RV-IR: An MLIR-Based Architecture-Aware Intermediate Representation for Heterogeneous RISC-V AI Acceleration

Zexin Jian
SKLP, Institution of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences
Beijing, China
jianzexin25z@ict.ac.cn

Shuhui Jia
SKLP, Institution of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences
Beijing, China
jiashuhui@ict.ac.cn

Chunwei Xia
University of Leeds
Leeds, United Kingdom
c.xia@leeds.ac.uk

Di Wang
Peking University
Beijing, China
wayne.wangdi@outlook.com

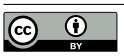
Chenxi Wang
SKLP, Institution of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences
Beijing, China
wangchenxi@ict.ac.cn

Abstract

The growing interest in RISC-V-based AI accelerators creates an opportunity to build open and customizable machine learning systems, but it also exposes a compiler gap between generic tensor programs and accelerator-specific execution semantics. In practical heterogeneous CPU–NPU deployments, the compiler must reason about explicit memory spaces, accelerator invocation boundaries, asynchronous coordination, and software-managed data movement. Existing MLIR infrastructures provide strong support for high-level tensor optimization and progressive lowering, yet these generic abstractions do not always directly encode the architectural contracts needed by RISC-V AI backends.

This paper presents RV-IR, an MLIR-based compilation framework centered on a RISC-V-oriented intermediate representation that serves as an architecture-aware layer between generic tensor dialects and backend-specific code generation. Rather than replacing existing MLIR dialects, RV-IR complements them by making accelerator-relevant concepts explicit, including custom compute operators, memory-space-aware allocation and transfer, hierarchical execution constructs, and synchronization points. The framework supports lowering from PyTorch through torch-mlir into RV-IR, and then into either a generic LLVM-oriented path or an accelerator-oriented path that interfaces with custom RISC-V runtime symbols and custom instruction stubs.

We implement the proposed design in a research prototype based on torch-mlir. Experimental results on simulator-based RISC-V heterogeneous platforms demonstrate the effectiveness of our approach in enabling efficient execution of modern ML workloads.



ICS Workshops '26, Belfast, United Kingdom

ACM ISBN 979-8-4007-2300-1/2026/07
https://doi.org/10.1145/3774895.3812195

Keywords

MLIR, RISC-V, AI accelerator, Heterogeneous compilation, Intermediate representation, Custom instruction extension

ACM Reference Format:
Zexin Jian, Shuhui Jia, Chunwei Xia, Di Wang, and Chenxi Wang. 2026. RV-IR: An MLIR-Based Architecture-Aware Intermediate Representation for Heterogeneous RISC-V AI Acceleration. In *2026 International Conference on Supercomputing Workshops (ICS Workshops '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3774895.3812195

1 Introduction

The increasing demand for efficient machine learning (ML) inference has driven the development of domain-specific accelerators[6]. Recently, RISC-V[7] has attracted growing interest as an open instruction set architecture (ISA) for customizable AI systems because it exposes a clean extension path for domain-specific instructions while retaining a broad software ecosystem. Open full-stack accelerator efforts such as Gemmini further demonstrate the value of integrating custom DNN acceleration into the RISC-V ecosystem[1]. In many realistic deployments, a RISC-V CPU cooperates with a specialized neural accelerator, forming a heterogeneous system that combines flexible control with high-throughput compute.

However, compiling ML workloads for such systems remains difficult. Many RISC-V AI accelerators expose explicit local memories, software-managed data movement, and accelerator-specific invocation interfaces rather than the largely uniform execution model found in conventional CPU or GPU targets[1]. As a result, the compiler must do more than lower tensor operators: it must also preserve the information required to decide where computation runs, how data is placed and transferred, and how CPU and accelerator activities are synchronized.

Current compiler infrastructures already provide important building blocks for this problem, especially PyTorch[5], LLVM IR[2], and MLIR-based multi-level compilation[3]. Nevertheless, in their

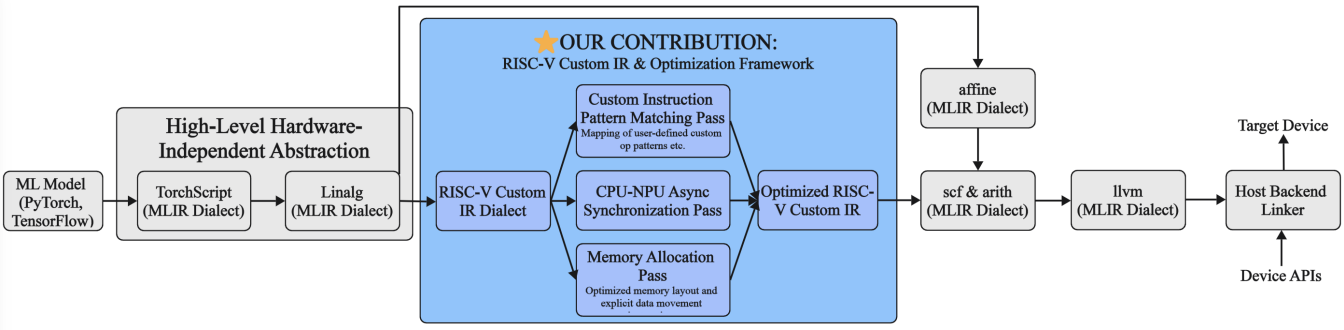


Figure 1: The RISCV AI compiler. The abstraction lowers from left to right.

generic form, these abstractions are not always sufficient to directly express the architectural contracts of a specific RISC-V AI backend. In typical RISC-V heterogeneous systems where a general-purpose CPU is coupled with an NPU, execution follows a split model in which the CPU orchestrates control flow while compute-intensive kernels are offloaded to the accelerator. Such architectures expose explicit synchronization between CPU and NPU, software-managed on-chip buffers, and non-transparent data movement across the memory hierarchy. However, these hardware-level semantics are not natively captured in existing compiler stacks: synchronization is often implicit, memory spaces are abstracted away, and data transfers are introduced only late in lowering or hidden behind opaque runtime calls.

Overall, this work aims to narrow the gap between high-level ML programming frameworks and low-level RISC-V AI hardware by introducing an architecture-aware intermediate layer. We develop RV-IR, a RISC-V-oriented intermediate representation and end-to-end compilation framework for heterogeneous AI acceleration. Positioned between generic tensor dialects and low-level backends, RV-IR makes accelerator-relevant semantics explicit while preserving compatibility with the surrounding MLIR ecosystem, thereby providing a practical foundation for compiler–architecture co-design in open accelerator systems.

Compared with a direct lowering strategy from generic MLIR dialects to backend-specific code, RV-IR offers two practical advantages. First, it provides a dedicated place to model accelerator-oriented compute operations, explicit memory-space management and asynchronous coordination. Second, it exposes a stable interface between compiler IR and backend runtime or custom instruction implementations, which is particularly useful for simulator-driven development and custom RISC-V extensions. target-specific transformation passes.

Our contributions can be summarized as follows:

- We design a RISC-V-oriented dialect that explicitly represents accelerator-facing compute operators, memory spaces, data movement, synchronization, and hierarchical execution constructs.
- We implement an end-to-end compilation framework that maps PyTorch models to RISC-V AI accelerators based on our RV-IR.
- We evaluate our approach on RISC-V simulator, demonstrating its effectiveness in supporting varies modern deep learning (DL) workloads.

2 Design

In this section, we present the design of our RISCV-oriented intermediate representation (RISC-V IR) and the corresponding compilation framework. Our goal is to bridge the gap between high-level tensor abstractions and the low-level execution semantics of heterogeneous RISC-V CPU–NPU systems.

2.1 Design Overview

Figure 1 illustrates the overall compilation pipeline. Our system uses torch-mlir[4] to translate PyTorch programs into MLIR, leverages MLIR as the frontend infrastructure to represent tensor computations using existing dialects such as Linalg. The program is then progressively lowered into our proposed RV-IR, which introduces architecture-aware abstractions tailored for heterogeneous execution. Finally, the RISC-V IR is lowered to hardware-specific instructions, including custom RISC-V extensions for AI acceleration. RV-IR is introduced as a specialization layer between high-level MLIR dialects (e.g., Linalg/vector) and low-level LLVM IR to explicitly capture architecture-specific semantics that are not naturally expressed in existing abstractions. By exposing concepts such as accelerator invocation boundaries, local memory management, data movement, and CPU–NPU synchronization early in the compilation flow, RV-IR enables analyzable, target-aware transformations and provides a stable interface between compiler and backend execution.

2.2 Design Principles of RV-IR

The design of RISC-V IR is guided by the following principles:

Architecture Awareness. Rather than relying exclusively on hardware-agnostic tensor abstractions, RV-IR explicitly models accelerator-facing features such as custom compute operators, explicit memory spaces, and heterogeneous execution boundaries. This enables target-aware reasoning before the program has been flattened into low-level code.

Explicit Data Movement. Many accelerators in our target class use software-managed local memories instead of transparent cache coherence. RV-IR therefore provides explicit transfer operations

Table 1: The partial `riscv` dialect operations

Category	Operation	Description
Computation	`riscv.add`	Scalar or element-wise addition
	`riscv.conv`	Convolution operation
	`riscv.matmul`	Matrix multiplication on tensor tiles
Memory	`riscv.load`	Load data from global memory
	`riscv.store`	Store data to global memory
Communication	`riscv.wait`	Wait for a specific task to complete
	`riscv.signal`	Signal completion of a task
Parallelism	`riscv.parallel_for`	Parallel loop construct
Data Reuse	`riscv.tile`	Tile tensor into smaller blocks

and memory-space annotations so that data motion can be analyzed, scheduled, and eventually mapped to backend services.

Heterogeneous Execution Semantics. To support CPU–NPU collaboration, the IR distinguishes between control-flow execution on the CPU and compute-intensive operations on the accelerator. This separation allows the compiler to selectively offload tensor computations while keeping control logic on the CPU.

Composability and Progressive Lowering. Built on top of MLIR, RV-IR integrates into a multi-level compilation flow instead of bypassing it. High-level tensor operations are progressively rewritten into architecture-aware operations, preserving optimization opportunities while still allowing fallback to more conventional LLVM-oriented lowering paths.

2.3 Architecture-Aware IR Abstraction

RV-IR captures heterogeneous execution through a set of abstractions that directly correspond to the implementation in our system.

Compute Operators. The dialect provides dedicated operators for accelerator-relevant kernels such as matrix multiplication, batch matrix multiplication, convolution, transpose, reduction, and vector operations. These operators preserve high-level intent longer than a direct fall-through to low-level scalar code and therefore create a natural anchor point for target-specific lowering.

Memory Spaces and Transfers. The IR distinguishes memory operations from pure computation and supports explicit allocation, deallocation, and transfer. In the implementation, memory spaces such as global memory, scratchpad-like spaces, and local memory are represented as attributes in the dialect, allowing backend passes to connect IR-level buffers to accelerator memory-management services.

Parallelism. To expose hardware parallelism, RISC-V IR adopts a structured parallel execution model inspired by SPMD programming. Parallel regions are explicitly defined, allowing the compiler to map computation tiles to multiple accelerator units. This abstraction decouples parallelism specification from hardware details, enabling flexible mapping strategies.

Communication and Synchronization. Heterogeneous execution requires explicit coordination between CPU and NPU. RV-IR introduces asynchronous execution constructs and synchronization primitives to manage dependencies between tasks. These constructs allow overlapping of computation and communication, improving overall system efficiency.

Data Reuse. To improve memory efficiency, the IR models data reuse at the granularity of tiles. By explicitly representing allocation, lifetime, and reuse of data in local memory, the compiler can apply optimizations such as tiling and buffering to reduce redundant data movement.

Figure 1 illustrates the overall architecture of our compilation framework, which orchestrates a sequence of multi-level lowering passes to map high-level ML models onto the target RISC-V platform.

Frontend Lowering. The framework starts from PyTorch models and lowers them into MLIR through torch-mlir. At this stage, high-level tensor semantics are represented using generic dialects such as Linalg, which provide a normalized and hardware-agnostic form suitable for frontend analysis and canonical optimization.

RV-IR Synthesis. We then apply a dedicated lowering stage from Linalg to RV-IR. In the current system, this includes mappings such as `linalg.matmul` to `riscv.matmul`, `linalg.transpose` to `riscv.transpose`, and analogous rewrites for reduction, element-wise arithmetic, convolution, and pooling operators. This step is where target-facing semantics first become explicit.

Architecture-Aware Transformation. Once the program has entered RV-IR, the compiler can reason about accelerator-specific execution boundaries, memory spaces, transfer operations, and synchronization points. This is the appropriate level for inserting target-specific passes because the representation still exposes structured tensor intent while already carrying backend-relevant information.

Backend Code Generation. Finally, the optimized RISC-V IR undergoes instruction selection for the target hardware. Control-flow constructs are emitted as standard RISC-V scalar instructions, while compute-intensive segments are lowered into target-specific NPU intrinsics and custom extensions. This results in a synchronized, heterogeneous binary ready for execution on the RISC-V AI acceleration platform.

Together, these stages form an end-to-end compilation stack that combines frontend reuse with target-aware specialization. The main benefit is not merely code generation to RISC-V, but the ability to preserve accelerator-specific meaning long enough for dedicated analysis and transformation.

2.4 Case Study: Mapping Matrix Multiplication to RISC-V IR

```
%2 = linalg.generic {indexing_maps = [#map, #map1],
     iterator_types = ["parallel", "parallel", "parallel
     "]} ins(%transposed : tensor<2048x128256xf32>) outs
     (%1 : tensor<1x2048x128256xf32>) {
^bb0(%in: f32, %out: f32):
  linalg.yield %in : f32
} -> tensor<1x2048x128256xf32>
%3 = tensor.empty() : tensor<1x128x128256xf32>
%4 = linalg.fill ins(%cst_0 : f32) outs(%3 : tensor<1
     x128x128256xf32>) -> tensor<1x128x128256xf32>
%5 = linalg.batch_matmul ins(%arg0, %2 : tensor<1
     x128x2048xf32>, tensor<1x2048x128256xf32>) outs(%4 :
      tensor<1x128x128256xf32>) -> tensor<1
     x128x128256xf32>
```

(a)

```
scf.for %arg1 = %c0 to %c1_3 step %c1_4 {
    scf.for %arg2 = %c0_5 to %c128_6 step %c64_7 {
        scf.for %arg3 = %c0_8 to %c512 step %c128_9 {
            scf.for %arg4 = %c0_10 to %c2048 step %
                 c256_11 {
                riscv.matmul ins(%alloc_12, %alloc_17 :
                     memref<1x64x256xf32>, memref<1
                     x256x128xf32>) outs(%alloc_22 :
                     memref<1x64x128xf32>)
            }
        }
    }
}
```

(b)

Figure 2: GEMM kernel at different abstractions.

To illustrate how RV-IR enables architecture-aware compilation, we present a case study of mapping a matrix multiplication (GEMM) operator from a high-level ML program to a heterogeneous RISC-V system, as shown in Fig. 2.

High-Level Representation. At the frontend level, GEMM is represented as a high-level tensor operation using the PyTorch framework. Subsequently, the frontend model is lowered into the MLIR torch dialect (representing the TorchScript level) using the torch-mlir compiler. This representation captures the high-level semantics of the operation but does not specify how computation is mapped to hardware, nor how data movement and parallelism are handled.

Lowering to RISC-V IR. During lowering, the compiler transforms the high-level tensor operation into structured linear algebra operations in MLIR, as shown in Fig. 2a, and further lowers them into RISC-V IR with explicit loop nests and accelerator invocations (Fig. 2b). Our target NPU exposes a hardware matrix multiplication primitive, `riscv.matmul`, which computes a tile-level GEMM of the form $C_{i,j} += A_{i,k} \times B_{k,j}$. This primitive operates on operands stored in the NPU scratchpad memory and is designed to execute a single tile-sized matrix multiplication per invocation. Due to limited on-chip buffer capacity, `riscv.matmul` imposes fixed input shape constraints: the operand tiles must fit within the scratchpad, which must simultaneously accommodate input matrices and output accumulators. As a result, it cannot directly process arbitrarily large matrices. To support general GEMM workloads, the compiler applies multi-level tiling, as reflected by the nested `scf.for` loops in Fig. 2b. The original large matrix multiplication is decomposed into smaller tiles that match the hardware-supported shapes (e.g., 64×256 and 256×128 in this example). Each tile is explicitly moved into the scratchpad and mapped to a single `riscv.matmul` invocation. The memref operands in Fig. 2b (e.g., `memref<1x64x256xf32>`) are therefore allocated in the NPU scratchpad rather than main memory, making data placement explicit in the IR. This explicit modeling enables the compiler to orchestrate data movement between main memory and the scratchpad, as well as to schedule computation and reuse efficiently. In contrast, scalar or control-heavy operations remain on the CPU.

Lowering to Hardware Instructions. Finally, RV-IR is lowered to hardware-specific instructions. Data movement operations are translated into load/store or DMA instructions, while riscv.matmul is mapped to custom RISC-V tensor instructions.

This example demonstrates how RISC-V IR bridges the gap between high-level tensor abstractions and low-level hardware execution. By explicitly modeling computation, memory, parallelism, and synchronization, our IR enables efficient mapping of ML workloads onto heterogeneous RISC-V systems.

3 Experiment Results

In this section, we evaluate the effectiveness of our RV-IR and compilation framework on heterogeneous RISC-V AI platforms.

3.1 Experimental Setup

3.1.1 Hardware Platform. We evaluate our framework on a simulator-based RISC-V heterogeneous system consisting of a general-purpose RISC-V CPU and a custom AI accelerator (NPU). The NPU supports a set of custom tensor instructions, including matrix multiplication and vector operations, and exposes an explicit memory hierarchy with global memory (GMEM) and on-chip local memory (LMEM). The CPU and NPU communicate through memory-mapped interfaces and synchronization primitives.

3.1.2 Compiler Configuration. Our compilation flow starts from PyTorch models and lowers them through MLIR to our RV-IR, followed by backend lowering to RISC-V instructions. We enable architecture-aware optimizations at the IR level, including tiling, memory placement. Unless otherwise specified, all results are obtained with these optimizations enabled.

3.1.3 Baselines. We compare our approach against two baselines:

(1) CPU-only execution, where all operations are executed on the RISC-V CPU without accelerator offloading;
(2) A naive offloading strategy, where tensor operations are offloaded to the NPU.

These baselines allow us to isolate the impact of our IR design and compiler optimizations.

3.1.4 Workloads. We evaluate a set of representative ML workloads, including:

- Fully connected neural networks (FC)
- Transformer-based models (e.g., attention layers)

These workloads cover a range of computation patterns, including dense linear algebra, convolution, and sequence modeling.

Table 2: Operator-level performance comparison (cycles).

Operator	CPU	CPU+NPU	Speedup (×)
GEMM	694,459	3,019	230.0
Conv	6,353,602	8,434	753.3
FC	1,047,333	3,339	313.7
Multiply	769,260	28,709	28.5
ReLU	962,011	18,080	53.2
Sigmoid	7,314,928	1,878,977	3.9
LayerNorm	1,391,198	89,989	15.5
ReduceMax	813,208	15,937	51.0

3.2 End-to-End Compilation

We first evaluate whether our framework can successfully compile and execute end-to-end ML models on the heterogeneous RISC-V system. All tested models are successfully lowered from PyTorch to RISC-V IR and further to executable code.

Our results show that the compiler can automatically identify compute-intensive tensor operations and offload them to the NPU, while retaining control logic on the CPU. This demonstrates the correctness and completeness of our compilation flow.

3.3 Performance Evaluation

We next evaluate the performance benefits of our approach across different operators and end-to-end workloads.

At the operator level (Table 2), offloading to the NPU provides substantial performance improvements over CPU-only execution. Compute-intensive operators such as GEMM, Conv, and FC achieve speedups of up to 753.3×, highlighting the effectiveness of accelerator utilization. Even for less compute-dense operators, such as element-wise and normalization operations, we still observe consistent speedups ranging from 3.9× to 53.2×.

Compared to a naive offloading baseline, our optimized compilation further improves performance by enabling architecture-aware transformations in RV-IR, such as tiling and data reuse, which reduce memory traffic and improve execution efficiency.

3.4 End-to-End LLM Performance

Table 3: Prefill and Decode performance for Llama3-3B INT4

Model	Prefill	Decode
Llama3-3B INT4	721.3M cycles	10.8M cycles

To further evaluate real-world performance, we measure the execution of a Llama3-3B model quantized to INT4.

As shown in Table 3, for a prompt length of 512, the prefill stage achieves a time-to-first-token (TTFT) of 721.3M cycles, while the decode stage requires 10.8M cycles per token.

These results demonstrate that our system can efficiently support both:

- compute-intensive prefill workloads, and
- latency-sensitive autoregressive decoding,

highlighting its applicability to modern large language model inference.

4 Conclusion

This paper presents RV-IR, an MLIR-based architecture-aware intermediate representation for heterogeneous RISC-V AI acceleration. The central idea is to insert a dedicated RISC-V-oriented layer between generic tensor dialects and backend-specific execution so that custom compute operations, explicit memory handling, and CPU–accelerator coordination remain visible to compiler passes. Our system demonstrates that this approach integrates naturally with torch-mlir and supports both generic LLVM-oriented lowering and accelerator-oriented backend interfacing.

Beyond the current system, the broader significance of RV-IR is methodological: it provides a useful abstraction boundary for co-designing open RISC-V accelerator hardware, runtime interfaces, and compiler transformations. We expect this representation to support future work on richer scheduling, automated partitioning, and more mature backend support as the target hardware and simulator stack evolve.

Acknowledgments

This work was supported in part by the Pioneer Initiative Talent Program B of Chinese Academyof Sciences (No. E545030, E401430)

References

[1] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 769–774. doi:10.1109/DAC18074.2021.9586216

[2] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. doi:10.1109/CGO.2004.1281665

[3] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL] https://arxiv.org/abs/2002.11054

[4] LLVM Project Contributors. 2026. *Torch-MLIR.* https://github.com/llvm/torch-mlir GitHub repository.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library.* Curran Associates Inc., Red Hook, NY, USA.

[6] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2022. AI and ML Accelerator Survey and Trends. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–10. doi:10.1109/hpec55821.2022.9926331

[7] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi. 2014. The RISC-V Instruction Set Manual. https://api.semanticscholar.org/CorpusID:61619035