays...")
input_arrays = helper.load_input_arrays(program_description)

# Initialize output arrays
print("Initializing output arrays...")
output_arrays = {
    arr_name: np.zeros(
        program_description["dimensions"],
        dtype=program_description["program"][arr_name]["data_type"].type)
    for arr_name in program_description["outputs"]
}

# Compile program (if running emulation)
build_folder = os.path.join(".dacecache", name, "build")
if args.mode == "emulation":
    print("Compiling emulation kernel...")
    sp.run(
        ["make", "intelfpga_compile_" + name + "_emulator"],
        cwd=build_folder,
        check=True)
elif args.mode == "hardware":
    if not os.path.exists(os.path.join(build_folder, name + "_hardware.aocx")):
        raise FileNotFoundError("Hardware kernel has not been built.")

if args.skip_execution:
    exit()

# Run program
dace_args = {
    key + "_host": val
    for key, val in itertools.chain(input_arrays.items(),
                                    output_arrays.items())
```

```
}
print("Executing DaCe program...")
program(**dace_args)
print("Finished running program.")

if args.print_result:
    for key, val in output_arrays.items():
        print(key + ":", val)

# Write results to file
output_folder = os.path.join("results", name)
os.makedirs(output_folder, exist_ok=True)
helper.save_output_arrays(output_arrays, output_folder)
print("Results saved to " + output_folder)

# Compare simulation result to fpga result
if args.simulation:
    print("Comparing the results.")
    all_match = True
    for outp in output_arrays:
        print("fpga:")
        print(np.ravel(output_arrays[outp]))
        print("simulation")
        print(np.ravel(simulation_result[outp]))
        if not helper.arrays_are_equal(np.ravel(output_arrays[outp]), np.ravel(simulation_result[outp])):
            all_match = False
    if all_match:
        print("Output matched!")
    else:
        print("Output did not match!")
```