High-level Synthesis for Domain Specific Computing

Hanchen Ye
hanchen8@illinois.edu
University of Illinois at Urbana-Champaign, USA

Hyegang Jun
hgjun2@illinois.edu
University of Illinois at Urbana-Champaign, USA

Jin Yang
jin.yang@intel.com
Intel, USA

Deming Chen
dchen@illinois.edu
University of Illinois at Urbana-Champaign, USA

ABSTRACT

This paper proposes a High-Level Synthesis (HLS) framework for domain-specific computing. The framework contains three key components: 1) ScaleHLS, a multi-level HLS compilation flow. Aimed to address the lack of expressiveness and hardware-dedicated representation of traditional software-oriented compilers. ScaleHLS introduces a hierarchical intermediate representation (IR) for the progressive optimization of HLS designs defined in various high-level languages. ScaleHLS consists of three levels of optimizations, including graph, loop, and directive levels, to realize an efficient compilation pipeline and generate highly-optimized domain-specific accelerators. 2) AutoScaleDSE is an automated design space exploration (DSE) engine. Real-world HLS designs often come with large design spaces that are difficult for designers to explore. Meanwhile, the connections between different components of an HLS design further complicate the design spaces. In order to address the DSE problem, AutoScaleDSE proposes a random forest classifier and a graph-driven approach to improve the accuracy of estimating the intermediate DSE results while reducing the time and computational cost. With this new approach, AutoScaleDSE can evaluate thousands of HLS design points and find the Pareto-dominating design points within a couple of hours. 3) PyTransform is a flexible pattern-driven design customization flow. Existing HLS flows demand manual code rewriting or intrusive compiler customization to conduct domain-specific optimizations, leading to unscalable or inflexible compiler solutions. PyTransform proposes a Python-based flow that enables users to define custom matching and rewriting patterns at a high level of abstraction, being able to be incorporated into the DSL compilation flow in an automatic and scalable manner. In summary, ScaleHLS, AutoScaleDSE, and PyTransform aim to address the challenges present in the compilation, DSE, and customization of existing HLS flows, respectively. With the three key components, our newly proposed HLS framework can deliver a scalable and extensible solution for designing domain-specific languages to automate and speed up the process of designing domain-specific accelerators.

ISPD '23, March 26–29, 2023, Virtual Event, USA

ACM ISBN 978-1-4503-9978-4/23/03…$15.00
https://doi.org/10.1145/3569052.3580027

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**.

KEYWORDS

HLS, domain-specific language, domain-specific computing, MLIR, design space exploration

ACM Reference Format:
Hanchen Ye, Hyegang Jun, Jin Yang, and Deming Chen. 2023. High-level Synthesis for Domain Specific Computing. In *Proceedings of the 2023 International Symposium on Physical Design (ISPD '23), March 26–29, 2023, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3569052.3580027

1 INTRODUCTION

The exploding complexity and efficiency requirements of modern applications are stimulating a strong demand for hardware acceleration using heterogeneous platforms such as FPGAs. However, a high-quality FPGA design is very hard to implement and optimize as it requires FPGA expertise and proficiency in expressing the design using low-level hardware descriptive languages (HDL). As a result, the design iteration cycle is prolonged, ultimately extending the development time and lower cost for developing hardware designs. In contrast, software applications can reach large-scale deployment at an accelerated rate and cost, made possible due to the high-level abstractions modern programming languages provide to the developers.

The disparity between the software and hardware design process has prompted the development of high-level hardware abstraction languages targeting certain computing domains, colloquially termed domain-specific languages (DSL). The raising of the level of abstraction of hardware design allows users to implement their designs quickly due to being able to leverage the properties of high-level programming models. As a result, DSL languages allow the user to develop, optimize, verify, and reuse designs at the level higher than hardware description languages, thereby significantly shortening the development cycle and improving productivity.

In this work, we focus on the implementations of DSL that can use High-Level Synthesis (HLS) to generate hardware descriptions. However, existing design flows with HLS face several major challenges. First, there exists a representation problem of the hardware as HLS tools mainly rely on existing programming languages designed for sequential execution. Second, the current HLS input languages, such as C/C++/OpenCL, do not represent the recent advancement in the software development community. Most software applications nowadays are developed with languages like

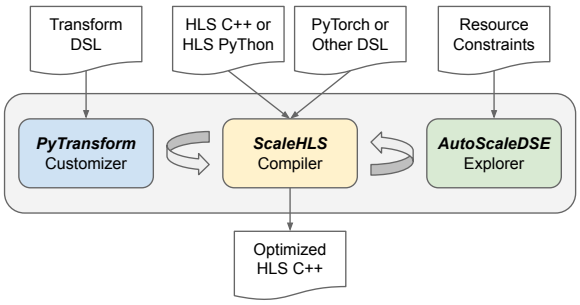

Figure 1: The proposed HLS framework for domain-specific computing.

Python, which is at a higher level of abstraction than existing hardware design flows. Third, the current HLS tools are mainly built for small-scale module-based designs. As a result, handling large-scale designs and searching for optimal designs in complex design spaces remain challenging.

Recently there has been a growing interest in the research community on high-level programming languages for FPGAs. HeteroCL [21] is one recent work that builds on the TVM framework [3]. HeteroCL is designed to be an API library of the Python Language where a runtime compiler is used to interpret the design expressed as API calls using the HeteroCL library to a C representation optimized for the Merlin HLS compiler [7]. Similarly, the PyLog [13] compiler interprets computation represented using the built-in Python syntax to generate optimized C code for Vivado HLS [14]. Dahlia [28] is another high-level programming language that compiles to HLS C code. Dahlia uses Scala-like syntax and a type system to enforce design constraints in the HLS code so that unrolling and memory partitioning factors match, with the addition of compiler directives to express the level of parallelism of the design.

In addition to the efforts in elevating the input abstraction level, we have also witnessed a large number of papers investigating the accurate and efficient quality of results (QoR) estimation methods and automating the design space exploration (DSE) for HLS. The authors of [40–42] proposed analysis-based approaches for estimating the latency, power, and resource utilization of HLS designs. Necessary design information is extracted from static dataflow graphs or dynamic execution traces and then passed to analytical models to generate the estimation. Apart from analysis-based approaches, [25, 29] introduced machine learning methods to extract features that cannot be parameterized by analytical models from the input HLS designs, thereby deducing more accurate estimation and handling more complicated designs. On top of QoR estimation, authors of [40, 41] proposed automatic DSE tools based on the guidance of resource utilization and performance metrics. More advanced techniques, including polyhedral analysis and integer linear programming (ILP) algorithms, are exploited in [43, 44] that propose qualified design candidates and search for the optimal solution under hardware resource constraints.

However, we argue that existing HLS frameworks lack a compilation system dedicated for domain-specific hardware accelerators. To address this problem, Section 3 presents a source-to-source compilation tool named ScaleHLS [37–39] that automates the tedious HLS optimization process. Then, we identify that the existing DSE algorithms are not scalable to handle large HLS accelerators. Therefore, Section 4 presents AutoScaleDSE [17] that we have done to address the automation of the design space exploration process. Finally, customized optimizations are essential for domain-specific accelerators to achieve high performance. In Section 5, we present a new and novel method named PyTransform for representing compiler optimizations in a high-level language that allows the user to configure custom compiler optimizations during the development of the application. Figure 1 shows how the three components above are integrated in our proposed domain-specific HLS framework. The framework can take HLS C++/Python or DSLs, such as PyTorch, as input designs and compile them to optimized HLS C++ through the ScaleHLS compiler. On the right side, resource constraints can be passed to the AutoScaleDSE explorer and automatically explore the design space of HLS designs in ScaleHLS. On the left side, we propose a transform DSL that can be passed to the PyTransform customizer to optimize HLS designs with user-defined patterns.

In the remainder of this paper, we first present a brief survey of current domain-specific languages that are used to accelerate the development of FPGAs in Section 2. Then, Sections 3, 4, and 5 present the details of ScaleHLS, AutoScaleDSE, and PyTransform, respectively. Section 6 discusses future works and in Section 7 we conclude this paper.

2 BACKGROUND

2.1 Domain-Specific Programming Languages

Developing a DSL that can accurately and flexibly capture the desired hardware description has long been a challenge that has seen much active research. As a result, a vast body of work exists that tries to solve this representation problem using various methods spanning many different programming languages and models. In this section, we briefly introduce the various DSL works with a focus on HLS.

2.1.1 PyLog. PyLog [13] presents a Python-based DSL for FPGAs using general Python-compatible syntax. By raising the abstraction level to Python from the comparatively low-level C/C++, users of PyLog can express their designs for both the host and device using the high-level operators that PyLog provides. Designs expressed with these high-level operators can be efficiently compiled by the PyLog compiler to highly optimized HLS designs, automating the process of exploring design space. The optimization process is aided by the PyLog intermediate representation, where computational patterns that can be parallelized are automatically identified by the PyLog compiler, transforming it into an efficient HLS C/C++ implementation. Furthermore, users of PyLog are able to express both the host and device code at the same level of abstraction, allowing a unified framework for the functional simulation of the design. In addition, the unified representation enables the user to co-design the host and device code, automating the host and device code generation using the PyLog compiler.

2.1.2 HeteroCL. HeteroCL [21] proposes a DSL that can efficiently represent applications targeting the CPU+FPGA platform. The programming model of HeteroCL uses a blend of declarative symbolic expressions based on the sequential Python language. Using this approach, HeteroCL expresses computations in a manner that exposes

high-level optimization opportunities. HeteroCL's strength lies in its ability to represent computations using symbolic expressions to decouple the algorithmic specification from the three hardware design dimensions: compute, data type, and memory architecture, with the added capability to capture the interdependence between these dimensions. Through HeteroCL, the user can efficiently experiment with various design choices and experiment with various degrees of parallelism, data types, and memory architecture due to the design choices being independent through the aforementioned decoupling. Internally HeteroCL extends the Halide IR used by TVM [3, 4] to express, optimize, and output designs for various target platforms and language frameworks such as HLS C/C++ [14–16], OpenCl [18], and Merlin C [7]. Furthermore, through symbolic representation, computation can be efficiently and effectively mapped to spatial architecture templates using the SODA [5] framework. This work is notable for its potential to raise the level of abstraction while not obfuscating the underlying hardware design.

2.1.3 FCUDA. FCUDA [8, 30, 31] presents a compilation flow that compiles an FPGA design expressed in the CUDA [24] programming model to RTL. FCUDA leverages CUDA's expressiveness, which captures a design's parallelism using CUDA-specific built-it variables and primitives. FCUDA, using CUDA code annotated with information regarding the desired resources of the target FPGA, transforms the input code into a High-Level Synthesis (HLS) compatible C representation. This intermediate C representation is then further compiled by an HLS tool, such as AutoPilotC (now part of Vitis) [15], into RTL code. The key contribution of FCUDA lies in the compiler's ability to transform CUDA code that expresses parallelism using built-in variables into an explicit C representation that expresses the execution of CUDA light-weight kernel threads in a repackaged coarse-grained multi-threading execution model on top of FPGAs. As a result, developers of FCUDA can leverage their existing CUDA knowledge and skills to program FPGAs, which can provide significant performance and productivity improvements over traditional FPGA programming methods.

2.2 HLS Compilation

Traditional HLS compilers typically consist of three compilation stages, scheduling, allocation, and binding, to compile HLS programs written in different languages into RTL accelerators. However, the large design space brought by HLS designs is known to be challenging to explore. In order to overcome this limitation of HLS, we have seen multiple HLS compilers proposed for exploring and optimizing HLS designs automatically. In this section, we will introduce several representative works in this direction.

2.2.1 Merlin. The Merlin compiler [7] is a source-source compilation flow that raises the level of abstraction from a hardware-oriented C/OpenCL representation to a software-oriented C/C++ representation. The Merlin compiler takes an algorithmic software description as input with minimal compiler directives that only tell which code portion to accelerate. Based on this description, the Merlin compiler can exploit fine-grained loop-level parallelism and pipelining and coarse-grained task-level parallelism and pipelining.

2.2.2 SODA-OPT. SODA-OPT [1] is another source-to-source compilation tool that automates the optimization process for HLS. The SODA-OPT extends the MLIR compiler infrastructure, developing SODA-OPT compiler passes and custom IR (intermediate representation). By representing the target design at various levels of abstraction, SODA-OPT can efficiently apply compiler optimization passes at the most optimal level of abstraction. Furthermore, the optimization passes that transform the code (used to increase the level of parallelism and memory bandwidth and optimize the order of operations) are applied directly to the compiler IR allowing SODA-OPT to target various different HLS tools. Notably, SODA-OPT can identify computational patterns suited for acceleration through pattern matching and separate the algorithmic input design into host and device codes. Following this separation, the SODA-OPT design space exploration engine automates the process of finding a suitable combination of compilation passes by interfacing with the SODA-OPT compiler passes and the HLS backend.

2.2.3 HPVM2FPGA. HPVM [19] is an explicitly parallel extension of LLVM IR for heterogeneous systems, designed to enable performance portability across different parallel hardware. On top of HPVM, HPVM2FPGA [9] adds an optimization framework that uses compiler optimizations and design space exploration (DSE) to automatically tune a hardware-agnostic program for a given FPGA. Notably, HPVM2FPGA supports host-device partitioning and host code generation at the IR level.

3 MULTI-LEVEL HLS COMPILATION

3.1 Motivation

Although DSLs have proven to be powerful tools in raising the level of abstraction of hardware design, accurate representation of the desired hardware has long been a challenge. Works such as PyLog [13], FCUDA [8, 30, 31], and HeteroCL [21] aim to solve this representation problem at a higher level of abstraction, above the mature and widely used C/C++ programming language/model. Although previous works show promise in their ability to capture the desired hardware at this higher level of abstraction, as a side-effect, this added abstraction further removes the user from the final hardware design. Thus, it increasingly becomes challenging to understand how a design decision made at a higher level of abstraction is translated into the low-level hardware design, having to go through multiple compiler optimization passes and compilation flows. In addition, achieving efficient and high-performing designs using a high-level DSL requires the user to have expertise in how the high-level design decisions impact the final design. Even then, due to the complex interplay between various compiler passes, implementing high-performing hardware design using a high-level DSL remains challenging. With this in mind, we developed ScaleHLS to automate the process of optimizing hardware designs written in Python (PyTorch [32]) and C/C++ by embracing the MLIR [22] framework and the community. Furthermore, the transformation process is designed to preserve the transparency of the compiler optimization passes from the algorithmic description to the HLS-compatible optimized C/C++ design.

3.2 ScaleHLS Compiler

3.2.1 Framework. We propose ScaleHLS [37–39], a compiler tool for HLS that aims to address the representation problem of DSL.

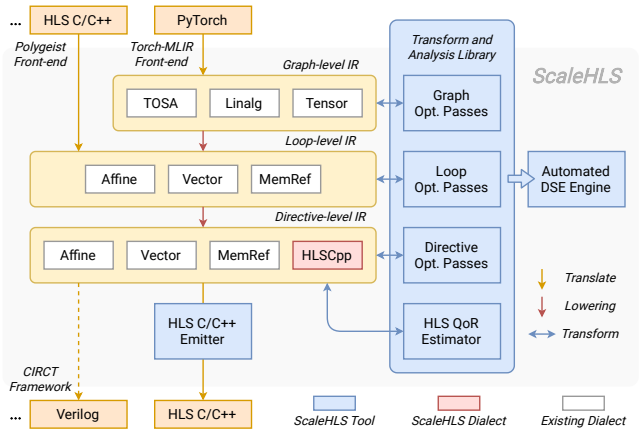

Figure 2: ScaleHLS framework.

ScaleHLS leverages the multi-level hierarchy of HLS designs and advanced compilation techniques to address the automation and scalability issues of existing HLS flows. Figure 2 shows the overall architecture of the ScaleHLS framework. ScaleHLS is built upon MLIR [22] and supports C/C++ and PyTorch programs as inputs through the Polygeist [27] and Torch-MLIR [2] 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 [22] to represent the tensor-level computation graph, where graph optimizations, such as node fusion and coarse-grained pipelining, can be performed efficiently. Dialects are used to express the code at a certain level of abstraction, allowing for the design of highly-efficient compiler passes. In MLIR, dialects refer to a set of operations, types, attributes, and other language abstractions used to represent computations and data and are the building blocks that constitute the intermediate representations of MLIR. At the middle level, ScaleHLS uses `affine`, `vector`, and `memref` (Memory Reference) dialects [22] 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.

3.2.2 Optimizations. ScaleHLS aims to enhance the HLS compilation process by providing a collection of classes and methods for analyzing and optimizing HLS designs and generating code. The library features three main components, the `Estimator`, the `Explorer`, and the `Emitter`, designed to construct and solve the QoR estimation, DSE, and HLS C++ emission problems, respectively. The `Estimator` is integrated into the `Explorer` for efficient design evaluation, and is structured in a hierarchical manner to avoid full estimation with each iteration. The `Emitter` is also hierarchical, enabling the compiler to have a more precise interaction with RTL generation tools.

ScaleHLS also features a set of optimization APIs organized into four categories: *Graph*, *Loop*, *Memory*, and *Directive*. The *Graph* optimizations aim to improve the `tosa` dialect and encompass simplification and operation fusion. *Loop* optimizations, based on the `affine` dialect libraries, include methods for loop perfection, variable bound loop removal, tiling, and unrolling to enhance data locality and computation parallelism. The *Memory* optimizations aim to boost the efficiency of memory access, given the scarcity of memory bandwidth in hardware accelerators. The *Directive* optimizations are built on top of our `hlscpp` dialect abstractions and include loop and function pipelining and array partition.

Table 1: Evaluation results of ScaleHLS on representative DNN models. *Speedup* is with respect to the baseline designs compiled from PyTorch by ScaleHLS but without the multi-level optimization.

Model	Speedup	Compile Time	Mem.	DSP	DSP Effi.	TVM [26] DSP Effi.
ResNet-18	3825.0×	60.8s	91.7Mb	1326	1.343	0.344
VGG-16	1505.3×	37.3s	46.7Mb	878	0.744	0.296
MobileNet	1509.0×	38.1s	79.4Mb	1774	0.791	0.468

3.2.3 Evaluation. We evaluate ScaleHLS's ability to handle large and complicated HLS designs using three representative DNN (deep neural networks), ResNet-18 [11], VGG-16[36], and MobileNet [12], targeting the CIFAR-10 [20] image classification task. These DNN models are constructed with a large number of different hidden layers and have sophisticated inter-layer dependencies. The target platform is one SLR (super logic region) of Xilinx VU9P FPGA, which is a large FPGA containing 115.3 Mb on-chip memory, 2280 DSPs, and 394,080 LUTs on each SLR. The PyTorch [32] implementations are parsed into ScaleHLS and optimized using the proposed multi-level optimization methodology. Graph, loop, and directive optimization passes are applied sequentially to improve the design quality at the corresponding IR level.

The experimental results are shown in Table 1. By combining three optimization levels, we can observe that the generated HLS designs achieve significant speedups ranging from 1505.3× to 3825.0× in terms of throughput compared to the baseline designs, which are compiled from PyTorch to HLS C/C++ through ScaleHLS but without the multi-level optimization. Notably, as shown in Table 1, ScaleHLS only consumes 37.3 to 60.8 seconds to optimize the large HLS designs, demonstrating the efficiency and scalability of our optimization methodology.

4 DESIGN SPACE EXPLORATION

4.1 Motivation

Fully automating the design space exploration process of HLS has long been a challenge. The difficulty lay in the DSE engine's ability to accurately and efficiently explore the vast design space HLS designs present and is currently the primary limiting factor of automating the DSE process. The final synthesized RTL design is *nearly* directly linked to how the design is represented using the high-level language. Therefore, depending on the structure and factor of loops, arrays, functions, and compiler directives significantly impact the final design's quality. Accordingly, the design space is proportional to the number of tunable knobs in the design space, where the code structure and the compiler directive constitute the

Table 2: QoR DSE results of computation kernels. *Speedup* is with respect to the baseline designs from PolyBench-C [33] without the optimization of DSE.

Kernel	Prob. Size	Speedup	Tiling Sizes	Pipeline II
GEMM	4096	768.1×	[8, 1, 16]	3
GESUMMV	4096	199.1×	[8, 16]	9
SYR2K	4096	384.0×	[8, 4, 4]	8
SYRK	4096	384.1×	[64, 1, 1]	3
TRMM	4096	590.9×	[4, 4, 32]	13

tunable knobs of the design space. The method for effective DSE may vary from brute force searching to more intelligent methods, incorporating different forms of supervised learning.

A recent survey of HLS DSE [35] reveals that due to the lack of compiler support and due to the immense design space, current DSE engines can only target the compiler directives and cannot optimize for the code structure. Moreover, due to existing DSE engines having to rely on the computationally and time-intensive HLS synthesis compiler to evaluate intermediate design points, DSE engines were limited to exploring only a handful of design points in the exploration process. Furthermore, we found it impossible to beat hand-tuned designs through compiler directives alone without code restructuring.

ScaleHLS was built with the desire to automate the design process of HLS, and automating the design space exploration process to explore the vast design space that includes code transformation was a natural next step in this line of thought. The compiler passes that we developed allowed us to explore the code optimization dimension of the design space. Furthermore, these passes allowed us to estimate intermediate design points' performance without invoking the computationally and time-intensive HLS synthesis compiler, enabling us to explore the design space to a greater degree.

4.1.1 QoR Design Space Exploration Engine. Downstream RTL generation tools like Vivado HLS [14], Vitis HLS [15], and Intel HLS [16] can take anywhere from a few minutes to several hours to report the synthesis results. Resulting in (1) restricting the number of design points assessed during Design Space Exploration and (2) significantly increasing the DSE time to up to tens of hours. Our Quality of Result (QoR) estimator aims to address this issue using the ALAP (as late as possible) algorithm to schedule each MLIR IR block of the design.

By utilizing the compiler passes and intermediate representation (IR) we developed, as well as additional heuristics that explore neighboring Pareto-optimal design points, we could explore thousands of design points in the order of minutes. Using this approach, we significantly increased the performance of single loop kernels summarized in Table 2. As a case study, for the GEMM kernel, our QoR-based DSE engine was notable in its ability to outperform designs where an experienced HLS user manually restructured code and systematically searched the compiler directive design space.

4.1.2 Limits of the QoR Design Space Exploration Engine. The result of Table 2 clearly demonstrates the QoR-based DSE engine's ability to produce high-quality solutions. Using this approach, we uncovered more efficient but unintuitive solutions that were nearly impossible to be deduced by the user. We theorized that these unintuitive but efficient solutions performed well because these solutions synergized with the heuristics the downstream HLS compiler used to schedule and bind the operations. However, as a result of these HLS heuristics and the exponentially growing design space of large-scale designs with multi-loops, the QoR-based DSE engine incurred significant scalability issues and degradation in the quality of the final solutions.

Table 3: QoR-Base vs AutoScaleDSE using the PolyBench [33] medium dataset.

	Implementation	Latency	Speedup	DSP	FF	LUT
BICG	Baseline	16 *ms*	1x	0%	0%	0%
	QoR-Base	42.050 *µs*	380x	50%	24%	37%
	AutoScaleDSE	42.050 *µs*	380x	50%	24%	37%
Correlation	Baseline	1.367 *s*	1x	0%	0%	1%
	QoR-Base	0.942 *s*	1.5x	15%	18%	51%
	AutoScaleDSE	15.467 *ms*	88x	4%	7%	27%
2MM	Baseline	1.542 *s*	1x	0%	0%	0%
	QoR-Base	8.281 *ms*	186x	55%	21%	46%
	AutoScaleDSE	1.722 *ms*	895x	76%	17%	46%
3MM	Baseline	2.054 *s*	1x	0%	0%	0%
	QoR-Base	72.930 *ms*	28x	35%	20%	33%
	AutoScaleDSE	1.996 *ms*	1029x	61%	17%	37%

Table 4: AutoScaleDSE evaluation results for MachSuite [34] and Rodinia [6] benchmarks.

	Implementation	Latency	Speedup	DSP	FF	LUT
Backprop	Baseline	754 *ms*	1x	24%	19%	32%
MachSuite	AutoScaleDSE	62.669 *µs*	12x	87%	41%	75%
Lud - tiled	Baseline	1.650 *s*	1x	1%	1%	3%
Rodinia	AutoScaleDSE	0.846 *µs*	2x	15%	21%	55%

An initial solution to this scalability problem would be dividing and conquering the whole design space. This approach aims to tame the exponential growth of the design space by exploring the design space for individual loops and finding the Pareto optimal point that corresponds to a specific tiling, loop parallelism, and memory bandwidth strategy. Afterward, the separate design spaces are combined to create a global design space to determine the final Pareto optimal design.

However, one major flaw in this approach is that the overall design cannot be divided into independent design spaces. As real-world HLS designs have multiple loops that share resources and communicate with each other over common memory elements, an optimal solution for one sub-design space can lead to a worse global solution due to incompatibility between sub-solutions. As a result, a combined global design point cannot simply be constructed from the sum of the optimization strategies and performance estimations for each sub-design.

4.2 AutoScaleDSE Explorer

As a solution to these problems, we present AutoScaleDSE [17], a scalable DSE engine capable of finding the Pareto optimal solution under resource constraints for large real-world HLS designs. Our design space exploration engine aimed to balance the time spent on design space exploration and HLS compilation. By identifying a key

```
1  def montgomery_redc(A, M, v, k, z):
2      C = []
3      for i in range(0, len(A)):
4          pytransform.require(lambda v: v == -1, v)
5          S = z + 1
6          pytransform.require(lambda S, k: S == 1 << k, S, k)
7
8          a = A[i]
9          s = (a * v) & z
10         r = (a + s * M) >> k
11         if r < M:
12             C.append(r)
13         else:
14             C.append(r - M)
15     return C
```

Listing 1: A Montgomery reduction example in Python.

limitation in accurately estimating the intermediate DSE results without invoking the downstream HLS compiler, we were able to decrease the DSE runtime to a couple of hours while also being able to evaluate thousands of design points.

The key contributions of this approach stem from the use of a random forest classifier and a code-aware graph-driven approach. For large-scale HLS designs, the traditional approach of stitching together sub-solutions was not viable due to the incompatibility of sub-solutions having an adverse effect on the final solution. However, having recognized that a good amount of sub-solutions were compatible, we used a random forest classifier to merge sub-solutions (generated using the QoR estimator-based DSE) that were compatible. We theorized that the compatibility of sub-solutions was due to the heuristics of the downstream HLS compiler and aimed to predict the compatibility of sub-solutions without invoking the HLS compiler using the random forest classifier. Furthermore, a lexical analyzer was developed to extract essential information regarding the design to address the characteristics of large HLS designs consisting of multiple functions and loops, of which not all share resources. Based on this information, a graph representation of the HLS design was constructed and used to guide the DSE process, intelligently merging sub-solutions, giving weight to more tightly correlated sub-solutions that share resources.

We evaluated the quality of this new approach using four PolyBench [33] benchmarks with the addition of two large-scale benchmarks from the MachSuite [34] and Rodinia [6] benchmark sets. The presented results in Table 3 and Table 4 have been collected from the report generated by Vivado HLS 2019.2 targeting the 'xc7z045-ffg900-2' device. For the experiments in Table 3 compared to the QoR-based DSE, AutoScaleDSE outperformed the previous approach in terms of latency and resource utilization by up to 59X. Furthermore, for the experiments in Table 4, we demonstrate AutoScaleDSE's ability to explore previously non-viable large designs, being able to decrease the latency of designs by up to 12X.

5 PATTERN-DRIVEN DESIGN OPTIMIZATION

5.1 Motivation

Although existing HLS frameworks have exposed common HLS optimizations, such as loop unrolling and array partitioning, domain-specific computing often demands different customized optimizations to improve the efficiency of certain computation patterns and fine-tune the hardware performance. For example, Listing 1 shows a

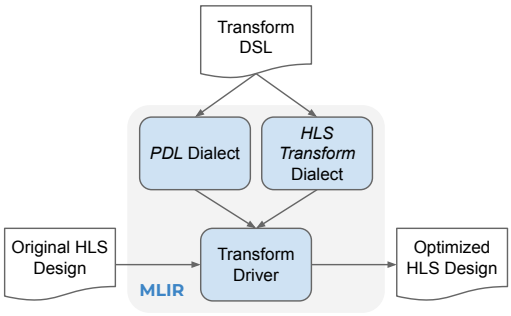

Figure 3: Architecture of PyTransform flow.

Montgomery reduction implemented in Python. Montgomery algorithm is a method for performing modular arithmetics, particularly those involving large integers. The algorithm allows for efficient modular multiplication, which is used in many cryptographic algorithms such as the RSA algorithm. As shown in Listing 1, as long as the pre-requirements at line 4 and 6 are met, the expression at line 9 can be rewritten to a new form with the following equations:

$$\begin{aligned} s &= (a * v)\ \&\ z = (a * -1)\ \%\ S \\ &= S - (a\ \%\ S) = S - (a\ \&\ z) \end{aligned} \tag{1}$$

where a is a number in the normal or Montgomery space, v and z are twiddle factors, and S is equal to $z + 1$. We can observe that after the rewriting, the original resource-consuming multiplication operator is replaced with much cheaper operators. Such customized optimizations often only have impacts in specific domains but are essential for generating efficient accelerators of these domains. However, general-purpose HLS frameworks or libraries are not possible for covering these customized optimizations of different domains beforehand. As a result, to apply these optimizations, one way is manually rewriting the HLS code into new forms, which is apparently not scalable for large HLS designs with hundreds of applicable optimizations. Another more scalable way is extending existing HLS frameworks or libraries with new customized optimization passes to automatically transform the HLS designs. However, this way requires the user to implement compiler passes with low-level languages, such as C++, thus heavily relies on compiler expertise. Therefore, this approach offers low accessibility to hardware designers and also runs counter to the philosophy of HLS that aims to raise the abstraction level of hardware design.

5.2 PyTransform Customizer

5.2.1 Overview of the Flow. In order to bridge this gap and provide a scalable and high-level design methodology for the optimization of domain-specific accelerators, we introduce a pattern-driven design optimization flow called PyTransform. Figure 3 shows the architecture of the PyTransform flow. A Python-based HLS transform DSL is first introduced to describe 1) the patterns to be matched and optimized, and 2) the rewriting rules to be applied. The patterns and rewriting rules are then parsed into an existing *PDL* (Pattern Descriptor Language) dialect and a new HLS *Transform* dialect, which is an extension of the ScaleHLS dialects. Finally, the two dialects are used to guide a transform driver to automatically traverse the IRs parsed from the original HLS design, match and rewrite

```
1  @pytransform.is_pattern(benefit=1)
2  def pattern_expr():
3      dtype = pytransform.Int(32)
4      v = pytransform.value(dtype)
5      pytransform.require_pattern(lambda v: v == -1, v)
6
7      z = pytransform.value(dtype)
8      S = z + 1
9      k = pytransform.value(dtype)
10     pytransform.require_pattern(lambda S, k: S == 1 << k, S, k)
11
12     a = pytransform.value(dtype)
13     s = (a * v) & z
14     expr_rewrite(s, S, a, z)
15
16 @pytransform.is_rewrite
17 def expr_rewrite(s, S, a, z):
18     new_s = S - (a & z)
19     pytransform.replace(s, new_s)
```

Listing 2: A expression rewriting pattern in Python.

the patterns, and generate the optimized HLS design. Both the IRs and transform driver are implemented within the MLIR framework. With such a design methodology, once a pattern rewriting rule is described with our python-based DSL, the corresponding optimization can be applied to any HLS designs without any manual effort. Meanwhile, the DSL is designed with flexibility and scalability in mind and can provide high-level interfaces for hardware designers to describe the desired optimization patterns for domain-specific accelerators. The remainder of this section will use two examples to demonstrate the key features and advantages of our DSL.

5.2.2 Expression Rewriting. In Listing 2, an expression rewriting pattern, `pattern_expr`, is defined as a Python function annotated with an `@is_pattern` decorator. In this pattern, we first attempt to match an int32 value `v` that equals to -1 at line 3-5. Similarly, we then attempt to match another value `z` at line 7. Then, at line 8, we use an expression of the matched value `z` to match another value `S` that is equal to `z + 1`. One can observe that in the pattern definition function of our DSL, expressions between matched values are interpreted as a pattern to match new values. Once we have successfully matched values `a`, `v`, and `z`, we can attempt to match the target expression at line 13. Finally, at line 14, we pass the matched values to a rewriting function, `expr_rewrite`, which is defined at line 17. In this rewriting function, we construct a new expression with the matched values, `S`, `a`, and `z`, and replace the original expression with the new one at line 18-19. One can observe that different with the pattern definition functions, the expressions inside of rewriting functions are used for the construction of new IRs. Note that in our DSL, the pattern definition function should always be terminated with a function call to a rewriting function.

5.2.3 Loop Rewriting. Apart from expression matching and rewriting, our DSL also supports more advanced transformations such as loop unrolling, etc. In Listing 3, we first attempt to match three values, `S`, `a`, and `z`, and an expression `S - (a & z)` at line 3-7 similar to what we did in Listing 2. However, instead of rewriting the matched expression into a new form, we first obtain the parent loop enclosing the expression at line 12. Then, we split the matched loop into two loops, `outer` and `inner`. Finally, we unroll and pipeline the inner and outer loop, respectively, at line 14 and 15. The `unroll` and `pipeline` primitives are implemented as building blocks of our

```
1  @pytransform.is_pattern(benefit=1)
2  def pattern_loop():
3      dtype = pytransform.Int(32)
4      S = pytransform.value(dtype)
5      a = pytransform.value(dtype)
6      z = pytransform.value(dtype)
7      s = S - (a & z)
8      loop_rewrite(s)
9
10 @pytransform.is_transform
11 def loop_rewrite(s):
12     loop = pytransform.parent_loop(s)
13     outer, inner = pytransform.split(loop, factor=2)
14     pytransform.unroll(inner, factor=2)
15     pytransform.pipeline(outer, initial_interval=1)
```

Listing 3: A loop rewriting pattern in Python.

Table 5: PyTransform result of Montgomery reduction.

Implementation	Latency	Latency Compare	DSP	DSP Compare
Original	262	1.00×	192	1.00×
PyTransform	265	1.01×	130	0.68×

DSL that can be combined flexibly at a high level to define various rewriting rules.

5.2.4 Evaluation. We evaluated the performance of the Montgomery reduction example shown in Listing 1 with Vitis HLS [15]. Table 5 shows the evaluation results. The original design is compiled without the optimization of PyTransform, while the PyTransform design is optimized by a set of rewriting patterns defined in our proposed DSL. One can observe that the PyTransform design maintained a similar latency while only utilizing 0.68× of DSP resources compared to the original design, demonstrating the effectiveness of our PyTransform customizer.

5.2.5 Discussion. Our pattern-driven transform DSL can precisely capture the optimization opportunities in domain-specific accelerators and conduct the optimizations in a scalable way. On one hand, compared to manual code rewriting, our DSL can describe the rewriting pattern in a structured manner and once a pattern is implemented in our DSL, the rewriting can be applied automatically with no human intervention. On the other hand, compared to the customized compiler passes, our DSL significantly reduces the difficulty of implementing an HLS optimization and allows hardware designers without compiler expertise to customize their HLS design. Note that our DSL still has some building blocks being implemented in C++ and hidden from the users, such as the loop `unroll` and `pipeline`. This is designed intentionally because we want to abstract away the low-level programming details and strike the balance between flexibility and complexity of programming. By integrating PyTransform into the ScaleHLS compiler, users are able to customize new optimizations conveniently and combine with the existing optimizations provided by ScaleHLS to generate customized domain-specific accelerator designs.

6 FUTURE WORKS

This work proposed ScaleHLS, AutoScaleDSE, and PyTranform to tackle the challenges present in the compilation, design space

exploration, and customized optimization of domain-specific HLS designs. This work still has several research directions remaining for future works:

6.1 Optimization of dataflow architecture. ScaleHLS proposed a primary support for automatically generating and legalizing dataflow architecture in HLS designs. However, ScaleHLS leveraged the conventional function call graph to represent the hierarchical dataflow structure, where each function call is corresponding to a dataflow stage. However, the function call graph is originally designed for software compilation. Due to the lack of expressiveness on hardware properties, such representation is not suitable for serving the flexible construction and low-level optimization of HLS dataflow structures. As a result, a systematic framework for optimizing the hierarchical dataflow structures in domain-specific accelerators are desired in the future.

6.2 Optimization of external memory access. With more and more data involved in the computation of different domains and the scarce bandwidth of external memory, the efficiency of external memory access, including DDR and HBM, becomes more important than ever in domain-specific accelerators. Through static analysis on the HLS IR, the efficiency of external memory access could be optimized through: 1) Vectorization, which can bundle multiple contiguous memory accesses into a single one and increase the bitwidth of the memory interface. 2) Tiling and scratchpad generation, which can effectively improve the data locality and expose more opportunities of data reuse. 3) Memory layout permutation, which can accommodate the tiled/sliced memory access pattern and increase the burst length of the memory interfaces.

6.3 Dataflow-aware design space exploration. AutoScaleDSE proposed a scalable DSE solution to find the Pareto-dominating HLS design. Although AutoScaleDSE has considered the connectedness between different loop nests during the exploration, the efficiency of dataflow execution is not thoroughly considered yet. To enable the dataflow-aware solution, at least two levels of design space are envisioned to be explored properly: 1) At the high level, different node fusion and decomposition may result in different trade-offs of performance and resource utilization. 2) At the low level, the latency balancing of dataflow nodes can significantly impact the performance, thus demands wise selection of node parallelization and buffer partition strategies.

6.4 Verifiable HLS. Apart from the generation of highly-optimized hardware designs, another critical challenge is to ensure the correctness of the high-level design flow in a scalable manner that can effectively deal with the increased design complexity. Due to the complicated functionality and hardware hierarchy, formal properties for verification are difficult to establish, while the complexity of proving correctness restricts the scalability of the verification procedure. As a result, a need for a more coherent multi-level IR has recently arisen to overcome these challenges. The new IR will be more advantageous for representing the hardware designs of various scales and design verification semantics, which can deliver highly-optimized design solutions and alternatives while achieving provably correct and scalable verification of high-level designs.

6.5 Low-latency domain-specific applications. In the recent decade, AI and ML algorithms have shown remarkable capabilities in making sense of extensive datasets. Following this trend, many argue and advocate using these algorithms in the pipeline of various domain-specific research. For example, in the field of High-Energy Particle Physics (HE), the Large Hadron Collider (LHC) is expected to experience an increase in data rates in the order of petabits per second following the upgrades in 2025 [**?**]. This far exceeds any storage and analysis capacity of any device and, thus, creates a need for a computing platform that can process these large data streams in real-time within a small time frame. Similarly, the domain of Multi-Messenger Astrophysics (MMA) studies astronomical events using multiple channels of observation, which include channels of photons, gravitational waves, and neutrinos. Thus, to have the best chance of capturing transient astronomical events using multiple channels, the detection of an event at any one of the channels must be as fast as possible. As for the neuroscience domain, there is a need for low-latency computing platforms that can facilitate the timely detection and recognition of brain states [10, 23]. This small selection of domain-specific research fields has the commonality of the workload being sensitive to latency, and FPGAs are well-positioned to meet this demand. With this in mind, we are working towards expanding our framework, focusing on the three domains HE, MMA, and neuroscience as part of the ongoing research at the A3D3 center. We expect that in the future, domain-specific researchers will be able to implement and deploy AI/ML workloads tailored to their needs on an FPGA programmed using DSLs of their choice.

7 CONCLUSION

In this paper, we explore the current state-of-the-art works in domain-specific languages designed to raise the level of abstraction for FPGAs. In the process, we identified key challenges in developing a suitable domain-specific language and framework proposing solutions for each. ScaleHLS addresses the representation problem for HLS, presenting compiler intermediate representations that the compiler can leverage to optimize the algorithmic description of the design. AutoScaleDSE further builds upon the compiler infrastructure to develop a scalable design space exploration engine that can systematically and efficiently explore the vast design space of HLS designs. Finally, PyTransform proposes a systematic methodology that allows the user to express compiler optimization strategies at a higher programming level. In conclusion, we present an arsenal of tools that FPGA developers can leverage to significantly accelerate the process of implementing design using domain-specific languages.

ACKNOWLEDGMENTS

We thank Jeremy Casas and Zhenkun Yang of Intel for the insightful discussions. This work is supported in part by NSF 2117997 grant through the A3D3 center and SRC 2023-CT-3175 grant.

REFERENCES

[1] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. 2022. An MLIR-Based Compiler Flow for System-Level Design and

Hardware Acceleration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 9 pages. https://doi.org/10.1145/3508352.3549424

[2] Torch-MLIR Authors. 2022. Torch-MLIR. https://github.com/llvm/torch-mlir.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[5] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with Optimized Dataflow Architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240850

[6] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding Performance Differences of FPGAs and GPUs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 93–96. https://doi.org/10.1109/FCCM.2018.00023

[7] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. 2016. Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers: Invited Paper. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design* (San Francisco Airport, CA, USA) *(ISLPED '16)*. Association for Computing Machinery, New York, NY, USA, 154–155. https://doi.org/10.1145/2934583.2953984

[8] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[9] Adel Ejjeh, Leon Medvinsky, Aaron Councilman, Hemang Nehra, Suraj Sharma, Vikram Adve, Luigi Nardi, Eriko Nurvitadhi, and Rob A Rutenbar. 2022. HPVM2FPGA: Enabling true hardware-agnostic FPGA programming. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–10.

[10] Charles Gilbert and Mariano Sigman. 2007. Brain States: Top-Down Influences in Sensory Processing. *Neuron* 54 (2007), 677–696.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[12] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[13] Sitao Huang et al. 2021. Pylog: An algorithm-centric python-based FPGA programming and synthesis flow. *IEEE Trans. Comput.* 70, 12 (2021), 2015–2028.

[14] Xilinx Inc. 2020. *Vivado High-Level Synthesis User Guide UG902 (v2020.1)*.

[15] Xilinx Inc. 2022. *Vitis High-Level Synthesis User Guide UG1399 (v2022.1)*.

[16] Intel. 2023. Intel High Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[17] Hyegang Jun, Hanchen Ye, Hyunmin Jeong, and Deming Chen. 2022. AutoScaleDSE: A Scalable Design Space Exploration Engine for High-Level Synthesis. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* (2022).

[18] Khronos. 2023. OPEN STANDARD FOR PARALLEL PROGRAMMING OF HETEROGENEOUS SYSTEMS. https://www.khronos.org/opencl/.

[19] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. Hpvm: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 68–80.

[20] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[21] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. https://doi.org/10.1145/3289602.3293910

[22] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *arXiv preprint arXiv:2002.11054* (2020).

[23] Seung-Hee Lee and Yang Dan. 2012. Neuromodulation of Brain States. *Neuron* 76 (2012), 209–222.

[24] David Luebke. 2008. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 836–838. https://doi.org/10.1109/ISBI.2008.4541126

[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)*. IEEE, 397–403.

[26] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: an open hardware-software stack for deep learning. *arXiv preprint arXiv:1807.04188* (2018).

[27] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 45–59.

[28] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[29] Kenneth O'Neal, Mitch Liu, Hans Tang, Amin Kalantar, Kennen DeRenard, and Philip Brisk. 2018. Hlspredict: Cross platform performance prediction for fpga high-level synthesis. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[30] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *2009 IEEE 7th Symposium on Application Specific Processors*. 35–42. https://doi.org/10.1109/SASP.2009.5226333

[31] Alexandros Papakonstantinou, Yun Liang, John A Stratton, Karthik Gururaj, Deming Chen, Wen-Mei W Hwu, and Jason Cong. 2011. Multilevel granularity parallelism synthesis on FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 178–185.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[33] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012), 1–1.

[34] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 110–119. https://doi.org/10.1109/IISWC.2014.6983050

[35] Benjamin Carrion Schafer and Zi Wang. 2020. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2628–2639. https://doi.org/10.1109/TCAD.2019.2943570

[36] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[37] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 741–755.

[38] Hanchen Ye, Cong Hao, Hyunmin Jeong, Jack Huang, and Deming Chen. 2021. ScaleHLS: Achieving scalable high-level synthesis through MLIR. In *Proceedings of the Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'21)*.

[39] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1355–1358.

[40] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 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.

[41] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. 2016. Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[42] Wei Zuo, Warren Kemmerer, Jong Bin Lim, Louis-Noël Pouchet, Andrey Ayupov, Taemin Kim, Kyungtae Han, and Deming Chen. 2015. A polyhedral-based SystemC modeling and generation framework for effective low-power design space exploration. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 357–364.

[43] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. 9–18.

[44] Wei Zuo, Louis-Noel Pouchet, Andrey Ayupov, Taemin Kim, Chung-Wei Lin, Shinichi Shiraishi, and Deming Chen. 2017. Accurate high-level modeling and automated hardware/software co-design for effective SoC design space exploration. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.