full_flow_xilinx

April 30, 2024

1 Xilinx Full Flow Demo (Xilinx)

This is an interactive notebook version fo the `full_flow_xilinx.py` demo.

1.1 HLSFactory Imports

Below, we import all the necessary items from the HLSFactory library.

```
[ ]: from hlsfactory.datasets_builtin import (
         datasets_builder,
     )
     from hlsfactory.flow_vitis import (
         VitisHLSImplFlow,
         VitisHLSImplReportFlow,
         VitisHLSSynthFlow,
     )
     from hlsfactory.framework import (
         count_total_designs_in_dataset_collection,
     )
     from hlsfactory.opt_dsl_frontend import OptDSLFrontend
     from hlsfactory.utils import (
         DirSource,
         ToolPathsSource,
         get_tool_paths,
         get_work_dir,
         remove_and_make_new_dir_if_exists,
     )
```

1.2 Run Setup

Below we setup the work directory (where everything will run and be stored on disk).

The call to `get_work_dir` uses an `.env` file looks for a variable/key called `HLSFACTORY_WORK_DIR`.

Then we create a sub-directory in this work directory for this specific demo.

```
[ ]: WORK_DIR_TOP = get_work_dir(dir_source=DirSource.ENVFILE)
     WORK_DIR = WORK_DIR_TOP / "demo_full_flow_xilinx"
     remove_and_make_new_dir_if_exists(WORK_DIR)
```

Next we setup the number of parallelism to use for our flows.

```
[ ]: N_JOBS = 32
     CPU_AFFINITY = list(range(N_JOBS))
```

We also setup the paths to the Xilinx tools.

The call to `get_tool_paths` uses an `.env` file and for variables/keys called `HLSFACTORY_VITIS_HLS_PATH` and `HLSFACTORY_VIVADO_PATH`.

```
[ ]: VITIS_HLS_PATH, VIVADO_PATH = get_tool_paths(tool_paths_source=ToolPathsSource.
      ↪ENVFILE)
     VIVADO_BIN = VIVADO_PATH / "bin" / "vivado"
     VITIS_HLS_BIN = VITIS_HLS_PATH / "bin" / "vitis_hls"
```

1.3 Dataset Setup

Below we setup some datasets supported by the Xilinx flows.

We use the built-in `polybench`, `machsuite`, and `chstone` datasets.

The call to `datasets_builder` copies the built-in datasets, the work directory, renames them according to the user provided labels, and returns a `DatasetCollection` (dict) object of `DesignDatasets` objects, one for each dataset.

```
[ ]: datasets = datasets_builder(
         WORK_DIR,
         [
             "polybench",
             "machsuite",
             "chstone",
         ],
         dataset_labels=[
             "polybench_xilinx",
             "machsuite_xilinx",
             "chstone_xilinx",
         ],
     )
```

The commented code below show an alternative way to setup the the datasets the same way.

```
[ ]: """
     dataset_polybench_xilinx = dataset_polybench_builder("polybench_xilinx",␣
      ↪WORK_DIR)
     dataset_machsuite_xilinx = dataset_machsuite_builder("machsuite_xilinx",␣
      ↪WORK_DIR)
     dataset_chstone_xilinx = dataset_chstone_builder("chstone_xilinx", WORK_DIR)

     datasets: DesignDatasetCollection = {
         "polybench_xilinx": dataset_polybench_xilinx,
         "machsuite_xilinx": dataset_machsuite_xilinx,
         "chstone_xilinx": dataset_chstone_xilinx,
     }
     """
```

We can count the total number of designs in the dataset collection.

[ ]:
```
total_count = count_total_designs_in_dataset_collection(datasets)
print(f"Total Designs: {total_count}")
```

1.4 Frontend for Design Space Elaboration

The selected datasets can be utilized within the Optimization DSL (Opt. DSL) Frontend to generate a variety of designs, each applying different combinations of optimization directives.

To achieve this, create an instance of `OptDSLFrontend` and invoke the `execute_*` method to operate this frontend. This process will generate a new dataset collection comprising various design datasets that include the sampled designs.

Below, we specify the number of randomly sampled designs with varying optimization directive combinations. Additionally, a seed can be set for this sampling to ensure reproducibility.

[ ]:
```
N_RANDOM_SAMPLES = 12
RANDOM_SAMPLE_SEED = 64
```

We then create the `OptDSLFrontend` object and call the `execute_multiple_design_datasets_fine_grained_parallel` run the frontend for all the designs for all the datasets in parallel.

[ ]:
```
opt_dsl_frontend = OptDSLFrontend(
    WORK_DIR,
    random_sample=True,
    random_sample_num=N_RANDOM_SAMPLES,
    random_sample_seed=RANDOM_SAMPLE_SEED,
    log_execution_time=True,
)
datasets_post_frontend = (
    opt_dsl_frontend.execute_multiple_design_datasets_fine_grained_parallel(
        datasets,
        True,
        lambda x: f"{x}__post_frontend",
        n_jobs=N_JOBS,
        cpu_affinity=CPU_AFFINITY,
    )
)
```

We can count the number of designs in the dataset collection post-frontend.

[ ]:
```
total_count_post_frontend = count_total_designs_in_dataset_collection(
    datasets_post_frontend,
)
print(f"Total Designs post-frontend: {total_count_post_frontend}")
```

1.5 HLS Synthesis and Implementation Flows

Now, we take all the designs, execute HLS synthesis on each, and subsequently run the normal FPGA synthesis and implementation for each design.

As before, create instances of `VitisHLSSynthFlow` and `VitisHLSImplFlow`, and then invoke their `execute_*` methods to operate the flows.

It is important to note that the `VitisHLSSynthFlow` manages both the HLS synthesis and the automatic extraction of data from the synthesis reports within the same flow, eliminating the need for an additional step.

For the HLS synth. and impl. flow steps, the user can set the timeout for each step.

If a flow step takes longer than the timeout, then that design flow is terminated and that design is not collected to output.

[ ]:
```
TIMEOUT_HLS_SYNTH = 60.0 * 8   # 8 minutes
TIMEOUT_HLS_IMPL = 60.0 * 30   # 30 minutes
```

We can also create a worst case estimation of the total build time based on the number of designs and the timeout for each step.

[ ]:
```
total_time_estimation = (
    total_count_post_frontend * (TIMEOUT_HLS_SYNTH + TIMEOUT_HLS_IMPL) / N_JOBS
)
print(
    f"Estimated worst-case build time:\n{total_time_estimation}␣
  ↪seconds\n{total_time_estimation / 60} minutes\n{total_time_estimation /␣
  ↪3600} hours",
)
```

Below we setup and run both flows.

[ ]:
```
toolflow_vitis_hls_synth = VitisHLSSynthFlow(
    vitis_hls_bin=str(VITIS_HLS_BIN),
    env_var_xilinx_hls=str(VITIS_HLS_PATH),
    env_var_xilinx_vivado=str(VIVADO_PATH),
)
datasets_post_hls_synth = (
    toolflow_vitis_hls_synth.
  ↪execute_multiple_design_datasets_fine_grained_parallel(
        datasets_post_frontend,
        False,
        n_jobs=N_JOBS,
        cpu_affinity=CPU_AFFINITY,
        timeout=TIMEOUT_HLS_SYNTH,
    )
)
```

```
[ ]: toolflow_vitis_hls_implementation = VitisHLSImplFlow(
         vitis_hls_bin=str(VITIS_HLS_BIN),
         env_var_xilinx_hls=str(VITIS_HLS_PATH),
         env_var_xilinx_vivado=str(VIVADO_PATH),
     )
     datasets_post_hls_implementation = toolflow_vitis_hls_implementation.
      ↪execute_multiple_design_datasets_fine_grained_parallel(
         datasets_post_hls_synth,
         False,
         n_jobs=N_JOBS,
         cpu_affinity=CPU_AFFINITY,
         timeout=TIMEOUT_HLS_IMPL,
     )
```

1.6 Vivado Reporting Flow

We also run one last flow to extract data from the Vivado reports including power, timing, and resource utilization data.

Unlike the `VitisHLSSynthFlow`, the `VitisHLSImplFlow` does not automatically extract data from the reports. In the future, we might refactor the impl. flow to also run the report extraction or we refactor the HLS synth. flow to break out the HLS report extraction into a separate step; ideally we would do this to maintain API consistency.

```
[ ]: toolflow_vitis_hls_impl_report = VitisHLSImplReportFlow(
         vitis_hls_bin=str(VITIS_HLS_BIN),
         vivado_bin=str(VIVADO_BIN),
         env_var_xilinx_hls=str(VITIS_HLS_PATH),
         env_var_xilinx_vivado=str(VIVADO_PATH),
     )
     toolflow_vitis_hls_impl_report.
      ↪execute_multiple_design_datasets_fine_grained_parallel(
         datasets_post_hls_implementation,
         False,
         n_jobs=N_JOBS,
         cpu_affinity=CPU_AFFINITY,
     )
```

1.7 Demo Conclusion and Next Steps

All the output data from HLS synthesis and FPGA implementation is located in the `WORK_DIR`.

Our flows have extracted the relevant data into data files alongside the design sources and project files.

This data can be aggregated and further process and analyzed, also using HLSFactory, or by the user, for specific research or development needs.