ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation

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@illinois.edu, callie.hao@gatech.edu, jianyi.cheng17@imperial.ac.uk, hyunmin2@illinois.edu,
jackh4@illinois.edu, stephenn@xilinx.com, dchen@illinois.edu

***Abstract*—High-level synthesis (HLS) has been widely adopted as it significantly improves the hardware design productivity and enables efficient design space exploration (DSE). Existing HLS tools are built using compiler infrastructures largely based on a single-level abstraction, such as LLVM. However, as HLS designs typically come with intrinsic structural or functional hierarchies, different HLS optimization problems are often better solved with different levels of abstractions. This paper proposes *ScaleHLS*[1], a new scalable and customizable HLS framework, on top of a multi-level compiler infrastructure called MLIR. ScaleHLS represents HLS designs at multiple representation levels and provides an HLS-dedicated analysis and transform library to solve the optimization problems at the suitable levels. Using this library, we provide a DSE engine to generate optimized HLS designs automatically. 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 an end-to-end compilation flow. Experimental results show that, comparing to the baseline designs without manual directives insertion and code-rewriting, that are only optimized by Xilinx Vivado HLS, ScaleHLS improves the performances with amazing quality-of-results – up to 768.1$\times$ better on computation kernel level programs and up to 3825.0$\times$ better on neural network models.**

***Keywords*-High-Level Synthesis; MLIR; Compiler; FPGA; Optimization; Design Space Exploration;**

I. 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 [1], [2]. In recent years, HLS has been widely used in many application developments, such as neural networks [3], [4], IoT applications [5]–[7], and video processing [8]. Existing algorithmic HLS tools typically focus on extracting parallelism from algorithmic descriptions and compiling the result into a parallel hardware execution model [9], [10]. Thus, HLS tools would enable a designer to implement different algorithmic choices quickly, identify high-level area-performance tradeoffs, and avoid premature optimizations [11]. 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 [12]. 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' directives to achieve good design quality [13].

In recent years, we have witnessed many studies for investigating different design space exploration (DSE) methods of setting HLS directives [12]. These efforts can be classified into two main types of methods: synthesis-based and model-based. Synthesis-based methods [13]–[16] invoke downstream HLS tools to evaluate the quality of result (QoR), including the latency, throughput, and resource utilization, of discovered design points. Model-based methods [17]–[21] 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 [22]–[25] 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 [14], linear programming [20], or other dedicated heuristics [13], [16], etc. Apart from different DSE methods, some other studies [9], [26], [27] leverage parallel-programming languages, such as CUDA [28], as inputs to expose the parallelism of the accelerator designs and generate synthesizable C code with HLS directives inserted.

However, we find that existing research 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) [29], traditional software compiler intermediate representation (IR) [30], or C/C++ source-level IR [31], [32], 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.

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

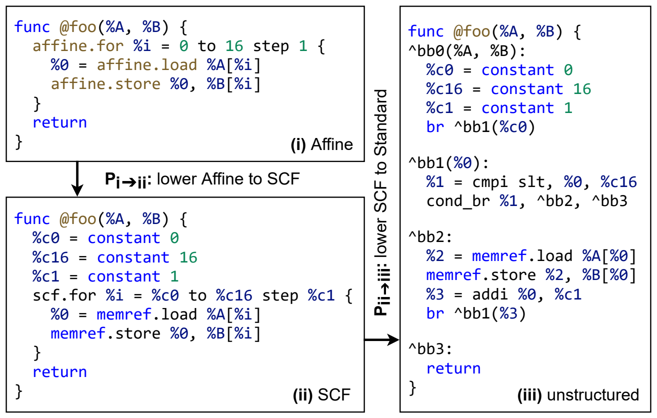
For example, task/module level parallelization should be applied on high-level operators, such as convolution operators, rather than nested loops to avoid conservative assumption and sophisticated memory dependency analysis. Directly combining different levels of representation from different frameworks 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.

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 DSE. 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. 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 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 an HLS-dedicated transform and analysis library, which 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 providing an HLS C front-end and a synthesizable HLS C/C++ emission back-end for bridging the gap between the multi-level IR and C-based HLS designs, thus enabling an end-to-end HLS compilation flow.


```
func @foo(%A, %B) {
  affine.for %i = 0 to 16 step 1 {
    %0 = affine.load %A[%i]
    affine.store %0, %B[%i]
  }
  return
}
```
```
func @foo(%A, %B) {
  %c0 = constant 0
  %c16 = constant 16
  %c1 = constant 1
  scf.for %i = %c0 to %c16 step %c1 {
    %0 = memref.load %A[%i]
    memref.store %0, %B[%i]
  }
  return
}
```
```
func @foo(%A, %B) {
^bb0(%A, %B):
  %c0 = constant 0
  %c16 = constant 16
  %c1 = constant 1
  br ^bb1(%c0)

^bb1(%0):
  %1 = cmpi slt, %0, %c16
  cond_br %1, ^bb2, ^bb3

^bb2:
  %2 = memref.load %A[%0]
  memref.store %2, %B[%0]
  %3 = addi %0, %c1
  br ^bb1(%3)

^bb3:
  return
}
```
(i) Affine
$P_{i\rightarrow ii}$: lower Affine to SCF
(ii) SCF
$P_{ii\rightarrow iii}$: lower SCF to Standard
(iii) unstructured

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

The remaining of this paper is organized as follows. Section II introduces the background. In Section III, we provide an overview of the ScaleHLS framework. In Sections IV and V, we introduce the details of the multi-level representation and optimization for HLS designs, respectively. In Section VI, we present the front-end and back-end integration of ScaleHLS. In Sections VII and VIII, we provide the evaluation results and conclude this paper, respectively.

II. Background

A. MLIR Framework

ScaleHLS is built on top of MLIR [33], [34], 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 [35] where an *Operation* is the minimal unit of code. Each operation accepts a set of typed *Operand*s and produces a set of typed *Result*s. Connections between the results of one operation and the operands of another operation describe the SSA-style flow of data. For instance, `%3 = addi %0, %c1` in Fig. 1(iii) is an operation with operands `%0` and `%c1` and result `%3`. Each operation can also be parameterized by a set of *Attribute*s 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* 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. In MLIR, *Function* is defined as a built-in callable operation always owning one region. For instance, function `@foo` in Fig. 1(iii) owns one region containing four blocks, `bb0` to `bb3`.

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 to define new dialects at different abstraction levels. *Pass* is a key component of compiler which traverses the IR for the purpose of optimization or analysis. Similar to LLVM [30], 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 IR.

B. 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 in order to make dependence analysis and loop transformations efficient and reliable. The `affine` dialect defines *Affine Map* as a mathematical function that transforms a list of affine values into a list of results. Affine operations (e.g., `affine.for` and `if`) 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 basic arithmetic operations (e.g., `addf`) and unstructured

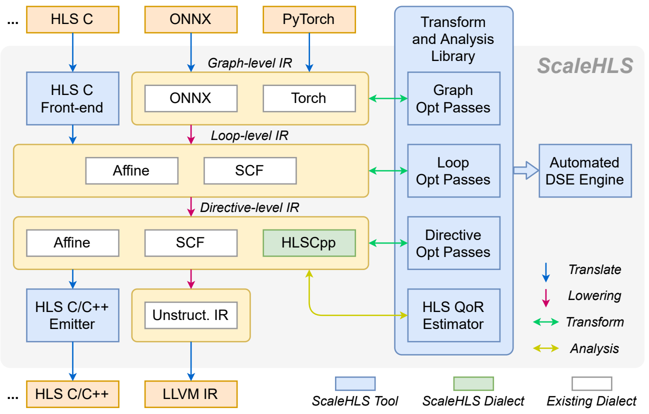


Figure 2. ScaleHLS framework.

control flow operations (e.g., `br` and `cond_br`). Taking Fig. 1 as example, the structured control flows in Fig. 1(i) and (ii) represented with `affine` and `scf` operations are flattened to the unstructured `br` and `cond_br` operations in Fig. 1(iii).

C. Relevant MLIR Front-ends

ScaleHLS uses existing third-party front-ends, Torch-MLIR [36] and ONNX-MLIR [37], to parse PyTorch [38] and ONNX [39] models, respectively. Torch-MLIR first translates PyTorch models into `torch` dialect, then lowers the IR to `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.

III. SCALEHLS FRAMEWORK OVERVIEW

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

A. Representation

Graph-level IR (Section IV-A) exploits existing third-party `onnx` [37] dialect to represent computation graphs constructed with tensor operations. **Loop-level IR** (Section IV-B) exploits MLIR built-in `affine` and `scf` dialects to represent the loop structures and leverage the powerful loop transformation and analysis infrastructures of MLIR. **Directive-level IR** (Section IV-C) is enabled by our customized `hlscpp` dialect to represent the HLS-specific structures and program directives, which provides the capability of conducting directive optimizations and supports the emission of synthesizable C/C++ code.

B. Optimization

On each level of IR, we have designed a set of **Optimization passes** (Section V-A to V-D) to improve the HLS design quality automatically. The hierarchical IR allows the passes to be applied at the most suitable abstraction level, thereby minimizing the processing complexity and improving the scalability. In order to efficiently explore the large design spaces brought by large HLS designs, we propose a fast **HLS QoR estimator** (Section V-E1) based on analytical models, which can estimate the latency and resource utilization of programs on top of the structured directive-level IR.

C. Exploration

The interfaces of the QoR estimator and transform passes of every abstraction level are packaged into an **HLS transform and analysis library** (Section V). All the interfaces in the library are highly parameterized and can be tuned by DSE engines. This library turns the HLS optimization techniques from manual code rewriting to callable and tunable interfaces at different abstraction levels. Leveraging the HLS transform and analysis library, we have designed an **Automated DSE engine** (Section V-E2) to search for the Pareto frontier of the multi-dimensional design space, where each dimension corresponds to a tunable parameter of a transform pass. The DSE engine is extensible to support different optimization algorithms in the future.

D. Integration

We have implemented an **HLS C front-end** (Section VI-A) based on Clang that directly translates input C programs into the `scf` dialect. An `scf` to `affine` raising pass identifies affine regions and converts `scf` operations to their corresponding `affine` operations automatically, which enables subsequent affine transformations and analyses. In the end of compilation, the structured directive-level IR is translated into synthesizable C++ code by an **HLS C/C++ emitter** (Section VI-B). Meanwhile, LLVM IR [30] can also be generated, enabling software simulation and direct interfacing with existing LLVM-compatible tools, such as Xilinx Vitis HLS [40].

IV. ScaleHLS Representation

ScaleHLS features an 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 introduce the graph, loop, and directive level IRs of ScaleHLS in detail.

A. Graph-level IR

ScaleHLS exploits existing third-party `onnx` dialect from ONNX-MLIR [37] to represent and optimize graph-level IR. The assembly form of an `onnx.Conv` operation is (attributes and non-tensor operands are omitted for simplicity):

Table I
Supported HLS directives.

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

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

where the matrix operands and result are typed as tensors. Operations of this dialect 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 analyses and transformations, such as graph node merging, can be easily supported in a graph-level IR by manipulating tensor operations. The graph-level transformations implemented in ScaleHLS are discussed further in Section V-A.

B. 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 Fig. 5 shows the loop-level IR of an SYRK (symmetric rank-k update of a matrix) computation kernel [41] in MLIR where types and attributes of all operations are omitted for simplicity. Memory access and arithmetic operations are nested in `affine.for` operations, which explicitly represent the loop structures. Similarly, the code block (iii) of Fig. 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 analyses and transformations on `affine` operations. The loop-level optimizations are discussed in detail in Section V-B.

C. HLS Directives

HLS tools typically use program directives to guide the hardware generation and fine-tune the latency-area tradeoff. In this section, we introduce how ScaleHLS represents the function, loop, and array HLS directives shown in Tab. I.

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 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 can accept and process 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 `struct` MLIR attribute named `FuncDirective` in `hlscpp` dialect. The customized attribute contains two Boolean parameters, `dataflow` and `pipeline`, and one integer parameter, `targetII`, which triggers the generation of appropriate directives compatible with downstream HLS tools, such as Xilinx Vivado HLS [42]. In ScaleHLS, the function *inline* directive is not explicitly represented with an MLIR attribute, but instead directly inlines the target function in the IR to ease the transformation and analysis.

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 hierarchy, which helps to further improve the pipeline 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 corresponding loop transformation on the target loops in the IR, which is semantically equivalent to applying the directives.

3) Array Partition: Array partition is one of the most important HLS directives because HLS designs require enough on-chip memory bandwidth to comply with the computation parallelism. However, single on-chip memory block has limited read/write ports and hence 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*). Fig. 3 shows three 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 II-B, 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 has 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 Fig. 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.

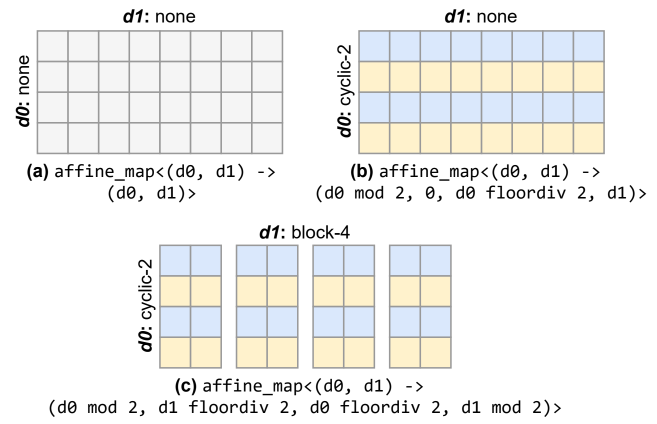

d1: none
d0: none
(a) affine_map<(d0, d1) -> (d0, d1)>
d1: none
d0: cyclic-2
(b) affine_map<(d0, d1) -> (d0 mod 2, 0, d0 floordiv 2, d1)>
d1: block-4
d0: cyclic-2
(c) affine_map<(d0, d1) -> (d0 mod 2, d1 floordiv 2, d0 floordiv 2, d1 mod 2)>

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.

By encoding the partition information into memory types, ScaleHLS can flexibly support different partition fashions, and quickly infer the partition index and physical index through affine expression composition. This technique is used in the QoR estimator (Section V-E1) and the `-array-partition` pass (Section V-C2). Note that unsupported memory partition fashions by the downstream HLS tools are disallowed in the directive-level IR of ScaleHLS.

4) Array Resource and Interface: The HLS-based accelerators can use different kinds of memories, including on-chip memories (e.g., BRAM) 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

Table II
SCALEHLS 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 band	tile-sizes
	-affine-loop-unroll	loop	unroll-factor
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	-

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

software, where BRAM and DRAM respond to 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 [43] or naive BRAM interface) of the array according to its memory space.

V. 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 are listed in Tab. II, together with their transform targets (e.g., function) and the tunable parameters, where a *Loop Band* refers to a continuous set of loops. These passes traverse the whole 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 the QoR estimator and the automated DSE algorithm. In addition, downstream HLS tools are observed unpredictable: changing parameters in the program that should improve performance can counter intuitively yield slower and larger designs [44]. ScaleHLS deals with this problem with predictable transform passes and the integrated QoR estimator, which will be elaborated further in this section.

A. Graph Transform Passes

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 executions of sub-functions are not allowed [42]. 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 and unpredictable since careless rewriting can easily result in incorrect functionality or undesired effect in terms of performance and resource utilization 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. Previous work [45] proposes an automatic method to enable thread-level dataflow on GPGPUs, yet the dataflow issue in HLS designs has not been well-studied.

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. Fig. 4(a) shows an example dataflow containing five procedures, where each edge corresponds to a tensor delivering. We can observe that Fig. 4(a) is illegal as there is a path between *Proc0* and *Proc3* bypassing *Proc1-2*. This dataflow can be conservatively legalized to Fig. 4(b) through the `-legalize-dataflow` pass. To eliminate the bypass path, *Proc1-3* are organized into the same dataflow stage, thereby *Proc0*, *Proc1-3*, and *Proc4* can construct a 3-stages dataflow. Note that in Fig. 4(b), the output buffers of *Proc0* and *Proc3* are automatically double buffered after the directive is successfully applied, with the cost of utilizing more memory resources than the original dataflow in Fig. 4(a).

Alternatively, the dataflow can be aggressively legalized to Fig. 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 proce-

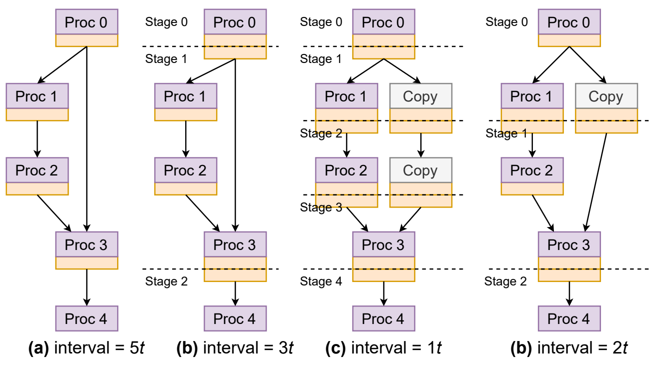


Figure 4. Graph-level dataflow optimization. (a) original dataflow; (b) legalized dataflow without copy nodes; (c) legalized dataflow with inserting copy nodes; (d) dataflow with a minimum granularity of 2.

dure 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, ScaleHLS will provide such diagnostics back to users and avoid unpredictable design transforms to be applied.

2) Split Function: Once the dataflow is legalized, the original function can be splitted into a top function and multiple sub-functions by a `-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 a throughput-area tradeoff space can be explored by merging adjacent dataflow stages into one. For example, in Fig. 4(d), every two adjacent stages are merged together, constructing a new 3-stages dataflow with less resource utilization compared to Fig. 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.

B. Loop Transform Passes

We use the SYRK computation kernel shown in Fig. 5 as the example in the following discussion. In this section, we introduce the loop transform passes provided by ScaleHLS, which corresponds to the $P_{ii \rightarrow iii}$ transformation of Fig. 5.

1) Loop Perfectization: Operations between loop statements, such as Fig. 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.

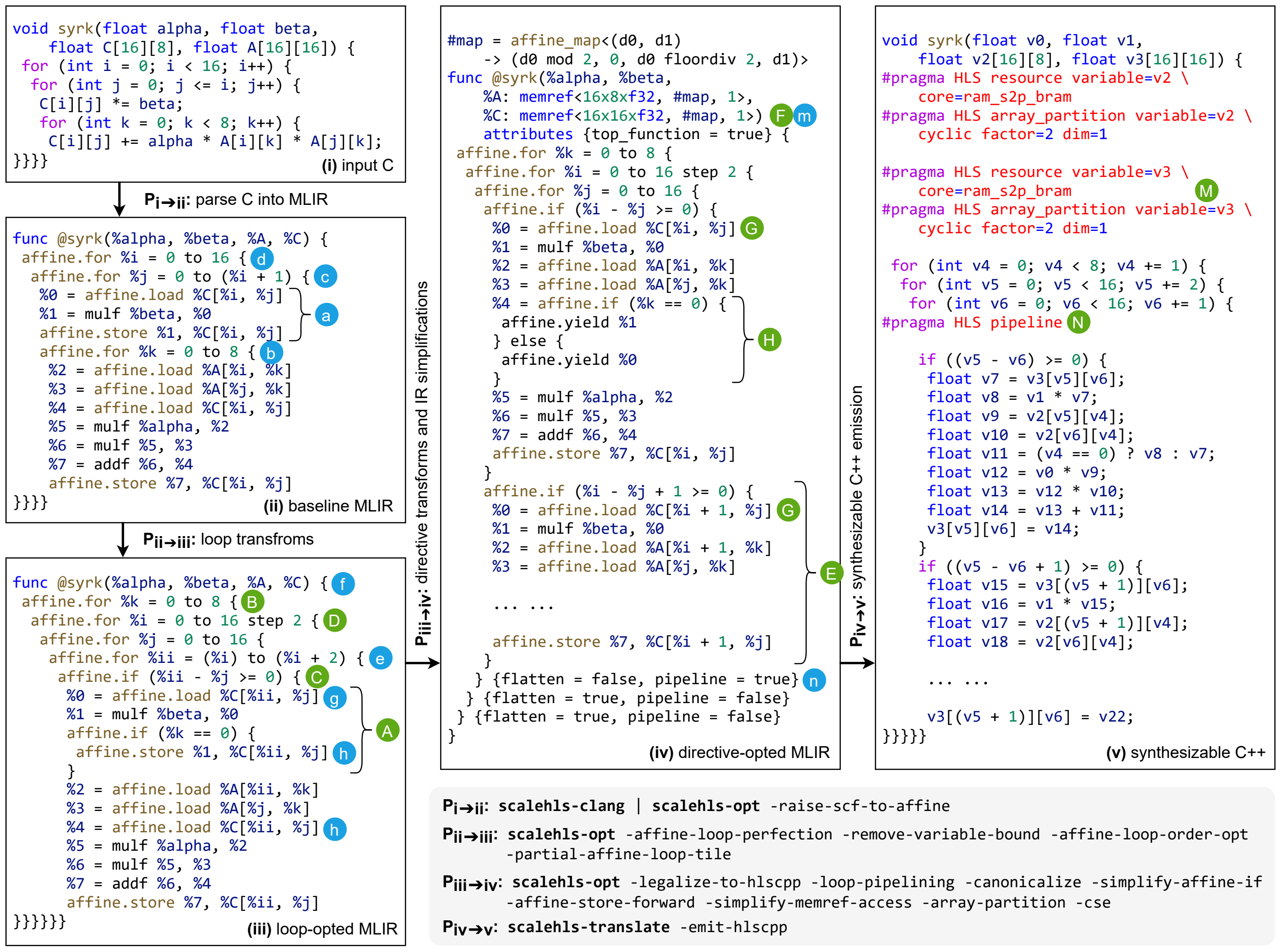
2) Loop Order Optimization: Loop permutation can change the distance of loop-carried memory dependencies, thereby reducing the achievable II of loop pipelining and reducing latency. The `-affine-loop-order-opt` pass can automatically perform affine-based memory dependency analysis and apply the best legal loop order to the targeted loop band. Specifically, loops associated with loop-carried dependencies are permuted to the outside in order to increase the distance of the dependencies. 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 loop band, where positions are from the outermost loop to inner loops.

3) Remove Variable Loop Bound: Because MLIR focuses on rectangular iteration spaces, there are limitations on analyzing 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 constants and an `affine.if` operation (5Ⓒ) is generated in the innermost loop for the conditional execution of the 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.

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 through affine analysis before the transform is applied. The tiling size is determined by a `tile-size` parameter which can be tuned by the DSE engine.

C. Directive Transform Passes

In this section, we introduce the directive-level transform passes of ScaleHLS, which manipulate HLS-specific direc-



```
void syrk(float alpha, float beta,
    float C[16][8], float A[16][16]) {
 for (int i = 0; i < 16; i++) {
  for (int j = 0; j <= i; j++) {
   C[i][j] *= beta;
   for (int k = 0; k < 8; k++) {
    C[i][j] += alpha * A[i][k] * A[j][k];
}}}}
```

```
func @syrk(%alpha, %beta, %A, %C) {
 affine.for %i = 0 to 16 {
  affine.for %j = 0 to (%i + 1) {
   %0 = affine.load %C[%i, %j]
   %1 = mulf %beta, %0
   affine.store %1, %C[%i, %j]
   affine.for %k = 0 to 8 {
    %2 = affine.load %A[%i, %k]
    %3 = affine.load %A[%j, %k]
    %4 = affine.load %C[%i, %j]
    %5 = mulf %alpha, %2
    %6 = mulf %5, %3
    %7 = addf %6, %4
    affine.store %7, %C[%i, %j]
}}}}
```

```
func @syrk(%alpha, %beta, %A, %C) {
 affine.for %k = 0 to 8 {
  affine.for %i = 0 to 16 step 2 {
   affine.for %j = 0 to 16 {
    affine.for %ii = (%i) to (%i + 2) {
     affine.if (%ii - %j >= 0) {
      %0 = affine.load %C[%ii, %j]
      %1 = mulf %beta, %0
      affine.if (%k == 0) {
       affine.store %1, %C[%ii, %j]
      }
      %2 = affine.load %A[%ii, %k]
      %3 = affine.load %A[%j, %k]
      %4 = affine.load %C[%ii, %j]
      %5 = mulf %alpha, %2
      %6 = mulf %5, %3
      %7 = addf %6, %4
      affine.store %7, %C[%ii, %j]
}}}}}}
```

```
#map = affine_map<(d0, d1)
    -> (d0 mod 2, 0, d0 floordiv 2, d1)>
func @syrk(%alpha, %beta,
    %A: memref<16x8xf32, #map, 1>,
    %C: memref<16x16xf32, #map, 1>)
    attributes {top_function = true} {
 affine.for %k = 0 to 8 {
  affine.for %i = 0 to 16 step 2 {
   affine.for %j = 0 to 16 {
    affine.if (%i - %j >= 0) {
     %0 = affine.load %C[%i, %j]
     %1 = mulf %beta, %0
     %2 = affine.load %A[%i, %k]
     %3 = affine.load %A[%j, %k]
     %4 = affine.if (%k == 0) {
      affine.yield %1
     } else {
      affine.yield %0
     }
     %5 = mulf %alpha, %2
     %6 = mulf %5, %3
     %7 = addf %6, %4
     affine.store %7, %C[%i, %j]
    }
    affine.if (%i - %j + 1 >= 0) {
     %0 = affine.load %C[%i + 1, %j]
     %1 = mulf %beta, %0
     %2 = affine.load %A[%i + 1, %k]
     %3 = affine.load %A[%j, %k]

     ... ...

     affine.store %7, %C[%i + 1, %j]
    }
   } {flatten = false, pipeline = true}
  } {flatten = true, pipeline = false}
 } {flatten = true, pipeline = false}
}
```

```
void syrk(float v0, float v1,
    float v2[16][8], float v3[16][16]) {
#pragma HLS resource variable=v2 \
    core=ram_s2p_bram
#pragma HLS array_partition variable=v2 \
    cyclic factor=2 dim=1

#pragma HLS resource variable=v3 \
    core=ram_s2p_bram
#pragma HLS array_partition variable=v3 \
    cyclic factor=2 dim=1

 for (int v4 = 0; v4 < 8; v4 += 1) {
  for (int v5 = 0; v5 < 16; v5 += 2) {
   for (int v6 = 0; v6 < 16; v6 += 1) {
#pragma HLS pipeline

    if ((v5 - v6) >= 0) {
     float v7 = v3[v5][v6];
     float v8 = v1 * v7;
     float v9 = v2[v5][v4];
     float v10 = v2[v6][v4];
     float v11 = (v4 == 0) ? v8 : v7;
     float v12 = v0 * v9;
     float v13 = v12 * v10;
     float v14 = v13 + v11;
     v3[v5][v6] = v14;
    }
    if ((v5 - v6 + 1) >= 0) {
     float v15 = v3[(v5 + 1)][v6];
     float v16 = v1 * v15;
     float v17 = v2[(v5 + 1)][v4];
     float v18 = v2[v6][v4];

     ... ...

     v3[(v5 + 1)][v6] = v22;
}}}}}
```

```
Pi→ii: scalehls-clang | scalehls-opt -raise-scf-to-affine
Pii→iii: scalehls-opt -affine-loop-perfection -remove-variable-bound -affine-loop-order-opt
         -partial-affine-loop-tile
Piii→iv: scalehls-opt -legalize-to-hlscpp -loop-pipelining -canonicalize -simplify-affine-if
         -affine-store-forward -simplify-memref-access -array-partition -cse
Piv→v: scalehls-translate -emit-hlscpp
```

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.

tives to further improve the design quality. The effect of the discussed passes are showcased in the $P_{iii \to iv}$ transformation of Fig. 5.

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 [42]. 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 succeeds, loop pipeline directive is applied to the target loop with the specified II. 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 will recognize and diagnose illegal transform targets to avoid unpredictable compilation and allow specifying the targeted II for exploring the tradeoff design space between throughput and resource utilization.

2) Array Partition: ScaleHLS enhances the method proposed in [19] to automatically detect the memory access pattern of a program and apply the suitable array partition factor and fashion to each dimension of each array. The array partition metric P of the d-th dimension of the i-th array can be represented with:

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

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

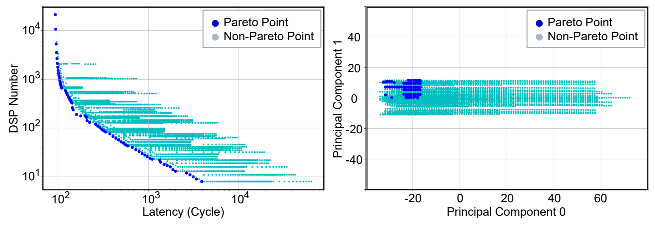


Figure 6. Design space profiling of a GEMM kernel. (a) the latency-area space; (b) PCA of the multi-dimensional design space.

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 accessed by sub-functions, an inter-procedural 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.

D. IR Redundancy Elimination

In addition to the graph, loop, and directive transforms, ScaleHLS adopts the methodology described in [46] and provides multiple passes to remove the redundant operations in the IR. The `-simplify-affine-if` pass eliminates dead branches of `affine.if` operations by detecting always-true/false conditions using affine analysis. The `-affine-store-forward` pass eliminates redundant memory access operations and unused memory instances through store-to-load forwarding. The `-simplify-memref-access` pass folds identical memory access operations if no dependency conflict exists. 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) [34], to eliminate the redundancies in the IR and further optimize the quality of the HLS design.

E. Automatic Design Space Exploration

On top of the representation and optimization of ScaleHLS, we can construct a multi-dimensional design space, where each dimension corresponds 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.

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 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 are viewed as nodes in the dependency graph. By accurately modeling the latency and resource utilization, ScaleHLS can estimate the effect of design transforms in the early stage of compilation, which makes the HLS optimizations more predictable.

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 Tab. II, 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. The design space profiling of a GEMM (general matrix multiply) kernel [41] is shown in Fig. 6, where we employ principle component analysis (PCA) for dimensionality reduction and exploratory analysis. We can observe that the Pareto points (deep blue) in the latency-area space are clustered in the PCA space. For instance, 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 5-step neighbor-traversing algorithm to explore the design space.

In **Step1 Initial Sampling**, we randomly sample the whole design space and evaluate the latency and resource utilization of each sampled design point using the QoR estimator. Then, the initial Pareto frontier is extracted from all sampled points. In **Step2 Point Proposal**, we randomly select a design point in the current Pareto frontier and propose its closest neighbor as the new point to evaluate. This point proposal method can effectively inherit beneficial optimization parameters from evaluated design points. In **Step3 Point Evaluation**, we call the QoR estimator to evaluate the new design point proposed by Step2. The current Pareto frontier is then updated if any point in the current frontier is dominated by the new design point. In **Step4 Frontier Evolution**, we repeat Step2 and Step3 until no eligible neighbor can be found or meeting the early-termination criteria (e.g., maximum iteration number). In this procedure, the discovered Pareto frontier evolves in each iteration and progressively approaches the "real" Pareto frontier. In **Step5 Design Finalization**, we sort the discovered Pareto points in ascending order of latency and then select the first point meeting the resource constraints as the final solution. The finalized design is emitted as synthesizable C++ code. This DSE algorithm is implemented as an MLIR transform pass called `-dse` which can be applied on the input HLS designs without any manual efforts. The evaluation of the DSE engine is performed in Section VII. Note that given the HLS transform and analysis library of ScaleHLS, the DSE engine is extensible to support different optimization algorithms.

VI. End-to-End Integration

A. 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 an `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 $P_{i \rightarrow ii}$ transformation of Fig. 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` is a set of exclusive pointers to the memory and size parameters of the memory [33]. The `memref` type solves delinearization problem of parametrically sized arrays, which was not well-supported in LLVM [47]. The translation to `memref` is simplified in our front-end because common HLS tools, such as Vivado HLS, only accepts a subset of C [42]. For instance, all the arrays have to have fixed sizes. 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.

B. 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 $P_{iv \rightarrow v}$ transformation of Fig. 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++, the emitter always converts returned values to input pointers.

VII. 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 reported by Vivado HLS.

A. Large-Scale Computation Kernels

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 [48] 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 optimized solutions. Finally, the generated designs are evaluated and the results are shown in Tab. III. 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 or manual code rewriting. Tab. III also lists the parameters selected by DSE 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 solutions discovered by the DSE engine, we find the performance gains come from multiple sources: (1) loop perfectization and variable loop bound removal 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 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

Table III
DSE RESULTS OF LARGE-SCALE COMPUTATION KERNELS.

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]

The data types of all kernels are 32-bits 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.

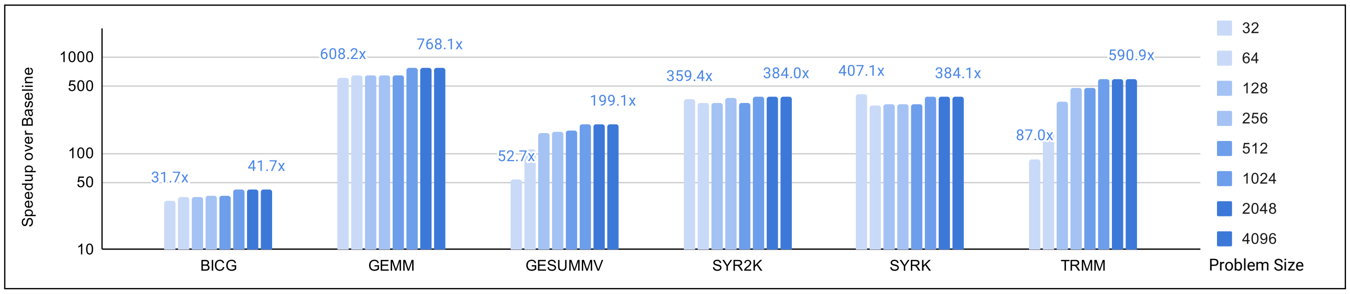


Figure 7. Scalability study of computation kernels. The problem sizes of computation kernels are scaled from 32 to 4096 and the DSE engine is launched to search for the optimized solutions under each problem size.

Table IV
CASE STUDY OF GEMM KERNEL WITH A PROBLEM SIZE OF 4096.

Design	Cycles	Speedup	DSP (Util. %)
Unoptimized	1.237×10^{12}	1.0×	5 (2.3%)
DSE Optimized	1.610×10^{9}	768.1×	217 (98.6%)
Manually Optimized	2.684×10^{9}	460.9×	220 (100.0%)
Theoretical Bound	1.562×10^{9}	791.9×	220 (100.0%)

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 improvements under the constrained on-chip resources available on the edge FPGA platform.

To better understand the quality of solutions found by the DSE, we perform a case study on the GEMM kernel. We manually implement an HLS design on the same FPGA platform using a rich set of directives as well as code-rewriting driven by human design experience and expertise. We also calculate the best achievable latency on the targeted FPGA by assuming all DSPs on chip run without any stall and the kernel can be perfectly parallelized. The case study results are shown in Tab. IV. We can observe that the HLS design generated by our DSE achieves 0.97× of the theoretical bound and is around 1.67× better than the manually optimized HLS design. Note that the DSE only takes minutes to find the design, while the manual design takes us about 10 hours to finish.

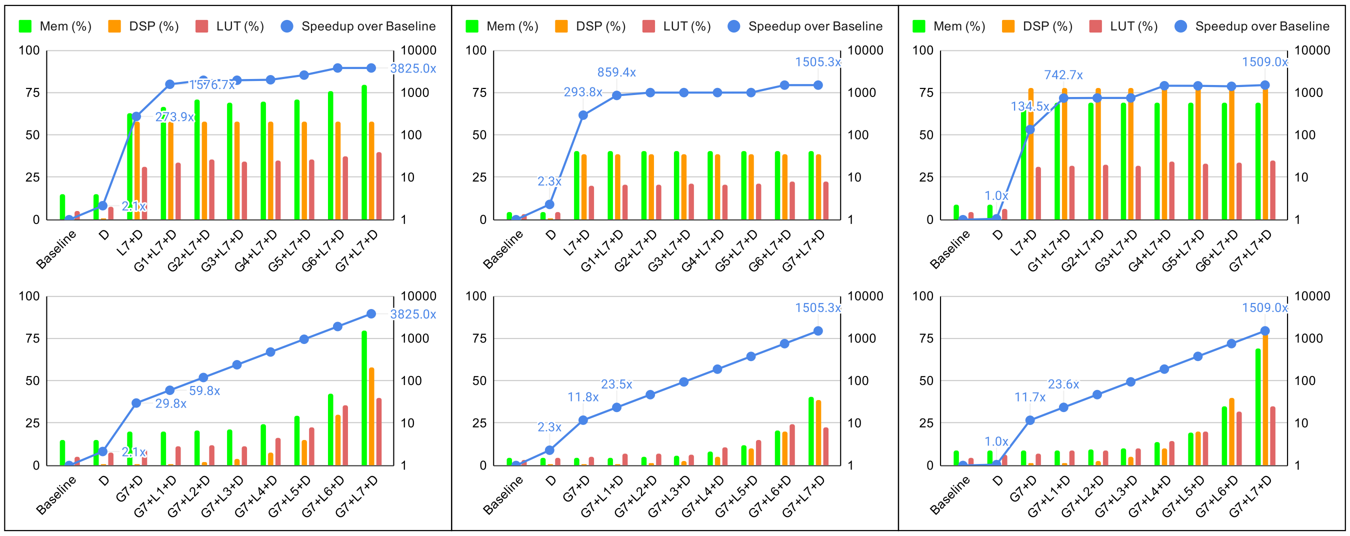
2) Comparison with Previous Works: Previous efforts [17]–[19] also investigate automatic DSE methods to optimize computation kernel level algorithms. However, they only support directive optimizations, thus are difficult to comprehensively explore the design space and find reasonable design points when the problem sizes are large. For example, on the six scaled-up benchmarks shown in Tab. III, the open-sourced framework [19] 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 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.

3) Scalability Study: To understand the performance of our framework on different problem sizes, we scale the

Table V
OPTIMIZATION RESULTS OF REPRESENTATIVE DNN MODELS.

Model	Speedup	Runtime (seconds)	Memory (SLR Util. %)	DSP (SLR Util. %)	LUT (SLR Util. %)	Our DSP Effi. (OP/Cycle/DSP)	DSP Effi. of TVM-VTA [49]
ResNet-18	3825.0×	60.8	91.7Mb (79.5%)	1326 (58.2%)	157902 (40.1%)	1.343	0.344
VGG-16	1505.3×	37.3	46.7Mb (40.5%)	878 (38.5%)	88108 (22.4%)	0.744	0.296
MobileNet	1509.0×	38.1	79.4Mb (68.9%)	1774 (77.8%)	138060 (35.0%)	0.791	0.468

Speedup is with respect to the baseline designs compiled from PyTorch by ScaleHLS but without the multi-level optimization.



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

Figure 8. 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.

problem sizes of the six benchmarks from 32 to 4096 and launch the DSE engine to search for the optimized solution under each setting. Fig. 7 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 on a wide range of problem sizes.

B. Large and Complicated Algorithms

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 [50] image classification task, ResNet-18 [51], VGG-16 [52], and MobileNet [53]. 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 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 Tab. V. We can observe that by combining 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 Tab. V, ScaleHLS only consumes 37.3 to 60.8 seconds to optimize the large 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 the MLIR framework.

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 [17]–[19] focus on small-scale algorithms, while compilation flows dedicated for DNNs rely on pre-defined IP libraries [49], [54], [55] or parameterized templates [56], [57] to generate the accelerator, which can not be generalized to applications other than DNNs. To better understand the optimization results of DNN models, we compare the DSP efficiency with TVM-VTA [49], a widely accepted DNN acceleration framework written in HLS. DSP efficiency is a common metric for comparing the efficiency of DNN accelerators across different platforms, which can be calculated as:

$$Effi_{DSP} = \frac{OP/s}{Num_{DSP} \times Freq} \tag{2}$$

As shown in Tab. V, 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 large and complicated algorithms.

3) Ablation Study: To quantify the speedup contributed by each of the three optimizations (directive, loop, and graph), we perform an ablation study shown in Fig. 8. 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 is 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.

VIII. Conclusion

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 optimized 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 hardware IPs into the compilation flow, making the integration and optimization of IPs an interesting research direction. (2) DSE algorithms. The transform and analysis library provided by ScaleHLS enables a great opportunity to further 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) RTL code generation within MLIR. Currently ScaleHLS leverages external HLS tools for generating the RTL code. However, a direct RTL code generation within MLIR can keep more information from the higher level IR and exploit the RTL-level representation and optimization (CIRCT [58]) to further improve the quality of the accelerator designs.

Acknowledgements

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] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *2011 9th IEEE International Conference on ASIC*. IEEE, 2011, pp. 1102–1105.

[2] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," *ArXiv e-prints*, May 2018.

[3] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "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*, 2019, pp. 73–82.

[4] X. Zhang, H. Lu, C. Hao, J. Li, B. Cheng, Y. Li, K. Rupnow, J. Xiong, T. Huang, H. Shi *et al.*, "Skynet: a hardware-efficient method for object detection and tracking on embedded systems," *arXiv preprint arXiv:1909.09709*, 2019.

[5] D. Chen, J. Cong, S. Gurumani, W.-m. Hwu, K. Rupnow, and Z. Zhang, "Platform choices and design demands for iot platforms: cost, power, and performance tradeoffs," *IET Cyber-Physical Systems: Theory & Applications*, vol. 1, no. 1, pp. 70–77, 2016.

[6] Z. Zhang, D. Chen, S. Dai, and K. Campbell, "High-level synthesis for low-power design," *IPSJ Transactions on System LSI Design Methodology*, vol. 8, pp. 12–25, 2015.

[7] X. Zhang, A. Ramachandran, C. Zhuge, D. He, W. Zuo, Z. Cheng, K. Rupnow, and D. Chen, "Machine learning on fpgas to face the iot revolution," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 894–901.

[8] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen, "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*, 2016, pp. 224–233.

[9] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "Fcuda: Enabling efficient compilation of cuda kernels onto fpgas," in *2009 IEEE 7th Symposium on Application Specific Processors*. IEEE, 2009, pp. 35–42.

[10] A. Papakonstantinou, Y. Liang, J. A. Stratton, K. Gururaj, D. Chen, W.-M. W. Hwu, and J. Cong, "Multilevel granularity parallelism synthesis on fpgas," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 178–185.

[11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[12] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2628–2639, 2019.

[13] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "Autodse: Enabling software programmers design efficient fpga accelerators," *arXiv preprint arXiv:2009.14381*, 2020.

[14] B. C. Schafer, T. Takenaka, and K. Wakabayashi, "Adaptive simulated annealer for high level synthesis design space exploration," in *2009 International Symposium on VLSI Design, Automation and Test*. IEEE, 2009, pp. 106–109.

[15] A. Cilardo and L. Gallo, "Interplay of loop unrolling and multidimensional memory partitioning in hls," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 163–168.

[16] L. Ferretti, G. Ansaloni, and L. Pozzi, "Lattice-traversing design space exploration for high level synthesis," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 210–217.

[17] W. Zuo, W. Kemmerer, J. B. Lim, L.-N. Pouchet, A. Ayupov, T. Kim, K. Han, and D. Chen, "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, 2015, pp. 357–364.

[18] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: a high-level performance analysis tool for fpga-based accelerators," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[19] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "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, 2017, pp. 430–437.

[20] W. Zuo, L.-N. Pouchet, A. Ayupov, T. Kim, C.-W. Lin, S. Shiraishi, and D. Chen, "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*, 2017, pp. 1–6.

[21] S. Wang, Y. Liang, and W. Zhang, "Flexcl: An analytical performance model for opencl workloads on flexible fpgas," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[22] K. O'Neal, M. Liu, H. Tang, A. Kalantar, K. DeRenard, and P. Brisk, "Hlspredict: Cross platform performance prediction for fpga high-level synthesis," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[23] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang, "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, 2018, pp. 129–132.

[24] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad, "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, 2019, pp. 397–403.

[25] N. Wu, Y. Xie, and C. Hao, "Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning," *arXiv preprint arXiv:2102.08138*, 2021.

[26] K. Shagrithaya, K. Kepa, and P. Athanas, "Enabling development of opencl applications on fpga platforms," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 2013, pp. 26–30.

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

[28] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[29] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.

[30] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 2004, pp. 75–86.

[31] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[32] S. Lee and J. S. Vetter, "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*, 2014, pp. 115–120.

[33] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[34] M. contributors, "Mlir: Multi-level intermediate representation," https://github.com/llvm/llvm-project/tree/main/mlir, 2021.

[35] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[36] T.-M. contributors, "Torch-mlir: Mlir based compiler toolkit for pytorch programs," https://github.com/llvm/torch-mlir, 2021.

[37] T. D. Le, G.-T. Bercea, T. Chen, A. E. Eichenberger, H. Imai, T. Jin, K. Kawachiya, Y. Negishi, and K. O'Brien, "Compiling onnx neural network models using mlir," *arXiv preprint arXiv:2008.08272*, 2020.

[38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[39] O. contributors, "Onnx: Open neural network exchange," https://github.com/onnx/onnx, 2021.

[40] X. Inc., "Vitis hls front-end," https://github.com/Xilinx/HLS, 2021.

[41] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[42] X. Inc., *Vitis High-Level Synthesis User Guide: UG1399 (v2020.2)*, 2020.

[43] ——, *Vivado Design Suite AXI Reference Guide: UG1037 (v4.0)*, 2017.

[44] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 393–407.

[45] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," *ACM SIGARCH computer architecture news*, vol. 42, no. 3, pp. 205–216, 2014.

[46] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[47] T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop, "Optimistic delinearization of parametrically sized arrays," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 351–360.

[48] L.-N. Pouchet *et al.*, "Polybench: The polyhedral benchmark suite," https://www.cs.colostate.edu/~pouchet/software/polybench/, 2012.

[49] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Vta: an open hardware-software stack for deep learning," *arXiv preprint arXiv:1807.04188*, 2018.

[50] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[51] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[52] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[53] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[54] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[55] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "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, 2018, pp. 1–8.

[56] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "Hybriddnn: A framework for high-performance hybrid dnn accelerator design and implementation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[57] Y. Meng, S. Kuppannagari, R. Kannan, and V. Prasanna, "Dynamap: Dynamic algorithm mapping framework for low latency cnn inference," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 183–193.

[58] C. contributors, "Circt: Circuit ir compilers and tools," https://github.com/llvm/circt/tree/main/, 2021.