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

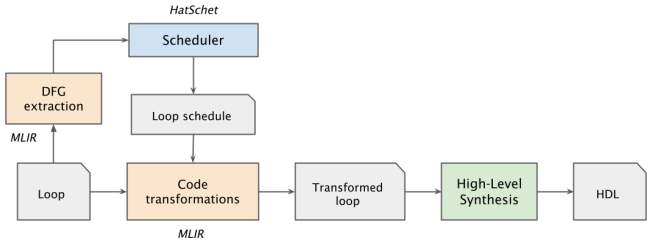
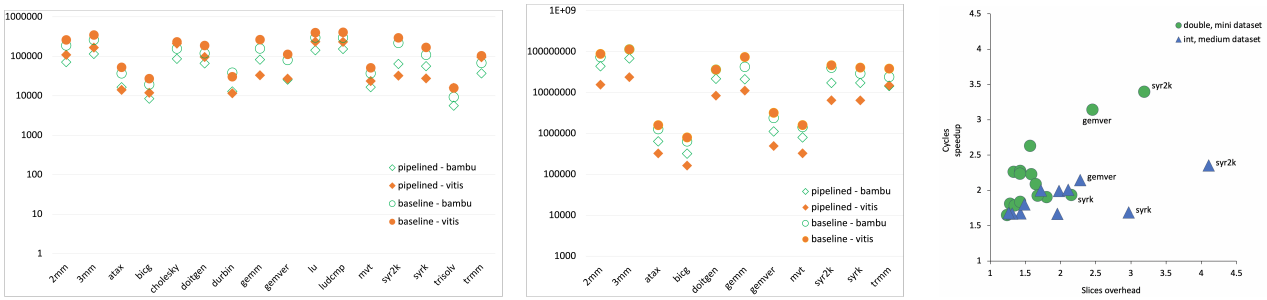


Figure 1: Overview of the proposed optimization flow.

*Also with PNNL.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
PACT '22, October 10–12, 2022, Chicago, IL, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9868-8/22/10.
https://doi.org/10.1145/3559009.3569688



(a) Latency on double precision floating-point, mini dataset. **(b) Latency on integer, medium dataset.** **(c) Resource consumption overhead and performance improvement.**

Figure 2: PolyBench kernels synthesized with Bambu (pipelined at the MLIR level) and Vitis HLS (pipelined in the backend).

Table 1: Gemm accelerators generated from LLVM IR.

Version	Tool	Cycles	Slices	Speedup
double, mini	Bambu	82 362	1303	1.91x
double, mini	Vitis HLS	206 821	1119	1.29x
int, medium	Bambu	21 160 402	506	2.01x
int, medium	Vitis HLS	31 764 201	322	2.34x

Table 2: Affine optimizations on gemm (double, mini).

Optimizations	Cycles	Slices	Speedup
none	157 122	724	baseline
loop pipelining	82 362	1303	1.91x
loop permutation + pipelining	81 182	1306	1.93x
loop unrolling + pipelining	17 642	8075	8.91x

one iteration to the next one, and this is usually solved in hardware by using dedicated registers: in MLIR, we can use affine yield operations and iteration arguments.

If one of the loop bounds is a variable, we introduce a check at runtime to assess whether there are enough iterations to execute the new loop safely (at least the iterations started in the prologue, plus one), falling back on the original loop if this is not the case. This causes additional area overhead in the generated accelerator but no degradation in performance.

EXPERIMENTAL RESULTS

We perform a set of experiments on the PolyBench benchmark suite to validate the effectiveness and portability of our approach using Vitis HLS and Bambu [1]. Figures 2a-b show that loop pipelining provides a significant reduction in clock cycles, as expected, both when it is applied within the HLS tool and when it is implemented as a high-level MLIR optimization.

Pipelining loops increases resource consumption; however, in most cases the area overhead is adequately compensated by the reduction in the number of clock cycles. Figure 2c visualizes this trend by plotting the performance increase with respect to the overhead in slices utilization in the experiments with Bambu and MLIR-based loop pipelining (labeled points are outliers due to multiple loops in the code or variable loop bounds).

We also synthesized the generated LLVM IRs through Vitis HLS, setting up a compilation flow that bypasses the standard frontend to feed LLVM IR directly to the closed-source backend (Table 1). The introduction of loop pipelining as an MLIR high-level optimization positively affects accelerator performance also through the Vitis HLS backend. The compilation flow is more experimental than the one through Bambu, as Vitis HLS is optimized primarily for annotated C/C++ code; nevertheless, these results provide motivation to further explore synthesis-oriented transformations in MLIR that can benefit multiple HLS backends.

Finally, the introduction of loop pipelining as a high-level affine pass allows to combine it with other affine passes in MLIR. Table 2 shows that, for example, the Bambu backend benefits from an increase in the number of iterations in the pipelined loop, which can be obtained through loop permutation: this reduces the number of cycles with a minimal increase in resource utilization. Increasing the size of the loop body through unrolling, instead, results in an even faster design at the cost of significant area consumption.

These experiments open the way to further explore the introduction of new optimization techniques that can benefit HLS when they are applied at a higher level of abstraction than existing solutions. Code for our implementation is available at https://gitlab.pnnl.gov/sodalite/soda-opt/-/tree/experimental/loop_pipelining.

ACKNOWLEDGMENTS

This research was partially supported by the PNNL Data-Model Convergence Laboratory-Directed R&D Initiative, the DARPA Real-Time Machine Learning (RTML) program, the P38 DoD/DOE collaboration, and the H2020 EVEREST (No 957269) and HERMES (No 101004203) projects.

REFERENCES

[1] F. Ferrandi et al. 2021. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *58th ACM/IEEE Design Automation Conference (DAC)*.
[2] M. S. Lam. 1989. Software pipelining. In *A Systolic Array Optimizing Compiler*. Springer, 83–124.
[3] C. Lattner et al. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14.