POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

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

Tesi di Laurea Magistrale in
High Performance Computing Engineering -
Ingegneria del Calcolo ad Alte Prestazioni

Author: **Michele Miotti**

Student ID: 249499
Advisor: Prof. Giovanni Agosta
Academic Year: 2024-25

Abstract

Deep learning models require vast amounts of computational power and data movement, posing significant challenges for traditional computing architectures. As the need for more efficient processing grows, leveraging specialized hardware, such as ReRAM-based architectures, presents a potential solution for optimizing performance. ReRAM enables processing-in-memory, reducing data transfer bottlenecks and enhancing computational efficiency, making it a promising approach for accelerating deep learning tasks. Particularly, this works focuses on Convolutional Neural Networks (CNNs).

Unlike traditional architectures, ReRAM enables computation near memory, significantly enhancing energy efficiency and performance for AI workloads, provided that all operations are structured in such a way as to take advantage of the different, more specialized architecture.

This thesis presents a redesigned convolution transformation pass within the RAPTOR compiler [10], replacing the existing approach with a more modular, debuggable, and extensible solution. The new implementation distributes convolution weights across crossbars, improving computational efficiency. Additionally, a refined intermediate representation (IR) results in a shorter, more concise computational graph, simplifying debugging, and clarifying the main logical structure of the operation. A validation mechanism for the generated code was also introduced, ensuring correctness by automatically performing all relevant operations, memory accesses, and core-to-core communication.

The proposed improvements contribute to the broader effort of accelerating deep learning on ReRAM architectures by providing a more maintainable and generalizable compilation framework. This work lays the groundwork for future optimizations and extensions in ReRAM-based deep learning acceleration.

Keywords: ONNX, MLIR, ReRAM, PIM, Deep Learning, Compilers

Abstract - Italiano

I modelli di deep learning richiedono un'enorme quantità di potenza computazionale e data movement, creando sfide significative per le architetture di calcolo tradizionali. Per migliorare l'efficienza del calcolo, una possibile soluzione è l'adozione di hardware specializzati, come le architetture basate su ReRAM. Questa tecnologia consente di eseguire i calcoli direttamente nella memoria, riducendo i bottleneck legati al trasferimento dei dati e aumentando l'efficienza di calcolo, rendendola particolarmente adatta per accelerare i carichi di lavoro nel deep learning. In particolare, questo lavoro si concentra sull'ottimizzazione delle Reti Neurali Convoluzionali (CNN).

A differenza delle architetture convenzionali, la ReRAM permette di eseguire le operazioni vicino alla memoria, con un notevole miglioramento dell'efficienza energetica e delle prestazioni. Tuttavia, per sfruttarne appieno il potenziale, le operazioni devono essere strutturate in modo da adattarsi alle specificità di questa architettura più specializzata.

Questa tesi introduce una nuova versione del convolution transformation pass all'interno del compilatore RAPTOR [10], sostituendo l'approccio precedente con una soluzione più modulare, chiara, e facilmente estendibile. La nuova implementazione gestisce in modo più efficace la distribuzione dei pesi delle convoluzioni tra le crossbar, riducendo il tempo di compilazione. Inoltre, una intermediate representation (IR) più compatta e leggibile semplifica il debug e rende più chiara la struttura logica dell'operazione. È stato inoltre introdotto un meccanismo di validazione automatizzato, che verifica la correttezza del codice generato simulando le operazioni, gli accessi alla memoria e la comunicazione tra i core.

I miglioramenti proposti rappresentano un passo avanti nell'accelerazione del deep learning su architetture ReRAM, offrendo un framework di compilazione più manutenibile e adattabile a future ottimizzazioni ed estensioni.

Parole chiave: ONNX, MLIR, ReRAM, PIM, Deep Learning, Compilatori

Contents

1 | Introduction

With the rapid advancement of deep learning models, the demand for efficient computation and data movement has become increasingly critical. Particularly, convolutional neural networks (CNNs) have gained significant attention due to their ability to recognize patterns and features in large grid-like data structures, such as images, as well as audio and video. Whenever spatial hierarchies are present, CNNs are the go-to solution for a variety of tasks, which explains their prominence in the field of deep learning.

However, the computational requirements of these models have begun to outpace the capabilities of traditional computing architectures, where data has to be moved back and forth between memory and processing units. This data movement can create significant bottlenecks, leading to increased latency and energy consumption. As a result, there is a growing need for specialized hardware architectures that can efficiently handle the unique demands of deep learning workloads.

In the context of CNNs, convolution operations are particularly computationally intensive, as they involve applying a set of filters to often large, multi-dimensional input tensors. These processes can be expressed as sets of matrix-vector multiplications, which can be efficiently executed on hardware architectures that keep data close to the processing units. One such architecture is ReRAM (Resistive Random Access Memory), which enables processing-in-memory (PIM) capabilities. This allows for computation to be performed directly within the memory, significantly reducing data transfer bottlenecks and improving overall performance.

Compiling deep learning models for ReRAM architectures requires a specialized approach, which may significantly change depending on the underlying hardware. Crucially, the presence of a single compiler, capable of lowering networks to an architecture-agnostic intermediate representation (IR), is clearly needed. This idea was originally proposed by the RAPTOR compiler [10], which is capable of lowering ONNX-based deep learning models through a chain of transformation passes, one of which expresses all operations in a unified intermediate representation, that does not strictly depend on the underlying hardware, only keeping a set of assumptions that are considered to be common with

respect to all ReRAM architectures.

In order to achieve this, the RAPTOR compiler explicitly expresses some operations as a series of matrix-vector multiplications, which can be efficiently executed on ReRAM architectures. However, the original approach had a number of limitations, particularly in terms of modularity and maintainability. All operations and tensor slices were explicitly expressed within IRs, up to a pixel-level granularity, which made the generated code difficult to debug and understand for large, complex networks. This also made it slower to compile, as RAPTOR had to generate a large number of operations, which could be expressed more concisely, and pass them through the entire compilation process. This thesis presents a number of improvements to the RAPTOR compiler:

- A redesigned convolution lowering pass, which expresses convolutions as a series of matrix-vector multiplications, faithfully following the original approach, while keeping the overall structure of the operation clear and concise, measurably improving the compilation time.
- A new `applyFilters` operation, which allows for a more modular and maintainable representation of the convolution operation, making it possible to express both regular matrix-vector operations and convolutions in a fraction of the size. Despite its name, the `applyFilters` operation is not limited to convolutions: it was also used to rewrite the general matrix-vector multiplication operation, demonstrating its versatility and general applicability to a broader class of linear operations.
- A validation mechanism for the generated code, which ensures correctness by automatically performing all relevant operations, memory accesses, and core-to-core communication.
- Compatibility for ResNet, a well-known deep learning architecture, which could not be fully supported by the original set of allowed operations.

The proposed improvements contribute to the broader effort of accelerating deep learning on ReRAM architectures by providing a more maintainable and generalizable compilation framework. This lays the groundwork for future optimizations and extensions in ReRAM-based deep learning acceleration.

2 | Background

In this section, we provide an overview of the key concepts and technologies that form the foundation of this work. First, we delve into ReRAM architectures, a cutting-edge memory technology that enables processing-in-memory, offering significant advantages for deep learning workloads. Next, we introduce our base compiler framework (ONNX-MLIR) that bridges the gap between high-level machine learning models and hardware-specific implementations. Finally, we introduce the RAPTOR compiler (on which this work is based), which extends the ONNX-MLIR compiler to target ReRAM architectures, enabling efficient execution of deep learning models.

2.1. ReRAM Architectures

Resistive Random-Access Memory (ReRAM) is a memory technology that stores binary information using changes in electrical resistance [2]. Unlike charge-based memories such as DRAM or SRAM, ReRAM encodes data by switching between high-resistance and low-resistance states within a material.

Each resistive memory cell consists of two electrodes with a switching layer sandwiched in between. When a voltage is applied across the electrodes, changes occur within the switching material: either the movement of defects (such as oxygen vacancies) or metal ions, depending on the device type. In the low-resistance state (LRS), a continuous conductive filament forms between the electrodes, allowing electrons to flow easily through the device. This is depicted on the left side of the diagram. Applying a different voltage can break this filament, interrupting the flow of electrons and returning the cell to a high-resistance state (HRS), as shown on the right. These two states are used to represent binary data. [5]

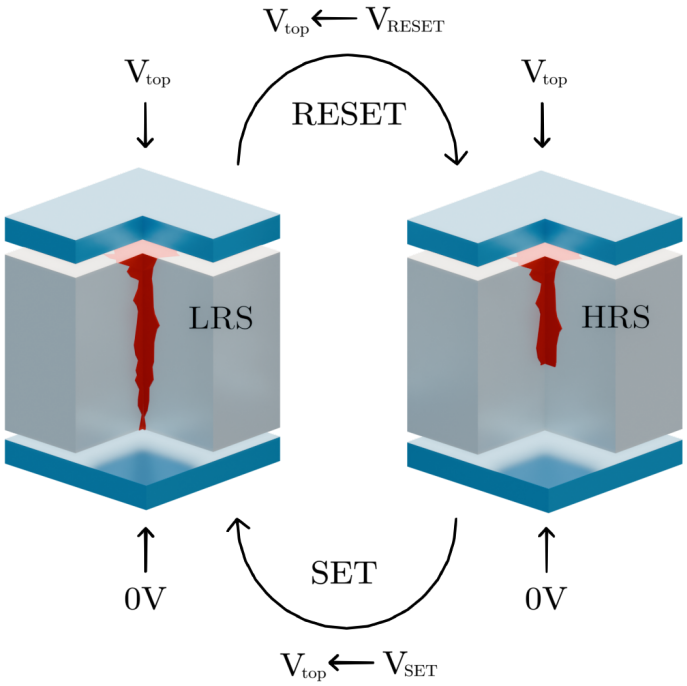

Figure 2.1: A simplified representation of a ReRAM cell, showing the low-resistance state (LRS) and high-resistance state (HRS).

A key architectural feature of ReRAM is its organization into dense crossbar arrays. Each intersection in the array can act as a computational unit as well as a memory cell. When voltages are applied along the wordlines and sensed across the bitlines, the current measured at each bitline corresponds to the sum of the products of the input voltages and the conductances of the cells. This property allows ReRAM arrays to perform analog matrix-vector multiplications efficiently, harnessing Kirchhoff's and Ohm's laws directly in hardware.

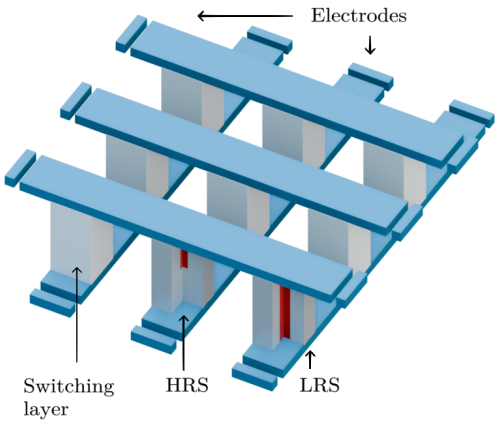

Figure 2.2: A simplified representation of a ReRAM crossbar array.

This ability to perform computation directly in the memory array eliminates the need to transfer data between memory and processing units, a limitation of traditional von Neumann architectures. As a result, ReRAM is especially well suited for workloads that involve dense linear algebra, such as those found in deep neural networks.

Performing Matrix-Vector Multiplications

As shown in figure 2.2, a ReRAM crossbar consists of a two-dimensional grid of programmable resistive devices, organized at the intersections of horizontal word lines and vertical bit lines. Each ReRAM device at intersection (i, j) exhibits a conductance G_{ij}, which is used to encode a weight or matrix element.

To perform MVM, an input vector $\mathbf{V} = [V_1, \ldots, V_n]^\top$ is applied as voltages to the word lines. Due to Ohm's Law, the current flowing through each ReRAM cell is given by $I_{ij} = G_{ij}V_i$. These currents flow into the bit lines, where they are summed according to Kirchhoff's Current Law. The total current measured at bit line j is thus:

$$I_j = \sum_i G_{ij}V_i$$

This operation is equivalent to computing the dot product between the input vector $\mathbf{V}$

and the j-th column of the conductance matrix $\mathbf{G}$, yielding the output current vector $\mathbf{I} = \mathbf{G}^T\mathbf{V}$.

This process allows for all outputs of the matrix-vector multiplication to be computed simultaneously in a single step, exploiting the inherent parallelism of the crossbar architecture. In contrast to traditional von Neumann architectures, where matrix-vector multiplication requires sequential data transfers and computation, the ReRAM crossbar approach significantly reduces energy and latency overheads.

Constant-Time Operations

One of the most striking features of ReRAM crossbar arrays is that they can perform matrix-vector multiplications in apparent constant time. Since all multiply-accumulate operations occur simultaneously across the physical array (where conductances represent matrix weights and input voltages encode the vector) the entire MVM resolves in a single analog step, provided the matrix fits within the array. From a computational perspective, this enables what looks like a truly constant-time operation, something fundamentally out of reach for traditional digital architectures.

However, this impression depends on idealized conditions. The matrix size must be small enough to fit in a single array, and the time required to load inputs and read outputs (handled by DACs and ADCs) does not disappear. So while the core compute step is constant-time in principle, the overall cost still reflects practical constraints.

The focus of this thesis is contributing to a compiler that can generate code for ReRAM-based architectures. By mapping high-level operations, such as convolutions in neural networks, to low-level instructions that leverage the parallelism of ReRAM arrays, the compiler aims to exploit the architectural strengths of ReRAM for efficient execution.

2.2. ONNX-MLIR

In order to interpret and compile deep learning models, relying on the Open Neural Network Exchange (ONNX) format for model representation makes it easier to work with different frameworks and hardware architectures. ONNX [7] is an open format for representing models and networks for machine learning tasks; in particular, it's designed for interoperability between different deep learning frameworks (such as PyTorch or TensorFlow). As an example, training a model in PyTorch and then exporting it to the ONNX format allows the model to be used in other frameworks for inference, even if PyTorch is not natively supported by them.

Emerging hardware architectures, such as ReRAM-based systems (discussed in section 2.1), can also benefit from ONNX's flexibility. By providing a common representation for models, ONNX enables the development of specialized compilers and optimization tools that can target specific hardware architectures, such as ReRAM. This helps avoid the need for custom implementations for each framework, allowing developers to focus on optimizing the performance of their models on specific hardware. Furthermore, ONNX represents networks as a graph of operations, where each node corresponds to a specific operation (such as convolution, activation functions, etc.) and the edges represent the data flow between these operations. This graph-based representation allows for easier optimization and transformation of the model, especially through MLIR (Multi-Level Intermediate Representation) [4].

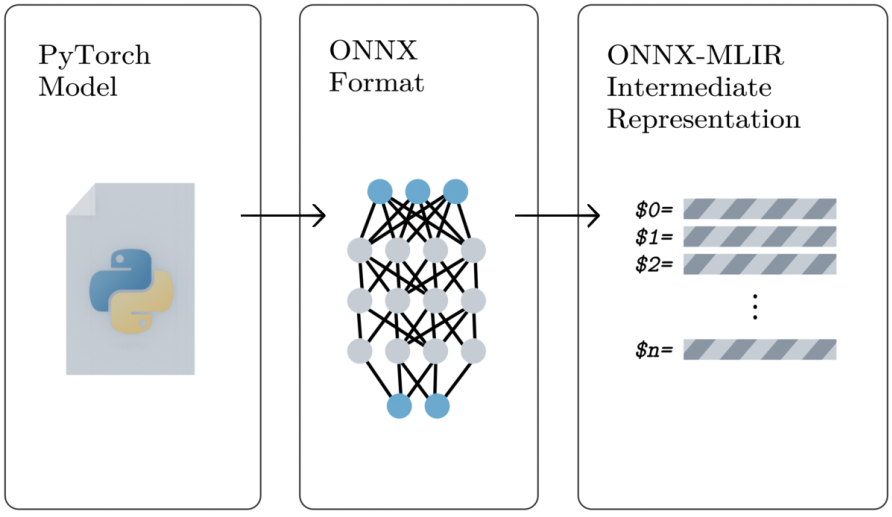

Figure 2.3: A PyTorch model may be exported to the ONNX format, which would then be compiled into a set of MLIR operations.

2.2.1. Multiple Levels of Intermediate Representation

MLIR's strength lies in its ability to represent different levels of abstraction, from high-level operations to low-level hardware-specific instructions through the use of dialects. Each dialect can represent a specific set of operations or data types, allowing the developer

to express the model in a way that is most suitable for the current stage of compilation.

This process aims to separate the compilation process into a set of stages, where each stage can be optimized independently, while still allowing for operations belonging to different dialects to be expressed in a single IR. In this work, it is primarily used to separate the architecture-independent aspects of the compilation process from the architecture-specific ones. For instance, organizing some operation's weights across ReRAM crossbars is considered architecture-independent, as it can be applied to any ReRAM architecture. However, memory allocation and core-to-core communication may vary significantly between different technologies and are therefore considered architecture-specific. By separating these two abstraction classes, the compilation process can be more modular and maintainable, allowing for easier updates and improvements in the future.

2.2.2. Network Expression Through MLIR

ONNX-MLIR is a compiler that translates ONNX models into a set of MLIR operations, exploiting the graph-based representation of ONNX to express the model in a way that is suitable for optimization and transformation. Nodes in the ONNX graph are translated into MLIR operations, while edges are translated into MLIR values. This is done thanks to the addition of a new "ONNX" dialect to the MLIR framework, which provides a set of operations and types, forming a one-to-one mapping between ONNX operations and ONNX-MLIR operations.

Listing 2.1 shows a simplified example of an ONNX-MLIR IR. The operations are expressed in a way that is similar to the original ONNX graph while using MLIR's syntax and semantics. The `onnx.Conv` operation represents a convolution operation, while the `%4` value represents its output, which can later be used as an input to some other operation, further down the IR. This way, the core graph-based representation of ONNX is faithfully preserved, while still allowing for the use of MLIR's powerful optimization and transformation capabilities.

```
1 %4 = "onnx.Conv"(%0, %1, %2) {kernel_shape = [3, 3], pads = [1, 1, 1, 1], strides = [1,
      1]} : (tensor<1x512x14x14xf32>, tensor<512x512x3x3xf32>, tensor<512xf32>) -> tensor
      <1x512x14x14xf32>
2 %5 = "onnx.Relu"(%4) : (tensor<1x512x14x14xf32>) -> tensor<1x512x14x14xf32>
3 %6 = "onnx.MaxPoolSingleOut"(%5) {kernel_shape = [2, 2], pads = [0, 0, 0, 0], strides =
      [2, 2]} : (tensor<1x512x14x14xf32>) -> tensor<1x512x7x7xf32>
4 %7 = "onnx.Flatten"(%6) {axis = 1 : si64} : (tensor<1x512x7x7xf32>) -> tensor<1
      x25088xf32>
```

Listing 2.1: A simplified example of an ONNX-MLIR IR.

This compiler is primarily meant to be used to generate machine code for specific hardware architectures. Crucially, it is open to extensions, allowing developers to add custom dialects and operations. This is particularly useful for the RAPTOR compiler, which will be introduced in the next section. ONNX-MLIR's flexibility and extensibility make it a powerful tool for optimizing deep learning models for various hardware architectures, including ReRAM-based systems.

2.3. The RAPTOR Compiler

Exploiting the ONNX-MLIR compiler's extensibility, the RAPTOR compiler is built on top of it, providing a set of custom dialects and operations that are specifically designed for ReRAM architectures. This allows the compiler to take advantage of the unique features of ReRAM, such as processing-in-memory and near-memory computation, while still being able to leverage the powerful optimization and transformation capabilities of MLIR.

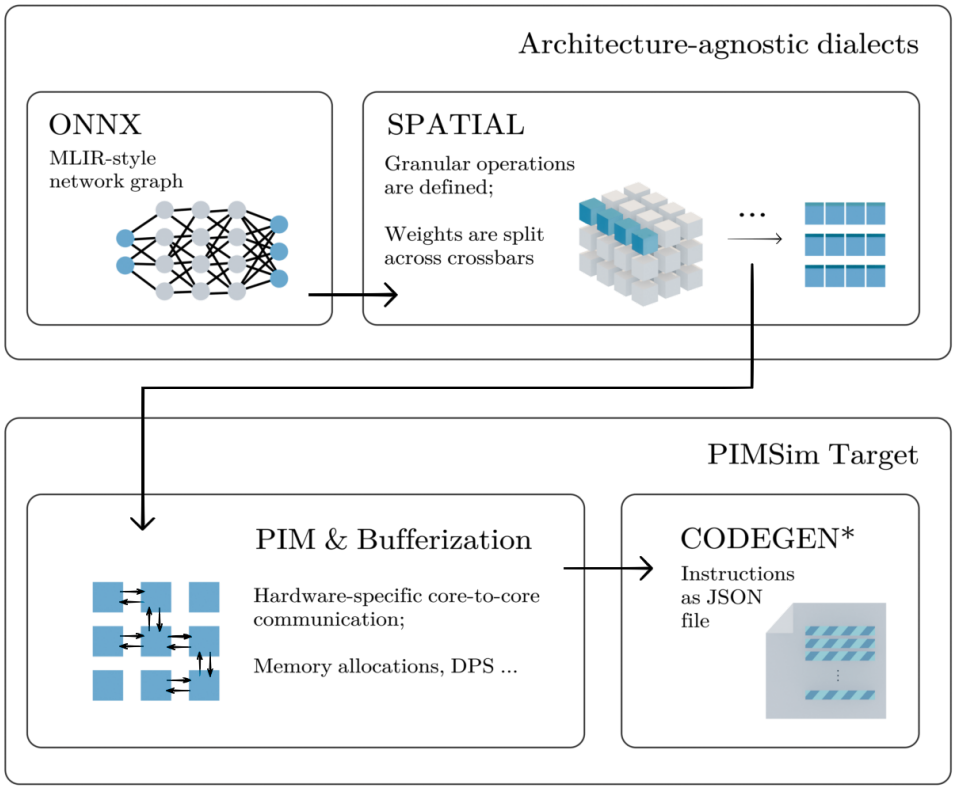

Figure 2.4: The RAPTOR compiler's main stages, expressed as a series of custom dialects. Note that code generation is not a dialect in itself, but rather the last stage of the compilation process.

Any compatible network is compiled by passing through a series of transformation passes, resulting in as many JSON files as there are physical cores in the target architecture. Each JSON file contains the set of operations that, once loaded into the target architecture (in RAPTOR's case, PIMSim), faithfully express the original network. The compilation process is divided into a number of stages, briefly described below. A more thorough description of the compilation process can be found in the original paper [10].

- **ONNX**: The stage flow that the network has to follow begins with the ONNX-MLIR compiler, which translates the ONNX model into a set of MLIR operations. This provides a starting point for the overall compilation process.
- **SPAT**: The first stage of the lowering pipeline is the Spatial (SPAT) transformation pass, which unpacks common operations into an implementation that takes ReRAM architectures into account. This includes operations such as convolutions, pooling, and general matrix-vector multiplications. The SPAT pass is responsible for partitioning weights across crossbars, and describing the set of operations that need to be performed in order to reconstruct the original macro-operation, without taking the specific communication protocol into account.
- **PIM & Bufferization**: Once the SPAT pass has been applied, two stages implement all hardware-specific operations, such as memory allocation and core-to-core communication. Destination Passing Style (DPS) [6] is used to pre-allocate output buffers for each operation, and all data types are converted from MLIR's tensor types to direct memory references. This is essentially a bufferization pass, right before all machine code is generated.
- **CODEGEN**: The final stage of the compilation process is the CODEGEN pass, which generates machine code for the target architecture. MLIR operations are translated into a set of machine instructions that can be executed on the target architecture. The generated code is then stored in a set of JSON files, one for each core in the target architecture.

RAPTOR's strength lies in its separation of architecture-independent and architecture-specific stages, which avoids the need for custom implementations for each framework.

3 | State of the Art

This section provides an overview of the current state of the art in ReRAM and PIM-based deep learning accelerators, focusing on both hardware architectures and their associated compilation frameworks. We discuss representative systems and compilers, highlighting their architectural features. By examining these solutions, we aim to contextualize the contributions of this particular work within the broader landscape of deep learning acceleration.

3.1. PIMCOMP

PIMCOMP [11] is a universal compilation framework for Non-Volatile Memory (NVM) crossbar-based Processing-in-Memory Deep Neural Network (DNN) accelerators, used to compile DNNs into a set of instructions that can be executed on a variety of PIM architectures. Its simulator, PIMSim, was used as the architectural target for the RAPTOR compiler.

Architecture PIMCOMP is built upon an abstract configurable PIM accelerator template that is compatible with hierarchical structures like Crossbar/IMA/Tile/Chip. This abstract architecture typically consists of a series of cores connected to a global memory, with each core containing PIM matrix units and vector functional units.

Compiler PIMCOMP is an end-to-end DNN compiler that takes a high-level DNN description (like ONNX) and translates it into pseudo-instructions for the abstract PIM accelerator. The compilation process [12] involves four main stages: node partitioning, weight replicating, core mapping, and dataflow scheduling. It supports two compilation modes for high-throughput and low-latency scenarios.

3.2. CIM-MLC

CIM-MLC [8] is a universal multi-level compilation framework for general Computing-In-Memory (CIM) architectures.

Architecture CIM-MLC establishes a general hardware abstraction for CIM architectures and computing modes to represent various CIM accelerators. Similarly to PIM-Comp, it models CIM-based DNN accelerators as a hierarchical architecture with three tiers: chip, core, and crossbar. It also defines a three-level computing abstraction (Core Mode, Crossbar Mode, Wordline Mode) corresponding to these tiers.

Compiler CIM-MLC features multi-level scheduling techniques to optimize the inference of DNNs on CIM architectures. The scheduling process optimizes computations from coarse-level computing graphs to fine-level vector computing operations, generating a meta-operator flow for the CIM accelerator.

3.3. ISAAC

ISAAC (In-Situ Analog Arithmetic in Crossbars) is a pipelined system designed specifically for accelerating the inference phase of Convolutional Neural Networks (CNNs) and Deep Neural Networks (DNNs).

As shown in figure 3.1, ISAAC organizes memristor crossbar arrays into tiles, with each tile also containing eDRAM buffers and digital units for shift-and-add operations, activation functions, and pooling. It leverages memristor-based crossbar arrays to execute multiply-accumulate operations directly where the weights are stored, thereby drastically reducing data movement and improving energy efficiency.

At the top level, an ISAAC chip comprises multiple computational tiles, organized in a 2D grid. These tiles communicate over an on-chip interconnect, which provides high-bandwidth, low-latency communication pathways. Each chip also includes an external I/O interface, which is responsible for injecting inputs and extracting final outputs, as well as for coordinating communication between different ISAAC chips in a multi-chip system.

3.3.1. Tile Microarchitecture

Each tile is a self-contained computational unit responsible for a portion of the overall CNN workload. A tile includes the following key components:

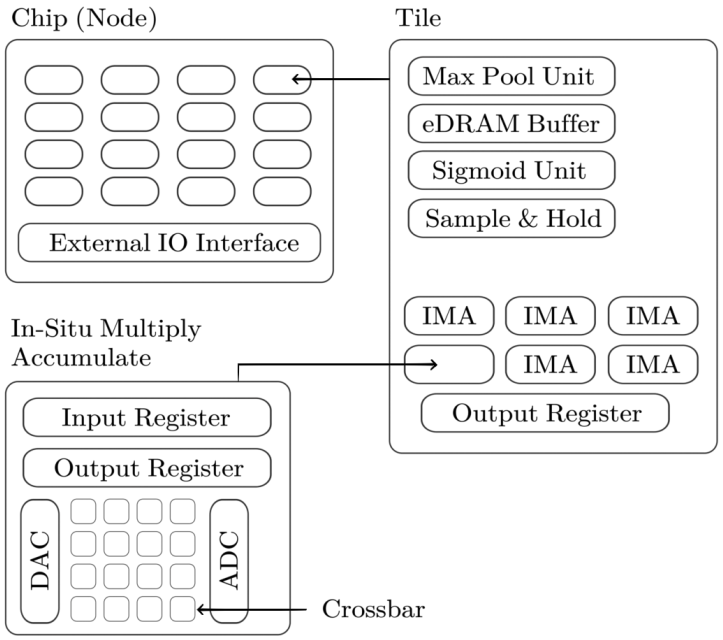

Figure 3.1: ISAAC architecture hierarchy, illustrating the composition of a chip from tiles, tiles from IMAs, and IMAs from memristor crossbars and conversion logic.

- **eDRAM Buffer:** Acts as a staging area for incoming input activations and intermediate results. It supports local reuse of data across multiple computation cycles.
- **In-situ Multiply Accumulate (IMA) Units:** Each tile contains several IMA units that carry out the core analog dot-product computations using memristive crossbars.
- **Output Registers (OR):** These registers accumulate and temporarily store the digital outputs produced by the IMA units before they are passed to nonlinear and pooling units or written to eDRAM.
- **Functional Units:** These include sigmoid (σ) units for non-linear activation functions and Max Pooling (MP) units for spatial downsampling.

All components within the tile are connected via an internal shared bus, which facilitates high-speed data movement between storage, computation, and control units.

3.3.2. Inference Flow and Control

During inference, pre-trained weights are programmed into the memristor crossbars via a one-time write process. Additionally, each tile receives control vectors that configure its internal finite state machines (FSMs), dictating how data should be routed and processed over the course of an inference pass.

The inference process begins when input activations are injected into the system through the external I/O interface. These inputs are routed to the appropriate tiles assigned to handle the first convolutional layer of the network. Within each tile, the FSM dispatches these inputs to the IMA units, where the dot products corresponding to convolution operations are computed in analog. These analog results are captured and digitized, aggregated through shift-and-add units, and passed through any required nonlinear transformations or pooling.

The resulting feature maps are stored in the tile's eDRAM buffer and routed to the next layer's tiles, repeating the process until the final output is computed. The hierarchical organization of ISAAC (chips, tiles, IMAs, and crossbars) enables deep pipelining and efficient mapping of CNN layers, ensuring scalability and high throughput.

3.3.3. Design Philosophy

The hierarchical design allows ISAAC to minimize data movement, one of the major energy bottlenecks in modern deep-learning accelerators. The process of mapping CNN/DNN applications to the ISAAC architecture is done offline. This mapping involves layer partitioning across tiles, weight loading into memristor cells, control vector loading, static scheduling of data transfers, and aggregation of results. This offline mapping process [9] serves a similar function to a compiler by configuring the hardware for a specific neural network.

3.4. PRIME

PRIME is a processing-in-memory (PIM) architecture that allows a portion of ReRAM crossbar arrays within the main memory to be dynamically configured as either neural network accelerators or as conventional memory.

3.4.1. Architecture

PRIME's architecture involves a ReRAM main memory design where a portion of the memory arrays (Full Function Subarrays in Figure 3.2) can be dynamically configured as NN accelerators or as normal memory. It includes Buffer Subarrays for efficient data transfer and a PRIME controller to manage the operation [3]. Unlike traditional NN accelerators that rely on dedicated processing units (PUs), PRIME performs inference directly within memory arrays, eliminating data movement bottlenecks and maximizing parallelism.

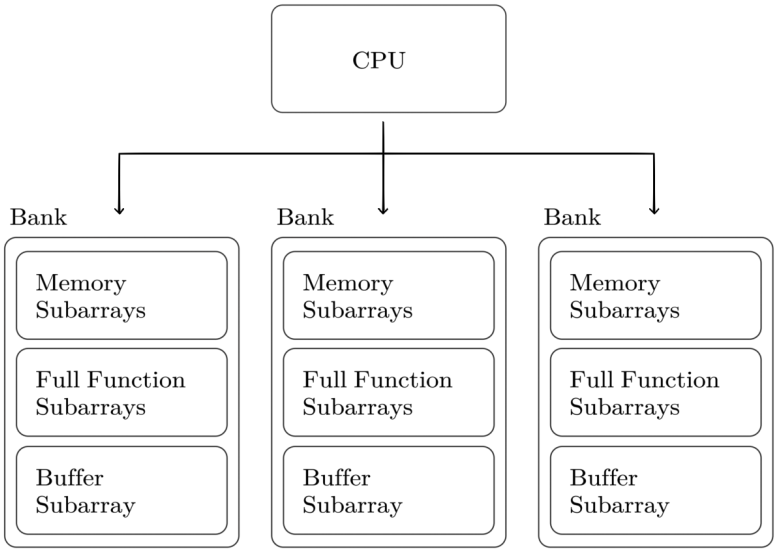

Figure 3.2: PRIME architecture with compute-enabled ReRAM banks.

PRIME departs from conventional architectures by embedding neural network computation capabilities within the memory layer itself. A ReRAM bank in PRIME is partitioned into three regions:

- **Memory Subarrays:** Function as conventional memory and are optimized for storage. They do not participate in computation.
- **Full Function Subarrays:** Capable of both storage and computation. In memory mode, they behave like regular memory; in compute mode, they perform vector-matrix multiplications directly in ReRAM cells. Mode switching is controlled by a dedicated PRIME controller.
- **Buffer Subarrays:** Temporarily store intermediate data for FF subarrays. They

are physically close to FF units and connected via private ports to reduce contention on memory bandwidth. When not used as buffers, they revert to standard memory use.

The PRIME controller orchestrates the operation of FF subarrays and manages mode switching. Importantly, PRIME enables parallel operation between the CPU and the memory compute units, improving system throughput. Since data remains largely in-place during computation, PRIME minimizes energy-intensive memory transfers that are common in traditional architectures.

Compiler The PRIME compiler takes a high-level description of a neural network and optimizes its mapping and execution on the PRIME architecture. This involves mapping the NN topology, programming synaptic weights, configuring data paths, and issuing execution commands. The compiler performs optimizations based on the size of the neural network, using strategies like replication for small NNs, split-merge for medium-sized NNs, and inter-bank communication for large-scale NNs. It also optimizes data allocation within the memory hierarchy.

3.5. PUMA & PUMA-Compiler

PUMA (Programmable Ultra-efficient Memristor-based Accelerator) [1] is a three-tiered spatial architecture designed for efficient machine learning inference using memristor crossbars.

Three-Tier Spatial Architecture

	Contents	Interconnections
`Core`	A core consists of functional units (FUs), crossbars, and the instruction execution pipeline	Interconnected through tiles, using a shared memory to pass information from core to core.
`Tile`	A set of multiple cores, connected through shared memory.	Through an on-chip network, tiles can be connected to each other, allowing for communication and synchronization.
`Node`	A node is a set of tiles, connected through an on-chip network.	Chip-to-chip interconnects can also be used to connect multiple nodes together, allowing for larger systems.

Table 3.1: A summary of the three-tier architecture of PUMA.

Architecture PUMA consists of cores, tiles, and nodes. Cores are the basic building blocks, featuring an instruction execution pipeline, a Matrix-Vector Multiplication Unit (MVMU) with memristor crossbars, a Vector Functional Unit (VFU) for linear and non-linear vector operations, a register file, and a memory unit. Tiles comprise multiple cores connected to a shared memory, which facilitates communication and synchronization. Nodes consist of multiple tiles connected via an on-chip network, with the potential for chip-to-chip interconnects for larger systems.

PUMA-Compiler The compiler is a runtime compiler implemented as a C++ library that translates high-level code (including ONNX models) to PUMA assembly code for each core and tile. The compilation process involves graph partitioning, instruction scheduling (aiming to reduce register pressure, coalesce MVM operations, and avoid deadlock), and register allocation. PUMA's microarchitecture techniques are exposed through a specialized Instruction Set Architecture (ISA) designed to retain the efficiency of in-memory computing and analog circuitry while ensuring programmability.

4 | Implementation and Design

This work aims to fundamentally redesign the lowering process for convolution and matrix-vector multiplication operations, with the dual goals of fully leveraging the capabilities of ReRAM architectures (as discussed in section 2.1) and achieving a compilation pipeline that is demonstrably faster, more efficient, and easier to maintain. To motivate and clarify the design choices behind the new lowering strategy, we first provide a concise overview of the convolution operation itself. This sets the stage for the subsequent implementation details. Additional discussion of related operations, such as matrix-vector multiplication, is provided in section B.1; however, the most significant improvements are put in place in the context of convolutions.

Convolutions are a fundamental operation in deep learning, particularly in convolutional neural networks (CNNs). They involve applying a number of filters (or kernels) to a generally four-dimensional input tensor, to produce an output tensor. The filter is smaller than the input tensor and slides over the input, performing element-wise multiplication and summation at each position, as illustrated in Figure 4.1.

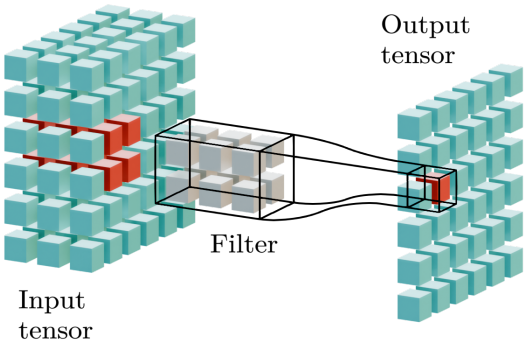

Figure 4.1: Visualization of a filter sliding over an input tensor. The filter is applied to each position of the input, performing element-wise multiplication and summation to produce the output tensor.

For each position of the filter, the resulting value is stored in the output tensor, resulting in a new tensor that captures the features of the input. This may result in a different number of channels, depending on the number of filters used. In figure 4.1, a single filter is shown for clarity (thus a single output channel is produced), but in practice, multiple filters are applied simultaneously, resulting in a multi-channel output tensor. The filter, input, and output shapes are considered to be known at compile time for our purposes.

Formally, a convolution operation can be expressed as equation 4.1, where the resulting value Y is the output tensor, value X is the input tensor and W is the filter (or kernel). Indices i and j represent the spatial dimensions of the output tensor (i.e., the pixel position), while k represents the depth (or channel) dimension. The kernel's spatial dimensions are represented by k_x and k_y, which correspond to the width and height of the filter, respectively. The overall convolution operation involves iterating over the input tensor's spatial coordinates and applying the filter at each position, which is here represented by the summation over k_x and k_y.

Simplifications Classical convolution operations also include padding and stride parameters. However, for the sake of simplicity, we will not include them in this discussion: their effects can easily be expressed by changing loop bounds and strides at the very last stage of the lowering process. Furthermore, the addition of bias terms is not considered in this work, as it can be expressed as a simple addition operation after a bias-less convolution operation is performed.

$$Y(i,j,k) = \sum_{k_x} \sum_{k_y} \sum_{c} X(i+k_x, j+k_y, c) \cdot W(k_x, k_y, c, k) \tag{4.1}$$

The intuition for our novel lowering process stems from the original strategy put in place by the RAPTOR compiler, which was to express the convolution operation as a collection of matrix-vector multiplications. This is a well-known technique that allows for more efficient computation of the resulting value, especially on hardware architectures that accelerate matrix operations. In order to clearly explain the process, we will provide a quick summary of the original approach for lowering convolutions to a set of matrix-vector multiplications, before presenting our new technique.

4.1. Groundwork for Lowering Convolutions

Clearly, a number of distinct approaches can be considered, depending on the way the input and weight data is grouped and partitioned across the architecture's resources. Re-

ferring to equation 4.1, and reducing the granularity along the channel axis, the following equality can be expressed:

$$\mathbf{y}(i,j) = \sum_{k_x} \sum_{k_y} \sum_{c} X(i+k_x, j+k_y, c) \cdot \mathbf{w}(k_x, k_y, c) \tag{4.2}$$

The weight tensor is now expressed as a vector (from now on identified by bold font), which is the result of concatenating all channels of the filter at a specific kernel position. The inner-most summation now represents a matrix-vector multiplication, since it's a sum of scalar-vector products. We can therefore assume that all $\mathbf{w}(k_x, k_y, c)$ vectors are columns of a matrix and that scalars $X(i+k_x, j+k_y, c)$ are the elements of a vector.

$$\mathbf{y}(i,j) = \sum_{k_x} \sum_{k_y} \left(\sum_{c} x_c(i,j,k_x,k_y) \cdot \mathbf{w}_c(k_x,k_y) \right) \tag{4.3}$$

This means that the convolution operation can be expressed as a series of matrix-vector multiplications, where each matrix depends on the kernel position, and each vector additionally depends on the input tensor's pixel position. The resulting value is then obtained by summing all the resulting vectors, which correspond to the different channels of the output tensor.

$$\mathbf{y}(i,j) = \sum_{k_x} \sum_{k_y} \mathbf{A}(k_x,k_y) \cdot \mathbf{x}(i,j,k_x,k_y) \tag{4.4}$$

Distributing all weights across as few crossbars (often described or referred to as simple matrices) as possible, and expressing the input as a series of one-dimensional vectors is the first stage of the original approach, of which a simplified representation is shown in Figures 4.2 and 4.3. The input tensor is sliced pixel-wise (a single slice is marked in red in Figure 4.2), meaning that, for each pixel position, a long vector is extracted from the input tensor. This vector contains all channels of the input tensor at that specific pixel position.

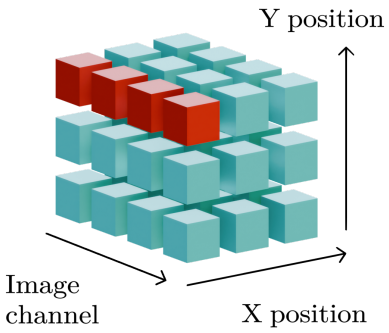

Figure 4.2: A four-channel, 3 × 3 input tensor, where a 4 channel-long slice is marked in red, extracted from the input tensor at pixel position (1, 1).

Additionally, all weights are grouped pixel-wise with respect to the filters, meaning that for each kernel position, a full matrix (illustrated as a grid of cyan squares in Figure 4.3) is assembled. Each filter contributes a single column to the matrix, and the number of rows is equal to the number of channels in the input tensor. Given this example, as many as 9 distinct matrices can be created, one for each kernel position.

Performing a matrix-vector multiplication between some input slice and weight matrix pair will produce a partial result for the corresponding output pixel, which must then be summed with the other 8 partial results to produce the final output value. Note that, although a single matrix-vector multiplication does not produce the final output value, its result spans the whole output tensor channel dimension, as the matrix contains information about all filters simultaneously.

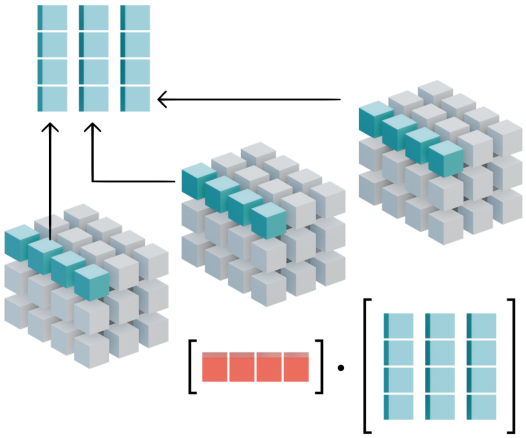

Figure 4.3: A set of three filters. The top-left kernel pixel is highlighted for each one. Their column-wise concatenation produces a matrix that can be multiplied by some extracted input slice, as shown in Figure 4.2. A total of 9 distinct matrices could be created, one for each kernel position.

All resulting matrix-vector multiplications have to be performed for each input tensor slice, and for each relevant weight slice. The input and weight slices are then multiplied together following a set of predefined rules which will be explained later, which ensure that concatenating all partial results will yield the correct output tensor. The result of each matrix-vector multiplication is a vector, which corresponds to some contribution for the corresponding output pixel.

4.2. Strategy

The new lowering process is based on the same principles as the original one, introducing a number of improvements that make it faster to compile and easier to maintain. The main idea is to group data in a way that allows for a more compact intermediate representation, while still implicitly expressing the overall computation as a series of matrix-vector multiplications.

Established CNNs usually perform convolutions on large, multi-channel input tensors. Hypothetically setting a 512 pixel-wide square image as the input tensor would imply creating roughly 200,000 distinct input slices, and individually sending them to the correct cores and crossbar combinations. Moreover, considering the relatively small size of the crossbars, additional slices may be needed to fit all channels of the input tensor. Expressing each operation clearly slows down the compilation process, and makes it difficult

to debug the generated code, which explicitly describes all matrix-vector multiplications that need to be performed, even if the overall structure contains redundant patterns.

Instead of slicing the input tensor pixel-wise, doing so in a way that allows to group multiple pixels together is a more efficient approach. We introduce a new `applyFilters` operation that reads a set of prepared crossbar matrices and an image comprising all pixels of the input tensor, in a single batch (albeit with a smaller number of channels). This operation then produces a portion of the output tensor, avoiding the need to express it as a group of pixel-wise slices, as shown in Figure 4.4. We will go more in-depth into the details of how these operands are created and how information flows between them in the next sections.

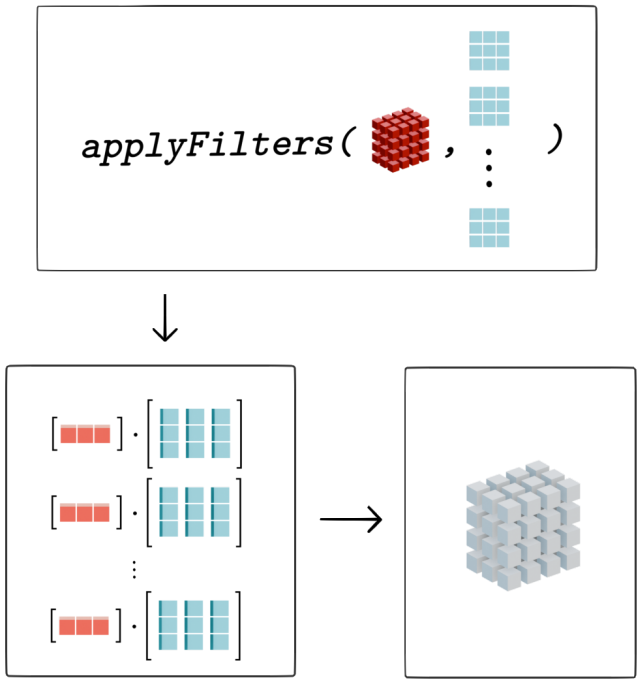

Figure 4.4: The `applyFilters` operation takes a channel-wise sliced input tensor (red) and a set of crossbar matrices (cyan) as input. The resulting portion of the output tensor is marked in white.

The `applyFilters` operation therefore acts as a black box, hiding the complexity of the underlying operations, while still allowing for a clear and concise representation of the general structure. This effectively leaves space to express how macro-slices are moved from one core to another, or how they're reduced and concatenated, as well as speeding up the compilation process. During the last phase of compilation, `applyFilters` operations are unwrapped, generating machine code in a fraction of the time.

4.3. Coarse Tensor Slicing

With respect to the original implementation, slices are gathered channel-wise, instead of pixel-wise; since the crossbar size might not be large enough to fit all input and output tensors' channels, these tensors must be nevertheless split into different groups. This means that multiple `applyFilters` operations might be needed for a single convolution operation, depending on the number of channels in the input and output tensors, and the size of the crossbars.

4.3.1. Input and Output

The input and output tensors are sliced channel-wise, keeping all slices as wide as the crossbar size, which is set to 4 in this particular example. Individual slices are marked by white or cyan colors, while empty black boxes represent missing tensor values whose contents are mapped to zero in the extracted vector. Extracting a vector from the input tensor is restricted to a single tile, which is a 4-wide channel slice (marked in red in Figure 4.5).

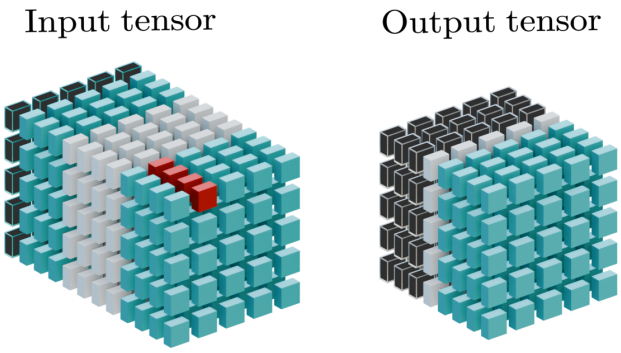

Figure 4.5: An 11-channel input tensor, and a 5-channel output tensor, where slices are extracted in 4-wide channel tiles. An example vector that could be extracted from one of these tiles is marked as red.

4.3.2. Weights

Weights are distributed across crossbars similarly to the original approach. In this example, we analyze a 3 × 3 kernel, with 8 channels and 4 filters. Fixing the crossbar size to 3, we cannot proceed as before, since this would lead to a 4 × 8 matrix: too large to fit within a single crossbar. As we show in Figure 4.6, simply slicing across the channel dimension may not be enough, as the number of filters might be greater than the crossbar size; the first three filters are considered, and the last one is postponed to the next crossbar group.

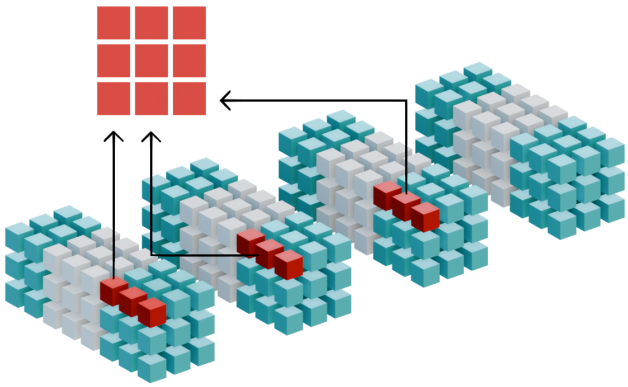

Figure 4.6: A set of four filters, where three distinct channel tiles are highlighted by alternating cyan and white coloring. At most three filter-wise slices can be extracted from each group, since the crossbar size is 3.

We can add a slicing dimension across the filters, ensuring that all resulting matrices are small enough to fit in the crossbar. In Figure 4.7, 6 distinct tiles are created, each of which is sliced to be expressed as a set of 9 matrices, one for each kernel position. These matrices have slightly different meanings with respect to the simplified version shown in equation 4.4, as their resulting vector depends on a specific tile.

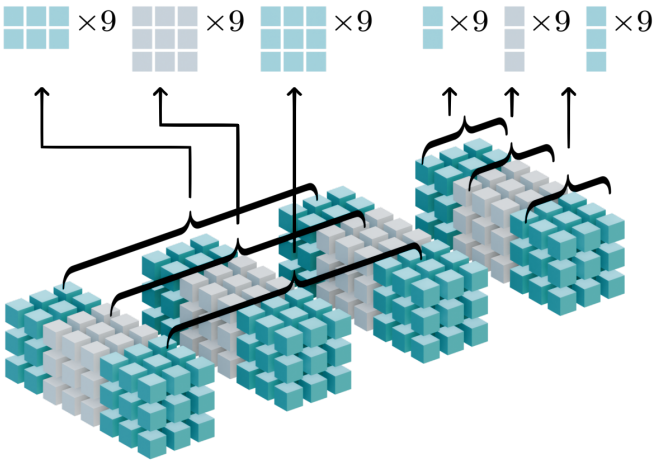

Figure 4.7: Slicing a $8 \times 4 \times 3 \times 3$ kernel across the channel and filter dimensions yields six distinct groups, each containing nine crossbar matrices.

4.4. Reduction and Concatenation

Unless the crossbar size is greater or equal to the number of channels in the output tensor, gathering the entire output tensor by simply summing the results of the matrix-vector multiplications is not possible. This is because the multiple matrix groups (as seen in Figure 4.7) refer to their respective output tile, and only account for their respective input tile as well. Resulting vectors belonging to the same input tile must be reduced, and resulting vectors belonging to different output tiles must be concatenated. `applyFilters` operations must then pass data within each other while keeping the data flow consistent with the original convolution operation. At this stage, it's also important to note that not only each crossbar has a limited amount of rows and columns, but also that each physical core contains a limited number of crossbars. This introduces an additional constraint on how the crossbars can be grouped together, as they must fit within the available resources of each core.

Expressing the relationships between crossbars and tiles in a table allows us to group them accordingly in the next phase of the compilation process. It's crucial to categorize these crossbars, as their results' processing will depend on the input and output tiles they belong to.

Crossbar Dependency Table

	Output Tile 1	Output Tile 2
Input Tile 1	 ×9	 ×9
Input Tile 2	 ×9	×9
Input Tile 3	 ×9	 ×9

Table 4.1: All crossbar groups' relationships with the input and output tiles, using matrices from Figure 4.7.

This categorization is also essential to minimize unnecessary inter-core communication, which would otherwise slow down the compilation process. In the example in Figure 4.8, a maximum amount of 5 crossbars is assigned to each core, so the six groups of nine crossbars shown in table 4.1 must be split across 11 cores. To minimize inter-core communication, crossbars belonging to the same input/output tile group are grouped together, and assigned to the same core whenever possible.

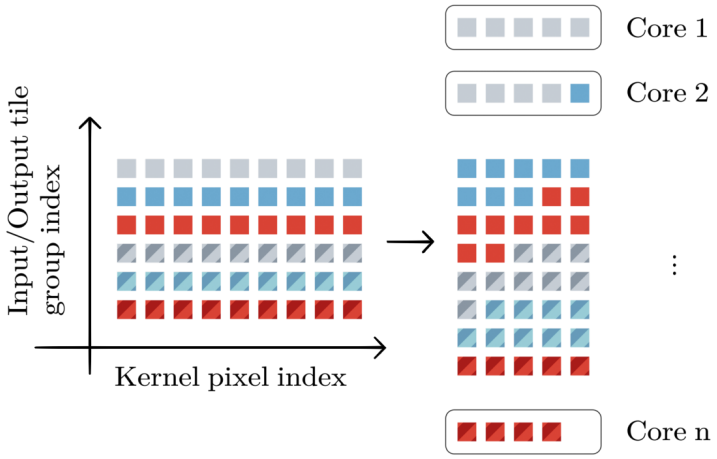

Figure 4.8: Groups belonging to the same input and output tiles are kept together, splitting across different cores if necessary.

After having subdivided all crossbars across the physical cores (represented as rounded boxes in Figure 4.9), we can feed them to the `applyFilters` operations (sharp boxes in Figure 4.9). Because a given `ApplyFilters` operation only produces output relative to a specific output tile, a single core may contain more than one such operation, depending on the crossbar's grouping tags. Appropriate reduction and concatenation operations are then added to account for the different input and output tiles, following the previously discussed relationships. Crucially, a single `applyFilters` operation does not need to take any relationship into account within itself, as all its inputs are already taken care of by the rest of the pipeline. This means that the `applyFilters` operation can be agnostically treated during code generation, without the need to know where its inputs originate or where its outputs belong. A simplified, commentated resulting spatial IR is shown in listing 4.1, which follows the same structure described by Figure 4.9.

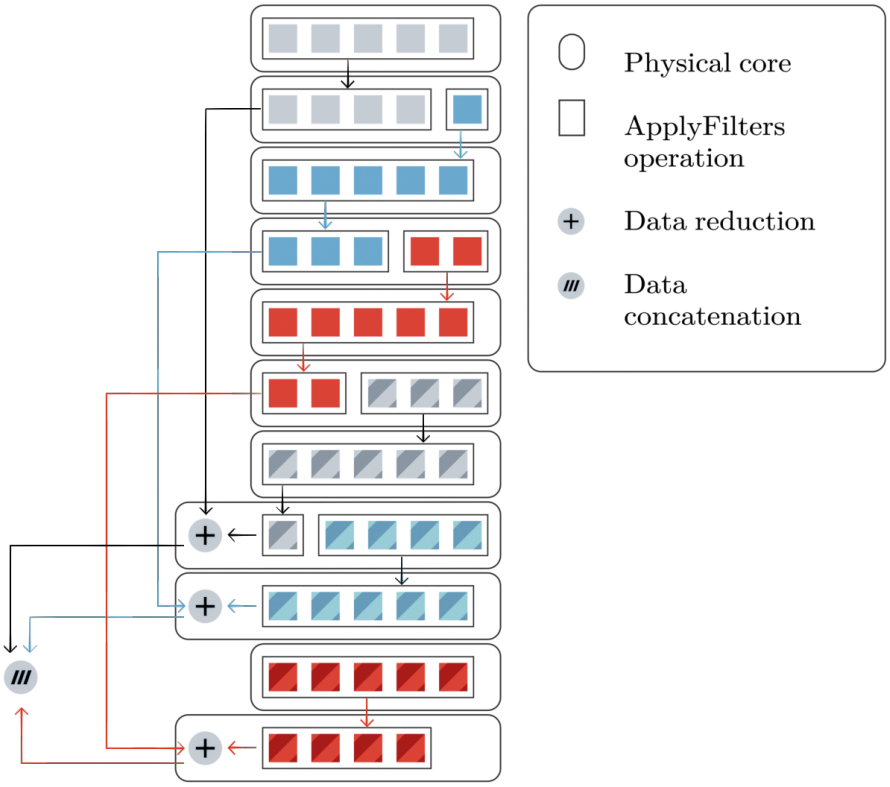

Figure 4.9: The first stage of the compilation process is a pipeline, where separate `ApplyFilters` operations' results cascade into the next stage, before being reduced and concatenated.

4.5. Convolutions Within the Spatial IR

In this section, we provide an example of how the convolution-lowering process is expressed within the `Spatial` dialect, which only holds very general assumptions about the target architecture, without explicitly describing data loading and core-to-core communication. The example shows a simplified version of the resulting spatial IR for a single convolution operation, where the input tensor is sliced into smaller tiles, and the weights are distributed across multiple crossbars.

Note that all cores, described by the `spat.compute` operations, consider two distinct sets of arguments, only one of which is explicitly used by the `applyFilters` operation.

- **Weights**: Within square brackets, the crossbar weights are specified. These are two-dimensional tensors that will be used by the `applyFilters` operation to perform the convolution. However, they are not directly passed to the operation itself but rather referred to as indices. The overall assumption is that no weight is dynamically

loaded through the architecture's instruction set, but rather that all weights are already present within the crossbars before execution, and that the `applyFilters` operation simply refers to them by their index. This implies that all weight tensors and their contents will be discarded at a later stage of the compilation process, and that the `applyFilters` operation will be unwrapped, generating machine code that directly refers to the crossbar weights. However, this crucial optimization leads to a significant limitation for validation purposes, as recovering the true weight values is not part of the compilation process while being a fundamental requirement to check the correctness of the produced output.

- **Input**: Round brackets are used to specify the input tensor slices. This is the set of values that will be passed to the `applyFilters` operation, and are used to perform the convolution. Input slices may be directly gathered from the input tensor, or they may be passed as arguments from previous `spat.compute` operations. Both cases are shown in listing 4.1, accordingly commented. When multiple network layers that take advantage of the `ApplyFilters` operations are connected in series, each `ApplyFilters` operation's output is already shaped in such a way as to match all crossbar size requirements, so no further slicing is expected to be performed: most `ApplyFilters` operations in the pipeline directly feed their output to another `ApplyFilters` operation, which may perform its computation on a separate physical core.

```
module attributes { ... } {
func.func @main(%arg0: tensor<1x4x4x4xf32> { ... }) -> tensor<1x8x4x4xf32> {
// The weights are stored in a constant tensor.
%0 = onnx.Constant dense_resource< ... > : tensor<8x4x3x3xf32>

// Which is then sliced into 54 distinct crossbar matrices.
%slice = tensor.extract_slice %0 [...]
%slice_0 = tensor.extract_slice %0 [...]

... < 50 more slices> ...

%slice_51 = tensor.extract_slice %0 [...]
%slice_52 = tensor.extract_slice %0 [...]

// The input tensor is sliced on the fly if necessary, and passed to the
// compute, which contains an applyFilters operation. In square brackets, the
// needed crossbar weights are specified.
%slice_53 = tensor.extract_slice %arg0 [...]
%1 = spat.compute[%slice ... %slice_3] (%slice_53) {
   ^bb0( ... ):
   %12 = spat.apply_filters %arg1 {weightIndices = [0, 1, 2, 3, 4]}
   spat.yield %12
}
```

```
24
25  // This compute performs reduction with the previous compute's output,
26  // and two applyFilters operations, for two distinct input tiles.
27  %2:2 = spat.compute[%slice_4 ... %slice_8] (%slice_53, %1) {
28     ^bb0( ... ):
29     %12 = spat.apply_filters %arg1 {weightIndices = [0, 1, 2, 3]}
30     %13 = spat.vadd %arg2, %12
31     %14 = spat.apply_filters %arg1 {weightIndices = [4]}
32     spat.yield %13, %14
33  }
34
35  // ... <5 more compute operations, elided for brevity> ...
36
37  // The first output of this particular compute will be reduced, as
38  // no more compatible input/output tile groups are present.
39  %8:2 = spat.compute[%slice_34 ... %slice_38] (%slice_54, %7, %4#0) {
40     ^bb0( ... ):
41     %12 = spat.apply_filters %arg1 {weightIndices = [0]}
42     %13 = spat.vadd %arg2, %12
43     %14 = spat.apply_filters %arg1 {weightIndices = [1, 2, 3, 4]}
44     %15 = spat.vadd %arg3, %14
45     spat.yield %13, %15
46  }
47
48  %9 = spat.compute[%slice_39 ... %slice_43] (%slice_54, %8#1) {
49     ^bb0( ... ):
50     %12 = spat.apply_filters %arg1 {weightIndices = [0, 1, 2, 3, 4]}
51     %13 = spat.vadd %arg2, %12
52     spat.yield %13
53  }
54
55  %10 = spat.compute[%slice_44 ... %slice_48] (%slice_54, %6#0) {
56     ^bb0( ... ):
57     %12 = spat.apply_filters %arg1 {weightIndices = [0, 1, 2, 3, 4]}
58     %13 = spat.vadd %arg2, %12
59     spat.yield %13
60  }
61
62  %11 = spat.compute[%slice_49 ... %slice_52] (%slice_54, %10) {
63     ^bb0( ... ):
64     %12 = spat.apply_filters %arg1 {weightIndices = [0, 1, 2, 3]}
65     %13 = spat.vadd %arg2, %12
66     spat.yield %13
67  }
68
69  // A concatenation operation is added to the end of the pipeline, to gather all
70  // relevant output tiles together, and form the final output tensor.
71  %concat = tensor.concat dim(1) %8#0, %9, %11 -> tensor<1x8x4x4xf32>
72  return %concat : tensor<1x8x4x4xf32>
73  }
74 }
```

Listing 4.1: Simplified spatial IR for a $8 \times 4 \times 3 \times 3$ kernel convolution, as shown in previous examples.

Additionally, this process expresses the overall computation as a pipeline, since no dataflow returns to any previous stage. Furthermore, each stage in this pipeline contains a modestly uniform amount of operations. This general structure could be further optimized in the future, in order to exploit larger batch sizes and perform distinct inferences in parallel. This would allow for a more efficient use of the crossbars, as multiple input tensors could be processed simultaneously, further improving the overall performance of the system.

4.6. PIM Dialect and Bufferization

Most additions to the RAPTOR compiler have been implemented in the `Spatial` dialect, an intermediate representation of operations in a ReRAM architecture described in section 2.3. In order to complete the compilation process, the `Spatial` dialect (and the data exchange described in section 4.4) must be expressed in a way that is compatible with the target architecture. This includes data load and store operations and core-to-core communication.

Memory & Communication As shown in listing 4.2, all core-to-core communication is expressed as a series of `channel_new`, `channel_send` and `channel_receive` operations. These operations are used to create new channels, send data to other cores, and receive data from other cores, respectively. Excluding non-runtime data (such as crossbar weights, that are assumed to be already loaded), all data is gathered on a `tensor.empty` operation, which creates a new tensor in local memory. This tensor is then used to store data loaded from the global memory, using the `pim.load` operation.

```
%7 = spat.channel_new
%8 = spat.channel_new
pim.core(%extracted_slice_14 ... %extracted_slice_18) {coreId = 3 : i32} {
    // The input data is no longer part of the core arguments,
    // but rather loaded from the main memory on an empty tensor.
    %15 = tensor.empty() : tensor<1x3x4x4xf32>
    %16 = pim.load(%1, %15) {globalSrcOffset = 0 : i32, size = 48 : i32}

    %17 = spat.apply_filters %16 {weightIndices = [0, 1, 2]}
    %18 = spat.channel_receive %6  // Receive some other core's result.
    %19 = spat.vadd %18, %17
    spat.channel_send %19 to %7
    %20 = spat.apply_filters %16 {weightIndices = [3, 4]}

    // Some partial results are sent to the next core, through
    // a channel. PIMSim "send" and "receive" instructions are
    // implemented during the last stage of the compilation process.
    spat.channel_send %20 to %8
    pim.halt
```

```
}

%9 = spat.channel_new
pim.core(%extracted_slice_19 ... %extracted_slice_23) {coreId = 4 : i32} {
    %15 = tensor.empty() : tensor<1x3x4x4xf32>
    %16 = pim.load(%1, %15) {globalSrcOffset = 0 : i32, size = 48 : i32}
    %17 = spat.apply_filters %16 {weightIndices = [0, 1, 2, 3, 4]}

    // The result of the previous core is received, and added to
    // the current core's result.
    %18 = spat.channel_receive %8
    %19 = spat.vadd %18, %17
    spat.channel_send %19 to %9
    pim.halt
}
```

Listing 4.2: Simplified spatial IR for a $8 \times 4 \times 3 \times 3$ kernel convolution, as shown in previous examples.

Whenever structural or communication changes must be made (such as implementing a distinct compiler for a different architecture), changing this part of the code, and the following stages is sufficient to adapt the generated code to the new architecture, without having to modify the computations expressed in the `Spatial` dialect. However, before concluding the compilation procedure to the very last, a bufferization pass is performed, which converts all tensor types into direct memory references.

Bufferization Bufferization is the process of converting higher-level tensor types into low-level memory references, which can be directly accessed by the target architecture (PIMSim explicitly requires a precise memory address to load and store data in registers, before performing any computation). This is done by replacing all tensor operations with a corresponding memory reference operation, and adding additional memory buffers if needed. Because the `applyFilters` operation is a higher-level operation that requires the manipulation of larger amounts of data, an additional memory address range is added as a special temporary buffer; its use will be described in detail in the following sections.

```
pim.core(%54, %55, %56, %57) {coreId = 10 : i32} {
    %alloc_56 = memref.alloc() {alignment = 64 : i64}
    %70 = pim.load(%alloc_0, %alloc_56) {globalSrcOffset = 48 : i32, size = 16 : i32}
    %alloc_57 = memref.alloc() : memref<1x2x4x4xf32>
    %alloc_58 = memref.alloc() : memref<1x2x1x1xf32>

    %71 = pim.apply_filters(
        input = %70,
        outBuf = %alloc_57,
        accumBuf = %alloc_58
    ) {weightIndices = [0, 1, 2, 3]}
```

```
12
13      [ ... ]
14
15      pim.halt
16  }
```

Listing 4.3: Simplified spatial IR for a $8 \times 4 \times 3 \times 3$ kernel convolution, as shown in previous examples.

4.7. Instruction Generation

To generate the final machine code, the `ApplyFilters` operation is unwrapped, replacing it with a series of PimSIM ISA instructions, saved as a set of `.json` files, tagged by physical core ID. As this operation is higher-level, its conversion requires a number of careful considerations, which will be discussed in the following paragraphs.

Result Buffering As shown in Algorithm 4.1, the result of the `applyFilters` operation is stored in a temporary buffer (introduced in Listing 4.1), for each matrix-vector multiplication, as the PIMSim ISA [11] does not support accumulating results directly yielded by the crossbars. Although each `ApplyFilters` operation declares its own one-time-use buffer, which might waste memory resources, RAPTOR's already implemented buffer reuse strategy will eliminate redundant memory allocation when possible. This means that the same buffer may be reused for multiple `applyFilters` operations, as long as they are not executed in parallel, within the same physical core.

Algorithm 4.1 Skipless Crossbar Accumulation

1: **for** $x = 0$ **to** $input\_width - 1$ **do**
2: **for** $y = 0$ **to** $input\_height - 1$ **do**
3: **for** each $crossbar$ in $crossbars$ **do**
4: $buffer \leftarrow input[x, y] \times crossbar$
5: $output[x, y] \leftarrow output[x, y] + buffer$
6: **end for**
7: **end for**
8: **end for**

Weight Skipping During a classical convolution operation, the kernel is applied to each pixel of the input tensor, meaning that all pixels are used at least once within the computation. However, considering a single kernel position, its values are not necessarily actively used to perform operations with all pixels of the input tensor, depending on

the kernel's size and position. This is visualized in Figure 4.10, where the considered kernel position is marked in cyan on the right, and the pixels that are not affected by its value, and which do not require it during computation, are marked as white, on the left. Generating unnecessary instructions to perform operations on pixels that are not used to contribute to the final result is a waste of resources, and can be avoided by simply skipping them.

This is particularly important in ReRAM architectures, where the number of crossbars is limited. Furthermore, in our representation of the convolution operation, a single crossbar is a one-to-one mapping of a single kernel position, meaning that performing operations on pixels that are not used in the computation means wasting the entire crossbar's resources.

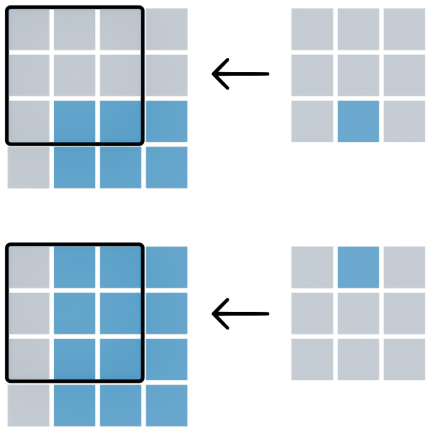

Figure 4.10: Input tensors (left) and filters (right) are marked depending on the specific kernel position (cyan) that is being processed. Some input pixels (white) are not used, as they are outside the kernel's range.

To address this, all `ApplyFilters` operations are linked to a set of attributes that specify the kernel's size and position, one for each crossbar matrix for which they were generated. During code generation (as shown in Algorithm 4.2), an additional check will be performed to verify whether the current pixel is within the valid kernel's range or not, and skip the computation accordingly. Checking whether the current pixel's coordinates are less than the kernel's size is a simple way to determine whether the pixel will be within the kernel's valid range.

Algorithm 4.2 Crossbar Accumulation

```
1: for x = 0 to input_width − 1 do
2:     for y = 0 to input_height − 1 do
3:         for each crossbar in crossbars do
4:             if x < ker_x[crossbar] or y < ker_y[crossbar] then
5:                 continue
6:             end if
7:             buffer ← input[x, y] × crossbar
8:             output[x, y] ← output[x, y] + buffer
9:         end for
10:    end for
11: end for
```

Only the pixels that are used in the computation are therefore processed, resulting in a smaller amount of required crossbars. Further results are collected in section 5.3.

5 | Validation System

A number of correctness and performance changes were introduced to the RAPTOR compiler, thanks in part to an accurate validation process. The original approach's aim was mainly focused on generating machine code for a number of sample networks, before performing energy and power evaluations through the use of the PIMSim simulator. Crucially, PIMSim was developed to be an architectural simulator, meaning that its main role is to assert how energy is consumed by the various system components, and not to perform each instruction's operation on simulated registers, or to otherwise produce any global or local memory state, that could be analyzed to assess the code's correctness.

5.1. Crossbar Content Pass

As mentioned in section 4.5, RAPTOR code generation considers all weights as being already accordingly loaded into the crossbars, and expects all input tensors to be already stored in such a way as to facilitate slicing and extraction (as an example, any input's pixel's channel values are expected to be contiguous). Moreover, this implies that the generated instructions do not express the precise location of data within the crossbars in any way. Instead, the code specifies the relevant crossbar's local index, while its contents are assumed to be already in place. Changing values within crossbars is assumed to be a slow process, which is not suitable for runtime operations, therefore the generated code is only responsible for managing the dataflow between cores, crossbar operations, and memory accesses.

However, in order to perform accurate validation, the original approach could not be sufficient, as no information about the contents of the crossbars was provided. This made it impossible to accurately validate the generated code against the expected output, as there was no way to assess what values were actually being used in the computations, or to perform any vector-matrix operations at all. Furthermore, the process of splitting the weights across multiple crossbars and cores was also non-validatable, as ensuring that any weight was correctly being distributed and accessed during the computations was crucial for the overall correctness of the system.

To perform accurate validation, we introduce a new mechanism within the RAPTOR compiler, that optionally exports a `.json` file for all cores, containing the precise location and value of all weights within crossbars, right before their deletion by the lowering pipeline. This way, the files can be manually inspected to verify that the weights are being correctly distributed, and that all generated code may automatically be validated by simulating the entire process, including exact memory accesses, and core-to-core communication, while relying on true values.

5.2. Validation Process

The validation process, illustrated in Figure 5.1, is implemented as a standalone Python program that orchestrates the full pipeline of testing the RAPTOR compiler's output. This process begins by randomly generating a set of input tensors. These inputs, along with a high-level description of a target operation, either a convolution or a general matrix-vector multiplication, are saved as an MLIR file. This file serves as the entry point for the compilation phase.

The RAPTOR compiler reads the MLIR file and transforms the input into hardware-specific representations. This includes producing low-level machine instructions and serializing the configuration of the underlying crossbar arrays. Both the instruction streams and the crossbar contents are exported as JSON files, which are then used as the input for the simulation phase of the validator.

Simulating core execution involves a detailed emulation of the architecture's behavior. Each core runs its instruction stream, accessing memory, performing computations, and communicating with other cores using the simulated Network-on-Chip (NoC). These operations model the full execution environment, including register reads/writes and memory hierarchies.

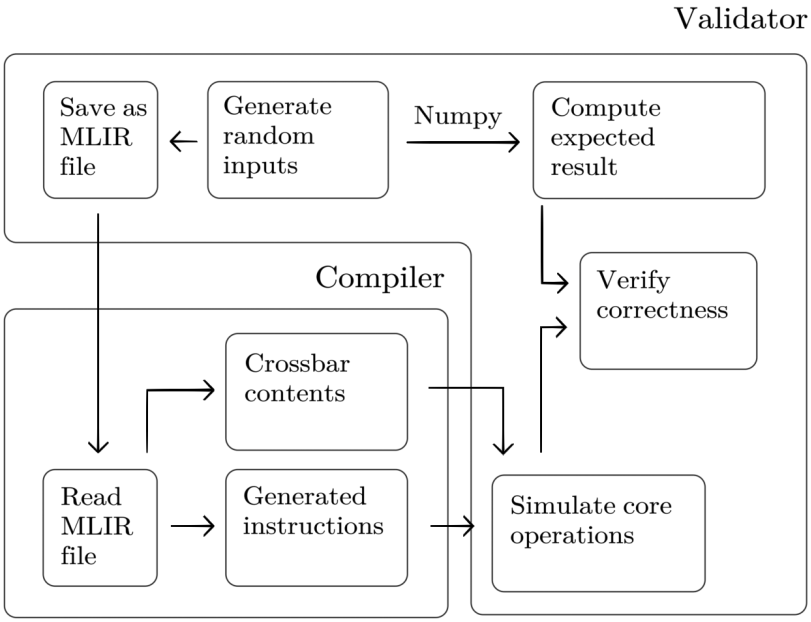

Figure 5.1: The sequence of steps performed by the validation process.

In parallel to the simulation, the same input tensors are used to compute the expected result using NumPy. This serves as a golden reference, relying on standard, well-tested numerical routines to evaluate the same operation described in the original MLIR file.

Finally, the output generated by the simulated core execution is compared to the expected NumPy result. Any mismatch is flagged, allowing the validator to detect miscompilations, instruction misinterpretations, or incorrect handling of data movement and synchronization.

Table 5.1 contains the list of all instructions that are currently supported by the validation process. The `applyFilters` operation is not included, as it is not a PimSIM ISA operation, but rather a higher-level aggregation of multiple instructions.

Supported Operations

	Arguments	Description
`sldi`	An immediate value, and the register it is stored in.	Load an immediate value into the register.
`ld`	The source and destination addresses, and the data size.	Load data from global memory into local memory.
`st`	The source and destination addresses, and the data size.	Store data from local memory into global memory.
`mvmul`	Input and output addresses for local memory, and an index representing the crossbar with respect to the core.	Perform a matrix-vector multiplication on a crossbar.
`vvadd`	Two input addresses, an output address for local memory, and a value representing the size of the vectors.	Perform a vector-vector addition on local memory.
`send`	An address for data to be sent, within local memory, its size, and the destination core index.	Send data to another core.
`recv`	An address for data to be received, within local memory, its size, and the source core index.	Receive data from another core.

Table 5.1: A list of supported operations for the validation process.

Sequential Instruction Execution Algorithm 5.1 validates a group of simulated cores by repeatedly cycling through them and executing one instruction per core at each step. The cores are initialized with their corresponding weights and instruction sets, then processed in a round-robin fashion until all have completed execution.

Algorithm 5.1 Validate Core Instructions

```
Load all cores from configuration:
for each core in config[array_group_map] do
    id ← extract core ID from core
    weights ← load_weights(networks_dir/core_id_weights.json)
    instructions ← load JSON from networks_dir/core_id.json
    Initialize Core with id, weights, instructions
end for
Create a cyclic iterator over all cores
current ← next core in cycle
while not all cores are done do
    if current is not done then
        current.run_instruction()
    end if
    current ← next core in cycle
end while
```

Although the algorithm runs sequentially, the core behavior, particularly around communication, ensures that the final state of each core's memory is consistent with parallel execution. This is because certain instructions, like send and receive, are blocking: when a core hits such an instruction, it marks itself as not done and waits for the communication to complete. This effectively prevents further progress until the necessary data transfer occurs, accurately modeling inter-core synchronization.

In this context, the validator does not aim for performance, but correctness. Its sequential nature is sufficient because it enforces the same dependencies and ordering constraints that would naturally arise in a real, parallel system. Therefore, even without true concurrency, the result faithfully represents what would happen if all cores were running simultaneously.

5.3. Results

In the new compilation pipeline, data is processed and considered at a higher level of granularity rather than at the pixel level. This means that operations such as send, receive, load, and store (which would originally need to handle data at the pixel level, resulting in a large number of instructions) are now performed at a coarser granularity. By grouping data and reducing the frequency of these operations, the number of instructions required for these tasks is significantly reduced.

Figure 5.2 illustrates this improvement by showing the normalized proportion of different instructions generated for a convolution operation. The new strategy demonstrates a noticeably lower number of send, receive, load, and store operations compared to the original implementation. This reduction is particularly beneficial because load and store operations are inherently slow due to their interaction with memory. By minimizing these operations, the overall speed of the system can be improved, as fewer memory-bound instructions are executed. This optimization highlights the efficiency of the new approach in reducing overhead and improving performance.

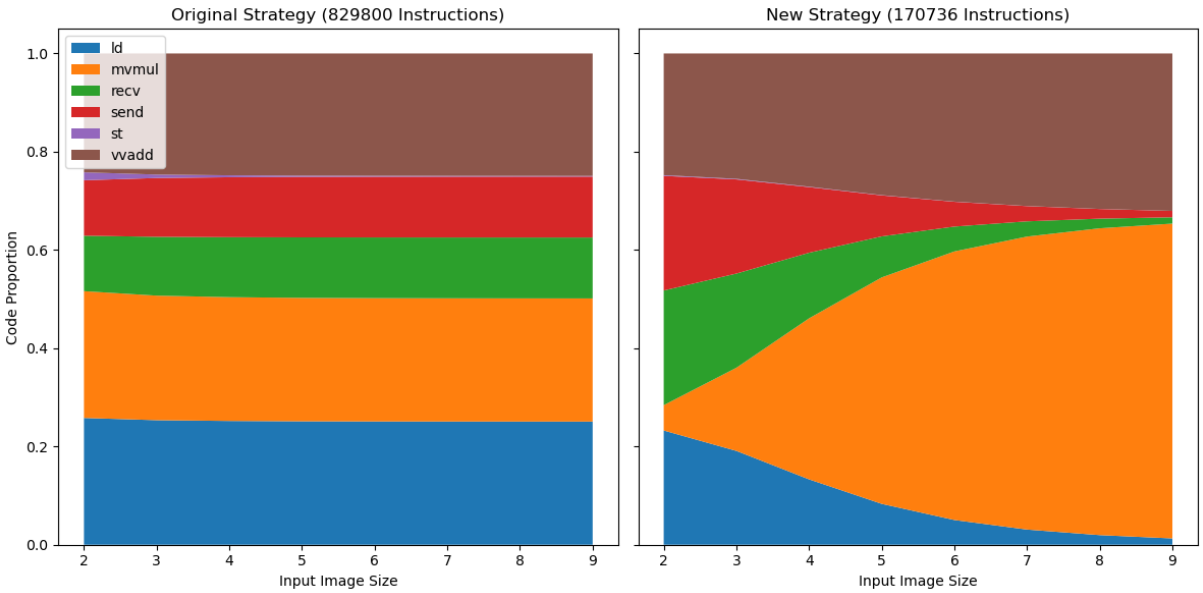

Figure 5.2: Normalized proportion of different instructions generated for a convolution operation, on the original and new implementations.

However, it is important to note that while the number of these operations has decreased, the size of the data chunks they handle has proportionally increased. This implies a tradeoff: fewer operations are needed, but each operation processes a larger amount of data. This tradeoff is acceptable in scenarios where the architecture can efficiently handle larger data transfers, but it may require careful consideration in systems with limited bandwidth or memory throughput.

Figure 5.2 further illustrates the distribution of instructions as the image size increases, with widths ranging from 2 to 9. These image sizes, while not representative of real-world scenarios, have been deliberately chosen for simplicity to better highlight the observed trends. The target crossbar size was intentionally set to a small value (three) to ensure that scenarios requiring core-to-core communication due to insufficient crossbar capacity are encountered. This configuration highlights how the new pipeline adapts to such

constraints, demonstrating the trade-offs between instruction count and communication overhead as the image size grows. The results highlight the critical importance of balancing crossbar size, data granularity, and communication requirements to achieve optimal performance.

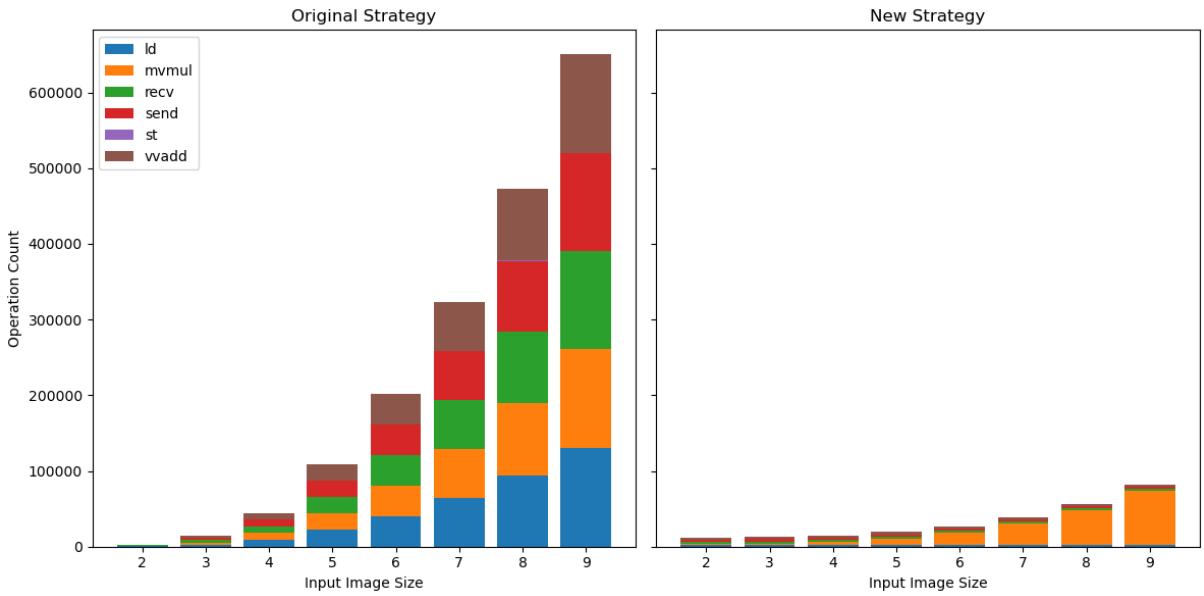

Figure 5.3: Absolute instruction counts for each operation as input image size increases, complementing the normalized view in Figure 5.2.

Note that in both Figure 5.2 and Figure 5.3, the `sldi` operation is omitted. This instruction is responsible for loading immediate values into registers and serves as a setup step for each other operation, since every operation requires its operands to be available in registers. As a result, the number of `sldi` instructions is roughly proportional to the total number of other operations, making its inclusion redundant for the purpose of these comparisons.

Figure 5.4 demonstrates that the new implementation significantly accelerates the compilation process, achieving speeds that are five to eight times faster than the original implementation, depending on the input image size. This improvement is primarily attributed to the reduced number of MLIR operations generated throughout the code. By increasing the granularity of the operations, the new approach avoids the need to explicitly express computations at a pixel-wise level, resulting in a more compact and efficient intermediate representation. This reduction in the number of operations not only speeds up the compilation process but also simplifies the overall structure of the generated code, making it easier to debug and maintain.

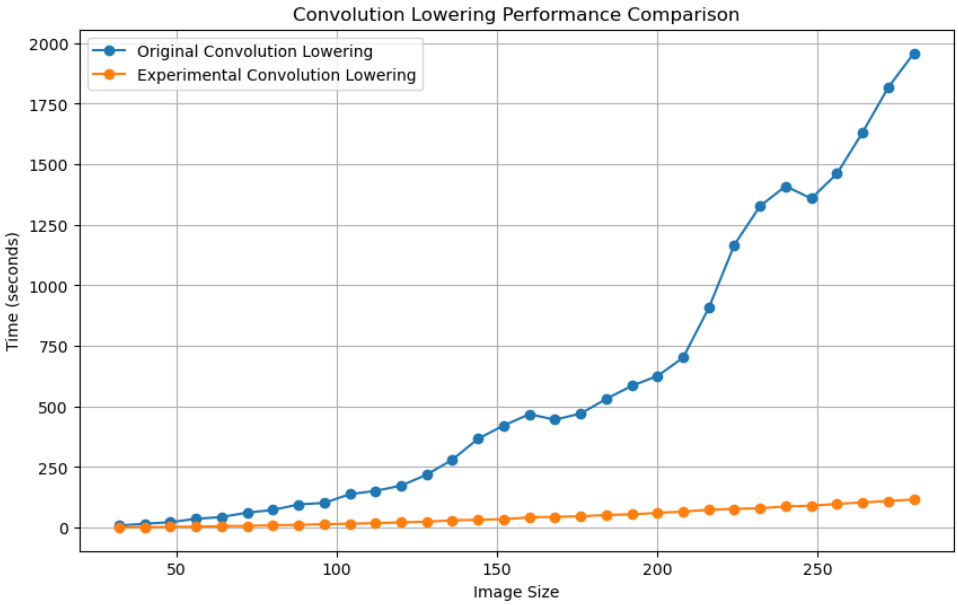

Figure 5.4: Compilation speed comparison between the original and new implementations, varying the input image size.

However, the positive results gathered for convolutions are not found for matrix-vector multiplications, as shown in Figure 5.5. Here, it seems that both the original and new implementations achieve nearly identical compilation speeds. This might be because the novel `applyFilters` operation has much less data to group together in the case of matrix-vector multiplications. As described in B.1, performing MVMs does not take advantage of the same data grouping strategy as convolutions, as the input tensor is a simple vector, and the kernel is a matrix. This means that the new implementation does not have the same opportunity to reduce the number of operations as it does with convolutions. The `applyFilters` operation is still used, but its impact on the overall compilation speed is almost negligible in this case.

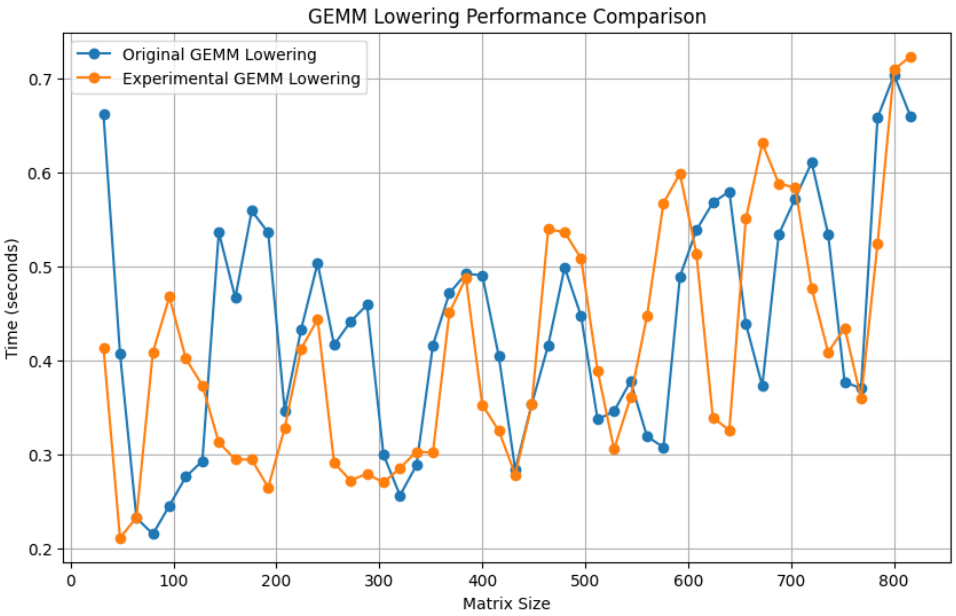

Figure 5.5: Compilation speed comparison between the original and new implementations of the `applyFilters` operation.

6 | Conclusion

This thesis has focused on improving the compilation of deep learning models for ReRAM architectures. By introducing the `applyFilters` operation and redesigning the convolution-lowering process, it has aimed to make the compilation process more modular, maintainable, and efficient. The following sections summarize the contributions, and outline the limitations and potential future work.

6.1. Contributions

Several key contributions were introduced in this work, improving the compilation process for ReRAM-based architectures. These contributions enhanced modularity, maintainability, and efficiency, while also expanding the range of supported networks. The main contributions are summarized as follows:

- **Introduction of the `applyFilters` Operation:** A novel `applyFilters` operation was introduced to streamline the compilation of convolutional operations and matrix-vector multiplications. This operation abstracts the complexity of distributing weights across crossbars and processing input tensors, allowing for a more modular and maintainable intermediate representation. By grouping data into channel-wise slices rather than pixel-wise slices, the `applyFilters` operation significantly reduces the size of the intermediate representation, leading to faster compilation times. Additionally, this operation is versatile enough to handle both convolutional operations and general matrix-vector multiplications, making it a cornerstone of the redesigned compilation pipeline.
- **Faster Compilation of Convolutions:** The redesigned convolution lowering process, centered around the `applyFilters` operation, has led to a measurable improvement in compilation speed. By avoiding the explicit expression of pixel-wise operations and instead leveraging higher-level abstractions, the new approach reduces the number of intermediate operations that need to be processed. This not only accelerates the compilation process but also simplifies debugging and enhances

the overall maintainability of the compiler.

- **Validation System for Machine Code:** A comprehensive validation system was developed to ensure the correctness of the generated machine code. This system includes a mechanism to export the precise contents of crossbars, allowing for accurate simulation of the generated code. By comparing the simulated output with a golden reference computed using NumPy, the validation system can detect and flag any discrepancies, ensuring the reliability of the compilation process. The validation system also supports a wide range of operations, including core-to-core communication and memory accesses, making it a robust tool for verifying the correctness of the entire compilation pipeline.
- **Support for ResNet Architectures:** Independently of the new additions, support for ResNet architectures, specifically ResNet-18, was added to the compiler. This required addressing challenges such as handling residual connections and implementing operations like global average pooling. The `ONNXReduceMeanV13Op` was lowered to a specialized `ONNXAveragePoolOp`, and a new `vsDivOp` instruction was introduced to perform vector-scalar division during the CodeGen phase. These contributions, detailed in Appendix B, expand the range of networks that can be compiled using the RAPTOR framework, making it more versatile and applicable to real-world deep learning workloads.

These contributions collectively enhance the RAPTOR compiler's ability to efficiently and correctly compile deep learning models for ReRAM architectures, paving the way for further optimizations and extensions in this domain.

6.2. Limitations

The introduction of the `applyFilters` operation brings significant improvements in terms of modularity and efficiency. However, it also introduces a fundamental incompatibility with the operations from the previous implementation. This incompatibility arises from the way data is split and organized in memory, which differs substantially between the two approaches.

6.2.1. Data Splitting and Memory Layout

In the previous implementation, tensors were split pixel-wise. Each pixel, along with all its channels, was treated as a discrete unit of computation. This approach allowed operations like max-pooling to analyze each pixel individually, as all the channel values

for a single pixel were stored contiguously in memory. Figure 6.1 illustrates this memory layout, where each pixel's channel values are stored sequentially, making it straightforward to access and process them.

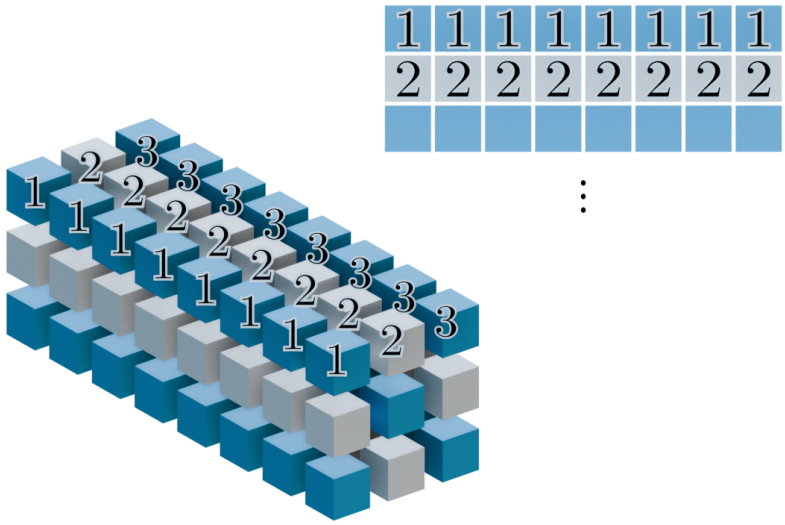

Figure 6.1: Pixel-wise memory layout: Each pixel's channel values are stored contiguously, enabling operations like max-pooling to process individual pixels efficiently.

In contrast, the new implementation splits tensors into channel-wise chunks, where each chunk contains all pixels for a subset of the channels. This approach is more efficient for operations like convolutions, as it allows the `applyFilters` operation to process entire channel chunks in a single step. However, it fundamentally changes the memory layout, as shown in Figure 6.2. Here, the memory is organized such that all pixels of the first channel chunk are stored sequentially, followed by all pixels of the second channel chunk, and so on. Within each channel chunk, pixels are stored in a row-major order.

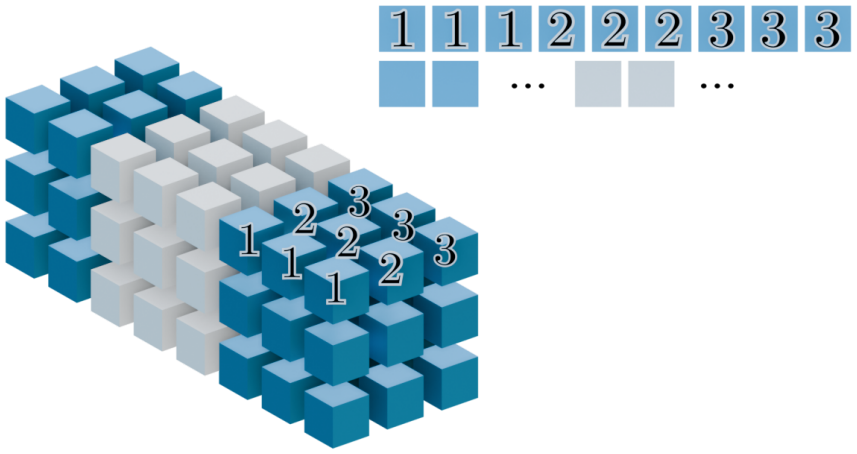

Figure 6.2: Channel-wise memory layout: Pixels are grouped by channel chunks, with all pixels of the first chunk stored sequentially, followed by the second chunk, and so on.

6.2.2. Implications for Compatibility

The new memory layout introduces a critical limitation: operations from the previous implementation, which relied on pixel-wise data access, cannot be directly mixed with the new channel-wise layout. For example, a max-pooling operation in the old implementation would iterate over each pixel, accessing all its channel values directly. In the new layout, performing such an operation would require adapting the operation to the channel-wise structure, as the channel values for a single pixel are distributed across different channel chunks.

To exemplify this, we may consider a max-pooling operation that needs to analyze a 2×2 region of the input tensor. In the previous implementation, this operation could directly access the channel values for each pixel in the region, as they were stored contiguously. In the new implementation, the operation must be adapted to work with the channel-wise layout. While this requires rethinking the data access strategy, it is still possible to implement an efficient max-pooling operation by processing each channel chunk independently and combining the results.

While the `applyFilters` operation offers significant advantages for convolutional operations, its channel-wise data splitting strategy introduces challenges for adapting other operations to the new memory layout. This work does not address these challenges, as the focus has been on improving convolutional operations. Future work may explore efficient implementations of other operations, such as max-pooling, within this new framework.

Bibliography

[1] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W. mei Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference, 2019. URL `https://arxiv.org/abs/1901.10351`.

[2] Y. Chen. Reram: History, status, and future. *IEEE Transactions on Electron Devices*, 67(4):1420–1433, 2020. doi: 10.1109/TED.2019.2961505.

[3] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016. doi: 10.1109/ISCA.2016.13.

[4] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.

[5] E. W. Lim and R. Ismail. Conduction mechanism of valence change resistive switching memory: A survey. *Electronics*, 4(3):586–613, 2015. ISSN 2079-9292. doi: 10.3390/electronics4030586. URL `https://www.mdpi.com/2079-9292/4/3/586`.

[6] MLIR Contributors. Mlir bufferization - destination-passing style. `https://mlir.llvm.org/docs/Bufferization/#destination-passing-style`, 2025. Accessed: 2025-05-15.

[7] ONNX Contributors. ONNX: Open neural network exchange. `https://github.com/onnx/onnx`, 2024.

[8] S. Qu, S. Zhao, B. Li, Y. He, X. Cai, L. Zhang, and Y. Wang. Cim-mlc: A multi-level compilation stack for computing-in-memory accelerators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages*

and Operating Systems, Volume 2, ASPLOS '24, page 185–200. ACM, Apr. 2024. doi: 10.1145/3620665.3640359. URL http://dx.doi.org/10.1145/3620665.3640359.

[9] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. Isaac: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 14–26. IEEE Press, 2016. ISBN 9781467389471. doi: 10.1109/ISCA.2016.12. URL https://doi.org/10.1109/ISCA.2016.12.

[10] A. SOMAINI. A novel mlir-based compilation flow for accelerating convolutional neural networks on process in memory architectures. 2024.

[11] X. Sun, X. Wang, W. Li, L. Wang, Y. Han, and X. Chen. Pimcomp: A universal compilation framework for crossbar-based pim dnn accelerators, 2023. URL https://arxiv.org/abs/2307.01475.

[12] X. Sun, X. Wang, W. Li, Y. Han, and X. Chen. Pimcomp: An end-to-end dnn compiler for processing-in-memory accelerators, 2024. URL https://arxiv.org/abs/2411.09159.

A | Appendix A

The details for a separate contribution to the RAPTOR compiler, which is not directly related to the work presented in this thesis, is described in this appendix, so as not to hinder the flow of the main text. This contribution is a small, separate set of additions, performed as a parallel effort, unrelated to the novel operations and dataflow.

The original RAPTOR compiler was evaluated through a set of sample networks. Image classification is RAPTOR's main use case, as its large amount of data and operations are well suited for the ReRAM architecture. Particularly, two networks were used to evaluate the compiler's performance: `GoogLeNet` and `VGG-16`.

- `GoogLeNet` is a deep convolutional neural network, which was chosen for its set of multiple convolutions performed in parallel over the same input tensor. These results are then concatenated together, and passed to the next layer. This is a particularly interesting case for the ReRAM architecture, as testing parallel, high-bandwidth operations is crucial to assess the compiler's performance. `GoogLeNet` is composed of 22 layers, and has a total of 6 million parameters.
- `VGG-16` was therefore chosen as a second network, as it can be considered a more traditional convolutional neural network, with a sequential structure, where each layer's output is passed to the next layer. This is a particularly interesting case for the ReRAM architecture, as testing sequential operations is crucial to assess the compiler's performance. `VGG-16` is composed of 16 layers, and has a total of 138 million parameters.

More detailed analysis of these network's descriptions and simulations are available in the original RAPTOR paper [10], and in the RAPTOR GitHub repository.

A.1. ResNet-18

ResNet-18 presents a number of convolutional layers organized into a deep network that incorporates residual learning to improve both training efficiency and model performance. It consists of 18 layers with learnable parameters, including convolutional layers, batch

normalization, and ReLU activations, grouped into a series of residual blocks. Each block includes a shortcut, or skip connection, which allows the input to bypass one or more layers and be added directly to the output. This structure helps alleviate the vanishing gradient problem common in deep networks and allows the model to train more effectively by focusing on learning residual functions.

As shown in Figure A.1, ResNet-18 starts with an initial convolution and max-pooling layer, followed by four stages of residual blocks. Each stage increases the number of channels while reducing spatial resolution, enabling the model to learn increasingly abstract features. The use of residual blocks allows the network to maintain performance despite its depth, and its modular, repetitive structure makes it a good benchmark for analyzing architectural behavior.

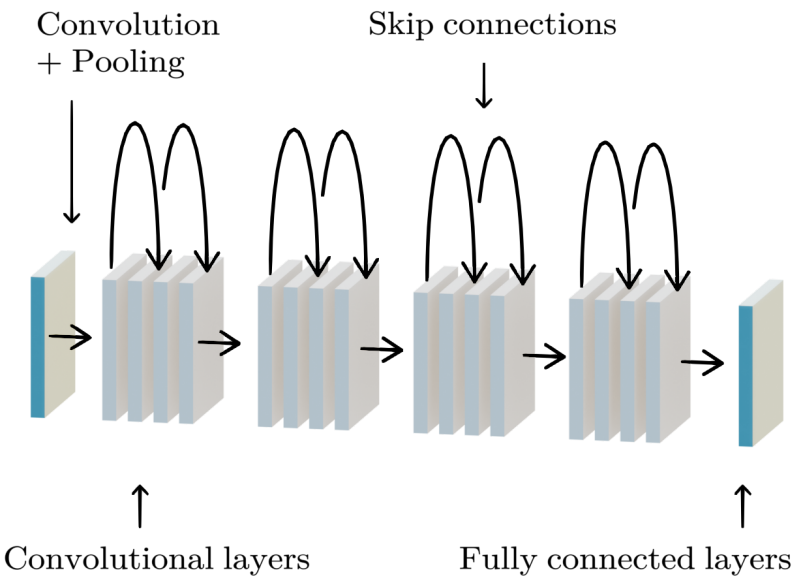

Figure A.1: ResNet-18 architecture, showing the arrangement of convolutional layers and residual blocks.

For testing ReRAM-based compilers, ResNet-18 is a particularly suitable choice. It includes a diverse set of common operations: convolutions, nonlinear activations, and shortcut additions, that appear frequently in modern deep learning workloads. The presence of skip connections also introduces nontrivial dataflow that stress-tests the compiler's ability to manage memory access patterns, operation fusion, and data reuse, core issues in compiling for ReRAM.

Lastly, it is deep enough to be realistic and representative of real-world models, yet small enough to keep simulations manageable and results interpretable. This makes it ideal for evaluating the performance characteristics of a ReRAM-oriented compiler pipeline.

A.2. RAPTOR Compatibility

RAPTOR's compiler is able to handle these skip connections, and generate the appropriate code to perform the necessary computations, thanks to the underlying ONNX dialect, which already expresses these relationships through its graph-based structure. A number of contributions were put in place to allow for the correct generation of the necessary code, as some minor changes were needed to the original compiler's pipeline.

- `ONNXReduceMeanV13Op`: This operation is present in the original ONNX model and needs to be supported to run ResNet-18. Since the `Spatial` dialect does not directly support generic reduction operations, each `ONNXReduceMeanV13Op` is lowered to a specialized `ONNXAveragePoolOp` right after the ONNX-to-MLIR conversion. This works because global average pooling—used at the end of ResNet-18—is semantically equivalent to a spatial mean reduction, and average pooling is more naturally expressed in the Spatial dialect, which is built around tiling and applying filters to spatial data.
- `vsDivOp`: This is a custom instruction added to the CodeGen phase to target the PIMSim ISA. It performs element-wise division between a vector and a scalar, and is used to implement the division step that follows summation in a reduce-mean operation. Without this instruction, there would be no way to express the mean calculation required by the lowered average pool, so `vsDivOp` is essential for correctly simulating the effect of mean reduction on ReRAM-based architectures.

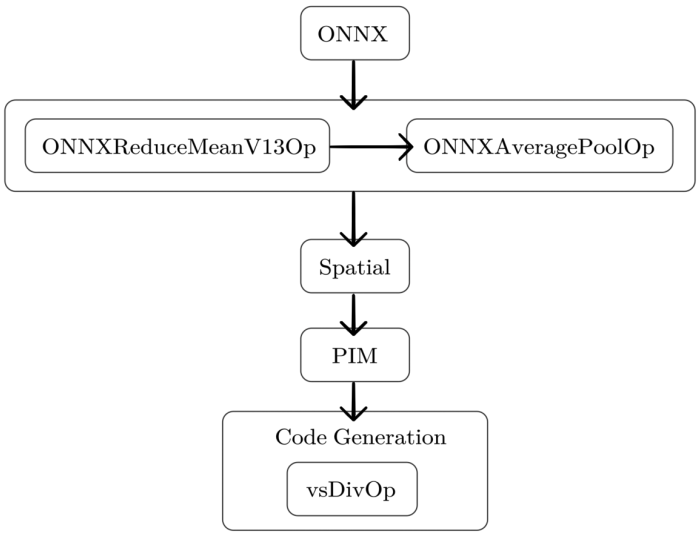

Figure A.2: Reduce mean operations are interpreted as specialized average pooling operations before being processed by the `Spatial` pass. During code generation, vector-scalar division operations must be performed.

A.3. Results

Figure A.3 compares the performance of the two ReRAM-targeted in-memory computing compilers: pimcomp and RAPTOR. The evaluation was conducted across a range of core counts, measuring throughput (samples per second) and energy consumption per sample (in μWh/sample).

The left-hand graph presents throughput. RAPTOR demonstrates a substantial advantage across all core counts, achieving throughput values that are consistently one to two orders of magnitude higher than pimcomp. For example, at 200 cores, RAPTOR reaches 2070 samples/s, while the best-performing pimcomp configuration (at 160 cores) peaks at only around 103 samples/s. This indicates a 20× performance gap in favor of RAPTOR under those conditions.

The right-hand graph shows energy efficiency. Here too, RAPTOR significantly outperforms pimcomp, consuming as little as 0.025 μWh/sample at 200 cores, compared to pimcomp's lowest of around 0.79 µWh/sample (at 150 cores). At higher core counts, pimcomp's energy consumption fluctuates and, in some cases, rises drastically — for instance, reaching over 300 µWh/sample at 225 cores under the GA=50 configuration. In contrast,

RAPTOR maintains low and gradually increasing energy usage as the number of cores increases, which suggests much better scalability, although further investigation may be needed to understand the exact cause of this behavior.

It's also notable that while pimcomp's performance appears sensitive to the number of genetic algorithm iterations, this tuning yields limited benefit. In several cases, increasing GA iterations correlates with a drop in throughput and a spike in energy cost, rather than any meaningful improvement.

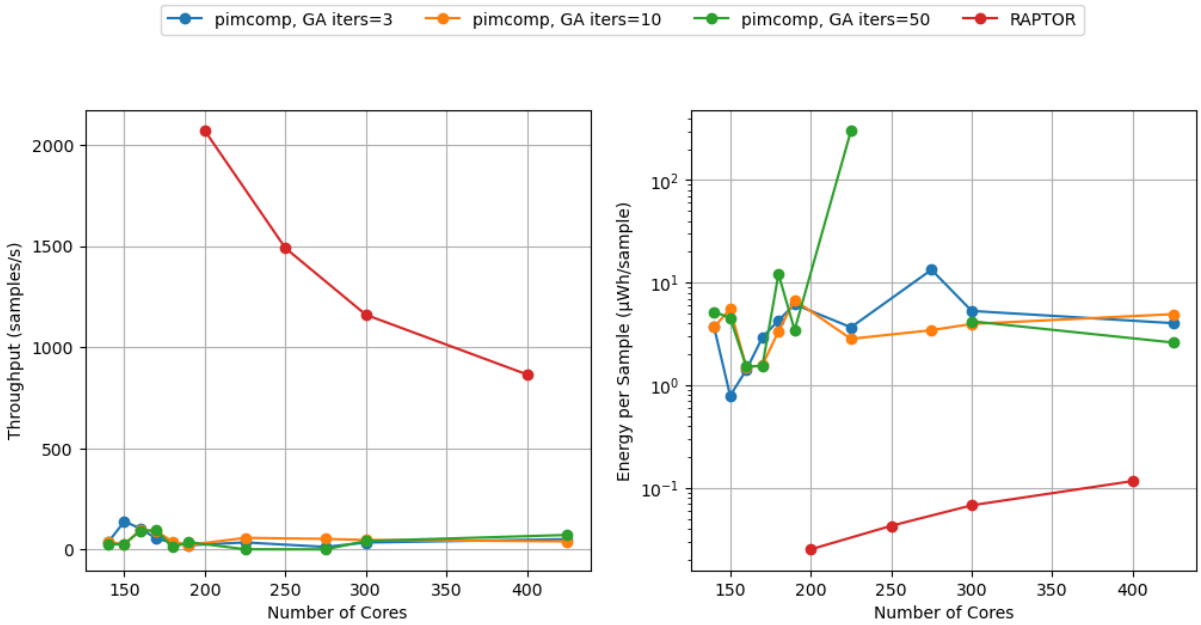

Figure A.3: Two graphs showing the results of the ResNet-18 model. The left graph details the throughput over the number of cores, while the right graph shows the energy consumption per sample over the number of cores.

These results were already confirmed in the original RAPTOR paper, and are now further validated by the addition of ResNet-18 support. The results are consistent across different core counts, indicating that RAPTOR's design and optimizations are effective in leveraging the ReRAM architecture for deep learning workloads.

B | Appendix B

Given a limited count of available free slots per crossbar, and a limited amount of crossbars per physical core, performing a large matrix-vector multiplication also requires a number of intermediate steps, to split the input tensor into smaller vectors, and to split the matrix to be multiplied into smaller sub-matrices. The novel `ApplyFilters` operation can be also exploited to process these operations.

B.1. General Matrix-Vector Multiplication

A matrix-vector multiplication can be expressed as a particular case of the general convolution operation, where the input tensor is shaped as a single vector, and the matrix to be multiplied is split column-wise, and expressed as an array of vector-shaped filters. This lets us exploit the previously discussed pipeline, without needing a separate process for this particular operation type. Using the same notation as before, we can set indices i and j to 0, and k to the input vector's index.

$$Y(0,0,k) = \sum_{k_x}\sum_{k_y}\sum_{c} X(k_x,k_y,c)\cdot W(k_x,k_y,c,k) = \sum_{r=0}^{R-1} \mathbf{x}\cdot\mathbf{w}^{(r)} \tag{B.1}$$

Where $\mathbf{w}^{(r)}$ is the r-th filter, and R is the number of filters. The result of this operation is a one-dimensional vector containing all $Y(0,0,k)$ values for all k.

$$\mathbf{y} = W^{\top}\mathbf{x} \tag{B.2}$$

Where $\mathbf{y}$ is a vector containing $Y(0,0,k)$ for all k, $\mathbf{x}$ is the input vector, and W is a matrix whose columns are the flattened filters $\mathbf{w}^{(k)}$.

List of Figures

List of Tables

Acknowledgements

I am deeply thankful to my advisor, Prof. Giovanni Agosta, for his helpful and direct advice and guidance. I also wish to thank Lev Denisov from the RAPTOR team, for his support and encouragement throughout this journey, and the original RAPTOR author, Andrea Somaini, for his helpful suggestions and insights during the early stages of this work.

I am grateful to my family for their constant support and encouragement. A special thanks goes to my parents, their love and support have been invaluable throughout this process. I mostly want to thank my aunt and uncle for their generosity in providing me with a place to stay during my studies, which allowed me to focus on my work without worrying about accommodation, as well as some of the best cooking a university student could ask for.

Lastly, I would like to thank my friends Francesco and Emma for, respectively, their insight and encouragement throughout my academic journey. Their friendship has been a source of strength and motivation, and I am grateful for their presence in my life.