Pre-Scheduling of Affine Loops for HLS Pipelining *

Serena Curzel[1 [0000-0002-8202-1627]], Sofija Jovic[1 [0000-0003-2061-0802]], Michele Fiorito[1 [0000-0001-8675-6703]], Antonino Tumeo[2 [0000-0001-9452-120X]], and Fabrizio Ferrandi[1 [0000-0003-0301-4419]]

[1] Politecnico di Milano
[2] Pacific Northwest National Laboratory

Abstract. Loop transformations are essential to improve the quality of results of accelerators generated through High-Level Synthesis (HLS). Loop pipelining is usually performed on a low-level intermediate representation (IR) of the code, which includes the notion of time and has access to information about available resources. In this paper, we introduce loop pipelining as a pre-optimization outside the HLS tool, applying scheduling and code generation to transform affine loops in a high-level frontend based on MLIR. Working on such an abstract level simplifies the analysis of dependencies and the implementation of code generation steps, it does not require access to low-level architectural details, and nevertheless, it can achieve comparable accelerator performance to state-of-practice HLS loop pipelining. The proposed approach does not depend on a specific HLS backend, and it can be easily integrated with existing and future high-level optimizations.

Keywords: FPGA · High-Level Synthesis · MLIR · loop pipelining.

1 Introduction

High-Level Synthesis (HLS) tools have become a critical part of the hardware design process, as they allow the automatic translation of general-purpose software specifications, primarily written in C/C++, into an HDL description ready for logic synthesis and implementation. Thanks to HLS, developers can describe the kernels they want to accelerate at a high level of abstraction and obtain efficient designs for Field Programmable Gate Arrays (FPGAs) or as application-specific integrated circuits (ASICs) without being experts in low-level circuit design. Because of the mismatch between the requirements of hardware abstractions and the characteristics of general-purpose programming languages used to write input specifications, HLS tools often require users to augment their input code through *pragma* annotations (i.e., compiler directives) that guide the synthesis

* This research was partially supported by the Spoke 1 - *FutureHPC & BigData* of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Mission 4 - Next Generation EU.

process, for example, towards a specific performance-area trade-off. Pragmas can trigger loop optimizations, which are essential to improve the quality of results (QoR) of accelerators generated through HLS. Loop pipelining aims at overlapping the execution of different iterations by issuing a new iteration before the previous one has finished executing; this requires a transformation including an analysis of dependencies and a scheduling process. The ideal target is obtaining a loop with an Initiation Interval (II) of one, meaning that a new iteration can start executing every clock cycle, which may be possible if there are enough available computational resources and if dependencies between operations are respected.

In this paper, we present an implementation of loop pipelining for HLS that exploits the Multi-Level Intermediate Representation (MLIR) framework [6]. MLIR is a recent contribution to the LLVM project that enables the implementation of reusable compiler infrastructures; its key feature is providing mechanisms to define new abstraction levels ("dialects") that solve compiler transformation and optimization problems through specialized representations. Lowering passes provide methods to move between dialects; the last step in the lowering process is the LLVM dialect, which can be directly translated into an LLVM IR. MLIR was conceived initially to be applied within machine learning (ML) frameworks and to build compilers for design-specific languages (DSL), and several ML and high-level software frameworks provide an interface to MLIR dialects. Previous works that applied MLIR to HLS and hardware design either heavily relied on the optimization capabilities of a specific HLS tool in the backend [17], or implemented a new HLS tool from scratch [16]. Our approach is different, and, in a way, it tries to combine the best of both worlds: we intend to apply meaningful transformations exploiting specialized MLIR representations and benefit at the same time from mature HLS tools backed by decades of research. Implementing loop pipelining (and other optimizations) as a high-level pre-processing step does not require in-depth knowledge about the allocation, scheduling, and binding steps within the HLS tool, simplifying the introduction and exploration of new techniques. Finally, the proposed approach does not involve pragma annotations or code patterns inherently tied to a single backend HLS tool, resulting in portable pre-optimized code.

In summary, this paper presents the following contributions:

- we show how a dedicated high-level abstraction facilitates scheduling and code generation for loop pipelining;
- we apply MLIR-based loop pipelining to HLS, obtaining comparable results to standard approaches despite abstracting most of the architectural details;
- we demonstrate performance portability across different HLS backends and the benefits of coupling loop pipelining with other high-level optimizations.

2 Proposed Approach

Loop pipelining aims at overlapping the execution of multiple iterations, and it is a suitable transformation when there are not enough hardware resources

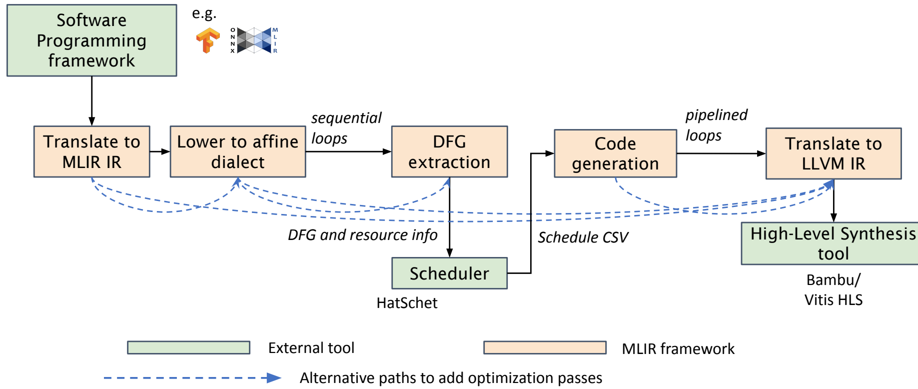


Fig. 1. Overview of the proposed design flow.

to accommodate an unrolled loop. The technique has been successfully used in compiler infrastructures for decades [5], and it generally consists of two steps: loop scheduling and code generation. Scheduling uses information about dependencies and available hardware resources to find a new instruction execution order; code generation creates a pipelined loop according to the result of the scheduling phase (completing partial iterations before and after the new loop with a *prologue* and an *epilogue*). Depending on the available computation and memory resources, their latency, and if inter-iteration dependencies allow it, a pipelined loop can issue the execution of a new iteration at every clock cycle.

Our approach follows the steps illustrated in Figure 1. The input code may originate from any high-level programming framework with a translation into MLIR (e.g., TensorFlow, ONNX-MLIR, or C through Polygeist [10]). After lowering it to the affine dialect, the code contains one or more sequential `for` loops, which are analyzed by a pass we implemented to extract data flow graphs (DFGs) representing the operations in each loop body as nodes and their dependencies as edges. Each DFG is passed to a scheduler to obtain a loop iteration schedule; resource constraints may be set iteratively to achieve a specific trade-off between performance and area consumption. Achieving an optimal or close to optimal schedule (i.e., a list of operations assigned to a clock cycle and to a resource) is an NP-complete problem; to solve it we use the HatSchet [14] scheduling library, an open-source scheduling tool for HLS that offers various algorithms and heuristics for the construction of pipelined schedules. We then implemented a code generation pass which rewrites loops in the input code using each schedule generated by HatSchet to produce a pipelined loop. Additional optimization passes can be introduced along the way, before or after pipelining the loop; finally, the MLIR IR is translated into an LLVM IR and passed to the HLS tool to generate an accelerator description in Verilog/VHDL.

The HLS tool processes the LLVM IR without any knowledge of the high-level transformations it went through; it considers the dependencies between

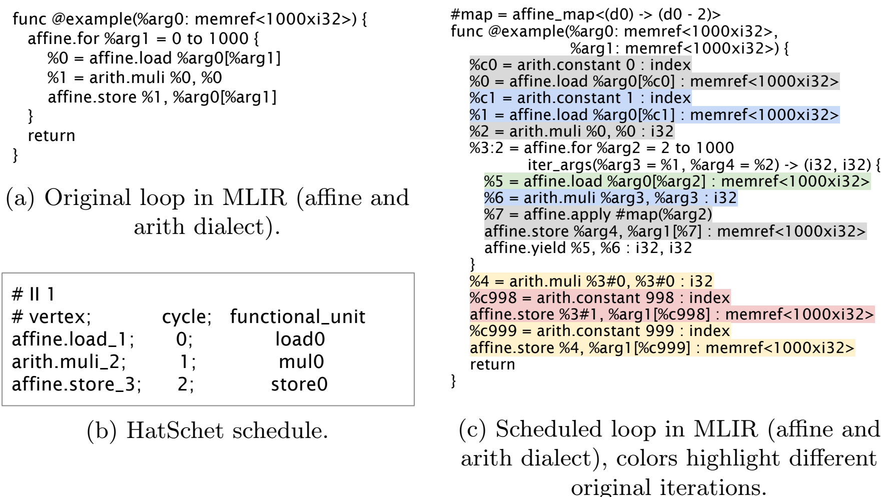
operations in the new loop body and, because they have been constructed to be independent, it is free to schedule them in parallel if enough functional units are available. The II corresponds to the latency of one iteration in the new loop, as each of them starts a new iteration of the original loop and advances the execution of the ones that were previously started.

The proposed solution represents an alternative to delegating the scheduling and code generation steps to a later stage within the HLS tool itself. Implementing a transformation such as loop pipelining outside the HLS tool has several advantages: it increases modularity, as it can be easily enabled, disabled, or combined with other compilation passes, it increases portability across HLS backends, and it requires less time than implementing the same transformation within the HLS tool (when this is possible, as most HLS tools are closed-source). A fundamental difference is the level of abstraction of the IR that needs to be modified: a `for` loop described in MLIR is a concise representation that encodes all necessary information in a few lines of code; the same loop, when lowered to an LLVM IR in the frontend of an HLS tool, becomes a list of basic blocks with a much larger number of instructions in static single-assignment (SSA) form. The number of instructions has a significant impact on the scheduling complexity; moreover, the pre-scheduling process in MLIR only has to reorder instructions in the loop body so that they can all be executed in parallel in the new loop, while the HLS tool has to assign each instruction to a precise clock cycle in order to build the FSM controller. Because of this, the pre-scheduling process does not need as much information about the availability and latency of functional units; a simplified resource model is enough to obtain a correct schedule.

3 Implementation

The core of the proposed implementation is composed of the MLIR passes that extract DFGs and generate code for the pipelined loops, while scheduling is performed through the external HatSchet tool. Additional passes were implemented to handle non-trivial cases (i.e., loops that need to propagate values across multiple iterations, loops with variable bounds, and loops containing if-else blocks).

DFG Extraction - Starting from an affine loop such as the one in Figure 2a, the DFG extraction pass extracts precedence and data dependencies between operations and encodes them in the HatSchet input format. A precedence dependency refers to an operation using the result of another operation in the same loop iteration; data dependencies exist between two memory operations accessing the same memory location, and they have an attribute to express the distance between the loop iterations that contain the two operations. Information about available resources is also encoded in this phase in a separate file. Extracting precedence dependencies from MLIR code requires visiting all operations in the loop; data dependency analysis is less trivial, but it can be solved through an existing MLIR affine method that analyzes a pair of memory operations and decides if a dependency exists (the distance can also be deduced from



(a) Original loop in MLIR (affine and arith dialect).

(b) HatSchet schedule.

(c) Scheduled loop in MLIR (affine and arith dialect), colors highlight different original iterations.

Fig. 2. Code generation for high-level loop pipelining in MLIR.

its output). This is a clear example of how existing MLIR constructs simplify the implementation of new optimizations through specialized levels of abstraction.

Code Generation - The code generation pass loads the HatSchet schedule from its textual format (Figure 2b) into a suitable data structure and uses it to generate code for the pipelined loop. HatSchet provides only the new loop iteration schedule, so the MLIR pass needs to generate also the code for prologue and epilogue. The final result is the loop in Figure 2c, where the first two iterations of the original loop are started in the prologue, the new loop contains independent operations that can be executed in parallel, and the epilogue completes the two original loop iterations started in the last iteration of the new loop.

Results Forwarding - When loop pipelining is implemented during the low-level hardware generation process, dedicated registers can be instantiated to pass results from one loop iteration to the next one. If the lifetime of variables spans across multiple iterations, architectural support is available to implement rotating register files. In our implementation, instead, forwarding of results is solved by adding MLIR *iteration arguments* and affine *yield* operations: at the end of each iteration, operands are yielded and they become available as arguments for the next iteration. If a value has to cross multiple iterations, we introduce additional arguments to shift values at the end of each iteration until they are used.

Conditional Pipelining - The generated loop produces correct results if all iterations started in the prologue, and at least one iteration of the new loop, are always executed (with the epilogue taking care of finalizing incomplete iterations). If one of the loop bounds is a variable, it is not possible to assess at compile time whether enough iterations will be executed to cover the prologue and new loop iterations. To allow loop pipelining in such a situation, we introduce a check at runtime to assess whether there are enough new loop iterations

to safely execute the pipelined loop, falling back on the original loop if this is not the case. This is a simple and effective solution that can be easily implemented in MLIR with affine if operations and affine sets; versioning the loop in this way results in having both the original and the pipelined loop in the code, causing additional area consumption in the generated accelerator but no degradation in performance.

If-conversion - An if-conversion pass was implemented following [15], which allows pipelining loops containing `if` and `else` blocks through speculative execution. The pass is run as a preprocessing step before DFG extraction. In the generated code `if` and `else` constructs are removed, and all operations are extracted out of them; the `if` condition is used in a `select` operation that decides which result to keep. This transformation was designed for software, but with our MLIR-based approach it can be seamlessly applied in isolation or together with loop pipelining.

4 Experimental Results

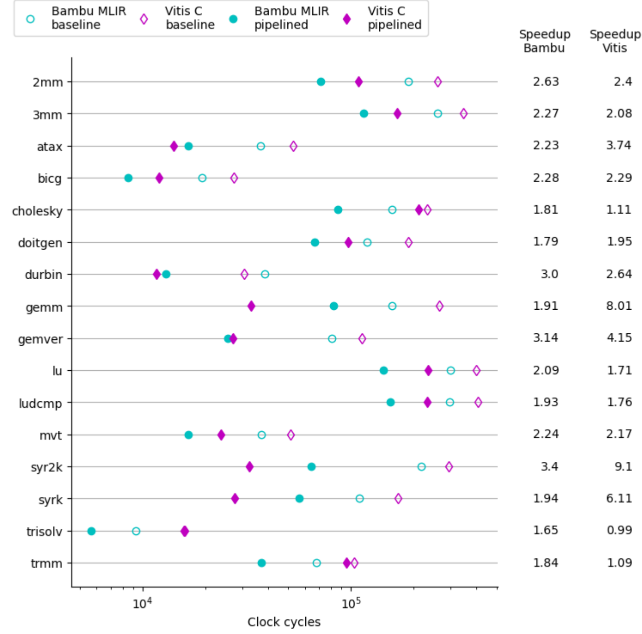
In this section, we present experiments that validate the effectiveness of our approach. We chose to focus our experiments on kernels from the PolyBench benchmark suite[3], in two versions: double-precision floating-point operations on the 'mini' dataset, and integer (32 bit) operations on the 'medium' dataset. (Kernels in the 'solvers' category are only available in floating-point.) Simulation times and resource consumption of floating-point units are considerably higher than their integer counterparts, so we selected a smaller dataset (i.e., smaller loop bounds) for the double precision experiments.

We use the MLIR version of PolyBench kernels provided by Polygeist [10], and the standard C version for comparison. Concerning scheduling options, different HatSchet configurations were tested to conclude that, on PolyBench, the ILP-based modulo scheduling algorithm [11] produces the best result in an acceptable amount of time. Similarly, we verified that a model with infinite resources allows reaching the highest performance, as PolyBench loop bodies contain few instructions (in the range of 5-10 each) that never risk depleting the available FPGA resources; the target is a Xilinx Zynq-7000 FPGA at 100 MHz frequency, and we assume that inputs and outputs are stored in on-chip BRAMs. All accelerator performance and resource utilization results are reported post simulation and post place-and-route, respectively.

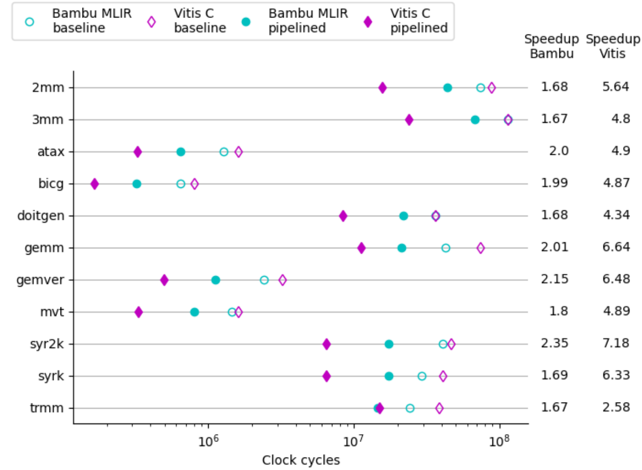
4.1 Effectiveness of Loop Pipelining

We show here the performance increase achieved by our implementation of loop pipelining, applying it to the open-source HLS tool Bambu [4] which does not support loop pipelining. We translate the PolyBench MLIR kernels into LLVM IR and synthesize them with Bambu, first without any optimization and then

[3] https://web.cs.ucla.edu/~pouchet/software/polybench/



(a) Double precision floating-point, mini dataset.



(b) Integer, medium dataset.

Fig. 3. Execution times in clock cycles of PolyBench kernels synthesized with Bambu (loops pipelined at the MLIR level) and Vitis HLS (C code, pipelined in the HLS backend).

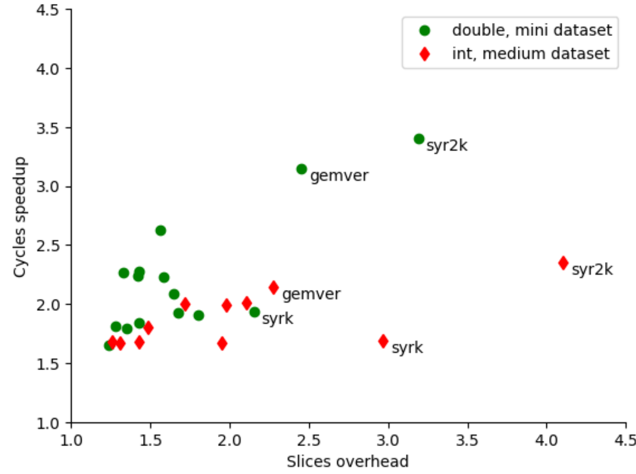


Fig. 4. Resource consumption overheads and performance improvements introduced in Bambu by loop pipelining. Labeled points present the most relevant area overheads.

applying our affine loop pipelining passes before the translation. To compare our approach with low-level loop pipelining applied during the HLS process, we also synthesize the standard C version of PolyBench kernels with Vitis HLS 2021.1, adding specific directives that control loop optimizations to obtain baseline and pipelined accelerators.

Figure 3 shows the execution time of the generated accelerators: loop pipelining provides a significant reduction in clock cycles, as expected, and this is verified both when pipelining is applied within the HLS tool and when it is implemented as a high-level MLIR optimization. In the 'double' experiments, Bambu usually performs better than Vitis HLS, because the floating-point functional units in Bambu take fewer clock cycles to execute and neither tool pipelines the functional units execution. For the integer experiments, loop pipelining reduces the number of clock cycles with both backends, but Vitis HLS produces better results because it pipelines the execution of load operations during scheduling.

Loop pipelining increases FPGA resources consumption because it requires extra states in the controller for prologue/epilogue and larger multiplexers in front of each functional unit; however, if we compare the achieved speedup with the area overhead, in most cases the price to pay in terms of area is adequately compensated by the reduction in the number of clock cycles. Figure 4 visualizes this trend by plotting the ratio between performance increase and the overhead in slices utilization for the experiments with Bambu and MLIR-based loop pipelining; while most benchmarks are clustered in the bottom left of the plot, some kernels show a disproportionate area overhead. The innermost loops in syrk and syr2k have an upper bound depending on the induction variable of the outermost loop, which requires introducing conditional pipelining. Gemver,

instead, is the only kernel that contains four independent loops to be pipelined, and so it incurs four times the area overhead for the introduction of prologue and epilogue.

Kernel	Version	Tool	Cycles	DSPs	LUTs	Slices	Registers	Frequency	Speedup	Slices overhead
gemm	double, mini	Bambu	157 122	10	1678	723	1397	101.82MHz	baseline	baseline
gemm	double, mini	Bambu	82 362	20	3024	1303	3576	101.48 MHz	1.91x	1.80x
gemm	double, mini	Vitis HLS	266 800	14	981	531	1409	120.38 MHz	baseline	baseline
gemm	double, mini	Vitis HLS	206 821	28	2504	1119	3580	118.37 MHz	1.29x	2.54x
gemm	int, medium	Bambu	42 512 202	9	585	248	429	134.64 MHz	baseline	baseline
gemm	int, medium	Bambu	21 160 402	27	1355	506	949	105.93 MHz	2.01x	2.21x
gemm	int, medium	Vitis HLS	74 328 230	9	199	95	348	175.69 MHz	baseline	baseline
gemm	int, medium	Vitis HLS	31 764 201	21	649	322	1001	173.25 MHz	2.34x	3.39x
syr2k	double, mini	Bambu	218 387	20	2240	906	2363	98.64 MHz	baseline	baseline
syr2k	double, mini	Bambu	64 249	40	6841	2826	6463	97.58 MHz	3.40x	3.19x
syr2k	double, mini	Vitis HLS	294 171	25	1211	640	1900	116.90 MHz	baseline	baseline
syr2k	double, mini	Vitis HLS	137 463	50	4095	1736	5399	104.54 MHz	2.14x	2.71x
syr2k	int, medium	Bambu	40 700 162	12	595	255	553	144.97 MHz	baseline	baseline
syr2k	int, medium	Bambu	17 285 566	78	2579	1030	1988	104.52 MHz	2.35x	4.10x
syr2k	int, medium	Vitis HLS	46 479 441	12	253	137	427	175.68 MHz	baseline	baseline
syr2k	int, medium	Vitis HLS	28 691 013	70	1491	794	2109	119.23 MHz	1.62x	5.79x

Table 1. Accelerators generated through different HLS backends from the same LLVM IR (baseline or pipelined).

Optimizations	Cycles	DSPs	LUTs	Slices	Registers	Frequency	Speedup	Slices overhead
none	157 122	10	1678	724	1397	102.27 MHz	baseline	baseline
loop pipelining	82 362	20	3024	1303	3576	101.48 MHz	1.91x	1.80x
loop permutation + pipelining	81 182	20	3006	1306	3413	100.94 MHz	1.93x	1.80x
loop unrolling + pipelining	17 642	100	21 380	8075	18 671	90.39 MHz	8.91x	11.15x

Table 2. Effect of affine optimizations on gemm (double, mini) synthesized by Bambu.

4.2 Portability

An advantage of implementing loop pipelining as an MLIR transformation is that it does not require to introduce annotations for a specific backend HLS tool: after applying our passes, the code contains a new loop in the MLIR affine dialect, and after lowering and translation it only contains standard LLVM code. We can thus synthesize the generated LLVM IRs also with Vitis HLS, setting up a compilation flow that bypasses the C/C++ frontend to feed LLVM IRs directly to the backend. Table 1 reports results for kernels pipelined through our implementation and then synthesized by Bambu and by Vitis HLS (only two kernels are shown for the sake of conciseness, one with constant and one with variable loop bounds). Speedups and overheads are calculated with respect to a

baseline LLVM IR translated from MLIR without loop pipelining, and they prove that the introduction of loop pipelining as an MLIR high-level optimization has a positive effect on accelerator performance also through the Vitis HLS backend.

4.3 Design Space Exploration

The modularity and flexibility provided by MLIR allow us to introduce new optimizations and to experiment with existing ones to generate optimized IRs for HLS. The affine dialect provides a growing set of loop-oriented transformations; even if some of them are also available as backend HLS optimizations triggered by pragmas, applying them at the MLIR level allows decoupling loop optimizations (which do not necessarily require hardware-related considerations) from the backend HLS tool, and thus enhance portability. Loop pipelining can provide performance benefits on its own, but it can also be coupled with different optimizations to explore different design points with different performance/area trade-offs. We explored a few different options on the gemm kernel with the Bambu backend: Table 2 shows that it can be beneficial to increase the number of iterations in the pipelined loop through loop permutation, which reduces the number of cycles with a minimal increase in resource utilization. If we increase the size of the loop body through unrolling, instead, we obtain an even faster design at the cost of significant area consumption. The same exploration of design points would require manual modifications on the code when done at the C/C++ level; for typical HLS optimizations such as loop unrolling, this can be as simple as adding a pragma, but it can require significant code rewriting for other transformations (including loop permutation). In an MLIR-based design flow, instead, optimizations can be exposed as compiler passes and compiler options that are easier to enable/disable in a design space exploration phase.

5 Related Work

Loop transformations in general, and loop pipelining in particular, are key optimizations to improve HLS quality of results, and thus they have been explored from several different perspectives. A non-exhaustive list of related work includes approaches that leverage the polyhedral model [20, 13, 19, 18]; most of these works run C/C++ inputs through polyhedral optimizers and write back restructured C/C++ annotated with HLS directives, so even if they improve the performance of generated accelerators, they are hard to combine with other optimizations, and the code they generate is not portable across HLS tools. POLSCA [18] proposed to exploit MLIR and the Vitis HLS LLVM frontend to improve the interaction between polyhedral tools and HLS. Our approach leverages a high-level abstraction designed for polyhedral optimizations (the MLIR affine dialect) to implement loop pipelining, and it works independently of the backend HLS tool.

Multiple previous efforts aimed at improving the loop pipelining technique itself, both for software and for HLS. For example, [3] tackles nested loop optimizations and proposes to merge epilogue and prologue of adjacent iterations;

works HatSchet itself [14] focus on improving scheduling time and quality. Irregular loops with variable bounds and complex dependencies are especially challenging and require dedicated solutions: for example, [2] applies speculative loop pipelining for HLS, [7] supports loops with non-constant dependencies, [9] exploits polyhedral frameworks to implement dynamic loop pipelining.

Finally, it is worth mentioning that several other tools have been proposed to couple the use of MLIR and HLS tools [1, 17, 12, 16, 8]. Thanks to the modular nature of MLIR, our implementation of high-level loop pipelining can be integrated into any of them, regardless of the capabilities of the chosen HLS tool.

6 Conclusion

The dedicated abstraction provided by the MLIR affine dialect facilitates dependency analysis and code generation for loop pipelining, and when MLIR high-level transformations are applied to HLS they can improve quality of results of the generated accelerators even if architectural details are not visible at the MLIR level. Implementing loop optimizations as compiler transformations in a preliminary step before HLS, as opposed to implementing them inside the tool, also increases developer productivity and decouples the optimizations from a specific backend tool. To support these claims, we implemented a set of compiler passes supporting affine loop pipelining and applied them to different HLS backends. We obtained similar or better results than state-of-practice solutions for HLS of C code; we then demonstrated performance portability by synthesizing the same optimized code with both Bambu and Vitis HLS. Finally, we showed how custom and existing MLIR passes can work together, providing a convenient way of exploring different design choices without manually modifying the input code. Our approach opens the way to further research into high-level optimization techniques for HLS, which can also be successfully integrated into modular MLIR-based hardware design flows.

References

1. Bohm Agostini, N., Curzel, S., Zhang, J.J., Limaye, A., Tan, C., Amatya, V., et al.: Bridging Python to Silicon: The SODA Toolchain. IEEE Micro **42**(5), 78–88 (2022)
2. Derrien, S., Marty, T., Rokicki, S., Yuki, T.: Toward Speculative Loop Pipelining for High-Level Synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(11), 4229–4239 (2020)
3. Fellahi, M., Cohen, A.: Software pipelining in nested loops with prolog-epilog merging. In: International Conference on High-Performance Embedded Architectures and Compilers. pp. 80–94. Springer (2009)
4. Ferrandi, F., Castellana, V.G., Curzel, S., Fezzardi, P., Fiorito, M., Lattuada, M., et al.: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In: DAC 2021: 58th ACM/IEEE Design Automation Conference (2021)
5. Lam, M.S.: Software pipelining. In: A Systolic Array Optimizing Compiler, pp. 83–124. Springer (1989)

6. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., et al.: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In: CGO 2021: IEEE/ACM International Symposium on Code Generation and Optimization. pp. 2–14. IEEE (2021)
7. Li, P., Pouchet, L.N., Cong, J.: Throughput optimization for high-level synthesis using resource constraints. In: IMPACT '14: International Workshop on Polyhedral Compilation Techniques (2014)
8. Liang, G.M., Yuan, C.Y., Yuan, M.S., Chen, T.L., Chen, K.H., Lee, J.K.: The Support of MLIR HLS Adaptor for LLVM IR. In: Workshop Proceedings of the 51st International Conference on Parallel Processing (2023)
9. Liu, J., Wickerson, J., Bayliss, S., Constantinides, G.A.: Polyhedral-based dynamic loop pipelining for high-level synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **37**(9), 1802–1815 (2017)
10. Moses, W.S., Chelini, L., Zhao, R., Zinenko, O.: Polygeist: Raising c to polyhedral mlir. In: PACT 2021: 30th International Conference on Parallel Architectures and Compilation Techniques. pp. 45–59 (2021)
11. Oppermann, J., Koch, A., Reuter-Oppermann, M., Sinnen, O.: ILP-based Modulo Scheduling for High-Level Synthesis. In: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES). pp. 1–10 (2016)
12. Pilato, C., Böhm, S., Brocheton, F., Castrillón, J., Cevasco, R., Cima, V., et al.: EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms. In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021. pp. 1320–1325. IEEE (2021)
13. Pouchet, L.N., Zhang, P., Sadayappan, P., Cong, J.: Polyhedral-Based Data Reuse Optimization for Configurable Computing. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA). pp. 29–38 (2013)
14. Sittel, P., Oppermann, J., Kumm, M., Koch, A., Zipf, P.: HatScheT: A Contribution to Agile HLS. In: FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers. pp. 1–8. VDE (2018)
15. Stoutchinin, A., Gao, G.: If-conversion in ssa form. In: European Conference on Parallel Processing. pp. 336–345. Springer (2004)
16. Urbach, M., Petersen, M.B.: HLS from PyTorch to System Verilog with MLIR and CIRCT (2022), 2nd Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)
17. Ye, H., Hao, C., Cheng, J., Jeong, H., Huang, J., Neuendorffer, S., Chen, D.: ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In: 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). pp. 741–755. IEEE (2022)
18. Zhao, R., Cheng, J., Luk, W., Constantinides, G.A.: POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations. In: International Conference on Field-Programmable Logic and Applications (FPL) (2022)
19. Zuo, W., Li, P., Chen, D., Pouchet, L.N., Zhong, S., Cong, J.: Improving polyhedral code generation for high-level synthesis. In: CODES+ISSS 2013: International Conference on Hardware/Software Codesign and System Synthesis. pp. 1–10 (2013)
20. Zuo, W., Liang, Y., Li, P., Rupnow, K., Chen, D., Cong, J.: Improving high level synthesis optimization opportunity through polyhedral transformations. In: FPGA '13: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. p. 9–18 (2013)