Phism: Polyhedral High-Level Synthesis in MLIR

(Work in Progress)

Ruizhe Zhao
ruizhe.zhao15@imperial.ac.uk
Imperial College London
UK

Jianyi Cheng
jianyi.cheng17@imperial.ac.uk
Imperial College London
UK

ABSTRACT

Polyhedral optimisation, a methodology that views nested loops as polyhedra and searches for their optimal transformation regarding specific objectives (parallelism, locality, etc.), sounds promising for mitigating difficulties in automatically optimising hardware designs described by high-level synthesis (HLS), which are typically software programs with nested loops. Nevertheless, existing polyhedral tools cannot meet the requirements from HLS developers for platform-specific customisation and software/hardware co-optimisation. This paper proposes ϕ_{sm} (Phism), a polyhedral HLS framework built on MLIR, to address these challenges through progressive lowering multi-level intermediate representations (IRs) from polyhedra to HLS designs.

1 INTRODUCTION

High-level synthesis (HLS) can transform software programs in C-like languages into hardware designs, and polyhedral optimisation can provide elegant solutions for various problems in this process, e.g., loop pipelining and splitting [16, 17], loop transformations [26], design generation [7, 24], etc. It is mainly due to many HLS programs, originally described in C-like languages, have regions with control flow and dependence relations that can be formulated as affine expressions at compile time. These regions, conventionally referred as Static Control Parts (SCoPs) by polyhedral research, are where polyhedral optimisation can be fully leveraged.

Nevertheless, existing polyhedral tools are incapable of keeping up with the recent progress in HLS: more target platforms, design models [6, 13], and applications are being supported, while polyhedral tools, e.g., isl [22] and Pluto [3], are not versatile enough to be customised for them. Existing papers [2, 25] can partially address this challenge by customising some processing stages when lowering from polyhedral representations, but we need a more comprehensive approach to meet the current and future demands.

Inspired by recent work on domain-specific compiler [12] and hardware synthesis [6], we are motivated to *progressively lower* from polyhedra to HLS designs, so that we can customise each processing stage at its right abstraction level and define composable and reusable transformations for future extension. MLIR [15] is a perfect fit for our objectives as a compiler infrastructure that effectively supports multi-level intermediate representations (IRs) definition and transformation, under the concept of *dialect*. Therefore, this paper proposes ϕ_{sm} (Figure 1), the first MLIR-based polyhedral HLS tool featuring progressively lowering by:

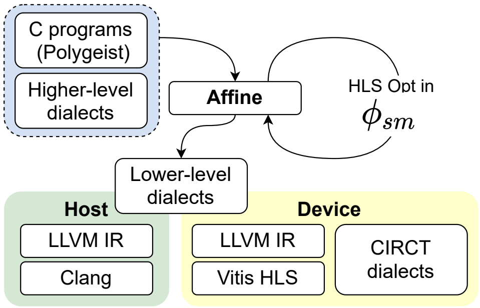


Figure 1: ϕ_{sm} overview. Polygeist [20] provides both the conversion from C and the polyhedral optimisation on Affine.

(1) implementing HLS optimisation at the right abstraction levels during progressively lowering;
(2) leveraging dialects, e.g., Affine [18], to build transformations for HLS on polyhedral representations;
(3) connecting with various sources, e.g., C or other higher-level dialects, and targets, e.g., Vitis [14] or CIRCT [6].

In this way, ϕ_{sm} can better leverage polyhedral optimisation for HLS, and therefore, provide an efficient hardware design method.

2 BACKGROUND AND RELATED WORK

Polyhedral optimisation. There are decades of research on representing programs in polyhedra [9] and transforming them for better performance [3]. Polyhedral optimisation transforms polyhedra extracted from original programs, and the resulting polyhedra should be converted back through non-trivial polyhedral code generation [1, 5, 11] for various platforms [23, 25].

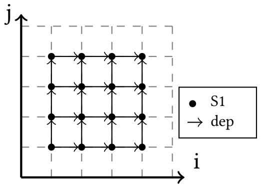
MLIR. MLIR is a compiler infrastructure for building IRs and their transformations [15]. Here, IRs are in static single assignment (SSA) [21] forms, and MLIR provides the *dialect* mechanism to define IRs for domain-specific problems within the ecosystem [8]. This paper focuses on Affine [18], which is designed for representing polyhedral programs, and can be translated from C and optimised by existing polyhedral tools enabled by Polygeist [20].

Related work. PoTHoLeS [2] provides a polyhedral compilation tool for HLS, and Zuo et al. [25] describe several polyhedral code generation techniques for HLS. ϕ_{sm} is compatible with these prior approaches and is more versatile to cover recent HLS techniques and software/hardware co-optimisation with progressive lowering.

LATTE '21, April 15, 2021, Virtual, Earth

```
#pragma scop
for (i = 1; i < N; i ++)
  for (j = 1; j < N; j ++)
S1: A[i][j] = A[i-1][j] + A[i][j-1];
#pragma endscop
```

(a) The input C code. Its nested loops are both in the SCoP region and can be represented as a 2-D polyhedron.



(b) The polyhedron. Each S1 instance, i.e., a loop iteration, is a dot. Dependencies are marked as arrows.

```
for (t1=0;t1<=floord(n-1,16);t1++) {
  lbp=max(0,ceild(32*t1-n+1,32));
  ubp=min(floord(n-1,32),t1);
  for (t2=lbp;t2<=ubp;t2++) { // parallelisable
    for (i=max(1,32*t1-32*t2);i<=min(n-1,32*t1-32*t2+31);
         i++) {
      for (j=max(1,32*t2);j<=min(n-1,32*t2+31);j++) {
        A[i][j] = A[i-1][j] + A[i][j-1];
} } } }
```

(c) The Pluto [3] transformed AST. Loop i and j are tiled by 32, and t_2 is parallelisable and can be unrolled if given constant lower/upper bounds.

Figure 2: Some representations of an example program [4] that can be polyhedral transformed by Pluto [3].

3 OVERVIEW

ϕ_{sm} aims to carry out polyhedral HLS at the right abstraction level for progressive lowering stages. Here, we first demonstrate what the abstraction levels that existing tools can provide for a concrete example, and why they are insufficient for the growing demands from polyhedral HLS applications. Next, we discuss what new abstraction layers that ϕ_{sm} can introduce to improve polyhedral HLS.

3.1 Limitations in Existing Tools

Using existing tools like Pluto [3] and CLooG [1], we can represent the input C code in Figure 2a, which has an annotated SCoP region, as a polyhedron (Figure 2b) and then optimise it into a C-like AST form as in Figure 2c. There is a big gap between this product and what an HLS tool normally expect, specifically:

(1) It is uncertain which loops should be placed on hardware.
(2) Optimisation directives, e.g., pipelining, unrolling, etc. should be inserted to achieve higher performance.
(3) Details about host interacts with the hardware, e.g., data transfer patterns, are opaque.

Unfortunately, abstraction layers from existing polyhedral tools hinder us from filling the gap. There are only two representations available, *polyhedron representation* and *C-like AST*. Polyhedron representations can encode iteration domain, schedule, and memory access as matrices [10], but they is too abstract to describe HLS optimisation. An HLS optimisation normally views its input as concrete loops and conditionals, which are unknown from the polyhedron representation unless we export it using an extra, irreversible code generator [1], e.g., a single statement in a stencil-based computation can be represented by a single polyhedron, but there can be hundreds of loops and conditionals generated from that to deal with various boundaries. C-like ASTs, on the other hand, are too close to what HLS tools take, any higher-level optimisation opportunities may already be missed at this low abstraction level since polyhedral information are not there anymore. Only non-polyhedral source-to-source transformation is possible.

3.2 Our Approach

ϕ_{sm} aims to fill the gap through progressive lowering in MLIR, specifically, using the Affine dialect [18]. Affine can perfectly address the aforementioned issues in existing tools: Affine describes polyhedron that polyhedral transformation can work on, and it has loops and conditionals for us to describe HLS optimisation. For example, Figure 3 shows the Affine code equivalent to the C-like AST produced by Pluto (Figure 2c), and since Affine restricts that its loops are bounded by affine combinations, this code piece also describes the transformed polyhedron.

```
#map0 = affine_map<()[s0] -> ((s0-1) floordiv 16 + 1)>
#map1 = affine_map<(d0)[s0] -> (0, (d0*32-s0+1) ceildiv 32)>
#map2 = affine_map<(d0)[s0] -> ((s0-1) floordiv 32 + 1, d0+1)>
#map3 = affine_map<(d0,d1) -> (1, d0*32-d1*32)>
#map4 = affine_map<(d0,d1)[s0] -> (s0, d0*32-d1*32+32)>
#map5 = affine_map<(d0) -> (1, d0*32)>
#map6 = affine_map<(d0)[s0] -> (s0, d0*32+32)>
affine.for %t1 = 0 to #map0()[%N] {
  affine.parallel_for %t2 = max #map1(%t1)[%N] to
                            min #map2(%t1)[%N] {
    affine.for %i = max #map3(%t1, %t2) to
                    min #map4(%t1, %t2)[%N] {
      affine.for %j = max #map5(%t2) to min #map6(%t2)[%N] {
        call @S1(%A, %i, %j)
} } } }
```

Figure 3: An Affine representation of the Pluto-transformed code in Figure 2c from [20]. Syntax details are in [18].

The *sub-bounding-box tiling* algorithm [25], which unifies the tile bounds for uniform workload distribution among processing units, is a perfect example showing the advantage of ϕ_{sm} leveraging Affine. Its original implementation needs to reproduce polyhedra from CLooG-generated code to find parallelogram hulls and regenerate C code in the end, while using Affine, we can calculate the hulls and transform the code directly in the same representation, which is more efficient, less error-prone, and easier to integrate with precedent and subsequent transformations.

After Affine, we can progressively lower the abstraction level to other dialects. During this procedure, we can describe software/hardware partition, design space exploration, data layout optimisation, and many other techniques as MLIR transformations. Once we reach Standard [19], the dialect at a level right above LLVM IR, we can decide whether export to Vitis [14], or continue lowering to hardware description dialects in CIRCT [6] (Figure 1), to finally produce an accelerator design.

4 SUMMARY

This paper presents the concepts of ϕ_{sm}, an polyhedral HLS tool built upon MLIR adapting progressive lowering. ϕ_{sm} can narrow the gap between polyhedral representation and HLS optimisation by lowering from the MLIR Affine dialect and transforming IRs at the right abstraction levels. More details and the current progress can be found in https://github.com/kumasento/polymer.

REFERENCES

[1] Cédric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *PACT*. IEEE, 7–16.

[2] Samuel Bayliss. 2014. PoTHoLeS: Polyhedral Compilation Tool for High Level Synthesis. https://github.com/SamuelBayliss/Potholes

[3] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*. 101–113.

[4] Uday Kumar Bondhugula. 2008. *Effective automatic parallelization and locality optimization using the polyhedral model*. Ph.D. Dissertation. The Ohio State University.

[5] Chun Chen. 2012. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 499–508.

[6] CIRCT. [n.d.]. "CIRCT" / Circuit IR Compilers and Tools. https://github.com/llvm/circt.

[7] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *ICCAD*. IEEE, 1–8.

[8] MLIR developers. [n.d.]. MLIR Language Reference. https://mlir.llvm.org/docs/LangRef/.

[9] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International journal of parallel programming* 21, 6 (1992), 389–420.

[10] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming* 34, 3 (2006), 261–317.

[11] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 4 (2015), 1–50.

[12] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2020. Domain-specific Multi-Level IR rewriting for GPU. *arXiv preprint arXiv:2005.13014* (2020).

[13] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 127–136.

[14] Vinod Kathail. 2020. Xilinx Vitis unified software platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 173–174.

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

[16] Junyi Liu, John Wickerson, Samuel Bayliss, and George A Constantinides. 2017. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 9 (2017), 1802–1815.

[17] Junyi Liu, John Wickerson, and George A Constantinides. 2016. Loop splitting for efficient pipelining in high-level synthesis. In *FCCM*. IEEE, 72–79.

[18] MLIR. [n.d.]. 'affine' Dialect. https://mlir.llvm.org/docs/Dialects/Affine/.

[19] MLIR. [n.d.]. 'std' Dialect. https://mlir.llvm.org/docs/Dialects/Standard/.

[20] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Affine C in MLIR. In *IMPACT*.

[21] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–27.

[22] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, 299–302.

[23] Sven Verdoolaege and Gerda Janssens. 2017. Scheduling for PPCG. *Report CW* 706 (2017).

[24] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *FPGA*.

[25] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving polyhedral code generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 1–10.

[26] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013. Improving high level synthesis optimization opportunity through polyhedral transformations. In *FPGA*. 9–18.