2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 979-8-3503-9313-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPCA57654.2024.00017

An Optimizing Framework on MLIR for Efficient FPGA-based Accelerator Generation

Weichuang Zhang*, Jieru Zhao*†, Guan Shen, Quan Chen, Chen Chen, Minyi Guo†
Department of Computer Science and Engineering, Shanghai Jiao Tong University
{1064080006, zhao-jieru, shenguan}@sjtu.edu.cn, {chen-quan, chen-chen, guo-my}@cs.sjtu.edu.cn

***Abstract*—With the increasing demand for computing capability given limited resource and power budgets, it is prominent to deploy applications to customized accelerators like FPGAs. However, FPGA programming is non-trivial. Although existing high-level synthesis (HLS) tools improve productivity to a certain extent, they are limited in scope and capability to support sufficient FPGA-oriented transformations and optimizations.**

This paper focuses on FPGA-based accelerators and proposes POM, an end-to-end optimizing framework built on multi-level intermediate representation (MLIR). POM has several features which demonstrate its scope and capability of performance optimization. First, most HLS tools depend exclusively on a single-level IR like LLVM IR to perform all the optimizations, introducing excessive information into the IR and making debugging an arduous task. In contrast, POM explicitly introduces three layers of IR to perform operations at suitable abstraction levels, streamlining the implementation and debugging process and exhibiting better flexibility, extensibility, and systematicness. Second, POM integrates the polyhedral model into MLIR and hence enables advanced dependence analysis and a wide range of FPGA-oriented loop transformations. By representing nested loops with integer sets and maps at suitable IR, loop transformations can be conducted conveniently through a series of manipulations on polyhedral semantics. Finally, to further relieve design effort, POM is equipped with a user-friendly programming interface (DSL) that allows a concise description of computation and includes a rich collection of scheduling primitives. An automatic design space exploration (DSE) engine is also provided to search for high-performance optimization schemes efficiently and generate optimized accelerators automatically. Experimental results show that POM achieves a $6.46\times$ average speedup on typical benchmark suites and a $6.06\times$ average speedup on real-world applications compared to the state-of-the-art.

I. Introduction

With the rapid growth of computation-intensive applications, FPGA-based accelerators are becoming increasingly popular to achieve high processing speeds given limited resource and energy budgets. However, programming on FPGAs is challenging. To improve productivity, high-level synthesis (HLS) tools are proposed to synthesize behavioral descriptions specified in high-level languages (such as C/C++) into dedicated hardware accelerators [14]. This allows designers to focus on the behavioral implementation of algorithms without dealing with complex and error-prone digital design. However, there are new challenges to overcome. While existing tools such as Xilinx Vitis HLS offer pragmas to guide the hardware code generation, the quality of generated accelerators largely depends on the user's ability to select appropriate HLS pragmas [38, 39]. In many cases, it is necessary to restructure source code manually to loosen tight dependencies and achieve high parallelism [41]. This process can be complex and iterative, requiring careful consideration and experimentation, which exacerbates programming and optimization difficulties.

*The authors contributed equally to this work.
†Jieru Zhao and Minyi Guo are the corresponding authors.

Recent years have witnessed multiple compilation frameworks that cope with programming and optimization difficulties for different platforms, including ① general processors like CPUs and GPUs, ② specialized processors with predefined hardware like DSAs, and ③ FPGA-based accelerators where data paths can be fully customized and reconfigured. Optimization techniques vary for different hardware and we classify representative frameworks into two categories:

Frameworks for non-FPGA back-ends (①②): There is a growing trend towards optimizing applications written in scheduling languages on CPUs, GPUs [9–11, 13, 26, 29, 36] and specialized processors like DSAs [19, 25, 37]. Halide [29] proposes a domain-specific language (DSL) with decoupled computation and schedule for image processing. TVM [13] extends Halide DSL and proposes an optimizing compiler for deep learning. Both of them mainly focus on GPU acceleration. VTA [25] works as a back-end for TVM and optimizes tensors on an ISA-based DSA processor with predefined architecture and FPGA is utilized for prototyping. This is different from commonly used FPGA-based accelerators where the data path is fully customized and reconfigured. Exo [19] provides an abstract programming model for DSAs, which allows developers to write libraries for emerging accelerators. Simultaneously, *polyhedral techniques* [32] have shown success in efficient loop analysis and transformation. PENCIL [9], Pluto [11], PolyMage [26], Tiramisu [10], and AlphaZ [36] take high-level languages or DSLs as input and generate optimized code automatically for CPUs/GPUs. AKG [37] utilizes polyhedral schedulers and accelerates tensors for NPUs. Despite the good performance, none of these works target FPGA-based accelerators, and their loop optimization strategies cannot be adopted directly to FPGAs. Take a nested loop as an example, their strategies seek to parallelize outer loop levels for multi-thread computation and process inner loop levels sequentially within each thread. For AI applications on GPUs and NPUs, loop fusion is utilized to improve data locality and reduce the cost of kernel launch. In contrast, FPGA-friendly optimization strategies tend to pipeline outer loop levels and parallelize inner loop levels through unrolling, and loop fusion mainly reduces resource usage.

Frameworks for FPGA accelerators with customized data paths (③): Recent advances in HLS frameworks have explored optimization methods for FPGAs [21, 24, 28, 34, 35]. Halide-HLS [28] and HeteroHalide [24] work as FPGA back-ends for Halide and generate customized pipelines for image processing. HeteroCL [21] and HeteroFlow [34] extend TVM DSL and generate spatial architectures on FPGAs. However, these frameworks have limited capabilities in dependence analysis and loop transformation, resulting in a reduced ability to exploit parallelism. All of these works use a single loop-level intermediate representation (IR) for HLS optimization. However, there exist many other schedule methods, and each corresponds to different optimization granularity, which should be applied at or across different abstraction levels for better performance. Depending exclusively on a single IR for all optimizations may introduce excessive information into the IR and make debugging an arduous task. In contrast, using multiple layers of IRs to represent schedule methods streamlines the implementation process of different analysis, transformation, and optimization methods, which exhibits greater flexibility and systematicness and allows for more efficient design space exploration. Most related tools with multi-level IRs target CPUs, GPUs, and specialized processors [10, 12, 17, 26, 37]. ScaleHLS [35] proposes an HLS framework for FPGAs on top of the multi-level intermediate representation (MLIR) compiler infrastructure [23]. It receives C code and expands MLIR with a back-end to generate synthesizable HLS code. However, critical schedule methods and strategies cannot be supported, leading to non-optimal accelerators. Moreover, since the input is C code, designers still need to fully restructure the source code even if the schedule is slightly adjusted.

In this paper, we present *POM*, an open-source optimizing framework on MLIR, which generates efficient FPGA accelerators automatically. POM explicitly divides the compilation process into three layers with hybrid IRs, namely dependence graph IR, polyhedral IR, and annotated affine dialect, which enable various optimizations at appropriate abstraction levels. The dependence graph IR is used for advanced dependence analysis at the graph level of applications. The polyhedral IR is designed to reduce the implementation effort for a wide range of schedule strategies. And the annotated MLIR affine dialect explicitly represents HLS pragmas in loop hierarchies, working as a suitable bridge between polyhedral semantics (the previous layer) and synthesizable HLS code (the output). Our main contributions are summarized as follows:

- **Programmability:** POM provides a decoupled DSL that enables concise descriptions of functions, loops, and arrays. A rich collection of scheduling primitives is provided for flexible customization, leading to much fewer lines of code while maintaining high performance.
- **Extensibility:** POM explicitly introduces three layers of IR to perform operations at suitable abstraction levels in a unified framework, streamlining the implementation and debugging process and reducing the effort of supporting various optimization methods.

```
for (int i = 1; i <= 4; ++i)
  for (int j = 1; j <= 4; ++j)
 S: A[i][j] = A[i-1][j-1] * 2 + 3
```

For iteration vectors:

$\boldsymbol{v_1} = (2,\ 2), \quad \boldsymbol{v_2} = (3,\ 3)$

in iteration domain:

$\{S(i,j) : 1 \leq i,\ j \leq 4\}$

The dependence is represented as:

Distance Vector: $\mathbf{d} = \boldsymbol{v_2} - \boldsymbol{v_1} = (1,\ 1)$

Direction Vector: $\mathbf{D} = (<, <)$

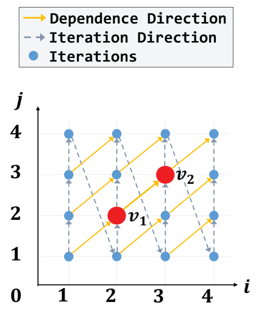


Fig. 1: Illustration of loop dependence analysis

- **Quality:** POM provides a rich set of optimization methods and performs FPGA-oriented schedule operations at proper levels, relieving tight loop-carried dependence, exploiting parallelism, and improving overall performance.
- **Automation:** POM contains a design space exploration (DSE) engine to search for high-performance schedule schemes automatically and efficiently, while also allowing designers to set user-specified schedules.

Experimental results show that POM achieves 6.46× average speedup on typical benchmark suites and 6.06× average speedup on real-world applications compared to SOTA [35].

II. Background and Motivation

A. Polyhedral semantics and dependence analysis

The polyhedral model is a powerful compilation technique that represents programs as polyhedrons. These polyhedrons are analyzed and transformed to facilitate program acceleration. Generally, polyhedral semantics extracted from a nested loop can be expressed as an *iteration domain*, *data accesses*, *schedules*, and *dependencies* [20]. Take the nested loop in Fig. 1 as an example. *Iteration domain* refers to the set of all statement instances satisfying the loop-bound constraints and is visualized in an n-dimensional iteration space. We can represent a statement instance of an n-level loop as an *iteration vector* with n entries, each of which corresponds to each loop iterator, e.g., $v_1 = (2, 2)$ in Fig. 1. *Data access* is the affine representation of memory references and *schedule* determines the execution order of statement instances [16]. *Dependence* reflects the data dependence between statement instances. Given two dependent statement instances, their dependence relationship is represented using *distance vector* and *direction vector*. As shown in Fig. 1, the *distance vector* **d** denotes the distance between the source iteration vector v_1 and the sink iteration vector v_2, and each entry d_k of **d** is computed as $(v_2)_k - (v_1)_k$, i.e., $\mathbf{d} = (1, 1)$. The *direction vector* has three representations at each entry: $<$, $=$, and $>$, based on whether the corresponding entry of the distance vector is larger than, equal to, or less than zero, correspondingly. In this example, the direction vector **D** is $(<, <)$. This dependence relationship implies that there exists a loop-carried dependence between loop iterations, which may impede the potential parallelism, especially for FPGA-based accelerators.

Dependence analysis: POM contains an efficient dependence analysis tool that analyzes distance and direction vectors between dependent statement instances at the *dependence graph IR* level. After identifying implicit data dependence, proper loop transformations can be selected at the following layers to exploit potential parallelism without violating loop dependencies. Details will be introduced in Section V.

B. *MLIR Infrastructure*

MLIR [23] is a compilation stack built on LLVM [22]. Unlike Clang [5] and other mature compilers with fixed abstraction levels, MLIR provides multi-level IRs and enables flexible optimization and transformation methods at different IR levels. MLIR offers a well-defined infrastructure for users to organize values, operations, types, and attributes in *dialects*. There are dozens of *dialects* in MLIR's ecological system, such as the tensor dialect [8] for tensor creation and manipulation and the CIRCT dialect [4] for efficient hardware designs. POM uses a mixture of dialects, including the affine dialect [2], the arith dialect [3], and the memref dialect [6], to describe the lower-level IR that is converted from our polyhedral IR. The loop body and operations like load and store are represented with the affine dialect which provides an abstraction for affine operations and is a suitable IR to which our polyhedral IR is lowered. The arith dialect is intended to perform fundamental arithmetic operations such as binary and ternary arithmetic operations on integer and floating point numbers. The memref dialect provides a memory reference of arrays and tensors. Details will be introduced in Section V.

C. *Issues in prior works*

We compare representative and latest frameworks to illustrate existing issues in Table I. Existing automatic tools like Pluto [11] have shown the power of polyhedral techniques [9–11, 26, 36]. However, none of them target FPGAs and deliver low performance if their schedule strategies are directly reused for FPGA accelerators. FPGA-related studies using polyhedral techniques [15, 33, 40, 41] have limitations in either performance or generality. PolySA [15] and AutoSA [33] generate systolic arrays for dense matrices. However, the performance degrades on workloads of which the dependence distance is larger than one, such as the stencil computation *Seidel* [27]. Zuo et al. use the polyhedral model to handle data-dependent modules with stencils in a small design space, limiting its upper bound of optimization [41]. POLSCA [40] introduces Pluto into MLIR and directs Pluto to generate codes that can be consumed by HLS tools. However, it demonstrates limited performance by adopting the Pluto schedule which is better suited for multi-core CPUs.

Recent advances in scheduling languages for FPGA frameworks also demonstrate the potential to improve productivity. HeteroCL [21] and HeteroFlow [34] extend TVM DSL and optimize code using Halide IR. Halide-HLS [28] and Hetero-Halide [24] work as FPGA back-ends of Halide for efficient image processing. We compare HeteroCL and HeteroHalide in Table I. They perform several hardware optimizations on a loop-level IR extended from Halide IR. However, they have

TABLE I: Comparison between representative frameworks.

Feature	**Pluto**	**POLSCA**	**HeteroCL**	**Hetero-Halide**	**Scale-HLS**	**POM**
Productivity						
Scheduling language	✗	✗	✓	✓	✗	✓
Multi-level IR	✗	✓	✗	✗	✓	✓
User-specified scheduling	✗	✗	✓	✓	✗	✓
Automated DSE/scheduling	✓	✓	✗	✓	✓	✓
Capability and Efficiency						
Polyhedral model	✓	✓	✗	✗	✗	✓
FPGA-oriented loop transformation	✗	✗	Limited	Limited	Limited	✓
HLS hardware optimization	✗	✓	✓	✓	✓	✓
Ability of data type customization	✗	✗	✓	✓	✗	✓
Generality						
Apply to multiple domains	✓	✓	✓	✗	✓	✓

limited capabilities of dependence analysis and loop transformations based on a single IR. As discussed before, different schedule methods correspond to different optimization granularity and should be applied at appropriate abstraction levels. Adopting multi-level IRs provides a flexible and systemic way to simplify this implementation process and makes it easier to achieve better performance.

ScaleHLS [35] proposes the first HLS framework for FPGAs on MLIR to optimize the input HLS C code. However, all the loop-level transformations are performed at MLIR dialects, and it is not convenient to add a new schedule method since the implementation requires modifying IR operations and passes. Therefore, some critical FPGA-oriented schedule strategies are not considered, leading to non-optimal performance. Also, designers cannot freely determine user-specified schedules, limiting the flexibility. In contrast, POM introduces a new polyhedral IR into MLIR and hence enables advanced dependence analysis and a wide range of FPGA-oriented loop transformations. By representing nested loops as integer sets and maps at our polyhedral IR, loop transformations can be conducted conveniently through a series of manipulations on polyhedral semantics. Moreover, POM lowers IRs progressively by performing proper operations at suitable IR levels, provides a DSL for users to specify a customized schedule, and is applicable to multiple domains, such as image processing, linear algebra, stencils, and deep learning. A DSE engine is also provided to search for a proper schedule automatically.

D. *The motivating example*

We take *BICG* [27] in Fig. 2(a) as a motivating example to illustrate the effects of POM. We evaluate the performance of the original code without optimization (baseline) and the optimized code generated by four frameworks, namely Pluto [11], POLSCA [40], ScaleHLS [35], and POM. The target device is Xilinx XC7Z020 FPGA. The performance varies in *latency* and *speedup*, as shown in Fig. 2(b). We also present corresponding generated schedules in Fig. 2(c)(d)(e). The horizontal axis denotes clock cycles and the vertical axis denotes the execution order of loop iterations. For each iteration, there are two rows to represent the two statements in the loop. Unrolled iterations are omitted for simplification.

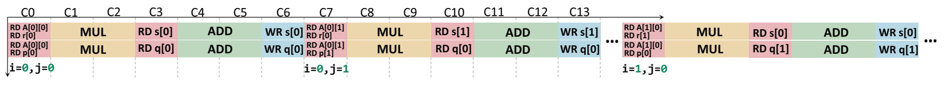
By default, loop iterations are executed sequentially as shown in Fig. 2(c). Pluto enables automatic loop transformation based on the polyhedral model and target CPU/GPU

```
void bicg(
float A[4096][4096],
float s[4096],
float r[4096],
float q[4096],
float p[4096]
){
  for (int i = 0; i < 4096; i += 1){
    for (int j = 0; j < 4096; j += 1){
      s[j] += A[i][j] * r[i];
      q[i] += A[i][j] * p[j];
    }
  }
}
```

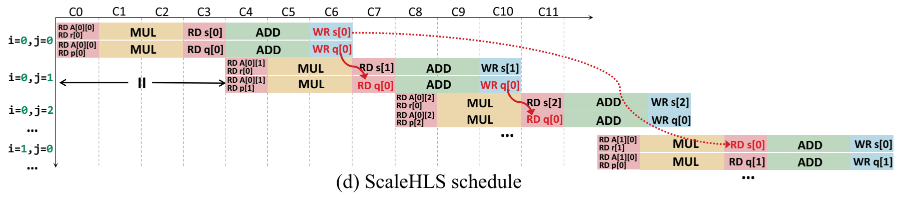
(a) Code snippet of BICG

Framework	Latency/cycles	Speedup
Baseline	234889217	1x
Pluto	236405889	0.99x
POLSCA	109634566	2.1x
ScaleHLS	5648898	41.7x
POM(ours)	**1048588**	**224.0x**

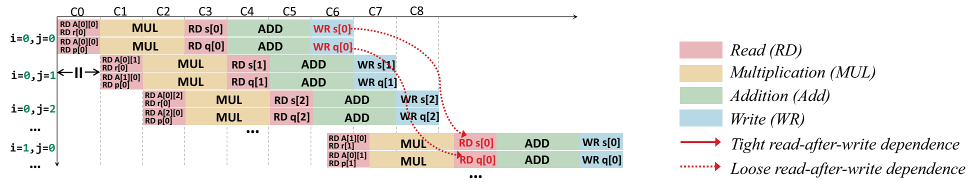
(b) Performance comparison



(c) Baseline schedule



(d) ScaleHLS schedule



(e) POM schedule

Fig. 2: Motivating example: (a) presents the code snippet of BICG; (b) compares *latency* and *speedup* achieved by different frameworks; (c)(d)(e) illustrate schedules for BICG generated by the baseline, ScaleHLS, and POM, correspondingly.

acceleration. Therefore, it focuses on dividing loops into tiles and improving data locality, while parallelizing iterations at outermost loops. This strategy is not suitable for FPGA accelerators where high performance is usually achieved by a deeply pipelined datapath and innermost loops are unrolled for greater parallelism. The generated schedule of Pluto is similar to Fig. 2(c) with slight differences in the execution order of iterations. POLSCA utilizes Pluto to perform automatic loop transformations and then conduct several HLS hardware optimizations. However, there still exists loop-carried dependence in the generated code, restricting the parallelism degree. Additionally, it fails to perform proper array partitioning for large sizes like 4096. As a consequence, it results in a schedule with an unsatisfying initiation interval, i.e., $II = 167$.

ScaleHLS attempts to relieve tight loop-carried dependence by performing *loop interchange*. For example, q[i] is written in each iteration and read by subsequent iterations along the j-dimension, incurring a tight loop-carried dependence (highlighted with red solid lines). To solve this issue, *loop interchange* is applied to move the j-level loop to the outermost, enlarging the distance between dependent read and write operations. However, the II between loop iterations cannot be reduced, because s[j] would be written and read by subsequent iterations along the current inner dimension, i.e., i-dimension, after interchanging loop levels. Therefore, ScaleHLS cannot relieve the tight dependence for all statements simultaneously and achieves non-optimal performance. The actual II is 43 considering unrolled iterations. POM captures this sophisticated dependence and performs *loop split-interchange-merge* to relieve tight dependence, generating an optimized schedule with $II = 2$. As shown in Fig. 2(e), the distances between dependent reads and writes are enlarged (highlighted with red dash lines) and the generated accelerator executes as a highly efficient pipeline. Transformation details are shown in Fig. 10.

III. Framework Overview

Figure 3 depicts a high-level overview of POM. The POM DSL describes the algorithm specification and schedule as input. Then based on our multi-level IR infrastructure, POM first captures data dependencies and generates an optimized data-dependence graph, which is represented as *dependence graph IR*. Next, it extracts polyhedral semantics, perform transformations on polyhedral representations, and yields the *polyhedral IR*. The polyhedral IR is then lowered to MLIR affine dialect with HLS attributes, where HLS hardware optimizations are performed. Finally, the optimized and annotated affine dialect is translated into synthesizable HLS code. Various operations of analysis, transformation, and optimization are performed at different stages. To help users find a high-performance design choice in an extremely large design space, an automatic design space exploration (DSE) engine is integrated into POM. Details will be introduced in corresponding sections.

IV. The Programming Model

POM is equipped with a declarative DSL embedded in C++ to describe loop nests, functions, and arrays. Our DSL inherits the idea of Halide [29] and decouples the algorithm specification from the schedule. This decoupled programming model allows users to write an architecture-independent algorithm and specify a set of scheduling primitives that determines the execution order of operations. By setting user-specified scheduling primitives, programmers can explore different transformation and optimization strategies freely without restructuring the code heavily. *Different from Halide DSL, our DSL is well-designed to extract polyhedral semantics easily, while maintaining simplicity and efficiency.* With clear abstractions to represent variables, multi-dimensional arrays, loop nests, and functions, POM DSL is capable of describing a wide variety

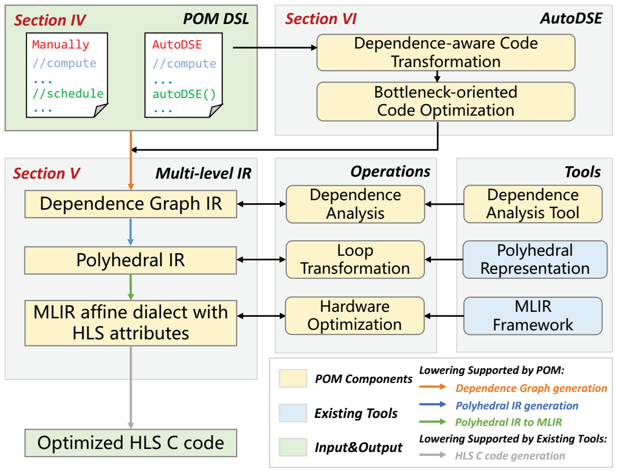


Fig. 3: Framework overview

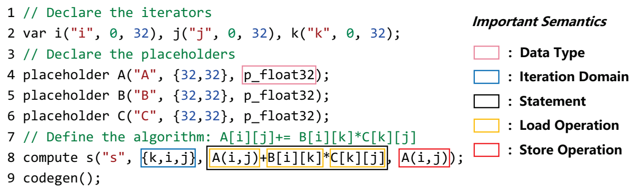

```
1 // Declare the iterators
2 var i("i", 0, 32), j("j", 0, 32), k("k", 0, 32);
3 // Declare the placeholders
4 placeholder A("A", {32,32}, p_float32);
5 placeholder B("B", {32,32}, p_float32);
6 placeholder C("C", {32,32}, p_float32);
7 // Define the algorithm: A[i][j]+= B[i][k]*C[k][j]
8 compute s("s", {k,i,j}, A(i,j)+B[i][k]*C[k][j], A(i,j));
9 codegen();
```
Important Semantics
: Data Type
: Iteration Domain
: Statement
: Load Operation
: Store Operation

Fig. 4: Matrix multiplication with POM DSL.

```
1 // A loop transformation example that tile
2 // dimensions i and j with factors 4 and 4.
3 var i0("i0"), j0("j0"),i1("i1"),j1("j1");
4 s.tile(i, j, 4, 4, i0, j0, i1, j1);
```

Fig. 5: Loop tiling on the algorithm in Fig. 4.

of computation-intensive algorithms, such as linear algebra, stencils, image processing, and deep learning.

A. Algorithm specification

POM DSL describes an algorithm specification using the `compute` operation. Figure 4 presents a matrix multiplication kernel described with POM DSL. We first declare names and ranges of loop iterators (L2) and declare three placeholders that represent arrays A, B, and C (L4-L6). Initialization steps are omitted for simplicity. Then we instantiate a `compute` operation to describe the matrix multiplication algorithm (L8). Instead of explicitly writing a loop, programmers can define the iteration domain, the statement, and the destination placeholder for results in a single line. This simplifies the transformation from `compute` to its polyhedral representation. On the one hand, polyhedral semantics such as iteration domain can be directly obtained from `compute`; On the other hand, the data dependence is explicitly shown by load and store operations of each `compute`. Finally, `codegen` can be added (L9) to generate the corresponding HLS code.

Additionally, algorithms implemented with different data types vary in performance on FPGAs. To enable flexible customization, POM supports multiple data types to specify variables and arrays, including signed and unsigned integers with 8, 16, 32, 64 bits, 32-bit single-precision floating-point, and 64-bit double-precision floating-point. Note that our DSL can be easily extended to support more customized data types.

B. Scheduling primitives

POM automates and simplifies performance optimization with a few lines of code specifying scheduling primitives. A rich set of scheduling primitives is provided, as shown in Table II. Programmers can explore different schedule strategies by instantiating desired scheduling primitives without modifying the algorithm specification. We also provide a primitive `f.auto_DSE()` for automatic design space exploration.

Primitives for loop transformations: Effective loop transformations are necessary to restructure the code and make it better fit with following hardware optimizations. Table II lists transformation primitives we have supported and presents how they work on the iteration domain. For example, *loop skewing* changes the dependence direction by *skewing* the iteration domain. We continue to use the example in Fig. 4 for demonstration. As shown in Fig. 5, *loop tiling* is applied using the `tile` primitive (L4). The dimension `i` and `j` are divided into four new dimensions `i0`, `j0`, `i1`, `j1` with given factors `(4, 4)`. Note that the rationale of loop tiling for FPGA accelerators is not the same as that for CPUs and GPUs which mainly exploit data localities. Loop tiling here sets proper factors to partition loop levels without loop-carried dependence and move them into inner loop levels, which will be unrolled at the lower IR level. Moreover, determining the execution order of computations is crucial when there exist multiple computations. This requires discerning the dimension of the first computation after which the second one is executed. POM addresses it by organizing the sequence of loops using the `after` method, providing a fine-grained control.

Primitives for HLS hardware optimizations: HLS hardware optimizations, represented as HLS pragmas, are supported by POM with a set of primitives. as shown in Table II. These primitives will be translated into corresponding HLS pragmas during code generation. We continue to perform hardware optimizations following Fig. 5, as shown in Fig. 6. To exploit parallelism, we apply *loop pipelining* at loop level $j0$ (L2) and *unroll* its inner loops $i1$ and $j1$ (L3-L4) completely (setting unroll factor to 4). To guarantee parallel memory accesses, *array partitioning* with suitable partition options and factors is applied to improve the memory performance (L5). The equivalent HLS C code is shown in L7-L18.

Primitive for automatic design space exploration: Despite a series of primitives provided, it requires the programmer's expertise to select a proper combination. Moreover, exploring the huge design space of combinations manually is time-consuming and can easily fall into sub-optimal designs. Considering these issues, POM provides a `f.auto_DSE()` primitive, with which programmers can rely on POM to generate high-quality accelerators automatically. Details of DSE will be introduced in Section VI.

```
// Hardware scheduling primitives
s.pipeline(j0,1);
s.unroll(i1, 4);
s.unroll(j1, 4);
A.partition({4,4},"cyclic");

// The equivalent HLS C code after hardware optimizations
#pragma HLS array_partition variable=A cyclic factor=4 dim=1
#pragma HLS array_partition variable=A cyclic factor=4 dim=2
for(int k = 0; k < 32; k++)
  for(int i0 = 0; i0 < 8; i0++)
    for(int j0 = 0; j0 < 8; j0++)
      #pragma HLS pipeline II=1
      for(int i1 = 0; i1 < 4; i1++)
        #pragma HLS unroll factor=4
        for(int j1 = 0; j1 < 4; j1++)
          #pragma HLS unroll factor=4
          A[i0*4+i1][j0*4+j1] += ...
```

Fig. 6: Hardware optimizations and the equivalent HLS code.

TABLE II: Scheduling primitives provided by POM.

Primitive	**Description**
	Loop Transformation
s.interchange(i, j)	Interchange loop level i and j of compute s.
s.split(i, t, i0, i1)	Split loop level i of compute s with factor t. The generated loop levels are (i0, i1).
s.tile(i, j, t1, t2, i0, j0, i1, j1)	Tile loop levels (i, j) of compute s with factors (t1, t2), generating loop levels (i0, j0, i1, j1).
s.skew(i, j, t1, t2, i’, j’)	Skew loop levels (i, j) of compute s with factors (t1, t2), generating loop levels (i’, j’).
s1.after(s2, j)	Compute s1 is executed after compute s2 at loop level j.
	Hardware Optimization
s.pipeline(i, t)	Pipeline the loop at level i with II = t.
A.partition({t1, t2}, “cyclic”)	Partition the array A with factor t1 at the first dimension and t2 at the second dimension.
s.unroll(i, t)	Unroll the loop level i with factor t.
	Design Space Exploration
f.auto_DSE(“PATH”)	Perform design space exploration for function f automatically.

V. Multi-level IR in POM

Figure 7 shows the complete compilation flow. POM explicitly divides the compilation process into three layers with hybrid IRs, lowering the DSL progressively and transforming the code at different abstraction levels with a given schedule. Dependence analysis, code transformation, and hardware optimization are performed at appropriate IR levels, namely dependence graph IR, polyhedral IR, and MLIR affine dialect with HLS pragma attributes.

A. Dependence Graph IR

The initial level of IR in POM is referred to as the dependence graph IR, which facilitates both coarse-grained and fine-grained dependence analyses.

Coarse-grained dependence analysis: Figure 8② illustrates the data dependence graph construction process and the dependence analysis conducted on it. The POM DSL represents explicit producer-consumer relation in the definition of `compute`, reflecting coarse-grained data dependence between different loops. POM captures these coarse-grained data dependencies between `computes` by extracting and analyzing the `load` and `store` operations and preserves them using a dependence map. With the information from the dependence map, a dependence graph is constructed, where each `node` represents a nested loop, and each `edge` signifies the dependence between two loops. POM employs a Depth-First Search (DFS)-based approach to traverse the graph and collect all the `data paths`, which is critical information for subsequent design space exploration.

Fine-grained dependence analysis: Once the dependence graph is constructed, fine-grained data dependence analysis is carried out to analyze dependencies between consecutive iterations of loops. These dependencies, known as loop-carried dependencies, can pose challenges to achieving maximum parallelism, particularly in the context of FPGA-based accelerators. Therefore, it is essential to identify loop-carried dependencies in the dependence graph IR and guide lower-level transformations and optimizations. Specifically, POM traverses each node in the graph, analyzes the dependence by calculating the *distance and direction vectors* of loops, and stores related information as *node attributes*. Figure 8③ illustrates an example of fine-grained data dependence analysis for node S4. POM first captures the data access pattern of the placeholder (D) that stores the updated value, which is (i, j). Considering both the data access pattern and iteration dimensions (i, j, k), POM determines the reduction dimension, which in this case is *k*. The distance vector for node S4, with iteration dimensions (i, j, k), is then computed as (0, 0, 1), indicating the presence of loop-carried dependence in the *k* dimension. This information is valuable for guiding loop transformations, specifically loop interchange in this example, which involves swapping the inner loop *k* with tight dependencies with the outer loop. Moreover, the identification of loop-carried dependence can serve as a hint to users, directing them to set the HLS DEPENDENCE pragma.

B. Polyhedral IR

Once the dependence analysis is complete, the dependence graph IR is lowered to the polyhedral IR, where various loop transformations are implemented at this level. In this section, we discuss the rationale for the polyhedral IR, its construction process, and the loop transformations implemented on it.

Rationale for the polyhedral IR: Although the affine dialect does support partial polyhedral loop transformations, we have chosen to introduce an additional polyhedral IR to implement FPGA-friendly loop transformations with the integer sets and maps from Integer Set Library (isl) [31]. We have two considerations. 1) *Efficiency*: as is discussed in the MLIR official document [7], performing specific transformations on integer sets and maps, rather than the entire nested loop structure, offers more simplicity, particularly when dealing with complex cases such as skewing. 2) *Scope*: isl library is able to perform various set operations to handle constraints between iteration domains, which makes it capable of generating code for any arbitrary affine schedule, whereas MLIR affine dialect lacks this functionality [1]. This means that the transformations available in the affine dialect have certain restrictions com-

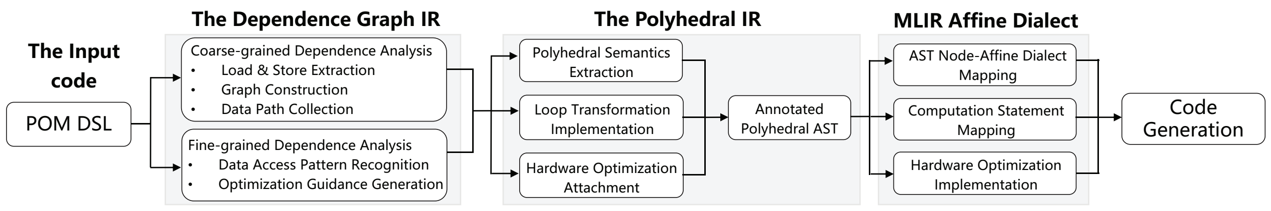
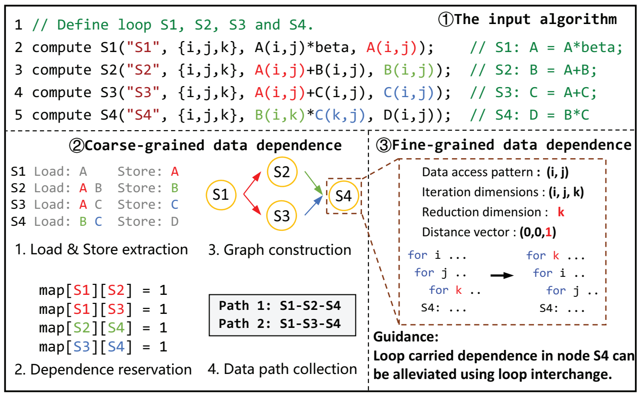

The Input code
POM DSL
The Dependence Graph IR
Coarse-grained Dependence Analysis
Load & Store Extraction
Graph Construction
Data Path Collection
Fine-grained Dependence Analysis
Data Access Pattern Recognition
Optimization Guidance Generation
The Polyhedral IR
Polyhedral Semantics Extraction
Loop Transformation Implementation
Hardware Optimization Attachment
Annotated Polyhedral AST
MLIR Affine Dialect
AST Node-Affine Dialect Mapping
Computation Statement Mapping
Hardware Optimization Implementation
Code Generation

Fig. 7: The compilation flow in POM.



①The input algorithm

```
// Define loop S1, S2, S3 and S4.
compute S1("S1", {i,j,k}, A(i,j)*beta, A(i,j));     // S1: A = A*beta;
compute S2("S2", {i,j,k}, A(i,j)+B(i,j), B(i,j));  // S2: B = A+B;
compute S3("S3", {i,j,k}, A(i,j)+C(i,j), C(i,j));  // S3: C = A+C;
compute S4("S4", {i,j,k}, B(i,k)*C(k,j), D(i,j));  // S4: D = B*C
```

②Coarse-grained data dependence

```
S1 Load: A     Store: A
S2 Load: A B   Store: B
S3 Load: A C   Store: C
S4 Load: B C   Store: D
```

1. Load & Store extraction

```
map[S1][S2] = 1
map[S1][S3] = 1
map[S2][S4] = 1
map[S3][S4] = 1
```

2. Dependence reservation

3. Graph construction

```
Path 1: S1-S2-S4
Path 2: S1-S3-S4
```

4. Data path collection

③Fine-grained data dependence

Data access pattern : (i, j)
Iteration dimensions : (i, j, k)
Reduction dimension : k
Distance vector : (0,0,1)

```
for i ...          for k ...
 for j ..    →      for i ..
  for k ..           for j ..
 S4: ...            S4: ...
```

Guidance:
Loop carried dependence in node S4 can be alleviated using loop interchange.

Fig. 8: Illustration of dependence graph IR.

pared to more generic polyhedral models like isl, which can handle any affine schedule. For instance, loop fusion in affine dialect currently only fuse loop nests with single-writer/single-reader dependence with the same constant loop bounds.

Construction of the polyhedral IR: Figure 9(c) illustrates the construction process of our polyhedral IR. In the first step, the polyhedral semantics of each node (nested loop) in the dependence graph IR, such as iteration domain and schedules, are extracted and represented by integer sets and maps efficiently. In the second step, given user-specified primitives (①) in DSL, iteration domains are transformed by our pre-implemented loop transformation methods correspondingly. Similarly, schedules are modified to guarantee the execution order of `computes`, as specified in DSL (③). This is based on the lexicographic order theory [32]. In the third step, a union map is created by collecting all the domains and schedules of different loops in one integer map. Then an *ast_build* method from isl is invoked to build the *polyhedral AST* from the union map. The generated AST contains four types of nodes: *if-node*, *for-node*, *block-node*, and *user-node*, which can be seamlessly translated to MLIR affine dialect. Additionally, since the *polyhedral AST* lacks representation for computation, we attach critical information such as computation statements to *user-nodes* (⑦). During IR lowering, this information is retrieved to generate computation statements in affine dialect, as shown in Fig. 9(d) (⑦). Furthermore, the hardware optimization information is attached to the corresponding node within the AST. This enables the optimization information to be preserved and utilized in the next layer of IR.

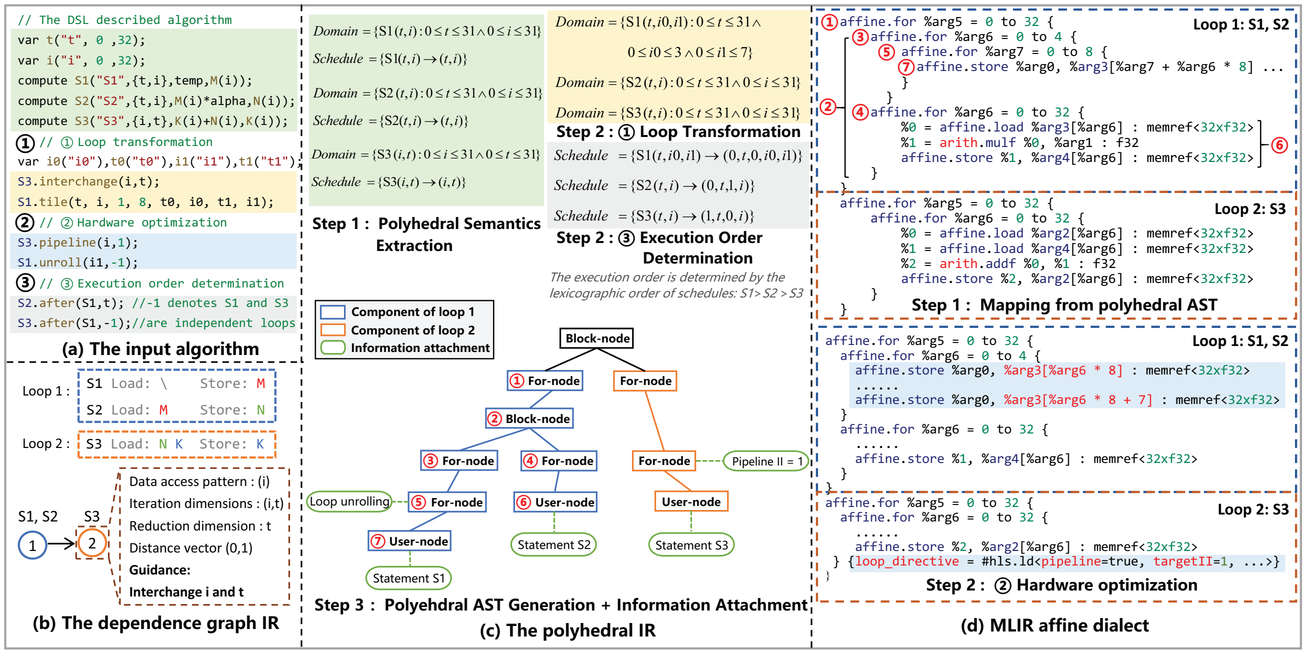
Implementation of loop transformations: By representing nested loops with integer sets and maps at the polyhedral IR level, loop transformations can be formulated as a series of manipulations on polyhedral semantics. These manipulations include the interchange between loop dimensions, the calculation of new iteration domains through mathematical methods, the modification of array indexes, etc. We take the loop tiling process in Fig. 9 as an example. Tiling the loop level i with factor 8 will change the iteration domain $\{S(t,i)\}$. The computation process of the new domain is $\{S(t,i0,i1) : 0 \leq t \leq 31 \wedge i0 = floor(i/8) \wedge i1 = i\%8 \wedge 0 \leq i \leq 31\}$, namely $\{S(t,i0,i1) : 0 \leq t \leq 31 \wedge 0 \leq i0 \leq 3 \wedge 0 \leq i1 \leq 7\}$. Note that besides operations on rectangular iteration domains, POM is capable of handling non-rectangular iteration domains through more complex loop transformations such as *loop skewing* with the guidance of dependence analysis.

We implement the most commonly used loop transformations as a library in Table II. Users only need to specify APIs provided in our DSL and invoke POM to perform automatic transformations. Thanks to the efficient representation with integer sets and maps, POM can be easily extended to support more customized transformations.

C. *MLIR affine dialect with HLS attributes*

The polyhedral IR is lowered to the MLIR affine dialect with HLS attributes, where hardware optimizations are performed. The affine dialect provides abstractions for affine operations and can be naturally mapped from the polyhedral AST. Besides, the affine dialect provides explicit loop structures required for hardware optimizations, which the abstract polyhedral IR lacks. This makes it a suitable IR to insert HLS pragma-related information as attributes for code generation.

Mapping from polyhedral AST to affine dialect: Given the polyhedral IR, which consists of an annotated AST with hardware optimization information, POM maps different types of nodes in the AST to corresponding operations described in affine dialect, as shown in Fig. 9(d). For example, a `for-node` within the AST signifies a *for* loop in the affine dialect (①③④⑤). It captures essential loop attributes such as the lower bound, upper bound, and iteration step size. Similarly, a `user-node` within the AST represents user-defined statements in the `compute` (⑥⑦), which is reserved in POM DSL. POM is equipped with a recursive method to parse the statements and data accesses described in POM DSL and transform them into correct affine dialect representations. By implementing automated translation from polyhedral IR to MLIR affine dialect, POM bridges the gap between the powerful polyhedral model and MLIR. Therefore, it can freely interact with other excellent works in MLIR's ecosystem while fully exploiting the advantages of the polyhedral model.



```
// The DSL described algorithm
var t("t", 0 ,32);
var i("i", 0 ,32);
compute S1("S1",{t,i},temp,M(i));
compute S2("S2",{t,i},M(i)*alpha,N(i));
compute S3("S3",{i,t},K(i)+N(i),K(i));
① // ① Loop transformation
var i0("i0"),t0("t0"),i1("i1"),t1("t1");
S3.interchange(i,t);
S1.tile(t, i, 1, 8, t0, i0, t1, i1);
② // ② Hardware optimization
S3.pipeline(i,1);
S1.unroll(i1,-1);
③ // ③ Execution order determination
S2.after(S1,t); //-1 denotes S1 and S3
S3.after(S1,-1);//are independent loops
```

$$Domain = \{S1(t,i): 0 \le t \le 31 \wedge 0 \le i \le 31\}$$
$$Schedule = \{S1(t,i) \rightarrow (t,i)\}$$
$$Domain = \{S2(t,i): 0 \le t \le 31 \wedge 0 \le i \le 31\}$$
$$Schedule = \{S2(t,i) \rightarrow (t,i)\}$$
$$Domain = \{S3(i,t): 0 \le i \le 31 \wedge 0 \le t \le 31\}$$
$$Schedule = \{S3(i,t) \rightarrow (i,t)\}$$

$$Domain = \{S1(t,i0,i1): 0 \le t \le 31 \wedge 0 \le i0 \le 3 \wedge 0 \le i1 \le 7\}$$
$$Domain = \{S2(t,i): 0 \le t \le 31 \wedge 0 \le i \le 31\}$$
$$Domain = \{S3(t,i): 0 \le t \le 31 \wedge 0 \le i \le 31\}$$

$$Schedule = \{S1(t,i0,i1) \rightarrow (0,t,0,i0,i1)\}$$
$$Schedule = \{S2(t,i) \rightarrow (0,t,1,i)\}$$
$$Schedule = \{S3(t,i) \rightarrow (1,t,0,i)\}$$

```
①affine.for %arg5 = 0 to 32 {
  ③affine.for %arg6 = 0 to 4 {
    ⑤ affine.for %arg7 = 0 to 8 {
      ⑦affine.store %arg0, %arg3[%arg7 + %arg6 * 8] ...
      }
    }
②  ④affine.for %arg6 = 0 to 32 {
        %0 = affine.load %arg3[%arg6] : memref<32xf32>
        %1 = arith.mulf %0, %arg1 : f32                     ⑥
        affine.store %1, %arg4[%arg6] : memref<32xf32>
    }
}
affine.for %arg5 = 0 to 32 {
    affine.for %arg6 = 0 to 32 {
        %0 = affine.load %arg2[%arg6] : memref<32xf32>
        %1 = affine.load %arg4[%arg6] : memref<32xf32>
        %2 = arith.addf %0, %1 : f32
        affine.store %2, %arg2[%arg6] : memref<32xf32>
    }
}
```

```
affine.for %arg5 = 0 to 32 {
  affine.for %arg6 = 0 to 4 {
    affine.store %arg0, %arg3[%arg6 * 8] : memref<32xf32>
    ......
    affine.store %arg0, %arg3[%arg6 * 8 + 7] : memref<32xf32>
  }
  affine.for %arg6 = 0 to 32 {
    ......
    affine.store %1, %arg4[%arg6] : memref<32xf32>
  }
}
affine.for %arg5 = 0 to 32 {
  affine.for %arg6 = 0 to 32 {
    ......
    affine.store %2, %arg2[%arg6] : memref<32xf32>
 } {loop_directive = #hls.ld<pipeline=true, targetII=1, ...>}
}
```

Fig. 9: The lowering process of multi-level IR in POM

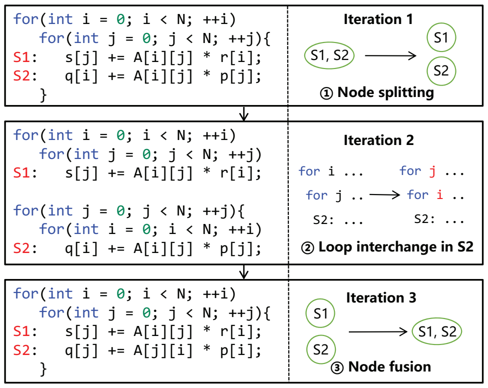
Implementation of hardware optimization: Upon completing the mapping, hardware optimizations attached to the nodes are eventually performed. POM provides a set of FPGA-specific hardware optimizations that are conducted at the affine dialect by inserting HLS pragma-related attributes into the corresponding code hierarchy. For example, by specifying a scheduling primitive `S3.pipeline(i, 1)` in DSL, a related pipeline attribute is attached to the loop level i with an initiation interval of one in the affine dialect in Fig. 9(d), indicating the specific *loop pipelining* pragma-type operation. After manipulations on the affine dialect, the fully optimized IR is sent to the back-end to generate synthesizable HLS C code, where all of the attributes are translated to HLS pragmas.

VI. Design Space Exploration

POM enables a wide range of transformation and optimization methods, all of which form an exponentially increasing design space. To reduce effort, POM provides a two-stage DSE engine to automatically search for design choices that produce high-quality FPGA accelerators.

A. Dependence-aware code transformation

Dependence-aware code transformation is performed at first to alleviate tight loop-carried dependencies and facilitate parallelism as much as possible. As is introduced, the input function is represented as a data dependence graph where each node denotes a loop. The DSE engine traverses the graph and our dependence analysis tool checks the loop-carried dependence to give hints for loop transformations. More specifically, if any loop-carried dependence is captured inside the loop, loop interchange will be considered since it effectively changes the



```
for(int i = 0; i < N; ++i)
   for(int j = 0; j < N; ++j){
S1:    s[j] += A[i][j] * r[i];
S2:    q[i] += A[i][j] * p[j];
   }
```

```
for(int i = 0; i < N; ++i)
   for(int j = 0; j < N; ++j)
S1:    s[j] += A[i][j] * r[i];

for(int j = 0; j < N; ++j){
   for(int i = 0; i < N; ++i)
S2:    q[i] += A[i][j] * p[j];
```

```
for(int i = 0; i < N; ++i)
   for(int j = 0; j < N; ++j){
S1:    s[j] += A[i][j] * r[i];
S2:    q[j] += A[j][i] * p[i];
   }
```

Fig. 10: Illustration of dependence-aware code transformation. We use C code and graph nodes for illustration.

dependence distance. We further consider the situation where two conflicting loop interchange strategies are proposed by the tool. In these cases, we need to leverage other transformations such as loop splitting and loop skewing to restructure the code. To make sure that data dependencies after transformation have been alleviated as much as possible, the dependence analysis tool will iteratively recheck loop-carried dependencies after each transformation and additional transformations are applied if needed. This iterative process will be terminated if every node no longer has the tight dependence issue or the number of iterations has reached its pre-defined bounds. Then the transformed code is well-prepared for the second stage.

We take Fig. 10 as an example. During the first iteration, the dependence analysis tool checks loop-carried dependence in both S1 and S2 and detects contradictory transformation strategies: S1 can execute efficiently without loop-carried dependence at the inner loop and hence tends to retain the current loop orders, while S2 has tight loop-carried dependence at the loop level j and tends to interchange i- and j- loop levels. To solve this issue, POM splits S1 and S2 into two independent loops, as shown in Fig. 10①. Then the dependence analysis tool continues iteratively rechecking dependencies inside each node, performing loop interchange to S2 in the second iteration (②) and conservatively fuse S1 and S2 together in the third iteration (③). After a series of transformations based on iterative dependence analysis, the potential parallelism of functions can be fully exploited.

B. Bottleneck-oriented code optimization

Since loop-carried dependencies are alleviated as much as possible during the first stage, we can focus on exploring parallelism by evaluating the combination of different loop-tiling strategies and HLS hardware optimizations. The core idea is to prioritize performance optimization (i.e., latency reduction) of the bottleneck node in the critical path.

At first, POM estimates the latency of each node in the dependence graph and gets the latency of each path, using the in-house model from [35][38], which has been integrated into MLIR. After estimation, paths are ordered by their latency, and the critical path with the longest latency is selected to be optimized first. POM DSE further chooses the node with the longest latency in the critical path and performs a series of optimization strategies on it, varying types and factors of loop tiling and HLS optimizations. The set of types and factors are determined before the search and users can specify suitable groups of strategies and parameters. In this paper, we specify a range of strategies for loop tiling and HLS optimizations to cover different parallelism degrees. When optimizing the bottleneck node in the critical path, the DSE engine increases the parallelism degree gradually and sets the strategies correspondingly. Once the current node or path is not the bottleneck, the algorithm will switch to the new bottleneck for optimization and repeat the process. An exit mechanism is considered to avoid continuously optimizing the same node if it is always the longest one: the optimization will stop if the current node achieves its maximum parallelism degree or consumes resources that exceed resource constraints. We use a list to store all the nodes to be optimized. If the exit mechanism is triggered for one node, this node will be removed from the list. The DSE terminates when the optimization list is empty.

VII. Evaluation

A. Experimental setup

Xilinx Vitis HLS and Vivado 2022.1 are utilized for HLS synthesis and hardware implementation. The reported performance and resource statistics are collected from HLS synthesis reports and power statistics are obtained from implementation reports. The target device is Xilinx XC7Z020 FPGA, containing 220 DSPs, 53,200 LUTs, 106400 FFs, and 4.9 Mb memories. All the benchmarks are tested at the 100MHz target frequency with data types of 32-bit floating-point.

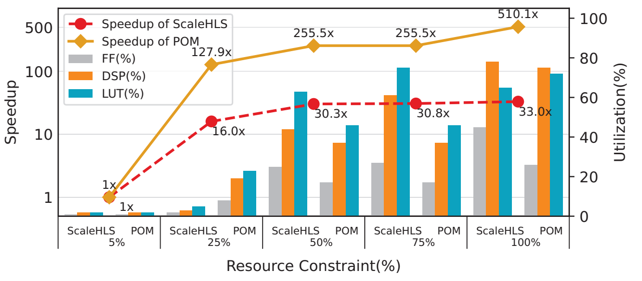


Fig. 11: Speedup and resource utilization of *2MM*.

B. Evaluation on typical HLS benchmarks

We compare POM with state-of-the-art HLS frameworks on MLIR, POLSCA [40] and ScaleHLS [35]. Evaluation is performed on the same typical HLS benchmarks [27] with large problem sizes (4096), namely GEMM, BICG, GESUMMV, 2MM, and 3MM. Table III compares latency speedups, resource usage, power, the achieved II and tile sizes, and the parallelism degree of the accelerators generated by the three frameworks, correspondingly. We adopt the same target clock (10ns) as reported in ScaleHLS [35] for comparison. The DSE time costs are also compared in the last column.

The *latency speedup* is computed by dividing the latency (*#clock cycles*) of the original C code without any optimization by the latency (*#clock cycles*) of the optimized HLS C code. Experimental results show that POM significantly improves the overall performance of the baseline, resulting in speedups ranging from 223.2× to 575.9× across all the benchmarks. The achieved *II* reflects the degree of parallelism between successive iterations in a pipeline loop, with smaller values indicating greater parallelism. The achieved tile sizes and unroll factors denote the number of parallel copies of computation units being executed, with larger values indicating greater parallelism. Therefore, to quantify the attained parallelism, we compute the *parallelism* degree by dividing the product of tile sizes by the achieved *II*, with higher values indicating greater parallelism achieved by the optimized accelerators.

We can see that POLSCA achieves limited speedups. This is primarily due to the presence of loop-carried dependencies in the code generated by Pluto in POLSCA, which negatively impacts the level of *parallelism* achieved. Additionally, when handling large problem sizes, POLSCA does not properly partition arrays, resulting in a further decline in performance. Since Pluto works as a black box, its tile sizes and unroll factors are unknown to users.

Compared to ScaleHLS, POM achieves better speedups on most benchmarks with a 6.46× performance improvement on average. This improvement mainly comes from three aspects. First, with our accurate dependence analysis at the dependence graph IR, tight loop-carried dependencies are detected and alleviated, resulting in better achieved *II* and larger tile sizes/unroll factors. This greatly improves our parallelism degree. For example, POM improves the performance of BICG by 224x speedup with *parallelism=16*, while ScaleHLS achieves 41.7x speedup with *parallelism=3* due to tight dependence inside the

TABLE III: Evaluation and comparison on typical HLS benchmarks. The vector $[m,\ n]$ denotes tiling sizes at different loop levels. 2MM and 3MM contain multiple loops with a sequence of tiling vectors.

Benchmark	Framework	Prob. Size	Speedup	DSP (Util.%)	FF (Util.%)	LUT (Util.%)	Power (W)	Achieved II	Achieved tile sizes and unroll factors	Paral-lelism	DSE Time(s)
GEMM	POLSCA	4096	2.3×	7 (3%)	4980 (4%)	7817 (14%)	-	248	-	-	-
	ScaleHLS	4096	576.1×	214 (97%)	41616 (39%)	42676 (80%)	0.767	4	[2, 4, 16]	32	24.4
	POM	4096	575.9×	166 (75%)	23067 (21%)	30966 (58%)	0.459	1	[1, 2, 16]	32	11.4
BICG	POLSCA	4096	2.1×	5 (2%)	4665 (4%)	8150 (15%)	-	161	-	-	-
	ScaleHLS	4096	41.7×	32 (14%)	17326 (16%)	10386 (19%)	0.176	43	[16, 8]	3.0	6.3
	POM	4096	224.0×	160 (72%)	27189 (25%)	43823 (82%)	0.782	2	[1, 32]	16	5.7
GESUMMV	POLSCA	4096	1.4×	8 (3%)	6112 (5%)	10979 (20%)	-	161	-	-	-
	ScaleHLS	4096	199.1×	158 (72%)	49838 (46%)	35848 (67%)	0.643	9	[8, 16]	14.2	6.4
	POM	4096	223.2×	160 (72%)	19409 (18%)	27595 (51%)	0.490	1	[1, 16]	16	4.7
2MM	POLSCA	4096	2.0×	8 (3%)	7137 (6%)	11355 (21%)	-	248	-	-	-
	ScaleHLS	4096	31.0×	166 (75%)	34912 (32%)	45419 (85%)	0.462	4, 1	[1, 8, 16], [1, 1, 1]	1.9	77.2
	POM	4096	510.1×	166 (75%)	28039 (26%)	38577 (72%)	0.537	1	[1, 2, 16], [1, 2, 16]	32	24.5
3MM	POLSCA	4096	1.8×	10 (4%)	10128 (9%)	16831 (31%)	-	256	-	-	-
	ScaleHLS	4096	40.1×	115 (52%)	34522 (32%)	38007 (71%)	0.599	1, 3, 3	[1, 1, 1], [1, 8, 8], [1, 8, 8]	2.7	56.8
	POM	4096	335.4×	160 (72%)	21928 (20%)	32995 (62%)	0.513	1	[1, 2, 8], [1, 2, 8], [1, 2, 8]	16	31.3

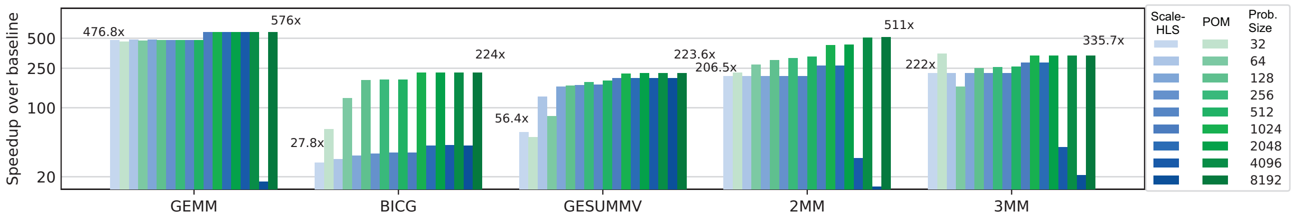


Fig. 12: Comparison and evaluation with different problem sizes on typical HLS benchmarks.

loop. Second, by introducing the powerful polyhedral model into MLIR at the polyhedral IR level, more effective loop transformations can be conducted efficiently while ensuring the correctness of the code. This actually enlarges the design space and increases the possibility of finding a design choice with higher performance. Third, although the design space is enlarged, our two-stage DSE engine still explores the large design space efficiently and finds high-performance design choices successfully. For example, 3MM consists of multiple loops in multiple paths. The DSE engine of POM prioritizes the optimization of the bottleneck loop and switches to other loops once a new bottleneck appears, leading to concurrent optimization for all the loops. In contrast, ScaleHLS optimizes some loops heavily without leaving additional optimization space for other loops. For example, the first loop of 3MM is not tiled and unrolled with factors $[1, 1, 1]$.

For the DSE time cost, it takes a shorter DSE time for POM on all benchmarks. Although POM achieves a smaller but closer speedup compared to ScaleHLS for GEMM (0.99x), it takes 50.1% time cost for POM to find this design. Note that the code generation process from MLIR to HLS C typically completes within 0.1s. Therefore, the DSE time can be considered as the toolchain's runtime. The rest columns in Table III present resource usage with utilization ratios on the same FPGA. POM fully utilizes available resources given the same constraints which are set to the total resources on the board. We vary resource constraints to different percentages of the total resources, run frameworks, and evaluate *speedup* and *resource utilization* of generated accelerators. Figure 11 shows

TABLE IV: Comparison with manual optimization on BICG.

Design	Cycles	Speedup	DSP(Util.%)	FF(Util.%)	LUT(Util.%)
Unoptimized	234889217	1×	10(4%)	1101(1%)	1618(3%)
Manual opt.	1458178	161.1x	208(94%)	23454(22%)	50899(95%)
DSE opt.	1048588	224.0x	160(72%)	27189(25%)	43823(82%)

the results of 2MM and POM achieves higher performance given different resource constraints. Besides resource costs, we also compare power consumption in Table III. POM achieves superior or competitive performance speedup while consuming less power for GEMM, GESUMMV, and 3MM. For BICG and 2MM, POM achieves remarkable speedups of 5.37x and 16.45x with an increase in power consumption by a factor of 4.44x and 1.16x, respectively, demonstrating a better performance-per-watt compared to ScaleHLS.

C. Comparison with manual optimization

To better evaluate the quality of the automatically generated design, we compare it to a design that is manually optimized, leveraging our expertise in FPGA optimization. BICG is used for this case study. A series of HLS pragmas and code rewriting is performed to improve the parallelism during manual optimization. The results are shown in Table IV. We can see that the FPGA design generated by POM achieves a 1.39x speedup compared to the manually optimized HLS design and consumes fewer resources on the same FPGA device.

D. Evaluation on scalability with different problem sizes

To evaluate the scalability of POM, we compare the performance of POM and ScaleHLS across various problem sizes on

TABLE V: Comparison on image processing and DNN applications.

Applications		Prob. Size	Speedup ScaleHLS	Speedup POM	Speedup P/S	DSP (Utilization%) ScaleHLS	DSP POM	DSP P/S	FF (Utilization%) ScaleHLS	FF POM	FF P/S	LUT (Utilization%) ScaleHLS	LUT POM	LUT P/S
Image Processing	EdgeDetect	4096	19.1×	344.0×	18.0	23(10%)	183(83%)	8.0	10130(9%)	30686(28%)	3.0	11438(21%)	47872(89%)	4.2
	Gaussian	4096	111.4×	312.0×	2.8	87(39%)	177(80%)	2.0	33104(31%)	51203(48%)	1.5	28849(54%)	52751(99%)	1.8
	Blur	4096	59.3×	356.0×	6.0	18(8%)	48(21%)	2.7	7300(6%)	11378(10%)	1.6	7006(13%)	14549(27%)	2.1
DNN	VGG-16	512	33.6×	86.8×	2.6	137(62%)	40(18%)	0.3	38498(36%)	40127(37%)	1.0	53819(101%✘)	52837(99%)	1.0
	ResNet-18	512	50.8×	46.4×	0.9	212(96%)	30(13%)	0.1	57882(54%)	38174(35%)	0.6	87662(164%✘)	52484(98%)	0.6

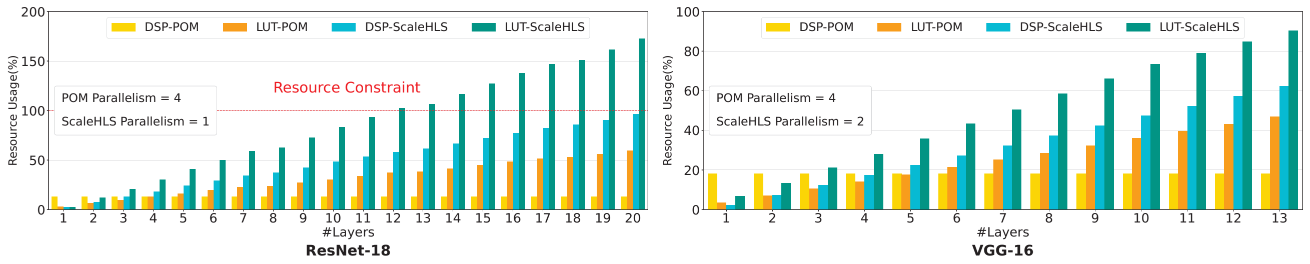


Fig. 13: Accumulated resource usage for DNN workloads. We omit resource consumption of small loops such as initialization.

TABLE VI: Comparison on optimization for critical loops.

Benchmark	Tile Size ScaleHLS	Tile Size POM	Achieved II ScaleHLS	Achieved II POM	Parallelism ScaleHLS	Parallelism POM
EdgeDetection	[1,2,3]	[2,2,3]	12, 6	1	0.67	12
Gaussian	[1,3,3]	[1,3,3]	3	1	3	9
Blur	[2,1,3]	[2,2,3]	3	1	2	12

TABLE VII: Evaluation on complicated code patterns.

Benchmark	Speedup	DSP(Util.%)	FF(Util.%)	LUT(Util.%)
Jacobi-1d	47.6×	14(6%)	2794(2%)	3746%(7%)
Jacobi-2d	136.0×	44(20%)	8542(8%)	13178%(24%)
Heat-1d	22.9×	20(9%)	2466(2%)	3832%(7%)
Seidel	53.8×	73(33%)	25327(23%)	20892%(39%)

typical HLS benchmarks, as shown in Fig. 12. The problem sizes vary from 32 to 8192. We can see that POM achieves superior performance for the majority of problem sizes. For problem sizes ranging from 32 to 2048, both POM and ScaleHLS exhibit stable performance improvement. However, as the problem size scales up to 4096 and 8192, there is a noticeable performance decline for ScaleHLS. For example, at a problem size of 8192, ScaleHLS encounters difficulties in generating efficient designs for benchmarks like GEMM, 2MM, and 3MM, providing only basic hardware optimizations such as loop pipelining. In contrast, POM continues to generate high-quality designs even as problem sizes expand to 8192. For certain benchmarks with very small problem sizes, such as GESUMMV with a problem size of 32, POM exhibits slightly inferior performance compared to ScaleHLS. This occurs because POM prioritizes the optimization of bottleneck loops while placing relatively less emphasis on simpler loops. These simple loops account for a higher proportion of latency when problem sizes (i.e., loop bounds) are small. The slightly lower performance is acceptable since the absolute latency difference is quite minor for small problem sizes.

E. Evaluation on image processing and DNN applications

To demonstrate the ability of POM in accelerating complicated real-world applications, we evaluate the performance of ScaleHLS and POM on several image processing and DNN applications, including EdgeDetection [10], Gaussian [10], Blur [29], VGG-16 [30] and ResNet-18 [18] on the same FPGA device. Table V shows the detailed results, where P/S denotes the relative ratio between POM (P) and ScaleHLS (S). Compared to the baseline without optimization, POM significantly improves the performance with speedups from $46.4\times$ to $356.0\times$ within minutes. Compared to ScaleHLS, POM achieves $6.06\times$ speedup on average at similar time costs.

For image processing, Table VI compares the tile size, achieved *II*, and *parallelism* for critical loops. With accurate dependence analysis and an effective DSE engine, POM is able to achieve a higher parallelism degree compared to ScaleHLS.

For DNN workloads, POM and ScaleHLS use different optimization strategies, as illustrated in Fig. 13. Due to space limit, statistics for critical loops are visualized in the figure. Here critical loops refer to the nested loops with a loop level exceeding four in the neural network. For instance, ResNet-18 has 20 critical loops, including 17 convolution loops and 3 residual loops, and VGG-16 has 13 critical loops, all of which are convolution loops. POM improves the parallelism of each loop and resources are reused between different layers. Given a fixed number of total resources, this resource reuse increases the available resources for each layer in POM, resulting in higher parallelism for each layer. Consequently, DNN layers in POM are executed in sequence but the *parallelism* of each layer (i.e., 4) is maximized due to resource reuse. In contrast, ScaleHLS makes layers executed in a pipelined dataflow, and resources are not shared among different layers. The overall

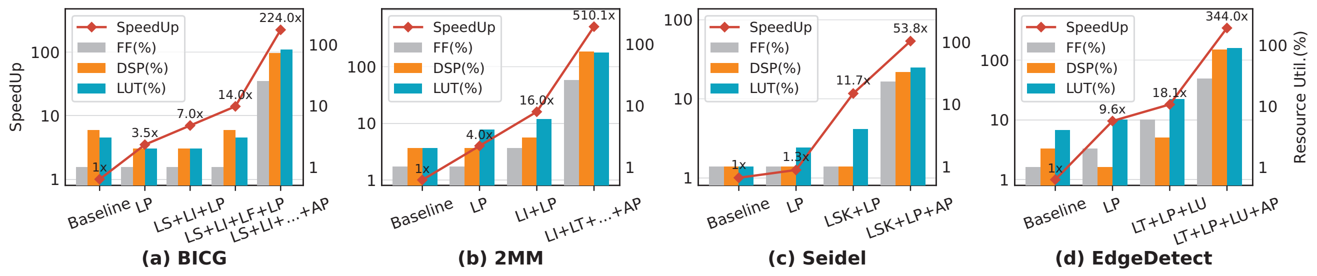


Fig. 14: Impact analysis of scheduling primitives. LI, LT, LS, LF, LSK, LP, LU and AP denote *loop interchange*, *loop tiling*, *loop splitting*, *loop fusion*, *loop skewing*, *loop pipelining*, *loop unrolling*, and *array partitioning*, respectively.

latency is equal to that of the bottleneck layer. Since each layer occupies resources, the parallelism of each loop is degraded (i.e., 1), influencing the overall performance greatly, especially for DNNs with large #layers. The pipelined dataflow will stall due to unmatched computation paces between successive loops. Therefore, POM achieves a 2.6× speedup on VGG-16 compared to ScaleHLS. For ResNet-18, POM achieves a relatively lower speedup (0.9×) while consuming much fewer resources (0.1× DSPs and 0.6× LUTs), satisfying the resource constraints of our FPGA device. The optimized designs from ScaleHLS are not feasible since their resource usage exceeds the total resources on the target FPGA.

F. Evaluation on complicated data access patterns

We extensively test four applications with more complicated code patterns, including Jacobi-1d, Jacobi-2d, Heat-1d, and Seidel. For example, Seidel is a stencil computation with complex data access patterns with tight loop-carried dependence. For these benchmarks, ScaleHLS and POLSCA fail to find an optimization strategy that improves the performance greatly. In contrast, POM generates high-quality designs within seconds. Experimental results show that POM continuously performs well on these benchmarks and improves the overall performance of the baseline by 22.9× to 136.0× (65.08× on average). The baseline is the original implementation without optimization. The reason for the improvement is that POM supports more useful loop transformations, such as *loop skewing*. We also notice that resource utilization ratios are relatively small for these benchmarks. This is because their loop-carried dependence degrades the parallelism of loops, even though we have relieved the dependence to some extent.

G. Impact analysis of scheduling primitives

To understand the impact of different scheduling primitives, we conducted an ablation study on representative benchmarks in Fig. 14. Performance speedup and resource usage are presented. We notice that different benchmarks may benefit from different primitives due to their specific loop structures and data access patterns. For example, EdgeDetect gains 9.6× speedup from *loop pipelining*, while the improvement of Seidel applied with the same optimization is limited. This is due to the loop-carried dependence inside the Seidel loop and the overall performance is improved significantly after loop skewing is applied. 2MM benefits a lot from combinations of

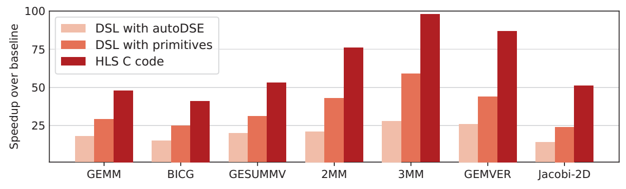
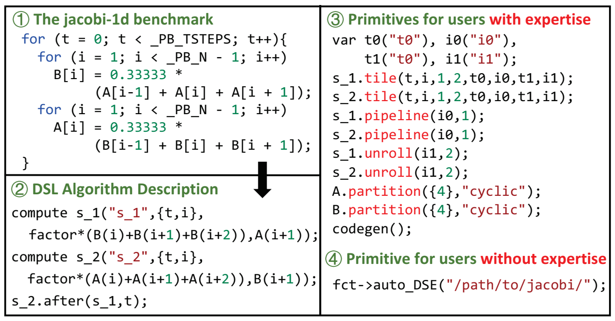


Fig. 15: Comparison of lines of code (LoC).


① The jacobi-1d benchmark

```
for (t = 0; t < _PB_TSTEPS; t++){
  for (i = 1; i < _PB_N - 1; i++)
    B[i] = 0.33333 *
          (A[i-1] + A[i] + A[i + 1]);
  for (i = 1; i < _PB_N - 1; i++)
    A[i] = 0.33333 *
          (B[i-1] + B[i] + B[i + 1]);
}
```

② DSL Algorithm Description

```
compute s_1("s_1",{t,i},
  factor*(B(i)+B(i+1)+B(i+2)),A(i+1));
compute s_2("s_2",{t,i},
  factor*(A(i)+A(i+1)+A(i+2)),B(i+1));
s_2.after(s_1,t);
```

③ Primitives for users with expertise

```
var t0("t0"), i0("i0"),
    t1("t0"), i1("i1");
s_1.tile(t,i,1,2,t0,i0,t1,i1);
s_2.tile(t,i,1,2,t0,i0,t1,i1);
s_1.pipeline(i0,1);
s_2.pipeline(i0,1);
s_1.unroll(i1,2);
s_2.unroll(i1,2);
A.partition({4},"cyclic");
B.partition({4},"cyclic");
codegen();
```

④ Primitive for users without expertise

```
fct->auto_DSE("/path/to/jacobi/");
```

Fig. 16: Jacobi-1d described with POM DSL.

loop transformations and hardware optimizations because the parallelism is fully explored. These results highlight the necessity of integrating both loop transformations and hardware optimizations to fully exploit parallelism.

H. Evaluation of DSL expressiveness

To evaluate the expressiveness of POM DSL, we first compare the number of lines of code (LoC) between the POM DSL and the equivalent HLS C code, as shown in Fig. 15. Since the automatic DSE engine provided in POM frees programmers from explicitly setting scheduling primitives, we further consider two cases, namely DSL with the `autoDSE` primitive and DSL with *manually-specified* primitives. Representative benchmarks with different complexities are chosen for evaluation. We describe these benchmarks with POM DSL, use the `autoDSE` primitive to automatically decide the scheduling, and utilize our tool to generate equivalent HLS C code. We also implement the same optimizations by manually setting the primitives in POM DSL. This ensures that the performance of these three kinds of code remains the same. Compared to HLS C code, POM DSL is able to express and optimize the same algorithm with much fewer lines of code. It takes less than one-third of the code for DSL with `autoDSE` to represent benchmarks with multiple loops such as 3mm.

These results prove that POM DSL saves great engineering efforts with clear definitions of computations and arrays and efficient scheduling primitives.

To demonstrate the DSL usage, we also conduct a case study with the stencil benchmark, Jacobi-1d [27], in Fig. 16①. POM utilizes `compute` and `after` to represent the nested loop (②). For users with expertise, POM provides various scheduling primitives to help users explore different designs quickly (③). For users lacking FPGA expertise, they can utilize the `autoDSE` primitive (④) to automatically generate high-quality designs without explicitly specifying any other primitives. Note that the `autoDSE` primitive in ④ is able to generate the same design as ③.

VIII. Conclusion

This paper proposes POM, an end-to-end optimizing framework on MLIR, to generate high-quality FPGA-based accelerators automatically. Experimental results show that accelerators generated by POM achieve significant speedup compared to SOTA. The whole compilation stack of POM is **open-sourced** at https://github.com/sjtu-zhao-lab/pom.

IX. Acknowledgements

This work is partially sponsored by the National Natural Science Foundation of China (62102249, 62232015, 61832006) and Shanghai Pujiang Program (21PJ1408200).

References

[1] "About mlir polyhedral optimization." [Online]. Available: https://discourse.llvm.org/t/about-mlir-polyhedral-optimization/1268

[2] "Affine dialect." [Online]. Available: https://mlir.llvm.org/docs/Dialects/Affine/

[3] "Arith dialect." [Online]. Available: https://mlir.llvm.org/docs/Dialects/ArithOps/

[4] "Circuit ir compilers and tools." [Online]. Available: https://circt.llvm.org/

[5] "Clang: a c language family frontend for llvm." [Online]. Available: https://clang.llvm.org/

[6] "Memref dialect." [Online]. Available: https://mlir.llvm.org/docs/Dialects/MemRef/

[7] "Mlir: The case for a simplified polyhedral form." [Online]. Available: https://mlir.llvm.org/docs/Rationale/RationaleSimplifiedPolyhedralForm/

[8] "Tensor dialect." [Online]. Available: https://mlir.llvm.org/docs/Dialects/TensorOps/

[9] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. Van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "Pencil: A platform-neutral compute intermediate language for accelerator programming," in 2015 International Conference on Parallel Architecture and Compilation (PACT), 2015, pp. 138–149.

[10] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2019, pp. 193–205.

[11] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," in Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008). Citeseer, 2008.

[12] L. Chelini, A. Drebes, O. Zinenko, A. Cohen, N. Vasilache, T. Grosser, and H. Corporaal, "Progressive raising in multi-level ir," in 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2021, pp. 15–26.

[13] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze et al., "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 578–594.

[14] 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.

[15] J. Cong and J. Wang, "Polysa: Polyhedral-based systolic array auto-compilation," in 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018, pp. 1–8.

[16] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: Dsl for linear algebra and neural net computations on gpus," in Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2018, pp. 42–51.

[17] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, "Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation," ACM Transactions on Architecture and Code Optimization (TACO), vol. 18, no. 4, pp. 1–23, 2021.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," CoRR, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[19] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 703–718. [Online]. Available: https://doi.org/10.1145/3519939.3523446

[20] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," in Proceedings

of the 34th ACM SIGPLAN conference on Programming language design and implementation, 2013, pp. 127–138.

[21] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2019, pp. 242–251.

[22] 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.

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

[24] J. Li, Y. Chi, and J. Cong, "Heterohalide: From image processing dsl to efficient fpga acceleration," 02 2020, pp. 51–57.

[25] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin et al., "A hardware–software blueprint for flexible deep learning specialization," IEEE Micro, vol. 39, no. 5, pp. 8–16, 2019.

[26] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," ACM SIGARCH Computer Architecture News, vol. 43, no. 1, pp. 429–443, 2015.

[27] L.-N. Pouchet, U. Bondhugula, and C. Bastoul, "Polybench: The polyhedral benchmark suite," http://web.cse.ohio-state.edu/~pouchet/software/polybench/, 2012.

[28] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing dsl," ACM Trans. Archit. Code Optim., vol. 14, no. 3, aug 2017. [Online]. Available: https://doi.org/10.1145/3107953

[29] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: Decoupling algorithms from schedules for high-performance image processing," Communications of the ACM, vol. 61, no. 1, pp. 106–115, 2017.

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

[31] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in International Congress on Mathematical Software. Springer, 2010, pp. 299–302.

[32] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule trees," Jan. 2014, 4th International Workshop on Polyhedral Compilation Techniques, IMPACT 2014 ; Conference date: 20-01-2014 Through 20-01-2014. [Online]. Available: http://impact.gforge.inria.fr/impact2014/

[33] J. Wang, L. Guo, and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga," in The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 93–104. [Online]. Available: https://doi.org/10.1145/3431920.3439292

[34] S. Xiang, Y.-H. Lai, Y. Zhou, H. Chen, N. Zhang, D. Pal, and Z. Zhang, "Heteroflow: An accelerator programming model with decoupled data placement for software-defined fpgas," in Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2022, pp. 78–88.

[35] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation," in 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2022, pp. 741–755.

[36] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "Alphaz: A system for design space exploration in the polyhedral model," in International Workshop on Languages and Compilers for Parallel Computing. Springer, 2012, pp. 17–31.

[37] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li, P. Di, K. Zhang, and X. Jin, "Akg: Automatic kernel generation for neural processing units using polyhedral transformations," in Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1233–1248. [Online]. Available: https://doi.org/10.1145/3453483.3454106

[38] 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.

[39] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance modeling and directives optimization for high-level synthesis on fpga," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 7, pp. 1428–1441, 2020.

[40] R. Zhao, J. Cheng, W. Luk, and G. A. Constantinides, "Polsca: Polyhedral high-level synthesis with compiler transformations," in 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL). Los Alamitos, CA, USA: IEEE Computer Society, sep 2022, pp. 235–242. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/FPL57034.2022.00044

[41] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, 2013, pp. 9–18.

Appendix

A. Abstract

POM is open-sourced on GitHub. Users can compile the project from the source code following the instructions provided in the README. Additionally, a docker image containing a pre-built POM stack is available on Docker Hub, facilitating the reproduction of experimental results in the paper. To enable fast and convenient compilation and execution, a collection of scripts is provided, and a unified script is available for automating the entire workflow of the experiment.

B. Artifact check-list (meta-information)

- **Compilation:** CMake is used to compile the whole project, and version 3.22.0 is recommended. Versions later than 3.14 should also be compatible.
- **Run-time environment:** Ubuntu 20.04.6 LTS is compatible for the experiments. Other Linux distributions may also work but have not been tested.
- **Execution:** `./ae_script.sh`
- **Metrics:** Latency, speedup, resource (DSP, FF, and LUT) usage, power, achieved II, and execution time. More detailed descriptions of these metrics can be found in Section VII-B in the paper.
- **Output:** Experimental results are collected in `/usr/src/workspace/experimental_results.csv`.
- **Experiments:** The key results in the paper are reproduced, including results in TABLE III, V, and VII and Fig. 12. A total of 59 experiments are conducted to reproduce these results.
- **How much disk space required (approximately)?:** About 60GB for software dependencies (namely Vivado and Vitis). About 18GB for the docker image and another 5GB for the generated Vitis_HLS/Vivado reports.
- **How much time is needed to prepare workflow (approximately)?:** About 20 minutes to download the docker image.
- **How much time is needed to complete experiments (approximately)?:** Except for the hardware implementation of the BICG_4096, the other 58 experiments can be completed within 2-3 hours. The hardware implementation of BICG_4096 requires 25-30 hours to complete.
- **Publicly available?:** Yes, the source code of POM is released on GitHub: https://github.com/sjtu-zhao-lab/pom, Docker Hub: https://hub.docker.com/repository/docker/jason0048/pom/general, and Zenodo (DOI: 10.5281/zenodo.10183958)

C. Description

1) How to access: The source code can be accessed on the GitHub repository sjtu-zhao-lab/pom. A docker image containing a pre-built POM stack is accessible on Docker Hub for users to try out POM quickly. The links to the repositories are listed in Section B in the appendix. The repository is also publicly archived on Zenodo (DOI: 10.5281/zenodo.10183958).

2) Software dependencies: Xilinx Vitis HLS 2022.1 and Xilinx Vivado 2022.1 are required in the environment. Version 2022.2 is also compatible with most of the experiments. The detailed download and installation process can be found at https://docs.xilinx.com/r/2022.1-English/ug973-vivado-release-notes-install-license/Xilinx-Unified-Installer.

Note that there are some differences in the internal optimization and scheduling strategies across different versions of Vivado, leading to some differences in the final synthesis results. Results obtained using versions 2022.1/2022.2 should align consistently with those presented in the paper, while using other versions such as 2019 might introduce slight deviations. The experiments are conducted on a Ubuntu 20.04.6 LTS system.

D. Installation

Before experiments, ensure that Xilinx Vitis HLS 2022.1 and Xilinx Vivado 2022.1 are installed in the environment. The docker image can be downloaded from the Docker Hub repository by the instructions below:

```
$ docker pull jason0048/pom:v1
```

When building a new docker container, the directory of Vitis HLS and Vivado need to be mounted. You can verify your Vitis directory with the following instructions:

```
$ ls $(YOUR_VITIS_DIR)
  DocNav  Downloads  Model_Composer
  Vitis  Vitis_HLS  Vivado  xic
```

Then the docker container can be built from the provided docker image:

```
$ docker run -it -v $(YOUR_VITIS_DIR):$
  (YOUR_VITIS_DIR) --name pom jason0048
  /pom:v1 /bin/bash
```

After executing the above instructions, we will enter the docker container automatically, and the current directory will be: `/usr/src/workspace`. We need to perform the following instructions to prepare for the experiments:

```
$ sudo su
$ source ${YOUR_VITIS_DIR}/Vitis/2022.1/
  settings64.sh
$ source ${YOUR_VITIS_DIR}/Vitis_HLS/202
  2.1/settings64.sh
```

E. Experiment workflow

A unified script: `./ae_script.sh` is available for automating the entire workflow of the experiment. The workflow can also be divided into five steps:

- `./build-pom.sh`: Build the POM project, which takes about 1-2 minutes.
- `./run-code.sh`: Compile and run target files, which takes about 20 minutes.
- `./tcl-gen.sh`: Generate the scripts for HLS synthesis and hardware implementation, which takes about 1 minute.
- `./vitis-reports.sh`: Conduct HLS synthesis and hardware implementation, requiring approximately 30 hours in total. It is important to note that the majority of this time is dedicated to the hardware implementation of the BICG_4096 benchmark, aimed at collecting its power statistics. All other experimental results, excluding the power statistics of BICG for problem size 4096, can be generated within 2.5 hours.
- `./results-gen.sh`: Collect and present the experimental results in `experimental_results.csv`, which take about 1 minute.

F. Evaluation and expected results

The results of all 59 experiments are organized in table formats within the file `experimental_results.csv`. This layout provides users with a clear and easily accessible means of verification. The tables include metrics such as speedup, resource usage, power statistics, achieved II, and execution time for benchmarks and applications across different problem sizes. Additionally, the tables also include the actual latency of each design, which is not presented in the paper due to the limited space. The experimental results generated by script `./ae_script.sh` should closely match or be identical to the results shown in Fig. 12 and Table III, V, and VII.

G. Experiment customization

The experiments can be customized in multiple ways. The `config.json` file in the `samples` subdirectory can be adjusted to fit other target platforms. The loop bounds and the optimization strategies of different benchmarks/applications in subdirectory `testbench` can be changed manually. Users can rebuild the target files with the following instructions:

```
$ cd build
$ cmake --build . --target gemm
$ ./bin/gemm
```

H. Notes

We provide a pre-built POM stack for artifact evaluation. The workflow of building the project from the source can be seen in the GitHub repository. Here are some notes:

- **Quick check on the experimental results:** Reproducibility can be achieved for 58 out of 59 experimental results within 3 hours. For a quick check on the results, running `./results-gen.sh` in a separate shell within the Docker environment provides access to partial results. To customize the number of parallel jobs, modify the `max_parallel` factor in `./vitis-reports.sh`. Additionally, excluding the hardware implementation of BICG_4096 can be done by commenting out lines 89-101 in `./vitis-reports.sh`.
- **Potential errors:** When you perform the following instructions within the docker environment, the cmake version will automatically switch to 3.3.2.

```
$ source ${YOUR_VITIS_DIR}/Vitis/2022.1/
settings64.sh
$ source ${YOUR_VITIS_DIR}/Vitis_HLS/202
2.1/settings64.sh
```

If you intend to re-compile the POM project, remember to source `YOUR_VITIS_DIR` only after completing all re-compilations.

I. Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html