Invited: ScaleHLS: a Scalable High-Level Synthesis Framework with Multi-level Transformations and Optimizations

Hanchen Ye[1], HyeGang Jun[1], Hyunmin Jeong[1], Stephen Neuendorffer[2], Deming Chen[1]
[1]University of Illinois at Urbana-Champaign, [2]Advanced Micro Devices Inc.
{hanchen8, hgjun2, hyunmin2, dchen}@illinois.edu, stephenn@amd.com

ABSTRACT

This paper presents an enhanced version of a scalable HLS (High-Level Synthesis) framework named ScaleHLS, which can compile HLS C/C++ programs and PyTorch models to highly-efficient and synthesizable C++ designs. The original version of ScaleHLS achieved significant speedup on both C/C++ kernels and PyTorch models [14]. In this paper, we first highlight the key features of ScaleHLS on tackling the challenges present in the representation, optimization, and exploration of large-scale HLS designs. To further improve the scalability of ScaleHLS, we then propose an enhanced HLS transform and analysis library supported in both C++ and Python, and a new design space exploration algorithm to handle HLS designs with hierarchical structures more effectively. Comparing to the original ScaleHLS, our enhanced version improves the speedup by up to 60.9× on FPGAs. ScaleHLS is fully open-sourced at https://github.com/hanchenye/scalehls.

1 INTRODUCTION

With the end of Moore's law, general-purpose computing platforms are facing significant challenges on meeting the performance and power efficiency demands of emerging applications simultaneously. DSA (Domain-Specific Accelerator) has been attracting wide attention from both industry and academia [6, 16]. However, as today's DSA can contain billions of transistors and cost hundreds of engineer-years on the design and verification of the SoC (System-on-Chip) circuit [7], the development of DSA is considerably challenging to match the rapid evolvement of the targeted domain. To bridge the gap, existing literature has explored new techniques on multiple fronts, including compilation tools [2, 9], programming languages [4, 11], and agile development methodologies [1, 7]. Among these efforts, HLS (High-Level Synthesis) is widely adopted and becomes a promising technique due to its unique advantages on productivity and rapid DSE (Design Space Exploration). However, current HLS tools are still limited to small functional modules without complicated computation and memory hierarchy. To improve the scalability of HLS solutions, we proposed ScaleHLS [14], which supports a hierarchical and multi-level IR (Intermediate Representation) of HLS designs. On top of the hierarchical IR, different HLS optimization and management problems can be abstracted to the suitable level and solved in a scalable manner. To further advance the ScaleHLS framework, we propose two unique technical improvements in this paper as follows.

- *Transform and Analysis Library.* We propose an enhanced library to improve the modularity and productivity of HLS optimizations. A set of commonly used classes and methods for HLS transform and analysis are designed and packaged, which support rapid integration in both C++ and Python.

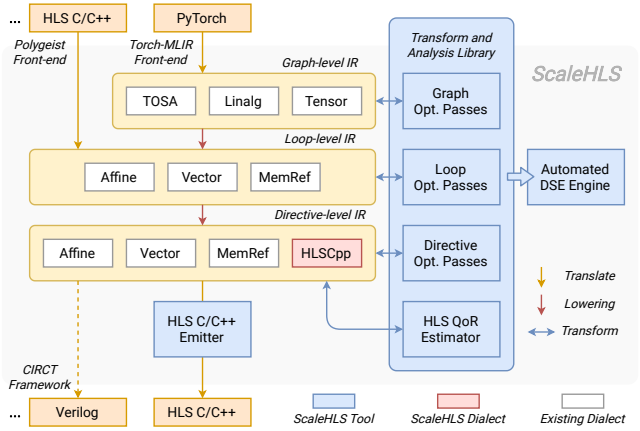


Figure 1: ScaleHLS Framework

- *Design Space Exploration.* We propose a new DSE algorithm based on dynamic programming and evolutionary algorithm that can handle hierarchical HLS designs with higher scalability.

In the remainder of this paper, we first introduce the key features of ScaleHLS and the new improvements in Section 2. Then, we provide the evaluation results in Section 3. Finally, we conclude the paper and propose some directions of future works in Section 4.

2 SCALEHLS FRAMEWORK

2.1 Framework Overview

The goal of ScaleHLS is to leverage the multi-level hierarchy of HLS designs and advanced compilation techniques to address the automation and scalability issues of existing HLS flows. Figure 1 shows the overall architecture of ScaleHLS framework. ScaleHLS is built upon MLIR [9] and supports C/C++ and PyTorch programs as inputs through the Polygeist [10] and Torch-MLIR [3] front-ends, respectively. Once the input programs are parsed into MLIR, ScaleHLS supports three levels of representation to apply the HLS-oriented optimizations progressively. At the highest level, ScaleHLS uses `tosa` (Tensor Operator Set Architecture), `linalg` (Linear Algebra), and `tensor` dialects [9] to represent the tensor-level computation graph, where graph optimizations, such as node fusion and coarse-grained pipelining, can be performed efficiently. At the middle level, ScaleHLS uses `affine`, `vector`, and `memref` (Memory Reference) dialects [9] to explicitly represent the loop structures in an affine format in order to perform the affine loop analyses and optimizations. Finally, at the lowest level, we introduce an `hlscpp` dialect to represent the HLS-specific directives (such as loop pipelining) and primitives (such as multiplication primitive) to fine-tune the hardware micro-architecture and enable an efficient code generation.

On top of each level of IR, ScaleHLS provides a set of transform passes to optimize HLS designs. By performing each transform pass

Table 1: ScaleHLS Transform and Analysis Library

Type	Name	Description
Class	ScaleHLSEstimator	QoR estimator class
	ScaleHLSExplorer	Design space explorer class
	ScaleHLSEmitter	HLS C++ emitter class
Graph Opts.	TosaSimplifyGraph	Remove redundant TOSA ops
	TosaNodeFusion	Heuristic TOSA op fusion
	LegalizeDataflow	Legalize multi-consumer ops and bypass paths in TOSA dataflow
Loop Opts.	AffineLoopPerfection	Perfectize loop bands
	RemoveVariableBound	Remove variable loop bounds
	AffineLoopTile	Tile perfect loop bands
	AffineLoopOrderOpt	Permute loops to increase distances of loop-carried dependencies
	AffineLoopUnrollJam	Unroll and jam perfect loop bands
	SimplifyAffineIf	Remove redundant if statements
Mem. Opts.	AffineStoreForward	Replace redundant memory loads with forwarded scalars
	SimplifyMemrefAccess	Remove redundant loads/stores
	ReduceInitialInterval	Reduce loop pipeline II
Direct. Opts.	FuncPipelining	Apply function pipelining
	LoopPipelining	Apply loop pipelining
	CreateHLSPrimitive	Generated HLS-specific primitives
	ArrayPartition	Automatic array partition

at the "correct" level of abstraction, ScaleHLS is able to leverage the intrinsic hierarchy of HLS designs and reduce the algorithmic complexity of transforms. Meanwhile, a static schedule-based QoR (Quality of Results) estimator is designed to predict the resource utilization and performance through IR analysis. The QoR estimator can rapidly evaluate the benefits of different transform passes to avoid immature combination of optimizations and guide the configuration of tunable parameters. Furthermore, to leverage the reusability advantages of the library-driven design philosophy, we parameterize and encapsulate the APIs under the hood of the aforementioned transform passes and QoR estimator into an HLS-dedicated transform and analysis library available in both C++ and Python. The Python APIs are packaged into a `scalehls` module and can be layered on the MLIR built-in `mlir` module to traverse and transform HLS designs using Python. The unique techniques included in this library are elaborated in Section 2.2.

Using the transform and analysis library, we propose a DSE engine to automatically tune the optimization parameters exposed by the transform APIs. To efficiently iterate in the large design space coming with large-scale HLS designs, the DSE engine integrates the QoR estimator to evaluate the discovered design points. The algorithmic details of the proposed DSE are introduced in Section 2.3. Finally, the optimized IR is emitted as synthesizable HLS C/C++ code, which can be passed to downstream RTL-generation tools, such as Xilinx Vitis HLS [5]. Meanwhile, integration with the CIRCT [2] framework to directly generate RTL from MLIR is currently under development.

2.2 ScaleHLS Transform and Analysis Library

Based on the broad recognition of the “as a library” philosophy of LLVM [8], we envision that an HLS-dedicated optimization library will effectively improve the modularity and reusability of HLS compilation flows. Hence, we propose an enhanced transform and

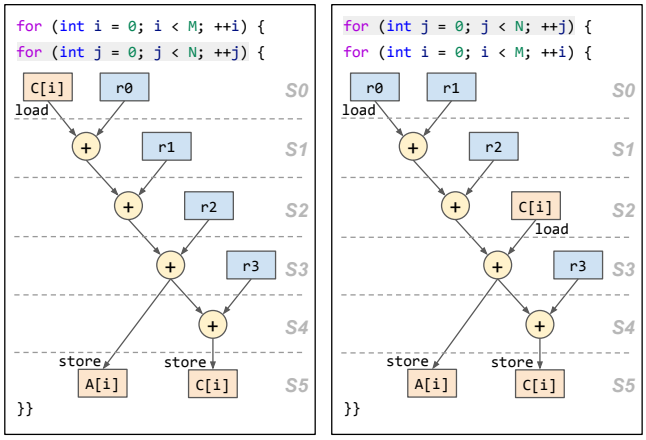
```python
import scalehls
import mlir.ir
import mlir.dialects import func

# Parse C/C++ into MLIR.
... ...
mod = mlir.ir.Module.parse(fin, ...)

# Traverse all functions in MLIR.
for f in mod.body:
    if not isinstance(f, func.FuncOp): pass
    f.__class__ = func.FuncOp

    # Transform loop bands and arrays in the function.
    for band in scalehls.LoopBandList(f):
        scalehls.loop_perfection(band)
        factors = np.full(band.depth, 4)
        scalehls.loop_tiling(band, factors)

    for array in scalehls.ArrayList(f):
        type = array.type
        type.__class__ = mlir.ir.MemRefType
        factors = np.full(type.rank, 4)
        scalehls.array_partition(array, factors, "cyclic")

# Emit transformed MLIR to HLS C++.
buf = io.StringIO()
scalehls.emit_hlscpp(mod, buf)
```

Listing 1: Using `scalehls` Python Library

analysis library in ScaleHLS, which provides a set of unique classes and methods for the evaluation and optimization of HLS designs as well as the code generation. Table 1 lists the current supported classes and methods in the library. The three classes, `Estimator`, `Explorer`, and `Emitter`, are designed to construct and solve the QoR estimation, DSE, and HLS C++ emission problems, respectively. As we mentioned in Section 2.1, the `Estimator` is integrated into the `Explorer` as an object in order to rapidly evaluate the discovered design points. As the DSE problem is often better solved by partitioning into a number of small problems, the `Estimator` is designed in a hierarchical way such that different pieces of the HLS designs can be estimated separately to avoid estimating the entire design in every iteration. Similarly, the `Emitter` is also designed hierarchically, which enables the compiler to interact with downstream RTL-generation tools in a more fine-grained manner.

All the optimization methods are categorized into four groups, *Graph*, *Loop*, *Memory*, and *Directive*. 1) Graph optimizations are focused on the `tosa` dialect, such as TOSA graph simplification and TOSA operation fusion. 2) Loop optimizations are built using the libraries of the `affine` dialect. We introduce loop perfection and variable bound loop removal methods to regularize loop structures and pave the path for subsequent optimizations. Then, loop tiling and unrolling are supported to improve the data locality and computation parallelisms, respectively. 3) Considering memory bandwidth is one of the most scarce resources in hardware accelerators, we introduce multiple memory optimizations to improve the efficiency of memory access. 4) Finally, directive optimizations, including loop and function pipelining and array partition, are built on top of the abstractions of our `hlscpp` dialect.

2.2.1 Python Binding. As ScaleHLS is implemented in C++, all the classes and methods described above are also exposed as C++ APIs. While C++ allows library users to get lower-level control of the optimizations, HLS designers or researchers may prefer a more user-friendly and productive language, such as Python, to access



(a) Dataflow before Opts. (b) Dataflow after Opts.

Figure 2: Example of Pipeline Optimizations

the library. Additionally, recent literature [13, 15] has shown great interests in leveraging machine learning or graph algorithms to advance HLS optimizations. To take advantage of the existing Python libraries supporting these algorithms and enable the seamless integration with ScaleHLS, we design a `scalehls` Python module by wrapping and binding the C/C++ APIs to Python. Listing 1 shows an example of using the `scalehls` library in Python. In line 1-3, the `scalehls` module is imported together with the `mlir` modules, which exposes the IR and built-in dialect APIs, including operation, type, value, and a `func` dialect. In line 5-12, we parse the input C/C++ code into an MLIR module instance called `mod` through the Polygeist front-end and start to traverse the contained functions. To help with the function transformation, we provide two handy Python classes (shown in line 15 and 20), `LoopBandList` and `ArrayList`, that can be initialized with a function instance and iterate on all loop bands and arrays inside of the function. Then, in line 15-18, we perform loop perfection and tiling for each loop band. Notably, the `loop_tiling` API is deeply integrated with the `numpy` library at C/C++ level such that a `numpy` array can be directly used as tiling factors. This feature allows the ScaleHLS transform APIs to directly interact with the algorithm-side frameworks in Python. Similarly, in line 20-24, each array in the function is cyclically partitioned with a factor of 4 on each dimension. Finally, in line 26-28, the transformed MLIR is exported to an output buffer as synthesizable HLS C++ by calling the `emit_hlscpp` method.

2.2.2 Pipeline Optimizations. Loop pipelining is an essential HLS optimization to improve throughput by enabling fine-grained spatial parallelism. However, existing HLS tools typically demand designers to manually implement different pipeline design choices to achieve a satisfactory design quality. In the ScaleHLS library shown in Table 1, two main passes are introduced to automatically optimize the performance of pipeline, `AffineLoopOrderOpt` and `ReduceInitialInterval`. Figure 2 shows the dataflow of an example before and after the pipeline optimizations. This example contains two nested loops, loop-`i` and loop-`j`, while the loop body is pipelined into 6 stages, *S0* to *S5*. We can observe the memory load and store of `C[i]` has an anti-dependence carried by loop-`j`. The minimum *II* (Initiation Interval) constrained by loop-carried dependencies [17] can be calculated as:

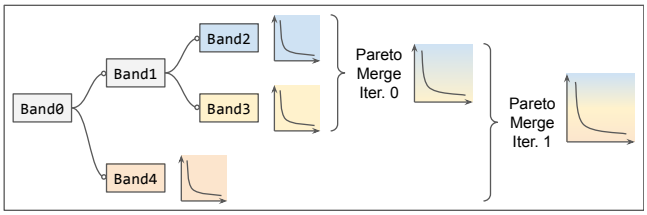


Figure 3: Example of Pareto Frontier Merging

$$II_{min}^{dep} = \min_{p} \left(\left\lceil \frac{Delay_p}{Distance_p} \right\rceil \right), \tag{1}$$

where p denotes each pair of dependence existed in the loop nest. To minimize the *II*, `AffineLoopOrderOpt` will permute the loops associated with dependencies to outermost to maximize the $Distance_p$ in the loop iteration space, while `ReduceInitialInterval` will manipulate the order of operations in the loop body to minimize the $Delay_p$ between dependent operations. As shown in Figure 2(b), loop-`j` is permuted to the outermost loop level. Meanwhile, as the memory load of `C[i]` dominates the store through a chain of commutative operators, the load can be moved around different entries of the chain without changing the functionality. Note that as the adder result in *S3* is consumed by a store of `A[i]` apart from the subsequent adder, the load of `C[i]` can only be moved to *S2* instead of *S3* as the final optimization result shown in Figure 2(b).

2.3 Design Space Exploration

The tunable parameters exposed by the transform APIs of ScaleHLS library can lead to a large and multi-dimensional design space. In addition, existing literature [15] has shown that different HLS optimization techniques, such as loop unrolling and array partition, may have correlation or conflict with each other, making the design space more complicated to explore. To tackle this challenge, [14] has proposed a neighbor-traversing algorithm to search for the Pareto dominant design points of the latency-area space. When scaling up to multiple loop bands with hierarchical structure, we propose a DP (Dynamic Programming) algorithm in this paper to avoid the explosion of complexity: we first break down the global optimization problem into a number of sub-problems and then merge the Pareto design points of each sub-problem to deliver the final result. The left side of Figure 3 shows a tree of loop bands, where *Band1* and *Band4* are contained by *Band0*, and *Band2-3* are contained by *Band1*. We define the optimization of each "leaf" loop band (*Band2-4*) as a sub-problem and employ the DSE algorithm in [14] to find the local Pareto frontier. Then, we hierarchically merge the Pareto frontiers until all sub-problems are merged. In the case of Figure 3, *Band2* and *Band3* are merged in the first iteration, and the intermediate result is then merged with *Band4* in the second iteration, which produces the final Pareto frontier of the entire design.

However, our experiments showed that for cases when there are numerous sub-problems, indeed, due to the correlation between the sub-problems, our initial DP approach produced a sub-optimal design. The main complexity lies in the array partition scheme, as the arrays represent the on-chip buffers, naturally are shared between sub-problems. As a result, an analytical method for finding the optimal array partition scheme by inspection of the code alone is challenging. This problem is further exasperated when the array sizes are not regular, not having a common array size and

Table 2: Evaluation Results Compared to ScaleHLS [14]

		Latency	Speedup	DSP	FF	LUT
2mm	Baseline	26 *ms*	1x	2%	1%	2%
	ScaleHLS	784 μs	33x	78%	45%	65%
	Ours (DP+EA)	143 μs	182x	58%	42%	78%
3mm	Baseline	404 *ms*	1x	2%	1%	3%
	ScaleHLS	1,755 μs	230x	27%	22%	67%
	Ours (DP+EA)	414 μs	976x	41%	54%	87%
atax	Baseline	2,738 μs	1x	2%	0%	1%
	ScaleHLS	670 μs	4.1x	4%	1%	3%
	Ours (DP+EA)	11 μs	249x	82%	58%	78%
doitgen	Baseline	50 *ms*	1x	2%	0%	2%
	ScaleHLS	1,997 μs	26x	9%	10%	18%
	Ours (DP+EA)	151 μs	331x	62%	97%	90%
spam-filter	Baseline	7,842 *ms*	1x	5%	2%	7%
	ScaleHLS	112 *ms*	70x	76%	22%	57%
	Ours (DP+EA)	112 *ms*	70x	76%	22%	57%

dimension. A solution to this problem is inspired by Chimera [15], which presents a method that, through the use of EA (Evolutionary Algorithm), can subsequently evolve specific parameters that the authors of the paper call tune-able knobs. However, this method still relies on the user to manually specify the knobs to optimize. Building upon Chimera, we design an automated algorithm that identifies the critical knobs. This method identifies certain arrays that are shared between sub-problems and weights them according to the criticality to the overall design. Using the weight information, the array partition scheme is evolved for the selected knobs, where higher weighted knobs are explored to a greater extent.

3 EVALUATION

In this section, we evaluate the proposed framework using benchmarks from PolyBench [12] and Rosetta benchmark suite [18], which are beyond the benchmarks used in original ScaleHLS [14]. All the results are collected with Xilinx Vivado HLS 2019.1 targeting Xilinx XC7Z020 FPGA, an edge FPGA with 220 DSPs and 53,200 LUTs. Table 2 shows the evaluation results compared to ScaleHLS [14] and a baseline, where the *Baseline* rows show the results of original designs directly going through Vivado HLS without any optimizations. We can observe the enhanced framework proposed in this paper performs the best due to the new pipeline optimizations and DSE algorithm. The performance gains reported for 2mm and 3mm is mainly attributed to EA being able to find array partition schemes suitable for correlated sub-problems. At the same time, the huge performance gains reported for atax and doitgen, although benefiting from the optimized array partition scheme, is mainly due to EA being able to explore loops further. Due to how the DP approach breaks the global design space into smaller sub-problems, resource allocation for each sub-problem will be either greedy or based on some heuristics. We complement the DP approach with the globally scoped EA exploring the resource allocation between sub-problems, and this approach is most beneficial for the atax and doitgen benchmarks. As for the spam-filter Rosetta benchmark, we are not able to further optimize the benchmark due to sub-problems having the same loop trip count. As a result, when we search for a more optimized design, we could only match the design initially produced by ScaleHLS.

4 CONCLUSION AND FUTURE WORKS

In this paper, we introduce a ScaleHLS framework with multi-levels of IR and scalable support on the optimization of HLS designs. We propose an HLS-dedicated transform and analysis library in both C++ and Python that improves the modularity of the framework. A hierarchical DSE algorithm is proposed, which achieves promising speedup on a set of benchmarks from PolyBench and Rosetta. Several directions are left as future works: 1) Memory management. The on-chip and off-chip memory resources are supposed to be carefully managed in order to achieve the best computing efficiency. 2) Hierarchical dataflow. Dataflow optimization is essential for multi-kernel HLS designs to enable the spatial parallelism and improve the throughput. 3) RTL generation. We are working on the integration of ScaleHLS and CIRCT [2] aiming to directly generate optimized RTL designs from MLIR. 4) IP integration. We can leverage existing highly-optimized IPs to reduce the size of design spaces to explore and improve the quality of generated designs.

ACKNOWLEDGMENTS

This work is supported in part by Xilinx Center of Excellence at UIUC, Xilinx Adaptive Compute Cluster (XACC) initiative, and BAH HT 15-1158 contract.

REFERENCES

[1] Alon Amid et al. 2020. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro* 40, 4 (2020), 10–21.
[2] CIRCT Authors. 2022. CIRCT: Circuit IR Compilers and Tools. https://github.com/llvm/circt.
[3] Torch-MLIR Authors. 2022. Torch-MLIR. https://github.com/llvm/torch-mlir.
[4] Sitao Huang et al. 2021. Pylog: An algorithm-centric python-based FPGA programming and synthesis flow. *IEEE Trans. Comput.* 70, 12 (2021), 2015–2028.
[5] Xilinx Inc. 2022. Xilinx Vitis HLS LLVM 2021.2. https://github.com/Xilinx/HLS.
[6] Norman P Jouppi et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.
[7] Brucek Khailany et al. 2018. A modular digital VLSI flow for high-productivity SoC design. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[8] Chris Lattner et al. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.
[9] Chris Lattner et al. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).
[10] William S Moses et al. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 45–59.
[11] Rachit Nigam et al. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 393–407.
[12] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012), 1–1.
[13] Nan Wu et al. 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.
[14] Hanchen Ye et al. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *The 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE.
[15] Mang Yu et al. 2021. Chimera: A Hybrid Machine Learning-Driven Multi-Objective Design Space Exploration Tool for FPGA High-Level Synthesis. In *IDEAL*.
[16] Xiaofan Zhang et al. 2020. DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.
[17] Jieru Zhao et al. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 430–437.
[18] Yuan Zhou et al. 2018. 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*. 269–278.