Higher-Level Synthesis: experimenting with MLIR polyhedral representations for accelerator design

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
Pacific Northwest National Laboratory
Richland, Washington, USA
antonino.tumeo@pnnl.gov

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

Abstract

High-Level Synthesis (HLS) tools simplify the design of hardware accelerators by automatically generating Verilog/VHDL code starting from a general purpose software programming language. They include a wide range of optimization techniques in the process, most of them performed on a low-level intermediate representation of the code. We believe that introducing optimizations on a higher level of abstraction could significantly contribute to the automated design process results; in particular, polyhedral techniques for the manipulation of loops could have a significant impact on the generated accelerators. This paper focuses on loop pipelining, which we use as a case study to explore the introduction of compiler-based transformations on top of an existing HLS process. We leverage the Multi-Level Intermediate Representation (MLIR) and evaluate the impact of the proposed transformations on an open-source HLS tool that would not otherwise support loop pipelining. Preliminary results confirm that our implementation increased the performance of the generated accelerators, without any modification to the underlying HLS tool.

ACM Reference Format:
Serena Curzel, Sofija Jovic, Michele Fiorito, Antonino Tumeo, and Fabrizio Ferrandi. 2022. Higher-Level Synthesis: experimenting with MLIR polyhedral representations for accelerator design. In *Proceedings of IMPACT '22: 12th International Workshop on Polyhedral Compilation Techniques (IMPACT 22).* 10 pages.

1 Introduction

High-Level Synthesis (HLS) tools have become a critical element of the hardware design process. By allowing developers to describe an algorithm with a general purpose programming language (typically C or C++) and translating it into an implementation in a hardware description language (HDL) such as Verilog or VHDL, they significantly reduce the hardware design productivity gap [1, 20]. In particular, HLS tools enable to program Field Programmable Gate Arrays (FPGAs) without writing any HDL code, making them more accessible to users from different domains. Systems integrating FPGAs can thus be exploited to implement highly specialized accelerators, as demonstrated by successes in machine learning and data analysis [8].

HLS tools are effectively compiler frameworks, as they translate a language into another language at a lower level of abstraction. As such, they benefit from the same compiler optimizations that identify instruction, memory, and data parallelism for general purpose and specialized processors. However, they also need to consider the very specific needs of low-level circuit design, such as the notion of time, the difference between synchronous and asynchronous logic, and wiring delays, so they typically work on a low-level intermediate representation that is close to the actual hardware design. Because of the mismatch between the requirements of hardware abstractions and the characteristics of general purpose programming languages, HLS tools often require the addition of specific directives (pragmas) that augment the input C/C++ specification to guide the generation of specialized hardware.

Polyhedral optimization techniques are particularly suited to workloads that involve deeply nested loops and arrays, such as linear algebra solvers for high-performance scientific simulation, or tensor-based machine learning frameworks [2, 9]. Typical targets for polyhedral compilers include modern general purpose processing architectures with vector units and large multi-level caches, and specialized accelerators such as general purpose graphic processing units or tensor processing units.

Recent works, however, have argued the importance of the polyhedral model in the field of HLS and automated hardware generation, usually by transforming and annotating the input C/C++ code, either through ad hoc implementations to demonstrate specific optimizations or with the help

*Also with Pacific Northwest National Laboratory.

of external frameworks [4, 13, 26]. Such approaches have limited flexibility to allow the introduction of new types of optimizations, and they risk loss of semantic information in the process. In this paper, instead, we discuss the use of a higher-level compiler framework to perform optimizations for HLS in a modular way, at a specific level of abstraction.

More specifically, we implement our approach exploiting the MultiLevel Intermediate Representation (MLIR) framework [11]. MLIR is a recent contribution to the LLVM project that enables and encourages 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. As MLIR was conceived initially to be applied within machine learning frameworks, efficiently supporting the introduction of polyhedral optimization was a key objective of the project, and one of the first areas specifically addressed through a dedicated abstraction, i.e., the "affine" dialect.

In this paper, we choose to focus our attention on loop pipelining, an optimization technique that is easily enabled by polyhedral frameworks and that can have a significant impact on HLS. Loop pipelining overlaps iterations depending on available computational resources and memory dependencies, with the aim of parallelizing as many operations as possible. 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. We evaluate the impact of a loop pipelining transformation implemented in MLIR by using an open-source HLS tool, Bambu [7], and by comparing our approach to traditional C-based HLS.

In summary, this paper makes the following contributions:

- we present an implementation of loop pipelining for HLS leveraging a higher level of abstraction (MLIR) than conventional HLS approaches;
- we show how the polyhedral abstraction provided by the MLIR affine dialect facilitates the implementation of such a transformation;
- we highlight how a modern reusable compiler infrastructure provides opportunities to improve the HLS process, and to easily integrate new or unsupported optimizations.

The paper proceeds as follows. Section 2 introduces the main concepts and tools used in the implementation of our approach; Sections 3 and 4 dive deeper into the details of the design flow we are proposing. We present experimental results in Section 5. Section 6 describes current approaches for supporting polyhedral optimization in HLS and other related work, Section 7 draws conclusions and outlines future research directions.

2 Background

This section briefly describes the compiler frameworks, tools, and concepts that are used throughout the paper.

2.1 MLIR

MLIR [11] aims to build a reusable and extensible compiler infrastructure, allowing optimizations on different levels of abstraction through the concept of dialects. Elementary MLIR defines an SSA-based IR consisting of MLIR operations, but dialects can extend the core MLIR structures by representing a set of new operations, attributes, and types, sharing a specific purpose.

MLIR operations consist of a name, operands, attributes, results, and, optionally, nested regions. A region represents an ordered list of MLIR blocks, while blocks represent ordered lists of operations with a single terminator operation at the end. Blocks are compiler basic blocks that compose the control flow graph of a program. MLIR provides the concept of passes to expose entry points for IR analyses and transformations which traverse the program to either collect useful information or apply transformations.

Dialects typically operate at higher levels of abstraction with respect to C/C++, and to the instruction-level IRs of HLS tools; moreover, the possibility of combining different dialects in the same representation opens the way to the integration of novel compilation passes and optimizations. In particular, high-level affine structures can coexist with low-level operations on SSA values, allowing the application of both polyhedral loop transformations and traditional compiler optimizations. This is an improvement with respect to traditional polyhedral frameworks, as they usually require relevant parts of the program to be "raised" to a higher level of abstraction; conversion to and from such a different representation is not trivial and risks the loss of valuable information.

Basic operations such as memory or computation operations are provided within the "standard" dialect; we will focus on the standard and affine dialects in our implementation, but other dialects that can be lowered to standard and affine could exploit our loop pipelining passes, and may provide future opportunities for optimization.

2.2 Loop pipelining

Loop optimizations have been widely studied both in software and hardware research. A single loop iteration usually does not contain enough optimization potential, so various techniques have been used to overcome this problem. The simplest example is loop unrolling: by replicating the loop body multiple times, the single iteration becomes larger and exposes more possibilities to execute instructions in parallel. Loop pipelining, instead, aims at overlapping the execution of multiple iterations, and it involves a more complicated transformation process than loop unrolling. This technique

has been successfully used in compiler infrastructures for decades [10], and it generally consists of two steps: loop scheduling and code generation. Depending on the available computation and memory resources, and if inter-iteration data dependencies allow it, a pipelined loop can issue the execution of a new iteration at every clock cycle.

2.3 HatSchet scheduler

Loop scheduling requires knowledge about both the operations contained in the loop body, and the available computational resources. Achieving the optimal or close to optimal operation schedule, i.e., a list of operations assigned to a clock cycle number and a resource identifier, is not a trivial process: this job is performed by schedulers using various scheduling algorithms and heuristics. We use the library provided by HatSchet 0.8 [18], an open-source scheduling tool with the purpose of making HLS processes more efficient. HatSchet requires data flow graph and resource availability information as inputs, and it offers various scheduling algorithms, enabling control over scheduling run time and quality trade-offs.

2.4 High-Level Synthesis tools

We choose two different HLS tools to validate our approach: Bambu and Vitis HLS. Bambu [7] is an open-source HLS tool compatible with both C/C++ and LLVM IR inputs, and it has no internal support for loop pipelining. Vitis HLS, instead, is a commercial tool by Xilinx supporting C/C++ inputs augmented by custom pragmas; in its backend Vitis HLS automatically tries to pipeline all loops in the code with an Iteration Interval of 1, progressively relaxing the constraint if necessary.

Bambu is the main target of our case study, precisely because it is able to synthesize LLVM IR (which is a natural target for MLIR lowering), and because it would not otherwise be able to pipeline loops. Xilinx recently released an open-source frontend for Vitis HLS that operates on LLVM IR, but it is tightly connected with the custom pragma annotations on one side, and to the hardware generation (closed-source) backend on the other: this makes it difficult to implement optimizations for Vitis HLS that can be reused by other tools. We use Vitis HLS to verify that our MLIR-based approach to polyhedral transformations has similar or greater effects on the generated accelerator performance with respect to a low-level implementation of the same optimization.

3 Proposed approach

Our approach aims to leverage high-level code optimizations to provide a hardware-oriented input description to High-Level Synthesis. This section presents an overview of the proposed design flow, leaving implementation details to the next section. Figure 1 shows the main steps and tools involved: our input code contains a loop to be pipelined, so

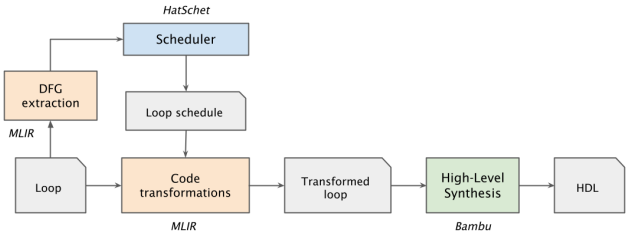

Figure 1. Overview of the proposed optimization flow.

the code is first passed to a *scheduler* to obtain a loop iteration schedule. Then, we implement *code transformations* that work on the input code and use the schedule to produce the pipelined loop. The resulting code is finally translated and processed by the *HLS tool* to generate an accelerator description in Verilog/VHDL.

As previously mentioned, loop pipelining requires a scheduling phase and a code generation phase; we will introduce here a simple example that will be useful to illustrate these steps more in detail. Let us consider a for loop that reads values from an array, multiplies them with a constant, and writes them into another array. A single iteration of this simple loop contains three operations: load, multiply, and store. Figure 2a represents the data flow graph of one iteration; clearly, the three operations depend on each other and cannot be parallelized.

Loop pipelining allows to schedule operations from different original iterations together: as these operations would not depend on each other, they could be executed in parallel without constraints. The result is shown in Figure 2b, where each column represents one iteration of the new loop, and operations originating from the same original iteration use the same color. By overlapping original iterations, loop pipelining eliminates the parallelization constraints: all operations within the same iteration are independent now, so they can be executed in parallel. Incomplete iterations at the beginning form a loop *prologue*; the last few iterations are also incomplete, and they form a loop *epilogue*. The new loop is built of the complete iterations between prologue and epilogue. In the example shown in Figure 2b, iterations I1 and I2 belong to the loop prologue, I N+1 and I N+2 represent the epilogue, while the actual new loop starts from I3. If we assume that all functional units execute in one clock cycle, the achieved II in this simple example is equal to 1.

Up to now, we did not consider resource availability. Our example loop cannot be pipelined if there are not enough memory elements available to run two operations (one load and one store) simultaneously: this is why we need to perform scheduling before transforming the loop into its pipelined version. If we tell the scheduler that one load and one store unit are available, and that all functional units have a delay of one cycle, it will correctly schedule our example loop as

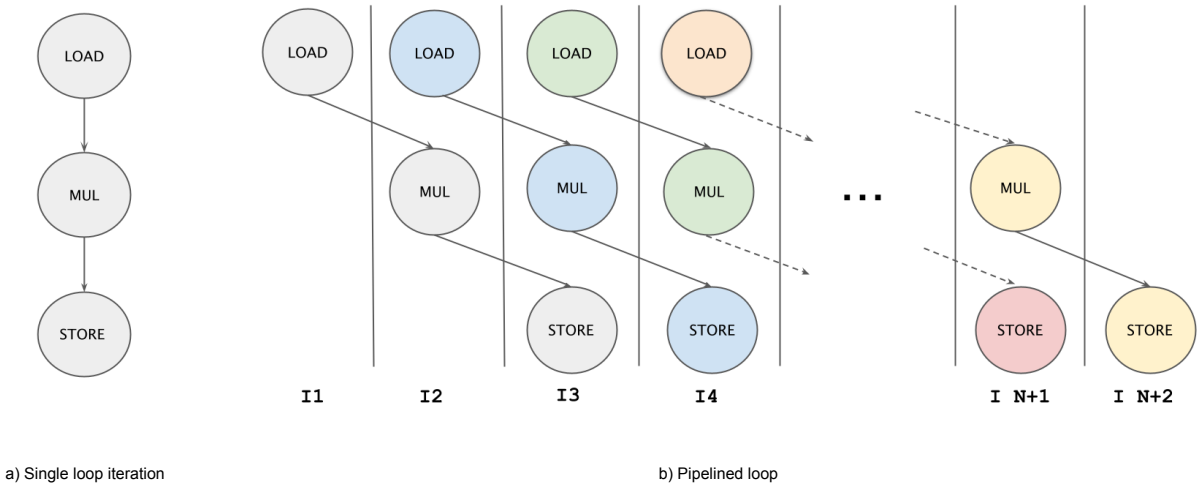

Figure 2. Creation of a pipelined loop.

CYCLE	LOAD0	STORE0	MUL0	
0	LOAD			PROLOGUE
1	LOAD		MUL	
2	LOAD	STORE	MUL	
3	LOAD	STORE	MUL	NEW LOOP ITERATIONS
	...	...	...	
N+1		STORE	MUL	EPILOGUE
N+2		STORE		

Figure 3. Pipelined loop schedule.

shown in Figure 3, providing the necessary information to produce a pipelined loop with II=1. For the sake of simplicity, in the rest of the discussion we will always assume that the scheduler has an infinite number of resources available for each type of operation, and that they all execute in one clock cycle.

In our optimization flow, scheduling is performed by HatSchet, and code generation is implemented as a set of transformations in MLIR; the pipelined loop is then passed to Bambu to obtain an HDL implementation. It represents an alternative to other loop pipelining approaches that delegate scheduling and pipelining to the HLS tool itself: this is the case of Vitis HLS, where optimizations can be controlled through pragmas in the input code, but even in absence of user-specified directives the tool tries to transform the code in order to achieve the lowest possible II [21]. Bringing loop pipelining (and possibly other optimizations) outside the scope of the HLS tool has significant advantages: for example, the developer is more in control of the applied techniques, as their effects are visible in the transformed IR. Moreover, applying transformations on a specialized, higher-level abstraction increases flexibility, portability, and requires less time than implementing and exploring different techniques within the HLS tool. Finally, MLIR is built to allow easy integration between different optimizations: this means that loop pipelining may be combined with other techniques to create inputs to the HLS tool that are more appropriate to generate efficient hardware accelerators.

4 Implementation

We implemented two MLIR passes to support the proposed design flow: the first one extracts a data flow graph from the MLIR loop body, the second one generates the pipelined loop code according to the schedule produced by HatSchet. We also implemented an if-conversion pass following [19] that allows to pipeline loops that contain if and else blocks, which we will not describe in detail as it is less relevant to the main focus of the paper. In the following, we will keep referring to the example loop described in Section 3; Figure 4a shows its description in MLIR.

4.1 Data Flow Graph Extraction Pass

This MLIR pass visits all the operations in the loop and extracts their dependencies to build a data flow graph that HatSchet will be able to schedule. Nodes of this graph represent operations; edges represent dependencies between operations. There are two kinds of dependencies: precedence and data. Precedence dependency refers to a simple result usage in the code; data dependencies exist between two memory operations accessing the same memory location. A data dependency can occur between two memory operations only if at least one of them is a store operation (two loads are always independent of each other). Data dependencies have a distance attribute to express the distance between the loop iterations that contain the two operations. Precedence dependencies do not have a distance attribute: the result of an operation in an iteration cannot be used in a different iteration without passing through memory.

```
1 func @example(%arg0: memref<1000xi32>) {
2   affine.for %arg1 = 0 to 1000 {
3     %0 = affine.load %arg0[%arg1]
4     %1 = muli %0, %0
5     affine.store %1, %arg0[%arg1]
6   }
7   return
8 }
```

(a) Original loop in MLIR.

```
# II 1
# vertex;          cycle;   functional_unit
affine.load_1;       0;          load0
muli_2;              1;          mul0
affine.store_3;      2;          store0
```

(b) HatSchet schedule.

```
1  #map = affine_map<(d0) -> (d0 - 2)>
2  func @example(%arg0: memref<1000xi32>,
                 %arg1: memref<1000xi32>) {
3    %c0 = constant 0 : index
4    %0 = affine.load %arg0[%c0] : memref<1000xi32>
5    %c1 = constant 1 : index
6    %1 = affine.load %arg0[%c1] : memref<1000xi32>
7    %2 = muli %0, %0 : i32
8    %3:2 = affine.for %arg2 = 2 to 1000
           iter_args(%arg3 = %1, %arg4 = %2) -> (i32, i32) {
9      %5 = affine.load %arg0[%arg2] : memref<1000xi32>
10     %6 = muli %arg3, %arg3 : i32
11     %7 = affine.apply #map(%arg2)
12     affine.store %arg4, %arg1[%7] : memref<1000xi32>
13     affine.yield %5, %6 : i32, i32
14   }
15   %4 = muli %3#0, %3#0 : i32
16   %c998 = constant 998 : index
17   store %3#1, %arg1[%c998] : memref<1000xi32>
18   %c999 = constant 999 : index
19   store %4, %arg1[%c999] : memref<1000xi32>
20   return
21 }
```

(c) Scheduled loop in MLIR.

Figure 4. Creation of a pipelined loop through HatSchet and MLIR.

Extracting nodes and precedence edges from MLIR code is trivial; a simple loop operations visit is sufficient. Data dependence analysis represents a more significant challenge: we solve it by using an existing MLIR affine method named `checkMemRefAccessDependence`. This method analyses a pair of memory operations and decides if a dependency exists (the distance can also be easily deduced from its output). The implementation of the data flow graph extraction pass was greatly simplified by existing affine constructs, confirming that the MLIR dialect-based approach provides a convenient system for the quick introduction of new optimizations.

Algorithm 1 Prologue extraction

```
1:  function EXTRACT_PROLOGUE(schedule, II,
        originalIterationSize)
2:      repeat
3:          iteration = new_empty_iteration()
4:          start_cycle = 0
5:          current_prologue_iteration = prologue.size
6:          while (current_prologue_iteration >= 0) do
7:              cycles ← schedule.get_cycles(
8:                  start: start_cycle, end: start_cycle + II)
9:              iteration.schedule_cycles(cycles)
10:             current_prologue_iteration =
11:                 current_prologue_iteration - 1
12:             start_cycle = start_cycle + II
13:         end while
14:         if iteration.size < originalIterationSize then
15:             prologue.add(iteration)
16:         end if
17:     until iteration.size == originalIterationSize
18: end function
```

4.2 Loop Transformation Pass

This pass loads the HatSchet schedule format (Figure 4b) into a convenient data structure and uses it to generate code for the pipelined loop. The following paragraphs will highlight all the steps that are needed to transform the original loop of Figure 4a into the scheduled loop of Figure 4c.

Prologue and epilogue extraction. HatSchet provides only the new loop iteration schedule, without prologue and epilogue, so the MLIR loop transformation pass needs to extract operations and generate code for prologue and epilogue. Prologue and epilogue are not loops, but it is useful to reason in terms of prologue and epilogue *iterations* to identify the blocks of operations that are issued together (for example, columns I1 and I2 in Figure 2b).

The algorithm for prologue extraction (Algorithm 1) starts iterations from the original loop sequentially until it arrives at the first iteration of the new loop: the exit condition is satisfied once it generates an iteration containing all operations that are in the schedule (this iteration is not included in the prologue). The first prologue iteration will contain operations with an overall latency of II, extracted from the first original loop iteration. Then, each following prologue iteration will start a new original loop iteration and continue previously started original iterations with blocks of operations that cover II cycles.

Looking at Figure 2b, we have a loop with II=1 and three operations in each iteration: this means that the prologue extraction algorithm will schedule a single load operation in I1, then schedule the load operation from the next iteration

Algorithm 2 Prologue generation

```
1:  function GENERATE_PROLOGUE(prologue_iterations, operand_maps)
2:      for iteration : prologue_iterations do
3:          for cycle : iteration do
4:              for operation, original_iteration : cycle do
5:                  for operand : operation.operands do
6:                      if operand == original_index_variable then
7:                          constant_operation = generate_constant_operation(value: original_iteration)
8:                          operand_maps[original_iteration].insert(operand, constant_operation.result)
9:                      end if
10:                     new_operation = operation.clone(maps[original_iteration])
11:                     maps[original_iteration].insert(operation.results, new_operation.results)
12:                 end for
13:             end for
14:         end for
15:     end for
16: end function
```

and the multiplication to continue the previous iteration in I2. The algorithm stops when it generates I3 and sees that it has the same operations of the scheduled iteration, so it will be discarded.

The epilogue can be extracted similarly. However, in this case we are not starting new iterations but instead finishing the ones started in the loop iterations: the exit condition is satisfied when the size of the generated iteration equals zero.

Operations mapping. Subsequent code generation stages consist of cloning original loop operations and updating their operands to reference new operation results. MLIR offers a mechanism to create a deep copy of an operation while assigning new values to its operands using a map: keys in the map represent the initial results of each operation, while values contain the results of the new cloned operations. We use separate maps for each original iteration to ensure we keep the correct operation dependencies; we record a mapping each time we generate a new operation.

In the example code of Figure 4c the MLIR pass creates five maps: two for the original iterations starting in the prologue, one for all original iterations starting in the new loop, and two for the original iterations finishing in the epilogue. In fact, the original iterations fully executed within the new loop body can use a single map because there is no need to distinguish them from each other.

Prologue generation. Prologue generation traverses the prologue extracted from the schedule and creates operations by cloning the original ones and populating the correct maps (Algorithm 2). As we create new operations in order, the cloning map always contains the new values needed to substitute old operands. Other operands that need to be replaced are the ones that depend on the loop index variable: in this case, we calculate a new variable summing the lower bound of the loop index with the number of the original iteration. In simple cases where the lower bound is a constant, the new expression is also a constant.

In our example, prologue generation needs to generate three operations - two loads and one multiply. The first load is cloned from the original operation within the loop and moved in front of the loop. Since the load address depends on the loop index variable, we also need to replace it with an additional constant: we get its value by summing the original iteration number (zero) with the loop lower bound (zero). The second load is cloned and moved in the same way; its constant will have a value of one since the operation comes from original iteration one. When the multiply operation is cloned and moved, it correctly uses the result of the first load: in fact, as soon as the load operation is generated, its result is correctly allocated in the map so that all operations that used it in the original loop are redirected to its new value.

New loop generation. Similarly to what happens during prologue generation, each operation in the new loop is created by cloning the original loop operation and mapping its operands. If an operand requires the result of an operation from inside the loop body, this is handled by the default mapping mechanism; if an operand requires a result from an iteration started in the prologue, the loop index variable will be adjusted by a map to refer to the correct value. For example, line 11 in Figure 4c produces the correct index by subtracting 2 from %arg2.

Inter-iteration argument passing. Loop pipelining requires the loop to pass results from one iteration to the next one, and this is usually solved in hardware by using dedicated registers. As we work on a higher level of abstraction, we need to pass these results explicitly in the MLIR code. MLIR offers a mechanism to pass arguments to each new iteration of the loop: operands that need to be passed to a following iteration of the loop can *yield* their result, which

Table 1. Performance of selected Polybench kernels with Vitis HLS pipelining and MLIR pipelining. Two different measures of speedup are provided: pipelined over baseline, and MLIR high-level optimization over traditional HLS.

Benchmark	*Number of clock cycles*				*Speedup*			
	C + Vitis		**MLIR opt + Bambu**		**Pipelining**		**MLIR opt**	
	baseline	*pipelined*	*baseline*	*pipelined*	*Vitis*	*Bambu*	*baseline*	*pipelined*
2mm	261 186	107 908	214 914	76 482	2.42x	2.81x	1.22x	1.41x
3mm	347 775	166 457	304 576	117 428	2.09x	2.59x	1.14x	1.42x
atax	52 826	13 915	41 911	16 869	3.80x	2.48x	1.26x	0.82x
bicg	27 298	11 842	41 887	8749	2.31x	4.79x	0.65x	1.35x
doitgen	188 421	95 282	130 742	69 222	1.98x	1.89x	1.44x	1.38x
gemm	266 061	121 082	244 622	83 002	2.20x	2.95x	1.09x	1.46x
gemver	112 764	25 836	90 122	25 845	4.36x	3.49x	1.25x	1.00x
mvt	51 442	23 522	43 362	16 722	2.19x	2.59x	1.19x	1.41x
syr2k	294 841	76 066	227 582	70 910	3.88x	3.21x	1.30x	1.07x
syrk	168 841	75 766	153 182	57 490	2.23x	2.66x	1.10x	1.32x
trmm	103 541	94 502	74 362	37 392	1.10x	1.99x	1.39x	2.53x

```
8   %3:3 = affine.for %arg2 = 2 to 1000
          iter_args(%arg3 = %1, %arg4 = %2, %arg5 = %3)
                    -> (i32, i32, i32) {
9     %5 = affine.load %arg0[%arg2] : memref<1000xi32>
10    %6 = muli %arg3, %arg3 : i32
11    %7 = affine.apply #map(%arg2)
12    affine.store %arg4, %arg1[%7] : memref<1000xi32>
13    affine.yield %5, %arg5, %6 : i32, i32
14  }
```

Figure 5. Inter-iteration argument passing.

will be available as *iteration argument* at the beginning of the next iteration.

In our example, the new loop body consists of three operations, all from different original iterations: the load operation does not need any mapping of the operands, the multiplication uses the result of the load from the prologue (first iteration argument), the store uses the result of the multiplication from the prologue (second iteration argument). After each iteration of the new loop, load and multiply results are yielded to serve as iteration arguments for the next iteration.

Scheduling might overlap instructions in such a way that lifetimes of operation results span multiple iterations, introducing the risk of overwriting an old value before using it. Rai et al. [17] suggest two different solutions to this problem: loop unrolling and rotating register files. We decided to avoid unrolling since it introduces further code expansion, while rotating register files requires architectural support that we cannot provide working on a higher level of abstraction. Instead, we solved this by using additional iteration arguments, such as the third iteration argument in Figure 5.

Epilogue generation and old loop removal. Epilogue generation is similar to prologue generation, with an additional step that maps the results of yield operations from the last loop iteration to epilogue operands. Finally, the old loop is removed from the code.

5 Experimental results

In this section we present preliminary results that validate our approach. We chose to focus our experiments on selected kernels from the Polybench benchmark suite, and to compare our MLIR-based optimization with the one performed by Vitis HLS on its internal IR.

To this end, we first synthesize the standard C version of Polybench kernels with Vitis HLS 2021.1, selecting always the smallest kernel size ('mini' dataset, version 4.2.1) and double-precision floating point operations. We obtain baseline results by adding a specific directive that forces Vitis HLS to never apply pipelining, then we remove the directive to trigger the automatic pipelining behaviour and we verify from synthesis logs that only the innermost loops were pipelined. Subsequently, we use Polygeist [15] to translate the same kernels to MLIR. We apply our high-level optimizations to the innermost loops, translate the kernels into LLVM IR and synthesize them with Bambu 0.9.7 as depicted in Figure 1. With both Vitis HLS and Bambu we synthesize for a Xilinx Zynq-7000 FPGA with a target frequency of 100 MHz, we assume that all data is stored in on-chip BRAMs, and all results are reported post place-and-route.

Table 1 shows the execution time of the generated accelerators: pipelining loops 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. A more interesting comparison can be drawn between the two approaches to HLS: moving from C code optimized by Vitis HLS to our proposed MLIR-based flow provides a speedup in almost every benchmark, with and without pipelining.

Table 2. Resource consumption of selected Polybench kernels before and after MLIR pipelining. The last column represents the speedup introduced by loop pipelining normalized by the area overhead.

Benchmark		*DSPs*	*Registers*	*LUTs*	*Slices*	*Slices overhead*	*Speedup / Slices overhead*
2mm	*baseline*	10	2306	2679	1053	1.78x	1.58x
	pipelined	20	4534	5124	1876		
3mm	*baseline*	10	2756	3067	1246	1.68x	1.54x
	pipelined	10	4626	4946	2096		
atax	*baseline*	10	1870	2121	803	2.76x	0.90x
	pipelined	30	4701	5658	2216		
bicg	*baseline*	10	1787	1825	708	4.08x	1.17x
	pipelined	60	6570	6917	2889		
doitgen	*baseline*	10	1558	2174	797	1.47x	1.28x
	pipelined	10	2728	2906	1174		
gemm	*baseline*	10	1788	2085	829	2.56x	1.15x
	pipelined	20	4548	5488	2126		
gemver	*baseline*	20	3270	3727	1463	3.16x	1.10x
	pipelined	70	11265	13943	4629		
mvt	*baseline*	10	1858	2192	843	1.51x	1.71x
	pipelined	10	3083	3174	1275		
syr2k	*baseline*	20	2711	2640	1030	3.65x	0.88x
	pipelined	90	9106	9972	3760		
syrk	*baseline*	10	1920	2037	786	3.06x	0.87x
	pipelined	30	5459	6675	2404		
trmm	*baseline*	10	1567	1852	736	1.36x	1.46x
	pipelined	10	2423	2753	1003		

When we schedule the affine code in MLIR, all operations in the new loop body are independent and can be executed in parallel: this means that measuring the II actually corresponds to measuring the execution latency of one loop iteration. By inspecting the Finite State Machines generated by Bambu and the Vitis synthesis logs, we observed that in several occasions the II reported by Vitis is higher than the iteration latency for the loop synthesized by Bambu. We suppose that this happened when Vitis was not able to correctly resolve operation dependencies in its internal low-level IR, while our approach facilitates this task by working at a higher level of abstraction in the specialized MLIR affine dialect.

Looking at area consumption, Table 2 reports the resource utilization of the benchmarks we synthesized from MLIR. Pipelining loops increases digital signal processing slices (DSPs), registers, look-up tables (LUTs) and slices consumption: however, if we divide the speedup by the area overhead we obtain that in most cases the price we are paying in terms of area is adequately compensated by the reduction in the number of clock cycles. The benchmarks where this does not happen may be too complex to benefit from innermost loop pipelining only. For example, the innermost loops in syrk and syr2k have an upper bound depending on the induction variable of the outermost loop: if the iterations executed in the pipelined loop are less than the iterations started in the prologue, the results may be wrong. To avoid this risk, we introduced a check at runtime to assess whether there are enough iterations to safely execute the pipelined loop, falling back on the original loop if this is not the case; however, having both the original and the pipelined loop in the code caused significant area overhead.

These preliminary results are a promising starting point to explore synthesis-oriented polyhedral optimizations in MLIR. Introducing loop pipelining as a high-level transformation was considerably easier than modifying the internal mechanisms of Bambu, and it gives more control to the user. While in a few experiments the MLIR-based design flow delivered worse performance than what Vitis HLS could achieve, it is important to notice that the experiments were relying on automatic optimizations performed by Vitis. There might have been many other transformations involved (e.g. loop unrolling, memory access optimizations) that are invisible to the user, and were not available or not triggered during Bambu synthesis.

6 Related work

There is significant interest in leveraging the polyhedral model to perform optimizations for HLS, especially when synthesizing for FPGAs [25, 26]. Polyhedral optimization, in fact, provides unique opportunities for parallelization, pipelining, and optimization of memory accesses, which are

all fundamental aspects to exploit the spatial parallelism provided by reconfigurable devices. Existing polyhedral approaches and frameworks for HLS are fundamentally source-to-source translators that process C/C++ inputs and rewrite restructured C/C++ code, possibly annotated with tool-specific optimization directives. As a consequence, even if they show great promise to improve performance of the synthesized circuits, it is inherently difficult to combine their polyhedral transformations with other types of optimizations.

As previously highlighted, loop pipelining is a key optimization for HLS, and thus it has been explored from several different perspectives. These include works like [12], which focuses on obtaining an II close to 1 even for loops that have non-constant dependencies, or [5], which applies speculative loop pipelining to HLS. Other works exploit polyhedral frameworks to implement dynamic loop pipelining [13, 14]. In all these approaches, polyhedral analysis enables the generation of specialized logic that decides when to safely run all loop iterations in the pipeline and when to interrupt the pipeline execution to resolve memory conflicts. However, even these approaches in the end resort to re-generating annotated C/C++ code for a specific HLS tool, limiting the possibilities to explore further non-polyhedral optimizations.

Our paper shows how a high-level abstraction designed for polyhedral optimization (the MLIR affine dialect) can be leveraged to effectively implement loop pipelining for HLS. Such a compiler-based approach, that works outside of the HLS tool, naturally lends itself to the addition of further optimizations. For example, Fellahi and Cohen [6] tackle nested loop optimizations and propose merging epilogue and prologue of adjacent iterations: this could be easily implemented in the future as another MLIR pass on top of the one we presented in this paper. Zhao and Cheng [24] have recently proposed Phism, a workflow similar to ours that bridges the gap between polyhedral tools and High-Level Synthesis. They intend to use Polygeist as a frontend for C/C++ programs, and to exploit the MLIR Affine dialect to implement progressive lowerings where each optimization can benefit from a different level of abstraction. Our loop pipelining passes are theoretically compatible with any other optimization implemented within MLIR, so in the future they could be combined with any other polyhedral optimization that will be proposed in Phism.

One of the main features provided by MLIR is to enable specialized abstractions (dialects) to solve different optimization problems; dialects are meant to be reusable across compilation pipelines, and they can co-exist together in the same MLIR representation. This will allow polyhedral optimization to be part of more complex and complete MLIR-based HLS infrastructures that bridge the gap from high-level, domain specific frameworks to hardware design [16, 23]. Other research efforts have also also started to look at MLIR as a mean to improve the HLS process. These include ScaleHLS [22], which proposes to use MLIR to analyze and transform input code to HLS, as we do, but then regenerates again a C program containing directives targeting Vivado HLS, effectively leaving a specific backend tool in charge of applying the optimizations. The CIRCT project [3], instead, intends to prove the adaptability of MLIR by applying it to implement interoperable hardware design tools at a lower level of abstraction.

7 Conclusion

We believe that applying polyhedral optimizations at the appropriate level of abstraction can be beneficial to improve HLS results. To support this claim, we implemented loop pipelining as a compiler pass within the MLIR framework, which offers the affine representation specifically to enable polyhedral optimizations. We evaluated the effect of our high-level transformation by translating the obtained MLIR code into LLVM IR and feeding it to Bambu, an open source HLS tool that does not natively support loop pipelining. We compared the performance of accelerators generated with our MLIR-based design flow to the ones obtained by synthesizing C programs with Vitis HLS (which instead automatically pipelines loops), and showed how our design flow can produce similar or better results.

These preliminary experiments open the way to further research to explore polyhedral optimization techniques that can benefit HLS at a higher level of abstraction than existing solutions producing C/C++ code. A first step in this direction will be to evaluate how our loop pipelining implementation interacts with existing MLIR optimizations at the affine dialect level, and then to add other HLS-oriented compiler passes. Finally, thanks to the modular nature of MLIR, our loop pipelining pass could also be included into existing and future compiler-based design flows that work with the MLIR affine dialect.

Acknowledgments

This research was partially supported by the Pacific Northwest National Laboratory's (PNNL) Data-Model Convergence (DMC) Laboratory-Directed R&D Initiative, the Defense Advanced Research Project Agency's (DARPA) Real Time Machine Learning (RTML) program, and the EVEREST project funded by the EU Horizon 2020 Program under grant agreement No 957269.

References

[1] Semiconductor Industry Association et al. 1999. International Technology Roadmap for Semiconductors 1999 Edition. *http://www. itrs. net/ntrs/publntrs. nsf* (1999).

[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *CGO 2019: Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. 193–205.

[3] CIRCT Developers. 2020. "CIRCT" / Circuit IR Compilers and Tools. https://github.com/llvm/circt

[4] Gaël Deest, Nicolas Estibals, Tomofumi Yuki, Steven Derrien, and Sanjay Rajopadhye. 2016. Towards Scalable and Efficient FPGA Stencil Accelerators. In *IMPACT'16 - 6th International Workshop on Polyhedral Compilation Techniques, held with HIPEAC'16*.

[5] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. 2020. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4229–4239.

[6] Mohammed Fellahi and Albert Cohen. 2009. Software pipelining in nested loops with prolog-epilog merging. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 80–94.

[7] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *DAC 2021: 58th ACM/IEEE Design Automation Conference*.

[8] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *ISCA 2018: ACM/IEEE 45th Annual International Symposium on Computer Architecture*. 1–14.

[9] Tobias Grosser, Groesslinger Armin, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation. *Parallel Processing Letters* 22, 04 (28 Dec. 2012).

[10] Monica S. Lam. 1989. Software pipelining. In *A Systolic Array Optimizing Compiler*. Springer, 83–124.

[11] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO 2021: IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 2–14.

[12] Peng Li, Louis-Noël Pouchet, and Jason Cong. 2014. Throughput optimization for high-level synthesis using resource constraints. In *IMPACT '14: International Workshop on Polyhedral Compilation Techniques*.

[13] 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.

[14] Junyi Liu, John Wickerson, and George A Constantinides. 2016. Loop splitting for efficient pipelining in high-level synthesis. In *FCCM 2016: IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines*. 72–79.

[15] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *PACT 2021: 30th International Conference on Parallel Architectures and Compilation Techniques*. 45–59.

[16] Christian Pilato, Stanislav Böhm, Fabien Brocheton, Jerónimo Castrillón, Riccardo Cevasco, Vojtech Cima, Radim Cmar, Dionysios Diamantopoulos, Fabrizio Ferrandi, Jan Martinovic, Gianluca Palermo, Michele Paolino, Antonio Parodi, Lorenzo Pittaluga, Daniel Raho, Francesco Regazzoni, Katerina Slaninová, and Christoph Hagleitner. 2021. 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*. IEEE, 1320–1325.

[17] B Ramakrishna Rau, Michael S Schlansker, and Parthasarathy P Tirumalai. 1992. Code generation schema for modulo scheduled loops. *ACM SIGMICRO Newsletter* 23, 1-2 (1992), 158–169.

[18] Patrick Sittel, Julian Oppermann, Martin Kumm, Andreas Koch, and Peter Zipf. 2018. HatScheT: A Contribution to Agile HLS. In *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*. VDE, 1–8.

[19] Arthur Stoutchinin and Guang Gao. 2004. If-conversion in SSA form. In *European Conference on Parallel Processing*. Springer, 336–345.

[20] Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *SNAPL 2019: 3rd Summit on Advances in Programming Languages*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 136. 7:1–7:21.

[21] Xilinx. 2021. Optimization Techniques in Vitis HLS. https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/vitis_hls_optimization_techniques.html

[22] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2021. ScaleHLS: Scalable High-Level Synthesis through MLIR. (2021). arXiv:2107.11673

[23] Jeff Jun Zhang, Nicolas Bohm Agostini, Shihao Song, Cheng Tan, Ankur Limaye, Vinay Amatya, Joseph Manzano, Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, Gu-Yeon Wei, and David Brooks. 2021. Towards Automatic and Agile AI/ML Accelerator Design with End-to-End Synthesis. In *ASAP 2021: IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors*. 218–225.

[24] Ruizhe Zhao and Jianyi Cheng. 2021. Phism: Polyhedral High-Level Synthesis in MLIR. In *LATTE virtual workshop, April 15, 2021*. arXiv:2103.15103.

[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 *CODES+ISSS 2013: International Conference on Hardware/Software Codesign and System Synthesis*. 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 '13: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 9–18.