ScaleHLS: Scalable High-Level Synthesis through MLIR

Hanchen Ye[1], Cong Hao[2], Jianyi Cheng[3], Hyunmin Jeong[1], Jack Huang[1], Stephen Neuendorffer[4], Deming Chen[1]
[1]University of Illinois at Urbana-Champaign, [2]Georgia Institute of Technology, [3]Imperial College London, [4]Xilinx Inc.
{hanchen8, hyunmin2, jackh4, dchen}@illinois.edu, callie.hao@ece.gatech.edu, jianyi.cheng17@imperial.ac.uk, stephenn@xilinx.com

ABSTRACT

High-level Synthesis (HLS) has been widely adopted as it significantly improves the hardware design productivity and enables efficient design space exploration (DSE). HLS tools can be used to deliver solutions for many different kinds of design problems, which are often better solved with different levels of abstraction. While existing HLS tools are built using compiler infrastructures largely based on a single-level abstraction (e.g., LLVM), we propose *ScaleHLS*[1], a next-generation HLS compilation flow, on top of a multi-level compiler infrastructure called MLIR, for the first time. By using an intermediate representation (IR) that can be better tuned to particular algorithms at different representation levels, we are able to build this new HLS tool that is more scalable and customizable towards various applications coming with intrinsic structural or functional hierarchies. ScaleHLS is able to represent and optimize HLS designs at multiple levels of abstraction and provides an HLS-dedicated transform and analysis library to solve the optimization problems at the suitable representation levels. On top of the library, we also build an automated DSE engine to explore the multi-dimensional design space efficiently. In addition, we develop an HLS C front-end and a C/C++ emission back-end to translate HLS designs into/from MLIR for enabling the end-to-end ScaleHLS flow. Experimental results show that, comparing to the baseline designs only optimized by Xilinx Vivado HLS, ScaleHLS improves the performances with amazing quality-of-results – up to 768.1× better on computation kernel level programs and up to 3825.0× better on neural network models.

1 INTRODUCTION

High-level synthesis (HLS) automatically translates high-level languages into dedicated hardware accelerators, thereby removing the reliance of the cumbersome and potentially error-prone hardware design practices that use dedicated hardware description languages [21, 41]. In recent years, HLS has been widely used in many application developments, such as neural networks [4, 51], IoT applications [3, 52], and video processing [29]. Existing algorithmic HLS tools typically focus on extracting parallelism from algorithmic descriptions and compiling the result into a parallel hardware execution model [36, 37]. Thus, HLS tools would enable a designer to implement different algorithmic choices quickly, identify high-level area-performance tradeoffs, avoid premature optimizations [6], and achieve working designs faster. While some of these alternatives can be explored automatically, it is also true that large-scale designs often make it very challenging to comprehensively explore the resulting large design space and produce high-quality design solutions [43]. As a result, existing HLS tools often provide user-specified directives to control or guide the HLS process to generate different micro-architectures, which means the tools would rely on designers for writing 'good' code and setting 'good' compiler directives in order to achieve good design quality [46].

In recent years, we have witnessed many studies for investigating different design space exploration (DSE) methods of setting HLS directives [43]. These efforts can be classified into two main types of methods: synthesis-based and model-based. Synthesis-based methods [5, 14, 42, 46] invoke downstream HLS tools to evaluate the quality of result (QoR) of discovered design points. Model-based methods [48, 54–57] instead extract necessary design information from static dataflow graphs or dynamic execution traces and pass such information to predefined analytical models for estimating the QoR without invoking HLS tools. Recently, machine learning methods are also investigated [12, 30, 35, 49] to extract unique features that cannot be easily characterized by analytical models and deduce estimations for more complicated designs. Once performance and resource utilization estimates can be determined, the DSE process can be regularized and solved through simulated annealing [42], integer linear programming [57], or other dedicated heuristics [14, 46], etc. Apart from different DSE methods, some other studies [27, 36, 44] leverage parallel-programming languages, such as CUDA [34], as inputs to expose the parallelism of the accelerator designs and generate synthesizable C code with HLS directives inserted.

However, we find that existing efforts and solutions face significant difficulty to handle large-scale HLS designs containing a large number of sub-modules and sophisticated inter-dependencies. The challenges mainly come from three aspects:

Representation. Existing works exploit C/C++ abstract syntax tree (AST) [23], traditional software compiler intermediate representation (IR) [24], or C/C++ source-level IR [13, 28], to represent and analyze HLS designs. These representations are originally designed for software compilation and only contain a single operation-level abstraction. However, HLS optimizations can often be carried out at or across different levels of abstraction for better results. For example, task/module level parallelization should be applied on high-level operators, such as convolution operators, rather than nested loops to avoid conservative assumption and sophisticated memory dependency analysis. Directly combining different levels of representation from different frameworks could cause significant fragmentation and cumbersome and inconsistent cross-level optimizations. We argue that we should have a systematic approach to represent HLS designs at multiple abstraction levels in order to honor the intrinsic hierarchies of HLS designs. This representation should act as the foundation of HLS optimization and address the various fragmentation and inconsistency issues that we are facing.

[1]ScaleHLS is open-sourced at https://github.com/hanchenye/scalehls.

Optimization. Existing works leave many important HLS optimizations, such as task/module level resource-sharing and parallelization, hardware IP integration, and loop level analysis and transformation, to human designers done by manual code rewriting. Such an approach is not productive and scalable enough to deal with large HLS designs and may obstruct the comprehensive exploration of the HLS design space. We argue that HLS optimizations should be fully automated and parameterized rather than relying on manual code rewriting. These optimizations should be carried out at multiple different abstraction levels automatically to reduce the complexity of program analysis and make the compilation flow more scalable to large HLS designs.

Exploration. In the domain of compiler development, the parameters of each optimization technique are typically determined by a *cost model* indicating the 'benefit' of the combination of such parameters. However, in HLS designs, because the effects of different HLS optimizations correlate (and sometimes in conflict) with one another, we cannot calculate the 'benefit' of one optimization in isolation of the other optimizations. In order to solve this problem, a global DSE engine is desired to take all HLS optimizations across different levels of abstraction into consideration and explore the large design space effectively.

In this paper, we propose a new tool, named as *ScaleHLS*, to tackle the challenges present in the representation, optimization, and exploration of HLS designs. ScaleHLS represents HLS designs with a multi-level IR for the first time, solves HLS optimization problems at the right levels of abstraction, and automates such optimizations through a new end-to-end flow without relying on manual code rewriting. ScaleHLS can optimize large HLS designs and still deliver high QoR for FPGA hardware implementation. We summarize the main contributions of our work as follows.

- To the best of our knowledge, ScaleHLS is the first end-to-end automated HLS compilation flow built on top of multiple levels of design abstraction naturally honoring intrinsic structural or functional hierarchies of large-scale designs.
- ScaleHLS proposes a hierarchical and scalable HLS representation and optimization methodology, which optimizes HLS designs at graph, loop, and directive levels holistically, to handle the complexity of the increasing HLS design space.
- ScaleHLS provides a transform and analysis library dedicated for HLS designs. This library turns a set of HLS optimization techniques from manual code rewriting to callable and tunable interfaces, saves significant amount of human effort and establishes the foundation of automated DSE.
- ScaleHLS contains a novel automated DSE engine to search for the Pareto frontier of the latency-area tradeoff space. A QoR estimator is also developed to evaluate design points discovered by the DSE engine rapidly.
- ScaleHLS expands the MLIR framework by adding an HLS C front-end and a synthesizable HLS C/C++ emission back-end for bridging the gap between the MLIR compilation framework and C-based HLS designs, thus enabling an end-to-end HLS compilation flow.

The remaining of this paper is organized as follows. Section 2 introduces the background. In Section 3, we provide an overview of the ScaleHLS framework. In Section 4 and 5, we introduce the details of the multi-level representation and optimization for HLS designs, respectively. In Section 6, we present the front-end and back-end integration of ScaleHLS. In Section 7 and 8, we provide the evaluation results and conclude this paper.

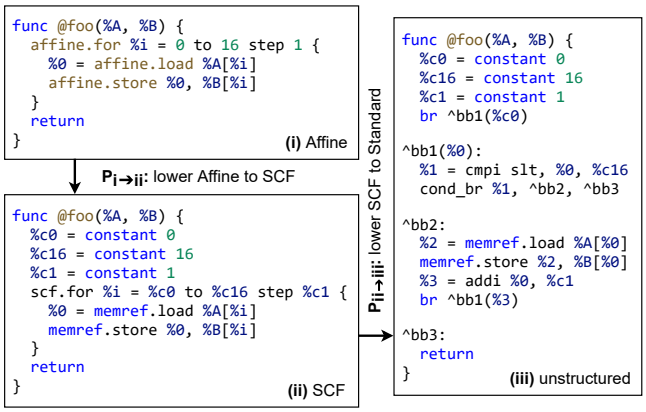


Figure 1: An IR example, where `affine` and `scf` dialect can represent structured control flow. `affine` dialect can be lowered to `scf` and then lowered to unstructured IR. All types are omitted for simplicity.

2 BACKGROUND

2.1 MLIR Framework

ScaleHLS is built on top of MLIR [8, 25], a compilation framework supporting multiple levels of functional and representational hierarchy. In the remainder of this paper, we use *MLIR* to refer to the MLIR compilation framework and *IR* for the intermediate representation of programs in MLIR. MLIR includes a single static assignment (SSA) style IR [11] where an *Operation* is the minimal unit of code. Each operation accepts a set of typed *Operands* and produces a set of typed *Results*. Connections between the result of one operation and the operands of another operation describe the SSA-style flow of data where every operand is connected to exactly one result. Each operation can also be parameterized by a set of *Attributes* indicating important characteristics of the operation. Unlike operands, which typically model values produced by other operations when a program is executed, attributes have values that are known and fixed at compile time. A sequential list of operations without control flow is defined as a *Block* (or *Basic Block*) and a control flow graph (CFG) of blocks is organized into a *Region* in MLIR. Regions are, in turn, contained by operations, enabling the description of arbitrary design hierarchy. For instance, a *Function* is defined as a built-in callable operation containing one region. Figure 1(i) shows an IR example, where the top function contains a region with two operations. The `affine.for` operation contains another region as the loop body.

A *Dialect* in MLIR defines a namespace for a group of related operations, attributes, and types. MLIR not only provides multiple built-in dialects to represent common functionalities, but also features an open infrastructure allowing users to define new dialects. *Pass* is a key component of compiler which traverses the IR for the purpose of optimization or analysis and is typically invoked through command line tools. Similar to LLVM, users can design

Transform and *Analysis* passes in MLIR to perform the IR transformation and analysis, respectively. However, in the context of MLIR, *Transform* typically refers to the transformation within a dialect. The transformation between different dialects is typically referred as *Conversion*, while the transformation between MLIR and external representation is referred as *Translation*. *Lowering* is a terminology referring to the process of lowering the abstraction level of the IR. Notably, MLIR provides powerful utilities for conducting common IR optimizations (e.g., loop transformation) and analyses (e.g., dependency analysis).

2.2 Relevant MLIR Dialects

Many dialects in MLIR are immediately applicable for representing nested loop programs commonly used in HLS. The `affine` dialect provides a powerful abstraction for affine operations and structures in order to use techniques from polyhedral compilation to make dependence analysis and loop transformations efficient and reliable. The `affine` dialect defines two kinds of affine SSA values, *Symbol* and *Dimension*, and *Affine Map* is defined as a mathematical function that transforms a list of affine values into a list of results with affine expressions. Affine operations (e.g., `affine.for`, `if`, and `apply`) must take affine values as input operands, therefore the loop bounds of `affine.for` operation and conditions of `affine.if` operation must be the expression of affine values. The `scf` (structured control flow) dialect defines control flow operations (e.g., `scf.for` and `if`) whose loop bounds or conditions can be any SSA values. Therefore, `scf` operations are not constrained by the affine requirements and can represent a wider range of programs. MLIR also provides several fundamental built-in dialects to represent unstructured control flow operations (e.g., `br`, `cond_br`, `call`, and `return`), basic arithmetic operations (e.g., `math.log` and `math.exp`), and non-affine memory access operations (e.g., `memref.load` and `memref.store`). Taking Figure 1 as example, the structured control flows in Figure 1(i) and (ii) represented with `affine` and `scf` operations are flattened to the unstructured `br` and `cond_br` operations in Figure 1(iii).

2.3 Relevant MLIR Front-ends

ScaleHLS takes advantage of third-party front-ends, NPComp [9] and ONNX-MLIR [26], to parse PyTorch [38] and ONNX [10] models, respectively. NPComp aims to compile numerical python programs into MLIR. NPComp first translates PyTorch models into `aten` dialect, then lowers the IR to `tcf` (tensor compute front-end) dialect and generates `affine` dialect as the end of compilation. ONNX-MLIR defines a subset of ONNX operations in an `onnx` dialect for translating ONNX models into MLIR. The `onnx` operations are then lowered to `krnl` (kernel) dialect and finally lowered to `affine` dialect by the ONNX-MLIR compilation flow.

ScaleHLS also includes its own C front-end for MLIR. The existing C front-end Polygeist [33] requires the users to manually identify the affine region in C using `scop` pragmas, while our approach can take arbitrary C code and automatically identifies the affine region in MLIR. Also, most HLS applications contain partially affine loops. Their approach is an all-or-nothing process, which means that a non-affine statement in a given region can cause failure in translating the whole region to the `affine` dialect, while our approach supports more precise granularity and translates other affine statements in this region into `affine` operations.

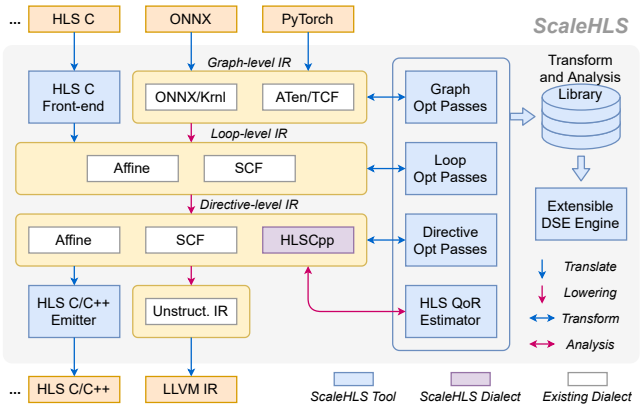


Figure 2: ScaleHLS framework.

3 SCALEHLS FRAMEWORK OVERVIEW

ScaleHLS compiles programs described in HLS C code or high-level programming frameworks, such as PyTorch, to optimized and synthesizable HLS C/C++ designs. Figure 2 shows the architecture of ScaleHLS. In this section, we organize the main components of ScaleHLS into four categories according to the challenge they are tackling (representation, optimization, exploration, and integration) and introduce them one by one.

3.1 Representation

Graph-level IR. We adopt existing third-party `aten` [9] dialect and `onnx` [26] dialect as the graph-level IR. The programs in high-level programming frameworks are parsed into computation graphs constructed with tensor operations of these dialects, on which graph optimizations can be conveniently applied. Details are discussed in Section 4.1.

Loop-level IR. We adopt MLIR built-in `affine` and `scf` dialects as the loop-level IR, which enables ScaleHLS to leverage the powerful transformation and analysis libraries provided by MLIR to conduct loop-level optimizations. The `affine` and `scf` operations can be lowered from the graph level IR or generated from the HLS C front-end. Details are discussed in Section 4.2.

Directive-level IR. We have designed an `hlscpp` dialect for representing the HLS-specific structures and program directives (e.g., loop pipelining). The customized dialect not only provides the capability of conducting directive optimizations, but also supports synthesizable C/C++ code emission. HLS directives are organized into three categories, function, loop, and array directives, of which the details are discussed in Section 4.3.

3.2 Optimization

Optimization passes. On each level of IR, including graph, loop, and directive level, we have implemented optimization passes to improve the HLS design quality. The unique hierarchical IR of ScaleHLS allows the optimization passes to be implemented at the most suitable abstraction level, thereby minimizing the processing complexity and improving the scalability. Details of the transform passes are discussed in Section 5.1 to 5.4.

HLS QoR estimator. In order to efficiently explore the large design spaces brought by large HLS designs, we have developed a fast QoR estimator based on analytical models, which estimate the latency and resource utilization of programs in the structured directive-level IR. Details are discussed in Section 5.5.1.

3.3 Exploration

Transform and analysis library. The interfaces of the QoR estimator and transform passes of all abstraction levels are packaged into an HLS transform and analysis library. All the interfaces in the library are highly parameterized and can be tuned by developers or DSE engines. This library turns the HLS optimization techniques from manual code rewriting to callable and tunable interfaces at different abstraction levels, ranging from task-level dataflow, to loop transformation and directives insertion. Details are discussed in Section 5.

Automated DSE engine. Leveraging the HLS transform and analysis library, we designed an automated DSE engine to search for the Pareto frontier of the multi-dimensional design space, where each dimension corresponds to a tunable parameter of a transform pass. Details of the DSE algorithm are discussed in Section 5.5.2. The DSE engine can be extended to support other optimization algorithms in the future.

3.4 Integration

HLS C front-end. We have implemented an HLS C front-end based on Clang that directly translates input C programs into the `scf` dialect by traversing the Clang AST (Abstract syntax tree). An `scf` to `affine` raising pass automatically identifies affine regions and converts `scf` operations to their corresponding `affine` operations, enabling subsequent polyhedral transformations and analyses. Details are discussed in Section 6.1.

HLS C/C++ emitter. After the completion of all conversions and optimizations, we translate the structured directive-level IR into synthesizable C++ code which is passed to external HLS tools to generate RTL code. Details are discussed in Section 6.2. Meanwhile, LLVM IR [24] can also be generated, enabling software simulation and direct interfacing with other existing LLVM-compatible tools, such as Xilinx Vitis HLS [20].

4 SCALEHLS REPRESENTATION

ScaleHLS features a unique multi-level representation which allows the transform and analysis passes to be applied on multiple abstraction levels, thereby exploring more comprehensive design spaces and improving scalability. In this section, we first introduce the graph and loop-level IRs of ScaleHLS in detail. Then, we introduce the representation of function, loop, and array directives of HLS.

4.1 Graph-level IR

ScaleHLS exploits existing third-party dialects in MLIR, including the ONNX dialect from ONNX-MLIR [26] and the ATen dialect from NPComp [9], to represent and transform graph-level IR. An example of the assembly form of an `onnx.Conv` operation is (attributes and non-tensor operands are omitted for simplicity):

```
%output = "onnx.Conv"(%input, %weight, ...) {...} :
    (tensor<1x3x34x34xf32>, tensor<64x3x3x3xf32>, ...)
    -> tensor<1x64x32x32xf32>
```

Table 1: Supported HLS directives.

	Function	Loop	Array
Directives	dataflow pipeline inline	dataflow pipeline unroll merge	partition resource interface

where the matrix operands and result are typed as tensors. Operations of these dialects consume and produce tensor-type values, which allows optimizing the IR at this level through simple define-use analysis. If these operations are lowered to loop-level and tensors are bufferized to memories, tensor data must be accessed through memory read and write operations, making optimization and transformation more cumbersome due to the need for sophisticated memory dependency analysis. In contrast, many high level transformations and optimizations, such as graph node merging, can be easily supported in a graph-level IR by manipulating tensor operations. The transformations and optimizations implemented in ScaleHLS are discussed further in Section 5.1.

4.2 Loop-level IR

Once the graph-level optimizations are completed, the IR will be lowered to loop-level for further optimization. ScaleHLS exploits the MLIR built-in dialects, particularly `affine` and `scf`, to represent loop-level IR for reusing the powerful analysis and transform libraries provided by MLIR. The code block (ii) of Figure 5 shows the loop-level IR of an SYRK (symmetric rank-k update of a matrix) computation kernel [2] in MLIR where types and attributes of all operations are omitted for simplicity. Memory access and math operations are nested in `affine.for` operations, which explicitly represent the loop structure of the program. Similarly, the code block (iii) of Figure 5 shows the structured representation of a conditionally executed MLIR block contained by an `affine.if` operation. Compared to the unstructured IR, the structured loop-level IR enables more flexible and efficient loop optimizations (e.g., loop tiling). Furthermore, the fast affine expression composition and the use of affine transformation theory allow ScaleHLS to perform efficient and comprehensive analysis and transformation on `affine` operations. The loop-level optimizations are discussed in detail in Section 5.2.

4.3 HLS Directives

HLS tools typically use program directives to guide the hardware generation and fine-tune the performance-area tradeoff. In this section, we introduce how ScaleHLS represents the common function, loop, and array HLS directives shown in Table 1. The directive representation enables ScaleHLS to systematically conduct directive optimizations for improving the design quality.

4.3.1 Function Directives. ScaleHLS supports coarse-grained and fine-grained parallelism through applying directives. The *dataflow* directive enables task parallelism by pipelining all sub-functions that appear in the target function. In the generated hardware, the top-module will be ready to accept a new frame of data once the first sub-module is done, which effectively improves the throughput of the top-module. The *pipeline* directive enables operation parallelism by scheduling all operations in the target function into

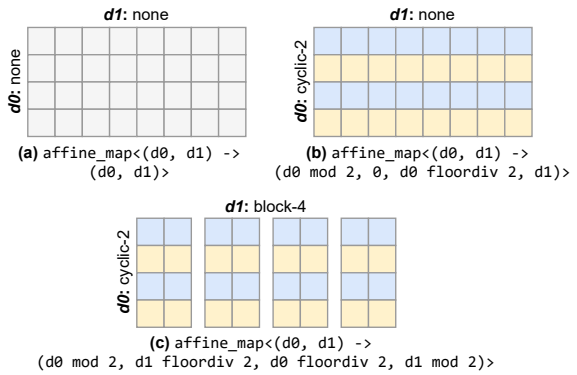


Figure 3: Affine-based array partition. $d\{n\}$ indicates the n-th dimension of the array. Partition fashions and factors: (a) without partition; (b) cyclic partitioned along the dimension-0 with a factor of 2; (c) block partitioned along dimension-1 with a factor of 4.

multiple pipelined stages that can be executed in parallel. For the *pipeline* directive, ScaleHLS allows specifying the targeted initiation interval (*II*), which indicates that the pipeline accepts and processes a new input every *II* clock cycles, impacting the resource usage and performance of the generated pipeline. To represent and parse these directives in ScaleHLS, we customize a structure MLIR attribute named `FuncDirective` in `hlscpp` dialect. The customized attribute contains two Boolean parameters, `dataflow` and `pipeline`, and one integer parameter, `targetII`, which triggers generation of appropriate directives compatible with downstream tools, such as Xilinx Vivado HLS [19]. In ScaleHLS, the function *inline* directive is not explicitly represented with attributes, but instead directly inlines the target function in the IR to ease the transformation and analysis.

4.3.2 Loop Directives. The throughput and latency of loop regions can also be optimized by applying the *dataflow* and *pipeline* directives, which largely share the same characteristics with the corresponding function directives. Note that ScaleHLS can automatically identify perfectly nested loops and flatten them into a single loop hierarchy, which helps to further improve the throughput and latency. Similar to function directives, ScaleHLS also exploits customized MLIR attributes to represent the loop *dataflow* and *pipeline* directives and the targeted *II*. A `LoopDirective` attribute is defined in `hlscpp` dialect and attached to the corresponding `affine.for` or `scf.for` operations when directives are applied.

The computation parallelism of loops can be improved by applying the loop *unroll* directive with the cost of consuming more resources. The *merge* directive is used to fuse adjacent loop nests to improve data locality and decrease the loop control overhead. ScaleHLS does not explicitly represent these two directives through MLIR attributes, but instead directly performs loop transformation on the target loops in the IR, which is semantically equivalent to applying the directives.

4.3.3 Array Partition. Array partition is one of the most important HLS directives because the HLS design requires enough on-chip memory bandwidth to comply with the computation parallelism. However, single on-chip memory block has limited read/write ports and therefore needs to be partitioned into multiple physical blocks to enable massive simultaneous read and write. As MLIR attaches an affine map to each memory type for encoding the memory layout, ScaleHLS reuses the affine-based memory typing system of MLIR to flexibly represent the partition factor (the number of memory blocks after partition) and various partition fashions (e.g., *cyclic* and *block*). Figure 3 shows three running examples including: (a) array without partition, (b) partitioned along the first dimension, and (c) partitioned along both two dimensions. The partition fashions and factors and the corresponding affine map are annotated to each example as well. As we introduced in Section 2.2, affine map is a transform function mapping a list of affine inputs to a list of results. To represent array partition in ScaleHLS, assuming an N-dimensional target array, the attached affine map always have N inputs and $2N$ results. While the inputs are the logical indices of the array, the first and last N results are used to encode the expressions of partition indices and physical indices after array partition, respectively. Taking the affine map of Figure 3(b) as an example, the partition index and physical index of $d0$ can be calculated as $d0 \% 2$ and $\lfloor d0 / 2 \rfloor$ when dimension-0 is partitioned cyclically with a factor of two.

By encoding the partition information into the memory types, ScaleHLS can flexibly support different partition fashions, and can quickly infer the partition index and real accessing address of a memory read/write operation through affine expression composition. This technique is used in the QoR estimator (Section 5.5.1) and the `-array-partition` pass (Section 5.3.2). Note that unsupported memory partition fashions by the downstream HLS tools are disallowed in the directive-level IR of ScaleHLS.

4.3.4 Array Resource and Interface. The HLS-based accelerators can use different kinds of memories, including on-chip memories (e.g., BRAM and URAM) and off-chip memories (e.g., DRAM). The resource directive is introduced for indicating what kind of memories should an array be allocated to. This is similar to the concept of memory space in the software, where BRAM, URAM, and DRAM respond to L1 cache, L2 cache, and main memory of a common computer system. As MLIR also encodes the memory space into the memory type system, ScaleHLS reuses this for representing resource directive by mapping different kinds of memories into different memory spaces. Notably, ScaleHLS distinguishes single port, simple dual-port, and true dual-port on-chip memories to precisely control the resource utilization. Additionally, if an array is identified as a function argument or returned value, ScaleHLS will automatically determine the interface category (e.g., AXI [18] or naive BRAM interface) of the array according to its memory space.

5 SCALEHLS OPTIMIZATION

On top of the hierarchical representation of ScaleHLS, we propose a multi-level HLS optimization methodology to address the challenges of optimizing large HLS designs. This methodology is implemented using a set of MLIR transformation passes, each operating on MLIR dialects at an appropriate abstraction level, either the graph, loop, or directive levels described above. All ScaleHLS transform passes, their transform targets (e.g., function), and the tunable parameters are listed in Table 2, where a *Loop Band* in MLIR refers to a continuous set of loops. These passes traverse the whole

Table 2: ScaleHLS passes. Boldface ones are new passes provided by ScaleHLS, while others are MLIR built-in passes.

	Passes	Target	Parameters
Graph	**-legalize-dataflow**	function	insert-copy
	-split-function	function	min-gran
Loop	**-affine-loop-perfectization**	loop band	-
	-affine-loop-order-opt	loop band	perm-map
	-remove-variable-bound	loop band	-
	-affine-loop-tile	loop	tile-size
	-affine-loop-unroll	loop	unroll-factor
	-affine-loop-fusion	loop	-
Direct.	**-loop-pipelining**	loop	target-ii
	-func-pipelining	function	target-ii
	-array-partition	function	part-factors
Misc.	**-simplify-affine-if**	function	-
	-affine-store-forward	function	-
	-simplify-memref-access	function	-
	-canonicalize -cse	function	-

IR and operate on all suitable targets in the IR, making it difficult to apply different combinations of passes on different targets through the command line tool. To solve this problem, we also expose the functionality of each transform pass as a callable method, allowing precise control on where transforms are applied. These methods together with the QoR estimator are packaged into an HLS transform and analysis library, which opens the opportunity to perform comprehensive DSEs by applying different combinations of transforms on different targets in the IR and tuning their parameters. In this section, we first introduce the graph, loop, and directive passes accordingly. Then, we introduce other transform passes provided by ScaleHLS for eliminating redundancies. Finally, we introduce details of the QoR estimator and the automated DSE algorithm.

5.1 Graph Transform Passes

5.1.1 Legalize Dataflow. Downstream HLS tools often support dataflow pipelining with specific restrictions in coding style. In particular, for Vivado HLS each intermediate result must have only one producer and one consumer, bypass and feedback paths are not allowed, and conditional execution of sub-functions are not allowed [19]. Previously, users were required to manually legalize the target function by splitting the function body into multiple sub-functions and rewriting the code structure to eliminate the bypass, multi-producer, or multi-consumer data paths. This procedure is (1) error-prone since careless rewriting can easily result in incorrect functionality and (2) less effective since large HLS designs containing tens of sub-functions can take up to hours for human designers to reorganize and split. The drawbacks of such manual efforts obstruct the existing HLS tools to effectively explore different configurations of the dataflow pipelining.

To address this problem, we introduce a `-legalize-dataflow` pass in ScaleHLS to analyze the dependencies between dataflow nodes and automatically legalize the targeted function. Figure 4(a) shows an example dataflow containing five procedures, where each edge corresponds to a tensor delivering. We can observe that Figure 4(a) is illegal as there is a path between *Proc 0* and *Proc 3* bypassing *Proc 1-2*. This dataflow can be conservatively legalized to Figure

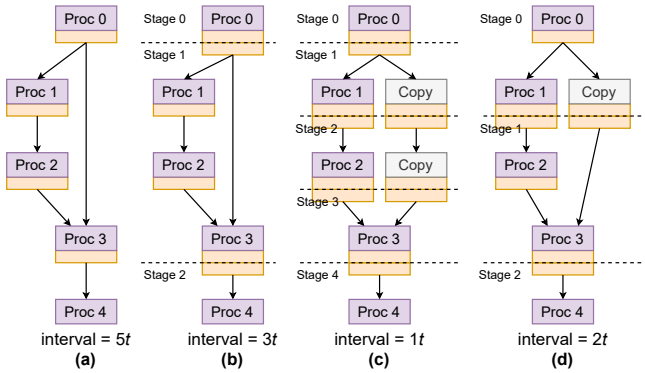


Figure 4: Graph-level dataflow optimization. (a) original dataflow; (b) legalized dataflow without copy nodes; (c) legalized dataflow with inserting copy nodes; (d) dataflow with a minimum granularity of 2.

4(b) through the `-legalize-dataflow` pass. To eliminate the bypass path, *Proc 1-3* are organized into the same dataflow stage, thereby *Proc 0* -> *Proc 1-3* -> *Proc 4* can construct a 3-stages dataflow. Note that in Figure 4(b), the output buffers of *Proc 0* and *Proc 3* are automatically ping-pong buffered after the dataflow pipelining directive is successfully applied, with the cost of utilizing more memory resources than the original dataflow in Figure 4(a).

Alternatively, the dataflow can be aggressively legalized to Figure 4(c) through inserting *Copy* nodes. The original bypass path is broken by the two inserted *Copy* nodes, which enable a more fine-grained 5-stage dataflow. Assuming each procedure in the dataflow has a latency of $1t$, the conservative and aggressive legalization improves the dataflow interval from $5t$ to $3t$ and $1t$, respectively. However, the downside of the aggressive legalization is more computation and memory resources are consumed. The strategy of inserting copy nodes can be tuned through a `insert-copy` pass option. If the `insert-copy` option is enabled, copy nodes are inserted until the main path and the bypass path have the same number of nodes on them. Note that if the target function cannot be legalized, the dataflow pipelining directive will not be applied.

5.1.2 Split Function. Once the dataflow is legalized, the original function can be splitted into a top function and multiple sub-functions by the `-split-function` pass. Procedures or inserted copy nodes organized into the same dataflow stage can be safely clustered into a new sub-function and converted to a function call. At this stage, we find that a throughput-area tradeoff space can be explored by merging adjacent dataflow stages into one. For example, in Figure 4(d), every two adjacent stages are merged together, constructing a new 3-stages dataflow with less resource utilization compared to Figure 4(c) and an interval of $2t$. To enable this design space, we define *granularity* as the number of adjacent dataflow stages to be merged. The `-split-function` pass supports a `min-gran` parameter to specify the minimum granularity during the splitting. Therefore, at least `min-gran` adjacent dataflow stages are splitted into the one sub-function and converted to one function call.

5.2 Loop Transform Passes

We use the SYRK computation kernel shown in Figure 5 as the example in the following discussion. In this section, we introduce

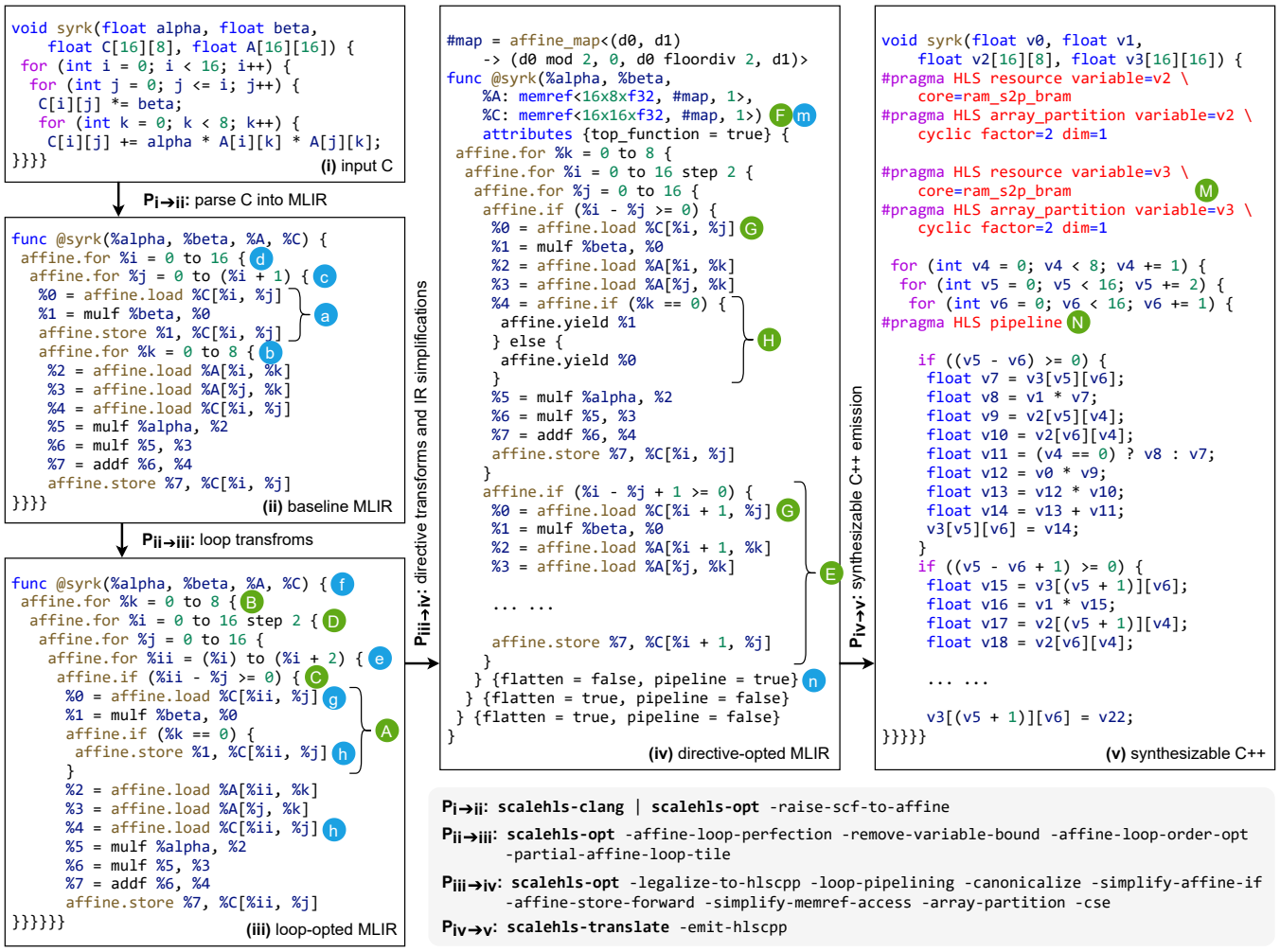


Figure 5: An SYRK computation kernel example. `scalehls-clang` compiles C program into the MLIR framework. `scalehls-opt` is the command line tool for conducting all conversion, transform, and analysis passes of ScaleHLS, while `scalehls-translate` is for the MLIR to C/C++ translation. Some operation attributes or types are omitted for simplicity.

the loop transform passes provided by ScaleHLS, which is corresponding to the $P_{ii \to iii}$ transformation of Figure 5.

5.2.1 Loop Perfectization. Operations between loop statements, such as Figure 5ⓐ (hereinafter referred to as 5ⓐ, 5ⓑ, etc.), result in imperfect loops that may interfere with some important optimizations (e.g., loop tiling) and prevent the outer loops from being flattened for reducing latency. The `-affine-loop-perfectization` pass relocates the three in-between operations (5ⓐ) into the innermost loop and transforms them to 5Ⓐ, where all in-between operations are moved into a newly created `affine.if`. Then, operations except the state-modifying operations, such as stores, are hoisted out of the conditional.

5.2.2 Loop Order Optimization. Loop permutation can change the distance of loop-carried memory dependencies, thereby reducing the achievable *II* of loop pipelining and reducing latency. The `-affine-loop-order-opt` pass can automatically perform affine-based memory dependency analysis and apply the best legal loop order to the targeted loop band. In the SYRK example, the original innermost `%k`-loop (5ⓑ) is permuted to the outermost location (5Ⓑ) by the loop order optimization pass. This pass also accepts an optional integers list, `perm-map`, allowing the loop order to be explicitly specified. The *i*-th element of `perm-map` indicates the new position of the *i*-th loop in the original loop band, where positions are from outermost loop to inner.

5.2.3 Remove Variable Loop Bound. Because MLIR focuses on rectangular iteration spaces, there are limitations on analyzing and transforming non-rectangular nested loops in MLIR. As a result, variable loop bounds may obstruct some loop optimizations and disrupt QoR estimation. The `remove-variable-bound` pass can calculate the minimum or maximum value of the expression of a variable loop bound as long as each item is a loop induction variable and has known lower and upper bounds. In the SYRK example, the variable loop bound of the `%j`-loop (5ⓒ) is substituted with the constant value and an `affine.if` operation (5Ⓒ) is generated in the innermost loop for the conditional execution of the whole loop body. Although this pass may increase the overall iteration number of the loop band, it opens opportunities for subsequent optimizations which may offset the negative side effect.

5.2.4 Loop Tiling. Loop tiling is a common loop transform to improve data locality and accommodate the limited capacity of on-chip buffers. In the SYRK example, the `%i`-loop (5ⓓ) is tiled with a factor of 2 and transformed into 5Ⓓ, and the generated intra-tile `%ii`-loop is relocated into the innermost loop. The legality of loop tiling is validated before the transform is applied to maintain the correct functionality. The tiling size is determined by a `tile-size` parameter which can be tuned by the DSE engine.

5.3 Directive Transform Passes

In this section, we introduce the directive transform passes of ScaleHLS, which manipulate HLS-specific directives for improving the design quality. The effect of the discussed passes are showcased in the $P_{iii \rightarrow iv}$ transformation of Figure 5.

5.3.1 Function and Loop Pipelining. A legal pipeline directive allows no hierarchy in the target function or loop, thus all the sub-loops must be fully unrolled and all the sub-functions should be also pipelined [19]. The `-loop-pipelining` pass first attempts to legalize the targeted loop by fully unrolling all contained loops and pipelining all sub-functions. If the legalization successes, loop pipeline directive is applied to the target loop. In the SYRK example, loop pipelining is applied to the `%j`-loop and thus the contained `%ii`-loop (5ⓔ) is fully unrolled and the duplicated loop body after loop unrolling is shown in 5Ⓔ. The `%j`-loop is annotated as `pipeline` and all outer perfectly nested loops, `%k` and `%i`-loop, are annotated as `flatten`.

The `-function-pipelining` pass uses the same mechanism to legalize the targeted function before setting the function pipeline directive. Both the loop and function pipelining allow specifying the targeted *II* for exploring the tradeoff design space between throughput and on-chip resource utilization.

5.3.2 Array Partition. ScaleHLS enhances the method proposed in [54] to automatically detect the memory access pattern of a program and apply the suitable array partition directive to each dimension of each on-chip memory. The array partition metric P of the d-th dimension of the i-th array can be represented with:

$$P_{i,d} = \frac{Accesses_i}{\underset{m,n}{max}(index^m_{i,d} - index^n_{i,d} + 1)}, \tag{1}$$

where $Accesses_i$ is the number of unique memory accesses in the targeted MLIR blocks, $index^m_{i,d}$ and $index^n_{i,d}$ are the indices of the m-th and n-th memory access operations. Note that m and n can be any two different memory accesses. The `-array-partition` pass applies *cyclic* and *block* partitions to the d-th dimension of the i-th array when $P_{i,d} >= 1$ and $P_{i,d} < 1$, respectively, with the partition factor set to $Accesses_i$. Taking the first dimension of the `%C`-array (5Ⓕ) as example, the index distance between the only two memory accesses (5Ⓖ) is $(\%i + 1) - \%i + 1 = 2$. Therefore, we have $P = 1$ and the applied partition fashion is *cyclic*, which is encoded into the affine map of `%C`-array.

As instantiated arrays can be accesses by sub-functions through reference or pointer passing, inter-procedure analysis is conducted to ensure: (1) the array partition directives are applied in the correct function scopes; (2) the globally optimal partition strategies are selected. The array partitioning process can also be guided by specifying the partition factors of each array which appears in the function through the `part-factors` parameter.

5.4 IR Redundancy Elimination

In addition to the graph, loop, and directive transform passes, ScaleHLS adopts the methodology in [1] and implements multiple other passes to remove the redundant operations in HLS designs. The `-simplify-affine-if` pass eliminates dead branches of `affine.if` operations by detecting always-true or always-false conditions using affine analysis. The `-affine-store-forward` pass eliminates redundant memory read or write operations and unused memory instances through store-to-load forwarding. The `-simplify-memref-access` pass folds identical memory read or write operations if no memory dependency conflict is found. In the SYRK example, the memory access operations (5ⓗ) are eliminated and the IR is transformed to 5Ⓗ. ScaleHLS also exploits MLIR built-in passes, such as `-canonicalize` and `-cse` (common subexpression elimination) [8], to further simplify the IR and optimize the quality of the HLS design.

5.5 Automatic Design Space Exploration

In the previous sections, we first introduced the ScaleHLS representation which provides a comprehensive capability to represent HLS designs from multiple abstraction levels. Then, we introduced the transform passes which provide convenient and tunable interfaces to optimize the design. On top of the representation and optimization of ScaleHLS, we can construct a multi-dimensional design space, where each dimension is corresponding to the on/off or a tunable parameter of a transform pass. In this section, we propose an automated DSE engine assisted with an analytical model-based QoR estimator for exploring the design space.

5.5.1 QoR Estimation. The RTL generation downstream tools, such as Vivado HLS, can take minutes to hours to complete the compilation and to report the synthesis results, which (1) limits the total number of design points that can be evaluated during DSE, thus results in sub-optimal solutions and (2) significantly increases the DSE time to up to tens of hours. To solve these problems and rapidly evaluate the design points found by the DSE engine, we develop a QoR estimator based on the structured IR to estimate the latency and resource utilization of the HLS designs. We adopt an ALAP (as late as possible) algorithm to schedule each MLIR block in the design. The memory ports are considered as non-shareable resources and constrained in the scheduling except between two or more memory read operations with identical address indices. The dependencies between operations are extracted through define-use and memory dependency analysis, where function calls and loops in the MLIR block are viewed as nodes in the dependency graph.

As discussed in Section 4.3.2, ScaleHLS directly unrolls the loops in the IR when loop unrolling directives are applied, thus loop unrolling does not need to be separately handled in the estimation. Since loop pipelining directives are represented with customized MLIR attributes, the estimator will parse the attribute and estimate the minimal *II* if a loop is pipelined. We adopt the algorithm proposed in [40] for calculating the minimal *II*. We have:

$$II_{min} = max(II^{res}_{min}, II^{dep}_{min}), \tag{2}$$

where II_{min}^{res} and II_{min}^{dep} are the minimal resource-constrained II and dependency-constrained II, which can be calculated as:

$$II_{min}^{res} = \max_{i,p} \left(\left\lceil \frac{Accesses_{i,p}}{Ports_{i,p}} \right\rceil \right), \tag{3}$$

$$II_{min}^{dep} = \max_{d} \left(\left\lceil \frac{Delay_d}{Distance_d} \right\rceil \right). \tag{4}$$

$Accesses_{i,p}$ and $Ports_{i,p}$ are the number of memory access operations and memory ports of the p-th partition of the i-th array. $Delay_d$ and $Distance_d$ are the scheduling delay and distance (calculated from the dependency vector) of each pair of loop-carried dependencies.

5.5.2 DSE Algorithm. The target of the DSE engine is to search for the Pareto frontier of the latency-area tradeoff space. By tuning the parameters of the transform passes shown in Table 2, we can construct a multi-dimensional design space for each input HLS design. Although the proposed QoR estimator can rapidly map a design point discovered in the multi-dimensional design space to the latency-area space, the powerful ScaleHLS transform passes can easily generate millions of design points, making exhaustive search impossible. Through sampled profiling of the design spaces, we find that the Pareto points in the latency-area space are clustered in the multi-dimensional design space, which is also observed in previous works [14, 46]. For example, if pipeline $II = 2$ is a Pareto point for a nested loop, there is a high possibility that its neighbors (e.g., pipeline $II = 3$) are also Pareto points with different latency-area tradeoffs. Based on this observation, we design a 4-step neighbor-traversing algorithm for solving the optimization problem:

In step (1), we sample the whole design space and evaluate each sampled design point using the QoR estimator. In step (2), the Pareto frontier is extracted from all evaluated design points. In step (3), we evaluate the closest neighbor of a random selected design point in the current Pareto frontier. Finally, in step (4), we repeat step (2) and (3) to update the discovered Pareto frontier until no eligible neighbor can be found or meeting the early-termination criteria (e.g., maximum iteration number). This DSE algorithm is implemented as an MLIR transform pass called `-multiple-level-dse` which can be applied on the input HLS designs without any manual efforts. Note that given the HLS transform and analysis library of ScaleHLS, the DSE engine is extensible to support different optimization algorithms.

6 END-TO-END INTEGRATION

6.1 HLS C Front-end

The C front-end takes synthesizable HLS C code and emits the corresponding MLIR in the `scf` dialect. The `scf` dialect provides an abstraction for static control flow and has a similar set of operations to statements in C, which reduces the analysis process in the front-end. For instance, a `for` loop can be directly translated to a `scf.for` operation. The output in the `scf` dialect is then *raised* into the `affine` dialect using an ScaleHLS pass called `-raise-scf-to-affine`. This pass checks whether an `scf.for` operation is an affine loop and translates it into an `affine.for` operation if it is. Otherwise, the loop remains as an `scf.for` operation. Also, the MLIR pass raises each memory statement to an `affine` operation if its address indices have affine formats. The $P_{i \to ii}$ transformation of Figure 5 shows the procedure of parsing HLS C codes into the MLIR framework.

MLIR has its unique memory and indexing types. First, the memory type `memref`, also known as memory reference, is a set of exclusive pointers to the memory and size parameters of the memory [25]. The `memref` type solves delinearization problem of parametrically sized arrays, which was not well-supported in LLVM [15]. The translation to `memref` is simplified in our front-end because common HLS tools, such as Vivado HLS, only accepts a subset of C [19]. For instance, all the arrays have to have fixed sizes, and pointers have to point to scalars. These types are directly translated to fixed-size `memref` types. A pointer that points to a scalar has a 1×1 `memref` type in MLIR. If an unsupported struct such as pointer to pointer is found, the input code is rejected by the C front-end.

Second, the `index` type in MLIR is an integer type with a platform-specific bit width. The `index` types are typically used for a set of constructs such as loop iterators and memory indices. During the translation, our C front-end automatically checks whether an integer variable can be used as one of these constructs. For instance, the iterator in `index` type of a `for` loop should not overflow, otherwise the loop is translated into a `scf.while` loop.

6.2 HLS C/C++ Code Emission

After the completion of all conversions and optimizations, the structured IR can be emitted as synthesizable C/C++ code for generating the RTL code. The $P_{iv \to v}$ transformation of Figure 5 shows the MLIR to C++ emission of the SYRK example. The HLS C/C++ emitter of ScaleHLS requires the control flow to be represented by `affine` or `scf` operations. Then, it can directly translate `affine/scf.for` and `if` operations to the `for` and `if` statements in C/C++. The array partition, resource, and interface information is decoded from the type of memories (5ⓜ) and emitted as `pragma` directives (5Ⓜ). Meanwhile, the applied HLS-specific optimizations represented as attributes (5ⓝ) are also parsed by the emitter accordingly and inserted into the corresponding code region. Notably, to ensure the synthesizability of the generated C/C++ code, the emitter always converts returned scalars and memories to input pointers and memory references, respectively. The `memref` types and `index` types are directly translated into fixed-size array types and integer types.

7 EXPERIMENTAL RESULTS

To evaluate the ScaleHLS compilation framework, we conduct comprehensive experiments and ablation studies in this section. Xilinx Vivado HLS 2019.1 is adopted for generating RTL code. All reported performances and resources utilization are collected from the synthesis results of Vivado HLS.

7.1 Large-Scale Computation Kernels

7.1.1 Automatic DSE results. We evaluate the DSE engine on six different computation kernels (BICG, GEMM, GESUMMV, SYR2K, SYRK, and TRMM) picked from PolyBench-C [39] with a problem size of 4096. The target platform is Xilinx XC7Z020 FPGA, which is an edge FPGA with 4.9 Mb memories, 220 DSPs, and 53,200 LUTs. The resource constraints and non-optimized computation kernels written in C are passed into the DSE engine, which is then launched to search for the optimal solutions. Finally, the generated designs

Table 3: DSE results of large-scale computation kernels. The data types of all kernels are single-precision floating-points. *Speedup* is with respect to the baseline designs from PolyBench-C without the optimization of DSE. *LP* and *RVB* denote *Loop Perfectization* and *Remove Variable Bound*, respectively. In the *Loop Order Optimization*, the i-th loop in the loop nest is permuted to location $PermMap[i]$, where locations are from the outermost loop to inner.

Kernel	Prob. Size	Speedup	LP	RVB	Perm. Map	Tiling Sizes	Pipeline II	Array Partition Factors
BICG	4096	41.7×	No	No	[1, 0]	[16, 8]	43	A:[8, 16], s:[16], q:[8], p:[16], r:[8]
GEMM	4096	768.1×	Yes	No	[1, 2, 0]	[8, 1, 16]	3	C:[1, 16], A:[1, 8], B:[8, 16]
GESUMMV	4096	199.1×	Yes	No	[1, 0]	[8, 16]	9	A:[16, 8], B:[16, 8], tmp:[16], x:[8], y:[16]
SYR2K	4096	384.0×	Yes	Yes	[1, 2, 0]	[8, 4, 4]	8	C:[4, 4], A:[4, 8], B:[4, 8]
SYRK	4096	384.1×	Yes	Yes	[1, 2, 0]	[64, 1, 1]	3	C:[1, 1], A:[1, 64]
TRMM	4096	590.9×	Yes	Yes	[1, 2, 0]	[4, 4, 32]	13	A:[4, 4], B:[4, 32]

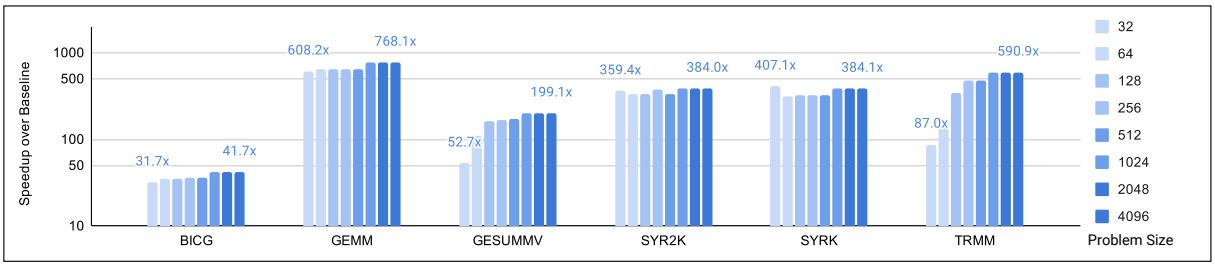


Figure 6: Scalability study of computation kernels. The problem sizes of computation kernels are scaled from 32 to 4096 and the DSE engine is launched to search for the optimal solutions under each problem size.

are evaluated and the results are shown in Table 3. Among all six benchmarks, a speedup ranging from 41.7× to 768.1× is obtained compared to the baseline design, which is the original computation kernel from PolyBench-C without the optimization of DSE. Table 3 also lists the optimal parameters selected for each transform pass. Notably, in the procedure of loop tiling, all generated intra-loops are absorbed into the innermost loop region and fully unrolled for increasing the computation parallelism.

After studying the optimal solutions discovered by the DSE engine, we find the performance gains come from multiple sources: (1) loop perfectization and variable loop bound elimination regularize the target loop bands and enable the subsequent optimizations; (2) loop permutation alleviates (or eliminates) the impact of memory dependencies and improves the achievable pipeline II by reducing the $Distance_d$ in Equation 4 of loop-carried dependencies; (3) the computation parallelism and resource utilization are increased through loop tiling and intra-tile loop unrolling; (4) loop pipelining is applied and the target II is fine-tuned to tradeoff between resource-sharing and throughput while accommodating the resource constraints; (5) array partitioning strategies are automatically selected to match the memory access patterns after loop transformations. The BICG benchmark cannot benefit from loop permutation because every loop in the loop nests is associated with critical loop-carried dependency which prevents the DSE engine to effectively reduce the pipeline II. However, the DSE engine still discovers a reasonable solution for the BICG benchmark and achieves a 41.7× speedup through increasing the computation parallelism. Benchmarks except BICG benefit from all speedup sources above, and achieve significant performance improvement under the constrained on-chip resources available on the edge FPGA platform.

7.1.2 Comparison with Previous Works. Previous efforts [54–56] have also investigated automatic DSE methods to optimize computation kernel level algorithms. However, they only support directive optimizations, thus are difficult to comprehensively explore the design space and find reasonable design points when the problem sizes are large. For example, on the six scaled-up benchmarks shown in Table 3, the open-sourced framework [54] either generates solutions that cannot be synthesized by Vivado HLS or takes an unreasonable long time on exploring the large design spaces. Meanwhile, as previous DSE efforts do not support HLS-dedicated representation and the transform and analysis library featured by ScaleHLS, they still rely on human to provide optimization hints or rewrite the code before launching the DSE, leading to low-efficient and partially-automated compilation flows. Our multi-level representation and automated optimization enable ScaleHLS to find previously unachievable design points, explore a more comprehensive design space, and directly generate synthesizable HLS designs.

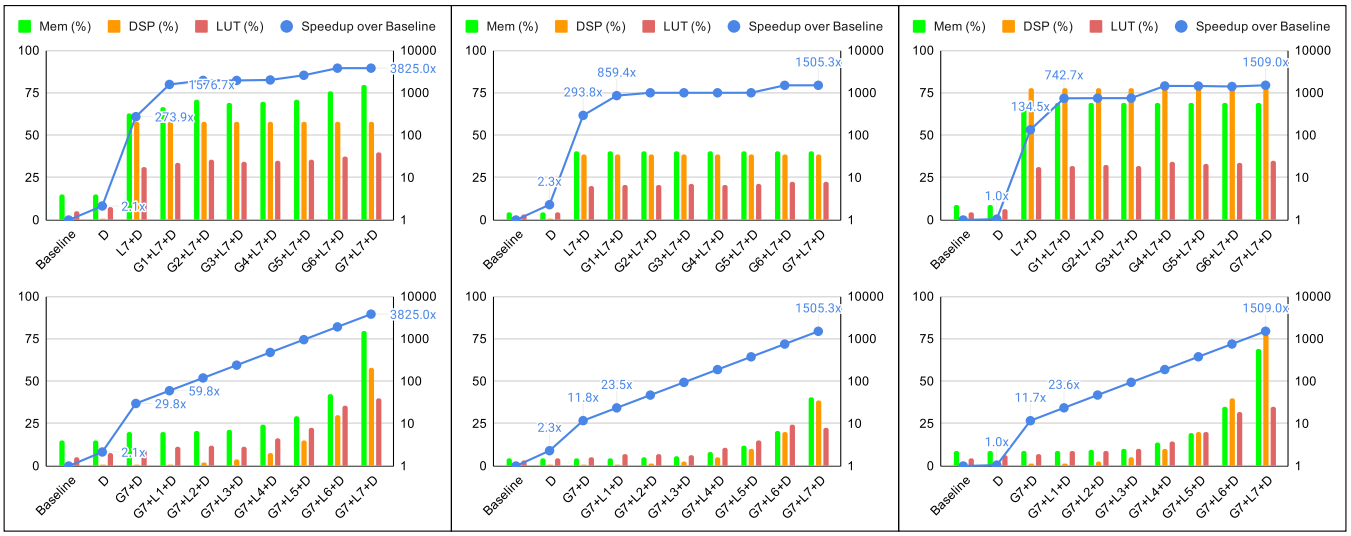
7.1.3 Scalability Study. To understand the performance of our framework on different problem sizes, we scale the problem sizes of the six benchmarks from 32 to 4096 and launch the DSE engine to search for the optimized solution under each setting. Figure 6 shows the experimental results. We can observe that for BICG, GEMM, SYR2K, and SYRK benchmarks, the DSE engine can achieve stable speedup under all problem sizes. For GESUMMV and TRMM, the speedups for small problem sizes are lower because the small design space prevents the DSE engine from fully utilizing the available on-chip resources. Overall, our framework shows a strong scalability and can effectively optimize computation kernel level algorithms under a wide range of problem sizes.

7.2 Large and Complicated Algorithms

7.2.1 Optimization Results. We experiment the ability of handling large and complicated HLS designs of ScaleHLS by evaluating three representative DNN (deep neural networks) models for the CIFAR-10 [22] image classification task, ResNet-18 [16], VGG-16[45], and MobileNet [17]. These DNN models are constructed with a large

Table 4: Optimization results of representative DNN models. *Speedup* is with respect to the baseline designs compiled from PyTorch by ScaleHLS but without the multi-level optimization.

Model	Speedup	Runtime (seconds)	Memory (SLR Util. %)	DSP (SLR Util. %)	LUT (SLR Util. %)	FF (SLR Util. %)	Our DSP Effi. (OP/Cycle/DSP)	DSP Effi. of TVM-VTA [32]
ResNet-18	3825.0×	60.8	91.7Mb (79.5%)	1326 (58.2%)	157902 (40.1%)	54766 (6.9%)	1.343	0.344
VGG-16	1505.3×	37.3	46.7Mb (40.5%)	878 (38.5%)	88108 (22.4%)	31358 (4.0%)	0.744	0.296
MobileNet	1509.0×	38.1	79.4Mb (68.9%)	1774 (77.8%)	138060 (35.0%)	56680 (7.2%)	0.791	0.468



(a) ResNet-18 **(b)** VGG-16 **(c)** MobileNet

Figure 7: Ablation study of DNN models. D, $L\{n\}$, and $G\{n\}$ denote directive, loop, and graph optimizations, respectively. Larger n indicates larger loop unrolling factor and finer dataflow granularity for loop and graph optimizations, respectively.

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 memories, 2280 DSPs and 394,080 LUTs on each SLR. The PyTorch [38] 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 4. We can observe that by combining all three levels of optimization, the generated HLS designs achieve significant speedups ranging from 1505.3× to 3825.0× on the metric of throughput compared to the baseline designs, which are compiled from PyTorch to HLS C/C++ through ScaleHLS but without the multi-level optimization applied. Notably, as shown in Table 4, ScaleHLS only consumes 37.3 to 60.8 seconds to optimize the large and complicated HLS designs with a single line of command, which demonstrates the efficiency and scalability of our optimization methodology. The runtime is collected by using `-pass-timing`, a built-in statistic pass provided by MLIR.

7.2.2 Comparison with Previous Works. To the best of our knowledge, ScaleHLS is the first general-purpose HLS flow which can optimize and generate ResNet-18 level DNN accelerators without human-designed IPs or templates. Previous HLS optimization flows [54–56] focus on small-scale algorithms, while compilation flows dedicated for DNNs rely on pre-defined IP libraries [32, 47, 53] or parameterized templates [31, 50] to generate the accelerator, which can not be generalized to applications other than DNNs. To better understand optimization results of DNN models, we compare the DSP efficiency with TVM-VTA [32], a widely accepted DNN accelerator written in HLS. DSP efficiency is a common metric for comparing the efficiency of DNN accelerators across different platforms, which can be calculated as:

$$Effi_{DSP} = \frac{OP/s}{Num_{DSP} \times Freq} \tag{5}$$

As shown in Table 4, ScaleHLS reaches a better DSP efficiency, while saves hundreds of human hours for designing the dedicated hardware IPs. These experimental results demonstrate that ScaleHLS can achieve fruitful productivity improvement on accelerating large and complicated algorithms.

7.2.3 Ablation Study. To quantify the speedup contributed by each of the three optimizations (directive, loop, and graph) and evaluate the proposed multi-level optimization methodology, we perform a set of ablation studies and the results are shown in Figure 7. We can observe that the directive (D), loop ($L7$), and graph ($G7$) optimizations contribute 1.8×, 130.9×, and 10.3× average speedups on the three DNN benchmarks, respectively, demonstrating the effectiveness of our multi-level optimization methodology. Note that because the effect of array partitioning will become larger as the loop unrolling factors increase, the actual speedup of directive optimizations are larger than 1.8× when combining with loop optimizations. ScaleHLS allows to tune the optimization level n between 1 to 7 for loop and graph optimizations, which enables to explore the tradeoff space between area and speedup. Larger n

indicates larger loop unrolling factor and finer dataflow granularity for loop and graph optimizations, respectively, leading to higher throughput and more on-chip resources utilization. By comparing the speedup achieved by $G1 + L7 + D$ and $G7 + L7 + D$, we can observe that the speedup margin between $G1$ and $G7$ is 2.1× on average. Similarly, the speedup margin between $L1$ and $L7$ is 64.0× on average.

8 CONCLUSION AND FUTURE WORKS

This paper presents ScaleHLS, an MLIR-based HLS compilation flow, which features multi-level representation and optimization of HLS designs and supports a transform and analysis library dedicated for HLS. ScaleHLS enables an end-to-end compilation pipeline by providing an HLS C front-end and a C/C++ emission back-end. An automated and extensible DSE engine is developed to search for optimal solutions in the multi-dimensional design spaces. Experimental results demonstrate that ScaleHLS has strong scalability to optimize large-scale and sophisticated HLS designs and achieves significant performance and productivity improvements on a set of benchmarks. In addition, ScaleHLS is an open-source project and we hope ScaleHLS could become an advanced open infrastructure of new HLS research in the future and boost the innovation in this area to face new challenges.

ScaleHLS leaves several directions for future works: (1) IP integration. The graph-level IR of ScaleHLS opens the opportunity to integrate existing HLS IPs into the compilation flow, making the integration and optimization of HLS IPs an interesting research direction. (2) DSE algorithms. The transform and analysis library provided by ScaleHLS enables a large opportunity to investigate the optimization algorithms for the multi-dimensional DSE problem of HLS. (3) Machine-learning based QoR estimation. Machine-learning methods can potentially capture more features from the hierarchical IR of ScaleHLS, thereby generating better estimation results than the analytical model-based methods. (4) Generate RTL code within MLIR. Currently ScaleHLS leverages external HLS tools for generating the RTL code. However, a direct RTL code generation can keep more information from the higher level IR and exploit the RTL-level representation and optimization (CIRCT [7]) to further improve the quality of the accelerator designs.

ACKNOWLEDGMENTS

We thank Eric Cheng of Laboratory for Physical Sciences (LPS) and Samuel Bayliss of Xilinx for insightful discussions. 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] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[2] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[3] Deming Chen, Jason Cong, Swathi Gurumani, Wen-mei Hwu, Kyle Rupnow, and Zhiru Zhang. 2016. Platform choices and design demands for IoT platforms: cost, power, and performance tradeoffs. *IET Cyber-Physical Systems: Theory & Applications* 1, 1 (2016), 70–77.

[4] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. 2019. Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs. In *Proceedings of the 2019 ACM/SIGDA international symposium on field-programmable gate arrays*. 73–82.

[5] Alessandro Cilardo and Luca Gallo. 2015. Interplay of loop unrolling and multi-dimensional memory partitioning in HLS. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 163–168.

[6] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[7] CIRCT contributors. 2021. CIRCT: Circuit IR Compilers and Tools. https://github.com/llvm/circt/tree/main/.

[8] MLIR contributors. 2021. MLIR: Multi-Level Intermediate Representation. https://github.com/llvm/llvm-project/tree/main/mlir.

[9] NPComp contributors. 2021. NPComp: MLIR based compiler toolkit for numerical python programs. https://github.com/llvm/mlir-npcomp.

[10] ONNX contributors. 2021. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[11] 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.

[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] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. 2009. Cetus: A source-to-source compiler infrastructure for multicores. *Computer* 42, 12 (2009), 36–42.

[14] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. 2018. Lattice-traversing design space exploration for high level synthesis. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 210–217.

[15] Tobias Grosser, Jagannathan Ramanujam, Louis-Noel Pouchet, Ponnuswamy Sadayappan, and Sebastian Pop. 2015. Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 351–360.

[16] 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.

[17] 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).

[18] Xilinx Inc. 2017. *Vivado Design Suite AXI Reference Guide: UG1037 (v4.0)*.

[19] Xilinx Inc. 2020. *Vitis High-Level Synthesis User Guide: UG1399 (v2020.2)*.

[20] Xilinx Inc. 2021. Vitis HLS Front-end. https://github.com/Xilinx/HLS.

[21] R. Kastner, J. Matai, and S. Neuendorffer. 2018. Parallel Programming for FPGAs. *ArXiv e-prints* (May 2018). arXiv:1805.03648

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

[23] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.

[24] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[25] 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).

[26] Tung D Le, Gheorghe-Teodor Bercea, Tong Chen, Alexandre E Eichenberger, Haruki Imai, Tian Jin, Kiyokuni Kawachiya, Yasushi Negishi, and Kevin O'Brien. 2020. Compiling ONNX Neural Network Models Using MLIR. *arXiv preprint arXiv:2008.08272* (2020).

[27] Seyong Lee, Jungwon Kim, and Jeffrey S Vetter. 2016. Openacc to fpga: A framework for directive-based high-performance reconfigurable computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 544–554.

[28] Seyong Lee and Jeffrey S Vetter. 2014. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 115–120.

[29] Xinheng Liu, Yao Chen, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. 2016. High level synthesis of complex applications: An H. 264 video decoder. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 224–233.

[30] 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.

[31] Yuan Meng, Sanmukh Kuppannagari, Rajgopal Kannan, and Viktor Prasanna. 2021. DYNAMAP: Dynamic Algorithm Mapping Framework for Low Latency CNN Inference. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 183–193.

[32] 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).

[33] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Affine C in MLIR. (2021).

[34] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue* 6, 2 (2008), 40–53.

[35] 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.

[36] 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*. IEEE, 35–42.

[37] 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.

[38] 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.

[39] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. https://www.cs.colostate.edu/~pouchet/software/polybench/.

[40] B Ramakrishna Rau. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*. 63–74.

[41] Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. 2011. A study of high-level synthesis: Promises and challenges. In *2011 9th IEEE International Conference on ASIC*. IEEE, 1102–1105.

[42] Benjamin Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi. 2009. Adaptive simulated annealer for high level synthesis design space exploration. In *2009 International Symposium on VLSI Design, Automation and Test*. IEEE, 106–109.

[43] Benjamin Carrion Schafer and Zi Wang. 2019. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2628–2639.

[44] Kavya Shagrithaya, Krzysztof Kępa, and Peter Athanas. 2013. Enabling development of OpenCL applications on FPGA platforms. In *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 26–30.

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

[46] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2020. AutoDSE: Enabling Software Programmers Design Efficient FPGA Accelerators. *arXiv preprint arXiv:2009.14381* (2020).

[47] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–74.

[48] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[49] Nan Wu, Yuan Xie, and Cong Hao. 2021. IronMan: GNN-assisted Design Space Exploration in High-Level Synthesis via Reinforcement Learning. *arXiv preprint arXiv:2102.08138* (2021).

[50] Hanchen Ye, Xiaofan Zhang, Zhize Huang, Gengsheng Chen, and Deming Chen. 2020. HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[51] Xiaofan Zhang, Haoming Lu, Cong Hao, Jiachen Li, Bowen Cheng, Yuhong Li, Kyle Rupnow, Jinjun Xiong, Thomas Huang, Honghui Shi, et al. 2019. SkyNet: a hardware-efficient method for object detection and tracking on embedded systems. *arXiv preprint arXiv:1909.09709* (2019).

[52] Xiaofan Zhang, Anand Ramachandran, Chuanhao Zhuge, Di He, Wei Zuo, Zuofu Cheng, Kyle Rupnow, and Deming Chen. 2017. Machine learning on FPGAs to face the IoT revolution. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 894–901.

[53] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[54] 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.

[55] 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.

[56] 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.

[57] 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.