POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

MODERN HIGH-LEVEL SYNTHESIS: IMPROVING PRODUCTIVITY WITH A MULTI-LEVEL APPROACH

Doctoral Dissertation of:
Serena Curzel

Supervisor:
Prof. Fabrizio Ferrandi

Tutor:
Prof. Cristina Silvano

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

2022 – XXXV

Acknowledgements

MY PhD journey was supported by many wonderful people, on a professional and personal level. First and foremost, I send my love and gratitude to Paolo and my family, for being there through the good and the bad times. Thanks to all my friends (especially Matteo and Iris), and to the Compass Club in Tri-Cities. You all kept me anchored to life outside work, and I could not have done it without your support.

I was fortunate to collaborate with two research groups, led by prof. Ferrandi at Politecnico di Milano and by dr. Tumeo at Pacific Northwest National Laboratory; I am grateful for the opportunities they gave me, for everything I learned and for all the people I met, and I hope we will continue to do great research together. A special mention goes to Michele and Nicolas, who were always helpful and supportive in front of my questions and doubts.

Finally, I also want to thank prof. Fummi and prof. Carloni for giving helpful feedback that turned my initial draft into the thesis you are about to read.

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, usually C/C++. They include a wide range of optimization techniques in the process, most of them performed on a low-level intermediate representation (IR) of the code. Because of the mismatch between the requirements of hardware descriptions and the characteristics of input languages, HLS tools often rely on users to add specific directives (pragmas) that augment the input specification to guide the generation of optimized hardware. A good result thus still requires hardware design knowledge and non-trivial design space exploration, which might be an obstacle for domain scientists seeking to accelerate applications written, for example, in Python-based programming frameworks.

This thesis proposes a modern approach based on multi-level compiler technologies to bridge the gap between HLS and high-level frameworks, and to use domain-specific abstractions to solve domain-specific problems. The key enabling technology is the Multi-Level Intermediate Representation (MLIR), a framework that supports building reusable compiler infrastructure inspired by (and part of) the LLVM project. The proposed approach uses MLIR to introduce new optimizations at appropriate levels of abstraction outside the HLS tool while still relying on years of HLS research in the low-level hardware generation steps; users and developers of HLS tools can thus increase their productivity, obtain accelerators with higher performance, and not be limited by the features of a specific (possibly closed-source) backend.

The presented tools and techniques were designed, implemented, and tested to synthesize machine learning algorithms, but they are broadly applicable to any input specification written in a language that has a translation to MLIR. Generated accelerators can be deployed on Field Programmable Gate Arrays or Application-Specific Integrated Circuits, and they can reach ~10-100 GFLOPS/W efficiency without any manual optimization of the code.

Contents

CHAPTER 1

Introduction

THIS thesis proposes a multi-level, compiler-based approach to overcome the limitations of High-Level Synthesis tools, improving productivity for both users and developers. The introduction illustrates the problems that the proposed approach aims to solve, and briefly describes the main tools and methods involved. A comparison is drawn between the proposed "modern" approach and the "classic" way HLS tools are used, focusing especially on the acceleration of machine learning models; the final section outlines the content of the following chapters.

The chapter contains material from:

S. Curzel, N. Bohm Agostini, A. Tumeo, and F. Ferrandi, “Hardware Acceleration of Complex Machine Learning Models through Modern High-Level Synthesis,” in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 2022, pp. 209–210.

1.1 Motivation

The exponential growth of data science and machine learning (ML), coupled with the diminishing performance returns of silicon at the end of Moore's law and Dennard scaling, is leading to widespread interest in domain-specific architectures and accelerators [2]. Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) can provide the necessary hardware specialization with higher performance and energy efficiency than multi-core processors or Graphic Processing Units (GPUs). ASICs are the best solution in terms of performance, but they incur higher development costs; FPGAs are more accessible and can be quickly reconfigured, allowing to update accelerators according to the requirements of new applications or to try multiple configurations in a prototyping phase before committing to long and expensive ASIC manufacturing.

ASICs and FPGAs are designed and programmed through hardware description languages (HDLs) such as Verilog or VHDL, which require developers to identify critical kernels, build specialized functional units and memory components, and explicitly manage low-level concerns such as clock and reset signals or wiring delays. The distance between traditional software programming and HDLs creates significant productivity and time-to-market gaps [3, 4] and traditionally required manual coding from expert hardware developers. The introduction of High-Level Synthesis (HLS) simplified this process, as HLS tools allow to automatically translate general-purpose software specifications, primarily written in C/C++, into an HDL description ready for logic synthesis and implementation [5, 6]. Thanks to HLS, developers can describe the kernels they want to accelerate at a high level of abstraction and obtain efficient designs without being experts in low-level circuit design.

Due to the mismatch between the requirements of hardware descriptions and the characteristics of general-purpose programming languages, HLS tools often require users to augment their input code through *pragma* annotations (i.e., compiler directives) and configuration options that guide the synthesis process, for example, towards a specific performance-area trade-off. Different combinations of pragmas and options result in accelerator designs with different latency, resource utilization, or power consumption. An exhaustive exploration of the design space does not require extensive changes to the input code, and it does not change the functional correctness of the algorithm, but it is still not a trivial process: the effect of combining multiple optimization directives can be unpredictable, and the HLS user needs a good understanding of their impact on the generated hardware.

Data scientists who develop and test algorithms in high-level, Python-based programming frameworks (e.g., TensorFlow [7] or PyTorch [8]) typically do not have any hardware design expertise: therefore, the abstraction gap that needs to be overcome is not anymore from C/C++ software to HDL (covered by mature commercial and academic HLS tools), but from Python to annotated C/C++ for HLS. The issue is exacerbated by the rapid evolution of data science and ML, as no accelerator can be general enough to support new methods efficiently, and a manual translation of each algorithm into HLS code is highly impractical. Compilers like XLA [9], Glow [10], and TVM [11] map tensor-based operations from ML frameworks to accelerators such as general-purpose GPUs with tensor cores, systolic arrays, or dataflow accelerators; their contribution is valuable, but their targets are still fixed architectures that have been optimized for a subset of primitives and cannot be easily extended. Design flows based on HLS provide more flexibility to generate efficient accelerators tailored to specific algorithms.

This thesis proposes a multi-level, compiler-based approach to bridge the gap between high-level frameworks and HLS. The key enabling technology is the Multi-Level Intermediate Representation (MLIR) [12], a reusable and extensible infrastructure in the LLVM project for the development of domain-specific compilers. MLIR allows defining specialized intermediate representations (IRs) called *dialects* to implement analysis and transformation passes at different levels of abstraction, and it can interface with multiple software programming frameworks. An MLIR-based approach is a "modern" solution to automate the design of hardware accelerators for high-level applications through HLS, as opposed to "classic" approaches that rely on hand-written template libraries. Section 1.2 compares the two alternatives in more detail focusing on the acceleration of ML models.

A practical realization of the proposed approach is the SOftware Defined Architectures (SODA) Synthesizer [13, 14], an open-source hardware compiler composed of an MLIR frontend [15] and an HLS backend [16]. SODA provides an end-to-end agile development path from high-level software frameworks to FPGA and ASIC accelerators, supports the design of complex systems, and allows to introduce and explore optimizations at many different levels of abstraction, from high-level algorithmic transformations to low-level hardware-oriented ones. Translation across different levels of abstraction is performed through progressive lowering between IRs, allowing each step to leverage information gathered in other phases of the compilation. In the frontend, domain-specific MLIR dialects (existing and introduced by SODA) allow developers to work on specialized abstractions to address system-level concerns and pre-optimize the code. The integration of an open-source tool in the backend allows to exploit years of

HLS research and to introduce new features in the low-level hardware generation steps when necessary.

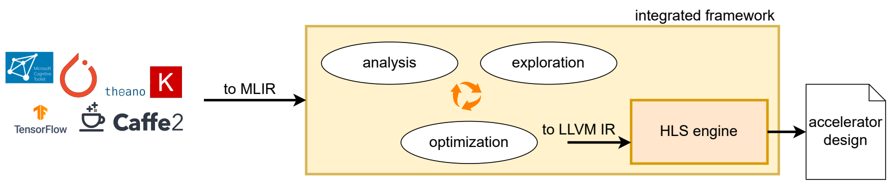
The following chapters will describe the proposed multi-level approach, showing how it improves productivity for users and developers of existing HLS tools, and how it can unlock multiple research opportunities for the design and optimization of domain-specific accelerators.

1.2 "Classic" and "modern" approaches

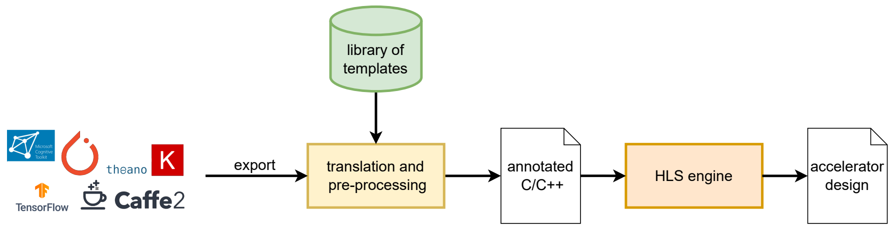
Machine learning (ML) and deep learning algorithms are well suited to process and analyze large amounts of data, as repeatedly proven in applications such as image classification, natural language processing, or recommendation systems. They continue to receive significant attention from industry and research, and they have also become a fundamental component in large-scale scientific experiments, where instruments acquire large amounts of raw data that need to be processed close to the sensors with extremely low latency. Both ML training and inference are compute- and memory-intensive, leading to widespread adoption of heterogeneous systems containing specialized hardware accelerators. GPUs are often the platform of choice to accelerate training; however, they are often too power-hungry to run inference tasks at the edge or cannot meet the strict latency requirements of real-time applications. A variety of custom solutions implemented on FPGA or ASIC have been proposed in their place, with different degrees of specialization: from generic "neural processors" such as Google's Tensor Processing Unit (TPU) [17], to accelerators that support a narrow set of models with great efficiency (e.g., targeting low-precision convolutional neural networks [18]).

High-Level Synthesis plays a crucial role in bridging the productivity gap between the design of a new ML model and its implementation on FPGA/ASIC. Several previous works proposed to exploit HLS by using C/C++ as an intermediate representation of the input model, augmenting it with tool-specific directives that drive the synthesis to obtain an efficient design. The most popular frameworks that help automate the design of ML accelerators (e.g., hls4ml [19], FINN [20]) use commercial HLS tools as backend, in what can be called a "classic" HLS-based design flow: they parse a model exported from popular ML frameworks and replace operators with C/C++ functions taken from a library of templates that already contains pragma directives (Figure 1.1a). The HLS tool processes this intermediate C/C++ representation and produces a corresponding accelerator design without further manual intervention.

What this thesis proposes, instead, is a "modern" compiler-based approach (Figure 1.1b): an integrated framework that exploits MLIR to interface with high-level



(a) *"Classic" HLS flow with intermediate C/C++.*



(b) *"Modern" compiler-based HLS flow.*

Figure 1.1: *Two different approaches to generate ML accelerators through HLS.*

ML frameworks, progressively lowers and optimizes the input model through different MLIR dialects, and delegates only the low-level hardware generation steps to the HLS tool. Such a compiler-based flow has two fundamental differences with respect to the "classic" approach. First, instead of directly translating a high-level description of the input into C/C++ code, the proposed approach embraces the MLIR multi-level approach to work with progressive lowerings: in this way, for example, the computational graph that describes the input ML model can be analyzed and transformed while it still contains information on the flow of data between layers, which is harder to reconstruct at the C/C++ level. Second, the HLS tool is more tightly integrated with the rest of the flow: this is particularly beneficial if the tool of choice is open source, as it is possible to have more control over the underlying synthesis process. For example, detailed information can be exchanged between MLIR analysis passes and hardware generation passes, or new MLIR optimizations can be implemented knowing precisely what their effect on HLS will be. The result is a fully automated flow where users never need to modify their input code, and developers can experiment with new techniques in a modular environment.

The proposed multi-level approach overcomes two critical limitations of existing "classic" frameworks: portability and scope. The library of templates, in fact, is necessarily tied to a specific HLS tool and a narrow set of supported models, as it requires expert HLS developers to implement in advance the best version of all necessary operators for a pre-determined backend tool. Portability is a problem for HLS in general, as typically there is one commercial tool for each hardware vendor and each tool expects coding patterns, annotations, and configuration directives that are not recognized by other tools. In a framework that heavily relies on a library of templates, switching to a new hardware target thus requires a new version of the library, as incompatible coding patterns and directives would be ignored by the new HLS backend, resulting in inefficient designs. In the proposed compiler-based framework, instead, the input to the HLS tool is an LLVM IR optimized in the MLIR frontend by target-agnostic high-level transformations. When a new backend tool (accepting LLVM IR as input) is introduced, any syntactic modification that is needed can be applied in an LLVM pass at the end of the compilation pipeline, while the rest of the design flow remains unchanged; thus the proposed modular multi-level approach improves portability between HLS tools.

A second limitation of the existing "classic" frameworks is that they usually focus on a narrow set of models, specifically deep and convolutional neural networks (DNNs/CNNs). Machine learning is an umbrella term that covers a broad spectrum of algorithms, while research works about hardware acceleration and HLS-based design flows have mostly been focused on the subset of ML models based on dense convolutions and matrix multiplications. Sometimes their scope is further limited by application requirements: for example, the original implementation of hls4ml was optimized for small, fully-connected models under tight latency constraints, reflecting the needs of a high-energy physics experiment at CERN. Appropriate implementation choices for such a specific case may not always be beneficial: hls4ml proposed to store network weights inside on-chip logic and unroll all loops to increase parallelism, which quickly depletes FPGA resources when considering a neural network with more layers and weights. Different application requirements and different operators in the input model require once again expert HLS developers to write and optimize new C/C++ templates.

While it is true that DNNs and CNNs cover a significant part of ML applications (especially in the computer vision field), there is ample room for exploring other classes of models, for example to accelerate scientific applications that work on sparse data structures or graphs. Large models are often compressed to reduce their computation and memory requirements, either by employing low-precision data types (quantization) or by removing operations with zero values (pruning). Quantization is well suited to hardware acceleration since custom precision operators can be implemented quickly and efficiently (also through dedicated HLS libraries). Sparsity, on the other hand, implies irregular computation, communication, and memory access patterns, which result in poor efficiency when mapped on accelerators or templates designed for dense models. Graph structures provide great expressive power to represent and analyze data in a variety of applications, from chemistry to language, social networks, recommendation systems, etc. Graph neural networks (GNNs) could benefit from hardware acceleration and require unique design choices: models that work on graphs include both sparse (aggregation) and dense (feature extraction) computation patterns, which are also affected by the input graph size; such characteristics could benefit from a task-based parallelism paradigm. Existing HLS-based design flows are good at extracting data- and instruction-level parallelism (e.g. by unrolling loops), but they are not equipped to deal with the irregular task-based patterns required by graph processing.

Finally, a narrow focus limits the possibility of quickly adapting to new algorithmic approaches, which would instead be desirable in a rapidly evolving field such as ML (and data science in general). A multi-level, MLIR-based approach allows to easily introduce specialized abstractions (dialects) to solve domain-specific optimization problems and support a diverse set of input models efficiently. The compilation flow can be tailored to the needs of each input specification by enabling and disabling compiler passes, exploring different solutions with no manual modifications to the code until application requirements are met. As the MLIR project grows, it provides more and more interfaces to high-level programming frameworks and domain-specific languages: the proposed design flow and its optimization features will then progressively be available to more classes of algorithms, not exclusively ML models. In the context of a scientific experiment, this will allow translating into hardware also other stages in the data acquisition pipeline, such as pre-processing, analysis, and simulation.

1.3 Thesis structure

The rest of the thesis is structured as follows:

- Chapter 2 presents a survey of the state of the art, focusing in particular on solutions to the limitations of existing HLS tools, automated design of ML accelerators, and methodologies to improve HLS results;

- Chapter 3 describes high-level optimizations that can be effectively applied before HLS to improve developer productivity, portability between different backends, and performance of the generated accelerators;

- Chapter 4 presents an end-to-end flow based on the proposed multi-level approach that can automatically generate systems composed of multiple accelerators, coupling high-level optimizations with low-level synthesis methodologies for the generation of dataflow accelerators;

- Chapter 5 describes the introduction of an MLIR domain-specific abstraction to support the simulation of spiking neural networks and the generation of correspondent neuromorphic accelerators;

- Chapter 6 presents and discusses experimental results obtained with the proposed multi-level approach;
- Chapter 7 wraps up the thesis and outlines future research directions.

CHAPTER 2

State of the art

PREVIOUS works have proposed methodologies and research directions that are related to the topics covered in this thesis. This chapter presents the most relevant ones, divided into three macro-categories: techniques to overcome the limitation of existing HLS tools, automated hardware design flows for ML, and methodologies to improve the HLS process and quality of results.

2.1 Overview

The proposed multi-level approach to HLS draws inspiration from several previous works, and it is related to many recent efforts to solve similar problems. Figure 2.1 groups them into three areas of active research, which correspond to the three following sections of this chapter. The first group (Section 2.2) contains works that take existing HLS tools as a baseline and overcome their limitations by carefully restructuring input specifications. The second group (Section 2.3) contains research on specific optimizations that improve the quality of HLS results, and new techniques that aim to replace existing HLS tools. The third group (Section 2.4) focuses on the automated generation of accelerators for ML, which is often based on HLS. This subdivision is useful to compare and discuss related work, but often the scope of a paper will span across more than one group, so it will be cited and discussed in multiple sections according to its contribution to the corresponding area.

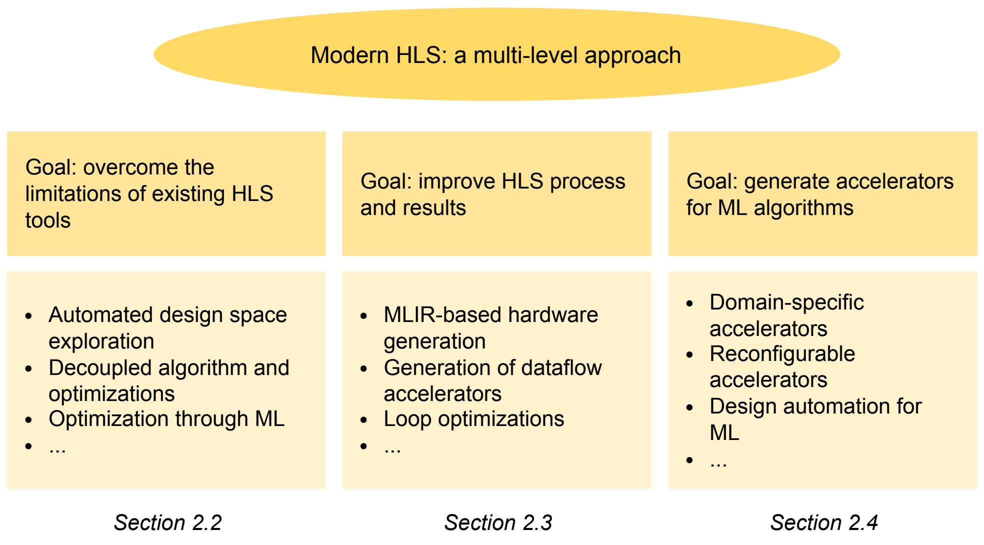

Modern HLS: a multi-level approach
Goal: overcome the limitations of existing HLS tools
Goal: improve HLS process and results
Goal: generate accelerators for ML algorithms
Automated design space exploration
Decoupled algorithm and optimizations
Optimization through ML
...
MLIR-based hardware generation
Generation of dataflow accelerators
Loop optimizations
...
Domain-specific accelerators
Reconfigurable accelerators
Design automation for ML
...
Section 2.2
Section 2.3
Section 2.4

Figure 2.1: *Areas of research covered in Chapter 2.*

2.2 Challenges for productive HLS

The special session "New perspectives in High-Level Synthesis" at the 2022 Design Automation Conference gathered leading experts in HLS research to discuss challenges and opportunities introduced by innovations in technologies and applications. Decades of research produced HLS tools that are very efficient at implementing FPGA/ASIC accelerators starting from general-purpose programming languages; however, obtaining a design that meets specific latency, throughput, area, or power consumption requirements still depends on the ability of users to understand the effect of optimization directives. A common theme in the special session was therefore how to improve productivity and automate design space exploration (DSE), especially to the benefit of domain scientists working in Python-based high-level frameworks.

HeteroCL [21, 22] tackles this problem by decoupling the definition of the algorithm from compute, data type, and memory optimizations, following the separation of concerns paradigm introduced by Halide [23] for image processing. An extension to Halide already existed to design FPGA-friendly schedules and generate corresponding HLS code [24]; however, HeteroCL is a more general solution that supports more optimization opportunities and a broader range of applications. Specifically, HeteroCL defines a domain-specific language (DSL) based on Python to partition an algorithm into host code to be executed on a general-purpose processor, and kernel code to be synthesized as an FPGA accelerator. The programming model based on decoupled algorithm and hardware customization raises the level of abstraction and increases productivity, as annotated HLS code for the kernel is automatically generated by a compiler according to directives specified in Python by the user. HeteroCL also includes templates for the generation of systolic arrays and dataflow accelerators. Other works such as PyLog [25], or Hot&Spicy [26] also produce annotated code for HLS tools starting from Python code; none of them, however, provides an interface to popular data science and machine learning frameworks.

ScaleHLS [27, 28] exploits MLIR to analyze and transform input code from C or PyTorch, generating annotated code for Vivado HLS (a slightly old version of the Xilinx HLS tool which does not apply any automated optimization). The multiple levels of abstraction provided by existing MLIR dialects allow ScaleHLS to reason about graph-level, loop-level, and directive-level optimizations; a custom dialect helps the translation into C++ with pragmas. A quality of results (QoR) estimator and a DSE engine automatically identify the best combination of optimizations following user-defined constraints, without requiring long simulation or synthesis runs to evaluate the effect of changes in the optimization directives. The portability of code generated by ScaleHLS is limited: as happens with every tool that relies on pragma annotations, the optimizations it applies are only effective if the backend HLS tool recognizes them. Moreover, analyzing high-level IRs or early HLS estimates may lead to an underestimation of resource consumption, as was shown in several experiments where designs produced by ScaleHLS could not conclude place and route because they required more resources than the ones available in the target FPGA.

High-level design choices significantly affect QoR of the generated accelerators, and their impact can first be seen at the end of the HLS process, which takes minutes to hours to complete. Since the design space to explore is considerably large, it can be beneficial to implement fast prediction methods rather than exhaustively evaluate every configuration. Multiple solutions based on different ML methods have been proposed in the past, e.g., [29–31]; GNNs are used in [32–34] because input programs can be easily represented as graphs. Supervised learning methods rely on the availability of large training sets, which in this case means sampling the design space and synthesizing a subset of the possible configuration for each kernel. [35] applies transfer learning to exploit knowledge about a design and predict the QoR of a different one, reducing the effort of creating a new training set for each input kernel. Improving upon an existing GNN model for DSE, [36] addresses the same issue through advanced meta-learning techniques that reduce the accuracy drop on new inputs.

A radically different approach to improve productivity is the multi-level, compiler-based design flow proposed by this thesis and implemented in SODA [13, 14]. Users of SODA do not need to rewrite their applications in a new DSL, nor do they have to augment them with optimization directives: thanks to MLIR, SODA has a direct interface with high-level programming frameworks, and it exposes optimizations as compiler passes providing a convenient entry point for DSE [15]. The SODA Synthesizer is described in more detail in Chapter 4.

2.3 Improvements to the HLS process and results

While the approaches in Section 2.2 mostly use existing HLS tools as "black boxes", exploiting as much as possible the optimization opportunities they expose, this section describes proposals to innovate the HLS process itself in order to obtain better QoR and increase productivity for developers. Research in this field is sometimes hindered by the proprietary nature of established HLS tools [37] (Bambu [16] is a notable exception). However, there is a trend toward the democratization of hardware design, as attested for example by the open-source release of the Xilinx Vitis HLS frontend [38] or by the OpenROAD project for ASIC synthesis [39].

The CIRCT project [40] intends to use MLIR to build a new generation of interoperable tools and compilers for hardware design, starting from the definition of circuit-level IRs and working upwards to higher levels of abstraction (e.g., dataflow models or finite state machines). Part of the project is dedicated to HLS [41], particularly to the implementation of static and dynamic scheduling through MLIR and CIRCT dialects. CIRCT could be an essential building block for future industrial and academic design flows. However, to obtain the results described in this thesis it was preferable to integrate mature HLS backends with optimized synthesis algorithms and resource libraries supported by decades of research.

Chapters 3 and 4 will describe methods to improve the HLS process and results through high-level loop optimizations and the generation of dataflow accelerators; Sections 2.3.1 and 2.3.2 collect related work on these two specific topics.

2.3.1 Loop optimizations

Loop transformations are essential to improve the QoR of accelerators generated by HLS since a single loop iteration usually does not contain enough optimization potential. Inter- and intra-iteration dependencies determine the amount of parallelism that can be extracted from nested loops in the input code, and the user can use loop transformations to choose a trade-off between area and performance according to application requirements. Loop unrolling replicates the body of the innermost loop multiple times, exposing opportunities to execute instructions in parallel; this is the most straightforward option, and it is available in generic compilers, but it results in large area overheads. Loop pipelining, instead, aims at overlapping the execution of different iterations by issuing a new iteration before the previous one has finished executing. This is a more complicated transformation, which involves a scheduling process.

The polyhedral model provides unique opportunities to improve parallelization, pipelining, and memory accesses, which are fundamental targets in mapping software programs to the spatial parallelism of FPGAs and ASICs. Polyhedral compilers are particularly suited to workloads containing deeply nested loops and arrays, such as linear algebra solvers for high-performance scientific simulation or tensor-based ML algorithms. Typical targets for polyhedral optimization techniques include general-purpose processors with vector units and large multi-level caches, GPUs, and spatial accelerators. Previous works that applied the polyhedral model to HLS include [42–44], which run C/C++ inputs through polyhedral optimizers and write back restructured C/C++ annotated with HLS directives. Even if these approaches show great potential to improve the performance of generated accelerators, they are hard to combine with other optimizations in a complete design flow, and the code they generate is not portable across HLS tools. More recently, POLSCA [45] proposed a modular approach exploiting the MLIR *affine* dialect and the Vitis HLS LLVM frontend to bridge the gap between polyhedral tools and HLS.

Loop pipelining is an essential, nontrivial optimization for HLS which has been explored from several different perspectives. Irregular loops with variable bounds and complex dependencies are especially challenging and require dedicated solutions: for example, [46] applies speculative loop pipelining for HLS, [47] supports loops with non-constant dependencies, [48] and [49] exploit polyhedral frameworks to implement dynamic loop pipelining. Polyhedral analysis in this case enables the generation of specialized logic that, at runtime, verifies whether it is safe to run all loop iterations in the pipeline or interrupts its execution to resolve memory conflicts.

The multi-level approach described in this thesis uses existing MLIR transformations and implements new ones when needed. For what concerns loop pipelining, the MLIR *structured control flow* (SCF) dialect provides an experimental pass that generates pipelined loops according to a schedule manually encoded by the developer, and the CIRCT *staticlogic* dialect implements the pipelined loop at a lower level of abstraction, explicitly dividing the instructions of a loop into pipeline stages. As previously mentioned, CIRCT has been considered to be at a very early stage for integration in the proposed design flow, while the SCF loop pipelining pass was considered too restrictive and replaced by a custom implementation (this will be discussed in Chapter 3).

2.3.2 Generation of dataflow accelerators

HLS tools typically follow the finite state machine with datapath (FSMD) paradigm to generate hardware designs, where each state determines which datapath components are activated. The FSMD model is particularly suited for extracting instruction-level parallelism, but it is based on a centralized controller that does not scale well to large designs with parallel execution flows. Previous research proposed to decompose and distribute the FSM controller, restructuring it in a hierarchical way [50], but this is not an efficient solution to manage the concurrent execution of independent units. In such conditions, the complexity of a centralized, statically scheduled FSM controller grows exponentially, leading to significant area and performance overheads [51]. A solution to this issue is to implement the accelerator following the dataflow paradigm, where a distributed controller activates hardware components (which can be functional units or more complex modules, depending on the chosen granularity) as soon as its inputs are available.

The Bluespec compiler [52] implements an event-driven execution paradigm based on rules and atomic transactions starting from BSV, a language close to behavioral HDL (albeit more abstract than Verilog or VHDL). Dynamatic [53] is an HLS tool that generates dynamically scheduled designs starting from C code using the dataflow paradigm; it does not support resource sharing and abstracts memory by decoupling it from the accelerator through a single load/store queue, thus not taking advantage of memory-level parallelism. CIRCT also has a *handshake* dialect to implement dataflow circuits inspired by Dynamatic. All these tools mainly focus on supporting dataflow at the instruction level, with functional units exchanging signals that drive the execution.

A dynamically scheduled design based on the dataflow model results in simpler accelerators exploiting parallelism across basic block boundaries; however, FSMDs provide higher QoR (both in performance and area) extracting instruction-level parallelism from inside a function or a basic block: for this reason, it is sometimes beneficial to combine the two models. Dynamatic has been extended to couple dynamic with static scheduling in [54], supporting resource reuse and a simple memory abstraction, but still focusing on dataflow at the instruction level rather than supporting coarser-grained parallel tasks. Several other research projects can generate accelerators that exploit coarse-grained parallelism by combining dataflow concepts with FSMDs: for example, the Spatial DSL [55] allows marking modules as dataflow or FSMD at different levels of the hierarchy. The Xilinx HLS tools support dataflow pipelining mechanisms across functions or loops (marked by the user with a specific pragma), provided that they have a similar initiation interval [56].

The SODA Synthesizer supports the generation of dynamically scheduled accelerators by integrating the methodology presented in [57], as will be described in more detail in Chapter 4. In its original version, the methodology allowed to synthesize parallel tasks annotated by users with OpenMP-like pragmas, while the proposed multi-level approach provides an entirely automated flow from MLIR specification to hardware that does not require additional input from the user.

2.4 Hardware acceleration for machine learning

The multi-level, MLIR-based design flow proposed by this thesis can be especially useful for domain scientists seeking to accelerate ML inference. In fact, ML training can be efficiently run offline on clusters of GPUs, as throughput is critical and floating-point data types with high precision are mandatory. Inference instead can be run in a variety of different settings with different application requirements (including more and more frequently edge devices [58]), where the most critical metric may be latency or power consumption; for these cases, specialized accelerators or reconfigurable devices are often the best solutions, and a variety of different architectures have been proposed based on ASICs [59] or FPGAs [60].

Commercial solutions to accelerate ML algorithms include specialized functional units in programmable devices, such as the Tensor Cores in NVIDIA GPUs [61] or the AI engines in Xilinx platforms [62], and entire chips based on tensor processing (e.g., the Google TPU [17]). Companies such as SambaNova [63], Graphcore [64], and Cerebras [65] propose architectures based on the dataflow paradigm, with varying degrees of generality in their processing elements. Microsoft's project Brainwave [66] accelerates ML algorithms on an FPGA-based platform supporting specialized numeric formats to reduce resource utilization and increase efficiency.

Considerable attention has been dedicated to the acceleration of DNNs [67, 68]; however, it would also be beneficial to design accelerators that can support multiple classes of ML algorithms. Efforts in this direction include PuDianNao [69], supporting multiple ML methods for both training and inference, and SpiNNaker [70] for Spiking Neural Network (SNN) simulation. IBM's TrueNorth [71] and Intel's Lohihi [72] are neuromorphic, brain-inspired chips built to run SNN models; SyncNN [73] leverages HLS to implement digital SNN accelerators. TABLA [74] provides a programming model and pre-optimized templates to design FPGA accelerators for different ML algorithms starting from their gradient descent function. DNNBuilder [75] is also based on pre-built register transfer level (RTL) components representing DNN layers, providing a design flow that configures them to compose FPGA accelerators and a dedicated DSE flow to identify efficient configurations [76]. SIGMA [77] specializes a systolic array architecture to efficiently support sparse and irregular computation through reconfigurable interconnect; MAERI [78] also uses reconfigurable interconnects to support multiple ML operators on the same accelerator.

Other works focus on the compilation and mapping of high-level ML models to hardware accelerators or architectural templates. VeriGOOD-ML [79] uses the PolyMath compiler [80] to map ML models in the ONNX format to three different architecture templates designed for different types of models: the TABLA template for generic ML algorithms, an architectural template for DNNs, and one for small, extremely specialized operators. GEMMINI [81] offloads operations from specific layers of ONNX models to a systolic array connected to a RISC-V core, after building the systolic array itself starting from a parametrized generator in Chisel. TVM's VTA architecture [82] is a configurable FPGA co-processor for matrix multiplication; the TVM high-level framework compiles ML models into instructions for VTA.

"Classic" HLS-based approaches translate high-level ML models into a form that can be ingested by existing HLS tools (typically, C/C++ code with optimization directives). ScaleHLS [27] aims at facilitating and optimizing HLS through high-level transformations implemented in MLIR, generating annotated C code for Vivado HLS for ML models translated from PyTorch. LeFlow [83] takes a TensorFlow graph as input, transforms it into an LLVM IR through the XLA compiler, and synthesizes it with LegUp [84]. The accelerators generated by LeFlow work under the assumption that both weights and inputs of the model are available on-chip; this strongly limits the possibility of using it on real-world networks with millions of parameters. Moreover, XLA is built to generate code for CPUs and GPUs, so the final LLVM IR is not optimized for HLS. The semantic information that gets lost in the process hinders the application of meaningful transformations. Other tools are focused on FPGA acceleration of CNN models [85], so they benefit from the regular computation patterns, redundancy, and reduced dynamic range typical of convolutional layers. Most of them support only fixed-point data types, arguing that the additional accuracy of floating-point calculation does not compensate for the resource utilization overhead; some are based on HLS, while others provide hand-written RTL kernels.

Hls4ml [19] is one of the most popular tools following the "classic" HLS approach: it translates DNN models from high-level ML frameworks into annotated C++ code for HLS, offering a complete and user-friendly design process that has been enthusiastically adopted in physics research. Hls4ml is under active development, but its core functionality remains tied to a library of C++ components representing standard DNN operators. The library is optimized for Vivado HLS, it contains parametric data types to allow post-training quantization, and it assumes that the accelerator will be composed of a sequence of operators representing the different network layers. Hls4ml was initially developed for a specific use case where a multi-layer perceptron was used to perform a classification task within a high-energy physics experiment; optimizations introduced to comply with the strict latency constraints imposed by the experiment included quantization, pruning, and frequent loop unrolling. Network compression was exploited by enforcing sparsity in the training phase and relying on Vivado HLS to eliminate operations with zero weights. Following studies presented additions to the tool to support quantization-aware training and automatic heterogeneous quantization [86], binary and ternary networks [87], and CNNs [88].

FINN [20] is a similar framework with a specific focus on quantized DNNs with extremely low bit-widths (less than 4 bits fixed-point), and it contains a PyTorch library for quantization-aware training to ensure reasonable accuracy even with such small data types. The user can choose between two accelerator templates with different resource utilization and throughput characteristics: a feed-forward streaming dataflow pipeline (for small networks that can afford to store all weights on-chip) or a computation engine that offloads part of the data to external memory (for larger networks or smaller FPGAs). FINN supports the whole design and implementation flow, from the definition and training of the network to the deployment of the accelerator on cloud FPGA instances, following a network/hardware co-design approach that is not always applicable to a generic pre-trained DNN. Underneath the surface, FINN is based on a library of parametric C++ components optimized for Vivado HLS.

As discussed in Section 1.2, the specific implementation choices of tools like hls4ml and FINN are not guaranteed to provide an optimized result in a general case, and implementing a library of templates limits portability across different HLS backends. The library implementation of activation functions for hls4ml contains a practical example of features that are only applicable when synthesizing hardware with Vivado HLS: non-trivial activation functions are implemented as tables of constant values representing the function output for a given range of inputs; Vivado HLS recognizes a specific C++ pattern at compile-time so that the final RTL design only contains a pre-computed lookup table stored in on-chip memories (BRAMs). Other HLS tools may not be able to interpret the C++ pattern as the construction of a constant table, wasting resources and increasing the overall latency to implement the actual mathematical functions and control logic contained in the code. Another example of extreme specialization can be found in the implementation of the softmax layer, where a default fixed-point type was empirically selected to fill the available BRAM size on specific Xilinx FPGA models. With such an extreme degree of specialization, every new backend tool requires an entirely new implementation of the template library.

This thesis does not propose yet another accelerator design or architectural template, where different applications would achieve different performances depending on how well they exploit the available computational resources: instead, the compiler-based approach bringing together MLIR and HLS represents a new methodology to design and implement an FPGA or ASIC accelerator starting from a high-level description of a specific ML algorithm. Compared to other tools that provide a bridge between high-level programming frameworks and hardware generation, the proposed method is considerably more flexible in the high-level frameworks and backend HLS tools it can support. Optimizations are applied at appropriate levels of abstraction without generating intermediate C/C++ code, and the modular compilation pipeline can easily be adapted to the needs of new classes of input models.

CHAPTER 3

High-level optimizations

APPLYING optimizations to high-level specifications before HLS can significantly contribute to developers' productivity, portability across HLS tools, and performance. In this chapter, a high-level implementation of loop pipelining is introduced leveraging a specialized MLIR dialect and integrated into a frontend optimizer for HLS.

This chapter contains material from:

S. Curzel, S. Jovic, M. Fiorito, A. Tumeo, and F. Ferrandi, "Higher-Level Synthesis: experimenting with MLIR polyhedral representations for accelerator design," in *12th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2022, pp. 1–10;

S. Curzel, S. Jovic, M. Fiorito, A. Tumeo, and F. Ferrandi, "MLIR loop optimizations for High-Level Synthesis: a case study," in *31st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2022, pp. 1–2;

N. Bohm Agostini, S. Curzel, V. Amatya, C. Tan, M. Minutoli, V. G. Castellana, J. Manzano, D. Kaeli, and A. Tumeo, "An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022, pp. 1–9.

3.1 Overview

HLS tools are effectively compilers, as they translate a programming language into another one at a lower level of abstraction. In their frontend, they benefit from the same compiler optimizations that identify instruction, memory, and data parallelism in software for general-purpose processors; in the backend they work on a low-level IR close to the actual hardware description, and they perform allocation of resources, scheduling of instructions, binding of instructions to resources, finally generating HDL code for the required FPGA or ASIC target (Figure 3.1). Optimizations are usually triggered by users through annotations or configuration options; the HLS tool then applies the corresponding transformations to its internal IR.

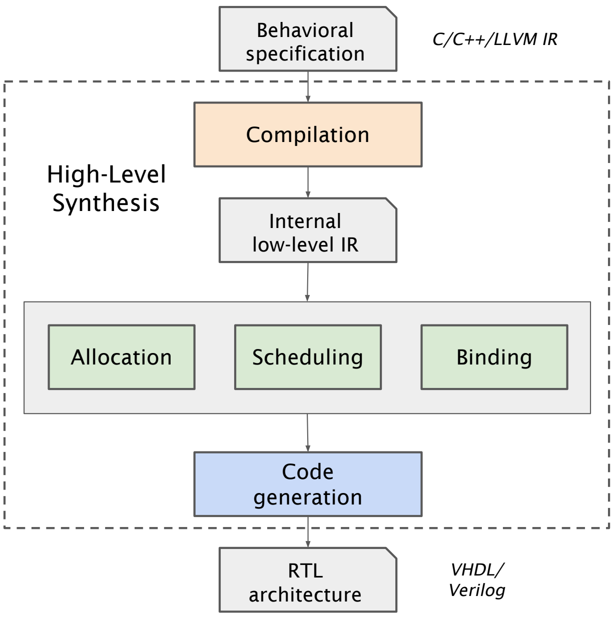


Figure 3.1: *Steps in the HLS process.*

To introduce new optimizations, researchers and developers need access to the code (which might already be an obstacle, given the proprietary nature of most HLS tools) and knowledge about the hardware generation process. The alternative is resorting to modifications on the input code, limiting the exploration of transformations to what can be expressed through C/C++ code augmented by pragmas, as is typically done in research works that apply polyhedral optimizations to HLS (e.g., [42, 49]). Such an approach can produce good results, but it has limited flexibility, and it risks loss of semantic information as the input specification is translated multiple times across different formats for different tools.

What this thesis proposes, instead, is to exploit recent advances in compiler technology to introduce and apply optimizations outside of the HLS tool in a modular way, taking advantage of state-of-the-art HLS tools that support LLVM IRs as input. Specifically, this chapter presents the advantages of using MLIR to build a high-level compiler frontend for HLS where optimizations can be added at an appropriate level of abstraction through dedicated dialects, and without relying on tool-specific annotations. An MLIR implementation of loop pipelining is used as a case study since it has a significant effect on HLS QoR and it is not as straightforward as loop unrolling (which is already available in MLIR, in standard compilers, and in HLS frontends).

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. The MLIR affine dialect provides structures and methods to analyze and transform loops (in fact, it was initially introduced to support polyhedral optimizations for ML frameworks), and the higher level of abstraction allows to identify more complex dependencies than what is possible on an LLVM IR or low-level HLS IR. Two custom MLIR passes were implemented in the affine dialect following the principles of software loop pipelining [91], and no modification was necessary to low-level hardware generation steps inside an HLS tool. These factors contribute to an improvement in the productivity of developers: implementing loop pipelining (or another nontrivial optimization) within an HLS tool that does not already support it would typically have required extensive modifications to the scheduling and code generation steps, and it would have had access to a less powerful dependency analysis.

Another advantage of the proposed approach is that it generates portable optimized code, instead of exploiting directives and code patterns that trigger optimizations for a specific HLS tool. Chapter 6 starts by evaluating the impact of MLIR-based loop pipelining on an open-source HLS tool, Bambu [16], but the transformed code does not contain anything specific to Bambu, and it can be synthesized by any other tool accepting MLIR or LLVM code as input (including recent versions of Xilinx Vitis HLS [38]). Finally, MLIR passes are modular and they can be easily combined in a compilation pipeline that acts as a frontend optimizer for HLS tools: this MLIR-based loop pipelining implementation has been integrated into one such tool, SODA-OPT [15], which follows the proposed multi-level approach to HLS. Experiments in Chapter 6 will demonstrate the portability across HLS tools and increased performance provided by MLIR-based loop pipelining, both in isolation and combined with other high-level optimizations.

In summary, this chapter presents the following contributions:

- a high-level implementation of loop pipelining for HLS based on the MLIR affine dialect;

- a demonstration of the improvements in productivity, portability, and performance deriving from the use of MLIR;

- an optimization flow that applies selected compiler transformations before HLS to improve the performance of the generated accelerators.

Section 3.2 introduces the main techniques and tools used in the rest of the chapter, Section 3.3 presents the proposed design flow and Section 3.4 dives deeper into implementation details, Section 3.5 presents the integration of the loop pipelining pass in a frontend optimizer for HLS, Section 3.6 draws conclusions and outlines future research directions.

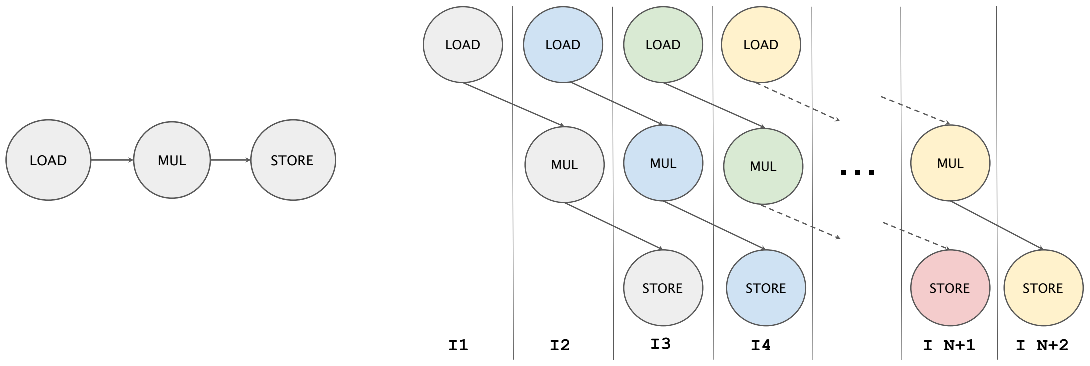
3.2 Techniques and tools involved

This section describes techniques and tools that are used throughout the chapter.

3.2.1 Loop pipelining

Loop optimizations have been widely studied both in software and hardware research, and they are critical to improve the performance of hardware accelerators generated through HLS. Loop pipelining overlaps the execution of multiple iterations and it is useful, for example, when there are not enough hardware resources to accommodate an unrolled loop. The technique has been successfully used in compiler infrastructures for decades [91], and it generally consists of two steps: loop scheduling and code generation. Depending on the available computation and memory resources, their latency, and the presence of inter-iteration dependencies, a pipelined loop can issue the execution of a new iteration at every clock cycle (II=1).

A simple example useful to illustrate the procedure in more detail is a for loop that reads values from an array, multiplies them with a constant, and writes them into another array. A single iteration of the loop contains three operations: load, multiply, and store (Figure 3.2a); clearly, the three operations depend on each other and cannot be executed in parallel. Loop pipelining produces a new loop such as the one in Figure 3.2b, where each column represents one iteration of the new loop, and operations originating from the same iteration of the original loop are highlighted with the same color.



(a) *Single loop iteration.* **(b)** *Pipelined loop.*

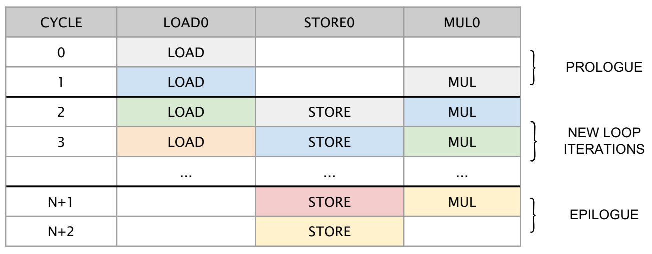
Figure 3.2: *Example of a pipelined loop.*

Each iteration of the pipelined loop contains operations from different iterations of the original loop: as these operations do not depend on each other, they can be executed in parallel without constraints. By overlapping original iterations, loop pipelining thus eliminates obstacles to parallelization. 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 3.2b, iterations I 1 and I 2 belong to the loop prologue, I N+1 and I N+2 represent the epilogue, while the actual new loop starts from I3. Assuming that all functional units execute in one clock cycle, the achieved II in this simple example is equal to 1.

3.2.2 Scheduling

The example loop of Figure 3.2 cannot be pipelined if there are not enough memory elements available to run two operations (one load and one store) simultaneously. Before generating the pipelined loop, a scheduling step analyzes available hardware resources and dependencies across operations to produce a schedule, i.e., a list of operations assigned to a clock cycle and to a resource, and compute the II. If one load and one store unit are available for the example loop, and all functional units have a delay of one cycle, the scheduler will correctly produce a schedule such as the one shown in Figure 3.3, which contains the necessary information to pipeline the loop with II=1.

Achieving an optimal or close to optimal schedule is an NP-complete problem; rather than building a new scheduling tool, the implementation described in this chapter uses the HatSchet library [92], an open-source scheduling tool for HLS that offers various algorithms and heuristics for the construction of schedules. HatSchet requires a



CYCLE	LOAD0	STORE0	MUL0
0	LOAD		
1	LOAD		MUL
2	LOAD	STORE	MUL
3	LOAD	STORE	MUL
	...	...	...
N+1		STORE	MUL
N+2		STORE	

PROLOGUE
NEW LOOP ITERATIONS
EPILOGUE

Figure 3.3: *Pipelined loop schedule.*

data flow graph (DFG) of the loop body and resource availability information as inputs, and it produces a schedule in a textual format. The availability of different algorithms enables to trade scheduling time for quality.

3.2.3 MLIR

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 specialized abstractions (dialects) to solve specialized optimization problems. MLIR was conceived initially to be applied within ML frameworks; several ML tools offer an interface to MLIR dialects, and it is expected that more and more programming frameworks will do the same in the near future.

MLIR defines an IR in static single-assignment (SSA) form consisting of MLIR operations, which dialects can extend by representing 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 exposes entry points for analysis and transformation passes which traverse the IR to either collect useful information or modify it.

Many dialects in the main MLIR repository operate at higher levels of abstraction with respect to C/C++, and to the instruction-level IRs of HLS tools; moreover, it is possible to combine different dialects in the same representation, opening the way to the integration of novel compilation passes and optimizations that operate on specific parts of the program. 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. 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. The proposed implementation of loop pipelining is based on the affine dialect, but other dialects that can be lowered to affine can exploit it, and they may provide future opportunities for optimization. Many ML frameworks implement custom MLIR dialects with interfaces to high-level abstractions such as linear algebra (*linalg*) or tensor operations (*tosa*); the proposed approach can thus be applied to ML algorithms, and to any other application written in a programming environment or DSL with an interface to MLIR dialects.

The SCF dialect provides a different representation of for loops, and it also offers a loop pipelining pass; however, it only implements the code generation phase, expecting developers to manually provide a schedule for the loop operations. As will be explained in more detail in the following sections, to build an efficient schedule it is essential to compute the dependencies between operations, and the affine dialect provides information about memory dependencies (affine load and store operations) that are lost when lowering to the SCF dialect (memref load and store operations). Applying a schedule containing affine operations to a set of SCF operations would not be appropriate, as there can be mismatches between the original and the lowered operations (e.g., an affine apply operation is lowered to more than one SCF operation), so it was eventually decided to implement both dependencies extraction and code generation at the affine level.

3.2.4 HLS tools

Two different HLS tools were used to validate the proposed approach: Bambu and Vitis HLS. Bambu is a state-of-the-art open-source HLS tool compatible with both C/C++ and LLVM IR inputs; it can perform (some) loop optimizations in the frontend (e.g., unrolling) but 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 absence of user-defined directives, if a loop in the code is considered too expensive to be unrolled Vitis HLS tries to pipeline it with an II of 1, progressively relaxing the constraint if necessary [93].

Bambu was the first target of the 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 LLVM frontend for Vitis HLS [38], which can be bypassed to use an LLVM IR as input to the (closed-source) synthesis backend. This way of using Vitis HLS is rather experimental, with little documentation and support, but it allowed to verify whether high-level transformations can have a similar effect on different HLS tools. By synthesizing annotated C code with Vitis HLS it is also possible to assess whether MLIR-based loop pipelining has a similar or better effect on the generated accelerator performance with respect to a low-level implementation of the same optimization triggered through pragma annotations.

3.3 Proposed design flow

The goal of the proposed approach is to leverage high-level code optimizations to provide a pre-transformed input description to HLS, without binding it to the requirements of a specific HLS tool (most notably in terms of pragma annotations). This section presents an overview of the proposed design flow, leaving implementation details to the next section; Figure 3.4 shows the main steps and tools involved.

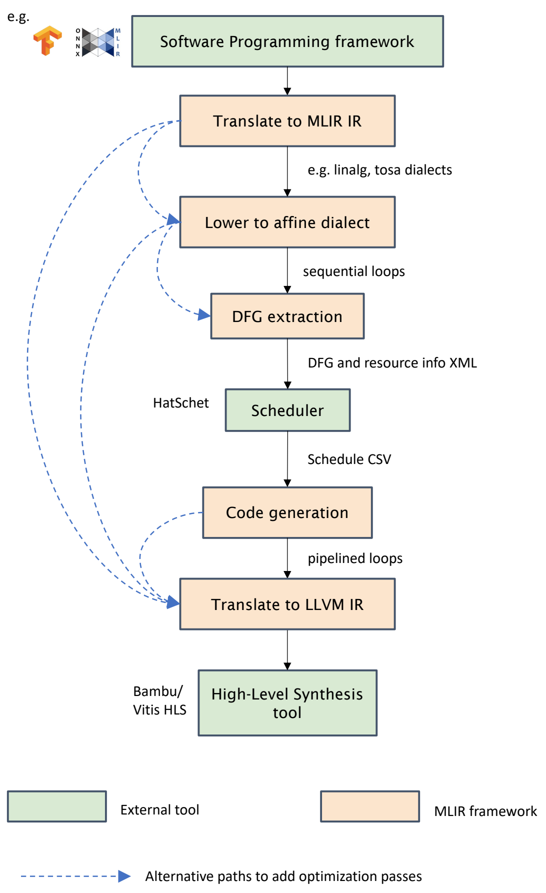


Figure 3.4: *Overview of the proposed optimization flow.*

The input code may originate from any high-level programming framework with a translation into MLIR (e.g., TensorFlow [7] or ONNX-MLIR [94]). After lowering it to the affine dialect, the code contains one or more sequential for loops, which are analyzed by the first newly introduced pass to extract a DFG representing the operations in the loop body. The DFG is passed to an external scheduler (HatSchet), together with information on the available hardware resources. When targeting an FPGA, the resource constraints may be set iteratively to achieve a specific trade-off between performance and area consumption, also considering the rest of the design besides the loop(s) of interest; in simple cases where the number of instructions in the loop body is limited, it is also acceptable not to impose any resource constraint at this stage. The second part of the new implementation is a pass that rewrites the input code using the schedule generated by HatSchet to produce the 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 proposed design flow represents an alternative to delegating scheduling and pipelining to the HLS tool itself: this is the standard programming model of Vitis HLS, where optimizations are controlled by the user through pragmas in the input C code, or automatically triggered in the backend. Bringing loop pipelining (and other optimizations) outside the scope of the HLS tool has significant advantages: for example, the developer can immediately verify the application of the optimization 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 (when this is possible, as most HLS tools are closed-source). Finally, MLIR is built to allow integration and reuse of different compiler passes: this means that loop pipelining may be combined with other techniques to create pre-optimized inputs to the HLS tool that result in even more efficient hardware accelerators.

3.4 Implementation of high-level loop pipelining

Two custom MLIR passes were implemented to realize the proposed design flow: one extracts a DFG from the MLIR loop body, and another one generates the pipelined loop code according to the schedule produced by HatSchet. In the following, the loop described in Section 3.3 will be used as example; Figure 3.5a shows its MLIR implementation (affine dialect for loops and memory operations, *arith* dialect for basic arithmetic operations).

```
func @example(%arg0: memref<1000xi32>) {
  affine.for %arg1 = 0 to 1000 {
    %0 = affine.load %arg0[%arg1]
    %1 = arith.muli %0, %0
    affine.store %1, %arg0[%arg1]
  }
  return
}
```

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

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

(b) *HatSchet schedule.*

```
#map = affine_map<(d0) -> (d0 - 2)>
func @example(%arg0: memref<1000xi32>,
              %arg1: memref<1000xi32>) {
  %c0 = arith.constant 0 : index
  %0 = affine.load %arg0[%c0] : memref<1000xi32>
  %c1 = arith.constant 1 : index
  %1 = affine.load %arg0[%c1] : memref<1000xi32>
  %2 = arith.muli %0, %0 : i32
  %3:2 = affine.for %arg2 = 2 to 1000
           iter_args(%arg3 = %1, %arg4 = %2) -> (i32, i32) {
    %5 = affine.load %arg0[%arg2] : memref<1000xi32>
    %6 = arith.muli %arg3, %arg3 : i32
    %7 = affine.apply #map(%arg2)
    affine.store %arg4, %arg1[%7] : memref<1000xi32>
    affine.yield %5, %6 : i32, i32
  }
  %4 = arith.muli %3#0, %3#0 : i32
  %c998 = arith.constant 998 : index
  affine.store %3#1, %arg1[%c998] : memref<1000xi32>
  %c999 = arith.constant 999 : index
  affine.store %4, %arg1[%c999] : memref<1000xi32>
  return
}
```

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

Figure 3.5: *Code generation for high-level loop pipelining in MLIR.*

3.4.1 Data flow graph extraction pass

A first MLIR pass was implemented to visit all the operations in the loop body and extract their dependencies; information retrieved by the analysis is used to build a DFG that HatSchet is able to schedule. Nodes in the graph represent operations; edges represent dependencies between operations (precedence and data dependencies). Precedence dependency refers to an operation that uses the result of another operation in the code; data dependency exists 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 as the result of an operation in an iteration cannot be used in a different iteration without passing through memory. The pass encodes all this information in an XML file format parsable by HatSchet, and a second XML file is passed to the scheduler representing the type and amount of available functional units.

The extraction of nodes and precedence edges is performed by visiting all operations in the loop. Data dependence analysis is simplified by an existing MLIR affine method, `checkMemRefAccessDependence`, which analyzes a pair of memory operations and evaluates whether a dependency exists (the distance can also be deduced from its output). Existing affine constructs thus simplified the implementation of the DFG extraction pass, confirming the hypothesis that the MLIR dialect-based approach provides a convenient framework for the quick introduction of new optimizations.

Algorithm 1

```
 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.$get\_cycles(start : start\_cycle, end : start\_cycle + II)$
 8:             iteration.$schedule\_cycles(cycles)$
 9:             current_prologue_iteration = current_prologue_iteration - 1
10:             start_cycle = start_cycle + II
11:         end while
12:         if iteration.$size()$ < originalIterationSize then
13:             prologue.$add(iteration)$
14:         end if
15:     until iteration.size == originalIterationSize
16: end function
```

Figure 3.6: *Prologue extraction algorithm.*

3.4.2 Code generation pass

The second pass that was implemented loads the HatSchet schedule from its textual format (Figure 3.5b) into a suitable data structure and uses it to generate code for the pipelined loop. The following paragraphs highlight all the steps that are needed to transform the original loop of Figure 3.5a into the scheduled loop of Figure 3.5c.

Prologue and epilogue extraction. HatSchet provides only the new loop iteration schedule, without prologue and epilogue, so the MLIR code generation pass needs to generate additional instructions 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 blocks of operations that can be issued in parallel (for example, columns I1 and I2 in Figure 3.2b).

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. 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 of the algorithm is satisfied once it generates an iteration containing all operations that are in the schedule (this iteration is not included in the prologue).

Looking at Figure 3.2b, the loop has 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, and the multiplication to continue the previous iteration in I2. The algorithm stops when it generates I3 and sees that it contains the same operations of the scheduled iteration, so it will be discarded.

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

Operations mapping. During code generation, original loop operations are cloned, and their operands are updated to reference new operation results. MLIR provides 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. A separate map is built for each original iteration to ensure that the correct operation dependencies are maintained, and a key-value pair is recorded each time a new operation is generated.

In the example code of Figure 3.5c 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 new operations are created in order, the cloning map always contains 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, a new variable is calculated by 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 the 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, it needs to be replaced with an additional constant: its value is the sum of the original iteration number (zero) with the loop lower bound (zero). The second load

Algorithm 2

```
function GENERATE_PROLOGUE($prologue\_iterations, operand\_maps$)
    for iteration : prologue_iterations do
        for cycle : iteration do
            for operation : cycle do
                for operand : operation.operands do
                    if operand == original_index_variable then
                        constant_operation =
                            $generate\_constant\_operation(value : original\_iteration)$
                        operand_maps[original_iteration].$insert(operand,$
                            $constant\_operation.result)$
                    end if
                    new_op = operation.$clone(maps[original\_iteration])$
                    maps[original_iteration].$insert(operation.results, new\_op.results)$
                end for
            end for
        end for
    end for
end function
```

Figure 3.7: *Prologue generation algorithm*

is cloned and moved in the same way; its constant will have a value of one since the operation comes from the first original iteration. 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 recorded in the map so that all operations that used it in the original loop are redirected to its new value.

New loop generation. In a similar way to what happens during prologue generation, each operation in the new loop is created by cloning the original loop operation and mapping its operands (Algorithm 3). 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 the loop index of an iteration started in the prologue, the variable will be adjusted by a map to refer to the correct value. For example, operation %7 in Figure 3.5c produces the correct index for the store operation in line 12 by subtracting 2 from %arg2.

Inter-iteration argument passing. Loop pipelining requires the loop to pass results from one iteration to the next, which is usually solved in hardware through dedicated registers. As the MLIR implementation works on a higher level of abstraction, this needs to happen explicitly in the code. MLIR provides a mechanism to pass arguments

Algorithm 3

```
1: function GENERATE_NEW_OP($original\_op, original\_iteration, operand\_maps$)
2:     new_op = original_op.$clone(operand\_maps[original\_iteration])$
3:     for operand : new_op.$operands()$ do
4:         if operand.$is\_index\_substitution\_result()$ == true then
5:             apply_op = $create\_affine\_apply(index\_variable, affine\_map)$
6:             operand_maps[original_iteration].$insert(index\_variable, apply\_op)$
7:         end if
8:         if operand.$is\_result\_from\_prologue()$ == true then
9:             $add\_prologue\_result\_to\_iter\_args()$
10:            operand_maps[original_iteration].$insert(original\_op\_result, iter\_arg)$
11:        end if
12:        operand_maps[original_iteration].$insert(original\_op.result, new\_op.result)$
13:    end for
14: end function
```

Figure 3.8: *New loop generation algorithm.*

to each new iteration of the loop: operations that need to pass their result to a following iteration of the loop can *yield* a value, which will be available as *iteration argument* at the beginning of the next iteration.

In the 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. Two different solutions to this problem are proposed in [95]: loop unrolling and rotating register files. Unrolling introduces further code expansion, while rotating register files requires architectural support that cannot be modeled on such a high level of abstraction. A simpler solution was implemented using additional iteration arguments, such as the %arg5 iteration argument in Figure 3.9, which is simply shifted to %arg4 at the end of the iteration.

Epilogue generation and old loop removal. Epilogue generation follows a similar procedure to prologue generation, with an additional step that maps the results of yield

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

Figure 3.9: *Inter-iteration argument passing.*

operations from the last loop iteration to epilogue operands. Finally, the old loop is removed from the code.

3.4.3 Support for results forwarding

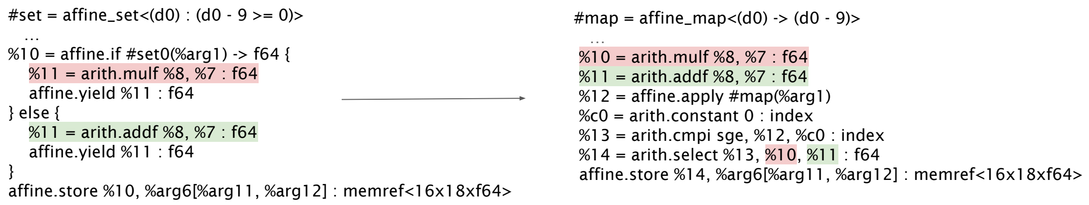
When a loop reads and writes to the same memory address at each iteration (e.g., the innermost one in a nest accessing an address that depends on loop indexes from outer loops), this creates an inter-iteration dependency: an iteration cannot start before the previous one has finished writing to memory, limiting the possibility to apply loop pipelining. This issue was solved by the introduction of an MLIR pass that forwards results from one iteration to another: the load and store operations are moved outside of the innermost loop, and inter-iteration argument passing forwards intermediate results through yield operations and iteration arguments. The pass is applied before the DFG extraction pass to facilitate loop pipelining, but it provides performance benefits also in isolation.

3.4.4 Support for variable loop bounds

The new loop produces correct results under the assumption that 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 both the lower and upper bounds of the original loop are constant, and the number of iterations does not satisfy this condition, the loop transformation pass will not attempt to modify the loop and return an error message. If one of the loop bounds is a constant and the other one is a variable (e.g., depending on an input argument or on a parent loop index), 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, a check was introduced 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 results in having both the original and the pipelined loop in the code, causing additional area overhead in the generated accelerator but no degradation in performance. This is a simple and effective solution that can be easily implemented in MLIR with affine if operations and affine sets, as opposed to modifications within the HLS tool which would require expertise with low-level abstractions (for example, to generate a dedicated controller able to stop the pipelined execution at runtime), and would need to be reimplemented for each different HLS backend.

3.4.5 Support for if-else statements

An if-conversion pass was also implemented in MLIR following [96], which allows to pipeline loops containing if and else blocks. If-conversion is run as a preprocessing step before DFG extraction to allow parallelization of the operations inside the if and else blocks; affine if and else blocks would otherwise be treated as a single operation in the scheduling phase. 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 is the correct one (Figure 3.10). The pass is only run if there are no operations that write to memory or call external functions, to preserve the correctness of results even if both if and else blocks are speculatively executed. This is a transformation that was designed for software, but thanks to the higher level of abstraction provided by MLIR it can be seamlessly applied, in isolation or together with loop pipelining, to benefit HLS results. Additional transformations such as this one could easily be implemented in the future as MLIR passes working together with the ones that were presented: an example is the technique proposed in [97], which tackles nested loop optimizations and merges epilogue and prologue of adjacent iterations.



```
#set = affine_set<(d0) : (d0 - 9 >= 0)>
  ...
%10 = affine.if #set0(%arg1) -> f64 {
   %11 = arith.mulf %8, %7 : f64
   affine.yield %11 : f64
} else {
   %11 = arith.addf %8, %7 : f64
   affine.yield %11 : f64
}
affine.store %10, %arg6[%arg11, %arg12] : memref<16x18xf64>
```

```
#map = affine_map<(d0) -> (d0 - 9)>
  ...
%10 = arith.mulf %8, %7 : f64
%11 = arith.addf %8, %7 : f64
%12 = affine.apply #map(%arg1)
%c0 = arith.constant 0 : index
%13 = arith.cmpi sge, %12, %c0 : index
%14 = arith.select %13, %10, %11 : f64
affine.store %14, %arg6[%arg11, %arg12] : memref<16x18xf64>
```

Figure 3.10: *Effect of the if-conversion pass.*

3.5 A frontend optimizer for HLS

SODA-OPT [15] is an MLIR-based compiler tool supporting system-level design and hardware acceleration of applications developed in high-level programming frameworks, and it is part of the SODA Synthesizer, which will be described in more detail in Chapter 4. After selecting a kernel for acceleration, SODA-OPT applies a modular optimization pipeline that restructures it so that it will be better suited for hardware synthesis, exploiting high-level MLIR optimizations. Some of the optimizations available in SODA-OPT are standard compiler optimizations while some of them are custom passes, and the loop pipelining implementation that was described in this chapter has been integrated into the set of available SODA-OPT optimizations (Table 3.1 lists the most relevant ones). The default set of active optimizations privileges passes that result in faster accelerators when synthesized, exposing instruction- and data-level parallelism and removing unnecessary operations. However, the optimization pipeline is not monolithic: developers can easily enable, disable, reuse, or modify optimizations, providing ample opportunities to customize the process for different applications and implement automated DSE strategies.

Table 3.1: *Partial list of high-level optimizations available in SODA-OPT.*

Optimization pass	Effect	Default active
Loop unrolling	Expose instruction-level parallelism	yes
Loop tiling	Balance computation and memory transfer	no
Loop pipelining	Parallelize loop iterations	no
If-conversion	Speculative execution of if-else blocks	yes
Results forwarding	Remove unnecessary memory transfers	yes
Temporary buffer allocation	Reduce accesses to external memory	yes
Common sub-expression elimination	Remove unnecessary operations	yes

3.6 Conclusion

Implementing loop optimizations as compiler transformations in a preliminary step before HLS, as opposed to implementing them inside the tool, can improve the performance of the generated accelerators, increase developer productivity, and decouple the optimizations from a specific backend tool. To support this claim, a set of compiler passes supporting loop pipelining has been implemented exploiting the MLIR affine dialect, a specialized representation designed to enable polyhedral optimizations. Such an approach opens the way to further research to explore optimization techniques that can benefit HLS when they are applied at a higher level of abstraction than existing solutions. Thanks to the modular nature of MLIR, the proposed loop pipelining pass has been seamlessly integrated into a compiler-based frontend optimizer for HLS.

CHAPTER 4

End-to-end synthesis

THE SOfware Defined Architectures (SODA) Synthesizer is a practical realization of the multi-level approach to HLS proposed by this thesis: an MLIR-based, modular hardware compiler that provides an automated path from high-level programming tools to FPGA/ASIC. SODA answers the demand for specialized hardware accelerators caused by the ever-changing nature of data science workloads, and it can assemble multiple generated accelerators in a custom dataflow system driven by a distributed controller.

This chapter contains material from:

S. Curzel, N. Bohm Agostini, V. G. Castellana, M. Minutoli, A. Limaye, J. Manzano, J. Zhang, D. Brooks, G.-Y. Wei, F. Ferrandi, and A. Tumeo, "End-to-end Synthesis of Dynamically Controlled Machine Learning Accelerators," to appear in *IEEE Transactions on Computers*;

N. Bohm Agostini, S. Curzel, J. J. Zhang, A. Limaye, C. Tan, V. Amatya, M. Minutoli, V. G. Castellana, J. Manzano, D. Brooks, G.-Y. Wei, and A. Tumeo, "Bridging Python to Silicon: The SODA Toolchain," *IEEE Micro*, vol. 42, no. 5, pp. 78–88, 2022.

4.1 Overview

Next-generation edge systems will operate under conditions where exporting all the acquired data for centralized processing is inconvenient or impossible [58]. For example, monitoring infrastructure for highly dynamic systems (e.g., sensor networks for smart cities, the power grid, and environmental monitoring) will need to operate in low-power settings with limited bandwidth available for communication [99]. Experimental instruments such as the ones owned by the US Department of Energy (e.g., particle accelerators, mass spectrometers, and electron microscopes), already generate volumes of data that are impossible to store or transfer without pre-processing [100]. Extreme conditions require highly specialized processing systems, optimized along a variety of metrics that include energy, performance, latency, size, and thermal dissipation; such requirements combined are usually impossible to meet for general-purpose processors, so application-specific accelerators become a necessity.

On the application side, domain experts design and validate their algorithms in high-level programming frameworks. Especially in the fields of data science and ML, both algorithms and programming frameworks evolve quickly, outpacing conventional hardware design processes and highlighting their significant productivity limitations: manually designing custom accelerators is complex, expensive, and time-consuming, preventing effective exploration of alternative architectures and often requiring a new design cycle to support innovations efficiently. General and automated solutions are needed to quickly transition from the formulation of an algorithm to the implementation of a dedicated accelerator, as no single design can support the extreme diversity and fast-paced growth of data science.

Hardware designers usually extract key computational patterns from the algorithms that need to be accelerated, identify parallelism and data reuse opportunities, and design custom functional units for specific kernels; a common alternative to accelerate this process is to implement the functional units in C/C++ and convert them to HDL through HLS tools. In both cases, after functional verification, the kernels are passed to downstream logic synthesis and physical design tools and finally integrated into a system. The interaction between multiple Computer-Aided Design (CAD) tools at different levels of abstraction, with part manual coding and part automated processing, requires considerable effort to propagate changes across different stages of the design flow, and the quality of the final design highly depends on the designers' expertise.

The modern perspective on HLS proposed in this thesis addresses these issues through a multi-level approach, which has been realized in practice through the SOftware Defined Architectures (SODA) Synthesizer [13, 14], an open-source, multi-level, modular, extensible, no-human-in-the-loop hardware compiler that translates high-level ML models into domain-specific accelerators. The SODA Synthesizer comprises a compiler-based frontend that leverages MLIR (SODA-OPT) and a compiler-based backend that integrates state-of-the-art HLS methodologies (Bambu); it generates highly specialized designs that can be synthesized with both commercial and open-source tools on FPGAs or ASICs, and it allows the exploration of design metrics through compilation passes and parameters, enabling the identification of optimal trade-offs depending on the target application requirements. Such an exploration would require multiple expensive redesigns with traditional HDL- or HLS-based approaches; SODA, instead, provides a no-human-in-the-loop end-to-end exploration flow where no modifications to the input code are needed, and its modular and extensible approach facilitates the introduction of new analysis and transformation passes.

Large compute- and memory-intensive ML algorithms (e.g., DNNs) frequently represent a challenge for HLS tools, and they need to be broken down into smaller kernels as the complexity of the synthesis, and of the generated FSM controller, would grow exponentially. The issue is especially evident when input models contain multiple parallel execution flows, e.g., in presence of coarse-grained parallelism, or when the model needs to process streaming inputs in a pipelined fashion. The SODA Synthesizer can target a system-on-chip (SoC) with a central general-purpose microcontroller that drives multiple accelerators (based on the FSMD model) implementing different layers of an ML model; however, in such a system the data movement between the host microcontroller, the accelerators, and memory quickly becomes a performance bottleneck. For this reason, it has been extended to support the generation of a second type of system: a dynamically scheduled architecture where custom FSMD accelerators are composed in a dataflow system and are driven by a distributed controller. In this architecture, multiple accelerators can perform computations in parallel on different portions of streaming input data, without requiring orchestration from the host microcontroller, and can communicate with each other without going through external memory.

SODA integrates and improves the methodology presented in [57] to synthesize parallel C code, annotated with OpenMP-like directives, into a dataflow architecture with support for spatial parallelism, resource reuse, and memory access parallelism. That approach could identify certain degrees of parallelism by analyzing program dependencies, but it was constrained by conservative alias analysis; user-provided annotations were needed to simplify the dependency analysis and expose dynamic parallelism. By leveraging MLIR, instead, the SODA Synthesizer has access to high-level representations that explicitly capture hierarchy, parallelism, and how data flows through operators and memory in a computational graph, removing the need for complex alias analysis and simplifying the generation of dataflow architectures.

In summary, the contributions of this chapter are:

- an automated, modular, multi-level, compiler-based design flow from high-level ML frameworks to optimized FPGA or ASIC accelerators implemented following the FSMD model;
- a search and outlining methodology to automatically extract accelerator kernels and their dependencies from an MLIR input specification;
- a system integration methodology to assemble FSMD accelerators into a coarse-grained, dynamically scheduled dataflow architecture with distributed control.

The SODA Synthesizer is introduced in Section 4.2, and detailed in Sections 4.3-4.4, Section 4.5 draws conclusion and outlines future research directions.

4.2 The SODA Synthesizer

Figure 4.1 provides an overview of the SODA Synthesizer, which is composed of two main parts: a compiler-based frontend and a compiler-based hardware generation engine. Optimizations at all levels of the toolchain are implemented as compiler passes, significantly influencing the generated hardware designs in terms of performance, area, and power; DSE is thus possible by enabling and disabling compiler passes or tuning their options. The SODA Synthesizer frontend interfaces with high-level programming frameworks through MLIR, partitions the input applications by identifying key computational kernels for hardware acceleration, and performs high-level optimizations that improve the performance of the generated specialized systems. The frontend then generates an LLVM IR as output, which is the starting point for hardware generation. The backend integrates Bambu, a state-of-the-art open-source HLS tool, to generate RTL code for the hardware accelerators. To compile code that will be executed on a host processor, instead, SODA uses standard LLVM tools.

SODA-OPT [15] is the high-level compilation frontend of the SODA Synthesizer. One possible entry point to SODA-OPT is through the `tf-mlir-translate` and `tf-opt` tools from TensorFlow, which compile ML models defined and trained in TensorFlow into an MLIR representation. SODA-OPT implements analysis and transformation passes that parse MLIR inputs from high-level programming frameworks, identify key operation groups, and mark them for hardware acceleration. Selected

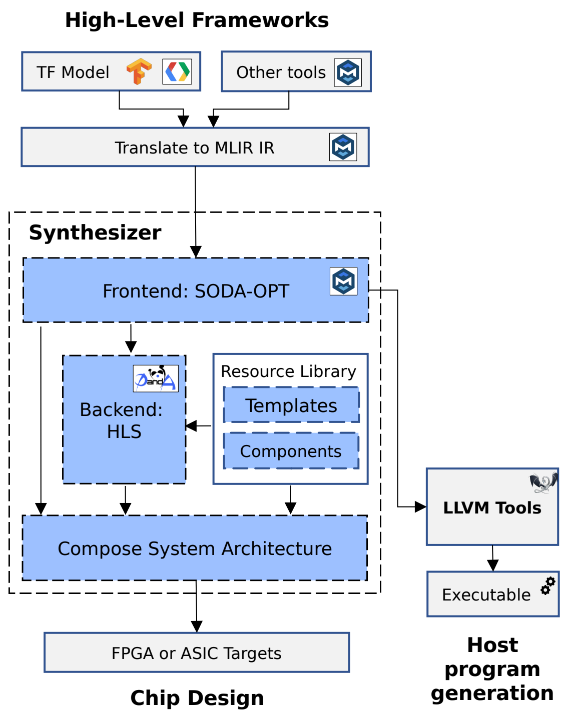


Figure 4.1: *The SODA Synthesizer and its interfaces towards external tools.*

kernels undergo an optimization pipeline with progressive lowerings through different MLIR dialects (e.g., *linalg* $\rightarrow$ *affine* $\rightarrow$ *SCF* $\rightarrow$ *cf* $\rightarrow$ *llvm*), and they are finally translated into an LLVM IR purposely restructured for HLS. SODA-OPT can lower the remaining operations in two different ways, depending on the desired target: they can represent orchestrating code executed by a host microcontroller in a centralized SoC, or the relationship between accelerators in a distributed control (dataflow) architecture. In the first case, SODA-OPT generates another LLVM IR file that will be compiled with standard LLVM tools, including runtime calls to control the generated accelerators. In the second case, operations are transformed into a function-based representation (task graph) that allows to generate the required distributed controller logic and memory interfaces; accelerators and controller modules will then be assembled together to form the dataflow architecture. Section 4.3 describes more in detail how SODA-OPT operates the selection and optimization of kernels.

The SODA Synthesizer backend, Bambu [16], leverages state-of-the-art HLS techniques to synthesize the LLVM IR produced by the frontend into a hardware accelerator for FPGA or ASIC. By default, Bambu synthesizes RTL designs in Verilog following the FSMD model, but it has also been extended with novel methodologies that enhance modularity and allow the generation of dynamically scheduled accelerators. For example, synthesized modules representing functions within a larger specification can be reused thanks to the technique presented in [101], providing opportunities for modular and hierarchical designs. Bambu was extended to allow the integration of FSMD modules as processing elements in a coarse-grained dataflow design [57], and in multithreaded parallel accelerators [102]. These synthesis methodologies were initially developed by integrating support for parallel C specifications annotated with a set of OpenMP directives: users would identify parallel sections in the input code through annotations, allowing Bambu to generate custom accelerator modules and to combine them in a dynamically scheduled or multithreaded architecture. With the proposed multi-level approach, instead, it is possible to exploit MLIR to significantly improve and automate the design of systems composed of multiple accelerators. As high-level MLIR specifications are naturally parallel and hierarchical, there is no requirement for the user to manually identify code regions of interest; moreover, MLIR facilitates the implementation of analysis and transformation passes that identify kernels to be accelerated, analyze their interactions, and compose them in a system. These are all tasks that are better solved at higher levels of abstraction, allowing the HLS engine to focus only on the generation of optimized accelerators.

The SODA Synthesizer can interface with commercial and open-source logic synthesis tools to generate hardware accelerators: Bambu is a multi-platform tool supporting both FPGA (from Xilinx, Intel, Lattice, NanoXplore) and ASIC targets (through OpenROAD or the Synopsis Design Compiler) without any modification in the input code; such flexibility enables a smooth transition from high-level design, to FPGA prototyping, to ASIC manufacturing and deployment. Finally, it also provides verification features to ensure that the generated designs are functionally correct, as Bambu includes a suite of tools that enable automatic testbench generation and validation of results through external open-source and commercial simulators. The SODA-OPT frontend feeds simulation inputs to Bambu; Bambu, in turn, generates testbenches, scripts, and glue code to drive the execution of a simulator and automatically verifies that the output values of the simulation correspond to the results from the execution of the original application with the same inputs.

4.3 Kernel selection and optimization

As mentioned in Section 4.2, SODA-OPT is the high-level compiler frontend of the SODA Synthesizer; its main tasks include interfacing with high-level programming frameworks, selecting relevant kernels for acceleration, and preparing them for hardware generation (Figure 4.2). This process can be divided in *search*, *outlining*, *optimization*, and *dispatch* phases.

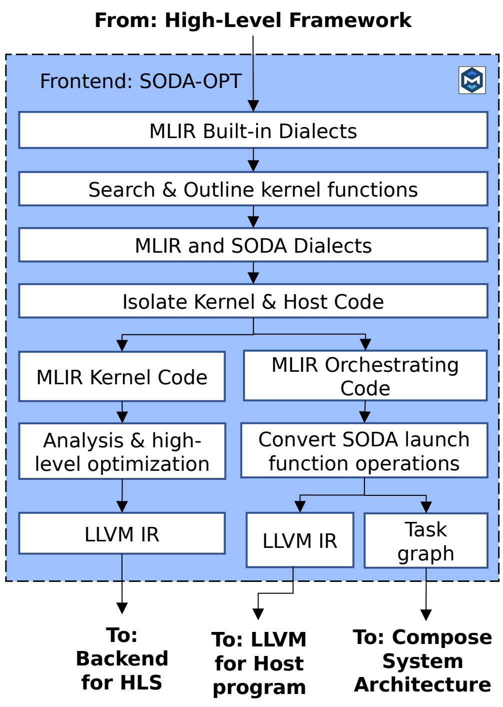

From: High-Level Framework
Frontend: SODA-OPT
MLIR Built-in Dialects
Search & Outline kernel functions
MLIR and SODA Dialects
Isolate Kernel & Host Code
MLIR Kernel Code
MLIR Orchestrating Code
Analysis & high-level optimization
Convert SODA launch function operations
LLVM IR
LLVM IR
Task graph
To: Backend for HLS
To: LLVM for Host program
To: Compose System Architecture

Figure 4.2: *The SODA-OPT high-level compilation frontend.*

Entry points to SODA-OPT are any of the MLIR dialects that are maintained in tree, along with the core MLIR framework; here they will be referred to as *built-in* dialects. Built-in dialects include abstractions for linear algebra (linalg), polyhedral analysis (affine), structured control flow (SCF), and others that are directly contributed to the MLIR repository for their broad applicability. High-level programming frameworks for various domains including ML (e.g., TensorFlow, ONNX, PyTorch) implement custom MLIR dialects, optimization passes, and lowering methods to translate their programs into built-in MLIR dialects. SODA-OPT introduces a custom dialect as well, the *soda* dialect, to partition input algorithms into kernels that will be translated into hardware accelerators and logic that controls their execution. Table 4.1 describes the soda dialect operations; the following sections detail the search and outlining process that uses them, and subsequent optimization and dispatching phases.

4.3.1 Search phase

SODA-OPT automatically identifies operations that are well suited for acceleration by matching key patterns at the earliest stages of the compilation process (*search* phase). Searched patterns are mainly linear algebra operations or affine structures wrapping arithmetic operations, selected among the most common computations in ML applications. SODA-OPT can easily be extended by adding new patterns of in-

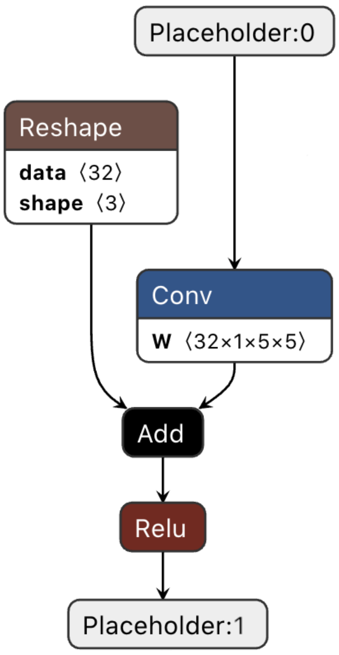
Table 4.1: *Operations in the soda dialect.*

Operation	Semantics
`soda.launch`	Marks the beginning of a region with MLIR operations to be outlined and extracted into a kernel.
`soda.terminator`	Marks the end of a region to be outlined and extracted.
`soda.module`	Block of outlined operations, it will become a unique accelerator.
`soda.func`	Defines an outlined function with its interface.
`soda.return`	Indicates the end of an outlined function.
`soda.launch_func`	Calls the accelerated function from the controller code.

terest beyond ML, as could happen when the input is a scientific computing application translated to MLIR from a different domain-specific framework. Search passes wrap a `soda.launch` operation around the operations to be outlined, and inject a `soda.terminator` operation at its end. Looking at Figure 4.3a, representing a small portion of a CNN, a user might decide to separately accelerate each node in the computational graph (one reshape operation, one convolution, one bias add, and one ReLu activation function). When the model is lowered to the MLIR linalg dialect, each of them is represented by a `linalg.generic` construct (Figure 4.3b), which SODA-OPT recognizes as a region of interest and marks with launch and terminator operations. Users can freely decide which regions to mark when generating accelerators for a centralized SoC, as there is no constraint on what part of the code is accelerated when kernel execution is orchestrated by a host microcontroller. When targeting the dataflow architecture, instead, SODA-OPT individually marks for outlining all operations in the MLIR file, so that each of them will be synthesized as a dataflow stage and driven by a distributed controller. In the future, the search strategy can be improved to support flexible granularity so that operations are fused together or partitioned to generate accelerators with similar computational intensity, aiming for balanced execution time and resource utilization across dataflow stages.

4.3.2 Outlining phase

At the beginning of the *outlining* phase, SODA-OPT extracts each region of code enclosed within marks during the search phase into a separate MLIR module, inlining any functions invoked inside it. SODA-OPT adds an attribute to the module to indicate the target architecture (centralized or dataflow), and to later select the corresponding backend compilation or synthesis flow. The outlining process proceeds by analyzing use-def chains of values inside each module to generate the interface of a top-level kernel function, adding to the function arguments also memory buffers allocated outside the `soda.launch` region, but referenced inside it. Constant values are instead pulled



(a) *Graphical representation.*

```
func @main(...) {
  ...
  linalg.generic #reshape_trait ... {
  ^bb0(...):
    linalg.yield %arg2 : f32
  }
  linalg.generic #conv2d_trait ... {
  ^bb0(...):
    %7 = arith.mulf %arg2, %arg3 : f32
    %8 = arith.addf %arg4, %7 : f32
    linalg.yield %8 : f32
  }
  linalg.generic #add_trait ... {
  ^bb0(...):
    %7 = arith.addf %arg2, %arg3 : f32
    linalg.yield %7 : f32
  }
  linalg.generic #relu_trait ... {
  ^bb0(...):
    %7 = arith.cmpf olt, %arg2, %cst_0 : f32
    %8 = arith.select %7, %cst_0, %arg2 : f32
    %9 = arith.cmpf olt, %cst, %arg2 : f32
    %10 = arith.select %9, %cst, %8 : f32
    linalg.yield %10 : f32
  }
  return
}
```

(b) *MLIR representation (simplified).*

Figure 4.3: *Example input to SODA-OPT (section of a CNN model).*

inside the kernel. The process ends with the generation of a `soda.func` for each each `soda.launch` block, placed in a separate `soda.module`; outlined kernels are substituted by `soda.launch_func` operations in the top-level code that will orchestrate their execution (`main` function in Figure 4.4).

4.3.3 Optimization phase

After outlining, each kernel is *optimized* separately, passing through progressive lowering steps that eventually translate it into an LLVM IR. SODA-OPT exploits several dialect-specific optimization passes from built-in dialects, together with some custom, HLS-oriented transformations (as introduced in Section 3.5), providing a modular optimization pipeline that restructures the kernels so that the final low-level IR is well suited for hardware synthesis. The default optimization pipeline produces an LLVM IR that presents simpler dependency chains, few or no redundant instructions, and regular load-compute-store patterns: such characteristics improve the resource allocation and static scheduling of operations performed by the HLS engine, resulting in significant performance gains. Available optimizations significantly influence the generated hardware designs in terms of performance, area, and power consumption, and they are all

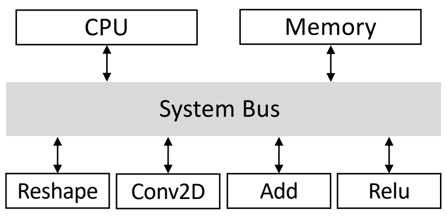
```
func @main(...) {
  ...
  soda.launch_func @df0_reshape::@f args(...)
  soda.launch_func @df1_conv2d::@f args(...)
  soda.launch_func @df2_add::@f args(...)
  soda.launch_func @df3_relu::@f args(...)
return


soda.func @f(...) {
    linalg.generic #reshape_trait // ...
    soda.return
}

soda.func @f(...) {
    linalg.generic #conv2d_trait // ...
    soda.return
}

soda.func @f(...) {
    linalg.generic #add_trait // ...
}

soda.func @f(...) {
    linalg.generic #relu_trait // ...
    soda.return
}
```

Figure 4.4: *MLIR representation after SODA-OPT search and outlining (simplified).*

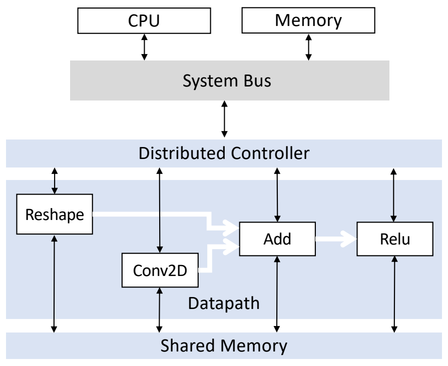
implemented as compiler passes; users can thus perform an exhaustive exploration of the design space without manual interventions on the code to find the combination of passes that better meets the application requirements. While the focus of the optimization pipeline is the generation of hardware accelerators, SODA-OPT can be extended to apply optimizations also on the host code generation path: for example, to enable parallel execution of different accelerators through non-blocking calls, better use of the CPU cache hierarchy, and automatic re-use of accelerators when possible.

4.3.4 Dispatch phase

Dispatching separates the kernels from the logic that orchestrates their execution: at the end of the compilation, SODA-OPT generates a separate file for each kernel that does not contain references to the rest of the code and collects all orchestrating logic in another file. Bambu will generate an FSMD accelerator for each of the IR files containing the kernels, later integrated into one of two possible system-level architectures. The SODA Synthesizer currently supports two types of architectures: a conventional system-on-chip where a microcontroller drives one or more accelerators connected through a bus (centralized architecture, Figure 4.5a), and a single accelerator where kernels are connected together in a dynamically scheduled dataflow architecture (Figure 4.5b). In the first case, the orchestrating logic extracted by SODA-OPT will contain function calls for the outlined kernels, which will be substituted by driver calls to the corresponding accelerators in the compiled host program. Instead, when targeting the dataflow architecture, SODA-OPT generates a task graph representing interactions between the kernels, containing information that will be used to assemble the accelerators and the distributed controller. In particular, the task graph includes the name of each kernel with the direction (input/output) of its arguments, and the sizes of exchanged data structures retrieved by leveraging the *memref* MLIR dialect. In all cases, SODA-OPT provides the backend with LLVM IRs for the kernels and test values to verify the correct behavior of generated accelerators.



(a) *Centralized architecture*



(b) *Dataflow architecture*

Figure 4.5: *Available architectural targets supported by the SODA Synthesizer.*

4.4 Dataflow architecture generation

In the SODA Synthesizer backend, Bambu generates FPGA/ASIC accelerators for each kernel selected and optimized by the frontend, and the isolated FSMD kernels are then composed together in one of the two available architectures in Figure 4.5. Bambu performs standard HLS steps and optimizations (e.g., loop transformations, bitwidth analysis, scheduling, and binding), and finally generates both RTL code for the accelerators and testbenches for verification. After HLS, Bambu launches a simulator chosen by the user and verifies that the output values from the generated kernel correspond to golden results gathered from the execution of the input.

During code generation, HLS tools combine RTL descriptions of functional units from a resource library implementing the operations present in the IR (adders, subtractors, multipliers, etc.); to effectively drive the synthesis algorithms, these functional units are characterized in terms of performance (e.g., latency of the critical path) and area for each target technology or device. Area and performance estimates directly affect many optimization passes and synthesis algorithms: for example, they help decide whether functional units can be chained together by removing intermediate registers if their combined latency does not exceed the required clock period. The characterization of functional units through the OpenROAD flow and the FreePDK 45 nm cell technology library allowed to provide a completely open-source, end-to-end hardware generation flow from high-level programming frameworks to ASIC.

The process to generate the dynamically scheduled dataflow architecture instantiates components from the Bambu resource library, derived from the ones presented in [57] and now integrated in the SODA Synthesizer, where the generation process can take advantage of the outlining, analysis, and transformation passes performed by SODA-OPT. The main components involved are a *distributed controller*, that activates FSMD accelerators at runtime, and a *hierarchical memory interface* that manages concurrent memory access to shared memory. The distributed controller starts the execution of each FSMD synthesized from a kernel outlined by SODA-OPT according to data dependencies described in the task graph (also generated by SODA-OPT).

4.4.1 Distributed controller

The distributed controller employs dedicated hardware components to check, at runtime, when to start the execution of the FSMD accelerators. By allowing concurrent execution of multiple modules even when their latency depends on input data, or when they simultaneously access shared memory, the controller allows pipelined execution of kernels, which is essential to run ML inference on streaming inputs with low latency.

The controller generation flow instantiates for each FSMD accelerator a dedicated *resource manager* (RM), and for each execution of the same accelerator in the task graph an *execution manager* (EM); internally, EMs are composed of three parts: the *activation manager* (AM), the *operation manager* (OM), and the *status manager* (SM), as can be seen in Figure 4.6. The EM triggers the execution of FSMD accelerators when activating conditions for an operation are met, while the RM acts as an arbiter to handle requests for a module that is shared among multiple operations. All EM components are based on combinational logic, so they do not add delay cycles to the execution time of the FSMD modules.

Activating conditions for each FSMD accelerator are derived from the data dependencies between operators in the task graph provided by SODA-OPT so that each operator can start execution as soon as its inputs are ready. Within each execution manager, the AM module compares the activation conditions for its associated operation against

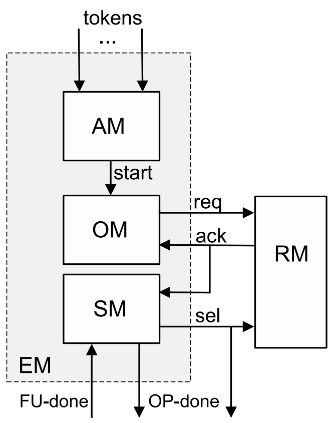


Figure 4.6: *Execution manager components interfacing with a resource manager.*

tokens received from other execution managers; once all necessary tokens are received from producer operations, the activation manager notifies the operation manager to start execution. The operation manager interacts with the resource manager to check whether the required accelerator is available; if this is the case, the status manager can send required control signals to the FSMD and prevent the associated resource manager from accepting new requests.

When its execution is concluded, the FSMD module sends a completion signal (*FU-done*) to notify consumer operations that its outputs are ready. The signal is received by all execution managers that share the FSMD module, but it is ignored unless the SM module indicates that the operation is running; this procedure allows each EM to discriminate between the end of their associated operation and the end of other operations mapped on the same module. At this point, the EM releases control on the resource manager and emits *OP-done* token signals to notify the end of the execution to EMs associated with consumer operations.

In statically scheduled FSMDs, operations that execute concurrently are not allowed to share the same hardware module, preventing resource conflicts by design; in a dataflow architecture, instead, resource conflicts are resolved dynamically at runtime through resource managers. During the synthesis process, the module binding phase maps operations to resources: in the proposed design flow, operations are neural network layers (or other coarse-grained linear algebra algorithms), and resources are the statically scheduled FSMD accelerators. Module binding aims at heuristically reducing the number of resource conflicts stalling the execution. It is implemented as the solution of clique covering problem on a Weighted Compatibility Graph (WCG) [103] where nodes represent operations, and edges represent compatibility relations (i.e., if two operations are connected, they can share a hardware resource); weights express the profitability in terms of area and latency of sharing a module between two compatible operations, with lower values discouraging sharing and higher values favoring it. While clique covering is an NP-complete algorithm, the heuristic always completes in a reasonable time because it only needs to handle intentionally small graphs, as nodes in the proposed approach correspond to layers in a neural network (or in any case to large portions of the input application). After binding, the distributed controller generation process defines tie-breaking rules for resource managers that will determine how to resolve at runtime any structural conflict that may occur if multiple operations request a module at the same time. In the current implementation, tie-breaking rules are based on the topological order of the operations in the input task graph; a different method may lead to a different execution order and overall performance, but the execution output would remain the same because the system is built to respect dependencies between operations. When operations require the same module at different times, there is no competition and RMs simply process requests following the order of arrival.

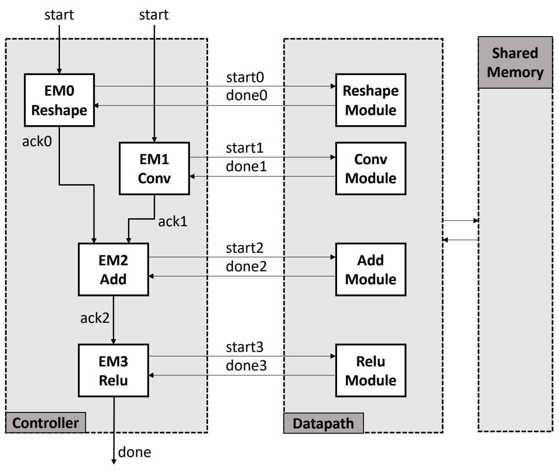


Figure 4.7: *Dataflow accelerator schematic for the model of Figure 4.3a.*

The synthesis process allocates one resource manager for each shared module according to the results of module binding, and then it traverses the task graph instantiating an execution manager for each operation, each with a custom activation manager synthesized according to operation dependencies. Figure 4.7 shows a schematic of the dataflow accelerator for the ML model of Figure 4.3a: after SODA-OPT optimization and dispatching, the task graph contains four calls to the different kernel functions; resource managers are not shown in the figure as there is only one call for each FSMD module in the task graph. Bambu synthesizes the four kernel functions using the standard FSMD approach, and the necessary distributed controller components are instantiated according to the task graph describing the dependencies between functions. FSMD modules are then assembled with their execution managers, resource managers, and the memory interface (Section 4.4.2) to generate the dataflow accelerator.

4.4.2 Hierarchical memory interface

After generating the datapath and the distributed controller, the accelerator needs to be connected to memory. Bambu provides several options to connect accelerators to memories: for example, it can generate one read and one store port for a whole module, or read and store ports for each argument of the module (which can be used concurrently if the arguments do not alias); then it instantiates and connects the accelerator interfaces to multi-ported scratchpads (or BRAMs for FPGAs). By default, Bambu connects a dual-ported scratchpad memory to each couple of load/store ports, assuming a fixed latency of 1 clock cycle for read operations and 2 clock cycles for store operations.

The dynamically scheduled design leverages a specialized memory controller called hierarchical memory interface, or HMI, that allows independent accelerators to concurrently access shared memory. The HMI is a multi-ported controller that dynamically assigns concurrent requests to external memory channels, computing destination addresses at runtime with no additional delay, extended from the design of the custom memory interface controller described in [104]. Load and store ports from FSMD accelerators are connected to the HMI which, in turn, connects them to a multi-ported shared memory; the design can be connected to high-performance multi-banked scratchpad memories or interface with external multi-ported DRAM controllers (e.g., Xilinx AXI DRAM controllers for FPGAs). SODA-OPT analysis passes compute the amount of data exchanged between kernels, and consequently determine the required size of the shared memory, accounting for double buffering and concurrent execution of the accelerators.

Figure 4.8 shows the schematic representation of the HMI for two generic FSMD modules *x* and *y*: each module has a memory interface (MI) that is connected to the others in a chain, and only the top-level datapath is directly interfaced with the external memory (in this case through a single channel). Each MI performs only one memory operation at a time, but all MIs can operate in parallel; signals that request read and write access (*sel_load* and *sel_store*) are propagated through MIs following the hierarchy of functions in the task graph. If the destination addresses of different memory operations collide, the HMI serializes the memory accesses.

As there is a finite number of channels toward external memory, structural con-

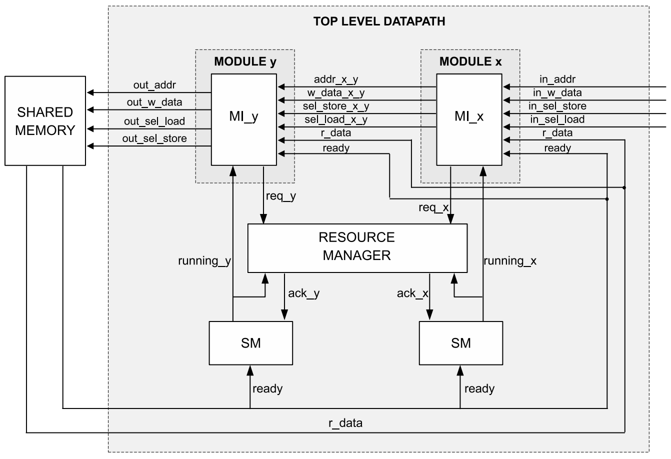


Figure 4.8: *Structure of the hierarchical memory interface.*

flicts may arise when multiple operations request access to the same memory channel. Statically scheduled designs prevent conflicts by pre-determining the order of operations and executing only one operation at a time; however, this can degenerate into sequential execution of modules that could instead execute simultaneously for part of their computation. In the proposed dataflow architecture, instead, the HMI integrates additional control logic exploiting resource manager and status manager blocks from the distributed controller, obtaining a design that can resolve conflicts at runtime: a resource manager intercepts memory access requests, and if a memory channel is available it accepts the request notifying a dedicated status manager component associated with the MI that issued the request. In Figure 4.8, *SM_x* is associated with the MI of module x, while *SM_y* is associated with the MI of y. As in the case of the distributed controller, the HMI design is modular and easy to assemble; if the top module encapsulating x and y also needs to access memory, a third SM and a third MI can easily be added to the interface.

4.5 Conclusion

The SODA Synthesizer is an example of the modern, multi-level approach to HLS proposed in this thesis, applied to the generation of specialized accelerators for ML: an open-source, automated hardware compiler that transforms specifications from high-level programming frameworks into efficient FPGA/ASIC accelerators and composes them either in a centralized SoC or in a dynamically scheduled dataflow architecture. Its frontend, SODA-OPT, leverages the MLIR framework to identify kernels for acceleration, generate orchestrating code, and apply high-level optimizations; in the backend, the SODA Synthesizer integrates the state-of-the-art HLS tool Bambu with novel synthesis methodologies. SODA can assemble highly optimized FSMD accelerators in a coarse-grained, dynamically scheduled dataflow design, which provides better performance in the case of pipelined execution on streaming inputs compared to a centralized architecture driven by a microcontroller. The compiler-based toolchain is modular by construction, and thus it enables further research opportunities to introduce new optimization techniques, automate design space exploration, and explore new architectural models.

CHAPTER 5

Introduction of a domain-specific abstraction

NEUROMORPHIC computing and spiking neural networks (SNNs) could perform artificial intelligence tasks with considerably higher efficiency than current tensor-based ML models and accelerators; however, the generation and mapping of specialized hardware for SNNs present significant challenges. The proposed multi-level and modular approach for hardware generation allows to introduce domain-specific abstractions as MLIR dialects to solve this kind of challenge; this chapter describes the implementation of a new dialect that can express the concept of spike sequences, enables mapping of spiking neurons to dedicated hardware components, and supports the synthesis of SNN accelerators. The new abstraction opens opportunities to integrate existing SNN tools within the hardware compilation infrastructure, providing a path toward the generation of complex hybrid artificial intelligence systems.

This chapter contains material from:

S. Curzel, N. Bohm Agostini, S. Song, I. Dagli, A. Limaye, C. Tan, M. Minutoli, V. G. Castellana, V. Amatya, J. Manzano, A. Das, F. Ferrandi, and A. Tumeo, "Automated Generation of Integrated Digital and Spiking Neuromorphic Machine Learning Accelerators," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–7.

5.1 Overview

A modern, MLIR-based approach to HLS can adapt more quickly to innovations than a classic approach based on a library of C/C++ templates, which is a desirable feature considering the rapid evolution of ML and data science. In fact, MLIR facilitates the introduction of domain-specific abstractions to solve domain-specific problems, and parts of the modular compilation pipeline can be enabled or disabled according to the requirements of different classes of input models. For example, a branch of artificial intelligence research is dedicated to the exploration of neuromorphic computing, which mimics how a biological brain operates; the resulting *spiking* neural networks (SNNs) are significantly different from classic artificial neural networks such as DNNs and CNNs. The introduction of a specialized representation within a multi-level design flow can support SNN simulation, mapping of SNN models to neuromorphic hardware, and the generation of hardware accelerators for SNNs, highlighting the benefits of an MLIR-based approach when facing the challenges of a new computational paradigm.

Artificial neural networks are models of computation inspired by the way neurons transmit information in a brain and adapted to available computational units; artificial neurons are commonly used because they are easy to connect in a network and map on digital devices, but they are only a rough approximation of their biological counterparts. Spiking neurons, on the other hand, reflect more closely the time-dependent behavior of biological neurons, and they have sparked interest in the exploration of dedicated neuromorphic computing devices. SNNs are composed of neurons that are activated when their “membrane potential" crosses a threshold, encoding information in the arrival time of spikes with orders of magnitude higher efficiency than conventional artificial neurons [106]. After training, ML models based on spiking neurons can run on specialized neuromorphic devices based on digital [71, 72] or analog processing elements [107] or FPGAs [73, 108, 109] (especially for simulation purposes [110, 111]). Despite the widespread interest in SNN acceleration, existing approaches are often device-specific, require non-trivial interfacing between conventional ML frameworks and SNN tools, and lack support for the integration between analog and digital systems.

This chapter discusses how an end-to-end, multi-level, compiler-based framework can be adapted to support SNN models and enable research on complex heterogeneous systems containing both conventional tensor-based accelerators and neuromorphic devices. In particular, the chapter presents the following contributions:

- a new MLIR dialect to model basic SNN concepts (e.g., integrate-and-fire neurons, spike trains with timestamps);

- a roadmap for the integration of the NeuroXplorer toolchain [112] for SNN mapping and simulation in the SODA Synthesizer;
- a perspective on the challenges of automated synthesis of neuromorphic hardware, and on how modern hardware compilers can help to solve them.

Section 5.2 describes the principles of operation of spiking neurons and SNNs and the traditional compilation flow for neuromorphic hardware; Section 5.3 outlines a roadmap for the synthesis of SNN accelerator through the SODA Synthesizer, with Section 5.3.2 describing the dedicated MLIR dialect; Section 5.4 draws conclusions and outlines future research directions.

5.2 Spiking neural networks and neuromorphic accelerators

Biological neurons are interconnected through synapses that carry electrical signals; when a neuron is activated it *fires*, i.e., it produces a current spike on its output synapses. Neurons in artificial neural networks apply an activation function (e.g., sigmoid or ReLU) on a weighted sum of their inputs, mimicking with continuous output values the firing rates of biological neurons; these characteristics allow efficient training through gradient-based methods and backpropagation, leading to the explosion of deep learning [113]. However, it has been proved that neurons encode information not by varying their firing rate, but in the arrival times of current spikes; spiking neurons are thus a model of computation that more closely resembles the biological process [106]. In a spiking neuron model, currents injected from input synapses raise the membrane voltage of a neuron until its membrane voltage crosses a threshold that causes the neuron to fire. Figure 5.1 depicts a leaky integrate-and-fire (LIF) spiking neuron model represented as an RC circuit that integrates the input current $U(t)$; and produces an output spike when the potential crosses the threshold V_{th} [114]. Figure 5.2 show the timing behavior of a LIF neuron; spikes can last from 1 μs to several ms.

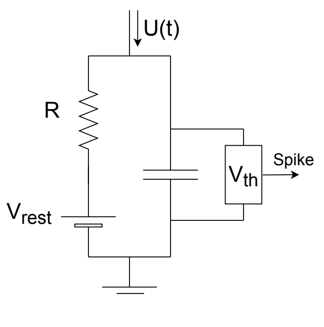
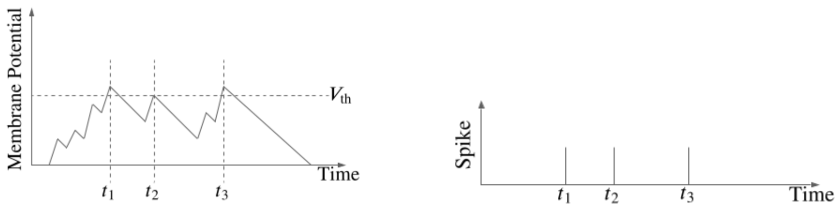


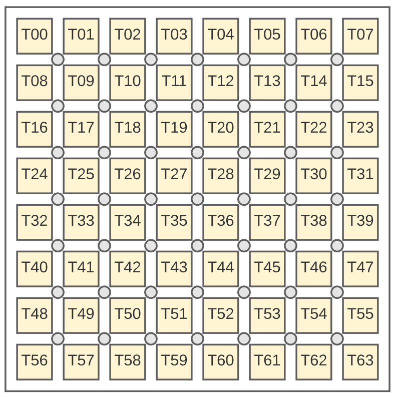
Figure 5.1: *A leaky integrate-and-fire (LIF) neuron model.*

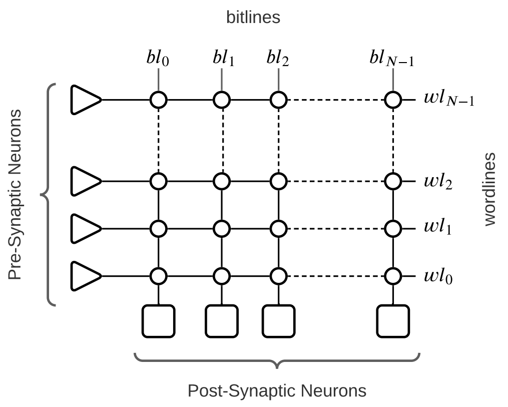


(a) *Neuron membrane potential.* **(b)** *Neuron firing times.*

Figure 5.2: *Membrane potential and output spikes of a LIF neuron over time.*

Neural networks composed of spiking neurons (SNNs) can solve artificial intelligence tasks with similar approaches to standard neural networks based on layers of artificial neurons, whether through supervised [115], unsupervised [116], or reinforcement learning [117]. Trained neural network models may be converted into SNNs to run inference on neuromorphic hardware, which is based on dedicated analog circuits and thus provides orders of magnitude higher energy efficiency than tensor processing accelerators for DNNs. More in detail, neuromorphic accelerators such as TrueNorth [71], Loihi [72], and SpiNNaker [70] are tile-based many-core architectures where processing cores exchange spike packets over a shared interconnect (Figure 5.3a). Each tile contains a processing core with neuron circuitry and synaptic storage, and a network interface to communicate with a Network-on-Chip (NoC) [118] or a Segmented Bus [119]; processing cores are often designed as a resistive crossbar (Figure 5.3b), i.e., synaptic cells are organized in a two-dimensional grid with neuron circuitry placed along bitlines and wordlines [120].


T00 T01 T02 T03 T04 T05 T06 T07
T08 T09 T10 T11 T12 T13 T14 T15
T16 T17 T18 T19 T20 T21 T22 T23
T24 T25 T26 T27 T28 T29 T30 T31
T32 T33 T34 T35 T36 T37 T38 T39
T40 T41 T42 T43 T44 T45 T46 T47
T48 T49 T50 T51 T52 T53 T54 T55
T56 T57 T58 T59 T60 T61 T62 T63



(a) *A many-core neuromorphic accelerator.* **(b)** *An analog crossbar-based neuromorphic processing core.*

Figure 5.3: *Tile-based neuromorphic accelerator and crossbar architecture.*

Compiling and mapping SNNs on neuromorphic hardware is supported by tools such as NEUTRAMS [121], SpiNeMap [122], Corelet [123], and NeuroXplorer [112]. These software frameworks usually include a compiler to partition an SNN into clusters and a run-time manager to allocate clusters to cores. Compilers ensure that each cluster fits onto a processing core of the chosen accelerator; run-time managers can be specialized to improve energy consumption [124], reliability [125], or device lifetime [126, 127].

Dataflow-based techniques are commonly used to analyze and partition an SNN model [128]. A network is represented as a directed graph $G_{SNN} = (N, S)$ with a finite set N of nodes representing neurons, and a finite set S of edges representing synapses between pairs of pre- and post-synaptic neurons (Figure 5.4, left). A dataflow compiler partitions the graph into clusters containing a subset of neurons and synapses of the SNN, obtaining a directed clustered graph $G_{CSNN} = (C, E)$ composed of a finite set C of clusters and a finite set E of edges between them (Figure 5.4, right). The partitioning algorithm takes into consideration the resource constraints of a synaptic core in a specific neuromorphic accelerator so that each cluster can fit in a tile. Moreover, compilers often use a variant of the Kernighan-Lin graph partitioning heuristic [129] to minimize communication between clusters and consequently reduce latency on the shared interconnect, where inter-cluster links are mapped [122, 130, 131].

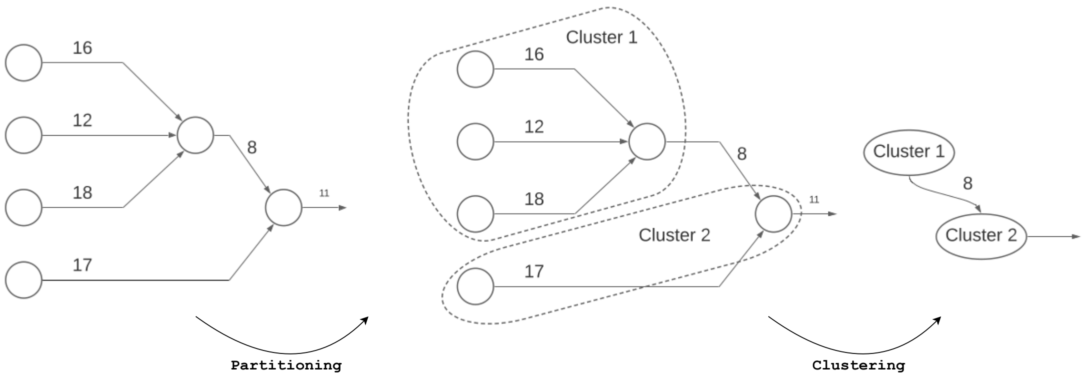


Figure 5.4: *SNN as a data flow graph partitioned into clusters.*

The clustered SNN graph can be represented as a synchronous data flow graph (SDFG), where each cluster is an actor and inter-cluster communication channels are edges; actors communicate by exchanging tokens on edges, which represent spikes. Multiple incoming and outgoing edges may exist between any pair of clusters. When a neuron in an actor fires, it generates tokens only on the output edges associated with that neuron; therefore, in the SDFG representation of a clustered SNN graph, tokens are generated on a subset of the output edges, rather than on all output edges as in a basic SDFG formalism.

5.2.1 The NeuroXplorer framework

NeuroXplorer [112] is a framework that can compile and simulate neural network models on neuromorphic hardware (Figure 5.5) using dataflow analysis techniques. Inputs to NeuroXplorer can be both artificial neural networks (ANN workloads, developed and trained in high-level frameworks such as Tensorflow and PyTorch) and biology-inspired SNNs. In the case of ANNs, the model is first converted into an SNN through the SNN Conversion unit. SNN models, instead, can be specified in PyNN [132] or PyCARL [133], which are Python interfaces to SNN simulators such as CARLsim [134], Brian [135], NEST [136], and Neuron [137].

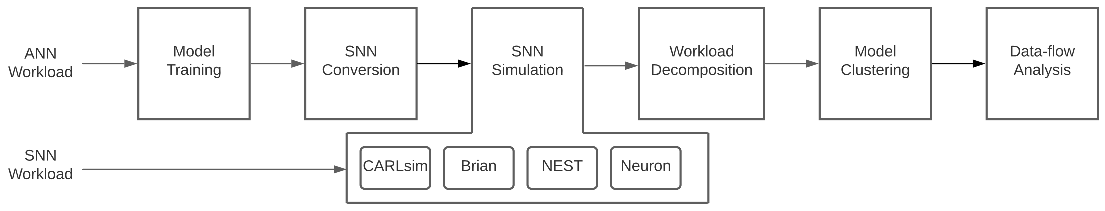


Figure 5.5: *SNN simulation through NeuroXplorer.*

NeuroXplorer uses simulators to feed representative training data to the input SNN and gather spike timing information that will be useful during compilation and mapping. The next step in the flow is workload decomposition, where the simulated SNN model is decomposed into fan-in-of-two (FIT) neural units to allow mapping on the processing cores of a neuromorphic device [138]; the decomposed SNN workload is then clustered aiming to minimize inter-cluster spike communication. Finally, NeuroXplorer uses dataflow analysis techniques to convert the clustered SNN graph into SDFG representation and estimate its performance for a given mapping of clusters to cores of the hardware [139]. Figure 5.6 illustrates how NeuroXplorer performed the mapping of SDFG actors (circles) generated from a LeNet model [140] converted into an SNN to the tiles of a neuromorphic device (rectangles).

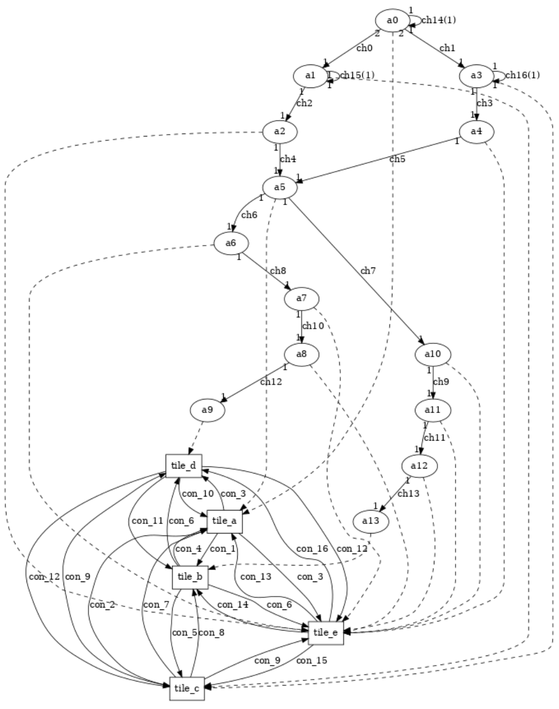


Figure 5.6: *Mapping of LeNet on neuromorphic hardware.*

5.3 Automated synthesis of SNN accelerators

This section discusses how the SODA Synthesizer can be extended and adapted to generate accelerators for SNNs, integrating features and lessons learned from the NeuroXplorer framework described in Section 5.2. The final goal is to exploit MLIR dialects and optimizations to translate a pre-trained ANN into an SNN, optimize it, and deploy it on a general-purpose processor for emulation, on existing neuromorphic devices, or on specialized hardware generated by the SODA Synthesizer. Section 5.3.1 describes the transformations involved in the SODA SNN deployment flow, while Section 5.3.2 dives into the details of a newly designed dialect to model SNNs.

5.3.1 SNN deployment flow

Figure 5.7 divides the steps needed to generate an SNN accelerator with the SODA Synthesizer into three groups, which correspond to three different development phases. In the first phase, the SODA-OPT frontend is modified to allow the conversion of pre-trained models to SNN and software emulation to gather spike timing data. The second and third phases represent two alternative deployment flows: one exploits the NeuroXplorer tools to program an existing neuromorphic chip, while the other synthesizes a custom hardware accelerator.

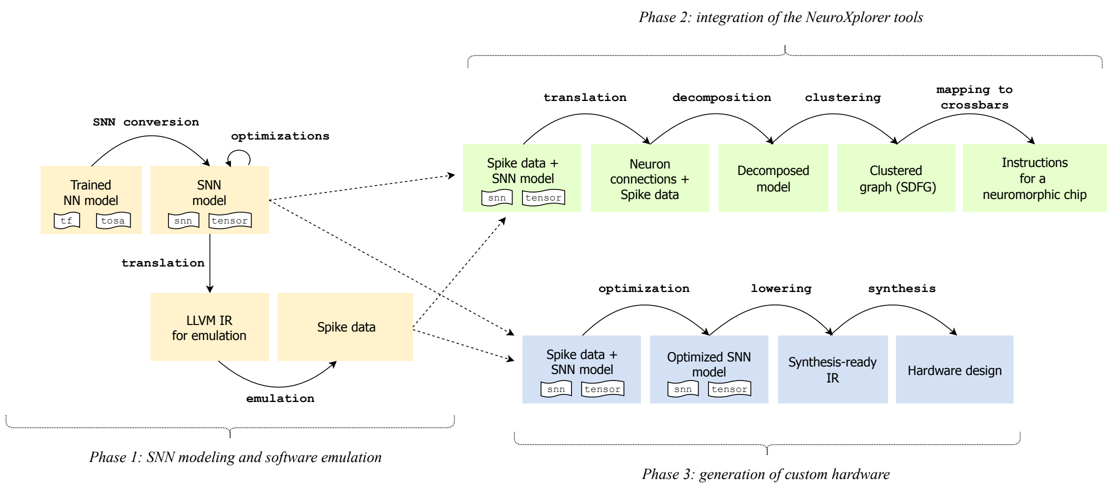


Figure 5.7: *Evolution of the SNN development flow in SODA.*

As in the baseline SODA design flow, the input is a neural network model trained in a software framework for ML and translated to MLIR, containing operations in high-level dialects such as TensorFlow (*tf*) or Tensor Operator Set Architecture (*tosa*). Any of the lowering and optimization passes that are already present in the SODA-OPT frontend can be applied, if beneficial, before lowering the IR to convert the model into a new domain-specific abstraction, the *SNN* dialect. The SNN conversion is the most relevant aspect of this first development phase, as it can impact the accuracy of the network: this motivates the development of a specific dialect that can represent neurons, spikes, and other SNN characteristics. A dedicated dialect also exposes further opportunities for optimizations that are specific to SNN models and not accessible to the NeuroXplorer design flow. Operations and types of the SNN dialect are then lowered and translated to LLVM IR for software execution, in order to emulate the behavior of the network on a test dataset, verify whether the initial accuracy was maintained within an acceptable error range, and collect data about spike times for all neurons in the network.

The second development phase aims at integrating NeuroXplorer in the design flow, providing an automated path to map DNN and SNN models to neuromorphic crossbar-based accelerators. The first steps of the flow in Figure 5.5 are substituted by the SNN conversion and software emulation within the SODA Synthesizer; the missing link before decomposition, clustering, and mapping is a translation from the MLIR SNN model into an IR compatible with NeuroXplorer, containing information about how neurons are connected in the network and timing of individual spikes gathered from the emulation phase.

Instead of relying on existing neuromorphic architectures, the third development phase aims at generating custom hardware designs, possibly combining analog and digital elements: digital neurons can be generated through the HLS backend, while analog components could be added as pre-existing IP blocks. In this phase, the SNN model generated by the frontend can undergo further optimizations taking into consideration both information about the target hardware and the spike data gathered during software emulation. In the existing SODA HLS flow, the SNN model would finally be translated into an LLVM IR and fed to Bambu HLS to generate Verilog code for ASIC/FPGA. For a hybrid architecture with neuromorphic elements, a new low-level IR may be needed to describe both analog and digital blocks, and the interfaces between them; the backend would then combine pre-existing IPs and custom logic to generate the design and integrate it into a full system with other accelerators generated by the SODA Synthesizer.

5.3.2 SNN dialect

The proposed SNN dialect implements an intermediate representation that captures intrinsic characteristics of SNN models. The dialect includes types to represent data exchanged between neurons during SNN execution and operations representing the transformations applied to the data. Figure 5.8 shows the key concepts that are captured in the dialect by depicting the operation of a LIF neuron: input spikes arrive at the neuron as spike trains; the neuron in turn, whenever its membrane potential crosses a threshold, produces output spikes that form a new train.

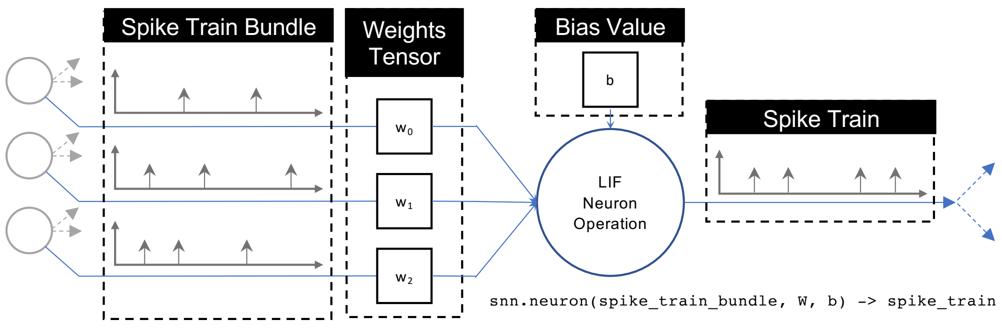


Figure 5.8: *The snn.neuron operation and key snn types.*

Key types in the SNN dialect are spike trains, i.e., lists of timestamps on which spikes occur, and N-dimensional bundles of spike trains (Figure 5.9). Conversions to and from the SNN domain are achieved through spike train conversion operations (Figure 5.10a), which generate a spike train from a tensor containing a list of times, or extract the list of timestamps from a spike train into a tensor.

Bulk conversions of input data can be achieved through spike bundle conversions (Figure 5.10b), which convert multiple tensors into multiple spike trains. Isolated spike trains can be grouped in a bundle, and a specific spike train can be extracted from a bundle (Figure 5.10c); the SNN dialect also implements additional operations to provide finer grained control on spike train bundles such as adding and removing a spike train, slicing a bundle into smaller bundles, and exchanging the ordering of spike trains in a bundle.

```
!spkt_t = snn.spike_train<?xf32>
!spk1db_t = snn.spike_bundle<?x snn.spike_train<?xf32>>
!spk2db_t = snn.spike_bundle<?x?x snn.spike_train<?xf32>>
```

Figure 5.9: *SNN dialect types.*

```
%spike_train = snn.encode_spike_train(%list_of_times : tensor<?xf32>) : !spkt_t
%list_of_times = snn.decode_spike_train(%spike_train : !spkt_t) : tensor<?xf32>
```

(a) *Spike train conversions.*

```
%spk_bundle = snn.tensor_to_spike_train_bundle(data : tensor<?x?xf32>) : !spk2db_t
%data = snn.spike_bundle_to_tensor(%sb : !spk2db_t ) : tensor<?x?xf32>
```

(b) *Spike bundle conversions.*

```
%spk_train = snn.spike_bundle_select (%spk_bundle[%i, %j] : !spk2db_t) : !spkt_t
%spk_bundle2 = snn.spike_bundle_collect (%spk : !spkt_t , %spk : !spkt_t , ...) :
    !spk1db_t
```

(c) *Spike train bundling and selecting.*

```
%id0 = snn.neuron (%in : !spkt_t) : !spkt_t // Input neuron
%id1 = snn.neuron (%in_spikes : !spk1db_t , W : tensor<?xf32> , bias : f32) :
     !spkt_t
```

(d) *Neuron operations.*

```
%d_order = snn.get_order(%out_spikes : !spk1db_t) : tensor<?xindex>
%r_order = snn.get_rev_order(%out_spikes : !spk1db_t) : tensor<?xindex>
```

(e) *Ordering operations.*

Figure 5.10: *SNN dialect operations.*

The core operation in the SNN dialect is the neuron operation (Figure 5.10d), which models the behavior of a LIF neuron as the one depicted in in Figures 5.1 and 5.8. A neuron operation multiplies input spike trains by a set of weights representing the LIF neuron resistance R to produce an output spike train; intrinsic neuron attributes include the threshold voltage V_{th}, the resting potential V_{rest} which can be modified through the bias value, and a multiplying constant k that affects the neuron discharge rate. Ordering operations (Figure 5.10e) create a tensor with the direct (or reverse) order of which spike trains spiked first; they are used to evaluate which category is more important at the output of an SNN model used to solve a classification problem.

Figure 5.11 shows an example of a 2-input, 2-output fully connected SNN model described using the SNN dialect. The example demonstrates how the SNN model interfaces with digital inputs represented as tensors by using spike train conversions, spike bundle conversions, and ordering operations, as well as how spike trains are manipulated in the SNN domain to represent the fully connected model.

```
module attributes {soda.snn.container_module, snn.lif.k=1,
                   snn.lif.vth=30, snn.lif.vrest=-20}{
  !spkt_t = snn.spike_train<?xf32>
  !spk1db_t = snn.spike_bundle<?x snn.spike_train<?xf32>>

  func @dense_2(%data: tensor<2xf32> ,
                %W : tensor<2x2xf32> ,
                %B : tensor<2xf32>) -> tensor<2xindex>){
    // Use tensor.extract_slice and tensor.extract ops
    //   to collect %w_1_0, %w_1_1, %b_1_0, %b_1_1
    //   from %W and %B input arguments.

    // Input Layer
    %in = snn.tensor_to_spike_train_bundle(%data : tensor<2xf32>) : !spk2db_t
    %in_0 = snn.spike_bundle_select (%in[0] : !spk1db_t) : !spkt_t
    %in_1 = snn.spike_bundle_select (%in[1] : !spk1db_t) : !spkt_t
    %v_0_0 = snn.neuron (%in_0 : !spkt_t)
    %v_0_1 = snn.neuron (%in_1 : !spkt_t)
    %v_0 = snn.spike_bundle_collect (%v_0_0 : !spkt_t , %v_0_1 : !spkt_t) : !spk1db_t

    // Hidden/Output Layer
    %v_1_0 = snn.neuron (%v_0 : !spk1db_t , %w_1_0 : tensor<2xf32> , %b_1_0 : f32 )
    %v_1_1 = snn.neuron (%v_0 : !spk1db_t , %w_1_1 : tensor<2xf32> , %b_1_1 : f32 )
    %v_1 = snn.spike_bundle_collect (%v_1_0 : !spkt_t , %v_1_1 : !spkt_t) : !spk1db_t

    // Output transformation
    %order = snn.get_order(%out_spikes : !spk1db_t) : tensor<?xindex>
    %out = tensor.cast %order : tensor<?xindex> to tensor<2xindex>
    return %out
  }
}
```

Figure 5.11: *Two-neuron fully connected model described in MLIR.*

In addition to the types and operations previously described, Figure 5.11 also includes important module attributes associated with LIF neurons characteristics shared by all neurons in a given target chip, i.e., the firing threshold and resting potential in mV, and the discharge rate multiplying constant. These values must be known to perform SNN emulation, and they can guide domain-specific low-level optimizations.

The example contains representations from two domains, a purely digital domain captured by tensors and the SNN domain abstracted by types and operations in the SNN dialect but ultimately governed by the analog behavior of LIF neurons. Neuromorphic devices are based on resistive circuits, but they can be simulated through digital logic (e.g., on FPGA); the SNN dialect could then be lowered and synthesized through HLS as an initial step in the integration flow, followed by the addition of dedicated mixed-circuit designs in the resource library for the generation of hybrid systems with both traditional and spiking accelerators.

5.4 Conclusion

This chapter discussed the opportunities to integrate conventional artificial neural network techniques and spiking neural network elements through the proposed modern multi-level hardware compiler. An initial approach was presented for the integration of the NeuroXplorer framework with the SODA Synthesizer. Leveraging the MLIR framework, a new specialized IR (the SNN dialect) was introduced to deal with the complexities related to SNN representation and mapping. The SODA Synthesizer can generate Verilog code representing spiking neurons in digital logic to be mapped onto FPGAs or ASICs; in the future, the available compiler-based infrastructure could allow to define and integrate ML accelerators composed of both digital and analog components in a complex heterogeneous system.

CHAPTER 6

Experimental results

THIS chapter presents experimental results obtained through the modern MLIR-based approach to HLS described in previous chapters; the effectiveness of the proposed approach is demonstrated by automatically generating ASIC and FPGA accelerators for polyhedral benchmarks and layers of popular CNN models. The MLIR implementation of loop pipelining produces accelerators on average 2.1x faster when used together with Bambu and 1.85x faster with Vitis HLS, demonstrating the advantages it can provide in terms of performance and portability. High-level MLIR optimizations result in up to 74x speedup on isolated accelerators for individual CNN layers, and the dynamically scheduled dataflow architecture yields an additional 3x performance improvement when combining accelerators to handle streaming inputs.

6.1 Overview

The proposed multi-level approach to HLS improves productivity for users and developers of HLS tools by providing convenient levels of abstraction for the introduction of new optimizations, removing the need for low-level hardware synthesis expertise, and providing compiler-based exploration of different design points. Moreover, it also provides measurable improvements in the QoR of generated accelerators and enables portability across different HLS backends. This chapter focuses on demonstrating the latter aspects through a series of experiments on polyhedral and ML benchmarks; in particular, the experiments described in this chapter were designed to answer the following questions:

- Is MLIR-based loop pipelining effective in improving accelerator performance? (Section 6.2.1)
- Are MLIR high-level optimizations portable across HLS tools? (Sections 6.2.2 and 6.2.4)
- Can the proposed loop pipelining pass be combined with other high-level transformations? (Section 6.2.3)
- Is the high-level optimization pipeline in SODA-OPT effective in improving accelerator performance? (Sections 6.2.4 and 6.3.1)
- Can the SODA Synthesizer generate efficient ML accelerators? (Sections 6.3.1 and 6.3.2)
- Is the proposed dataflow architecture more efficient than a centralized architecture driven by a microcontroller? (Section 6.3.3)

6.2 Experiments on polyhedral benchmarks

This section presents experiments on parts of the PolyBench benchmark suite [141], which provides a collection of kernels containing deeply nested loops with floating-point arithmetic operations on multi-dimensional tensors. The benchmark suite is mostly used to test polyhedral optimization techniques, and it contains examples of representative computation patterns that are common in high-level programming frameworks for scientific computing and machine learning.

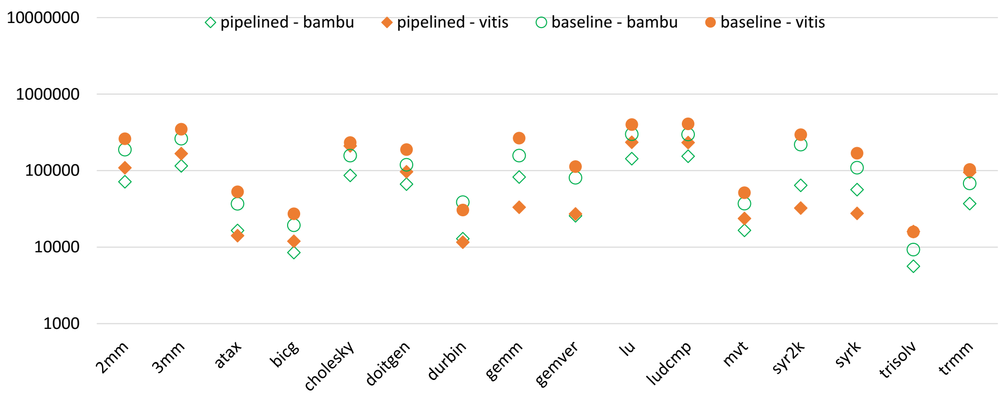
Kernels written in C were taken directly from version 4.2.1 of the benchmark, while their MLIR counterparts were provided by Polygeist [142]. To test the flexibility of the proposed tools and techniques, the kernels were synthesized with a variety of data types and kernel sizes; this also allowed to keep simulation times and resource consumption reasonably low by selecting smaller loop bounds for floating-point experiments.

6.2.1 Effectiveness of loop pipelining

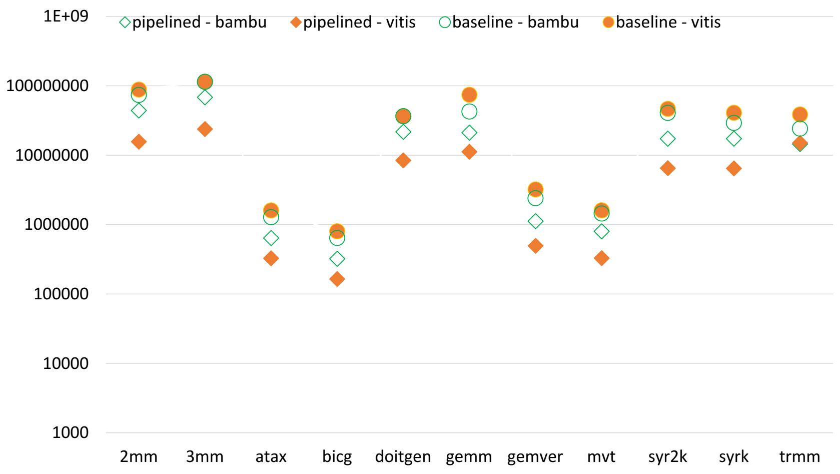
The MLIR implementation of loop pipelining presented in Chapter 3 has the greatest impact on the QoR of accelerators generated by Bambu, since Bambu would not otherwise support loop pipelining. This first set of experiments thus compares the performance of accelerators generated by Bambu through MLIR-based loop pipelining against the performance of accelerators generated by Vitis HLS through loop pipelining triggered by pragmas. Such experiments (partially discussed in [89] and [90]) validate how effective the proposed loop pipelining implementation is in reducing the execution latency of loops by issuing multiple iterations in parallel.

To this end, the standard C version of PolyBench kernels was synthesized with Vitis HLS 2021.1, adding specific directives that disable other loop optimizations (e.g., unrolling, flattening) and turn on/off loop pipelining to obtain baseline and pipelined accelerators. The MLIR affine versions of Polybench kernels were translated into LLVM IR and synthesized with Bambu 0.9.7, first without any optimization and then applying the proposed high-level loop pipelining transformations. Concerning scheduling options (Section 3.2.2), different HatSchet configurations were tested to conclude that, on PolyBench, the ILP-based modulo scheduling algorithm [143] produces the best result in an acceptable amount of time. Similarly, it was 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. With both Vitis HLS and Bambu, the target was a Xilinx Zynq-7000 FPGA at 100 MHz frequency; inputs and outputs were assumed to be stored in on-chip BRAMs. Performance is measured through simulation, resource utilization is reported post place-and-route.

Figure 6.1 shows the execution time in clock cycles of the generated accelerators, with two versions of PolyBench kernels: double-precision floating-point operations on the ‘mini’ dataset size, and 32-bit integer operations on the ‘medium’ dataset size. (Kernels in the ‘solvers’ category are only meant to be executed in floating-point, so they are not present in the integer experiments.) Pipelining loops provides a significant reduction in clock cycles, as expected; this is verified both when pipelining is applied



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



(b) *32-bit integer, medium dataset.*

Figure 6.1: *Execution times with different HLS tools and loop pipelining strategies.*

within the HLS tool and when it is implemented as a high-level MLIR optimization. After scheduling the MLIR affine code, all operations in the new loop body are independent and can be executed in parallel; this means that measuring the II corresponds to measuring the execution latency of one loop iteration, i.e., the latency of the slowest operation in the loop body. By inspecting the finite state machines generated by Bambu and the Vitis HLS synthesis logs, it is possible to observe that the II reported by Vitis HLS is always higher than the iteration latency for the loop synthesized by Bambu in the floating-point experiments; in general, on double-precision computations Bambu tends to perform better than Vitis HLS because the floating-point functional units in the Bambu resource library are faster. For the integer experiments, instead, Vitis HLS can always reach an II of one, which Bambu sometimes fails to obtain despite always receiving an input IR with no dependencies between operations in the innermost loop.

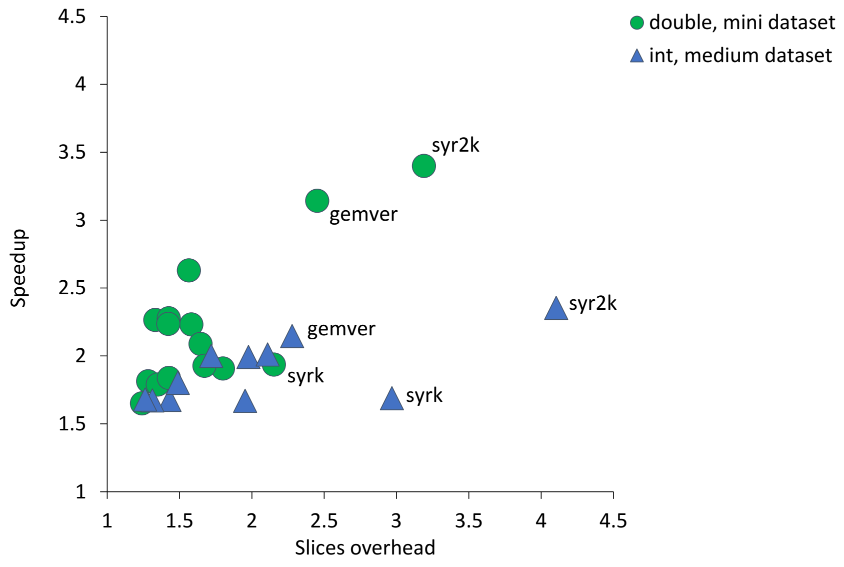


Figure 6.2: *Loop pipelining performance improvement and area overhead.*

Pipelining loops increases FPGA resource consumption (i.e., digital signal processing elements, registers, look-up tables, and slices) to accommodate the extra states for prologue and epilogue, and instantiate larger multiplexers in front of each functional unit; however, by comparing 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 6.2 visualizes this trend by plotting the performance increase with respect to the overhead in slice utilization in the experiments with Bambu and MLIR-based loop pipelining. While most benchmarks are clustered in the bottom left corner, some kernels show a disproportionate area overhead (labeled points in the graph). For example, 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 as described in Section 3.4.4. 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.

6.2.2 Portability of MLIR-based loop pipelining

An advantage of implementing software pipelining as an MLIR transformation is that it does not require annotations for a specific backend HLS tool: after applying the newly introduced passes, the code contains the pipelined loop in the MLIR affine dialect, and after lowering and translating it only contains standard LLVM code. The generated LLVM IR can be synthesized through recent versions of Vitis HLS [38] by setting up a compilation flow that bypasses the C/C++ frontend to feed the LLVM IR directly

Table 6.1: *QoR of pipelined accelerators generated through different HLS tools.*

Kernel	Version	Tool	Cycles	Slices	Speedup	Slices overhead
gemm	double, mini	Bambu	82 362	1303	1.91x	1.80x
gemm	double, mini	Vitis HLS	206 821	1119	1.29x	2.54x
gemm	int, medium	Bambu	21 160 402	506	2.01x	2.11x
gemm	int, medium	Vitis HLS	31 764 201	322	2.34x	3.39x
syr2k	double, mini	Bambu	64 249	2826	3.40x	3.19x
syr2k	double, mini	Vitis HLS	137 463	1736	2.14x	2.71x
syr2k	int, medium	Bambu	17 285 566	1030	2.35x	4.10x
syr2k	int, medium	Vitis HLS	28 691 013	794	1.62x	5.79x

into the closed-source backend. (After applying the same MLIR pipeline used in the Bambu experiments, a final LLVM pass was needed to “downgrade" the IRs, as Xilinx tools work with an old LLVM version that is not compatible with MLIR; code for the pass was provided in the Phism project [144].) Table 6.1 reports post place-and-route results (partially presented in [90]) for two pipelined accelerators synthesized by Bambu and by Vitis HLS, one with constant and one with variable loop bounds. Speedups and overheads are calculated with respect to a baseline accelerator derived from an LLVM IR translated from MLIR code without loop pipelining; the target was again a Zynq-7000 board at 100 MHz frequency and performance results are obtained through simulation.

The introduction of loop pipelining as an MLIR high-level optimization positively affected accelerator performance through both the Bambu and the Vitis HLS backend in these experiments. While Bambu natively supports LLVM IRs as input, the Vitis HLS compilation and synthesis flow for LLVM is rather experimental: Vitis HLS is optimized primarily for annotated C/C++ code; existing documentation describes the use of pragmas in C/C++, and it is not trivial to understand if and how these would need to be added in the generated LLVM IR. Nevertheless, results such as the ones presented in this section provide sufficient motivation to further explore synthesis-oriented transformations in MLIR that can benefit multiple HLS backends.

6.2.3 Loop pipelining and other high-level transformations

The modularity and flexibility of a multi-level approach to HLS based on MLIR allow to easily introduce new optimizations in the compilation pipeline with the aim of generating optimized IRs for hardware synthesis, and, if beneficial, to reuse existing passes that were originally designed for a different purpose (e.g., generic compiler passes for software or transformations for a specific hardware architecture). The affine dialect provides a growing collection of transformations that can be explored in the context

Table 6.2: *Effect of affine optimizations on gemm (double, mini).*

Optimizations	Cycles	Slices	Speedup	Slices overhead
none	157 122	724	baseline	baseline
loop pipelining	82 362	1303	1.91x	1.80x
loop permutation + pipelining	81 182	1306	1.93x	1.80x
loop unrolling + pipelining	17 642	8075	8.91x	11.15x

of a frontend to HLS to optimize loop structures; the introduction of loop pipelining as a high-level pass operating on the affine dialect thus also allows to combine it with other affine optimizations. As was demonstrated in previous sections, loop pipelining provides performance benefits on its own, but it can also be coupled with other optimizations to explore design points with different performance/area trade-offs. Even if some of the techniques implemented by the affine dialect are also available as backend HLS optimizations triggered by pragmas, applying them at the MLIR level allows to decouple loop optimizations (which do not necessarily require hardware-related considerations) from the backend HLS tool, and thus enhance portability.

Table 6.2 reports simulated execution time and post place-and-route slices consumption of accelerators obtained by coupling loop pipelining with a few other passes on the generalized matrix multiplication (gemm) kernel with the Bambu backend, keeping the same experimental setup of the previous sections. The results (partially discussed in [90]) show 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 an almost nonexistent increase in resource utilization. If the size of the loop body is increased through unrolling, instead, it is possible to obtain an even faster design at the cost of significant area consumption. The same exploration of design points would require manual modifications to 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 such as loop permutation. In an MLIR-based framework, instead, all of them are exposed as compiler passes and compiler options, reducing the risk of errors and the time needed for DSE.

6.2.4 Effects of high-level optimization pipelines on performance

The SODA-OPT frontend (Section 3.5) provides a default optimization pipeline built of MLIR passes that restructure input IRs in order to improve their performance when synthesized into hardware through HLS; in this regard, it can be compared to ScaleHLS [27], which is also an MLIR-based tool providing automated optimizations for HLS. This section presents results obtained with SODA-OPT and ScaleHLS on PolyBench kernels, discussing the strengths and weaknesses of the two tools in light of the absolute execution times and relative performance improvements they achieve (as was also partially done in [15]). The experiments were run on a version of the kernels that operates on single-precision floating-point data types and with four different kernel sizes (referring to the size of all dimensions of the involved tensors); the hardware target is a Xilinx Virtex7 FPGA with 100 MHz frequency.

Table 6.3 reports execution times and highlights for every kernel and every size which is the configuration that resulted in the lowest number of clock cycles after simulation. Before drawing a direct comparison between SODA-OPT and ScaleHLS, an important observation must be made on the different backends that the two tools support: SODA-OPT supports Bambu and Vitis HLS (through its LLVM frontend), ScaleHLS only supports Vivado HLS, which is an older version of the Xilinx HLS tool. To account for differences in performance that derive from capabilities of the backends rather than optimizations in the frontends, the table reports separate baselines: synthesis of unoptimized LLVM IR with Bambu 0.9.7, synthesis of unoptimized C with Vitis HLS 2021.1, and synthesis of unoptimized C with Vivado HLS 2019.2. Errors sometimes occurred when ScaleHLS produced annotated C++ code that was correctly synthesized by Vivado HLS, but later failed in the place-and-route phase: ScaleHLS performs DSE by analyzing high-level IRs and early HLS estimates, which may lead to an underestimation of resource consumption; when this happens, designs produced by ScaleHLS cannot conclude place-and-route because they require more resources than the ones available in the target FPGA.

Looking at absolute numbers of clock cycles, SODA-OPT outperforms ScaleHLS in 26 kernels out of 36, through either the Bambu or the Vitis HLS backend. The SODA-OPT default optimization pipeline is particularly well suited to kernels with dot product or matrix multiplication structures (e.g., it provides 66.38x performance increase on *2mm* and 50.43x on *gemm*); its effect is more limited, instead, on kernels that contain irregular loop structures such as *syr2k*. A regular structure also benefits the default optimization process in baseline Vitis HLS designs, which for example surpasses the performance of designs optimized by SODA-OPT for smaller versions of the *doitgen* kernel. Similarly, it is reasonable to assume that it is the irregular structure of *syr2k* loops that causes ScaleHLS to generate accelerators with lower performance than baseline Vivado HLS designs. *Atax* designs generated by ScaleHLS, instead, encounter an error during IP export for logic synthesis, probably due to an error in the synthesis of accelerator interfaces.

On average, both ScaleHLS and SODA-OPT for Bambu provide considerable performance improvements over the respective baselines, while the performance improve-

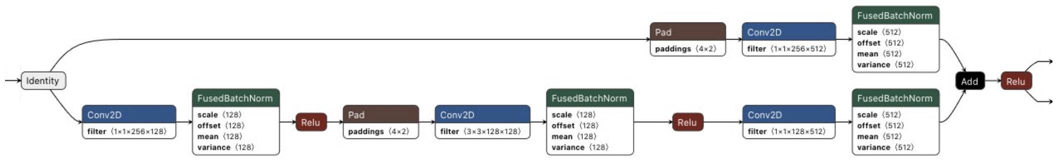
Table 6.3: *Execution times of accelerators optimized with different frontend tools.*

Kernel	Frontend	Backend	Size 2	Size 4	Size 8	Size 16	Avg. speedup
2mm	none	Bambu	176	1375	11218	87842	66.38
2mm	SODA-OPT	Bambu	25	43	98	784	
2mm	none	Vitis HLS	43	115	599	4239	3.67
2mm	SODA-OPT	Vitis HLS	26	48	106	848	
2mm	none	Vivado HLS	162	1138	9698	75586	72.94
2mm	ScaleHLS	Vivado HLS	38	63	114	410	
3mm	none	Bambu	220	1743	14042	111410	35.24
3mm	SODA-OPT	Bambu	22	40	320	2560	
3mm	none	Vitis HLS	37	109	593	4233	3.73
3mm	SODA-OPT	Vitis HLS	23	45	103	824	
3mm	none	Vivado HLS	207	1467	12723	99939	54.86
3mm	ScaleHLS	Vivado HLS	57	97	169	797	
atax	none	Bambu	76	299	1171	4643	15.38
atax	SODA-OPT	Bambu	21	34	60	157	
atax	none	Vitis HLS	37	58	125	361	1.77
atax	SODA-OPT	Vitis HLS	22	39	74	163	
atax	none	Vivado HLS	66	242	1058	4162	n.a.
atax	ScaleHLS	Vivado HLS	Error	Error	Error	Error	
bicg	none	Bambu	73	294	1162	4626	18.69
bicg	SODA-OPT	Bambu	13	28	45	141	
bicg	none	Vitis HLS	27	47	121	339	2.33
bicg	SODA-OPT	Vitis HLS	12	22	47	143	
bicg	none	Vivado HLS	33	121	529	2081	12.18
bicg	ScaleHLS	Vivado HLS	22	31	49	64	
doitgen	none	Bambu	165	2486	38986	344338	24.20
doitgen	SODA-OPT	Bambu	15	166	1158	9624	
doitgen	none	Vitis HLS	25	100	590	57602	2.20
doitgen	SODA-OPT	Vitis HLS	33	146	1090	8443	
doitgen	none	Vivado HLS	161	2361	35025	541473	217.75
doitgen	ScaleHLS	Vivado HLS	14	26	94	1367	
gemm	none	Bambu	103	794	6538	42514	50.43
gemm	SODA-OPT	Bambu	16	28	71	568	
gemm	none	Vitis HLS	24	52	140	5635	6.78
gemm	SODA-OPT	Vitis HLS	15	29	71	259	
gemm	none	Vivado HLS	99	669	5593	42801	43.29
gemm	ScaleHLS	Vivado HLS	19	27	56	Error	
gemver	none	Bambu	154	606	2377	9620	20.10
gemver	SODA-OPT	Bambu	36	49	75	300	
gemver	none	Vitis HLS	73	117	251	715	2.39
gemver	SODA-OPT	Vitis HLS	39	56	90	255	
gemver	none	Vivado HLS	142	512	2076	8116	14.87
gemver	ScaleHLS	Vivado HLS	61	87	103	261	
mvt	none	Bambu	74	290	1155	4611	19.47
mvt	SODA-OPT	Bambu	13	21	45	141	
mvt	none	Vitis HLS	34	63	147	399	2.90
mvt	SODA-OPT	Vitis HLS	12	22	47	143	
mvt	none	Vivado HLS	66	242	1058	4162	13.78
mvt	ScaleHLS	Vivado HLS	36	62	72	120	
syr2k	none	Bambu	99	706	4834	35650	3.82
syr2k	SODA-OPT	Bambu	19	270	1417	8835	
syr2k	none	Vitis HLS	97	367	2627	18179	4.90
syr2k	SODA-OPT	Vitis HLS	50	159	509	1785	
syr2k	none	Vivado HLS	73	265	1089	4225	0.73
syr2k	ScaleHLS	Vivado HLS	93	353	1665	Error	

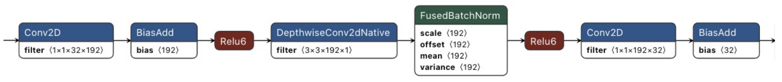
ment is smaller when comparing SODA-OPT for Vitis HLS against the Vitis HLS baseline. This is however not surprising, as Vitis HLS is the only backend tool that applies loop optimizations even in absence of user directives; accelerators produced by Vitis HLS are thus significantly faster than the other baseline designs, and the optimizations introduced by SODA-OPT provide only a slight improvement over the default ones applied by Vitis HLS. The optimizations introduced by ScaleHLS greatly improve accelerator performance with respect to baseline designs synthesized by Vivado HLS; however, not supporting Vitis HLS puts ScaleHLS at a disadvantage, as the pragma syntax changed enough between the two versions that annotated C++ code for Vivado HLS is not guaranteed to maintain the same performance when synthesized with Vitis HLS. The MLIR-based approach of SODA-OPT, instead, transforms the input code before feeding it to the HLS tool ensuring portability across different backends; by not relying on pragma annotations, SODA-OPT obtains designs with performance that does not depend on whether the backend HLS tool recognizes optimization directives or not.

6.3 Experiments on neural network models

This section reports experimental results obtained by generating neural network accelerators through the end-to-end design flow of the SODA Synthesizer, as also presented in [98]. The kernels used for the experiments are convolutions, batch normalization layers, and activation functions from ResNet and MobileNet CNN models in 32-bit floating-point (Figure 6.3), implemented and trained in TensorFlow and later translated into high-level built-in MLIR dialects through `tf-opt` and `tf-mlir-translate`. Each kernel is synthesized in isolation at first and then composed together with the others to evaluate the difference between the two available system architectures (i.e., centralized SoC and dataflow accelerator).



(a) *Operators from ResNet50.*



(b) *Operators from MobileNetV2.*

Figure 6.3: *DNN operators used in the experiments of Section 6.3.*

All experiments target the generation of ASIC accelerators with 45 nm technology through Bambu 0.9.7 and the OpenRoad flow, with an operating frequency of 500MHz; each synthesized kernel has two ports connecting it to shared memory with 2 cycles of read latency, and 1 cycle of write latency. The SODA Synthesizer is also able to generate different solutions for memory interfacing, with dedicated load/store ports for each kernel argument and a parametric number of load/store ports; however, these experiments only employ two ports per kernel to limit the growth of the HMI complexity when they are subsequently combined in the dataflow design (Section 4.4). Memory parallelism is achieved in multi-accelerator system architectures through the concurrent operation of interconnected FSMD modules.

6.3.1 QoR of generated designs

For the experiments described in this section, individual operators from Figure 6.3 were automatically outlined and synthesized in two different configurations to assess the impact of the SODA-OPT high-level optimization pipeline on ASIC accelerators for ML. In the baseline configuration, each operator is outlined, lowered without applying optimization passes, translated to LLVM IR, and synthesized. In the optimized configuration, the SODA-OPT default optimization pipeline is introduced with the goal of reducing execution time.

As discussed in Sections 3.5 and 4.3, SODA-OPT automatically optimizes IRs by exploiting existing and custom MLIR passes to increase instruction-level and data-level parallelism and reduce the number of redundant instructions. The transformations allow Bambu to compute more efficient schedules and better leverage the available hardware resources, resulting in shorter execution times. For large and arithmetically intensive kernels such as convolutions, SODA-OPT can perform loop tiling to balance computation and memory transfer and apply the optimization pipeline to the inner tile, obtaining a smaller accelerator that is invoked multiple times to execute a long operation. In that case, the execution time is measured by multiplying the latency of the accelerated tile by the number of invocations needed to complete the computation.

Table 6.4 shows post floorplanning characteristics of the optimized version of neural network accelerators generated by the SODA Synthesizer. The efficiency in GFLOPS/W is calculated by counting the total number of floating point arithmetic operations (mostly multiplications and sums) performed during the execution of a kernel, and dividing the number by the latency and power consumption reported by OpenROAD after floorplanning. All the generated accelerators reach high energy efficiency, with power consumption ranging from 20 to 440 mW; simple operators such as ReLU achieve up

Table 6.4: *QoR of DNN accelerators synthesized with high-level optimizations.*

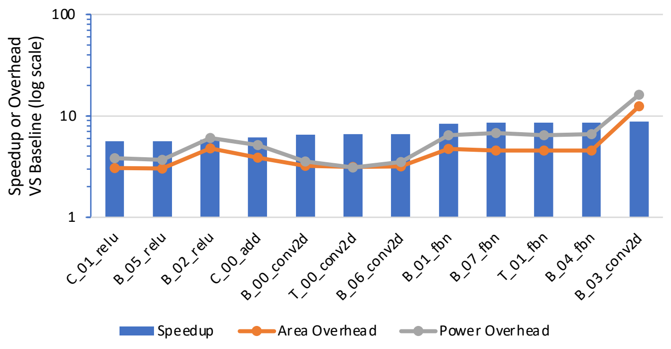
		ResNet50		
Operator	**Cycles**	**Area(um^2)**	**Power(W)**	**GFLOPS/W**
B_00_conv2d	2,554,953,728	175,874	0.069	10.3
B_01_fbn	25,619,335	662,899	0.042	19.1
B_02_relu	3,353,684	70,949	0.032	141.3
B_03_conv2d	2,860,150,784	517,396	0.237	6.2
B_04_fbn	6,602,595	639,438	0.042	19.7
B_05_relu	870,770	48,977	0.021	185.6
B_06_conv2d	1,277,263,872	173,603	0.069	10.8
B_07_fbn	26,395,323	638,947	0.044	19.2
C_00_add	5,724,970	78,439	0.0378	17.8
C_01_relu	3,480,074	49,253	0.0217	183.0
T_00_conv2d	2,552,758,272	174,580	0.0627	11.6
T_01_fbn	26,395,323	638,929	0.042	19.5

B (bottom), T (top) and C (centered) refer to the branches of Figure 6.3.

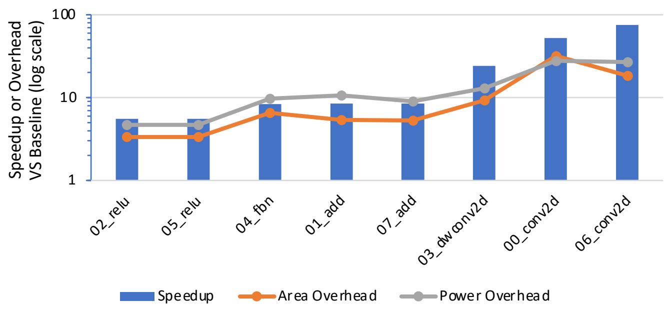
		MobileNetV2		
Operator	**Cycles**	**Area(um^2)**	**Power(W)**	**GFLOPS/W**
00_conv2d	6,058,752	1,281,674	0.440	7.2
01_add	707,350	83,958	0.049	8.7
02_relu	648,214	42,050	0.023	82.2
03_dwconv2d	3,622,080	407,501	0.204	7.3
04_fbn	3,468,402	758,623	0.055	7.5
05_relu	648,214	42,050	0.023	82.2
06_conv2d	4,246,144	724,024	0.383	11.8
07_add	117,910	81,636	0.041	62.1

to hundreds of GFLOPS/W efficiency, while more complex operators such as convolutions reach $\sim$10 GFLOPS/W. Such disparity is a consequence of the SODA-OPT optimization strategy, which increases the number of computational units in parallel and therefore produces large accelerators with increased power consumption; smaller kernels, even if completely unrolled, consume less power than larger kernels and reach higher efficiency.

Figure 6.4 compares the performance, area, and power consumption of accelerators obtained with the baseline and optimized configurations. By enabling the SODA-OPT high-level optimization pipeline, the execution time in clock cycles decreases, obtaining an average speedup of 7.2x over baseline results for ResNet50 operators, and an average speedup of 23.5x with peaks of $\sim$52-74x in the convolutional layers for MobileNetV2 operators. In all cases, introducing the SODA-OPT pipeline results in a trade-off between performance and area/power consumption, with power and area overheads increasing proportionately to the obtained speedup. This effect is to be ex-



(a) *ResNet50 operators.*



(b) *MobileNetV2 operators.*

Figure 6.4: *Speedup, area overhead, and power overhead introduced by SODA-OPT.*

pected, as the default version of the SODA-OPT optimization pipeline aims at reducing execution time by generating bigger designs with more resources that can operate in parallel (especially through loop unrolling). Application developers that require a different trade-off between performance, area, and power consumption metrics can easily modify the default pipeline by enabling, disabling, or modifying available passes to obtain accelerators with the desired QoR.

6.3.2 Qualitative comparison with ML accelerators and design flows

Countless commercial and academic designs have been proposed to accelerate the execution of DNN models such as the ones considered in the previous set of experiments, with degrees of generality and efficiency that vary, depending on application requirements, from ultra-low-power wearable devices to cloud servers. A one-to-one comparison between accelerators generated by the SODA Synthesizer and state-of-the-art designs is complicated by the different technology nodes, operating frequencies, and degree of programmability; nevertheless, this section attempts a qualitative comparison of energy efficiency between matrix multiplication accelerators generated by the SODA Synthesizer and metrics for a selection of DNN accelerators as reported in [145].

Table 6.5: *Comparison among DNN accelerators.*

Floating-point accelerators

Platform	Technology node	Precision	Power (W)	Clock (MHz)	GFLOPS/W	Notes
V100 GPU	12 nm	FP32	300	1246	52.33	Theoretical peak
A100 GPU	7 nm	FP32	400	1410	48.75	Theoretical peak
TPU v3	16 nm	FP32	450	940	8.89	Theoretical peak
SODA MatMul	45 nm	FP32	0.42	500	17.46	Derived from execution time

Accelerators with different numerical formats

Platform	Technology node	Precision	Power (W)	Clock (MHz)	GOPS/W	Notes
A100 GPU	7 nm	TF32	400	1410	780	Theoretical peak
TPU v3	12 nm	FP16	450	940	273.33	Theoretical peak
TPU v4	7 nm	BF16	175	N/A	1432.29	Theoretical peak
SIGMA Sparse	28 nm	FP16	22.3	500	480	Average across GEMMs
DNNBuilder	20 nm	Fixed16	22.9	235	90.2	Batch execution of VGG
SODA MatMul	45 nm	Fixed16	0.05	500	162.25	Derived from execution time

The evaluated MatMul accelerator was lowered from an MLIR matrix multiplication operation on single-precision floating-point in the linear algebra dialect, optimized through SODA-OPT, and synthesized by Bambu with 8 parallel memory ports for an ASIC target with the 45 nm OpenROAD backend; the same process was repeated for a version of the kernel operating on 16-bit fixed-point data. The SODA Synthesizer generates FSMD accelerators where the number of functional units depends on the amount of exposed parallel arithmetic operations in the kernel (which is controlled by high-level optimizations), while other devices based on systolic arrays (e.g., TPUs) contain thousands of processing units that may or may not be fully utilized depending on the operation that is mapped on them. This is the main factor that prevents quantitative comparisons: programmable devices such as GPUs and TPUs advertise peak execution rates that may or may not be achieved depending on how optimized is the input code; instead, accelerators generated through tools and techniques proposed by this thesis are extremely specialized and always execute the same operation (e.g., matrix multiplication, matrix-vector multiplication, or ML operators and models). In fact, the G(FL)OPS/W efficiency results reported in Table 6.5 are all calculated from theoretical peak throughput, power consumption, and frequency, except for the accelerators generated by SODA where the actual throughput and efficiency are calculated based on their latency.

Another challenge for qualitative comparisons is the support for different numerical formats: as the accuracy of DNN models has been proven to be resilient to the loss of precision in weights and computation, DNN accelerators often employ specialized data types that trade off accuracy for efficiency, including ML-specific floating-point

Table 6.6: *Comparison among acceleration toolchains.*

Toolchain	Input	Backend tools	F1	F2	F3	F4	F5
ScaleHLS	MLIR	Vivado HLS	✓	✗	✓	✓	✗
CIRCT HLS	MLIR	CIRCT	✓	✗	✗	✗	✓
hls4ml	DNN model	Multiple HLS tools	✓	✗	✗	✓	✗
FINN	DNN model	Vivado HLS	✓	✗	✗	✓	✗
HeteroCL	Python	Intel or Vitis HLS	✗	✗	✗	✓	✗
POLSCA	C/C++	Vitis HLS	✓	✗	✗	✓	✗
SODA	MLIR	Bambu, Vitis HLS	✓	✓	✓	✓	✓

F1: Optimizations without manual code rewriting; **F2**: Partitioning of host and kernel code; **F3**: Design space exploration; **F4**: FPGA support; **F5**: ASIC support.

formats (e.g., bfloat16 in recent versions of Google TPUs and tensorfloat32 in Nvidia Tensor Cores). FPGA-based custom accelerators, instead, typically focus on integer/-fixed point formats because implementing floating-point units on fine-grained reconfigurable devices is considered to be inefficient; for example, DNNBuilder targets a Kintex UltraScale FPGA and leverages a fixed-point 16-bit format. The modular and extensible nature of the SODA Synthesizer would easily allow to introduce passes that parse quantized models and generate functional units with specialized number formats.

In conclusion, matrix multiplication accelerators generated by the SODA Synthesizer present an energy efficiency $sim15$ GFLOPS/W on floating-point and $sim160$ on fixed-point computation; a raw performance comparison with existing ML accelerators needs to consider that these values have been calculated from actual execution latency rather than theoretical peak throughput and that the 45 nm technology node is considerably older than the ones used to fabricate the other devices. Finally, accelerators generated through the SODA Synthesizer did not require any manual hardware design or DSE effort to reach the reported efficiency; there is ample room to introduce further optimizations in the process that would produce even better QoR.

A second qualitative comparison can be drawn between the SODA Synthesizer and other toolchains that facilitate the generation of specialized architectures starting from high-level programming frameworks. Table 6.6 (adapted from [15]) lists a selection of tools among the ones presented in Chapter 2, highlighting key features that reduce the effort needed to translate the software description of an algorithm into a hardware accelerator. The SODA Synthesizer is the only toolchain in the list to provide both automated optimizations and DSE through compiler optimizations, and system-level design features such as partitioning between host code and accelerator code or generating multi-accelerator architectures.

6.3.3 Dataflow architecture and centralized architecture

The SODA Synthesizer can compose multiple generated FSMD accelerators into one of two different types of system, i.e., the *centralized* architecture and the *dataflow* architecture. The centralized architecture is a system like the one depicted in Figure 4.5a, where individual accelerators are attached to a central bus, a microcontroller drives their execution, and the data they exchange is stored in and retrieved from external memory. The dataflow architecture, instead, is a system that uses the distributed controller proposed in Section 4.4 to orchestrate the execution of accelerators accessing shared memory, as depicted in Figure 4.5b. This section compares the performance of the two architectures by using the blocks of DNN layers of Figure 6.3, which are first synthesized and simulated individually with the SODA-OPT high-level optimizations enabled, and then connected together. The evaluation considers two application scenarios: a single input image processed by all operators in sequence, and a stream of 100 input images.

While the simulation already accounts for shared memory accesses, the cost of communication between accelerators and external memory needs to be estimated taking into consideration the type and size of the inputs and outputs for each kernel. The estimation considers a memory bandwidth of 6400MB/s, typical of DDR3 RAM modules using 45 nm technology cells, to calculate transfer times as seen by the accelerator in terms of clock cycles at 500MHz. In the centralized architecture, every accelerator communicates with its producers and consumers through external memory; in the dataflow architecture, instead, only input arguments to the first kernel and output arguments of the last one go through external memory, while intermediate results are kept within a shared on-chip scratchpad memory. The shared scratchpad memory is assumed to have as many ports as there are FSMD accelerators in the system, so that by using the HMI memory interface described in Section 4.4 the architecture can support conflict-free concurrent accelerator execution, allowing pipelined execution of streaming workloads. This assumption is reasonable, as high-performance scratchpad designs with up to 16 independent banks already exist, which is enough to support the benchmarks used in these experiments. Accesses to the shared scratchpad are accounted for in the same way as other accesses to on-chip memories, i.e., with 2 cycles latency for read operations and 1 cycle latency for write operations.

The overall latency of the centralized architecture is estimated by summing the execution time of each accelerator with the time it takes to transfer data to/from external memory before and after its execution. The streaming latency is calculated by multiplying the single input execution latency by the number of inputs in the stream; in

Table 6.7: *Execution times on the centralized and dataflow architecture.*

ResNet50

	Single Input			**Streaming (100 inputs)**		
Architecture	**Computation**	**Memory**	**Total**	**Computation**	**Memory**	**Total**
Centralized	1,146,101,992	7,152,635	1,153,254,627	114,610,199,200	715,263,486	115,325,462,686
Dataflow	806,742,427	656,320	807,398,747	34,677,385,627	656,320	34,678,041,947
Speedup	**1.4**	**10.9**	**1.4**	**3.3**	**1089.8**	**3.3**

MobileNetV2

	Single Input			**Streaming (100 inputs)**		
Architecture	**Computation**	**Memory**	**Total**	**Computation**	**Memory**	**Total**
Centralized	19,517,066	3,726,301	23,243,367	1,951,706,600	372,630,149	2,324,336,749
Dataflow	19,517,066	64,345	19,581,411	625,392,266	64,345	625,456,611
Speedup	**1.0**	**57.9**	**1.2**	**3.1**	**5,791.2**	**3.7**

fact, although the synthesized accelerators could execute in parallel on different inputs, SODA-OPT currently does not support the generation of host code with non-blocking function calls.

The model to estimate the performance of the dataflow architecture, instead, takes into account the support for concurrent and pipelined execution provided by the distributed controller. The execution time for a single input is calculated by looking at the application task graph and identifying the longest path from input to output; the latency of all kernels along the critical path is then summed together with memory access latency at the beginning at the end of the execution. In this way, the model accounts for fork-join patterns in the application data flow graph, where multiple branches can be executed in parallel and the overall latency is determined by the execution of the slowest branch. In the streaming execution scenario, the dataflow architecture latency is calculated as the latency of a single input execution plus N - 1 times the initiation interval, where N is the number of elements in the input stream, and the initiation interval is the latency of the slowest kernel or memory transfer.

Table 6.7 reports the execution latency in clock cycles for the two blocks of layers in Figure 6.3, and uses the results from the centralized architecture as a baseline to assess the performance improvement provided by the dataflow architecture. For the Resnet50 block, using the proposed dataflow architecture, accelerators implementing operators in the upper edge of the graph in Figure 6.3a can execute in parallel with accelerators implementing operators in the lower branch; compared to the centralized architecture baseline, this results in a speedup of 1.4x during single input execution, and a speedup of 3.5x when streaming a batch of 100 inputs. For MobileNetV2, although the task graph does not have parallel branches, the dataflow architecture still provides significant improvements due to the reduced access to external memory; in fact, the centralized system spends 57.9x more cycles with a single input and 5,791.2x more cycles in the streaming scenario to transfer data between accelerators and external memory.

6.4 Conclusion

The experiments described in this chapter demonstrated the benefits of a modern multi-level approach to HLS in terms of increased performance, productivity, and portability. Results obtained by generating accelerators for polyhedral benchmarks proved that the MLIR-based implementation of loop pipelining described in Chapter 3 improves performance with a reasonable area overhead, that performance improvement obtained through high-level optimizations is portable across HLS tools, and that custom transformations can be combined with existing MLIR passes to explore different design points. The high-level SODA-OPT optimization pipeline described in Chapters 3 and 4 was tested on polyhedral benchmarks and ML operators to assess its effect on accelerator QoR, also in comparison to state-of-the-art tools and techniques. Finally, the end-to-end hardware generation flow described in Chapter 4 was applied to DNN models to assess its capabilities to synthesize efficient accelerators, qualitatively comparing them to other existing solutions, and to compare the two different system architectures it supports.

CHAPTER 7

Conclusions

MODERN tools and techniques presented in this thesis contribute to enabling domain scientists and high-level application developers to efficiently use HLS tools. Thanks to the diffusion of the MLIR framework, the proposed approach can be applied to synthesize hardware accelerators for a growing number of input applications; transformations applied as MLIR passes are portable across different HLS tools, and they do not require manual intervention on the code. Finally, the modularity of the proposed design flow and the availability of multiple levels of abstraction facilitate the exploration of multiple design alternatives and the introduction of innovative optimization techniques. This chapter concludes the thesis by summarizing its main contributions and presenting further research opportunities in the field of design automation.

7.1 Summary of contributions

In the last few years, High-Level Synthesis has become an invaluable tool to simplify the development of hardware accelerators on FPGA and ASIC, providing higher and higher quality of results to users with little expertise in low-level RTL design. State-of-the-art HLS tools support C/C++ and LLVM IR inputs; however, they often require manually annotating the input code with compiler directives to guide the synthesis process and overcome the semantic mismatch between hardware design and general-purpose programming languages. Such a programming model still expects some hardware design knowledge from users, especially when the accelerator needs to be optimized to meet tight application requirements or when different configurations need to be evaluated looking for a specific trade-off between quality metrics.

This requirement prevents widespread adoption of HLS by domain scientists that develop data science and artificial intelligence algorithms in high-level, Python-based programming frameworks. Moreover, research that aims at improving the efficiency of the HLS process itself or the quality of generated accelerators is typically limited by the expressiveness of C/C++ code and by the annotations supported by a specific, closed-source backend tool. This thesis has proposed to solve the two issues by coupling established HLS tools with the modern compiler infrastructure provided by the MLIR framework: such a multi-level approach allows seamless integration with high-level ML frameworks, encourages the introduction of innovative optimization techniques at specific levels of abstraction, and can exploit multiple state-of-the-art HLS tools in the backend.

The proposed design flow allows to implement and apply high-level optimizations before HLS, as compiler passes supported by dedicated MLIR abstractions (dialects); such an approach can improve productivity, performance, and portability of optimizations. A new implementation of loop pipelining based on the MLIR affine dialect has been introduced as a case study, to test whether high-level compiler transformations can benefit HLS results without needing to modify the HLS tool itself. The proposed implementation can analyze dependencies between operations in the loop body and overlap the execution of iterations to increase parallelism; it can also forward results from one iteration to the other, support loops with variable bounds, and speculate execution of if-else blocks. All these transformations contribute to increasing the performance of the generated accelerators, and, since they do not introduce tool-specific annotations or code patterns, they also allow portability across different HLS tools. Productivity is increased on the user's side, as optimizations can be explored more easily and safely through compiler passes than through manual code rewriting, but also on the developer's side: dedicated MLIR dialects are built to solve domain-specific optimization problems, and there is no need to access the backend HLS code nor to be expert in low-level synthesis techniques.

Loop pipelining has been used as an example of the intrinsic optimization potential in a multi-level design and optimization flow; it has also been seamlessly integrated into the SODA-OPT frontend for HLS thanks to the modular nature of MLIR. SODA-OPT is part of the SODA Synthesizer, an end-to-end automated hardware compiler based on the multi-level approach to HLS presented in its thesis; among other features, SODA-OPT provides a set of compiler passes that can be enabled or disabled to explore different design points and a default optimization pipeline aimed at improving the performance of the generated accelerators.

The SODA Synthesizer is a practical realization of the concepts explored in this thesis, applied to generate optimized FPGA and ASIC accelerators for ML through an open-source, modular, compiler-based design flow. Its frontend provides a search and outlining methodology to automatically extract accelerator kernels and their data dependencies from an MLIR input specification; optimized kernels are synthesized by the backend HLS tool to generate FSMD accelerators and later composed in multi-accelerator systems. The SODA Synthesizer integrates a low-level synthesis methodology for the generation of coarse-grained, dynamically scheduled dataflow architectures with distributed control which are particularly suited to support streaming execution; analysis and transformation passes in the MLIR frontend support the low-level synthesis process and improve its results.

A multi-level compiler-based framework can adapt more easily to innovative input algorithms and hardware targets with respect to tools that generate code for HLS through a library of annotated C/C++ templates. For example, spiking neural networks are built of biologically-inspired integrate-and-fire neurons, and they are usually mapped on analog neuromorphic hardware; a new MLIR dialect has been introduced in the SODA Synthesizer to support the synthesis of SNN models into neuromorphic components. The dialect models concepts from the analog domain of spiking neurons through new types and operations that describe sequences of current spikes as lists of timestamps signaling their arrival. A plan has been drafted to integrate the NeuroXplorer toolchain into the SODA Synthesizer through the new dialect, first to simulate SNN models and map them onto neuromorphic devices, and later to generate hybrid systems composed of both digital and analog accelerators.

7.2 Future research directions

This thesis is the first step in a new line of research that integrates established HLS tools and novel compiler infrastructure to improve the automated synthesis process of accelerators for high-level applications. The proposed multi-level approach is modular and extensible by design, so different parts can be easily reused and adapted to the needs of different input applications, requirements, and research scenarios. Experimental results showed strengths and weaknesses of the approach, indicating possible next steps to improve the QoR of generated accelerators and the applicability of the proposed tools and techniques.

The proposed high-level loop pipelining pass can be further improved to support more irregular loop structures, and an extensive study can be carried out to identify which combinations of existing and custom MLIR passes are most profitable to HLS. More complex polyhedral analysis and optimization techniques can be implemented as affine passes and integrated into the proposed design and optimization flow. The SODA-OPT frontend provides initial support for DSE of available compiler passes, which can be enhanced through learning-based techniques to predict the best combination of optimizations without going through long synthesis runs. The current outlining strategy of the SODA Synthesizer can lead to imbalanced execution times and utilization in the dataflow architecture, as small kernels remain idle waiting until larger kernels have completed; the outlining process can be improved to analyze the computational graph and fuse operators together, or further partition them into smaller primitives, aiming at kernels with similar computational intensity and utilization of resources. Further improvements to the dataflow architecture include extending the memory interface design to support FSMD accelerators with multiple ports and better managing buffers between nodes, increasing memory parallelism. The SODA Synthesizer can generate Verilog code representing spiking neurons in digital logic to be mapped onto FPGAs or ASICs; in the future, it could allow to integrate ML accelerators composed of both digital and neuromorphic components in a complex heterogeneous system.

The availability of multiple levels of abstraction and domain-specific representations opens the door to new possibilities to study and implement innovative design automation methods, ranging from the exploration of techniques that can benefit HLS when applied at a high level of abstraction to the introduction of new synthesis methodologies and architectural models. Code for the tools developed in this thesis has been released in open-source to foster collaboration; parts of them can be easily reused or integrated with future research efforts.

List of Figures

List of Tables

Bibliography

[1] S. Curzel, N. Bohm Agostini, A. Tumeo, and F. Ferrandi, “Hardware Acceleration of Complex Machine Learning Models through Modern High-Level Synthesis,” in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 2022, pp. 209–210.

[2] J. Hennessy and D. Patterson, “A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 27–29.

[3] Semiconductor Industry Association, “International Technology Roadmap for Semiconductors 1999 Edition,” 1999.

[4] L. Truong and P. Hanrahan, “A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity,” in *3rd Summit on Advances in Programming Languages (SNAPL)*, vol. 136, 2019, pp. 7:1–7:21.

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-Level Synthesis for FPGAs: From Prototyping to Deployment,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[6] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, “FPGA HLS Today: Successes, Challenges, and Opportunities,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 1–42, 2022.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: A System for Large-Scale Machine Learning,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[9] XLA and TensorFlow teams, "XLA: Optimizing Compiler for Machine Learning," 2017. [Online]. Available: https://www.tensorflow.org/xla

[10] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph Lowering Compiler Techniques for Neural Networks," 2019. [Online]. Available: http://arxiv.org/abs/1805.00907

[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 578–594.

[12] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[13] N. Bohm Agostini, S. Curzel, J. J. Zhang, A. Limaye, C. Tan, V. Amatya, M. Minutoli, V. G. Castellana, J. Manzano, D. Brooks, G.-Y. Wei, and A. Tumeo, "Bridging Python to Silicon: The SODA Toolchain," *IEEE Micro*, vol. 42, no. 5, pp. 78–88, 2022.

[14] N. B. Agostini, S. Curzel, A. Limaye, V. Amatya, M. Minutoli, V. G. Castellana, J. Manzano, A. Tumeo, and F. Ferrandi, "The SODA Approach: Leveraging High-Level Synthesis for Hardware/Software Co-Design and Hardware Specialization," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 1359–1362.

[15] N. Bohm Agostini, S. Curzel, V. Amatya, C. Tan, M. Minutoli, V. G. Castellana, J. Manzano, D. Kaeli, and A. Tumeo, "An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022, pp. 1–9.

[16] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo, "Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications," in *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1327–1330.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni,

K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[18] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang, "Accelerating low bit-width convolutional neural networks with embedded FPGA," in *27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[19] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, N. Tran, and Z. Wu, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.

[20] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu *et al.*, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, pp. 1–23, 2018.

[21] Y. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 242–251.

[22] D. Pal, Y.-H. Lai, S. Xiang, N. Zhang, H. Chen, J. Casas, P. Cocchini, Z. Yang, J. Yang, L.-N. Pouchet, and Z. Zhang, "Accelerator Design with Decoupled Hardware Customizations: Benefits and Challenges," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 1351–1354.

[23] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013, pp. 519–530.

[24] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming Heterogeneous Systems from an Image Processing DSL," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 3, 2017.

[25] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W. Hwu, "Pylog: An algorithm-centric python-based FPGA programming and synthesis flow," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2015–2028, 2021.

[26] S. Skalicky, J. Monson, A. Schmidt, and M. French, "Hot & Spicy: Improving Productivity with Python and HLS for FPGAs," in *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 85–92.

[27] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 741–755.

[28] H. Ye, H. Jun, H. Jeong, S. Neuendorffer, and D. Chen, “ScaleHLS: A Scalable High-Level Synthesis Framework with Multi-Level Transformations and Optimizations,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 1355–1358.

[29] A. Mahapatra and B. C. Schafer, “Machine-learning based simulated annealer method for high level synthesis design space exploration,” in *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*, 2014, pp. 1–6.

[30] H.-Y. Liu and L. P. Carloni, “On learning-based methods for design-space exploration with High-Level Synthesis,” in *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–7.

[31] Z. Lin, J. Zhao, S. Sinha, and W. Zhang, “HL-Pow: A Learning-Based Power Modeling Framework for High-Level Synthesis,” in *25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020, pp. 574–580.

[32] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, “Accurate Operation Delay Prediction for FPGA HLS Using Graph Neural Networks,” in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[33] N. Wu, Y. Xie, and C. Hao, “IRONMAN: GNN-Assisted Design Space Exploration in High-Level Synthesis via Reinforcement Learning,” in *Proceedings of the 2021 Great Lakes Symposium on VLSI (GLSVLSI)*, 2021, pp. 39–44.

[34] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, “Automated Accelerator Optimization Aided by Graph Neural Networks,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 55–60.

[35] J. Kwon and L. P. Carloni, “Transfer Learning for Design-Space Exploration with High-Level Synthesis,” in *2nd ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2020, pp. 163–168.

[36] Y. Bai, A. Sohrabizadeh, Y. Sun, and J. Cong, “Improving GNN-Based Accelerator Design Automation with Meta Learning,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 1347–1350.

[37] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner, “Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains,” *IEEE Access*, vol. 8, pp. 174 692–174 722, 2020.

[38] AMD-Xilinx, “Vitis HLS LLVM 2021.2,” 2021. [Online]. Available: https://github.com/Xilinx/HLS

[39] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo, and B. Xu, “Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project,” in *Proceedings of the 56th Annual Design Automation Conference (DAC)*, 2019, pp. 1–4.

[40] CIRCT Developers, “CIRCT / Circuit IR Compilers and Tools,” 2020. [Online]. Available: https://circt.llvm.org/

[41] M. Urbach and M. B. Petersen, "HLS from PyTorch to System Verilog with MLIR and CIRCT," 2022, 2nd Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE).

[42] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving High Level Synthesis Optimization Opportunity through Polyhedral Transformations," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2013, pp. 9–18.

[43] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-Based Data Reuse Optimization for Configurable Computing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2013, pp. 29–38.

[44] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong, "Improving Polyhedral Code Generation for High-Level Synthesis," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013, pp. 1–10.

[45] R. Zhao, J. Cheng, W. Luk, and G. A. Constantinides, "POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[46] S. Derrien, T. Marty, S. Rokicki, and T. Yuki, "Toward Speculative Loop Pipelining for High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4229–4239, 2020.

[47] P. Li, L.-N. Pouchet, and J. Cong, "Throughput Optimization for High-Level Synthesis using Resource Constraints," in *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2014.

[48] J. Liu, J. Wickerson, and G. A. Constantinides, "Loop Splitting for Efficient Pipelining in High-Level Synthesis," in *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 72–79.

[49] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Polyhedral-Based Dynamic Loop Pipelining for High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1802–1815, 2017.

[50] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 742–760, 1999.

[51] V. G. Castellana and F. Ferrandi, "Abstract: Speeding-Up Memory Intensive Applications through Adaptive Hardware Accelerators," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 1415–1416.

[52] R. S. Nikhil, "Bluespec: A General-Purpose Approach to High-Level Synthesis based on Parallel Atomic Transactions," in *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008, pp. 129–146.

[53] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically Scheduled High-Level Synthesis," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018, pp. 127–136.

[54] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining Dynamic & Static Scheduling in High-Level Synthesis," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020, pp. 288–298.

[55] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A Language and Compiler for Application Accelerators," in *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 296–311.

[56] AMD-Xilinx, "Exploiting Task Level Parallelism: Dataflow Optimization," in *Vitis HLS User Guide*, 2021, pp. 283–296.

[57] V. G. Castellana, A. Tumeo, and F. Ferrandi, "High-Level Synthesis of Parallel Specifications Coupling Static and Dynamic Controllers," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 192–202.

[58] T. Tuttle, "Keynote: Accelerating the IoT," 2017, 54th ACM/IEEE Design Automation Conference (DAC).

[59] R. Machupalli, M. Hossain, and M. Mandal, "Review of ASIC accelerators for deep neural network," *Microprocessors and Microsystems*, vol. 89, p. 104441, 2022.

[60] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A Survey of FPGA-Based Neural Network Inference Accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, no. 1, pp. 1–26, 2019.

[61] NVIDIA, "Tensor Cores," 2017. [Online]. Available: https://developer.nvidia.com/tensor-cores

[62] AMD-Xilinx, "AI Engine Technology," 2020. [Online]. Available: https://www.xilinx.com/products/technology/ai-engine.html

[63] SambaNova Systems, "Accelerated Computing with a Reconfigurable Dataflow Architecture," 2021. [Online]. Available: https://sambanova.ai/

[64] Graphcore, "Grapchore: Accelerating machine learning for a world of intelligent machines," 2016. [Online]. Available: https://www.graphcore.ai/

[65] Cerebras, "Cerebras Systems: Achieving Industry Best AI Performance Through A Systems Approach," 2021. [Online]. Available: https://www.cerebras.net/

[66] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14.

[67] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[68] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[69] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, “PuDianNao: A Polyvalent Machine Learning Accelerator,” in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 369–381.

[70] M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E. Painkras, and S. Furber, “SpiNNaker: Mapping Neural Networks onto a Massively-Parallel Chip Multiprocessor,” in *IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 2008, pp. 2849–2856.

[71] M. V. DeBole, B. Taba, A. Amir, F. Akopyan, A. Andreopoulos, W. P. Risk, J. Kusnitz, C. Ortega Otero, T. K. Nayak, R. Appuswamy, P. J. Carlson, A. S. Cassidy, P. Datta, S. K. Esser, G. J. Garreau, K. L. Holland, S. Lekuch, M. Mastro, J. McKinstry, C. di Nolfo, B. Paulovicks, J. Sawada, K. Schleupen, B. G. Shaw, J. L. Klamo, M. D. Flickner, J. V. Arthur, and D. S. Modha, “TrueNorth: Accelerating From Zero to 64 Million Neurons in 10 Years,” *Computer*, vol. 52, no. 5, pp. 20–29, 2019.

[72] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning,” *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[73] S. Panchapakesan, Z. Fang, and J. Li, “Syncnn: Evaluating and accelerating spiking neural networks on fpgas,” *ACM Transactions on Reconfigurable Technology and Systems*, pp. 1–26, 2022.

[74] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, “TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 14–26.

[75] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, “DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[76] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W. Hwu, and D. Chen, “DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator,” in *ICCAD*, 2020, pp. 1–9.

[77] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, “SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[78] H. Kwon, A. Samajdar, and T. Krishna, “MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects,” in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 461–475.

[79] H. Esmaeilzadeh, S. Ghodrati, J. Gu, S. Guo, A. B. Kahng, J. K. Kim, S. Kinzer, R. Mahapatra, S. D. Manasi, E. S. Mascarenhas, S. Sapatnekar, R. Varadarajan, Z. Wang, H. Xu, B. R. Yatham,

and Z. Zeng, "VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021, pp. 1–7.

[80] S. Kinzer, J. K. Kim, S. Ghodrati, B. Yatham, A. Althoff, D. Mahajan, S. Lerner, and H. Esmaeilzadeh, "A Computational Stack for Cross-Domain Acceleration," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 54–70.

[81] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao *et al.*, "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021, pp. 769–774.

[82] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin *et al.*, "A Hardware-Software Blueprint for Flexible Deep Learning Specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.

[83] D. H. Noronha, B. Salehpour, and S. J. Wilton, "LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks," in *5th International Workshop on FPGAs for Software Programmers (FSP)*, 2018, pp. 1–8.

[84] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, pp. 1–27, 2013.

[85] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions," *ACM Computing Surveys*, vol. 51, no. 3, 2018.

[86] C. N. Coelho, A. Kuusela, S. Li, H. Zhuang, J. Ngadiuba, T. K. Aarrestad, V. Loncar, M. Pierini, A. A. Pol, and S. Summers, "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors," *Nature Machine Intelligence*, vol. 3, no. 8, pp. 675–686, 2021.

[87] J. Ngadiuba, V. Loncar, M. Pierini, S. Summers, G. Di Guglielmo, J. Duarte, P. Harris, D. Rankin, S. Jindariani, M. Liu *et al.*, "Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 1, p. 015001, 2020.

[88] T. Aarrestad, V. Loncar, N. Ghielmetti, M. Pierini, S. Summers, J. Ngadiuba, C. Petersson, H. Linander, Y. Iiyama, G. Di Guglielmo *et al.*, "Fast convolutional neural networks on FPGAs with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 4, p. 045015, 2021.

[89] S. Curzel, S. Jovic, M. Fiorito, A. Tumeo, and F. Ferrandi, "Higher-Level Synthesis: experimenting with MLIR polyhedral representations for accelerator design," in *12th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2022, pp. 1–10.

[90] S. Curzel, S. Jovic, M. Fiorito, A. Tumeo, and F. Ferrandi, "MLIR loop optimizations for High-Level Synthesis: a case study," in *31st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2022, pp. 1–2.

[91] M. S. Lam, “Software pipelining,” in *A Systolic Array Optimizing Compiler*. Springer, 1989, pp. 83–124.

[92] P. Sittel, J. Oppermann, M. Kumm, A. Koch, and P. Zipf, “HatScheT: A Contribution to Agile HLS,” in *5th International Workshop on FPGAs for Software Programmers (FSP)*, 2018, pp. 1–8.

[93] AMD-Xilinx, “Optimization techniques in Vitis HLS,” 2021. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/vitis_hls_optimization_techniques.html

[94] T. D. Le, G.-T. Bercea, T. Chen, A. E. Eichenberger, H. Imai, T. Jin, K. Kawachiya, Y. Negishi, and K. O’Brien, “Compiling ONNX Neural Network Models Using MLIR,” 2020. [Online]. Available: http://arxiv.org/abs/2008.08272

[95] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, “Code generation schema for modulo scheduled loops,” *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 158–169, 1992.

[96] A. Stoutchinin and G. Gao, “If-conversion in SSA form,” in *European Conference on Parallel Processing*, 2004, pp. 336–345.

[97] M. Fellahi and A. Cohen, “Software Pipelining in Nested Loops with Prolog-Epilog Merging,” in *International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, 2009, pp. 80–94.

[98] S. Curzel, N. Bohm Agostini, V. G. Castellana, M. Minutoli, A. Limaye, J. Manzano, J. Zhang, D. Brooks, G.-Y. Wei, F. Ferrandi, and A. Tumeo, “End-to-end Synthesis of Dynamically Controlled Machine Learning Accelerators,” to appear in *IEEE Transactions on Computers*.

[99] P. Beckman, C. Catlett, M. Ahmed, M. Alawad, L. Bai, P. Balaprakash, K. Barker, R. Berry, A. Bhuyan, G. Brebner, K. Burkes, A. Butko, F. Cappello, R. Chard, S. Collis, J. Cree, D. Dasgupta, A. Evdokimov, J. M. Fields, P. Fuhr, C. Harper, Y. Jin, R. Kettimuthu, M. Kiran, R. Kozma, P. A. Kumar, Y. Kumar, L. Luo, L. Mashayekhy, I. Monga, B. Nickless, T. Pappas, E. Peterson, T. Pfeffer, S. Rakheja, V. R. Tribaldos, S. Rooke, S. Roy, T. Saadawi, A. Sandy, R. Sankaran, N. Schwarz, S. Somnath, M. Stan, C. Stuart, R. Sullivan, A. Sumant, G. Tchilinguirian, N. Tran, A. Veeramany, A. Wang, B. Wang, A. Wiedlea, S. Wielandt, T. Windus, Y. Wu, X. Yang, Z. Yao, R. Yu, Y. Zeng, and Y. Zhang, “5G Enabled Energy Innovation: Advanced Wireless Networks for Science,” *Workshop Report*, March 2020. [Online]. Available: https://www.osti.gov/biblio/1606538

[100] E. W. Bethel and M. G. eds, “Report of the DOE Workshop on Management, Analysis, and Visualization of Experimental and Observational data - The Convergence of Data and Computing,” *Workshop Report*, May 2016. [Online]. Available: https://www.osti.gov/biblio/1525145

[101] M. Minutoli, V. G. Castellana, A. Tumeo, and F. Ferrandi, “Inter-Procedural Resource Sharing in High-Level Synthesis through Function Proxies,” in *25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[102] M. Minutoli, V. Castellana, N. Saporetti, S. Devecchi, M. Lattuada, P. Fezzardi, A. Tumeo, and F. Ferrandi, “Svelto: High-Level Synthesis of Multi-Threaded Accelerators for Graph Analytics,” *IEEE Transactions on Computers*, vol. 71, no. 3, pp. 520–533, 2022.

[103] C.-J. Tseng and D. P. Siewiorek, “Automated Synthesis of Data Paths in Digital Systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, 1986.

[104] V. G. Castellana, A. Tumeo, and F. Ferrandi, “An Adaptive Memory Interface Controller for Improving Bandwidth Utilization of Hybrid and Reconfigurable Systems,” in *Design, Automation and Test in Europe Conference (DATE)*, 2014, pp. 1–4.

[105] S. Curzel, N. Bohm Agostini, S. Song, I. Dagli, A. Limaye, C. Tan, M. Minutoli, V. G. Castellana, V. Amatya, J. Manzano, A. Das, F. Ferrandi, and A. Tumeo, “Automated Generation of Integrated Digital and Spiking Neuromorphic Machine Learning Accelerators,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–7.

[106] W. Maass, “Networks of Spiking Neurons: The Third Generation of Neural Network Models,” *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[107] J. Hasler, “Large-Scale Field-Programmable Analog Arrays,” *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1283–1302, 2020.

[108] D. Neil and S.-C. Liu, “Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2621–2628, 2014.

[109] H. Fang, Z. Mei, A. Shrestha, Z. Zhao, Y. Li, and Q. Qiu, “Encoding, Model, and Architecture: Systematic Optimization for Spiking Neural Network in FPGAs,” in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[110] D. Pani, P. Meloni, G. Tuveri, F. Palumbo, P. Massobrio, and L. Raffo, “An FPGA Platform for Real-Time Simulation of Spiking Neuronal Networks,” *Frontiers in neuroscience*, vol. 11, pp. 1–13, 2017.

[111] R. M. Wang, C. S. Thakur, and A. Van Schaik, “An FPGA-Based Massively Parallel Neuromorphic Cortex Simulator,” *Frontiers in neuroscience*, vol. 12, pp. 1–18, 2018.

[112] A. Balaji, S. Song, T. Titirsha, A. Das, J. Krichmar, N. Dutt, J. Shackleford, N. Kandasamy, and F. Catthoor, “NeuroXplorer 1.0: An Extensible Framework for Architectural Exploration with Spiking Neural Networks,” in *International Conference on Neuromorphic Systems (ICONS)*, 2021, pp. 1–9.

[113] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural networks*, vol. 61, pp. 85–117, 2015.

[114] S. Fusi and M. Mattia, “Collective Behavior of Networks with Linear (VLSI) Integrate-and-Fire Neurons,” *Neural Computation*, vol. 11, no. 3, pp. 633–652, 1999.

[115] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, “Deep learning in spiking neural networks,” *Neural Networks*, vol. 111, pp. 47–63, 2019.

[116] A. Das, P. Pradhapan, W. Groenendaal, P. Adiraju, R. T. Rajan, F. Catthoor, S. Schaafsma, J. L. Krichmar, N. Dutt, and C. Van Hoof, “Unsupervised heart-rate estimation in wearables with Liquid states and a probabilistic readout,” *Neural Networks*, vol. 99, pp. 134–147, 2018.

[117] M. S. Shim and P. Li, "Biologically Inspired Reinforcement Learning for Mobile Robot Collision Avoidance," in *International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 3098–3105.

[118] X. Liu, W. Wen, X. Qian, H. Li, and Y. Chen, "Neu-NoC: A High-efficient Interconnection Network for Accelerated Neuromorphic Systems," in *23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 141–146.

[119] A. Balaji, Y. Wu, A. Das, F. Catthoor, and S. Schaafsma, "Exploration of Segmented Bus As Scalable Global Interconnect for Neuromorphic Computing," in *Proceedings of the 2019 Great Lakes Symposium on VLSI (GLSVLSI)*, 2019, pp. 495–499.

[120] C. Liu, B. Yan, C. Yang, L. Song, Z. Li, B. Liu, Y. Chen, H. Li, Q. Wu, and H. Jiang, "A Spiking Neuromorphic Design with Resistive Crossbar," in *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[121] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "NEUTRAMS: Neural Network Transformation and Co-design under Neuromorphic Hardware Constraints," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[122] A. Balaji, A. Das, Y. Wu, K. Huynh, F. G. Dell'Anna, G. Indiveri, J. L. Krichmar, N. D. Dutt, S. Schaafsma, and F. Catthoor, "Mapping Spiking Neural Networks to Neuromorphic Hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 76–86, 2020.

[123] A. Amir, P. Datta, W. P. Risk, A. S. Cassidy, J. A. Kusnitz, S. K. Esser, A. Andreopoulos, T. M. Wong, M. Flickner, R. Alvarez-Icaza, E. McQuinn, B. Shaw, N. Pass, and D. S. Modha, "Cognitive Computing Programming Paradigm: A Corelet Language for Composing Networks of Neurosynaptic Cores," in *International Joint Conference on Neural Networks (IJCNN)*, 2013, pp. 1–10.

[124] T. Titirsha, S. Song, A. Balaji, and A. Das, "On the Role of System Software in Energy Management of Neuromorphic Computing," in *Proceedings of the 18th ACM International Conference on Computing Frontiers (CF)*, 2021, pp. 124–132.

[125] S. Song, J. Hanamshet, A. Balaji, A. Das, J. L. Krichmar, N. D. Dutt, N. Kandasamy, and F. Catthoor, "Dynamic Reliability Management in Neuromorphic Computing," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 17, no. 4, pp. 1–27, 2021.

[126] T. Titirsha, S. Song, A. Das, J. Krichmar, N. Dutt, N. Kandasamy, and F. Catthoor, "Endurance-Aware Mapping of Spiking Neural Networks to Neuromorphic Hardware," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 288–301, 2021.

[127] S. Song, T. Titirsha, and A. Das, "Improving Inference Lifetime of Neuromorphic Systems via Intelligent Synapse Mapping," in *32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 17–24.

[128] S. Song, M. L. Varshika, A. Das, and N. Kandasamy, "A Design Flow for Mapping Spiking Neural Networks to Many-Core Neuromorphic Hardware," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[129] B. W. Kernighan and S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs,” *The Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[130] A. Das, Y. Wu, K. Huynh, F. Dell’Anna, F. Catthoor, and S. Schaafsma, “Mapping of local and global synapses on spiking neuromorphic hardware,” in *Design, Automation and Test in Europe Conference (DATE)*, 2018, pp. 1217–1222.

[131] C.-K. Lin, A. Wild, G. N. Chinya, T.-H. Lin, M. Davies, and H. Wang, “Mapping Spiking Neural Networks onto a Manycore Neuromorphic Architecture,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 78–89.

[132] A. P. Davison, D. Brüderle, J. M. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, “PyNN: a common interface for neuronal network simulators,” *Frontiers in neuroinformatics*, vol. 2, p. 11, 2009.

[133] A. Balaji, P. Adiraju, H. J. Kashyap, A. Das, J. L. Krichmar, N. D. Dutt, and F. Catthoor, “PyCARL: A PyNN Interface for Hardware-Software Co-Simulation of Spiking Neural Network,” in *International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–10.

[134] T.-S. Chou, H. J. Kashyap, J. Xing, S. Listopad, E. L. Rounds, M. Beyeler, N. Dutt, and J. L. Krichmar, “CARLsim 4: An Open Source Library for Large Scale, Biologically Detailed Spiking Neural Network Simulation using Heterogeneous Clusters,” in *International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–8.

[135] M. Stimberg, R. Brette, and D. F. Goodman, “Brian 2, an intuitive and efficient neural simulator,” *eLife*, vol. 8, pp. 1–41, 2019.

[136] J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig, “PyNEST: a convenient interface to the NEST simulator,” *Frontiers in neuroinformatics*, pp. 1–12, 2009.

[137] M. L. Hines and N. T. Carnevale, “The NEURON simulation environment,” *Neural computation*, vol. 9, no. 6, pp. 1179–1209, 1997.

[138] A. Balaji, S. Song, A. Das, J. Krichmar, N. Dutt, J. Shackleford, N. Kandasamy, and F. Catthoor, “Enabling Resource-Aware Mapping of Spiking Neural Networks via Spatial Decomposition,” *IEEE Embedded Systems Letters*, vol. 13, no. 3, pp. 142–145, 2021.

[139] S. Song, A. Balaji, A. Das, N. Kandasamy, and J. Shackleford, “Compiling Spiking Neural Networks to Neuromorphic Hardware,” in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2020, pp. 38–50.

[140] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[141] L.-N. Pouchet, “PolyBench - the Polyhedral Benchmark suite,” 2012. [Online]. Available: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[142] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, “Polygeist: Raising C to Polyhedral MLIR,” in *30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021, pp. 45–59.

[143] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen, “ILP-based Modulo Scheduling for High-Level Synthesis,” in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2016, pp. 1–10.

[144] R. Zhao and J. Cheng, “Phism: Polyhedral High-Level Synthesis in MLIR,” in *1st Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*, 2021.

[145] N. B. Agostini, S. Dong, E. Karimi, M. T. Lapuerta, J. Cano, J. L. Abellán, and D. Kaeli, “Design Space Exploration of Accelerators and End-to-End DNN Evaluation with TFLITE-SOC,” in *32nd IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 10–19.