POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Compiler techniques for Processing In Memory

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Claudia Cucuccio**

Student ID: 940428
Advisor: Prof. Giovanni Agosta
Academic Year: 2024-25

Abstract

As modern applications become more and more data-intensive and memory technology reaches the end of Dennard scaling, Processing in Memory emerges as a new architectural paradigm that reduces expensive data movement in computing systems. The deployment of PIM systems needs support from a robust software ecosystem, inside of which compilers take on the task of generating optimized code for the novel hardware configurations. This thesis discusses compiler solutions for PIM architectures, describing the approaches they adopt for facing these new, unique challenges, their best features and shortcomings, and possible future research directions.

Keywords: Processing in Memory, Processing Near Memory, Compiler techniques, PIM compilers, Offloading techniques

Abstract in lingua italiana

Nel contesto odierno, in cui le applicazioni diventano sempre più data-intensive e le tecnologie delle memorie scalano con sempre più difficoltà, il Processing in Memory emerge come nuovo paradigma architetturale per ridurre i costosi flussi di dati nei sistemi di calcolo. L'implementazione di sistemi PIM necessita del supporto di un solido ecosistema software, nel quale i compilatori hanno il compito di generare codice ottimizzato per le nuove configurazioni dell'hardware. Questa tesi tratta delle soluzioni compilative per le architetture PIM, descrivendo gli approcci adottati nell'affrontare queste nuove e singolari sfide, i loro punti di forza e debolezza, e possibili direzioni di ricerca future.

Parole chiave: Processing in Memory, Processing Near Memory, tecniche di compilazione, compilatori PIM, tecniche di offloading

Contents

Introduction

Processing in Memory, also known as Computing in Memory, is a new design paradigm that has been establishing itself in computer architectures to answer to the clash of two opposing trends. On one hand, the hardware manufacturing technology has hit the long forecast "memory wall": memory scaling is reaching its physical limits, and the gap between processor speed and memory bandwidth is steadily increasing. When applied to a traditional von Neumann architecture, this means that the latency of memory accesses has become the main bottleneck in computing performance. On the other hand, applications in fields like Machine Learning, graph computing and image processing are rapidly becoming more and more data-intensive, as their datasets grow in size. This contrast leads to the inevitable conclusion that traditional computing architectures are not a suitable fit anymore for this kind of applications. Processing in Memory addresses this issue by proposing a shift in how architectures are conceived: since separating computation and storage units results in high data movement overheads, they should be brought closer together. In particular, computation units can be integrated on the same chip as the memory (Processing Near Memory), or the memory itself can be leveraged to perform some computation (Processing Using Memory). Recent advancements in technology, like the advent of non-volatile memories and of 3D memory stacks, have allowed designers to propose concrete implementations of this paradigm. Like any computing system, PIM architectures need a software stack to be deployed in real-life scenarios; compilers in particular are crucial in enabling programmability. Given how these architectures are fundamentally different from traditional ones, they present their own unique challenges that require new techniques, methods and strategies to be engineered. As compiler designers are rising to the occasion, several compilation flows and frameworks have been proposed to tackle these questions. This thesis examines some of these works to characterize what are the most pressing topics that are being addressed and how they are being handled.

The rest of this thesis is organized as follows. In Chapter 1 we provide a background for the Processing in Memory paradigm, the technologies that implement it, and the role of compilers in its adoption. Chapter 2 presents compiler solutions that target Processing

Using Memory architectures, while Chapter 3 examines compiler techniques developed for Processing Near Memory systems. Finally, in Chapter 4, we take stock of the works analyzed in the previous chapters and comment over possible future developments in the field.

1 | Background

Traditional von Neumann architectures [81], as illustrated in fig. 1.1 (a), draw a clear separation between the memory, where the data resides, and the computation unit, where the data is elaborated, and require that the data moves back and forth between these two locations. As applications in fields like machine learning, bio-informatics, graph computing, and many more are rapidly evolving to be more and more data-intensive, this model is proving itself to be more and more inadequate: the movement of large amount of data is expensive in terms of both latency and energy consumption, as the bus that connects memory and CPU can only provide a limited bandwidth. Requests to main memory are by far the biggest performance bottleneck in modern systems, and are responsible for a significant percentage of applications power consumption; a 64-bit row access in DRAM can use up to twice as much energy as a double-precision floating point operation [84].

Moreover, memory technology itself is struggling to meet the needs of such systems. Memory manufacturing is nearing the end of Dennard scaling as DRAM channel width reaches its physical limits at just a few nanometers. Further miniaturization in an attempt to improve memory density is, if not unfeasible, technically difficult and economically impractical. The solutions that have been implemented to mitigate these limitations for general-purpose computing, namely complex memory systems with multiple layers of cache hierarchy, do not solve the problem in many of these emerging application scenarios. Workloads in application domains like graph computing and image processing exhibit irregular memory access patterns and poor cache locality, so the extra logic and mechanisms needed to manage the cache may in fact further worsen performance. Last but not least, optimizing memory usage is becoming increasingly necessary as, in recent years, major memory manufacturers are facing a supply crisis, which has led to a components shortage and high memory costs.

Since clearly this issue cannot be overcome while staying within the boundaries of the von Neumann model, computer architects have been exploring a different design paradigm: rather than moving data to the computation, move computation to the data, in an ap-

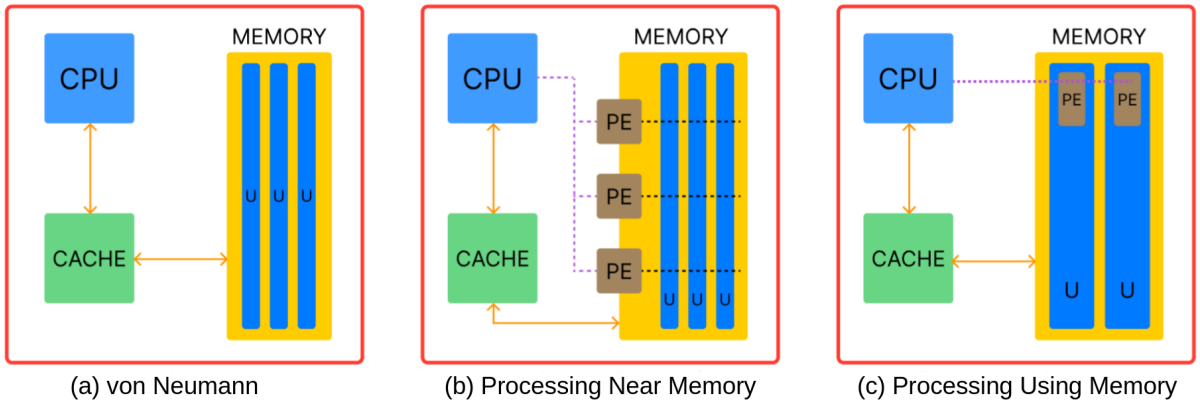

Figure 1.1: Comparison of von Neumann (a), Processing Near Memory (b), and Processing Using Memory (c) generic systems. Each system is composed of a CPU, a cache hierarchy, a memory composed of multiple memory units (U), and additional Processing Elements (PE). Adapted from [66].

proach called Processing In Memory (PIM) or Computing In Memory (CIM). This base idea can be declined to a number of implementation possibilities, that fall generally in two broad categories. The first option, Processing Near Memory (PNM), depicted in fig. 1.1 (b), is to integrate computation units in the memory chip or in the memory controllers; the other approach is Processing Using Memory (PUM), shown in fig. 1.1 (c), which consists in making marginal changes to chips to exploit the intrinsic properties of the memory cells to efficiently perform common operations.

1.1. Hardware support for PIM

PIM is not a new notion; it had already been proposed in the 1970s [69], but it had not gained traction due to a lack of hardware support that made the implementation too challenging. Recent advances in technology, however, have bridged this gap and allowed to design and propose serviceable PIM architectures. Two of these advances, in particular, are worth mentioning. In order to bypass the DRAM scaling limitations, research efforts have been spent for developing new memory technology that can store data at higher density than existing DRAMs, in the form of byte-addressable Non-Volatile Memories (NVMs). The main varieties of this kind of memory are metal-oxide Resistive RAM (ReRAM or memristor) [70], Magnetic Memory (MRAM)[37], and Phase-Change Memory (PCM)[41]. Each of these NVMs technologies has its own features, pros and cons, but what they have in common is that non-volatility and high density are their main selling points, while their biggest design challenge is providing memory access latencies that

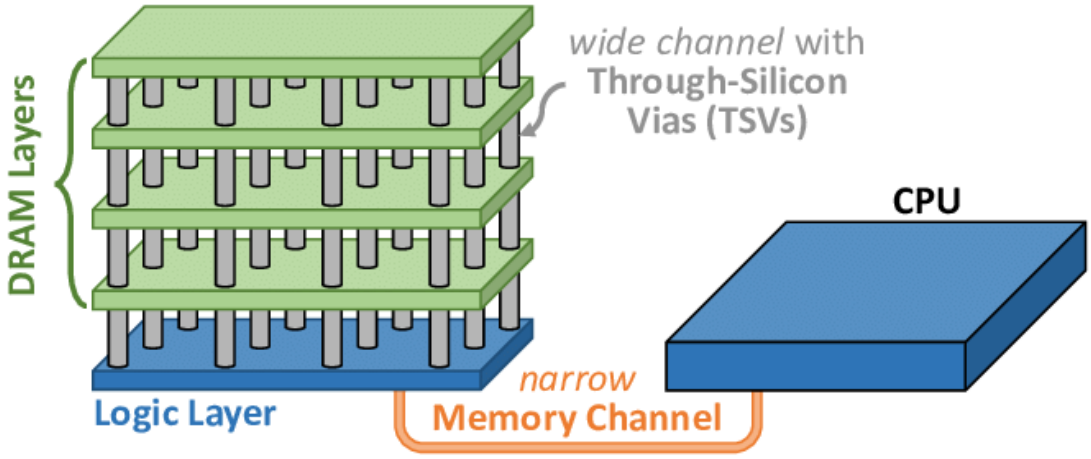

Figure 1.2: High-level overview of a 3D-stacked DRAM architecture. [50]

are competitive with DRAM. Other than functioning as memories, NVMs are also able to perform logic and arithmetic operations [14, 79, 83], making them prime candidates for implementing PUM functionalities. More specifically, as will be detailed in Chapter 2, they can be arranged in a crossbar layout, which allows to perform Matrix-Vector Multiplication (MVM), a common operation within ML and image processing workloads, with very low latency. Moreover, their high area efficiency means that a single chip can host many of these crossbars, allowing to exploit the high data parallelism that characterizes these workloads. NVM crossbars are thus an highly attractive option for designing accelerators, especially in the ML domain.

The other notable technological development that boosted the interest in PIM is the implementation of 3D-stacked memories. A 3D-stacked memory consists in multiple layers of memory (typically DRAM) stacked on top of each other and connected vertically using Through-Silicon Vias (TSVs), as illustrated in fig. 1.2. Since thousands of TSVs can be placed in a single memory chip, 3D-stacked memories can provide much higher internal bandwidth than usual DRAM designs. This type of memory also has lower energy consumption with respect to a conventional DRAM. Many 3D-stacked DRAM architectures, like High-Bandwidth Memory (HBM) [36] and Hybrid Memory Cube (HMC) [58], also integrate a logic layer as the bottom-most layer of the stack, that can be used by architects to implement processing functionalities. This logic layer is connected to the memory layers through the same TSVs, so it can exploit the massive available bandwidth for memory requests with a much lower latency than through a traditional off-chip memory bus, together with a significantly shorter data movement distance. This makes 3D-stacked DRAM particularly suitable for PNM designs; the drawbacks are the limitations imposed to the logic layer design by the nature of the architecture: area, energy and thermal con-

straints only allow to design a simple processor. Some architectures, like HMC, organize the stack vertically into several vaults, each with its own memory controller. Often this means that PIM architects will opt for implementing a computing unit for each vault, to exploit vault-level parallelism, which further reduces the available area for each one of them. Another option is the so called 2.5D solution, integrating a computing unit in the same die package as the stack, leveraging silicon interposers, i.e. in-package wires that connect directly to the TSVs.

1.2. Software support for PIM

Hardware support, however, is not the only element necessary for the adoption of PIM architectures in real-life application scenarios. PIM systems must also be programmable, and for that there needs to be a software infrastructure that is tailored to the novel characteristics of this new design paradigm. APIs, compilation flows, runtime support, and simulation frameworks are all necessary elements for successfully employing PIM architectures.

Traditional compilers cannot be reused as-is, but need to be rethought to fit the new architecture features. Compiler developed for von Neumann architectures, bottlenecked by memory accesses, deploy optimization methods to enhance memory efficiency like loop unrolling and unfolding to improve data locality [43]. This is not as relevant, for example, in PUM architectures, which perform computation inside of memory and should favor improving in-situ computation efficiency instead. Thus, new compilation tools should be devised. The role that compilers take on in the PIM ecosystem is twofold. First, they need to address the offloading problem, i.e. deciding which parts of the application code in question will execute on PIM. It is a problem that has several dimensions: what are the characteristics that make instructions suitable for execution on specific PIM architectures, when the offloading process is actually beneficial to performance, at which granularity the code should be offloaded. Once a policy has been established, identifying the candidates in the code needs to be performed automatically by the compiler, because doing it manually would require significant effort and detailed understanding of both the application domain and the underlying hardware on the part of the programmer. Second, the compiler must also deal with mapping and scheduling the code on the architecture and performing optimizations to make the code more efficient and improve hardware utilization. This is also a multi-faceted issue to tackle, as the mapping and optimization strategies closely depend on the features of the underlying hardware: what type of memory devices are being targeted, how the architecture is structured, what programming interface is being

exposed, what kind of operations does the memory system support. Because of this, many compiler solutions are tailor-made for specific architectures, but there have been proposals for more general compilation tools that work on suitable hardware abstractions.

2 | Processing Using Memory

Processing Using Memory (PUM) is the more literal approach to PIM, that tries to limit data movement by executing computations directly inside the memory. The operational principles of the memory technology themselves are leveraged to implement the computation of a variety of functions, from bulk bit-wise operations to arithmetic operations. They usually specialize in accelerating a specific operation, but it has been proven that these kinds of in-memory computation units, like [44] or [27], can even be functionally complete.

2.1. NVM Crossbars

As mentioned in section 1.1, Non-Volatile Memories are often arranged in a 2D array, i.e., a crossbar. This configuration allows them to exploit basic electrical circuit laws to perform Vector-Matrix Multiplication in the analog domain very efficiently.

As an example, fig. 2.1 shows a ReRAM crossbar; each cross-point is a memory cell, that can store an element G_{ij} of a matrix G as its conductance. If a voltage vector V, representing the input vector of the MVM, is applied to the rows of the crossbar, Ohm's law gives us the current value at each cross point as $I_{ij} = G_{ij}V_j$, and the accumulated current value that can be read at each column, for Kirchhoff's law, is $I_i = \sum_j G_{ij}V_j$. The current vector I represents the result of the dot product between G and V. Since

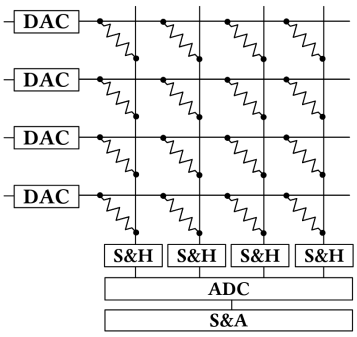

Figure 2.1: A ReRAM crossbar with peripheral devices [72].

the computation happens in the analog domain, it is necessary to also equip the crossbar with peripheral circuits, so that it can interface with the rest of the (digital) system: ADC/DAC, Sample and Hold (S&H), Shift and Add (S&A). Despite this, and despite inevitable analog inaccuracies that must be accounted for, this process allows to perform in constant time an operation that in traditional computing systems would require polynomial time. Executing a MVM on a memory crossbar is thus more efficient for two different reasons: not only is it more time-efficient, but it also saves the latency, the bandwidth and the energy consumption that the data would need to travel to the CPU and back. As MVM operations are ubiquitous in Machine Learning applications, and especially Neural Networks, NVM-based crossbars are being explored as accelerators for these workloads. These accelerators exploit the high area efficiency of this technology to implement a significant amount of crossbars on a single chip, organized hierarchically in multiple tiers (e.g. crossbars into tiles, tiles into cores). Their adoption, however, necessitates software mechanisms that allow to actually run code on them; compilers are one of the ways in which this matter is addressed.

2.1.1. Offloading techniques

Several of the compilation tools for NVM crossbar-based accelerators choose to integrate the polyhedral model [13] in their flow. The polyhedral model provides a representation for loop nests and allows to perform transformations and optimizations on them. The computations that will be performed at runtime are defined by their iteration domain, i.e. a set of statement instances executed inside a loop nest; each instance in the iteration domain is arranged in a schedule that determines the execution order. These schedules are represented by a schedule tree [78], that consists of multiple types of nodes. The root is called a domain node, and identifies the iteration domain of the whole tree. Sequence and set nodes encode the scheduling order and indicate sequential and concurrent execution, respectively. Their children are always filter nodes, that limit the schedule domain of their sub-trees to a subset of the iteration domain. Context nodes are optional and provide information about program parameters; band nodes encode partial schedules, and also indicate if the band dimensions are parallel or permutable; extension nodes add auxiliary statements like synchronization primitives and data transfer loops. Finally mark nodes are used to annotate a subtree with additional information needed for the code generation phase.

Vadivel, Chelini et al. [73] design TDO-CIM, Transparent Detection and Offloading for Computation In-Memory, an end-to-end compilation flow for devices equipped with PCM crossbars based on LLVM [1, 38], that automatically and transparently detects, optimizes,

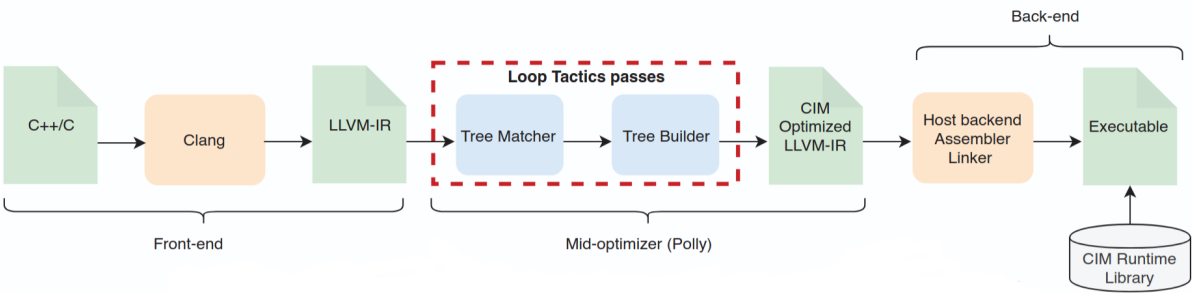

Figure 2.2: TDO-CIM compilation flow [73].

and offloads kernels suitable for in-memory acceleration. The workflow is shown in fig. 2.2. The input of the compilation flow is an application written in a high-level language, that is lowered to the LLVM Intermediate Representation (IR) by the corresponding LLVM front-end. The LLVM-IR is then fed to the polyhedral (declarative) optimizer Polly [42] that detects, extracts and models compute kernels. Polly represents the schedule of each detected kernel as a schedule tree, mapping each dynamic statement instance with its execution order using the nodes parent-child relationships within the tree as detailed above. Loop optimizations and device mappings are performed as tree modifications by Loop Tactics [16, 86], which works as additional passes within Polly. Loop Tactics outputs a CIM-optimized schedule tree, that Polly will then lower back to an imperative Abstract Syntax Tree (AST), and then to LLVM-IR. The compiler back-end will translate it to an executable and link it with the TDO-CIM runtime library.

Drebes et al. [22] also develop an end-to-end compilation flow, TC-CIM, that leverages the polyhedral representation, but they focus more on Machine Learning applications. Their starting point is Tensor Comprehensions [74], a framework generating highly optimized kernels for accelerators from an abstract, mathematical notation for tensor operations, which they map to fixed-function memristor-based hardware blocks. They also choose to detect offloadable operations using Loop Tactics [16, 86], a declarative framework to describe computational patterns in a polyhedral representation. Simulation results show that TC-CIM is able to reliably recognize tensor operations commonly used in ML workloads across multiple benchmarks.

Tensor Comprehensions targets GPU architectures, so it needs to be adapted to work for the memristor-based architectures targeted by TC-CIM, that can only support fixed functions and are not programmable for general purposes. This means that parallelism can only be exploited within the offloading patterns and there is no concept of threads executing in parallel. The instructions that do not fit the fixed offloading pattern must therefore be executed by a general-purpose host CPU, while the fixed patterns are of-

floaded via specialized instructions. To support CIM accelerators, the authors added a backend that generates sequential C code; it is based on the existing Tensor Comprehensions CUDA backend, but disregards all the components partitioning for blocks and threads, synchronization between instructions executing in parallel, memory promotion and any CUDA-specific primitives and library calls.

The TC-CIM compilation flow matches to the default flow of Tensor Comprehensions, up until the generation of the polyhedral IR. Before the generation of the isl AST [76], however, the flows diverge as TC-CIM carries out the pattern detection pass based on Loop Tactics. The pass consists in searching the schedule tree with the appropriate pattern matchers, and recording the found matches using mark nodes. The mark nodes are preserved in the isl AST and processed by the custom printer generating C code, which finally emits function calls to a CIM accelerator library. The Loop Tactics pass focuses on detection and offloading of the individual operations supported by the hardware, and does not include inter-operation optimizations (e.g., minimizing data movement between the host and the accelerator, operator fusion for combined instructions, etc.). This is why pattern matching is performed on the schedule tree rather than on TC expressions, to benefit from affine scheduling and its ability to expose permutability and parallelism properties, and coalesce input bands into a single band when possible, which keeps the matching patterns simpler. On the downside, this means that the scheduling is unaware of the higher-level information extracted by the matchers.

TC-CIM's patterns look for three operations: plain matrix-vector multiplication, plain matrix multiplication and batched matrix multiplication, accounting also for loop permutations and transposed accesses. The schedule tree patterns of these three operations are similar enough that recognition can be implemented with a single matcher and appropriate functions distinguishing between the operations when processing a match. The matcher is also provided with a callback function to verify that a candidate match is not actually a false positive that includes operations not supported by the target hardware. When a match is found, the root node of the matched sub-tree is tagged with a mark node whose name indicates the matched operation (i.e., _mvt, _gemm or _batched_gemm) and whose pointer for user-defined data receives the address of a structure with all meta-information that is needed to emit the call to the specialized library function with the correct parameters during code generation. This information is preserved during the generation of the isl AST, and is processed by the printer generating the C code.

TDO-CIM and TC-CIM both have some limitations. First, they detect and offload matrix-matrix multiplications (MM) and matrix-vector multiplications (MVM), but are not able to recognize other common operators in NN workloads, such as convolution (CONV).

Second, they both rely on Loop Tactics, so they use a detect-and-rebuild approach where the schedule tree is updated right after a target operation is identified; this prevents them from taking advantage of optimization opportunities for the NN as a whole. Han et al. [32] try to overcome these limitations by proposing a source-to-source Polyhedral-Based Compilation Framework for memristor-based Neural Network inference accelerators (PBC). They extend the set of operations that can be detected by pattern matching, and add support for pipeline generation to exploit the parallelism in the NN workloads and achieve better utilization of hardware resources, which allows to improve performance by an order of magnitude.

The main body of the compilation flow works on the polyhedral IR of the original source code. The target operators are detected, their respective function calls are built, then the polyhedral IR is used to generate the output source code. The proposed compilation framework is implemented using isl [76] and pet [77]. The authors chose not to use Loop Tactics, because they found that the Loop Tactics tree builder is not suitable for describing the pipelined output source code; they prefer to build the offloadable operators AST nodes directly in the schedule tree rebuilding phase. The authors work with C for both the original and generated source code, despite it not being very efficient in describing NN workloads; however, the core idea of the framework is based on analyzing and transforming a polyhedral model, so it can be extended to work with any front-end that can be lowered to a polyhedral IR.

Once the source code has been lowered to a polyhedral schedule tree, operator detection is implemented as sub-tree matching. In the usual approach, the target operator is described specifying its sub-tree structure (the node types and topological relations between them) and memory access relations (the array indices accessed by each statement instance). The original source code schedule tree is searched in a bottom-up manner to find the sub-trees that match the target description; the operator type is also checked to verify that it is a multiply-and-accumulate (MAC) statement. The authors extend this approach to identify other operations that are common in NN workloads:

- Convolution (CONV): it has a similar sub-tree structure to MM, but with higher dimensionality and more complex access relations, which require dedicated processing.
- Pooling: a variation on CONV, where the number of input and output channels is the same and the statement is a call to a pooling function instead of a MAC.
- Activation functions: some memristor-based accelerators co-locate MM and activation units [18, 64] to reduce the amount of data movement, and others directly

Order	Operator
1	CONV + INIT + ACT
2	CONV + ACT
3	CONV + INIT
4	Bare CONV
5	Pooling
6	MM + INIT + ACT
6	MM + ACT
8	MM + INIT
9	Bare MM
10	MV + INIT + ACT
11	MV + ACT
12	MV + INIT
13	Bare MV

Table 2.1: Supported Operators for PBC [32]

implement fused operators to benefit from the efficient peripheral circuit design [31, 34]. Therefore, activation functions are not detected alone, but fused with MM or CONV.

In addition, the matcher distinguishes the detection of operators with initialization from bare ones where initialization statements are not included. The detection of the different operators is carried out sequentially in the order listed in table 2.1. Fused operators have priority over their bare counterparts, and operators with higher dimensions precede those with lower dimensions.

When a match is found, a mark node is inserted into the schedule tree to tag the sub-tree as a detected operator. It is also attached with information like the partial iteration domain of the operator, identifiers of the accessed arrays, etc. During the schedule tree rebuilding phase, the inserted mark nodes and the attached information are used to build the AST node that will generate the calls to the corresponding API functions for the operators. This AST node is attached to the mark node and during the AST generation, that proceeds top-down on the schedule tree, it will be inserted into the output AST tree in place of the sub-tree of the mark-node.

To account for the practical details of the accelerator design, the rebuilding should handle the following issues:

- **Transformation of CONV into MM:** the CONV kernels are converted into a weight matrix, the shape of the input features is adjusted accordingly, and the necessary load and store instructions are inserted before and after the MM function.

- **Operator tiling:** MM operations need to be tiled to fit into the crossbar size. The resulting AST node is a *for* node that contains the outer loops of the tiled loop nest, whereas the inner loops are carried out by the generated function call.
- **Fused operators:** when dealing with the fused operator of MM and activation, the activation should be carried out only after the final accumulation. Therefore, a guarding *if* node is generated to select whether a bare MM or a fused MM activation is called according to the loop index.
- **Invocation granularity:** according to specific usage scenarios, the proposed framework can be configured to generate different types of AST nodes in different invocation modes: (a) coarse-grained invocation: if the accelerator is equipped with its customized toolchain, the AST node is built as a user node that contains only one function call that will pass the operands and the iteration domains to the toolchain, who will lower the operation on its own; (b) fine-grained invocation: the execution details are controlled by the proposed framework.

Tested on benchmark kernels, the PBC framework detected all the operations present, even some that TC-CIM missed because it is only able to match MMs with an initialization statement. The rebuilding phase also proved to be reliable. Detecting and optimizing all the operators before rebuilding proves to be crucial to fully exploiting the potential of the memristor-based accelerators, as it allows to optimize the workloads from the whole network perspective. Case studies have been conducted on two typical memristor-based architectures [34, 64], demonstrating the generality of the proposed framework over different architectural designs.

Another proposal of compiler for memristor-based accelerators for ML workloads comes from Siemeniuk et al. [65], in the form of OCC (Optimizing Compiler for Computing-in-memory). Unlike the previously mentioned works, OCC does not adopt a polyhedral model. The authors opt for a compiler based on multilevel IR rewriting, where the source code gets progressively lowered level by level, and code transformations and optimizations are performed at the most suitable level of abstraction. The IRs on which the various levels work are in Static Single-Assignment (SSA) form, which allows to reuse existing optimizations developed for general-purpose compilers. The lowering pipeline uses MLIR [39, 40], an LLVM compiler infrastructure with a customizable IR for heterogeneous hardware, that allows developers to design their own IR dialects to match the desired abstraction level. OCC uses the Linalg, CIM, SCF, and Standard dialects.

Figure 2.3 details the OCC lowering pipeline. Like TC-CIM, the entry point in the compilation flow is a ML model expressed in Tensor Comprehensions. The front-end of

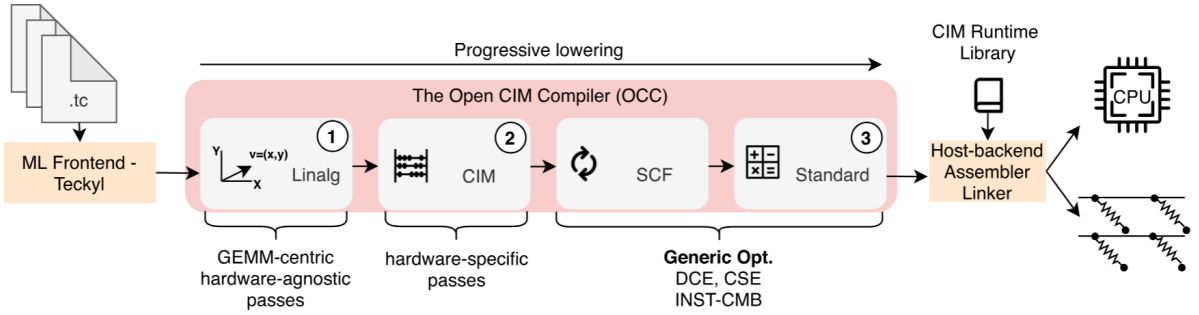

Figure 2.3: OCC pipeline [65]

choice is Teckyl [21], but it can be replaced with whatever front-end can be lowered to the Linalg/CIM dialects. Across the lowering pipeline, there are three types of transformations that work in symbiosis.

1. Hardware-Agnostic Rewriting Passes: these passes work at the Linalg level and their purpose is rewriting computational patterns using matrix-matrix multiplication.

- The Transpose Transpose GEMM Transpose (TTGT) pass rewrites contractions by applying the TTGT transformation scheme [67]. Contractions are detected by looking for operations with exactly two inputs and one output, and that contain a multiply-accumulate computation (verified by analyzing the innermost loop's operations); moreover the dimensions of the operands must fulfill the following criteria: (i) the dimensions common to both of the inputs are the reduction dimensions and shall not be present in the output; (ii) the input dimensions other than the reduction dimensions shall be present in the output; (iii) the output shall not contain any dimensions that are not present in the inputs. The identified contractions are rewritten as a composition of transpose, reshape and GEMM operations, the latter of which will be offloaded to the CIM accelerator.
- The Im2Col pass rewrites convolutions by applying the Im2Col transformation [75]. The pass looks for operations for which one of the outputs represents a sliding window. The Im2Col transformation considers a convolution as a dot product between the kernel filters and the local regions of the moving window. By stacking each window column-wise, and the filters row-wise, and multiplying the resulting matrices, the result is equivalent to that of the original convolution. The GEMM computations that appear with this transformation are offloaded to the CIM accelerator.

This approach allows OCC to reliably identify all offloading opportunities, even catching some that TDO-CIM and TC-CIM miss. This is because these frameworks are not able to identify and map contractions; moreover, Loop Tactics, on which they both rely for

pattern matching, works on a representation that is too low-level, whereas progressive lowering allows OCC to perform operator detection at a more suitable level, preserving semantic information until it is needed.

2. Hardware-Specific Passes: these passes work at the CIM dialect level, that functions as an interface to the underlying accelerator hardware. The optimizations performed focus on data layout and computation reorder, with the goal of enabling computation on the crossbars, maximizing hardware utilization and transfer bandwidth, and extending crossbar lifetime.

- tiling pass: splits matrix multiplications that do not fit in a single crossbar into multiple smaller GEMMs, and takes care of the accompanying memory/buffer/partial results/accumulation modifications;
- loop interchange pass: performed to reduce the number of crossbar writes performed during a a tiled GEMM computation;
- loop unrolling pass: unrolls the inner dimension loop of a tiled GEMM so that the computation can be parallelized across multiple tiles, improving latency.

3. Lowering to CIM Library Calls: the accelerator exposes an API through a runtime library. Each CIM dialect operation corresponds one-to-one to a runtime library function call, to which they are mapped during the lowering to the SCF and then Standard dialects; the rest of the operations get lowered to the other existing MLIR to be compiled for the host core.

Li et al. [45] try to further extend the range of identifiable operators. To achieve this, they implement a compiler framework, HARMONY, that works over a hardware abstraction that describes the diverse landscape of crossbar architectures; they develop a hardware IR to abstract the intrinsic computation and memory behaviors of NVM-based accelerators into a programmable form, and they design an algorithm that matches the computations described in the software IR with the intrinsic behaviors defined in the hardware IR. They automatically identify all offloadable operators, and find the most suitable strategy to map operations onto crossbars with a reinforcement learning (RL)–based search algorithm. Fig. 2.4 depicts the HARMONY workflow.

The front-end of HARMONY works in parallel over a high-level description of a NN model, described using a neural network framework like MXNet [9], TensorFlow [5] or PyTorch [55], and the hardware configuration, which specifies the architectural parameters of the target accelerator. The NN model is translated into a tensor-level IR, using a Domain Specific Language (the authors use the TVM's compiler framework DSL [17]).

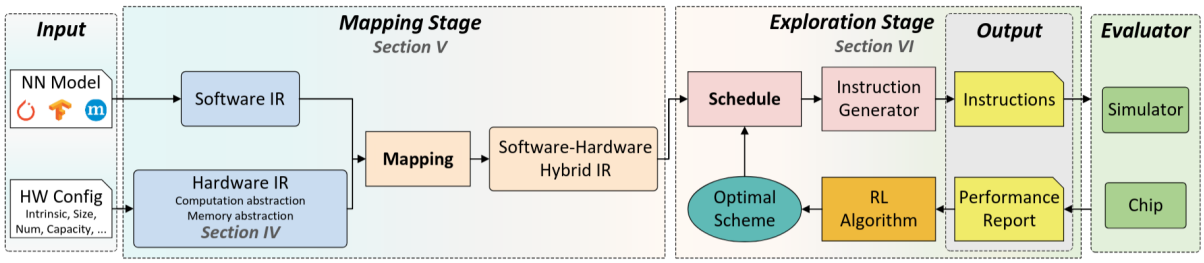

Figure 2.4: HARMONY workflow [45]

HARMONY uses the same DSL to construct a compatible hardware IR, that will describe the inputted hardware configuration and capture its computation and memory behavior.

Once the hardware and software descriptions are converted to their respective IRs, HARMONY's mapping algorithm determines which parts of the software should be offloaded to the crossbars, and how the software operations correspond to the hardware components. The hardware IR allows to identify and enumerate all feasible mapping schemes for the given software operations. The mapping process happens in two stages: first the software operations are mapped onto an idealized hardware, with the same computational primitives (e.g. MVM) as the target hardware, but no constraints on memory capacity or computation granularity; then, the mapping is concretized by applying partitioning, padding and remapping to adhere to the hardware constraints. Both mapping stages employ loop transformations such as split and reorder to restructure the nested computation loops; moreover, to accommodate the transformed operations, the input and output tensors or buffers are reconstructed, leveraging compiler passes such as layout transformation and InferBound. The mapping phase finally generates a hybrid IR, which consists of the outer software loops together with the inner hardware IR that represents the CIM primitives.

HARMONY then tries to determine the optimal scheduling strategy for this hybrid IR. As the scheduling space is too vast for exhaustive search, and the use of heuristic would require significant manual effort and specialized knowledge, the authors opt for a RL-based exploration. For each possible schedule, the compiler generates low-level instructions through the instruction generator, and executes them on either a cycle-accurate simulator or a real chip. The measured performance result is used by the RL algorithm to guide further explorations, until convergence to the optimal schedule that balances the optimization objectives.

Comparisons with other compiler frameworks (like PUMA [10], PIMCOMP [72], and OCC [65]) show that HARMONY manages to identify and map operators with more complex

dataflows and richer optimization opportunities. Naturally the iterative training phase makes for higher compilation overhead with respect to the mentioned compilers, but the overall compilation time of HARMONY remains acceptable, since the scheduling process is performed only once for each workload and the optimized configuration can be reused across multiple deployments.

2.1.2. Code optimizations

Other than identifying the operations that can be executed on the crossbars, PIM compilers must also optimize the code for improving efficiency and hardware utilization. Some of the most common issues to solve for crossbar architectures are:

- partitioning the weights of a single matrix to multiple crossbars, to account for the physical crossbars limited size;
- mapping the operations to the various architecture tiers (crossbars, tiles, cores), minimizing the costs of communication among the various components;
- for NNs, replicating the weight of a layer over multiple crossbars (or groups of crossbars) to improve their execution latency;
- for NNs, pipelining of the layers to exploit parallelism.

Chi, Li, et al. [18] use ReRAM crossbars to develop PRIME, one of the first PUM Neural Networks accelerators. The ReRAM crossbars can be configured both as accelerators and as regular memory. The PRIME compiler implements optimizations for mapping the NN, and for exploiting the bank-level parallelism of the ReRAM memory for further acceleration. The NN mapping is optimized based on the network scale:

- *Small-Scale NN:* they are NNs that can be mapped to single mat of reconfigurable subarrays (full-function subarrays, or FFs). To utilize the cells that would otherwise be idle, and to amortize the latency of the peripheral circuits that could overwhelm such small computations, the NN can be replicated on another portion of the mat; if other mats are idle, the replication strategy can extend to them as well, as long as there is enough available bandwidth.
- *Medium-Scale NN:* they are NNs that cannot be mapped to a single FF mat, but can fit to the FF subarrays of one bank. The NN is split into small-scale NNs at compile time, and then their results are merged.
- *Large-Scale NN:* they are NNs that cannot be mapped to the FF subarrays in a single bank. The naive solution of splitting a large-scale NN into several medium-scale

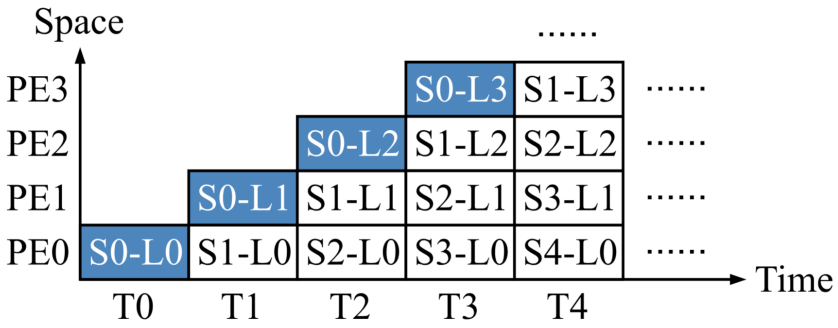

Figure 2.5: Pipeline example with four PEs. S_0–S_4 are five samples in one batch [32].

ones and executing them serially on the same bank, introduces excessive overhead for reprogramming the FF subarrays at each step. It is better to use multiple banks, exploiting the inter-bank data bus implemented in PRIME. The execution on the banks can even be pipelined to improve throughput.

The output of compiling is the metadata for synaptic weights mapping, data path configuration, and execution commands with data dependency and flow control.

The Polyhedral-Based Compiler framework from Han et al. [32] does not only focus on the detection and offloading of common NN operands, but also aims to improve performance by leveraging the parallel nature of the NN workloads and improving hardware utilization. Previous polyhedral-based compilers, like TC-CIM [22], and TDO-CIM [73], that generate offloading code directly after identifying one operator and then continue to look for the next one; this leads to significant performance loss due to underutilization of the hardware when large batches of data are available. Conversely, PBC analyzes the detected operators before the rebuilding phase to check if they can be pipelined, then it allocates the Processing Elements (PEs) among the operators to maximize the pipeline throughput. The code is pipelined making use of the batch and layer dimension of NN workloads. The PEs are assigned to process a certain layer of the NN for the various samples, as shown in fig. 2.5.

Since some operators reuse their weights multiple times, the compilation framework will assign them more PEs to reduce their latency. Given the total number of PEs, resource allocation decides how many times each operator will be replicated, i.e. its replication degree. The allocation algorithm starts by initializing the replication degree of each layer to 1. Iteratively, the algorithm identifies the bottleneck layers with the highest latency; if there are enough free PEs to fit one more copy of their weights, their replication degree is increased and the process repeats, otherwise the procedure ends. The PE indices thus

generated will be passed as arguments to the API function calls.

Fujiki et. al (2018) [26] design a general purpose In-Memory Data Parallel processor (IMP), based on NVM crossbars, extended to support in-situ operations beyond dot product (i.e., addition, element-wise multiplication, and subtraction). Their design goal is to leverage the underlying parallelism in the hardware by merging the concepts of data-flow and vector processing. Data-flow exposes the Instruction Level Parallesism (ILP) in the program, while vector processing exposes the Data Level Parallelism (DLP) through a Single Instruction Multiple Data (SIMD) execution model.

The main feature of the IMP architecture is the number of supported operations: other than the widely implemented dot product, the ReRAM arrays can also execute dot product, addition, multiplication and subtraction. To further extend the set of supported operations, as the architecture is designed with general purpose computing in mind, more complex operations (division, exponents, transcendental functions, etc.) are reduced to a set of Lookup Tables (LUTs), additions and multiplications. The compiler achieves this using either Newton-Raphson or Maclaurin-Goldschmidt methods, that consist in iteratively applying a set of instructions to a seed from a look-up table; these algorithms were chosen over simpler ones (e.g., SRT division) because they do not require bit shifts or bit-wise logical operations (tricky to implement on ReRAM arrays), and are less space-consuming. The compiler also lowers convolution nodes in the Data-Flow Graph to the memory ISA by mapping the input data to the array and streaming in the filter. The convolution is decomposed into a series of matrix-vector dot-products that are executed simultaneously on different input matrix slices.

They design an architecture-specific compiler that takes TensorFlow [5] as input. The TensorFlow programs are Data-Flow Graphs (DFG) where each operator node works on tensor operands. The compiler refers to a DFG operating on one element of a vector as a ‘module’; the input DFG is translated into a collection of data-parallel modules with the same machine code. The proposed architecture processes data in a SIMD execution model at the granularity of module. At runtime, different instances of a module execute the same instructions on different elements of input vectors in a lock-step manner. The compiler generates a module by unrolling a single dimension of multi-dimensional input vectors. Each module is composed of one or more Instruction Blocks (IB); multiple IBs in a module may execute in parallel to expose ILP. The compiler explores several optimizations to increase the number of concurrent IBs in a module and thereby exposes the ILP inside a module.

The compiler tool-chain is developed using Python 3.6 and C++. The compilation flow

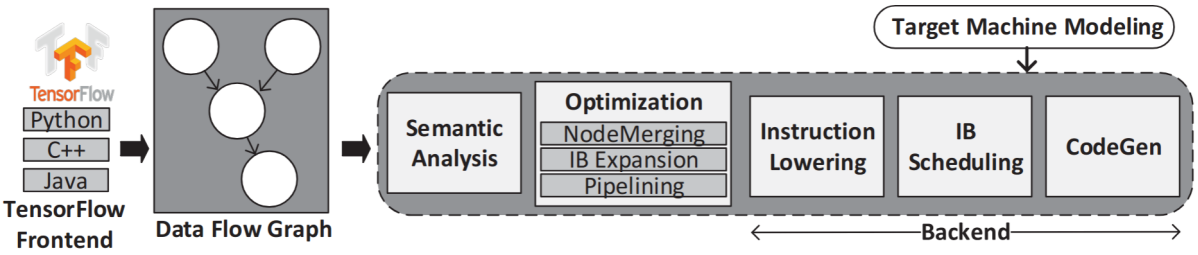

Figure 2.6: IMP compilation flow [26].

is shown in fig. 2.6. The compiler front-end uses the TensorFlow core framework to parse a TensorFlow DFG in the protocol buffer format, and analyzes the semantics of the input DFG which has vector/matrix operands, and creates a module with a single IB with required control flow.

The IMP compiler implements several implementation passes.

- **Node merging:** a node merging pass looks for series of 2-operand compute nodes of a module that can be collapsed into a single compute node with n operands. The maximum value of n is bounded by the number of array rows, and by the resolution of the ADCs, which is proportional to their power consumption. The compiler can generate code for an arbitrary resolution n chosen by the chip architects based on their power constraints. The pass also merges certain combinations of nodes to reduce intermediate writes to the memory arrays.
- **Instruction Block scheduler:** the compiler implements the Bottom-Up-Greedy (BUG) algorithm for VLIW architectures [24] to exploit ILP and concurrently schedule independent Instruction Blocks (IBs) in a module. The algorithm traverses the DFG in a bottom-up fashion looking for candidate assignments of the instructions, until it reaches the input (define) node. It then walks a top-down path to make a final assignment, trying to minimize the data transfer latency by taking into account both the operand and the successor locations. The BUG algorithm is adapted to account for read/write latency, network resource collision latency, operation latency, and, most importantly, the fact that for in-memory computing a functional unit coincides with the data location.
- **Instruction Block expansion:** to further exploit ILP and DLP, blocks that process multi-dimensional vectors can be expanded into several instruction blocks with lower-dimension vectors. The expansion pass traverses a DFG's nodes bottom-up and breadth-first to detect and expand the sub-trees that process multi-dimensional vectors of the same size; it also inserts 'pack' and 'unpack' pseudo-operations to

ensure that the dimensions are consistent between the sub-tree regions, which will be later converted to *mov* instructions.

- **Pipelining:** the compute and write-back phases of the instructions are pipelined, using one array for computation and a separate array as the write-back destination. In the worst case scenario this drops the arrays utilization by a half, so this optimization is applied only when the number of modules needed for the input data is lower than the aggregate SIMD capacity of the memory chip.
- **Balancing inter-module and intra-module parallelism:** to avoid performance loss due to IBs across all module instances exceeding the number of available SIMD slots, the compiler can generate code for arbitrary upper bounds on the number of IBs per module and can flexibly tune the intra-module parallelism with respect to inter-module parallelism. The authors develop an analytical model to approximate execution time given the number of IBs per module and number of module instances. At runtime, when the number of module instances is known, the optimal code is chosen.

In [8] Ambrosi et al. present a software stack designed for a memristor-based NN accelerator. The stack comprises an ONNX converter, that allows compatibility for neural network models developed on frameworks that support the ONNX format [25] (like CNTK [61], Caffe2 [80], TensorFlow [5], etc.); then an application optimizer, to adapt the models to the underlying hardware, a compiler, to generate executable ISA code, and an emulator, for hardware design-space exploration and testing.

Ambrosi et al. implement several optimizations:

- **Quantization:** enables the casting of large and complex types (like 32-bit floating points) to a lower precision format available on the memristor-based accelerator, with negligible accuracy loss. It allows for reduced memory footprint, faster inference and lower energy consumption. The presented software stack provides an automatic quantization process for pre-trained neural network models based on normalization values provided by the user. The quantization calibrates tensors between layers by using operations such as `clip(vector, min, max)`, which saturates values greater than `max` and less than `min` to `max` and `min` respectively, and `normalize(vector)`, that adjusts the distribution of the vector values. This is sufficient for simple models, like Multi-Layer Perceptrons and Recurrent Neural Networks; more complex models require a more robust method to minimize accuracy loss, but the authors had not yet implemented it at the time of writing. Using the performance simulator, the authors verified that reducing the number of

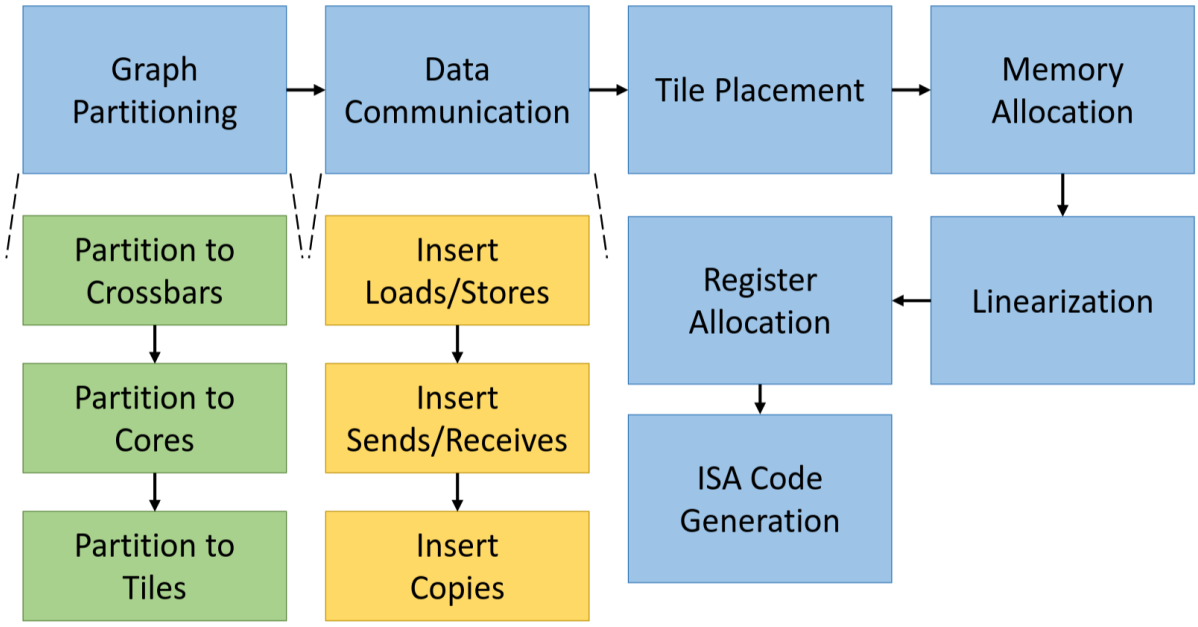

Figure 2.7: Compilation flow from [8]

bits used to represent weight data results in the reduction of both the number of memristive crossbars used and the number of operation executing on the crossbars, thus improving the costs of ALU and memory access and resulting in lower energy consumption and faster execution per inference.

- **Node Aggregation:** to exploit the accelerator to its fullest it's convenient to increase the amount of MVM performed by a model/decrease the number of non-MVM operations, for example by transforming (A) into (B)

  (A) vector<input>* matrix<weights> + vector<bias>

  (B) vector<input>* matrix<weights>

  incorporating the bias vector in the weight matrix, or by aggregating batch normalization layers into the convolution.

- **Layers replication:** if the regular dataflow pipeline leaves unused or dormant resources, some layers of the model can be replicated to improve hardware utilization and increase performance. It may happen that the pipeline of the model is already balanced and there are still available resources, in which case the entire model can be replicated. This increases the number of inferences per second, with only a minimal increase in latency due to layer synchronization and data distribution.

The input of the compiler workflow (fig. 2.7) is a graph representation of a NN model that is either constructed by the ONNX back-end or supplied directly by the programmer via

the accelerator's custom API. In the first stage, the graph is partitioned and the operations are virtually mapped to the different crossbars, cores, and tiles, according to the hardware hierarchy, using a bottom-up approach. Firstly, all MVM operations that use the same constant matrix are assigned to the same virtual crossbar. Then, non-MVM operations are assigned to the virtual crossbar that contains the MVM operation for which it has the most 'affinity'. Affinity spreads from each MVM operation to all source (destination) operations that feed exclusively into (out of) that MVM operation; when a non-MVM operation has multiple sources (destinations) with affinity to different virtual crossbars, the final choice is made using heuristics. The second level of partitioning works on a graph obtained from the original one, where each node represents the sub-graph of the operations assigned to a given virtual crossbars, and the edges across sub-graphs are aggregated into a single edge. This new graph is partitioned by a third-party graph partitioning software such as KaHIP [63], with the goal of mapping virtual crossbars that communicate frequently together in the same virtual core, while respecting the maximum number of crossbars per physical core. The third level of partitioning works in the same fashion, mapping frequently communicating virtual cores into the same virtual tile, without exceeding the maximum number of cores in a tile.

After the graph has been partitioned, the compiler inserts data communication operations between producer and consumer operations assigned to different cores and tiles. For all producer-consumer edges going across cores, a store operation is inserted on the producer core and a load operation is inserted on the consumer core. The same is done for tiles with the ISA send and receive operations for the store-load edges going across tiles. Finally, data communication is also handled across register spaces within a core (crossbar input/output registers and the register file). Whenever there is a mismatch between the register spaces of a producer and consumer operation, the required intermediate copy instructions are inserted. The graph partitioning strategy explained above, compared to a baseline that partitions the graph randomly, reduces the number of loads, stores, sends, and receives, hence the overall energy.

Next, the virtual tiles obtained during tile partitioning need to be mapped onto the physical tiles, preferably keeping the ones that communicate frequently close together to reduce the data movement overhead. Minimizing the communication distance between tiles would be an NP-complete problem; the heuristic adopted places the tiles in the order in which their matrices are used by the program, so that adjacent layers that communicate frequently are mapped to adjacent tiles. The mapping of virtual cores onto the physical cores of their tile, and similarly for virtual crossbars, is trivial because cores within a tile, and crossbars within a core, are logically equidistant to each other.

The memory allocation phase is performed by allocating a new tile data memory location for every store and receive operation performed on a tile. The compiler does not support reuse of memory locations. After memory allocation, the graph is linearized into a sequence of instructions for each tile and core; it is then possible to analyze the live ranges of the virtual registers and perform register allocation for each core with register reuse. Finally, assembly code is generated for each tile and core from the linearized instruction sequences. The authors developed a variant of the standard ELF format catering to NN applications, which includes the weights, biases, and activation functions that can be accessed by the runtime.

Building over the effort of [8], Ankit et al. designed PUMA [10] with the design goal to preserve the storage density of memristor crossbars to enable the mapping of ML applications using on-chip memory only. The PUMA compiler is a runtime compiler implemented as a C++ library. Like [8], the input NN model graph representation can be either provided directly through the PUMA native C++ interface, or lowered from description in the ONNX format for interoperability with common DNN frameworks. The first steps of the compilation process, namely graph partitioning and insertion of communication instructions, proceed in a very similar way to [8].

Since PUMA is a spatial architecture (not a data-parallel processor), each core/tile executes a different set of instructions, obtained through the linearization of the partitioned graph. This instruction scheduling phase has three main objectives:

1. Reducing register pressure: the graph is traversed in a reverse post-order fashion, which prioritizes consuming produced values before producing new ones. Since this reduces the number of live values, register pressure is alleviated.

2. Capturing ILP of MVM operations: in order to run multiple MVM units simultaneously in the same core and reduce latency, an optimization named MVM coalescing is performed, which consists in fusing independent MVM operations that would otherwise run on different MVMUs. Coalescing is possible when the MVMs have no data dependencies between them; it is also preferable for their results to be consumed soon after one another to reduce register pressure. Before linearization, the compiler first fuses MVMs that are tiles of the same large MVM operation; then it traverses the graph in reverse post-order and coalesces the remaining MVMs with the first eligible candidate it finds in the traversal order. The dependence information is updated after every fusion. The graph is then traversed one last time for linearization.

3. Avoiding deadlocks: the linearization process introduces control edges between the

cores sub-graphs, and, since communication across cores is blocking, this can potentially cause deadlocks. To avoid this, sub-graphs are not linearized independently, but all at once, ensuring to maintain a globally consistent linearization order.

The final step of the compilation flow is register allocation. Each core has three sets of registers: XbarIn, that can be written by any instruction and read only by MVM operations, XbarOut, that can be read by any instruction and written only by MVM operations, and general purpose, that can be both read and written by any instruction. Liveness analysis is performed on each set separately. Register conflicts in the XbarIn and XbarOut sets are spilled in the general purpose registers, and conflicts on the general purpose registers are spilled to shared memory with load/store instructions.

The PUMA compiler is extended by its authors to support compilation for the PANTHER architecture [11] for neural network training. Unlike inference, training a neural network requires frequent updating of the weight matrices, so ReRAM crossbars, with their high-latency writes, are not an immediate solution. The authors adapt an existing technique that allows to perform the required operations without serial reads and writes [49, 53], and update the PUMA architecture and compiler so they can support M^TVM and Outer Product Accumulate operations.

Sun et al. [72] design PIMCOMP a compiler for DNN accelerators that works for an abstract crossbar architecture organized in the usual crossbar/IMA/tile/chip hierarchy. The compilation flow (fig. 2.8) operates in four stages: node partitioning, weight replicating, core mapping, and dataflow scheduling, with the goal of making efficient use of the hardware according to the DNN execution details. The workflow supports two compilation modes, that correspond to two different inter-layer pipeline granularities, to satisfy the requirements of both high-throughput and low-latency applications. High Throughput (HT) mode is suited for application scenarios with continuous input data that comes in large batches; it produces code that processes the data layer by layer: when the pipeline is full, different layers process data from different inferences, and parallelism between layers is high as there is no inter-layer data communication. Low Latency (LL) mode is better adjusted to application scenarios with intermittent input of small amounts of data; once a layer produces its output, it immediately passes it to the next layers that need it, and when a layer receives enough input data it immediately starts operatings so the overall latency is reduced.

PIMCOMP takes as input a DNN model in the ONNX format, which is parsed to get the model description (e.g. node information and topological relatioships) on which the four backend compiler stages will work.

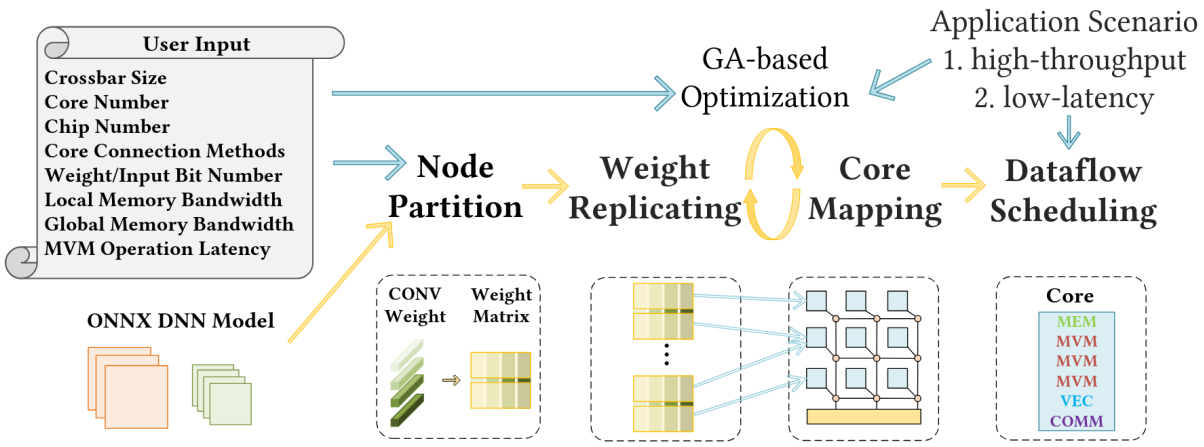

Figure 2.8: PIMCOMP compilation flow [72].

The first stage is node (or layer) partitioning: the weight data of the convolutional layers and fully connected layers of the model needs to be mapped on multiple crossbars, to account for their limited size. The layers are converted in MVM operations: the weights of each kernel are flattened into a column and the resulting weight matrix is partitioned into several Array Groups (AG), each containing several crossbars. Once the weights have been partitioned among the crossbars, the AGs are mapped to the cores. It is not necessary that AGs of the same node are mapped on the same core, nor that a core contains only AGs of the same node; it is however preferable, to avoid moving the partial results across the cores when they need to be accumulated to obtain the complete result. Moreover this mapping strategy can reduce control complexity and repeated access to the Input Register, which alleviates on-chip bandwidth and buffer pressure.

Since the contribution of weight replicating and core mapping to the accelerator performance is intertwined, the former improving computation parallelism and the latter managing structural conflicts and data dependencies, these two stages are performed simultaneously. The authors approach this optimization problem with a genetic algorithm. Each gene encodes several AGs of a node; they select a maximum number of nodes that can be mapped on a single core, to make sure that core-to-core communication does not become a significant bottleneck. The chromosome length is the number of cores times the maximum number of nodes in a core, and the position of each gene in the chromosomes will ascertain the index of the core to which the respective AGs will be mapped. The initialization consists in a random selection of the replication number for each node, and a random mapping of the AGs to the cores. The fitness function depends on the selected compilation mode and is detailed below. The crossover phase is skipped, as it has no significance in this particular problem. In the mutation stage, a random individual is se-

lected and undergoes one of the following operations: (i) randomly select a node, increase its replication number, and randomly map the expanded AG to cores; (ii) randomly select a node, reduce its replication number, and recover the crossbar arrays occupied by that replicated block; (iii) randomly select a gene and spread its AG to other cores; (iv) randomly select a gene and merge its AG into the same node of other cores. This process is repeated until convergence.

The fitness function of the genetic algorithm depends on the compilation mode:

- in HT mode, the performance is determined by the overall inference time. The estimate execution time of the i-th core is determined as:

$$time_i = f(n) = n > \frac{T_{MVM}}{T_{interval}} \ ? \ n \times T_{interval} : T_{MVM}$$

  where n is the number of AGs in the core, T_{MVM} is the time required to complete a single MVM operation, and $T_{interval}$ is the time interval at which the AGs start to execute in turns (assuming that no structural conflicts occur). Thus, the fitness function for HT mode is

$$F_{HT} = max_i \ time_i \ ;$$

- in LL mode, after a producer node has generated enough data, the consumer node starts executing and generates all its output without pausing. For each node i, the waiting percentage W_i can be calculated. Given T_m, the uninterrupted execution time of node m, and r, the ratio of the replication number of m over the replication number of its consumer node n, node n takes $T_m \times r \times (1 - W_n)$ to complete its calculations, after waiting for $T_m \times W_n$; the overall time for both nodes to complete their work is $T_m \times (W_n + r \times (1 - W_n))$. The estimated runtime of node i can therefore be derived as

$$time_i = T/R_0 \times \{W_0 + (E_0 \times f_0) \times [W_1 + (E_1 \times f_1) \times (W_2 + (E_2 \times f_2) \times (...))]\}$$

  For convenience, $f_x = \min(\frac{R_{p(x)}}{R_x}, 1)$, where R_x is the replication number of node x, and $p(x)$ is the index of the provider node for x. $E_x = 1 - W_x$ is the percentage of execution of node x. T is the execution time of the first node without extra replication. This is iterated over the whole node set, and the fitness function F_{LL} is the final estimated time.

The last stage of the PIMCOMP compiler is dataflow scheduling, that generates the final instruction sequence based on the selected pipeline mode.

Algorithm 2.1 HT Dataflow Scheduling Algorithm for PIMCOMP [72]

```
1: for each core do
2:    while have unfinished AG do
3:       load data from global memory
4:       for each unfinished AG do
5:          perform one MVM operation
6:       end for
7:       accumulate results across AGs within core
8:       accumulate results across AGs between cores
9:       apply activation function to the results
10:      store data to global memory
11:   end while
12: end for
13: allocate other operations to cores
```

- **HT dataflow:** the dataflow scheduling for HT mode is shown in Algoritm 2.1. Since the on-chip local memory cannot store all the data in an inference for each node, it is necessary to periodically move input and output data to and from the global memory after the final result of each operation has been accumulated from all the cores in which it has been partitioned (lines 3-10). To improve parallelism, other operations such as POOL, CONCAT, ELTWISE are distributed among several cores (Line 13).
- **LL Dataflow:** in LL mode, when a node computes an output, it is immediately passed to its consumer nodes, and when a consumer node receives enough data, it starts executing. The required input and expected output flows of each replicated block can be obtained analytically. To improve computational parallelism and reduce data movement between cores, non-convolution operations are divided to multiple cores according to the replication number of their predecessor convolutional layer.

To optimize the utilization of the limited on-chip local memory, and avoid expensive accesses to the global memory, PIMCOMP also implements a memory reuse mechanism rather than allocating a new memory block for each AG.

The simulation results for PIMCOMP showed a more even distribution of the computations with respect to PUMA, that tends to have longer execution times for some cores while others finish early; this translates in a slight increase in static energy for PIMCOMP, as more cores are active at the same time, but shorter overall runtime. More in general, the PIMCOMP compiling time measured is reasonable, with the weight replicating and core mapping phase impacting more in HT mode, and dataflow scheduling being more

relevant in LL mode.

RNC [47], designed by Loong et al., adopts the compilation steps introduced in PIM-COMP and combines them with the MNSIM 2.0 simulation framework [85] to develop a ReRAM based tool for Neural Architecture Search, a subfield of machine learning that automates the process of designing neural network structures by systematically exploring a predetermined search space. They introduce a technique to support depth-wise convolution, and approach weight replication as an optimization problem instead.

2.2. Computing in-DRAM and in-cache

Other than NVMs, Processing Using Memory can also be implemented on DRAM or SRAM memory banks; in the latter case it is known as in-cache computing. Similarly to NVM crossbars, the memory arrays are set up to perform logic and arithmetic operations; some examples are Compute Caches [6] and Neural Cache [23]. Like NVM crossbars, the reduction of data movement and the parallelism opportunities enable high efficiency. The difference in the underlying hardware, however, means that the offloading process, and especially the related optimizations, need to be approached in a different way.

2.2.1. Offloading techniques

Srivastava, Kang, et al. [68] build PROMISE (PROgrammable MIxed-Signal accElerator) on SRAM compute memory for accelerating ML applications. Their main goal, beyond programmability, is allowing software control over the energy-vs-accuracy trade-offs by mapping error tolerance specifications at the application level to low-level hardware parameters to minimize energy consumption. This optimization will be detailed in paragraph 2.2.2.

The compilation flow for PROMISE is based on LLVM, but develops its own IR. The IR works with an abstraction of the Tasks, the wide-word vector macro instruction of the ISA, aptly called AbstractTask, that is oblivious to hardware-specific parameters (e.g. length of the bit-cell array, size of the bit-cell array, etc.) and to the specific kind of vector operation described, which doesn't matter during the optimization phases but just during code generation. An AbstractTask has the following ten fields:

- (F1) W: address of a 2D data array;
- (F2) X: address of a 1D data array;
- (F3) output: address of the output 1D data array;

- (F4) vecOp: element-wise vector operation between a row of W and X;
- (F5) redOp: reduction operation on the output of vecOp;
- (F6) digitalOp: unary operation on the output of redOp;
- (F7) vectorLen: number of elements in X;
- (F8) loopIterations: number of iterations of the loop executed by the task;
- (F9) threshold: threshold value for Class-4 threshold operation of Task;
- (F10) swing: voltage swing at which the Task should run on PROMISE; initialized to maximum accuracy by the front-end, and fine-tuned during the energy optimization pass.

The PROMISE compiler IR is a directed acyclic graph where each node is an AbstractTask and a directed edge represents dataflow from the output of a node to an output of the other. Despite the iterative nature of the Tasks, the graph is acyclic because the loop count is embedded in the loopIterations field of the AbstractTask itself. If a loop contains a sequence of multiple tasks, it is always executed on the host processor and not on PROMISE.

The front-end of the compiler translates a Julia [12] program to the PROMISE compiler IR. Julia was chosen by the authors because in such a high-level language it's easier to identify offloadable computation patterns without the need for sophisticated compiler analysis; Julia is also commonly used for ML applications and libraries (e.g. MXNet, Flux), and already had a working open-source LLVM front-end at the time of writing. After using the Julia front-end to translate the application to LLVM IR, the PROMISE pass analyzes every LLVM function to identify computation patterns that are suitable to offloading. The LLVM IR representation is a collection of Static Single-Assignment (SSA) data-flow graphs, one graph per function in a program. Each node in an SSA graph corresponds to an LLVM IR instruction. Before pattern matching, to avoid missing patterns because of minor variations, the compiler converts all single basic block loops into canonical loops. The PROMISE pass looks for matrix-matrix operations with a reduction component and it verifies whether it is enclosed in a single basic block "natural loop", using the homonymous, widely used LLVM analysis. If it is, the loop and its induction variable are used to check if their SSA graph matches with the SSA graph pattern in fig. 2.9; this pattern captures many widely used ML inference kernels (e.g. template matching, support vector machines, k-nearest neighbor, matched filtering, matrix-vector multiplication, etc.).

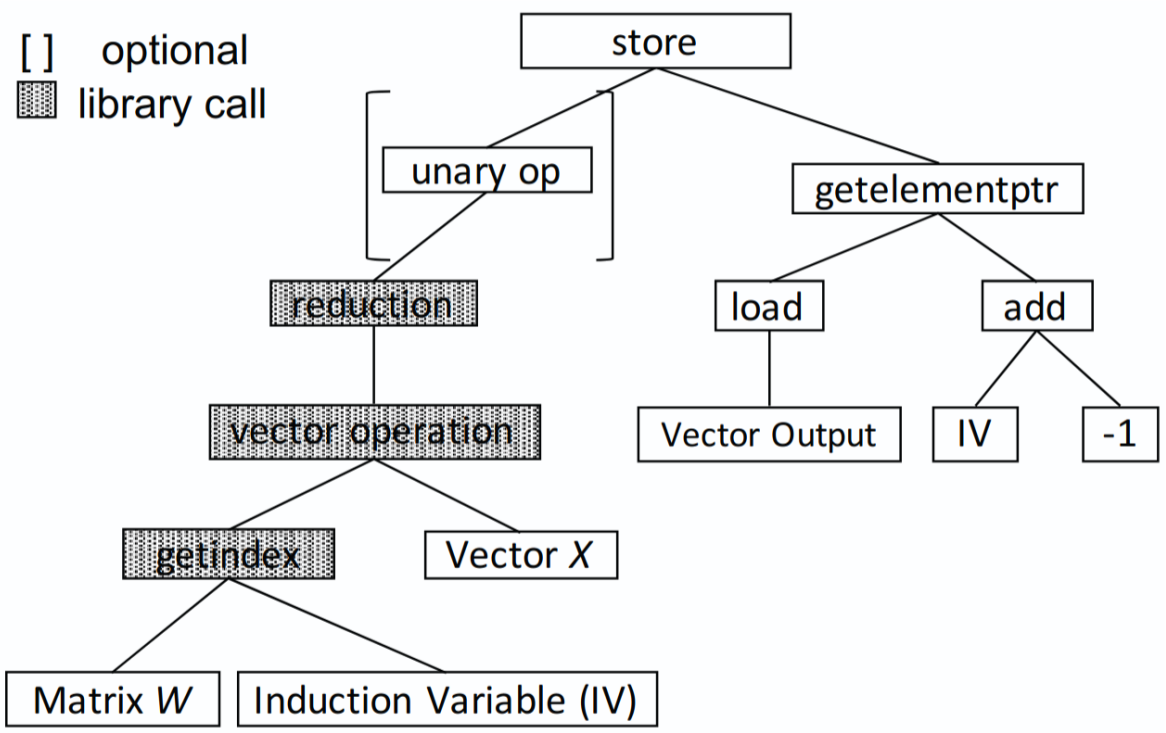

Figure 2.9: SSA Pattern for single basic block loops. The shaded nodes are calls to Julia library. The part enclosed in square brackets is optional for pattern matching [68].

If it's a match, the loop can be offloaded to PROMISE and is translated into an AbstractTask. The SSA nodes Matrix W, Vector X and Vector Output, as seen in fig. 2.9 map to the W, X, and Output fields of a AbstractTask, respectively; LoopIterations will be computed at run-time from the induction variable IV and the corresponding conditional branch at the end of the basic block; VectorLen can be obtained from X. Finally, the swing field is calculated as described in subsection 2.2.2. Once the AbstractTask is ready, the corresponding loop is replaced with a call to the PROMISE runtime, with the fields of the AbstractTask as parameters.

2.2.2. Code optimizations

As mentioned in the previous subsection, PROMISE [68] also implements an interesting approach to the minimization of energy consumption, enabling automatic mapping of high-level error tolerance specifications to hardware voltage swing parameters. To achieve energy savings, PROMISE allows the application programmer to express the upper bound on the additional error their model can tolerate, called mismatch probability p_m, and expressed as the maximum difference between the classification accuracy of the ML model,

p_{model}, and the classification accuracy of the algorithm running on PROMISE, $p_{PROMISE}$:

$$p_{model} - p_{PROMISE} \leq p_m \tag{2.1}$$

The compiler determines the swing field for each AbstractTask in the application that ensures that this error tolerance is still met.

This mapping is broken down in two steps: 1) determining the minimum bit precision required to achieve a given p_m, which is a hardware-independent algorithmic property, and 2) mapping the obtained bit precision to the right hardware swing voltage, which is dependent on the PROMISE architecture.

1. To determine the minimum bit precision that allows to achieve a given error tolerance p_m, the authors use the results from Sakr et al. [62]. This work analyzes the quantization (floating-point to fixed-point conversion) tolerance of Neural Networks and gives a relationship between the accuracy degradation and the bit precisions used to store the activations and weights of a neural network model. Given the bit precision of weights and activations (B_W and B_A), a bound can be computed on the mismatch probability $p_{fl} - p_{fp}$, where p_{fl} is the accuracy of the floating-point model, and p_{fp} is the accuracy of the same model quantized to fixed-precision representation for weights and activations. This bound can be expressed as:

$$p_m \leq \Delta_A^2 E_A + \Delta_W^2 E_W \tag{2.2}$$

   where E_A and E_W are statistics obtained while training the model, and $\Delta_A = 2^{-B_A-1}$, $\Delta_W = 2^{-B_W-1}$. Since in PROMISE a neural network inference can be rewrote as a sequence of AbstractTasks, it is possible to use Sakr's model to calculate the bit precision B_X for X in each AbstractTask given the mismatch probability p_m for the model and weight precision $B_W = 7$, since the PROMISE bit-cell array uses 8-bits to store a value, including one sign bit.

2. Once the necessary bit-precision B_W has been determined, it must be mapped to the right swing voltage. To achieve B_W-bit precision in the final output, the error introduced must be less than $1/2^{B_W+1}$. When lowering the swing voltage, the most relevant source of error in PROMISE comes from aREAD operations, i.e. analog reads on the compute SRAM. The output of aREAD follows a normal distribution $\hat{W} \sim \mathcal{N}(W, \sigma_W^2)$, where $\sigma_W = |W| \cdot f(swing)$, and $f(swing)$ is a function of the AbstractTask *swing* parameter ranging 0.08 ~ 0.75. The *swing* parameter and $f(swing)$ are inversely proportional, so σ is minimized with a higher *swing* param-

eter. After reading and aggregating N such vector elements, the standard deviation of the aggregated value σ_{agg} of the output is $\sigma_W/\sqrt{N}$. Since W is in the range [-1, 1], we assume $|W| = 1$ for all values to maximize σ_W and σ_{agg}. The authors choose a confidence level of 99%, which corresponds to $2.6 \times \sigma_{agg}$. To guarantee B_W-bit precision at the output of aggregation, this yields:

$$2.6\sigma_{agg} = 2.6\frac{f(swing)}{\sqrt{N}} < \frac{1}{2^{B_W+1}}$$

Putting it all together, the back propagation statistics, E_A and E_W of the trained application model, along with the desired p_m, are plugged into equation 2.2 to estimate B_X, as B_A. Then B_X and the vectorLength field are used as input to equation 2.1 to estimate the minimum swing voltage. Note that the first step is limited to Neural Networks workloads, as that is what the results from [62] refer to. However, it is still possible to optimize kernel from other application domains by brute-force sweeping through all eight possible swing voltage levels to look for the optimal value. The evaluation performed by the authors, comparing the case in which all tasks use maximum swing vs. the optimized swings estimated using $p_m = 1\%$, showed average energy savings of 15%.

Fujiki et al. (2019) [27] design yet a different kind of in-memory architecture. Duality Cache is an in-cache Single Instruction Multiple Threads (SIMT) computation architecture that targets general purpose data parallel applications. The cache is repurposed to function both as a regular cache and a computation unit that performs floating point arithmetic and transcendental functions, further enriching the set of available instructions proposed by prior works like [6] and [23], that limited themselves to logical and fixed-point arithmetic operations. To expose the applications parallelism to the hardware, they adopt CUDA [54] as a programming model and develop a compiler to translate CUDA programs to the Duality Cache VLIW ISA.

Duality Cache's compiler is built on top of GPU Ocelot dynamic compilation framework [20] and it works on CUDA source code; the compilation flow starts by compiling it using nvcc, NVIDIA's CUDA compiler. The output CUDA executable includes three kinds of object files (i.e. elf, PTX, and SASS). PTX is NVIDIA's low-level parallel thread execution virtual machine ISA; the back-end of the DC compiler optimizes the PTX IR, schedules instructions, allocates resources, and finally translates it into the VLIW-style Duality Cache ISA (DC-PTX), that has the form of an AST. DC-PTX is a subset of PTX: some GPU-specific instructions are eliminated; however several fields are added to PTX to include operand locations. The DC-PTX kernels will be loaded and executed by API calls to DC-Runtime library in a similar way as the CUDA runtime.

The DC compiler tackles the classic VLIW trade-off between minimizing register use and avoiding spilling (especially relevant in the DC architecture, that has a limited number of private registers), versus maximizing parallelism; resource allocation and instruction scheduling are performed at the same time combining the Bottom-Up Greedy (BUG) algorithm [24], for scheduling, and linear scan register allocation. Other optimizations implemented by the DC compiler are:

- *AST balancing:* the operands of a chain of associative binary operations are distributed evenly to available VLIW slots, to maximize ILP.
- *Thread independent variable isolation:* the register pressure is reduced by avoiding to store thread independent variables. For example, a fixed length loop is unrolled by Duality Cache runtime and the induction variable is provided as a constant if necessary. Thread independent variables are identified by conducting dependency analysis and instructions that only process thread independent variables are marked with the appropriate metadata.

As previously noted, the Duality Cache architecture can utilize memory arrays in Last Level Cache for both computing and caching. This particular feature is exploited through compiler analysis. The compiler evaluates the kernel dimension and shared memory usage to determine whether to allocate the cache for computing or for actual caching, so as to leverage the locality of the applications.

2.3. Hybrid architectures

CIM-MLC, developed by Qu, Zhao, et al. [59], is a universal multi-level compilation framework for general CIM architectures, whether they are NVM, DRAM, or SRAM-based. The goal is to provide a compilation tool that is not bound to a specific architecture, by establishing a hardware abstraction that can model diverse PUM accelerators in their different devices, architectures and programming interfaces. More importantly, this abstraction allows CIM-MLC to explore mapping and scheduling strategies across multiple architectural tiers, to achieve better optimization results than prior CIM-oriented compilation frameworks, that optimize on a single computational level.

The hardware abstraction that they propose works in three tiers, that reflect both the architecture hierarchy and the compute modes. The architecture layers are, from top to bottom, (a) chip, (b) core, and (c) crossbar. Each layer exposes to the compiler several parameters (e.g. number of cores or crossbars, Network-on-Chip (NoC) type, computing capacity, buffer size and bandwidth, type of storage cell, DAC/ADC precision),

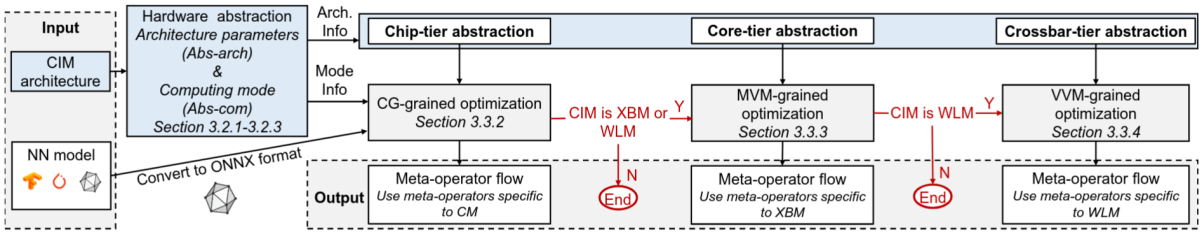

Figure 2.10: CIM-MLC workflow [59]

which abstract hardware details in order to adequately describe diverse architectures. The compute granularity is instead abstracted with (a) Core Mode (CM), (b) Crossbar Mode (XBM), and (c) Wordline Mode (WLM), in a one-to-one correspondence with the architecture tiers.

The CIM-MLC workflow is illustrated in fig. 2.10. The optimization of the DNN inference workloads, to obtain low latency and energy efficiency, is achieved through a multi-level scheduling strategy; each computing mode has its own optimization method. The compilation process starts with an ONNX description of a DNN model, then runs its sequence of optimizations in a top-down order, from coarse to fine granularity, up to the lowest level supported by the target architecture:

- Computational Graph Grained (CG-Grained): tailored to the CM. Its inputs are the ONNX model and chip-tier hardware parameters, focuses on operator duplication and pipeline strategies. The optimization information is recorded as attributes in the operator nodes in the ONNX graph.
- Matrix-Vector Multiplication Grained (MVM-Grained): tailored to the XBM. Its inputs are the results from the previous step, plus the chip and core-tiers hardware parameters. Focuses on finer operator duplication and pipeline optimization, and maps the operators to the crossbars.
- Vector-Vector Multiplication Grained (VVM-Grained): tailored to the WLM. Builds on the optimizations of the previous steps and performs optimization at the granularity of row scheduling.

The goals of Computational Graph Grained optimization stage is mapping DNN operators onto cores, and optimizing the use of hardware resources, thereby reducing latency and power consumption. It does two things: operator duplication under the constraint of the total number of cores to enhance computational throughput, and inter-operator pipelining to enhance execution efficiency. The authors adopt a resource-adaptive compute graph segmentation and intra-segment dynamic balancing pipelined duplication algorithm, that

outputs the subgraph set of the ONNX computing graph and the duplication results for CIM-supported operators.

The algorithm starts by initializing the resources and computation latency of each node of the computing graph. It uses dynamic programming to search for all operators' duplication numbers under the constraint of the number of cores. Then, to avoid the pipeline stall because of the imbalance between computing and data access of adjacent layers, it adjusts the duplication number for each node. Meanwhile, it will update the duplication number to also keep the data transfer amount within the NoC and global buffer capability. If a node is followed by a CIM-unsupported operator, the duplication number is updated considering also the ALU constraints. If the CIM resources are not able to hold a whole DNN, the algorithm iteratively constructs the maximal sub-graphs that can fit within CIM capacity. Each constructed computation sub-graph is updated by successively popping the last node from the subgraph and using dynamic programming to get the latency of the remaining computation subgraph. Once the latency no longer decreases, the construction of that subgraph is complete. The nodes that pop out will be used to construct the next maximal subgraphs until all nodes are part of a single subgraph and have duplication numbers assigned.

The optimization process then moves to the Matrix-Vector Multiplication Grained stage. What it does is unrolling the CIM-supported operators to matrix-vector multiplication, and then maps and schedules the MVMs to the crossbars. More specifically, it explores two techniques: the duplication of the operator in the crossbars and an MVM-grained computing pipeline to boost the computing throughput under the power limitation. The duplication number D^{O_i} of an operator O_i within a crossbar, determined by the CG-grained optimization, is updated using the following equation:

$$D'^{O_i} = \lfloor \frac{num_{core}^{O_i} * D^{O_i} * Core_{VXB}}{num_{VXB}^{O_i}} \rfloor,$$

where $Core_{VXB}$ is the number of virtual crossbars in each core, $num_{core}^{O_i}$ is the number of cores occupied by operator O_i, and $num_{VXB}^{O_i}$ is the number of virtual crossbars occupied by O_i. Usually, one operator demands multiple VXBs to store its weights and complete the calculation.

The MVM-grained computing pipeline strategically staggers the activation time of different crossbars to reduce peak power consumption. When mapping a matrix-vector multiplication to multiple crossbars, traditional scheduling usually waits until all crossbars have received their inputs before computing. The proposed pipeline strategy, however,

activates a crossbar as soon as it receives its input. This reduces the number of simultaneously activated crossbars, thus lowering peak power consumption.

The last step performs Vector-Vector Multiplication Grained optimizations. In the WLM, multiple partial rows within the crossbar can be activated at once, providing a more compact interface for vector-matrix multiplication compared to XBM, which translates into additional parallelism opportunities. To further improve computing throughput, they employ a remapping strategy that distributes the data contributing to the same computation to different crossbars.

They applied the CIM-MLC to the PUMA architecture, and it resulted in a 75% reduced peak power with respect to the schedule obtained with the PUMA compiler. They also compared their method with the compiler tool from PBC [32]. PBC supports compilation for CM and XB, but CIM-MLC employs a finer-grained scheduling approach; comparing the scheduling results on the same CIM architecture, they find that PBC reduces the number of computation cycles by 84% with respect to an unoptimized code, while CIM-MLC reaches 95%, achieving a 3.2x speedup over PBC.

2.4. Final considerations

PIM is still a brand-new field, and PIM compilers all the more so. Being it still immature, it is to be expected that it is approached first with familiar tools. Many of the compilers presented in this chapter are developed using the LLVM framework (PROMISE), and LLVM projects like MLIR (OCC) and Polly (TDO-CIM); tried and true methods like the polyhedral model (TDO-CIM, TC-CIM, PBC) and multi-level rewriting (OCC) have been adopted. The already-existing tools that have been leveraged have stood the test, but as the research field grows, new tools and frameworks will probably emerge.

The main limitation of the compiler ecosystem for CIM architectures that compute using memory is their specificity in terms of both application domain and target architecture. We have seen how the hardware solutions that enable Processing Using Memory are especially suitable for performing Machine Learning operations, and indeed most of the compilers that we have examined in this chapter target this specific application domain, even limiting themselves to particular sub-domains like Deep Neural Networks. Only TDO-CIM, IMP and Duality Cache develop systems, and therefore compilers, oriented to general purpose computing. Moreover, some of the compilers examined make very specific assumptions about the underlying target hardware, or tailor their optimizations to specific technology features that other systems do not share; for example, the weight replication strategies that work well for NVM-based systems, with their massive number of

high-density crossbars, cannot be adopted in resource-constrained settings like in-SRAM computing systems. CIM-MLC deals with this issue by proposing a general hardware abstraction that models a variety of diverse hardware, but there is still work to be done in this direction. Future work will have to deal with extending the scope of compilers to encompass diverse application domains, or even just less specialized fields of the ML domain, and to support a more heterogeneous hardware range.

A detail that none of the mentioned works confronts is the intrinsic imprecisions and errors that computing in the analog domain brings. Machine Learning applications are inherently error-tolerant, and we've seen how PROMISE leverages this error tolerance to achieve energy savings by fine-tuning the voltage swing. All the proposed solutions, however, lack support for compensating for these inaccuracies, for example with error-correcting codes, which could be necessary in certain application domains.

Finally, most of these compilers, with the exception of PRIME, IMP, and Duality Cache, fail to take into account the ability of the crossbars to switch between functioning as a compute unit and as a memory, missing out on mapping optimization possibilities. This is a feature that, moving forward, will need to be explored better.

3 | Processing Near Memory

The Processing Near Memory, or Near Data Computing, approach to PIM consists in placing computation units close to the memories, preferably connected with high-bandwidth links, so that the data movement overhead is reduced in terms of both latency and energy consumptions. This is generally implemented by exploiting the logic layer of 3D-stacked memories, or by integrating small processors next to the memory hierarchy.

3.1. 3D-stacked memories

Section 1.1 presented the general hardware architecture of 3D-stacked memories, consisting in layers of DRAM connected by high-bandwidth Through-Silicon Vias, with a logic layer at the bottom that can be used to implement PIM functionalities. It is a completely different architectural model with respect to the PUM systems presented in Chapter 2, so it comes with completely different challenges, not only when it comes to hardware-specific optimizations, but also for offloading.

The offloading problem in PNM architectures has many more dimensions than for PUM. First, since they are not just fixed-function accelerators, but full-on processors, it must be decided at what granularity the code is to be offloaded. A coarse granularity, like an entire application, means losing opportunities to execute on the main host the code portions that would actually benefit from more complex logic and caching mechanisms; a finer granularity, for example at the level of an instruction, allows better control, but also causes more overhead, potentially nullifying the benefits of offloading. Usually, a middle ground is chosen, and computation are offloaded at the granularity of a kernel, be it a code bock, a function, a loop, or a vector operation. Second, it must be determined what are the criteria that make a code portion suitable for offloading; and third, and perhaps most important, it must be evaluated if the offloading candidates that meet these criteria would actually benefit from the offloading process. Sometimes memory bandwidth considerations or runtime conditions make offloading not convenient. This is why most works prefer to detect offloading candidates at compile time and make the final decision during execution, and some, like [56] prefer to use runtime techniques altogether.

3.1.1. Offloading techniques

Some of the early compilation proposals, like Active Memory Cube [52], choose to ignore the offloading problems to focus on other optimizations, and leave it to the programmer to mark which portions of the code to offload with Open MP 4.0 parallel directives [2]. This approach is clearly not ideal, given the complexity of the problem, and requires both significant effort and in-depth knowledge of the hardware on the part of the programmer. Subsequent works have explored how to handle the issue transparently, with different approaches.

Hameeza et al. [7] develop the LLVM-based Processing-In-Memory cOmpiler (PRIMO), that targets a NDC architecture with large vector units; in particular they focus on the HMC. Their approach to instruction offloading is very straightforward: during the instruction selection stage, PRIMO's Instruction Offloader module uses the instruction vector width as a metric. Large size vector operations are performed using the memory vector units, and smaller size operations are performed in the host processor.

In [33] Hsieh et al. adopt a more elaborate strategy. They propose a Transparent Offloading and Mapping (TOM) mechanism for GPU workloads. The authors assume an architecture that connects a main processor to multiple 3D-stacked memories and offloads bandwidth-intensive computations to a GPU in each of the logic layers. TOM consists of two components: first a compiler identifies the code to offload using a cost-benefit analysis; second, a runtime software/hardware mechanism assists the offloading process by predicting which memory pages will be accessed by offloaded code, and placing them in the memory stack closest to the offloaded code.

The TOM compiler distinguishes between memory intensive blocks to execute on the 3D memory logic layer, and compute intensive blocks to execute on the main GPU; the behavior of these sections, however may change dynamically during execution. To account for this, the compiler selects offloading candidates by estimating which code blocks have maximum potential memory bandwidth savings; the actual offloading decision is then taken at runtime based on dynamic system condition (e.g. Streaming Multiprocessors and bandwidth utilization). At runtime is also when the issue of co-locating the offloaded computation and its data can also be tackled: placing data in the 3-D memory stack might cause potential performance degradation of the non-offloaded code due to the potential increase in memory stack contention. The best memory mapping is learned by observing the behavior of a small number of offloading candidates initial instances. There is a short learning phase during which everything executes on the main processor and the corresponding data is in CPU memory, after which the collected info is used to determine

what data to copy to the GPU memory.

To identify the offloading candidates, the compiler needs to determine whether the memory bandwidth savings from executing the load and stores of the offloaded block in the memory stack exceed the overhead of the offloading itself, i.e. transferring the execution context to memory and return the results to the main GPU. The changes in bandwidth consumption due to offloading a block on the transmit channel (TX, from the GPU to the memory stack) and the receive channel (RX, from the memory stack to the GPU) are calculated as:

$$BW_{TX} = REG_{TX} - (N_{LD} + 2 \cdot N_{ST})$$

$$BW_{RX} = REG_{RX} - (N_{LD} + 1/4 \cdot N_{ST})$$

where REG_{TX} and REG_{RX} are the number of registers transmitted to and received from the memory stacks (representing the offloading cost on the bandwidth), N_{LD} and N_{ST} are the number of load and stores in the block, assuming that each load transmits an address and receives the corresponding data, while each store transmits both an address and data and receives an acknowledgment that is $\frac{1}{4}$ of the size of addresses, data and registers. The underlying assumption is also that loads and stores are executed independently for each thread.

This model is however too simplistic, as the GPU will offload code block instances at a warp granularity rather than a single thread granularity. This means that loads and stores will be coalesced by the load-store unit and caches; moreover, the size of addresses and data for loads are different because data is fetched at a cache line granularity. To take all this into account, a more accurate estimate of the bandwidth changes would be

$$BW_{TX} = (REG_{TX} \cdot S_W) - (N_{LD} \cdot Coal_{LD} \cdot Miss_{LD} + N_{ST} \cdot (S_W + Coal_{ST}))$$

$$BW_{RX} = (REG_{RX} \cdot S_W) - (N_{LD} \cdot Coal_{LD} \cdot S_C \cdot Miss_{LD} + 1/4 \cdot N_{ST} \cdot Coal_{ST})$$

where S_W is the size of a warp, S_C is the cache line size to address size ratio, $Coal_{LD}$ and $Coal_{ST}$ are the average coalescing ratios for loads and stores, $Miss_{LD}$ is the cache miss rate for loads. These equations show the reason why the authors choose to implement this process at compile time: determining REG_{TX} and REG_{RX} at runtime would introduce significant hardware complexity, but is straightforward for the compiler, since it already performs a live-in/live-out registers analysis for register allocation and instruction scheduling. The values that are not known at compile time are the coalescing ratios and the cache miss rate, for which an estimate is needed. The authors conservatively assume

a coalescing ratio of 1 (perfectly coalesced memory instructions in a warp) and a cache miss rate of 50%, close to GPU miss rates reported in literature.

An instruction block is a potential offloading candidate if $BW_{TX} + BW_{RX}$ (i.e. the overall expected change in memory bandwidth) is negative. The compiler tags the opcode of each candidate with a 2-bit value that indicates whether offloading is expected to save RX bandwidth, TX bandwidth, or both. This tag is then used at runtime by the hardware to determine whether to actually offload the candidate or not. If a candidate block contains a loop, the offloading overhead stays constant, but the benefits depend on the number of executed iterations, that act as a multiplier for the loads and stores inside the loop. There are three possible cases: 1) the loop trip count can be determined statically: the compiler uses the number to evaluate the equations above; 2)the loop trip count will be known before entering the loop at runtime: the compiler marks the block as a conditional candidate, and provides the condition for which the offloading will be beneficial; the hardware will offload the candidate only if the condition holds true; 3) loop trip count depends on its execution: the compiler assumes it to be one, and marks the block only if offloading the loop body results beneficial.

There are further limitations that candidate blocks must satisfy:

- it must not contain on-chip shared memory accesses, since the compute units in memory stacks cannot access the main GPU's shared memory without going through the off-chip link;
- if the code involves divergent threads, they must converge at the end of offloaded execution, to avoid over-complicating the management of the control divergence stack;
- it must not contain memory barrier, synchronization or atomic instructions, as synchronization primitives between the main GPU and the logic layer SMs are not supported.

Cost-benefit analysis is also the offloading technique of choice for Hadidi et al. [30]. They develop CAIRO, a Compiler-Assisted technique and decision model for enabling InstRuction-level Offloading for PIM, building on their previous work GraphPIM [51], extending it from CPU to GPU workloads and elaborating a new compilation flow. Their compiler targets the Hybrid Memory Cube, and, unlike TOM, they choose to offload work at the granularity of a single instruction, rather than code blocks. This is because they identify the execution overhead of atomic instruction as the main bottleneck in graph computing applications, and they want to exploit the atomic instructions supported by the

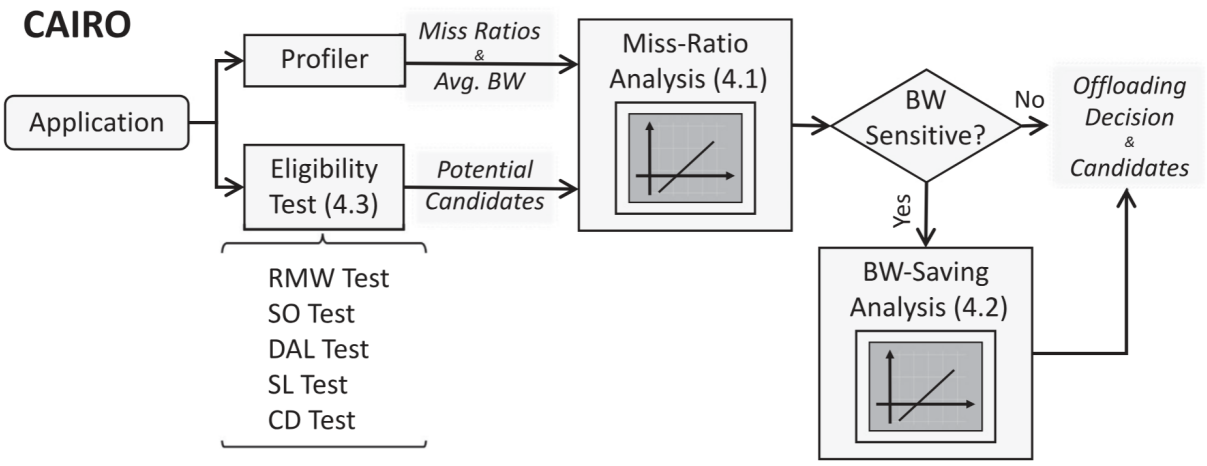

Figure 3.1: CAIRO workflow [30]

HMC 2.0, as they incur in lower overhead than the corresponding host atomic instructions. Moreover, by offloading a single atomic instruction to the HMC instead of the whole Read-Modify-Write instruction sequence that they replace, one can save memory bandwidth, which can help mitigate the inefficiency of the memory subsystem due to the irregular data access patterns exhibited by graph workloads.

The CAIRO workflow (fig. 3.1) starts by running a simple profiler that estimates the Last Level Cache (LLC) miss ratio of memory accesses and average bandwidth utilization of each candidate during the entire application runtime, as these quantities will be necessary in the cost-benefit estimation. Meanwhile, CAIRO also performs an eligibility test to identify offloading candidates. It scans for instructions that can be translated to HMC-atomic operations: first, groups of instructions from a single thread that perform RMW from/to a single location, and second, generic host atomic instructions. This is the only step in CAIRO that depends on the details of the target hardware, and it's implemented with simple compiler passes, which are:

1. Read-modify-write (RMW) test: checks that the outcome of converting an RMW operation on a single target is an HMC-atomic instruction, and that offloading the RMW would not violate sequential consistency;

2. Simple-operation (SO) test: checks that HMC supports the operation performed on a candidate;

3. Direct-address limitation (DAL) test: checks that load and store instructions use a direct memory address (i.e., embedded in the instruction code or accessible during execution);

4. Size-limitation (SL) test: checks that the data field of the HMC-atomic instructions is a single variable of 16 or 8 bytes, as per the specifications;
5. Candidate-density (CD) test: checks that the density of offloading candidates per memory region exceeds a certain amount, so that the performance impact of their offloading will not be negligible.

After performing the eligibility tests, CAIRO performs a "Miss-Ratio analysis" on each candidate, to evaluate the impact that bypassing the cache by executing on the HMC would have on performance. The speedup for offloading a host atomic instruction can be computed as a linear function of the LLC miss ratio MR of the candidate and the density ρ_{HA} of host atomic instructions in its memory region:

$$SU_{tot} = SU_{MR} + SU_{HA} = f(MR) + g(\rho_{HA}) = (C_1 \times MR) + (C_2 \times \rho_{HA}) + C_3$$

The coefficients C_1, C_2, and C_3 of the model are machine-dependent constants that can be obtained by a one-off execution of a micro-benchmark on the target architecture.

For applications that have been profiled as bandwidth-insensitive, mostly memory intensive applications like GPU workloads, performance is also affected by how much memory traffic can be saved by offloading, so before making a decision a "Bandwidth-saving Analysis" is also necessary. Bandwidth savings directly depend on the cache miss ratio. For a given candidate, there are three decision regions where its cache miss ratio can fall. If the miss ratio is lower than a certain threshold $MissRatio_L$, the instruction has high cache locality and should not be offloaded. If the miss ratio is higher than a threshold $MissRatio_H$, the performance benefits calculated in the previous analysis are guaranteed and the candidate is offloaded. If the miss ratio is between $MissRatio_L$ and $MissRatio_H$, the speedup obtained from bandwidth savings needs to be explicitly calculated and evaluated against the speedup value computed in the previous step. This computation also involves machine-dependent constants, obtained as explained previously. The values of the miss ratio thresholds can be customized at need; the authors conservatively use 30% and 80% for $MissRatio_L$ and $MissRatio_H$, respectively.

Using the same mechanism already deployed in GraphPIM, when a candidate is identified, CAIRO marks its offloading memory region. During execution, when an instruction accesses a marked region, the memory controller, instead of issuing regular memory instruction to the HMC, issues the corresponding HMC-atomic instruction.

3.1.2. Code optimizations

As seen in the previous section, both TOM and CAIRO, with their cost-benefit approach, give up on offloading when the data movement overhead overpowers the offloading benefits. Ghiasi et. al [28] try to increase the number of offloadable code segments by reducing this data movement overhead. Their target is a general multi-core system connected to a 3D-stacked memory equipped with cores on its logic layer, henceforth called NDP cores. The technique they propose, ALP, proactively transfer the data produced by a code segment to the segment that will consume it, immediately after it is ready; this way, the data movement latency is hidden by the operations in between the production and the consumption.

The first step of the ALP compiler is to identify and mark tightly-connected segments, i.e. the application segments that have significant data movement between them, enough to potentially cancel out the advantages of offloading one of them. The compiler detects pairs of tightly-connected segments by calculating their connectivity, defined as:

$$connectivity = max(\frac{inter\_segment\_data}{reg\_in1 + reg\_out1}, \frac{inter\_segment\_data}{reg\_in2 + reg\_out2})$$

where reg_in1 and reg_out1 are the number of live registers moving in and out of the first segment, reg_in2 and reg_out2 are the number of registers moving in and out of the second segment, $inter\_segment\_data$ is the number of the overlapping registers in reg_out1 and reg_in2 sets. This is all information already provided by the compiler's liveness analysis. If connectivity exceeds a set threshold, the two segments are marked as tightly-connected. The threshold is architecture-dependent; it can be calculated with a one-time offline profiling and it depends on the latency and bandwidth of the off-chip link between main memory and host system, the internal latency and bandwidth of main memory, the latency, bandwidth, and size of NDP and host caches, and NDP and host processor core features, such as their frequency and issue width. This search goes on iteratively until all the segments of the program are clustered with their tightly-connected segments. The compiler marks these clusters with two custom ALP ISA instructions, `CLSTR.BEGIN` and `CLSTR.END`, which also include a cluster identifier.

After detecting the tightly-connected segments, ALP determines what data will need to be proactively transferred between them. In most cases, the instructions that generate the inter-segment data are the same across different executions of the program for different input datasets. To identify them, the application is profiled using the Data Marshaling technique [71]. This process is performed before the register allocation, while the IR is

still in SSA form. The profiling algorithm is applied on each couple of tightly-connected segments identified in the previous step. For every memory access in the current segment, it checks if the instruction that wrote to this address is from the previous segment. In that case, the last writer instruction to this address from the previous segment is marked as a generator by adding a TRANSFER prefix to the instruction. For each write in the current segment, the algorithm collects the memory addresses, and the Program Counter in the current last writer list, in case they are generator instructions for the next segment. After the end of each segment, we empty the previous last writer list for the previous segment, and set the current segment's current last writer list to be previous segment's list previous last writer list.

The decision of which segments will actually be offloaded to the NDP cores is made during execution using runtime information as a metric, and with the support of additional ALP hardware components, such as an Offload Management Unit, Monitor Units, and Offload Table. When the program executes the instructions that have been marked as generators, if the two segments map to different cores, ALP proactively transfers the data from one core's cache to the cache of the core executing the next segment, as soon as the data is available.

We have already mentioned PRIMO [7] in the context of identifying offloading candidates; its compilation also tries to optimize register allocation in order to exploit the HMC features. HMC is divided into 32 vertical sections called vaults, to achieve higher bandwidth by exploring vault parallelism. Several designs take advantage of this organization by including a computing unit per vault in order to improve the parallelism of memory access and processing. Hence, in these cases 32 independent processors and their respective 32 independent register files exist. PRIMO implements a Vector Processor Unit (VPU) component, that operates during the register allocation stage and maps the computations to the most suitable vault. The proposed algorithm is generic for a PIM consisting of large Vector Processing Units.

The input of the algorithm is the number of the current Virtual Register (VR). The algorithm checks whether the accessed register is the first VR of the set. In the affirmative case the VR gets directly mapped to the available physical register belonging to the current vault, without any further checking. If the accessed VR is not the first register, the algorithm checks for dependencies between current and past VRs, by comparing the the end and start indexes of the current VR with end indexes of previous VRs through the live interval set. If a dependence is found, the current VR is mapped to a physical register belonging to the same vault as the previously dependent register. If there is no dependency, the vault number of the previous VR is incremented by 1 (mod 32); the

current VR is thus mapped to the next available physical register of a new vault, resulting in increased vault level parallelism. This process is repeated for each VR. The authors evaluate that the utilization increase that derives from this uniform distribution is up to 3.1x.

When designing the Active Memory Cube [52], Nair et al. also focus on exploiting the available parallelism to optimize hardware utilization and energy efficiency. The AMC is a design for a HMC logic layer optimized for a scientific exascale application scenario. The AMC processing elements (called lanes, each divided in four slices) are general-purpose cores, designed to be power and and area-efficient for the streaming vector functions common in scientific workloads. The AMC architecture implements a balanced mix of multiple forms of parallelism: heterogeneous processors, multithreading, instruction-level parallelism, vector, and spatial SIMD. Further, AMC processing lanes have radically different memory access properties (latency, bandwidth, coherence) than typical processor or accelerator cores. The complexity of the architecture motivates why marking the instructions to offload (using OpenMP 4.0 directives [2]) is left to the programmer, while the compiler, with the support of runtime OS additions, deals with balancing resource allocation and instruction scheduling on each lane, to better exploit the parallelism and ensure that the AMC-host communication doesn't cancel out the NMC benefits.

The AMC compiler supports C, C++, and FORTRAN language front-ends. The compiler analyzes loop nests in the target code and applies the necessary transformations (e.g. loop blocking, loop versioning, loop unrolling) to vectorize the code and exploit all forms of parallelism available, while trying to keep the resulting large instruction footprint in check.

The compiler backend maps instructions onto one of the four slices using a multilevel graph partitioning heuristic [35]. It then employs software pipelining to improve utilization of the AMC lane functional units. For scheduling, it employs a polynomial-time heuristic based on the Swing Modulo Scheduling algorithm [46]. Since, to minimize hardware complexity, the pipeline is not transparent to the compiler, the scheduling needs to explicitly take into account instruction latencies; it also tries to reduce the load/store issue rate and to schedule them as early as possible to hide their latency.

These compiler optimization techniques have the downside of hogging the 512 entries of the Lane Instruction Buffer (LIB), i.e. the instruction cache. To overcome this, the LIB is partitioned into multiple slots of the same size; the compiler then arranges the code in blocks that are smaller or equal than these slots, and uses a specialized ISA instruction (LoadLIB) to dynamically load the blocks according to the program control flow. The code is further optimized to limit the overhead of branching and of loading new sections

of code.

3.2. Near-memory architectures

With near-memory architectures we refer to systems where a simple processor is integrated next to the memory, the cache, or the memory controller, to reduce the distance that data has to travel when it is loaded and stored for computations. Since a 3D-memory stack is also a type of memory, these architectures share a lot of similarities with the ones detailed in section 3.1.

3.2.1. Offloading techniques

In [57] Pattnaik et al. set to develop NDC runtime techniques to minimize on-chip data transfer between the computing cores and Last-Level Cache (LLC) on a GPU architecture. Instead of sending computations to be performed on an off-chip system, like a 3D-stacked DRAM, they offload Load-Compute-Store instruction chains to a compute unit closer to where data resides on-chip, at the last level of the cache hierarchy. While the offloading process happens at runtime, because it needs to take into account data locality considerations, they rely on a compiler to identify offloading candidates. The computations will be sent as a "Compute Packet" to either the LLCs, or the ALU of a different GPU core if it results in lower on-chip data movement, employing the hardware mechanisms developed by the authors.

They choose to offload at the granularity of sets of instructions, looking for "offloading chains" that correspond to a high-level language statement. The instruction sequences considered for offloading are ones frequently used in applications such as machine learning kernels, linear algebra algorithms, cryptography, high performance computing applications and big-data analytics; furthermore, authors limit their examination only to patterns that can be packed into a single flit. The nine patterns chosen for offloading all start with a load instruction and end either with an ALU operation or a store operation. For applications with good L1 cache locality, offloading loads for computation without caching them locally would increase unnecessary data movement for these loads, so loads with further reuse at the core in the same thread wavefront are not considered for offloading. These offload chains can be found in a single compiler pass; multiple passes can allow the compiler to statically generate the data locality information [15] about the loads that will be used to make the final offloading decision. To reduce the building time of the Compute Packet at run time, the compiler also moves up the offset calculations of the load/store instructions in the offloading candidates, so that the offload chains' instructions are stored

contiguously. Finally, the compiler marks the opcodes of the offloadable instructions with a two-bit tag to indicate the first, intermediate and last instructions in the sequence.

Devic et al. [19] reason on the offloading problem in the context of a Bank-Level In-Memory general purpose Processor architecture (BLIMP), that supplies each memory bank with a general purpose RISC-V [82] processor, that will have considerably lower performance than the host, and can only access the data within their own bank. The authors also propose the addition of vector units to the RISC-V processors, terming them BLIMP-V, which add another option to the possible computing units where one can offload the code. They determine that offloadable vector regions are best suited for BLIMP-V execution, offloadable parallel regions are best suited for BLIMP execution, non-offloadable vector regions are best suited for the host AVX vector engine(s) [3], non-offloadable parallel regions are best suited for a multicore processor, and finally static regions are best suited for single threaded CPU execution. Using the LLVM framework, the authors adopt an approach to offloading to the BLIMP and BLIMP-V units consisting of four steps: identification, separation, compilation, and recombination.

The first step, which can be implemented either as a compiler pass or with manual identification through pragmas, identifies the code regions that are vectorizable, or at least parallelizable, and it verifies if they are suitable for being offloaded. First, the compiler checks for vectorizability: the code region must have no inter-loop data dependence, when an iteration sequence begins, all iterations must assume executability, and any branches present must be present in every iteration and maskable. If the code region is not vectorizable, it could still be parallelizable, which is checked by verifying if inter-loop element invariance holds. Vectorizable and parallelizable regions are further checked with an offloadability test. This happens in two steps.

1. First, the sum of (i) code size/span, (ii) input data size, (iii) working/scratchpad data size, and (iv) output data size must be able to be chunked into bank-sized "threads". These attributes cannot be determined statically, so they are estimated using conservative heuristics and/or manual pragmas; if the information available at compile time is insufficient, the check is postponed at runtime.

2. The second step to checking for offloadability is the usual comparison between the costs and the benefits of the offloading process. The system designed by the authors relies on the host cores to move input and output data to and from the bank. The cost in time for relaying r bytes of data out to the BLIMP units o_B and relaying data from the host CPU o_C to a single Double Data Rate DRAM bank on a 64-bit

host can be expressed numerically as:

$$o_B(r) = \frac{r}{8} \times (t_r + 8t_w + 64t_{ops})$$

$$o_C(r) = \frac{r}{8} \times (8t_r + t_w + 64t_{ops})$$

where t_r is the time for a fetched memory read, t_w is the time for a committed memory write, and t_{ops} is the time to execute a logic (AND, ADD, SHIFT) operation. This cost should be lower than the potential speedup. To make the final decision on offloadability, the compiler compares the estimated execution time on the BLIMP cores t_B vs the host core t_C, calculated as:

$$t_B(r, ops, n) = o_B(r_i) + \frac{ops_m \times AMAT_B + ops_s}{n \times s_B} + o_C(r_o)$$

$$t_C(r, ops, n) = \frac{ops_m \times AMAT_C + ops_s}{n \times s_C \times c(n)}$$

where r is the total size of the working memory region in bytes, ops is the number of operations or instructions per thread, n is the parallelization factor or thread count, r_i is the input data region size, r_o is the output data region size, ops_m is the number of memory operations, ops_s is the number of scalar operations, $AMAT_B$ and $AMAT_C$ are the average memory access time as seen by BLIMP and the CPU respectively, s_B and s_C are the speed of the BLIMP and CPU cores, and finally c is the contention factor when accessing shared resources or memory. If t_B is larger than t_C, the code is not offloaded. The authors experimentally measured that these static estimates for the offloading costs fall within 12.6% of the actual costs at runtime, and so are acceptable metrics for the offloading decision. Indeed, against the main trend, this work chooses to not postpone the actual offloading decision at runtime.

For the second step of the compiler flow, the code blocks marked as offloadable and their scopes are moved into another source file as function stubs, and substituted in the original code with an offload preamble (that deals with any necessary runtime checks and input data relayout), the offload syscall, and a postamble that retrieves the output data. In the third step, the stubbed function blocks are compiled into bank-level code with a RISC-V compiler, updating all address reference for the memory bank on which it will execute, while the rest of the code goes through the host compiler. Finally, as the fourth step, the resulting RISC-V ELFs are placed in the data segment of the final compiled application.

3.3. Hybrid architectures

Hybrid near-memory architectures combine compute units placed close to different levels of the memory hierarchy, for example near-bank units next to the main memory and near-LLC units next to the cache, or near-cache compute units and the logic layers of a 3D-stacked memory.

3.3.1. Offloading techniques

Maity et al. [48] base their work on a hybrid NMP computation framework, that comprises a traditional multi-core processor, NMP-enabled 3-D memories, and NMP-enabled LLCs. They develop CoaT, a Compiler-Assisted Two-Stage offloading approach for data-intensive applications.

The authors identify Loop-Stream regions (loop bodies consisting of stream operations) as the most suitable offloading candidates for a hybrid NMP system. To locate these regions, they implement a compiler pass based on LLVM 6.0. After lowering the source code into an IR, the pass identifies all the loop regions using LoopInfoWrapperPass [4], and searches the loop body for instructions accessing a stream data structure. If it finds it, the loop is marked with a function called `zsim_PIM_function_begin()` after its pre-header, and `zsim_PIM_function_end()` at its exit block(s). The actual offloading decision is made at runtime, with the support of an Offload Management Unit integrated on the CPU side, based on the region's overall execution time and, most importantly, the data locality offered by the LLC.

3.3.2. Code optimizations

Gu, Xie, et al. [29] develop iPIM, a programmable in-memory image processing accelerator, that uses the 3D-stacking near-bank architecture with a top-down hierarchy of cube, vault, process group, and process engine near-bank architecture, where compute-logic and lightweight buffers are integrated with a DRAM bank. iPIM provides an end-to-end compilation flow that exploits the front-end of Halide [60], chosen for its wide use as a image processing DSL, modified to include the customized schedules that the authors designed for iPIM, and a back-end to optimize code performance for the architecture features. First, the simple in-order core design imposes to spread virtual registers into several different physical registers to prevent data hazards due to register contention, rather that minimizing the number of allocated registers as it is customary. Second, because of the timing characteristics of the DRAM bank, the instruction reorder phase needs to optimize row

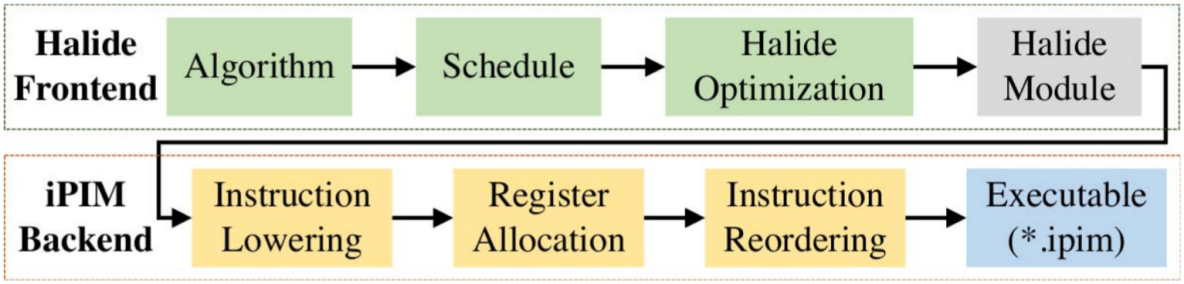

Figure 3.2: iPIM compilation flow [29]

buffer locality when exploiting ILP, which is tackled by adding new virtual dependencies.

The compilation flow is shown in fig. 3.2. The iPIM backend starts by lowering the Halide module into iPIM instructions. The resulting code then undergoes the following optimizations:

- **Register allocation:** the goal is to map each virtual register to a physical register. Usually, this is tackled trying to minimize the number of allocated registers; iPIM, however, uses a simple in-order control core to avoid hardware overheads, and does not support register renaming. Using the traditional method could translate in an increase of conflicts between the physical register, introducing more instruction dependencies, and, as a consequence, more pipeline stalls. To avoid such conflicts, the compiler uses the virtual registers' liveness analysis information to build a register interference graph. It then performs a depth-first search on the graph and tries to assign to each virtual register a different physical register than the most recently used one, essentially translating the allocation problem to a graph coloring problem.
- **Instruction reordering:** the goal of this step is to exploit ILP to the fullest. As mentioned before, having an in-order core means that dependencies between adjacent instructions lead to pipeline stalls. To minimize this issue, the compiler starts by building an instruction dependency graph. The instruction reordering algorithm traverses this (directed) graph in topological order, order that will correspond to the final schedule. Each node is marked with a timestamp T that estimates the earliest time at which it will be ready to be issued, initialized to zero. After a node is visited and inserted in the schedule, the incoming degree and the timestamp of its outgoing neighbors are updated. When there are multiple instructions available at a given time step, the scheduling will choose a load instruction with T smaller than the current time step, if present, otherwise it will pick the node with the smallest T. The time complexity is $O(|V|log|V| + |E|)$, where $|V|$ is the number of nodes,

i.e. instructions, and $|E|$ is the number of edges in the directed dependency graph.

- **Memory order enforcement:** in addition to data dependencies, the compiler also deals with pipeline stalls due to resource contention for DRAM between instructions. To avoid issuing too many memory instructions in a row, filling the memory queue, the compiler inserts dependencies between load and store instructions to distance them in the schedule. A third kind of dependency is also added to enforce the order of the memory accesses to DRAM, since DRAM access latency varies from the case of row buffer hit to row buffer miss, as to improve row buffer locality. After this, the dependency graph is passed to the instruction reordering stage explained earlier.

3.4. Final considerations

Processing Near Memory architectures, not unexpectedly, prove themselves to be somewhat more versatile than the PUM architectures discussed in Chapter 2. The examined works target workloads like graph processing, image processing, and many of them are general-purpose, even if still dedicated to applications that can benefit from the PIM model.

Like already noted for PUM architectures, compiler designs for near-memory architectures also leverage commonly used frameworks and tools, like LLVM or the Halide schedules. In this case as well, more specialized tools and framework are likely to be developed and implemented, especially considering the unique quality of the offloading problem, as detailed in section 3.1.

Future research might want to explore in the direction of more hybrid architectures, that combine computation units with different properties at different level of the memory hierarchy, to harness their different benefits. To support such architectures, compilers must devise more sophisticated offloading strategies, with cost models that take into account the diverse features of the hardware. Another aspect that compiler-based offloading strategies should integrate is accounting for the way in which the costs for offloading a code region might influence the costs for another region, as they are not independent; approaching the offloading decision in a holistic way will result in a more optimized code.

4 | Wrap-up

Table 4.1 outlines the characteristics of the compiler solutions for Processing Using Memory architectures examined in Chapter 2. As already noted in section 2.4, the fact that the PUM architectures are so well suited to Machine Learning workloads means that almost all of the proposals that implement them are targeting this application domain, with the exception of TDO-CIM, IMP, and Duality Cache that focus on general purpose computation.

The compilers perform primarily two kinds of tasks: identifying the operators to offload, mainly using pattern matching on a tree IR, and mapping the detected operators on the underlying hardware, with varying degrees of optimization. We can observe how most of the works identify and offload very similar operations. Despite this, the mapping, scheduling and optimization methods that they employ are often architecture or hardware specific, and not necessarily portable on other systems. As the field develops, it is likely that more compilation frameworks will move in the same direction as CIM-MLC and propose less specialized solutions that work on hardware abstractions to target a wider range of architectures.

Other possible functionalities that can be explored in compiler design have to do with leaning into the peculiarities of CIM itself: enabling a compiler controlled switch between memory mode and computation mode to improve mapping optimizations, and implementing error-correcting mechanisms to account for the inherent non-idealities of analog computing, or exploiting this loss of accuracy to obtain energy savings.

Table 4.2 summarizes the features of the compiler solutions for Near Data Computing discussed in Chapter 3. The range of application domains for which NDC architectures are developed is broader than that of PUM architectures: other than general purpose computing, we find GPU workloads, image processing, and vector-intensive applications.

Another difference we can remark on is how in the NDC context the offloading problem assumes a completely different dimension. Deciding what to offload and at what granularity requires more careful consideration, and the compiler techniques are more varied. The most common approach is setting a cost model to evaluate the potential benefit or

	Target HW	Application domain	Arch. dependent	Offloading granularity	E2E	Key features
TDO-CIM[73]	PCM	General purpose	N	MVM, MM	Y	Polyhedral based
TC-CIM[22]	ReRAM	ML	N	MVM, MM	Y	Polyhedral based
PBC[32]	ReRAM	NN	N	MVM, MM, CONV, pooling	N	Polyhedral based
OCC[65]	ReRAM	ML	N	MM	Y	Multilevel rewriting
HARMONY[45]	abstract NVM	DNN	N	MVM, MM, CONV, pooling	Y	HW IR
PRIME[18]	ReRAM	NN	Y	MVM	N	Reconfigurable ReRAM
IMP[26]	ReRAM	General purpose	Y	MVM, add, mult, sub	N	Complex operators lowering
HW-SW[8]	ReRAM	NN	N	MVM	Y	Graph partitioning
PUMA[10]	ReRAM	ML	Y	MVM	N	Specialized ISA
PANTHER[11]	ReRAM	NN Training	Y	MVM, MTVM OPA	N	NN training support
PIMCOMP[72]	SRAM	DNN	N	MVM	Y	Genetic algorithm
PROMISE[68]	SRAM	ML	Y	Task	Y	Energy-accuracy trade-off
DC[27]	SRAM	General purpose	Y	Arithmetic operations	N	Cache CU switch
CIM-MLC[59]	General abstraction	DNN	N	VVM, MVM DNN operators	N	HW abstraction

Table 4.1: Compilers for PUM architectures

	Target HW	Application domain	Offloading method	Offloading granularity	E2E	Key features
AMC[52]	HMC	Scientific exascale systems	Annotations	Code section	N	Multiple forms of parallelism
PRIMO[7]	HMC	Vector applications	Vector size	Vector instruction	N	VPU selector
TOM[33]	3D-Stack	GPU workloads	Cost-benefit analysis	Code block	N	Cost-benefit analysis
CAIRO[30]	HMC	GPU workloads	Cost-benefit analysis	HMC-atomic instruction	N	Miss Ratio analysis
ALP[28]	3-D Stack	General purpose	Runtime	Code block	N	Alleviating data movement overhead
Oppor-tunistic[57]	near-LLC	GPU workloads	Pattern matching	Instruction sequence	N	Earliest Meet Node
To PIM or not[19]	near-bank	General purpose	Cost-benefit analysis	Code region	N	BLIMP-V
CoaT[48]	3D-Stack + near-LLC	General purpose	Pattern matching	Loop	N	Loop-Stream region characteriz.
iPIM[29]	3D-Stack + near-bank	Image processing	-	Instruction	Y	Halide schedules

Table 4.2: Compilers for Near Data Computing architectures.

loss in performance of offloading a candidate; other compilation flows, similarly to the PUM offloading process, detect sequences of instructions or code blocks that match some offloading criteria.

Future compilation tools challenges might revolve around refining the offloading strategies, both to hone the already existing methods to produce more optimized code, and to follow the lead of possible new, more complex architecture, with several computation in units at different levels of the memory hierarchy.

Conclusions and future developments

In this thesis, we presented the emerging architectural paradigm of Processing in Memory, motivating its rise to relevance in the context of the proliferating of memory-intensive applications in an era when the memory wall has made memory accesses a major performance bottleneck. We went over the hardware technology advancements that have allowed the implementation of this paradigm, and commented over the necessity of specialized software support, especially compilation, to enable the widespread adoption of these architectures.

We discussed the compilation tools and frameworks that have been proposed to fill this gap, detailing the strategies that have been devised to face the novel challenges that the new paradigm raises. A variety of hardware categories allows to decline the core idea of PIM to completely different systems, from the integration of simple processing units in the same memory chip, to designing the memory themselves with the capability of performing computations. The analyzed compilers have embraced this diversity and developed specialized techniques to offload computations to PIM systems and optimize code performance and hardware utilization; despite this heterogeneity, however, more general approaches are still possible.

Finally, we took stock of the overall picture painted by the examined compiler ecosystem and reasoned over possible future research directions. PIM compilers will have to adapt to more complex architectures as they are designed, both by refining their optimization strategies and by shifting their focus on a broader hardware abstraction, rather than implementing specialized solutions; another potential source of improvement is playing more with the features of the PIM-enabling hardware technologies, for example by exploring their memory/computation unit duality, or designing error-correction mechanisms to deal with the imprecision of analog computations.

Bibliography

[1] The llvm compiler infrastructure. URL `https://llvm.org/`.

[2] Openmp arbopenmp application program interface version 4.0, 2013. URL `http://www.openmp.org/`.

[3] Intel® advanced vector extensions 512 (intel® avx-512) overview, 2021. URL `https://www.intel.in/content/www/in/en/architecture-and-technology/avx-512-overview.html`.

[4] llvm::loopinfowrapperpass class reference, 2026. URL `https://llvm.org/doxygen/classllvm_1_1LoopInfoWrapperPass.html`.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association. ISBN 9781931971331.

[6] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, 2017. doi: 10.1109/HPCA.2017.21.

[7] H. Ahmed, P. C. Santos, J. P. C. Lima, R. F. Moura, M. A. Z. Alves, A. C. S. Beck, and L. Carro. A compiler for automatic selection of suitable processing-in-memory instructions. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 564–569, 2019. doi: 10.23919/DATE.2019.8714956.

[8] J. Ambrosi, A. Ankit, R. Antunes, S. R. Chalamalasetti, S. Chatterjee, I. E. Hajj, G. Fachini, P. Faraboschi, M. Foltin, S. Huang, W.-M. Hwu, G. Knuppe, S. V. Lakshminarasimha, D. Milojicic, M. Parthasarathy, F. Ribeiro, L. Rosa, K. Roy, P. Silveira, and J. P. Strachan. Hardware-software co-design for an analog-digital accelerator for machine learning. In *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–13, 2018. doi: 10.1109/ICRC.2018.8638612.

[9] T. C. andMu Li andYutian Li andMin Lin andNaiyan Wang andMinjie Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL `http://arxiv.org/abs/1512.01274`.

[10] A. Ankit, K. Roy, D. Milojicic, I. Hajj, S. r. Chalamalasetti, G. Ndu, M. Foltin, S. Williams, P. Faraboschi, W.-m. Hwu, and J. P. Strachan. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. pages 715–731, 04 2019. doi: 10.1145/3297858.3304049.

[11] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, S. Agarwal, M. Marinella, M. Foltin, J. P. Strachan, D. Milojicic, W.-M. Hwu, and K. Roy. Panther: A programmable architecture for neural network training harnessing energy-efficient reram. *IEEE Transactions on Computers*, 69(8):1128–1142, 2020. doi: 10.1109/TC.2020.2998456.

[12] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014. URL `http://arxiv.org/abs/1411.1607`.

[13] L. Bougé. The data parallel programming model: A semantic perspective. In *The Data Parallel Programming Model*, 1996. URL `https://api.semanticscholar.org/CorpusID:10255262`.

[14] G. Burr, R. Shelby, C. di Nolfo, J. Jang, R. Shenoy, P. Narayanan, K. Virwani, E. Giacometti, B. Kurdi, and H. Hwang. Experimental demonstration and tolerancing of a large-scale neural network (165,000 synapses), using phase-change memory as the synaptic weight element. In *2014 IEEE International Electron Devices Meeting*, pages 29.5.1–29.5.4, 2014. doi: 10.1109/IEDM.2014.7047135.

[15] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, page 252–262, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916603. doi: 10.1145/195473.195557. URL `https://doi.org/10.1145/195473.195557`.

[16] L. Chelini, O. Zinenko, T. Grosser, and H. Corporaal. Declarative loop tactics for domain-specific optimization. *ACM Trans. Archit. Code Optim.*, 16(4), Dec. 2019. ISSN 1544-3566. doi: 10.1145/3372266. URL `https://doi.org/10.1145/3372266`.

[17] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. Tvm: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference*

on Operating Systems Design and Implementation, OSDI'18, page 579–594, USA, 2018. USENIX Association. ISBN 9781931971478.

[18] 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.

[19] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno. To pim or not for emerging general purpose processing in ddr memory systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 231–244, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3527431. URL `https://doi.org/10.1145/3470496.3527431`.

[20] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 353–364, 2010.

[21] A. Drebes. Teckyl: An mlir frontend for tensor operations., 2020. URL `https://github.com/andidr/teckyl`.

[22] A. Drebes, L. Chelini, O. Zinenko, A. Cohen, H. Corporaal, T. Grosser, K. Vadivel, and N. Vasilache. Tc-cim: Empowering tensor comprehensions for computing-in-memory. 2020. URL `http://impact.gforge.inria.fr/impact2020/`. 10th International Workshop on Polyhedral Compilation Techniques, IMPACT 2010 ; Conference date: 22-01-2020 Through 22-01-2020.

[23] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 383–396, 2018. doi: 10.1109/ISCA.2018.00040.

[24] J. R. Ellis. *Bulldog: a compiler for VLSI architectures*. MIT Press, Cambridge, MA, USA, 1986. ISBN 026205034X.

[25] L. Foundation. Onnx: Open neural network exchange format. URL `https://onnx.ai`.

[26] D. Fujiki, S. Mahlke, and R. Das. In-memory data parallel processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for*

Programming Languages and Operating Systems, ASPLOS '18, page 1–14, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450349116. doi: 10.1145/3173162.3173171. URL https://doi.org/10.1145/3173162.3173171.

[27] D. Fujiki, S. Mahlke, and R. Das. Duality cache for data parallel acceleration. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2019.

[28] N. M. Ghiasi, N. Vijaykumar, G. F. Oliveira, L. Orosa, I. Fernandez, M. Sadrosadati, K. Kanellopoulos, N. Hajinazar, J. G. Luna, and O. Mutlu. Alp: Alleviating cpu-memory data movement overheads in memory-centric systems. *IEEE Transactions on Emerging Topics in Computing*, 11(2):388–403, 2023. doi: 10.1109/TETC.2022.3226132.

[29] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie. ipim: Programmable in-memory image processing accelerator using near-bank architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817, 2020. doi: 10.1109/ISCA45697.2020.00071.

[30] R. Hadidi, L. Nai, H. Kim, and H. Kim. Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory. *ACM Trans. Archit. Code Optim.*, 14(4), Dec. 2017. ISSN 1544-3566. doi: 10.1145/3155287. URL https://doi.org/10.1145/3155287.

[31] J. Han, H. Liu, M. Wang, Z. Li, and Y. Zhang. Era-lstm: An efficient reram-based architecture for long short-term memory. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1328–1342, 2020. doi: 10.1109/TPDS.2019.2962806.

[32] J. Han, X. Fei, Z. Li, and Y. Zhang. Polyhedral-based compilation framework for in-memory neural network accelerators. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 18(1):1–23, 2021.

[33] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 204–216, 2016. doi: 10.1109/ISCA.2016.27.

[34] Y. Ji, Y. Zhang, X. Xie, S. Li, P. Wang, X. Hu, Y. Zhang, and Y. Xie. Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture, 01 2019.

[35] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 07 2006. doi: 10.1137/S1064827595287997.

[36] J. Kim and Y. Kim. Hbm: Memory solution for bandwidth-hungry processors. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–24, 2014. doi: 10.1109/HOTCHIPS.2014.7478812.

[37] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, 2013. doi: 10.1109/ISPASS.2013.6557176.

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

[39] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 02 2020.

[40] 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.

[41] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. 37(3):2–13, June 2009. ISSN 0163-5964. doi: 10.1145/1555815.1555758. URL `https://doi.org/10.1145/1555815.1555758`.

[42] C. Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22, 12 2012. doi: 10.1142/S0129626412500107.

[43] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, Mar. 2021. ISSN 2161-9883. doi: 10.1109/tpds.2020.3030548. URL `http://dx.doi.org/10.1109/TPDS.2020.3030548`.

[44] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. Drisa: A dram-based

reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301, 2017.

[45] X. Li, W. Yang, N. Jing, Q. Wang, Z. Mao, and W. Sheng. Harmony: A hardware-aware mapping and optimizing framework for computing-in-memory accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2025. doi: 10.1109/TCAD.2025.3641044.

[46] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*, pages 80–86, 1996. doi: 10.1109/PACT.1996.554030.

[47] K. C. Loong, S. Han, S. Liu, N. Lin, and Z. Wang. Rnc: Efficient rram-aware nas and compilation for dnns on resource-constrained edge devices. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, pages 154–161, 2024. doi: 10.1109/ICCD63220.2024.00032.

[48] S. Maity, M. Goel, and M. Ghose. Coat: Compiler-assisted two-stage offloading approach for data-intensive applications under nmp framework. *IEEE Transactions on Emerging Topics in Computing*, 13(3):753–767, 2025. doi: 10.1109/TETC.2024.3495218.

[49] M. J. Marinella, S. Agarwal, A. Hsia, I. Richter, R. Jacobs-Gedrim, J. Niroula, S. J. Plimpton, E. Ipek, and C. D. James. Multiscale co-design analysis of energy, latency, area, and accuracy of a reram analog neural training accelerator. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(1):86–101, 2018. doi: 10.1109/JETCAS.2018.2796379.

[50] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun. A modern primer on processing in memory. *CoRR*, abs/2012.03112, 2020. URL `https://arxiv.org/abs/2012.03112`.

[51] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 457–468, 2017. doi: 10.1109/HPCA.2017.54.

[52] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S.

Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17:1–17:14, 2015. doi: 10.1147/JRD.2015.2409732.

[53] P. Narayanan, A. Fumarola, L. L. Sanches, K. Hosokawa, S. C. Lewis, R. M. Shelby, and G. W. Burr. Toward on-chip acceleration of the backpropagation algorithm using nonvolatile memory. *IBM Journal of Research and Development*, 61(4/5):11:1–11:11, 2017. doi: 10.1147/JRD.2017.2716579.

[54] NVIDIA. Nvidia cuda., 2007. URL `https://developer.nvidia.com/cuda`.

[55] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[56] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 31–44, 2016. doi: 10.1145/2967938.2967940.

[57] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. Opportunistic computing in gpu architectures. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 210–223, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366694. doi: 10.1145/3307650.3322212. URL `https://doi.org/10.1145/3307650.3322212`.

[58] J. T. Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24, 2011. doi: 10.1109/HOTCHIPS.2011.7477494.

[59] 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. pages 185–200, 04 2024. doi: 10.1145/3620665.3640359.

[60] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462176. URL `https://doi.org/10.1145/2499370.2462176`.

[61] M. Research. Cntk: The microsoft cognitive toolkit. URL `https://cntk.ai/`.

[62] C. Sakr, Y. Kim, and N. Shanbhag. Analytical guarantees on numerical precision of deep neural networks. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3007–3016. PMLR, 06–11 Aug 2017.

[63] P. Sanders and C. Schulz. Think locally, act globally: Perfectly balanced graph partitioning. *CoRR*, abs/1210.0477, 2012. URL `http://arxiv.org/abs/1210.0477`.

[64] 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 *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016. doi: 10.1109/ISCA.2016.12.

[65] A. Siemieniuk, L. Chelini, A. A. Khan, J. Castrillon, A. Drebes, H. Corporaal, T. Grosser, and M. Kong. Occ: An automated end-to-end machine learning optimizing compiler for computing-in-memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1674–1686, 2022. doi: 10.1109/TCAD.2021.3101464.

[66] A. Somaini. A novel mlir-based compilation flow for accelerating convolutional neural networks on process in memory architectures. Master's thesis, Politecnico di Milano, 2024. URL `https://hdl.handle.net/10589/230106`.

[67] P. Springer and P. Bientinesi. Design of a high-performance gemm-like tensor-tensor multiplication. *CoRR*, abs/1607.00145, 2016. URL `http://arxiv.org/abs/1607.00145`.

[68] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag. Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56, 2018. doi: 10.1109/ISCA.2018.00015.

[69] H. S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, C-19 (1):73–78, 1970. doi: 10.1109/TC.1970.5008902.

[70] D. B. Strukov, G. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, 2008. URL `https://api.semanticscholar.org/CorpusID:4367148`.

[71] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 441–450, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300537. doi: 10.1145/1815961.1816020. URL https://doi.org/10.1145/1815961.1816020.

[72] 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.

[73] K. Vadivel, L. Chelini, A. BanaGozar, G. Singh, S. Corda, R. Jordans, and H. Corporaal. Tdo-cim: Transparent detection and offloading for computation in-memory. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1602–1605, 2020. doi: 10.23919/DATE48585.2020.9116464.

[74] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. 16(4), Oct. 2019. ISSN 1544-3566. doi: 10.1145/3355606. URL https://doi.org/10.1145/3355606.

[75] A. Vasudevan, A. Anderson, and D. Gregg. Parallel multi channel convolution using general matrix multiplication. *CoRR*, abs/1704.04428, 2017. URL http://arxiv.org/abs/1704.04428.

[76] S. Verdoolaege. isl: An integer set library for the polyhedral model. volume 6327, pages 299–302, 09 2010. ISBN 978-3-642-15581-9. doi: 10.1007/978-3-642-15582-6_49.

[77] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. 01 2012. doi: 10.13140/RG.2.1.4213.4562.

[78] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen. Schedule trees. 01 2014. doi: 10.13140/RG.2.1.4475.6001.

[79] A. F. Vincent, J. Larroque, N. Locatelli, N. Ben Romdhane, O. Bichler, C. Gamrat, W. S. Zhao, J.-O. Klein, S. Galdin-Retailleau, and D. Querlioz. Spin-transfer torque magnetic memory as a stochastic memristive synapse for neuromorphic systems. *IEEE Transactions on Biomedical Circuits and Systems*, 9(2):166–174, 2015. doi: 10.1109/TBCAS.2015.2414423.

[80] B. Vision and L. Center. Caffe2: Lightweight, modular, and scalable deep learning framework. URL `https://caffe2.ai/`.

[81] J. von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. doi: 10.1109/85.238389.

[82] VV.AA. Risc-v, 2010. URL `https://riscv.org/`.

[83] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, 2012. doi: 10.1109/JPROC.2012.2190369.

[84] D. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, L. Xu, and M. Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. 06 2014. ISBN 978-1-4503-2749-7. doi: 10.1145/2600212.2600213.

[85] Z. Zhu, H. Sun, T. Xie, Y. Zhu, G. Dai, L. Xia, D. Niu, X. Chen, X. S. Hu, Y. Cao, Y. Xie, H. Yang, and Y. Wang. Mnsim 2.0: A behavior-level modeling tool for processing-in-memory architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(11):4112–4125, 2023. doi: 10.1109/TCAD.2023.3251696.

[86] O. Zinenko, L. Chelini, and T. Grosser. Declarative transformations in the polyhedral model. 2018. URL `https://api.semanticscholar.org/CorpusID:86493041`.

Acronyms

ADC: Analog to Digital Converter

AG: Array Group

ALU: Arithmetic Logic Unit

API: Application Programming Interface

AST: Abstract Syntax Tree

AVX: Advanced Vector Extension

BLIMP: Bank-Level In-Memory general purpose Processor

BUG: Bottom-Up-Greedy

CG: Computational Graph

CIM: Computing in Memory

CM: Core Mode

CPU: Central Processing Unit

CUDA: Compute Unified Device Architecture

DAC: Digital to Analog Converter

DFG: Data-Flow Graph

DLP: Data-Level Parallelism

DNN: Deep Neural Network

DRAM: Dynamic RAM

DSL: Domain-Specific Language

E2E: End-to-End

ELF: Executable Linkable Format

FF: Full-Function subarray

GEMM: GEneral Matrix Multiplication

GPU: Graphic Processing Unit

HBM: High-Bandwidth Memory

HMC: Hybrid Memory Cube

HT: High-Throughput

IB: Instruction Block

ILP: Instruction-Level Parallelism

IR: Intermediate Representation

ISA: Instruction Set Architecture

isl: integer set library

IV: Induction Variable

LIB: Lane Instruction Buffer

LL: Low-Latency

LLC: Last-Level Cache

LUT: LookUp Table

ML: Machine Learning

MLIR: Multi-Level Intermediate Representation

MM: Matrix Multiplication

MRAM: Magnetic RAM

MVM: Matrix-Vector Multiplication

MVMU: Matrix-Vector Multiplication Unit

NDC: Near Data Computing

NDP: Near Data Processing

NN: Neural Network

NoC: Network on Chip

NVM: Non-Volatile Memory

ONNX: Open Neural Network eXchange

PCM: Phase-Change Memory

PE: Processing Element

PIM: Processing in Memory

PNM: Processing Near Memory

PTX: Plain TeXt

PUM: Processing Using Memory

RAM: Random-Access Memory

ReRAM: Resistive RAM

RMW: Read-Modify-Write

RX: Receive Channel

SASS: Streaming ASSembler

SCF: Structured Control Flow

SIMD: Single Instruction Multiple Data

SIMT: Single Instruction Multiple Threads

SRAM: Static RAM

SSA: Static Single-Assignment

TC: Tensor Comprehensions

TSV: Through-Silicon Vias

TTGT: Transpose Transpose GEMM Transpose

TX: Transmit Channel

VLIW: Very Long Instruction Word

VPU: Vector Processor Unit

VR: Virtual Register

VVM: Vector-Vector Multiplication

VXB: Virtual Crossbar

WLM: WordLine Mode

XBM: Crossbar Mode

List of Figures

List of Tables