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 $\phi_{sm}$ (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 $\phi_{sm}$ (Figure 1), the first MLIR-based polyhedral HLS tool featuring progressively lowering by:

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, $\phi_{sm}$ 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. $\phi_{sm}$ is compatible with these prior approaches and is more versatile to cover recent HLS techniques and software/hardware co-optimisation with progressive lowering.
what an HLS tool normally expect, specifically: 𝜙
versible code generator [1], e.g., a single statement in a stencil-based
has loops and conditionals for us to describe HLS optimisation. For
polyhedron that polyhedral transformation can work on, and it
dress the aforementioned issues in existing tools: Affine describes
specifically, using the Affine dialect [18]. Affine can perfectly ad-
3.2 Our Approach
non-polyhedral source-to-source transformation is possible.
level since polyhedral information are not there anymore. Only
are too close to what HLS tools take, any higher-level optimisa-
to deal with various boundaries. C-like ASTs, on the other hand,
can be hundreds of loops and conditionals generated from that
to deal with various boundaries. C-like ASTs, on the other hand,
to convert to C-like ASTs, and this representation can be used
as a concrete example, and why they are insufficient for the growing demands
from polyhedral HLS applications. Next, we discuss what new ab-
straction layers that 𝜙
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 mem-
ory access as matrices [10], but they are 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, irre-
versible 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,
to deal with various boundaries. C-like ASTs, on the other hand,
to convert to C-like ASTs, and this representation can be used
as a concrete example, and why they are insufficient for the growing demands
from polyhedral HLS applications. Next, we discuss what new ab-
straction layers that 𝜙
introduce to improve polyhedral HLS.

3.2 Our Approach
𝜙
aims to fill the gap through progressive lowering in MLIR,
specifically, using the Affine dialect [18]. Affine can perfectly ad-
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

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

3 OVERVIEW
𝜙
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 ab-
straction layers that 𝜙
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 mem-
ory access as matrices [10], but they are 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, irre-
versible 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,
to deal with various boundaries. C-like ASTs, on the other hand,
to convert to C-like ASTs, and this representation can be used
as a concrete example, and why they are insufficient for the growing demands
from polyhedral HLS applications. Next, we discuss what new ab-
straction layers that 𝜙
introduce to improve polyhedral HLS.

3.2 Our Approach
𝜙
aims to fill the gap through progressive lowering in MLIR,
specifically, using the Affine dialect [18]. Affine can perfectly ad-
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

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

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.
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
leveraging
Affine. Its original implementation needs to reproduce polyhedral
from CLooG-generated code to find parallelogram hulls and regen-
erate 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 soft-
ware/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
, an polyhedral HLS tool
built upon MLIR adapting progressive lowering. 𝜙
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


