custom_datasets

April 30, 2024

1 Custom Designs and Datasets Demo

This demo shows how to use the Design and DesignDataset classes as part of HLSFactory to load your own custom designs and datasets of HLS designs.

Once loaded these designs ca be run though the provided HLSFactory flows and data aggregation steps to create packaged data for your custom designs.

This demo only deals with **"concrete" designs**, meaning these designs don't require front end flows for preprocessing or enumeration, and are ready be passed into HLS synthesis and implementation tools. Another demo (*currently under development*) will show how to use **"abstract" designs** that can be used front end flows for more complex experiments or runs.

If you have a set of HLS designs you already can run through HLS tools, this demo will id show you how to use the in the HLSFactory framework.

1.1 Design and Design Dataset Overview

In HLSFactory a "Design" is a single HLS design and a "Design Dataset" is a collection of HLS designs.

We represent these as the `Design` and `DesignDataset` classes in HLSFactory.

We adapt the convention that a "Design" is simpl a directory/folder on disk and a "Design Dataset" is a directory/folder containing multiple "Design" directories/folders.

The main job of the `Design` and `DesignDataset` classes is to keep track of the deigns and design dataset directory paths as well as simple metadata like design name or dataset name.

Most importantly, these classes provide many helper functions to help build, modify, copy, and move designs and design datasets.

Therefore, a user can also use these classes to build their own HLS design datasets to run HLSFactory flows and data aggregation on a users's custom HL designs.

1.2 Imports and Setup

Below is some imports and setup for HLSFactory that you would do to setup some experiments or runs.

[ ]:
```
from hlsfactory.datasets_builtin import (
    datasets_builder,
)
```

```
from hlsfactory.flow_vitis import (
    VitisHLSImplFlow,
    VitisHLSImplReportFlow,
    VitisHLSSynthFlow,
)
from hlsfactory.framework import (
    Design,
    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,
)

from hlsfactory.framework import DesignDatasetCollection
```

[ ]:
```
# Looks for a variable/key in an an .env file called HLSFACTORY_WORK_DIR
WORK_DIR_TOP = get_work_dir(dir_source=DirSource.ENVFILE)
WORK_DIR = WORK_DIR_TOP / "demo_custom"
remove_and_make_new_dir_if_exists(WORK_DIR)

# Number of cores to run stuff in parallel
N_JOBS = 32
CPU_AFFINITY = list(range(N_JOBS))

# Looks for variables/keys in an an .env file 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 Current Directory Setup

For this demo we need to use the current directory to load a custom design nd custom design dataset stored locally to this Juypter notebook. Juypter notebooks do not have a `__file__` to derive the local directory so we need manually specify the current directory.

Important Note: For this to work correctly as is, you need to run the notebook with the "current working directory" being the root of this repository. If you are running this notebook from a different "current working directory" you will need to change the `current_directory` variable to the correct path or change your current working directory you launch the notebook. VSCode has a setting called "Notebook File Root" which you can set. If you open the repository root in VSCode, you can set this setting to "${workspaceFolder}" to make the current working directory the root

of the repository.

```
[ ]: from pathlib import Path

CURRENT_DIR = Path("./demos/demo_custom_datasets")
assert CURRENT_DIR.exists()
```

1.4 Loading a Single Custom Design and Building a Design Dataset

This section will show how to load a single design into a `Design` object. It also shows how to make an `DesignDataset` object and add the single design to it.

We first define the local path of the design we want to load (should be sibling to this notebook).

```
[ ]: custom_single_design_fp = CURRENT_DIR / "custom_design_concrete"
assert custom_single_design_fp.exists()
```

Then we create a `Design` object. We pass in a name and a path to the "Design" class to create a "Design" object.

```
[ ]: my_design = Design("my_design", custom_single_design_fp)
```

This creates the `Design` referencing the design source "in-place". For example, if we run a flow on this design, it might modify the source files and generate project files in the local design directory.

We do not want this to happen. Ideally we can keep our source files in one place clean and organized, and copy he design to a new working directory / run directory where we then run flows on the design.

The code below shows how to use on of the built-in functions the Design class provides to help with the task of moving, copying , and renaming deigns.

Here, we just copy the design to a new working directory using the `copy_to_new_parent_dir` function of the `Design` class.

```
[ ]: my_design = my_design.copy_to_new_parent_dir(WORK_DIR)
```

Note that there are several helper functions similar to this that we recommend you look at to make setting up and running custom experiments and runs easier. There also exists helper functions for `DesignDataset` classes in addition to `Design` classes.

Now we show how to create a `DesignDataset` object that will contain our single design.

We first create the DesignDataset object by providing it with a dataset name, a directory for the dataset, and an empty list of designs. We use the raw class initializer here to create the as opposed to one of the helper classmethods.

We then add the design to the dataset using the `add_design_copy` method of the `DesignDataset` class. This will copy the design directory into the dataset directory and then update all the relevant metadata data in each object.

```
[ ]: from hlsfactory.framework import DesignDataset
```

```
dataset_dir = WORK_DIR / "my_dataset_with_one_design"

my_dataset_with_one_design = DesignDataset(
    "my_dataset_with_one_design",
    dataset_dir,
    designs=[],
)

my_dataset_with_one_design.add_design_copy(my_design)
```

[ ]:
```
DesignDataset(name=my_dataset_with_one_design, dataset_dir=/usr/scratch/skaram7/
hlsfactory_work_dir/demo_custom/my_dataset_with_one_design, len(designs)=1)
```

2 Building a Design Dataset from a Directory of Designs

If we have a folder of individual design directories, we can also build a DesignDataset object with all the designs in that directory using the helper `from_dir` classmethod of the `DesignDataset` class.

Below we do just that with the local directory `./custom_dataset_folder` which contains a `./custom_dataset_folder/design_a` directory and a `./custom_dataset_folder/design_b` directory for two designs.

[ ]:
```
dataset_sources = CURRENT_DIR / "custom_dataset_folder"
assert dataset_sources.exists()

my_dataset_with_several_designs = DesignDataset.from_dir(
    "my_dataset_with_several_designs",
    dataset_sources,
).copy_dataset(WORK_DIR)
```

2.1 What is Required in HLSFactory Designs for Different Flows

Anyone can create an HLS design in infinitely different ways using different combinations of source files, headers, TCL scripts, build scripts, Python scripts, testing flows, and so on. Therefore, any design directory can contain an arbitrary organization of files run in an arbitrary way to actually run HLS synthesis and FPGA implementation for the design.

Therefore, as a framework, HLSFactory must impose some very loose requirements on the design directory structure to be able to run HLSFactory flows.

In practice, this means that each flow may impose different requirements on the design directory structure or contents in order to run that flow. These requirements should be checked by the code/logic of the flow implementation itself and the user must conform to that if using that flow. Therefore, we leave it up to the user to understand the requirements of the flow they are using and to make sure their design directory conforms to those requirements.

However, this is very vague and doesn't specify how this is implemented for different flows.

Below we will outline different requirements that different built-in flows have for the design directory structure and conte

2.1.1 Design Requirements for `VitisHLSSynthFlow`, `VitisHLSImplFlow`, `VitisHLSImplReportFlow`

The core Xilinx-based HLS flows all require "entrypoint" Tcl scripts as a main requirement for the design directory. These entrypoint Tcl scripts must be present in the design directory and are used to allow the user to define the necessary logic for running each flow. Additionally, these entry point scripts must perform certain tasks that the flows depend on for their output. If a user fails to provide these entrypoint scripts or the scripts don't perform the necessary tasks, the flow will fail. Certain basic checks are done by the flows as they are run to also provide more helpful error messages as early as possible.

The `VitisHLSSynthFlow` requires that a `dataset_hls.tcl` script file must be present at the top level of the design directory. This script also needs to create a single Vitis HLS solution and run `csynth_design` on that solution. We also adopt the convention that each design should only create one solution for simplicity. If different solutions are needed (e.g. different C-flags, defines, part numbers, clock speeds), the user should create different design directories with different parameters (either manually or programmatically) and run the flow on each design directory.

It is the user's responsibility to make sure this script is present and performs the necessary tasks to run HLS synthesis on the design. If the script is not present or does not create a solution and run `csynth_design` successfully for that solution, the flow will fail. If these requirements are met, the flow will run successfully.

The `VitisHLSImplFlow` requires that a `dataset_hls_ip_export.tcl` script file must be present at the top level of the design directory, as well as a solution created that has already been synthesized by Vitis HLS; most likely from the previous `VitisHLSSynthFlow`. This script needs to run the Vitis HLS "export_design" command with the `-flow impl` flag at some point to run FPGA synthesis and implementation on the design. If the script is not present or does not run `export_design` successfully, the flow will fail. If these requirements are met, the flow will run successfully. We also include a handy check if the user uses the `open_project` command with the `-reset` flag to avoid running a script that will delete the already synthesized solution.

The `VitisHLSImplReportFlow` does not require an entry point but does require the presence of a single Vivado project somewhere in the design directory, most likely from the previous `VitisHLSImplFlow`, that contains the design already fully implemented. The flow will run Vivado reporting for power, timing, and resource utilization of the design. If the project is not present or does not contain a fully implemented design, the flow will fail. If these requirements are met, the flow will run successfully.

2.2 Further Design Requirements Information

2.2.1 Examples of Built-In Design

You can use the built-in designs of good examples of how to setup the required entrypoints and scripts for the different flows. The built-in designs are located in the `hlsfactory/hls_dataset_sources` directory of the repository. We recommend looking at the `vitis_examples` designs for examples of how to setup the required entrypoints and scripts for the Xilinx-based HLS flows in HLSFactory.

2.2.2 Quick Adaptation of Existing Vitis HLS Design Scripts

If you already have a Tcl script, `my_hls_synth_script.tcl` that runs Vitis HLS synthesis and you want to adapt it for the `VitisHLSSynthFlow`, you can juts make a new script called `dataset_hls.tcl` and have a single line that sources your script: `source my_hls_synth_script.tcl`. The same applies for the VitisHLSImplFlow if you already have a script that runs Vitis HLS export IP with the `-flow impl` flag.

2.2.3 Other Flow Requirements

HLSFactory also has support for Intel HLS and Quartus flows. Documentation for these flows and their requirements is a work in progress and will be added in the future.

2.3 Running the Xilinx Flows on the Custom Design Dataset

Now that we understand how to setup a custom design dataset as well as what is required of the designs to run the Xilinx-based HLS flows, we can run the flows on the custom design dataset.

To make things easier we can also group design datasets into a `DesignDatasetCollection` which is nothing more than a type alias for `dict[str, DesignDataset]`.

[ ]:
```
datasets: DesignDatasetCollection = {
    "my_dataset_with_several_designs": my_dataset_with_several_designs,
}
```

Now we setup and run the flows. These would not complete and throw errors if we are missing any of the design requirements for the flows.

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

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,
        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,
)


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,
)
```