



**POLITECNICO**  
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE  
E DELL'INFORMAZIONE

EXECUTIVE SUMMARY OF THE THESIS

# Channel-Wise Lowering of ONNX-Based Deep Learning Models for ReRAM Architectures

LAUREA MAGISTRALE IN HIGH PERFORMANCE COMPUTING ENGINEERING -  
INGEGNERIA DEL CALCOLO AD ALTE PRESTAZIONI

**Author:** MICHELE MIOTTI

**Advisor:** PROF. GIOVANNI AGOSTA

**Academic year:** 2024-2025

## 1. Introduction

This work presents a comprehensive redesign of the compilation pipeline for convolution and matrix-vector multiplication operations targeting ReRAM-based processing-in-memory (PIM) architectures. It is mainly motivated by the need to fully exploit the parallelism and efficiency of ReRAM crossbar arrays, while also making the compilation process faster, more maintainable, and scalable for large deep learning models.

Convolutions are a cornerstone of deep learning, especially in convolutional neural networks (CNNs).

## 2. Background

The basis for this work is the RAPTOR compiler, which lowered convolutions by expressing them as a collection of matrix-vector multiplications, slicing input tensors pixel-wise and distributing weights accordingly [3].

### 2.1. Crossbar Arrays

Resistive Random-Access Memory (ReRAM) is a non-volatile memory technology that stores information by changing the resistance of a material. An example of this can be seen in Figure 1, which illustrates a  $3 \times 3$  ReRAM cross-

bar array. Its low power consumption, high density, and ability to perform analog computations make it particularly attractive for processing-in-memory (PIM) architectures. A key feature of ReRAM-based systems is the crossbar array: a two-dimensional grid where memory cells are located at the intersection of word lines and bit lines. Each crossbar can efficiently perform vector-matrix multiplications in a single operation by exploiting Ohm's and Kirchhoff's laws, enabling massive parallelism for deep learning workloads.

### 2.2. PimSIM Architecture

The PimSIM architecture, which is the target of the RAPTOR compiler, is composed of multiple interconnected processing cores. Each core contains a fixed number of ReRAM crossbars, local memory, and a set of registers. Cores are connected via a Network-on-Chip (NoC), allowing for data movement and communication between them. Computation is distributed across cores and crossbars, with careful mapping required to maximize parallelism and minimize communication overhead.

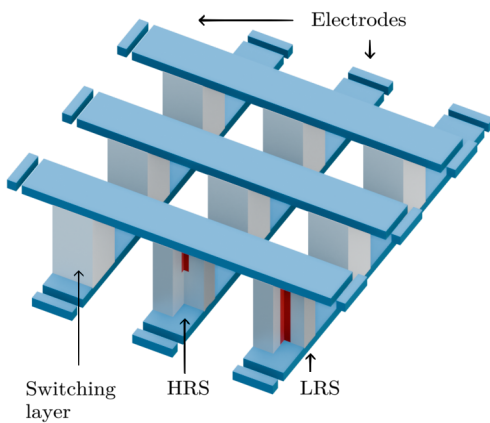


Figure 1: A  $3 \times 3$  ReRAM crossbar array, illustrating high and low resistance states (HRS and LRS) for storing binary values. The crossbar can perform vector-matrix multiplications by applying voltages across the rows and measuring currents across the columns.

### 2.3. The RAPTOR Compiler

RAPTOR is a compiler designed to lower high-level deep learning operations, such as convolutions and matrix-vector multiplications, to the PimSIM architecture. It translates these operations into sequences of instructions that orchestrate data movement, computation, and synchronization across the available cores and crossbars. The compiler’s goal is to efficiently utilize the hardware’s parallelism while maintaining correctness and scalability for large models.

## 3. Novel Methodology

While the previous approach is functionally correct and leverages ReRAM’s strengths, it leads to a very large number of fine-grained operations, making the compilation process slow and the resulting intermediate representation (IR) difficult to debug and maintain.

### 3.1. Channel-Wise Slicing

We introduce a novel lowering strategy that slices tensors channel-wise, keeping the pixel-wise operations implicit, and performing them at a later stage. This change enables the grouping of multiple pixels together, reducing the number of required operations and allowing for more efficient use of crossbar resources. The main logical component behind this approach is the new `applyFilters` operation, which ab-

stracts the complexity of distributing weights and processing input tensors. Instead of expressing every matrix-vector multiplication explicitly, `applyFilters` operates on channel-wise slices and a set of crossbar matrices, producing a portion of the output tensor in a single step. This abstraction not only reduces the size of the IR but also makes the compilation pipeline more modular and easier to extend. This operation can be reused for both convolutions and general matrix-vector multiplications, demonstrating its versatility.

### 3.2. Crossbar Mapping

The new approach slices both input and output tensors into channel-wise tiles, each as wide as the crossbar size. Weights are similarly partitioned, ensuring that each crossbar only stores as much data as it can handle. When the number of channels or filters exceeds the crossbar size, additional slicing is performed across both dimensions. This results in a set of crossbar matrices (represented as two-dimensional squares in Figure 2) that are mapped to physical cores, with careful grouping to minimize inter-core communication.

The use of an implicit crossbar dependency table to categorize and group crossbars according to their input and output tiles is used for minimizing communication overhead and ensuring that related computations are assigned to the same core whenever possible.

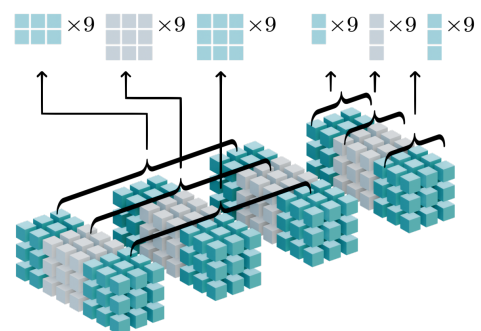


Figure 2: A visual representation of the mapping of weight matrices to crossbars in the new compilation pipeline.

### 3.3. Reduction and Concatenation

Because the output tensor may be split across multiple crossbars and cores, the pipeline includes explicit reduction and concatenation steps. As shown in Figure 3, partial results from different crossbars are reduced (summed) when they correspond to the same output tile, and concatenated when they belong to different tiles. This ensures that the final output tensor is correctly assembled from its distributed components.

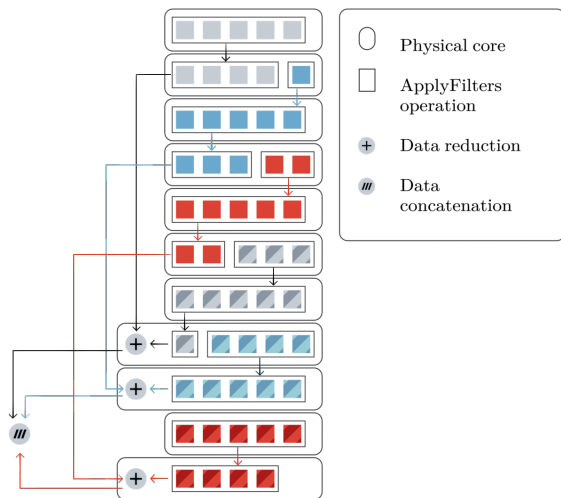


Figure 3: Reduction and concatenation of output tiles from multiple crossbars in the new compilation pipeline.

### 3.4. Bufferization

The use of the Spatial dialect as an intermediate representation abstracts away hardware-specific details while capturing the essential computation and dataflow. The IR expresses the computation as a pipeline of `spat.compute` operations, each invoking `applyFilters` and performing reductions or concatenations as needed.

To target specific hardware, the Spatial IR is further lowered to the PIM dialect, which introduces explicit memory allocation, data loading, and core-to-core communication instructions. A bufferization pass converts high-level tensor operations into direct memory references, preparing the IR for final code generation.

### 3.5. Instruction Generation

In the final stage, the `applyFilters` operation is unwrapped and replaced with a sequence of low-level PIMSim ISA instructions. Addi-

tional strategies for efficient result buffering and memory reuse are performed at this stage, as well as optimizations such as weight skipping, and avoiding unnecessary computations for input pixels that do not contribute to the output due to kernel boundaries.

## 4. Validation

The validation system is a critical component that ensures the correctness and reliability of the compiler’s output, bridging the gap between high-level model specifications and low-level hardware execution.

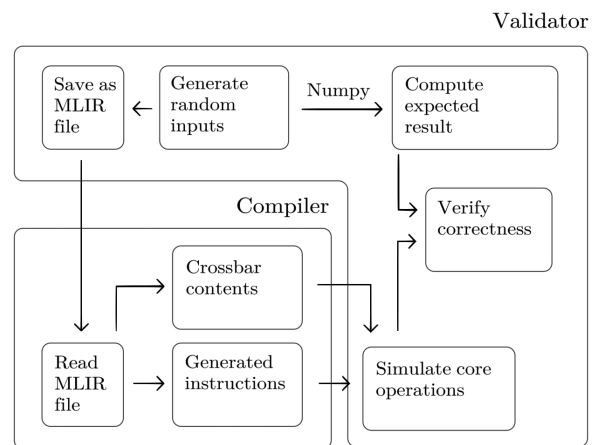


Figure 4: Overview of the validation workflow for the RAPTOR compiler, briefly described in section 4.3. The workflow includes input generation, compilation, simulation, and comparison with reference outputs.

### 4.1. Motivation and Background

The original RAPTOR workflow was primarily focused on generating machine code for sample neural networks and evaluating energy and power consumption using the PIMSim simulator [4]. However, PIMSim is an architectural simulator: it models energy usage but does not simulate the actual register-level execution or memory state changes resulting from instruction execution. This limitation made it impossible to verify the functional correctness of the generated code, as there was no way to check if the output matched the expected results for a given input.

## 4.2. Crossbar Content Pass

A fundamental challenge in validating ReRAM-based compilation is the opaque handling of weights and data within crossbars. The RAPTOR compiler assumes all weights are preloaded into crossbars and that input tensors are stored contiguously for efficient slicing. However, the generated instructions only reference crossbar indices, not the actual data layout or values. This abstraction, while efficient for code generation, complicates validation, as it is not possible to reconstruct the precise computations performed by the hardware.

To address this, the validation system introduces a crossbar content export mechanism. Before the lowering pipeline deletes weight information, the compiler exports a distinct file for each core, detailing the exact location and value of every weight in the crossbars. This enables both manual inspection and automated validation, ensuring that weights are correctly distributed and accessed during computation.

## 4.3. Validation Workflow

The validation workflow is implemented as a standalone Python program and consists of several key stages:

- **Input Generation:** Random input tensors are generated, along with a high-level MLIR [1, 2] description of the target operation (convolution or matrix-vector multiplication).
- **Compilation:** The RAPTOR compiler processes the MLIR file, producing hardware-specific machine instructions and exporting both instruction streams and crossbar contents.
- **Simulation:** The validator simulates the execution of each core, faithfully modeling memory accesses, computations, and inter-core communication via a simulated Network-on-Chip (NoC). This simulation includes register operations and memory hierarchies, closely mirroring the intended hardware behavior.
- **Comparison:** In parallel, the same input tensors are processed using NumPy to compute the expected (reference) output, leveraging well-tested numerical routines. The outputs from the simulated execution are compared to the NumPy reference. Any

discrepancies are flagged, enabling the detection of miscompilations, instruction misinterpretations, or errors in data movement and synchronization.

The validator supports a comprehensive set of instructions, including memory loads/stores, matrix-vector multiplications, vector additions, and inter-core send/receive operations. The higher-level `applyFilters` operation is not directly validated, as it is decomposed into lower-level instructions during code generation.

## 5. ResNet-18 Support

A separate contribution to the RAPTOR compiler was put in place, focused on extending its compatibility with additional deep learning models, specifically ResNet-18. Performance comparisons show that RAPTOR significantly outperforms the pimcomp compiler [4] in both throughput and energy efficiency across various core counts, confirming RAPTOR’s scalability and effectiveness for deep learning workloads on ReRAM architectures. Figure 5 illustrates the performance comparison of the RAPTOR compiler against the pimcomp compiler.

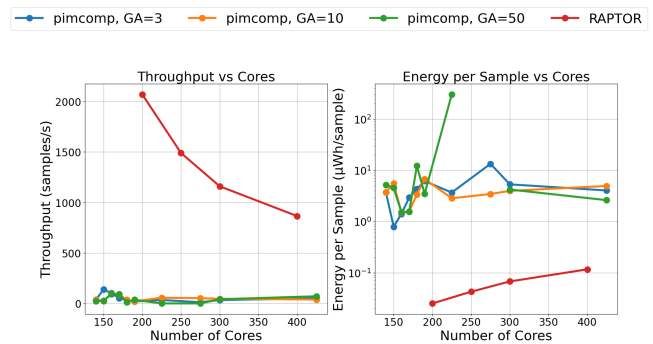


Figure 5: Performance comparison of the RAPTOR compiler against the pimcomp compiler for ResNet-18 on ReRAM architectures.

## 6. Conclusions

The redesigned compilation pipeline, centered on channel-wise data processing, has led to measurable improvements in both efficiency and maintainability. By operating at a higher granularity, the system reduces the number of instructions required for data movement and computation, as larger data blocks are processed per operation. This results in a significant decrease in memory-bound instructions, as demonstrated in

Figure 6 and Figure 7, and contributes to faster compilation times and enhanced overall system performance.

This optimization, however, introduces a trade-off: while fewer instructions are needed, each handles larger data transfers. This approach is well-suited to architectures capable of efficiently managing large data blocks, but may require further adaptation for systems with other types of resource constraints.

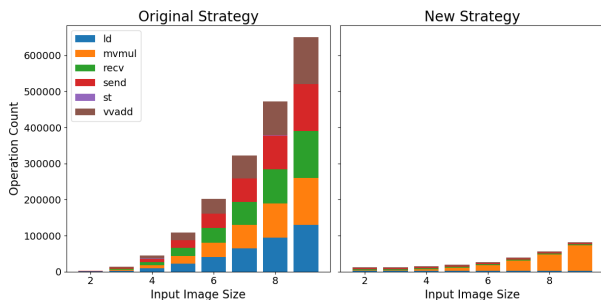


Figure 6: Reduction in memory-bound instructions achieved with the new channel-wise slicing approach compared to the previous pixel-wise method.

Quantitative analysis shows that the new pipeline achieves compilation speeds five to eight times faster than the previous approach, primarily due to the streamlined intermediate representation and reduced instruction count. This not only accelerates the development workflow but also simplifies debugging and future extensions.

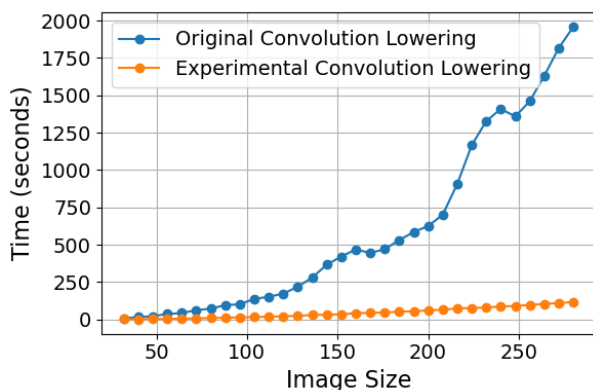


Figure 7: Compilation speedup achieved with the new channel-wise slicing approach compared to the previous pixel-wise method.

### 6.1. Limitations

While the new pipeline delivers clear benefits for convolutional operations, its impact is less pro-

nounced for pure matrix-vector multiplications, where opportunities for data grouping are more limited with respect to the large, channel-wise tensors used in convolutions. Additionally, the current focus has been on optimizing and validating the core set of supported operations; extending these strategies to a broader range of computations and more complex dataflows represents a valid direction for future research.

## References

- [1] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.
- [2] ONNX Contributors. ONNX: Open neural network exchange. <https://github.com/onnx/onnx>, 2024.
- [3] Andrea Somaini. A novel mlir-based compilation flow for accelerating convolutional neural networks on process in memory architectures. 2024.
- [4] Xiaotian Sun, Xinyu Wang, Wanqian Li, Lei Wang, Yinhe Han, and Xiaoming Chen. Pimcomp: A universal compilation framework for crossbar-based pim dnn accelerators, 2023.