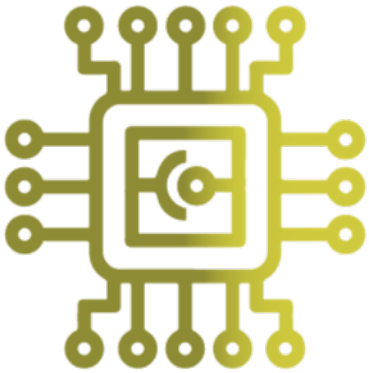
Ref. Ares(2026)4318414 - 27/04/2026



CONVOLVE

Seamless Design of Smart Edge Processors

Project funded by the European Commission

under the grant agreement number 101070374

Deliverable 5.4

Integrated Tool Flow



CONVOLVE

Title of deliverable	Integrated Tool Flow
WP contributing to the deliverable	WP5
Task contributing to the deliverable	Task 5.4
Dissemination Level	PU - Public
Due submission date	June 2025
Authors	Tobias Grosser [CAM] Alexandre Lopoukhine [CAM] Siddharth Bhat [CAM] Georgios Alexandris [ICCS] Sotirios Xydis [ICCS] Alexandra Jimborean [UMU] Emily Sillars [UMU] Shreya Alladi Subhash Chandra [UMU] Ravikiran Ravidranath Reddy [UMU]
Internal reviewers	Henk Coporaal [TUE] Mottaqiallah Taouil [TUD] Ouassim Karrakchou [UIR]

Document Version	Date	Change
V0.1	06/02/2025	Initial Version
V0.2	19/12/2025	First complete draft
V0.3	26/03/2025	Feedback Incorporated
V1.0	27/04/2026	Final version

Executive Summary

CONVOLVE's 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 design stack***. WP5 provides flexible, transparent, and secure compilation tools to accelerate this lifecycle. It adheres to CONVOLVE's design methodology, specifically the ***"compositional and fast design flow"*** pillar. By offering an open and transparent ("gray-box") compilation environment, we enable the development of abstractions and tools for mapping CONVOLVE's use cases to its many-accelerator SoC platform. We provide an ***integrated compilation flow built on an ecosystem of loosely coupled tools.*** In particular, we contribute:

- A ***high-level programming environment*** through our Python-based MLIR framework, xDSL, with over 450 GitHub Stars, 4600 commits, and 100 contributors, and for which we delivered numerous new productivity advances over the last months ***(CAM)***
- Frontend to ***support CONVOLVE critical edge AI application use cases (UMU)***
- ***An end-to-end compiler for AI apps on RISC-V and custom software optimizations (UMU)*** (also D5.3)
  - Data layout optimizations and instruction Fusion **(UMU)**
  - Reduction detection to enable automatic vectorization **(UMU)**
- ***Reducing compilation time*** by reusing the results of complex state-of-the-art alias analyses across compilation phases and even across different compiler runs **(UMU)**
- ***Accelerator configuration abstractions*** for SNAX-compatible accelerators ***(CAM, KUL)***
- An ***end-to-end PyTorch to accelerator compiler***, *Quidditch*, based on the open IREE compiler ***(CAM)***
  - Integration of peak-performance micro-kernel generator for Snitch PE ***(CAM)***
  - Compiler support (Quidditch) for the Snitch clusters as used in CONVOLVE's Chimera SOC ***(CAM)***
  - Preliminary design for SNAX-based acceleration ***(CAM)***
- Integrating advanced memory allocation schemes (developed as part of deliverable D5.5) within our Quidditch compiler ***(ICCS, CAM)***
- Integrating a tiling cost model for the snitch cluster into Quidditch ***(UMU, CAM)***
- Initial exploration of ZigZag integration in Quidditch, with further work delegated to the complementary Praxis compilation flow **(UMU, KUL)**
- Cost model and its relationship to constrained-aware space exploration and PRAXIS by KUL (PRAXIS includes ZigZag integration). **(UMU, KUL, CAM)**
- ***Foundations for secure compilation*** via formal semantics for core MLIR abstractions ***(CAM)*** Covering a formal model of MLIR's SSA model, First-class semantic dialects for MLIR, and a formal semantics for RISC-V in Lean.

We contribute the following concrete results under these overarching goals:

1. Over 90% FPU utilization, of ETH Zurich Snitch accelerators at a single processing element (PE).

2. Efficiently map partner use cases to CONVOLVE's multi-accelerator SoC, achieving up to 22x speedup on Snitch clusters for Jabra's NsNet2 over state-of-the-art compiler (LLVM).

Our gray-box compiler has been instrumental in navigating the fast-changing AI ecosystem, enabling collaboration across work packages and fostering active contributions to our open compilation stack. We also organized numerous workshops around xDSL, plus two MLIR Schools in Jan and Sep 2025.

Table of Contents

1 Introduction and Objectives **7**
2 Applications and Target Platforms **8**
2.1 Application Use Cases 8
2.2 Target Platforms 10
2.2.1 Snitch: An Energy-Efficient Compute Core 11
2.2.2 Occamy: A Chiplet-Based Many-Core AI Accelerator 12
2.2.3 OpenGeMM / SNAX: A High-Utilization GeMM Accelerator Generator 13
2.2.4 CGRA Accelerators 13
3 Architecture & Compilation Paths **15**
3.3 Foundational Compilation Frameworks 18
3.3.1 LLVM 18
3.3.3 MLIR 19
3.3.5 IREE 20
3.3.6 ONNX-MLIR 20
3.3.7 OpenASIP 21
3.4 Our High-Level Programming Environment xDSL 21
3.5 Memory Allocation 22
4 Universal Compilation Flow **23**
4.1 End-to-End: RISC-V via Stand-Alone Baseline Libraries 24
4.1.1 Optimized using default optimizations of the mainstream compilers 26
4.1.2 Hand-compiled libraries for RISC-V to support ONNX 26
4.2 Custom optimizations: OML-Vect 27
4.3 Leveraging Alias analysis without porting 27
4.4 Custom optimizations: LLVM RISC-V Backend (CAIF) 28
4.5 Evaluation and Results Universal Flow 29
4.5.1 OML Results 30
4.5.2 OML-Vect Results 31
4.5.3 LLVM RISC-V Backend Optimizations Results 32
5 Accelerated Flow (LinAlg + AI) **32**
5.1 Efficient Micro-kernel Compilation for the Snitch PE 33
5.2 Mapping a DNN to MicroKernels 34
5.2.1 Code Generation Pipeline 35
5.2.1.1 Matrix Multiply Pipeline 36
5.1.1.2 Lowering Pass Pipeline 37
5.2.2 TCDM Management 37

5.2.3 Double Buffering 38
5.3 Cost Model Integration for Automatic Tiling 38
5.3.2 Proof of Concept: ZigZag Integration 39
5.3.2.1 Vanilla RISC-V on the Snitch Cluster 39
5.3.2.2 "True" Snitch Backend: RISC-V + Snitch Extensions 40
5.3.2.3 Pivot to Designing the Myrtle Cost Model 40
5.3.3 Myrtle Cost Model for Automatic Tiling 41
5.3.3.1 Cost Model Design 41
Mode C: Manual C Code 41
Mode Q: Quidditch 42
5.3.4 Myrtle Integration with Quidditch 42
Effect on Compilation Time 43
Open Source Availability 43
5.4 OpenGeMM/SNAX Acceleration 44
5.5 Memory Allocation 44
5.5.1 Source Code Changes 45
5.5.2 Build System Changes 49
5.5.3 Compilation Latency Overheads 50
5.6 Evaluation and Results Accelerated Flow 51
5.6.1 Default AI Pipeline vs Quidditch for Accelerator Programming 51
5.6.2 Myrtle Tiling Results 52
Mode C: Manual C Code 52
Mode Q: Quidditch 52
5.6.3 Memory Allocation Optimization 53
6 CGRA Compilation Flow 54
6.1 CGRA Scheduling Flow 54
6.2 Approximation-Aware Scheduling 55
6.3 CGRA Scheduling Optimizations and Results 56
7 Semantics for Security and Search 59
7.1 A Formal Model of SSA via Lean-MLIR 60
7.2 First-class Semantic Dialects For MLIR 61
7.3 Formal Semantics for RISC-V in Lean 62
8 Training Activities and Outreach 63
9 Conclusion 64
Highlights of the Convolve Project — Compiler Contributions 65
Performance: Hardware Acceleration & Beyond 66
Demos 66
Appendix 68

A1) Key Publications from CONVOLVE Members 68
A2) Abbreviations, Acronyms, and Names 71
A3) References 74

1 Introduction and Objectives

Accelerating the design of smart edge AI processors enabled by a flexible and highly effective software-hardware stack is the central CONVOLVE mission.

In WP5, this mission was set via the following core objectives:

- **01:** Design a ***modular compiler framework*** that ensures ***security, reliability, and performance*** by maintaining and exploiting constraints and opportunities from application to hardware.
- **02:** ***Developer-guided domain-specific languages*** to target many-accelerator ULP hardware, generated for the same, write-once frontend code, guaranteeing compatibility for future accelerators.
- **03:** ***Semi-automatic & assisted compilation*** for ULP hardware from general-purpose languages, via ***opportunity-matching through static analysis*** and test-oriented generation.
- **04:** ***Choose application variants*** that use a given heterogenous ULP many accelerator SOC ***effectively***.

The "Integrated Tool Flow" (D5.4) addresses these objectives jointly with the static analysis deliverable D5.3, focusing on the infrastructure and integration aspects of these challenges. In particular, the challenge to establish a compiler framework, make it secure and performant, and enabling developers to take a front-row seat when optimizing for their accelerators.

A key challenge for the CONVOLVE compilation flow is diversity across numerous dimensions of this software-to-hardware mapping. Fast-evolving neural network designs, numerous highly specialized hardware accelerators, and a diverse set of compilation frameworks for modern edge hardware create an unlimited design space, which a one-size-fits-all approach cannot cover. Hence, WP5 decided to built a set of foundational building blocks that interconnect to end-to-end compilation flows.

In the following deliverable, we keep the introduction and the work of the first 13 months of CONVOLVE short and report on the second reporting period in WP5.

We introduce in Section 2 the applications and target platforms WP5 worked on and give in Section 3 a high-level overflow over the compilation tools and flows developed in WP5. We introduce in Section 4 our universal compilation flow, present in Section 5 our accelerator workflow and in Section 6 our CGRA flow. Section 7 talks about our work on establishing formal semantics for security and search during compilation. Section 8 discusses our training and outreach activities, while Section 9 concludes.

2 Applications and Target Platforms

We first introduce background and related work covering application use cases, target platforms, and compiler frameworks.

2.1 Application Use Cases

In the following table, we present use cases from our partners as well as some external NN models from which computationally intensive kernels have been identified. A detailed analysis of these use cases was given in D5.1. Key insights that influenced the compiler design are:

- The application use cases range from ***kernels to full neural networks***, with full networks dominating
- All application use cases are available as ***PyTorch or ONNX input*** files, which a compiler framework should connect to
- There is ***a need for supporting all use cases*** in the CONVOLVE compiler
- Show for that particularly important kernels and networks can be executed on accelerators at ***peak performance through domain-specific kernel code generation***
- Significant domain expertise across the convolve project is needed to deliver the results expected.

To address these concerns we designed an integrated tool flow architecture, described in detail in Section 3, which allows us to compile the numerous DNNs and kernels arising in the CONVOLVE application use cases. All inputs are supported by our tool flow and we map a selection to run on accelerators available in the CONVOLVE project. **In the "Compilers Support" column, we identify which flow of the Convolve compiler we use to support kernels from each of these models. In the "Results" column we identify the sections that provide results for the different application use cases.** The Universal Flow (made up of two parts, OML/OML-Vect and CAIF) is the default RISC-V compilation path and acts as a "safety-net" for being able to fully run the Convolve use cases. The Accelerated Flow (LinAlg+Tensor) and Accelerated CGRA Flow act as accelerator-specific compilation paths.

Use Cases	Relevant Kernels	Compiler Support	Performance Results
NsNet2	Linear GRU Sigmoid ReLU	Universal Flow *(Section 4)* (OML, OML-Vect, CAIF) Accelerated Flow *(Section 5)* CGRA Flow *(Section 6)*	4.5.1 - OML Results 4.5.2 - OML-Vect Results 4.5.3 - CAIF Results 5.6.1 - Accelerated Results (kernels + full network) 6.3-Accelerated CGRA Results
ConvTasNet	Conv1D ConvTranspose1D Sigmoid PReLU GroupNorm Conv1D: point-wise, depth-wise, causal	Universal Flow (OML)	Section 4.5.1 - OML Results Section 5.6.1 - Accelerate Results (selected kernels)
Yolov6	Conv2D	Universal Flow (OML-Vect) CGRA Flow	Section 4.5.1 - OML Results Section 4.5.2 - OML-Vect Results Section 5.6.1 - Accelerate Results (selected kernels)
Yolov7	Conv2D	Universal Flow (OML-Vect)	Section 4.5.1 - OML Results Section 4.5.2 - OML-Vect Results Section 5.6.1 - Accelerate Results (selected kernels)
TCNN	Linear Conv1D ReLU BatchNorm1D PReLU Conv1D: point-wise, depth-wise, causal	Universal Flow (OML)	Section 4.5.1 - OML Results Section 5.6.1 - Accelerate Results (selected kernels)
Transformers	**Relevant Kernels**	**Users**	
LLaMav2	GeMM	Universal Flow (OML-Vect) CGRA Flow	Section 4.5.1 - OML Results Section 4.5.2 - OML-Vect Results

Bert Whisper Roberta GPTNeox Googlenet	MatMul (from attention layers) SoftMax	Universal Flow (OML-Vect, CAIF)	Section 4.5.1 - OML Results Section 4.5.2 - OML-Vect Results Section 4.5.3 - CAIF Results
CNNs	**Relevant Kernels**	**Users**	
Mobilenet v2 Mobilenet v3 Resnet18 Resnet50 MNIST	Conv1D, Conv2D, Conv3D ReLU Clip Sigmoid SoftMax	Universal Flow (OML-Vect, CAIF)	Section 4.5.1 - OML Results Section 4.5.2 - OML-Vect Results Section 4.5.3 - CAIF Results Section 5.6.1 - Accelerate Results (selected kernels)
Feedforward NNs	**Relevant Kernels**	**Users**	
MLP	Full Model	Universal Flow (OML-Vect, CAIF)	Section 4.5.1 - OML Results Section 4.5.2 - OML-Vect Results Section 4.5.3 - CAIF Results

Table 1. A summary of CONVOLVE partner use cases focusing on speech and audio processing, some additional NN models tested, and their corresponding important kernels.

2.2 Target Platforms

We are first offering additional details on some of the hardware WP5 is targeting. In particular, we target the Snitch Linear Algebra Accelerator, the Occamy Many-Core processor, KU Leuven's OpenGeMM / SNAX accelerator template generator, as well as CGRA accelerators.

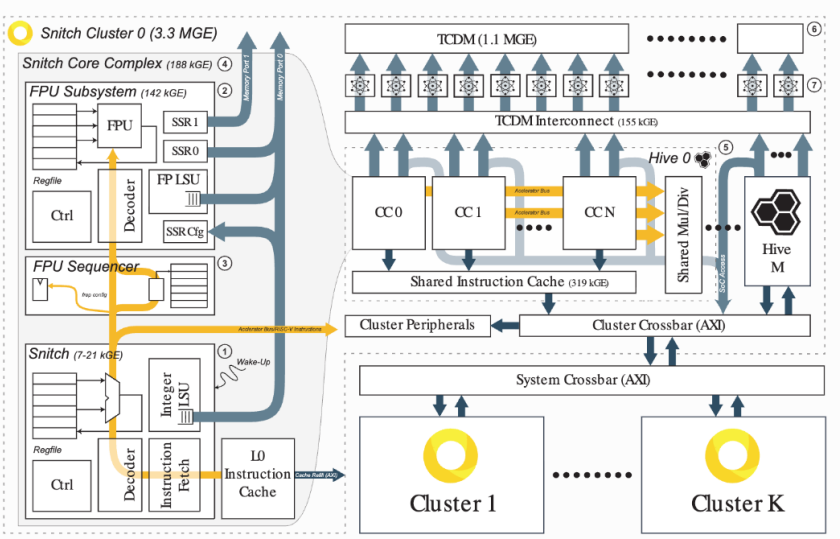


Fig. SNITCH A FMA Linear-Algebra Accelerator with Streaminging Extensions (https://iis-people.ee.ethz.ch/~zarubaf/snitch)

2.2.1 Snitch: An Energy-Efficient Compute Core

Snitch is a research-developed, in-order, 32-bit RISC-V core aiming at area and energy-efficient execution of compute-intensive floating-point (FP) workloads. It foregoes many expensive and complex features that are standard in modern CPUs, such as branch prediction, multiple execution units, register renaming, multi-issue frontends, and out-of-order execution. Instead, Snitch offers the following novel instruction set architecture (ISA) extensions:

- A set of stream-enabled FP registers replace explicit load-store instructions with prefetching of subsequent memory accesses that follow preconfigured affine patterns, hiding their latency.
- A hardware repetition instruction expresses iteration over FP instructions, eliding the typical use of unstructured control flow (i.e., branching) and other bookkeeping (i.e., induction variables).

Combining these extensions allows the decoupled operation of the floating-point unit (FPU) from the other parts of the core, alleviating the von Neumann bottleneck by eliding the need for explicit integer, control flow, and memory access operations. Snitch achieves peak FPU utilization and has been successfully integrated as the base compute unit of several parallel, hierarchical architectures at scale (e.g., Manticore, MemPool, TeraPool, STX, and Occamy).

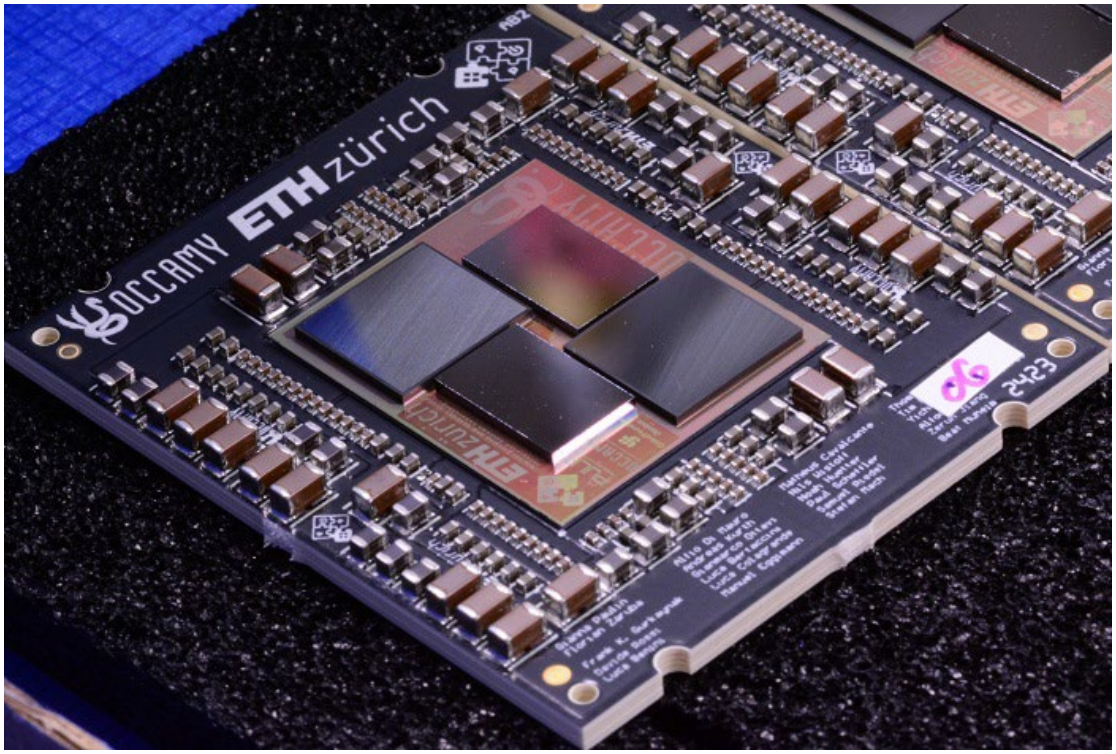


Fig. OCCAMY A Chiplet-Based Many-Core AI Accelerator based on Snitch (https://pulp-platform.org/occamy)

2.2.2 Occamy: A Chiplet-Based Many-Core AI Accelerator

Occamy (Fig. OCCAMY) is a flexible, general-purpose, ultra-efficient, RISC-V, chiplet-based architecture for data-parallel FP workloads. In more detail, it consists of two chiplets, each coupled with a 16GiB high-bandwidth memory (HBM), where each chiplet is organized in six groups of four clusters. Each cluster comprises nine Snitch cores, one operating as a direct memory access (DMA) unit, and the remaining eight serving as CUs. The Snitch cores for each cluster share 128 KiB of tightly coupled data memory (TCDM), which is used both as a small, fast (single-cycle) access memory and as the memory aliased by stream-enabled FP registers. The DMA unit is responsible for data movement between TCDM and HBM via a high-bandwidth 512-bit data interconnect. Moreover, each chiplet integrates a Linux-capable, 64-bit RISC-V host core which manages the groups, peripherals and work dispatch. A runtime application programming interface (API) exposes several initialization, synchronization, data movement and introspection primitives (e.g., all cores in a cluster can be synchronized via a barrier or a core can identify if it is a compute or DMA unit). Occamy achieves 1.7× to 5.2× higher FPU utilizations when compared to state-of-the-art CPUs and GPUs.

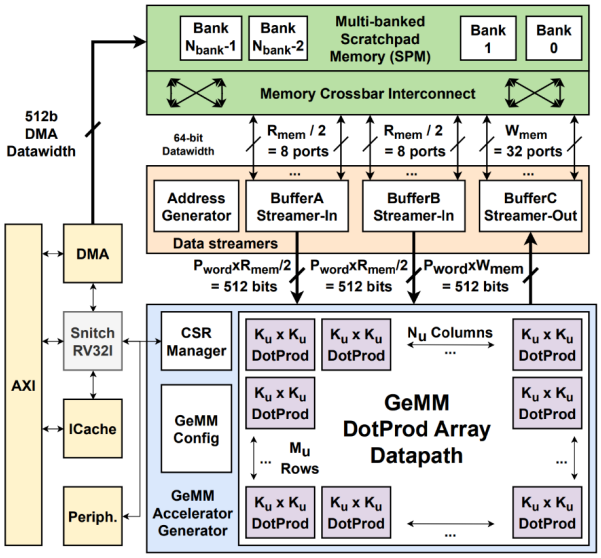


Fig. OpenGeMM A high-utilization GeMM Accelerator Generator (Yi et al. 2025)

2.2.3 OpenGeMM / SNAX: A High-Utilization GeMM Accelerator Generator

Starting from the original Snitch Cluster, KU Leuven developed for CONVOLVE a novel high-utilization GeMM accelerator named OpenGeMM [20]. OpenGeMM (codename SNAX) is a parameterized Chisel-coded GeMM accelerator. OpenGEMM uses the same high-level layout as a Snitch Cluster (Fig. OpenGeMM): a lightweight RISC-V processor (Snitch) paired with a tightly coupled multi-banked scratchpad memory. In contrast to the original snitch, OpenGeMM replaces the stream floating point FMA unit with a Chisel-coded GeMM accelerator. OpenGeMM then offers three techniques to improve performance: configuration pre-loading, input pre-fetching with output buffering, and programmable strided memory access. Especially the SNAX configuration pre-loading requires novel techniques to ensure that the end-to-end CONVOLVE compiler can effectively target OpenGeMM.

2.2.4 CGRA Accelerators

CGRAs are a widely used type of hardware accelerator, combining flexibility with great performance gains. They are more application-specific than an FPGA, gaining performance, and also more reconfigurable than an ASIC, providing a great performance/reconfigurability trade-off to the community.

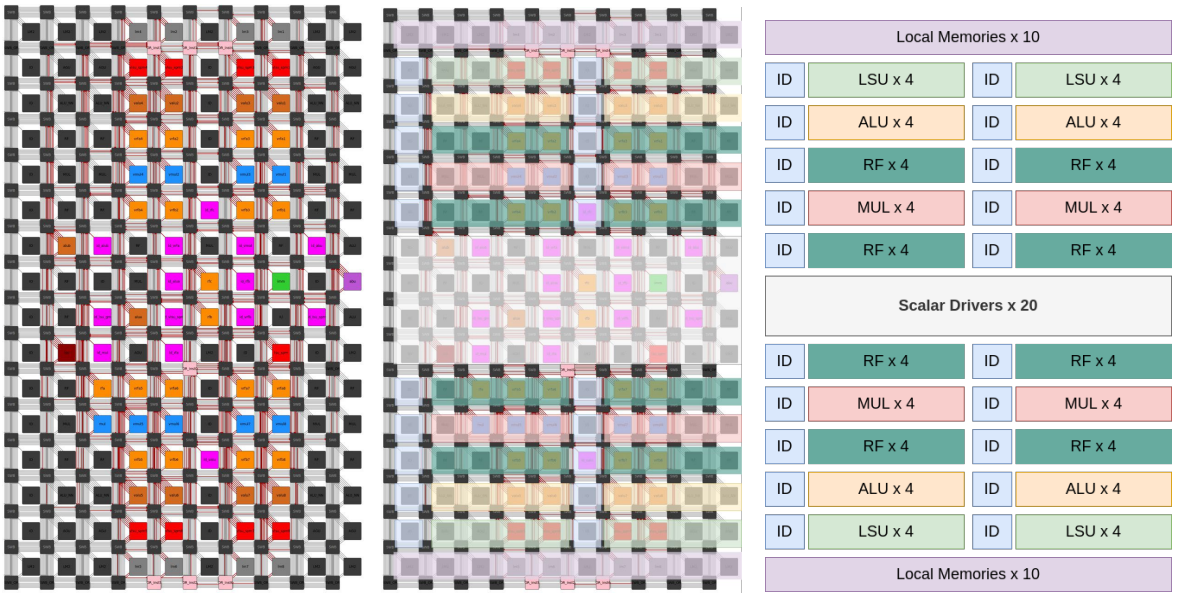
In the CONVOLVE project, we utilize the R-Blocks CGRA [33], in which we introduce state-of-the-art approximate multiplication units based on DRUM [34], gaining more power and area, with minimal accuracy loss on the evaluation use-cases.

In the CGRA architecture, each tile in the architecture can be individually configured to perform various operations and includes internal local SRAM memory to facilitate faster data transfers. In addition to standard processing elements (PEs), the system also incorporates approximation multiplication tiles, referred to as Ax MUL.

In the scalar configuration (SISD), each tile is connected to its own Instruction Decode (ID) tile. Alternatively, in the vectorized configuration (SIMD), multiple tiles can be controlled by a single ID, executing the same instruction across all participating PEs, with the vector size determined by the number of tiles involved.

The tiles are interconnected via two programmable Network-on-Chip (NoC) infrastructures—one dedicated to control signals and the other to data transfer. These networks handle both instruction words and data segments, supporting direct tile-to-tile communication. The NoC is structured as a standard 2D mesh and utilizes Wilton switchboxes to ensure robust and scalable connectivity.

Approximate R-Blocks CGRA is configured as an L2 accelerator inside the CONVOLVE project, so it communicates directly with AXI bus with the memory and host system. This interface includes a Master port for accessing external memory address spaces and a Slave port used to load the executable program (i.e., the CGRA bitstream) into the local instruction memory, as well as to configure the functional units.



A B C

Fig.ICCS2 Finalized CGRA Architecture

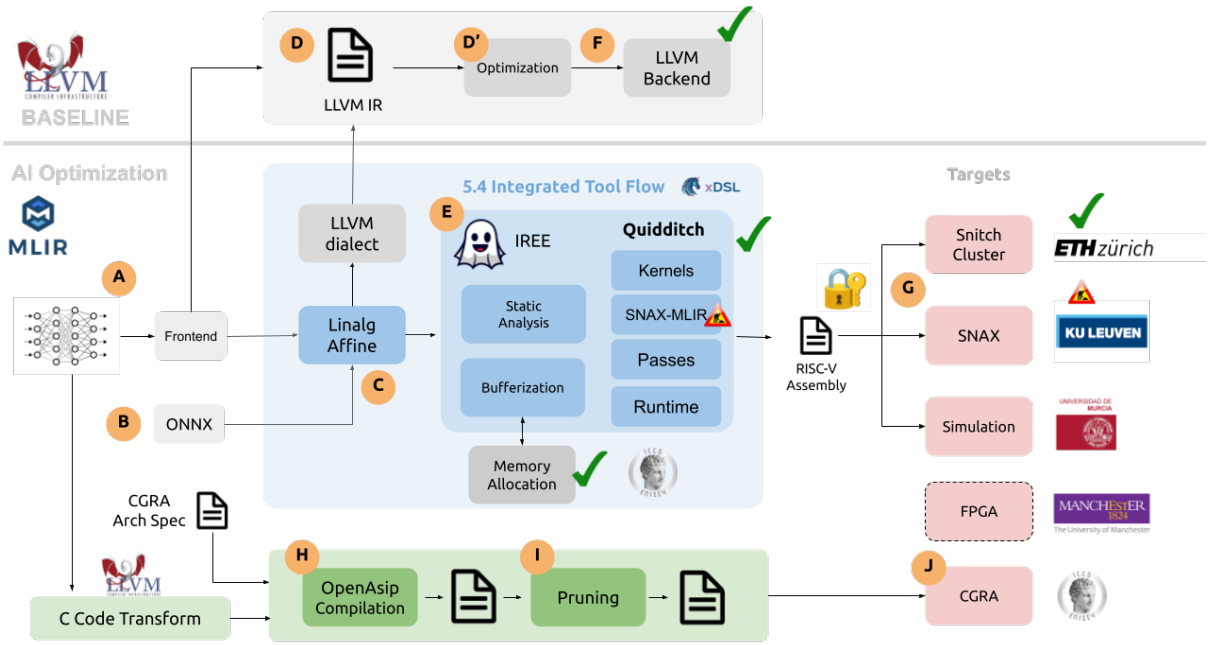
Figure ICCS2 represents the finalized architecture we use in the project to accelerate the CONVOLVE-given DNN cases. The placement of different tiles, as well as the data channels' interconnections, are present in Figure ICCS2.A, and the abstractly defined resource report is preset in Figure ICCS2.C with Figure ICCS2.B defining how the mapping represents each resource allocation area. Each colored line in ICCS2.C represents either 2 vectors of size 4 or one vector of size 8. Our accelerator, up to now, is utilizing a **silicon area of 1.1mm²** with the power consumption of **17.1mW**. Performance-wise, the approximate CGRA can achieve **7.42 peak GOPs** with the **300 GOPs/W** metrics based on the inference of the conv2d layers of ERBlock4 of the **YOLOv6s CNN**, which is provided by the ViNotion CONVOLVE partner, as well as the **NSNet2 RNN** provided by Jabra CONVOLVE partner.

3 Architecture & Compilation Paths

Our compilation framework bridges deep learning applications with heterogeneous accelerator platforms through a modular, progressive approach. The flow ingests neural networks from partner frameworks, transforms them through progressive lowering to optimized linear algebra primitives, and generates efficient code for target architectures - including generic RISC-V cores, base processing elements (Snitch Clusters) and specialized accelerators (SNAX). This end-to-end flow combines hardware-aware optimizations with simulation capabilities to maximize accelerator utilization while supporting dynamic neural networks and novel ML architectures.

The main advantages of a modular compilation approach are:

1. **Frontend Flexibility**: Supports ingestion from multiple ML frameworks while maintaining a common IR for hardware-agnostic optimizations.
2. **Target Extensibility**: Enables pluggable backend support for diverse architectures (Snitch/SNAX and others) through shared interfaces and lowering passes.
3. **Enabling Co-Design**: Progressive feedback-driven compilation (via simulation and utilization analysis) drives joint optimization of algorithms and hardware capabilities.
4. **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).



Within our CONVOLVE modular compilation framework, neural networks can traverse multiple transformation paths depending on the target capabilities:

Section 4

- **Universal Flow:** a universal flow through a state-of-the-art general-purpose compiler (LLVM) to the standard RISC-V ISA. This represents our default execution path, connecting CONVOLVE use cases with the hardware (SoC RISC-V host and SoC RISC-V Snitch cluster), not optimized for the accelerator targets. This path serves as a generic, robust path, i.e. a "safety-net" for being able to compile, optimize, and run the CONVOLVE use cases entirely, including the parts for which no dedicated accelerator has been built. Moreover, it provides a performance baseline for the kernels that can be accelerated on the CONVOLVE hardware. **(A->B->C->D->F)**

  +Universal CPU-Style Abstractions, +Industry Grade Compiler (LLVM), +All Use Cases

- **Universal Flow with RISC-V Acceleration: General-purpose RISC-V ISA:**
  - OML/OML-vect: from high-level descriptions, enabling vectorization and integrating hand-tuned libraries with ONNX support **(A->B->C->D->D'->F)**
  - CAIF: from high-level descriptions using backend target-specific optimizations **(A->D->D'->F)**

Section 5

- **Accelerated Flow (LinAlg+Tensor Hardware)** **(A->C->E->G)**: This compilation flow can target hardware accelerators to take advantage of specialized ASICs for accelerating our networks. In particular, we target:
  - on Snitch PE cluster utilizing tailored micro-kernel compilation
  - on SNAX fixed-function GEMM accelerator

+High-Level Abstractions, +Industry-Standard AI Framework (MLIR),
+Direct Access to Domain-Specific Accelerators,
+Absolute Controlt Matching Human-Designed Accelerators,
+Peak-Performance

Section 6

- **CGRA Flow** An accelerated flow specialized for targeting the CGRAs. **(H->I->J)**

+Independence of Compilex Frameworks, +Minimal Coupling
+Maximize Development Speed and Code Reuse for Stand-Alone Compilation Task

Section 7

- **Secure Compilation:** We establish a secure compilation flow for the foundational parts of Section 4, in particular, the RISC-V abstractions.

+Mathematical Proven Correctness
+Trusted Foundations by Offering Connections to Authorative RISC-V Semantics

Key Properties of Flows

- ***Complementary***: The flows are strictly complementary and avoid redundancy by specializing on specific requirements, e.g., CPU code generation in the universal flow, AI accelerator code generation in the Accelerated Flow, and CGRA code generation in the CGRA flow.
- ***Maximize Sharing***: If components can be shared, e.g., the PyTorch frontends, then the flows share as much as possible
- ***Reuse of Established Frameworks***: Our flows reuse existing frameworks as much as possible, either directly via LLVM or MLIR, or indirectly through our own framework xDSL.

As a result, this minimizes duplication and maintenance overhead, enables us to scale to real-world use cases, and provides a streamlined path to impact for our development.

Further Unification in the Future

Our use of two compiler frameworks, LLVM and MLIR, mirrors the current industrial practice of having a specialized AI compiler framework. There is ongoing work in the LLVM community to integrate these two frameworks better. Our work will directly benefit from such efforts.

3.3 Foundational Compilation Frameworks

The CONVOLVE tool flow relies on and interconnects with well-established open-source compiler frameworks such as LLVM, MLIR, xDSL, IREE and OpenASIP. It also supports exchange formats such as ONYX, develops novel tiling and memory optimization techniques.

3.3.1 LLVM

LLVM is a robust, widely used open-source compiler framework that has gained significant traction and is in production use in the software industry. Its strength lies in its active community, which continuously enhances the compilation pipeline with optimizations suitable for various computing contexts.

In the CONVOLVE project, we are leveraging the complementary capabilities of MLIR and LLVM. MLIR provides an advanced infrastructure for representing, transforming, and optimizing computational graphs, while LLVM handles the final compilation stage, generating efficient code across multiple architectures. This strategic combination allows us to implement project-specific optimizations and simultaneously benefit from LLVM's extensive existing optimization techniques.

The integration is further streamlined by MLIR's status as an LLVM project component, which facilitates smooth transitions between different compilation stages. LLVM's comprehensive code generation support spans numerous architectures, with particular emphasis on emerging platforms like RISC-V. This extensive compatibility ensures that the CONVOLVE project's compiler framework can effectively target diverse hardware environments, creating a versatile foundation for deploying neural network applications across different computing platforms.

Key Limitations:

- LLVM can only target CPU and GPU
- No tensor-level abstractions for DNNs

- Little control over ISA-level code generators

3.3.3 MLIR

Multi-Level Intermediate Representation (MLIR) is a state-of-the-art compiler framework that enables rapid intermediate representation (IR) construction based on static single-assignment (SSA) form. MLIR's innovative design provides a rich set of core abstractions and tools that facilitate the development of domain-specific languages (DSLs) and compilers.

Several research and industry efforts inform MLIR's IR design principles on various intermediate representation approaches. MLIR enables the construction of IR variants, called dialects, each encapsulating the semantics of closely related concepts and operations. By introducing a dialect-based system, MLIR enables developers to define custom operations, type systems, and optimization strategies with unprecedented flexibility. The framework provides sophisticated features, including advanced pass management, multi-threading support, comprehensive diagnostics, and flexible serialization capabilities.

MLIR's modular architecture allows compiler development teams to work independently across different abstraction levels. Neural network optimization experts can focus on high-level computational transformations, while hardware acceleration specialists simultaneously develop low-level implementation strategies. This collaborative approach creates a unified compilation environment where complex optimizations can be comprehensively analyzed and applied across multiple compilation phases, bridging the gap between theoretical computational models and practical hardware implementations.

While MLIR offers significant advantages for projects like CONVOLVE, its community-driven, open-source nature also introduces maintenance considerations and risks. The ecosystem's dynamic evolution means development directions can shift, and the maturity of subcomponents can vary. Researchers and developers must remain adaptable, continuously monitoring the framework's development and being prepared to adjust their implementation strategies to leverage its most current capabilities.

Key Limitations:

- C++ based compiler stack, makes iterating slow
- A framework built for AI code generation, but no end-to-end code generator available

3.3.5 IREE

The Intermediate Representation Execution Environment (IREE) framework[30] is a unified end-to-end compiler and runtime stack that enables flexible deployment of ML workloads to

hardware targets of varied heterogeneity and scale. Its focus on ML model portability allows targeting both resource-limited platforms and larger deployment targets. IREE follows a comprehensive approach to ML compilation by offering: i) support for a wide range of advanced ML formats (e.g., ONNX, TOSA, etc.) and their features (e.g., streaming, dynamic shapes, etc.), ii) code generation of IRs that encapsulate both scheduling, which manages aspects of host and target device communication (e.g., device placement, data dependencies), and execution logic, which encodes computation tailored to the target accelerator. By integrating with MLIR, IREE can leverage its rich abstraction capabilities across the compilation stack, enabling flexible specialization and retargeting at every level.

Key Limitations:

- End-to-End code generation support only for GPU

3.3.6 ONNX-MLIR

ONNX-MLIR is an open-source compiler that generates optimized inference code from ONNX models. Built on the MLIR infrastructure in LLVM, it uses a structured lowering pipeline: the ONNX graph is first converted to the ONNX dialect and then lowered to the Krnl dialect for loop-level optimizations. At the Krnl stage, operations like MatMul and GEMM are optimized using manual libraries with techniques such as tiling, unroll-and-jam, and vectorization. The code is then lowered to the Affine and LLVM dialects. The final LLVM dialect is compiled into a shared library, which, combined with runtime libraries and a main program, enables efficient inference across diverse platforms. (Matmul implementation is explained in the Deliverable 5.3 see section 3.2.1.3)

Key Limitations:

- Only offers frontend support

3.3.7 OpenASIP

CGRA compilation flow is based on the OpenASIP [35] toolchain, scheduling different kernel for Transport Triggered Architectures (TTA) [36]. This "unconventional" architecture implementation lets the definition of heterogenous computational units, accompanied with buffering units (Register Files), load store units for memory communication and control units for the whole core. The scheduler in this case schedules transfers among different units between buses rather than time-schedule operations, used by the conventional cores (e.g. x86, ARM and RISC-V).

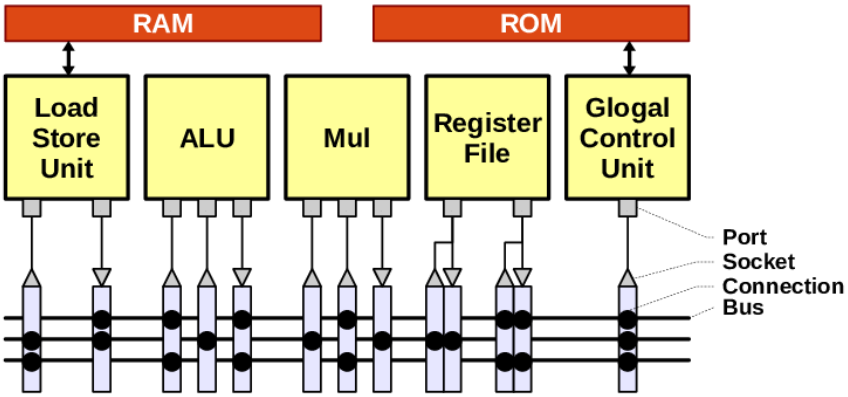


Fig. ICCSTTA: Abstract Definition of the TTA Processor

An abstract definition of the TTA processor can be seen in Figure ICCSTTA. The compilation flow for the CGRA accelerator comes from the LLVM compilation flow, following scheduling schemes coming from the Transport Triggered Architecture (TTA). The scheduling is possible by defining the kernel description in high-level C language along with custom instruction driven intrinsics, fine-tuning the dataflow and enabling optimizations like approximate operation driving and SIMD computation.

Key Limitations:

- Standalone CGRA Flow for a specific TTA hardare

3.4 Our High-Level Programming Environment xDSL

CONVOLE's WP5 offers and continues to advance xDSL [6], a high-level programming environment that facilitates the design and exploration of domain-specific compilers. xDSL is a Python-native compiler toolkit designed to be a more research and prototype-friendly companion to MLIR. It is an open-source project led by the University of Cambridge, used by several other research groups and companies, and is leveraged for several CONVOLVE projects in their NN compilation flows. It has over 450 GitHub Stars, 4600 commits, and 100 contributors.

Like MLIR, the xDSL project comprises core infrastructure for IR definitions, several target-specific utilities, and many related helpers and subprojects. It has a vibrant open-source community, mainly consisting of students using it for their research, as it is easier to get started with than MLIR.

xDSL was used to implement the RISC-V backend for both the Snitch and SNAX flows, described below. In both projects, abstractions and transformations were contributed to the repository for sharing with future projects, including those outside the scope of CONVOLVE. For example, two European startups, one doing quantum error correction (https://www.riverlane.com) and the

other low-power neural network processing (https://semron.ai), have integrated the framework into their compilers. xDSL has been added to the CPython performance benchmarking suite, used by the core team to optimize Python performance.

Since May of 2024, there have been over 2500 commits by 81 contributors, and 82 releases. New features include an x86 dialect, and ARM dialect, generic register allocation infrastructure, handling x86 with AVX512 in addition to RISC-V, Interface APIs for operation definition, a backend for emitting Cerebras Software Language (CSL), sparse dataflow analysis. We added benchmarking infrastructure to examine the compile-time impact of individual transformations of the IR, including a Python bytecode tracing package for fine-grained measurement. We added an interpreter for MLIR's declarative rewrite state machine, allowing users of MLIR and xDSL to reuse peephole rewrite patterns across framework.

Key Limitations (before CONVOLVE):

- No backend support
- Not very mature yet

3.5 Memory Allocation

Memory management is important for machine learning kernels due to the large amount of data transfers [21]. Memory management in the context of machine learning accelerators is split into two parts: (i) planning and (ii) runtime data transfer management. The planning part is further divided into bufferization, i.e., assigning memory buffers to tensors, and offset assignment, i.e., placing each of the distinct buffers to contiguous chunks of on-chip memory. "Memory allocation" refers to work on the offset assignment front, which is orthogonal to both the bufferization and data transfer parts.

Bufferization is usually conducted independently from memory allocation, but synergistic approaches exist [22-24]. The CONVOLVE compiler keeps bufferization and offset assignment separate, but its infrastructure is such that future attempts towards combined memory management remain feasible (if such attempts are viewed as an iterative process repeatedly invoking existing parts).

Buffers are annotated with lifetimes, i.e., moments of logical time between which a buffer is considered "live". Temporally overlapping buffers must remain spatially disjoint, while non-overlapping ones may share the same memory space. As long as the compiler has a concrete definition of logical time, this lifetime analysis is trivial; an indirect consequence of the scheduling and bufferization steps, given the network's static architecture. The final sequence

of lifetime-annotated buffers is the input of memory allocation. In the literature, it comes by many names: buffer allocation [25], static memory allocation [26], static memory planning [27], memory layout optimization [28], and offset assignment [29]. We will be using the terms "buffer allocation", "memory allocation", and "offset assignment" interchangeably, though the essence remains the same: each buffer receives an address space offset during that analysis, its goal being to utilize as little memory as possible.

4 Universal Compilation Flow

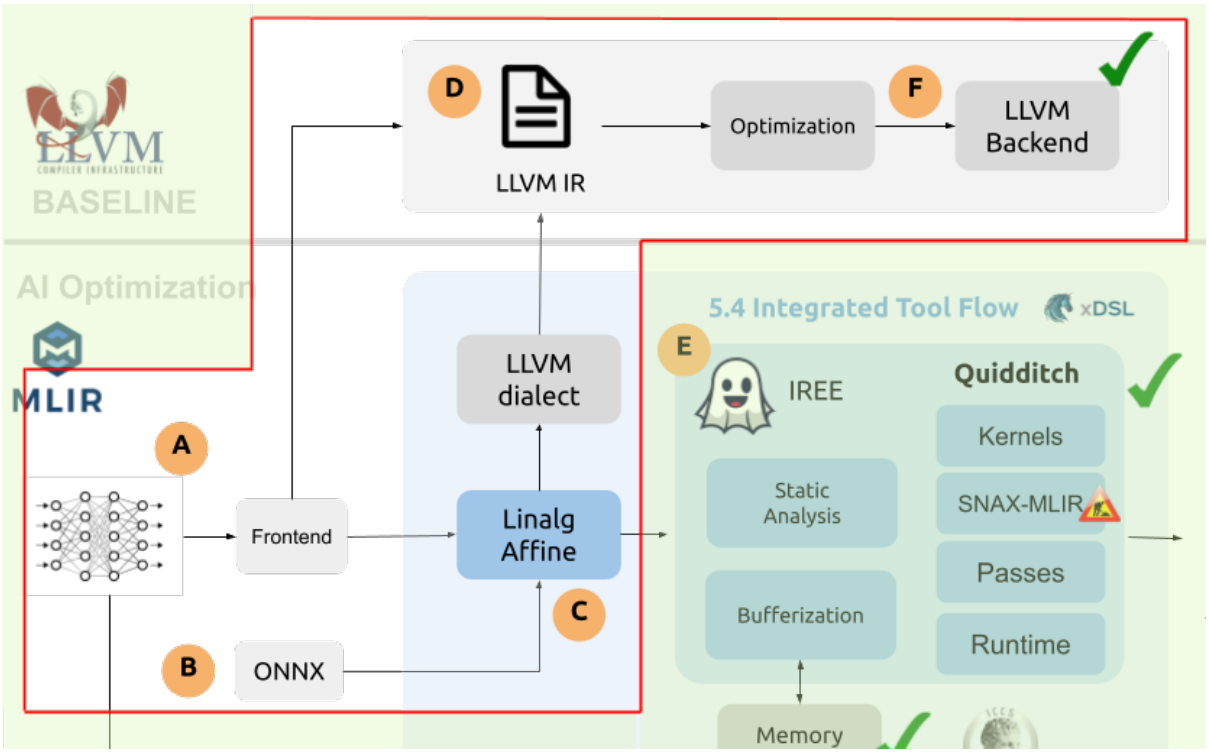


Fig. UMU1 Highlight of the Universal Flow within the Convolve compiler

The CONVOLVE compilation flow is strategically designed to optimize the mapping of deep neural networks (DNNs) onto heterogeneous chip architectures, with a core focus on leveraging the generic capabilities of RISC-V processors as a foundational benchmarking and optimization framework. At the heart of this approach are the RISC-V CPU clusters integrated both within the System-on-Chip (SoC) and on the host system, which serve as critical reference points for performance characterization and baseline measurements.

By utilizing the standardized and open RISC-V architecture, the compilation flow establishes a consistent and reproducible baseline against which the performance of specialized CONVOLVE

accelerators can be rigorously evaluated. The generic RISC-V processors provide a baseline computational model that allows for:

1. **Performance Normalization:** Establishing a standard computational baseline that enables direct comparisons between different neural network mappings and acceleration strategies.
2. **Comparative Performance Analysis:**
    a. **For baseline MLIR-LLVM pipelines:** Leveraging the flexibility of RISC-V processors to understand baseline computational characteristics (e.g., computational efficiency, etc.) and our improvements over them.
    b. **For CONVOLVE accelerators:** Using the RISC-V processors as a reference point to quantify the performance gains achieved by the specialized CONVOLVE accelerators.

The compilation flow methodically employs these RISC-V processors to generate baseline performance metrics, which then inform the subsequent optimization strategies for mapping neural networks onto the more specialized CONVOLVE accelerator architectures. This approach ensures a systematic and data-driven methodology for maximizing computational efficiency across the heterogeneous chip ecosystem.

4.1 End-to-End: RISC-V via Stand-Alone Baseline Libraries

We use an ONNX - MLIR - LLVM - RISC-V End-to-end flow, OML, for RISC-V creating stand-alone baseline libraries.

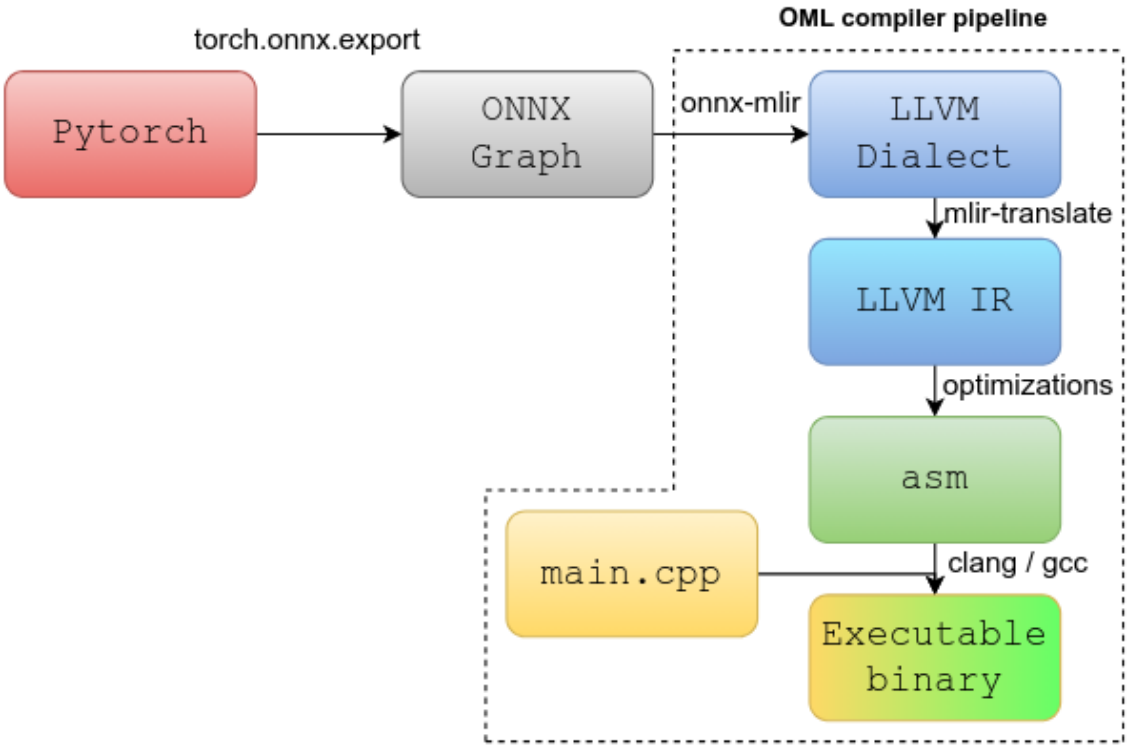


Fig. UMU2 Magnification of OML, **(A->B->C->D->F)** in the Universal Flow

OML generates executable binaries for neural networks by leveraging MLIR and LLVM IR. The process begins with exporting the neural network to an ONNX computation graph. Various frameworks, including PyTorch, TensorFlow, and Scikit-learn, provide functions to convert models into this format.

Next, the ONNX graph is translated into the LLVM dialect of MLIR through ONNX-MLIR, passing through the affine dialect. The LLVM dialect is a wrapper for LLVM IR, defining corresponding operations and types within MLIR. However, ONNX-MLIR does not directly expose LLVM IR, making it difficult to generate stand-alone executable binaries. OML addresses this limitation by lowering the LLVM dialect to obtain LLVM IR, facilitating binary generation. The resulting LLVM IR is an entry point to the LLVM compiler, enabling optimizations and conversion to assembly code through the LLVM static compiler backend. A native assembler and linker then process this assembly code to produce a fully executable binary.

Additionally, ONNX-MLIR does not natively support the RISC-V instruction set architecture. To overcome this, runtime libraries such as OMInstrument, OMRandomNormal, and OMTensor are statically compiled to create RISC-V-compatible runtime libraries. These libraries, along with the generated assembly code and inference-related components, are linked using a compiler such as Clang or GCC to produce an optimized executable for the RISC-V architecture.

The proposed approach generates statically linked binaries for neural networks, eliminating the runtime overhead associated with Python and shared libraries. Unlike ONNX-MLIR, which primarily produces shared library files and has higher memory requirements, OML allows for more efficient execution. Additionally, OML provides access to MLIR dialects and LLVM IR, enabling extensive optimizations at both high and low levels. These include vectorization, buffer management, macro-operations fusion, fine-tuned vectorization, and hardware-specific enhancements, ensuring full utilization of available hardware capabilities.

4.1.1 Optimized using default optimizations of the mainstream compilers

ONNX-MLIR for vectorizing MatMul operations. Assuming a matrix multiplication C=AB, for each element of matrix A, the corresponding row of matrix B is selected. Specifically, for element A[i][k], the k-th row of matrix B is chosen, and the value of A[i][k] is multiplied by each element in the k-th row of B. This multiplication is carried out using vectorized operations to enhance performance. **For a more detailed summary of our proposal, see deliverable 5.3 section 3.2.1**

4.1.2 Hand-compiled libraries for RISC-V to support ONNX

To support Hand-compiled libraries within the onnx-mlir toolchain, we extended onnx-mlir to explicitly lower the custom operations. At a high level, this involves first translating the EP ops into the ONNX-IR dialect, a high-level MLIR representation of ONNX ops. This step ensures that all necessary attributes and type information are preserved, allowing the ops to move correctly through the compilation pipeline. Following this, the custom ops are lowered into the internal runtime representation by allocating output buffers and constructing appropriate runtime calls that are compatible with the LLVM backend. **For a more detailed summary of our proposal, see deliverable 5.3 section 3.7.**

4.2 Custom optimizations: OML-Vect

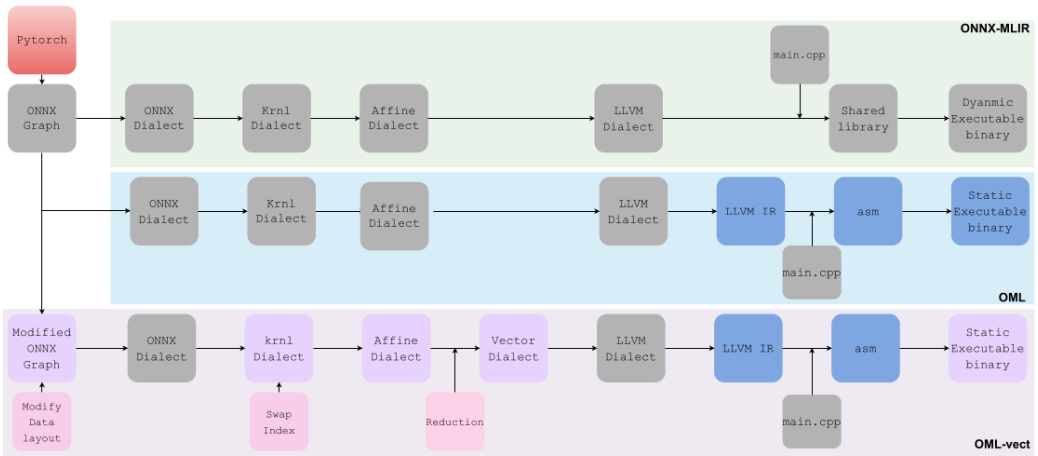


Fig. UMU3 Comparison of ONNX-MLIR, OML, and OML-vect flows

We propose a technique OML-vect [13] that addresses limitations in the MLIR framework in order to enable and enhance auto-vectorization. The pipeline begins by exporting the pre-trained neural network to ONNX graphs. The ONNX graph is then traversed, and its data layouts are modified to optimize the code for auto-vectorization. Subsequently, the graph is lowered to the ONNX dialect, followed by the krnl dialect, and finally to the affine dialect using ONNX-MLIR. At the affine dialect level, reduction patterns are identified and the "Affine loop-vectorizer" is invoked. The vectorized code is then progressively lowered to SCF, CF, and LLVM dialects, ultimately generating LLVM IR, assembly (asm), and eventually standalone binaries are generated. **For a more detailed summary of our proposal, see deliverable 5.3 section 3.2.4.** *Our work was published at the flagship Intl .Conference on Code Generation and Optimization (CGO 2026), which is the most relevant conference for this research area (code optimization and code generation) available at doi =* https://doi.org/10.1109/CGO68049.2026.11395226. *We have made the artifact [47, 48] available at https://doi.org/10.5281/zenodo.18006548 and open sourced code at https://github.com/CAPS-UMU/onnx-mlir-caps/tree/oml-vect-cgo-2026*

4.3 Leveraging Alias analysis without porting

Alias Analysis (AA) is one of the crucial analysis passes used for compile-time optimization. It disambiguates pointers to determine whether they may refer to the same memory location. Higher precision in AA enables optimization and transformation passes to perform more effective program transformations. However, AA is computationally expensive because it often requires examining the entire program, rather than individual functions, to track how pointers propagate. When the code is transformed during compilation, previously computed AA results can become invalid and must be recomputed. As a result, AA may run multiple times within a single compilation process, and improving its efficiency can significantly reduce compilation time. Currently AA in LLVM is very conservative, about 85% of pointer pairs are conservatively reported as MayAlias. With MayAlias transformation cannot benefit as it may point to the same memory location, what is important is NoAlias/MustAlias which are precise AA results. To improve precision of AA we integrate high precision external AA called SVF[44], although such analyses is both memory-intensive and time-intensive because it must maintain large, detailed points-to sets and propagate precise pointer relationships across the program (often flow- and context-sensitive), which increases memory consumption and leads to significantly higher analysis time, especially for large codebases, and is often part of older versions of compilers. To address these problems, we rely on the insight that the compiler Intermediate Representation (IR) changes infrequently and stays consistent across different compiler versions. By leveraging this, we can encode results from external alias analyses (even those implemented on older compiler versions) and then read these analysis outcomes in newer compiler versions. We propose an AA database, which decouples SVF and LLVM, making the more recent LLVM version operate with older SVF versions. We integrated database with AAManager from which the

transformation passes can query the database without having to build the graph to disambiguate pointers multiple times, by doing this we save compilation time, memory usage and improve performance without having to port high precision external AA into the recent version of LLVM. With the help of databases we reduce the compilation time by 85.22% compared to the naive implementation of building SVF graphs at every function. The 85% reduction in analysis time is equivalent to 6.66x speed-up in compilation, merely from reusing the analysis outcome. We presented AADB[16] at CGO (Code Generation and Optimization) Student Research Competition 2026.

4.4 Custom optimizations: LLVM RISC-V Backend (CAIF)

Instruction fusion is a micro-architectural optimization that combines multiple architectural instructions (µ-op) into a single operation (Macro operation), improving performance by freeing up resources and improving performance (more work per µ-op).
We use Compiler-Assisted Instruction Fusion (CAIF)[12] to enable instruction fusion in hardware, performing selective reordering of memory operations, which requires precise dependence analysis.

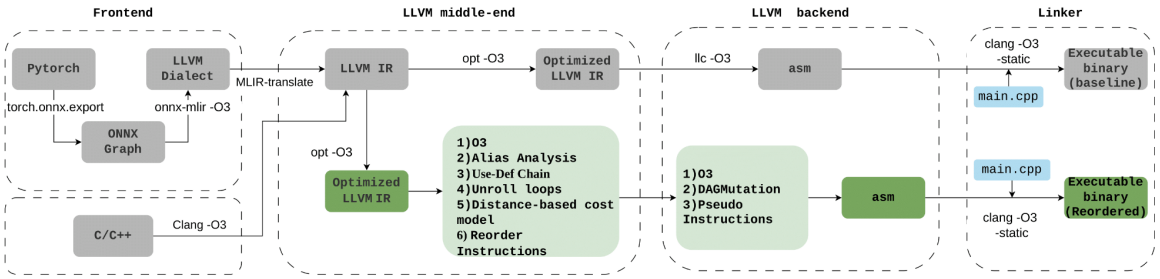


Fig. UMU4 Magnification of CAIF flow, **(A->D->F)** in the Universal Flow

Highlighted by light green boxes in figure UMU4, our CAIF static analysis pipeline combines LLVM's Scalar Evolution (SCEV), use-def chains, and adapted alias analysis to support aggressive yet sound instruction reordering, enabling hardware-level fusion opportunities. CAIF was published at the flagship International Symposium on Code Generation and Optimization (CGO'26), which is the most relevant conference for this research area (code optimization and code generation). CAIF compiler is available on github[49].
For more details on each step of our static analysis, see deliverable 5.3 section 3.5 Instruction Fusion, and specifically subsections 3.5.2.1 Validity of Reordering, 4.3 Chained Alias Analysis using a database to reduce compilation-time, and 3.5.3.1 Contribution: Extension of the DAGMutation. For more details on the baseline refer to deliverable 5.3 section 4.5.

4.5 Evaluation and Results Universal Flow

In this section we present our results and discussion on OML in section 4.5.1, OML-vect in section 4.5.2 and CAIF: LLVM RISC-V Backend Optimizations Results in section 4.5.3.

4.5.1 OML Results

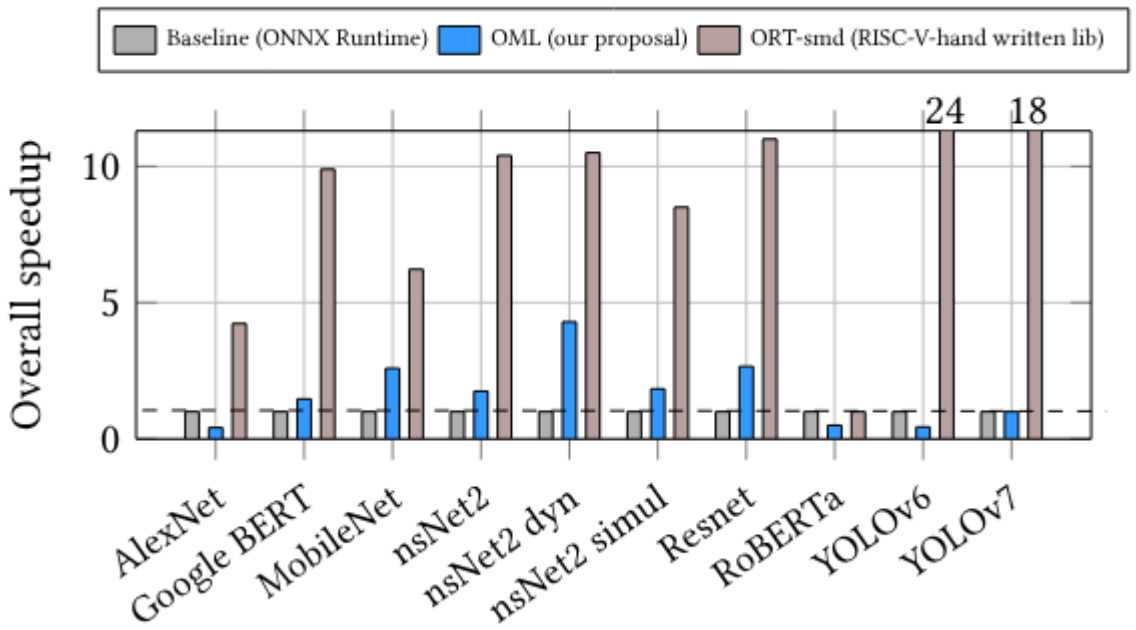


Figure UMU5 Overall speedup of NN Kernels on RISC-V comparing ORT, OML (our proposal) and ORT-smd (RISC _V hand written library)

We evaluated 10 pre-trained neural network models using datasets referenced in their respective sources. These include six basic models (MobileNet, AlexNet, YOLOv6, YOLOv7, ResNet_101, nsNet2), two dynamic versions of nsNet2, and two transformer models (BERT and RoBERTa). Model correctness and performance were assessed using multiple inference methods: PyTorch, PyTorch JIT, ONNX Runtime (ORT-default), and our OML compiler pipeline. Experiments were run on two platforms: an x86_64 Intel Xeon CPU and a Xilinx U55C FPGA emulating a RISC-V core.

For RISC-V, we compared three approaches: (1)Baseline - ORT (open-source, cross-compiled), (2) OML (our compiler pipeline), and (3) ORT-SMD (Semidynamics' optimized ORT with vector/tensor EP). PyTorch could not be tested on RISC-V due to library limitations. OML generally performs between ORT-default and the hand-optimized ORT-SMD.While ORT and ORT-SMD optimizes Convolution and GEMM/MatMul kernels, OML optimized only GEMM/MatMul, we are working on extending this support to Convolution kernels as a result models where Convolution is the main bottleneck doesnt yield expected performance benefits. ORT-SMD offers targeted optimizations (e.g., GEMM, matmul), OML benefits from MLIR and LLVM-based compile-time and hardware-specific optimizations, making it adaptable for emerging architectures without adding any compile-time overhead. **For a more detailed summary of results, see deliverable 5.3 section 4.1**

4.5.2 OML-Vect Results

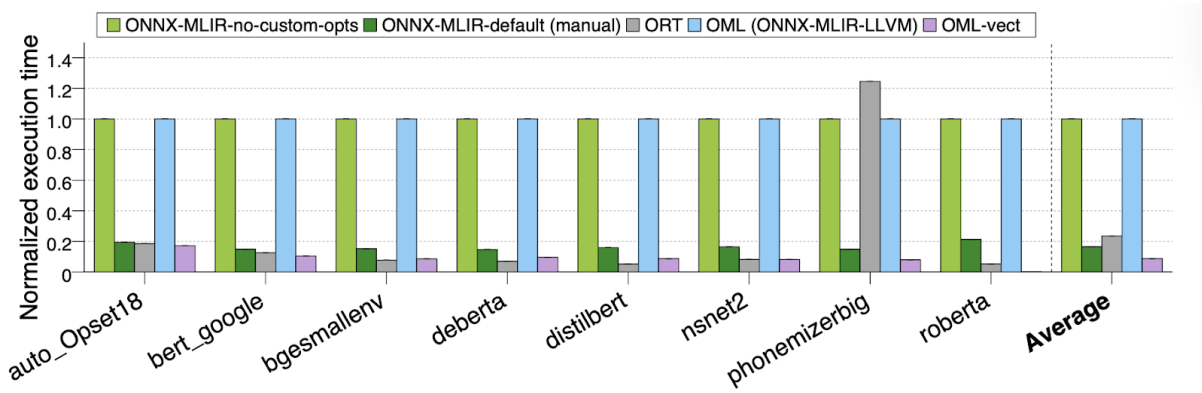


Figure UMU6 Execution time of NN Kernels on RISC-V CPU comparing various compiler pipelines discussed in section 4.2

OML-vect [13, 47, 48] outperforms ORT on RISC-V for 5 out of 8 models. On average, auto-vectorization in OML-vect (OML-reduction + data layout optimizations) achieves 92% performance improvements over the baseline. These findings highlight the necessity of combining the reduction pass with data layout optimizations to fully leverage vectorization capabilities and achieve significant performance gains. We reduce the memory footprint by 35% and for RISC-V after adding our proposed passes i.e., transpose and reduction passes, RISC-V compilation rises from 11sec to 23.1sec (≈ 40% slower). The increase stems from passes exposing additional vectorizable loops, which expand RVV code, create larger machine-IR graphs for legalization and scheduling, and prolong LLVM's instruction selection and linking resulting in optimized RVV binaries. **For a more detailed summary of results, see deliverable 5.3 section 4.3**

4.5.3 LLVM RISC-V Backend Optimizations Results

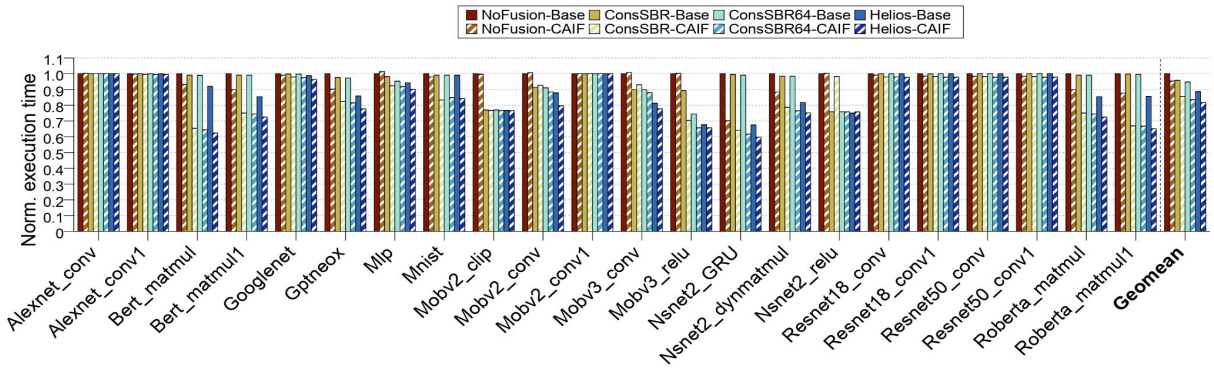


Figure UMU7 Execution time of NN Kernels targeting varying hardware complexity

Figure UMU3 summarizes the performance benefits of CAIF (Compiler Assisted Instruction Fusion) on Convolve use cases and transformers targeting varying hardware configurations.

- When hardware fusion is disabled (No-Fusion), CAIF introduces no overhead and does not increase register pressure compared to the compiler baseline.
- With a simple fusion hardware (ConsSBR), CAIF reduces average execution time by 14.4% compared to the No-Fusion baseline, significantly outperforming the standard compiler's 4.2% reduction.
- When a more advanced non-speculative fusion technique is used (ConsSBR64), CAIF further reduces execution time by approximately 16.6% compared to the No-Fusion baseline, again substantially better than the standard compiler's 5.3% reduction.

In conclusion, the results show that CAIF's instruction reordering dramatically improves the effectiveness of consecutive instruction fusion, even on hardware with simple fusion capabilities. For more details, see deliverable 5.3 section 4.5 Performance results on Instruction reordering and fusion.

5 Accelerated Flow (LinAlg + AI)

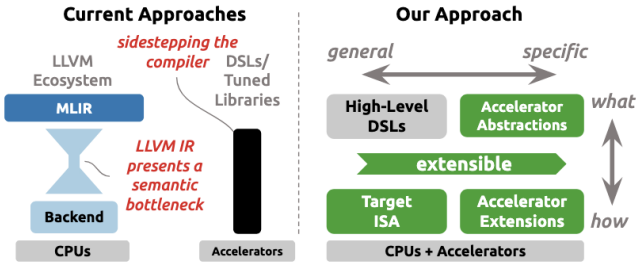
To run the CONVOLVE application use cases on ultra-low-power accelerators, we built an end-to-end compilation flow around the IREE compiler [30]. While the Universal Flow enabled breadth by generating general-purpose CPU code for all CONVOLVE applications, the Accelerated Flow targets AI accelerators.

When targeting AI accelerators, a compiler faces several additional requirements:

- *Peak-Performance:* Best-effort code generation as offered by compilers such as LLVM and provided in our universal flow is typically not enough. Instead, we aim for specialized kernels that reach peak-performance through through maximal control over the generated code (Section 5.1)
- *Tensor-Level Reasoning to Exploit Domain Information:* LLVM (and our universal flow) work with LLVM-IR as the unifying abstraction level. However, at a level of scalar loads, stores and arithmetic, much of domain information is lost. Compilers such as IREE maintain this domain information, and allow us to effectively map a DNN to micro-kernels, exploiting domain information for efficiently orchestrating compute and memory management (Section 5.2)

5.1 Efficient Micro-kernel Compilation for the Snitch PE

The end-to-end Quidditch flow leverages a micro-kernel compiler targeting Snitch [4] implemented in xDSL [6]. Our work takes as inputs an intermediate representation defined in the `linalg` dialect and lowers it to RISC-V assembly, leveraging Snitch ISA extensions. It extends MLIR's multi-level compilation flow to the backend, applying analysis and transformations at the most appropriate level of abstraction. When evaluated on a range of kernels commonly used in neural networks, our measurements reached close to peak performance. We presented this flow at CGO in March 2025.



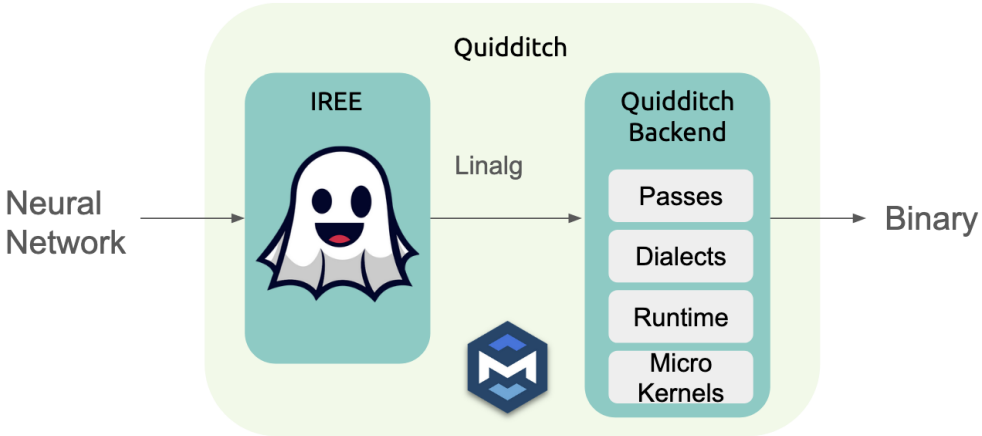
Some projects in the MLIR ecosystem share abstractions for ease of interoperability. Our compiler leverages the `linalg` dialect, a collection of IR constructs that represent linear algebra operations. Each construct in this dialect consists of input and output memory buffers, a description of the iteration space, and an algebraic description of a mapping from the inputs to the outputs. This level of abstraction corresponds to the workloads that the Snitch core is optimized for, a property that we leverage for the compilation flow.

Our progressive lowering approach allows us to implement high-performance compiler backends as a sequence of small transformations on the IR. We perform instruction selection as a collection of local rewrites going from a target-independent representation with structured control flow to a RISC-V-assembly-specific one. Keeping the structured control flow in an assembly-level representation lets us perform register allocation in a novel and simpler fashion than the state-of-the-art, which requires complex analysis over control flow graphs. This

compilation approach also makes it easier to add novel targets, as ISA extensions can be modelled as dialects, with similarly simple rewrites for progressive lowering from high-level abstractions to the low-level.

We evaluated this compiler on a range of common linear algebra operations on multiple floating-point sizes. Elementwise operations consistently reached near-peak FPU utilization in our tests, and matrix multiplication on f32 reached 70%. We performed more extensive measurements for f64 matrix multiplication, which reached above 90% for much of the exploration space. Our evaluation code is open source and available.

5.2 Mapping a DNN to MicroKernels



The rapid adoption of deep neural networks (DNNs) across a wide range of applications has led to the development of a plethora of hardware accelerators custom-built for these machine learning (ML) workloads, giving way to a self-reinforcing cycle of hardware and software innovation. This has led to a proliferation of custom hardware features that are increasingly difficult to program for and exploit with traditional compilation tools. Much of the burden of programming these accelerators falls on the shoulders of performance experts, who often resort to handwritten low-level code (e.g., micro-kernels) to evaluate their designs. While this approach is effective for evaluating individual hardware features, it does not scale to full DNNs nor does it permit the evaluation of the entire software stack. On the other hand, full-scale solutions are critical for evaluating the performance of the entire software stack, including the compiler and runtime, and for understanding the impact of the hardware on the end-to-end performance of DNNs.

Existing approaches to tackle the scalability and deployment tasks require significant development effort to either develop a tailored compiler or adapt a general-purpose ML

compiler to the new hardware. The challenge with both of these approaches stems from several key issues: i) target accelerator platforms typically employ heterogeneous architectures, often with custom architectures, ii) these are adorned with complex, multi-level memory hierarchies, and iii) have varying requirements during inference (e.g., throughput and latency constraints). To automatically generate and deploy efficient code for such platforms, advanced compiler techniques must be adopted, ranging from memory hierarchy-aware tiling, data movement optimizations and scheduling, to code generation for the custom features.

We developed **Quidditch**, an end-to-end deep learning (DL) compiler for highly-concurrent accelerators with software-managed caches. Built on top of MLIR and IREE, Quidditch is a compiler that targets the Occamy architecture and emits assembly in Snitch's extended RISC-V instruction set. We also target SNAX, a variant of the aforementioned architecture, featuring a novel low-power, fixed-operation GEMM accelerator (Section 5.4). Our compiler offers a set of novel static single assignment (SSA)-based optimizations, such as SSA-based double buffering and redundant transfer elimination by modelling explicitly managed memory. We combine these with a detailed optimization pipeline offering fusion, scheduling, buffer allocation, asynchronous scratchpad memory transfers, and memory and multi-level concurrency-guided tiling. In terms of efficient code generation for the individual compute elements, we rely on: i) a tailored micro-kernel generator for the Snitch core, and ii) a hand-written library-based approach for the GEMM accelerator. Evaluating our compiler on a set of real-world DNN models (e.g., NsNet2 used for background noise reduction), we show that it achieves a 22× speedup compared to LLVM, while offering a more scalable and maintainable solution.

Our contributions are:

- A novel end-to-end DNN compilation pipeline for highly-concurrent accelerators with software-managed caches.
- A set of novel SSA-based optimizations, such as SSA-based double buffering and redundant transfer elimination, exploiting software-managed memory.
- A thorough evaluation on a set of real-world DNN models, achieving an up to 22× speedup compared to a general-purpose state-of-the-art compiler, targeting two variants of a massively-parallel tiled hardware architecture, each offering a different set of custom accelerator features.

5.2.1 Code Generation Pipeline

The first pass during code generation is the configuration pass. The configurations pass responsibility is to inspect the kernel that should be compiled and depending on the linear algebra operation contained, the shape of the operations, and target specific information to pick and parameterize the optimization pass pipeline to use. Quidditch at the time of writing consists of just one pass pipeline specialized for matrix multiply operations which is parameterized using tile sizes required to satisfy the constraints shown in Section 3.1. After the

optimization pass pipeline, a generic lowering pass pipeline is run which lowers IR operating on tensors all the way down to the MLIR llvm dialect.

5.2.1.1 Matrix Multiply Pipeline

Before the optimization passes, the IREE front end groups neural network layers into functions called dispatches. For example, pytorch's linear layer gets grouped into an MLIR function containing two linalg operations, a matrix multiplication transpose operation followed by an elementwise addition. **For more details on dispatches, see deliverable 5.3 section 3.3.2.1 Quidditch Dispatches.** Quidditch's matrix multiplication optimization pass operates on the matrix multiplication operations within each dispatch.

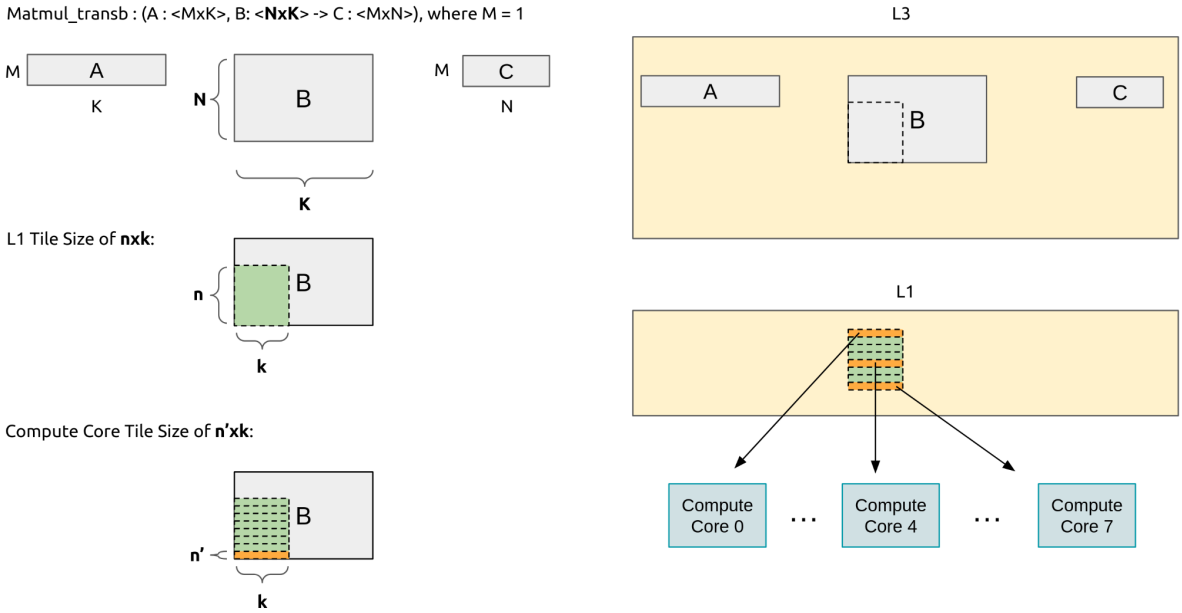


Figure UMU9. A Matmul Transpose operation tiled twice, once to fit in the L1 cache, and then once more for parallelism across 8 compute cores.

Performing matrix multiplication with full hardware utilization requires multiple levels of tiling [10]. Quidditch performs tiling to achieve two goals: 1. Reduce the working set of a linalg operation to fit into TCDM and 2. Split and distribute work across multiple CUs. The first tiling, L1-tiling (highlighted green in Figure UMU9), uses either the Myrtle Cost Model's tile selection algorithm or user-specified tile sizes to subdivide the computation such that a single tile optimally fills the cluster's TCDM L1 cache; these tile size choices reduce ping-pong between the TCDM and HBM and improve FPU utilization. The second tiling, workgroup-tiling (highlighted orange in Figure UMU9), splits and distributes work across the CUs in a single Snitch cluster, ideally matching the number of tiles to the number of CUs in the Snitch cluster (8 in the case of Occamy). This ensures even distribution of work among the CUs, with no redundant memory transfers being performed without enabling further parallelism and leaving no CU starved of work.

5.1.1.2 Lowering Pass Pipeline

The lowering pass pipeline performs the conversions from linalg operations operating on tensors down to the MLIR llvm dialect. The process starts by performing bufferization of all tensors followed by lowering higher-level operations such as scf.forall. An important step performed at this level are TCDM buffer assignments. Allocations in TCDM after bufferization are represented as memref.alloca operations with a custom attribute representing the TCDM as the memory space. During lowering, the allocations are replaced using subview operations into a global memref representing the TCDM memory. The last step before lowering to LLVM is core specialization. Up until this point of the pass pipeline, a unified function has been used for operations that may only execute on either a CU or DMA core. This pass copies the function to create versions specific for each kind of core. Where data dependencies between operations of different cores exist, barriers are inserted to synchronize the entire cluster. The current logic does this by inserting a barrier before every microkernel_fence and wait_for_dma_transfers operation.

5.2.2 TCDM Management

After hive tiling, the working set of the linalg operation is now small enough to be transferred from system memory to the TCDM. Due to the low bandwidth and high latency of system memory it is particularly important to remove any redundant memory transfers from the IR. We achieve this by introducing transfer operations (Listing 1) which operate on tensor values rather than on memory. An initial start_tensor_copy operation starts the transfer of a tensor and produces a so-called transfer tensor and a DMA token. The transfer tensor represents the unspecified state of the tensor between the start of the asynchronous DMA transfer and awaiting its result. The DMA token is a value identifying the transfer and used in a subsequent wait_for_tensor_copy operation to await completion. The wait_for_tensor_copy then yields the final tensor value in TCDM memory. It additionally takes as input the transfer tensor to preserve the use-def-chain of tensor operations and the original tensor being copied. The latter is used during bufferization to model the wait operation as reading the tensor. The bufferization framework then ensures that no writes to the underlying memory of the tensor can happen until the transfer is complete. This representation achieves multiple goals:

- As we are operating on tensors where the semantics of our operations are pure, existing compiler optimizations such as common subexpression elimination (CSE) and loop invariant code motion (LICM) can be used to remove most redundant transfers.
- Operating on tensors circumvents phase ordering problems as the majority of optimizations such as tiling or fusion occur on tensors.
- Since the transfer is modelled as an asynchronous process, transformations can insert computation in between the “start” and “wait” operation. This can be as simple as sinking the “wait” operation to more complicated transformations such as double buffering via software pipelining.

• Late bufferization allows us to recognize whether the buffer to be copied is already in TCDM by inspecting its address space during bufferization. In this case the copy can be elided entirely, lowering the start_tensor_copy operation to a noop. This functionality makes the operations closer to assertions of tensors being in TCDM, rather than actual memory transfers.

The sum of these optimization guarantees make the implementation of promoting linalg operation operands to TCDM as simple as indiscriminately transferring all of them. Later passes are then capable of cleaning up the IR until only the minimum amount of memory transfers to TCDM remains.

5.2.3 Double Buffering

Quidditch implements a novel SSA-based double buffering optimization that allows for overlapping computation and memory transfers. Our double buffering implementation builds upon Quidditch's SSA-based memory model, introducing asynchronous start_tensor_copy and wait_tensor_copy operations. These operations explicitly decouple transfer initiation from synchronization, enabling compiler-directed overlap of computation and data movement. The IR maintains explicit use-def chains through transfer tensors and DMA tokens, preserving semantics while allowing aggressive optimizations like common subexpression elimination. Buffer swaps are implemented as pointer reassignments to the TCDM address space, verified during late bufferization when physical addresses are resolved.

5.3 Cost Model Integration for Automatic Tiling

In its final form, cost analysis (Myrtle Cost Model) and automatic tiling (a Quidditch pass) occur in MLIR at the linalg dialect level, within step E of the **(A->C->E->G)** flow (see UMU8).

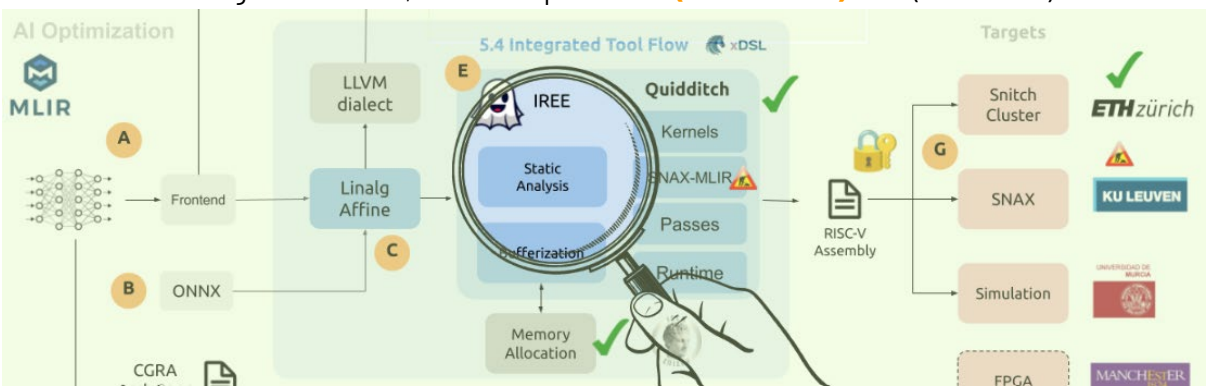


Figure UMU8. Cost Model Integration occurs within step E of the **(A->C->E->G)** flow.

5.3.2 Proof of Concept: ZigZag Integration

UMU started their deep dive into static analysis by first knitting together partner KUL's ZigZag [31] DSE tool (which also includes tiling recommendations) with the CONVOLVE compiler.

5.3.2.1 Vanilla RISC-V on the Snitch Cluster

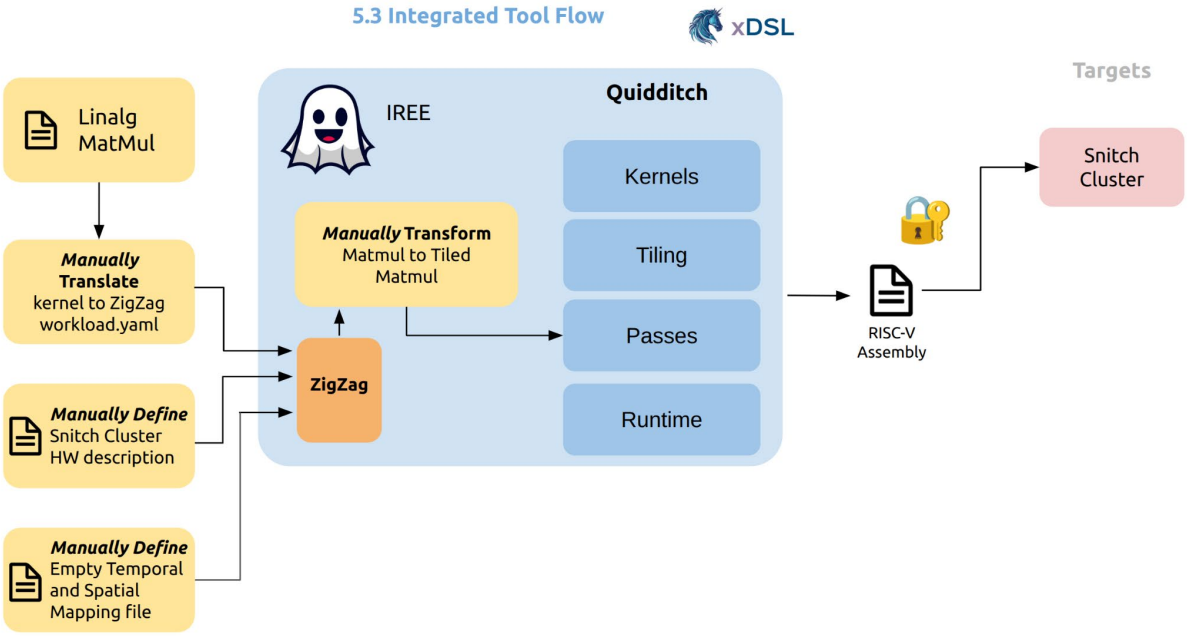


Figure UMU10. ZigZag-Tiled Matmul running on the Snitch Cluster using vanilla RISC-V

Our first integration was completely manual. We took a linalg.matmul operation, fed it to ZigZag along with a Snitch hardware description, and then manually tiled the matmul according to ZigZag's specifications. Interspersed in this tiled MLIR code, we also manually inserted the necessary operations to copy tiles from L3 to the scratchpad. This tiled MLIR ran as a test case in the Quidditch repository, but only executed vanilla RISC-V instructions on the Snitch cores.

5.3.2.2 "True" Snitch Backend: RISC-V + Snitch Extensions

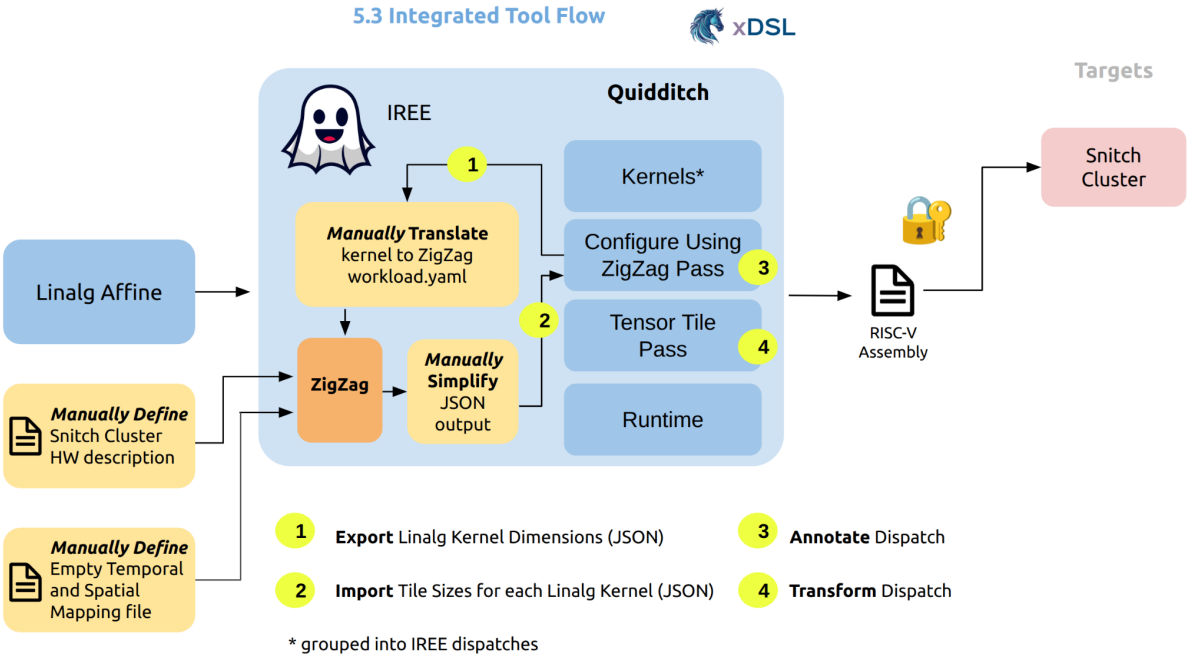


Figure UMU11. Semi-Automatic Tiling for the Snitch Cluster using ZigZag prescribed tiling schemes.

Our second integration was semi-automatic. We added an automatic tiling pass to Quidditch, ConfigureUsingZigZag, that exports an IREE dispatch description, and then imports a json file with tiling schemes for each dispatch (figure UMU10, steps 1 and 2) . Once ConfigureUsingZigZag exports a dispatch description, we manually translate the dispatch description to a workload.yaml, pass the workload to ZigZag with our snitch hardware description, and then manually translate ZigZag's results into a simpler JSON representation. Finally, ConfigureUsingZigZag imports the ZigZag recommended tiling schemes from the simplified JSON, and annotates each dispatch with tiling information (figure UMU9, step 3). Quidditch's subsequent TensorTile pass automatically performs the tiling transformation and generates the memory movement from L3 to the scratchpad (figure UMU9, step 4). This semi-automatic tiling flow compiles test cases in the Quidditch repository using its Snitch Cluster backend. Thus, UMU successfully integrated the tools developed by our partners: KUL's ZigZag and CAM's Quidditch, in a semi-automated proof-of-concept.

5.3.2.3 Pivot to Designing the Myrtle Cost Model

Through discussions with our partners and empirical experiments, we concluded that ZigZag was suboptimal for Snitch, as it was designed for a different class of accelerators; this class includes systolic-array based accelerators with multi-dimensional arrays of processing

elements instead of hardware loops. As a consequence, to avoid work duplication and to ensure the availability of automatic tiling for other kinds of accelerators, UMU pivoted to designing our own tiling cost model for the Snitch Cluster, Myrtle. In parallel, KUL developed a complementary MLIR-based compiler flow integrated with ZigZag (see deliverable 5.6 section 3.3.4 Praxis).

5.3.3 Myrtle Cost Model for Automatic Tiling

Myrtle is a tiling cost model for an 8-core Snitch Cluster, parameterized by three categories of input: application, hardware, and low-level scheduling details. We combine memory footprint calculation, streaming register configuration counts, identification of streaming vs regular register loads, and heuristics for pruning to generate a promising search space and automatically select a close to optimal tile size. Building upon a Snitch-specific tile layout, we take advantage of the regularity of the Snitch architecture to develop a highly interpretable cost model trained with Support Vector Regression (SVR).

5.3.3.1 Cost Model Design

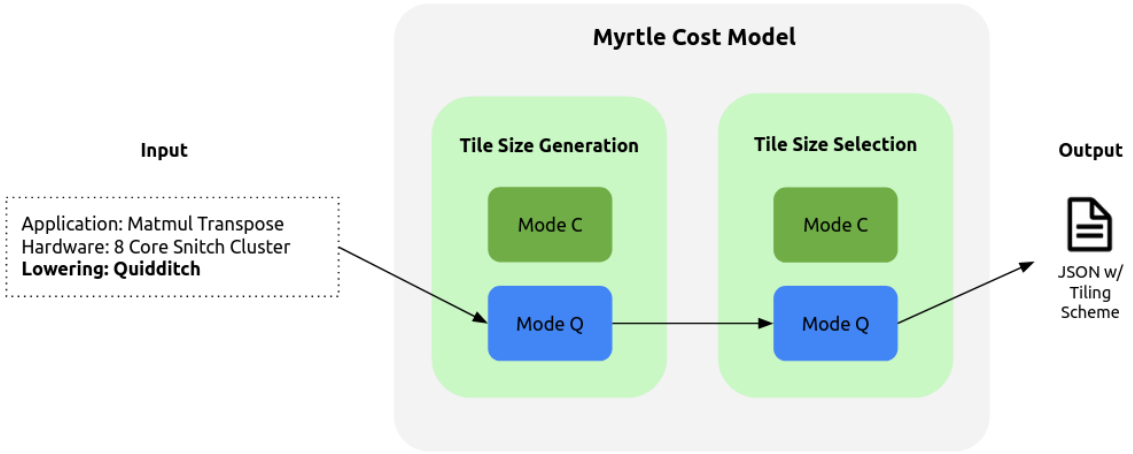


Figure UMU12. Myrtle taking Application, Hardware, and Lowering Details as Input.

Depending on the Snitch lowering method to be applied (either manual C code or an automatic Quidditch lowering), Myrtle uses a different method to pick the optimal or near-optimal tile size, denoted as scheduling modes C and Q.

Mode C: Manual C Code

In scheduling mode C, Myrtle recommends tile sizes for a manually written C code + Snitch assembly matmul lowering; this specialized lowering reduces banking conflicts [32], and allows us to experiment with a variety of matmul dimensions.

- **Tile Size Generation:** The search space is pruned to only schedules that can be run with the manual C code implementation (constraints include each matmul operand tile taking up a maximum of 8 banks in the scratchpad, a fixed unroll and jam factor of 8, double buffering, and no padding.)
- **Tile Size Selection:** Through sensitivity analysis, we observed that the L1 footprint of each compute core, the number of fmadd Instructions executed per core, the total number of L3 Loads/Stores and finally the total number of regular L1 loads were the most significant factors contributing to tiled matmul execution time. Using these factors as features, we trained an SVR offline to predict kernel execution time given a tiling scheme as input. Myrtle's Tile Selection step queries this pre-trained SVR model and selects the tiling scheme with the fastest predicted execution time.

We presented this work-in-progress scheduling mode as part of the CGO (Code Generation and Optimization) Student Research Competition 2026, earning a Silver Medal [14].

Mode Q: Quidditch

This scheduling mode takes into account Quidditch-specific double buffering and its xDSL mat-vec lowering.

- **Tile Size Generation:** Myrtle prunes its search space of tiling schemes to the most promising options. The initial search space generated is constrained to schedules that can be generated by the Quidditch backend (for example, only the B operand can be double buffered for matrix-vector operations, the unroll and jam factor is constrained to between a value of 1 and 7 etc.) Based on empirical observations, Myrtle then prunes to avoid padding wherever possible.
- **Tile Size Selection:** Myrtle assigns a ranking to each potential tiling scheme by applying a series of filters. Filter steps are applied to minimize the number of microkernel runs (also minimizes the overhead of streaming register configuration, which occurs once before each microkernel run), maximize L1 usage, and favor fast, streaming loads from the scratchpad.

For more details on sensitivity analysis and our empirical observations, see deliverable 5.3 3.3.5.3 Sensitivity Analysis.

5.3.4 Myrtle Integration with Quidditch

Paired with a modular tiling pass in Quidditch, Myrtle automatically tiles NN linear layers to run efficiently on the Snitch Cluster.

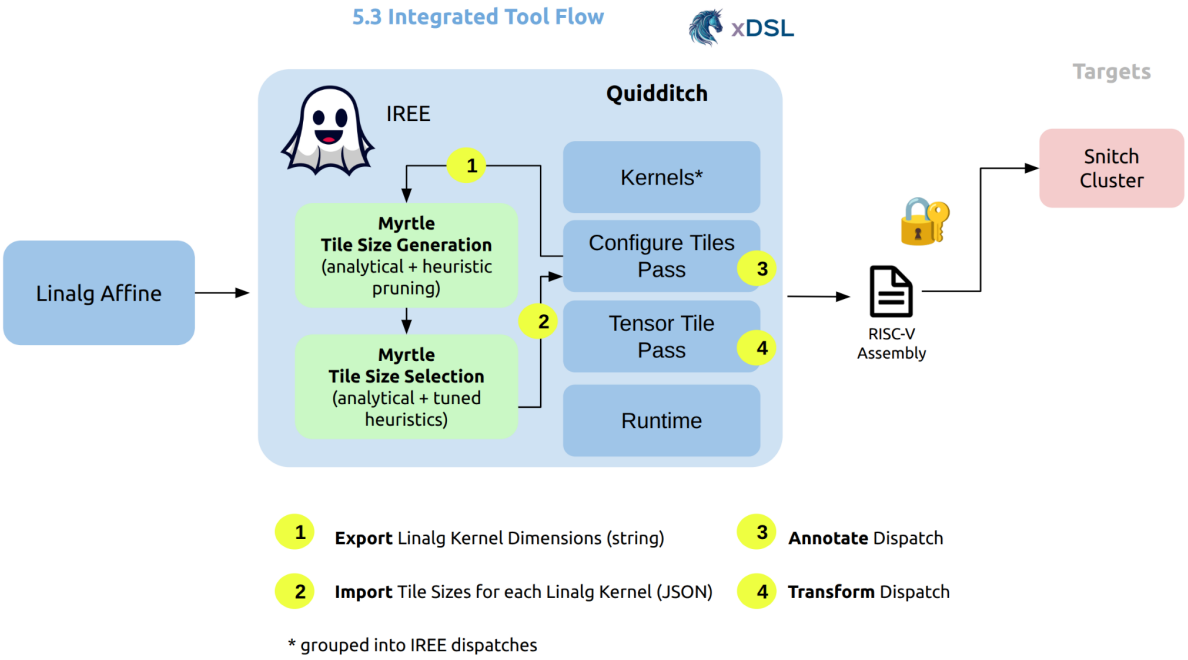


Figure UMU13. Myrtle Cost Model + Automatic Tiling Passes

Myrtle functions as an external tool written in python. It interfaces with Quidditch through the ConfigureTiles pass, which exports the name of each dispatch (whose naming convention includes the dispatch's input sizes) as a string command line argument to Myrtle (figure UMU10, step 1). The format of this input string signifies to Myrtle that Quidditch dispatches will be examined, enforcing scheduling mode Q. Myrtle then performs its analysis and returns a JSON file mapping dispatch name to tile sizes (figure UMU10, step 2). The ConfigureTiles pass uses this mapping to annotate each dispatch with a temporary MLIR attribute called LoweringConfig (figure UMU10, step 3). Later, Quidditch's TensorTile pass uses the LoweringConfig attached to each dispatch to perform the tiling transformation (figure UMU10, step 4).

Effect on Compilation Time

Integrating Myrtle's tile generation and selection steps into the Iree-Quidditch compilation stack does not introduce significant overhead; compilation times for individual dispatches as well the entire NsNet2 are observed to be less than one minute.

Open Source Availability

All source code for Myrtle is open source and available on github [45]. Code for integration with Quidditch is available as a pending pull request in the Quidditch github repository [46].

5.4 OpenGeMM/SNAX Acceleration

To enable further accelerators from CONVOLVE to be compiled using Quidditch, we developed an accelerator configuration abstraction for SNAX which will eventually be targeted by Quidditch. The key idea is that many fixed-function accelerators spend non-trivial amounts of time setting up the accelerator configuration. Our introduction of a first-class accelerator configuration abstraction enables us to (a) easily target these accelerators from Quidditch and (b) optimize the configuration directives to minimize configuration overhead at execution time. To optimize the configuration cost we established a static analysis named "the configuration roofline" (further details in deliverable 5.3, section 2.5).

5.5 Memory Allocation

Quidditch relies on IREE for memory allocation. Specifically, the operation of assigning offsets to lifetime-annotated memory buffers takes place during the `LayoutSlices` transform of the `Stream` dialect:

```
// Packs a set of statically-sized slices by greedy strip packing.
//
// This is the same algorithm used in tflite here:
// https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/simple_memory_arena.cc
// It's not fantastic and can end up with a significant amount of wastage.
// They do the packing at runtime and as such care more about performance than
// we do while doing the packing here offline. For this initial version I
// wanted to ensure we matched apples/apples their implementation.
//
// We should use a set of heuristics and pick the smallest one. There are also
// some really great papers that have approximations (as all of these are -
// 2D strip packing is NP-hard) such as
// https://www.sciencedirect.com/science/article/pii/S0925772113001016 that
// someone with a brain able to parse mathy papers can try implementing.
//
// Slice packed offset SSA values will be updated and start at the given
// |baseOffset|. Returns |baseOffset| + the total size of the allocation
// aligned to the requirements of |resourceConfig|.
static Value
packStaticSlicesGreedily(IREE::Stream::ResourcePackOp packOp, Value baseOffset,
                         MutableArrayRef<Slice> slices,
                         IREE::Stream::ResourceConfigAttr resourceConfig,
                         IndexSet &indexSet, OpBuilder &builder) {
```

Figure IDEALLOC_1: Memory allocation in IREE.

Our work on idealloc, exposed in great detail in D5.6, amounts to implementing the state of the art from those "mathy papers" [37] mentioned in the comment of Figure IDEALLOC_1. Indeed our method is superior to IREE's vanilla scheme, which relies on allocation time-ordered best fit. In this section we describe how we incorporated `idealloc` in the `LayoutSlices` transform. The matter at hand can be broken in two parts: changes at the source code level and at the build system level.

5.5.1 Source Code Changes

`idealloc` is a Rust project, while IREE is written (to the largest extent) in C++. An industry-standard tool for establishing interoperability between these two programming languages is cxx. As Figure IDEALLOC_2 illustrates, it can be used both ways. In our case, cxx is used to invoke `idealloc`, packaged as a Rust function, from `packStaticSlicesGreedily`, a C++ function (see Fig. IDEALLOC_1).

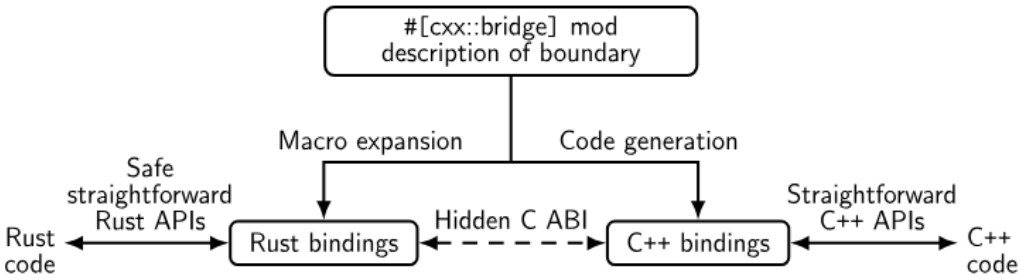


Figure IDEALLOC_2: Overview of the Rust ⟷ C++ interoperability tool used.

`packStaticSlicesGreedily` consists mostly of boilerplate code (initializations, MLIR updates). The actual kernel implementing memory allocation occupies less than 30% of its body. This kernel works as follows:

- memory buffer information, i.e., lifetimes and sizes, are read from a vector carrying elements of type `Slice`
- allocation results, i.e., the offset-buffer mapping, is stored in a vector carrying elements of type `Reservation`
- for each `Slice`, the kernel initializes a `Reservation` and runs a quadratic complexity best-fit placement over *previously processed* reservations

On the `idealloc` side, the crucial data structure is the `Job`: it represents a single memory buffer and holds its allocation and deallocation logical timestamps, its size, alignment requirements, etc. The bulk of the integration task is to translate C++ `Slice` objects to Rust `Job` objects, run the

idealloc core, and translate its output to C++ `Reservation` objects. The definitions of these three crucial types are depicted in Figure IDEALLOC_3.

```
struct Slice {
  int64_t lifetimeStart;
  int64_t lifetimeEnd;
  Value dynamicSize;
  Value packedOffset;
```

```
struct Reservation {
  const Slice *slice = nullptr;
  int64_t staticOffset = 0;
  int64_t staticSize = 0;
};
```

```
pub struct Job {
    // The bigger the type, the wider a variety of workloads is allowable.
    // This is the *ALLOCATED* size of the job! As it moves along the boxing
    // pipeline, it will sometimes be viewed as having a different height--
    // but the absolute truth is this.
    //
    // Earlier implementation had a "current" size field which had to be
    // somehow mutable--in turn, this ruined our chances for parallelism.
    pub size:           ByteSteps,
    pub birth:          ByteSteps,
    pub death:          ByteSteps,
    pub req_size:       ByteSteps,
    pub alignment:      Option<ByteSteps>,
```

Figure IDEALLOC_3: Basic data structures involved in the integration presented.

We list the main components of our integration scheme below.

1. We introduce via `cxx` the shared type `UnplacedSlice`, i.e., a struct carrying 64-bit integer values for a buffer's lifetime, size and alignment. Shared types are visible from both Rust and C++, since they are made of primitive elements.

2. We let existing boilerplate code create 64-bit integer values for buffer size and alignment, and "steal" them along with the lifetime of each `Slice` into an `UnplacedSlice` instance (see Figure IDEALLOC_4).

3. We pass a vector containing elements of type `UnplacedSlice` over to Rust via invoking the `place_slices` function. The function runs the core `idealloc` algorithm and returns a Rust vector of 64-bit integers, each standing for a buffer offset (see Figure IDEALLOC_4). For a more complete view of what happens on the Rust side, see Figure IDEALLOC_5.

4. We then create a vector containing elements of type `Reservation`, storing to each one the corresponding offset from the Rust vector spawned during the previous step (see Figure IDEALLOC_6). The rest of the code is untouched boilerplate.

To accommodate the above, the following source files need to be either modified or added to IREE's source tree:

- `idealloc` Rust source (added to IREE's root)
- `cxx` wrapper/FFI Rust source (same as above)
- C++ source and header files generated by the `cxx` wrapper[1], containing the necessary FFI definitions (added to Stream/Transforms root)
- Modified `LayoutSlices.cpp`

```
// RUST INTEGRATION INIT START
// --------------------------

// `Slice` is defined in Stream/IR/StreamOps.td
// `UnplacedSlice` is an almost-copy of `Slice`, defined in Stream/Transforms/RusToy.h
// The `trojan` vector will carry all slices over the FFI
std::vector<UnplacedSlice> trojan;

//-------------------------
// RUST INTEGRATION INIT END
int64_t offsetAlignment = resourceConfig.getMinBufferOffsetAlignment();
int64_t rangeAlignment = resourceConfig.getMinBufferRangeAlignment();

for (auto &slice : slices) {
  int64_t staticSize =
      cast<arith::ConstantIndexOp>(slice.dynamicSize.getDefiningOp()).value();
  int64_t alignedSize = IREE::Util::align(staticSize, rangeAlignment);

  // RUST INTEGRATION BODY START
  // --------------------------

  // `UnplacedSlice` is a "shared" type, i.e.,
  // usable by CPP.
  UnplacedSlice ps;
  ps.start = slice.lifetimeStart;
  ps.end = slice.lifetimeEnd;
  ps.size = alignedSize;
  ps.align = offsetAlignment;
  trojan.push_back(ps);
  // --------------------------
  // RUST INTEGRATION BODY END
}

// Runs idealloc. Offset at index i of the returned vector
// must be given to the i-th Slice.
rust::cxxbridge1::Vec<int64_t> offsets = place_slices(trojan);
```

[1] This step is rather straightforward via the cxxbridge-cmd package.

Figure IDEALLOC_4: Preparation for crossing the C++/Rust FFI boundary, and `idealloc` invocation.

For the above additions/modifications to take effect, some minimal changes must be made to IREE's build system. The next section sheds light on this final aspect of integration.

5.5.2 Build System Changes

IREE uses the Bazel build system, which in turn invokes CMake. To a large extent, the compilation of each IREE module is orchestrated by a module-local `CMakeLists.txt` file[2].

```rust
/// Gatekeeper to `idealloc`.
fn place_slices(data: &CxxVector<UnplacedSlice>) -> Vec<i64> {
    // Offsets will be written here.
    let mut res = vec![0; data.len()];

    let mut idealloc_inp: Vec<Job> = vec![];
    for (id, s) in data
        .iter()
        .enumerate() {
        idealloc_inp.push(
            Job {
                size:              s.size as ByteSteps,
                birth:             s.start as ByteSteps,
                // `idealloc` core has exclusive lifetime semantics.
                // IREE has inclusive. To convert and keep aliasing
                // intact, we add one to the buffer's deallocation time.
                death:             s.end as ByteSteps + 1,
                req_size:          s.size as ByteSteps,
                alignment:         Some(s.align as ByteSteps),
                contents:          None,
                originals_boxed:   0,
                id:                id as u32,
            }
        );
    }

    // In theory, we're ready.
    let (reg, _makespan) = coreba::algo::idealloc(coreba::jobset::init(idealloc_inp).unwrap(), 1.0, 0, 3);

    for (id, pj) in &reg {
        res[*id as usize] = pj.offset.get() as i64;
    }

    res
}
```

Figure IDEALLOC_5: The code running on the Rust side.

[2] We assume the reader to be familiar with the structure of such files.

Integrating idealloc into IREE amounts to adding the following in Dialect/Stream/Transforms/CMakeLists.txt:

- `cxxbridge`-generated source and header files under `SRCS` and `HDRS` respectively
- a new compilation target defining the static library containing the `cxx`-generated FFI
- addition of the above target under `DEPS`

This concludes our exposition. We have tested the described integration scheme both with vanilla GPU-targeting IREE, and also with the CONVOLVE-specific Quidditch backend.

5.5.3 Compilation Latency Overheads

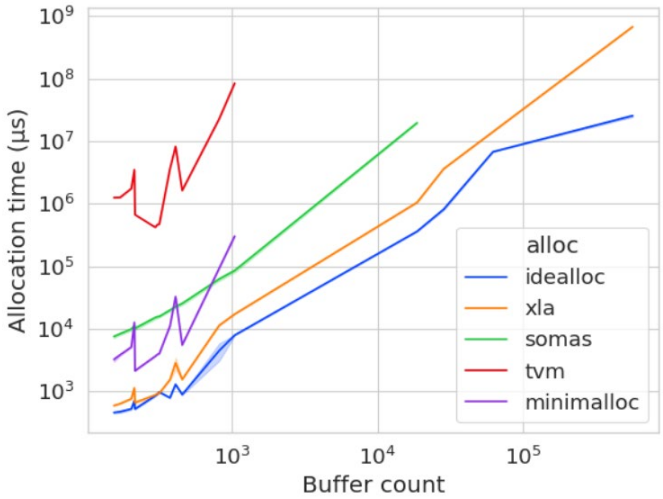


Figure IDEALLOC_6: Idealloc's per iteration latency impact compared to SoA allocators

Latency wise, figure **IDEALLOC_6** showcases the allocation time overheads of different allocators as well as the implementation of idealloc with increasing buffer sizes, when we configure **a single iteration run** on different benchmarks mentioned also above and more analytical in the Deliverable 5.5. We define the total compilation time of a workload in our case as the **vanilla compilation time (how the program should compile without custom allocation) + allocation time overhead of each allocator** as presented in the figure.
In our comparison the vanilla compilation time of all the different SoA allocator evaluation **is kept the same per allocator change**, so only the allocation time increases the total time in each case. Taking this into consideration we can see that there is a clear linear relation between buffer count and allocation times on the idealloc, xla and somas allocators, with tvm and minimalloc showing fluctuations and finally failing on buffer count larger than 10^3. We can also observe that the idealloc is constantly more efficient than the SoA comparison, **with the margin growing**

bigger as we achieve bigger buffer counts, showing our sophisticated graph-based optimization techniques on the allocator.
Finally when we scale idealloc to 10 iterations on the LevelDB benchmark and 100 on the other evaluated workloads, idealloc **is constantly 2x faster than the tvm allocator**, being also **the fastest one on buffer sizes > 10^5** , indicating the robustness of the design on bigger and more complex applications.

5.6 Evaluation and Results Accelerated Flow

Complementing the results for our universal flow as reported in section 4.4, we now demonstrate performance results on our accelerated compiler flow. We first demonstrate the benefit of Quidditch when executing neural networks on accelerator hardware. We then discuss the benefit of improved tiling strategies. Finally, we show the performance improvements of our CGRA compiler.

5.6.1 Default AI Pipeline vs Quidditch for Accelerator Programming

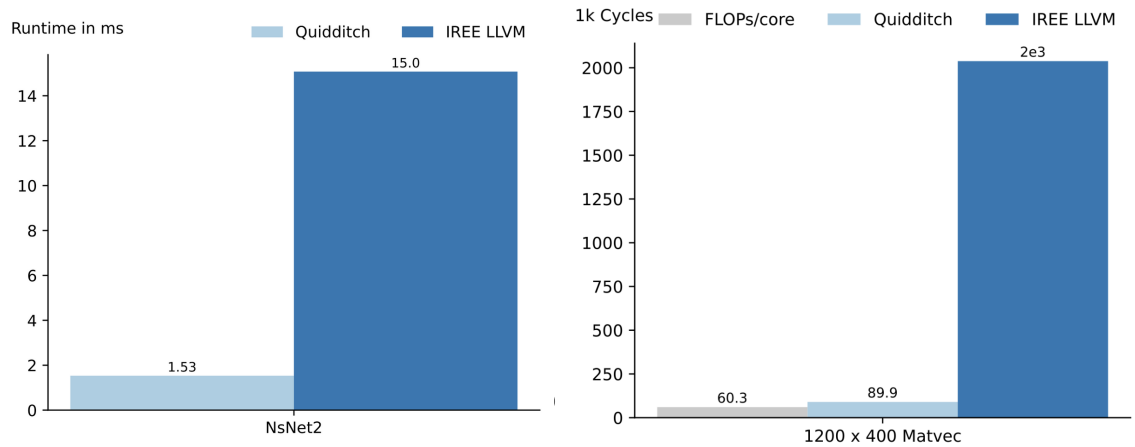
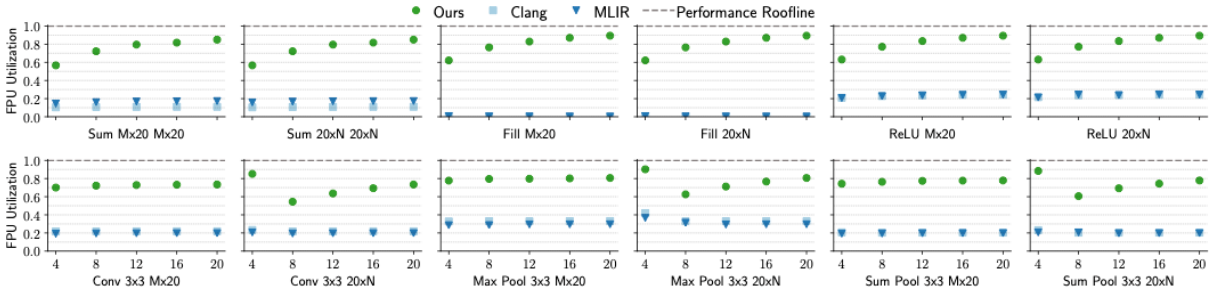


Figure ICCSQUID1: Execution time of Quidditch vs LLVM RISC-V codegen on a full DNN (left) and matrix-vector multiplication kernel (right)

The experimental setup to evaluate **Quidditch's** microkernel generation pits it against LLVM's RISC-V codegen. As a control, both setups use IREE as the main compiler, with the only difference being the selected backend. By using Quidditch to create microkernels to fully utilize the chiplet's TCDM and FPUs, we observe a near **10x improvement** in the runtime execution of the NsNet2 neural network and a **22x improvement** in a 1200 x 400 matrix-vector multiplication (Figure ICCSQUID1), demonstrating the effectiveness of our microkernel generation.



These are enabled by leveraging the RISC-V backend [4]. (Section 5.1) Our linear algebra backend generates efficient code for linear algebra operators over a range of TCDM-resident inputs.

5.6.2 Myrtle Tiling Results

We present results for both scheduling methods, mode C for the manually coded matmul kernels, and mode Q for the Quidditch dispatches automatically tiled within full runs of NsNet2.

Mode C: Manual C Code

We trained our Support Vector Regression (SVR) model offline on a Square Matrix Multiplication of size 256x256x256, using the entire search space (pruned for validity) of 42 data points to predict execution time. In figure UMU14, we test this scheduling mode on a larger square matmul of size 512x512x512.

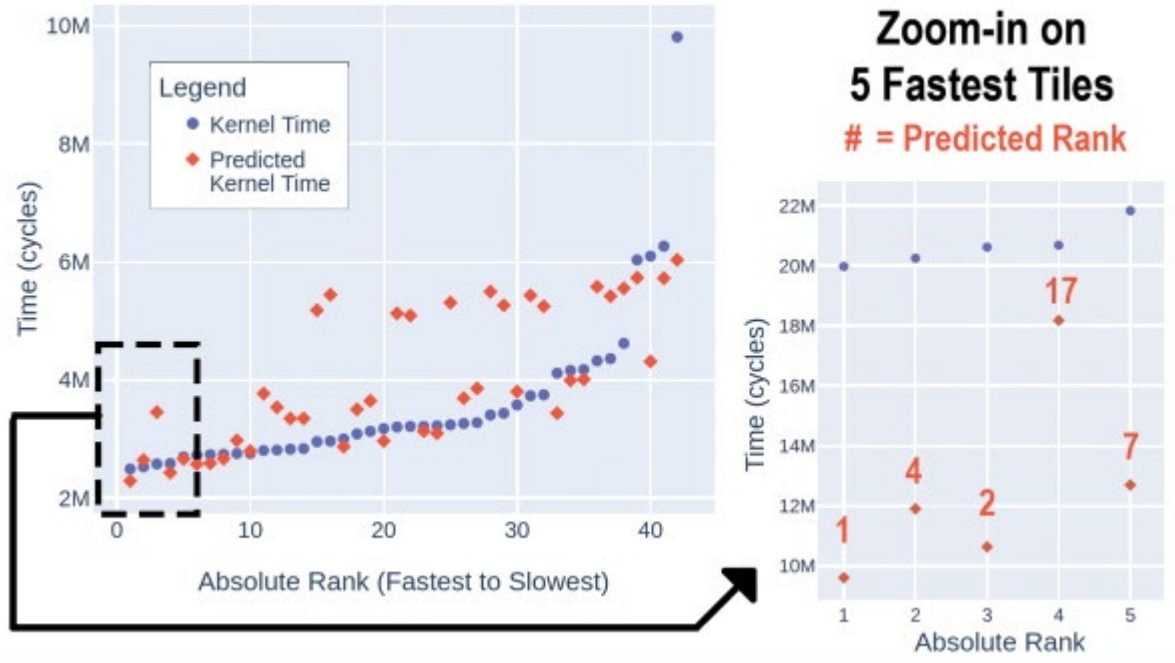


Figure UMU14. SVR used in mode C selects the best tile for 512x512x512 Matmul.

While execution time accuracy varies, ranking (kernel execution time in relation to all other points in the search space; a rank of 1 signifies the fastest observed tiling scheme) is fairly accurate, resulting in the best tile selected.

Mode Q: Quidditch

For four linear layers of NsNet2, Myrtle picks the optimal or near optimal tiling scheme.

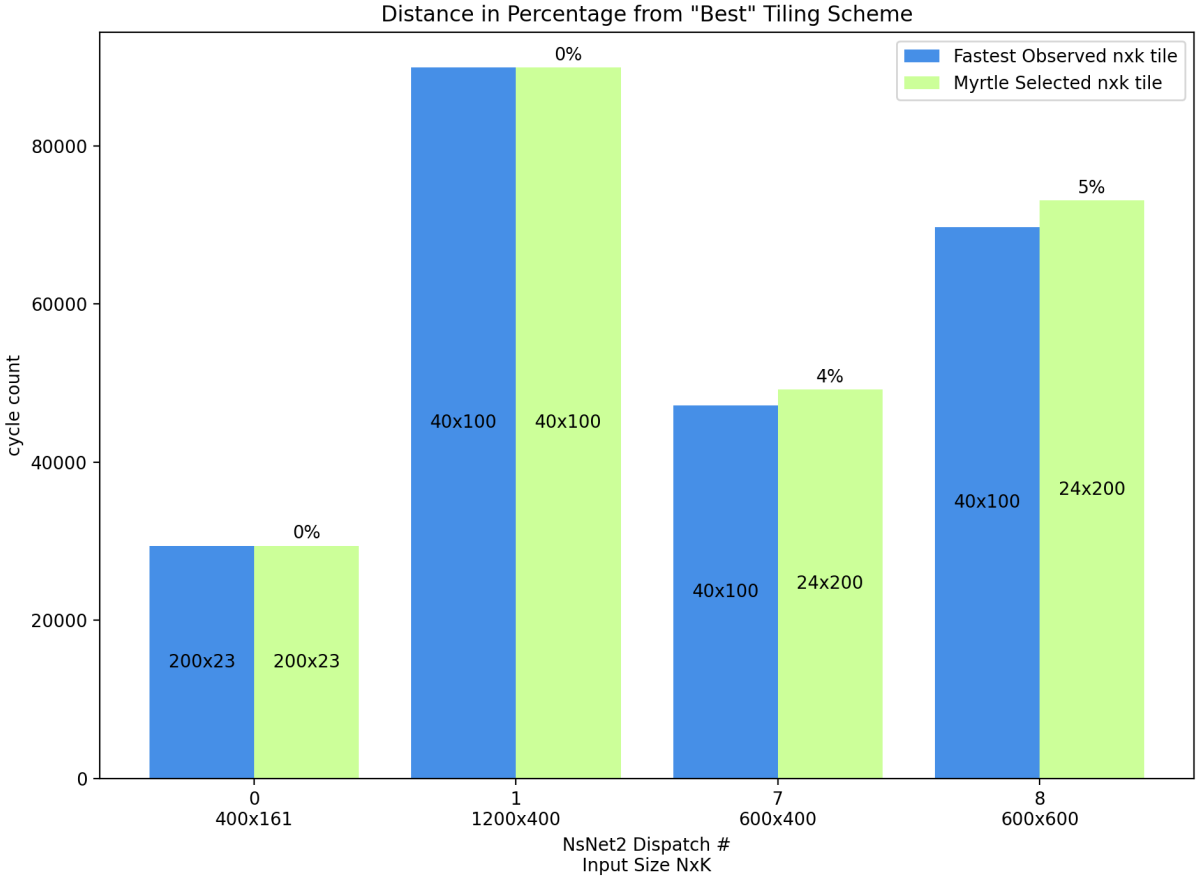


Figure UMU15. For four Quidditch dispatches from NsNet2, we compare the fastest observed tiling scheme with the tiling scheme automatically selected by Myrtle (mode Q).

In figure UMU11, we plot the execution time in cycles of each linear layer tiled two ways: with the fastest observed tiling scheme (blue) and with the automatically selected tiling scheme from Myrtle (green). In each case, Myrtle automatically selects a tiling scheme within 5% of the fastest observed tile.

5.6.3 Memory Allocation Optimization

The performance of **idealloc** was evaluated offline against state-of-the-art allocators by isolating each compiler's allocation code and running it on open-source TPU benchmarks from

the minimalloc repository. Memory-footprint results showed that idealloc remained within **20–40%** of the benchmarks' theoretical lower bound, with only **minimalloc** [38] consistently achieving optimal values, while MindSpore [39] and TVM hillclimb [40] performed comparably to idealloc. Across all 11 benchmarks, minimalloc outperformed idealloc in only 7 cases, motivating a deeper investigation of execution time. Evaluation also discovered that minimalloc suffered from **significant scalability issues**, requiring manual configuration to converge and taking over 80 minutes for a 5K allocation attempt without tuning, whereas idealloc did not exhibit these problems. Although the comparison is somewhat unfair due to idealloc's stochastic design, the results, along with planned improvements using **Monte Carlo Tree Search** [41], highlight promising potential for further advancing idealloc's performance. Extended evaluation of the idealloc optimization can be found in **CONVOLVE's deliverable 5.5**.

6 CGRA Compilation Flow

As mentioned in Section 3, the CGRA compilation flow takes advantage of the TTA compilation flow using the OpenASIP toolchain, primarily by making an "analogue" of this architecture paradigm of the data transferring mechanism that this architecture uses into the CGRA interconnection and instruction mapping. In this subsection this compilation flow will be analyzed and then an extension of this mapping mechanism will be presented that, based on a user's tolerance for accuracy drop, expressed by a QoR threshold α, will map certain NN kernel to either approximate or accurate multiplication components, providing a trade-off between accuracy and the accelerator's power and area constraints.

The CGRA flow has been designed independently of the Universal and Accelerated flow. Historically, much of this flow existed outside of LLVM and MLIR, such that the initial prototypes did not lead to a natural integration with LLVM and MLIR. Given that no tight coupling was needed between the CGRA flow and the Universal and Accelerated flow, we decided to maintain independence, and use the CGRA flow to demonstrate flexibility.

6.1 CGRA Scheduling Flow

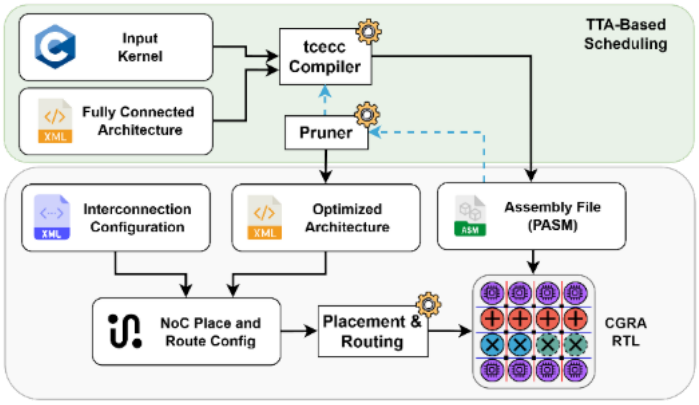


Figure ICCSSched: Proposed CGRA Compilation and Scheduling Flow

Figure ICCSSched presents an overview of the AxCGRA compilation flow. Applications targeting our CGRA architecture begin as high-level C programs and are compiled using the OpenASIP [35] TCE toolchain (tcecc), which models the CGRA as a Transport-Triggered Architecture (TTA). In this model, scheduling is expressed as timed data and control movements between functional units. The compiler produces a Parallel Assembly (PASM) file—an independent yet cycle-synchronized instruction stream for each functional unit, with one instruction issued per clock cycle. PASM is subsequently translated into a binary image that populates the CGRA's Instruction Memory (IM), from which each tile fetches and decodes its instructions during execution.

Initially, tcecc operates on a fully connected virtual representation of the CGRA, allowing scheduling without considering physical connectivity. This abstraction is gradually refined through a pruning stage that aligns it with AxCGRA's actual 2D-mesh Network-on-Chip (NoC) topology. The Pruner restructures data and control transfers, removing unnecessary links while maintaining full functional reachability.

In the final stage, the refined virtual architecture and scheduled PASM are mapped onto the physical 2D-mesh. Functional units are assigned to specific tiles, and routing paths between switchboxes are generated for both instruction and data traffic. This NoC configuration, together with the compiled program, constitutes the final synthesizable AxCGRA design, which is subsequently validated through RTL-level and post-synthesis simulations.

6.2 Approximation-Aware Scheduling

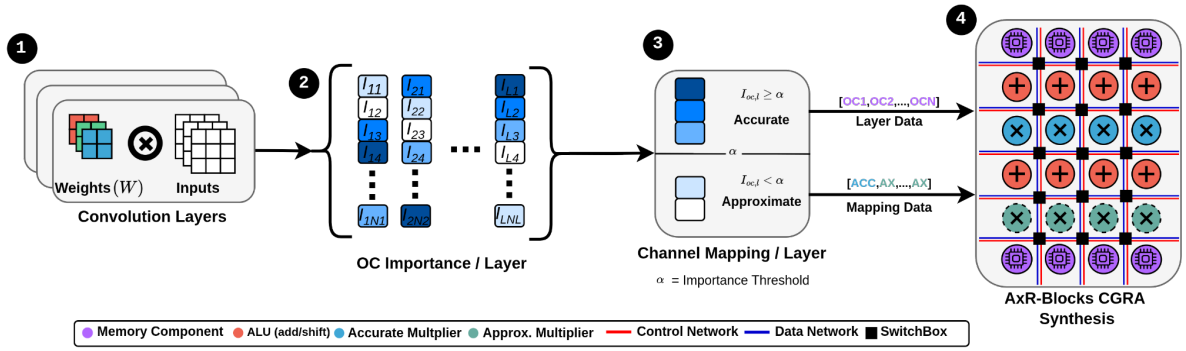


Figure ICCSAWS: CNN Approximate Aware Scheduling

Figure ICCSAWS illustrates our mapping framework, which provides an end-to-end methodology for translating high-level PyTorch convolution kernels into a synthesized CGRA accelerator. The key decision variable is the *Importance* of each Output Channel (OC), which determines whether it is executed on accurate or approximate hardware, while satisfying a user-defined QoR constraint *a*.

At the high level, convolution layers from the target CNN model are extracted using PyTorch ①. For each selected layer, we compute the Importance of every OC ②. We unroll the OC dimension to match the accelerator's architectural characteristics. Our Importance metric—adapted from Molchanov et al. [42], who introduced it for pruning—quantifies the sensitivity of each OC by comparing the original model output with the output when only a single channel's multiplications are approximated. The Importance is defined as the Mean Squared Error between these outputs, computed using a modified Brevitas quantization backend that allows selective approximation.

After computing Importance values, OCs within each layer are sorted, and those above the QoR threshold *a* are assigned to accurate multipliers, while the rest are mapped to approximate units ③. To maximize ILP utilization, we map channels in accurate-approximate pairs whenever possible, ensuring balanced resource use even when the split between accurate and approximate channels is uneven.

The collected layer data and mapping annotations are then integrated into the CGRA bitstream ④. Since the CGRA grid is already synthesized, mapping only reconfigures the NoC and updates the instruction memory, enabling streamlined deployment without requiring hardware regeneration.

It is finally worth mentioning that latency-wise, the Importance calculation (①) takes around 2 days on our server infrastructure (Intel Xeon GOLD 6530 CPU with 256GB of RAM) for per layer

characterization per output chanel per layer in the MobileNetV2 network, but this analysis is performed **once per model**. The other parts of our flow (②, ③, ④) are heavily dependent of the CGRA architecture as well as the QoR constrain, but the latency is not substantial, **taking <1 minute in the average case**.

6.3 CGRA Scheduling Optimizations and Results

Vectorization Impact

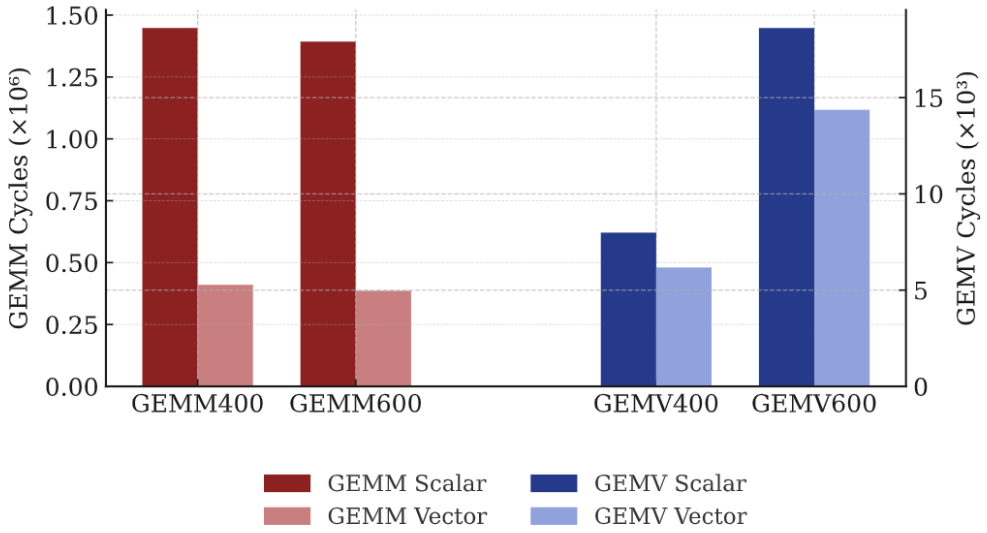


Figure ICCSVEC1: Execution Time in Vectorize VS Scalar CGRA Configurations

Figure ICCSVEC1 shows the vectorization impact on the CGRA. 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. 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 3.6x** compared to the unoptimized version on the GEMM benchmark, which drops to 1.3x. Powerwise, architectures with vector-based optimization can be up to 33% more energy efficient than scalar-based.

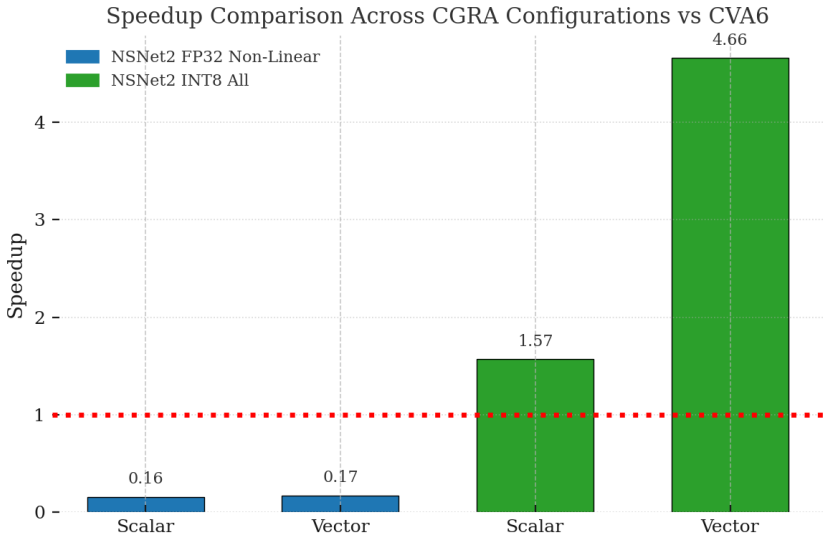


Figure ICCSVEC2: Vectorization impact on NSNet2 Model Inference

Providing an end-to-end execution paradigm, we provide four execution paradigms on the CGRA accelerator, vector and scalar in an fully quantized model (presented in green) and in a model with FP32 type non-linear operands (presented in blue), against a baseline CVA6 [43] baseline RISC-V core. In the vectorization side, we are using **8-lane arithmetic vector units in our architecture**. CVA6 performance marked with the dotted red line, marking the normalization metric for our evaluation.

Missing proper division support and not scaling great, the FP32 execution suffers from stalls due to both fully quantization-dequantization of the kernels, along with the improper division unit, making it perform way worse than the base CVA6 core.

When the model transformed in a fully INT8 implementation, with re-scaling instead of quantization/dequantization of units, and a LUT-based non-linear kernel inference, then the impact of vectorization becomes clear, providing results **up to 4.66x in performance** against our baseline. The bottleneck in this case would be the memory fetches, which will be further explored and analyzed in future implementations.

Approximation-Aware Scheduling Impact

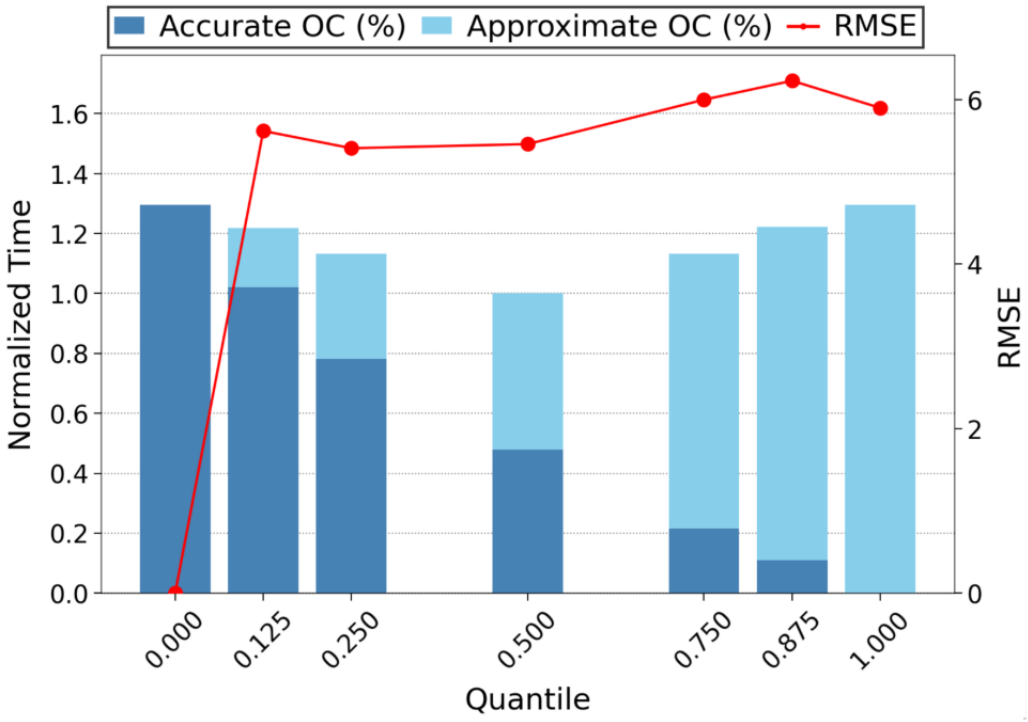


Figure ICCSPPE: Performance-Error Analysis on MobileNetV2

To also provide an end-to-end execution evaluation on the approximate-aware scheduling optimizations, we evaluated each convolution layer of the MobileNetV2 CNN, taking different threshold values (α), choosing the values as the quantiles 0.125, 0.25, 0.75, 0.875. As expected, when the approximate-accurate layer mapping is evenly balanced (different blue shades on the figure), the performance reports the lowest value. Additionally, the overall error reports a relatively small variation, with also the 0.5 quantile reporting one of the lowest values. We can also observe that there is a near-linear correlation in the error and the actual approximated output channels of the given CNN.

7 Semantics for Security and Search

CONVOLVE builds shared compiler abstractions for machine learning and high-performance computing. In doing so, it both reuses existing compiler technology, such as LLVM, and also builds new SSA-based compiler IRs to represent high-level concepts in compilers for machine learning. This uniformity in representation provides us the ability to not only effectively build

compiler abstractions, but also to *reason* about them. Toward this, we have built lean-mlir, a framework for reasoning about MLIR compiler abstractions in an interactive theorem prover. We validate lean-mlir by instantiating it with compiler IRs that model LLVM's arithmetic semantics, high-level semantics for quotients of polynomial rings, which are key for fully homomorphic encryption, as well as IRs that capture high-level control flow constructs (if condition and while loops) versus an unstructured sea of basic blocks. These formal semantics form a bedrock for multiple abilities: (1) formally reason about program semantics, which we leverage to reason about RISC-V programs in Lean, (2) automatically decide the equivalence of programs, which can be leveraged for automatic peephole rewrite generation and search space exploration.

7.1 A Formal Model of SSA via Lean-MLIR

The lean-mlir framework introduces a formal, machine-verified model of MLIR's core data structures, focusing on its Static Single Assignment (SSA) representation. Using the Lean theorem prover, it rigorously defines MLIR's SSA-based intermediate representation (IR), including core components like blocks, operations, and values. This formalization enables precise verification of compiler transformations and optimizations, ensuring correctness-by-construction for MLIR's infrastructure.

Toward this goal, the framework formalizes a core calculus for SSA-based IRs that is generic over the IR and covers so-called regions. The mechanization in the Lean proof assistant provides a user-friendly frontend for translating MLIR syntax into our calculus. We provide scaffolding for defining and verifying peephole rewrites, offering tactics to eliminate the abstraction overhead of our SSA calculus. We prove correctness theorems about peephole rewriting, as well as two classical program transformations. To evaluate our framework, we consider three use cases from the MLIR ecosystem that cover different levels of abstractions: (1) bitvector rewrites from LLVM, (2) structured control flow, and (3) fully homomorphic encryption. We show that our framework allows us to successfully model these concepts. Our work has been published at ITP'24, as "Verifying Peephole Rewriting in SSA Compiler IRs" (https://doi.org/10.4230/LIPIcs.ITP.2024.9).

Furthermore, to enable effective bitvector rewrite reasoning, we also helped to implement Lean's inbuilt fixed-width automated bitvector reasoning tactic (bv_decide). This tactic is quite effective in practice,and can solve 96.4% of SAT instances that the best solver, Bitwuzla, is capable of solving. Similarly, we can solve 88% of the UNSAT instances that Bitwuzla is capable of solving. This work has been published at OOPSLA'25, as "Interactive Bitvector Reasoning using Verified Bitblasting", and builds the foundation for bitvector reasoning in Lean.

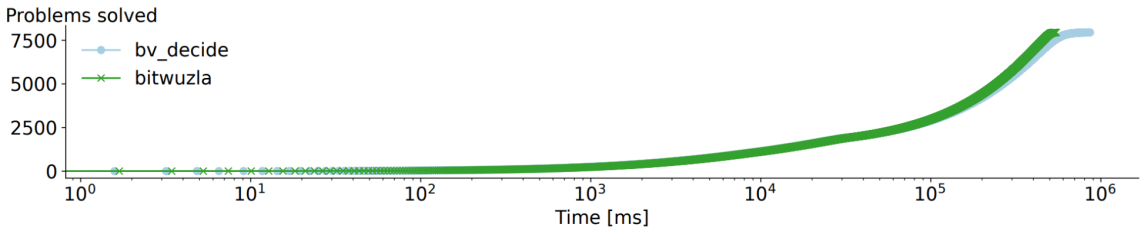


Figure BvDecide: #Problems Solved v/s Time Elapsed on LLVM Peephole Rewrite Goals

In Figure BvDecide, we see that when verifying LLVM's InstCombine peephole optimizer, 7948 theorems are solved by both our verified solver (bv_decide), as well as by the state of the art unverified solver, Bitwuzla and only few theorems trigger timeouts for bv_decide (17) or even both tools (13). This is a major milestone, as we can formally verify parts of LLVM with a previously unavailable degree of trust and automation.

7.2 First-class Semantic Dialects For MLIR

In the previous section, we have explored heavyweight formal models for MLIR using the Lean proof assistant. Here, we introduce our work on *lightweight* formal methods for the MLIR ecosystem. We provide a lightweight, tightly integrated way for MLIR developers to define their semantics. Having an MLIR community-driven semantic definition that lives inside the main MLIR project is a key step in bringing formal semantics into the everyday workflow of this community. This work of lightweight formal methods complements heavyweight formal methods: The lightweight methods provide day-to-day formal guarantees that cover common cases, while heavyweight formal methods prove end-to-end correctness properties that are out of reach of lightweight methods.

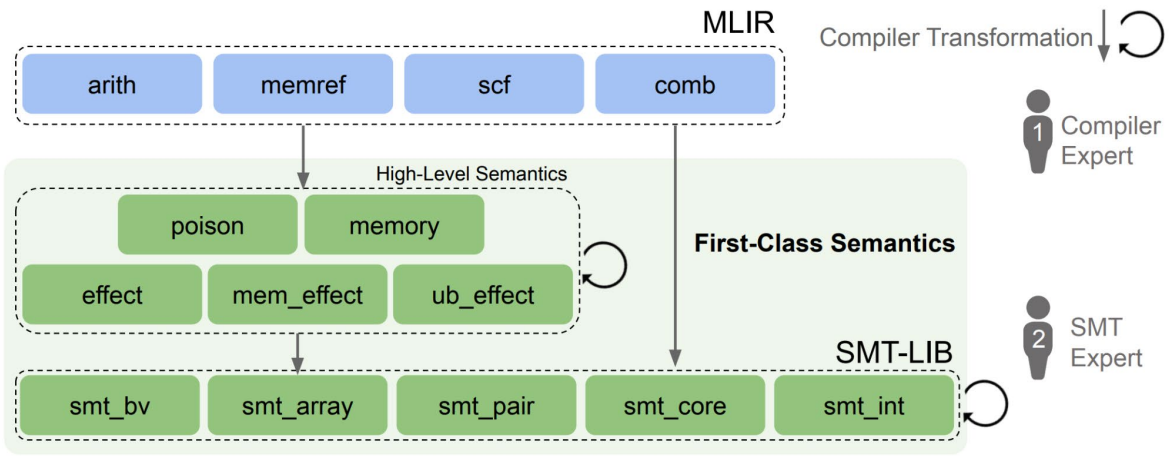


Figure MLIRSMT: We allow compiler developers to define high-level semantics as lowerings.

Towards the goal of creating an ecosystem for lightweight formal methods in MLIR, we define a collection of semantics-supporting dialects, based on SMT-LIB, to encode the semantics of MLIR IRs. MLIR developers can define semantics for their dialects using a progressive lowering approach from their dialects to our semantic dialects. Then, with the help of SMT experts, we can provide an efficient translation from our semantic dialects to SMT queries, represented as an MLIR translation to a set of SMT-LIB dialects we define, which recently got upstreamed to the MLIR repository. This work has been published at PLDI'25, as "First-Class Verification Dialects for MLIR", and builds the foundation for user-defined SMT encodings for MLIR.

Using these reusable semantics and compiler transformations over them, MLIR developers can then create a set of semantics-driven verification tools for MLIR. For instance, we built a translation validation tool to validate the correctness of a pass on a program, a peephole rewrite verification tool, and a dataflow analysis verification tool. We used these tools to verify the correctness of core passes in the MLIR repository, and found bugs that we then fixed upstream. We believe that having semantics as part of the MLIR project, and then building tools around these semantics decreases the average rate of bugs in this ecosystem, and overall, helps to raise the correctness standard of the MLIR ecosystem.

7.3 Formal Semantics for RISC-V in Lean

We have established the necessary infrastructure to begin formally verifying the properties of compiler IRs. The next logical step is to provide formal verification for lower level parts of the stack, such as the instruction set architecture (ISA) itself. Toward this, we have begun work to implement and formally prove a RISC-V backend in Lean-MLIR (see Figure RISCVLEAN), by defining a RISC-V MLIR dialect, whose semantics we need to specify to get an end-to-end proof that the behavior of the compiled program corresponds to the intended behavior of the source program. To achieve the highest level of trustworthiness for this part of the specification, we decided to use the official formal specification of the RISC-V instruction set architecture.

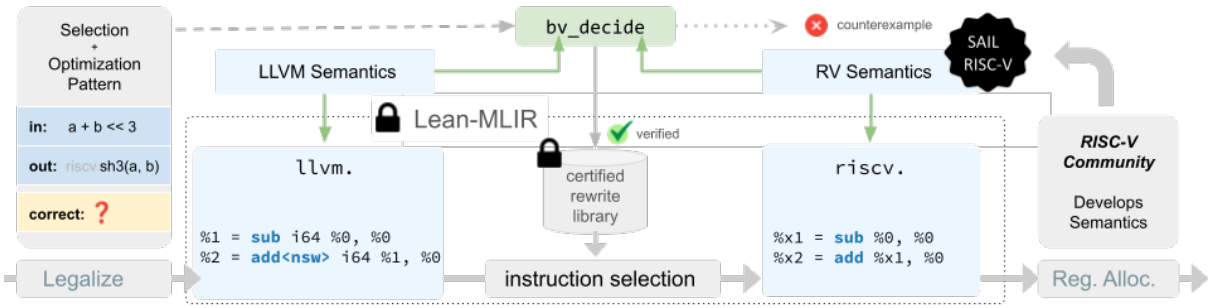


Figure RISCVLEAN: We create authoritative semantics for RISCV from the SAIL specification.

This formal specification is expressed in a specialized specification language called Sail, and expresses how to decode machine code and what the effect of executing this machine code is. As Lean-MLIR is developed using the Lean theorem prover, we undertook the effort to add a new backend to the Sail compiler. This translates the RISCV specification into Lean code. This new backend is being directly developed upstream and is part of the official Sail source code. The backend has reached a state where it is usable: it can compile the whole RISC-V specification (including all the extensions that have been formally defined in Sail) into valid Lean code, and it is being used by engineers in industry to verify zero knowledge virtual machines (zkVM). We are currently working with the Sail developers to add support for concurrency models to the generated Lean specification.

From the Sail specification, we obtain the semantics of our RISC-V dialect, which are thus validated to match the authoritative Sail semantics. We also started implementing a prototype of instruction selection from the LLVM dialect to the RISC-V dialect, following the structure of LLVM's instruction selector. Mechanizing instruction selection is a foundational step to make LLVM's backend trustworthy, as it is one of the key steps in backend code generation. With this approach, we obtain an end-to-end mechanically verified proof of semantic preservation from the LLVM intermediate representation to the authoritative semantics of the RISC-V architecture, which aligns with our long-term vision of using the RISC-V semantics to verify the Lean compiler itself.

8 Training Activities and Outreach

In 2025, we gave two workshops and tutorials to showcase the compiler infrastructure we built as part of CONVOLVE: HiPEAC on January 20th 2025, and ASPLOS on March 30th 2025.

- HiPEAC 2025 Workshop
- ASPLOS 2025 Tutorial
- MLIR Winter School
- 2025 MLIR (Un)School
- 2026 MLIR School

We also co-organized a winter and a summer school in 2025. In January 2025, Fabrice Rastello's group at INRIA reached out to us to ask if we would want to co-organize an MLIR school where they teach MLIR concepts using xDSL. Our group helped with publicity and ran the Introduction to MLIR workshop on the first day of the school, presenting tutorial notebooks we developed to introduce domain experts to APIs to construct and rewrite IRs in xDSL.

In September 2025, our group co-organised the MLIR (Un)School in Cambridge, bringing together participants from across Europe to explore developments, emerging technologies, and research trends in compiler technology. The event provided a mix of lectures, hands-on workshops, as well as networking opportunities to promote collaboration and knowledge exchange between academics, students, and industry professionals.

Our summer school in Cambridge attracted about 90 participants from Asia, Europe, South and North-America. Attendees included postgraduate students, early-career researchers, and professionals from both academia and industry.

The MLIR (Un)School met its objectives by providing an engaging platform for learning, discussion, and collaboration. The organisers received positive feedback regarding the academic content, venue, and organisation.

9 Conclusion

The second half of WP5 allowed us to advance our compilation flow in depth and breadth. To create a compiler pipeline that best served the interests of Convolve partners and targeted the rapidly developing Convolve accelerators, we **first established a robust RISC-V compiler flow capable of compiling and running every convolve use case**. The OML flow created the groundwork for this end-to-end deep learning compiler flow capable of performing automated, hardware specific optimizations. With OML-vect, we demonstrate the strength of the OML framework, extending it to support automatic vectorization that outperforms out-of-the-box MLIR optimizations. Our analysis shows that not all memory operations can be vectorized, for example in matmul only 33% memory operations were vectorised, leaving potential for memory ops fusion, and in CAIF we exploit such non-vector memory operations. The CONVOLVE micro-kernel backend compiler targeting the RISC-V based Snitch accelerator core has been a success, with a significant improvement over existing automated solutions. The progressive lowering approach made it easier to map the high-level domain-specific abstractions used in the wider MLIR ecosystem to the accelerator's capabilities. **Leveraging existing abstractions has allowed us to integrate it with IREE [30], an open-source production tool** maintained by AMD. While leveraging these production systems imposes a steep learning curve on the researchers seeking to extend it, it allows us to leverage production-ready infrastructure. This work has sparked much interest both in industry and in the wider research community. xDSL, the open-source reimplementation of MLIR in Python, designed to be accessible to researchers and domain experts has been a powerful vehicle for collaboration. We are continuing to engage with the wider community on xDSL, backend dialects, and open-source machine learning compilation infrastructure. **With the Myrtle cost model from UMU, we additionally provide Snitch-tailored, automatic tiling to the CAM's**

accelerator-specific Quidditch backend. In each of these contributions, we weighed the advantages of the robust, classical LLVM compiler and the newer, more modular MLIR framework which preserves high level information from the source code. For newer hardware targets like Snitch, we chose to dive straight into the newer MLIR framework to take advantage of its high level IR, which helps make reasoning about tiling and tiling transformations easier. For our Universal Flow, we succeeded in crafting a careful balance of MLIR and LLVM backend optimizations, blending old (i.e. robust) and new (i.e. modern) to achieve automatic, fine-grained optimizations for Convolve applications. Finally, we made significant progress towards bringing formal semantics into the foundations of the AI compiler ecosystem by introducing formal models to both RISC-V and MLIR, which serve as building blocks for **secure AI compilers [3][7][9][10]**.

Highlights of the Convolve Project – Compiler Contributions

WP5 improves the CONVOLVE compiler across multiple dimensions. In the following, we summarize our baselines and the improvements across the various dimensions.

Convolve's Objective 2 aims to **reduce design time by 10x**. To this end, we built an end-to-end compilation tool-chain that includes the Universal flow, the xDSL Python-Native MLIR-style compiler, our MLIR-based Accelerated flow, and the CGRA Flow.

In the Universal flow, we speed-up compile-time analysis by proposing an Alias Analysis Database, that enables the **reuse of costly state-of-the-art static-analyses** results across various compilation phases and even across different compilations. Storing the analysis results in a database eliminates the need to port and rerun tools and avoids frequent rebuilding of expensive analysis graphs. This reduces compilation time by 85% compared to a standard approach (baseline) where the analysis graphs would be regenerated. The 85% reduction in analysis time is equivalent to 6.66x speed-up in compilation, merely from reusing the analysis outcome.

Through our xDSL Python-Native compiler we bring state-of-the-art compiler development to the Python community. Over 450 GitHub stars, adoption by industry users such as Xanadu, Riverlane, as well as in CONVOLVE projects such as our Accelerated Flow, evidence that our xDSL Python-Native compiler is a tool developers choose. **xDSL users name accelerated compiler design as key reason for adoption**, which is both due to their pre-existing familiarity with Python and the overall fast iteration speed Python and xDSL offer. Compared to a C++-based MLIR flow as used tools like IREE, xDSL can require an **order of magnitude less hard-disk space and download-volume for installation** and **eliminates the often hour-long recompilation phase entirely**.

Our accelerated flow for the Snitch/Occamy AI accelerator has demonstrated that within the scope of just three years and the relatively modest resources of an EU project in comparison to the typical size of industrial compiler teams (10s-100s of engineers), a peak-performance producing AI-accelerator compiler can be built. In particular, we demonstrated that developing and adjusting less than 5 compiler passes is enough to build a full end-to-end accelerator flow, that enables **reuse of the existing IREE compiler code base consisting of more than 10x the compiler passes and over 100x the size of our own code development**.

Finally, our work on semantics for security and search (Sec 7) has demonstrated that local compiler rewrites can be fully-automatically certified using an integrated SMT solver. We **reduced the time to formally prove compiler rewrites from hours to seconds** by replacing manual proof development with the use of a certified SMT solver which finish proofs in seconds.

Performance: Hardware Acceleration & Beyond

Under the Universal Flow (Sec 4), the Convolve project delivered significant advances in compiler-driven optimization for RISC-V–based accelerators. Our proposed OML-vect auto-vectorization approach achieves on average **59% performance improvement over the manually optimized ONNX-MLIR libraries on RISC-V**. Unlike static, handcrafted libraries that struggle to adapt to RISC-V's evolving ecosystem, the OML-vect pipeline is fully automatic and architecture-aware, ensuring scalability and portability across diverse workloads without manual intervention.

Beyond vectorization, we proposed compiler-assisted instruction fusion for RISC-V. The CAIF compiler framework enhances neural network workload performance on RISC-V hardware supporting instruction fusion. When hardware fusion is enabled, a standard compiler(-O3) achieves only modest speedups of about 1.2% compared to running on a non-fusion baseline hardware (Sec 4.5.3). In contrast, UMU's **CAIF delivers an average 20% performance improvement over a non-fusion baseline [12]**, showing how crucial cross-layer co-optimization between compiler and hardware is.

Under the Accelerated Flow (Sec 5), our micro-kernel generator and the Quidditch NN compiler **accelerated the CONVOLVE use case NSNet by 9.8x** (from 15ms to 1.53ms), while its kernel code moved **12 NN kernels from ~20% FPU utilization to 80% and more FPU utilization** in comparison to the default LLVM compiler. The Myrtle component introduces a tiling cost model for an 8-core Snitch cluster, parameterized by application, hardware, and scheduling details. By reducing the search space through efficient pruning, Myrtle automatically identifies near-optimal tile sizes, **selecting schedules within the top 5% of performance** for matrix multiply workloads within Transformer-based applications. Moreover, the integration of UMU's cost model into CAM's Quidditch compiler demonstrates successful end-to-end collaboration, validated through the NsNet2 Convolve use case.

Demos

We will present two key advances of the WP5 in one 10' demo video consisting of two parts:

- Universal Flow: Compiling and Optimizing NNs via general-purpose LLVM/MLIR
- An End-to-end Mapping of the Full Neural Network "NSNet" to an AI Accelerator

Appendix

A1) Key Publications from CONVOLVE Members

2023

[1] CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions

Singh S, Feliu J, Acacio M E, Jimborean A, Ros A
In: 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT'23)
https:doi: 10.1109/PACT58117.2023.00009

2024

[2] Alternate Path μ-op Cache Prefetching

Singh S, Perais A, Jimborean A, Ros A
In: ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA'24),
https:doi: 10.1109/ISCA59077.2024.00092

[3] Verifying Peephole Rewriting in SSA Compiler IRs

Bhat S, Keizer A, Hughes C, Goens A, Grosser T
In 15th International Conference on Interactive Theorem Proving (ITP'24),
https://doi.org/10.4230/LIPIcs.ITP.2024.9

2025

[4] A Multi-level Compiler Backend for Accelerated Micro-kernels Targeting RISC-V ISA Extensions

Lopoukhine A, Ficarelli F, Vasiladiotis C, Lydike A, Van Delm J, Dutilleul A, et al.
In: Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization. New York, NY, USA: Association for Computing Machinery; 2025 (CGO'25),
https://dl.acm.org/doi/10.1145/3696443.3708952

[5] Insights into Interpreters, Compilers, and Optimizers for Neural Networks.

Alladi S, Ravindranath R, Ros A, Jimborean A

In Proceedings of Workshop on Compilers, Deployment, and Tooling for Edge AI 2025. New York, NY, USA: Association for Computing Machinery;2025.(CODAI '25). Available from: (pending)

[6] xDSL: Sidekick Compilation for SSA-Based Compilers

Fehr M, Weber M, Ulmann C, Lopoukhine A, Lücke MP, Degioanni T, et al

In: Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization. New York, NY, USA: Association for Computing Machinery; 2025. (CGO'25), https://dl.acm.org/doi/10.1145/3696443.3708945

[7] First-Class Verification Dialects for MLIR

Fehr M, Fan Y, Pompougnac H, Regehr J, Grosser T

In the 46th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'25), https://dl.acm.org/doi/10.1145/3729309

[8] Compressed and Parallelized Structured Tensor Algebra

Mahdi Ghorbani, Emilien Bauer, Tobias Grosser, Amir Shaikhha

Proceedings of the ACM on Programming Languages (OOPSLA'25), Volume 9, Issue OOPSLA1 https://dl.acm.org/doi/10.1145/3720506

[9] Certified Decision Procedures for Width-Independent Bitvector Predicates

Siddharth Bhat, Leo Stefanesco, Chris Hughes, Tobias Grosser

Proceedings of the ACM on Programming Languages (OOPSLA'25), Volume 9, Issue OOPSLA2 https://dl.acm.org/doi/10.1145/3763148

[10] Interactive Bitvector Reasoning using Verified Bit-Blasting

Henrik Böving, Siddharth Bhat, Luisa Cicolini, Alex Keizer, Léon Frenot, Abdalrhman Mohamed, Léo Stefanesco, Harun Khan, Joshua Clune, Clark Barrett, Tobias Grosser

Proceedings of the ACM on Programming Languages (OOPSLA'25), Volume 9, Issue OOPSLA2 https://dl.acm.org/doi/10.1145/3763167

2026

[11] The configuration wall: Characterization and elimination of accelerator configuration overhead

Josse Van Delm, Anton Lydike, Joren Dumoulin, Jonas Crols, Xiaoling Yi, Ryan Antonio, Jackson Woodruff, Tobias Grosser, Marian Verhelst, to appear at ASPLOS 2026

[12] Compiler Assisted Instruction Fusion

Ravindranath R, Singh S, Perais A, Ros A, Jimborean A, at CGO 2026 https://doi.org/10.1109/CGO68049.2026.11394845

[13] Enabling Automatic Vectorization for Transformers

Alladi S, Ros A, Jimborean A, CGO 2026 https://doi.org/10.1109/CGO68049.2026.11395226

[14] Effective Tiling for the Snitch Cluster

Emily Sillars, Alexandra Jimborean

Code Generation and Optimization (CGO '26) SRC (Student Research Competition) Silver Medal

https://2026.cgo.org/details/cgo-2026-student-research-competition/7/Effective-Tiling-for-the-Snitch-Cluster

[15] Multi-Partner Project: Optimize edge AI processing through innovative compilation techniques

S. Alladi, A. Lopoukhine, G. Alexandris, A. Nardi-Dei, R. Reddy, C. Lamprakos, P. Chaidos, A. Maras, A Ros, T. Grosser, S. Xydis, D. Soudris, M. Geilen, S. Stuijk, H. Corporaal, and A. Jimborean, to appear at DATE 2026

[16] Leveraging Alias Analysis Without Porting

Ravikiran, Alberto Ros, and Alexandra Jimborean

Code Generation and Optimization (CGO '26) SRC (Student Research Competition)

https://2026.cgo.org/details/cgo-2026-student-research-competition/6/Leveraging-Alias-Analysis-Without-Porting

A2) Abbreviations, Acronyms, and Names

AI	Artificial Intelligence
AXI	Advanced eXtensible Interface, a high-performance synchronous on-chip bus protocol
CAM	University of Cambridge - CONVOLVE partner
CF	The MLIR Control Flow dialect
CGRA	A Course Grained Reconfigurable Architecture is a data-flow hardware design
CAIF	Compiler-Assisted Instruction Fusion
CGO	International Conference for Code Generation and Optimization
Clang	A C++ compiler built on top of the LLVM compiler Infrastructure
CNN	Convolutional Neural Network
CONVOLVE	This EU project
CPU	Central Processing Unit
CSE	Common Subexpression Elimination, a compiler optimization that removes redundant sub expressions.
CSL	Cerebras Software Language, a programming language to program the Cerebras Waverscale Engine
CU	Compute Unit
DMA	Direct Memory Access
DNN	Deep Neural Network
DSE	Design Space Exploration

DSL	Domain Specific Language
EP	Execution Provider, a name for a target backend in IREE. Execution providers may have specialized dialect operations that are translated to the target of the execution provider.
FMA	Fused Multiply Add
FP	Floating Point
FPU	Floating Point Unit
GeMM	Generalized Matrix Multiplication
GCC/G++	A C and C++ compiler framework
GOPs	Giga Operations (a unit)
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
ICCS	Institute of Communication and Computer Systems - CONVOLVE Partner
ID	Instruction Decode Unit
IR	Intermediate Representation, the data-structures a compiler uses to store programs throughout the compilation process
IREE	A modern end-to-end compiler framework for AI
JSON	Short for Java Script Object Notation, a textual data exchange format
KUL	KU Leuven - CONVOLVE Partner
L1 / L2 / L3	The Level 1 / 2 / 3 Cache
LICM	Loop Invariant Code Motion
LinAlg	The MLIR linear algebra dialect
LLVM	An SSA-based Compiler framework that underlies compilers for C++ (clang), Rust (rustc), Swift, and many more.
ML	Machine Learning
MLIR	Acronym and name for the "Multi-Level Intermediate Representation", a modern Compiler Framework
NN	Neural Network
NoC	Network on Chip
NSNet2	A deep neural Network for noise suppression
OML	Our ONNX - MLIR - LLVM - RISC-V End-to-end flow,
ONNX	Open Neural Network Exchange is an open source format exchanging machine learning models between different compile frameworks.
ONNX-MLIR	A frontend that enables importing and modeling of ONNX in MLIR
ORT	ONNX runtime - An execution provider for ONNX
ORT-default	The default ONNX execution provider that runs on the CPU
ORT-SMD	The ONNX execution provider that runs on the CPU exploiting SIMD operations
OpenASIP	An Open Application-Specific Instruction Processor toolset
PASM	Parallel Assembly
PE	Processing Element

PyTorch	An open-source machine learning framework that allows prototyping in Python while offering good performance via various compiler backends.
RISC-V	An open-source instruction set architecture
RNN	Recurrent Neural Network
Sail	A programming language for specifying instruction set architecture semantics
SCF	The MLIR SCF dialect
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
SMT	Satisfiability Modulo Theory
SMTLIB	The SMT Library Standard, that defines different SMT abstractions
SNAX	Project name for KULeuven's OpenGEMM Accelerator Generator
SoC	System on Chip
SSA	Static Single Assignment form, a canonical form of a compiles IR, where each variable is assigned only once.
TCDM	Tightly Coupled Data Memory
TTA	Transport Triggered Architecture
TensorFlow	An open-source machine learning framework developed by Google, offering the best performance for Google's TPUs
UMU	University of Murcia - CONVOLVE Partner
xDSL	A Python-Native clone of MLIR, designed for fast development of compilers
zkVM	Zero Knowledge Virtual Machines

A3) References

[20] OpenGeMM: A High-Utilization GeMM Accelerator Generator with Lightweight RISC-V Control and Tight Memory Coupling. Yi, Xiaoling, Ryan Antonio, Joren Dumoulin, Jiacong Sun, Josse Van Delm, Guilherme Paim, and Marian Verhelst. 2025. ASPDAC 25.
[21] AI and Memory Wall, A. Gholami et al., IEEE Micro, vol. 44, no. 03, pp. 33-39, May-June 2024, doi: 10.1109/MM.2024.3373763
[22] ROAM: memory-efficient large DNN training via optimized operator ordering and memory layout, Huiyao Shu, Ang Wang, Ziji Shi, Hanyu Zhao, Yong Li, Lu Lu, arXiv, 2023
[23] Breaking On-device Training Memory Wall: A Systematic Survey
Shitian Li, Chunlin Tian, Kahou Tam, Rui Ma, Li Li, arXiv, 2023
[24] Combined Scheduling, Memory Allocation and Tensor Replacement for Minimizing Off-Chip Data Accesses of DNN Accelerators, Yi Li, Aarti Gupta, Sharad Malik, arXiv, 2023
[25] Triton: an intermediate language and compiler for tiled neural network computations

Philippe Tillet, H. T. Kung, David Cox, MAPL 2019: Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2019
[26] MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning Author: Michael D. Moffitt, ASPLOS '23: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 Pages 238 - 252, https://doi.org/10.1145/3623278.3624752, 2023
[27] TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators, Martin Maas, Ulysse Beaugnon, Arun Chauhan, Berkin Ilbeyi, ASPLOS 2023: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, Pages 123 - 137, https://doi.org/10.1145/3567955.3567961
[28] ORDERING CHAOS: MEMORY-AWARE SCHEDULING OF IRREGULARLY WIRED NEURAL NETWORKS FOR EDGE DEVICES, Byung Hoon Ahn, Jinwon Lee., Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, Hadi Esmaeilzadeh, Proceedings of the 3 rd MLSys Conference, Austin, TX, USA, 2020
[29] Safe Optimized Static Memory Allocation for Parallel Deep Learning
Ioannis Lamprou, Zhen Zhang, Javier de Juan, Hang Yang, Yongqiang Lai, Etienne Filhol, Cedric Bastoul, Proceedings of Machine Learning and Systems 5 (MLSys 2023)
[30] IREE Intermediate Representation Execution Environment, IREE Developer Team, https://iree.dev, retrieved 05/01/2026
[31] ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators,Mei, L., Houshmand, P., Jain, V., Giraldo, S. and Verhelst, M, *IEEE Transactions on Computers*. 70, 8 (Aug. 2021), 1160–1174. https://doi.org/10.1109/TC.2021.3059962.
[32] Towards Zero-Stall Matrix Multiplication on Energy-Efficient RISC-V Clusters for Machine Learning Acceleration, Luca Colagrande, Lorenzo Leone, Maximilian Coco, Andrei Deaconeasa and Luca Benini, arXiv, 2026. URL: https://arxiv.org/abs/2506.10921
[33] Barry de Bruin, Kanishkan Vadivel, Mark Wijtvliet, Pekka Jääskeläinen, and Henk Corporaal. 2024. R-Blocks: an Energy-Efficient, Flexible, and Programmable CGRA. ACM Trans. Reconfigurable Technol. Syst. 17, 2, Article 34 (June 2024), 34 pages. https://doi.org/10.1145/3656642
[34] S. Hashemi, R. I. Bahar and S. Reda, "DRUM: A Dynamic Range Unbiased Multiplier for approximate applications," 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, 2015, pp. 418-425, doi: 10.1109/ICCAD.2015.7372600.
[35] K. Hepola, J. Multanen and P. Jääskeläinen, "OpenASIP 2.0: Co-Design Toolset for RISC-V Application-Specific Instruction-Set Processors," 2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP), Gothenburg, Sweden, 2022, pp. 161-165, doi: 10.1109/ASAP54787.2022.00034.
[36] Henk Corporaal and Marnix Arnold. 1998. Using Transport Triggered Architectures for Embedded Processor Design. Integr. Comput.-Aided Eng. 5, 1 (January 1998), 19–38.

[37] Adam L. Buchsbaum, Howard Karloff, Claire Kenyon, Nick Reingold, and Mikkel Thorup. 2003. OPT versus LOAD in dynamic storage allocation. In Proceedings of the thirty-fifth annual ACM symposium on Theory of computing (STOC '03). Association for Computing Machinery, New York, NY, USA, 556–564. https://doi.org/10.1145/780542.780624
[38] Michael D. Moffitt. 2024. MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23). Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3623278.3624752
[39] https://github.com/mindspore-ai/mindspore
[40] https://github.com/apache/tvm/tree/cfe1711934f82e56f147f2f5f9f928b5a9b92b3e
[41] C. B. Browne et al., "A Survey of Monte Carlo Tree Search Methods," IEEE Trans. Comput. Intell. AI Games, vol. 4, no. 1, pp. 1–43, Mar. 2012, doi: 10.1109/TCIAIG.2012.2186810.
[42] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance Estimation for Neural Network Pruning," Jun. 2019, arXiv: http://arxiv.org/abs/1906.10771
[43] https://cva6.readthedocs.io/en/latest/
[44] Y. Sui and M. Barbar, SVF: Static Value-Flow Analysis Framework. GitHub repository. Available: https://github.com/SVF-tools/SVF. Accessed: 05/03/2026
[45] Quidditch, IREE compiler and runtime for Snitch. Github repository. Available: https://github.com/opencompl/Quidditch. Accessed: 13/03/2026.
[46] Myrtle, Tiling Cost Model for the Snitch Cluster. Github repository. Available: https://github.com/CAPS-UMU/myrtle/. Accessed: 13/03/2026.
[47] S. Alladi, A. Rosand A. Jimborean, 'OML-vect: Enabling Automatic Compiler-Driven Vectorization of Transformers'. Zenodo, Dec. 21, 2025. doi: 10.5281/zenodo.18006548.
[48] OML-vect: Enabling Automatic Compiler-Driven Vectorization of Transformers. Github repository. Available: https://github.com/CAPS-UMU/onnx-mlir-caps/tree/oml-vect-cgo-2026 . Accessed: 13/03/2026.
[49] CAIF: Compiler Assisted Instruction Fusion. Github repository. Available:https://github.com/CAPS-UMU/llvm-project-caps . Accessed: 13/03/2026.