Optimize edge AI processing through innovative compilation techniques

Shreya Alladi*, Alexandre Lopoukhine†, Georgios Alexandris‡, Andrea Nardi-Dei§, Ravikiran Ravindranath Reddy*, Christos P. Lamprakos‡, Panagiotis Chaidos‡, Alexis Maras‡, Alberto Ros*, Tobias Grosser†, Sotirios Xydis‡, Dimitrios Soudris‡, Marc Geilen§, Sander Stuijk§, Henk Corporaal§, and Alexandra Jimborean*

*University of Murcia, †University of Cambridge, §Eindhoven University of Technology, ‡National Technical University of Athens

Abstract—**Heterogeneous architectures became a compelling choice for edge processors executing complex DNN workloads, as they provide an ideal blend of openness, customization, energy-efficient heterogeneity, and scalable performance. Compiler optimization for DNNs on heterogeneous System-on-Chip (SoC) architectures however, must navigate complex hardware-software co-design, data movement minimization, aggressive parallelism exploitation, and advanced static/dynamic code transformations to deliver high performance and energy efficiency.**

This paper presents a novel compiler ecosystem for highly heterogeneous SoCs with multiple back-end targets, spanning from typical CPUs, to programmable RISC-V clusters and up to dedicated and reconfigurable accelerators. It puts together static analysis, optimization, and scheduling infrastructure to overcome the limitations of current state-of-the-art tools for heterogeneous edge AI processors. Our compilation pipeline introduces several innovative features: (1) an automatic end-to-end flow for RISC-V-based platforms, (2) efficient data layout remapping (reducing memory footprint by 35% on average) and recognition of complex ternary reductions for auto-vectorization, (3) code layout adaptation for hardware simplification, (4) a novel MLIR-based RISC-V backend supporting optimized matrix-multiplication micro-kernels that reach 90% of peak performance, (5) periodic scheduling capabilities for layer-fused CNNs, and (6) automated mapping and scheduling onto heterogeneous CGRA templates for advanced parallel kernel execution, delivering 33% higher energy efficiency than the scalar implementation and up to 3.6× higher performance. These advances enable hardware-aware compilation that reduces manual optimization effort, lowers energy consumption through memory and computation optimization, and minimizes memory footprint and data transfers.

Index Terms—**Compilers, MLIR, LLVM, RISC-V, Systems on Chip, CGRAs.**

I. Introduction

Recently, artificial intelligence (AI) has seen exceptional growth, fueled by breakthroughs in learning algorithms, the vast increase in available training data, and the evolution of advanced software frameworks, and computing hardware capabilities. These developments have enabled increasingly complex models capable of achieving impressive accuracy across vision, speech, and decision-making tasks. Yet, the computational intensity of such models presents a major challenge, where power consumption, memory capacity, hardware resources, and real-time responsiveness are critical constraints.

To meet these constraints, the AI hardware landscape has shifted from traditional homogeneous architectures toward heterogeneous computing platforms that integrate domain-specific accelerators. Among these, RISC-V has emerged as a compelling open instruction set architecture (ISA) for the next generation of edge AI systems, offering extensibility, modularity, and a rapidly growing ecosystem of customizable hardware components [21]. In this evolving landscape, CONVOLVE's [9] mission is to advance the EU's position in the development lifecycle of smart edge AI processors using a holistic approach across the software-hardware co-design stack. CONVOLVE leverages the xDSL [13], MLIR [19] and LLVM [18] frameworks to provide an end-to-end compiler and automatic optimizations across all abstraction levels. By building on MLIR, an industry-supported framework for optimizing programs via progressive lowering through multiple domain-specific intermediate representations (IRs), CONVOLVE enables the construction of new compilation flows that can scale to the diverse and domain-specific requirements of modern application stacks.

However, existing and established frameworks such as PyTorch, ONNX-Runtime (ORT), and optimization techniques have been primarily developed for mature high-performance cloud computing platforms with established hardware such as x86 CPUs (Intel, AMD) and GPUs, neglecting the specialized accelerator-based architectures usually found in edge computing deployments. Although efforts have been made to port PyTorch to emerging RISC-V platforms, several challenges were encountered. Essential libraries are not readily available for RISC-V (e.g., cpuinfo), and significant engineering effort is required to adapt low-level manual optimizations to the RISC-V architecture. This has led to a noticeable performance gap between x86 and RISC-V platforms [8]. There exists a widening disconnect between the potential of RISC-V-based edge AI accelerators and the ability of existing software ecosystems to fully exploit that potential. Existing compilers and runtime systems rely heavily on hand-optimized kernels and architecture-specific heuristics and they struggle to adapt to rapidly evolving architectures such as the RISC-V ecosystem. For instance, there are no available non-commercial hand-tuned libraries for RISC-V. Moreover, they often fail to efficiently exploit heterogeneous

978-3-9826741-1-7/DATE26/© 2026 EDAA

computing environments, leading to suboptimal performance and limited portability across platforms.

In the post-Moore's era, software and hardware must be co-designed to maximize performance and energy efficiency. Towards this direction, modern compilation flows for edge AI face challenges in orchestrating computation across heterogeneous accelerator-based architectures. The static memory allocation problem becomes particularly acute due to limited on-chip SRAM and asymmetric organization, requiring global optimization of buffer lifetimes, tensor reuse, and operator fusion under tight latency and energy constraints. The heterogeneity of accelerators, i.e. processing elements ranging from specialized NPUs and multi-core SW-programmable clusters up to reconfigurable vectorized array, necessitates fine-grained scheduling strategies that jointly optimize DNN layer partitioning, data flows and locality. Achieving convergence among these dimensions remains a fundamental barrier to compiler scalability, generality, and performance portability in highly heterogeneous edge AI systems.

Furthermore, neural networks are a rapidly evolving domain with continuously changing models and kernels, supporting SoCs featuring heterogeneous accelerators and ISA extensions is challenging. These heterogeneous accelerators require distinct optimization trajectories, particularly in the backend compilation stages. A fallback option to the generic RISC-V path is essential to support a wide range of use cases. Therefore, a multi-path compiler approach is necessary to support both diverse models via the generic RISC-V path and specialized accelerator-specific optimizations for targeted use cases, all built upon the open RISC-V architecture.

Addressing these challenges requires a rethinking of compiler design and automated optimization strategies for the edge domain. In this work, we present research outcomes delivered in CONVOLVE [9] EU funded project with respect to the static analysis, compiler tools, and infrastructure necessary to develop the compiler support required to address limitations of state-of-the-art approaches for heterogeneous edge AI processors.

To this end we propose a compilation framework and tools to bridge deep learning applications with heterogeneous accelerator platforms through a modular, progressive approach.

The main advantages of a modular compilation approach are:

- Frontend Flexibility: Supports ingestion from multiple ML frameworks while maintaining a common IR for hardware-agnostic optimizations (see section III).
- Target Extensibility: Enables pluggable backend support for diverse architectures (Snitch/SNAX and others) through shared interfaces and lowering passes (see section IV-A, IV-B).
- Enabling Co-Design: Progressive feedback-driven compilation (via simulation, emulation, and utilization analysis) drives joint optimization of algorithms and hardware capabilities (see section III-B, III-C, IV-B).
- Flow Substitution and Interoperability: Allows the replacement of various parts and components with alternative flows, while still enabling reuse wherever possible (e.g., CGRA compilation pipeline)(see Section V).

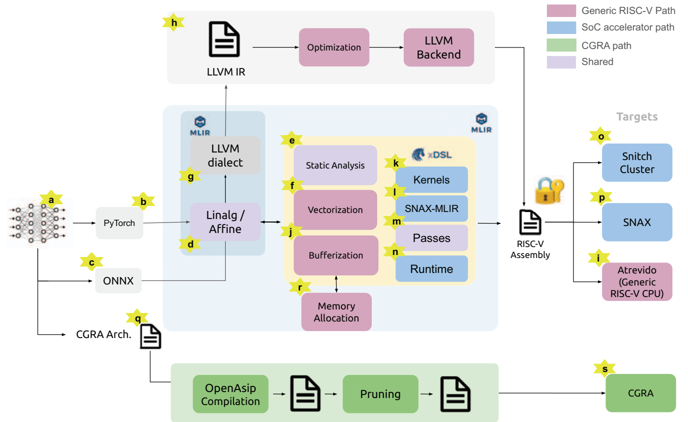


Fig. 1: CONVOLVE modular compilation framework

II. Convolve approach overview

We propose a multi-stage, multi-path compilation framework, as shown in Figure 1, that bridges deep learning applications with heterogeneous accelerator platforms through a modular and progressive approach. Built on the MLIR–LLVM infrastructure, the framework ingests neural network models and progressively lowers them into optimized linear algebra primitives along multiple compilation paths, enabling flexible optimization and specialization for diverse hardware targets. MLIR provides structured, multi-level representations that capture high-level semantics while allowing fine-grained transformations, whereas LLVM ensures mature backend support and portability across architectures. It generates highly efficient code for a range of architectures, including generic RISC-V cores, base processing elements (Snitch Clusters [32]), and specialized accelerators (SNAX [3]). This end-to-end flow integrates hardware-aware optimizations and simulation capabilities to maximize accelerator utilization while maintaining support for dynamic neural networks and emerging machine learning architectures. The modular compilation approach offers key advantages: frontend flexibility for integrating multiple ML frameworks through a unified IR, target extensibility via pluggable backends (accelerators) for diverse architectures, hardware–software co-designs, and flow interoperability enabling component substitution and reuse across compilation pipelines.

The fast-moving nature of Neural Network (NN)s motivates System on Chip (SoC)s with heterogeneous RISC-V-based accelerators that need distinct compiler backend optimizations, all orchestrated in an open RISC-V ecosystem for versatility and innovation. Within the CONVOLVE modular compilation framework, NNs can follow multiple transformation paths, depending on the target hardware:

- Generic RISC-V path: A Flexible compiler frontend path that compiles via standard toolchains (e.g., LLVM) to the generic RISC-V ISA, providing a robust, fully functional fallback for all CONVOLVE use cases. this path is useful for parts of applications that are not accelerated by the Convolve accelerators , such that we can still run apps completely. This approach unlocks capabilities for compiler analysis and co-design of edge CPUs, and establishes

a performance baseline for accelerator comparison as shown in path $a \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow j \rightarrow r \rightarrow h \rightarrow i$ in Figure 1 (for additional details, see section III, Figure 2 and code repository [2, 27]).
- SoC accelerator path: Extends compiler support and co-design to target CONVOLVE accelerators, including Snitch PE clusters with tailored micro-kernels and the SNAX fixed-function GEMM accelerator as shown in path $a \rightarrow b \rightarrow d \rightarrow e \rightarrow k \rightarrow l \rightarrow j \rightarrow m \rightarrow n \rightarrow o/p$ in Figure 1 (see section IV and code repository [23]).
- CGRA path: Provides interoperability to target hardware accelerators that can be (re)configured with statically compiled bitstreams/overlays. It compiles native C code through OpenASIP's LLVM toolchain, enhanced with HW-specific intrinsics, thus enabling the user to explore and orchestrate CGRA-specific mappings on a high level, as shown in path $a \rightarrow q \rightarrow s$ in Figure 1 (see section V).

III. Generic RISC-V path

We propose two multi-stage, end-to-end compiler pipelines named ONNX-MLIR-LLVM (OML), optimization to OML pipeline such as vectorization ONNX-MLIR-LLVM-vect (OML-vect), with blue, purple, and yellow boxes, respectively. Unmodified sections of the pipeline are shown in gray. Our pipeline leverages MLIR to implement compiler passes highlighted in pink, such as a reduction pass and a data preparation pass, to enhance auto-vectorization and improve performance.

However, many edge devices face severe resource constraints and often lack hardware acceleration or vectorization support, making efficient AI inference a major challenge. To address these limitations, we introduce CAIF [28], a compiler-assisted instruction fusion framework that boosts execution efficiency through lightweight, hardware-aware fusion.

A. End-to-end baseline compilation for RISC-V: oml

We propose OML [1], a multi-stage compiler pipeline (Figure 2, blue) that exposes compiler intermediate representations to enable advanced optimizations and generates fully static neural network executables using the MLIR ecosystem. OML extends the existing compilation flow by integrating and compiling ONNX-MLIR runtime libraries, porting them to RISC-V platforms, and overcoming the compatibility constraints of current state-of-the-art tools, which are typically limited to x86 architectures. ONNX graphs are lowered through the MLIR dialects via ONNX-MLIR, after which OML further lowers the LLVM dialect to stand-alone LLVM IR, allowing full LLVM optimization and assembly generation. In contrast to ONNX-MLIR, which produces shared libraries and incurs runtime overhead, OML generates static binaries with reduced memory usage and improved efficiency through direct MLIR- and LLVM-level optimizations such as vectorization. OML achieves 2x speedup, on average, compared to ORT-default (hand-optimized C/C++ hardware-specific libraries) on RISC-V CPU (Atrevido [4] RISC-V core).

B. Acceleration through vectorization: oml-vect

Our proposal OML-vect [1] extends OML with MLIR-based passes (Fig. 1, pink) for data layout optimization (compile-time matrix transposition) and reduction analysis, enhancing auto-vectorization, performance, and memory efficiency. Transposing matrices enables consecutive memory access, and together with reduction detection, allows the compiler to perform effective vectorization. Unlike ONNX-MLIR, which increases partial computations and memory operations, OML-vect's combined reduction-based vectorization and data layout optimization reduce vector loads by 18.83% and stores by 99%, significantly improving memory efficiency and data reuse.

1) data-layout optimization: We analyze and identify inefficient data layouts that hinder auto-vectorization due to the performance penalties associated with non-contiguous memory accesses. To address this, we implement a compile-time data layout optimization pass that automatically transposes vectorization-unfriendly matrices, converting them into vectorization-friendly layouts with contiguous memory access patterns. As shown in Figure 3, which shows the data layout before and after transformation, our approach reorders data to expose contiguous memory accesses suitable for auto-vectorization. Unlike runtime-based transformations (e.g., memref.transpose), which introduce additional runtime instructions, overhead, and additional memory consumption, our compile-time approach eliminates such costs by performing the transformation during compilation. This optimization not only removes runtime overhead and data copying but also enables the auto-vectorizer to fully exploit hardware-level parallelism.

2) reduction: The MLIR framework's match-reduction-test pass is designed to identify simple reduction patterns, such as addition (addf) operations from the arith dialect. However, it fails to detect more complex reductions, including operations from the math dialect such as fused multiply-add (FMA), maximum, or minimum.

To overcome this limitation, OML-reduction introduces a systematic decomposition strategy capable of identifying and vectorizing complex ternary reductions, including FMA operations to identify and vectorize complex ternary reductions, including operations such as fused multiply-add (FMA). This ability to decompose compound operations and recognize advanced reduction patterns fills a critical gap in the current MLIR infrastructure. Leveraging this approach, OML-reduction achieves up to 2.5× greater reduction coverage in neural networks compared to the default MLIR reduction pass.

3) Results: Our automatic transformation, OML-vect, unlocks the full potential of the MLIR affine super-vectorizer, minimizing dependence on manual vectorization. Evaluations across eight neural network and Transformer models on both x86 and RISC-V architectures show that automatic vectorization with OML-vect achieves 94% and 91% performance on x86 and RISC-V respectively, relative to the baseline. Moreover, it delivers 2% and 8% higher performance than manually vectorized libraries on x86 and RISC-V, respectively, demonstrating an efficient and portable solution for neural network deployment on edge devices.

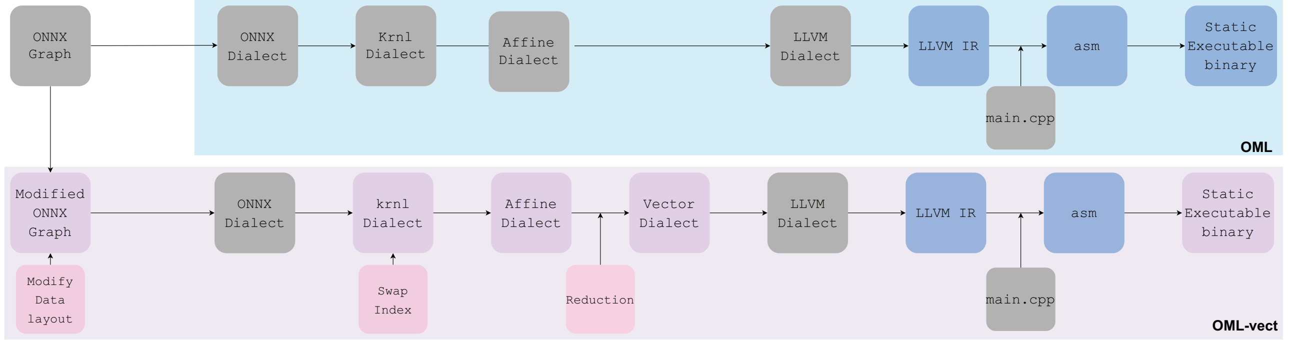
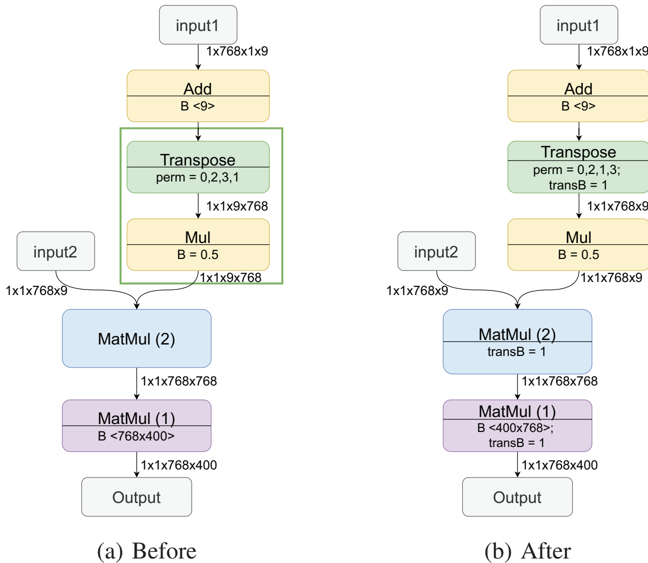


Fig. 2: Overview of OML, and OML-vect (generic-RISCV path from Figure 1) compilation flow [1]



(a) Before (b) After

Fig. 3: ONNX graph before and after data layout optimization. Figure from Alladi et al. [1]

C. Compiler assisted Instruction fusion : CAIF

Instruction fusion is a microarchitectural optimization that combines multiple μ-ops into a single macro-operation, improving throughput by executing more work per cycle and freeing hardware resources. Most benefits arise from fusing memory operations, so we focus on enabling fusion-friendly code layouts through compiler transformations. While cache data arrays can support fused memory accesses using existing logic, hardware fusion faces challenges in memory disambiguation—particularly when load/store queues must handle accesses crossing cacheline boundaries, which increases hardware complexity. To address this, we propose a compiler-driven instruction fusion strategy that reorders instructions to transform non-consecutive memory accesses into consecutive ones—without speculative hardware fusion. Unlike approaches such as Helios [29], which rely on complex predictive hardware, our method eliminates misprediction overhead, reduces energy and area costs, and achieves comparable performance with simpler hardware. We enable fusion in hardware by selective reordering of memory operations through Compiler-assisted instruction fusion (CAIF) [28].

Our experimental results show that for neural network workloads, hardware supporting only consecutive instruction fusion achieves an average performance gain of 1.2% over the no-fusion baseline when applications are compiled with a standard compiler, and 19.6% when compiled with CAIF. Furthermore, when non-consecutive hardware fusion (Helios) is enabled, CAIF improves performance from 6.6% to 20.3%. CAIF also demonstrates its effectiveness on more statically complex, general-purpose applications, achieving performance improvements on SPEC CPU 2017 from 2.4% to 6.4%, and from 14.4% to 17.7%, respectively, on an Intel Grace Mount–like microarchitecture configuration.

D. Static Memory Allocation Optimization: Idealloc

The problem of static memory allocation, where a set of buffers with predefined lifespans are mapped onto offsets in global memory) is critical in edge AI inference, where memory resources are scarce. In the context of AI compilers, memory allcoation consists of two phases: (i) planning and (ii) runtime data transfer management. The planning phase is further divided into bufferization, i.e., assigning memory buffers to tensors, and offset assignment, i.e., mapping each of the distinct buffers to contiguous chunks of on-chip memory.

To approach static memory allocation optimizations, we developed the *idealloc*[17] memory allocator as a bufferization mechanism for our compilation stack. *Idealloc*[17] integrates the $(2+\epsilon)$-approximate static memory optimization algorithm of Buchsbaum [26] into Convolve compilation stack, where ϵ is an input-dependent "sufficiently small" real number. *Idealloc* offers a low-fragmentation scalable allocator able to adapt to constrained memory configurations found in heterogeneous edge SoCs. Initial comparisons with prior-art in static memory allocators such as XLA[25] and minimalloc[24], idealloc achieves 100% success rate on allocation, as well as XLA, while minimalloc achieves 20% with a maximum fragmentation of up to 771.7 MiB. This is 0.48× the memory footprint XLA allocates, as it reports up to 1.6GiB fragmentation.

IV. SoC Accelerator Path

Snitch [32] is a CPU core designed by ETH Zurich to maximize performance per Watt. It is based on the RISC-V instruction set, with custom ISA extensions to express packed SIMD vector instructions, memory transfers between RAM

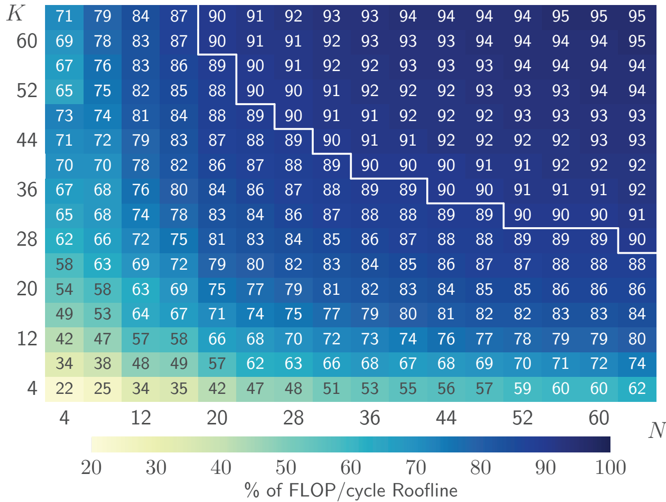


Fig. 4: Sustained throughput of the 64-bit MatMul kernel ($C_{M\times N} = A_{M\times K}B_{K\times N}$ when $M = 1$). We achieve throughput of over 90% ($\geq$ 1.80 floating/point operations (FLOPs)/cycle) of the theoretical peak (above the white border) as shape sizes increase, indicating that the computation offsets constant overheads. Figure from Lopoukhine et al. [22].

and tightly/coupled data memory (TCDM), loop unrolling, and memory access configuration. These operations correspond to features found in modern CPU hardware, namely caching, branch prediction, and out-of-order execution, and shift the burden from hardware to software for managing compute resources effectively. In order to generate software that will fully utilize the available hardware, both the high-level semantics of the code being executed and the low-level capabilities of the chip must be considered together. The input abstractions used by neural network frameworks contain the high-level information necessary to reason about control flow and memory access patterns, but are discarded by traditional MLIR flows before code generation is done by LLVM, limiting the ability to consider all constraints at the same time. To this end, we have developed a novel compiler backend to address this problem for linear algebra micro-kernel code generation.

A. A Multi-level Compiler Backend for Accelerated Micro-kernels Targeting RISC-V ISA Extensions

Our approach [22] extends MLIR's progressive lowering and multi-level abstractions into the backend. First, we implement a generic RISC-V backend using MLIR, implementing instruction selection and register allocation. Second, we extend this backend to generate assembly for the Snitch core. The resulting code reaches over 90% of peak performance on selected micro-kernels over 64-bit floating-point date. (Figure 4)

In contrast to LLVM's design, which uses different data structures and APIs for target-independent and target-specific code representations, we use MLIR's extensible IR design to represent code at all levels through assembly. MLIR uses *dialects* to group together definitions of intermediate representation (IR) components, allowing the use of multiple dialects within the same module of code. We leverage this mechanism to represent assembly-level code with a collection of custom dialects. One of these dialects represents low-level operations such as arithmetic, memory operations, and jumps. Two dialects represent higher-level control flow constructs such as function definitions and calls, and for loops. For loops in our target-specific dialect are represented similarly to the target-agnostic ones defined in MLIR, leveraging the recursive structure of MLIR IR to nest the operations comprising the loop body within the operation. Higher-level, target-specific constructs make it easier to reason about the live ranges of values in the IR, which we leverage for our reimplementation of register allocation. Our register allocator operates one function at a time, using a traditional approach for register allocation in static single assignment (SSA) [15] within a basic block, and leverages the structure of `for` loops to extend the live ranges of values used within a loop but defined outside. The `for` loops are thus lowered after register allocation. Instruction selection is implemented as peephole rewrites leveraging the existing rewrite infrastructure. This simple and modular infrastructure makes for an effective base to build on to add Snitch support.

MLIR's extensible design allows for easy extensions with custom dialects and transformations. We followed a similar design to our base RISC-V dialect to represent the ISA extensions at the assembly level. The floating/point repetition (FREP) instruction was an exception to this, as it's high-level nature is most directly represented as a loop operation with nested operations comprising its body. We also define a dialect for the high-level abstractions that encode the constraints on the accelerator capabilities. Finally, we implement a progressive lowering approach that maps high-level information provided by machine learning (ML) frameworks, such as parallel and reduction iterators and memory access patterns, to the high-level operations that express Snitch's accelerator capabilities.

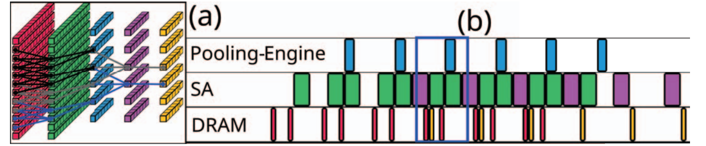


Fig. 5: (a) The periodic data-dependence of line-based layer-fusion of a simplified CNN composed of loads(red), 3x3 Conv(green and purple), 2x2 max-pool (cyan), store (yellow). (b) An execution trace of a CNN executed in a line-based layer-fused manner on an heterogeneous SoC, with highlighted a period of the periodic regime of the execution trace.

B. Layer-Fused Scheduling of CNNs

An important optimization opportunity for the execution of CNNs is layer-fusion (aka depth-first or cascaded execution), which consists in the interleaved execution of convolutional layers (Figure 5). The method has two advantages. Firstly, this method reduces the required memory of execution, allowing for a smaller on-chip memory footprint and fewer accesses to off-chip memory [14][31]. Secondly, this method allows the pipelining of computation and data-transfers, maximizing parallelism and utilization of processors and communication channels [31].

The frequent interleaved execution of convolutional layers makes compact schedule representation non-trivial. Hence, we adopt the formalism of Synchronous Dataflow Graphs(SDF) [20] to model the periodic data-dependence and eventually achieve a compact schedule representation. Such periodic data-dependence highlighted in Figure 5(a). Further, the modeling of the periodic data-dependence of layer-fusion as an SDF allows using proven compact scheduling methodologies [6][30].

To enable a scheduling that is aware of the hardware and architectural properties of the system, we have adapted the scheduling flow of Stream [31] with an SDF based scheduling methodology (Stream being an analytical design space exploration tool best fit for analysis of layer-fused CNN computations on heterogeneous SoC). To facilitate an SDF based scheduling methodology, the data-dependence representation of line-based layer-fusion, internal to Stream, was adapted to be modeled as an SDF.

Intuitively, the periodic regime of the schedule may be used to reconstruct the entirety of the schedule (Figure 5(b)). The obtained schedules are asserted to be valid schedules by the internal validity checks performed by Stream, and a periodicity aware schedule provides schedule representation, at times, 500x smaller in the number of recorded events, compared to the execution trace provided by an unmodified flow of Stream.

V. CGRA PATH

On the reconfigurable side, accelerators require configuration for programming both the data transfers between heterogeneous Processing Elements (PEs) within the architecture and the PEs themselves. The CGRA accelerator included in the CONVOLVE's heterogenous SoC, is based on the R-Blocks [7] architecture extended with two vector lanes of approximate and accurate MAC arithmetic (SIMD configuration). The CGRA implements two Network on Chip (NoC) infrastructures for routing data and control signals, respectively.

The CGRA's compilation stack is supported from the OpenASIP [16] suite. It accepts as input a C-code kernel and the CGRA architecture description, generating the LLVM-IR. The IR is scheduled as a Transport-Triggered Architecture (TTA) back-end [11], according to the micro-architectural constraints (#PEs, #LocalMemoryModules, #RouterChannels etc), producing the CGRA executable bitstream. Intrinsics in the C-code input generate the mapping to specialized HW-operators, e.g. approximate vector lanes, such that both SISD/SIMD parsing and the approximate multiplication are handled from the C-level, providing flexibility on kernel mapping.

This approach is evaluated on two benchmarks: General Matrix-Matrix Multiplication (GEMM - red bars) and General Matrix-Vector Multiplication (GEMV - blue bars), which are common kernels in many ML applications (6). Two configurations were selected for our CGRA compilation stack: no compiler optimizations (Scalar - dark colors) and intrinsic-driven 8-lane vector implementation (light colors). The vector optimized version achieves a speedup of 3.6× compared to the unoptimized version on the GEMM benchmark, which drops to

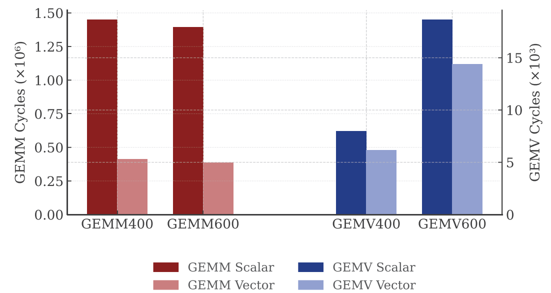


Fig. 6: Execution time in vectorized vs. scalar CGRA configs

1.3×. Powerwise, architectures with vector-based optimization can be up to 33% more energy efficient than scalar-based.

VI. CONCLUSION

This work presents a vision of compiler construction for effective code generation for heterogeneous AI accelerators.

Recognizing that the future of SOC will be heterogeneity, we observe that hardware platforms become increasingly diverse, which raises compiler complexity. We address this challenge through multi-path, modular compiler designs that enable optimization across different execution strategies and hardware configurations. Our approach is flexible, target-extensible, supports analysis and co-design, and facilitates interoperability across diverse applications and platforms. By virtue of being embedded in the MLIR-based ecosystem of compiler tools, components can be gracefully integrated into larger compilation flows, using a common IR as the interchange format. Thus, the tools excel at their particular problem area, whether it is code generation for novel hardware, specialized algorithms for memory allocation, or high-level neural network scheduling. The components described in this paper bear this out, reaching over 90% peak performance for novel targets (section IV), reducing memory footprint by 35% (subsection III-B, and 33% higher energy efficiency (section V, and improving over existing results for energy efficiency and performance. Developed directly with the domain experts developing either the target hardware or machine learning use-cases, each compiler component encodes the relevant domain-specific knowledge for transformations at the appropriate level of abstraction. The modular design of the CONVOLVE compiler suite enables rapid iteration cycles for its users, forming a productive foundation for AI hardware software co-design. Further details are available in the CONVOLVE publications [1, 28, 22, 29, 13, 12, 5, 17] and in CONVOLVE publication website [10].

VII. ACKNOWLEDGMENT

This work has been funded by the EU's Horizon 2021 research and innovation program (CONVOLVE, g.a. no. 101070374 under HORIZON-CL4-2021-DIGITAL-EMERGING-01).

References

[1] S Alladi et al. "Enabling Automatic Compiler-Driven Vectorization of Transformers". In: *Proceedings of the 25th ACM/IEEE Int. Symp. Code Gener. Optim. (CGO '26)*. Sydney, NSW, Australia: Association for Computing Machinery, 2026.

[2] S Alladi et al. *OML-vect: Enabling Automatic Compiler-Driven Vectorization of Transformers*. Dec. 2025. DOI: 10.5281/zenodo.18006548.

[3] R A Antonio et al. *An Open-Source HW-SW Co-Development Framework Enabling Efficient Multi-Accelerator Systems*. DOI: 10.48550/arXiv.2508.14582.

[4] *Atrevido 423 with V8 vector unit (online)*. https://semidynamics.com/en/products/atrevido. 2025.

[5] Siddharth Bhat et al. "Verifying Peephole Rewriting in SSA Compiler IRs". In: *15th International Conference on Interactive Theorem Proving (ITP 2024)*. Ed. by Yves Bertot et al. Vol. 309. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. DOI: 10.4230/LIPIcs.ITP.2024.9.

[6] S S Bhattacharyya et al. *Software synthesis from dataflow graphs*. Springer Science & Business Media, 2012.

[7] B de Bruin et al. "R-Blocks: an Energy-Efficient, Flexible, and Programmable CGRA". In: *ACM Trans. Reconfigurable Technol. Syst.* 17.2 (). DOI: 10.1145/3656642.

[8] I Colonnelli et al. "Experimenting with PyTorch on RISC-V". In: Barcelona, Spain.

[9] *CONVOLVE*. https://convolve.eu.

[10] *CONVOLVE Publications*. https://convolve.eu/scientific-publications/.

[11] Henk Corporaal et al. "Using Transport Triggered Architectures for Embedded Processor Design". In: *Integr. Comput.-Aided Eng.* 5.1 (Jan. 1998), pp. 19–38.

[12] Mathieu Fehr et al. "First-Class Verification Dialects for MLIR". In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025). DOI: 10.1145/3729309.

[13] Mathieu Fehr et al. "xDSL: Sidekick Compilation for SSA-Based Compilers". In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '25. Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 179–192. DOI: 10.1145/3696443.3708945.

[14] Koen Goetschalckx et al. "Breaking High-Resolution CNN Bandwidth Barriers With Enhanced Depth-First Execution". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 323–331. DOI: 10.1109/JETCAS.2019.2905361.

[15] Sebastian Hack et al. "Register Allocation for Programs in SSA-Form". In: *Compiler Construction*. Ed. by Alan Mycroft et al. Berlin, Heidelberg: Springer, 2006, pp. 247–262. DOI: 10.1007/11688839_20.

[16] Kari Hepola et al. *OpenASIP 2.0: Co-Design Toolset for RISC-V Application-Specific Instruction-Set Processors*. July 2022.

[17] C Lamprakos et al. *Futureproof Static Memory Planning*.

[18] Chris Lattner et al. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, CGO*. IEEE, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[19] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[20] Edward A Lee et al. "Synchronous data flow". In: *Proceedings of the IEEE* 75.9 (2005), pp. 1235–1245.

[21] Qiankun Liu et al. "Exploring RISC-V Based DNN Accelerators". In: *2024 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. 2024, pp. 1–6. DOI: 10.1109/COINS61597.2024.10622495.

[22] Alexandre Lopoukhine et al. "A Multi-level Compiler Backend for Accelerated Micro-kernels Targeting RISC-V ISA Extensions". In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '25. Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 163–178. DOI: 10.1145/3696443.3708952.

[23] Alexandre Lopoukhine et al. *Artifact of "A Multi-Level Compiler Backend for Accelerated Micro-Kernels Targetting RISC-V ISA Extensions"*. Version 1.0. Nov. 2024. DOI: 10.5281/zenodo.14052014.

[24] Michael D. Moffitt. "MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning". In: ASPLOS '23. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 238–252. DOI: 10.1145/3623278.3624752.

[25] *OpenXLA*. https://openxla.org/.

[26] Sriram V Pemmaraju et al. "Buffer minimization using max-coloring." In: *SODA*. Vol. 4. 2004, pp. 562–571.

[27] R Reddy et al. *CAIF*. https://github.com/CAPS-UMU/llvm-project-caps. Dec. 2025.

[28] R Reddy et al. "Compiler-Assisted Instruction Fusion". In: *Proceedings of the 25th ACM/IEEE Int. Symp. Code Gener. Optim. (CGO '26)*. Sydney, NSW, Australia: Association for Computing Machinery, 2026.

[29] S Singh et al. "Exploring Instruction Fusion Opportunities in General Purpose Processors". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 199–212. DOI: 10.1109/MICRO56248.2022.00026.

[30] Sundararajan Sriram et al. *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2018.

[31] Arne Symons et al. "Stream: Design space exploration of layer-fused dnns on heterogeneous dataflow accelerators". In: *IEEE Transactions on Computers* (2024).

[32] Florian Zaruba et al. "Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads". In: *IEEE Transactions on Computers* 70.11 (Nov. 2021), pp. 1845–1860. DOI: 10.1109/TC.2020.3027900.