MLIR Loop Optimizations for High-Level Synthesis: a Case Study

Serena Curzel*
Politecnico di Milano
Milano, Italy
serena.curzel@polimi.it

Sofija Jovic
Politecnico di Milano
Milano, Italy
sofija.jvc@gmail.com

Michele Fiorito
Politecnico di Milano
Milano, Italy
michele.fiorito@polimi.it

Antonino Tumeo
PNNL
Richland, Washington, USA
antonino.tumeo@pnnl.gov

Fabrizio Ferrandi
Politecnico di Milano
Milano, Italy
fabrizio.ferrandi@polimi.it

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; **Emerging languages and compilers**.

KEYWORDS

FPGA, High-Level Synthesis, MLIR

ACM Reference Format:
Serena Curzel, Sofija Jovic, Michele Fiorito, Antonino Tumeo, and Fabrizio Ferrandi. 2022. MLIR Loop Optimizations for High-Level Synthesis: a Case Study. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3559009.3569688

INTRODUCTION

High-Level Synthesis (HLS) tools automatically translate code from a general-purpose programming language (typically C or C++) into a hardware description language (HDL) such as Verilog or VHDL, significantly reducing the hardware design productivity gap. HLS benefits from the same compiler optimizations that identify instruction, memory, and data parallelism for general-purpose processors. However, they also need to consider specific needs of low-level circuit design, such as the notion of time, synchronous and asynchronous logic, and wiring delays. Because of the mismatch between hardware abstractions and general-purpose programming languages, HLS tools often require the addition of *pragma* directives in the input code to guide hardware generation.

In this work we propose to apply high-level optimizations before HLS, leveraging dedicated abstractions and without relying on tool-specific annotations. As a case study, we implement a high-level loop pipelining pass exploiting the Multi-Level Intermediate Representation (MLIR) framework [3], a recent contribution to the LLVM project that enables and encourages the implementation of reusable compiler infrastructures. Loop pipelining overlaps iterations with the aim of parallelizing as many operations as possible; the ideal target is obtaining a loop where a new iteration can start executing every clock cycle. With our approach (inspired by classic software pipelining [2]), there is no need to interfere with the low-level hardware generation steps, and we can take advantage of loop-specific constructs provided by MLIR dialects. Moreover, our design flow generates portable pre-optimized code that is not restricted to a specific HLS tool. The experimental evaluation confirms that our loop pipelining implementation improves the performance of the generated accelerators.

PROPOSED APPROACH

We leverage high-level code optimizations to provide a transformed input description to HLS, without binding it to the requirements of a specific HLS tool (most notably pragma annotations). The proposed solution is an alternative to delegating transformations to the HLS tool itself: for example, Vitis HLS lets users trigger optimizations in the backend through pragmas in the input C code. Applying transformations on a specialized, higher-level abstraction increases flexibility, and portability, and requires less time than implementing and exploring different techniques within the HLS tool (when this is possible, as most HLS tools are closed-source). Moreover, MLIR is built to allow integration and reuse between different optimizations: this means that loop pipelining may be combined with other techniques to generate more efficient hardware accelerators.

We implemented two custom MLIR passes to pipeline affine loops: the first one extracts a data flow graph from the MLIR loop body, and the second one generates the pipelined loop code according to the schedule produced by an external scheduler, HatSchet. Existing affine constructs significantly simplify the implementation, confirming that the MLIR dialect-based approach provides a convenient framework for the introduction of new optimizations. For example, loop pipelining requires the loop to pass results from

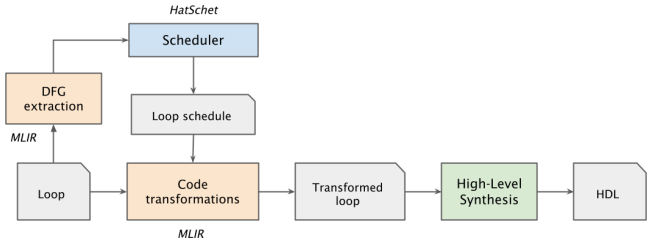

Figure 1: Overview of the proposed optimization flow.

*Also with PNNL.

PACT '22, October 10–12, 2022, Chicago, IL, USA

ACM ISBN 978-1-4503-9868-8/22/10.
https://doi.org/10.1145/3559009.3569688