POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations

Ruizhe Zhao[*], Jianyi Cheng[†], Wayne Luk[*], George A. Constantinides[†]
[*]Department of Computing, Imperial College London
[†]Department of Electrical and Electronic Engineering, Imperial College London
Email: {ruizhe.zhao15, jianyi.cheng17, w.luk, g.constantinides}@imperial.ac.uk

***Abstract*—Polyhedral optimization can parallelize nested affine loops for high-level synthesis (HLS), but polyhedral tools are HLS-agnostic and can worsen performance. Moreover, HLS tools require user directives which can produce unreadable polyhedral-transformed code. To address these two challenges, we present POLSCA, a compiler framework that improves polyhedral HLS workflow by automatic code transformation. POLSCA decomposes a design before polyhedral optimization to balance code complexity and parallelism, while revising memory interfaces of polyhedral-transformed code to make partitioning explicit for HLS tools; it enables designs to benefit more easily from polyhedral optimization. Experiments on Polybench/C show that POLSCA designs are 1.5 times faster on average compared with baseline designs generated directly from applying HLS on C code.**

***Index Terms*—polyhedral model, high-level synthesis, compiler**

I. INTRODUCTION

To alleviate the heavy effort in building RTL designs, engineers start to describe designs using high-level languages and let the *high-level synthesis* (HLS) tools generate low-level representations. After years of development, HLS has shown its potential in many domains, e.g., high-performance computing and deep learning, but it still has many challenging problems, most of which rely on precise dependence analysis among statements within nested loops. If the input code is restricted to static control flow using only affine expressions for bounds and addresses, i.e., *Static Control Parts* (SCoPs), we can leverage polyhedral models to find solutions [1]–[3]. Using the polyhedral model, which represents loop statement instances as integral points within polyhedra, we can not just formally analyze dependencies, but also automatically transform nested loops for better parallelism [4]. Despite their promise, polyhedral transformation tools are not often used with HLS due to the lack of a proper framework.

The current polyhedral HLS workflow has two stages: using existing polyhedral tools (e.g., Pluto [4], isl [5]) to transform C/C++ code and passing its result to backend HLS tools (e.g., Xilinx Vitis) that produce hardware. Unfortunately, polyhedral and HLS tools are independently developed following their own agenda, which consequently leads to unsatisfactory interfacing and poor user experience. There are two challenges:

1) *SCoP annotation*. Users should annotate their code to indicate which parts are SCoPs that polyhedral tools should transform. These SCoPs should satisfy the affine constraints imposed by polyhedral models, which HLS engineers are not familiar with. Moreover, since polyhedral tools are HLS-agnostic, their optimization can be suboptimal for HLS if SCoPs are annotated improperly.
2) *HLS directives*. Code generators [6], [7] scan the transformed polyhedral representation to generate the optimized program. Unfortunately, this optimized program is normally unreadable, on which users can hardly specify HLS directives to produce efficient hardware. Meanwhile, in most cases we may further transform polyhedral optimized code, e.g., to apply memory partitioning, which is challenging due to the code complexity.

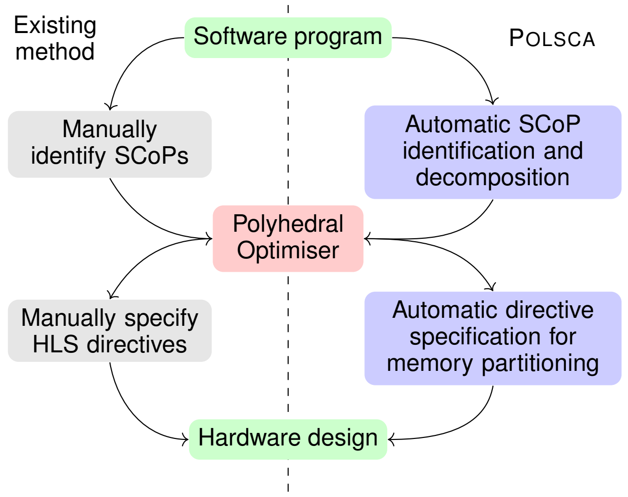

Existing method
Software program
POLSCA
Manually identify SCoPs
Automatic SCoP identification and decomposition
Polyhedral Optimiser
Manually specify HLS directives
Automatic directive specification for memory partitioning
Hardware design

Fig. 1: Summary of our contributions (right) in polyhedral HLS compared with existing methods (left).

Both challenges indicate the lack of automated processes between polyhedral methods and HLS. This paper presents POLSCA[1], a compiler framework that aims to address these challenges by applying pre- and post-polyhedral code transformations. Pre-polyhedral transformations identify and annotate SCoPs by heuristics that balance the code complexity and parallelism, and post-polyhedral transformations inject HLS directives and make certain optimization opportunities, e.g., memory partitioning, explicit for HLS tools. Our prototype POLSCA tool consumes HLS designs in C/C++ and produces

[1]POLSCA = POLyhedral high-level Synthesis using Compiler Automation. Code repository: https://github.com/kumasento/polsca.

```
L0: for p = 0 to P
  // get hashed input
  L1: for i = 0 to M
    for l = 0 to L
      D[i][l] = array[addrIn[p][i][l]]
  // two consecutive matrix multiplication
  L2: for i = 0 to M
    for j = 0 to N
      T[i][j] = 0.0
      for k = 0 to K
        T[i][j] += a * A[p][i][k] * B[p][k][j]
  L3: for i = 0 to M
    for l = 0 to L
      D[i][l] *= b
      for j = 0 to N
        D[i][l] += T[i][j] * C[p][j][l]
  // get hashed output
  L4: for i = 0 to M
    for l = 0 to N
      array[addrOut[p][i][l]] = D[i][l]
```

Fig. 2: A motivating example for POLSCA. It is modified from 2mm [9] with additional indirect memory addresses that create non-SCoPs. There are four separated loop nests below the outermost `L0`: `L1` gathers data from `array` into `D` through a hash table `addrIn`, `L2` and `L3` run two GEMMs (generalised matrix multiplication) `tmp = alpha * A * B` and `D = beta * D + tmp * C`, and `L4` distributes `D` back to `array` using another hash table `addrOut`.

optimization results to backend HLS tools using LLVM IR. This paper is based on our work-in-progress proposal [8].

To summarise, this paper presents three major contributions:

1) auto-identification of SCoPs and generation of polyhedral representation from arbitrary HLS code;
2) transformations of polyhedral-generated code for HLS tools to produce optimized designs;
3) performance evaluation on Polybench/C [9] that contains fully polyhedral-optimisable examples demonstrating the benefits of POLSCA by achieving 1.5 times latency reduction compared with applying HLS on C code.

Two steps will illustrate how polyhedral tools can be integrated with HLS as a framework: first, adopt a polyhedral optimizer that can deal with a wide variety of HLS designs to support the proposed approach; second, further optimize the results from the polyhedral optimizer. This paper focuses on the first step.

II. A MOTIVATING EXAMPLE

Fig. 2 shows a motivating example for POLSCA. This example blends typical polyhedral workload (`L2` and `L3`) as in *2mm* from PolyBench/C [9], with common non-SCoP program parts (`L1` and `L4`) as in *AES* from MachSuite [10]. `L1` and `L4` are not SCoPs due to the existence of non-affine memory accesses to `array`. This example resonates with the two challenges previously mentioned.

First, we need to annotate which parts should be SCoPs. Even though it is trivial for polyhedral experts that `L1` and `L4` should be disregarded due to non-affine accesses, and only `L2` and `L3` are SCoPs, HLS engineers can find it difficult. Moreover, we can realize variables like `M` and `N` should be constants, which is a prerequisite that loops here have affine bounds, but there are cases we should leverage static analysis tools to gather such information. We also need to ensure these variables are constants and propagate them into array definitions so that HLS tools can accept this code.

Second, after applying polyhedral transformation on `L2` and `L3`, we can get several tiled nested loops, but the resulting code can be about 10 times longer to handle different combination of conditions on memory sizes. Even these loops are tiled, HLS tools can hardly exploit the parallelisation opportunities since the complicated code hinders accurate dependence analysis. Partitioning the memory into non-intersected parts makes it explicit that each tile has no conflicting accesses with others. HLS tools provide memory partitioning directives, but they cannot help when a memory size is not divisible by the tiling factor, and there is no automatic procedure to make decisions when there are several valid partitioning tactics and a memory are accessed across SCoPs and non-SCoPs.

We realize that compiler techniques can help address these challenges and overcome the limitations imposed by existing tools. The next section reviews the essential background on polyhedral methods.

III. BACKGROUND

A. Affine expression

An affine expression can be represented as $f(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x}+\boldsymbol{b}$, which maps an integer vector $\boldsymbol{x} \in \mathbb{Z}^n$ into $f(\boldsymbol{x})$ by first multiplying it with a constant integral matrix $\boldsymbol{A} \in \mathbb{Z}^{m\times n}$ and then adding it with a constant integral vector $\boldsymbol{b} \in \mathbb{Z}^m$. Quasi-affine expressions further allow integer divisions using `floordiv` $\lfloor x/c \rfloor$, `ceildiv` $\lceil x/c \rceil$, and the modulo operator `mod`, given the right-hand side operand c an integer constant.

B. Polyhedral representation

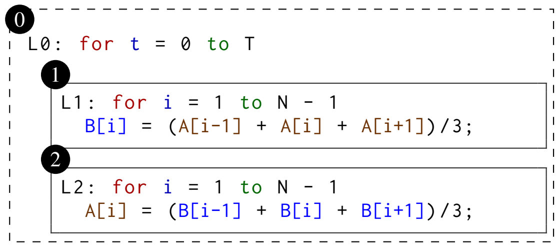
The polyhedral representation of programs uses three key relations [11] to describe a statement:

- *Iteration domains* specify the *affine constraints* on the loop induction variables of a statement.
- *Scattering functions* map from induction variables to logical execution times that abstractly represent *schedule*.
- *Access functions* map from induction variables to the accessed memory addresses.

These polyhedral representations can be derived from the domains of nested loops and the memory access addresses from expressions in the source code. From polyhedral representations, we can construct linear programs and use appropriate solvers to find dependency among statements.

C. Polyhedral scheduling and codegen

This paper adopts Pluto [4], one of the most popular polyhedral scheduling algorithms, for our experiments. Its key idea is to solve an optimization problem that minimizes communication cost—modeled by the sum of distances between dependent memory accesses—among loop tiles to achieve better locality and higher level of parallelism.



```
L0: for t = 0 to T
  L1: for i = 1 to N - 1
    B[i] = (A[i-1] + A[i] + A[i+1])/3;
  L2: for i = 1 to N - 1
    A[i] = (B[i-1] + B[i] + B[i+1])/3;
```

Fig. 3: Jacobi-1d algorithm. We can either annotate a big SCoP ⓿ over the outermost loop `L0`, or two small SCoPs ❶ and ❷ covering inner loops `L1` and `L2`. Memory accesses to arrays `A[N]` and `B[N]` make `L0` and `L1` interdependent.

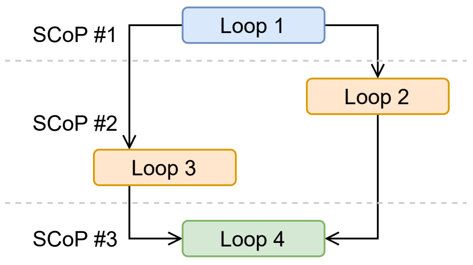


Fig. 4: Demonstrating the SCoP decomposition algorithm. In this particular example, we have 4 consecutive loops at the same depth, and there are interdependecies (arrows) among loop 1 and 2, 1 and 3, 2 and 4, 3 and 4. The minimum cuts required for SCoP covering loop 2 and 3 is 0, which will propagate through Eq. (1) and lead to the optimal solution as shown that uses 2 cuts (dashed horizontal lines).

The solutions provide a transformation matrix on the scattering functions. The result of transformation is still a polyhedral representation, and we should leverage polyhedral code generators to convert it back to textual source code [6], [7]. Depending on the shape of the input polyhedra and certain configurable hyperparameters, the generated code can vary much: some can be challenging for further optimization. Note that polyhedral optimizers other than Pluto can also be used.

IV. Methodology

A. Pre-polyhedral transformation

POLSCA searches within the user input program for nested loops and discovers those that satisfy the affine constraints—the loop bounds should be affine expressions, and all memory accesses reach affine addresses—imposed by polyhedral methodology. To get valid input for polyhedral tools, POLSCA should annotate certain loop nests as SCoPs that will be transformed later. An annotated SCoP will be in the polyhedral representation that the polyhedral tools can recognize and operate on. This section demonstrates how POLSCA decides which loop nests to annotate, specifically, how to determine the SCoPs when there are multiple affine loop nests.

1) Hazardous interdependent loops: Consecutive loop nests can be covered by a single SCoP, or by multiple separated SCoPs. These two approaches can produce significantly different results if the loop nests are interdependent, i.e., they have hazardous read and write accesses to the same array, at the same address. When provided a large SCoP, polyhedral algorithms intend to schedule interdependent loops closer to improve locality and parallelism. This strategy would be beneficial for multi-core CPUs, however, the resulting program can be too complicated to be optimized by HLS tools. For example, a polyhedral optimizer can produce about 30 loops with sophisticated control from the Jacobi-1d code (Fig. 3), which originally just have 2 loops that are interdependent on arrays `A` and `B`. This situation also applies to our motivating example (Fig. 2), specifically, the two consecutive matrix multiplication loop nests.

2) SCoP annotation heuristic: Instead of modifying the sophisticated internals of Pluto, POLSCA addresses this problem by a heuristic: avoiding SCoPs containing interdependent loops can produce simple and efficient HLS code. This heuristic is based on our observation that SCoPs without interdependent loops will not trigger major loop reconstruction in polyhedral optimizer, and therefore, the optimized code will still close to the simple form of the human-readable user input. The simpler the optimized code is, the more efficient the final design will be, since the backend HLS tools need less effort to resolve statement dependencies, and consequently, better identify parallelisation and pipelining opportunities. Future HLS releases may be capable of handling more complicated code, in which case POLSCA can search for the optimal set of SCoPs that should apply such heuristics.

In Fig. 3, annotating two SCoPs on the two inner loops produces a design with just two independently tiled loops, which is sufficient to achieve much higher parallelism than the original code. Moreover, only separately annotating the two inner affine loop nests in Fig. 2 produces two tiled matrix multiplication subroutines, which HLS tools are familiar with.

3) SCoP decomposition: To apply this heuristic, we should decompose SCoPs having interdependent internal loops into pieces such that their internal loops only have external dependencies. The simplest solution is to annotate a SCoP per loop, which is suboptimal since loops that are not interdependent cannot be transformed together, e.g., we cannot further simplify loops accessing different arrays through fusion.

We prefer a top-down solution: starting from a big SCoP containing all consecutive affine loop nests, POLSCA tries to find *the minimum number of cuts* on all inter-loop dependencies to remove them. Adapted from graph theory, we define a cut on the inter-loop dependency graph of a single SCoP, in which vertices are loop nests within the SCoP and edges are their dependencies. Each cut decompose a SCoP into two parts, and aiming at minimizing the number of cuts avoid creating excessively small SCoPs that hinder optimization.

This paper considers the minimum cut problem on a graph consisting of consecutive loop nests at the same depth. Suppose there are N loop nests, we can assign an integer label to each loop following their execution order, e.g., 0 to the first loop nest and $N-1$ to the last, and any partition of this graph

```
// Tile loops
for i = 0 to 2
  for j = 0 to 3
    for k = 0 to 4
      // Point loops
      for ti = i * 32 to (i + 1) * 32
        for tj = j * 32 to (j + 1) * 32
          for tk = k * 32 to (k + 1) * 32
            C[ti][tj] += A[ti][tk] * B[tk][tj];
```

$\Downarrow$ outlining PEs

```
for i = 0 to 2
  for j = ...
    for k = ...
      PE(i, j, k, A, B, C);
```

Fig. 5: Sketch of polyhedral-transformed matrix multiplication. The loop nest initially has a depth of 3, and each loop has been tiled by a factor of 32 as listed here. The 3 outer loops `i`, `j`, `k` are referred to as tile loops, and the inner loops `ti`, `tj`, `tk` are point loops. After PE outlining, the inner 3 point loops are wrapped into a new function `PE`.

can be represented as an interval. For instance, an interval $[S, E)$ covers the loop nests with labels from S to $E-1$. Moreover, we can formulate a cut as a partition of intervals.

$$\min_{cut}[S,E) \leftarrow \mathop{\mathbf{minimum}}_{i\in[S+1,E)} \left\{\min_{cut}[S,i) + \min_{cut}[i,E) + 1\right\} \quad (1)$$

Following the discussion above, we can describe this minimum cut problem as Eq. (1). For a range of loops labeled from S to $E-1$, the minimum number of cuts ($\min_{cut}$) equals to the minimum sum of the solutions for two sub-ranges cut at one particular position i. The base case is those ranges without interdependent loops, which requires 0 minimum cut. This equation can lead to a straightforward dynamic programming algorithm with polynomial time complexity. One example solution found by this algorithm is shown in Fig. 4.

B. *Post-polyhedral transformations*

This section introduces transformations applied after receiving the optimized code generated from the polyhedral tools, which are not optimal to describe HLS designs due to the lack of directives and their undesirable code structure. Directives help HLS tools find better designs based on user-provided domain knowledge, and well-structured code can ensure HLS tools exploring all dependencies among instructions.

This section mainly describes how POLSCA makes memory partitioning opportunities, which are critical to the achievable level of parallelism, visible to HLS tools. We also demonstrate additional loop transformations that set the boundary between software and hardware. The expected final production is an HLS design that clearly indicates parts that are hardware processing elements (PEs) with properly partitioned memory interface and HLS directives.

1) Point loops as processing elements: A tiled loop nests normally have two separable parts, *tile loops* that iterate over tiles, and *point loops* that iterate within each tile (Fig. 5). Heuristically, it is desirable to unroll the tile loops so that the HLS tools can have a better chance to utilize the parallelism that tiling implies. Point loops are less common for unrolling, and we normally leverage pipelining to improve performance.

POLSCA draws a boundary between tile and point loops by *outlining* point loops into functions (Fig. 5). Replacing nested loops with a single caller can help HLS tools realize there is no dependence among tiles and find parallelisation opportunities.

2) Memory partitioning: HLS tools can be too conservative to parallelize PE instances across tiles, mainly because each PE accesses to the same memory object, which does not specify whether they access non-intersected partitions. For instance, in Fig. 5, PEs of different `i`, `j` values access to different 32×32 partition of array `C`, but it is not explicit.

A common approach to address this problem is memory partitioning, i.e., breaking a single memory block into smaller parts that are accessed by different PE instances. Here we assume only using the block partition scheme [3]. There are corresponding optimization directives in HLS tools that can transform the input code, e.g., turn `A[64]` into `A[2][32]` if partitioned by blocks with a factor of 32, to apply memory partitioning. But it is still challenging for users to identify which memory blocks should be partitioned.

Concretely, there are two major challenges:

❶ A single PE can access multiple partitions, e.g., the statement `A[i-1] + A[i] + A[i+1]` as in Jacobi-1d (Fig. 3) accesses two neighboring tiles on either boundary (`i` = 0 or 31 supposing the factor is 32).

❷ We can decide the partitioning factor by the size of the iteration domain of the memory access, e.g., if `i` iterates 32 times, we partition `A[i]` by a factor of 32. But the same array can be accessed in different loop nests of various domain sizes, and we should reconcile them to one value. These dependencies should be explicit to HLS.

POLSCA provides two corresponding solutions, one to analyze how many partitions that each PE instance needs to access given a block partition factor, and the other one reconcile a single partition factor size for each array.

3) Affine analysis on accessed partitions: Figuring out the number of partitions that a single PE instance needs to access is crucial to understanding the profitability by using a specific block partition factor.

Definition 1 (Profitable partition factor)**.** A partition factor T is more profitable than another T' if it can get a PE instance that accesses fewer partitions, or has the same partitions but exposes higher level of parallelism, denoted by $T' \prec T$.

For instance, partitioning `A` in Fig. 5 by 32 is more profitable than 33, which requires some PE instances to access more than 1 partition; and 64, the PE instance of which still accesses 1 partition but reduces the potential level of parallelism by half.

As suggested previously, PEs only have point loops resulted from polyhedral transformations, which are SCoPs and satisfy the prerequisites for affine analysis. We limit the access pattern to *scalar* address in the following discussion, while these results can be extended to general cases.

```
void top(float A[70], int B[70])
  for i = 0 to 70 { A[B[i]] = ... } // Non-SCoP
  for i = 0 to 3  { PE(i, A) }       // SCoP
void PE(int i, float A[70])
  for ti = i * 32 to (i + 1) * 32
    A[ti] = A[ti] + A[ti-1]
```

(a) Before memory partitioning. Array A is accessed by both non-SCoP and SCoP loops. The tile factor in the polyhedral-transformed SCoP is 32. Special handling for invalid access A[-1] is omitted.

```
void top(float A[3][32], int B[70])
  for i = 0 to 70 { A[B[i] / 32][B[i] % 32] = ... }
  for i = 0 to 3  { PE(i, A[i], A[i-1]) }
void PE(int i, float P[32], float Q[32])
  for ti = 0 to 32
      P[ti] = P[ti] + (ti == 0 ? Q[(ti-1) % 32] :
                                         P[(ti-1) % 32] )
```

(b) After memory partitioning. A is partitioned by a factor of 32 after reconciling both accesses. Dynamic addresses from B are transformed to access the partitioned array, and the PE interface is extended to pass in an extra partition for the boundary accesses.

Fig. 6: A memory partitioning example that applies partition factor reconciliation and PE interface extension.

Problem 1 (Determine partitions accessed differently)**.** For any two addresses $f(i)$ and $g(i)$ being accessed, where i is an induction variable, f and g are affine transformations from i to addresses, e.g., $f(i) = ai + b$, they access different partitions if there exists solution for the following:

$$\lfloor f(i)/B \rfloor \neq \lfloor g(i)/B \rfloor, \quad Tk \leq i < Tk + T,$$

in which T is the tile size for the point loop, k is an induction variable of an outer loop that encloses the loop that i belongs, and B is the block partition factor.

By solving Problem 1 among all memory accesses for a block partition factor B, we can get the number of partitions that a PE will access, which consequently determines the profitability (Definition 1) of B.

Even though Problem 1 is solvable by a linear solver, the *candidate set* of B is as large as the target memory size, e.g., B used to partition a 1-D memory of size 1,024 can take at most the value of 1,024. Repeatedly running a linear solver for all B candidates can be computationally intractable. Therefore, we propose the following theorem that directly relates partition factor B and a predetermined parameter, tile size T.

Theorem 1. *Given block partition factor B equals to a predetermined tile size T, and i has an iteration domain $[Tk, Tk + T)$ where k is an outer loop induction variable, then the partitions accessed for a set of addresses, $\{f_j(i) = a_j i + b_j, j = 1, \dots, M\}$, are included in the following set:*

$$\bigcup_{j=1}^{M} \left[a_j k + \left\lfloor \frac{b_j}{T} \right\rfloor, a_j k + 1 + \left\lfloor \frac{b_j - 1}{T} \right\rfloor \right].$$

Proof. First of all, $\lfloor ax{+}b/x \rfloor = a + \lfloor b/x \rfloor$. Given that i has a iteration domain $[Tk, Tk + T)$, for each address f_j, we can apply this equation to get the lower and upper bound of $f_j(i)$. Specifically,

- its lower bound is $\lfloor f_j(Tk)/T \rfloor$ when $i = Tk$;
- and its upper bound is $\lfloor f_j(Tk{+}T)/T \rfloor$ when $i = Tk + T$.

The result can be achieved by taking the union of the lower and upper bounds from all addresses. □

Example 1. For statement A[i-1] + A[i] + A[i+1], we have:

$$(a_1, b_1) = (1, -1) \quad (a_2, b_2) = (1, 0) \quad (a_3, b_3) = (1, 1).$$

Using Theorem 1, the union of all of their partitions is:

$$\{k - 1, k\} \cup \{k\} \cup \{k, k + 1\} = \{k - 1, k, k + 1\}.$$

For tile k, there are three neighboring partitions accessed.

We leverage Theorem 1 to determine the optimal partition factor in the following sections.

4) Partition factor reconciliation: The optimal partition factor for a memory block is mainly determined by its accesses and their parent loop nests. Given that the amount of valid candidates for a partition factor can be enormous, and finding a global optimum is computationally intractable, we choose to search for a local optimum using an algorithm derived from the following propositions.

Theorem 2. *If a dimension of an array is accessed by an induction variable i, which has an affine bound of $[Tk, Tk{+}T)$ where T is a predetermined tile size and k is an outer loop induction variable, then it is more profitable to have a block partition factor B that equals to T than other B candidates.*

Proof. Let us consider four possibilities for the value of B: B equals to T; B is an integer multiples of T, denoted by nT; B is a factor of T, denoted by T/n; and none of the above. Here, n is an integer that is larger than 1.

1) If $B = T$, there is a single partition that will be accessed by a PE instance;
2) If $B = nT$ and $n > 1$, there is also a single partition that will be accessed, but within each partition, there are n number of sub-partitions of size T that can potentially be parallelised, which makes nT less profitable than T.
3) If $B = T/n$ and $n > 1$, a PE instance needs to access n partitions, which induces T/n is less profitable than T.
4) When partitioning by any of the rest candidates, which are not an integer multiple or a factor of T, a PE instance must access at least two partitions, which makes it less profitable than T.

Therefore, we conclude that $B = T$ is the most profitable block partition factor under the given presumptions. □

Theorem 2 helps determining the partition factor for each access of a memory block. And if it is satisfied, based on Theorem 1, we can compute the total number of partitions accessed. In case the determined values are not identical, for every pair of access A and block partition factor B, suppose $N(A, B)$ is the number of accessed partitions per PE instance, and $P(A, B)$ is the achievable level of parallelism, we introduce Definition 2 for reconciliation.

Definition 2 (Profitability among multiple accesses)**.** Given memory accesses $A_{1\dots n}$, each of which is an address that is an

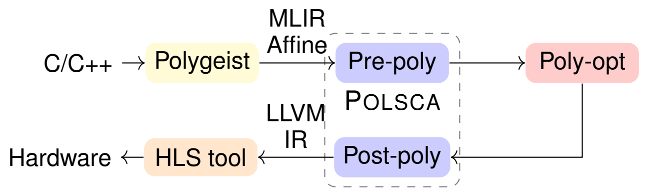


Fig. 7: The overall workflow of POLSCA. Our contributions are highlighted in the dashed region. Pre-poly and post-poly are the transformations described in Section IV.

affine mapping from loop induction variables to a scalar, and block partition factors $B_{1\dots n}$, B_p is (local-)optimally profitable if for any other B_q:

- B_p has less number of partitions, i.e., $\sum_i N(A_i, B_p) < \sum_i N(A_i, B_q)$, or
- B_p can provide higher level of paralellism when the number of partitions is the same, i.e., $\sum_i P(A_i, B_p) > \sum_i P(A_i, B_q) \wedge \sum_i N(A_i, B_p) = \sum_i N(A_i, B_q)$.

Notably, we use the properties aggregated by summation over all accesses, which implies that we treat every access of equal importance. This approach is potentially not global-optimal if some PEs have much longer latency than the others, and in those cases, we should properly weight each term before aggregation. These particular cases make no appearance in the benchmark examples that this paper covers, and therefore, we decide to stick to the local-optimal solution.

Finding the most profitable factor under Definition 2 for each memory has computationally tractable time complexity of $\mathcal{O}(n^2)$, imposed by calculating $N(\cdot,\cdot)$ and $P(\cdot,\cdot)$. Suppose there are m memory blocks in the input program, the overall time complexity for the partition factor reconciliation algorithm becomes $\mathcal{O}(mn^2)$. After that, we can transform the input code to accommodate the partition factor, e.g., `A[i]` to `A[i floordiv 32][i mod 32]` if partitioned by a factor of 32.

V. EVALUATION

A. Implementation details

POLSCA is built upon MLIR [12], a compiler infrastructure that features multi-level representation and transformation. The polyhedral representation and affine analysis are implemented upon the Affine dialect, a collection of operations and representations that are dedicated to polyhedral modeling. The input code that POLSCA accepts is in C/C++, which are translated into MLIR through Polygeist [13]. The internal polyhedral optimizer is Pluto [4], and POLSCA translates the polyhedral representation between MLIR and Pluto. The final outcome from POLSCA is in LLVM [14] IR that can be consumed by backend HLS tools. HLS directives are specified in the LLVM IR as well, following the format that the target HLS tool understands. Fig. 7 shows how they interconnect.

We experiment POLSCA on Xilinx Vitis HLS (2020.2) and leverage its LLVM frontend following open-sourced instructions[2]. We setup an ideal scenario that the FPGA board is large enough (Zynq UltraScale+ XQZU29DR), since this paper mainly focuses on pushing the performance boundary by polyhedral transformations without considering design space exploration. We use co-simulation to collect precise cycle numbers of latency and verify the correctness of the generated design. As the top module we synthesize only contains memory interfaces and no memory instances are instantiated, there is no BRAM or URAM usage introduced by our designs.

The baseline of our experiment is using POLSCA without any polyhedral transformation, i.e., the input C/C++ will be translated to MLIR through Polygeist first, then translated to LLVM IR immediately without any optimization, and finally passed to Vitis HLS. The rationale behind this choice is explained in Section V-E.

B. Polybench/C

POLSCA is evaluated on Polybench/C [9], one of the most popular benchmark suites for high-performance computing that only contains code that is fully polyhedral-optimizable, i.e., their core functions are all affine loop nests. While POLSCA can process all Polybench examples, we select 7 typical examples covering all categories, including linear algebra (gemm, gesummv, bicg, mvt), stencil (jacobi-1d, jacobi-2d), and medley (floyd-warshall). Even though POLSCA is able to use arbitrary tile sizes, the HLS backend cannot synthesize array partitioned with a large (> 20) factor, therefore, we manually select a set of tile sizes for each example, which can be automated in the future.

Table I summarizes our results. POLSCA can achieve 1.5 times speed-up averaged by geomean over the baseline directly applying HLS tools on C source code. More importantly, without POLSCA, applying polyhedral optimization directly results in worse performance than the baseline. In general, the performance benefit gained from POLSCA over plain polyhedral designs is mainly from:

1) heuristic SCoP annotation that can avoid generating over-complicated polyhedral-transformed code, especially for stencil examples like jacobi-1d and jacobi-2d, which will result in at least 10 times longer code after polyhedral optimization without conventionally recognizable tiling structure as in Fig. 5;
2) memory partitioning that practically generates parallelizable designs, for instance, gemm being polyhedral-transformed into a blocked version is a typical HLS input that can be parallized, however, due to the lack of directives and code transformation, the performance ($63,580$k cycles) is much slower than the original ($10,610$k cycles).

Among these examples, linear algebra and stencil applications can be optimized to a level (about 2 times) better than the others. There are two examples that POLSCA does not perform well: bicg and floyd-warshall, which can attribute to the default behavior of the polyhedral optimizer Pluto.

Although POLSCA optimized bicg has worse wall clock time than the baseline, it is mainly due to the fact that Fmax is reduced more (about 1.71 times) than the level of reduction in the number of cycles (about 1.65 times). This

[2]https://github.com/Xilinx/HLS/tree/2020.2

TABLE I: Evaluation of our approach on a set of benchmarks. **NA** stands for whether an example contains non-affine loop nests or not. The *base* column covers the designs directly synthesised using Polygeist, the *poly* column covers the designs directly synthesised using the MLIR from polyhedral, and *ours* covers the designs synthesised using the code from POLSCA. **Geom. Mean** measures the geometric mean of values from the baseline over other corresponding scenarios.

Benchmarks	NA	**LUTs**			**DSPs**			**Cycles (k)**			**Fmax (MHz)**			**Wall clock time (μs)**		
		base	poly	ours	base	poly	ours	base	poly	ours	base	poly	ours	base	poly	ours
bicg	✗	1,525	3,259	17,144	15	18	144	960	1,625	580	103	60	60	9,318	27,077	9,661
floyd-warshall	✗	641	1,439	5,326	1	0	24	250,000	250,024	203,192	106	109	103	2,358,490	2,293,798	1,972,737
gemm	✗	1,895	4,116	13,720	26	29	191	10,610	63,580	6,009	103	102	103	103,009	623,335	58,344
gesummv	✗	1,584	3,418	18,610	15	27	137	375	563	132	103	60	60	3,641	9,388	2,191
jacobi-1d	✗	1,463	5,215	7,321	14	14	56	163	1,082	123	103	100	103	1,581	10,818	1,197
jacobi-2d	✗	3,015	13,955	18,169	17	28	150	36,908	19,244	14,297	99	96	103	372,807	3,229,785	138,801
mvt	✗	1,635	1,943	8,010	14	16	56	1,290	669	698	60	60	60	21,506	11,146	11,641
radix sort	✓	3,827	2,149	21,817	0	0	0	197	229	148	78	78	76	1,407	1,633	1,025
Geom. Mean		1×	0.5×	0.1×	1×	0.8×	0.2×	1×	0.5×	1.7×	1×	1.2×	1.1×	1×	0.4×	**1.5**×

suggests that the improvement in parallelism leads to increased design complexity, which makes the HLS tools less effective in enhancing performance. The major source of complexity is the additional loops generated by the polyhedral optimizer originally targeting better CPU cache locality.

Many polyhedral optimizes such as Pluto are not exceptionally good at optimizing the Floyd-Warshall algorithm [13]. The resulting program can be tiled and partitioned, but the dependencies among the tiles are hard to exploit for enhancing parallelism, so performance improvement is minor.

C. Resource usage

We notice that the usage of resources is scaled at a higher rate than performance gain, which is mainly due to the general-purpose nature of the Pluto polyhedral optimizer. Pluto optimizes for less communication among tiles, which does not explicitly lead to solutions that are more resource-efficient, e.g., using local buffers to reduce control and save resources.

Also, the memory partition expression we introduce, which involves floordiv and modulo arithmetic, makes it more difficult for Vitis HLS to find expressions like MAC (multiply and accumulate), so leads to higher resource usage. For instance, regarding the core arithmetic statements in gemm, the DSP usage scales from 25 to 103 (4.1 times), while LUT usage increases from 1,197 to 9,733 (8.1 times), when comparing the original design with the POLSCA-optimized one. Our future work will include additional optimizations to reduce resource.

D. Partially polyhedral-optimizable code

We evaluate POLSCA on a partially polyhedral-optimizable example selected from MachSuite [10], radix sort. Radix sort contains affine loop nests, as well as non-affine ones that have indirect memory accesses, i.e., accessing by addresses looked up from another memory like `A[B[i]]`.

POLSCA can not just recognize affine loop nests and annotate them as SCoPs, but also perform memory partition based on a factor reconciled from all affine and non-affine accesses. We can gain about 37% speed-up over the baseline using POLSCA. We also run just polyhedral optimization on the affine parts of the code without any POLSCA transformations, and the result becomes about 16% worse due to the lack of memory partitioning and the increased code complexity. It is worth mentioning that all these transformations are included in POLSCA, and therefore, easy for application builders to adopt.

E. Choice of baseline

The baseline of our approach is obtained from applying HLS to the output of Polygeist (Fig. 7). Another possible baseline is applying HLS directly on the C/C++ design without going through Polygeist. The reason for our choice is to exclude the benefits of Polygeist in our evaluation, and to minimize the effects due to the disparity between Clang, the frontend compiler that Vitis HLS uses, and MLIR, which POLSCA uses.

Moreover, we notice that, for the examples that we evaluate, the geomean of LUT and DSP usage from Clang baselines are 1.25 and 1.20 times more than the MLIR baselines, while the geomean of the wall clock time stays the same between both baselines. This suggests that the MLIR baselines are even stronger than Clang baselines, which can support our choice.

VI. RELATED WORK

A. Polyhedral HLS modeling

The following papers provide the theoretical foundation for polyhedral HLS. Zuo et al. [15] focus on optimizing data-dependent multi-block HLS designs by polyhedral modeling. They design their own polyhedral optimizer that can leverage loop transformations like skewing that can result in efficiently parallelised code. The programs that they focus on are similar to the SCoPs with interdependent loop nests (Section IV-A1), however, as we aim to apply the Pluto optimizer for any input code, their approach is not applicable in our scenario, and our solution by decomposing SCoPs cannot be replaced.

Pouchet et al. [2] address the data-reuse problem in HLS designs through polyhedral modeling, and they propose an approach that can minimize the data-transfer bandwidth with under on-chip buffer usage constraints. As POLSCA currently assumes sufficient on-chip memory capacity, POLSCA focuses on the achievable parallelism constrained by memory partitioning, which is orthogonal to the problem domain in [2]. Wang et al. [3] provide a general solution to memory partitioning problem in HLS using polyhedral modeling, which covers all variants of partitioning techniques besides blocking and takes into account the constraints of available ports per memory. POLSCA can be extended to cover the scenarios as in [3], and still, POLSCA has its novel contribution to solving memory partition on more complicated scenarios that involve multiple loop nests. There are other research that improves the polyhedral optimizer for HLS, e.g., Liu et al. [1] leverage polyhedral modeling for loop splitting that can benefit pipelining.

Moreover, PolySA [16] and AutoSA [17] apply polyhedral modeling on a dedicated architecture, systolic array, and achieve significant performance improvement. POLSCA does not target a specific architecture: its architectures are determined by the Pluto objective function that minimizes communication cost among tiles, which is a general rule and will not lead to a specific architecture. Nevertheless, it would be interesting to see if POLSCA can be extended to further enhance the designs generated from other polyhedral optimizers such as AutoSA.

B. *Polyhedral HLS toolchain*

Potholes [18] shares the idea of building compiler transforms as ours. Potholes implements various transformations within the LLVM infrastructure and can optimize for typical affine programs in image processing, while Potholes does not support annotating SCoPs for more complicated code. Also, Potholes is restricted to C semantics, which makes it more difficult to extend its transformations and carry out affine analysis as POLSCA.

Zuo et al. [19] describe several practical techniques to improve polyhedral code generation for HLS, including address expression optimization, tiling optimization, etc., which can significantly improve the performance of polyhedral HLS, especially for stencil examples. POLSCA instead mainly tackles problems at the toolchain-level, e.g., annotating SCoPs and generating memory partitioned code, which are orthogonal to [19]. It will be promising to integrate the techniques in [19] into POLSCA to make polyhedral HLS more efficient.

ScaleHLS [20] is a new framework that also leverages MLIR, especially Affine, for HLS, however, there is no application of polyhedral scheduling in ScaleHLS.

VII. SUMMARY AND FUTURE WORK

This paper presents POLSCA, a novel compilation framework that optimizes the workflow of applying polyhedral transformations on HLS designs. POLSCA is designed to take arbitrary C/C++ code as input, and produce the corresponding HLS code that are optimized for better consumption by the backend HLS tools to generate high-performance hardware. POLSCA has two major passes, one that happens before polyhedral transformations can auto identify and decompose SCoPs from input HLS code, the other one takes the code generated from the polyhedral scheduler and turns it into a form with PE functions and memory partitions. These pre- and post-polyhedral passes together produce optimized polyhedral HLS designs with simplicity and explicit dependencies.

For future work, we will further explore the design space to optimize designs for resource efficiency. It is also interesting to search for the best set of polyhedral scheduler hyper-parameters with POLSCA. Moreover, we will adapt POLSCA to support other polyhedral schedulers, such as those in isl [5] and AutoSA [17], to provide alternative solutions to Pluto.

ACKNOWLEDGMENTS

The support of the United Kingdom EPSRC (grant numbers EP/I012036/1, EP/L00058X/1, EP/L016796/1, EP/N031768/1, EP/K034448/1 and EP/P010040/1), European Union Horizon 2020 Research and Innovation Programme (grant number 671653), Corerain, Intel, and Maxeler is gratefully acknowledged. For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

REFERENCES

[1] J. Liu, J. Wickerson, and G. A. Constantinides, "Loop Splitting for Efficient Pipelining in High-Level Synthesis," in *FCCM*, 2016.
[2] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *FPGA*, 2013.
[3] Y. Wang, P. Li, and J. Cong, "Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis," in *FPGA*, 2014.
[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI*, 2008.
[5] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *ICMS*, 2010.
[6] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT*, 2004.
[7] T. Grosser, S. Verdoolaege, and A. Cohen, "Polyhedral AST Generation Is More Than Scanning Polyhedra," *TPLS*, 2015.
[8] R. Zhao and J. Cheng, "Phism: Polyhedral High-Level Synthesis in MLIR," 2021.
[9] T. Yuki, "Understanding polybench/c 3.2 kernels," in *IMPACT*, 2014.
[10] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014.
[11] C. Bastoul, "OpenScop: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools," Tech. Rep., 2012.
[12] 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," 2020.
[13] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist : Raising C to Polyhedral MLIR," in *PACT*, 2021.
[14] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2014.
[15] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *FPGA*, 2013.
[16] J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *ICCAD*, 2018.
[17] J. Wang, L. Guo, and J. Cong, "AutoSA: A polyhedral compiler for high-performance systolic arrays on fpga," in *FPGA*, 2021.
[18] S. Bayliss, "PoTHoLeS: Polyhedral Compilation Tool for High Level Synthesis," https://github.com/SamuelBayliss/Potholes, 2014.
[19] W. Zuo, P. Li, D. Chen, L. N. Pouchet, S. Zhong, and J. Cong, "Improving polyhedral code generation for high-level synthesis," in *CODES+ISSS*, 2013.
[20] 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 *HPCA*, 2021.