HLSFactory: A Framework Empowering High-Level Synthesis Datasets for Machine Learning and Beyond

Stefan Abi-Karam[1,2], Rishov Sarkar[1], Allison Seigler[3], Sean Lowe[4], Zhigang Wei[3], Hanqiu Chen[1], Nanditha Rao[5], Lizy John[3], Aman Arora[4], Cong Hao[1]

[1]Georgia Institute of Technology, [2]Georgia Tech Research Institute, [3]The University of Texas at Austin, [4]Arizona State University, [5]International Institute of Information Technology Bangalore

{stefanabikaram, rishov.sarkar, hanqiu.chen, callie.hao}@gatech.edu, {aseigler, zw5259, ljohn@ece.utexas.edu}@utexas.edu, {slowe8, aman.kbm}@asu.edu, {nanditha.rao}@iiitb.ac.in

Abstract

Machine learning (ML) techniques have been applied to high-level synthesis (HLS) flows for quality-of-result (QoR) prediction and design space exploration (DSE). Nevertheless, the scarcity of accessible high-quality HLS datasets and the complexity of building such datasets present great challenges to FPGA and ML researchers. Existing datasets either cover only a subset of previously published benchmarks, provide no way to enumerate optimization design spaces, are limited to a specific vendor, or have no reproducible and extensible software for dataset construction. Many works also lack user-friendly ways to add more designs to existing datasets, limiting wider adoption and sustainability of such datasets.

In response to these challenges, we introduce HLSFactory, a comprehensive framework designed to facilitate the curation and generation of high-quality HLS design datasets. HLSFactory has three main stages: 1) a design space expansion stage to elaborate single HLS designs into large design spaces using various optimization directives across multiple vendor tools, 2) a design synthesis stage to execute HLS and FPGA tool flows concurrently across designs, and 3) a data aggregation stage for extracting standardized data into packaged datasets for ML usage. This tripartite architecture not only ensures broad coverage of data points via design space expansion but also supports interoperability with tools from multiple vendors. Users can contribute to each stage easily by submitting their own HLS designs or synthesis results via provided user APIs. The framework is also flexible, allowing extensions at every step via user APIs with custom frontends, synthesis tools, and scripts.

To demonstrate the framework functionality, we include an initial set of built-in base designs from PolyBench, MachSuite, Rosetta, CHStone, Kastner et al.'s Parallel Programming for FPGAs, and curated kernels from existing open-source HLS designs. We report the statistical analyses and design space visualizations to demonstrate the completed end-to-end compilation flow, and to highlight the effectiveness of our design space expansion beyond the initial base dataset, which greatly contributes to dataset diversity and coverage.

In addition to its evident application in ML, we showcase the versatility and multi-functionality of our framework through seven case studies: I) Building an ML model for post-implementation QoR prediction; II) Using design space sampling in stage 1 to expand the design space covered from a small base set of HLS designs; III) Demonstrating the speedup from the fine-grained design parallelism backend; IV) Extending HLSFactory to target Intel's HLS flow across all stages; V) Adding and running new auxiliary designs using HLSFactory; VI) Integration of previously published HLS data in stage 3; VII) Using HLSFactory to perform HLS tool version regression benchmarking.

Code available at https://github.com/sharc-lab/HLSFactory.

ACM Reference Format:
Stefan Abi-Karam[1,2], Rishov Sarkar[1], Allison Seigler[3], Sean Lowe[4], Zhigang Wei[3], Hanqiu Chen[1],, Nanditha Rao[5], Lizy John[3], Aman Arora[4], Cong Hao[1]. 2024. HLSFactory: A Framework Empowering High-Level Synthesis Datasets for Machine Learning and Beyond. In *2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD '24), September 9–11, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3670474.3685961

MLCAD '24, September 9–11, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0699-8/24/09
https://doi.org/10.1145/3670474.3685961

1 Introduction

Machine learning (ML) techniques have been widely applied to different electronic design automation (EDA) flows including high-level synthesis (HLS) [9, 12, 17, 20–25, 30] for quality-of-result (QoR) prediction, optimization, and design space exploration (DSE). A key enabler to the success of such ML techniques is high-quality datasets, most of which are developed for individual studies e.g., [22, 25, 30]. Some recent works have contributed open-source datasets that can be used by other reseachers, e.g., [6, 14–16, 33, 38].

Despite the great benefits of these datasets, there are still fundamental **limitations** that hinder their wider adoption for ML applications and FPGA research. First, these datasets are usually small or homogeneous, containing only a subset of previously published HLS benchmarks [1, 18, 26, 38], and frequently consisting exclusively of designs that work with one HLS tool from a single vendor. For example, Spector [15] contains only 9 Intel HLS designs, HLSDataset [33] contains 34 AMD/Xilinx HLS designs, and Rosetta [38] contains 6 AMD/Xilinx HLS designs. Second, because of these separately developed HLS datasets, the designs and intermediate/final tool outputs, which serve as important ML model features, are often reported organized in non-standard *ad hoc* ways. Some datasets contain only source code [1, 18, 26], some datasets contain only resource usage and end-to-end throughput [38] but no clock frequency or power numbers, while some contain only post-implementation results [6]. HLSyn [6] is a dataset for HLS designs targeted towards predicting design quality of FPGAs. It consists of a wider range of programs and compiler directives, enabling performance optimization of designs. However, existing datasets require huge manual effort and deep domain-specific knowledge for ML practitioners if they need a complete, unified, and larger dataset, where they must execute all related HLS tools on their own to re-collect and organize the needed information. Third, it is challenging for external users who want to extend the existing datasets by contributing their own designs, primarily caused by ad-hoc data formats and missing details when building these datasets (e.g., tool version, target FPGA device, clock frequency, implementation flow settings). **The fundamental limitation is, however, not the lack of another complete and rich HLS dataset, but rather the lack of a flexible and extensible framework to enable continuous contributions to a standardized and sustainable dataset.**

Therefore, in this work, we introduce **HLSFactory**, the first framework that takes a principled approach to HLS dataset generation, collection, expansion, and integration, aiming to facilitate a continuous and community-wide effort to contribute to the richest HLS dataset, which will keep expanding easily. HLSFactory boasts the following features:

- **Complete and easily extensible with user inputs at multiple stages.** HLSFactory has an end-to-end compilation flow including three main stages: design space expansion stage to elaborate single HLS designs at the source-code level into large design spaces; design synthesis stage to execute HLS and FPGA tools; and data aggregation stage for extracting standardized data organization. HLSFactory uses a modular design that allows users to plug in their own designs and tool flows to the dataset with minimal effort at arbitrary stages.
- **Diverse and comprehensive.** The initially included dataset covers a wide variety of HLS designs, containing both simple designs synthesized with AMD/Xilinx and Intel tool flows and complex designs using Xilinx-specific features. In addition, HLSFactory has a novel design space expansion and sampling approach, allowing the generation of many design points from a single HLS design, improving overall design space coverage. Further, HLSFactory has comprehensive data metrics from synthesis to implementation, e.g., HLS synthesis reported resource and latency, and post-implementation resource, timing, power, etc.
- **Reproducible and user-friendly.** HLSFactory features push-button ease-of-use to run the entire end-to-end dataset generation workflow,

Table 1. A comparison of HLSFactory with the existing work. ●: feature supported; ○: feature unsupported; ◐: feature partially supported.

Contributions	DB4HLS	HLSyn	HLSDataset	HLSFactory
Benchmark — Polybench	○	●	●	●
Benchmark — MachSuite	●	●	●	●
Benchmark — Rosetta	○	○	●	●
Benchmark — CHStone	○	○	●	●
Collection — PP4FPGA	○	○	○	●
Collection — Accelerators (§5.5)	○	○	○	●
Post-HLS Latency	●	●	○	●
Post-HLS Resources	●	●	●	●
Post-HLS Artifacts	○	○	○	●
Post-Impl. Data	○	○	●	●
HLS Optimization DSL	●	○	●	●
Fine-Grained Parallel Builds	◐	○	○	●
Xilinx HLS Support	●	●	●	●
Intel HLS Support	○	○	○	●
User Extendable to Other Tools	○	○	○	●
Programmable API	○	○	○	●
Open Source	●	●	●	●

allowing anyone to replicate our generated results, and to easily contribute to the framework and the dataset. Specifically, our framework makes it extremely easy for researchers in the FPGA community to contribute data for various FPGA devices.

- **ML-ready and multi-purpose** Beyond simply being used for ML training, as demonstrated with a post-implementation QoR prediction ML model (case study in §5.1), HLSFactory is useful for any task where a large, diverse set of HLS runs is needed, like HLS tool version regression testing (case study in §5.7).
- **High performance and open-source.** HLSFactory maximizes parallelism for fast dataset generation of large numbers of designs. HLSFactory is open-source and available on GitHub, including both the end-to-end framework and a large set of sample designs.

In Sec. 2, we first provide background on the prior works in existing HLS benchmarks and datasets. Sec. 3 introduces our HLSFactory framework detailing the three stages. Sec. 4 dives into the implementation of HLSFactory, including how it is configured and extended, and our fine-grained parallelism technique to speed up dataset generation. We then perform several case studies in Sec. 5 that demonstrate the multi-purpose of the proposed framework.

2 Related Work

HLS community has multiple standard benchmarks for assessing HLS tools including PolyBench [1], CHStone [18], and MachSuite [26], which in total provide around 67 benchmark designs and are far from sufficient for ML training. Rosetta [38], Dai [12], MLSBench [16], DB4HLS [14], HLSDataset [33], and Spector [15] are all recently proposed HLS datasets, where the former four use AMD/Xilinx tools and the last uses Intel tools. MLSBench provides a sampling from different combinations of directives (pragmas) on top of CHStone and MachSuite. DB4HLS provides exhaustive design exploration on 39 designs from MachSuite with a domain-specific language (DSL) for DSE and parallelized synthesis runs. HLSDataset aims to cover all four commonly used benchmarks (PolyBench, CHStone, MachSuite, Rosetta) with a DSL for specifying the design space to sample from. They also illustrate two ML-based case studies for post-implementation resource and power prediction. HLSyn [6] uses control data flow graphs (CDFGs) of compiled HLS kernels for QoR prediction using graph neural network approaches; their designs are sampled from PolyBench and MachSuite. The features of selected prior works and HLSFactory are shown in Table 1.

While existing HLS datasets serve as a solid foundation for empowering ML in HLS, they are inherently limited. First, each dataset covers only a subset of commonly used HLS benchmarks, employing ad-hoc data organization, synthesis tools, configurations, and reported metrics, lacking standardization. This fragmentation makes it exceedingly difficult for ML practitioners to effectively utilize all available datasets for training without significant efforts in data reorganization and tool re-execution. Consequently, the quality of ML models is compromised, impeding the advancement of ML in HLS. Second, the lack of standardized data organization and metric reporting poses challenges to dataset extensibility and long-term sustainability, hindering broader user contributions to HLS datasets.

Therefore, rather than introducing yet another HLS dataset, the ML for HLS community urgently requires a standard, extensible, and user-friendly **framework**. Such a framework would streamline the collection, generation, elaboration, synthesis, and organization of HLS designs and data from diverse sources and community users. This would facilitate the long-term maintenance and expansion of HLS datasets. The pressing need for such a solution is the driving force behind our proposed HLSFactory.

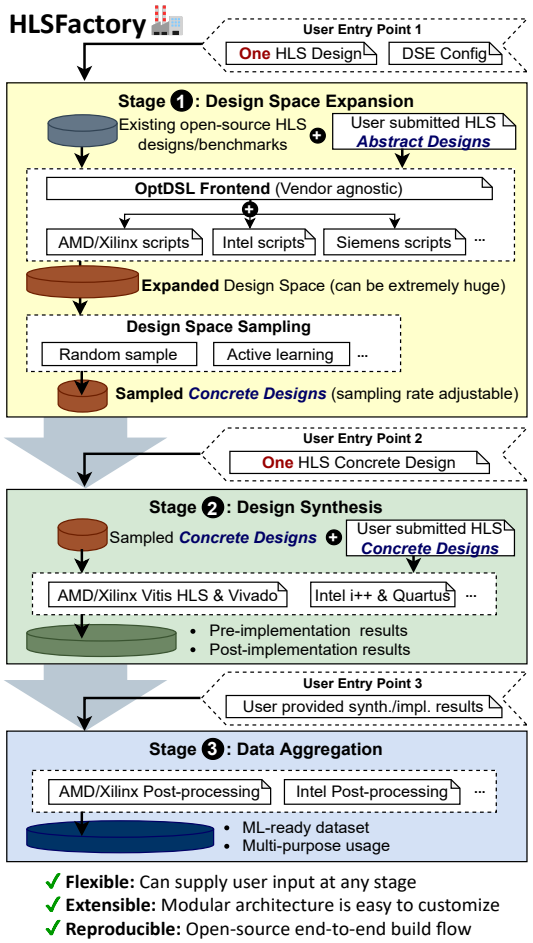

Figure 1. A complete overview of the HLSFactory framework with three stages and three entry points where users can contribute their own designs.

3 HLSFactory Framework

3.1 HLSFactory Overview

As depicted in Fig. 1, HLSFactory is composed of three **stages** in its end-to-end synthesis and data extraction flow; before each stage, there is an **entry point** where users can submit designs and data.

Stage ❶ is **design space expansion**, aiming at expanding a single HLS design into multiple designs by enumerating different combinations of optimization directives (pragmas), which can significantly increase the number of data points for ML applications. In this stage, users can submit one or more HLS designs with possible design space configurations and HLSFactory will extrapolate and expand the complete design space via a frontend. Note that the presented design space *expansion* stage is explicitly different than traditional design space *exploration* (commonly abbreviated as DSE). Design space expansion is not optimization guided (as detailed in §3.2.3), so suboptimal designs are included, broadening design space coverage needed to building robust, accurate ML models. This frontend features multi-vendor support and allows for random sampling of generated designs to reduce the number of designs to be synthesized by HLS and implementation tools (e.g., Vivado), if needed for large design spaces, to shorten the execution time.

We will showcase this usage in Section 5.4.

Stage ❷ is **design synthesis** stage, where vendor-specific HLS and implementation tools are invoked to synthesize HLS designs into RTL code and then placed and routed. In this stage, users can submit their HLS designs to be directly synthesized without extrapolating. We will showcase this usage in Section 5.5.

Stage ❸ is **data aggregation**, where statistics and artifacts are collected from the implemented designs and compiled into a tool-agnostic format for use by downstream tasks such as ML training and benchmarking. In

```
loop_opt,3,2
0,lp2,pipeline,unroll,[1 2 4 8]
1,lp3,pipeline,unroll,[1 2 4 8]
2,lp3,,unroll,[1 2 4 8]
set_directive_unroll -factor [factor] k2mm/[name]
set_directive_pipeline k2mm/[name]
```

Figure 2. A snippet demonstrating the OptDSL syntax.

this stage, users can submit their synthesized post-implementation results or datasets to be merged. We will showcase this usage in Section 5.6.

3.2 Stage 1: Design Space Expansion and Sampling

This stage aims at expanding a single HLS design into multiple by enumerating combinations of optimization directives (either inline or in a separate file), such as loop unroll factors, array partitioning schemes, and whether to pipeline loops. Such expansion is critical for ML usage because of two reasons. First, the original HLS designs and benchmarks are far from sufficient for ML training, and obtaining additional HLS designs is challenging. Second, a key application of ML for HLS is to help designers choose the best optimization directives for their HLS designs by predicting post-HLS-synthesis and post-implementation metrics from HLS source code and directives (e.g., [6, 33, 35, 36]). Therefore, an ML-ready HLS dataset must provide wide coverage of how different choices of optimization directives can impact a design. Note that the design space expansion is expected to be across *multiple vendors, tools, and devices*.

On the other hand, the expanded design space can be huge, and synthesizing and implementing each design may be prohibitively time-consuming. Therefore, design space sampling is needed.

We define the concept of a *frontend* pass, which *lowers* an HLS *abstract design* to a certain number of *concrete designs*. An HLS abstract design is not directly synthesizable but contains parameterized directives that require preprocessing. An HLS concrete design is a copy of the abstract HLS design and is augmented with one possible combination of optimization directives from the design space.

3.2.1 Vendor-agnostic OptDSL Frontend. For design space expansion, all possible combinations of optimization directives for a certain HLS design must be explicitly specified. We propose a frontend using a domain-specific language (DSL), named OptDSL, to specify the design space using a DSE configuration file. Fig. 2 shows an example of the OptDSL syntax, which specifies the choices for how to pipeline or unroll two loops `lp2` and `lp3`.

OptDSL is vendor-agnostic but based on a modified version of a Vitis HLS Tcl script, minimizing the learning curve for designers already accustomed to writing scripts for Vitis HLS. The main feature of OptDSL is the bracket notation that parameterizes an optimization directive with multiple choices. The overall design space is the Cartesian product of the choices for each parameterized directive.

3.2.2 Vendor-specific Concrete Design Generation. While abstract designs can be vendor-agnostic, concrete designs are vendor-specific. I.e., different vendor tools have different HLS syntax and directive formats; therefore, during the lowering process, the frontend needs vendor-specific logic to target different tool flows, as depicted in Fig. 1 stage 1. HLSFactory currently provides support for AMD/Xilinx and Intel flows, while other vendors can be easily supported.

The OptDSL file is provided within the abstract design as a file named `opt_template.tcl`. To lower the abstract design for AMD/Xilinx tools, we generate `opt.tcl`, a version of `opt_template.tcl` with bracketed parameters replaced with different concrete values for each design point. Once these bracketed parameters are substituted, the OptDSL script becomes a valid Tcl script that can be used directly with Vitis HLS.

To support other vendors, the frontend can parse the OptDSL file and identify the specific optimization directives used within it together with their parametrizations. If the provided OptDSL is not sufficient to describe a desired DSE, HLSFactory provides the necessary infrastructure to allow users to specify their own entirely custom frontend as Python code, as long as it conforms to the specified API interface (to be discussed in Sec. 4.1). For instance, a new frontend pass can easily be introduced to parameterize constants in the HLS source code itself: simply copy the existing OptDSL frontend and modify the templating logic and syntax to work with files other than `opt_template.tcl`.

3.2.3 Design Space Sampling. The design space created by the parameterized optimization directives may be extremely large for even a single design, growing exponentially with the addition of each directive. Therefore, it is almost impossible to enumerate every possible design point in the specified design space and execute synthesis and implementation.

HLSFactory natively supports random sampling of design points from the Cartesian product of all combinations of optimization directives. Users can specify the number of sampled design points, trading off design space coverage for dataset build time and storage.

Unlike design space exploration, HLSFactory's enumeration and random sampling approaches are not guided or optimization-driven. The focus is solely on collecting a wide range of designs, including suboptimal designs, which are important for building ML datasets and training ML models that can interpolate to as many unseen designs during evaluation and deployment, not just optimal designs.

In the future, HLSFactory can be extended to support user-customizable heuristics for selecting design points, utilizing expert knowledge to determine which combinations of optimizations are more useful to sample from and which combinations may result in invalid or redundant designs. For example, the sampling stage can be combined with active learning to determine meaningful design points to be synthesized.

3.3 Stage 2: Design Synthesis

The second stage of HLSFactory synthesizes and implements each concrete HLS design, a process we collectively refer to as the design synthesis. This stage also has an entry point for user input—vendor-specific concrete designs can be provided directly at this point without going through design space expansion. This is useful for easy integration of third-party HLS designs where parametrization of the design space may be difficult or unnecessary.

Design synthesis is broken down into two steps: (1) `HLSSynth`, where an HLS design is synthesized to RTL code, and (2) `HLSImpl`, where the resulting RTL code is implemented, resulting in a fully placed-and-routed design. For AMD/Xilinx designs, Vitis HLS is used for `HLSSynth` and Vivado for `HLSImpl`. However, any vendor tool can easily be integrated into the HLSFactory framework, for example, Yosys [34] or Intel HLS (to be demonstrated in Sec. 5.4), by providing Python code for the desired `ToolFlow` subclasses.

3.4 Stage 3: Data Extraction and Aggregation

Once all the frontends and tool flows have been executed on a pool of designs, relevant design data must be extracted and aggregated into structured formats. HLSFactory provides `DataAggregator` classes to package HLS synthesis data (estimated latency, resource usage), post-implementation data (timing, resource, and power data), tool execution metadata (version, runtime), and build artifacts (LLVM IR, IP blocks) into shareable datasets.

Furthermore, as in stage 2, users may want to provide input directly at this stage, e.g., when integrating pre-generated data from prior works, where the build process is not reproducible and thus an earlier entry point cannot be used. Therefore, HLSFactory provides an entry point to the data aggregation stage. This entry point can accept fully synthesized and implemented designs, from which HLSFactory's built-in data aggregators can extract the relevant data, or pre-generated metrics in whatever form is available, which can be used with a custom `DataAggregator` subclass to adapt such metrics into HLSFactory's standard output format.

4 Implementation and Usage

4.1 Vendor Agnostic User API

HLSFactory is implemented as a Python library and provides a simple user API that allows the framework configuration to be expressed easily as a short Python script (while still allowing for full Python programming if complex configuration is desired).

Table 2. The HLSFactory User API.

API Functions	Description
`class Design` `class Dataset`	Single HLS design Multiple HLS designs

`class Flow(ABC)`
`Flow.execute(design)`
`Flow.execute_datasets_parallel(design)`
	Abstract class for arbitrary design flow
	Execute a flow on one design
	Execute a flow on many designs

`class Frontend(Flow)` `class OptDSLFrontend(Frontend)`	Abstract class for frontend design expansion Opt DSL frontend for Xilinx HLS designs

`class ToolFlow(Flow)`
`class VitisHLSSynthFlow(ToolFlow)`
`class VitisHLSImplFlow(ToolFlow)`
`class VitisHLSImplReportFlow(ToolFlow)`
	Abstract class for EDA tool
	Run Vitis HLS synthesis
	Run Vivado implementation (via Vitis HLS)
	Run Vivado reporting

```
datasets: DesignDatasetCollection = {
    "polybench_xilinx": dataset_polybench_builder(WORK_DIR),
    "machsuite_xilinx": dataset_machsuite_builder(WORK_DIR),
    "chstone_xilinx": dataset_chstone_builder(WORK_DIR),
}

opt_dsl = OptDSLFrontend(WORK_DIR, random_sample=True,
                         random_sample_num=N_RANDOM_SAMPLES)
hls_synth = VitisHLSSynthFlow()
hls_impl = VitisHLSImplFlow()
hls_impl_report = VitisHLSImplReportFlow()

datasets_post_frontend = opt_dsl.execute_datasets_parallel(
    datasets, n_jobs=N_JOBS)
datasets_post_synth = hls_synth.execute_datasets_parallel(
    datasets_post_frontend, n_jobs=N_JOBS)
datasets_post_hls_impl = hls_impl.execute_datasets_parallel(
    datasets_post_synth, n_jobs=N_JOBS)
hls_impl_report.execute_datasets_parallel(
    datasets_post_hls_impl, n_jobs=N_JOBS)
```

Figure 3. Example usage of the HLSFactory framework.

An example is shown in Fig. 3. The source HLS designs are located and copied to the desired work directory, and the `OptDSLFrontend` is invoked to sample 10 random design points from each design. The `VitisHLSSynthFlow` and `VitisHLSImplFlow` are then be invoked to synthesize and implement each design point, followed by data aggregation using the `VitisHLSImplReportFlow` to gather data from each implemented design in a standardized format. A full list of the available APIs is available in Table 2.

The API also includes abstract base classes (ABCs) that users can subclass to implement their own frontends and tool flows for HLSFactory, for instance, to support another vendor's HLS tools. HLSFactory abstracts away the complexities of integrating custom user subclasses into the overall dataset generation process, including the use of fine-grained parallelism (to be discussed in Sec. 4.3).

4.2 Directory Structure

Fig. 4 depicts a simple example of the directory structure accepted as input and produced as output of the HLSFactory workflow. As described throughout Sec. 3, we first sample the design space for each source abstract design and then run tool flows and data aggregation on the sampled concrete designs. The figure presents the directory structure for the inputs to this process: an abstract design specified in terms of HLS kernel code, an `opt_template.tcl` file to be used by the OptDSL frontend (described in Sec. 3.2.1), and auxiliary scripts for the AMD/Xilinx tool flows.

During dataset generation, each abstract design is enumerated into multiple concrete designs, shown in the figure under the newly generated directory `source_designs_xilinx__post_frontend`. Each concrete design is identified by the concatenation of the name of the original abstract design and a unique hash determined by the combination of optimization directives chosen for that design. This unique combination of optimization directives is generated as the concrete design's `opt.tcl` file.

Tool flows and data aggregation run directly within these concrete design directories. After HLS projects are created, synthesized, and implemented (within the `hls_prj` directory, as depicted), the data aggregation stage collects information from these projects into standardized JSON-formatted files. These JSON files are stored alongside the HLS project directory within each concrete design, making it clear exactly which combination of optimization directives were used to generate the data.

4.3 Parallel Build Backend

To build datasets with hundreds and thousands of data points, an efficient backend is needed to dispatch and execute multiple frontend and tool flows in parallel. In the case of HLS, the bottleneck of constructing such datasets is the runtime of the vendor tools themselves. The runtime for synthesizing an HLS design can range from minutes to hours. We may also want to run trial FPGA implementation flows, which can take hours.

To address these needs, every frontend and tool flow component is automatically augmented in a fine-grained parallel build backend based on multiprocessing. Since all frontend and tool flows are based on the abstract base class, we can easily provide this facility to the user. We take advantage of Python's `multiprocessing`. We also provide the option to pin each task to its own dedicated CPU core. This approach appears to be a good default to distribute design build workloads on many-core systems.

We also provide a way for users to pool parallelism across dataset collections rather than a single dataset. Users are able to describe

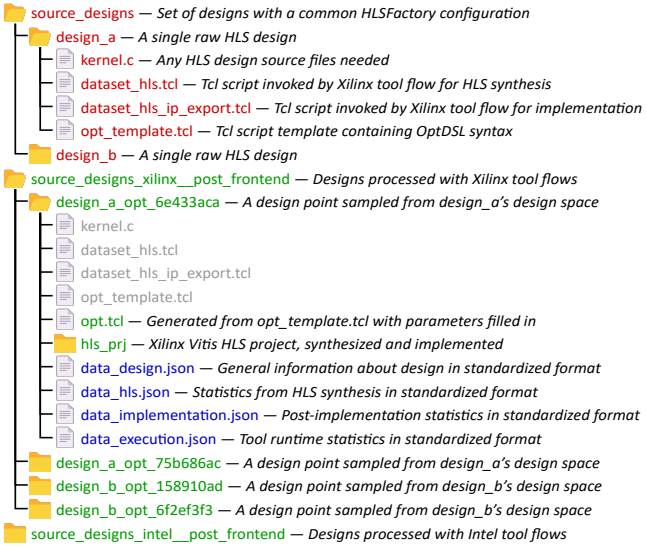

Figure 4. The directory structure that HLSFactory uses. Red are input files; green are the intermediate design points; blue are output files.

a collection of datasets, each with their own set of designs. Instead of dispatching each dataset's build workloads in its own parallel pool (i.e., naive parallelism), we aggregate all designs into a single parallel pool (i.e., fine-grained parallelism). This feature is automatic for every frontend and tool flow and transparent to the end user.

5 Evaluations

We evaluate our work through a series of seven case studies which demonstrate HLSFactory's multifunctionality and ease of use.

5.1 Case Study 1: ML Prediction of Post-Implementation QoR

HLS vendor tools provide resource usage estimates (e.g., #LUTs, #FFs, #DSPs, #BRAMs) and timing information (e.g., II violations, clock speed) for designs based on scheduling and binding results. However, HLS-estimated results often deviate significantly from post-implementation resource usage and may not correlate well with critical timing metrics (e.g., worst negative slack and worst hold slack). Previous works, such as S. Dai et al. [13], address this issue by using ML-based models to predict post-implementation quality-of-results (QoR) metrics based on HLS-reported metrics.

We demonstrate that HLSFactory can replicate the approach used by S. Dai et al. [13] to build ML models for post-implementation QoR predictions targeting Vitis HLS and Vivado. We use HLSFactory built-in Polybench, MachSuite, and CHStone design datasets which provide $n = 29$ base designs; using the the OptDSL frontend, design space expansion is performed resulting in $n = 257$ final designs. HLSFactory's APIs are also used run tool synthesis and implementation as well as bundle the HLS post-implementation data into a tabular dataset. A

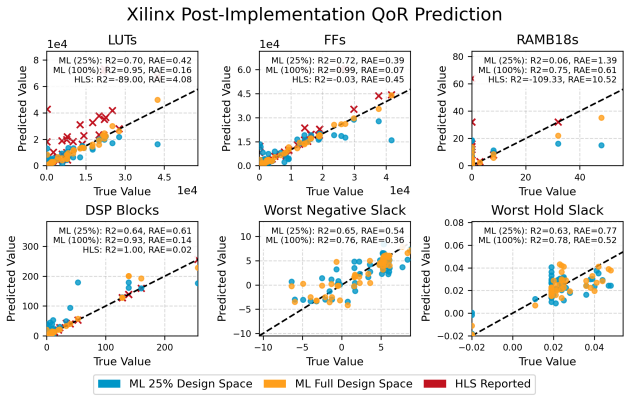

Figure 5. True-vs-predicted plots for the HLS-based ML QoR model. Test values are shown for models trained on the complete and partial subset of the training design space. "RAE": Relative Absolute Error ($|\hat{y}-y|/|y-\bar{y}|$), "R2": Coefficient of Determination

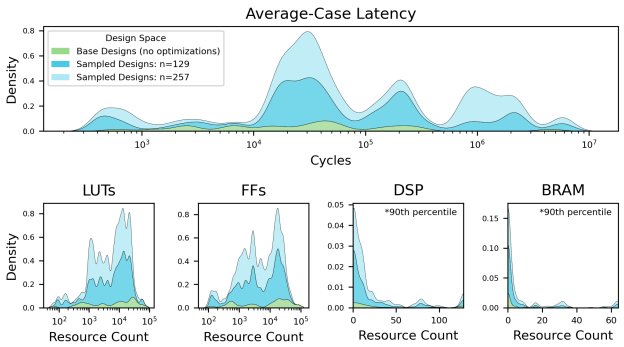

Figure 6. Effect of design sampling to cover more design space. Sampled designs cover a wider range of metrics than base designs with no optimizations. Latency is HLS estimated; resources are post-implementation. Note that these are *stacked* density plots to show the effect of cumulative design sampling.

histogram-based gradient boosting regression model is then trained to predict post-implementation reported resources and timing metrics using HLS-reported resources, latency, clock speed, and arithmetic/logic operation counts as model inputs. We train our model on an 80%/20% train-test split, as well as a 25% subset of the training data to demonstrate the utility of design space expansion in improving ML model performance.

Our results, shown in Fig. 5, indicate that the R^2 value and mean relative error are better for the larger training set achieved through design space expansion. We highlight that generating more data points using HLSFactory's design space expansion will result in higher prediction accuracy, even when randomly sampling from the entire design space and including suboptimal, i.e. "bad", designs (in terms of QoR metrics). For most resource prediction targets, our ML model also has a lower relative error than the HLS-reported values, showing improvement over the HLS tool itself. These results highlight the utility of HLSFactory applied to ML for EDA and the importance of design space expansion, even with a smaller sample size, for robust ML dataset construction and model training.

5.2 Case Study 2: Design Space Coverage

We evaluate how the use of design space expansion in HLSFactory quantitatively and qualitatively improves the overall design space of generated datasets in terms of latency (HLS-reported) and resource usage (post-implementation). In the context of ML, improved design space coverage for these metrics is important for robust model training on downstream tasks, such as ML-based QoR prediction. Thus we perform a case study comparing metrics of the base designs in Polybench, MachSuite, and CHStone ($n=29$) with the designs sampled from them ($n=257$); this is the same dataset used in §5.1.

We start with a quantitative evaluation. Fig. 6 illustrates the cumulative distributions of these metrics as a stacked histogram representing only base designs ($n=29$), half the sampled designs ($n=129$), and all the sampled designs ($n=257$). We highlight that the sampled designs cover a wider range of average-case latency, LUT usage, and FF usage, with denser coverage as n increases. In the case of DSP and BRAM usage, most base designs use none of these resources while sampled designs do.

We then illustrate the qualitative coverage of the design space in Fig. 7. This space is the 2-D embedding space of HLS-reported and post-implementation metrics generated using a PacMAP [32] dimensional reduction. Each of the base designs is depicted as large emphasized points within this embedding space; sampled designs from the same base design (top panel) or the same benchmark (bottom panel) have matching colors. The convex hulls around same-colored points show the portion of the embedding space covered by design space expansion from each base design or benchmark. This clearly shows that sampling from the expanded design space results in non-overlapping coverage that otherwise would not appear in the final dataset.

5.3 Case Study 3: Speedup of Fine-Grained Design Parallelism

We evaluate our fine-grained parallelism strategy described in Sec. 4.3 using a case study synthesizing designs sampled from Polybench, MachSuite, and CHStone using Vitis HLS across 32 CPU cores.

Results are shown in Fig. 8, showing that fine-grained parallelism achieves more than 20% speed up compared with the naive parallelism

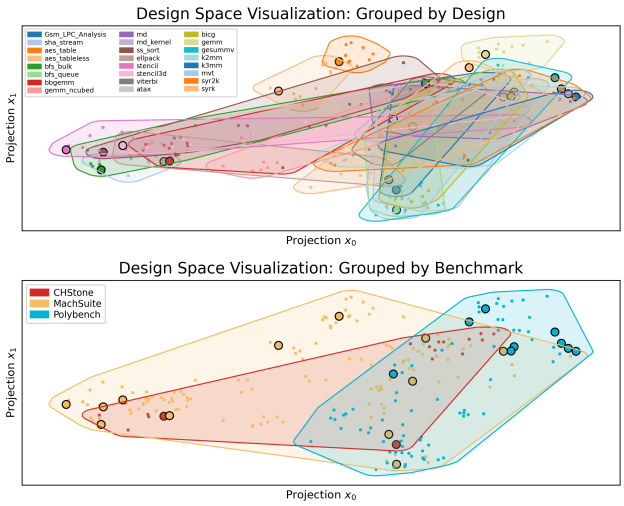

Figure 7. Embedding of sampled designs across selected benchmarks. Base designs without optimizations are emphasized. Design points and locations are the same between both panels; they are only colored and grouped differently.

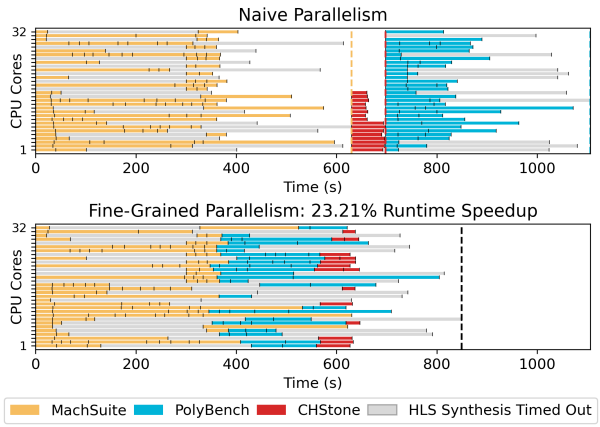

Figure 8. Parallel execution of Vitis HLS synthesis. Top panel shows core utilization over time with naive parallelism across datasets; bottom panel shows our fine-grained design parallelism across datasets.

approach. Such fine-grained parallelism is especially beneficial given the user-specified timeout threshold (annotated as gray bars).

5.4 Case Study 4: Targeting Different Vendors

To demonstrate the extensibility of the first stage of HLSFactory, we show how to add support for Intel's i++ HLS flow.

As described in Sec. 3.2.2, HLSFactory includes an OptDSL parser that recognizes Vitis HLS optimization directives in `opt_template.tcl`, such as the `set_directive_unroll` and `set_directive_array_partition` commands. We can therefore build our Intel-lowering frontend on top of this functionality.

Because i++ does not support specifying optimization directives in a separate file, our frontend instead transforms the HLS source code directly to add i++-compatible versions of each directive parsed from the `opt_template.tcl` file.

While our frontend can often generate exact equivalents for the specified directives, in some cases, i++ has no exact equivalent for a particular directive used by Vitis HLS, such as `array_partition` directives. In these cases, we substitute similar directives—in this case, a combination of Intel directives `hls_numbanks` and `hls_bankwidth` that achieve a similar memory partitioning result.

Since HLSFactory is agnostic to the specific directives being used and does not correlate specific AMD/Xilinx concrete designs with specific Intel concrete designs, directives need not match one-to-one. There is no impact on correctness; substituting similar directives still improves the diversity of the dataset.

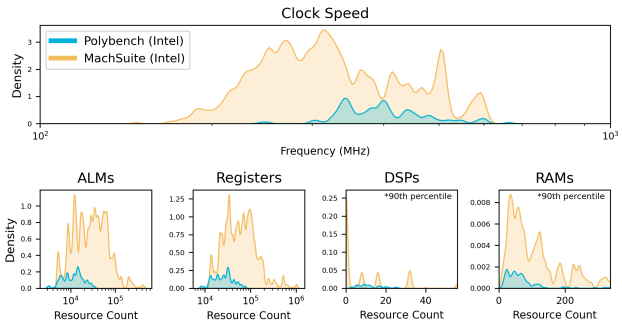

Figure 9. Distribution of post-implementation metrics for PolyBench and MachSuite designs ($n = 1340$) using Intel's HLS flow.

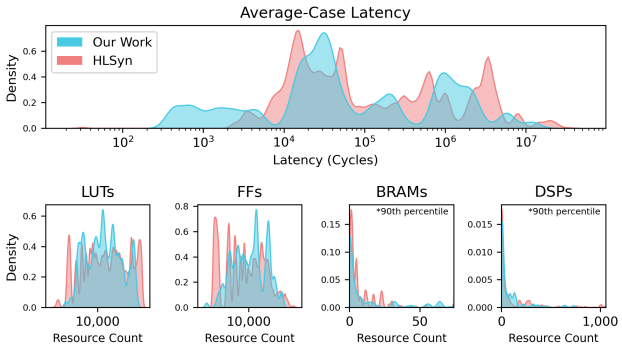

Figure 10. Distribution of HLS-estimated metrics from selected HLSFactory benchmarks (PolyBench, CHStone, and MachSuite; $n = 167$) vs. HLSyn ($n = 3371$).

In total, our end-to-end Intel flow extends the HLSFactory user APIs in Table 2 with three Intel equivalents: `OptDSLFrontendIntel(Frontend)` as described above, `IntelHLSSynthFlow(ToolFlow)` to invoke i++ for HLS, and `IntelQuartusImplFlow(ToolFlow)` to invoke Quartus for implementation. We run this flow on designs sampled from PolyBench and MachSuite and plot the resulting metrics in Fig. 9. Intel's HLS tool does not report overall latency estimates, but it optimizes each kernel's throughput by maximizing clock speed, which we use as a proxy for performance.

5.5 Case Study 5: Adding Auxiliary Design Collections

Third-party researchers may have existing, synthesizable, vendor-specific HLS designs to integrate into HLSFactory, but they may not want or need to create an OptDSL specification for them. For instance, the authors of LightningSim [28] collect 33 synthesizable open-source designs for AMD/Xilinx Vitis HLS to evaluate their simulation tool, including designs from AMD/Xilinx sample code repositories [4, 5], algorithm implementations from Kastner *et al.*'s *Parallel Programming for FPGAs* [19], and graph neural network implementations from FlowGNN [27]. These designs are all provided in a standard format, each having a Tcl script `setup.tcl` to set up a Vitis HLS project for synthesis.

Using the entry point at the design synthesis stage, one graduate student was able to integrate all of these designs into HLSFactory in less than one hour. To match the input directory structure in Fig. 4, we only needed to copy `setup.tcl` to `dataset_hls.tcl` with `csynth_design` appended (HLSFactory's `VitisHLSSynthFlow` expects it to setup the project *and* run synthesis) and add a four-line script `dataset_hls_ip_export.tcl` to invoke implementation from Vitis HLS. Since we used the entry point after design space expansion, these were concrete designs, not abstract designs, so no `opt_template.tcl` was required.

Many other works [7, 8, 29, 37] were also easily integrated with HLSFactory in a similar fashion; the code is available online.

5.6 Case Study 6: Integrating ReleasedData from Other Works

We may still want to incorporate previously published data have published to build a more comprehensive HLS dataset. HLSFactory's data aggregation step provides an entry point to incorporate external data sources into our dataset with ease.

We illustrate how HLSFactory can integrate pre-generated data from prior works—in this case, HLSyn [6]. HLSyn provides both the source code (with places to template optimization directives) of their selected kernels, as well as associated metrics for HLS-reported resource usage and latency for sampled designs. We write a `DataAggregator` subclass to integrate this data into HLSFactory.

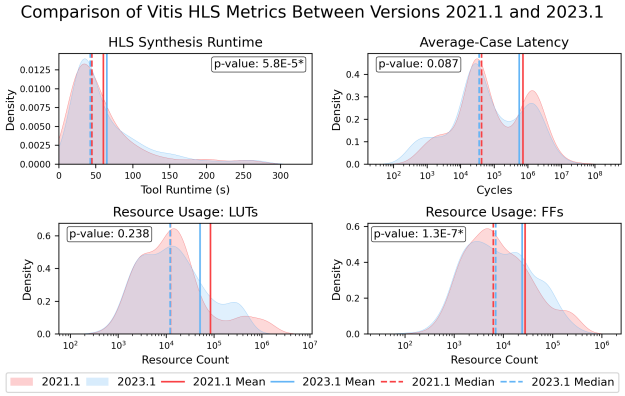

Figure 11. Distribution of HLS tool metrics from two versions of Vitis HLS.

The results are illustrated in Fig. 10, showing the distributions of reported HLS metrics sourced from the listed valid designs of HLSyn and a small sampled subset of designs from our base PolyBench, CHStone, and Machsuite datasets.

The HLSyn flow is built on top of AutoDSE [2, 31] and the Merlin compiler [3, 10], both of which are open-source software tools aimed at optimized design space exploration (DSE) and source-to-source translation. These tools suggest future work to integrate AutoDSE and the Merlin compiler as custom flows in HLSFactory, allowing designs to be built from the design space specifications defined in AutoDSE and synthesized with the Merlin compiler.

5.7 Case Study 7: Regression Benchmarking HLS Synthesis Tools

New versions of HLS vendor tools are periodically released and improve both the tool performance (e.g., faster synthesis) and the QoR of synthesized designs (e.g., less resource usage). However, quantifying such improvements across different tool versions is difficult without a way to benchmark a wide range of designs, similar to the regression testing used in traditional software development.

We demonstrate that HLSFactory streamlines regression testing on HLS tools. We compare Vitis HLS versions 2021.1 and 2023.1 using designs sampled from Polybench, Machsuite, and CHStone (with 16 samples per base design). We collect paired samples by synthesizing the same design with both tool versions.

This experiment was set up in a fully self-contained Python script and HLSFactory enabled this initial study to be completed by one graduate student in three hours.

The results are shown in Fig. 11. We show distributions of the tool runtime, HLS-estimated latency, LUT usage, and FF usage across tool versions. We also report the p-value for a paired two-tailed Wilcoxon signed-rank test [11, p. 350] and indicate cases with a p-value less than $\alpha = 0.05$ with an asterisk, indicating a statistically significant difference. Note that for certain metrics, the mean and median shift in opposite directions between tool versions.

6 Conclusion

HLSFactory brings a much-needed principled approach to generating datasets of HLS designs. Our case studies show a small sample of what can be done when a flexible, reproducible way to generate data from HLS designs is available. We demonstrate that there is substantial untapped potential for future research into how ML can be applied to HLS.

We also consider directions for future extensions of HLSFactory. Our framework currently has no support for collecting post-simulation metrics like vector-based power analysis or simulated latency. Introducing simulation to HLSFactory, particularly for designs where only a high-level C testbench is available rather than an RTL testbench, is a valuable direction for future work.

We hope that, through open-source, this work invites the research community to collaborate and contribute more designs and tool flows and accelerate ML research for EDA applications.

Acknowledgements

This research was supported in part by National Science Foundation (NSF) Grant #2326894, NVIDIA Applied Research Accelerator Program Grant, and the Texas Advanced Computing Center (TACC). Any opinions, findings, conclusions, or recommendations are those of the authors and not of the funding agencies. We also thank Georgia Tech Research Institute for direct funding of selected authors.

References

[1] [n. d.]. *PolyBench*. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[2] UCLA VAST Lab [n. d.]. *UCLA-VAST/AutoDSE*. UCLA VAST Lab. https://github.com/UCLA-VAST/AutoDSE

[3] Xilinx [n. d.]. *Xilinx/Merlin-Compiler*. Xilinx. https://github.com/Xilinx/merlin-compiler

[4] AMD/Xilinx. 2021. Basic Examples for Vitis HLS. GitHub.

[5] AMD/Xilinx. 2022. Vitis Accel Examples' Repository. GitHub.

[6] Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, and Jason Cong. 2023. Towards a Comprehensive Benchmark for High-Level Synthesis Targeted to FPGAs. In *Thirty-Seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[7] Hanqiu Chen and Cong Hao. 2022. Mask-Net: A Hardware-Efficient Object Detection Network with Masked Region Proposals. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, Gothenburg, Sweden, 131–138. https://doi.org/10.1109/ASAP54787.2022.00030

[8] Hanqiu Chen and Cong Hao. 2023. DGNN-booster: A Generic FPGA Accelerator Framework for Dynamic Graph Neural Network Inference. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Marina Del Rey, CA, USA, 195–201. https://doi.org/10.1109/FCCM57271.2023.00029

[9] Vidya A. Chhabria, Yanqing Zhang, Haoxing Ren, Ben Keller, Brucek Khailany, and Sachin S. Sapatnekar. 2021. MAVIREC: ML-Aided Vectored IR-Drop Estimation and Classification. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[10] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. [n. d.]. Source-to-Source Optimization for HLS. In *FPGAs for Software Programmers*, Dirk Koch, Frank Hannig, and Daniel Ziener (Eds.). Springer International Publishing, 137–163. https://doi.org/10.1007/978-3-319-26408-0_8

[11] W. J. Conover. 1999. *Practical Nonparametric Statistics* (3rd ed ed.). Wiley, New York.

[12] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline FY Young, and Zhiru Zhang. 2018. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 129–132.

[13] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F.Y. Young, and Zhiru Zhang. 2018. Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 129–132. https://doi.org/10.1109/FCCM.2018.00029

[14] Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca Carloni, and Laura Pozzi. [n. d.]. *DB4HLS: A Database of High-Level Synthesis Design Space Explorations*. https://doi.org/10.48550/arXiv.2101.00587 arXiv:2101.00587 [cs]

[15] Quentin Gautier, Alric Althoff, Pingfan Meng, and Ryan Kastner. 2016. Spector: An opencl fpga benchmark suite. In *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 141–148.

[16] Pingakshya Goswami, Masoud Shahshahani, and Dinesh Bhatia. [n. d.]. MLSBench: A Synthesizable Dataset of HLS Designs to Support ML Based Design Flows. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2020-02-24) *(FPGA '20)*. Association for Computing Machinery, 312. https://doi.org/10.1145/3373087.3375378

[17] Winston Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Frédéric Kaplan, Sabine Süsstrunk, and Giovanni De Micheli. 2018. Deep Learning for Logic Optimization Algorithms. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*.

[18] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. [n. d.]. CHStone: A Benchmark Program Suite for Practical C-based High-Level Synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)* (2008-05). 1192–1195. https://doi.org/10.1109/ISCAS.2008.4541637

[19] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel Programming for FPGAs. https://doi.org/10.48550/arXiv.1805.03648 arXiv:arXiv:1805.03648

[20] Ryan Gary Kim, Janardhan Rao Doppa, and Partha Pratim Pande. 2018. Machine Learning for Design Space Exploration and Optimization of Manycore Systems. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–6.

[21] Zhe Lin, Zike Yuan, Jieru Zhao, Wei Zhang, Hui Wang, and Yonghong Tian. 2022. PowerGear: Early-Stage Power Estimation in FPGA HLS via Heterogeneous Edge-Centric GNNs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. https://doi.org/10.23919/DATE54114.2022.9774682

[22] Zhe Lin, Jieru Zhao, Sharad Sinha, and Wei Zhang. 2020. HL-Pow: A Learning-Based Power Modeling Framework for High-Level Synthesis. In *25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. https://doi.org/10.1109/ASP-DAC47756.2020.9045442

[23] Dong Liu and Benjamin Carrion Schafer. 2016. Efficient and reliable High-Level Synthesis Design Space Explorer for FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*.

[24] Yixuan Luo, Cheng Tan, Nicolas Bohm Agostini, Ang Li, Antonino Tumeo, Nirav Dave, and Tong Geng. 2023. ML-CGRA: An Integrated Compilation Framework to Enable Efficient Machine Learning Acceleration on CGRAs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*.

[25] Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Houman Homayoun, and Setareh Rafatirad. 2019. Pyramid: Machine Learning Framework to Estimate the Optimal Timing and Resource Usage of a High-Level Synthesis Design. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*.

[26] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. [n. d.]. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)* (2014-10). 110–119. https://doi.org/10.1109/IISWC.2014.6983050

[27] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2023. FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Montreal, QC, Canada, 1099–1112. https://doi.org/10.1109/HPCA56546.2023.10071015

[28] Rishov Sarkar and Cong Hao. 2023. LightningSim: Fast and Accurate Trace-Based Simulation for High-Level Synthesis. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Marina Del Rey, CA, USA, 1–11. https://doi.org/10.1109/FCCM57271.2023.00010

[29] Rishov Sarkar, Hanxue Liang, Zhiwen Fan, Zhangyang Wang, and Cong Hao. 2023. Edge-MoE: Memory-Efficient Multi-Task Vision Transformer Architecture with Task-Level Sparsity via Mixture-of-Experts. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, San Francisco, CA, USA, 01–09. https://doi.org/10.1109/ICCAD57390.2023.10323651

[30] Gagandeep Singha, Dionysios Diamantopoulosb, Juan Gómez-Lunaa, Sander Stuijkc, Henk Corporaalc, and Onur Mutlu. 2022. LEAPER: Fast and Accurate FPGA-based System Performance Prediction via Transfer Learning. In *IEEE 40th International Conference on Computer Design (ICCD)*. https://doi.org/10.1109/ICCD56317.2022.00080

[31] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. [n. d.]. *AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators*. https://doi.org/10.48550/arXiv.2009.14381 arXiv:2009.14381 [cs]

[32] Yingfan Wang, Haiyang Huang, Cynthia Rudin, and Yaron Shaposhnik. 2021. Understanding How Dimension Reduction Tools Work: An Empirical Approach to Deciphering t-SNE, UMAP, TriMap, and PaCMAP for Data Visualization. *Journal of Machine Learning Research* 22, 201 (2021), 1–73. http://jmlr.org/papers/v22/20-1061.html

[33] Zhigang Wei, Aman Arora, Ruihao Li, and Lizy John. [n. d.]. HLSDataset: Open-Source Dataset for ML-assisted FPGA Design Using High Level Synthesis. In *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (Porto, Portugal, 2023-07). IEEE, 197–204. https://doi.org/10.1109/ASAP57973.2023.00040

[34] Clifford Wolf and Johann Glaser. 2013. Yosys - a Free Verilog Synthesis Suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. Linz, Austria.

[35] Nan Wu, Yuan Xie, and Cong Hao. 2021. Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*. 39–44.

[36] Nan Wu, Yuan Xie, and Cong Hao. 2022. IRONMAN-PRO: Multiobjective design space exploration in HLS via reinforcement learning and graph neural network-based modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 3 (2022), 900–913.

[37] Xiaofan Zhang, Haoming Lu, Cong Hao, Jiachen Li, Bowen Cheng, Yuhong Li, Kyle Rupnow, Jinjun Xiong, Thomas Huang, Honghui Shi, Wen-Mei Hwu, and Deming Chen. 2020. SkyNet: A Hardware-Efficient Method for Object Detection and Tracking on Embedded Systems. *Proceedings of Machine Learning and Systems* 2 (March 2020), 216–229.

[38] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. [n. d.]. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2018-02-15) *(FPGA '18)*. Association for Computing Machinery, 269–278. https://doi.org/10.1145/3174243.3174255

7 Artifact Appendix

7.1 Abstract

The HLSFactory framework includes multiple software and dataset components, which are available as public open-source releases and artifacts. We briefly outline these components and how to access them. We plan to expand many aspects of our work in the future (e.g., more built-in HLS benchmarks and designs, additional tool flow integrations, enhanced design frontends) and openly encourage contributions and use of HLSFactory.

For users strictly interested in running the artifact evaluation to reproduce data and results for various reported case studies, details can be found in §7.1.4 and at the following repository: https://github.com/sharc-lab/hlsfactory-artifact-eval.

7.1.1 HLSFactory Python Library. The HLSFactory Python library, `hlsfactory`, provides APIs and logic for various features including loading HLS designs (locally on disk or from built-in common HLS benchmarks and designs), expanding designs through design space sampling, running parallel tool flows for HLS vendor tools, and extracting+serializing+archiving structured HLS and FPGA tool data (including reports and build artifacts).

- Source code repository: https://github.com/sharc-lab/HLSFactory
  - Archived at https://zenodo.org/doi/10.5281/zenodo.12989544 (DOI: 10.5281/zenodo.12989544)
- Documentation: https://sharc-lab.github.io/HLSFactory/docs/
  - Archived as part of the source code repository
- Install via `pip`: `pip install git+https://github.com/sharc-lab/HLSFactory`
- Install via `conda`: `conda install -channel https://sharc-lab.github.io/HLSFactory/dist-conda hlsfactory`
- Install via `mamba`: `mamba install -channel https://sharc-lab.github.io/HLSFactory/dist-conda hlsfactory`

We highly encourage users to review the documentation for details on the framework, walkthroughs of various demos and case studies with accompanying code and Jupyter Notebooks, and information on how to extend and add new datasets and built-in designs.

7.1.2 HLSFactory's Collection of HLS Benchmarks and Designs. One core contribution of this work is the collection of source code for common HLS benchmarks and other open-source and academic HLS designs. We have created design space descriptions and entry point scripts for each design, necessary for the various tool flows supported by our work, and tested our flow on each design.

Our initial release includes designs from the following sources: Polybench [1], MachSuite [26], Rosetta [38], CHStone [18], the "Parallel Programming for FPGAs" textbook [19], AMD/Xilinx sample HLS designs [4, 5], and various HLS accelerators from Sharc Lab [7, 8, 29, 37]. These designs can be found in the HLSFactory Python library itself under the repository path `hlsfactory/hls_dataset_sources`.

A notable feature is that these designs are built into the packaged Python library, available via `pip` and `conda`. Users who install `hlsfactory` can load designs locally without additional downloads. Additionally, users can still load custom designs locally at runtime.

7.1.3 Pre-Generated Datasets of HLS Synthesized and Implemented Designs. While running our case studies, we ran various end-to-end dataset generations flows. The pre-generated datasets include design sources (with sampled optimization directives if design space sampling is used), HLS synthesized designs (including HLS reports, generated hardware IP, and HLS scheduling and binding data), and, in some runs, FPGA post-implementation reports. These datasets can save users and researchers significant time by providing a dataset fully synthesized and implemented HLS designs with important intermediate artifacts.

We include the following pre-generated datasets: `Design Space Base Dataset`, `Design Space Sampled Dataset`, `Intel Design Flow Dataset`, `Parallelization Test Dataset`, `Regression Benchmarking Test Dataset`.

We archive and host these datasets on Zenodo: https://zenodo.org/doi/10.5281/zenodo.13117901 (DOI: 10.5281/zenodo.13117901)

7.1.4 Scripts to Reproduce Case Study Results. We provide Python scripts to reproduce the results of various case studies, including figures and numerical results. This includes scripts to generate design datasets from scratch and perform the case study analyses. Generating design data requires HLS and FPGA vendor tools and can take over 24 hours for the largest datasets used in this work. Therefore, users can also use the pre-generated datasets from §7.1.3 and only run the required analysis scripts.

The code for running these scripts as an artifact evaluator, along with detailed instructions, is available at the GitHub repository: https://github.com/sharc-lab/hlsfactory-artifact-eval.

This repository is archived at https://zenodo.org/doi/10.5281/zenodo.13117886 (DOI: 10.5281/zenodo.13117886).

7.2 Artifact Check-List (meta-information)

- **Data set**: `Polybench`, `MachSuite`, `Rosetta`, `CHStone`, `"Parallel Programming for FPGAs"`, `Xilinx/Vitis-HLS-Introductory-Examples`
- **Run-time environment**: `Linux`, `Windows`, `MacOS`
- **Metrics**: `Runtime`, `HLS Reported Latency`, `HLS Reported Resource Usage`, `Post-Implementation Timing Metrics`, `Post-Implementation Resource Usage`, `Post-Implementation Power Estimation`
- **Output**: `HLS Synthesis Reports`, `HLS Synthesized Hardware IP`, `Post-Implementation Reports`
- **Experiments**: `"ML Prediction of Post-Implementation Quality-of-Results Metrics"`, `"Design Space Coverage"`, `"Speedup of Fine-Grained Design Parallelism"`, `"Targeting Different HLS Vendors"`, `"Integrating Released Data from Other Works"`, `"Regression Benchmarking of HLS Synthesis Tools"`
- **Proprietary EDA tools**: `AMD/Xilinx Vitis HLS 2023.1`, `AMD/Xilinx Vivado 2023.1`, `AMD/Xilinx Vitis HLS 2021.1`, `AMD/Xilinx Vivado 2121.1`, `Intel HLS Compiler (i++) 21.1.0`, `Intel Quartus Prime 21.1.0`
- **How much disk space required (approximately)?**: ≈ 200 GB
- **How much time is needed to prepare workflow (approximately)?**: ≈ 20 Minutes
- **How much time is needed to complete experiments (approximately)?**: ≈ 24 hours (using 32 cores)
- **Publicly available?**: Yes!
- **Code licenses (if publicly available)?**: `GNU AGPLv3`
- **Data licenses (if publicly available)?**: `CC BY-SA 4.0`
- **Archived (provide DOI)?**: Yes! Datasets: [`10.5281/zenodo.13117901`], Code Repositories: [`10.5281/zenodo.12989544`, `10.5281/zenodo.13117886`]

7.3 Description

7.3.1 How To Access. The main artifact evaluation code for reproducing results presented in the paper is hosted at this GitHub repository: https://github.com/sharc-lab/hlsfactory-artifact-eval.

7.3.2 Hardware Dependencies. No specialized hardware is needed. We recommend a desktop workstation or server with as many cores as possible (for faster parallel dataset generation) and a common Linux-based distrobution (such as Ubuntu).

7.3.3 Software Dependencies. HLSFactory and the artifact evaluation scripts are implemented in Python and require version 3.10 or higher. The `hlsfactory` library depends on `pandas`, `psutil`, `PyYAML`, `tqdm`, and `python-dotenv`. The artifact evaluation scripts additionally require `pacmap`, `scikit_learn`, `scipy`, `seaborn`, and `hlsfactory`.

7.3.4 Commercial Software Dependencies.

To run Xilinx-based dataset generation flows, AMD/Xilinx's Vitis HLS and Vivado are required, with most design runs using version `2023.1`. The regression testing case study requires version `2021.1`. For Intel-based flows, Intel's HLS Compiler and Quartus Prime are needed, with version `21.1.0` required for the Intel design run.

7.3.5 Datasets.

All the required HLS designs (source code, tool scripts, design space descriptions) are built into the `hlsfactory` package itself. For more details, refer to §7.1.2.

7.4 Installation

Installation of the `hlsfactory` package (as described in §7.1.1) and Python requirements can be done using `pip` or `conda` based tools.

7.5 Experiment Workflow

For details on running and generating case study results, please refer to the artifact evaluation repository (§7.1.4). The process involves obtaining an HLS dataset either by running a dataset generation script or by sourcing a pre-generated dataset from Zenodo. After obtaining the dataset, the user runs a specific case study analysis or visualization script to generate the relevant figures and results. We also specify which case study analyses require which datasets to be run or sourced.

7.6 Evaluation and Expected Results

The analysis scripts should produce figures and numerical results similar to those in the paper. The entire workflow is designed to be deterministic, assuming the vendor tools are deterministic. While we have identified most sources of randomness that we allow users to control with a random seed (e.g., random sampling in design space expansion), some elements remain beyond our control, such as `pacmap`'s fitting, which is not fully deterministic even with `random_state` set.

7.7 Experiment Customization

Users and evaluators can modify hardcoded parameters in the dataset generation runs or analysis scripts (e.g., random samples for design space expansion, dimensionality reduction parameters). As "proof-of-concept" demos, our case studies allow for modification and extension of `hlsfactory` to support new data and tools, both locally at runtime and as contributions to the published Python package.

7.8 Notes

For more detailed and complete instructions, please refer to the `README.md` in the artifact evaluation code repository.

7.9 Methodology

Submission, reviewing and badging methodology:
https://www.acm.org/publications/policies/artifact-review-and-badging-current,
http://cTuning.org/ae/submission-20201122.html, https://github.com/ml-eda/artifact-evaluation/.